

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Java 9

模块化开发

核心原则与实践

Java 9 Modularity: Patterns and Practices for Developing
Maintainable Applications

Sander Mak Paul Bakker 著

王净 等译



Java 9 模块化开发：核心原则与实践

Sander Mak
Paul Bakker 著
王净 等译

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

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社



图书在版编目 (CIP) 数据

Java 9 模块化开发: 核心原则与实践 / (荷) 桑德·马克 (Sander Mak) 等著; 王净等译. —北京: 机械工业出版社, 2018.5

(O'Reilly 精品图书系列)

书名原文: Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications

ISBN 978-7-111-60129-6

I. J… II. ①桑… ②王… III. JAVA 语言—程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 110247 号

北京市版权局著作权合同登记

图字: 01-2017-8670 号

© 2017 O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2018. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2017。

简体中文版由机械工业出版社出版 2018。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

封底无防伪标均为盗版

本书法律顾问

北京大成律师事务所 韩光 / 邹晓东

书 名 / Java 9 模块化开发: 核心原则与实践

书 号 / ISBN 978-7-111-60129-6

责任编辑 / 余 洁

封面设计 / Karen Montgomery, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)

印 刷 / 北京诚信伟业印刷有限公司

开 本 / 178 毫米 × 233 毫米 16 开本 15.75 印张

版 次 / 2018 年 6 月第 1 版 2018 年 6 月第 1 次印刷

定 价 / 69.00 元 (册)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010)88379426; 88361066

购书热线: (010)68326294; 88379649; 68995259

投稿热线: (010)88379604

读者信箱: hzit@hzbook.com



O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal



译者序

JDK 9 是 Java 开发工具包的第 9 个主要版本，于 2017 年 7 月下旬发布，它带来了许多令人兴奋的新功能。Java 9 定义了一套全新的模块系统。当代码库越来越大，创建盘根错节的“意大利面条式代码”的概率呈指数级增长，这时候就得面对两个基础问题。首先，很难真正地对代码进行封装，而系统对不同部分（也就是 JAR 文件）之间的依赖关系并没有明确的概念。每一个公共类都可以被类路径之下任何其他公共类所访问，这样就会导致无意中使用了并不想被公开访问的 API。其次，类路径本身也存在问题：你怎么知晓所有需要的 JAR 都已经有了，或者是不是会有重复的项呢？模块系统把这两个问题都解决了。

模块化的 JAR 文件都包含一个额外的模块描述符。在这个模块描述符中，对其他模块的依赖是通过 `requires` 来表示的。另外，`exports` 语句控制着哪些包是可以被其他模块访问的。所有不被导出的包默认都封装在模块里。

本书共分为三部分，第一部分包括 6 章。第 1 章主要介绍了什么是模块化以及 Java 9 模块的主要特点。第 2 章学习了如何定义模块，以及使用哪些概念管理模块之间的交互。第 3 章在第 2 章的基础上通过构建自己的模块进一步学习相关模块概念。第 4 章讨论了可以解耦模块的服务。第 5 章和第 6 章探讨了模块化模式，以及如何以最大限度地提高可维护性和可扩展性的方式使用模块。

第二部分包括 4 章。第 7 章和第 8 章重点介绍了如何将现有的代码迁移到模块中。第 9 章通过迁移案例详细讨论了如何实现迁移。如果你是一名库的创建者或者维护者，那么第 10 章将对你有所帮助，其介绍了如何向库添加模块支持。

第三部分也包括 4 章，主要介绍了一些模块化开发工具。第 11 章学习了主要的 IDE 以及构建工具。第 12 章介绍了如何对模块进行测试。第 13 章和第 14 章主要介绍了自定义运行时映像以及对模块化未来的展望。

本书图文并茂、技术新、实用性强，以大量的实例对 Java 9 模块系统做了详细的解释，



是学习 Java 9 的读者不可缺少的实用参考书籍。本书可作为 Java 编程人员的参考手册，适合计算机技术人员使用。此外，书中还提供了相关参考资料，如果在阅读过程中遇到不明白的方法或属性，可以参阅相关内容。

参与本书翻译的人有王净、田洪、范园芳、范楨、胡训强、晏峰、余佳隽、张洁、何远燕、任方燕。最终由王净负责统稿。在此，要感谢我们的家人，他们总是无怨无悔地支持我们的一切工作。

在翻译过程中，我们尽量保持原书的特色，并对书中出现的术语和难词难句进行了仔细推敲和研究。但毕竟有少量技术是译者在自己的研究领域中所不曾遇到过的，所以疏漏和争议之处在所难免，望广大读者提出宝贵意见。

最后，希望广大读者能多花些时间细细品味这本凝聚作者和译者大量心血的书籍，为将来的职业生涯奠定良好的基础。

王净

2018年3月于广州



序

什么是 Java 中的模块化？对于一些人来说，模块化是一个开发原则，即对接口进行编程并隐藏实现的细节，这就是所谓的封装学派 (*school of encapsulation*)。对于另外一些人来说，模块化是指依赖类加载器来提供动态执行环境，这就是所谓的隔离学派 (*school of isolation*)。还有一些人认为模块化指的是工件、存储库以及相关工具，这就是所谓的配置学派 (*school of configuration*)。虽然单独来看，这些观点都是正确的，但它们都太片面，感觉更像是一个不太清晰的“大故事”的几个片段。如果开发人员知道其部分代码仅供内部使用，那么他们为什么不能像隐藏类或字段一样容易地隐藏包呢？如果代码只能在依赖项存在的情况下编译和运行，那么这些依赖项为什么不能顺畅地从编译过程流向打包过程，再到安装过程，最后到执行过程呢？如果工具只有在提供了原始自描述工件时才能起作用，那么如何重用那些只是普通 JAR 文件的旧版本库呢？

Java 9 将模块作为 Java 平台的高级功能，从而很自然地引入了模块化概念。模块是一组用于重用的包，这个简单的概念对代码开发、部署以及运行的方式产生了非常深刻的影响。一旦将包放置到模块中，Java 中用来促进和控制“重用”的长期存在的机制（接口、访问控制、JAR 文件、类加载器以及动态链接）就能更好地工作。

首先，模块以其他机制无法实现的方式阐明了程序的结构。许多开发人员会惊讶地发现他们的代码结构并没有想象得那么好。例如，由多个 JAR 文件组成的代码库可以实现不同 JAR 文件中类之间的循环，但在不同模块中的类之间的循环却是禁止的。实现代码库模块化的动机之一是一旦实现了模块化，就可以避免出现因为循环依赖所产生的“泥球” (ball of mud)。[⊖]此外，使用模块进行开发还可以实现通过服务进行编程，从而减少耦合并进一步提高抽象性。

其次，模块产生了其他机制无法实现的代码责任感。从模块中导出包的开发人员实际上对 API 的稳定性做出了承诺，甚至模块本身的名称也是 API 的一部分。如果开发人员将

⊖ 泥球是指一个随意化的杂乱的结构化系统，只是代码的堆砌和拼凑，往往会导致很多错误或者缺陷。——译者注



太多的功能捆绑到单个模块中，那么就会导致该模块牵扯到大量与任何单一任务无关的依赖项；任何重用该模块的人都会意识到其杂乱无序的性质，即使模块的内部是隐藏的。使用模块进行开发可以促使每个开发人员思考其代码的稳定性和内聚性。

大多数人对桌布戏法都非常熟悉，即将桌布从桌子上迅速拿走，同时不能打翻盘子和杯子。对于那些使用 Java 9 的人来说，设计一个可以嵌入 Java 虚拟机（Java 虚拟机由自 20 世纪 90 年代以来所开发的数以百万计的类所控制）的模块系统感觉就像是反向表演桌布戏法。事实证明，模块化 JDK 导致了戏法的失败，因为一些知名的库为了自身的发展而不愿意将模块系统应用于 JDK 模块所带来的封装。Java 9 设计中的这种矛盾很难在学术上得到解决。最终，来自社区的长期反馈促使模块系统为开发人员提供了各种各样的“杠杆”和“调节盘”，使得模块化平台代码可以享受真正强大的封装，而模块化应用程序代码可以享受“足够强大”的封装。随着时间的推移，我们认为在模块化 JDK 方面进行的大胆选择将会使得代码更加可靠。

只有当一个模块系统适用于所有人时，该系统才是最好的。今天创建模块的开发人员越多，明天就会有更多的开发人员创建模块。但是那些尚未创建自己模块的开发人员又该怎么办呢？毫不夸张地说，Java 9 会像关注模块内的代码一样关注模块外的代码。代码库的作者是唯一应该对代码库进行模块化的开发人员，在完成模块化之前，模块系统必须为模块中的代码提供一种方法来“接触”模块之外的代码，而这也导致了自动模块的设计，本书将会详细介绍这部分内容。

Sander 和 Paul 都是 Java 方面的专家，同时也是 Java 9 生态系统可信任的指导者。他们身处 Java 9 开发的最前沿，是迁移流行开源库的先驱。本书面向那些对 Java 中模块化的核心原则和最佳实践感兴趣的人的，包括希望创建可维护组件的应用程序开发人员，寻求关于迁移和反射建议的库开发人员，以及希望利用模块系统高级功能的框架开发人员。我希望本书可以帮助你创建出程序结构经得起时间考验的 Java 程序。

Alex Buckley

Oracle Java 平台组

圣克拉拉，2017 年 7 月



目录

前言 1

第一部分 Java 模块系统介绍

第 1 章 模块化概述 8

1.1 什么是模块化 9

1.2 在 Java 9 之前 10

 1.2.1 将 JAR 作为模块? 11

 1.2.2 类路径地狱 13

1.3 Java 9 模块 14

第 2 章 模块和模块化 JDK 18

2.1 模块化 JDK 19

2.2 模块描述符 22

2.3 可读性 23

2.4 可访问性 24

2.5 隐式可读性 25

2.6 限制导出 29

2.7 模块解析和模块路径 29

2.8 在不使用模块的情况下使用模块化 JDK 31

第 3 章 使用模块 33

3.1 第一个模块 33

 3.1.1 剖析模块 33

 3.1.2 命名模块 35



3.1.3 编译	36
3.1.4 打包	37
3.1.5 运行模块	37
3.1.6 模块路径	39
3.1.7 链接模块	40
3.2 任何模块都不是一座孤岛	41
3.2.1 EasyText 示例介绍	41
3.2.2 两个模块	43
3.3 使用平台模块	46
3.3.1 找到正确的平台模块	46
3.3.2 创建 GUI 模块	47
3.4 封装的限制	51
第 4 章 服务	54
4.1 工厂模式	54
4.2 用于实现隐藏的服务	57
4.2.1 提供服务	57
4.2.2 消费服务	59
4.2.3 服务生命周期	61
4.2.4 服务提供者方法	62
4.3 工厂模式回顾	64
4.4 默认服务实现	65
4.5 服务实现的选择	66
4.6 具有服务绑定的模块解析	68
4.7 服务和链接	70
第 5 章 模块化模式	73
5.1 确定模块边界	74
5.2 精益化模块	76
5.3 API 模块	76
5.3.1 API 模块中应该包含什么	77
5.3.2 隐式可读性	78
5.3.3 带有默认实现的 API 模块	81
5.4 聚合器模块	82



5.4.1 在模块上构建一个外观	83
5.4.2 安全拆分模块	84
5.5 避免循环依赖	86
5.5.1 拆分包	86
5.5.2 打破循环	87
5.6 可选的依赖关系	90
5.6.1 编译时依赖关系	91
5.6.2 使用服务实现可选依赖关系	95
5.7 版本化模块	96
5.8 资源封装	99
5.8.1 从模块加载资源	100
5.8.2 跨模块加载资源	101
5.8.3 使用 ResourceBundle 类	102
第 6 章 高级模块化模式	104
6.1 重温强封装性	104
6.1.1 深度反射	105
6.1.2 开放式模块和包	106
6.1.3 依赖注入	109
6.2 对模块的反射	111
6.2.1 内省	112
6.2.2 修改模块	113
6.2.3 注释	114
6.3 容器应用程序模式	115
6.3.1 层和配置	116
6.3.2 层中的类加载	119
6.3.3 插件体系结构	122
6.3.4 容器体系结构	127
6.3.5 解析容器中的平台模块	132
第二部分 迁移	
第 7 章 没有模块的迁移	134
7.1 类路径已经“死”了?	135



7.2 库、强封装和 JDK 9 类路径	135
7.3 编译和封装的 API	138
7.4 删除的类型	141
7.5 使用 JAXB 和其他 Java EE API	142
7.6 jdk.unsigned 模块	145
7.7 其他更改	146
第 8 章 迁移到模块	148
8.1 迁移策略	148
8.2 一个简单示例	149
8.3 混合类路径和模块路径	150
8.4 自动模块	152
8.5 开放式包	155
8.6 开放式模块	157
8.7 破坏封装的 VM 参数	158
8.8 自动模块和类路径	158
8.9 使用 jdeps	161
8.10 动态加载代码	164
8.11 拆分包	166
第 9 章 迁移案例研究：Spring 和 Hibernate	167
9.1 熟悉应用程序	167
9.2 使用 Java 9 在类路径上运行	172
9.3 设置模块	173
9.4 使用自动模块	174
9.5 Java 平台依赖项和自动模块	176
9.6 开放用于反射的包	176
9.7 解决非法访问问题	177
9.8 重构到多个模块	178
第 10 章 库迁移	180
10.1 模块化之前	181
10.2 选择库模块名称	181
10.3 创建模块描述符	184



10.4 使用模块描述符更新库.....	186
10.5 针对较旧的 Java 版本.....	187
10.6 库模块依赖关系.....	188
10.6.1 内部依赖关系.....	188
10.6.2 外部依赖关系.....	191
10.7 针对多个 Java 版本.....	192
10.7.1 多版本 JAR.....	192
10.7.2 模块化多版本 JAR.....	195

第三部分 模块化开发工具

第 11 章 构建工具和 IDE.....198

11.1 Apache Maven.....	198
11.1.1 多模块项目.....	200
11.1.2 使用 Apache Maven 创建 EasyText 示例.....	200
11.1.3 使用 Apache Maven 运行模块化的应用程序.....	204
11.2 Gradle.....	205
11.3 IDE.....	205

第 12 章 测试模块.....207

12.1 黑盒测试.....	208
12.2 使用 JUnit 进行黑盒测试.....	210
12.3 白盒测试.....	212
12.4 测试工具.....	216

第 13 章 使用自定义运行时映像进行缩减.....217

13.1 静态链接和动态链接.....	218
13.2 使用 jlink.....	219
13.3 查找正确的服务提供者模块.....	223
13.4 链接期间的模块解析.....	223
13.5 基于类路径应用程序的 jlink.....	224
13.6 压缩大小.....	225
13.7 提升性能.....	227
13.8 跨目标运行时映像.....	228

第 14 章 模块化的未来	229
14.1 OSGi	230
14.2 Java EE	232
14.3 微服务	232
14.4 下一步	233

前言

Java 9 向 Java 平台引入了模块系统，这是一个重大的飞跃，标志着 Java 平台上模块化软件开发的一个新时代的开始。看到这些变化让人感到非常兴奋，希望读者看完本书后也会感到兴奋。在深入了解模块系统之前需要做好充分利用该系统的准备。

本书读者

本书为那些想要提高应用程序的设计和结构的 Java 开发者而编写。Java 模块系统改进了设计和构建 Java 应用程序的方法。即使你不打算马上使用模块，了解 JDK 模块化本身也是非常重要的一步。在熟悉了本书第一部分所介绍的模块之后，希望你能真正理解后续关于迁移的相关章节。

将现有代码移至 Java 9 和模块系统将成为一项越来越常见的任务。

本书绝不是对 Java 的一般性介绍。我们假设你拥有在一个团队中编写过较大 Java 应用程序的经验，在较大的 Java 应用程序中模块变得越来越重要。作为一名经验丰富的 Java 开发人员，应该认识到类路径所带来的问题，从而有助于理解模块系统及其功能。

除了模块系统之外，Java 9 中还有许多其他变化。然而，本书主要关注模块系统及其相关功能。当然，在适当的情况下，在模块系统的上下文中也会讨论其他 Java 9 功能。

编写本书的原因

很多读者从 Java 早期开始就是 Java 用户，当时 Applet 还非常流行。多年来，我们使用和喜欢过许多其他平台和语言，但 Java 仍然是主要工具。在构建可维护的软件方面，模块化是一个关键原则。多年来人们花费了大量精力来构建模块化软件，并逐渐热衷于开发模块化应用程序。曾经广泛使用诸如 OSGi 之类的技术来实现模块化，但 Java

平台本身并不支持这些技术。此外，还可通过 Java 之外的其他工具学习模块化，比如 JavaScript 的模块系统。当 Java 9 推出了期待已久的模块系统时，我们认为并不能只是使用该功能，还应该帮助其刚入职的开发人员了解模块系统。

也许在过去 10 年的某个时候你曾经听说过 Jigsaw 项目。经过多年的发展，Jigsaw 项目具备了 Java 模块系统许多功能的原型。Java 的模块系统发展断断续续。Java 7 和 Java 8 最初计划包含 Jigsaw 项目的发展结果。

随着 Java 9 的出现，长期的模块化尝试最终完成了正式模块系统的实现。多年来，各种模块系统原型的范围和功能发生了许多变化。即使你一直在密切关注该过程，也很难弄清楚最终 Java 9 模块系统真正包含什么。本书将会给出模块系统的明确概述，更重要的是将介绍模块系统能够为应用程序的设计和架构做些什么。

本书内容

本书共分为三个部分：

- 1) Java 模块系统介绍。
- 2) 迁移。
- 3) 模块化开发工具。

第一部分主要介绍如何使用模块系统。首先从介绍模块化 JDK 本身开始，然后学习创建自己的模块，随后讨论可以解耦模块的服务，最后探讨模块化模式以及如何以最大限度地提高可维护性和可扩展性的方式使用模块。

第二部分主要介绍迁移。有可能读者现在所拥有的 Java 代码不是使用专为模块系统而设计的 Java 库。该部分介绍如何将现有代码迁移到模块中，以及如何使用尚未模块化的现有库。如果你是一名库的编写者或者维护者，那么这部分中有一章专门介绍了如何向库添加模块支持。

第三部分（也是最后一部分）主要介绍工具。该部分介绍了 IDE 的现状以及构建工具。此外还会学习如何测试模块，因为模块给（单元）测试带来了一些新的挑战，也带来了机会。最后学习链接（linking）——模块系统另一个引人注目的功能。

虽然建议从头到尾按顺序阅读本书，但是请记住并不是所有的读者都必须这样阅读。建议至少详细阅读前四章，从而具备基本知识，以便更好地阅读本书的其他章节。如果时间有限并且有现有的代码需要迁移，那么可以在阅读完前四章后跳到本书的第二部分。一旦完成了迁移，就可以回到“更高级”的章节。

使用代码示例

本书包含了许多代码示例。所有代码示例都可以在 GitHub (<https://github.com/java9-modularity/examples>) 上找到。在该存储库中，代码示例是按照章节组织的。在本书中，使用下面的方法引用具体的代码示例：➡ `chapter3/helloworld`，其含义是可以在“<https://github.com/java9-modularity/examples/chapter3/helloworld>”中找到示例。

强烈建议在阅读本书时使用相关的代码，因为在代码编辑器中可以更好地阅读较长的代码段。此外还建议亲自动手改写代码，如重现书中所讨论的错误。动手实践胜过读书。

排版约定

下面列出的是书中所使用的字体约定：

斜体 (*Italic*)

表示新术语、URL、电子邮件地址、文件名以及文件扩展名。

等宽字体 (`Constant width`)

用于程序清单，以及在段落中引用程序元素，如变量或函数名称、数据库、数据类型、环境变量、语句和关键字。

等宽粗体 (`Constant width bold`)

显示应由用户逐字输入的命令或其他文本。

等宽斜体 (*Constant width italic*)

显示应该由用户提供的值或由上下文确定的值所替换的文本。



该图标表示一般注释。



该图标表示提示或者建议。



该图标表示警告或者提醒。

Safari 在线电子书

Safari (前身为 Safari Books Online) 是一个基于会员制的为企业、政府、教育工作者和个人提供培训和参考的平台。

会员可以访问来自 250 家出版商的书籍、培训视频、学习路径、交互式教程和精心策划的播放列表, 包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、AdobePress、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett, 以及 Course Technology, 等等。

更多信息, 请访问 <http://oreilly.com/safari>。

如何联系我们

对于本书, 如果有任何意见或疑问, 请按照以下地址联系本书出版商。

美国:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国:

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询 (北京) 有限公司

要询问技术问题或对本书提出建议, 请发送电子邮件至:

bookquestions@oreilly.com

要获得更多关于我们的书籍、会议、资源中心和 O'Reilly 网络的信息, 请参见我们的网站:

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

我们在 Facebook 上的主页: <http://facebook.com/oreilly>

我们在 Twitter 上的主页: <http://twitter.com/oreillymedia>

我们在 YouTube 上的主页: <http://www.youtube.com/oreillymedia>

致谢

编写本书的想法来源于 2015 年在 JavaOne 会议上与来自 O'Reilly 的 Brian Foster 的一次谈话，非常感谢你委托我们参与这个项目。从那时起，很多人对本书的编写提供了帮助。

感谢 Alex Buckley、Alan Bateman 和 Simon Maple 所给出的重要技术评论和对本书所提出的许多改进意见。此外，还要感谢 O'Reilly 的编辑团队，Nan Barber 和 Heather Scherer 考虑到了所有的组织细节。

如果没有妻子 Suzanne 的坚定支持，编写本书是不可能的。多少个夜晚和周末，我都无法陪伴妻子和三个孩子。感谢你一直陪我到最后！此外，还要感谢 Luminis (<http://luminis.eu/>) 为编写本书所提供的支持。我很高兴能成为公司的一员，我们的口号是“知识是共享的唯一财富”。

Sander Mak

我也要感谢妻子 Qiushi，在我编写这第二本书籍时始终支持我，即使在我们搬到世界的另一个位置的时候。此外，还要感谢 Netflix (<http://netflix.com/>) 和 Luminis (<http://luminis.eu/>)，感谢它们给予我编写本书的时间和机会。

Paul Bakker

本书第 1 章、第 7 章、第 13 章和第 14 章的漫画由 Oliver Widder (<http://geek-and-poke.com/>) 创建，并获得 Creative Commons Attribution 3.0 Unported (CC BY 3.0) (http://creativecommons.org/licenses/by/3.0/deed.en_US) 的许可。本书的作者将漫画改为横向和灰色。

模块化概述

你是否曾经因为困惑而不停地挠头，并问自己：“代码为什么在这个位置？它们如何与其他庞大的代码块相关联？我该从哪里开始呢？”或者在看完应用程序代码所捆绑的大量 Java 归档文件（Java Archive, JAR）后，目光是否会呆滞？答案是肯定的。

构建大型代码库是一项被低估的技术。这既不是一个新问题，也不是 Java 所特有的。然而，Java 一直是构建大型应用程序的主流语言之一，且通常会大量使用 Java 生态系统中的许多库。在这种情况下，系统可能会超出我们理解和有效开发的能力范围。经验表明，从长期来看，缺乏结构性所付出的代价是非常高的。

模块化是用来管理和减少这种复杂性的技术之一。Java 9 引入了一个新的模块系统，从而可以更容易地创建和访问模块。该系统建立在 Java 已用于模块化开发的抽象基础之上。从某种意义上讲，它促使了现有的大型 Java 开发最佳实践成为 Java 语言的一部分。

Java 模块系统对 Java 开发产生了深刻的影响。它代表了模块化成为整个 Java 平台高级功能这一根本转变，从根本上解决了模块化的问题，对语言、JVM（Java Virtual Machine, Java 虚拟机）以及标准库都进行了更改。虽然完成这些更改付出了巨大的努力，但却并不像 Java 8 中添加流和 Lambda 表达式那样“华而不实”。诸如 Lambda 表达式之类的功能与 Java 模块系统之间还存在另一个根本区别。模块系统关注的是整个应用程序的大型结构，而将内部类转换为一个 Lambda 表达式则是单个类中相当小的局部变化。对一个应用程序进行模块化会影响设计、编译、打包、部署等过程，显然，这不仅仅是另一种语言功能。

随着新 Java 版本的发布，你可能会迫不及待地想使用这个新功能了。为了充分利用模块系统，首先进退一步，先了解一下什么是模块化，更重要的是为什么要关注模块化。

1.1 什么是模块化

到目前为止，所讨论的只是实现模块化的目标（管理和减少复杂性），却没有介绍实现模块化需要什么？从本质上讲，*模块化*（*modularization*）是指将系统分解成独立且相互连接的模块的行为。*模块*（*module*）是包含代码的可识别工件，使用了元数据来描述模块及其与其他模块的关系。在理想情况下，这些工件从编译时到运行时都是可识别的。一个应用程序由多个模块协作组成。

因此，模块对代码进行了分组，但不仅于此。模块必须遵循以下三个核心原则：

1. 强封装性

一个模块必须能够对其他模块隐藏其部分代码。这样一来，就可以在可公开使用的代码和被视为内部实现细节的代码之间划定一条清晰的界限，从而防止模块之间发生意外或不必要的耦合，即无法使用被封装的内容。因此，可以在不影响模块用户的情况下自由地对封装代码进行更改。

2. 定义良好的接口

虽然封装是很好的做法，但如果模块需要一起工作，那么就不能将所有的内容都进行封装。从定义上讲，没有封装的代码是模块公共 API 的一部分。由于其他模块可以使用这些公共代码，因此必须非常小心地管理它们。未封装代码中任何一个重大更改都可能会破坏依赖该代码的其他模块。因此，模块应该向其他模块公开定义良好且稳定的接口。

3. 显式依赖

一个模块通常需要使用其他模块来完成自己的工作，这些依赖关系必须是模块定义的一部分，以便使模块能够独立运行。显式依赖会产生一个模块图：节点表示模块，而边缘表示模块之间的依赖关系。拥有模块图对于了解应用程序以及运行所有必要的模块是非常重要的。它为模块的可靠配置提供了基础。

模块具有灵活性、可理解性和可重用性。模块可以灵活地组合成不同的配置，利用显式依赖来确保一切工作正常。封装确保不必知道实现细节，也不会造成无意识间对这些细节的依赖。如果想要使用一个模块，只需知道其公共 API 就可以了。此外，如果模块在封装了实现细节的同时公开了定义良好的接口，那么就可以很容易地使用符合相同 API 的实现过程来替换被封装的实现过程。

模块化应用程序拥有诸多优点。经验丰富的开发人员都知道使用非模块化的代码库会发生什么事情。诸如意大利面架构（*spaghetti architecture*）、凌乱的巨石（*messy monolith*）以及大泥球（*big ball of mud*）之类的术语都描述了由此所带来的痛苦。但模块化也不是

一种万能的方法。它是一种架构原则，如果使用正确则可以在很大程度上防止上述问题的产生。

也就是说，本节中所提供的模块化定义是刻意抽象化的，这可能会让你想到基于组件的开发（20 世纪曾经风靡一时）、面向服务的体系结构或当前的微服务架构。事实上，这些范例都试图在各种抽象层面上解决类似问题。

在 Java 中实现模块需要什么呢？建议先花时间思考一下在 Java 中已经存在哪些模块化的核心原则以及缺少哪些原则。

思考完了吗？如果想好了，就可以进入下一节的学习。

1.2 在 Java 9 之前

Java 可用于开发各种类型和规模的应用程序，开发包含数百万行代码的应用程序也是很常见的。显然，在构建大规模系统方面，Java 已经做了一些正确的事情——即使在 Java 9 出现之前。让我们再来看一下 Java 9 模块系统出现之前 Java 模块化的三个核心原则。

通过组合使用包（*package*）和访问修饰符（比如 `private`、`protected` 或 `public`），可以实现类型封装。例如，如果将一个类设置为 `protected`，那么就可以防止其他类访问该类，除非这些类与该类位于相同的包中。但这样做会产生一个有趣的问题：如果想从组件的另一个包中访问该类，同时仍然防止其他类使用该类，那么应该怎么做呢？事实是无法做到。当然，可以让类公开，但公开意味着对系统中的所有类型都是公开的，也就意味着没有封装。虽然可以将类放置到 `.impl` 或 `.internal` 包中，从而暗示使用此类是不明智的，但谁会在意呢？只要类可用，人们就会使用它。因此没有办法隐藏这样的实现包。

在定义良好接口方面，Java 自诞生以来就一直做得很好。你可能已经猜到了，我们所谈论的是 Java 自己的 `interface` 关键字。公开公共接口是一种经过验证的方法。它同时将实现类隐藏在工厂类后面或通过依赖注入完成。正如在本书中所看到的，接口在模块系统中起到了核心作用。

显式依赖是事情开始出问题的地方。Java 确实使用了显式的 `import` 语句。但不幸的是，从严格意义上讲，这些导入是编译时结构，一旦将代码打包到 JAR 中，就无法确定哪些 JAR 包含当前 JAR 运行所需的类型。事实上，这个问题非常糟糕，许多外部工具与 Java 语言一起发展以解决这个问题。以下栏目提供了更多的细节。

用来管理依赖关系的外部工具：Maven 和 OSGi

Maven

使用 Maven 构建工具所解决的一个问题是实现编译时依赖关系管理。JAR 之间的依赖关系在一个外部的 POM (Project Object Model, 项目对象模型) 文件中定义。Maven 真正成功之处不在于构建工具本身, 而是生成了一个名为 Maven Central 的规范存储库。几乎所有的 Java 库都与它们的 POM 一起发布到 Maven Central。各种其他构建工具, 比如 Gradle 或 Ant (使用 Ivy) 都使用相同的存储库和元数据, 它们在编译时会自动解析 (传递) 依赖关系。

OSGi

Maven 在编译时做了什么, OSGi 在运行时就会做什么。OSGi 要求将导入的包在 JAR 中列为元数据, 称之为**捆绑包 (bundle)**。此外, 还必须显式定义导出哪些包, 即对其他捆绑包可见的包。在应用程序开始运行时, 会检查所有的捆绑包: 每个导入的捆绑包都可以连接到一个导出的捆绑包吗? 自定义类加载器的巧妙设置可以确保在运行时, 除了元数据所允许的类型以外, 没有任何其他类型加载到捆绑包中。与 Maven 一样, 需要在 JAR 中提供正确的 OSGi 元数据。然而, 通过使用 Maven Central 和 POM, Maven 取得了巨大的成功, 但支持 OSGi 的 JAR 的出现却没有给人留下太深刻的印象。

Maven 和 OSGi 构建在 JVM 和 Java 语言之上, 这些都是它们所无法控制的。Java 9 解决了 JVM 和 Java 语言的核心中存在的一些相同问题。模块系统并不打算完全取代这些工具, Maven 和 OSGi (及类似工具) 仍然有自己的一席之地, 只不过现在它们可以建立在一个完全模块化的 Java 平台之上。

就目前来看, Java 为创建大型模块化应用程序提供了坚实的结构。当然, 它还存在很多需要改进的地方。

1.2.1 将 JAR 作为模块?

在 Java 9 出现之前, JAR 文件似乎是最接近模块的, 它们拥有名称、对相关代码进行了分组并且提供了定义良好的公共接口。接下来看一个运行在 JVM 之上的典型 Java 应用程序示例, 研究一下 JAR 作为模块的相关概念, 如图 1-1 所示。

在图 1-1 中, 有一个名为 *MyApplication.jar* 的应用程序 JAR, 其中包含了自定义的应用程序代码。该应用程序使用了两个库: Google Guava 和 Hibernate Validator。此外, 还有三个额外的 JAR。这些都是 Hibernate Validator 的可传递依赖项, 可能是由诸如 Maven

之类的构建工具所创建的。MyApplication 运行在 Java 9 之前的运行时上（该运行时通过几个捆绑的 JAR 公开了 Java 平台类）。虽然 Java 9 之前的运行时可能是一个 JRE（Java Runtime Environment，Java 运行时环境）或 JDK（Java Development Kit，Java 开发工具包），但无论如何，都包含了 *rt.jar*（运行时库），其中包含了 Java 标准库的类。

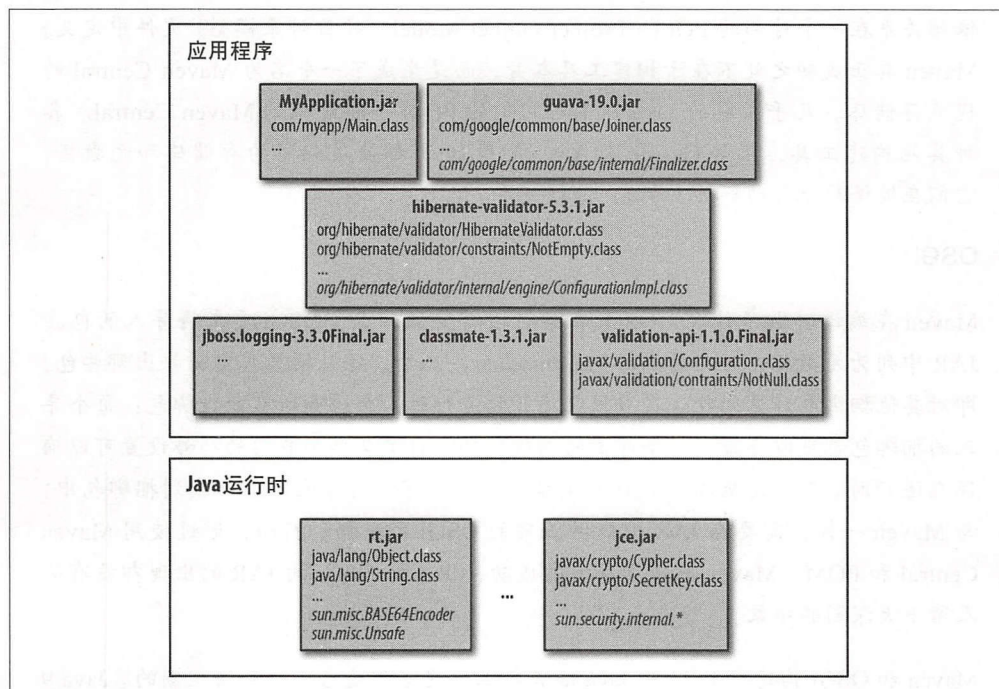


图 1-1: MyApplication 是一个典型的 Java 应用程序，其打包成一个 JAR 文件，并使用了其他库

当仔细观察图 1-1 时，会发现一些 JAR 以斜体形式列出了相关类。这些类都是库的内部类。例如，虽然在 Guava 中使用了 `com.google.common.base.internal.Finalizer`，但它并不是官方 API 的一部分，而是一个公共类，其他 Guava 包也可以使用 `Finalizer`。不幸的是，这也意味着无法阻止 `com.myapp.Main` 使用诸如 `Finalizer` 之类的类。换句话说，没有实现强封装性。

对于 Java 平台的内部类来说也存在同样的情况。诸如 `sun.misc` 之类的包经常被应用程序代码所访问，虽然相关文档严重警告它们是不受支持的 API，不应该使用。尽管发出了警告，但诸如 `sun.misc.BASE64Encoder` 之类的实用工具类仍然经常在应用程序代码中使用。从技术上讲，使用了这些类的代码可能会破坏 Java 运行时的更新，因为它们都是内部实现类。缺乏封装性实际上迫使这些类被认为是“半公共”API，因为 Java 高度重视向后兼容性。这是由于缺乏真正的封装所造成的结果。

什么是显式依赖呢？你可能已经看到，从严格意义上讲，JAR 不包含任何依赖信息。按照下面的步骤运行 MyApplication：

```
java -classpath lib/guava-19.0.jar:\
      lib/hibernate-validator-5.3.1.jar:\
      lib/jboss-logging-3.3.0.Final.jar:\
      lib/classmate-1.3.1.jar:\
      lib/validation-api-1.1.0.Final.jar \
      -jar MyApplication.jar
```

正确的类路径都是由用户设置的。由于缺少明确的依赖关系信息，因此完成设置并非易事。

1.2.2 类路径地狱

Java 运行时使用类路径 (*classpath*) 来查找类。上面的示例运行了 Main，所有从该类直接或间接引用的类都需要加载。可以将类路径视为可能在运行时加载的所有类的列表。虽然在“幕后”还有更多的内容，但是以上观点足以说明类路径所存在的问题。

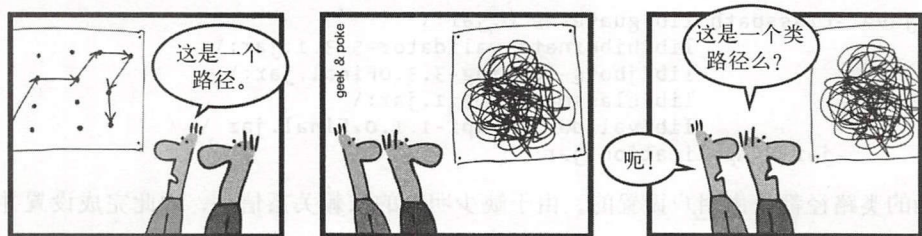
为 MyApplication 所生成的类路径简图如下所示：

```
java.lang.Object
java.lang.String
...
sun.misc.BASE64Encoder
sun.misc.Unsafe
...
javax.crypto.Cypher
javax.crypto.SecretKey
...
com.myapp.Main
...
com.google.common.base.Joiner
...
com.google.common.base.internal.Joiner
org.hibernate.validator.HibernateValidator
org.hibernate.validator.constraints.NotEmpty
...
org.hibernate.validator.internal.engine.ConfigurationImpl
...
javax.validation.Configuration
javax.validation.constraints.NotNull
```

此时，没有 JAR 或逻辑分组的概念了。所有类按照 `-classpath` 参数定义的顺序排列成一个平面列表。当 JVM 加载一个类时，需要按照顺序读取类路径，从而找到所需的类。一旦找到了类，搜索就会结束并加载类。

如果在类路径中没有找到所需的类又会发生什么情况呢？此时会得到一个运行时异常。由于类会延迟加载，因此一些不幸的用户在首次运行应用程序并点击一个按钮时会出现

找不到类的情况。JVM 无法在应用程序启动时有效地验证类路径的完整性，即无法预先知道类路径是否是完整的，或者是否应该添加另一个 JAR。显然，这并不够好。



当类路径上有重复类时，则会出现更为隐蔽的问题。假设尝试避免手动设置类路径，而是由 Maven 根据 POM 中的显式依赖信息构建将放到类路径中的 JAR 集合。由于 Maven 以传递的方式解决依赖关系问题，因此在该集合中出现相同库的两个版本（如 Guava 19 和 Guava 18）是非常常见的，虽然这并不是你的过错。现在，这两个库 JAR 以一种未定义的顺序压缩到类路径中。库类的任一版本都可能会被首先加载。此外，有些类还可能使用来自（可能不兼容的）其他版本的类。此时就会导致运行时异常。一般来说，当类路径包含两个具有相同（完全限定）名称的类时，即使它们完全不相关，也只有一个会“获胜”。

现在你应该明白为什么类路径地狱（*classpath hell*，也被称为 *JAR 地狱*）在 Java 世界如此臭名昭著了。一些人通过不断地摸索，逐步完善了调整类路径的方法——但该过程是相当艰苦的。脆弱的类路径仍然是导致问题和失败的主要原因。如果能够在运行时提供更多关于 JAR 之间关系的信息，那就最好了，就好像是一个隐藏在类路径中并等待被发现和利用的依赖关系图。接下来学习 Java 9 模块！

1.3 Java 9 模块

到目前为止，我们已经全面了解了当前 Java 在模块化方面的优势和局限。随着 Java 9 的出现，可以使用一个新的工具——Java 模块系统来开发结构良好的应用程序。在设计 Java 平台模块系统以克服当前所存在的局限时，主要设定了两个目标：

- 对 JDK 本身进行模块化。
- 提供一个应用程序可以使用的模块系统。

这两个目标是紧密相关的。JDK 的模块化可通过使用与应用程序开发人员在 Java 9 中使用的相同的模块系统来实现。

模块系统将模块的本质概念引入 Java 语言和运行时。模块既可以导出包，也可以强封装包。此外，它们显式地表达了与其他模块的依赖关系。可以看到，Java 模块系统遵循了

模块的三个核心原则。

接下来回到前面的 MyApplication 示例，此时示例建立在 Java 9 模块系统之上，如图 1-2 所示。

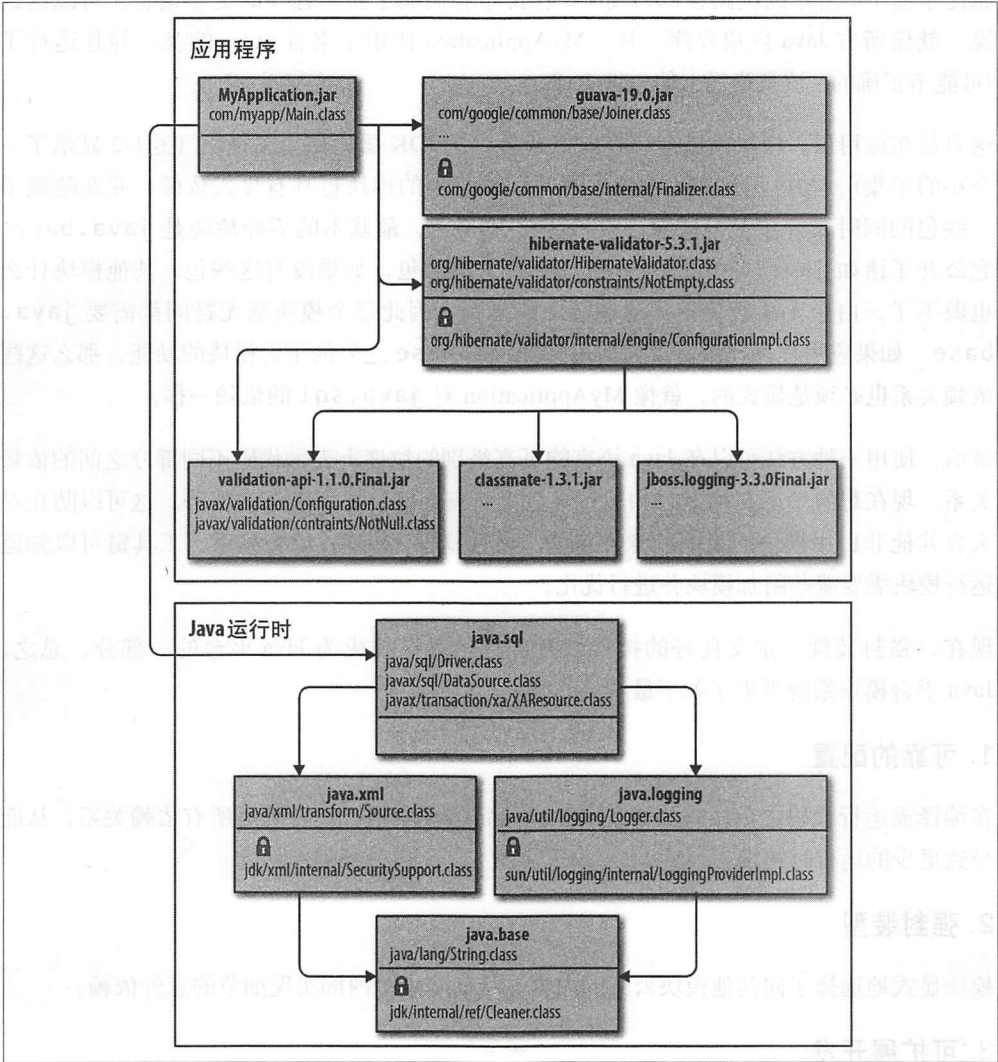


图 1-2: 建立在模块化 Java 9 之上的模块化应用程序 MyApplication

每个 JAR 都变成了一个模块，并包含了对其他模块的显式引用。hibernate-validator 的模块描述符表明了该模块使用了 jboss-logging、classmate 和 validation-api。一个模块拥有一个可公开访问的部分（位于顶部）以及一个封装部分（位于底部，并以一个挂锁表示）。这也就是为什么 MyApplication 不能再使用 Guava

的 `Finalizer` 类的原因。通过图 1-2，会发现 `MyApplication` 也使用了 `validation-api` 来注释它的类。此外，`MyApplication` 还显式依赖 JDK 中的一个名为 `java.sql` 的模块。

相比于图 1-1 所示的类路径图，图 1-2 告诉了我们关于应用程序的更多信息。可以这么说，就像所有 Java 应用程序一样，`MyApplication` 使用了来自 `rt.jar` 的类，并且运行了（可能不正确的）该类路径上的一堆 JAR。

这只是在应用层。模块的概念一直向下延伸，在 JDK 层也使用了模块（图 1-2 显示了一个小的子集）。与应用层中的模块一样，JDK 层中的模块也具有显式依赖，并在隐藏了一些包的同时公开了另一些包。在模块化 JDK 中，最基本的平台模块是 `java.base`。它公开了诸如 `java.lang` 和 `java.util` 之类的包，如果没有这些包，其他模块什么也做不了。由于无法避免使用这些包中的类型，因此每个模块毫无疑问都需要 `java.base`。如果应用程序模块需要任何来自 `java.base` 之外的平台模块的功能，那么这些依赖关系也必须是显式的，就像 `MyApplication` 对 `java.sql` 的依赖一样。

最后，使用一种方法可以在 Java 语言的更高级别的粒度上表示代码不同部分之间的依赖关系。现在想象一下在编译时和运行时获得所有这些信息所带来的优势。这可以防止对来自其他非引用模块的代码的意外依赖。通过检查（传递）依赖关系，工具链可以知道运行模块需要哪些附加模块并进行优化。

现在，强封装性、定义良好的接口以及显式依赖已经成为 Java 平台的一部分。总之，Java 平台模块系统带来了如下最重要的好处：

1. 可靠的配置

在编译或运行代码之前，模块系统会检查给定的模块组合是否满足所有依赖关系，从而导致更少的运行时错误。

2. 强封装型

模块显式地选择了向其他模块公开的内容，从而防止对内部实现细节的意外依赖。

3. 可扩展开发

显式边界能够让开发团队并行工作，同时可创建可维护的代码库。只有显式导出的公共类型是共享的，这创建了由模块系统自动执行的边界。

4. 安全性

在 JVM 的最深层次上执行强封装，从而减少 Java 运行时的攻击面，同时无法获得对敏

感内部类的反射访问。

5. 优化

由于模块系统知道哪些模块是在一起的，包括平台模块，因此在 JVM 启动期间不需要考虑其他代码。同时，其也为创建模块分发的最小配置提供了可能性。此外，还可以在 一组模块上应用整个程序的优化。在模块出现之前，这样做是非常困难的，因为没有可用的显式依赖信息，一个类可以引用类路径中任何其他类。

在下一章，将通过查看 JDK 中的模块，学习如何定义模块以及使用哪些概念管理模块之间的交互。JDK 包含了如图 1-2 所示更多的平台模块。

在第 2 章研究模块化 JDK 是了解模块系统概念的一个非常好的方法，同时还可以熟悉 JDK 中的模块。毕竟我们将首先使用这些模块创建自己的模块化 Java 9 应用程序。随后，在第 3 章我们将准备开始编写自己的模块。

模块和模块化 JDK

Java 有超过 20 年的发展历史。作为一种语言它仍然很受欢迎，这表明 Java 一直保持很好的状态。只要查看一下标准库，就会很明显地看到该平台长期的演变过程。在 Java 模块系统之前，JDK 的运行时库由一个庞大的 *rt.jar* 所组成（如前一章的图 1-1 所示），其大小超过 60MB，包含了 Java 大部分运行时类：即 Java 平台的最终载体。为了获得一个灵活且符合未来发展方向的平台，JDK 团队着手对 JDK 进行模块化——考虑到 JDK 的规模和结构，不得不说这是一个雄心勃勃的目标。在过去 20 年里，增加了许多 API，但几乎没有删除任何 API。

以 CORBA 为例，它曾经被认为是企业计算的未来，而现在是一种被遗忘的技术（对于那些仍然在使用 CORBA 的人，我们深表同情）。如今，JDK 的 *rt.jar* 中仍然存在支持 CORBA 的类。无论运行什么应用程序，Java 的每次发布都会包含这些 CORBA 类。不管是否使用 CORBA，这些类都在那里。在 JDK 中包含这些遗留类会浪费不必要的磁盘空间、内存和 CPU 时间。在使用资源受限设备或者为云创建小容器时，这些资源都是供不应求的。更不用说开发过程中在 IDE 自动完成和文档中出现这些过时类所造成的认知超载（cognitive overhead）。

但是，从 JDK 中删除这些技术并不是一个可行的办法。向后兼容性是 Java 最重要的指导原则之一，而移除 API 会破坏长久以来形成的向后兼容性。虽然这样做只会影响一小部分用户，但仍然有许多人在使用 CORBA 之类的技术。而在模块化 JDK 中，不使用 CORBA 的人可以选择忽略包含 CORBA 的模块。

另外，主动地弃用那些真正过时的技术也是可行的。只不过在 JDK 删除这些技术之前，可能还会再发布多个主要版本。此外，决定什么技术是真正过时的取决于 JDK 团队，这是一个非常难做的决定。



具体到 CORBA，该模块被标记为已弃用，这意味着它将有可能在随后主要的 Java 版本中被删除。

但是，分解整体 JDK 的愿望并不仅仅是删除过时的技术。很多技术对于某些类型的应用程序来说是有用的，而对于其他应用程序来说是无用的。JavaFX 是继 AWT 和 Swing 之后 Java 中最新的用户界面技术。这些技术当然是不能删除的，但显然它们不是每个应用程序都需要的。例如，Web 应用程序不使用 Java 中任何 GUI 工具包。然而，如果没有这些 GUI 工具包，就无法完成部署和运行。

除了便利与避免浪费之外，还要从安全角度进行考虑。Java 在过去经历过相当多的安全漏洞。这些漏洞都有一个共同的特点：不知何故，攻击者可以绕过 JVM 的安全沙盒并访问 JDK 中的敏感类。从安全的角度来看，在 JDK 中对危险的内部类进行强封装是一个很大的改进。同时，减少运行时中可用类的数量会降低攻击面。在应用程序运行时中保留大量暂时不使用的类是一种不恰当的做法。而通过使用模块化 JDK，可以确定应用程序所需的模块。

目前可以清楚地看到：极需要一种对 JDK 本身进行模块化的方法。

2.1 模块化 JDK

迈向模块化 JDK 的第一步是在 Java 8 中采用了**紧凑型配置文件** (*compact profile*)。配置文件定义了标准库中可用于针对该配置文件的应用程序的一个包子集。假设定义了三个配置文件，分别为 *compact1*、*compact2* 和 *compact3*。每个配置文件都是前一个配置文件的超集，添加了更多可用的包。使用这些预定义配置文件更新 Java 编译器和运行时。Java SE Embedded 8 (仅针对 Linux) 提供了与紧凑型配置文件相匹配的占用资源少的运行时。

如果你的应用程序符合表 2-1 中所描述的其中一个配置文件，那么就可以在一个较小的运行时上运行。但是，如果需要使用预定义配置文件之外的类，那么就比较麻烦了。从这个意义上讲，紧凑型配置文件的灵活性非常差，并且也无法解决强封装问题。作为一种中间解决方案，紧凑型配置文件实现了它的目的，但最终需要一种更灵活的办法。

表 2-1：为 Java8 定义的配置文件

配置文件	描述
compact1	带有 Java 核心类和日志记录以及脚本 API 的最小配置文件
compact2	使用 XML、JDBC 和 RMI API 扩展 compact1
compact3	使用安全和管理 API 扩展 compact2

从图 1-2 中可以看到 JDK 9 是如何被拆分为模块的。目前, JDK 由大约 90 个平台模块组成, 而不是一个整体库。与可由自己创建的应用程序模块不同的是, 平台模块是 JDK 的一部分。从技术上讲, 平台模块和应用模块之间没有任何技术区别。每个平台模块都构成了 JDK 的一个定义良好的功能块, 从日志记录到 XML 支持。所有模块都显式地定义了与其他模块的依赖关系。

图 2-1 显示了这些平台模块的子集及其依赖关系。每条边表示模块之间的单向依赖关系(稍后介绍实线和虚线之间的区别)。例如, `java.xml` 依赖于 `java.base`。如 1.3 节所述, 每个模块都隐式依赖于 `java.base`。在图 2-1 中, 只有当 `java.base` 是给定模块的唯一依赖项时, 才会显示这个隐式依赖关系, 比如 `java.xml`。

尽管依赖关系图看起来有点让人无所适从, 但是可以从中提取很多信息。只需观察一下该图, 就可以大概了解 Java 标准库所提供的功能以及各项功能是如何关联的。例如, `java.logging` 有许多传入依赖项 (*incoming dependencies*), 这意味着许多其他平台模块使用了该模块。对于诸如日志之类的中心功能来说, 这么做是很有意义的。模块 `java.xml.bind` (包含用于 XML 绑定的 JAXB API) 有许多传出依赖项 (*outgoing dependencies*), 包括 `java.desktop` (这是一个意料之外的依赖项)。事实上, 我们通过查看生成的依赖关系图并进行讨论而发现这个奇异之处, 这本身就是一个巨大的进步。由于 JDK 的模块化, 形成了清晰的模块边界以及显式的依赖关系。根据显式模块信息了解 JDK 之类的大型代码块是非常有价值的。

另一个需要注意的是, 依赖关系图中的所有箭头都是向下的, 图中没有循环。这是必然的: Java 模块系统不允许模块之间存在编译时循环依赖。



循环依赖通常是一种非常不好的设计。在 5.5.2 节中, 将讨论如何识别和解决代码库中的循环依赖。

除了 `jdk.httpserver` 和 `jdk.unsupported` 之外, 图 2-1 中的所有模块都是 Java SE 规范的一部分。它们的模块名都共享了前缀 `java.*`。每个认证的 Java 实现都必须包含这些模块。诸如 `jdk.httpserver` 之类的模块包含了工具和 API 的实现, 虽然这些实现不受 Java SE 规范的约束, 但是这些模块对于一个功能完备的 Java 平台来说是至关重要的。JDK 中还有很多的模块, 其中大部分在 `jdk.*` 命名空间。



通过运行 `java - -list-modules`, 可以获取平台模块的完整列表。

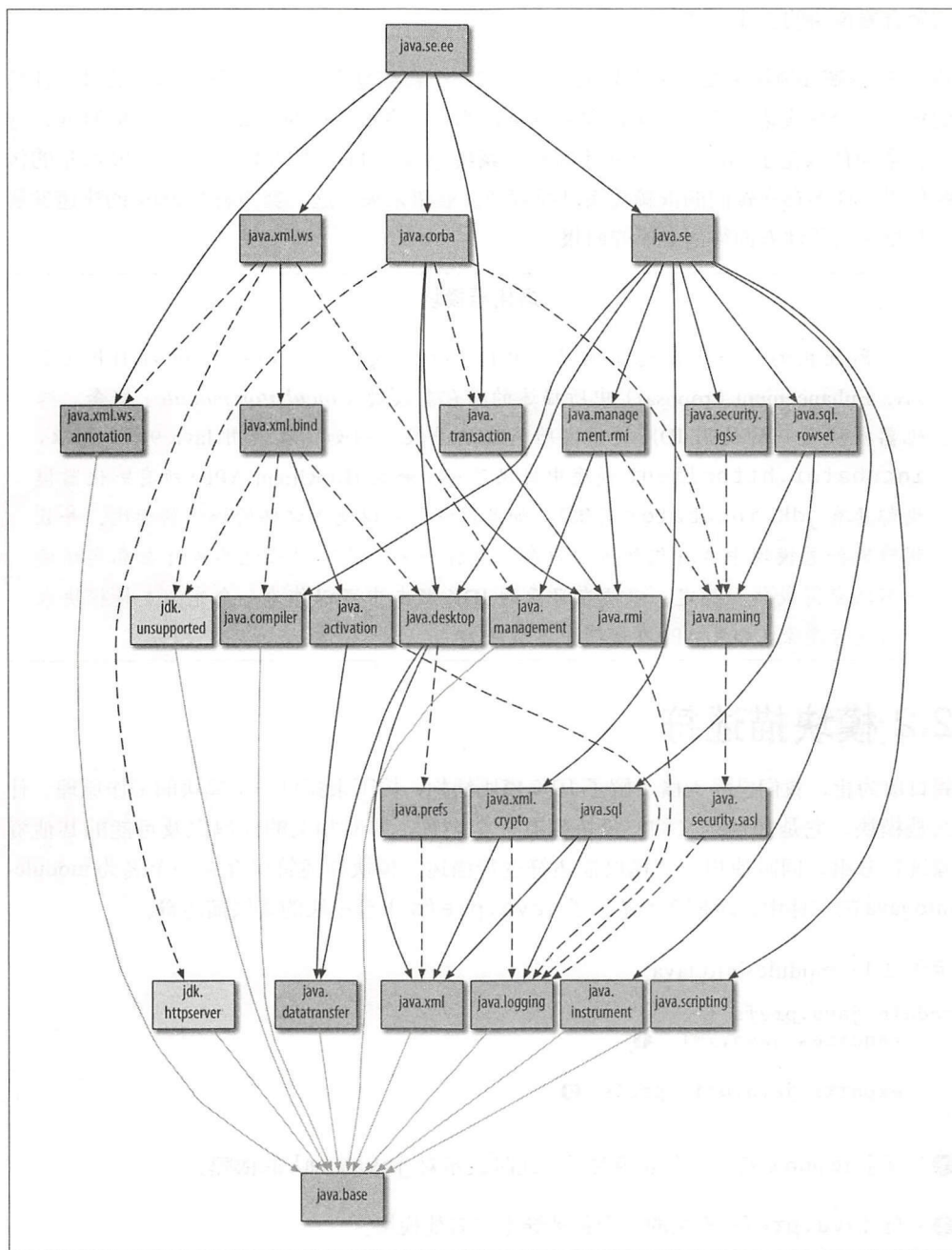


图 2-1: JDK 平台模块的子集

在图 2-1 的顶部可以找到两个重要的模块：`java.se` 和 `java.se.ee`。它们就是所谓的聚合器模块 (*aggregator module*)，主要用于对其他模块进行逻辑分组。本章稍后将介

绍聚合器模块的工作原理。

将 JDK 分解成模块需要完成大量的工作。将一个错综复杂、有机发展且包含数以万计类的代码库分解成边界清晰且保持向后兼容性的定义良好的模块需要花费大量的时间，这也就是为什么花了如此长的时间才将模块系统植入到 Java 中的原因。经过 20 多年的传统积累，许多存在疑问的依赖关系已经解开。展望未来，这一努力将在 JDK 的快速发展以及更大灵活性方面得到丰厚的回报。

孵化器模块

模块所提供的另一个改进示例是 JEP 11 (<http://openjdk.java.net/jeps/11> JEP 表示 Java Enhancement Proposal) 中所描述的孵化器模块 (*incubator module*) 概念。孵化器模块是一种使用 JDK 提供实验 API 的手段。例如，在使用 Java 9 时，`jdk.incubator.httpclient` 模块中提供了一个新的 `HttpClient` API (所有孵化器模块都具有 `jdk.incubator` 前缀)。如果愿意，可以使用这样的孵化器模块，并且明确地知道模块中 API 仍然可以更改。这样一来，就可以让这些 API 在真实环境中不断变得成熟和稳定，以便在日后的 JDK 版本中可以作为一个完全支持模块来使用或者删除 (如果 API 在实践中不成功)。

2.2 模块描述符

到目前为止，我们已经大概了解了 JDK 模块结构，接下来探讨一下模块的工作原理。什么是模块，它是如何定义的？模块拥有一个名称，并对相关的代码以及可能的其他资源进行分组，同时使用一个模块描述符进行描述。模块描述符保存在一个名为 `module-info.java` 的文件中。示例 2-1 显示了 `java.prefs` 平台模块的模块描述符。

示例 2-1: `module-info.java`

```
module java.prefs {
    requires java.xml; ❶
    exports java.util.prefs; ❷
}
```

❶关键字 `requires` 表示一个依赖关系，此时表示对 `java.xml` 的依赖。

❷来自 `java.prefs` 模块的单个包被导出到其他模块。

模块都位于一个全局命名空间中，因此，模块名称必须是唯一的。与包名称一样，可以使用反向 DNS 符号 (例如 `com.mycompany.project.somemodule`) 等约定来确保模块的唯一性。模块描述符始终以关键字 `module` 开头，后跟模块名称。而 `module-info.java` 的主体描述了模块的其他特征 (如果有的话)。

接下来看一下 `java.prefs` 模块描述符的主体。`java.prefs` 使用了 `java.xml` 中的代码从 XML 文件中加载首选项。这种依赖关系必须在模块描述符中表示。如果没有这个依赖关系声明，模块系统就无法编译（或运行）`java.prefs` 模块。声明依赖关系首先使用关键字 `requires`，然后紧跟模块名称（此时为 `java.xml`）。可以将对 `java.base` 的隐式依赖添加到模块描述符中。但这样做没有任何价值，就好比是将“`import java.lang.String`”添加到使用字符串的类中（通常并不需要这么做）。

模块描述符还可以包含 `exports` 语句。强封装性是模块的默认特性。只有当显式地导出一个包时（比如示例中的 `java.util.prefs`），才可以从其他模块中访问该包。默认情况下，一个模块中若没有导出的包则无法被其他模块所访问。其他模块不能引用封装包中的类型，即使它们与该模块存在依赖关系。从图 2-1 中可以看到，`java.desktop` 依赖 `java.prefs`，这意味着 `java.desktop` 只能访问 `java.prefs` 模块的 `java.util.prefs` 包中的类型。

2.3 可读性

在推理模块之间的依赖关系时，需要注意的一个重要的新概念是*可读性* (*readability*)，读取其他模块意味着可以访问其导出包中的类型。可以在模块描述符中使用 `requires` 子句设置模块之间的可读性关系。根据定义，每个模块都可以读取自己。而一个模块之所以读取另一个模块是因为需要该模块。

接下来，再次查看 `java.prefs` 模块，了解一下可读性的影响。在示例 2-2 的 JDK 模块中，`XmlSupport` 类导入并使用了 `java.xml` 模块中的类。

示例 2-2：类 `java.util.prefs.XmlSupport` 的节选

```
import org.w3c.dom.Document;
// ...
class XmlSupport {
    static void importPreferences(InputStream is)
        throws IOException, InvalidPreferencesFormatException
    {
        try {
            Document doc = loadPrefsDoc(is);
            // ...
        }
    }
    // ...
}
```

此时，导入了 `org.w3c.dom.Document`（以及其他类），该类来自 `java.xml` 模块。如示例 2-1 所示，由于 `java.prefs` 模块描述符包含了 `requires java.xml`，因此代码可以顺利完成编译。如果 `java.prefs` 模块的作者省略了 `require` 子句，那么

Java 编译器将报告错误。在模块 `java.prefs` 中使用来自 `java.xml` 的代码是一个经过深思熟虑并显式记录的选择。

2.4 可访问性

可读性关系表明了哪些模块可以读取其他模块，但读取一个模块并不意味着可以访问其导出包中的所有内容。在建立了可读性之后，正常的 Java 可访问性规则仍然会发挥作用。

从一开始，Java 就将可访问性规则内置到语言中。表 2-2 显示当前的访问修饰符及其影响。

表 2-2: 访问修饰符及其影响范围

访问修饰符	类	包	子类	无限制
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	
<code>- (default)</code>	✓	✓		
<code>private</code>	✓			

可访问性在编译时和运行时被强制执行。可访问性和可读性的结合可以确保在模块系统中实现强封装性。是否可以在模块 M1 中访问模块 M2 中的类型已经成为一个双重问题：

- 1) M1 是否可以读取 M2 ?
- 2) 如果可以，M2 导出包中的类型是否可以访问?

在其他模块中，只能访问导出包中的公共类型。如果导出包中的一个类型不是公共的，那么传统的可访问性规则将不允许使用该类型。如果类型是公共的，但没有导出，那么模块系统的可读性规则将阻止使用该类型。编译时的违规会导致编译器错误，而运行时的违规会导致 `IllegalAccessError`。

public 仍然表示公开的吗？

其他模块无法使用未导出包中的任何类型——即使包中的类型是公共的。这是对 Java 语言可访问性规则的根本变化。

在 Java 9 出现之前，事情非常简单明了。如果有一个公共类或接口，那么其他类就可以对它进行访问。但自从 Java 9 出现之后，`public` 意味着仅对模块中的其他包公开。只有当导出包包含了公开类型时，其他模块才可以使用这些类型。这就是强封装的意义所在。它迫使开发人员仔细设计包结构，将需要外部使用的类型与内部

实现过程分离开来。

在模块出现之前，强封装实现类的唯一方法是将这些类放置到单个包中，并标记为私有。这种做法使得包变得非常笨重，实际上，将类公开只是为了实现不同包之间的访问。通过使用模块，可以以任何方式构建包，并仅导出模块使用者真正必须访问的包。如果愿意的话，还可以将导出的包构成模块的 API。

关于可访问性规则的另一个方面是反射 (*reflection*)。在模块系统出现之前，所有反射对象都有一个有趣而又危险的方法 `setAccessible`。通过调用 `setAccessible(true)`，任何元素（不管元素是公共或是私有）都会变为可访问。虽然该方法目前仍然可以使用，但必须遵守前面所讨论的规则。想要在另一个模块导出的任意元素上调用 `setAccessible` 并期望像以前一样工作是不可能的了（即使反射不会破坏强封装性）。

可以使用多种方法实现模块系统中新的可访问性规则，其中大多数解决方法应该被视为迁移的辅助工具，详细内容参见第二部分。

2.5 隐式可读性

默认情况下，可读性是不可传递的。可以通过查看 `java.prefs` 的传入和传出“读取边”来说明这一点，如图 2-2 所示。

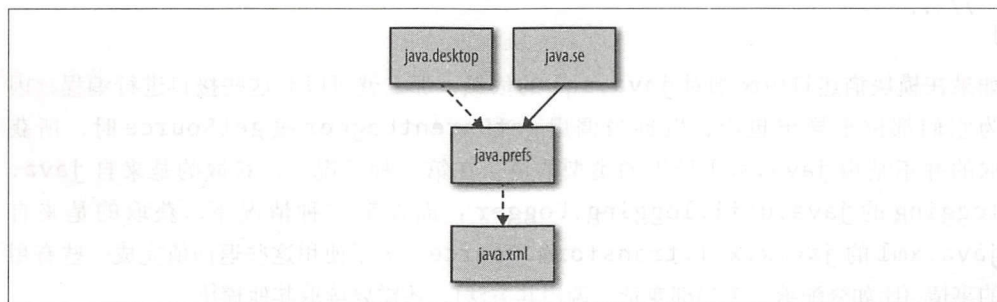


图 2-2：可读性是不可传递的：`java.desktop` 不能通过 `java.prefs` 读取 `java.xml`

此时，`java.desktop` 可以读取 `java.prefs`（为了简单起见，排除了其他模块）。这意味着 `java.desktop` 可以访问 `java.util.prefs` 包中的公共类型。然而，`java.desktop` 不能通过与 `java.prefs` 的依赖关系来访问 `java.xml` 中的类型，但此时 `java.desktop` 确实使用了 `java.xml` 中的类型，这是因为 `java.desktop` 的模块描述符中使用了 `requires java.xml` 子句。从图 2-1 中也可以看到这个依赖关系。

有时确实希望读取关系是可传递的，例如，当模块 M1 导出包中的一个类型需要引用模

块 M2 中的一个类型时。此时，如果不能读取 M2，那么需要 M1 进而需要引用 M2 中类型的模块就无法使用了。

这听起来非常抽象，所以下面依次提供示例加以说明。可以在 JDK 的 `java.sql` 模块中找到该现象的一个好示例。该模块包含两个接口（如示例 2-3 所示的 `Driver` 以及如示例 2-4 所示的 `SQLXML`），定义了结果类型来自其他模块的方法签名。

示例 2-3: `Driver` 接口（部分显示），允许检索 `java.logging` 模块中的 `Logger`

```
package java.sql;

import java.util.logging.Logger;

public interface Driver {
    public Logger getParentLogger();
    // ..
}
```

示例 2-4 `SQLXML` 接口（部分显示），来自模块 `java.xml` 的源代码，表示从数据库返回的 XML

```
package java.sql;

import javax.xml.transform.Source;

public interface SQLXML {
    <T extends Source> T getSource(Class<T> sourceClass);
    // ..
}
```

如果在模块描述符中添加对 `java.sql` 的依赖，那么就可以对这些接口进行编程，因为它们都位于导出包中。但每当调用 `getParentLogger` 或 `getSource` 时，所获取的并不是由 `java.sql` 导出的类型的值。在第一种情况下，获取的是来自 `java.logging` 的 `java.util.logging.Logger`；而在第二种情况下，获取的是来自 `java.xml` 的 `javax.xml.transform.Source`。为了使用这些返回值完成一些有用的事情（比如分配给一个局部变量、调用其方法），还需要读取其他模块。

当然，可以手动地在模块描述符中分别添加对 `java.logging` 或 `java.xml` 的依赖。但这样做是没有必要的，因为 `java.sql` 的作者已经意识到在其他模块上没有可读性的接口是不可用的。隐式可读性允许模块的作者模块描述符中表达这种可传递的可读性关系。

对于 `java.sql`，可完成如下修改：

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.xml;
    exports java.sql;
    exports javax.sql;
```



```
    exports javax.transaction.xa;
}
```

现在，关键字 `requires` 后面紧跟着修饰符 `transitive`，从而略微改变了一下语义。常见的 `requires` 只允许一个模块访问所需模块导出包中的类型，而 `requires transitive` 的语义更丰富。现在，任何需要 `java.sql` 的模块都将自动需要 `java.logging` 和 `java.xml`，这意味着可以通过隐式可读性关系访问这些模块的导出包。通过使用 `requires transitive`，模块作者可以为模块用户设置额外的可读性关系。

从使用者的角度来看，上述做法可以更容易地使用 `java.sql`。当你需要 `java.sql` 时，不仅可以访问导出包 `java.sql`、`javax.sql` 和 `javax.transaction.xa`（这些都直接由 `java.sql` 导出），还可以访问由模块 `java.logging` 和 `java.xml` 导出的所有包。这就好像这些包是由 `java.sql` 重新导出的一样，这一切都是使用 `requires transitive` 所设置的隐式可读性关系实现的。很明显，这并没有真的从其他模块重新导出包，但是这样考虑有助于理解隐式可读性的效果。

对于使用了 `java.sql` 模块的应用程序来说，其模块定义如下所示：

```
module app {
    requires java.sql;
}
```

使用上述模块描述符生成如图 2-3 所示的隐式可读性边。

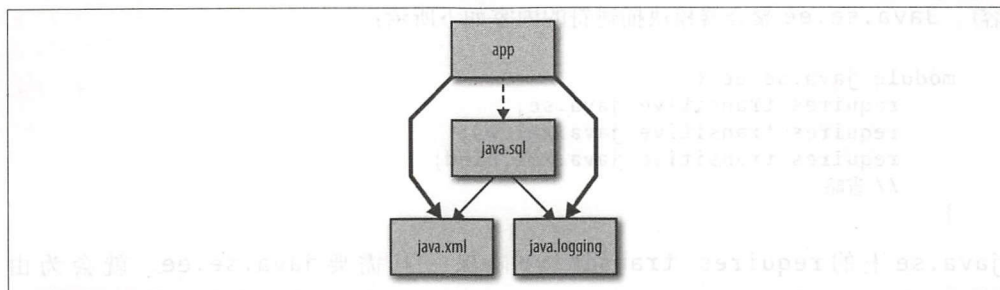


图 2-3：隐式可读性 (`requires transitive`) 的效果，用粗体边显示

`java.xml` 和 `java.logging` 上的隐式可读性被授予给 `app`，因为 `java.sql` 针对这些模块使用了 `requires transitive`（如图 2-3 所示的实边）。因为 `app` 没有导出任何内容，同时仅使用 `java.sql` 以完成封装的实现，所以使用常见的 `requires` 子句就足够了（如图 2-3 所示的虚线）。当需要其他模块供内部使用时，使用 `requires` 就足够了。但另一方面，如果需要在导出类型中使用另一个模块中的类型，那么就要使用 `requires transitive`。在 5.3 节中，将会更加详细地讨论隐式可读性的重要性及其何时对模块重要。

现在，可以再看看图 2-1。图中所有的实线边都是 `requires transitive` 依赖关系。而虚线边只是常见的 `requires` 依赖关系。*非传递依赖 (nontransitive dependency)* 意味着依赖关系是支持模块内部实现所必需的。*可传递依赖 (transitive dependency)* 则意味着依赖关系是支持模块 API 所必需的。后一种依赖关系更为重要，因此它们在本书的图形中用实线表示。

使用以上新的观点来观察图 2-1，可以看到隐式可读性的另一个用例：可用来将多个模块聚合到一个新模块中。以 `java.se` 为例，该模块不包含任何代码，只由一个模块描述符构成。在该模块描述符中，针对每个模块（Java SE 规则的一部分）都列出了一个 `requires transitive` 子句。当需要在一个模块中使用 `java.se` 时，凭借隐式可读性，可以访问 `java.se` 所聚合的每个模块导出的所有 API：

```
module java.se {
    requires transitive java.desktop;
    requires transitive java.sql;
    requires transitive java.xml;
    requires transitive java.prefs;
    // 省略
}
```

隐式可读性自身也是可传递的。以平台中另一个聚合器模块 `java.se.ee` 为例。从图 2-1 可以看到，相比于 `java.se`，`java.se.ee` 聚合了更多的模块。它使用了 `requires transitive java.se` 并添加了多个模块（包含了 Java EE 规范的部分内容）。`Java.se.ee` 聚合器模块描述符的内容如下所示：

```
module java.se.ee {
    requires transitive java.se;
    requires transitive java.xml.ws;
    requires transitive java.xml.bind;
    // 省略
}
```

`java.se` 上的 `requires transitive` 确保一旦需要 `java.se.ee`，就会为由 `java.se` 聚合的所有模块建立隐式可读性。此外，`java.se.ee` 还会对多个 EE 模块设置了隐式可读性。

最终，`java.se` 和 `java.se.ee` 为大量的模块提供了隐式可读性，只需通过这些可传递依赖关系就可以访问这些模块。



在应用程序模块中使用 `java.se.ee` 或 `java.se` 是不明智的，这意味着在模块中复制了 Java 9 之前可以访问的 `rt.jar` 的所有行为。依赖关系应尽可能细化。在模块描述符中要尽可能精确同时只添加实际使用的模块，这是非常重要的。

在 5.4 节将探讨聚合器模块模式如何帮助模块化库设计。

2.6 限制导出

在某些情况下，可能只需要将包暴露给特定的某些模块。此时，可以在模块描述符中使用**限制导出**。可以在 `java.xml` 模块中找到限制导出的示例：

```
module java.xml {
    ...
    exports com.sun.xml.internal.stream.writers to java.xml.ws
    ...
}
```

此时可以看到一个与另一个平台模块共享有用的内部代码的平台模块。导出的包只能由 `to` 之后指定的模块访问。可以用由逗号分隔的多个模块名称作为限制导出的目标。`to` 子句中没有提到的任何模块都不能访问包中的类型，即使在读取模块时也是如此。

限制导出的存在并不意味着就一定要使用它们。一般来说，应该避免在应用程序的模块之间使用限制导出。使用限制导出意味着在导出模块和允许的使用者之间建立了直接的联系。从模块化的角度来看，这是不可取的。模块之所以伟大，是因为它可以有效地对 API 的生产者与使用者进行解耦。而限制导出破坏了这个属性，因为现在使用者模块名称成为提供者模块描述符的一部分。

然而，对于模块化 JDK 来说，这只是一个小程序。如果想要使用遗留代码对平台进行模块化，那么限制导出是不可或缺的。许多平台模块封装了部分自身的代码，并通过限制导出公开了一些内部 API，以选择其他平台模块，同时对于应用程序中使用的公共 API 则使用正常的导出机制。通过使用限制导出，平台模块可以在不重复代码的情况下变得更为细致。

2.7 模块解析和模块路径

在模块之间建立显式依赖关系不仅仅有助于生成漂亮的图表。当编译和运行模块时，Java 编译器和运行时使用模块描述符来解析正确的模块。模块是从模块路径 (`module path`) 中解析出来的，而不是类路径。类路径是一个类型的平面列表 (即使使用 JAR 文件也是如此)，而模块路径只包含模块。如前所述，这些模块提供了导出包的显式信息，从而能够高效地对模块路径进行索引。当从给定的包中查找类型时，Java 运行时和编译器可以准确地知道从模块路径中解析哪个模块。而在以前，对整个类路径进行扫描是找到任意类型的唯一方法。

当需要运行一个打包成模块的应用程序时，还需要使用其所有的依赖项。模块解析是根据给定的依赖关系图并从该图中选择一个根模块 (*root module*) 来计算最低需求的模块集的过程。从根模块可访问的每个模块最终都在解析模块集中。在数学上讲，这相当于计算依赖关系图的传递闭包 (*transitive closure*)。尽管听起来很吓人，但这个过程还是非常直观的：

- 1) 首先从一个根模块开始，并将其添加到解析集中。
- 2) 向解析集添加所需的模块 (*module-info.java* 中的 `requires` 或 `requires transitive`)。
- 3) 重复第 2 步，将新的模块添加到解析集中。

上述过程不会无限制地进行下去，因为只有发现了新的模块才会重复该过程。此外，依赖关系图必须是非循环的。如果想要为多个根模块解析模块，则需要首先将该算法应用于每个根模块，然后再合并结果集。

接下来尝试一个示例。此时有一个应用程序模块 `app`，它是解析过程中的根模块，并且只使用了来自模块化 JDK 的 `java.sql`：

```
module app {  
    requires java.sql;  
}
```

现在，完成模块解析过程。当考虑到模块的依赖关系时，忽略 `java.base`，并假定它始终是解析模块的一部分。可以根据图 2-1 所示的边来完成下面的步骤：

- 1) 向解析集添加 `app`；并观察到它需要使用 `java.sql`。
- 2) 向解析集添加 `java.sql`；并观察到它需要 `java.xml` 和 `java.logging`。
- 3) 向解析集添加 `java.xml`；并观察到它不再需要其他模块。
- 4) 向解析集添加 `java.logging`，并观察到它不再需要其他模块。
- 5) 不再需要添加任何新模块；解析过程结束。

该解析过程的结果是生成了一个包含 `app`、`java.sql`、`java.xml`、`java.logging` 和 `java.base` 的集合。当运行 `app` 时，会按照上述过程解析模块，模块系统从模块路径中获取模块。

在解析过程中还会完成一些额外的检查。例如，具有相同名称的两个模块在启动时（而不是在运行过程出现类加载失败时）会产生错误。此外，还会检查导出包的唯一性。模块路径上只有一个模块可以公开一个给定的包。5.5.1 节将讨论多个模块导出相同包的问题。

版本

前面已经讨论了模块的解析过程，却没有提及版本问题。这看上去可能很奇怪，因为我们习惯于指定具有依赖关系的版本，如 Maven POM。将版本选择 (*version selection*) 放置在 Java 模块系统的范围之外是一个经过深思熟虑的设计决策。在模块解析过程中版本不起任何作用，5.7 节将会深入讨论这一决策。

模块解析过程以及额外的检查确保了应用程序在一个可靠的环境中运行，降低了运行时失败的可能性。在第 3 章，将会学习如何在编译和运行自己的模块时构建一个模块路径。

2.8 在不使用模块的情况下使用模块化 JDK

到目前为止，已经学习了模块系统所引入的许多新概念。此时，你可能想知道模块系统如何影响现有的代码，很显然，这些代码并没有实现模块化。难道真的需要将代码转换为模块才能开始使用 Java 9 吗？幸好不是这样。Java 9 可以像以前的 Java 版本一样使用，而不必将代码移动到模块中。模块系统完全适用于现有的应用程序代码，类路径仍然可以使用。

但 JDK 本身由模块组成。这两个世界是如何调解的呢？接下来看一下示例 2-5 所示的代码片段。

示例 2-5: *NotInModule.java*

```
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.LogRecord;

public class NotInModule {

    public static void main(String... args) {
        Logger logger = Logger.getGlobal();
        LogRecord message = new LogRecord(Level.INFO, "This still works!");
        logger.log(message);
    }
}
```

上面所示的仅仅是一个类，而不是任何模块。该代码使用了来自 JDK 的 `java.logging` 模块中的类型，但却没有任何模块描述符来表示这种依赖关系。当编译没有模块描述符的代码，然后将其放置到类路径并运行时，代码可以正常工作。怎么会这样呢？在一个模块之外编译和加载的代码最终都放在未命名模块 (*unnamed module*) 中。相比之下，目前所看到的模块都是显式模块，并在 `module-info.java` 中定义了它们的名称。未命名模块非常特殊：它可以读取所有其他模块，包括此例所读取的 `java.`

logging 模块。

通过使用未命名模块，尚未模块化的代码可以继续运行在 JDK 9 上。当将代码放在类路径上时，会自动使用未命名模块。这也意味着需要构建一个正确的类路径。可一旦使用了未命名模块，前面讨论的模块系统所带来的保障和好处也就没有了。

当在 Java 9 中使用类路径时，还需要注意两件事情。首先，由于平台是模块化的，因此对内部实现类进行了强封装。在 Java 8 和更早的版本中，可以使用这些不被支持的内部 API，而不会有任何不良影响。但如果使用 Java 9，则不能对平台模块中的封装类型进行编译。为了便于迁移，使用了内部 API 在早期版本的 Java 上编译的代码现在可以继续运行在 JDK 9 类路径上。



当在 Java 9 类路径上运行（而不是编译）一个应用程序时，使用了更为宽松的强封装形式。在 JDK 8 以及更早版本上可访问的所有内部类在 JDK 9 运行时上都是可访问的。但是当通过反射访问这些封装类型时，会出现警告信息。

在未命名模块中编译代码时需要注意的第二件事是编译期间 `java.se` 将作为根模块。如示例 2-5 所示，可以通过 `java.se` 访问任何可访问模块中的类型。这意味着 `java.se.aa`（而不是 `java.se`）下的模块（比如 `java.corba` 和 `java.xml.ws`）都无法解析，因此也就无法访问。此策略最突出的示例之一就是 JAXB API。以上两种限制背后的原因以及处理方法将在第 7 章更详细地讨论。

在本章，主要学习了如何对 JDK 进行模块化。虽然模块在 JDK 9 中起到了核心作用，但却是应用程序的可选项。如前所示，虽然目前已经采取谨慎措施，以确保在 JDK 9 之前的类路径上运行的应用程序可以继续运行，但也有一些注意事项。在下一章中，将会更加详细地讨论前面所介绍的模块概念，并使用它们来构建自己的模块。

第 3 章

使用模块

在本章，将迈出使用 Java 9 进行模块开发的第一步，即动手编写自己的第一个模块，而不仅仅是查看 JDK 中现有的模块。为了轻松地走好第一步，首先创建一个最简单的模块，可以将其称为 *Modular Hello World*。有了相关的经验之后，就可以开始创建带有多个模块的更复杂的示例了，此时将会介绍贯穿本书的运行示例 *EasyText*。随着对模块系统的进一步了解，该示例的设计也将逐步完善。

3.1 第一个模块

在前一章，已经给出了模块描述符的示例。然而，一个模块通常不只是一个描述符。因此，*Modular Hello World* 超越了单个源文件的级别：需要在上下文中来研究该示例。我们将从编译、打包和运行单个模块开始，以了解模块的新工具选项。

3.1.1 剖析模块

第一个示例的目的是将下面所示的类编译成一个模块并运行（见示例 3-1）。首先从一个包的单一类开始，编译成单一模块。模块可能仅包含包中的类型，所以需要包定义。

示例 3-1: HelloWorld.java (↪ *chapter3/helloworld*)

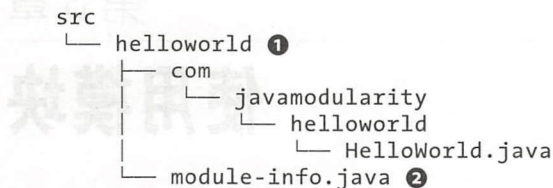
```
package com.javamodularity.helloworld;

public class HelloWorld {

    public static void main(String... args) {
        System.out.println("Hello Modular World!");
    }

}
```

文件系统中源文件的布局如下所示：



- ❶ 模块目录。
- ❷ 模块描述符。

相比于 Java 源文件的传统布局，上述布局存在两个主要区别。首先，有一个额外的间接层：在 `src` 的下面引入了另一个目录 `helloworld`，该目录以所创建的模块名称命名。其次，在模块目录中找到了源文件（像往常一样嵌套在包结构中）和一个模块描述符。模块描述符位于 `module-info.java` 文件中，是 Java 模块的关键组成部分。它的存在相当于告诉 Java 编译器正在使用的是一个模块，而不是普通的 Java 源代码。如本章的后续内容所述，与普通的 Java 源文件相比，当使用模块时，编译器的行为是完全不同的。模块描述符必须存在于模块目录的根目录中。它与其他源文件一起编译成一个名为 `module-info.class` 的二进制类文件。

模块描述符的内部是什么呢？*Modular Hello World* 示例非常简单：

```

module helloworld {
}

```

此时，使用新关键字 `module` 并紧跟模块名称声明了一个模块。该名称必须与包含模块描述符的目录名称相匹配。否则，编译器将拒绝编译并报告匹配错误。



仅在多模块模式下（这种情况是非常常见的）运行编译器时，才需要满足名称匹配要求。对于 3.1.3 节中所讨论的单模块方案，目录名称无关紧要。但在任何情况下，使用模块名称作为目录名称不失为一个好主意。

由于模块声明体是空的，因此不会从 `helloworld` 模块中导出任何内容到其他模块。默认情况下，所有包都是强封装的。即使目前在这个声明中没有任何依赖关系信息，但是请记住，该模块隐式地依赖 `java.base` 平台模块。

此时，你可能会问，向语言中添加新的关键字是否会破坏使用 `module` 作为标识符的现有代码。幸运的是，情况并非如此。仍然可以在其他源文件中使用名为 `module` 的标识符，因为 `module` 关键字是限制关键字（*restricted keyword*），它仅在 `module-info.java`

中被视为关键字。对于目前在模块描述符中所看到的 `requires` 关键字和其他新关键字来说，也是一样的。

名称 `module-info`

通常，Java 源文件的名称与其所包含的（公共）类型相对应。例如，包含 `HelloWorld` 类的文件名必须为 `HelloWorld.java`。但名称 `module-info` 打破了这种对应关系。此外，`module-info` 甚至不是一个合法的 Java 标识符，因为它包含了破折号。这样做的目的是防止非模块感知工具盲目地将 `module-info.java` 或 `module-info.class` 作为普通的 Java 类加以处理。

为特殊源文件保留名称在 Java 语言中并不是没有出现过。在 `module-info.java` 之前，就已经有了 `package-info.java`。该名称虽然可能相对比较陌生，但自 Java 5 之后它就出现了。在 `package-info.java` 中，可以向包声明中添加文档和注释。与 `module-info.java` 一样，它也是由 Java 编译器编译成一个类文件。

现在，已经拥有了一个仅包含模块声明的模块描述符以及一个源文件。接下来可以编译第一个模块了！

3.1.2 命名模块

命名事物虽然很难，但却很重要。尤其是对于模块而言更是如此，因为它们将传递应用程序的高级结构。

模块名称所在的全局命名空间与 Java 中其他命名空间是分开的。因此，从理论上讲，模块的名称可以与类、接口或包的名称相同。但实际上，这样做可能会导致混乱。

模块名称必须是唯一的：应用程序只能具有一个给定名称的模块。在 Java 中，通常使用反向 DNS 符号来确保包名称全局唯一。可以对模块应用相同的方法。例如，可以将 `helloworld` 模块重命名为 `com.javamodularity.helloworld`，但这样做会导致模块名称过长且有些笨重。

应用程序中的模块名称是否需要全局唯一呢？答案是肯定的，当模块是已发布的库并且在许多应用程序中使用时，选择全局唯一的模块名称就显得非常有意义了。在 10.2 节中，将会进一步讨论这个概念。对于应用程序模块来说，要尽量选择更短且更令人难忘的名词。

为了增加示例的可读性，本书选择使用更简短的模块名称。

3.1.3 编译

有一个源文件格式的模块是一回事，但要运行该模块，首先必须进行编译。在 Java 9 之前，Java 编译器使用目标目录以及一组源文件进行编译：

```
javac -d out src/com/foo/Class1.java src/com/foo/Class2.java
```

在实践中，通常是通过构建工具（如 Maven 或 Gradle）来完成的，但原理是一样的。在目标目录中输出类（此时使用了 `out`），其中包含代表输入（包）结构的嵌套文件夹。按照同样的模式，可以编译 Modular Hello World 示例：

```
javac -d out/helloworld \
  src/helloworld/com/javamodularity/helloworld/HelloWorld.java \
  src/helloworld/module-info.java
```

存在两个显著的区别：

- 输出到反映了模块名称的 `helloworld` 目录。
- 将 `module-info.java` 作为额外源文件进行编译。

在要编译的文件集中出现 `module-info.java` 源文件会触发 `javac` 的模块感知模式。下面显示的输出是编译后的结果，也被称为分解模块（*exploded module*）格式：

```
out
├── helloworld
│   ├── com
│   │   └── javamodularity
│   │       ├── helloworld
│   │       └── HelloWorld.class
│   └── module-info.class
```

最好在模块之后命名包含分解模块的目录，但不是必需的。最终，模块系统是从描述符中获取模块名称，而不是从目录名称中。在 3.1.5 节中，将会创建并运行这个分解模块。

编译多个模块

到目前为止所看到的都是所谓的 Java 编译器单模块模式（*single-module mode*）。通常，需要编译的项目由多个模块组成，这些模块还可能会相互引用。又或者项目是单个模块，但却使用了其他（已经编译）的模块。为了处理这些情况，引入了额外的编译器标志：`--module-source-path` 和 `--module-path`。这些标志都是 `-sourcepath` 和 `-classpath` 标志（长期以来，这些标志一直是 `javac` 的一部分）的模块感应对应项。在学习 3.2.2 节中的多模块示例时，将会解释其语义。请记住，模块源目录的名称必须与在多模块模式下 `module-info.java` 中声明的名称相匹配。

构建工具

直接通过命令行使用 Java 编译器、操作其标志以及手动列出所有源文件并不是常见的做法，更常见的做法是使用 Maven 或 Gradle 等构建工具来抽取这些细节信息。因此，本书不会详细介绍添加到 Java 编译器和运行时的每个新选项，可以在官方文档中找到相关详细信息 (<http://bitly/tools-comm-ref>)。当然，构建工具也需要适应新的模块化现实。在第 11 章中，将介绍一些可以与 Java 9 一起使用的最流行的构建工具。

3.1.4 打包

到目前为止，已经创建了单个模块并将其编译为分解模块格式，在下一节将会讨论如何运行分解模块。这种格式在开发环境下是可行的，但在生产环境中则需要以更方便的格式分发模块。为此，可以将模块打包成 JAR 文件并使用，从而产生了模块化 JAR 文件。模块化 JAR 文件类似于普通的 JAR 文件，但它还包含了 *module-info.class*。

JAR 工具已经进行了更新，从而可以使用 Java 9 中的模块。为了打包 Modular Hello World 示例，请运行下面的命令：

```
jar -cfe mods/helloworld.jar com.javamodularity.helloworld.HelloWorld \
  -C out/helloworld .
```

通过上述命令，在 mods 目录（请确保该目录存在）中创建了一个新存档文件（-cf）*helloworld.jar*。此外，还希望这个模块的入口点（-e）是 HelloWorld 类；每当模块启动并且没有指定另一个要运行的主类时，这是默认入口点。此时提供了完全限定的类名称作为入口点的参数。最后，指示 jar 工具更改（-C）为 *out/helloworld* 目录，并将此目录中的所有已编译文件放在 JAR 文件中。现在，JAR 的内容类似于分解模块，同时还额外添加了一个 *MANIFEST.MF* 文件：

```
helloworld.jar
├── META-INF
│   └── MANIFEST.MF
├── com
│   └── javamodularity
│       └── helloworld
│           └── HelloWorld.class
└── module-info.class
```

与编译期间的模块目录名称不同，模块化 JAR 文件的名称并不重要。可以使用任意喜欢的文件名，因为模块由绑定的 *module-info.class* 中声明的名称所标识。

3.1.5 运行模块

现在，回顾一下目前所完成的事情。首先从 Modular Hello World 示例开始，创建了一个

带有单个 `HelloWorld.java` 源文件和模块描述符的 `helloworld` 模块。然后，将模块编译成分解模块格式。最后，将分解模块打包成一个模块化 JAR 文件。这个 JAR 文件包含了已编译的类和模块描述符，并且知道要执行的主类。

尝试运行模块，分解模块格式和模块化 JAR 文件都可以运行。可以使用下面的命令运行分解模块格式：

```
$ java --module-path out \
    --module helloworld/com.javamodularity.helloworld.
HelloWorldHello Modular World!
```



还可以使用 `--module-path` 的缩写格式 `-p`。标志 `--module` 可以缩写为 `-m`。

除了基于类路径的应用程序之外，Java 命令还获得了新的标志来处理模块。请注意，此时将 `out` 目录（包含分解的 `helloworld` 模块）放在模块路径上。模块路径是原始类路径的模块感知对应项。

接下来使用 `--module` 标志提供所运行的模块。此时，模块名称后跟斜杠，然后是要运行的类。另一方面，如果运行模块化 JAR，则只需提供模块名称即可：

```
$ java --module-path mods --module helloworld
Hello Modular World!
```

这么做是有道理的，因为模块化 JAR 知道要从其元数据执行的类。在构建模块化 JAR 时已经显式地将入口点设置为 `com.javamodularity.helloworld.Hello World`。



带有相应模块名称（和可选主类）的 `--module` 或 `-m` 标志必须始终在最后。任何后续参数都将传递至从给定模块启动的主类。

以这两种方式中的任何一种启动都会使 `helloworld` 成为执行的根模块。JVM 从这个根模块开始，解析从模块路径运行根模块所需的任何其他模块。如前面的 2.7 节所述，解析模块是一个递归过程：如果新解析的模块需要其他模块，那么模块系统会自动考虑到这一点。

在简单的 `HelloWorld` 例子中，没有执行太多的解析。可以向 `java` 命令中添加 `--show-module-resolution`，从而跟踪模块系统所采取的操作：

```
$ java --show-module-resolution --limit-modules java.base \
    --module-path mods --module helloworld
root helloworld file:///chapter3/helloworld/mods/helloworld.jar
Hello Modular World!
```

(通过添加 `java.base` 标志 `--limit-modules`，可以阻止通过服务绑定解析其他平台模块。下一章将会详细介绍服务绑定。)

此时，除了隐式需要的平台模块 `java.base` 之外，不再需要其他模块来运行 `helloworld`。在模块解析输出中仅显示了根模块 `helloworld`。这意味着运行 `Modular Hello World` 示例仅涉及运行时的两个模块，即 `helloworld` 和 `java.base`，其他平台模块或者模块路径上的模块都没有解析。在类加载期间，没有任何资源被浪费在搜索与应用程序无关的类上。



通过使用 `-Xlog:module=debug`，可以显示更多关于模块解析的诊断信息。以 `-X` 开头的选项都是非标准的，那些不是基于 OpenJDK 的 Java 实现可能不支持这些选项。

如果需要另一个模块来运行 `helloworld`，并且该模块不存在于模块路径上（或不是 JDK 平台模块的一部分），那么在启动时就会遇到错误。这种形式的可靠配置解决了使用类路径所面临的问题。在模块系统出现之前，只有当 JVM 在运行时尝试加载不存在的类时才会注意到缺少的依赖项。通过使用模块描述符的显式依赖信息，模块解析可以确保在运行任何代码之前对模块进行工作配置。

3.1.6 模块路径

虽然 *模块路径* 听起来与 *类路径* 类似，但两者的行为却完全不同。模块路径是各个模块以及包含模块的目录的路径列表。模块路径上的每个目录都可以包含零个或多个模块定义，其中模块定义可以是分解模块或模块化 JAR 文件。包含三个选项的示例模块路径如下所示：`out/:myexplodedmodule/:mypackagedmodule.jar`。`out` 目录中的所有模块都在模块路径上，并与模块 `myexplodedmodule`（目录）以及 `mypackagedmodule`（模块化 JAR 文件）相结合。



模块路径上的条目由默认的平台分隔符分隔。在 Linux/macOS 上，分隔符是一个冒号 (`java -p dir1:dir2`)；而在 Windows 上，则使用分号 (`java -p dir1; dir2`)。标志 `-p` 是 `--module-path` 的缩写。

最重要的是，模块路径上的所有工件都有模块描述符（可能是在运行中合成，8.4 节将会

详细讨论相关内容)，解析器依赖此信息找到模块路径上的正确模块。当模块路径上相同目录中具有相同名称的多个模块时，解析器就会显示错误，并且不会启动应用程序。这样一来，就可以防止以前在类路径上可能发生的 JAR 文件冲突的问题。



当具有相同名称的多个模块位于模块路径上的不同目录中时，则不会产生错误，而是选择第一个模块，并忽略具有相同名称的后续模块。

3.1.7 链接模块

前一节所示的模块系统仅解析了两个模块：`helloworld` 和 `java.base`。如果可以利用前面所学到的知识创建一个 Java 运行时的特殊分布，其中包含运行应用程序所需的最少模块，岂不是很好？而这正是在 Java 9 中使用自定义运行时映像 (*custom runtime image*) 所完成的事情。

在编译和运行时阶段之间，Java 9 引入了一个可选的链接阶段。通过使用一个名为 `jlink` 的新工具，可以创建仅包含运行应用程序所需的模块的运行时映像。使用以下命令，创建一个以 `helloworld` 为根模块的运行时映像：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods \
  --add-modules helloworld \
  --launcher hello=helloworld \
  --output helloworld-image
```

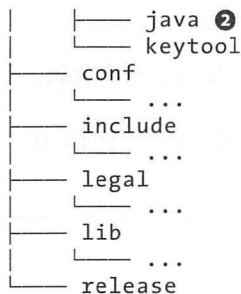


`jlink` 工具位于 JDK 安装目录下的 `bin` 目录。在默认情况下并没有将它添加到系统路径中，所以想要在示例中使用该工具，必须首先将其添加到路径中。

第一个选项是构造一个模块路径，其中包含 `mods` 目录 (`helloworld` 所在的位置) 以及要链接到映像中的平台模块的 JDK 安装目录。与 `javac` 和 `java` 不同，必须将平台模块显式添加到 `jlink` 模块路径中。随后，`--add-modules` 表示 `helloworld` 是需要运行时映像中运行的根模块。`--launcher` 定义了一个入口点来直接运行映像中的模块。最后，`--output` 表示运行时映像的目录名称。

运行上述命令的结果是生成一个新目录，包含了一个完全适合运行 `helloworld` 的 Java 运行时：

```
helloworld-image
├── bin
└── hello ①
```



- ① 直接启动 `helloworld` 模块的可执行脚本。
- ② 仅能解析 `helloworld` 及其依赖项的 Java 运行时。

由于解析器知道除了 `helloworld` 之外只需要使用 `java.base`，因此无须向运行时映像中添加更多的内容。因此，生成的运行时映像比完整的 JDK 小许多。可以在资源受限的设备上使用自定义运行时映像，或者将其作为在云中运行应用程序的容器映像的基础。虽然链接是可选的，却可以大大减少应用程序的占用空间。在第 13 章中，将会更详细地讨论自定义运行时映像的优点以及如何使用 `jlink`。

3.2 任何模块都不是一座孤岛

到目前为止，为了便于理解模块的创建机制和相关工具，特意将事情进行了简单化。然而，模块系统的真正魅力在于组合使用多个模块。也只有这样，模块系统的优势才会显现出来。

扩展 Modular Hello World 示例将是一件相当无聊的事情，因此，这里使用一个更有趣的示例应用程序 `EasyText`。首先从单个模块开始，然后逐渐创建一个多模块应用程序。虽然 `EasyText` 示例可能没有典型的企业应用程序那么大，但在实践中可以作为一种学习工具来使用。

3.2.1 EasyText 示例介绍

`EasyText` 是一个分析文本复杂性的应用程序。事实证明，可以将一些非常有趣的算法应用到文本以确定它的复杂性。如果对细节感兴趣，请阅读随后的“文本的复杂性”分析。

当然，我们关注的不是文本分析算法，而是构成 `EasyText` 应用程序的模块组合，主要目标是使用 Java 模块来创建灵活且可维护的应用程序。以下是 `EasyText` 模块化实现所需满足的要求：

- 必须能够在不修改或不重新编译现有模块的情况下添加新的分析算法。

- 不同的前端（如 GUI 和命令行）必须能够重用相同的分析逻辑。
- 必须支持不同的配置，同时无须重新编译，也无须部署每个配置的所有代码。

当然，即使不使用模块也可以满足所有这些需求。不过，这不是一件容易的工作。使用 Java 模块系统有助于更容易地满足这些要求。

文本的复杂性

虽然 EasyText 示例的重点是介绍解决方案的结构，但仍然可以学习其他新内容。文本分析是一个有着悠久历史的领域。EasyText 应用程序将可读性公式 (*readability formula*) 应用于文本。其中最受欢迎的可读性公式是 Flesch-Kincaid 评分：

$$\text{复杂度}_{\text{flesch_kincaid}} = 206.835 - 1.015 \frac{\text{totalwords}}{\text{totalsentences}} - 84.6 \frac{\text{totalsyllables}}{\text{totalwords}}$$

通过使用文本中一些相对容易的可推导性指标，可以计算得分。如果一个文本得分在 90 到 100 之间，那么普通 11 岁的学生可以很容易理解该文本，而得分在 0 到 30 范围内的文本则最适合研究生阶段的学生。

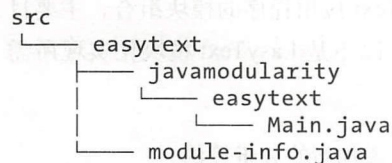
当然，还有很多其他可读性公式，如 Coleman-Liau 和 Fry 可读性公式，另外还有许多局部化公式。每个公式都有自己的范围，不存在最好的公式。当然，这也是一个让 EasyText 尽可能灵活的原因。

本章及其后续章节将会满足前面的所有要求。从功能角度来看，文本分析由以下步骤组成：

- 1) 读取输入文本（从文件、GUI 或者其他地方获取）。
- 2) 将文本拆分成句子和单词（因为许多可读性公式需要使用句子或单词）。
- 3) 对文本进行一次或者多次分析。
- 4) 向用户显示结果。

刚开始，其实现由单个模块 `easytext` 构成。从这一点上看，不存在关注点分离的问题。该模块内只有一个包，如示例 3-2 所示。

示例 3-2: 包含单个模块的 EasyText (`chapter3/easytext-singlemodule`)



模块描述符是空的。Main 类首先读取了一个文件，然后应用了一个可读性公式 (Flesch-Kincaid)，最后向控制台打印结果。在对包进行编译和打包之后，工作过程如下所示：

```
$ java --module-path mods -m easytext input.txt
Reading input.txt
Flesh-Kincaid: 83.42468299865723
```

显而易见，单个模块无法满足上述要求，接下来是时候添加更多的模块了。

3.2.2 两个模块

第一步，需要将文本分析算法和主程序分离成两个模块。这样一来，就可以在不同的前端模块上重复使用分析模块。主模块使用分析模块，如图 3-1 所示。

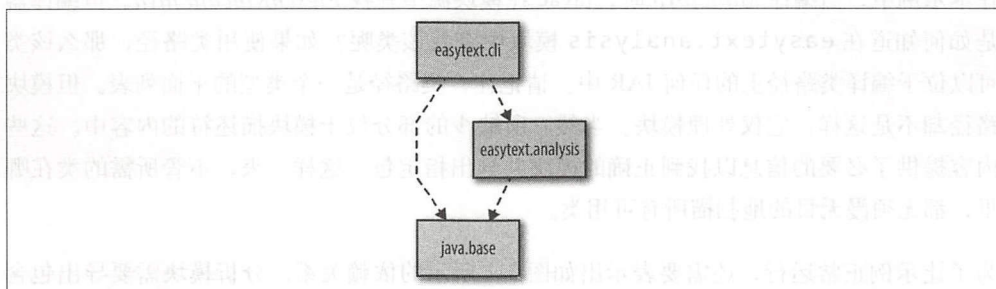


图 3-1: 两个模块中的 EasyText

easytext.cli 模块包含了命令行处理逻辑以及文件解析代码。easytext.analysis 模块包含了 Flesch-Kincaid 算法的实现过程。在分离单个 easytext 模块的过程中，在两个不同的包中创建了两个新模块，如示例 3-3 所示。

示例 3-3: 包含两个模块的 EasyText (`chapter3/easytext-twomodules`)

```
src
├── easytext.analysis
│   ├── javamodularity
│   │   ├── easytext
│   │   │   └── analysis
│   │   │       └── FleschKincaid.java
│   └── module-info.java
├── easytext.cli
│   ├── javamodularity
│   │   ├── easytext
│   │   │   └── cli
│   │   └── Main.java
│   └── module-info.java
```

所不同的是，现在 Main 类将算法分析委托给 FleschKincaid 类。因为有两个相互独立的模块，所以需要使用 javac 的多模块模式进行编译。

```
javac -d out --module-source-path src -m easytext.cli
```

此后，假设示例的所有模块总是被编译在一起的。只需使用 `-m` 指定要编译的实际模块即可，而无须列出所有源文件作为编译器的输入。在这种情况下，提供 `easytext.cli` 模块就足够了。编译器通过模块描述符知道，`easytext.cli` 也需要 `easytext.analysis`（也是通过模块源路径进行编译）。当然，也可以像示例 Hello World 那样只提供所有源文件列表[⊖]（不使用 `-m`）。

`--module-source-path` 标志告诉 `javac` 在编译期间去哪里查找源格式的其他模块。在多模块模式下进行编译时，必须使用 `-d` 提供目标目录。编译之后，目标目录包含了分解模块格式的编译模块。此输出目录还可以用作运行模块时模块路径上的一个元素。

在本示例中，当编译 `Main.java` 时，`javac` 在模块源中查找 `FleschKincaid.java`。但编译器是如何知道在 `easytext.analysis` 模块中查找该类呢？如果使用类路径，那么该类可以位于编译类路径上的任何 JAR 中。请记住，类路径是一个类型的平面列表。但模块路径却不是这样，它仅处理模块。当然，所缺少的部分位于模块描述符的内容中。这些内容提供了必要的信息以找到正确的模块并导出指定包。这样一来，不管所需的类在哪里，都无须漫无目的地扫描所有可用类。

为了让示例正常运行，还需要表示出如图 3-1 所示的依赖关系。分析模块需要导出包含 `FleschKincaid` 类的包：

```
module easytext.analysis {
    exports javamodularity.easytext.analysis;
}
```

通过使用关键字 `exports`，可以将模块中的包公开以供其他模块使用。通过声明导出包 `javamodularity.easytext.analysis`，其所有的公共类型都可以被其他模块使用。一个模块可以导出多个包。在本示例中，仅将 `FleschKincaid` 类导出。反之，模块中未导出的包都是模块私有的。

前面已经介绍了分析模块如何导出包含 `FleschKincaid` 类的包，而 `easytext.cli` 的模块描述符需要表达对分析模块的依赖：

```
module easytext.cli {
    requires easytext.analysis;
}
```

之所以需要 `easytext.analysis` 模块，是因为 `Main` 类导入了来自该模块的 `FleschKincaid` 类。完成了上述的模块描述符之后，就可以编译和运行代码了。

⊖ 在 Linux/ macOS 系统上，可以很容易地提供 `$(find -name '* .java')` 作为编译器的最后一个参数来实现此目的。

如果在模块描述符中省略 `requires` 语句，会发生什么事情呢？此时，编译器将产生如下所示的错误：

```
src/easytext.cli/javamodularity/easytext/cli/Main.java:11:
  error: package javamodularity.easytext.analysis is not visible
  import javamodularity.easytext.analysis.FleschKincaid;
                                     ^
  (package javamodularity.easytext.analysis is declared in module
  easytext.analysis, but module easytext.cli does not read it)
```

虽然 `FleschKincaid.java` 源文件对编译器可用（假设使用 `-m easytext.analysis, easytext.cli` 进行编译，以弥补所缺少的 `requires easytext.analysis`），但仍然会抛出一个错误。如果在分析模块的描述符中省略 `exports` 语句，也会产生类似的错误。此时，可以看到在软件开发过程的每个步骤中明确依赖关系的主要优势。模块只能使用它所需要的内容，编译器会强制执行此操作。在运行时，解析器使用相同的信息，以确保在启动应用程序之前所有模块都已存在。在编译时不会出现对库的意外依赖，只有在运行时才能发现这个库在类路径上不可用。

模块系统执行的另一个检查是循环依赖。在上一章已经讲过，在编译时，模块之间的可读性关系必须是非循环的。而在模块中，仍然可以在类之间创建循环关系，过去一直是这么做的。从软件工程的角度来看，是否真的需要这么做存在争议，但只要愿意，也是可以的。但是，在模块级别将别无选择。模块之间的依赖关系必须形成非循环的有向图。推而广之，不同模块中的类之间也不能存在循环依赖。如果引入了循环依赖关系，编译器就不会接受。向分析模块添加 `requires easytext.cli`，引入一个循环，如图 3-2 所示。

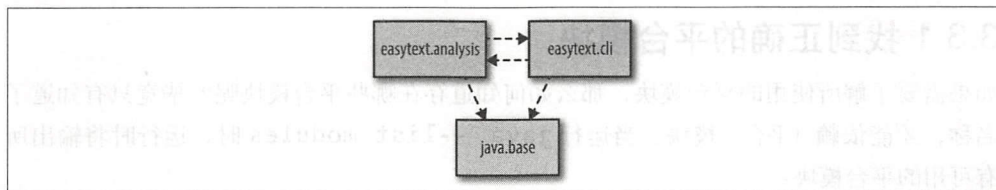


图 3-2：带有非法循环依赖关系的 EasyText 模块

如果尝试编译该模块，会出现如下所示的错误：

```
src/easytext.analysis/module-info.java:3:
  error: cyclic dependence involving easytext.cli
  requires easytext.cli;
                                     ^
```

请注意，循环也可以是间接的，如图 3-3 所示。虽然在实践中以下情况不太常见，但被视为与直接循环相同：它们导致 Java 模块系统发生错误。

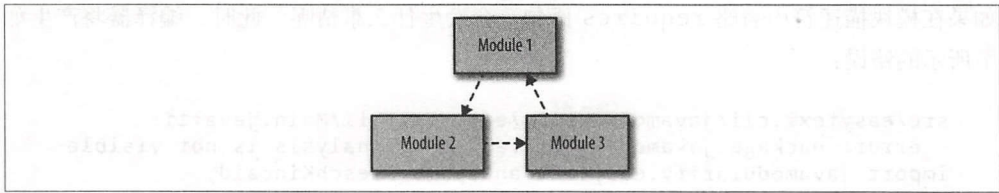


图 3-3: 循环也可以是间接的

许多实际应用程序的组件之间存在循环依赖关系。在 5.5.2 节，将会讨论如何防止和打破应用程序模块图中的循环。

3.3 使用平台模块

平台模块伴随着 Java 运行时，并提供了包括 XML 解析器、GUI 工具包在内的功能以及期望在标准库中看到的其他功能。图 2-1 显示了平台模块的一个子集。从开发人员的角度来看，它们的行为与应用程序模块相同。平台模块封装某些代码，可能导出包，并且可以依赖其他（平台）模块。使用模块化 JDK 意味着需要了解应用程序模块中所使用的平台模块。

在本节，将使用新的模块扩展 EasyText 应用程序。扩展后的应用程序使用平台模块，而不是前面所创建的模块。从技术上讲，我们已经使用了一个平台模块：`java.base` 模块。然而，这只是一个隐式依赖关系，接下来所创建的新模块与其他平台模块之间具有显式依赖关系。

3.3.1 找到正确的平台模块

如果需要了解所使用的平台模块，那么如何知道存在哪些平台模块呢？毕竟只有知道了名称，才能依赖（平台）模块。当运行 `java --list-modules` 时，运行时将输出所有可用的平台模块：

```
$ java --list-modules
java.base@9
java.xml@9
javafx.base@9
jdk.compiler@9
jdk.management@9
```

上面简短的输出显示了几种类型的平台模块。以 `java.` 为前缀的平台模块是 Java SE 规范的一部分。它们通过 Java SE 的 JCP (Java Community Process) 导出标准化的 API。JavaFX API 分布在共享 `javafx.` 前缀的模块中。以 `jdk.` 开头的模块包含了 JDK 特定的代码，在不同的 JDK 实现中可能会有所不同。

尽管 `--list-modules` 功能是发现平台模块的良好起点，但远远不够。如果导入一个不是由 `java.base` 导出的包时，则需要知道哪个平台模块提供了这个包，此时必须使用 `requires` 子句将该模块添加到 `module-info.java` 中。因此，让我们返回到示例应用程序，了解哪些模块与平台模块一起工作。

3.3.2 创建 GUI 模块

到目前为止，EasyText 使用了两个应用程序模块，命令行应用程序已经与分析模块分离开来。按照需求，希望在相同的分析逻辑之上支持多个前端。所以，接下来尝试创建除命令行版本之外的 GUI 前端。显然，该前端应该重用已经存在的分析模块。

使用 JavaFX 为 EasyText 创建一个合适的 GUI。从 Java 8 起，JavaFX GUI 框架就已经成为 Java 平台的一部分，旨在取代旧的 Swing 框架。此 GUI 如图 3-4 所示。

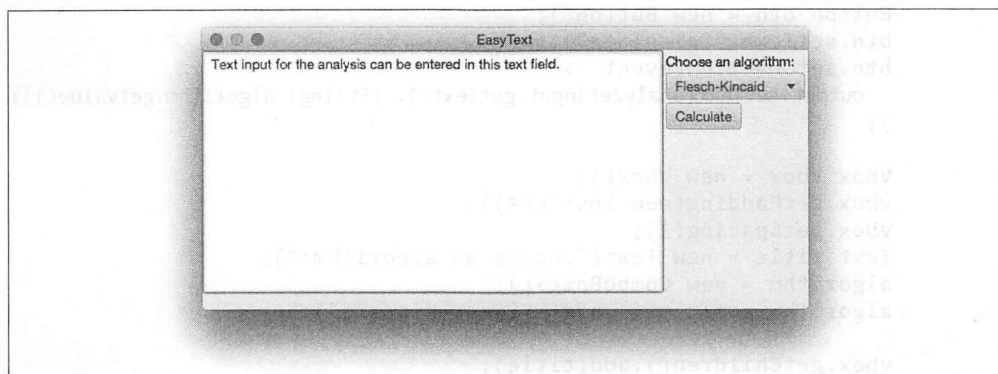


图 3-4: EasyText 简单的 GUI

当点击 Calculate 按钮时，分析逻辑在文本字段的文本上运行，并在 GUI 上显示结果值。虽然在下拉列表中只能选择一种分析算法，但稍后随着需求的扩展，会添加更多算法。目前暂时保持示例的简单性，假设 FleschKincaid 分析是唯一可用的算法。GUI Main 类的代码非常简单，如示例 3-4 所示。

示例 3-4: EasyText GUI 实现过程 (`chapter3/easytext-threemodules`)

```
package javamodularity.easytext.gui;

import java.util.ArrayList;
import java.util.List;

import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.*;
import javafx.scene.control.*;
```

```

import javafx.scene.layout.*;
import javafx.scene.text.Text;
import javafx.stage.Stage;

import javamodularity.easytext.analysis.FleschKincaid;

public class Main extends Application {

    private static ComboBox<String> algorithm;
    private static TextArea input;
    private static Text output;

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("EasyText");
        Button btn = new Button();
        btn.setText("Calculate");
        btn.setOnAction(event ->
            output.setText(analyze(input.getText(), (String) algorithm.getValue()))
        );

        VBox vbox = new VBox();
        vbox.setPadding(new Insets(3));
        vbox.setSpacing(3);
        Text title = new Text("Choose an algorithm:");
        algorithm = new ComboBox<>();
        algorithm.getItems().add("Flesch-Kincaid");

        vbox.getChildren().add(title);
        vbox.getChildren().add(algorithm);
        vbox.getChildren().add(btn);

        input = new TextArea();
        output = new Text();
        BorderPane pane = new BorderPane();
        pane.setRight(vbox);
        pane.setCenter(input);
        pane.setBottom(output);
        primaryStage.setScene(new Scene(pane, 300, 250));
        primaryStage.show();
    }

    private String analyze(String input, String algorithm) {
        List<List<String>> sentences = toSentences(input);
        return "Flesch-Kincaid: " + new FleschKincaid().analyze(sentences);
    }

    // 为简洁起见, 省略了 toSentences() 的实现过程
}

```

在 Main 类中导入了 8 个 JavaFX 包。如何知道 *module-info.java* 中需要哪些平台模块

呢？一种方法是使用 JavaDoc 找出包位于哪个模块中。对于 Java 9 来说，JavaDoc 已经进行了更新——包含了模块名称（类型是模块名称的一部分）。

另一种方法是使用 `java --list-modules` 检查可用的 JavaFX 模块。在运行完该命令后，可以看到名称中包含 `javafx` 的 8 个模块：

```
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
```

因为模块名称与其所包含的包之间并不总是一一对应的，所以选择正确的模块有点像根据上述列表所进行的一个猜测游戏。可以使用 `--describe-module` 检查平台模块的模块声明以验证猜测是否正确。例如，如果认为 `javafx.controls` 可能包含 `javafx.scene.control` 包，那么可以使用下面的代码加以验证：

```
$ java --describe-module javafx.controls
javafx.controls@9
exports javafx.scene.chart
exports javafx.scene.control ❶
exports javafx.scene.control.cell
exports javafx.scene.control.skin
requires javafx.base transitive
requires javafx.graphics transitive
...
```

❶ 模块 `javafx.controls` 导出了 `javafx.scene.control` 包。

事实上，所需要的包就包含在这个包中。以上述方法手动地找到正确平台模块的过程是相当单调乏味的。预计在 Java 9 提供了相应的支持之后，IDE 将会帮助开发人员完成此任务。对于 EasyText GUI 来说，只需要两个 JavaFX 平台模块：

```
module easytext.gui {
    requires javafx.graphics;
    requires javafx.controls;
    requires easytext.analysis;
}
```

根据上面的模块描述符，GUI 模块顺利编译。但是当尝试运行时，会出现下面的奇怪错误：

```
Exception in Application constructor
Exception in thread "main" java.lang.reflect.InvocationTargetException
...
Caused by: java.lang.RuntimeException: Unable to construct
```

```

Application instance:
    class javamodularity.easytext.gui.Main
      at javafx.graphics/..LauncherImpl.launchApplication1(LauncherImpl.
java:963)
      at javafx.graphics/..LauncherImpl.lambda$launchApplication$2(Launc
herImpl.java)
      at java.base/java.lang.Thread.run(Thread.java:844)
Caused by: java.lang.IllegalAccessException: class ..application.
LauncherImpl
    (in module javafx.graphics) cannot access class
    javamodularity.easytext.gui.Main
    (in module easytext.gui) because module easytext.gui does
not export
    javamodularity.easytext.gui to module javafx.graphics
      at java.base/..Reflection.newIllegalAccessException(Reflection.
java:361)
      at java.base/..AccessibleObject.checkAccess(AccessibleObject.
java:589)

```



Java 9 中另一个变化是，堆栈跟踪也会显示一个类来自哪个模块。斜杠 (/) 之前的名称是包含斜杠之后指定类的模块。

到底发生了什么事情呢？根本原因是由于无法加载 Main 类而产生了 `IllegalAccessException`。Main 类扩展了 `javafx.application.Application`（它位于 `javafx.graphics` 模块中），并从 `main` 方法调用了 `Application::launch`。这是启动 JavaFX 应用程序的一种典型方式，即将 UI 创建委托给 JavaFX 框架。然后 JavaFX 使用反射实例化 Main，随后调用 `start` 方法。这意味着 `javafx.graphics` 模块必须能够访问 `easytext.gui` 中的 Main 类。正如 2.4 节中所讲到的，访问另一个模块中的类需要满足两个条件：对目标模块的可读性以及目标模块必须导出给定的类。

在这种情况下，`javafx.graphics` 必须具有与 `easytext.gui` 的可读性关系。幸运的是，模块系统非常智能，可以动态地建立与 GUI 模块的可读性关系（显然，只要使用反射加载另一个模块中的类）。可问题是，包含 Main 类的包并不是从 GUI 模块中公开的。对于 `javafx.graphics` 模块来说，Main 是不可访问的，因为它没有被导出，而这也正是前面的错误信息所告诉我们的内容。

一种解决方案是在模块描述符中为 `javamodularity.easytext.gui` 包添加一个 `exports` 子句。只有这样才能将 Main 类暴露给任何需要 GUI 模块的模块。这真的是我们想要的结果吗？Main 类是否真的成为需要支持的公共 API 的一部分呢？答案是否定的。需要访问该类的唯一原因是 JavaFX 需要实例化它。而这恰恰是使用限制导出的时候：


```
module easytext.gui {  
    exports javamodularity.easytext.gui to javafx.graphics;  
    requires javafx.graphics;  
    requires javafx.controls;  
    requires easytext.analysis;  
}
```



在编译期间，限制导出的目标模块必须在模块路径上或者同时编译。显然对平台模块来说这不是问题，但是当对非平台模块进行限制导出时，这是一个需要注意的问题。

通过限制导出，只有 `javafx.graphics` 可以访问 `Main` 类。现在运行应用程序，JavaFX 能够实例化 `Main` 类。在 6.1.2 节，将会学习另一种方法来处理在运行时对模块内部的反射访问。

在运行时出现了一个有趣的情况。如上所述，`javafx.graphics` 模块在运行时动态地建立了与 `easytext.gui` 的可读性关系（如图 3-5 中粗箭头所示）。

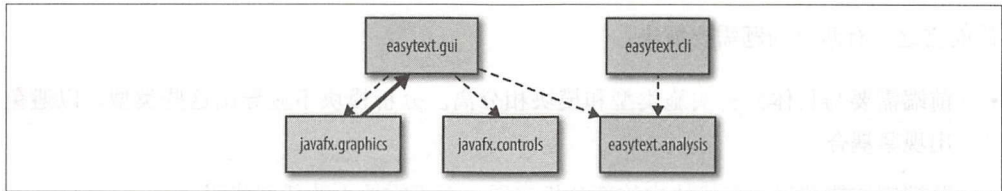


图 3-5：运行时的可读性边

这意味着可读性图中存在一个循环！这怎么可能？循环通常被认为是不可能的。在编译时可能存在循环。在本示例中，使用与 `javafx.graphics` 的依赖关系（也因此产生可读性关系）编译 `easytext.gui`。在运行时，当它以反射方式实例化 `Main` 时，`javafx.graphics` 自动建立与 `easytext.gui` 的可读性关系。在运行时，可读性关系可以是循环的。因为导出是受限的，所以只有 `javafx.graphics` 可以访问 `Main` 类。与 `easytext.gui` 建立可读性关系的任何其他模块将无法访问 `javamodularity.easytext.gui` 包。

3.4 封装的限制

回顾一下本章前面所学的内容，主要学习了如何创建和运行模块，以及如何将它们与平台模块结合使用。示例应用程序 `EasyText` 已从一个小型的整体应用程序发展成一个多模块 Java 应用程序。同时，还实现了两个规定要求：在重复使用相同的分析模块前提下支

持多个前端，以及创建针对命令行或 GUI 的模块的不同配置。

然而，看一下其他需求，会发现还有很多需要改进的地方。从目前情况来看，前端模块都会从分析模块中实例化一个特定的实现类（FleschKincaid）来完成自己的工作。虽然代码存在于单独的模块中，但此时却出现了紧耦合。如果想要使用不同的分析来扩展应用程序，那么应该怎么办呢？是否应该对每个前端模块进行修改以了解新的实现类呢？这听起来像是很差的封装。前端模块是否应该根据新引入的分析模块进行更新呢？这听起来很明显是非模块化的，也违反了在不修改或重新编译现有模块的情况下添加新的分析算法的要求。图 3-6 显示由两个前端和两个分析所带来的混乱（Coleman-Liau 是另一种众所周知的复杂性算法）。

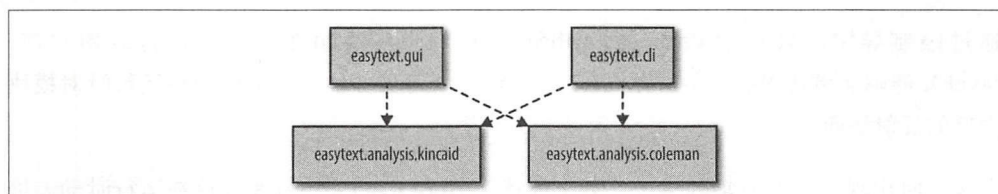


图 3-6：为了实例化导出的实现类，每个前端模块都需要依赖所有的分析模块

简而言之，有两个问题需要解决：

- 前端需要与具体分析实施类型和模块相分离。分析模块不应导出这些类型，以避免出现紧耦合。
- 前端应该能够发现新模块中的新分析实现，而不需要更改任何代码。

一旦解决了这两个问题，则只需将新的分析模块添加到模块路径上从而引入新的分析算法，而无须接触前端。

接口和实例化

在理想情况下，可以通过一个接口抽象出不同的分析。毕竟只是传递句子，并为每种算法返回一个分数：

```
public interface Analyzer {
    String getName();
    double analyze(List<List<String>> text);
}
```

只要能找到一种算法的名称（为了显示的需要）并计算复杂度就可以了，这种抽象类型恰恰是接口可以实现的。Analyzer 接口比较稳定，并且可以在自己的模块中使用，比

如 `easytext.analysis.api`。该接口是前端模块应该知道和关注的。分析实现模块也需要这个 API 模块，并实现 `Analyzer` 接口。到目前为止，一切都比较顺利。

然而，存在一个问题。即使前端模块仅关注如何通过 `Analyzer` 接口调用 `analyze` 方法，但它们仍然需要一个具体实例来调用该方法：

```
Analyzer analyzer = ???
```

如何才能找到一个实现了 `Analyzer` 接口却又不依赖特定实现类的实例呢？可以参考下面的代码：

```
Analyzer analyzer = new FleschKincaid();
```

不幸的是，如果想要上面的代码正常工作，仍然需要公开 `FleschKincaid` 类，这样一来又回到了原点。此时需要一种方法来获取不参考具体实现类的实例。

与计算机科学中的所有问题一样，可以尝试通过添加一个新的间接层来解决这个问题。在下一章中将会讨论这个问题的解决方案，其详细介绍了工厂模式及其如何启动服务。

第 4 章

服务

本章将学习如何使用服务 (*service*)，其是用来创建模块化代码库的一项重要功能。在学习了提供和消费服务的基本知识之后，会将所学的知识运用到 EasyText 应用程序中，使其更具有可扩展性。

4.1 工厂模式

在前一章中已经看到，当需要创建真正的解耦模块时，仅仅依靠封装是不够的。如果仍然编写以下代码：

```
MyInterface i = new MyImpl();
```

每次需要使用一个实现类，就意味着必须导出实现类。因此，实现类的消费者和提供者之间仍然存在强耦合：消费者要求提供者模块直接使用其导出的实现类。这样一来，实现过程中的任何更改都会直接影响到所有消费者。很快你就会看到，服务是解决这个问题的最佳方案。但在学习服务之前，先来看一下，是否可以根据目前对模块系统的了解使用现有的模式来解决这个问题。

工厂模式 (*factory pattern*) 是一个著名的创建型设计模式 (*creational design pattern*)，似乎解决了目前所面临的问题，其目标是将对象的消费者与特定类的实例化进行解耦。自从 Gamma 等人首次在《*Gang of Four Design Patterns*》(Addison-Wesley 出版社出版) 一书中描述了工厂模式以来，该模式已经发生了许多变化。接下来尝试实现一下该模式的简单变体，并看看它如何实现模块解耦。

下面再次使用 EasyText 应用程序作为演示示例，为 Analyzer 实例实现一个工厂类。获取给定算法名称的实现是非常简单的，如示例 4-1 所示。

示例 4-1: Analyzer 实例的工厂类 (chapter4/easytext-factory)

```
public class AnalyzerFactory {  
  
    public static List<String> getSupportedAnalyses() {  
        return List.of(FleschKincaid.NAME, Coleman.NAME);  
    }  
  
    public static Analyzer getAnalyzer(String name) {  
        switch (name) {  
            case FleschKincaid.NAME: return new FleschKincaid();  
            case Coleman.NAME: return new Coleman();  
            default: throw new IllegalArgumentException("No such analyzer!");  
        }  
    }  
}
```

可以从工厂类获取所支持的算法列表，并请求算法名称对应的 Analyzer 实例。现在，AnalyzerFactory 的调用者完全不知道分析器的任何底层实现类。

应该将该工厂模块放置在哪里呢？一方面，工厂模块本身仍然需要访问带有实现类的多个分析模块，否则，就无法在 getAnalyzer 中实例化各种实现类。可以将工厂模块放在 API 模块中，但这样一来，API 模块将与所有的实现模块之间存在编译时依赖关系，这不符合要求。API 不应与其实现紧密耦合。

接下来，将工厂模块暂时放置到自己的模块中，如图 4-1 所示。

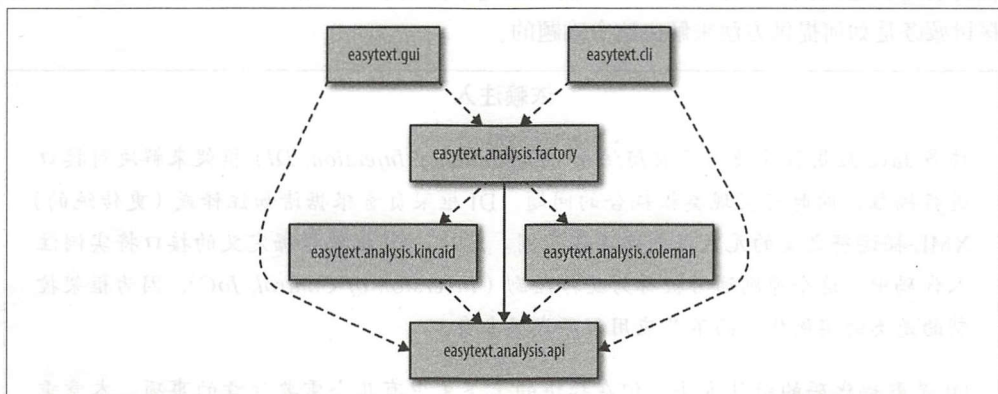


图 4-1：工厂模块将前端与分析实现模块分离开来（前端模块与分析实现模块之间不存在任何需求关系）

现在，前端模块仅知道 API 和工厂模块：

```
module easytext.cli {  
    requires easytext.analysis.api;
```

```
    requires easytext.analysis.factory;  
}
```

获取 Analyzer 示例变得非常简单:

```
Analyzer analyzer = AnalyzerFactory.getAnalyzer("Flesch-Kincaid");
```

除了增加了复杂性之外,使用工厂方法可以获得什么好处吗?

当然有好处。一方面,前端模块现在完全不需要了解分析模块以及实现类。消费者和分析提供者之间不存在直接的 `requires` 关系。前端模块可以独立于分析实现模块进行编译。当工厂提供了额外的分析时,前端可以非常容易地使用它们,而无须进行任何修改。(请记住,通过使用 `AnalyzerFactory::getSupportedAnalyses`,可以根据算法名称来请求实例。)

另一方面,在工厂模块以及以下级别,仍然存在相同的紧耦合问题。每当出现一个新的分析模块时,工厂就需要依赖它来扩展 `getAnalyzer` 实现。分析模块仍然需要导出其实现类,以便工厂使用。可以通过对工厂模块的限制导出(如 2.6 节中所述)来限制暴露范围,但是这样做的前提是分析模块“了解”工厂模块,而这是另一种不必要的耦合形式。

所以工厂模式只提供了部分解决方案。如果只是使用 `requires` 和 `exports` 关系,那么消费模块可以完成的事情是有限制的。虽然对接口进行编程非常好,但是我们必须牺牲封装来创建实例。幸运的是,Java 模块系统中提供了一个解决方案。在下一节中,将探讨服务是如何提供方法来解决这个难题的。

依赖注入

许多 Java 应用程序使用了依赖注入 (*Dependency Injection, DI*) 框架来解决对接口进行编程,同时与实现类松耦合的问题。DI 框架负责根据诸如注释或(更传统的)XML 描述符之类的元数据创建实现实例。然后,DI 框架根据定义的接口将实例注入代码中。这个原则通常被称为反转控制 (*Inversion of Control, IoC*),因为框架控制的是类的实例化,而不是应用程序代码本身。

DI 是解耦代码的绝佳方法,但在模块的上下文中有几个需要注意的事项。本章重点介绍了以服务形式进行解耦的模块系统解决方案。服务通过其他方式提供了 IoC (而不是通过依赖注入)。稍后,在 6.1.3 节中,将会学习如何将 DI 框架与模块结合使用,以实现相同级别的解耦。

4.2 用于实现隐藏的服务

前面尝试使用工厂模式来隐藏实现类，但仅取得了部分成功。主要的问题是工厂模块在编译时仍然必须知道所有可用的实现，并且必须导出实现类。这种解决方案与传统的通过扫描发现实现的类路径方法类似，无法解决目前的问题，因为仍然需要读取所有模块中的所有实现类。在不更改代码和重新编译的情况下，想要使用另一个实现（类似 EasyText 的新算法）来扩展应用程序也是不可能的。这听起来根本不像是无缝扩展！

解耦过程可以通过 Java 模块系统中的服务机制进行改进。使用服务，可以真正地共享公共接口，并将实现代码强封装到未导出的包中。请记住，与强封装（带有显式导出）和显式依赖（根据需要）不同，是否使用模块系统中的服务是完全可选的。可以不必使用服务，但服务却提供了一种令人信服的解耦模块的方法。

通过使用 `ServiceLoader` API 可在模块描述符和代码中表示服务。从这个意义上讲，使用服务是具有侵扰性的：需要设计一下应用程序才能使用它们。如 6.1.3 节中所述，除了使用服务之外，还可以使用其他方法来实现控制反转。在本章的其余部分，将学习服务如何带来更好的解耦和可扩展性。

接下来重新构建 EasyText 应用程序，以便使用服务，目标是让几个模块提供分析实现。前端模块可以在编译时不知道提供者模块的情况下使用这些分析实现。

4.2.1 提供服务

如果没有来自模块系统的特殊支持，想要将服务实现暴露给另一个模块而又不导出实现类是不可能的。Java 模块系统允许在 `module-info.java` 中添加提供和消费服务的声明性描述。

在 EasyText 代码中，已经定义了 `Analyzer` 接口，这就是服务接口类型。该接口由 `easytext.analysis.api` 模块（从严格意义上讲，该模块仅是一个 API 模块）导出。

```
package javamodularity.easytext.analysis.api;

import java.util.List;

public interface Analyzer {

    String getName();

    double analyze(List<List<String>> text);

}
```

通常，服务类型是一个接口（就像本示例所示的那样），但也可以是一个抽象甚至具

体的类，对此没有固有的技术限制。此外，Analyzer 类型可以直接由服务消费者使用，也可以公开类似于工厂或代理的服务类型。例如，如果 Analyzer 的实例化成本非常昂贵，或者需要额外的步骤或参数进行初始化，那么服务类型可能更加类似于 AnalyzerFactory。这种方法允许消费者对实例化进行更多的控制。

现在，重新构建第一个新的分析器实现，将 Coleman-Liau 算法（由 easytext.algorithm.coleman 模块提供）应用于服务提供者。只需要更改 module-info.java 即可，如示例 4-2 所示。

示例 4-2: 提供了 Analyzer 服务的模块描述符 (`chapter4/easytext-services`)

```
module easytext.analysis.coleman {  
  
    requires easytext.analysis.api;  
  
    provides javamodularity.easytext.analysis.api.Analyzer  
        with javamodularity.easytext.analysis.coleman.ColemanAnalyzer;  
  
}
```

`provides with` 语法声明该模块提供了 Analyzer 接口的一个实现（使用 ColemanAnalyzer 作为实现类）。服务类型（`provides` 之后）和实现类（`with` 之后）必须是完全限定类型名称。最重要的是，包含 ColemanAnalyzer 实现类的包并不从此提供者模块中导出。

只有当模块声明 `provides` 可以访问服务类型和实现类时，该结构才会起作用。通常这意味着一个接口（此示例为 Analyzer）要么是模块的一部分，要么是由所需的另一个模块导出。一般来说，实现类是提供者模块的一部分，位于一个封装（未导出）包中。

当在 `provides` 子句中使用了不存在或者无法访问的类型时，就无法对模块描述符进行编译，从而产生编译器错误。声明的 `with` 部分中所使用的实现类通常不导出。毕竟，使用服务的目的是隐藏实现细节。

不需要对服务类型或者实现类进行任何代码修改就可以将其作为服务提供。除了这个 `module-info.java` 声明，不需要做其他任何事情了。服务实现是普通的 Java 类，没有使用特殊的注释，也没有实现任何 API。

服务允许一个模块向其他模块提供实现，而不导出具体的实现类。模块系统具有访问提供者模块的特殊权限，从而代表消费者实例化非导出的实现类。这意味着服务的消费者可以使用此实现类的实例，而无须直接访问该实现类。此外，服务消费者不知道哪个模块提供了所需的实现，也不需要知道。因为提供者和消费者之间唯一的共享类型是服务

类型（通常是一个接口），所以实现了真正的解耦。

现在已经完成了第一个服务，可以针对其他 `Analyzer` 实现重复这个过程，且目前完成了一半的工作。同样，请注意，这些提供服务的模块没有导出任何包。乍一看，没有任何导出的模块似乎有点违反常理。然而，这些分析实现模块通过运行时的服务机制提供了有用的功能，而在编译时封装了实现细节。

剩下的另一半工作是消费服务。接下来重新构建 CLI 模块，以便使用 `Analyzer` 服务。

4.2.2 消费服务

如果其他模块可以消费服务，那么提供服务是非常有用的。在 Java 模块系统中，消费服务需要两个步骤。第一步是向 CLI 模块的 `module-info.java` 中添加 `uses` 子句：

```
module easytext.cli {
    requires easytext.analysis.api;

    uses javamodularity.easytext.analysis.api.Analyzer;
}
```

`uses` 子句告知 `ServiceLoader`（稍后将介绍该 API）该模块想要消费 `Analyzer` 的实现，然后 `ServiceLoader` 使 `Analyzer` 实例可用于模块。

`uses` 子句并不要求 `Analyzer` 实现在编译期间可用，毕竟服务实现可以由在编译时并不在模块路径上的模块提供。服务之所以提供了可扩展性，正是因为提供者 and 消费者仅在运行时才被绑定在一起。即使没有找到服务提供者，编译也不会失败。另一方面，服务类型 (`Analyzer`) 必须在编译时可访问，因此在模块描述符中需要使用 `requires easytext.analysis.api` 子句。

`uses` 子句同样无法保证在运行时存在提供者。即使没有任何服务提供者，应用程序也会成功启动。这意味着在运行时可能会提供零个或多个提供者，因此代码必须能够处理相应的情况。

现在，模块已经声明了要消费 `Analyzer` 实现，接下来可以开始编写使用该服务的代码。可以通过 `ServiceLoader` API 消费服务。从 Java 6 开始，`ServiceLoader` API 就已经存在了。虽然它在 JDK 中被广泛使用，但是很少有 Java 开发人员知道或使用 `ServiceLoader`，随后“Java 9 之前的 `ServiceLoader`”内容提供了更多的历史背景。

Java 模块系统对 `ServiceLoader` API 进行了略微的修改，以便使用模块，并且其是使用 Java 模块系统时一个非常重要的编程结构。接下来看一个示例（如示例 4-3 所示）。

示例 4-3: Main.java

```
Iterable<Analyzer> analyzers = ServiceLoader.load(Analyzer.class); ❶  
for (Analyzer analyzer: analyzers) { ❷  
    System.out.println(analyzer.getName() + ": " + analyzer.analyze(sentences));  
}
```

❶ 为服务类型 `Analyzer` 初始化 `ServiceLoader`。

❷ 遍历实例，调用 `analyze` 方法。

`ServiceLoader::load` 方法返回一个实现了 `Iterable` 的 `ServiceLoader` 实例。当在示例中进行遍历时，将为针对所请求的 `Analyzer` 接口所发现的所有提供者类型创建实例。请注意，此时仅获取实际的实例，而没有获取关于哪些模块提供了这些实例的其他信息。

遍历服务后，就可以像任何其他 Java 对象一样使用它们了。实际上，它们都是普通的 Java 对象，只不过是由 `ServiceLoader` 代我们实例化而已。作为普通的 Java 实例，调用服务的开销为零。调用服务上的方法只是一个直接的方法调用，没有代理或其他降低性能的间接方法。

完成了上述更改之后，将 `EasyText` 代码从部分解耦的工厂结构重构为完全模块化和可扩展的结构，如图 4-2 所示。

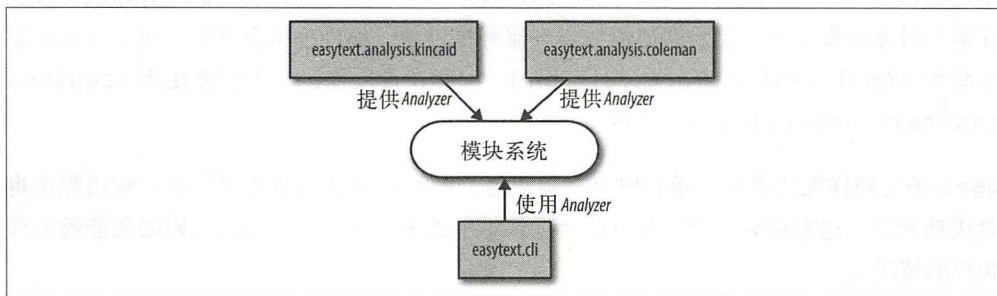


图 4-2: 使用 `ServiceLoader` 实现扩展的 `EasyText` 结构

此时代码完全解耦，因为 CLI 模块不需要知道关于提供 `Analyzer` 实现的模块的任何信息。该应用程序易于扩展，因为只需向模块路径添加新的提供者模块就可以添加新的 `Analyzer` 实现。由这些附加模块所提供的任何服务都将通过 `ServiceLoader` 服务发现程序自动获取，无须更改代码或重新编译。可以说，示例中最妙的是代码非常干净。虽然使用服务进行编程就像编写普通的 Java 代码一样简单（这是因为服务本身也是普通的 Java 代码），但对架构和设计的影响却是非常积极的。

Java 9 之前的 ServiceLoader

从 Java 6 开始, `ServiceLoader` 就已经存在了。其设计目的是让 Java 具有更好的可插拔性, 并在 JDK 的多个地方使用。尽管除了 JDK 本身之外, 还有几个框架和库依赖于旧的 `ServiceLoader`, 但它在应用程序开发中从未得到广泛的应用。

从原则上, `ServiceLoader` 与 Java 模块系统中的服务具有相同的目标, 但机制却是不同的, 且在模块出现之前没有真正意义上的强封装。为了注册一个提供者, 必须在 JAR 文件的 `META-INF` 文件夹中创建遵循特定命名方案的文件。例如, 为了提供一个 `Analyzer` 实现, 必须创建一个名为 `META-INF/services/javamodularity.easytext.analysis.api.Analyzer` 的文件。文件的内容应该是表示实现类的完全限定名称的单行, 如 `javamodularity.easytext.analysis.coleman.ColemanAnalyzer`。因为这些文件只是文本文件, 所以很容易造成编译器无法捕获的错误。

即使使用了 Java 模块系统, 也可以使用提供了“旧”方式的服务, 只要消费模块可以访问服务类型即可。

如前所示, 服务提供了一种简单的方法来实现解耦, 可以将服务视为模块化开发的基石。尽管确定模块边界的强大机制是迈向模块化设计的第一步, 但是仍需要通过服务来创建和使用严格解耦的模块。

4.2.3 服务生命周期

如果 `ServiceLoader` 负责创建所提供服务的实例, 那么了解它的工作方式就显得非常重要了。在示例 4-3 中, 遍历导致 `Analyzer` 实现类被实例化。`ServiceLoader` 工作“比较懒惰”, 这意味着 `ServiceLoader::load` 调用不会立即实例化所有已知的提供者实现类。

每次调用 `ServiceLoader::load` 时, 都会实例化一个新的 `ServiceLoader`。当请求提供者类时, 新的 `ServiceLoader` 会重新实例化它们。从现有的 `ServiceLoader` 实例请求服务将返回提供者类的缓存实例。

如下面代码所示:

```
ServiceLoader<Analyzer> first = ServiceLoader.load(Analyzer.class);
System.out.println("Using the first analyzers");
for (Analyzer analyzer: first) { ❶
    System.out.println(analyzer.hashCode());
}
```

```

Iterable<Analyzer> second = ServiceLoader.load(Analyzer.class);
System.out.println("Using the second analyzers");
for (Analyzer analyzer: second) { ❷
    System.out.println(analyzer.hashCode());
}

System.out.println("Using the first analyzers again, hashCode is the same");
for (Analyzer analyzer: first) { ❸
    System.out.println(analyzer.hashCode());
}

first.reload(); ❹
System.out.println("Reloading the first analyzers, hashCode is different");
for (Analyzer analyzer: first) {
    System.out.println(analyzer.hashCode());
}

```

- ❶ 遍历 first, ServiceLoader 实例化 Analyzer 实现。
- ❷ 新的 ServiceLoader——second 将实例化自己的 Analyzer 实现, 并返回不同于 first 的实例。
- ❸ 当再次遍历时, 从 first 返回最初实例化的服务, 因为第一个 ServiceLoader 实例缓存了这些服务。
- ❹ reload 之后, 最初的 ServiceLoader——first 提供了新的实例。

上述代码输出了如下所示内容 (当然, 实际输出的散列码可能会有所不同):

```

Using the first analyzers
1379435698
Using the second analyzers
876563773
Using the first analyzers again, hashCode is the same
1379435698
Reloading the first analyzers, hashCode is different
87765719

```

因为每次调用 `ServiceLoader::load` 都会生成新的服务实例, 所以使用相同服务的不同模块都拥有自己的实例。在使用包含状态的服务时, 需要铭记这一点。对于相同的服务类型, 如何没有其他规定, 那么状态在不同的 ServiceLoader 用法之间是不能共享的。与依赖注入框架中的典型情况不同, 没有单独的服务实例。

4.2.4 服务提供者方法

可以通过两种方式创建服务实例, 或者服务实现类必须具有公共的无参数构造函数, 或者使用静态提供者方法。一个服务实现类并不总是需要有一个公共的无参数构造函数。当需要向构造函数传递更多信息时, 静态提供者方法是更好的选择。或者, 也可以公开一个不带有无参数构造函数的现有类作为服务。

提供者方法是一个名为 `provider` 的 `public static` 无参数方法，其返回类型是服务类型。它必须返回正确类型（或子类型）的服务实例。在提供者方法中如何实例化服务完全取决于 `provider` 的实现，可能缓存并返回单一实例，或者为每个调用实例化一个新的服务实例。

当使用提供者方法时，`provides...with` 子句指的是包含提供者方法的类（`with` 之后）。这个类很可能是服务实现类本身，也可能是另一个类。出现在 `with` 后面的类必须拥有一个提供者方法或一个公共的无参数构造函数。如果没有提供静态提供者方法，则该类被假定为服务实现类本身，并且必须有一个公共的无参数构造函数。否则编译器就会产生“抱怨”。

接下来看一个提供者方法示例（见示例 4-4）。为了突出提供者方法的使用，将使用另一个 `Analyzer` 实现。

示例 4-4: `ExampleProviderMethod.java` ([↗ chapter4/providers/provider.method.example](#))

```
package javamodularity.providers.method;

import java.util.List;
import javamodularity.easytext.analysis.api.Analyzer;

public class ExampleProviderMethod implements Analyzer {

    private String name;


    ExampleProviderMethod(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public double analyze(List<List<String>> sentences) {
        return 0;
    }

    public static ExampleProviderMethod provider() {
        return new ExampleProviderMethod("Analyzer created by static method");
    }
}
```

虽然该 `Analyzer` 实现没有什么用处，但演示了提供者方法的使用。示例中的 `module-info.java` 与我们到目前为止所看到的完全一样；Java 模块系统将会找出正确的方法来实例化类。在本示例中，`provider` 方法是实现类的一部分。或者也可以将 `provider` 方法放在另一个类中，然后以该类作为服务实现的工厂类。示例 4-5 显示了这种方法。

示例 4-5: ExampleProviderFactory.java ( chapter4/providers/provider.factory.example)

```
package javamodularity.providers.factory;

public class ExampleProviderFactory {
    public static ExampleProvider provider() {
        return new ExampleProvider("Analyzer created by factory");
    }
}
```

现在，必须对 `module-info.java` 进行修改，从而反映上面所做的更改。`provides...with` 必须指向包含该静态提供者方法的类，如示例 4-6 所示。

示例 4-6: module-info.java ( chapter4/provides/provider.factory.examples)

```
module provider.factory.example {
    requires easytext.analysis.api;

    provides javamodularity.easytext.analysis.api.Analyzer
        with javamodularity.providers.factory.ExampleProviderFactory;
}
```




只有在提供者类是公共的情况下，`ServiceLoader` 才可以实例化一个服务。只有提供者类本身需要是公共的；第二个示例表明只要提供者类是公共的，实现类就可以是包私有的。

请注意，在所有情况下，公开的服务类型 `Analyzer` 是保持不变的。从消费者的角度来看，服务的实例化并没有什么不同，但静态提供者方法提供了更大的灵活性。在大多数情况下，服务实现类上的公共的无参数构造函数已经足够了。

模块系统中的服务没有提供关闭或服务注销机制。一个服务的死亡是隐式的，是通过垃圾回收机制完成的。对于服务实例来说，对其实施垃圾回收的行为与 Java 中的任何其他对象一样。一旦对象没有被强引用 (`hard reference`)，那么它就会被作为垃圾回收。

4.3 工厂模式回顾

消费者模块可以通过 `ServiceLoader` API 获取服务。如果需要，也可以使用有用的模式来避免使用此 API。或者，可以像本章开头的工厂示例中那样向消费者提供一个 API，这基于从 Java 8 开始可以在接口中使用静态方法。服务类型本身也是使用执行 `ServiceLoader` 查找的静态方法（工厂方法）进行扩展的，如示例 4-7 所示。

示例 4-7: 在服务接口上提供一个工厂方法 ( chapter4/easytext-services-factory)

```
public interface Analyzer {
    String getName();
}
```

```

double analyze(List<List<String>> text);

static Iterable<Analyzer> getAnalyzers() {
    return ServiceLoader.load(Analyzer.class); ❶
}
}

```

❶ 此时，查询在服务类型内部完成。

由于 `ServiceLoader` 查找是在 API 模块的 `Analyzer` 内完成的，因此其模块描述符必须使用 `uses` 约束：

```

module easytext.analysis.api {
    exports javamodularity.easytext.analysis.api;

    uses javamodularity.easytext.analysis.api.Analyzer;
}

```

现在，API 模块导出了接口并使用了 `Analyzer` 接口的实现。想要获得 `Analyzer` 实现的消费者模块不再需要使用 `ServiceLoader` 了（当然，仍然可以使用 `ServiceLoader`），取而代之，所有消费者模块需要做的就是请求 API 模块，并调用 `Analyzer::getAnalyzers`。从消费者角度来看，不再需要 `uses` 约束或 `ServiceLoader` API 了。

通过这种机制，可以悄无声息地使用服务的强大功能，不再强迫 API 用户去了解服务或 `ServiceLoader`，与此同时仍然可以获得解耦和可扩展性所带来的好处。

4.4 默认服务实现

到目前为止都是假设有一个 API 模块，并且有几个不同的提供者模块实现了这个 API。虽然该假设不无道理，但并不是唯一的方法，将实现放在导出服务类型的同一模块中是完全可能的。当一个服务类型具有默认实现时，为什么不直接从同一个模块中提供它呢？

在 JDK 自身使用服务的方式中会经常看到这种模式。即使可以为 `javax.sound.sampled.spi.Audio FileWriter` 或 `javax.print.PrintServiceLookup` 提供自己的实现，大多数情况下 `java.desktop` 模块所提供的默认实现也是足够的。这些服务类型从 `java.desktop` 导出，同时提供了默认的实现。

事实上，`java.desktop` 自身对这些服务类型也使用了 `uses` 约束。这表明一个模块可以同时扮演 API 所有者、服务提供者和服务消费者的角色。

将默认的服务实现与服务类型进行绑定可以确保至少有一个实现始终可用。在这种情况下

下，消费者不需要编写防御性代码，且某些服务依赖项是可选的。与服务类型处在同一模块中的默认实现排除了这种情况。此时需要一个单独的 API 模块。在 5.6.2 节会更详细地探讨该模式。

4.5 服务实现的选择

当存在多个提供者时，不一定要全部使用。可以根据某些特性筛选一种实现。



通常是消费者根据提供者的无属性决定使用哪种服务。因为提供者之间无法完全“彼此了解”，所以从提供者的角度来看无法支持某种实现。例如，如果两个提供者将自己指定为默认或最佳实施，那么会出现什么情况呢？选择正确的服务要依赖于应用程序，是由消费者确定的。

已经看到，`ServiceLoader` API 自身是有局限性的。到目前为止，仅对所有现有的服务实现进行了循环遍历。如果存在多个提供者，但只对“最好的”实现感兴趣，该怎么做呢？Java 模块系统不可能知道适合你的最佳实现。每个领域都有自己的需求。因此，将由自己根据服务类型配备方法，从而发现服务的功能，并根据这些方法进行决策。这个过程并不复杂，通常是将自描述方法添加到服务接口中。

例如，`Analyzer` 服务接口提供了一个 `getName` 方法。虽然 `ServiceLoader` 不知道或不关心该方法，但可以在消费者模块中使用它来识别一个实现。除了根据名称选择算法外，还可以考虑描述不同的特征，如使用 `getAccuracy` 或 `getCost` 方法。这样一来，`Analyzer` 服务的消费者可以在实现之间做出良好的选择。`ServiceLoader` API 没有必要显式支持：这些都归结于设计自描述接口。

服务类型检查和延迟实例化

在某些情况下，前面所描述的机制还远远不够。如果服务接口上没有方法来区分正确的实现，或者实例化服务的代价非常昂贵，那么应该怎么办呢？为了找到正确的服务实现（使用 `ServiceLoader` 循环遍历），必须承担所有服务实现的初始化成本。在大多数情况下，这并不是一个问题，但可以使用一种解决方案来解决该问题。

在 Java 9 中，`ServiceLoader` 的功能得到了增强，支持在实例化之前对服务实现类型进行检查。除了遍历目前所有的实例以外，还可以检查一连串的 `ServiceLoader.Provider` 描述。`ServiceLoader.Provider` 类使得在请求实例之前检查服务提供者成为可能。`ServiceLoader` 上的 `stream` 方法返回一串要检查的 `ServiceLoader.Provider` 对象。

接下来，再次以 EasyText 为基础查看一个示例。

首先，介绍一下在示例 4-8 中可用来选择正确的服务实现的注释。此注释可以是提供者
和消费者之间共享 API 模块的一部分。示例注释描述了 Analyzer 是否快速。

示例 4-8：定义用来注释服务实现类的注释（[↪ chapter4/easytext-filtering](#)）

```
package javamodularity.easytext.analysis.api;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Fast {

    public boolean value() default true;
}
```

现在，可以使用该注释向服务实现添加元数据。此时将其添加到示例 Analyzer 中：

```
@Fast
public class ReallyFastAnalyzer implements Analyzer {
    // analyzer 的实现
}
```

现在，只需添加一些代码就可以过滤 Analyzer：

```
public class Main {
    public static void main(String args[]) {
        ServiceLoader<Analyzer> analyzers =
            ServiceLoader.load(Analyzer.class);

        analyzers.stream()
            .filter(provider -> isFast(provider.type()))
            .map(ServiceLoader.Provider::get)
            .forEach(analyzer -> System.out.println(analyzer.getName()));
    }

    private static boolean isFast(Class<?> clazz) {
        return clazz.isAnnotationPresent(Fast.class)
            && clazz.getAnnotation(Fast.class).value() == true;
    }
}
```

通过使用 Provider 上的 type 方法，可以获取服务实现的 java.lang.Class 表示，
并将其传递给 isFast 方法，从而实现过滤。

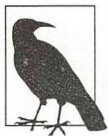
封装和 java.lang.Class

可以获取实现类的 `java.lang.Class` 表示看似非常奇怪。这不是违反了强封装吗？该类所在的包甚至没有被导出！

这是“地图不是领土”的一个明显的例子——即使 `Class` 描述了实现类，但无法使用它完成任何事情。当尝试通过反射（使用 `provider.type().newInstance()`）获取一个实例时，如果该类没有导出，那么将得到一个 `IllegalAccessException`。所以，拥有一个 `Class` 对象并不一定意味着可以在自己的模块中实例化它。模块系统的所有访问检查仍然适用。

`isFast` 方法检查 `@Fast` 注释是否存在，并显式检查该值是否为 `true`（这是默认值）。未被注释为快速的 `Analyzer` 实现将被忽略，而注释为 `@Fast` 或 `@Fast(true)` 的服务则被实例化和调用。如果从流管道中删除 `filter`，则可以任意地调用所有 `Analyzer`。

本章的示例表明，虽然 `ServiceLoader` API 是基础，但服务机制是强大的。当模块化代码时，服务是 Java 模块系统中重要的组成部分。



使用服务作为提高解耦性能的手段并不是一件新鲜的事，如 `OSGi` 也提供了基于服务的编程模型。要想在 `OSGi` 中创建真正的模块化代码，必须使用服务。所以上述内容都建立在一个成熟的概念之上。

4.6 具有服务绑定的模块解析

是否还记得在学习 2.7 节时曾讲过，模块是根据模块描述符中的 `requires` 子句进行解析的？从根模块开始以递归的方式寻找所有的 `requires` 关系，解析的模块集由模块路径上的模块构建。在此过程中可以检测到丢失的模块，从而提供了可靠的配置。如果缺少必需的模块，应用程序将无法启动。

服务的 `provides` 和 `uses` 子句为解析过程添加了另一个维度。`requires` 子句表示模块之间严格的编译时关系，而服务绑定则发生在运行时。因为服务提供者模块和消费者模块都在模块描述符中声明了各自的意图，所以在模块解析过程中也可以使用这些信息。

从理论上讲，即使在运行时没有绑定任何服务，应用程序也可以启动。此时调用 `ServiceLoader::load` 不会产生任何实例。这样做几乎没有任何用途，所以除了所

需的模块之外，模块系统还可以在启动时在模块路径上查找服务提供者模块。

当解析一个带有 `uses` 子句的模块时，模块系统将在模块路径上找到给定服务类型的所有提供者模块，并将其添加到解析过程中。这些提供者模块及其依赖项成为运行时模块图的一部分。

通过一个示例，可以更清楚地了解这个扩展对模块解析所产生的影响。在图 4-3 中，从模块解析的角度再看一下 EasyText 示例。

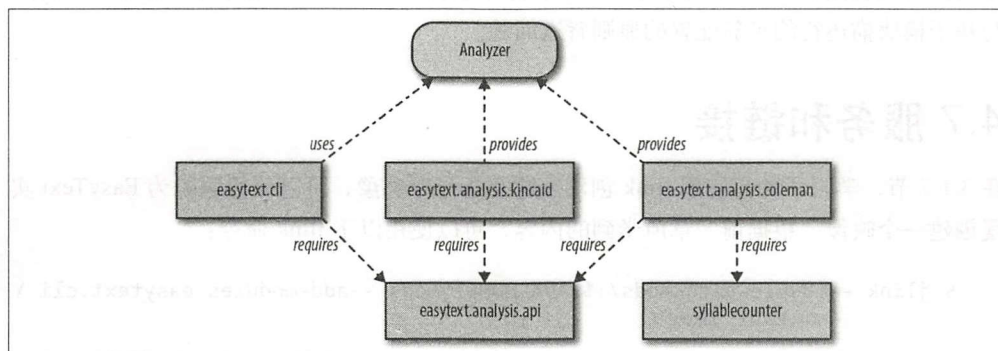


图 4-3：服务绑定影响模块解析

假设在模块路径上有五个模块：`cli`（根模块）、`api`、`kincaid`、`coleman` 以及一个虚构模块 `syllablecounter`。模块解析从 `cli` 开始。它与 `api` 存在需求关系，所以将 `api` 模块添加到已解析模块集中。到目前为止，没有什么新内容。

但是，`cli` 还有一个针对 `Analyzer` 的 `uses` 子句。在模块路径上，有两个实现了该接口的提供者模块。因此，提供者模块 `kincaid` 和 `coleman` 都被添加到已解析模块集中。由于再没有其他 `requires` 或 `uses` 子句，因此 `cli` 的模块解析停止。

对于 `kincaid` 模块来说，无须向已解析模块集添加任何内容。它所需要的 `api` 模块已经被解析。而对于 `coleman` 模块来说，事情更加有趣。服务绑定导致 `coleman` 被解析。在本示例中，`coleman` 模块需要另一个模块 `syllablecounter`。因此，`syllablecounter` 也被解析，并且添加到运行时模块图中。

如果 `syllablecounter` 自身具有 `requires`（或者 `uses`）子句，那么这些模块也会完成模块解析过程。相反，如果在模块路径上找不到 `syllablecounter`，则解析失败，应用程序将无法启动。即使消费者模块 `cli` 对提供者模块 `coleman` 没有任何了解，也可以通过服务绑定获取所有依赖项并完成解析。

对于消费者来说，则无法明确提出至少需要一个实现。当没有找到服务提供者模块时，

应用程序也会启动。使用 `ServiceLoader` 的代码需要考虑到这种可能性。前面已经看到许多 JDK 服务类型都具有默认实现。当模块中的默认实现公开了服务类型时，可以保证至少有一个可用的服务实现。

在示例中，即使 `coleman` 不在模块路径上，模块解析也会成功。此时，在运行时，`ServiceLoader::load` 调用只能从 `kincaid` 中查找服务实现。但是，正如前面所解释的那样，如果 `coleman` 在模块路径上，但 `syllablecounter` 不在，那么应用程序将因为模块解析失败而无法启动。对于模块系统来说，这个问题是可以忽略的，但却与基于模块描述符的可靠配置的原则背道而驰。

4.7 服务和链接

在 3.1.7 节，学习了如何使用 `jlink` 创建自定义运行时映像，可以使用服务为 `EasyText` 实现创建一个映像。根据前一章所学到的内容，可以使用以下 `jlink` 命令：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods --add-modules easytext.cli \
      --output image
```

`jlink` 创建了一个目录 `image`，其中包含了一个 `bin` 目录。可以使用下面的命令检查映像中所包含的模块：

```
$ image/bin/java --list-modules

java.base@9
easytext.analysis.api
easytext.cli
```

`api` 和 `cli` 模块是映像的一部分，但是两个分析提供者模块如何呢？如果以这种方式运行应用程序，它将正确启动，因为服务提供者是可选的。但是如何没有任何分析模块，这个应用程序是没有任何用处的。

`jlink` 从根模块 `easytext.cli` 开始执行模块解析，所有已解析的模块都包含在生成的映像中。但是，该解析过程与上一节所讨论的启动时由模块系统完成的解析过程是不同的。在模块解析期间，`jlink` 没有完成任何服务绑定，这意味着不会根据 `uses` 子句自动将服务提供者包含在映像中。

虽然对于那些不了解这一点的用户来说，这么做肯定会导致意想不到的结果，但这是一个经过深思熟虑的选择。服务通常用于提供可扩展性。`EasyText` 应用程序就是一个很好的示例，可以通过向模块路径添加新的服务提供者模块来添加新的算法类型。以这种方式使用的服务并不一定是运行应用程序所需的。要组合哪些服务提供者需要根据应用程序而定。在生成时，对服务提供者没有任何依赖，而在链接时，由所需映像的创建者决

定应该使用哪些服务提供者。



在 jlink 中不执行自动服务绑定的一个更切合实际的原因是 `java.base` 具有大量的 `uses` 子句。所有这些服务类型的提供者位于其他各种平台模块中，默认绑定所有这些服务将会增大映像的大小。如果 jlink 中没有执行自动服务绑定，就可以创建一个仅包含 `java.base` 和应用程序模块的映像，如本节示例。一般来说，自动服务绑定可能会产生意料之外的大型模块图。

接下来尝试为 EasyText 应用程序创建一个运行时映像，并通过命令行运行。为了包含分析程序，当为每个要添加的提供者执行 jlink 时，使用了参数 `--add-modules`：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods \  
  --add-modules easytext.cli \  
  --add-modules easytext.analysis.coleman \  
  --add-modules easytext.analysis.kincaid \  
  --output image  
  
$ image/bin/java --list-modules  
  
java.base@9  
easytext.analysis.api  
easytext.analysis.coleman  
easytext.analysis.kincaid  
easytext.cli
```

虽然一切看起来没有什么问题，但是当启动应用程序时仍然会发现一个问题：

```
$ image/bin/java -m easytext.cli input.txt
```

此时应用程序异常退出：`java.lang.IllegalStateException:SyllableCounter not found`。kincaid 模块使用了另一种 `SyllableCounter` 类型的服务。出现这种情况的原因是服务提供者使用了其他服务来实现其功能。前面已经讲过，jlink 不会自动包含服务提供者，所以包含 `SyllableCounter` 示例的模块也不会包括在内。再次使用 `--add-modules`，从而最终获得一个功能完整的映像：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods \  
  --add-modules easytext.cli \  
  --add-modules easytext.analysis.coleman \  
  --add-modules easytext.analysis.kincaid \  
  --add-modules easytext.analysis.naivesyllablecounter \  
  --output image
```

默认情况下，jlink 不包括服务提供者，因此在链接时需要完成一些额外的工作，尤其是当服务以传递的方式使用其他服务时。这样做的好处是可以非常灵活地微调运行时映像的内容。不同的映像可以为不同类型的用户提供服务，只需重新配置需要包含的服务提

供者即可。在 13.3 节，将会看到 jlink 提供了额外的选项来发现和链接相关联的服务提供者模块。

前面几章介绍了 Java 模块系统的基础知识。模块化主要关注的是设计和架构，这是它真正有趣的地方。下一章将学习一些模式，从而改善由模块构建的系统的可维护性、灵活性和可重用性。

模块化模式

从某些方面讲，掌握一门新技术或一项新语言功能感觉就像是掌握了一项新的超能力一样。此时，非常渴望立即看到超能力的潜力，并希望由此来改变世界。如果有什么区别的话，就是漫画中的超级英雄已经向我们展现了世界不是黑白分明的——即使你相信自己已经拥有了超能力。能力运用的好坏不会马上显现出来；往往能力越大，责任也就越大。

学习 Java 模块系统也是一样的。如果不通过了解模块化设计原则来掌握其功能，那么仅仅了解模块系统对你来说没有任何意义。模块化不仅仅是一个实现问题（该问题可通过引入新的语言特性来缓解），也是一个设计和架构的问题。应用模块化设计是一项长期的投资。通过模块化，可以应对需求、环境、团队以及其他不可预见事件所带来的变化。

本章将讨论通用模式，以提高使用模块所构建系统的可维护性、灵活性和可重用性。请记住，这些模式和设计实践中的大部分与技术无关。虽然本章提供了代码，但它们主要用于在 Java 模块系统上下文中说明这些抽象的模式，关注的重点应是如何通过应用已建立的模块化模式有效地模块化系统。

如果你对这些模式非常熟悉，那么恭喜了！你一直在从事模块化开发。尽管如此，Java 模块系统比以前的语言提供了更多的编写模块化代码的支持。这样做不仅仅是为了你，更是为了整个团队，甚至整个 Java 生态系统。而如果你是首次接触这些模式，那么也要恭喜你。通过学习和应用这些模式和设计实践，你开发的应用程序会变得更加容易维护和扩展。

本章将讨论应用程序开发中经常遇到的基本模块化模式。首先，从一些通用模块设计指南开始，然后再学习更具体的模块模式。第 6 章将会讨论更高级的模式，那些需要开发具有较高灵活性的应用程序的开发人员会对这些模式感兴趣，比如通用的应用程序容器或基于插件的系统。

5.1 确定模块边界

什么样的模块是一个好的模块？你会惊奇地发现，这个问题实际上由来已久。长久以来，将系统划分为小型的、可管理的模块已被认为是一项成功的策略。举个例子，下面的一段话引自 1972 年的一篇文章 (<http://bit.ly/parnas-on-the>):

“模块化”的有效性取决于将系统划分成模块所使用的标准。

—D.L. Parnas, 《*On the Criteria To Be Used in Decomposing Systems into Modules*》

该文提出的要点之一是在编写任何代码之前进行模块化。模块边界应该源自系统的设计和意图。在下面的栏目中，可以了解一下 Parnas 是如何应对这一挑战的。就像上面引文所说的那样，确定边界的标准决定了模块化工作的成功。那么这些标准是什么呢？通常，视情况而定。

Parnas 分区

根据 D.L.Parnas 在 1972 年的一篇文章中所述，他设计了一种称为 *Parnas 分区* (*Parnas partitioning*) 的模块化方法。在思考模块边界时，考虑到可能的变化总是一件好事。通过使用 Parnas 分区，可以构建一个隐藏假设列表 (*hiding assumption list*)。该列表包含了系统中预期会发生变化或存在设计决策争议的地方，以及日后变化的概率。根据该列表将功能划分为模块，可以最小化更改所带来的影响。隐藏假设列表中的高概率项是要封装在模块中的主要候选项。创建这样一个列表并不是什么难事，技术人员和非技术人员可以一起完成。

通过该引文 (<http://www.jodypaul.com/SWE/HAL/hal.html>)，可以了解更多关于构建隐藏假设列表的内容。

创建用于重用的模块化库与创建以可理解性和可维护性为主要关注点的大型企业应用程序之间存在着很大的区别。通常，在设计模块时可以划分几个参考标准：

- 可理解性

模块及其相互关系反映了系统的整体结构和意图。每当有人在没有先验知识的情况下查看代码库时，首先看到的是高级结构和功能。模块结构可以引导开发人员查找系统中特定的功能。

- 可变性

需求总是在不断地变化。通过使用模块来封装可能发生变化的决策，就会让变化所产生的影响降到最小。具有相似功能但具有不同预期变化范围的两个系统可具有不同的最佳

模块边界。

- *可重用性*

模块是理想的重用单元。为了提高可重用性，模块应该尽可能地集中并尽可能独立。可重复使用的模块可以在不同的应用程序中以多种方式组合。

- *团队合作*

有时，可以使用模块边界来明确划分多个团队的工作。可以将模块边界与组织边界对齐，而不需要考虑技术方面的问题。

这些标准之间存在一些冲突，这里无法给出一个适合所有人的答案。通过设计特定区域的变化，可以引入一些抽象概念，从而便于初次理解相关内容。比方说，将一个过程中的一个步骤放在一个单独的模块中，因为该步骤预计会比其他步骤更频繁地变化。从逻辑上讲，该步骤仍然属于主要过程，但它现在在一个单独的模块中。虽然从某种程度上讲这样做增加了整体的理解难度，但却增加了可变性。

另一个重要的权衡来自于重复使用和使用之间的冲突。通用组件或可重用库可能会变得非常复杂，因为它们需要适应不同的使用场景，而那些不需要重复使用的应用程序组件则可以更直接和更具体。

当设计重用时，可以考虑以下两点：

- 坚持 UNIX 哲学理念，只做一件事情，并且将其做好。
- 将模块本身的依赖关系数量减到最少。否则，所有重用的使用者都要承担可传递依赖关系所带来的负担。

以上两点并不一定适用于应用程序模块，对于应用程序模块来说，易用性、可理解性以及开发速度更为重要。如果使用一个库模块可加快应用程序模块的开发速度，并使应用程序模块更加简单，那么就可以这样做。另一方面，如果你创建了一个可重用的库模块，那么未来模块的使用者会感谢你并没有引入过多的传递性依赖关系。这里没有绝对的对与错，只有慎重的权衡。模块系统非常明确地做出了选择。

可重用模块通常比一次性使用的应用程序模块更小、更集中。另一方面，协调使用多个小模块也带来了一定的复杂性。当重复使用不是主要的关注点时，创建更大的模块可能更合理。

在下一节，将会探讨模块大小的概念。

5.2 精益化模块

模块应该有多大？这听起来像是一个很自然的问题，就好像是在问应用程序有多大一样。只要大到可以达到目标就可以了，但也不能太大——显然，这并不是一个非常有用的建议。

在考虑模块设计时除了大小之外，还有更重要的问题要解决。而在考虑模块的度量时，需要考虑两个指标：模块公共区域面积 (*surface area*) 的大小以及内部实现的大小。

出于两点理由，简化和最小化模块的公开导出部分是有益的。首先，一个简单而小巧的 API 比一个大而复杂的 API 更容易使用，模块用户不必考虑任何不必要的细节。模块化的主旨是将问题分解成可管理的块。

其次，最大限度地减少模块的公共部分可以减轻模块维护者的责任。维护者不需要考虑其他人无法访问的内容，模块作者可以自由地更改内部细节而不会产生严重的后果。公开的越少，消费者所依赖的也就越少，API 就越稳定。模块导出部分所放置的任何内容都将成为模块生产者和消费者之间的协议。如果要以向后兼容的方式（应该是默认方式）来发展模块，那么该问题是不容忽视的。建议尽量减少模块的公共区域面积。

还需要考虑另一个指标：模块非导出部分的大小。与模块的公共部分一样，在此谈论原始大小是没有多大意义的。模块的私有实现应该与实现其 API 约定所需的内容一样大。可更有趣的问题是：需要多少其他模块来完成目标呢？精益化模块要尽可能独立，在可能的情况下避免依赖于其他模块。如果只是想使用特定的模块，那么没有什么比看到系统中添加了大量的（可传递的）依赖项更让人沮丧的了。如前所述，当广泛的重复使用不是模块的主要关注点时，这就不再是一个问题了。

你可能注意到，开发精益化模块与围绕可重用微服务的最佳实践之间存在许多相似的地方：尽可能的小，与外界有定义良好的协议，同时尽可能保持独立。事实上，在系统架构中的不同层次上也存在类似问题。具有公共 API 和模块描述符的模块可以方便进程内重用和组合。通过（网络）进程间通信，微服务在架构的较高级别上起到了非常大的作用。因此，模块和微服务是互补的概念。微服务很可能在内部使用 Java 模块来实现。但两者之间的一个重要区别是，使用模块及其描述符不仅可以明确描述所提供（导出）的内容，还可以明确地描述所需要的内容，因此可以非常安全可靠地解析和链接 Java 模块系统中的模块，但在大多数微服务环境中却不是这样的。

5.3 API 模块

到目前为止已经看到，仔细研究模块的 API 是很有必要的，API 设计已经成为构建合适

模块的基本元素。这似乎听起来只有库的作者才会关注此问题，但事实并非如此。当对应用程序进行模块化时，构建应用程序模块的公共 API 是非常重要的。根据定义，模块的 API 指的是其导出包的总和。由模块组成的应用程序（这些模块导出了其所包含的所有内容）通常是一个警告信号，表明没有比使用模块更好的方法了。模块化应用程序隐藏了应用程序其他部分的实现细节，就像一个设计良好的库对应用程序隐藏了其内部实现细节。无论何时模块用在应用程序的哪个部分，或者由不同的团队使用，拥有一个定义良好且稳定的 API 是非常重要的。

5.3.1 API 模块中应该包含什么

如果希望一个接口只有一个实现，那么可以将 API 和实现组合到一个模块中。在这种情况下，导出的部分对模块的使用者是可见的，而实现包是隐藏的。可以通过使用 `ServiceLoader` 将这个实现作为一个服务公开。即使希望有多个实现，在某些情况下将默认实现捆绑到 API 模块中也是很有意义的。在本节的后部分假设 API 模块不包含这样的默认实现，而是独立存在。在 5.3.3 节中，将讨论模块包含导出的 API 和该 API 实现时的一些注意事项。

前面已经讲过，模块的公共部分应该尽可能地精益化。但是，应该从 API 模块中导出什么内容？前面已经多次提到过接口，因为它们构成了大多数 API 的骨架。当然，还有其他内容。

接口包含了带有参数和结果类型的方法。在最基本的形式中，接口是独立的，只使用 `java.base` 中的类型：

```
public interface SimpleTextRepository {
    String findText(String id);
}
```

实际上，拥有一个如此简单的接口是非常少见的。在诸如 `EasyText` 之类的应用程序中，希望完成一个存储库实现，通过 `get Text` 返回一个特定域的类型（参见示例 5-1）。

示例 5-1：带有非原始类型的接口

```
public interface TextRepository {
    Text findText(String id);
}
```

此时，`Text` 类被放置在 API 模块中。它可能是一个典型的 `JavaBean` 风格的类，描述了调用者期望从服务中获得的数据。因此，它是公共 API 的一部分。在接口的方法中声明的异常也是 API 的一部分。它们应该放在 API 模块中，并与（声明）抛出它们的方法一起导出。

接口是实现 API 提供者和消费者之间解耦的主要手段。当然，API 模块可以包含比接口更多的内容。例如，希望 API 消费者扩展的（抽象）基类、枚举、注释等。无论在 API 模块中放入什么内容，都要记住：简约大有裨益。

当另一个模块需要 API 模块时，你希望该模块是可用的，但并不希望为了读取接口中所使用的所有类型而使用额外的模块。如上所述，使 API 模块完全自包含是实现这一目标的一种方法。不过，该方法并不总是可行的。通常，接口方法返回或接收另一个模块中的类型参数。为了简化这种情况，模块系统提供了隐式可读性。

5.3.2 隐式可读性

2.5 节介绍了基于平台模块的隐式可读性。接下来看一个来自 EasyText 域的示例，了解一下隐式可读性如何帮助创建自包含和完全自描述的 API 模块。图 5-1 所示的示例由三个模块组成。

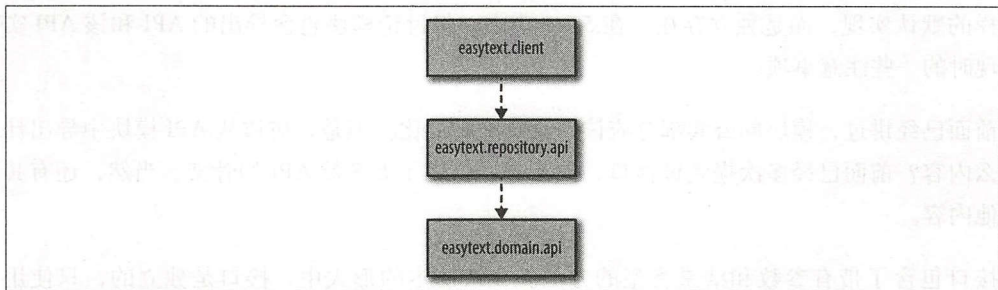


图 5-1：三个模块，但没有隐式可读性

在本示例中，TextRepository 接口位于模块 easytext.repository.api 模块中，而 findText 方法返回的 Text 类位于另一个模块 easytext.domain.api 中。easytext.client (调用 TextRepository) 的 module-info.java 如示例 5-2 所示。

示例 5-2：使用了 TextRepository 的模块的模块描述符 (chapter5/implied_readability)

```
module easytext.client {
    requires easytext.repository.api; ❶

    uses easytext.repository.api.TextRepository; ❷
}
```

❶ 需要 API 模块，因为需要访问该模块中的 TextRepository。

❷ 表明客户端想要使用一个实现了 TextRepository 的服务。

easytext.repository.api 又依赖于 easytext.domain.api，因为它使用了 Text 作为 TextRepository 接口中的返回类型：

```

module easytext.repository.api {
    exports easytext.repository.api; ❶
    requires easytext.domain.api; ❷
}

```

❶ 公开包含了 TextRepository 的 API 包。

❷ 需要域 API 模块，因为它包含了 TextRepository 接口所引用的 Text。

最后，easytext.domain.api 模块包含了 Text 类：

```

public class Text {
    private String theText;

    public String getTheText() {
        return this.theText;
    }

    public void setTheText(String theText) {
        this.theText = theText;
    }

    public int wordcount() {
        return 42; // Why not
    }
}

```

请注意，Text 拥有一个 wordcount 方法，在稍后的客户端代码中将会使用该方法。

Easytext.domain.api 模块导出了包含 Text 类的包：

```

module easytext.domain.api {
    exports easytext.domain.api;
}

```

客户端模块包含了以下对该存储库的调用：

```

TextRepository repository = ServiceLoader.load(TextRepository.class)
    .iterator().next();

repository.findText("HHGTTG").wordcount();

```

如果进行编译，编译器会产生如下所示错误：

```

./src/easytext.client/easytext/client/Client.java:13: error: wordcount() in
Text is defined in an inaccessible class or interface
    repository.findText("HHGTTG").wordcount();
                                ^

```

尽管没有直接在 easytext.client 中提到 Text 类型，但却试图调用这个类型的方法，因为它是从存储库返回的。因此，客户端模块需要读取导出了 Text 的 easytext.domain.api 模块。解决这个编译错误的一种方法是在客户端的模块描述

符中添加一个 `requires easytext.domain.api` 子句，但这并不是一个好的解决方案。为什么客户端模块必须处理库模块的传递依赖关系呢？更好的解决方案是改进存储库的模块描述符：

```
module easytext.repository.api {
    exports easytext.repository.api;
    requires transitive easytext.domain.api; ❶
}
```

❶ 通过添加 `transitive` 关键字设置隐式可读性。

请注意 `exports` 子句中额外的关键字 `transitive`。实际上是在说，存储库模块读取 `easytext.domain.api`，并且每个需要 `easytext.repository.api` 的模块也会自动读取 `easytext.domain.api`，如图 5-2 所示。

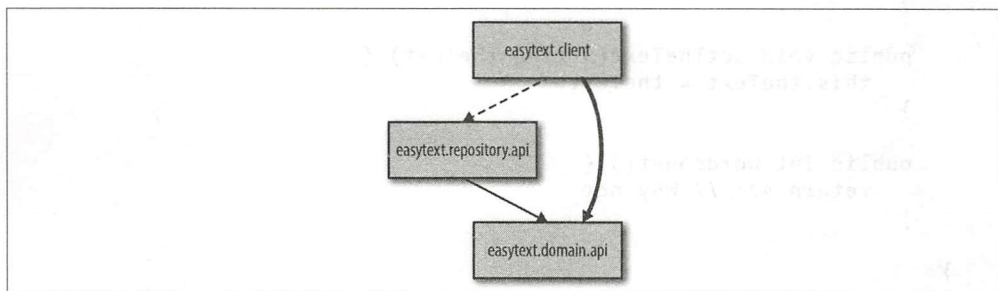


图 5-2：使用 `requires transitive` 设置隐式可读性（如粗边所示）

现在，示例可以顺利编译。通过存储库的模块描述符中的 `requires transitive` 子句，客户端模块可以读取 `Text` 类。隐式可读性允许存储库模块表达其自己导出的包不足以使用该模块。

上面所示的是一个返回类型来自不同模块的例子（需要使用 `requires transitive`）。只要一个公共导出类型引用了来自另一个模块的类型，就会使用隐式可读性。除了返回类型之外，也适用于参数类型以及从不同模块抛出的异常。

一个编译器标志有助于发现 API 模块中应该被标记为 `transitive` 的依赖项。如果使用 `-Xlint:exports` 进行编译，那么导出类型中应该以传递方式依赖但却没有依赖的类型就会产生警告信息。这样一来，就可以在编译 API 模块时发现问题。否则，就只有在编译缺少所需依赖项的消费模块时才会发现错误，因为没有建立隐式可读性。例如，如果在前面的 `easytext.repository.api` 模块描述符中省略修饰符 `transitive` 并使用 `-Xlint:exports` 进行编译，就会产生以下警告信息：

```
$ javac -Xlint:exports --module-source-path src -d out -m easytext.
repository.api
```

```
src/easytext.repository.api/easytext/repository/api/TextRepository.  
java:6:warning: [exports] class Text in module easytext.domain.api  
is not indirectly exported using requires transitive  
    Text findText(String id);  
      ^  
1 warning
```

在设计 API 模块时，隐式可读性是使模块更易于使用的强大技术。但依靠消费模块中的隐式可读性存在一个微妙的风险。在示例中，客户端模块可以通过隐式可读性的传递性来访问 `Text` 类。当通过接口使用类型时，这种方式也可以正常工作，如示例中 `repository.findText("HHGTTG").wordcount()` 所示。但是，如果在客户端模块中直接使用 `Text` 类，而不是通过接口的方法将其作为返回值，如直接实例化 `Text` 并将其存储在字段中，那么会发生什么事情呢？此时，可以正常地编译和运行。但客户端的模块描述符是否真正反映了我们的意图？你可以争辩说，在这种情况下，客户端模块必须显式地依赖 `easytext.domain.api` 模块。这样一来，即使客户端模块停止使用存储库模块（从而失去域 API 的隐式可读性），客户端代码也会继续编译。

这似乎是一个无关紧要的问题。代码仍然可以编译和工作，那么有什么大不了的？然而，作为模块的作者，你有责任根据代码声明正确的依赖关系。隐式可读性仅用于防止以前所看到的编译器错误等意外情况，将模块所拥有的真正依赖关系全权委托给隐式可读性是一种“懒惰的做法”。

5.3.3 带有默认实现的 API 模块

API 的实现应该放在同一个模块还是单独的模块中？这是一个有趣的问题。当希望有多个 API 的实现时，分离 API 和实现是一个有用的模式。而当只有一个实现时，将 API 和实现捆绑在同一个模块中则更有意义的。此外，提供默认实现作为一种便利手段也是有意义的。

组合的 API / 实现模块并不排除日后在单独模块中的替代实现。但是，这些替代实现模块则需要依赖于 API 的组合模块。同时，这个组合模块也包含一个实现。这样一来可能会产生多余的传递依赖关系，因为组合模块的实现依赖关系对替代实现没有任何用处，但仍需要被解析。

图 5-3 说明了这个问题。模块 `easytext.analysis.coleman` 和 `easytext.analysis` 提供了一个 `Analyzer` 接口实现作为服务，后者除了提供一个实现之外，还导出 API。但是，`easytext.analysis`（而不是 API）中的实现需要 `syllablecounter` 模块。此时如果 `syllablecounter` 不在模块路径上，即使 `easytext.analysis.coleman` 中的 API 替换实现不需要 `syllablecounter` 模块也无法运行。将 API 分离到自己的模块可以避免此类问题的出现，如图 5-4 所示。

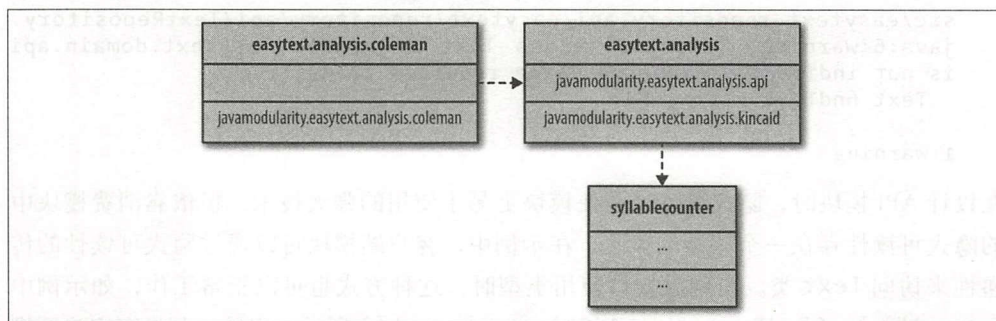


图 5-3: 当 API 位于一个带有实现的模块 (此处为 `easytext.analysis` 模块) 中时, 实现的依赖项 (此时为 `syllablecounter`) 也会传递给替代实现

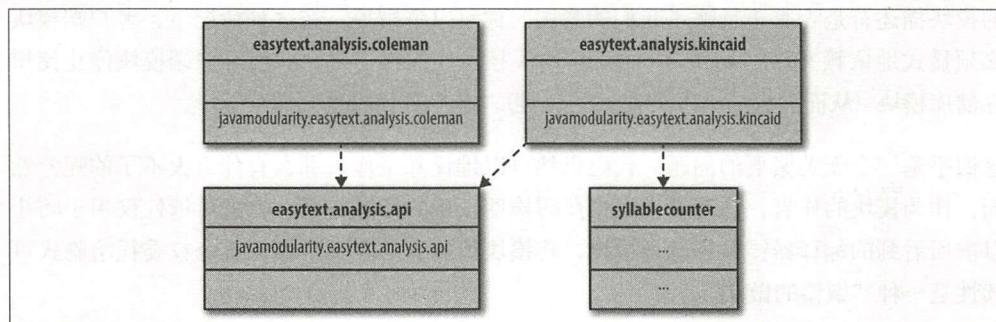


图 5-4: 当有多个实现时, 单独的 API 模块工作得更好

在第 3 章和第 4 章的末尾, 为 EasyText Analyzer 接口引入了一个单独的 API 模块。如果有 (或期望有) 多个 API 的实现, 那么将公共 API 提取到自己的模块中是很有意义的。此时就是这种情况: 主要想法是通过新的分析功能使 EasyText 更具有可扩展性。当实现作为服务提供时, 在提取 API 模块时就会产生如图 5-4 所示的模式。

最终会有多个不导出任何内容的实现模块, 这些模块依赖于 API 模块并将其实现作为服务发布。反之, API 模块仅导出包, 不包含任何封装的实现代码。通过这样的设置, `easytext.analysis.kincaid` 模块对 `syllablecounter` 的实现依赖关系就不会强加于 `easytext.analysis.coleman` 模块。而使用 `easytext.analysis.coleman` 时, 则无须在模块路径上放置 `easytext.analysis.kincaid` 或 `syllablecounter`。

5.4 聚合器模块

在了解隐式可读性之后, 接下来学习一个新的模块模式: 聚合器模块。假设有一个由几个关联度不高的模块组成的库, 根据不同的需求, 该虚拟库的用户可以使用一个或多个模块。到目前为止, 还没有什么新的内容。

5.4.1 在模块上构建一个外观

有时候，不希望让库的使用者面临具体使用哪个模块的问题。也许对库模块进行分割有益于库的维护者，但却可能让用户感到困惑。或者你希望有一种方法，可以让人们只需依赖一个代表整个库的单个模块就可以快速开发。一种方法是首先构建各个模块，然后再构建一个“超级模块”，将各个模块的所有内容组合在一起。该方法虽然有效，但不是一个特别好的解决方案。

实现类似结果的另一种方法是使用隐式可读性来构建聚合器模块，其本质上是通过现有的库模块构建一个外观 (*facade*)。聚合器模块不包含代码，它只有一个模块描述符，为所有其他模块设置了隐式可读性：

```
module library {  
  requires transitive library.one;  
  requires transitive library.two;  
  requires transitive library.three;  
}
```

现在，如果库用户添加了对库的依赖关系，那么所有三个库模块都将以传递的方式解析，并且它们的导出类型对应用程序是可读的。

图 5-5 显示了新的情况。当然，如果需要的话，依然可以依赖于一个特定的模块。

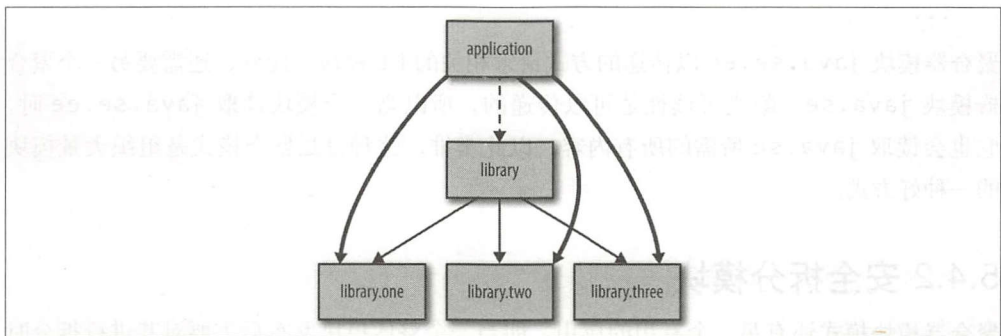


图 5-5：应用程序模块可以通过聚合器模块库使用三个库模块中的所有导出类型（隐式可读性边用粗边显示）

因为聚合器模块是轻量级的，因此完全可以创建多个不同的聚合器模块。例如，可以针对特定用户，提供库的多个配置文件或者分布。

就像第 2 章所看到的那样，JDK 就是该方法的一个很好的示例。它包含了多个聚合器模块，如表 5-1 所示。

表 5-1: JDK 中的聚合器模块

模块	聚合
java.se	所有正式属于 Java SE 规范的模块
java.se.ee	java.se 模块以及与 Java SE 平台捆绑在一起的所有 Java EE 模块

一方面,对于消费者来说,创建一个聚合器模块非常方便。消费者模块只需要聚合器模块即可,而不必过多考虑底层结构。另一方面,使用聚合器模块也存在风险。现在,消费者模块可以通过聚合器模块以可传递的方式访问聚合器模块中的所有导出类型,其中有些类型可能包括来自不希望依赖的模块。由于隐式可读性,当使用这些类型时,将不会收到来自模块系统的警告。准确地指定对底层模块的依赖可以避开这些缺陷。

使用聚合器模块是一种便利。而从 JDK 开发人员的角度来看,这又不仅仅是一种便利。聚合器模块是一种可以将平台组合成可管理且不重复的块的好方法。接下来,看看 JDK 中一个平台聚合器模块 `java.se.ee`。正如前所见,可以使用 `java --describe-module <modulename>` 来查看模块的模块描述符内容:

```
$ java --describe-modules java.se.ee
java.se.ee@9
requires java.se transitive
requires java.xml.bind transitive
requires java.corba transitive
...
```

聚合器模块 `java.se.ee` 以传递的方式请求相关的 EE 模块。此外,还需要另一个聚合器模块 `java.se`。隐式可读性是可以传递的,所以当模块读取 `java.se.ee` 时,它也会读取 `java.se` 所需的所有内容,以此类推。这种分层聚合模式是组织大量模块的一种好方式。

5.4.2 安全拆分模块

聚合器模块模式还有另一个有用的应用,即当一个整体模块发布后需要对其进行拆分时可以使用该模式。比如,模块变得太大而无法维护,或者需要分离出不相关的功能以提高可重用性。

假设需要对模块 `largelibrary` (如图 5-6 所示) 进行进一步的模块化,但是 `largelibrary` 的用户正在使用其公共 API,拆分需要以向后兼容的方式进行。也就是说,不能指望 `largelibrary` 的现有用户立即切换使用新的、更小的模块。

如图 5-7 所示的解决方案使用了一个同名的聚合器模块替换了 `largelibrary`。该聚合器模块为新的、更小的模块设置了隐式可读性。

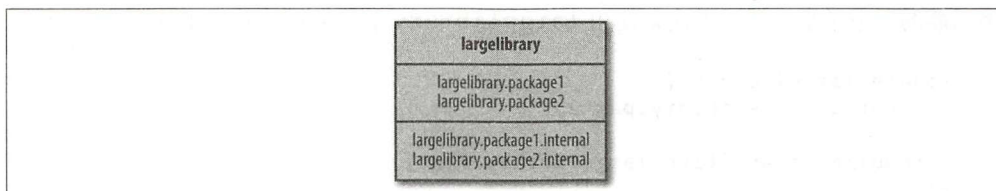


图 5-6: 拆分之前的 largelibrary 模块

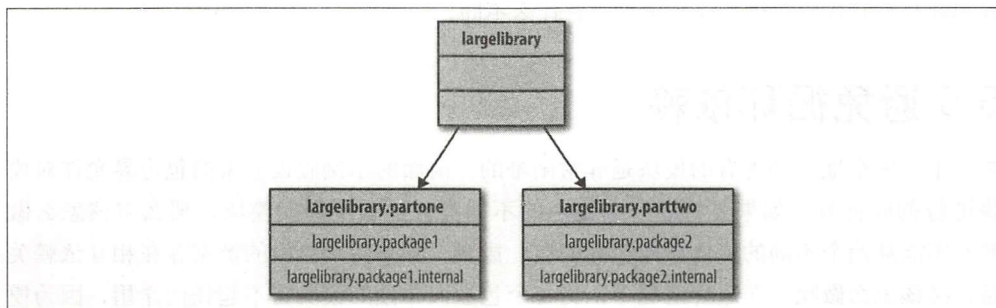


图 5-7: 拆分之后的 largelibrary 模块

现有的包（包括导出的和封装的包）分布在新引入的模块中。现在，该库的新用户既可以选择使用其中一个单独的模块，也可以选择使用 `largelibrary` 来提高整个 API 的可读性。升级到 `largelibrary` 的这个新版本时，该库的现有用户不必更改其代码或模块描述符。

没有必要总是创建一个纯粹的聚合器模块，即只包含一个模块描述符而没有自己的代码。通常一个库由独立有用的核心功能所组成。以库 `largelibrary` 为例，包 `largelibrary.part2` 可能会建立在 `largelibrary.part1` 之上。

此时，创建两个模块是有意义的，如图 5-8 所示。

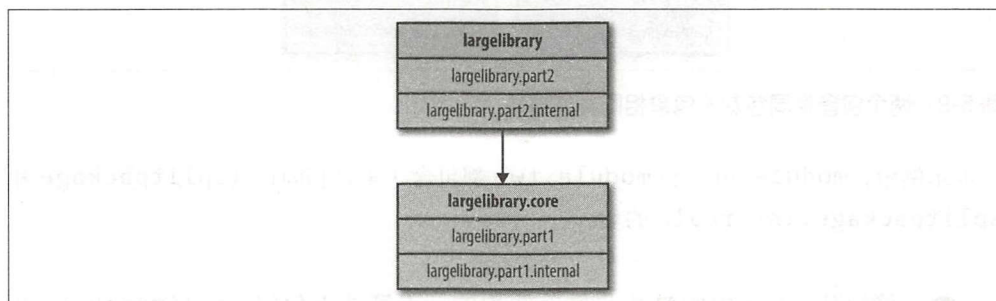


图 5-8: 拆分 largelibrary 的替换方法

`largelibrary` 的消费者可以继续使用它，或者只需 `largelibrary.core` 模块来使

用功能的一个子集。这个方法是使用 `largelibrary` 的模块描述符实现的：

```
module largelibrary {
    exports largelibrary.part2;

    requires transitive largelibrary.core;
}
```

如你所见，隐式可读性提供了一种拆分模块的安全方法。新聚合器模块的消费者不会注意到其与所有代码在单个模块中之间有什么不同。

5.5 避免循环依赖

实际上，安全地拆分现有的模块是非常困难的。前面的示例假设原来的包边界允许对模块进行彻底拆分。如果要将同一个包中的不同类放置到不同的模块，那么应该怎么做呢？不能从两个不同的模块导出相同的包。或者，如果两个包中的类型存在相互依赖关系，又该怎么做呢？在这种情况下，将每个包放入单独的模块将不起任何作用，因为模块依赖关系是不能循环的。

接下来，首先看一下拆分包所存在的问题，然后探讨一下如何重构模块之间的循环依赖。

5.5.1 拆分包

在拆分模块时，需要考虑的一种情况是拆分包的引入。拆分包 (*split package*) 是跨多个模块的单个包，如图 5-9 所示。当模块划分与现有的包边界不能保持一致时就会产生拆分包。

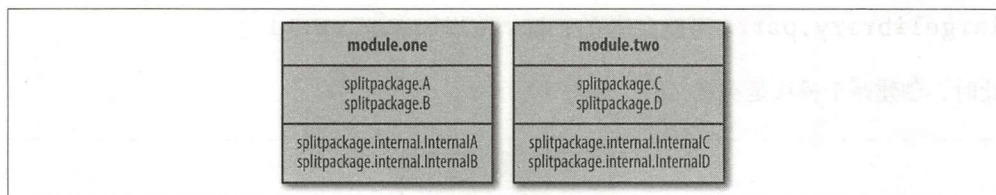


图 5-9：两个包含相同包却不包含相同类的模块

在本示例中，`module.one` 和 `module.two` 都包含了来自相同包 (`splitpackage` 和 `splitpackage.internal`) 的类。



请记住，Java 中的包是非层次结构的。不管外表如何，`splitpackage` 和 `splitpackage.internal` 是两个不相关却共享相同前缀的包。

如果将 `module.one` 和 `module.two` 放在模块路径上，那么在启动 JVM 时会产生错误。Java 模块系统不允许拆分包。只允许一个模块可以将给定的包导出到另一个模块。由于导出是根据包名称声明的，因此如果两个模块导出相同的包，那么就会产生不一致。如果允许这样做，那么两个模块就可能导出具有相同的完全限定名称的类。而当另外一个模块依赖于这两个模块并且想要使用这个类时，就会出现该类应该来自哪个模块的冲突。

即使拆分包不是从模块中导出的，模块系统也是不允许的。从理论上讲，有一个非导出的拆分包（如图 5-9 所示的 `splitpackage.internal`）并不是一个问题，毕竟拆分包进行了很好的封装。而实际上，模块系统从模块路径加载模块的方式禁止使用拆分包。模块路径中的所有模块都是在相同的类加载器中加载的。一个类加载器只能有一个包定义，是导出的还是封装的无关紧要。在 6.3 节中，你将看到模块系统的更多高级应用如何允许具有相同封装包的多个模块。

首先，避免使用拆分包的方法就是不要创建它们。当从头开始创建模块时，这种方法是很有效的，但如果是将现有的 JAR 转换为模块，那就比较困难了。

图 5-9 所示的示例演示了一个“干净”的拆分包，这意味着在不同模块中将不会出现具有相同完全限定名的类型。在将现有的 JAR 转换为模块时，遇到“不干净”拆分（多个 JAR 具有相同的类型）的情况比较少见。当然，在类路径上这些 JAR 可能会一起工作（但只是偶然情况），但在模块系统中是不可能的。此时，解决方案是将 JAR 及其重叠的包合并成一个模块。

请记住，模块系统会检查在所有模块中是否存在重叠包，其中包括平台模块。许多 JAR 尝试向 JDK 中模块所拥有的包添加类。对这些 JAR 进行模块化并将其放在模块路径上是不起任何作用的，因为它们与这些平台模块重叠。



当 JAR 所包含的包与 JDK 包重叠时，如果将这些 JAR 放到类路径上，那么它们的类型将被忽略，并且不会被加载。

5.5.2 打破循环

现在已经解决了拆分包的问题，但是仍然存在包之间循环依赖的问题。当所拆分的模块包含相互依赖的包时，就会产生循环的模块依赖关系。虽然可以创建这些模块，但是它们不会被编译。

在 2.7 节中已经讲过，模块之间的可读性关系在编译时必须是非循环的。两个模块不能

通过各自的模块描述符互相需求。随后，在 3.3.2 节中谈到了循环可读性关系可能会在运行时出现。另外请注意，服务可以互相使用，从而在运行时形成一个循环调用图。

那么，为什么在编译时严格禁止循环呢？JVM 可以在运行时延迟加载类，在出现循环的情况下允许多级解析策略。但是，只有在编译器所使用的所有其他类型已经被编译或在同一编译运行中正在被编译时，编译器才能编译该类型。

满足上述要求的最简单方法是始终将相互依赖的模块编译在一起。虽然这种做法并不是不可能，但会导致难以管理的版本和代码库。当然，这只是一个主观的说法。在编译时禁止循环模块依赖关系是 Java 模块系统所做的一个“固执”的选择，因为它认为循环依赖关系不利于进行模块化。

如果说包含循环的代码很难理解，这是没有任何争议的。尤其是因为循环可能隐藏在许多间接层次的后面，而不仅仅是简单的两个相互依赖的 JAR 中两个不同包的两个类。循环依赖使情况更加混乱，在由 Java 模块系统模块化的应用程序中毫无作用。

当需要对现有 JAR 之间具有循环依赖关系的应用程序进行模块化时，该怎么办？或者说，如果从 JAR 中拆分包时会产生循环依赖，那么应该怎么办？此时不能将它们转换成两个彼此需要的模块，因为编译器不允许这样的配置。

一个显而易见的解决方案是将这些 JAR 合并成一个模块。当两个组件之间存在如此紧密（循环）的关系时，就可以断定它们是一个模块的。当然，该解决方案的前提是循环关系是良性的。当循环是间接的并且涉及多个组件（而不仅仅是两个组件）时，该方案就行不通了，除非想要合并参与循环的所有组件，但这不太可能。

通常，存在循环说明设计是有问题的，这意味着需要进行一些重新设计来打破这个循环。常言道，计算机科学的一切问题都可以通过引入另一个间接层来解决（当然太多的间接层也会出现问题）。接下来看看如何使用一个间接的方法来打破循环。

首先，从两个 JAR 文件开始：*authors.jar* 和 *books.jar*。每个 JAR 包含一个类（分别是 *Author* 和 *Book*），并且相互引用。如果只是将现有的 JAR 转换为模块化的 JAR，那么循环依赖就变得更加明显，如图 5-10 所示。

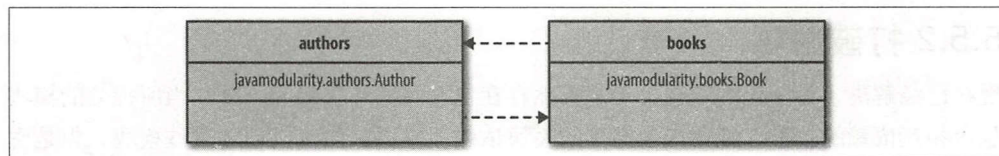


图 5-10：这些模块不会编译或解析，因为存在循环依赖

应该回答的第一个问题是：这些模块之间的正确关系是什么？这里没有一个通用的方法。需要仔细观察一下代码，看看它试图完成什么。只有这样才能回答上述问题。接下来将根据示例 5-3 来探讨这个问题。

示例 5-3: Author.java ([↪ chapter5/cyclic_dependencies/cycle](#))

```
public class Author {
    private String name;
    private List<Book> books = new ArrayList<>();

    public Author(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void writeBook(String title, String text) {
        this.books.add(new Book(this, title, text));
    }
}
```

一名 Author 拥有一个姓名，并且可以写书（所写的书将被添加到书籍列表中）：

```
public class Book {
    private Author author;
    private String title;
    private String text;

    public Book(Author author, String title, String text) {
        this.author = author;
        this.text = text;
        this.title = title;
    }

    public void printBook() {
        System.out.printf("%s, by %s\n\n%s", title, author.getName(), text);
    }
}
```

一本书可以由一名 Author 创作，并且包含一个标题和一些文本，创建后可以使用 printBook 打印图书。仔细看看上述代码，会发现它生成了 Book 对 Author 的依赖。此时之所以需要 Author，是因为在打印时需要获取一个姓名。这样就形成了一个新的抽象。Book 只关心得到一个名字。为什么要把它和 Author 耦合起来呢？也许除了作者之外，还有其他创作书籍的方式（我曾经听说过，深度学习正在取代我们完成编书的任务）。

所有的这些讨论都指向了一个新的抽象：即正在寻找的间接层。因为图书模块只对事物



名称感兴趣，所以可以引入一个名为 Named 的接口，如示例 5-4 所示。

示例 5-4: Named.java (`chapter5/cyclic_dependencies/without_cycle`)

```
public interface Named {  
    String getName();  
}
```

现在，Author 可以实现这个接口，且它已经有了所需的 getName 实现。Book 实现需要使用的是 Named，而不是 Author。最终，模块结构如图 5-11 所示。这样做还有一个额外的好处，即不再需要导出 javamodularity.authors 包。

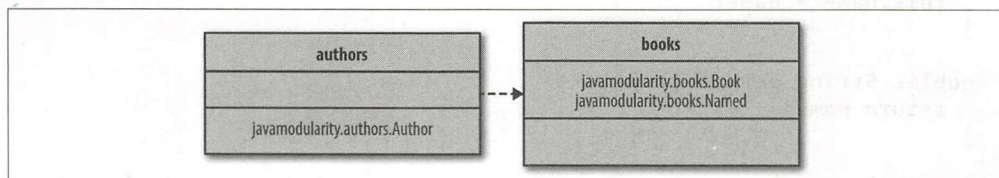


图 5-11: Named 接口打破了循环依赖

以上并不是唯一的解决方案。在一些大型系统（多个其他组件使用了 Author）中，也可以将 Named 接口放置在自己的模块中。在本示例中，这样将产生一个三角依赖关系图，Named 为顶部模块，书籍和作者模块都指向它。

一般来说，接口在打破循环依赖方面起着重要的作用。接口是 Java 中最强大的抽象手段之一，它启用了可以打破循环的依赖反转模式。

到目前为止，都是假定循环依赖的确切性质是已知的。当循环是间接的，并且通过很多步骤产生时，它们可能很难被发现，此时可以使用工具（如 SonarQube）帮助检测现有代码中的循环依赖。使用这些工具是很有帮助的。

5.6 可选的依赖关系

模块化应用程序的标志之一是其显式的依赖关系图。到目前为止，已经学习了如何通过模块描述符中的 requires 语句来构造这样一个图。但是，如果一个模块在运行时不是绝对需要的，那么又该怎么办呢？

目前许多框架都是按照以下方式工作的：向类路径中添加一个 JAR 文件（比如 fastjsonlib.jar），从而获得更多的功能。每当 fastjsonlib.jar 不可用时，框架就会使用回退机制或者不提供增强的功能。对于特定示例，解析 JSON 可能会慢一点，但它仍然有效。框架对 fastjsonlib 存在可选的依赖关系。如果应用程序已经使用了 fastjsonlib，那么框架也使用它，否则就不会使用。



Spring Framework 是一个存在很多可选依赖关系的框架示例。例如，它有一个 `Base64Utils` 辅助类，代表了 Java 8 的 `Base64` 类，或者 Apache Commons Codec `Base64` 类（如果它在类路径中的话）。无论运行时环境如何，Spring 本身都必须针对两种实现进行编译。

这些可选的依赖关系无法使用前面所介绍的模块描述符来表达，可通过 `requires` 语句表达编译时和运行时的单一模块依赖关系图。

服务提供了极大的灵活性，并且是解决应用程序中可选依赖关系的好方法，稍后将在 5.6.2 节中看到。但是，将服务绑定到 `ServiceLoader` API 可能是对现有代码一种侵入式更改。由于框架和库经常使用可选依赖关系，因此它们可能不希望强制用户使用服务 API。当不能选择使用服务时，可以使用模块系统的另一个特性来建立可选依赖关系：编译时依赖关系。

5.6.1 编译时依赖关系

顾名思义，编译时依赖关系指的是仅在编译时需要的依赖关系。通过将修饰符 `static` 添加到 `requires` 子句中，可以在模块上表达编译时依赖关系，如示例 5-5 所示。

示例 5-5：声明一个编译时依赖关系（`chapter5/optional_dependencies`）

```
module framework {  
    requires static fastjsonlib;  
}
```

通过添加 `static`，`fastjsonlib` 模块需要在编译时出现，而不是在运行框架时出现。从模块消费者的角度来看，这样做可以有效地让 `fastjsonlib` 成为框架的可选依赖关系。



编译时依赖关系还有其他应用。例如，模块可以导出仅在编译期间使用的 Java 注释。请求此类模块不应导致运行时依赖关系，所以 `requires static` 适用这种情况。

当然，此时的问题是在没有 `fastjsonlib` 的情况下运行 `framework` 时会发生什么事情呢？接下来看一下 `framework` 使用直接从 `fastjsonlib` 导出的 `FastJson` 类：

```
package javamodularity.framework;  
import javamodularity.fastjsonlib.FastJson;  
  
public class MainBad {
```



```
public static void main(String... args) {  
    FastJson fastJson = new FastJson();  
}  
}
```

如果在没有 `fastjsonlib` 的情况下运行 `framework`，会产生 `NoClassDefFoundError`。此时编译器并不会“抱怨”所缺少的 `fastjsonlib` 模块，因为它只是一个编译时依赖关系。但在运行时会产生错误，因为 `FastJson` 是运行 `framework` 所必需的。

在模块上表达编译时依赖关系的作用是防止在运行时出现上述问题。这意味着 `framework` 需要围绕编译时依赖关系对类的使用进行防御性编码。在 `MainBad` 中直接引用 `FastJson` 是有问题的，因为 VM 将总是尝试加载类并将其实例化，这样会导致 `NoClassDefFoundError`。

幸运的是，Java 对类具有延迟加载语义：即在最后可能的时间里加载类。如果可以以某种方式试探性地尝试使用 `FastJson` 类，并在该类不可用的情况下恢复正常，那么就可以实现我们的目标。通过使用反射以及适当的 `try-catch` 块，`framework` 可以防止运行时错误：

```
package javamodularity.framework;  
  
import javamodularity.fastjsonlib.FastJson;  
  
public class Main {  
  
    public static void main(String... args) {  
        try {  
            Class<?> clazz = Class.forName("javamodularity.fastjsonlib.FastJson");  
            FastJson instance =  
                (FastJson) clazz.getConstructor().newInstance();  
            System.out.println("Using FastJson");  
        } catch (ReflectiveOperationException e) {  
            System.out.println("Oops, we need a fallback!");  
        }  
    }  
}
```

当找不到 `FastJson` 类时，可以使用 `catch` 块作为一个后备。另一方面，如果 `fastjsonlib` 存在，则可以在反射实例化之后使用 `FastJson` 类。



即使 `fastjsonlib` 在模块路径上，`requires static fastjsonlib` 子句也不会导致 `fastjsonlib` 在运行时被解析！在 `fastjsonlib` 上需要有一个直接的 `requires` 子句，或者可以通过 `--add-modules`



`fastjsonlib` 将其添加为根模块，以便进行解析。在这两种情况下，`fastjsonlib` 都会被解析，并被 `framework` 读取和使用。

通过编译时依赖关系监视每个类及其使用可能是非常痛苦的。同时，延迟加载也意味着一个类的加载时间可能是出人意料的，一个臭名昭著的例子是类中静态初始化块。由此可见，`requires static` 可能并不是在模块之间创建可选耦合的最佳方式。

无论模块何时使用编译时依赖关系，都要尝试将这些可选类型的使用集中在模块的一部分中。首先保护好单个顶级类的实例化，而该类反过来（直接）引用了可选依赖关系中的类型。这样一来，模块就不会被防御性代码所破坏。

当然，在某些情况下也是需要使用 `requires static` 的。例如，引用另一个模块的编译时注释。一个 `requires` 子句（不带 `static`）也会产生在运行时需要的依赖关系。

有时，类上的注释仅在编译时使用。例如，执行静态分析（如检查 `@Nullable` 或 `@NonNull`），或将类型标记为代码生成的输入。当在运行时检索元素的注释并且注释类不存在时，JVM 会正常降级，并且不会抛出任何类加载异常。不过，编译代码时仍然需要访问注释类型。

接下来看一个可以放在实体上的虚构的 `@GenerateSchema` 注释。在构建时，这些注释用于查找类，根据其签名生成数据库模式。注释在运行时不使用。因此，希望使用 `@GenerateSchema` 注释的代码在运行时不要求使用 `schemagenerator` 模块（该模块导出了注释）。假设在模块 `application` 中有示例 5-6 中的类。

示例 5-6：注释类（`chapter5/optional_dependencies_annotations`）

```
package javamodularity.application;

import javamodularity.schemagenerator.GenerateSchema;

@GenerateSchema
public class BookEntity {

    public String title;
    public String[] authors;
}
```

`application` 的模块描述符应该包含正确的编译时依赖关系：

```
module application {
    requires static schemagenerator;
}
```

在 `application` 中，还有一个实例化 `BookEntity` 并尝试获取该类上注释的主类：



```
package javamodularity.application;

public class Main {
    public static void main(String... args) {
        BookEntity b = new BookEntity();
        assert BookEntity.class.getAnnotations().length == 0;
        System.out.println("Running without annotation @GenerateSchema present.");
    }
}
```

当运行不带 `schemagenerator` 模块的应用程序时，一切正常：

```
$ java -ea --module-path out/application \
    -m application/javamodularity.application.Main
Running without annotation @GenerateSchema present.
```

(`-ea` 标志启用 JVM 中的运行时断言。)

由于运行时缺少 `schemagenerator` 模块，因此不存在类加载或模块解析问题。在运行时缺少模块是允许的，因为它是一个编译时依赖关系。随后，JVM 像往常一样在运行时处理注释类的缺失。调用 `getAnnotations` 将返回一个空数组。

然而，如果显式地添加了 `schemagenerator` 模块，那么就会找到并返回注释 `@GenerateSchema`：

```
$ java -ea --module-path out/application:out/schemagenerator \
    -m application/javamodularity.application.Main
Exception in thread "main" java.lang.AssertionError
    at application/javamodularity.application.Main.main(Main.java:6)
```

此时，抛出了一个 `AssertionError`，因为现在返回了 `@GenerateSchema` 注释，注释数组不再为空了。

与前面所看到的编译时依赖关系示例不同的是，在运行时不需要使用防护代码来处理缺失的注释类型。JVM 已经在类加载和反射访问注释过程中处理了这个问题。

此外，还可以将 `requires` 上的修饰符 `static` 与 `transitive` 结合起来使用：

```
module questionable {
    exports questionable.api;
    requires transitive static fastjsonlib;
}
```

当在导出包中引用可选依赖关系的一个类型时，可能需要使用上面的代码。虽然 `static` 可以与 `transitive` 结合使用，但这样做并不是一个好主意。此时需要在 API 消费者上创建适当的防护代码，这显然不符合最小惊讶原则 (the principle of least surprise)。事实上，使用这种组合修饰符的唯一原因是通过这种模式来完成遗留代码的模块化。



虽然通过 `requires static`，可选依赖关系可以完成很多功能，但在模块系统中的服务可以做得更好！

5.6.2 使用服务实现可选依赖关系

虽然使用编译时依赖关系来建立可选依赖关系是可能的，但需要经常使用反射来保护类的加载。此时，使用服务更加合适。在第 4 章已经看到，服务消费者可以从服务提供者模块中获得零个或多个服务类型的实现。获取零个或一个服务实现只是这种通用机制的一个特例。

接下来使用可选的 `fastjsonlib` 将该机制应用于前面的框架示例。再次说明一下：首先从一个不成熟的重构开始，然后通过几个步骤将其精炼成一个真正的解决方案。

在示例 5-7 中，`framework` 变成了由 `fastjsonlib` 所提供的可选服务的消费者。

示例 5-7：消费一个在运行时可能可用，也可能不能用的服务（`chapter5/optional_dependencies_service`）

```
module framework {
    requires static fastjsonlib;
    uses javamodularity.fastjsonlib.FastJson;
}
```

由于使用了 `uses` 子句，因此可以在框架代码中使用 `ServiceLoader` 加载 `FastJson`：

```
FastJson fastJson =
    ServiceLoader.load(FastJson.class)
        .findFirst()
        .orElse(getFallback());
```

当 `FastJson` 可用时，不再需要在框架代码中使用反射来获取 `FastJson`。如果 `ServiceLoader` 没有找到任何服务（意味着 `findFirst` 返回了一个空 `Optional`），那么假设可以通过使用 `getFallback` 来实现回退操作。

当然，`fastjsonlib` 必须提供我们感兴趣的类作为服务：

```
module fastjsonlib {
    exports javamodularity.fastjsonlib;

    provides javamodularity.fastjsonlib.FastJson
        with javamodularity.fastjsonlib.FastJson;
}
```

有了上面的设置，解析器甚至可以在不显式添加 `fastjsonlib` 的情况下，根据 `uses` 和 `provides` 子句来解析 `fastjsonlib`。

不过，从纯粹的编译时依赖关系到服务的重构还有一些问题需要解决。首先，将一个类



直接公开为一个服务而不是公开一个接口并隐藏实现细节，这种做法本身就有点奇怪。将 `FastJson` 拆分成一个导出接口和封装的实现就可以解决这个问题。此外，该重构也使框架能够实现一个实现了相同接口的回退 (fallback) 类。

当尝试在没有 `fastjsonlib` 的情况下运行 `framework` 时会出现更大的问题。毕竟，`fastjsonlib` 是可选的，所以缺少 `fastjsonlib` 是完全可能的。当在模块路径上没有 `fastjsonlib` 的情况下启动 `framework` 时，会发生以下错误：

```
Error occurred during initialization of VM
java.lang.module.ResolutionException: Module framework does not read a module
that exports javamodularity.fastjsonlib
```

无论是否存在提供者模块，都不能在运行时无法读取的类型上使用 `uses` 声明服务依赖关系。明显的解决方案是将对 `fastjsonlib` 的编译时依赖关系改为正常的依赖关系 (使用不带 `static` 的 `requires`)。然而，这并不是一个理想的解决方案：对库的依赖应该是可选的。

从这一点上讲，更具有侵入性的重构是非常必要的。很显然，为了让上面的服务设置正常工作，在 `framework` 和 `fastjsonlib` 之间的关系中并不是所有的关系都是可选的。为什么 `FastJson` 接口 (假设将其重构成一个接口) 要位于 `fastjsonlib` 中呢？最终，将由框架决定所想要使用的功能。功能可由库或框架本身的回退代码有选择地提供。鉴于这个现实，将接口放在 `framework` 中或者框架和库之间共享的独立 API 模块中会更有意义。

这是一个侵入性重新设计，它几乎颠倒了框架和库之间的关系。此时，框架不再选择性地使用库，而是库必须使用框架 (或其 API 模块) 来实现一个接口，并将该实现作为一个服务提供。然而，当取消这种重新设计时，框架和库之间就会实现完美的解耦。

5.7 版本化模块

当谈论到框架和库的模块时，不可避免地会谈到版本化问题。模块是可独立部署和可重复使用的单元，可通过组合正确的部署单元 (模块) 来构建应用程序。仅使用模块名称来选择在一起工作的正确模块是不够的，此时还需要版本信息。

Java 模块系统中的模块不能在 `module-info.java` 中声明一个版本。不过，在创建模块化 JAR 时可以附加版本信息。可以使用 `jar` 工具的 `--module-version = <V>` 标志来设置版本。版本 `V` 被设置为已编译的 `module-info.class` 上的一个属性，并且在运行时可用。为模块化 JAR 添加版本是一个很好的做法，特别是本章前面讨论的 API 模块。



语义版本控制

版本化模块有很多不同的方法。版本控制中最重要的目标是将更改的影响传达给模块的消费者。新版本是否是以前版本的安全替代品，模块的 API 是否有任何更改？语义版本控制规范了被广泛使用和理解的版本控制方案：

MAJOR.MINOR.PATCH

破坏性的更改（例如，更改了接口中的方法签名）会导致 MAJOR 版本部分发生变化。对向后兼容的公共 API 的更改（例如，在公共类上添加一个方法）则会改变版本字符串的 MINOR 部分。最后，当改变实现细节时 PATCH 部分会增加，所改变的实现细节可以是一个 Bug 修复或代码优化。在任何情况下，PATCH 增量都应该是以前版本的安全替代品。

请注意，判断某些事情是主要变化还是次要变化并不总是那么简单的。例如，只有当接口的消费者应该实现该接口时，向接口添加方法才属于重大变化。而如果接口仅由消费者调用（而不是实现），那么在添加了方法之后并不会破坏消费者的使用体验。更复杂的是，即使在第一种情况下，接口上的默认方法（在 Java 8 中引入）也可以将接口上的方法添加变为次要变化。不管是哪种情况，重要的是始终从模块消费者的角度来推理：模块的下一个版本应该反映即将发生的变化对模块用户的影响。

模块解析和版本控制

即使支持向模块化 JAR 添加版本，模块系统也不会以任何方式使用它。模块完全是由名称进行解析的。这听起来可能很奇怪，因为前面已经讲过，版本在决定哪些模块可以一起很好地工作上扮演着重要的角色。在模块解析期间忽略版本并不是一个疏忽，而是 Java 模块系统中一个慎重的设计选择。

如何指出哪个版本的部署单元可以一起工作是一个颇有争议的话题。版本字符串的语法和语义是什么？是否指定了依赖关系的版本范围？或者只有确切的版本？如果最终同时需要两个版本，会发生什么情况呢？这些冲突必须解决。

对于上述问题，不同的工具和框架（例如，Maven 和 OSGi）都给出了自己的解决方案。事实证明，这些版本选择算法以及相关的冲突解决试探法既复杂又不同（有时仅存在微小的区别）。这就是为什么现在的 Java 模块系统在模块解析过程中避开了版本选择的概念。无论采用什么策略，都会在模块系统中变得根深蒂固，因此也会深入到编译器、语言规范和 JVM 中。策略的错误选择所造成的代价太高了，因此，模块描述符中的 `requires` 子句只使用模块名称，而不是模块版本（或范围）。



此时，开发人员仍然面临一项挑战。如何选择正确的模块版本放在模块路径上？答案非常简单，虽然有点不尽如人意：就像前面使用类路径那样，即将正确版本的依赖关系的选择和检索委托给现有的构建工具来完成。Maven 和 Gradle 等通过将依赖关系版本信息外部化到一个 POM 文件来处理这个问题，而其他工具可能会使用其他方法，但不管用什么方法，这些信息必须存储在模块之外。

图 5-12 显示了构建一个依赖于两个模块化 JAR (lib 和 foo) 的源模块 application 所涉及的步骤。在构建时，构建工具使用来自 POM 文件的信息从存储库下载正确版本的依赖项。在此过程中出现的任何版本冲突必须由构建工具的冲突解决算法来解决。下载模块化 JAR 被放在模块路径上进行编译。然后，Java 编译器解析由模块路径上的模块信息描述符以及 application 所引导的模块图。有关现有构建工具如何处理模块的更多详细信息请参阅第 11 章。

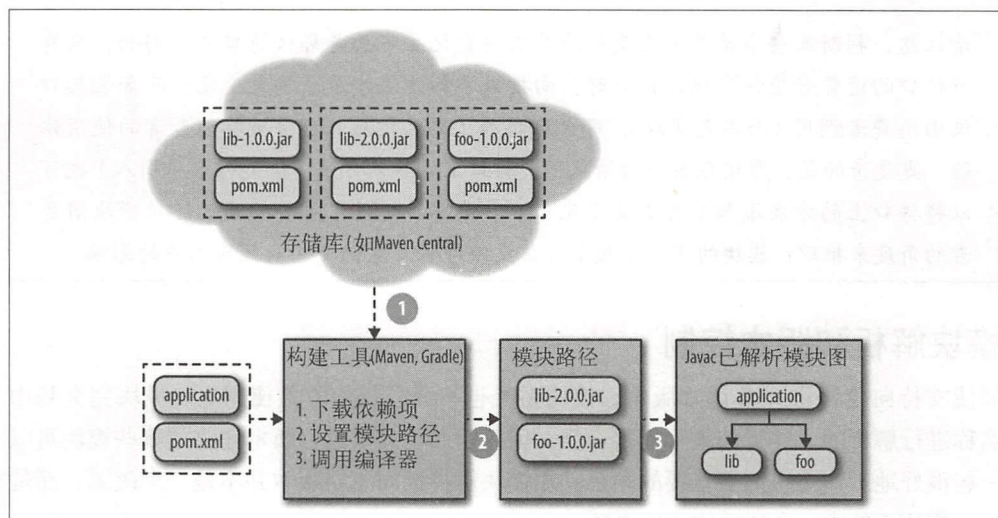


图 5-12：构建工具选择正确版本的依赖关系，并放在模块路径上

- ❶ 诸如 Maven 或 Gradle 之类的构建工具从存储库（如 Maven Central）下载依赖项。由构建描述符中的版本信息控制下载哪个版本。
- ❷ 构建工具使用下载的版本设置模块路径。
- ❸ 然后，Java 编译器或运行时从模块路径中解析模块图，为了保证应用程序正常运行，不能包含重复的版本。模块的重复版本会导致错误。

Java 模块系统确保在通过模块解析过程编译和运行 application 时，所有必需的模块都存在。但是，模块系统不知道模块的哪个版本被解析。只要存在一个正确名称的模块，它就会被解析。另外，如果在模块路径的目录中找到多个名称相同（但可能不同版



本) 的模块, 则会导致错误。



如果两个具有相同名称的模块位于模块路径上的不同目录中, 那么解析器将使用所遇到的第一个模块, 而忽略第二个模块。此时, 不会产生任何错误。

真实的情况是, 同时使用相同模块的多个版本只是权宜之计。前面已经看到, 在默认情况下, 当通过模块路径启动应用程序时, 上述情况是不支持的。这与模块系统之前的情况类似。在类路径上, 两个不同版本的 JAR 可能导致未定义的运行时行为, 这可能更糟。

在应用程序开发中, 强烈建议找到一种统一依赖于单个模块版本的方法。通常, 运行同一模块的多个并发版本的原因是因为懒惰, 而不是迫切需要。在开发类似于容器的通用应用程序时, 这种懒惰的做法往往并不可行。在第 6 章中, 将会看到有一种方法可以通过使用更复杂的模块系统 API 在运行时构建模块图来解析模块的多个版本。而另一种选择是采用现有的模块系统, 如 OSGi, 它则提供了可以同时运行多个版本的可能性。

5.8 资源封装

前面已经花了相当多的时间来讨论模块中的强封装代码。尽管这是强封装的最显著的用法, 但除了代码, 应用程序通常拥有更多的资源, 比如包含翻译(本地化资源包)的文件、配置文件、用户界面中使用的图像等。将这些资源封装在一个模块中是非常值得的, 可以将它们与使用资源的代码进行整合。一个模块使用另一个模块的资源就像依赖私有实现类一样糟糕。

从历史上看, 代码可以“免费地”使用类路径上的资源, 因为没有在资源上应用访问修饰符。任何类都可以读取类路径上的任何资源。但随着模块的出现, 情况也发生了改变。在默认情况下, 模块中包内的资源被强制封装, 这些资源只能在模块中使用, 就像非导出包中的类一样。

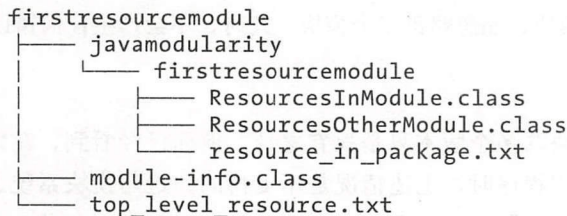
然而, 许多工具和框架依赖于查找资源, 而不管它们来自哪里。框架可能会扫描某些配置文件(例如, Java EE 中的 *persistence.xml* 或 *beans.xml*), 或者依赖于应用程序代码中的资源, 这就要求模块内的资源可以从其他模块访问。为了妥善处理这些情况并保持向后兼容性, 在模块中封装资源的默认设置中增加了许多例外情况。

首先, 看看如何从同一个模块中加载资源。其次, 了解一下模块如何共享资源, 以及默认强封装的例外情况。最后, 将注意力转向资源加载的一个特例: *ResourceBundles*。这个机制主要用于本地化, 并且已经针对模块系统进行了更新。



5.8.1 从模块加载资源

下面是一个已编译模块 `firstresourcemodule` 的示例，包含了代码和资源：



在本示例中，除了两个类以及模块描述符之外，还有两个资源：`resource_in_package.txt` 和 `top_level_resource.txt`。假设在构建过程中将资源放入模块中。

资源通常通过 `Class` API 提供的资源加载方法进行加载。在模块中的加载方法也是一样的，如示例 5-8 所示。

示例 5-8：在模块中加载资源的各种方法（`chapter5/resource_encapsulation`）

```
public static void main(String... args) throws Exception {
    Class clazz = ResourcesInModule.class;
    InputStream cz_pkg = clazz.getResourceAsStream("resource_in_package.txt"); ❶
    URL cz_tl = clazz.getResource("/top_level_resource.txt"); ❷

    Module m = clazz.getModule(); ❸
    InputStream m_pkg = m.getResourceAsStream(
        "javamodularity/firstresourcemodule/resource_in_package.txt"); ❹
    InputStream m_tl = m.getResourceAsStream("top_level_resource.txt"); ❺

    assert Stream.of(cz_pkg, cz_tl, m_pkg, m_tl)
        .noneMatch(Objects::isNull);
}
}
```

- ❶ 使用 `Class.getResource` 读取一个资源，并解析相对于类所在包的名称（此时为 `javamodularity.firstresourcemodule`）。
- ❷ 当读取顶级资源时，必须以斜杠为前缀，以避免资源名称的相对解析。
- ❸ 可以通过 `Class` 获取 `java.lang.Module` 实例（表示类来自哪个模块）。
- ❹ `Module` API 引入了从模块获取资源的新方法。
- ❺ `getResourceAsStream` 方法也适用于顶级资源。`Module` API 总是使用绝对名称，因此顶级资源不需要前导斜杠。

上述方法用于加载同一个模块中的资源，无须为了加载资源而修改现有的代码。只要调用 `getResource{AsStream}` 的 `Class` 实例属于包含资源的当前模块，就会返回正



确的 `InputStream` 或 `URL`。另一方面，当 `Class` 实例来自另一个模块时，由于资源封装，将返回 `null`。



也可以通过 `ClassLoader::getResource*` 方法加载资源。在模块上下文中，最好使用 `Class` 和 `Module` 上的方法。`ClassLoader` 方法不会像 `Class` 和 `Module` 方法那样考虑当前模块上下文，从而导致潜在的混乱结果。

还可以使用一种新的方法来加载资源。它是通过新的 `Module` API 来实现的，在“对模块的反射”一节（6.2 节）将对此进行更详细的讨论。`Module` API 公开了 `getResourceAsStream`，以便从模块加载资源。包中的资源可以以绝对方式引用，用斜杠替换包名称中的点并附加文件名即可。例如，`javamodularity.firstresourcemodule` 变成 `javamodularity/firstresourcemodule`。添加文件名后，加载包中资源的参数变为 `javamodularity/firstresourcemodule/resource_in_package.txt`。

同一模块中的任何资源，无论是在包中或是在顶层，都可以通过前面所介绍的方法加载。

5.8.2 跨模块加载资源

如果获取了表示当前模块以外的模块的 `Module` 实例，那么又会发生什么情况？看起来似乎可以在这个模块实例上调用 `getResourceAsStream`，并访问该模块中的所有资源。但由于资源的强封装性，情况并非如此。前面所介绍的资源访问规则存在几个例外情况，所以接下来向示例添加一个模块 `secondresourcemodule`，从而研究一下不同的场景：

```
secondresourcemodule
├── META-INF
│   └── resource_in_metainf.txt
├── javamodularity
│   └── secondresourcemodule
│       ├── A.class
│       └── resource_in_package2.txt
├── module-info.class
└── top_level_resource2.txt
```

假设 `firstresourcemodule` 和 `secondresourcemodule` 的模块描述符都是空的，这意味着没有包被导出。此时，分别有一个包含类 `A` 和资源的包、一个顶级资源以及 `META-INF` 目录下的一个资源。在查看下面的代码时，请记住，资源封装只适用于模块中包内的资源。

接下来尝试从 `firstresourcemodule` 的一个类中访问 `secondresourcemodule` 中的资源：

```
Optional<Module> otherModule =
```



```

ModuleLayer.boot().findModule("secondresourcemodule"); ❶

otherModule.ifPresent(other -> {
    try {
        InputStream m_tl = other.getResourceAsStream("top_level_resource2.txt"); ❷
        InputStream m_pkg = other.getResourceAsStream(
            "javamodularity/secondresourcemodule/resource_in_package2.txt"); ❸
        InputStream m_class = other.getResourceAsStream(
            "javamodularity/secondresourcemodule/A.class"); ❹
        InputStream m_meta =
            other.getResourceAsStream("META-INF/resource_in_metainf.txt"); ❺
        InputStream cz_pkg =
            Class.forName("javamodularity.secondresourcemodule.A")
                .getResourceAsStream("resource_in_package2.txt"); ❻

        assert Stream.of(m_tl, m_class, m_meta)
            .noneMatch(Objects::isNull);
        assert Stream.of(m_pkg, cz_pkg)
            .allMatch(Objects::isNull);

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

- ❶ 可以通过引导层获取一个 Module。下一章将会介绍相应的 Module Layer API。
- ❷ 可以随时加载来自其他模块的顶级资源。
- ❸ 在默认情况下，来自其他模块的包中的资源是被封装的，所以此时返回 null。
- ❹ .class 文件存在例外；这些文件可以始终从另一个模块加载的。
- ❺ 因为 META-INF 不是一个有效的包名称，所以可以访问该目录中的资源。
- ❻ 虽然通过使用 Class::forName 可以获得一个 Class<A> 实例，但就像第 3 步那样，通过该实例加载封装资源时将返回 null。

资源封装仅适用于包中的资源。但类文件资源（以 .class 结尾）是例外；即使它们在包内，也不会被封装。所有其他资源可以被其他模块自由使用，但并不意味着应该这么做。依赖来自另一个模块的资源并不是完全的模块化，最好只从同一个模块中加载资源。如果确实需要来自其他模块的资源，可以考虑通过使用导出类中的方法甚至作为服务来公开资源的内容。这样一来，依赖关系在模块描述符中就变得更加明确了。

公开包中资源

通过使用开放式模块（*open module*）或开放式包（*open package*），可以向其他模块公开包中封装的资源。这些概念将在“开放式模块和包”一节（6.1.2 节）中介绍。加载位于开放式模块或包中的资源就好像没有对资源进行封装一样。

5.8.3 使用 ResourceBundle 类

JDK 自身通过服务公开了资源的一个示例就是 ResourceBundle，ResourceBundle 提供了



一种机制将本地化作为 JDK 的一部分。从本质上讲，它们就是特定于语言环境的键值对列表。可以自己实现 `ResourceBundle` 接口或使用一个属性文件，后一种方法更为方便，因为默认情况下支持按照预定义的格式加载属性文件，如示例 5-9 所示。

示例 5-9：ResourceBundle 机制加载的属性文件，其中 Translations 是用户定义的基础名称

```
Translations_en.properties
Translations_en_US.properties
Translations_nl.properties
```

然后加载针对特定语言环境的资源包，并对密钥进行转换：

```
Locale locale = Locale.ENGLISH;
ResourceBundle translations =
    ResourceBundle.getBundle("javamodularity.resourcebundle.
        Translations", locale);
String translation = translations.getString("modularity_key");
```

转换属性文件位于模块的一个包中。按照惯例，`getBundle` 实现可以扫描类路径以加载文件。然后，根据语言环境选择最合适的属性文件，而不管它来自哪个 JAR。



解释 `ResourceBundle::getBundle` 如何根据给定的基础名称和语言环境选择正确的资源包已经超出了本书的范围。如果不熟悉此过程，`ResourceBundle` JavaDoc 包含了大量有关如何使用回退机制加载特定文件的信息。此外，除了属性文件，还支持其他基于类的格式。

如果使用了模块，那么就不存在类路径扫描。前面已经讲过，模块中的资源是被封装的。只考虑调用 `getBundle` 的模块中的文件。

将不同语言环境的转换结果放入单独的模块是可取的。相比于仅公开资源，开放这些模块或包（请参阅“公开包中资源”内容）会取得更好的结果。这就是在 Java 9 中引入基于服务的 `ResourceBundle` 机制的原因。它基于一个名为 `ResourceBundleProvider` 的新接口，包含一个带有签名 `ResourceBundle getBundle(String basename, Locale locale)` 的单一方法。每当一个模块想要提供额外的转换时，可以创建该接口的实现并将其注册为服务。然后，实现找到模块中正确的资源并返回它，如果模块中没有给定语言环境的合适转换，则返回 `null`。

在这种模式下，只需添加一个模块就可以扩展应用程序中支持的语言环境。只要注册了一个 `ResourceBundleProvider` 实现，请求语言环境转换的模块就可以通过 `ResourceBundle::getBundle` 自动获取该实现。在本章所附的代码中可以找到完整的示例（[chapter5/resourcebundles](#)）。



第 6 章

高级模块化模式

前一章介绍了模块化应用程序开发的通用设计指南和模式，本章包含了更多可能不适用于日常开发的高级模式和模块系统 API。不过，它们也是模块系统的重要组成部分。模块系统不仅可以被应用程序开发人员直接使用，而且还可以作为其他框架的基础，而高级 API 主要是用于后一种用法。

下一节将讨论目前许多库和框架所使用的反射。将开放式模块和包作为一项功能引入，可以在运行时减弱强封装性，这也是迁移过程中的一个重要功能，所以在第 8 章中还会重新讨论。

在介绍完开放式模块和包之后，关注点将转移到动态扩展应用程序的模式。思考一下基于插件的系统或应用程序容器，这些系统的核心是在运行时添加模块所面临的挑战，而不仅仅是使用来自模块路径中的模块的固定配置。



如果是第一次学习模块系统，可以跳过本章的后半部分，大多数应用程序都不会使用模块系统中更动态和更高级的功能。在获得了有关典型模块化方案的更多经验之后，可以再返回阅读本章的后半部分，从而更好地了解这些功能。

6.1 重温强封装性

在前面的章节中已经详细讨论了强封装的优点。一般来说，严格限制哪些是公开导出的 API 大有裨益。但实际上，限制的依据是很难确定的，存在许多依靠访问应用程序的实现类来完成工作的库和框架，如序列化库、对象关系映射器以及依赖注入框架。所有这些库都希望操作那些应该是内部实现细节的类。



对象关系映射器或序列化库需要访问实体类，以便可以实例化它们并填充正确的数据。即使实体类从不离开模块，ORM 库模块也需要访问它们。又如，依赖注入框架需要将服务实例注入服务实现类中，仅导出接口是不够的。通过强封装模块中的实现类，这些框架无法实现以前在类路径上所享有的访问。

反射毫无疑问是这些框架的首选工具。反射是 Java 平台的重要组成部分，允许代码在运行时检查代码。也许这听起来有点深奥，那是因为反射确实比较难理解。虽然在应用程序代码中并不一定要使用反射，但如果没有反射，许多通用框架（比如 Hibernate 或 Spring）是无法实现的。但正如前面的章节中所学到的，即使是反射也不能破坏模块中非导出包周围强封装所形成的壁垒。

此时，某些人可能会不加选择地导出所需的包。导出包意味着其 API 可以根据不同的模块进行编译和依赖。这并不是所希望的。此时需要一种机制来指示某些库可以获得（反射）某些类型的运行时访问权限。

6.1.1 深度反射

为了让传统的基于反射的库更好地利用强封装性，需要解决两个正交问题：

- 在不导出包的情况下访问内部类型。
- 允许反射访问这些类型的所有部分。

先解决第二个问题。假设导出一个可以让库访问的包。如前所述，这意味着可以针对包中的公共类型进行编译，但这是否也意味着可以在运行时使用反射来“闯入”这些类型的私有部分？这种深度反射（*deep reflection*）的做法被许多库所使用。例如，Spring 或 Hibernate 使用该方法将值注入类的非公共字段中。

回到前面的问题：能否从导出包对公共类型进行深度反射？答案是否定的。事实证明，即使一个类型被导出，也并不意味着可以无条件地通过反射“闯入”这些类型的私有部分。

从模块化的角度来看，这是正确的做法。当任意模块可以“闯入”导出类型的私有部分时，它们就会这样做。实际上，那些私有部分再次成为官方 API 的一部分。如 2.8 节所述，JDK 本身已经出现了类似情况。

防止访问非公开部分不仅是 API 是否“卫生”的问题：出于很多原因的考虑，私有的字段往往是私有的。例如，JDK 中有一个用于管理密钥和凭证的 `java.security.KeyStore` 类，该类的作者特别不希望任何人访问保护这些秘密的私有字段！

导出一个包并不意味着允许使用这些导出类型的模块反射包中的非公共部分。在 Java



中，深度反射由反射对象上的 `setAccessible` 方法提供支持。它绕过了检查，从而可以访问不可访问的部分。在模块系统和强封装出现之前，`setAccessible` 基本上不会失败。但如果使用了模块系统，那么规则就发生了变化。图 6-1 显示了不再适用的场景。

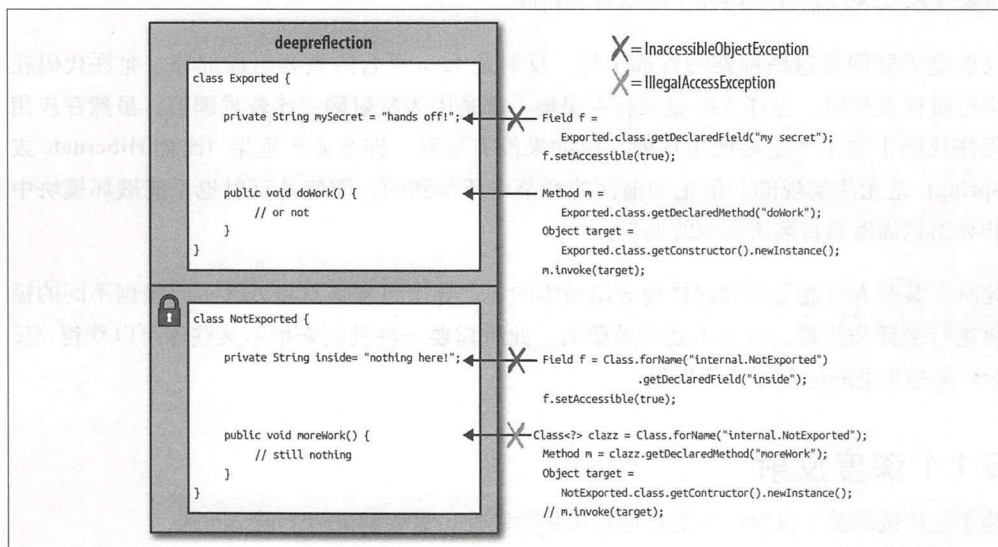


图 6-1：模块 `deepreflection` 公开了一个包含类 `Exported` 的包，同时封装了 `NotExported` 类。上述代码片段（假设位于另一个模块中）说明了反射仅能对导出类型的公共部分起作用。只有 `Exported:: doWork` 可以反射访问；访问其他内容都会导致异常

不管类型是否导出，许多流行的库都希望可以深度反射。但由于强封装性，它们很难做到这一点。而且，即使导出了实现类，也会禁止对非公共部分进行深度反射。

6.1.2 开放式模块和包

此时需要的是一种可以在不导出类型的情况下在运行时让类型可用于深度反射的方法。一旦具备了这样的功能，框架可以完成自己的工作，同时在编译时仍然可以保持强封装性。

开放式模块提供了这些功能的组合。当开放一个模块时，所有类型都可以在运行时被其他模块深度反射。无论是否导出包，该属性都是成立的。

只需在模块描述符中添加关键字 `open`，就可以开放一个模块：

```

open module deepreflection {
    exports api;
}
    
```




当开放了一个模块后，如图 6-1 所示的失败模式都消失了，如图 6-2 所示。

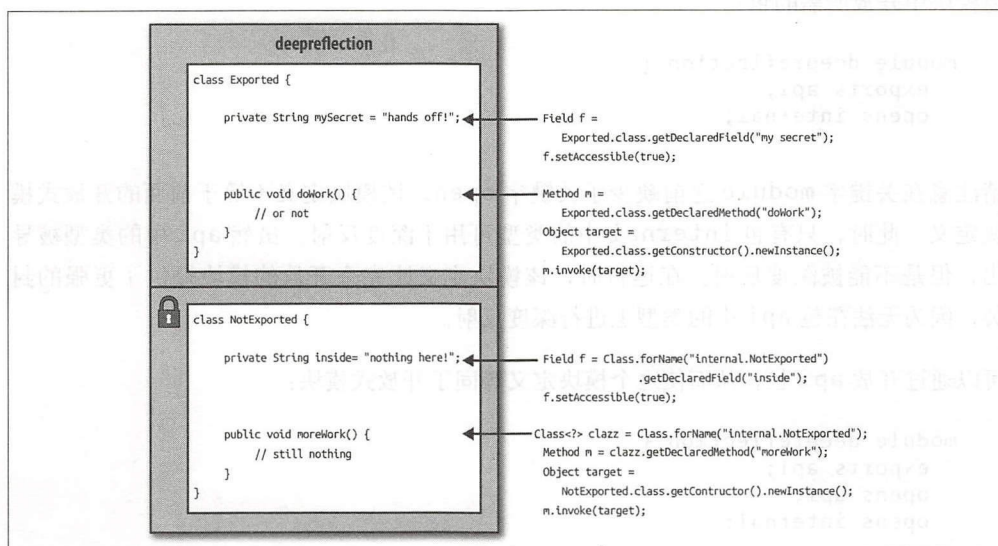


图 6-2：使用一个开放式模块后，所有包中的所有类型都可以在运行时进行深度反射。从另一个模块执行深度反射时不会引发异常

为了进行深度反射，open 关键字开放了模块中的所有包。除了开放包之外，还可以导出包，就像本例中导出包含 Exported 类的 api 包一样。在调用 setAccessible 之后，任何模块都可以反射访问 Exported 或 NotExported 中的非公共元素。当 JVM 在模块 deepreflection 的类型上使用反射时，会假定对该模块具有可读性，因此不需要编写特殊的代码就可以工作。在编译时，NotExported 仍然是不可访问的，而 Exported 因模块描述符中的 exports 子句而可以访问。从应用程序开发人员的角度来看，NotExported 类在编译时仍被强封装。但从框架的角度来看，NotExported 类在运行时是可以自由访问的。



在 Java 9 中，增加了两个反射对象的新方法：canAccess 和 trySetAccessible (都是在 java.lang.reflect.AccessibleObject 中定义的)。这些方法考虑到这样一个新的现实，即深度反射并不总是被允许的。你可以使用这些方法，而不是处理来自 setAccessible 的异常。

开放整个模块看似有点不妥。但是当不能确定在运行时库或框架使用什么类型时，这种做法是很方便的。因此，当将模块引入代码库时，开放式模块可以在迁移场景中发挥至关重要的作用。更多内容请参阅第 8 章。



然而，当知道需要开放哪些包时（大多数情况下都应该知道），可以有选择性地从一个普通模块中开放所需的包：

```
module deepreflection {
    exports api;
    opens internal;
}
```

请注意在关键字 `module` 之前缺少了关键字 `open`。该模块定义不等于前面的开放式模块定义。此时，只有包 `internal` 中的类型可用于深度反射。虽然 `api` 中的类型被导出，但是不能被深度反射。在运行时，该模块定义比完全开放的模块提供了更强的封装，因为无法在包 `api` 中的类型上进行深度反射。

可以通过开放 `api` 包，从而使这个模块定义等同于开放式模块：

```
module deepreflection {
    exports api;
    opens api;
    opens internal;
}
```

一个包可以同时导出和开放。



实际上，这种组合有点尴尬。一旦设计了导出包，其他模块就不需要对它们进行深度反射了。

`opens` 子句可以像 `exports` 子句那样被限定：

```
module deepreflection {
    exports api;
    opens internal to library;
}
```

此时的语义与期望的一样：只有模块 `library` 可以对包 `internal` 中的类型进行深度反射。合适的 `opens` 可以将范围缩小到一个或多个明确提到的模块。如果可以限定 `opens` 声明，那么最好这样做。这样一来，就可以防止任意模块通过深度反射窥探内部细节信息。

有时需要对第三方模块进行深度反射。在某些情况下，有些库甚至想反射访问 JDK 平台模块的私有部分。此时添加 `open` 关键字并重新编译模块是不可能的。针对这些情况，为 Java 命令引入了一个命令行标志：

```
--add-opens <module>/<package>=<targetmodule>
```



上面的命令行标志等同于将 `module` 中的 `package` 限定开放给 `targetmodule`。例如，如果框架模块 `myframework` 想要使用 `java.lang.ClassLoader` 的非公共部分，则可以将以下选项添加到 Java 命令：

```
--add-opens java.base/java.lang=myframework
```

该命令行选项应该被认为是一个逃生入口，在对那些没有使用模块系统编写的代码进行迁移时该选项是非常有用的。在第二部分中，还会重复出现该选项以及其他类似的选项。

6.1.3 依赖注入

在 Java 模块系统中，开放式模块和包是支持现有依赖注入框架的关键。在完全模块化的应用程序中，依赖注入框架使用开放式包来访问非导出类型。

取代反射

Java 9 为应用程序中非公共类成员的基于反射的框架访问提供了一种替代方案：`MethodHandles` 和 `VarHandles`。后者是通过 JEP193 (<http://openjdk.java.net/jeps/193>) 在 Java 9 中引入的。应用程序可以将具有适当权限的 `java.lang.invoke.Lookup` 实例传递给框架，显式委派私有查找功能。然后，框架模块使用 `MethodHandles.privateLookupIn(Class, Lookup)` 访问应用程序模块类中的非公共成员。随着时间的推移，框架将会转向使用更具原则性和性能友好的方法来访问应用程序内部构件。可以在本章附带的代码中找到该方法的一个示例（`chapter6/lookup`）。

为了说明开放式模块和包的抽象概念，接下来看一个具体的示例。该示例没有像第 4 章所描述的那样在模块系统中使用服务，而是在一个名为 `spruce` 的模块中提供了一个虚构的第三方依赖注入框架。图 6-3 显示了该示例。在包类型前面添加一个“open”标签来表示开放式图。

该示例应用程序涵盖两个领域：`orders` 和 `customers`。显而易见，它们都是单独的模块，同时，`customers` 域拆分为 API 和实现模块。`main` 模块使用了这两种服务，但又不希望与这两种服务的实现细节产生任何耦合。为此，对这两个服务实现类进行了封装，只有接口被导出并可以被 `main` 模块访问。

到目前为止，该示例所采用的方法与第 4 章关于服务的方法非常相似。`main` 模块与实现细节实现了很好的解耦。但此时并不会通过 `ServiceLoader` 来提供和消费这些服务，相反，将使用 DI 框架将 `OrderService` 和 `CustomerService` 实现注入 `Application` 中。要么使用正确的服务布线来显式配置 `spruce`，要么使用注释，



又或者结合使用这两种方法。`@Inject` 之类的注释通常用于标识与可注入实例类型匹配的注入点：

```
public class Application {
    @Inject
    private OrderService orderService;

    @Inject
    private CustomerService customerService;

    public void orderForCustomer(String customerId, String[] productIds) {
        Customer customer = customerService.find(customerId)
        orderService.order(customer, productIds);
    }

    public static void main(String... args) {
        // 启动 Spruce 并设置布线
    }
}
```

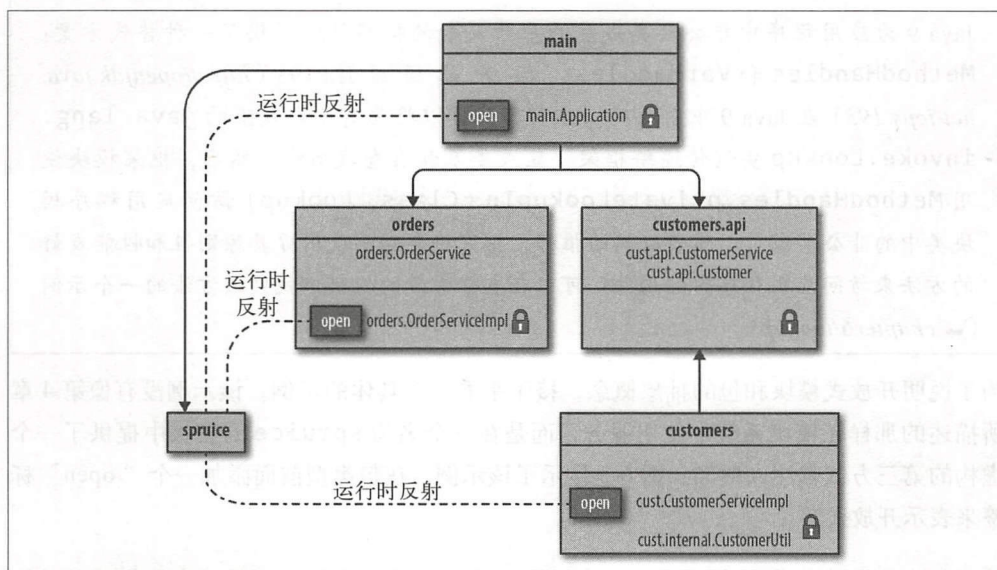


图 6-3：应用程序 main 概述（使用了依赖注入库 `spruce`，模块之间的关系显示为实线边，对开放式包中的类型进行的运行时深度反射显示为虚线边）

在 `Application` 类的 `main` 方法中，使用了 `spruce` API 来引导 DI 框架，因此，`main` 模块需要依赖 `spruce` 来实现其布线 API。服务实现类必须通过 `spruce` 进行实例化并注入注释字段（构造函数注入是另一个可行的选择）。然后，`Application` 准备好接收对 `orderForCustomer` 的调用。



与模块系统的服务机制不同，`spruice` 没有实例化封装类的特殊权限。可以做的是为需要实例化或注入的包添加 `opens` 子句，从而允许 `spruice` 在运行时访问这些类，并在必要时执行深度反射（例如，将 `OrderServiceImpl` 类实例化并注入到 `Application` 的 `OrderService` 私有字段）。仅在模块内部使用的包不需要开放，如 `customers` 模块中的 `cust.internal`。`opens` 子句可以限定仅对 `spruice` 开放。但不幸的是，这样做会将 `orders` 和 `customers` 模块绑定到这个特定的 DI 框架。如果对开放没有任何限制，那么就可能在重新编译这些模块的情况下改变 DI 实现。

图 6-3 揭示了 `spruice` 的真正含义：一个模块可以深入到正在构建的应用程序的每个角落。根据布线配置，它可以找到封装的实现类，然后进行实例化并注入 `Application` 的私有字段中。同时，该设置允许应用程序像使用服务和 `ServiceLoader` 一样实现很好的模块化（无须使用 `ServiceLoader` API 来检索服务）。它们就像是由魔法（反射）注入的一样。

此时失去的是 Java 模块系统了解和验证模块之间服务依赖关系的能力。模块描述符中没有 `provides/uses` 子句来验证。并且，应用程序模块中的包需要开放。可以使所有应用程序模块公开模块，因此应用程序开发人员不必为每个包做出选择。当然，这是以允许对每个应用程序模块中所有程序包进行运行时访问和深度反射为代价的。只需稍微了解一下库和框架所要完成的工作，就没有必要使用这个重量级的方法。

6.2 节将介绍对模块自身的反射。同样，这也是模块系统 API 的高级使用，不应该经常出现在正常应用程序开发中。

6.2 对模块的反射

反射允许在运行时对所有 Java 元素进行具体化。类、包、方法等都有反射表示。前面已经看到了开放式模块如何在运行时允许对这些元素进行深度反射。

随着 Java 新结构元素——模块的增加，反射也需要扩展。在 Java 9 中，可以使用 `java.lang.Module` 提供模块的运行时视图。该类的方法可以分为三个不同的职责：

内省

查询给定模块的属性。

修改

动态更改模块的特性。

访问



从模块内部读取资源。

最后一种情况在 5.8.1 节已经讨论过了，在本节的其余部分将主要介绍内省和修改模块。

6.2.1 内省

`java.lang.Module` 类是对模块反射的入口点。`Module` 及其相关类如图 6-4 所示。它有一个 `ModuleDescriptor`，提供了有关 `module-info` 内容的运行时视图。

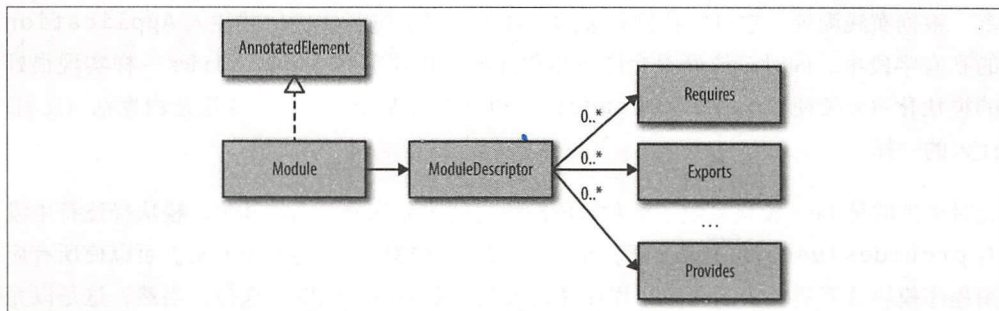


图 6-4: `Module` 及其相关类的简化类图

可以通过模块中的一个类获取 `Module` 实例：

```
Module module = String.class.getModule();
```

`getModule` 方法返回包含该类的模块。在该示例中，`String` 类无疑来自于 `java.base` 平台模块。在本章的后面，将会看到通过新的 `ModuleLayer` API 按名称获取模块实例的另一种方式，而不需要知道模块中的类。

可以使用多种方法查询模块中的信息，如示例 6-1 所示。

示例 6-1: 在运行时检查模块 (☛ [chapter6/introspection](#))

```
String name1 = module.getName(); // 在 module-info.java 中定义的名称
Set<String> packages1 = module.getPackages(); // 列出模块中所有的包

// 上述方法是从 Module 的 ModuleDescriptor 中返回信息的便利方法
ModuleDescriptor descriptor = module.getDescriptor();
String name2 = descriptor.name(); // 同 module.getName();
Set<String> packages2 = descriptor.getPackages(); // 同 module.getPackages();

// 通过 ModuleDescriptor，来自 module-info.java 的所有信息都被公开：
Set<Exports> exports = descriptor.getExports(); // 所有可能的导出
Set<String> uses = descriptor.getUses(); // 该模块使用的所有服务
```

上面的示例虽然并不详尽，但却说明了 `module-info.class` 中的所有信息都可以通过



`ModuleDescriptor` 类获得。`ModuleDescriptor` 的实例是只读的（不可变的）。例如，不可能在运行时更改模块的名称。

6.2.2 修改模块

可以执行模块上其他几个影响该模块及其环境的操作。假设有一个未导出的包，但根据运行时的决策需要将其导出：

```
Module target = ...; // 以某种方式获得想要导出到的模块
Module module = getClass().getModule(); // 获取当前类的模块
module.addExports("javamodularity.export.atruntime", target);
```

可以通过 `Module` API 向特定的模块添加限制导出。现在，目标模块可以访问先前封装的 `javamodularity.export.atruntime` 包中的代码。

此时你可能会怀疑这是否是一个安全漏洞：是否可以在任意模块上调用 `addExports`，从而放弃自身的秘密？事实并非如此。当尝试将导出添加到正在执行调用的当前模块之外的任何其他模块时，VM 就会引发异常。无法通过模块反射 API 从外部升级模块的权限。

调用者敏感

那些从不同地方调用时行为方式不同的方法被称为调用者敏感（*caller sensitive*）方法。可以在 JDK 源代码中找到许多用 `@CallerSensitive` 进行注释的方法，比如 `addExports`。调用者敏感方法可以根据当前的调用堆栈找出哪个类（和模块）正在调用它们。获取该信息和基于该信息做出决策的权限是为 `java.base` 中的代码保留的（虽然在 JDK 9 中通过 JEP 259 引入的新 `StackWalker` API 也为应用程序代码提供了这种可能性）。该机制的另一个示例可以在 `setAccessible` 的实现中找到，如“开放式模块和包”一节（6.1.2 节）中所述。从包未对其开放的模块上调用此方法会导致异常，而从允许进行深度反射的模块中调用它时则会成功。

`Module` 提供了四个允许运行时修改的方法：

```
addExports(String packageName, Module target)
```

将先前没有导出的包公开给另一个模块。

```
addOpens(String packageName, Module target)
```

开放一个包，以便另一个模块进行深度反射。

```
addReads(Module other)
```

添加当前模块到另一个模块的读取关系。



```
addUses(Class<?> serviceType)
```

表明当前模块想要通过 `ServiceLoader` 使用额外的服务类型。

此时，没有 `addProvides` 方法，因为公开在编译时未知的新实现是一种比较少见的情况。

了解模块上的反射 API 是很有好处的。但是，事实上该 API 在正常应用程序开发过程中很少使用。在使用反射之前通常会试图通过正常的方式在模块之间公开足够的信息。在运行时使用反射会改变模块的行为，从而违背 Java 模块系统的原理。如果出现了在编译时或启动时没有考虑到的隐式依赖关系，就会使模块系统在早期阶段所提供的许多保证变得无效。

6.2.3 注释

模块也可以注释。在运行时，可以通过 `java.lang.Module` API 读取这些注释。可以将一些作为 Java 平台一部分的默认注释应用于模块，如 `@Deprecated` 注释：

```
@Deprecated
module m {

}
```

添加该注释表明模块的用户应该寻找一个替代模块。



从 Java 9 开始，可以对那些不建议使用的元素进行标记，以便在未来的版本中将其删除：`@Deprecated(forRemoval = true)`。有关增强弃用功能的更多详细信息，请参阅 JEP 277 (<http://openjdk.java.net/jeps/277>)。在 JDK 9 中，有几个平台模块（例如，`java.xml.ws` 和 `java.corba`）被标记为删除。

当需要使用一个不建议使用的模块时，编辑器会生成一条警告信息。另一个可应用于模块的默认注释是 `@SuppressWarnings`。

也可以为模块定义自己的注释。为此，定义新的目标元素类型 `MODULE`，如示例 6-2 所示。

示例 6-2：注释一个模块（`chapter6/annotated_module`）

```
package javamodularity.annotatedmodule;

import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Retention(RetentionPolicy.RUNTIME)
```




```
@Target(value={PACKAGE, MODULE})
public @interface CustomAnnotation {
}
```

现在，`CustomAnnotation` 可以应用于包和模块。在一个模块上使用自定义注释揭示了另一个奇怪的事实：模块声明可以使用 `import` 语句。

```
import javamodularity.annotatedmodule.CustomAnnotation;

@CustomAnnotation
module annotated { }
```

如果没有 `import` 语句，模块描述符将无法编译。或者，也可以直接使用注释的完全限定名称，而不是 `import`。



也可以在模块描述符中使用 `import`，从而缩短 `uses/provides` 子句。

图 6-4 显示了 `Module` 如何实现 `AnnotatedElement`。在代码中，可以使用 `Module` 实例上的 `getAnnotations` 来获取模块上所有注释的数组。`AnnotatedElement` 接口还提供了其他方法来查找正确的注释。



只有在注释的保留策略设置为 `RUNTIME` 时才有效。

除了平台定义的注释（如 `@Deprecated`）之外，框架（甚至是构建工具）也会使用模块注释。

6.3 容器应用程序模式

接下来重点关注模块系统更高级的应用。此时重申一下本章开头所给出的建议：如果是第一次学习模块系统，那么可以放心地跳过本章的其余部分。

在初步了解了模块系统之后，可以开始深入研究高级 API！到目前为止，都是将模块化应用看作一个单一的实体。首先收集模块，并将它们放在模块路径，然后启动应用程序。尽管在大多数情况下这是一个行之有效的方法，但是另外一种类型的应用程序在结构上与其存在很大不同。



可以将一个应用程序视为其他应用程序的容器，或者假设有一个仅定义了核心功能的应用程序，并且期望由第三方对其进行扩展。在前一种情况下可以将新应用程序部署到正在运行的容器应用程序中，而后一种情况通常是通过类似插件的架构来实现的。这两种情况都不会从一个包含所有模块的模块路径开始。在容器应用程序的生命周期内，新的模块会不断地“来来去去”。目前，许多容器应用程序都建立在诸如 OSGi 或 JBoss Modules 之类的模块系统上。

本节将会学习如何使用 Java 模块系统构建一个容器或基于插件的系统。在研究这些容器应用程序模式的实现之前，首先将探索启用这些模式的新 API。请记住，这些新的 API 是专门为目前讨论的用例而引入的。如果不需要构建一个可扩展的容器式应用程序时，那么就不太可能使用这些 API。

6.3.1 层和配置

当使用 Java 命令启动模块时，将会解析模块图。解析器使用来自平台本身和模块路径的模块来创建一组解析模块。这主要是根据模块描述符中的 `requires` 子句和 `provides/uses` 子句完成的。完成解析后就不能再更改所生成的模块图了。

这似乎与容器应用程序的要求背道而驰。此时，向正在运行的容器添加新模块的能力是至关重要的。必须引入一个新的概念来实现这个功能：层。层对于模块来说就像是类加载器对类那样——一种加载和实例化机制。

已解析的模块图位于 `ModuleLayer` 内。层设置了一组连贯的模块。层本身可以引用父层，并形成非循环图。但是在进一步学习之前，先了解一下 `ModuleLayer`。

当使用 Java 启动一个模块时，Java 运行时将构建一个被称为引导层 (*boot layer*) 的初始层。该层包含了在解析提供给 Java 命令 (以带有 `-m` 的初始模块形式提供，或者使用 `--add-modules`) 的根模块之后所生成的模块图。

在图 6-5 中，显示了在启动需要 `java.sql` 的 `application` 模块之后的引导层的简化示例。(实际上，由于平台模块的服务绑定，引导层包含了更多的模块。)

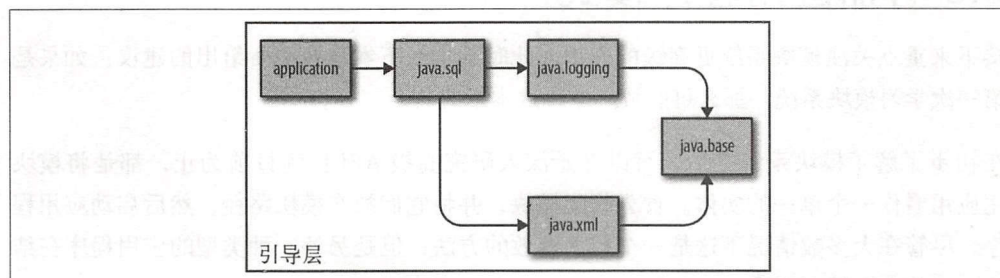


图 6-5: 在启动了模块 `application` 之后引导层中的模块



可以使用示例 6-3 所示的代码枚举引导层中的模块。

示例 6-3: 打印引导层中的所有模块 (`chapter6/bootlayer`)

```
ModuleLayer.boot().modules()
    .forEach(System.out::println);
```

由于引导层是 Java 运行时所构建的, 因此如何创建另一个层的问题仍然存在。在构建 `ModuleLayer` 之前, 必须在该层中创建一个描述模块图的 `Configuration`。同样, 对于引导层, 该过程也是在启动时隐式完成的。在构建 `Configuration` 时, 需要提供一个 `ModuleFinder` 来定位各个模块。设置一连串类以创建一个新的 `ModuleLayer`, 如下代码所示:

```
ModuleFinder finder = ModuleFinder.of(Paths.get("./modules")); ❶

ModuleLayer bootLayer = ModuleLayer.boot();

Configuration config = bootLayer.configuration()
    .resolve(finder, ModuleFinder.of(), Set.of("rootmodule")); ❷

ClassLoader scl = ClassLoader.getSystemClassLoader();
ModuleLayer newLayer = bootLayer.defineModulesWithOneLoader(config, scl); ❸
```

- ❶ 一种创建 `ModuleFinder` 的便捷方法, `ModuleFinder` 可以在文件系统的某一个或者多个路径上查找模块。
- ❷ 相对于父配置进行配置解析; 此时是相对于引导层的配置。
- ❸ 通过 `Configuration`, 可以构建一个 `ModuleLayer`, 从而根据配置具体化已解析的模块。

原则上, 模块可以来自任何地方。通常它们位于文件系统的某个位置, 所以 `ModuleFinder::of(Path...)` 工厂方法使用非常方便, 它返回一个可以从文件系统加载模块的 `ModuleFinder` 实现。除 `ModuleLayer::empty` 和 `Configuration::empty` 方法返回的实例之外, 每个 `ModuleLayer` 和 `Configuration` 都指向一个或多个父级。这些特殊实例作为模块系统中的 `ModuleLayer` 和 `Configuration` 层次结构的根。引导层和对应配置将空项作为父项。

在构建新层时, 使用引导层及其配置作为父级。在 `resolve` 调用中, `ModuleFinder` 作为第一个参数传递, 而第二个参数是另一个 `ModuleFinder` (在本示例中, 第二个参数为空), 当第一个查找器或父配置中找不到模块时就会查阅第二个查找器。

在解析新配置中的模块时也会考虑父配置中的模块, 新构建配置中的模块可以从父配置中读取模块。启动解析器的根模块将作为第三个参数传递给配置的 `resolve` 方法。在这种情况下, `rootmodule` 充当解析器的初始模块。新配置中的解析受到前面所介绍的



约束的限制。如果找不到根模块或其某个依赖项，则会失败。模块之间的循环是不允许的，同时导出同一个包的两个模块也不能被另外一个模块读取。

为了扩展这个示例，假设 `rootmodule` 需要 `javafx.controls` 平台模块和 `library`（一个驻留在 `./modules` 目录中的辅助模块）。解析完配置并构建新层后，结果如图 6-6 所示。

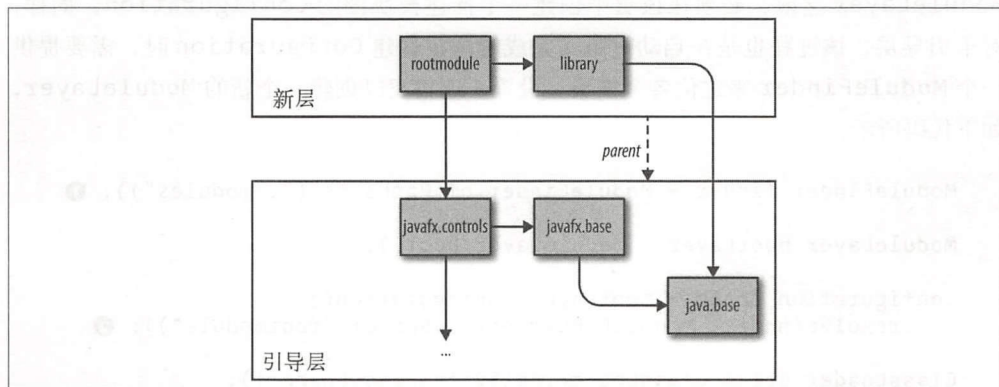


图 6-6：引导层作为父层的新层

可读性关系可以跨越层边界。`rootmodule` 到 `javafx.controls` 的 `requires` 子句被解析到引导层的平台模块中。另一方面，到 `library` 的 `requires` 子句在新构建的层中被解析，因为该模块与来自文件系统的 `rootmodule` 一起被加载。

除了 `Configuration` 上的 `resolve` 方法外，还有 `resolveAndBind`。考虑到新配置中模块的 `provides/uses` 子句，`resolveAndBind` 方法也会进行服务绑定。服务也可以跨越层边界。新层中的模块可以使用来自父层的服务，反之亦然。

最后，在父（引导）层调用 `defineModulesWithOneLoader` 方法，该方法将 `Configuration` 所解析的模块引用转化为新层内的 `Module` 实例。下一节将讨论传递给 `defineModulesWithOneLoader` 方法的类加载器的重要性。

到目前为止，所看到的所有层示例都是以引导层作为父层的新层所组成的。但是，层也可以指向除引导层之外的父层，甚至有可能一个层有多个父层。如图 6-7 所示：第 3 层指向第 1 层和第 2 层作为其父节点。

`ModuleLayer` 上的静态 `defineModules*` 方法在构建新层时接收一个父层列表，而不是直接调用父层实例上任何非静态的 `defineModules*` 方法。稍后在图 6-12 中将会看到这些静态方法是如何使用的。重要的是要记住层可以形成一个非循环图，就像一个层内的模块一样。

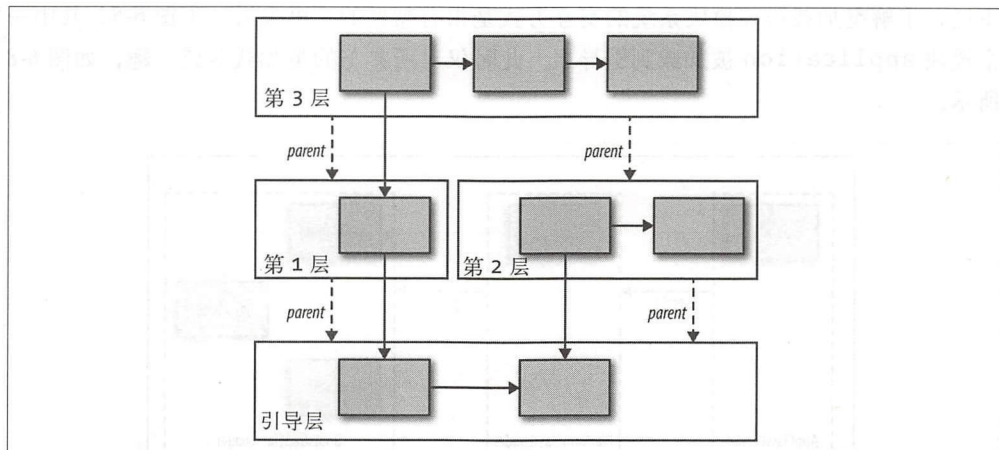


图 6-7：层可以形成一个非循环图

6.3.2 层中的类加载

此时，你可能仍然在想上一节中层构造代码的最后两行：

```
ClassLoader scl = ClassLoader.getSystemClassLoader();
ModuleLayer newLayer = bootLayer.defineModulesWithOneLoader(config, scl);
```

什么是类加载器和 `defineModulesWithOneLoader` 方法？回答这个问题不是那么容易的。在找到回答问题的方法之前，最好先了解一下什么是类加载器，及其对模块系统意味着什么。

毫无疑问，类加载器在运行时加载类。类加载器决定了可见性：如果某个类对于某个类加载器或其委托的其他类加载器可见，那么该类就可以加载。从某种意义上讲，在本书中介绍类加载器是非常奇怪的。像 OSGi 这样的早期模块系统使用类加载器作为强制封装的主要手段，每个包（OSGi 模块）都有它自己的类加载器，类加载器之间的委托遵循 OSGi 元数据中所表示的 bundle 布线。

在 Java 模块系统中却不是这样的。它建立了一个全新的机制，包括可读性和新的可访问性规则，而类加载通常是不变的。这是一个慎重的选择，因为使用类加载器进行隔离并不是一个万无一失的解决方案。加载类之后，Class 实例可以自由传递，绕过通过类加载器隔离和委托所设置的任何方案。虽然可以尝试在模块系统中执行此操作，但可以看到，由于存在封装，从不允许访问的 Class 对象创建实例会导致异常。模块系统在更深的层面上执行了封装。此外，类加载器只是一个运行时构造，而 Java 模块系统在编译时也会执行封装。最后，许多现有的代码库对默认情况下类的加载方式进行了假设，改变这些默认值（例如，给每个模块一个自己的类加载器）会破坏现有的代码。



不过，了解类加载器与模块系统的交互方式是很有帮助的。再来看一下图 6-5，其中一个模块 `application` 被加载到引导层。此时仅对所参与的类加载器感兴趣，如图 6-8 所示。

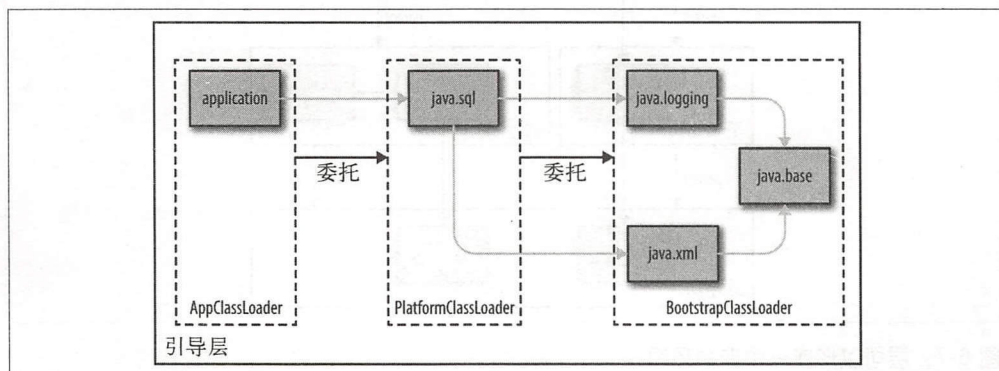


图 6-8：当启动模块 `application` 时引导层中的类加载器

当从模块路径运行应用程序时，有三个类加载器在引导层中处于活动状态。在委托层的底部是 `BootstrapClassLoader`，也被称为原始类加载器 (*primordial classloader*)。这是一个加载所有基本平台模块类的特殊类加载器。在该类加载器中已经采取了谨慎措施，以便从尽可能少的模块加载类，因为类在引导加载程序中加载时被授予所有安全权限。

接下来是 `PlatformClassLoader`，加载较少权限的平台模块类。最后一个是 `AppClassLoader`，负责加载用户定义的模块和一些特定于 JDK 的工具模块 (比如，`jdk.compiler` 或 `jdk.javadoc`)。所有的类加载器都委托给底层的类加载器。实际上，这使得所有类都对 `AppClassLoader` 可见。这种三路设置类似于类加载器在模块系统之前的工作方式，主要是为了向后兼容。

在本节的开头，曾经提过一个问题：为什么需要将一个类加载器传递给创建 `ModuleLayer` 的方法：

```

ClassLoader scl = ClassLoader.getSystemClassLoader();
ModuleLayer newLayer = bootLayer.defineModulesWithOneLoader(config, scl);
  
```

即使在引导层中预定义了从模块到类加载器的映射，新层的创建者也必须指出哪些类加载器为哪些模块加载类。`ModuleLayer::defineModulesWithOneLoader (Configuration, ClassLoader)` 方法是一个方便的方法。它设置了新的层，以便层中的所有模块都由一个新创建的类加载器加载。这个类加载器委托给作为参数传递的父类加载器。在本示例中，传递了 `ClassLoader::getSystemClassLoader` 的结果，并返回 `AppClassLoader`，它是负责在引导层中加载用户定义模块类的类加载器



(还有一个 `getPlatformClassLoader` 方法)。

因此，示例中新构建的层（如图 6-6 所示）的类加载器视图如图 6-9 所示。

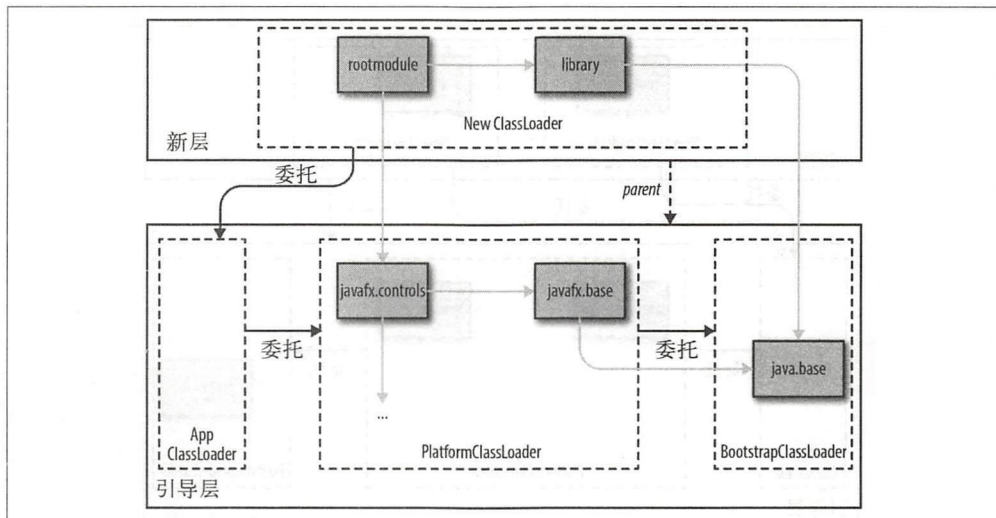


图 6-9：在新层中为所有的模块创建单个类加载器

即使是跨层，类加载器之间的委托也必须遵守模块之间的可读性关系。如果新的类加载器没有在引导层中委托给 `AppClassLoader`，而是委托给 `BootstrapClassLoader`，则会产生问题。由于 `rootmodule` 读取 `javafx.controls`，因此必须能够查看并加载这些类。而将新层的类加载器委托给 `AppClassLoader` 可以确保这一点。继而，`AppClassLoader` 委托给 `PlatformClassLoader`，后者从 `javafx.controls` 加载类。

还可以使用其他方法创建一个新层。另一个被称为 `defineModulesWithManyLoaders` 的简便方法为层中的每个模块创建一个新的类加载器，如图 6-10 所示。

同样，这些新的类加载器都委托给作为参数传递给 `defineModulesWithManyLoaders` 的父类加载器。如果需要对层中的类加载进行更多的控制，可以使用 `defineModules` 方法。它接收一个将字符串（模块名称）映射到类加载器的函数。当需要为新模块创建新类加载器或将现有类加载器分配给层中模块时，这种映射提供了极大的灵活性。可以在 JDK 本身找到这种映射的示例。引导层是使用 `defineModules` 创建的，具有自定义映射到图 6-8 所示的三个类加载器之一的功能。

为什么在使用层时控制类的加载非常重要？因为可以取消模块系统中许多限制。例如，前面讨论了单个模块如何才能包含（或导出）某个包。事实证明，这只是创建引



导层的一个副作用。一个包只能定义一个类加载器。来自模块路径的所有模块都由 `AppClassLoader` 加载。因此，如果这些模块中的任何一个模块包含相同的包（不管是否导出），那么一旦将它们定义到相同的 `AppClassLoader`，就会导致运行时异常。

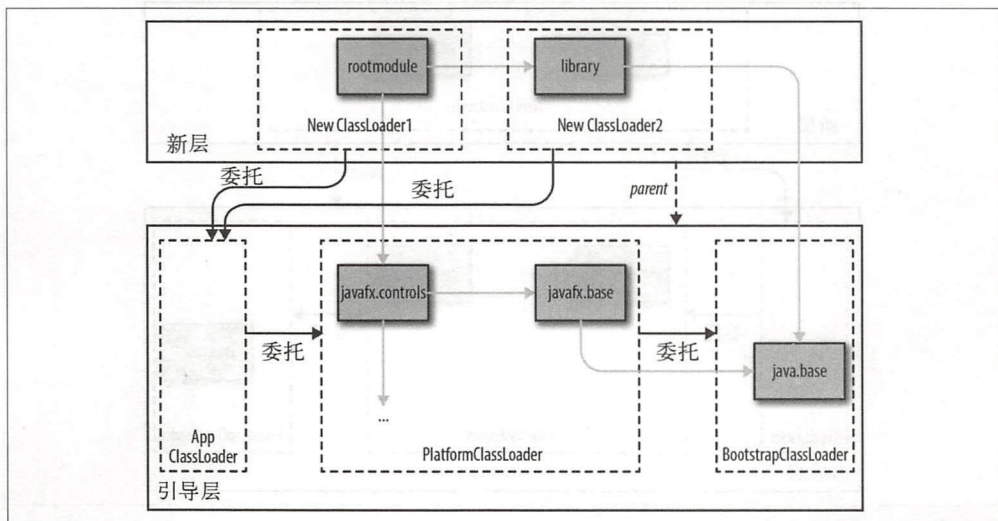


图 6-10：层中使用 `defineModulesWithManyLoaders` 构建的每个模块都拥有自己的类加载器

当使用新的类加载器实例化新层时，相同的程序包可能会出现在该层的不同模块中。在这种情况下，不会出现上述问题，因为包被定义为两个不同的类加载器。稍后会看到，这也意味着同一模块的多个版本可以存在于不同的层中。这是对 5.7 节中讨论的情况的很大改进，尽管这是以提高创建和管理层的复杂性为代价的。

6.3.3 插件体系结构

到目前为止，已经学习了层以及层中类加载的基础知识，接下来可以在实践中应用这些知识了。首先，创建一个可以在运行时使用新插件进行扩展的应用程序。在很多方面与第 4 章中使用 `EasyText` 和服务所完成的事情是相类似的。当使用服务方法时，只需将新的分析提供者放在模块路径上，应用程序在启动时就会提取它。虽然这种方法已经非常灵活了，但是如果这些新的提供者模块来自第三方开发者呢？如果这些分析提供者不是在启动时放在模块路径上，而是在运行时引入，又该怎么办呢？

为了满足上述需求，需要使用更加灵活的插件体系结构。Eclipse IDE 是一个众所周知的基于插件的应用程序示例。Eclipse 本身仅提供了基本的 IDE 功能，但可以通过添加插件以多种方式进行扩展。使用插件的应用程序的另一个例子是 JDK 中的新 `jlink` 工具，



它可以通过与接下来将要介绍的相类似的插件机制进行新的优化扩展。第 13 章讨论了 jlink 工具可以使用的插件的更多细节。

一般来说，可以找到一个通过插件扩展的插件主机应用程序，如图 6-11 所示。

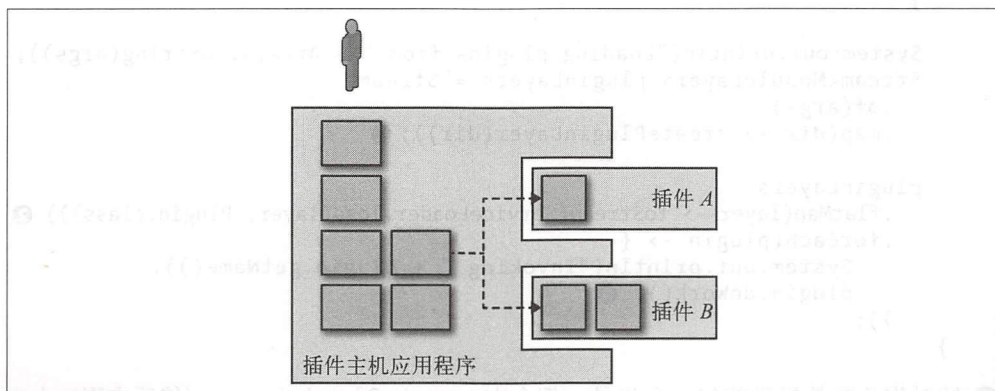


图 6-11：插件在主机应用程序功能基础之上提供了额外功能

虽然用户与主机应用程序进行交互，但凭借插件可以体验到扩展功能。在大多数情况下，主机应用程序可以在没有任何插件的情况下正常运行。插件通常独立于主机应用程序开发，主机应用程序和插件之间存在明确的界限。在运行时，主机应用程序通过调用插件来获取额外功能。为此，插件必须实现一致的 API。通常，插件实现的是一个接口。

你可能已经猜到，层在实现一个动态的、基于插件的应用程序中扮演着重要的角色。接下来将要创建一个 `pluginhost` 模块，并为每个插件创建一个新的 `ModuleLayer`。这些插件模块（在本示例中为 `plugin.a` 和 `plugin.b`）与其依赖项（如果有的话）位于不同的目录中。关键是，当启动 `pluginhost` 时这些目录不在模块路径上。

示例使用了带有 `pluginhost.api` 模块的服务，该模块公开了接口 `pluginhost.api.Plugin`，而该接口具有单一方法 `doWork`。虽然两个插件模块都需要该 API 模块，但与 `pluginhost` 应用程序模块之间不存在编译时关系。插件模块由作为服务提供的 `Plugin` 接口实现组成。示例 6-4 显示了 `plugin.a` 模块的模块描述符。

示例 6-4: `module-info.java` (`chapter6/plugins`)

```
module plugin.a {
    requires pluginhost.api;

    provides pluginhost.api.Plugin
        with plugina.PluginA;
}
```

插件实现类 `PluginA` 没有被导出。



在 pluginhost 中，main 方法从参数所提供的目录中加载插件模块：

```

if (args.length < 1) {
    System.out.println("Please provide plugin directories");
    return;
}

System.out.println("Loading plugins from " + Arrays.toString(args));
Stream<ModuleLayer> pluginLayers = Stream
    .of(args)
    .map(dir -> createPluginLayer(dir)); ❶

pluginLayers
    .flatMap(layer -> toStream(ServiceLoader.load(layer, Plugin.class))) ❷
    .forEach(plugin -> {
        System.out.println("Invoking " + plugin.getName());
        plugin.doWork(); ❸
    });
}
    
```

- ❶ 针对作为参数提供的每一个目录，都会在 createPluginLayer (稍后实现) 中实例化一个 ModuleLayer。
- ❷ ServiceLoader::load 调用是以每一层作为参数执行的，并从该层返回插件服务。
- ❸ 在将服务整合为单个流之后，就可以在所有插件上调用 doWork 方法。

现在使用的是 ServiceLoader::load 的一个尚未讨论过的重载。它需要以层作为参数。通过为插件传递新构建的层，该调用将从加载到层的插件模块中返回新加载的 Plugin 服务提供者。

通过从模块路径运行 pluginhost 模块来启动应用程序。同样，没有任何插件模块在模块路径上。插件模块位于不同的目录中，并由主机应用程序在运行时加载。

在以两个插件目录作为参数启动 pluginhost 之后，会出现如图 6-12 所示的运行时情况。

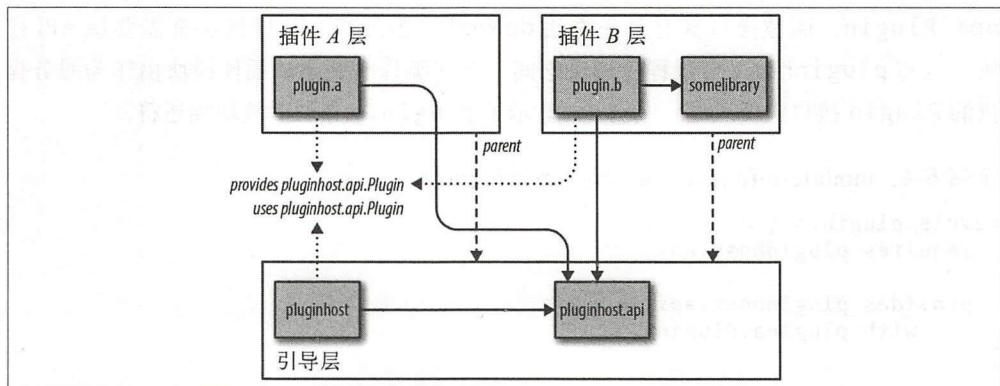


图 6-12：每个插件在各层被实例化



第一个插件由单个模块组成。而插件 *B* 依赖于 `somelibrary`，当为此插件创建配置和层时，将会自动解析此依赖关系。只要 `somelibrary` 和 `plugin.b` 在同一个目录下，程序就可以顺利运行。这两个插件都需要 `pluginhost.api` 模块，它是引导层的一部分。所有其他交互都是通过由插件模块发布并由主机应用程序使用的服务产生的。

`createPluginLayer` 方法如下所示：

```
static ModuleLayer createPluginLayer(String dir) {
    ModuleFinder finder = ModuleFinder.of(Paths.get(dir));

    Set<ModuleReference> pluginModuleRefs = finder.findAll();
    Set<String> pluginRoots = pluginModuleRefs.stream()
        .map(ref -> ref.descriptor().name())
        .filter(name -> name.startsWith("plugin")) ❶
        .collect(Collectors.toSet());

    ModuleLayer parent = ModuleLayer.boot();
    Configuration cf = parent.configuration()
        .resolve(finder, ModuleFinder.of(), pluginRoots); ❷

    ClassLoader scl = ClassLoader.getSystemClassLoader();
    ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl); ❸

    return layer;
}
```

- ❶ 为了在解析 `Configuration` 时识别根模块，保留了所有以 `plugin` 开头的模块。
- ❷ 相对于引导层，`Configuration` 已被解析，所以插件模块可以读取 `pluginhost.api`。
- ❸ 插件层中的所有模块将使用相同的（新鲜的）类加载器来定义。

由于针对每个插件目录都会调用 `createPluginLayer` 方法，所以会创建多个层。每个层都有一个能独立解析的根模块（分别为 `plugin.a` 和 `plugin.b`）。由于 `plugin.b` 需要 `somelibrary`，如果找不到该模块，将会抛出一个 `ResolutionException` 异常。只有满足模块描述符中表达的所有约束条件时才能创建配置和层。



除了调用 `resolve(..., pluginRoots)` 之外，也可以调用 `resolveAndBind(finder, ModuleFinder.of(), Set.of())`（没有提供要解析的根模块）。由于 `plugin` 模块公开了一个服务，因此服务绑定导致了 `plugin` 模块及其依赖项的解析。

使用新的类加载器将每个插件模块加载到自己的层中还有另一个好处。通过这种方式来隔离插件，可以让插件依赖于相同模块的不同版本。在 5.7 节中，讨论了模块系统如何加载导出了相同包的同名模块。现在我们知道这取决于类加载器的设置方式。但是当谈



到从模块路径构建的引导层时，这种做法会出现问题。

当构建多个层时，可以同时加载模块的不同版本。例如，如果插件 A、B 依赖于 somelibrary 不同的版本，就会出现上述情况，如图 6-13 所示。

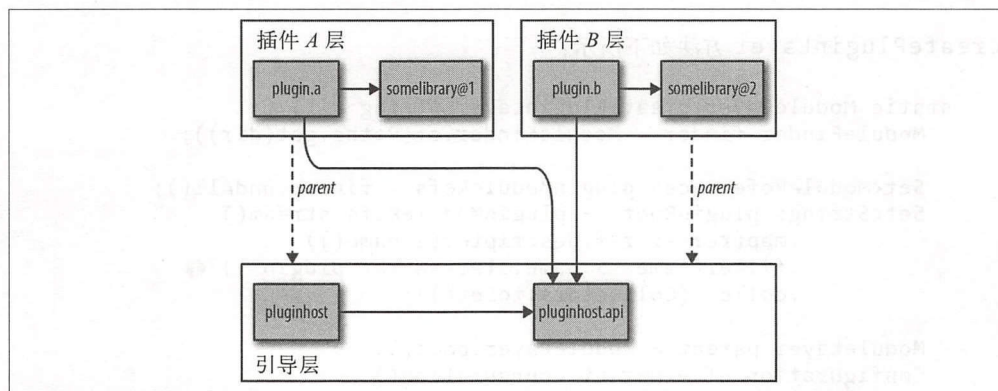


图 6-13：在多个层中可以加载同一模块的不同版本

此时，不需要进行任何代码修改。因为在层中，模块被加载到一个新的类加载器中，所以在包定义中不会发生冲突。

这种为每个插件设置一个新层的设置带来了许多好处。插件作者可以自由选择自己的依赖项。在运行时，保证不会与其他插件的依赖项发生冲突。

需要思考的一个问题是，当在两个插件层之上创建另一个层时会发生什么事情。层可以有多个父级，所以可以创建一个新层，并将两个插件层作为父级：

```
List<Configuration> parentConfigs = pluginLayers
    .map(ModuleLayer::configuration)
    .collect(Collectors.toList());
Configuration newconfig = Configuration.resolve(finder, parentConfigs, ❶
    ModuleFinder.of(), Set.of("topmodule"));
ModuleLayer.Controller newlayer = ModuleLayer.defineModulesWithOneLoader(
    newconfig, pluginLayers, ClassLoader.getSystemClassLoader()); ❷
```

❶ 此静态方法可以接收多个配置作为父级。

❷ 层结构也带有多个父级。

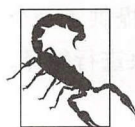
假设这个新层包含一个名为 topmodule 的单个（根）模块，该模块依赖 somelibrary。使用 somelibrary 的哪个版本来解析 topmodule 模块呢？静态 resolve 和 defineModulesWithOneLoader 方法都接收了一个 List 作为父类参数，这应该视作一个警告标志。由此可以看出，顺序非常重要。当解析完配置后，将按照所提供列表的顺序查阅父配置列表。因此，根据优先放入 parentConfigs 列表中的插件配置，



topmodule 将使用 somelibrary 的版本 1 或版本 2。

6.3.4 容器体系结构

另一种能够在运行时加载新代码的体系结构是应用程序容器体系结构。容器中应用程序之间的隔离是另一个大的主题。多年来，Java 已经了解了许多实现 Java EE 标准的应用程序容器或应用程序服务器。



尽管应用程序服务器在应用程序之间提供了一定程度的隔离，但最终所有部署的应用程序仍然运行在同一个 JVM 中。要实现真正的隔离（即限制内存和 CPU 利用率）需要一种更普遍的方法。

在围绕层设计需求时，Java EE 充当了灵感之一。这并不是说 Java EE 已经与 Java 模块系统保持一致。在编写本书时，尚不清楚哪个版本的 Java EE 将首先使用模块。但是，Java EE 支持 Web Archive (WAR) 和 Enterprise Application Archive (EAR) 的模块化版本并不是不可能的（可以说是相当合理的预期）。

为了了解层如何启用应用程序容器体系结构，接下来将创建一个小的应用程序容器。在查看实现之前，先回顾一下应用程序容器体系结构与基于插件的体系结构之间的不同之处（请参见图 6-14）。

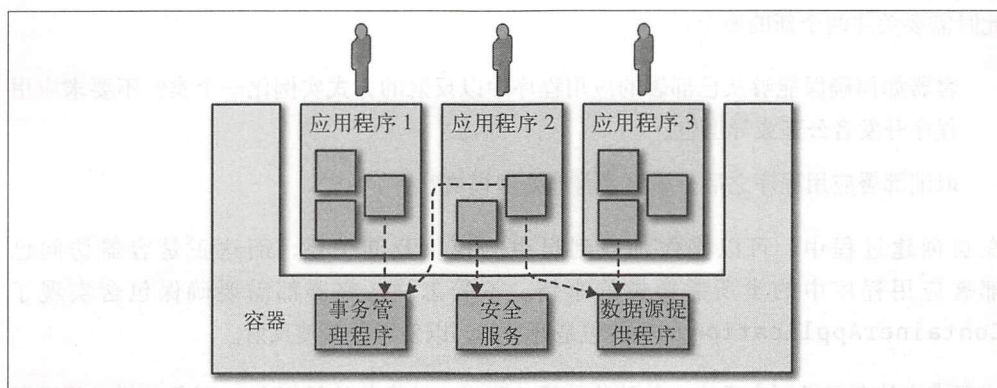


图 6-14：应用程序容器托管多个应用程序，提供了在这些应用程序中使用的公共功能

如图 6-14 所示，可以将多个应用程序加载到单个容器中。虽然应用程序拥有各自的内部模块结构，但更重要的是它们都使用了容器提供的公共服务。诸如事务管理或安全基础结构之类的功能通常由容器提供。应用程序开发人员不必为每个应用程序的运行重新设计“轮子”。



在某种程度上，调用图与图 6-11 正好相反：应用程序使用容器所提供的实现，而不是主机应用程序（容器）调用新加载的代码。当然，前提是必须存在容器和应用程序之间的共享 API。Java EE 就是此类 API 的一个示例。与基于插件的应用程序的另一个较大的区别是，系统的个人用户与动态加载的应用程序本身进行交互，而不是容器。可以考虑已部署的应用程序直接向用户公开 HTTP 端点、Web 界面或队列。最后，可以部署或取消部署容器中的应用程序，也可以由新版本取代。

从功能上讲，容器体系结构与基于插件的体系结构完全不同，但在实施方面却没有太大的区别。就像插件一样，可以在运行时将应用程序模块加载到新层。容器提供了一个 API 模块，该模块包含了所提供服务的接口，这些应用程序根据该模块进行编译。这些服务可以在运行时通过 `ServiceLoader` 在应用程序中使用。

为了让事情变得更有趣，接下来将创建一个可以部署和取消部署应用程序的容器。在前面的插件示例中，我们依靠插件模块公开了一个实现通用接口的服务。而对于容器，需要做完全不同的事情。部署完成后，查找一个实现了 `ContainerApplication` 的类，该类是容器提供的 API 的一部分。容器将以反射的方式从应用程序加载类。这样一来，就不能依赖服务机制与已部署的应用程序进行交互（虽然已部署的应用程序确实通过服务来使用容器功能）。在通过反射实例化该类之后，调用 `startApp` 方法（在 `ContainerApplication` 中定义）。而在取消部署之前，调用 `stopApp`，以便应用程序可以正常关闭。

此时需要关注两个新的概念：

- 容器如何确保能够从已部署的应用程序中以反射的方式实例化一个类？不要求应用程序开发者公开或导出包。
- 取消部署应用程序之后，如何妥善地处理模块？

在层创建过程中，可以操作加载到层中的模块及其关系，而这正是容器访问已部署应用程序中的类所需要做的事情。不管怎样，容器都需要确保包含实现了 `ContainerApplication` 的类的包是开放的，以便进行深度反射。

清理模块是在层级别完成的，并且非常简单明了，就像其他任何 Java 对象一样，层可以被垃圾回收。如果容器确定对层及其任何模块和类不再有强引用，那么层和相关的内容最终都会被垃圾回收。

现在可以看一些代码了。本章作为示例所提供的容器是一个简单的命令行启动器，列出了可以部署和取消部署的应用程序。示例 6-5 给出了它的主类。如果部署应用程序，它将一直运行，直到取消部署（或停止容器）。示例以 `out-appa/app.a/app.a.AppA`



格式将有关可部署应用程序的位置信息作为命令行参数传递：首先是包含应用程序模块的目录，然后是根模块名称，最后是实现了 `ContainerApplication` 的类的名称，全部以斜线分隔。

通常，应用程序容器具有一个部署描述符格式，它将相关信息作为应用程序包的一部分来传递。当然，也可以使用其他方法来实现类似结果，例如，通过将注释放在模块上（如 6.2.3 节所示）来指定一些元数据。为了简单起见，从命令行参数中导出包含此信息的 `AppDescriptor` 实例。

示例 6-5: 启动应用程序容器 (↪ `chapter6/container`)

```
public class Launcher {

    private static AppDescriptor[] apps;
    private static ContainerApplication[] deployedApps;

    public static void main(String... args) {
        System.out.println("Starting container");

        deployedApps = new ContainerApplication[args.length];
        apps = new AppDescriptor[args.length];
        for (int i = 0; i < args.length; i++)
            apps[i] = new AppDescriptor(args[i]);

        // 处理键盘输入和调用部署 / 取消部署 (略)
    }

    // 用于部署 / 取消部署的方法 (稍后讨论)
}
```

该容器的主要数据结构是一个应用程序描述符数组，以及一个用于跟踪已启动的 `ContainerApplication` 实例的数组。容器启动后，可以键入命令，比如 `deploy 1` 或 `undeploy 2`，又或者 `exit`。这些数字指的是 `apps` 和 `deployedApps` 数组中的索引。为每个已部署的应用程序创建一个层。在图 6-15 中，可以看到两个应用程序部署（在输入了 `deploy 1` 和 `deploy 2` 之后）在各自的层中。

除了 `provides` 和 `uses` 关系相反之外，该图与图 6-12 非常相似。在 `platform.api` 中，找到了容器功能的服务接口，比如 `platform.api.tx.TransactionManager`。`platform.container` 模块包含了这些接口的服务提供者，以及之前所看到的 `Launcher` 类。当然，现实世界中的容器和应用程序可能包含更多模块。

为应用程序创建一个层看起来与在加载插件时所完成的事情类似，但略微有所不同：

```
private static ModuleLayer.Controller createAppLayer(AppDescriptor appDescr) {
    ModuleFinder finder = ModuleFinder.of(Paths.get(appDescr.appDir));
    ModuleLayer parent = ModuleLayer.boot();
```



```

Configuration cf = parent.configuration()
    .resolve(finder, ModuleFinder.of(), Set.of(appDescr.rootmodule)); ❶

ClassLoader scl = ClassLoader.getSystemClassLoader();
ModuleLayer.Controller layer =
    ModuleLayer.defineModulesWithOneLoader(cf, List.of(parent), scl); ❷

return layer;
}

```

- ❶ 根据 AppDescriptor 元数据创建 ModuleFinder 和 Configuration。
- ❷ 使用静态 ModuleLayer.defineModulesWithOneLoader 方法创建层，并返回 ModuleLayer.Controller。

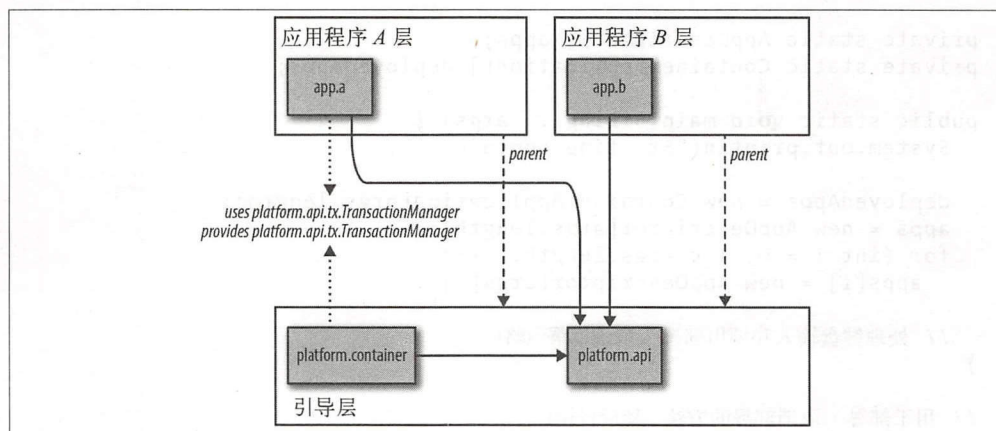


图 6-15: 部署在容器中的两个应用程序

使用一个新的类加载器将每个应用程序都加载到自己的隔离层中，即使应用程序包含具有相同包的模块，也不会发生冲突。通过使用静态 `ModuleLayer::defineModulesWithOneLoader`，可以得到一个 `ModuleLayer.Controller` 对象。

通过调用方法使用此控制器公开根模块中的包，该包包含了在部署应用程序中实现 `ContainerApplication` 的类：

```

private static void deployApp(int appNo) {
    AppDescriptor appDescr = apps[appNo];
    System.out.println("Deploying " + appDescr);

    ModuleLayer.Controller appLayerCtrl = createAppLayer(appDescr); ❶
    Module appModule = appLayerCtrl.layer() ❷
        .findModule(appDescr.rootmodule)
        .orElseThrow(() -> new IllegalStateException("No " + appDescr.rootmodule));

    appLayerCtrl.addOpens(appModule, appDescr.appClassPkg,

```




```
Launcher.class.getModule()); ❸
```

```
ContainerApplication app = instantiateApp(appModule, appDescr.appClass); ❹
deployedApps[appNo] = app;
app.startApp(); ❺
}
```

- ❶ 调用前面所定义的 createAppLayer 方法，获取 ModuleLayer.Controller。
- ❷ 从控制器中可以获取实际层并找到所加载的根模块。
- ❸ 在使用反射以实例化根模块中的应用程序类之前，首先确保给定的包是开放的。
- ❹ 现在，instantiateApp 实现可以使用反射以在不受限的情况下实例化应用程序类。
- ❺ 最后，通过调用 startApp 启动已部署的应用程序。

deployApp 中最有趣的一行代码是调用 ModuleLayer.Controller::addOpens。它将 AppDescriptor 中所提及的包从应用程序的根模块开放给容器模块，如图 6-16 所示。

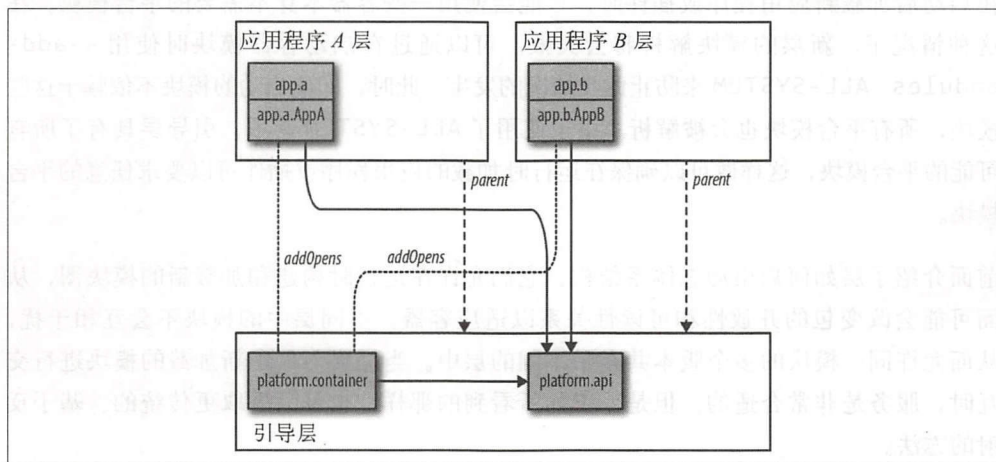


图 6-16: 在创建层的过程中，将包 app.a 和 app.b 开放给 platform.container 模块

这个受限 opens 方法能够让容器反射包 app.a 和 app.b。除了 addOpens(Module source,String pkg,Module target) 之外，还可以在层控制器上调用 addExports (Module source,String pkg,Module target) 或者 add Reads (Module source, Module target)。如果使用 addExports，可以导出先前封装的包（而无须开放）。此外，通过使用 addReads 建立可读性，目标模块可以访问源模块的导出包。无论何种情况，源模块必须来自已经构建的层。实际上，层可以重塑模块的依赖关系和封装边界。一如既往，权力越大，责任也就越大。



6.3.5 解析容器中的平台模块

到目前为止，所描述的容器体系结构中的隔离都是通过在自己的 `ModuleLayer` 中解析新应用程序或插件来实现的。每个应用程序或插件可能需要自己的库，甚至可能需要不同的版本。只要 `ModuleFinder` 可以找到 `ModuleLayer` 所需的模块，一切都没有问题。

但是，这些新加载的应用程序或插件与平台模块之间的依赖关系如何呢？乍一看，这个问题非常容易回答。`ModuleLayer` 可以解析父层中的模块，并最终到达引导层。引导层包含平台模块，所以一切应该没有问题。但事实真的是这样的吗？这取决于在启动容器时将哪些模块解析到引导层。

正常的模块解析规则是：启动的根模块决定哪些平台模块被解析。当根模块是容器启动器时，只考虑容器启动器模块的依赖项。只有根模块的（可传递）依赖项在运行时最终到达引导层。

在启动后加载新应用程序或插件时，可能会使用一些容器本身不需要的平台模块。在这种情况下，新层的模块解析将会失败。可以通过在启动容器模块时使用 `--add-modules ALL-SYSTEM` 来防止此类情况的发生。此时，即使启动的模块不依赖于这些模块，所有平台模块也会被解析。由于使用了 `ALL-SYSTEM` 选项，引导层具有了所有可能的平台模块，这样做可以确保在运行时加载的应用程序或插件可以要求任意的平台模块。

前面介绍了层如何启用动态体系结构。它们允许在运行时构建和加载新的模块图，从而可能会改变包的开放性和可读性关系以适应容器。不同层中的模块不会互相干扰，从而允许同一模块的多个版本共存于不同的层中。当需要与层中新加载的模块进行交互时，服务是非常合适的。但是，正如所看到的那样，也可以采取更传统的、基于反射的方法。

`ModuleLayer` API 在日常的应用程序开发中并没有广泛使用。从某些方面讲，该 API 在性质上类似于类加载器：一个非常强大的功能，主要由框架实现，使开发人员的工作更加轻松。现有的 Java 模块框架预计将使用层作为互操作性的一种手段，就像在过去的 20 年里使用类加载器一样，是否可以好好利用新的 `ModuleLayer` API 取决于框架。



第 7 章

没有模块的迁移

向后兼容性一直是 Java 的主要目标。通常，从开发人员的角度来看迁移到新的 Java 版本通常是微不足道的。模块化系统和模块化 JDK 可以说是自 Java 平台出现以来对整个平台最大的改变。但即便如此，向后兼容性也是重中之重。

将现有的应用程序迁移到 Java 9 最好分两步进行。本章重点介绍如何迁移现有代码以便在 Java 9 上构建和运行，而无须将代码迁移到模块。下一章将深入讨论如何将代码迁移到模块，并提供了实现这一目标的策略。



当不需要使用 Java 9 的旗舰功能——模块系统时，为什么还要迁移到 Java 9 呢？升级到 Java 9 后就可以访问属于 Java 9 的其他功能了，想一想新的 API、工具和性能改进。

究竟是一直使用模块还是从一开始就不使用模块要视情况而定。应用程序是否可以得到更多扩展和新功能？如果可以，那么模块化所带来的好处可以证明采取第二步所付出的代价是值得的。正如本章所述，当应用程序处于维护模式并且只需在 Java 9 上运行时，那么只需要完成执行第一步就可以了。

对于库的维护者来说，问题不在于 Java 9 的支持是否必要，而在于何时提供支持。相比于迁移应用程序，将库迁移到 Java 9 和模块会引发完全不同的问题，在第 10 章中将会解决这些问题。

但首先，如果没有为应用程序采用模块，那么将应用程序迁移到 Java 9 上需要做些什么呢？应该清楚的是，为 Java 8 或更早版本所开发的应用程序必须遵循最佳实践（仅使用公共 JDK API）才能正常工作。虽然 JDK 9 仍然是向后兼容的，但是已经做了许多内部改变。可能遇到的迁移问题通常是由 JDK 的错误使用造成的，而这些错误使用可能是应用程序代码本身的问题，或者更可能是库的问题。



当谈到迁移问题时，库可能是迁移失败的主要根源。许多框架和库对 JDK 的（非公共的，因此也是不支持的）实现细节进行了假设。从技术上讲，不能责怪 JDK 破坏了代码。事实上，事情更加微妙。7.2 节介绍了在不破坏现有库的情况下实现强封装的折中办法。

在一个理想的世界中，在 Java 9 发布之前库和框架就将它们的实现更新为与 Java 9 兼容。但不幸的是，这不是我们生活的世界。作为库和框架的用户，应该知道如何解决潜在的问题。本章的其余部分将重点介绍即使在非理想的世界中，应该采取哪些策略使应用程序在 Java 9 上运行。希望随着时间的流逝，这一章会变得过时。

7.1 类路径已经“死”了？

前面的章节介绍了模块路径。在许多方面，可以将模块路径视为类路径的继任者。这是否意味着类路径在 Java 9 中不存在了？或者说它消失了吗？绝对不是！历史将会告诉我们类路径是否应该从 Java 中移除。与此同时，类路径在 Java 9 中仍然可用，并且工作方式与以前的版本大致相同。在下一章将会看到，类路径甚至可以和新的模块路径结合使用。

如果忽略模块路径，而使用类路径来构建和运行应用程序，那么我们根本就没有在应用程序中使用新的模块功能。此时需要对现有代码进行最少的（如果有的话）更改。简单地说，当应用程序及其依赖项仅使用来自 JDK 的官方认可的 API 时，它应该可以在 JDK 9 上编译并运行而不会出现任何问题。

如果说更改是必须的，那是因为 JDK 本身已经实现了模块化。无论应用程序是否使用了模块，从 Java 9 开始，它运行的 JDK 始终由模块组成。虽然此时从应用程序的角度来看，可以忽略模块系统，但对 JDK 结构的更改仍然存在。在大多数情况下，模块化 JDK 不会给基于类路径的应用程序带来任何问题，但肯定会有一些警告。这些警告在大多数情况下与库有关。本章的其余部分将介绍可能存在的问题，更重要的是这些问题的解决方法。

7.2 库、强封装和 JDK 9 类路径

将基于类路径的应用程序迁移到 Java 9 时可能遇到的一个问题是平台模块中代码的强大封装所引起的。许多库使用了平台中用 Java 9 封装的类，或者使用深度反射来窥探平台类的非公共部分。

深度反射使用反射 API 来访问类的非公共元素。在 6.1.1 节中曾经讲过，从模块中导出包不会使其非公共元素可用于反射。但不幸的是，许多库调用了通过反射找到的私有元



素上的 `setAccessible`。

可以看到，当使用模块时，默认情况下 JDK 9 不允许访问封装的包以及深度反射其他模块（包括平台模块）中的代码。这样做的一个很好的理由是：平台内部类的滥用已经成为许多安全问题的源头，并且阻碍了 API 的发展。但是，在本章中仍然需要在模块化 JDK 之上处理基于类路径的应用程序。在类路径上，平台内部类的强封装并不是严格执行的，尽管它仍然起作用。

对 JDK 类型使用深度反射有时是非常让人费解的。为什么要使 JDK 类的私有部分可访问呢？事实证明，一些常用的库都是这么做的。`javassist` 运行时代码生成库就是一个例子，其他许多框架都使用它。

为了便于将基于类路径的应用程序迁移到 Java 9，在对平台模块中的类应用深度反射时，或者使用反射来访问非导出包中的类型时，JVM 默认显示警告。例如，当运行使用 `javassist` 库的代码时，会看到以下警告：

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by javassist.util.proxy.SecurityActions
  (...javassist-3.20.0-GA.jar) to method
  java.lang.ClassLoader.defineClass(...)
WARNING: Please consider reporting this to the maintainers of
  javassist.util.proxy.SecurityActions
WARNING: Use --illegal-access=warn to enable warnings of further illegal
  reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

接下来仔细思考一下。那些在 JDK 8 和更早的版本上运行没有任何问题的代码现在会在控制台上显示一个醒目的警告——即使是在生产环境中也是如此。这表明严重破坏了强封装。

除了这个警告之外，应用程序仍然照常运行。如警告消息所示，在下一个 Java 版本中行为将发生变化。将来，即使是类路径上的代码，JDK 也会强制执行平台模块的强封装。在未来的 Java 版本中，相同的应用程序将不能在默认设置下运行。因此，好好研究一下警告信息并解决潜在的问题是非常重要的。如果警告是由库引起的，那么通常意味着向维护者报告了相关问题。

在默认情况下，只会在第一次非法访问尝试时产生一个警告，而后续的尝试将不会产生额外的错误或警告。如果想要进一步调查问题的原因，可以使用 `--illegal-access` 命令行标志的不同设置来调整行为：

```
--illegal-access=permit
```

默认行为。允许对封装类型进行非法访问。当第一次尝试通过反射进行非法访问时会生



成一个警告。

```
--illegal-access=warn
```

与 permit 一样，但每次非法访问尝试时都会产生错误。

```
--illegal-access=debug
```

同时显示非法访问尝试的堆栈跟踪。

```
--illegal-access=deny
```

不允许非法的访问尝试。这将是未来的默认行为。

请注意，没有允许取消打印的警告的设置。这是由设计所决定的。在本章中，将学习如何解决潜在的问题，以消除非法访问警告。由于 `--illegal-access=deny` 将是未来的默认设置，因此下一步目标是使用此设置运行应用程序。

如果使用 `--illegal-access=deny` 运行使用 `javassist` 的代码，那么应用程序将无法运行，并且会看到以下错误：

```
java.lang.reflect.InaccessibleObjectException: Unable to make
protected final
    java.lang.Class java.lang.ClassLoader.defineClass(java.lang.
String,byte[],
                                                int,int,java.security.
ProtectionDomain)
    throws java.lang.ClassFormatError accessible: module java.base
does not
    "opens java.lang" to unnamed module @0x7b3300e5
```

这个错误解释了 `javassist` 试图调用 `java.lang.Class` 上的 `defineClass` 方法。可以使用 `--add-opens` 标志授予对模块中特定包的类路径深度反射访问。“深度反射”一节中已经详细地讨论了开放式模块和开放式包。现在复习一下相关内容，为了进行深度反射，需要开放包。当导出包时也是如此，就像此时所使用的 `java.lang` 一样。通常在模块描述符中开放一个包，这与导出包的方式类似。可以通过命令行对那些无法控制的模块（例如平台模块）执行相同的操作：

```
java --add-opens java.base/java.lang=ALL-UNNAMED
```

在本示例中，`java.base/java.lang` 是授权访问的模块 / 包。最后一个参数是获取访问权限的模块。因为代码仍然在类路径中，所以使用了 `ALL-UNNAMED`，它表示类路径。现在，这个包是开放的，所以深度反射不再是非法的，从而消除了警告（或者避免使用 `--illegal-access = deny` 运行时出错）。同样，当类路径上的代码尝试访问非导出包中的类型时，可以使用 `--add-exports` 来强制导出包。在下一节中将会看到这种情况的一个例子。请记住，这仅仅是一种解决方法。可以询问一个库的维护者，即



即使是使用正确修复的已更新库版本，也会导致非法访问问题。



默认设置 `--illegalaccess=permit` 允许对 JDK 9 之前已经存在但现在被封装的包进行非法访问。即使代码位于类路径中，JDK 9 中的任何新的封装包都会被强封装。

安全影响

`--add-opens` 和 `--add-exports` 如何影响安全性？在未来的 Java 版本中，默认情况下不允许对平台模块进行深度反射的原因之一是为了防止恶意代码到达危险的 JDK 内部。是否可以使用一个标志来简单地禁用这些检查，同时继续保持这种安全优势呢？一方面，答案是肯定的，但是当选择这么做时，无形间就打开了一个更大的攻击面。

但考虑到这点：仅仅通过执行 Java 代码，无法在运行时获得由 `--add-opens` 或 `--add-exports` 提供的权限。攻击者需要访问应用程序的启动脚本（命令行）来添加这些标志。当建立了这种访问级别时，攻击者已经可以进行任意修改了，而不仅仅是添加 JVM 选项了。

7.3 编译和封装的 API

JDK 包含许多私有的内部 API，这些 API 应该仅能被 JDK 所使用。从早期开始，这一规定就已被清楚地记载下来了。比如 `sun.*` 和 `jdk.internal.*` 包。作为应用程序开发人员，可能并不会直接使用这些类型。大多数的内部类仅用于一些极端情况，典型的应用程序通常不需要。在编写本书的时候，甚至很难从应用程序开发的角度提供一个恰当的示例。

当然，一些应用程序以及（较旧的）库仍然使用这些内部类。在以前，JDK 内部没有进行强封装，因为当时没有这样的机制。当使用内部类时，Java 9 之前的编译器会发出警告，但那些警告很容易被忽略。前面已经看到，随着时间的推移，因为 `--illegal-access=permit` 默认设置，使用了封装 JDK 类型且在旧版本 Java 上编译的代码仍然可以在 Java 9 上运行。

然而，相同的代码在 Java 9 上却无法通过编译！假设使用 JDK 8 编译器编译示例 7-1 所示的代码，该代码使用了 `sun.security.x509` 包中的类型。

示例 7-1: `EncapsulatedTypes.java` ([↪ chapter7/encapsulation](#))

```
package encapsulated;
```




```
import sun.security.x509.X500Name;

public class EncapsulatedTypes {

    public static void main(String... args) throws Exception {
        System.out.println(new X500Name("test.com", "test",
            "test", "US"));
    }
}
```

使用 JDK 9 编译上面的代码，会产生如下所示编译器错误：

```
./src/encapsulated/EncapsulatedTypes.java:3: error: package sun.
security.x509
is not visible
import sun.security.x509.X500Name;
      ^
(package sun.security.x509 is declared in module java.base, which
does not
export it to the unnamed module)
```

默认情况下，尽管代码使用封装包，但代码仍然可以在 Java 9 上成功运行。你可能想知道为什么在访问封装类型时，javac 和 java 之间存在区别。当无法编译相同的代码时，能够运行访问封装类型的代码又有什么意义呢？

这样的代码之所以仍然能够运行，其原因是为现有的库提供向后兼容性。而禁止相同封装类型编译的原因是为了防止将来可能出现的兼容性噩梦。对于自己可以控制的代码，当涉及封装类型时，应立即采取行动，并用非封装的替代类型替换它们。当所使用的库（用较旧的 Java 版本进行编译）使用了封装类型或对 JDK 内部进行深度反射时，情况就更加复杂了。此时无法自己修复这个问题，这样一来，会阻止你向 Java 9 的迁移。但由于较宽松的运行时，因此库仍然可以暂时使用。

允许在运行时使用封装的 JDK 类型只是一种临时情况。在未来的 Java 版本中，这种情况将被禁止。通过设置上一节中所介绍的 `--illegal-access=deny` 标志可以防止这类情况的出现。使用 `java--illegal-access=deny` 运行相同的代码会生成一个错误：

```
Exception in thread "main" java.lang.IllegalAccessError:
class encapsulated.EncapsulatedTypes (in unnamed module @0x2e5c649) cannot
access class sun.security.x509.X500Name (in module java.base) because module
java.base does not export sun.security.x509 to unnamed module @0x2e5c649
    at encapsulated.EncapsulatedTypes.main(EncapsulatedTypes.java:7)
```



请注意，如果使用 `deny` 之外的任何其他值配置 `--illegal-access`，则不会显示任何警告。只有反射性的非法访问会触发前面所看到的警告，而静态引用封装类型则不会。该限制非常实用：如果将 VM 更改为对封装类型的静态引用也会产生警告，那么这种更改就显得过于侵入性了。



正确的做法是将问题报告给库的维护者。但是，如果是自己的代码，同时需要使用 JDK 9 重新编译，但又不能立即更改代码，应该怎么做呢？更改代码总是有风险的，所以必须找到更改的合适时机。

也可以使用命令行标志在编译时打破封装。在上一节中，看到了如何使用 `--add-opens` 从命令行公开一个包。java 和 javac 都支持 `--add-exports`，顾名思义，可以使用它从模块导出一个封装的包。语法是 `--add-exports <module>/<package> = <targetmodule>`。因为代码仍然在类路径上运行，所以可以使用 `ALL-UNNAMED` 作为目标模块。请注意，导出封装的包并不意味着可以对其类型进行深度反射。要进行深度反射，必须开放包。目前，导出包就足够了。在示例 7-1 中，直接引用了封装类型，而没有涉及任何反射。而对于 `sun.security.x509.X500Name` 示例，可以使用下面的命令编译并运行：

```
javac --add-exports java.base/sun.security.x509=ALL-UNNAMED \
      encapsulated/EncapsulatedTypes.java

java --add-exports java.base/sun.security.x509=ALL-UNNAMED
     \ encapsulated.EncapsulatedTypes
```

不仅对 JDK 内部，`--add-exports` 和 `--add-opens` 标志可用于任何模块和包。在编译过程中，内部 API 的使用仍然会发出警告。理想情况下，`--add-exports` 标志是临时迁移步骤。可以一直使用它，直到将代码调整为公共 API，或（如果库违反规则）直到使用替代 API 的第三方库的新版本发布。

太多命令行标志

一些操作系统限制了可执行命令行的长度。当在迁移期间需要添加许多标志时，可能会触犯这些限制。此时的替代做法是使用一个文件将所有的命令行参数提供给 java/javac：

```
$ java @arguments.txt
```

参数文件必须包含所有必要的命令行标志。文件中的每一行都包含一个选项。例如，`arguments.txt` 可以包含以下内容：

```
-cp application.jar:javassist.jar
--add-opens java.base/java.lang=ALL-UNNAMED
--add-exports java.base/sun.security.x509=ALL-UNNAMED
-jar application.jar
```

即使没有命令行限制，相比于脚本中非常长的命令行，参数文件可能更清晰。



7.4 删除的类型

代码也可以使用目前已经完全删除的内部类型，虽然这与模块系统没有直接关系，但仍值得一提。其中一个在 Java 9 中删除的内部类是 `sun.misc.BASE64Encoder`，在 Java 8 引入 `java.util.Base64` 类之前，该类是非常流行的。示例 7-2 显示了使用 `BASE64Decoder` 的代码：

示例 7-2: `RemovedTypes.java` (↪ [chapter7/removedtypes](#))

```
package removed;

import sun.misc.BASE64Decoder;

// Compile with Java 8, run on Java 9: NoClassDefFoundError.
public class RemovedTypes {
    public static void main(String... args) throws Exception {
        new BASE64Decoder();
    }
}
```

该代码在 Java 9 上无法编译或运行。当尝试编译时，会看到以下所示错误：

```
removed/RemovedTypes.java:3: error: cannot find symbol
import sun.misc.BASE64Decoder;
                ^
    symbol:   class BASE64Decoder
    location: package sun.misc
removed/RemovedTypes.java:8: error: cannot find symbol
    new BASE64Decoder();
        ^
    symbol:   class BASE64Decoder
    location: class RemovedTypes
2 errors
```

如果使用旧版本的 Java 编译上面的代码，但试图在 Java 9 上运行，也会失败：

```
Exception in thread "main" java.lang.NoClassDefFoundError: sun/misc/
BASE64Decoder
    at removed.RemovedTypes.main(RemovedTypes.java:8)
Caused by: java.lang.ClassNotFoundException: sun.misc.BASE64Decoder
...
```

对于封装类型，可以通过使用命令行标志进行强制访问来解决此问题。但此时针对 `Base64Decoder` 示例无法这么做，因为该类不存在。理解这种差异是很重要的。

使用 `jdeps` 查找已删除或封装的类型及其替代方法

`jdeps` 是 JDK 附带的一个工具。`jdeps` 可以做的一件事是找到使用被删除或封装的 JDK 类型的地方，并建议替换。`jdeps` 总是在类文件上工作，而不是在源代码上。如果使用 Java 8 编译示例 7-2，则可以在生成的类上运行 `jdeps`：



```
jdeps -jdkinternals removed/RemovedTypes.class
```

```
RemovedTypes.class -> JDK removed internal API
removed.RemovedTypes -> sun.misc.BASE64Decoder
JDK internal API (JDK removed internal API)
```

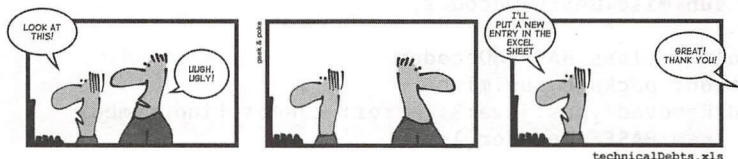
Warning: JDK internal APIs are unsupported and private to JDK implementation that are subject to be removed or changed incompatibly and could break your application.

Please modify your code to eliminate dependence on any JDK internal APIs. For the most recent update on JDK internal API replacements, please check: <https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool>

JDK Internal API	Suggested Replacement
sun.misc.BASE64Decoder	Use java.util.Base64 @since 1.8

类似地，示例 7-1 中诸如 X500Name 之类的封装类型也被 jdeps 报告并带有建议的替换方法。有关如何使用 jdeps 的更多详细信息，请参阅 8.9 节。

从 Java 8 开始，JDK 就包含了 `java.util.Base64`，这是一个更好的选择。在这种情况下，解决方案就很简单：必须迁移到公共 API 以便在 JDK 9 上运行。一般来说，迁移到 Java 9 将会暴露本章讨论的许多技术债务 (technical debt)。



7.5 使用 JAXB 和其他 Java EE API

在过去，JDK 附带的某些 Java EE 技术 (如 JAXB) 是与 Java SE API 一起提供的。这些技术在 Java 9 中仍然存在，但是需要特别注意。它们主要包含在以下模块列表中：

- `java.activation`
- `java.corba`
- `java.transaction`
- `java.xml.bind`
- `java.xml.ws`
- `java.xml.ws.annotation`

在 Java 9 中，这些模块已被弃用。`@Deprecated` 注释在 Java 9 中有一个新的参数 `forRemoval`。如果将其设置为 `true`，则意味着 API 元素将在未来版本中被删除。对



于属于 JDK 的 API 元素来说，这意味着在下一个主要版本中可能会被删除。有关弃用的更多细节可以在 JEP 277 (<http://openjdk.java.net/jeps/277>) 中找到。

从 JDK 中删除 Java EE 技术有很好的理由。JDK 中 Java SE 和 Java EE 之间的重叠一直是令人困惑的。Java EE 应用程序服务器通常提供了 API 的自定义实现。简单地讲，是通过将替代实现放在类路径上，覆盖默认的 JDK 版本来完成的。而在 Java 9 中，这就成为一个问题。模块系统不允许多个模块提供相同的包。如果在类路径中找到重复的包（因此在未命名的模块中），则会将其忽略。在任何情况下，若 Java SE 和应用程序服务器都提供 `java.xml.bind`，则不会实现预期的行为。

这是一个严重的实际问题，会破坏许多现有的应用服务器和相关工具。为了避免出现该问题，在基于类路径的场景中，默认情况下不会解析这些模块。接下来看一下图 7-1 所示的平台模块图。

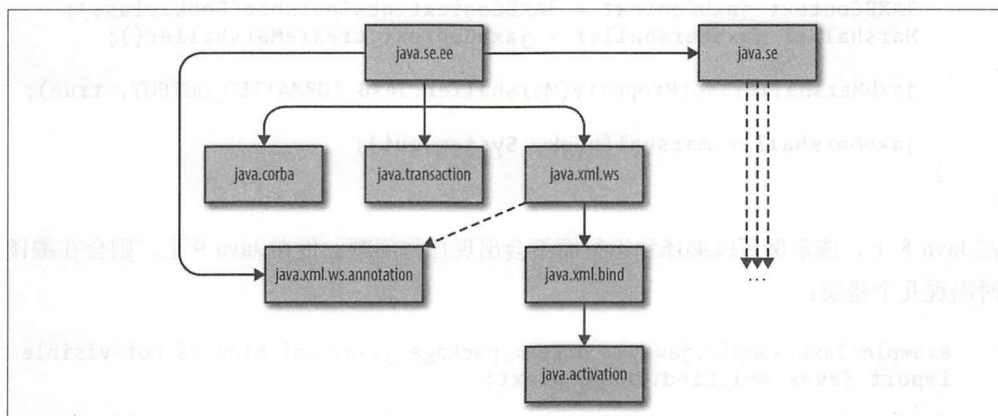


图 7-1：显示只能通过 `java.se.ee`（而不是 `java.se`）访问的模块的 JDK 模块图的子集

最顶层是 `java.se` 和 `java.se.ee` 模块。两者都是聚合器模块，这些模块不包含代码，而是将一组更细粒度的模块组合在一起。聚合器模块已经在 5.4 节中详细讨论过。大多数平台模块驻留在 `java.se` 中，图中并没有显示（但可以从图 2-1 中看到整个模块图）。`java.se.ee` 模块聚合了正在讨论的模块，它们不是 `java.se` 聚合器模块的一部分，其中包括包含了 JAXB 类型的 `java.xml.bind` 模块。

默认情况下，在编译和运行未命名模块中的类时，`javac` 和 `java` 都使用 `java.se` 作为根，代码可以访问由 `java.se` 的可传递依赖项导出的任何包。因此，在 `java.se.ee` 中却不在 `java.se` 下的模块不会被解析，所以它们也不会被未命名的模块读取。即使从模块 `java.xml.bind` 中导出包 `javax.xml.bind` 也没关系，因为在编译和运行时不会对其进行解析。



如果 `java.se.ee` 中的模块是必需的，那么就需要将它们显式地添加到已解析的平台模块集合中。可以使用 `javac` 和 `java` 的 `--add-modules` 标志将这些模块添加为根模块。

接下来看一下基于 JAXB 的示例 7-3。该示例将 `Book` 序列化为 XML。

示例 7-3: `JaxbExample.java` (↪ `chapter7/jaxb`)

```
package example;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class JaxbExample {
    public static void main(String... args) throws Exception {
        Book book = new Book();
        book.setTitle("Java 9 Modularity");

        JAXBContext jaxbContext = JAXBContext.newInstance(Book.class);
        Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

        jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

        jaxbMarshaller.marshal(book, System.out);
    }
}
```

在 Java 8 上，该示例可以编译和运行而不会出现任何问题。但在 Java 9 上，则会在编译时出现几个错误：

```
example/JaxbExample.java:3: error: package javax.xml.bind is not visible
import javax.xml.bind.JAXBContext;
                    ^
(package javax.xml.bind is declared in module java.xml.bind, which is not
 in the module graph)
example/JaxbExample.java:4: error: package javax.xml.bind is not visible
import javax.xml.bind.JAXBException;
                    ^
(package javax.xml.bind is declared in module java.xml.bind, which is not
 in the module graph)
example/JaxbExample.java:5: error: package javax.xml.bind is not visible
import javax.xml.bind.Marshaller;
                    ^
(package javax.xml.bind is declared in module java.xml.bind, which is not
 in the module graph)
3 errors
```

当使用 Java 8 进行编译并使用 Java 9 运行代码时，就会在运行时生成报告相同问题的异常。前面已经介绍了如何解决这个问题：将 `--add-modules java.xml.bind` 添加到 `javac` 和 `java` 调用中。



除了添加包含 JAXB 的平台模块之外，还可以添加一个为类路径提供 JAXB 的 JAR。几个流行的（开源）库提供了 JAXB 实现。由于 JDK 中的 Java EE 模块被标记为“删除”，因此这是一个更具前瞻性的解决方案。

请注意，如果使用模块路径，则不会遇到该问题。如果代码存在于一个模块中，那么它必须在 `java.base` 以外的任何模块上显式地定义一个需求。对于示例代码来说，包括对 `java.xml.bind` 的依赖。基于此原因，模块系统不需要命令行标志即可解析这些模块。

总结一下，当从 JDK 使用 Java EE 代码时要格外小心。当接收到包不可见的错误时，可以通过使用 `--add-modules` 添加相关模块。但是请注意，所添加的模块将在下一个主要的 Java 版本中被删除。可以将自己的这些技术版本添加到类路径中，以避免将来出现问题。

7.6 jdk.unsupported 模块

JDK 的一些内部类已经被证明更难以进行封装。一般来说，使用 `sun.misc.Unsafe` 等类的机会是比较少的。这些一直是不受支持的类，意味着只能在 JDK 内部使用。

由于性能原因，这些类中的一些被许多库广泛使用。显而易见这种做法是不对的，但在某些情况下这是唯一的选择。一个著名的示例是 `sun.misc.Unsafe` 类，它可以绕过 Java 的内存模型和其他安全网执行一些低级操作，而 JDK 之外的库无法实现相同的功能。

如果简单地封装这些类，那么依赖于它们的库将不再使用 JDK 9，至少没有警告。从理论上讲，这不是一个向后兼容的问题，毕竟这些库滥用了不支持的实现类。对于这些使用频率比较高的内部 API 来说，由于其在现实世界中的作用太大而无法忽视，特别是当它们所提供的功能没有替代方案时更是如此。

考虑到这一点，达成了妥协。JDK 团队研究了哪些 JDK 平台内部类是库使用最多的，哪些只能在 JDK 内部实现。这些类没有封装在 Java 9 中。

下面所示的是可以访问的特定类和方法的列表：

- `sun.misc.{Signal,SignalHandler}`
- `sun.misc.Unsafe`
- `sun.reflect.Reflection::getCallerClass(int)`
- `sun.reflect.ReflectionFactory::newConstructorForSerialization`



请记住，如果这些名字对你来说没有任何意义，那是一件好事。诸如 Netty、Mockito 和 Akka 之类的流行库都使用了这些类。不破坏这些库也是一件好事。

由于这些方法和类主要不是为在 JDK 之外使用而设计的，因此可以将它们移动到名为 `jdk.unsupported` 的平台模块中。这表明在未来的 Java 版本中，该模块中的类将被其他 API 所替换。`jdk.unsupported` 模块导出和 / 或公开了包含所讨论类的内部包。现有的用途包括深度反射。与 7.2 节中所讨论的场景不同，通过反射使用这些类不会导致警告。这是因为 `jdk.unsupported` 在其模块描述符中公开了必需的包，所以从这个角度来看不存在非法访问。



虽然可以在不破坏封装的情况下使用这些类型，但它们仍然不受支持，依旧不鼓励使用这些类型。预计在将来会提供支持的替代类型。例如，`Unsafe` 中一些功能被 JEP 193 (<http://openjdk.java.net/jeps/193>) 中所提出的变量句柄 (`variable handle`) 所取代。但在此之前，仍然维持现状。

当代码仍然存在于类路径上时，不需要进行任何更改。库可以像以前一样通过类路径使用这些类，并且在没有任何警告或错误的情况下运行。编译器针对 `jdk.unsupported` 中的类进行编译时会生成警告，而不是像编译封装类型一样产生错误：

```
warning: Unsafe is internal proprietary API and may be
         removed in a future release
```

如果想从模块中使用这些类型，则必须依靠 `jdk.unsupported`。在模块描述符中有这样一个 `requires` 语句就相当于是一个警告标志。在未来的 Java 版本中，其可能需要进行更改以适应公开支持的 API，而不是不支持的 API。

7.7 其他更改

JDK 9 中的许多其他更改可能会破坏代码。比如，这些更改会影响工具作者以及使用 JDK 扩展机制的应用程序。其中一些变化如下：

- **JDK 布局**

由于平台模块化，包含所有平台类的大 `rt.jar` 不再存在。正如 JEP 220 (<http://openjdk.java.net/jeps/220>) 中所记载的那样，JDK 本身的布局也发生了相当大的变化。依靠 JDK 布局的工具或代码必须适应这个新的现实。

- **版本字符串**

所有 Java 平台版本都以 1.x 为前缀开头的日子已经一去不复返了。Java 9 对应的是版本



9.0.0。版本字符串的语法和语义已经发生了很大的变化。如果应用程序对 Java 版本进行任何类型的解析，请阅读 JEP 223 (<http://openjdk.java.net/jeps/223>) 以了解所有的细节。

- 扩展机制

诸如授权标准覆盖机制 (*Endorsed Standard Override Mechanism*) 以及通过 `java.ext.dirs` 属性的扩展机制等功能已被删除。它们被可升级的模块所取代。更多的信息可以在 JEP 220 (<http://openjdk.java.net/jeps/220>) 中找到。

这些都是 JDK 的高度专业化功能。如果应用程序确实需要它们，那么它就不能使用 JDK 9。因为这些更改与 Java 模块系统没有真正的关系，所以在此不做详细讨论。链接相关的 JEP (JDK Enhancement Proposal) 包含有关如何在这些情况下继续进行操作的指导。

恭喜！现在你已经知道如何在 JDK 9 上运行现有的应用程序了。虽然有些事情可能出错，但在大多数情况下，还是比较顺利的。请记住，用 `--illegal-access = deny` 来运行你的应用程序，以便为将来做好准备。在解决了从类路径中运行现有的应用程序所存在的问题之后，接下来看一下如何使现有的应用程序更加模块化。





第 8 章

迁移到模块

在了解了前几章所介绍的模块优点之后，也许你非常希望能够开始使用 Java 模块系统。只要理解了基本概念，编写基于模块的新代码还是比较简单的。

在现实世界中，存在许多需要迁移到模块中的现有代码。上一章介绍了如何将现有的代码迁移到 Java 9，而无须将代码转换为模块。这只是任何迁移方案中的第一步。掌握了相关内容之后，接下来可以把注意力放在如何迁移到 Java 模块系统上了。



在此，并不建议迁移每个现有的应用程序以使用 Java 模块系统。如果一个应用程序不再被积极开发，那么这么做是不值得的。此外，小型应用程序可能不会真正从模块结构中受益。在合理的情况下进行迁移，以提高可维护性、可变性和可重用性——而不仅仅是为了迁移而迁移。

移植所需的工作量在很大程度上取决于代码库的结构如何，但即使对于结构良好的代码库，迁移到模块化运行时也是一项艰巨的任务。大多数应用程序都使用了第三方库，这是迁移时需要考虑的一个重要因素。这些库不一定是模块化的，当然你可能也不想承担这个责任。

幸运的是，Java 模块系统将向后兼容性和可移植性作为主要的关注点。在 Java 模块系统中引入了几个结构，从而可以对现有代码进行逐步迁移。在本章中，将了解这些构造，从而将自己的代码迁移到模块。从库维护者的角度来看，迁移当然是需要的，但是需要一个略微不同的过程（第 10 章将着重介绍该内容）。

8.1 迁移策略

一个典型的应用程序包含应用程序代码以及库代码，而应用程序代码可能会使用来自第



三方库的代码。在理想情况下，所有使用的库都已经是模块，此时只需专注于模块化自己的代码即可。在 Java 9 发布后的头几年，现实情况可能不是这样的，一些库还不能作为模块来使用，也许永远不能，因为它们不再被维护。

如果一直等待整个生态系统向模块方向发展，那么可能要等很长时间。此外，还需要更新到这些库的新版本，从而可能引起一系列问题。也可以手动修补库，添加一个模块描述符并将它们转换为一个模块。显然，这需要完成大量工作，并且需要对库进行拆分 (fork)，这使得未来的更新变得更加痛苦。如果能够专注于迁移自己的代码，而暂时保留库现有的工作方式，那就最好了。

8.2 一个简单示例

本章将介绍几个迁移示例，以了解实践中可能遇到的各种情况。首先看一个简单的应用程序，其使用 Jackson 库将 Java 对象转换为 JSON。对于这个应用程序，需要 Jackson 项目中的三个 JAR 文件：

- `com.fasterxml.jackson.core`
- `com.fasterxml.jackson.databind`
- `com.fasterxml.jackson.annotations`

本例所使用的版本 (2.8.8) 中的 Jackson JAR 文件并不是模块。它们是普通 JAR 文件，没有模块描述符。

应用程序由两个类组成，其中示例 8-1 列出了主类。这里没有列出的 `Book` 类是一个表示书的简单类，包含 `getter` 和 `setter`。`Main` 类包含一个使用 `com.fasterxml.jackson.databind` 的 `ObjectMapper` 将一个 `Book` 实例转换为 JSON 的 `main` 方法。

示例 8-1: `Main.java` ([↩ chapter8/jackson-classpath](#))

```
package demo;

import com.fasterxml.jackson.databind.ObjectMapper;

public class Main {

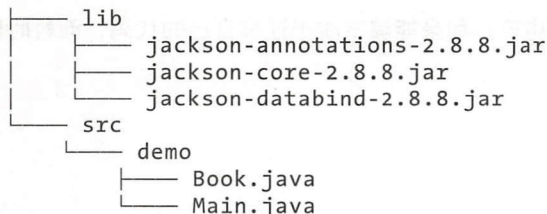
    public static void main(String... args) throws Exception {
        Book modularityBook =
            new Book("Java 9 Modularity", "Modularize all the things!");

        ObjectMapper mapper = new ObjectMapper();
        String json = mapper.writeValueAsString(modularityBook);
        System.out.println(json);
    }
}
```



```
}
}
```

示例中的 `com.fasterxml.jackson.databind.ObjectMapper` 类是 `jackson-databind-2.8.8.jar` 的一部分。这个 JAR 文件依赖于 `jackson-core-2.8.8.jar` 和 `jackson-annotations-2.8.8.jar`。但是，这种依赖关系是隐式的，因为 JAR 文件不是模块。示例项目开始时具有以下文件结构：



正如在上一章中看到的那样，类路径在 Java 9 中仍然可用。在开始迁移到模块之前，首先在类路径上构建和运行。可以使用示例 8-2 中的命令来构建和运行应用程序。

示例 8-2: `run.sh` (`↪ chapter8/jackson-classpath`)

```
CP=lib/jackson-annotations-2.8.8.jar:
CP+=lib/jackson-core-2.8.8.jar:
CP+=lib/jackson-databind-2.8.8.jar
```

```
javac -cp $CP -d out -sourcepath src $(find src -name '*.java')
```

```
java -cp $CP:out demo.Main
```

该应用程序可以使用 Java 9 编译和运行，而无须进行任何更改。

虽然 Jackson 库并不在我们的直接控制之下，但我们却可以控制 `Main` 和 `Book` 的代码，所以这些代码是迁移时所需要注意的。这是一个非常常见的迁移情况，将自己的代码转移到模块上，而不用担心库。Java 模块系统提供了一些技巧来实现逐步迁移。

8.3 混合类路径和模块路径

为了使逐步迁移成为可能，可以混合使用类路径和模块路径。但这不是一种理想的情况，因为只能部分受益于 Java 模块系统的优点。但是，“小步”迁移是非常有帮助的。

因为 Jackson 库不是我们自己的源代码，所以理想情况下根本不会更改它们。相反，首先通过迁移自己的代码开始自上而下的迁移。接下来将代码放到一个名为 `books` 的模块中。虽然这样做是远远不够的，但却可以通过为模块创建一个简单的 `module-info.java` 开始迁移之旅：



```
module books {
}
```

请注意，该模块还没有任何 `requires` 语句。这非常可疑，因为显而易见其需要依赖于 `jackson-databind-2.8.8.jar` 文件中的类。因为已经有了一个真正的模块，所以可以使用 `--module-source-path` 标志来编译代码。Jackson 库不是模块，所以暂时留在类路径上：

```
CP=lib/jackson-annotations-2.8.8.jar:
CP+=lib/jackson-core-2.8.8.jar:
CP+=lib/jackson-databind-2.8.8.jar

javac -cp $CP -d out --module-source-path src -m book
ssrc/books/demo/Main.java:3: error:
package com.fasterxml.jackson.databind does not exist

import com.fasterxml.jackson.databind.ObjectMapper;
                                ^
src/books/demo/Main.java:11: error: cannot find symbol
    ObjectMapper mapper = new ObjectMapper();
    ^
symbol:   class ObjectMapper
location: class Main
src/books/demo/Main.java:11: error: cannot find symbol
    ObjectMapper mapper = new ObjectMapper();
    ^
symbol:   class ObjectMapper
location: class Main
3 errors
```

此时，编译器显然不开心！尽管 `jackson-databind-2.8.8.jar` 仍然在类路径中，但编译器却告知无法在模块中使用。模块不能读取类路径，所以模块不能访问类路径上的类型，如图 8-1 所示。

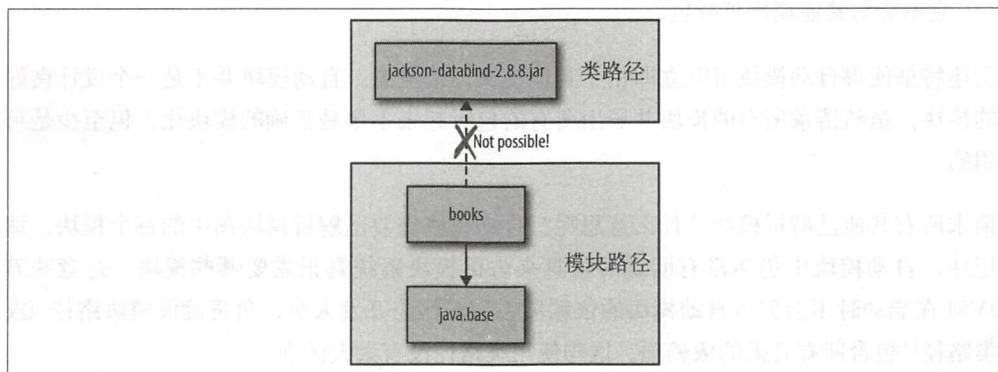


图 8-1：模块不能读取类路径



即使在迁移过程中需要做一些工作，而无法从类路径中读取却是一件好事，因为需要明确依赖关系。如果模块可以读取类路径，除了便于获取显式依赖关系之外，其他的优势就不复存在了。

即便如此，应用程序现在还无法编译，所以先试着解决这个问题。如果不能依赖类路径，那么唯一的方法就是将模块依赖的代码作为模块来使用。这就要求将 *jackson-databind-2.8.8.jar* 变成一个模块。

8.4 自动模块

Jackson 库的源代码是开源的，所以可以修补代码，将其变为一个模块。在使用长列表（可传递）依赖关系的大型应用程序中，并不需要修补所有的依赖关系。此外，我们可能还没有掌握正确模块化库所需的知识。

Java 模块系统提供了一个有用的功能来处理非模块的代码：*自动模块*。只需将现有的 JAR 文件从类路径移动到模块路径，而不改变其内容，就可以创建一个自动模块。这样一来，JAR 就转换为一个模块，同时模块系统动态生成模块描述符。相比之下，*显式模块*始终有一个用户自定义的模块描述符。到目前为止，所看到的所有模块（包括平台模块）都是显式模块。自动模块的行为不同于显式模块。自动模块具有以下特征：

- 不包含 *module-info.class*。
- 它有一个在 *META-INF/MANIFEST.MF* 中指定或者来自其文件名的模块名称。
- 通过 `requires transitive` 请求所有其他已解析模块。
- 导出所有包。
- 读取路径（或者更准确地讲，读取前面所讨论的未命名模块）。
- 它不能与其他模块拆分包。

上述特征使得自动模块可以立即用于其他模块。请注意，自动模块并不是一个设计良好的模块。虽然请求所有的模块并导出所有的包听起来不像是正确的模块化，但至少是可用的。

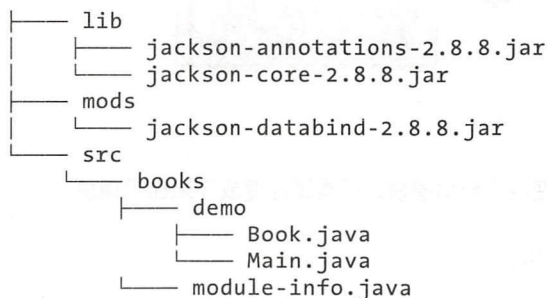
请求所有其他已解析模块是什么意思呢？自动模块需要已解析模块图中的每个模块。请记住，自动模块中仍然没有明确的信息来告诉模块系统真正需要哪些模块，这意味着 JVM 在启动时不会警告自动模块的依赖项丢失。作为开发人员，负责确保模块路径（或类路径）包含所有必需的依赖项。这与使用类路径没有太大区别。

模块图中的所有模块都需要通过自动模块传递。这实际上意味着，如果请求一个自动模



块，那么就可以“免费”获得所有其他模块的隐式可读性。这是一种权衡，将在稍后详细讨论。

请将 *jackson-databind-2.8.8.jar* 文件移动到模块路径，从而将其转换成一个自动模块。首先把 JAR 文件移动到一个新的目录，在本示例中将其命名为 *mods*：



接下来必须修改 *books* 模块中的 *module-info.java*，以便请求 *jackson.databind*：

```

module books {
    requires jackson.databind;
}

```

books 模块请求 *jackson.databind*，就好像它是一个正常的模块。但模块名称来自哪里呢？自动模块的名称可以在 *META-INF/MANIFEST.MF* 文件的新引入的 *Automatic-Module-Name* 字段中指定。这样一来，即使在将库完全迁移到模块之前，库维护人员也可以选择模块名称。有关以这种方式命名模块的更多详细信息，请参阅 10.2 节。

如果没有指定名称，则模块名称是从 JAR 的文件名派生的。命名算法大致如下：

- 使用点 (.) 替换破折号 (-)。
- 忽略版本号。

在 Jackson 示例中，模块名称是基于文件名称的。

现在，可以通过运行下面的命令成功地编译程序：

```

CP=lib/jackson-annotations-2.8.8.jar:
CP+=lib/jackson-core-2.8.8.jar

```

```

javac -cp $CP --module-path mods -d out --module-source-path src -m books

```

将 *jackson-databind-2.8.8.jar* 文件从类路径中移除，并配置了一个模块路径，指向 *mods* 目录。图 8-2 提供了所有代码的概述。

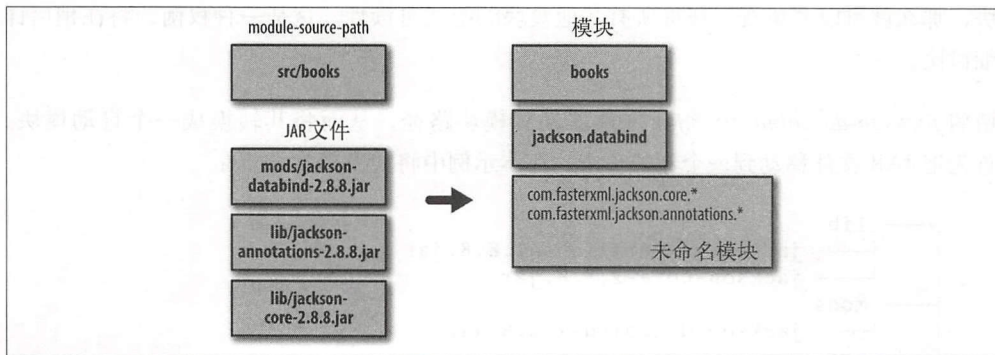


图 8-2：模块路径上的非模块化 JAR 变成了自动模块。而类路径变成了未命名模块

为了运行程序，还需要更新 Java 调用：

```
java -cp $CP --module-path mods:out -m books/demo.Main
```

对 Java 命令完成如下修改：

- 将 *out* 目录移至模块路径中。
- 将 *jackson-databind-2.8.8.jar* 文件从类路径 (*lib*) 移至模块路径 (*mods*) 中。
- 通过使用 *-m* 标志来指定模块，从而启动应用程序。



可以将所有 JAR 移动到模块路径，而不仅仅是移动 (*jackson-databind*)。虽然这样做使得移动过程变得容易一些，但是很难看到究竟发生了什么。当迁移自己的应用程序时，可以随意将所有 JAR 移动到模块路径。

现在已经很接近第一个迁移的应用程序了，但不幸的是在启动应用程序时仍然遇到了异常：

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
  Unable to make public java.lang.String demo.Book.getTitle() accessible:
  module books does not "exports demo" to module jackson.databind
  ...
```

虽然这是 Jackson Databind 所特有的问题，但并不罕见。此时使用 Jackson Databind 来编组 *Book* 类 (该类是 *books* 模块的一部分)。Jackson Databind 使用反射来查看类的字段以便进行序列化。因此，Jackson Databind 需要访问 *Book* 类；否则，就无法使用反射来查看它的字段。为此，包含该类的包必须通过其包含模块 (本例中的 *books* 模块) 被导出或开放。导出包限制了 Jackson Databind 只能反射公共元素，而开放包则还允许进行深度反射。在本示例中，反射公共元素就足够了。



此时陷入了一个困境。没有必要仅因为 Jackson 需要 Book 类而将包含 Book 的包导出到其他模块。如果这样做，就意味着放弃封装，而封装性是将现有应用程序转换为模块的主要原因之一！解决这个问题有多种方法，每种方法都有自己的权衡考虑。第一种方法是使用**限制导出**。通过使用限制导出，可以将包仅导出到 `jackson.databind`，同时又不会失去其他模块的封装：

```
module books {
    requires jackson.databind;
    exports demo to jackson.databind;
}
```

重新编译后，现在可以成功运行应用程序了！除了使用导出方法以外，还有另外一种方法可以更好地适应需求，该方法将在下一节中探讨。

使用自动模块时出现的警告

尽管自动模块对迁移至关重要，但应谨慎使用。每当在自动模块上写入一个 `requires` 时，请记住，稍后再回来。如果库作为一个显式模块发布，则应该使用该模块。

编译器中增加了两个警告，以帮助解决这个问题。请注意，Java 编译器支持这些警告只是一个建议，所以不同的编译器实现可能会有不同的结果。第一个警告是选择退出（默认情况下启用），并针对自动模块上的每个 `requires transitive` 发出警告。该警告可以通过 `-Xlint:-requires-transitive-automatic` 标志禁用。请注意冒号后的短划级（-）。第二个警告是选择进入（默认情况下禁用），并针对自动模块上的每个 `requires` 发出警告。该警告可以通过 `-Xlint:requires-automatic`（冒号后没有短划线）标志来启用。第一个警告之所以是默认启用的，原因在于这是一种更危险的情况。通过隐式可读性，会将一个（可能不稳定的）自动模块公开给模块的使用者。

当显式模块可用时，可以用显式模块替代自动模块，如果显式模块尚不可用，可以要求库维护者提供。另外请记住，这样的模块可能提供了受更多限制的 API，因为库维护者并不希望默认情况下导出所有包。当从自动模块切换到显式模块时会产生额外的工作，库维护者需要创建一个模块描述符。

8.5 开放式包

在反射的上下文中使用 `exports` 时有一些需要注意的地方。首先，需要将编译时可读性赋予一个包，这一点看上去非常奇怪，因为我们只期望运行时（反射）使用。虽然框架经常使用反射来处理应用程序代码，但是它们不需要编译时可读性。另外，不可能总



是事先知道哪个模块需要可读性，所以限制导出是不可能的。

使用 Java Persistence API (JPA) 就是这种情况的一个例子。当使用 JPA 时，通常编程为标准化的 API。而在运行时，会使用该 API 的实现，比如 Hibernate 或者 EclipseLink。API 和实现在不同的模块中。最终，实现需要访问类。如果在模块中将 `exports com.mypackage` 放到 `hibernate.core` 或类似的包中，就会连接到实现。如果想要更改 JPA 实现，就需要更改代码的模块描述符，而这恰恰是泄露实现细节的迹象。

如 6.1.2 节中所讨论的那样，当涉及反射时会出现另一个问题：导出包只导出了包中的公共类型，受保护的或者包专用的类以及导出类中的非公共方法和字段都是不可访问的。即使导出了包，深度反射（使用 `setAccessible` 方法）也不起作用。如果想要进行深度反射（许多框架需要该功能），一个包必须是开放的。

返回到 Jackson 示例，此时可以使用 `opens` 关键字，而不是对 `jsckson.databind` 进行限制导出：

```
module books {
    requires jackson.databind;

    opens demo;
}
```

一个开放式包允许任何模块对其类型进行运行时访问（包括深度反射），但编译时访问却是禁止的。这避免了其他人在编译时意外地使用了实现代码，而框架可以在运行时毫无问题地施展它们的“魔力”。当仅需要运行时访问时，在大多数情况下 `opens` 是一个不错的选择。请记住，一个开放式包并没有真正地封装，其他模块始终可以使用反射来访问这个包。但至少开发过程中受到了保护，避免了意外使用，并且清楚地表明这个软件包不能被其他模块直接使用。

与 `exports` 关键字一样，`opens` 关键字也可以被限制，从而向一组有限的模块开放包：

```
module books {
    requires jackson.databind;

    opens demo to jackson.databind;
}
```

现在已经看到了解决运行时可访问性问题的两种方法，但仍然存在一个问题：为什么只有在运行应用程序时才发现此问题，而不是在编译时呢？为了更好地理解这种情况，需要重新审视可读性规则。对于一个能够读取来自另一个模块中另一个类的类来说，需要满足以下条件：

- 该类必须是公共的（忽略深度反射的情况）。



- 另一个模块中的包必须被导出，或者在进行深度反射时是开放的。
- 消费模块与其他模块之间必须具有可读性关系 (`requires`)。

通常，可以在编译时对上述条件进行检查。然而，Jackson Databind 对所编写的代码并不存在编译时依赖。它之所以知道 `Book` 类，因为它将作为参数传入 `ObjectMapper`。这意味着编译器并不能帮助我们。在进行反射时，运行时会自动设置一个可读性关系 (`requires`)，所以该步骤要额外小心。接下来，它会发现该类在运行时没有导出或开放 (因此也不可访问)，并且不会由运行时自动“修复”。

既然运行时足够聪明，可以自动添加可读性关系，那么为什么不能开放包呢？这涉及意图和模块所有权的问题。当代码使用反射访问另一个模块中的代码时，从该模块的角度来看，其目的显然是读取其他模块，所以无须过多地明确这一点。但对于 `exports/opens` 来说却不是这样的。模块所有者应决定导出或开放哪些包。只有模块本身应该定义这个意图，所以该意图不能被其他模块的行为自动推断出来。

许多框架以类似的方式使用反射，所以在迁移之前进行测试通常是非常重要的。



在 7.2 节中已经讲过，在默认情况下，Java 9 使用 `--illegal-access = permit` 运行。那么为什么仍然为了进行反射而显式地开放包呢？请记住，`-illegal-access` 标志只影响类路径上的代码。在本示例中，`jackson.databind` 本身是一个模块，反映了我们所编写模块（不是平台模块）中的代码。而在涉及的类路径中没有代码。

8.6 开放式模块

在前面的章节中，学习了如何使用开放式包提供对包的运行时访问，这可以满足许多框架和库的反射需求。如果正在进行大规模的迁移，那么在一个尚未完全模块化的代码库中，哪些包需要开放可能并不是那么明显。理想情况下或许可以确切地知道所使用的框架和库是如何访问代码的，但我们可能对正在使用的代码库并不熟悉。这样一来，就会导致一个单调乏味的试错过程，试图找出哪些包需要开放。针对这些情况，可以使用开放式模块，这是一个不太精确但功能更强大的工具：

```
open module books {
    requires jackson.databind;
}
```

开放式模块提供了对其所有包的运行时访问，但并不会授予对包的编译时访问权限，而这正是迁移代码想要的。如果需要在编译时使用包，则必须将其导出。首先创建一个开



开放式模块以避免与反射有关的问题，这样做有助于首先关注需求 (requires) 和编译时使用 (exports)。一旦应用程序再次运行，可以通过从模块中删除 open 关键字来更好地调整对包的运行时访问权限，同时更具体地说明哪些包应该开放。

8.7 破坏封装的 VM 参数

在某些情况下，向模块添加 exports 或者 opens 并不是一个选项，可能是无法访问代码，或者只有在测试期间才需要访问。在这些情况下，可以使用 VM 参数来设置更多的导出。对于平台模块，已经在 7.3 节中看到了这一点，也可以对其他模块 (包括自己的模块) 执行相同的操作。

可以使用命令行标志来实现相同的功能，而不是向 books 模块描述符添加 exports 或 opens 子句：

```
--add-exports books/demo=jackson.databind
```

运行应用程序的完整命令如下所示：

```
java -cp lib/jackson-annotations-2.8.8.jar:lib/jackson-core-2.8.8.jar \  
  --module-path out:mods \  
  --add-exports books/demo=jackson.databind \  
  -m books/demo.Main
```

上述命令在启动 JVM 时设置了一个限制导出。可以使用一个类似的标志来打开包：--add-opens。虽然这些标志在特殊情况下是有用的，但它们应被视为最后的手段。如前一章所示，可以同样的机制访问内部的非导出包。虽然在代码正确迁移之前这可能是一个临时的解决方法，但应该非常谨慎地使用。不应该轻易地破坏封装。

8.8 自动模块和类路径

在前一章中已经看到过未命名模块，类路径上的所有代码都是未命名模块的一部分。在 Jackson 示例中已经讲过，正在编译的模块中的代码不能访问类路径上的代码。那么当 jackson.databind 自动模块所依赖的 Jackson Core 和 Jackson Annotations JAR 仍然在类路径上时该模块是如何正常工作的？该模块能正常工作是因为这些库位于未命名模块中。未命名模块导出类路径中的所有代码并读取所有其他模块。然而这存在一个很大的限制：未命名模块本身只能通过自动模块读取！

图 8-3 显示了当读取未命名模块时自动模块和显式模块之间的区别。显式模块只能读取其他显式模块和自动模块。而自动模块可读取所有模块，包括未命名模块。

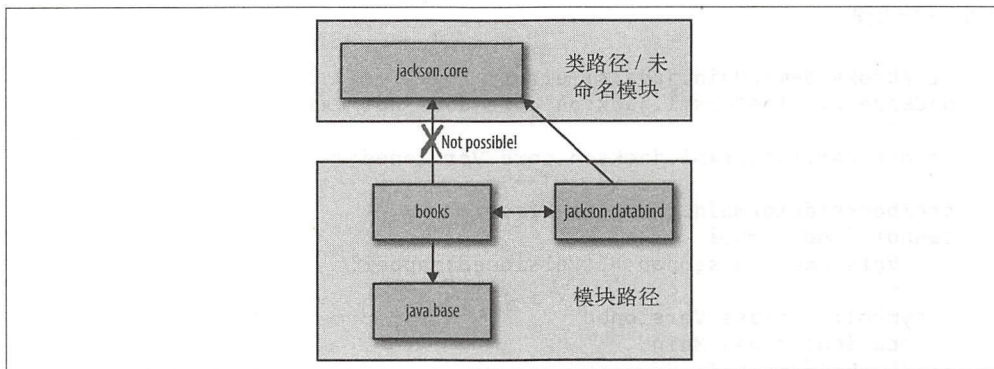


图 8-3: 只有自动模块可以读取类路径

未命名模块的可读性只是一种在混合类路径 / 模块路径迁移方案中有助于自动模块的机制。如果想要在代码中直接使用来自 Jackson Core 的类型（而不是来自自动模块），那么必须将 Jackson Core 移动到模块路径中。如示例 8-3 所示。

示例 8-3: Main.java (↪ *chapter8/readability_rules*)

```
package demo;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.Versioned; ❶

public class Demo {

    public static void main(String... args) throws Exception {
        Book modularityBook =
            new Book("Java 9 Modularity", "Modularize all the things!");

        ObjectMapper mapper = new ObjectMapper();
        String json = mapper.writeValueAsString(modularityBook);
        System.out.println(json);

        Versioned versioned = (Versioned) mapper; ❷
        System.out.println(versioned.version());
    }
}
```

❶ 从 Jackson Core 导出 Versioned 类型。

❷ 使用 Versioned 类型打印库版本。

Jackson Databind 的 ObjectMapper 类型实现了 Jackson Core 的 Versioned 接口。请注意，自在模块中显式地使用该类型之前都没有什么问题。当需要在一个模块中使用外部类型时，都应该立即考虑 requires。接下来通过编译该代码来证明这一点，此时将



产生一个错误：

```
src/books/demo/Main.java:4: error:
package com.fasterxml.jackson.core does not exist

import com.fasterxml.jackson.core.Versioned;
                                   ^
src/books/demo/Main.java:16: error:
cannot find symbol
    Versioned versioned = (Versioned)mapper;
    ^
symbol:   class Versioned
location: class Main
src/books/demo/Main.java:16: error:
cannot find symbol
    Versioned versioned = (Versioned)mapper;
    ^
symbol:   class Versioned
location: class Main
3 errors
```

尽管在未命名模块（类路径）中存在类型，并且 `jackson.databind` 自动模块可以访问它，但我们却无法从自己的模块中访问它。为了解决这个问题，需要将 Jackson Core 移动到模块路径中，使其成为一个自动模块。接下来将 JAR 文件移动到 `mods` 目录，并从类路径中移除它，就像为 Jackson Databind 所做的那样：

```
javac -cp lib/jackson-annotations-2.8.8.jar \
--module-path mods \
-d out \
--module-source-path src \
-m books
```

此时一切正常！后退一步想一想，为什么没有出现错误呢？显而易见，代码使用了 `jackson.core` 中的一个类型，但是 `module-info.java` 中并没有对 `jackson.core` 的 `requires`。编译为什么没有失败呢？请记住，自动模块对所有其他模块都会进行 `requires transitive`。这意味着通过请求 `jackson.databind`，也可以以传递的方式读取 `jackson.core`。虽然这样做很方便，但却需要一定的权衡。我们对一个没有明确需要的模块存在显式代码依赖。如果 `jackson.databind` 移动成为一个显式模块，并且 Jackson 维护者没有选择 `requires transitive jackson.core`，那么代码将突然中断。

请注意，虽然自动模块看起来像模块，但它们却缺少可以提供可靠配置的元数据。在这个特定的例子中，最好显式地向 `jackson.core` 添加一个 `requires`：

```
module books {
    requires jackson.databind;
    requires jackson.core;
```



```
    opens demo;
}
```

现在可以再次进行编译，也可以调整运行命令。必须从类路径中只删除 Jackson Core JAR 文件，因为已经正确配置了模块路径：

```
java \
  -cp lib/jackson-annotations-2.8.8.jar \
  --module-path out:mods \
  -m books/demo.Main
```

如果想要知道为什么 `jackson.core` 最先被解析（它并没有作为根模块显式添加到模块图中，也没有显式模块直接依赖它），那么可以回顾一下“模块解析和模块路径”一节所讨论的内容，即解析模块组是根据一组给定的根模块计算出来的。在自动模块的情况下，这将会产生混乱。自动模块没有显式依赖项，所以不会导致其传递依赖项（也是自动模块）也被解析。由于使用 `--add-modules` 手动添加依赖项是非常耗时的，因此当应用程序 `requires` 模块路径上任何一个自动模块时，所有自动模块都会被自动解析。这样一来，就可能导致未使用的自动模块也被解析（占用不必要的资源），所以请保持模块路径尽可能“干净”。

但这种行为与使用类路径时所产生的行为相类似，从而再次说明了自动模块的主要功能是实现从类路径到模块的迁移。

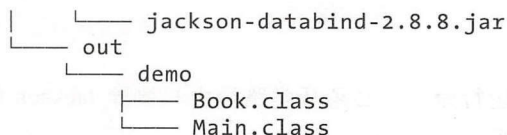
JVM 为什么没有那么“聪明”呢？它有权访问自动模块中的所有代码，那么为什么不分析依赖关系呢？如果想要分析这些代码是否调用其他模块，JVM 需要对所有代码执行字节码分析。虽然这不难实现，但却是一个昂贵的操作，可能会大量增加大型应用程序的启动时间。而且，这样的分析不会发现通过反射产生的依赖关系。由于存在这些限制，JVM 不可能也永远不会这样做。相反，JDK 附带了另一个工具 `jdeps`，它可以执行字节码分析。

8.9 使用 `jdeps`

在前面的 Jackson 示例中，使用了一种反复试验的方法来迁移代码。虽然这种方法很好地解释了所发生的事情，但效率不高。`jdeps` 是 JDK 附带的一个工具，用于分析代码并提供关于模块依赖关系的了解。可以使用 `jdeps` 来优化迁移 Jackson 示例的过程。

在迁移到（自动）模块之前，首先使用 `jdeps` 来分析示例的类路径版本。`jdeps` 分析的是字节码，而不是源文件，所以只对应用程序的输出文件夹和 JAR 文件感兴趣。为了便于参考，如本章开头所示，在使用类路径时使用了 `books` 示例的编译版本，如下所示：

```
|— lib
|   |— jackson-annotations-2.8.8.jar
|   |— jackson-core-2.8.8.jar
```



通过使用下面的命令，可以分析应用程序：

```

$ jdeps -recursive -summary -cp lib/*.jar out

jackson-annotations-2.8.8.jar -> java.base
jackson-core-2.8.8.jar -> java.base
jackson-databind-2.8.8.jar -> lib/jackson-annotations-2.8.8.jar
jackson-databind-2.8.8.jar -> lib/jackson-core-2.8.8.jar
jackson-databind-2.8.8.jar -> java.base
jackson-databind-2.8.8.jar -> java.desktop
jackson-databind-2.8.8.jar -> java.logging
jackson-databind-2.8.8.jar -> java.sql

jackson-databind-2.8.8.jar -> java.xml
out -> lib/jackson-databind-2.8.8.jar
out -> java.base
  
```

`-recursive` 标志确保可传递的运行时依赖项也被分析。如果没有它，`jackson-annotations-2.8.8.jar` 将不会被分析。顾名思义，`-summary` 标志总结了输出。在默认情况下，`jdeps` 输出每个包的依赖项的完整列表，这可能是一个非常长的列表。摘要仅显示模块相关性，并隐藏了包的详细信息。`-cp` 参数是希望在分析过程中使用的类路径，它应该与运行时类路径相对应。out 目录包含必须分析的应用程序的类文件。

从 `jdeps` 输出中可以学到以下内容：

- 代码对 `jackson-databind-2.8.8.jar` (当然也包括 `java.base`) 只有一个直接、编译时依赖。
- Jackson Databind 依赖 Jackson Core 和 Jackson Annotation。
- Jackson Databind 依赖多个平台模块。

根据上面的输出已经可以得出结论，为了将代码迁移到一个模块，还需要使 `jackson-databind` 成为一个自动模块。同时也看到，`jackson-databind` 依赖于 `jackson-core` 和 `jackson-annotations`，所以需要在类路径中提供它们或者以自动模块的形式提供。如果想知道为什么存在依赖关系，可以使用 `jdeps` 打印更多的细节。省略上述命令中的 `-summary` 参数，可以打印完整的依赖关系图，以及准确显示哪些包需要哪些其他包：

```

$ jdeps -cp lib/*.jar out

com.fasterxml.jackson.databind.util (jackson-databind-2.8.8.jar)
  
```




```
-> com.fasterxml.jackson.annotation jackson-annotations-2.8.8.jar
-> com.fasterxml.jackson.core jackson-core-2.8.8.jar
-> com.fasterxml.jackson.core.base jackson-core-2.8.8.jar
```

... Results truncated for readability

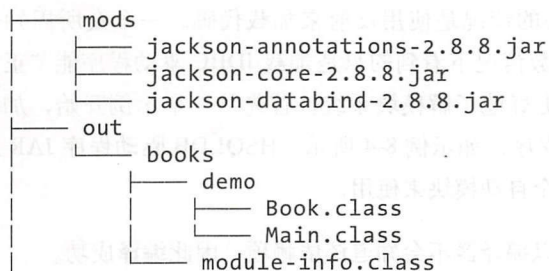
如果以上信息还不够详细，还可以指示 `jdeps` 以类别打印依赖项：

```
$ jdeps -verbose:class -cp lib/*.jar out
```

```
out -> java.base
      demo.Main (out)
        -> java.lang.Object
        -> java.lang.String
      demo.Main (out)
        -> com.fasterxml.jackson.databind.ObjectMapper jackson-databind-2.8.8.jar
```

... Results truncated for readability

到目前为止，已经在基于类路径的应用程序上使用了 `jdeps`。此外，其也可以用于模块。接下来在 Jackson 示例上使用 `jdeps`，其中所有的 Jackson JAR 都可以作为自动模块使用：



为了调用 `jdeps`，现在必须传入包含应用程序模块和自动 Jackson 模块的模块路径：

```
$ jdeps --module-path out:mods -m books
```

如前面一样，该命令打印了依赖关系图，从中可以看到以下内容：

```
module jackson.databind (automatic)
  requires java.base
  com.fasterxml.jackson.databind
    -> com.fasterxml.jackson.annotation jackson.annotations
  com.fasterxml.jackson.databind
    -> com.fasterxml.jackson.core jackson.core
  com.fasterxml.jackson.databind
    -> com.fasterxml.jackson.core.filter jackson.core
```

...

```
module books
  requires jackson.databind
  requires java.base
  demo -> com.fasterxml.jackson.databind jackson.databind
```



```
demo -> java.io java.base
demo -> java.lang java.base
```

可以看到 `jackson.databind` 对 `jackson.annotations` 和 `jackson.core` 都有依赖。同时，`books` 仅依赖于 `jackson.databind`。`books` 代码在编译时没有使用 `jackson.core` 类，同时没有为自动模块定义可传递依赖关系。请记住，JVM 在应用程序启动时不会执行此分析，这意味着必须亲自将 `jackson.annotations` 和 `jackson.core` 添加到类路径或模块路径中。`jdeps` 提供了正确设置所需的信息。



`jdeps` 可以使用 `-dotoutput` 标志输出模块图的点文件。该文件是表示图形的有用格式，可以很容易地通过该格式生成图像。关于该格式，Wikipedia 给出了非常详细的介绍 ([http://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](http://en.wikipedia.org/wiki/DOT_(graph_description_language)))。

8.10 动态加载代码

在迁移到模块时可能需要特别小心的情况是使用反射来加载代码。一个众所周知的例子是加载 JDBC 驱动程序。在大多数情况下看到的只是加载 JDBC 驱动程序能“正常工作”，但了解一些极端情况有助于更好地了解模块系统。首先从一个示例开始，加载位于项目 `mods` 目录下的 JDBC 驱动程序，如示例 8-4 所示。HSQLDB 驱动程序 JAR 还不是一个模块，所以只能把它作为一个自动模块来使用。

因为类的名称只是一个字符串，所以编译器不会知道该依赖项，因此编译成功。

示例 8-4: `Main.java` (↪ `chapter8/runtime_loading`)

```
package demo;

public class Main {
    public static void main(String... args) throws Exception {
        Class<?> clazz = Class.forName("org.hsqldb.jdbcDriver");
        System.out.println(clazz.getName());
    }
}
```

此时模块描述符 (如示例 8-5 所示) 为空；它没有请求 `hsqldb` (这是正在尝试加载的驱动程序)。虽然这是一个值得可疑的理由，但在理论上仍然成立，因为当在其他模块中对代码使用反射时，运行时会自动创建可读性关系。

示例 8-5: `module-info.java` (↪ `chapter8/runtime_loading`)

```
module runtime.loading.example {
}
```



如果运行代码，仍然会失败，并生成 `ClassNotFoundException`：

```
java --module-path mods:out -m runtime.loading.example/demo.Main

Exception in thread "main" java.lang.ClassNotFoundException:
    org.hsqldb.jdbcDriver
```

当应用程序至少使用其中一个时，所有可观察的自动模块都将被解析。解析发生在启动时，所以所创建的模块不会导致自动模块加载。如果有其他直接需要的自动模块，则会解析 `hsqldb` 模块并产生一些副作用。在这种情况下，可以使用 `--add-modules hsqldb` 自己添加自动模块。

现在驱动程序可以加载但又出现了另一个错误，因为驱动程序依赖于尚未解析的 `java.sql`。请记住，自动模块缺少元数据来具体要求其他模块。在实践中使用 JDBC 时，需要在模块中请求 `java.sql`，以便在加载驱动程序后能够使用 JDBC API。这意味着将其添加到模块描述符中，如示例 8-6 所示。

示例 8-6: `module-info.java` (`chapter8/runtime_loading`)

```
module runtime.loading.example {
    requires java.sql;
}
```

代码现在可以成功运行。有了模块所需的 `java.sql`，就可以看到另一个有趣的自动模块解析案例。如果再次删除 `--add-modules hsqldb`，应用程序仍然可以运行！那么请求 `java.sql` 为什么会导导致自动模块被加载呢？事实证明，`java.sql` 定义了一个 `java.sql.Driver` 服务接口，并且对这个服务类型也有一个 `uses` 约束。`hsqldb` JAR 提供了一个服务，它是通过在 `META-INF/services` 中使用文件的“旧”方式进行注册的。由于服务绑定，JAR 将从模块路径中自动解析。虽然这涉及了模块系统的细微之处，但却很好理解。

为什么不将 `requires hsqldb` 放到模块描述符中呢？虽然一般来说都希望将 `requires` 子句放置到模块描述符中，从而尽可能明确地表示依赖关系，但此时并不适用这种经验法则。要使用的 JDBC 驱动程序通常取决于应用程序的部署环境，其中确切的驱动程序名称在配置文件中配置。在这种情况下，应用程序代码不应该耦合到特定的数据库驱动程序（尽管在本示例中出现了耦合现象）。相反，只需确保通过添加 `--add-modules` 来解析驱动程序。包含驱动程序的模块将位于已解析的模块图中，反射实例化建立了与此模块的可读性关系。

如果 JDBC 驱动程序支持它（就像 HSQLDB 那样），那么最好避免应用程序代码的反射实例化，而使用服务。服务在第 4 章中已详细讨论。



8.11 拆分包

5.5.1 节已经介绍了拆分包所涉及的相关问题。现在复习一下，拆分包意味着两个模块包含相同的包。Java 模块系统不允许拆分包。

当使用自动模块时，也会遇到拆分包。在大型应用程序中，由于依赖关系管理不善，通常会发现拆分包。拆分包始终是一个错误，因为它们在类路径上不能可靠地工作。不幸的是，当使用解析可传递依赖项的构建工具时，很容易得到同一个库的多个版本。在类路径中找到的第一个类将被加载。当混合使用来自两个版本库的类时，往往会导致在运行时出现难以调试的异常。



现代构建工具通常有一个“在重复依赖项上失败”的设置，这使得依赖关系管理问题变得更加清晰，从而迫使尽早解决这些问题。强烈建议使用这个设置。

关于该问题，Java 模块系统比类路径要严格得多。当它检测到一个包从模块路径上的两个模块导出时，就会拒绝启动。相比于以前使用类路径时所遇到的不可靠情况，这种快速失败（fail-fast）机制要好得多。在开发过程中失败好过在生产过程中失败，尤其是当一些不幸的用户碰到一个由于模糊的类路径问题而被破坏的代码路径时。但这也意味着我们必须处理这些问题。盲目地将所有 JAR 从类路径移至模块路径可能导致在生成的自动模块之间出现拆分包。而这些拆分包将被模块系统所拒绝。

为了使迁移变得容易一些，当涉及自动模块和未命名模块时，上述规则存在一个例外，即承认很多类路径是不正确的，并且包含拆分包。当（自动）模块和未命名模块都包含相同的包时，将使用来自（自动）模块的包，而未命名模块中的包被忽略。对于作为平台模块一部分的包来说也是如此。通过在类路径上放置相关包来覆盖平台包是很常见的，但这种方法已经不再适用于 Java 9 了，从前面的章节中可以看到这一点。基于此原因，`java.se.ee` 模块不再包含在 `java.se` 模块中。

如果在迁移到 Java 9 时遇到了拆分包问题，那么是无法绕过的。即使从用户角度来看基于类路径的应用程序可以正确工作，你也必须处理这些问题。

本章介绍了许多逐步迁移到 Java 模块系统所需的技术。这些技术都非常有价值，因为在 Java 生态系统完全转移到 Java 模块系统之前需要花费一定的时间。自动模块在迁移场景中扮演着重要的角色，因此，充分了解它们的工作方式非常重要。



迁移案例研究：Spring 和 Hibernate

第 8 章介绍了将应用程序迁移到模块时可用的所有工具，本章结合前面所学的内容进行一个案例研究，将一个使用 Spring 和 Hibernate 的全功能应用程序迁移到模块。请注意，此时故意使用“传统的”Spring / Hibernate 开发的例子，而没有使用最现代的方式，并且使用 Java 9 之前的版本来创建一个有趣的案例研究。许多应用程序都是用这种方式编写的，这使得迁移这些应用程序变得更加有趣。这些框架的较新版本可以更好地支持 Java 9，基于这些版本的迁移也会变得更加容易。如果对这些框架不熟悉，也不要担心，因为没有必要为了了解在模块迁移时可能遇到的常见问题而熟悉所有代码和配置。

如果在阅读本章时仔细检查代码库并尝试迁移代码，那么就可以从中获得更多有价值的信息。在代码库中共提供了三个版本：

- *Chapter9/spring-hibernate-starter*：迁移之前应用程序的类路径版本。
- *Chapter9/spring-hibernate*：迁移后的应用程序。
- *Chapter9/spring-hibernate-refactored*：在进行了额外的模块化之后的迁移后的应用程序。

建议在编辑器中打开 *spring-hibernate-starter* 项目，并应用本章所介绍的每一步来处理代码。这样一来，可以得到与完成后的 *spring-hibernate* 示例大致相同的结果。

9.1 熟悉应用程序

该应用程序表示了一个书店。书籍通过使用 Hibernate 存储在数据库中。Spring 用来启动 Hibernate，包括事务管理和依赖注入。Spring 配置混合使用了 XML 和基于注释的配置。

在迁移之前，应用程序代码、直接依赖项和可传递依赖项都在类路径上，如图 9-1 所示。

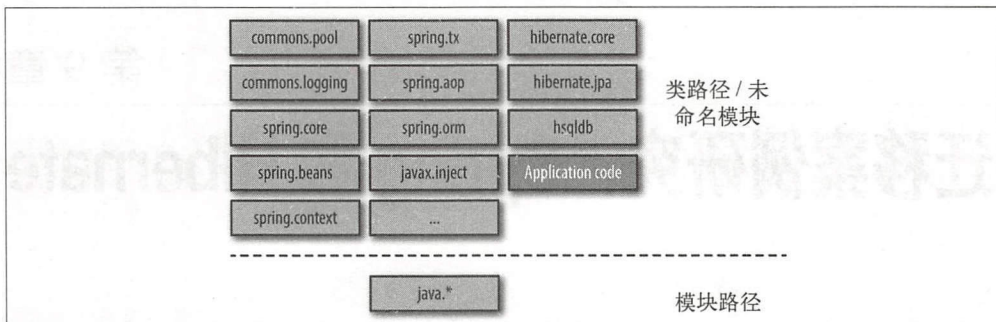


图 9-1: 迁移开始点 (↪ *chapter9/spring-hibernate-starter*)

迁移的最终结果是一个带有单个模块的代码库，并在必要时针对依赖项使用自动模块。图 9-2 显示了最终的结果。

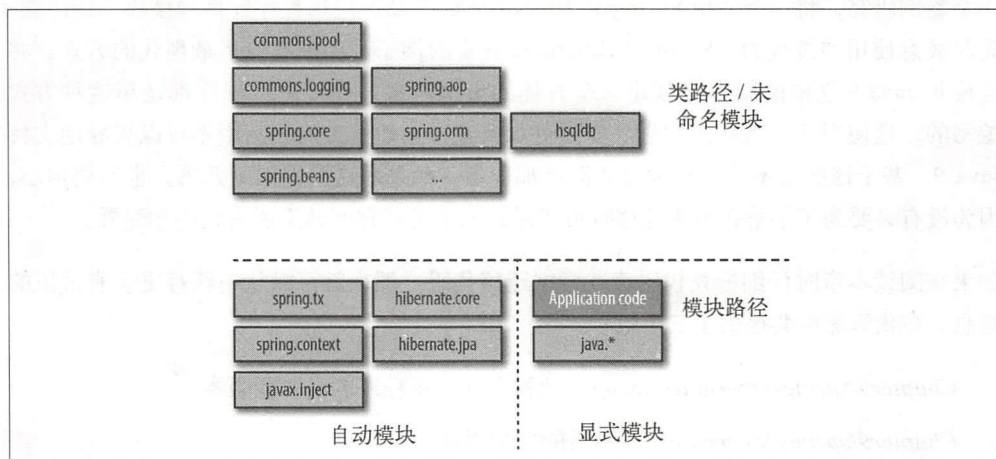


图 9-2: 迁移后的应用程序 (↪ *chapter9/spring-hibernate*)

在本章的最后，还将着重于对应用程序代码本身进行重构，以便更加模块化。

在开始考虑在模块中拆分代码之前，应该解决依赖关系中所存在的一些技术问题。请记住，正在处理的是 Java 9 之前的库，这些库不是为了与 Java 模块系统一起工作而设计的。具体来说，正在使用以下框架版本：

- Spring 4.3.2
- Hibernate 5.0.1

还要注意，在模块支持方面，这些框架已经取得了很大的进展。在编写本书的时候，Spring 5 的第一个 RC 版本已经发布了，并且明确支持用作自动模块。但本章示例中并



不使用这些更新后的版本，因为这样无法做出一个现实的迁移示例。即使没有框架和库的特殊支持，也可以迁移到模块。

本节的重点是为代码创建一个模块。这意味着需要为依赖关系定义 `requires`，并将库移动到自动模块。此外，还必须处理 `exports` 和 `opens`，以便代码可以被框架访问。一旦完成迁移，就可以仔细看看代码的设计，并将代码拆分成更小的模块。因为前面已经介绍过所有的技术问题，所以本示例成为一个很好的设计练习。

首先看看代码中最重要的部分，以便了解应用程序。为了获得最佳的代码可读性，强烈建议在你最喜欢的编辑器中打开代码。

`Book` 类（如示例 9-1 所示）是一个 JPA 实体，可以使用 Hibernate（或另一个 JPA 实现）将其存储在数据库中。它有诸如 `@Entity` 和 `@Id` 之类注释，以便将映射配置到数据库。

示例 9-1: `Book.java` (`chapter9/spring-hibernate`)

```
package books.impl.entities;

import books.api.entities.Book;
import javax.persistence.*;

@Entity
public class BookEntity implements Book {
    @Id @GeneratedValue
    private int id;
    private String title;
    private double price;
    // 简单起见，省略了 getter 和 setter
}

```

`HibernateBooksService` 是一个 `Spring Repository`。这是一个自动处理事务管理的 服务，用来将某些内容成功地存储到数据库中。它实现了服务接口 `BooksService`，并使用 Hibernate API（如 `SessionFactory`）存储和检索数据库中的书籍。

`BookstoreService` 是一个简单的接口，它在 `BookstoreServiceImpl` 中的实现（如示例 9-2 所示）可以计算给定书籍列表的总价格，并且使用了 Spring 的 `@Component` 进行注释，以便于依赖注入。

示例 9-2: `BookstoreServiceImpl.java` (`chapter9/spring-hibernate`)

```
package bookstore.impl.service;

import java.util.Arrays;
import books.api.entities.Book;
import books.api.service.BooksService;
import bookstore.api.service.BookstoreService;

```



```
import org.springframework.stereotype.Component;

@Component
public class BookstoreServiceImpl implements BookstoreService {

    private static double TAX = 1.21d;

    private BooksService booksService;

    public BookstoreServiceImpl(BooksService booksService) {
        this.booksService = booksService;
    }

    public double calculatePrice(int... bookIds) {
        double total = Arrays
            .stream(bookIds)
            .mapToDouble(id -> booksService.getBook(id).getPrice())
            .sum();

        return total * TAX;
    }
}
```

最后，有一个用来启动 Spring 并存储和检索书籍的主类，如示例 9-3 所示。

示例 9-3: Main.java ([↪ chapter9/spring-hibernate](#))

```
package main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import books.api.service.BooksService;
import books.api.entities.Book;
import bookstore.api.service.BookstoreService;

public class Main {

    public void start() {
        System.out.println("Starting...");

        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {"classpath:/main.xml"});

        BooksService booksService = context.getBean(BooksService.class);
        BookstoreService store = context.getBean(BookstoreService.class);

        // 创建一些书
        int id1 = booksService.createBook("Java 9 Modularity", 45.0d);
        int id2 = booksService.createBook("Modular Cloud Apps with OSGi", 40.0d);
        printf("Created books with id [%d, %d]", id1, id2);

        // 获取所创建的书
        Book book1 = booksService.getBook(id1);
        Book book2 = booksService.getBook(id2);
    }
}
```




```

printf("Retrieved books:\n %d: %s [%.2f]\n %d: %s [%.2f]",
      id1, book1.getTitle(), book1.getPrice(),
      id2, book2.getTitle(), book2.getPrice());

// 使用其他服务来计算总数
double total = store.calculatePrice(id1, id2);
printf("Total price (with tax): %.2f", total);
}

public static void main(String[] args) {
    new Main().start();
}

private void printf(String msg, Object... args) {
    System.out.println(String.format(msg + "\n", args));
}
}

```

Spring 通过使用需要 XML 配置的 `ClassPathXmlApplicationContext` 进行启动。在如示例 9-4 所示的配置中，设置了组件扫描，自动将 `@Component` 和 `@Repository` 注释类注册为 Spring bean，同时还设置了事务管理和 Hibernate。

示例 9-4: `main.xml` ([↩ chapter9/spring-hibernate](#))

```

<context:component-scan base-package="books.impl.service"/>
<context:component-scan base-package="bookstore.impl.service"/>

<bean id="myDataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:testdb"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>

<bean id="mySessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="annotatedClasses">
    <list>
        <value>books.impl.entities.BookEntity</value>
    </list>
    </property>

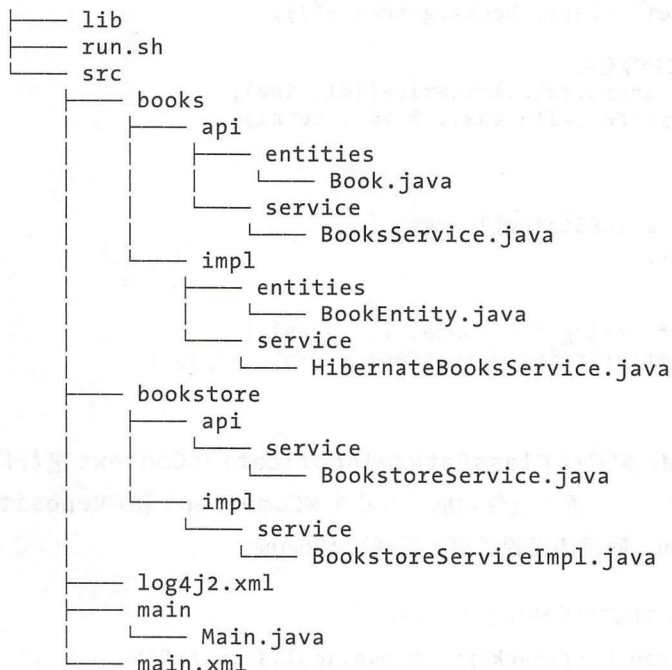
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.hbm2ddl.auto">create</prop>
    </props></property></bean> <bean id="transactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
</bean>

<tx:annotation-driven/>

```



项目的目录结构如下所示：



`src` 目录包含配置文件和源代码包。`lib` 目录包含 Spring、Hibernate 以及两者的可传递依赖项的 JAR 文件，这是一个包含 31 个 JAR 文件的长列表。为了构建和运行应用程序，可以使用以下命令：

```
javac -cp [list of JARs in lib] -d out -sourcepath src $(find src -name '*.java')
```

```
cp $(find src -name '*.xml') out
```

```
java -cp [list of JARs in lib]:out main.Main
```

9.2 使用 Java 9 在类路径上运行

向模块迁移的第一步应该从使用 Java 9 编译和运行代码开始，同时仍然使用类路径。这也显示了要解决的第一个问题。Hibernate 依赖于一些 JAXB 类。在 7.5 节中已经讲过，JAXB 是 `java.se.ee` 子图的一部分，但不是默认 `java.se` 模块子图的一部分。在不进行任何修改的情况下，运行 `Main` 会导致 `java.lang.ClassNotFoundException: javax.xml.bind`。此时，需要使用 `--add-modules` 标志将 JAXB 添加到应用程序中：

```
java -cp [list of JARs in lib]:out --add-modules java.xml.bind main.Main
```



接下来看到了另一个模糊的警告：

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by javassist.util.proxy.SecurityActions
(file:../lib/javassist-3.20.0-GA.jar)
to method java.lang.ClassLoader.defineClass(...)
WARNING: Please consider reporting this to the maintainers
of javassist.util.proxy.SecurityActions
WARNING: Use --illegal-access=warn to enable warnings of further illegal
reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

在 7.2 节中已经讨论过该问题。javassist 库尝试对 JDK 类型使用深度反射，默认情况下这是允许的，但会生成警告。如果用 `--illegal-access = deny` 来运行应用程序，甚至会变成错误。请记住，在未来的 Java 版本中，这将是默认设置。此时并不需要通过更新到 javassist 可能的固定版本来解决这个问题。不过，可以通过使用 `--add-opens` 来消除警告：

```
java -cp [list of JARs in lib]:out \
--add-modules java.xml.bind \
--add-opens java.base/java.lang=ALL-UNNAMED main.Main
```

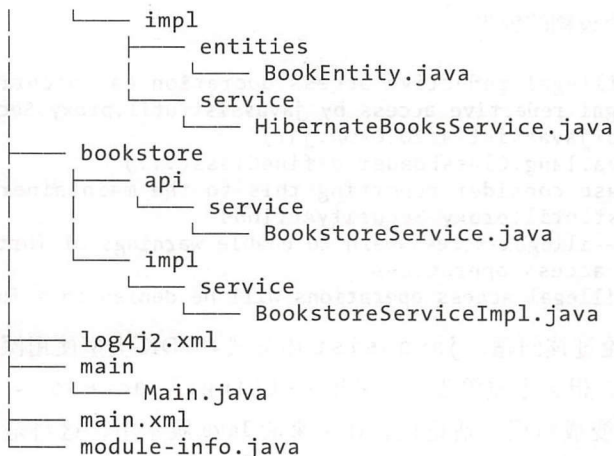
是否使用 `--add-opens` 来消除警告完全取决于你的决定。这样做可以为下一个 Java 版本做好准备，其对从类路径的非法访问的处理不会再像 Java 9 那样友好了。但 javassist 的问题行为仍然存在，所以向库护人员提出这些问题是正确的行为。

9.3 设置模块

解决了这些问题之后，就可以开始向模块迁移了。首先，当迁移到模块时，先保持代码内部结构不变是一个很好的策略。尽管此时的结构可能并不是你想要的最终结构，但把重点放在技术问题上可能更容易一些。

第一步是将 `-sourcepath` 更改为 `--module-source-path`。为此，需要稍微改变项目的结构。`src` 目录不应该直接包含包，而应该先包含一个模块目录。模块目录也应该包含 `module-info.java`：

```
├── lib
├── mods
├── run.sh
└── src
    └── bookapp
        └── books
            └── api
                ├── entities
                │   └── Book.java
                └── service
                    └── BooksService.java
```



修改编译 / 运行脚本，从而使用 `--module-source-path` 并从模块启动主类：

```

javac -cp [list of JARs in lib] \
  --module-path mods \
  -d out \
  --module-source-path src \
  -m bookapp

cp $(find src -name '*.xml') out/bookapp

java -cp out:[list of JARs in lib] \
  --module-path mods:out \
  --add-modules java.xml.bind \
  -m bookapp/main.Main
  
```

此时没有将任何库移动到模块路径，也没有将任何东西放在 `module-info.java` 中，所以前面的命令显然会失败。

9.4 使用自动模块

为了能够顺利编译模块，还需要在 `module-info.java` 中针对任何编译时依赖项添加 `requires` 语句。这也意味着需要将一些 JAR 文件从类路径移动到模块路径，以使它们成为自动模块。为了弄清楚有哪些编译时依赖项，可以在代码中查看 `import` 语句，或者使用 `jdeps`。对于示例应用程序，可以根据直接的编译时依赖项提出以下 `requires` 语句列表：

```

requires spring.context;
requires spring.tx;

requires javax.inject;

requires hibernate.core;
  
```



```
requires hibernate.jpa;
```

所有这些 `requires` 语句的含义都是不言自明的，来自这些模块的包可直接在示例代码中使用。为了能够请求这些库，可以将它们相应的 JAR 文件移动到模块路径，从而使它们成为自动模块。而可传递依赖项可以留在类路径上，如图 9-3 所示。



还记得 8.9 节吗？当需要在迁移过程中找出所需的模块时，`jdeps` 是非常有用的。例如，可以通过在应用程序的类路径版本上运行以下代码来找到 `requires`：

```
jdeps -summary -cp lib/*.jar out
out -> lib/hibernate-core-5.2.2.Final.jar
out -> lib/hibernate-jpa-2.1-api-1.0.0.Final.jar
out -> java.base
out -> lib/javax.inject-1.jar
out -> lib/spring-context-4.3.2.RELEASE.jar
out -> lib/spring-tx-4.3.2.RELEASE.jar
```

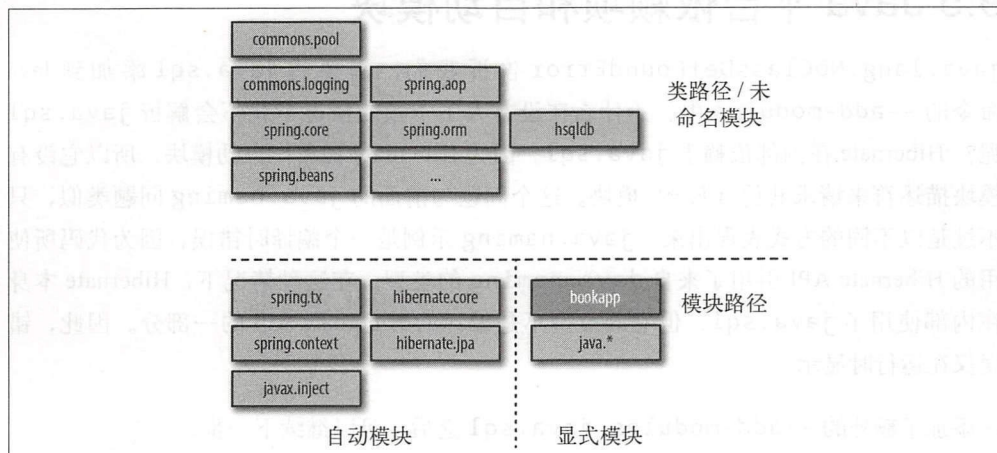


图 9-3：使用自动模块进行迁移

除了这些编译时依赖项之外，还有其他的编译时和运行时需求，可以使用 `--add-modules` 进行添加。如果要在 Hibernate API 中使用 `java.naming` 平台模块中的类型，就需要其在编译时可用，即使在代码中没有明确地使用该类型。如果没有显式地添加该模块，就会看到如下错误：

```
src/bookapp/books/impl/service/HibernateBooksService.java:19:
error: cannot access Referenceable
return sessionFactory.getCurrentSession().get(BookEntity.class, id);
      ^
class file for javax.naming.Referenceable not found
1 error
```



由于 Hibernate 被用作自动模块，因此不会导致额外的平台模块解析。不过，代码对其存在间接的编译时依赖关系，因为 Hibernate 在其 API 中使用了来自 `java.naming` 的类型（`SessionFactory` 接口扩展了 `Referenceable`）。这意味着存在编译时依赖关系，如果没有使用 `requires` 语句是行不通的。如果 Hibernate 是一个显式模块，那么它应该在 `module-info.java` 中包含 `requires transitive java.naming`，从而设置隐式可读性并防止上述问题出现。

在此之前，可以通过将 `--add-modules java.naming` 添加到 `javac` 来解决此问题。或者，可以在模块描述符中针对 `java.naming` 添加另一个 `requires`。正如前面所讨论的那样，应该避免针对间接依赖关系使用 `requires` 语句，因此选择使用 `--add-modules`。

应用程序现在编译成功，但运行时仍然会导致一些错误。

9.5 Java 平台依赖项和自动模块

`java.lang.NoClassDefFoundError` 告诉我们，需要将 `java.sql` 添加到 Java 命令的 `--add-modules` 中。为什么在没有人工干预的情况下就不会解析 `java.sql` 呢？Hibernate 在内部依赖于 `java.sql`。因为 Hibernate 被用作自动模块，所以它没有模块描述符来请求其他（平台）模块。这个问题与前面的 `java.naming` 问题类似，只不过是不同的方式表现出来。`java.naming` 示例是一个编译时错误，因为代码所使用的 Hibernate API 引用了来自 `java.naming` 的类型。在这种情况下，Hibernate 本身在内部使用了 `java.sql`，但它的类型不是编译的 Hibernate API 的一部分。因此，错误仅在运行时显示。

在添加了额外的 `--add-modules java.sql` 之后，可以继续下一步。

9.6 开放用于反射的包

现在离成功运行应用程序又近了一步，但重新运行应用程序仍然会导致错误。这一次的 error 是非常简单的：

```
Caused by: java.lang.IllegalAccessException:
  class org.springframework.beans.BeanUtils cannot access class
    books.impl.service.HibernateBooksService (in module bookapp) because module
    bookapp does not export books.impl.service to unnamed module @5c45d770
```

Spring 依靠反射来实例化类。为此，必须开放包含 Spring 实例化所需类的实现包。同样的道理，Hibernate 也使用反射来操纵实体类。Hibernate 需要访问 `Book` 接口和 `BookEntity` 实现类。



对于应用程序中的 API 包，导出它们是有意义的。稍后当拆分成更多的模块时，这些包也极有可能被其他模块所使用。对于实现包，则使用 `opens`。这样一来，框架可以完成自己的反射“魔术”，而我们在构建时仍然可以保持良好的封装：

```
exports books.api.service;
exports books.api.entities;

opens books.impl.entities;
opens books.impl.service;
opens bookstore.impl.service;
```

在更大型的应用程序中，一个比较好的选择是首先使用开放式模块，而不是指定单个要开放的包。使用 `opens/exports` 设置包后，会看到另一个熟悉的错误。

```
Java.lang.NoClassDefFoundError: javax/xml/bind/JAXBExceptio
```

其中一个库正在使用 JAXB（而不是我们所编写的代码），在 7.5 节中讲过，默认情况下不会解析 `java.xml.bind`。就像在类路径上运行该示例时所做的那样，将模块添加到 `--add-modules` 有助于摆脱这种情况。

就快大功告成了！

不幸的是，当试图运行应用程序时，`javassist` 库给了一个最难以理解的错误：

```
Caused by: java.lang.IllegalAccessError: superinterface check failed:
  class books.impl.entities.BookEntity_$$jvstced_0 (in module bookapp)
  cannot access class javassist.util.proxy.ProxyObject
  (in unnamed module @0x546621c4) because module bookapp
  does not read unnamed module @0x546621c4
```

9.7 解决非法访问问题

Hibernate 使用 `javassist` 库来动态创建实体类的子类。在运行时，应用程序代码使用的是这些子类而不是原来的类。因为代码是从一个模块运行的，所以生成的类最终成为同一个 `bookapp` 模块的一部分。所生成的类实现了 `javassist` 的一个接口 (`ProxyObject`)。但是，`javassist` 仍然在类路径中，这是显式模块无法访问的。因此，生成的类实现了一个在运行时无法访问的接口。虽然这是一个难以理解的错误，但却非常容易解决：将 `javassist` 从类路径移动到模块路径，使其成为自动模块，从而可以从其他模块访问。

但是，将 `javassist` 转换为自动模块引入了一个新问题。前面已经看到，`javassist` 在 JDK 类型上使用了非法的深度反射。在类路径中，由于默认值 `--illegal-access = permit` 较为宽容，因此只给出了一个警告。因为 `javassist` 现在是一个自动模块，所以 `--illegal-access` 机制不再适用，它只影响类路径上的代码。这意味着现



在会收到一个错误，从本质上讲，该错误与使用 `--illegal-access=deny` 运行类路径示例时所看到的错误是一样的：

```
Caused by: java.lang.reflect.InaccessibleObjectException:
  Unable to make protected final java.lang.Class
  java.lang.ClassLoader.defineClass(...)
  throws java.lang.ClassFormatError accessible:
  module java.base does not "opens java.lang" to module javassist
```

现在已经知道可以通过向 Java 命令添加 `--add-opens java.base/java.lang=javassist` 来解决上述问题。示例 9-5 给出了用来编译和运行应用程序的最终脚本。

示例 9-5: `run.sh` (↪ *chapter9/spring-hibernate*)

```
CP=[list of JARs in lib]

javac -cp $CP \
      --module-path mods \
      --add-modules java.naming \
      -d out \
      --module-source-path src \
      -m bookapp
cp $(find src -name '*.xml') out/bookapp

java -cp $CP \
     --module-path mods:out \
     --add-modules java.xml.bind,java.sql \
     --add-opens java.base/java.lang=javassist \
     -m bookapp/main.Main
```

通过将真正需要的库转换为自动模块来实现应用程序的迁移。或者，也可以通过将所有 JAR 文件复制到模块路径来开始迁移。虽然这样做通常可以更容易地使应用程序快速地运行，但是却难以以为应用程序模块提供合理的模块描述符。由于自动模块为所有其他模块设置了隐式可读性，因此它会在模块中隐藏所缺少的 `requires`。当将自动模块升级为显式模块时，事情可能会中断。尽可能明确地定义模块依赖关系是很重要的，所以应该在这方面多花些时间来研究。

9.8 重构到多个模块

现在已经有有了一个可以工作的应用程序，如果可以将代码拆分成更小的模块，并且在应用程序的设计中实现模块化，那就更好了。虽然这些内容超出了本章的讨论范围，但是 GitHub 资料库包含了一个带有多个模块的实现 (↪ *chapter9/spring-hibernate-refactored*)。图 9-4 为这种改进的结构提供了合理的设计。

应该可以看到，该设计进行了一定的权衡。例如，可以选择是创建一个单独的 API 模块还是从同样包含实现的模块中导出 API。第 5 章已经详细讨论过许多类似的选择。

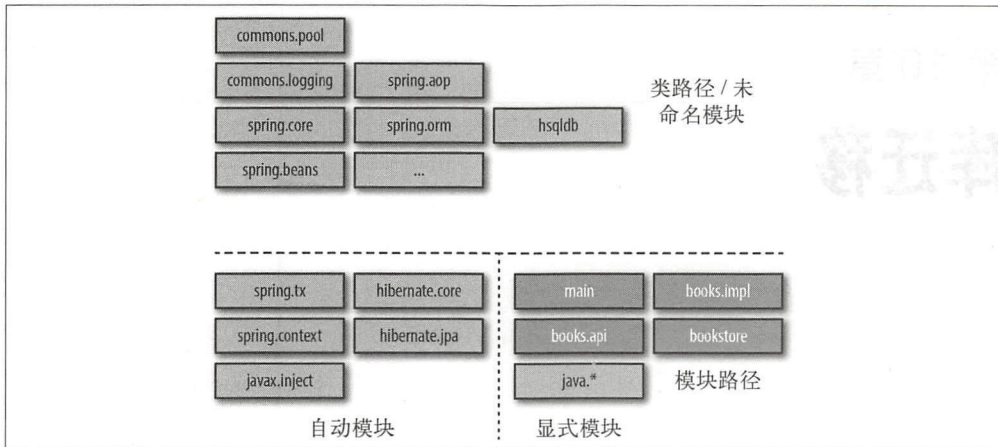


图 9-4: 重构的应用程序

通过这个案例研究，可以了解将现有的基于类路径的应用程序迁移到模块所需的所有工具和流程。使用 `jdeps` 分析现有的代码和依赖关系。将库移动到模块路径以使其转换为自动模块，从而允许为应用程序创建模块描述符。当应用程序使用涉及反射的库时（如依赖注入、对象关系映射或序列化库），则需要开放包和模块。

无论是在应用程序还是其库中，将应用程序迁移到模块可能会破坏强封装。正如前面所看到的，有时这可能会导致一些莫名其妙的错误，尽管可以使用对模块系统的了解来解释相关错误。在本章中，已经学会了如何解决这些问题。但是，如果库已经是具有显式模块描述符的正确 Java 9 模块了，那么一切就会更好了。下一章将介绍库维护人员如何支持 Java 9。



第 10 章

库迁移

前面的章节着重于将应用程序迁移到模块系统。如果是迁移一个现有库，那么很多内容也是适用的。但是与应用程序迁移相比，还有几个问题影响了库迁移，本章将会介绍这些问题及其解决方案。

迁移库和迁移应用程序之间最大的区别在于库被许多应用程序使用。这些应用程序可能运行在不同版本的 Java 上，所以库通常需要在各种 Java 版本上工作。期望库的用户在你迁移库的同时切换到 Java 9 是不现实的。幸运的是，Java 9 中的新功能组合为库维护人员和用户提供了无缝的体验。

本章的目标是将现有库逐步迁移为模块化的库。但你并不需要为了兴趣而成为流行开源项目的作者。如果编写与公司其他团队共享的代码，那么你和他们就在同一条船上。

库的迁移过程由以下步骤组成：

- 1) 确保库可以作为自动模块在 Java 9 上运行。
- 2) 使用 Java 9 编译器编译库（主要使用满足需求的最低 Java 版本），而不使用新的 Java 9 功能。
- 3) 添加一个模块描述符，并将库转换为显式模块。
- 4) 重构库的结构，以便增加封装性，识别 API，并尽可能分割成多个模块（可选）。
- 5) 开始使用库中的 Java 9 功能，同时保持向后兼容 Java 9 的早期版本。

虽然第 2 步是可选的，但建议完成该步骤。通过新引入的 `-- release` 标志，可以使用 Java 9 编译器可靠地针对 Java 的早期版本进行编译。在 10.5 节中将会介绍如何使用此选项。在所有步骤中，都可以保持与 Java 的早期版本的向后兼容性。最后一步可能是特别令人惊讶的，该步骤是使用本章最后所探讨的一个新特性（多版本 JAR）来实现的。



10.1 模块化之前

首先，需要确保库可以与 Java 9 一起使用。许多应用程序都是在类路径上使用库，甚至在 Java 9 上也是如此。此外，库维护者还需要修改库，以便在应用程序中作为自动模块使用。在许多情况下是不需要更改代码的，唯一需要做的更改是防止“搅局者”（showstopper），比如使用 JDK 中的封装或删除类型。

在将一个库转变为一个模块（或模块集合）之前，应该像迁移应用程序一样采取相同的初始步骤（详见第 7 章）。确保在 Java 9 上运行库意味着它不应该使用 JDK 中的封装类型。如果使用这种类型，库用户可能会遇到警告或异常（如果它们使用推荐的 `--illegal-access = deny` 设置运行应用程序）。这就迫使他们使用 `--add-opens` 或 `--add-exports` 标志。但即使在库中使用这些标志，也不是一个良好的用户体验。通常，库只是应用程序中众多应用中的一个，因此，追踪所有正确的命令行标志对用户来说是非常痛苦的。最好是使用 `jdeps` 来查找库中封装 API 的使用，并将其更改为建议的替换内容。如“使用 `jdeps` 查找已删除或封装的类型及其替代方法”内容中所述，可以使用 `jdeps -jdkinternals` 快速发现相关问题。如果这些替换 API 仅从 Java 9 之后开始可用，那么当需要支持较早的 Java 版本时就不能直接使用它们。在 10.7.1 节中，将会学习多版本 JAR 如何解决此问题。

目前，还没有为库创建模块描述符，可以推迟考虑需要导出哪些包。另外，库对其他库的依赖关系也可以是隐式的。无论该库是放在类路径上，还是作为自动模块放在模块路径上，都可以访问所需的所有内容，而无须显式的依赖关系。

在上述步骤之后，就可以在 Java 9 上使用该库了，目前在库的实现中没有使用 Java 9 中的任何新特性。事实上，如果没有使用封装或删除的 API，甚至不需要重新编译库。

10.2 选择库模块名称

计算机科学中只存在两个难题：缓存失效和命名问题。

—Phil Karlton

此时应该重点考虑的是模块在以后变成真正的模块时应该有的名称。对于库模块来说，什么是好名称呢？一方面，希望名称简单而难忘。另一方面，由于正在讨论的是一个可广泛重用的库，所以名称必须是独一无二的。对于任何给定的名称，模块路径上只能有一个模块。

在 Java 世界中创建全球唯一名称的传统方法是使用反向 DNS 表示法。当有一个名为 `mylibrary` 的库时，其模块名可以是 `com.mydomain.mylibrary`。将这种命名约定



应用于不可复用的应用程序模块是没有必要的，但是对于库来说，额外的视觉噪声是很有必要的。例如，将几个开源 Java 库命名为 `spark`。如果这些库维护人员没有采取预防措施，那么他们可能会声明相同的模块名称，这意味着应用程序不能在程序中一起使用这些库。声明一个反向的基于 DNS 的模块名是防止此类冲突的最好方法。



模块名称的最佳候选者是模块中所有包的最长公共前缀。假设使用反向 DNS 包名称，顶级包名称就是一个自然的模块标识符。在 Maven 术语中，这通常意味着将 *组 ID* 和 *工件 ID* 组合为模块名称。

数字在模块名称中是允许的。虽然在模块名称中添加一个版本号（例如，`com.mydomain.mylibrary2`）可能会更有吸引力，但是不要这样做。版本控制是识别库的另一个独立问题。虽然创建模块化 JAR 时可以设置版本信息（如 5.7 节中所述），但不应该是模块名称的一部分。因为用一个主版本来升级库并不意味着库的标识应该改变。很久以前，一些流行的库已经陷入了这种困境中。例如，Apache `commons-lang` 库在从版本 2 迁移到版本 3 时使用了 `commons-lang3` 名称。目前，版本控制属于构建工具和工件存储库领域，而不是模块系统。库的更新版本不应导致模块描述符的更改。

在第 8 章中曾经讲过，自动模块的名称来自 JAR 文件名。但不幸的是，最终的文件名通常是由构建工具或其他过程所确定的，而库维护者无法控制这些工具或过程。将库作为自动模块使用的应用程序通过模块描述符中的派生名请求库。当库日后切换成一个显式模块时，你就会被这个派生名所困扰。当派生文件名不完全正确或不是唯一标识时，就会出现这个问题。当不同的人使用库的不同文件名时，情况就更糟糕了。期望每个使用该库的应用程序将它们的 `requires` 子句更新为新的模块名称是不现实的。

这使得库维护人员处于一个尴尬的境地。尽管库本身不是一个模块，但它将通过自动模块功能被使用。当应用程序开始这样做时，实际上会被一个自动派生的模块名称所困扰，因为该名称可能并不是想要的。为了解决这个难题，可以采用保留模块名称的另一种方法。

首先在 `META-INF/MANIFEST.MF` 中添加一个 `Automatic-Module-Name:<module_name>` 条目到非模块化 JAR。当该 JAR 被用作自动模块时，它将采用清单中定义的名称，而不是从 JAR 文件名中派生名称。现在，库维护人员可以定义库模块应该具有的名称，而无须创建模块描述符。只需将具有正确模块名称的新条目添加到 `MANIFEST.MF` 并重新打包库就足够了。`jar` 命令有一个 `-m <manifest_file>` 选项，告诉它来自给定文件的条目添加到 JAR 中所生成的 `MANIFEST.MF` 中（`chapter10/modulename`）：



```
jar -cfm mylibrary.jar src/META-INF/MANIFEST.MF -C out/ .
```

通过使用该命令，来自 `src/META-INF/MANIFEST.MF` 的条目将被添加到输出 JAR 中生成的清单中。



通过使用 Maven，可以配置 JAR 插件来添加清单条目：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestEntries>
        <Automatic-Module-Name>
          com.mydomain.mylibrary
        </Automatic-Module-Name>
      </manifestEntries>
    </archive>
  </configuration></plugin>
```

在第 11 章，将会了解更多关于 Maven 对模块系统的支持。

在清单中使用 `Automatic-Module-Name` 来保留模块名称是应该尽快完成的事情。命名是很难的，应该有意识地选一个名称。但是，在解决了模块名称之后，使用 `Automatic-Module-Name` 保留模块名称就比较简单了。这是一个省力且高效的方法：不需要更改代码或重新编译。



只有在确保库可以在 JDK 9 上作为自动模块运行时，才能将 `Automatic-Module-Name` 添加到库的清单中。该清单条目的存在标志着 Java 9 的兼容性。在将库作为模块使用之前，必须解决前面章节中所描述的任何迁移问题。

为什么不创建一个模块描述符？有以下几个原因。

首先，要考虑公开哪些包。可以显式地导出所有内容，就像将库用作自动模块时那样。但是，如果在模块描述符中显式地导出所有内容，那么这种行为是不能轻易收回的。将库用作自动模块的人都知道，可以访问所有包是自动模块的副作用，稍后可能会更改。

其次，也是更重要的原因，库本身可能存在外部依赖。如果使用了模块描述符，那么库不再是模块路径上的自动模块。这意味着它不会自动与所有其他模块和类路径（未命名模块）存在一个 `requires` 关系。必须在模块描述符中显式确定所有的依赖关系。而那些外部依赖项可能还没有被模块化，这样一来就会阻碍库拥有正确的模块描述符。如果一个依赖项还没有 `Automatic-Module-Name` 清单条目，那么就不要再使用自动模块上



的该依赖项来发布库。因为此时依赖项的名称不稳定，当依赖的（派生的）模块名称改变时，就会导致模块描述符无效。

最后，模块描述符必须用 Java 9 编译器进行编译。这些都是需要时间才能正确完成的重要步骤。在执行所有这些步骤之前，在清单中使用简单的 `Automatic-Module-Name` 条目保留模块名称是非常明智的。

10.3 创建模块描述符

现在，库已正确命名并可用于 Java 9，接下来可以考虑将其变成一个显式模块。先假设库是一个单一 JAR (`mylibrary.jar`)，并转换为一个单一模块。之后，可能需要重新访问库的包装并进一步拆分。



在 10.6 节中将会看到更复杂的场景，其中库由多个模块组成或具有外部依赖关系。

关于创建模块描述符，有两个选择：从头创建一个或者使用 `jdeps` 根据当前的 JAR 生成一个。不管使用哪种方法，最重要的是模块描述符拥有与之前在清单 `Automatic-Module-Name` 条目中所选择的模块名称相同的模块名称。当被用作自动模块时，这样做可以让新模块成为老版本库的替代品。如果使用模块描述符，则可以删除清单条目。

示例 `mylibrary` (`chapter10/generate_module_descriptor`) 非常简单，由两个包中的两个类组成。核心类 `MyLibrary` 包含以下代码：

```
package com.javamodularity.mylibrary;

import com.javamodularity.mylibrary.internal.Util;

import java.sql.SQLException;
import java.sql.Driver;
import java.util.logging.Logger;

public class MyLibrary {

    private Util util = new Util();
    private Driver driver;

    public MyLibrary(Driver driver) throws SQLException {
        Logger logger = driver.getParentLogger();
        logger.info("Started MyLibrary");
    }
}
```



从功能上讲，上述代码所完成的事情并不重要，重点是在导入部分。在创建模块描述符时，需要确定所需的其他模块。目测检查会发现 `MyLibrary` 类使用了 JDK 中来自 `java.sql` 和 `java.logging` 的类型。…`internal.Util` 类来自同一个 `mylibrary.jar` 中的不同包。可以使用 `jdeps` 来列出所有依赖项，而不要尝试自己写出正确的 `requires` 子句。除了列出依赖项之外，`jdeps` 甚至可以生成一个初始模块描述符：

```
jdeps --generate-module-info ./out mylibrary.jar
```

上述代码可以在 `out/mylibrary/module-info.java` 中生成模块描述符：

```
module mylibrary {
    requires java.logging;
    requires transitive java.sql;
    exports com.javamodularity.mylibrary;
    exports com.javamodularity.mylibrary.internal;
}
```

`jdeps` 分析 JAR 文件并将依赖关系报告给 `java.logging` 和 `java.sql`。有趣的是，前者生成了一个 `requires` 子句，而后者生成一个 `requires transitive` 子句。这是因为 `MyLibrary` 中使用的 `java.sql` 类型是公共导出的 API 的一部分。`java.sql.Driver` 类型用作 `MyLibrary` 公共构造函数的参数。另一方面，`java.logging` 中的类型仅用于 `MyLibrary` 的实现，不会向库用户公开。在默认情况下，所有包都会导出到 `jdeps` 生成的模块描述符中。



当库包含包之外的类（在未命名的包中，俗称默认包）时，`jdeps` 会产生一个错误。模块中的所有类都必须是命名包的一部分。即使在模块出现之前，将类放在未命名的包中也是一种不好的做法——尤其是对于可重用的库而言。

此时，你可能会认为这个模块描述符提供了与 `mylibrary` 被用作自动模块时一样的行为。在一定程度上讲是这样的。但是，自动模块也是开放式模块。而生成的模块描述符并没有定义开放式模块，也没有开放任何包。库的用户只有在对 `mylibrary` 的类型进行深度反射时才会注意到这个区别。如果希望库用户及时注意到这一点，可以生成一个开放的模块描述符：

```
jdeps --generate-open-module ./out mylibrary.jar
```

上述代码生成如下所示的模块描述符：

```
open module mylibrary {
    requires java.logging;
    requires transitive java.sql;
}
```



此时，所有的包将被开放，因为生成了一个开放式模块。但并没有生成 `exports` 语句。如果向该开放式模块添加所有包的 `exports` 语句，则其行为类似于将原始 JAR 用作自动模块。

创建一个非开放式模块可能更好，即仅导出必要的最小数量的包。将库转换为模块的主要原因之一是强封装所带来的好处，这是开放式模块所不具备的。



应该始终将生成的模块描述符视为一个起点。

在大多数情况下，没有必要导出所有包。对于 `mylibrary` 示例，删除 `exports com.javamodularity.mylibrary.internal` 是有意义的。`mylibrary` 的用户不需要依赖内部的实现细节。

此外，如果库使用了反射，`jdeps` 将不会找到这些依赖项，需要为反射加载的模块添加正确的 `requires` 子句。如 5.6.1 节所述，如果依赖关系是可选的，那么这些子句可以是 `requires static`。如果库使用了服务，则必须手动添加 `uses` 子句。所提供的任何服务（通过 `META-INF/services` 中的文件）都会被 `jdeps` 自动提取，并转换为 `provides...with` 子句。

最后，`jdeps` 会根据文件名建议一个模块名称，就像自动模块一样。10.2 节讨论的注意事项此时仍然适用。对于库，最好使用反向 DNS 表示法来创建完全限定的名称。在本示例中，`com.javamodularity.mylibrary` 是首选的模块名称。当正在生成模块描述符的 JAR 已经包含 `Automatic-Module-Name` 清单条目时，建议使用此名称。

10.4 使用模块描述符更新库

在创建或生成模块描述符之后，留下了一个仍然需要编译的 `module-info.java`。只有 Java 9 可以编译 `module-info.java`，但这并不意味着需要将整个项目的编译切换到 Java 9。事实上，可以用已编译的模块描述符来更新现有的 JAR（使用较早的 Java 版本编译）。接下来看一下 `mylibrary.jar` 是如何工作的，此时生成并添加 `module-info.java`：

```
mkdir mylibrary
cd mylibrary
jar -xf ../mylibrary.jar ❶
cd ..
javac -d mylibrary out/mylibrary/module-info.java ❷
```




```
jar -uf mylibrary.jar -C mylibrary module-info.class ③
```

- ① 将类文件提取到 `./mylibrary`。
- ② 使用 Java 9 编译器将 `module-info.java` 编译到与所提取类相同的目录中。
- ③ 使用已编译的 `module-info.class` 更新现有的 JAR 文件。

通过以上步骤，可以由 Java 9 之前的 JAR 创建模块化的 JAR。模块描述符被编译到与所提取类相同的目录中，这样一来，`javac` 可以看到模块描述符中所提到的所有现有类和包，所以它不会产生错误。无须访问库的来源就做到这一点是可能的。没有必要重新编译现有的代码，当然除非需要改变代码，例如为了避免使用封装的 JDK API。

在完成上述步骤之后，可以在各种设置中使用生成的 JAR 文件：

- 在 Java 9 之前版本的类路径上使用。
- 在 Java 9 以及后续版本的模块路径上使用。
- 在 Java 9 的类路径上使用。

如果将 JAR 放在早期 Java 版本的类路径中，那么将忽略已编译的模块描述符。只有在 Java 9 或更高版本的模块路径上使用 JAR 时，模块描述符才会起作用。

10.5 针对较旧的 Java 版本

如果需要编译库的源文件以及模块描述符，那么又该怎么办呢？在很多情况下，需要针对 Java 9 之前的 Java 版本完成该操作。可以通过几种方式来实现。首先是使用两个 JDK 分别编译库源和模块描述符。

假设希望 `mylibrary` 在 Java 7 及更高版本上可用。实际上，这意味着库的源代码不能使用 Java 7 之后引入的任何语言功能，也不能使用 Java 7 之后添加的任何 API。通过使用两个 JDK，可以确保库的源代码不依赖于 Java 7+ 功能，同时仍然能够编译模块描述符：

```
jdk7/bin/javac -d mylibrary <all sources except module-info>  
jdk9/bin/javac -d mylibrary src/module-info.java
```

同样，对于两个编译运行来说，输出到同一个目录是至关重要的。这样一来，可以像前面的示例一样，将生成的类打包成模块化的 JAR。但管理多个 JDK 可能有点麻烦。在 JDK 9 中添加了一项新功能，允许针对较早的版本使用最新的 JDK。

可以使用 JDK 9 包含的 `--release` 新标志来编译 `mylibrary` 示例：

```
jdk9/bin/javac --release 7 -d mylibrary <all sources except module-info>  
jdk9/bin/javac --release 9 -d mylibrary src/module-info.java
```



这个新标志保证至少支持当前 JDK 的前三个主要版本。在 JDK 9 的情况下，这意味着可以针对 JDK 6、7 和 8 进行编译。这样做的额外好处是，即使库是针对早期版本开发的，也可以受益于 JDK 9 编译器中的错误修复和优化。如果需要支持更早版本的 Java，则随时可以使用多个 JDK。

release 标志

`--release` 标志是通过 JEP 247 (<http://openjdk.java.net/jeps/247>) 添加的。在此之前，可以使用 `-source` 和 `-target` 选项。这些标志确保不会使用错误级别的语言特性 (`-source`)，并且生成的字节码符合正确的 Java 版本 (`-target`)。但是，这些标志没有强制正确使用目标 JDK 的 API。当使用 JDK 8 进行编译时，可以指定 `-source 1.7 -target 1.7`，并在代码中仍使用 Java 8 API (尽管禁止使用 Lambda 表达式等语言功能)。当然，所生成的字节码不能在 JDK 7 上运行，因为它不提供新的 Java 8 API。必须使用外部工具，如 *Animals Sniffer* (<http://www.mojhaus.org/animal-sniffer/>) 来验证后向 API 兼容性。如果使用 `--release`，正确的库级别也由 Java 编译器强制执行——不再需要安装和管理多个 JDK。

10.6 库模块依赖关系

到目前为止，已经假定要迁移的库在 JDK 的模块之外没有任何依赖关系。实际上，情况通常并非如此。一个库存在依赖项通常有两个主要原因：

- 1) 库由多个相关联的 API 组成。
- 2) 库使用了外部库。

在第一种情况下，库中的 JAR 之间存在依赖关系。在第二种情况下，库需要其他外部 JAR。接下来将讨论这两种情况。

10.6.1 内部依赖关系

接下来，将根据第 8 章中所介绍的库 Jackson 来研究一下第一种情况。Jackson 由多个 JAR 组成。该示例基于 Jackson Databind 以及两个相关的 Jackson JAR，如图 10-1 所示。

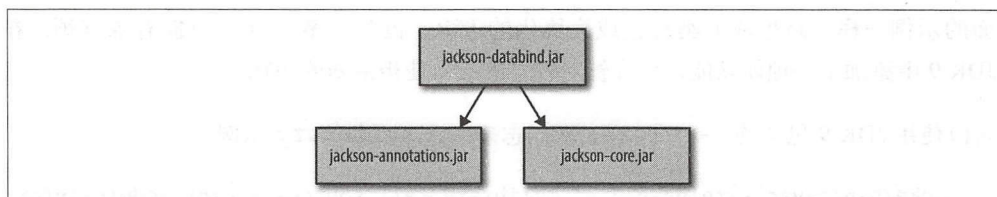


图 10-1：三个相关的 Jackson JAR



将这些 JAR 转换为模块是明智之举，可以保留当前的边界。幸运的是，jdeps 还可以为相关的 JAR 文件（[↪ chapter10/generate_module_descriptor_jackson](#)）同时创建多个模块描述符：

```
jdeps --generate-module-info ./out *.jar
```

上述代码生成三个模块描述符：

```
module jackson.annotations {
    exports com.fasterxml.jackson.annotation;
}

module jackson.core {
    exports com.fasterxml.jackson.core;
    // 简单起见，省略了其他包的导出
    provides com.fasterxml.jackson.core.JsonFactory with
        com.fasterxml.jackson.core.JsonFactory;
}

module jackson.databind {
    requires transitive jackson.annotations;
    requires transitive jackson.core;
    requires java.desktop;
    requires java.logging;
    requires transitive java.sql;
    requires transitive java.xml;
    exports com.fasterxml.jackson.databind;
    // 简单起见，省略了其他包的导出
    provides com.fasterxml.jackson.core.ObjectCodec with
        com.fasterxml.jackson.databind.ObjectMapper;
}
```

可以在最后两个模块描述符中看到，jdeps 也考虑到了服务提供者。如果 JAR 包含服务提供者文件（请参阅“Java 9 之前的 ServiceLoader”内容，以了解关于此机制的更多信息），那么这些文件将被翻译成 `provide ... with` 子句。



另一方面，服务 `uses` 子句不能由 jdeps 自动生成。这些子句必须根据库中的 ServiceLoader 使用情况手动添加。

`jacksons.databind` 描述符基于 jdeps 分析请求正确的平台模块。此外，它还需要其他 Jackson 库模块，其描述符在同一运行中生成。Jackson 的隐式结构在生成的模块描述符中自动变为显式。当然，标定模块实际 API 的艰巨任务则留给了 Jackson 维护者来完成。将所有的包导出是绝对不可取的。

Jackson 是一个在结构上已经模块化的库示例，由几个 JAR 组成。而其他库做出了不同的选择。例如，Google Guava 选择将其所有功能捆绑到一个 JAR 中。Guava 将许多独立



有用的部分聚合在一起，范围从可选集合实现到事件总线。但是，Guava 这么做完全是一种全无或全有的选择。Guava 维护者没有对该库进行模块化的主要原因是考虑到向后兼容性 (<https://github.com/google/guava/issues/605>)。未来的版本必须支持 Guava 作为一个整体。

创建一个表示整个库的聚合器模块是通过模块系统实现此功能的一种方法。在 5.4.1 节中已经简要地讨论了这个模式。对于 Guava，可能看起来如下所示：

```
module com.google.guava {
    requires transitive com.google.guava.collections;
    requires transitive com.google.guava.eventbus;
    requires transitive com.google.guava.io;
    // 等等
}
```

然后，每个单独的 Guava 模块都会导出相关的包。与前面一样，Guava 用户可以请求 `com.google.guava`，并且以传递的方式获取所有的 Guava 模块。隐式可读性确保用户可以访问由单个小型模块导出的所有 Guava 类型。或者，可以仅请求应用程序所需的单个模块。这是在开发时的易用性和较小的解析依赖关系图（在运行时占用更少的空间）之间常见的折中方案。

命名 Guava 模块

在这个假设的示例中，模块的名称为 `com.google.guava`。如果按照 10.2 节中所给出的建议（采用最长的公共包前缀），将会产生另一个模块名称：`com.google.common`。而该名称就是 Google Guava 团队在编写模块时已经确定的名字。一方面，使用该名称可以非常清楚地表明模块名称与所包含包之间的关系。另一方面，包命名方案毕竟不是那么好。

如果 Guava 项目中的模块名称不包含 Guava，那将是非常尴尬的事情。这也再次说明，选择一个好名字是很困难的。在这个过程中，围绕着责任和包所有权展开讨论是不可避免的。

当库包含单个大型 JAR 时，可以考虑在模块化时将其拆分。在许多情况下，库的可维护性越高，用户就可以更具体地了解所依赖的 API。通过一个聚合器模块，可以为那些想要一劳永逸的人提供向后兼容性。

尤其是当 API 的不同独立部分有不同的外部依赖关系时，对库进行模块化可以更好地帮助用户。他们只需请求所需 API 的个别部分，以避免不必要的依赖项，因此不必受累于强加给他们的 API 中不使用部分的依赖项。

作为一个具体的例子，回顾一下 `java.sql` 模块及其对 `java.xml` 的依赖关系（如 2.5



节所述)。存在该依赖关系的唯一原因是 SQLXML 接口。有多少 java.sql 模块的用户正在使用数据库的 XML 功能？可能并不是那么多。

不过，现在所有 java.sql 的使用者都可以在已解析的模块图中“免费”获取 java.xml。如果将 java.sql 拆分为 java.sql 和 java.sql.xml，则用户就可以进行选择。后一个模块包含 SQLXML 接口，执行 `requires transitive java.xml`（以及 java.sql）。此时，java.sql 本身不再需要 java.xml。对 XML 功能感兴趣的用户可以请求 java.sql.xml，而其他人可以请求 java.sql（不需要以模块图中的 java.xml 结束）。

因为上述模式需要将 SQLXML 放在自己的包中（不能将包拆分成多个模块），所以这不适用于 JDK。该模式更适用于已经在不同包中的 API。如果能把它脱离出来，那么根据外部依赖关系隔离模块可以极大地帮助库用户。

10.6.2 外部依赖关系

库之间的内部依赖关系可以在模块描述符中处理，甚至可以在生成初步模块描述符时由 `jdeps` 负责处理。那么对外部库的依赖关系应该如何处理呢？



在理想情况下，库没有外部依赖关系（框架却完全不同）。唉，我们不是生活在一个理想的世界里。

如果这些外部库是显式的 Java 模块，那么答案很简单：在库的模块描述符中添加 `requires (transitive)` 子句就足够了。

如果依赖项没有模块化又该怎么办呢？很多人会认为这没有问题，因为任何 JAR 都可以用作自动模块。虽然这是真的，但还是存在一个与命名有关的细微问题，在 10.2 节中已经谈到了这个问题。对于库模块描述符中的 `requires` 子句，需要一个模块名称。但是，自动模块的名称取决于 JAR 文件名，而 JAR 文件名不完全在我们的控制之下。日后，真正的模块名称可能会发生变化，从而导致使用库和外部依赖项的（目前）模块化版本的应用程序中出现模块解析问题。

这个问题没有万全的解决办法。只有当合理地确定模块名称是稳定的时候，才应该在外部依赖项上添加一个 `requires` 子句。确定模块名称是否稳定的方法之一是迫使外部依赖项的维护者使用清单中的 `Automatic-Module-Name` 头来声明模块名称。如你所见，这是一个相对较小且风险较低的变化。然后，可以使用这个稳定的名称安全地引用自动模块。或者，可以要求外部依赖项完全模块化，但这需要完成更多的工作。如果模



块名称不稳定，那么其他方法都可能以失败告终。



Maven Central 不鼓励发布指向没有稳定名称的自动模块的模块。库只需要具有 `Automatic-Module-Name` 清单条目的自动模块。

另一种技巧是使用多个库来管理外部依赖关系：*依赖阴影* (*dependency shading*)。其主要思想是通过将外部代码内联到库中来避免外部依赖。简而言之，外部依赖项的类文件被复制到库 JAR 中。为了防止原始外部依赖项也出现在类路径上时所发生的名称冲突，在内联过程中将重命名包。例如，来自 `org.apache.commons.lang3` 的类将会被重命名为 `com.javamodularity.mylibrary.org.apache.commons.lang3`。所有这些都是自动完成的，并通过后期处理字节码在构建时发生。这可以防止恶意软件包名称渗透到实际的源代码中。对于模块，阴影 (shading) 仍然是一个可行的选择。但是，建议仅用于库内部的依赖关系。不推荐导出阴影包，或者将已导出 API 中的阴影类型导出。

完成上述步骤之后，就可以控制库的内部和外部依赖关系。此时，库是模块或模块集合，针对所需支持的最低版本的 Java。但是，如果库实现可以使用新的 Java 特性，同时仍然能够在最低支持的 Java 版本上运行，岂不是更好？

10.7 针对多个 Java 版本

如果想要在不破坏向后兼容性的情况下在库实现中使用新的 Java API，一种方法是有选择性地使用它们。如 5.6 节所述，反射可用于定位新的平台 API (如果可用的话)。但不幸的是，这样做会导致代码变得脆弱且难以维护。而且，这种方法仅适用于使用新的平台 API。在库实现中使用新的语言功能仍然是不可能的。例如，在保持 Java 7 兼容性的情况下在库中使用 Lambda 表达式是不可能的。另一种方法是，维护和发布针对不同 Java 版本的同一个库的多个版本，该方法同样没有吸引力。

10.7.1 多版本 JAR

如果使用 Java 9，则可以引入一个新功能：多版本 JAR 文件。此功能允许将同一个类文件的不同版本打包到单个 JAR 中。相同类的这些不同版本可以针对不同的主要 Java 平台版本进行构建。在运行时，JVM 会为当前环境加载最适合的类版本。

需要注意的是，该功能是独立于模块系统的，尽管它与模块化 JAR 可以很好地一起工作。通过使用多版本 JAR，可以在库中使用当前平台的 API 和语言功能。而旧 Java 版本的用户仍然可以依靠同一个多版本 JAR 中以前的实现。

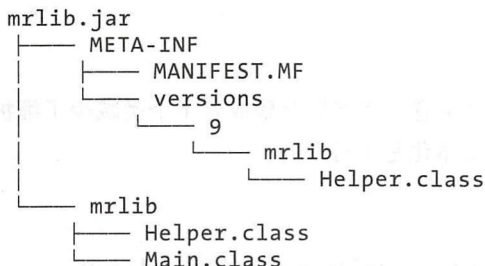


JAR 在符合特定布局并且其清单包含 `Multi-Release:true` 条目时启用了多版本。类的新版本需要位于 `META-INF/versions/<n>` 目录中，其中 `<n>` 对应于主要的 Java 平台版本号。不能专门为次要版本或补丁版本进行版本升级。



与所有清单条目一样，`Multi-Release:true` 条目周围不能有前导或尾随空格。条目的关键字和值不区分大小写。

下面显示了多版本 JAR 的内容 (👉 `chapter10/multirelease`)



这是一个带有两个顶级类文件的简单 JAR。`Helper` 类还有一个使用 `META-INF/versions/9` 下 Java 9 功能的替代版本。完全限定名称是完全一样的。从库用户的角度来看，该库只有一个版本，由 JAR 文件表示。多版本功能的内部使用不应违反用户期望。因此，所有类都应该在所有版本中具有完全相同的公共签名。请注意，Java 运行时并不对此进行检查，所以由开发人员和工具完成相关工作。

创建 `Helper` 类的 Java 9 特定版本有多种合理的原因。首先，类的原始实现可能使用了 Java 9 中已经删除或封装的 API。特定于 Java 9 的 `Helper` 版本可以使用 Java 9 中引入的替换 API，同时不用破坏早期 JDK 中所使用的实现。或者，`Helper` 类的 Java 9 版本可能会为了更快或更好地运行而使用新功能。

由于替代类文件位于 `META-INF` 目录下，因此早期的 JDK 将忽略它。但是，当在 JDK 9 上运行时，加载的是此类文件而不是顶级的 `Helper` 类。该机制在类路径和模块路径上都可以工作。JDK 9 中的所有类加载器都可以进行多版本 JAR 识别。由于在 JDK 9 中引入了多版本 JAR，因此 `versions` 目录下只能使用 9 及以上版本。任何早期的 JDK 都只能看到顶级类。

只需使用不同的 `--release` 设置编译不同的源就可以创建一个多版本 JAR：

```
javac --release 7 -d mrlib/7 src/<all top-level sources> ❶
javac --release 9 -d mrlib/9 src9/mrlib/Helper.java ❷
jar -cfe mrlib.jar src/META-INF/MANIFEST.MF -C mrlib/7 . ❸
```



```
jar -uf mrlib.jar --release 9 -C mrlib/9 . ④
```

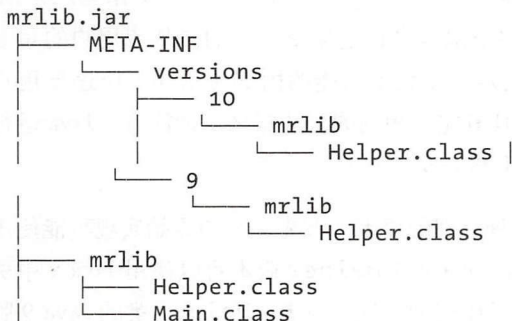
- ① 在所需的最低版本级别下编译所有常规源代码。
- ② 仅针对 Java 9 单独编译代码。
- ③ 使用正确的清单和顶级类创建一个 JAR 文件。
- ④ 使用新的 `--release` 标志更新 JAR 文件，将类文件放置到正确的 `META-INF/versions/9` 目录中。

在这种情况下，针对 Java 9 的特定 Helper 版本来自自己的 `src9` 目录。由此产生的 JAR 适用于 Java 7 及更高版本。只有在 Java 9 上运行时，才会加载针对 Java 9 编译的特定 Helper 版本。



尽量减少版本化类的数量是一个好主意。将差异分解成若干个类减少了维护的负担，但对 JAR 中的所有类进行版本化是不可取的。

在 Java 10 发布之后，可以使用针对该版本的特定 Helper 实现扩展 `mrlib` 库：



在 Java 8 及更低版本上运行这个多版本 JAR 与以前一样，使用顶级类。而在 Java 9 上运行时，则使用 `versions/9` 中的 Helper 类。同样，在 Java 10 上运行时，会加载与 `versions/10` 中的 Helper 最匹配的项。当前的 JVM 总是加载与 Java 运行时自身版本相匹配的该类的最新版本。资源遵守与类相同的规则。可以将不同 JDK 版本的特定资源放在 `versions` 目录中，并将按照相同的优先顺序加载它们。

`versions` 目录下的任何类都必须出现在顶层，但并不要求每个版本都有特定的实现。在前面的示例中，把 Helper 放在 `versions/9` 下是完全正确的。在 Java 9 上运行库意味着它回退到顶层实现，并且特定版本仅在 Java 10 及更高版本上使用。



10.7.2 模块化多版本 JAR

多版本 JAR 也可以是模块化的。只需在顶层添加模块描述符就可以了。如前所述，如果在 Java 9 之前的运行时上使用 JAR 时，*module-info.class* 将被忽略。也可以将模块描述符放在 *versions/9* 下。

这样做是否会引发具有不同版本的 *module-info.class* 的问题。的确，使用版本化的模块描述符是允许的，如 *versions/9* 下的模块描述符以及 *versions/10* 下的模块描述符。模块描述符之间所允许的差异应该尽可能小。这些差异不应导致 Java 版本之间可观察到的行为差异，就像普通类的不同版本必须具有相同的签名一样。

在实践中，下列规则适用于版本化模块描述符：

- 只有 *java.** 和 *jdk.** 模块上非传递性的 *requires* 子句可以有所不同。
- 无论服务类型如何，服务 *uses* 子句可能有所不同。

当在不同的 JDK 上使用多版本 JAR 时，服务的使用以及不同平台模块的内部依赖关系都会导致可观察到的差异。不允许对版本之间的模块描述符进行任何其他更改。如果需要添加（或删除）*requires transitive* 子句，则模块的 API 将发生更改。这超出了多版本 JAR 支持的范围。在这种情况下，就生成了整个库的新版本。

如果你是库的维护者，那么需要完成的工作大大减少。首先确定一个模块名称，并使用 *Automatic-Module-Name* 声明。用户可以将你的库作为自动模块来使用，并采取下一步措施对库进行真正的模块化。最后，多版本 JAR 既降低了在库实现中使用 Java 9 功能的障碍，同时又保持与早期 Java 版本的向后兼容性。



模块化开发工具

比特币的模块化开发工具，旨在为开发者提供一个灵活、可扩展的框架，以构建各种区块链应用。这些工具通常包括钱包、节点、交易所和支付网关等。模块化设计允许开发者根据需要组合不同的组件，从而快速开发和部署定制化的区块链解决方案。此外，模块化开发工具还通常提供丰富的文档和教程，以帮助开发者更好地理解和使用这些工具。

在比特币生态系统中，模块化开发工具扮演着至关重要的角色。它们不仅降低了区块链开发的门槛，还促进了创新和竞争。通过利用这些工具，开发者可以更专注于业务逻辑的实现，而无需担心底层基础设施的复杂性。同时，模块化设计也提高了系统的可维护性和可扩展性，使得开发者能够轻松应对不断变化的市场需求和技术挑战。

1.1 Apache Maven

Apache Maven 是一个流行的 Java 构建工具，它使用 XML 配置文件来管理项目的构建和依赖。Maven 提供了统一的项目模型，使得项目结构更加清晰和一致。此外，Maven 还支持依赖管理，可以自动下载和更新项目所需的库文件。对于比特币开发者来说，Maven 可以用于构建和管理复杂的区块链应用，提高开发效率和项目质量。

在比特币开发中，Maven 可以用于构建各种类型的区块链应用，如钱包、节点和交易所等。通过配置 Maven 的配置文件，开发者可以定义项目的依赖关系、构建目标和打包方式。Maven 还提供了丰富的插件支持，可以满足不同的构建需求。此外，Maven 还支持多平台构建，使得开发者可以轻松地将应用部署到不同的操作系统和硬件平台上。

除了 Maven 之外，比特币生态系统中还有许多其他的模块化开发工具。这些工具通常具有不同的特点和优势，开发者可以根据具体的项目需求选择合适的工具。例如，一些工具可能专注于钱包开发，而另一些则可能专注于节点开发。通过深入了解这些工具的特点和优势，开发者可以更好地利用模块化开发工具，提高区块链应用的开发效率和性能。



第 11 章

构建工具和 IDE

本书主要是直接在命令行上使用 `java` 和 `javac`，这并不是如今大多数应用程序的构建方式。目前多数项目都是使用 `Maven` 或 `Gradle` 等工具构建的。这些构建工具可以处理各种问题，比如在编译期间管理类路径、依赖关系管理以及构建工件（如 `JAR` 文件）等。最重要的是，大多数开发人员都使用 IDE，比如 `Eclipse`、`IntelliJ IDEA` 或 `NetBeans`。IDE 提供了诸如代码完成、错误突出显示、重构和代码导航等功能，从而使开发更容易。

构建工具和 IDE 都需要知道在给定上下文中哪些类型可用。工具通常与类路径进行交互以完成此操作。随着 `Java` 模块系统的引入，该过程发生了很大的变化。类路径不再是控制哪些类型可用的（唯一）机制，工具现在也必须考虑使用模块路径。而且，还可以混合使用显式模块、类路径和自动模块。在编写本书的时候，工具生态系统仍在努力支持 `Java 9`。本章介绍了一些可用的工具，并讨论了它们如何支持（或可能在不久的将来支持）`Java` 模块系统。

11.1 Apache Maven

使用 `Maven` 构建单个模块项目是非常容易的一件事情。接下来将完成构建的相关步骤，但是不会介绍代码或配置。`GitHub` 存储库中包含一个示例，如果愿意可以尝试一下：
↳ [chapter11/single-module](#)。

将 `module-info.java` 放在项目的 `src/main/java` 目录中，那么 `Maven` 将正确设置编译器以使用模块源路径。即使依赖项还没有模块化，它们通常也是放在模块路径上，这意味着还没有模块化的依赖项总是作为自动模块来处理的。

这与第 8 章和第 9 章中所做的不同，这两章混合使用了类路径和模块路径。两种方法都很好，尽管将所有内容放在模块路径上在未来可能会带来一些隐藏的问题。现在，除了项目的输出是一个模块化的 `JAR` 之外，实际上看不到其他内容。`Maven` 很好地处理了这



个问题。

虽然表面上看不到太多东西，但内部却发生了很多事情。Apache Maven 现在必须考虑 Java 模块系统的规则。为了支持 Java 模块系统，Apache Maven 完成了如下重要的更改：

- 在编译期间使用模块路径。
- 支持将显式模块和自动模块的混合体作为依赖项。

有趣的是，上面所做的更改并没有包含任何有关将 POM 与 *module-info.java* 集成的内容，虽然 POM 中的依赖项和 *module-info.java* 中的 `requires` 之间存在明确的关系。这其实并不奇怪。不妨这样来想：Apache Maven 仅配置了模块路径和类路径。Java 编译器接受该配置并使用它来编译源代码（包括 *module-info.java*）。Apache Maven 取代了本书中所使用的 shell 脚本，但不会取代 Java 编译器。显然，两者都是需要的，但为什么 Maven 不为我们生成一个 *module-info.java* 呢？这与模块的命名有很大关系。

此时共有三个名称：

- 在 *module-info.java* 中定义的模块名称。
- *pom.xml* 中定义的 Maven 项目名称。
- 由 Maven 生成的 JAR 文件名称。

当从其他 *module-info.java* 文件中引用模块（例如，请求模块）时，将会使用模块名称。而在 Maven 级别上，向 *pom.xml* 中添加依赖项时使用 Maven 名称。最后，由 Maven 构建生成的 JAR 文件将用于部署。

在 Maven 中，模块名称（也称为 Maven 坐标）包含三个部分：`groupId:artifactId:version`。`groupId` 用于命名空间。在包含许多模块的项目中，`groupId` 在逻辑上将这些模块组合在一起。通常，`groupId` 是项目的反向域名。`artifactId` 是模块的名称。不幸的是，不同的项目使用不同的命名策略。有时项目名称包含在 `artifactId` 中，有时不包含。最后，Maven 模块被版本化。

Java 模块系统中的模块没有 `groupId`，也没有使用版本信息。对于公共模块，建议在模块名称中包含项目的反向域名。虽然，在任何情况下，Maven 模块名称、Java 模块系统模块名称以及 Maven 工件名称可能存在某些关联，但并不相同。图 11-1 描述了这一点。

添加依赖项还需要完成一个两步法。首先，需要将依赖项添加到表示该模块的 Maven 工件中，以 `groupname:artifactname:version` 的形式使用其 Apache Maven 坐标。这与在 Java 模块系统之前的系统中使用 Apache Maven 没有什么不同。其次，在



`module-info.java` 中将依赖项作为 `requires` 语句添加，以便代码可以使用该模块导出的类型。如果没有在 POM 文件中添加依赖项，则编译器将在 `requires` 语句上失败，因为找不到该模块。但如果没有将依赖项添加到 `module-info.java`，那么依赖项仍然不会被使用。

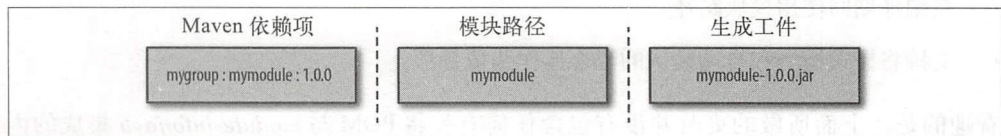


图 11-1: 工件命名

以上两处引用依赖项的事实说明模块名称不一定与 Apache Maven `group:artifact:version` 坐标相同。依赖项可能是也可能不是显式模块。如果找不到 `module-info.class`，则依赖项变成一个自动模块。这一切对用户是透明的：从 Apache Maven 用户的角度来看，使用显式模块或自动模块没有任何区别。

在下一节中将介绍一个多模块项目的完整代码示例。

11.1.1 多模块项目

在 Java 模块系统之前，通常使用 Apache Maven 创建多模块项目。即使没有 Java 模块系统带来的强约束，这也是模块化项目的一个良好的开始。多模块项目中的每个模块都有自己的 POM，这是一种 XML 格式的特定于 Maven 的构建描述符。在 POM 中配置了模块的依赖关系，包括对外部库的依赖以及对项目中其他模块的依赖。模块中使用的每种类型都必须是模块本身、JDK 或显式配置依赖项的一部分。从概念上讲，这与 Java 模块系统所看到的没有什么不同。

尽管多模块项目在 Apache Maven 中很常见，但你可能想知道 Java 9 之前的模块到底是什么样的。一个模块的表示方式为一个 JAR 文件，这是 Apache Maven 生成的常见工件。在编译时，Maven 配置类路径，使其仅包含被配置为依赖项的 JAR 文件。通过这种方式可以模拟 Java 模块系统在模块描述符中使用 `requires` 的类似行为。

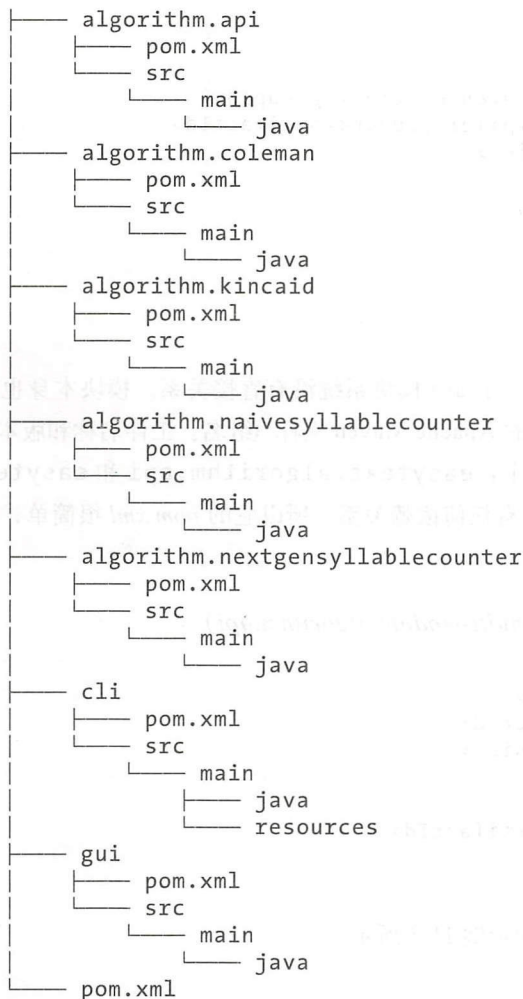
由于没有 Java 模块系统，因此 Apache Maven 不支持包的强封装。如果将依赖关系配置到另一个模块，则可以读取该模块中的每个类型。

11.1.2 使用 Apache Maven 创建 EasyText 示例

本节将 EasyText 示例迁移到 Maven，该示例是第 3 章所介绍的示例应用程序。代码本身没有改变，因此在这里不再一一列出。



首先，将 EasyText 目录结构更改为符合标准的 Apache Maven 目录结构。每个模块都有自己的目录，包含模块源文件（包括 `moduleinfo.java`）的 `src/main/java` 目录以及模块根目录下的 `pom.xml`。其中要注意项目根目录下的 `pom.xml`，这是一个父 POM，可以用一个命令编译所有的模块。该目录结构如下：



父 POM 包含对其子项目的引用，即实际的模块，其还将编译器插件配置为使用 Java 9。示例 11-1 是一个包含 `pom.xml` 文件最有趣部分的代码片段。

示例 11-1: `pom.xml` (↪ `chapter11/multi-module`)


```
<modules>
  <module>algorithm.api</module>
  <module>algorithm.coleman</module>
  <module>algorithm.kincaid</module>
```



```
<module>algorithm.naivesyllablecounter</module>
<module>algorithm.nextgensyllablecounter</module>
<module>gui</module>
<module>cli</module>
</modules>

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.1</version>
        <configuration>
          <release>9</release>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

请注意，modules 部分并不是新的，与 Java 模块系统没有直接关系。模块本身也有自己的 *pom.xml* 文件，其中指定了模块的 Apache Maven 坐标（组名、工件名称和版本）及其依赖关系。接下来看看其中的两个：*easytext.algorithm.api* 和 *easytext.algorithm.kincaid*。API 模块没有任何依赖关系，所以它的 *pom.xml* 很简单，如示例 11-2 所示。

示例 11-2: *pom.xml* ( *chapter11/multi-module/algorithm.api*)

```
<parent>
  <groupId>easytext</groupId>
  <artifactId>parent</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>

<artifactId>algorithm.api</artifactId>

<name>Algorithm API</name>
```

在其 *module-info.java* 中，模块定义如示例 11-3 所示。



示例 11-3: module-info.java (↪ chapter11/multi-module/algorithm.api)

```
module easytext.algorithm.api {  
    exports javamodularity.easytext.algorithm.api;  
}
```

请注意,从技术上讲,组/工件名称与模块的 *module-info.java* 中所指定的模块名称无关,它们可能完全不同。但不管怎样,只要记住当在 *pom.xml* 文件中创建一个依赖时,就需要使用 Apache Maven 坐标。当在 *module-info.java* 中 *requires* 一个模块时,使用在另一个模块的 *module-info.java* 中所指定的名称。Apache Maven 坐标在这个级别上不起任何作用。而且,还要注意目录名不要求与模块名相同。

运行程序,生成 *target/algorithm.api-1.0-SNAPSHOT.jar*。

接下来介绍一个使用了 *easytext.algorithm.api* 的模块。此时,必须在模块的 *pom.xml* 中添加一个依赖项,并在 *module-info.java* 中添加一个 *requires* 声明,如示例 11-4 所示。

示例 11-4: pom.xml (↪ chapter11/multi-module/algorithm.kincaid)

```
<groupId>easytext</groupId>  
<artifactId>algorithm.kincaid</artifactId>  
<packaging>jar</packaging>  
<version>1.0-SNAPSHOT</version>  
<name>algorithm.kincaid</name>  
  
<parent>  
    <groupId>easytext</groupId>  
    <artifactId>parent</artifactId>  
    <version>1.0-SNAPSHOT</version>  
</parent>  
  
<dependencies>  
    <dependency>  
        <groupId>easytext</groupId>  
        <artifactId>algorithm.api</artifactId>  
        <version>${project.version}</version>  
    </dependency>  
</dependencies>  
</project>
```

在 *module-info.java* 中,可以看到所期望的 *requires*, 如示例 11-5 所示。

示例 11-5: module-info.java (↪ chapter11/multi-module/algorithm.kincaid)

```
module easytext.algorithm.kincaid {  
    requires easytext.algorithm.api;
```




```
    provides javamodularity.easytext.algorithm.api.Analyzer
           with javamodularity.easytext.algorithm.kincaid.KincaidAnalyzer;

    uses javamodularity.easytext.algorithm.api.SyllableCounter;
}
```

删除 *pom.xml* 中的依赖项或 *module-info.java* 中的 `requires` 都会导致编译错误。具体的原因略微不同。从 *pom.xml* 中删除依赖项会导致以下错误：

```
module not found: easytext.algorithm.api
```

删除 `requires` 语句将导致以下错误：

```
package javamodularity.easytext.algorithm.api is not visible
```

如前所述，Apache Maven 只配置模块路径。Apache Maven 依赖项的工作方式与 *module-info.java* 中的 `requires` 语句不同。

如果以前使用过 Apache Maven，那么 *pom.xml* 文件应该看起来很熟悉。Apache Maven 在使用 Java 模块系统时不需要新的语法或配置。

该示例表明，一个正确模块化的 Apache Maven 应用程序很容易迁移到 Java 模块系统。从本质上来说，唯一需要做的是添加 *module-info.java* 文件。而作为回报，其将得到封装和更强大的运行时模型。

11.1.3 使用 Apache Maven 运行模块化的应用程序

虽然示例项目被配置为使用 Apache Maven 进行构建，但如何运行它呢？Maven 只是一个构建工具，在运行时没有任何作用。Maven 构建了想要运行的工件，但是最后仍然需要配置 Java 运行时，以便运行正确的模块路径和类路径。

虽然手动配置模块路径比类路径要容易得多，但它仍然是重复的工作，因为相关信息已经在 *pom.xml* 文件中了。Maven 有一个帮助完成该过程的 `exec` 插件。请记住，该插件仅配置了模块路径，在运行时并不存在。模块路径将根据 *pom.xml* 中列出的依赖项进行配置。只需要配置一个模块和主类来执行插件即可。示例 11-6 提供了 CLI 模块的配置。

示例 11-6: *pom.xml* (↪ *chapter11/multi-module/algorithm.cli*)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.6.0</version>
      <executions>
        <execution>
          <goals>
```



```
        <goal>exec</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <executable>${JAVA_HOME}/bin/java</executable>
    <arguments>
      <argument>--module-path</argument>
      <modulepath/>
      <argument>--module</argument>
      <argument>easytext.cli/javamodularity.easytext.cli.Main</argument>
      <argument>${easytext.file}</argument>
    </arguments>
  </configuration>
</plugin>
</plugins>
</build>
```

通过使用 `exec` 命令来启动应用程序，从而运行该插件：

```
mvn exec:exec
```

11.2 Gradle

不幸的是，在编写本书时 Gradle 还没有提供官方的 Java 模块系统支持。即使提供支持，其方式可能会类似于 Maven。在模块化代码方面，Gradle 已经非常优秀了。对多模块项目的支持是很好的，而这恰恰是准备使用 Java 模块系统的一个很好的开端。

11.3 IDE

即使在 Java 9 正式发布之前，IntelliJ、Eclipse 和 NetBeans 等 IDE 也都支持 Java 模块系统。支持 Java 模块系统的 IDE 的最重要特性是理解 `module-info.java` 文件中的 `requires` 和 `exports`。这些关键字控制模块可用的类型，IDE 应该使用它来完成语法，指出错误，并提供模块依赖关系的建议。上面所提到的三个 IDE 都支持这一点。这与 Java 模块映射到 IDE 中的项目、工作区和模块的方式密切相关。每个 IDE 都有自己的结构，而 Java 模块必须映射到这个结构。

在 Eclipse 中，每个项目都代表一个模块，假设它包含一个 `module-info.java`。一如既往，项目被分组在一个工作区中。IntelliJ 和 NetBeans 都已经有了自己的模块概念，现在可以直接映射到 Java 模块系统模块。

上述三个 IDE 也支持编辑 `module-info.java` 文件，包括模块名称的错误突出显示和语法完成。一些 IDE 甚至支持基于模块描述符的可视化模块图显示。

虽然这一切对于用户来说应该是透明的，但是在管理项目结构时，IDE 中显然存在一



些重复。IDE 有自己的模块内部表示 (比如说 Eclipse 中的项目)。过去, 该模型可以与 Maven 或 Gradle 等外部模型同步。现在, Java 模块系统模型是模块表示的第三级。虽然这些工具掩盖了这个事实, 但是当深入挖掘时, 仍然会变得有点混乱。图 11-2 解释了如何使用 Maven 和 `module-info.java` 在 IDE 中配置项目。对于未来的 Gradle 也是如此。

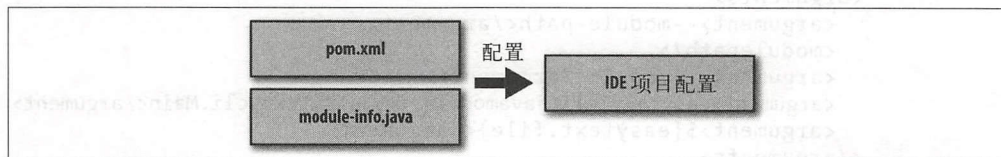


图 11-2: 在 IDE 中配置可见性

在将来的工具版本中, 会更好地支持重构、提示以及向模块的迁移。



测试模块

构建模块化代码库也涉及测试，Java 社区一直在提倡强大的自动化测试文化。单元测试在 Java 软件开发中起着重要的作用。

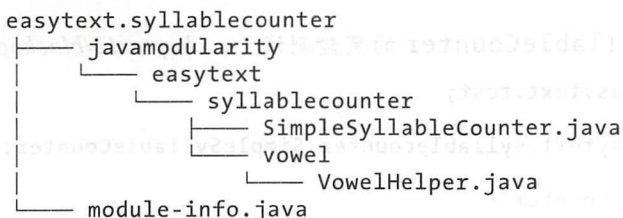
模块系统对现有测试实践有什么影响呢？我们希望能够测试模块内的代码。在本章中，将会介绍两种常见的测试方案：

- 1) **黑盒测试 (blackbox testing)**：从外部测试模块。黑盒测试主要使用模块的公共 API，而不需要了解模块内部情况（因此，盒子是不透明的），可以一次测试一个模块或多个模块。因此，也可以将这些测试描述为模块的集成测试。
- 2) **白盒测试 (whitebox testing)**：从内部测试模块。白盒测试不是采用外部视图，而是假定了解模块的内部。这些测试通常是单元测试，单独测试一个类或一个方法。

虽然还有其他的测试方案，但以上两种方案涵盖了大量现行的做法。黑盒测试在测试中受到更多限制，但是由于它们使用公共 API，所以也更加稳定。而白盒测试可以更容易地测试内部细节，但需要承担更多的维护风险。

本章的重点是突出介绍测试和模块系统之间的相互影响。接下来将使用构建工具和 IDE 完成本章中所描述的许多细节。不过，重要的是要了解测试方案如何与模块系统配合使用。

本章将会使用如下所示的待测试模块：





easytext.syllablecounter 的模块描述符如下所示:

```
module easytext.syllablecounter {
    exports javamodularity.easytext.syllablecounter;
}
```

包含 VowelHelper 的包不会被导出, 而 SimpleSyllableCounter 位于导出的包中。在内部, SimpleSyllableCounter 使用 VowelHelper 来实现音节计数算法。从黑盒测试转到白盒测试时, 这个区别就显得非常重要了。

在下面的章节中, 将看看如何对模块运行两种类型的测试。

12.1 黑盒测试

假设对 easytext.syllablecounter 模块进行测试。此时并不是对所有内部细节进行单元测试, 而是测试模块 API 的功能行为。在这种情况下, 就意味着测试 SimpleSyllableCounter 公开的公共 API。它有一个公共的方法: countSyllables。

最简单的方法是创建另一个需要 easytext.syllablecounter 的模块, 如图 12-1 所示, 也可以将模块及其测试放在类路径中。此时并不使用后一种方法, 因为我们将 easytext.syllablecounter 作为一个模块进行测试 (即包括带有 requires 和 uses 子句的模块描述符), 而不仅仅是作为类路径上的一些代码。在 12.3 节中将会测试模块内部, 并介绍类路径测试方法。

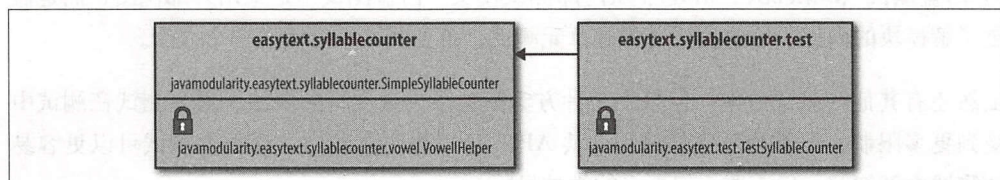


图 12-1: 使用单独的 easytext.syllablecounter.test 模块测试 easytext.syllablecounter 模块

为了慢慢深入, 将首先创建一个没有测试框架的测试。然后, 调整测试以使用流行的单元测试框架 JUnit。示例 12-1 给出了执行测试的代码, 其中使用了标准的 Java 断言来进行验证。

示例 12-1: 针对 SimpleSyllableCounter 的黑盒测试 (↪ chapter12/blackbox)

```
package javamodularity.easytext.test;

import javamodularity.easytext.syllablecounter.SimpleSyllableCounter;

public class TestSyllableCounter {
```



```
public static void main(String... args) {
    SimpleSyllableCounter sc = new SimpleSyllableCounter();

    assert sc.countSyllables("Bike") == 1;
    assert sc.countSyllables("Motor") == 2;
    assert sc.countSyllables("Bicycle") == 3;
}
}
```

上述代码非常简单：`main` 方法实例化公共导出的 `SimpleSyllableCounter` 类，并通过使用断言来验证其行为。测试类被放置在它自己的模块中，且具有以下描述符：

```
module easytext.syllablecounter.test {
    requires easytext.syllablecounter;
}
```

然后，像往常一样编译和运行代码。假设待测模块已经被编译到 `out` 目录中：

```
$ javac --module-path out \ ❶
--module-source-path src-test -d out-test -m easytext.syllablecounter.test
$ java -ea --module-path out:out-test \ ❷
-m easytext.syllablecounter.test/javamodularity.easytext.test.
TestSyllableCounter ❸
Exception in thread "main" java.lang.AssertionError
    at easytext.syllablecounter.test/javamodularity.easytext.test.
    TestSyllableCounter.main(TestSyllableCounter.java:12)
```

- ❶ 使用模块路径上的待测试模块编译测试。
- ❷ 在启用断言的情况下 (`-ea`) 运行，并使用模块路径上的待测试模块和测试模块。
- ❸ 启动测试模块中包含 `main` 方法的类。

此时抛出了 `AssertionError`，因为“幼稚的”音节计数算法在单词 *Bicycle* 上卡住了。这很好，因为这意味着正在进行黑盒测试。在开始介绍测试框架并运行测试之前，先回忆一下黑盒测试方法。

以这种方式测试模块有几个优点，但也有一些缺点。黑盒测试的一个优点是可以在自然环境下测试模块。在测试模块时，就像其他模块在应用程序中使用它一样。例如，如果待测模块提供了服务，则也可以对服务进行测试。只需在测试模块的模块描述符中添加一个使用约束，然后在测试中使用 `ServiceLoader` 加载服务。

另一方面，以黑盒测试方式测试模块意味着只能对导出的部分进行直接测试，而无法对封装类进行测试，如 `VowelHelper` 类。同时，也不能访问 `SimpleSyllableCounter` 的任何非公共部分。



例如，可以使用 `--add-exports` 或 `--add-opens` 标志来运行测试，以便测试模块可以访问封装部分。

另一个限制是测试类与待测试类需要位于不同的包中。对于 Java 中的单元测试，习惯上把测试类放在不同的源文件夹中，但是在相同的包名下，这样设置的目的是测试包私有 (*package-private*) 的元素 (例如，没有 `public` 修饰符的类)。在使用类路径的情况下，这样做是没有问题的；运行测试时，包就会合并。但是，包含相同包的两个模块不能在引导层中加载 (如 6.3.2 节所述)。

最后，必须满足待测模块和测试模块自身所有的依赖关系。如果 `easytext.syllablecounter` 需要其他模块，那么也需要将这些模块放在模块路径中。当然，在这些情况下可以创建模拟模块 (*mock module*)。可以创建一个具有相同名称的新模块 (仅包含了运行测试所需的代码)，而不是将实际的模块依赖项放在模块路径上。是否要这样做取决于测试的范围。使用实际模块运行测试更像是一个集成测试，而使用模拟模块运行测试则提供了更多的隔离和控制。

12.2 使用 JUnit 进行黑盒测试

此时，可以由单独的测试模块对 `easytext.syllablecounter` 运行测试代码。然而，通过在主方法中使用普通的断言来编写测试代码并不是所期望的。为了解决这个问题，需要重写测试代码，以便使用 JUnit 4，如示例 12-2 所示。

示例 12-2: 针对 `SimpleSyllableCounter` 的 JUnit 测试 (↪ *chapter12/blackbox*)

```
package javamodularity.easytext.test;

import org.junit.Test;
import javamodularity.easytext.syllablecounter.SimpleSyllableCounter;

import static org.junit.Assert.assertEquals;

public class JUnitTestSyllableCounter {

    private SimpleSyllableCounter counter = new SimpleSyllableCounter();

    @Test
    public void testSyllableCounter() {
        assertEquals(1, counter.countSyllables("Bike"));
        assertEquals(2, counter.countSyllables("Motor"));
        assertEquals(3, counter.countSyllables("Bicycle"));
    }
}
```



现在，测试模块依赖于JUnit。在运行时，JUnit 测试运行器将以反射的方式加载测试类来执行单元测试方法。为此，必须导出或开放测试包。一个开放式模块就足够了：

```
open module easytext.syllablecounter.junit {
    requires easytext.syllablecounter;
    requires junit;
}
```

图 12-2 显示了将 JUnit 添加到组合中的新情况。

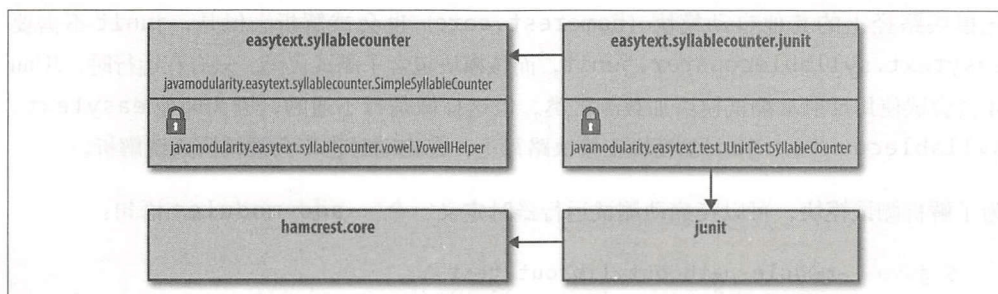


图 12-2: `easytext.syllablecounter.junit` 依赖 (自动) 模块 `junit`

为此，必须将 JUnit JAR (以及 Hamcrest 依赖项) 放到 `lib` 文件夹中：

```
lib
├── hamcrest-core-1.3.jar
└── junit-4.12.jar
```

然后，将 `lib` 文件夹放在模块路径上，从而将 JUnit 用作自动模块。派生模块名称是 `junit` 和 `hamcrest.core`。

与以前一样，假设 `out` 文件夹中有 `easytext.syllablecounter` 模块：

```
$ javac --module-path out:lib \ ❶
    --module-source-path src-test -d out-test -m easytext.
syllablecounter.junit
$ java --module-path out:lib:out-test \
    -m junit/org.junit.runner.JUnitCore \ ❷
    javamodularity.easytext.test.JUnitTestSyllableCounter
```

```
JUnit version 4.12
```

```
.E
Time: 0,002
There was 1 failure:
1) initializationError(org.junit.runner.JUnitCommandLineParseResult)
java.lang.IllegalArgumentException: Could not find class
[javamodularity.easytext.test.JUnitTestSyllableCounter]
```

❶ 使用模块路径中的待测模块和 JUnit 来编译测试代码。



② 从 JUnit (自动) 模块启动 JUnitCore 测试运行器。

运行测试是使用 JUnitCore 运行器类 (JUnit 的一部分) 完成的。这是一个简单的基于控制台的测试运行器, 需要提供测试类的类名作为命令行参数。但不幸的是, 运行单元测试后却得到了一个意想不到的异常: JUnit 找不到 JUnitTestSyllableCounter 类。为什么找不到测试类? 此时需要开放模块, 以便 JUnit 可以在运行时访问它。

问题出在测试模块从未被解析。JUnit 被用作启动运行器的根模块。由于它是自动模块, 因此模块路径上的其他自动模块 (hamcrest.core) 也会被解析。但是, junit 不需要 easytext.syllablecounter.junit, 而该模块包含了测试代码。只有在运行时, JUnit 才会尝试使用反射从测试模块加载测试类。但这样做是行不通的, 因为即使 easytext.syllablecounter.junit 模块在模块路径上, 在启动时也不会被模块系统解析。

为了解析测试模块, 可以在启动测试运行器时定义一个 --add-modules 语句:

```
$ java --module-path out:lib:out-test \  
  --add-modules easytext.syllablecounter.junit \  
  -m junit/org.junit.runner.JUnitCore \  
  javamodularity.easytext.test.JUnitTestSyllableCounter  
  
JUnit version 4.12  
.E  
Time: 0,005  
There was 1 failure:  
1) testSyllableCounter(javamodularity.easytext.test.JUnitTestSyllableCounter)  
java.lang.AssertionError: expected:<3> but was:<1>
```

现在得到了一个合法的测试失败提示, 这意味着 JUnit 能够运行测试类。

现在, 我们已经将黑盒测试扩展为使用外部测试框架。因为 JUnit 4.12 没有模块化, 所以它必须被用作自动模块。只有在测试模块被解析且测试类在运行时可访问时, 通过 JUnit 运行测试才是可能的。

12.3 白盒测试

现在, 如果想要测试 VowelHelper, 应该怎么做呢? 它是一个公共但非导出类, 带有一个公共方法 isVowel 以及包私有的方法 getVowels。对该类进行单元测试意味着从黑盒测试切换到白盒测试。

访问封装的 VowelHelper 类是必要的。另外, 在测试包私有的功能时, 需要测试类位于同一个包中。如何在模块化设置中获得这些功能呢? 使用 --add-exports 或 --add-opens 可以在一定程度上公开用于测试的类型。如果测试类需要与被测试类处于同一个包中, 那么测试模块与待测试模块之间会发生包冲突。



解决这个问题主要有两种方法，但哪一种方法在实践中是最好的还有待观察。此外，构建工具和 IDE 决定了使用哪种方法。接下来研究这两种方法，以便对底层机制有一个直观的了解：

- 使用类路径进行测试。
- 通过注入测试来修补模块。

第一种方法是最直接的，因为需要使用该方法的事情经常发生：

```
package javamodularity.easytext.syllablecounter.vowel;

import org.junit.Test;
import javamodularity.easytext.syllablecounter.vowel.VowelHelper;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class JUnitTestVowelHelper {

    @Test
    public void testIsVowel() {
        assertTrue(VowelHelper.isVowel('e'));
    }

    @Test
    public void testGetVowels() {
        assertEquals(5, VowelHelper.getVowels().size());
    }

}
```

可以在已编译模块中测试封装的代码，而不是在测试时将其作为模块处理。放在类路径上的模块的行为就好像它们没有模块描述符一样。此外，类路径上 JAR 之间的拆分包不会造成任何问题。如果测试类也是在模块之外编译的，那么事情就非常简单了：

```
$ javac -cp lib/junit-4.12.jar:out/easytext.syllablecounter \
-d out-test $(find . -name '*.java')

$ java -cp lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar:\
out/easytext.syllablecounter:out-test \
org.junit.runner.JUnitCore \
javamodularity.easytext.syllablecounter.vowel.
JUnitTestVowelHelper
JUnit version 4.12
..
Time: 0,004

OK (2 tests)
```

当然，使用类路径意味着无法从模块的自动解析中受益。模块描述符中声明的依赖项没



有被解析，因为类路径上的模块表现为常规的非模块 JAR。类路径需要手动构建。而且，被测模块的模块描述符中任何服务的 `provides/uses` 子句都被忽略。所以，即使基于类路径的测试方法可以使用，也存在几个缺点。最终，将一个模块作为模块进行测试似乎更好，而不是将其视为非模块 JAR。

有一种方法既可以保持模块结构的完整，又可以在同一个包中创建测试类。通过使用称为 *模块修补 (module patching)* 的功能，可以将新类添加到现有模块中。

在同一模块和包中创建白盒单元测试的过程如下所示。首先，需要使用 `--patch-module` 标志编译测试类：

```
$ javac --patch-module easytext.syllablecounter=src-test \ ❶  
    --module-path lib:out \  
    --add-modules junit \ ❷  
    --add-reads easytext.syllablecounter=junit \ ❸  
    -d out-test $(find src-test -name '*.java')
```

❶ 测试源代码被编译，就好像它们是 `easytext.syllablecounter` 模块的一部分。请注意，这些测试代码没有模块描述符。

❷ 由于没有测试模块描述符以表达需要 `junit`，因此必须显式添加。

❸ 随着测试类被添加到 `easytext.syllablecounter`，该模块现在必须读取 `junit`。因为原始模块描述符中没有 `requires` 子句，所以 `--add-reads` 是必需的。

通过修补模块，可以将单元测试类编译为已编译 `easytext.syllablecounter` 模块的一部分。由于测试代码位于同一个包中，因此可以调用包私有的 `getVowels` 方法。

动态地使测试类成为 `easytext.syllablecounter` 模块的一部分会带来一些挑战。必须采取措施确保模块可以读取 `junit`。在运行时，`junit` 必须能够访问非导出包中的测试类。这样一来就会导致以下 Java 调用：

```
$ java --patch-module easytext.syllablecounter=out-test \ ❶  
    --add-reads easytext.syllablecounter=junit \ ❷  
    --add-opens \ ❸  
    easytext.syllablecounter/javamodularity.easytext.syllablecounter.  
    vowel=junit \  
    --module-path lib:out \  
    --add-modules easytext.syllablecounter \ ❹  
    -m junit/org.junit.runner.JUnitCore \  
    javamodularity.easytext.syllablecounter.vowel.JUnitTestVowelHelper  
  
JUnit version 4.12  
..  
Time: 0,004  
  
OK (2 tests)
```



- ① 在运行时，必须使用已编译测试类来修补模块。
- ② 在编译期间，模块需要读取 `junit`（在模块描述符中并没有表示 `junit`）。
- ③ 因为 `junit` 以反射的方式实例化 `JUnitTextVowelHelper`，所以其包含的包必须是开放的或导出到 `junit`。
- ④ 同以前一样，`junit` 是初始模块，不需要 `easytext.syllablecounter`，因此必须显式添加。

与编译器调用不同，不需要使用 `--add-modules junit`。JUnit 作为根模块运行，因此已经解析了。

在图 12-3 中，可以看到如何修补一个模块以运行单元测试。

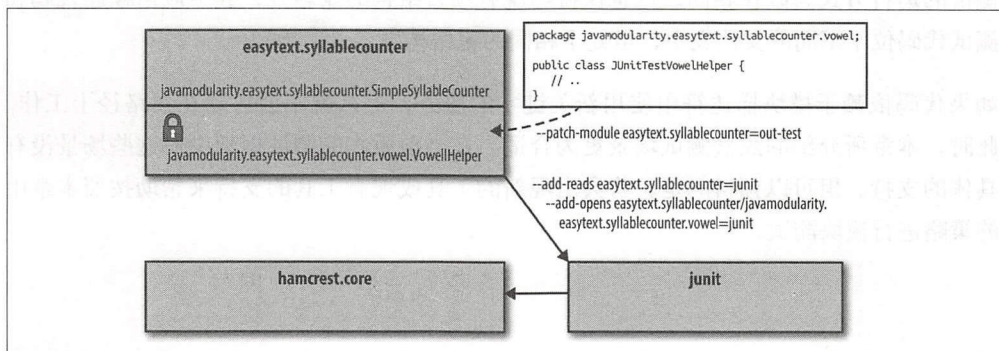


图 12-3：修补到 `easytext.syllablecounter` 模块的 `JUnitTextVowelHelper` 类（与 `VowelHelper` 位于同一包中）

第二种方法有很多可移动部件。有利的一面是，在测试时可以考虑模块的所有功能。在解析模块的依赖项的同时也会考虑 `uses/provides` 子句。而不利的一面是，设置所有正确的命令行标志似乎需要很多技巧。尽管可以解释为什么所有事情都需要这样做，但仍然需要付出很多努力。

在实践中，为单元测试场景设置所有复杂的细节不应该由开发人员来决定。构建工具和开发环境针对大多数流行的测试框架提供了自己的测试运行器，这些运行器应该负责建立环境，以便将测试代码自动修补到模块中。

出于其他原因而修补模块

除了运行单元测试以外，在其他一些场景中使用修补模块也是很方便的。例如，出于调试目的，可能希望添加或替换现有模块中的类。类和资源都可以修补到模块中，唯一不能被替换的是模块描述符。



`--patch-module` 标志也适用于平台模块。所以，从技术上讲，可以（重新）在 `java.base` 模块的 `java.lang` 包中放置类。模块修补将替换 `-Xbootclasspath:/ p`，这是在 JDK 9 中已删除的功能。

不鼓励在生产场景中使用模块修补功能。

12.4 测试工具

诸如 JUnit 和 TestNG 之类的测试框架得到了 IDE 以及构建工具的很好支持。通常，它们用于编写和运行白盒单元测试。即使代码包含模块描述符，大多数工具也都使用类路径来运行测试。这样一来，就像强封装一样，模块描述符基本上被忽略了。正因如此，测试的运行方式与以往相同。这也保持与现有项目结构的兼容性，其中应用程序代码和测试代码位于不同的文件夹中，但处于相同的包结构中。

如果代码依赖于模块描述符中使用新关键字的服务，那么就不会自动在类路径上工作。此时，本章所介绍的黑盒测试场景更为合适。在当前版本的测试框架中对这些场景没有具体的支持。但可以预料的是，将会出现新的工具或现有工具的支持来帮助按照本章中的策略进行模块测试。



使用自定义运行时映像进行缩减

到目前为止，已经学习了使用模块化应用程序所需的工具和流程，接下来可以进行进一步的探索。在 3.1.7 节中，介绍了如何为特定应用程序创建运行时映像。只有运行应用程序所需的模块才会成为映像的一部分。通过使用模块中的显式依赖信息，可以使用 jlink 自动生成最小的运行时映像。

创建自定义运行时映像有以下几个原因：

- **易用性**：jlink 提供了应用程序和 JVM 的一个独立分发。
- **减少占用空间**：只有应用程序使用的模块才会链接到运行时映像。
- **性能**：由于链接时优化的代价过高或不可能，因此自定义运行时可能会更快地运行。
- **安全性**：在自定义运行时映像中只需最少的平台模块，因此攻击面更小。

尽管创建自定义运行时映像是一个可选步骤，但由此产生的运行速度更快的二进制分发是创建自定义运行时映像一个令人信服的动机——尤其是当应用程序面向资源受限设备（如嵌入式系统）时或者当它们在云端运行时（此时，一切都被计量）。将自定义运行时映像放入 Docker 容器是创建资源高效的云部署的好方法。



目前很多举措正在改善 Java 对容器的支持。例如，OpenJDK 9 提供了一个 Alpine Linux 端口。在这个简约的 Linux 发行版之上运行 JDK 或自定义运行时映像是减少部署空间的另一种方法。

将应用程序与 Java 运行时一起分发的另一个优点是：已经安装的 Java 版本与应用程序需要版本之间不再有更多的不匹配。传统上，在运行 Java 应用程序之前必须安装 Java Runtime Environment (JRE) 或 Java Development Kit (JDK)。



JRE 是 JDK 的一个子集，用于运行 Java 应用程序，而不是开发 Java 应用程序。它始终作为一个单独的下载提供给最终用户。

自定义运行时映像是完全独立的。它将应用程序模块与 JVM 以及执行应用程序所需的一切捆绑在一起，而不需要其他的 Java 安装 (JDK/JRE)。例如，分发基于 JavaFX 的桌面应用程序会变得更加容易。通过创建自定义运行时映像，可以提供包含应用程序以及运行时环境的单个下载。另一方面，映像不可移植，因为它的目标是特定的操作系统和体系结构。在 13.8 节中，将讨论如何为不同的目标系统创建映像。

接下来深入了解一下 jlink 的能力。在介绍该工具本身之前，首先讨论一下链接以及它如何为 Java 应用程序开辟新的可能性。

13.1 静态链接和动态链接

可以将创建自定义运行时映像描述为模块的一种静态链接形式。链接 (*linking*) 是将已编译工件组合成高效可执行形式的过程。传统上，Java 一直在类级别上采用动态链接 (*dynamic linking*)。类在运行时被延迟加载，并且必要时在任何时间点进行动态链接。然后，虚拟机的即时 (*just-in-time*, JIT) 编译器负责在运行时将其编译为本机代码。在这个过程中，JVM 对生成的类集进行优化。虽然这个模型可以提供很大的灵活性，但是在更静态的情况下，简单的优化比在动态的场景中更难 (甚至不可能) 应用。

不同语言采取了不同的折中方案。比如，Go 语言偏向于将所有代码静态链接到一个二进制文件中。在 C++ 中，可以选择静态或动态链接。随着模块系统和 jlink 的引入，现在在 Java 中也有了这个选择。类仍然动态加载并链接到自定义运行时映像中。但是，可以静态预先确定从可用模块中加载哪些类。

静态链接的一个优点是可以提前对整个应用程序进行优化。实际上，这意味着从整个应用程序为出发点进行考虑，可以跨类和模块边界应用优化。这是可能的，因为通过模块系统，对整个应用程序的实际需求提前有了了解。所有模块都是已知的，从根模块 (应用程序的入口点) 到库，然后到所需的平台模块。已解析模块图显示了整个应用程序。

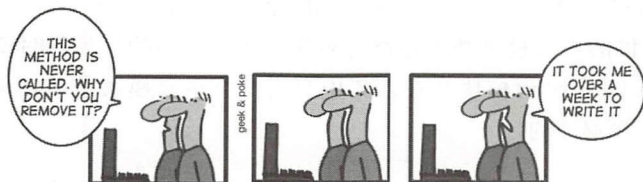


链接与提前 (AOT) 编译不同。用 jlink 生成的运行时映像仍然由字节码组成，而不是本地代码。

整个程序优化的示例包括死代码消除 (*dead-code elimination*)、常量合并 (*constant folding*) 和内联。虽然介绍这些优化已经超出了本书的范围，但幸运的是很多文献都提



供了相关内容。⊖



许多优化的适用性和有效性取决于相关代码同时可用的假设。链接阶段就是这个假设成立的时候，而 `jlink` 就是完成优化的工具。

13.2 使用 `jlink`

在 3.1.7 节中，创建了一个由 `helloworld` 模块和 `java.base` 组成的自定义运行时映像。在第 3 章的末尾，创建了一个更有趣的应用程序——`EasyText`，实现了多重分析和 CLI/GUI 前端。提醒一下，在完整的 JDK 上运行 GUI 前端是通过设置正确的模块路径并启动正确的模块来实现的：

```
$ java -p mods -m easytext.gui
```

假设 `mods` 目录包含用于 `EasyText` 的模块化 JAR，从而在运行时出现如图 13-1 所示的情况。

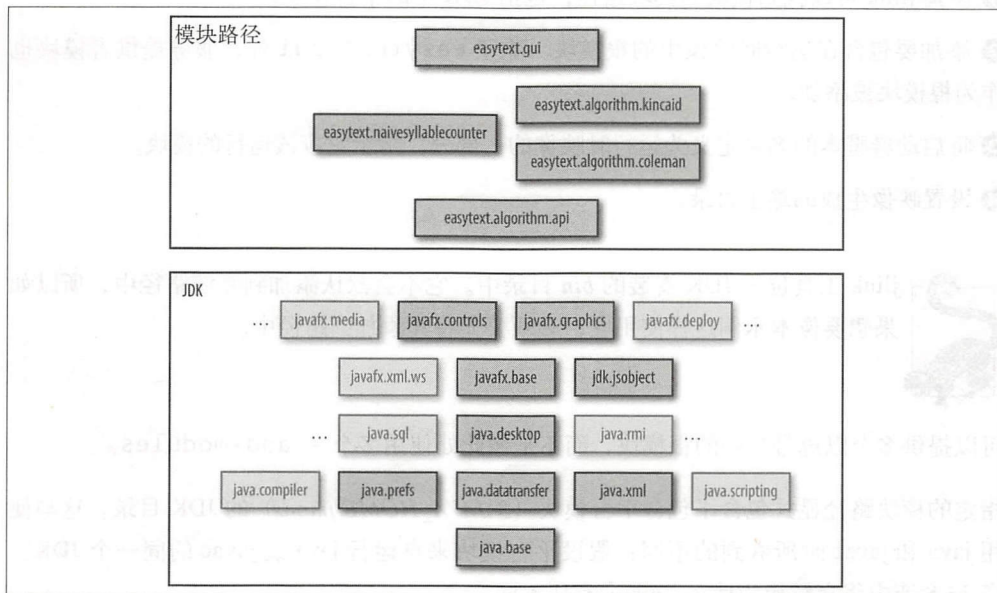


图 13-1: 当运行 `easytext.gui` 时在运行时解析模块。JDK 中提供了灰色模块（但尚未解析）

⊖ 可以在 Craig Chambers 等人所编写的《Whole-Program Optimization of Object-Oriented Languages》一书中找到与语言无关的相关介绍。



在 JVM 启动时，创建了该模块图。从根模块 `easytext.gui` 开始，所有依赖项都被递归解析。应用程序模块和平台模块都是解析模块图的一部分。但是，如图 13-1 所示，JDK 中所提供的平台模块并不是都是应用程序所必需的。灰色的模块只是冰山一角，因为大约有 90 个平台模块存在。其中只有 9 个平台模块是使用 JavaFX UI 运行 `easytext.gui` 所必需的。

接下来通过为 `EasyText` 创建一个自定义运行时映像来摆脱上述沉重负担。所要做的第一选择是确定哪些模块应该成为映像的一部分？是 GUI，或是 CLI，还是两者兼而有之？可以针对应用程序的不同用户组创建多个映像。针对这个问题没有常见的正确或错误的答案。链接显式地将模块组合成一个连贯的整体。

现在，为 `EasyText` 的 GUI 版本创建一个运行时映像。此时使用 `easytext.gui` 作为根模块调用 `jlink`：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods          \ ❶  
      --add-modules easytext.gui                      \ ❷  
      --add-modules easytext.algorithm.coleman        \  
      --add-modules easytext.algorithm.kincaid        \  
      --add-modules easytext.algorithm.naivesyllablecounter \  
      --launcher easytext=easytext.gui                \ ❸  
      --output image ❹
```

- ❶ 设置 `jlink` 可以找到模块的模块路径，包括 JDK 中的平台模块。
- ❷ 添加要包含在运行时映像中的根模块。除了 `easytext.gui` 外，服务提供者模块也作为根模块被添加。
- ❸ 将启动器脚本的名称定义为运行时映像的一部分，指示它应该运行的模块。
- ❹ 设置映像生成的输出目录。



`jlink` 工具位于 JDK 安装的 `bin` 目录中。它不会默认添加到系统路径中，所以如果想要像本示例这样使用它，必须要先将其添加到路径中。

可以提供多个以逗号分隔的根模块，而不是像此处使用多个 `--add-modules`。

指定的模块路径显式包含了包含平台模块 (`$JAVA_HOME/jmods`) 的 JDK 目录。这与使用 `java` 和 `javac` 时所看到的不同：假设平台模块来自运行 `java` 或 `javac` 的同一个 JDK。在 13.8 节中将会解释为什么 `jlink` 会有所不同。

正如在 4.7 节中所讨论的那样，服务提供者模块也必须作为根模块添加。只有通过 `requires` 子句才会对模块图进行解析；默认情况下 `jlink` 并不遵循 `uses` 和 `provides`



依赖关系。在 13.3 节中，将演示如何找到要添加的正确的服务提供者模块。



可以将 `--bind-services` 标志添加到 `jlink`，从而指示 `jlink` 在解析模块时考虑 `uses/provides`。但是，这样一来就绑定了平台模块之间的所有服务。因为 `java.base` 已经使用了很多（可选的）服务，所以这样做会导致重复解析更多不需要的已解析模块。

这些根模块中的每一个都已经被解析，这些根模块及其递归解析的依赖项成为 `.image` 中生成映像的一部分，如图 13-2 所示。

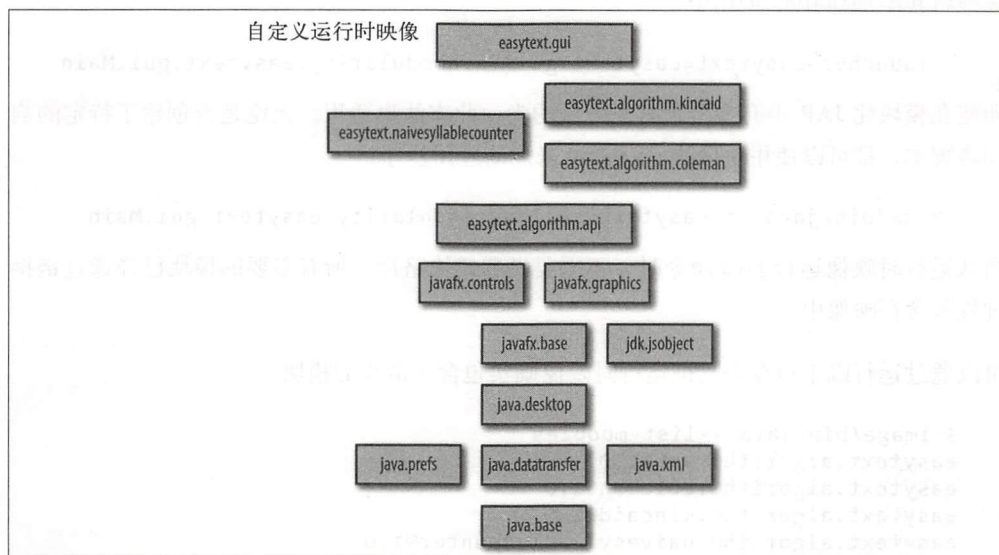


图 13-2: 自定义运行时映像仅包含应用程序所需的模块

生成的映像包含了以下所示类似于 JDK 的目录层次：

```

image
├── bin
├── conf
├── include
├── legal
└── lib
  
```

在生成的运行时映像的 `bin` 目录中可以找到一个 `easytext` 启动器脚本，它是因为 `--launcher easytext = easytext.gui` 标志而创建的。其中第一个参数是启动器脚本的名字，第二个参数是启动的模块。该脚本是一个可执行的便利包装器，它以 `easytext.gui` 作为运行的初始模块直接启动 JVM。可以从命令行通过调用 `image\bin\easytext` 直接执行脚本。在其他平台上，也会生成类似的脚本（请参阅 13.8 节以了解



如何针对其他平台生成类似脚本)。Windows 运行时映像获取的是批处理文件，而不是针对类似 UNIX 目标的 shell 脚本。

可以为那些包含一个入口点（静态主方法）的类的模块创建启动器脚本。如果按如下方式创建 `easytext.gui` 模块化 JAR，就属于这种情况：

```
jar --create \
  --file mods/easytext.gui.jar \
  --main-class=javamodularity.easytext.gui.Main \
  -C out/easytext.gui .
```

还可以通过分解模块构建运行时映像。此时，模块化 JAR 中没有主类属性，所以必须显式地将其添加到 `jlink` 调用中：

```
--launcher easytext=easytext.gui/javamodularity.easytext.gui.Main
```

即使在模块化 JAR 中有多个带有主方法的类，此方法也适用。无论是否创建了特定的启动器脚本，都可以使用映像的 `java` 命令来启动应用程序：

```
image/bin/java -m easytext.gui/javamodularity.easytext.gui.Main
```

当从运行时映像运行 `java` 命令时，不需要设置模块路径，所有必要的模块已经通过链接过程包含在映像中。

可以通过运行以下命令来验证运行时映像确实包含了最少的模块：

```
$ image/bin/java --list-modules
easytext.algorithm.api@1.0
easytext.algorithm.coleman@1.0
easytext.algorithm.kincaid@1.0
easytext.algorithm.naivesyllablecounter@1.0
easytext.gui@1.0
java.base@9
java.datatransfer@9
java.desktop@9
java.prefs@9
java.xml@9
javafx.base@9
javafx.controls@9
javafx.graphics@9
jdk.jsobject@9
```

上述内容与图 13-2 中所示的模块相对应。

除了包含了到目前为止所讨论的启动器脚本以外，`bin` 目录还可以包含其他可执行文件。在 EasyText GUI 映像中，还添加了 `keytool` 和 `appletviewer` 二进制文件。前者始终存在，因为它来自 `java.base`。后者是映像的一部分，因为所包含的 `java.desktop` 模块公开了小程序功能。由于其他 JDK 命令行工具（比如 `jar`、`rmic` 和 `javaws`）所依赖的模



块并没有包含在此运行时映像中，因此 jlink 足够聪明以忽略这些模块。

13.3 查找正确的服务提供者模块

在前面的 EasyText jlink 示例中，添加了几个服务提供者模块作为根模块。如前所述，可以使用 jlink 的 `--bind-services` 选项，并让 jlink 从模块路径中解析所有的服务提供者模块。虽然这种做法极具诱惑力，但也会导致映像中模块数量激增。盲目地为给定的服务类型添加所有可能的服务提供者是不正确的。需要认真考虑一下哪些服务提供者适合应用程序，并将其作为根模块添加。

幸运的是，可以通过使用 `--suggest-providers` 选项来帮助选择合适的服务提供者模块。在下面所示的 jlink 调用中，只添加了 `easytext.gui` 模块，并询问 Analyzer 类型的提供者模块的建议：

```
$ jlink --module-path mods/:$JAVA_HOME/jmods \  
      --add-modules easytext.gui \  
      --suggest-providers javamodularity.easytext.algorithm.api.Analyzer
```

Suggested providers:

```
module easytext.algorithm.coleman provides  
    javamodularity.easytext.algorithm.api.Analyzer,  
    used by easytext.cli,easytext.gui  
module easytext.algorithm.kincaid provides  
    javamodularity.easytext.algorithm.api.Analyzer,  
    used by easytext.cli,easytext.gui
```

然后，可以通过添加 `--add-modules <module>` 来选择一个或多个提供者模块。当然，当这些新添加的模块本身包含了 `uses` 子句时，则会按顺序再次调用 `--suggest-providers`。例如，在 EasyText 中，`easytext.algorithm.kincaid` 服务提供者模块本身对 `SyllableCounter` 服务类型使用了 `uses` 约束。

也可以在 `--suggest-providers` 之后放弃特定的服务类型并获得完整的概述，其中可能包括来自平台模块的服务提供者，所以输出可能会很快变得难以控制。

13.4 链接期间的模块解析

尽管 jlink 中的模块路径和模块解析在行为上与前面所介绍的其他工具类似，但也存在一些重要差异。其中一个例外是平台模块需要显式地添加到模块路径。

另一个重要的例外涉及自动模块。当使用 `java` 或 `javac` 在模块路径中放置非模块化 JAR 时，其将被视为满足所有意图和目的的有效模块（请参见 8.4 节）。但是，jlink 不会将模块路径上的非模块 JAR 识别为自动模块。只有当应用程序完全模块化时（包括所有库），



才可以使用 `jlink`。

原因是自动模块可以从类路径中读取，从而绕过模块系统的显式依赖关系。在自定义运行时映像中没有预定义的类路径，因此所生成映像中的自动模块可能会导致运行时异常。当它们依赖于类路径中不存在的类时，应用程序会在运行时崩溃。一旦出现这种情况，就会导致模块系统可靠配置保证的失效。

当然，如果确定一个自动模块的行为良好（也就是说，它仅需要其他模块，并且不需要从类路径中获取模块），那么就应该避免上述情况的出现。此时，可以通过将自动模块转换为显式模块来消除此限制。可以使用 `jdeps` 来生成模块描述符（如 10.3 节中所述），并将其添加到 JAR 中。现在有了一个可以放在 `jlink` 模块路径上的模块。但这并不是一个理想的情况；仅修补别人的代码永远不会得到理想的代码。自动模块只是一个用来帮助迁移的过渡功能。当遇到这种 `jlink` 限制时，也就是联系问题库的维护者并督促他们实现模块化的好时机。

最后，在链接期间，使用反射模块时的注意事项仍然适用，就像在完整的 JDK 上运行模块一样。如果一个模块使用了反射并且没有在模块描述符中列出依赖项，那么解析器对此将不予考虑。因此，包含被反射代码的模块可能并不包含在映像中。为了防止出现这种情况，请使用 `--add-modules` 手动添加模块，以便它们最终出现在运行时映像中。

13.5 基于类路径应用程序的 `jlink`

似乎 `jlink` 只能用于完全模块化的应用程序，这只能说是部分正确。还可以使用 `jlink` 创建 Java 平台的自定义映像，而不包括任何应用程序模块。例如，可以运行

```
jlink --module-path $JAVA_HOME/jmods --add-modules java.logging  
--output image
```

并获得一个只包含 `java.logging`（同时强制包含 `java.base`）模块的映像。虽然这么做用处不大，但却使事情变得更有趣。如果有一个现成的基于类路径的应用程序，则无法直接在应用程序代码上使用 `jlink` 为该应用程序创建自定义运行时映像。`jlink` 没有模块描述符来解析任何模块。

但是，可以使用诸如 `jdeps` 之类的工具来查找运行该应用程序所需的最小平台模块集。如果仅构建一个包含这些模块的映像，那么可以在此映像上启动基于类路径的应用程序，而不会出现问题。

这可能听起来有点抽象，接下来看一个简单的示例。在示例 2-5 中已经创建了一个简单的 `NotInModule` 类，它使用了 `java.logging` 平台模块。可以用 `jdeps` 检查该类，从而验证它是唯一的依赖项：



```
$ jdeps out/NotInModule.class
```

```
NotInModule.class -> java.base  
NotInModule.class -> java.logging  
  <unnamed>       -> java.lang           java.base  
  <unnamed>       -> java.util.logging      java.logging
```

对于较大的应用程序，可以使用相同的方式分析应用程序的 JAR 及其库。现在可以知道运行应用程序所需的平台模块了。对于上面的示例，只需要 `java.logging` 模块。同时，还创建了仅包含 `java.logging` 模块的映像。只需将 `NotInModule` 类放在运行时映像的类路径上，就可以在一个简化的 Java 发行版上运行这个基于类路径的应用程序：

```
image/bin/java -cp out NotInModule
```

由于 `NotInModule` 类在类路径上（假设位于 `out` 目录中），因此它位于未命名的模块中。未命名模块读取其他模块，在自定义运行时映像的情况下，该未命名模块是一个包含两个模块（`java.base` 和 `java.logging`）的小集合。通过这些步骤，甚至可以为基于类路径的应用程序创建自定义运行时映像。如果 `EasyText` 是基于类路径的应用程序，则使用相同的步骤会产生包含如图 13-1 所示的九平台模块的映像。

有一些需要关注的注意事项。当应用程序试图从不在运行时映像的模块中加载类时，会在运行时触发 `NoClassDefFoundError`。这类似于在类路径中缺少必要 JAR 的情况。哪些模块应该放到映像中完全由你来决定。在链接时，`jlink` 甚至不会查看应用程序代码，所以它无法像在一个完全模块化应用程序中那样帮助模块解析。虽然 `jdeps` 有助于估计所需的模块，但 `jlink` 的静态分析无法检测出应用程序中反射的使用。因此，在所产生的映像上测试应用程序是非常重要的。

此外，后续章节所讨论的性能优化并不都适用于基于类路径的场景。许多优化之所以有效，是因为所有的代码在链接时都可用——而前面所讨论的并不是这种情况。链接器并不会查看应用程序代码。在这种情况下，`jlink` 只能使用显式添加到映像中的平台模块以及已解析的平台模块依赖项。

`jlink` 的这种用法显示了 JDK 和 JRE 之间的界限在模块化的世界中如何变得模糊。链接允许使用任何所需的一组平台模块来创建 Java 分发，并且不局限于平台供应商提供的选项。

13.6 压缩大小

前面已经介绍了 `jlink` 的基本用法，接下来可以将注意力转移到优化方面的问题了。`jlink` 使用基于插件的方法来支持不同的优化。本节以及下一节介绍的所有标志都由 `jlink` 插件处理，其目的不是要对所有插件进行详尽的概述，而是突出一些插件来说明可能性。随着时间的



推移, jlink 插件的数量预计会稳步增长, 这些插件可能来自 JDK 团队或者整个社区。

可以通过运行 `jlink --list-plugins` 来获得所有当前可用插件的概述。一些插件默认是启用的, 它们将在上述命令的输出中显示。本节将着重介绍可以减小运行时映像的磁盘大小的插件。下一节将介绍运行时性能的改进。

就像为 EasyText 所做的那样创建一个自定义运行时映像可以省去不必要的平台模块, 从而减小所需磁盘的容量 (相对于使用完整的 JDK 而言)。除此之外, 还有更多的收获。可以使用多个标志进一步减小映像的大小。

第一个标志是 `--strip-debug`。顾名思义, 它将删除本地调试符号, 并从类中去除调试信息。对于生产版本来说, 上述功能都是需要的。但是在默认情况下该标志不启用。经验表明, 启用此标志后, 映像大小大约减小了 10%。

也可以使用 `--compress = n` 标志压缩生成的映像。目前, `n` 可以是 0、1 或 2。如果设置为最高值, 则完成了两件事情。首先, 创建一个在类中使用的所有字符串文本表, 从而在整个应用程序中共享表示形式。然后, 在模块上应用通用的压缩算法。

下一个优化会影响放入到运行时映像的 JVM。通过使用 `--vm = <vmtype>`, 可以选择不同类型的 VM。vmtype 的有效选项包括 `server`、`client`、`minimal` 和 `all` (此为默认值)。如果减少占用空间是最关心的问题, 那么请选择 `minimal` 选项。它所提供的 VM 仅包含一个垃圾收集器以及一个 JIT 编译器, 没有可维护性或检测支持。目前, 最小的 VM 选项仅适用于 Linux。

最后一个优化涉及语言环境。通常, JDK 附带许多语言环境, 以适应不同的日期/时间格式、货币和其他区域敏感信息。英文的默认语言环境是 `java.base` 的一部分, 因此始终可用。所有其他语言环境都是模块 `jdk.localedata` 的一部分。由于这是一个单独的模块, 因此可以选择是否将其添加到运行时映像。可以使用服务以公开来自 `java.base` 和 `jdk.localedata` 的语言环境功能。这意味着如果应用程序需要非英文语言环境, 则必须在链接期间使用 `--add-module jdk.localedata`。否则, 语言环境敏感代码会回到默认的英语语言环境, 因为在没有 `jdk.locale.data` 的情况下没有其他语言环境可用。请记住, 除非使用 `--bind-services`, 否则不会自动解析服务提供者模块。



当应用程序使用不属于默认集 (即 US-ASCII、ISO-8859-1、UTF-8、UTF-16) 的字符集时, 也会出现类似情况。传统上, 这些非默认字符集可以通过完整 JDK 中的 `charsets.jar` 获得。如果使用模块化 JDK, 为了使用非默认字符集, 需要将 `jdk.charsets` 模块添加到映像中。



然而, `jdk.locale` 相当大。添加该模块会增加大约 15MB 的磁盘大小。在许多情况下, 不需要在映像中包含所有的语言环境信息。如果是这样的话, 可以使用 `jlink` 的 `--include-locales` 标志。它将语言标记列表作为参数 (有关有效语言标记的更多信息请参阅 `java.util.Locale` JavaDoc)。然后 `jlink` 插件从所有其他语言环境中删除 `jdk.locale` 模块。最终, 只有指定语言环境的资源才会保留在映像中的 `jdk.locale` 模块中。

13.7 提升性能

上一节介绍了一系列有关磁盘上映像大小的优化, 而更令人感兴趣的是 `jlink` 优化运行时性能的能力。这一切都是通过插件实现的, 其中一些插件是默认启用的。请记住, `jlink` 及其插件目前还处于初级阶段, 主要是为了改善应用程序的启动时间。本节所讨论的许多插件在 JDK 9 中仍然是实验性的。大多数可用的性能优化需要深入了解 JDK 的工作原理。

很多实验性插件在链接时生成代码来提高启动性能。默认启用的一种优化是预先创建平台模块描述符缓存, 其主要想法是在构建映像时确切知道哪些平台模块是模块图的一部分。如果在链接时创建了模块描述符的组合表示, 那么就没有必要在运行时单独解析原始模块描述符了, 从而减少 JVM 启动时间。

另外, 其他插件在链接时执行了字节码重写, 从而提升了运行时性能。比如 `--class-for-name` 优化就是一个示例。它将形式为 `Class.forName("pkg.SomeClass")` 的指令重写为对该类的静态引用, 从而避免了在运行时反射式搜索类所带来的开销。另一个示例是预生成方法处理调用类的插件 (扩展 `java.lang.invoke.MethodHandle`), 否则这些类将会在运行时生成。虽然这可能听起来很深奥, 但 Java Lambda 表达式的实现使得可以大量使用这种方法处理机制。通过减少链接时类生成的成本, 大大加快了使用 Lambda 表达式的应用程序的启动速度。但不幸的是, 目前使用该插件需要对方法处理工作的方式有比较深入的了解。

正如所看到的, 许多插件提供了相当专业的性能调整方案, 有很多可能的优化是诸如 `jlink` 这样的工具所无法完成的。有些优化比其他优化更适合于某些应用程序。这是 `jlink` 提供基于插件的体系结构的主要原因之一。甚至可以编写自己的 `jlink` 插件, 尽管插件 API 本身在 Java 9 版本中被标记为实验性的。

对于传统上因为代价太高 JVM 无法即时执行的优化, 现在在链接时都是可行的。值得注意的是, `jlink` 可以优化来自任何模块的代码, 无论是应用程序模块还是所使用的库模块或是平台模块。不过, 因为 `jlink` 插件允许任意的字节码重写, 所以它们不仅仅只是用来增强性能。目前许多工具和框架都使用 JVM 代理在运行时执行字节码重写, 比如来



自对象关系映射器（如 OpenJPA）的检测代理或字节码增强代理。在某些情况下，这些转换可以在链接时应用。jlink 插件可以成为某些 JVM 代理实现的一个很好的选择（或补充）。



请记住，上述功能都是一些高级库和工具所完成的。不太可能将编写自己的 jlink 插件作为典型应用程序开发过程的一部分，就像不太可能编写自己的 JVM 代理一样。

13.8 跨目标运行时映像

由 jlink 创建的自定义运行时映像仅在特定的操作系统和体系结构上运行，这类似于 JDK 或 JRE 发行版在 Windows、Linux、macOS 等系统上存在的不同之处，这些版本包含运行 JVM 所需的特定于平台的本机二进制文件。可以为特定的操作系统下载 Java 运行时，并运行可移植的 Java 应用程序。

在前面的示例中，已经使用 jlink 构建了映像。虽然 jlink 是 JDK（其特定于平台）的一部分，但它可以为不同的平台创建映像，其过程非常简单。不要将当前运行 jlink 的 JDK 的平台模块添加到模块路径，而是指向具体的操作系统和体系结构的平台模块。获得这些替代平台模块就像下载并提取该平台的 JDK 一样简单。

假设我们在 macOS 上运行，并想为 Windows（32 位）创建一个映像。首先，为 32 位 Windows 下载正确的 JDK，并将其解压到 `~/jdk9-win-32`。然后，使用下面的 jlink 调用：

```
$ jlink --module-path mods/:~/jdk9-win-32/jmods ...
```

生成的映像包含来自 `mods` 的应用程序模块，而 `mods` 与来自 Windows 32 位 JDK 的平台模块相绑定。另外，映像的 `/bin` 目录包含了 Windows 批处理文件，而不是 macOS 脚本。现在剩下要做的就是将映像分发到正确的目标上！



使用 jlink 构建的运行系统映像不会自动更新。当发布新的 Java 版本时，需要构建更新的运行时映像。请确保仅分发根据最新 Java 版本而创建的运行时映像，以防止出现安全问题。

本章介绍了 jlink 如何创建简洁的运行时映像。当应用程序被模块化时，使用 jlink 是很简单的。虽然链接是一个可选的步骤，但是当面对资源受限的环境时链接能起到很大的作用。



第 14 章

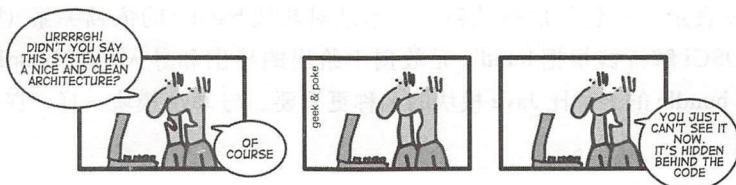
模块化的未来

到目前为止，即将结束 Java 模块系统的学习之旅——首先从模块化 JDK 开始，然后创建自己的模块并将现有代码迁移到模块。一方面，需要弄清楚围绕模块的许多新的功能。另一方面，Java 模块鼓励在模块化开发中采用经过实践检验切实可行的最佳实践。

除了最小的应用程序之外，建议针对新代码采用模块。如果在一开始就应用强封装并管理显式依赖关系，那么就为创建可维护系统奠定了坚实的基础。同时还开启了新功能，例如使用 jlink 创建自定义运行时映像。

进入 Java 的模块化未来需要遵守模块化原则。如果现有的应用程序采用了模块化设计，那么转换将会非常顺利。许多人已经使用多模块 Maven 或 Gradle 构建自己的应用程序。通常这些现有的模块边界可以自然映射到 Java 模块。在某些情况下，JDK 提供的 `ServiceLoader` 机制是成熟的依赖注入框架的一个很好的替代方法。

但是，当应用程序不是模块化时，转换将是极具挑战性的。



从前面的章节中已经看到，将现有的应用程序迁移到模块是可能的。但是，当它们的设计缺乏模块化时，在迁移过程中需要有效地解决两个问题。首先是解决架构问题，使其符合模块化原则。其次是实际迁移到 Java 9 及其模块系统。

是否值得花费精力实现模块化在一般指导方针中并没有给出很好的权衡分析，这取决于系统的应用范围、预期的生命周期以及许多其他依赖于上下文的可变因素。请注意，大量的 JDK 都成功地进行了模块化，虽然这需要花费数年的努力。显然，有志者事竟成。



所获得的收益是否超过成本是针对现有代码库需要回答的主要问题。在类路径上保留一个应用程序并不是一件丢脸的事情。

本章的其余部分将总结现有的模块化开发方法中的 Java 模块系统。

14.1 OSGi

早在 Java 模块系统之前，其他模块系统就已经出现了。这些现有的系统只提供了应用程序级的模块化，而 Java 模块系统对平台本身进行了模块化。目前最古老且最知名的模块系统是 OSGi。它通过在 OSGi 容器中运行 bundle（具有 OSGi 布线元数据的 JAR）来提供运行时模块化。通过巧妙地安排类加载器来控制类的可见性，从而实现了 bundle 之间的隔离。实际上，每个 bundle 都由独立的类加载器加载，并根据 bundle 的元数据委托给其他类加载器。在 OSGi 容器中通过类加载器实现的隔离仅在运行时发生。而在构建时，必须使用诸如 Bndtools 或 Eclipse PDE 之类的开发工具来执行强封装和依赖关系。

随着 Java 模块系统的引入，OSGi 过时了吗？事实并非如此。首先，现有的 OSGi 应用程序可以在 Java 9 上通过使用类路径继续运行。当有一个基于 OSGi 的系统时，不要急于将 bundle 转变为 Java 模块。另外，OSGi Alliance 正在进行 OSGi bundle 和 Java 模块之间互操作性的前期研究工作。

现在，问题就变成了：针对新的系统，何时使用 OSGi 以及何时使用 Java 模块系统？要回答这个问题，了解 OSGi 和 Java 模块系统之间的差异很重要。

OSGi 和 Java 模块系统之间存在以下显著差异：

- *包依赖关系*

OSGi bundle 表示了对包的依赖关系，而不是对其他 bundle 的依赖关系（尽管这也是可能的）。OSGi 解析器根据 bundle 元数据中给出的导出和导入包将 bundle 连接在一起。这使得 bundle 的名称比 Java 模块的名称更重要。与 Java 模块一样，在包级别导出 bundle。

- *版本控制*

与 Java 中的模块不同，OSGi 中 bundle 和包都有版本。依赖项可以用精确的版本或版本范围来表示。因为每个 bundle 都被加载到一个单独的类加载器中，所以 bundle 的多个版本可以共存，但也不是没有限制。

- *动态加载*

bundle 可以在 OSGi 运行时中加载、卸载、启动和停止。这些 bundle 生命周期事件都有



回调，因为 bundle 必须应对动态环境。可以在运行时在 `ModuleLayer` 中加载 Java 模块，并在稍后进行垃圾回收。与 OSGi bundle 相比，Java 模块没有定义明确的生命周期回调，因为它假设了更静态的配置。

- 动态服务

OSGi 还定义了一个带有中央服务注册表的服务机制。OSGi 服务的 API 比 Java 的 `ServiceLoader` 提供了更丰富的功能。许多高级框架（比如 `Declarative Services`）是在基本的 OSGi 服务之上提供的。OSGi 服务可以在运行时进出，而不是一成不变，因为提供服务的 bundle 可以动态地来回移动。带有 `ServiceLoader` 的 Java 服务仅在模块解析期间连线一次。只有使用 `ModuleLayer` 才能在运行时引入新的服务。与 OSGi 服务不同，Java 服务不支持启动和停止回调。

除了上述差异之外，还有一个更重要的区别：OSGi 在运行时支持更多动态的场景。OSGi 在这方面比 Java 模块系统具有更多的功能，这是因为 OSGi 植根于嵌入式系统。零停机更新是可能的，因为 OSGi bundle 可以热插拔。一旦插入新的硬件，支持它们的服务就会动态启动。

在企业软件中，同样的范例可以在运行时以动态可用性扩展到其他资源。如果需要这些动态性，就应该使用 OSGi。OSGi 的动态生命周期确实为开发人员添加了一些复杂性。部分固有的复杂性可以在开发过程中通过使用诸如 `Declarative Services` 之类的高级框架进行抽象化。实际上，许多应用程序（包括那些使用 OSGi 的应用程序）往往在启动时将服务连接在一起，之后没有任何动态变化。在这些情况下，Java 模块系统提供了足够的功能。

围绕 OSGi 的相关工具已经成为许多开发人员受挫的源头。即使十多年后，它也不具备足够的重要性来得到社区和供应商的充分关注。随着模块成为 Java 平台的一部分，供应商已经在其发布之前创建了工具和支持。由于 OSGi 框架仅在运行时才起作用，工具必须在开发过程中模仿其规则，而 Java 的模块系统规则在所有阶段（从开发到运行时）以一致的方式执行。

Java 模块系统还提供了 OSGi 不具备的功能，主要涉及迁移到模块。OSGi 中不存在自动模块的对等物。并不是所有的 Java 库都提供（正确的）OSGi 元数据，这意味着补丁或者 `pull` 请求是 OSGi 使用这些库的唯一方法。库有时候也会包含一些代码，而这些代码在 OSGi 独立的类加载设置中不起作用。在 Java 模块系统中，类加载以向后兼容的方式实现。由于 Java 模块系统不使用类加载器进行隔离，因此它提供了更强的封装。在 JVM 内部强制执行了基于可访问性和可读性的全新机制。

时间会告诉我们采用 Java 模块系统是否会更好。由于现在模块是 Java 平台本身的一部



分，因此期望 Java 社区认真对待模块化。

14.2 Java EE

模块是一个 Java SE 功能。然而，有许多开发人员开发 Java EE 应用程序。在 Java EE 中，Web Archive (WAR) 和 Enterprise Archive (EAR) 将 JAR 文件与部署描述符捆绑在一起。然后，再将这些 Java EE 应用程序部署到应用程序服务器上。

鉴于现在模块是 Java 平台的一部分，Java EE 将如何发展呢？可以合理地假设一下在未来的某个时刻 Java EE 规范中将包含模块。目前确定 Java EE 是什么样子还为时过早。Java EE 8 版本建立在 Java SE 8 之上。因此模块和 Java EE 最早在 Java EE 9 中融合，并且没有设置发布日期。但在此融合之前，Java EE 应用程序服务器可以像以前一样继续使用类路径。

在 6.3.4 节中看到，原则上，模块系统具有支持应用程序容器（比如，Java EE 应用程序服务器）的功能。模块化的 Java EE 应用程序看起来像什么还有待观察。对于应用程序来说，Java EE 模块化可能是模块化 WAR 或 EAR 文件的一种形式，也可能是一种全新的形式。

良好的第一步是将 Java EE API 作为带有标准化名称的模块发布。官方的 JAX-RS API 发行版已经有了一个模块描述符。通过使用模块，可以将 Java EE 视为一组松散的相关联的规范。没有必要等待整个应用程序服务器一次性支持所有规范。当可以请求所需的部分 Java EE 时，就打开了一扇新的大门。不过，请记住，这一切都是基于可能性而进行的推测。

14.3 微服务

在过去几年中，微服务作为一种架构风格已经取得了显著的地位。引用微服务的好处之一是它们可以实现模块化开发。通过将系统划分为可独立部署且可运行的部分，对系统实现了模块化。每个微服务都作为一个独立的过程运行，甚至可以用完全不同的技术来编写多个微服务。它们通过使用诸如 HTTP 和 gRPC 的标准协议在网络上进行通信。

事实上，这种架构风格固有地强化了模块边界。微服务如何满足模块化的其他两个原则，即定义良好的接口和显式依赖关系？定义微服务之间的接口的方式很多，从在接口定义语言 (Interface Definition Language, IDL)，比如 Protocol Buffers、RAML 或者 WSDL / SOAP 中的严格定义到 HTTP 上未指定的 JSON。微服务之间的依赖关系通常是在运行时通过动态发现而产生的，没有多少微服务栈提供了类似于模块描述符中 `requires` 的静态可验证依赖关系。



在本书中已经看到，在无须过程隔离的情况下可以实现模块化。通过使用 Java 模块系统，Java 编译器和 JVM 可以强制实现模块之间的强封装和显式依赖。借助于模块系统中的显式 `provides/uses`，可以使用 Java 接口和服务完成这种模块化开发方法。从这个角度来看，微服务有点像带有网络边界的模块。但是这些网络边界将微服务系统变成了带有相关缺点的分布式系统。如果仅仅为了模块化特性而选择微服务，那么请仔细考虑一下。使用 Java 模块可以为系统带来类似（甚至更强大的）模块化优势，同时没有微服务架构的操作复杂性。

公平地讲，除了模块化的原因之外，还有很多其他的原因使之选择微服务。考虑独立的更新和服务的扩展，或者每个服务使用不同的技术堆栈确实是有益的。此外，在模块或微服务之间进行选择不一定非此即彼。模块可以为微服务实现创建一个强大的内部结构，使其能够超越通常所认为的微观尺度。初始开发一个带有模块的系统是相当明智的，而在后续阶段，当操作需要时可以将一些模块提取到自己的微服务中。

14.4 下一步

前面已经讨论了几种可选的模块化方法以及它们与 Java 模块系统的关系，不管使用什么样的技术，最困难的部分仍然在于正确分解应用程序的域。Java 模块是工具箱中另一个强大的工具，可以创建结构良好的系统。

模块系统的当前状态是否完美呢？没有什么系统是绝对完美的，模块系统也不例外。在 Java 平台的生命周期中加入一个模块系统不可避免地会导致一定的损害。但 Java 9 仍然打下了一个坚实的基础。当然，`ModuleLayer` 除了目前的用处之外，解决诸如同时运行多个版本的模块等问题也是非常合适的。虽然当前在模块路径上并不支持 `ModuleLayer`，但这并不意味着其永远不会被支持。对于其他功能也是一样。模块系统还没有完成，可以肯定的是在后续版本中将会获得新的功能。目前，Java 生态系统和工具供应商都支持模块化 Java。

随着 Java 社区开始接受模块系统，更多的库将变为模块。这使得在自己的应用程序中使用模块变得更容易，虽然前面已经讲过自动模块是可以接受的临时解决方案。在一个新的环境中，首先获得模块系统的使用经验是有意义的。可以像本书中所看到的 `EasyText` 应用程序一样构建一个包含少量模块的小应用程序，从而了解一个“干净”的模块化应用程序应该是什么样子的。有了这些经验之后，就可以采取更加“雄心勃勃”的步骤，如模块化现有的应用程序。

将模块系统作为 Java 平台的一部分是一个改变“游戏规则”的尝试，这不是一夜之间就会发生的。Java 社区需要相当长的时间才能接受 Java 模块系统的概念，这是人的本性所造成的：模块化不是可以立马添加到现有代码库中的快速修复或新功能。然而模块化的



优点是显而易见的。通过微服务重新关注模块化说明了大多数人直观地把握到这一点。通过模块系统，Java 开发人员获得了构建可维护的大型系统的新选择。

到目前为止，已经了解了 Java 模块系统的概念。更重要的是，知道了模块化背后的原理以及如何使用模块系统来进行模块化。现在是时候在实践中使用这些知识了。愿你所创建的软件有一个模块化的未来！



作者简介

Sander Mak 是荷兰 Luminis 公司的一名研究员，开发了许多主要用于 JVM 上的模块化以及可扩展软件，但也会在需要的地方使用 TypeScript。他经常在各种会议上发言，并热衷于通过博客 (<http://branchandbound.net>) 和作为 Pluralsight 平台的讲师分享知识。可以在 Twitter 上通过 @sander_mak 来关注他。

Paul Bakker 是 Netflix 公司的一名高级软件工程师，在 Edge Developer Experience 团队主要从事工具的开发，以提高公司内部开发人员的工作效率。除了热爱编写图书之外，还喜欢与他人分享知识。与他人合著了《*Modular Cloud Apps with OSGi*》(由 O'Reilly 出版) 一书，本书是 Paul Bakker 的第二本书。Paul 经常在模块化、容器技术及相关主题的会议上发言。他经常在 <http://paulbakker.io> 上发布博客。可以在 Twitter 上通过 @pbakker 关注他。

封面简介

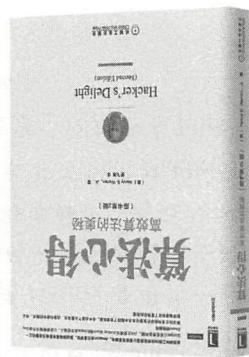
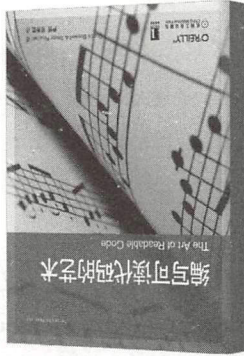
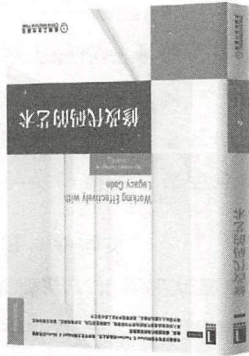
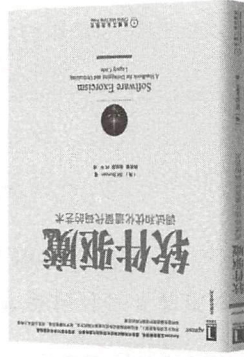
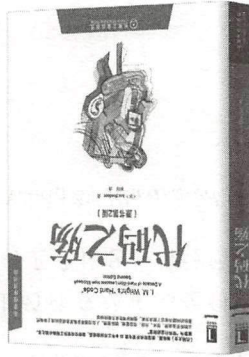
本书封面上的动物是一只黑尾膝鹑 (*Limosa limos*)，一种大型涉水鸟，由 Carl Linnaeus 于 1758 年首次描述，其生活在欧洲和亚洲。它的繁殖范围从冰岛一直延伸到印度北部，栖息于湿地、河口和湖边。

黑尾膝鹑的腿和喙特别长，从而可以让它们在沼泽栖息地上行走并寻找食物。它们的食物主要是昆虫和水生植物。当它们飞行时，最容易看到的是其由此得名的黑色尾巴。黑尾膝鹑大多遵循“一夫一妻制”。在繁殖地点交配后，雌性膝鹑会在巢中产卵三到六枚。单身雄膝鹑会为了领地而争斗，并展示自己以吸引雌性膝鹑。

在欧洲，黑尾膝鹑曾被认为是一种美食，但因为其数量的不断减少导致欧洲各国政府实施了狩猎限制 (尽管法国仍允许有限数量的狩猎)。黑尾膝鹑目前被列为濒危物种。

O'Reilly 封面上的许多动物都受到威胁；所有这些动物对世界都很重要。想要了解有关如何提供帮助的更多信息，请访问 animals.oreilly.com。

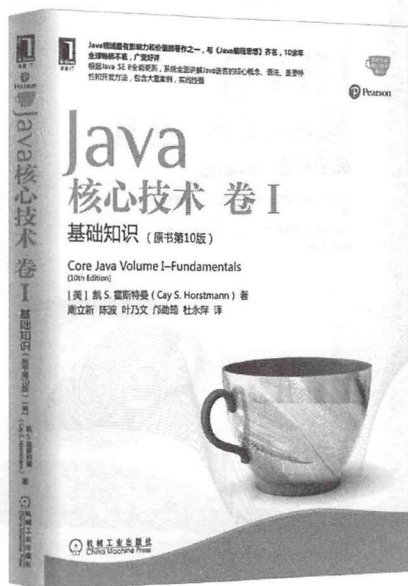
封面图片来自 Wood 的 *Illustrated Natural History*。



推荐阅读



推荐阅读



Java核心技术 卷I：基础知识（原书第10版）

书号：978-7-111-54742-6 作者：（美）凯 S. 霍斯特曼（Cay S. Horstmann） 定价：119.00元

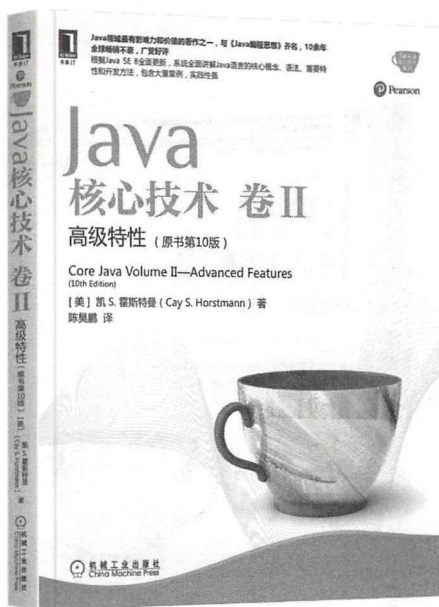
Java领域最有影响力和价值的著作之一，与《Java编程思想》齐名，10余年全球畅销不衰，广受好评

根据Java SE 8全面更新，系统全面讲解Java语言的核心概念、语法、重要特性和开发方法，包含大量案例，实践性强

本书为专业程序员解决实际问题而写，可以帮助你深入了解Java语言和库。在卷I中，Horstmann主要强调基本语言概念和现代用户界面编程基础，深入介绍了从Java面向对象编程到泛型、集合、lambda表达式、Swing UI设计以及并发和函数式编程的最新方法等内容。



推荐阅读



Java核心技术 卷II 高级特性 (原书第10版)

书号：978-7-111-57331-9 作者：Cay S. Horstmann 定价：139.00元

Java领域最有影响力和价值的著作之一，与《Java编程思想》齐名，10余年全球畅销不衰，广受好评

根据Java SE 8全面更新，系统全面讲解Java语言的核心概念、语法、重要特性和开发方法，包含大量案例，实践性强

本书为专业程序员解决实际问题而写，可以帮助你深入了解Java语言和库。在卷II中，Horstmann主要提供了对多个高级主题的深度讨论，包括新的流API、日期/时间/日历库、高级Swing、安全、代码处理等主题。

Java 9 模块化开发：核心原则与实践

Java 9向Java平台引入了模块系统，这是一个非常重要的飞跃，标志着模块化Java软件开发的新纪元。当需要创建灵活且易于维护的代码时，模块化是一个关键的架构设计原则。本书给出了Java模块系统的明确概述，并演示了如何通过创建模块化应用程序来帮助管理以及降低复杂性。

作者引导我们了解了模块系统中的相关概念以及工具，介绍了可以将现有代码迁移到模块中的模式并以模块的方式构建新的应用程序。

- 了解Java平台自身如何实现模块化
- 学习模块化如何影响应用程序的设计、编译、打包以及开发
- 编写自己的模块
- 使用模式改进任意代码库的可维护性、灵活性以及重用性
- 学习如何使用服务来创建解耦模块
- 将现有代码迁移到模块，并学习如何使用并不是模块的现有库
- 创建优化的自定义运行时映像，从而改变装载模块化Java应用程序的方式

Sander Mak是荷兰Luminis公司的一名研究员，开发了许多主要用于JVM上的模块化以及可扩展软件，但也会在需要的地方使用TypeScript。他经常在各种会议上发言，并热衷于通过博客 (<http://branchandbound.net>) 和作为Pluralsight平台的讲师分享知识。

Paul Bakker是Netflix公司的一名高级软件工程师，在其Edge Developer Experience团队主要从事工具的开发，以提高公司内部开发人员的工作效率。曾与他人合作编著了《Modular Cloud Apps with OSGi》（由O'Reilly出版公司出版）一书。Paul经常在与模块化、容器技术相关主题的会议上发言。

“本书提供了在Java 9中创建模块化应用程序时所需的基本实用知识。模块化是多年以来JDK中最重要功能之一，对于任何希望使用该功能的开发人员或架构师来说，本书都是必读的。”

——Simon Maple

ZeroTurnaround公司
开发者关系总监

“模块化是很难学习的。但幸运的是，我已经能够使用Paul和Sander所编写的书作为指南来编写自己的Java 9教程、讲义以及将jClarity的应用程序转换为使用Java的新模块化系统。目前我正在为jClarity的所有工程团队购买本书，这本书真是太好了！”

——Martijn Verburg

jClarity公司的CEO，
Sun/Oracle Java 冠军程序员

O'Reilly Media, Inc. 授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn



上架指导：计算机/程序设计

ISBN 978-7-111-60129-6



9 787111 601296 >

定价：69.00元