8 Постепенная модуляризация существующих проектов

В этой главе

- Работа с безымянными модулями.
- Помощь в модуляризации с помощью автоматических модулей.
- Наращиваемая модульная кодовая база.
- Смешивание пути класса и пути модуля.

В зависимости от того, насколько гладко прошел ваш переход на Java 9+ (см. главы 6 и 7), вы можете столкнуться с некоторыми из более неприятных последствий внедрения системы модулей в достаточно развитую экосистему. Хорошая новость — это того стоило! Как я кратко показал в подразделе 1.7.1, Java 9+ может многое предложить, помимо системы модулей. Если вы в состоянии повысить требования Java к своему проекту до 9, то можете сразу начать использовать их.

Можно наконец начать модуляризацию вашего проекта. Превратив артефакты в модульные JAR-файлы, вы и ваши пользователи можете получить выгоду от надежной конфигурации (см. подраздел 3.2.1), надежной инкапсуляции (см. подраздел 3.3.1), развязки с помощью сервисов (см. главу 10), образов среды выполнения, включая целые приложения (см. раздел 14.2), и других связанных с модулем ценных свойств. Как будет показано в подразделе 9.3.4, можно модулировать даже проекты, которые работают на Java 8 и более ранних версиях.

Существует два способа сделать JAR модульными:

- подождать, пока все зависимости не станут модульными, а затем создать дескрипторы модулей для всех артефактов сразу;
- начать с ранней модуляризации только ваших артефактов, возможно, всего парочки за раз.

С учетом того, что обсуждается в главах 3, 4 и 5, реализация первого варианта должна быть простой. Вам могут понадобиться некоторые более расширенные функции системы модулей, которые представлены в главах 10 и 11, но, кроме этого, можно пойти дальше: создать декларацию модуля для каждого нового артефакта и смоделировать их отношения, как было изучено ранее.

Возможно, ваш проект находится глубоко в дереве зависимостей и не придется ждать, пока все зависимости станут модульными. Или, вероятно, ваш проект слишком велик, чтобы превратить все артефакты в модули за один раз. В этих случаях вас может заинтересовать второй вариант, который позволяет постепенно наращивать модульные артефакты независимо от того, являются ли их зависимости модульными или простыми JAR-файлами.

Возможность использовать модульные и немодульные артефакты бок о бок не только важна для отдельных проектов, но и означает, что экосистема в целом может добавлять модули независимо друг от друга. Без этого модуляризация экосистемы могла бы занять несколько десятков лет — а так любой сможет сделать это в течение одного десятилетия.

Эта глава посвящена функциям, которые позволяют постепенно наращивать модульность существующих проектов: мы начнем с обсуждения комбинации путей класса и модуля, затем исследуем безымянный модуль и в заключение рассмотрим автоматические модули. По завершении ваш проект или его части получат выгоду от применения системы модулей, несмотря на потенциально немодулированные зависимости. Вы также будете хорошо подготовлены к главе 9, в которой рассматриваются стратегии модуляризации приложений.

8.1. Почему наращивание модульности необязательно

Прежде чем перейти к тому, как постепенно наращивать модульность проекта, я хочу разобраться, почему это необязательно. Системы модулей обычно требуют, чтобы все было модульно. Но если они не появляются вовремя (например, JPMS) или используются лишь небольшой частью своей экосистемы (скажем, модулями OSGi или JBoss), то вряд ли ожидается, что это произойдет. Нужно найти способ взаимодействовать с немодульными артефактами.

В данном разделе мы сначала подумаем о развитии событий, если каждый JAR должен быть модульным для запуска на Java 9+; это приводит к выводу, что должна быть возможность смешивать простые JAR и модули (см. подраздел 8.1.2). Затем я покажу, как использование пути класса и пути модуля бок о бок позволяет применить данный подход «смешай и соедини» (см. подраздел 8.1.3).

8.1.1. Если бы требовалось, чтобы каждый JAR был модульным

Если бы JPMS была строгой и требовала, чтобы все было модульно, то ею можно было бы пользоваться, только если все JAR содержали бы дескриптор модуля. А поскольку система модулей — неотъемлемая часть Java 9+, в результате ее нельзя было бы обновить, не модуляризовав весь код и зависимости. Представьте последствия, будь так на самом деле.

Одни проекты могут обновиться до Java 9+ на ранней стадии, что заставит всех их пользователей модуляризовать свои кодовые базы или прекратить взаимодействие с проектом. Другие могут не захотеть форсировать данное решение или же могут иметь другие причины не перепрыгивать на модули, а это сдерживает их пользователей. Я не хотел бы, чтобы у моего проекта были зависимости, которые принимали противоположные решения. Что я мог бы сделать?

С другой стороны, некоторые проекты будут поставлять отдельные варианты с дескрипторами модулей и без них, вследствие чего им придется задействовать два совершенно непересекающихся набора зависимостей (один с дескрипторами модулей и один без них). Кроме того, если бы они не вносили исправления в старые и новые версии, то пользователи были бы вынуждены одновременно выполнять много (вероятно, требующих много времени) обновлений, чтобы иметь возможность перейти на Java 9+. И это даже без учета проектов, которые больше не поддерживаются и быстро стали бы непригодными для Java 9+, даже если бы сами не имели никаких зависимостей.

Единственный способ избежать бесполезных усилий и глубокого раскола заключается в том, чтобы в сообществе наступил день, когда *все проекты* обновились до Java 9+ и начали выпускать модульные JAR-файлы. Но не существует способа сделать это. И как бы мы это ни скрывали, любой человек, запускающий JAR, должен знать, для какой версии Java тот был создан, поскольку не будет работать на 8 *и* 9. Итак, у нас большие проблемы!

8.1.2. Смешивание и соединение простых JAR-файлов с модулями

Чтобы обойти данную проблему, система модулей должна предлагать способ запуска немодульного кода поверх модульной JVM. Во введении к главе 6 я объясняю, что это действительно так и простые JAR-файлы в пути к классам работают так же, как и до Java 9+. (Как говорится в главах 6 и 7, код, который они содержат, может не работать, но это другой вопрос.) В разделе 8.2 освещается, как работает *режим пути к классам*.

Тот факт, *что* он работает, уже является важным открытием: система модулей может обрабатывать немодульные артефакты и знает, как перемещаться по границе между ними и явными модулями. Это хорошая и не единственная новость: данная граница не является незыблемой. Не нужно отделять JAR-файлы приложения от модулей JVM. Как показано на рис. 8.1 и исследуется на протяжении этой главы, система модулей позволяет перемещать эту границу, а также смешивать и сопоставлять модульные и немодульные JAR-файлы приложения с платформенными модулями, как того требуют ваши проекты.

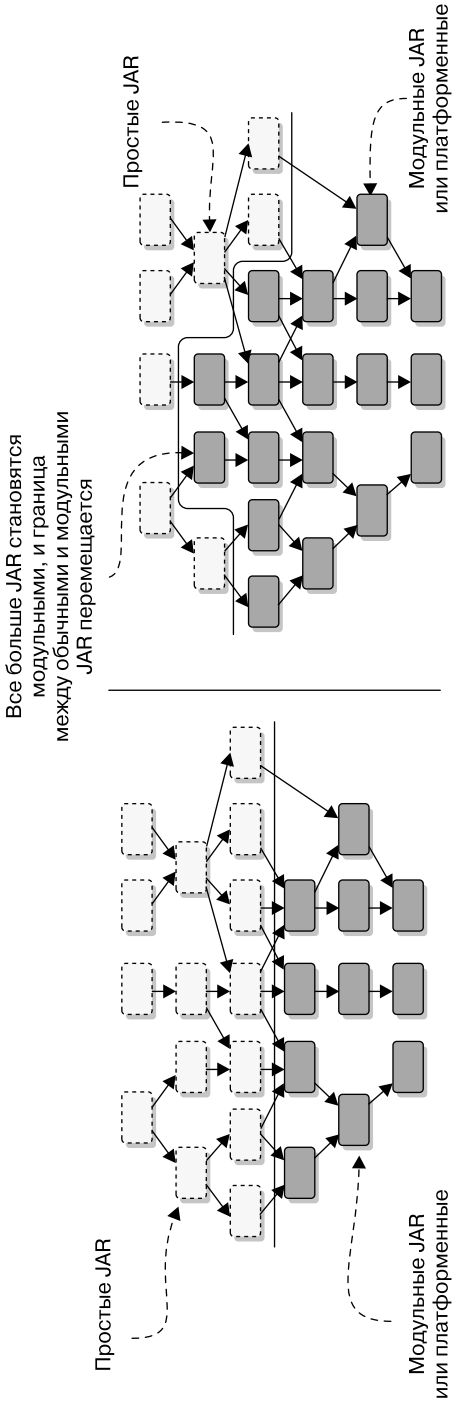

Рис. 8.1. Система модулей позволяет выполнять немодульный код на модульном JDK *(слева)*. Что еще более важно, она предоставляет инструменты для перемещения этой границы *(справа)*

8.1.3. Технические основы наращиваемой модульности

Основной принцип, который делает вероятной постепенную модуляризацию, заключается в том, что пути классов и модулей могут использоваться бок о бок. Нет необходимости перемещать все JAR приложения из пути класса в путь модуля за один раз. Вместо этого в существующих проектах рекомендуется начинать с пути класса, а затем медленно перемещать свои артефакты в путь модуля по мере приложения усилий по модуляризации.

Использование обоих путей одновременно с простыми и модульными JAR-файлами требует четкого понимания того, как соотносятся эти концепции. Можно подумать, будто JAR без дескриптор модуля находятся в пути класса, а модульные — в пути модуля. Хотя я никогда так не говорил, вас простят за то, что вы читали между строк. Тем не менее данная теория ошибочна, и сейчас самое время от нее отказаться.

Два механизма опровергают эту теорию и делают возможным наращивание модуляризации:

- *безымянный модуль* неявно создается системой модулей со всем содержимым, загруженным из пути классов. В нем живет хаос пути классов (в разделе 8.2 это объясняется подробно);
- система модулей создает *автоматический модуль* для каждого простого JAR-файла, найденного в пути модуля (этой концепции посвящен раздел 8.3).

Путь к классу не делает различий между обычными и модульными JAR-файлами: если те находятся в пути к классам, то попадают в безымянный модуль. Точно так же путь модуля еле различает простые и модульные JAR-файлы: если те находятся в пути к модулю, то рассматриваются как именованные модули. (Для простых JAR система модулей создает автоматический модуль; для модульных — явный модуль в соответствии с описанием.)

Чтобы понять остальную часть этой главы, а также провести модуляризацию, важно полностью усвоить данное поведение. Таблица 8.1 показывает двумерную переработку. Не тип JAR (простой или модульный), а путь, в котором он размещен (путь класса или путь модуля), определяет, станет ли он частью безымянного или именованного модуля.

Таблица 8.1. Не тип JAR, а путь, в котором он размещен, определяет, где будет класс: в именованном модуле или в безымянном

JAR	Путь класса	Путь модуля
Обычный	Безымянный модуль (см. раздел 8.2)	Автоматический модуль (см. раздел 8.3)
Модульный	—	Явный модуль (см. подраздел 3.1.4)

При принятии решения о том, помещать ли JAR в путь класса в или путь модуля, дело не в том, откуда берется код (является ли JAR модульным), а в том, где он необходим (в безымянном или именованном модуле). Путь класса предназначен для

кода, который вы хотите добавить в комок грязи, а путь модуля — для кода, предназначенного стать модулем.

Но как решить, куда отправить код? Обычно безымянный модуль касается *совместимости*, позволяя проектам, использующим путь классов, работать на Java 9+; тогда как автоматические модули — это *модульность*, позволяющая проектам использовать систему модулей, даже если зависимости еще не модульны.

Для более подробного ответа стоит присмотреться к безымянным и автоматическим модулям. Затем, в главе 9, мы определим более крупные стратегии модуляризации. Если вас интересует, стоит ли модулировать существующий проект, то обратитесь к подразделу 15.2.1.

ПРИМЕЧАНИЕ

Ваш инструмент сборки может принять многие из этих решений за вас. Тем не менее все равно есть вероятность оказаться в ситуациях, когда что-то пошло не так, и в данном случае вы можете применить материал этой главы для правильной настройки сборки.

8.2. Безымянный модуль, он же путь к классу

Есть один аспект, который я еще не объяснил подробно: как система модулей и путь к классам работают вместе? Первая часть книги дает четкое представление о том, как модульные приложения размещают все в пути модуля и работают с модульным JDK. Затем последовали главы 6 и 7, посвященные компиляции немодульного кода и запуску приложений из пути классов. Но как содержимое пути к классу взаимодействует с системой модулей? Какие из них разрешены и как? Почему содержимое пути класса может получить доступ ко всем платформенным модулям? Безымянный модуль отвечает на эти вопросы.

Их изучение имеет не только академическую ценность. Если приложение не является достаточно маленьким, то, вероятно, не может быть полностью модульным; но наращиваемая модульность включает в себя смешивание JAR-файлов и модулей, путей классов и модулей. Это очень важно для понимания основных деталей того, как работает режим пути к классам системы модулей.

ПРИМЕЧАНИЕ

Механизмы, окружающие безымянный модуль, обычно применяются во время компиляции и выполнения, но всегда упоминание и того и другого излишне раздувает текст. Вместо этого я описываю поведение во время выполнения и упоминаю время компиляции только тогда, когда поведение отличается.

Безымянный модуль содержит все одномодульные классы, которые:

- во время компиляции компилируются, если не содержат дескриптор модуля;
- во время компиляции и выполнения загружаются из пути к классам.

Как описано в подразделе 3.1.3, все модули имеют три основных свойства, и это также верно для безымянного модуля:

- *имя* — у безымянного модуля его нет (что имеет смысл, верно?), то есть никакой другой модуль не может упомянуть его в своих объявлениях (например, запросить его);
- *зависимости* — безымянный модуль считывает все другие модули, из которых состоит граф;
- *экспорты* — безымянный модуль экспортирует все свои пакеты, а также открывает их для рефлексии (подробности об открытых пакетах и модулях см. в разделе 12.2).

В отличие от безымянного модуля все остальные называются *именованными*. Сервисы, предоставляемые в `META-INF/services`, доступны для `ServiceLoader`. В главе 10 представлено введение в сервисы, и, в частности, в подразделе 10.3.6 описано их взаимодействие с безымянным модулем.

Хотя она и не настолько проста, концепция безымянного модуля имеет смысл. Здесь есть упорядоченный граф модулей, а там, немного в стороне, есть хаос путей к классам, сосредоточенный в собственном модуле, доступном для всех, с некоторыми особыми свойствами (рис. 8.2). (Чтобы не усложнять вопросы сверх необходимого, я не говорил ранее, но безымянный модуль лежит в основе глав 6 и 7, где можно заменить любое вхождение содержимого *пути класса безымянным модулем*.)

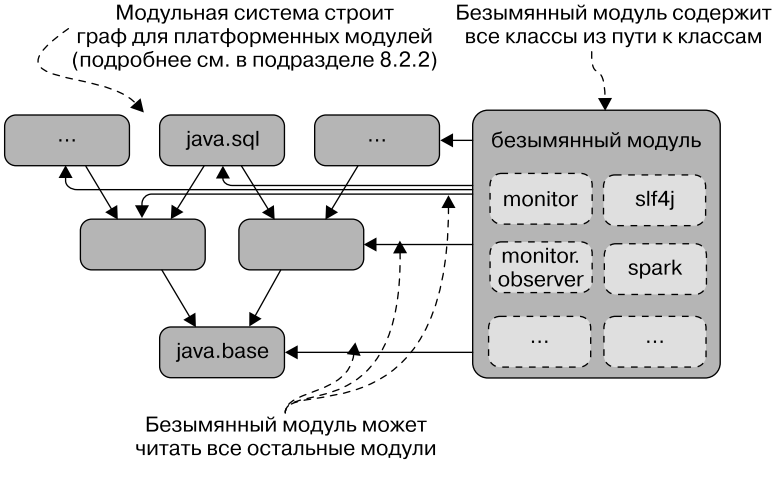

Рис. 8.2. Запущенная со всеми JAR-файлами приложения в пути к классам, система модулей строит граф из платформенных модулей *(слева)* и распределяет все классы из пути к классам к безымянному модулю *(справа)*, который может читать все другие модули

Вернемся к приложению *ServiceMonitor* и предположим, что оно было написано до Java 9. Код и его организация идентичны тем, которые мы обсуждали в предыдущих главах, но в нем отсутствуют объявления модулей, поэтому создаются простые JAR-файлы вместо модульных.

Предполагая, что папка `jars` содержит все JAR-файлы приложения, а `libs` — все зависимости, можно запустить приложение следующим образом:

```
$ java --class-path 'jars/*':'libs/*' monitor.Main
```

Это работает в Java 9+ и, кроме альтернативной формы параметра `--class-path`, делает то же самое, что в Java 8 и более ранних версиях. На рис. 8.2 показан граф модулей, который система модулей создает для данной конфигурации запуска.

Теперь, понимая это, вы хорошо подготовлены к запуску простых, одномодульных приложений из пути класса. Помимо этого базового варианта использования и особенно при медленной модуляризации приложения тонкости безымянного модуля становятся актуальными, вследствие чего мы рассмотрим их далее.

8.2.1. Хаос пути к классам, захваченного безымянным модулем

Основная цель безымянного модуля — захватить содержимое пути класса и заставить его работать в системе модулей. Поскольку в пути к классам никогда не было границ между JAR, нет смысла устанавливать их сейчас; поэтому наличие единственного безымянного модуля для всего пути класса — разумное решение. Внутри него, как и в пути к классам, доступны все публичные классы и не существует концепции разделенных пакетов.

Отличительная роль безымянного модуля и его ориентация на обратную совместимость придают ему несколько особых свойств. В разделе 7.1 вы заметили, что во время выполнения строгая инкапсуляция платформенных модулей в основном отключена для кода в безымянном модуле (по крайней мере в Java 9, 10 и 11). Когда мы обсуждали разделенные пакеты в разделе 7.2, обнаружилось, что безымянный модуль не сканируется, поэтому разделения пакетов между ним и другими модулями не обнаруживаются, а часть пути к классам недоступна.

Одна деталь, немного нелогичная, из-за которой легко ошибиться, — это то, что составляет безымянный модуль. Кажется очевидным, что модульные JAR-файлы становятся модулями и, следовательно, простые JAR-файлы переходят в безымянный модуль, верно? Как объяснено в подразделе 8.1.3, это неправильно: безымянный модуль отвечает за *все JAR в пути к классам*, модульные или нет.

Как следствие, модульные JAR-файлы не обязательно должны загружаться как модули! Если библиотека начинает предоставлять модульные JAR-файлы, то ее пользователи ни в коем случае не должны применять их в качестве модулей. Вместо этого они могут оставить их в пути к классам, где их код объединен в безымянный модуль. Как более подробно объясняется в разделе 9.2, это позволяет экосистеме модулировать части приложений практически независимо друг от друга.

В качестве примера запустим полностью модульную версию *ServiceMonitor*, один раз из пути класса и один раз из пути модуля:

```
$ java --class-path 'mods/*':'libs/*' -jar monitor
$ java --module-path mods:libs --module monitor
```

И тот и другой работают нормально и без каких-либо явных различий.

Один из способов увидеть, как система модулей обрабатывает оба случая, — использовать API, который мы более подробно рассмотрим в подразделе 12.3.3. Можно

вызвать `getModule` для класса, чтобы получить модуль, которому он принадлежит, а затем `getName` для этого модуля, чтобы получить его имя. Для безымянного модуля `getName` возвращает `null`.

Добавим следующие строки кода в `Main`:

```
String moduleName = Main.class.getModule().getName();
System.out.println("Module name: " + moduleName);
```

При запуске из пути класса в результате получаем `Module name: null`, что указывает на то, что класс `Main` оказался в безымянном модуле. При запуске из пути модуля получим ожидаемое `Module name: monitor`.

В подразделе 5.2.3 обсуждается, как система модулей инкапсулирует ресурсы в пакетах. Это лишь частично относится к безымянному модулю: в модуле нет ограничений доступа (вследствие чего все JAR-файлы из пути к классам могут обращаться к ресурсам друг друга), а безымянный модуль открывает все пакеты для рефлексии (так что все модули могут получать доступ к ресурсам из JAR-файлов в пути к классам). Тем не менее строгая инкапсуляция применяется для доступа из безымянного к именованному модулю.

8.2.2. Разрешение модулей для безымянного модуля

Важный аспект отношения безымянного модуля к остальной части графа — то, какие другие модули он может читать. Выше описано, что он может читать все модули, которые составляют граф. Но какие именно?

Напомню материал подраздела 3.4.1: разрешение модуля строит граф, начиная с корневых модулей (в частности, исходного), а затем итеративно добавляя все их прямые и переходные зависимости. Как это будет работать, если код в процессе компиляции или метод `main` приложения находится в безымянном модуле, как в случае запуска приложения из пути к классам? В конце концов, простые JAR-файлы не выражают никаких зависимостей.

Если начальный модуль является безымянным, то все системные модули (в первую очередь все модули *java.** и *jdk.**) и все модули на пути к модулю обновления, которые экспортируют хотя бы один пакет без квалификации, становятся корневыми.

- Точный набор модулей *java.**, которые становятся корневыми, зависит от наличия модуля *java.se* (модуля, представляющего весь API Java SE, — он присутствует в полных образах Java, но может отсутствовать в пользовательских образах среды выполнения, созданных с помощью `jlink`):
  - если *java.se* является обозреваемым, то становится корневым;
  - если это не так, то каждый системный модуль *java.** и модуль *java.** из обновленного пути к модулю, который экспортирует хотя бы один пакет без квалификации (то есть без ограничений на то, кто может получить доступ к пакету, см. раздел 11.3), становится корневым.
- Помимо модулей *java.**, каждый другой системный модуль и модуль из обновленного пути модуля, который не является инкубационным модулем и экспортирует

хотя бы один пакет без квалификации, становится корневым. Особенно это относится к модулям *jdk.** и *javafx.**.

- Модули, определенные с помощью `--add-modules` (см. подраздел 3.4.3), всегда являются корневыми.

Это кажется сложным (см. рис. 8.3 для визуализации), но может стать важным в крайних случаях. Эмпирическое правило о том, что все системные модули, кроме JEE и инкубационных, разрешены, должно охватывать как минимум 90 % случаев.

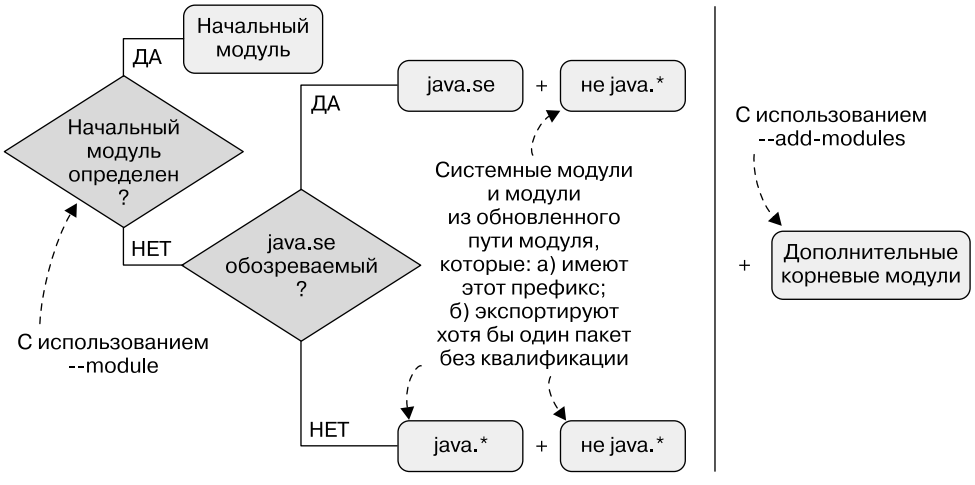

Рис. 8.3. Какие модули становятся корневыми для разрешения модулей (см. подраздел 3.4.1), зависит от того, был ли начальный модуль определен с помощью --module (если нет, то безымянный модуль становится исходным), и от того, является ли java.se обозреваемым. В любом случае модули, определенные с помощью --add-modules, всегда корневые

В качестве примера можно запустить `java --show-module-resolution` и наблюдать первые несколько строк вывода:

```
> root java.se jrt:/java.se
> root jdk.xml.dom jrt:/jdk.xml.dom
> root javafx.web jrt:/javafx.web
> root jdk.httpserver jrt:/jdk.httpserver
> root javafx.base jrt:/javafx.base
> root jdk.net jrt:/jdk.net
> root javafx.controls jrt:/javafx.controls
> root jdk.compiler jrt:/jdk.compiler
> root oracle.desktop jrt:/oracle.desktop
> root jdk.unsupported jrt:/jdk.unsupported
```

Это не весь вывод, и порядок в вашей системе может быть другим. Но, начав сверху, можно заметить, что *java.se* — единственный модуль *java.**. Кроме того, есть несколько модулей *jdk.** и *javafx.** (определяют *jdk.unsupported* из подраздела 7.1.1), а также модуль *oracle.** (понятия не имею, что он делает).

ВАЖНАЯ ИНФОРМАЦИЯ

Обратите внимание: с безымянным модулем в качестве исходного набор корневых модулей всегда является подмножеством системных модулей, содержащихся в образе выполнения. Модули, присутствующие в пути модуля, никогда не будут разрешены, если явно не добавлены с помощью --add-modules. В случае установления пути к модулю вручную, чтобы он содержал именно те модули, которые необходимы, можно добавить их все с помощью --add-modules ALL-MODULE-PATH, как описано в подразделе 3.4.3.

Можно легко наблюдать это поведение, запустив *ServiceMonitor* из пути к модулю без определения исходного модуля:

```
$ java --module-path mods:libs monitor.Main

> Error: Could not find or load main class monitor.Main
> Caused by: java.lang.ClassNotFoundException: monitor.Main
```

Выполнение той же команды с параметром `--show-module-resolution` подтверждает, что модули *monitor.** не разрешены. Исправить это можно с помощью `--add-modules monitor`, тогда модуль *monitor* добавляется в список корневых, или `--module monitor/monitor.Main`, в этом случае *monitor* становится единственным корневым модулем (начальным).

8.2.3. Зависимость от безымянного модуля

Одна из основных целей системы модулей — надежная конфигурация: модуль должен выражать свои зависимости, а система должна быть в состоянии гарантировать их присутствие. Мы установили это в разделе 3.2 для явных модулей с дескриптором. Что произойдет, если вы попытаетесь расширить надежную конфигурацию до пути к классам?

Проведем мысленный эксперимент. Представьте, что модули могут зависеть от содержимого пути к классам, скажем, с чем-то вроде `requires class-path` в дескрипторе. Какие гарантии может дать система модулей для такой зависимости? Выяснилось, что почти никаких. Пока существует хотя бы один класс из пути к классам, система модулей будет предполагать, что зависимость присутствует. Это не поможет (рис. 8.4).

Хуже того, это серьезно подорвет надежную конфигурацию, поскольку вы можете оказаться в зависимости от модуля с `requires class-path`. Хорошо, он не содержит никакой информации — что *именно* нужно положить в путь классу (опять же см. рис. 8.4)?

Развивая эту гипотезу, представьте, что два модуля, *com.framework* и *org.library*, зависят от одного и того же третьего модуля, скажем SLF4J. Один из них объявил зависимость до того, как SLF4J был модульным, и, следовательно, имеет `requires class-path`; другой объявил о своей зависимости от модульного SLF4J и, следовательно, имеет `requires org.slf4j` (при условии, что это имя модуля). Теперь, в какой путь кто-либо в зависимости от *com.framework* и *org.library* поместит JAR

SLF4J? Что бы они ни выбрали, система модулей должна была определить, что одна из двух переходных зависимостей не была заполнена. На рис. 8.5 показана эта гипотетическая ситуация.

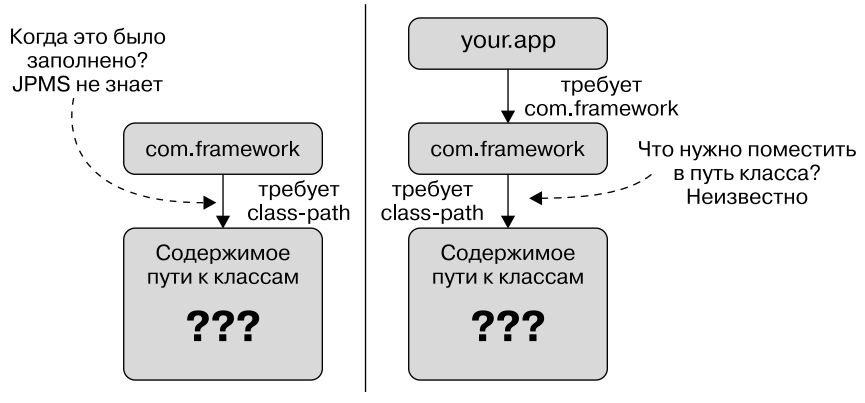

Рис. 8.4. Если com.framework зависит от некоего содержимого пути к классу с гипотетическим requires class-path, то система модулей не может определить, было ли выполнено это требование *(слева)*. Создав свое приложение на данной платформе, вы не будете знать, как заполнить эту зависимость *(справа)*

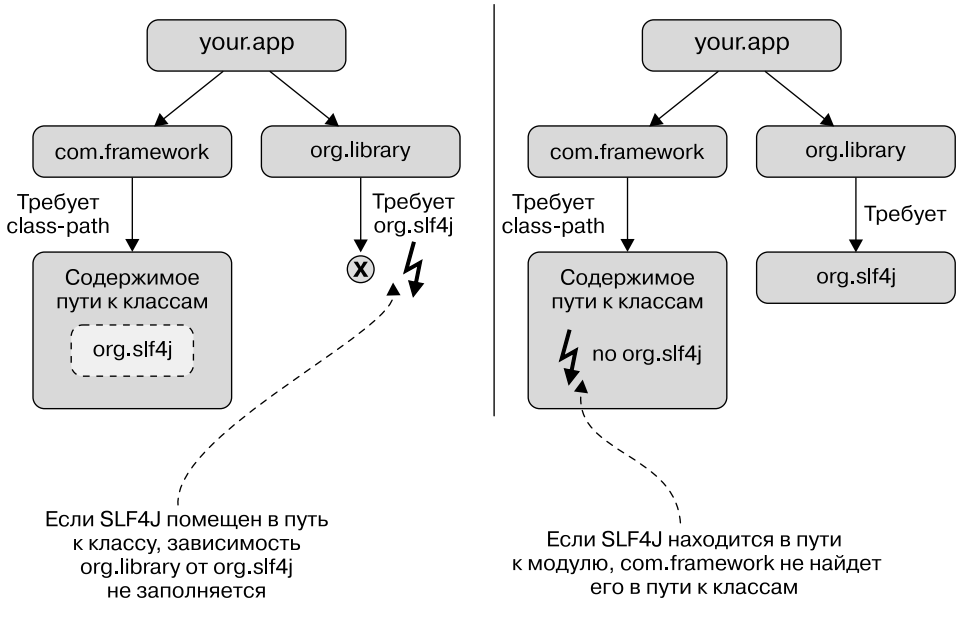

Рис. 8.5. Если com.framework зависит от SLF4J с гипотетическим requires class-path и org.library запрашивает его как модуль с requires org.slf4j, то не существовало бы способа удовлетворить оба требования. Независимо от того, был ли SLF4J размещен в пути к классам *(слева)* или в пути к модулям *(справа)*, одна из двух зависимостей будет считаться незаполненной

Размышление над всем вышесказанным приводит к выводу, что зависимость от произвольного содержимого пути к классу — не очень хорошая идея, если нужны надежные модули. И по этой причине `requires class-path` не существует.

Как лучше всего выразить, что модуль, в котором хранится содержимое пути к классу, не может зависеть от чего-либо? В системе модулей, использующей имена для ссылки на другие модули? Невозможность дать этому модулю имя, сделав его *безымянным*, звучит оправданно.

И вот что мы имеем: у безымянного модуля нет имени, поскольку ни один модуль не должен ссылаться на него в директиве `requires` — или любой другой директиве, если уж на то пошло. Без `requires` не будет ребра читабельности, а без него код в безымянном модуле недоступен для других модулей.

Таким образом, чтобы явный модуль зависел от артефакта, этот артефакт должен находиться в пути модуля. Как уже упоминалось в подразделе 8.1.3, это может означать, что при размещении простых JAR-файлов в пути модулей они превращаются в автоматические модули — концепция, которую мы рассмотрим далее.

8.3. Автоматические модули: простые JAR в пути модуля

Долгосрочная цель любых усилий по модуляризации — обновить простые JAR-файлы до модульных и переместить их из пути классов в путь модуля. Один из способов добиться этого — подождать, пока все зависимости станут модульными, а затем модульно преобразовать проект — так выглядит нисходящий подход. Однако он может занять много времени, поэтому система модулей также допускает модульность по направлению сверху вниз.

Раздел 9.2 объясняет оба подхода в деталях, но для того, чтобы нисходящий подход работал, вам нужен новый ингредиент. Подумайте об этом: как можно объявить модуль, если зависимости представлены в виде простых JAR-файлов? Как вы заметили в подразделе 8.2.3, при размещении их в пути к классам они окажутся в безымянном модуле и другой модуль не сможет получить к ним доступ. Но вы обратили внимание на подраздел 8.1.3, поэтому уже знаете, что простые JAR-файлы также могут находиться в пути модуля, где система модулей автоматически создает модули для них.

ПРИМЕЧАНИЕ

Механизмы, окружающие автоматические модули, обычно применяются во время компиляции и выполнения. Как я уже говорил ранее, постоянное упоминание и того и другого дает мало информации и затрудняет чтение текста.

Для каждого JAR в пути модуля, у которого нет дескриптора, система модулей создает *автоматический модуль*. Как и любой другой именованный, он имеет три основных свойства (см. подраздел 3.1.3):

- *имя* — имя автоматического модуля может быть определено в манифесте JAR с заголовком `Automatic-Module-Name`. Если он отсутствует, то система модулей генерирует имя из имени файла;