12

Модули на платформе Java

Начиная с версии 9 платформа Java была наконец-то оснащена долгожданной модульной системой. Это средство первоначально предполагалось включить в состав версии Java 7, выпущенной еще компанией Sun Microsystems, где язык Java разрабатывался до ее приобретения компанией Oracle. Но задача внедрения модульной системы оказалась намного более сложной и трудной, чем предполагалось.

Когда компания Oracle приобрела права на Java (как часть технологии, переданной компанией Sun Microsystems), Марк Рейнольд — главный архитектор языка Java предложил "план Б" (`https://mreinhold.org/blog/rethinking-jdk7`), согласно которому рамки версии Java 7 были сокращены с целью ускорить ее выпуск.

Таким образом, внедрение модулей на платформе Java (в рамках проекта "Project Jigsaw") было отложено наряду с лямбда-выражениями до версии Java 8. Но в процессе разработки версии Java 8 масштабы и сложности данного средства привели к решению (`https://mreinhold.org/blog/late-for-the-train`), что внедрение модулей лучше отложить до версии Java 9, чем задерживать выпуск версии Java 8 (и доступность лямбда-выражений и прочих долгожданных языковых средств).

В итоге возможность внедрить модули была отложена сначала до версии Java 8, а затем и до версии Java 9. Но и тогда объем работ оказался настолько велик, что привел к существенной задержке выпуска версии Java 9, и поэтому модули фактически появились лишь в сентябре 2017 года.

Эта глава служит кратким введением в *модульную систему на платформе Java* (Java Platform Modules System — JPMS). Но данная тема сложна и обширна, поэтому отсылаем интересующихся читателей к более подробно

изложенному материалу в книге *Java 9 Modularity* Зандера Мака (Sander Mak) и Пола Беккера (Paul Bakker, издательство O'Reilly, 2017 г.).

Модули являются относительно продвинутым языковым средством, имеющим, главным образом, отношение к упаковке и развертыванию целых приложений и их зависимостей. Начинающим программировать на Java совсем не обязательно полностью овладевать этим языковым средством, пока они еще учатся писать простые программы на Java.

Принимая во внимание продвинутый характер модулей, в этой главе предполагается, что читатели знакомы с такими современными инструментальными средствами сборки программ на Java, как Gradle или Maven. А начинающие изучать Java могут благополучно пренебречь ссылками на эти инструментальные средства и просто прочитать эту главу, чтобы получить первое общее представление о модульной системе JPMS.

Зачем нужны модули

Для внедрения модулей на платформе Java имелось несколько побудительных причин. К их числу относятся следующие.

- Строгая инкапсуляция.
- Вполне определенные интерфейсы.
- Явные зависимости.

Все эти причины относятся к уровню языка (и разработки приложений), и поэтому они были объединены со следующими обещаниями внедрить новые функциональные возможности на уровне платформы.

- Масштабируемая разработка.
- Повышенная производительность (особенно на стадии запуска проекта) и сокращенный объем занимаемой памяти.
- Сокращение видов атак и повышение безопасности.
- Развитие внутренних компонентов.

С точки зрения инкапсуляции потребность в модулях объяснялась тем, что в первоначальной спецификации языка Java поддерживались только закрытый, открытый, защищенный и открытый в пределах пакета уровни

доступности. В ней не предусматривался более тщательный контроль доступа для выражения следующих возможностей.

- В виде прикладного интерфейса API могут быть доступны только указанные пакеты, а другие пакеты являются внутренними и могут оказаться недоступными.
- Некоторые пакеты могут быть доступны из одного списка пакетов, но не из других пакетов.
- Определение строгого механизма экспорта.

Нехватка этих и связанных с ними возможностей была значительным ограничением для разработки архитектуры крупных систем на Java. А кроме того, без подходящего механизма защиты было бы очень трудно развивать внутренние компоненты JDK, поскольку ничто не мешало получить непосредственный доступ к классам реализации из пользовательских приложений. В модульной системе предпринимается попытка сразу же разрешить все эти затруднения и предоставить решение, пригодное как для комплекта JDK, так и для пользовательских приложений.

Модуляризация комплекта JDK

Монолитный комплект JDK, вошедший в состав версии Java 8, стал первой целью для модульной системы, в результате чего известный архив `rt.jar` был разбит на отдельные модули. Это было сделано на основе так называемых *компактных профилей* еще во время работы над версией Java 8, т.е. до того, как внедрение модулей было отложено до выпуска версии Java 9.

Модуль `java.base` содержит минимум функциональных средств, фактически требующихся для запуска прикладной программы на Java. В его состав входят базовые пакеты, включая перечисленные ниже, а также некоторые подпакеты и неэкспортируемые пакеты реализации вроде `sun.text.resources`.

- `java.io`
- `java.lang`
- `java.math`
- `java.net`
- `java.nio`
- `java.security`
- `java.text`
- `java.time`
- `java.util`
- `javax.crypto`
- `javax.net`
- `javax.security`

Некоторые отличия в режиме компиляции между версией Java 8 и модульной версией Java можно проследить на примере следующей простой

программы, в которой расширяется внутренний открытый класс из модуля java.base:

```
import java.util.Arrays;
import sun.text.resources.FormatData;

public final class FormatStealer extends FormatData {
  public static void main(String[] args) {
    FormatStealer fs = new FormatStealer();
    fs.run();
  }

  private void run() {
    String[] s =
            (String[]) handleGetObject("japanese.Eras");
    System.out.println(Arrays.toString(s));

    Object[][] contents = getContents();
    Object[] eraData = contents[14];
    Object[] eras = (Object[])eraData[1];
    System.out.println(Arrays.toString(eras));
  }
}
```

Если скомпилировать и выполнить эту программу в версии Java 8, на экран будет выведен следующий перечень эпох японской истории:

```
[, Meiji, Taisho, Showa, Heisei]
[, Meiji, Taisho, Showa, Heisei]
```

Но попытка скомпилировать исходный код этой программы в версии 11 приведет к следующему сообщению об ошибке:

```
$ javac javanut7/ch12/FormatStealer.java
javanut7/ch12/FormatStealer.java:4:
  error: package sun.text.resources is not visible[1]
import sun.text.resources.FormatData;
               ^

(package sun.text.resources is declared in module
  java.base, which does not export it to the
   unnamed module)[2]
javanut7/ch12/FormatStealer.java:14:
```

[1] Ошибка: пакет sun.text.resources недоступен

[2] (пакет sun.text.resources объявлен в модуле java.base, который не экспортирует его в безымянный модуль)

```
    error: cannot find symbol[3]
    String[] s =
            (String[]) handleGetObject("japanese.Eras");
                       ^
    symbol: method handleGetObject(String)
    location: class FormatStealer[4]
javanut7/ch12/FormatStealer.java:17:
    error: cannot find symbol
    Object[][] contents = getContents();
                          ^
    symbol: method getContents()
    location: class FormatStealer
3 errors[5]
```

В модульной версии Java даже открытые классы могут оказаться недоступными, если не экспортировать их явным образом из того модуля, в котором они определены. Компилятор можно вынудить временно пользоваться внутренним пакетом, по существу, утвердив снова прежние правила доступа. Для этого достаточно указать параметр `--add-exports` в командной строке, как показано ниже.

```
$ javac --add-exports
  java.base/sun.text.resources=ALL-UNNAMED \
  javanut7/ch12/FormatStealer.java
javanut7/ch12/FormatStealer.java:5:
    warning: FormatData is internal proprietary API and
             may be removed in a future release[6]
import sun.text.resources.FormatData;
                         ^
javanut7/ch12/FormatStealer.java:7:
    warning: FormatData is internal proprietary API and
             may be removed in a future release
public final class FormatStealer extends FormatData {
                                         ^
2 warnings[7]
```

---

[3] ошибка: не удается найти символ

[4] symbol: метод handleGetObject(String)
местоположение: класс FormatStealer

[5] 3 ошибки

[6] предупреждение: FormatData — оригинальный прикладной интерфейс API, который предназначен для внутреннего пользования и может быть удален в последующей версии

[7] 2 предупреждения

В данном случае необходимо указать, что экспорт поручается *безымянному модулю*, поскольку класс компилируется самостоятельно, а не как часть модуля. При этом компилятор выдает предупреждение о применении прикладного интерфейса API, который предназначен для внутреннего пользования и может больше не поддерживаться в последующей версии Java.

Любопытно, что если попытаться выполнить программу из рассматриваемого здесь примера в версии Java 11, то результат окажется несколько иным:

```
[, Meiji, Taisho, Showa, Heisei, NewEra]
[, Meiji, Taisho, Showa, Heisei, NewEra]
```

Дело в том, что с 1 мая 2019 года в японской истории произойдет смена эпохи Хэйсэй на новую эпоху. По традиции наименование новой эпохи заранее неизвестно, и поэтому оно заполняется названием `NewEra`, вместо которого в последующей версии Java будет подставлено официальное наименование новой эпохи. Для символа, представляющего наименование новой эпохи, в Юникоде зарезервирована кодовая точка `U+32FF`.

Несмотря на то что в модуле `java.base` предоставляется абсолютный минимум функциональных средств, требующихся для запуска прикладной программы на Java, во время компиляции доступная платформа должна как можно точнее соответствовать нашим ожиданиям на основании опыта работы с версией Java 8. Это означает, что на практике мы пользуемся намного бо́льшим рядом модулей, входящих в состав *обобщающего* модуля `java.se`. Граф зависимостей этого модуля приведен на рис. 12.1.

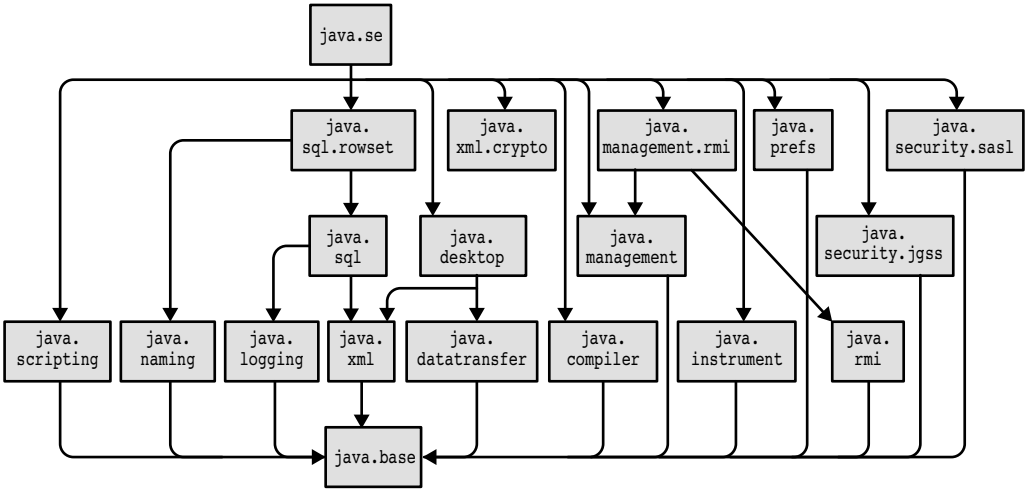

Рис. 12.1. Граф зависимостей модуля **`java.se`**

Таким образом, доступными оказываются почти все классы и пакеты, которыми предполагает пользоваться большинство разработчиков прикладных программ на Java. И хотя модули, в которых определяются прикладные интерфейсы CORBA и Java EE API, в модуле `java.se` не требуются, они все же требуются в модуле `java.se.ee`.

Это означает, что любой проект, зависящий от прикладных интерфейсов Java EE API (или CORBA), не будет по умолчанию скомпилирован, начиная с версии Java 9. Для этой цели потребуется специально сконфигурированная сборка.

Для компиляции подобных проектов потребуются прикладные интерфейсы API вроде JAXB, поэтому в сборку придется явным образом включить модуль `java.se.ee`. Помимо упомянутых выше изменений в доступности отдельных компонентов для компиляции вследствие модуляризации комплекта JDK, модульная система предназначена и для того, чтобы разработчики модуляризировали свой код.

Разработка собственных модулей

В этом разделе описываются основные понятия, которые требуется знать, чтобы приступить к написанию модульных прикладных программ на Java.

Основной синтаксис модулей

Ключевая роль в модуляризации принадлежит новому файлу `module-info.java`, содержащему описание модуля. Это так называемый *описатель модуля*. Для надлежащей компиляции модуль размещается в файловой системе следующим образом.

- В корневом каталоге исходного кода проекта (`src`) должен находиться каталог, который называется так же, как и модуль (`moduledir`).
- В каталоге `moduledir` должен находиться файл `module-info.java` на том же уровне, где и начинаются пакеты.

Сведения о модуле компилируются в двоичном формате и сохраняются в файле `module-info.class`, где содержатся метаданные, которые будут использоваться модульной исполняющей средой при попытках скомпоновать и выполнить прикладную программу. Ниже приведен простой пример содержимого файла `module-info.java`.

```
module kathik {
  requires java.net.http;

  exports kathik.main;
}
```

В данном примере применяются следующие элементы нового синтаксиса: `module`, `exports` и `requires`, но на самом деле они не являются полноценными ключевыми словами в принятом смысле. В спецификации на язык Java (Java Language Specification SE 9) об этом говорится следующее.

> Ограниченными ключевыми словами являются следующие десять последовательностей символов: `open`, `module`, `requires`, `transitive`, `exports`, `opens`, `to`, `uses`, `provides` и `with`. Эти последовательности символов преобразуются в лексемы по отдельности как ключевые слова там, где они появляются как оконечные в разработках на основе классов `ModuleDeclaration` и `ModuleDirective`.

Таким образом, эти ключевые слова могут появляться только в метаданных модуля и компилируются в двоичной форме компилятором `javac`. Ниже вкратце поясняется назначение основных ограниченных ключевых слов; остальные ключевые слова будут представлены далее в главе.

`module`
: Начинает объявление метаданных модуля.

`requires`
: Обозначает модуль, от которого зависит данный модуль.

`exports`
: Объявляет пакеты, экспортируемые как прикладной интерфейс API.

В рассматриваемом здесь примере эти ограниченные ключевые слова означают, что объявляется модуль `kathik`, зависящий прямо от модуля `java.net.http`, стандартизированного в версии Java 11, а косвенно — от модуля `java.base`. В этот модуль экспортируется единственный пакет `kathik.main`, и только он будет доступен из других модулей во время компиляции.

Построение простого модульного приложения

В качестве примера рассмотрим построение простого инструментального средства, проверяющего, пользуются ли уже веб-сайты сетевым протоколом

HTTP/2. Для этой цели воспользуемся прикладным интерфейсом API, упоминавшимся в главе 10:

```
import static java.net.http.HttpResponse
                  .BodyHandlers.ofString;

public final class HTTP2Checker {
  public static void main(String[] args)
         throws Exception {
    if (args.length == 0) {
      System.err.println("Provide URLS to check");
    }
    for (final var location : args) {
      var client = HttpClient.newBuilder().build();
      var uri = new URI(location);
      var req = HttpRequest.newBuilder(uri).build();
      var response = client.send(req,
      ofString(Charset.defaultCharset()));
      System.out.println(location +": "
                      + response.version());
    }
  }
}
```

Построение данного приложения основывается на двух модулях: `java.net.http` и общедоступном `java.base`. Файл описания модуля данного приложение очень прост:

```
module http2checker {
  requires java.net.http;
}
```

Принимая во внимание простое стандартное расположение модулей, данное приложение можно скомпилировать по следующей команде:

```
$ javac -d out/http2checker\
    src/http2checker/javanut7/ch12/HTTP2Checker.java\
    src/http2checker/module-info.java
```

В итоге будет создан скомпилированный модуль, размещаемый в каталоге `out/`. Чтобы воспользоваться им, необходимо упаковать его в архивный JAR-файл следующим образом:

```
$ jar -cfe httpchecker.jar javanut7.ch12.HTTP2Checker\
      -C out/http2checker/ .
```

Для установки *точки входа* в модуль в приведенной выше команде указан параметр `-e`. Это означает, что соответствующий класс будет выполнен, если модуль используется как приложение. Ниже показано, как это осуществляется на практике.

```
$ java -jar httpchecker.jar http://www.google.com
  http://www.google.com: HTTP_1_1
$ java -jar httpchecker.jar https://www.google.com
  https://www.google.com: HTTP_2
```

Таким образом, на момент написания данной книги веб-сайт компании Google обслуживал свою главную страницу по сетевому протоколу HTTPS, используя протокол HTTP/2, хотя для устаревших HTTP-служб он по-прежнему действовал по протоколу HTTP/1.1.

Итак, показав, каким образом компилируется и выполняется простое модульное приложение, перейдем к рассмотрению других базовых средств модуляризации, требующихся для построения и выполнения полномасштабных приложений.

Путь к модулям

Многим разработчикам прикладных программ на Java знакомо понятие пути к классам. Но для разработки модульных приложений на Java вместо него необходимо пользоваться понятием *пути к модулям*. Это новое понятие введено для модулей, чтобы заменить собой путь к классам везде, где только можно.

Модули содержат метаданные об экспорте их компонентов и зависимостях, хотя это и не очень длинный список типов данных. Это означает, что граф зависимостей модулей можно легко построить и эффективно продолжить разрешение модулей.

Код, который еще не модуляризирован, может быть и далее расположен по пути к классам. Этот код загружается в *безымянный модуль*, который имеет особое назначение и может получить доступ ко всем остальным модулям, входящим в состав модуля `java.se`. Безымянный модуль применяется автоматически, когда файлы классов размещаются по пути к классам.

Благодаря этому предоставляется переходной путь для принятия модульной исполняющей среды Java без необходимости переходить на путь к полностью модульному приложению. Но у такого подхода имеются два следующих недостатка: ни одно из преимуществ модулей не будет доступно до тех пор,

пока приложение не будет полностью переведено на модульную платформу, а внутреннюю непротиворечивость пути к классам придется поддерживать вручную до тех пор, пока не завершится модуляризация.

Автоматические модули

Одно из ограничений модульной системы состоит в том, что из именованных модулей нельзя обращаться к архивным JAR-файлам по пути к классам. Это ограничение наложено из соображений безопасности, поскольку разработчики модульной системы стремились к тому, чтобы метаданные полностью использовались в графе зависимостей модулей. Но иногда в модульном коде требуется все же обращаться к пакетам, которые еще не модуляризированы. В качестве выхода из этого затруднительного положения можно разместить немодифицированный архивный JAR-файл непосредственно в пути к модулям, удалив его из пути к классам. Такой подход дает следующие возможности.

- Архивный JAR-файл, расположенный по пути к модулям, становится *автоматическим модулем*.
- Имя такого модуля выводится из имени архивного JAR-файла (или извлекается из файла манифеста `MANIFEST.MF`).
- Экспортируется каждый пакет.
- Требуются все остальные модули, включая и безымянный.

Эти возможности могут также облегчить переход на модульную платформу. Хотя, применяя автоматические модули, приходится все же идти на некоторые уступки в отношении безопасности.

Открытые модули

Как отмечалось ранее, простое объявление метода открытым (`public`) больше не гарантирует, что этот элемент кода будет доступен везде. Вместо этого его доступность теперь зависит также от того, экспортируется ли в определяющий его модуль тот пакет, в котором содержится этот элемент. Еще один важный вопрос, возникающий при разработке модулей, связан с применением рефлексии для доступа к классам.

Рефлексия является настолько обширным, универсальным механизмом, что на первый взгляд трудно понять, как согласовать его с целями строгой инкапсуляции, преследуемыми в модульной системе JPMS. Хуже того,

большинство самых важных библиотек и каркасов в экосистеме Java основываются на рефлексии, включая модульное тестирование, внедрение зависимостей и многое другое. Поэтому отсутствие подходящего решения для рефлексии способно сделать невозможным перевод любого реального приложения на модульную платформу.

Предоставляемое решение оказывается двояким. Во-первых, сам модуль можно объявить как открытый (`open`) следующим образом:

```
open module kathik {
  exports kathik.api;
}
```

Такое объявление имеет следующие последствия.

- Все пакеты могут быть доступны в модуле через рефлексию.
- Во время компиляции доступ к неэкспортированным пакетам *не* предоставляется.

Это означает, что такая конфигурация ведет себя во время компиляции как стандартный модуль. Общая цель состоит в том, чтобы обеспечить простую совместимость с существующим кодом и библиотеками и облегчить переход на модульную платформу. Благодаря открытому модулю удается восстановить предыдущее стремление получить рефлективный доступ к коду. Кроме того, для открытых модулей сохраняется (обычно запрещенный) доступ к закрытым и другим методам с помощью метода `setAccessible()`.

Более тщательный контроль рефлективного доступа предоставляется также с помощью ограниченного ключевого слова `opens`. При этом отдельные пакеты становятся выборочно открытыми для рефлективного доступа благодаря явному их объявлению как доступных через рефлексию.

```
module kathik {
  exports kathik.api;
  opens kathik.domain;
}
```

Вполне вероятно, что этому можно найти полезное применение, например, в модульно-ориентированной системе объектно-реляционного преобразования, где предоставляется модель предметной области и требуется полноценный рефлексивный доступ к базовым типам предметной области из модуля.

Можно пойти еще дальше, ограничив рефлексивный доступ к отдельным клиентским пакетам с помощью ключевого слова `to`. И хотя такой принцип проектирования стоит применять там, где это возможно, он все же не годится

для разработки универсальных библиотек вроде тех, что предназначены для объектно-реляционного преобразования.

Аналогично экспорт можно ограничить только отдельными внешними пакетами. Но такая возможность введена, главным образом, для оказания помощи в модуляризации самого комплекта JDK и находит ограниченное применение в пользовательских модулях.

Помимо этого, пакет можно экспортировать и открывать, хотя это и не рекомендуется делать на стадии перехода на модульную платформу. Ведь в идеальном случае доступ к пакету должен предоставляться компилятивно или же рефлексивно, но не обоими способами вместе.

Если же требуется рефлексивный доступ к пакету, уже входящему в состав модуля, то в качестве временной меры, действующей в течение переходного периода, на платформе Java предоставляются специальные параметры командной строки. В частности, параметр `--add-opens module/package=ALL-UNNAMED` может быть указан в команде `java` с целью открыть в модуле конкретный пакет для рефлексивного доступа ко всему коду по пути к классам, переопределив тем самым поведение модульной системы.

Подобным способом можно также получить рефлексивный доступ к отдельному модулю в коде, который уже стал модульным.

При переходе на модульную платформу Java любой код с рефлексивным доступом к внутреннему коду другого модуля должен выполняться с указанным выше параметром командной строки до тех пор, пока положение не исправится окончательно. С рассматриваемым здесь вопросом рефлексивного доступа (и особым его случаем) связано распространенное применение прикладных интерфейсов API внутренней платформы в библиотеках. Этот вопрос небезопасного применения подобных функциональных средств будет рассмотрен ближе к концу главы.

Службы

В состав модульной системы входит механизм *служб*, разрешающий еще одно затруднение, связанное с расширенной формой инкапсуляции. Данное затруднение легко объяснить, рассмотрев следующий вполне знакомый фрагмент кода:

```
import services.Service;
Service s = new ServiceImpl();
```

Даже если интерфейс `Service` находится в экспортированном пакете, приведенный выше фрагмент кода не будет скомпилирован, если не экспортировать также пакет, содержащий конструктор `ServiceImpl()`. В данном случае требуется механизм, позволяющий организовать тщательный контроль доступа к классам, реализующим интерфейсы служб, не импортируя весь пакет. Это дает возможность написать, например, следующий фрагмент кода:

```
module kathik {
  exports kathik.api;
  requires othermodule.services;

  provides services.Service;
  with kathik.services.ServiceImpl;
}
```

Теперь класс `ServiceImpl` доступен во время выполнения как реализующий интерфейс `Service`. Однако пакет `services` должен входить в состав другого модуля, который требуется текущему модулю, чтобы обеспечить нормальную работу данной службы.

Многоверсионные архивные JAR-файлы

Чтобы объяснить затруднение, которое позволяют разрешить многоверсионные архивные JAR-файлы, рассмотрим простой пример обнаружения идентификатора процесса, выполняющегося в настоящий момент (т.е. виртуальной машины JVM, выполняющей прикладной код).

Здесь не используется рассмотренный ранее пример выявления сетевого протокола HTTP/2, поскольку в версии Java 8 отсутствует прикладной интерфейс API для сетевого протокола HTTP/2. Ведь иначе пришлось бы приложить немало труда, чтобы предоставить равнозначные функциональные средства, а по существу, сделать полноценную вставку в прежний вариант для версии Java 8.

На первый взгляд, решить данную задачу нетрудно, но в версии Java 8 для этого придется написать поразительно много стереотипного кода:

```
public class GetPID {
  public static long getPid() {
```

```
    // В этом довольно неуклюжем вызове метода
    // применяется технология JMX для возврата имени,
    // представляющего выполняющуюся в настоящий момент
    // виртуальную машину JVM. Это имя возвращается в
    // формате <идентификатор_процесса>@<имя_хоста>, но
    // только для виртуальных машин OpenJDK и Oracle, а для
    // версии Java 8 переносимое решение не гарантируется
    final String jvmName =
          ManagementFactory.getRuntimeMXBean().getName();
    final int index = jvmName.indexOf('@');
    if (index < 1)
      return -1;

    try {
      return Long.parseLong(jvmName.substring(0, index));
    } catch (NumberFormatException nfe) {
      return -1;
    }
  }
}
```

Как видите, такое решение не является простым, как нам бы хотелось. Хуже того, оно не имеет стандартной поддержки во всех реализациях версии Java 8. Правда, начиная с версии Java 9 для решения рассматриваемой здесь задачи можно воспользоваться классом `ProcessHandle` из нового прикладного интерфейса API для процессов, как показано ниже.

```
public class GetPID {
  public static long getPid() {
    // воспользоваться новым прикладным интерфейсом API
    // для процессов, внедренным в версии Java 9...
    ProcessHandle processHandle = ProcessHandle.current();
    return processHandle.getPid();
  }
}
```

И хотя теперь применяется стандартный прикладной интерфейс API, это все равно вызывает следующий важный вопрос: как написать такой прикладной код, который будет гарантированно выполняться во всех текущих версиях Java? Ответ на этот вопрос означает построение и правильное выполнение проекта в нескольких версиях Java. С одной стороны, приходится полагаться на библиотечные классы, доступные только в последних версиях Java, а с другой — выполнять прикладной код в прежних версиях, делая в него некоторые вставки. В итоге должен быть создан единый архивный JAR-файл, не

требующий переводить проект в многомодульный формат. И на самом деле такой архивный JAR-файл должен действовать как автоматический модуль.

Рассмотрим в качестве примера проект, который должен правильно выполняться как в версии Java 8, так и в версии Java 11. Основная кодовая база данного проекта должна быть построена в версии Java 8, а некоторая ее часть — в версии Java 11. Эта последняя часть должна быть изолирована от основной кодовой базы, чтобы исключить сбои во время компиляции, хотя она может зависеть от артефактов построения сборки в версии Java 8.

Чтобы упростить конфигурацию построения сборки, допустим, что многоверсионный характер данного проекта определяется в приведенной ниже записи из файла манифеста `MANIFEST.MF`, находящегося в архивном JAR-файле.

```
Multi-Release: True
```

Альтернативный код (т.е. код для последующей версии) хранится в специальном каталоге, находящемся в каталоге `META-INF`. В данном случае это каталог `META-INF/versions/11`.

Для исполняющей среды Java, реализующей многоверсионный характер данного проекта, любые классы из каталога конкретной версии заменяют их версии из корневого каталога содержимого. С другой стороны, для Java 8 и более ранних версий запись в файле манифеста и содержимое каталога `versions/` игнорируются и обнаруживаются только классы, находящиеся в каталоге содержимого.

Преобразование в многоверсионный архивный JAR-файл

Чтобы развернуть программное обеспечение в виде многоверсионного архивного JAR-файла, необходимо выполнить следующие действия.

1. Вычленить код для конкретной версии JDK.
2. Разметить этот код по возможности в пакете или группе пакетов.
3. Начисто построить проект для версии Java 8.
4. Создать новый отдельный проект для вспомогательных классов.
5. Установить единственную зависимость для нового проекта (с артефактом версии Java 8).

Для сборки с помощью инструментального средства Gradle можно воспользоваться понятием *исходного набора* и скомпилировать код в версии

Java 11, используя другой (более современный) компилятор. Тогда архивный JAR-файл может быть построен следующим образом:

```
jar {
  into('META-INF/versions/11') {
    from sourceSets.java11.output
  }
  manifest.attributes(
    'Multi-Release': 'true'
  )
}
```

Для сборки с помощью инструментального средства Maven проще всего воспользоваться подключаемым к нему модулем Maven Dependency Plug-in и ввести модульные классы в общий архивный JAR-файл как часть отдельной стадии `generateresources`.

Переход на модульную платформу

Многим разработчикам прикладных программ на Java приходится решать нелегкий вопрос: когда следует переносить свои приложения, чтобы воспользоваться преимуществами модульной системы?

Модули должны быть выбраны по умолчанию для всех разрабатываемых заново приложений и особенно спроектированных в стиле микрослужб.

Принимая решение о переносе существующего приложения (особенно с монолитной архитектурой) на модульную платформу, можно руководствоваться следующими соображениями.

1. Обновить исполняющую среду приложения до версии Java 11, первоначально запустив ее по пути к классам.
2. Выявить любые зависимости приложений, которые были модуляризированы, и перенести их в модули.
3. Сохранить любые немодуляризированные зависимости в виде автоматических модулей.
4. Внедрить единый *монолитный модуль* всего прикладного кода.

В итоге минимально модуляризированное приложение должно быть готово к развертыванию в условиях эксплуатации. Такой модуль, как правило, будет открытым на данной стадии процесса. На следующей стадии

осуществляется реорганизация архитектуры, когда приложения разбиваются на отдельные модули по мере необходимости.

Как только прикладной код станет выполняться в модулях, рефлексивный доступ к нему, возможно, стоит ограничить с помощью ключевого слова `opens`. Такой доступ можно ограничить конкретными модулями (например, для объектно-реляционного преобразования или внедрения зависимостей) в качестве первого шага, направленного на запрет любого излишнего доступа.

Пользователям инструментального средства Maven следует помнить, что оно не является модульной системой, но у него все же имеются зависимости, привязанные к конкретной версии, в отличие от зависимостей модульной системы JPMS. Инструментальное средство Maven по-прежнему развивается в сторону полной интеграции с модульной системой JPMS (и на момент написания данной книги многие подключаемые модули еще не достигли этой цели). Тем не менее ниже даются некоторые общие рекомендации для построения модульных проектов в Maven.

- Старайтесь получить один модуль на каждую объектную модель проекта Maven POM.
- Не модуляризируйте проект Maven до тех пор, пока не будете готовы к этому полностью или хотя бы частично.
- Помните, что для выполнения прикладного кода в исполняющей среде версии Java 11 не требуется построение в наборе инструментальных средств Java 11.

В последней рекомендации указывается, что один путь для переноса проектов Maven на модульную платформу состоит в том, чтобы приступить к его построению в виде проекта для версии Java 8 и гарантии аккуратного развертывания артефактов Maven (в виде автоматических модулей) в исполняющей среде версии Java 11. И лишь после того, как первая стадия данного процесса будет доведена до вполне работоспособного состояния, можно приступать к стадии модуляризации.

Для оказания действенной помощи на стадии модуляризации имеется неплохая инструментальная поддержка. Начиная с версии 8 в состав платформы Java входит утилита `jdeps` (см. главу 13), предназначенная для определения тех пакетов и модулей, от которых зависит прикладной код. Это очень удобно для перехода от версии Java 8 к Java 11, поэтому утилитой `jdeps` рекомендуется пользоваться при переделке архитектуры приложений.

Специальные образы файлов на стадии выполнения

Одна из главных целей модульной системы JPMS состоит в том, чтобы дать возможность оперировать в приложении более мелким подмножеством модулей вместо того, чтобы обеспечивать наличие каждого класса в традиционной для версии Java 8 монолитной исполняющей среде. Такие приложения могут занимать намного меньший объем оперативной памяти, сокращая издержки на время запуска и оперативную память. Более того, если требуются не все классы сразу, то почему бы не поставлять приложение вместе со специально сокращенным образом файлов на стадии выполнения, включающим в себя лишь самое необходимое?

Чтобы продемонстрировать эту идею на практике, упакуем рассмотренное ранее инструментальное средство для проверки веб-сайтов на применение сетевого протокола HTTP/2 в самостоятельный модуль со специальной исполняющей средой. Этой цели можно добиться с помощью утилиты `jlink`, входящей в состав платформы Java с версии 9, следующим образом:

```
$ jlink --module-path httpchecker.jar:$JAVA_HOME/jmods \
        --add-modules http2checker \
        --launcher http2chk=http2checker \
        --output http2chk-image
```

В данном случае предполагается, что архивный JAR-файл `httpchecker.jar` был создан с главным классом в качестве точки входа. Результат сохраняется в выходном каталоге `http2chk-image`, занимающем около 39 Мбайт, т.е. намного меньше, чем полный образ, особенно если учесть, что рассматриваемому здесь инструментальному средству потребуются библиотеки для обеспечения безопасности, шифрования и прочего, поскольку в нем применяется новый модуль HTTP.

Инструментальное средство `http2chk` можно запустить на выполнение непосредственно из каталога со специальным образом и убедиться, что оно работает даже в том случае, если на компьютере отсутствует требующаяся версия утилиты `java`.

```
$ java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
$ ./bin/http2chk https://www.google.com
https://www.google.com: HTTP_
```

Развертывание специальных образов файлов на стадии выполнения довольно новое средство, но оно обладает немалым потенциалом для сокращения объема памяти, занимаемой прикладным кодом, помогая Java выдерживать конкуренцию в эпоху микрослужб. В будущем утилиту `jlink` можно будет даже применять вместе с новыми средствами компиляции, включая компилятор Graal, которым можно пользоваться как для статической, так и для динамической компиляции (см. описание утилиты `jaotc` в главе 13). Тем не менее совместное использование утилит `jlink` и `jaotc` в версии Java 11 пока еще не дает никаких решающих преимуществ в отношении производительности.

Трудности, связанные с модулями

Несмотря на то что модульная система является главным новшеством в версии Java 9, которому было уделено немало времени и сил разработчиков языка Java, она все же не лишена недостатков. И это, вероятно, было неизбежно, поскольку модульная система коренным образом изменяет порядок проектирования и выпуска прикладных программ на Java. Ведь было бы практически невозможно избежать некоторых трудностей, переводя на модульную платформу столь крупную и зрелую экосистему, как Java.

Класс `Unsafe` и затруднения, связанные с ним

Класс `sun.misc.Unsafe` широко применяется разработчиками библиотек и каркасов и другими реализаторами программного обеспечения в среде Java. Но этот класс относится к внутренней реализации и не входит в состав стандартного прикладного интерфейса API на платформе Java, на что ясно указывает имя его пакета. Да и само имя этого класса вполне определенно указывает на то, что он не предназначен для применения в прикладных программах на Java.

Класс `Unsafe` относится к неподдерживаемому внутреннему прикладному интерфейсу API, поэтому он может быть отвергнут или модифицирован в любой новой версии Java, не принимая во внимание последствия для пользовательских приложений. Любой код, в котором применяется этот класс, формально связан непосредственно с виртуальной машиной HotSpot, потенциально считается нестандартным и может не выполняться в других реализациях.

И хотя класс `Unsafe` официально не входит в состав платформы Java SE, он так или иначе стал фактически стандартной и главной составляющей реализации, по существу, каждой основной библиотеки. В продолжение последовательного ряда версий он развился в нечто вроде места свалки для нестандартных, но необходимых функциональных средств. Эта мешанина представляет собой разнородную массу функциональных средств с разной степенью безопасности, надежности и предоставляемых возможностей. Ниже перечислены некоторые примеры применения класса `Unsafe`.

- Быстрая сериализация и десериализация.
- Потокобезопасный 64-разрядный платформенно-ориентированный доступ к оперативной памяти (например, за пределами "кучи").
- Атомарные операции в оперативной памяти (например, сравнение с обменом).
- Быстрый доступ к полям и памяти.
- Замена интерфейса JNI на многоперационную систему.
- Доступ к элементам массива с изменчивой семантикой.

Значительные трудности связаны с тем, что многие библиотеки и каркасы не удавалось перевести на версию Java 9 без замены некоторых функциональных средств класса `Unsafe`. И это оказывает влияние на всех, кто пользуется любыми библиотеками и каркасами, а по существу, каждым приложением в экосистеме Java.

Для устранения этого недостатка в компании Oracle были разработаны новые прикладные интерфейсы API для поддержки тех, кому требуются подобные функциональные средства, а также выделены в отдельный пакет `jdk.unsupported` прикладные интерфейсы API, которые нельзя инкапсулировать со временем в модуль. Благодаря этому разработчикам ясно дано понять, что такие прикладные интерфейсы официально не поддерживаются, а следовательно, они вольны пользоваться ими на свой страх и риск.

Это дает классу `Unsafe` временный мандат на применение в течение строго ограниченного периода времени, поощряя в то же время разработчиков библиотек и каркасов переходить на новые прикладные интерфейсы API. Характерным примером такой замены служит прикладной интерфейс API для дескрипторов переменных, расширяющих понятие *дескрипторов методов* (см. главу 11) и вводящих новые функциональные возможности (например,

режимы для барьеров параллелизма в версии Java 9). Эти средства наряду с обновлениями модели памяти Java (JMM) призваны составить стандартный прикладной интерфейс API для доступа к новым низкоуровневым средствам ЦП, запрещая в то же время разработчикам полный доступ к опасным функциональным возможностям, которые можно было обнаружить раньше в классе `Unsafe`. Подробнее о классе `Unsafe` и связанных с ним платформенно-ориентированных методиках можно узнать в книге *Optimizing Java* (издательство O'Reilly, 2018 г.).

Отсутствие контроля версий

В стандарт на модульную систему JPMS в версии Java 11 не входит контроль версий зависимостей.

Такое проектное решение было принято намеренно, чтобы упростить выпускаемую модульную систему. Хотя это не препятствует в перспективе включению в модули зависимостей, привязанных к версии.

В сложившемся теперь положении для контроля версий зависимостей отдельных модулей требуются внешние инструментальные средства. Такие средства могут, например, входить в состав объектной модели проекта (POM), создаваемой в инструментальном средстве сборки Maven. Преимущество такого подхода заключается в том, что загрузку и контроль версий можно осуществлять в локальном хранилище инструментального средства сборки.

Но, несмотря на все это, непреложным остается простой факт: сведения о версиях зависимостей должны храниться за пределами модуля и не входить в состав архивного JAR-файла. И от этого довольно скверного факта никуда не деться, хотя положение сейчас не хуже, чем прежде, когда зависимости выводились из пути к классам.

Медленные темпы внедрения

После версии Java 9 модель выпусков версий Java коренным образом изменилась. Так, в версиях Java 8 и 9 применялась модель ключевых выпусков, где одно главное функциональное средство (например, лямбда-выражения или модули), по существу, определяло весь выпуск, а следовательно, дата его появления на свет зависела от окончательной готовности главного

функционального средства. Недостаток такой модели заключался в неэффективности выпусков из-за неопределенности их дат появления на свет. В частности, появления мелкого функционального средства, не удостаивавшегося отдельного выпуска, приходилось ждать вплоть до следующей главной версии.

В итоге в версии Java 10 была принята новая модель выпусков, внедрявшая *строго ограниченный по времени контроль версий*, включая следующее.

- Версии Java теперь классифицируются как функциональные, выпускаемые с регулярной периодичностью через каждые шесть месяцев.
- Функциональные средства не внедряются на платформе до тех пор, пока не будут окончательно готовы.
- Хранилище с основной веткой разработки постоянно находится в состоянии выпуска.

Такие выпуски пригодны лишь в течение шести месяцев, по истечении которых они больше не поддерживаются. Каждые три года выпускается специальная версия, предназначенная для *долгосрочной поддержки* (LTS), имеющей расширенную поддержку.

Несмотря на то что сообщество программирующих на Java отнеслось, в общем, положительно к новому ускоренному циклу выпусков, темпы внедрения Java 9 и последующих версий стали намного меньшими, чем предыдущих версий. Возможно, это объясняется стремлением крупных предприятий пользоваться более продолжительными циклами поддержки, а не обновлять каждое функциональное средство по истечении всего лишь шести месяцев.

Кроме того, обновление Java 8 до версии 9 не является упрощенной заменой, в отличие от перехода от версии 7 к версии 8 и в меньшей степени — от версии 6 к версии 7. Модульная система коренным образом изменяет многие свойства платформы Java, даже если модули не применяются в приложениях, рассчитанных на конечных пользователей. Вследствие этого разработчики неохотно переходят от Java 8 к новой версии, если они не видят в этом явных преимуществ.

Это приводит к проблемам типа "курицы и яйца", когда разработчикам трудно отделить причину от следствия, и поэтому они не спешат переходить на новую версию из-за того, что в библиотеках и прочих применяемых ими компонентах еще не поддерживается модульная система Java. С другой стороны, компании и сообщества разработчиков программного обеспечения с открытым кодом, поддерживающие библиотеки и прочие инструментальные

средства, считают поддержку следующих после Java 8 версий низкоприоритетной, поскольку круг пользователей модульной системы Java все еще невелик. Появление Java 11 — первой после Java 8 версии с долгосрочной поддержкой — может поправить это положение, поскольку в ней предоставляется поддерживаемая среда, которую разработчики корпоративных приложений могут посчитать удобной целью для перехода на новую версию.

Резюме

Модульные средства впервые внедрены в версии Java 9 и нацелены на разрешение сразу нескольких затруднений. В частности, они позволяют сократить время запуска прикладных программ, объем занимаемой памяти и сложность путем отказа в доступе к внутренним элементам. Долгосрочных целей, направленных на то, чтобы усовершенствовать архитектуру приложений и приступить к обдумыванию новых подходов к компиляции и развертыванию, еще предстоит достичь.

Тем не менее непреложным остается тот простой факт, что после выпуска версии Java 11 немногие разработчики и проекты целиком и полностью перешли на модульную платформу. В связи с этим ожидается, что модуляризация окажется долгосрочным проектом, который не скоро окупится сторицей и опирается на сетевые эффекты в экосистеме Java, чтобы достичь полной выгоды.

Так или иначе, разработчики должны определенно и с самого начала рассматривать возможность построения новых приложений модульным способом. Хотя общая история модуляризации платформы в экосистеме Java только начинается.