

Оглавление

Предисловие	16
Вступление	18
Благодарности	20
О книге	22
Кому стоит прочитать эту книгу.....	22
Структура книги.....	23
Части и главы.....	23
Выберите собственный путь	24
Предостережение.....	26
О коде	26
О версии Java	27
Оформление листингов	28
Оформление имен методов	28
Заместители во фрагментах кода	29
Команды и их вывод.....	29
Дискуссионный форум liveBook.....	29
От издательства	30
Об авторе	31
Об обложке	32

Часть I. Привет, модули

Глава 1. Первый элемент головоломки.....	35
1.1. Что такое модульность.....	36
1.1.1. Визуализация ПО в виде графов	37
1.1.2. Влияние принципов проектирования.....	39
1.1.3. Что такое модульность.....	40
1.2. Уничтожение модулей до Java 9	41
1.3. Трудности до Java 9	44
1.3.1. Невыраженные зависимости между JAR-файлами	45

1.3.2.	Затенение классов с одинаковыми именами.....	46
1.3.3.	Конфликты разных версий на одном проекте.....	48
1.3.4.	Сложная загрузка класса	49
1.3.5.	Слабая инкапсуляция в JAR-файлах	49
1.3.6.	Проверки безопасности должны быть выполнены вручную.....	51
1.3.7.	Низкая производительность при загрузке	51
1.3.8.	Негибкая среда выполнения в Java	52
1.4.	Взгляд на модули с высоты птичьего полета	52
1.4.1.	Все на свете — это модули.....	52
1.4.2.	Ваш первый модуль	55
1.4.3.	Система модулей в действии.....	55
1.4.4.	Ваш немодульный проект в целом будет в порядке.....	60
1.5.	Цели системы модулей	61
1.5.1.	Надежная настройка: не оставлять после себя JAR	62
1.5.2.	Надежная инкапсуляция: делаем внутренний код модуля недоступным.....	62
1.5.3.	Автоматическая безопасность и улучшенное удобство сопровождения	63
1.5.4.	Улучшенная производительность при запуске.....	63
1.5.5.	Масштабируемая платформа Java	63
1.5.6.	Не цели	64
1.6.	Навыки, старые и новые.....	64
1.6.1.	Чему вы научитесь.....	65
1.6.2.	Что нужно знать	65
	Резюме	66
Глава 2.	Структура модульного приложения.....	67
2.1.	Знакомство с ServiceMonitor.....	68
2.2.	Модуляризация ServiceMonitor	72
2.3.	Разделение ServiceMonitor на модули	72
2.4.	Организация файлов в структуре каталогов	73
2.5.	Объявление и описание модулей.....	74
2.5.1.	Объявление зависимостей от других модулей.....	76
2.5.2.	Объявление публичных API модуля.....	76
2.5.3.	Визуализация ServiceMonitor с помощью модульного графа	77
2.6.	Компиляция и упаковка модулей	79
2.7.	Запуск ServiceMonitor.....	81
2.8.	Расширение модульной кодовой базы	81
2.9.	Разбор полетов: эффекты системы модулей.....	82
2.9.1.	Что система модулей делает для вас.....	82
2.9.2.	Что еще модульная система может сделать для вас	84
2.9.3.	Добавление необязательных зависимостей.....	86
	Резюме	86

Глава 3. Определение модулей и их свойств	87
3.1. Модули: строительные блоки модульных приложений	88
3.1.1. Модули Java (JMOD), поставляемые с JDK	88
3.1.2. Модульные файлы JAR: доморощенные модули	89
3.1.3. Декларация модулей: определение свойств модулей	90
3.1.4. Множество типов модулей	95
3.2. Читательность: соединяем части вместе	97
3.2.1. Достижение надежной настройки	98
3.2.2. Эксперименты с ненадежными настройками	99
3.3. Доступность: объявление публичных API	105
3.3.1. Обеспечение надежной инкапсуляции	107
3.3.2. Инкапсуляция переходных зависимостей	109
3.3.3. Столкновения инкапсуляций	110
3.4. Путь модуля: позвольте Java узнать о модулях	114
3.4.1. Регулирование модулей: анализ и верификация структуры приложения ...	115
3.4.2. Граф модуля: представление архитектуры приложения	117
3.4.3. Добавление модулей в граф	120
3.4.4. Добавление ребер в граф	121
3.4.5. Доступность — постоянная цель	122
Резюме	122
Глава 4. Построение модулей от исходного кода до JAR	124
4.1. Структурирование проекта с помощью каталогов	125
4.1.1. Новое предложение — новое соглашение?	125
4.1.2. Созданная структура каталогов	126
4.1.3. Место для деклараций модулей	127
4.2. Компиляция отдельного модуля	128
4.2.1. Компиляция модульного кода	128
4.2.2. Модульное или немодульное?	129
4.3. Компиляция нескольких модулей	131
4.3.1. Безыскусный подход	132
4.3.2. Путь к исходнику модуля: информирование компилятора о структуре проекта	132
4.3.3. Звездочка как токен для имени модуля	133
4.3.4. Несколько путей к исходникам модулей	135
4.3.5. Настройка начального модуля	136
4.3.6. Стоит ли оно того?	137
4.4. Параметры компилятора	138
4.5. Упаковка модульного JAR-файла	139
4.5.1. Быстрый обзор jar	139
4.5.2. Анализ JAR	140

4.5.3. Объявление точки входа	141
4.5.4. Параметры архиватора	141
Резюме	142
Глава 5. Запуск и отладка модульных приложений.....	143
5.1. Запуск JVM с модулями.....	144
5.1.1. Определение основного класса	144
5.1.2. Что, если начальный модуль и основной модуль — не один и тот же.....	145
5.1.3. Передача параметров приложению.....	146
5.2. Загрузка ресурсов из модулей	147
5.2.1. Загрузка ресурсов до Java 9	148
5.2.2. Загрузка ресурсов, начиная с Java 9 и позже	148
5.2.3. Загрузка ресурсов за пределами модуля	150
5.3. Отладка модулей и модульных приложений	151
5.3.1. Анализ отдельных модулей	152
5.3.2. Проверка набора модулей.....	152
5.3.3. Проверка модульного графа	153
5.3.4. Перечисление обозреваемых модулей и зависимостей.....	154
5.3.5. Исключение модулей во время разрешения	156
5.3.6. Наблюдение за системой модулей с помощью ведения журнала	159
5.4. Параметры виртуальной машины Java	162
Резюме	164

Часть II. Адаптация под реальные проекты

Глава 6. Проблемы совместимости при переходе на Java 9 и выше	167
6.1. Работа с модулями JEE	169
6.1.1. Что особенного в модулях JEE?.....	170
6.1.2. Разрешение модулей JEE вручную	171
6.1.3. Вмешиваемся в сторонние реализации модулей JEE.....	172
6.2. Приведение к URLClassLoader	173
6.2.1. Загрузчики классов приложения, тогда и сейчас.....	173
6.2.2. Обходимся без URLClassLoader	175
6.2.3. Поиск проблемных приведений.....	176
6.3. Обновленный макет каталога образов среды выполнения.....	176
6.4. Выбор, замена и расширение платформы.....	179
6.4.1. Компактных профилей больше нет	179
6.4.2. Удален механизм расширения.....	180
6.4.3. Удален механизм переопределения утвержденных стандартов	180
6.4.4. Удалены некоторые параметры пути к загрузочному классу	180

6.4.5. Компиляции для Java 5 больше нет.....	180
6.4.6. Удален выбор версии JRE.....	181
6.5. Мелочи, которые приводят к большим неприятностям	181
6.5.1. Новый формат версий.....	182
6.5.2. Исчезновение инструментов	183
6.5.3. Всякие мелочи	183
6.5.4. Новые устаревшие элементы в Java 9, 10 и 11	184
Резюме	184
Глава 7. Повторяющиеся проблемы при переходе на Java 9 и выше.....	186
7.1. Инкапсуляция внутренних API.....	187
7.1.1. Внутренние API под микроскопом	188
7.1.2. Анализ зависимостей с помощью JDepс.....	192
7.1.3. Компиляция с помощью внутренних API.....	194
7.1.4. Выполнение с помощью внутренних API	195
7.1.5. Параметры компилятора и JVM для доступа к внутренним API	200
7.2. Улучшение разделенных пакетов	202
7.2.1. Что не так с разделенными пакетами.....	203
7.2.2. Эффекты разделенных пакетов.....	204
7.2.3. Множество способов справиться с разделенными пакетами.....	208
7.2.4. Исправление модулей: последнее средство для обработки разделенных пакетов.....	209
7.2.5. Поиск разделенных пакетов с помощью JDepс	210
7.2.6. Примечание о конфликте версий зависимостей	211
Резюме	211
Глава 8. Постепенная модуляризация существующих проектов.....	213
8.1. Почему наращивание модульности необязательно	214
8.1.1. Если бы требовалось, чтобы каждый JAR был модульным	215
8.1.2. Смешивание и соединение простых JAR-файлов с модулями.....	215
8.1.3. Технические основы наращиваемой модульности	217
8.2. Безымянный модуль, он же путь к классу	218
8.2.1. Хаос пути к классам, захваченного безымянным модулем	220
8.2.2. Разрешение модулей для безымянного модуля	221
8.2.3. Зависимость от безымянного модуля	223
8.3. Автоматические модули: простые JAR в пути модуля.....	225
8.3.1. Имена автоматических модулей: мелочь, имеющая большое значение	227
8.3.2. Разрешение модулей для автоматических модулей.....	230
8.3.3. Все в автоматических модулях?	238
8.3.4. Зависимость от автоматических модулей	239
Резюме	242

Глава 9. Стратегии миграции и модуляризации	243
9.1. Стратегии миграции	244
9.1.1. Подготовительные обновления	244
9.1.2. Оценка усилий	245
9.1.3. Непрерывная интеграция в Java 9+	246
9.1.4. Мысли о параметрах командной строки	250
9.2. Стратегии модуляризации	253
9.2.1. Восходящая модуляризация: когда все зависимости — модульные	255
9.2.2. Нисходящая модуляризация: когда нет времени дожидаться всех зависимостей	256
9.2.3. Модуляризация изнутри наружу: если проект находится в середине стека	257
9.2.4. Применение данных стратегий к проекту	258
9.3. Делаем JAR-файлы модульными	259
9.3.1. Открытые модули как промежуточный шаг	259
9.3.2. Генерация модульных деклараций с помощью JDepс	260
9.3.3. Взлом сторонних JAR-файлов	263
9.3.4. Публикация модульных JAR-файлов для Java 8 и выше	265
Резюме	267

Часть III. Расширенные функции системы модулей

Глава 10. Использование сервисов для разделения модулей	271
10.1. Изучение потребности в сервисах	272
10.2. Сервисы в модульной системе платформы Java	274
10.2.1. Использование, предоставление и потребление услуг	274
10.2.2. Разрешение модулей для сервисов	279
10.3. Эффективное проектирование сервисов	282
10.3.1. Типы, которые могут стать сервисами	283
10.3.2. Использование фабрик в качестве сервисов	283
10.3.3. Изоляция потребителей от глобального состояния	285
10.3.4. Организация сервисов, потребителей и поставщиков в модули	288
10.3.5. Использование сервисов для разбиения циклических зависимостей	289
10.3.6. Объявление сервисов в разных версиях Java	291
10.4. Доступ к сервисам с помощью ServiceLoader API	294
10.4.1. Загрузка и доступ к сервисам	294
10.4.2. Особенности загрузки сервисов	296
Резюме	297
Глава 11. Уточнение API и зависимостей	300
11.1. Подразумеваемая читабельность: передача зависимостей	301
11.1.1. Выявление зависимостей модуля	302

11.1.2. Модификатор <code>transitive</code> : подразумеваемая читабельность зависимости.....	304
11.1.3. Когда использовать подразумеваемую читабельность.....	306
11.1.4. Когда стоит полагаться на подразумеваемую читабельность	307
11.1.5. Рефакторинг модулей с подразумеваемой читабельностью.....	309
11.1.6. Рефакторинг модулей путем их слияния	313
11.2. Необязательные зависимости.....	314
11.2.1. Загадка надежной конфигурации	315
11.2.2. Модификатор <code>static</code> : маркировка зависимостей как необязательных	316
11.2.3. Разрешение модулей для необязательных зависимостей	318
11.2.4. Написание кода с необязательной зависимостью.....	319
11.3. Квалифицированный экспорт: ограничение доступа к конкретным модулям	321
11.3.1. Предоставление внутренних API	322
11.3.2. Экспорт пакетов в модули.....	323
11.3.3. Когда использовать квалифицированный экспорт.....	325
11.3.4. Экспорт пакетов в командной строке	327
Резюме	327
Глава 12. Рефлексия в модульном мире.....	329
12.1. Почему директивы <code>exports</code> не подходят для рефлексии	331
12.1.1. Врываемся в немодульный код.....	332
12.1.2. Принудительная публикация внутренних типов	332
12.1.3. Квалифицированный экспорт создает связь с определенными модулями	333
12.1.4. Нет поддержки глубокой рефлексии	333
12.2. Открытые пакеты и модули: предназначены для использования с рефлексией	334
12.2.1. Открытие пакетов для доступа в режиме выполнения	335
12.2.2. Открытие пакетов для определенных модулей.....	336
12.2.3. Экспорт и открытие пакетов	337
12.2.4. Открытие модулей: закрытие рефлексии	338
12.3. Рефлексия модулей.....	339
12.3.1. Обновление рефлексивного кода для модулей (или нет)	340
12.3.2. Использование обработчика переменных вместо рефлексии.....	342
12.3.3. Анализ свойств модуля с помощью рефлексии.....	344
12.3.4. Изменение свойств модуля с помощью рефлексии.....	347
12.3.5. Пересылка открытых пакетов.....	348
12.4. Динамическое создание диаграмм модулей со слоями.....	349
12.4.1. Что такое слои.....	350
12.4.2. Анализ слоев	352
12.4.3. Создание модульных слоев	355
Резюме	359

Глава 13. Версии и модули: возможное и невозможное	362
13.1. Отсутствие поддержки версий в JPMS	363
13.1.1. Нет поддержки нескольких версий.....	363
13.1.2. Нет поддержки выбора версии.....	366
13.1.3. Что может принести будущее.....	368
13.2. Запись информации о версии	369
13.2.1. Запись версий при сборке модулей	369
13.2.2. Доступ к версиям модуля	370
13.3. Запуск нескольких версий модуля в отдельных слоях.....	372
13.3.1. Зачем нужен стартер для раскрутки дополнительных слоев.....	373
13.3.2. Раскручивание слоев для вашего приложения, Apache Twill и Cassandra Java Driver	374
Резюме	377
Глава 14. Настройка образа среды выполнения с помощью jlink	378
14.1. Создание пользовательских образов среды выполнения	379
14.1.1. Начало работы с jlink	380
14.1.2. Содержание и структура образа	381
14.1.3. Включение сервисов в образы среды выполнения	382
14.1.4. Правильно подобранные образы с помощью jlink и jdeps	385
14.2. Создание автономных образов приложений	387
14.2.1. Включение модулей приложений в образы	388
14.2.2. Создание собственного средства запуска приложения	391
14.2.3. Безопасность, производительность и стабильность.....	393
14.3. Генерация образов в операционных системах	393
14.4. Использование плагинов jlink для оптимизации образов	395
14.4.1. Плагины для jlink	395
14.4.2. Уменьшение размера образа	398
14.4.3. Улучшение производительности во время выполнения	402
14.5. Параметры для jlink.....	403
Резюме	404
Глава 15. Собираем все вместе	406
15.1. Добавляем навороты в ServiceMonitor.....	406
15.1.1. Разнообразные зависимости	410
15.1.2. Снижение видимости	411
15.1.3. Отделение от сервисов	411
15.1.4. Загрузка кода со слоями во время выполнения	412
15.1.5. Обработка зависимостей от простых JAR-файлов	412
15.2. Советы для разработчиков модульных приложений	413
15.2.1. Модуль или нет?	413
15.2.2. Идеальный модуль	414

15.2.3. Позаботьтесь о декларациях модуля	419
15.2.4. Взлом кода путем редактирования модульных деклараций	422
15.3. Технологический ландшафт	424
15.3.1. Maven, Gradle и другие инструменты сборки	425
15.3.2. OSGi.....	427
15.3.3. Микросервисы.....	431
15.4. Размышления о модульной экосистеме.....	433
Резюме	434

Приложения

Приложение А. Резюмируем путь класса.....	436
А.1. Использование пути к классам для загрузки JAR-файлов приложений	436
А.2. Путь к классам, начиная с Java 9	437
Приложение Б. Обзор API рефлексии	439
Б.1. Основные типы и методы	441
Б.2. Взлом API с помощью setAccessible	442
Б.3. Аннотации помечают код для рефлексии	442
Приложение В. Наблюдение за JVM с унифицированным ведением журнала.....	443
В.1. Что такое унифицированное ведение журнала	444
В.2. Определяем, какие сообщения должны отображаться.....	444
В.3. Определяем, куда вывести сообщения	447
В.4. Определяем, что выводить в сообщениях.....	447
В.5. Настройка всего конвейера регистрации	448
Приложение Г. Анализ зависимостей проекта с помощью JDeps	450
Г.1. Знакомство с JDeps	451
Г.2. Добавление зависимостей в анализ.....	452
Г.3. Настройка вывода JDeps	453
Г.4. Детализация в зависимости от вашего проекта	454
Г.5. JDeps понимает модули	456
Приложение Д. Ориентация на несколько версий Java с мультиверсионными JAR-файлами	458
Д.1. Создание мультиверсионного JAR.....	459
Д.2. Внутренняя работа MB-JAR	461
Д.3. Рекомендации по использованию	461
Д.3.1. Организация исходного кода.....	461
Д.3.2. Организация байт-кода	462
Д.3.3. Когда использовать MB-JAR.....	462

Постепенная модуляризация существующих проектов

В этой главе

- Работа с безымянными модулями.
- Помощь в модуляризации с помощью автоматических модулей.
- Нарастиваемая 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 получают 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.

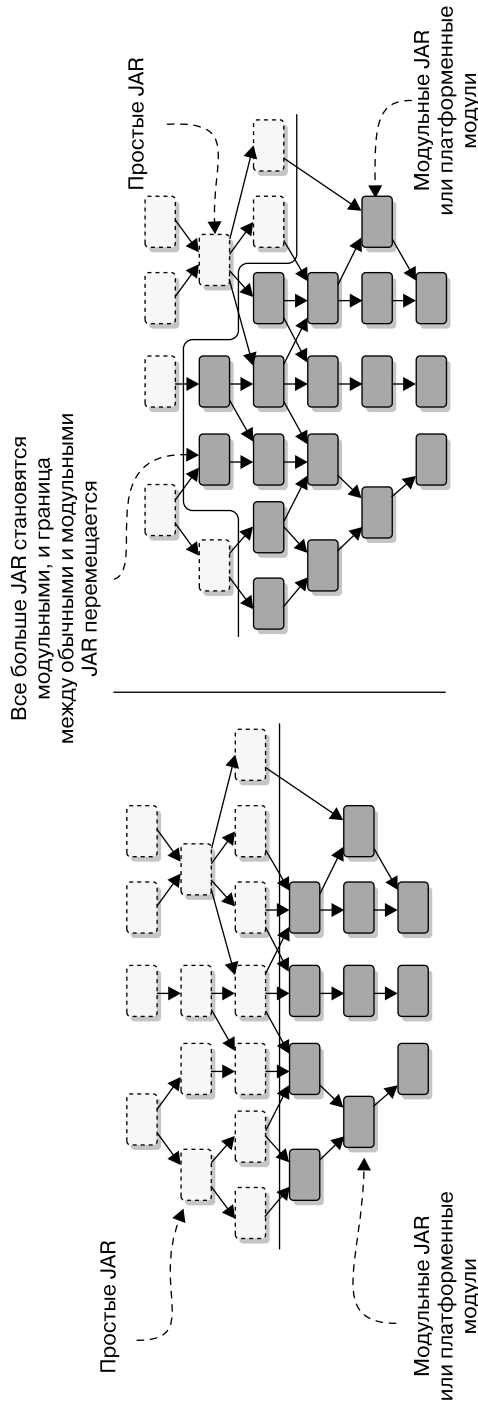


Рис. 8.1. Система модулей позволяет выполнять немодульный код на модульном JDK (слева). Что еще более важно, она предоставляет инструменты для перемещения этой границы (справа)

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 или в 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*.)

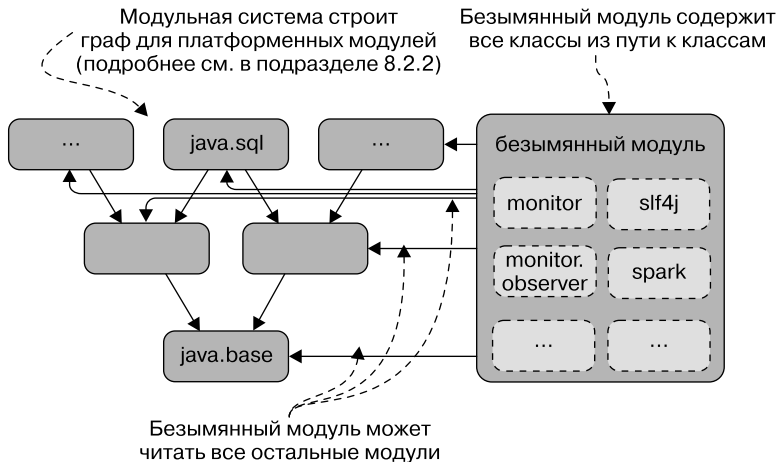


Рис. 8.2. Запущенная со всеми JAR-файлами приложения в пути к классам, система модулей строит граф из платформенных модулей (слева) и распределяет все классы из пути к классам к безымянному модулю (справа), который может читать все другие модули

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 `libs` 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, однодольных 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?

Напомним 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.

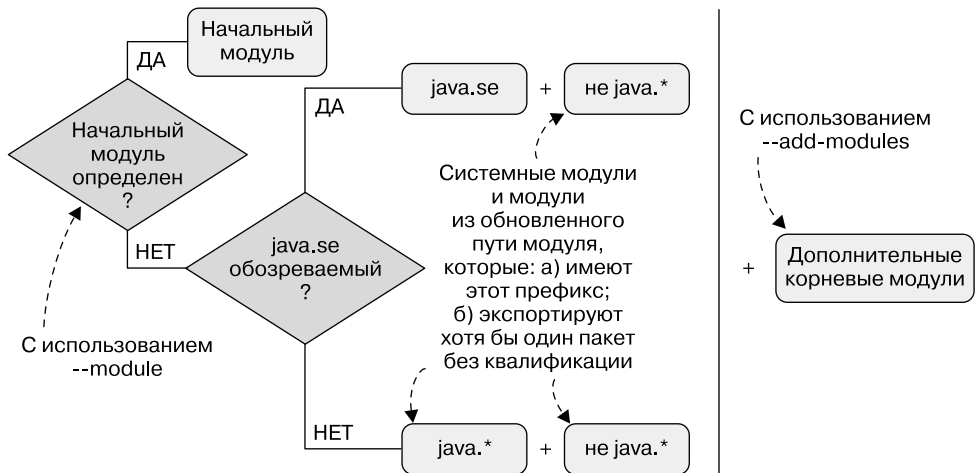


Рис. 8.3. Какие модули становятся корневыми для разрешения модулей (см. подраздел 3.4.1), зависит от того, был ли начальный модуль определен с помощью `--module` (если нет, то безымянный модуль становится исходным), и от того, является ли *java.se* обозреваемым. В любом случае модули, определенные с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

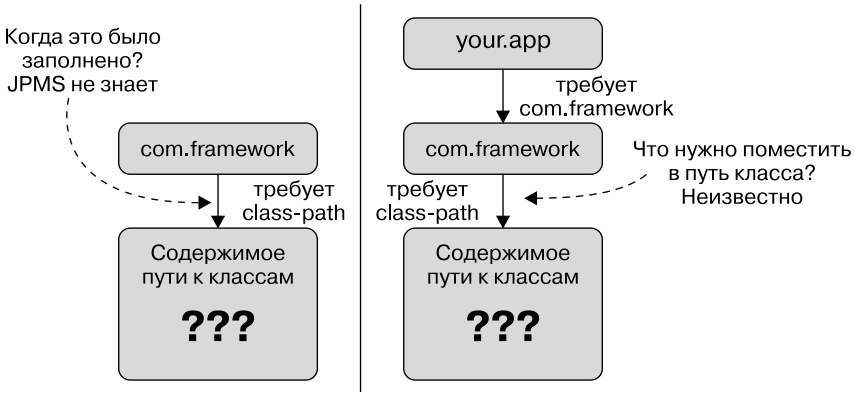


Рис. 8.4. Если `com.framework` зависит от некоего содержимого пути к классу с гипотетическим `requires class-path`, то система модулей не может определить, было ли выполнено это требование (слева). Создав свое приложение на данной платформе, вы не будете знать, как заполнить эту зависимость (справа)

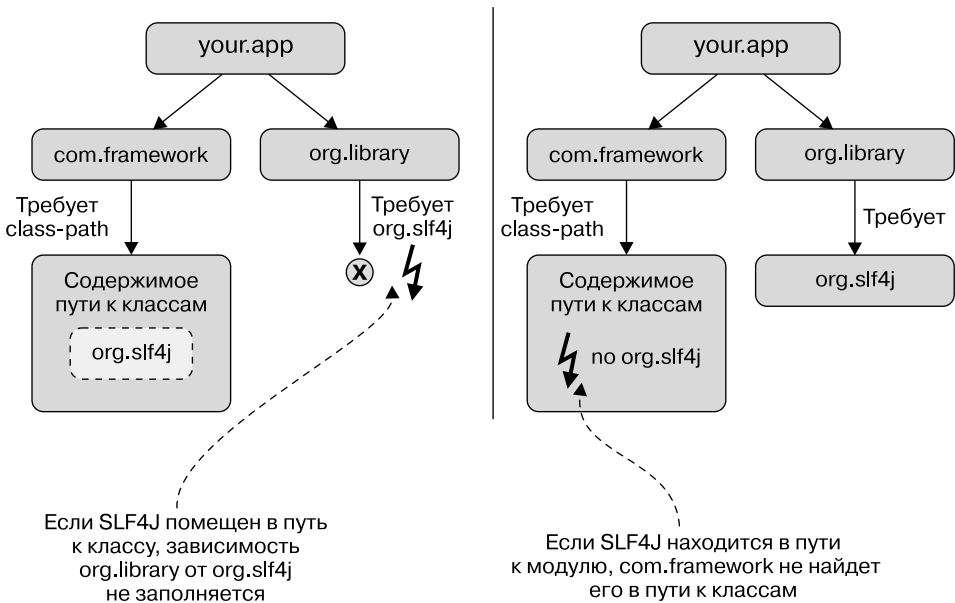


Рис. 8.5. Если `com.framework` зависит от SLF4J с гипотетическим `requires class-path` и `org.library` запрашивает его как модуль с `requires org.slf4j`, то не существовало бы способа удовлетворить оба требования. Независимо от того, был ли SLF4J размещен в пути к классам (слева) или в пути к модулям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