Содержание

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).

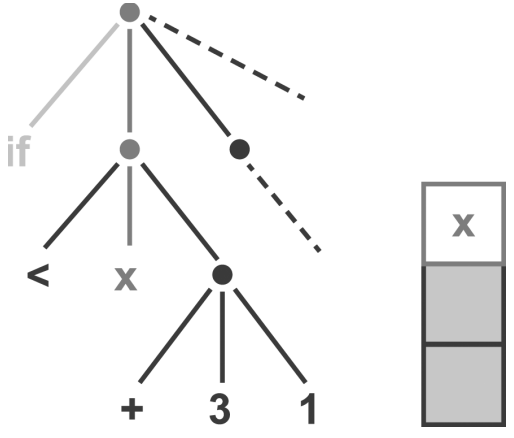

Рис. 9.1. Начальное состояние интерпретатора

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.

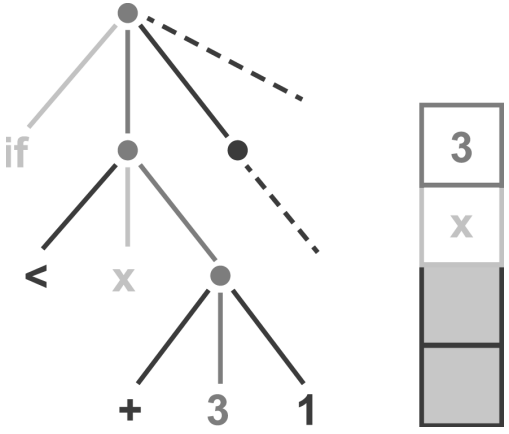

Рис. 9.2. Вычисление поддерева

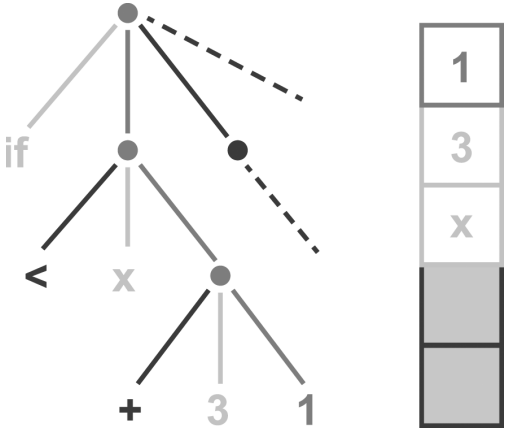

Рис. 9.3. Вычисление поддерева

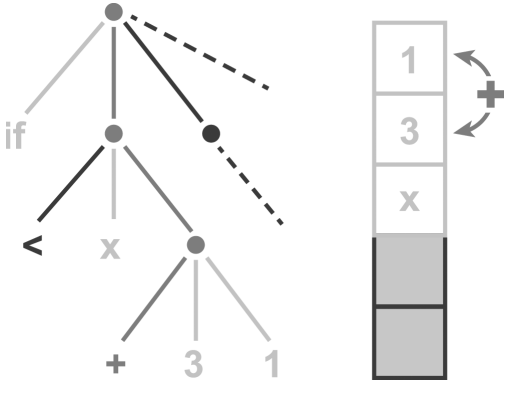

Рис. 9.4. Вычисление поддерева

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.

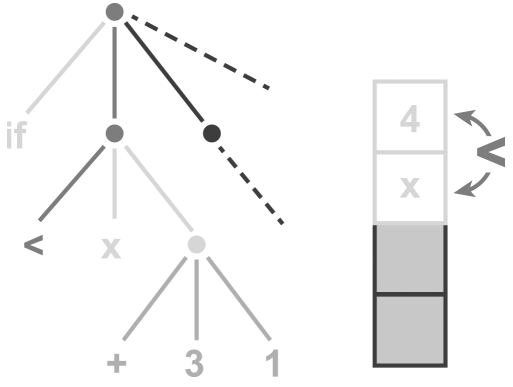

Рис. 9.5. Вычисление последнего поддерева

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`	`(i1)`	Загружает значение из локальной переменной `i1` в стек
`store`	`(i1)`	Сохраняет значение на вершине стека в локальную переменную `i1`
`ldc`	`c1`	Загружает значение из `CP#c1` в стек
`const`		Загружает простое константное значение в стек
`pop`		Снимает значение с вершины стека
`dup`		Дублирует значение на вершине стека
`getfield`	`c1`	Загружает в стек значение из поля, указанного `CP#c1`, из объекта на вершине стека
`putfield`	`c1`	Сохраняет значение на вершине стека в поле, указанное `CP#c1`
`getstatic`	`c1`	Загружает в стек значение из статического поля, указанного `CP#c1`
`putstatic`	`c1`	Сохраняет значение на вершине стека в статическое поле, указанное `CP#c1`

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`	`(i1)`	Ветвление к точке, указанной в качестве аргумента, если условие имеет значение `true`
`goto`	`i1`	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

Окончание табл. 9.4

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`	`c1`	Выделяет память для объекта типа, найденного в `CP#c1`
`newarray`	`prim`	Выделяет память для примитивного массива типа `prim`
`anewarray`	`c1`	Выделяет память для массива объектов типа, найденного в `CP#c1`
`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.

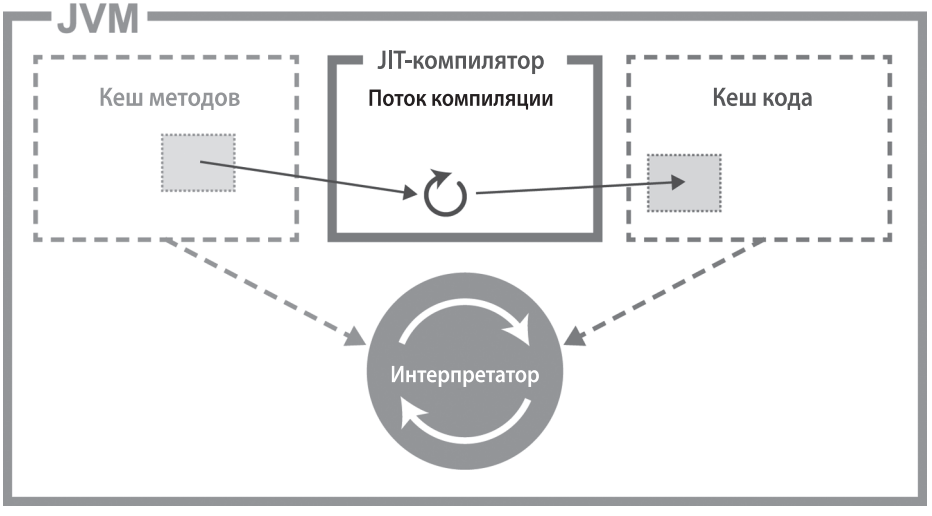

Рис. 9.6. Простая компиляция одного метода

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.