O'REILLY®

JAVA
ОПТИМИЗАЦИЯ ПРОГРАММ

ПРАКТИЧЕСКИЕ МЕТОДЫ ПОВЫШЕНИЯ
ПРОИЗВОДИТЕЛЬНОСТИ ПРИЛОЖЕНИЙ В JVM

Бенджамин Эванс, Джеймс Гоф и Крис Ньюланд

Java: оптимизация программ

Практические методы повышения производительности приложений в JVM

Optimizing Java

Practical Techniques for Improving JVM Application Performance

Benjamin J. Evans, James Gough, and Chris Newland

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Java: оптимизация программ

Практические методы повышения производительности приложений в JVM

Бенджамин Эванс, Джеймс Гоф и Крис Ньюланд

Москва · Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Э14
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Карася*
Рецензент канд. физ.-мат. наук *Д. Е. Намиот*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Эванс, Бенджамин Дж., **Гоф**, Джеймс, **Ньюланд**, Крис.

Э14 Java: оптимизация программ. Практические методы повышения производительности приложений в JVM. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 448 с. : ил. — Парал. тит. англ.

ISBN 978-5-907114-84-5 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Бенджамин Дж. Эванс, Джеймс Гоф, Крис Ньюланд

Java: оптимизация программ. Практические методы повышения производительности приложений в JVM

Подписано в печать 31.01.2019. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 36,12. Уч.-изд. л. 24,0.
Тираж 300 экз. Заказ № 1203.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907114-84-5 (рус.)
ISBN 978-1-492-02579-5 (англ.)

Оглавление

Содержание

Эта книга посвящена моей жене, Анне, которая не только красиво ее проиллюстрировала, но и частично помогла ее отредактировать; а самое главное — она часто была первым человеком, с которым я обсуждал свои идеи.

— Бенджамин Эванс

Эта книга посвящена моей семье — Меган, Эмили и Анне. Эта книга не стала бы реальностью без их помощи и поддержки. Я хотел бы также поблагодарить моих родителей, Хизер и Пола, которые всегда поддерживали меня и всегда поощряли мою тягу к знаниям.

Я хотел бы также выразить признательность за руководство и дружбу Бенджамину Эвансу — было очень приятно вновь поработать вместе с ним.

— Джеймс Гоф

Эта книга посвящена моей жене, Рине, которая поддерживала меня и поощряла мои усилия, и моим сыновьям, Джошуа и Хьюго (надеюсь, они вырастут умными и любознательными).

— Крис Ньюланд

Об авторах

Бен Эванс является соучредителем и главным инженером в jClarity — стартапе, который обеспечивает инструменты производительности, призванные помочь командам разработчиков и тем, кто сопровождает программные системы. Он помогает организовать Лондонское сообщество программистов на Java и представляет его в Исполнительном комитете сообщества Java (Java Community Process Executive Committee), где работает над определением новых стандартов экосистемы Java. Бен является Java Champion и JavaOne Rockstar, соавтором книги *The Well-Grounded Java Developer* и регулярно выступает с докладами о производительности, параллелизме и смежных вопросах на платформе Java.

Джеймс (Джим) Гоф — Java-разработчик и автор книг. Джим впервые стал интересоваться Java во время учебы в Университете Варвика, по окончании которого стал членом Лондонского сообщества программистов на Java. Сообщество остается главным потребителем работ Джима, которые включают в себя проектирование и тестирование JSR-310 и работу в Исполнительном комитете сообщества Java на протяжении нескольких лет. Джим регулярно выступает с докладами на конференциях, а в настоящее время работает над бизнес-приложениями в Morgan Stanley. Он четыре года изучал Java и C++ в различных странах мира.

Крис Ньюланд является старшим разработчиком в ADVFN, где использует Java для обработки данных фондового рынка в режиме реального времени. Он является автором проекта JITWatch — анализатора журнальных файлов с открытым исходным кодом для визуализации и проверки решений, принимаемых при JIT-компиляции в Hot-Spot JVM. Крис является Java Champion и на ряде конференций представлял методы JIT-компиляции.

Об изображении на обложке

Животное на обложке книги — винторогий козел, или мархур (*Capra falconeri*). Этот вид дикого козла отличают его борода и скрученные наподобие штопора рога. Найденные в горных районах Западной и Центральной Азии, эти козлы обитают в высокогорных муссонных лесах на высотах от 600 до 3600 метров.

Мархуры — травоядные животные, которые, в первую очередь, питаются разнообразной растительностью, включая травы, листья, фрукты и цветы. Как и другие дикие козлы, мархуры играют важную роль в экосистеме, поедая листья низкорослых деревьев и кустарников и распространяя с навозом их семена.

Во время брачного сезона, приходящегося на зимний период, самцы соперничают между собой, пытаясь зацепить один другого рогами и вывести из равновесия. Последующее рождение молодняка приходится на конец апреля — начало июня; самка приносит одного-двух козлят. Взрослые самцы живут в одиночку и предпочитают жизнь в лесу, в то время как самки и молодняк живут стадами выше, на скалистых грядах.

Многие из животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они имеют важное значение для мира. Чтобы узнать больше о том, как вы можете им помочь, перейдите на сайт по адресу `animals.oreilly.com`.

Предисловие

Как вы определите *производительность*?

Большинство разработчиков, когда их спрашивают о производительности их приложений, полагают, что имеется в виду определенная мера скорости — что-то вроде количества транзакций в секунду, обрабатываемых гигабайтов данных... словом, выполнение как можно большего количества работы за минимально возможное время. Если вы — прикладной архитектор, то можете измерять производительность с помощью разнообразных метрик. Вас может в большей степени интересовать использование ресурсов, чем простое выполнение вычислений. Вы можете уделять больше внимания производительности соединений между службами, чем самим службам. Если вы создаете бизнес-решения для своей компании, то производительность приложений, вероятно, будет измеряться не в затраченном времени, а в долларах.

Но независимо от того, какую роль вы играете в компании, важны все упомянутые метрики (как и многие другие).

Я начал разрабатывать приложения на языке программирования Java в 1996 году. Тогда я только что сменил мою первую работу — написание CGI-сценариев AppleScript для поддержки приложений на языке программирования Perl на стороне сервера для команды веб-разработчиков бизнес-школы Университета штата Миннесота. Тогда язык программирования Java был только появившейся новинкой — его первая стабильная версия, 1.0.2, была выпущен ранее в том же году, и мне было поручено найти для него полезное применение.

В те дни наилучшим способом добиться повышения производительности приложения на Java было переписать его на другом языке. Так было до того, как язык Java получил возможность своевременной компиляции Just-in-Time (JIT), параллельную сборку мусора, и задолго до того, как технологии Java стали доминирующими на стороне сервера. Тем не менее многие из нас хотели использовать Java, и мы разрабатывали разные хитрые трюки, чтобы заставить хорошо работать наш код. Мы писали гигантские методы, чтобы избежать накладных расходов по диспетчеризации методов. Мы объединяли и повторно использовали объекты, потому что сборка мусора была медленной и зачастую разрушительной. Мы использовали множество глобальных состояний и статических методов. Мы писали действительно ужасный Java-код, но он работал... некоторое время.

В 1999 году ситуация начала меняться.

После того как мы годами пытались использовать Java не только для приложений, в которых не важна скорость, технологии JIT-компиляции начали приходить и к нам. Появление компиляторов, которые могли встраивать методы, сделало количество вызовов методов менее важным, чем разбиение гигантских монолитов на более мелкие части. Мы смогли перейти к реальному объектно-ориентированному проектированию, разделить наши методы на небольшие части и работать с интерфейсами. К нашему удивлению, каждый новый выпуск Java позволял добиваться все лучших и лучших результатов. Мы могли писать хороший Java-код, который нравился JIT-компиляторам. Java дорос до других серверных технологий, позволив нам создавать более крупные и сложные приложения с более богатыми абстракциями.

В то же время быстро улучшались и сборщики мусора. Теперь накладные расходы на создание пулов ресурсов стали превосходить стоимость их выделения. Многие сборщики мусора стали многопоточными операциями, и мы наконец получили сборщики мусора, которые работали быстро и без задержек, — так что стандартной практикой все чаше становилось беззаботное создание и выбрасывание объектов, которое с развитием достаточно умных сборщиков мусора в конечном итоге обещало перестать мешать работе наших приложений. И это работало... некоторое время.

Проблема любой технологии в том, что она всегда сводит на нет собственные достижения. По мере совершенствования технологий JIT и сборки мусора стало сложно ориентироваться при поиске путей повышения производительности приложений. Несмотря на то, что JVM может оптимизировать наш код и сделать объекты почти бесплатными, требования приложений и пользователей продолжают расти.

Часть времени (может быть, даже бо́льшую часть времени) превалируют "хорошие" шаблоны кодирования: малые методы корректно встраиваются, интерфейсы и проверки типов становятся недорогими, машинный код, создаваемый JIT-компилятором, оказывается компактным и эффективным. Но в другие моменты нам приходится вручную оптимизировать наш код, приспосабливать абстракции и архитектуру к ограничениям компилятора и процессора. Часть времени объекты действительно практически бесплатны, и мы можем игнорировать существование пропускной способности памяти и потребление процессорного времени сборщиком мусора. В другие моменты времени нам приходится иметь дело с терабайтными (или даже бо́льшими) наборами данных, которые ставят в тупик даже самые лучшие сборщики мусора и подсистемы памяти.

Ответ на вопрос о производительности в наши дни состоит в знании инструментария. Часто это означает знание не только того, как работает язык программирования Java, но и как работают и взаимодействуют библиотеки, память, компилятор, сборщик мусора и аппаратное обеспечение при выполнении ваших приложений на JVM. Работая над проектом JRuby, я узнал непреложную истину о JVM: не существует единственного решения всех проблем с производительностью, но для каждой проблемы производительности есть решение. Основная проблема — найти эти решения

и выбрать те из них, которые лучше всего отвечают вашим потребностям. Но теперь у вас есть секретное оружие в боях за производительность: это книга, которую вы собираетесь читать.

Переверните страницу, друзья, и откройте доступное для вас богатство инструментов и методов. Узнайте, как сбалансировать дизайн приложения с имеющимися ресурсами. Узнайте, как управлять виртуальной машиной Java и настраивать ее. Узнайте, как использовать последние разработки в Java, которые более эффективны, чем старые библиотеки и шаблоны. Узнайте, как заставить Java летать!

Сегодня — захватывающее время для Java-разработчиков, и никогда еще у них не было так много возможностей для создания эффективных и гибких приложений на платформе Java. Итак, приступим.

— Чарли Наттер (Charlie Nutter)
Главный инженер Red Hat Middleware

Введение

Соглашения, принятые в этой книге

В этой книге используются следующие типографские соглашения о шрифтах и оформлении.

Курсив
: Обозначает новые термины и важные понятия.

`Моноширинный шрифт`
: Используется для листингов программ, а также для элементов программ, таких как ключевые слова, имена переменных и функций, базы данных, типы и др. Используется также для консольных команд и их вывода, для имен файлов, каталогов и URL.

`<Моноширинный шрифт>` *в угловых скобках*
: Указывает текст, который должен быть заменен значениями, вводимыми пользователем или определяемыми из контекста.

Совет или предложение.

Общее замечание.

Предупреждение или предостережение.

Использование примеров кода

Сопутствующие материалы (примеры кода, упражнения и т.п.) доступны по адресу `http://bit.ly/optimizing-java-1e-code-examples`.

Эта книга призвана помочь вам выполнить вашу работу. В общем случае вы можете использовать примеры кода из нее в своих программах и документации. Вам не нужно связываться с издательством для получения разрешения, если только вы не воспроизводите значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, разрешение получать не нужно. Однако для продажи или распространения на CD-ROM примеров из книг издательства O'Reilly необходимо отдельное разрешение. Ссылаться на книгу или пример кода в ответах на вопросы можно и без разрешения. Но для включения значительного объема кода из этой книги в документацию вашего продукта следует получить разрешение.

Мы не требуем точного указания источника при использовании примеров кода, но были бы признательны за него. Обычно достаточно названия книги, фамилии автора, названия издательства и ISBN.

Если вы полагаете, что использование примеров кода выходит за рамки описанных разрешений, не постесняйтесь связаться с нами по адресу `permissions@oreilly.com`.

Благодарности

Авторы хотели бы поблагодарить многих людей за неоценимую помощь. За написание предисловия:

- Чарли Наттер (Charlie Nutter)

За предоставление высококвалифицированной технической помощи, включая информацию и знания, недоступные где-либо еще:

- Кристина Флуд (Christine Flood)
- Крис Ситон (Chris Seaton)
- Кирк Пеппердайн (Kirk Pepperdine)
- Саймон Риттер (Simon Ritter)
- Моника Бэевиз (Monica Beckwith)
- Дэвид Джонс (David Jones)
- Ричард Варбартон (Richard Warburton)
- Стивен Коннолли (Stephen Connolly)
- Ярослав Тулач (Jaroslav Tulach)

За общую помощь, поддержку и советы:

- Джордж Болл (George Ball)
- Стив Пул (Steve Poole)
- Ричард Поллок (Richard Pollock)
- Эндрю Бинсток (Andrew Binstock)

Наши технические рецензенты:

- Майкл Хсу (Michael Hsu)
- Дмитрий Вязеленко (Dmitry Vyazelenko)
- Джулиан Темпльман (Julian Templeman)
- Алекс Блевитт (Alex Blewitt)

Команда из O'Reilly:

- Вирджиния Вильсон (Virginia Wilson)
- Сьюзен Конант (Susan Conant)
- Коллин Коль (Colleen Cole)
- Рейчел Монахан (Rachel Monaghan)
- Нан Барбер (Nan Barber)
- Брайан Фостер (Brian Foster)
- Линдси Вентимиглиа (Lindsay Ventimiglia)
- Морин Спенсер (Maureen Spencer)
- Хизер Шерер (Heather Scherer)

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

ГЛАВА 1

Оптимизация и производительность

Оптимизация производительности Java (или любого другого кода) часто выглядит черной магией. Существует мистическое представление об анализе производительности — он обычно рассматривается как ремесло, практикуемое хакерами-отшельниками, погруженными в размышления и не видящими ничего, кроме экранов своих компьютеров (один из любимых мифов Голливуда о компьютерах и людях, которые с ними работают), видящими систему насквозь и с помощью неведомой обычным людям магии способными заставить ее работать быстрее.

Этот образ часто сочетается с грустной (но, увы, слишком распространенной) ситуацией, когда производительность среди приоритетов команды разработчиков уходит на второй план. В результате анализ выполняется уже после того, как система оказывается в беде и нуждается в "супермене", который ее спасет.

Истина же заключается в том, что анализ производительности является смесью приземленного эмпиризма с эмпиреями человеческой психологии. Одновременно важными факторами являются как количество наблюдаемых метрик, так и то, что конечные пользователи и заинтересованные лица *думают* о них. Разрешению этого кажущегося парадокса и посвящена остальная часть этой книги.

Производительность Java — ошибочный подход

На протяжении многих лет одним из трех наилучших результатов поиска в Google для *Java performance tuning* ("Настройки производительности Java") была статья 1997 года, которая очень рано попала в базу данных Google. Эта страница предположительно оставалась одним из топ-результатов поиска потому, что ее первоначальный рейтинг обеспечивал активное ее посещение, создавая тем самым петлю обратной связи.

На этой странице были приведены совершенно устаревшие советы, во многих случаях наносившие ущерб создаваемым приложениям. Однако ее расположение

в верхней части результатов поиска приводило к тому, что очень многие разработчики использовали эти ужасные советы в своей практике.

Например, в очень ранних версиях Java диспетчеризация методов имела ужасную производительность. В качестве временного решения некоторые разработчики на языке программирования Java выступали за написание огромных монолитных методов, избегая написания небольших методов. Конечно, со временем производительность виртуальной диспетчеризации значительно улучшилась. Помимо этого, современные виртуальные машины Java (JVM) и их возможности автоматически управляемого встраивания вызовов привели к тому, что виртуальная диспетчеризация в большинстве мест вызова была полностью устранена. Код, который следует совету "объединять все в один метод", теперь оказывается имеющим существенный недостаток, который препятствует его нормальной производительной работе с современными JIT-компиляторами.

Увы, нет никакого способа узнать, сколько неприятностей и ухудшений кода было сделано из-за плохих советов, но этот случай демонстрирует опасность отказа от количественных, поддающихся проверке подходов к повышению производительности. Он также служит прекрасным примером того, что нельзя верить всему, что вы читаете в Интернете.

Скорость выполнения кода Java является очень динамичным показателем, который существенно зависит от используемой виртуальной машины Java. Старый фрагмент кода Java может выполняться на более поздних JVM быстрее даже без перекомпиляции исходного текста.

Как вы догадываетесь, по этой причине (и другим, которые мы обсудим позднее) данная книга не является книгой готовых рецептов по подъему производительности, которые вы должны просто бездумно применять к своему коду. Вместо этого мы концентрируем внимание на ряде аспектов, которые собраны вместе, чтобы обеспечить хорошую производительность.

- Методология обеспечения производительности работы в течение жизненного цикла программного обеспечения
- Теория тестирования в приложении к производительности
- Измерения, статистика и инструментарий
- Навыки анализа (как системного, так и данных)
- Базовые технологии и механизмы

Позже в книге мы представим некоторые эвристики и методы оптимизации на уровне кода, но все это сопровождается оговорками и компромиссами, о которых разработчик должен знать перед их использованием.

Пожалуйста, не пропускайте следующие далее разделы и не пытайтесь применять описываемые далее методы без надлежащего понимания контекста, в котором дается тот или иной совет. Все эти методы без глубокого понимания того, как они должны применяться, способны наделать больше вреда, чем принести пользы.

В общем случае:

- нет никаких переключателей JVM для ускорения ее работы;
- нет никаких советов и хитростей по ускорению работы Java;
- нет никаких секретных алгоритмов, которые от вас скрывают.

Далее мы будем обсуждать эти заблуждения более подробно — как и некоторые другие распространенные ошибки, которые разработчики часто допускают при попытках выполнения анализа производительности Java, а также связанные с этой темой вопросы. Вы все еще здесь? Хорошо. Тогда поговорим о производительности.

Обзор производительности Java

Давайте начнем с рассмотрения классической цитаты Джеймса Гослинга (James Gosling), создателя Java:

> *Java — язык синих воротничков. Это не материал докторской диссертации, а язык для работы.*

Иначе говоря, Java всегда был чрезвычайно практическим языком. Его отношение к производительности первоначально было следующим: пока среда является *достаточно быстрой*, производительность может быть принесена в жертву производительности работы программиста. Поэтому язык программирования Java стал пригодным для высокопроизводительных вычислительных приложений только начиная со сравнительно недавнего времени, по достижении нынешних зрелости и изысканности таких JVM, как HotSpot.

Эта практичность проявляется на платформе Java во многих отношениях, но одним из наиболее очевидных ее проявлений является использование *управляемых подсистем* (managed subsystems). Идея заключается в том, что разработчик отказывается от некоторых аспектов низкоуровневого управления в обмен на то, что ему больше не нужно беспокоиться о подробностях реализации возможностей, находящихся под управлением системы.

Наиболее очевидным примером является, конечно же, управление памятью. JVM обеспечивает автоматическое управление памятью в форме подключаемой подсистемы сборки мусора, так что вопросы работы с памятью не должны решаться программистом вручную.

Управляемые подсистемы встречаются по всей JVM, и их существование привносит дополнительные сложности в поведение среды выполнения приложений JVM.

Как выяснится в следующем разделе, сложное поведение JVM-приложений во время выполнения требует рассмотрения приложений как экспериментов, что заставляет нас думать о статистике наблюдаемых измерений, — и здесь мы делаем неприятное открытие.

Наблюдаемые измерения производительности JVM-приложений очень часто распределены не в соответствии с нормальным законом распределения. Это означает, что элементарные статистические методики (например, *стандартное отклонение* и *дисперсия*) плохо подходят для обработки результатов выполнения JVM-приложений. Дело в том, что многие фундаментальные статистические методы неявно опираются на предположение о нормальном распределении результатов.

В случае JVM-приложений могут быть очень значительными выбросы значений времени работы — чтобы понять это, достаточно взглянуть на приложения с малым временем задержки, например на приложения для торговли. Это означает, что выборка измерений также является проблематичной, поскольку может легко пропустить точные события, которые имеют наиболее важное значение.

И наконец, пару предостережений. Измерения производительности Java могут очень легко ввести в заблуждение. Сложность среды означает, что изолировать отдельные аспекты системы очень трудно.

Измерения также имеют собственные накладные расходы, так что частая выборка (или запись каждого результата) может привести к наблюдаемому воздействию на записываемые характеристики производительности. Характер показателей производительности Java требует определенного статистического подхода, так что прямолинейные, наивные методы исследований часто приводят к неверным результатам при их применении к приложениям Java/JVM.

Исследование производительности как экспериментальная наука

Программное обеспечение Java/JVM, как и большинство современного программного обеспечения, является очень сложным. Фактически из-за высокой оптимизации и адаптивной природы JVM производственные системы, построенные поверх JVM, могут иметь невероятно тонкое и сложное поведение производительности. Эта сложность стала возможной благодаря закону Мура и беспрецедентному росту возможностей оборудования.

Самым удивительным достижением индустрии программного обеспечения является постоянное сведение на нет устойчивых и ошеломляющих успехов, достигаемых в области аппаратного обеспечения.

— Генри Петроски (Henry Petroski)

Хотя некоторые программные системы бездарно потратили впустую исторические достижения промышленности, JVM представляет собой определенный инженерный триумф. С момента создания в конце 1990-х годов JVM превратилась в широко применяемую среду выполнения общего назначения с очень высокой производительностью. Однако, как и любая сложная высокопроизводительная система, чтобы получить максимальную отдачу, JVM требует от своих пользователей мастерства и опыта.

Неточно выполненные измерения хуже чем просто бесполезны.

— Эли Голдратт (Eli Goldratt)

Таким образом, настройка производительности JVM является синтезом технологии, методологии, измерений и инструментов. Ее целью является достижение измеримых результатов, требуемых владельцами или пользователями системы. Другими словами, настройка производительности является экспериментальной наукой — она достигает желаемого результата путем

- определения желаемого результата;
- измерения существующей системы;
- определения того, что должно быть сделано для достижения требуемого результата;
- получения усовершенствованного кода;
- повторного тестирования;
- определения, достигнута ли поставленная цель.

В процессе указания и определения желаемой производительности строится множество количественных целей. Важно установить, что именно должно быть измерено, и записать цели, которые затем образуют часть эталонов и конечных результатов проекта. Рассматривая указанный процесс, мы видим, что анализ производительности основывается на определении нефункциональных требований с последующим их достижением.

Этот процесс, как уже говорилось, не является гаданием на кофейной гуще или каким-то иным ненаучным методом. Мы опираемся на статистические данные и применение соответствующих методов обработки результатов. В главе 5, "Микротесты и статистика", мы рассмотрим фундаментальные статистические методы, которые

требуются для точной обработки данных, получаемых при анализе производительности JVM.

Для многих реальных проектов, несомненно, требуется более глубокое понимание данных и статистических методов. Статистические методы, которые вы встретите в этой книге, служат не более чем отправной точкой для более глубокого их изучения, и ни в малейшей мере не претендуют на полноту описания.

Систематизация производительности

В этом разделе описаны некоторые основные метрики. В нем предоставлен словарь в области анализа производительности, и этот раздел поможет вам формулировать цели настройки проекта в количественном выражении. Эти параметры являются нефункциональными требованиями, определяющими цели производительности. Распространенный базовый набор метрик производительности включает следующие характеристики.

- Пропускная способность
- Задержка
- Нагрузка
- Использование ресурсов (утилизация)
- Эффективность
- Масштабируемость
- Деградация

Мы поочередно вкратце обсудим каждую из характеристик. Обратите внимание, что для большинства проектов повышения производительности одновременно оптимизируются не все показатели. Обычно на одной итерации повышения производительности улучшаются только некоторые из метрик, и вполне реальна ситуация, когда это все, чего можно добиться при попытках одновременного улучшения нескольких характеристик. В реальных проектах также вполне может возникнуть ситуация, когда оптимизация одной метрики выполняется в ущерб другой метрике или группе метрик.

Пропускная способность

Пропускная способность (throughput) — это показатель (метрика), характеризующий скорость работы системы или подсистемы. Обычно пропускная способность выражается как количество единиц работы за некоторый период времени. Например, нас может интересовать, как много транзакций в секунду может выполнить система.

Чтобы пропускная способность играла роль в реальной ситуации, она должна включать описание референтной платформы, для которой она получена. Например, для определения пропускной способности важны аппаратное обеспечение, операционная система и стек программного обеспечения, — как при тестируемой системе, состоящей из одного сервера, так и для кластера. Кроме того, от теста к тесту должно выполняться одно и то же количество транзакций (или единиц работы). По сути, нам следует обеспечить от запуска к запуску одну и ту же нагрузку тестов пропускной способности.

Задержка

Метрики производительности иногда объясняются с использованием сантехнических метафор. Если имеется водонапорная башня, способная подавать в водопровод 100 литров воды в секунду, то объем поставляемой за 1 секунду воды (100 литров) является пропускной способностью. При использовании этой метафоры задержка (latency) представляет собой длину трубы, т.е. время, необходимое для обработки одной транзакции и обнаружения ее результата на другом конце трубы.

Обычно задержку называют временем "из конца в конец". Она зависит от рабочей нагрузки, так что общий подход заключается в создании графика, показывающего задержку как функцию от рабочей нагрузки. Пример такого графика вы увидите ниже, в разделе "Графики производительности".

Нагрузка

Нагрузка (capacity) представляет собой степень параллелизма, демонстрируемую системой, т.е. количество единиц работы (например, транзакций), которые могут одновременно выполняться в системе.

Нагрузка очевидным образом связана с пропускной способностью, и мы должны ожидать, что с увеличением параллельной нагрузки системы будут затронуты и пропускная способность, и задержка. По этой причине нагрузка обычно трактуется (определяется) как возможности по обработке данных при заданном значении задержки или пропускной способности.

Использование ресурсов (утилизация)

Одной из наиболее распространенных задач анализа производительности является достижение эффективного использования системных ресурсов. В идеале процессор должен работать, а не простаивать (или тратить время на задания операционной системы или другие задачи обслуживания).

В зависимости от загрузки разница между уровнями использования различных ресурсов может быть огромной. Например, интенсивные вычисления (например,

обработка графики или шифрование) могут загрузить процессор практически на 100%, но при этом использовать только небольшой процент доступной памяти.

Эффективность

Деление пропускной способности системы на используемые ресурсы дает меру общей эффективности системы. Интуитивно это имеет смысл, так как требование большего количества ресурсов для получения той же пропускной способности указывает на меньшую эффективность.

Можно также (при работе с более крупными системами) использовать для измерения эффективности стоимость. Если решение А имеет общую стоимость, в два раза бо́льшую, чем решение В при той же пропускной способности, то очевидно, что это решение вдвое менее эффективно.

Масштабируемость

Пропускная способность и нагрузка системы зависят от доступных для работы ресурсов. Изменение пропускной способности при добавлении ресурсов является мерой масштабируемости системы или приложения. Святой Грааль масштабируемости системы — когда ее пропускная способность растет точно в той же степени, что и ресурсы.

Рассмотрим систему, основанную на кластере серверов. Если кластер расширяется, например, путем удвоения его размера, то какая пропускная способность кластера может быть достигнута? Если новый кластер может обрабатывать в два раза больше транзакций, то такая система демонстрирует "идеальное линейное масштабирование". Его очень трудно добиться на практике, особенно при широком спектре возможных нагрузок.

Масштабируемость системы зависит от ряда факторов и обычно не является простым константным множителем. Часто возникает ситуация, когда система близка к линейному масштабированию для некоторого диапазона ресурсов, но при высоких нагрузках начинают работать те или иные ограничения, не позволяющие добиться идеального масштабирования.

Деградация

При увеличении нагрузки системы — путем увеличения числа запросов (или клиентов) или увеличения частоты их поступления — мы можем обнаружить изменения наблюдаемых задержек и/или пропускной способности.

Обратите внимание, что это изменение зависит от использования ресурсов. Если система использует ресурсы не в полном объеме, то у нее имеется некоторый резерв перед тем, как начнут наблюдаться изменения в ее работе, но если ресурсы задействованы полностью, то остановка роста пропускной способности или увеличение

задержки будет наблюдаться немедленно. Эти изменения обычно называют деградацией системы при дополнительной нагрузке.

Связи между наблюдаемыми характеристиками

Поведение одних наблюдаемых характеристик производительности обычно тем или иным образом связано с поведением других. Подробности этой связи будут зависеть от того, работает ли система при пиковой нагрузке. Например, в общем случае использование ресурсов будет изменяться по мере увеличения нагрузки на систему; однако если система нагружена недостаточно, то повышение нагрузки может не привести к заметному увеличению использования ресурсов. И наоборот, если система уже перегружена, то увеличение нагрузки может приводить к наблюдению изменения других характеристик производительности.

В качестве еще одного примера масштабируемость и деградация представляют собой изменения поведения системы при увеличении нагрузки. Что касается масштабируемости, то при увеличении нагрузки и имеющихся ресурсах центральным вопросом становится вопрос о том, в состоянии ли система их использовать. С другой стороны, при увеличении нагрузки без выделения дополнительных ресурсов ожидаемым наблюдаемым результатом является деградация некоторых характеристик производительности (например, задержки).

В редких случаях дополнительная нагрузка может привести к контринтуитивным результатам. Например, если изменение нагрузки приводит к тому, что некоторая часть системы переключается в более ресурсоемкий режим работы, но с более высокой производительностью, то в результате при росте количества запросов задержка может даже уменьшиться.

Приведем один пример. В главе 9, "Выполнение кода в JMV", мы будем подробно рассматривать JIT-компилятор HotSpot. Чтобы быть рассматриваемым в качестве претендента на JIT-компиляцию, метод должен выполняться в режиме интерпретации "достаточно часто". Таким образом, возможна ситуация, когда при низкой нагрузке ключевые методы "застревают" в режиме интерпретации, но при повышенных нагрузках оказываются подходящими для компиляции за счет увеличения частоты вызова. Это приводит к тому, что последующие вызовы того же метода работают гораздо быстрее, чем предыдущие.

Различные рабочие нагрузки могут приводить к очень различающимся характеристикам. Например, торговые операции на финансовых рынках, будучи просматриваемыми от начала до конца, могут иметь время выполнения (т.е. задержку) в несколько часов или даже дней. Однако в крупном банке в любой момент времени могут обрабатываться миллионы из них. Таким образом, нагрузка системы оказывается огромной, но и задержка очень велика.

Но давайте рассмотрим только одну подсистему банка. Сопоставление покупателей продавцам (которые, по сути, представляют собой стороны, договорившиеся о цене) известно как *сопоставление заказов* (order matching). Такие отдельные подсистемы могут в любой момент времени иметь сотни только отложенных заказов, но задержка от принятия заказа до завершения сопоставления может быть всего лишь порядка миллисекунды (или даже меньше, в случае торговли с малой задержкой).

В этом разделе мы упомянули только наиболее часто встречающиеся наблюдаемые характеристики производительности. Иногда используются несколько отличные от приведенных определения или даже иные метрики, но в большинстве случаев они являются числовыми характеристиками базовой системы, обычно используемыми для настройки производительности, и представляют собой средство систематизации для обсуждения эффективности рассматриваемой системы.

Графики производительности

В заключение этой главы давайте посмотрим на некоторые распространенные модели поведения, которые встречаются в тестах производительности. Мы изучим данный вопрос, рассматривая реально наблюдаемые на практике графики, а по мере работы над книгой вы познакомитесь со многими другими примерами графиков.

График на рис. 1.1 показывает внезапное увеличение деградации производительности (в данном случае рост задержки) при увеличении нагрузки — обычно график такого вида называют *локтем производительности* (performance elbow).

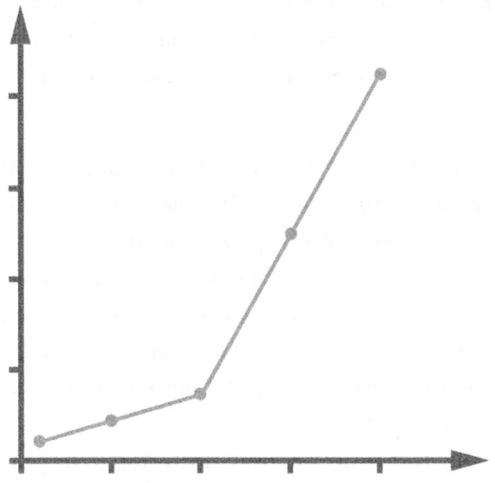

Рис. 1.1. Локоть производительности

С другой стороны, на рис. 1.2 показан гораздо более счастливый случай почти линейного масштабирования по мере добавления машин в кластер. Это почти идеальное поведение, которое может быть достигнуто лишь в чрезвычайно благоприятных

обстоятельствах, например масштабирование протокола без состояния, у которого отсутствует необходимость связи сеансов с одним и тем же сервером.

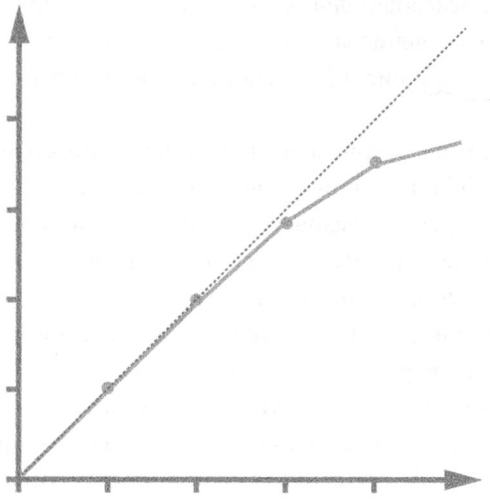

Рис. 1.2. Масштабирование, близкое к линейному

В главе 12, "Методы повышения производительности параллельной работы", мы встретимся с законом Амдала, названным в честь знаменитого ученого Джина Амдала (Gene Amdahl) из IBM. На рис. 1.3 показано графическое представление этого фундаментального ограничения на масштабируемость. Здесь показано максимально возможное ускорение вычислений как функция от количества процессоров, выделенных для решения задачи.

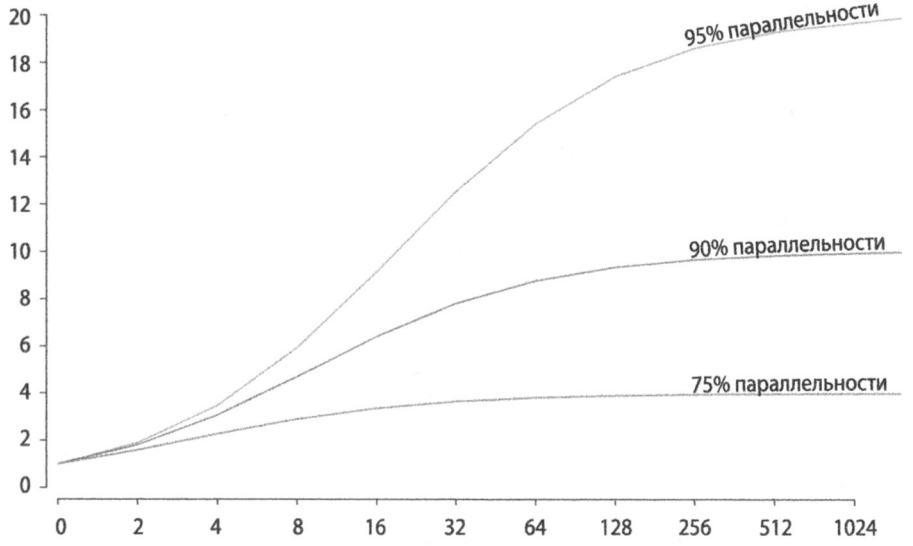

Рис. 1.3. Закон Амдала

Здесь показаны три случая: когда основная задача распараллеливаема на 75%, 90% и 95%. Это ясно показывает, что, если рабочая нагрузка имеет любую часть, которая должна выполняться последовательно, линейная масштабируемость невозможна, и существует строгое ограничение максимально достижимой масштабируемости. Это оправдывает комментарии к рис. 1.2 — даже в лучшем случае линейная масштабируемость недостижима.

Ограничения, вводимые законом Амдала, носят удивительно ограничительный характер. В частности, обратите внимание, что по оси X масштаб логарифмический, так что даже если алгоритм последовательный всего лишь на 5%, для ускорения в 12 раз требуется 32 процессора. Что еще хуже — максимальное ускорение для этого алгоритма не более чем 20-кратное, независимо от того, сколько ядер используется. На практике же многие алгоритмы последовательны куда более чем на 5%, и поэтому их максимально возможное ускорение и того меньше.

Как мы увидим в главе 6, "Сборка мусора", базовая технология в подсистеме сбора мусора JVM для нормальных не слишком нагруженных приложений естественным путем приводит к шаблону использования памяти типа "зубьев пилы"; соответствующий пример зависимости используемой памяти от времени (полученный с помощью инструментария jClarity Censum) показан на рис. 1.4.

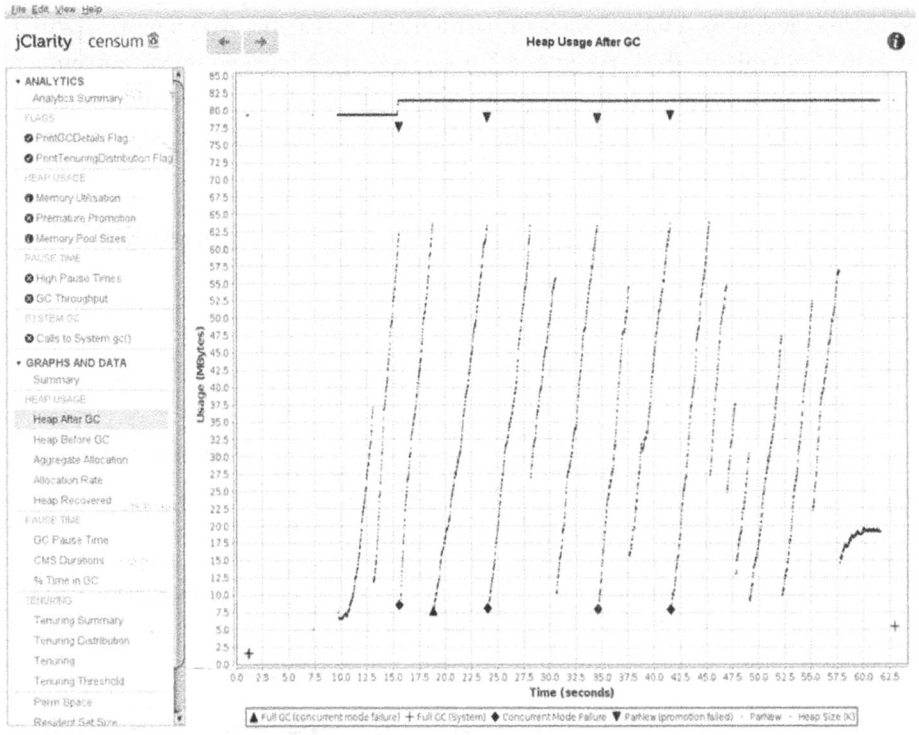

Рис. 1.4. Нормальное использование памяти приложением

На рис. 1.5 показан другой график использования памяти, который может иметь большое значение при настройке скорости распределения памяти приложением. Этот небольшой пример демонстрирует работу приложения по вычислению чисел Фибоначчи. Он ясно указывает на резкое снижение темпа распределения примерно на 90-й секунде работы.

Другие графики, предоставляемые тем же инструментарием jClarity Censum, показывают, что в этот момент приложение начинает страдать от проблемы сбора мусора и не может выделить достаточного количества памяти из-за того, что в борьбу за процессорное время вступают потоки сборки мусора.

Можно также увидеть, что подсистема распределения памяти работает в наряженном режиме — выделяя свыше 4 Гбайт памяти в секунду. Это значительно превышает рекомендуемую максимальную нагрузку большинства современных систем (включая аппаратные средства класса серверов). Мы поговорим о распределении памяти подробнее, когда будем обсуждать сборку мусора в главе 6, "Сборка мусора".

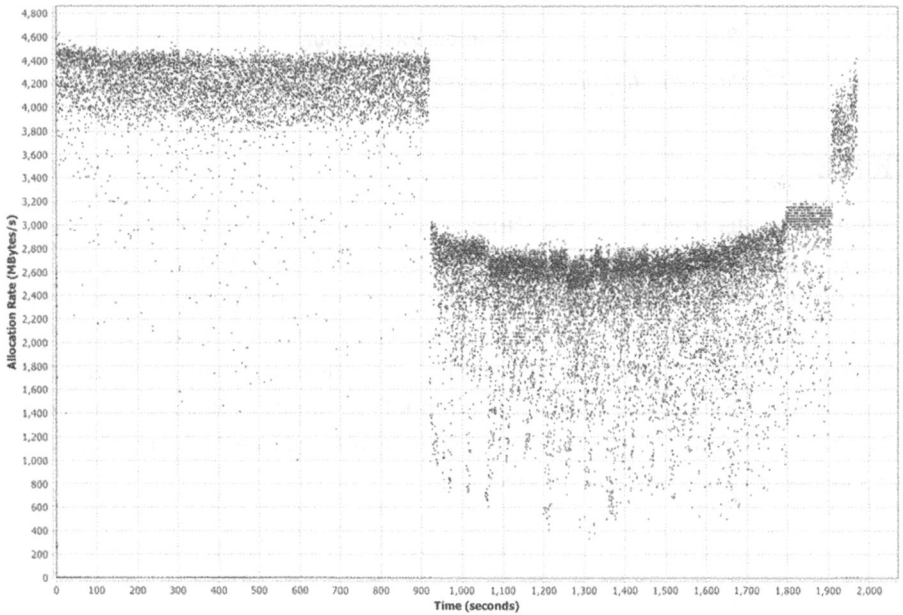

Рис. 1.5. Пример проблематичной скорости выделения памяти

Утечка ресурсов в системе чаще всего проявляется так, как показано на рис. 1.6, где наблюдаемая характеристика (в данном случае задержка) медленно деградирует с увеличением нагрузки — до достижения точки перегиба, после которой скорость деградации системы резко возрастает.

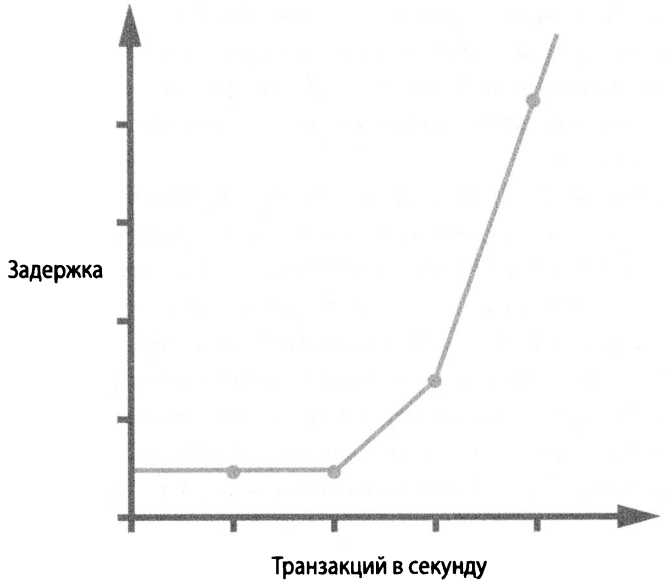

Рис. 1.6. Деградация задержки при высокой нагрузке

Резюме

В этой главе мы начали обсуждение того, что такое производительность Java и чем она не является. Мы познакомили вас с азами эмпирического подхода и измерений, с базовой терминологией и с наблюдаемыми характеристиками производительности. Наконец, мы рассмотрели некоторые распространенные ситуации, часто встречающиеся в результатах тестов производительности. Давайте теперь приступим к рассмотрению некоторых основных аспектов JVM и подготовим фундамент для понимания всех тех тонкостей, которые делают оптимизацию производительности на основе JVM особенно сложной проблемой.

ГЛАВА 2
Обзор JVM

Нет никаких сомнений в том, что Java является одной из крупнейших технологических платформ на планете, обладая примерно 9–10 миллионами разработчиков (согласно информации Oracle). Дизайн языка таков, что большинству разработчиков не нужно знать о низкоуровневых тонкостях платформы, с которой они работают. В результате обычно разработчики впервые сталкиваются с этими аспектами языка, только когда клиент жалуется на производительность приложения.

Для разработчиков, заинтересованных в производительности, однако, важно понять основы стека технологий JVM. Понимание технологии JVM позволяет разработчикам писать лучшее программное обеспечение и предоставляет теоретические основы, необходимые для изучения проблем, связанных с производительностью.

В этой главе описано, как JVM выполняет Java-программы, чтобы заложить основу для более глубокого изучения этих тем далее в книге. В частности, в главе 9, "Выполнение кода в JMV", более глубоко исследуется байт-код. Одним из вариантов ваших действий может быть прочтение данной главы, а затем, дойдя до главы 9, вы сможете еще раз вернуться к ней — уже с более глубоким пониманием других тем.

Интерпретация и загрузка классов

В соответствии со спецификацией, определяющей виртуальную машину Java (Java Virtual Machine), JVM является стековой интерпретирующей машиной. Это означает, что вместо регистров (как у физического аппаратного процессора) она использует стек частичных результатов и выполняет вычисления, работая со значением (или значениями) на вершине этого стека.

Базовое поведение интерпретатора JVM можно рассматривать как, по сути, "конструкцию `switch` внутри цикла `while`" — независимую обработку каждого кода операции в программе с использованием стека вычислений для хранения промежуточных значений.

Когда мы запускаем наше приложение с помощью команды `java HelloWorld`, операционная система запускает процесс виртуальной машины (бинарный файл `java`). При этом настраивается виртуальная среда Java и инициализируется стековая

машина, которая и будет фактически выполнять пользовательский код в файле класса `HelloWorld`.

Как мы увидим, когда углубимся во внутренности Oracle/OpenJDK VM (HotSpot), ситуация в случае реальных промышленных интерпретаторов Java может быть более сложной, но `switch`-внутри-`while` пока что остается вполне приемлемой мысленной моделью.

Точкой входа в приложение является метод `main()` класса `HelloWorld.class`. Для передачи управления этому классу он должен быть загружен виртуальной машиной до начала выполнения.

Для достижения этой цели используется механизм загрузки классов (classloading) Java. При инициализации нового Java-процесса используется цепочка загрузчиков. Первоначальный загрузчик известен как самозагрузчик (Bootstrap classloader) и содержит классы ядра среды времени выполнения Java. В версиях Java до 8 включительно они загружаются из файла `rt.jar`. В версии 9 и более поздних среда выполнения модуляризована и концепция загрузки классов претерпела ряд изменений.

Основная цель самозагрузчика — загрузить минимальный набор классов (который включает такие фундаментальные вещи, как `java.lang.Object`, `Class` и `Classloader`), чтобы позволить другим загрузчикам классов загрузить остальную часть системы.

Java моделирует загрузчики классов как объекты в собственной среде выполнения и системе типов, поэтому имеется необходимость в некотором отдельном способе загрузки первоначального набора классов — иначе в определении того, что такое загрузчик классов, будет наблюдаться зацикленность.

Следующим создается загрузчик Extension (расширение); его родительским загрузчиком является Bootstrap, и при необходимости он может выполнять делегирование своему родителю. Расширения широко не используются, но могут предоставлять перекрытия и машинный код для конкретных операционных систем и платформ. В частности, среда выполнения Nashorn JavaScript, введенная в Java 8, загружается именно загрузчиком Extension.

Наконец, создается загрузчик классов Application (приложение); он отвечает за загрузку пользовательских классов из определенных путей. К сожалению, в некоторых описаниях его называют го системным загрузчиком классов (System). Этого термина следует избегать по той простой причине, что этот загрузчик не загружает системные классы (что делает загрузчик Bootstrap). Загрузчик Application встречается очень часто; загрузчик Extension выступает по отношению к нему в качестве родительского.

Java загружает новые классы, когда они впервые встречаются во время выполнения программы. Если загрузчик не может найти класс, поиск обычно

делегируется родительскому загрузчику. Если цепочка поисков достигает загрузчика классов Bootstrap и не находит искомый класс, генерируется исключение `ClassNotFoundException`. Важно, чтобы разработчики использовали процесс построения, который эффективно выполняет компиляцию с точно тем же путем к классам, который будет использоваться в производственной системе, так как это помогает смягчить указанную проблему.

При нормальных обстоятельствах Java загружает класс только один раз, и при этом создается объект `Class` для представления класса в среде выполнения. Однако важно понимать, что один и тот же класс потенциально может быть загружен дважды разными загрузчиками. В результате класс в системе идентифицируется как загрузившим его загрузчиком, так и полностью квалифицированным именем класса (включающим имя пакета).

Выполнение байт-кода

Важно понимать, что исходный код Java проходит перед выполнением через значительное количество преобразований. Первым является шаг компиляции с использованием Java-компилятора `javac`, который зачастую вызывается в качестве части более широкого процесса построения.

Задание `javac` заключается в преобразовании кода Java в `.class`-файлы, содержащие байт-код. Это достигается путем довольно простой процедуры трансляции исходного кода Java, показанной на рис. 2.1. Во время компиляции `javac` выполняются очень немногие оптимизации, так что получаемый байт-код по-прежнему достаточно читаем и распознаваем как Java-код при просмотре с помощью дизассемблирующего инструмента, такого как стандартный `javap`.

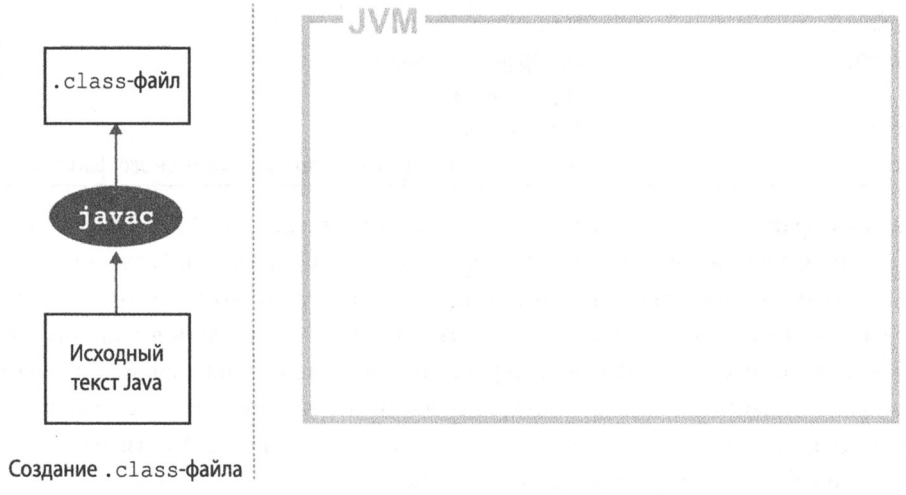

Рис. 2.1. Компиляция исходного текста Java

Байт-код — это промежуточное представление программы, не привязанное к конкретной машинной архитектуре. Независимость от архитектуры машины обеспечивает переносимость, означающую, что уже разработанное (или скомпилированное) программное обеспечение может работать на любой платформе, поддерживающей JVM и абстракции языка Java. Это первый важный момент в понимании того, как JVM выполняет код.

В настоящее время язык программирования Java в значительной степени независим от виртуальной машины Java, так что буква "J" в аббревиатуре "JVM" немного вводит в заблуждение, поскольку JVM в состоянии выполнять любой язык JVM, который может сгенерировать корректный файл класса. Фактически на рис. 2.1 может так же легко быть изображен компилятор `scalac` языка программирования Scala, генерирующий байт-код для выполнения в JVM.

Независимо от исходного кода, использованного компилятором, результирующий файл класса имеет точно определенную структуру, указанную в спецификации VM (табл. 2.1). Любой класс, который загружается JVM, будет проверяться на соответствие ожидаемому формату перед тем, как будет позволено его выполнение.

Таблица 2.1. Анатомия файла класса

Компонент	Описание
"Магическое число"	`0xCAFEBABE`
Версия формата	Младшая и старшая версии файла класса
Пул констант	Пул констант класса
Флаги доступа	Является ли класс абстрактным, статическим и т.д.
Текущий класс	Имя текущего класса
Суперкласс	Имя суперкласса
Интерфейсы	Интерфейсы класса
Поля	Поля класса
Методы	Методы класса
Атрибуты	Атрибуты класса (например, имя исходного файла и т.п.)

Каждый файл класса начинается с магического числа `0xCAFEBABE`. Эти 4 байта служат для указания того, что файл имеет формат файла класса. Следующие 4 байта представляют собой младшую и старшую части версии компилятора, использованного для компиляции файла класса; эти значения проверяются для того, чтобы убедиться, что целевая JVM имеет версию не ниже использовавшейся для компиляции файла класса. Младшая и старшая части версии проверяются загрузчиком классов для обеспечения совместимости; если версии не совместимы, во время выполнения будет сгенерировано исключение `UnsupportedClassVersionError`,

свидетельствующее о том, что среда выполнения имеет более раннюю версию, чем файл скомпилированного класса.

Магические числа предоставляют для операционных систем типа Unix способ определения типа файла (в то время как операционная система Windows обычно полагается на расширение имени файла). По этой причине один раз принятое значение очень трудно изменить, а значит, в Java в обозримом будущем будет использоваться вызывающее недовольство феминисток сексистское значение `0xCAFEBABE` (хотя в Java 9 для файлов модулей введено магическое число `0xCAFEDADA`).

Пул констант хранит используемые кодом константные значения (например, имена классов, интерфейсов и полей). Когда JVM выполняет код, таблица пула констант используется для того, чтобы использовать ссылки на значения, а не полагаться схему памяти во время выполнения.

Флаги доступа используются для определения модификаторов, примененных к классу. Первая часть флагов определяет общие свойства, например является ли класс открытым, нет ли у него модификатора `final`, так что он является окончательным и не может быть наследован. Флаги также определяют, представляет ли файл класса интерфейс или абстрактный класс. В заключительной части флагов указывается, не представляет ли файл класса синтетический класс (который отсутствует в исходном коде), тип аннотации или перечисление.

Записи класса `this`, суперкласса и интерфейсов индексируются в пуле констант для определения иерархии типов, принадлежащих классу. Поля и методы определяют сигнатурообразную структуру, включающую модификаторы, которые применяются к полю или методу. Затем для представления структурированных элементов более сложных структур и структур без фиксированного размера используется набор атрибутов. Например, методы используют атрибут Code для представления байт-кода, связанного с этим конкретным методом.

В приведенном далее очень простом примере кода можно наблюдать результат работы `javac`:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 10; i++)
        {
            System.out.println("Hello World");
        }
    }
}
```

Java поставляется с дизассемблером файлов классов под названием `javap`, который позволяет изучать `.class`-файлы. Взяв файл класса `HelloWorld` и запустив `javap -c HelloWorld`, мы получим следующий результат:

```
public class HelloWorld
{
    public HelloWorld();
      Code:
          0:
            aload_0
          1:
            invokespecial #1 // Метод java/lang/Object."<init>":()V
          4:
            return

    public static void main(java.lang.String[]);
      Code:
          0:
            iconst_0
          1:
            istore_1
          2:
            iload_1
          3:
            bipush        10
          5:
            if_icmpge     22
          8:
            getstatic     #2 // Поле java/lang/System.out ...
          11:
            ldc           #3 // Строка Hello World
          13:
            invokevirtual #4 // Метод java/io/PrintStream.println ...
          16:
            iinc          1, 1
          19:
            goto          2
          22:
            return
}
```

Эта схема описывает байт-код файла `HelloWorld.class`. Для вывода большего количества подробностей `javap` имеет параметр `-v`, который приводит к предоставлению полной информации о заголовке файла класса и пуле констант. Файл класса содержит два метода, хотя в исходном файле имеется только метод `main()`; это результат работы `javac`, который автоматически добавил к классу конструктор по умолчанию.

Первая инструкция, выполняемая в конструкторе, — aload_0, которая помещает ссылку this в первую позицию стека. Затем вызывается команда invokespecial, которая вызывает метод экземпляра, который отвечает за обработку вызова суперконструкторов и создание объектов. В конструкторе по умолчанию вызываемый метод соответствует конструктору по умолчанию класса Object, так как перекрытие предоставлено не было.

Коды операций JVM краткие и представляют тип, операцию и взаимодействия между локальными переменными, пулом констант и стеком.

Перейдем к методу main(). iconst_0 помещает целочисленную константу 0 в стек вычислений. istore_1 сохраняет это константное значение в локальной переменной со смещением 1 (представлена в цикле как i). Смещения локальных переменных начинаются с 0, но в методах экземпляров нулевая запись всегда представляет собой this. Затем переменная со смещением 1 загружается обратно в стек, как и константа 10, сравнение с которой выполняется с помощью инструкции if_icmpge (целочисленное сравнение "больше или равно"). Тест завершается успешно, только если текущее целочисленное значение не меньше 10.

Для нескольких первых итераций этот тест на сравнение оказывается непройденным, так что продолжается выполнение инструкции 8. Здесь выполняется разрешение статического метода из System.out, после чего загружается строка "Hello World" из пула констант. Следующий вызов, invokevirtual, вызывает метод экземпляра на основе класса. Целочисленное значение затем увеличивается, вызывается goto для возврата к инструкции 2.

Этот процесс продолжается до тех пор, пока не будет успешно пройдено сравнение if_icmpge (пока переменная цикла не станет >=10). На этой итерации цикла управление передается инструкции 22 и выполняется возврат из метода.

Введение в HotSpot

В апреле 1999 года Sun предоставил одно из самых больших изменений в Java с точки зрения производительности. Виртуальная машина HotSpot представляет собой ключевую функциональную возможность Java, которая развивалась с целью обеспечить производительность, сопоставимую (или даже превышающую) с производительностью таких языков, как C и C++ (рис. 2.2). Чтобы объяснить, как это возможно, давайте углубимся немного в проектирование языков, предназначенных для разработки приложений.

Проектирование языка и платформы часто включает в себя принятие решений о компромиссах между желаниями и возможностями. В данном случае раздел расположен между языками, которые находятся "близко к железу" и основаны на таких

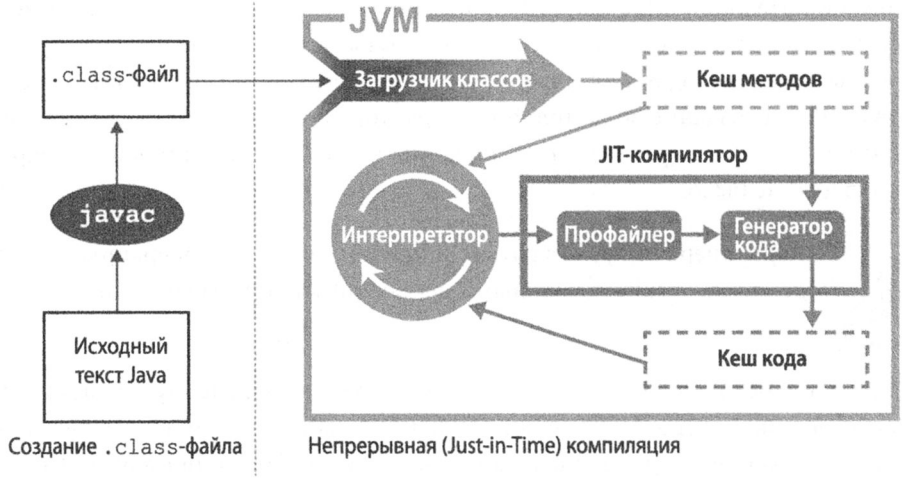

Рис. 2.2. HotSpot JVM

идеях, как "абстракции нулевой стоимости", и языками, которые заботятся, в первую очередь, о производительности разработчиков и о том, чтобы "все было сделано", не опускаясь до низкоуровневого контроля.

> *Реализации C++ подчиняются принципу нулевых накладных расходов: вы не платите за то, что не используете. И кроме того, то, что вы используете, вы не сможете лучше закодировать вручную.*
>
> *— Бьярне Страуструп (Bjarne Stroustrup)*

Принцип нулевых накладных расходов замечательно звучит в теории, но требует от всех пользователей языка работать с низкоуровневой реальностью операционных систем и компьютеров. Это значительное бремя, накладываемое на разработчиков, которые могут не заботиться о производительности как основной цели.

Но это не все. Такой язык требует также, чтобы исходный код компилировался в машинный код соответствующей платформы во время построения — такой подход обычно называют *Ahead-of-Time* (AOT, досрочной) компиляцией. Это связано с тем, что альтернативные модели выполнения, такие как интерпретаторы, виртуальные машины и слои переносимости, определенно, не являются программным обеспечением с нулевыми накладными расходами.

Принцип также скрывает сложные детали[1] под лозунгом "то, что вы используете, вы не сможете лучше закодировать вручную". Это предполагает наличие целого ряда вещей, для которых разработчик, в принципе, может создать код, лучший, чем автоматизированная система. В общем случае это небезопасное предположение. Очень

[1] В оригинале — идиоматическое выражение "can of worm". — *Примеч. пер.*

немногие люди хотят программировать на языке ассемблера, так что использование автоматизированных систем (таких, как компиляторы) для получения кода, очевидно, дает выгоды для большинства программистов.

Язык программирования Java никогда не подписывался на нулевые накладные расходы. Вместо этого подход, принятый виртуальной машиной HotSpot, состоит в анализе поведения среды выполнения программы и грамотном применении оптимизации там, где она обеспечит максимальный выигрыш в производительности. Цель виртуальной машины HotSpot заключается в том, чтобы позволить вам писать идиоматические программы на Java и следовать принципам хорошего дизайна, а не укладывать вашу программу в прокрустово ложе виртуальной машины.

Введение в JIT-компиляцию

Программы Java начинают выполняться в интерпретаторе байт-кода, где их инструкции выполняются виртуализированной стековой машиной. Это абстрагирование от процессора дает преимущество переносимости файлов классов, но чтобы добиться максимальной производительности, ваши программы должны выполняться непосредственно процессором, используя машинные возможности.

HotSpot достигает этого путем компиляции модулей программы из интерпретированного байт-кода в машинный код. Модулями компиляции в HotSpot VM являются метод и цикл. Такой подход известен как *своевременная* (Just-in-Time — JIT) компиляция.

JIT-компиляция работает путем мониторинга приложения, выполняемого в режиме интерпретации, и выявления наиболее часто выполняемых фрагментов кода. В ходе анализа собирается информация, которая позволяет выполнять более сложную оптимизацию. Когда выполнение некоторого конкретного метода переходит установленный порог, профайлер запрашивает компиляцию и оптимизацию этого фрагмента кода.

JIT-подход к компиляции имеет много преимуществ, но одним из главных является то, что он основан на данных трассировки, собранной на этапе интерпретации, что позволяет HotSpot принять более обоснованные и разумные решения, касающиеся оптимизации.

И это не все. В HotSpot вложены сотни (если не более) человеко-лет инженерных усилий, и каждая его новая версия привносит новые оптимизации и предоставляет программистам новые преимущества. Это означает, что любое Java-приложение, работающее на новой версии HotSpot, может воспользоваться новыми возможностями по оптимизации производительности, присутствующими в этой версии виртуальной машины, при этом не требуя перекомпиляции приложения.

После трансляции исходного кода Java в байт-код и еще одного этапа JIT-компиляции фактически выполняемый код очень существенно отличается от написанного исходного кода. Это ключевой момент, который будет управлять нашим подходом к исследованиям производительности. Код после JIT-компиляции, выполняющийся виртуальной машиной, может выглядеть не имеющим ничего общего с оригинальным исходным кодом на Java.

Общая картина такова, что языки наподобие C++ (и многообещающего Rust), как правило, имеют более предсказуемую производительность, но ценой перекладывания многих низкоуровневых сложностей на плечи пользователя.

Обратите внимание, что "более предсказуемый" не обязательно означает "лучший". AOT-компиляторы создают код, который может выполняться на широком классе процессоров, и в результате может быть не в состоянии выяснить, какие возможности доступны при выполнении на том или ином конкретном процессоре.

Среды с использованием оптимизации на основе профилирования (profile-guided optimization — PGO), такие как Java, имеют возможность использования информации времени выполнения такими способами, которые просто невозможны для большинства платформ AOT. В результате возможно повышение производительности, такое как динамическое встраивание и оптимизирующее устранение виртуальных вызовов. HotSpot может даже определить точный тип процессора, на котором выполняется виртуальная машина, и использовать эту информацию для выполнения оптимизации, использующей особенности конкретного процессора, если таковые имеются.

Методика выявления возможностей конкретного процессора известна как *встроенные (внутренние) средства JVM* (JVM intrinsics) встроенных функций, которые не следует путать с внутренней блокировкой (intrinsic locks), связанной с применением ключевого слова `synchronized`.

Полное описание PGO и JIT-компиляции можно найти в главах 9, "Выполнение кода в JMV", и 10, "JIT-компиляция".

Сложный подход HotSpot приносит большую пользу для большинства обычных разработчиков, но принятый компромисс (отказ от абстракции нулевой стоимости) означает, что в каждом конкретном случае высокопроизводительных Java-приложений разработчик должен быть очень осторожным, чтобы избежать рассуждений на основе "здравого смысла" и слишком упрощенных умозрительных моделей выполнения Java-приложений.

Анализ производительности малых фрагментов кода Java (выполнение *микротестов* (microbenchmarks)) обычно сложнее анализа всего приложения и представляет собой очень специализированную задачу, браться за которую большинству разработчиков не следует. Мы вернемся к этой теме в главе 5, "Микротесты и статистика".

Подсистема компиляции HotSpot является одной из двух наиболее важных подсистем, предоставляемых виртуальной машиной. Второй является автоматическое управление памятью, которое являлось одним из самых важных достоинств Java в ранние годы этого языка программирования.

Управление памятью в JVM

В таких языках, как C, C++ и Objective-C, программист отвечает за управление выделением и освобождением памяти. Преимуществами управления памятью и временем существования объектов являются более детерминированная производительность и способность привязать жизненный цикл ресурсов к созданию и удалению объектов. Но эти преимущества даются дорогой ценой — чтобы написать корректно работающую программу, разработчики должны быть способны точно и аккуратно работать с динамически выделяемой памятью.

К сожалению, многолетний практический опыт показал, что многие разработчики плохо понимают идиомы и схемы управления памятью. Более поздние версии C++ и Objective-C улучшили ситуацию путем применения идиомы интеллектуальных указателей в стандартной библиотеке. Однако во времена создания Java одной из основных причин ошибок приложений было плохое управление памятью.

Язык программирования Java пытался помочь решить эту проблему путем введения автоматического управления динамической памятью с помощью процесса, известного как *сборка мусора* (garbage collection — GC). Попросту говоря, сборка мусора представляет собой недетерминированный процесс, который запускается для освобождения и повторного использования более ненужной памяти в ситуации, когда от JVM требуется выделение большего количества памяти.

Однако на самом деле история сборки мусора не совсем так проста, и на протяжении всей истории Java для сборки мусора были разработаны и применялись различные алгоритмы. Сборка мусора не бесплатна: когда она работает, зачастую *мир останавливается*, а это означает, что во время сборки мусора выполнение приложения приостанавливается. Обычно эти паузы очень малы, но при высокой нагрузке могут увеличиваться.

Сборка мусора является одной из основных тем в рамках оптимизации производительности Java, поэтому детальному рассмотрению сборки мусора мы выделим целых три главы — 6, "Сборка мусора", 7, "Вглубь сборки мусора", и 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора".

Многопоточность и модель памяти Java

Одним из основных достижений Java с первой же версии языка была встроенная поддержка многопоточного программирования. Платформа Java позволяет разработчикам создавать новые потоки выполнения. Например, вот как это выглядит в Java 8:

```
Thread t = new Thread(() -> {System.out.println("Hello World!");});
t.start();
```

Это не все — сама среда Java по своей природе многопоточна. Это привносит дополнительные сложности в поведение программ на языке Java и делает работу аналитика производительности еще более трудной.

В большинстве основных реализаций JVM каждый поток приложения Java в точности соответствует одному выделенному потоку операционной системы. Альтернатива, состоящая в использовании общего пула потоков для выполнения всех потоков приложения Java (подход, известный под названием *зеленые потоки* (green threads)), как выяснилось, не в состоянии обеспечить приемлемую производительность и добавляет излишнюю сложность в реализацию.

Можно безопасно считать, что каждый поток приложения JVM связан с единственным уникальным потоком операционной системы, который создается при вызове метода `start()` для соответствующего объекта `Thread`.

Подход Java к многопоточности датируется концом 1990-х годов и основан на следующих основополагающих принципах.

- Все потоки процесса Java совместно используют единую динамическую память со сборкой мусора.
- Обратиться к объекту, созданному одним потоком, может любой другой поток, имеющий ссылку на этот объект.
- По умолчанию объекты изменяемы, т.е. значения, хранящиеся в полях объекта, могут быть изменены, если только программист явно не укажет ключевое слово `final` для маркировки их как неизменяемых.

Модель памяти Java (Java Memory Model — JMM) является формальной моделью памяти, которая объясняет, каким образом различные потоки выполнения видят изменения значений в объектах. То есть, если и поток A, и поток B имеют ссылки на объект `obj` и поток A изменяет его, то что происходит со значением, наблюдаемым потоком B?

Этот, казалось бы, простой вопрос на самом деле куда сложнее, чем кажется, поскольку планировщик операционной системы (с которым мы встретимся в главе 3, "Аппаратное обеспечение и операционные системы") может принудительно

вытеснять потоки из ядер процессора. Это может привести к тому, что прежде чем исходный поток завершит свою работу с ним, начнется выполнение другого потока с обращением к объекту, и в результате этот другой поток потенциально может увидеть объект в поврежденном или недопустимом состоянии.

Единственный способ защиты от этого потенциального повреждения объекта во время выполнения параллельного кода, предоставляемый ядром Java, — блокировки взаимного исключения, которые могут оказаться весьма сложными при использовании в реальных приложениях. В главе 12, "Методы повышения производительности параллельной работы", подробно рассматриваются работа JMM и практические аспекты работы с потоками и блокировками.

Встреча с JVM

Многие разработчики могут быть знакомы только с единственной реализацией Java — от корпорации Oracle. Мы уже встречались с виртуальной машиной HotSpot из реализации Oracle, но существуют и другие реализации, которые (с различной степенью глубины) мы будем рассматривать в этой книге.

OpenJDK

OpenJDK представляет собой интересный частный случай. Это проект с открытым кодом (GPL), который обеспечивает эталонную реализацию Java. Проект находится под руководством и поддержкой Oracle и предоставляет основу для его выпусков Java.

Oracle

Java от Oracle является наиболее широко известной реализацией. Она основана на OpenJDK, но распространяется под частной лицензией Oracle. Почти все изменения в Oracle Java начинаются как исправления в публичном хранилище OpenJDK (за исключением исправлений безопасности, которые еще не были открыты для широкой публики).

Zulu

Zulu является бесплатной (с лицензией GPL) реализацией OpenJDK, полностью Java-сертифицированной и предоставляемой компанией Azul Systems. Она свободна от патентованных лицензий и представляет собой свободно распространяемый пакет. Azul — один из немногих производителей, предоставляющих платную поддержку OpenJDK.

IcedTea

Red Hat был первым отличным от Oracle поставщиком, производившим полностью сертифицированную реализацию Java на основе OpenJDK. IcedTea

представляет собой полностью сертифицированный распространяемый пакет.

Zing

Zing — высокопроизводительная фирменная JVM. Это полностью сертифицированная реализация Java, производящаяся Azul Systems. Ее единственная версия — 64-разрядная версия для Linux, предназначенная для серверных систем с большим объемом памяти (десятки и сотни гигабайтов) и большим количеством процессоров.

J9

J9 от IBM начала свою жизнь как фирменная JVM, но со временем стала системой с открытым исходным кодом (так же, как HotSpot). В настоящее время является надстройкой над открытым проектом Eclipse и образует основу для фирменных продуктов IBM. J9 полностью соответствует сертификации Java.

Avian

Реализация Avian не является на 100% Java-совместимой с точки зрения сертификации. Она включена в данный список, потому что это интересный проект с открытым исходным кодом и отличный обучающий инструмент для разработчиков, заинтересованных в понимании деталей работы JVM, а не в получении готового решения.

Android

Проект Android от Google иногда рассматривается как "основанный на Java". Однако на самом деле картина немного сложнее. Первоначально Android использовал отличную от стандартной реализацию библиотек классов Java (из проекта Harmony) и кросс-компилятор для преобразования в иной формат файла (`.dex`) для виртуальной машины, не являющейся JVM.

Из перечисленных реализаций большая часть книги посвящена HotSpot. Этот материал в равной степени относится к Oracle Java, Azul Zulu, Red Hat IcedTea и всем другим JVM, производным от OpenJDK.

Между различными реализациями на основе HotSpot, по существу, нет различий в смысле производительности.

Мы также включили в книгу некоторые материалы, относящиеся к IBM J9 и Azul Zing, что должно просто обеспечить ознакомление с этими альтернативными решениями, но не служить в качестве полного руководства. Некоторые читатели, возможно, пожелают более глубоко изучить эти технологии, и в таком случае им

предлагается перейти от жесткой установки на получение высокой производительности к обычным измерениям и сравнениям.

Android перешел на использование библиотек классов OpenJDK 8 с непосредственной поддержкой в системе времени выполнения Android. Поскольку эта технология достаточно далека от других примеров, в этой книге мы не будем рассматривать Android вообще.

Замечания о лицензиях

Почти все рассматриваемые в книге виртуальные машины являются программным обеспечением с открытым исходным кодом, и фактически большинство из них являются наследниками HotSpot с лицензией GPL. Исключениями являются IBM Open J9 с лицензией Eclipse и коммерческий продукт Azul Zing (хотя Azul Zulu имеет лицензию GPL).

Ситуация с Oracle Java (по состоянию на Java 9) немного более сложная. Несмотря на происхождение из базы кода OpenJDK, Oracle Java является собственностью компании и не является системой с открытым исходным кодом. Oracle достигает этого, заставляя всех участников OpenJDK подписать лицензионное соглашение, которое допускает двойное лицензирование их вклада — и как GPL OpenJDK, и как закрытой лицензии Oracle.

Каждый выпуск обновления Oracle Java получается как ответвление от основной линии OpenJDK, которое затем не используется для обновления для получения будущих выпусков. Это предотвращает расхождение Oracle и OpenJDK и приводит к отсутствию значимых различий между бинарными файлами Oracle JDK и OpenJDK, основанными на одном и том же источнике.

Это означает, что единственным реальным различием между Oracle JDK и OpenJDK является лицензия. Это может показаться неважным, но лицензия Oracle содержит несколько положений, о которых разработчики должны быть осведомлены.

- Oracle не предоставляет право передачи бинарных файлов за пределы вашей организации (например, как в случае Docker image).
- Не разрешается применять бинарные исправления к бинарным файлам Oracle без его согласия (которое обычно означает контракт на поддержку).

Есть также ряд других коммерческих функций и инструментов, которые Oracle делает доступными и которые работают только с Oracle JDK и в рамках соответствующей лицензии. Однако эта ситуация будет меняться в будущих версиях Java от Oracle, о чем мы поговорим в главе 15, "Java 9 и будущие версии".

При планировании новой возможности развертывания разработчики и архитекторы должны серьезно подойти к вопросу о выборе производителя JVM. Некоторые крупные организации, например Twitter и Alibaba, даже поддерживают собственные

частные сборки OpenJDK, хотя необходимые для этого инженерные усилия для многих компаний недоступны.

Мониторинг и инструментарий JVM

JVM — достаточно зрелая платформа выполнения, предоставляющая ряд альтернативных технологий для измерений, мониторинга и наблюдения за выполняемыми приложениями. Основными технологиями для выполнения этих видов действий над JVM-приложениями являются следующие.

- Java Management Extensions (JMX)
- Java agent
- The JVM Tool Interface (JVMTI)
- The Serviceability Agent (SA)

JMX представляет собой мощную, универсальную технологию управления и мониторинга виртуальных машин и приложений, запущенных на них. Она обеспечивает возможность обобщенного изменения параметров и вызова методов из клиентского приложения. К сожалению, хотя JMX (и его соответствующий сетевой транспорт, RMI) является основополагающим аспектом возможностей управления JVM, полное рассмотрение этой технологии выходит за рамки данной книги.

Агент Java — инструментальный компонент, написанный на Java (отсюда и его название), который использует интерфейсы из `java.lang.instrument` для модификации байт-кода методов. Для инсталляции Java agent используется соответствующий флаг командной строки при запуске JVM:

```
-javaagent:<путь_к_jar_агента>=<параметры>
```

JAR-файл агента должен содержать манифест и атрибут `Premain-Class`. Этот атрибут содержит имя класса агента, который должен реализовывать открытый статический метод `premain()`, который действует как регистрационная точка входа для агента Java.

Если API инструментов Java недостаточно, то можно использовать JVMTI. Это машинный интерфейс JVM, поэтому использующие его агенты должны быть написаны на компилируемом в машинный код языке — по сути, на C или C++. Этот интерфейс можно рассматривать как коммуникационный интерфейс, обеспечивающий мониторинг с использованием собственного агента JVM для отслеживания событий JVM. Чтобы установить такой агент, требуется использовать немного иную командную строку:

```
-agentlib:<имя_библиотеки_агента>=<параметры>
```

или

`-agentpath:<путь_к_агенту>=<параметры>`

Требование, чтобы агенты JVMTI были написаны на машинном коде, означает, что в этом случае гораздо проще написать код, который может повредить запущенным приложениям и даже привести к краху JVM.

Там, где это возможно, обычно предпочтительно писать агенты Java, а не код JVMTI. Писать агенты гораздо проще, но через Java API доступна не вся информация, и для доступа к этим данным применение JVMTI может оказаться единственной доступной возможностью.

Последним подходом является Serviceability Agent (агент работоспособности). Это набор API и инструментов, которые могут показывать как объекты Java, так и структуры данных HotSpot.

Serviceability Agent не требует никакого запуска кода в целевой виртуальной машине. Вместо этого HotSpot Serviceability Agent использует такие примитивы, как поиск символов и чтение памяти процесса для реализации отладочных возможностей. Serviceability Agent обладает возможностью отладки как работающих Java-процессов, так и core-файлов, также известных как *файлы аварийного дампа памяти* (crash dump files).

VisualVM

JDK наряду с хорошо известными бинарными файлами наподобие `javac` и `java` содержит ряд полезных дополнительных инструментов. Одним из таких инструментов, который часто недооценивается, является VisualVM, представляющий собой графический инструмент, основанный на платформе NetBeans.

`jvisualvm` представляет собой замену для ныне устаревшего средства `jconsole` из более ранних версий Java. Если вы все еще используете `jconsole`, то вам следует перейти на VisualVM (чтобы позволить запускать в VisualVM плагины `jconsole`, имеется специальный плагин совместимости).

Последние версии Java поставляются с вполне надежной версией VisualVM, так что версии, наличествующей в вашем JDK, обычно вполне достаточно. Однако если у вас есть необходимость в самой последней версии, можете найти ее по адресу `http://visualvm.java.net/`. После загрузки вы должны убедиться, что бинарный файл `visualvm` добавлен в ваш путь; в противном случае вы получите бинарный файл JRE по умолчанию.

Начиная с Java 9, VisualVM удален из основного дистрибутива, так что разработчикам придется загружать соответствующий бинарный файл отдельно.

Когда инструмент VisualVM запускается в первый раз, он выполняет калибровку машины, на которой запущен, поэтому в этот момент не должны быть запущены другие приложения, которые могут повлиять на производительность во время калибровки. После калибровки VisualVM завершит настройку и выведет экран-заставку. Наиболее известным для большинства пользователей представлением VisualVM является экран Monitor, который подобен показанному на рис. 2.3.

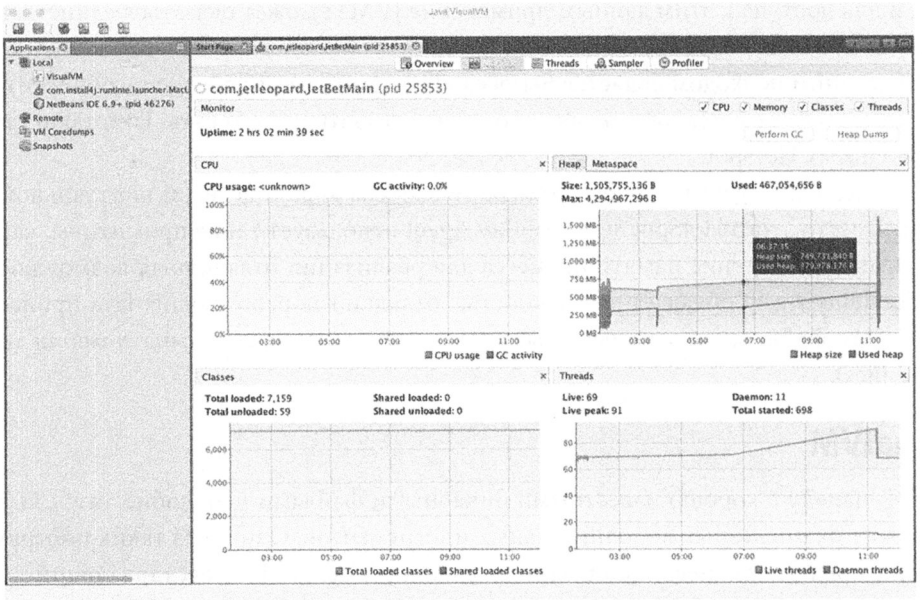

Рис. 2.3. Экран Monitor в VisualVM

Инструментарий VisualVM используется для оперативного мониторинга запущенного процесса и использует *механизм присоединения* (attach mechanism) JVM. Работа инструмента несколько различается в зависимости от того, каким является процесс — локальным или удаленным.

Локальные процессы довольно просты. VisualVM перечисляет их в левой части экрана. Двойной щелчок на одном из них приводит к его отображению в новой вкладке на панели справа.

Чтобы подключиться к удаленному процессу, удаленная сторона должна принимать входящие подключения (через JMX). Для стандартных Java-процессов это означает, что на удаленном хосте должна быть запущена программа `jstatd` (читайте более подробную информацию в руководстве по `jstatd`).

Многие серверы приложений и контейнеры исполнения обеспечивают функциональность, эквивалентную `jstatd`, непосредственно на сервере. Таким процессам не требуется отдельный процесс `jstatd`.

Чтобы подключиться к удаленному процессу, введите имя компьютера и отображаемое имя, которое будет использоваться на вкладке. По умолчанию для подключения используется порт 1099, но эту настройку можно легко изменить.

VisualVM в исходной ("из коробки") конфигурации предоставляет пользователю пять вкладок.

Overview

Предоставляет краткую информацию о ваших Java-процессах. Сюда включаются все переданные флаги и все системные свойства. Здесь также отображается версия Java.

Monitor

Это вкладка, наиболее похожая на старое представление JConsole. Она показывает высокоуровневую телеметрию JVM, в том числе использование процессора и динамической памяти. Она также показывает количество загруженных и выгруженных классов и количество работающих потоков.

Threads

Каждый поток работающего приложения отображается с использованием временной шкалы. Сюда включены как потоки приложений, так и потоки виртуальной машины. Здесь можно увидеть состояние каждого потока с небольшим количеством истории. При необходимости могут быть сгенерированы дампы потоков.

Sampler и Profiler

В этих представлениях можно получить доступ к упрощенной выборке использования процессора и памяти. Подробнее этот вопрос рассматривается в главе 13, "Профилирование".

Архитектура VisualVM с подключаемыми модулями позволяет легко добавить к основной платформе дополнительные средства для расширения функциональности. Следует упомянуть подключаемые модули, которые обеспечивают взаимодействие с консолями JMX, возможности старых средств JConsole и очень полезный модуль сборки мусора VisualGC.

Резюме

В этой главе кратко представлена анатомия JVM. Мы остановились только на некоторых из наиболее важных вопросов, и практически каждая упомянутая в этой главе тема имеет свою богатую историю, которая будет рассмотрена ниже.

В главе 3, "Аппаратное обеспечение и операционные системы", мы рассмотрим некоторые детали работы операционных систем и аппаратного обеспечения, что должно заложить необходимую основу для понимания наблюдаемых результатов в ходе анализа производительности Java. Мы также более подробно рассмотрим подсистему измерения времени как завершенный пример взаимодействия виртуальной машины и машинных (платформенных) подсистем.

ГЛАВА 3

Аппаратное обеспечение и операционные системы

Почему программистов на языке Java должно беспокоить аппаратное обеспечение?

На протяжении многих лет развитие компьютерной индустрии описывалось законом Мура, гипотезой, высказанной основателем Intel Гордоном Муром (Gordon Moore) о долгосрочных тенденциях в области возможностей процессора. Закон (в действительности — эмпирическое наблюдение или экстраполяция) может быть сформулирован по-разному, но наиболее часто он выглядит следующим образом:

> *Количество транзисторов в серийных микросхемах удваивается примерно каждые 18 месяцев.*

Это явление демонстрирует экспоненциальный рост мощности компьютеров с течением времени. Первоначально он был процитирован еще в 1965 году, так что представляет собой невероятно долгосрочную тенденцию, практически беспрецедентную в истории человеческого развития. Последствия закона Мура привели к существенным преобразованиям во многих (если не в большинстве) областях современного мира.

На протяжении десятилетий неоднократно провозглашалась смерть закона Мура. Однако есть очень веские причины полагать, что для практических целей этот невероятный прогресс в развитии технологии микросхем (наконец) пришел к своему концу.

Аппаратное обеспечение становится все более и более сложным, чтобы судить о современных компьютерах по количеству "транзисторов". Программные платформы, работающие на этом оборудовании, также демонстрируют увеличение сложности для использования новых аппаратных возможностей, так что программное обеспечение имеет в своем распоряжении гораздо больший потенциал, чем простое использование более мощной техники для увеличения производительности.

Главным результатом этого огромного увеличения производительности, доступного для разработчика обычных приложений, стали расцвет и повсеместное распространение сложного программного обеспечения. Программные приложения теперь пронизывают каждую область глобального сообщества.

Иными словами:

Программы съедают мир.

— Марк Андрессен (Marc Andreessen)

Как мы увидим, Java был одним из бенефициаров постоянно растущей вычислительной мощности. Дизайн языка и среды выполнения хорошо подходил для использования растущих возможностей процессоров. Однако действительно заботящийся о производительности программист на языке Java, чтобы как можно лучше использовать имеющиеся в его распоряжении ресурсы, должен понимать принципы и технологии, лежащие в основе используемой им платформы.

В последующих главах мы будем изучать архитектуру программного обеспечения современных JVM и методов оптимизации приложений Java на уровне платформы и кода. Но прежде чем перейти к этим темам, давайте бросим беглый взгляд на современные аппаратные средства и операционные системы, поскольку понимание этих вопросов очень поможет в понимании последующих материалов.

Введение в современное аппаратное обеспечение

Многие университетские курсы по аппаратным архитектурам до сих пор используют традиционный, устаревший дидактический подход. Используемое им представление аппаратных средств основано на простейшем представлении о регистровой машине с арифметическими и логическими операциями, а также операциями загрузки и сохранения. В результате придается чрезмерное значение языку программирования C как источнику истины, а не тому, что на самом деле делает процессор. В современную эпоху это просто мировоззренчески некорректный подход.

Начиная с 1990-х годов мир разработчиков приложений в значительной степени вращался вокруг архитектуры Intel x86/x64. Это область технологии, которая претерпела радикальные изменения, так что теперь важной частью данного ландшафта стали многие дополнительные функциональные возможности. Простая умозрительная модель работы процессора в настоящее время полностью неверна, так что основанные на ней интуитивные рассуждения способны привести к совершенно неправильным выводам.

Чтобы помочь решить эту проблему, в данной главе мы будем обсуждать некоторые из этих дополнительных функциональных возможностей, достигнутых в технологии процессоров. Мы начнем с поведения памяти, так как оно, безусловно, является наиболее важным для разработчиков современных Java-приложений.

Память

Экспоненциально растущее согласно закону Мура количество транзисторов первоначально использовалось для все большего и большего ускорения тактовой частоты процессоров. Причины этого очевидны: тактовая частота означает больше выполняемых за одну секунду команд. Соответственно, чрезвычайно выросла скорость процессоров, и сегодняшние процессоры с тактовой частотой более 2 ГГц в сотни раз быстрее процессоров на 4,77 МГц в первых IBM PC.

Однако увеличение тактовой частоты вскрыло еще одну проблему. Более быстрые процессоры требуют более быстрого потока данных, с которыми он должен работать. Как показано на рис. 3.1[1], с годами скорость работы основной памяти все больше отстает от требований процессора.

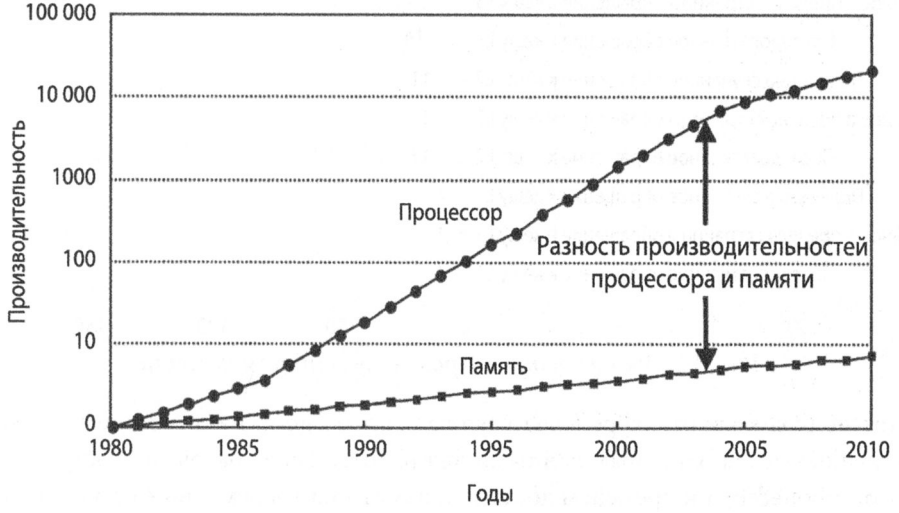

Рис. 3.1. Скорость работы памяти и количество транзисторов (Hennessy and Patterson, 2011)

В результате, если процессор находится в состоянии ожидания данных, более высокая тактовая частота не дает никакой выгоды: процессор просто будет простаивать в ожидании требуемых данных.

Кеши памяти

Для решения этой проблемы были созданы кеши процессора. Это области памяти в процессоре, которые работают медленнее, чем регистры процессора, но быстрее основной памяти. Идея заключается в том, что процессор может заполнить кеш

[1] John L. Hennessy and David A. Patterson, *From Computer Architecture: A Quantitative Approach, 5th ed.* (Burlington, MA: Morgan Kaufmann, 2011).

копиями часто используемых ячеек памяти и работать с ними, вместо того чтобы постоянно обращаться к одним и тем же ячейкам медленной основной памяти.

Современные процессоры имеют несколько уровней кеша, причем кеши с наиболее часто запрашиваемой информацией находятся ближе к ядру процессора. Ближайший к процессору кеш обычно называют *L1* (кеш уровня 1 — level 1 cache), следующий кеш именуется *L2* и т.д. Различные процессоры имеют различное количество и конфигурации кешей, но обычное решение — когда каждое ядро имеет собственные выделенные для него кеши *L1* и *L2*, а кеш *L3* совместно используется некоторыми (или всеми) ядрами. Влияние этих кешей на ускорение доступа показано на рис. 3.2[2].

Рис. 3.2. Время доступа к различным типам памяти

Этот подход к архитектуре кеша улучшает время доступа и помогает сохранить ядро полностью заполненным данными для работы. Из-за разрыва между тактовой частотой процессора и временем доступа к памяти на современных процессорах все больше транзисторов отдается на оснащение кешей.

Получающийся в результате дизайн можно увидеть на рис. 3.3. Здесь показаны кеши L1 и L2 (индивидуальные для каждого ядра процессора) и общий кеш L3, общий для всех ядер процессора. Доступ к основной памяти осуществляется через компонент северного моста и проходит через шину, что вызывает существенное ухудшение времени доступа к основной памяти.

Хотя кеширование чрезвычайно улучшает пропускную способность процессора, оно же добавляет и новые проблемы. Эти проблемы включают в себя определение того, как получать данные в кеш и записывать их обратно из кеша в память. Решения этой проблемы обычно именуются *протоколами согласованности кеша* (cache consistency protocols).

[2] Значения времени доступа указаны в количествах тактов процессора на одну операцию; информация предоставлена Google.

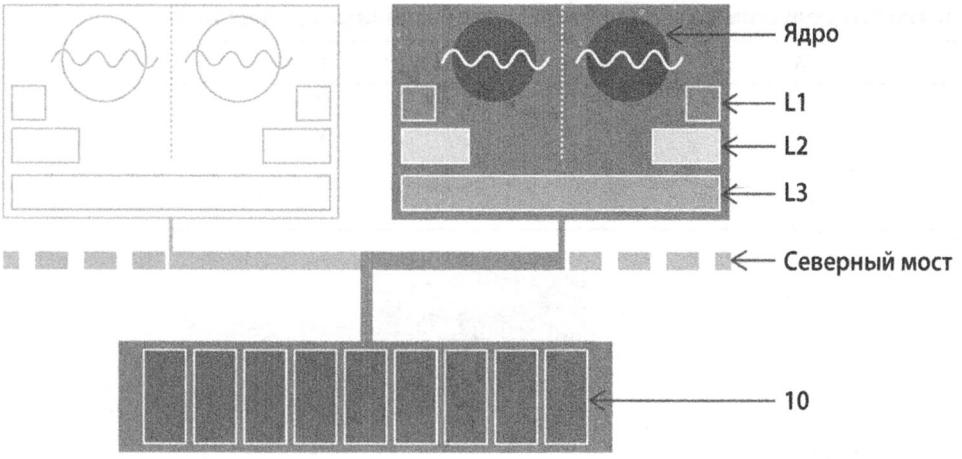

Рис. 3.3. Архитектура процессора и памяти

Как мы увидим ниже в этой книге, есть и другие проблемы, которые возникают, когда этот тип кеширования применяется в условиях параллельной обработки.

На самом низком уровне для широкого спектра процессоров используется протокол под названием "MESI" (и его варианты). Он определяет четыре состояния любой строки в кеше. Каждая строка (обычно из 64 байт) находится в одном из четырех состояний.

- **M**odified (измененная) (но еще не сброшенная в основную память).
- **E**xclusive (исключительная) (содержится только в данном кеше, но соответствует содержимому основной памяти).
- **S**hared (совместно используемая) (может находиться в нескольких кешах; соответствует содержимому основной памяти).
- **I**nvalid (недействительная) (не может использоваться; будет удалена, как только это станет иметь смысл с практической точки зрения).

Суть протокола в том, что несколько процессоров могут одновременно находиться в состоянии Shared. Однако если процессор переходит к любому другому корректному состоянию (Exclusive или Modified), то это заставляет все прочие процессоры перейти в недопустимое состояние (это показано в табл. 3.1).

Протокол работает путем широковещательного оповещения о намерении процессора изменить состояние. Электрический сигнал передается по шине общей памяти, и другие процессоры уведомляются о предстоящем изменении состояния. Полная логика переходов между состояниями показана на рис. 3.4.

Таблица 3.1. Допустимые состояния MESI для различных процессоров

	M	E	S	I
M	-	-	-	Y
E	-	-	-	Y
S	-	-	Y	Y
I	Y	Y	Y	Y

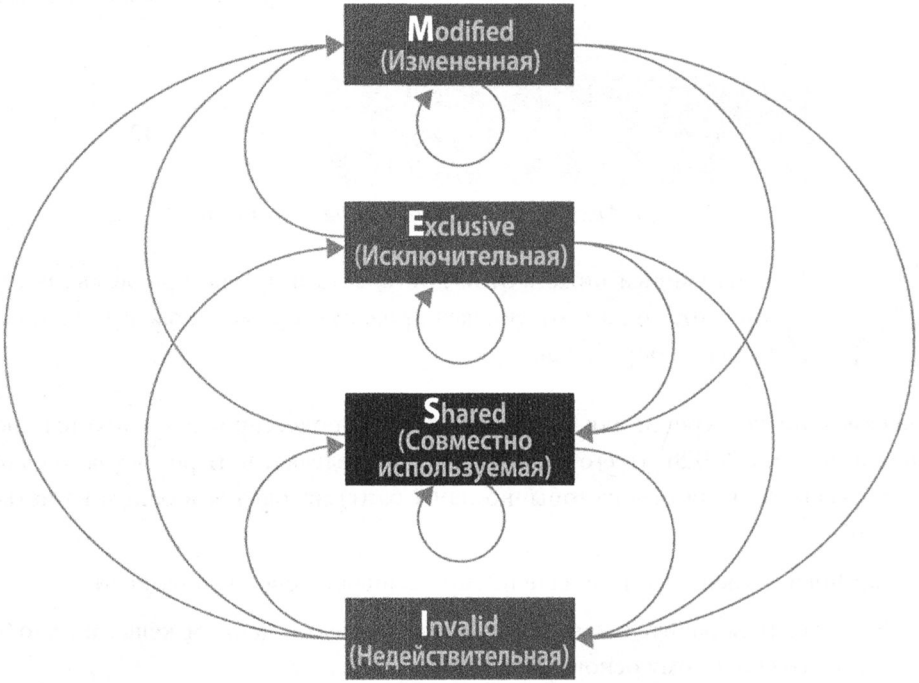

Рис. 3.4. Диаграмма переходов между состояниями MESI

Первоначально процессоры писали все операции кеша непосредственно в основную память. Это поведение называлось *сквозной записью* (write-through), но было очень неэффективно и требовало большой пропускной способности памяти. Более свежие процессоры реализуют поведение *обратной записи* (writeback), когда запись в основную память значительно уменьшается за счет того, что процессоры записывают измененные кеш-блоки в память, только когда блоки кеш-памяти заменяются другими.

Общим результатом применения технологии кеширования является значительное увеличение скорости записи информации в память и чтения из нее, выражающееся в терминах пропускной способности памяти. *Пакетная скорость* (burst rate), или теоретический максимум пропускной способности, зависит от нескольких факторов:

- тактовой частоты памяти;
- ширины шины памяти (обычно — 64 бита);
- количества интерфейсов (на современных машинах — обычно два).

Все это умножается на два в случае памяти DDR (DDR означает двойную скорость обращения к данным (double data rate), так как обращение выполняется на обоих концах тактового сигнала). Применение указанной формулы к обычному для 2015 года неспециализированному оборудованию дает теоретическую максимальную скорость записи 8–12 Гбайт/с. На практике, конечно же, она может быть ограничена многими другими факторами в системе. В существующем виде скромная польза от такой оценки заключается в возможности увидеть, насколько близко аппаратное и программное обеспечение позволяет к ней подобраться.

Напишем простой код для использования аппаратного кеша, приведенный в листинге 3.1.

Листинг 3.1. Пример использования кеша

```
public class Caching
{
    private final int ARR_SIZE = 2 * 1024 * 1024;
    private final int[] testData = new int[ARR_SIZE];
    private void run()
    {
        System.err.println("Старт: " + System.currentTimeMillis());

        for (int i = 0; i < 15_000; i++)
        {
            touchEveryLine();
            touchEveryItem();
        }

        System.err.println("Разогрев закончен: " + System.currentTimeMillis());
        System.err.println("Элемент    Строка");

        for (int i = 0; i < 100; i++)
        {
            long t0 = System.nanoTime();
            touchEveryLine();
            long t1 = System.nanoTime();
            touchEveryItem();
            long t2 = System.nanoTime();
            long elItem = t2 - t1;
            long elLine = t1 - t0;
            double diff = elItem - elLine;
            System.err.println(elItem + " " + elLine + " " +
                               (100 * diff / elLine));
```

```
        }
    }
    private void touchEveryItem()
    {
        for (int i = 0; i < testData.length; i++)
            testData[i]++;
    }
    private void touchEveryLine()
    {
        for (int i = 0; i < testData.length; i += 16)
            testData[i]++;
    }
    public static void main(String[] args)
    {
        Caching c = new Caching();
        c.run();
    }
}
```

Интуитивно метод `touchEveryItem()` должен выполнять в 16 раз больше работы, чем `touchEveryLine()`, так как должно быть обновлено в 16 раз больше элементов данных. Однако цель этого простого примера — показать, как сильно интуиция может привести в заблуждение при работе с производительностью JVM. Давайте рассмотрим пример вывода класса `Caching`, показанный на рис. 3.5.

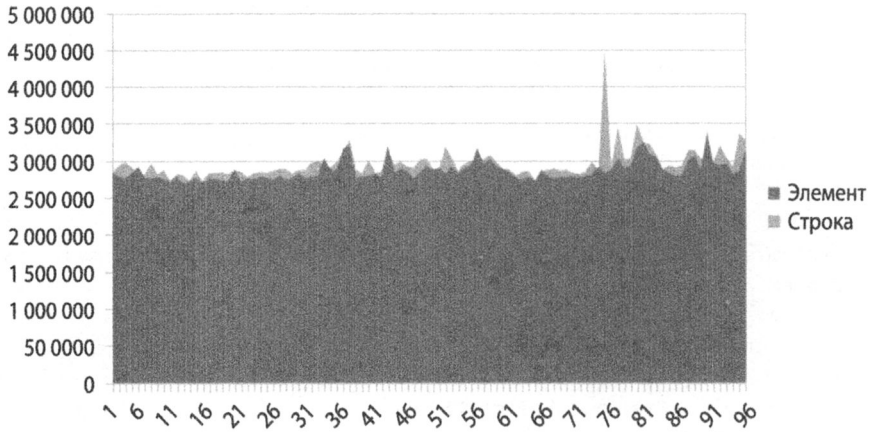

Рис. 3.5. Прошедшее время в примере `Caching`

На графике показаны результаты 100 выполнений каждой функции. Этот график призван продемонстрировать несколько различных эффектов. Во-первых, обратите внимание, что результаты для обеих функций на удивление похожи с точки зрения затраченного времени, так что интуитивно ожидаемое "в 16 раз больше работы" явно оказывается ложным.

Вместо этого доминирующее влияние на этот код оказывает шина памяти, передающая содержимое массива из основной памяти в кеш, с тем чтобы с этими данными могли работать методы `touchEveryItem()` и `touchEveryLine()`.

С точки зрения статистики, хотя результаты оказываются достаточно последовательными и согласованными, существуют отдельные выбросы, значения которых на 30–35% отличаются от среднего значения.

В целом мы можем видеть, что каждая итерация простой функции, работающей с памятью, требует около 3 мс (2,86 мс в среднем) для передачи блока памяти размером 100 Мбайт, что соответствует эффективной пропускной способности чуть менее 3,5 Гбайт/с. Это меньше теоретического максимума, но все еще разумное значение.

Современные процессоры обладают системой аппаратной предвыборки, которая может обнаружить предсказуемую схему обращения к данным (обычно это простое "перешагивание" данных). В нашем примере мы используем преимущества данного факта для того, чтобы приблизиться к реалистичному максимуму пропускной способности при обращении к памяти.

Одной из ключевых тем, связанных с производительностью Java, является чувствительность приложений к скорости выделения объектов. Мы еще не раз вернемся к этому вопросу, но данный простой пример дает нам представление о том, насколько высоким может стать эта скорость распределения.

Возможности современных процессоров

Инженеры-электронщики иногда говорят о новых функциональных возможностях процессоров, ставших возможными благодаря закону Мура, как о как "растратах транзисторного бюджета". Кеши памяти являются наиболее очевидной областью использования растущего количества транзисторов, но с течением времени появились и другие технологии.

Буфер быстрого преобразования адреса

Одним очень важным применением этих дополнительных транзисторов является еще одна разновидность кеша — буфер быстрого преобразования адреса (Translation Lookaside Buffer — TLB). Он действует как кеш таблиц страниц, которые отображают адреса виртуальной памяти на физические адреса, что значительно ускоряет частую операцию — обращение к физическому адресу, лежащему в основе виртуального адреса.

Имеется связанная с памятью программная функциональная возможность JVM, которая также обозначается аббревиатурой TLB (с ней мы познакомимся позже). Всегда помните, о чем именно идет речь, когда упоминается аббревиатура "TLB".

Без TLB поиск для каждого виртуального адреса занимает 16 тактов, даже если таблица страниц находится в кеше L1. Такая низкая производительность неприемлема, поэтому TLB обязателен для всех современных процессоров.

Предсказание ветвлений и упреждающее выполнение

Одним из методов повышения производительности, используемых современными процессорами, является предсказание ветвлений. Оно используется для предотвращения необходимости ожидания вычисления значения, необходимого для условного ветвления. Современные процессоры имеют многоступенчатые конвейеры команд. Это означает, что выполнение одного такта процессора разбивается на ряд отдельных этапов, и (на разных стадиях) могут одновременно выполняться несколько команд.

В этой модели условное ветвление проблематично, потому что до тех пор, пока условие не будет вычислено, неизвестно, какой будет следующая команда. Это может привести к приостановке процессора на определенное количество тактов (на практике — до 20), пока он при ветвлении очищает многоступенчатый конвейер.

Упреждающее выполнение оказалось причиной серьезных проблем безопасности, затронувших очень большое количество процессоров в начале 2018 года.

Во избежание этого процессор может выделить транзисторы для эвристического решения вопроса о том, какая ветвь, скорее всего, окажется выбранной. Используя это решение, процессор заполняет конвейер. Если решение было верным, то процессор продолжает работу, как будто ничего не произошло. Если же догадка оказалась неверной, частично выполненные команды сбрасываются, и процессор теряет время на опустошение конвейера.

Аппаратные модели памяти

Фундаментальным вопросом о памяти, на который должен быть дан ответ в многоядерной системе, является вопрос "Как несколько разных процессоров могут согласованно обращаться к одной и той же памяти?"

Ответ на этот вопрос очень сильно зависит от используемого аппаратного обеспечения, но в общем случае JIT-компилятор `javac` и процессор могут вносить

изменения в порядок выполнения кода — при условии, что любые изменения не влияют на результат, наблюдаемый текущим потоком.

Например, предположим, что у нас есть следующий фрагмент кода:

```
myInt = otherInt;
intChanged = true;
```

Между присваиваниями нет никакого кода, поэтому выполняющийся поток не обязан заботиться о порядке их выполнения, и, таким образом, среда выполнения может произвольно изменить порядок этих инструкций.

Однако это может означать, что в другом потоке, который видит эти элементы данных, может измениться порядок чтения, и значение `myInt`, прочитанное другим потоком, может иметь старое значение, несмотря на то что значение переменной `intChanged` видимо как `true`.

Такой тип переупорядочения (одни сохранения перемещаются и выполняются после других) невозможен на процессорах x86, но, как показано в табл. 3.2, имеются другие архитектуры, в которых такое переупорядочение может произойти[3].

Таблица 3.2. Аппаратная поддержка памяти

	ARMv7	POWER	SPARC	x86	AMD64	zSeries
Перемещение загрузок после загрузок	Y	Y	–	–	–	–
Перемещение загрузок после сохранений	Y	Y	–	–	–	–
Перемещение сохранений после сохранений	Y	Y	–	–	–	–
Перемещение сохранений после загрузок	Y	Y	Y	Y	Y	Y
Атомарное перемещение с загрузками	Y	Y	–	–	–	–
Атомарное перемещение с сохранениями	Y	Y	–	–	–	–
Не связанные команды	Y	Y	Y	Y	–	Y

В среде Java модель памяти Java (JMM) явно спроектирована как слабая модель, с учетом различий в согласованности обращений к памяти у разных типов процессоров. Правильное использование блокировок и модификатора `volatile` является основой обеспечения корректной работы многопоточного кода. Это очень важная тема, к которой мы вернемся в главе 12, "Методы повышения производительности параллельной работы".

В последние годы среди разработчиков программного обеспечения наблюдается тенденция добиваться более глубокого понимания механизмов функционирования оборудования для того, чтобы получить лучшую производительность. Для описания такого подхода Мартин Томпсон (Martin Thompson) и другие придумали особый

[3] В таблице перечисляются методы переупорядочения памяти (см., например, `https://en.wikipedia.org/wiki/Memory_ordering`). — *Примеч. пер.*

термин — *механическое взаимопонимание* (mechanical sympathy), особенно в применении к областям с низкой задержкой и высокой производительностью. Этот подход можно увидеть в недавних исследованиях алгоритмов и структур данных без блокировок, с которыми мы встретимся в конце книги.

Операционные системы

Назначение операционной системы — в управлении доступом к ресурсам, которые должны совместно использоваться несколькими выполняемыми процессами. Все ресурсы конечны, а все процессы — жадны, поэтому необходимость центральной системы арбитража и измерения доступа имеет важное значение. Среди всех этих дефицитных ресурсов есть два наиболее важных — как правило, это память и процессорное время.

Ключевой функциональной возможностью, позволяющей управлять обращениями к памяти и предотвращать повреждение памяти одного процесса другим, является виртуальная адресация с использованием блока управления памятью (memory management unit — MMU) и ее таблицы страниц.

Буфера быстрого преобразования адреса (TLB), с которыми мы уже встречались в этой главе, представляет собой аппаратное решение, которое сокращает время поиска физической памяти. Использование буферов повышает производительность доступа программного обеспечения к памяти. Однако блок управления памятью обычно представляет собой слишком низкий уровень для разработчиков, чтобы они непосредственно работали с ним или должны были быть хорошо с ним знакомы. Поэтому давайте лучше взглянем внимательнее на планировщик процессов операционной системы, поскольку он управляет доступом к процессору и является гораздо более видимой частью ядра операционной системы.

Планировщик

Доступ к центральному процессору контролируется планировщиком процессов. Он использует очередь, известную как *очередь выполнения* (run queue), в качестве зоны ожидания для потоков или процессов, которые могут выполняться, но должны пока что ожидать своей очереди на получение процессорного времени. В современной системе всегда есть больше потоков/процессов, которые хотят выполняться, чем количество потоков/процессов, которые могут выполняться, поэтому этот конфликт за процессор требует механизма его решения.

Работа планировщика состоит в том, чтобы отвечать на прерывания и управлять доступом к ядрам процессора. Жизненный цикл потока Java показан на рис. 3.6. Теоретически спецификация Java допускает потоковые модели, в соответствии с которыми потоки Java не обязательно соответствуют потокам операционной системы.

Однако на практике такие "зеленые потоки" не доказали свою полезность и в обычных операционных средах не используются.

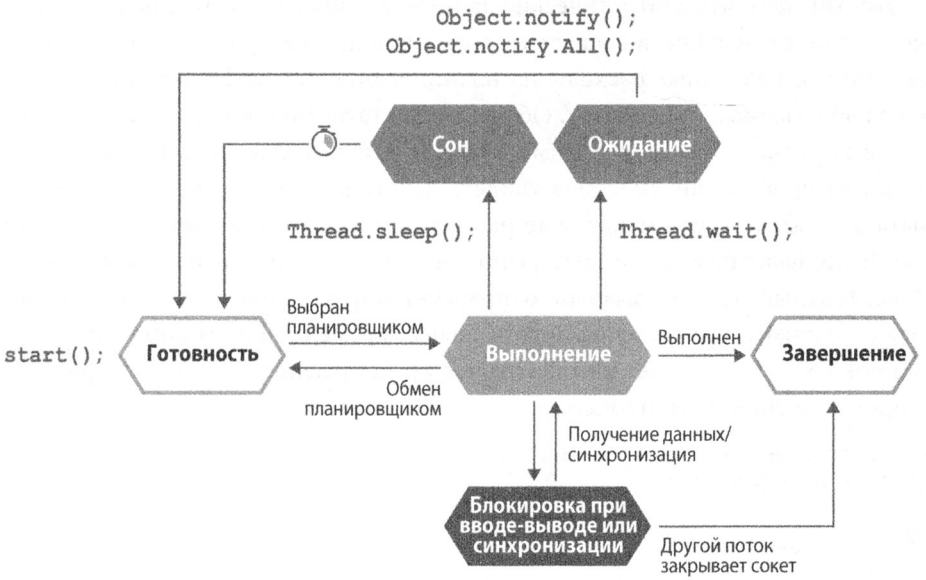

Рис. 3.6. Жизненный цикл потока

В таком относительно простом представлении планировщик операционной системы перемещает потоки в единственное ядро системы и извлекает их оттуда. В конце кванта времени (зачастую — 10 мс, или 100 мс в старых операционных системах) планировщик перемещает поток в конец очереди выполнения для ожидания, пока он не достигнет начала очереди и не получит снова право на выполнение.

Если поток хочет добровольно отказаться от своего кванта времени, это можно сделать, либо приостановив выполнение на фиксированное количество времени (с помощью вызова `sleep()`), либо до тех пор, пока не будет выполнено некоторое условие (с помощью вызова `wait()`). Наконец, поток может быть заблокирован операцией ввода-вывода или программной блокировкой.

Когда вы встречаетесь с этой моделью в первый раз, вам может помочь представление о машине как имеющей только одно ядро выполнения. Реальное оборудование, конечно же, куда более сложное, да и практически любая современная машина имеет несколько ядер, обеспечивающих истинную многозадачность, так что рассуждения о выполнении потоков в многопроцессорной среде очень сложны и зачастую контринтуитивны.

Часто забываемой особенностью операционных систем является то, что по своей природе они вводят периоды времени, когда процессор не выполняет код. Процесс, который завершил свой квант времени, не будет выполняться до тех пор, пока вновь не дойдет до начала очереди ожидания. В сочетании с тем фактом, что процессор

является дефицитным ресурсом, это означает, что код находится в состоянии ожидания большее количество времени, чем в состоянии работы.

А это означает, что статистические данные, которые мы хотим получать от процессов, в действительности зависят от поведения других процессов в системах. Эти флуктуации и накладные расходы на планирование являются основной причиной шума в наблюдаемых результатах. Обработку и статистические свойства реально наблюдаемых результатов мы обсудим в главе 5, "Микротесты и статистика".

Один из простейших способов увидеть действие и поведение планировщика — попытаться обнаружить накладные расходы операционной системы на планирование. В приведенном далее коде выполняется 2000 отдельных вызовов `sleep()` на 2 мс. Каждый из этих вызовов отправляет поток в конец очереди выполнения и должен ожидать наступление нового кванта времени. Таким образом, общее затраченное кодом время дает нам некоторое представление о накладных расходах на планирование типичного процесса:

```
long start = System.currentTimeMillis();
for (int i = 0; i < 2000; i++)
{
    Thread.sleep(2);
}
long end = System.currentTimeMillis();
System.out.println("Millis elapsed: " + (end - start) / 4000.0);
```

Выполнение этого кода дает очень сильно различающиеся результаты в зависимости от операционной системы. Большинство UNIX-систем демонстрируют накладные расходы примерно в 10–20%. В ранних версиях Windows были заведомо плохие планировщики, с накладными расходами в некоторых версиях Windows XP до 180% (так что 1000 вызовов `sleep()` на 1 мс каждый требовали до 2,8 с времени). Попадались даже сообщения, что некоторые производители вставляли в свои операционные системы специальный код для выявления запусков испытаний производительности с тем, чтобы обмануть их.

Измерение времени имеет решающее значение для измерения производительности, планирования процессов и для многих других частей стека приложений, так что давайте посмотрим, как обстоят дела с измерением времени у платформы Java (и немного больше углубимся в вопросы его поддержки виртуальными машинами Java и операционными системами).

Вопросы измерения времени

Несмотря на существование отраслевых стандартов, таких как POSIX, различные операционные системы могут демонстрировать весьма различное поведение. Например, рассмотрим функцию `os::javaTimeMillis()`. В OpenJDK она содержит вызовы конкретной операционной системы, которые фактически и выполняют

работу, и в конечном итоге предоставляют значение, которое и возвращается методом `System.currentTimeMillis()`.

Как уже обсуждалось в разделе "Многопоточность и модель памяти Java" главы 2, "Обзор JVM", поскольку этот метод опирается на функции операционной системы, в которой он работает, `os::javaTimeMillis()` должен быть реализован как машинный метод. Вот реализация этой функции в BSD Unix (например, для операционной системы Apple macOS):

```
jlong os::javaTimeMillis()
{
    timeval time;
    int status = gettimeofday(&time, NULL);
    assert(status != -1, "bsd error");
    return jlong(time.tv_sec) * 1000 + jlong(time.tv_usec / 1000);
}
```

Версии для Solaris, Linux и даже AIX невероятно похожи на версию для BSD, но код для Microsoft Windows совершенно иной:

```
jlong os::javaTimeMillis()
{
    if (UseFakeTimers)
    {
        return fake_time++;
    }
    else
    {
        FILETIME wt;
        GetSystemTimeAsFileTime(&wt);
        return windows_to_java_time(wt);
    }
}
```

Windows использует 64-битный тип `FILETIME` для хранения времени в единицах по 100 нс, прошедшее с начала 1601 года, а не структуру `timeval` Unix. В Windows также есть понятие "реальная точность" системных часов, зависящая от того, какое именно аппаратное обеспечение для измерения физического времени доступно. Таким образом, поведение вызовов для измерения времени в Java может сильно варьироваться на различных машинах с Windows.

Как мы увидим в следующем разделе, различия между операционными системами не ограничиваются измерением времени.

Переключения контекстов

Переключение контекста (context switch) представляет собой процесс, с помощью которого планировщик операционной системы удаляет выполняющийся в настоящее

время поток или задание и заменяет его находившимся в состоянии ожидания. Существует несколько различных типов переключений контекста, но в общем случае все они предполагают обмен выполняемых команд и состояния стека потока между контекстами.

Переключение контекста может быть дорогостоящей операцией, будь то переключение между пользовательскими потоками или из пользовательского режима в режим ядра (*переключение режима* — mode switch). Последний случай особенно важен, потому что потоку пользователя может понадобиться перейти в режим ядра для выполнения некоторых функций во время своего кванта времени. Однако это переключение приводит к опустошению кеша команд и других кешей, так как области памяти, к которым обращается пользовательский код, обычно не имеют ничего общего с ядром.

Переключение контекста в режим ядра делает недействительными TLB и потенциально другие кеши. При возврате из вызова эти кеши должны заполниться заново, и, таким образом, влияние переключения режима ядра сохраняется даже после того, как управление возвращается пользовательскому пространству. Это приводит к маскировке реальной стоимости системного вызова, как показано на рис. 3.7[4].

Рис. 3.7. Воздействие системных вызовов (Soares and Stumm, 2010)

Чтобы по возможности избежать этого влияния, Linux обеспечивает механизм, известный как *vDSO* (virtual dynamically shared object — виртуальный динамически совместно используемый объект). Это область памяти в пользовательском пространстве, которая применяется для ускорения системных вызовов, в действительности не требующих привилегий ядра. Таким образом, увеличение скорости достигается за счет того, что фактического переключения контекста в режим ядра не происходит.

[4] Livio Stoares and Michael Stumm, "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls," in OSDI'10, *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA: USENIX Association, 2010), 33–46.

Давайте рассмотрим пример, чтобы увидеть, как это работает в случае реального системного вызова.

Очень часто используется системный вызов Unix `gettimeofday()`, который возвращает показания часов в формате, воспринимаемом операционной системой. За сценой этот вызов на самом деле представляет собой простое чтение структуры данных ядра для получения системного времени. Поскольку побочных действий у этого вызова нет, ему на самом деле не требуется привилегированный доступ.

Если мы можем использовать vDSO для отображения этой структуры данных на адресное пространство пользовательского процесса, то переключение в режим ядра выполнять будет не нужно. В результате не придется платить снижением производительности за повторное заполнение кешей, как показано на рис. 3.7.

Учитывая, как часто большинству приложений Java требуется доступ к данным о времени, это повышение производительности стоит затраченных усилий. Механизм vDSO несколько обобщает этот пример и может быть полезной технологией, даже если она доступна только на Linux.

Простая модель системы

В этом разделе мы рассмотрим простую модель для описания основных источников проблем с производительностью. Эта модель выражена в терминах наблюдаемых основных подсистем операционной системы и может быть непосредственно связана с выводом стандартных инструментов командной строки Unix.

Модель основана на простой концепции Java-приложения, запущенного в операционной системе Unix (или Unix-подобной). На рис. 3.8 показаны основные компоненты модели, из которых она состоит.

- Аппаратное обеспечение и операционная система, выполняющие приложение.
- JVM (или контейнер), выполняющая приложение.
- Код самого приложения.
- Все внешние системы, которые вызывает приложение.
- Входящий трафик запросов к приложению.

Любой из этих аспектов системы может отвечать за проблемы с производительностью. Как мы увидим в следующем разделе, есть несколько простых методов диагностики, которые можно использовать для сужения поиска или изоляции отдельных частей системы как потенциальных источников проблем с производительностью.

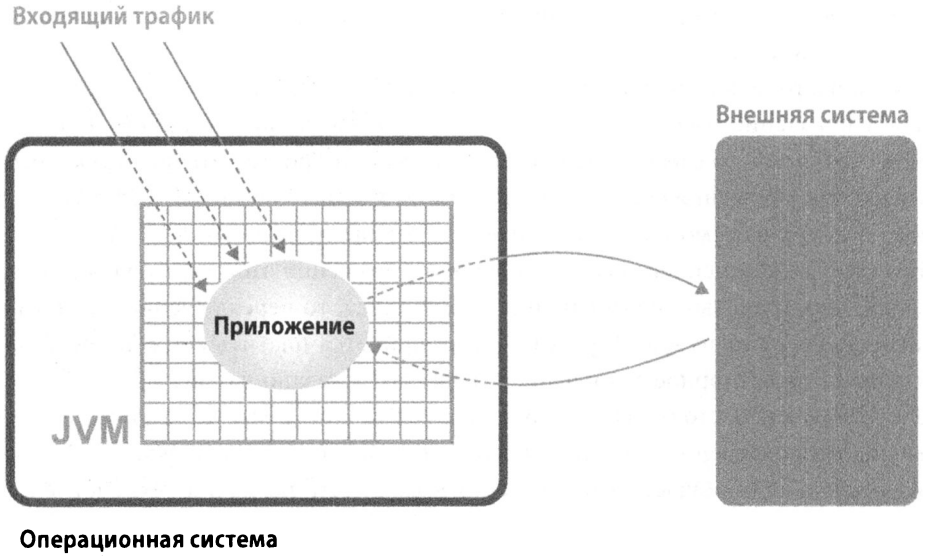

Рис. 3.8. Простая модель системы

Основные стратегии обнаружения источников проблем

Одним из определений высокопроизводительных приложений является эффективное использование системных ресурсов. Оно включает в себя использование процессора, памяти и пропускную способность сети или системы ввода-вывода.

Если приложение вызывает исчерпание одного или нескольких ресурсов, то в результате оно столкнется с проблемами производительности.

Первым шагом в любой диагностике производительности должно быть распознавание ресурса, для которого достигнут предел. Мы не сможем настроить соответствующие метрики производительности без борьбы с нехваткой ресурсов — путем повышения количества доступного ресурса или эффективности его использования.

Следует также отметить, что сама операционная система в обычной ситуации не может быть значимым фактором в использовании системы. Роль операционной системы — управление ресурсами от имени пользовательских процессов, но не их потребление. Единственным исключением из этого правила является ситуация, когда ресурсы настолько скудны, что операционная система испытывает трудности в выделении ресурсов, достаточных для удовлетворения требований пользователей. В большинстве современных систем с оборудованием серверного класса единственный случай, когда такое должно происходить, — это когда требования операций ввода-вывода (иногда — памяти) значительно превышают возможности системы.

Использование процессора

Ключевой метрикой производительности приложения является использование процессора. Такты процессора часто являются наиболее важным ресурсом, необходимым приложению, и, таким образом, их эффективное использование имеет важное значение для получения высокой производительности. Приложения должны быть ориентированы на как можно более близкое к 100% использование процессора в периоды высокой нагрузки.

В ходе анализа производительности приложения система должна быть в достаточной степени им загружена. Изучать поведение простаивающего приложения обычно не имеет смысла.

Двумя основными инструментами, о которых должен знать каждый инженер, имеющий дело с производительностью, являются `vmstat` и `iostat`. В Linux и других UNIX-системах эти инструменты командной строки обеспечивают непосредственную, часто очень полезную информацию, облегчающую понимание текущего состояния виртуальной памяти и подсистемы ввода-вывода соответственно. Эти инструменты предоставляют информацию на уровне всей машины, но этого часто достаточно, чтобы указать дальнейший путь. Давайте в качестве примера взглянем на работу `vmstat`:

```
$ vmstat 1
r b swpd   free    buff    cache si so bi  bo  in   cs  us sy  id wa st
2 0    0  759860  248412  2572248  0  0  0  80  63  127   8  0  92  0  0
2 0    0  759002  248412  2572248  0  0  0   0  55  103  12  0  88  0  0
1 0    0  758854  248412  2572248  0  0  0  80  57  116   5  1  94  0  0
3 0    0  758604  248412  2572248  0  0  0  14  65  142  10  0  90  0  0
2 0    0  758932  248412  2572248  0  0  0  96  52  100   8  0  92  0  0
2 0    0  759860  248412  2572248  0  0  0   0  60  112   3  0  97  0  0
```

Параметр `1` в командной строке `vmstat` означает, что мы хотим от `vmstat` постоянный вывод (до прерывания посредством нажатия <Ctrl+C>) вместо однократного снимка состояния. Новые строки выводятся каждую секунду, что позволяет инженеру в области производительности получать текущую информацию (или записывать ее в журнал) во время проведения начальной проверки производительности приложения.

Вывод `vmstat` сравнительно легко понять; при этом он содержит большой объем полезной информации, разделенной на следующие разделы.

1. Первые два столбца показывают количество работающих (`r`) и заблокированных (`b`) процессов.

2. В разделе, посвященном памяти, показано количество памяти подкачки и свободной памяти, после чего следует количество памяти, используемой в виде буфера и кеша.
3. В разделе памяти подкачки показаны значения количества памяти, загруженной с диска и сброшенной на диск (`si` и `so` соответственно). Современные машины серверного класса обычно не демонстрируют большой активности подкачки.
4. Количества загруженных и выведенных блоков (`bi` и `bo`) показывают число 512-байтных блоков, полученных из блочных устройств ввода-вывода и отправленных в них.
5. В системном разделе выводится количество прерываний (`in`) и переключений контекста (`cs`) в секунду.
6. Раздел процессора содержит ряд иных, непосредственно связанных с процессором метрик, выраженных в процентах времени процессора. Это время работы в режиме пользователя (`us`), в режиме ядра (`sy`), время простоя (`id`), время ожидания (`wa`) и "украденное время" (`st`, имеет смысл для виртуальных машин).

В оставшейся части этой книги мы будем встречать многие другие, более сложные инструменты, однако важно не пренебрегать и такими базовыми инструментами, имеющимися в нашем распоряжении. Сложные инструменты часто демонстрируют поведение, которое может ввести в заблуждение, в то время как простые инструменты, работающие близко к процессам и операционной системе, могут предоставить ясную и точную картину того, как в действительности ведут себя наши системы.

Давайте рассмотрим еще один пример. В разделе "Переключения контекстов" выше в этой главе мы обсуждали влияние на производительность переключения контекста и видели потенциальное воздействие полного переключения контекста в пространство ядра на рис. 3.7. Однако, выполняется ли переключение контекста между пользовательскими потоками или в пространство ядра, потери ресурсов процессора неизбежны.

Хорошо настроенная программа должна использовать свои ресурсы, в особенности процессор, по максимуму. Для задач, которые, в первую очередь, зависят от вычислений, главной целью является добиться почти 100% использования процессора.

Иначе говоря, если мы наблюдаем, что загрузка процессора далека от 100% пользовательского времени, то мы должны задаться следующим очевидным вопросом — "Почему так происходит?" Что является причиной того, что программа не в состоянии добиться полной отдачи от процессора? Проблема в непреднамеренных переключениях контекста, вызванных блокировками? В блокировках, вызванных конфликтом операций ввода-вывода?

Инструмент `vmstat` в большинстве операционных систем (особенно в Linux) в состоянии показать количество переключений контекста, так что выполнение команды `vmstat 1` позволяет аналитику увидеть результат переключений контекста в реальном времени. Процесс, который не в состоянии добиться 100% использования процессора в пользовательском режиме и демонстрирует большое количество переключений контекста, вероятно, заблокирован либо операциями ввода-вывода, либо блокировками потоков.

Однако выводимой `vmstat` информации недостаточно, чтобы полностью устранить неоднозначность. `vmstat` может помочь аналитикам обнаружить проблемы ввода-вывода, так как обеспечивает грубое представление операций ввода-вывода, но чтобы обнаружить конфликты блокировок потоков в режиме реального времени, следует использовать иные инструменты, такие как VisualVM, который может показать состояние потоков работающего процесса. Распространенным дополнительным инструментом является статистический профайлер потоков, который выполняет выборки стеков для представления блокированного кода.

Сборка мусора

Как мы увидим в главе 6, "Сборка мусора", в наиболее распространенной на сегодняшний день виртуальной машине HotSpot JVM память выделяется при запуске и управляется из пространства пользователя. Это означает, что системные вызовы (такие, как `sbrk()`) для выделения памяти не нужны. В свою очередь, это означает, что для сборки мусора достаточно минимального количества переключений в режим ядра.

Таким образом, если система демонстрирует высокий уровень системного использования процессора, то она, определенно, не тратит значительную часть времени на сборку мусора, так как сборка мусора тратит процессорное пользовательское время и не влияет на использование процессора в режиме ядра.

С другой стороны, если процесс JVM использует 100% (или около этого) процессорного времени в пространстве пользователя, то часто причиной этого является сборка мусора. Анализируя проблемы производительности, если простые инструменты (такие, как `vmstat`) показывают постоянное использование процессора на 100% и при этом почти все время потребляется пространством пользователя, мы должны спросить "Кто отвечает за такое использование процессора: JVM или пользовательский код?" Почти всегда высокое использование процессора виртуальной машиной в пользовательском пространстве вызывается подсистемой сборки мусора, поэтому полезное эмпирическое правило заключается в регулярной проверке журнала сборки мусора, чтобы посмотреть, как часто в него добавляются новые записи.

Протоколирование сборки мусора в JVM невероятно дешево, так что даже наиболее точные измерения общей стоимости не могут надежно отличить его от случайного фонового шума. Это протоколирование очень полезно как источник данных

для анализа. Поэтому важно включать протоколирование сборки мусора для всех процессов JVM, особенно для находящихся в эксплуатации.

Нам еще придется многое сказать о сборке мусора и ее протоколах позже в книге, а пока что мы хотели бы призвать читателей проконсультироваться со своим техническим персоналом и убедиться, что протоколирование сборки мусора при эксплуатации включено. Если нет, то одним из ключевых пунктов главы 7, "Вглубь сборки мусора", является построение соответствующей стратегии.

I/O

Файловый ввод-вывод традиционно является одним из самых мрачных аспектов общей производительности системы. Частично это происходит из-за его тесных отношений с неприятностями физического оборудования, которое инженеры в шутку называют "вращающейся ржавчиной" (а то и похуже), но дело еще и в том, что операциям ввода-вывода не хватает абстракций того же уровня чистоты, что и в других местах операционных систем.

В случае памяти виртуальная память отлично работает в качестве механизма разделения. Однако ввод-вывод не имеет сопоставимой абстракции, которая обеспечила бы соответствующую изоляцию разработчика приложений.

К счастью, большинство Java-программ включает только некоторые простые операции ввода-вывода, так что класс приложений с активным использованием подсистем ввода-вывода оказывается относительно небольшим. В частности, большинство приложений не пытаются "забить" каналы ввода-вывода одновременно с полной загрузкой процессора или памяти.

Сложившаяся практика привела к культуре программирования, когда разработчики заранее осведомлены об ограничениях ввода-вывода и активно контролируют процессы с интенсивным его использованием.

Что касается аналитиков/инженеров, имеющих дело с производительностью, то им достаточно иметь понимание того, как ведет себя ввод-вывод в наших приложениях. Такие инструменты, как `iostat` (и даже `vmstat`), содержат основные счетчики (например, количества читаемых и записываемых блоков), которых часто вполне достаточно для постановки предварительного диагноза, особенно при наличии в системе только одного приложения с активным вводом-выводом.

Наконец стоит отметить еще один аспект операций ввода-вывода, который все шире используется классом Java-приложений, которые активно используют операции ввода-вывода в условиях жестких требований к производительности приложения.

Обход ввода-вывода ядром

Для некоторых высокопроизводительных приложений стоимость использования ядра для копирования данных, например, из буфера сетевой карты в область

пользовательского пространства оказывается непомерно высокой. Вместо этого для отображения данных непосредственно из сетевой карты в область, доступную пользователю, применяются специализированные аппаратные и программные средства. Такой подход позволяет избежать "двойного копирования", а также пересечения границы между пространством пользователя и ядром, как показано на рис. 3.9.

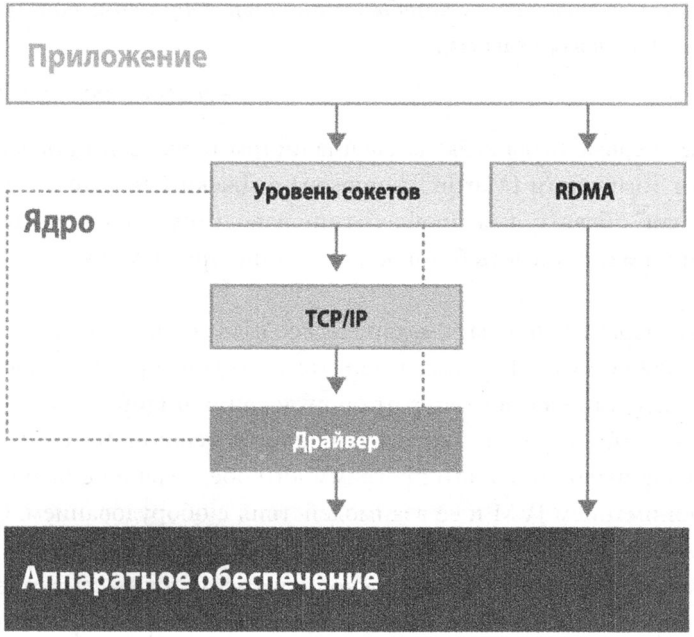

Рис. 3.9. Обход ввода-вывода ядром

Однако Java не предоставляет конкретную поддержку этой модели, и вместо этого приложения, которые хотят ее использовать, вынуждены полагаться на пользовательские (машинные) библиотеки для реализации указанной семантики. Этот полезный шаблон все чаще реализуется в системах, которые требуют очень высокой производительности операций ввода-вывода.

В некоторых отношениях это напоминает новый API ввода-вывода Java (New I/O — NIO), добавленный для того, чтобы позволить обойти кучу (heap memory) в Java и работать непосредственно с машинной памятью и базовыми операциями ввода-вывода.

В этой главе мы пока что обсуждали операционные системы, работающие поверх "голого железа". Однако все чаще системы запускаются в виртуализированных средах, так что в завершение этой главы кратко рассмотрим, как виртуализация может коренным образом изменить наше представление о производительности приложений Java.

Механическое взаимопонимание

Механическое взаимопонимание представляет собой идею о том, что понимание аппаратного оборудования имеет неоценимое значение для тех случаев, когда нужно выжать дополнительную производительность.

> *Не нужно быть инженером, чтобы стать автогонщиком, но понимание механики является обязательным.*
>
> *— Джеки Стюарт (Jackie Stewart)*

Эта цитата впервые была сказана специалистом в области производительности Java Мартином Томпсоном (Martin Thompson) о Джеки Стюарте и его автомобиле. Однако и в такой области, как проблемы производительности программного обеспечения, очень полезно иметь базовое понимание проблем, изложенных в данной главе.

Для многих разработчиков Java механическое взаимопонимание является проблемой, которую можно игнорировать. Дело в том, что JVM предоставляет уровень абстракции, отделяющий разработчика от оборудования и широкого спектра проблем производительности. Разработчики могут вполне успешно применять Java и JVM в области высокопроизводительного программного обеспечения с низкими задержками, обладая пониманием JVM и ее взаимодействия с оборудованием. Следует отметить один важный момент — JVM затрудняет рассуждения о производительности и механическом взаимопонимании, поскольку требуется рассматривать большее количество вопросов. В главе 14, "Высокопроизводительное протоколирование и обмен сообщениями", мы расскажем, как работают высокопроизводительное протоколирование и системы оповещений и как ценится при этом механическое взаимопонимание.

Давайте рассмотрим пример — поведение строк кеша.

В этой главе мы уже обсуждали преимущества процессорного кеширования. Использование строк кеша обеспечивает выборку блоков памяти. В многопоточной среде строки кеша могут привести к проблемам, если у вас есть два потока, которые пытаются одновременно выполнять чтение или запись переменной, расположенной в одной и той же строке кеша.

Когда два потока пытаются изменить одну и ту же строку кеша, возникает ситуация гонки. Первый поток делает строку кеша для второго потока недействительной, заставляя его перечитать строку из памяти. После того как второй поток выполняет эту операцию, строка кеша становится недействительной для первого потока. Это поведение в стиле пинг-понга приводит к спаду производительности, известному как *ложное совместное использование* (false sharing). Но как можно исправить ситуацию?

Механическое взаимопонимание предполагает, что сначала мы должны понять, что же происходит, а только после этого пытаться найти решение. В Java не

гарантируется схема размещения полей в объекте, а это означает, что легко найти переменные, совместно использующие одну и ту же строку кеша. Одним из способов обойти эту неприятную ситуацию могло бы быть заполнение памяти вокруг переменных, чтобы заставить их храниться в другой строке кеша. В разделе "Очереди" главы 14, "Высокопроизводительное протоколирование и обмен сообщениями", будет продемонстрирован один из способов, как можно добиться этого с помощью низкоуровневой очереди в проекте Agrona.

Виртуализация

Виртуализация может иметь разные формы, но одной из наиболее распространенных является запуск копии операционной системы как единого процесса поверх уже работающей операционной системы. Это приводит к ситуации, показанной на рис. 3.10, на котором виртуальная среда выполняется как процесс внутри невиртуализованной ("реальной") операционной системы, которая работает на "голом железе".

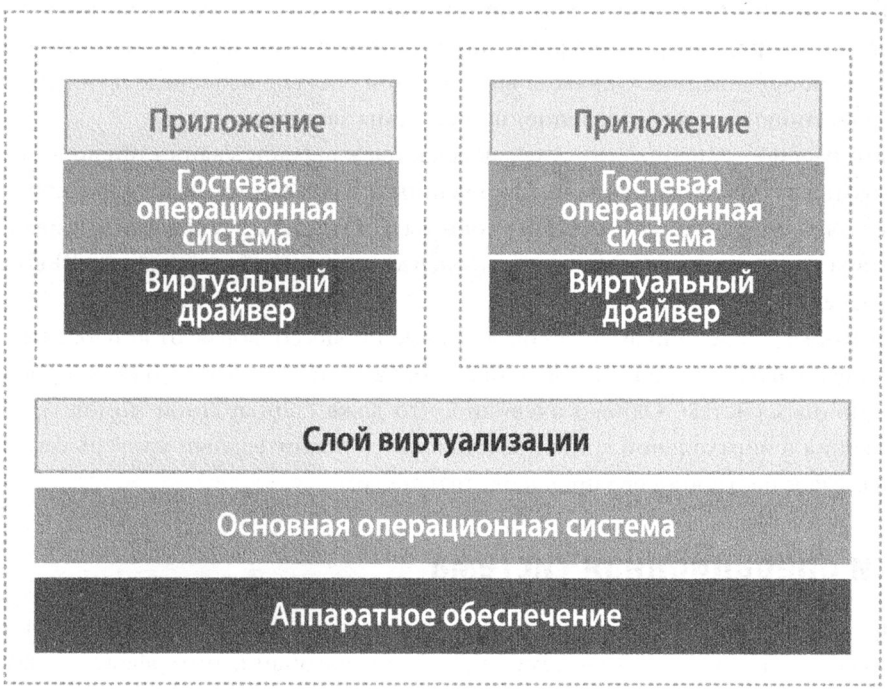

Рис. 3.10. Виртуализация операционных систем

Полное обсуждение виртуализации, соответствующей теории и ее последствий для настройки производительности приложений могут завести нас слишком далеко. Однако представляется уместным упоминание некоторых отличий виртуализации,

особенно учитывая все большее и большее количество приложений, запускаемых в виртуальных, или облачных, средах.

Хотя виртуализация была первоначально разработана в средах мейнфреймов IBM еще в 1970-х годах, она не получала распространения до недавнего времени, пока архитектура x86 не стала способной к "истинной" виртуализации. Она обычно характеризуется следующими тремя условиями.

- Программы, работающие в виртуализированной операционной системе, должны вести себя, по сути, точно так же, как и при работе на "голом железе" (т.е. невиртуализованно).
- Гипервизор должен выступать в роли посредника для всех обращений к аппаратным ресурсам.
- Накладные расходы виртуализации должны быть как можно меньшими и занимать незначительную часть времени выполнения.

В обычной невиртуализованной системе ядро операционной системы работает в особом, привилегированном режиме (отсюда вытекает необходимость переключения в режим ядра). Это дает операционной системе возможность непосредственного доступа к оборудованию. Однако в виртуальной системе непосредственный доступ к оборудованию гостевой операционной системы запрещен.

Один распространенный подход заключается в переписывании привилегированных команд в терминах непривилегированных. Кроме того, некоторые структуры данных ядра операционной системы должны обзавестись "теневыми копиями" для предотвращения излишних перегрузок содержимого кеша (например, TLB) во время переключений контекста.

Некоторые современные Intel-совместимые процессоры имеют аппаратные средства, предназначенные для повышения производительности виртуализированных операционных систем. Однако очевидно, что даже с аппаратной помощью запуск приложения в виртуальной среде представляет дополнительный уровень сложности для анализа и настройки его производительности.

JVM и операционная система

Виртуальная машина JVM обеспечивает переносимую среду выполнения, не зависящую от операционной системы, предоставляя общий интерфейс к коду Java. Однако для некоторых базовых служб, таких как планирование потоков (или даже для такого простого действия, как получение значения времени от системных часов), требуется доступ к базовой операционной системе.

Эта возможность обеспечивается машинными методами, которые помечаются с помощью ключевого слова `native`. Они написаны на C, но доступны как обычные методы Java. Этот интерфейс известен как машинный интерфейс Java (Java Native

Interface — JNI). Например, `java.lang.Object` объявляет следующие открытые машинные методы:

```
public final native Class<?> getClass();
public native int hashCode();
protected native Object clone() throws CloneNotSupportedException;
public final native void notify();
public final native void notifyAll();
public final native void wait(long timeout) throws InterruptedException;
```

Поскольку все эти методы имеют дело с относительно низкоуровневыми возможностями платформ, давайте рассмотрим более простой и знакомый пример — получение системного времени.

Рассмотрим функцию `os::javaTimeMillis()`. Это (системно-зависимый) код, отвечающий за реализацию статического метода Java `System.currentTimeMillis()`. Код, выполняющий фактическую работу, реализован на языке программирования C++, но доступ к нему из Java осуществляется с помощью "моста" из кода на языке C. Давайте посмотрим, как этот код на самом деле вызывается в HotSpot.

Как можно увидеть на рис. 3.11, метод `System.currentTimeMillis()` отображается на входную точку метода JVM `JVM_CurrentTimeMillis()`. Это отображение осуществляется с помощью JNI-механизма `Java_java_lang_System_registerNatives()`, содержащегося в `java/lang/System.c`.

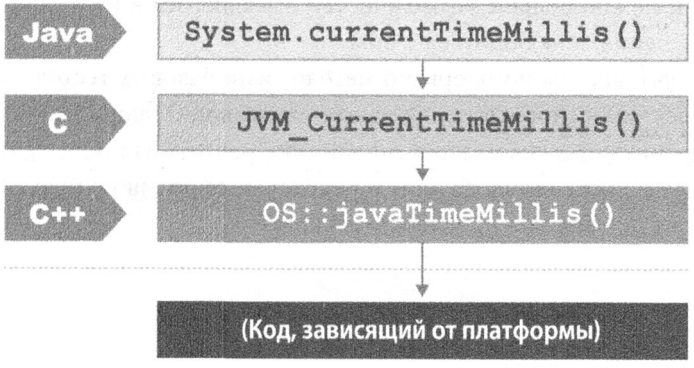

Рис. 3.11. Стек вызовов HotSpot

`JVM_CurrentTimeMillis()` представляет собой вызов метода входной точки VM. Он представляет собой функцию на языке C (в действительности — функцию на языке C++, экспортированную с соглашением о вызовах языка программирования C). Этот вызов сводится к вызову `os::javaTimeMillis()`, обернутому в пару макросов OpenJDK.

Данный метод определен в пространстве имен `os`, и неудивительно, что он зависит от конкретной операционной системы. Определения этого метода находятся в специфичных для данной операционной системы подкаталогах исходного кода

в OpenJDK. Это простая демонстрация того, как платформонезависимые части Java могут вызывать службы, которые предоставляются базовой операционной системой и аппаратным обеспечением.

Резюме

Дизайн процессоров и современное аппаратное обеспечение существенно изменились за последние 20 лет. Обусловленные законом Мура и инженерными ограничениями (в особенности относительно медленной скоростью памяти), достижения в проектировании процессоров стали слишком сложными и понятными только посвященным. Частота промахов при обращении к кешу стала наиболее очевидным ведущим показателем производительности приложения.

В области Java дизайн JVM позволяет использовать дополнительные процессорные ядра даже для однопоточного приложения. Это означает, что приложения Java получили в области производительности от развития аппаратных средств более значительные преимущества по сравнению с другими средами.

По мере прекращения выполнения закона Мура внимание будет вновь сосредоточено на относительной производительности программного обеспечения. Разработчики, заинтересованные в повышении производительности, должны понимать как минимум азы современных аппаратных средств и операционных систем, чтобы использовать как можно больше возможностей имеющегося в их распоряжении аппаратного обеспечения, а не бороться с ним.

В следующей главе мы поговорим о методологии базовых тестов производительности. Мы обсудим основные типы тестов производительности, задачи, которые необходимо решить, и в целом жизненный цикл работы над повышением производительности. Мы также каталогизируем некоторые наилучшие практики и антишаблоны в области повышения производительности.

ГЛАВА 4

Паттерны и антипаттерны тестирования производительности

Тестирование производительности предпринимается по целому ряду причин. В этой главе мы рассмотрим различные типы тестов, которые может захотеть выполнить команда программистов, и обсудим наилучшие практики для каждого из типов.

Во второй половине главы мы рассмотрим некоторые из наиболее распространенных антипаттернов, которые могут свести на нет весь тест производительности, и расскажем о подходах, позволяющих предотвратить такие проблемы.

Типы тестов производительности

Тесты производительности часто выполняются по неправильным причинам или выполняются плохо. Причины этого различны, но часто они коренятся в непонимании природы анализа производительности и вере в то, что "что-то — лучше, чем ничего". Как мы неоднократно убедимся, такая вера зачастую оказывается в лучшем случае опасной полуправдой.

Одной из наиболее распространенных ошибок является разговор о тестировании производительности "вообще", без учета конкретной специфики. На самом деле имеется много различных типов крупномасштабных тестов производительности, которые могут проводиться в системе.

Хорошие тесты производительности носят количественный характер. Они задают вопросы, на которые получают числовые ответы, которые, в свою очередь, могут быть обработаны как результат эксперимента и подвергнуты статистическому анализу.

Типы тестов производительности, которые мы будем обсуждать в этой книге, в основном имеют независимые (хотя и несколько перекрывающиеся) цели, поэтому вы должны принять это во внимание, когда рассматриваете область применимости любого отдельно взятого теста. Хорошее эмпирическое правило при планировании теста производительности — просто записать количественные вопросы, на которые

должен ответить тест, и почему они важны для тестируемого приложения (а также получить подтверждение этих вопросов от руководства/клиента).

Вот некоторые из наиболее распространенных типов тестов и примеры вопросов для каждого из них.

Тест задержки

Каково время выполнения транзакции от начала до конца?

Тест пропускной способности

С каким количеством одновременных транзакций может работать система при текущей пропускной способности?

Тест нагрузки

В состоянии ли система выдержать определенную нагрузку?

Стресс-тест

Какова "точка поломки" системы?

Тест на долговечность

Какие аномалии производительности обнаруживаются при работе системы в течение длительного периода времени?

Тест планирования нагрузки

Обладает ли система ожидаемой масштабируемостью при добавлении дополнительных ресурсов?

Деградация

Что происходит при частичном отказе системы?

Давайте поочередно рассмотрим каждый из этих типов тестов более подробно.

Тест задержки

Это один из наиболее распространенных видов тестов производительности, обычно потому что он может быть тесно связан с наблюдаемой системой, которая представляет непосредственный интерес для управления: как долго наши клиенты ждут выполнения транзакции (или загрузки страницы)? Это обоюдоострый меч, потому что количественный вопрос, на который пытается ответить тест задержки, представляется столь очевидным, что может затушевывать необходимость количественных вопросов других типов тестов производительности.

Однако даже в простейших случаях тест задержки имеет некоторые тонкости, которые следует рассматривать с осторожностью. Одной из наиболее заметных

тонкостей является то, что (как мы будем обсуждать подробно в разделе "Статистика производительности JVM" главы 5, "Микротесты и статистика") простое среднее является не очень полезным в качестве меры того, как приложение реагирует на запросы.

Цель настройки задержки обычно заключается в том, чтобы непосредственно улучшить условия работы пользователей или добиться соответствия достигнутым соглашениям об уровне обслуживания.

Тест пропускной способности

Пропускная способность, вероятно, является вторым по распространенности тестируемым количественным показателем производительности. В некотором смысле она может даже рассматриваться как эквивалент задержки.

Например, когда мы проводим тест задержки, важно указать (и контролировать) количество параллельно выполняемых транзакций при получении распределения результатов задержки.

Наблюдаемая задержка системы должна быть указана для известных и контролируемых уровней пропускной способности.

Точно так же мы обычно проводим тестирование пропускной способности во время наблюдений за задержкой. Мы определяем "максимальную пропускную способность", замечая, когда внезапно изменяется распределение задержки — по сути, обнаруживая "точку излома" (именуемую также *точкой перегиба* (inflection point)) системы. Целью стресс-теста, как мы увидим, является обнаружение таких точек и уровней нагрузки, при которых они появляются.

С другой стороны, тест пропускной способности измеряет наблюдаемую максимальную пропускную способность, демонстрируемую системой перед деградацией.

Тест нагрузки

Тест нагрузки отличается от теста пропускной способности (или стресс-теста) тем, что обычно действует как тест "да/нет": "Может ли рассматриваемая система обрабатывать прогнозируемую нагрузку или нет?" Тесты нагрузки иногда проводятся заранее, перед ожидаемыми событиями, например перед добавлением нового клиента или рынка, который, как ожидается, значительно увеличат трафик приложения. Другими примерами возможных событий, которые могут потребовать выполнения тестов этого вида, являются, например, рекламные кампании или события, связанные со средствами массовой информации.

Стресс-тест

Стресс-тест можно рассматривать как способ определить, какой запас работоспособности имеется в системе. Тест обычно выполняется путем помещения системы в установившийся режим выполнения транзакций, т.е. с определенным уровнем пропускной способности (часто — при текущей пиковой нагрузке). Затем тестируемая система постепенно нагружается параллельными транзакциями до тех пор, пока не начнет наблюдаться деградация.

Значения перед началом наблюдения деградации определяют максимальную пропускную способность, достигнутую в тесте производительности.

Тест на долговечность

Некоторые проблемы проявляются только в течение длительных периодов времени (часто измеряемых днями). К ним относятся утечки памяти, загрязнение кеша и фрагментация памяти (особенно для приложений, использующих сборщик мусора Concurrent Mark and Sweep, о чем будет сказано в разделе "CMS" главы 7, "Вглубь сборки мусора").

Чтобы обнаружить эти проблемы, обычно применяется тест на долговечность (известный также как тест на выносливость). Эти тесты выполняются при средней (или высокой) степени использования ресурсов, но в пределах обычных наблюдаемых нагрузок для системы. В ходе испытания выполняется постоянный контроль уровня ресурсов с целью обнаружения сбоев или исчерпания ресурсов.

Этот тип теста очень распространен в системах быстрого реагирования (с низкими задержками), так как очень часто для этих систем оказываются неприемлемыми приостановки выполнения, вызванные полным циклом сборки мусора (см. главу 6, "Сборка мусора", и последующие).

Тест планирования нагрузки

Тест планирования нагрузки имеет много сходства со стресс-тестом; тем не менее это отдельная разновидность теста. Роль стресс-теста заключается в том, чтобы узнать, с какой нагрузкой в состоянии справиться нынешняя система, в то время как тест планирования нагрузки более направлен на перспективу и пытается выяснить, с какой загрузкой сумеет справиться обновленная система.

По этой причине тесты планирования нагрузки часто проводятся в рамках запланированных мероприятий, а не в ответ на определенное событие или угрозы.

Тест деградации

Тест деградации известен также как испытание на частичный отказ. Общее обсуждение тестирования устойчивости и сбоев выходит за рамки этой книги, но достаточно сказать, что в наиболее регулируемых и тщательно настраиваемых средах

(включая банки и финансовые учреждения) к тестированию наработки на отказ и восстановления относятся чрезвычайно серьезно, и обычно такое тестирование планируется особенно педантично.

Для наших целей единственный тип теста устойчивости, который мы рассмотрим, — это тест на деградацию. Основной подход к этому тесту заключается в том, чтобы посмотреть, как поведет себя система при потере компонента или целой подсистемы при работе с моделируемой обычной производственной нагрузкой. Примерами могут служить кластеры серверов приложений, которые внезапно теряют своих членов, базы данных, которые внезапно теряют диски RAID, или внезапно упавшая пропускная способность сети.

Основные наблюдаемые во время теста деградации параметры включают распределение задержки транзакций и пропускную способность.

Один из особенно интересных подтипов испытания на частичный отказ известен как *Chaos Monkey*. Это название появилось после проекта, который был выполнен компанией Netflix для проверки надежности своей инфраструктуры.

Идея, лежащая в основе Chaos Monkey, заключается в том, что в действительно устойчивой архитектуре отказ одного компонента не должен вызвать каскадный сбой или оказать значимое влияние на работу всей системы в целом.

Chaos Monkey пытается продемонстрировать это с помощью прекращения работы случайным образом выбираемых процессов, которые используются в производственной среде.

Для успешной реализации соответствующей системы организация должна иметь высокий уровень "системной гигиены", проектирования служб и операционного совершенства. Тем не менее это область интереса и устремлений все большего и большего количества компаний и команд.

Примеры наилучших практик

Решая, где же сосредоточить свои усилия по настройке производительности, воспользуйтесь тремя золотыми правилами.

- Определите, что вас волнует, и подумайте, как это измерить.
- Оптимизируйте то, что нужно оптимизировать, а не то, что легко оптимизировать.
- Начинайте с игры по-крупному.

Второй совет имеет версию, которая рекомендует не попасть в ловушку придания слишком большого значения величинам, которые легко измеряются. Не каждый наблюдаемый объект имеет важное значение для конечного результата, но иногда очень соблазнительно воспользоваться результатами простых (а не правильных) измерений.

Нисходящий подход к производительности

Один из аспектов производительности Java, на который многие инженеры первоначально не обращают внимания, — это то, что выполнить крупномасштабное измерение производительности приложений Java обычно куда легче, чем пытаться получить точные значения для небольших фрагментов кода. Мы обсудим этот вопрос подробнее в главе 5, "Микротесты и статистика".

Подход, начинающийся с анализа поведения производительности всего приложения в целом, обычно называется *нисходящим* (top-down) подходом к производительности.

Для реализации большей части нисходящих подходов команда тестирования нуждается в тестовой среде, ясном понимании того, что необходимо измерять и оптимизировать, и понимании того, как работа по повышению производительности вписывается в общий жизненный цикл разработки программного обеспечения.

Создание тестовой среды

Настройка тестовой среды является одной из первых задач, которые необходимо решить большинству команд тестирования производительности. Везде, где это возможно, эта среда должна быть точной копией производственной среды во всех аспектах. Сюда включаются не только серверы приложений (серверы должны иметь такое же количество процессоров, те же версии операционной системы и системы времени выполнения Java и т.д.), но и веб-серверы, базы данных, балансировщики нагрузки, сетевые брандмауэры и т.д. В представительной среде тестирования производительности должны быть имитированы все службы (включая, например, сторонние сетевые службы, которые нелегко воспроизвести или которые не имеют возможности обработки производственной нагрузки).

Иногда команды пытаются использовать существующие среды контроля качества (QA) для тестирования производительности. Это может быть возможно для небольших сред или для разовых испытаний, но не следует недооценивать накладные расходы управления и планирования, а также материально-технические проблемы, которые может вызвать это решение.

Средам тестирования производительности, значительно отличающимся от производственных сред, которые они пытаются представлять, часто не удастся добиться результатов, обладающих какой-либо пользой или возможностью прогнозирования при работе в реальных средах.

Для традиционных сред (не использующих облачные технологии) получить тестирующую среду, имитирующую производственную, относительно просто: команда

просто покупает столько и таких физических машин, сколько их используется в рабочей среде, а затем настраивает их в точности так же, как и в производственной среде.

Менеджмент зачастую противится тратам на дополнительную инфраструктуру. Почти всегда это ложная экономия, но, к сожалению, многие организации неверно учитывают стоимость простоев. Это может привести к вере в значимость экономии от отсутствия корректной среды для тестирования производительности, поскольку не удается точно учесть риски от того, что в качестве среды тестирования производительности используется имеющаяся среда контроля качества, не отражающая реалии производственной среды.

Последние разработки, в особенности появление облачных технологий, изменили описанную традиционную картину. Автоматически масштабируемая, базирующаяся на запросах (on demand) инфраструктура означает, что все большее количество современных архитектур не вписывается в модель "купите серверы, нарисуйте диаграмму сети, установите программное обеспечения". Производственный подход, рассматривающий серверную инфраструктуру как "рабочий скот, а не домашнее животное", означает, что все большее распространение получает динамический подход к управлению инфраструктурой.

Это делает построение среды для тестирования производительности, схожей с производственной средой, потенциально более сложным. Однако при этом повышается возможность применения среды тестирования, которая, будучи не используемой, может быть выключена (это может вести к значительной экономии, но требует надлежащего процесса запуска и отключения среды тестирования в соответствии с планом).

Определение требований к производительности

Давайте вспомним простую модель системы, с которой мы встретились в разделе "Простая модель системы" главы 3, "Аппаратное обеспечение и операционные системы". Она ясно показывает, что общая производительность системы не определяется только кодом вашего приложения. Контейнер, операционная система и аппаратное обеспечение — все они играют свою роль.

Таким образом, метрики, которые мы будем использовать для оценки производительности, не должны рассматриваться только с точки зрения кода. Вместо этого мы должны рассматривать системы как единое целое, а также наблюдаемые величины, которые важны для клиентов и руководства. Они обычно называются *нефункциональными требованиями* к производительности и являются ключевыми показателями, которые мы хотим оптимизировать.

Одни цели очевидны.

- Сократить время 95% транзакций на 100 мс.
- Усовершенствовать систему так, чтобы стало возможным пятикратное увеличение пропускной способности на существующем оборудовании.
- Улучшение среднего времени отклика на 30%.

Другие могут быть менее очевидными.

- Уменьшить стоимость ресурсов для обслуживания среднего клиента на 50%.
- Гарантировать, что система удерживается в пределах 25% от целевого отклика даже при деградации кластеров приложений на 50%.
- Уменьшить отток пользователей на 25% на каждые 25 мс задержки.

Важное значение имеет открытое обсуждение с заинтересованными сторонами относительно того, что именно следует измерять и какие цели должны быть достигнуты. В идеале такое обсуждение должно стать частью первого стартового совещания, посвященного производительности.

Вопросы, специфичные для Java

Большая часть науки об анализе производительности применима к любой современной программной системе. Однако природа JVM такова, что у нее есть некоторые дополнительные осложнения, о которых следует знать и которые следует учитывать инженерам по производительности. Эти осложнения главным образом вытекают из динамических возможностей самоуправления JVM, таких как динамическая настройка областей памяти.

Одна из особенно важных особенностей Java связана с JIT-компиляцией. Современные JVM анализируют использование методов для выявления кандидатов для JIT-компиляции в оптимизированный машинный код. Это означает, что если метод не является JIT-компилированным, то справедливо одно из двух утверждений о нем:

- он недостаточно часто вызывается, чтобы оправдать компиляцию;
- он слишком большой или сложный для анализа преимуществ его компиляции.

Второе утверждение справедливо гораздо реже первого. Однако один из первых методов повышения производительности приложений на основе JVM состоит в использовании протоколирования, какие именно методы были скомпилированы, чтобы убедиться, что важные методы на ключевых путях выполнения приложения компилируются.

В главе 9, "Выполнение кода в JMV", мы обсудим JIT-компиляцию подробнее и познакомим вас с некоторыми простыми методами обеспечения того, чтобы важные методы приложений стали целевыми для JIT-компиляции в JVM.

Тестирование производительности как часть жизненного цикла разработки программного обеспечения

Некоторые компании и команды предпочитают считать тестирование производительности случайной, одноразовой деятельностью. Однако более "продвинутые" и мудрые команды, как правило, делают тесты производительности и, в частности, регрессионное тестирование производительности неотъемлемой частью жизненного цикла разработки программного обеспечения (software development lifecycle —SDLC).

Этот подход требует сотрудничества между командами разработчиков и инфраструктуры, чтобы контролировать, какие версии кода присутствуют в среде тестирования производительности в каждый момент времени. Кроме того, его практически невозможно реализовать без выделенной среды тестирования.

Обсудив некоторые из наиболее распространенных наилучших практик тестирования производительности, давайте теперь обратим внимание на ловушки и антипаттерны, жертвами которых могут стать команды.

Антипаттерны тестирования производительности

Антипаттерн представляет собой нежелательное поведение программного проекта или команды, которое наблюдается в большом количестве проектов[1]. Частота возникновения такого поведения заставляет сделать вывод (или по меньшей мере подозревать), что за это поведение отвечают некоторые фундаментальные факторы. Одни антипаттерны, на первый взгляд, могут казаться обоснованными, так как их неидеальность очевидна не сразу. Другие же являются результатом медленного накопления со временем негативных практик в проекте.

В некоторых случаях плохое поведение может быть обусловлено социальными или командными ограничениями или распространенными неверными методами управления, или простой природой человека (и разработчика). Классифицируя и категоризируя эти нежелательные особенности, мы разрабатываем "язык паттернов" для их обсуждения и надеемся, что они минуют наши проекты.

Настройка производительности всегда должна рассматриваться как весьма целенаправленный процесс с точным указанием целей на начальном этапе планирования. Это легче сказать, чем сделать: когда команда находится под давлением или работает в стесненных обстоятельствах под неразумным управлением, эти вопросы легко могут оказаться вне зоны внимания.

Многие читатели увидят ситуации, когда появляется новый клиент или запускается новая функциональная возможность и при этом происходит неожиданный

[1] Этот термин получил распространение после выхода в свет книги William J. Brown, Raphael C. Malvo, Hays W. McCormick III, and Thomas J. Malbray *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (New York: Wiley, 1998).

отказ — если повезет, то это случится во время приемного тестирования пользователями (UAT[2]), но зачастую это происходит уже в производственной версии. Тогда команде остается только не спать ночами, судорожно пытаясь найти и исправить узкое место приложения. Обычно это означает, что тестирование производительности вовсе не было проведено или что командный "ниндзя" сделал предположение, что все будет работать, и исчез (ниндзи хорошо умеют это делать).

Работающая таким образом команда оказывается жертвой антипаттернов куда чаще, чем команда, которая следует лучшим практикам тестирования производительности и открыта для дискуссий. Как и во многих других вопросах разработки, к проблемам в разрабатываемых приложениях куда чаще приводит человеческий фактор, такой как проблемы в общении, а не технические сложности.

Один интересный способ классификации предложил в своем блоге Кери Флишель (Carey Flichel): "Почему разработчики придерживаются плохого выбора технологий". В этой статье приведены пять основных причин, которые приводят к плохому выбору разработчиков. Давайте рассмотрим их по очереди.

Скука

Большинство разработчиков сталкивались с таким явлением, как скука. Для кого-то это был непродолжительный опыт, пока он искал новые интересные задачи или новую роль в компании (или в другом месте). Однако может быть и так, что новые возможности в компании отсутствуют, а покинуть ее по тем или иным причинам невозможно.

Вероятно, многие читатели сталкивались с разработчиками, которые выходят из этого затруднительного положения, возможно, путем поисков более простой жизни. Однако скучающие разработчики могут нанести вред проекту множеством способов. Например, они могут ввести в код излишнюю, ненужную сложность, например, написав алгоритм сортировки вручную вместо простого вызова `Collections.sort()`. Они могут выразить свою скуку, находя для построения компонентов технологии, которые непригодны или неизвестны, — просто чтобы иметь возможность их использовать, что приводит нас к следующему разделу.

[2] User acceptance testing (UAT — тестирование на приемлемость для пользователя) — обычно один из последних этапов процесса разработки программного обеспечения, когда приложение бесплатно передается конечным пользователям для тестирования функциональности, причем это может быть сделано в виде оценочной версии продукта для широкого круга пользователей либо для определенной фокус-группы. Цель подобного тестирования — получить от пользователей замечания и пожелания, чтобы внести в программы последние корректировки и доработки, прежде чем предложить продукт потребителю. — *Примеч. пер.*

Заполнение резюме

Иногда чрезмерное использование технологий связано не со скукой, а с желанием разработчика получить опыт работы с той или иной технологией для украшения (и повышения оценки) своего резюме. В этом случае разработчик предпринимает активные попытки увеличить свою потенциальную зарплату и конкурентоспособность, так как собирается вновь выйти на рынок труда. Маловероятно, что многие готовы покинуть хорошо функционирующую команду, но такая их деятельность может оказаться причиной выбора неверного пути развития проекта.

Последствия применения ненужных технологий — из-за скуки ли или для заполнения резюме — могут быть далекоидущими и долгоживущими и проявляться на протяжении многих лет после того, как этот разработчик ушел на более доходное место.

Давление коллег

Технические решения, выбор которых производится без озвучивания и обсуждения, зачастую оказываются худшими из возможных. Такая ситуация может проявляться в нескольких вариантах. Например, возможно, младший разработчик не хочет сделать ошибку перед лицом более опытных старших членов своей команды ("синдром самозванца"). Может быть, разработчик просто боится, что его сочтут несведущим в конкретной теме, с которой ему приходится работать. Еще одним типом давления со стороны коллег в малоконкурентных командах является желание добиться высокой скорости разработки, что приводит к экономии на всестороннем рассмотрении последствий принимаемых ключевых решений.

Недостаток понимания

Разработчики могут искать новые инструменты для решения имеющихся проблем просто потому, что не знают полного потенциала инструментов, имеющихся в их распоряжении. Часто заманчиво обратиться к компоненту с новой и интересной технологией, потому что он идеально подходит для выполнения одной конкретной задачи. Однако в связи с увеличением при этом технической сложности следует рассмотреть, что же реально могут сделать уже имеющиеся в настоящий момент инструменты и нельзя ли обойтись ими.

Например, каркас Hibernate иногда рассматривается как возможность упрощения связи между базами данных и объектами домена. Если в команде имеется лишь ограниченное понимание того, что такое Hibernate, разработчики могут принять решение о пригодности этого каркаса в своем проекте просто на том основании, что он успешно используется в другом.

Такое отсутствие понимания может привести к переусложненному использованию Hibernate и неустранимому выходу из строя конечной системы. Если же,

напротив, переписать весь уровень данных с использованием простых вызовов JDBC, это позволит разработчику оставаться на знакомой территории. Один из авторов книги преподавал курс Hibernate, придерживаясь именно этой позиции. Он пытался обучить Hibernate в достаточной степени для того, чтобы студенты могли увидеть, что если приложение может быть восстановлено, то это достигается за счет полного "выдирания" Hibernate из проекта в течение выходных — определенно, незавидное положение.

Неверно понятая/несуществующая проблема

Разработчики зачастую могут использовать технологию для решения конкретного вопроса, не выполнив при этом надлежащего исследования проблемы. Не измерив значения производительности, почти невозможно понять успех того или иного конкретного решения. Часто правильный подбор метрик производительности позволяет лучше понять проблему.

Чтобы избежать антипаттернов, важно обеспечить обмен мнениями всех участников команды о технических проблемах и активно его поощрять. Там, где что-то остается неясным, сбор фактических данных и работа над прототипами может помочь направить решения группы в нужную сторону. Технология может выглядеть очень привлекательно, но если прототип не достигает нужных показателей, то команда может принять более обоснованные решения.

Каталог антипаттернов производительности

В этом разделе мы представим краткий каталог антипаттернов производительности. Это отнюдь не исчерпывающий перечень, и, несомненно, есть многое, что еще предстоит открыть и добавить в него.

Отвлекаться на блестки

Описание

Новейшие или "крутейшие" технологии часто оказываются первой целью настройки производительности просто потому, что исследовать, как работают новые технологии, — куда более захватывающее занятие, чем копаться в устаревшем коде. Возможно также, что код, сопровождающий новую технологию, лучше описан и проще в обслуживании. Оба эти факта подталкивают разработчиков к новым компонентам приложения.

Пример комментария

Это проблемы старта (начальной фазы) — мы должны с этим разобраться.

Реальность

- Часто это просто "выстрел вслепую" вместо того, чтобы направить усилия на целевую настройку или измерения производительности приложения.
- Разработчик может не в полной мере понимать новую технологию и будет возиться с ней вместо тщательного изучения документации — зачастую создавая при этом новые проблемы.
- Что касается новых технологий, то представленные в Интернете примеры часто пригодны только для малых или образцовых наборов данных, и при их представлении нет надлежащего рассмотрения практики масштабирования приложений до размеров предприятия.

Обсуждение

Этот антипаттерн распространен в новых или малоопытных командах. Хотят ли они повысить самооценку или избежать привязки к тому, что они самонадеянно считают *устаревшими* системами, — они часто выступают за бездумное применение новейших технологий.

Поэтому логический подсознательный вывод заключается в том, что к любой проблеме производительности следует подходить, сначала внимательно рассмотрев новую технологию со всех сторон. В конце концов, новинки трудны для понимания, так что свежая пара глаз всегда будет только полезна.

Выводы

- Для определения реального узкого места системы нужны измерения.
- Следует обеспечить надлежащее протоколирование работы нового компонента.
- Следует ознакомиться как с наилучшими практиками, так и с упрощенными демонстрационными примерами.
- Следует убедиться, что команда понимает новые технологии, и установить уровень применения наилучших практик для всей команды.

Отвлечение простотой

Описание

Команда нацеливается сначала на самые простые части системы, вместо того чтобы выполнить профилирование приложения в целом и найти его объективные узкие места. В системе могут также быть части, считающиеся настолько узкоспециальными, что их может редактировать только написавший их мастер.

Примеры комментариев

Давайте начнем с частей, которые мы понимаем...

Эту часть писал Джон, а он сейчас в отпуске. Придется подождать, пока он приедет и разберется с ее производительностью.

Реальность

- Разработчик подсистемы понимает, как настраивается (только?) эта часть системы.
- Не было никакого обмена знаниями или парного программирования различных компонентов системы, все создается одиночками.

Обсуждение

Будучи дуальным антипаттерну "Отвлекаться на блестки", данный антипаттерн часто встречается в более устоявшихся командах и может больше использоваться при поддержке программного обеспечения. Если в приложении недавно была применена новая технология, команда может чувствовать себя не в своей тарелке или просто хотеть заниматься новыми системами.

В этих обстоятельствах разработчики могут чувствовать себя более комфортно, профилируя только хорошо знакомые части системы, в надежде, что они смогут достичь желаемых целей, не выходя за пределы зоны комфорта.

Особо следует отметить, что оба эти антипаттерна вызваны реакцией на неизвестное. В антипаттерне "Отвлекаться на блестки" это проявляется как желание разработчика (или команды) узнать побольше и получить преимущество — по сути, мы видим здесь наступательное поведение. Напротив, антипаттерн "Отвлечение простотой" представляет собой защитную реакцию, попытку игры на знакомом поле вместо взаимодействия с потенциально угрожающей новой технологией.

Выводы

- Для определения реального местоположения узкого места необходимы измерения.
- Если проблема заключается в незнакомом компоненте, обратитесь за помощью к экспертам в данной области.
- Убедитесь, что разработчики понимают все компоненты системы.

Мастер настройки производительности

Описание

Руководство компании увлеклось голливудским образом "одинокого гения" и наняло вписывающегося в этот стереотип хакера, который должен переходить из

одного подразделения компании в другое и исправлять все проблемы с производительностью, используя свои превосходные навыки по настройке производительности.

Конечно, существуют настоящие эксперты и компании по настройке производительности, но большинство согласится с тем, что любую проблему следует измерить и исследовать. Маловероятно, что одно и то же решение будет применимо ко всем способам использования конкретной технологии во всех ситуациях.

Пример комментария

Я уверен, что знаю, в чем тут проблема...

Реальность

- Вероятно, единственное, что сделает такой мастер-супергерой, — нарушит принятый в компании дресс-код.

Обсуждение

Этот антипаттерн может отталкивать тех разработчиков в команде, которые считают себя недостаточно хорошо подкованными для решения проблем производительности. Это важно, так как во многих случаях относительно небольшое количество оптимизации на основе показаний профайлера может привести к хорошей производительности (см. главу 13, "Профилирование").

Мы не пытаемся доказать, что не существует специалистов, которые могут помочь в случае трудностей с конкретными технологиями, но сама мысль о существовании одинокого гения, который с самого начала будет понимать все проблемы производительности, является абсурдной. Многие технологи, являющиеся экспертами в области производительности, являются специалистами в области измерений и решения проблем на основе этих измерений.

Супергерой в команде может быть очень контрпродуктивным, если он не готов делиться знаниями или подходами, предпринимаемыми им для решения конкретного вопроса.

Выводы

- Для определения реального местоположения узкого места необходимы измерения.
- Убедитесь, что любой эксперт, принятый в команду, готов делиться знаниями и умениями и действовать как часть команды.

Настройка по совету

Описание

Во время отчаянных попыток найти решение проблемы производительности в производственной версии некоторый член команды находит "магический" параметр конфигурации на веб-сайте. Безо всякого тестирования отчаявшаяся команда следует совету и применяет его в производственной версии — ну, как же, ведь производительность должна улучшиться в разы, точно так же, как в рассказе какого-то человека в Интернете...

Пример комментария

Я нашел эту подсказку в Stack Overflow. Она меняет всё.

Реальность

- Разработчик не понимает, на чем основан полученный совет или контекст его применения; его настоящее влияние на производительность тоже неизвестно.
- Возможно, этот совет работал для некоторой конкретной системы, но это не означает, он вообще даст какое-то улучшение в другой системе. На самом деле он может даже ухудшить ситуацию.

Обсуждение

Такой совет по повышению производительности представляет собой обходной путь для известной проблемы — по сути, здесь не задача ищет решение, а решение ищет задачу. Советы по повышению производительности имеют срок годности и обычно портятся со временем, так что обязательно будут встречаться советы, которые в более поздней версии программного обеспечения или платформы в лучшем случае будут бесполезны.

Одним из особенно плохих источников советов по производительности являются руководства администраторов. Они содержат общие советы, лишенные контекста. Адвокаты часто настаивают на применении таких расплывчатых рекомендаций и "рекомендуемых конфигураций" как дополнительной линии обороны в случае подачи иска.

Производительность Java имеет свой определенный контекст с большим количеством разнообразных факторов, влияющих на производительность. Если этот контекст отбросить, тогда то, что остается, почти не дает пищи для размышлений из-за сложности среды выполнения.

Кроме того, постоянное развитие платформы Java означает, что параметр, который может обеспечить повышение производительности благодаря какому-то обходному пути (трюку) в одной версии Java, не сработает в другой версии.

Например, от версии к версии часто изменяются настройки, используемые для управления алгоритмами сборки мусора. То, что работало в старых виртуальных машинах (версии 7 или 6), может просто отсутствовать в текущей версии (Java 8). Есть даже ключи командной строки, которые работают и приносят пользу в версии 7, но не позволяют запустить виртуальную машину в будущей версии 9.

Настройка конфигурации может состоять в изменении одного или двух символов, но без аккуратного управления может оказывать значительное влияние на работу в производственной среде.

Выводы

- Применяйте только проверенные и понятные методы, которые непосредственно влияют на наиболее важные аспекты системы.
- Ищите и испытывайте параметры во время тестирования продукта пользователями, но при этом пользу от вносимых изменений параметров важно доказать с профайлером в руках — как, впрочем, и для любых других изменений.
- Постоянно проводите обзоры и обсуждения конфигурации с другими разработчиками.

Козел отпущения

Описание

Некоторые компоненты постоянно считаются проблемными, даже если не имеют ничего общего с текущими проблемами.

Например, один из авторов книги в свое время столкнулся с массовыми выходами из строя при пользовательском тестировании буквально за день до выпуска окончательной версии. Определенные пути выполнения кода приводили к блокировке одной из таблиц центральной базы данных. Происходила ошибка в коде, и блокировка не снималась, приводя к тому, что остальная часть приложения становилась неработоспособной до тех пор, пока не выполнялась полная перезагрузка. В качестве уровня доступа к данным в системе использовался каркас Hibernate, который и был немедленно обвинен во всех бедах. Однако в этом случае проблема была не в Hibernate, а в пустом блоке `catch` для исключения тайм-аута, в котором не выполнялась очистка подключения к базе данных. Пришлось потратить целый рабочий день для того, чтобы заставить разработчиков прекратить обвинять Hibernate и критически взглянуть на собственный код, чтобы найти ошибку.

Пример комментария

> *Опять эта вечная проблема с этой чертовой JMS/Hibernate/еще_какой-то_библиотекой.*

Реальность

- Заключение было сделано без проведения достаточно полного анализа.
- При расследовании единственным подозреваемым был постоянный подозреваемый.
- Команда не желает взглянуть на проблему шире и установить истинную ее причину.

Обсуждение

Этот антипаттерн часто встречается в управлении или бизнесе, так как во многих случаях менеджмент не имеет полного представления о технических деталях, зато имеет когнитивное искажение в силу незнания. Однако и "технари" далеки от иммунитета к нему.

Разработчики часто становятся жертвой этого антипаттерна, когда плохо понимают базу кода или библиотеки, помимо тех, которые обычно обвиняют в проблемах. Часто гораздо проще назвать источником проблем часть приложения, которая нередко является таковой, чем провести новое объективное расследование. Это может быть признаком усталости команды, которой приходится решать многочисленные производственные вопросы.

Прекрасным примером этого является каркас Hibernate; во многих ситуациях Hibernate доходит до момента, когда он неверно настроен или неверно используется. После этого команда обычно начинает обвинять во всех грехах новую технологию, так как они уже сталкивались с проблемами с ней в прошлом. Однако проблема может с тем же успехом крыться в запросе к базе данных, использовании неподходящего индекса, физическом подключении к базе данных или в чем-то ином. Чтобы точно локализовать проблему, следует воспользоваться профайлером.

Выводы

- Не торопитесь с выводами!
- Выполняйте анализ, как обычно.
- Поделитесь результатами анализа со всеми заинтересованными сторонами (для более точного представления о причинах проблем в следующий раз).

Отсутствие общей картины

Описание

Команда становится одержимой профилированием и внесением изменений в небольшие части приложения без полной оценки влияния изменений на все приложение в целом. Инженеры начинают пробовать различные настройки JVM в попытке получить лучшую производительность, возможно, на основе имеющегося примера или другого приложения своей же компании.

Команда может также пытаться профилировать меньшие части приложения с помощью микротестов (с помощью которых, как мы узнаем в главе 5, "Микротесты и статистика", заведомо трудно получить правильные результаты).

Примеры комментариев

Если я просто изменю эти настройки, мы получим куда лучшую производительность.

Если бы мы могли ускорить всего лишь время диспетчеризации метода...

Реальность

- Команда не полностью понимает влияние вносимых изменений.
- Команда не профилирует приложение в целом с новыми настройками JVM.
- Воздействие на систему в целом по результатам микротестов не обнаружено.

Обсуждение

У JVM есть буквально сотни переключателей. Это дает нам очень гибко настраиваемую среду выполнения, но порождает при этом большое искушение использовать всю эту настраиваемость. Обычно это ошибка: настроек по умолчанию и возможности самоуправления, как правило, вполне достаточно. Некоторые из переключателей могут комбинироваться один с другим самыми неожиданными способами, что делает изменения вслепую еще более опасными. Приложения даже одной и той же компании, вероятно, будут совершенно по-разному работать и давать совершенно разные результаты профилирования, поэтому стоит потратить время на исследование параметров, которые рекомендуется использовать.

Настройка производительности представляет собой статистическую деятельность, которая опирается на весьма специфический контекст выполнения. Отсюда вытекает, что проверять производительность более крупных систем обычно легче, чем мелких, потому что в более крупных системах закон больших чисел работает в пользу инженера, нивелируя эффекты платформы, которые искажают отдельные события.

И напротив, чем больше мы пытаемся сосредоточиться на одном аспекте системы, тем сложнее разделить сложную среду, составляющую платформу (по крайней мере, в Java/C#), на отдельные подсистемы (например, потоки, сборка мусора, планирование, JIT-компиляция). Это очень трудно сделать, а обработка статистического материала — не самое часто встречающееся у программистов умение. Это приводит к очень легкому появлению чисел, которые совершенно неверно представляют поведение различных частей системы, которые разработчики полагают протестированными.

Все описанное имеет прискорбную тенденцию сочетаться с человеческим стремлением видеть определенные схемы и шаблоны даже там, где их не существует. Все это, вместе взятое, приводит к театру одного актера, в котором разработчик страстно ратует за результаты тестов производительности или за эффекты, которые его коллеги просто не в состоянии воспроизвести.

Есть несколько моментов, о которых здесь следует упомянуть. Во-первых, трудно оценить эффективность оптимизации без среды тестирования, которая полностью эмулирует производственную. Во-вторых, нет никакого смысла в оптимизации, которая помогает приложению только в ситуациях повышенных нагрузок, но снижает производительность в общем случае; однако получение наборов данных, которые являются типичными для обычного использования приложения, но при этом обеспечивают значимые результаты тестов при высоких нагрузках, может быть затруднено.

Выводы

Перед тем как вносить изменения настроек в "живую" систему, необходимо сделать следующее.

1. Измерить производительность.
2. Изменять одновременно по одной настройке во время пользовательского тестирования.
3. Убедитесь, что среда пользовательского тестирования имеет те же стрессовые нагрузки, что и производственная.
4. Убедитесь, что доступны тестовые данные, представляющие нормальную нагрузку в производственной системе.
5. Протестируйте изменение в UAT.
6. Выполните повторное тестирование в UAT.
7. Пусть ваши рассуждения проверит кто-то еще.
8. Обсудите с ним свои выводы.

Среда UAT — моя настольная машина

Описание

Среды тестирования UAT зачастую значительно отличаются от производственных, хотя и не всегда настолько, насколько ожидается. Многие разработчики вынуждены работать в обстановке, когда маломощные настольные машины используются для написания кода для мощных производственных серверов. Однако все чаще встречается и обратная ситуация — когда компьютер разработчика оказывается более мощным, чем малые производственные серверы. Низкомощные микросреды, как правило, не являются проблемой, поскольку их зачастую можно виртуализировать

так, чтобы разработчик мог иметь одну из них на своей машине. Но это не так в случае мощных производственных машин, которые часто имеют значительно больше ядер, памяти и систем эффективного ввода-вывода, чем компьютер разработчика.

Пример комментария

Полноценная среда UAT — это слишком дорого!

Реальность

- Потери из-за разницы в средах почти всегда больше цены пары компьютеров.

Обсуждение

Данный антипаттерн происходит из когнитивных искажений, отличных от тех, с которыми мы встречались ранее. В данном случае это искажение настаивает, что любое тестирование лучше, чем никакого. К сожалению, это мнение совершенно не учитывает сложную природу корпоративных сред. Чтобы была возможна сколь-нибудь значимая экстраполяция, среда UAT должна мало отличаться от производственной.

В современных адаптивных средах подсистемы времени выполнения используют имеющиеся ресурсы наилучшим образом. Если среды тестирования радикально отличаются от целевых систем, для которых предназначены разрабатываемые программы, то они будут принимать решения, отличные от решений, принимаемых в производственных системах, а это делает наши надежды на экстраполяцию в лучшем случае бесполезными.

Выводы

- Оцените стоимость выхода программного обеспечения из строя и убытки, связанные с потерей клиентов.
- Приобретите среду UAT, идентичную производственной.
- В большинстве случаев первая стоимость намного превосходит вторую, что должно натолкнуть менеджеров на принятие правильного финансового решения.

Сложность получения реальных данных

Описание

Известный также как "Облегченные данные", этот антипаттерн относится к мало распространенным ошибкам, с которыми люди сталкиваются при попытке представить реальные производственные данные. Рассмотрим обработку фьючерсных и опционных сделок в крупном банке. Обычно такая система должна обрабатывать

миллионы сообщений в день. Теперь рассмотрим следующие стратегии UAT и связанные с ними потенциальные проблемы.

1. Для упрощения тестирования создается механизм для создания небольшой выборки из сообщений, поступающих в течение дня. Затем все захваченные сообщения пропускаются через систему UAT.

   Этот подход не позволяет имитировать взрывоподобное поведение, с которым может столкнуться система при реальной работе. Он также не может моделировать "перегрев", вызванный большим количеством фьючерсных сделок на конкретном рынке до открытия другого рынка, торгующего опционами.

2. Для упрощения тестирования используются только простые данные для сопоставления фьючерсов и опционов.

   Так мы оказываемся очень далекими от реальных производственных данных. В ситуации использования внешней библиотеки или системы для опционов наличие нашего множества данных для UAT не позволяет выяснить, что именно вызывает проблемы с производительностью, так как диапазон вычислений выполнятся над крайне упрощенным подмножеством производственных данных.

3. Для упрощения все значения передаются через систему за один раз. Это частая практика в UAT, но при этом она не позволяет протестировать возможные проблемы, связанные с передачей данных с разной скоростью.

Большую часть времени набор данных для тестирования в UAT оказывается упрощенным, но это редко приводит к получению *полезных* результатов.

Примеры комментариев

Слишком сложно синхронизировать производство и тестирование.

Слишком сложно управлять данными так, чтобы они соответствовали тому, что ожидает систему.

Производственные данные закрыты для разработчиков из соображений секретности. Разработчики не должны их видеть!

Реальность

Данные при тестировании UAT для получения точных результатов должны быть такими же, как и при работе в реальной производственной среде. Если данные недоступны по соображениям безопасности, то их следует зашифровать (замаскировать, спрятать их суть), но использовать именно их для значимых тестов. Другим вариантом является разделение UAT так, чтобы разработчики не видели данные, но могли видеть результаты тестов производительности, чтобы иметь возможность выявлять проблемы.

Обсуждение

Этот антипаттерн также попадает в ловушку "хоть что-то — лучше, чем ничего". Идея антипаттерна в том, что тестирование пусть даже с устаревшими и непредставительными данными все же лучше, чем отсутствие тестирования вовсе.

Как и ранее, это чрезвычайно опасная линия рассуждений. Хотя тестирование *чего-то* (даже если это что-то не имеет ничего общего с производственными данными) и в состоянии выявить недостатки и упущения при тестировании системы, оно предоставляет ложное чувство безопасности.

Когда система выходит в самостоятельное плавание и реальное ее использование не соответствует тем ожидаемым нормам, которые были представлены в данных для UAT, команды разработчиков могут обнаружить, что они напрасно успокоились благодаря результатам UAT и теперь совершенно не готовы к тем ужасам, которые быстро обнаруживаются при переходе к реальным производственным масштабам.

Выводы

- Проконсультируйтесь с экспертами по данным в предметной области и инвестируйте средства и усилия в процесс передачи производственных данных для UAT с шифрованием или запутыванием данных для безопасности в случае необходимости.
- Приготовьтесь к необходимости выпуска продукта, для которого ожидается большее количество клиентов или сделок.

Когнитивные искажения и тестирование производительности

Люди обычно плохо реагируют на необходимость быстрого формирования точного мнения даже тогда, когда, столкнувшись с проблемой, могут опираться на опыт прошлого и подобные ситуации.

Когнитивное искажение (cognitive bias) — это психологический эффект, который заставляет человеческий мозг делать неправильные выводы. Это особенно проблематично потому, что человек, испытывающий такое искажение, обычно об этом не подозревает и считает, что он совершенно прав.

Многие из рассмотренных в этой главе антипаттернов вызваны, целиком или частично, одним или несколькими когнитивными искажениями, которые, в свою очередь, основаны на неосознанных предположениях.

Например, в случае антипаттерна "Козел отпущения", если компонент вызвал несколько последних отказов, то команда разработчиков может предвзято считать, что

именно этот компонент является причиной всех новых проблем производительности. Любые анализируемые данные могут рассматриваться как более достоверные, если подтверждают мысль о том, что во всем виноват этот компонент-козел. Антипаттерн сочетает в себе такие когнитивные искажения, как искажение подтверждения и искажение новизны (тенденция полагать, что все, что происходило в последнее время, будет продолжать происходить).

Один и тот же компонент в Java может вести себя по-разному от приложения к приложению в зависимости от того, как он оптимизирован во время выполнения. Для того чтобы устранить любые существующие когнитивные искажения, важно взглянуть на приложение в целом.

Искажения могут быть взаимодополняющими или дуальными одно к другому. Например, некоторые разработчики могут предвзято предполагать, что проблема кроется не в программном обеспечении, а причиной всех бед является инфраструктура, в которой оно работает. Это часто встречается в антипаттерне "Работай на меня", который характеризуется заявлениями наподобие "Все отлично работало во время UAT, так что проблема в производственной среде". Оборотной стороной этой медали является мнение, что любая проблема вызвана исключительно программным обеспечением, потому что это та часть системы, которую разработчик хорошо знает и на которую может непосредственно влиять.

Упрощенное мышление

Это когнитивное искажение основано на аналитическом подходе, который утверждает, что, разделив систему на достаточно мелкие фрагменты, вы сможете понять ее, разобравшись в ее составных частях. Понимание каждой части означает уменьшение вероятности неправильных предположений.

Проблема этой точки зрения в том, что в сложных системах это просто неверно. Нетривиальные системы программного (или физического) обеспечения очень часто демонстрируют неожиданное поведение, так как целое в данном случае — это нечто большее, чем простая сумма составных частей.

Искажение подтверждения

Искажение подтверждения может привести к значительным проблемам, когда речь заходит о тестировании производительности или попытке субъективного взгляда на приложение. Обычно искажение подтверждения вносится непреднамеренно, при выборе плохого проверочного набора данных или при отсутствии надежного статистически достоверного анализа результатов теста. Искажению подтверждения довольно трудно противостоять из-за наличия сильных мотивационных или

эмоциональных факторов (таких, как неординарная личность в команде, пытающаяся навязать свою точку зрения).

Рассмотрим такой антипаттерн, как "Отвлекаться на блестки", когда член команды стремится привнести в проект новейшую и самую лучшую базу данных NoSQL. Команда выполняет несколько тестов с данными, которые не являются реальными производственными данными — просто потому, что предоставление полной схемы данных оказывается слишком сложным для оценки. Они быстро доказывают, что на тестовом наборе данных база данных NoSQL демонстрирует превосходное время доступа на локальном компьютере команды. Увидев результаты, команда с энтузиазмом берется за реализацию приложения с новой базой данных. Здесь мы видим сразу несколько антипаттернов.

Надувательство с переключателями

"Настройка по совету" и "Отсутствие общей картины" являются примерами антипаттернов, которые (по крайней мере, отчасти) вызваны сочетанием упрощенчества и искажения подтверждения. Одним из особенно вопиющих примеров является подтип антипаттерна "Настройка по совету", известный как *Надувательство с переключателями*.

Этот антипаттерн возникает потому, что, хотя виртуальная машина и пытается выбрать настройки, подходящие для обнаруженного оборудования, существуют некоторые ситуации, в которых инженеру нужно вручную задать флаги для настройки производительности кода. Само по себе это не вредно, но здесь есть скрытая когнитивная ловушка — в чрезвычайно настраиваемой с помощью командной строки природе JVM.

Чтобы увидеть весь список флагов виртуальной машины, воспользуйтесь следующим переключателем:

```
-XX:+PrintFlagsFinal
```

В версии Java 8u131 вы получите свыше 700 возможных переключателей. Но это не все; есть также дополнительные переключатели настройки, доступные только тогда, когда виртуальная машина работает в режиме диагностики. Чтобы увидеть их, добавьте следующий переключатель:

```
-XX:+UnlockDiagnosticVMOptions
```

Это разблокирует еще около 100 переключателей. Совершенно невозможно, чтобы человек мог правильно предсказать совокупный эффект применения возможных комбинаций этих переключателей. Кроме того, в большинстве случаев экспериментальные наблюдения покажут, что эффект от изменения значений переключателя очень мал — часто гораздо меньше, чем ожидают разработчики.

Туман войны (искажение действий)

Это искажение обычно проявляется во время сбоев или ситуаций, в которых система не работает так, как от нее ожидалось. Среди наиболее распространенных причин следующие.

- Изменения в инфраструктуре, в которой работает система; возможно, без уведомления или понимания влияния изменений.
- Изменения библиотек, от которых зависит система.
- Странная ошибка или состояние гонки, обнаруженное в самый загруженный день года.

В приложении с высоким уровнем поддержки, достаточным протоколированием и мониторингом должны создаваться точные сообщения, которые приведут службу поддержки к причинам проблем.

Однако слишком много приложений не тестируются на поведение во время сбоев и в них отсутствует соответствующее протоколирование. В этих обстоятельствах даже опытные инженеры могут попасть в ловушку — и над приложением опускается "туман войны".

Если члены команды не подходят к возникающим проблемам систематически, в игру могут вступать многие человеческие факторы, обсуждавшиеся ранее в этой главе. Например, такой антипаттерн, как "Козел отпущения", может резко сократить проводимое исследование проблемы и повести команду по краткому пути — на котором нет и не может быть полного представления о картине происходящего. Кроме того, у команды может появиться соблазн разделить систему на составные части и исследовать код на низком уровне без предварительного выяснения, в какой из подсистем в действительности имеются проблемы. Там, где в дело вмешиваются человеческие эмоции, может быть очень трудно принять правильное решение, в особенности в напряженной ситуации выхода приложения из строя.

Искажение риска

Люди по своей природе не склонны к риску и стараются избегать изменений. В основном это связано с тем, что каждый встречал массу примеров того, как изменения приводили к худшему результату, так что в попытках избежать этого риска люди сопротивляются любым изменениям. Однако это очень разочаровывает, когда речь идет о малых, просчитанных рисках для развития продукта. Можно значительно снизить искажение риска, имея широкий набор модульных и регрессионных тестов. Если любой из этих тестов не заслуживает доверия, изменение становится чрезвычайно трудным, а фактор риска не контролируется.

Это искажение часто проявляется в неспособности исследовать проблемы приложения (даже полный выход из строя) и принять соответствующие меры для их устранения.

Парадокс Эллсберга

В качестве примера того, насколько плохо люди воспринимают вероятности, рассмотрим парадокс Эллсберга (он назван в честь известного американского журналиста Даниэля Эллсберга (Daniel Ellsberg)), который связан с тягой человека к "известным неизвестным" по сравнению с "неизвестным неизвестным".

Обычная формулировка парадокса Эллсберга состоит в проведении простого мысленного эксперимента, посвященного вероятности. Рассмотрим мешок, содержащий 90 цветных шариков. Известно, что 30 из них — синего цвета, а остальные являются либо красными, либо зелеными. Точное соотношение количества красных и зеленых шариков неизвестно, но мешок, шарики, а следовательно, и все вероятности остаются фиксированными на протяжении всего эксперимента.

Первый шаг парадокса состоит в выборе ставки. Игрок может выбрать любую из двух ставок.

А.Игрок выиграет 100 долларов, если выбранный случайным образом шар — синий.

Б.Игрок выиграет 100 долларов, если выбранный случайным образом шар — красный.

Большинство людей выбирают первый вариант, так как при этом точно известны шансы на победу: вероятность выигрыша составляет ровно 1/3. Однако (при условии, что выбранный мяч помещается обратно в мешок и мешок встряхивается), если игроку предлагается другой вариант выбора, происходит нечто удивительное. В этом случае вариантами выбора являются следующие.

В.Игрок выиграет 100 долларов, если выбранный случайным образом шар — синий или зеленый.

Г.Игрок выиграет 100 долларов, если выбранный случайным образом шар — красный или зеленый.

В этом случае второй выбор соответствует известным шансам (вероятность победы — 2/3), так что почти все выбирают именно этот вариант.

Парадокс в том, что набор вариантов А и Г иррационален. Выбор А неявно выражает мнение о распределении красных и зеленых шариков — о том, что зеленых шариков больше, чем красных. Таким образом, если уж выбирать ставку А, то тогда во втором варианте логичнее выбирать ставку В — так как это даст больше шансов, чем безопасный выбор Г.

Резюме

Оценивая результаты производительности, важно обрабатывать данные надлежащим образом, не впадая в ненаучное или субъективное мышление. В этой главе мы встретили некоторые разновидности тестов, лучшие практики тестирования и антипаттерны, присущие анализу производительности.

В следующей главе мы перейдем к рассмотрению измерений производительности на низком уровне, познакомимся с ловушками микротестов и с некоторыми статистическими методами обработки "сырых" результатов, полученных при измерениях JVM.

ГЛАВА 5

Микротесты и статистика

В этой главе мы рассмотрим особенности непосредственного измерения показателей производительности Java. Динамическая природа JVM означает, что показатели производительности часто сложнее в обработке, чем ожидают многие разработчики. В результате в Интернете встречается множество неточных или вводящих в заблуждение показателей производительности.

Основная цель настоящей главы — ознакомить вас со всеми возможными ловушками, чтобы вы оперировали только теми показателями, на которые можно положиться. В частности, выполнение измерений для небольших фрагментов кода Java (микротестирование (microbenchmarking)) является крайне тонким и трудным делом, которое сложно выполнить правильно, так что микротестирование и его надлежащее использование инженерами по производительности является основной темой всей данной главы.

> *Первый принцип заключается в том, что вы должны не дурачить самих себя, — а себя одурачить легче всего.*
>
> *— Ричард Фейнман (Richard Feynman)*

Во второй части главы описано использование золотого стандарта инструментария микротестирования: JMH. Если (даже после всех предупреждений и предостережений) вы действительно чувствуете, что ваши приложения и сценарии использования требуют применения микротестов, то вы избежите многочисленных хорошо известных ловушек и капканов, начав с самых надежных и передовых из имеющихся инструментов.

Наконец мы обратимся к теме статистики. JVM обычно генерирует ряд показателей, которые требуют бережного обращения. Числовые показатели микротестов обычно особенно чувствительны ко всем воздействиям, поэтому на плечи инженера по производительности ложится задача трактовки наблюдаемых результатов с применением статистической обработки. В последних разделах данной главы поясняются некоторые из методов работы с данными о производительности JVM и проблемы интерпретации данных.

Введение в измерение производительности Java

В разделе "Обзор производительности Java" главы 1, "Оптимизация и производительность", мы описали анализ производительности как дисциплину, которая, по сути, является экспериментальной наукой. То есть, если мы хотим написать хороший тест (или микротест), может быть очень полезно рассматривать его как научный эксперимент.

Этот подход приводит нас к представлению теста производительности в виде "черного ящика", который имеет входы и выходы, и мы хотим собрать данные, на основании которых сможем выдвинуть гипотезы или вывести результаты. Однако мы должны быть осторожны — недостаточно просто *собрать* данные. Мы должны не стать *обманутыми* собранными нами данными.

> *Результаты тестов сами по себе не имеют значения. Важно, какие модели вы выведете из полученных числовых данных.*
>
> *— Алексей Шипилёв (Aleksey Shipilëv)*

Поэтому в идеале наша цель — сделать тест производительности точным, чтобы по возможности мы изменяли только один аспект системы, и обеспечить контроль над всеми прочими внешними факторами. В идеальном мире все остальные аспекты системы должны оставаться полностью неизменными от испытания к испытанию, но на практике такое везение встречается крайне редко.

Даже если цель научно чистого справедливого теста недостижима на практике, важно, чтобы наши тесты были по крайней мере повторяемыми, поскольку это является основой любых эмпирических результатов.

Одной из центральных проблем написания тестов производительности для платформы Java является сложность системы времени выполнения Java. Значительная часть этой книги посвящена автоматической оптимизации, применяемой к коду разработчика со стороны JVM. Когда мы рассматриваем наши тесты производительности как научное исследование в контексте этих оптимизаций, варианты наших действий сразу становятся ограниченными.

Иначе говоря, полностью понять и учесть точное воздействие этих оптимизаций невозможно. Точные модели "реальной" производительности кода нашего приложения трудны в создании и крайне ограничены в смысле применимости.

Другими словами, в действительности мы не можем отделить выполнение кода Java от JIT-компилятора, управления памятью и других подсистем, предоставляемых средой выполнения Java. Мы не можем игнорировать влияние операционной системы, оборудования или условий выполнения тестов (например, загруженности компьютера).

Ни один человек не является самодостаточным островом.

— Джон Донн (John Donne)

Сгладить все эти эффекты проще, имея дело с большей совокупностью (всей системы или подсистемы). И наоборот, когда мы имеем дело с маломасштабными или микротестами, надежно изолировать код приложения от фонового поведения среды выполнения гораздо более трудно. Это основная причина, по которой микротесты так трудны.

Давайте рассмотрим некий, как представляется, очень простой пример — тест производительности кода, который сортирует список из 100 000 чисел. Мы хотим проанализировать его с точки зрения создания по-настоящему точного теста производительности.

```
public class ClassicSort
{
    private static final int N =   1_000;
    private static final int I = 150_000;
    private static final List<Integer> testData = new ArrayList<>();
    public static void main(String[] args)
    {
        Random randomGenerator = new Random();

        for (int i = 0; i < N; i++)
        {
            testData.add(randomGenerator.nextInt(Integer.MAX_VALUE));
        }

        System.out.println("Тестирование алгоритма сортировки");
        double startTime = System.nanoTime();

        for (int i = 0; i < I; i++)
        {
            List<Integer> copy = new ArrayList<Integer>(testData);
            Collections.sort(copy);
        }

        double endTime = System.nanoTime();
        double timePerOperation = ((endTime - startTime) /
                                   (1_000_000_000L * I));
        System.out.println("Результат: " + (1 / timePerOperation)
                                         + " операций/с");
    }
}
```

Тест создает массив случайных целых чисел и, завершив эту часть работы, регистрирует время начала теста. Затем тест производительности циклически копирует массив и выполняет сортировку данных. После выполнения `I` итераций время

выполнения преобразуется в секунды и делится на количество итераций, чтобы дать время, затрачиваемое на одну операцию.

Первый вопрос, встающий по поводу теста производительности, заключается в том, что он тут же переходит в тестирование кода, без каких бы то ни было соображений по поводу "разогрева" JVM. Рассмотрим случай, когда сортировка выполняется в серверном производственном приложении, которое, скорее всего, работает на протяжении часов, а возможно, даже дней. Как мы знаем, JVM включает JIT-компилятор, который преобразует интерпретированный байт-код в высокооптимизированный машинный код. Но компилятор делает это только после выполнения метода определенное количество раз.

Поэтому приведенный тест производительности не является представительным в отношении того, как код будет вести себя в производственном приложении. При попытке проведения теста JVM будет тратить время на оптимизацию вызова. Этот эффект можно увидеть, запустив сортировку с определенными флагами JVM:

```
java -Xms2048m -Xmx2048m -XX:+PrintCompilation ClassicSort
```

Флаги `-Xms` и `-Xmx` управляют размером кучи, в данном случае закрепляя ее размер равным 2 Гбайтам. Флаг `PrintCompilation` приводит к выводу строки в журнал при компиляции метода (или некоторого иного события компиляции). Вот фрагмент вывода:

```
Testing Sort Algorithm
73       29   3 java.util.ArrayList::ensureExplicitCapacity (26 bytes)
73       31   3 java.lang.Integer::valueOf (32 bytes)
74       32   3 java.util.concurrent.atomic.AtomicLong::get (5 bytes)
74       33   3 java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
74       35   3 java.util.Random::next (47 bytes)
74       36   3 java.lang.Integer::compareTo (9 bytes)
74       38   3 java.lang.Integer::compare (20 bytes)
74       37   3 java.lang.Integer::compareTo (12 bytes)
74       39   4 java.lang.Integer::compareTo (9 bytes)
75       36   3 java.lang.Integer::compareTo (9 bytes) made not entrant
76       40   3 java.util.ComparableTimSort::binarySort (223 bytes)
77       41   3 java.util.ComparableTimSort::mergeLo (656 bytes)
79       42   3 java.util.ComparableTimSort::countRunAndMakeAscending (123 bytes)
79       45   3 java.util.ComparableTimSort::gallopRight (327 bytes)
80       43   3 java.util.ComparableTimSort::pushRun (31 bytes)
```

JIT-компилятор выполняет дополнительную работу по оптимизации частей иерархии вызовов, чтобы сделать код более эффективным. Это означает, что производительность во время тестирования изменяется, и мы случайно оставили неконтролируемый фактор в нашем эксперименте. Поэтому желательно добавление периода "прогрева", что позволит JVM войти в устойчивый режим работы, прежде чем мы приступим к измерениям. Обычно это включает некоторое количество итераций кода, который мы хотим замерять, без выполнения замеров времени.

Еще один внешний фактор, который мы должны рассмотреть, — сборка мусора. В идеале мы хотим предотвратить запуск сборки мусора во время измерений времени. Из-за недетерминированного характера сборки мусора ее невероятно трудно контролировать.

Одно из улучшений, которые мы, определенно, обязаны сделать, — это гарантировать, что мы не измеряем время, когда, вероятнее всего, запущена сборка мусора. Мы могли бы запросить у системы запуск сборки мусора и подождать некоторое короткое время перед началом измерений, но система может решить игнорировать этот вызов. Диапазон измеряемого времени в этом тесте слишком широкий, так что нам нужна более подробная информация о событиях сборки мусора, которые могут происходить.

Кроме точек измерения времени, следует также выбрать разумное количество итераций, которое тоже сложно оценить, так что приходится работать методом проб и ошибок. Влияние сборки мусора можно увидеть, если воспользоваться другим флагом виртуальной машины (детали формата журнального файла приводятся в главе 7, "Вглубь сборки мусора"):

```
java -Xms2048m -Xmx2048m -verbose:gc ClassicSort
```

Так мы получим записи журнального файла наподобие следующих:

```
Тестирование алгоритма сортировки
[GC (Allocation Failure) 524800K->632K(2010112K), 0.0009038 secs]
[GC (Allocation Failure) 525432K->672K(2010112K), 0.0008671 secs]
Результат: 9838.556465303362 операций/с
```

Еще одна распространенная ошибка в тестах производительности состоит в том, что на самом деле используется не результат, полученный из тестируемого кода. В тесте `copy`, по сути, является "мертвым кодом", так что JIT-компилятор может определить его как не выполняющийся путь и выбросить при оптимизации то, что мы на самом деле пытаемся тестировать.

Далее, единственный результат измерения времени, пусть и усредненный, не дает нам полной картины выполнения нашего теста производительности. В идеале мы хотим получить границы погрешности, чтобы понимать надежность собранных значений. Если погрешность высока, это может указывать на неконтролируемые факторы или на то, что написанный код на самом деле не работает. В любом случае без определения погрешности нет никакой возможности идентифицировать даже наличие проблемы.

Тест производительности даже очень простой сортировки может содержать ловушки, означающие, что его результаты нужно просто выбросить. Но с ростом сложности все становится гораздо-гораздо хуже. Рассмотрим тест производительности, который оценивает многопоточный код. Многопоточный код чрезвычайно трудно тестировать, так как, чтобы получить сколь-нибудь точные результаты, после запуска

требуется дождаться, чтобы все потоки полностью заработали. Если это не так, погрешность будет очень высокой.

Когда дело доходит до тестирования параллельного кода, встают также вопросы, касающиеся аппаратного обеспечения, и они выходят за рамки простой конфигурации оборудования.

Получение правильной производительности кода — сложная задача, предполагающая рассмотрение множества факторов. Для нас как разработчиков главной задачей являются не все только что затронутые вопросы, а код, который мы профилируем. Все вместе упомянутые проблемы создают ситуацию, когда — если только вы не эксперт по JVM — очень легко что-то пропустить и получить ошибочные результаты производительности.

Существует два способа решения этой проблемы. Первый заключается в том, чтобы тестировать только систему как единое целое. В этом случае низкоуровневые значения просто игнорируются и не накапливаются. Большое количество различных эффектов усредняется и позволяет получить значимый общий результат. Именно этот подход востребован в большинстве ситуаций и большинством разработчиков.

Второй подход — попытаться решить многие из вышеупомянутых проблем с помощью общего каркаса, чтобы обеспечить значимое сравнение соответствующих низкоуровневых результатов. Идеальный каркас позволяет забыть о некоторых из только что обсуждавшихся факторов. Такой инструмент должен следовать линии магистрального развития OpenJDK, чтобы уметь работать с новыми оптимизациями и внешними переменными.

К счастью, такой инструмент действительно существует и является предметом рассмотрения в следующем разделе. Для большинства разработчиков его следует рассматривать исключительно как справочный материал, так что его можно безопасно пропустить и переходить сразу к чтению следующего раздела — "Статистика производительности JVM".

Введение в JMH

Мы начнем с примера (и предостережения), как и почему можно легко ошибиться с микротестированием, если легкомысленно к нему подойти. Затем мы представим набор эвристик, которые подсказывают, годится ли ваш конкретный случай для использования микротестирования. Результат для подавляющего большинства случаев будет отрицательным.

Не занимайтесь микротестированием, если можете найти причину проблем (быль)

После очень долгого дня в офисе один из авторов уже уходил домой и по пути к выходу прошел мимо коллеги, все еще работавшей за своим монитором с одним

упорно не поддающимся методом Java. На следующий день ситуация повторилась. Спустя два дня опять та же картина — такие же метод на экране, усталость и раздражение на лице коллеги. Не требовалось никакого дедуктивного мышления, чтобы понять, что у нее проблемы.

Приложение, над которым она работала, имело легко наблюдаемые проблемы производительности. Новая версия также не хотела нормально работать, несмотря на использование новейших версий хорошо известных библиотек. Коллега потратила часть времени на удаление фрагментов кода и написание небольших тестов производительности в попытке найти источник проблем.

Это не совсем удачный подход — как поиск иголки в стоге сена. Вместо этого автор с коллегой поработали вместе с использованием другого подхода и быстро убедились, что приложение максимизирует использование процессора. Как известно, это хорошая ситуация для применения профайлера (читайте главу 13, "Профилирование", для получения подробной информации о том, когда следует использовать профайлеры), так что через 10 минут профилирования приложения истинная причина неприятностей была найдена. Разумеется, проблема оказалась не в коде приложения, а в новой библиотеке инфраструктуры, которую использовала команда.

Эта история иллюстрирует — увы, слишком частый! — подход к производительности Java. Разработчики могут оказаться одержимыми идеей, что во всем виновато их неумение и их собственный код, и не рассматривать более широкую картину.

Разработчики часто начинают охоту на проблемы, внимательно рассматривая небольшие фрагменты кода, но сравнительный анализ на этом уровне является чрезвычайно сложным и имеет некоторые опасные ловушки.

Эвристические правила, когда следует применять микротесты

Как мы кратко обсуждали в главе 2, "Обзор JVM", динамический характер платформы Java и такие функциональные возможности, как сборка мусора и агрессивная JIT-оптимизация, ведут к тому, что непосредственно о производительности трудно что-либо говорить. Что еще хуже, числовые характеристики производительности часто зависят от точных обстоятельств выполнения измерений.

Почти всегда легче анализировать истинную производительность всего приложения Java, чем небольшого фрагмента кода Java.

Однако иногда нужно непосредственно анализировать производительность отдельного метода или даже отдельного фрагмента кода, и к этому анализу не следует

подходить легкомысленно. В целом имеется три основных варианта ситуаций, когда применим низкоуровневый анализ или микротесты.

- Вы занимаетесь разработкой библиотеки кода общего назначения с широким диапазоном использования.
- Вы являетесь разработчиком на OpenJDK или другой реализации платформы Java.
- Вы разрабатываете чрезвычайно чувствительный к задержкам код (например, для торговли).

Обоснования для каждого из трех случаев немного различаются.

Библиотеки общего назначения (по определению) имеют ограниченное знание о контекстах, в которых они будут использоваться. Примерами таких библиотек являются Google Guava или Eclipse Collections (первоначально предоставленные Goldman Sachs). Они должны обеспечить приемлемую или лучшую производительность в очень широком диапазоне вариантов использования — от наборов данных, содержащих несколько десятков элементов, до сотен миллионов элементов.

Ввиду широты применения библиотеки общего назначения иногда вынуждены использовать микротестирование в качестве прокси для более традиционных методов тестирования производительности и нагрузки.

Разработчики платформ представляют собой ключевое сообщество пользователей микротестов, и инструмент JMH был создан командой OpenJDK главным образом для собственного использования. Однако это средство оказалось полезным для более широкого сообщества экспертов в области производительности.

Наконец, есть ряд разработчиков, которые работают на переднем крае производительности Java и которые, возможно, захотят использовать микротесты с целью выбора алгоритмов и методов, которые наилучшим образом соответствуют их приложениям и экстремальным вариантам использования. Сюда включаются, например, торговля с низкой задержкой и относительно небольшое количество других случаев.

Хотя это должно быть очевидно, если вы являетесь разработчиком, работающим над OpenJDK или библиотекой общего назначения, все же могут быть разработчики, которых сбивает с толку то, что их требования к производительности предусматривают применение микротестов.

> *Ужасной правдой о микротестах является то, что они всегда производят числовые данные, даже если эти числовые данные не имеют смысла. Они измеряют что-то; мы просто не уверены, что именно.*
>
> *— Брайан Гётц (Brian Goetz)*

В общем случае использовать микротесты должны только самые экстремальные приложения. Не существует точных правил, но если ваше приложение удовлетворяет

большинству или всем следующим критериям, вы вряд ли получите подлинную выгоду от микротестирования своего приложения.

- Общее время всего пути выполнения, определенно, менее 1 мс и, вероятно, менее 100 мкс.
- Вы должны измерить скорость выделения памяти (объектов) (подробности — в главах 6, "Сборка мусора", и 7, "Вглубь сборки мусора"), и она должна быть не больше 1 Мбит/с, а в идеале близка к нулю.
- Вы должны использовать почти 100% доступной производительности процессора, но коэффициент использования системы должен быть постоянно низким (менее 10%).
- Вы уже используете профайлер (см. главу 13, "Профилирование") для понимания распределения методов, потребляющих процессорное время. В распределении должно быть не более двух или трех доминирующих методов.

Учитывая все сказанное, должно быть очевидно, что микротестирование является передовой и редко используемой методикой. Тем не менее полезно понимать некоторые основные теории и сложности, которые оно отражает, так как это приводит к лучшему пониманию проблем производительности при работе в менее экстремальных приложениях на платформе Java.

> *Любой нанотест производительности, который не выполняет дизассемблирования и анализа сгенерированного кода, не может быть достоверным. Точка.*
>
> *— Алексей Шипилёв (Aleksey Shipilëv)*

В оставшейся части этого раздела микротестирование исследуется более тщательно, описываются некоторые из инструментов, а также приводятся соображения, которые разработчики должны принимать во внимание для того, чтобы получать надежные результаты, не приводящие к неверным выводам. Этот материал должен быть полезен для всех аналитиков в области производительности, независимо от того, имеет ли он непосредственное отношение к вашим текущим проектам.

Каркас JMH

JMH спроектирован как каркас для устранения проблем, которые мы только что обсудили.

> *JMH — это Java-модуль для создания, запуска и анализа нано/микро/милли/макропоказателей производительности, написанный на Java и других языках, ориентированных на JVM.*
>
> *— OpenJDK*

В прошлом было предпринято несколько попыток создания простых библиотек тестирования производительности, причем Google Caliper был одним из самых популярных среди разработчиков. Однако все эти каркасы имели свои проблемы, и часто то, что кажется рациональным способом настройки или измерения производительности кода, может иметь некоторые тонкие ловушки. Это особенно верно в отношении постоянно меняющейся природы JVM при применении новых оптимизаций.

JMH в этом отношении очень отличается, и его разработали те же инженеры, которые создавали JVM. Таким образом, авторы JMH знают, как избежать ловушек измерений и оптимизации, которые существуют в каждой версии JVM. JMH развивается, будучи тесно связанным с каждой версией JVM, что позволяет разработчикам просто сосредоточиться на использовании инструмента и самого кода.

JMH принимает во внимание некоторые дополнительные ключевые проблемы проектирования модулей измерения производительности, помимо уже освещенных здесь.

Каркас тестирования производительности должен быть динамичным, поскольку не знает содержимое теста во время компиляции. Один из очевидных способов обхода этой проблемы — выполнить тесты, написанные пользователем, с использованием рефлексии. Однако это включает в пути выполнения теста другую сложную подсистему JVM. Вместо этого JMH работает путем генерации дополнительного исходного текста Java из теста путем обработки аннотаций.

Многие распространенные каркасы Java на основе аннотаций (например, JUnit) используют для достижения своих целей рефлексию, поэтому использование процессора, который генерирует дополнительный исходный текст, может быть несколько неожиданным для ряда разработчиков на языке Java.

Одна из проблем заключается в том, что если каркас тестирования производительности должен вызывать код пользователя большое количество раз, то может быть запущена оптимизация цикла. Это означает, что фактический процесс запуска теста производительности может привести к проблемам с надежностью результатов.

Чтобы избежать проблем оптимизации цикла, JMH генерирует код теста производительности, обертывая тестируемый код в цикл, счетчик итераций которого устанавливается равным тщательно выбранному значению, которое позволяет избежать оптимизации.

Выполнение тестов производительности

Сложности, связанные с выполнением JMH, в основном скрыты от пользователя, а настройка простого теста с использованием Maven несложная. Мы можем создать новый проект JMH, выполнив следующую команду:

```
$ mvn archetype:generate \
        -DinteractiveMode=false \
        -DarchetypeGroupId=org.openjdk.jmh \
        -DarchetypeArtifactId=jmh-java-benchmark-archetype \
        -DgroupId=org.sample \
        -DartifactId=test \
        -Dversion=1.0
```

Эта команда загружает необходимые артефакты и создает единую заглушку тестирования для размещения кода.

Тест аннотируется с помощью аннотации `@Benchmark`, указывающей, что модуль будет выполнять метод для его тестирования (после того, как каркас выполнит различные задачи настройки):

```
public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {
        // Заглушка для кода
    }
}
```

Автор теста может выполнить настройку параметров выполнения теста производительности. Параметры можно задать как в командной строке, так и в методе `main()` теста, как показано далее:

```
public class MyBenchmark
{
    public static void main(String[] args) throws RunnerException
    {
        Options opt = new OptionsBuilder()
        .include(SortBenchmark.class.getSimpleName())
        .warmupIterations(100)
        .measurementIterations(5).forks(1)
        .jvmArgs("-server", "-Xms2048m", "-Xmx2048m").build();
        new Runner(opt).run();
    }
}
```

Параметры командной строки перекрывают любые параметры, заданные в методе `main()`.

Обычно тест производительности требует некоторых настроек, например создание набора данных или создание условий, необходимых для получения статистически независимого набора показателей для сравнения производительности.

Состояние и контролирующее состояние — еще одна особенность каркаса JMH. Аннотация `@State` может использоваться для определения состояния, а значения перечисления `Scope` — для определения того, где состояние видимо: `Benchmark`,

Group или Thread. Объекты, которые аннотируются с помощью @State, достижимы на протяжении всего жизненного цикла теста производительности. При этом может потребоваться выполнение определенной настройки.

Многопоточный код также требует особого внимания, чтобы гарантировать отсутствие смещения результатов из-за плохо управляемого состояния.

В общем случае, если выполняемый в методе код не имеет побочных действий, а его результат не используется, то такой метод является кандидатом для удаления с помощью JVM. JMH необходимо предотвратить это удаление, и на самом деле это чрезвычайно простая задача для опытного автора тестов производительности. Из метода тестирования могут возвращаться единичные результаты, и каркас гарантирует, что это значение неявно присваивается *черной дыре* (blackhole) — разработанному авторами каркаса механизму с пренебрежимо малыми накладными расходами.

Если тест выполняет несколько вычислений, объединение и возврат результатов из тестируемого метода могут оказаться дорогостоящими. В этом случае может потребоваться использование явной черной дыры путем создания теста, который принимает черную дыру в качестве параметра.

Черные дыры обеспечивают четыре гарантии, относящиеся к оптимизации, которые могут потенциально влиять на тест производительности. Одни из них связаны с предотвращением излишней оптимизации из-за ограниченной области видимости, другие — с избеганием предсказуемой схемы данных во время выполнения, чего не может быть во время обычного запуска системы. Вот эти гарантии.

- Устранение распознавания мертвого кода, который может быть удален при оптимизации во время выполнения.
- Предотвращение свертки повторных вычислений в константы.
- Предотвращение ложного совместного использования, когда чтение или запись значения может повлиять на текущую строку кеша.
- Защита от "стен записи".

Термин *стена* (wall) в области производительности обычно относится к точке насыщения ваших ресурсов, которая фактически становится узким местом вашего приложения. Достижение "стены записи" (write wall) может влиять на кеши и загрязнять буфера, используемые для записи. Если вы поступаете таким образом в вашем тесте, то это потенциально очень сильно сказывается на его результатах.

Как описано в документации Blackhole (и как отмечалось ранее), чтобы обеспечить выполнение этих гарантий, требуется глубокое знание JIT-компилятора, позволяющее создать тест производительности, избегающий оптимизации.

Бросим беглый взгляд на два метода consume(), используемые черными дырами, чтобы получить представление о некоторых приемах, которые использует JMH (вы можете пропустить этот фрагмент, если вас не интересует реализация JMH):

```
public volatile int i1 = 1, i2 = 2;
/**
* Объект потребления. Данный вызов обеспечивает
* в качестве побочного действия предотвращение в JIT
* устранения зависимых вычислений.
*
* @param i потребляемое значение int.
*/
public final void consume(int i)
{
    if (i == i1 & i == i2)
    {
        // ВЫПОЛНЕНИЕ НИКОГДА НЕ ДОЛЖНО СЮДА ПОПАСТЬ
        nullBait.i1 = i; // Неявное исключение нулевого указателя
    }
}
```

Мы повторяем этот код для потребления всех примитивов (изменяя `int` на соответствующий примитивный тип). Переменные `i1` и `i2` объявлены как `volatile`, а это означает, что среда выполнения должна заново их вычислять. Оператор `if` никогда не может быть истинным, но компилятор должен позволить коду выполниться. Обратите также внимание на использование побитового оператора AND (`&`) внутри конструкции `if`. Это позволяет избежать проблематичной дополнительной логики ветвления и приводит к более равномерной производительности.

Вот второй метод:

```
public int tlr = (int) System.nanoTime();
/**
* Объект потребления. Данный вызов обеспечивает
* в качестве побочного действия предотвращение в JIT
* устранения зависимых вычислений.
*
* @param obj потребляемый объект.
*/
public final void consume(Object obj)
{
    int tlr = (this.tlr = (this.tlr * 1664525 + 1013904223));

    if ((tlr & tlrMask) == 0)
    {
        // ПРИ ИЗМЕРЕНИЯХ ПОЧТИ НИКОГДА НЕ ДОЛЖНО ПРОИЗОЙТИ
        this.obj1 = obj;
        this.tlrMask = (this.tlrMask << 1) + 1;
    }
}
```

Когда дело доходит до объектов, на первый взгляд кажется, что можно применить ту же самую логику, поскольку у пользователя нет ничего, что могло бы быть равным объектам, которыми владеет `Blackhole`. Однако компилятор пытается быть достаточно умным в этом отношении. Если анализ компилятора утверждает, что объект никогда не равен чему-то другому, то возможно, что само сравнение может быть оптимизировано и заменено простым возвратом `false`.

Вместо этого объекты потребляются только при крайне редко выполняемом условии. Значение `tlr` вычисляется и побитово сравнивается с `tlrMask`, чтобы уменьшить вероятность появления нулевого значения, но не устранить его полностью. Это гарантирует, что объекты потребляются в основном без необходимости их присваивания. Код каркаса для микротестов чрезвычайно интересен, поскольку очень сильно отличается от реальных приложений Java. Фактически, если бы такой код был найден в любом месте производственного Java-приложения, вероятно, его разработчик был бы тут же уволен.

Помимо чрезвычайно точного инструмента микротестирования, авторам удалось создать впечатляющую документацию по классам. Если вас интересует происходящее за кулисами волшебство, обратите внимание на комментарии: они отлично его поясняют.

После знакомства с представленной информацией написание простого теста и его выполнение не отнимет много времени, но у JMH имеются и некоторые дополнительные возможности. Официальная документация[1] содержит примеры каждой из них — и все они заслуживают вашего внимания.

Интересные возможности, демонстрирующие всю мощь JMH и его относительную близость к JVM, включают следующее.

- Возможность управления компилятором.
- Имитация уровня использования процессора во время тестирования производительности.

Еще одной интересной особенностью является использование черных дыр для фактического потребления процессорного времени, что позволяет моделировать тесты при различных нагрузках процессора.

Чтобы попросить компилятор не включить, явно встроить или исключить метод из компиляции, можно использовать аннотацию `@CompilerControl`. Это чрезвычайно полезно, если вы сталкиваетесь с проблемой производительности и подозреваете, что она связана со встраиванием или компиляцией JVM:

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@OutputTimeUnit(TimeUnit.SECONDS)
```

[1] По адресу `http://openjdk.java.net/projects/code-tools/jmh/`.

```
@Fork(1)
public class SortBenchmark
{
    private static final int N = 1_000;
    private static final List<Integer> testData = new ArrayList<>();
    @Setup
    public static final void setup()
    {
        Random randomGenerator = new Random();

        for (int i = 0; i < N; i++)
        {
            testData.add(randomGenerator.nextInt(Integer.MAX_VALUE));
        }

        System.out.println("Setup Complete");
    }
    @Benchmark
    public List<Integer> classicSort()
    {
        List<Integer> copy = new ArrayList<Integer>(testData);
        Collections.sort(copy);
        return copy;
    }
    @Benchmark
    public List<Integer> standardSort()
    {
        return testData.stream().sorted().collect(Collectors.toList());
    }
    @Benchmark
    public List<Integer> parallelSort()
    {
        return testData.parallelStream().sorted().collect(
                    Collectors.toList());
    }
    public static void main(String[] args) throws RunnerException
    {
        Options opt = new OptionsBuilder()
        .include(SortBenchmark.class.getSimpleName())
        .warmupIterations(100)
        .measurementIterations(5).forks(1)
        .jvmArgs("-server", "-Xms2048m", "-Xmx2048m")
        .addProfiler(GCProfiler.class)
        .addProfiler(StackProfiler.class)
        .build();
        new Runner(opt).run();
    }
}
```

Выполнение этого теста дает следующий вывод:

```
Benchmark                              Mode Cnt     Score     Error Units
optjava.SortBenchmark.classicSort     thrpt 200 14373.039 ± 111.586 ops/s
optjava.SortBenchmark.parallelSort    thrpt 200  7917.702 ±  87.757 ops/s
optjava.SortBenchmark.standardSort    thrpt 200 12656.107 ±  84.849 ops/s
```

Глядя на этот тест, можно легко прийти к быстрому выводу, что классический метод сортировки более эффективен, чем использование потоков. Оба кода используют один и тот же массив и одну и ту же сортировку, поэтому все вроде бы верно. Разработчики могут взглянуть на низкую погрешность и высокую пропускную способность и заключить, что тест производительности должен быть правильным.

Но давайте рассмотрим некоторые причины, по которым наш тест может давать неточную картину производительности, — в основном пытаясь ответить на вопрос, является ли этот тест выборочным испытанием. Для начала рассмотрим влияние сборки мусора в тесте `classicSort`:

```
Iteration     1:
[GC (Allocation Failure) 65496K->1480K(239104K), 0.0012473 secs]
[GC (Allocation Failure) 63944K->1496K(237056K), 0.0013170 secs]
10830.105 ops/s
Iteration     2:
[GC (Allocation Failure) 62936K->1680K(236032K), 0.0004776 secs]
10951.704 ops/s
```

Из этого снимка ясно, что имеется один цикл сборки мусора, запускаемый (приблизительно) на каждой итерации. Интересно сравнить этот результат с параллельной сортировкой:

```
Iteration     1:
[GC (Allocation Failure) 52952K->1848K(225792K), 0.0005354 secs]
[GC (Allocation Failure) 52024K->1848K(226816K), 0.0005341 secs]
[GC (Allocation Failure) 51000K->1784K(223744K), 0.0005509 secs]
[GC (Allocation Failure) 49912K->1784K(225280K), 0.0003952 secs]
9526.212 ops/s
Iteration     2:
[GC (Allocation Failure) 49400K->1912K(222720K), 0.0005589 secs]
[GC (Allocation Failure) 49016K->1832K(223744K), 0.0004594 secs]
[GC (Allocation Failure) 48424K->1864K(221696K), 0.0005370 secs]
[GC (Allocation Failure) 47944K->1832K(222720K), 0.0004966 secs]
[GC (Allocation Failure) 47400K->1864K(220672K), 0.0005004 secs]
```

Таким образом, добавляя флаги, чтобы увидеть, что вызывает такой неожиданный разрыв, мы можем видеть, что в тесте имеется шум (в данном случае — из-за сборки мусора).

Вывод, который следует сделать, заключается в том, что легко предположить, что тест производительности представляет собой контролируемую среду, но правда

может оказаться гораздо более скользкой. Часто трудно обнаружить неконтролируемые факторы, поэтому даже с использованием такого инструмента, как JMH, необходимо соблюдать осторожность. Мы также должны позаботиться о том, чтобы исправить искажения в наших подтверждениях и убедиться, что то, что мы отслеживаем, действительно отражает поведение нашей системы.

В главе 9, "Выполнение кода в JMV", мы встретимся с инструментом JITWatch, который даст нам еще один взгляд на то, что JIT-компилятор делает с нашим байт-кодом. Это может помочь понять, почему байт-код, созданный для конкретного метода, может привести к тому, что тест выполняется не так, как ожидалось.

Статистика производительности JVM

Если анализ эффективности действительно является экспериментальной наукой, то мы неизбежно столкнемся с распределением получаемых данных. Статистики и ученые знают, что результаты, которые добываются из реального мира, практически никогда не представлены чистыми, отдельными сигналами. Мы должны иметь дело с миром, какой он есть, а не с идеализированным, каким мы хотели бы его видеть.

> *Богу мы верим. Все остальные передают данные.*
>
> *— Майкл Блумберг (Michael Bloomberg)*

Все измерения содержат некоторое количество ошибок. В следующем разделе мы опишем два основных типа ошибок, с которыми разработчик на языке Java может столкнуться при выполнении анализа производительности.

Типы ошибок

Есть два основных источника ошибок, с которыми может столкнуться инженер.

Случайная ошибка

Ошибка измерения или обусловленная некоррелированным влиянием не связанных факторов.

Систематическая ошибка

Неучтенный фактор, влияющий на измерения наблюдаемого явления коррелированным образом.

Есть определенные слова, связанные с каждым типом ошибки. Например, термин *точность* (accuracy) используется для описания уровня систематической ошибки измерения; высокая точность соответствует низкой систематической ошибке. Аналогично *прецизионность* (precision) представляет собой термин, соответствующий

понятию случайной ошибки; высокая прецизионность означает низкую случайную ошибку[2].

На рис. 5.1 показано влияние этих двух типов ошибок на измерение. Крайнее слева изображение показывает кластеризацию измерений вокруг истинного результата ("центр мишени"). Эти измерения имеют высокую точность и высокую прецизионность. Второе изображение подвержено систематической ошибке — все выстрелы ложатся кучно, но не в центре: это измерения с высокой точностью, но низкой прецизионностью. Третье изображение показывает слабо кластеризованные попадания вокруг центра, обладающие низкой точностью, но высокой прецизионностью. И наконец, последнее изображение иллюстрирует низкую точность и низкую прецизионность.

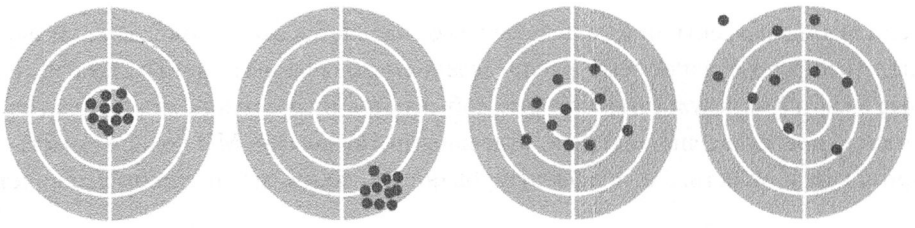

Рис. 5.1. Типы ошибок

Систематическая ошибка

В качестве примера рассмотрим тест производительности, работающий с группой веб-сервисов Java, которые отправляют и получают информацию в формате JSON. Этот очень распространенный тип теста, применяющийся тогда, когда непосредственно использовать интерфейс приложения для тестирования нагрузки проблематично.

Рис. 5.2 был создан пакетом расширения JP GC к инструментарию генерации нагрузки Apache JMeter. В нем фактически действуют два систематических фактора. Первый — линейная картина, показанная верхней линией и представляющая собой медленное истощение некоторого ограниченного ресурса сервера. Эта картина часто связана с утечкой памяти или другого ресурса, который используется потоком во время обработки запроса и после этого не освобождается (и представляет собой первого кандидата на исследование — дело выглядит так, что эта утечка может быть реальной проблемой).

Необходим дальнейший анализ, который подтвердит тип ресурса, подвергшегося влиянию утечки; нельзя просто так сделать вывод, что это утечка памяти.

[2] В русском языке обычно используется одно слово — *точность*; конкретный смысл обычно очевиден из контекста. — *Примеч. пер.*

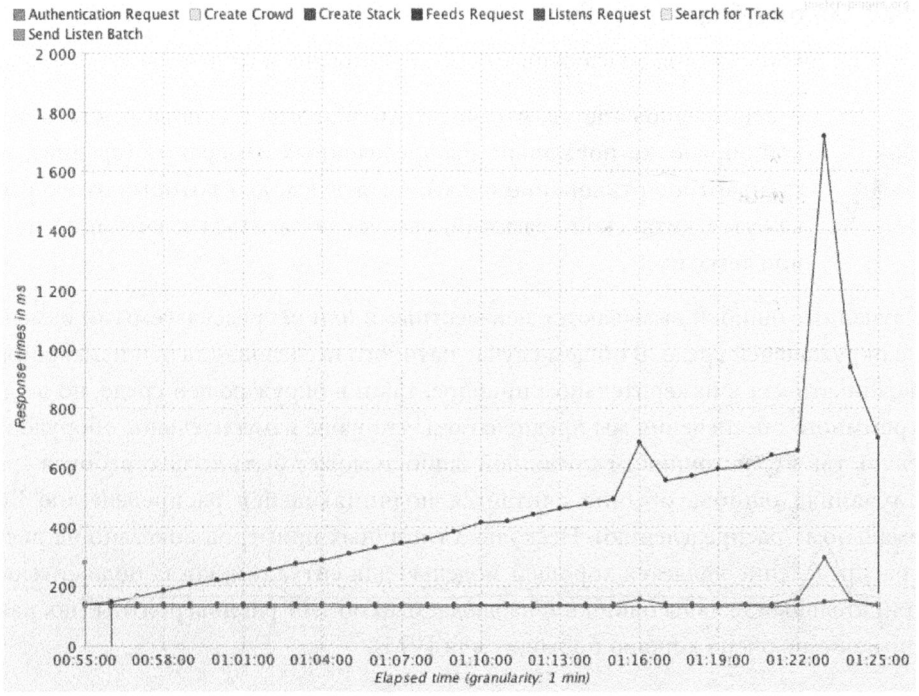

Рис. 5.2. Систематическая ошибка

Второй эффект, о котором следует упомянуть, — это согласованное время работы большинства других служб на уровне примерно 180 мс. Это подозрительно, так как службы выполняют достаточно разное количество работы в ответ на запрос. Тогда почему же результаты настолько похожи?

Ответ заключается в том, что, хотя тестируемые службы находятся в Лондоне, сам тест нагрузки был выполнен из Мумбаи (Индия). Наблюдаемое время отклика включает задержки при передаче по сети из Мумбаи в Лондон и обратно. Они находятся в диапазоне 120–150 мс, поэтому подавляющее большинство наблюдаемых времен (помимо выбросов) так похожи.

Этот большой, систематический эффект нивелирует различия в фактических временах отклика (так как время отклика служб гораздо меньше, чем 120 мс). Это пример систематической ошибки, которая не связана с проблемой в нашем приложении. Ошибка связана с проблемой в нашей тестовой конфигурации, и хорошая новость заключается в том, что этот артефакт полностью исчез (как и ожидалось), когда тест был повторно проведен из Лондона.

Случайная ошибка

Случайная ошибка тоже заслуживает упоминания, несмотря на свою банальность.

Здесь предполагается, что читатели знакомы с основами статистической обработки нормально распределенных измерений (среднее, мода, стандартное отклонение и т.д.); читателям, для которых эти термины кажутся китайской грамотой, следует обратиться к учебнику по данной тематике.

Случайные ошибки вызываются неизвестными или непредсказуемыми изменениями в окружающей среде. В общем случае научного исследования эти изменения могут возникать как в измерительном приборе, так и в окружающей среде, но в случае программного обеспечения мы предполагаем, что наше измерительное оборудование надежно, так что источником случайной ошибки может быть только рабочая среда.

Случайная ошибка обычно считается подчиняющейся распределению Гаусса (нормальному распределению). Несколько типичных примеров показано на рис. 5.3. Это распределение является хорошей моделью для ситуации, когда положительный и отрицательный вклады ошибки в наблюдаемый объект равновероятны (но, как мы увидим, оно не очень хорошо подходит для JVM).

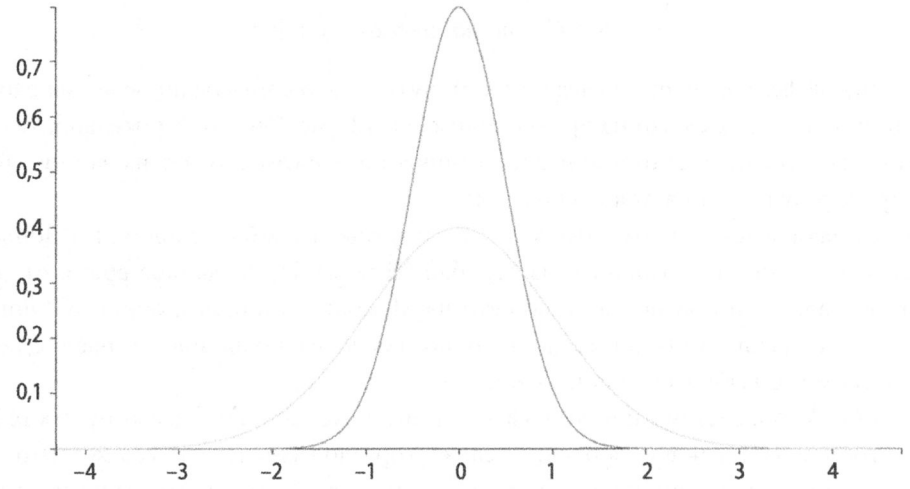

Рис. 5.3. Распределение Гаусса (нормальное распределение, колоколообразная кривая)

Ложная корреляция

Один из самых известных афоризмов о статистике — "корреляция не означает причинно-следственной связи", т.е. то, что две переменные ведут себя подобно одна другой, никоим образом не означает, что между ними существует связь.

В качестве самых экстремальных примеров корреляцию можно найти между совершенно несвязанными измерениями. Например, на рис. 5.4 можно видеть, что потребление курятины в США хорошо коррелирует с импортом сырой нефти[3].

Рис. 5.4. Пример ложной корреляции

Очевидно, что эти цифры явно не связанны причинно-следственными связями; не существует фактора, управляющего импортом сырой нефти и потреблением курятины.

На рис. 5.5 мы видим, что доходы от видеоигр коррелируют с количеством защищенных диссертаций в области компьютерных наук. Не слишком сложно представить себе социологическое исследование, обнаруживающее связь между этими значениями и, возможно, утверждающее, что "находящиеся в состоянии стресса соискатели ученой степени дают выход эмоциям с помощью видеоигр". Такого рода спекуляции оказываются удручающе распространенными, несмотря на отсутствие реального общего фактора.

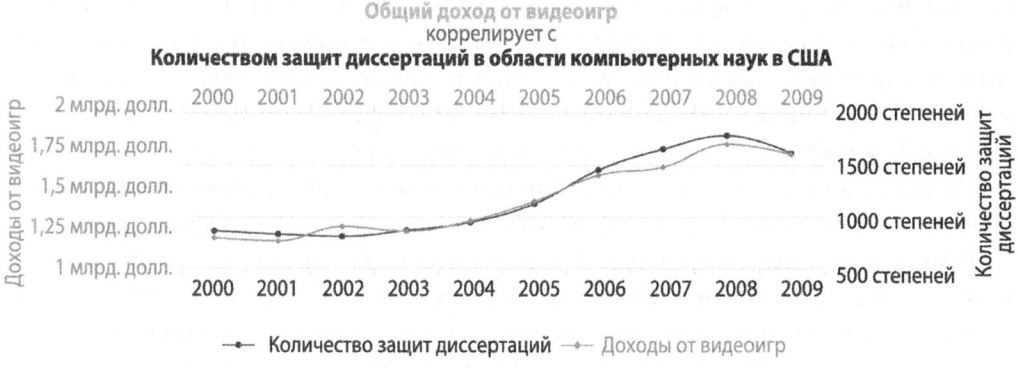

Рис. 5.5. Менее ложная корреляция?

[3] http://tylervigen.com/spurious-correlations

В области JVM и анализа производительности мы должны быть особенно осторожны, чтобы не приписывать причинно-следственную связь измерениям исключительно на основе корреляции и того, что такая связь "кажется правдоподобной". Это можно рассматривать как один из аспектов принципа Фейнмана — *вы должны не дурачить самих себя.*

Теперь, когда мы познакомились с некоторыми примерами источников ошибок и упомянули об известной ловушке ложной корреляции, перейдем к обсуждению одного аспекта измерения производительности JVM, требующего особой осторожности и внимания к деталям.

Статистика, отличная от нормальной

Статистика, основанная на нормальном распределении, не требует большой математической сложности. По этой причине стандартный подход к статистике, который обычно преподается на уровне до колледжа или бакалавриата, в значительной степени фокусируется на анализе именно нормально распределенных данных.

Студентов учат вычислять среднее и стандартное отклонения (или дисперсию), а иногда и более высокие моменты, такие как перекос (skew) и эксцесс (kurtosis). Однако эти методы имеют серьезный недостаток, поскольку результаты могут легко искажаться, если в распределении есть даже несколько достаточно отдаленных точек.

Что касается производительности Java, то здесь выбросы — это медленные операции и недовольные клиенты. Мы должны обратить особое внимание на эти точки и избегать статистических методов, которые нивелируют выбросы.

Можно рассмотреть это с другой точки зрения: маловероятно, что целью является улучшение среднего времени отклика, если только на него не жалуется большинство клиентов. Разумеется, это улучшит ситуацию для всех, но гораздо более обычным явлением является настройка снижения задержки для нескольких недовольных клиентов. Это означает, что события с выбросами, вероятно, представляют больший интерес, чем опыт удовлетворительного обслуживания большинства.

На рис. 5.6 показана более реалистичная кривая распределения времен вызова метода (или выполнения транзакции). Это явно не нормальное распределение.

Форма распределения на рис. 5.6 показывает то, что мы интуитивно знаем о JVM: у нее есть "горячие пути", на которых весь код уже скомпилирован JIT-компилятором, нет циклов сборки мусора и т.д. Они представляют собой оптимальный (хотя и достаточно распространенный) сценарий; просто нет вызовов, которые случайно оказываются еще немного быстрее.

Это нарушает фундаментальную предпосылку гауссовой статистики и заставляет нас рассматривать распределения, отличные от нормального.

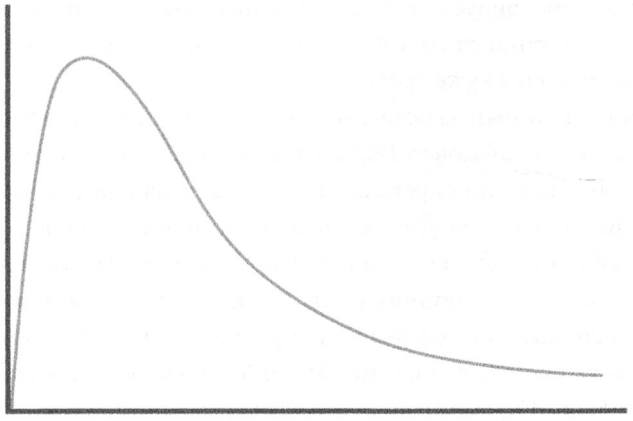

Рис. 5.6. Более реалистичное представление распределения времен транзакций

Для распределений, которые не являются нормальными, нарушаются многие "фундаментальные правила" статистики нормальных распределений. В частности, стандартное отклонение/дисперсия и другие высшие моменты в основном оказываются бесполезными.

Один из методов, которые очень полезны для обработки не нормальных, "длиннохвостых", распределений, которые создает JVM, заключается в использовании модифицированной схемы процентилей. Помните, что распределение представляет собой целый график, т.е. распределение — это форма данных, которая не может быть представлена одним числом.

Вместо вычисления одного среднего значения, которое представляет собой попытку выразить все распределение с помощью единственного результата, мы можем использовать выборку распределения по интервалам. Выборка для нормально распределенных данных обычно производится через равные промежутки. Однако внесение небольших изменений позволяет использовать этот метод для статистики JVM.

Эти изменения заключаются в использовании выборки, которая учитывает длинный хвост распределения, начиная со среднего, затем — 90-го процентиля и логарифмически перемещаясь далее, как показано в приведенных далее результатах. Это означает, что мы выполняем выборку по схеме, которая точнее соответствует форме данных:

```
50.0%    выполнений быстрее    23 нс
90.0%    выполнений быстрее    30 нс
99.0%    выполнений быстрее    43 нс
99.9%    выполнений быстрее   164 нс
99.99%   выполнений быстрее   248 нс
99.999%  выполнений быстрее  3458 нс
99.9999% выполнений быстрее 17463 нс
```

Эта выборка демонстрирует, что, хотя среднее время выполнения метода и составляет 23 нс, для 1 запроса из 1000 время на порядок хуже, а для 1 запроса из 100 000 оно на *два* порядка хуже среднего.

Распределения с длинным хвостом могут также именоваться распределениями *с высоким динамическим диапазоном* (high dynamic range). Динамический диапазон наблюдаемой величины обычно определяется как максимальное записанное значение, деленное на минимальное (при условии, что оно отлично от нуля).

Логарифмические процентили являются простым полезным инструментом для понимания распределений с длинными хвостами. Однако для более сложного анализа мы можем использовать общедоступную библиотеку для обработки данных с высоким динамическим диапазоном. Эта библиотека называется `HdrHistogram` и доступна на GitHub[4]. Первоначально она была разработана Жилом Тене (Gil Tene) из Azul Systems с дополнениями от Майка Баркера (Mike Barker), Дараша Энниса (Darach Ennis) и Кода Хейла (Coda Hale).

Гистограмма представляет собой способ подытоживания данных с использованием конечного набора диапазонов (называемых *корзинами* (bucket)) и отображения, как часто данные попадают в каждую из корзин.

HdrHistogram доступна также в репозитории Maven Central. На момент написания книги текущей ее версией была 2.1.9, и вы могли добавить ее в свои проекты с помощью следующего фрагмента зависимости в `pom.xml`:

```
<dependency>
    <groupId>org.hdrhistogram</groupId>
    <artifactId>HdrHistogram</artifactId>
    <version>2.1.9</version>
</dependency>
```

Давайте рассмотрим простой пример с использованием HdrHistogram. В этом примере выполняется считывание чисел из файла и выполняется вычисление HdrHistogram разности между последовательными результатами:

```
public class BenchmarkWithHdrHistogram
{
    private static final long NORMALIZER = 1_000_000;
    private static final Histogram HISTOGRAM
        = new Histogram(TimeUnit.MINUTES.toMicros(1), 2);
    public static void main(String[] args) throws Exception
    {
        final List<String> values = Files.readAllLines(Paths.get(args[0]));
        double last = 0;
```

[4] https://github.com/HdrHistogram/HdrHistogram

```
        for (final String tVal : values)
        {
            double parsed = Double.parseDouble(tVal);
            double gcInterval = parsed - last;
            last = parsed;
            HISTOGRAM.recordValue((long)(gcInterval * NORMALIZER));
        }

        HISTOGRAM.outputPercentileDistribution(System.out, 1000.0);
    }
}
```

На выходе мы получаем время между последовательными сборками мусора. Как мы увидим в главах 6, "Сборка мусора", и 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора", сборка мусора происходит через разные, не равные один другому промежутки времени, и знание распределения частоты сборки мусора может оказаться полезным. Вот какая гистограмма создается для образца журнала сборки мусора:

```
       Value     Percentile   TotalCount   1/(1-Percentile)
       14.02   0.000000000000          1             1.00
     1245.18   0.100000000000         37             1.11
     1949.70   0.200000000000         82             1.25
     1966.08   0.300000000000        126             1.43
     1982.46   0.400000000000        157             1.67
...
    28180.48   0.996484375000        368           284.44
    28180.48   0.996875000000        368           320.00
    28180.48   0.997265625000        368           365.71
    36438.02   0.997656250000        369           426.67
    36438.02   1.000000000000        369
#[Mean    =   2715.12,  StdDeviation  =   2875.87]
#[Max     =  36438.02,  Total count   =       369]
#[Buckets =        19,  SubBuckets    =       256]
```

Необработанный вывод довольно трудно анализировать, но, к счастью, проект HdrHistogram включает в себя онлайн-форматировщик[5], который может использоваться для создания визуальных гистограмм из необработанных данных.

В данном примере мы получаем визуализацию, показанную на рис. 5.7.

Для многих наблюдаемых параметров, которые мы хотим измерить при настройке производительности Java, статистика часто носит характер, крайне далекий от нормального, а HdrHistogram может быть очень полезным инструментом, помогающим понять и визуализировать вид данных.

[5] http://hdrhistogram.github.io/HdrHistogram/plotFiles.html

Рис. 5.7. Пример визуализации HdrHistogram

Интерпретация статистики

Эмпирические данные и наблюдаемые результаты существуют не в вакууме, и довольно часто одно из сложнейших заданий лежит в области интерпретации результатов, которые мы получаем при измерениях нашего приложения.

Независимо от того, что вам говорят, проблема всегда в людях.

— Джеральд Вайнберг (Gerald Weinberg)

На рис. 5.8 показана скорость выделения памяти в реальном приложении Java. Этот пример демонстрирует достаточно хорошо функционирующее приложение. Этот график — результат работы анализатора сборки мусора Censum, с которым мы встретимся в главе 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора".

Интерпретация данных о выделении памяти относительно проста, так как здесь присутствует ясный сигнал. За рассматриваемый период (почти сутки) скорость выделения в основном стабильна, между 350 и 700 Мбайт/с. Имеется определенная тенденция к снижению скорости, начиная примерно через 5 ч после запуска JVM, и четкий минимум между 9 и 10 ч, после чего скорость выделения снова начинает расти.

Такие тенденции в наблюдаемых параметрах весьма распространены, поскольку скорость выделения памяти обычно отражает объем работы, которую выполняет приложение, а он сильно варьируется в зависимости от времени суток. Однако, когда мы интерпретируем реальные наблюдения, картина может быстро усложниться. Это может привести к тому, что иногда называют проблемой "шляпы/слона" (вспомните "Маленького принца" Антуана де Сент-Экзюпери).

Эта проблема проиллюстрирована на рис. 5.9. Все, что мы можем просто увидеть при беглом взгляде, — это сложная гистограмма времени отклика на HTTP-запрос. Однако (как и рассказчик в книге), если мы можем представить себе (или

проанализировать) немного больше, то увидим, что сложная картина на самом деле состоит из нескольких довольно простых частей.

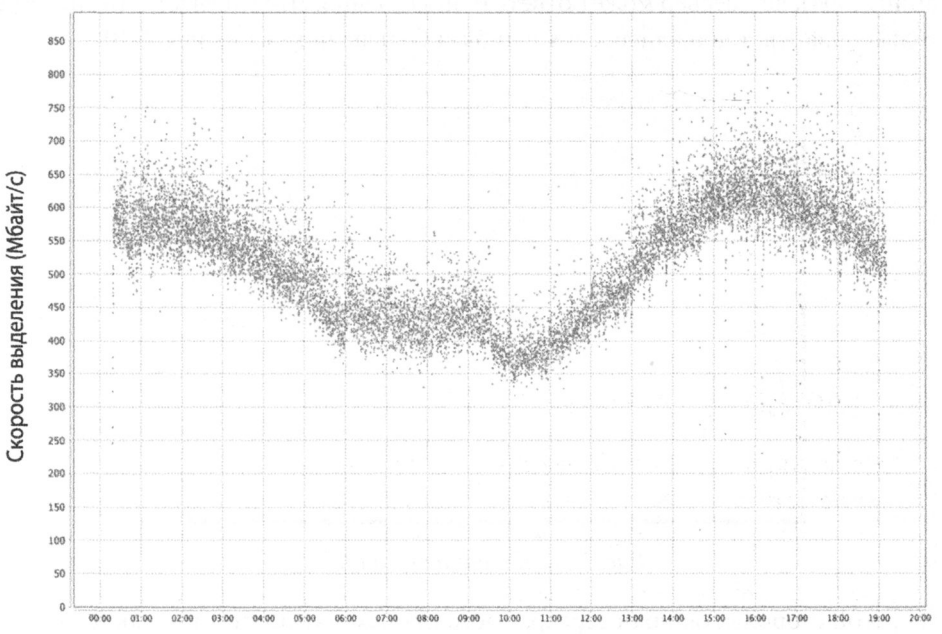

Рис. 5.8. Скорость выделения памяти

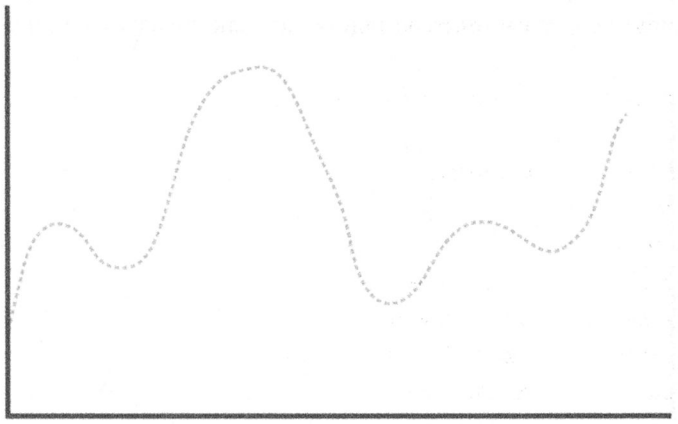

Рис. 5.9. Шляпа или слон, проглоченный питоном?

Ключом к декодированию гистограммы отклика является понимание того, что "отклик веб-приложения" является очень общей категорией, включающей успешные запросы (так называемые ответы 2xx), клиентские ошибки (4xx, включая вездесущую ошибку 404) и ошибки сервера (5xx, в особенности *500 Internal Server Error*).

Каждый тип ответа имеет различное распределение характеристик времени отклика. Если клиент делает запрос URL-адреса, которому не соответствует ни одна страница (404), веб-сервер может ответить немедленно. Это означает, что гистограмма только для ответов о клиентских ошибках больше похожа на рис. 5.10.

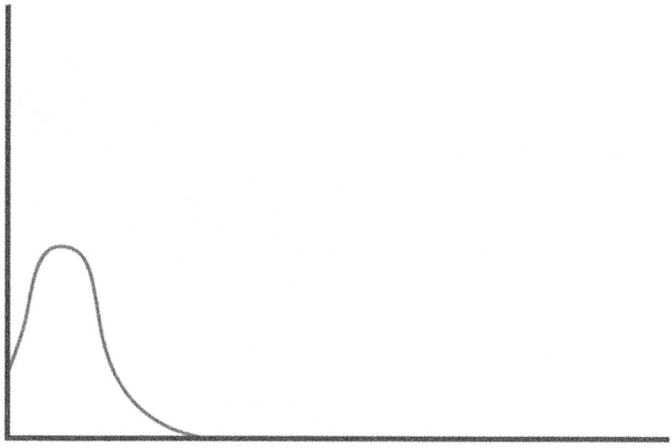

Рис. 5.10. Клиентские ошибки

С другой стороны, ошибки сервера часто происходят после того, как на обработку было израсходовано большое количество времени (например, из-за нагрузки на ресурсы сервера или превышение времени ожидания). Таким образом, гистограмма ответов об ошибке сервера может выглядеть так, как показано на рис. 5.11.

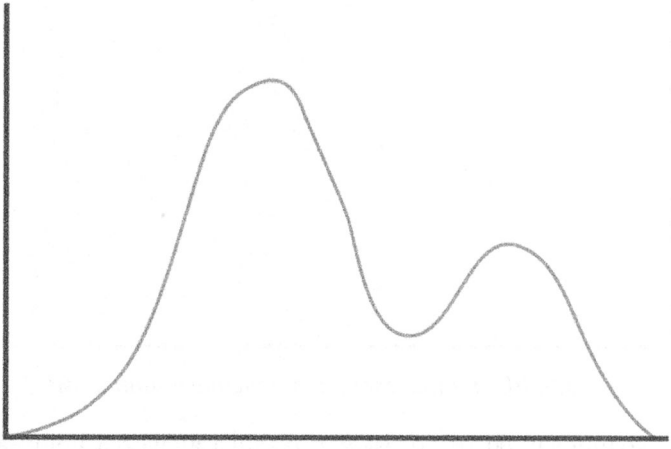

Рис. 5.11. Ошибки сервера

Успешные запросы будут иметь распределение с длинным хвостом, но на самом деле можно ожидать, что распределение откликов будет мультимодальным

и иметь несколько локальных максимумов. Пример такого распределения показан на рис. 5.12 и представляет собой возможность того, что в приложении могут быть два основных пути выполнения с совершенно разными значениями времени отклика.

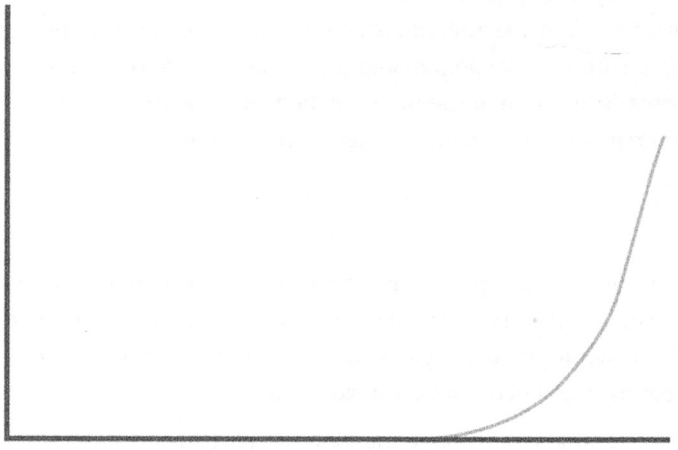

Рис. 5.12. Успешные запросы

Объединение этих различных типов ответов в один график приводит к структуре, показанной на рис. 5.13. Мы получаем нашу первоначальную "шляпу" из отдельных гистограмм.

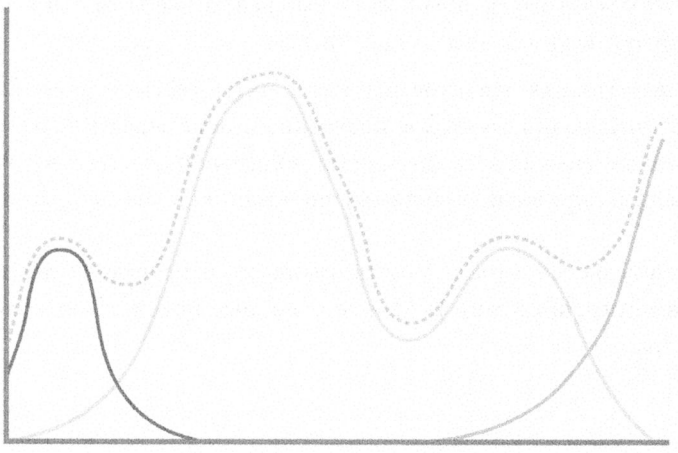

Рис. 5.13. Шляпа или слон — при детальном рассмотрении

Концепция разбиения общего наблюдаемого явления на несколько значимых подъявлений является весьма полезной и показывает, что, прежде чем пытаться делать выводы из полученных результатов, мы обязаны убедиться, что хорошо понимаем наши данные и саму предметную область. Мы можем захотеть разделить наши

результаты на еще меньшие наборы; например, успешные запросы могут иметь весьма различные распределения для запросов, которые преимущественно выполняют чтение информации, в отличие от запросов обновления или загрузки.

Команда инженеров PayPal описала свое использование статистических данных и анализа; у них есть блог[6], который содержит немало отличных ресурсов. В частности, раздел *Статистика для программного обеспечения*[7] Махмуда Хашеми (Mahmoud Hashemi) является отличным введением в используемую ими методологию и включает версию рассмотренной выше проблемы "шляпа/слон".

Резюме

В микротестировании вопросы изучения производительности Java оказываются наиболее близкими к термину "темное искусство". Тем не менее это все еще инженерная дисциплина, используемая разработчиками на Java. И все же микротестирование требует от разработчиков особой осторожности.

- Не проводите микротестирование, если не уверены, что оно пригодно в вашем конкретном случае.
- Если вы вынуждены прибегнуть к микротестированию, используйте JMH.
- Предоставьте свои результаты для открытого (насколько это возможно) публичного обсуждения и не забудьте обязательно обсудить их с коллегами.
- Будьте готовы к тому, чтобы быть совершенно неправым, и к тому, что ваше мнение будет неоднократно оспариваться.

Одним из позитивных аспектов работы с микротестами является то, что они демонстрируют высокодинамическое поведение и показывают не являющиеся нормальными распределения результатов, производимых низкоуровневыми системами. Это, в свою очередь, приводит к лучшему пониманию и мысленным моделям сложности JVM.

В следующей главе мы отойдем от методологии и начнем глубокое техническое погружение во внутренний мир JVM и ее основные подсистемы. Начнем изучение со сборки мусора.

[6] https://www.paypal-engineering.com/

[7] https://www.paypal-engineering.com/2016/04/11/statistics-for-software/

ГЛАВА 6

Сборка мусора

Среда Java имеет несколько знаковых функциональных возможностей, и сборка мусора — одна из наиболее узнаваемых. Однако когда платформа была выпущена впервые, наблюдалась определенная враждебность к сборке мусора. Она была вызвана тем фактом, что язык программирования Java намеренно не предоставлял никакого языкового средства для управления поведением сборщика мусора (и продолжает так поступать даже в современных версиях)[1].

Это означало, что в первые дни существования Java имелось немалое количество разочарований по поводу производительности сборки мусора в Java, и эти разочарования оказали влияние и на восприятие платформы в целом.

Однако представление об обязательной, не контролируемой пользователем сборки мусора было более чем оправданным, и в наши дни только очень немногие разработчики приложений попытаются отстоять мнение о том, что память должна управляться вручную. Даже современные языки системного программирования (например, такие как Go и Rust) рассматривают управление памятью как область ответственности компилятора и среды времени выполнения, а не программиста (кроме некоторых исключительных обстоятельств).

Суть сборки мусора в Java заключается в том, что вместо требования от программиста понимания точного времени жизни каждого объекта в системе, среда выполнения должна сама отслеживать объекты от имени программиста и автоматически избавляться от объектов, которые больше не нужны. Автоматически восстановленную память можно затем использовать повторно.

Существует два основных правила сбора мусора, которые распространяются на все реализации.

- Алгоритм должен собирать весь мусор.
- Ни один "живой" объект не должен быть собран.

Из этих двух правил второе является более важным. Сбор в качестве мусора активного используемого ("живого") объекта может привести к ошибкам сегментации

[1] Метод `System.gc()` существует, но для любых практических целей по сути бесполезен.

или (что еще хуже) к молчаливому повреждению данных программы. Алгоритмы сборки мусора в Java должны гарантировать, что они никогда не будут собирать как мусор объект, все еще используемый в программе.

Идея о том, что программист отказывается от некоторых возможностей низкоуровневого управления в обмен на то, что ему не приходится учитывать каждую низкоуровневую деталь вручную, является сутью управляемого подхода Java и выражает концепцию Джеймса Гослинга о Java как о языке "синих воротничков".

В этой главе мы познакомимся с некоторыми из основных теорий, лежащих в основе сборки мусора Java, и объясним, почему это одна из самых трудных для понимания и управления частей платформы. Мы также расскажем об основных возможностях среды выполнения системы HotSpot, включая такие детали, как способы, которыми HotSpot представляет объекты в куче во время выполнения.

К концу этой главы мы представим простейшие из сборщиков HotSpot и объясним некоторые детали, которые делают их настолько полезными для различных рабочих нагрузок.

Алгоритм маркировки и выметания

Большинство программистов на языке программирования Java вспомнят, что сборка мусора в Java опирается на алгоритм *маркировки и выметания* (mark and sweep[2]), но большинство из них не смогут вспомнить какие-либо подробности о том, как он работает на самом деле.

В этом разделе мы расскажем о базовой разновидности этого алгоритма и покажем, как он может использоваться для автоматического освобождения памяти. Это намеренно упрощенная разновидность алгоритма, предназначенная только для того, чтобы ввести несколько основных концепций, и не являющаяся репрезентативным представлением о том, как промышленные JVM в действительности выполняют сборку мусора.

Эта вводная разновидность алгоритма маркировки и выметания использует выделенный список объектов для хранения указателей на каждый выделенный, но еще не очищенный объект. Общий алгоритм сборки мусора может в этом случае быть выражен следующим образом.

1. Цикл по списку выделения со сбросом бита метки.
2. Поиск живых объектов, начиная от корня сборки мусора.
3. Установка бита метки у каждого достигнутого объекта.
4. Цикл по списку выделения, в котором для каждого объекта, бит метки которого не был установлен, выполняется следующее:

[2] Дословный перевод с английского — "пометить и вымести". — *Примеч. пер.*

а) освобождение памяти в куче и помещение ее в список свободной памяти;

б) удаление объекта из списка выделения.

Живые объекты обычно располагаются в глубине, и полученный граф называется *графом живых объектов* (live object graph). Его иногда также называют *транзитивным замыканием достижимых объектов* (transitive closure of reachable objects), и соответствующий пример можно увидеть на рис. 6.1.

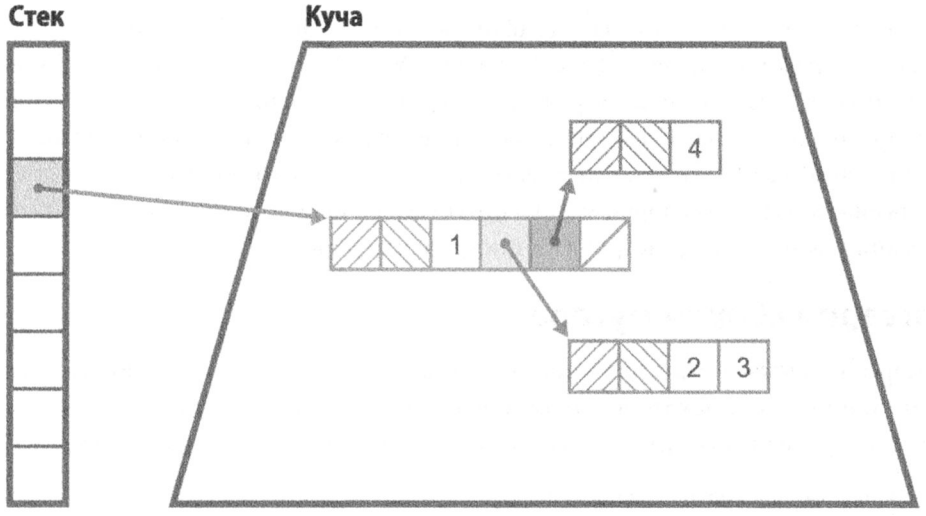

Рис. 6.1. Простое представление схемы памяти

Состояние кучи может быть трудно визуализировать, но, к счастью, есть некоторые простые инструменты, способные нам помочь. Одним из простейших является утилита командной строки `jmap -histo`. Эта утилита показывает количество байтов, выделенных типу, и количество экземпляров, которые коллективно отвечают за это использование памяти. Ее вывод имеет примерно следующий вид:

```
num #instances #bytes class name
----------------------------------------------
 1:   20839  14983608  [B
 2:  118743  12370760  [C
 3:   14528   9385360  [I
 4:     282   6461584  [D
 5:  115231   3687392  java.util.HashMap$Node
 6:  102237   2453688  java.lang.String
 7:   68388   2188416  java.util.Hashtable$Entry
 8:    8708   1764328  [Ljava.util.HashMap$Node;
 9:   39047   1561880  jdk.nashorn.internal.runtime.CompiledFunction
10:   23688   1516032  com.mysql.jdbc.Co...$BooleanConnectionProperty
11:   24217   1356152  jdk.nashorn.internal.runtime.ScriptFunction
12:   27344   1301896  [Ljava.lang.Object;
```

```
13:   10040   1107896  java.lang.Class
14:   44090   1058160  java.util.LinkedList$Node
15:   29375    940000  java.util.LinkedList
16:   25944    830208  jdk.nashorn.interna...FinalScriptFunctionData
17:      20    655680  [Lscala.concurrent.forkjoin.ForkJoinTask;
18:   19943    638176  java.util.concurrent.ConcurrentHashMap$Node
19:     730    614744  [Ljava.util.Hashtable$Entry;
20:   24022    578560  [Ljava.lang.Class;
```

Имеется также инструмент с графическим интерфейсом: вкладка Sampling у VisualVM, представленная в главе 2, "Обзор JVM". Плагин VisualGC для VisualVM обеспечивает представление в режиме реального времени о том, как изменяется куча, но в общем случае представления кучи в разные моменты времени недостаточно для точного анализа, так что вместо этого для лучшего понимания мы должны использовать журналы сборки мусора (это будет основной темой главы 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора").

Глоссарий сборки мусора

Жаргон, используемый для описания алгоритмов сборки мусора, иногда немного запутан (а смысл некоторых терминов со временем изменялся). Для ясности мы включим в материал главы базовый глоссарий, поясняющий основные термины.

Остановка мира (Stop-the-world — STW)

Цикл сборки мусора требует, чтобы все потоки приложений были приостановлены на время сбора. Это предотвращает ситуации, когда код приложения делает некорректным состояние кучи, видимое потоком сборщика мусора. Это обычная для большинства простых алгоритмов сборки мусора ситуация.

Параллельность (Concurrent)

Потоки сборки мусора могут выполняться во время работы потоков приложения. Достичь этого очень, очень сложно, и к тому же это очень дорого с точки зрения стоимости вычислений. Практически никакие алгоритмы не являются действительно параллельными. Вместо этого используются различные сложные трюки для получения большинства преимуществ одновременной сборки мусора. В разделе "CMS" главы 7, "Вглубь сборки мусора", мы встретимся со сборщиком мусора Concurrent Mark and Sweep (CMS) из HotSpot, который, несмотря на его название, лучше всего описывается как "в основном параллельный" сборщик мусора.

Многопоточность (Parallel)

Для выполнения сборки мусора используется несколько потоков выполнения.

Точность (Exact)

Точная схема сборки мусора имеет достаточно информации о типах при рассмотрении состояния кучи, так что она в состоянии обеспечить сбор всего мусора за один цикл. Попросту говоря, точная схема обладает тем свойством, что всегда может отличить `int` от указателя.

Консервативность (Conservative)

В консервативной схеме отсутствует информация о точной схеме. В результате консервативные схемы часто "транжирят" ресурсы и, как правило, намного менее эффективны из-за их фундаментального незнания системы типов, которую они должны представлять.

Перемещение (Moving)

В перемещающем сборщике мусора объекты могут быть перенесены в памяти. Это означает, что они не имеют стабильного адреса. Среды, которые обеспечивают доступ к простым указателям (например, C++), непригодны для применения перемещающих сборщиков мусора.

Уплотнение (Compacting)

В конце цикла сборки мусора выделенная память (т.е. выжившие объекты) оказывается организованной как единая смежная область памяти, и имеется указатель, указывающий на начало пустого пространства, которое доступно для записи объектов. Уплотняющий сборщик мусора избегает фрагментации памяти.

Эвакуация (Evacuating)

В конце цикла сборки мусора обработанная область памяти полностью пуста, а все живые объекты полностью перемещены (эвакуированы) в другую область памяти.

В большинстве других языков и сред используются те же термины. Например, среда времени выполнения JavaScript веб-браузера Firefox (SpiderMonkey) также позволяет использовать сборку мусора, и в последние годы в нее добавлены функциональные возможности (например, точность и уплотнение), которые присутствуют в реализациях сборки мусора в Java.

Введение в среду времени выполнения HotSpot

В дополнение к общей терминологии сборки мусора HotSpot вводит термины, более специфичные для данной реализации. Чтобы получить полное представление о том, как работает сборка мусора на этой JVM, нам нужно получить представление о некоторых деталях внутреннего устройства HotSpot.

Для начала будет полезно напомнить, что Java имеет значения только двух видов:

- примитивные типы (`byte`, `int` и т.п.);
- ссылки на объекты.

Многие программисты Java слишком вольно используют термин *объект*, но для наших целей важно помнить, что, в отличие от C++, Java не имеет общего механизма разыменования адресов и может использовать только *оператор смещения* (оператор `.`) для доступа к полям и вызова методов с использованием *ссылок на объекты*. Также помните, что семантика вызова метода в Java использует только передачу по значению, хотя для ссылок на объекты это означает, что копируемое значение является адресом объекта в куче.

Представление объектов во время выполнения

HotSpot представляет объекты Java во время выполнения с помощью структуры, именуемой в английском сообществе программистов *oop*. Это название является сокращением от *ordinary object pointer* (обычный указатель на объект) и представляет собой подлинный указатель в смысле языка программирования C. Эти указатели могут быть помещены в локальные переменные ссылочного типа; они указывают из кадра стека метода Java в область памяти, содержащую кучу Java.

Имеется несколько разных структур данных, которые составляют семейство обычных указателей на объекты, а разновидность, представляющая экземпляры классов Java, называется *instanceOop*.

Схема расположения памяти в *instanceOop* начинается с двух машинных слов заголовка, присутствующих у каждого объекта. Первым из них идет *слово метки* (mark word), которое является указателем, указывающим на метаданные, специфичные для конкретных экземпляров. Далее следует *слово класса* (klass word), которое указывает на метаданные уровня класса.

В Java 7 и более ранних версий слово класса экземпляра *instanceOop* указывает на область памяти, именуемую *PermGen*, которая была частью кучи Java. Общее правило заключается в том, что в куче Java все должно иметь заголовок объекта. В этих более старых версиях Java мы ссылаемся на метаданные как на *klassOop*. Схема памяти *klassOop* проста — это просто заголовок объекта, непосредственно за которым следуют метаданные класса.

Начиная с Java 8, классы хранятся за пределами основной части кучи Java (но не за пределами кучи C процесса JVM). Слова классов в этих версиях Java не требуют заголовка объекта, поскольку указывают на область вне кучи Java.

Буква *k* в начале слова используется для того, чтобы помочь не путать klassOop и instanceOop, представляющий собой объект `Class<?>` Java; это не одно и то же.

На рис. 6.2 мы видим разницу: по сути, klassOop содержит таблицу виртуальных функций (vtable) для класса, в то время как объект `Class` содержит массив ссылок на объекты `Method` для использования при рефлексивном вызове. Мы поговорим об этом в главе 9, "Выполнение кода в JMV", когда будем обсуждать JIT-компиляцию.

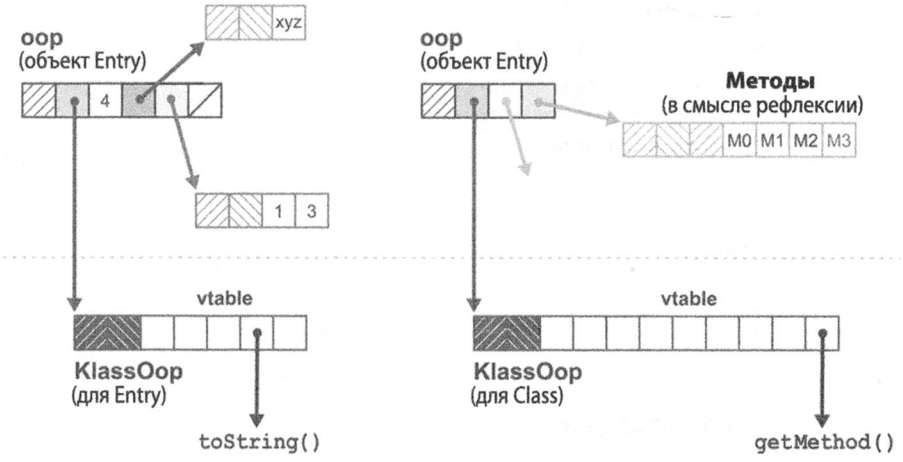

Рис. 6.2. Объекты klassOops и Class

Обычные указатели на объекты, как правило, представляют собой машинные слова, поэтому их размер — 32 бита на 32-разрядной машине и 64 бита на современном процессоре. Тем не менее это потенциально может вести к потере значительного объема памяти. В связи с этим HotSpot предоставляет методику, именуемую *сжатым обычным указателем на объект*. Если установлен флаг

```
-XX:+UseCompressedOops
```

(это значение по умолчанию для 64-разрядной кучи, начиная с Java 7 и выше), то следующие обычные указатели на объекты в куче будут сжаты:

- слово класса каждого объекта в куче;
- поля экземпляра ссылочного типа;
- каждый элемент массива объектов.

В общем случае это означает, что заголовок объекта HotSpot состоит из:

- слова метки полного машинного размера;
- слова класса (возможно, сжатого);
- слова длины, если объект представляет собой массив — всегда 32 бита;
- 32-битовый зазор (если требуется правилами выравнивания).

Сразу же после заголовка следуют поля экземпляра объекта. Для klassOop после слова класса следует таблица виртуальных функций. Распределение памяти для сжатых обычных указателей на объекты можно увидеть на рис. 6.3.

Рис. 6.3. Сжатые обычные указатели на объекты

В прошлом некоторые приложения с высокой чувствительностью к задержкам иногда добивались улучшения при отключении функции сжатия обычных указателей на объекты — за счет увеличения размера кучи (обычно на 10–50%). Однако класс приложений, для которых это действие принесет ощутимые преимущества в производительности, очень мал, и для большинства современных приложений это будет классический пример антипаттерна "Надувательство с переключателями", с которым мы встречались в главе 4, "Паттерны и антипаттерны тестирования производительности".

Как мы помним из основ Java, массивы являются объектами. Это означает, что массивы JVM также представлены как обычные указатели на объекты. Вот почему в массивах, помимо обычных слов метки и класса, есть третье слово метаданных. Это третье слово — длина массива, что объясняет, почему индексы массива в Java ограничены 32-битовыми значениями.

Использование дополнительных метаданных для переноса длины массива облегчает целый класс проблем, имеющихся на C и C++, в которых отсутствие информации о длине массива означает необходимость передачи в функции дополнительного параметра.

Управляемая среда JVM не позволяет Java ссылаться на точку где угодно, кроме instanceOop (или `null`). Это означает, что на низком уровне:

- значение Java представляет собой набор битов, соответствующий либо примитивному значению, либо адресу instanceOop (ссылке);
- любая ссылка Java рассматривается как указатель, ссылающийся на адрес в основной части кучи Java;
- целевые адреса ссылок Java содержат слова меток, за которыми следуют слова классов — в качестве следующего машинного слова;
- klassOop и экземпляр `Class<?>` различаются (первый располагается в области метаданных кучи), и klassOop не может быть помещен в переменную Java.

HotSpot определяет иерархию oops в `.hpp`-файлах, хранящихся в каталоге `hotspot/src/share/vm/oops` в дереве исходных файлов OpenJDK 8. Общая иерархия наследования обычных указателей на объекты выглядит следующим образом:

```
oop (абстрактный базовый класс)
    instanceOop    (объекты экземпляров)
    methodOop      (представление методов)
    arrayOop       (абстрактный базовый класс для массиов)
    symbolOop      (внутренний класс символов/строк)
    klassOop       (заголовок классов) (только Java 7 и более ранние)
    markOop
```

Это использование структур обычных указателей на объекты для представления объектов во время выполнения, с одним указателем на размещенные метаданные уровня класса и другим — на размещенные метаданные экземпляра, не столь уж необычное. Многие другие JVM и другие среды выполнения используют подобный механизм. Например, аналогичную схему для представления объектов использует iOS от Apple.

Корни и арены сборки мусора

Статьи и сообщения в блогах о HotSpot часто ссылаются на *корни сборки мусора* (GC roots). Это "опорные точки" для памяти, по существу — известные указатели, которые исходят извне относительно интересующего нас внешнего пула памяти и указывают в него. Они являются *внешними* указателями, в противоположность внутренним указателям, которые исходят из пула памяти и указывают на другое место памяти в этом же пуле.

Примеры корней сборки мусора мы видели на рис. 6.1. Однако, как мы вскоре увидим, имеются и другие разновидности корней сборки мусора, включая следующие:

- кадры стека;
- • JNI;
- регистры (для случаев перемещенных переменных);
- корни кода (из кеша кода JVM);
- глобальные переменные;
- метаданные класса загруженных классов.

Если это определение кажется слишком сложным, то простейшим примером корня сборки мусора является локальная переменная ссылочного типа, которая всегда указывает на объект в куче (при условии, что она не является `null`).

Сборщик мусора HotSpot работает в терминах областей памяти, называемых *аренами*. Это очень низкоуровневый механизм, и разработчикам Java, как правило, не требуется столь подробно рассматривать работу системы памяти. Однако специалистам по производительности иногда приходится закапываться во внутренние структуры JVM, поэтому знакомство с понятиями и терминами, используемыми в литературе, будет полезным.

Один важный факт, о котором следует помнить, заключается в том, что HotSpot не использует системные вызовы для управления кучей Java. Вместо этого, как говорилось в разделе "Основные стратегии обнаружения источников проблем" главы 3, "Аппаратное обеспечение и операционные системы", HotSpot управляет размером кучи из кода пользовательского пространства, так что мы можем использовать простые наблюдения, чтобы определить, вызывает ли подсистема сборки мусора какие-либо проблемы производительности.

В следующем разделе мы рассмотрим две наиболее важные характеристики, которые управляют поведением сборки мусора Java или JVM. Знание этих характеристик весьма важно для любого разработчика, который хочет действительно понимать факторы, управляющие сборкой мусора Java (которая является одним из ключевых факторов производительности Java).

Выделение памяти и время жизни

Существует два основных фактора, управляющих поведением сборки мусора приложения Java:

- скорость выделения памяти;
- время жизни объектов.

Скорость выделения памяти (allocation rate) представляет собой количество памяти, используемое вновь созданными объектами за некоторый период времени (обычно измеряется в мегабайтах в секунду). Этот параметр непосредственно JVM не записывается, но его относительно легко оценить, а такие инструменты, как Censum, могут определить его точное значение.

И напротив, измерить (или даже оценить) время жизни объекта обычно намного сложнее. Фактически одним из основных аргументов против использования ручного управления памятью является сложность, связанная с истинным пониманием времени жизни объекта в реальном приложении. В результате время жизни объекта является чем-то даже более фундаментальным, чем скорость распределения.

Сборку мусора также можно рассматривать как "восстановление и повторное использование памяти". Возможность использовать один и тот же физический фрагмент памяти снова и снова, потому что объекты недолговечны, является ключевым положением методов сборки мусора.

Идея о том, что объекты создаются и существуют какое-то время, а затем память, используемая для хранения их состояния, может быть восстановлена, представляет собой основу сборки мусора, без которой сборка мусора вообще не могла бы работать. Как мы увидим в главе 7, "Вглубь сборки мусора", существует ряд различных компромиссов, на которые сборщики мусора должны идти, и некоторые наиболее важные из этих компромиссов обусловлены проблемами жизненного цикла и распределения.

Слабая гипотеза поколений

Один из ключевых элементов управления памятью JVM основан на наблюдаемом эффекте времени выполнения программных систем — слабой гипотезе поколений (Weak Generational Hypothesis):

> *Распределение времени жизни объектов в JVM и других подобных программных системах является бимодальным — с подавляющим большинством очень недолговечных объектов; вторая популяция объектов имеет гораздо более продолжительную жизнь.*

Эта гипотеза (которая на самом деле представляет собой экспериментально наблюдаемое эмпирическое правило поведения объектно-ориентированных рабочих нагрузок) приводит к очевидному выводу: кучи, в которых выполняется сборка мусора, должны быть структурированы таким образом, чтобы можно было легко и быстро собирать короткоживущие объекты, и в идеале долгоживущие объекты должны быть отделены от короткоживущих.

HotSpot использует несколько механизмов в попытках воспользоваться преимуществами слабой гипотезы поколений.

- Она отслеживает "количество поколений" для каждого объекта (количество сборок мусора, которые до настоящего времени пережил объект).
- За исключением больших объектов, она создает новые объекты в пространстве "Эдем" (также именуемом "Ясли") и планирует перемещение выживших объектов.
- Она поддерживает отдельную область памяти ("старое" или "бессрочное" поколение) для хранения объектов, которые считаются прожившими достаточно долго для того, чтобы быть признанными долгожителями.

Такой подход приводит к виду, показанному в упрощенной форме на рис. 6.4, где объекты, пережившие определенное количество циклов сборки мусора, продвигаются к бессрочному поколению. Обратите внимание на непрерывный характер областей, показанных на диаграмме.

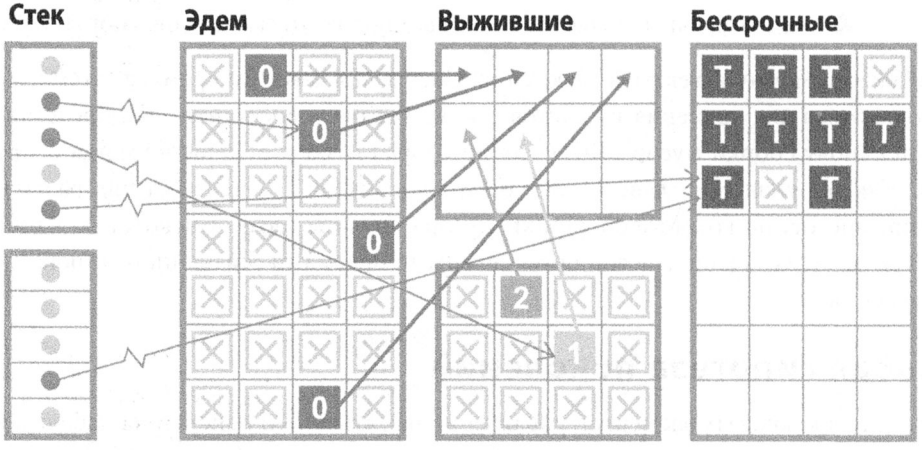

Рис. 6.4. Коллекция поколений

Разделение памяти на разные области для целей сбора с учетом поколений имеет некоторые дополнительные последствия с точки зрения того, как HotSpot реализует сборку с использованием алгоритма маркировки и выметания. Одна важная методика заключается в отслеживании указателей, указывающих извне на молодое поколение. Это спасает цикл сборки мусора от обхода всего графа объектов в поисках все еще живых молодых объектов.

В качестве второй части слабой гипотезы поколений иногда используется "Имеется несколько ссылок из старых объектов на молодые".

Чтобы облегчить этот процесс, HotSpot поддерживает структуру, именуемую *таблицей карт* (card table), которая помогает записывать, какие объекты старого

поколения потенциально могут указывать на молодые объекты. Таблица карт представляет собой массив байтов, управляемых JVM. Каждый элемент массива соответствует 512-байтовой области пространства старого поколения.

Основная идея заключается в том, что, когда поле ссылочного типа в старом объекте o изменяется, запись в таблице карт для карты, содержащей instanceOop, соответствующий o, помечается как "грязная". HotSpot достигает этого с помощью простого *барьера записи* (write barrier) при обновлении полей ссылок. Это, по существу, сводится к выполнению после сохранения поля следующего фрагмента кода:

```
cards[*instanceOop >> 9] = 0;
```

Обратите внимание, что "грязное" значение карты равно 0, а сдвиг вправо на 9 бит дает размер таблицы карты, равный 512 байтам.

Наконец, следует отметить, что описание кучи в терминах старой и молодой областей исторически является способом управления памятью сборщиками мусора Java. С появлением Java 8u40 новый сборщик мусора ("Garbage First", или G1) достиг промышленного качества. G1 имеет несколько иное представление о схеме кучи, как мы увидим в разделе "CMS" главы 7, "Вглубь сборки мусора". Этот новый подход к управлению кучей будет приобретать все большее и большее значение, поскольку намерение Oracle состоит в том, чтобы G1 стал сборщиком мусора по умолчанию, начиная с Java 9 и далее.

Сборка мусора в HotSpot

Напомним, что в отличие от C/C++ и других подобных сред, Java не использует операционную систему для управления динамической памятью. Вместо этого JVM выделяет (или резервирует) память, когда запускается процесс JVM, и управляет единым непрерывным пулом памяти из пользовательского пространства.

Как мы видели, этот пул памяти состоит из разных областей с отдельным предназначением, и адрес, по которому находится объект, очень часто изменяется со временем, когда сборщик мусора перемещает объекты, обычно создаваемые в области "Эдем". Сборщики мусора, которые выполняют перемещение, известны как "эвакуирующие", как указано в разделе "Глоссарий сборки мусора". Многие из сборщиков, которые может использовать HotSpot, являются эвакуирующими.

Выделение памяти, локальное для потока

JVM использует усовершенствование производительности при управлении областью "Эдем". Эта область критична для эффективного управления, так как именно там создается большинство объектов, а очень короткоживущие объекты (те, у которых время жизни меньше, чем оставшееся до следующего цикла сборки мусора время) никогда не будут находиться где-либо еще.

Для повышения эффективности JVM разбивает область "Эдем" на буфера и выделяет отдельные подобласти для потоков приложений для использования в качестве областей для новых объектов. Преимущество такого подхода состоит в том, что каждый поток знает, что ему не нужно учитывать возможность выделения памяти в этом буфере другими потоками. Такие регионы называются *буферами выделения памяти, локальными для потока* (thread-local allocation buffers— TLAB).

HotSpot динамически изменяет размеры TLAB, которые он предоставляет потокам приложения. Поэтому, если поток активно использует память, ему могут быть предоставлены бо́льшие по размеру TLAB, чтобы уменьшить накладные расходы при предоставлении памяти потоку.

Исключительный контроль, который поток приложения получает над своими TLAB, означает, что выделение для потоков JVM выполняется за время O(1). Это связано с тем, что, когда поток создает новый объект, для него выделяется необходимая память, а локальный указатель потока при этом обновляется до следующего свободного адреса памяти. В терминах системы времени выполнения C это простой указатель, т.е. для его перемещения вперед достаточно одной дополнительной команды.

Это поведение можно увидеть на рис. 6.5, на котором каждый поток приложения хранит буфер для выделения новых объектов. Если поток приложения заполняет буфер, JVM предоставляет указатель на новую область Эдема.

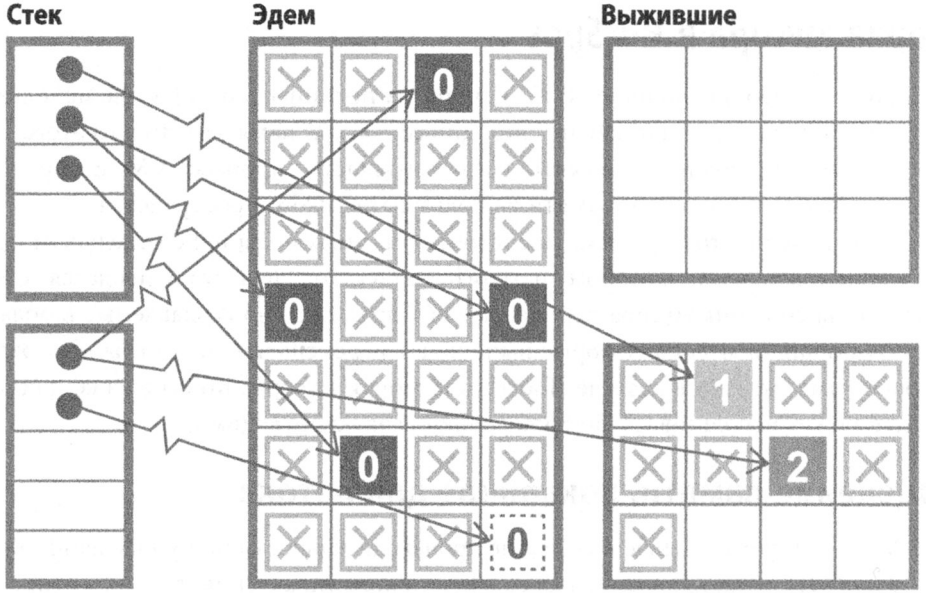

Рис. 6.5. Выделение памяти, локальное для потока

Полусферическая сборка

Следует отметить один особый случай эвакуирующего сборщика. Иногда его называют *полусферическим эвакуирующим сборщиком мусора* (hemispheric evacuating collector). Этот тип сборщика мусора использует два (обычно равных) пространства. Основная идея состоит в том, чтобы использовать эти пространства как временную область хранения для объектов, на самом деле недолговечных. Тем самым предотвращается загромождение бессрочного поколения короткоживущими объектами и уменьшается частота полных сборок мусора. Эти пространства обладают парой основных свойств.

- Когда сборщик мусора обрабатывает текущее "живое" полушарие, объекты с уплотнением перемещаются в другое полушарие, а обработанное полушарие опустошается для повторного использования.
- Половина пространства все время поддерживается в пустом состоянии.

Разумеется, этот подход использует вдвое больше памяти, чем может фактически храниться в одной полусферической части сборщика. Это несколько расточительно, но часто это оказывается полезным методом, если размер этих пространств является не чрезмерным. HotSpot использует такой полусферный подход для создания сборщика для молодого поколения в сочетании с пространством "Эдем".

Полусферическая часть "молодой" кучи HotSpot называется *пространством выживших* (survivor spaces). Как можно видеть из представленного на рис. 6.6 результата работы VisualGC, это пространство, как правило, относительно мало по сравнению с Эдемом, а роль пространства выживших меняется при каждой сборке молодых поколений. Как настроить размер пространства выживших, будет рассмотрено в главе 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора".

Плагин VisualGC для VisualVM, о котором упоминалось в разделе "VisualVM" главы 2, "Обзор JVM", является очень полезным инструментом для отладки сборки мусора. Как мы обсудим в главе 7, "Вглубь сборки мусора", журналы сборки мусора содержат гораздо более полезную информацию и позволяют выполнить гораздо более глубокий анализ сборки, чем это возможно из данных, получаемых в JMX от момента времени к моменту, которые использует VisualGC. Однако, приступая к новому анализу, зачастую полезно просто наблюдать за использованием памяти приложением.

С помощью VisualGC можно увидеть некоторое совокупное воздействие сборки мусора, такое как перемещенные в куче объекты, или переключение пространства выживших, которое происходит при каждой сборке молодого поколения.

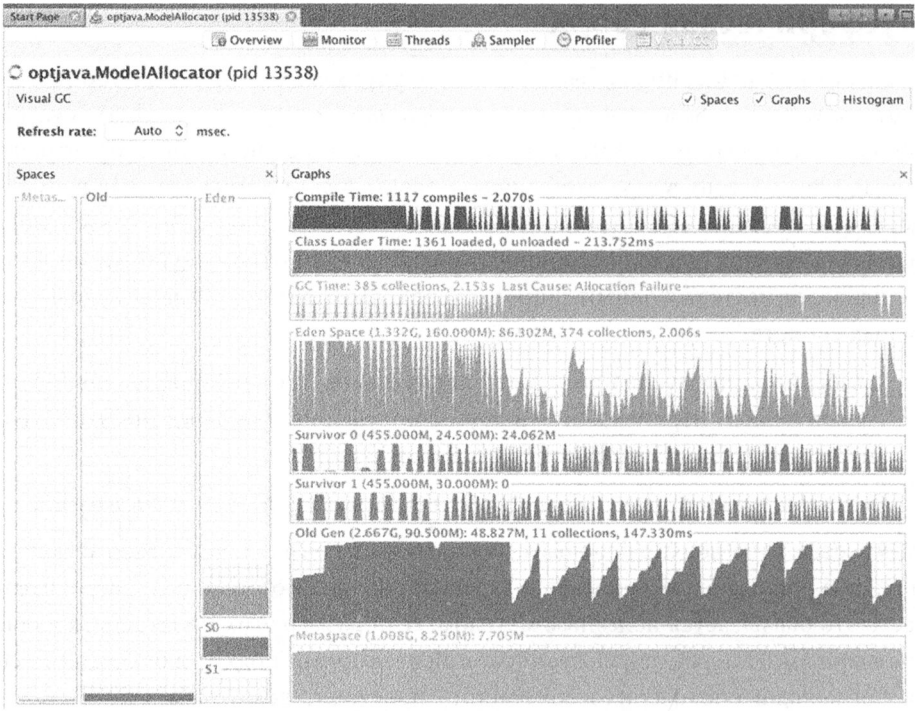

Рис. 6.6. Плагин VisualGC

Многопоточные сборщики мусора

В Java 8 и более ранних версиях сборщики мусора JVM по умолчанию являются параллельными сборщиками. Они выполняют полную "остановку мира" для сборок (как молодой, так и полной) и оптимизированы для достижения максимальной пропускной способности. После остановки всех потоков приложения многопоточные сборщики мусора используют все доступные ядра процессора, чтобы выполнить сборку мусора как можно быстрее. Доступными многопоточными сборщиками мусора являются следующие.

Parallel GC

Простейший сборщик для молодого поколения.

ParNew

Небольшая вариация Parallel GC, используемая со сборщиком CMS.

ParallelOld

Многопоточный сборщик мусора для старого (бессрочного) поколения.

Многопоточные сборщики мусора в определенном отношении аналогичны один другому — они разработаны таким образом, чтобы использовать несколько потоков для как можно более быстрой идентификации "живых" объектов с минимальными накладными расходами. Однако между ними имеются и некоторые различия, так что давайте посмотрим на две основные разновидности сборки.

Многопоточная сборка юного поколения

Наиболее распространенным типом сборки мусора является сборка молодых поколений. Это обычно происходит, когда поток пытается выделить память для объекта в Эдеме, но не имеет достаточно свободного места в его TLAB, так что JVM не может выделить свежий TLAB для потока. Когда это происходит, у JVM нет иного выбора, кроме как остановить все потоки приложения, — потому что, если один поток не может выделить память, очень скоро это будет с каждым потоком.

Потоки могут также выделять память и вне TLAB (например, для больших блоков памяти). Однако желательно, чтобы скорость выделения памяти вне TLAB была низкой.

После того, как остановлены все потоки приложения (или пользователя), HotSpot просматривает молодое поколение (которое определяется как пространство Эдем и непустое в настоящий момент пространство выживших) и определяет все объекты, которые не являются мусором. Для этого в качестве отправной точки для параллельной маркировки используются корни сборки мусора (и таблица карт для определения корней сборки мусора из старого поколения).

Сборщик мусора Parallel GC затем перемещает (эвакуирует) все сохранившиеся объекты в пустое в настоящее время пространство выживших (и при перемещении увеличивает их счетчик поколений). Наконец, Эдем и только что эвакуированное пространство выживших помечаются как пустое пространство, пригодное для повторного использования. Потоки приложения запускаются, так что процесс раздачи TLAB потокам приложения может начаться заново. Этот процесс показан на рис. 6.7 и 6.8.

Этот подход пытается воспользоваться слабой гипотезой поколений, работая только с живыми объектами. Он также пытается быть максимально эффективным и работать с использованием всех возможных ядер, чтобы насколько это возможно сократить время остановки потоков приложения.

Старые многопоточные сборки мусора

Сборщик ParallelOld в настоящее время (по состоянию на Java 8) является сборщиком по умолчанию для старого поколения. Он имеет как сильное сходство с Parallel GC, так и ряд фундаментальных различий. В частности, Parallel GC является

полусферическим эвакуирующим сборщиком мусора, тогда как ParallelOld является уплотняющим сборщиком с единым непрерывным пространством памяти.

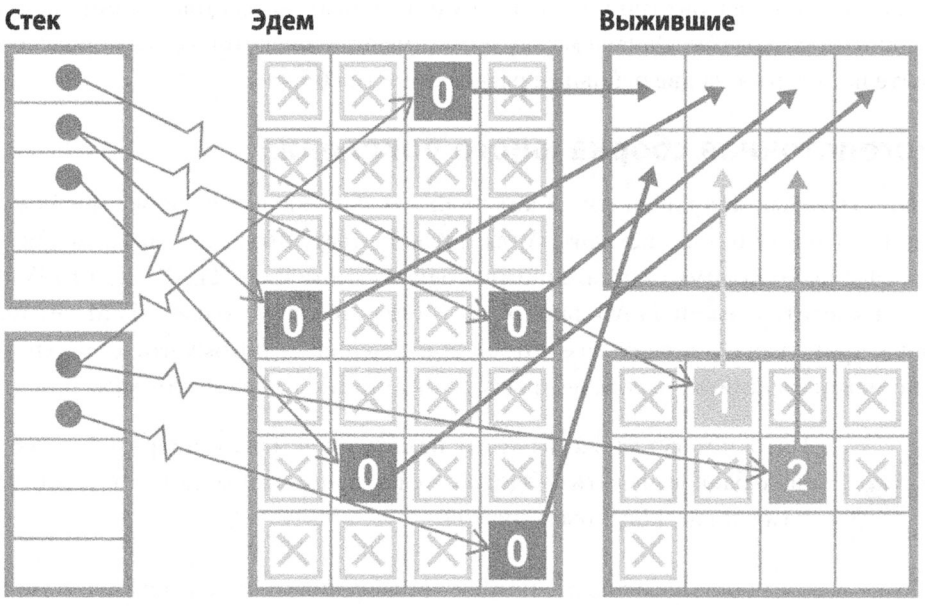

Рис. 6.7. Сборка молодого поколения

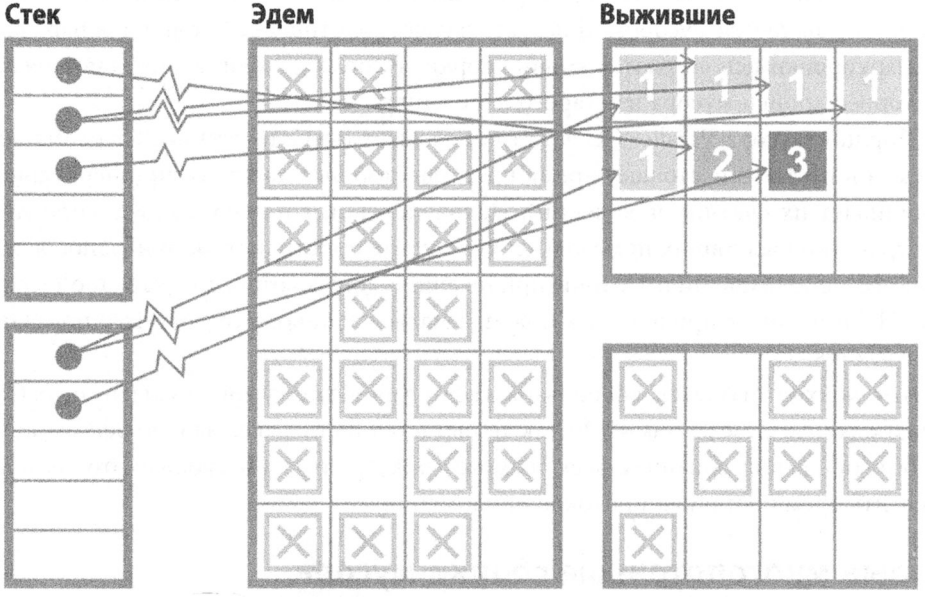

Рис. 6.8. Эвакуация молодого поколения

Это означает, что когда нет места для эвакуации старого поколения, многопоточный сборщик пытается переместить объекты внутри старого поколения для освобождения пространства, которое могло остаться после гибели старых объектов. Таким образом, потенциально сборщик может быть очень эффективным в своем использовании памяти и не будет страдать от ее фрагментации.

Это приводит к очень эффективной схеме памяти за счет использования потенциально большого количества процессорного времени в течение полных циклов сборки мусора. Разницу между этими двумя подходами можно увидеть на рис. 6.9.

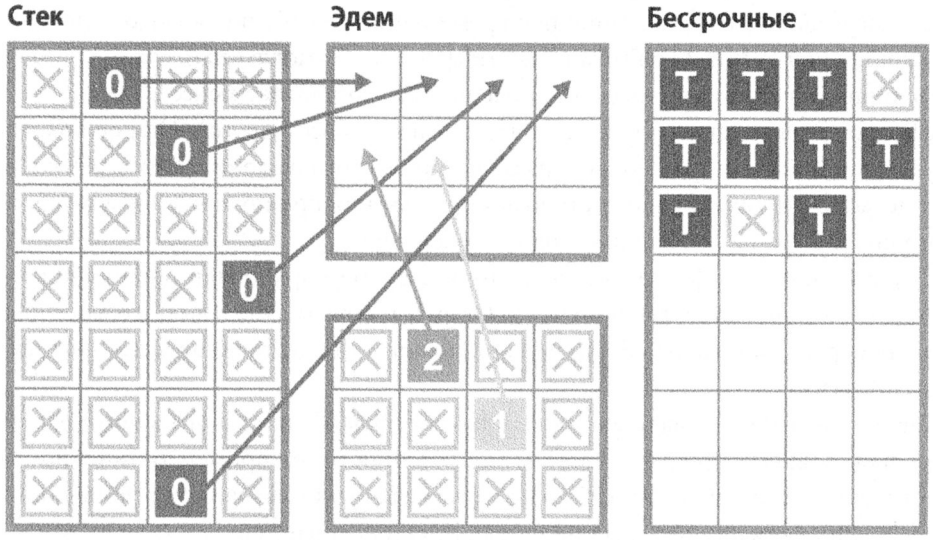

Рис. 6.9. Эвакуация и уплотнение

Поведение этих двух пространств памяти достаточно сильно различается, так как они служат разным целям. Цель сборки молодого поколения — работа с короткоживущими объектами, поэтому занятость молодого пространства радикально меняется с выполнением распределения и очистки во время событий сборки мусора.

Напротив, старое пространство так очевидно не изменяется. Иногда большие объекты будут создаваться непосредственно в бессрочной области, но независимо от этого пространство будет изменяться только во время сборок — либо за счет объектов из молодого поколения, либо путем полного сканирования и перегруппировки во время сборки старого поколения и полной сборки мусора.

Ограничения многопоточных сборщиков

Многопоточные сборщики работают со всем содержимым поколения сразу и пытаются выполнить сборку максимально эффективно. Однако этот дизайн имеет и некоторые недостатки. Во-первых, они полностью останавливают работу приложения.

Обычно это не является проблемой для молодых коллекций, так как слабая гипотеза поколений означает, что выжить должны очень немногие объекты.

Дизайн многопоточных сборщиков молодых поколений таков, что они никогда не работают с "мертвыми" объектами, так что длина этапа маркировки пропорциональна (малому) количеству живых объектов.

Этот базовый дизайн в сочетании с обычно малыми размерами молодых областей кучи означает, что время пауз при сборке молодых поколений для большинства рабочих нагрузок очень мало. Типичное время паузы для сборки молодого поколения на современной JVM с 2 Гбайтами памяти (размер по умолчанию) может составлять всего лишь несколько миллисекунд и очень часто составляет менее 10 мс.

Однако сборка старого поколения часто бывает совсем другой. Старое поколение по умолчанию в семь раз больше размера молодого поколения. Один только этот факт делает ожидаемую длину остановки приложения при полной сборке мусора намного больше, чем для сборки молодого поколения.

Другим ключевым фактом является то, что время маркировки пропорционально количеству живых объектов в области. Старые объекты могут быть долговечными, поэтому пережить полную сборку мусора может потенциально большее количество старых объектов.

Такое поведение объясняет также ключевую слабость старого многопоточного сборщика — время приостановки им приложения будет примерно линейно расти с увеличением размера кучи. Поскольку размеры кучи постоянно увеличиваются, с точки зрения времени паузы сборщик ParallelOld начинает сильно ухудшаться.

Новички в теории сборки мусора иногда увлекаются теориями, что незначительные модификации алгоритмов маркировки и выметания могут помочь снизить время паузы приложения. Однако это не так. Сборка мусора была очень активно изучаемой областью информатики в последние более чем 40 лет, и никакого улучшения "можно просто..." так никогда и не было найдено.

Как мы увидим в главе 7, "Вглубь сборки мусора", в настоящее время в основном используются многопоточные сборщики мусора, которые могут значительно сокращать время пауз. Однако они не являются панацеей, так что определенные фундаментальные трудности со сборкой мусора все равно остаются.

В качестве примера одной из центральных трудностей простейшего подхода к сборке мусора рассмотрим распределение TLAB. Оно значительно повышает производительность распределения, но не помогает уменьшить время циклов сборки мусора. Чтобы понять, почему это так, рассмотрим следующий код:

```
public static void main(String[] args)
{
    int[] anInt = new int[1];
    anInt[0] = 42;
```

```
    Runnable r = () ->
    {
        anInt[0]++;
        System.out.println("Changed: " + anInt[0]);
    };
    new Thread(r).start();
}
```

Переменная `anInt` представляет собой объект массива, содержащий единственное значение `int`. Память для нее выделяется из TLAB основного потока, но сразу же после этого она передается в новый поток. Другими словами, ключевое свойство TLAB, что эти буфера являются частными по отношению к своему потоку, истинно только в точке выделения памяти. Это свойство может быть нарушено тут же, как только объект выделен.

Способность Java-среды тривиально создавать новые потоки является фундаментальной и чрезвычайно мощной частью платформы. Однако это значительно усложняет картину сборки мусора, поскольку новые потоки приводят к созданию стеков выполнения, каждый кадр которых является источником корней сборки мусора.

Роль выделения памяти

Процесс сборки мусора Java чаще всего запускается при запросе выделения памяти, когда свободной памяти для запрашиваемого выделения недостаточно. Это означает, что циклы сборки мусора выполняются не согласно некоторому фиксированному или предсказуемому графику, а исключительно по мере необходимости.

Это один из самых важных аспектов сборки мусора: он не детерминирован и не выполняется регулярным образом. Вместо этого цикл сборки мусора запускается тогда, когда одно или несколько пространств памяти кучи, по сути, оказываются заполненными, и дальнейшее создание объектов становится невозможным.

Этот характер работы "по необходимости" делает журналы сборки мусора сложными для обработки с использованием традиционных методов анализа временных рядов. Отсутствие регулярности событий сборки мусора является аспектом, который большинство библиотек для работы с временными рядами не в состоянии легко обработать.

Когда происходит сборка мусора, все потоки приложений приостанавливаются (поскольку они не могут создавать новые объекты, а никакая существенная часть кода Java не может долго работать без создания новых объектов). JVM занимает все ядра процессора выполнением сборки мусора и восстанавливает память перед перезапуском потоков приложений.

Чтобы лучше понять, почему выделение памяти настолько критично, давайте рассмотрим следующую очень упрощенную ситуацию. Параметры кучи настроены так,

как показано далее, и мы предполагаем, что со временем они не меняются. Конечно, реальное приложение обычно имеет динамически изменяющуюся кучу, но данный пример служит не более чем упрощенной иллюстрацией.

Область кучи	Размер
Вся память	2 Гбайт
Старое поколение	1.5 Гбайт
Молодое поколение	500 Мбайт
Эдем	400 Мбайт
S1	50 Мбайт
S2	50 Мбайт

После достижения приложением устойчивого состояния наблюдаются следующие метрики сборки мусора:

```
Скорость выделения памяти              100 Мбайт/с
Время сборки молодого поколения          2 мс
Полное время сборки                    100 мс
Время жизни объекта                    200 мс
```

Отсюда видно, что Эдем будет заполнен за 4 с, поэтому в устойчивом состоянии сборка молодого поколения будет происходить каждые 4 с. Эдем заполняется, и запускается сборка мусора. Большинство объектов в Эдеме мертвы, но любой объект, который все еще жив, будет эвакуирован в пространство выживших (SS1 для определенности). В этой простой модели любые объекты, созданные за последние 200 мс, еще не успели умереть, поэтому они выживут. Итак, мы имеем:

```
GC0 @ 4 s 20 MB Eden → SS1 (20 MB)
```

Спустя еще 4 с Эдем вновь заполняется и требует эвакуации (на этот раз в область SS2). Однако в этой упрощенной модели ни один объект, который был перемещен в SS1 во время сборки GC0, не выжил — время жизни объектов составляет всего 200 мс, а прошло уже 4 с, так что все объекты, выделенные до GC0, давно мертвы. Теперь мы имеем:

```
GC1 @ 8.002 s 20 MB Eden → SS2 (20 MB)
```

Это можно сказать и по-другому: после сборки GC1 содержимое SS2 состоит исключительно из объектов, вновь прибывших из Эдема, и ни один объект в SS2 не имеет возраста поколения, большего 1. После еще одной сборки мусора шаблон должен стать очевидным:

```
GC2 @ 12.004 s 20 MB Eden → SS1 (20 MB)
```

Эта идеализированная, простая модель приводит к ситуации, когда никакие объекты не становятся бессрочными, так что это пространство остается пустым на протяжении всей работы. Это, конечно, очень нереалистично.

Вместо этого слабая гипотеза поколений указывает, что времена жизни объектов имеют некоторое распределение, и из-за неопределенности этого распределения некоторые объекты в конечном итоге выживут, чтобы достичь бессрочного состояния.

Давайте рассмотрим очень простой симулятор этого сценария выделения памяти. Он выделяет объекты, большинство из которых очень недолговечны, но некоторые из них имеют значительно более длительный срок жизни. Он также имеет несколько параметров, управляющих распределением: x и y, крайние значения, между которыми располагаются значения размера каждого объекта; скорость распределения (mbPerSec); время жизни короткоживущего объекта (shortLivedMS) и количество потоков, которые приложение должно имитировать (nThreads). Значения по умолчанию показаны в следующем листинге:

```
public class ModelAllocator implements Runnable
{
    private volatile boolean shutdown = false;
    private double chanceOfLongLived = 0.02;
    private int multiplierForLongLived = 20;
    private int x = 1024;
    private int y = 1024;
    private int mbPerSec = 50;
    private int shortLivedMs = 100;
    private int nThreads = 8;
    private Executor exec = Executors.newFixedThreadPool(nThreads);
```

Опустив `main()` и прочий код настройки и установки параметров, мы получим следующий остаток кода `ModelAllocator`:

```
    public void run()
    {
        final int mainSleep = (int)(1000.0 / mbPerSec);

        while (!shutdown)
        {
            for (int i = 0; i < mbPerSec; i++)
            {
                ModelObjectAllocation to =
                    new ModelObjectAllocation(x, y, lifetime());
                exec.execute(to);

                try
                {
                    Thread.sleep(mainSleep);
                }
                catch (InterruptedException ex)
                {
                    shutdown = true;
                }
```

```
            }
        }
    }
    // Простая функция для моделирования слабой гипотезы поколений.
    // Возвращает ожидаемое время жизни объекта - обычно очень короткое,
    // но с небольшой вероятностью объект оказывается "долгоживущим".
    public int lifetime()
    {
        if (Math.random() < chanceOfLongLived)
        {
            return multiplierForLongLived * shortLivedMs;
        }

        return shortLivedMs;
    }
}
```

Основная функция распределителя объединена с простым макетным объектом, используемым для представления выделения памяти, выполняемого приложением:

```
public class ModelObjectAllocation implements Runnable
{
    private final int[][] allocated;
    private final int lifeTime;
    public ModelObjectAllocation(final int x, final int y,
                                 final int liveFor)
    {
        allocated = new int[x][y];
        lifeTime = liveFor;
    }
    @Override
    public void run()
    {
        try
        {
            Thread.sleep(lifeTime);
            System.err.println(System.currentTimeMillis() + ": " +
                               allocated.length);
        }
        catch (InterruptedException ex)
        {
        }
    }
}
```

При просмотре в VisualVM вы увидите простой пилообразный шаблон, который часто наблюдается в поведении памяти приложений Java, эффективно использующих кучу. Такого рода график можно увидеть на рис. 6.10.

Больше и подробнее об инструментах и визуализации эффектов, связанных с памятью, мы поговорим в главе 7, "Вглубь сборки мусора". Заинтересованный читатель может взять описанный в главе симулятор распределения и жизненного цикла и, устанавливая различные значения параметров, посмотреть влияние изменения скорости выделения памяти и процентного количества долгоживущих объектов.

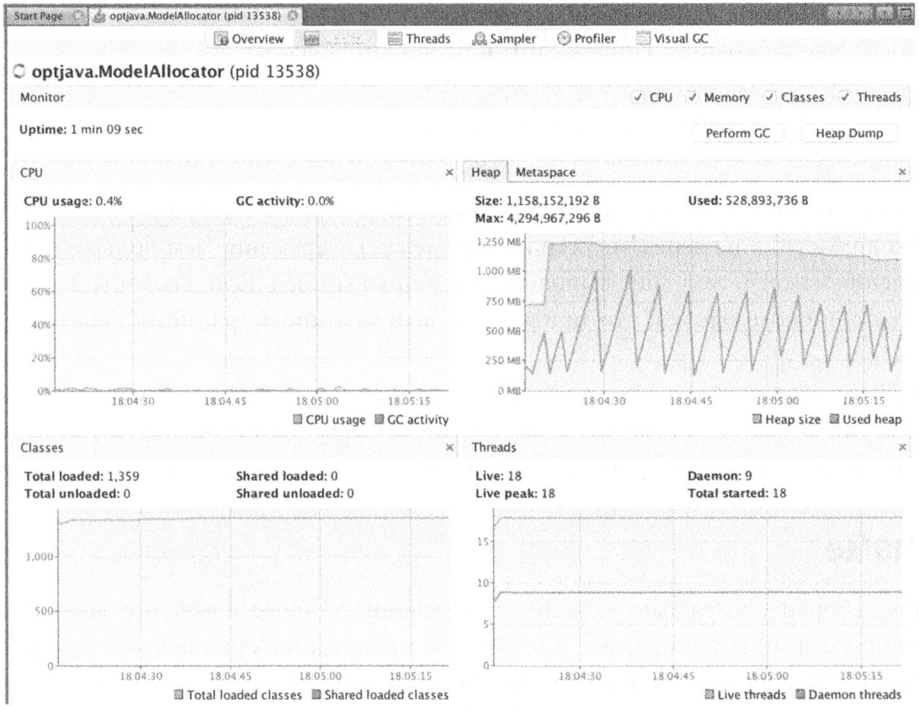

Рис. 6.10. Простой пилообразный шаблон

Чтобы завершить обсуждение выделения памяти, мы хотим обратить внимание на очень распространенный аспект его поведения. В реальном мире скорости выделения памяти могут быть сильно изменчивыми и "взрывными". Рассмотрим следующий сценарий для приложения с устойчивым поведением, как описано выше:

```
  2 s   Установившееся выделение памяти       100 Мбайт/с
  1 s   Пиковое выделение памяти                1 Гбайт/с
100 s   Возврат к установившемуся выделению   100 Мбайт/с
```

Начальное установившееся выполнение выделило 200 Мбайт в Эдеме. При отсутствии долгоживущих объектов вся эта память имеет время жизни службы 100 мс. Затем происходит пиковое выделение памяти. Оно выделяет остальные 200 Мбайт пространства Эдема всего за 0,2 с, из которых 100 Мбайт в возрасте до порогового значения 100 мс. Размер выжившей когорты больше, чем пространство выживших, поэтому у JVM нет иного выбора, кроме как переместить эти объекты непосредственно в бессрочные. Таким образом, мы имеем:

```
GC0 @ 2.2 s 100 MB Eden → Tenured (100 MB)
```

Резкое увеличение скорости выделения памяти привело к 100 Мбайт выживших объектов, хотя следует отметить, что в этой модели все "выжившие" объекты на самом деле недолговечны и очень быстро становятся мертвыми объектами, замусоривающими поколение бессрочных объектов. Они не будут собраны до тех пор, пока не будет выполнена полная сборка мусора.

После еще нескольких сборок картина становится ясной:

```
GC1  @ 2.602 s   200 MB Eden → Tenured (300 MB)
GC2  @ 3.004 s   200 MB Eden → Tenured (500 MB)
GC2  @ 7.006 s    20 MB Eden → SS1 (20 MB) [+ Tenured (500 MB)]
```

Обратите внимание, что, как уже говорилось, сборщик мусора работает по мере необходимости, а не через регулярные промежутки времени. Чем больше скорость выделения памяти, тем чаще выполняются сборки мусора. Если скорость выделения памяти слишком высока, это приводит к вынужденному раннему перемещению объектов.

Данное явление называется *преждевременным продвижением* (premature promotion); это один из самых важных косвенных эффектов сборки мусора и отправная точка для многих настроек, как мы увидим в следующей главе.

Резюме

Тема сборки мусора была активно обсуждаемой сообществом программистов Java с момента создания платформы. В этой главе мы представили ключевые концепции, которые инженеры по производительности должны понимать для эффективной работы с подсистемой сборки мусора JVM. К ним относятся:

- алгоритм маркировки и выметания;
- внутреннее представление времени выполнения объектов в HotSpot;
- слабая гипотеза поколений;
- практические аспекты подсистем памяти HotSpot;
- многопоточные сборщики мусора;
- выделение памяти и роль, которую оно играет.

В следующей главе мы обсудим настройку, мониторинг и анализ сборки мусора. Некоторые из тем, которые мы встретили в этой главе — в особенности выделение памяти и определенные эффекты, такие как преждевременное продвижение, — будут иметь особую важность для предстоящих целей и тем, так что может быть полезным частое обращение к этой главе при изучении последующего материала.

ГЛАВА 7

Вглубь сборки мусора

В предыдущей главе мы представили основы теории сборки мусора Java. Принимая ее за отправную точку, мы будем двигаться вперед, к пониманию современного состояния сборки мусора в Java. Это та область, в которой неизбежны компромиссы, руководящие выбором конкретного сборщика.

Сначала мы рассмотрим другие сборщики мусора, которые предоставляет HotSpot JVM. К ним относятся в основном параллельный сборщик с ультранизкой паузой (CMS) и современные сборщики общего назначения (G1).

Мы также рассмотрим некоторые более редко применяемые сборщики, такие как

- Shenandoah;
- C4;
- Balanced;
- устаревшие сборщики HotSpot.

Не все эти сборщики мусора используются в виртуальной машине HotSpot — мы будем рассматривать сборщики мусора и двух других виртуальных машин: IBM J9 (JVM с ранее закрытым исходным кодом, которую в настоящее время IBM планирует открыть) и Azul Zing (патентованная JVM). Мы уже встречались с этими виртуальными машинами в разделе "Встреча с JVM" главы 2, "Обзор JVM".

Компромиссы и подключаемые сборщики мусора

Один из аспектов платформы Java, который не всегда сразу распознают новички, заключается в том, что, хотя Java и имеет сборщик мусора, спецификации языка и виртуальной машины не говорят о том, как он должен быть реализован. Фактически были реализации Java (например, Lego Mindstorms), которые вообще не реализовывали никакой сборщик мусора![1]

[1] В такой системе очень трудно программировать, поскольку каждый создаваемый объект должен быть повторно использован, а любой объект, выходящий из области видимости, приводит к утечке памяти.

В среде Sun (ныне — Oracle) подсистема сборки мусора рассматривается как подключаемая подсистема. Это означает, что одна и та же Java-программа может выполняться с разными сборщиками мусора без изменения семантики программы, хотя производительность программы может значительно варьироваться в зависимости от того, какой именно сборщик мусора используется.

Основной причиной наличия подключаемых сборщиков мусора является то, что сборка мусора является вычислительной задачей с очень общим характером. В частности, один и тот же алгоритм может подходить не для каждой рабочей нагрузки. В результате алгоритмы сборки мусора представляют собой компромисс между конкурирующими подходами.

Не существует единого алгоритма сборки мусора общего назначения, который можно было бы оптимизировать для всех задач сборки одновременно.

Основные вопросы, которые разработчикам часто необходимо учитывать при выборе сборщика мусора, включают следующие:

- продолжительность паузы;
- пропускная способность (в виде процентного соотношения времени сборки мусора ко времени работы приложения);
- частота пауз (как часто сборщик мусора вынужден останавливать работу приложения);
- эффективность восстановления (как много мусора может быть собрано за один рабочий цикл сборки);
- согласованность пауз (все паузы примерно одной длины?).

Из всего списка непропорционально большое количество внимания часто привлекает время паузы. Хотя этот показатель важен для многих приложений, он не должен рассматриваться изолированно.

Для многих нагрузок время паузы не является важной или полезной характеристикой производительности.

Например, параллельная пакетная обработка или приложение для работы с большими данными, скорее всего, будет гораздо больше озабочено пропускной способностью, чем продолжительностью приостановки. Для многих пакетных заданий время пауз даже в десятки секунд не является слишком актуальным, так что алгоритм сборки мусора, который способствует эффективному использованию процессора и высокой пропускной способности, существенно предпочтительнее алгоритма, который любой ценой обеспечивает непродолжительные паузы.

Инженер по производительности должен также заметить, что существует ряд других недостатков и проблем, которые иногда оказываются важными при выборе сборщика мусора. Однако в случае HotSpot выбор ограничен доступными сборщиками.

В Oracle/OpenJDK, начиная с версии 10, имеются три основных сборщика мусора для общего промышленного использования. Мы уже встречались с параллельными сборщиками, которые легче понимать как с теоретической, так и с алгоритмической точек зрения. В этой главе мы познакомимся с двумя другими сборщиками и рассмотрим, чем они отличаются от Parallel GC.

К концу этой главы, в разделах "Shenandoah" и следующими за ним, мы также встретимся с некоторыми другими доступными сборщиками, но, пожалуйста, обратите внимание, что не все из них рекомендуются к промышленному применению, а некоторые сегодня являются устаревшими. Мы также бегло рассмотрим сборщики мусора, доступные в HotSpot JVM.

Теория параллельных сборщиков мусора

В специализированных системах, таких как системы вывода графики или анимации, часто существует фиксированная частота кадров, которая обеспечивает сборщику мусора возможность регулярного, с фиксированной частотой, выполнения.

Однако сборщики мусора, предназначенные для общего использования, не имеют такого знания о предметной области, которое могло бы улучшить детерминизм их пауз. Что еще хуже, недетерминированное поведение вызывается непосредственно поведением выделения памяти, и многие из систем, используемых Java, демонстрируют высоковариативное поведение.

> *Задержка вычислений является всего лишь незначительным недостатком; основным недостатком в таком случае является непредсказуемость промежутков между сборками мусора.*
>
> *— Эдсгер Дейкстра (Edsger Dijkstra)*

Отправной точкой современной теории сборки мусора является попытка решить указанную Дейкстрой проблему о том, что основным раздражающим фактором при использовании сборки мусора является неопределенный характер пауз приостановки выполнения приложения (как в смысле их продолжительности, так и в смысле частоты).

Один из подходов состоит в том, чтобы использовать параллельный сборщик мусора (или по крайней мере частично, или в основном параллельный), чтобы сократить время паузы, выполняя некоторые необходимые для сбора мусора действия параллельно с потоками приложений. Это неизбежно уменьшает вычислительную

мощность, доступную для реальной работы приложения, а также усложняет код, необходимый для выполнения сборки мусора.

Перед тем как приступить к обсуждению параллельных сборщиков мусора, нужно рассмотреть важную часть терминологии и технологий сборки мусора, поскольку для дальнейшей работы нам совершенно необходимо понимать природу и поведение современных сборщиков мусора.

Точки безопасности JVM

Чтобы выполнить сборку мусора с приостановкой выполнения приложения, как, например, выполняемую параллельными сборщиками HotSpot, все потоки приложения должны быть остановлены. Это кажется почти тавтологией, но до сих пор мы не обсуждали, *как именно* JVM достигает этого.

> *JVM в действительности не является полностью вытесняющей многопоточной средой.*
>
> — *Секрет*

Это не означает, что это чисто кооперативная среда. Операционная система все еще может вытеснять поток (удалять его из ядра) в любое время. Это делается, например, когда поток исчерпал свой временной интервал или перешел в состояние ожидания (`wait()`).

Помимо этой основной функциональности операционной системы, JVM должна выполнять координирующие действия. Для облегчения этой задачи среда выполнения требует, чтобы каждый поток приложения имел специальные точки выполнения, именуемые *точками безопасности* (safepoint), в которых внутренние структуры данных потока находятся в точно определенном состоянии. В это время поток может быть приостановлен для выполнения скоординированных действий.

Мы можем видеть работу точек безопасности в сборщиках мусора с приостановкой приложения (классический пример) и при синхронизации потоков, но есть и другие их применения.

Чтобы понять суть точек, рассмотрим случай сборщика мусора с полной остановкой приложения. Для своей работы он требует стабильного графа объектов. Это означает, что все потоки приложения должны быть приостановлены. Однако нет никакого способа, который сборщик мусора может потребовать от операционной системы выполнить это требование по отношению к потокам приложения, так что потоки приложения (которые выполняются как часть процесса JVM) должны сотрудничать для достижения этой цели. Существует два основных правила, которые регулируют подход JVM к точкам безопасности:

- JVM не может заставить поток перейти в безопасное состояние;
- JVM не может предотвратить выход потока из безопасного состояния.

Это означает, что реализация интерпретатора JVM должна содержать код, способный приостановиться на барьере, если требуется точка безопасности. Для JIT-скомпилированных методов эквивалентные барьеры должны быть внедрены в сгенерированный машинный код. Общий случай достижения точки безопасности выглядит следующим образом.

1. JVM устанавливает глобальный флаг "время для точки безопасности".
2. Отдельные потоки приложения выполняют опрос и видят, что флаг был установлен.
3. Они приостанавливаются и ожидают, когда будут активированы вновь.

Когда этот флаг установлен, все потоки приложений должны остановиться. Потоки, которые останавливаются быстро, должны ожидать медленных (и это время может не полностью учитываться в статистике времени паузы).

Обычные потоки приложений используют механизм опроса. Они всегда будут выполнять проверку между выполнением любых двух байт-кодов в интерпретаторе. В скомпилированном коде наиболее распространены случаи, когда JIT-компилятор вставляет код опроса для точек безопасности при выходе из скомпилированного метода и при обратном движении в цикле (например, при переходе выполнения к вершине цикла).

Возможно, что потоку потребуется много времени для попадания в точку безопасности, а теоретически он может просто никогда не остановиться (но это редкий патологический случай, который должен быть преднамеренно спровоцирован).

Идея, что все потоки должны полностью остановиться до начала сборки мусора, похожа на использование защелки, например, реализованной в `CountDownLatch` библиотеки `java.util.concurrent`.

Некоторые частные случаи точек безопасности заслуживают особого упоминания. Поток выполнения автоматически находится в точке безопасности, если:

- он заблокирован монитором;
- он выполняет код JNI.

Поток *не* обязательно находится в точке безопасности, если:

- он находится в процессе выполнения байт-кода (режим интерпретации);
- он был прерван операционной системой.

Позже мы снова встретимся с механизмом точек безопасности, так как он представляет собой важную часть внутренней работы JVM.

Трехцветная маркировка

Статья Дейкстры и Лампорта (Lamport) 1978 года[2], описывающая разработанный ими алгоритм трехцветной маркировки, была ориентиром как для доказательств корректности параллельных алгоритмов, так и для сборки мусора, а основной описанный алгоритм и ныне остается важной частью теории сборки мусора.

Алгоритм работает следующим образом.

- Корни сборки окрашиваются в серый цвет.
- Все прочие объекты окрашиваются в белый цвет.
- Маркирующий поток переходит к случайному серому узлу.
- Если узел имеет белые дочерние узлы, маркирующий поток сначала окрашивает их в серый цвет, а затем окрашивает сам узел в черный цвет.
- Этот процесс повторяется, пока не останется ни одного серого узла.
- Все черные объекты являются доказанно достижимыми и должны оставаться живыми.
- Белые узлы пригодны для сборки и соответствуют объектам, которые больше недоступны.

Описанные действия являются базовой формой алгоритма; у реального алгоритма имеются некоторые усложнения. Пример работы алгоритма показан на рис. 7.1.

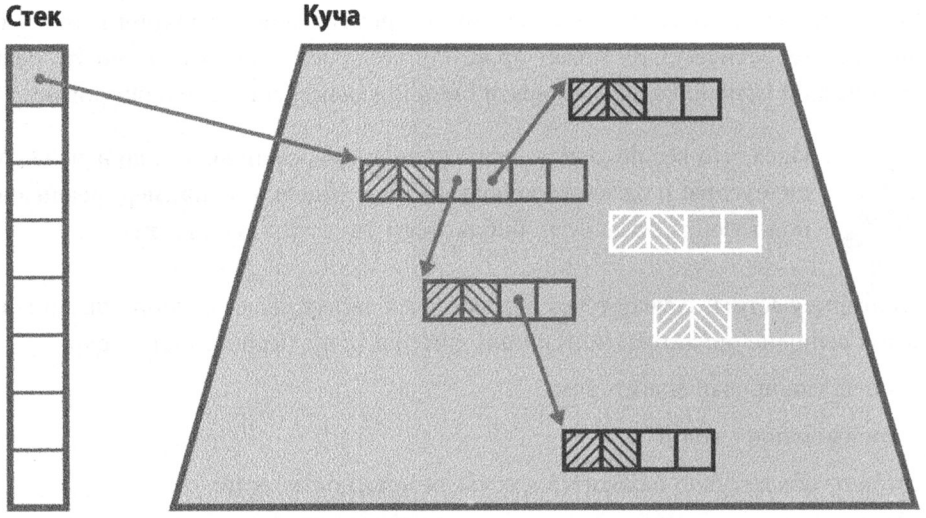

Рис. 7.1. Трехцветная маркировка

[2] Edsger Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM* 21 (1978): 966–975.

Параллельная сборка также часто использует методику, именуемую *снимком в начале* (snapshot at the beginning — SATB). Это означает, что сборщик рассматривает объекты как живые, если они достижимы в начале цикла сбора или были выделены с того момента. Это добавляет некоторые незначительные сложности в работу алгоритма, такие как потоки модификаторов, которые должны создавать новые объекты в черном состоянии, если сборка запущена, и в белом состоянии, если сборка не выполняется.

Алгоритму трехцветной маркировки должно сопутствовать небольшое количество дополнительной работы, гарантирующее, что изменения, вносимые выполняемыми потоками приложений, не приведут к сборке живых объектов. Дело в том, что в параллельном сборщике в то время, когда маркирующие потоки выполняют трехцветный алгоритм, потоки приложения (модификаторы) могут менять граф объектов.

Рассмотрим ситуацию, когда объект уже окрашен в черный цвет маркирующим потоком, а затем обновляется модифицирующим потоком таким образом, что начинает указывать на белый объект. Эта ситуация показана на рис. 7.2.

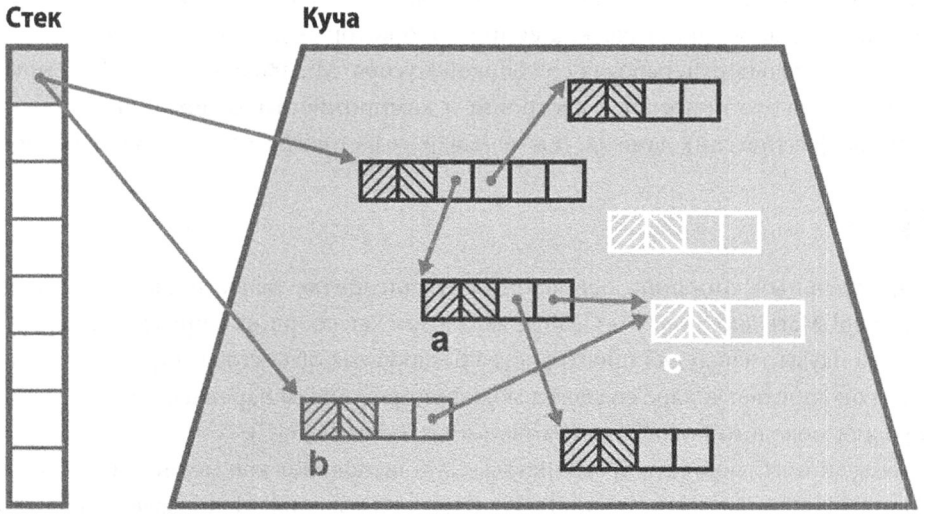

Рис. 7.2. Поток модификатора может помешать трехцветной маркировке

Если теперь удалить все ссылки от серых объектов на новый белый объект, мы получим ситуацию, когда белый объект должен быть доступен, но будет удален, поскольку, в соответствии с правилами алгоритма, не будет найден.

Эту проблему можно решить несколькими способами. Например, когда поток-модификатор выполняет обновление, можно изменять цвет черного объекта обратно на серый, добавляя его обратно во множество узлов, которые нуждаются в обработке.

Этот подход, используя "барьер записи" для обновления, имел бы то хорошее алгоритмическое свойство, что он поддерживал бы *трехцветный инвариант* на протяжении всего цикла маркировки.

Ни один узел черного объекта не может хранить ссылку на узел белого объекта в процессе параллельной маркировки.

— Трехцветный инвариант

Альтернативный подход состоит в том, чтобы сохранить очередь всех изменений, которые могли бы потенциально нарушить инвариант, а затем использовать вторичный этап исправления, который начинается после завершения основного этапа. Различные сборщики могут разрешать описанную проблему с трехцветной маркировкой по-разному, исходя из таких критериев, как производительность или требуемая блокировка.

В следующем разделе мы встретимся со сборкой с малой паузой, CMS. Мы рассматриваем этот сборщик до других несмотря на то, что он является сборщиком с ограниченным диапазоном применимости. Дело в том, что разработчики часто не знают о том, в какой степени и каких компромиссов требует настройка сборки мусора.

Рассматривая CMS в числе первых сборщиков мусора, мы можем продемонстрировать некоторые из практических вопросов, о которых должны знать специалисты по производительности, работая со сборкой мусора. Мы надеемся на то, что при выборе сборщика это приведет к настройке и компромиссам, основанным в большей степени на фактических данных, и в меньшей — на программистском фольклоре.

CMS

Параллельный сборщик, основанный на алгоритме маркировки и выметания (Concurrent Mark and Sweep — CMS), разработан как сборщик с чрезвычайно низким временем паузы только для пространства бессрочных объектов (старого поколения). Обычно он работает в паре со слегка модифицированным параллельным сборщиком мусора для молодого поколения (ParNew вместо Parallel GC).

Чтобы минимизировать время паузы, CMS выполняет как можно большее количество работы во время активности потоков приложения. Используемый алгоритм маркировки и выметания представляет собой разновидность алгоритма трехцветной маркировки, а это, конечно, означает, что граф объектов может изменяться в процессе сканирования кучи сборщиком. В результате CMS должен исправлять свои записи, чтобы избежать нарушения второго правила сборщиков мусора — сборки в качестве мусора объекта, который все еще жив.

Это приводит к более сложному множеству этапов в CMS, чем в других параллельных сборщиках. Обычно это следующие этапы.

1. Первоначальная маркировка (пауза)
2. Параллельная маркировка

3. Параллельная предварительная очистка
4. Повторная маркировка (пауза)
5. Параллельное выметание
6. Параллельный сброс

На большинстве этапов сборка мусора работает одновременно с потоками приложений. Однако для двух этапов (начальная и повторная маркировки) все потоки приложений должны быть остановлены. Общий результат состоит в том, чтобы заменить одну длинную паузу с приостановкой всех потоков приложения двумя, обычно очень короткими паузами.

Цель этапа первоначальной маркировки — обеспечение стабильного множества начальных точек для сборки мусора, которые находятся в пределах региона; они известны как *внутренние указатели* (internal pointers) и обеспечивают эквивалентное множество для корней GC для целей цикла сборки мусора. Преимущество такого подхода заключается в том, что он позволяет этапу маркировки сфокусироваться на одном пуле сборки, без необходимости рассматривать другие области памяти.

После того как первоначальная маркировка завершена, начинается фаза параллельной маркировки. По сути, здесь запускается алгоритм трехцветной маркировки в куче, отслеживая любые изменения, которые впоследствии могут потребовать исправлений.

Этап параллельной предварительной очистки пытается как можно больше сократить продолжительность паузы этапа повторной маркировки. Этап повторной маркировки использует таблицы карт для того, чтобы исправить маркировку, на которую могли повлиять модифицирующие потоки во время этапа параллельной маркировки.

Наблюдаемыми результатами использования CMS для большинства рабочих нагрузок являются следующие.

- Потоки приложения приостанавливаются на меньшее время.
- Единый цикл полной сборки мусора требует больше времени.
- Снижается пропускная способность приложения во время цикла сборки мусора CMS.
- Сборка мусора использует большее количество памяти для отслеживания объектов.
- Для выполнения сборки мусора требуется значительно большее время процессора.
- CMS не выполняет уплотнения кучи, так что пространство бессрочных объектов может стать фрагментированным.

Внимательный читатель заметит, что не все указанные характеристики положительны. Вспомните, что в сборке мусора нет панацеи, а есть просто набор вариантов, которые могут быть подходящими (или приемлемыми) для конкретной рабочей нагрузки, которую настраивает инженер.

Как работает CMS

Одним из наиболее часто упускаемых аспектов CMS, как ни странно, является его большая сила. CMS в основном работает одновременно с потоками приложений. По умолчанию CMS будет использовать половину доступных потоков для выполнения параллельных этапов сборки мусора, оставляя вторую половину для выполнения Java-кода потоками приложения, *неизбежно связанными с выделением памяти для новых объектов*. Это звучит очевидной банальностью, но имеет непосредственное следствие. Что произойдет, если Эдем заполнится во время работы CMS?

Ответ, и это неудивительно, состоит в том, что, поскольку потоки приложений не смогут продолжаться, они приостановятся, и параллельно с работой CMS будет выполнена сборка мусора молодого поколения (с остановкой потоков приложения). Это выполнение сборки молодого поколения, как правило, отнимает больше времени, чем в случае параллельных сборщиков, потому что в ее распоряжении имеется только половина ядер, доступных для сборки мусора (на другой половине ядер работает CMS).

В конце этой сборки молодого поколения некоторые объекты, как правило, будут иметь право на перемещение в бессрочные. Эти объекты должны быть перенесены в область бессрочных во время работы CMS, что требует некоторой координации между двумя сборщиками. Вот почему CMS требует немного другой сборщик мусора для молодого поколения.

В обычных условиях при сборке молодого поколения только небольшое количество объектов становятся бессрочными, и работа CMS над старым поколением завершается нормально, освобождая место в области бессрочных объектов. Затем приложение возвращается к нормальной работе, причем все ядра освобождаются для использования потоками приложений.

Однако рассмотрим случай очень активного выделения памяти, возможно, вызывающего преждевременное перемещение (о чем говорилось в конце раздела "Роль выделения памяти" главы 6, "Сборка мусора") в молодой сборке. Это может привести к ситуации, когда сборка молодого поколения имеет слишком много объектов для перемещения в доступное пространство бессрочных объектов (рис. 7.3).

Это явление известно как *сбой параллельного режима* (concurrent mode failure — CMF), и у JVM в данный момент нет иного выбора, кроме как вернуться к сборке с использованием ParallelOld, сборщика мусора, который является сборщиком с полной приостановкой приложения. По существу, нагрузка выделения памяти была настолько велика, что у CMS не было времени, чтобы закончить обработку старого

поколения до того, как будет заполнено пространство для размещения вновь перемещаемых объектов.

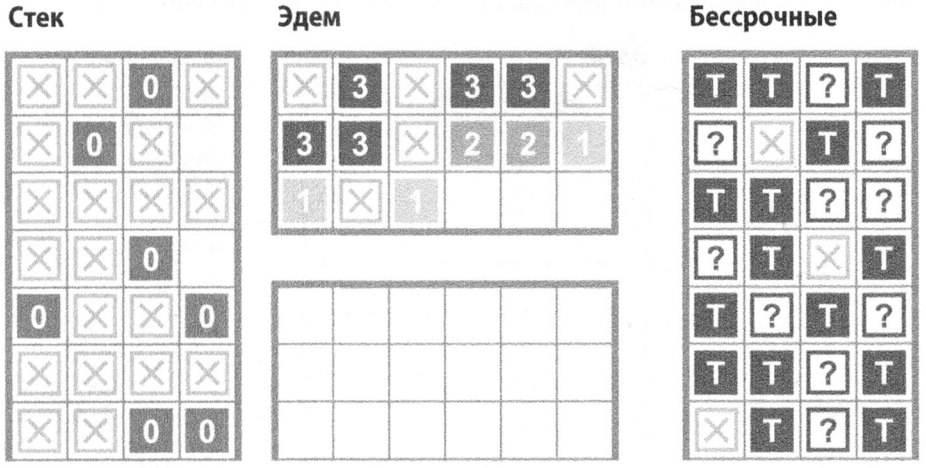

Рис. 7.3. Сбой параллельного режима из-за нагрузки выделения памяти

Чтобы избежать частых сбоев параллельного режима, CMS необходимо запускать цикл сбора до того, как будет полностью заполнено пространство бессрочных объектов. Уровень занятости кучи этого пространства, при котором CMS начнет сборку, контролируется наблюдаемым поведением кучи. На него можно влиять с помощью переключателей (флагов), и по умолчанию CMS начинает работу при 75%-ной заполненности области бессрочных объектов.

Есть еще одна ситуация, которая может привести к сбою параллельного режима, — это фрагментация кучи. В отличие от ParallelOld, CMS не упаковывает пространство бессрочных объектов при запуске. Это означает, что после завершения работы CMS свободное пространство в пространстве бессрочных объектов не является одним непрерывным блоком, а объекты, которые перемещаются, должны заполнять промежутки между существующими объектами.

В определенный момент сборка молодого поколения может столкнуться с ситуацией, когда объект не может быть перемещен в пространство бессрочных объектов из-за отсутствия достаточного количества пространства, чтобы скопировать объект (рис. 7.4).

Это сбой параллельного режима, вызванный фрагментацией кучи; и, как и прежде, единственным решением является возврат к полной сборке с использованием ParallelOld (сборщика мусора, который *является* уплотняющим), поскольку это должно освобождать достаточно непрерывное пространство, чтобы позволить переместить объект.

Как в случае фрагментации кучи, так и в случае, когда сборка молодого поколения опережает CMS, необходимость возврата к полностью останавливающему

приложение сборщику ParallelOld может стать важным событием для приложения. Фактически настройка приложений с малой задержкой, использующих CMS для избегания сбоев параллельного режима, сама по себе является важной темой.

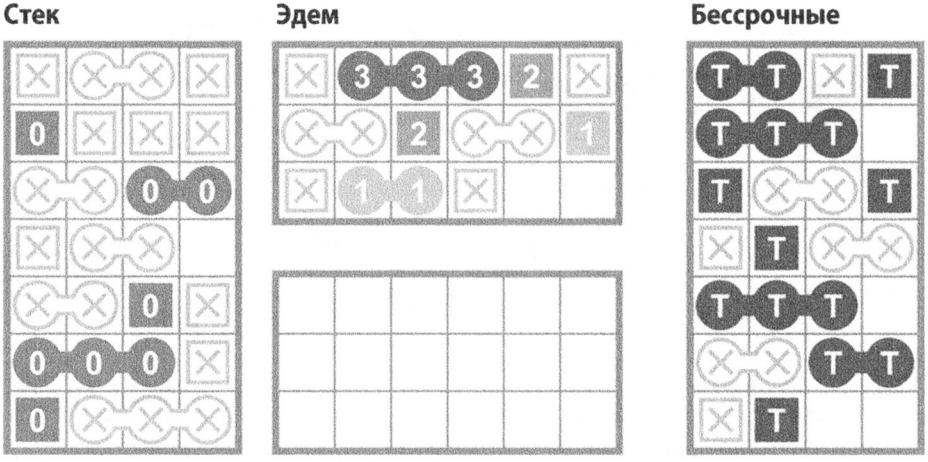

Рис. 7.4. Сбой параллельного режима из-за фрагментации

Внутренне CMS использует свободный список *блоков* (chunks) памяти для управления доступным свободным пространством. На заключительном этапе (параллельного выметания) смежные свободные блоки будут объединены потоками выметания. Это делается для того, чтобы обеспечить большие блоки свободного пространства и попытаться избежать сбоев параллельного режима, вызванных фрагментацией.

Однако выметание работает параллельно с модифицирующими потоками. Таким образом, если только поток выметания и потоки выделения не будут должным образом синхронизированы, свежевыделенный блок может быть неверно выметен. Чтобы предотвратить это, поток выметания блокирует свободные списки на время выполнения выметания.

Основные флаги JVM для настройки CMS

Сборщик мусора CMS активируется с помощью флага

```
-XX:+UseConcMarkSweepGC
```

В современных версиях HotSpot этот флаг также активирует `ParNewGC` (небольшую вариацию параллельного сборщика молодого поколения).

В общем случае у CMS имеется очень большое количество флагов (более 60), которые можно настроить. Иногда может возникнуть соблазн поучаствовать в попытках оптимизировать производительность, тщательно регулируя различные настройки, предоставляемые CMS. Этому соблазну нужно противиться, так как в большинстве случаев это соответствует антипаттернам "Отсутствие общей картины"

и "Настройка по совету" (см. раздел "Каталог антипаттернов производительности" главы 4, "Паттерны и антипаттерны тестирования производительности").

Более подробно настройку CMS мы рассмотрим в разделе "Настройка Parallel GC" главы 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора".

G1

Сборщик мусора G1 (Garbage First) представляет собой совершенно иную разновидность сборщика по сравнению с параллельными сборщиками или CMS. Он был впервые представлен в экспериментальной и нестабильной форме в Java 6, но затем основательно переписан за время жизни Java 7 и стал действительно стабильным и готовым к промышленной работе с выпуском Java 8u40.

Не рекомендуется использовать G1 с любой версией Java до 8u40, независимо от типа рассматриваемой нагрузки.

G1 изначально предназначался как замена сборщика мусора с низкими задержками, который бы

- был гораздо проще, чем CMS;
- был менее чувствителен к преждевременному переносу;
- имел лучшее поведение при масштабировании (в особенности время пауз) для больших куч;
- мог бы устранить сборку с полной приостановкой приложения или существенно снизить необходимость в ней.

Однако со временем G1 превратился в то, что можно рассматривать, в первую очередь, как универсальный сборщик с лучшими временами пауз на больших кучах (которые все чаще рассматриваются как "новая норма").

Oracle утверждает, что G1 станет сборщиком по умолчанию в Java 9, обеспечив, таким образом, переход от параллельных сборщиков, независимо от влияния на конечных пользователей. Поэтому очень важно, чтобы аналитики в области производительности хорошо понимали G1 и чтобы любые приложения, переносимые с Java 8 на Java 9, были должным образом протестированы (как часть процедуры перехода на новую версию).

Сборщик мусора G1 имеет дизайн, который пересматривает до сих пор использовавшееся понятие поколений. В отличие от параллельных сборщиков и CMS, G1

не имеет выделенных непрерывных областей памяти для поколений. Кроме того, как мы увидим, он не следует схеме полусферической кучи.

Схема кучи G1 и регионы

Куча G1 основана на концепции регионов (region). Это области, которые по умолчанию имеют размер 1 Мбайт (но могут быть и больше в больших кучах). Использование регионов допускает несмежные поколения и позволяет иметь сборщик мусора, который не должен собирать весь мусор при каждом запуске.

Вся куча G1 все еще непрерывна в памяти — это просто память, внутри которой умещается каждое поколение.

Схему кучи G1 на основе регионов можно рассмотреть на рис. 7.5.

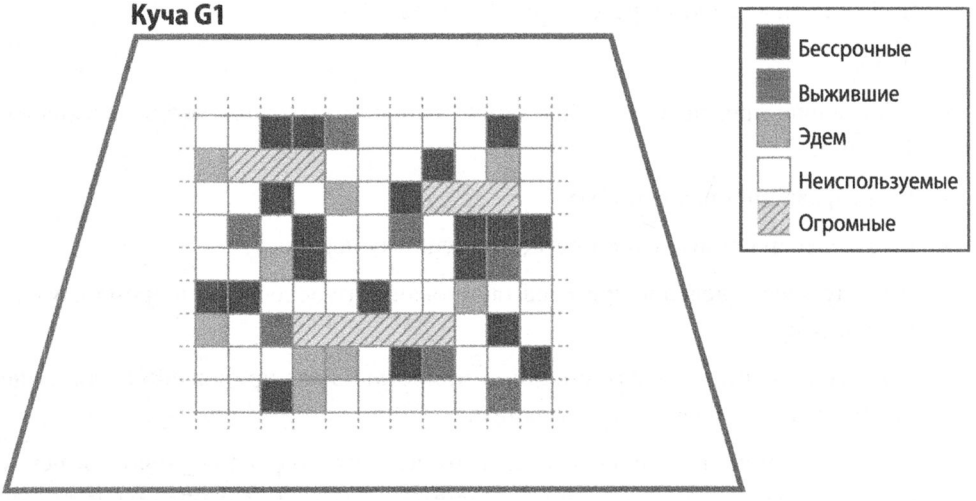

Рис. 7.5. Регионы G1

Алгоритм G1 допускает регионы размером 1, 2, 4, 8, 16, 32 или 64 Мбайт. По умолчанию ожидается от 2048 до 4095 регионов в куче, и размер региона будет корректироваться следующим образом для достижения этого значения.

Чтобы вычислить размер региона, вычислим

```
<Размер кучи> / 2048
```

А затем округлим его вниз до ближайшего разрешенного значения размера региона. Затем можно вычислить количество регионов:

```
Количество регионов = <Размер кучи> / <Размер региона>
```

Как обычно, мы можем изменить это значение, применив соответствующий переключатель времени выполнения.

Алгоритм G1

Высокоуровневая картина сборки мусора заключается в том, что G1:

- использует параллельный этап маркировки;
- представляет собой эвакуирующий сборщик мусора;
- обеспечивает "статистическое уплотнение".

Во время разогрева сборщик мусора отслеживает статистику того, как много "типичных" регионов может быть собрано в течение рабочего цикла сборки мусора. Если можно собрать количество памяти, достаточное, чтобы сбалансировать новые объекты, которые были выделены с момента последней сборки мусора, то G1 не уступает выделению памяти (сборщик мусора освобождает достаточно памяти для размещения новых объектов).

Концепции выделения TLAB, эвакуации в пространство выживших и перемещение в пространство бессрочных объектов в целом аналогичны другим системам сборки мусора HotSpot, которые мы уже встречали.

Объекты, которые занимают больше половины размера региона, считаются огромными (humongous) объектами, которые выделяются непосредственно в особых *огромных регионах* (humongous regions), которые представляют собой свободные смежные регионы, немедленно становящиеся частью поколения бессрочных объектов (а не Эдема).

G1 все еще имеет концепцию молодого поколения, состоящего из регионов Эдема и выживших, но, конечно, регионы, которые составляют поколение, не являются смежными в G1. Размер молодого поколения адаптивен и основан на целевом времени паузы.

Напомним, что когда мы встречались со сборщиком ParallelOld, мы обсуждали эвристику "несколько ссылок от старых к молодым объектам" в разделе "Слабая гипотеза поколений" главы 6, "Сборка мусора". HotSpot использует механизм, именуемый таблицами карт, чтобы помочь использовать это явление в параллельном и CMS сборщиках.

У сборщика G1 есть соответствующая функциональная возможность, которая помогает отслеживать регионы. *Запоминаемые множества* (Remembered Sets, обычно используется сокращенное название — *RSets*) представляют собой записи в пределах региона, которые отслеживают внешние ссылки, указывающие на регион кучи. Это означает, что вместо отслеживания всей кучи в поисках ссылок, указывающих

в регион, G1 просто нужно изучить RSets, а затем сканировать эти регионы в поисках ссылок.

На рис. 7.6 показано, как RSets используются для реализации подхода G1 к разделению работы по сборке мусора между аллокатором и сборщиком.

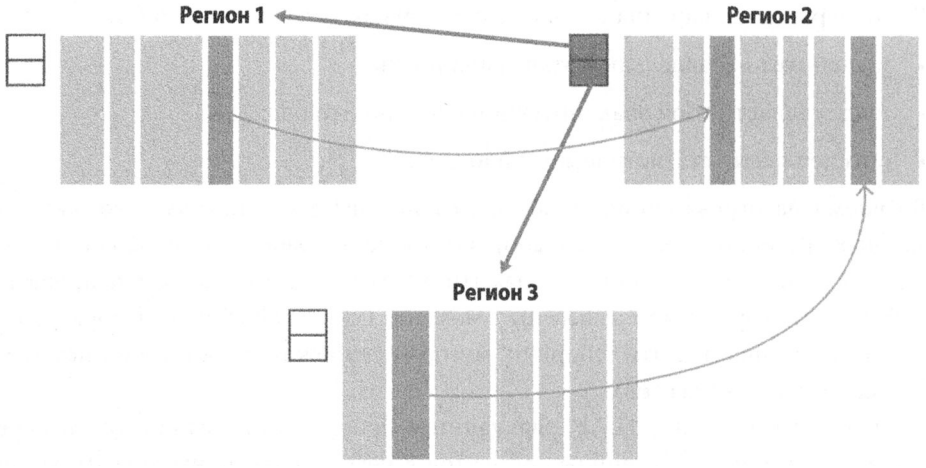

Рис. 7.6. Запоминаемые множества

И RSets, и таблицы карт представляют собой подходы, которые могут помочь с решением проблемы сборки мусора, называемой *плавающим мусором* (floating garbage). Эта проблема возникает, когда объекты, которые в противном случае были бы мертвы, поддерживаются в живом состоянии ссылками от мертвых объектов извне текущего множества сборки. То есть глобальная маркировка может увидеть, что они мертвы, но более ограниченная локальная маркировка может неправильно трактовать их как живые, в зависимости от используемого набора корней.

Этапы сборки мусора G1

G1 содержит набор этапов, подобных тем, с которыми мы уже встречались, в частности, при рассмотрении CMS.

1. Первоначальная маркировка (пауза)
2. Параллельное сканирование корней
3. Параллельная маркировка
4. Повторная маркировка (пауза)
5. Очистка (пауза)

Параллельное сканирование корней является параллельным этапом, на котором выполняется сканирование выживших регионов при первоначальной маркировке ссылок на старые поколения. Этот этап должен завершиться до того, как начнется

следующая сборка молодого поколения. На этапе повторной маркировки цикл маркировки завершается. На этом этапе выполняется также обработка ссылок (включая слабые и мягкие ссылки) и очистки, связанной с реализацией подхода SATB.

Очистка в основном выполняется с приостановкой потоков приложения и включает учет и "промывание" RSet. Задачи учета определяют регионы, которые в настоящее время полностью свободны и готовы к повторному использованию (например, как регионы Эдема).

Основные флаги JVM для G1

Чтобы включить G1 (в Java 8 и более ранних версиях), используется параметр командной строки:

```
+XX:UseG1GC
```

Напомним, что целью G1 является уменьшение времени паузы. Разработчик может указать желаемое максимальное количество времени, на которое приложение может приостанавливаться на каждом цикле сборки мусора. Это указание является целью, так что нет никакой гарантии, что приложение действительно его выполнит. Если заданное значение окажется слишком низким, подсистема сборки мусора будет не в состоянии достичь поставленной цели.

Сборка мусора руководствуется выделением памяти, которое может во многих приложениях Java оказаться весьма непредсказуемым. Это может ограничить возможности G1 в достижении указанных целей или сделать такое достижение полностью невозможным.

Вот переключатель, который управляет базовым поведением сборщика мусора:

```
-XX:MaxGCPauseMillis=200
```

Это означает, что целевое время паузы по умолчанию равно 200 мс. На практике очень трудно надежно достичь времени паузы, равного менее 100 мс, и такие цели могут не быть удовлетворены сборщиком. Еще одна могущая быть полезной настройка — возможность изменения размера региона, перекрывающая значение по умолчанию:

```
-XX:G1HeapRegionSize=<n>
```

Обратите внимание, что <n> должно быть степенью 2, от 1 до 64, и указывать значение в мегабайтах. О других флагах G1 мы поговорим в главе 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора", когда речь будет идти о настройке G1.

В целом сборщик G1 стабилен с точки зрения алгоритма и полностью поддерживается корпорацией Oracle (и рекомендован начиная с версии Java 8u40). Что касается низких значений задержек, то он все еще не обладает такими низкими задержками,

как CMS для большинства рабочих нагрузок, и непонятно, сможет ли он когда-либо обойти в этом отношении такой сборщик, как CMS. Однако данный сборщик мусора постоянно улучшается и находится в центре внимания усилий инженеров Oracle в рамках команды, работающей над JVM.

Shenandoah

Так же, как Oracle работает над созданием сборщика мусора общего назначения следующего поколения, так и Red Hat работает над своим собственным сборщиком, который называется *Shenandoah*, в рамках проекта OpenJDK. Пока что это все еще экспериментальный сборщик мусора, не готовый к реальному использованию на момент написания книги. Однако он демонстрирует некоторые многообещающие характеристики и заслуживает хотя бы беглого знакомства.

Цель Shenandoah, как и G1, заключается в сокращении времени паузы (в особенности на больших кучах). Подход Shenandoah состоит в выполнении параллельного уплотнения. Этапы сборки мусора в Shenandoah таковы.

1. Первоначальная маркировка (пауза)
2. Параллельная маркировка
3. Окончательная маркировка (пауза)
4. Параллельное уплотнение

Эти фазы Shenandoah могут, на первый взгляд, показаться похожими на те, которые имеются в CMS и G1 и в некоторых аналогичных подходах (например, SATB). Однако здесь есть некоторые фундаментальные отличия.

Одним из самых ярких и важных аспектов Shenandoah является использование *указателя Брукса* (Brooks pointer)[3]. Этот метод использует дополнительное слово памяти для каждого объекта, чтобы указать, был ли объект перемещен на предыдущем этапе сборки мусора, и указать местоположение новой версии содержимого объекта.

Полученная схема кучи, используемая Shenandoah для своих обычных указателей на объекты, показана на рис. 7.7. Этот механизм иногда называют подходом с "указателем пересылки" (forwarding pointer). Если объект не был перемещен, указатель Брукса просто указывает на следующее слово памяти.

Механизм указателя Брукса основан на доступности аппаратной команды сравнения и замены (compare-and-swap — CAS) для атомарных обновлений адресов пересылки.

[3] Rodney Brooks, "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," in LFP'84, *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (New York: ACM, 1984), 256–262.

Рис. 7.7. Указатель Брукса

Этап параллельной маркировки проходит по куче и помечает все живые объекты. Если ссылка на объект указывает на обычный указатель на объект, который содержит указатель пересылки, то такая ссылка обновляется и указывает непосредственно на новое местоположение обычного указателя на объект (рис. 7.8).

Рис. 7.8. Обновление указателя пересылки

На этапе окончательной маркировки сборщик Shenandoah приостанавливает приложение для повторного сканирования множества корней, затем копирует и обновляет корни таким образом, чтобы они указывали на эвакуированные объекты.

Параллельное уплотнение

Потоки сборщика мусора (работающие параллельно с потоками приложения) выполняют эвакуацию следующим образом.

1. Копируют объект в TLAB (спекулятивно, т.е. с возможностью ошибки).
2. Для обновления указателя Брука таким образом, чтобы он указывал на эту спекулятивную копию, используется атомарная команда сравнения и замены.

3. В случае успеха поток уплотнения выигрывает гонку, и все будущие обращения к этой версии объекта будут выполняться через указатель Брукса.
4. В случае неудачи поток уплотнения проигрывает, отменяет спекулятивное копирование и следует указателю Брукса, оставленному победившим потоком.

Поскольку Shenandoah является параллельным сборщиком, во время работы цикла сборки потоками приложения создается новый мусор. Таким образом, при работающем приложении сборка не должна отставать от распределения памяти.

Получение Shenandoah

Сборщик мусора Shenandoah в настоящее время недоступен ни как часть поставки Oracle Java, ни в большинстве дистрибутивов OpenJDK. Он поставляется как часть бинарных файлов IcedTea в некоторых дистрибутивах Linux, включая Red Hat Fedora.

Для прочих пользователей требуется выполнение компиляции из исходных текстов (на момент написания книги). Это достаточно просто в Linux, но может быть не такой простой задачей в других операционных системах из-за различий компиляторов (например, macOS использует `clang`, а не `gcc`) и других аспектов операционной среды.

Как только рабочая сборка получена, Shenandoah может быть активирован следующим переключателем:

```
-XX:+UseShenandoahGC
```

Сравнение времен задержки Shenandoah по сравнению с другими сборщиками показано на рис. 7.9.

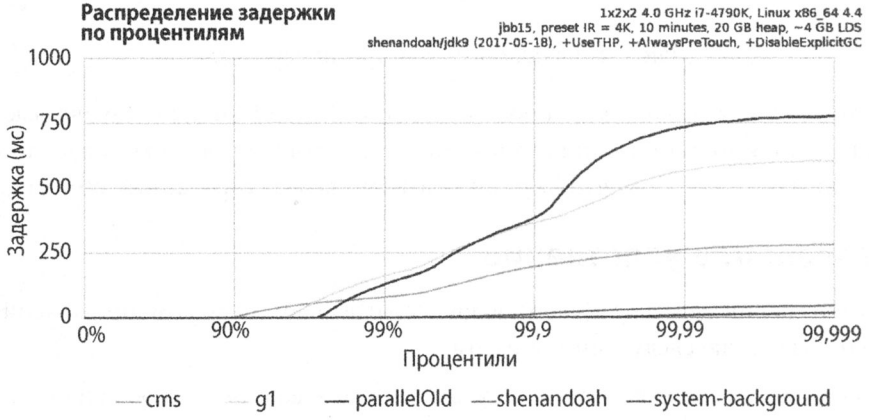

Рис. 7.9. Сравнение Shenandoah с другими сборщиками мусора

Один из недооцененных аспектов Shenandoah заключается в том, что он не является сборщиком поколений. По мере приближения к готовой к промышленному

использованию версии последствия этого проектного решения станут более ясными, но пока оно является потенциальным источником беспокойства для приложений, чувствительных к производительности.

C4 (Azul Zing)

Azul Systems выпускает два различных предложения для платформы Java. Одно из них, Zulu, представляет собой решение FOSS на базе OpenJDK, доступное для нескольких платформ. Другое — Zing, коммерческая патентованная платформа, доступная только для Linux. Она использует версию библиотек классов Java, производную от OpenJDK (хотя и по закрытой лицензии Oracle), но совершенно иную виртуальную машину.

Одним из важных аспектов Zing является то, что это решение было изначально разработано для 64-разрадных машин, и поддержка 32-разрядных архитектур никогда даже не планировалась.

Zing VM содержит несколько новых программных технологий, включая сборщик мусора C4 (Continuously Concurrent Compacting Collector — непрерывный параллельный уплотняющий сборщик) и некоторые новые технологии JIT, включая ReadyNow и компилятор Falcon.

Подобно Shenandoah, Zing использует параллельный алгоритм уплотнения, но не использует указатели Брукса. Вместо этого заголовок объекта Zing представляет собой одно 64-битовое слово (а не заголовок из двух слов, который использует HotSpot). Единственное слово заголовка содержит идентификатор класса (kid — klass ID), который является не указателем на класс, а числовым идентификатором класса (длиной около 25 бит).

На рис. 7.10 показан заголовок объекта Zing, в том числе использование некоторых из битов обычных указателей на объекты для барьера загруженных значений (loaded value barrier — LVB) вместо битов адреса.

Нижние 32 бита слова заголовка используются для информации о блокировке. Она включает состояние блокировки, а также дополнительную информацию, зависящую от состояния блокировки. Например, в случае *тонкой блокировки* (thin lock) это будет идентификатор потока — владельца тонких блокировок. Более подробную информацию о тонких блокировках вы найдете в разделе "Построение параллельных библиотек" главы 12, "Методы повышения производительности параллельной работы".

Zing не поддерживает сжатые обыкновенные указатели на объекты или иную эквивалентную технологию, поэтому для куч размером меньше порядка 30 Гбайт заголовки объектов оказываются большими и занимают больше места в куче, чем в эквивалентной куче HotSpot.

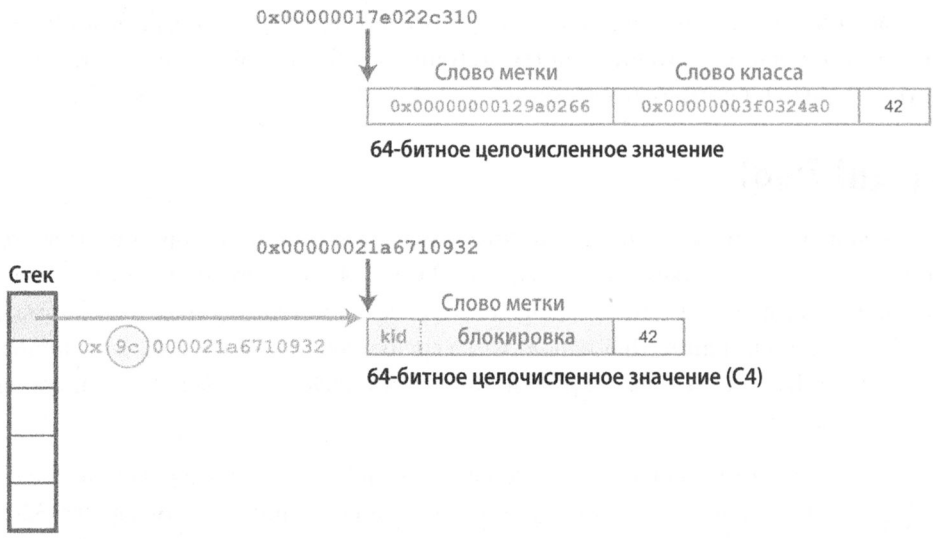

Рис. 7.10. Схема заголовка объекта в Zing

Выбор реализации только для 64-разрядных архитектур означал, что метаданные Zing никогда не планировалось располагать в 32 битах (или выделять для них расширенные структуры, адресуемые косвенно с помощью указателя), тем самым избегая ряда мучений с указателями, присутствующими в 32-разрядном HotSpot.

Барьер загруженных значений

В сборщике Shenandoah потоки приложений могут загружать ссылки на объекты, которые могли быть перемещены, а для отслеживания их нового местоположения используется указатель Брукса. Основная идея барьера загруженных значений (Loaded Value Barrier — LVB) заключается в том, чтобы избежать этой схемы, а вместо нее предоставить решение, в котором каждая загруженная ссылка, как только загрузка будет завершена, указывает непосредственно на текущее местоположение объекта. Azul называет эту методику *самоисцеляющимся барьером* (self-healing barrier).

Если Zing следует по ссылке на объект, который был перемещен сборщиком, то перед тем, как делать что-либо еще, поток приложения обновляет ссылку на новое местоположение объекта, тем самым "исцеляя" причину проблемы перемещения. Это означает, что каждая ссылка обновляется не более одного раза, и если ссылка никогда не используется снова, то не выполняется никакая работа по поддержанию ее в актуальном состоянии.

Как и заголовочное слово, ссылки на объекты Zing (например, из локальной переменной, расположенной в стеке, на объект, хранящийся в куче) используют некоторые из битов ссылки для указания метаданных о состоянии сборки объекта. Это

экономит некоторое количество памяти, позволяя использовать биты самой ссылки, а не биты единственного слова заголовка.

Zing определяет структуру `Reference`, которая имеет примерно следующий вид:

```
struct Reference
{
    unsigned inPageVA : 21;  // биты 0-20
    unsigned PageNumber: 21; // биты 21-41
    unsigned NMT : 1;        // бит 42
    unsigned SpaceID : 2;    // биты 43-44
    unsigned unused : 19;    // биты 45-63
};
int Expected_NMT_Value[4] = {0, 0, 0, 0};
// Значения идентификатора памяти:
// 00  NULL и  указатели не в кучу
// 01  Ссылки на старое поколение
// 10  Ссылки на новое поколение
// 11  Не используется
```

Бит метаданных `NMT` (Not Marked Through — не промаркированный) используется для указания, был ли данный объект отмечен в текущем цикле сборки. C4 поддерживает целевое состояние, которым должны быть помечены живые объекты, и когда объект обнаруживается в процессе маркировки, то бит `NMT` устанавливается таким образом, чтобы он был равен целевому состоянию. В конце цикла сборки C4 меняет значение бита целевого состояния, так что теперь все выжившие объекты готовы к следующему циклу сборки.

Этапы цикла сборки мусора в C4 таковы.

1. Маркировка
2. Перемещение
3. Перераспределение

Этап перемещения, как и в G1, сосредоточивается на разреженных страницах — как и следовало ожидать от эвакуирующего сборщика мусора.

Для обеспечения непрерывного уплотнения C4 использует методику, именуемую *быстрым* (hand-over-hand) уплотнением. Она опирается на возможности системы виртуальной памяти — отсоединения (рассогласования) физического и виртуального адресов. При нормальной работе подсистема виртуальной памяти поддерживает отображение виртуальных страниц в адресном пространстве процесса на физические страницы памяти.

В отличие от HotSpot, которая не использует системные вызовы для управления памятью кучи Java, Zing использует вызовы функций ядра как часть цикла сборки мусора.

Эвакуация в Zing осуществляет перемещение объектов путем их копирования в другую страницу, которая, естественно, соответствует другому физическому адресу. После копирования всех объектов со страницы физическая страница может быть освобождена и возвращена операционной системе. Однако при этом в приложении будут ссылки, которые указывают на текущую, ныне не отображаемую на физическую память виртуальную страницу. Однако LVB позаботится о таких ссылках и исправит их до возникновения ошибки обращения к памяти.

Сборщик мусора C4 Zing в любой момент времени выполняет два алгоритма сборки: один — для молодых и один — для старых объектов. Это, очевидно, приводит к накладным расходам, но, как мы увидим в главе 8, "Протоколирование, мониторинг, настройка и инструменты сборки мусора", при настройке параллельного сборщика (такого, как CMS) с точки зрения накладных расходов и пропускной способности полезно предполагать, что при работе сборщик может находиться в режиме следования циклов один за другим (back-to-back). Это не так сильно отличается от режима непрерывной работы, которую демонстрирует C4.

В конечном счете инженер в области производительности должен тщательно изучить преимущества и компромиссы, связанные с переходом на Zing и сборщик C4. Философия "измеряйте, а не гадайте" применима к выбору виртуальной машины так же, как и ко многому другому.

Balanced (IBM J9)

IBM выпускает JVM под названием "J9". Исторически это была патентованная JVM, но сейчас IBM находится в процессе открытия исходных текстов и переименовывает ее в "Open J9". В этой виртуальной машине имеется несколько различных сборщиков мусора, которые могут быть включены — в том числе высокопроизводительный сборщик по умолчанию, подобный параллельному сборщику HotSpot.

Однако в этом разделе мы обсудим сборщик Balanced. Этот основанный на работе с регионами сборщик доступен в 64-разрядной JVM J9M и рассчитан на кучи размером более 4 Гбайт. Основными целями проекта являются:

- повышение масштабируемости времен пауз для больших куч Java;
- минимизация времен задержек в наихудшем случае;
- использование информации о производительности неравномерного доступа к памяти (non-uniform memory access — NUMA).

Для достижения первой цели куча разделяется на несколько регионов, которые управляются и собираются независимо. Как и G1, сборщик Balanced управляет не более чем 2048 регионами, а потому выбирает размер региона соответствующим образом. Размер региона представляет собой степень 2, как и у G1, но Balanced допускает наличие регионов размером 512 Кбайт.

Как и следовало ожидать от сборщика, работающего с поколениями, каждый регион имеет связанный с ним возраст, причем для размещения новых объектов используются регионы нулевого возраста (Эдем). Когда пространство Эдема заполнено, необходимо выполнить сборку мусора. IBM использует для этого действия термин *частичная сборка мусора* (partial garbage collection — PGC).

Частичная сборка мусора — это операция с приостановкой приложения, которая выполняет сборку мусора во всех регионах Эдема и дополнительно может выбрать для сборки регионы с более высоким возрастом (если сборщик решит, что сборка в них имеет смысл). Таким образом, частичные сборки мусора подобны смешанным сборкам G1.

Когда частичная сборка мусора завершена, возраст регионов, содержащих выжившие объекты, увеличивается на 1. Их иногда называют *регионами поколений* (generational regions).

Еще одним преимуществом по сравнению с другими стратегиями сборки мусора J9 является то, что выгрузка классов может выполняться инкрементно. Во время частичной сборки мусора Balanced может собирать загрузчики классов, являющиеся частью текущего множества сборки. Этим данный сборщик отличается от других сборщиков J9, в которых загрузчики классов могут собираться только во время глобальной сборки.

Один из недостатков заключается в том, что, поскольку PGC видит только регионы, выбранные для сборки мусора, этот тип сборки может страдать от плавающего мусора. Чтобы решить эту проблему, Balanced использует *этап глобальной маркировки* (global mark phase — GMP). Это частично параллельная операция, которая сканирует всю кучу Java, помечая мертвые объекты для сборки. После завершения GMP последующая PGC работает с этими данными. Таким образом, количество плавающего мусора в куче ограничено количеством объектов, ставших мертвыми со времени последнего запуска GMP.

Последний тип операции сборки мусора, выполняемый Balanced, — *глобальная сборка мусора* (global garbage collection — GGC). Это полная сборка с приостановкой приложения, которая уплотняет кучу. Она похожа на полные сборки, запускаемые в HotSpot при сбое параллельного режима.

Заголовки объектов J9

Основной заголовок объекта J9 представляет собой *слот класса* (class slot), размер которого составляет 64 бита (или 32 бита при включенных сжатых ссылках).

Сжатые ссылки (compressed references) являются выбором по умолчанию для куч размером менее 57 Гбайт; они похожи на сжатые обычные указатели на объекты HotSpot.

Однако заголовок может иметь дополнительные слоты в зависимости от типа объекта:

- синхронизируемые объекты имеют слоты мониторов;
- объекты, помещенные во внутренние структуры JVM, имеют слоты хешей.

Кроме того, слоты мониторов и хешей необязательно смежны с заголовком объекта — они могут храниться в любом месте объекта, используя пространство, которое в противном случае было бы не использовано из-за выравнивания. Схему объекта J9 можно увидеть на рис. 7.11.

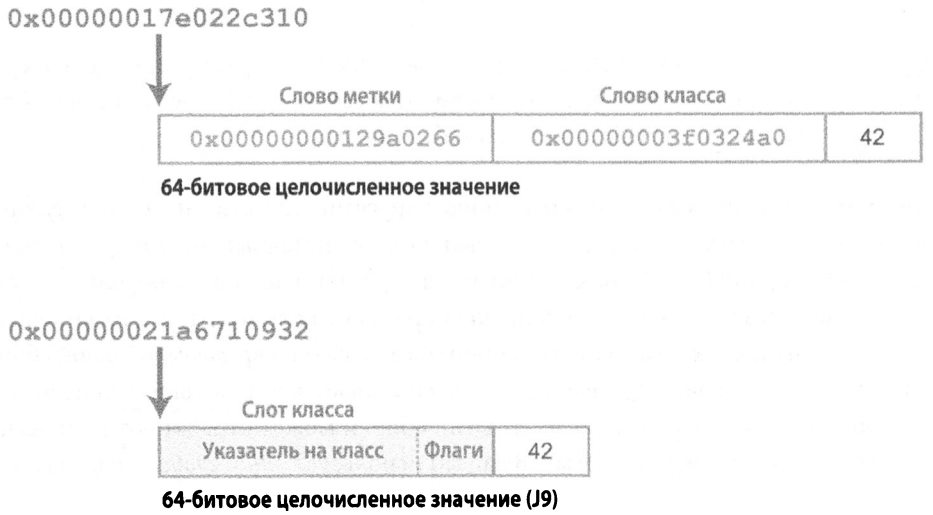

Рис. 7.11. Схема объекта J9

Старшие 24 (или 56) бит слота класса представляют собой указатель на структуру класса, которая находится вне кучи, подобно Metaspace в Java 8. Младшие 8 бит представляют собой флаги, которые применяются для различных целей, в зависимости от используемой стратегии сборки мусора.

Большие массивы в Balanced

Выделение больших массивов в Java является распространенным триггером для уплотняющих сборок, так как должно быть найдено достаточное количество непрерывного пространства для удовлетворения требований выделения памяти. Мы видели один такой аспект при обсуждении CMS, где объединения свободных списков иногда оказывалось недостаточно, чтобы освободить нужное пространство для выделения большого количества памяти, так что в результате получался сбой выделения в параллельном режиме.

Для сборщика мусора на основе регионов вполне возможно выделить в Java объект массива, который превышает размер одного региона. Чтобы решить эту проблему, для больших массивов Balanced использует альтернативное представление, позволяющее выделять память в несмежных блоках. Это представление известно как *эррейлеты* (arraylets), и это единственная ситуация, когда объекты кучи могут простираться на несколько регионов.

Представление эррейлета является невидимым для пользовательского кода Java и обрабатывается JVM прозрачно для него. Аллокатор будет представлять большой массив как центральный объект, называемый *стволом* (spine), и набор *листьев* (leaves) массива, которые содержат фактические записи массива и на которые указывают записи ствола. Это позволяет считывать записи с дополнительными накладными расходами всего лишь в одно косвенное обращение (рис. 7.12).

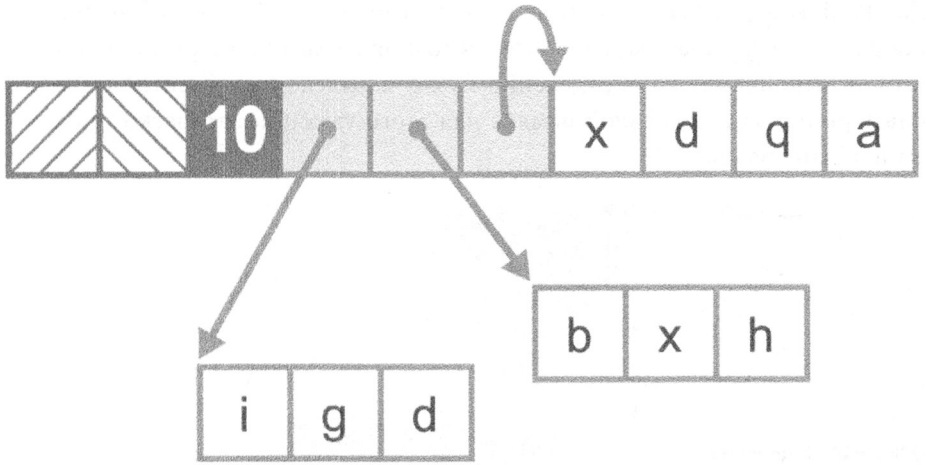

Рис. 7.12. Эррейлеты в J9

Такое представление массива потенциально видимо через API JNI (хотя и не из обычного кода Java), поэтому программист должен знать, что при переносе кода JNI из другой JVM может потребоваться учитывать представление в виде ствола и листьев.

Выполнение частичных сборок мусора в регионах уменьшает среднее время паузы, хотя общее время, затрачиваемое на выполнение операций сборки мусора, может оказаться выше из-за накладных расходов на поддержание информации о регионах ссылающегося объекта и объекта, на который выполняется ссылка.

Крайне важно, чтобы вероятность сборки или уплотнения с глобальной остановкой приложения (наихудшего случая в смысле времени паузы) была как можно ниже, поскольку это обычно происходит в предельном случае заполненной кучи.

Имеются свои накладные расходы на управление регионами и несмежными большими массивами, так что сборщик мусора Balanced подходит для приложений, в которых отсутствие больших пауз важнее пропускной способности.

NUMA и Balanced

Неравномерный доступ к памяти (non-uniform memory access — NUMA) — это архитектура памяти, используемая в многопроцессорных системах, обычно от средних до больших серверов. Такая система включает концепцию расстояния между памятью и процессором, причем процессоры и память расположены в узлах. Процессор на данном узле может получить доступ к памяти любого узла, но время доступа значительно короче для локальной памяти (т.е. для памяти, принадлежащей тому же узлу).

Для JVM, которые выполняются с несколькими узлами NUMA, сборщик Balanced может разделить кучу Java между ними. Потоки приложений устроены так, что предпочитают выполнение на определенном узле, а выделение объектов выполняется в пользу регионов в локальной памяти для этого узла. Схематически такая компоновка показана на рис. 7.13.

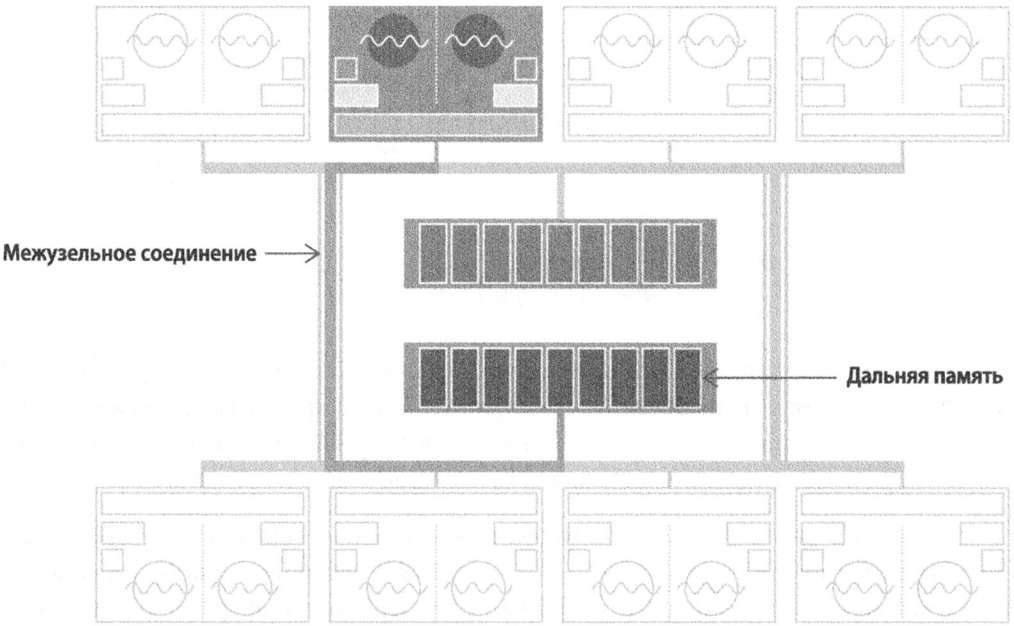

Рис. 7.13. Неравномерный доступ к памяти

Кроме того, частичная сборка мусора будет пытаться переместить объекты ближе (в смысле расстояния памяти) к объектам и потокам, которые на них ссылаются. Это

означает, что память, с которой работает поток, будет размещена более локально, что ведет к повышению производительности. Этот процесс для приложения невидим.

Старые сборщики мусора HotSpot

В более ранних версиях HotSpot были доступны другие сборщики мусора. Для полноты картины мы упомянем их здесь, но ни один из них больше не рекомендуется для производственного использования, а несколько комбинаций были объявлены устаревшими и нерекомендованными в Java 8 и окончательно запрещены (или удалены) в Java 9.

Serial и SerialOld

Сборщики мусора Serial и SerialOld работали аналогично сборщикам Parallel GC и ParallelOld, с одним важным отличием: они использовали только одно процессорное ядро для выполнения сборки мусора. Несмотря на это, они не были параллельными сборщиками, являясь сборщиками с полной приостановкой приложения. Разумеется, на современных многоядерных системах невозможно получить преимущества в производительности с использованием этих сборщиков, поэтому их не следует использовать.

В качестве инженера по производительности вы должны знать об этих сборщиках и их флагах, чтобы гарантировать, что если вы когда-либо столкнетесь с приложением, которое их устанавливает, то сможете их распознать и удалить.

Эти коллекторы устарели и не рекомендованы к употреблению еще в Java 8, поэтому они будут встречаться только в старых приложениях, которые все еще используют очень старые версии Java.

Incremental CMS (iCMS)

Сборщик мусора Incremental CMS, обычно именуемый iCMS, был древней попыткой разработать параллельную сборку путем внедрения в CMS некоторых идей, которые позже приведут к G1. Этот режим CMS включался с помощью следующего переключателя командной строки:

```
-XX:+CMSIncrementalMode
```

Некоторые эксперты по-прежнему утверждают, что существуют ситуации (для приложений, развернутых на очень старом оборудовании с одним или двумя ядрами), в которых iCMS может быть вполне допустимым выбором с точки зрения производительности. Однако практически все современные серверные приложения не должны использовать iCMS; в Java 9 этот сборщик мусора был удален.

Если только нет экстраординарных доказательств того, что ваша рабочая нагрузка выиграет от применения указанного сборщика мусора, вы не должны использовать его из-за недостатков, в частности, связанных с безопасностью.

Не рекомендуемые к применению и удаленные комбинации сборщиков мусора

Обычная процедура устаревания и удаления тех или иных функциональных возможностей заключается в том, что они помечаются как устаревшие и не рекомендуемые к употреблению в одной версии Java, а затем удаляются в следующей или более поздней версии. Соответственно, в Java 8 комбинации флагов сборщиков мусора, показанные в табл. 7.1, были отмечены как устаревшие, а в Java 9 были удалены.

Таблица 7.1. Устаревшие комбинации сборщиков мусора

Комбинация	Флаги
DefNew + CMS	`-XX:-UseParNewGC -XX:+UseConcMarkSweepGC`
ParNew + SerialOld	`-XX:+UseParNewGC`
ParNew + iCMS	`-Xincgc`
ParNew + iCMS	`-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC`
DefNew + iCMS	`-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC`
CMS переднего плана	`-XX:+UseCMSCompactAtFullCollection`
CMS переднего плана	`-XX:+CMSFullGCsBeforeCompaction`
CMS переднего плана	`-XX:+UseCMSCollectionPassing`

Начиная использовать новое взаимодействие, вы должны свериться с этой таблицей, чтобы убедиться, что настраиваемое приложение не использует устаревшую конфигурацию.

Epsilon

Сборщик Epsilon не является устаревшим сборщиком мусора. Он включен в этот раздел, потому что *не должен использоваться в производственных приложениях ни при каких обстоятельствах*. В то время как другие сборщики, если они встречаются в вашей среде, должны быть немедленно отмечены как чрезвычайно рискованные и предназначенные для немедленного удаления, Epsilon немного от них отличается.

Epsilon — экспериментальный сборщик мусора, предназначенный только для тестирования. Это коллекционер с *нулевым действием* (zeroeffort). Это означает, что он не предпринимает никаких действий по сборке мусора. Каждый байт памяти кучи, выделенный при работе с Epsilon, фактически является утечкой памяти. Он не может быть восстановлен, и это в конечном итоге ведет к тому, что JVM (вероятно, очень быстро) исчерпает память и приведет к сбою.

Разработан сборщик мусора, который обрабатывает только выделение памяти, но не реализует никакого реального механизма ее освобождения. Когда доступная куча Java исчерпана, выполните обычное выключение JVM.

— Документация Epsilon (JEP — JDK Enhancement Proposals)

Такой "сборщик мусора" может быть очень полезен для следующих целей:

- тестирование и микротестирование производительности;
- регрессионное тестирование;
- тестирование приложения или библиотечного кода Java с малым (или отсутствующим) выделением памяти.

В частности, тесты JMH выигрывают от способности уверенно исключать любые события сборки мусора из-за нарушения показателей производительности последними. Становятся также простыми в использовании регрессионные тесты распределения памяти, гарантирующие, что измененный код не сильно повлияет на поведение распределения. Разработчики могут писать тесты, которые выполняются такой конфигурацией Epsilon, которая разрешает только ограниченное количество выделений, а затем при любом дополнительном выделении приводит к сбою из-за исчерпания памяти кучи.

Наконец, предлагаемый интерфейс "виртуальная машина — сборщик мусора" также выигрывает от применения Epsilon как минимального тестового случая для самого интерфейса.

Резюме

Сборка мусора является поистине фундаментальным аспектом анализа и настройки производительности Java. Богатый набор сборщиков мусора Java — это большая сила платформы, но она может оказаться пугающей для новичков, в особенности с учетом дефицита документации, в которой рассказывалось бы о компромиссах и последствиях каждого выбора.

В этой главе мы изложили проблемы, стоящие перед инженерами по производительности, и компромиссы, на которые они должны идти, когда решают использовать тот или иной сборщик мусора для своих приложений. Мы рассмотрели некоторые из основополагающих теорий и встретились с рядом современных алгоритмов сборки мусора, которые реализуют эти идеи.

В следующей главе мы будем использовать теории на практике и познакомимся с протоколированием, мониторингом и соответствующим инструментарием, чтобы внести научную строгость в наше обсуждение настройки производительности сборки мусора.

Протоколирование, мониторинг, настройка и инструменты сборки мусора

В этой главе мы рассмотрим протоколирование и мониторинг сборки мусора. Это один из наиболее важных и заметных аспектов настройки производительности Java, и в то же время один из наиболее часто неправильно понимаемых.

Введение в протоколирование сборки мусора

Журнал сборки мусора представляет собой отличный источник информации. Он особенно полезен для анализа проблем постфактум, например при необходимости понять, почему произошел крах приложения. Это может позволить аналитику работать даже тогда, когда уже нет "живого" диагностируемого процесса.

Каждое серьезное приложение должно всегда

- генерировать журнал сборки мусора;
- хранить его в отдельном от вывода приложения файле.

Это особенно верно для промышленных приложений. Как мы увидим, протоколирование сборки мусора не имеет реальных наблюдаемых накладных расходов и поэтому всегда должно быть включено для любого важного процесса JVM.

Включение протоколирования сборки мусора

Первое, что нужно сделать, — это добавить некоторые переключатели запуска приложения. Лучше всего рассматривать их как "флаги обязательного протоколирования сборки мусора", которые должны быть включены для любого приложения Java/JVM (за исключением, возможно, приложений рабочего стола). Этими флагами являются:

```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintTenuringDistribution
-XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
```

Давайте рассмотрим каждый из этих флагов более подробно. Их назначение описано в табл. 8.1.

Таблица 8.1. Обязательные флаги сборки мусора

Флаг	Назначение
`-Xloggc:gc.log`	Указывает журнальный файл для записи событий сборки мусора
`-XX:+PrintGCDetails`	Запись подробностей событий сборки мусора
`-XX:+PrintTenuringDistribution`	Запись дополнительных подробностей событий сборки мусора, важных для инструментальной обработки
`-XX:+PrintGCTimeStamps`	Вывод времени события (в секундах от запуска виртуальной машины)
`-XX:+PrintGCDateStamps`	Вывод реального времени события

Инженерам по производительности следует принять к сведению следующую информацию о некоторых из этих флагов.

- Флаг `PrintGCDetails` замещает старую версию `verbose:gc`. Приложения должны удалить старый флаг.
- Флаг `PrintTenuringDistribution` отличается от других, поскольку предоставляемая им информация не воспринимается человеком непосредственно. Этот флаг предоставляет необработанные данные, необходимые для вычисления ключевых эффектов нехватки памяти и событий, таких как преждевременное повышение[1].
- Оба флага — `PrintGCDateStamps` и `PrintGCTimeStamps` — необходимы, так как первый используется для связи событий сборки мусора с системными событиями (в файлах журналов приложения), а последний — для связи сборки мусора и прочих внутренних событий JVM.

Этот уровень детализации протоколирования не оказывает заметного влияния на производительность JVM. Разумеется, объем генерируемых журналов зависит от многих факторов, в том числе от скорости распределения, используемого сборщика мусора и от размера кучи (более мелкие кучи нуждаются в сборке мусора более часто, и поэтому они будут генерировать журналы быстрее).

Определенное представление о ситуации дает тот факт, что 30-минутный запуск модельного примера аллокатора (см. главу 6, "Сборка мусора") дает примерно 600 Кбайт журнала при 30-минутном выделении 50 Мбайт/с.

В дополнение к обязательным флагам есть несколько флагов (показанных в табл. 8.2), которые управляют ротацией файлов журналов сборки мусора, которые

[1] Premature promotion — объекты в "молодом" поколении не утилизируются сборщиком мусора, а переводятся в "старое" поколение.

многие группы поддержки приложений считают очень полезными в производственных средах.

Таблица 8.2. Флаги ротации журналов сборки мусора

Флаг	Назначение
`-XX:+UseGCLogFileRotation`	Включение ротации журналов
`-XX:+NumberOfGCLogFiles=<n>`	Установка максимального количества поддерживаемых файлов журналов
`-XX:+GCLogFileSize=<size>`	Установка максимального размера файла перед ротацией

Настройка разумной стратегии ротации журналов должна проводиться совместно с работающим с приложением персоналом. Варианты такой стратегии и обсуждение соответствующих протоколов и инструментов выходит за рамки этой книги.

Журналы сборки мусора и JMX

В разделе "Мониторинг и инструментарий JVM" главы 2, "Обзор JVM", мы встретились с инструментом VisualGC, который способен отображать в реальном времени состояние кучи JVM. Этот инструмент фактически использует интерфейс Java Management eXtensions (JMX) для сбора данных из JVM. Полное обсуждение JMX выходит за рамки этой книги, но поскольку JMX воздействует на сборку мусора, инженер по производительности должен знать следующее.

- Файл журнала сборки мусора управляется фактическими событиями сборки, в то время как исходные данные JMX получаются путем выборки.
- Файл журнала сборки мусора имеет крайне малые накладные расходы, тогда как JMX имеет неявную стоимость применения прокси-серверов и вызовов удаленных методов (Remote Method Invocation — RMI).
- Файл журнала сборки мусора содержит более 50 аспектов данных о производительности, связанных с управлением памятью Java, в то время как JMX имеет их менее 10.

Традиционно преимуществом JMX перед журналами как источником данных о производительности является то, что JMX может предоставлять данные в виде потоков сразу, что называется "из коробки". Однако современный инструментарий, такой как jClarity Censum (см. раздел "Инструменты анализа журнала" далее в этой главе), предоставляет API для потока данных журнала сборки мусора, сводя это преимущество на нет.

Для грубого анализа тенденций базового использования кучи JMX является довольно быстрым и простым решением; однако для более глубокого диагноза проблем его быстро становится недостаточно.

Компоненты, доступные через JMX, являются стандартными и легко доступны. Инструмент VisualVM предоставляет один из способов визуализации соответствующих данных; на рынке имеется также множество других подобных инструментов.

Недостатки JMX

Клиенты, производящие мониторинг приложения с использованием JMX, обычно полагаются на выборку времени выполнения, чтобы получить обновление текущего состояния. Для получения непрерывного потока данных клиент должен постоянно опрашивать компоненты JMX во время выполнения.

В случае сборки мусора это приводит к проблеме: у клиента нет возможности узнать, когда запущен сборщик. Это также означает, что неизвестно состояние памяти до и после каждого цикла сборки мусора, что исключает выполнение более глубоких и более точных методов анализа данных сборки.

Анализ, основанный на данных JMX, по-прежнему полезен, но ограничен определением долгосрочных тенденций. Но если мы хотим точно настроить сборщик мусора, нам нужно что-то лучшее; в частности, чрезвычайно полезна возможность знать состояние кучи до и после каждой сборки мусора.

Кроме того, имеется множество чрезвычайно важных анализов, связанных с нагрузкой памяти (т.е. со скоростью ее выделения), которые просто невозможны из-за способа сбора данных в JMX.

Это еще не все. Текущая реализация спецификации `JMXConnector` основана на RMI. Таким образом, использование JMX подвержено всем проблемам, которые возникают с любым каналом связи на основе RMI. К ним относятся:

- открытие портов в брандмауэре, чтобы могли быть установлены вторичные подключения сокетов;
- применение прокси-объектов для облегчения вызова метода `remove()`;
- зависимость от финализации в Java (финализация в Java может задержать сборку мусора).

Для небольшого количества соединений RMI объем работы, связанной с закрытием соединения, является незначительным. Однако съем информации зависит от финализации. Это означает, что для освобождения объекта сборщик мусора должен работать.

Природа жизненного цикла соединения JMX означает, что чаще всего это приводит к тому, что объект RMI не будет собран до выполнения полной сборки мусора. Более подробная информация о влиянии финализации и о том, почему ее всегда следует избегать, можно найти в разделе "Избегайте финализации" главы 11, "Языковые методы повышения производительности".

По умолчанию любое приложение, использующее RMI, вызывает запуск полной сборки мусора один раз в час. Для приложений, которые уже используют RMI, использование JMX не будет привносить дополнительные затраты. Однако приложения, которые не используют RMI, обязательно получат дополнительные расходы, если решат использовать JMX.

Преимущества данных журналов сборки мусора

Современные сборщики мусора состоят из многих разных частей, которые, будучи взяты вместе, приводят к чрезвычайно сложной реализации. Настолько сложной, что производительность сборщика трудно, если вовсе неневозможно, предсказать. Эти типы программных систем известны как системы *внезапного запуска* (emergent), поскольку их конечное поведение и производительность являются следствием работы всех компонентов. Различные нагрузки на разные компоненты и их различная реакция приводят к модели с очень высокой изменчивостью.

Первоначально разработчики сборки мусора в Java добавили протоколирование для помощи в отладке своих реализаций, а следовательно, значительная часть данных, создаваемых с участием почти 60 связанных со сборкой мусора флагов, предназначена для отладки производительности.

Со временем те, кому было поручено настраивать процесс сборки мусора в своих приложениях, стали признавать, что, учитывая сложности настройки системы сборки мусора, они также получили выгоду от точной картины того, что происходит во время выполнения. Таким образом, возможность собирать и читать журналы сборки мусора теперь является инструментальной частью любого режима настройки.

Протоколирование сборки мусора внутри HotSpot JVM выполняется с использованием механизма неблокирующей записи. Он имеет практически нулевое влияние на производительность приложения и должен быть включен для всех производственных приложений.

Поскольку необработанные данные в журналах сборки мусора могут быть привязаны к конкретным событиям сборки, мы можем выполнять на их основе всевозможные полезные анализы, могущие дать представление о стоимости сборки и, следовательно, о том, какие действия по настройке с большей вероятностью приведут к положительным результатам.

Инструменты анализа журнала

Для сообщений журнала сборки мусора нет никакого стандартного формата, определяемого языком или спецификацией виртуальной машины. Это оставляет содержание любого отдельного сообщения на волю прихоти команды разработчиков

сборки мусора в HotSpot. Форматы могут изменяться (и меняются) между различными версиями, в том числе при незначительных изменениях версий.

Ситуация еще более усложняется тем, что в то время как простейшие форматы журналов легко анализируются, по мере добавления флагов получающийся в итоге журнал становится значительно более сложным. Это в особенности верно для журналов, генерируемых параллельными сборщиками мусора.

Слишком часто системы с ручными анализаторами журналов выходят из строя после внесения некоторых изменений в конфигурацию сборщика мусора, изменяющих формат записываемых в журнал данных. Нередко при изучении причин сбоя и анализа журналов команда обнаруживает, что доморощенный анализатор не в состоянии справиться с изменившимся форматом журнала — именно в тот момент, когда информация в журнале наиболее важна и нужна.

Разработчикам не рекомендуется самостоятельно анализировать журналы сборки мусора. Вместо этого следует использовать соответствующий инструментарий.

В этом разделе мы рассмотрим два активно поддерживаемых инструмента (коммерческий и с открытым исходным кодом), доступных для этой цели. Есть и другие инструменты, такие как GarbageCat, поддерживаемые только спорадически (а то и не поддерживаемые вовсе).

Censum

Анализатор памяти Censum представляет собой коммерческий инструмент, разработанный jClarity. Он доступен и как настольный инструмент (для практического анализа одной JVM), и как служба мониторинга (для больших групп JVM). Назначением этого инструмента является предоставление наилучшего доступного анализа журнала сборки мусора, извлечение информации и автоматической аналитики.

На рис. 8.1 представлена настольная версия Censum, показывающая уровни выделения для финансового торгового приложения, использующего сборщик мусора G1. Даже из этого простого представления мы видим, что данное приложение имеет периоды очень низкого выделения, соответствующие затишью на рынке.

Еще одно представление, доступное в Censum, — график времени паузы, показанный на рис. 8.2.

Одним из преимуществ использования службы мониторинга Censum SAAS является возможность одновременного просмотра работоспособности всего кластера, как показано на рис. 8.3. Для постоянного мониторинга это обычно проще, чем одновременно работать только с одной JVM.

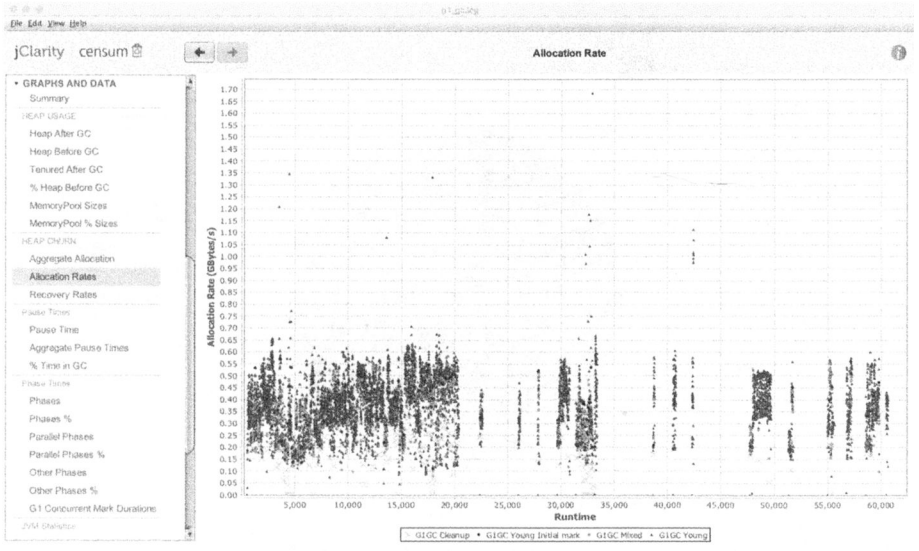

Рис. 8.1. Представление выделения памяти в Censum

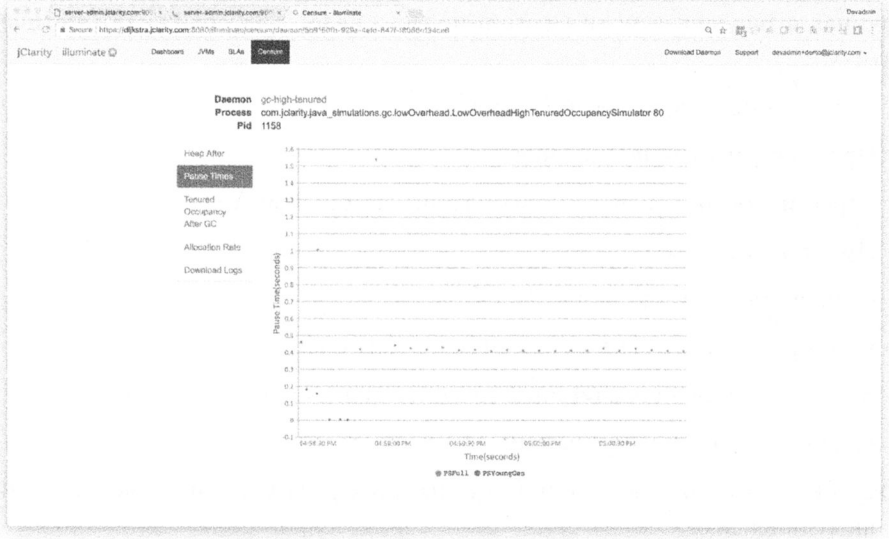

Рис. 8.2. Представление времени пауз в Censum

При разработке Censum особое внимание было уделено форматам журналов с отслеживанием всех изменений в OpenJDK, которые могут повлиять на ведение журнала. Censum поддерживает все версии Sun/Oracle Java от 1.4.2 до текущих, все сборщики мусора и наибольшее количество конфигураций журнала сборки по сравнению с любым иным инструментом на рынке.

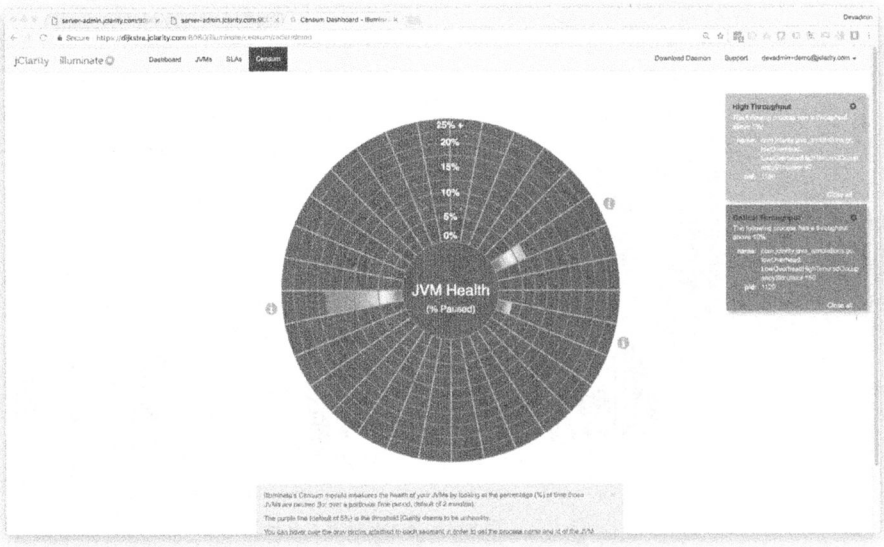

Рис. 8.3. Обзор работоспособности кластера в Censum

Начиная с текущей версии Censum поддерживает следующую автоматическую аналитику.

- Точные скорости выделений памяти
- Преждевременное повышение
- Агрессивное выделение или выделение с выбросами
- Простой системы
- Обнаружение утечек памяти
- Планирование размера кучи и рабочей нагрузки
- Взаимодействие операционной системы с виртуальной машиной
- Неудачные размеры пулов памяти

Подробнее о Censum можно прочесть на сайте jClarity[2]; там же можно получить и пробную лицензию.

GCViewer

GCViewer представляет собой настольный инструмент, который предоставляет некоторые основные возможности анализа и графического представления журнала сборки мусора. Его важной положительной стороной является то, что он является бесплатным программным обеспечением с открытым исходным кодом. Однако по

[2] `https://www.jclarity.com/`

сравнению с коммерческими инструментами в нем не хватает множества возможностей.

Чтобы использовать GCViewer, необходимо загрузить его исходные тексты[3]. После компиляции и построения можно упаковать файлы в исполняемый JAR-файл.

После этого журнальные файлы сборки мусора можно открыть в главном окне GCViewer (рис. 8.4).

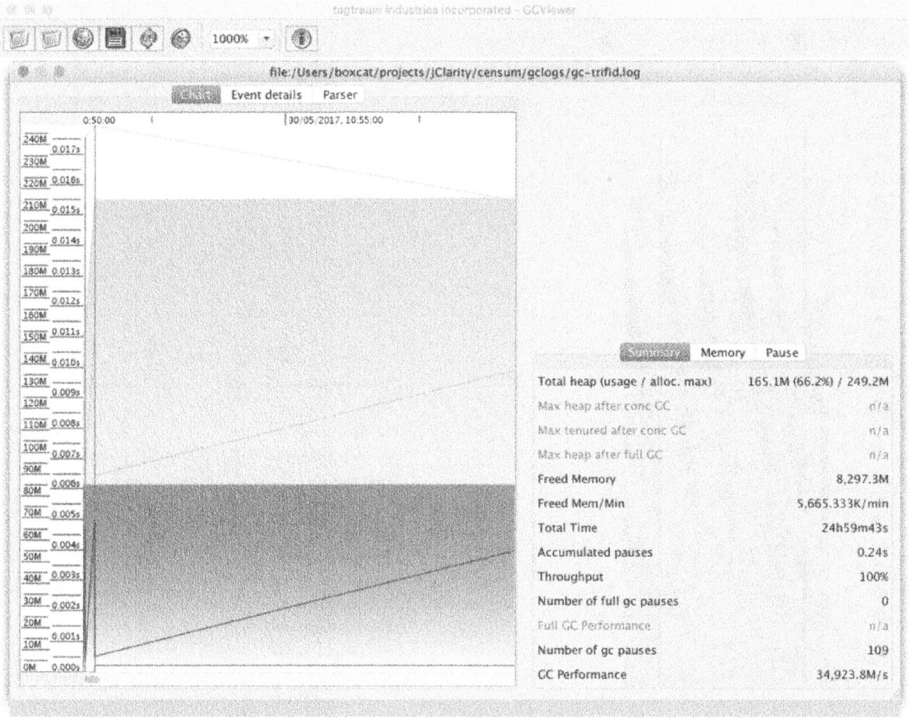

Рис. 8.4. GCViewer

GCViewer испытывает недостаток в аналитике и может анализировать только некоторые из возможных форматов журнала сборки мусора, которые могут создаваться в HotSpot.

Можно использовать GCViewer в качестве библиотеки анализа и экспортировать данные в визуализатор, но для этого требуется дополнительная разработка поверх существующей базы с открытым исходным кодом.

Различные визуализации одних и тех же данных

Следует иметь в виду, что различные инструменты могут создавать различные визуализации одних и тех же данных. Простой пилообразный шаблон, который мы

[3] `https://github.com/chewiebug/GCViewer`

видели в разделе "Роль выделения памяти" главы 6, "Сборка мусора", основан на выборке общего размера кучи в качестве наблюдаемого объекта.

Когда журнал сборки мусора, который генерирует такой шаблон, строится с помощью представления "Heap Occupancy after GC" ("Занятость кучи после сборки мусора") GCViewer, в результате мы имеем визуализацию наподобие показанной на рис. 8.5.

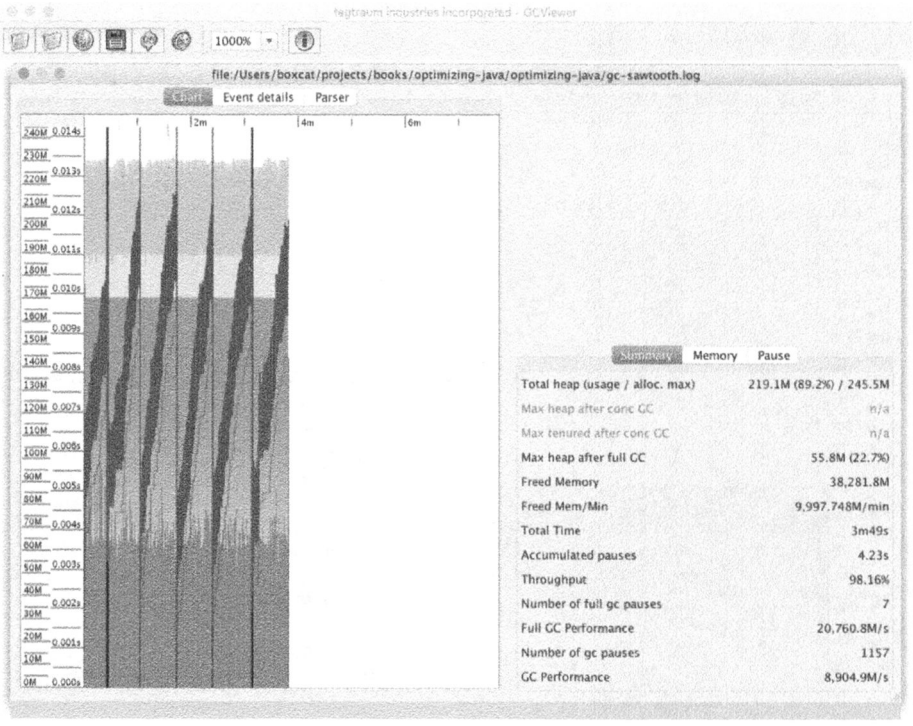

Рис. 8.5. Простой пилообразный шаблон в GCViewer

Теперь взгляните на тот же простой пилообразный шаблон, выведенный в Censum (рис. 8.6).

Хотя представленные образы и визуализации различны, инструмент сообщает в обоих случаях одно и то же — куча функционирует нормально.

Базовая настройка сборки мусора

Когда инженеры рассматривают стратегии настройки JVM, часто встает вопрос "Когда следует настраивать сборку мусора?" Как и в случае любой другой техники настройки, настройка сборки мусора должна составлять часть общего диагностического процесса. Очень полезно запомнить следующие факты о настройке сборки мусора.

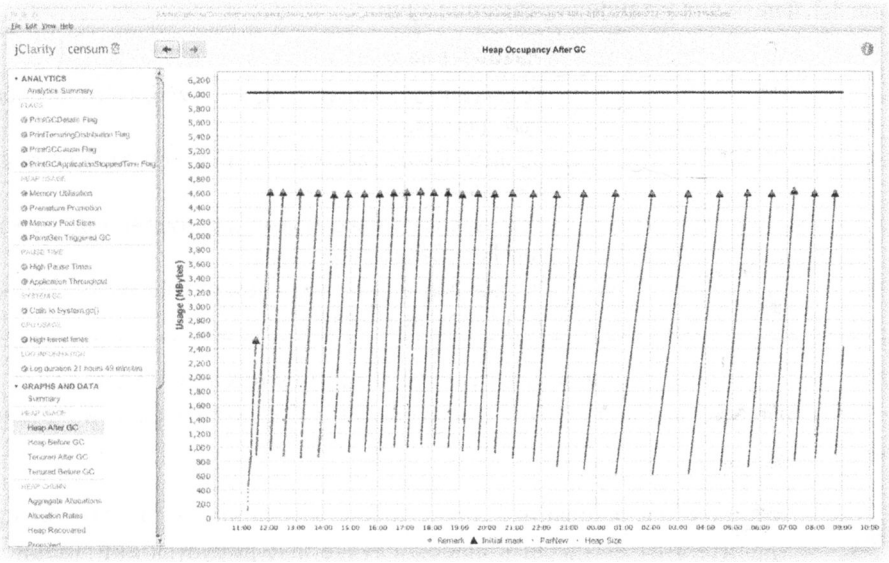

Рис. 8.6. Простой пилообразный шаблон в Censum

1. Подтвердить или опровергнуть утверждение о том, что сборка мусора является причиной проблем с производительностью, достаточно просто и недорого.
2. Легко и недорого добавить флаги сборки мусора во время приемного тестирования (UAT).
3. Настроить профайлеры производительности или выполнения — задача не из дешевых.

Инженеры должны также знать, что во время настройки следует изучить и измерить четыре основных фактора.

- Выделение памяти
- Чувствительность к паузам
- Поведение пропускной способности
- Время жизни объектов

Очень часто самым важным из этого списка является выделение памяти.

Пропускная способность может зависеть от целого ряда факторов, например от того факта, что параллельные сборщики во время своей работы занимают ядра процессора.

Давайте взглянем на некоторые из основных флагов, связанных с установкой размеров кучи, перечисленные в табл. 8.3.

Таблица 8.3. Флаги для установки размеров кучи

Флаг	Назначение
`-Xms<size>`	Устанавливает минимальный размер, зарезервированный для кучи
`-Xmx<size>`	Устанавливает максимальный размер, зарезервированный для кучи
`-XX:MaxPermSize=<size>`	Устанавливает максимальный разрешенный размер для PermGen (Java 7)
`-XX:MaxMetaspaceSize=<size>`	Устанавливает максимальный разрешенный размер для Metaspace (Java 8)

Флаг `MaxPermSize` является устаревшим и применяется только в Java 7 и более ранних версиях. Начиная с Java 8 и выше PermGen был удален и заменен на Metaspace.

Если вы настраиваете `MaxPermSize` для приложения Java 8, просто удалите этот флаг. Он в любом случае игнорируется виртуальной машиной и никак не влияет на ваше приложение.

Что касается дополнительных флагов для настройки сборки мусора, то:

- добавляйте одновременно только по одному флагу;
- убедитесь, что вы понимаете действие каждого флага;
- помните, что некоторые комбинации обладают побочным действием.

Проверка того факта, что сборка мусора является причиной проблем с производительностью, относительно проста в предположении, что событие происходит "вживую". Первым шагом является просмотр высокоуровневых показателей машины с помощью `vmstat` или аналогичного инструмента, как описано в разделе "Основные стратегии обнаружения источников проблем" главы 3, "Аппаратное обеспечение и операционные системы". Сначала подключитесь к машине с проблемой производительности и убедитесь, что:

- уровень использования процессора близок к 100%;
- подавляющее большинство времени (более 90%) затрачивается на работу в пользовательском пространстве;
- журнал сборки мусора демонстрирует активность, показывая, что сборка мусора в настоящий момент работает.

Это предполагает, что проблема есть прямо сейчас и инженер может ее наблюдать в режиме реального времени. Для предыдущих событий должны быть доступны достаточные данные мониторинга (включая использование процессора и журналы сборки мусора, имеющие временные метки).

Если все три перечисленные условия выполнены, то в качестве наиболее вероятной причины текущей проблемы производительности следует исследовать и настроить сборку мусора. Проверка очень проста и дает четкий результат: либо "со сборкой мусора все в порядке", либо "со сборкой мусора проблемы".

Если GC оказывается источником проблем с производительностью, то следующим шагом будет изучение поведения выделения памяти и времени пауз, затем — настройка сборки мусора и при необходимости — применение профайлера памяти.

Исследование выделения памяти

Анализ скорости выделения памяти чрезвычайно важен для определения не только того, как настроить сборщик мусора, но и действительно ли вы можете настроить сборщик мусора так, чтобы улучшить производительность.

Мы можем использовать данные из событий молодого поколения, чтобы рассчитать объем выделенных данных и время между двумя сборками. Затем эту информацию можно использовать для расчета средней скорости выделения памяти в течение этого временного интервала.

Вместо того чтобы тратить время и усилия, вручную рассчитывая скорости выделения памяти, как правило, лучше использовать соответствующий инструмент.

Опыт показывает, что устойчивые скорости выделения памяти 1 Гбайт/с почти всегда указывают на проблемы с производительностью, которые не могут быть исправлены путем настройки сборщика мусора. Единственный способ повысить производительность в таких случаях — повысить эффективность использования памяти с помощью рефакторинга, чтобы исключить выделение памяти в критических частях приложения.

Простая гистограмма памяти, демонстрируемая VisualVM (см. раздел "Мониторинг и инструментарий JVM" главы 2, "Обзор JVM") или даже `jmap` (раздел "Алгоритм маркировки и выметания" главы 6, "Сборка мусора"), может служить отправной точкой для понимания поведения выделения памяти. Полезной стратегией первоначального размещения является концентрация на четырех простых областях:

- тривиальное выделение объектов, которого можно избежать (например, сообщения журнала отладки);
- расходы на упаковку;
- объекты предметной области;
- большое количество объектов каркаса, не относящегося к JDK.

Первый случай представляет собой просто вопрос обнаружения и удаления ненужного создания объекта. Чрезмерная упаковка может быть разновидностью этой

ситуации, но возможными источниками расточительного создания объектов могут быть и другие действия (такие, как автоматически генерируемый код для сериализации/десериализации для JSON, или код ORM).

Объекты предметной области редко являются основным источником использования памяти в приложении. Гораздо более распространенными являются такие типы, как:

- `char[]`: символы, составляющие строки;
- `byte[]`: необработанные бинарные данные;
- `double[]`: данные вычислений;
- записи отображений;
- `Object[]`;
- внутренние структуры данных (такие, как methodOops и klassOops).

Простая гистограмма может часто обнаруживать утечку или чрезмерное создание ненужных объектов предметной области просто по их присутствию на вершине гистограммы кучи. Часто все, что необходимо, — это быстрый расчет для определения ожидаемого объема данных объектов предметной области, чтобы увидеть, соответствует ли наблюдаемый объем ожиданиям.

В разделе "Выделение памяти, локальное для потока" главы 6, "Сборка мусора", мы встречались с локальным для потоков выделением памяти, целью которого является предоставление закрытой области для каждого потока для размещения новых объектов и, таким образом, достижение выделения памяти за время O(1).

Размеры TLAB меняются динамически для каждого потока, а регулярные объекты выделяются в TLAB, если там имеется пространство для этого. Если же нет, поток запрашивает новый TLAB у виртуальной машины и повторяет попытку.

Если объект не поместится в пустой TLAB, то виртуальная машина попытается выделить объект непосредственно в Эдеме, в области вне TLAB. Если это не удастся, следующим шагом будет выполнение сборки мусора молодого поколения (что может привести к изменению размера кучи). Наконец, если и это не удастся и свободной памяти все еще будет недостаточно, будет испытано последнее средство — передать объект непосредственно в область бессрочных объектов.

Из этого можно видеть, что единственными объектами, для которых память действительно может быть выделена непосредственно в области бессрочных объектов, являются большие массивы (особенно массивы байтов и символов).

У HotSpot есть пара флагов настройки, которые относятся к TLAB и распределению больших объектов:

```
-XX:PretenureSizeThreshold=<n>
-XX:MinTLABSize=<n>
```

Как и все прочие переключатели, они не должны использоваться без проверок и убедительных доказательств того, что они будут давать требуемый результат. В большинстве случаев встроенное динамическое поведение даст отличные результаты, и любые изменения практически не будут иметь видимого результата.

Скорости выделения памяти также влияют на количество объектов, которые были повышены в бессрочные. Если предположить, что недолговечные Java-объекты имеют фиксированное время жизни, то более высокие скорости выделения памяти приведут к тому, что сборка мусора молодого поколения будет выполняться чаще. Если сборки становятся слишком частыми, то недолговечные объекты, возможно, не будут успевать умереть и будут ошибочно продвигаться в бессрочные.

Другими словами, всплески выделения памяти могут привести к проблеме преждевременного повышения, с которой мы встретились в разделе "Роль выделения памяти" главы 6, "Сборка мусора". Чтобы избежать этого, JVM будет динамически изменять размеры пространства выживших, чтобы иметь возможность принять больший объем выживших данных без повышения объектов в бессрочные. Иногда для решения проблем бессрочных объектов и преждевременного повышения может быть полезен один переключатель JVM:

```
-XX:MaxTenuringThreshold=<n>
```

Он контролирует количество сборок мусора, в которых объект должен выжить, чтобы быть повышенным до бессрочного. По умолчанию это значение равно 4, но может быть установлено любым от 1 до 15. Изменение этого значения представляет собой компромисс между двумя проблемами:

- чем выше этот порог, тем больше будет копироваться действительно долгоживущих объектов;
- если порог слишком низок, будет выполняться повышение некоторых короткоживущих объектов и увеличится нагрузка на область бессрочных объектов.

Одним из следствий слишком низкого порога может быть более частое выполнение полных сборок мусора из-за большего объема объектов, повышаемых до бессрочных, что приводит к более быстрому заполнению соответствующей области памяти. Как всегда, вы не должны использовать переключатель без тестов производительности, четко указывающих на то, что вы улучшите производительность путем установки значения, отличного от значения по умолчанию.

Исследование времени паузы

Разработчики часто страдают от когнитивного искажения относительно времени паузы. Многие приложения могут легко допустить время паузы в 100 мс и более. Человеческий глаз может обрабатывать только 5 обновлений одного элемента данных в секунду, поэтому пауза в 100–200 мс оказывается ниже порога видимости

для большинства приложений, ориентированных на человека (например, веб-приложений).

Одна полезная эвристика настройки состоит в том, чтобы разделять приложения на три широких диапазона времени паузы. Эти диапазоны основаны на потребности приложений в реагировании, выражаемом как время паузы, которое может допускать приложение.

1. Более 1 с.
2. От 1 с до 100 мс.
3. Менее 100 мс.

Если мы объединим чувствительность приложения ко времени паузы с ожидаемым размером кучи, то сможем построить простую таблицу предполагаемых лучших сборщиков для каждого случая. Результат показан в табл. 8.4.

Таблица 8.4. Первоначальный выбор сборщика мусора

Допустимое время паузы			Размер кучи
> 1 с	1 с – 100 мс	< 100 мс	
Parallel	Parallel	CMS	< 4 GB
Parallel	Parallel/G1	CMS	< 4 GB
Parallel	Parallel/G1	CMS	< 10 GB
Parallel/G1	Parallel/G1	CMS	< 20 GB
Parallel/G1	G1	CMS	> 20 GB

Эти руководящие принципы и эмпирические правила предназначены быть отправной точкой при настройке, а не однозначными стопроцентными правилами.

Заглядывая в будущее, по мере достижения сборщиком G1 определенных зрелости и законченности, разумно ожидать, что он будет охватывать все больше ситуаций, в которых в настоящее время применяется ParallelOld. Также возможно (но, пожалуй, менее вероятно), что он будет расширяться и теснить также CMS.

Когда используется параллельный сборщик, необходимо уменьшить выделение памяти, прежде чем пытаться настроить время паузы. Уменьшенное выделение памяти означает, что недостаток памяти у параллельного коллектора будет меньше; циклам сборки мусора будет легче идти в ногу с потоками выделения. Это уменьшит вероятность сбоев в параллельном режиме, что обычно является событием, которого (если это вообще возможно) должны избегать приложения, чувствительные к паузе.

Потоки сборщика и корни сборки мусора

Одно полезное умственное упражнение — "думать, как поток сборки мусора". Это может дать представление о том, как сборщик ведет себя при различных

обстоятельствах. Однако, как и во многих других аспектах сборки мусора, здесь имеются фундаментальные компромиссы (взаимовлияния), например то, что время сканирования, необходимое для определения корней сборки мусора, зависит от следующих факторов.

- Количество потоков приложения
- Количество скомпилированного кода в кеше кода
- Размер кучи

Даже при рассмотрении этого единственного аспекта сборки мусора то, какое именно из перечисленных событий будет доминирующим в сканировании корней сборки мусора, зависит от условий выполнения сборки и степени применяемого параллелизма.

Например, рассмотрим случай большого массива `Object[]`, обнаруженного на этапе маркировки. Он будет сканирован единственным потоком; никакая передача работы невозможна. В экстремальном случае это время однопоточного сканирования будет доминировать в общем времени маркировки.

Фактически чем сложнее граф объектов, тем более выраженным становится этот эффект, и это означает, что время маркировки становится тем хуже, чем больше в графе имеется "длинных цепочек" объектов.

Большое количество потоков приложения также будет влиять на время сборки мусора, поскольку они представляют больше кадров стека, которые следует сканировать, и для достижения точки безопасности требуется больше времени. Они также оказывают большее давление на планировщики потоков как в аппаратных, так и в виртуализированных средах.

Помимо этих традиционных примеров корней сборки мусора, существуют другие источники корней, включая кадры JNI и кеш JIT-скомпилированного кода (с которым мы встретимся в разделе "Кеш кода" главы 9, "Выполнение кода в JMV").

Сканирование корня сборки мусора в кеше кода является однопоточным (по крайней мере, в Java 8).

Из указанных трех факторов сканирование стека и кучи достаточно хорошо распараллеливается. Сборщики поколений также отслеживают корни, происходящие из других пулов памяти, используя такие механизмы, как RSets в G1 и таблицы карт в Parallel GC и CMS.

Например, рассмотрим таблицы карт, представленные в разделе "Слабая гипотеза поколений" главы 6, "Сборка мусора". Они используются для указания блока памяти, который имеет ссылку в молодое поколение из старого. Поскольку каждый байт представляет 512 байтов старого поколения, ясно, что для каждого гигабайта памяти старого поколения необходимо просканировать 2 Мбайта таблицы карт.

Чтобы прочувствовать, насколько длинным является сканирование таблицы карт, рассмотрим простой тест производительности, который имитирует сканирование таблицы карт для кучи размером 20 Гбайт:

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(1)
public class SimulateCardTable
{
    // Старое поколение занимает 3/4 кучи; для 1Гбайта старшего
    // поколения требуется таблица карт размером 2 Мбайта
    private static final int SIZE_FOR_20_GIG_HEAP = 15*2*1024*1024;
    private static final byte[] cards = new byte[SIZE_FOR_20_GIG_HEAP];
    @Setup
    public static final void setup()
    {
        final Random r = new Random(System.nanoTime());

        for (int i = 0; i < 100_000; i++)
        {
            cards[r.nextInt(SIZE_FOR_20_GIG_HEAP)] = 1;
        }
    }
    @Benchmark
    public int scanCardTable()
    {
        int found = 0;

        for (int i = 0; i < SIZE_FOR_20_GIG_HEAP; i++)
        {
            if (cards[i] > 0)
                found++;
        }

        return found;
    }
}
```

Выполнение этого теста дает на выходе примерно следующий результат:

```
Result "scanCardTable":
    108.904 ±(99.9%) 16.147 ops/s [Average]
    (min, avg, max) = (102.915, 108.904, 114.266), stdev = 4.193
    CI (99.9%): [92.757, 125.051] (assumes normal distribution)

# Run complete. Total time: 00:01:46
```

```
Benchmark Mode Cnt Score Error Units
SimulateCardTable.scanCardTable thrpt 5 108.904 ± 16.147 ops/s
```

Как видите, для сканирования таблицы карт для кучи размером 20 Гбайт требуется около 10 мс. Это, конечно, результат для однопоточного сканирования; однако он дает полезную грубую нижнюю оценку для времени паузы для молодых коллекций.

Мы рассмотрели некоторые общие методы, которые должны быть применимы для настройки большинства сборщиков, так что теперь давайте перейдем к рассмотрению некоторых подходов для конкретных сборщиков мусора.

Настройка Parallel GC

Parallel GC является простейшим из сборщиков мусора, так что не должно быть удивительным, что настроить его проще всего. Однако обычно ему требуется только минимальная настройка. Цели и компромиссы Parallel GC очевидны:

- сборщик мусора с полной приостановкой приложения;
- высокая пропускная способность сборщика при низкой вычислительной стоимости;
- отсутствие возможности частичной сборки мусора;
- линейная зависимость роста времени паузы от размера кучи.

Если ваше приложение устраивают характеристики Parallel GC, то он может стать очень эффективным выбором, особенно в небольших кучах, например размером до 4 Гбайт.

Старые приложения могли использовать явные флаги размера для управления относительным размером различных пулов памяти. Эти флаги перечислены в табл. 8.5.

Таблица 8.5. Старые флаги для размеров кучи

Флаг	Назначение
`-XX:NewRatio=<n>`	(Старый флаг) Устанавливает отношение размера молодого поколения к размеру кучи
`-XX:SurvivorRatio=<n>`	(Старый флаг) Устанавливает отношение размера области выживших к размеру молодого поколения
`-XX:NewSize=<n>`	(Старый флаг) Устанавливает минимальный размер молодого поколения
`-XX:MaxNewSize=<n>`	(Старый флаг) Устанавливает максимальный размер молодого поколения
`-XX:MinHeapFreeRatio`	(Старый флаг) Устанавливает минимальный процент свободной кучи после сборки мусора во избежание расширения
`-XX:MaxHeapFreeRatio`	(Старый флаг) Устанавливает максимальный процент свободной кучи после сборки мусора во избежание сжатия

Для указанных флагов действуют следующие соотношения:

```
Установленные флаги:
    -XX:NewRatio=N
    -XX:SurvivorRatio=K

Молодое поколение = 1 / (N+1) от размера кучи
Старое поколение  = N / (N+1) от размера кучи

Эдем = (K - 2) / K от размера молодого поколения
Выжившие1 =  1 / K от размера молодого поколения
Выжившие2 =  1 / K от размера молодого поколения
```

Для большинства современных приложений эти явные указания размеров использоваться не должны, поскольку эргономичный размер почти во всех случаях будет работать лучше, чем люди. Прибегать к этим переключателям для Parallel GC — это что-то близкое к последнему шансу.

Настройка CMS

Сборщик мусора CMS имеет репутацию трудно настраиваемого. Для этого имеются определенные основания: сложности и ограничения, связанные с достижением наивысшей производительности CMS, не следует недооценивать.

К сожалению, среди многих разработчиков распространена упрощенная позиция "время паузы — плохо, поэтому параллельно маркирующий сборщик — хорошо". Сборщик с низким временем паузы, такой как CMS, в действительности следует рассматривать как последнее средство, которое должно использоваться, только если действительно требуется очень небольшое время приостановки выполнения приложения. Если вы поступите иначе, это может означать, что команда отягощена сборщиком мусора, который трудно настроить и который не дает реального ощутимого преимущества для производительности приложения.

CMS имеет очень большое количество флагов (почти 100 в Java 8u131), и некоторые разработчики могут испытывать соблазн начать изменять значения этих флагов в попытке повысить производительность. Однако это может легко привести к нескольким антипаттернам, с которыми мы уже встречались в главе 4, "Паттерны и антипаттерны тестирования производительности", в том числе к таким.

- Надувательство с переключателями
- Настройка по совету
- Отсутствие общей картины

По соображениям производительности разработчики должны противостоять искушению пасть жертвой в любой из этих когнитивных ловушек.

Большинство приложений, использующих CMS, при изменении значений флагов командной строки, вероятно, не увидят никаких реально наблюдаемых улучшений.

Несмотря на эту опасность, существуют некоторые обстоятельства, при которых для улучшения (или получения приемлемой) производительности настройка CMS необходима. Начнем с рассмотрения поведения пропускной способности.

Во время работы сборщика CMS сборкой по умолчанию занимается половина ядер. Это неизбежно приводит к сокращению пропускной способности. Одно полезное практическое правило может состоять в том, чтобы рассмотреть, какой была ситуация со сборщиком непосредственно перед сбоем параллельного режима.

В этом случае, как только одна сборка CMS завершится, сразу же начнется новая сборка CMS. Такая ситуация известна как *непрерывная сборка* (back-to-back collection), и в случае параллельного сборщика указывает, что параллельная сборка близка к краху. Если приложение будет выделять память быстрее, восстановление не сможет соответствовать выделению и в результате мы получим сбой параллельного режима.

В случае непрерывной сборки пропускная способность будет уменьшена на 50%, по сути, для всего приложения. При выполнении настройки инженер должен рассмотреть, допустимо ли для данного приложения такое наихудшее поведение. Если нет, решение следует запустить на машине с большим количеством доступных ядер.

Альтернативным решением является уменьшение количества ядер, назначаемых сборке мусора во время цикла CMS. Конечно, это опасно, так как это уменьшает количество процессорного времени, доступного для выполнения сборки, а потому делает приложение менее устойчивым к всплескам выделения памяти (и в свою очередь, может сделать его более уязвимым для сбоя параллельного режима). Вот переключатель для управления этим параметром:

```
-XX:ConcGCThreads=<n>
```

Должно быть очевидно, что если приложение не может достаточно быстро освобождать память с параметром по умолчанию, то сокращение количества потоков сборки мусора будет только ухудшать положение.

CMS имеет два отдельных этапа с приостановкой приложения.

Первоначальная маркировка

Маркирует внутренние узлы, на которые непосредственно указывают корни сборки мусора.

Повторная маркировка

Использует таблицы карт для идентификации объектов, которые могут потребовать действий по исправлению.

Это означает, что в пределах одного цикла CMS дважды требуются остановка всех потоков приложения, а следовательно, и точка безопасности. Этот эффект может стать важным для некоторых приложений с низкой задержкой, которые чувствительны к поведению точек безопасности.

Вот два флага, которые часто видят вместе:

```
-XX:CMSInitiatingOccupancyFraction=<n>
-XX:+UseCMSInitiatingOccupancyOnly
```

Эти флаги могут быть очень полезными. Они также еще раз иллюстрируют проблему нестабильных скоростей выделения памяти.

Флаг инициирующей занятости памяти используется для определения того, когда CMS должен начать сборку мусора. Для успешной работы требуется некоторый запас памяти в куче — в качестве свободного места для объектов, которые будут перемещаться из сборок молодых поколений, которые могут запускаться во время работы CMS.

Как и многие другие аспекты подхода HotSpot к сборке мусора, размер этого запасного пространства контролируется статистикой, собираемой самой JVM. Однако для первого запуска CMS по-прежнему требуется оценка. Размер запаса для этой первоначальной оценки контролируется флагом `CMSInitiatingOccupancyFraction`. Значение по умолчанию для этого флага означает, что первая полная сборка мусора CMS начнется, когда куча будет заполнена на 75%.

Если установлен также флаг `UseCMSInitiatingOccupancyOnly`, то динамическое указание размера для начального заполнения отключается. Не следует относиться к этому легкомысленно. На практике редко приходится уменьшать запас (увеличивать значение параметра выше 75%).

Однако для некоторых приложений CMS, которые имеют очень большие выбросы скоростей выделения памяти, одной из стратегий может быть увеличение запаса (уменьшение значения параметра) при отключении адаптивного изменения размера. Цель в этом случае заключается в попытке сократить сбои параллельного режима ценой более частого выполнения параллельных сборок мусора CMS.

Сбои параллельного режима из-за фрагментации

Давайте рассмотрим другой случай, когда данные, необходимые для проведения анализа настройки, доступны только в журналах сборки мусора. В этом случае мы хотим использовать *статистику свободных списков* (free list statistics) для прогнозирования, когда JVM может страдать от сбоя параллельного режима, вызванного фрагментацией кучи. Данный тип сбоя вызывается свободным списком, поддерживаемым CMS, с которым мы встречались в разделе "Точки безопасности JVM" главы 7, "Вглубь сборки мусора".

Чтобы увидеть дополнительный вывод, нам нужен другой переключатель JVM:

```
-XX:PrintFLSStatistics=1
```

Вывод, который производится в журнале сборки мусора, выглядит следующим образом (здесь показан результат теста производительности `BinaryTreeDictionary`):

```
Total Free Space: 40115394
Max Chunk Size: 38808526
Number of Blocks: 1360
Av. Block Size: 29496
Tree Height: 22
```

В этом случае мы можем получить представление о распределении размеров блоков памяти — зная средний и максимальный размеры блоков. Если у нас заканчиваются фрагменты, достаточно большие для того, чтобы поддерживать перенос больших живых объектов в область бессрочных, повышение сборки мусора деградирует до рассматриваемого сбоя параллельного режима.

Чтобы уплотнить кучу и объединить список свободных пространств, JVM возвращается к Parallel GC, что, вероятно, приведет к длительной приостановке приложения. Этот анализ полезен при выполнении в реальном времени, поскольку он может сигнализировать о неизбежности длительной паузы. Вы можете наблюдать за ним, анализируя журналы или используя инструмент Censum, который может автоматически определять приближающийся сбой параллельного режима.

Настройка G1

Глобальная цель настройки G1 заключается в том, чтобы позволить конечному пользователю просто установить максимальный размер кучи и параметр `MaxGCPauseMillis`, а сборщик сам позаботится обо всем остальном. Тем не менее текущая реальность все еще далека от описанного.

Подобно CMS, G1 поставляется с большим количеством настроек конфигурации, причем некоторые из них все еще экспериментальны и не полностью видимы в виртуальной машине (в смысле видимых настраиваемых метрик). Это затрудняет понимание влияния любых изменений настройки. Если такие параметры необходимы для настройки (а в настоящее время они предназначены для некоторых сценариев настройки), то должен быть указан следующий флаг:

```
-XX:+UnlockExperimentalVMOptions
```

В частности, этот флаг должен быть указан, если используется флаг `-XX:G1NewSizePercent=<n>` или `-X:G1MaxNewSizePercent=<n>`. В некоторый момент в будущем некоторые из этих флагов могут стать официальным мейнстримом и больше не требовать включения флага экспериментальных возможностей, но пока что он необходим.

На рис. 8.7 можно увидеть интересное представление кучи G1. Это изображение сгенерировано приложением JavaFX `regions`.

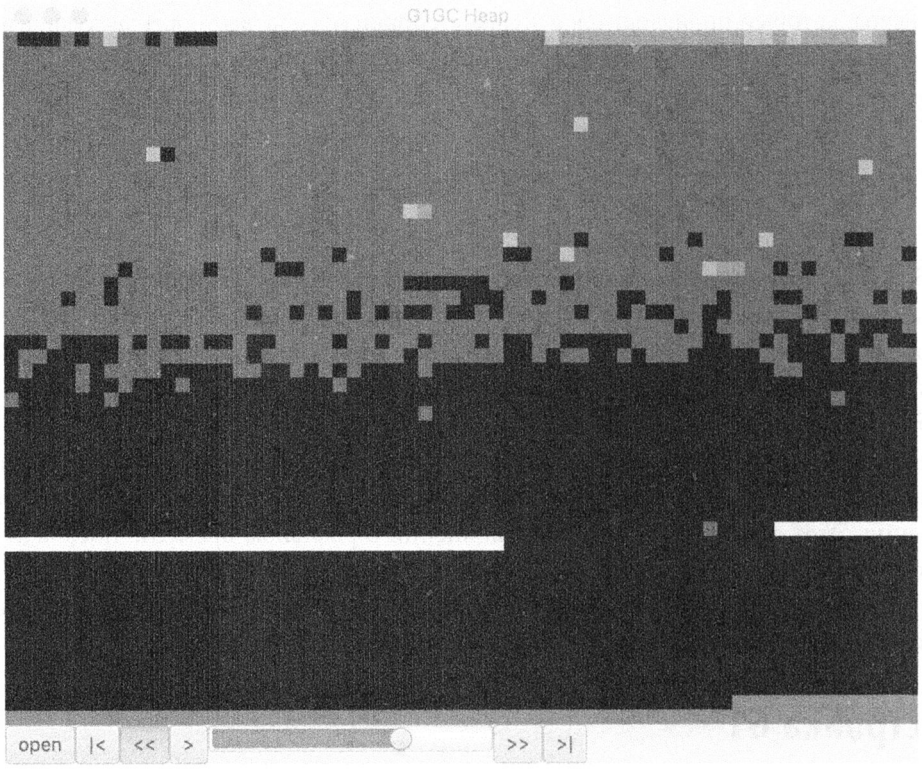

Рис. 8.7. Визуализация кучи G1 по регионам

Это небольшое приложение с открытым исходным кодом, которое анализирует журнал сборки мусора G1 и предоставляет возможность визуализировать схему регионов кучи G1 в течение всего времени жизни журнала сборки мусора. Инструмент был написан Кирком Пеппердайном (Kirk Pepperdine) и может быть загружен с GitHub[4]. На момент написания книги он все еще находится в активно развивающемся состоянии.

Одной из основных проблем настройки G1 является то, что его внутреннее устройство значительно изменилось со времен первых дней жизни этого сборщика мусора. Это ведет к существенной проблеме с антипаттерном "Настройка по совету", поскольку многие старые статьи о G1 сегодня могут быть просто неверны.

Поскольку G1 начиная с Java 9 становится сборщиком по умолчанию, инженерам по производительности, безусловно, придется обратиться к теме о настройке G1, но пока что это достаточно сложная задача.

[4] `https://github.com/kcpeppe/regions`

Однако давайте закончим этот раздел, сосредоточив внимание на том, где был достигнут определенный прогресс и где G1 обещает превзойти CMS. Напомним, что CMS не является уплотняющим сборщиком мусора, поэтому со временем куча может фрагментироваться. Это в конечном итоге приведет к сбою параллельного режима, и JVM потребуется выполнить полную параллельную сборку (возможно, со значительной паузой приостановки приложения).

В случае G1 при условии, что сборщик не отстает от скорости выделения памяти, инкрементное уплотнение дает возможность избежать сбоев параллельного режима. Приложение, которое имеет высокий, стабильный уровень выделения памяти и которое создает в основном краткосрочные объекты, должно:

- установить большой размер молодого поколения;
- увеличить порог для перемещения объектов в область бессрочных, вероятно, до максимального значения (15);
- установить максимальное целевое время паузы, допустимое для данного приложения.

Такое конфигурирование Эдема и пространства выживших объектов дает наилучшие шансы на то, что действительно недолговечные объекты не будут перемещаться из пространства молодого поколения. Это снижает нагрузку на старое поколение и уменьшает необходимость очистки старых регионов. В настройках сборки мусора редко можно что-либо гарантировать, но это пример рабочей нагрузки, где G1 может значительно превзойти CMS, хотя и ценой некоторых усилий по настройке кучи.

jHiccup

Мы уже встречались с HdrHistogram в разделе "Статистика, отличная от нормальной" главы 5, "Микротесты и статистика". Связанным с ним инструментом является jHiccup, который может быть загружен с GitHub[5]. Это инструмент, который предназначен для отображения "икоты", когда JVM не может работать непрерывно. Одной из распространенных причин такого поведения является приостановка приложения сборщиком мусора, но есть и другие эффекты, связанные с операционной системой и платформой, которые также могут вызывать "икоту". Поэтому он полезен не только для настройки сборки мусора, но и для работы с ультранизкими задержками.

Фактически мы также уже видели пример того, как работает jHiccup. В разделе "Shenandoah" главы 7, "Вглубь сборки мусора", мы познакомились со сборщиком мусора Shenandoah и продемонстрировали график производительности этого сборщика

[5] `https://github.com/giltene/jHiccup`

по сравнению с другими. График сравнительной производительности на рис. 7.9 был подготовлен с помощью jHiccup.

jHiccup — отличный инструмент, который можно использовать при настройке HotSpot, хотя автор Джил Тен (Gil Tene) охотно признает, что он был написан для выявления недостатков HotSpot по сравнению с Zul JVM от Azul.

jHiccup обычно используется в качестве Java-агента с помощью параметра командной строки Java `-javaagent:jHiccup.jar`. Его также можно использовать посредством API Attach (как и некоторые другие инструменты командной строки). Соответствующая команда имеет следующий вид:

```
jHiccup -p <pid>
```

Здесь происходит внедрение jHiccup в запущенное приложение.

jHiccup генерирует вывод в виде журнала гистограмм, который может быть принят HdrHistogram. Давайте посмотрим на все это в действии, начиная с напоминания о модельном приложении выделения памяти, представленном в разделе "Роль выделения памяти" главы 6, "Сборка мусора".

Для запуска с помощью набора флагов протоколирования сборки мусора и работы jHiccup в качестве агента рассмотрим небольшой фрагмент сценария оболочки, настраивающий запуск:

```
#!/bin/bash

# Простой сценарий для запуска jHiccup с модельным приложением

CP=./target/optimizing-java-1.0.0-SNAPSHOT.jar

JHICCUP_OPTS=
    -javaagent:~/.m2/repository/org/jhiccup/jHiccup/2.0.7/jHiccup-2.0.7.jar

GC_LOG_OPTS="-Xloggc:gc-jHiccup.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps
    -XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution"

MEM_OPTS="-Xmx1G"

JAVA_BIN=`which java`

if [ $JAVA_HOME ]; then
    JAVA_CMD=$JAVA_HOME/bin/java
elif [ $JAVA_BIN ]; then
    JAVA_CMD=$JAVA_BIN
else
    echo "Для запуска этой команды должно быть установлено значение"
    echo " переменной среды $JAVA_HOME либо Java должен находиться"
```

```
    echo " в каталоге, содержащемся в пути $PATH."
    exit 1
fi

exec $JAVA_CMD -cp $CP $JHICCUP_OPTS $GC_LOG_OPTS $MEM_OPTS
    optjava.ModelAllocator
```

Этот сценарий позволяет создавать журнал сборки мусора и файл `.hlog`, который можно передавать инструменту `jHiccupLog Processor`, поставляемому как часть jHiccup. Простое, ненастроенное представление данных jHiccup показано на рис. 8.8.

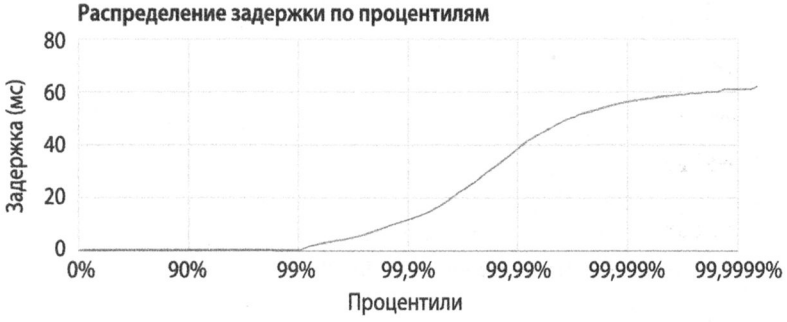

Рис. 8.8. Представление jHiccup для программы ModelAllocator

Оно получено с помощью очень простого вызова jHiccup:

```
jHiccupLogProcessor -i hiccup-example2.hlog -o alloc-example2
```

Имеются некоторые другие полезные переключатели. Чтобы увидеть все доступные флаги, воспользуйтесь командой

```
jHiccupLogProcessor -h
```

Очень часто инженерам по производительности требуется несколько разных представлений о поведении одного и того же приложения, поэтому для полноты давайте посмотрим, как одно и то же приложение выглядит в Censum (рис. 8.9).

Этот график размера кучи после сборки мусора показывает, что HotSpot пытается изменить размер кучи и не может найти стабильное состояние. Это часто возникающая ситуация даже для таких простых приложений, как ModelAllocator. JVM — очень динамичная среда, и по большей части разработчикам следует избегать чрезмерной заботы о низкоуровневых деталях эргономики сборки мусора.

Наконец, некоторые очень интересные технические подробности об HdrHistogram и jHiccup можно найти в блоге Ницана Вакарта (Nitsan Wakart)[6].

[6] `https://github.com/kcpeppe/regions`

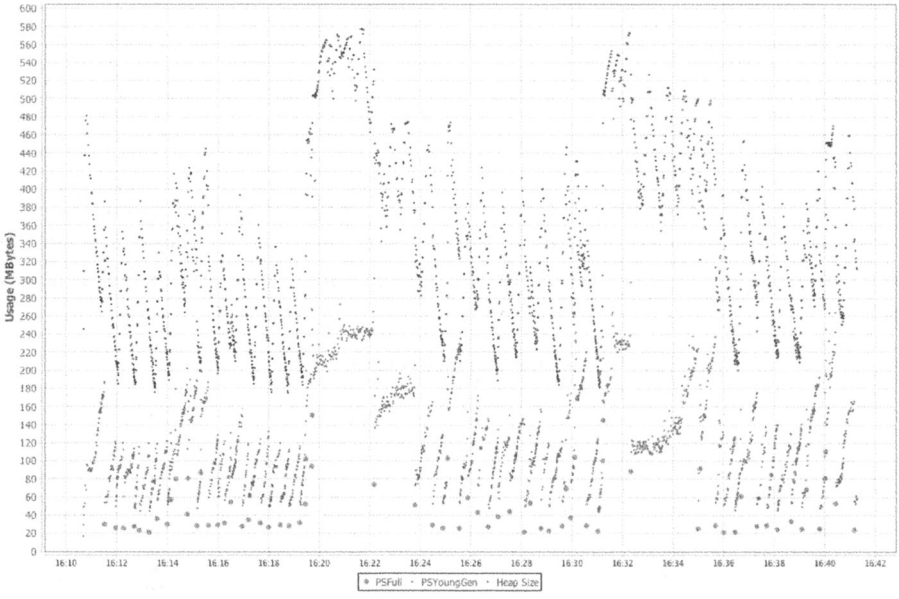

Рис. 8.9. Представление ModelAllocator в Censum

Резюме

В этой главе мы только слегка коснулись необъятной темы искусства настройки сборки мусора. Методы, которые мы продемонстрировали, в основном специфичны для отдельных сборщиков, но есть и ряд базовых методик, применимых в общем случае. Мы также рассмотрели некоторые основные принципы работы с журналами сборки мусора и познакомились с некоторыми полезными инструментами.

Двигаясь дальше, в следующей главе мы перейдем к следующей из основных подсистем JVM: к выполнению кода приложения. Мы начнем с обзора интерпретатора, а затем рассмотрим JIT-компиляцию, в том числе то, как она соотносится со стандартной (или предварительной) компиляцией.

ГЛАВА 9

Выполнение кода в JMV

Две основные службы, предоставляемые любой JVM, — это управление памятью и простой в использовании контейнер для выполнения кода приложений. С определенной степенью глубины мы рассмотрели сборку мусора в главах 6–8, и в этой главе переходим к новой теме — выполнению кода.

Напомним, что в спецификации виртуальной машины Java, которую обычно называют VMSpec, описывается, каким образом реализация Java, соответствующая спецификации, должна выполнять код.

Спецификация VMSpec определяет выполнение байт-кода Java в терминах интерпретации. Однако, вообще говоря, интерпретируемые среды имеют плохую производительность по сравнению с программными средами, непосредственно выполняющими машинный код. Большинство современных промышленных сред Java решают эту проблему, предоставляя возможность динамической компиляции.

Как мы обсуждали в главе 2, "Обзор JVM", эта способность известна также как *компиляция Just-in-Time*, или просто *JIT-компиляция*. Она представляет собой механизм, с помощью которого JVM отслеживает, какие методы выполняются, с тем чтобы определить, следует ли скомпилировать отдельные методы непосредственно в исполняемый код.

В этой главе мы начнем с краткого обзора интерпретации байт-кода и выяснения, чем HotSpot отличается от других интерпретаторов, с которыми вы можете быть знакомы. Затем мы рассмотрим основные концепции оптимизации на основе профилирования. Мы обсудим кеширование кода, а затем познакомимся с основами подсистемы компиляции HotSpot.

В следующей главе мы объясним механизмы, лежащие в основе некоторых распространенных оптимизаций HotSpot, и как они используются для получения очень быстрых скомпилированных методов, до какой степени они могут быть настроены, а также их ограничения.

Обзор интерпретации байт-кода

Как мы вкратце узнали из раздела "Интерпретация и загрузка классов" главы 2, "Обзор JVM", интерпретатор JVM работает как стековая машина. Это означает, что, в отличие от физических процессоров, в ней нет регистров, которые используются как области хранения данных для вычислений. Вместо этого все значения, с которыми выполняется работа, размещаются в *стеке вычислений* (evaluation stack), и команды стековой машины работают путем преобразования значений на вершине стека.

JVM предоставляет три основные области для хранения данных.

- *Стек вычислений* (evaluation stack), локальный для конкретного метода.
- *Локальные переменные* (local variables) для временного хранения результатов (также локальные для методов).
- *Куча объектов* (object heap), совместно используемая методами и потоками.

На рис. 9.1–9.5 можно увидеть последовательность операций виртуальной машины, которая использует стек для выполнения вычислений (их можно рассматривать как разновидность псевдокода, который должен быть понятен программистам на языке Java).

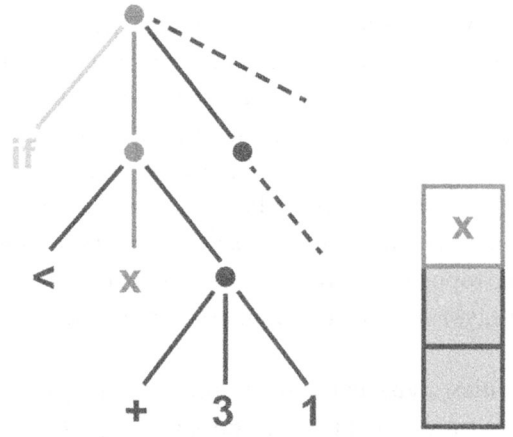

Рис. 9.1. Начальное состояние интерпретатора

Сейчас интерпретатор должен вычислить правое поддерево, чтобы определить значение для сравнения с содержимым переменной x.

Первое значение следующего поддерева, которое представляет собой константу 3, загружается в стек.

Далее в стек вносится значение 1. В реальной JVM эти значения загружаются из области констант файла класса.

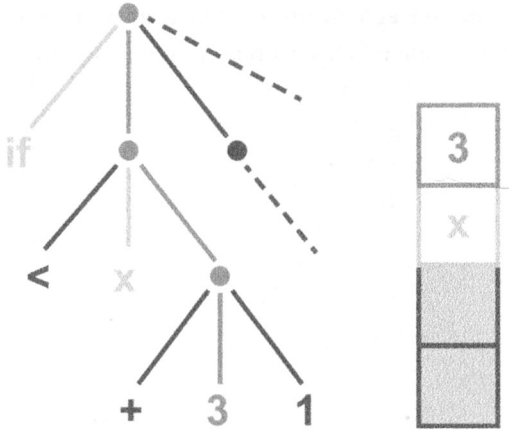

Рис. 9.2. Вычисление поддерева

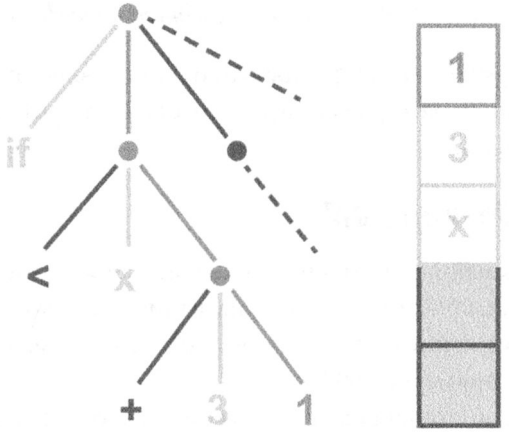

Рис. 9.3. Вычисление поддерева

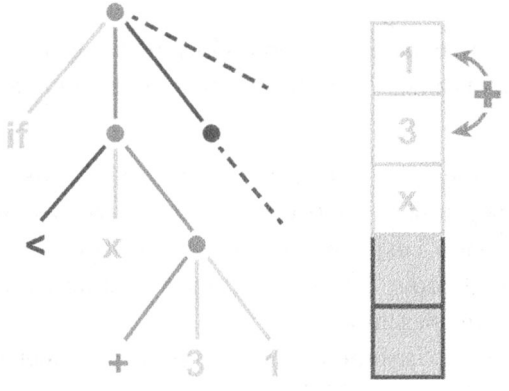

Рис. 9.4. Вычисление поддерева

В этот момент над двумя верхними элементами в стеке выполняется операция суммирования, которая удаляет их из стека и замещает их результатом сложения этих двух чисел.

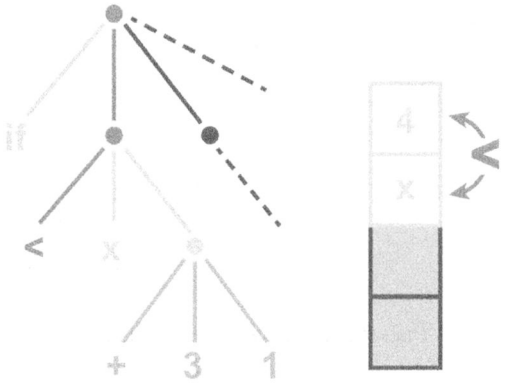

Рис. 9.5. Вычисление последнего поддерева

Теперь результирующее значение доступно для сравнения со значением, содержащимся в переменной x, которое оставалось в стеке на протяжении всего процесса вычислений поддеревьев.

Введение в байт-код JVM

В случае JVM каждый код операции стековой машины (операционный код, опкод) представлен одним байтом (откуда и название "байт-код" (*bytecode*)). Соответственно, опкоды могут принимать значения от 0 до 255, из которых по состоянию на Java 10 используются примерно 200.

Команды байт-кодов типизированы в том смысле, что `iadd` и `dadd` ожидают правильные примитивные типы (два значения `int` и два `double` соответственно) в двух верхних позициях стека.

Многие команды байт-кодов объединены в "семейства", по одной команде для каждого примитивного типа и одной для ссылки на объект.

Например, в семействе `store` каждая команда имеет конкретный смысл: `dstore` означает "сохранить верхний элемент стека в локальную переменную типа `double`", тогда как `astore` означает "сохранить верхний элемент стека в локальную переменную ссылочного типа". В обоих случаях тип локальной переменной должен соответствовать типу входящего значения.

Поскольку язык программирования Java разрабатывался с учетом высокой переносимости, спецификация JVM разработана таким образом, чтобы иметь

возможность запускать один и тот же байт-код без модификации на архитектурах как с *прямым*, так и с *обратным порядком байтов* (little-endian и big-endian). В результате байт-код JVM должен принять решение о том, какому соглашению о порядке байтов следовать (с учетом того, что аппаратное обеспечение с противоположными порядками байтов должно по-разному обрабатываться программным обеспечением).

Байткод использует обратный порядок байтов, так что старший байт любой многобайтной последовательности идет первым.

Некоторые семейства опкодов, такие как `load`, имеют *сокращенные формы* (shortcut forms). Это позволяет опустить аргумент и, таким образом, сэкономить байты аргументов в файле класса. В частности, `aload_0` помещает текущий объект (т.е. `this`) на вершину стека. Поскольку это очень распространенная операция, такое решение приводит к значительной экономии размера файла класса.

Однако поскольку классы Java, как правило, довольно компактны, это проектное решение, вероятно, имело более важное значение в первые дни платформы, когда файлы классов — зачастую аплеты — должны были загружаться через модем на скорости 14,4 Кбит/с.

Начиная с Java 1.0 был добавлен только один новый байт-код (`invoke dynamic`), а два (`jsr` и `ret`) объявлены устаревшими.

Использование сокращенной формы и команд для конкретных типов сильно увеличивает количество необходимых кодов операций, поскольку некоторые из них используются для представления концептуально одной и той же операции. Количество назначенных опкодов, таким образом, намного больше, чем количество представляемых байт-кодами базовых операций, и концептуально на самом деле байт-код очень прост.

Давайте познакомимся с некоторыми основными категориями байт-кодов, упорядоченными в соответствии с их семействами. Обратите внимание, что в следующих таблицах `c1` указывает 2-байтовый индекс пула констант, а `i1` указывает локальную переменную в текущем методе. Скобки указывают, что у семейства есть некоторые сокращенные опкоды.

Первая категория, с которой мы встретимся, — это категория *загрузки и сохранения*, приведенная в табл. 9.1. Эта категория содержит коды операций, которые перемещают данные в стек и снимают их оттуда, например, путем загрузки из пула констант или путем сохранения вершины стека в поле объекта в куче.

Таблица 9.1. Операции загрузки и сохранения

Имя семейства	Аргумент	Назначение
`load`	`(il)`	Загружает значение из локальной переменной `il` в стек
`store`	`(il)`	Сохраняет значение на вершине стека в локальную переменную `il`
`ldc`	`cl`	Загружает значение из `CP#cl` в стек
`const`		Загружает простое константное значение в стек
`pop`		Снимает значение с вершины стека
`dup`		Дублирует значение на вершине стека
`getfield`	`cl`	Загружает в стек значение из поля, указанного `CP#cl`, из объекта на вершине стека
`putfield`	`cl`	Сохраняет значение на вершине стека в поле, указанное `CP#cl`
`getstatic`	`cl`	Загружает в стек значение из статического поля, указанного `CP#cl`
`putstatic`	`cl`	Сохраняет значение на вершине стека в статическое поле, указанное `CP#cl`

Разницу между `ldc` и `const` следует пояснить. Байт-код `ldc` загружает константу из пула констант текущего класса. Он содержит строки, примитивные константы, литералы классов и другие (внутренние) константы, необходимые для работы программы[1].

С другой стороны, опкоды `const` не принимают никаких параметров и занимаются загрузкой конечного количества истинных констант, как, например, `aconst_null`, `dconst_0` и `iconst_m1` (последний из которых загружает `-1` как `int`).

Следующая категория — *арифметические байт-коды*, которые применяются только к примитивным типам, и ни один из них не принимает аргументы, поскольку они представляют собой чисто стековые операции. Эта простая категория показана в табл. 9.2.

Таблица 9.2. Арифметические операции

Имя семейства	Назначение
`add`	Суммирует два значения на вершине стека
`sub`	Вычитает два значения на вершине стека
`div`	Делит два значения на вершине стека
`mul`	Умножает два значения на вершине стека
`(cast)`	Преобразует значение на вершине стека в значение другого примитивного типа
`neg`	Меняет знак значения на вершине стека
`rem`	Вычисляет остаток целочисленного деления двух значений на вершине стека

[1] Последние версии JVM позволяют работать также с более экзотическими константами для поддержки современных технологий виртуализации.

В табл. 9.3 показана категория *управления потоком выполнения*. Эта категория содержит представления на уровне байт-кодов конструкций циклов и ветвлений уровня исходного языка. Например, в опкоды управления потоком выполнения трансформируются после выполнения компиляции такие конструкции Java, как `for`, `if`, `while` и `switch`.

Таблица 9.3. Команды управления потоком выполнения

Имя семейства	Аргумент	Назначение
`if`	`(il)`	Ветвление к точке, указанной в качестве аргумента, если условие имеет значение `true`
`goto`	`il`	Безусловный переход с переданным в качестве аргумента смещением
`tableswitch`		Выходит за рамки книги
`lookupswitch`		Выходит за рамки книги

Детальное описание работы операций `tableswitch` и `lookupswitch` выходит за рамки данной книги.

Категория управления потоком выполнения кажется очень маленькой, но истинное количество операций управления потоком выполнения на удивление велико. Это связано с тем, что существует большое количество членов семейства опкодов `if`. Мы встретили код операции `if_icmpge` (if-integer-compare-more-or-equal, если целое сравнение больше или равно) в примере `javap` в главе 2, "Обзор JVM", но есть и множество других, которые представляют разные варианты конструкции `if` в Java.

Устаревшие байт-коды `jsr` и `ret`, которые больше не выводятся `javac` начиная с Java 6, также являются частью этого семейства. Они не являются корректными опкодами для современных версий платформы и поэтому не включены в эту таблицу.

Одна из наиболее важных категорий кодов операций показана в табл. 9.4. Это категория *вызова метода*, являющаяся единственным механизмом, с помощью которого программа Java позволяет передать управление новому методу. То есть в платформе полностью разделяются локальное управление потоком выполнения и передача управления другому методу.

Таблица 9.4. Операции вызова метода

Имя опкода	Аргумент	Назначение
`invokevirtual`	`c1`	Вызывает метод, найденный в `CP#c1`, с помощью виртуальной диспетчеризации
`invokespecial`	`c1`	Вызывает метод, найденный в `CP#c1`, с помощью "специальной" (т.е. точной) диспетчеризации
`invokeinterface`	`c1, count, 0`	Вызывает метод интерфейса, найденный в `CP#c1`, с помощью поиска смещения интерфейса

Имя опкода	Аргумент	Назначение
`invokestatic`	`c1`	Вызывает статический метод, найденный в `CP#c1`
`invokedynamic`	`c1, 0, 0`	Динамический поиск вызываемого метода и его вызов

Дизайн JVM и использование явных опкодов *вызова метода* означает отсутствие эквивалента операции `call`, имеющейся в машинном коде.

Вместо этого байт-код JVM использует определенную специальную терминологию; мы говорим о *месте вызова* (call site), которое представляет собой место внутри вызывающего метода, где вызывается другой метод. Это не все — в случае вызова нестатического метода всегда есть какой-то объект, который мы разрешаем для вызова метода. Этот объект известен как *объект-получатель* (receiver object), а его тип времени выполнения называется *типом получателя* (receiver type).

Вызов статического метода всегда преобразуется в опкод `invokestatic` и не имеет объекта-получателя.

Программисты на языке Java, которые являются новичками в виртуальных машинах, могут быть удивлены, узнав, что вызовы метода для Java-объектов на самом деле преобразуются в один из трех возможных байткодов (`invokevirtual`, `invokespecial` или `invokeinterface`) в зависимости от контекста вызова.

Может быть весьма полезно написать некоторый код Java и посмотреть, при каких обстоятельствах реализуется та или иная возможность (дизассемблируя простой класс Java с помощью `javap`).

Вызов метода экземпляра обычно преобразуется в команду `invokevirtual`, за исключением случаев, когда статический тип объекта-получателя известен только как тип интерфейса. В этом случае вызов будет представлен кодом операции `invokeinterface`. Наконец, в некоторых случаях (например, для закрытых методов или вызовов суперкласса), когда точный метод известен во время компиляции, создается команда `invokespecial`.

Это приводит к вопросу о том, каково место `invokedynamic` в этой картине. Краткий ответ заключается в том, что прямой поддержки языкового уровня для `invokedynamic` в Java нет, даже в версии 10.

Фактически, когда опкод `invokedynamic` был добавлен в среду выполнения в Java 7, не было никакого способа заставить `javac` генерировать новый байт-код. В этой старой версии Java `invokedynamic` был добавлен только для поддержки долгосрочных экспериментов и динамических языков, отличных от Java (в частности, JRuby).

Однако начиная с Java 8 invokedynamic стал важной частью языка Java и используется для поддержки расширенных языковых возможностей. Давайте рассмотрим простой пример из лямбда-выражений Java 8:

```
public class LambdaExample
{
    private static final String HELLO = "Hello";
    public static void main(String[] args) throws Exception
    {
        Runnable r = () -> System.out.println(HELLO);
        Thread t = new Thread(r);
        t.start();
        t.join();
    }
}
```

Это тривиальное применение лямбда-выражения генерирует следующий байт-код:

```
public static void main(java.lang.String[]) throws
java.lang.Exception;
  Code:
     0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable;
     5: astore_1
     6: new #3              // class java/lang/Thread
     9: dup
    10: aload_1
    11: invokespecial #4    // Метод java/lang/Thread.
                            // "<init>":(Ljava/lang/Runnable;)V
    14: astore_2
    15: aload_2
    16: invokevirtual #5    // Метод java/lang/Thread.start:()V
    19: aload_2
    20: invokevirtual #6    // Метод java/lang/Thread.join:()V
    23: return
```

Даже если бы мы ничего не знали о команде invokedynamic, ее форма указывает, что вызывается какой-то метод и возвращаемое значение этого вызова помещается в стек.

Копаясь в байт-коде дальше, мы обнаруживаем, что это значение является ссылкой на объект (и это неудивительно), соответствующей лямбда-выражению. Она создается фабричным методом платформы, который вызывается с помощью команды invokedynamic. Этот вызов ссылается на расширенные записи в пуле констант класса, поддерживающие динамический характер выполнения вызова.

Это, пожалуй, самый очевидный вариант использования invokedynamic для Java-программистов, но не единственный. Этот код операции широко используется в JVM языками программирования, отличными от Java, такими как JRuby и Nashorn

(JavaScript), а также все чаще — каркасами Java. Однако по большей части он остается чем-то вроде курьеза, хотя инженер в области производительности должен о нем знать. В дальнейшем мы рассмотрим некоторые связанные аспекты `invokedynamic`.

Последняя категория кодов операций, которую мы рассмотрим, — это *коды операций платформы*. Они показаны в табл. 9.5 и включают такие операции, как выделение новой памяти кучи и управление внутренними блокировками (мониторами, используемыми синхронизацией) отдельных объектов.

Таблица 9.5. Опкоды операций платформы

Имя опкода	Аргумент	Назначение
`new`	`cl`	Выделяет память для объекта типа, найденного в `CP#cl`
`newarray`	`prim`	Выделяет память для примитивного массива типа `prim`
`anewarray`	`cl`	Выделяет память для массива объектов типа, найденного в `CP#cl`
`arraylength`		Заменяет массив на вершине стека его длиной
`monitorenter`		Блокирует монитор объекта на вершине стека
`monitorexit`		Разблокирует монитор объекта на вершине стека

Для `newarray` и `anewarray` длина выделяемого массива при выполнении опкода должна находиться на вершине стека.

В каталоге байт-кодов имеется четкое различие между "крупно-" и "мелкозернистыми" байткодами с точки зрения сложности осуществления каждой операции.

Например, арифметические операции очень "мелкозернистые" и реализуются в HotSpot на чистом ассемблере. Напротив, грубые операции (например, операции, требующие поиска в пуле констант, в частности диспетчеризация метода) будут требовать обратного вызова HotSpot VM.

Наряду с семантикой отдельных байт-кодов мы должны также сказать пару слов о точках безопасности в интерпретированном коде. В главе 7, "Вглубь сборки мусора", мы познакомились с концепцией точек безопасности JVM как точек, в которых JVM необходимо выполнить некоторые обслуживающие действия, для чего требуется согласованное внутреннее состояние. Сюда включается граф объектов (который, конечно же, изменяется текущими потоками приложений самым общим образом).

Чтобы достичь этого непротиворечивого, согласованного состояния, все потоки приложений должны быть остановлены (тем самым предотвращается изменение ими совместно используемой кучи на протяжении всей обслуживающей активности JVM). Как это делается?

Решение состоит в том, чтобы вспомнить, что каждый поток приложений JVM является истинным потоком операционной системы[2]. Это не все — когда должен быть диспетчеризован код операции, поток приложения, выполняющий интерпретируемый метод, выполняет код JVM-интерпретатора, а не код пользователя. Поэтому

[2] Как минимум в основных, наиболее распространенных JVM.

куча должна находиться в согласованном состоянии, а поток приложения может быть остановлен.

Следовательно, "между байткодами" — идеальный момент, чтобы остановить поток приложения, и один из простейших примеров точки безопасности.

Ситуация для JIT-скомпилированных методов более сложна, но, по сути, в сгенерированный JIT-компилятором машинный код должны быть вставлены эквивалентные барьеры.

Простые интерпретаторы

Как уже упоминалось в главе 2, "Обзор JVM", простейший интерпретатор можно рассматривать как конструкцию `switch` внутри цикла. Подобный пример можно найти в проекте Ocelot[3], который представляет собой частичную реализацию JVM-интерпретатора, предназначенную для обучения. Для тех, кто не знаком с реализацией интерпретаторов, версия 0.1.1 является хорошей отправной точкой.

Метод `execMethod()` интерпретатора выполняет интерпретацию единственного метода байт-кода. Реализовано достаточное количество опкодов (некоторые из них — с фиктивной реализацией), чтобы можно было выполнять целочисленные математические вычисления и выводить "Hello World".

Полная реализация, способная выполнять хотя бы очень простые программы, требует для корректной работы выполнения сложных операций, таких как просмотр пула констант. Однако даже скелетного наброска достаточно, чтобы стала очевидной базовая структура интерпретатора:

```
public EvalValue execMethod(final byte[] instr)
{
    if (instr == null || instr.length == 0)
        return null;

    EvaluationStack eval = new EvaluationStack();
    int current = 0;
LOOP:

    while (true)
    {
        byte b = instr[current++];
        Opcode op = table[b & 0xff];

        if (op == null)
        {
            System.err.println("Нераспознанный опкод: " + (b & 0xff));
            System.exit(1);
        }
```

[3] https://github.com/kittylyst/ocelotvm

```
byte num = op.numParams();

switch (op)
{
    case IADD:
        eval.iadd();
        break;

    case ICONST_0:
        eval.iconst(0);
        break;

    // ...
    case IRETURN:
        return eval.pop();

    case ISTORE:
        istore(instr[current++]);
        break;

    case ISUB:
        eval.isub();
        break;

    // Фиктивная реализация
    case ALOAD:
    case ALOAD_0:
    case ASTORE:
    case GETSTATIC:
    case INVOKEVIRTUAL:
    case LDC:
        System.out.print("Выполнение " + op +
                         " с параметрами: ");

        for (int i = current; i < current + num; i++)
        {
            System.out.print(instr[i] + " ");
        }

        current += num;
        System.out.println();
        break;

    case RETURN:
        return null;

    default:
```

```
                System.err.println("Встретили " + op + " : этого не "
                                   "может быть. Аварийный выход.");
                System.exit(1);
        }
    }
}
```

Байт-коды считываются по одному за раз и диспетчеризуются на основе их кодов. В случае кодов операций с параметрами они также считываются из потока, чтобы гарантировать, что позиция чтения остается корректной.

Временные значения вычисляются в `EvaluationStack`, которая является локальной переменной `execMethod()`. Для выполнения целочисленных математических операций арифметические коды операций работают с этим стеком.

Вызов метода в простейшей версии Ocelot не реализован, но если бы он был реализован, то он выполнял бы поиск метода в пуле констант, находил байт-код, соответствующий вызываемому методу, а затем рекурсивно вызывал `execMethod()`. Версия 0.2 кода демонстрирует этот способ для вызова статических методов.

Детали, специфичные для HotSpot

HotSpot — это JVM промышленного качества, которая не только полностью реализована, но и имеет расширенные возможности, предназначенные для быстрого выполнения кода даже в интерпретируемом режиме. В отличие от простого стиля, с которым мы встретились в учебном примере Ocelot, HotSpot — это *шаблонный интерпретатор*, который динамически создает интерпретатор при каждом запуске.

Этот подход значительно сложнее для понимания и делает вызовом для новичков даже простое чтение исходного кода интерпретатора. HotSpot использует относительно большое количество ассемблера для реализации простых (таких, как арифметические) операций VM и использует схему стека базовой (нативной) платформы для дальнейшего повышения производительности.

HotSpot также определяет и использует специфичные для JVM (так называемые закрытые) байт-коды, которых нет в VMSpec. Они используются для того, чтобы позволить HotSpot отличать распространенные горячие случаи использования от более общих случаев использования конкретного опкода.

Цель такого дизайна — помочь HotSpot справиться с удивительным количеством краевых случаев. Например, метод, помеченный как `final`, нельзя перекрывать, поэтому разработчик может решить, что при вызове такого метода `javac` будет генерировать опкод `invokepecial`. Однако спецификация Java Language Specification 13.4.17 говорит об этом случае следующее:

> *Изменение метода, объявленного как окончательный (`final`), как более не являющегося таковым, не нарушает совместимость с ранее существовавшими бинарными файлами.*

Рассмотрим следующий фрагмент кода Java:

```
public class A
{
    public final void fMethod()
    {
        // ... некоторые действия
    }
}
public class CallA
{
    public void otherMethod(A obj)
    {
        obj.fMethod();
    }
}
```

Теперь предположим, что `javac` скомпилировал вызовы окончательных методов в опкоды `invokespecial`. Байт-код для `CallA::otherMethod` будет выглядеть примерно так:

```
public void otherMethod()
  Code:
    0: aload_1
    1: invokespecial #4 // Метод A.fMethod:()V
    4: return
```

Теперь предположим, что код для `A` изменяется так, что `fMethod()` становится неокончательным, и может быть перекрыт в подклассе; будем называть этот подкласс `B`. Теперь предположим, что экземпляр `B` передается в `otherMethod()`. Из байт-кода будет вызываться команда `invokespecial`, и в результате будет вызываться неправильная реализация метода.

Это является нарушением правил объектной ориентированности языка программирования Java. Строго говоря, это нарушает *принцип подстановки* Лисков (названный по имени Барбары Лисков (Barbara Liskov), одного из пионеров объектно-ориентированного программирования), который, попросту говоря, утверждает, что экземпляр подкласса можно использовать где угодно, где ожидается экземпляр суперкласса. Этот принцип представлен буквой *L* в знаменитых принципах разработки программного обеспечения *SOLID*.

По этой причине вызовы окончательных методов должны быть скомпилированы в команды `invokevirtual`. Однако, поскольку JVM знает, что такие методы не могут быть перекрыты, интерпретатор HotSpot имеет закрытый байт-код, который используется исключительно для диспетчеризации методов, объявленных как `final`.

Вот еще один пример: спецификация языка гласит, что объект, подлежащий финализации (читайте обсуждение механизма финализации в разделе "Избегайте

финализации" главы 11, "Языковые методы повышения производительности"), должен быть зарегистрирован в подсистеме финализации. Эта регистрация должна выполняться сразу же по завершении вызова конструктора `Object::<init>`. В случае JVMTI и других потенциальных перезаписей байт-кода местоположение указанного кода может стать неясным. Чтобы обеспечить строгое соответствие спецификации, HotSpot использует частный байт-код, которым помечает возврат из "истинного" конструктора `Object`.

Список кодов операций можно найти в файле `hotspot/src/share/vm/interpreter/bytecodes.cpp`; специфичные для HotSpot случаи перечислены в нем как "байт-коды JVM".

AOT- и JIT-компиляция

В этом разделе мы обсудим и сравним раннюю (Ahead-of-Time — AOT) компиляцию и компиляцию оперативную (Just-in-Time — JIT) в качестве альтернативных подходов к созданию исполняемого кода.

JIT-компиляция была разработана недавно, позже, чем AOT-компиляция, но ни один из этих подходов не стоял на месте более 20 лет существования Java и каждый из них заимствовал успешные технологии у другого.

Ранняя компиляция

Если у вас есть опыт программирования на таких языках, как C или C++, то вы знакомы с AOT-компиляцией (возможно, вы всегда называли ее просто "компиляцией"). Это процесс, при котором внешняя программа (компилятор) принимает исходный текст (в удобочитаемом для человека виде) и на выходе дает непосредственно исполняемый машинный код.

Ранняя компиляция исходного кода означает, что у вас есть только одна возможность воспользоваться преимуществами любых потенциальных оптимизаций.

Скорее всего, вы захотите создать исполняемый файл, предназначенный для конкретной платформы и архитектуры процессора, на которой вы собираетесь его запускать. Такие тщательно настроенные бинарные файлы смогут использовать любые преимущества процессора, которые могут ускорить работу программы.

Однако в большинстве случаев исполняемый файл создается без знания конкретной платформы, на которой он будет выполняться. Это означает, что AOT-компиляция должна делать консервативное предположение о том, какие возможности процессора могут быть доступны. Если код скомпилирован в предположении доступности некоторых возможностей, а затем все оказывается не так, этот бинарный файл не будет запускаться совсем.

Это приводит к ситуации, когда AOT-скомпилированные бинарные файлы не в состоянии в полной мере использовать имеющиеся возможности процессора.

JIT-компиляция

Оперативная компиляция ("в точный момент времени") — это общая технология, когда программы преобразуются (обычно из некоторого удобного промежуточного формата) в высоко оптимизированный машинный код непосредственно во время выполнения. HotSpot и большинство других основных производителей JVM в значительной степени полагаются на применение этого подхода.

При таком подходе во время выполнения собирается информация о вашей программе и создается профиль, который можно использовать для определения того, какие части вашей программы используются наиболее часто и больше всего выиграют от оптимизации.

Эта методика известна также как *оптимизация на основе профилирования* (profile-guided optimization — PGO).

Подсистема JIT использует ресурсы VM совместно с вашей запущенной программой, поэтому стоимости такого профилирования и любых выполняемых оптимизаций должны быть сбалансированы с ожидаемым приростом производительности.

Стоимость компиляции байт-кода в машинный код платится во время выполнения; компиляция расходует ресурсы (процессорное время, память), которые в противном случае могли бы быть использоваться для выполнения вашей программы. Поэтому JIT-компиляция выполняется экономно, а VM собирает статистику о вашей программе (ищет "горячие пятна"), чтобы знать, где лучше всего выполнять оптимизацию.

Вспомните общую архитектуру, показанную на рис. 2.2: подсистема профилирования отслеживает работающие методы. Если метод пересекает порог, делающий его пригодным для компиляции, то соответствующая подсистема запускает поток компиляции для преобразования байт-кода в машинный код.

Дизайн современных версий `javac` предназначен для производства "тупого байт-кода" (dumb bytecode). Он выполняет лишь весьма ограниченные оптимизации, выдавая вместо этого представление программы, легко понимаемое JIT-компилятором.

В разделе "Введение в измерение производительности Java" главы 5, "Микротесты и статистика", мы встретились с проблемой "разогрева" JVM в результате выполнения оптимизации, управляемой профилированием. Этот период нестабильной производительности при запуске приложения часто заставлял разработчиков Java

задавать такие вопросы, как "Нельзя ли сохранить скомпилированный код на диск и использовать его при следующем запуске приложения?" или "Разве не слишком расточительно принимать решения об оптимизации и компиляции при каждом запуске приложения?"

Проблема в том, что эти вопросы содержат некоторые базовые предположения о природе выполняемого кода приложения, и обычно они не верны. Чтобы проиллюстрировать проблему, давайте рассмотрим пример из финансовой индустрии.

Показатели безработицы в США выпускаются раз в месяц. Этот день объявления nonfarm payroll[4] генерирует трафик в торговых системах, который достаточно необычен и не наблюдается в течение оставшейся части месяца. Если бы оптимизация была сохранена с момента запуска в другой день и применялась в день NFP, работа приложения была бы не столь эффективна, как оптимизация, вычисленная для данного конкретного запуска приложения. Это привело бы к тому, что система, использующая предварительно вычисляемые оптимизации, оказалась бы менее конкурентоспособной, чем приложение с использованием PGO.

Такое поведение, при котором производительность приложения значительно варьируется между различными запусками приложения, является весьма распространенным явлением и представляет собой разновидность информации о предметной области, от которой среда наподобие Java должна защищать разработчика.

По этой причине HotSpot не пытается сохранять какую-либо профилирующую информацию и отбрасывает ее по окончании работы виртуальной машины; поэтому профилирование каждый раз выполняется с нуля.

Сравнение AOT- и JIT-компиляции

AOT-компиляция обладает тем преимуществом, что она относительно проста для понимания. Машинный код создается непосредственно из исходного текста и доступен непосредственно в виде ассемблера. Это, в свою очередь, предоставляет возможность получения кода с простыми характеристиками производительности.

Оборотной стороной медали является тот факт, что AOT-компиляция означает отказ от доступа к важной информации времени выполнения, которая могла бы помочь в принятии решений по оптимизации. В настоящее время в gcc и других компиляторах начинают появляться такие методы, как оптимизация времени компоновки (linktime optimization — LTO) и разновидности PGO, но все они находятся на ранних стадиях разработки по сравнению с их коллегами в HotSpot.

Нацеленность во время AOT-компиляции на возможности, специфичные для конкретного процессора, дает выполнимый файл, совместимый только с этим процессором. Это может быть полезной методикой для случаев, когда необходимы

[4] NFP — скомпилированное название для товаров, строительных и производственных компаний в США; охватывает 80% ВВП. — *Примеч. пер.*

низкие значения задержек или экстремальная эффективность; построение приложения на том же аппаратном обеспечении, на котором оно будет в дальнейшем работать, гарантирует, что компилятор может воспользоваться преимуществами всех доступных оптимизаций процессора.

Однако эта методика не является масштабируемой: если вы хотите получить максимальную производительность для целого ряда целевых архитектур, вам нужно будет создавать отдельные исполняемые файлы для каждой из них.

HotSpot же, напротив, может добавлять оптимизацию для новых возможностей процессора, как только они будут доступны, и при этом приложениям не придется перекомпилировать классы и JAR-файлы, чтобы ими воспользоваться. Нет ничего необычного в том, что производительность программ заметно улучшается от выпуска к выпуску HotSpot VM по мере усовершенствования JIT-компилятора.

Теперь давайте рассмотрим упорный миф о том, что "Java-программы не могут быть AOT-компилируемыми". Это просто не так: коммерческие виртуальные машины, которые предлагают AOT-компиляцию программ Java, доступны уже несколько лет, а в ряде сред они представляют основной способ развертывания приложений Java.

Наконец, начиная с Java 9 HotSpot VM начала предлагать AOT-компиляцию в качестве одного из вариантов действий, первоначально для основных классов JDK. Это первый (и весьма ограниченный) шаг по созданию AOT-скомпилированных бинарных файлов из исходных текстов Java, но он представляет собой отход от традиционных JIT-сред, которые так много сделали для популяризации Java.

Основы JIT-компиляции HotSpot

Базовая единица компиляции в HotSpot — отдельный метод, поэтому одновременно в машинный код компилируется весь байт-код, соответствующий одному методу. HotSpot также поддерживает компиляцию "горячих циклов" с использованием метода, называемого *заменой на стеке* (on-stack replacement — OSR).

OSR используется, чтобы помочь в ситуации, когда метод вызывается недостаточно часто для компиляции, но содержит цикл, который был бы подходящим кандидатом для компиляции, если бы тело цикла само по себе было методом.

Как мы увидим в следующем разделе, HotSpot использует таблицы виртуальных функций (vtables), имеющиеся в структуре метаданных класса (на которые указывает слово класса обычного указателя объекта) в качестве основного механизма для реализации JIT-компиляции.

Слова классов, таблицы виртуальных функций и настройка указателей

HotSpot — многопоточное приложение, разработанное на C++. Это может показаться упрощенным утверждением, но стоит помнить, что в результате каждая

выполняемая Java-программа на самом деле с точки зрения операционной системы всегда является частью многопоточного приложения. Даже однопоточные приложения всегда выполняются вместе с потоками VM.

Одной из наиболее важных групп потоков в HotSpot являются потоки, которые составляют подсистему JIT-компиляции. Сюда входят потоки профилирования, которые выясняют, когда метод подходит для компиляции, и потоки компилятора, которые генерируют фактический машинный код.

Общая картина заключается в том, что, когда требуется компиляция, метод помещается в поток компилятора, который и выполняет компиляцию в фоновом режиме. Общая картина процесса показана на рис. 9.6.

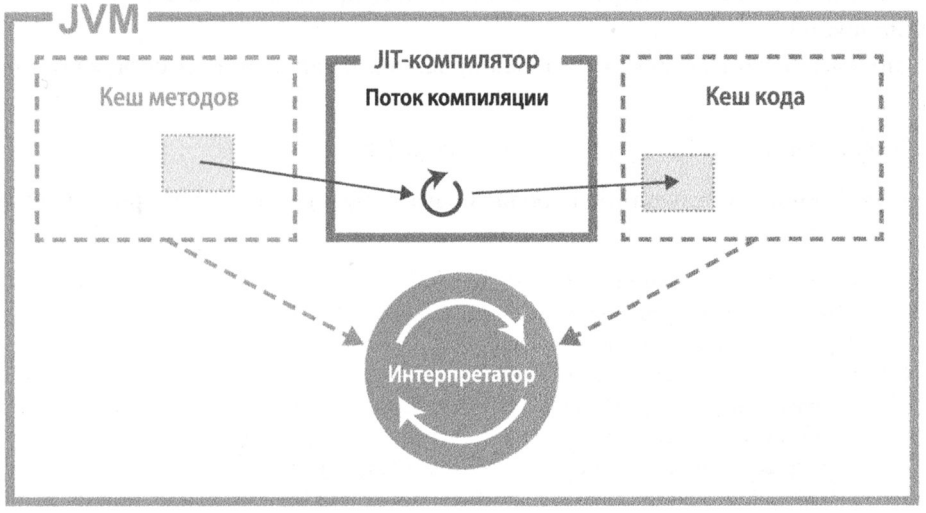

Рис. 9.6. Простая компиляция одного метода

Когда доступен оптимизированный машинный код, запись в таблице виртуальных функций соответствующего класса обновляется и указывает на вновь скомпилированный код.

Обновление указателя таблицы виртуальных функций называется *настройкой указателя* (pointer swizzling).

Это означает, что любые новые вызовы метода получат скомпилированную форму, тогда как потоки, которые в настоящее время выполняют интерпретируемую разновидность метода, завершат текущий вызов в интерпретируемом режиме, но при следующем вызове будут работать с новой скомпилированной формой метода.

OpenJDK широко переносится на множество разных архитектур, в основном на x86, x86-64 и ARM. В той или иной степени поддерживаются SPARC, Power, MIPS

и S390. В качестве операционных систем Oracle официально поддерживает Linux, MacOS и Windows. Существуют проекты с открытым исходным кодом, которые поддерживают более широкий выбор, включая BSD и встроенные системы.

Протоколирование JIT-компиляции

Важным флагом JVM, о котором должны знать все инженеры по работе с производительностью, является следующий:

```
-XX:+PrintCompilation
```

Он приводит к выводу журнала событий компиляции в стандартный поток STDOUT и позволяет инженеру получить базовое представление о том, что именно компилируется.

Например, если вызвать пример кеширования из листинга 3.1 следующим образом:

```
java -XX:+PrintCompilation optjava.Caching 2>/dev/null
```

то получающийся в результате журнал (в Java 8) будет иметь примерно следующий вид:

```
56  1   3 java.lang.Object::<init> (1 bytes)
57  2   3 java.lang.String::hashCode (55 bytes)
58  3   3 java.lang.Math::min (11 bytes)
59  4   3 java.lang.String::charAt (29 bytes)
60  5   3 java.lang.String::length (6 bytes)
60  6   3 java.lang.String::indexOf (70 bytes)
60  7   3 java.lang.AbstractStringBuilder::ensureCapacityInternal (27 bytes)
60  8 n 0 java.lang.System::arraycopy (native) (static)
60  9   1 java.lang.Object::<init> (1 bytes)
60  1   3 java.lang.Object::<init> (1 bytes) made not entrant
61 10   3 java.lang.String::equals (81 bytes)
66 11   3 java.lang.AbstractStringBuilder::append (50 bytes)
67 12   3 java.lang.String::getChars (62 bytes)
68 13   3 java.lang.String::<init> (82 bytes)
74 14 % 3 optjava.Caching::touchEveryLine @ 2 (28 bytes)
74 15   3 optjava.Caching::touchEveryLine (28 bytes)
75 16 % 4 optjava.Caching::touchEveryLine @ 2 (28 bytes)
76 17 % 3 optjava.Caching::touchEveryItem @ 2 (28 bytes)
76 14 % 3 optjava.Caching::touchEveryLine @ -2 (28 bytes) made not entrant
```

Обратите внимание, что, поскольку подавляющее большинство стандартных библиотек JRE написано на Java, они также имеют право на JIT-компиляцию наряду с кодом приложения. Поэтому мы не должны удивляться тому, что в скомпилированном коде присутствует множество методов, которых нет в приложении.

Точный набор скомпилированных методов может немного различаться от выполнения к выполнению даже в очень простом тесте. Это побочный эффект динамического характера PGO, который не должен вызывать беспокойства.

Вывод `PrintCompilation` форматируется относительно просто. Сначала указывается время, когда метод был скомпилирован (в миллисекундах от начала времени работы виртуальной машины). Затем идет число, которое указывает порядок, в котором метод был скомпилирован при этом выполнении приложения. Некоторые другие поля включают следующее.

- `n`: метод машинный
- `s`: метод синхронизирован
- `!`: метод имеет обработчик исключений
- `%`: метод был скомпилирован с использованием OSR

Уровень детализации, доступный в `PrintCompilation`, несколько ограничен. Для доступа к более подробной информации о решениях, принятых JIT-компиляторами HotSpot, можно использовать следующий параметр:

```
-XX:+LogCompilation
```

Это диагностический параметр, который мы должны разблокировать с помощью дополнительного флага:

```
-XX:+UnlockDiagnosticVMOptions
```

Этот параметр дает указание виртуальной машине выводить файл журнала, содержащий дескрипторы XML, представляющие информацию об очередности и оптимизации байт-кода в машинный код. Флаг `LogCompilation` может привести к подробному выводу и генерации сотен мегабайтов в формате XML.

Однако, как мы увидим в следующей главе, инструмент JITWatch с открытым исходным кодом может проанализировать этот файл и представить информацию в более легко воспринимаемом формате.

Другие виртуальные машины, такие как IBM J9 с Testarossa JIT, также можно заставить протоколировать информацию JIT-компилятора. Однако стандартного формата журналов JIT нет, поэтому разработчики должны либо научиться самостоятельно интерпретировать каждый формат журнала, либо использовать соответствующие инструменты.

Компиляторы в HotSpot

В JVM HotSpot фактически имеется не один, а два компилятора JIT. Они хорошо известны как C1 и C2, но иногда их называют клиентским и серверным компиляторами соответственно. Исторически C1 использовался для приложений GUI

и других "клиентских" программ, тогда как C2 использовался для длительных "серверных" приложений. Современные приложения Java, вообще говоря, размывают это различие, и HotSpot изменился таким образом, чтобы воспользоваться имеющимися новыми возможностями.

Скомпилированная единица кода иногда называется *мметодом* (от "машинный метод")[5].

Общий подход, который используют оба компилятора, заключается в том, чтобы для инициирования компиляции полагаться на ключевую меру — *количество вызовов метода* (invocation count). Как только счетчик вызовов метода достигнет определенного порога, об этом будет уведомлена виртуальная машина, которая и рассмотрит вопрос о выборе метода компиляции.

Процесс компиляции начинается с создания внутреннего представления метода. Затем к нему применяются оптимизации, которые учитывают информацию профилирования, собранную на этапе интерпретируемого выполнения. Однако внутреннее представление кода, которое создают компиляторы C1 и C2, совершенно иное. C1 спроектирован таким образом, чтобы быть более простым и иметь более короткое время компиляции, чем C2. Однако при этом имеется определенное компромиссное решение, заключающееся в том, что C1 оптимизирует код не столь полно, как C2.

Один из методов, который является общим для обоих компиляторов, — *единственное статическое присваивание* (single static assignment). Оно, по сути, преобразует программу в форму, в которой не происходит переприсваивания переменных. В терминах программирования Java программа, по сути, переписывается таким образом, что содержит только переменные `final`.

Многоуровневая компиляция в HotSpot

Начиная с Java 6 JVM поддерживает режим, именуемый *многоуровневой компиляцией* (tiered compilation). Он часто не совсем точно поясняется как работа в интерпретируемом режиме до тех пор, пока не будет доступна простая скомпилированная C1 форма, после чего будет выполнено переключение на использование этого скомпилированного кода, пока C2 не завершит более "продвинутые" оптимизации.

Однако это описание не совсем точное. Рассматривая содержимое файла `advancedThresholdPolicy.hpp`, мы видим, что в виртуальной машине имеется пять возможных уровней выполнения.

- Уровень 0: интерпретатор
- Уровень 1: C1 с полной оптимизацией (без профилирования)

[5] По-английски это звучит как *nmethod* (от native method). — *Примеч. пер.*

- Уровень 2: C1 со счетчиками вызовов и ссылок на предшествующие узлы
- Уровень 3: C1 с полным профилированием
- Уровень 4: C2

Мы также можем увидеть в табл. 9.6, что не каждый уровень используется каждым подходом к компиляции.

Таблица 9.6. Пути компиляции

Путь	Описание
0-3-4	Интерпретатор, C1 с полным профилированием, C2
0-2-3-4	Интерпретатор, C2 занят, поэтому быстрая компиляция C1, затем полная компиляция C1, затем C2
0-3-1	Тривиальный метод
0-4	Отсутствие многоуровневой компиляции (непосредственная компиляция C2)

В тривиальном случае метод начинает интерпретироваться как обычно, но C1 (с полным профилированием) может определить, что метод является тривиальным. Это означает, что компилятор C2 совершенно очевидно не создаст лучшего кода, чем C1, и поэтому компиляция завершается.

Многоуровневая компиляция использовалась в течение некоторого времени по умолчанию, и обычно нет необходимости настраивать ее работу во время настройки производительности. Однако понимание принципов ее работы полезно, поскольку она зачастую может усложнять наблюдаемое поведение скомпилированных методов и потенциально вводить в заблуждение неосведомленного инженера.

Кеш кода

JIT-скомпилированный код хранится в области памяти, именуемой *кешем кода*. Эта область также хранит и другой машинный код, относящийся к самой виртуальной машине, например к части интерпретатора.

Кеш кода имеет фиксированный максимальный размер, который устанавливается при запуске виртуальной машины. Он не может превышать этот предел, поэтому возможно его заполнение. На этом этапе дальнейшие JIT-компиляции невозможны, и далее нескомпилированный код будет выполняться только в интерпретаторе. Это повлияет на производительность и может привести к тому, что приложение будет значительно менее эффективным, чем могло бы быть.

Кеш кода реализован как куча, содержащая нераспределенную область и связанный список освобожденных блоков. Каждый раз, когда машинный код удаляется, его блок добавляется в список свободных блоков. Процесс, называемый *выметателем* (sweeper), отвечает за освобождение блоков и их возврат в свободную память.

Когда должен быть сохранен новый машинный метод, в списке свободных блоков выполняется поиск блока, достаточно большого для хранения скомпилированного кода. Если ни один подходящий блок не найден, то при условии, что кеш кода имеет достаточно свободного пространства, из нераспределенной памяти будет выделен новый блок.

Машинный код может быть удален из кеша кода, если:

- он деоптимизирован (предположения, лежащие в основе оптимизации, оказались ложными);
- он заменен другой скомпилированной версией (в случае многоуровневой компиляции);
- выгружен класс, содержащий данный метод.

Управлять максимальным размером кеша кода можно с помощью следующего переключателя виртуальной машины:

```
-XX:ReservedCodeCacheSize=<n>
```

Обратите внимание, что при включении многоуровневой компиляции нижнего порога компиляции для клиентского компилятора C1 достигнет большее количество методов. С учетом этого максимальный размер по умолчанию должен быть больше, чтобы хранить эти дополнительные скомпилированные методы.

В Java 8 на Linux x86-64 максимальными размерами по умолчанию для кеша кода являются следующие:

```
251658240 (240MB) при включенной многоуровневой
                  компиляции (-XX:+TieredCompilation)
 50331648  (48MB) при выключенной многоуровневой
                  компиляции (-XX:-TieredCompilation)
```

Фрагментация

В Java 8 и более ранних версиях кеш-код мог стать фрагментированным, если многие промежуточные компиляции компилятором C1 удаляются после их замены компиляциями C2. Это может привести к тому, что нераспределенная область будет полностью использована и все свободное пространство будет находиться в списке свободных блоков.

Распределитель кеша кода должен будет обходить этот связанный список, пока не найдет блок, достаточно большой для хранения машинного кода новой компиляции. В свою очередь, у выметателя также будет больше работы по сканированию блоков, которые могут быть перемещены в список свободных блоков.

В конечном итоге любая схема сборки мусора, которая не перемещает блоки памяти, будет подвержена фрагментации, и кеш-код не является исключением.

Без схемы уплотнения кеш кода может фрагментироваться, а это может привести к остановке компиляции; в конце концов, фрагментирование — не более чем просто еще одна форма исчерпания кеша.

Простая настройка JIT-компиляции

При выполнении настройки кода относительно легко обеспечить использование приложением JIT-компиляции.

Общий принцип простой настройки JIT-компиляции незамысловат: "любому методу, который требуется скомпилировать, для этого должны быть предоставлены ресурсы". Чтобы достичь этой цели, следуйте простой контрольной карте.

1. Сначала запустите приложение со включенным переключателем `Print Compilation`.
2. Соберите журналы, которые указывают, какие методы были скомпилированы.
3. Увеличьте размер кеша кода с помощью `ReservedCodeCacheSize`.
4. Перезапустите приложение.
5. Взгляните на множество скомпилированных методов при увеличенном кеше.

Инженеры по производительности должны учитывать небольшой недетерминизм, присущий JIT-компиляции. Помня об этом, взгляните на пару очевидных подсказок, которые легко наблюдать.

- Значимо ли увеличивается множество скомпилированных методов при увеличении размера кеша?
- Скомпилированы ли все методы, играющие важную роль на основном пути выполнения?

Если по мере увеличения размера кеша количество скомпилированных методов не увеличивается (это указывает на то, что кеш кода не используется полностью), то при условии, что схема нагрузки является репрезентативной, JIT-компилятору просто не хватает ресурсов.

На этом этапе должно быть легко подтвердить, что все методы, которые являются частью "горячих" путей выполнения, встречаются в журналах компиляции. Если это не так, то следующим шагом будет определение основной причины, по которой эти методы не компилируются.

По сути, эта стратегия гарантирует, что JIT-компиляция никогда не отключается, путем обеспечения того факта, что JVM никогда не исчерпает пространство кеша кода.

Далее в книге мы рассмотрим более сложные методы, но, несмотря на незначительные вариации между различными версиями Java, простой подход к настройке

JIT-компиляции может помочь повысить производительность для удивительного количества приложений.

Резюме

Сначала JVM работает в режиме интерпретатора байт-кода. Мы изучили основы работы интерпретатора, поскольку рабочие знания байт-кода жизненно необходимы для правильного понимания выполнения кода в JVM. Мы также ознакомились с базовой теорией JIT-компиляции.

Однако для большинства характеристик производительности поведение JIT-скомпилированного кода гораздо важнее любого аспекта интерпретатора. В следующей главе мы углубимся в теорию и практику JIT-компиляции.

Для многих приложений достаточно показанной в этой главе простой настройки кеша кода. Приложения, особенно чувствительные к характеристикам производительности, могут потребовать более глубокого изучения поведения JIT-компиляции. В следующей главе будут также описаны инструменты и методы настройки приложений с такими более строгими требованиями к производительности.

ГЛАВА 10

JIT-компиляция

В этой главе мы углубимся во внутреннюю работу JIT-компилятора JVM. Большая часть материала непосредственно применима к HotSpot; не гарантируется, что она будет одинаковой для других реализаций JVM.

Можно сказать, что тема JIT-компиляции достаточно хорошо изучена, и соответствующие реализации можно найти во многих современных средах программирования, а не только в JVM. В результате многие из описываемых методов JIT-технологий применимы и к другим JIT-компиляторам.

Из-за абстрактной и технически сложной природы данного предмета мы будем использовать различные инструменты, чтобы помочь понять и визуализировать внутреннюю работу JVM. Основным инструментом, который мы будем использовать, является JITWatch, и мы представим его в самом начале главы. После этого мы сможем объяснить конкретные JIT-оптимизации и возможности и показать, как можно наблюдать применяемые технологии и их результаты с помощью JITWatch.

Введение в JITWatch

JITWatch представляет собой инструмент JavaFX с открытым исходным кодом[1], разработанный и построенный одним из авторов этой книги — Крисом Ньюландом (Chris Newland) — в качестве личного проекта. Этот инструмент в настоящее время обеспечивается и поддерживается инициативой AdoptOpenJDK в рамках программы, проводимой лондонским сообществом программистов Java, чтобы повысить степень участия сообщества в экосистеме Java.

JITWatch позволяет разработчикам или командам лучше понимать, что на самом деле HotSpot делает с байт-кодом во время выполнения приложения. Можно настроить параметры компиляции, чтобы повысить производительность наших приложений, но при этом очень важно, чтобы у нас был механизм измерения для любых улучшений, которые мы сделали.

JITWatch обеспечивает для сравнения объективные измерения. Без этих измерений существует реальная опасность ловушки антипаттерна "Отсутствие общей

[1] `https://github.com/AdoptOpenJDK/jitwatch/`

картины", обсуждаемого в разделе "Каталог антипаттернов производительности" главы 4, "Паттерны и антипаттерны тестирования производительности".

Любой метод, который должен быть проанализирован, должен использоваться в "горячем пути выполнения" и претендовать на компиляцию. Интерпретируемые методы не являются подходящей целью для серьезной оптимизации.

Для функционирования JITWatch анализирует подробный журнал компиляции HotSpot для запущенного приложения Java и отображает его в графическом интерфейсе JavaFX. Это означает, что для работы инструмента требуется запуск приложения с определенным набором флагов.

Для работы JITWatch следует добавить соответствующие флаги (как и любые флаги, необходимые для обычных запусков приложения), если они еще не использованы:

```
-XX:+UnlockDiagnosticVMOptions -XX:+TraceClassLoading -XX:+LogCompilation
```

При включении этих флагов JVM создаст журнал, который может быть загружен в JITWatch.

Основные представления JITWatch

По завершении выполнения приложения можно запустить JITWatch, загрузить файл журнала и получить представление, подобное показанному на рис. 10.1, которое основано на реальном запуске приложения.

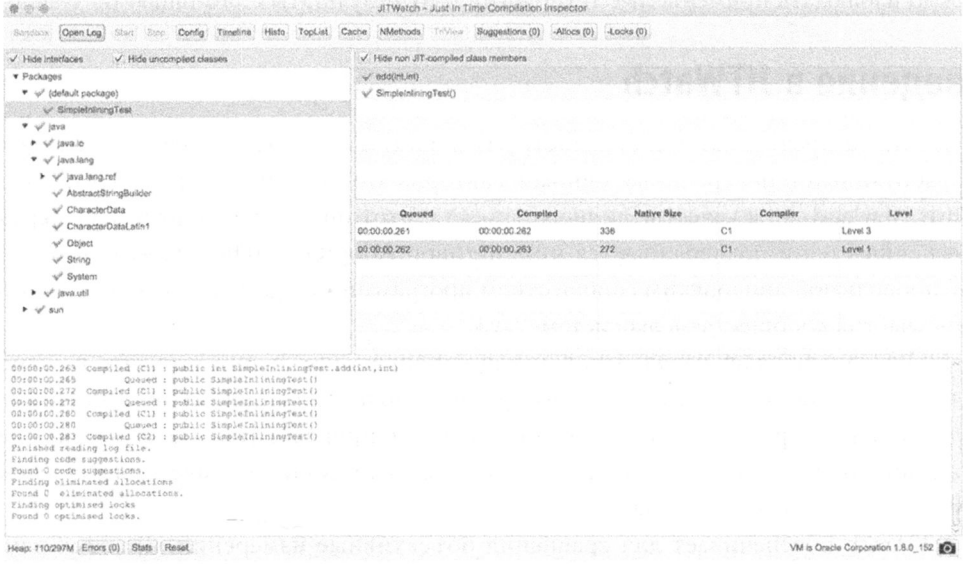

Рис. 10.1. Основное окно JITWatch

Помимо загрузки журналов из исполняемых программ, JITWatch предоставляет среду для экспериментов с поведением JIT, называемую *песочницей* (sandbox). Это представление позволяет быстро прототипировать небольшие программы и увидеть решения JIT, принятые JVM. Пример такого представления показан на рис. 10.2.

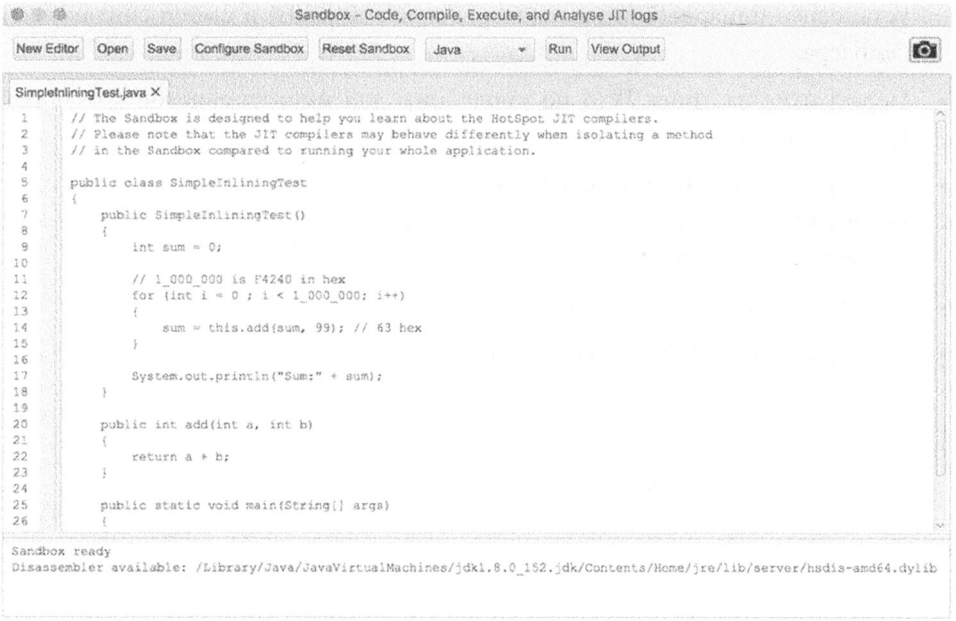

Рис. 10.2. Песочница JITWatch

Рабочий процесс песочницы позволяет создавать или загружать программу, написанную на Java или одном из других поддерживаемых JVM языков программирования, в качестве альтернативы загрузке существующего файла журнала.

После щелчка на кнопке Run JITWatch выполняет следующие действия.

1. Компилирует вашу программу в байт-код.
2. Выполняет программу в JVM со включенным протоколированием JIT.
3. Загружает журнальный файл JIT в JITWatch для анализа.

Песочница предназначена для обеспечения быстрой обратной связи, позволяющей сразу увидеть, как небольшие изменения могут повлиять на выбор оптимизации, выполняемый JVM. Помимо Java, вы можете использовать песочницу с языками Scala, Kotlin, Groovy и JavaScript (Nashorn), настроив соответствующие пути.

Песочница может быть невероятно полезной, поэтому вам следует обратить внимание на предупреждение, показанное в окне редактирования. Всегда помните, что код, выполняемый внутри песочницы, может вести себя совсем не так, как в реальном приложении!

Песочница позволяет также экспериментировать с переключателями VM, которые управляют подсистемой JIT (рис. 10.3). Например, используя настройки конфигурации песочницы, вы можете изменить поведение JVM JIT, включая следующее.

- Вывод дизассемблированных машинных методов (требуется, чтобы в JRE был установлен бинарный дизассемблер, такой как hsdis) и выбор синтаксиса ассемблера.
- Перекрытие настроек JVM по умолчанию для многоуровневой компиляции (с использованием JIT-компиляторов C1 и C2).
- Перекрытие использования сжатых обычных указателей на объекты (отключение этой настройки может упростить чтение ассемблерного листинга в силу удаления инструкций сдвига адресов).
- Отключение замены на стеке.
- Перекрытие ограничений встраивания по умолчанию.

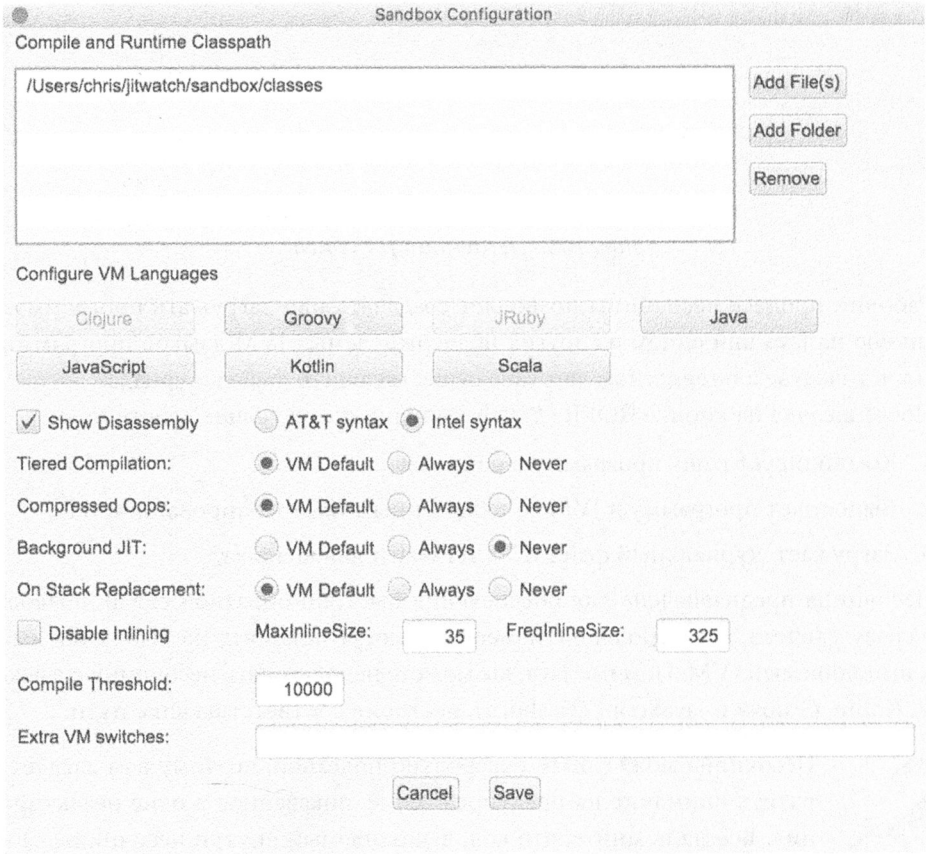

Рис. 10.3. Конфигурация песочницы JITWatch

Изменение этих установок может оказать значительное влияние на производительность JVM. Изменения категорически не рекомендуются для производственной системы, кроме экстремальных случаев, — но даже тогда они не должны изменяться без тщательного тестирования.

Песочница отличается от обычного запуска приложения JVM еще и тем, что в полномасштабном приложении JVM будет иметь возможность комбинировать оптимизации с большим представлением кода, а не только с фрагментом в песочнице. Агрессивные компиляторы JIT (как, например, C2) могут даже применять специальный набор оптимизаций в качестве первого прохода, чтобы расширить область действия программного кода, видимого оптимизатору.

Например, методика встраивания (inlining) вносит код методов в вызывающий код. Это означает, что теперь оптимизатор может рассмотреть дальнейшие возможности оптимизации (например, больший уровень встраивания или иные разновидности), которые были не очевидны перед первоначальным встраиванием. В свою очередь, это означает, что игрушечное приложение в песочнице, которое имеет только тривиальное поведение встраивания, может обрабатываться JIT-компилятором иначе, чем метод реального приложения с реальным встраиванием.

В результате большинство практиков предпочитают более сложное представление компиляции приложений, чем песочница. К счастью, основное представление *TriView*, предоставляемое JITWatch, оказывается очень емким. Оно показывает, как исходный код компилируется как в байт-код, так и в ассемблер. Такой пример показан на рис. 10.4; это свидетельствует о том, что компилятор JIT удаляет некоторые ненужные выделения памяти для объектов — важная оптимизация в современных JVM, с которой мы встретимся ниже в этой главе.

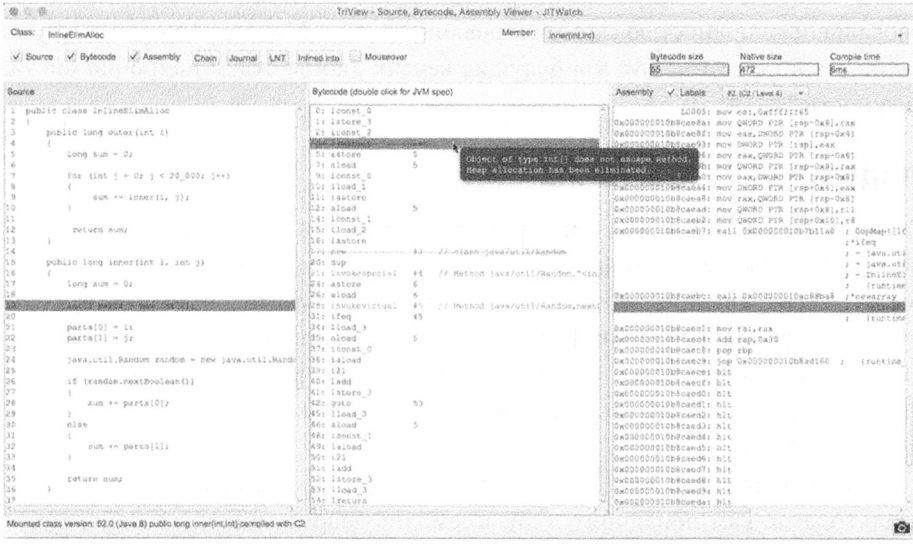

Рис. 10.4. Представление TriView в JITWatch

JITWatch позволяет визуализировать, где именно в кеше кода хранится каждый скомпилированный метод. Это относительно новая функция, находящаяся в стадии активной разработки; текущее представление показано на рис. 10.5.

Рис. 10.5. Схема кеша кода JITWatch

В Java 8 и более ранних версиях кеш кода хранит профилированные скомпилированные методы, непрофилированные скомпилированные методы и собственный машинный код VM в одной области.

В Java 9 кеш кода сегментирован, и в нем разные типы машинного кода хранятся в разных областях для сокращения времени фрагментации и выметания и повышения локальности полностью скомпилированного кода. Мы подробно рассмотрим эту особенность в главе 15, "Java 9 и будущие версии", когда будем подробно обсуждать возможности Java 9.

Отладочные JVM и hsdis

Если вы хотите углубиться в настройку и получить статистику от подсистемы JIT, то можете сделать это с помощью *отладочной JVM*. Это виртуальная машина, которая была создана для производства отладочной информации, выходящей за пределы доступной в производственной JVM, ценой уменьшения производительности.

Отладочные виртуальные машины обычно не предлагаются для загрузки основными поставщиками JVM, но отладочную JVM HotSpot можно построить из исходных текстов OpenJDK.

Если вы хотите проверить дизассемблированный машинный код, созданный JIT-компиляторами, вам понадобится дизассемблирующий бинарный файл, такой как

hsdis. Его можно построить из исходного кода OpenJDK; инструкции, как это сделать, можно найти в статье википедии JITWatch *Building hsdis*[2].

Бинарные файлы отладочной виртуальной машины для платформы Linux x86_64 могут быть загружены с сайта автора по адресу `https://chriswhocodes.com/`.

Чтобы сообщить VM о необходимости вывода ассемблерного кода метода, добавьте следующий переключатель:

```
-XX:+PrintAssembly
```

Дизассемблирование машинного кода в удобочитаемый язык ассемблера выполняется непосредственно после того, как JIT-компилятор создает этот метод. Это дорогостоящая операция, которая может влиять на производительность вашей программы, и ее следует использовать с осторожностью.

Теперь, когда мы познакомились с JITWatch, давайте взглянем на некоторые технические детали JIT-компилятора HotSpot.

Введение в JIT-компиляцию

В дополнение к представлению скомпилированного кода соответствующим инструментарием, инженер-разработчик должен знать о том, как VM собирает данные и какие оптимизации производит над выполняющейся программой.

Мы уже видели, что HotSpot использует оптимизацию на основе профилирования (PGO) для руководства решениями о JIT-компиляции. "За сценой" HotSpot хранит данные профиля о запущенной программе в структурах, называемых объектами данных метода (method data object — MDO).

MDO используются интерпретатором байт-кода и компилятором C1 для записи информации, используемой JIT-компиляторами при определении того, какие оптимизации нужно сделать. MDO хранят информацию, такую как вызванные методы, предпринятые ветвления, и типы, наблюдаемые в точках вызовов.

Кроме того, поддерживаются счетчики, которые записывают "горячесть" профилируемого свойства, при этом их значения во время профилирования постоянно заменяются новыми с уменьшением значимости старых значений — это гарантирует, что методы компилируются только в том случае, если они все еще "горячие", когда достигают начала очереди компиляции.

После сбора данных профилирования и принятия решения о компиляции учитываются конкретные детали отдельного компилятора. Компилятор создает внутреннее

[2] `https://github.com/AdoptOpenJDK/jitwatch/wiki/Building-hsdis`

представление компилируемого кода; точный характер представления зависит от того, какой именно компилятор (C1 или C2) используется.

На основе этого внутреннего представления компилятор будет выполнять максимальную оптимизацию кода. JIT-компиляторы от HotSpot способны выполнять широкий спектр современных методов оптимизации, в том числе следующие:

- встраивание (inlining);
- разворачивание циклов (loop unrolling);
- escape-анализ[3];
- пропуск блокировки (lock elision) и укрупнение блокировки (lock coarsening);
- мономорфная диспетчеризация[4];
- внутренние (встроенные в компилятор) операции (интринсики — intrinsics);
- замена на стеке (on-stack replacement).

В следующих разделах мы поочередно рассмотрим все эти методы.

По мере знакомства с каждой из методик важно помнить, что большинство этих оптимизаций частично или полностью зависят от информации и поддержки времени выполнения.

Два JIT-компилятора в HotSpot также используют разные подмножества методов оптимизации и имеют разные представления о том, как подходить к компиляции. В частности, C1 не занимается *спекулятивной оптимизацией* (speculative optimization) (это использование оптимизаций, которые основаны на недоказанном предположении о характере исполнения). Агрессивные оптимизаторы (например, C2) делают предположения на основе наблюдаемого поведения времени выполнения и прибегают к оптимизации на их основе. Такое упрощающее допущение может позволить добиться большого (иногда очень большого) ускорения производительности.

Чтобы защититься от неверных предположений, позднее признанных недействительными, спекулятивная оптимизация всегда защищена проверкой, известной под названием *сторож* (guard). Сторож убеждается, что предположение все еще выполняется, и этот факт проверяется непосредственно перед каждым запуском оптимизированного кода.

Если сторож постоянно терпит неудачу, скомпилированный код перестает быть безопасным и должен быть удален. HotSpot немедленно *деоптимизирует* метод и понижает его до интерпретируемого режима, чтобы предотвратить выполнение некорректного кода.

---

[3] Escape analysis, или анализ локальности, — определение области достижимости для указателей объектов. — *Примеч. пер.*

[4] Monomorphic dispatch — прямой вызов скомпилированного кода метода вне зависимости от объекта. — *Примеч. пер.*

Встраивание

Встраивание (inlining) представляет собой процесс копирования содержимого вызываемого метода в место его вызова.

Тем самым устраняются накладные расходы, связанные с вызовом метода, которые, хотя и не велики, могут включать в себя:

- настройку передаваемых параметров;
- поиск точного вызванного метода;
- создание новых структур данных времени выполнения для нового кадра стека (таких, как локальные переменные и стек выполнения);
- передача управления новому методу;
- возможный возврат результата вызывающему методу.

Встраивание — одна из первых оптимизаций, применяемых JIT-компиляторами, известная как шлюзовая оптимизация (*gateway optimization*), поскольку она соединяет связанный код, устраняя границы метода.

Пусть имеется следующий код:

```
int result = add(a, b);
private int add(int x, int y) {
    return x + y;
}
```

Оптимизация встраивания копирует тело метода `add()` в точку вызова, по сути, давая следующий код:

```
int result = a + b;
```

Встраивание компилятором HotSpot позволяет разработчику писать хорошо организованный и повторно используемый код. Он отображает представление о том, что разработчику не нужно заниматься ручной микрооптимизацией. Вместо этого HotSpot использует автоматический статистический анализ для определения того, когда вызываемый код следует объединить с вызывающим. Таким образом, встраивание расширяет горизонты для других методов оптимизации, в том числе:

- анализ локальности;
- удаление "мертвого" кода;
- разворачивание циклов;
- пропуск блокировки.

Границы встраивания

Иногда виртуальной машине необходимо устанавливать ограничения на подсистему встраивания. Например, виртуальной машине может потребоваться управлять:

- количеством времени, которое JIT-компилятор затрачивает на оптимизацию метода;
- размером созданного машинного кода (а следовательно, использованием памяти кеша кода).

Без этих ограничений компилятор мог бы встраивать очень глубокие цепочки вызовов или заполнять кеш кода огромными машинными методами. Общий принцип — драгоценность ресурсов JIT-компиляции — здесь снова в силе.

HotSpot рассматривает различные факторы при определении того, следует ли встраивать метод, в том числе следующие:

- размер байт-кода встраиваемого метода;
- глубина встраиваемого метода в текущей цепочке вызовов;
- объем памяти в кеше кода, уже занятый скомпилированной версией этого метода.

На рис. 10.6 мы видим визуализацию JITWatch того, как JIT-компилятор встроил цепочку вызовов методов в их конечный вызывающий метод, но отклонил встраивание метода, специально созданного так, чтобы превысить максимальный предел встраивания по умолчанию.

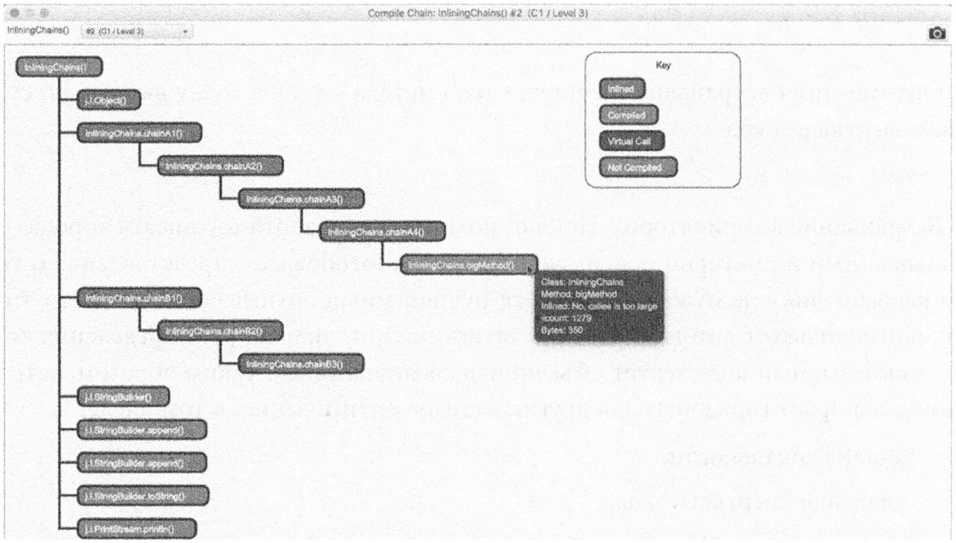

Рис. 10.6. Цепочка вызовов методов встроена в родительский метод

Эта глубина встраивания является типичной и помогает расширить сферу оптимизации.

Настройка подсистемы встраивания

После того как мы определили, что все важные методы скомпилированы, и скорректировали кеш кода для размещения наиболее значимых методов приложения, следующим шагом будет рассмотрение встраивания. В табл. 10.1 мы видим основные переключатели JVM, используемые для управления поведением подсистемы встраивания. Эти переключатели могут использоваться для расширения методологии простой настройки JIT, показанной в разделе "Простая настройка JIT-компиляции" главы 9, "Выполнение кода в JMV".

Таблица 10.1. Переключатели встраивания

Переключатель	Значение по умолчанию (JDK 8, Linux x86_64)	Пояснение
`-XX:MaxInlineSize=<n>`	35 байт байт-кода	Встраивать методы до этого размера
`-XX:FreqInlineSize=<n>`	325 байт байт-кода	Встраивать "горячие" (часто вызываемые) методы до этого размера
`-XX:InlineSmallCode=<n>`	1000 байт машинного кода (одноуровневая компиляция) 2000 байт машинного кода (многоуровневая компиляция)	Не встраивать методы, если уже имеется компиляция последнего уровня, которая занимает в кеше кода более указанного количества памяти
`-XX:MaxInlineLevel=<n>`	9	Не встраивать кадры вызовов глубже указанного уровня

Если важные методы не встроены (например, из-за того, что они слишком велики для встраивания), то в некоторых случаях может быть целесообразной такая настройка параметров JVM, чтобы сделать эти методы встроенными. В этих условиях начните с корректировки либо `-XX:MaxInlineSize`, либо `-XX:FreqInlineSize` и проверьте, не привело ли такое изменение к наблюдаемым улучшениям.

Любое действие по настройке, затрагивающее указанные параметры, должно быть, как всегда, проверено с использованием наблюдаемых данных. Неспособность рассмотреть фактические данные и доказать необходимость изменения параметра лежит в основе антипаттерна "Надувательство с переключателями" (читайте соответствующий раздел в главе 4, "Паттерны и антипаттерны тестирования производительности").

Разворачивание циклов

Когда какой-либо вызов метода внутри цикла оказывается встроенным (где это возможно), компилятор лучше понимает размер и стоимость каждой итерации

цикла. Исходя из этого он может рассмотреть возможность *разворачивания цикла* (unrolling the loop), чтобы уменьшить количество раз, когда выполнение должно вернуться к началу цикла.

Каждая *обратная ветвь* (back branch) может отрицательно влиять на процессор, так как процессор при этом выгружает конвейер входящих команд. В общем случае чем короче тело цикла, тем выше относительная стоимость обратной ветви. Поэтому HotSpot принимает решения о том, следует ли разворачивать циклы, на основе нескольких критериев, включая следующие.

- Тип переменной-счетчика цикла (обычно `int` или `long`, а не объектный тип).
- Шаг цикла (как именно счетчик цикла изменяется на каждой итерации).
- Количество точек выхода (`return` или `break`) в цикле.

Рассмотрим некоторые примеры методов, которые выполняют суммирование данных, последовательно извлекаемых из массива. Эта схема доступа может быть представлена в ассемблере с использованием адресации `[base, index, offset]` ([база, индекс, смещение]).

- Регистр `base` содержит начальный адрес данных в массиве.
- Регистр `index` содержит счетчик цикла (который умножается на размер типа данных).
- `offset` используется для смещения каждого развернутого обращения.

```
add rbx, QWORD PTR [base register + index register * size + offset]
```

Поведение разворачивания цикла может меняться от версии к версии HotSpot и сильно зависит от используемой архитектуры.

При цикле над массивом `long[]` мы можем посмотреть на условия, при которых цикл будет развернут. Когда цикл обращается к массиву, HotSpot может исключить проверки границ массива, разделив цикл на три раздела, как показано в табл. 10.2.

Таблица 10.2. Устранение проверок границ

Раздел цикла	Есть ли проверка границ	Назначение
Предварительный	Да	Выполняет начальные итерации с проверкой выхода за границы
Основной	Нет	Шаг цикла используется для вычисления максимального количество итераций, которые могут быть выполнены без необходимости проверки границ
Окончательный	Да	Выполняет оставшиеся итерации с проверкой выхода за границы

Вот пример кода настройки для создания массива, с которым будет работать цикл:

```
private static final int MAX = 1_000_000;
private long[] data = new long[MAX];
private void createData()
{
    java.util.Random random = new java.util.Random();
    for (int i = 0; i < MAX; i++)
    {
        data[i] = random.nextLong();
    }
}
```

Мы можем использовать тест JMH для сравнения производительности итераций по одному и тому же массиву с использованием счетчиков `int` и `long`:

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class LoopUnrollingCounter
{
    private static final int MAX = 1_000_000;
    private long[] data = new long[MAX];
    @Setup
    public void createData()
    {
        java.util.Random random = new java.util.Random();
        for (int i = 0; i < MAX; i++)
        {
            data[i] = random.nextLong();
        }
    }
    @Benchmark
    public long intStride1()
    {
        long sum = 0;
        for (int i = 0; i < MAX; i++)
        {
            sum += data[i];
        }
        return sum;
    }
```

```
    @Benchmark
    public long longStride1()
    {
        long sum = 0;
        for (long l = 0; l < MAX; l++)
        {
            sum += data[(int) l];
        }
        return sum;
    }
}
```

Результаты имеют следующий вид:

```
Benchmark                          Mode  Cnt    Score   Error Units
LoopUnrollingCounter.intStride1   thrpt 200 2423.818 ± 2.547 ops/s
LoopUnrollingCounter.longStride1 thrpt 200 1469.833 ± 0.721 ops/s
```

Цикл со счетчиком `int` выполняет примерно на 64% операций в секунду больше.

Если бы мы погрузились в ассемблер, то увидели бы, что тело цикла со счетчиком `long` развернуто не будет, а цикл будет также содержать опрос точек безопасности. Эти проверки точек безопасности вставлены JIT-компилятором для уменьшения вероятности того, что скомпилированный код будет работать в течение длительного времени без проверки флага точек безопасности (см. раздел "Точки безопасности JVM" в главе 7, "Вглубь сборки мусора").

Другие микротесты могут включать использование *переменного шага*, когда инкремент сохраняется в переменной и не является константой времени компиляции. При переменном шаге мы увидим, что цикл не разворачивается, а точка безопасности вставляется перед обратной ветвью.

Резюме к разворачиванию циклов

HotSpot содержит ряд оптимизаций для разворачивания циклов.

- Он может оптимизировать циклы со счетчиком, которые используют счетчик типа `int`, `short` или `char`.
- Он может разворачивать тела циклов и удалять опросы точек безопасности.
- Разворачивание цикла уменьшает количество обратных ветвей и связанные с ними затраты на прогнозирование ветвлений.
- Удаление опросов точек безопасности дополнительно уменьшает работу, выполняемую каждой итерацией цикла.

Однако вы всегда должны убеждаться в достигнутом результате самостоятельно, не считая эти примеры справедливыми для всех архитектур и версий HotSpot.

Анализ локальности

HotSpot может выполнять анализ на основе областей видимости для выяснения, является ли работа, выполненная внутри метода, видимой или имеет побочные эффекты за пределами этого метода. Эта методика анализа локальности (escape analysis) может использоваться для определения того, является ли объект, созданный (размещенный) внутри метода, видимым вне области метода.

Оптимизация на основе анализа локальности выполняется после того, как имеет место какое-либо встраивание. Последнее копирует тело вызываемого метода в точку вызова, что предотвращает маркировку объектов, которые передаются только как аргументы метода, как нелокальных (видимых вне метода).

На этапе анализа локальности HotSpot подразделяет потенциальные нелокальные (видимые вне метода) объекты на три типа. Следующая часть кода из файла `hotspot/src/share/vm/opto/escape.hpp` описывает различные возможные сценарии нелокальности:

```
typedef enum {
    NoEscape = 1,     // Объект не выходит за рамки метода или
                      // потока и не передается вызову. Он может
                      // быть  заменен скаляром.
    ArgEscape = 2,    // Объект не выходит за рамки метода или потока,
                      // но передается в качестве  аргумента вызова
                      // (или аргумент ссылается на него) и во время
                      // вызова не покидает метод.
    GlobalEscape = 3 // Объект покидает метод или поток (виден извне).
}
```

Устранение выделения памяти в куче

Создание новых объектов внутри циклов может увеличить нагрузку на подсистему распределения памяти. Генерация большого количества короткоживущих объектов требует частых небольших сборок мусора для их очистки. Скорость выделения может быть настолько высокой, что молодое поколение кучи заполнится и короткоживущие объекты будут досрочно перенесены в старое поколение. Если это произойдет, для их очистки потребуется более дорогостоящая полная сборка мусора.

Оптимизация на основе анализа локальности HotSpot предназначена для того, чтобы разработчики могли писать идиоматический код Java, не беспокоясь о скорости выделения объектов.

Убедившись, что выделенный объект не покидает метод (классифицирован как `NoEscape`), виртуальная машина может применить оптимизацию, называемую

скалярной заменой (scalar replacement). Поля в объекте становятся скалярными значениями, подобно тому, как если бы все поля были локальными переменными, а не полями объекта. Затем они могут быть размещены в регистрах процессора компонентом HotSpot, именуемым *распределителем регистров* (register allocator).

Если доступных свободных регистров недостаточно, то скалярные значения могут быть помещены в текущий кадр стека (это действие известно как *вытеснение стека* (stack spill)).

Цель анализа локальности состоит в том, чтобы определить, можно ли избежать выделения памяти из кучи. Если это возможно, объект может быть автоматически выделен в стеке, и, таким образом, нагрузка на сборку мусора может быть немного уменьшена.

Давайте рассмотрим пример выделения объекта, который будет классифицироваться как `NoEscape`, поскольку экземпляр `MyObj` не покидает область видимости метода:

```
public long noEscape()
{
    long sum = 0;
    for (int i = 0; i < 1_000_000; i++)
    {
        MyObj foo = new MyObj(i); // foo не покидает метод (NoEscape)
        sum += foo.bar();
    }
    return sum;
}
```

Ниже приведен пример выделения объекта, который будет классифицирован как `ArgEscape`, поскольку экземпляр `MyObj` передается в качестве аргумента методу `extBar()`.

```
public long argEscape()
{
    long sum = 0;
    for (int i = 0; i < 1_000_000; i++)
    {
        MyObj foo = new MyObj(i);
        sum += extBar(foo); // foo передается как аргумент
    }                       // методу extBar (ArgEscape)
    return sum;
}
```

Если вызов `extBar()` был встроен в тело цикла до выполнения анализа локальности, то `MyObj` будет классифицирован как `NoEscape` и может не выделяться в куче.

Блокировки и анализ локальности

HotSpot может использовать анализ локальности и некоторые связанные с ними методы для оптимизации производительности блокировок.

Это относится только ко встроенным блокировкам (тем, которые используют синхронизацию). Блокировки из `java.util.concurrent` не годятся для этих оптимизаций.

Доступны следующие ключевые оптимизации блокировок:

- удаление блокировок для локальных (nonescaping) объектов (*пропуск блокировки* — lock elision);
- слияние последовательных блокируемых областей, совместно использующих одну и ту же блокировку (*укрупнение блокировок* — lock coarsening);
- обнаружение блоков, в которых одна и та же блокировка многократно захватывается без разблокирования (*вложенные блокировки* — nested locks).

Когда встречаются последовательные блокировки для одного и того же объекта, HotSpot проверяет, нельзя ли увеличить блокируемую область. Для этого, когда HotSpot встречает блокировку, выполняется поиск в обратном направлении, чтобы попытаться найти разблокирование для того же самого объекта. Если совпадение найдено, будет рассматриваться вопрос о том, нельзя ли объединить эти две области блокировки, чтобы создать единую область большего размера.

Подробнее об этом можно узнать в спецификации JVM[5]. На рис. 10.7 можно увидеть этот эффект непосредственно в JITWatch.

Оптимизация укрупнения блокировок включена по умолчанию, но, чтобы увидеть ее действие, ее можно отключить с помощью переключателя `-XX:-EliminateLocks`.

HotSpot также в состоянии обнаружить вложенные блокировки, которые блокируют один и тот же объект, и удалить внутренние блокировки, так как эта блокировка уже захвачена потоком.

На момент написания книги устранение вложенных блокировок в Java 8 работает с блокировками, объявленными как `static final`, и с блокировками над `this`.

Оптимизация вложенных блокировок включена по умолчанию, но ее можно отключить с помощью переключателя `-XX:-EliminateNestedLocks`. Обнаружение вложенных блокировок в JITWatch можно увидеть на рис. 10.8.

[5] `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-3.html#jvms-3.14`

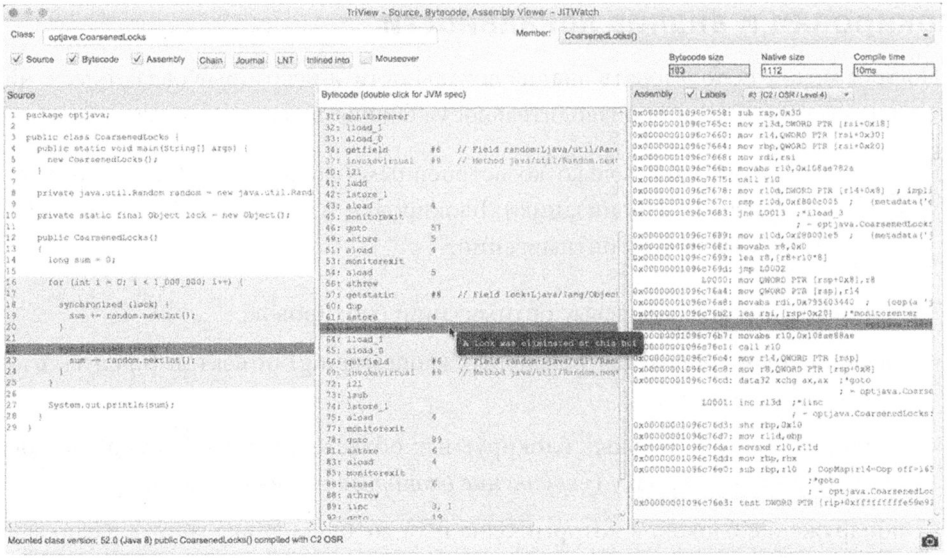

Рис. 10.7. Укрупнение блокировок

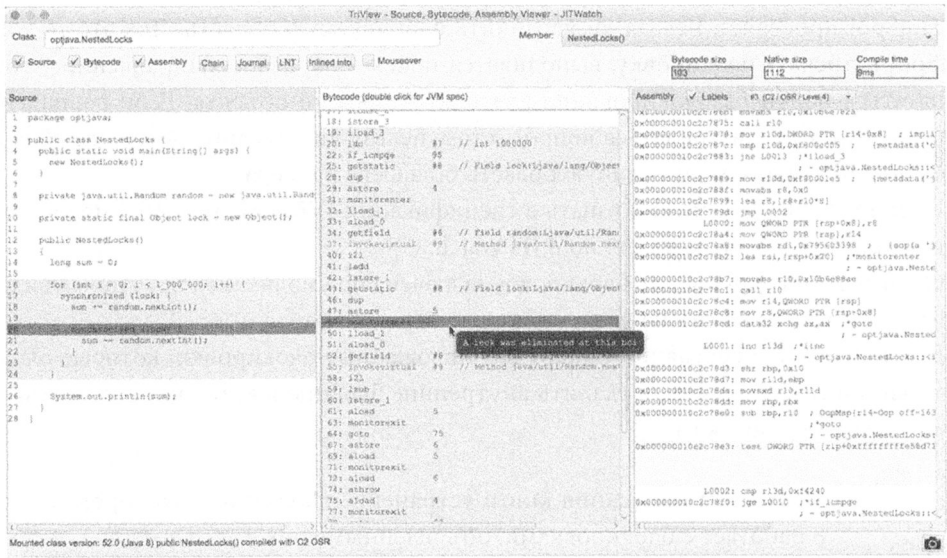

Рис. 10.8. Устранение вложенных блокировок

HotSpot автоматически вычисляет, когда блокировки можно безопасно укрупнить или устранить. Такие инструменты, как JITWatch, могут визуализировать, где именно были оптимизированы блокировки, и если вы используете отладочную версию JVM, то сможете вывести дополнительную информацию о блокировках.

Ограничения анализа локальности

Анализ локальности, как и другие оптимизации, имеет свои ограничения, поскольку каждое выделение памяти, сделанное не в куче, должно происходить где-то в другом месте, а регистры процессора и память стека — относительно скудные ресурсы. Одно из ограничений в HotSpot заключается в том, что по умолчанию массивы более чем из 64 элементов не рассматриваются в процессе анализа локальности. Этот размер управляется следующим переключателем VM:

```
-XX:EliminateAllocationArraySizeLimit=<n>
```

Рассмотрим горячий путь кода, который содержит выделение временного массива для чтения из буфера. Если массив не выходит за пределы области метода, то анализ локальности должен предотвращать выделение памяти в куче. Однако если длина массива составляет более 64 элементов (даже если не все они используются), то он должен храниться в куче, что может быстро увеличить скорость выделения памяти у этого кода.

В следующем тесте JMH методы тестирования выделяют локальные (не выходящие из области видимости метода) массивы с размерами 63, 64 и 65.

Размер массива, равный 63, тестируется для того, чтобы убедиться, что скорость работы с размером в 64 элемента не выше скорости работы с размером в 65 элементов просто из-за выравнивания памяти.

В каждом тесте используются только два элемента массива, `a[0]` и `a[1]`, но рассматриваемое ограничение касается только длины массива, а не максимального используемого индекса:

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class EscapeTestArraySize {
    private java.util.Random random = new java.util.Random();
    @Benchmark
    public long arraySize63() {
        int[] a = new int[63];
        a[0] = random.nextInt();
        a[1] = random.nextInt();
        return a[0] + a[1];
    }
    @Benchmark
```

```
    public long arraySize64() {
        int[] a = new int[64];
        a[0] = random.nextInt();
        a[1] = random.nextInt();
        return a[0] + a[1];
    }
    @Benchmark
    public long arraySize65() {
        int[] a = new int[65];
        a[0] = random.nextInt();
        a[1] = random.nextInt();
        return a[0] + a[1];
    }
}
```

Результаты показывают значительное снижение производительности после того, как выделение массива не может извлечь выгоду из оптимизации анализа локальности:

```
Benchmark                          Mode Cnt        Score      Error  Units
EscapeTestArraySize.arraySize63 thrpt 200 49824186.696 ±  9366.780 ops/s
EscapeTestArraySize.arraySize64 thrpt 200 49815447.849 ±  2328.928 ops/s
EscapeTestArraySize.arraySize65 thrpt 200 21115003.388 ± 34005.817 ops/s
```

Обнаружив, что нужно выделить более крупный массив в горячем коде, вы можете поручить виртуальной машине разрешить оптимизацию больших массивов. Повторный запуск теста с пределом в 65 элементов показывает, что производительность восстанавливается:

```
$ java -XX:EliminateAllocationArraySizeLimit=65 -jar target/benchmarks.jar

Benchmark                          Mode Cnt        Score      Error Units
EscapeTestArraySize.arraySize63 thrpt 200 49814492.787 ± 2283.941 ops/s
EscapeTestArraySize.arraySize64 thrpt 200 49815595.566 ± 5833.359 ops/s
EscapeTestArraySize.arraySize65 thrpt 200 49818143.279 ± 2347.695 ops/s
```

Другим важным ограничением является то, что HotSpot не поддерживает *частичный анализ локальности* (partial escape analysis) (известный также как анализ локальности, *чувствительный к потоку выполнения* (flow-sensitive)).

jRockit JVM поддерживала частичный анализ локальности, но эта технология не переносилась в HotSpot после слияния этих двух JVM.

Если обнаружено, что объект выходит из области видимости метода в любой ветви, то оптимизация, позволяющая избежать выделения объекта в куче, применяться не будет. В следующем примере, в предположении использования обеих ветвей

объект иногда может выходить из метода и должен быть классифицирован как ArgEscape. Это увеличит скорость выделения памяти и добавит дополнительную нагрузку на сборку мусора.

```
for (int i = 0; i < 100_000_000; i++)
{
    Object mightEscape = new Object(i);
    if (condition)
    {
        result += inlineableMethod(mightEscape);
    } else {
        result += tooBigToInline(mightEscape);
    }
}
```

Если в коде, как здесь показано, можно локализовать выделение объекта в пределах ветви, из которой нет утечки объекта, то анализ локальности в этом пути, определенно, принесет выгоду.

```
for (int i = 0; i < 100_000_000; i++)
{
    if (condition)
    {
        Object mightEscape = new Object(i);
        result += inlineableMethod(mightEscape);
    } else {
        Object mightEscape = new Object(i);
        result += tooBigToInline(mightEscape);
    }
}
```

Мономорфная диспетчеризация

Многие из умозрительных оптимизаций, которые предпринимает компилятор HotSpot C2, основаны на эмпирических исследованиях. Одним из примеров является методика, называемая *мономорфной диспетчеризацией* (monomorphic dispatch) (термин *мономорфный* происходит из греческого языка и означает "единственная форма").

Он опирается на странный, но мощный наблюдаемый факт: в коде, написанном человеком, очень часто типом объекта-получателя в каждой отдельной точке вызова оказывается только один тип времени выполнения.

Это в общем случае является отражением способа проектирования людьми объектно-ориентированного программного обеспечения, а не некой особенностью конкретно Java или JVM.

То есть, когда мы вызываем метод объекта (если мы исследуем тип времени выполнения этого объекта при первом вызове этого метода), скорее всего, он всегда будет одного и того же типа для всех последующих вызовов.

Если это умозрительное предположение верно, то вызов метода в данной точке может быть оптимизирован. В частности, можно устранить косвенность поиска метода в таблице виртуальных функций. Если это всегда один и тот же тип, то мы можем вычислить целевой тип вызова один раз и заменить команду `invokevirtual` быстрой проверкой типа (сторожем), а затем — ветвлением к скомпилированному телу метода.

Другими словами, поиск виртуальных функций и связанное с ним косвенное использование указателя класса и таблицы виртуальных функций должны выполняться только один раз и могут быть кешированы для будущих вызовов в этой точке.

Для точки вызова `invokevirtual` единственными возможными типами, которые можно увидеть, являются базовый тип, определяющий метод, который должен быть выполнен, и любые его подтипы. Это принцип подстановки Лисков в другом обличье.

Рассмотрим следующий фрагмент кода:

```
java.util.Date date = getDate();
System.out.println(date.toInstant());
```

Если метод `getDate()` всегда возвращает экземпляр `java.util.Date`, то вызов `toInstant()` можно считать мономорфным. Однако, если после многих итераций этого кода `getDate()` внезапно возвращает экземпляр `java.sql.Date`, то предположение о мономорфности становится недействительным, так как теперь нужно вызывать совершенно другую реализацию `toInstant()`.

Решение HotSpot состоит в том, чтобы отказаться от оптимизации и вернуться в точке вызова к полной виртуальной диспетчеризации. Сторож, который используется для защиты мономорфного вызова, очень прост: это простое равенство слов классов, и оно проверяется перед каждой командой вызова, чтобы гарантировать, что неправильный код выполняться не будет.

В типичном приложении очень большое количество вызовов будет мономорфным. У HotSpot есть также еще одна оптимизация, которая используется реже, — *биморфная диспетчеризация* (bimorphic dispatch). Это позволяет обрабатывать аналогично мономорфному случаю два разных типа — путем кеширования двух разных слов класса для точки вызова.

Точки вызовов, не являющиеся мономорфными или биморфными, известны как *мегаморфные* (от греческого "много форм"). Если вы обнаружите, что у вас есть мегаморфная точка вызова с небольшим количеством наблюдаемых типов, то можете использовать один трюк для некоторого повышения производительности. Он работает

путем "отслаивания" типов от исходной точки вызова с использованием проверок instanceof, так что вы оставляете только биморфную точку вызова, которая наблюдает за двумя конкретными типами.

Пример этого подхода можно увидеть ниже:

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

interface Shape {
    int getSides();
}

class Triangle implements Shape {
    public int getSides() {
        return 3;
    }
}

class Square implements Shape {
    public int getSides() {
        return 4;
    }
}

class Octagon implements Shape {
    public int getSides() {
        return 8;
    }
}

@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class PeelMegamorphicCallsite {
    private java.util.Random random = new java.util.Random();
    private Shape triangle = new Triangle();
    private Shape square = new Square();
    private Shape octagon = new Octagon();
    @Benchmark
    public int runBimorphic() {
        Shape currentShape = null;
        switch (random.nextInt(2))
        {
        case 0:
            currentShape = triangle;
```

```
            break;
        case 1:
            currentShape = square;
            break;
        }
        return currentShape.getSides();
    }
    @Benchmark
    public int runMegamorphic() {
        Shape currentShape = null;
        switch (random.nextInt(3))
        {
        case 0:
            currentShape = triangle;
            break;
        case 1:
            currentShape = square;
            break;
        case 2:
            currentShape = octagon;
            break;
        }
        return currentShape.getSides();
    }
    @Benchmark
    public int runPeeledMegamorphic() {
        Shape currentShape = null;
        switch (random.nextInt(3))
        {
        case 0:
            currentShape = triangle;
            break;
        case 1:
            currentShape = square;
            break;
        case 2:
            currentShape = octagon;
            break;
        }
        // Убираем один наблюдаемый тип из исходной точки вызова
        if (currentShape instanceof Triangle) {
            return ((Triangle) currentShape).getSides();
        }
        else {
            return currentShape.getSides(); // Теперь - биморфный
        }
    }
}
```

Выполнение теста производительности дает следующий результат:

```
Benchmark                                  Mode  Cnt     Score    Error Units
PeelMega...Callsite.runBimorphic           thrpt 200 75844310 ±  43557 ops/s
PeelMega...Callsite.runMegamorphic         thrpt 200 54650385 ±  91283 ops/s
PeelMega...Callsite.runPeeledMegamorphic thrpt 200 62021478 ± 150092 ops/s
```

Когда в точке вызова наблюдаются две реализации, выполняется биморфное встраивание, которое выполняет на 38% больше операций в секунду, чем в точке мегаморфного вызова, наблюдающего за тремя реализациями (в результате здесь диспетчеризация метода остается виртуальным вызовом). Обратите внимание, что это не совсем справедливое сравнение, так как поведение кода отличается.

Когда один из наблюдаемых типов переносится в другую точку вызова, программа выполняет на 13% больше операций в секунду, чем в случае мегаморфного кода.

Диспетчеризация методов и вытекающие из нее последствия для производительности — большая и глубокая тема. Алексей Шипилев (Aleksey Shipilëv) дает мастер-класс на эту тему в своем блоге *Черная магия диспетчеризации методов в Java*.[6]

Встроенные операции

Встроенная операция (интринсик — intrinsic) — это тщательно настроенная машинная реализация метода, который заранее известен JVM, а не генерируется подсистемой JIT динамически. Такие операции используются для критически важных методов, в которых функциональность поддерживается конкретными возможностями операционной системы или архитектуры процессора. Это делает их привязанными к платформе, так что некоторые встроенные операции могут не поддерживаться на каждой платформе.

При запуске JVM выполняется тестирование процессора и создается список его доступных возможностей. Это означает, что решение о том, какие оптимизации использовать, может быть отложено до времени выполнения и не обязательно должно выполняться во время компиляции кода.

Встроенные операции могут быть реализованы как в интерпретаторе, так и в JIT-компиляторах C1 и C2.

Примеры некоторых распространенных встроенных операций показаны в табл. 10.3.

Шаблоны встраиваемых операций доступны для просмотра в исходном коде HotSpot в OpenJDK. Они содержатся в файлах `.ad` (суффикс означает "architecture dependent" —"зависимый от архитектуры").

[6] `https://shipilev.net/blog/2015/black-magic-method-dispatch/`

Таблица 10.3. Примеры встроенных методов

Метод	Описание
`java.lang.System.arraycopy()`	Быстрое копирование с использованием поддержки векторов процессором
`java.lang.System.currentTimeMillis()`	Быстрая реализация, предоставляемая большинством операционных систем
`java.lang.Math.min()`	На некоторых процессорах может быть выполнен без ветвления
Прочие методы `java.lang.Math`	На некоторых процессорах поддерживаются непосредственные соответствующие команды
Криптографические функции (например, AES)	Аппаратное ускорение может привести к значительному повышению производительности

В Java 9 в исходные тексты добавлена аннотация `@HotSpotIntrinsicCandidate` для указания, что может быть доступна встроенная операция.

Для архитектуры x86_64 их можно найти в файле `hotspot/src/cpu/x86/vm/x86_64.ad`.

Например, чтобы вычислить логарифм по основанию 10 некоторого числа, мы можем использовать метод из `java.lang.Math`:

```
public static double log10(double a)
```

На архитектуре x86_64 этот расчет может быть выполнен с помощью двух команд математического сопроцессора (FPU).

1. Вычисление логарифма по основанию 10 константы 2.
2. Умножение полученного значения на логарифм по основанию 2 вашего аргумента.

Код встроенной операции для этих действий имеет следующий вид:

```
instruct log10D_reg(regD dst) %{
    // Источник и результат – операнды double в регистрах XMM
    // match(Set dst (Log10D dst));
    // fldlg2 ; поместить log_10(2) в стек FPU; 80-битное значение
    // fyl2x  ; вычисление log_10(2) * log_2(x)
    format %{ "fldlg2\t\t\t#Log10\n\t"
              "fyl2x\t\t\t# Q=Log10*Log_2(x)\n\t"
        %}
    ins_encode(Opcode(0xD9), Opcode(0xEC),
               Push_SrcXD(dst),

               Opcode(0xD9), Opcode(0xF1),
```

```
            Push_ResultXD(dst));
    ins_pipe( pipe_slow );
%}
```

Если исходный код фундаментального Java-метода выглядит неоптимальным, проверьте, действительно ли виртуальная машина имеет зависимую от платформы реализацию с использованием встроенных операций.

Рассматривая добавление новой встроенной операции, необходимо оценить компромисс между дополнительной сложностью и вероятностью пользы встроенной операции.

Например, можно представить себе встроенные операции, которые выполняют основные арифметические вычисления, такие как сумма первых `n` чисел. Традиционный код Java требует O(*n*) операций для вычисления этого значения, но имеется тривиальная формула, по которой это значение вычисляется за время O(1).

Должны ли мы реализовать встроенную операцию для вычисления суммы за константное время? Ответ зависит от того, сколько классов, наблюдаемых в "дикой природе", хотят вычислять такую сумму — в данном случае их явно будет мало. Такая встроенная операция явно будет обладать ограниченной пользой и почти наверняка не стоит дополнительной сложности в JVM.

Это подчеркивает тот факт, что встроенные операции могут оказывать существенное влияние на производительность только для тех операций, которые действительно часто встречаются в реальном коде.

Замена на стеке

Иногда приходится сталкиваться с кодом, который содержит горячий цикл в методе, который вызывается недостаточно часто для запуска компиляции, например в методе `main()` программы Java.

Тем не менее HotSpot может оптимизировать этот код, используя методику, именуемую *заменой на стеке* (on-stack replacement — OSR). Этот трюк подсчитывает обратные ветви цикла в интерпретаторе; когда они пересекут пороговое значение, интерпретируемый цикл будет скомпилирован, и выполнение переключится на эту скомпилированную версию.

Компилятор отвечает за то, чтобы любые изменения состояния, такие как локальные переменные и блокировки, которые были доступны в интерпретируемом цикле, стали доступными и для скомпилированной версии. После выхода из скомпилированного цикла все изменения состояния должны быть видимыми в точке, в которой продолжается выполнение.

Например, у нас есть такой горячий цикл внутри метода `main()`:

```
package optjava;
public class OnStackReplacement
{
    // Однократно вызываемый метод
    public static void main(String[] args)
    {
        java.util.Random r = new java.util.Random();
        long sum = 0;
        // Первый "долгоиграющий" цикл
        for (int i = 0; i < 1_000_000; i++)
        {
            sum += r.nextInt(100);
        }
        // Второй "долгоиграющий" цикл
        for (int i = 0; i < 1_000_000; i++)
        {
            sum += r.nextInt(100);
        }
        System.out.println(sum);
    }
}
```

Тогда байт-код этого метода выглядит примерно следующим образом:

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=5, args_size=1
     0: new           #2 // class java/util/Random
     3: dup
     4: invokespecial #3 // Метод java/util/Random."<init>":()V
     7: astore_1
     8: lconst_0
     9: lstore_2
    10: iconst_0
    11: istore         4
    13: iload          4
    15: ldc           #4 // int 1000000
    17: if_icmpge     36
    20: lload_2
    21: aload_1
    22: bipush 100
    24: invokevirtual #5 // Метод java/util/Random.nextInt:(I)I
    27: i2l
    28: ladd
    29: lstore_2
    30: iinc           4, 1
```

```
33: goto          13
36: getstatic     #6 // Поле java/lang/System.out:Ljava/io/PrintStream;
39: lload_2
40: invokevirtual #7 // Метод java/io/PrintStream.println:(J)V
43: return
```

Байт-код `goto` у индекса 33 возвращает поток управления к проверке условия цикла в индексе 13.

Обратное ветвление выполняется, когда цикл достигает конца своего тела, проверяет его условие выхода и — если цикл не завершен — возвращается обратно к началу цикла.

HotSpot может выполнять компиляцию OSR с использованием как JIT-компилятора C1, так и JIT-компилятора C2.

JITWatch может показать как в байт-коде, так и в исходном коде, какой цикл был скомпилирован с использованием OSR, как показано на рис. 10.9.

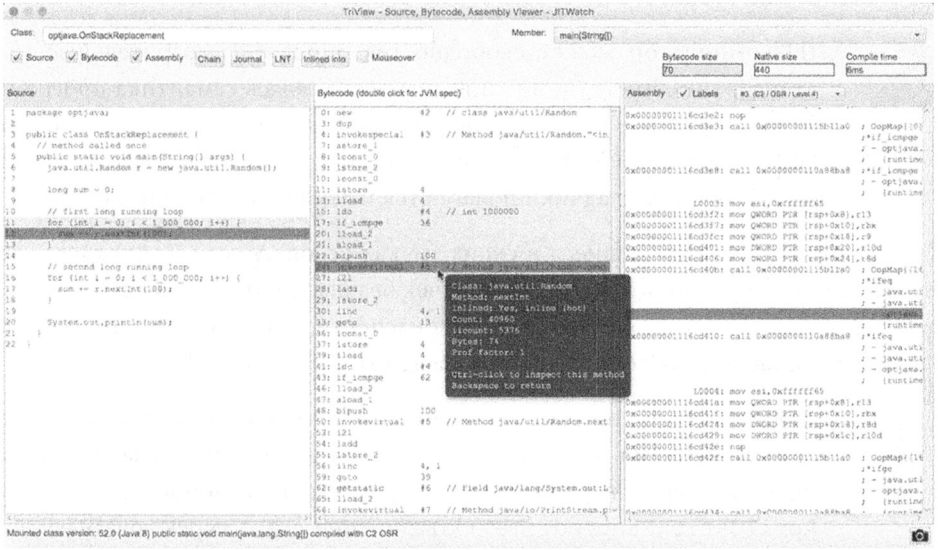

Рис. 10.9. Замена на стеке

Еще раз о точках безопасности

Прежде чем мы оставим тему JIT-компиляции, имеет смысл объединить в список все условия в JVM, которые требуют, чтобы виртуальная машина находилась в точке безопасности. Помимо событий сборки мусора с приостановкой приложения, требование, чтобы все потоки находились в точке безопасности, выдвигают следующие действия:

- деоптимизация метода;
- создание дампа кучи;
- отмена неверной блокировки;
- переопределение класса (например, для работы инструментария).

За генерацию проверок точек безопасности в скомпилированном коде отвечает JIT-компилятор (как ранее при разворачивании цикла), и в HotSpot он будет их генерировать:

- в обратных ветвях циклов;
- при возврате из методов.

Это означает, что иногда потоки могут потребовать определенного количества времени для достижения точки безопасности (например, если они выполняют цикл, содержащий большое количество арифметического кода без вызовов методов). Если цикл развернут, может пройти значительное время до того момента, когда будет достигнута точка безопасности.

JIT-компилятор может свободно генерировать умозрительные и нестандартные инструкции, лишь бы сохранялась семантика программы. Когда виртуальная машина достигнет точки безопасности, состояние скомпилированного кода будет соответствовать состоянию программы в этой точке. Отладчики полагаются именно на такое поведение.

Когда потоки, уже находящиеся в своих точках безопасности, вынуждены ожидать достижения другими потоками своих точек безопасности, компилятор пытается сбалансировать стоимость опроса точек безопасности, избегая длительных промежутков времени до достижения точки безопасности (Time to SafePoint — TTSP).

Увидеть общее время, затраченное в точках безопасности, включая время ожидания, когда все потоки достигнут точек безопасности, можно с помощью переключателя виртуальной машины `-XX:+PrintGC ApplicationStoppedTime`. Объединение его с переключателем `-XX: +PrintSafepointStatistics` дает дополнительную информацию о точках безопасности.

Мы вновь вернемся к точкам безопасности в главе 13, "Профилирование", так как они являются важной проблемой, оказывающей влияние на многие подсистемы JVM.

Методы базовой библиотеки

Чтобы завершить эту главу, бегло рассмотрим некоторые аспекты влияния размеров методов базовой библиотеки JDK на JIT-компиляцию.

Верхний предел размера метода для встраивания

Поскольку решения о встраивании принимаются на основе размера байт-кода метода, можно определить методы, которые слишком велики для встраивания, используя статический анализ файлов классов.

Инструмент JarScan с открытым исходным кодом (часть дистрибутива JITWatch — его сценарии запуска находятся в корневой папке JITWatch) может идентифицировать все методы в папке класса или JAR-файле, размер байт-кода которых превышает заданный порог.

Запуск этого инструмента для базовых библиотек Java 8, найденных в `jre/lib/rt.jar`, с помощью такой команды, как показано ниже, дает некоторые интересные результаты:

```
$ ./jarScan.sh --mode=maxMethodSize \
               --limit=325 \
               --packages=java.* \
               /path/to/java/jre/lib/rt.jar
```

В Java 8u152, на Linux x86_64, в дереве пакетов `java.*` имеется 490 методов размером более 325 байт байт-кода (предел `FreqInlineSize` для этой платформы), и некоторые из них — это методы, встретив которые в горячем коде, вы не удивились бы.

Например, методы `toUpperCase()` и `toLowerCase()` из класса `java.lang.String` оказываются удивительно большими — по 439 байт байт-кода каждый (вне обычного диапазона для встраивания).

Причина такого большого размера заключается в том, что в некоторых локалях преобразование символа изменяет количество значений типа `char`, необходимых для его хранения. Поэтому методы преобразования регистра символов должны быть в состоянии обнаружить такую ситуацию, изменить размер и копировать массивы строки, как показано в реализации по умолчанию метода `toUpperCase()`:

```
public String toUpperCase(Locale locale) {
    if (locale == null) {
        throw new NullPointerException();
    }
    int firstLower;
    final int len = value.length;
    /* Проверка наличия символов, требующих преобразования. */
    scan: {
        for (firstLower = 0 ; firstLower < len; ) {
            int c = (int)value[firstLower];
            int srcCount;
            if ((c >= Character.MIN_HIGH_SURROGATE)
                    && (c <= Character.MAX_HIGH_SURROGATE)) {
                c = codePointAt(firstLower);
```

```
            srcCount = Character.charCount(c);
        } else {
            srcCount = 1;
        }
        int upperCaseChar = Character.toUpperCaseEx(c);
        if ((upperCaseChar == Character.ERROR)
                || (c != upperCaseChar)) {
            break scan;
        }
        firstLower += srcCount;
    }
    return this;
}

/* Результат может вырасти, так что позицией
   записи является i+resultOffset */
int resultOffset = 0;
char[] result = new char[len]; /* Может расти */

/* Просто копируем несколько первых символов upperCase. */
System.arraycopy(value, 0, result, 0, firstLower);
String lang = locale.getLanguage();
boolean localeDependent =
    (lang == "tr" || lang == "az" || lang == "lt");
char[] upperCharArray;
int upperChar;
int srcChar;
int srcCount;
for (int i = firstLower; i < len; i += srcCount) {
    srcChar = (int)value[i];
    if ((char)srcChar >= Character.MIN_HIGH_SURROGATE &&
            (char)srcChar <= Character.MAX_HIGH_SURROGATE) {
        srcChar = codePointAt(i);
        srcCount = Character.charCount(srcChar);
    } else {
        srcCount = 1;
    }
    if (localeDependent) {
        upperChar = ConditionalSpecialCasing.
                    toUpperCaseEx(this, i, locale);
    } else {
        upperChar = Character.toUpperCaseEx(srcChar);
    }
    if ((upperChar == Character.ERROR)
            || (upperChar >=
                Character.MIN_SUPPLEMENTARY_CODE_POINT)) {
        if (upperChar == Character.ERROR) {
            if (localeDependent) {
```

```
                    upperCharArray =
                        ConditionalSpecialCasing.
                        toUpperCaseCharArray(this, i, locale);
                } else {
                    upperCharArray =
                            Character.toUpperCaseCharArray(srcChar);
                }
            } else if (srcCount == 2) {
                resultOffset +=
                        Character.toChars(upperChar,
                                          result, i + resultOffset)
                            - srcCount;
                continue;
            } else {
                upperCharArray = Character.toChars(upperChar);
            }
            /* Grow result if needed */
            int mapLen = upperCharArray.length;
            if (mapLen > srcCount) {
                char[] result2 =
                        new char[result.length + mapLen - srcCount];
                System.arraycopy(result, 0, result2,
                                 0, i + resultOffset);
                result = result2;
            }
            for (int x = 0; x < mapLen; ++x) {
                result[i + resultOffset + x] = upperCharArray[x];
            }
            resultOffset += (mapLen - srcCount);
        } else {
            result[i + resultOffset] = (char)upperChar;
        }
    }
    return new String(result, 0, len + resultOffset);
}
```

Повышение производительности с помощью методов предметной области

Если вы можете быть уверены, что в вашей предметной области набор входных символов не требует такой гибкости (возможно, все ваши входные символы находятся в пределах ASCII), то вы можете создать версию `toUpperCase()`, специфичную для вашей предметной области, байт-код которой легко уложится в пределы ограничений встраивания.

Приведенная далее реализация, предназначенная для ASCII, состоит всего из 69 байт байт-кода:

```
package optjava.jmh;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;
@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class DomainSpecificUpperCase {

    private static final String SOURCE =
        "The quick brown fox jumps over the lazy dog";

    public String toUpperCaseASCII(String source) {
        int len = source.length();
        char[] result = new char[len];
        for (int i = 0; i < len; i++) {
            char c = source.charAt(i);
            if (c >= 'a' && c <= 'z') {
                c -= 32;
            }
            result[i] = c;
        }
        return new String(result);
    }

    @Benchmark
    public String testStringToUpperCase() {
        return SOURCE.toUpperCase();
    }

    @Benchmark
    public String testCustomToUpperCase() {
        return toUpperCaseASCII(SOURCE);
    }
}
```

Сравнение производительности пользовательской реализации с методом `String.toUpperCase()` базовой библиотеки показывает следующие результаты:

```
Benchmark                                    Mode Cnt    Score   Error Units
DomainS...UpperCase.testCustomToUpperCase thrpt 200 20138368 ± 17807 ops/s
DomainS...UpperCase.testStringToUpperCase thrpt 200  8350400 ±  7199 ops/s
```

В этом тесте версия предметной области выполняла приблизительно в 2,4 раза больше операций в секунду по сравнению с версией базовой библиотеки.

Преимущества малых методов

Другим преимуществом поддержки малых размеров методов является то, что это увеличивает количество перестановок встраивания. Изменение данных во время выполнения может приводить к тому, что горячими будут становиться разные пути.

Сохраняя методы малыми, мы можем создавать различные встраиваемые деревья, чтобы оптимизировать большее количество этих горячих путей. При использовании более крупных методов ограничение на размеры может быть достигнуто раньше, что оставит некоторые пути неоптимизированными.

Верхний предел размера метода для компиляции

В HotSpot есть еще один предел, который мы сейчас продемонстрируем. Это размер байт-кода, при превышении которого методы не будут скомпилированы: 8000 байт. В производственной JVM этот предел не может быть изменен, но если вы используете отладочную JVM, то можете использовать переключатель `-XX:HugeMethodLimit=<n>` для установки максимального размера байт-кода компилируемого метода.

Найти эти методы в базовых библиотеках JDK можно с помощью JarScan следующим образом:

```
./jarScan.sh --mode=maxMethodSize --limit=8000 /path/to/java/jre/lib/rt.jar
```

Результаты показаны в табл. 10.4.

Таблица 10.4. Методы базовой библиотеки слишком большого размера

Метод	Размер (байт)
`javax.swing.plaf.nimbus.NimbusDefaults.initializeDefaults()`	23 103
`sun.util.resources.LocaleNames.getContents()`	22 832
`sun.util.resources.TimeZoneNames.getContents()`	17 818
`com.sun.org.apache.xalan.internal.xsltc.compiler.CUP$XPathParser$actions.CUP$XPathParser$do_action()`	17 441
`javax.swing.plaf.basic BasicLookAndFeel.initComponentDefaults()`	15 361
`com.sun.java.swing.plaf.windows.WindowsLookAndFeel.initComponentDefaults()`	14 060
`javax.swing.plaf.metal.MetalLookAndFeel.initComponentDefaults()`	12 087
`com.sun.java.swing.plaf.motif.MotifLookAndFeel.initComponentDefaults()`	11 759
`com.sun.java.swing.plaf.gtk.GTKLookAndFeel.initComponentDefaults()`	10 921

Метод	Размер (байт)
`sun.util.resources.CurrencyNames.getContents()`	8578
`javax.management.remote.rmi._RMIConnectionImpl_Tie()`	8152
`org.omg.stub.javax.management.remote.rmi._RMIConnectionImpl_Tie()`	8152

Ни один из этих огромных методов, скорее всего, не будет найден в горячем коде. Они в основном являются инициализаторами подсистемы пользовательского интерфейса или предоставляют ресурсы для списков валют, стран или имен локалей.

Чтобы продемонстрировать эффект от потери JIT-компиляции, мы рассмотрим результаты тестов двух почти идентичных методов: один — с размером меньше, а второй — больше значения `HugeMethodLimit`:

```
private java.util.Random r = new java.util.Random();
@Benchmark
public long lessThan8000()
{
    return r.nextInt() +
    r.nextInt() +
    ... // До общего размера метода чуть меньше 8000 байт байт-кода
}
@Benchmark
public long moreThan8000()
{
    return r.nextInt() +
    r.nextInt() +
    ... // До общего размера метода чуть больше 8000 байт байт-кода
}
```

Мы получим следующий результат:

```
Benchmark                 Mode Cnt     Score     Error Units
HugeMethod.lessThan8000 thrpt 100 89550.631 ±  77.703 ops/s
HugeMethod.moreThan8000 thrpt 100 44429.392 ± 102.076 ops/s
```

Метод `moreThan8000()` не был JIT-скомпилирован и работал примерно с половиной скорости JIT-скомпилированного метода `lessThan8000()`. Существует множество причин не создавать такие огромные методы (например, из соображений удобочитаемости, обслуживания, отладки), но вот вам еще одна.

Реальная ситуация, когда могут быть созданы огромные методы, — автоматически генерируемый код. Некоторые программы автоматически генерируют код для представления запросов к хранилищам данных. Затем код запроса компилируется, с тем чтобы его можно было выполнить на JVM с высокой производительностью. Если запрос достаточно сложен, его размер может выйти за описанный предел HotSpot,

поэтому может быть полезна проверка размеров методов с помощью такого инструмента, как JarScan.

Имеется много параметров настроек JIT-компилятора, но какой бы из них вы ни меняли, всегда следует оценивать свою систему до и после изменений, так как могут возникнуть неожиданные побочные эффекты, связанные с пространством кеша кода, длиной очереди JIT-компилятора и даже нагрузкой сборки мусора.

Резюме

В этой главе мы рассмотрели основы подсистемы JIT-компиляции HotSpot и некоторые из ее оптимизаций. Мы рассмотрели параметры, которые можно использовать для настройки JIT-компиляторов, и изучили некоторые из получающихся результатов компиляции.

С точки зрения практических методов мы можем использовать флаг `-XX:+PrintCompilation` для подтверждения оптимизации отдельных методов. Общий принцип "сначала писать хороший код, а оптимизировать только тогда, когда это необходимо", определенно, применим и здесь. Знание ограничений встраивания и других порогов компиляции позволяет разработчикам выполнять рефакторинг, оставаясь в этих пределах (или, в редких случаях, изменять пороговые значения). Понимание существования мономорфной диспетчеризации и точного определения типов означает, что приложения могут быть написаны по классическому принципу: проектирование интерфейсов, даже если у интерфейса имеется только одна реализация. Это наилучший из подходов: класс может все еще развиваться и тестироваться, но наличие единственной реализации сохраняет мономорфную диспетчеризацию.

ГЛАВА 11

Языковые методы повышения производительности

До сих пор мы изучали механизмы, с помощью которых JVM преобразует код, написанный разработчиком, в байт-код и оптимизирует его до высокопроизводительного скомпилированного кода.

Было бы замечательно, если бы каждое приложение Java состояло из высококачественного кода, спроектированного с учетом производительности. Однако реальность часто бывает совсем иной. Тем не менее во многих случаях JVM может принимать субоптимальный код и заставлять его работать с высокой производительностью, что является свидетельством мощности и надежности среды.

Даже приложения с огромной базой кода, которые очень трудно поддерживать, зачастую могут вполне адекватно работать в промышленной среде. Конечно, никому не хочется их поддерживать или модифицировать. Что же остается разработчику, когда необходимо настроить такое приложение для получения высокой производительности?

После внешних факторов, таких как сетевое подключение, ввод-вывод и базы данных, одним из самых вероятных потенциальных узких мест с точки зрения производительности является дизайн кода. Невероятно трудно спроектировать код правильно, и ни один дизайн не идеален.

Оптимизация Java — очень сложная задача, и в главе 13, "Профилирование", мы рассмотрим, как средства профилирования могут помочь в поиске кода, который не работает как должно.

Несмотря на это, имеется ряд базовых аспектов кода, о которых постоянно должен помнить разработчик, заботящийся о производительности своих приложений. Например, невероятно важно, как данные хранятся в приложении. По мере изменения бизнес-требований может возникнуть необходимость в изменении представления данных. Чтобы понимать, какие варианты хранения данных доступны

разработчику, следует ознакомиться со структурами данных, доступными в API коллекций Java, и деталями их реализации.

Выбор структуры данных без точного знания о том, как она будет обновляться и запрашиваться, опасен. Хотя некоторые разработчики имеют своих "любимчиков" среди классов Java и используют их, не задумываясь о результатах, добросовестный разработчик должен тщательно рассмотреть, как будут запрашиваться данные и какие алгоритмы будут наиболее эффективными для этой цели. Существует множество алгоритмов и операций, которые `java.lang.Collections` предоставляет в виде статических методов.

Прежде чем реализовывать распространенные алгоритмы (например, вручную писать алгоритм пузырьковой сортировки) для использования в своем рабочем коде, проверьте, нет ли готовых алгоритмов в `java.lang.Collections`, которые могут быть вами использованы.

Понимание объектов предметной области и времени их жизни в системе может существенно повлиять на производительность. Есть несколько эвристик, которые стоит рассмотреть, а способ использования объектов предметной области в приложении может влиять на сборку мусора и добавлять излишние накладные расходы времени выполнения.

В этой главе мы рассмотрим каждую из упомянутых проблем, начиная с того, что разработчик должен знать о коллекциях.

Оптимизация коллекций

Большинство библиотек языков программирования предоставляют как минимум два общих типа контейнеров.

- *Последовательные контейнеры* (sequential containers) хранят объекты в определенных позициях, описываемых с помощью числовых индексов.
- *Ассоциативные контейнеры* (associative containers) используют сами объекты для выяснения того, где они должны храниться в коллекции.

Для того чтобы определенные методы контейнера работали правильно, хранящиеся объекты должны иметь концепцию сравнимости и эквивалентности. В базовом API Java Collections это обычно выражается утверждением, что объекты должны реализовывать `hashCode()` и `equals()` в соответствии с контрактом, описанным Джошуа Блохом (Josh Bloch) в его книге *Java: эффективное программирование*[1].

[1] Джошуа Блох, *Java: эффективное программирование*, 3-е изд. — М.: Диалектика, 2019.

Как мы видели в разделе "Введение в среду времени выполнения HotSpot" главы 6, "Сборка мусора", поля ссылочного типа хранятся в куче в виде ссылок. Следовательно, хотя мы говорим об объектах, которые хранятся последовательно, на самом деле это не сами объекты хранятся в контейнере, а ссылки на них. Это означает, что обычно вы не сможете получить такую же производительность, как при использовании массива или вектора C/C++.

Это пример того, как подсистема управляемой памяти Java требует от вас отказаться от низкоуровневого управления памятью в обмен на автоматическую сборку мусора. Обычный пример отказа от низкоуровневого управления — это ручное управление выделением и освобождением памяти, но здесь мы наблюдаем отказ от низкоуровневого управления памятью.

Еще одно замечание, относящееся к этой проблеме, — что Java, по крайней мере в настоящий момент, не имеет возможности предоставить эквивалент массива структур языка C.

Джил Тен (Gil Tene) из Azul Systems часто говорит об этом ограничении как об одном из последних больших барьеров производительности между Java и C. На вебсайте ObjectLayout[2] имеется дополнительная информация о том, как можно стандартизировать схему памяти, и приведенный там код будет работать и компилироваться на Java 7 и выше. Его цель состоит в том, чтобы оптимизированная JVM могла принимать такие типы и корректно размещать структуры с описанием семантики, не нарушающей совместимости с другими JVM.

В главе 15, "Java 9 и будущие версии", мы обсудим будущее среды Java и опишем усилия по внесению в платформу *типов-значений* (третий тип, который будет существовать вместе с примитивами и объектными типами). Это обеспечит значительно большие возможности, чем в случае кода с размещением объектов.

API коллекций определяет набор интерфейсов, указывающий операции, которые должны быть у контейнера того или иного типа. На рис. 11.1 показана схема основных типов контейнеров.

В дополнение к интерфейсам имеется несколько реализаций коллекций, доступных внутри JDK. Выбор правильной реализации на основе дизайна является важной частью проблемы, но нам также необходимо понимать, что наш выбор может повлиять на общую производительность приложения.

[2] `http://objectlayout.org/`

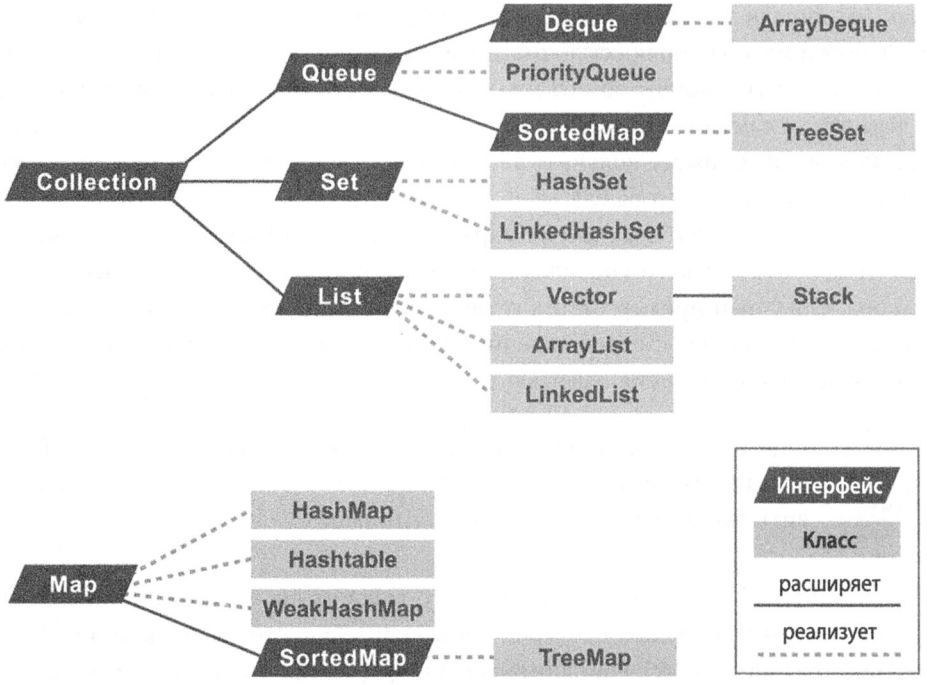

Рис. 11.1. Иерархия классов Java Collections API

Вопросы оптимизации списков

В ядре Java есть в два основных варианта представления списков — `ArrayList` и `LinkedList`.

Хотя Java также имеет классы `Stack` и `Vector`, первый добавляет дополнительную семантику, которая обычно излишня, а второй сильно устарел. Ваш код не должен использовать `Vector`, и вы должны выполнить рефакторинг таким образом, чтобы удалить из приложения любое его использование.

Поговорим о каждом из этих вариантов, начав с `ArrayList`.

`ArrayList`

`ArrayList` поддерживается с помощью массива с фиксированным размером. Записи могут добавляться в массив до достижения максимального размера массива поддержки. Когда массив заполнен, класс выделяет новый массив большего размера и копирует в него старые значения. Таким образом, программист, заботящийся о производительности, должен при выборе взвешивать стоимость операции

изменения размера и гибкость, позволяющую не знать заранее, насколько большим будет размер списка. `ArrayList` изначально поддерживается пустым массивом. При первом добавлении в список выделяется массив емкостью 10 элементов. Мы можем предотвратить изменение размера путем передачи конструктору необходимого значения начальной емкости. Мы также можем использовать `securityCapacity()` для увеличения емкости `ArrayList`, позволяющего избежать изменения размера.

Задавать емкость всякий раз, когда это возможно, — разумное решение, позволяющее избежать расходов на операции изменения размера.

Далее приведен микротест в JMH (см. раздел "Введение в JMH" главы 5, "Микротесты и статистика"), который демонстрирует описанный эффект:

```
@Benchmark
public List<String> properlySizedArrayList() {
    List<String> list = new ArrayList<>(1_000_000);
    for(int i=0; i < 1_000_000; i++) {
        list.add(item);
    }
    return list;
}

@Benchmark
public List<String> resizingArrayList() {
    List<String> list = new ArrayList<>();
    for(int i=0; i < 1_000_000; i++) {
        list.add(item);
    }
    return list;
}
```

Микротесты в этом разделе предназначены для иллюстрации, а не в качестве официальной рекомендации. Если ваши приложения действительно оказываются чувствительными к влиянию операций такого типа на производительность, вам следует рассмотреть альтернативы стандартным коллекциям.

Вот как выглядит результат микротеста:

```
Benchmark                              Mode Cnt   Score   Error Units
ResizingList.properlySizedArrayList thrpt  10 287.388 ± 7.135 ops/s
ResizingList.resizingArrayList      thrpt  10 189.510 ± 4.530 ops/s
```

Тест `properlySizedArrayList` может выполнять около 100 дополнительных операций в секунду, когда речь идет о вставке, несмотря на амортизацию стоимости перераспределения памяти. Всегда лучше, когда это возможно, указывать правильный размер `ArrayList`.

LinkedList

`LinkedList` имеет более динамичную схему роста; он реализуется как дважды связанный список, и это (среди прочего) означает, что добавление к списку всегда будет выполняться за константное время O(1). Каждый раз, когда элемент добавляется в список, создается новый узел, на который ссылается предыдущий элемент. Пример данной структуры данных показан на рис. 11.2.

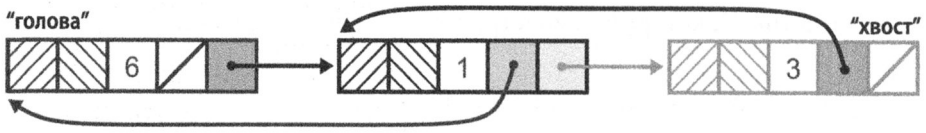

Рис. 11.2. `LinkedList`

Сравнение ArrayList и LinkedList

Реальное решение о том, что следует использовать (`ArrayList`, `LinkedList` или другую нестандартную реализацию списка), зависит от схемы обращения и изменения данных. Вставка в конец списка как в `ArrayList`, так и в `LinkedList` представляет собой операцию с константным временем работы (в случае `ArrayList` — с амортизированным константным временем работы).

Однако добавление в `ArrayList` по индексу требует, чтобы все остальные элементы были сдвинуты на одну позицию вправо. С другой стороны, `LinkedList` вынужден обходить все ссылки на узлы, чтобы найти позицию в списке, куда требуется вставка (но сама вставка включает в себя только создание нового узла и настройку двух ссылок, одна из которых указывает на предыдущий узел, а другая — на следующий узел в списке. Приведенный далее тест показывает разницу в производительности вставки в начало каждого типа списка:

```
Benchmark                      Mode Cnt   Score    Error Units
InsertBegin.beginArrayList    thrpt  10   3.402 ±  0.239 ops/ms
InsertBegin.beginLinkedList   thrpt  10 559.570 ± 68.629 ops/ms
```

Удаление из списка ведет себя аналогично; дешевле удалить элемент из `LinkedList`, так как при этом необходимо изменить не более двух ссылок. В `ArrayList` все элементы справа от удаляемого должны сдвинуться на одну позицию влево.

Если список в основном используется для случайных обращений, `ArrayList` является лучшим выбором, поскольку обращение к любому его элементу выполняется

за время O(1), тогда как LinkedList требует прохода от начала списка до элемента с указанным индексом. Стоимость обращения по конкретному индексу к элементам различных типов списков показана в следующих результатах теста:

```
Benchmark                          Mode Cnt      Score       Error Units
AccessingList.accessArrayList     thrpt  10 269568.627 ± 12972.927 ops/ms
AccessingList.accessLinkedList    thrpt  10      0.863 ±     0.030 ops/ms
```

В общем случае рекомендуется использовать ArrayList (если только вам не требуется конкретное поведение LinkedList), в особенности если вы используете алгоритм, требующий произвольного доступа к элементам списка. Если это возможно, установите емкость ArrayList заблаговременно, чтобы избежать изменений размера во время работы. Коллекции в современном Java придерживаются того взгляда, что затраты на синхронизацию для всех обращений должен нести разработчик, и он должен либо использовать параллельные коллекции, либо вручную управлять синхронизацией, когда это необходимо. Во вспомогательном классе Collections имеется метод synchronizedList(), который фактически является декоратором, заворачивающим все вызовы методов List в блок synchronized. В главе 12, "Методы повышения производительности параллельной работы", мы более подробно обсудим, как использовать java.util.concurrent для структур, которые можно более эффективно использовать при написании многопоточных приложений.

Вопросы оптимизации отображений

Отображение в общем случае описывает взаимосвязь между ключом и ассоциированным с ним значением (откуда и альтернативное название —*ассоциативный массив*). В Java отображения следуют интерфейсу java.util.Map<K,V>. Разумеется, как ключ, так и значение должны быть ссылочного типа.

HashMap

Во многих отношениях класс HashMap Java можно рассматривать как классическую хеш-таблицу из введения в курс компьютерных наук, но с несколькими дополнительными украшениями, чтобы сделать ее подходящей для современных сред.

Урезанная версия HashMap (с выброшенными обобщенными типами и парой ключевых возможностей, к которым мы вернемся позже) содержит некоторые ключевые методы, которые выглядят следующим образом:

```
public Object get(Object key)
{
    // УПРОЩЕНИЕ: нулевые ключи не поддерживаются
    if (key == null) return null;
    int hash = key.hashCode();
    int i = indexFor(hash, table.length);
```

```
    for (Entry e = table[i]; e != null; e = e.next)
    {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}

private int indexFor(int h, int length)
{
    return h & (length-1);
}

// Узел связанного списка
static class Node implements Map.Entry
{
    final int hash;
    final Object key;
    Object value;
    Node next;
    Node(int h, Object k, Object v, Entry n)
    {
        hash = h;
        key = k;
        value = v;
        next = n;
    }
}
```

В этом случае класс `HashMap.Node` имеет ограниченный доступ на уровне пакета `java.util`; таким образом, это классический вариант использования статического класса.

Схема `HashMap` показана на рис. 11.3.

Первоначально записи в слотах (блоках) будут сохранены в списке. Когда дело доходит до поиска значения, вычисляется хеш-значение ключа, а затем для поиска ключа в списке используется метод `equals()`. Из-за механизма хеширования ключа и использования равенства для его поиска в списке дублирование ключей в `HashMap` не разрешено. Вставка одного и того же ключа просто замещает ключ, который в настоящее время хранится в `HashMap`.

В современных версиях Java одним из улучшений является то, что `indexFor()` был заменен кодом, который использует `hashCode()` для объекта ключа и применяет маску для разброса старших битов значения хеша среди младших битов.

Этот способ был разработан как компромисс, гарантирующий, что `HashMap` учитывает старшие биты при вычислении слота, в который хешируется ключ. Если

бы это не было сделано, старшие биты могли бы не использоваться при вычислении индекса. По ряду причин это весьма проблематично, и не в последнюю очередь из-за того, что при этом нарушаются условия *Строгого лавинного критерия* (Strict Avalanche Criteria) Шеннона (Shannon) — идея, что даже произвольно малые изменения входных данных должны приводить к потенциально очень большим изменениям на выходе хеш-функции.

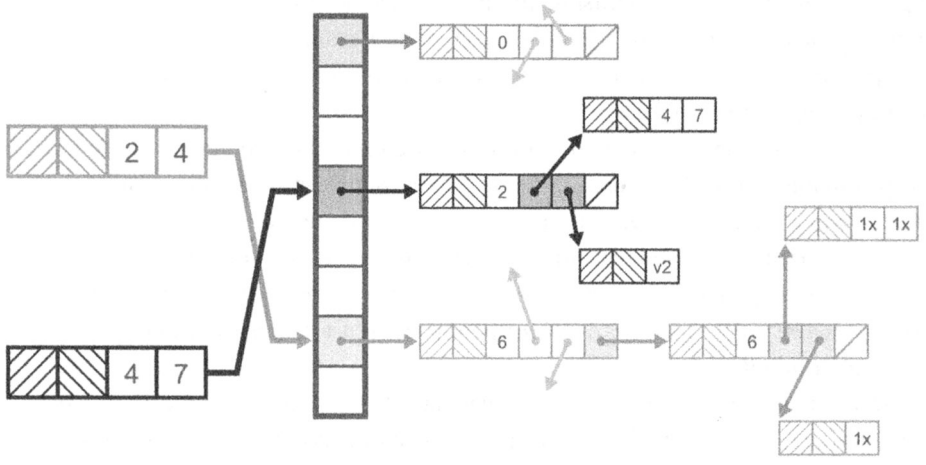

Рис. 11.3. Простая структура данных `HashMap`

На производительность `HashMap` влияют две важные переменные: `initialCapacity` и `loadFactor`; обе они могут быть установлены с помощью параметров, передаваемых конструктору. Емкость `HashMap` представляет собой текущее количество созданных слотов, по умолчанию равное 16. Значение `loadFactor` указывает, насколько может быть заполнена хеш-таблица до того, как ее емкость будет автоматически увеличена. Увеличение емкости и перерасчет хеш-значений известно как *перехеширование*, которое удваивает количество доступных слотов и перераспределяет хранящиеся данные.

Установка `initialCapacity` для `HashMap` следует тем же правилам, что и для `ArrayList`: если вы знаете, какой объем информации будет сохранен, вы должны его указать.

Точное значение `initialCapacity` позволит избежать необходимости автоматического перехеширования по мере роста таблицы. Можно также настроить `loadFactor`, но значение по умолчанию `0.75` обеспечивает хороший баланс между перерасходом памяти и временем доступа. Значение `loadFactor` выше `0.75` уменьшит количество перехеширований, но сделает доступ медленнее, так как слоты в общем случае станут более заполненными. Установка `initialCapacity` равным

максимальному количеству элементов, деленному на loadFactor, предотвратит выполнение перехеширования.

HashMap обеспечивает константное время выполнения операций get() и put(), но итерирование коллекции может оказаться дорогостоящим. Как упоминается в JavaDoc, установка параметра initialCapacity и loadFactor сильно влияет на производительность итерирования.

Другим фактором, влияющим на производительность, является процесс, называемый *одеревенением* (treeifying). Это относительно недавнее новшество — внутренняя деталь реализации HashMap, но она потенциально полезна для инженеров, отвечающих за производительность.

Рассмотрим случай, когда слот становится сильно заполненным. Если элементы слота реализованы как LinkedList, то обход, необходимый для поиска элемента, становится все более дорогостоящим по мере роста списка.

Чтобы противодействовать этому линейному эффекту, современные реализации HashMap используют новый механизм. Когда слот достигает порога, определяемого параметром TREEIFY_THRESHOLD, он преобразуется в корзину элементов TreeNode (и ведет себя подобно TreeMap).

Почему бы не сделать это с самого начала? Применение TreeNode примерно удваивает размер узла списка, так что повышается расход памяти. Хорошо распределяющая элементы хеш-функция редко приводит к необходимости использования TreeNode в слотах. Если это произойдет в вашем приложении, вам пора задуматься о пересмотре хеш-функции, а также параметров initialCapacity и loadFactor для HashMap.

Как и во всем остальном в области производительности, практические методы управляются компромиссами и прагматизмом, и вы должны принять аналогичный подход к анализу собственного кода — на основе упорных размышлений и экспериментальных данных.

LinkedHashMap

LinkedHashMap является подклассом HashMap, который поддерживает порядок вставки элементов, используя двусвязный список, объединяющий все элементы.

По умолчанию LinkedHashMap поддерживает порядок вставки, но можно переключить режим для фиксации порядка обращений. LinkedHashMap часто используется там, где для потребителя имеет значение порядок, и его применение обходится не так дорого, как использование TreeMap.

В большинстве случаев ни порядок вставки, ни порядок доступа не имеют значения для пользователей Map, поэтому должно быть относительно мало ситуаций, когда правильным выбором коллекции является LinkedHashMap.

TreeMap

TreeMap, по сути, представляет собой реализацию красно-черного дерева. Этот тип дерева является структурой бинарного дерева с дополнительными метаданными (цветом узла), которая пытается препятствовать разбалансировке дерева.

Для того чтобы узлы дерева (TreeNode) можно было рассматривать как упорядоченные, необходимо, чтобы ключи использовали компаратор, согласующийся с методом equals().

TreeMap невероятно полезен, когда требуется диапазон ключей, обеспечивая возможность быстрого доступа к подмножеству отображения. TreeMap можно также использовать для разделения данных от начала до некоторой точки или от точки до конца.

TreeMap обеспечивает производительность O(*log n*) для операций get(), put(), containsKey() и remove().

На практике большую часть времени HashMap соответствует требованиям Map, но давайте рассмотрим один пример — ситуацию, когда часть отображения нужно обработать с использованием потоков или лямбда-выражений. В этих случаях может иметь больший смысл использование реализации, которая понимает разбиение данных, например TreeMap.

Отсутствие MultiMap

Java не обеспечивает реализацию MultiMap (отображение, в котором для одного и того же ключа может быть несколько значений). Причина, указанная в документации, заключается в том, что такое требуется нечасто, и в большинстве случаев может быть реализовано как Map<K,List<V>>. Существует несколько реализаций MultiMap для Java с открытым исходным кодом.

Вопросы оптимизации множеств

В Java имеется три типа множеств, и для всех них соображения производительности, по сути, ничем не отличаются от таковых для Map.

Фактически, если начать с более внимательного изучения версии HashSet (урезанной для краткости), то становится очевидно, что она реализована через HashMap (или LinkedHashMap в случае LinkedHashSet):

```
public class HashSet<E> extends AbstractSet<E>
                        implements Set<E>, Serializable
{
    private transient HashMap<E,Object> map;
```

```
    // Фиктивное значение для связи с Object в поддерживающий Map
    private static final Object PRESENT = new Object();
    public HashSet()
    {
        map = new HashMap<>();
    }
    HashSet(int initialCapacity, float loadFactor, boolean dummy)
    {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
    public boolean add(E e)
    {
        return map.put(e, PRESENT)==null;
    }
}
```

Поведение `Set` состоит в том, чтобы не допускать повторяющихся значений, что в точности то же самое, что делает отображение со значениями ключей. В методе `add()` множество `HashSet` просто вставляет элемент `E` в качестве ключа в `HashMap` и использует фиктивный объект `PRESENT` в качестве значения. Накладные расходы такого решения минимальны, поскольку объект `PRESENT` создается только один раз, после чего везде используются ссылки на него. У `HashSet` есть второй (защищенный) конструктор, который принимает аргумент `LinkedHashMap`; он может использоваться для имитации соответствующего поведения при отслеживании порядка вставки. `HashSet` имеет константное (O(1)) время вставки, удаления и операции проверки наличия элемента во множестве; этот класс не поддерживает упорядочение элементов (если только он не используется как `LinkedHashSet`), а стоимость итерирования зависит от значений `initialCapacity` и `loadFactor`.

`TreeSet` реализуется так же, как ранее описанное дерево `TreeMap`. Использование `TreeSet` сохраняет естественное упорядочение ключей, определенное компаратором, что делает операции на основе диапазонов и итераций намного более эффективными в `TreeSet`. `TreeSet` гарантирует временную стоимость O(*log n*) для операций вставки, удаления и проверки наличия элемента в множестве и поддерживает упорядочение элементов. Итерирование и выборка подмножеств являются эффективными операциями, так что любые решения на основе диапазона или упорядочения будут лучше работать при использовании `TreeSet`.

Объекты предметной области

Объекты предметной области (domain objects) — это код, который выражает важные для ваших приложений бизнес-концепции. Примерами могут быть `Order`, `OrderItem` и `DeliverySchedule` (заказ, пункт заказа и расписание доставки) для

сайта электронной торговли. Обычно такие типы взаимосвязаны (так что `Order` имеет несколько связанных с ним экземпляров `OrderItem`), например:

```
public class Order {
    private final long id;
    private final List<OrderItem> items = new ArrayList<>();
    private DeliverySchedule schedule;
    public Order(long id) {
        this.id = id;
    }
    public DeliverySchedule getSchedule() {
        return schedule;
    }
    public void setSchedule(DeliverySchedule schedule) {
        this.schedule = schedule;
    }
    public List<OrderItem> getItems() {
        return items;
    }
    public long getId() {
        return id;
    }
}
public class OrderItem {
    private final long id;
    private final String description;
    private final double price;
    public OrderItem(long id, String description, double price) {
        this.id = id;
        this.description = description;
        this.price = price;
    }
    @Override
    public String toString() {
        return "OrderItem{" + "id=" + id + ", description=" +
               description + ", price=" + price + '}';
    }
}
public final class DeliverySchedule {
    private final LocalDate deliveryDate;
    private final String address;
    private final double deliveryCost;
    private DeliverySchedule(LocalDate deliveryDate, String address,
                             double deliveryCost) {
        this.deliveryDate = deliveryDate;
        this.address = address;
        this.deliveryCost = deliveryCost;
    }
```

```
    public static DeliverySchedule of(LocalDate deliveryDate, String address,
                                      double deliveryCost) {
        return new DeliverySchedule(deliveryDate, address, deliveryCost);
    }
    @Override
    public String toString() {
        return "DeliverySchedule{" + "deliveryDate=" + deliveryDate +
               ", address=" + address + ", deliveryCost=" + deliveryCost + '}';
    }
}
```

Между типами предметной области имеются отношения владения, как мы видим на рис. 11.4. Однако в конечном счете подавляющее большинство элементов данных в листьях графа объектов предметной области являются простыми типами данных, такими как строки, примитивы и объекты `LocalDateTime`.

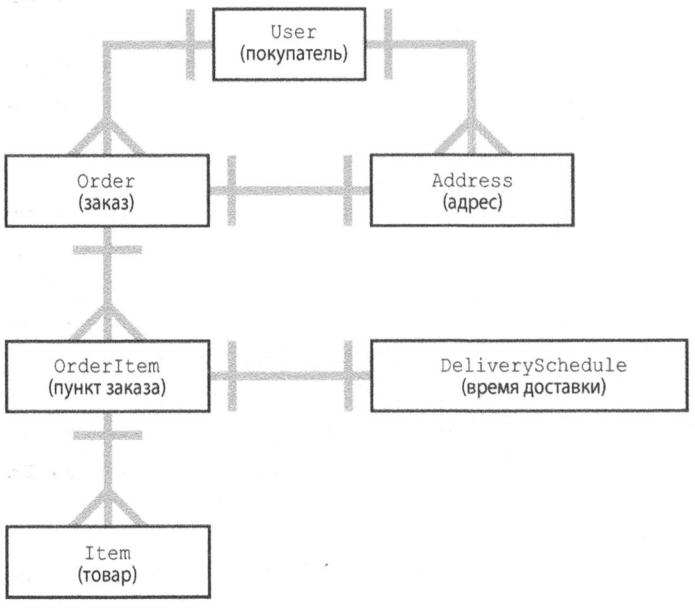

Рис. 11.4. Граф объектов предметной области

В разделе "Алгоритм маркировки и выметания" главы 6, "Сборка мусора", мы встречались с командой `jmap -histo`. Она обеспечивает быстрое представление о состоянии кучи Java; эквивалентная функциональность доступна посредством инструментария с графическим интерфейсом, такого как VisualVM. Стоит научиться использовать один из этих очень простых инструментов, так как они могут помочь диагностировать утечку памяти объектов предметной области в некоторых (ограниченных, но довольно распространенных) ситуациях.

Объекты предметной области в приложении имеют несколько особый статус. Поскольку в приложении они представляют интересы бизнеса первого порядка, они очень заметны, когда вы ищете ошибки, такие как утечки памяти.

Чтобы понять, почему это так, нам следует рассмотреть несколько основных фактов о куче Java.

- Структуры данных, для которых память выделяется наиболее часто, включают строки, массивы символов, массивы байтов и экземпляры типов коллекций Java.
- Данные, соответствующие утечке памяти, будут показаны в `jmap` как аномально большие наборы данных.

То есть мы ожидаем, что первые записи, касающиеся как объема памяти, так и количества экземпляров, будут обычно встречающимися структурами данных из ядра JDK. Если объекты предметной области, специфичные для приложений, оказываются среди первых 30 или около того записей, сгенерированных `jmap`, то это возможный (но не окончательный) признак утечки памяти.

Другим распространенным поведением для утечки объектов предметной области является эффект "всех поколений". Этот эффект связан с тем, что сборка (утилизация) объектов определенного типа не выполняется тогда, когда должна. Это означает, что в конечном итоге такие объекты будут жить достаточно долго, чтобы стать бессрочными, и будут показаны со всеми возможными значениями поколений после того, как они выживут в достаточном количестве сборок мусора.

Если мы построим гистограмму распределения байтов для каждого поколения (по типу данных), то увидим, что объекты предметной области с потенциальной утечкой будут отображаться во всех поколениях. Это может быть связано с тем, что они искусственно поддерживаются живыми за пределами их естественной продолжительности существования.

Один быстрый способ поиска источника проблем — посмотреть на размер рабочего множества данных, соответствующего объектам предметной области, и решить, насколько он разумен и в пределах ожидаемых границ количества объектов.

На другом конце спектра короткоживущие объекты предметной области могут стать причиной другой разновидности проблемы с *плавающим мусором*, с которой мы встречались ранее. Вспомните ограничение SATB для параллельных сборщиков мусора — идею о том, что любые объекты независимо от того, насколько они недолговечны, считаются живыми, если были выделены после запуска цикла маркировки.

Объекты предметной области могут служить своеобразной "канарейкой в угольной шахте" (по сути — чувствительным детектором неприятностей) для многих приложений. Поскольку они являются наиболее очевидными и естественными

представлениями бизнес-вопросов, похоже, они более восприимчивы к утечкам. Разработчики, ориентированные на эффективность, должны знать свою предметную область и размеры соответствующих рабочих множеств.

Объекты предметной области с утечкой иногда могут проявляться в качестве виновника длительного этапа маркировки сборки мусора. Основной причиной этого является то, что один долгоживущий объект удерживает в живом состоянии целую длинную цепочку объектов.

Избегайте финализации

Механизм `finalize()` в Java является попыткой обеспечить автоматическое управление ресурсами, аналогично идиоме *Захват ресурса есть инициализация* (Resource Acquisition Is Initialization — RAII) из C++. В этой идиоме имеется метод деструктора (известный в Java как `finalize()`), который включает в себя автоматическую очистку и освобождение ресурсов при уничтожении объекта.

Таким образом, основная схема работы механизма довольно проста. Когда объект создается, он получает в собственность какой-либо ресурс, и это владение ресурсом сохраняется на протяжении всего жизненного цикла объекта. Затем, когда объект уничтожается, владение ресурсом автоматически завершается.

Еще один способ описания этого механизма — *Автоматическое управление ресурсом* (Automatic Resource Management — ARM).

Стандартным примером такого подхода является наблюдение, что, после того как программист открыл файл, слишком легко забыть вызвать функцию его закрытия `close()`, когда файл больше не нужен.

Давайте посмотрим на быстрый и простой пример на C++, который показывает, как использовать RAII для создания оболочки вокруг файлового ввода-вывода в стиле C.

```
class file_error {};
class file {
public:
    file(const char* filename) : _h_file(std::fopen(filename, "w+"))
    {
        if (_h_file == NULL)
    {
            throw file_error();
        }
    }
    // Деструктор
```

```
    ~file()
    {
        std::fclose(_h_file);
    }
    void write(const char* str)
    {
        if (std::fputs(str, _h_file) == EOF)
        {
            throw file_error();
        }
    }
    void write(const char* buffer, std::size_t numc)
    {
        if (numc != 0 && std::fwrite(buffer, numc, 1, _h_file) == 0)
        {
            throw file_error();
        }
    }
private:
    std::FILE* _h_file;
};
```

Это способствует хорошему дизайну, особенно когда единственная причина существования типа — работа в качестве владельца ресурса, такого как файл или сетевой сокет. В этом случае привязка владения ресурсом ко времени жизни объекта имеет четкий смысл. Автоматическое избавление от ресурсов объекта становится ответственностью платформы, а не программиста.

История войны: забытая уборка

Как и многие истории в области программного обеспечения, эта история начинается с производственного кода, который отлично работал в течение многих лет. Это была служба, которая подключалась через TCP-соединение к другой службе для установления разрешений и прав доступа. Служба предоставления прав была относительно стабильной и имела хорошую балансировку нагрузки, обычно реагируя на запросы мгновенно. Для каждого запроса открывалось новое TCP-соединение, что было далековато от идеального проектирования.

Однажды было сделано изменение, приведшее к тому, что система стала отвечать немного медленнее. В результате TCP-соединение время от времени закрывалось из-за тайм-аута, т.е. выполнение шло по пути, с которым ранее в производственной работе приложения сталкиваться не приходилось. Было сгенерировано и перехвачено исключение `TimeOutException`, ничего не было запротоколировано в журнальный файл, как не было и блока `finally`. Ранее функция `close()` вызывалась на успешном пути выполнения.

К сожалению, проблемы на этом не закончились. Невызванная функция `close()` означает, что TCP-соединение оставлено открытым. В конце концов на машине, на которой работало приложение, закончились файловые дескрипторы, что повлияло на другие работающие на ней процессы. Резолюция заключалась в том, чтобы прежде всего переписать код так, чтобы TCP-соединение закрывалось сразу же, и передать этот исправленный вариант в работу; а затем следовало провести рефакторинг, выполнив объединение соединений в пул, чтобы не открывать новое соединение для каждого ресурса.

Забыть вызвать метод `close()` очень легко, в особенности при использовании библиотек сторонних производителей.

Почему не следует решать проблему путем финализации

В Java первоначально был предложен метод `finalize()`, который находится в `Object`; по умолчанию он представляет собой отсутствие каких-либо действий (и обычно он должен таким и оставаться). Однако можно перекрыть `finalize()` и обеспечить некоторое его поведение (какие-то действия). JavaDoc описывает это следующим образом.

> Вызывается сборщиком мусора для объекта, когда сборщик мусора определяет, что больше нет ссылок на этот объект. Подкласс перекрывает метод `finalize()` для утилизации системных ресурсов или для выполнения иной очистки.

Способ реализации финализации — использование сборщика мусора JVM в качестве подсистемы, которая может окончательно выяснить, что объект умер. Если для его типа задан метод `finalize()`, то все объекты этого типа рассматриваются специальным образом. Объект, который перекрывает `finalize()` обрабатывается сборщиком мусора особым образом. JVM реализует эту обработку путем регистрации отдельных финализируемых объектов при успешном возврате из тела конструктора `java.lang.Object` (который должен вызваться в некоторой точке на пути создания любого объекта).

Одна из деталей HotSpot, о которой нам нужно знать на данном этапе, состоит в том, что VM содержит специальные, специфичные для конкретной реализации байт-коды в дополнение к стандартным командам Java. Эти специальные байт-коды используются для перезаписи стандартных байт-кодов, чтобы справиться с некоторыми особыми обстоятельствами.

Полный список определений байт-кодов, как стандартных байт-кодов Java, так и байт-кодов HotSpot, можно найти в файле `hotspot/share/interpreter/bytecodes.cpp`. Нас интересует конкретно особый байт-код `return_register_finalizer`. Он необходим, потому что, например, JVMTI может переписать байт-код для `Object.<init>()`. Чтобы соответствовать стандарту, необходимо определить

точку, в которой `Object.<init>()` завершается (без перезаписи), и особый байт-код используется для того, чтобы отметить эту точку.

Код фактической регистрации объекта, который нуждается в финализации, можно увидеть в интерпретаторе HotSpot. Файл `src/hotspot/cpu/x86/c1_Runtime.cpp` содержит ядро интерпретатора HotSpot, специфичное для архитектуры x86. Ядро должно быть специфичным для конкретного процессора, потому что HotSpot обильно использует низкоуровневый ассемблер и машинный код. В `register_finalizer_id` содержится регистрационный код.

После того как объект был зарегистрирован как требующий финализации, вместо немедленного освобождения во время цикла сборки мусора объект следует расширенному жизненному циклу:

1. финализируемые объекты перемещаются в очередь;
2. после перезапуска потоков приложения отдельный поток финализации изымает объекты из очереди и выполняет метод `finalize()` для каждого объекта;
3. по окончании работы метода `finalize()` объект готов к реальной сборке мусора в следующем цикле.

В целом это означает, что все объекты, подлежащие финализации, сначала должны быть признаны недостижимыми с использованием маркировки сборки мусора, затем — финализированы, а затем сборка мусора должна при очередном запуске выполнить освобождение их памяти. Это означает, что финализируемые объекты сохраняются по крайней мере в течение одного дополнительного цикла сборки мусора. В случае ставших бессрочными объектов это может означать значительное количество времени. Обработку очереди финализации можно увидеть на рис. 11.5.

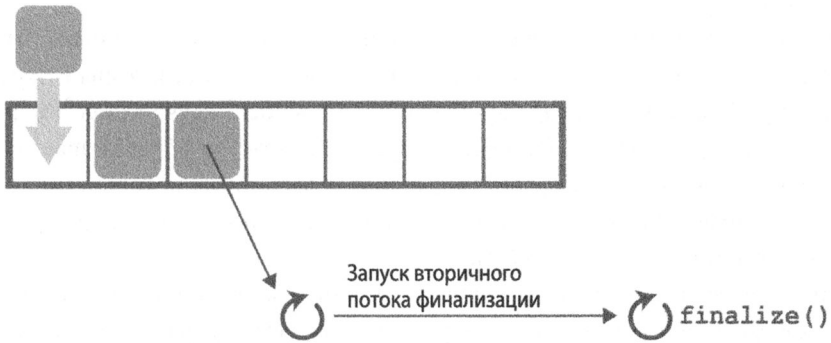

Рис. 11.5. Освобождение очереди финализации

Имеются и другие проблемы, связанные с `finalize()`; например, что произойдет, если метод выдаст исключение во время его выполнения потоком финализации? В этот момент в пользовательском коде приложения нет контекста, так что

исключение просто игнорируется. У разработчика нет возможности узнать о нем или выполнить некоторые действия по восстановлению после исключения.

Возможно также, что финализация может содержать блокирующую операцию и поэтому требует, чтобы JVM порождала поток для запуска метода `finalize()` с соответствующими накладными расходами на его создание и запуск. Опять же, создание потока и управление им находятся вне контроля разработчика, но при этом необходимо избегать блокировки всей подсистемы JVM.

Большая часть реализации финализации фактически выполняется на Java. JVM имеет отдельные потоки для выполнения финализации, которые работают одновременно с потоками приложений. Основная функциональность содержится в доступном на уровне пакета классе `java.lang.ref.Finalizer`, который должен быть достаточно прост для чтения.

Класс также дает некоторое представление о том, как определенные классы получают дополнительные привилегии во время выполнения. Например, он содержит такой код:

```
/* Вызов виртуальной машиной */
static void register(Object finalizee)
{
    new Finalizer(finalizee);
}
```

Конечно, в обычном коде приложения этот метод был бы бессмысленным, так как создавал бы неиспользуемый объект. Если только конструктор не имеет побочных эффектов (что обычно считается плохим проектным решением на Java), этот метод ничего не делает. В данном случае целью является "перехват" нового финализируемого объекта.

Реализация финализации также в значительной степени зависит от класса `FinalReference`. Это подкласс `java.lang.ref.Reference`, который представляет собой класс, известный среде выполнения, как особый случай. Как и хорошо известные мягкие и слабые ссылки, объекты `FinalReference` обрабатываются подсистемой сборки мусора особым образом, включая механизм, который обеспечивает интересное взаимодействие между виртуальной машиной и Java-кодом (как кодом платформы, так и пользовательским).

Однако при всей своей интересности с технической точки зрения реализация является смертельно ошибочной из-за различий в схемах управления памятью двух языков. В случае C++ динамическая память обрабатывается вручную, с явным управлением жизненным циклом объектов под управлением программиста. Это означает, что уничтожение может происходить при удалении объекта, и поэтому захват и освобождение ресурсов напрямую привязаны ко времени жизни объекта.

Подсистема управления памятью Java представляет собой сборщик мусора, который работает по мере необходимости, в ответ на нехватку доступной памяти для

выделения. Поэтому сборщик мусора работает с недетерминированными интервалами, а метод `finalize()` запускается только тогда, когда объект обработан сборщиком, — что произойдет в неизвестный момент времени.

Другими словами, финализация не позволяет безопасно реализовать автоматическое управление ресурсами, поскольку сборщик мусора не имеет никаких гарантий времени выполнения. Это означает, что в данном механизме нет ничего, что связывает освобождение ресурса со временем жизни объекта, так что всегда существует вероятность исчерпания ресурсов.

Финализация не подходит для первоначально намеченной основной цели. Рекомендации, предоставляемые разработчикам Oracle (и Sun) на протяжении многих лет, заключались в том, чтобы избегать финализации в коде обычных приложений; метод `Object.finalize()` в Java 9 объявлен устаревшим и не рекомендованным к употреблению.

try-с-ресурсами

До Java 7 ответственность за закрытие ресурса находилась исключительно в руках разработчика. Как сказано выше, в разделе "История войны: забытая уборка", его легко забыть и при этом не заметить влияние такой забывчивости до тех пор, пока проблема не будет выявлена при производственном применении приложения. В следующем образце кода показана ответственность разработчика до выхода Java 7:

```
public void readFirstLineOld(File file) throws IOException
{
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(file));
        String firstLine = reader.readLine();
        System.out.println(firstLine);
    } finally {
        if (reader != null)
        {
            reader.close();
        }
    }
}
```

Разработчик обязан:

1. создать объект типа `BufferedReader` и инициализировать его значением `null`, чтобы гарантировать его видимость в блоке `finally`;
2. сгенерировать или перехватить и обработать исключение `IOException` (а возможно, исключение `FileNotFoundException`, которое оно скрывает);

3. выполнить бизнес-логику взаимодействия с внешним ресурсом;
4. проверить, что значение `reader` не равно `null`, а затем закрыть открытый ресурс.

В этом примере используется только один внешний ресурс, но сложность резко возрастает при обработке нескольких внешних ресурсов. Если вам не хватает примеров, взгляните на необработанные вызовы JDBC.

Конструкция уровня языка `try`-с-ресурсами, добавленная в Java 7, позволяет создавать ресурс в круглых скобках после ключевого слова `try`. В скобках `try` может использоваться любой объект, реализующий интерфейс `AutoCloseable`. По завершении области видимости `try` метод `close()` будет вызываться автоматически, так что разработчик не обязан помнить о вызове этой функции. Приведенный далее вызов метода `close()` ведет себя так же, как и в предыдущем примере кода, и запускается независимо от сгенерированного в бизнес-логике исключения:

```
public void readFirstLineNew(File file) throws IOException
{
    try (BufferedReader reader =
                new BufferedReader(new FileReader(file)))
    {
        String firstLine = reader.readLine();
        System.out.println(firstLine);
    }
}
```

Используя `javap`, мы можем сравнить байт-код, генерируемый этими двумя версиями. Вот байт-код первого примера:

```
public void readFirstLineOld(java.io.File) throws java.io.IOException;
  Code:
    0: aconst_null
    1: astore_2
    2: new           #2 // Класс java/io/BufferedReader
    5: dup
    6: new           #3 // Класс java/io/FileReader
    9: dup
    10: aload_1
    11: invokespecial #4 // Метод java/io/FileReader."<init>":
                         // (Ljava/io/File;)V
    14: invokespecial #5 // Метод java/io/BufferedReader."<init>":
                         // (Ljava/io/Reader;)V
    17: astore_2
    18: aload_2
    19: invokevirtual #6 // Метод java/io/BufferedReader.readLine:
                         // ()Ljava/lang/String;
    22: astore_3
    23: getstatic     #7 // Поле
```

```
                    //java/lang/System.out:Ljava/io/PrintStream;
  26: aload_3
  27: invokevirtual #8 // Метод java/io/PrintStream.println:
                       // (Ljava/lang/String;)V
  30: aload_2
  31: ifnull        54
  34: aload_2
  35: invokevirtual #9 // Метод java/io/BufferedReader.close:()V
  38: goto          54
  41: astore         4
  43: aload_2
  44: ifnull        51
  47: aload_2
  48: invokevirtual #9 // Метод java/io/BufferedReader.close:()V
  51: aload          4
  53: athrow
  54: return
Таблица исключений:
  from   to   target   type
     2   30     41      any
    41   43     41      any
```

Эквивалентный байт-код из версии `try`-с-ресурсами выглядит следующим образом:

```
public void readFirstLineNew(java.io.File) throws java.io.IOException;
  Code:
    0: new             #2 // Класс java/io/BufferedReader
    3: dup
    4: new             #3 // Класс java/io/FileReader
    7: dup
    8: aload_1
    9: invokespecial   #4 // Метод java/io/FileReader."<init>":
                          // (Ljava/io/File;)V
   12: invokespecial   #5 // Метод java/io/BufferedReader."<init>":
                          // (Ljava/io/Reader;)V
   15: astore_2
   16: aconst_null
   17: astore_3
   18: aload_2
   19: invokevirtual   #6 // Метод java/io/BufferedReader.readLine:
                          // ()Ljava/lang/String;
   22: astore          4
   24: getstatic       #7 // Поле
                          // java/lang/System.out:Ljava/io/PrintStream;
   27: aload           4
   29: invokevirtual   #8 // Метод java/io/PrintStream.println:
                          // (Ljava/lang/String;)V
```

```
   32: aload_2
   33: ifnull        108
   36: aload_3
   37: ifnull         58
   40: aload_2
   41: invokevirtual  #9 // Метод java/io/BufferedReader.close:()V
   44: goto          108
   47: astore          4
   49: aload_3
   50: aload           4
   52: invokevirtual #11 // Метод java/lang/Throwable.addSuppressed:
                         // (Ljava/lang/Throwable;)V
   55: goto          108
   58: aload_2
   59: invokevirtual  #9 // Метод java/io/BufferedReader.close:()V
   62: goto          108
   65: astore          4
   67: aload           4
   69: astore_3
   70: aload           4
   72: athrow
   73: astore          5
   75: aload_2
   76: ifnull        105
   79: aload_3
   80: ifnull        101
   83: aload_2
   84: invokevirtual  #9 // Метод java/io/BufferedReader.close:()V
   87: goto          105
   90: astore          6
   92: aload_3
   93: aload           6
   95: invokevirtual #11 // Метод java/lang/Throwable.addSuppressed:
                         // (Ljava/lang/Throwable;)V
   98: goto          105
  101: aload_2
  102: invokevirtual #9 // Метод java/io/BufferedReader.close:()V
  105: aload          5
  107: athrow
  108: return
Таблица исключений:
  from   to   target   type
    40   44     47     Класс java/lang/Throwable
    18   32     65     Класс java/lang/Throwable
    18   32     73     any
    83   87     90     Класс java/lang/Throwable
    65   75     73     any
```

На первый взгляд, try-с-ресурсами — это просто механизм компилятора для автогенерации шаблона. Однако при последовательном использовании это очень полезное упрощение, которое может избавить классы от необходимости знать, как освобождать и очищать другие классы. В результате достигается более высокая степень инкапсуляции и менее склонный к ошибкам код.

Механизм try-с-ресурсами является рекомендуемой передовой практикой для реализации чего-то похожего на идиому RAII в C++. Использование этого шаблона ограничено кодом с областью видимости блока, но это связано с тем, что платформе Java не хватает низкоуровневой видимости времени жизни объекта. Разработчик Java должен просто проявлять дисциплину при работе с объектами ресурсов и максимально сокращать их области видимости, что само по себе является хорошей практикой проектирования.

К настоящему времени должно быть ясно, что эти два механизма (финализация и try-с-ресурсами), несмотря на одну и ту же цель проектирования, радикально различаются.

Финализация полагается на ассемблерный код глубоко внутри среды времени выполнения для регистрации объектов для специальной обработки при сборке мусора. Затем она использует сборщик мусора, чтобы выполнить очистку с использованием очереди ссылок и отдельного специализированного потока финализации. В частности, в байт-коде мало (если они вообще есть) следов финализации; эта возможность предоставляется специальными механизмами виртуальной машины.

Напротив, try-с-ресурсами — это возможность времени компиляции, которая может рассматриваться как синтаксический сахар, создающий обычный байт-код и не имеющий никакого специального поведения во время выполнения. Единственное возможное влияние использования try-с-ресурсами на производительность заключается в том, что, поскольку эта конструкция приводит к большому количеству автоматически генерируемого байт-кода, это может повлиять на способность JIT-компилятора эффективно встраивать или компилировать методы, которые используют этот подход.

Однако, как и во всех других потенциальных влияниях тех или иных действий на производительность, инженер должен измерять влияние применения конструкции try-с-ресурсами на производительность времени выполнения и тратить усилия на рефакторинг, только если можно четко показать, что использованная функциональная возможность вызывает проблемы.

Таким образом, финализация не подходит для управления ресурсами, как и почти во всех других случаях. Финализация зависит от сборки мусора, которая сама по себе является недетерминированным процессом. Это означает, что ничто, полагающееся на финализацию, не имеет никакой гарантии относительно времени освобождения захваченного ресурса.

Независимо от того, придет ли Java к окончательному удалению финализации из языка, совет остается прежним: не пишите классы, которые перекрывают `finalize()`, и рефакторизуйте любые такие классы, которые вы найдете в своем коде.

Дескрипторы методов

В главе 9, "Выполнение кода в JMV", мы встретились с командой `invokedynamic`. Эта важная команда, введенная начиная с Java 7, обеспечивает большую гибкость при определении того, какой метод должен выполняться в точке вызова. Ключевым моментом является то, что точка вызова `invokedynamic` до времени выполнения не определяет, какой именно метод должен быть вызван.

Вместо этого интерпретатор достигает точки вызова, и вызывается специальный вспомогательный метод (известный как *метод начальной загрузки* (bootstrap method — BSM)). BSM возвращает объект, представляющий фактический метод, который должен быть вызван в точке вызова. Он известен как *цель вызова* (call target) и вставляется в точку вызова.

В простейшем случае поиск цели вызова выполняется только один раз — при первой встрече точки вызова, — но бывают и более сложные случаи, когда точка вызова может стать недействительной и потребуется повторный поиск (который, возможно, найдет другую цель вызова).

Ключевой концепцией является *дескриптор метода* (method handle) — объект, представляющий метод, который должен быть вызван в точке вызова `invokedynamic`. Эта концепция несколько напоминает концепции рефлексии, но присущие рефлексии ограничения делают ее неудобной для использования с `invokedynamic`.

Вместо этого в Java 7 было добавлено несколько новых классов и пакетов (в частности, `java.lang.invoke.MethodHandle`) для представления *непосредственно выполняемых* (directly executable) ссылок на методы. Эти объекты дескрипторов методов имеют группу из нескольких связанных методов, позволяющих выполнять базовый метод. Наиболее распространенным из них является `invoke()`, но есть и дополнительные вспомогательные методы, и небольшие вариации первичного метода вызова `invoke()`.

Точно так же, как и для рефлексивных вызовов, базовый метод дескриптора метода может иметь любую сигнатуру. Поэтому упоминаемые выше методы вызова, присутствующие в дескрипторах методов, должны иметь очень разрешительную сигнатуру для обеспечения полной гибкости. Однако дескрипторы методов имеют также и новые функциональные возможности, выходящие за рамки рефлексии.

Чтобы понять эти новые возможности и почему они важны, давайте сначала рассмотрим простой код, который рефлексивно вызывает метод:

```
Method m = ...
Object receiver = ...
Object o = m.invoke(receiver, new Object(), new Object());
```

Это приводит к следующему довольно тривиальному фрагменту байт-кода:

```
17: iconst_0
18: new           #2 // Класс java/lang/Object
21: dup
22: invokespecial #1 // Метод java/lang/Object."<init>":()V
25: aastore
26: dup
27: iconst_1
28: new           #2 // Класс java/lang/Object
31: dup
32: invokespecial #1 // Метод java/lang/Object."<init>":()V
35: aastore
36: invokevirtual #3 // Метод java/lang/reflect/Method.invoke
                     // :(Ljava/lang/Object;[Ljava/lang/Object;)
                     // Ljava/lang/Object;
```

Коды операций `iconst` и `aastore` используются для сохранения нулевых и первых элементов аргументов переменной длины в массив, передаваемый `invoke()`. После этого ясно, что общая сигнатура вызова в байт-коде — `invoke:(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;`, поскольку метод принимает единственный аргумент — объекта (получатель), за которым следует переменное число параметров, которые будут переданы рефлексивному вызову. В конечном итоге он возвращает `Object`, и все это указывает на то, что во время компиляции об этом вызове метода ничего не известно — мы откладываем все аспекты вызова до времени выполнения.

В результате это очень обобщенный вызов, который может завершиться неудачно во время выполнения, если объект получателя и объект `Method` не соответствуют один другому или если список параметров неверен.

Для контраста рассмотрим аналогичный простой пример, выполненный с помощью дескрипторов методов:

```
MethodType mt = MethodType.methodType(int.class);
MethodHandles.Lookup l = MethodHandles.lookup();
MethodHandle mh = l.findVirtual(String.class, "hashCode", mt);

String receiver = "b";
int ret = (int) mh.invoke(receiver);
System.out.println(ret);
```

Вызов состоит из двух частей: сначала выполняется поиск дескриптора метода, а затем — его вызов. В реальных системах эти две части могут быть далеко разнесены по времени или коду; дескрипторы методов являются неизменяемыми стабильными объектами и могут быть легко кешированы и сохранены для последующего использования.

Механизм поиска выглядит как дополнительный стандартный текст, но используется для решения вопроса, который был настоящей проблемой рефлексии с момента ее создания — контроля доступа.

При первоначальной загрузке класса байт-код всесторонне проверяется. Сюда включаются проверки на то, что класс не пытается злонамеренно вызвать методы, к которым у него нет доступа. Любая попытка вызова недоступного метода приводит к сбою процесса загрузки класса.

По соображениям производительности, как только класс загружен, дальнейшие проверки не выполняются. Тем самым приоткрывается щелка, которую может попытаться использовать рефлексивный код. Исходный выбор дизайна, сделанный подсистемой отражения (в Java 1.1), является не полностью удовлетворительным по нескольким различным причинам.

API дескрипторов методов использует иной подход — контекст поиска. Чтобы его использовать, мы создаем объект контекста, вызывая метод `MethodHandles.lookup()`. Возвращаемый неизменяемый объект имеет состояние, записывающее, какие методы и поля были доступны в точке, где объект контекста был *создан*.

Это означает, что объект контекста может быть как немедленно использован, так и сохранен. Такая гибкость позволяет использовать схемы, в которых класс может разрешать выборочный доступ к своим закрытым методам (путем кеширования объекта поиска и фильтрации доступа к нему). У рефлексии же, напротив, есть только инструмент для хака `setAccessible()`, который полностью аннулирует возможности безопасности системы управления доступом в Java.

Давайте посмотрим на байт-код раздела поиска из примера с дескрипторами методов:

```
0: getstatic      #2 // Поле java/lang/Integer.TYPE:Ljava/lang/Class;
3: invokestatic   #3 // Метод java/lang/invoke/MethodType.methodType:
                     // (Ljava/lang/Class;)Ljava/lang/invoke/MethodType;
6: astore_1
7: invokestatic   #4 // Метод java/lang/invoke/MethodHandles.lookup:
                     // ()Ljava/lang/invoke/MethodHandles$Lookup;
10: astore_2
11: aload_2
12: ldc           #5 // Класс java/lang/String
14: ldc           #6 // String hashCode
16: aload_1
17: invokevirtual #7 // Метод
          // java/lang/invoke/MethodHandles$Lookup.findVirtual:
```

```
            // (Ljava/lang/Class;Ljava/lang/String;Ljava/lang/invoke/
            // MethodType;)Ljava/lang/invoke/MethodHandle;
20: astore_3
```

Этот код сгенерировал объект контекста, который может видеть каждый метод, доступный в точке, где происходит статический вызов `lookup()`. Отсюда можно использовать `findVirtual()` (и связанные методы), чтобы получить дескриптор любого метода, видимого в данной точке. Если попытаться получить доступ к методу, который не видим через контекст поиска, то будет сгенерировано исключение `IllegalAccessException`. В отличие от рефлексии, программист не может обойти или отключить эту проверку доступа.

В нашем примере мы просто ищем общедоступный, не требующий специального доступа метод `hashCode()` для `String`. Тем не менее мы все равно должны использовать механизм поиска, и платформа все равно проверит, имеет ли объект контекста доступ к запрошенному методу. Теперь давайте посмотрим на байт-код, сгенерированный вызовом дескриптора метода:

```
21: ldc            #8 // String b
23: astore          4
25: aload_3
26: aload           4
28: invokevirtual  #9 // Метод java/lang/invoke/MethodHandle.invoke
                      // :(Ljava/lang/String;)I
31: istore          5
33: getstatic     #10 // Поле java/lang/System.out:Ljava/io/PrintStream;
36: iload           5
38: invokevirtual #11 // Метод java/io/PrintStream.println:(I)V
```

Этот код существенно отличается от рефлективного случая, потому что вызов `invoke()` представляет собой не просто абсолютно универсальный вызов, который принимает любые аргументы, но описывает ожидаемую сигнатуру метода, который должен вызываться во время выполнения.

Байт-код для вызова дескриптора метода содержит лучшую информацию о статическом типе вызова, чем в соответствующем рефлективном случае.

В нашем случае сигнатура вызова имеет вид `invoke:(Ljava/lang/String;)I`, и ничто в JavaDoc для `MethodHandle` не указывает, что класс имеет такой метод.

Вместо этого компилятор исходного кода `javac` выводит подходящую сигнатуру и генерирует соответствующий вызов, хотя такой метод в `MethodHandle` и не существует. Байт-код, генерируемый `javac`, также настраивает стек таким образом, что этот вызов будет диспетчеризован обычным способом (при условии, что он может быть связан), без какой-либо упаковки аргументов переменной длины в массив.

Любая среда выполнения JVM, загружающая этот байт-код, требует связывания этого вызова метода "как есть" с ожиданием того, что дескриптор метода во время выполнения будет представлять вызов корректной сигнатуры и что вызов `invoke()` будет, по сути, заменен делегированным вызовом базового метода.

Эта несколько странная особенность языка Java известна как *полиморфизм сигнатур* и применяется только к дескрипторам методов.

Эта возможность, конечно, выглядит совершенно не в стиле языка программирования Java, и в случае использования намеренно искажается разработчиками языка и платформы (как это было сделано, например, с `Dynamic` в C#).

Для множества разработчиков проще всего представлять дескрипторы методов как предоставляющие те же возможности, что и базовая рефлексия, но делающие это в современном стиле, с максимально возможной безопасностью статических типов.

Резюме

В этой главе мы рассмотрели некоторые аспекты производительности стандартного API коллекций Java. Мы также познакомились с ключевыми проблемами работы с объектами предметной области.

Кроме того, мы рассмотрели два других аспекта производительности приложений, в большей степени связанные с уровнем платформы: финализацию и дескрипторы методов. Обе эти концепции не принадлежат к тем, с которыми разработчики сталкиваются каждый день. Тем не менее для инженера, ориентированного на производительность, их знание и понимание будут полезными дополнениями к набору используемых технологий.

В следующей главе мы перейдем к обсуждению нескольких важных библиотек с открытым исходным кодом, включая предоставляющие альтернативы стандартным коллекциям, а также протоколирования и связанных с ним вопросов.

ГЛАВА 12

Методы повышения производительности параллельной работы

Так уж сложилась история вычислительной техники и вычислений, что на сегодняшний день разработчики программного обеспечения обычно записывают код в последовательном формате. Языки программирования и аппаратное обеспечение в общем случае поддерживали возможность выполнения одновременно только одной команды. Во множестве ситуаций при необходимости повысить производительность приложений просто покупалось новейшее оборудование. Увеличение количества транзисторов в чипе привело к созданию все более и более совершенных процессоров.

Многие читатели сталкивались с ситуацией, когда решением проблем с пропускной способностью было перемещение программного обеспечения в *б*ольшую или более новую железную коробку, а не выполнение исследования проблем производительности или применение иной парадигмы программирования.

Закон Мура изначально предсказывал удвоение количества транзисторов на чипе примерно каждый год. Позже оценка была уточнена — каждые 18 месяцев. Закон Мура продержался около 50 лет, но он начинает сбоить. Импульс, который поддерживал темпы развития вычислительной техники в течение 50 лет, все труднее поддерживать. На рис. 12.1, взятом из статьи Гебра Саттера (Herb Sutter) "Бесплатные обеды закончились"[1], хорошо видно начало современной эры анализа производительности.

Теперь мы живем в мире, в котором многоядерные процессоры являются обычным явлением. Хорошо написанные приложения могут (и все чаще должны) использовать преимущества распределенного выполнения приложений несколькими ядрами. Такие платформы выполнения приложений, как JVM, имеют явное преимущество, связанное с тем, что всегда есть потоки VM, которые могут использовать

[1] Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal* 30 (2005): 202–210.

возможности наличия нескольких процессорных ядер для таких операций, как, например, JIT-компиляция. Это означает, что даже приложения JVM, имеющие только один прикладной поток, выигрывают от многоядерности процессора.

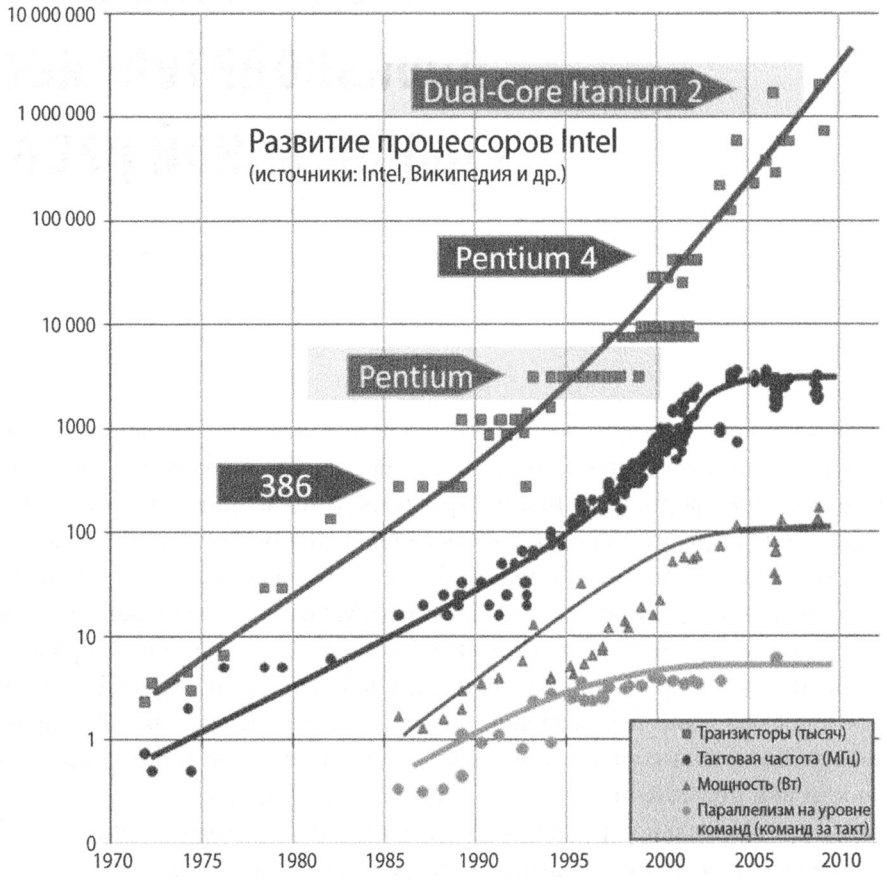

Рис. 12.1. "Бесплатные обеды закончились" (Саттер, 2005)

Чтобы в полной мере использовать текущее аппаратное обеспечение, современный профессионал Java должен иметь как минимум базовое понимание параллелизма и его влияния на производительность приложений. Данная глава является вступительным обзором и не рассчитана на полный охват огромной темы параллелизма в Java. Если вас интересует эта тема, обратитесь к другим книгам, таким как *Java Concurrency in Practice* Брайана Гётца (Brian Goetz) и др.

Введение в параллельные вычисления

В течение почти 50 лет одноядерная скорость увеличивалась, а затем примерно в 2005 году она начала выходить на плато со скоростью около 3 ГГц. В сегодняшнем

многоядерном мире основой для повышения скорости выполнения расчетных задач стал закон Амдала.

Мы встретились с законом Амдала в разделе "Графики производительности" главы 1, "Оптимизация и производительность", но теперь нам нужно более формальное его описание. Рассмотрим вычислительную задачу, которая может быть разделена на две части: одну часть, которая может выполняться параллельно, и часть, которая должна выполняться только последовательно (например, для объединения результатов или для диспетчеризации заданий для параллельного выполнения).

Давайте обозначим последовательную часть как `S`, а общее время, необходимое для решения задачи, как `T`. Мы можем использовать столько процессоров, сколько хотим, — обозначим их количество как `N`. Таким образом, мы должны записывать время `T` как функцию общего количества процессоров — `T(N)`. Параллельная часть работы равна `T-S`, и если ее можно распределить поровну между `N` процессорами, то общее время, затрачиваемое на выполнение задачи, составляет

```
T(N) = S + (1/N)*(T - S)
```

Это означает, что независимо от того, сколько процессоров используется, общее время работы не может быть меньше времени выполнения последовательной части. Таким образом, если накладные расходы на последовательный интерфейс составляют, скажем, 5% от общего количества времени, то независимо от того, сколько ядер используется, результирующее ускорение никогда не превысит 20 раз. Понимание этого закона и его формула являются основополагающей теорией обсуждения закона Амдала в главе 1, "Оптимизация и производительность". Другое представление влияния количества процессоров на время работы приложения показано на рис. 12.2.

Уменьшить значение `S` могут только улучшения однопоточной производительности, такие как более быстрые ядра. К сожалению, современные тенденции в современном аппаратном обеспечении означают, что сколь-нибудь значимого повышения тактовой частоты процессора ожидать не приходится. Из-за того что одноядерные процессоры больше не становятся быстрее, закон Амдала часто указывает практический предел масштабирования программного обеспечения.

Одним из следствий закона Амдала является то, что если связь между параллельными задачами или другой последовательной обработкой не требуется, то теоретически возможно неограниченное ускорение. Такой класс задач известен как *чрезвычайно параллельный* (embarrassingly parallel), и в этом случае достижение параллельной обработки оказывается достаточно простым.

В этом случае обычный подход заключается в разделении рабочей нагрузки между несколькими рабочими потоками без каких-либо совместно используемых данных. Если же имеются совместно используемые потоками данные, сложность рабочей нагрузки возрастает, и в приложение неизбежно добавляются некоторая последовательная обработка и накладные расходы на межпроцессорное взаимодействие.

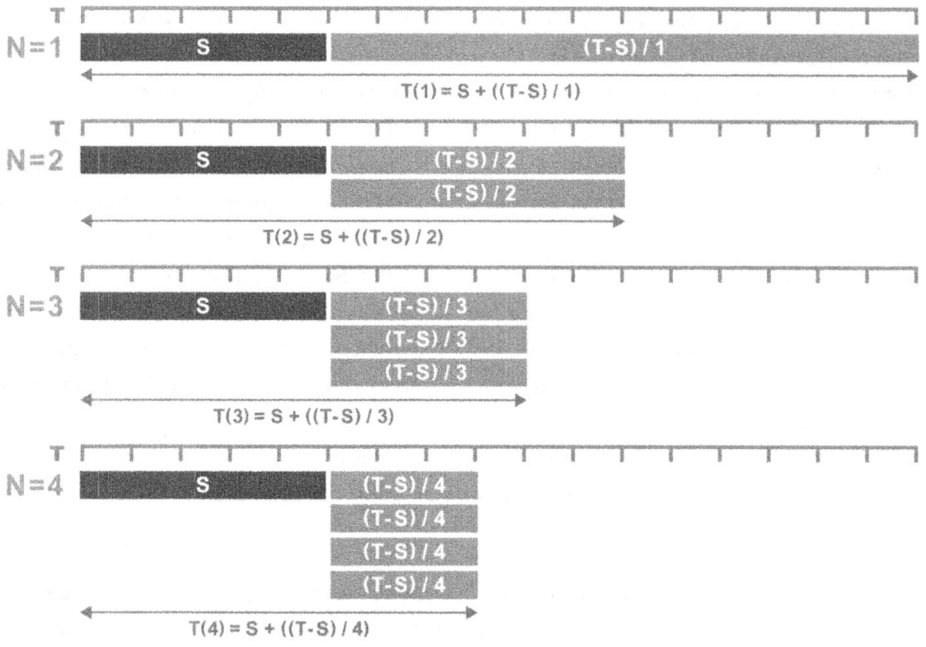

Рис. 12.2. Иллюстрация закона Амдала

> *Написать правильную программу трудно, но написать правильную параллельную программу еще труднее. Просто в параллельной программе гораздо больше вещей, которые могут пойти не так, чем в последовательной программе.*
>
> — *Брайан Гетц и др.* Java Concurrency in Practice

В свою очередь, это означает, что любая рабочая нагрузка с совместно используемыми данными требует правильной защиты и управления. Для рабочих нагрузок, запущенных на JVM, платформа предоставляет набор гарантий по работе с памятью, именуемых моделью памяти Java (Java Memory Model — JMM). Давайте рассмотрим некоторые простые примеры, объясняющие проблемы параллелизма в Java, прежде чем ближе познакомиться с этой моделью.

Основы параллельных вычислений в Java

Один из первых уроков, извлеченных из контринтуитивной природы параллельных вычислений, — это осознание того, что увеличение счетчика не является единственной операцией. Взгляните:

```
public class Counter
{
    private int i = 0;

    public int increment()
```

```
    {
        return i = i + 1;
    }
}
```

Анализируя байт-код для этого фрагмента исходного текста, мы видим последовательности команд, которые выполняют загрузку, увеличение и сохранение значения:

```
public int increment();
  Code:
     0: aload_0
     1: aload_0
     2: getfield      #2 // Поле i:I
     5: iconst_1
     6: iadd
     7: dup_x1
     8: putfield      #2 // Поле i:I
    11: ireturn
```

Если счетчик не защищен соответствующей блокировкой и доступ к нему осуществляется многопоточным способом, может случиться так, что загрузка произойдет до того, как другой поток выполнит сохранение. Это может привести к утрате обновлений.

Чтобы рассмотреть ситуацию более подробно, рассмотрите два потока, A и B, которые вызывают метод `increment()` для одного и того же объекта. Для простоты предположим, что они работают на машине с одним процессором и что байт-код точно представляет низкоуровневое выполнение (без переупорядочения, кеш-эффектов или других деталей работы реальных процессоров).

Так как планировщик операционной системы предпринимает переключение контекста потоков недетерминированным образом, даже при работе всего лишь двух потоков возможны многие различные последовательности выполняемых байт-кодов.

Предположим, что единственный процессор выполняет байт-коды так, как показано далее (обратите внимание, что здесь есть точно определенный порядок выполнения инструкций, которого может не быть в реальной многопроцессорной системе):

```
A0: aload_0
A1: aload_0
A2: getfield      #2 // Поле i:I
A5: iconst_1
A6: iadd
A7: dup_x1
B0: aload_0
```

```
B1: aload_0
B2: getfield      #2 // Поле i:I
B5: iconst_1
B6: iadd
B7: dup_x1
A8: putfield      #2 // Поле i:I
A11: ireturn
B8: putfield      #2 // Поле i:I
B11: ireturn
```

Каждый поток будет иметь отдельный стек вычислений от своего отдельного входа в метод, поэтому мешать одна другой могут только операции над полями (поскольку поля объекта расположены в куче, которая является совместно используемой).

Результирующее поведение заключается в том, что если начальное состояние `i` равно `7` до того, как начнется выполнение `A` и `B`, то, если порядок выполнения будет таким, как показано выше, оба вызова вернут `8`, и состояние поля будет обновлено до этого значения, несмотря на тот факт, что `increment()` был вызван дважды.

Эта проблема вызвана ни чем иным, как планировщиком операционной системы; для этой проблемы не требуется никакого аппаратного трюка, и это было бы проблемой даже на очень старом процессоре без современных возможностей.

Следующее заблуждение состоит в том, что добавление ключевого слова `volatile` делает операцию инкремента безопасной. Заставляя значение всегда быть перечитанным кешем, `volatile` гарантирует, что любые обновления одним потоком будут видны другому. Тем не менее это не мешает возникновению только что рассмотренной проблемы потерянного обновления, так как она связана со сложной природой оператора инкремента.

В следующем примере показаны два потока, совместно использующие ссылку на один и тот же счетчик:

```
package optjava;

public class CounterExample implements Runnable
{
    private final Counter counter;

    public CounterExample(Counter counter)
    {
        this.counter = counter;
    }

    @Override
    public void run()
    {
```

```
        for (int i = 0; i < 100; i++)
        {
            System.out.println(Thread.currentThread().getName()
                    + " Value: " + counter.increment());
        }
    }
}
```

Счетчик не защищен с помощью `synchronized` или подходящей блокировки. При каждом запуске программы выполнение двух потоков может потенциально чередоваться по-разному. В некоторых случаях код будет работать, как ожидалось, и счетчик будет увеличиваться. Это зависит только от слепой удачи программиста! В других случаях чередование может показывать повторяющиеся значения счетчика из-за потерянных обновлений, как показано далее:

```
Thread-1 Value: 1
Thread-1 Value: 2
Thread-1 Value: 3
Thread-0 Value: 1
Thread-1 Value: 4
Thread-1 Value: 6
Thread-0 Value: 5
```

Другими словами, параллельная программа, которая выполняется успешно большую часть времени, — это не то же самое, что и правильная параллельная программа. Доказать ошибочность программы так же сложно, как и доказать ее правильность; однако достаточно найти один пример отказа, чтобы продемонстрировать, что она неверна.

Что делает ситуацию еще хуже, так это то, что воспроизведение ошибок в параллельном коде может быть чрезвычайно сложным. Известный принцип Дейкстры, "тестирование показывает присутствие, а не отсутствие ошибок", применимо к параллельному коду еще более, чем к однопоточным приложениям.

Чтобы решить вышеупомянутые проблемы, можно использовать ключевое слово `synchronized` для управления обновлением простого значения, такого как `int`, — и до Java 5 это было единственным вариантом.

Проблема с использованием синхронизации заключается в том, что для этого требуются тщательный дизайн и предварительные размышления. Без этого простое добавление синхронизации может не ускорить работу программы, а замедлить.

Это совершенно противоречит целям параллелизма — увеличению пропускной способности. Соответственно, любое действие по параллелизации базы кода должно поддерживаться тестами производительности, которые должны полностью подтверждать преимущества внесения дополнительной сложности в программу.

Добавление блоков синхронизации, особенно бесполезных, намного дешевле, чем в более старых версиях JVM (но их все равно не нужно добавлять, если в них нет необходимости). Более подробную информацию можно найти в разделе "Блокировки и анализ локальности" главы 10, "JIT-компиляция".

Чтобы сделать нечто лучшее, чем простой бессистемный подход к синхронизации, следует понимать низкоуровневую модель памяти JVM и то, как она применима в практических методах параллельных приложений.

Понимание JMM

Java имеет формальную модель памяти (JMM), начиная с версии 1.0. Эта модель была достаточно серьезно пересмотрена, и ряд проблем был исправлен в спецификации JSR 133[2], выпущенной как часть Java 5.

В спецификациях Java JMM появляется как математическое описание памяти. Она имеет несколько грозную репутацию, и многие разработчики считают ее самой непонятной частью спецификации Java (за исключением, возможно, обобщенных типов).

JMM стремится предоставить ответы на такие вопросы.

- Что случится, когда два ядра обратятся к одним и тем же данным?
- Когда они гарантированно увидят одно и то же значение?
- Как кеши памяти влияют на ответы на эти вопросы?

Везде при обращении к совместно используемому состоянию платформа обеспечит выполнение обещаний, сделанных в JMM. Эти обещания делятся на две основные группы: гарантии, связанные с упорядочением и касающиеся видимости обновлений между потоками.

Поскольку аппаратное обеспечение перешло от одноядерных к многоядерным системам, природа модели памяти становится все более и более важной. Упорядочение и видимость потоков больше не являются теоретическими проблемами; теперь это сугубо практические вопросы, непосредственно влияющие на код работающих программистов.

На высоком уровне имеются два возможных подхода, которые могла бы использовать такая модель памяти, как JMM.

Сильная модель памяти

В любой момент времени все ядра всегда видят одни и те же значения.

[2] Платформа Java развивается через запросы спецификаций Java (Java Specification Requests — JSR), которые отслеживают усовершенствования стандартов платформы.

> Ядра могут видеть различные значения, и имеются специальные правила кеша, управляющие тем, когда это может быть.

С точки зрения программирования сильная модель памяти кажется очень привлекательной — не в последнюю очередь потому, что не требует от программистов особой осторожности при написании кода приложения.

На рис. 12.3 представлен (значительно) упрощенный вид современной многопроцессорной системы. Мы уже видели его в главе 3, "Аппаратное обеспечение и операционные системы", и в главе 7, "Вглубь сборки мусора" (где оно обсуждалось в контексте архитектур NUMA).

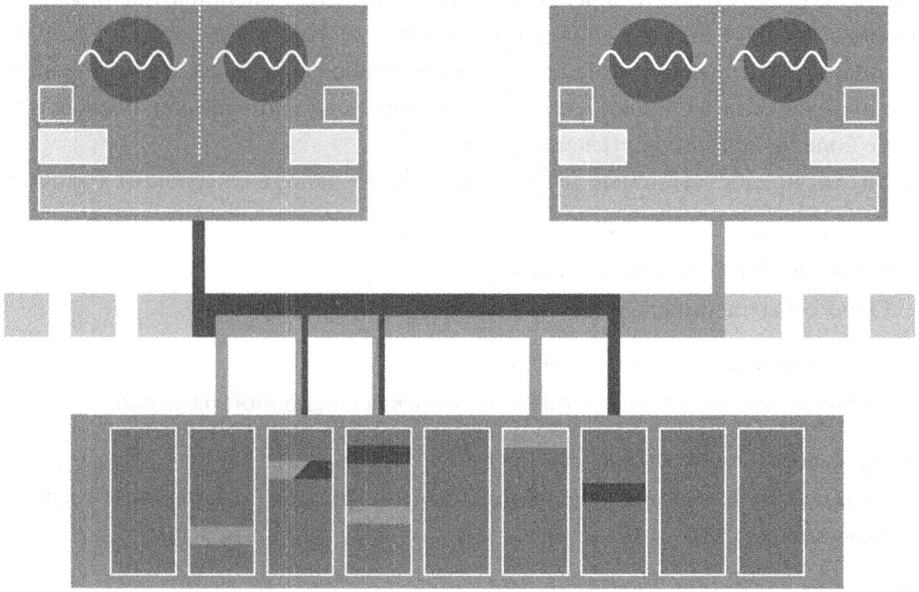

Рис. 12.3. Современные многопроцессорные системы

Если бы сильная модель памяти была реализована поверх этого аппаратного обеспечения, это было бы эквивалентно подходу обратной записи в память. Уведомления о недействительности кеша будут затапливать шину памяти, а эффективные скорости передачи в основную память и из нее резко упадут. Проблема будет только усугубляться по мере увеличения количества ядер, что делает этот подход принципиально непригодным для многоядерного мира.

Стоит также вспомнить, что язык программирования Java спроектирован как независимая от архитектуры среда. Это означает, что если бы JVM специфицировала сильную модель памяти, потребовалась бы дополнительная работа по ее реализации в программном обеспечении, работающем поверх не поддерживающего сильную модель памяти аппаратного обеспечения. В свою очередь, это значительно увеличило бы работу по переносу JVM для работы на слабом оборудовании.

В действительности JMM имеет очень слабую модель памяти. Это лучше соответствует тенденциям в реальной архитектуре процессора, включая MESI (см. раздел "Кеши памяти" главы 3, "Аппаратное обеспечение и операционные системы") и упрощает перенос, поскольку JMM дает мало гарантий.

Очень важно понимать, что JMM является только минимальным требованием. Реальные реализации JVM и процессоры могут делать больше, чем требует JMM, как описано в разделе "Аппаратные модели памяти" главы 3, "Аппаратное обеспечение и операционные системы".

Это может привести к тому, что разработчики приложений будут усыплены ложным чувством безопасности. Если приложение разрабатывается на аппаратной платформе с более сильной моделью памяти, чем JMM, то необнаруженные ошибки параллелизма могут выжить, просто потому что они не проявляются на практике из-за аппаратных гарантий. Когда то же самое приложение развертывается на более слабом оборудовании, ошибки параллелизма могут стать проблемой, поскольку приложение больше не защищается аппаратно.

Гарантии, предоставляемые JMM, основаны на наборе следующих базовых понятий.

Происходит до (Happens-Before)

Одно событие определенно происходит до второго.

Синхронизировано с (Synchronizes-With)

Событие вызывает синхронизацию объекта с основной памятью.

Как в последовательном случае (As-If-Serial)

За пределами выполняющегося потока инструкции отображаются в порядке выполнения.

Освобождение до захвата (Release-Before-Acquire)

Блокировки будут освобождены одним потоком до того, как будут захвачены другим.

Одним из наиболее важных методов обработки совместно используемого изменяемого состояния является блокировка посредством синхронизации. Это фундаментальная часть представления Java о параллелизме, и чтобы адекватно работать с JMM, нам нужно будет в определенной мере ее обсудить.

Для разработчиков, которые заинтересованы в производительности приложения, недостаточно знакомства с классом `Thread` и базовыми примитивами уровня языка из механизма параллелизма Java.

В этом представлении потоки имеют собственное описание состояния объекта, и любые изменения, сделанные потоком, должны быть сброшены в основную память

и затем перечитаны любыми другими потоками, обращающимися к тем же данным. Это хорошо согласуется с представлением об аппаратном обеспечении в контексте MESI, но в реализации JVM имеется много кода, который является оболочкой для низкоуровневого доступа к памяти.

С этой точки зрения сразу становится ясно, к чему относится ключевое слово Java `synchronized`: оно означает, что локальное представление потока, содержащего монитор, синхронизировано с основной памятью.

Синхронизированные методы и блоки определяют контрольные точки, в которых потоки должны выполнять синхронизацию. Они также определяют блоки кода, которые должны полностью завершиться до того, как смогут начаться другие синхронизированные блоки или методы.

JMM ничего не говорит о несинхронизированном доступе. Нет никаких гарантий о том, когда изменения, сделанные в одном потоке, станут видимыми другим потокам (если вообще станут видимыми). Если требуются такие гарантии, то доступ для записи должен быть защищен синхронизированным блоком, инициирующим обратную запись кешированных значений в основную память. Аналогично доступ для чтения также должен содержаться в синхронизированном разделе кода, чтобы выполнялось принудительное перечитывание данных из памяти.

До появления современного параллелизма Java использование ключевого слова `synchronized` было единственным механизмом, гарантирующим упорядочение и видимость данных нескольким потокам.

JMM обеспечивает такое поведение и предлагает различные гарантии в отношении безопасности Java и памяти. Однако традиционная блокировка `synchronized` имеет несколько ограничений, которые становятся все более и более серьезными:

- все операции `synchronized` над заблокированным объектом обрабатываются одинаково;
- захват и освобождение блокировки должны выполняться на уровне метода или внутри блока `synchronized` внутри метода;
- если блокировка не захватывается, поток блокируется; нет возможности попытаться захватить блокировку и продолжить работу, если захватить блокировку оказалось невозможно.

Очень распространенная ошибка заключается в забывании, что операции над заблокированными данными должны обрабатываться одинаково. Если приложение использует `synchronized` только для операций записи, это может привести к потере обновлений.

Например, может показаться, что чтение как будто не требует блокировки, но оно должно использовать ключевое слово `synchronized`, чтобы гарантировать видимость обновлений из других потоков.

Синхронизация Java между потоками является механизмом сотрудничества, который не будет корректно работать, если хотя бы один из участвующих потоков не будет следовать правилам.

Одним из ресурсов для новичков в JMM является *JSR-133 Cookbook for Compiler Writers*[3]. В нем содержится упрощенное объяснение концепций JMM, не подавляющее читателя обилием технических деталей.

Например, в рамках модели памяти вводится и обсуждается ряд абстрактных барьеров. Они предназначены для того, чтобы разработчики JVM и авторы библиотек могли рассматривать правила параллелизма Java относительно независимо от используемого процессора.

Правила, которым должны следовать реализации JVM, подробно описаны в спецификациях Java. На практике фактические команды, которые реализуют каждый абстрактный барьер, могут быть разными для разных процессоров. Например, модель процессора Intel автоматически аппаратно предотвращает некоторые переупорядочения, поэтому ряд барьеров, описанных в упомянутой выше книге, фактически представляют собой отсутствие операций.

Еще одно, последнее, соображение: в вычислительной технике все находится в постоянном развитии. Ни эволюция аппаратного обеспечения, ни границы параллелизма не застыли в момент создания JMM. В результате описание JMM является неадекватным представлением современного аппаратного обеспечения и памяти.

В Java 9 модель JMM была расширена в попытке догнать (хотя бы частично) реальность современных вычислительных систем. Одним из ключевых аспектов является совместимость с другими средами программирования, особенно с C++11, которые адаптировали идеи из JMM, а затем расширили их. Это означает, что модель C++11 предоставляет определения концепций, выходящих за рамки Java 5 JMM (JSR 133). Java 9 обновляет JMM, чтобы привнести некоторые из этих концепций в платформу Java и позволить низкоуровневому, зависящему от аппаратного обеспечения Java-коду последовательно и согласованно взаимодействовать с C++11.

Если вы хотите углубиться в вопросы, связанные с JMM, обратитесь к блогу Алексея Шипилева "Близкие встречи с моделью памяти Java"[4], который является отличным источником комментариев и очень подробной технической информации.

Построение параллельных библиотек

Несмотря на всю успешность, JMM трудно понять, и еще труднее перевести это понимание в практическое использование. Это связано с недостатком гибкости, обеспечиваемой встроенной блокировкой.

[3] `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`

[4] `https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/`

В результате начиная с Java 5 наблюдается тенденция к стандартизации высококачественных параллельных библиотек и инструментов как части библиотеки классов Java и отхода от встроенной поддержки на уровне языка. В подавляющем большинстве случаев использования (даже в чувствительных к производительности приложениях) эти библиотеки обеспечивают лучшее решение проблем, чем создание новых абстракций "с нуля".

Библиотеки в `java.util.concurrent` были разработаны с целью упрощения написания многопоточных приложений в Java. Задача разработчика на языке программирования Java заключается в выборе уровня абстракции, который наилучшим образом соответствует имеющимся требованиям, и удачное совпадение заключается в том, что выбор хорошо отлаженных библиотек `java.util.concurrent` обеспечивает также лучшую производительность.

Предоставляемые фундаментальные строительные блоки делятся на несколько общих категорий:

- блокировки и семафоры (locks and semaphores);
- атомарные объекты (atomics);
- блокирующие очереди (blocking queues);
- защелки (latches);
- исполнители (executors).

На рис. 12.4 мы можем видеть представление типичного современного параллельного приложения Java, построенное из примитивов параллелизма и бизнес-логики.

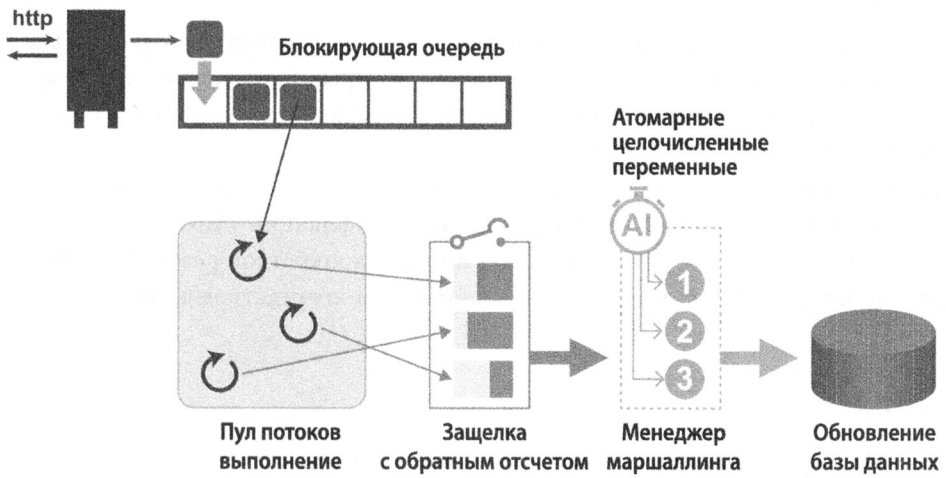

Рис. 12.4. Пример параллельного приложения

Некоторые из этих строительных блоков обсуждаются в следующем разделе, но прежде чем перейти к их рассмотрению, давайте взглянем на некоторые из основных

методов реализации, используемых в библиотеках. Понимание того, как реализуются параллельные библиотеки, позволит разработчикам, ориентированным на производительность, использовать их наилучшим образом. Для разработчиков, работающих "на переднем крае", знание, как работают библиотеки, дает командам, переросшим стандартную библиотеку, отправную точку для выбора (или разработки) замены со сверхвысокой производительностью.

В общем случае библиотеки пытаются отказаться от использования операционной системы и по возможности побольше работать в пользовательском пространстве. Этот подход имеет ряд преимуществ, не в последнюю очередь из-за того, что поведение библиотек, как мы надеемся, более последовательно, а не находится во власти небольших, но важных различий между Unix-подобными операционными системами.

Некоторые из библиотек (в частности, блокировки и атомарные объекты) полагаются на низкоуровневые команды процессора и особенности операционной системы для реализации метода, известного как *Сравнение и обмен* (Compare and Swap — CAS).

Эта методика принимает пару значений, "ожидаемое текущее значение" и "желаемое новое значение", а также местоположение в памяти (указатель). Две следующие операции выполняются атомарно, как единое целое.

1. Ожидаемое текущее значение сравнивается с содержимым по указанному месту в памяти.
2. Если они совпадают, текущее значение обменивается с желаемым новым значением.

CAS — фундаментальный строительный блок для нескольких критически важных функций параллелизма более высокого уровня, который представляет собой классический пример того, что производительность и аппаратное обеспечение не замерли в развитии с момента создания JMM.

Несмотря на то что команда CAS реализована в большинстве современных процессоров аппаратно, она не является частью спецификации JMM или Java. Вместо этого она должна рассматриваться как расширение конкретной реализации, поэтому доступ к аппаратной команде CAS предоставляется посредством класса `sun.misc.Unsafe`.

Unsafe

`sun.misc.Unsafe` представляет собой внутренний класс реализации, и, как и предполагает имя пакета, не является частью стандартного API платформы Java. Таким образом, в общем случае он вообще не должен использоваться разработчиками приложений напрямую; ключ к его разгадке кроется в самом имени класса.

Любой код, который технически его использует, непосредственно связан с виртуальной машиной HotSpot и потенциально неустойчив.

`Unsafe` представляет собой неподдерживаемый внутренний API, и поэтому технически может быть отозван, снят или изменен в любой момент без учета влияния этого шага на пользовательские приложения. В Java 9, как мы увидим позже, он помещен в модуль `jdk.unsupported`.

Тем не менее так или иначе `Unsafe` стал ключевой частью реализации практически каждого важного каркаса. Он предоставляет возможность изменения стандартного поведения JVM. Например, `Unsafe` обеспечивает такие возможности, как:

- выделение памяти для объекта без выполнения его конструктора;
- обращение к неформатированной памяти и арифметике указателей;
- использование аппаратных возможностей, специфичных для конкретного процессора (таких, как CAS).

Эти операции обеспечивают такие возможности высокоуровневого каркаса, как:

- быстрая сериализация и десериализация;
- безопасное с точки зрения потоков обращение к памяти (например, к памяти вне кучи или 64-разрядное индексированное обращение);
- атомарные операции с памятью;
- эффективная схема размещения объектов/памяти;
- пользовательские барьеры памяти;
- быстрое взаимодействие с машинным кодом;
- замена JNI для нескольких операционных систем;
- обращение к элементам массива с семантикой `volatile`.

Хотя `Unsafe` не является официальным стандартом для Java SE, его широко распространенное использование в промышленности сделало его стандартом де-факто. Кроме того, он стал чем-то вроде свалки для нестандартных, но необходимых функций. Особенное влияние на него оказало появление Java 9, и ожидается дальнейшее развитие `Unsafe` в следующих версиях Java.

Давайте посмотрим на CAS в действии, исследуя атомарные классы, введенные начиная с Java 5.

Атомарность и CAS

Атомарные классы предоставляют составные операции для суммирования, инкремента и декремента, которые в сочетании с `get()` возвращают полученный

результат. Это означает, что операция инкремента в двух отдельных потоках будет возвращать currentValue+1 и currentValue+2. Семантика атомарных переменных является расширением volatile, но они более гибкие, так как могут безопасно выполнять обновления, зависящие от состояния.

Атомарные классы не наследуются от базового типа, который они обертывают, и не допускают непосредственной замены. Например, AtomicInteger не расширяет Integer, к тому же java.lang.Integer является классом, объявленным как final.

Давайте посмотрим, как Unsafe обеспечивает реализацию простого атомарного вызова:

```
public class AtomicIntegerExample extends Number
{
    private volatile int value;
    // Настройка для использования
    // Unsafe.compareAndSwapInt для обновления
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;
    static
    {
        try
        {
            valueOffset = unsafe.objectFieldOffset(
                AtomicIntegerExample.class.getDeclaredField("value"));
        }
        catch (Exception ex)
        {
            throw new Error(ex);
        }
    }
    public final int get()
    {
        return value;
    }
    public final void set(int newValue)
    {
        value = newValue;
    }
    public final int getAndSet(int newValue)
    {
        return unsafe.getAndSetInt(this, valueOffset, newValue);
    }
    // ...
```

Этот код опирается на некоторые методы Unsafe, а ключевые методы здесь являются машинными и включают вызов JVM:

```
public final int getAndSetInt(Object o, long offset, int newValue)
{
    int v;

    do
    {
        v = getIntVolatile(o, offset);
    }
    while (!compareAndSwapInt(o, offset, v, newValue));

    return v;
}
public native int getIntVolatile(Object o, long offset);
public final native boolean compareAndSwapInt(Object o, long offset,
                                              int expected, int x);
```

Здесь продемонстрировано использование цикла в `Unsafe` для многократного повторения операции CAS. Для эффективного использования атомарности жизненно важно, чтобы разработчики использовали предоставленные средства и не применяли собственные реализации, скажем, атомарного приращения с использованием цикла, поскольку реализация `Unsafe` внутренне уже использует эту методику.

Атомарные переменные не используют блокировки и, следовательно, не могут тормозить код. Часто атомарность сопровождается внутренним повторяемым циклом для обработки ситуации сбоя сравнения и обмена. Обычно это происходит, когда другой поток выполняет просто обновление.

Такой повторяющийся цикл, если для обновления переменной требуются многократные попытки, приводит к линейному ухудшению производительности. При рассмотрении производительности важно следить за уровнем конфликтов, чтобы обеспечить высокий уровень пропускной способности.

Интересно использование `Unsafe` для обеспечения доступа к низкоуровневым аппаратным командам, так как Java обычно позволяет разработчику полностью абстрагироваться от машины. Однако в этом случае доступ к машинным командам имеет решающее значение для обеспечения необходимой семантики атомарных классов.

Блокировки и спин-блокировки

Внутренние блокировки, с которыми мы встречались до сих пор, работают путем вызова операционной системы из кода пользователя. Операционная система используется, чтобы помещать поток в состояние неограниченного ожидания до тех пор, пока не будет получен сигнал. Это может быть сопряжено с огромными накладными расходами, если ресурс, который вызывает конфликт, используется только в течение очень короткого периода времени. В этом случае может оказаться гораздо более эффективным, чтобы заблокированный поток оставался активным, но не выполнял

никакой полезной работы и занимал процессорное время, повторяя попытки блокировки до тех пор, пока она не станет доступной.

Этот метод известен как *циклическая* или *спин-блокировка* (spinlock) и предназначен в качестве более легкой блокировки — по сравнению с полной взаимоисключающей блокировкой исключения. В современных системах такие блокировки обычно реализуются с помощью CAS (в предположении, что аппаратное обеспечение поддерживает эту команду). Давайте посмотрим на простой пример на низкоуровневом ассемблере x86:

```
locked:
    dd 0

spin_lock:
    mov eax, 1
    xchg eax, [locked]
    test eax, eax
    jnz spin_lock
    ret

spin_unlock:
    mov eax, 0
    xchg eax, [locked]
    ret
```

Точная реализация спин-блокировки варьируется от процессора к процессору, но фундаментальная концепция остается одной и той же во всех системах:

- операция "проверка и установка", реализованная здесь с помощью команды `xchg`, должна быть атомарной;
- при наличии конфликта за захват спин-блокировки ожидающий процессор выполняет цикл.

По сути, CAS обеспечивает безопасное обновление значения одной командой, если ожидаемое значение является правильным. Это помогает нам сформировать строительные блоки для блокировки.

Краткий обзор параллельных библиотек

Мы познакомились с низкоуровневыми методами, используемыми для реализации атомарных классов и простых блокировок. Теперь давайте посмотрим, как стандартная библиотека использует эти возможности для создания полнофункциональных производственных библиотек общего назначения.

Блокировки в `java.util.concurrent`

В версии Java 5 блокировки были переделаны, а в `java.util.concurrent.locks.Lock` был добавлен более общий интерфейс блокировок. Этот интерфейс предлагает большие возможности, чем поведение встроенных блокировок.

`lock()`

Традиционный захват блокировки; поток остается в заблокированном состоянии, пока блокировка не станет доступной.

`newCondition()`

Создает условия для блокировки, позволяя использовать ее более гибко. Обеспечивает возможность разделения проблем внутри блокировок (например, чтение и запись).

`tryLock()`

Попытка захвата блокировки (с необязательным временем тайм-аута), позволяющая потоку продолжить работу в ситуации, когда блокировка не становится доступной.

`unlock()`

Освобождение блокировки. Это вызов, соответствующий вызову `lock()` и следующий за ним.

В дополнение к возможности создания блокировок различных типов теперь блокировки могут охватывать несколько методов, поскольку можно выполнить захват блокировки в одном методе, а ее освобождение — в другом. Если потоку требуется захват блокировки способом, который не будет блокировать работу потока, сделать это можно с помощью метода `tryLock()` и, если блокировка недоступна, продолжить выполнение с учетом этого факта.

`ReentrantLock` является главной реализацией `Lock` и в основном использует `compareAndSwap()` со значением `int`. Это означает, что в неконфликтной ситуации захват блокировки выполняется без использования блокировок. Это может значительно повысить производительность системы с небольшим количеством конфликтов блокировок, а также обеспечить дополнительную гибкость различных стратегий блокировки.

Идея о том, что поток может заново захватить одну и ту же уже захваченную блокировку, известна как *реентерабельная блокировка* (reentrant locking) и предотвращает блокировку потока самим собой. Большинство современных схем блокировки уровня приложения являются реентерабельными.

Фактические вызовы compareAndSwap() и использование Unsafe можно найти в статическом подклассе Sync, который является расширением AbstractQueuedSynchronizer. Класс AbstractQueuedSynchronizer использует класс LockSupport, который имеет методы, позволяющие потокам приостанавливаться, а затем и возобновлять работу.

Класс LockSupport работает, выдавая потокам разрешения, и, если доступных разрешений нет, поток должен переходить в состояние ожидания. Идея разрешений аналогична концепции выдачи разрешений в семафорах, но здесь имеется только одно разрешение (бинарный семафор). Если разрешение недоступно, поток будет "припаркован", а после того как корректное разрешение станет доступным, поток продолжит работу. Методы этого класса заменяют давно устаревшие методы Thread.suspend() и Thread.resume().

Существует три формы park(), которые влияют на работу следующего базового псевдокода.

```
while (!canProceed()) { ... LockSupport.park(this); }}
```

Вот они.

park(Object blocker)

> Блокирует работу, пока другой поток не выполнит вызов unpark(), поток не будет прерван или не произойдет ложное пробуждение.

parkNanos(Object blocker, long nanos)

> Ведет себя так же, как и park(), но по истечении указанного времени (в наносекундах) завершает работу.

parkUntil(Object blocker, long deadline)

> Подобен parkNanos(), но добавляет тайм-аут для сценариев, которые заставляют метод выполнить возврат.

Блокировки чтения/записи

Многие компоненты в приложениях демонстрируют дисбаланс между количеством операций чтения и операций записи. Чтение не изменяет состояние, в то время как суть записи именно в его изменении. Использование традиционного synchronized или ReentrantLock (без условий) будет следовать стратегии единственной блокировки. В таких ситуациях, как кеширование, когда может быть много читателей и один писатель, структура данных может тратить много времени на совершенно излишнее блокирование читателей из-за другого чтения.

Класс ReentrantReadWriteLock предоставляет ReadLock и WriteLock, которые могут использоваться в коде. Преимущество их применения состоит в том, что чтение нескольких потоков не приводит к блокировке одних потоков чтения другими.

Единственная операция, которая будет блокировать чтение, — это запись. Используя этот шаблон блокировки там, где количество читателей велико, можно значительно улучшить пропускную способность потока и уменьшить затраты на блокировку. Можно также установить блокировку в "справедливый режим" (fair mode), при котором производительность несколько упадет, но будет гарантироваться обработка потоков в порядке очереди.

Следующая реализация для AgeCache является значительным улучшением по сравнению с версией, использующей единую блокировку:

```
package optjava.ch12;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class AgeCache
{
    private final ReentrantReadWriteLock rwl = new
    ReentrantReadWriteLock();
    private final Lock readLock = rwl.readLock();
    private final Lock writeLock = rwl.writeLock();
    private Map<String, Integer> ageCache = new HashMap<>();
    public Integer getAge(String name)
    {
        readLock.lock();

        try
        {
            return ageCache.get(name);
        }
        finally
        {
            readLock.unlock();
        }
    }
    public void updateAge(String name, int newAge)
    {
        writeLock.lock();
        try
        {
            ageCache.put(name, newAge);
        }
        finally
        {
            writeLock.unlock();
        }
    }
}
```

Однако мы могли бы поступить еще более оптимально, рассмотрев базовую структуру данных. В этом примере более разумной абстракцией будет параллельная коллекция, которая обеспечит более высокую производительность.

Семафоры

Семафоры предлагают уникальную методику для обеспечения доступа к ряду доступных ресурсов, например потоков в пуле или объектов подключения к базе данных. Семафор работает с исходным условием, что "доступ разрешен не более чем X объектам", и функционирует, имея определенное количество разрешений для управления доступом:

```
// Семафор с двумя разрешениями и справедливой моделью
private Semaphore poolPermits = new Semaphore(2, true);
```

`Semaphore::acquire()` уменьшает количество доступных разрешений на единицу, и, если доступных разрешений нет, вызывающий поток будет заблокирован. `Semaphore::release()` возвращает разрешение, так что ожидающий поток (если таковой имеется) будет разблокирован. Поскольку семафоры часто используются там, где ресурсы потенциально блокируются или помещаются в очередь, семафор, вероятнее всего, будет инициализирован как справедливый, чтобы избежать голодания.

Семафор с одним разрешением (бинарный семафор) эквивалентен мьютексу, но с одним отличием. Мьютекс может быть освобожден только потоком, который заблокировал мьютекс, тогда как семафор может быть освобожден невладеющим потоком. Сценарий, в котором это может потребоваться, — разрешение взаимоблокировки. Преимущество семафоров заключается в возможности запросить и освободить несколько разрешений. Если используется несколько разрешений, важно использовать справедливый режим — в противном случае увеличивается вероятность голодания потоков.

Параллельные коллекции

В главе 11 мы рассмотрели методы оптимизации, которые могут быть применены к коллекциям Java. Начиная с Java 5 в языке были реализованы интерфейсы коллекций, специально разработанных для параллельного использования. Со временем эти параллельные коллекции были изменены и улучшены так, чтобы обеспечить наилучшую производительность потока.

Например, реализация отображения (`ConcurrentHashMap`) использует разделение на слоты или сегменты, и мы можем воспользоваться этой структурой, чтобы добиться реальных успехов в смысле производительности. Каждый сегмент может иметь собственную стратегию блокировки, т.е. собственный ряд блокировок.

Наличие блокировки как чтения, так и записи позволяет многим читателям читать ConcurrentHashMap, а если требуется запись, блокировка должна быть только в одном отдельном записываемом сегменте. Читатели в общем случае не блокируются и могут безопасно перекрывать операции в стиле put() и remove(). Читатели будут наблюдать для завершенной операции обновления упорядочение "происходит до".

Важно отметить, что итераторы (и разделители, используемые для параллельных потоков данных) захватываются как своего рода моментальный снимок, а это означает, что они не будут генерировать исключение ConcurrentModificationException. При слишком большом количестве коллизий таблица будет динамически расширяться, что может оказаться дорогостоящей операцией. Стоит (как и в случае HashMap) предоставлять при создании приблизительный размер — если, конечно, вы знаете его на момент написания кода.

В Java 5 также вошли CopyOnWriteArrayList и CopyOnWriteArraySet, которые в определенных схемах использования могут улучшить производительность при многопоточности. С их применением любая изменяющая операция для структуры данных создает новую копию базового массива. При этом любые существующие итераторы могут продолжать перемещения по старым массивам, а как только все ссылки будут потеряны, старая копия массива будет готова для сборки мусора. И вновь такой стиль моментального снимка гарантирует, что исключение ConcurrentModificationException сгенерировано не будет.

Этот компромисс хорошо работает в системах, в которых доступ для чтения такой структуры данных осуществляется гораздо чаще, чем доступ для изменения. Если вы планируете использовать этот подход, внесите изменения только с учетом результатов измерения с помощью хорошего набора тестов.

Защелки и барьеры

Защелки и барьеры являются полезными методиками управления выполнением набора потоков. Например, может быть написана система, в которой рабочие потоки:

1. получают данные от API и анализируют их;
2. записывают результат в базу данных;
3. вычисляют результаты на основе SQL-запроса.

Если бы система просто запускала все потоки, то не было бы никакой гарантии порядка событий. Желаемое действие состоит в том, чтобы разрешить всем потокам выполнить задачу № 1, а затем — задачу № 2 перед запуском задачи № 3. Одна из возможностей для этого — использовать *защелку* (latch). Предположив, что у нас есть пять потоков, мы могли бы написать код следующим образом:

```
package optjava.ch12;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class LatchExample implements Runnable
{
    private final CountDownLatch latch;
    public LatchExample(CountDownLatch latch)
    {
        this.latch = latch;
    }
    @Override
    public void run()
    {
        // Вызов API
        System.out.println(Thread.currentThread().getName() +
                           " Вызов API выполнен");

        try
        {
            latch.countDown();
            latch.await();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName()
                           + " Продолжение работы");
    }
    public static void main(String[] args)
                      throws InterruptedException
    {
        CountDownLatch apiLatch = new CountDownLatch(5);
        ExecutorService pool = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 5; i++)
        {
            pool.submit(new LatchExample(apiLatch));
        }

        System.out.println(Thread.currentThread().getName()
                           + " ожидание в main..");
        apiLatch.await();
        System.out.println(Thread.currentThread().getName()
                           + " ожидание в main завершено..");
```

```
        pool.shutdown();

        try
        {
            pool.awaitTermination(5, TimeUnit.SECONDS);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        System.out.println("Обработка API завершена");
    }
}
```

В этом примере защелка получает значение 5; при этом каждый поток выполняет вызов `countdown()`, уменьшая это число на единицу. Когда счетчик достигнет 0, защелка откроется, и все потоки, удерживаемые функцией `await()`, будут освобождены и продолжат работу.

Важно понимать, что этот тип защелки разовый. Это означает, что как только ее значение станет равно 0, защелка не сможет быть повторно использована; сброс для защелки отсутствует.

Защелки чрезвычайно полезны в таких примерах, как заполнение кеша во время запуска и многопоточное тестирование.

В нашем примере мы могли бы использовать две разные защелки: одну — для завершения обращения к API, а другую — для завершения обращения к базе данных. Еще один вариант — использовать `CyclicBarrier`, который может быть сброшен. Однако выяснение того, какой поток должен управлять сбросом, оказывается довольно сложной задачей и включает другой тип синхронизации. Одной из распространенных наилучших практик является использование одного барьера/защелки для каждого этапа конвейера.

Абстракция исполнителей и заданий

На практике большинство программистов на Java не должны иметь дело с низкоуровневыми проблемами потоков выполнения. Вместо этого стоит подыскать функции `java.util.concurrent`, которые поддерживают параллельное программирование на подходящем уровне абстракции. Например, поддержание потоков в занятом состоянии с использованием некоторых библиотек `java.util.concurrent` обеспечит лучшую производительность потока (т.е. поток поддерживается в рабочем состоянии, а не в заблокированном или в состоянии ожидания).

Уровень абстракции, который предлагает малое количество проблем с потоками, можно описать как *параллельное задание* (concurrent task), т.е. как единицу кода или работы, которая должна быть выполнена одновременно в текущем контексте выполнения. Рассмотрение единиц работы как заданий упрощает написание параллельной программы, так как разработчику не нужно учитывать жизненный цикл для фактических потоков, выполняющих задания.

Введение в асинхронное выполнение

Одним из способов осуществления абстракции задания в Java является использование интерфейса `Callable` для представления задания, которое возвращает значение. Интерфейс `Callable<V>` представляет собой обобщенный интерфейс, определяющий одну функцию, `call()`, которая возвращает значение типа `V` и генерирует исключение в случае, если результат не может быть вычислен. Внешне `Callable` выглядит очень похоже на `Runnable`; однако `Runnable` не возвращает результат и не генерирует исключение.

Если `Runnable` генерирует неперехваченное непроверяемое исключение, оно передается далее по стеку, и по умолчанию выполняемый поток прекращает работу.

Работа с исключениями в течение жизненного цикла потока является сложной программной проблемой и может привести к тому, что при неправильном управлении программа Java продолжит работу в странном состоянии. Следует также отметить, что потоки могут рассматриваться как процессы в стиле операционной системы; это означает, что в некоторых операционных системах их создание может быть достаточно дорогостоящим. Получение любого результата от `Runnable` также может добавить дополнительную сложность, в частности в плане координации возврата выполнения с другим потоком.

Тип `Callable<V>` предоставляет нам способ легко справиться с абстракцией задания, но как эти задания выполняются в действительности?

`ExecutorService` — это интерфейс, который определяет механизм выполнения заданий пулом управляемых потоков. Фактическая реализация `ExecutorService` определяет, как должны управляться потоки в пуле и сколько их должно быть. `ExecutorService` может использовать `Runnable` или `Callable` через метод `submit()` и его перегрузки.

Вспомогательный класс `Executors` имеет ряд фабричных методов `new*`, которые создают службу и поддерживают пул потоков в соответствии с выбранным поведением. Эти фабричные методы — обычное средство создания новых объектов-исполнителей.

`newFixedThreadPool(int nThreads)`

Создает `ExecutorService` с пулом потоков фиксированного размера, в котором потоки будут повторно использоваться для выполнения нескольких заданий. Это позволяет избежать затрат на многократное создание потоков для каждого задания. Когда все потоки используются, новые задачи сохраняются в очереди.

`newCachedThreadPool()`

Создает ExecutorService, который будет создавать новые потоки по мере необходимости и повторно их использовать, если это возможно. Созданные потоки хранятся в течение 60 с, после чего будут удалены из кеша. Использование этого пула потоков может обеспечить лучшую производительность при небольших асинхронных заданиях.

`newSingleThreadExecutor()`

Создает службу `ExecutorService`, поддерживаемую одним потоком. Любые вновь поставленные задания ставятся в очередь в ожидании, пока поток будет доступен. Этот тип исполнителя может быть полезен для управления количеством параллельно выполняемых заданий.

`newScheduledThreadPool(int corePoolSize)`

Имеется ряд дополнительных методов, позволяющих выполнять задание в будущем (которые принимают объект `Callable` и задержку).

Переданное задание будет обрабатываться асинхронно, а передающий код может выбрать блокировку или опрос результата. Вызов `submit()` в `ExecutorService` возвращает объект `Future<V>`, который позволяет вызывать блокирующий метод `get()` или метод `get()` с тайм-аутом, или обычный неблокирующий вызов с использованием `isDone()`.

Выбор `ExecutorService`

Выбор правильного `ExecutorService` обеспечивает хороший контроль асинхронной обработки и может принести значительные преимущества в производительности, если вы выберете правильное количество потоков в пуле.

Можно также написать собственный класс `ExecutorService`, но это необходимо не так уж часто. Одним из способов помощи со стороны библиотеки является возможность настройки путем предоставления объекта `ThreadFactory`. Применение `ThreadFactory` позволяет автору написать код создания настраиваемого потока, который устанавливает свойства потоков, такие как имя, состояние демона и приоритет потока.

Иногда `ExecutorService` требует эмпирической настройки в установках всего приложения. Важной частью общей картины настройки является хорошее знание аппаратного обеспечения, на котором будет работать служба, а также ресурсов, за обладание которыми может возникнуть конкуренция.

Одной из обычно используемых метрик является количество ядер — по сравнению с количеством потоков в пуле. Выбор количества потоков для одновременного запуска, превышающего количество доступных процессоров, может привести к проблемам и конфликтам. Операционная система должна будет планировать выполнение потоков, ведущее к переключению контекстов.

Конфликт, достигнув определенного порогового значения, может отрицательно влиять на производительность параллельной обработки. Вот почему необходимы хорошая модель производительности и возможность измерения увеличения (или уменьшения) эффективности. В главе 5, "Микротесты и статистика", обсуждаются методы тестирования производительности и антипаттерны, которых следует избегать при проведении тестирования такого типа.

Fork/Join

Java предлагает несколько различных подходов к параллелизму, которые не требуют от разработчиков контроля и управления собственными потоками. В Java 7 представлен каркас Fork/Join, который предоставляет новый API для эффективной работы с несколькими процессорами. Он основан на новой реализации `ExecutorService` под названием `ForkJoinPool`. Этот класс предоставляет пул управляемых потоков, который имеет две специальные возможности:

- может использоваться для эффективной обработки подзаданий;
- реализует алгоритм *захвата работы* (work-stealing).

Поддержка подзаданий вводится классом `ForkJoinTask`. Это потокообразная сущность, которая является более легкой, чем стандартный поток Java. Предполагаемый вариант использования основан на том, что потенциально большое количество заданий и подзаданий может размещаться в небольшом количестве реальных потоков в исполнителе `ForkJoinPool`.

Ключевым аспектом `ForkJoinTask` является то, что он может подразделяться на "меньшие" задания, пока размер задания не станет достаточно мал для непосредственного вычисления. По этой причине каркас подходит только для определенных типов заданий, таких как вычисление чистых функций или других "чрезвычайно параллельных" задач. Но даже тогда может потребоваться переписать алгоритмы или код так, чтобы в полной мере воспользоваться преимуществами этой части Fork/Join.

Тем не менее часть алгоритма захвата работы каркаса Fork/Join может использоваться независимо от разбиения задания. Например, если один поток завершил всю выделенную ему работу, а другой отстает, то первый поток захватит работу из

очереди занятого потока. Такое перераспределение работы среди нескольких потоков — довольно простая, но умная идея, приносящая большую пользу. На рис. 12.5 мы видим представление этого алгоритма.

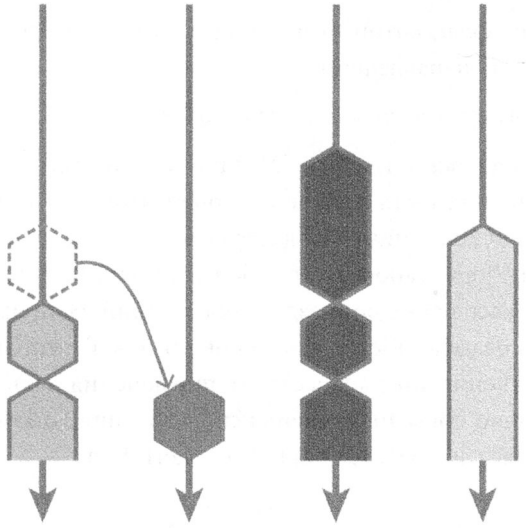

Рис. 12.5. Алгоритм захвата работы

`ForkJoinPool` имеет статический метод `commonPool()`, который возвращает ссылку на общесистемный пул. Это не позволяет разработчикам создавать собственный пул и предоставляет возможность совместного использования общего пула. Инициализация этого общего пула выполняется "ленивым" образом, т.е. он будет создан только в случае необходимости.

Размер пула определяется как `Runtime.getRuntime().availableProcessors()-1`. Однако этот метод не всегда возвращает ожидаемый результат.

Как написал в рассылке Java Specialists[5] Хайнц Кабуц (Heinz Kabutz), он обнаружил случай, когда машина 16-4-2 (16 сокетов, каждый с 4 ядрами и 2 гиперпотоками на ядро) вернула значение 16. Это значение кажется очень низким; наша наивная интуиция, основанная на тестировании ноутбуков, может подсказывать, что следует ожидать значения 16*4*2 = 128. Однако при запуске на этом компьютере Java 8 общий пул Fork/Join был бы сконфигурирован как имеющий параллелизм, равный 15.

> *У VM нет своего мнения о том, с каким процессором она работает; она просто запрашивает соответствующее значение у операционной системы. Точно так же и операционная система обычно ограничивается запросом аппаратного обеспечения. Аппаратное обеспечение возвращает число, которое, как правило,*

[5] `http://www.javaspecialists.eu/`

является количеством "аппаратных потоков". Операционная система верит аппаратному обеспечению. Виртуальная машина верит операционной системе.

— Брайан Гётц (Brian Goetz)

К счастью, имеется флаг, который позволяет разработчику программно установить требуемый уровень параллелизма:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=128
```

Однако, как обсуждалось в главе 5, "Микротесты и статистика", с такими "волшебными" флагами нужно быть крайне осторожными. И, как мы увидим при выборе опции `parallelStream`, ничто не дается бесплатно!

Захват работы Fork/Join становится все более и более используемым разработчиками библиотек и каркасов даже без разделения заданий. Например, каркас Akka, который мы встретим в разделе "Методы на основе актеров", использует `ForkJoinPool` в основном для получения преимуществ от применения захвата работы. Приход Java 8 также значительно повысил уровень использования Fork/Join, так как за кулисами `parallelStream()` использует общий пул Fork/Join.

Параллельность в современном Java

Параллельность в Java первоначально была разработана для среды, в которой долговременные блокируемые задания могли бы чередоваться с другими потоками, чтобы разрешить выполнение последних во время ожидания первых, например операций ввода-вывода и других подобных медленных операций. В настоящее время практически каждая машина, для которой разработчик пишет код, будет многопроцессорной системой, поэтому эффективное использование доступных ресурсов процессора становится очень важной задачей.

Однако, когда понятие параллелизма было встроено в Java, это было не совсем то, с чем индустрия программного обеспечения накопила большой опыт работы. Фактически Java стал первым промышленным стандартом поддержки потоков на уровне языка. В результате именно в Java разработчикам пришлось становиться первопроходцами и оплачивать опыт работы с параллельностью множеством набитых шишек и болезненных уроков. Подход Java в общем случае заключался в том, чтобы не отказываться от имеющихся функциональных возможностей (в особенности фундаментальных), так что Thread API все еще является частью Java и всегда будет таковой.

Это привело к ситуации, когда в современной разработке приложений потоки выполнения представляют собой довольно низкий уровень по сравнению с уровнем абстракции, на котором привыкли писать код программисты на языке программирования Java. Например, в Java никто не занимается ручным управлением памятью, так зачем же программистам Java иметь дело с низкоуровневым созданием потоков и другими событиями их жизненного цикла?

К счастью, современный язык Java предлагает среду, обеспечивающую значительную производительность благодаря абстракциям, встроенным в язык и в стандартную библиотеку. Это позволяет разработчикам получать преимущества от параллельного программирования с меньшим количеством разочарований и меньшим количеством шаблонов.

Потоки данных и параллельные потоки данных

Безусловно, самым большим изменением в Java 8 (возможно, самым большим изменением в языке вообще) стало введение лямбда-выражений и потоков данных (stream). Используемые вместе лямбда-выражения и потоки предоставляют своего рода "волшебную палочку", которая позволяет разработчикам на языке Java получить доступ к некоторым преимуществам функционального стиля программирования.

Оставляя в стороне довольно сложный вопрос о том, насколько Java 8 на самом деле является функциональным языком, можно сказать, что теперь Java имеет новую парадигму программирования. Этот более функциональный стиль включает сосредоточение на данных, а не на императивном объектно-ориентированном подходе, которым он всегда отличался.

Поток данных в Java — это неизменяемая последовательность элементов данных, которая передает элементы из источника данных. Поток может быть из любого источника (коллекции, ввода-вывода) типизированных данных. Мы работаем с потоками, используя операции для манипуляции данными, такие как `map()`, которые для управления данными принимают лямбда-выражения или функциональные объекты. Этот переход от внешней итерации (традиционные циклы `for`) ко внутренней итерации (потокам) дает нам некоторые приятные возможности параллелизации данных и отложенного вычисления сложных выражений.

В настоящее время все коллекции предоставляют метод `stream()` из интерфейса `Collection`. Это метод по умолчанию, который предоставляет реализацию для создания потока из любой коллекции (за кулисами при этом создается `ReferencePipeline`).

Второй метод, `parallelStream()`, может использоваться для параллельной работы с элементами данных и рекомбинации результатов. Использование `parallelStream()` предполагает разделение работы с использованием `Spliterator` и выполнение вычислений в общем пуле Fork/Join. Это удобная техника для работы с чрезвычайно параллельными задачами, потому что элементы потока исходно неизменяемые, а потому позволяют избежать проблем с изменением состояния при параллельной работе.

Введение потоков привело к синтаксически более дружественному способу работы с Fork/Join, чем кодирование с использованием `RecursiveAction`. Выражение задач в терминах данных подобно абстрагированию задач в том, что помогает

разработчику избежать необходимости рассматривать низкоуровневую механику потоков и проблемы с изменяемыми данными.

Может возникнуть соблазн использовать `parallelStream()` всегда и везде, но при этом подходе есть определенные затраты. Как и при любом параллельном вычислении, необходимо выполнить работу, чтобы разделить задание между несколькими потоками, а затем рекомбинировать результаты — непосредственный пример применения закона Амдала.

При небольших коллекциях последовательные вычисления в действительности могут оказаться значительно более быстрыми. При использовании `parallelStream()` всегда следует соблюдать осторожность и выполнять проверки производительности. Правило "измеряй, а не гадай" применимо здесь в полной мере. В смысле использования параллельных потоков для повышения производительности преимущество должно быть непосредственным и измеримым, поэтому не конвертируйте бездумно последовательные потоки в параллельные.

Методы, свободные от блокировок

Технологии без блокировок исходят из предпосылки, что применение блокировок плохо сказывается на пропускной способности и может ухудшить производительность. Проблема с блокировкой заключается в том, что она дает возможность показать операционной системе возможность переключения контекста потока.

Рассмотрим приложение с двумя потоками, `t1` и `t2`, на двухъядерном компьютере. Сценарий блокировки может привести к тому, что при переключении контекста потоки могут переходить для выполнения на другой процессор. Кроме того, время, затрачиваемое на паузу и пробуждение, может оказаться значительным, так что применение блокировки может привести к намного более медленной работе, чем технология, свободная от блокировок.

Один из современных проектных шаблонов, который подчеркивает потенциальные выигрыши в производительности параллелизма без блокировок, — шаблон Разрушитель (Disruptor), первоначально представленный London Multi Asset Exchange (LMAX). По сравнению с `ArrayBlockingQueue` шаблон Disruptor превосходит его на порядки. На странице проекта в GitHub[6] показаны некоторые из проведенных сравнений. В табл. 12.1 представлены данные о производительности из одного из примеров, приведенных на этой странице.

Эти удивительные результаты достигнуты с помощью спин-блокировки. Синхронизация между двумя потоками эффективно управляется вручную посредством `volatile`-переменной (для гарантии видимости потоками):

```
private volatile int proceedValue;
//...
```

[6] `https://github.com/LMAX-Exchange/disruptor/wiki/Performance-Results`

```
while (i != proceedValue)
{
    // Цикл занятости
}
```

Таблица 12.1. Статистика производительности LMAX: пропускная способность в операциях в секунду

	Блокирующая очередь массивов	Disruptor
Unicast: 1P–1C	5 339 256	25 998 336
Pipeline: 1P–3C	2 128 918	16 806 157
Sequencer: 3P–1C	5 539 531	13 403 268
Multicast: 1P–3C	1 077 384	9 377 871

Поддержание выполнения цикла ядром процессора означает, что после получения данных с ними сразу же можно начинать работать на этом ядре без необходимости переключения контекста.

Разумеется, методы без применения блокировок имеют свою цену. Занимая время процессорного ядра, эти методы дорогостоящи, например, с точки зрения использования и потребления энергии: процессор очень занят и выполняет массу работы, ничего не делая по сути, — но "очень занят" влечет за собой "более горячий", а это означает большие расходы энергии для охлаждения ядра, которое не делает ничего полезного.

Выполнение приложений, нуждающихся в такой пропускной способности, часто требует от программиста хорошего понимания низкоуровневых последствий применения соответствующего программного обеспечения. Оно должно быть дополнено механическим взаимопониманием того, как код будет взаимодействовать с оборудованием. Не случайно термин *механическое взаимопонимание* (mechanical sympathy) был придуман Мартином Томпсоном (Martin Thompson), одним из создателей проектного шаблона Disruptor.

> *Термин "механическое взаимопонимание" происходит от великого гонщика Джеки Стюарта (Jackie Stewart), который три раза был чемпионом мира в Формуле-1. Он считал, что, чтобы работать в гармонии с машиной, водителю необходимо понимать, как она работает.*
>
> *— Мартин Томпсон (Martin Thompson)*

Методы на основе актеров

В последние годы появилось несколько разных подходов к представлению заданий, которые несколько *меньше* потока. Мы уже встречались с этой идеей в классе `ForkJoinTask`. Другой популярный подход — *парадигма актера* (actor paradigm).

Актеры представляют собой небольшие автономные единицы обработки, которые содержат собственное состояние, имеют собственное поведение и включают систему почтовых ящиков для общения с другими актерами. Актеры управляют проблемой состояния, но не используя совместно какое-либо изменяющееся состояние, а только обмениваясь один с другим посредством неизменяемых сообщений. Связь между актерами является асинхронной, и актеры реагируют на получение сообщений для выполнения своего конкретного задания.

Образуя сеть, в которой каждый из них имеет конкретные задания в рамках параллельной системы, актеры полностью абстрагируются от базовой модели параллелизма.

Актеры могут жить в пределах одного и того же процесса, но не обязаны это делать. Это предоставляет немалое преимущество, заключающееся в том, что актерские системы могут быть многопроцессорными и потенциально даже охватывать несколько машин. Наличие множества машин и кластеризация позволяют эффективно работать системам на базе актеров, от которых требуется высокая степень отказоустойчивости. Чтобы обеспечить успешную работу актеров в коллаборативной среде, обычно применяется стратегия fail-fast[7].

Для языков на основе JVM популярным каркасом для разработки актерских систем является Akka. Он написан на Scala, но также имеет Java API, что делает его пригодным для использования с Java и другими языками JVM.

Мотивация Akka и системы на основе актеров базируется на нескольких проблемах, затрудняющих параллельное программирование. Документация Akka[8] выделяет три основные мотивации для рассмотрения использования Akka вместо традиционных схем блокировки:

- инкапсуляция изменяемого состояния в рамках модели предметной области может быть сложной, особенно если ссылка на внутреннее представление объекта может быть бесконтрольно доступной извне;
- защита состояния с использованием блокировок может привести к существенному снижению пропускной способности;
- блокировки могут привести к взаимоблокировке и другим проблемам.

Упомянутые дополнительные проблемы включают трудность правильного использования совместно используемой памяти и проблемы с производительностью, которые могут возникнуть, когда строки кеша совместно используются несколькими процессорами.

---

[7] "Быстрый провал" — немедленное сообщение об ошибке вместо продолжения работы. — *Примеч. пер.*

[8] `http://doc.akka.io/docs/akka/2.5/java/guide/actors-motivation.html`

Последняя из обсуждаемых мотиваций связана с ошибками в традиционных моделях потоков и стеках вызовов. В низкоуровневом потоковом API нет стандартного способа обработки сбоя потока или восстановления после сбоя. Akka стандартизирует этот момент и предоставляет разработчику точно определенную схему восстановления.

В целом модель актера может быть полезным дополнением к набору инструментария разработчика параллельных приложений. Однако она не является заменой общего назначения для всех других методик. Если для конкретной ситуации эта модель применима (асинхронная передача неизменяемых сообщений, изменяемое состояние, не используемое совместно, или ограниченное по времени выполнение каждого процессора сообщений), то это может оказаться отличным быстрым решением. Если же дизайн системы включает в себя синхронную обработку запросов и ответов, совместно используемое изменяемое состояние или неограниченное выполнение, то аккуратные разработчики могут выбрать другую абстракцию для построения своих систем.

Резюме

В этой главе только поверхностно затронуты те темы, которые вы должны рассмотреть, прежде чем планировать повышение производительности приложений с помощью многопоточности. При преобразовании однопоточного приложения в параллельный дизайн:

- убедитесь, что производительность простой линейной обработки может быть точно измерена;
- внесите изменения и протестируйте, действительно ли производительность увеличилась;
- убедитесь, что тесты производительности легко выполнить повторно, особенно если объем данных, обрабатываемых системой, может измениться.

Избегайте соблазна:

- везде использовать параллельные потоки данных;
- создавать сложные структуры данных с ручной блокировкой;
- заново изобретать структуры, уже имеющиеся в `java.util.concurrent`.

Стремитесь:

- повысить производительность горячих потоков с использованием параллельных коллекций;
- применять конструкции доступа, которые используют преимущества базовых структур данных;

- уменьшать количество блокировок в приложении;
- предоставлять подходящие задания/асинхронные абстракции для предотвращения ручной работы с потоками.

Параллелизм является ключом к будущему высокопроизводительному коду. Однако:

- совместно использовать изменяемые состояния трудно;
- блокировки трудно использовать правильно;
- необходимы модели как синхронизированного, так и асинхронного совместного использования состояний;
- JMM представляет собой низкоуровневую гибкую модель;
- абстракция потока крайне низкоуровневая.

Тенденцией современного параллелизма является переход к высокоуровневой модели параллелизма и отход от потоков, которые все чаще выглядят как "ассемблерный язык параллелизма". Последние версии Java увеличили количество классов и библиотек более высокого уровня, доступных для программиста. В целом программирование переходит к модели параллелизма, в которой гораздо больше ответственности за безопасные параллельные абстракции несет среда выполнения и библиотеки.

ГЛАВА 13

Профилирование

Термин *профилирование* (profiling) среди программистов используется с разными смыслами. Фактически есть несколько возможных подходов к профилированию, из которых наиболее распространенными являются следующие два:

- выполнение;
- выделение памяти.

В этой главе мы рассмотрим обе эти темы. Наше основное внимание будет сосредоточено на профилировании выполнения, и именно его мы используем для ознакомления с инструментами, доступными для профилирования приложений. Ниже в этой главе мы расскажем о профилировании памяти и рассмотрим, как различные инструменты предоставляют эту возможность.

Одной из ключевых тем, которые мы изучим, является важность понимания разработчиками на языке Java и инженерами по производительности того, как вообще работают профайлеры. Профайлеры могут легко неверно интерпретировать поведение приложения и демонстрировать заметные искажения.

Профилирование выполнения является одной из областей анализа производительности, в которой эти искажения выходят на первый план. Осторожный инженер по производительности знает об этой возможности и будет компенсировать ее различными способами, включая профилирование несколькими инструментами, чтобы понять, что происходит на самом деле.

Для инженеров одинаково важно справляться со своими когнитивными искажениями и не искать подтверждений ожидаемого поведения производительности. Антипаттерны и когнитивные ловушки, которые мы рассматривали в главе 4, "Паттерны и антипаттерны тестирования производительности", являются хорошей отправной точкой для борьбы с этими проблемами.

Введение в профилирование

В общем случае средства профилирования и мониторинга JVM работают с использованием некоторого низкоуровневого инструментария и передают данные обратно на графическую консоль или сохраняют их в журнале для последующего

анализа. Низкоуровневый инструментарий обычно принимает форму либо агента, загружаемого при запуске приложения, либо компонента, который динамически подключается к работающей JVM.

С агентами мы встречались в разделе "Мониторинг и инструментарий JVM" главы 2, "Обзор JVM"; это очень обобщенная технология, широко применяемая в инструментах Java.

В широком смысле нам нужно различать *инструменты мониторинга* (monitoring tools, основная цель которых — наблюдение за системой и ее текущим состоянием), *системы оповещения* (alerting systems, используемые для обнаружения аномального поведения) и *профайлеры* (profilers, которые предоставляют глубокую информацию в запущенном приложении). Эти инструменты имеют разные, хотя и часто взаимосвязанные цели, и хорошо продуманное производственное приложение может использовать их все.

Однако в центре внимания этой главы находится профилирование. Цель профилирования (выполнения) — идентифицировать пользовательский код, к которому следует применить рефакторинг и оптимизации производительности.

Профилирование обычно достигается путем присоединения пользовательского агента для выполнения приложения JVM.

Как обсуждалось в разделе "Основные стратегии обнаружения источников проблем" главы 3, "Аппаратное обеспечение и операционные системы", первым шагом в диагностике и исправлении проблемы производительности является определение того, какой именно ресурс вызывает эту проблему. Неправильная идентификация на этом этапе может очень дорого стоить впоследствии.

Проведя анализ профилирования приложения, который не ограничивается тактами процессора, очень легко неверно трактовать результаты работы профайлера. Вспомним цитату Брайана Гётца (Brian Goetz) из раздела "Введение в JMH" главы 5, "Микротесты и статистика", о том, что инструменты всегда производят какие-то числа — просто часто неясно, что некоторое конкретное число имеет отношение к рассматриваемой проблеме. Именно по этой причине мы, ознакомившись с основными типами искажений в разделе "Когнитивные искажения и тестирование производительности" главы 4, "Паттерны и антипаттерны тестирования производительности", отложили обсуждение методов профилирования до данной главы.

> *Хороший программист... будет достаточно мудр, чтобы внимательно рассмотреть критический код; но только после того, как этот код будет найден.*
>
> *— Дональд Кнут (Donald Knuth)*

Это означает, что перед проведением профилирования инженеры должны идентифицировать проблему с производительностью. Кроме того, они должны доказать, что виноват в этом код приложения. Они будут точно знать, что это так, если приложение потребляет почти 100% времени процессора в пользовательском режиме.

Если эти критерии не выполняются, инженер должен искать источник проблемы в другом месте и не должен пытаться диагностировать проблему с помощью профайлера выполнения.

Даже если процессор в пользовательском режиме (не в режиме ядра) полностью загружен, существует еще одна возможная причина, которая должна быть исключена перед профилированием: этап полной приостановки приложения при сборке мусора. Поскольку все приложения, которые серьезно озабочены производительностью, должны регистрировать события сборки мусора, указанная проверка достаточно проста: проконсультируйтесь с журналами сборки мусора и приложения на машине и убедитесь, что журнал сборки мусора "молчит", а журнал приложения демонстрирует активность. Если же журнал сборки мусора активен, то следующим шагом должна быть настройка сборки мусора, а не профилирование выполнения.

Выборка и искажение точек безопасности

Одним из ключевых аспектов профилирования выполнения является то, что оно обычно использует выборку для получения данных (трассировки стека) о том, какой код работает в этот момент. Проведение измерений имеет свою стоимость, так что, чтобы предотвратить чрезмерную стоимость сбора данных, все входы в методы и выходы из них обычно не отслеживаются. Вместо этого делается моментальный снимок выполнения потока, но без неприемлемых накладных расходов он может делаться только относительно редко.

Например, New Relic Thread Profiler (один из инструментов, доступных как часть стека New Relic) выполняет выборку каждые 100 мс. Этот предел часто рассматривается как наилучшее предположение о том, как часто можно выполнять выборку, не вызывая высоких накладных расходов.

Интервал выборки для инженера в области производительности представляет собой определенный компромисс. Стоит осуществлять выборку слишком часто, как накладные расходы становятся неприемлемыми, в особенности для приложения, которое заботится о производительности. С другой стороны, слишком редкая выборка слишком сильно увеличивает вероятность пропустить важное поведение, поскольку такая редкая выборка может не отражать реальную производительность приложения.

> *К тому времени, когда вы используете профилировщик, остается только выяснить некоторые детали — и это не должно вас удивлять.*
>
> *— Кирк Пеппердайн (Kirk Pepperdine)*

Выборка предоставляет не только возможности для сокрытия данных о проблеме; в большинстве случаев выборка выполняется только в точках безопасности. Это явление известно как *искажение точек безопасности* (safepointing bias) и имеет два основных последствия:

- перед тем как будет осуществлена выборка, все потоки должны достичь точек безопасности;
- собираемая информация описывает состояние приложения только в точках безопасности.

Первое из них налагает дополнительные накладные расходы на создание профилирующей выборки из выполняемого процесса. Второе последствие сводит на нет распределение точек выборки, так как выборка выполняется только в состояниях, о которых известно, что они являются точками безопасности.

Большинство профайлеров выполнения для сбора выборок стека каждого потока приложений используют функцию `GetCallTrace()` из C++ API HotSpot. Обычный дизайн состоит в том, чтобы собирать образцы внутри агента, а затем протоколировать данные или выполнять иную последующую обработку.

Однако функция `GetCallTrace()` обладает довольно большими накладными расходами: если имеется *N* активных потоков приложения, то сбор образцов стеков приводит к *N* достижениям JVM точек безопасности. Эти накладные расходы являются одной из основных причин, по которым устанавливается верхний предел частоты, с которой могут выполняться выборки.

Поэтому внимательный инженер в области производительности будет следить за тем, сколько времени используется приложением в точках безопасности. Если в них затрачено слишком много времени, будет страдать производительность приложения, и любое действие по настройке производительности может быть вызвано неточными данными. Вот флаг JVM, который может быть очень полезен для отслеживания случаев с высоким временем в точках безопасности:

```
-XX:+PrintGCApplicationStoppedTime
```

Этот фаг приведет к записи дополнительной информации о времени в точках безопасности в журнал сборки мусора. Некоторые инструменты (в частности, jClarity Censum) могут автоматически обнаруживать проблемы на основе данных, полученных при использовании этого флага. Censum также может различать время в точках безопасности и время паузы, вызванное ядром операционной системы.

Один из примеров проблем, вызванных искажением точек безопасности, можно проиллюстрировать *циклом со счетчиком* (counted loop). Это простой цикл, аналогичный использованному в следующем фрагменте:

```
for (int i = 0; i < LIMIT; i++)
{
```

```
    // Только "простые" операции в теле цикла
}
```

Мы преднамеренно не поясняем смысл "простой" операции в этом примере, поскольку ее поведение зависит от точной оптимизации, которую может выполнять JIT-компилятор. Более подробную информацию можно найти в разделе "Разворачивание циклов" главы 10, "JIT-компиляция".

Примеры простых операций включают арифметические операции над примитивными типами и вызовы методов, которые полностью встроены (так что никакие вызовы методов фактически внутри тела цикла не присутствуют).

Если значение `LIMIT` велико, компилятор JIT транслирует этот Java-код непосредственно в эквивалентную скомпилированную форму, включая обратную ветвь для возврата в начало цикла. Как обсуждалось в разделе "Еще раз о точках безопасности" главы 10, "JIT-компиляция", JIT-компилятор добавляет проверки точек безопасности в обратные ветви. Это означает, что для большого цикла возможность достижения точки безопасности предоставляется по одному разу на итерацию цикла.

Однако для достаточно малого значения `LIMIT` это не произойдет, и вместо этого JIT-компилятор развернет этот цикл. Это означает, что поток, выполняющий цикл с достаточно малым количеством итераций, не достигнет точки безопасности до тех пор, пока цикл не завершится.

Таким образом, выборка только в точках безопасности ведет прямо к искажающему поведению, которое существенно зависит от размера циклов и характера операций, которые в них выполняются.

Это, безусловно, не идеальный подход для получения строгих и надежных результатов. Но это и не теоретическая проблема — развертывание цикла может сгенерировать значительное количество кода, что приведет к большим фрагментам кода с длительным временем выполнения, в которых не будут выполнены никакие выборки.

Мы еще вернемся к проблеме искажения точек безопасности, но она остается прекрасным примером тех ограничений, о которых должны быть осведомлены специалисты в области производительности.

Инструменты профилирования выполнения для разработчиков

В этом разделе мы обсудим несколько различных инструментов профилирования выполнения с графическими интерфейсами. На рынке имеется довольно много инструментов, поэтому мы ограничимся рассмотрением наиболее распространенных, не пытаясь выполнить исчерпывающий обзор.

Профайлер VisualVM

В качестве первого примера инструмента профилирования рассмотрим VisualVM[1]. Он включает в себя профайлер как выполнения, так и памяти, и представляет собой очень простой инструмент. Он весьма ограничен в том смысле, что его редко можно использовать в качестве производственного инструмента, но может быть полезен инженерам в области производительности, которые хотят понять поведение своих приложений в средах разработки и контроля качества.

На рис. 13.1 мы видим, как выглядит представление профилирования выполнения в VisualVM.

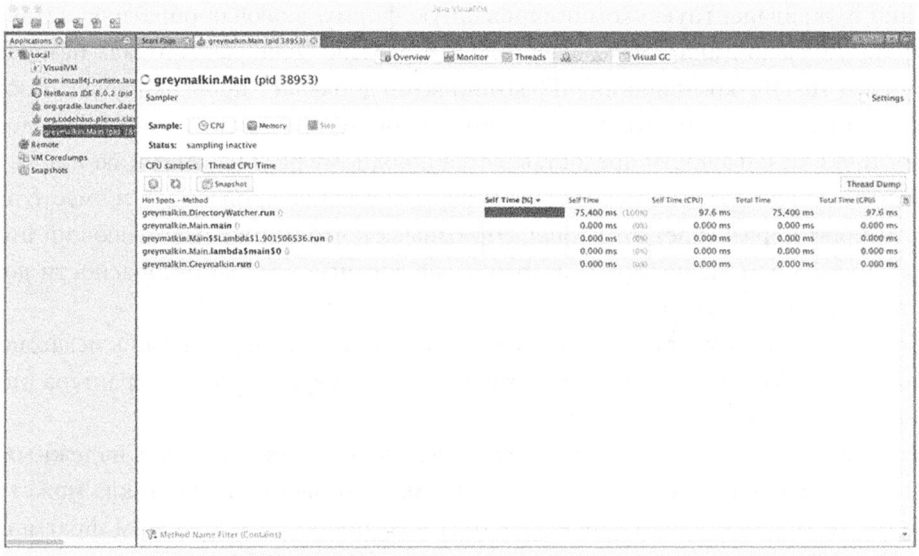

Рис. 13.1. Профайлер выполнения VisualVM

Здесь показано простое представление выполнения методов и относительное потребление ими времени процессора. Объем развертки, доступный в VisualVM, фактически очень ограничен. В результате большинство инженеров очень быстро перерастают этот инструмент и обращаются к одному из более серьезных инструментов, имеющихся на рынке. И тем не менее VisualVM может быть полезным первым инструментом для тех инженеров, которые пока что являются новичками в области профилирования.

JProfiler

Одним популярным коммерческим профайлером является JProfiler от ej-technologies GmbH. Это профайлер на основе агентов, способный работать как в

[1] `https://visualvm.github.io/`

режиме инструмента с графическим интерфейсом пользователя, так и без него, для профилирования локальных или удаленных приложений. Он совместим с довольно широким спектром операционных систем, включая такие, как FreeBSD, Solaris и AIX, а также более обычные Windows, macOS и Linux.

Когда приложение для настольных компьютеров запускается в первый раз, отображается экран, подобный показанному на рис. 13.2.

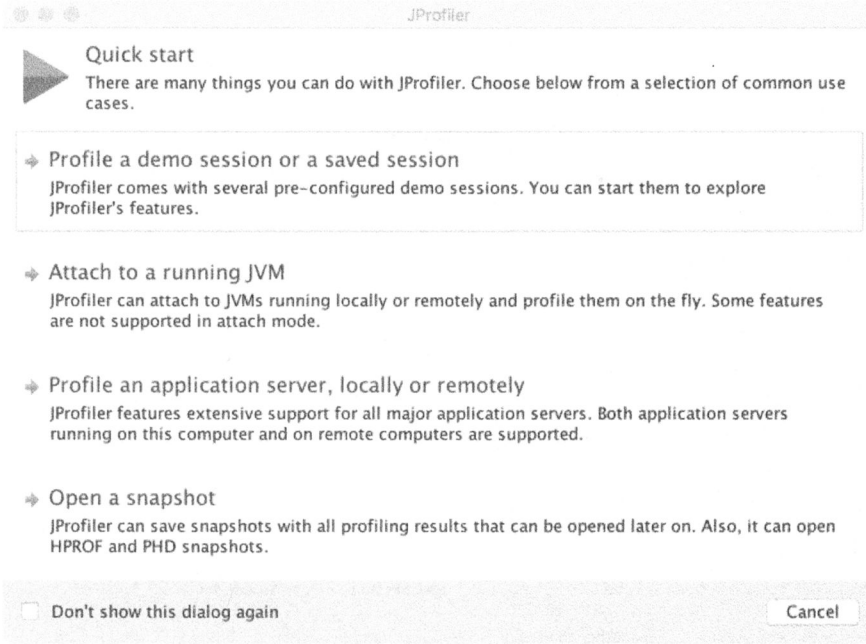

Рис. 13.2. Мастер настройки JProfiler

Очистка этого экрана дает представление по умолчанию, показанное на рис. 13.3.

Щелчок на кнопке Start Center в левом верхнем углу дает множество вариантов действий, в том числе Open Session (Открыть сеанс, который содержит некоторые предварительные примеры для работы) и Quick Attach (Быстрое подключение). На рис. 13.4 показан вариант Quick Attach, при котором мы выбираем профилирование приложения AsciidocFX (оно является авторским инструментом, с помощью которого написана большая часть этой книги).

Подключение к целевой JVM вызывает диалоговое окно конфигурации, показанное на рис. 13.5. Обратите внимание, что профайлер уже в окне конфигурации предупреждает о влиянии на производительность, а также о необходимости эффективных фильтров и осведомленности о накладных расходах профилирования. Как обсуждалось ранее в этой главе, профилирование выполнения — очень сложный процесс, и, чтобы избежать неприятностей, инженер должен действовать крайне осторожно.

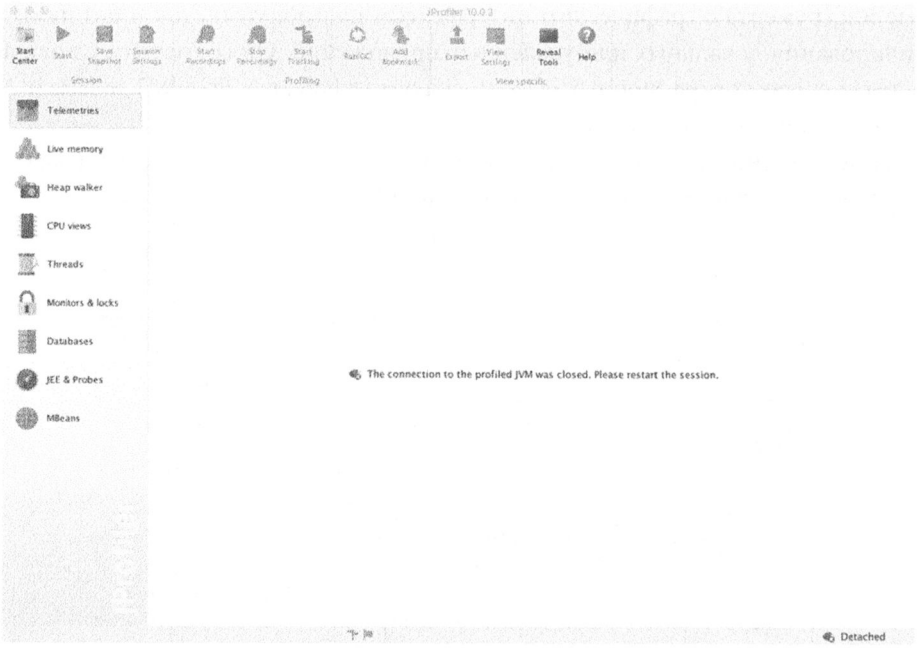

Рис. 13.3. Начальный экран JProfiler

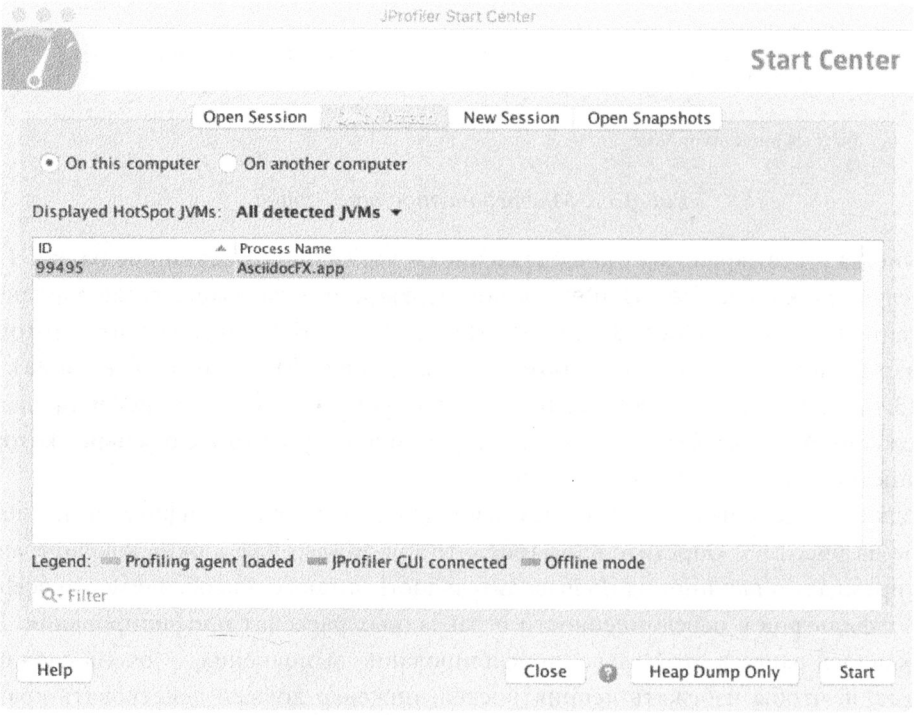

Рис. 13.4. Экран Quick Attach *профайлера JProfiler*

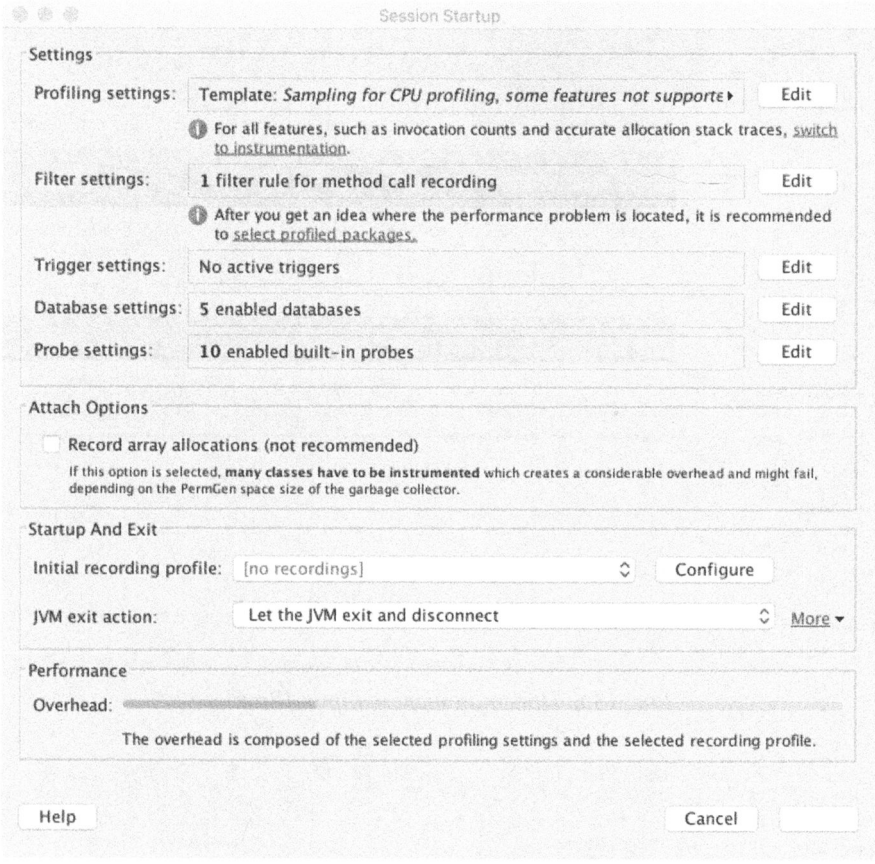

Рис. 13.5. Конфигурация подключения JProfiler

После первоначального сканирования JProfiler возвращается к жизни. На начальном экране показано представление телеметрии, аналогичное представлению VisualVM, но со скроллируемым, а не с меняющим размеры со временем окном. Это представление показано на рис. 13.6.

Из этого экрана доступны все основные представления, но без включения некоторых записей увидеть можно не так уж много. Чтобы просмотреть хронометраж методов, выберите представление Call Tree (Дерево вызовов) и щелкните на кнопке для начала записи. Через несколько секунд начнут появляться результаты профилирования. Это должно выглядеть примерно так, как показано на рис. 13.7.

Древовидное представление можно развернуть, чтобы показать внутреннее время выполнения методов, которые вызывают другие методы.

Чтобы использовать агент JProfiler, добавьте в конфигурацию запуска следующий переключатель:

```
-agentpath:<path-to-agent-lib>
```

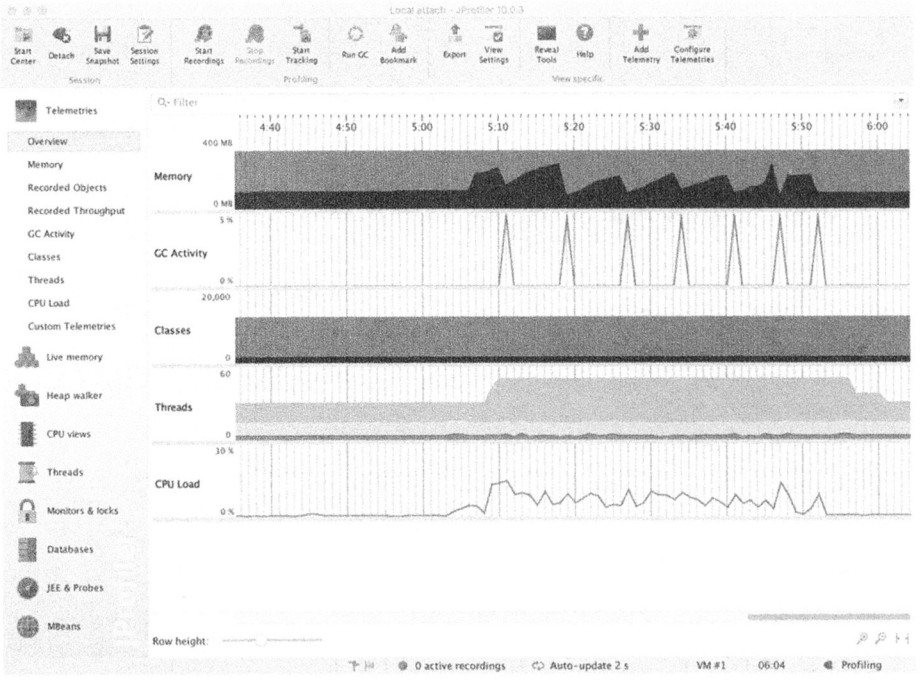

Рис. 13.6. Простая телеметрия JProfiler

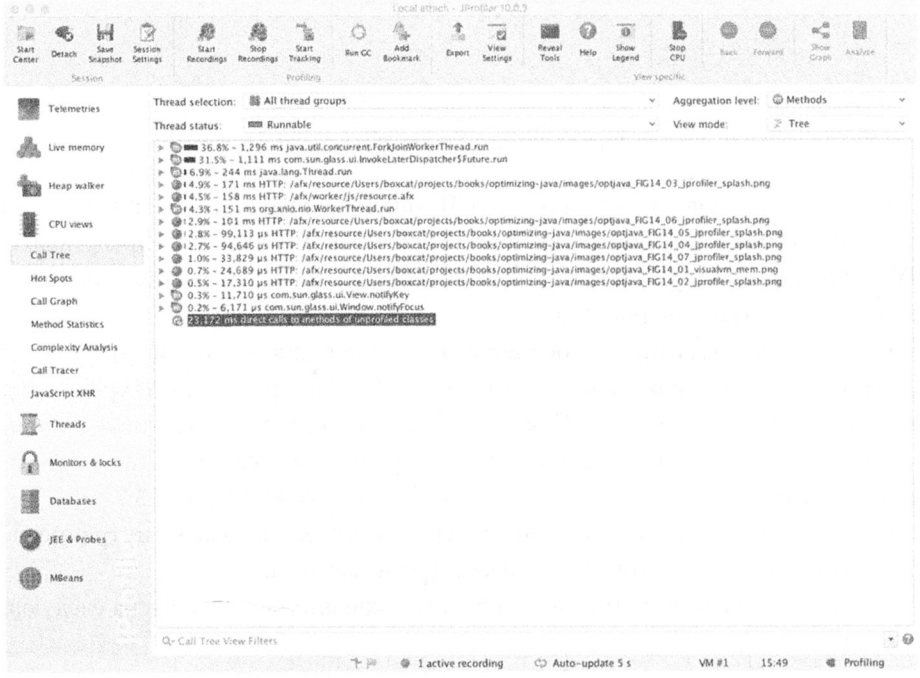

Рис. 13.7. Хронометраж работы процессора в JProfiler

В конфигурации по умолчанию это приведет к приостановке профилированного приложения при запуске и ожидание подключения графического интерфейса пользователя. Смысл заключается в том, чтобы загружать инструментальные средства классов приложений во время запуска, чтобы после этого приложение нормально работало.

В случае с производственными приложениями подключение графического интерфейса пользователя выглядело бы не нормально. В этом случае лучше добавить агент JProfiler с конфигурацией, которая указывает, какие данные следует записывать. Эти результаты сохраняются только в файлах моментальных снимков, которые будут загружены в графический интерфейс позже. JProfiler предоставляет специальный мастер для настройки удаленного профилирования и соответствующие настройки для добавления к удаленной JVM.

Наконец, внимательный читатель должен заметить, что на копиях экрана мы показываем профиль приложения графического интерфейса, которое не привязано к процессору, поэтому показанные здесь результаты приведены исключительно для демонстрационных целей. Процессор нигде не используется с нагрузкой, близкой к 100%, поэтому здесь мы видим нереалистичный вариант использования JProfiler (или любого иного инструмента профилирования).

YourKit

Профайлер YourKit[2] — это еще один коммерческий профайлер, созданный YourKit GmbH. Инструмент YourKit в определенной степени похож на JProfiler, предлагая компонент с графическим интерфейсом пользователя, а также агент, который может либо подключаться динамически, либо настраиваться для работы при запуске приложения.

Чтобы развернуть агент, используйте следующий синтаксис (для 64-разрядной Linux):

```
-agentpath:<каталог-профайлера>/bin/linux-x86-64/libyjpagent.so
```

С точки зрения графического интерфейса он имеет настройки и начальные экраны телеметрии, аналогичные тем, которые мы видели в VisualVM и JProfiler.

На рис. 13.8 мы можем видеть представление моментального снимка процессора с подробным описанием того, на что процессор тратит свое время. Этот уровень детализации выходит далеко за рамки возможностей VisualVM.

При тестировании режим подключения YourKit иногда выводит некоторые ошибки, такие как замораживание приложений с графическим интерфейсом пользователя. Однако в целом возможности профилирования выполнения YourKit в целом соответствуют тем, которые предлагаются в JProfiler; некоторые инженеры могут

[2] https://www.yourkit.com/

просто выбрать один или другой инструмент, исходя из своих личных предпочтений, а не их функциональных возможностей.

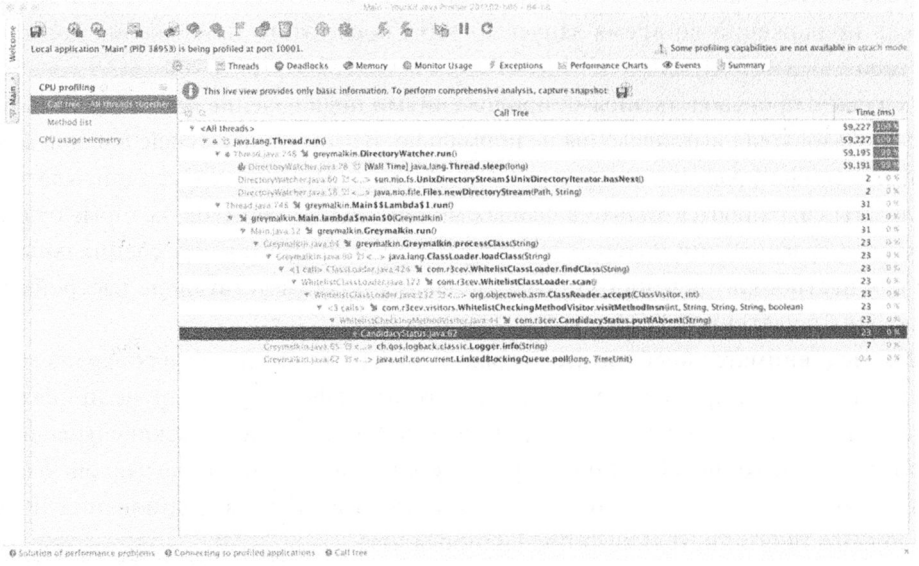

Рис. 13.8. Хронометраж процессора в YourKit

Если это возможно, то имеет смысл профилирование с помощью как YourKit, так и JProfiler (хотя и не одновременно, так как это привело бы к дополнительным накладным расходам), которое может выявить различные представления приложения, могущие быть полезными для диагностики.

Оба инструмента используют подход, основанный на выборке в точках безопасности, обсуждавшийся ранее; поэтому оба инструмента потенциально подвержены одним и тем же типам ограничений и искажений, вызываемых этим подходом.

Flight Recorder и Mission Control

Инструменты Java Flight Recorder и Mission Control (JFR/JMC)[3] — это технологии профилирования и мониторинга, полученные Oracle в рамках приобретения BEA Systems. Ранее они были частью инструментального предложения для JRockit JVM от BEA. Эти инструменты были перенесены в коммерческую версию Oracle JDK в рамках процесса завершения поддержки JRockit.

Начиная с Java 8 Flight Recorder и Mission Control являются коммерческими патентованными инструментами. Они доступны только для JVM Oracle и не будут работать со сборками OpenJDK или с любой другой JVM.

[3] `https://docs.oracle.com/javacomponents/index.html`

Поскольку JFR доступен только для Oracle JDK, вы должны передать следующие ключи при запуске JVM Oracle с Flight Recorder:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

В сентябре 2017 года Oracle объявила о существенном изменении графика выпуска Java, изменив его цикл с двухлетнего на шестимесячный. Это стало результатом того, что две предыдущие версии (Java 8 и 9) значительно задержались.

В дополнение к решению об изменении продолжительности цикла выпуска Oracle также объявила, что после Java 9 основным JDK, распространяемым Oracle, станет OpenJDK, а не Oracle JDK. В рамках этого изменения Flight Recorder и Mission Control станут инструментами с открытым исходным кодом, которые после этого смогут свободно использоваться.

На момент написания книги детальный план доступности JFR/JMC в качестве бесплатных пакетов с открытым исходным кодом объявлен не был. Также не было объявлено, потребуется ли развертывание Java 8 или 9 для оплаты производственного применения JFR/JMC.

Первоначальная инсталляция JMC включает консоль JMX и JFR, хотя в Mission Control можно легко установить дополнительные подключаемые модули.

JMC представляет собой графический компонент, запускаемый бинарным файлом `jmc` из каталога `$JAVA_HOME/bin`. Начальный экран Mission Control показан на рис. 13.9.

Рис. 13.9. Начальный экран JMC

Для профилирования Flight Recorder должен быть включен в целевом приложении. Вы можете сделать это либо путем включения флагов, либо путем динамического подключения после того, как приложение уже запущено.

После подключения введите конфигурацию сеанса записи и события профилирования, как показано на рис. 13.10 и 13.11.

Рис. 13.10. Настройка записи JMC

После начала записи обычно выводится окно времени, показанное на рис. 13.12.

Для поддержки JFR от JRockit виртуальная машина HotSpot была оснащена инструментами для создания большого набора счетчиков производительности, аналогичных собранным в Serviceability Agent.

Эксплуатационные инструменты

Профайлеры по своей природе являются инструментами разработчика для диагностики проблем или для понимания поведения приложений на низком уровне. На

другом конце инструментального спектра находятся инструменты эксплуатационного мониторинга. Они призваны помочь команде визуализировать текущее состояние системы и определить, как работает система: нормально или аномально.

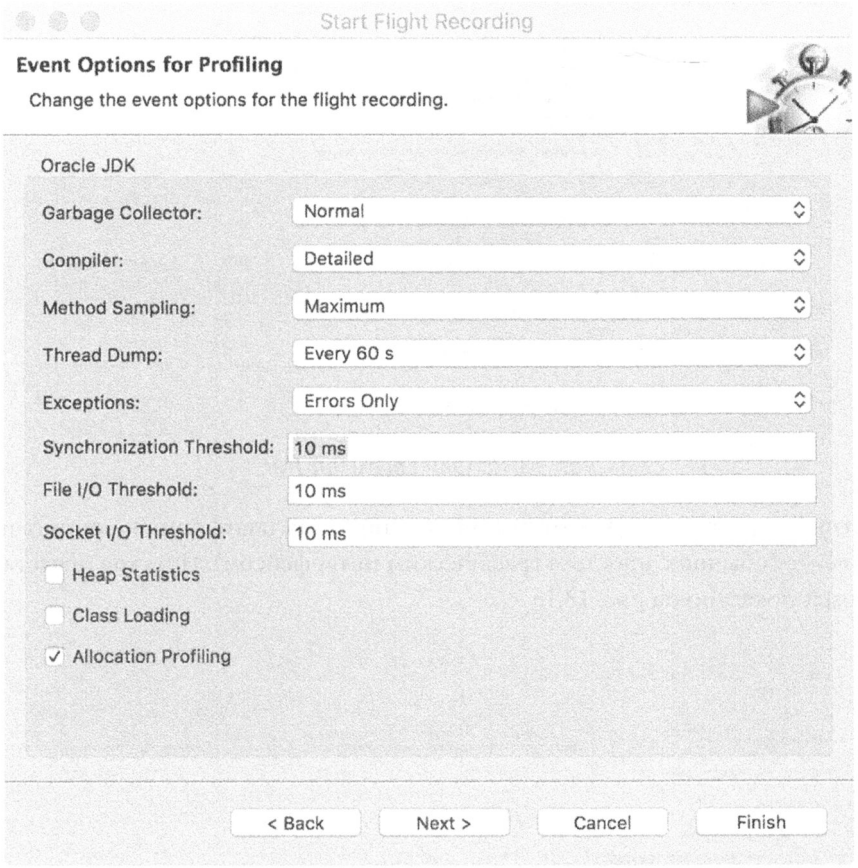

Рис. 13.11. Настройки события профилирования JMC

Это огромная тема, полное обсуждение которой выходит далеко за рамки этой книги. Мы ограничимся кратким рассмотрением трех инструментов — двух коммерческих и одного с открытыми исходными текстами.

Red Hat Thermostat

Thermostat[4] — это решение Red Hat с открытым исходным кодом для обеспечения работоспособности и мониторинга JVM на базе HotSpot. Он доступен с той же лицензией, что и OpenJDK, и обеспечивает мониторинг как отдельных машин, так и кластеров. Для хранения информации он использует базу данных MongoDB.

[4] `http://icedtea.classpath.org/thermostat/`

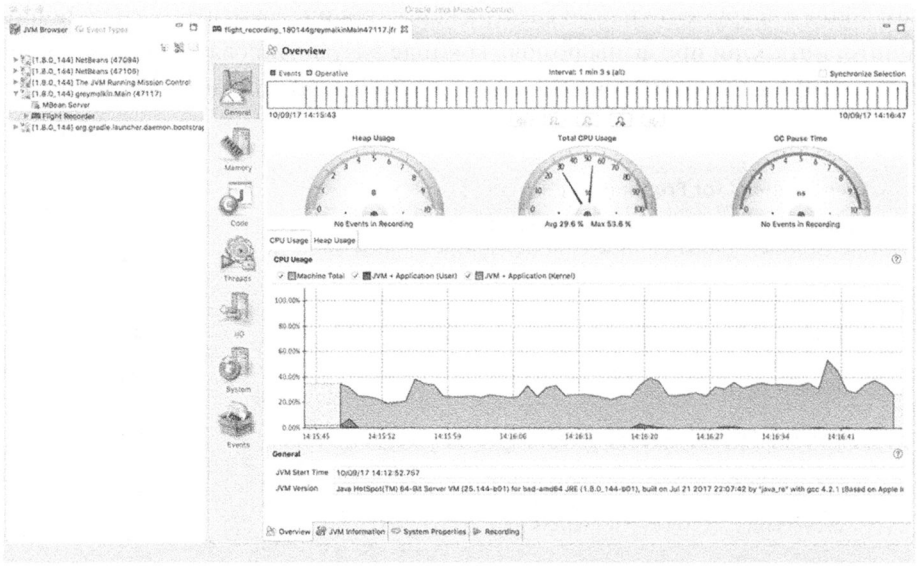

Рис. 13.12. Окно времени JMC

Thermostat разработан как открытая, расширяемая платформа и состоит из агента и клиента (обычно с простым графическим интерфейсом). Простое представление Thermostat показано на рис. 13.13.

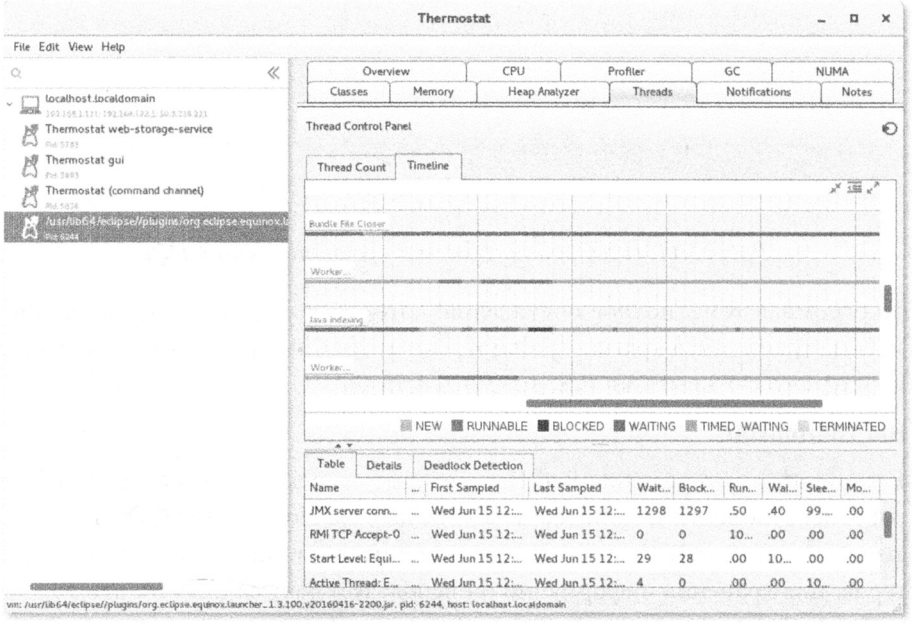

Рис. 13.13. Red Hat Thermostat

Архитектура Thermostat допускает расширение. Например, вы можете:

- собирать и анализировать собственные пользовательские метрики;
- выполнять инъекции собственного кода для выполнения требуемых измерений;
- писать собственные подключаемые модули и интегрировать дополнительные инструменты.

Большая часть встроенных функциональных возможностей Thermostat в действительности реализована как подключаемые модули.

New Relic

Инструмент New Relic[5] — продукт SaaS, предназначенный для облачных приложений. Это универсальный набор инструментов, охватывающий гораздо больше, чем просто JVM.

В пространстве JVM установка требует загрузки агента, передачи переключателя JVM и перезапуска сервера. После этого New Relic будет отображать представления, подобные показанному на рис. 13.14.

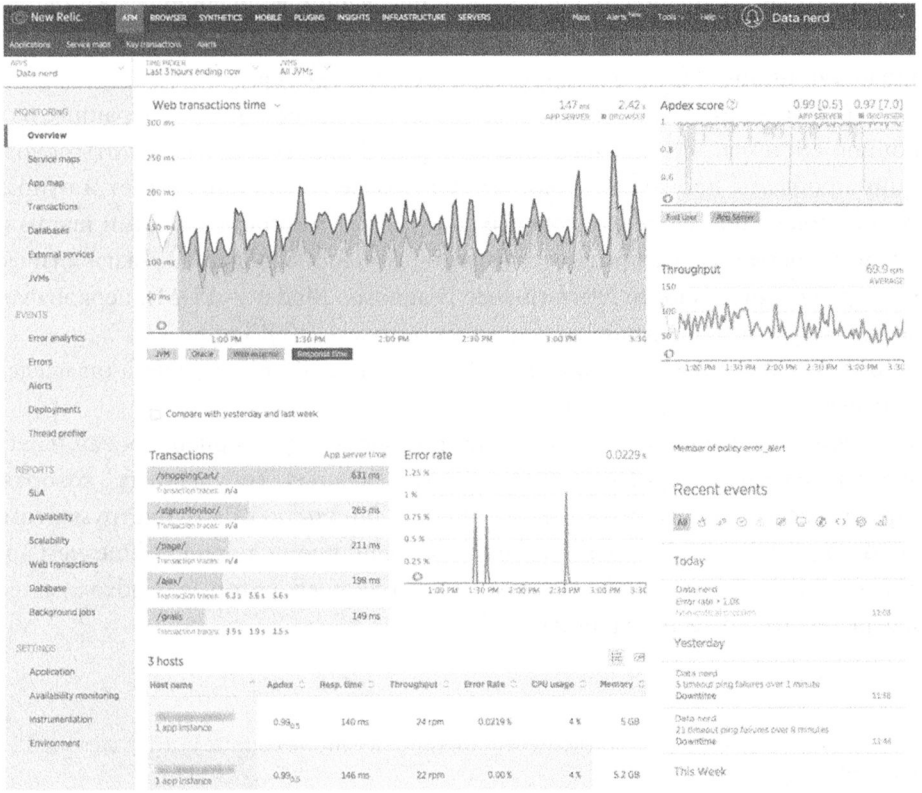

Рис. 13.14. New Relic

[5] https://newrelic.com/java

Мониторинг и полная (всеохватывающая) поддержка, предоставляемые New Relic, могут сделать ее привлекательным инструментом разработчика. Однако, будучи обобщенным инструментом, он не настроен конкретно на технологии JVM и в исходном состоянии полагается на менее сложные источники данных, доступные в JVM. Это означает, что для получения глубинной информации из JVM может потребоваться его объединение с более узконаправленными инструментами.

New Relic предоставляет API Java-агента[6]; возможна также реализация пользовательских инструментов[7] для расширения базовой функциональности.

New Relic генерирует огромное количество данных, и в результате иногда трудно обнаружить на выходе что-либо, помимо очевидных тенденций.

jClarity Illuminate

Инструмент, который обеспечивает мост между инструментами профилирования разработчика и оперативным мониторингом, — это jClarity Illuminate[8]. Это не традиционный профайлер на основе выборки; вместо этого он работает в режиме мониторинга с отдельным демоном вне процесса, который наблюдает за основным приложением Java. Обнаружив аномалию в поведении запущенной JVM, например нарушение соглашения об уровне обслуживания (service-level agreement — SLA), Illuminate инициирует глубокое зондирование приложения.

Алгоритм машинного обучения Illuminate анализирует данные, собранные из операционной системы, журналов сборки мусора и JVM, чтобы определить основную причину проблемы производительности. Он создает подробный отчет и отправляет его пользователю вместе с некоторыми возможными последующими шагами для устранения проблемы. Алгоритм машинного обучения основан на Диагностической модели производительности (Performance Diagnostic Model — PDM), первоначально созданной Кирком Пеппердайном, одним из основателей jClarity.

На рис. 13.15 мы можем увидеть режим *триады* Illuminate, когда он исследует автоматически обнаруженную проблему.

Инструмент основан на методах машинного обучения и ориентирован на углубленный анализ корней проблемы, а не на подавляющую "гору данных", которая нередко встречается в инструментах мониторинга. В попытках быть инструментом мониторинга производительности с более низким влиянием на приложение, чем другие инструменты, он значительно сокращает объем данных, которые необходимо собирать, перемещать по сети и хранить.

---

[6] https://docs.newrelic.com/docs/agents/java-agent/custom-instrumentation/java-agent-api

[7] https://docs.newrelic.com/docs/agents/java-agent/custom-instrumentation/java-custom-instrumentation

[8] https://www.jclarity.com/illuminate/

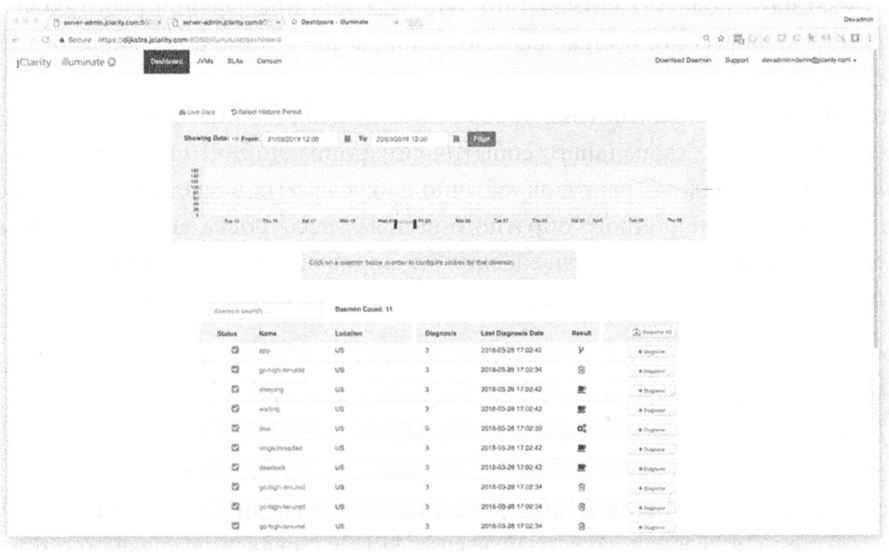

Рис. 13.15. jClarity Illuminate

Современные профайлеры

В этом разделе мы обсудим три современных инструмента с открытым исходным кодом, которые могут обеспечить лучшее понимание и более точные показатели производительности, чем традиционные профайлеры. Вот эти инструменты.

- Honest Profiler
- perf
- Async Profiler

Honest Profiler[9] — относительно недавний инструмента для профилирования. Это проект с открытым исходным кодом, возглавляемый Ричардом Уорбертоном (Richard Warburton) и произошедший от прототипа с открытым исходным текстом, написанного Джереми Мэнсоном (Jeremy Manson) из инженерной команды Google.

Основные цели Honest Profiler таковы:

- устранение искажения точек безопасности, которым страдает большинство профайлеров;
- работа с существенно сниженными накладными расходами.

Для этого он использует закрытый вызов API `AsyncGetCallTrace` в HotSpot. Это, конечно, означает, что Honest Profiler не будет работать на JVM без OpenJDK. Он будет работать с JVM от Oracle, Red Hat и Azul Zulu, а также с JVM HotSpot.

[9] `https://github.com/jvm-profiling-tools/honest-profiler`

Реализация использует сигнал Unix SIGPROF для прерывания работающего потока. Затем информация о стеке вызовов собирается с помощью закрытого метода AsyncGetCallTrace().

Этот метод прерывает потоки только по отдельности, поэтому никогда не происходит какого-либо глобального события синхронизации, что позволяет избежать конфликтов и накладных расходов, обычно наблюдаемых в традиционных профайлерах. В рамках асинхронного обратного вызова трассировка вызовов записывается в кольцевой буфер (без использования блокировок). Отдельный выделенный для этой цели поток затем записывает данные в журнал без приостановки приложения.

Honest Profiler — не единственный профайлер, использующий данный подход. Например, Flight Recorder также использует вызов AsyncGetCallTrace().

Исторически Sun Microsystems предлагала и старый продукт Solaris Studio, который также использовал закрытый вызов API. К сожалению, название продукта приводило к путанице, так как фактически он работал и на других операционных системах, а не только на Solaris, и в результате Solaris Studio не получил широкого распространения.

Один из недостатков Honest Profiler заключается в том, что он может показывать "Неизвестно" для некоторых потоков. Это побочный эффект встроенных функций JVM; в этом случае профайлер не в состоянии выполнить корректное отображение трассировки стека Java.

Чтобы использовать Honest Profiler, необходимо установить агент профайлера:

```
-agentpath:<путь-к-liblagent.so>=interval=<n>,logPath=< путь-к-log.hpl>
```

Honest Profiler включает относительно простой графический интерфейс для работы с профилями. Он основан на JavaFX и требует установки OpenJFX для работы на базе OpenJDK (например, с Azul Zulu или Red Hat IcedTea).

На рис. 13.16 показан типичный экран графического интерфейса пользователя Honest Profiler.

На практике такие инструменты, как Honest Profiler, чаще всего запускаются в без графического интерфейса в качестве инструмента для сбора данных. При таком подходе визуализация обеспечивается другими инструментами или пользовательскими сценариями.

В проекте предоставляются бинарные файлы, но они имеют довольно ограниченное применение — только для последних сборок Linux. Большинство серьезных пользователей Honest Profiler должны будут строить собственные бинарные файлы "с нуля" (тема, выходящая за рамки данной книги).

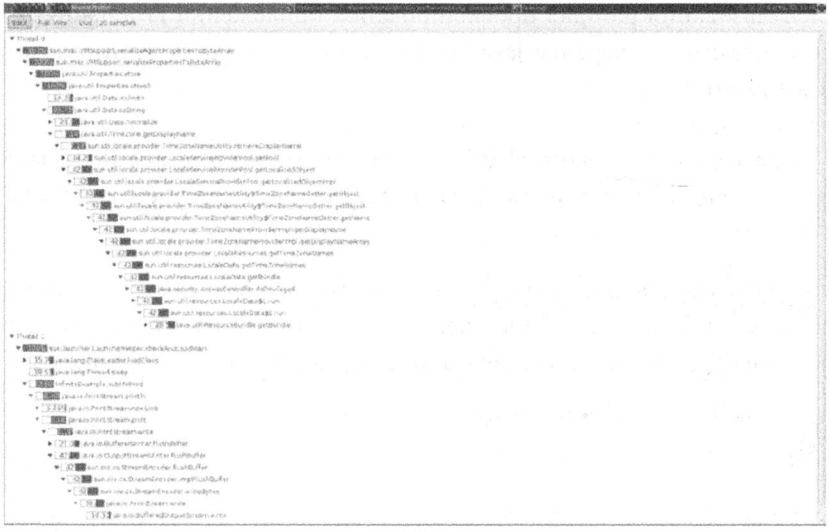

Рис. 13.16. Honest Profiler

Инструмент perf[10] представляет собой достаточно полезный простой профайлер для приложений, работающих в Linux. Он не является приложением, специфичным для Java/JVM, просто считывая аппаратные счетчики производительности, и включен в ядро Linux как `tools/perf`.

Счетчики производительности — это физические регистры, подсчитывающие аппаратные события, представляющие интерес для аналитиков в области производительности. К ним относятся выполненные команды, промахи кеша и неверные предсказания ветвлений. Все это создает основу для профилирования приложений.

Для perf Java представляет некоторые дополнительные проблемы из-за динамического характера среды выполнения Java. Чтобы использовать perf с Java-приложениями, нам нужен мост для обработки отображения динамических частей Java-исполнения.

Этот мост представляет собой агент perf-map-agent[11], который генерирует динамические символы для perf из неизвестных областей памяти (включая JIT-скомпилированные методы). Из-за динамически созданного интерпретатора HotSpot и таблиц переходов для виртуальной диспетчеризации они также должны иметь сгенерированные записи.

perf-map-agent состоит из агента, написанного на C, и небольшого загрузчика на Java, который при необходимости подключает агент к запущенному процессу Java. В Java 8u60 был добавлен новый флаг, обеспечивающий лучшее взаимодействие с perf:

```
-XX:+PreserveFramePointer
```

[10] `https://perf.wiki.kernel.org/index.php/Main_Page`

[11] `https://github.com/jvm-profiling-tools/perf-map-agent`

При использовании perf для профилирования приложений Java настоятельно рекомендуется работать с версией Java 8u60 или новее, чтобы получить возможность использовать этот флаг.

Применение этого флага отключает оптимизацию JIT-компилятора, что несколько (до 3% согласно тестам) снижает производительность.

Одну поразительную визуализацию получаемых с помощью perf данных производит инструмент flame graph[12]. Он очень подробно показывает, на что расходуется время выполнения. Пример такой визуализации можно увидеть на рис. 13.17.

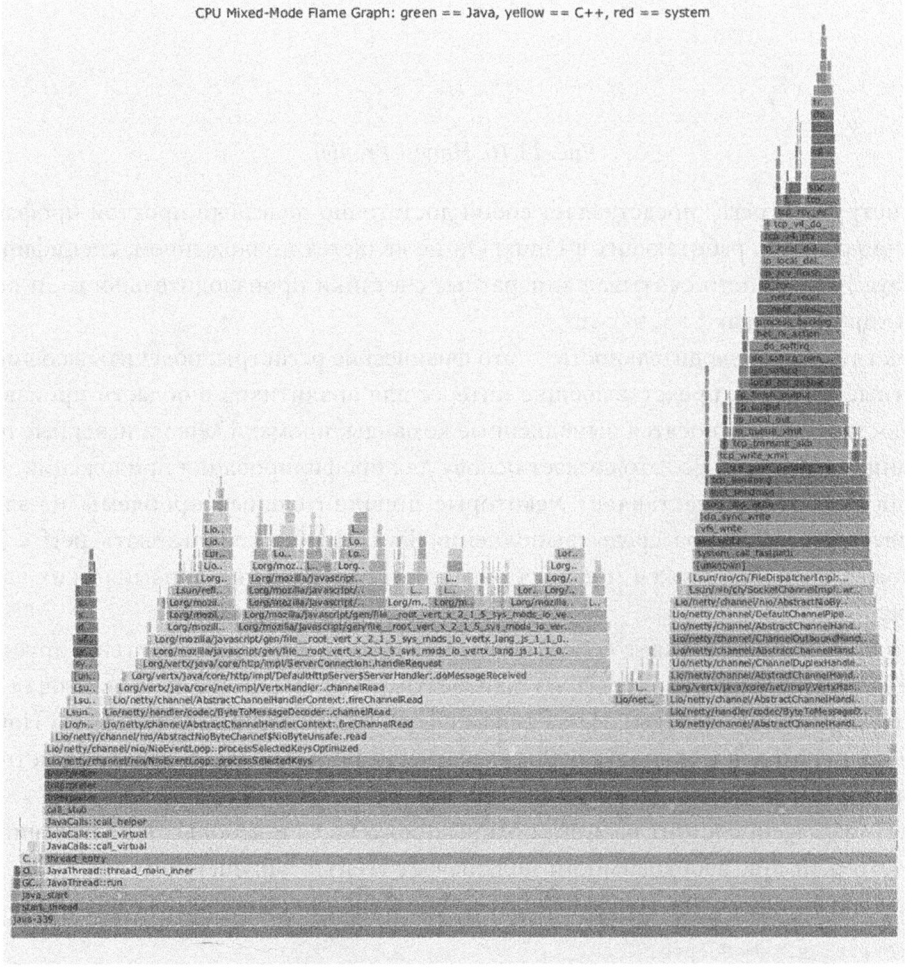

Рис. 13.17. Java flame graph

[12] https://github.com/brendangregg/FlameGraph

В блоге Netflix Technology Blog подробно освещается[13], как команда реализовывала flame graph на своих JVM.

Наконец, альтернативным для Honest Profiler вариантом является Async Profiler[14]. Он использует тот же внутренний API, что и Honest Profiler, и является инструментом с открытым исходным кодом, который работает только на JVM HotSpot. Его зависимость от perf означает, что Async Profiler работает только с операционными системами, в которых работает perf (в основном Linux).

Профилирование выделения памяти

Профилирование выполнения является важным, но не единственным аспектом профилирования! В большинстве приложений требуется также некоторый уровень профилирования памяти, и один из стандартных подходов состоит в рассмотрении поведения выделения памяти приложения. Существует несколько возможных подходов к профилированию выделения памяти.

Например, мы можем использовать подход `HeapVisitor`, на котором основаны такие инструменты, как `jmap`. На рис. 13.18 можно увидеть представление профилирования памяти VisualVM, которое использует этот простой подход.

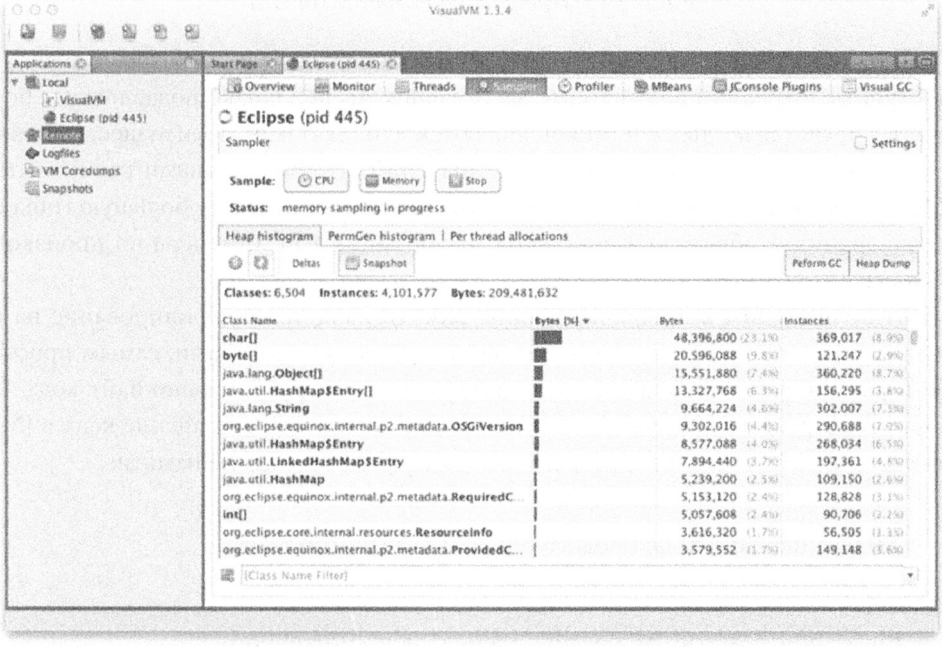

Рис. 13.18. Профайлер памяти VisualVM

[13] https://medium.com/netflix-techblog/java-in-flames-e763b3d32166

[14] https://github.com/jvm-profiling-tools/async-profiler

Аналогичное представление можно увидеть на рис. 13.19, где те же возможности продемонстрированы с использованием профайлера памяти YourKit.

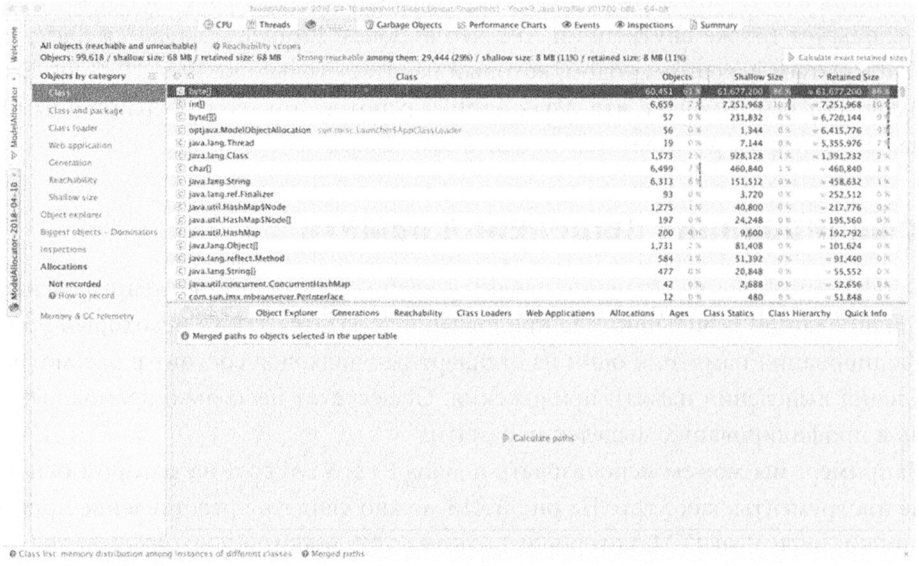

Рис. 13.19. Профайлер памяти YourKit

Для инструмента JMC статистика сборки мусора содержит некоторые значения, недоступные в традиционном агенте Serviceability Agent. Однако подавляющее большинство представленных счетчиков являются дубликатами. Преимущество заключается в том, что стоимость сбора этих значений, чтобы они могли отображаться в JMC, у JFR намного меньше, чем у SA. JMC также обеспечивает большую гибкость с точки зрения отображения детальной информации для инженера по производительности.

Другим подходом к профилированию памяти является профилирование на основе агентов. Оно может быть выполнено несколькими способами, самым простым из которых является оснащение соответствующим инструментарием байт-кода. Как говорилось в разделе "Введение в байт-код JVM" главы 9, "Выполнение кода в JMV", имеются три байт-кода, которые запрашивают у JVM выделение памяти:

`NEW`
: Выделение памяти для нового объекта указанного типа

`NEWARRAY`
: Выделение памяти для массива элементов примитивного типа

`ANEWARRAY`
: Выделение памяти для массива объектов указанного типа

Следует отслеживать лишь перечисленные байт-коды, поскольку только они могут вызвать выделение памяти.

Простой инструментальный подход состоит в том, чтобы найти каждый экземпляр любого из кодов операций выделения памяти и вставить вызов статического метода, который протоколирует выделение памяти перед тем, как будет выполнен соответствующий код операции.

Давайте рассмотрим скелет такого профилирования выделения памяти. Нам нужно настроить агент, используя инструментальный API с помощью перехвата `premain()`:

```
public class AllocAgent
{
    public static void premain(String args,
                               Instrumentation instrumentation)
    {
        AllocRewriter transformer = new AllocRewriter();
        instrumentation.addTransformer(transformer);
    }
}
```

Для выполнения такого рода инструментирования байт-кода обычно используется библиотека, а не выполнение преобразования с помощью написанного вручную кода. Одной из широко распространенных библиотек манипулирования байт-кодом является ASM[15], которую мы будем использовать для демонстрации профилирования памяти.

Чтобы добавить требуемый код инструментирования, нам нужен переписывающий класс. Он обеспечивает мост между API инструментирования и ASM, и выглядит следующим образом:

```
public class AllocRewriter implements ClassFileTransformer
{
  @Override
  public byte[] transform(ClassLoader loader, String className,
                          Class<?> classBeingRedefined,
                          ProtectionDomain protectionDomain,
                          byte[] originalClassContents)
              throws IllegalClassFormatException
  {
    final ClassReader reader = new ClassReader(originalClassContents);
    final ClassWriter writer = new ClassWriter(reader,
        ClassWriter.COMPUTE_FRAMES | ClassWriter.COMPUTE_MAXS);
    final ClassVisitor coster = new ClassVisitor(Opcodes.ASM5, writer)
    {
      @Override
```

[15] http://asm.ow2.org/

```
      public MethodVisitor visitMethod(final int access,
                                       final String name,
                                       final String desc,
                                       final String signature,
                                       final String[] exceptions)
      {
        final MethodVisitor baseMethodVisitor =
        super.visitMethod(access, name, desc, signature, exceptions);
        return new AllocationRecordingMethodVisitor(baseMethodVisitor,
            access, name, desc);
      }
    };
    reader.accept(coster, ClassReader.EXPAND_FRAMES);
    return writer.toByteArray();
  }
}
```

Здесь использован метод посетителя для инспекции байт-кода и вставки инструментирующих вызовов для отслеживания выделения памяти:

```
public final class AllocationRecordingMethodVisitor
                extends GeneratorAdapter
{
  private final String runtimeAccounterTypeName =
                                "optjava/bc/RuntimeCostAccounter";
  public AllocationRecordingMethodVisitor(MethodVisitor methodVisitor,
                                          int access, String name, String desc)
  {
    super(Opcodes.ASM5, methodVisitor, access, name, desc);
  }
  /**
    * Этот метод вызывается при посещении кода операции с единственным
    * операндом типа int. Для наших целей это код операции NEWARRAY.
    *
    * @param opcode
    * @param operand
    */
  @Override
  public void visitIntInsn(final int opcode, final int operand)
  {
    if (opcode != Opcodes.NEWARRAY)
    {
      super.visitIntInsn(opcode, operand);
      return;
    }

    // Опкод NEWARRAY - recordArrayAllocation:(Ljava/lang/String;I)V
    // Значение операнда должно быть одним из Opcodes.T_BOOLEAN,
```

```
    // Opcodes.T_CHAR, Opcodes.T_FLOAT, Opcodes.T_DOUBLE,
    // Opcodes.T_BYTE, Opcodes.T_SHORT, Opcodes.T_INT
    // или Opcodes.T_LONG.
    final int typeSize;

    switch (operand)
    {
      case Opcodes.T_BOOLEAN:
      case Opcodes.T_BYTE:
        typeSize = 1;
        break;

      case Opcodes.T_SHORT:
      case Opcodes.T_CHAR:
        typeSize = 2;
        break;

      case Opcodes.T_INT:
      case Opcodes.T_FLOAT:
        typeSize = 4;
        break;

      case Opcodes.T_LONG:
      case Opcodes.T_DOUBLE:
        typeSize = 8;
        break;

      default:
        throw new IllegalStateException(
            "Некорректная операция: NEWARRAY получает: "
            + operand);
    }

    super.visitInsn(Opcodes.DUP);
    super.visitLdcInsn(typeSize);
    super.visitMethodInsn(Opcodes.INVOKESTATIC, runtimeAccounterTypeName,
                          "recordArrayAllocation", "(II)V", true);
    super.visitIntInsn(opcode, operand);
  }
  /**
    * Этот метод вызывается при посещении кода операции с единственным
    * операндом, который представляет собой тип (представленный здесь
    * как String).
    *
    * Для наших целей это либо NEW, либо ANEWARRAY.
    *
    * @param opcode
    * @param type
```

```
    */
  @Override
  public void visitTypeInsn(final int opcode, final String type)
  {
    // Опкод либо NEW - recordAllocation:(Ljava/lang/String;)V
    // либо ANEWARRAY - recordArrayAllocation:(Ljava/lang/String;I)V
    switch (opcode)
    {
      case Opcodes.NEW:
        super.visitLdcInsn(type);
        super.visitMethodInsn(Opcodes.INVOKESTATIC, runtimeAccounterTypeName,
                              "recordAllocation",
                              "(Ljava/lang/String;)V", true);
        break;

      case Opcodes.ANEWARRAY:
        super.visitInsn(Opcodes.DUP);
        super.visitLdcInsn(8);
        super.visitMethodInsn(Opcodes.INVOKESTATIC, runtimeAccounterTypeName,
                              "recordArrayAllocation", "(II)V", true);
        break;
    }

    super.visitTypeInsn(opcode, type);
  }
}
```

Требуется также небольшой компонент времени выполнения:

```
public class RuntimeCostAccounter
{
  private static final ThreadLocal<Long> allocationCost =
    new ThreadLocal<Long>()
  {
    @Override
    protected Long initialValue()
    {
      return 0L;
    }
  };
  public static void recordAllocation(final String typeName)
  {
    // Требуется более сложный подход
    // Например, кеширование приближенных размеров
    // для встреченных нами типов
    checkAllocationCost(1);
  }
  public static void recordArrayAllocation(final int length,
```

```
      final int multiplier)
  {
    checkAllocationCost(length * multiplier);
  }
  private static void checkAllocationCost(final long additional)
  {
    final long newValue = additional + allocationCost.get();
    allocationCost.set(newValue);
    // Выполнение действия? Например, сбой при
    // превышении некоторого порога.
  }
  // Может быть показан, например, через счетчик JMX
  public static long getAllocationCost()
  {
    return allocationCost.get();
  }
  public static void resetCounters()
  {
    allocationCost.set(0L);
  }
}
```

Цель этих двух фрагментов состоит в том, чтобы обеспечить простое инструментирование выделения памяти. Они используют библиотеку манипулирования байт-кодом ASM (полное описание которой, к сожалению, выходит за рамки этой книги). Посетитель метода добавляет вызов метода записи перед каждым экземпляром байт-кодов `NEW`, `NEWARRA` и `ANEWARRAY`.

С помощью такого преобразования кода при создании нового объекта или массива вызывается метод записи. Эта деятельность должна поддерживаться во время выполнения небольшим классом, `RuntimeCostAccounter` (который должен находиться в пути к классам). Этот класс для каждого потока поддерживает подсчет количества памяти, выделенной инструментированным кодом.

Такая методика на уровне байт-кода довольно грубая, но для заинтересованных читателей она должна стать отправной точкой для разработки собственных простых измерений выделения памяти потоками. Например, этот подход может быть использован в модульном или регрессионном тесте, чтобы гарантировать, что изменения кода не приводят к большому количеству дополнительно выделяемой памяти.

Однако для промышленного использования этот подход может оказаться неприемлемым. Дополнительные вызовы методов при каждом выделении памяти приводят к огромному количеству дополнительной работы. В этой ситуации может помочь JIT-компиляция, встраивающая инструментальные вызовы, но в целом, вероятно, такое профилирование будет оказывать очень существенное влияние на производительность приложения.

Еще одним подходом к профилированию памяти является исчерпание TLAB. Например, Async Profiler поддерживает выборку с использованием TLAB. Для получения уведомлений используются обратные вызовы, специфичные для HotSpot:

- когда для объекта выделена память во вновь созданном TLAB;
- когда для объекта выделена память вне TLAB ("медленный путь").

В результате подсчитывается не каждое выделение памяти для объектов. Вместо этого записываются распределения каждых *n* Кбайт, где *n* — средний размер TLAB (напомним, что размер TLAB со временем может меняться).

Этот дизайн направлен на то, чтобы быть пригодным для производственного применения, для чего требуется сделать выборку кучи достаточно дешевой. С другой стороны, собранные данные получены с помощью выборки, а следовательно, могут быть неполными. Замысел состоит в том, что на практике будут отражены основные источники выделения памяти, по крайней мере, достаточное время, чтобы информация была полезна для инженера по производительности.

Чтобы использовать выборку TLAB, требуется JVM HotSpot версии 7u40 или более поздней, поскольку именно в этой версии впервые появились обратные вызовы TLAB.

Анализ дампа кучи

С профилированием памяти тесно связан *анализ дампа кучи* (heap dump analysis). Это использование инструмента для изучения моментального снимка всей кучи и определения важных фактов, таких как множество активных объектов, количество и типы объектов, а также форма и структура графа объектов.

Загрузив дамп кучи, инженеры по производительности могут затем обходить и анализировать моментальный снимок кучи в момент создания дампа. Они могут видеть как живые объекты, так и объекты, которые уже уничтожены, но еще не собраны сборщиком мусора.

Основным недостатком дампа кучи является их размер. Размер дампа кучи часто может составлять 300–400% от размера сбрасываемой памяти. Для многогигабайтной кучи это очень существенный размер. Мало того что куча должна быть записана на диск; при реальном производственном использовании она должна быть еще и загружена по сети. После получения дамп должен быть загружен на рабочую станцию с достаточными ресурсами (особенно памятью) для его обработки без внесения чрезмерных задержек в рабочий процесс. Работа с большими дампами кучи на машине, которая не может загрузить весь дамп сразу, может быть очень болезненной, поскольку страницы рабочей станции размещают части файла дампа на диске и вне его.

Для создания файла кучи требуется приостановка выполнения приложения (STW).

YourKit поддерживает захват снимков памяти как в формате `hprof`, так и в закрытом патентованном формате. На рис. 13.20 показано представление анализатора дампа кучи.

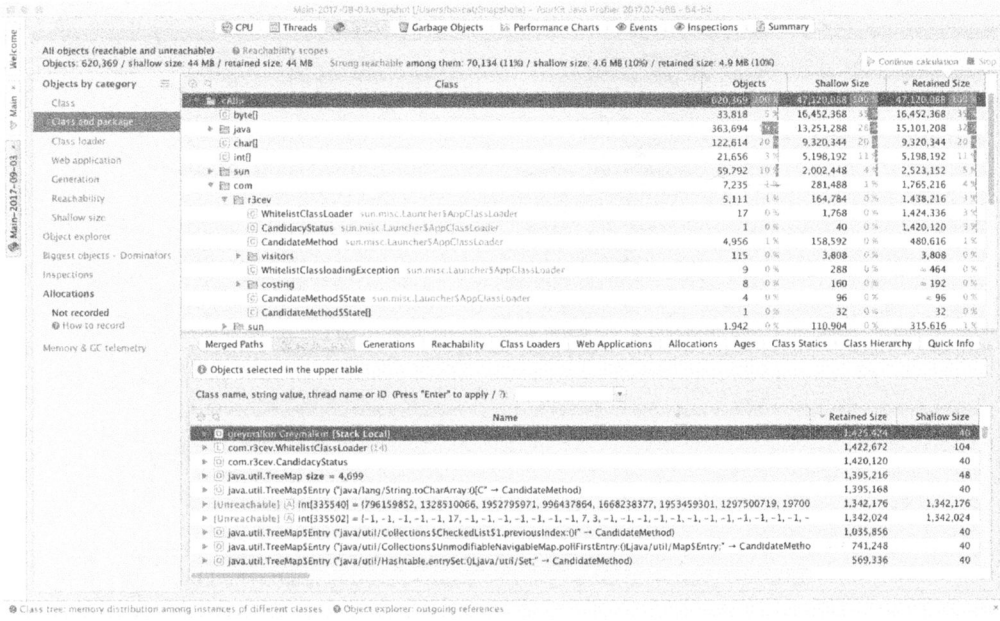

Рис. 13.20. Профайлер дампа памяти YourKit

Среди коммерческих инструментов YourKit обеспечивает хороший выбор фильтров и различных представлений кучи. Следует упомянуть о возможности разделить его на загрузчик классов и веб-приложение, что может привести к более быстрой диагностике проблем с кучей.

Стоит также упомянуть о таком инструменте, как представление Allocations из JMC/JFR. Оно способно отображать распределение TLAB, которое используется также инструментом Async Profiler. На рис. 13.21 показан пример представления распределения в JMC.

Профилирование распределения и кучи представляет интерес для большинства приложений, которые должны быть профилированы, и инженерам по производительности рекомендуется не переоценивать профилирование выполнения по сравнению с профилированием памяти.

hprof

Профайлер `hprof` поставляется с JDK начиная с версии 5. Он задуман в основном как эталонная реализация для технологии JVMTI, а не как профайлер производственного уровня. Несмотря на это он часто упоминается в документации, что заставило некоторых разработчиков пересмотреть свое отношение и считать `hprof` подходящим инструментом для реального использования.

Начиная с Java 9 (JEP 240) `hprof` удален из JDK. Вот что говорится об этом в документации.

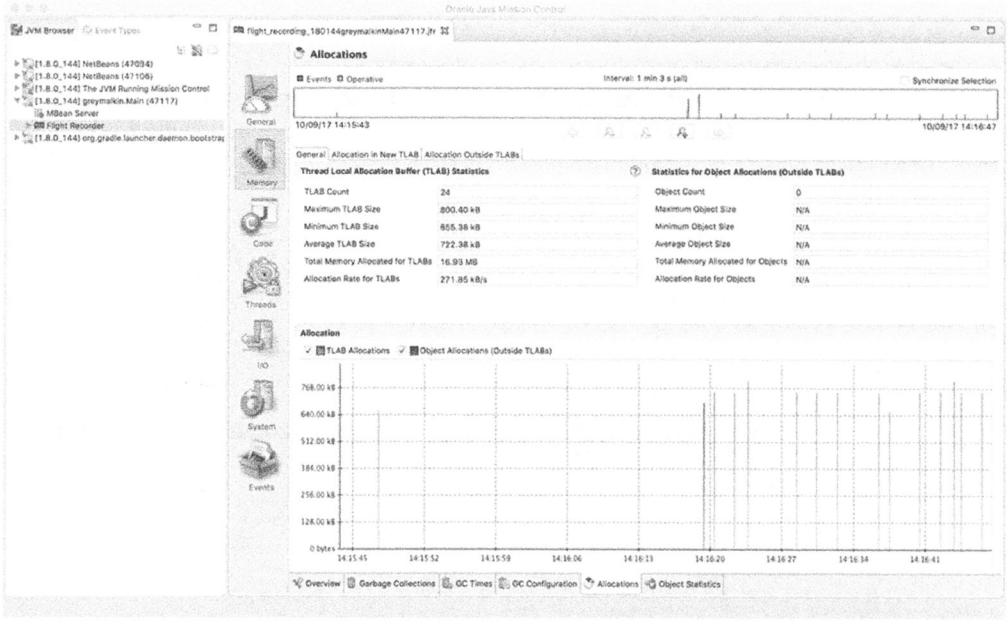

Рис. 13.21. Представление JMC Allocations

> Агент `hprof` был написан как демонстрационный код для JVM Tool Interface и не рассматривался как производственный инструмент.

Кроме того, код и документация `hprof` содержат ряд утверждений общего вида:

> Это демонстрационный код для интерфейса JVM TI и использования BCI; это не официальный продукт и не часть JDK.

По этой причине не следует полагаться на `hprof`, кроме как в качестве устаревшего формата для снимков кучи. Возможность создания дампов кучи будет сохраняться в Java 9 и в обозримом будущем.

Резюме

Тема профилирования — одна из тем, которые разработчики часто понимают неправильно. Как профилирование выполнения, так и профилирование памяти — оба они являются необходимыми. Однако очень важно, чтобы инженеры в области производительности понимали, что они делают и *почему*. Простое использование инструментов "вслепую" может привести к совершенно неверным или нерелевантным результатам и к пустой трате огромного количества времени.

Профилирование современных приложений требует соответствующих инструментов, и у разработчика имеется богатый их выбор, включая как коммерческие инструменты, так и варианты с открытым исходным кодом.

ГЛАВА 14

Высокопроизводительное протоколирование и обмен сообщениями

По сравнению с такими языками, как C ++, использование Java и JVM иногда рассматривается как компромисс. Язык программирования Java добился повышения производительности разработчиков за счет сокращения количества низкоуровневых проблем, которыми они должны заниматься в повседневной практике. Предполагаемый компромисс — языковые абстракции более высокого уровня, приводящие к повышению производительности разработчиков за счет низкоуровневого управления и производительности.

C++ исповедует иной подход — независимо от того, какие новые возможности добавляются в язык, производительность не должна страдать от этого. Философия C++ приводит к сложному уровню управления за счет того, что каждый разработчик должен вручную управлять ресурсами или писать код, использующий соответствующие идиомы.

Платформа Java использует подход, при котором разработчик не должен беспокоиться о низкоуровневых деталях. Преимущество автоматического управления памятью не следует недооценивать как всего лишь огромное увеличение скорости разработки — многолетний опыт авторов свидетельствует о большом ущербе, который может нанести своими ошибками неопытный разработчик на C++.

Однако сборка мусора и другие высокоуровневые управляемые абстракции JVM могут внести определенную непредсказуемость, когда речь идет о производительности. Этот недетерминизм, естественно, должен быть сведен к минимуму в приложениях, чувствительных к задержкам.

Означает ли это, что Java и JVM являются неподходящей платформой для высокопроизводительных систем?

В этой главе мы рассмотрим некоторые общие проблемы, с которыми приходится иметь дело разработчикам при работе с высокопроизводительными приложениями, чувствительными к задержкам. Здесь также рассматриваются подходы к проектированию систем с малой задержкой и требования, предъявляемые к последним. Два соображения, являющиеся центральными для высокопроизводительных систем с низкими значениями времени задержек, — это протоколирование и обмен сообщениями.

Ведение журнала должно быть предметом озабоченности любого разработчика на Java, поскольку Java-система с разумным уровнем поддержки и сопровождения обычно содержит большое количество журнальных сообщений. Однако для разработчиков высокопроизводительных приложений с малыми временами задержки протоколирование может приобретать особое значение. К счастью, это область, в которой было проведено множество исследований и разработок, некоторые из которых мы и рассмотрим в этой главе.

Системы обмена сообщениями обеспечили одну из самых успешных архитектурных моделей последних лет. В результате они обычно находятся на переднем крае систем с малой задержкой и обычно характеризуются количеством обработанных за секунду сообщений. Количество сообщений, которые система может обрабатывать, часто имеет решающее значение для получения конкурентного преимущества. Способность привязывать материальную ценность к пропускной способности (например, количество торгов в секунду) означает тщательно исследованную и финансируемую область. Ниже мы обсудим некоторые современные методы обмена сообщениями.

Протоколирование

Многие разработчики не считают библиотеку протоколирования важной частью проекта, и ее выбор часто оказывается "путем наименьшего сопротивления". Это противоречит тому, как в проект добавляются многие другие библиотеки, для которых затрачивается немалое время на изучение их возможностей и, возможно, на выполнение некоторых эталонных тестов производительности.

Вот несколько антипаттернов, связанных с выбором производственной системы протоколирования.

10 лет протоколирования

> *Некто* однажды удачно сконфигурировал систему протоколирования. Куда проще позаимствовать готовую конфигурацию, чем создавать новую.

Единый журнал проекта

> *Некто* однажды написал оболочку для протоколирования, чтобы избежать необходимости перенастраивать протоколирование для каждой части проекта по-своему.

Единая система фирмы

> *Некто* однажды создал систему протоколирования, которая с того времени используется во всех проектах фирмы.

Конечно, *некто* никогда не намеревался создавать будущую проблему. Обычно существует допустимый вариант выбора архитектуры журналов — например, интеграция с другими функциями в организации, ведущая к журналу на уровне фирмы. Проблема часто связана с поддержкой системы протоколирования, поскольку она обычно не рассматривается как критически важная для бизнеса. Такое пренебрежение приводит к технической задолженности, которая может охватить всю организацию. Несмотря на то что системы протоколирования не назовешь особо увлекательными и их выбор часто следует одному из упомянутых выше антипаттернов, они играют важную роль во всех приложениях.

Во многих высокопроизводительных средах точность обработки и отчетность важны не меньше, чем скорость. Нет смысла делать что-то быстро, но неправильно; а зачастую могут возникнуть дополнительные требования к аудиту о точном протоколировании выполняемых действий. Журналы помогают идентифицировать производственные проблемы, причем информации в них должно быть достаточно, чтобы команда разработчиков могла исследовать проблему post factum. Важно, чтобы регистратор рассматривался не просто как дополнительная стоимость, а как любой другой компонент в системе, который требует тщательного контроля и продуманного включения в проект.

Микротестирование протоколирования

В этом разделе будет рассмотрен набор микротестов, предназначенных для объективного сопоставления производительности трех самых популярных регистраторов (Logback, Log4j и `java.util.logging`) с использованием различных шаблонов журналов.

Статистика основана на проекте Стивена Коннолли (Stephen Connolly) с открытым исходным кодом и может быть найдена на GitHub[1]. Проект хорошо разработан и представлен в виде готового набора тестов производительности с многократными запусками регистраторов с различными конфигурациями.

Эти тесты исследуют каждый регистратор в сочетании с различными форматами протоколирования, чтобы дать нам представление об общей производительности каркаса протоколирования и о том, влияет ли схема применения (архитектура) на производительность.

На данном этапе очень важно четко объяснить, почему мы используем подход с микротестированием. Обсуждая детали этих конкретных технологий, мы

[1] `https://github.com/stephenc/java-logging-benchmarks`

столкнулись с проблемой, стоящей перед авторами библиотек: мы хотели получить представление о производительности разных регистраторов на основе разных конфигураций, но знали, что очень сложно найти хорошую совокупность приложений, использующих точные конфигурации протоколирования, нужные для наших целей и дающих результаты, которые обеспечили бы содержательное сравнение регистраторов.

В этой ситуации, когда код будет работать во многих различных приложениях, микротестирование дает оценку будущей работы кода. Для микротестирования мы используем "универсальный бессмысленный набор" применений регистраторов.

Полученные результаты дают нам общую картину, но все равно необходимо профилировать приложение до и после добавления протоколирования, чтобы получить истинную картину изменений перед тем, как внедрять тот или иной регистратор в реальное производственное приложение.

Итак, давайте посмотрим на результаты и на то, как они были получены.

Отсутствие протоколирования

Отсутствие протоколирования — это тест для измерения стоимости ничего не делающего регистратора в ситуации, когда сообщения журнала находятся ниже его текущего порогового значения. Этот тест можно рассматривать как контрольную группу эксперимента.

Формат Logback

```
14:18:17.635 [Имя потока] INFO c.e. Имя_регистратора - Сообщение журнала
```

Наши микротесты используют формат версии Logback 1.2.1.

Формат `java.util.logging`

```
Feb 08, 2017 2:09:19 PM com.example.Имя_регистратора Имя_метода
INFO: Сообщение журнала
```

Формат Log4j

```
2017-02-08 14:16:29,651 [Имя потока]
        ↳ INFO com.example. Имя_регистратора - сообщение
```

Микротесты используют формат версии Log4j 2.7.

Измерения

С целью сравнения тесты производительности выполнялись на iMac и на экземпляре AWS (Amazon Web Services) EC t2.2xlarge (табл. 14.1 и 14.2). Профилирование на macOS может вызвать проблемы из-за различных технологий энергосбережения, а AWS имеет тот недостаток, что на результаты эталонного теста могут влиять другие контейнеры. Ни одна среда не идеальна: всегда есть шум, и, как обсуждалось в

главе 5, "Микротесты и статистика", микротесты полны рисков. Надеемся, что сравнение двух наборов данных для этих тестов поможет выявить полезные схемы для управления профилированием реальных приложений. Помните принцип Фейнмана — "вы не должны одурачить себя" — при обработке экспериментальных данных.

Таблица 14.1. Тесты производительности на iMac (нс/операция)

	Отсутствие протоколирования	Формат		
		Logback	java.util.logging	Log4j
Java.util.logging	158,051 (±0,762)	42404,202 (±541,229)	86054,783 (±541,229)	74794,026 (±2244,146)
Log4j	138,495 (±94,490)	8056,299 (±447,815)	32755,168 (±27,054)	5323,127 (±47,160)
Logback	214,032 (±2,260)	5507,546 (±258,971)	27420,108 (±37,054)	3501,858 (±47,873)

Регистратор `java.util.logging` выполняет свои операции протоколирования за время от 42 до 86 мкс на операцию. Наихудшая производительность для этого регистратора достигается при использовании формата `java.util.logging` — более чем в 2,5 раза хуже, чем при использовании того же формата для Log4j.

В целом наилучшую производительность в этом тесте на iMac обеспечивает Logback, причем лучше всего использовать формат log4j.

Таблица 14.2. Тесты производительности на AWS EC t2.2xlarge (нс/операция)

	Отсутствие протоколирования	Формат		
		Logback	java.util.logging	Log4j
Java.util.logging	1376,597 (±106,613)	54658,098 (±516,184)	144661,388 (±10333,854)	109895,219 (±5457,031)
Log4j	1699,774 (±111,222)	5835,090 (±27,592)	34605,770 (±38,816)	5809,098 (±27,792)
Logback	2440,952 (±159,290)	4786,511 (±29,526)	30550,569 (±39,951)	5485,938 (±38,674)

Взглянув на тесты для AWS, мы увидим, что результаты показывают общую картину результатов, аналогичную iMac. Logback немного быстрее Log4j. Из этих результатов можно сделать некоторые ключевые выводы.

Правильный способ измерения воздействия на приложение — это профилирование приложения до и после изменения конфигурации на производственном оборудовании. Тесты, показанные здесь, должны повторяться в конфигурации, которая отражает вашу конкретную производственную машину, а не принимается "как есть".

Тем не менее AWS имеет значительно более быструю скорость выполнения в целом, что может быть связано с проблемами энергосбережения на iMac или с другими факторами, которые не были учтены.

Результаты регистраторов

Тесты производительности показывают, что ряд результатов зависит от используемого формата ведения журнала, каркаса для ведения журнала и используемой конфигурации. В общем случае протоколирование `java.util.logging` обеспечивает наихудшую производительность в смысле времени выполнения. Формат Log4j, как представляется, дает наиболее последовательные результаты во всем диапазоне регистраторов; наилучшее время демонстрирует Logback.

В реальных системах следует проверять производительность выполнения на производственном наборе данных, в особенности когда получаемые результаты близки. В реальных системах воспринимать как свидетельство чего-либо следует только самые ясные и однозначно трактуемые результаты; различия в несколько десятков процентов обычно недостаточно.

Опасность микротестов, как говорилось в главе 5, "Микротесты и статистика", заключается в том, что рассмотрение проблемы в малом диапазоне потенциально может замаскировать ее влияние на наше приложение в целом. В результате возможно принять на основе микротестов решение, которое в конечном итоге повлияет на работу всего приложения совершенно иным, неожиданным образом.

Одной из таких проблем может оказаться количество мусора, генерируемого каркасом протоколирования, — равно как и время процессора, затрачиваемое на ведение журнала вместо обработки критических параллельных задач. Конструкция библиотеки протоколирования и механизм ее работы так же важны, как и результаты непосредственной работы в микротестах.

Проектирование регистратора с малым влиянием на приложение

Ведение журнала является критическим компонентом любого приложения, но в приложениях с низкой задержкой крайне важно, чтобы регистратор не стал узким местом для производительности бизнес-логики. Ранее в этой главе мы рассмотрели идею о том, что разработчик часто не осознает важность процесса выбора подходящего регистратора и не выполняет тестирования. Во многих случаях регистрация выглядит проблемной только тогда, когда начинает играть слишком большую (или даже доминирующую) роль в приложении.

> *Пока что в своей практике я редко сталкивался с клиентами, на систему которых протоколирование оказывало бы отрицательное влияние. Самым*

экстремальным был случай клиента, у которого из 4,5 секунды работы ведение журнала составляло 4,2 секунды.

— Кирк Пеппердайн (Kirk Pepperdine)

Версия Log4j 2.6 направлена на решение озвученных Кирком проблем путем создания стационарного регистратора без сборки мусора.

В документации приведен простой тест, состоящий из запуска приложения в Java Flight Recorder, которое выполняет журнальную запись строки как можно чаще в течение 12 секунд. Регистратор был настроен как асинхронный с помощью `RandomAccessFile` с шаблоном

```
%d %p %c{1.} [%t] %m %ex%n
```

На рис. 14.1 для сравнения приведены нестационарный сборщик мусора и простой профайл. Цель заключается не в проведении точного микротестирования, а в демонстрации профилирования собственно поведения протоколирования. Профайлер демонстрирует значительные циклы сборки мусора: 141 сборка со средним временем паузы около 7 мс и максимальным временем паузы 52 мс.

Сравните это с рис. 14.2, чтобы увидеть разницу между Log4j версий 2.5 и 2.6 (в последней за тот же период не был выполнен ни один цикл сборки мусора).

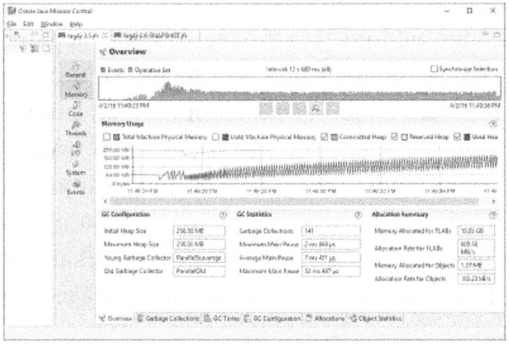

Рис. 14.1. Образец работы Log4j 2.5

Рис. 14.2. Образец работы Log4j 2.6

Log4j 2.6 предоставляет некоторые очевидные преимущества при настройке для выполнения, показанной на рис. 14.2. Однако есть некоторые ограничения, связанные с реализацией регистратора с нулевым выделением памяти — как обычно, бесплатных обедов не бывает.

Log4j достигает наблюдаемой производительности путем повторного использования объектов вместо создания временных объектов в каждом сообщении журнала. Это классическое использование проектного шаблона *Пул объектов* (Object Pool) со всеми вытекающими последствиями. Log4j 2.6 использует объекты повторно с помощью `ThreadLocal`-полей, а также с помощью повторного использования буферов для преобразования строк в байты.

Это один из результатов, которые мы никак не могли получить исключительно с помощью микротестирования. Как всегда, следует учитывать дизайн и общую картину происходящего.

Объекты `ThreadLocal` могут оказаться проблематичными в веб-контейнерах, в частности когда веб-приложения загружаются и выгружаются из контейнера. Log4j 2.6 не будет использовать `ThreadLocal` при работе в веб-контейнере, но по-прежнему будет применять некоторые совместно используемые и кешированные структуры, чтобы повысить производительность.

Если приложение уже использует более старую версию Log4j, необходимо рассмотреть возможность ее обновления до версии 2.6 и пересмотреть ее конфигурацию. Log4j уменьшает количество выделений памяти с помощью вызовов с переменным количеством аргументов, создавая временный массив для параметров, передаваемых в инструкцию записи в журнал. Если Log4j используется посредством SLF4J, то данный фасад будет поддерживать только два параметра; использование большего количества параметров в SLF4J потребует отказа от подхода без сборки мусора либо рефакторинга всего кода с целью непосредственного использования библиотек Log4j2.

Низкие задержки с использованием библиотек Real Logic

Real Logic[2] — британская компания, основанная Мартином Томпсоном (Martin Thompson). Мартин известен своим новаторским подходом к механическому взаимопониманию, основанным на понимании того, как низкоуровневые детали влияют на высокопроизводительный дизайн. Одним из наиболее известных вкладов Мартина в развитие Java является проектный шаблон *Разрушитель* (Disruptor)[3].

[2] `https://real-logic.co.uk/`

[3] `https://lmax-exchange.github.io/disruptor/`

Блог Мартина[4] под названием Mechanical Sympathy, а также связанный с ним список рассылки[5] представляют собой отличный ресурс для разработчиков, которые хотели бы повысить производительность своих приложений.

На странице GitHub[6] компании Real Logic имеется ряд популярных проектов с открытым исходным кодом, разработанных Мартином и другими программистами. Здесь вы можете найти следующие проекты.

Agrona

Высокопроизводительные структуры данных и полезные сервисные методы для Java.

Simple Binary Encoding (SBE)

Высокопроизводительный кодек сообщений.

Aeron

Эффективная и надежная одноадресная передача UDP, многоадресная передача UDP и транспорт сообщений IPC.

Artio

Устойчивый и высокопроизводительный шлюз FIX.

В следующих разделах будут рассмотрены эти проекты и философия дизайна, позволяющего этим библиотекам достигать более высокой производительности Java.

У Real Logic имеется гибкий высоконадежный шлюз FIX, использующий указанные библиотеки, однако его рассмотрение выходит за рамки данной книги.

Agrona

Проект Agrona[7] (название которого происходит из кельтской мифологии Уэльса и Шотландии) представляет собой библиотеку, содержащую строительные блоки для приложений с малым временем задержки. В разделе "Построение параллельных библиотек" главы 12, "Методы повышения производительности параллельной работы", мы обсуждали идею использования `java.util.concurrent` на соответствующем уровне абстракции, чтобы не изобретать велосипед.

[4] `http://mechanical-sympathy.blogspot.co.uk/`

[5] `https://groups.google.com/forum/?fromgroups#!forum/mechanical-sympathy`

[6] `https://github.com/real-logic`

[7] `https://github.com/real-logic/agrona`

Agrona предоставляет аналогичный набор библиотек для приложений с очень низкой задержкой. Если вы уже доказали, что стандартные библиотеки не отвечают вашим требованиям, то в качестве следующего шага имеет смысл рассмотреть эти библиотеки, прежде чем разрабатывать собственные библиотеки самостоятельно. Проект хорошо протестирован и документирован и имеет активное сообщество пользователей.

Буфера

Ричард Уорбертон (Richard Warburton) написал отличную статью[8] о буферах и проблемах с ними в Java.

В общих чертах Java предлагает класс `ByteBuffer`, который предлагает абстракцию для буфера, который может быть непосредственным (прямым) или косвенным (непрямым). Детали рассмотрены в указанной выше статье, из которой и заимствована данная классификация.

Непосредственный буфер располагается за пределами обычной кучи Java (но все же в пределах "кучи языка C" всего процесса JVM). В результате он зачастую оказывается более медленным в смысле выделения и освобождения памяти по сравнению с буфером в куче (косвенным). Непосредственный буфер обладает тем преимуществом, что JVM будет пытаться выполнять команды непосредственно над этой структурой, без промежуточного отображения.

Основная проблема `ByteBuffer` заключается в обобщенности, означающей, что к нему не применяются оптимизации, специфичные для конкретного типа буфера. Например, не поддерживаются атомарные операции, что становится проблемой при попытках создания модели в духе производителя/потребителя с использованием буфера. `ByteBuffer` требует, чтобы каждый раз при "оборачивании" новой структуры выделялся новый базовый буфер. В Agrona копирование устранено и поддерживаются четыре типа буфера с различными характеристиками, позволяя программисту определять и управлять возможными взаимодействиями с каждым объектом буфера.

- Интерфейс `DirectBuffer` обеспечивает возможность только чтения буфера и образует высший уровень иерархии.
- Интерфейс `MutableDirectBuffer` расширяет `DirectBuffer` и добавляет обращение для записи в буфер.
- Интерфейс `AtomicBuffer` расширяет `MutableDirectBuffer`, предлагая упорядоченное поведение.
- `UnsafeBuffer` представляет собой класс, который использует `Unsafe` для реализации `AtomicBuffer`.

На рис. 14.3 показана иерархия наследования классов буферов Agrona.

[8] `http://insightfullogic.com/2015/Apr/18/agronas-threadsafe-offheap-buffers/`

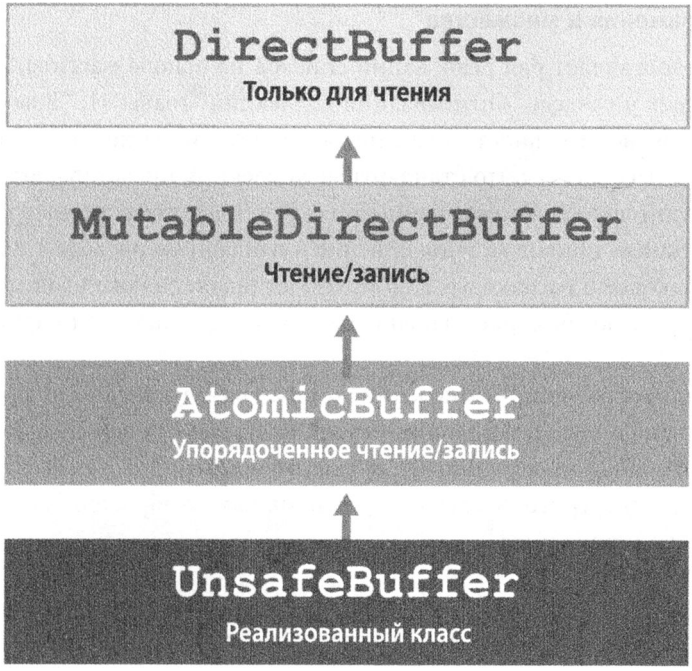

Рис. 14.3. Буфера Agrona

Код в Agrona, как вы можете представить, очень низкоуровневый и активно использующий `Unsafe`, как, например, в этом фрагменте:

```
// Эта жуть используется для того, чтобы заставить JVM
// вызвать memset(), когда адрес четный.
// TODO: проверить, применим ли этот способ в Java 9!!!
UNSAFE.putByte(byteArray, indexOffset, value);
UNSAFE.setMemory(byteArray, indexOffset + 1, length - 1, value);
```

Это не значит, что Agrona является *хаком* — совсем наоборот. Необходимость такого кода заключается в том, чтобы обойти старую оптимизацию, которая применялась внутри JVM, а теперь является антиоптимизацией. Библиотека дошла до показанного уровня детализации, чтобы обеспечить максимальную производительность.

Буфера Agrona позволяют получать доступ к базовым данным с помощью различных методов `get`, например `getLong(int index)`. Несмотря на то что буфер "обернут", разработчик должен знать, по какому индексу располагаются данные. Кроме того, операции `put` позволяют разместить значение в определенном месте в буфере. Обратите внимание, что буфер не имеет никакого единственного типа; разработчик сам должен выбирать и управлять соответствующей структурой своих данных. Проверка выхода за границы может быть включена или отключена, так что "мертвый код" может быть оптимизирован и устранен JIT-компилятором.

Списки, отображения и множества

Agrona предоставляет ряд реализаций списков на основе массива `int` или `long`. Как упоминалось в разделе "Оптимизация коллекций" главы 11, "Языковые методы повышения производительности", Java не имеет механизма непрерывного размещения объектов в массиве, так что стандартные коллекции представляют собой массив ссылок. Принудительное использование объектов (а не примитивных типов) в стандартных коллекциях приводит, в дополнение к накладным расходам памяти, к автоматической упаковке и распаковке. Agrona предоставляет также утилиты `ArrayList`, которые обеспечивают быстрое удаление из `ArrayList`, но ценой нарушения упорядоченности.

Реализации отображений и множеств Agrona хранят ключи и значения вместе в структуре хеш-таблицы. При возникновении коллизии ключей следующее значение сохраняется в таблице немедленно после указанной позиции. Такая структура хорошо подходит для быстрого доступа к примитивным отображениям, находящимся в одной и той же строке кеша.

Очереди

Agrona имеет собственный пакет параллельности, который содержит полезные параллельные утилиты и структуры, включая очереди и кольцевые буфера.

Очереди следуют стандартному интерфейсу `java.util.Queue`, так что их можно использовать вместо стандартной реализации очереди. Очереди Agrona также реализуют интерфейс `org.agrona.concurrent.Pipe`, который добавляет поддержку последовательно обрабатываемых контейнеров. В частности, для легкого взаимодействия с клиентами очередей `Pipe` добавляет поддержку операций подсчета количества элементов, емкости и очистки. Очереди не используют блокировки, но прибегают к `Unsafe`, чтобы их можно было использовать в системах с малой задержкой. Интерфейс `org.agrona.concurrent.AbstractConcurrentArrayQueue` обеспечивает первый уровень поддержки очередей, которые предоставляют разные модели производителей/потребителей. Одной из интересных частей этого API являются следующие классы:

```
/**
 * Дополняет строку кеша слева от полей производителя, чтобы
 * избежать ложного совместного использования.
 */
class AbstractConcurrentArrayQueuePadding1
{
    @SuppressWarnings("unused")
    protected long p1, p2, p3, p4, p5, p6, p7, p8, p9,
                   p10, p11, p12, p13, p14, p15;
}
/**
 * Значения производителя (Producer), которые должны быть дополнены.
```

```
 */
class AbstractConcurrentArrayQueueProducer
    extends AbstractConcurrentArrayQueuePadding1
{
    protected volatile long tail;
    protected long headCache;
    protected volatile long sharedHeadCache;
}
/**
 * Дополнение строки кеша между полями производителя и потребителя
 * для предотвращения ложного совместного использования.
*/
class AbstractConcurrentArrayQueuePadding2
    extends AbstractConcurrentArrayQueueProducer
{
    @SuppressWarnings("unused")
    protected long p16, p17, p18, p19, p20, p21, p22, p23, p24, p25,
              p26, p27, p28, p29, p30;
}
/**
 * Значения потребителя (Consumer), которые должны быть дополнены.
 */
class AbstractConcurrentArrayQueueConsumer
    extends AbstractConcurrentArrayQueuePadding2
{
    protected volatile long head;
}
/**
 * Дополнение строки кеша между полями производителя и потребителя
 * для предотвращения ложного совместного использования.
 */
class AbstractConcurrentArrayQueuePadding3
    extends AbstractConcurrentArrayQueuePadding2
{
    @SuppressWarnings("unused")
    protected long p31, p32, p33, p34, p35, p36, p37, p38, p39, p40,
              p41, p42, p43, p44, p45;
}
/**
 * Остающиеся неизменными поля очереди.
 */
public abstract class AbstractConcurrentArrayQueue<E>
    extends AbstractConcurrentArrayQueuePadding3
    implements QueuedPipe<E>
{
    ...
}
```

Следует отметить, что `sun.misc.contended` (или `jdk.internal.vm.annotation.Contended`) в будущем могут стать доступны для генерации дополнения строк кеша такого рода.

Фрагмент кода из `AbstractConcurrentArrayQueue` демонстрирует принудительно обеспечиваемую сложную компоновку памяти очереди, которая позволяет избежать ложного совместного использования, когда обращение к очереди выполняется потребителем и производителем одновременно. Причина, по которой мы требуем такое дополнение, заключается в том, что схема размещения полей в памяти не гарантируется ни языком Java, ни JVM.

Помещение производителя и потребителя в отдельные строки кеша гарантирует, что структура может адекватно работать в условиях низких задержек и высокой пропускной способности. Строки кеша используются для обращения к памяти, и если бы производитель и потребитель совместно использовали одну и ту же строку кеша, то при одновременном обращении к ней возникли бы проблемы.

Имеются три конкретные реализации, и их разделение, показанное далее, обеспечивает координацию в коде только в случае необходимости.

`OneToOneConcurrentArrayQueue`

Выбирая одного производителя и одного потребителя, мы выбираем стратегию, согласно которой единственное одновременное обращение происходит тогда, когда и производитель, и потребитель обращаются к структуре в один и тот же момент времени. Основная интересующая нас точка — позиции головы и хвоста, поскольку они обновляются одновременно только одним потоком.

Голова может быть обновлена только с помощью операции опроса или очистки, которая выполняет выборку из очереди, а хвост может быть обновлен только операцией `put()`. Выбор этого режима позволяет избежать излишней потери производительности из-за дополнительных проверок координации, которые требуются двумя другими типами очередей.

`ManyToOneConcurrentArrayQueue`

В качестве альтернативы, если мы выбрали множество производителей, для обновления позиции хвоста необходимо дополнительное управление (так как она может быть обновлена другим производителем). Использование `Unsafe.compareAndSwapLong` в цикле `while`, пока хвост не будет обновлен, гарантирует безопасный способ обновления хвоста очереди без блокировок. На стороне потребителя такого состязания нет, поскольку мы гарантируем наличие только одного потребителя.

ManyToManyConcurrentArrayQueue

Наконец, если мы выбираем много производителей и потребителей, то для обновления как головы, так и хвоста необходимы координационные меры. Этот уровень координации и управления достигается с помощью цикла `while`, который обертывает команду `compareAndSwap`. Здесь по сравнению с другими альтернативами требуется больший уровень координации, так что ее следует использовать только в случае, когда такой уровень безопасности необходим.

Кольцевые буфера

Agrona предоставляет интерфейс для обмена бинарно кодированными сообщениями для межпроцессной коммуникации `org.agrona.concurrent.RingBuffer`. Для управления хранением сообщений вне кучи используется `DirectBuffer`. Немного поупражнявшись в ASCII-художествах, мы можем показать вам в исходном коде, как сообщения хранятся в структуре `RecordDescriptor`:

```
*   0                   1                   2                   3
*   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*  |R|                          Длина                                |
*  +-+---------------------------------------------------------------+
*  |                            Тип                                |
*  +---------------------------------------------------------------+
*  |                Кодированное сообщение                         ...
* ...                                                              |
*  +---------------------------------------------------------------+
```

В Agrona реализованы два типа кольцевого буфера — `OneToOneRingBuffer` и `ManyToOneRingBuffer`. Операция записи получает буфер-источник для записи сообщения в буфер, тогда как операция чтения использует обработку в стиле обратного вызова (callback) для `onMessage`. Когда в `ManyToOneRingBuffer` пишут многие производители, ручной вызов `Unsafe.storeFence()` управляет синхронизацией памяти. Барьер записи "гарантирует отсутствие переупорядочения сохранений перед барьером с загрузками и сохранениями после него".

У Agrona имеется много структур низкого уровня и интересных аспектов. Если программная система, которую вы хотите построить, находится на этом уровне детализации, то такой проект является отличным первым шагом для экспериментов.

Существуют и другие доступные проекты, такие как JCTools[9], которые также предоставляют варианты параллельных структур очередей. Если только не требуется соответствие очень уж специфичным требованиям, следует вспомнить о существовании подобных библиотек с открытым исходным кодом, и избегать изобретения велосипеда.

[9] `https://github.com/JCTools/JCTools`

Простое бинарное кодирование

Простое бинарное кодирование (Simple Binary Encoding — SBE) было разработано для удовлетворения потребности в представлении бинарного кодирования, подходящего для производительности с низкими задержками. Это кодирование было создано специально для протокола FIX, используемого в финансовых системах.

> *Простое бинарное кодирование обеспечивает характеристики, отличные от всех прочих бинарных кодировок. Оно оптимизировано для низких задержек. Эта новая бинарная кодировка FPL дополняет существующую бинарную кодировку (FAST), разработанную в 2005 году с акцентом на сокращение полосы пропускания для рыночных данных.*
>
> — *Спецификация Simple Binary Encoding, Release Candidate 1*

SBE представляет собой решение уровня приложения, используемое для кодирования и декодирования сообщений; в SBE используются буфера Agrona. SBE оптимизировано таким образом, чтобы позволить сообщениям с низкой задержкой проходить через структуру данных без запуска сборки мусора и оптимизировать такие вещи, как доступ к памяти. Эта кодировка была разработана специально для высокочастотной торговой среды, в которой ответы на рыночные события зачастую должны следовать через микро- или наносекунды после события.

Кодировка SBE была определена в организации протокола FIX просто в противовес конкурирующим предложениям по кодированию FIX с помощью Google Protocol Buffers и ASN.1.

Другим ключевым нефункциональным требованием высокочастотной торговой среды является то, что операция должна быть быстрой постоянно. Один из авторов был свидетелем системы, которая обрабатывала сообщения с достаточно высокой пропускной способностью, а затем внезапно остановилась на две минуты из-за ошибки сборки мусора. Такая пауза совершенно неприемлема в приложении с малой задержкой, так что одним из способов обеспечения постоянной производительности является полное избежание сборки мусора. Эти типы проблем производительности будут идентифицированы с помощью теста стабильности (soak test) или иных длительных тестов производительности.

Цель приложений с малой задержкой заключается в том, чтобы выжать все возможные показатели производительности приложения. Это может привести к "гонке вооружений", когда команды конкурирующих торговых фирм пытаются превзойти одна другую в сокращении времени ожидания на критическом пути выполнения торгового приложения.

Авторы SBE предложили ряд принципов проектирования[10], отражающих эти проблемы и объясняющих их размышления. В следующих разделах мы рассмотрим некоторые проектные решения и то, как они соотносятся с разработкой системы с малой задержкой.

Технология "без копирования" и отображение Java-типов на машинные инструкции

Копирование имеет свою стоимость, и любой, кто занимался программированием на C++, вероятно, время от времени обнаруживал, что случайно копировал объекты. Копирование не такое уж дорогое, когда объекты небольшие, но по мере увеличения размера объектов увеличивается и стоимость процесса копирования.

Программисты на Java, работающие на высоком уровне, не всегда в состоянии рассмотреть эту проблему, поскольку они привыкли работать со ссылками и автоматическим управлением памятью. Используемая в SBE технология "без копирования" призвана гарантировать, что при кодировании или декодировании сообщений не используются никакие промежуточные буфера.

Непосредственная запись в базовый буфер выполняется ценой издержек проектирования: не поддерживаются большие сообщения, которые не могут разместиться в буфере. Для их поддержки требуется разработка протокола для разбиения сообщений на части и их повторной сборки.

При разработке дизайна без копирования помогают также типы, которые отображаются на удобные машинные команды. Наличие отображения, соответствующего хорошему выбору ассемблерных команд, значительно повышает производительность выборки полей.

Постоянное (стационарное) выделение памяти

Выделение памяти для объектов в Java представляет собой естественную проблему, когда речь идет о приложениях с низкой задержкой. Выделение памяти само по себе требует процессорного времени (даже если оно очень мало; например, такое как выделение TLAB); кроме того, затем возникает проблема удаления объекта после завершения его использования.

Сборка мусора часто приостанавливает работу всего приложения, приводя к паузе. Это справедливо даже для самых современных интеллектуальных сборщиков мусора, работающих в основном в параллельном режиме. Даже при ограничении абсолютного времени паузы процесс сборки мусора может внести значимое отклонение в модель производительности.

[10] `https://github.com/real-logic/simple-binary-encoding/wiki/Design-Principles`

Казалось бы естественно думать, что C++ может помочь решить эту проблему, но механизмы выделения и освобождения памяти могут вызвать проблемы. В частности, некоторые пулы памяти могут использовать механизм блокировки, который наносит ущерб производительности.

В SBE отсутствует распределение памяти, потому что он использует проектный шаблон *Приспособленец* (Flyweight) поверх базового буфера.

Потоковый доступ и доступ, выровненный по границе слова

Доступ к памяти представляет собой нечто, над чем в Java мы обычно не имеем контроля. В разделе "Оптимизация коллекций" главы 11, "Языковые методы повышения производительности", мы обсуждали ObjectLayout — предложение для хранения объектов с выравниванием, как в векторах C++. Обычно массивы в языке Java представляют собой массивы ссылок, и это означает, что последовательное чтение данных из памяти не представляется возможным.

SBE предназначается для кодирования и декодирования сообщений в прямой последовательности, которая обеспечивает хорошее выравнивание на границу слова. Без такого хорошего выравнивания могут начаться проблемы с производительностью на уровне процессора.

Работа с SBE

Сообщения SBE определены как файлы схемы XML, указывающие схему сообщений. Хотя XML в настоящее время вызывает неприятие, схемы обеспечивают хороший механизм для точной спецификации интерфейса сообщений. XML также поддерживается инструментарием интегрированных сред разработки, например, в Eclipse и Intellij.

SBE предоставляет утилиту командной строки `sbe-tool`, которая по заданной схеме позволяет генерировать соответствующие кодировщики и декодировщики. Вот как выглядят шаги для выполнения данной работы.

```
# Клонирование (или ветвление) проекта
git clone git@github.com:real-logic/simple-binary-encoding.git

# Построение проекта с помощью избранного инструментария
gradle

# Утилита sbe-tool будет создана в каталоге sbe-tool/build/libs

# Запуск sbe-tool со схемой (образец схемы имеется по адресу
# https://github.com/real-logic/simple-binary-encoding/blob/master/
#      sbe-samples/src/main/resources/example-schema.xml)
java -jar sbe-tool-1.7.5-SNAPSHOT-all.jar message-schema.xml
```

```
# По завершении работы утилита генерирует
# ряд .java-файлов в исходном каталоге

$ ls
BooleanType.java                    GroupSizeEncodingEncoder.java
BoostType.java                      MessageHeaderDecoder.java
BoosterDecoder.java                 MessageHeaderEncoder.java
BoosterEncoder.java                 MetaAttribute.java
CarDecoder.java                     Model.java
CarEncoder.java                     OptionalExtrasDecoder.java
EngineDecoder.java                  OptionalExtrasEncoder.java
EngineEncoder.java                  VarStringEncodingDecoder.java
GroupSizeEncodingDecoder.java       VarStringEncodingEncoder.java
```

Важно помнить, что одной из основных частей протокола SBE является то, что сообщения должны читаться по порядку, что, по сути, означает "так, как определено схемой". Чтобы начать работу с этими сообщениями, познакомьтесь с руководством *SBE Java Users Guide*[11].

Хотя в приведенной выше демонстрации использован инструментарий SBE командной строки, скорее всего, этот инструмент будет интегрирован в конвейер построения.

Aeron

Aeron — это последний продукт в стеке Real Logic, который мы рассмотрим. Неудивительно, что мы оставили его напоследок, ведь он основан как на SBE, так и на Agrona. Aeron — это транспорт сообщений на основе IPC (Inter-Process Communication) или на базе одно- и многоадресного UDP (User Datagram Protocol), написанного для Java и C ++. Он является нейтральным к уровню среды передачи данных, т.е. будет так же хорошо работать с InfiniBand.

По сути, это общий, всеобъемлющий протокол обмена сообщениями, который можно использовать, чтобы заставить приложения общаться одно с другим посредством IPC на одном и том же компьютере или по сети. Aeron рассчитан на максимально возможную пропускную способность и нацелен на достижение самых низких и предсказуемых задержек (как мы уже говорили выше, в разделе "Простое бинарное кодирование", стабильность производительности является крайне важной характеристикой). В этом разделе рассматривается API Aeron и обсуждаются некоторые из его проектных решений.

Зачем создавать нечто новое?

Одной из основных причин создания нового продукта, такого как Aeron, является то, что некоторые продукты на рынке становятся более обобщенными. Это не

[11] https://github.com/real-logic/simple-binary-encoding/wiki/Java-Users-Guide

критика продуктов, но когда клиенты требуют определенных возможностей (и обычно платят за них), это может подтолкнуть продукт в определенном направлении. В результате продукты начинают раздуваться и обеспечивать гораздо больше возможностей, чем предполагалось первоначально, и возможно, даже становятся каркасами.

Системы обмена сообщениями в низкоуровневом Java довольно интересны для разработки и могут начать жизнь как домашний проект внутри большой компании или в сообществе с открытым исходным кодом. Потенциальная проблема заключается в том, что отсутствие специфического опыта, необходимого с точки зрения низкой латентности, может затруднить развитие таких "домашних" проектов и их готовность к реальному промышленному применению. Создание "с нуля" продукта для работы с низкими задержками, при этом не жертвуя производительностью, является сложной задачей.

Как подчеркивалось в этой главе, важным принципом проектирования Aeron является то, что он разработан как библиотека компонентов. Это означает, что вы не привязаны к каркасу, и если вам нужна только низкоуровневая структура данных, то Agrona предоставит ее без необходимости внесения множества других зависимостей.

Издатели

Перед тем как перейти к детальному обсуждению Aeron, полезно взглянуть на некоторые из его высокоуровневых компонентов, показанных на рис. 14.4 и описанных выше.

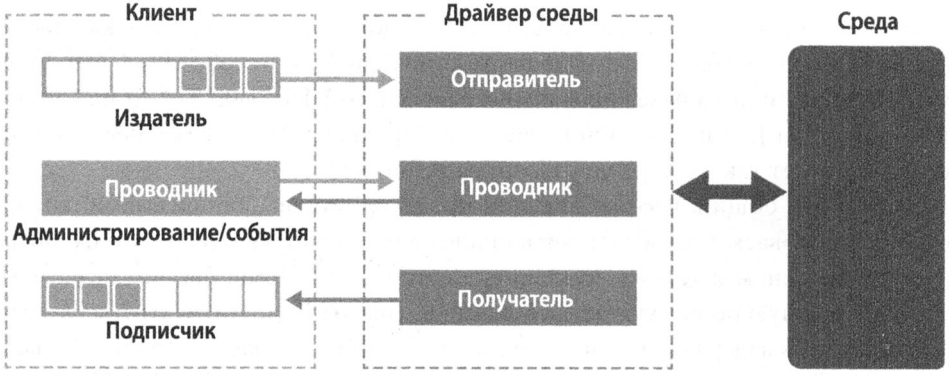

Рис. 14.4. Обзор архитектуры основных компонентов Aeron

В частности.

- *Среда* (Media) означает механизм, поверх которого осуществляет сообщение Aeron; например, это может быть UDP или IPC. Это также может быть InfiniBand или иная среда. Главное — что Aeron как клиент абстрагируется от среды.

- *Драйвер среды* (media driver) означает связь между средой и Aeron, обеспечивая настройку и коммуникации с использованием указанного транспорта.
- *Проводник* (conductor) отвечает за администрирование, такое как настройка буферов и прослушивание запросов новых подписчиков и издателей. Он также просматривает сигналы NAK (Negative Acknowledgments — отсутствие подтверждения приема) и обеспечивает повторную передачу. Проводник позволяет отправителю и получателю заниматься просто манипуляциями байтами для достижения максимальной пропускной способности.
- *Отправитель* (sender) читает данные у производителя и отправляет их в сокеты.
- *Получатель* (receiver) читает данные из сокетов и передает их соответствующим каналам и сеансам.

Драйвер среды обычно представляет собой отдельный процесс, предоставляющий буфера, которые будут использоваться для передачи и приема сообщений. Для разных аппаратных конфигураций могут использоваться различные драйверы среды; конфигурация, которая настраивает оптимизацию для этого драйвера, — `MediaDriver.Context`. Драйвер среды можно также запустить встроенным в рамках одного и того же процесса; встроенный процесс может быть настроен с помощью контекста или с использованием системных свойств. Запуск встроенного драйвера среды можно выполнить следующим образом:

```
final MediaDriver driver = MediaDriver.launch();
```

Приложения Aeron должны подключаться к драйверу среды либо как издатели, либо как подписчики. Класс `Aeron` делает это довольно простым действием. `Aeron` имеет также внутренний класс `Context`, который можно использовать для настройки параметров:

```
final Aeron.Context ctx = new Aeron.Context();
```

Затем Aeron может подключиться к публикации для передачи по данному каналу и потоку. Поскольку `Publication` является `AutoClosable`, объект будет автоматически очищен, когда блок `try` завершит выполнение:

```
try (Publication publication = aeron.addPublication(CHANNEL, STREAM_ID))
{...}
```

Чтобы отправить сообщение, издателю предлагается буфер, и результат предложения определяет состояние сообщения. `Publication` имеет ряд констант типа `long`, которые представляют собой отображение ошибок, которые можно сравнить с результатом метода `offer()`, также имеющим тип `long`:

```
final long result = publication.offer(BUFFER, 0, messageBytes.length);
```

Отправка сообщения столь же простая, но для того чтобы она могла быть использована, подписчик должен прослушивать тот же драйвер среды.

Подписчики

Запуск подписчика аналогичен загрузке издателя: к нему должен быть подключен драйвер среды, а затем подсоединен клиент Aeron. Компоненты потребителя отражают изображение, показанное на рис. 14.4. Пользователь регистрирует функцию обратного вызова, которая запускается при получении сообщения:

```
final FragmentHandler fragmentHandler =
    SamplesUtil.printStringMessage(STREAM_ID);

try (Aeron aeron = Aeron.connect(ctx);
     Subscription subscription =
         aeron.addSubscription(CHANNEL, STREAM_ID))
{
    SamplesUtil.subscriberLoop(fragmentHandler,
        FRAGMENT_COUNT_LIMIT, running).accept(subscription);
}
```

Эти примеры демонстрируют только что рассмотренные основные настройки; проект Aeron содержит более сложные примеры[12].

Дизайн Aeron

Рассказ Мартина Томпсона (Martin Thompson) в Strange Loop[13] дает очень хорошее представление об Aeron и о причине его создания. В этом разделе будут рассмотрены некоторые из видеообсуждений в сочетании с открытой документацией.

Требования к транспорту

Aeron представляет собой четвертый транспортный уровень OSI для сообщений; это означает, что есть ряд требований, которым он должен удовлетворять.

Порядок (Ordering))

Пакеты от транспорта более низкого уровня будут получаться неупорядоченными, так что Aeron отвечает за переупорядочение находящихся вне верной последовательности сообщений.

Надежность (Reliability)

Проблемы возникают при потере данных, так что в этом случае должен быть сделан запрос на повторную передачу потерянных данных. Пока выполняется

[12] https://github.com/real-logic/aeron/tree/master/aeron-samples/src/main/java/io/aeron/samples

[13] https://youtu.be/tM4YskS94b0

запрос старых данных, процесс получения новых данных не должен создавать ему помех. Надежность в этом контексте означает надежность уровня соединения, а не надежность уровня сеанса (например, отказоустойчивость при перезапуске нескольких процессов).

Обратное давление (Back pressure)

Под этим термином понимается ситуация, когда данные поступают быстрее, чем система может их обрабатывать. В сценариях с большими объемами данных подписчики будут испытывать давление, поэтому служба должна поддерживать контроль потока и принимать соответствующие меры.

Затор (Congestion)

Это проблема, которая может возникать в насыщенных сетях, но при создании приложения с малой задержкой она не должна быть первоочередной. Aeron предоставляет необязательную возможность активации контроля перегрузок; пользователи, находящиеся в сетях с малой задержкой, могут ее отключить, а пользователи, чувствительные к другому трафику, могут ее включить. Управление заторами может влиять на продукт, который в оптимальных условиях выполнения имеет достаточную пропускную способность сети.

Мультиплексирование (Multiplexing)

Транспорт должен быть способен обрабатывать передачу нескольких потоков данных по одному каналу без ущерба для общей производительности.

Задержка и принципы приложения

Aeron управляется восемью проектными принципами, описанными в Aeron Wiki[14].

В установившемся состоянии сборки мусора нет

Паузы сборки мусора — причина задержек и непредсказуемости Java-приложений. Aeron разработан таким образом, чтобы гарантировать, что в установившемся состоянии сборка мусора будет отсутствовать. Это, помимо прочего, означает, что Aeron может быть включен в приложения, которые соблюдают это же дизайнерское решение.

Алгоритм Smart Batching на пути сообщений

Smart Batching представляет собой алгоритм, который предназначен для обработки получения пакетных сообщений. Во многих системах обмена сообщениями нельзя предполагать, что сообщения будут приниматься постоянно

[14] `https://github.com/real-logic/aeron/wiki/Design-Principles`

в течение дня. Сообщения зачастую поступают пакетами из-за определенных бизнес-событий. Если при обработке сообщения получено другое сообщение, оно также может быть включено в тот же сетевой пакет, вплоть до заполнения доступной емкости. Используя соответствующие структуры данных, Aeron позволяет работать пакетно, без остановки производителей, осуществляющих запись в совместно используемый ресурс.

Алгоритмы без блокировок на пути сообщений

Блокировка приводит к конфликтным ситуациям, когда одни потоки могут быть заблокированы, а другие в это же время выполняются; даже захват и освобождение блокировки могут замедлить приложение. Избегание блокировок устраняет замедление, которое они могут вызвать. Блокировки и технологии, свободные от блокировок, рассматриваются в главе 12, "Методы повышения производительности параллельной работы".

Неблокирующий ввод-вывод на пути сообщений

Блокирующий ввод-вывод может приводить к блокировке потока, а стоимость пробуждения последнего достаточно высока. Применение неблокирующих операций ввода-вывода позволяет избежать этих расходов.

Отсутствие исключительных ситуаций на пути сообщений

Приложения тратят большую часть своего времени на выполнение основных сценариев, а не на мелкие сценарии для редких исключительных ситуаций. Такие исключительные ситуации должны быть обработаны, но не ценой скорости выполнения основных сценариев.

Применение принципа единственного писателя

Как говорилось при обсуждении `ManyToOneConcurrentArrayQueue` в разделе "Очереди" ранее в этой главе, наличие нескольких писателей связано с высокой степенью контроля и координации доступа к очереди. Наличие единственного писателя значительно упрощает эту стратегию и уменьшает конфликты и состязательность при записи данных. Объекты публикации Aeron являются потокобезопасными и поддерживают несколько писателей, но подписчики потокобезопасными не являются, поэтому для каждого потока, на который вы хотите выполнить подписку, требуется один автор.

Предпочтительны состояния, не используемые совместно

Единственный писатель решает проблему конфликтов в очереди, но при этом добавляет еще одну точку, в которой совместно используются изменяемые данные. Поддержание закрытого или локального состояния гораздо

предпочтительнее во всех сферах разработки программного обеспечения, поскольку значительно упрощает используемую модель данных.

Избегание излишнего копирования данных

Как мы уже упоминали, обычно копирование данных достаточно дешево, но инвалидация строк кеша и потенциальное вытеснение их другими данными приводит к проблемам как в Java, так и в C++. Минимизация копирования помогает предотвратить случайные сбои памяти.

Как это работает за кулисами

Многие существующие протоколы в попытках построения эффективных систем обработки сообщений вводят усложненные структуры данных, такие как списки с пропусками. Эта усложненность, главным образом за счет косвенности указателей, приводит к системам, которые имеют непредсказуемые характеристики задержек.

По сути, Aeron создает реплицированный перманентный журнал сообщений.

— Мартин Томпсон (Martin Thompson)

Aeron был разработан, чтобы обеспечить самый чистый и простой способ построения последовательности сообщений в структуре. Хотя первоначально это может показаться не самым подходящим выбором, Aeron активно использует концепцию файла. Файлы представляют собой структуры, которые могут совместно использоваться заинтересованными процессами, а возможность отображения файла на память в Linux перенаправляет все обращения к файлу в память, а не в физический файл.

По умолчанию Aeron выполняет отображение на временное файловое хранилище `tmpfs` (которое является оперативной памятью, смонтированной как файл). При этом достигается более высокая производительность, чем при работе с отображенным в память дисковым файлом.

Чтобы отслеживать, где было записано последнее сообщение, используется указатель на хвост. На рис. 14.5 показано одно записываемое в текущий файл сообщение вместе с заголовком.

Весьма интересным является упорядочение событий. Указатель хвоста резервирует пространство для сообщения в файле. Приращение хвоста является атомарным, поэтому писатель знает начало и конец своего раздела.

Критическое обновление со встроенной функцией для этого атомарного увеличения было введено как часть Java 8.

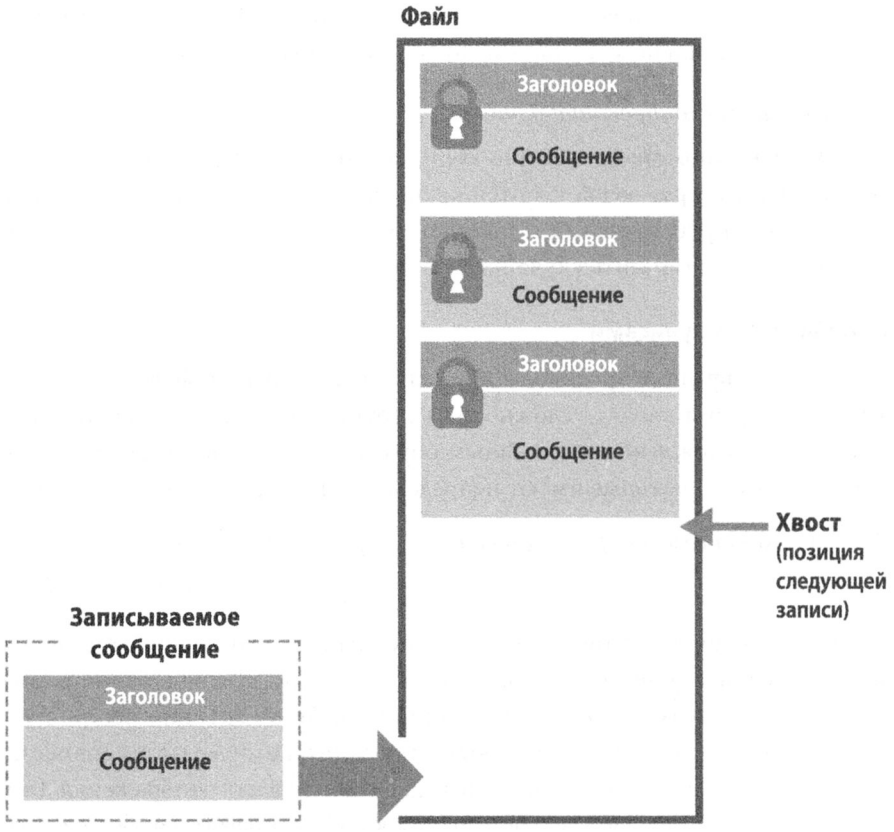

Рис. 14.5. Сообщения, записанные в файл

Это позволяет нескольким писателям обновлять файл без использования блокировок, что, по сути, устанавливает протокол записи файлов. Затем сообщение записывается, но как можно утверждать, что оно закончено? Заголовок — это последнее, что нужно атомарно записать в файл. Его наличие говорит нам, что сообщение завершено.

Файлы — это перманентные структуры, растущие при записи и не изменяющиеся. Чтение записей из файла не требует блокировок, так как процесс может открыть файл только для чтения. Но можно ли иметь файл, растущий бесконечно?

Это приводит к множеству проблем из-за ошибок отсутствия страниц в памяти и сбоям памяти в нашем отображенном в память файле. Мы решаем эти проблемы с помощью трех файлов: *активного* (active), *грязного* (dirty) и *чистого* (clean). Активный файл — это файл, который записывается в настоящее время, грязный — был записан ранее, а чистый — следующий файл для записи. Чтобы избежать задержек, вызванных большими файлами, выполняется ротация файлов.

Сообщениям не разрешается размещаться в разных файлах. Если хвост выходит за конец активного файла, процесс вставки выполняет заполнение оставшейся части файла и записывает сообщение в чистый файл. Из грязного файла можно выполнять архивацию и постоянное глубокое сохранение журнала транзакций.

Механизм обработки отсутствующих сообщений также очень интеллектуальный и избегает применения списков с пропусками и других упоминавшихся ранее структур. Заголовок сообщения содержит информацию о порядке. При вставке выпадающего из последовательности сообщения следует оставить место для предыдущих сообщений. Когда отсутствующее сообщение получено, его можно вставить в нужное место в файле. Это дает постоянно растущую последовательность сообщений без каких-либо пропусков или других структур. Инкрементное упорядочение данных приносит дополнительную выгоду невероятно быстрой работы с точки зрения механического взаимопонимания.

Водяной знак представляет текущую позицию последнего полученного сообщения. Если водяной знак и хвост различны в течение определенного периода времени, это указывает на отсутствующие сообщения. Чтобы решить эту проблему, для их запроса отправляется сигнал NAK (negative acknowledge — отсутствие подтверждения приема). NAK может быть отправлен для получения сообщения и подтвержден после того, как сообщение получено.

Одним из интересных побочных эффектов этого протокола является то, что каждое полученное сообщение имеет уникальный способ идентификации байтов в каждом сообщении на основе `streamId`, `sessionId`, `termId` и `termOffset`. Для записи и воспроизведения потоков сообщений может использоваться Aeron Archive[15]. Объединив архив и это уникальное представление, можно однозначно идентифицировать все сообщения на протяжении всей истории.

Файл журнала является основой способности Aeron поддерживать скорость и состояние. Это также простой, изящно выполненный дизайн, который позволяет изделию конкурировать (а во многих случаях и побеждать) хорошо зарекомендовавшие себя (и дорогостоящие) многоадресные продукты.

Резюме

Ведение журнала является неотъемлемой частью всех приложений производственного уровня, а тип используемого регистратора может существенно влиять на общую производительность приложений. Когда речь заходит о протоколировании (а не только о выполнении инструкции записи в журнал), а также о влиянии, которое оно оказывает на другие подсистемы JVM, такие как использование потоков и сбор мусора, важно рассматривать все приложение в целом.

[15] `https://github.com/real-logic/aeron/wiki/Aeron-Archive`

В этой главе содержатся некоторые простые примеры библиотек с низкой задержкой, начиная с самого низкого уровня и вплоть до полной реализации систем обмена сообщениями. Понятно, что цели и задачи систем с малой задержкой должны применяться во всем программном стеке, от очередей самого низкого уровня и до верхнего уровня приложения. Системы с малой задержкой и высокой пропускной способностью требуют большого искусства, сложных решений и контроля, и многие из обсуждаемых здесь проектов с открытым исходным кодом были построены на основе огромного опыта их разработчиков. Когда вы создаете новую систему с низкой задержкой, применение этих проектов экономит вам дни (если не недели) времени разработки, при условии что вы придерживаетесь целей проектирования низкого уровня вплоть до верхнего уровня приложения.

В главе рассказывается, в какой степени Java и JVM могут использоваться для высокопроизводительных приложений. Написание высокопроизводительных приложений с низкими задержками представляет собой сложную задачу на любом языке, но Java по сравнению с другими языками предоставляет одни из лучших инструментов и возможностей получения высокой производительности. Однако Java и JVM вносят дополнительный уровень абстракции, которым требуется управлять, а в некоторых случаях — обойти его. При этом важно учитывать имеющееся аппаратное обеспечение, производительность JVM и множество проблем работы с низким уровнем.

Эти проблемы более низкого уровня во время повседневной разработки на Java обычно никак не проявляются. Использование новых библиотек протоколирования, которые не содержат выделения памяти, а также структуры данных и протоколы обмена сообщениями, обсуждаемые в этой главе, значительно облегчают жизнь разработчиков, поскольку большая часть сложностей уже решена сообществом разработчиков программ с открытым исходным кодом.

ГЛАВА 15

Java 9 и будущие версии

Во время написания этой книги Java 9 находился в активной разработке. Новый выпуск содержит ряд возможностей, связанных с производительностью, и усовершенствований, важных для прикладных программистов Java/JVM.

В первой части этой главы мы рассмотрим новые и измененные аспекты платформы Java 9, о которых должны знать инженеры в области производительности.

Для большинства разработчиков истина заключается в том, что Java 9 состоит из "модулей и всего остального". Так же, как Java 8 представлялся, в первую очередь, как лямбда-выражения и следствия из них (потоки данных, методы по умолчанию и мелкие аспекты функционального программирования), так и Java 9 в основном представляет собой модули.

Модули — это новое средство построения и развертывания программного обеспечения, и их нелегко принять по частям. Они предоставляют очень современный способ построения хорошо спроектированных приложений. Тем не менее может потребоваться несколько команд и проектов, просто чтобы увидеть долгосрочные преимущества внедрения модулей. С точки зрения наших целей модули не имеют какого-либо реального влияния на производительность, поэтому мы не делаем никаких попыток подробного их обсуждения, и вместо этого сосредоточиваемся на небольших изменениях, оказывающих большее влияние на производительность.

Читатели, заинтересованные в информации о модулях Java 9, могут обратиться к соответствующей литературе, такой как книга издательства O'Reilly *Java 9 Modularity*[1] Сандера Мака (Sander Mak) и Пола Бэккера.

Большая часть этой главы посвящена обсуждению будущего Java, каким оно представляется на момент написания данной книги. В экосистеме платформы Java есть ряд инициатив, которые могут радикально изменить ландшафт производительности для приложений JVM. Чтобы завершить книгу, мы рассмотрим эти проекты и их важность для профессионалов в области производительности Java.

[1] http://shop.oreilly.com/product/0636920049494.do

Небольшие улучшения производительности в Java 9

В этом разделе обсуждаются усовершенствования Java 9, имеющие отношение к производительности. Некоторые из них весьма невелики, но могут оказаться значительными для некоторых приложений. В частности, мы обсудим такие изменения, как:

- сегментированный кеш кода;
- компактные строки;
- новая конкатенация строк;
- усовершенствования компилятора C2;
- изменения G1.

Сегментированный кеш кода

Одним из усовершенствований Java 9 является разделение кеша кода[2] на отдельные области:

- код, не принадлежащий методам, такой как код интерпретатора;
- профилированный код (уровни 2 и 3 компилятора клиента);
- непрофилированный код (уровни 1 и 4).

Это должно привести к сокращению времени выметания (область кода, не принадлежащего методам, выметания не требует) и большей локальности полностью оптимизированного кода. Недостатком сегментированного кеша кода является возможность переполнения одной области, в то время как в других областях имеется свободное место.

Компактные строки

В Java содержимое строки всегда хранилось как `char[]`. Поскольку `char` является 16-битным типом, это означает, что для хранения ASCII-строки используется примерно вдвое больше памяти, чем действительно необходимо. Платформа всегда рассматривала эти накладные расходы как цену, которую стоит заплатить за упрощение обработки Unicode.

В Java 9 появилась возможность использовать *компактные строки* (compact strings). Это возможность оптимизации на уровне строк. Если строка может быть представлена кодировкой Latin-1, она представляется как массив байтов (с байтами, соответствующими символам Latin-1), так что экономятся пустые нулевые байты представления типа `char`. В исходном коде класса Java 9 `String` это изменение выглядит следующим образом:

[2] `http://openjdk.java.net/jeps/197`

```
private final byte[] value;
/**
 * Идентификатор кодировки используется для кодировки байтов в
 * {@code value}. В данной реализации поддерживаются значения
 *
 * LATIN1
 * UTF16
 *
 * @implNote Это поле является доверенным полем VM и предметом для
 * свертки констант, если экземпляр String является константой.
 * Перезапись поля после построения вызовет проблемы.
 */
private final byte coder;
static final byte LATIN1 = 0;
static final byte UTF16 = 1;
```

В Java 9 поле `value` теперь имеет тип `byte[]`, а не `char[]`, как в более ранних версиях.

Эту возможность можно отключить или включить с помощью параметров `-XX:-CompactStrings` и `-XX:+CompactStrings` (по умолчанию).

Это изменение наиболее сильно влияет на приложения, куча которых содержит много строковых данных, которые представляют собой строки в кодировке Latin-1 (или ASCII), например ElasticSearch, кеши и другие связанные с ними компоненты. Для таких приложений переход к среде выполнения Java 9 может быть оправдан одним лишь этим улучшением.

Новая конкатенация строк

Рассмотрим следующий простой фрагмент кода Java:

```
public class Concat
{
    public static void main(String[] args)
    {
        String s = "("+ args[0] + " : "+ args[1] +")";
        System.out.println(s);
    }
}
```

Начиная с Java 5 эта функция языка была превращена в ряд вызовов методов с участием типа `StringBuilder`, давая в результате довольно большое количество байт-кода:

```
public static void main(java.lang.String[]);
  Code:
```

```
    0: new           #2 // Класс java/lang/StringBuilder
    3: dup
    4: invokespecial #3 // Метод java/lang/StringBuilder."<init>":()V
    7: ldc           #4 // String (
    9: invokevirtual #5 // Метод java/lang/StringBuilder.append:
                        // (Ljava/lang/String;)Ljava/lang/StringBuilder;
   12: aload_0
   13: iconst_0
   14: aaload
   15: invokevirtual #5 // Метод java/lang/StringBuilder.append:
                        // (Ljava/lang/String;)Ljava/lang/StringBuilder;
   18: ldc           #6 // String :
   20: invokevirtual #5 // Метод java/lang/StringBuilder.append:
                        // (Ljava/lang/String;)Ljava/lang/StringBuilder;
   23: aload_0
   24: iconst_1
   25: aaload
   26: invokevirtual #5 // Метод java/lang/StringBuilder.append:
                        // (Ljava/lang/String;)Ljava/lang/StringBuilder;
   29: ldc           #7 // String )
   31: invokevirtual #5 // Метод java/lang/StringBuilder.append:
                        // (Ljava/lang/String;)Ljava/lang/StringBuilder;
   34: invokevirtual #8 // Метод java/lang/StringBuilder.toString:
                        // ()Ljava/lang/String;
   37: astore_1
   38: getstatic     #9 // Поле java/lang/System.out:
                        // Ljava/io/PrintStream;
   41: aload_1
   42: invokevirtual #10 // Метод java/io/PrintStream.println:
                         // (Ljava/lang/String;)V
   45: return
```

Однако в Java 9 компилятор генерирует совершенной иной байт-код:

```
public static void main(java.lang.String[]);
  Code:
    0: aload_0
    1: iconst_0
    2: aaload
    3: aload_0
    4: iconst_1
    5: aaload
    6: invokedynamic  #2, 0 // InvokeDynamic #0:makeConcatWithConstants:
                            // (Ljava/lang/String;Ljava/lang/String;)
                            // Ljava/lang/String;
   11: astore_1
   12: getstatic     #3    // Поле java/lang/System.out:
                            // Ljava/io/PrintStream;
```

```
15: aload_1
16: invokevirtual #4    // Метод java/io/PrintStream.println:
                        // (Ljava/lang/String;)V
19: return
```

Этот пример основан на коде `invokedynamic`, который был представлен в разделе "Введение в байт-код JVM" главы 9, "Выполнение кода в JMV". Глядя на подробный вывод `javap`, мы видим метод самозагрузки в пуле констант:

```
0: #17 REF_invokeStatic java/lang/invoke/StringConcatFactory.
  makeConcatWithConstants:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/
  lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/String;
  [Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
```

Здесь использован фабричный метод под названием `makeConcatWithConstants()` от `StringConcatFactory` для создания метода конкатенации. Эта технология может использовать ряд различных стратегий, включая написание байт-кода для нового пользовательского метода. В определенной мере это схоже с использованием подготовленного оператора для выполнения SQL вместо простой сборки строк.

Влияние этих небольших изменений на общую производительность, как ожидается, для многих приложений не будет существенным. Однако изменения указывают на более широкое использование `invokedynamic` и иллюстрируют общее направление эволюции платформы.

Усовершенствования компилятора C2

В настоящее время компилятор C2 достаточно зрелый, и широко распространено мнение (например, у таких компаний, как Twitter, и даже у таких экспертов, как Клифф Клик (Cliff Click)), что в рамках существующего проекта никакие крупные усовершенствования более невозможны. Это означает, что любые улучшения здесь, по сути, минимальны. Однако есть одна область, которая потенциально может обеспечить более высокую производительность, — это использование расширений *SIMD* (Single Instruction, Multiple Data — одна команда, много данных), имеющихся в современных процессорах.

По сравнению с другими языками программирования, Java и JVM находятся в достаточно хорошем положении для использования SIMD благодаря следующим возможностям платформы:

- байт-код не зависит от платформы;
- JVM выполняет проверку процессора при запуске, так что ей известно, какими именно его возможностями можно воспользоваться при работе;
- JIT-компиляция представляет собой динамическую генерацию кода, так что она может использовать все доступные команды процессора.

Как уже говорилось в разделе "Встроенные операции" главы 10, "JIT-компиляция", средством реализации этих улучшений являются встроенные функции виртуальной машины.

> *Метод является встраиваемым[3], когда виртуальная машина HotSpot заменяет аннотированный метод написанным вручную на ассемблере и/или на промежуточном представлении компилятора встраиваемым компилятором кодом для повышения производительности.*
>
> — `@HotSpotIntrinsicCandidate` *JavaDoc*

HotSpot поддерживает некоторые команды x86 SIMD, включая следующие.

- Автоматическая векторизация кода Java.
- Оптимизация SuperWord (форма автоматической векторизации) в C2 для получения кода SIMD из последовательного кода.
- Встраиваемый код SIMD в JVM, включая копирование массивов, их заполнение и сравнение.

Выпуск Java 9 содержит ряд исправлений, которые могут быть улучшены путем применения SIMD и иных подобных возможностей процессора. Из примечаний к выпуску следует, что устранены и улучшены путем применения встроенного кода (intrinsic) следующие проблемы.

- Маскированная (с выборочным применением) векторизация циклов.
- Использование оптимизации SuperWord в ходе анализа развертывания циклов.
- Многоверсионное (в смысле поддержки различных вариантов) устранение проверки выхода за границы диапазона.
- Поддержка векторизации `sqrt` с двойной точностью.
- Улучшенная векторизация параллельных потоков.
- Расширение SuperWord для поддержки векторного условного перемещения (CMovVD) в процессорах Intel AVX.

В целом встроенные функции (intrinsic) должны рассматриваться как точечные исправления, а не общие методы. Они обладают тем преимуществом, что они мощные, легкие и гибкие, но имеют потенциально высокие затраты на разработку и техническое обслуживание, так как должны поддерживаться для нескольких архитектур. Технологии SIMD полезны и желательны в применении, но очевидно, что отдача от них инженерам по настройке производительности сокращается.

---

[3] Не путать это с inline-оптимизацией компилятора, когда тело метода вставляется вместо его вызова. — *Примеч. ред.*

Новая версия G1

Как описано в разделе "G1" главы 7, "Вглубь сборки мусора", сборщик мусора G1 предназначен для решения сразу нескольких задач, предлагая такие функции, как более легкая настройка и лучший контроль времени паузы. В Java 9 он стал сборщиком мусора по умолчанию. Это означает, что на приложения, переносимые с Java 8 на Java 9, которые явно не устанавливают используемый сборщик мусора, будет влиять изменение алгоритма сборки мусора. Но это не все — версия G1, которая поставляется с Java 9, отличается от версии в Java 8.

Компания Oracle заявила, что, согласно ее микротестам, новая версия работает значительно лучше, чем версия, представленная в Java 8. Однако это утверждение не поддержано открытой публикацией каких-либо результатов исследований, и пока что в лучшем случае у нас есть только какие-то слухи.

Надеемся, это изменение алгоритма не повлияет отрицательно на большинство приложений. Однако все приложения, переносимые на платформу Java 9, должны пройти полный тест производительности (если они использовали Java 8 со сборщиком мусора по умолчанию или G1).

Java 10 и будущие версии

На момент написания книги только-только вышла версия Java 9. В результате теперь усилия разработчиков полностью переключились на следующую версию платформы — Java 10. В этом разделе, перед тем как приступить к рассмотрению того, что нам известно о Java 10, мы обсудим модель нового выпуска, которая будет в силе, пока не выйдет новая версия.

Процесс выпуска новых реализаций

За несколько дней до выпуска Java 9 Oracle анонсировала совершенно новую модель выпусков Java начиная с Java 10. Исторически выпуски были ориентированы на определенные функциональные особенности; крупные изменения готовились для конкретного выпуска, и при необходимости выпуск новой версии откладывался до тех пор, пока требуемая функциональная возможность не была готова. Этот подход вызвал значительные задержки с выпуском Java 9, а также повлиял на выпуск Java 8.

Цикл выпусков, управляемый функциональными возможностями, имел гораздо более глубокие последствия для общей скорости разработки платформы Java. Так, крупные функциональные изменения фактически блокировали разработку более мелких возможностей из-за длительных циклов, необходимых для создания и полного тестирования выпуска. Отсрочка выпуска, близкого к завершению цикла разработки, означает, что репозитории исходных текстов находятся в заблокированной

или полузаблокированной форме для гораздо большего процента доступного цикла разработки.

Начиная с Java 10 проект перешел на строгую временную модель. Новая версия Java, содержащая новые функциональные возможности, будет выпускаться каждые шесть месяцев. Эти выпуски будут именоваться *функциональными выпусками* (feature releases) и будут эквивалентны основным выпускам старой модели.

Функциональные выпуски, как правило, не будут содержать такого количества новых возможностей или изменений, как основные выпуски старой модели Java. Тем не менее крупные изменения также будут осуществляться через функциональные выпуски.

Oracle планирует предоставлять *долгосрочную поддержку* (Long-Term Support, LTS) для некоторых функциональных выпусков. Сюда будут входить только те выпуски, которые Oracle будет выпускать с закрытым JDK. Все прочие выпуски будут представлять собой бинарные файлы OpenJDK под открытой лицензией GNU (GNU Public License — GPL) с привилегией Classpath (т.е. с возможностью объединять этот код со своими библиотеками, которые не следуют GPL) — той же лицензией, которая всегда использовалась для проектов Java с открытыми исходными кодами. Другие поставщики могут также предлагать поддержку своих бинарных файлов, при этом приняв решение поддерживать и иные выпуски, помимо LTS-версий.

Java 10

На момент написания книги информация о Java 10 все еще находится в неопределенном состоянии. Таким образом, существует возможность значительных изменений объема и содержания между тем, что имеется в настоящий момент и что будет официально выпущено в свет. Например, в течение нескольких недель сразу после выпуска Java 9 шли публичные дебаты о схеме нумерации версий, которая будет использоваться в будущем.

Новые возможности и улучшения JVM отслеживаются с помощью процесса улучшения Java — Java Enhancement Process[4]. Каждое предложение по улучшению JDK (JDK Enhancement Proposal — JEP) имеет номер, который позволяет его отслеживать. Вот основные функциональные возможности, поставляемые как часть Java 10; не все из них связаны с производительностью или вообще оказывают влияние на разработчиков.

- 286. Вывод типов локальных переменных.
- 296. Объединение репозиториев JDK в единый репозиторий.
- 304. Интерфейс сборщика мусора.

[4] `http://openjdk.java.net/jeps/1`

- 307. Полностью параллельная сборка мусора в G1.
- 310. Совместное использование классов в приложениях (классы могут размещаться в разделяемых архивах).
- 312. Обратные вызовы из процессов. В данном случае будет обеспечена поддержка обращений на локальном уровне, без блокировки всей виртуальной машины (Thread-Local Handshakes).

JEP 286 позволяет разработчику сократить количество стереотипного кода в объявлениях локальных переменных, так что следующий код становится корректным кодом Java:

```
var list = new ArrayList<String>(); // Вывод ArrayList<String>
var stream = list.stream();         // Вывод Stream<String>
```

Этот синтаксис будет ограничен локальными переменными с инициализаторами и локальными переменными циклов `for`. Эта возможность, конечно, реализована исключительно в компиляторе исходного кода и не оказывает никакого реального влияния на байт-код или производительность. Тем не менее обсуждение и реакция на это изменение иллюстрируют один важный аспект дизайна языков программирования, известный как закон Уадлера (по имени функционального программиста и компьютерного ученого Филиппа Уадлера (Philip Wadler)):

> *Эмоциональная интенсивность дебатов по языковой функциональной возможности возрастает по мере того, как вы перемещаетесь вниз по следующей шкале: семантика, синтаксис, лексика, комментарии.*

Среди прочих изменений JEP 296 является чисто вспомогательным, а JEP 304 увеличивает изоляцию кода различных сборщиков мусора и вводит интерфейс сборщиков мусора в JDK. Ни то, ни другое изменение не влияет на производительность.

Остальные три изменения имеют некоторое (хотя и потенциально небольшое) влияние на производительность. JEP 307 решает проблему со сборщиком мусора G1, о которой мы ранее не говорили; если он когда-либо возвращается к полной сборке мусора, следует ожидать неприятного удара по производительности. Начиная с Java 9 текущая реализация полной сборки мусора для G1 использует однопоточный (т.е. последовательный) алгоритм маркировки-выметания-уплотнения (mark-sweep-compact). Цель JEP 307 состоит в том, чтобы распараллелить этот алгоритм, так что в маловероятном случае в G1 может использоваться такое же количество потоков, как и в параллельных сборщиках.

JEP 310 расширяет функциональную возможность под названием *Class-Data Sharing* (CDS), которая была введена в Java 5. Идея состоит в том, что JVM обрабатывает набор классов и записывает их в совместно используемый архивный файл. Затем этот файл можно отобразить в память при следующем выполнении приложения, сокращая тем самым время запуска. Он также может совместно использоваться

разными JVM и, таким образом, уменьшать общий объем используемой памяти, когда несколько JVM работают на одном компьютере.

По состоянию на Java 9 CDS позволяет начальному загрузчику классов загружать заархивированные классы. Цель этого JEP — расширение данного поведения таким образом, чтобы позволить прикладным и пользовательским загрузчикам классов использовать архивные файлы. Эта возможность существует, но в настоящее время она доступна только в Oracle JDK, но не в OpenJDK. JEP, таким образом, по сути, переводит эту функцию в открытый репозиторий из закрытых исходных текстов Oracle.

Наконец, JEP 312 закладывает основу для повышения производительности виртуальных машин, позволяя выполнять обратный вызов в потоках приложений без выполнения глобальной точки безопасности VM. Это означает, что JVM может останавливать отдельные потоки, а не только все сразу. Некоторые из улучшений, которые достигаются благодаря данному изменению, включают:

- уменьшение воздействия на производительность при получении экземпляра трассировки стека;
- обеспечение лучшей выборки трассировки стека путем уменьшения зависимости от сигналов; улучшение смещения блокировки путем остановки только отдельных потоков для отмены смещения;
- устранение некоторых барьеров памяти в JVM.

В целом, Java 10 вряд ли содержит какие-либо крупные повышения производительности; скорее это просто первый выпуск нового, более частого и постоянного процесса выпуска версий Java.

`Unsafe` в Java версии 9 и выше

Никакое обсуждение будущего Java не было бы полным без упоминания споров, связанных с классом `sun.misc.Unsafe` и связанных с ним неприятностей. Как мы видели в разделе "Построение параллельных библиотек" главы 12, "Методы повышения производительности параллельной работы", `Unsafe` является внутренним классом, который не является частью стандартного API, хотя начиная с Java 8 стал стандартом де-факто.

С точки зрения разработчика библиотек, `Unsafe` содержит смесь различных функциональных возможностей различной степени безопасности. Методы, такие как используемые для доступа к аппаратной команде CAS, в основном полностью безопасны, но нестандартны. Другие методы не являются дистанционно (удаленно) безопасными (в смысле использования RMI) и включают такие вещи, как эквивалент арифметики указателей. Однако некоторая из "удаленно небезопасных" функциональностей не может быть получена каким-либо иным способом. Oracle относится

к этим возможностям как к *критическим внутренним API*, что и описано в соответствующем JEP[5].

Основная проблема заключается в том, что без замены некоторых функциональных возможностей в `sun.misc.Unsafe` и его друзьях основные каркасы и библиотеки не смогут продолжать корректно функционировать. Это, в свою очередь, косвенно влияет на все приложения, использующие широкий диапазон каркасов, а в современной среде это практически каждое приложение экосистемы.

В Java 9 был добавлен переключатель `--illegal-access` для управления доступностью времени выполнения к этим API. Критические внутренние API-интерфейсы в будущей версии должны быть заменены поддерживаемыми альтернативами, но завершить эту работу полностью до выпуска Java 9 не удалось. В результате должен поддерживаться доступ к следующим классам.

- `sun.misc.{Signal,SignalHandler}`
- `sun.misc.Unsafe`
- `sun.reflect.Reflection::getCallerClass(int)`
- `sun.reflect.ReflectionFactory`
- `com.sun.nio.file.{ExtendedCopyOption,ExtendedOpenOption, Extended WatchEventModifier,SensitivityWatchEventModifier}`

В Java 9 эти API определены и экспортируются модулем JDK `jdk.unsupported`, который имеет следующее объявление:

```
module jdk.unsupported
{
    exports sun.misc;
    exports sun.reflect;
    exports com.sun.nio.file;

    opens sun.misc;
    opens sun.reflect;
}
```

Несмотря на эту временную (и довольно неохотную) поддержку от Oracle, во многих каркасах и библиотеках имеются проблемы с переходом на Java 9, и пока что не было сделано никакого объявления о том, когда будет отозвана временная поддержка критических внутренних API.

Следует сказать, что был достигнут определенный прогресс в создании альтернатив этим API. Например, функциональность `getCallerClass()` доступна в API-интерфейсе стека, определенном в JEP 259[6]. Существует также еще один очень важ-

[5] `http://openjdk.java.net/jeps/260`

[6] `http://openjdk.java.net/jeps/259`

ный новый API, целью которого является замена функциональности Unsafe; мы рассмотрим его ниже.

VarHandles в Java 9

Мы уже встречались с дескрипторами методов в главе 11, "Языковые методы повышения производительности", и с Unsafe в главе 12, "Методы повышения производительности параллельной работы".

Дескрипторы методов предоставляют способ работы с непосредственно выполнимыми ссылками на методы, но исходная функциональность не является стопроцентной, потому что для полей предоставляется доступ только к методам получения и установки значения. Этого недостаточно, поскольку платформа Java предоставляет режимы доступа для данных, выходящих за рамки этих простейших вариантов использования.

В Java 9 дескрипторы методов расширены и включают *дескрипторы переменных* (variable handles), определенные в JEP 193[7]. Одна из целей этого предложения состояла в том, чтобы заполнить имеющиеся пробелы и при этом обеспечить безопасную замену некоторых API из Unsafe. Конкретные замены включают функциональность CAS и доступ к volatile полям и массивам. Еще одна цель — разрешить низкоуровневый доступ к режимам упорядочения обращений к памяти, доступным в JDK 9 в качестве части обновлений JMM.

Давайте взглянем на небольшой пример, который показывает подход, которым можно воспользоваться для замены Unsafe:

```
public class AtomicIntegerWithVarHandles extends Number
{
    private volatile int value = 0;
    private static final VarHandle V;
    static
    {
        try
        {
            MethodHandles.Lookup l = MethodHandles.lookup();
            V = l.findVarHandle(AtomicIntegerWithVarHandles.class, "value",
                                int.class);
        }
        catch (ReflectiveOperationException e)
        {
            throw new Error(e);
        }
    }
    public final int getAndSet(int newValue)
    {
```

[7] http://openjdk.java.net/jeps/193

```
        int v;

        do
        {
            v = (int)V.getVolatile(this);
        }
        while (!V.compareAndSet(this, v, newValue));

        return v;
    }
    // ....
```

- Этот код, по сути, эквивалентен примеру с атомарным целочисленным значением, который мы видели в разделе "Построение параллельных библиотек" главы 12, "Методы повышения производительности параллельной работы", и который демонстрирует, как `VarHandle` может заменить использование методов `Unsafe`.
- На момент написания книги класс `AtomicInteger` фактически не был изменен для использования механизма `VarHandle` (из-за циклических зависимостей) и по-прежнему был основан на использовании `Unsafe`. Тем не менее Oracle настоятельно рекомендует как можно скорее перенести все библиотеки и каркасы на новые поддерживаемые механизмы.

Проект Valhalla и типы значений

Миссия проекта Valhalla[8] формулируется как "место для изучения и выращивания кандидатов для перспективных виртуальных машин Java и для новых функциональных возможностей языка программирования". Вот как выглядят основные цели проекта.

- Выравнивание поведения схемы памяти JVM с моделью стоимости современного аппаратного обеспечения.
- Расширение обобщенных типов таким образом, чтобы были допустимы абстракции над всеми типами, включая примитивные типы, значения и даже `void`.
- Обеспечение возможности совместного развития существующих библиотек, особенно JDK, таким образом, чтобы полностью использовать преимущества этих функциональных возможностей.

В этом описании скрыто упоминание об одном из самых больших усилий в рамках проекта: об изучении возможности *типов значений* (value types) в JVM.

[8] `https://wiki.openjdk.java.net/display/valhalla/Main`

Напомним, что до версии 9 включительно Java имел только два типа значений: примитивные типы и ссылки на объекты. Иначе говоря, среда Java намеренно не обеспечивает низкоуровневый контроль над схемой памяти. В частности, это означает, что Java не имеет такой вещи, как структуры, а к любому составному типу данных можно получить доступ только по ссылке.

Чтобы понять последствия этого, давайте посмотрим на схему памяти массивов. На рис. 15.1 мы видим массив примитивных `int`. Поскольку эти значения не являются объектами, они располагаются в соседних ячейках памяти.

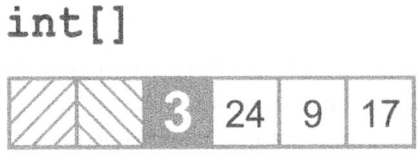

Рис. 15.1. Массив значений типа `int`

Упакованное же целое число, напротив, представляет собой объект и поэтому передается по ссылке. Это означает, что массив объектов `Integer` будет представлять собой массив ссылок (рис. 15.2).

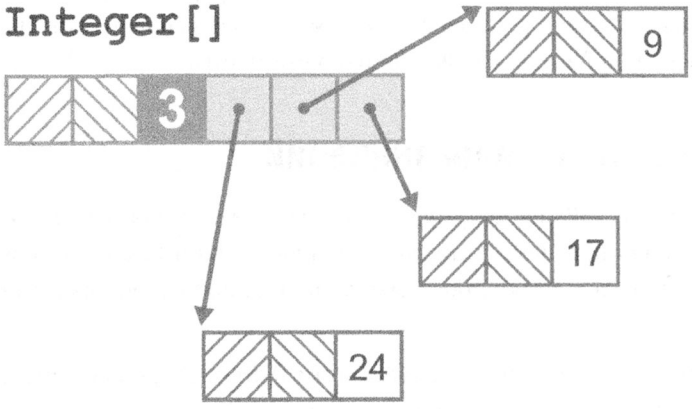

Рис. 15.2. Массив объектов `Integer`

На протяжении более 20 лет платформа Java использовала эту схему памяти. Она имеет преимущество простоты, но представляет собой компромиссное решение в смысле производительности: массивы объектов подразумевают неизбежные косвенные обращения и промахи кеширования.

В результате многие ориентированные на производительность программисты хотели бы иметь возможность определять типы, которые могут быть более эффективно размещены в памяти. Это также включает удаление накладных расходов на полные заголовки объектов для каждого элемента составных данных.

Например, точка в трехмерном пространстве `Point3D` действительно содержит только три пространственные координаты. По состоянию на Java 9 такой тип может быть представлен только как тип объекта с тремя полями:

```
lic final class Point3D
{
    private final double x;
    private final double y;
    private final double z;

    public Point3D(double a, double b, double c)
    {
        x = a;
        y = b;
        c = z;
    }
    // Методы доступа и иные методы опущены
}
```

Таким образом, массив точек располагается в памяти так, как показано на рис. 15.3.

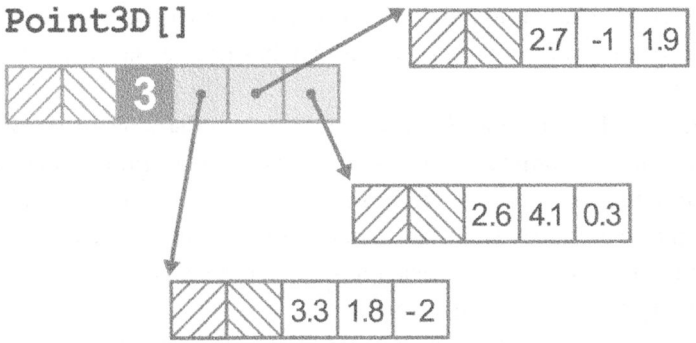

Рис. 15.3. Массив объектов `Point3D`

При обработке массива для получения координат каждой точки доступ к каждой записи выполняется через дополнительную косвенность. Потенциально это может привести к промахам кеша для каждой точки массива, что приводит к отсутствию реальной выгоды от него.

Кроме того, идентичность объекта для типа `Point3D` не имеет смысла. Объекты равны тогда и только тогда, когда все их поля равны. В общем случае это то, что подразумевается под *типом значения* (value type) в экосистеме Java.

Если эта концепция может быть реализована в JVM, то для простых типов (таких, как пространственные точки) показанная на рис. 15.4 схема памяти (фактически массив структур) будет намного более эффективной.

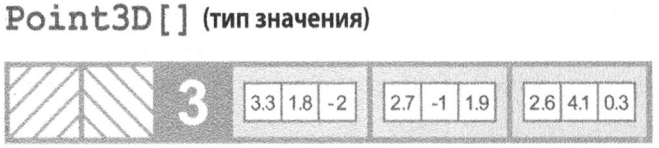

Рис. 15.4. Массив "структуроподобных" `Point3D`

Это не все — появляются и другие возможности (такие, как пользовательские типы, которые ведут себя аналогично встроенным примитивным типам).

Однако в этой области есть некоторые ключевые концептуальные трудности. Одна из важных проблем связана с проектными решениями, выполненными как часть добавления обобщенных типов в Java 5. Это тот факт, что система типов Java не имеет *верхнего типа* (top type), поэтому нет типа, который был бы супертипом как для `Object`, так и для `int`. Можно также сказать, что система типов Java не имеет *единственного корня*.

Вследствие этого обобщенные типы Java создаются только поверх ссылочных типов (являются подтипами `Object`), и нет очевидного способа получить согласованный смысл (по факту — общий тип), например, для `List<int>`. Java использует *затирание типов* (type erasure) для реализации обратно совместимых обобщенных типов поверх ссылочных типов, но этот отвратительный механизм не несет ответственности за отсутствие верхнего типа и, как следствие, за отсутствие примитивных коллекций.

Если платформа Java должна быть расширена, чтобы включить типы значений, естественно возникает вопрос о том, могут ли типы значений использоваться в качестве значений параметров типов. Если нет, то это, по-видимому, будет сильно ограничивать их полезность. Поэтому дизайн типов значений всегда включал в себя предположение, что они будут корректными в качестве значений параметров типов в расширенной форме обобщенных типов.

На момент написания книги это привело к дизайну[9], в котором должны иметься три разновидности типов классов и интерфейсов JVM.

- Ссылочные типы (`R`), которые представляют ссылки на экземпляры классов, имеющих идентичность, или представляют собой `null`.
- Типы значений (`Q`), которые являются экземплярами *классов значений*, у которых отсутствует идентичность.
- Универсальные типы (`U`), которые могут быть либо `R`, либо `Q`.

Это приводит к вопросу "Как следует понимать информацию о типах в существующих файлах классов?" То есть являются ли существующие типы `L` (соответствующие текущим типам в файлах классов Java 9) в действительности R-типами, или на

[9] `http://cr.openjdk.java.net/~dlsmith/values-notes.html`

самом деле они являются U-типами, и это просто так получилось, что мы никогда не видели тип Q ранее?

По соображениям совместимости и чтобы позволить расширять определение обобщенных типов, включая типы Q, под типами L понимаются типы U, а не типы R.

Это довольно незрелый прототип, и до сих пор остается много проблем проектирования, например вопрос о том, требуют ли типы значений на уровне виртуальной машины значений переменной ширины.

В Java 9 на уровне виртуальной машины все типы имеют значения фиксированной ширины (фиксированного размера). Примитивные типы имеют ширину 1, 2, 4 или 8 байт, а ссылки на объекты являются указателями (поэтому их ширина равна одному машинному слову). На современном оборудовании это означает, что ссылки являются 32- или 64-разрядными в зависимости от аппаратной архитектуры машины.

Будет ли добавление типов значений означать, что байт-код должен работать с типами переменной ширины? Есть ли место в байт-коде для необходимых для этого команд? В настоящее время считается, что необходимо добавить только два новых кода:

- `vdefault`, который генерирует экземпляр по умолчанию класса значения с Q-типом;
- `withfield`, который производит новое значение входного типа и генерирует исключение для входных данных, не являющихся значением или `null`.

Некоторые байт-коды также потребуют определенной модернизации для обработки новых Q-типов. Чтобы обеспечить соответствующее развитие основных библиотек, требуется также обширная работа на уровне выше уровня VM.

> *Valhalla может быть продиктована соображениями производительности, но лучше всего рассматривать этот проект как расширение абстракции, инкапсуляции, безопасности, выразительности и поддерживаемости — без отказа от производительности.*
>
> *— Брайан Гётц (Brian Goetz)*

Из-за изменений в расписании выпусков неясно, в каком именно выпуске Java в конечном итоге в качестве производственной функции будут введены типы значений. Наилучшим предположением авторов является то, что они войдут в один из выпусков 2019 года, но Oracle это не подтверждает.

Graal и Truffle

Компилятор C2, входящий в HotSpot, был чрезвычайно успешным. Тем не менее в последние годы его популярность несколько снизилась, а за последние несколько лет в компилятор не было внесено никаких существенных улучшений. С точки зрения

стоявших перед ним целей и задач C2 завершил свой жизненный цикл и теперь должен быть заменен.

Текущее направление исследований, призванное привести к появлению новых продуктов этого класса, основано на *Graal* и *Truffle*. Первый из них представляет собой специализированный JIT-компилятор, а последний является генератором интерпретаторов для возможности работы различных языков со средой выполнения JVM.

Один из возможных путей улучшения JIT-компилятора заключается в том, что компилятор C2 написан на C++, и это привносит некоторые потенциально серьезные проблемы. C++ — небезопасный язык, который использует ручное управление памятью, поэтому ошибки в коде C2 могут привести к сбою виртуальной машины. Но это не все: код C2 неоднократно модифицировался, так что его стало очень сложно поддерживать и расширять.

Чтобы добиться прогресса, Graal пытается использовать другой подход; это JIT-компилятор для JVM, написанный на Java. Интерфейс, который JVM использует для сообщения с Graal, называется JVM Compiler Interface (JVMCI); он был добавлен в платформу в качестве JEP 243. Он позволяет подключать Java-интерфейс как JIT-компилятор, аналогично тому, как Java-агенты могут быть подключены к JVM.

Представление проекта заключается в том, что JIT-компилятор в действительности должен быть способен просто принимать байт-код JVM и создавать на его основе машинный код. На низком уровне компилятор просто преобразует `byte[]` (байт-код) в другой `byte[]` (машинный код), поэтому нет причин, по которым это невозможно реализовать на Java.

Такой подход Java-в-Java имеет ряд преимуществ, в том числе простоту, безопасность с точки зрения памяти и возможность использования стандартного инструментария Java, такого как интегрированная среда разработки и отладчики, вместо того чтобы требовать от разработчиков компилятора навыков в эзотерическом диалекте C++.

Эти преимущества позволяют использовать мощные новые оптимизации, такие как частичный анализ локальности (см. раздел "Ограничения анализа локальности" главы 10, "JIT-компиляция"), которые будут реализованы в Graal, а не C2. Еще одно преимущество заключается в том, что Graal позволяет командам изменять его части для их собственных приложений, например разрабатывать собственные встроенные функции (например, для пользовательского оборудования) или проходы оптимизации.

Truffle представляет собой каркас для разработки интерпретаторов для языков в JVM. Он предназначен для совместной работы с Graal в качестве библиотеки для автоматической генерации JIT-компилятора с высокой производительностью для входного языка на основе интерпретатора с использованием академической методики под названием *Futamuru projection*. Это методика из области академической информатики, известная как *частичная специализация* (partial specialization), которая

в последнее время стала более практичной для использования в реальных системах (хотя некоторые ее идеи были использованы несколько лет назад в реализации PyPy Python).

Truffle является альтернативой подходу, заключающемуся в генерации байт-кода во время выполнения (который используется существующими реализациями языков для JVM, такими как JRuby, Jython и Nashorn). Измерения производительности пока показывают, что комбинация Truffle и Graal может обеспечить гораздо более высокую производительность, чем наблюдавшаяся ранее.

Зонтичным (обобщающим) проектом всей этой работы является новый проект — *Project Metropolis*. Это попытка переписать большую часть VM на Java, начиная с JIT-компиляторов HotSpot и, возможно, интерпретатора.

Технология Metropolis/Graal представлена и поставляется в выпуске Java 9, хотя она все еще остается во многом экспериментальной. Для включения нового JIT-компилятора используется следующий переключатель командной строки:

```
-XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler
```

Есть еще один способ использования Graal в Java 9: режим предварительной компиляции (Ahead-of-Time). При этом исходный текст Java компилируется непосредственно в машинный код, аналогично тому, как это происходит в C и C++. Java 9 включает команду `jatoc`, которая использует Graal и имеет единственную цель — ускорить время запуска, пока не начнется нормальная многоуровневая компиляция. В настоящее время этот инструмент поддерживает только модуль `java.base` на единственной платформе (Linux/ELF), но ожидается, что поддержка будет расширяться в следующих нескольких версиях Java.

Чтобы создать бинарные файлы из файлов классов Java, новый инструмент `jaotc` используется следующим образом:

```
jaotc --output libHelloWorld.so HelloWorld.class
jaotc --output libjava.base.so --module java.base
```

Наконец, упомянем *SubstrateVM* — исследовательский проект (также использующий Graal) для дальнейшего применения этой функциональности и для компиляции всей JVM, написанной на Java, вместе с Java-приложением с целью создания единого статически скомпонованного с ней исполняемого файла. Идея заключается в том, что это приведет к созданию машинных бинарных файлов, которые не нуждаются в JVM ни в какой форме, могут иметь размер всего несколько килобайтов и запускаться за несколько миллисекунд.

Будущее развитие байт-кода

Одним из самых больших изменений в VM было появление `invokedynamic`. Этот новый байт-код открыл дверь для переосмысления того, как можно писать байт-код

JVM. В настоящее время предпринимается попытка расширить технологию, используемую этим кодом операции, чтобы обеспечить еще большую гибкость платформы.

Например, вспомните различие между кодами `ldc` и `const` в разделе "Обзор интерпретации байт-кода" главы 9, "Выполнение кода в JMV". Оно является куда бо́льшим, чем кажется на первый взгляд. Давайте рассмотрим простой фрагмент кода:

```
public static final String HELLO = "Hello World";
public static final double PI = 3.142;

public void showConstsAndLdc()
{
    Object o = null;
    int i = -1;
    i = 0;
    i = 1;
    o = HELLO;
    double d = 0.0;
    d = PI;
}
```

Он приводит к генерации следующей довольно простой последовательности байт-кодов:

```
public void showConstsAndLdc();
  Code:
    0: aconst_null
    1: astore_1
    2: iconst_m1
    3: istore_2
    4: iconst_0
    5: istore_2
    6: iconst_1
    7: istore_2
    8: ldc          #3 // String Hello World
    10: astore_1
    11: dconst_0
    12: dstore_3
    13: ldc2_w      #4 // double 3.142d
    16: dstore_3
    17: return
```

Имеются также несколько дополнительных записей в пуле констант:

```
#3 = String         #29           // Hello World
#4 = Double         3.142d
...
#29 = Utf8          Hello World
```

Основная схема понятна: "истинные константы" отображаются как команды const, в то время как загрузки из пула констант представлены командами ldc.

Первые представляют собой небольшой конечный набор констант, таких как примитивы 0, 1 и null. Напротив, любое неизменяемое значение можно рассматривать как константу для ldc, а последние версии Java значительно увеличили количество разных константных типов, которые могут находиться в пуле констант.

Например, рассмотрим следующий фрагмент кода Java 7 (или показанный выше), который использует API дескрипторов методов, с которыми мы встречались в разделе "Дескрипторы методов" главы 11, "Языковые методы повышения производительности":

```
public MethodHandle getToStringMH() throws NoSuchMethodException,
       IllegalAccessException
{
    MethodType mt = MethodType.methodType(String.class);
    MethodHandles.Lookup lk = MethodHandles.lookup();
    MethodHandle mh = lk.findVirtual(getClass(), "toString", mt);
    return mh;
}

public void callMH()
{
    try
    {
        MethodHandle mh = getToStringMH();
        Object o = mh.invoke(this, null);
        System.out.println(o);
    }
    catch (Throwable e)
    {
        e.printStackTrace();
    }
}
```

Чтобы увидеть влияние дескрипторов методов на пул констант, добавим простой метод к нашему тривиальному примеру с ldc и const:

```
public void mh() throws Exception
{
    MethodType mt = MethodType.methodType(void.class);
    MethodHandle mh = MethodHandles.lookup().findVirtual
            (BytecodePatterns.class, "mh", mt);
}
```

Это даст нам следующий байт-код:

```
public void mh() throws java.lang.Exception;
  Code:
```

```
    0: getstatic      #6 // Поле java/lang/Void.TYPE:Ljava/lang/Class;
    3: invokestatic   #7 // Метод java/lang/invoke/MethodType.methodType:
                         // (Ljava/lang/Class;)Ljava/lang/invoke/MethodType;
    6: astore_1
    7: invokestatic   #8 // Метод java/lang/invoke/MethodHandles.lookup:
                         // ()Ljava/lang/invoke/MethodHandles$Lookup;
    10: ldc           #2 // Класс optjava/bc/BytecodePatterns
    12: ldc           #9 // String mh
    14: aload_1
    15: invokevirtual #10 // Метод java/lang/invoke/MethodHandles$Lookup.
                          // findVirtual:(Ljava/lang/Class;Ljava/lang/
                          // String;Ljava/lang/invoke/MethodType;)Ljava/
                          // lang/invoke/MethodHandle;
    18: astore_2
    19: return
}
```

Здесь содержатся дополнительный `ldc` для литерала `BytecodePatterns.class` и обходной путь для объекта `void.class`, (который должен быть гостем в типе `java.lang.Void`). Однако константы класса лишь немного более интересны, чем константы-строки или примитивы.

Это не вся история, и влияние на пул констант применения дескрипторов методов оказывается весьма значительным. Первое место, где мы можем видеть это влияние, — появление некоторых новых типов записей пула:

```
#58 = MethodHandle  #6:#84 // invokestatic java/lang/invoke/LambdaMetafactory.
                           // metafactory:(Ljava/lang/invoke/MethodHandles
                           // $Lookup;Ljava/lang/String;Ljava/lang/invoke/
                           // MethodType;Ljava/lang/invoke/MethodType;Ljava/
                           // lang/invoke/MethodHandle;Ljava/lang/invoke/
                           // MethodType;)Ljava/lang/invoke/CallSite;
#59 = MethodType    #22    // ()V
#60 = MethodHandle  #6:#85 // invokestatic optjava/bc/BytecodePatterns.
                           // lambda$lambda$0:()V
```

Эти новые типы констант необходимы для поддержки `invokedynamic`, и для развития платформы начиная с Java 7 все больше и больше используется эта технология. Общая цель состоит в том, чтобы сделать вызов метода посредством `invokedynamic` таким же производительным и готовым к JIT-компиляции, как и обычные вызовы `invokevirtual`.

Другие направления будущей работы включают возможность "константной динамики" — аналога `invokedynamic`, но для записей пула констант, которые неразрешимы во время компоновки и вычисляются при первом обращении.

Ожидается, что эта область JVM будет оставаться очень активной темой исследований в будущих версиях Java.

Будущие направления в области параллельности

Как говорилось в главе 2, "Обзор JVM", одним из основных нововведений Java было внедрение автоматического управления памятью. В наши дни практически ни один разработчик не сочтет ручное управление памятью положительной особенностью, которую должен использовать любой новый язык программирования.

Мы можем видеть частичное отражение этого в эволюции подхода Java к параллельным вычислениям. Исходный дизайн потоковой модели Java предполагал, что все потоки должны быть явно управляемыми программистом, а изменяемое состояние должно быть защищено блокировками в этом, по сути, кооперативном дизайне. Если одна часть кода неправильно реализует схему блокировки, она может повредить состояние объекта.

Основополагающий принцип потоковой модели Java можно выразить следующим образом: "Несинхронизированный код не смотрит на состояние блокировок объектов, не заботится о них и может получить доступ к состоянию объекта или повредить его по своему желанию".

По мере развития Java новые версии постепенно переходили от этого дизайна к подходу более высокого уровня с меньшим ручным управлением и в целом с большим уровнем безопасности — по сути, к *управляемому во время выполнения параллелизму* (runtime-managed concurrency).

Одним из аспектов этого развития является недавно объявленный *Project Loom*. Этот проект касается поддержки параллелизма JVM на более низком уровне, чем это делалось в JVM до сих пор. Существенной проблемой потоков Java является то, что каждый поток имеет стек. Стеки дороги и не бесконечно масштабируемы; как только у нас будет, скажем, 10 000 потоков, необходимая для них память составит гигабайты.

Одним из решений являются шаг назад и рассмотрение иного подхода: единицы выполнения, которые не могут быть запланированы непосредственно операционной системой, имеют низкие накладные расходы и могут быть "в основном бездействующими" (т.е. не расходующими активно процессорное время).

Это хорошо сочетается с подходом, используемым другими языками (как основанными на JVM, так и прочими). Зачастую они имеют более низкоуровневые кооперативные конструкции, такие как горутины (goroutines – параллельно выполняемые методы), волокна (та же концепция, что и корутины – облегченные потоки) и продолжения (состояние программы в определенный момент, которое может быть сохранено и использовано для перехода в это состояние). Эти абстракции должны быть кооперативными, а не вытесняющими, поскольку они работают ниже видимости операционной системы и сами по себе не являются планируемыми субъектами.

В соответствии с этим подходом требуются два основных компонента: представление вызываемого кода (например, `Runnable` или аналогичный тип) и компонент

планировщика. Как ни странно, начиная с версии 7 у JVM был хороший компонент планирования для этих абстракций, несмотря на отставание других частей.

Появившийся в Java 7 Fork/Join API (описанный в разделе "Fork/Join" главы 12, "Методы повышения производительности параллельной работы") был основан на двух концепциях — идее рекурсивной декомпозиции выполнимых заданий и идее захвата работы, когда простаивающие потоки могут выполнять работу из очередей занятых потоков. В основе этих двух концепций лежит исполнитель `ForkJoinPool`, отвечающий за реализацию алгоритма захвата работы.

Оказывается, что рекурсивная декомпозиция для большинства задач полезна не всегда. Однако пул потоков исполнителей с захватом работы может применяться во многих различных ситуациях. Например, каркас Akka принял `ForkJoinPool` в качестве своего исполнителя.

Пока что Project Loom только начинает свой путь, но вполне возможно, что исполнитель `ForkJoinPool` будет использоваться и в качестве компонента планирования для легких объектов выполнения. И вновь, стандартизация этой возможности в виртуальных машинах и основных библиотеках значительно снизит необходимость использования внешних библиотек.

Заключение

С момента своего первоначального релиза Java претерпел огромные изменения. Из языка, который не разрабатывался как высокопроизводительный, он стал одним из таковых языков программирования. Ядро Java, сообщество и экосистема оставались здоровыми и яркими, даже когда применимость Java расширилась до новых областей.

Новые смелые инициативы, такие как Project Metropolis и Graal, переосмысливают основы виртуальных машин. Код `invokedynamic` позволил HotSpot выйти из своей эволюционной ниши и превзойти самого себя в течение десятилетия. Java показал, что не боится амбициозных изменений, таких как добавление типов значений и возврат к решению сложных проблем обобщенных типов.

Производительность Java/JVM — очень динамичная область, и в этой главе мы увидели, что, кроме нее, развиваются еще многие области. Существует множество других проектов, о которых мы не успели упомянуть, включая взаимодействие Java с машинным кодом (Project Panama) и новые сборщики мусора, такие как Oracle ZGC.

В результате эта книга ни в одной затронутой теме и близко не подошла к завершенности. Тем не менее мы надеемся, что это было полезное введение в мир производительности Java, которое предоставило некоторые дорожные указатели читателям для их собственного путешествия по миру производительности.

Предметный указатель

O'REILLY®

Java: оптимизация программ

Настройка производительности — наука экспериментальная, но это не означает, что инженеры должны прибегать к догадкам и фольклору, чтобы выполнить свою работу (хотя часто случается именно так). С помощью этой практической книги разработчики средней и высокой квалификации, работающие со сложными стеками технологий, научатся настраивать высокую производительность Java-приложений, используя количественный, поддающийся проверке подход.

В большинстве информационных ресурсов о производительности, как правило, обсуждаются теория и внутреннее устройство виртуальных машин Java, но в этой книге основное внимание уделяется практическим возможностям настройки производительности путем изучения широкого ряда аспектов. В книге нет простых рецептов, советов и трюков или алгоритмов. Настройка производительности — это процесс внесения изменений и измерения полученных результатов, требующий усердия.

- Узнайте, как принципы и технологии Java наилучшим образом используют современные аппаратные средства и операционные системы
- Исследуйте различные тесты производительности и распространенные антипаттерны, которые могут завести вашу команду в тупик
- Изучите ловушки измерений показателей производительности Java и недостатки микротестирования
- Погрузитесь в сборку мусора, протоколирование, мониторинг, настройки и инструменты JVM
- Исследуйте JIT-компиляцию и методы повышения производительности в языке Java
- Изучите аспекты производительности API-коллекций и вопросы параллельных вычислений в Java

Бенджамин Эванс — соучредитель и член команды технологов jClarity, стартапа по производству инструментария для работы в области оценки производительности, призванного помочь командам разработчиков на Java.

Джеймс Гоф — разработчик на Java и автор книг. Работает в Morgan Stanley над созданием бизнес-приложений.

Крис Ньюланд — старший разработчик и руководитель команды в ADVFN, где он использует Java для обработки данных фондового рынка в режиме реального времени. Также является изобретателем JITWatch.

"За последние 20 лет я потратила немало времени на понимание деталей того, что происходит внутри JVM, но, читая эту книгу, я научилась кое-чему новому. Она хорошо написана, легко читается и содержит массу полезной информации как для начинающего программиста, так и для профессионала. Независимо от того, на чем вы работаете, на выделенном сервере с двумястами ядрами или на небольшой машине с ограниченными ресурсами под управлением Linux, эта книга поможет вам получить максимальную отдачу от вашего Java-приложения".

Кристин Флуд
(Christine H. Flood)
Red Hat, Inc.

ISBN 978-5-907114-84-5

9 785907 114845 19003

ДИАЛЕКТИКА

www.williamspublishing.com