首本讲解基于Java标准规范实现REST的专著，阿里巴巴Java技术专家12年开发经验结晶，3位业内著名技术专家联袂推荐！

深刻解读JAX-RS标准和API设计，Jersey的使用要点和实现原理，以及基于REST的Web服务的设计思想、原则和特点。

Java核心技术系列

HZ BOOKS 华章科技

Java RESTful Web Service实战

Java RESTful Web Services in Action

韩陆 著

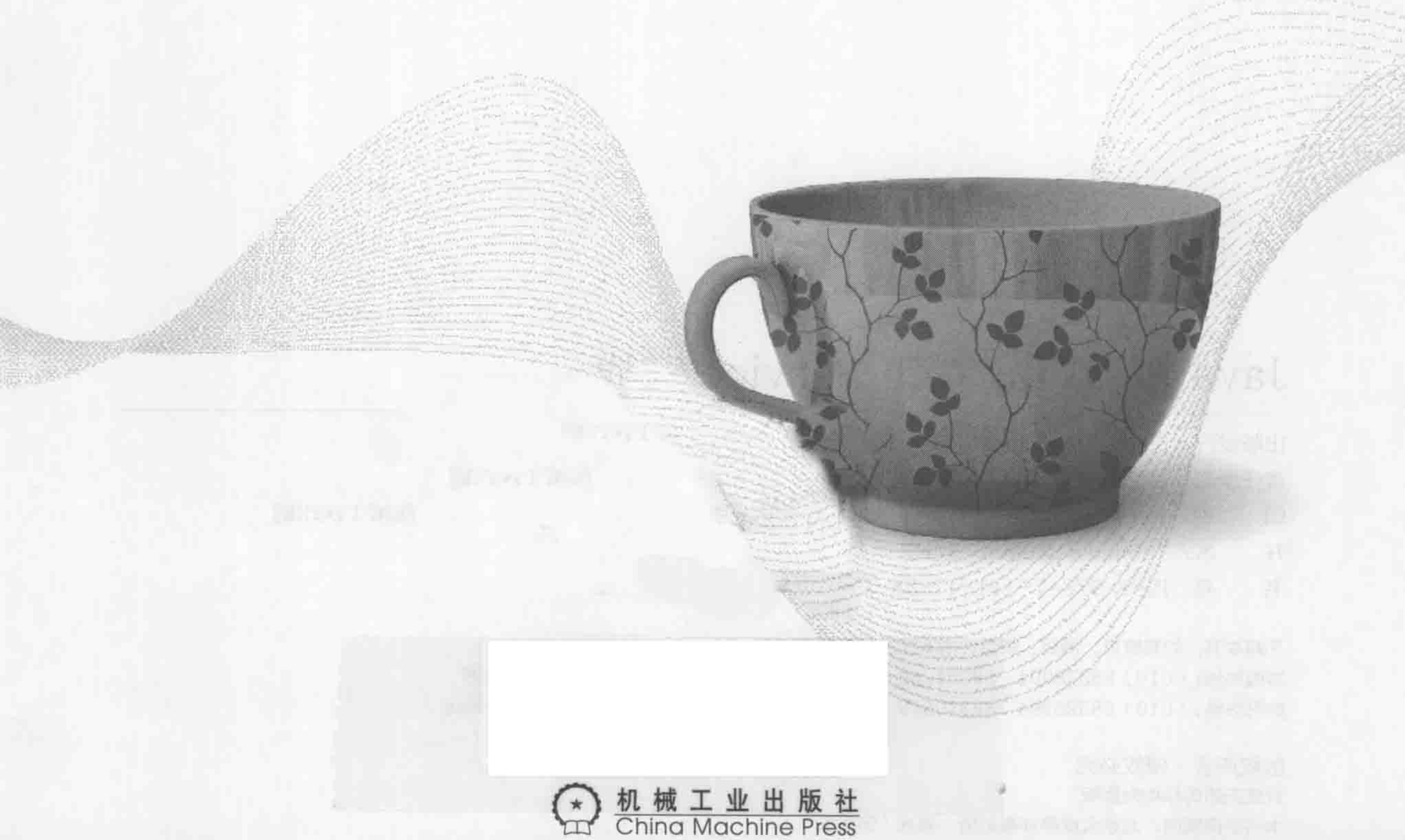
机械工业出版社
China Machine Press

Java 核心技术系列

Java RESTful Web Service实战

Java RESTful Web Services in Action

韩陆 著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Java RESTful Web Service 实战 / 韩陆著 . —北京：机械工业出版社，2014.10
（Java 核心技术系列）

ISBN 978-7-111-47888-1

I. J…　II. 韩…　III. JAVA 语言 – 程序设计　IV. TP312

中国版本图书馆 CIP 数据核字（2014）第 207228 号

Java RESTful Web Service 实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：高婧雅　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2014 年 10 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：19.75
书　　号：ISBN 978-7-111-47888-1　　定　　价：69.00 元

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）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有 · 侵权必究
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

Foreword 推荐序一

——REST 开发的理想与现实

REST 是一种分布式应用的架构风格，也是一种大流量分布式应用的设计方法论。REST 是由（构成了 Web 基础架构的）HTTP、URI 等规范的主要设计者 Roy Fileding 博士在其 2000 年的博士论文（中文版名为《架构风格与基于网络应用软件的架构设计》）中提出的。到目前为止，关于 REST 最系统、最全面的论述，仍然是 Fielding 的博士论文。

REST 就是 Web（World Wide Web，简称 Web 或者 WWW）本身的架构风格，是设计、开发 Web 相关规范、Web 应用、Web 服务的指导原则。不符合 REST 风格要求的架构和技术，很难在 Web 这个生态系统中得到繁荣发展。在我看来，Roy Fielding 博士就是 15 年以来对于分布式应用架构设计理论贡献最大的人。Fielding 在 HTTP 规范的设计过程中，并没有采用当时大行其道的 DO（Distributed Object，分布式对象）风格，而是自出机杼、另辟蹊径，提出了一整套新的设计方法论。Fielding 的开创性工作，极大地推动了分布式应用设计理论的发展。

有趣的是，其实基于 SOAP/WSDL 的“大 Web Service”（以下简称 Web Service），几乎是与 REST 同时发展起来的。虽然在 Web Service 中也使用了对象，但是 Web Service 其实是 RPC 风格的，而不是 DO 风格的。Web Service 在最初几年发展很快，很大原因是它解决了 DO 风格难以解决的异构系统（不同的硬件系统、不同操作系统、不同的编程语言，等等）之间互操作性的问题。

然而遗憾的是，设计 Web Service 协议栈的核心人员，几乎都是来自于企业应用阵营的，尤其是来自于 IBM 和微软两家公司的人。这些企业应用的专家们没有充分认识到 Web 基础架构的巨大优点，甚至可以说并没有理解 HTTP 协议究竟是用来做什么的、为何要如此设计。在 Web Service 协议栈的设计之中，仍然有深深的企业应用痕迹。Web Service 虽然宣称能够很好地支持互操作，然而因为协议栈的复杂性很高，在实战中互操作性并不好（例如升

级过程困难而且复杂）。此外，Web Service 仅仅将 HTTP 协议当做一种传输协议来使用，还依赖 XML 这种冗余度很高的文本格式，这导致 Web Service 应用性能低下。很多开发团队宁可使用 Hessian 等轻量级的 RPC 协议，也不愿意使用 Web Service。在面向互联网的大流量 Web 应用（包括 Web 服务在内）这种运行环境中，Web Service 在复杂性、互操作性、性能、可伸缩性等方面的短板更加突出。因此，设计今日面向互联网的 API，已经很少有人会考虑 Web Service。这使得 Web Service 的使用被局限在企业应用运行环境之中，其名称中的“Web”更像是一个笑话（除了都使用 HTTP 协议，基本上与 Web 没什么关系）。假如在 2000 年，设计 Web Service 规范的专家们，能够认真读一下 Fielding 的博士论文，或者找 HTTP、URI 等 Web 基础架构规范的核心设计人员深入交流一下，Web Service 很可能就不是现在这个样子了。不过，历史是无法假设的。

在 Java 世界中，与大 Web Service 相对应的规范是 JAX-WS。在大 Web Service 已经成为明日黄花之后，Java 世界急需一套新的规范来取代 JAX-WS。这套新的规范就是 JAX-RS：Java 世界开发 RESTful Web Service（与 RESTful API 含义相同，可混用）的规范。虽然起步很晚，毕竟走上了正确的道路。

从 Java EE 6 开始，JAX-RS 在 Java EE 版图中，作为最重要的组成部分之一，逐步取代了 JAX-WS 的地位。在所有 Java EE 相关规范中，JAX-RS 是优点很突出的一个。例如，完全基于 POJO、很容易做单元测试、将 HTTP 作为一种应用协议而不是可替代的传输协议（因此提高了性能）、优秀的 IDE 集成，等等。可以说，在大多数场合，JAX-RS 完全可以取代 JAX-WS，作为 Java Web Service 开发的主要技术。JAX-RS 同样也可以完全取代 Hessian 等基于 HTTP 协议的 RPC 风格远程调用协议。毕竟 HTTP 本身就是一种 REST 风格的应用协议，以 REST 风格来使用 HTTP，才是最高效的使用方式。

Jersey、CXF 等支持 JAX-RS 规范的 REST 开发框架还支持输出 WADL。WADL 支持客户端代码自动生成，还可以将 WADL 导入到 SoapUI 等测试工具中，然后做自动化集成测试。从开发 Java 企业应用、取代 JAX-WS 的角度来看，JAX-RS 已经做得非常棒了。

尽管如此，不可不提的是，JAX-RS 这套规范，仍然存在着很多遗憾。需要特别指出的是，JAX-RS 规范并不等于 REST 架构风格本身，REST 的内涵要比 JAX-RS 广泛得多。学会了使用 JAX-RS 了，并不等于就完全理解了 REST，开发者仍然需要下工夫认真学习一下本源的 REST 究竟是什么。

例如，JAX-RS 规范对于应该如何定义一个资源，以及应该如何使用 HTTP 作为一个统一接口来操作资源，显然缺乏必要的指导。有很多开发者只是简单地将以前 JAX-WS 中的一个 endpoint 接口转换成一个资源接口。接口的方法太多，导致映射到的 HTTP 方法不够用，这也难不倒他们，在 URI 路径中加一些字符串就能够解决（例如，三个接口方法都映射到

POST，但是其 PATH 不同）。这样的开发方式非常常见，虽然开发者使用了 JAX-RS 规范，但是开发方式完全是 RPC 风格的，可以说与 REST 风格没有任何关系。

此外，JAX-RS 规范目前尚不支持 HATEOAS（将超文本作为应用状态的引擎，REST 风格的核心特征之一），从著名的 Richardson 成熟度模型（由《RESTful Web APIs》的作者 Richardson 提出）来衡量，基于 JAX-RS 规范实现的 RESTful API 仅仅能够达到成熟度模型的第二级，即支持资源抽象、统一接口的“CRUD 式 Web 服务”。

可以这样说，JAX-RS 规范与真正的 REST 风格，覆盖的范围其实是不同的。JAX-RS 覆盖的是简单基于 HTTP 协议（没有使用 SOAP/WSDL）的各种远程调用需求，很多需求对于可伸缩性、松耦合的要求并不高，仅仅是希望使用 HTTP 本身来取代大 Web Service 作为一种轻量级、容易测试的远程调用协议。REST 架构风格的严格要求，在这些场合并不是非常重要。慵懒是人类的天性，大多数开发者写代码只是为了解决手头的问题，JAX-WS 并不好用，JAX-RS 解救了他们。

如果按照 Roy Fielding 博士的严格要求（REST APIs must be hyper-text driven），那么包括 JAX-RS 规范在内都不能算是真正的 RESTful。然而，从实战角度，我认为革命不分先后，只要能够达到 Richardson 成熟度模型第一级，即有清晰的资源抽象，就可以认为是 RESTful API 了。如果连第一级都达不到，所设计的架构根本就不是面向资源的，那八成还是 RPC 风格的，就没有必要非要往 RESTful API 阵营里面挤了。从来没有人说过 RPC 就是万恶的，RPC 在企业应用的大多数场合其实都非常有效，只是不适合面向互联网的大流量 Web 应用而已。

因此，能够完美支持 HATEOAS，攀登到成熟度模型第三级，是一种理想情况（当然也是值得追求的）。而通过部分拥抱 REST 风格的要求，来更好地解决手头的问题，是更多开发者所面对的现实情况。JAX-RS 反映的正是这种现实情况，从实战的角度，它是一套非常有用也很好用的规范。

韩陆兄的新著《Java RESTful Web Service 实战》是 JAX-RS 规范方面的专著，也是国内第一本 REST 开发的原创著作。这本书的实战性非常强，全面介绍了 JAX-RS 2.0 的方方面面，可以作为一线 Java 分布式应用开发者的案头必备书。如同我在前面所指出的，JAX-RS 规范并不等于 REST 架构风格本身，它们有着不同的覆盖范围。在本书中，作者也介绍了很多设计 RESTful API 的最佳实践，这些内容假如读者不理解 REST，甚至在亲自阅读了 JAX-RS 规范之后也未必能够总结出来。读者在阅读本书的过程中，不应该仅仅满足于掌握了 JAX-RS 开发的基本方法、解决了手头的问题、用其完全取代 JAX-WS，更重要的是，读者还应该就 REST 架构风格本身做更多的学习。幸运的是，除了本书之外，目前 REST 设计和开发方面的图书资料已经非常多了。

本书的内容非常严谨，有非常好的系统性，对于设计开发大流量 Web 服务会面临的各种问题都有涉及。特别是在自动化测试方面着墨颇多，在我看来是本书的一大亮点。RESTful API 的自动化测试非常重要，需要在设计之初就充分考虑到。本书是一本难得的原创佳作，值得所有 Java 分布式应用的开发者购买。

理想富丽丰满，现实贫瘠骨感，追求理想和注重解决现实问题其实并不矛盾。JAX-RS 规范的发展，反映出了 Java 社区在更好地开发 RESTful Web Service 方面的求索。尽管存在争议，在我看来，规范化是推动 RESTful Web Service 取得更大发展的必由之路。目前对于优秀的 RESTful API 有哪些判断标准，Web 开发者社区已经达成了高度共识，也积累了大量非常有价值的成果。JAX-RS 规范的发展，离不开 Web 开发者社区的这些成果。在未来的 JAX-RS 3.0 规范中，我们将会看到更多令人兴奋的成果被规范化。JAX-RS 2.0 已经做得不错了，但是在 RESTful Web Service 规范化的道路上，其实才刚刚起步，任重而道远。

李锟 于上海

Foreword 推荐序二

半年前初识韩陆的时候，我们就聊到他正在写的这本书，当得知我从 2006 年就参与了 Apache CXF 开发，他立即邀请我为他的新书写序，我也就欣然答应了。

Apache CXF 作为 JAXWS 以及 JAX-RS 规范的实现框架，已经成为很多 Web 服务开发者必选的开发框架。作为这一框架的开发维护者之一，我的日常工作经常需要熟悉这些 JSR 规范，并实现 JSR 所定义的 API，解决最终用户的使用问题。

熟悉 Java 的人大多都听说过 JSR（Java Specification Requests）、JCP（Java Community Process），通过 JSR 可以就 Java 某一方面的应用定义一组标准的 API 或者服务。对于最终用户来说，他们的代码只需要调用 JSR 定义的标准 API，不做任何修改就可以调用不同的 JSR 实现。这里常见的例子就是 Java Servlet 应用，用户开发的 Web 应用可以不做任何修改就部署到 Tomcat、JBoss 等不同的 Web 容器中。

JAXRS 是 JCP 为 Java RESTful Web Service 定义的一套 API。由于 Web 服务的描述模型与 Java 类和接口有一定的差距，JAX-RS 定义了很多 annotation，通过这些 annotation 我们可以很方便地将 Java 类描述成为相关的 REST 服务。由于 RESTful Web Service 通常需要部署到 Web 容器中，JAX-RS 也定义了相关服务来发现部署到容器中的 JAX-RS 应用。

读过 JSR 规范的朋友或多或少都会有这样的体会，JSR 作为规范文档，其目标是将 API 定义以及实现功能描述清楚、完备，其目标读者是相关 API 的实现人员，或者是相关 API 的高级使用人员。如果读者对相关的背景知识还不熟悉的话，JSR 文档读起来会比较晦涩而且难以理解。加之绝大部分 JSR 文档都没有相关的中文翻译，对于绝大多数初学者来说，通过阅读 JSR 文档来学习相关的 API 的知识是一个艰难的过程。

如果我们想要对 JAX-RS 规范有一个比较快速并且全面的了解应该怎么办呢？一般来我们可以通过 JSR 的相关参考实现入手，我们不但可以通过运行相关的参考实现的例子快速入门，还可以通过跟踪相关的代码对实现细节有一个全面的了解。韩陆的这本新作以 JAX-RS

的参考实现 Jersey 为蓝本，由浅入深地向大家介绍了 JAX-RS 的由来，以及与 RESTful Web 服务开发的相关 API，并结合实例分享了作者的实战经验。

好了，现在打开你熟悉的 IDE 工具，加载 Jersey 代码库，沿着本书的指引去探索 Java RESTful Web Services 开发世界吧。

RedHat 姜宁

Preface 前　言

从我启动本书到完稿，历时一年有余。

此间，Java EE 7 得到了更多服务器软件的支持，Jersey 升级了 9 个小版本——我在动笔开始文字和示例代码编写的时候，Jersey 刚刚推出 2.0 版本，到本书完毕时，版本号是 2.9，这也是本书的最终版本。此后新版本带来的改变只有通过本书提供的源代码来同步更新。

此间，我积极参与和关注着 Jersey 项目的动向，通过关注 Jersey 官方文档、Jersey 在 GitHub 托管系统的源代码、Jersey 的 Jira 缺陷管理系统、Jersey 的 StackOverflow 问答系统，对其修复缺陷、引入新功能和如何使用 Jersey 等事宜不断跟进。

我之所以这么做，目的只有一个，即希望为读者呈现的是一本 Java 领域 REST 开发的好书。

为什么要写这本书

REST 式的 Web 服务有多流行，相信每一位翻阅本书的读者都很清楚，冒昧地猜测，你可能想要看到的是一本讲述如何使用 Java 语言和 Java EE 平台，来实现这一风格的服务或者应用的书。这也正是我这两年来努力写作的初心和原动力。我相信，读者希望看到的内容不单单是追逐流行、风靡一时的“快餐”。作为开发者，我知道拥有对新技术、新标准的敏锐嗅觉非常重要，但我认为更难能可贵的是把一个业内认可的标准学好和用好。Java EE 7 中包含了 JAX-RS 2.0 标准，是 Java 领域 REST 式的 Web 服务的规范；GlassFish 是 Java EE 7 的参考实现项目集，Jersey 是其子项目，是 JAX-RS 2.0 的参考实现。本书的目的就是要把 JAX-RS 2.0 说清楚，把如何用 Jersey 写好 REST 式的服务讲明白。

本书特色

- 第一本完整讲述使用 Java 标准规范实现 REST 的书籍。
- 第一本完整讲述以 JAX-RS 2.0 参考实现实践 Jersey 的书籍。

- 给出深度学习和实践 JAX-RS 的线路图和解决方案。

读者对象

本书从实践角度，完整地诠释了 JAX-RS 2.0（JSR 339），即 Jersey 2.0 的核心元素和 REST 开发过程。面向所有在 Java 领域学习和使用 REST 的读者。同样欢迎 REST 领域的其他语言的使用者通过本书了解 REST 的实现。

- 技术路线：架构师、技术主管、研发工程师、REST 小白（网络用语，本书指新手）；
- 管理路线：部门经理、项目经理、产品经理；
- 敏捷实践者。

如何阅读本书

本书收纳了笔者近三年的 RESTful 实战经验，将 REST 理论与 Java 实现相结合，循序渐进地将使用 Java 开发 REST 式的 Web 服务中遇到的知识点和经验呈现给读者。每个章节中的知识点都精心设计了相应的示例代码，便于读者更好地理解和更及时地进行实践。从第 11 章开始，笔者从敏捷角度为读者呈现了一个完整和相对复杂的 REST 式的 Web 服务实例，相信这个实例能让读者更好地理解相关内容，同时，可以对敏捷开发和自动化测试有新的认识。

全书分为 3 篇，共 11 章。

第一篇共 5 章（第 1 ~ 5 章），讲述 REST 的基本理论和 Jersey 的基本实践。完成第一篇的阅读和示例代码的实践，读者可以学会使用 Java 开发 REST 式的 Web 服务的基本能力。

第 1 章 分别阐述了 REST、REST 式的 Web 服务、JAX-RS 2.0 和 Jersey 2.x 的基本情况。

第 2 章 讲述了使用 Jersey 2.x 开发一个基于 JAX-RS 2.0 标准的应用的基本知识以及如何使用 Jersey 来集成 Spring 和 JPA 以快速开发一个 REST 式的 Web 服务。本章包含 10 个示例项目。

第 3 章 深入阐述了如何使用 Jersey 设计和实现 REST 式的 Web 服务的 API。本章包含 5 个示例项目。

第 4 章 深入阐述了 Jersey 的 Providers 对 REST 请求的处理。

第 5 章 讲述了 Jersey 的客户端开发的基本实践和常用配置。

第二篇共 5 章（第 6 ~ 10 章），讲述写好 REST 程序的必要知识点。完成第二篇的阅读和实践，读者可以全面了解如何写好一个完整的 REST 式的 Web 服务。

第 6 章 全面讲述了如何实现一个安全的 REST 式的 Web 服务。

第 7 章 讲述了 Jersey 的测试框架及其使用。

第 8 章 讲述了 Jersey 对 HTML5 的 SSE 的支持和异步请求处理。

第 9 章 讲述了 Jersey 1.x 迁移到 Jersey 2.x 的要素和经验分享。

第 10 章 分享了 REST 式的 Web 服务的性能调优的经验。

第三篇包含 1 章（第 11 章），分享了 5 年外企工作中，我对自动化测试和敏捷的体会。完成本部分内容的阅读和实践，读者可以更宏观地审视 REST 的应用场景，起到抛砖引玉的作用。

第 11 章 讲述一个完整的 REST 项目的全过程。

全文由三个小版组成："阅读指南"、"小白讲堂"（为某些在知识点上比较"小白"的同学介绍概念性的知识）和"宅人坑事"（技术宅最自豪和最担惊受怕的就是"踩坑"），旨在和读者分享基础知识和心得体会，只为交流，切勿对号入座。

第一篇推荐研发工程师和 REST 小白完整阅读，这部分包含了了解和使用 JAX-RS 2.0 完成学习和工作的必要章节。对于有基础的技术人员，可以作为实践的参考有选择地阅读。

第二篇推荐致力于提高自己的技术人员完整阅读，这部分包含了 JAX-RS 2.0 的高级功能。永不满足和持续学习的精神，会让你在关键时刻成为"关键先生"。架构师和项目经理在考虑安全、性能等问题时，可以参考相关章节。

第三篇推荐渴望实战指导的技术人员阅读和跟随实践。同时，该篇结合了笔者参与敏捷实践的体会，以 scrum 的方式进行开发管理。因此，敏捷实践者和相关的部门经理可以参考。

产品经理可以阅读与 REST 特性相关的章节，这样可帮你在设计应用方面有所提高。

源代码

章节	源代码地址
第 1 章 ~ 第 10 章	https://github.com/feuyeux/jax-rs2-guide
第 11 章	https://github.com/feuyeux/jax-rs2-atup

勘误和交流

作为开发者，非常欢迎读者能与我一起交流 JAX-RS 2.0 相关的技术。我的邮箱是：feuyeux@gmail.com，新浪微博是：六爷 1_1。

本书的勘误统计在 https://github.com/feuyeux/jax-rs2-guide/wiki 中，欢迎读者批评指正。

致谢

感谢我的夫人 Caroline。写作期间，没有时间陪你共度周末，甚至没有精力陪你共度以 2 开头的最后一个生日，也没有营造任何浪漫的气氛。我们的女儿从蹒跚学步到会唱儿歌，我没有更多的时间陪伴你们，而你一如既往地支持我，在此我对你表示深深的感谢。

感谢华章公司的杨福川。我最难忘的是最初谈到要出本书时与君一拍即合的快意，这为我此后动笔增添了无穷的动力和信心。同时，阁下给予我的信任和鼓励也极大地推动了本书的顺利完成。在此后一年多的交往中，我深深感受到了业内对杨福川专业水准的评价和他敬业的态度。

感谢华章公司编辑高婧雅专业和耐心的审阅和指正。我曾一度被这位可敬的东北妹子的严谨和勤奋打击，心生愧对华章精品精神的挫败感。但随着婧雅的不断鼓励和支持，我一遍遍改进着稿件，自信心得到了恢复，同时，内心得到了安慰，我没有偏离初心和那份原动力。非常感谢华章公司和婧雅。

感谢 RedHat 公司姜宁师兄的技术校对，感谢阿里巴巴公司许晓斌的敏捷校对。

最后，我要感谢 Technicolor 敏捷团队中的每一位成员，正是与各位度过的每一天，才让我有资本和信心与广大读者分享在 REST 式的开发中的经验和教训，并在本书中介绍自动化测试和敏捷实践的相关知识。

Contents 目　录

推荐序一
推荐序二
前言

第一篇　够用就好——JAX-RS 2.0 基础

第 1 章　JAX-RS 2.0 入门 ····· 2
1.1　解读 REST ····· 2
1.1.1　一种架构风格 ····· 3
1.1.2　基本实现形式 ····· 3
1.2　解读 REST 服务 ····· 4
1.3　解读 JAX-RS ····· 5
1.4　Jersey 项目概要 ····· 8
1.5　Java 领域的其他 REST 实现 ····· 17
1.5.1　其他 JAX-RS 实现 ····· 18
1.5.2　其他 REST 实现 ····· 19
1.6　本章小结 ····· 20

第 2 章　JAX-RS 2.0 快速实现 ····· 21
2.1　第一个 Java REST 服务 ····· 21
2.1.1　环境准备 ····· 22
2.1.2　创建服务 ····· 23
2.1.3　扩展服务 ····· 28
2.1.4　测试和运行服务 ····· 31
2.2　第一个 Servlet 容器服务 ····· 32
2.2.1　创建和分析 Web 服务 ····· 32
2.2.2　Jetty 插件与 REST 服务 ····· 35
2.2.3　运行在 Servlet 容器 ····· 38
2.2.4　运行在 Java EE 容器 ····· 39
2.3　REST 服务类型 ····· 40
2.4　REST 应用描述 ····· 45
2.4.1　应用的描述 ····· 46
2.4.2　资源的描述 ····· 46
2.4.3　WADL 的配置 ····· 47
2.5　第一个完整的 REST 服务 ····· 48
2.5.1　定义资源 ····· 48
2.5.2　集成 Spring ····· 51
2.5.3　集成 JPA ····· 53
2.5.4　集成 jQuery ····· 55
2.5.5　请求处理流程分析 ····· 57
2.6　REST 调试工具 ····· 64

2.6.1 命令行调试工具 cURL ····· 64
2.6.2 基于浏览器的图形化调试插件 ··· 66
2.7 本章小结 ····· 69

第 3 章 REST API 设计 ····· 70

3.1 REST 统一接口 ····· 70
3.1.1 GET 方法 ····· 71
3.1.2 PUT 方法 ····· 73
3.1.3 DELETE 方法 ····· 75
3.1.4 POST 方法 ····· 76
3.1.5 WebDAV 扩展方法 ····· 77
3.2 REST 资源定位 ····· 79
3.2.1 资源地址设计 ····· 79
3.2.2 @QueryParam 注解 ····· 81
3.2.3 @PathParam 注解 ····· 83
3.2.4 @FormParam 注解 ····· 86
3.2.5 @BeanParam 注解 ····· 88
3.2.6 @CookieParam 注解 ····· 88
3.2.7 @Context 注解 ····· 89
3.3 REST 传输格式 ····· 90
3.3.1 基本类型 ····· 90
3.3.2 文件类型 ····· 90
3.3.3 InputStream 类型 ····· 91
3.3.4 Reader 类型 ····· 92
3.3.5 XML 类型 ····· 93
3.3.6 JSON 类型 ····· 96
3.4 REST 连通性 ····· 112
3.4.1 过渡型链接 ····· 113
3.4.2 结构型链接 ····· 114
3.5 REST 响应处理 ····· 114
3.5.1 返回类型 ····· 115
3.5.2 处理异常 ····· 117
3.6 REST 内容协商 ····· 119
3.6.1 @Produces 注解 ····· 119
3.6.2 @Consumes 注解 ····· 121
3.7 本章小结 ····· 122

第 4 章 REST 请求处理 ····· 123

4.1 REST 和 AOP ····· 123
4.2 Providers 详解 ····· 124
4.2.1 实体 Providers ····· 124
4.2.2 上下文 Providers ····· 129
4.3 REST 请求流程 ····· 130
4.4 REST 过滤器 ····· 132
4.4.1 ClientRequestFilter ····· 132
4.4.2 ContainerRequestFilter ····· 133
4.4.3 ContainerResponseFilter ····· 134
4.4.4 ClientResponseFilter ····· 135
4.4.5 访问日志 ····· 136
4.5 REST 拦截器 ····· 138
4.6 绑定机制 ····· 140
4.7 优先级 ····· 144
4.8 本章小结 ····· 145

第 5 章 REST 客户端 ····· 146

5.1 客户端接口 ····· 146
5.1.1 Client 接口 ····· 147
5.1.2 WebTarget 接口 ····· 148
5.1.3 Invocation 接口 ····· 148
5.2 资源释放 ····· 149
5.3 连接器 ····· 150
5.4 封装 Client ····· 153
5.5 本章小结 ····· 154

第二篇　全面掌握——JAX-RS 2.0 进阶

第 6 章　REST 安全 ····· 156

6.1　身份认证 ····· 157

6.1.1　基本认证 ····· 157

6.1.2　摘要认证 ····· 158

6.1.3　表单认证 ····· 158

6.1.4　证书认证 ····· 159

6.2　资源授权 ····· 160

6.2.1　容器管理权限 ····· 160

6.2.2　应用管理权限 ····· 163

6.3　认证与授权实现 ····· 163

6.3.1　基本认证与 JDBCRealm ····· 164

6.3.2　摘要认证与 UserDatabase-Realm ····· 170

6.3.3　表单认证与 DataSource-Realm ····· 173

6.3.4　表单认证与 JAASRealm ····· 177

6.3.5　证书认证与 UserDatabase-Realm ····· 180

6.4　JAX-RS 2.0 实现 ····· 184

6.5　其他安全考虑 ····· 187

6.6　本章小结 ····· 188

第 7 章　REST 测试 ····· 189

7.1　Jersey 测试框架 ····· 189

7.2　单元测试 ····· 192

7.2.1　集成 Spring 的单元测试 ····· 192

7.2.2　异步测试 ····· 194

7.3　集成测试 ····· 194

7.4　日志增强 ····· 195

7.5　本章小结 ····· 195

第 8 章　REST 推送与异步通信 ····· 196

8.1　服务器-浏览器通信 ····· 196

8.1.1　Polling 技术 ····· 197

8.1.2　Comet 技术 ····· 197

8.1.3　SSE 技术 ····· 199

8.1.4　WebSocket 技术 ····· 199

8.2　SSE 详述 ····· 200

8.2.1　Java 并发 ····· 200

8.2.2　SSE 流程 ····· 202

8.2.3　SSE 实现 ····· 204

8.3　异步通信 ····· 209

8.4　JAX-RS 2.0 实现异步通信 ····· 211

8.4.1　服务端实现 ····· 211

8.4.2　客户端实现和测试 ····· 213

8.5　本章小结 ····· 215

第 9 章　Jersey 1.x 迁移 ····· 216

9.1　变更 Maven 依赖定义 ····· 216

9.2　客户端迁移 ····· 217

9.2.1　Client 接口迁移 ····· 217

9.2.2　WebTarget 接口迁移 ····· 218

9.2.3　QueryParam ····· 219

9.3　服务器端迁移 ····· 219

9.4　本章小结 ····· 220

第 10 章　JAX-RS 调优 ····· 221

10.1　使用缓存优化负载 ····· 221

10.1.1　缓存协商 ····· 221

10.1.2　条件 GET ····· 223

10.1.3 REST 缓存实践 …… 224
10.1.4 ab 测试 …… 226
10.2 使用版本号优化服务 …… 226
10.2.1 何时使用版本号 …… 227
10.2.2 如何使用版本号 …… 227
10.3 使用参数配置优化服务 …… 229
10.3.1 通用配置 …… 229
10.3.2 服务器端配置 …… 230
10.3.3 客户端配置 …… 231
10.4 Java 虚拟机调优 …… 232
10.4.1 虚拟机概述 …… 232
10.4.2 内存溢出与内存泄漏 …… 235
10.5 本章小结 …… 236

第三篇 实践分享——JAX-RS 2.0 综合

第 11 章 统一自动化测试平台 …… 238
11.1 ATUP 的定义 …… 238
11.1.1 需求仓库 …… 239
11.1.2 需求分析 …… 241
11.1.3 迭代规划 …… 242
11.2 ATUP 的设计 …… 244
11.2.1 开发和部署环境 …… 244
11.2.2 模块定义和拓扑 …… 247
11.2.3 持续集成流程 …… 248
11.3 ATUP 的实现 …… 250
11.3.1 Sprint1 核心功能 …… 250
11.3.2 Sprint2 模块功能 …… 281
11.3.3 Iteration1 的演示和回顾 …… 288
11.3.4 Sprint3 持续交付 …… 291
11.3.5 交付和总结 …… 293
11.4 本章小结 …… 293

附录 Web 简史 …… 294

参考资料 …… 297

后记 …… 298

第一篇 *Part 1*

够用就好——JAX-RS 2.0 基础

- 第 1 章　JAX-RS 2.0 入门
- 第 2 章　JAX-RS 2.0 快速实现
- 第 3 章　REST API 设计
- 第 4 章　REST 请求处理
- 第 5 章　REST 客户端

Chapter 1 第 1 章

JAX-RS 2.0 入门

本章逐一介绍以下 4 个概念：REST、REST 式的 Web 服务、JAX-RS 标准和 Jersey 项目。首先简要介绍一下这 4 个概念之间的联系：REST 是一种跨平台、跨语言的架构风格，REST 式的 Web 服务是对 REST 在 Web 领域的实现；JAX-RS 标准是在 Java 领域，对 REST 式的 Web 服务制定的实现标准，Jersey 是 JAX-RS 标准的参考实现，是 Java EE 参考实现项目 GlassFish 的成员项目。

1.1 解读 REST

REST（Representational State Transfer，表述性状态转移），源于 REST 之父 Roy Thomas Fielding 博士在 2000 年就读加州大学欧文分校期间发表的一篇学术论文——《Architectural Styles and the Design of Network-based Software Architectures》[⊖]。论文中提出了 REST 的 6 个特点，分别是：客户端 - 服务器的、无状态的、可缓存的、统一接口、分层系统和按需编码。在此不敢“班门弄斧”，希望读者阅读一下这部开山之作，以此理解 REST 产生的原因和特点。

REST 具有跨平台、跨语言的优势。从其诞生开始，广泛地得到了诸多语言的快速支持，比较著名的是 ROR（Ruby on Rails）。ROR 非常强调规约大于配置的理念，这一理念被诸多框架和工具支持——包括以 Python 领域的 Django、Groovy 领域的 Grails 为代表的框架，以及 Maven、Gradle 为代表的构建工具等，但 REST 本身只规定了面向资源，并没有包含如何定义和约束一个资源的标准。因此要提示读者，很多实现了 REST 的框架和工具所具备的特质，其全部特征未必都是 REST 定义的，包括本书要讲的 JAX-RS 标准的定义和 Jersey 等 JAX-RS 的实现。

⊖ 该论文的中文版译于 2007 年，中文名为《架构风格与基于网络的软件架构设计》，此后再次修订，名为《架构风格与基于网络应用软件的架构设计（中文修订版）》，详见本书的参考资料。

REST 在众多平台和语言上的支持，不仅包括上述列举的传统编程语言、脚本语言，还包括 Node.js 这样新兴的服务器端脚本语言。REST 的无状态，代理友好性等优势，使对其支持的实现更加方便和简洁。

阅读指南　至于为什么 REST 的出现影响了今天的互联网，以及 Web 的发展历程，可参阅附录中的 Web 简史。

1.1.1　一种架构风格

REST 是一种架构风格。在 REST 架构风格中，对象被抽象为一种资源（resource），资源的命名使用概念清晰的名词来定义。表述性状态是对资源数据在某个瞬间状态的快照，资源的某个瞬时状态被定义为一种表述（representation），这种描述性的状态包括资源数据的内容、表述格式（比如 XML、JSON、Atom）等信息，一种资源可以对应多种表述。REST 的资源是可寻址的，通过 HTTP 协议（RFC 2616）定义的通用动词方法（比如 GET、PUT、DELETE、POST），并使用 URI 协议（RFC 3305）来唯一标识某个资源公布出来的接口。请求一个资源的过程可以理解为，访问一个具有指定性和描述性的 URI，经由 HTTP，将资源的表述从服务器转移到客户端，或者相反方向。

REST 不是一种技术（technology），也不是一个标准（standard）或协议（protocol），它使用既有标准：HTTP+URI+XML[⊖]，来实现其要求的架构风格。因此，与之对应的不是 SOAP，而是像 RPC 这样的架构风格。

1.1.2　基本实现形式

HTTP+URI+XML 是 REST 的基本实现形式，但不是唯一的实现形式。REST 从推出时就使用已有的 HTTP 协议、URI 协议来描述，而对如何使用一种编程语言来实现，并没有进行任何描述和规定，甚至对于 REST 应该包含哪些传输类型或者数据格式也没有描述，但通常的实现至少包含 XML 格式。具体而言，HTTP 协议和 URI 用于统一接口和定位资源，文本、二进制流、XML 和 JSON 等格式用来作为资源的表述。正如采用已有技术 XMLHttpRequest+JavaScript+XML（XML 后来几乎被 JSON 替代）实现 AJAX 一样，使用 HTTP+URI+XML 实现 REST 的好处是让开发者持有这些已知的技术来开发 REST，入门门槛较低，关注点更容易放到 REST 的核心概念和业务逻辑上。

这里要提示读者的是，以 HTTP+URI+XML 实现的应用并不一定是 REST 式的，但对于 AJAX，这个逆命题（以 XMLHttpRequest+JavaScript+XML 实现的应用一定是 AJAX 应用）是成立的。因为 AJAX 是一种“技术流派”，而 REST 是一种架构风格。学习和使用 REST 的关键是掌握这种思想，而不是具体的实现形式。

⊖ XML 似乎成为了数据格式的借指，不仅代表 XML 本身。

1.2 解读 REST 服务

英文 RESTful 对应的中文是 REST 式的。RESTful 的应用或者 Web 服务是常见的两种 REST 式的项目部署、存在的方式。本节将介绍 RESTful Web Services 并对比和传统 Web Services 的不同。

1. REST 式的 Web 服务

RESTful Web Services（REST 式的 Web 服务）是一种遵守 REST 式风格的 Web 服务。REST 式的 Web 服务是一种 ROA（Resource-Oriented Architecture，面向资源的架构）的应用。其主要特点是方法信息存在于 HTTP 的方法中（比如 GET、PUT），作用域存在于 URI 中。例如，在一个获取设备资源列表的 GET 请求中，方法信息是 GET，作用域信息是 URI 中包含的对设备资源的过滤、分页和排序等条件。

2. 对比 RPC 风格

相比 Web 服务领域广为流行的 RPC（Remote Procedure Call，远程过程调用）风格，REST 风格更轻量和快速。从方法信息角度看，REST 采用标准的 HTTP 方法，而 RPC 请求都是 HTTP 协议的 POST 方法，其方法信息包含在 SOAP 协议包或 HTTP 协议包中，方法名称不具有通用性。从作用域角度看，REST 采用 URI 显式定义作用域，而 RPC 的这一信息同样包含于协议包中，不能直观呈现。

RPC 风格的开发关注于服务器 - 客户端之间的方法调用，而不关注基于哪个网络层的哪种协议。也就是说，RPC 是面向方法调用过程的，相比而言，REST 是面向资源状态的。RPC 风格的两个代表是 XML-RPC 和大 Web 服务。

（1）XML-RPC

XML-RPC 是一种使用 XML 格式封装方法调用，并使用 HTTP 协议作为传送机制的 RPC 风格的实现。XML-RPC 的请求方法都是 HTTP 协议的 POST 方法，请求和响应的数据格式均为 XML。

XML-RPC 的数据格式和使用 XML 作为资源的表述的 REST 外观上很相似，但数据的内容则大相径庭。REST 式的 XML 信息的主体是对一个资源状态的表述，无须包含方法信息，因为其请求的 HTTP 方法就已经决定了这一点。XML-RPC 的请求数据结构额外包含方法调用信息和参数信息。

对于响应信息的内容，两者也截然不同，REST 式的通常会包含响应实体信息以及 HTTP 状态码和可选的异常信息，而 XML-RPC 的返回信息仅仅是对方法调用的响应信息。

XML-RPC 是一种遗留技术，已经被 SOAP 取代。在 Java 领域，JAX-RPC 标准已经被 JAX-WS 取代。XML-RPC 的应用依然存在，著名的测试用例管理系统 TestLink 的对外接口就是使用 PHP 开发的 XML-RPC。

（2）大 Web 服务

大 Web 服务（Big Web Services）是 Leonard Richardson 和 Sam Ruby 在其所著的《RESTful

Web Services》一书中，对基于 SOAP+ WSDL+UDDI+WS- 标准栈等技术实现 RPC 风格的大型 Web 服务的统称。事实上，“大 Web 服务”这一说法也被 Java EE 7 的布道者在多次演讲中使用。在 Java 领域，对应的标准主要是 JAX-WS 2.0/2.1/2.2（JSR 224）。相较 REST 式的 Web 服务，大 Web 服务功能更强大、设计更复杂。大 Web 服务同样是跨平台、跨语言的，对复杂的数据类型的支持也非常好。大 Web 服务是基于 RPC 风格的重量级的设计，因此方法和作用域无法通过直观断定，需要定义在消息中，而且方法名不是统一和通用的。同时，大 Web 服务走 HTTP 协议时，请求都是基于 POST 方法的。

对比 RPC 风格的 Web 服务，REST 式的 Web 服务形式更简单、设计更轻量、实现更快捷。REST 无须引入 SOAP 消息传输层，无须注册服务，也没有客户端 stub 的概念等。但是，REST 风格的 Web 服务并没有像大 Web 服务那样提供诸如安全策略等全面的标准规范。

大 Web 服务和 REST 式的 Web 服务各有其优势，并不是一种代换关系。在实际开发中，两者共存于一个项目中也是一种解决方案。

3. 对比 MVC 风格

MVC 风格的出现将模型、视图、控制解耦，其亮点是从前到后的一致性，其结构简洁、逻辑清晰，易于扩展和增强。MVC 在 Java 领域的普遍实现方式是在 Web 前端使用标签库来对应服务端的模型类实例和控制类实例，标签库和服务端依赖库可以是松散的耦合——比如 Spring 生态系统，也可以是全栈式的统一体系，比如 JSF 体系。但无论如何实现，在 Web 前端的开发过程中，必须时刻考虑页面标签和服务端的映射关系，包括模型类的匹配和转换，数据结构，控制类输入和输出的参数类型和数量等。

因此，MVC 风格偏重于解决服务器端的逻辑分层，客户端是逻辑分层的延伸。MVC 的标签库，虽然其形态已经和 HTML 页面融合，但本质上还是 Java 编写的装饰模式的类实例，对应的是服务器端使用 Java 编写的模型类或者控制器类，因此 MVC 很难实现跨语言解耦。而 REST 风格偏重于统一接口，因此具体实现就可以跨平台和跨语言。正如附录中 Web 简史中所述，REST 推动了 Web 开发的新时代，使用纯 HTML 作为客户端，没有服务器端和客户端的耦合。显而易见，使用纯 HTML 开发的 REST 客户端和使用 Java 开发的 REST 服务器端并不存在语言上的耦合。

MVC 和 REST 式并不是互斥的，像 Spring 的 MVC 模块已经支持 REST 式的开发。Jersey 作为 JAX-RS 标准的实现，也实现了 MVC 的功能，可参考相关模块：jersey-mvc、jersey-mvc-freemarker 和 jersey-mvc-jsp。需要说明的是，本书致力于讲述 JAX-RS，对 Jersey 实现中的 JAX-RS 之外的功能只做必要的讲述。由于 MVC 和 REST 之间有更多的并行存在性，因此本书余文没有将 MVC 放入讲述之列。

1.3　解读 JAX-RS

JAX-RS 是 Java 领域的 REST 式的 Web 服务的标准规范。如果我们只学习工具的使用而

不知标准是怎样定义的，那么可能不会理解其本质。因此，我们首先来学习和掌握 JAX-RS 规范，这是使用 Java 完成 REST 式的 Web 服务的基础和前提。

1. JAX-RS2 标准

Java 领域中的 Web Services 是指实现 SOAP 协议的 JAX-WS。直到 Java EE 6（发布于 2008 年 9 月）通过 JCP（Java Community Process）组织定义的 JSR 311（http://www.jcp.org/en/jsr/detail?id=311），才将 REST 在 Java 领域标准化。JSR 311 名为 The Java API for RESTful Web Services，即 JAX-RS，其参考实现是 GlassFish 项目中的 Jersey 1.0。此后，JSR 311 进行了一次升级（2009 年 9 月），即 JAX-RS 1.1。JAX-RS 诞生后，时隔 5 年（2013 年 5 月）发布的 Java EE 7 包含了 JSR 339，将 JAX-RS 升级到 JAX-RS 2.0（http://www.jcp.org/en/jsr/detail?id=339）。JAX-RS 2.0 在前面版本的基础上增加了很多实用性的功能，比如对 REST 客户端 API 的定义、异步 REST 等，对 REST 的支持更加完善。

JAX-RS 的版本对应的参考实现的 Jersey 项目版本信息参见表 1-1。

表 1-1 JAX-RS 标准和 Jersey 版本信息

JAX-RS 标准	JAX-RS 名称	标准发布时间	JAX-RS 实现
311	JAX-RS 1.0	2008 年 9 月 8 日	Jersey 1.x
311	JAX-RS 1.1	2009 年 9 月 17 日	
339	JAX-RS 2.0	2013 年 5 月 22 日	Jersey 2.x

2. JAX-RS 2.0 的目标

JAX-RS 2.0 标准（即 JSR 339）中定义了目标、非目标和元素等内容。JSR 339 标准中的这部分内容通常被以实现业务功能为目的的开发人员所忽视，在此和读者分享的一个开发经验是，要掌握一项技术，要先掌握它背后标准的定义。我们首先来看看 JAX-RS 2.0 的目标。

1）基于 POJO：JAX-RS 2.0 的 API 提供一组注解（annotation）和相关的接口、类，并定义了 POJO（Plain Ordinary Java Objects）对象的生命周期和作用域。规定使用 POJO 来公布 Web 资源。

2）以 HTTP 为中心：JAX-RS 2.0 采用 HTTP 协议，并提供清晰的 HTTP 和统一资源定位（URI）元素来映射相关的 API 类和注解。JAX-RS 2.0 的 API 不但支持通用的 HTTP 使用模式，还对 WebDAV 和 Atom 等扩展协议提供灵活的支持。

3）格式独立性：JAX-RS 2.0 对传输数据（HTTP Entity）的类型 / 格式的支持非常宽泛，允许在标准风格之上使用额外的数据类型。

4）容器独立性：JAX-RS 2.0 的应用可以部署在各种 Servlet 容器中，比如 Tomcat/Jetty，也可以部署在支持 JAX-WS 的容器中，比如 GlassFish。

5）内置于 Java EE：JAX-RS 2.0 是 Java EE 规范的一部分，它定义了在一个 Java EE 容器内的 Web 资源类的内部，如何使用 Java EE 的功能和组件。

小白讲堂

WebDAV（Web-based Distributed Authoring and Versioning，基于 Web 的分布式创作和版本控制）是 IETF 组织的 RFC 2518 协议。WebDAV 基于并扩展了 HTTP 1.1，在 HTTP 标准方法以外添加了：

- Mkcol：创建集合；
- PropFind/PropPatch：针对资源和集合检索和设置属性；
- Copy/Move：管理命名空间上下文中的集合和资源；
- Lock/Unlock：改写保护，支持文件的版本控制。

针对在 REST 风格的 Web 服务中是否应该使用 WebDAV，业内的声音并不一致，持反对意见的人的主要观点是 WebDAV 带来了非统一的接口，这违背了 REST 的初衷。本书的示例将不采用 WebDAV，但文字部分将讲述如何支持 WebDAV。Atom 类型传输格式将在 3.3 节讲述。

3. 非 JAX-RS 2.0 的目标

那么哪些不是 JAX-RS 2.0 的目标呢？

1）对 J2SE 6.0 之前版本的支持：JAX-RS 2.0 中大量使用了注解（annotation），需要 J2SE 6.0 以及更新的版本，因此不提供对 J2SE 6.0 以下版本的支持。

2）对服务的描述、注册和探测：JAX-RS 2.0 没有定义也无须支持任何服务的描述（description）、服务的注册（registration）和服务的探测（discovery）。

3）HTTP 协议栈：JAX-RS 2.0 没有定义新的 HTTP 协议栈。承载 JAX-RS 2.0 应用的容器提供对 HTTP 的支持。

4）数据类型 / 格式类：JAX-RS 2.0 没有定义处理实体内容的类，它将这一类型的类交由使用 JAX-RS 2.0 的应用中的类去实现。

4. 解读 JAX-RS 元素

最后，我们来看看 JAX-RS 2.0 中定义了哪些元素。

1）资源类：使用 JAX-RS 注解来实现相关 Web 资源的 Java 类。如果用 MVC 的三层结构来解读，那么资源类位于最前端，用于接收请求和返回响应。通常但不是约定使用 resource 作为包名，三层的包定义形如：resource-service-dao。

2）根资源类：使用 @Path 注解，提供资源类树的根资源及其子资源的访问。资源类分为根资源类和子资源类。由于 Jersey 默认提供 WADL（参见 2.4 节），因此每个应用公布的全部资源接口可以通过 WADL 页面查阅。

3）请求方法标识符：使用运行期注解 @HttpMethod，用来标识处理资源的 HTTP 请求方法。该方法将被资源类的相应方法处理，标准的方法包括 DELETE、GET、HEAD、OPTIONS、POST、PUT，详见 3.1 节。

4）资源方法：资源类中定义的方法，使用了请求方法标识符，用来处理相关资源的请

求。就是上面提到的资源类的相应方法。

5）子资源标识符：资源类中定义的方法，用来定位相关资源的子资源。

6）子资源方法：资源类中定义的方法，用来处理相关资源的子资源的请求。

7）Providers：一种 JAX-RS 扩展接口的实现类，扩展了 JAX-RS 运行期的能力。第 4 章详述了各种 Providers 及其实现。

8）Filter：一种用于过滤请求和响应的 Provider，详见 4.4 节。

9）Entity Interceptor：一种用于处理拦截消息读写的 Provider。

10）Invocation：一种用于配置发布 HTTP 请求的客户端 API 对象，详见 5.1.3 节。

11）WebTarget：一种使用 URI 标识的 Invocation 容器对象，详见 5.1 节。

12）Link：一种携带元数据的 URI，包括媒体类型、关系和标题等，详见 3.4 节。

1.4 Jersey 项目概要

Jersey 是 JAX-RS 标准的参考实现，是 Java 领域中开发 REST 式的 Web 服务的“正统”工具。这样讲述的目的是避免读者混淆 REST 领域诸多工具的地位，并无偏好推荐之意。本节将带领读者走进 Jersey 的世界。

Jersey 项目是 GlassFish 项目的一个子项目，专门用来实现 JAX-RS（JSR 311 和 JSR 339）标准，并提供了扩展特性。

1. 获得 Jersey

Jersey 项目的网址是 https://jersey.java.net，这是 Jersey 的官方网站。该网站同时提供了 JAX-RS 和 JAX-RS 2.0 两个并行版本，分别是 Jersey 1.1（截至本书发稿，最新版本是 Jersey 1.18）和 Jersey 2.0（截至本书发稿，最新版本是 Jersey 2.9）。读者可以通过单击“latest Jersey User Guide”获取和阅读最新版本的用户手册，这是官方发布的第一手的参考资料。

Jersey 项目的下载地址 https://jersey.java.net/download.html，Jersey 项目使用 Maven 管理项目，该页面至上而下分别是：

- JAX-RS 标准列表链接。
- Jersey 最新参考实现的 jar 包下载。
- Jersey 最新参考实现的示例代码下载。
- 创建基于 Maven 管理的 JAX-RS 容器 Jersey 应用。
- 创建基于 Maven 管理的 Servlet 容器 Jersey 应用。
- JAX-RS 1.1 的参考实现包下载。

2. Jersey 源代码

Jersey 2 使用 GitHub 管理源代码，GitHub 的客户端非常简洁。单击首页下方的“Source Control”进入 https://jersey.java.net/scm.html 页面，该页面提供了如何使用 GIT 来克隆 Jersey

的源代码的介绍。Jersey 源代码的托管地址是 https://github.com/jersey/jersey（见图 1-1）。

如图 1-1 所示，Jersey 项目的源代码是使用 Maven 构建的多模块项目。每个目录代表一个模块，比如 core-server 目录就是 Jersey 的 Server 模块。bom 目录是个特殊的目录，是 Maven 的元数据目录，用于多模块 Maven 项目的组织和定义。通过浏览该页面可以观察到 Jersey 框架的模块，本章随后会介绍 Jersey 的各个模块。第 2 章将对基于 Maven 管理的 Jersey 应用做详细的讲述和演示。

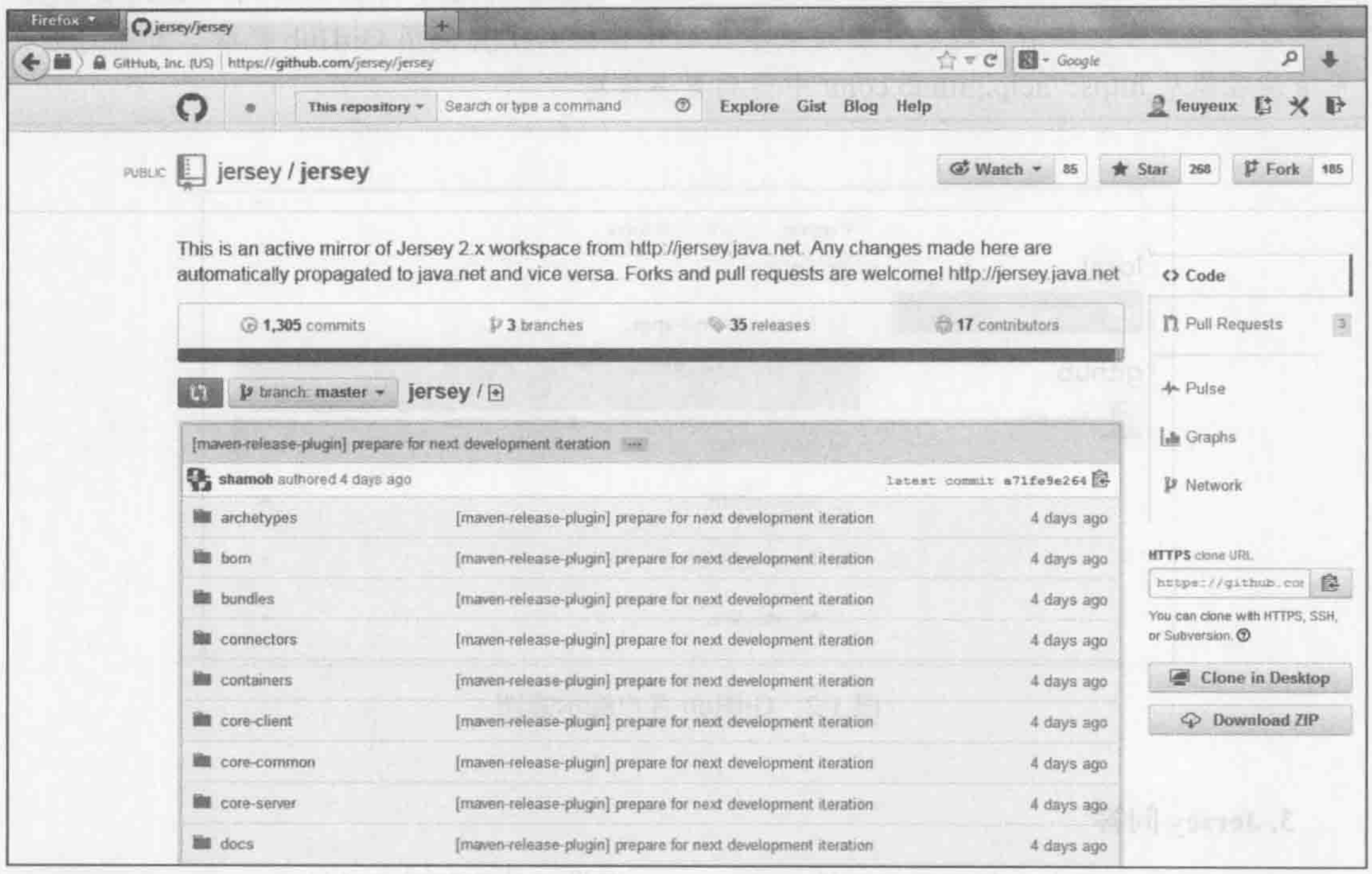


图 1-1　Jersey 源代码仓库

小白讲堂

Maven（http://maven.apache.org/）是 Apache 开源组织的项目，该项目用于构建和组织 Java 的项目。读者如果对 Maven 不熟悉，推荐阅读许晓斌（Juven Xu）所著的《Maven 实战》(机械工业出版社出版)。

如果读者对命令行不熟悉或者没有兴趣，可以使用 GUI 工具来完成源代码的管理。对于托管在 GitHub 的项目，可以使用 GitHub 官方提供的客户端来管理源代码，Windows 版本下载地址：http://windows.github.com，Mac 版本下载地址：http://mac.github.com。这款工具的使用非常简单，如图 1-2 所示，左侧树包括本地仓库和 GitHub 远程仓库两个结点。右侧是仓库列表，示意图中包括本书的示例项目 jax-rs2-guide、第 11 章示例项目 jax-rs2-atup 和 Jersey

源代码等仓库。需要指出的是，其处理版本冲突的能力不足，需要借助命令行来完成。

小白讲堂

GIT 是一种开源的分布式 SCM（Source Code Management，源代码管理）工具，下载地址为 http://git-scm.com/download。读者如果对 GIT 不熟悉，推荐阅读蒋鑫所著的《Git 权威指南》(机械工业出版社出版)。

GitHub 是 GIT 服务器中比较知名和成功的，它提供了基于身份认证的项目托管等功能。时下非常多的知名开源软件都将版本控制迁移到 GIT 并使用 GitHub 托管。如果可以，希望读者能从 https://help.github.com/ 中得到更多信息。

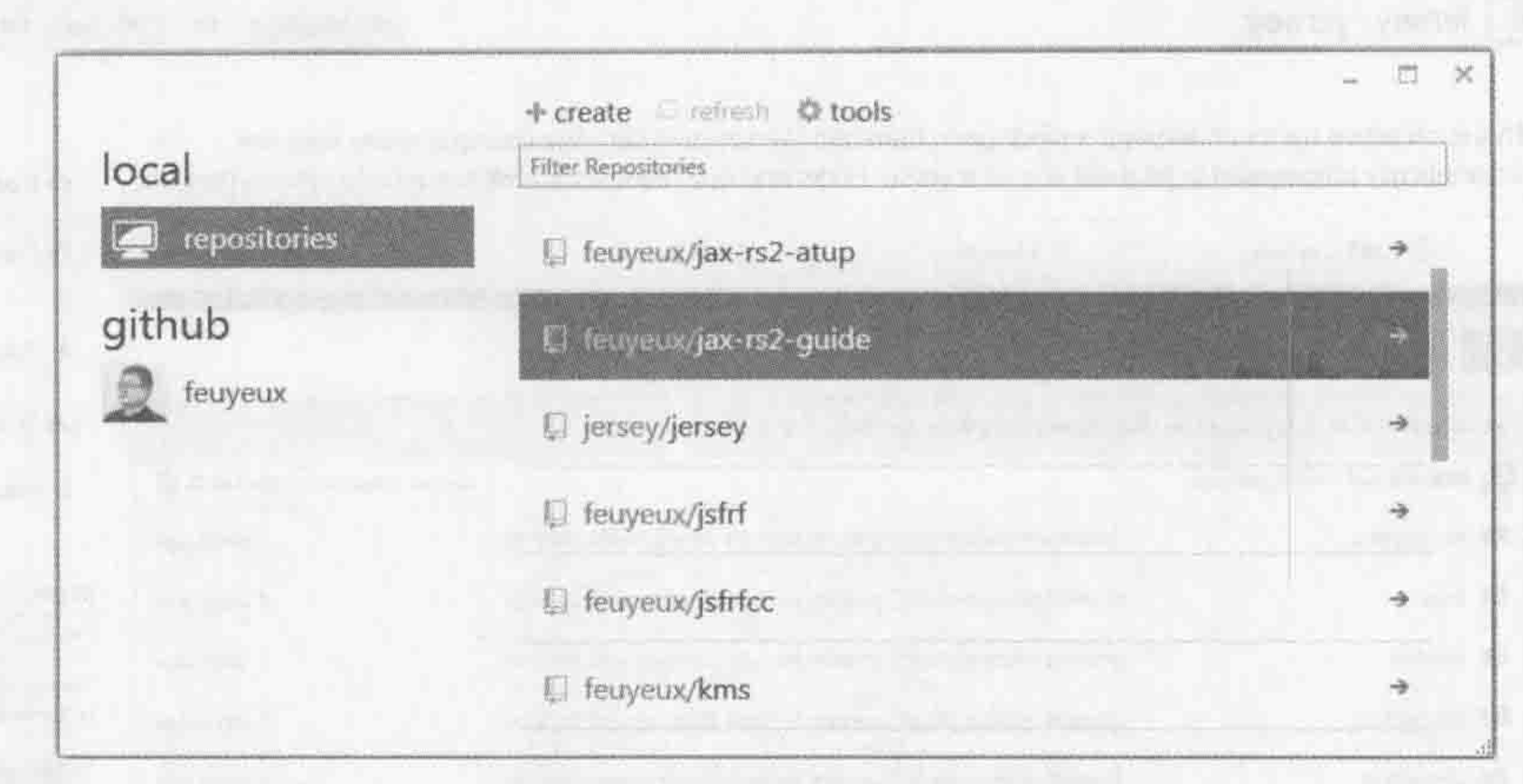


图 1-2 GitHub 客户端示意图

3. Jersey 问答

StackOverflow 是专业的程序员问答系统，Jersey 提供了其问题列表地址：http://stackoverflow.com/questions/tagged/jersey。从中可以看出开源社区对问答系统的重视。另外，邮件列表也是一种知识共享的途径，读者可以自行订阅，地址是：https://jersey.java.net/mailing.html。

实践趣事

这里想和读者分享的是，一名不断成长的程序员[⊖]一定是勤奋和富有热情的，这是开源社区和 StackOverflow 这样的问答系统的生存基石。社区和个人的成长是相辅相成的。推荐读者在遇到问题时，第一个想到的是搜 Google、搜 StackOverflow，而不是张嘴就问，问周边的人应该是正确做法中最后的办法。顺便要说的是，英语是程序员绕不开的技能，一起加油吧！

⊖ 与年龄无关，个人很厌恶 30 岁分界的观点。

4. Jersey 项目管理

Jersey 使用 JIRA 作为项目管理平台，地址是 https://java.net/jira/browse/JERSEY。JIRA 和 StackOverflow 不同的是，这个平台是 Jersey 团队日常开发的管理平台，也就是说，这是 Jersey 官方的缺陷管理平台，用于上报缺陷和改进意见，而不是社区性质的交流平台。我们可以从中了解到 Jersey 项目的进展情况。Jersey 是一个非常活跃的项目，不仅可以从 GitHub 的源代码的提交活动中看到该项目频繁的更新，在 JIRA 中，也可以看到该项目推进的速度。

宅人坑事

这里为喜欢开源社区活动的读者举个例子。我在撰写本书的开始，Jersey 2.0 并不支持与 Spring 的集成，因为 Jersey 的 IoC 容器由 GlassFish 的另一个子项目 HK2 来支持。随后，我在 JIRA 上发现一个 Jersey 2.x 支持与 Spring 集成的任务被创建了（https://java.net/jira/browse/JERSEY-1957），此后我经常观察其进展状态，最终看到了这个功能在 Jersey 2.2 中以扩展包的形式发布了。因此，当使用 Jersey 时，如果遇到 Jersey 本身的问题，可以跟踪 Jersey 的 JIRA 平台来查看 Bug 的修复状态，包括将在哪个版本修复，有什么样的临时解决办法（workaround）。同时，跟踪 JIRA 也可以了解新版本的发布情况，包括新增哪些功能，升级给哪一部分带来性能、安全的提升等。换句话说，JIRA 展示了 Jersey 项目的缺陷修复和新功能发版的计划（roadmap）。

---

5. Jersey 许可

开发者使用开源软件的前提是了解它的许可证版本，否则可能会带来侵权问题。相信在正规的公司，大家都有被开发管理部门的人“恐吓”的经历。开发者需感谢开发管理部门所做的工作，他们为公司规避了商业侵权的风险，因为引用的源代码如果出自“传染性”许可，该项目是不能用于闭源的商业用途的。

Jersey 的许可证说明地址是：https://jersey.java.net/license.html。从中我们可以了解到 Jersey 使用的是双许可证：CDDL（Common Development and Distribution License，开源通用开发和分发许可证）1.1 和 GPLv2（类路径例外）许可证。双重许可是依照两套（或更多套）不同的条款和条件分发相同软件的做法。在为软件授予双重许可时，接收人可以选择他们希望依照哪种条款获得软件。使用双重许可的两个常见动机是遵循商业模式和保持许可证兼容性。GPLv2.0 许可证为无法依照 CDDL 许可证使用 Jersey 的供应商提供了一个额外选项。Jersey 许可证使整套产品和包保持一致（GlassFish 项目同样依照 CDDL 和 GPLv2（类路径例外）授予双重许可）。

6. Jersey 的模块

Jersey 框架是由核心模块、容器模块、连接器模块、Media 模块、扩展模块、测试框架模块和 GlassFish Bundle 模块 7 个大的模块组成。这不是一成不变的，随着 Jersey 的不断发展和完善，会有新的模块加入进来。推荐读者浏览官方文档，最新版本的地址为：https://

jersey.java.net/documentation/latest/modules-and-dependencies.html，从而获得最新的信息。

小白讲堂

类路径例外是由自由软件基金会的 GNU/ 类路径项目制定的。它允许用户将依照任何许可证提供的应用程序链接到依照 GPLv2 许可的软件中包含的库，而该应用程序不受 GPL 要求公开其本身的限制。为什么需要使用类路径例外？因为作为“基于 [GPL] 程序的作品”的一部分提供的所有代码还应获得 GPL 许可。因此，需要指定 GPL 许可证例外情况，以便明确将链接到 GPL 实现的任何应用程序从该许可要求中排除。类路径例外实现了这一目的。

(1) 核心模块

Jersey 核心模块由通用包、服务器实现和客户端实现 3 个子模块组成，是使用 Jersey 必需的依赖，见表 1-2。

表 1-2 Jersey 核心模块列表

模块名称	模块说明	源代码相对目录
jersey-client	Jersey 核心客户端实现	core-client
jersey-common	Jersey 通用包	core-common
jersey-server	Jersey 核心服务器实现	core-server

(2) 容器模块

Jersey 提供了 3 种对应于外部容器的 HTTP 容器，分别是 Grizzly 2、JDK-HTTP 和 SIMPLE-HTTP，Grizzly 2 同时提供了 Servlet 容器。对于 Servlet 的实现，Jersey 提供了对 Servlet 2.x 和 Servlet 3.x 两个版本的实现。它们对应的包名见表 1-3，详情可参考第 2 章。

表 1-3 Jersey 容器模块列表

模块名称	模块说明	源代码相对目录
jersey-container-grizzly2-http	Grizzly 2 版 HTTP 容器	containers/grizzly2-http
jersey-container-grizzly2-servlet	Grizzly 2 版 Servlet 容器	containers/ grizzly2-servlet
jersey-container-jdk-http	JDK 版 HTTP 容器	containers/ jdk-http
jersey-container-servlet	Jersey 核心 Servlet 3.x 实现	containers/ jersey-servlet
jersey-container-servlet-core	Jersey 核心 Servlet 2.x 实现	containers/ jersey-servlet-core
jersey-container-simple-http	简单版 HTTP 容器	containers/simple-http

(3) 连接器模块

Jersey 客户端底层依赖连接器实现网络通信，如果标准的客户端模块无法实现功能需求，可以考虑引入 Grizzly 连接器或者 Apache 连接器，见表 1-4，详情可参考第 5 章。

表 1-4 Jersey 连接器模块列表

模块名称	模块说明	源代码相对目录
jersey-grizzly-connector	Jersey 客户端 Grizzly 连接器	connectors/grizzly-connector
jersey-apache-connector	Jersey 客户端 Apache 连接器	connectors/apache-connector

（4）Media 模块

Jersey Media 模块是支持 Jersey 处理传输数据媒体类型的模块。该模块提供了 4 种处理 JSON 的方式，分别是 Jackson、Jettison、MOXy 和 JSON-P。另外，Multipart 包用于对 Form 表单的支持，SSE 用于对 Server Sent Events 支持，见表 1-5，详情可参考第 3 章。

（5）扩展模块

Jersey 生态环境中包含了许多 JAX-RS 2.0 标准之外的功能，比如 MVC、Bean 验证等辅助 REST 实现的模块，还有像 Spring 支持包这样的对第三方框架支持的模块见表 1-6。

表 1-5 Jersey Media 模块列表

模块名称	模块说明	源代码相对目录
jersey-media-json-jackson	Jersey JSON Jackson 包	media/json-jackson
jersey-media-json-jettison	Jersey JSON Jettison 包	media/json-jettison
jersey-media-json-processing	Jersey JSON-P（JSR 353）包	media/json-processing
jersey-media-moxy	Jersey JSON EclipseLink MOXy 包	media/moxy
jersey-media-multipart	Jersey Multipart 支持包	media/multipart
jersey-media-sse	Jersey Server Sent Events 支持包	media/sse

表 1-6 Jersey 扩展模块列表

模块名称	模块说明	源代码相对目录
jersey-bean-validation	Bean Validation（JSR-349）支持包	ext/bean-validation
jersey-mvc	MVC 支持包	ext/mvc
jersey-mvc-freemarker	Freemarker 模板支持包	ext/mvc-freemarker
jersey-mvc-jsp	JSP 模板支持包	ext/mvc-jsp
jersey-proxy-client	高层客户端代理支持包	ext/proxy-client
jersey-servlet-portability	兼容 Jersey 1.x 和 Jersey 2.x 的 Servlet 容器支持包	ext/servlet-portability
jersey-wadl-doclet	Javadoc 支持包	ext/wadl-doclet
jersey-spring3	Spring 3 支持包	ext/spring3

（6）测试框架模块

Jersey 提供了非常易用且功能强大的测试框架，见表 1-7。测试框架提供了测试过程中完整的服务器端和客户端功能，通过测试框架，开发者可以减少很多模板代码的编写。详情可参考 7.1 节。

表 1-7 Jersey 测试框架模块列表

模块名称	模块说明	源代码相对目录
jersey-test-framework-core	Jersey 核心测试框架	test-framework /core
jersey-test-framework-provider-bundle	测试框架 Bundle 支持包	test-framework/providers/bundle
jersey-test-framework-provider-default-client	测试框架客户端包	test-framework/providers/default-client
jersey-test-framework-provider-external	测试框架扩展容器	test-framework/providers/external
jersey-test-framework-provider-grizzly2	测试框架 Grizzly 2 容器	test-framework/providers/grizzly2

（续）

模块名称	模块说明	源代码相对目录
jersey-test-framework-provider-inmemory	测试框架内存容器	test-framework/providers/inmemory
jersey-test-framework-provider-jdk-http	测试框架 JDK 版 HTTP 容器	test-framework/providers/jdk-http
jersey-test-framework-provider-simple	测试框架简单版 HTTP 容器	test-framework/providers/simple

（7）GlassFish Bundle 模块

GlassFish Bundle 模块是 Jersey 提供的用于以 Bundle 方式支持 GlassFish 服务器的模块，包括 CDI 和 EJB 集成的扩展包，见表 1-8。

表 1-8 GlassFish Bundle 模块列表

模块名称	模块说明	源代码相对目录
jersey-gf-cdi	Jersey 集成 GlassFish 的 CDI 包	containers\glassfish\jersey-gf-cdi
jersey-gf-ejb	Jersey 集成 GlassFish 的 EJB 包	containers\glassfish\jersey-gf-ejb

宅人坑事

Jersey 在 2.6 版本做了一次包重构，清除了对 Guava 和 ASM 的自然依赖。如果读者的项目需要做 Jersey 版本迁移，需要注意这一点。新的包名为：jersey.repackaged.com.google.common 和 jersey.repackaged.objectweb.asm。

7. GlassFish 项目

GlassFish 项目地址为 https://glassfish.java.net。GlassFish 比较著名的是 Java EE 服务器项目 Oracle GlassFish Server，该项目还同时包含 Java EE 中包含的一系列标准规范的参考实现，这些参考实现集成于 GlassFish Server，为其 Java EE 容器提供支持。其中对 JAX-RS 2.0 的实现项目是 Jersey。

为什么要在 JAX-RS 2.0 的介绍中提及 GlassFish 项目集呢？因为 Jersey 处于 GlassFish 生态环境中，GlassFish 又是 Java EE 生态环境的实现描述。举个例子，当你需要在 REST 项目中访问数据库时，也许出于首选基于标准的实现的考虑，你会选择使用 JPA。而 JPA 的参考实现 EclipseLink 就是 GlassFish 项目集的成员。这就像在你观赏深海的一条鱼的时候，势必有兴趣了解它会以哪些生物为生，又为哪些生物提供了生存保障。通过了解 GlassFish 项目，可以更好地完成 REST 式的 Web 服务。尤其，当你用户用 NetBeans 作为开发工具时，GlassFish 服务器和 GlassFish 项目集是你生产中的“日用消费品”。

这里所列的项目是除 Jersey 以外其他的 GlassFish 项目，排列顺序并不严谨，大体上以其与 Jersey 的紧密关系降序排列。

（1）HK2

HK2 项目地址为 http://hk2.java.net。HK2 是轻量级 DI 架构，实现 IoC 和 DI 的内核，是 Jersey 实现容器内管理 Bean 的基础，见表 1-9。

（2）Grizzly

Grizzly 的项目地址为 https://grizzly.java.net。Grizzly 是一个异步 I/O 的、高效而健壮的

服务器，可以用作 HTTP 服务器、Servlet 容器，支持 AJP、Comet、WebSocket，以及相对于 RESTFul 的另一种 Web Service 实现 JAX-WS，见表 1-10。

如何使用和配置 Grizzly 依赖于用户的需求，也就是说，用户的项目不必加载 Grizzly 提供的全部 jar 包。表 1-11 列出了 Grizzly 项目的各个模块信息，可以参考该表定义用户的项目对 Grizzly 的依赖。

表 1-9　HK2 项目情况简表

项目名称	JSR 编号	JSR 名称	当前版本
HK2	JSR-330	Dependency Injection for Java	2.2

表 1-10　Grizzly 项目情况简表

项目名称	中文直译	JSR 编号	JSR 名称	当前版本
Grizzly	灰熊	JSR-356	Java API for WebSocket	2.3.3

表 1-11　Grizzly 模块列表

模块名称	描述
grizzly-framework	Grizzly 核心包
grizzly-http-server	HTTP 服务器包
grizzly-http-servlet	Servlet 容器包
grizzly-portunif	Port Unification 包，支持多种协议和端口的协同工作
grizzly-comet	Comet 包，支持 Comet 方式的 BS 访问。典型的场景是长轮询
grizzly-WebSockets	WebSockets 包，支持 WebSocket 协议（RFC 6455），可以和 HTML5 协同工作
grizzly-http-ajp	AJP 包，支持 AJP 协议。AJP 是一种使 Web 服务器和应用服务器通信的协议，典型的应用是 Apache 和 Tomcat 的通信
grizzly-http-server-jaxws	支持 JAX-WS 标准的 HTTP 服务器包

（3）EclipseLink

EclipseLink 项目地址为 http://www.eclipse.org/eclipselink。EclipseLink 是 JPA 2.1 的一个实现，同时它还实现了其他的 JSR 作为扩展。JPA 2.1 是 Java EE 7 的成员，是对 JSR 317(JPA 2.0）的升级。在 JPA 2.1 的实现中，最常用的是 JBoss 的 Hibernate，该项目从 4.3 版本开始实现 JPA 2.1。也就是说，Hibernate 4.2 是 JPA 2.0 的最后一个版本。读者在开发的时候要注意依赖项目版本对标准的支持。EclipseLink 项目情况见表 1-12。

表 1-12　EclipseLink 项目情况简表

项目名称	JSR 编号	JSR 名称	当前版本
EclipseLink	JSR-338	JPA 2.1	2.5.0
	JSR-222	JAXB 2.2	
	JSR-235	SDO 2.1.1	
	JSR-353	Java API for Processing JSON	

JPA 标准还有其他的实现，可参考 http://en.wikipedia.org/wiki/Java_Persistence_API。

（4）Metro

Metro 项目地址为 https://metro.java.net。Metro 是 JSR 中多个标准的官方实现集，目的是实现全栈式的 Web Service。Metro 项目情况见表 1-13。

表 1-13 Metro 项目情况简表

项目名称	JSR 编号	JSR 名称	当前版本
Metro	JSR-224	JAX-WS 2.2	2.2.8
	JSR-222	JAXB 2.2	
	JSR-206	JAXP 1.4.6	
	JSR-067	SAAJ 1.3	

Metro 项目中的多个标准的作用各有不同。

- JAX-WS 标准结合了 XML-RPC 使用 SOAP 协议来实现 Web Service。JAX-WS 实现中，不可不提的另外两个实现分别是 Apache 的 CXF 和 Axis。
- WSIT 的前身是 Tango，是一种 JAX-WS 和 .NET 互操作的技术，实现了 WS* 标准。
- SAAJ 规范的作用是基于 SOAP 协议 XML 格式传递带附件的 SOAP 消息。
- JAXP 标准涵盖了 Java 对 XML 过程式处理的诸多技术，包括 DOM、SAX 和 StAX，同时该标准定义了解读 XML 样式的 XSLT。
- JAXB 标准是 Java 处理 XML 和 POJO 映射的技术，是 Jersey 中处理传输数据的重要依赖。

（5）Open MQ

Open MQ 项目地址为 https://mq.java.net。Open MQ 是 JMS 2.0 的参考实现。JSR 343 是 Java EE 7 的成员，旨在简化 JMS 的 API。关于消息队列的实现数量，恐怕是其他任何一个标准都望尘莫及的。几乎每一个有能力开发服务器软件、中间件的公司都有自己的 MQ，可参考 http://en.wikipedia.org/wiki/Message_queue。目前支持 JMS 2.0 的其他实现并不多，HornetQ 2.4.0+ 兼容 JMS 1.1 和 JMS 2.0 是为数不多的 JMS 2.0 的其他实现。Open MQ 简介见表 1-14。

表 1-14 Open MQ 项目情况简表

项目名称	中文直译	JSR 编号	JSR 名称	当前版本
Open Message Queue	开源消息队列	JSR-343	JMS 2.0	5.0

（6）Mojarra

Mojarra 项目地址为 https://Java SErverfaces.java.net。Mojarra 是 JSF 2 的官方实现。JSF 是一种全栈式的、事件驱动的 B/S 开发模式框架，它包括浏览器端的丰富组件，服务器端覆盖 Java EE 的各种特性，见表 1-15。JSF 相对于 Spring，借鉴了其核心思想 IoC 和 AOP，同时给出了标准规范。这有点类似 JPA 借鉴了 Hibernate 的 O/R Mapping 思想并标准化。

JSF 的另一个实现是 Apache 的 MyFaces，当前版本为 2.0.18。另外，JBoss 的 RichFaces

是基于 JSF 的扩展中最为完善和常用的。更多有关 JSF 的内容和原理，可参考笔者的《JSF 2 和 RichFaces 4 使用指南》。

表 1-15　Mojarra 项目情况简表

项目名称	中文直译	JSR 编号	JSR 名称	当前版本
Mojarra	银鲈	JSR-344	JSF 2.2	2.2.1

（7）OpenJDK

OpenJDK 项目地址为 http://openjdk.java.net。OpenJDK 是开源的 JDK，从 1.7 版本开始成为官方 JDK 的先行版本，因此是 Java 开发者研究 Java 发展的第一线的资源。同时也是活跃的 Linux 发行版本 Ubuntu 和 Fedora 等默认安装的 JDK 版本。OpenJDK 项目情况见表 1-16。

表 1-16　OpenJDK 项目情况简表

项目名称	中文直译	当前版本
OpenJDK	开源 JDK	JDK8.0

小白讲堂

下面讲一下 JDK 版本号升级规则。

自 JDK 5.0 发布开始，Java 一直以两种方式发布更新。

1）有限升级（Limited Update）：包含新功能和非安全修正。

2）重要补丁升级（Critical Patch Updates，CPUs）：只包含安全修正。

有限升级发行序号为 20 的倍数，即一个偶数；重要补丁升级序号为顺延上一个 CPUs 的版本号 +5 的倍数并取奇数（必要时 +1）。

举例来说，下一个有限升级的版本号为 7u40，那么接下来的 3 个 CPUs 版本号依次为 40+5=7u45、45+5+1=7u51 和 51+5=7u55[⊖]。再下一个有限升级的版本号为 7u60，随后的 CPUs 版本号依次为 7u65、7u71 和 7u75。

这种命名规则会为重要补丁升级保留几个版本序号，以便新的 CPUs 版本号可以取区间值和而不是在最新版本号上顺延。

1.5　Java 领域的其他 REST 实现

Java 领域的 REST 实现以是否遵循 JAX-RS 标准分为两个阵营。前者为参考实现之外的厂商实现，后者要么出现较早、要么干脆跳出了标准的框架，以自身所追求的目标，比如性能、框架一致性等，实现了一套独有的对 REST 开发的支持。本节将做概括性介绍，以便读者有所对比和选择。

1.5.1 其他 JAX-RS 实现

JAX-RS 标准发布后，诸多厂商推出了自己的基于 JAX-RS 标准的实现。其中比较有影响力的应该是来自 JBoss 社区的 RESTEasy 和来自 Apache 社区的 CXF。本节将简述这两个项目。如果读者的项目确实和它们结合得比较紧密，Jersey 未必是最佳选择，读者尽可“拥抱”这两个基于 JAX-RS 标准的项目。

1. JBoss 的 RESTEasy

RESTEasy 是 JBoss 社区提供的 JAX-RS 项目。值得说明的是，JBoss 这一名词目前已经不再代表 Java EE 容器，曾经的 JBoss 已经更名为 WildFly，而 JBoss 一词现在特指 RedHat 公司旗下的开源社区。RESTEasy 自 2009 年 1 月第一个 GA 版本以来，发展到 3.0.x。从版本 3.0.0.Final 开始支持 JAX-RS 2.0。

官方文档提供单页面 HTML、按章节 HTML 和 PDF 三种格式，可以按照阅读习惯选择。参见：

```
http://www.jboss.org/resteasy/docs
```

该项目源代码由 GitHub 托管，地址为：

```
https://github.com/resteasy/Resteasy
```

项目下载由 http://sourceforge.net 托管，当前版本为 3.0.7.Final。地址为：

```
http://sourceforge.net/projects/resteasy/files/Resteasy%20JAX-RS
```

2. Apache 的 CXF

CXF 是 Apache 开源社区提供的 JAX-RS 项目，CXF 的名称是由 Celtix 项目和 XFire 项目合并而来。其中 Celtix 由 IONA Technologies 开发，XFire 来自 Codehaus。CXF 是 JAX-WS 的著名实现，同时实现了 JAX-RS，从版本 2.7.0 开始几乎全面支持 JAX-RS 2.0 全部特性。官方文档参见 http://cxf.apache.org/docs/jax-rs.html。

Apache CXF 当前版本为 2.7.11，下载地址为：

```
http://cxf.apache.org/download.html
```

源代码由 Apache 的 GIT 服务器托管，地址为：

```
https://git-wip-us.apache.org/repos/asf?p=cxf.git
```

源代码克隆命令参考如下：

```
git clone https://git-wip-us.apache.org/repos/asf/cxf.git
```

1.5.2 其他 REST 实现

JAX-RS 是 Java 领域实现 REST 式 Web 服务的标准规范。但需要注意的是，Java 领域支持 REST 式 Web 服务开发的工具未必遵循 JAX-RS 规范。其中，大名鼎鼎的 Spring MVC 就是一个支持 REST 开发的非 JAX-RS 规范的实现。本节将带领读者认识 Java 领域的非 JAX-RS 规范的著名 REST 支持工具。

（1）Restlet 项目

Restlet 是一款遵从 REST 风格的、基于 Java 平台的轻量级框架。Restlet 是开源的，提供 REST 开发的完整支持。Restlet 官网地址为：

```
http://restlet.org
```

Restlet 源代码由 GitHub 托管，地址为：

```
https://github.com/restlet
```

Restlet 学习指南文档地址为：

```
http://restlet.org/learn/tutorial
```

（2）LinkedIn 的 Rest.li

Rest.li 是社交网站 LinkedIn 开发的 REST+JSON 的开源 REST 式服务框架。Rest.li 的官方地址是：

```
http://rest.li
```

Rest.li 源代码由 GitHub 托管，地址为：

```
https://github.com/linkedin/rest.li
```

文档的 wiki 地址为：

```
https://github.com/linkedin/rest.li/wiki
```

（3）Spring WEB MVC 项目

Spring 框架使用 Gradle 构建和管理项目，使用 GIT 管理源代码，地址为：

```
https://github.com/spring-projects/spring-framework
```

Spring-Web 是 Spring 项目的一个模块，详情可参考：

```
http://spring.io/projects
```

Spring-Web 从 3.0 版本开始提供了对 REST 式应用开发的支持，但 Spring-Web 目前并没有推出一个实现 JAX-RS 标准的模块。Spring-Web MVC 模块提供了 REST 功能，但没有采用 JAX-RS 提出的标准。本质上，Spring-Web MVC 控制流程是使用 Controller 处理 Model 在某种动词性的业务逻辑操作，而 JAX-RS 的控制流程是使用资源类 Resource 处理名词性的资源表述。

小白讲堂

Gradle 是一个基于 Apache Maven 概念的项目自动化建构工具。和 Maven 使用传统的 XML 配置项目不同的是，Gradle 使用 Groovy 语言（一种领域特定语言，即 DSL）来配置项目构建信息。Gradle 官网地址是：http://www.gradle.org。

1.6 本章小结

本章的目的是为读者厘清在 Java 领域开发 REST 式的 Web 服务的概念、上下文关系等知识点。先后解读了 REST 这个概念、REST 式的 Web 服务、JAX-RS 2.0 标准中的重要概念。接着对 JAX-RS 2.0 的参考实现项目 Jersey 进行了简单而全面的概述。最后介绍了其他基于 JAX-RS 2.0 标准的项目和其他非 JAX-RS 2.0 标准的、著名的 Java 项目。

希望通过阅读本章，读者可以扫清 Java 领域开发 REST 式的 Web 服务的概念上的障碍。下一章将通过实际操作来讲解 JAX-RS 2.0 实现，带领读者实现从理论到实践的飞跃。

第 2 章 Chapter 2

JAX-RS 2.0 快速实现

学习和使用一种新技术，说起来并没有多少玄机。相信每一位深得其法的读者，都遵循着这样一条学习路线，那就是，首先了解背景知识，收集相关概念、标准和实现工具，在脑海里形成根枝结构，希望第 1 章已经带给读者这样的帮助。接着就是找到一个好用的、有引领性的例子来快速学习，这是不断成长的过程，是走向“枝繁叶茂”的第一步。当然，最后会找寻一两本该领域的权威书籍来深入掌握，最终达到运用熟练的地步。这是本书后续章节希望为读者呈现的。

本章是入门章节，希望读者可以掌握 JAX-RX 2.0 应用开发的相关技能。本章可以形象地比喻为通往掌握 Java RESTful 开发的快速干道，读者将从若干不同场景的例子中了解实现和部署 REST 应用的过程。

实践一个 REST 应用需要考虑两点：第一点是如何定义一个资源，包括以什么方式发布一个请求，它的输入和输出是什么；第二点要考虑的是如何部署一个 Java RESTful Web Service 应用，以匹配既有的 REST 服务类型。

阅读指南 关于对资源设计和定位的思考远不止本章所述，详细情况，读者可以阅读第 3 章。作为入门章节，本章去除“噪声”，让读者只关注能实现 REST 的最少知识。正像迪米特法则那样，知道的最少时最整洁。

举个例子，有一个资源是关于更新设备的 API，我们需要考虑该 API 将以 PUT 方式发布还是其他方式发布？它的输入是 XML 格式、JSON 格式，还是流呢？它的输出是否覆盖更新设备所遇到的所有情形，输出格式是否合理，信息是否完整且合理？另一个要考虑的是这个关于设备的 REST 应用是否部署到 Servlet 容器，以及该容器的版本。带着这样的疑问走进

本章，你一定迫切地想知道答案。首先，我们必须做一些准备工作。

2.1 第一个 Java REST 服务

本节讲述基于 Java SE 环境的 Jersey 官方文档中提供的示例 simple-service（参考地址：https://jersey.java.net/documentation/latest/user-guide.html），并在此基础上扩展自定义的 REST 资源服务。

2.1.1 环境准备

在动手之前，我们需要准备开发 REST 服务的环境，包括 JDK、Maven 和 IDE。

阅读指南

2.1 节示例所在目录是 jax-rs2-guide\sample\2\0simple-service。
源代码地址为：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/0simple-service。

1. 配置 JDK

Jersey 对 JDK 的版本要求是 1.6 及以上，读者可根据项目情况，参考 1.4 节的 Jersey 2 依赖和 GlassFish 项目中的讲述，选择 JDK 1.6 以后的版本来开发基于 Jersey 的 REST 项目。本章示例使用的操作系统是 Windows 7 的 64 位版本，JDK 版本是 jdk-7u25-windows-x64.exe。下载并安装好 JDK，然后修改系统环境变量：添加 JAVA_HOME 参数，并将其 bin 目录追加到 path 中。示例如下。

```
JAVA_HOME=D:\Program Files\Java\jdk1.7.0_25
Path=%JAVA_HOME%\bin;……
```

设置完毕后，Windows 操作系统需要重启控制台使 Java 生效；在 Linux 下，需要使用命令 source path2profile 启用 Java。

2. 配置 Maven

Maven 是 Apache 的项目，是当今流行的项目构建工具。读者可以通过其官网（http://maven.apache.org）了解更多信息。需要注意的是，Maven 版本中的 3.x 相比 2.x 有性能上的优势，推荐使用 Maven 3.x。Maven 3.0.x 和 Maven 3.1.x 的区别在于其内部实现，作为用户使用而言，笔者没有推荐倾向。对于工具版本的选择，并不推荐使用最新版，除非新版本更加稳定可靠。比如 3.0.5 这个版本是 Maven 3.0.x 的修复和维护版本，这意味着该版本较之前面的版本更趋稳定。作为演示，本例选择的是最新版本 maven-3.1.0。下载并解压 Maven，然后在系统环境变量中定义 M2_HOME 并指向 Maven 解压后的路径，在 Path 中添加 M2_HOME 下的 bin 目录，或定义 M2 变量为 M2_HOME 下的 bin 目录，然后将其添加到 Path 中，如下所示。

```
M2_HOME=D:\-aquarius\apache-maven-3.1.0
MAVEN_OPTS=-Xms128m -Xmx512m
Path=%M2_HOME%\bin;
……
```

MAVEN_OPTS 用于定义 Maven 运行时 JVM 虚拟机参数，通常至少需要定义 JVM 堆的最大值 -Xmx 以支持构建较大的项目。Maven 的版本测试命令是 mvn-v，其测试结果如下所示。

```
mvn -v
Apache Maven 3.1.0 (893ca28a1da9d5f51ac03827af98bb730128f9f2; 2013-06-28 10:15:32+0800)
Maven home: D:\-aquarius\apache-maven-3.1.0
Java version: 1.7.0_25, vendor: Oracle Corporation
Java home: D:\Program Files\Java\jdk1.7.0_25\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

3. 使用 IDE

也许读者希望就某种 IDE 讲述示例，由于本书的全部示例都是基于 Maven 的 Java 项目，当今的主流 IDE 对集成 Maven 的支持都非常好，因此本书在讲述技术细节时，将不特别针对某种 IDE 的使用有倾向性的推荐和描述。

本书尽可能确保所提供的全部示例的源代码在以下 IDE 中编译、运行和测试无误。Eclipse Indigo（3.7.2）IDE 中的服务器还是要自行配置，否则纯属环境配置问题，而非示例代码问题。下面是常见的 IDE 列表。

- Eclipse Juno（4.3）。
- IntelliJ IDEA 12.1.6+。
- NetBeans IDE 7.3.1。

读者可以根据个人使用偏好选择 IDE。Jersey 的开发和 Web 开发相比，形式上基本一致，所以不必更换 IDE 来运行本书示例。

2.1.2　创建服务

准备好环境，我们就可以动手操作了。我们从 Maven 原型开始创建，然后测试并分析该示例。

1. 从 Maven 原型创建项目

Jersey 官方文档中提供的例子 simple-service 是一个 Maven 原型项目，我们从这里开始。所谓“原型项目”即指通过 Maven 命令即可从 Maven 中央仓库取回一个已经具备基本功能、依赖包完好、编译和运行测试无误的示例项目。mvn archetype：generate 字面解读为：以指定的原型为模板，生成或者创建新的 Maven 项目。在控制台执行如下命令来生成我们想要的 simple-service 项目，项目的目标存储路径由读者自行选择。

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false
```

```
-DgroupId=com.example -DartifactId=simple-service -Dpackage=com.example
-DarchetypeVersion=2.9
```

控制台命令成功执行后，会在当前目录下创建 simple-service 目录。该目录包含了最简 REST 示例，即 simple-service 项目的全部源代码。

阅读指南 如果读者在确保本地配置无误的情况下，执行上述命令失败，说明该官方例子项目已经失效或者由于非技术原因导致的无法访问，读者可以直接使用本书提供的源代码。Maven 管理的项目版本清晰，一般活跃的项目的失效的可能性比较低。

---

2. 测试项目可用性

simple-service 项目在本地创建成功后，首先要确定该项目在本机的可用性。进入 simple-service 项目的根目录，然后执行 Maven 的测试命令，若 simple-service 项目在环境中已经可用，则控制台将输出如下信息。

```
simple-service>mvn clean test
[INFO] ------------------------------------------------------------------------
[INFO] Building simple-service 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
------
-------------------------------------------------------
TESTS
-------------------------------------------------------
------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.791s
[INFO] Finished at: Sat Jul 27 12:10:43 CST 2013
[INFO] Final Memory: 13M/224M
```

该项目默认使用 JDK 1.6 编译，如果用户的 JDK 版本不同，在执行测试之前，首先要修改 Maven 的配置文件 pom.xml，设置编译插件的 JDK 版本参数。pom.xml 文件位于项目根目录下。使用读者熟悉的编辑器打开 pom 文件，下面以文本编辑器为例：

```
notepad D:\simple-service\pom.xml
```

搜索关键字 maven-compiler-plugin，定位 <source> 和 <target> 两行。以 JDK 1.7 为例，修改为 <source>1.7</source> 和 <target>1.7</target>，保存并退出。最后执行 Maven 命令“mvn clean test”进行测试。

3. 项目分析

项目测试通过意味着本地环境没有问题了，接下来可以开始学习 simple-service 项目了，

为下一步扩展项目做好准备。

在 simple-service 项目的根目录执行“tree /f”命令，可以纵览 simple-service 的文件结构，如图 2-1 所示。

在图 2-1 所示的目录结构中，对开发者有价值的内容包括源代码 Main.java、MyResource.java 和测试代码 MyResourceTest.java。

首先，定位该项目中的有用信息以去除项目中的噪声。什么是噪声呢？无论使用哪一种 IDE 做开发，其自身都会产生具有该 IDE 方言的项目文件。除非特殊情况，我们不必理会这些项目文件，如果是团队开发，在提交到版本控制库的时候，应该对其过滤（使用 GIT 做 SCM 时，过滤文件位于项目根目录，名称为 .gitignore）。target 目录是 Maven 的生成目录，类似构建工具 Ant、Gradle 的 build 或者 bin 目录，我们也可以忽略。

小白讲堂

Eclipse 产生的项目文件包括 .settings 目录，以及 .classpath 和 .project 文件。

IntelliJ 产生的项目文件包括 .idea 目录，以及项目同名的 .iml 文件。

NetBeans 产生的项目文件包括 nb-configuration.xml 文件。

（1）资源类分析

套用 Web 开发中典型的三层逻辑，资源类位于逻辑分层的最高层——API 层，其下为 Service 层和数据访问层，如图 2-2 所示。在三层逻辑中，API 层用于对外公布接口，对于 REST 应用，API 层的资源类用于对外公布 REST 服务接口。其下的两层，REST 应用的开发和标准 Web 开发的区别不是很大。

本例中的 API 层资源类是 MyResource，代码示例如下。

```
package com.example;
...
// 关注点 1: 资源路径
@Path ("myresource")
public class MyResource {
// 关注点 2: 资源方法
    @GET
    @Produces (MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

在这段代码中，资源类 MyResource 使用了 JAX-RS 标准中定义的 @Path 注解来声明名为 myresource 的资源路径，见关注点 1。该类只有一个处理 HTTP 协议的 GET 方法的 getIt() 方法，没有输入参数，输出是一个 String 类型，传输格式是字符串类型（MediaType.TEXT_PLAIN），返回字符串值是“Got it!”，见关注点 2。

在没有测试、运行这个例子之前，我们可以推测通过 REST 访问 getIt() 方法的资源路径如下，接下来的讲述将验证该资源路径。资源路径即是对外公布的 REST 服务接口。

HTTP 服务器路径 /REST 服务名称 /myresource/

```
Administrator: C:\Windows\system32\cmd.exe

D:\simple-service>tree /f
Folder PATH listing for volume d
Volume serial number is 436A-3EC4
D:.
|   pom.xml
|
|---src
|   |---main
|   |   \---java
|   |       \---com
|   |           \---example
|   |               Main.java
|   |               MyResource.java
|   |
|   \---test
|       \---java
|           \---com
|               \---example
|                   MyResourceTest.java
|
\---target
    |---classes
    |   \---com
    |       \---example
    |           Main.class
    |           MyResource.class
    |
    |---generated-sources
    |   \---annotations
    |---generated-test-sources
    |   \---test-annotations
    |---surefire-reports
    |       com.example.MyResourceTest.txt
    |       TEST-com.example.MyResourceTest.xml
    |
    \---test-classes
        \---com
            \---example
                MyResourceTest.class
```

图 2-1 simple-service 项目源代码组织结构

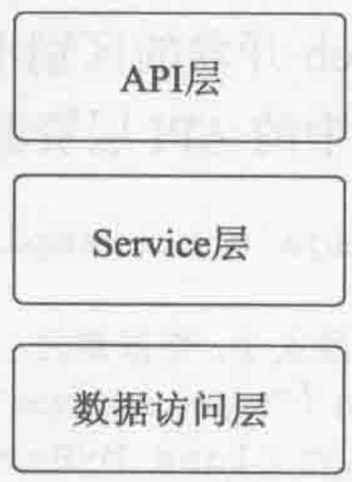


图 2-2 逻辑分层

(2) 入口类分析

因为这是一个 Java SE 的应用，所以需要一个入口类来启动服务器并加载项目资源。对于 Java EE 的应用则无须定义这样的入口类，因为 Java EE 容器本身扮演着入口类的角色。Main 类是 simple-service 项目的主类，即入口类，代码示例如下。

```
package com.example;
...
public class Main {
// 关注点 1: 服务器路径
    public static final String BASE_URI = "http://localhost:8080/myapp/";
    public static HttpServer startServer() {
// 关注点 2: 加载资源
```

```
        final ResourceConfig rc = new ResourceConfig().packages("com.example");
    //关注点 3: Grizzly HTTP 服务器
        return GrizzlyHttpServerFactory.createHttpServer(URI.create(BASE_URI), rc);
    }
    public static void main(String[] args) throws IOException {
        final HttpServer server = startServer();
        System.out.println(String.format("Jersey app started with WADL available at "
                + "%sapplication.wadl\nHit enter to stop it...", BASE_URI));
        System.in.read();
        server.shutdownNow();
    }
}
```

Main 类定义了 HTTP 服务器的路径（BASE_URI），即 http://localhost:8080/myapp/，见关注点 1。在其构造中映射了源代码中资源所在的包名 new ResourceConfig().packages(“com.example”)，这意味着，服务器启动时会自动扫描该包下的所有类，根据该包中所含类的 REST 资源路径的注解，在内存中做好映射，见关注点 2。这样一来，客户端请求指定路径后，服务器就可以根据映射，分派请求给相应的资源类实例的相应方法了。这就是 Jersey 中的 IoC 机制。这个例子不是运行在 Servlet 容器中的，相反，它是一个 Java SE 应用。该服务自带了 HTTP 服务器的实现，本例使用的服务器是 Grizzly，见关注点 3。

宅人坑事

在 1.4 节中，我们对 Grizzly 有过介绍，这里额外要说的是，Grizzly 是 Jersey 提供的集成测试中默认的内嵌测试服务器，有了 Grizzly，我们就可以在不启动额外 Servlet 容器服务器的情况下，测试 REST 服务。这点对于 Maven 老用户来说，一定会联想到在运行 Servlet 容器测试时使用 Jetty 插件的情形。略有不同的是，Jetty 的角色之于 Maven 是一个声明式的（配置参数）、无须编码的插件，而 Grizzly 是编码式的。

（3）测试类分析

最后我们来看看真正用于单元测试的测试类 MyResourceTest。它只有一个测试方法 testGetIt() 用来测试 MyResource 类公布的资源路径 myresource。其代码示例如下。

```
package com.example;
...
public class MyResourceTest {
//关注点 1: 全局字段
    private HttpServer server;
    private WebTarget target;
//关注点 2: 准备测试环境
    @Before
    public void setUp() throws Exception {
        server = Main.startServer();
        Client c = ClientBuilder.newClient();
        target = c.target(Main.BASE_URI);
    }
//关注点 3: 释放测试环境
```

```
    @After
    public void tearDown() throws Exception {
        server.shutdownNow();
    }
//关注点 4: 测试 GET 方法
    @Test
    public void testGetIt() {
        String responseMsg = target.path("myresource").request().get(String.class);
        assertEquals("Got it!", responseMsg);
    }
}
```

MyResourceTest 定义了两个全局字段，分别是 Grizzly 服务器类 HttpServer 和 JAX-RS 2.0 的客户端资源定位类 WebTarget，见关注点 1。

注解为 @Before 的 setUp() 方法是 JUnit 测试前要执行的方法，为测试准备环境。首先启动 Grizzly 服务器，然后实例化一个 Client，最后将服务器的资源地址作为参数，传给客户端示例，获取到客户端资源定位类实例 target，见关注点 2。

testGetIt() 方法用于测试 GET 方法，使用资源定位类 WebTarget 实例 target 向 myresource 资源发出 get 请求，将返回值作为相等断言的输入参数，和“Got it!”进行比对，见关注点 4。

注解为 @After 的方法 tearDown() 在测试结束后执行，以释放测试环境中的资源，如停止服务器实例、释放资源等，见关注点 3。这里需要注意的是，释放资源过程无须处理客户端实例，该实例的连接已经在获取响应实体时断开，详见后文的讲述。

到此，我们已经掌握了 simple-service 项目。在入门学习中，模仿是以后做到收放自如的第一步。接下来，我们重用 Main 作为环境支撑，模仿 MyResource 写一个自定义的资源类，并进行扩展性的尝试：引入实体类、逻辑分层类，支持更多的传输类型。还记得本章开始时关于设计一个更新设备的 API 的疑问吗？继续轻松的学习之旅吧。

2.1.3 扩展服务

在 Maven 原型示例的基础上进行模仿来扩展该项目的 REST 服务接口是本节的第二个任务，通过这一过程实现对 Jersey 使用的初步认识。下面我们要完成的任务是实现一个更新设备的 API，这将包括开发设备实体类、资源类和逻辑分层类，然后对其进行测试，以检验我们的成果。

1. 增加设备实体类

资源类和逻辑分层类在图 2-2 中已经展示，实体类是资源自身的信息，用于序列化和持久化资源。设备的实体类用于传输和持久化设备资源，设备的实体类命名为 Device，该类是一个最基本的 POJO 类，包含两个属性，分别是设备的 IP 和设备的状态。其代码示例如下。

```
//关注点 1: JAXB 根元素
@XmlRootElement(name = "device")
public class Device {
```

```
    private String deviceIp;
    private int deviceStatus;
    public Device() {
    }
    public Device(String deviceIp) {
        super();
        this.deviceIp = deviceIp;
    }
  //关注点 2: JAXB 属性 @XmlAttribute
    public String getIp() {
        return deviceIp;
    }
    public void setIp(String deviceIp) {
        this.deviceIp = deviceIp;
    }
    @XmlAttribute
    public int getStatus() {
        return deviceStatus;
    }
    public void setStatus(int deviceStatus) {
        this.deviceStatus = deviceStatus;
    }
}
```

该类标注了 JAXB 标准定义的 @XmlRootElement 和 @XmlAttribute 注解，以便将 Device 类和 XML 格式的设备数据相互转换并在服务器和客户端之间传输，见关注点 1 和关注点 2。

小白讲堂

Jersey 内部使用 JAXB 处理 Java 类（POJO）和 XML 格式的信息、JSON 格式的信息映射，JAXB 通过 POJO 中定义的 XML 注解（比如 @XmlRootElement 代表根节点，@XmlAttribute 代表一个节点的属性等）将其与 XML 格式的信息对应起来。

---

2. 增加设备资源类

创建了设备实体类后，我们需要一个资源类来公布设备的 REST API。源代码中已经提供了资源类 MyResource，为实现设备管理，我们模仿该类，创建设备资源类 DeviceResource，代码示例如下。

```
@Path("device")
public class DeviceResource {
//关注点 1: 注入 Dao 实例
    private final DeviceDao deviceDao;
    public DeviceResource() {
        deviceDao = new DeviceDao();
    }
//关注点 2: GET 方法
    @GET
```

```
    @Produces ({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Device get (@QueryParam ("ip") final String deviceIp) {
    Device result = null;
        if (deviceIp != null) {
            result = deviceDao.getDevice (deviceIp) ;
        }
        return result;
    }
//关注点 3: PUT 方法
    @PUT
    @Produces ({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Device put (final Device device) {
        Device result = null;
        if (device != null) {
            result = deviceDao.updateDevice (device) ;
        }
        return result;
    }
}
```

在这段代码中，关注点 2 中的 GET 方法和 2.1.2 节定义的 GET 方法作用相同，用于获取设备信息，输入参数 deviceIp 是以设备 IP 作为查询条件，输出是设备实体类的实例，其表述格式可以是 JSON 或者 XML。关注点 3 是 DeviceResource 类处理 HTTP 协议的 PUT 方法。put() 方法定义了两个注解，@PUT 是标识处理 PUT 请求，@Produces 是标识返回实体的类型，本例中同时支持 JSON 格式和 XML 格式。细心的读者会质疑持久层 Dao 类 DeviceDao 的实例化是在资源类中完成的，见关注点 1，可能会发现这不符合三层设计，也不符合 IoC。我们会在后续示例中逐渐做到更加严谨，这只是第一个入门例子，放轻松。

3. 增加设备逻辑类

本例演示了简单的分层结构，数据的持久化由 DeviceDao 类完成，该类简单地模拟了设备持久化。其示例代码如下。

```
public class DeviceDao {
    ConcurrentHashMap<String, Device> fakeDB = new ConcurrentHashMap<>();
    public DeviceDao() {
    //关注点 1: 测试数据，初始化了两个设备实例
        fakeDB.put("10.11.58.163", new Device("10.11.58.163"));
        fakeDB.put("10.11.58.184", new Device("10.11.58.184"));
    }
    public Device getDevice(String ip) {
        return fakeDB.get(ip);
    }
    public Device updateDevice(Device device) {
        String ip = device.getIp();
        fakeDB.put(ip, device);
        return fakeDB.get(ip);
    }
}
```

在这段代码中，DeviceDao 类简单地使用 HashMap 实现了内存级别的持久化。为了单元测试的需要，本例使用了硬编码的测试数据，持久层加载时，为测试环境初始化了两个设备实例，见关注点 1。这部分代码在实际项目的单元测试中，不要使用硬编码，可以使用内存数据库比如 H2、Derby 等替换；在集成环境中，应使用真实的数据库进行替换。

2.1.4 测试和运行服务

到此，完成了 simple-service 扩展的代码开发部分，接下来为代码编写单元测试类，以检验扩展的 REST 接口是否正常工作。

DeviceResourceTest 类模仿测试类 MyResourceTest，分别测试了上述的 GET 和 PUT 请求处理方法，示例代码如下。

```
public class DeviceResourceTest {
    private HttpServer server;
    private WebTarget target;
    @Before
    public void setUp() throws Exception {
        server = Main.startServer();
        final Client c = ClientBuilder.newClient();
        target = c.target(Main.BASE_URI);
    }
    @After
    public void tearDown() throws Exception {
        server.shutdownNow();
    }
    // 关注点 1: 测试 GET 方法
    @Test
    public void testGetDevice() {
        final String testIp = "10.11.58.184";
        // 关注点 2: 带参数的 GET 请求
        final Device device = target.path("device")
       .queryParam("ip", testIp).request().get(Device.class);
        // 关注点 3: 设备 IP 的断言
        Assert.assertEquals(testIp ,device.getIp());
    }
        // 关注点 4: 测试 PUT 方法
    @Test
    public void testUpdateDevice() {
        final String testIp = "10.11.58.163";
        final Device device = new Device(testIp);
        device.setStatus(1);
        Entity<Device> entity = Entity.entity(device, MediaType.APPLICATION_XML_TYPE);
        final Device result = target.path("device").request().put(entity, Device.class);
        // 关注点 5: 设备状态的断言
        Assert.assertEquals(1, result.getStatus());
    }
}
```

GET 测试方法 testGetDevice() 测试资源路径为 device 的 get 请求，见关注点 1。在关注点 2，target 传递了一个字符串参数 ip，其值为“10.11.58.184”。断言验证返回的设备 IP 是否与预期相同，见关注点 3。PUT 测试方法 testUpdateDevice() 创建了一个 Device 实例，并

以 put 方法将其提交到 device 资源路径，见关注点 4。断言验证返回设备状态是否与预期一致，见关注点 5。

完成测试类后，打开控制台，在项目的存储目录再次运行“mvn clean test”命令，如果测试通过，即断言验证成功，说明扩展实现已经成功结束。

宅人坑事

关于单元测试，想和诸君分享一点体会。单元测试是对实现方法所设置的第一道检验屏障，是持续集成中至关重要的环节，是质量保证体系中测试覆盖率的基础。成为优秀开发者就要遵守良好的习惯，单元测试不可忽视，甚至应该遵循测试先行、测试驱动开发这些软件工程中优秀的方法论。

如果开发者写不好单元测试或者忽略其重要性，将这一步骤交由工具来生成，以便完成集成测试的流程和代码覆盖率的 KPI，那么，笔者所持的观点是，这是一种倒退性实践，极力反对这种方式。

最后，我们运行 Main 类，通过浏览器来体验一下第一个 REST 服务。在浏览器地址栏中输入 http://localhost:8080/myapp/device?ip=10.11.58.184，预期得到的输出为：<device ip="10.11.58.184" status="0"/>。同时，这个测试也验证了前文中推测的资源地址 URL 规则：

```
HTTP 服务器路径 /REST 服务名称 /myresource/
```

到此，我们已经快速掌握了一个非常简单的 Jersey 2 的 Java SE 应用。接下来，我们介绍更加常用的基于 Servlet 容器的 REST Web 服务。

2.2 第一个 Servlet 容器服务

上一节讲述的 REST 服务是基于 Java SE 环境的，本节将介绍 Java EE 环境下的 REST 服务，即 REST 式的 Web 服务。接下来介绍基于 Servlet 环境的 Jersey 官方示例 simple-service-webapp，并演示这个 Web 服务如何在 Maven 插件、Servlet 容器和 Java EE 容器中运行该示例。

阅读指南

2.2 节示例所在目录是 jax-rs2-guide\sample\2\0simple-service-webapp-jetty。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/0simple-service-webapp-jetty。

2.2.1 创建和分析 Web 服务

simple-service-webapp 项目也是 Jersey 提供的官方文档中的例子，同样是一个 Maven 原型。在控制台执行如下命令来生成 Maven 原型项目 simple-service-webapp，项目的目标存储

路径由读者自行选择。执行生成的命令如下所示：

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-webapp
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false
-DgroupId=com.example -DartifactId=simple-service-webapp -Dpackage=com.example
-DarchetypeVersion=2.9
```

获取源代码后，通过“tree /f”命令纵览 simple-service-webapp 项目，如图 2-3 所示。对比 simple-service 项目可以发现，这个 Servlet 容器内的版本没有用于服务器处理的入口类，多出来 Web 工程中两个典型的文件 index.jsp 和 web.xml。下面通过分析文件来了解 Web RESTful 服务。

```
Administrator: C:\Windows\system32\cmd.exe

D:\x\simple-service-webapp>tree /f
Folder PATH listing for volume d
Volume serial number is 436A-3EC4
D:.
|   pom.xml
|
└───src
    └───main
        ├───java
        │   └───com
        │       └───example
        │               MyResource.java
        │
        ├───resources
        └───webapp
            |   index.jsp
            |
            └───WEB-INF
                    web.xml
```

图 2-3　simple-service-webapp 项目源代码组织结构

1. Servlet 依赖

如图 2-3 所示，位于项目的根目录下的 pom.xml 文件是 Maven 项目的配置文件。在 pom.xml 文件中，示例默认使用的是支持向下兼容 Servlet 2.x 的包 jersey-container-servlet-core。如果读者需要使用 Servlet 3 版本，那么应当首先修改这个配置，再进行编译、测试和调试。关于 Servlet 容器的依赖包说明，见 1.4 节的详述。pom.xml 中的相关配置如下所示。

```
<!--servlet 2.x -->
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet-core</artifactId>
</dependency>
<!--servlet 3.x -->
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet</artifactId>
</dependency>
```

2. 资源类分析

使用读者熟悉的编辑器打开这个资源文件，这里以文本编辑器为例，示例如下。

```
notepad src\main\java\com\example\MyResource.java
```

从该类的实现中可以发现，REST 的资源类的定义在 Java SE 环境和 Servlet 容器内是一样的。由此可以理解，REST 服务的资源路径是由服务器路径和资源的相对路径组成。一个项目的相对路径是固定的，因此资源类及其子资源类的定义并不关心所处的容器环境，即服务器路径，而只关心其相对的资源路径。

3. Web 服务首页分析

index.jsp 是该示例的首页，需要指出的是该示例其实是一个纯 HTML 实现，虽然扩展名是 jsp，但其实现中并无 JSP 脚本。使用文本编辑器打开 index.jsp 文件：notepad src\main\webapp\index.jsp，示例如下。

```
<html>
<body>
<h2>Jersey RESTful Web Application!</h2>
//关注点 1：资源路径链接
<p><a href="webapi/myresource">Jersey resource</a>
<p>Visit <a href="http://jersey.java.net">Project Jersey Website</a>
    for more information on Jersey!
</body>
</html>
```

在这个简单的首页文件中，关键的一行是指向 webapi/myresource 这个资源路径的链接，见关注点 1。当用户单击这个链接时，浏览器发起 GET 请求，资源路径为 webapi/myresource，流程进入资源类 MyResource 的 getIt() 方法。

4. Web 服务配置

配置文件 web.xml 中定义了 Web 服务的信息。使用文本编辑器打开 web.xml 文件：

```
notepad src\main\webapp\WEB-INF\web.xml
```

该文件可以分成 3 块进行阅读，示例如下。

第一块：定义 Servlet 版本。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE http://java.sun.com/
xml/ns/Java EE/web-app_2_5.xsd">
```

在这段代码中，列出了对 Servlet 版本的定义，本示例默认使用的版本是 2.5。需要说明的是，如果使用 Servlet 3.0，那么 web.xml 文件并不是必要的，我们将在本章稍后详述。

第二块：定义 Servlet。

```
<servlet>
  <servlet-name>Jersey Web Application</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
       <param-name>jersey.config.server.provider.packages</param-name>
       <param-value>com.example</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

在这段代码中，对 Servlet 进行了定义，包括 servlet-name、servlet-class 和 init-param。对于 2.5 版本，servlet-class 是不可或缺的。初始化参数指定了加载服务时要扫描的包名，这里定义的是 com.example 包。

第三块：定义 servlet-mapping，即 Servlet 的作用域。

```
<servlet-mapping>
  <servlet-name>Jersey Web Application</servlet-name>
  <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

在这段代码中，列出了 servlet-mapping 的定义，将上述定义的 Servlet 的作用域匹配到 /webapi/* 的路径上。

在分析了 simple-service-webapp 项目的内容后，我们来测试这个 Servlet 项目在容器内的运行情况。在接下来的 3 个小节中，我们将 simple-service-webapp 项目分别运行在三种不同的容器环境中，为不同需求的读者提供参考，分别是 Maven 插件 Jetty、Servlet 容器 Tomcat 和 Jave EE 容器 GlassFish。

2.2.2 Jetty 插件与 REST 服务

首先，我们使用 Maven+Jetty 这种经典组合来运行 REST 服务，即通过执行 Maven 命令启动用于测试 REST 服务的 Servlet 容器 Jetty 插件。图 2-4 展示了 Maven 的生命周期，结合启动和停止 Jetty 的配置文件，读者可以清楚地了解 Jetty 插件在 Maven 工程的生命周期中所处的位置。

在 Maven 生命周期的每个阶段中，如果失败，则流程结束于该阶段，否则执行至 Maven 命令中定义的阶段，并成功结束。例如，执行了如下命令。

```
mvn clean install
```

如果成功执行至 install 阶段，流程将在 install 结束后，成功结束，不再执行后面的 deploy。如果在中间某个阶段失败，比如在 test 阶段失败，那么流程结束于 test，后面的 package 等阶段将不会再执行。

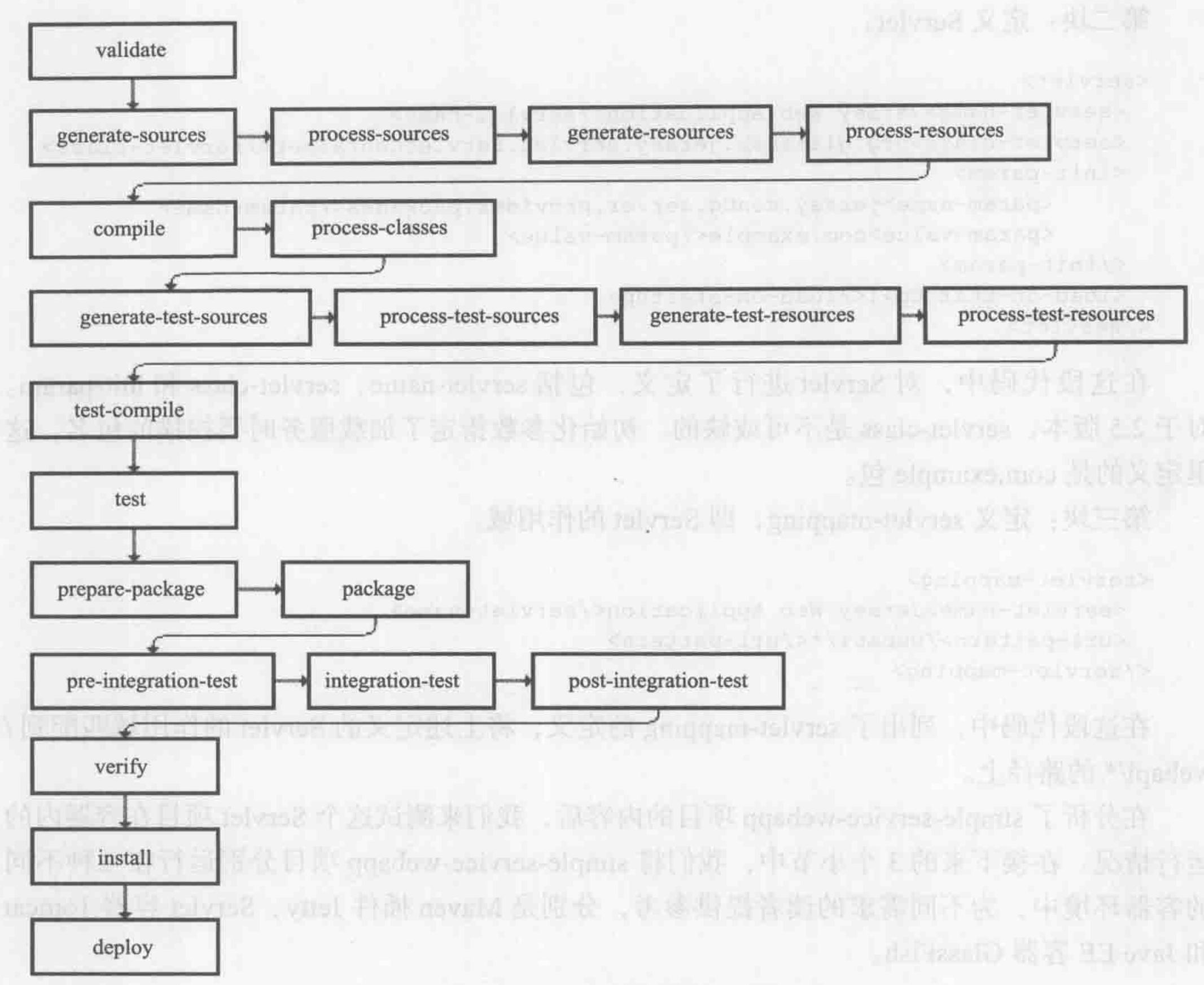


图 2-4 Maven 生命周期示意图

小白讲堂

如果读者对 Maven 的生命周期的各个阶段的含义并不了解，那么建议阅读许晓斌所著的《Maven 实战》(机械工业出版社出版)。

1. 插件配置

在图 2-4 中，integration-test 阶段的前后各有一个扩展点 pre 和 post，Jetty 插件就是在这两个点启动和停止内置的 Jetty 服务器的。插件配置（这是 Jetty 插件早期的配置，接下来会说明）如下。

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.26</version>
  <executions>
```

```
        <execution>
            <id>start-jetty</id>
            // 关注点 1: 在 Maven 的 pre-integration-test 生命周期执行 run
            <phase>pre-integration-test</phase>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
        <execution>
            <id>stop-jetty</id>
            // 关注点 2: 在 Maven 的 post-integration-test 生命周期执行 stop
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Jetty 的运行发生在 pre-integration-test 阶段，见关注点 1；Jetty 的结束发生在 post-integration-test 阶段，见关注点 2。

读者在使用 Jetty 插件时，应注意该插件的版本信息。从 Jetty 7 开始，Jetty 的 Maven 插件更改了命名方式以适应 Maven 插件的统一命名规则，相应的配置文件如下。

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    // 关注点 1: Jetty 插件 artifactId 命名
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.1.0.RC0</version>
    <executions>
        <execution>
            <id>start-jetty</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>start</goal>
            </goals>
        </execution>
        <execution>
            <id>stop-jetty</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

从关注点 1 可以看到，Jetty 插件的名称从老版本的 maven-jetty-plugin 改为 jetty-maven-plugin。

2. 运行插件

可以进入命令行，在项目根目录下执行如下命令，或者在 IDE 中配置 Maven 使用如下命令启动：

```
mvn jetty:run
```

Jetty 服务器启动后，若要终止当前测试用的 Jetty 服务器，在命令行可以执行 <Ctrl+C> 操作，在 IDE 中可以停止 Maven 服务的运行。

Jetty 服务器启动后，打开浏览器，输入地址：http://localhost:8080/simple-service-webapp，访问 Web 服务首页，如图 2-5 所示。

图 2-5 simple-service-webapp 首页

在图 2-5 中有上下两个链接，上面的链接就是本示例 Web 服务的资源地址：http://localhost:8080/simple-service-webapp/webapi/myresource，单击该链接会得到资源类 GET 对应方法，该方法返回的字符串信息是“Got it！”。本示例的 WADL 地址是 http://localhost:8080/simple-service-webapp/webapi/application.wadl，关于 WADL 的详细信息，参见 2.4 节。

2.2.3 运行在 Servlet 容器

Jetty 插件多用于 Web 服务的调试和集成测试，在该阶段通过以后，通常会将该项目部署到以 Tomcat 为代表的 Servlet 容器来搭建测试环境和运行环境。Tomcat 的安装、配置和部署实现步骤如下。

1）从 Tomcat 的官方网址 http://tomcat.apache.org 下载 Tomcat，并解压缩到 D 盘根目录。注意，读者可以自行选择下载版本和本地存储路径。本例使用的是 Tomcat 7.0.42，解压缩后的目录为：D:\apache-tomcat-7.0.42。

2）使用 Maven 命令编译打包（clean 是清除 target 目录，package 是打包，-D 是为 Maven 命令设置运行时参数，skipTests 参数是忽略测试生命周期）。

```
mvn clean package -D skipTests=true
```

3）复制 WAR 包到 Tomcat 应用目录。

```
cp target\simple-service-webapp.war D:\apache-tomcat-7.0.42\webapps\simple-service.war
```

4）运行 Tomcat 服务器。若要结束运行 Tomcat 服务器，则可以按 <Ctrl+C> 组合键。

```
cd D:\apache-tomcat-7.0.42\bin
catalina.bat run
```

5）打开浏览器，输入地址：http://localhost:8080/simple-service-webapp 访问 Web 服务首页。

其他 Servlet 容器的部署和运行类似 Tomcat，可参考相关容器的文档，这里不再列举。

2.2.4　运行在 Java EE 容器

同时，Web 服务可以部署到以 GlassFish 为代表的 Java EE 容器来搭建运行环境。GlassFish 的安装、配置和部署实现步骤如下。

1）GlassFish 的官方网址是 https://glassfish.java.net，读者可以自行选择下载版本。本例使用的是 GlassFish 4.0，解压缩后的目录为：D:\glassfish4。下载地址参考如下：

```
http://dlc.sun.com.edgesuite.net/glassfish/4.0/release/glassfish-4.0.zip。
```

2）使用 Maven 命令编译打包，如下所示：

```
mvn clean package -D skipTests=true
```

3）复制 WAR 包到 GlassFish 默认 domain 目录。

```
cp target\simple-service-webapp.war
D:\glassfish4\glassfish\domains\domain1\autodeploy\simple-service.war
```

4）运行 GlassFish 服务器。

```
cd D:\glassfish4\bin
asadmin start-domain
```

5）打开浏览器，输入 http://localhost:4848 进入 GlassFish 的管理界面，如图 2-6 所示。

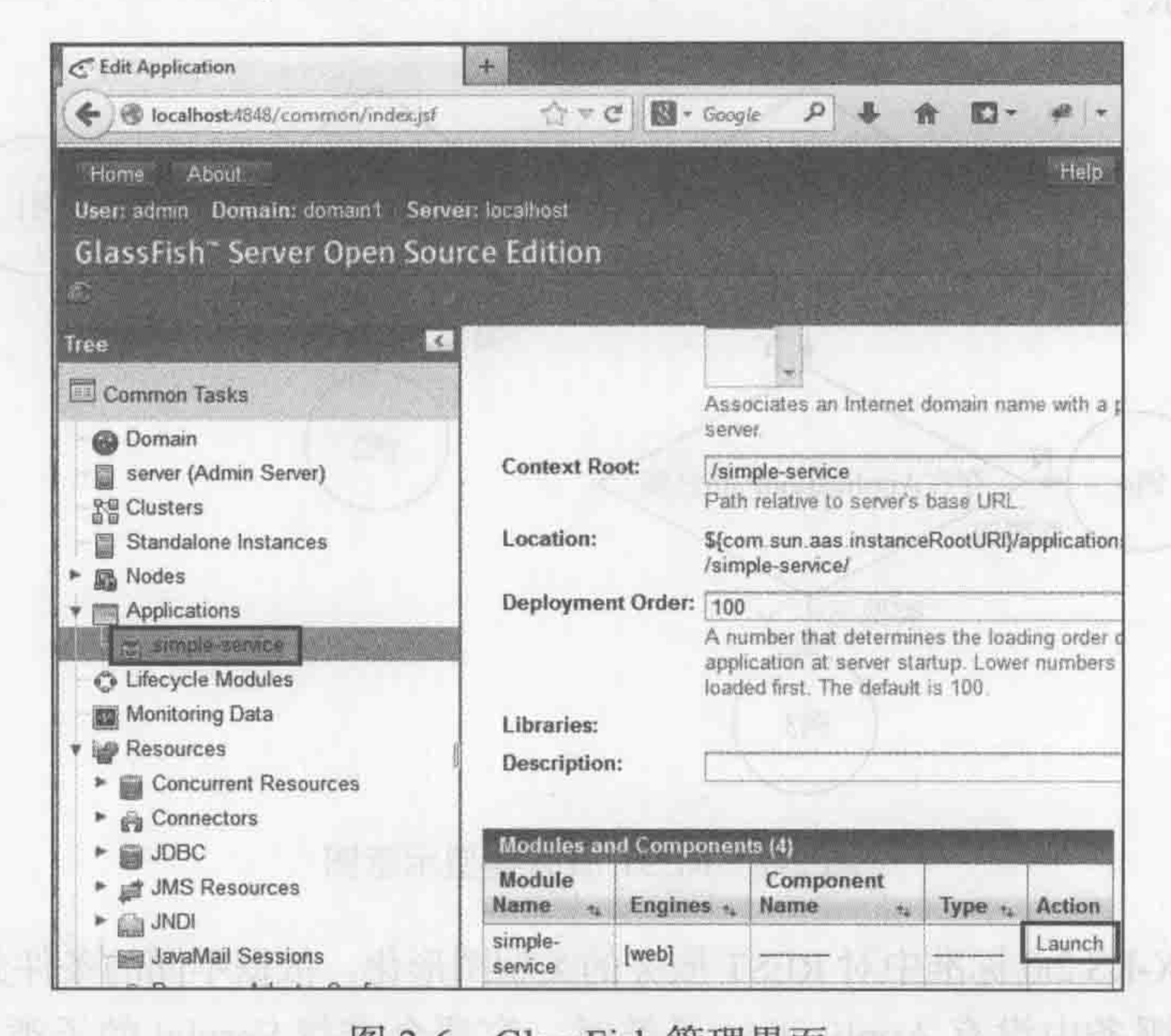


图 2-6　GlassFish 管理界面

在图 2-6 中，REST 服务出现在 GlassFish 管理界面的应用列表中。单击“Launch”启动应用，页面自动跳转到“http:// 主机名 :8080/simple-service”页面。

其他 Jave EE 容器的配置和部署类似 GlassFish，可参考相关容器的文档。

到此，读者已经掌握了开发基于 JAX-RS 2.0 的 Java SE 应用和 REST 式 Web 服务的基本方法。接下来的一节将讲述如何实现和部署不同形式的 JAX-RS 2.0 服务，通过这一节的知识介绍，读者将掌握灵活实现一个 REST 式 Web 服务的技能。

2.3 REST 服务类型

通过前面两节的讲述，我们对 Jersey 官方提供的 REST 示例有了基本了解，并初步掌握如何从头编写一个 REST 服务。在 REST 服务中，资源类是接收 REST 请求并完成响应的核心类，而资源类是由 REST 服务的“提供者”来调度的。这一概念类似其他框架中自定义的 Servlet 类，该类会将请求分派给指定的 Controller/Action 类来处理。本节将讲述 REST 中的这个提供者，即 JAX-RS 2.0 中定义的 Application 以及 Servlet。

Application 类在 JAX-RS 2.0（JSR 339，详见参考资料）标准中定义为 javax.ws.rs.core.Application，相当于 JAX-RS 2.0 服务的入口。作为应用的入口，Application 需要知道具体的资源文件，这里可以通过包扫描或直接指定类文件的方式获得。如果 REST 服务没有自定义 Application 的子类，容器将默认生成一个 javax.ws.rs.core.Application 类。

本节根据 JAX-RS 2.0 规范上文中对 REST 服务场景的定义，将 REST 服务分为 4 种类型，如图 2-7 所示。

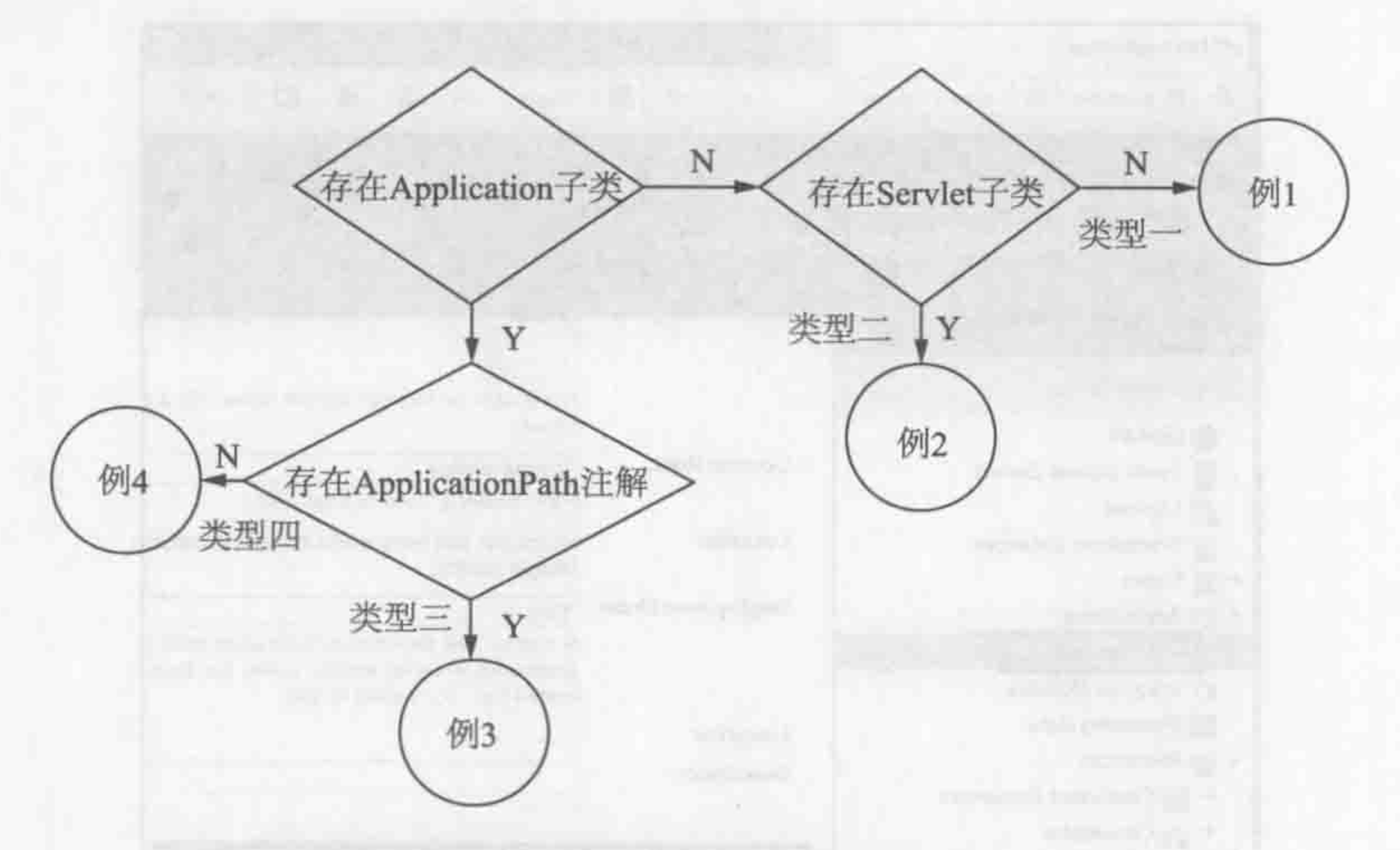


图 2-7 REST 服务类型示意图

图 2-7 将 JAX-RS 2.0 标准中对 REST 服务的类型图形化，依据不同的条件分为了 4 种类型。

类型一：当服务中没有 Application 子类时，容器会查找 Servlet 的子类来做入口，如果

Servlet 的子类也不存在，则 REST 服务类型为类型一，对应图 2-7 中的例 1。

类型二：当服务中没有 Application 子类，但存在 Servlet 的子类时，则 REST 服务类型为类型二，对应图 2-7 中的例 2。

类型三：服务中定义了 Application 的子类，而且这个 Application 的子类使用了 @ApplicationPath 注解，则 REST 服务类型为类型三，对应图 2-7 中的例 3。

类型四：如果服务中定义了 Application 的子类，但是这个 Application 的子类没有使用 @ApplicationPath 注解，则 REST 服务类型为类型四，对应图 2-7 中的例 4。

上面提到的 4 个示例在下面的“阅读指南”中给出了源代码目录和 GitHub 下载地址，需要读者仔细体会示例之间的差异，以更好地理解和使用不同类型的 REST 服务。

1. REST 服务类型一

类型一对应的是图 2-7 中所示的例 1，相应的逻辑是服务中同时不存在 Application 的子类和 Servlet 的子类。在 JAX-RS 2.0（JSR 339）中定义在这种情况下应进行如下处理：为 REST 服务动态生成一个名为 javax.ws.rs.core.Application 的 Servlet 实例，并自动探测匹配资源。与此同时，需要根据 Servlet 的不同版本，在 web.xml 定义 REST 请求处理的 Servlet 为这个动态生成的 Servlet，并定义该 Servlet 对资源路径的匹配。在没有 Application 的子类存在的情况下，在 web.xml 中定义 Servlet 是必不可少的配置。

阅读指南

REST 服务类型一所对应的示例，即例 1 所在目录是：

1）jax-rs2-guide\sample\2\1simple-service-webapp-servlet2-webxml

2）jax-rs2-guide\sample\2\1simple-service-webapp-servlet3-webxml

源代码地址：

1）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/1simple-service-webapp-servlet2-webxml

2）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/1simple-service-webapp-servlet3-webxml

---

REST 服务类型一的示例包含两个小项目，分别对应 Servlet 2 和 Servlet 3 两种容器依赖场景。我们只需关注 Maven 配置文件（pom.xml）和 Web 服务配置文件（web.xml）的区别即可理解存在无 Application 子类的情况下，如何实现基于 Servlet 2 和 Servlet 3 容器内的服务。

Servlet 3 的最简配置示例代码如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java
EE/web-app_3_0.xsd">
<servlet>
```

```
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
    </servlet>
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/webapi/*</url-pattern>
    </servlet-mapping>
</web-app>
```

相对于 Servlet 2 而言，在 Servlet 3 中，Servlet 的定义可以只包含 servlet-name。再次强调，Jersey 的 Servlet 3 的容器支持包是 jersey-container-servlet。

Servlet 2 的最简配置示例代码如下。

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java
EE/web-app_2_5.xsd">
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>com.example</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Web Application</servlet-name>
        <url-pattern>/webapi/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Servlet 的定义包含 servlet-name 和 servlet-class，其初始化参数需要显示给出要加载的资源类所在的包名，可以看出 Servlet 2 的支持包 jersey-container-servlet-core 不具备自动扫描资源类的功能。

2. REST 服务类型二

类型二对应的是图 2-7 中所示的例 2，相应的逻辑是不存在 Application 的子类但存在 Servlet 的子类。

阅读指南

REST 服务类型二对应的示例所在目录是：

jax-rs2-guide\sample\2\2simple-service-webapp-subservlet

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/2simple-service-webapp-subservlet

---

本例定义了 Servlet 子类 AirServlet，该类继承自 org.glassfish.jersey.servlet.ServletContainer 类，这是 Jersey 2 中 Servlet 的基类，继承自 HttpServlet。AirServlet 类的代码示例如下。

```
@WebServlet(
initParams = @WebInitParam(
name = "jersey.config.server.provider.packages", value = "com.example"),
urlPatterns = "/webapi/*",
loadOnStartup = 1)
public class AirServlet extends ServletContainer {
```

AirServlet 使用了 WebServlet 注解来配置 Servlet 参数。包括初始化参数 initParams 中定义扫描的资源类所在的包名："com.example"，Servlet 匹配的资源路径：urlPatterns="/webapi/*"和启动时的加载标识：loadOnStartup=1。

例 2 是基于 Servlet 3 容器的 REST 服务，使用了 WebServlet 注解和无 web.xml 等 Servlet 3 引入而 Servlet 2 没有的功能。在自定义 Servlet 3.x 子类的场景下，web.xml 可以省略，但需要修改 Maven 的 maven-war-plugin 插件的配置，添加 failOnMissingWebXml 为 false，这样编译时才不会报错。Maven 配置文件中相关信息如下所示。

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-war-plugin</artifactId>
   <version>2.3</version>
   <configuration>
      <failOnMissingWebXml>false</failOnMissingWebXml>
   </configuration>
</plugin>
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>3.1.0</version>
   <scope>provided</scope>
</dependency>
```

3. REST 服务类型三

类型三对应的是图 2-7 中所示的例 3，相应的逻辑是存在 Application 的子类并且定义了 @ApplicationPath 注解。

阅读指南

REST 服务类型三对应的示例所在目录是：

1）jax-rs2-guide\sample\2\3simple-service-webapp-servlet2-rc

2）jax-rs2-guide\sample\2\3simple-service-webapp-servlet3-application

源代码地址：

1）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/3 simple-service-webapp-servlet3-application

2）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/3simple-service-webapp-servlet2-rc

---

REST 服务类型三的示例包含两个小项目。其中，servlet2-rc 项目基于 Servlet 2，AirResourceConfig 类继承自 Application 的子类 ResourceConfig 类；servlet3-application 项目基于 Servlet 3，AirApplication 类继承自 Application 类。基于 Servlet 2 的 REST 服务需要定义 web.xml（但内容可以是“空的”，即只有 web-app 的基本定义），基于 Servlet 3 的 REST 服务可以省略此文件。

AirApplication 类代码示例如下。

```
@ApplicationPath("/webapi/*")
public class AirApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(MyResource.class);
        return classes;
}
}
```

AirApplication 类覆盖了 getClasses() 方法，注册了资源类 MyResource，这样在服务启动后，MyResource 类提供的资源路径将被映射到内存，以便请求处理时匹配相关的资源类和方法。

AirResourceConfig 类代码示例如下。

```
@ApplicationPath ("/webapi/*")
public class AirResourceConfig extends ResourceConfig {
   public AirResourceConfig() {
       packages ("com.example") ;
   }
}
```

AirResourceConfig 类在构造子中提供了扫描包的全名，这样在服务启动后，“com.example”包内资源类所提供的资源路径将被映射到内存。

4. REST 服务类型四

类型四对应的是图 2-7 中所示的例 4，相应的逻辑是“一有二无”：一有是存在 Application 的子类；二无是不存在 Servlet 子类、不存在或者不允许使用注解 @ApplicationPath。

阅读指南

REST 服务类型四对应的示例所在目录是：

1）jax-rs2-guide\sample\2\4simple-service-webapp-servlet2-application

2）jax-rs2-guide\sample\2\4simple-service-webapp-servlet3-application

源代码地址：

1）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/4simple-service-webapp-servlet2-application

2）https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/4simple-service-webapp-servlet3-application

---

REST 服务类型四的示例包含两个小项目，演示了基于 Servlet 2 和 Servlet 3 两个版本的 REST 服务，其差异仅此而已，关于差异性配置前面的例子已经讲过，不再赘述。下面以

servlet3-application 为例说明。

AirApplication 类是 Application 的子类，代码示例如下。

```
public class AirApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(MyResource.class);
        return classes;
    }
}
```

上述代码和类型三的示例相仿，但是该类没有定义 @ApplicationPath 注解，因此我们需要在 web.xml 中配置 Servlet 和映射资源路径。其代码示例如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"xsi:schemaLocation=
"http://java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java EE/
web-app_3_0.xsd" version="3.0">
    <servlet>
        <servlet-name>com.example.AirApplication</servlet-name>
    </servlet>
    <servlet-mapping>
        <servlet-name>com.example.AirApplication</servlet-name>
        <url-pattern>/webapi/*</url-pattern>
    </servlet-mapping>
</web-app>
```

在 servlet-name 中使用自定义的 Application 子类 com.example.AirApplication 的全名作为 Servlet 名称，并在 url-pattern 中映射资源路径。

2.4 REST 应用描述

在明白如何创建和部署各种类型的 REST 服务后，我们来了解一下部署好的 REST 服务中的一个特殊的成员——REST 应用的描述：以 XML 格式展示当前 REST 环境中所提供的 REST 服务接口。这种 XML 格式的描述就是 WADL。

WADL（Web Application Description Language，Web 应用描述语言）是用来描述基于 HTTP 的 REST 式 Web 服务部署情况的。它采用 XML 格式，支持多种数据类型的描述。WADL 由 Sun 公司提出，尚未成为 W3C 或者 OASIS 的标准，JAX-RS 标准中并没有关于 WADL 的定义和说明。Jersey 作为 JAX-RS 2.0 的参考实现，默认支持服务的 WADL。通过浏览器访问“服务根路径 /application.wadl”即可打开该服务的 WADL 内容。相对于 WADL，WSDL 更为人们所熟知，WSDL 是 RPC 风格的基于 SOAP 的 Web 服务的描述语言。两者缩写类似而且都使用 XML 格式，此外共性不多。

2.4.1 应用的描述

以 2.3 节的 REST 服务类型四的示例项目 simple-service-webapp-servlet3-application 为例，该应用的 WADL 路径如下：

```
http://localhost:8080/simple-service-webapp-servlet3-application/webapi/application.wadl
```

通过浏览器访问该路径，可以浏览 WADL 的 schema 结构。WADL 的最外层标签是 application，代表应用。然后自上而下分别是 doc、grammars 和 resources。resources 是应用提供的资源集合，里面至少包含 application.wadl，以及应用中包含的资源描述，比如本例的资源信息描述在资源路径 myresource 之内，如下所示。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application>
   <doc jersey:generatedBy="Jersey: 2.3 2013-09-20 13:59:07"/>
   <grammars/>
<resources
base="http://localhost:8080/simple-service-webapp-servlet3-application/webapi/">
        <resource path="myresource">...</resource>
        <resource path="application.wadl">...</resource>
   </resources>
</application>
```

2.4.2 资源的描述

可以展开 myresource 来查看具体某个方法的 WADL，也可以通过发送一条请求并定义请求头信息来获取。以 cURL（该工具的介绍见 2.6.1 节）为例，命令为：

```
curl -X OPTIONS -H "Allow: application/vnd.sun.wadl+xml" -v
http://localhost:8080/simple-service-webapp-servlet3-application/webapi/myresource
```

REST 服务类型四的示例项目 simple-service-webapp-servlet3-application 提供的资源接口，对照服务器返回的 XML，可以更加清晰地理解 WADL 的内容。其 WADL 内容如下所示。

```
<resource path="myresource">
  <method id="getIt" name="GET">
        <response>
              <representation mediaType="text/plain"/>
        </response>
  </method>
  <method id="apply" name="OPTIONS">
        <request>
              <representation mediaType="*/*"/>
        </request>
        <response>
              <representation mediaType="application/vnd.sun.wadl+xml"/>
        </response>
  </method>
  <method id="apply" name="OPTIONS">
```

```
        <request>
            <representation mediaType="*/*"/>
        </request>
        <response>
            <representation mediaType="text/plain"/>
        </response>
    </method>
    <method id="apply" name="OPTIONS">
        <request>
            <representation mediaType="*/*"/>
        </request>
        <response>
            <representation mediaType="*/*"/>
        </response>
    </method>
</resource>
```

在这段代码中，公布了 4 个方法。其中，getIt() 方法代码如下。其他 3 个 OPTIONS 请求方法是 Jersey 默认实现的，用以描述 getIt() 方法，分别返回 text/plain 类型、*/* 类型和 application/vnd.sun.wadl+xml 类型。

```
@GET
@Produces (MediaType.TEXT_PLAIN)
public String getIt() {
return "Got it!";
}
```

getIt() 方法定义为 GET 请求方法，@Produces 中定义的媒体类型是 MediaType.TEXT_PLAIN，即响应过程中生产的数据，其表述性状态以 text/plain 媒体类型转移。

2.4.3 WADL 的配置

上述 OPTIONS 请求方法的实现是 Jersey 默认支持的，如果读者不希望在 REST 服务中让 Jersey 自动生成，可以通过配置 jersey.config.server.wadl.disableWadl=true 来实现。代码示例如下。

```
public class AirApplication extends ResourceConfig {
    public AirApplication() {
        property (ServerProperties.WADL_FEATURE_DISABLE, true);
        packages ("com.example.resource");
    }
}
```

在构造函数中，我们通过定义 ServerProperties.WADL_FEATURE_DISABLE 属性为 true 实现去 WADL 自动生成的功能。

或者，可以通过修改 Web 配置文件中 servlet 启动参数来实现，代码示例如下。

```
<servlet>
   <servlet-name>com.example.AirApplication</servlet-name>
   <init-param>
```

```
        <param-name>jersey.config.server.wadl.disableWadl</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
```

配置文件中定义了启动参数 jersey.config.server.wadl.disableWadl，其值定义为 true，以实现去 WADL 自动生成的功能。

有关 Jersey 的配置详情和示例，读者可参考本书的 10.3 节。

2.5 第一个完整的 REST 服务

通过本章前面 4 节的讲述，我们已经知道如何为使用不同的配置，在不同的环境下使用 Jersey 实现 REST 式的 Web 服务了。但前面的示例不够完整，在快速实现的学习阶段，读者最希望看到的是一个完整的服务。本节将结合 Spring、JPA 和 jQuery，展示一个完整的 REST 式的 Web 服务，该服务用来完成对图书资源的存取。

阅读指南

2.5 节示例所在目录是：jax-rs2-guide\sample\2\5simple-service-webapp-spring-jpa-jquery。源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/5simple-service-webapp-spring-jpa-jquery。

实体部分抽象到通用模块中，所在目录是：jax-rs2-guide\sample\common。

作为示例，本节定义的资源的实体相对简单，只有一个图书实体。更复杂的示例可参考第 11 章的综合示例。

2.5.1 定义资源

REST 服务的核心是对外公布的资源 API。定义资源通常包括资源实体及其表述的设计、资源路径的定义，最后是使用 JAX-RS 2.0 对定义好的资源 API 进行编码实现。在本例中，我们要定义并发布图书资源存取的 REST 服务。

因此，资源定义包括以下几个方面：

- 图书实体类的定义。
- 资源路径的设计。
- 通过图书资源类实现图书资源存取服务。

1. 图书实体类

首先定义图书资源的实体类，最简的图书类应包括主键、图书名称和出版人。示例代码如下。

```
@XmlRootElement
public class Book{
    private Long bookId;
    private String bookName;
    private String publisher;
```

Book 类是一个简单的 POJO，使用了 JAXB 的注解 @XmlRootElement，以实现 XML 和 POJO 之间的转换。@XmlRootElement 标识 Book 作为 XML 结构的根元素。Book 类包含一个长整型的主键字段 bookId、一个字符串型的图书名称字段 bookName 和一个字符串类型的出版社字段 publisher。

2. 资源路径

有了图书实体类，接下来根据业务为其定义资源路径，以对外提供 REST 服务。图书资源路径见表 2-1。

表 2-1　图书资源路径列表

资源路径	REST 接口说明	HTTP 方法	输入	输出
/books	获取全部图书资源	GET		Books 对象
/books/{bookId:[0-9]*}	通过主键获取指定图书资源	GET	bookId	Book 对象
/books/book?id=…	通过主键获取指定图书资源	GET	id	Book 对象
/books	新增图书资源	POST	Book 对象	Book 对象
/books/{bookId:[0-9]*}	通过主键更新指定图书资源	PUT	bookId Book 对象	Book 对象
/books/{bookId:[0-9]*}	通过主键删除指定图书资源	DELETE	bookId	删除结果字符串

表 2-1 列出了最基本的资源存取功能，其中定义了 3 个查询接口，包括查询全部图书资源接口和根据主键 ID 查询指定图书资源的两种形式的接口；写入接口包括增加图书、根据主键删除和修改图书等 3 个接口。在资源路径的绑定定义中，可以使用正则表达式，详情见 3.2 节。

在增加、删除、修改、查询的 REST 接口定义中，新增和修改功能该使用 POST 还是 PUT，意见并不统一。笔者秉持的是 ROA 风格，修改使用 PUT 方法，新增使用 POST 方法。这看似字面略有不同的定义背后，是对 REST 式的 Web 服务的正确解读。定义资源路径就是设计 REST 接口的过程，一方面需要对业务有深刻的理解，另一方面需要对 REST 风格有真知灼见。3.1 节将着重讲述 REST 风格的 API 设计。

REST 接口的实体类定义中，要考虑集合数据的处理。本例对 Book 类的集合封装了 Books 类，用来专门作为返回值使用的 POJO。示例代码如下。

```
@XmlRootElement (name = "books")
public class Books implements Serializable {
private List<Book> bookList;
    @XmlElement (name = "book")
//关注点 1: JAXB 包装元素
    @XmlElementWrapper
    public List<Book> getBookList() {
        return bookList;
```

```
    }
    ...
}
```

在这段代码中，Books 类是 Book 的集合类型，以集合封装一类资源，其结果集逻辑清晰、结构工整。值得注意的是 @XmlElementWrapper 注解的使用，见关注点 1。在包装了一层 books 集合后，所得到的 XML 或者 JSON 格式的数据将多一层封装。这在 Web 页面渲染解析时，需要谨慎编写相关的处理脚本。

3. 图书资源类

接下来是图书资源类 BookResource，该类是对上述设计的具体实现。和本章前述示例相同的是使用 @Path、@GET 等注解绑定资源路径和资源方法，本例较本章前面的例子更完整，引入了逻辑中间层。

图书资源类 BookResource 定义如下。

```
@Path("books")
public class BookResource {
private static final Logger LOGGER = Logger.getLogger(BookResource.class);
// 关注点 1: Service 实例用于处理业务逻辑
@Autowired
private BookService bookService;
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Books getBooks() {
        final Books books = bookService.getBooks();
        BookResource.LOGGER.debug(books);
        return books;
    }
    @Path("{bookId:[0-9]*}")
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Book getBookByPath(@PathParam("bookId") final Long bookId) {
        final Book book = bookService.getBook(bookId);
        BookResource.LOGGER.debug(book);
        return book;
    }
    @Path("/book")
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Book getBookByQuery(@QueryParam("id") final Long bookId) {
        final Book book = bookService.getBook(bookId);
        BookResource.LOGGER.debug(book);
        return book;
    }
    @POST
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Consumes({MediaType.APPLICATION_JSON,
MediaType.APPLICATION_XML, MediaType.TEXT_XML})
```

```
    public Book saveBook(final Book book) {
        return bookService.saveBook(book);
    }
    @Path("{bookId:[0-9]*}")
    @PUT
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML, MediaType.TEXT_XML})
    public Book updateBook(@PathParam("bookId") final Long bookId, final Book book) {
        if (book == null) {
            return null;
        }
        return bookService.updateBook(bookId, book);
    }
    @Path("/{bookId:[0-9]*}")
    @DELETE
    public String deleteBook(@PathParam("bookId") final Long bookId) {
        if (bookService.deleteBook(bookId)) {
            return "Deleted book id=" + bookId;
        } else {
            return "Deleted book failed id=" + bookId;
        }
    }
}
```

从上面的代码片段中可以看出，资源类 BookResource 实现了设计中定义的 6 个接口，分别处理图书资源的查询和写入。根据 HTTP 请求方法和 URI 分派到匹配的方法，并将方法的处理结果返回。在每个方法中，具体业务逻辑由使用 @Autowired 注解的业务逻辑类 BookService 来处理，见关注点 1。@Autowired 是 Spring 的 IoC 容器自动探测 Bean 实例所使用的注解。需要注意的是，Jersey 内部使用的 IoC 容器是 HK2，因此集成 Spring 需要扩展包才能实现。

2.5.2 集成 Spring

Spring 作为轻量级 Java EE 开发框架，降低了中间件和逻辑层的开发难度，提升了开发效率。结合 Spring，可以轻松实现事务、Bean 的容器管理，以及面向切面的需求。因此，本节将介绍 Jersey 如何与 Spring 集成。

（1）定义依赖包

Jersey 对 Spring 的依赖定义如下所示。

```
<dependency>
  <groupId>org.glassfish.jersey.ext</groupId>
  <artifactId>jersey-spring3</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

org.glassfish.jersey.ext 这个包名是为 Jersey 扩展功能所使用的。1.4 节曾经介绍过 Jersey 的扩展包 jersey-spring3，该扩展包是从 Jersey 2.2 开始随 Jersey 2.x 发布的，因此，需要注意

的是，如果要集成 Spring，使用的 Jersey 版本不要小于 2.2。

(2) 使用依赖注入

定义好依赖包，我们就可以在当前项目中使用 Spring 的特性了。BookResource 类中使用的 @Autowired 注解的图书逻辑类 BookService 定义如下所示。

```
@Service
public class BookService {
@Path("books")
public class BookResource {
    @Autowired
    private BookService bookService;
```

BookService 类使用了 Spring 的 @Service 注解，表示该类是一种作为服务的组件（@Component）。

和 Jersey 的注册或者扫描类路径相似，Spring 的 IoC 是通过配置文件来定义的。Spring 的配置文件默认路径和名称为：src\main\resources\applicationContext.xml。和容器管理 Bean 相关的配置如下所示。

```
<context:component-scan base-package="com.example" />
```

这一行配置的作用是，当系统启动时，Spring 会根据该配置扫描 com.example 路径下所有类，将注解为 @component 的类以及注解为 @Service 这样的本身使用了 @component 注解的类加载。

(3) 面向切面的事务管理

集成 Spring 的一个优点是可以将数据库事务管理交给容器。需要注意的是，JPA 的写操作需要显式地提交事务，如果不通过容器管理事务，就得在每个写操作的代码块中，加入事务管理的代码，这样的模板代码显然是开发者不喜欢编写和阅读的。使用 Spring 后，在其配置文件 applicationContext.xml 中使用一行 XML 配置，即可定义统一的容器管理事务功能。示例如下。

```
<tx:annotation-driven transaction-manager="transactionManager" />
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

这里用到了 transactionManager，这是面向切面进行事务管理实例，其内部与 JPA 工厂类相连，为 Spring 容器提供了数据库访问切面的控制管理能力。

在写操作的方法上使用注解 @Transactional 即可实现容器级别的 AOP 事务处理。示例代码如下：

```
@Transactional
public void save(final Book entity) {
  entityManager.persist(entity);
}
```

save() 方法在开始执行前，Spring 容器会启用事务的 begin 方法，在 save() 方法成功执

行后，Spring 容器会 commit 该事务，否则 rollback 该事务。

(4) 集成测试

在完成了对 Spring 的集成后，我们要为各个类写单元测试代码。那么如何在测试中保持 Spring 容器的存在性呢？示例如下。

```
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class TUMyServiceTest {
...
}
```

集成测试代码使用 @RunWith 注解利用 Spring 的 SpringJUnit4ClassRunner 类，使 JUnit 测试环境具备 Spring 容器的功能，可以在单元测试和集成测试中模拟 Spring 容器生产环境的场景。

2.5.3 集成 JPA

JPA 2.1 是 Java EE 7 的标准之一，本例使用的 JPA 是其参考实现 EclipseLink。数据库驱动使用 MySQL 5.1。

1. JPA 2.1 配置

JPA 2.1 的配置默认使用 src\main\resources\META-INF\persistence.xml 文件。示例代码如下。

```
<persistence-unit name="jpaMysql" transaction-type="RESOURCE_LOCAL">
  <!-- provider -->
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  //关注点 1: 实体类定义
  <class>com.example.domain.Book</class>
  <!-- Connection JDBC -->
  <properties>
       //关注点 2: JPA 配置
       <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
       <property name="javax.persistence.jdbc.url"
       value="jdbc:mysql://localhost:3306/simple_service_book"/>
       <property name="javax.persistence.jdbc.user" value="root"/>
       <property name="javax.persistence.jdbc.password" value="root"/>
       <!--<property name="eclipselink.ddl-generation" value="create-tables"/> -->
  </properties>
</persistence-unit>
```

上述配置中定义了实体类 com.example.domain.Book，该类将对应数据库的“simple_book”表，见关注点 1。数据库 JDBC 的配置均使用 javax.persistence 前缀作为属性的键值，这是标准的 JPA 有别于使用 Hibernate 的地方，见关注点 2。

2. 数据表实体类

数据表的实体类用于将 POJO 和数据库的表进行一一映射。示例代码如下。

```
// 关注点 1: 实体注解
@Entity
@Table (name = "simple_book")
public class Book implements Serializable {
    private Long bookId;
    private String bookName;
    private String publisher;
 ...
// 关注点 2: 主键注解
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator (name = "EMP_SEQ")
    @Column (unique = true, nullable = false, name = "BOOKID")
    public Long getBookId() {
        return bookId;
    }
// 关注点 3: 字段注解
    @Column (length = Book.NAME_LENGTH, name = "BOOKNAME")
    public String getBookName() {
        return bookName;
    }
    @Column (length = Book.NAME_LENGTH, name = "PUBLISHER")
    public String getPublisher() {
        return publisher;
}
}
```

实体类 Book 使用 JPA 的注解 @Entity 和 @Table（name = "simple_book"），使 Object 和 Relationship 的数据库表映射起来，见关注点 1。主键使用数据库的 sequence 做自增管理。字段长度可以通过 @Column 注解的 length 参数来定义，字段名称通过 @Column 注解的 name 来定义，见关注点 2 和关注点 3。

3. MySQL 配置

本例使用 MySQL 5.6.1，UTF8 字符集，配置如下所示。

```
[client]
default-character-set=utf8
[mysql]
default-character-set=utf8
[mysqld]
character-set-server=utf8
basedir=D:/+aquarius/mysql-5.6.11-winx64
datadir=D:/+aquarius/mysql-5.6.11-winx64/data
#log
log-error=D:/+aquarius/mysql-5.6.11-winx64/logs/error.log
```

该配置是 MySQL 的简约配置，分别定义了 MySQL 客户端和服务端的字符集，服务器端 MySQL 路径、MySQL 数据路径和错误日志路径。

2.5.4　集成 jQuery

由于 REST 式的 Web 服务的客户端只需要关注 HTTP 请求和响应的展示，因此与服务器端天然解耦，开发的方法很多，可以根据具体需求，利用开源的工具进行快速开发。本节使用纯 HTML+JavaScript 库 jQuery 来实现 Browser/Server 模式的客户端。

jQuery 的优点是普及度高、易于上手，项目和社区非常活跃，文档和第三方支持良好。jQuery 的开发和执行的效率都比较理想，对于重构和需求的增量改进的响应比较迅速。

关于 jQuery 的文档和书籍很多，这里简述 3 个知识点。第一个知识点是 jQuery 的引入方式，我们选择了 WebJar，原因后面会讲；jQuery 的第二个知识点是 AJAX 请求的 API，因为 AJAX 是我们使用 jQuery 完成 REST 请求的基础；第三个知识点是 jQuery 的选择器的基本使用。

小白讲堂

jQuery 是一个活跃的 JavaScript 库项目，升级较为频繁。其版本信息可以通过 http://zh.wikipedia.org/zh-cn/JQuery 来获得。jQuery 的下载地址为 http://jquery.com/download。

---

1. 使用 WebJar 引入脚本库

在 Web 项目中，使用第三方的 JavaScript 库的传统方式有两种。一种是到其官网上下载合适的版本，然后将其放入项目的文件目录中，比如 webapp\lib\javascript 目录下；另一种是找到该依赖库合适版本的 CDN 地址，然后在依赖该库的脚本文件中引用该 CDN 地址。

下面是这两种方式的对比：

- 下载的好处是加载速度快，缺点是版本升级和管理不方便，源程序总体较大。
- CDN 的好处是源代码规模较小，但运行时加载时间较长。

那么有没有一种技术可以综合这两种方式的优点，却没有其缺点呢？ WebJar 就是基于这种思考产生的。WebJar 是一种利用 Maven 来管理和封装 JavaScript 库的 Jar 包文件，通过在 pom.xml 文件中显式地定义 Jar 包的版本号，轻松地管理项目对 JavaScript 库的依赖。Jar 包只在编译期下载，源程序中只有声明并不包含 Jar 包，因此源代码体积较小、从 SCM 中迁出速度快，且版本切换方便。可以看出，WebJar 具备了 CDN 方式中源代码体积小的优点和本地下载方式中运行时速度快的优点。目前，已经有很多脚本库支持 WebJar 这种技术（详见 http://www.webjars.org/），其中就包括 jQuery。因此，本示例将引入 jQuery 的 WebJar 来实现客户端页面的功能。

2. AJAX 的基本使用

虽然 AJAX 的概念已经广为人知，但是 jQuery 的官方文档还是对 AJAX 函数进行了详细介绍，读者可以通过 http://api.jquery.com/jQuery.ajax 获取。这里为了方便读者学习本例，只介绍相关的主要参数。

（1）$ 和 jQuery 符号

符号 $ 是 jQuery 的标准命名空间的缩略符号，用来代表 jQuery 对象。但是，如果项目

中同时使用了其他脚本库，就要面对一个问题：如果每个脚本库都是用 $ 符号来代表其内部对象，浏览器和开发者该如何区分呢？jQuery 框架提供了在与其他 JavaScript 库共存且 $ 被其他 JavaScript 库使用时避免冲突的声明。这一声明本可以使用任何字符，但约定俗成地使用 jQuery 这个名字。定义如下：

```
jQuery.noConflict()
```

（2）jQuery 的 AJAX 请求处理

jQuery 在处理异步请求时，使用 jQuery.ajax() 方法。该方法在 jQuery 发展过程中形成了两种不同参数的重载方法：

- jQuery 1.0 引入的 jQuery.ajax（[settings]）。
- jQuery 1.5 引入的 jQuery.ajax（url [, settings]）。

本节使用 jQuery 1.5 引入的 AJAX 方法版本。

jQuery 1.5 定义的 jQuery.ajax() 方法主要参数简述如下。

- url 参数是请求地址，settings 是可选的键值对参数集合。
- async 参数是发送异步请求，默认是 true，当不显式声明该键时，均发送异步请求。当设置为 false 时，执行同步请求。当请求为跨域或者 dataType 设置为 jsonp 时，同步请求无效。
- contentType 参数是请求内容的类型，对应 HTTP HEAD 中的 Content-Type，默认值为 'application/x-www-form-urlencoded; charset=UTF-8'，REST 应用中更为常见的 Content-Type 的设置是 "application/json" 和 "application/xml"，分别代表 JSON 和 XML 类型。
- data 参数是请求数据。
- dataType 参数是服务器返回数据的类型，可以是 xml、json、script、html、jsonp 和 text 类型。
- type 参数是请求方法类型，默认是 GET。本节示例还用到了 POST、PUT 和 DELETE。

（3）jQuery 的 AJAX 响应处理

在请求的响应阶段，如果请求成功，那么 jQuery.ajax() 使用回调函数“jqXHR.done(function (data, textStatus, jqXHR) {});”来处理响应返回值和渲染页面。jqXHR 是 jQuery 的 XMLHTTP-Request 对象。同时，jqXHR.done() 方法可以代替过时的 jQuery 方法 jqXHR.success()。如果请求失败，jQuery.ajax() 使用回调函数“jqXHR.fail(function(jqXHR, textStatus, errorThrown){});”来处理异常和故障并渲染页面。同时，jqXHR.fail() 方法可以代替过时的方法 jqXHR.error()。

3. Selector 的基本使用

jQuery 的一个显著的特点是提供了强大的选择器（Selector）。通过 $() 或者 jQuery() 可以获取指定的页面元素。下面示例展示了使用选择器读取 pathUrl 元素的值和写入 resultDiv 元素。

```
var url = $("#pathUrl").val();
jQuery('#resultDiv').html(errorThrown + " status=" + textStatus.status);
```

到此，相信读者已经掌握了本例的关键技术点，如果再结合本节的源代码进一步操练，一定会完成快速入门 JAX-RS 2.0 的任务。按照前面对 REST 服务的部署的讲述，部署并启动本示例，读者将看到如图 2-8 所示的首页。

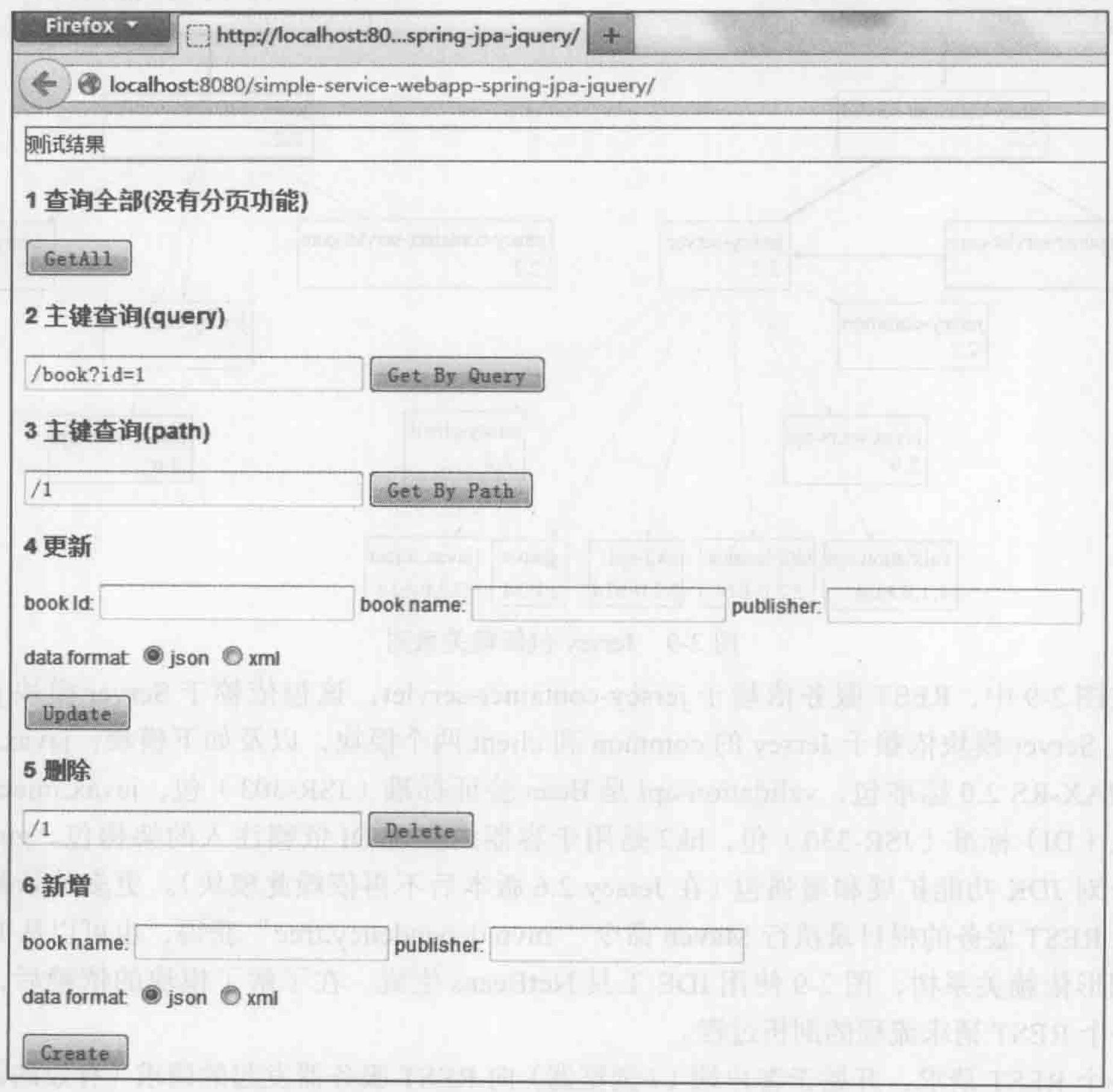


图 2-8　第一个完整的 REST 服务首页

在图 2-8 中，页面提供了图书资源的基本的增加、删除、修改、查询功能。填充输入框并单击相应按钮，流程将进入 jQuery 实现的客户端请求，每个请求都指向本示例服务器端提供的一个 REST 资源地址。

在本章即将结束时，通过我们刚刚完成的这个完整的示例，来一起学习一下 Jersey 2.x 对 REST 请求处理的流程分析。

2.5.5　请求处理流程分析

在进入 Jersey 2.x 请求处理流程分析之前，我们首先要清楚流程的上下文。这个流程是

存在于 Servlet 容器中的，Servlet 的核心处理依赖于 Jersey 的 Server 模块。首先从宏观上要弄清楚 REST 服务开发中的依赖关系。本例的依赖关系如图 2-9 所示。

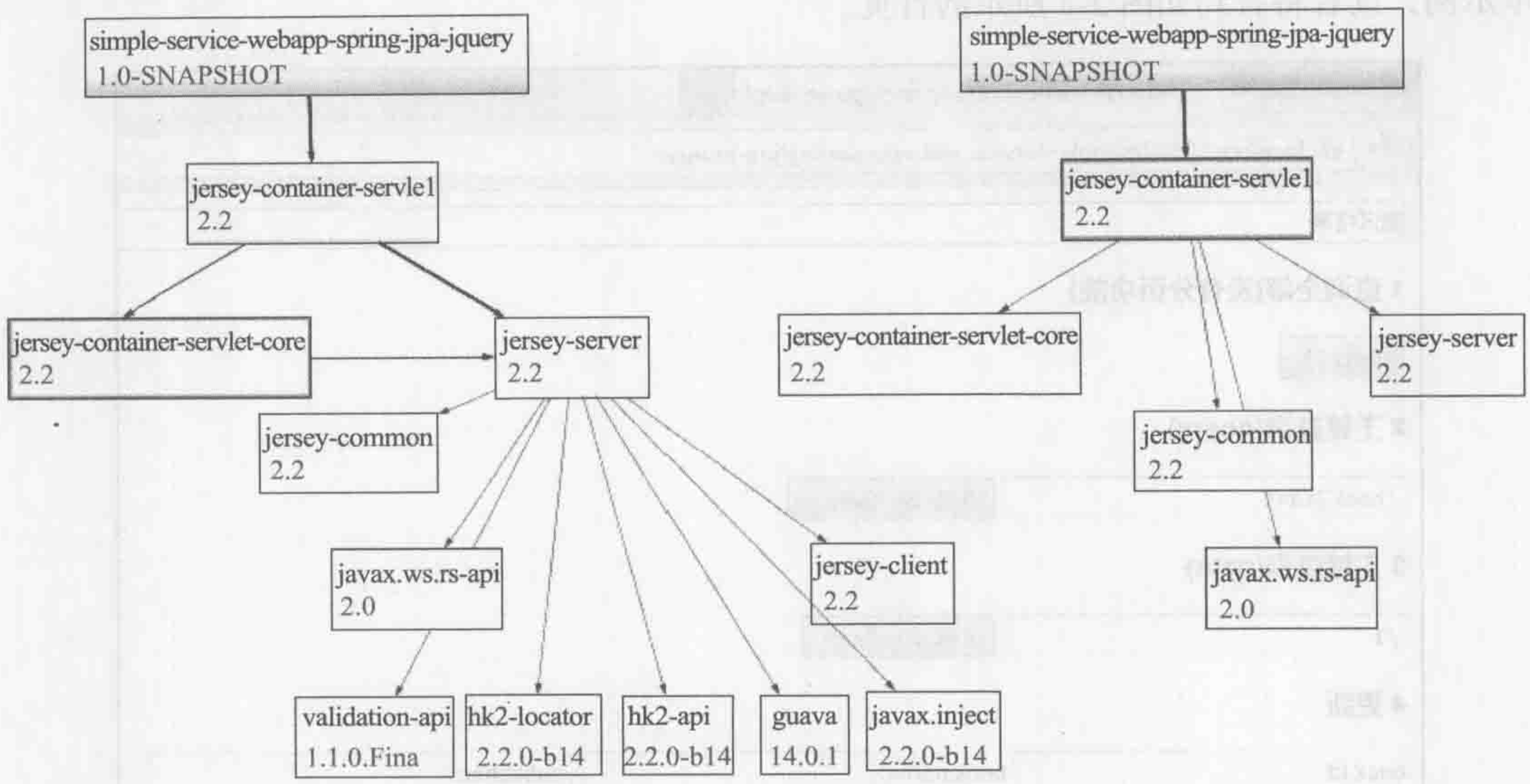


图 2-9 Jersey 包依赖关系图

在图 2-9 中，REST 服务依赖于 jersey-container-servlet，该包依赖于 Server 模块 jersey-server。Server 模块依赖于 Jersey 的 common 和 client 两个模块，以及如下模块：javax.ws.rs-api 是 JAX-RS 2.0 标准包，validation-api 是 Bean 验证标准（JSR-303）包，javax.inject 是依赖注入（DI）标准（JSR-330）包，hk2 是用于容器内实现 DI 依赖注入的架构包。guava 是 Google 对 JDK 功能扩展和增强包（在 Jersey 2.6 版本后不再依赖此模块）。更多的依赖关系可以从 REST 服务的根目录执行 Maven 命令“mvn dependency:tree”获得，也可以从 IDE 中得到图形依赖关系树，图 2-9 使用 IDE 工具 NetBeans 生成。在了解了模块的依赖后，我们开始一个 REST 请求流程的剖析过程。

一个 REST 请求，开始于客户端（/浏览器）向 REST 服务器发起的请求（有效的请求必须包含一个有效的 RESTful Web Service 资源的地址），结束于服务器返回一个客户端可接收的资源表述（比如 JSON）。因此，流程分析的关键切入点有两个：

- 将请求地址映射到资源类的相应方法上，并执行该方法。
- 将返回值转换成请求所需要的表述，并返回给客户端。

对于第一个切入点，是请求阶段的核心过程，我们可以将其分解为如下 3 个步骤。

- URI 到服务入口：从客户端向服务器的 URI 发起请求到服务器端接收该请求。
- URI 资源定位和方法匹配：服务器端处理该请求并匹配相应的资源方法。
- 资源方法调用：调用匹配到的资源方法。

对于第二个切入点，处于请求处理的响应阶段，我们可以将其分解为如下两个步骤。

- 准备响应实例：从资源方法结束到 Response 实例准备完毕。
- 处理表述：特定表述的拦截器实现类处理资源方法返回值，并最终返回响应到客户端。

理清了思路后，我们开始动手操作。本节使用 cURL 发起 REST 请求，资源地址是：http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1，客户端期待的表述媒体类型是 JSON，匹配的资源方法是 com.example.resource.BookResource.getBookByQuery(Integer)。请求命令如下：

```
curl -H " Accept:application/json"
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1
```

该脚本发送了一个 HTTP GET 请求，使用 -H 参数定义了 HTTP 头信息指定 Accept 类型，期待的响应实体类型是 JSON：Accept:application/json。

下面我们进入第一个切入点，分别讲解 3 个步骤的流程，从中可以掌握 Jersey 是如何将请求地址匹配到资源类的相应方法上，并触发该方法的。

1. 从请求到资源方法

首先，我们在第一步先来看从请求一个 URL 到进入资源方法的流程。

（1）service 入口方法

cURL 客户端发起请求后，服务器由 Servlet Container 类实例来接收请求并将处理分派下去。ServletContainer 类的继承关系树，如图 2-10 所示。

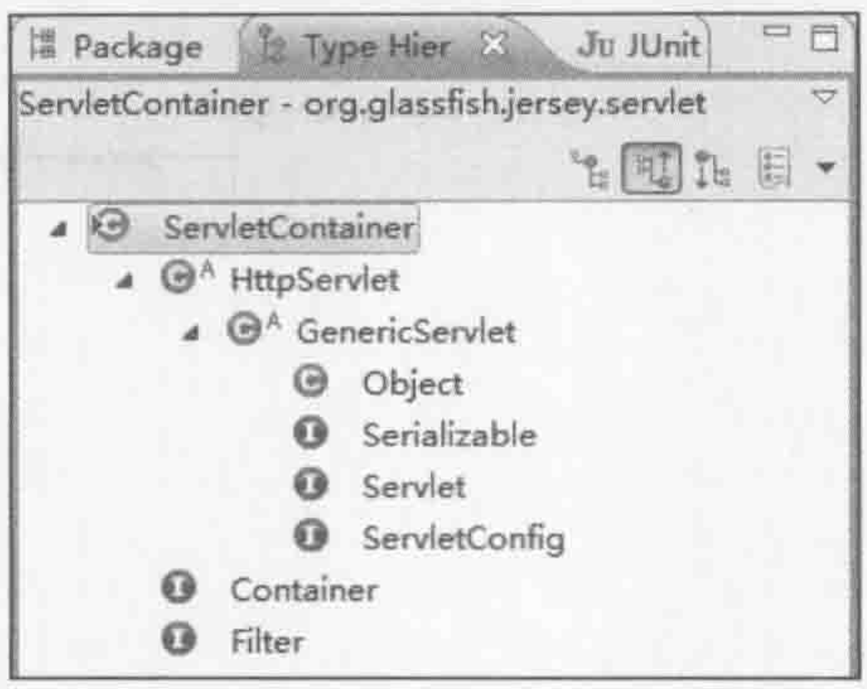


图 2-10　ServletContainer 继承关系

从图 2-10 可知，ServletContainer 是 HttpServlet 的子类，位于 Jersey 的 Servlet 容器包中（jersey-container-servlet-core-2.x.jar），作为 Servlet 接口的实现类同时实现了 JAX-RS 2.0 的过滤器接口 Filter 和容器接口 Container。我们知道，Servlet 接口的 service 方法是 Web 工程中请求的入口方法，任何 HTTP 方法的请求都要先经过 HttpServlet 类及其子类的 service 方法。因此，我们首先关注 ServletContainer.service 方法栈变量，示例如下。

```
ServletContainer.service(HttpServletRequest, HttpServletResponse) line: 248
baseUri
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/
requestUri
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1
```

断点栈是当前线程的快照，我们从中关注两个重要的变量：请求地址信息的 baseUri 和 requestUri，这两个变量将作为后续流程的输入。baseUri 的值为 http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/，这是 REST 服务的基本路径。requestUri 的值为 http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1，这是请求服务的资源路径。

从容器层级上看，请求上下文变量除了包括请求地址信息外，还包括请求头信息。这里我们关注请求头中的 accept 字段信息，该字段对流程分析至关重要。

```
ServletContainer.service (URI, URI, HttpServletRequest, HttpServletResponse) line: 372
requestContext.header {user-agent=[curl/7.26.0], host=[localhost:8080],
accept=[application/json]}
```

在断点栈中，ServletContainer.service 方法处理请求上下文的 header 信息，可以看到 accept 的值为 [application/json]，代表接收 JSON 类型的数据。

如果断点位于 service 方法，可以从断点继续单步执行，如果断点直接设在资源方法，可以观察栈调用信息，流程是按照 ServletContainer → WebComponent → ApplicationHandler → ServerRuntime 的顺序来处理 Container Request 的，参见图 2-11。到此，URI 到服务入口的分析就结束了。

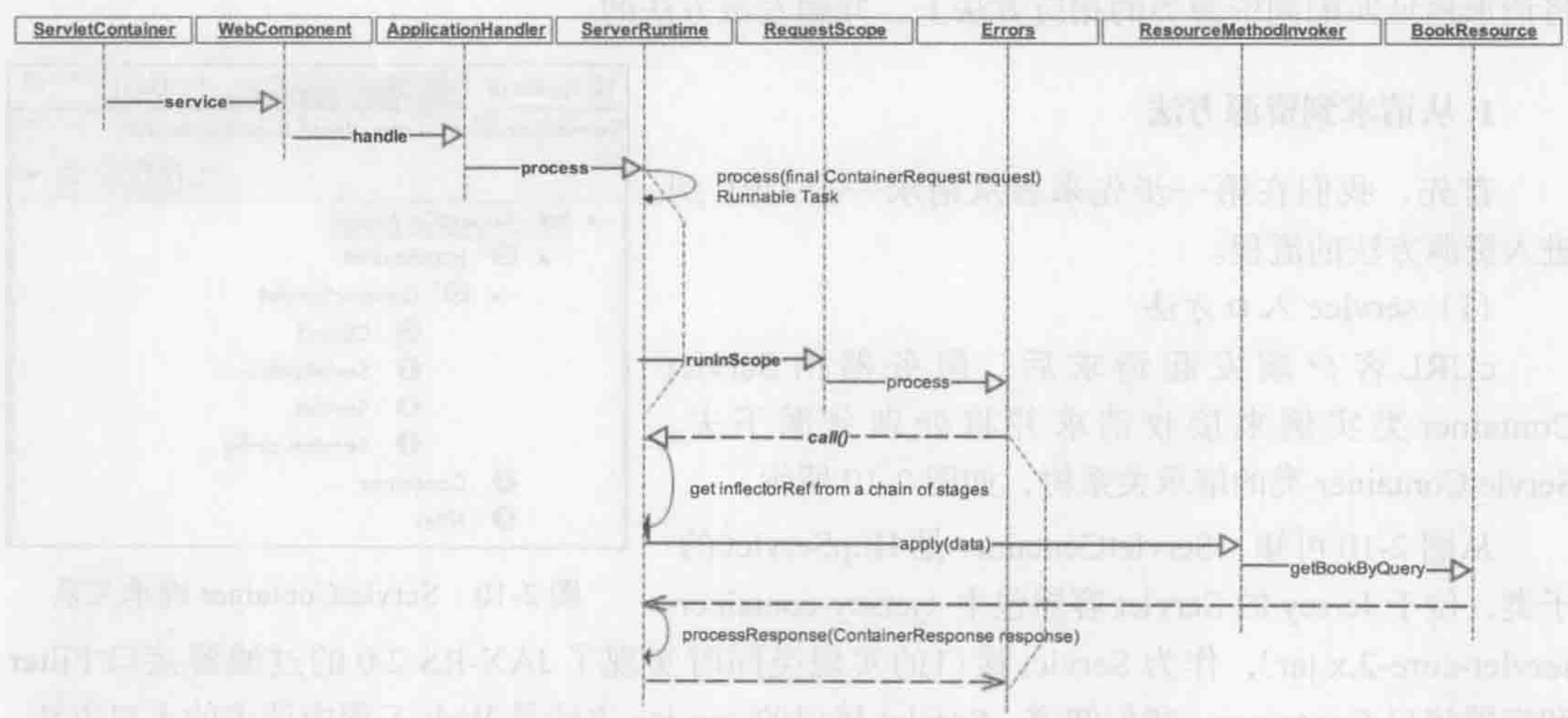


图 2-11 Jersey 请求处理时序图

如图 2-11 所示，在 ServerRuntime 类的 process 方法中，创建了一个线程任务，该任务在静态类 Errors 中被回调执行，Errors 会处理该任务出现的异常。因此，流程的重点就在这个线程任务中。

（2）匹配对应方法

接下来我们进入第二个步骤（URI 资源定位和方法匹配）的分析。刚刚创建的处理线程首先会调用静态类 Stages 从请求处理 Stage 链的根部一端走一遍到链的端部，逐个 Stage 调用其 apply 方法。示例代码如下。

```
final Stage<ContainerRequest> rootStage = Stages
  .chain(locator.createAndInitialize(ReferencesInitializer.class))
  .to(locator.createAndInitialize(ContainerMessageBodyWorkersInitializer.class))
  .to(preMatchRequestFilteringStage)
  .to(routingStage)
```

```
.to(resourceFilteringStage)
.build(routedInflectorExtractorStage);
```

本例是最简单的流程，因此没有其他 Stage 做 apply 操作。当在 Stages 类中找到这个端点后，会将其放入 inflectorRef 中，流程回到处理线程。示例代码如下。

```
ServerRuntime$1.run() line: 237
  final Ref<Endpoint> endpointRef = Refs.emptyRef();
  final ContainerRequest data = Stages.process(request, requestProcessingRoot, endpointRef);
  final Endpoint endpoint = endpointRef.get();
  ...
  asyncContextFactoryProvider.get().set(asyncResponderHolder);
  final ContainerResponse response = endpoint.apply(data);
  Stage.Continuation<DATA> continuation = Stage.Continuation.of(data, lastStage);
  while (continuation.next() != null) {
      lastStage = continuation.next();
      continuation = lastStage.apply(continuation.result());
  }
  inflectorRef.set(Stages.<DATA, RESULT, T>extractInflector(lastStage));
```

线程流程走到 endpointRef.get() 时返回端点实例，这个端点就是 ResourceMethodInvoker 实例。调用该实例的 apply 方法，流程就进入请求地址匹配的方法了。因此，第 2 步流程到这里完毕。

（3）调用匹配方法

接下来是第三步（URI 资源定位和方法匹配）的分析。ResourceMethodInvoker 实例会调用匹配好的 REST 请求处理方法。示例代码如下。

```
ResourceMethodInvoker.invoke(ContainerRequest, Object) line: 353
dispatcher.dispatch(resource, requestContext);
JavaResourceMethodDispatcherProvider$TypeOutInvoker(
AbstractJavaResourceMethodDispatcher).invoke(Object, Object...) line: 158
invokeMethodAction.run();
ResourceMethodInvocationHandlerFactory$1.invoke(Object, Method, Object[]) line: 81
BookResource.getBookByQuery(Long) line: 71
```

从断点栈中可以看到，最终调用的 REST 请求处理方法就是我们期待的 getBookByQuery() 方法，此后的 ResourceMethodInvocationHandlerFactory 的 invoke() 方法将调用这个方法处理对应的 REST 请求。

到此，请求地址到资源方法调用的流程就结束了。在处理业务逻辑方法 BookResource.getBookByQuery() 后，Jersey 将开始处理响应信息。

2. 响应和表述的处理

响应流程开始于资源方法执行结束。在资源方法完成业务逻辑处理后，也将结束 JDK 动态代理的 invoke() 方法。因此，观测点可以选择在 InvocationHandler 接口的 invoke() 方法中。示例代码如下。

```
@Singleton
public final class ResourceMethodInvocationHandlerFactory
implements ResourceMethodInvocationHandlerProvider {
    private static final InvocationHandler DEFAULT_HANDLER = new InvocationHandler() {
        @Override
        public Object invoke(Object target, Method method, Object[] args)
              throws IllegalAccessException,
              IllegalArgumentException, InvocationTargetException {
            //关注点 1: 反射调用资源方法
            return method.invoke(target, args);
        }
    };
```

(1) 准备响应实例

在 invoke() 方法中，target 是资源类实例，method 是资源方法，args 是该方法的输入参数。method.invoke(target, args) 一行通过反射机制执行该方法，见关注点 1。

流程会返回方法 method.invoke() 执行结果对象作为 invoke() 方法的返回值，然后在 JavaResourceMethodDispatcherProvider 类实例的分派方法 doDispatch() 中，将这个返回对象置入 response 的实体中。示例代码如下。

```
JavaResourceMethodDispatcherProvider$TypeOutInvoker.doDispatch(Object, Request) line: 198
o Book(id=2686)
Response response = Response.ok().entity(o).build();
```

在断点栈中，类型为图书实体类 Book 的实例 o 是 invoke() 方法的返回值，该对象作为响应的实体传入 HTTP 状态为 ok 的 Response 实例中。

REST 请求的响应信息至少包括响应实体（某种媒体格式的表述信息）和响应头信息（包括 HTTP 响应代码，通常一个成功的请求应该得到 200 OK）。这就是上述断点栈最后一行代码所做的事情。在设置好 response 的 entity 后，需要将该 entity 对象转化成请求所接受的表述返回给客户端 / 浏览器。但在此之前，需要将 ContainerResponse 响应对象返回给 ServerRuntime。

接下来 ServerRuntime 将使用 processResponse() 方法处理响应实例，示例代码如下。

```
private ContainerResponse processResponse(ContainerResponse response) {
...
  if(respondingRoot != null) {
      response = Stages.process(response, respondingRoot);
  }
  writeResponse(response);
...
}
```

processResponse() 方法主要做两件事情：第一，逐个 Stage 执行 apply，和前面 request 流程类似。第二，执行资源的后置过滤器和拦截器，拦截器会将返回值写向通向客户端 / 浏览器的通道。

写入返回值的流程起始于 ServerRuntime 的 process() 处理方法，该方法会将资源方法的

返回对象实体写入连接客户端的通道。

```
ServerRuntime$Responder.process(ContainerResponse) line: 351
ServerRuntime$Responder.writeResponse(ContainerResponse) line: 486
entity Book (id=7076)
executor WriterInterceptorExecutor (id=7125)
```

从栈中可以看到，这个阶段重点的两个对象是返回对象和写处理对象，流程要做的就是使用后者将前者恰当地写入返回客户端的通道中，这个“恰当”的工作就包括了对表述格式的处理。接下来，我们进入表述处理流程。

（2）JSON 表述

本例中我们定义的 accept 媒体格式是 JSON。REST 的表述不局限于 JSON，这里我们以 JSON 作为常用的表述类型为例，来研究 Jersey 的表述处理。

Jersey 对 JSON 的支持有 4 种方式，在 3.3 节会有详细的讲述。本例使用的是默认的方式，即使用 EclipseLink 项目的 MOXy 包来处理 JSON 数据，MOXy 包同时也是 JPA 的底层依赖（另一个著名的 JPA 实现是 JBoss 项目集下大名鼎鼎的 Hibernate）。MOXy 包依赖定义示例如下。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-moxy</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

继续刚才说到的响应流程的写入阶段，此时要做的事情就是将返回对象转换为 JSON 格式的响应实体。这正是 MOXy 要做的工作，如下是流程进入 MOXy 的代码栈信息。

```
JsonWithPaddingInterceptor.aroundWriteTo(WriterInterceptorContext) line: 91
WriterInterceptorExecutor$TerminalWriterInterceptor.aroundWriteTo(
WriterInterceptorContext) line: 188
ConfigurableMoxyJsonProvider(MOXyJsonProvider).writeTo(Object, Class<?>, Type,
Annotation[], MediaType, MultivaluedMap<String,Object>, OutputStream) line: 782
object Book (id=7076)
type Class (com.example.domain.Book)(id=568)
genericType Class (com.example.domain.Book)(id=568)
annotations Annotation[3] (id=7098)
mediaType AcceptableMediaType (id=7120)
httpHeaders StringKeyIgnoreCaseMultivaluedMap (id=7121)
entityStream CommittingOutputStream (id=7124)
```

在这段栈信息中，JsonWithPaddingInterceptor 是拦截器实现类，aroundWriteTo() 方法实现了 JSON 格式数据的写入。再往下跟踪，可以看到 JAXB 的“身影”，因为 MOXy 同样实现了 JAXB 标准，流程中处理 Java 对象为 JSON 数据的内部实现原理是以 XML 的 Marshal 方式处理 Java 对象成 XML 格式的数据（OXM:Object-XML-Mapping），然后再将 XML 转化为 JSON 数据（XML-2-JSON），详见 4.2 节。接下来的流程是将对象转化为 JSON 数据，这是一个 marshal 的过程。

接着流程会返回到处理线程，执行到如下一行。最后是将 response 返回并释放资源。

```
ServerRuntime$Responder.processResponse(ContainerResponse) line: 362
```

流程分析到此结束。希望读者可以通过这一次 REST 请求的分析之旅对 Jersey 的 REST 请求处理流程中的各个步骤有个总体印象，这对以后更好地编写 REST 服务代码会有帮助。也借此让初学者了解如何利用断点来调试和分析流程。

2.6 REST 调试工具

在上节中，我们不但知道了 REST 请求处理流程，而且对 IDE 中设置断点、观察服务器端运行时变量有了了解。本节将讲述如何在客户端对 REST 服务进行调试。

为何需要在客户端调试 REST 服务呢？因为在 REST 开发过程中，需要对请求资源地址、资源所支持的数据媒体类型和返回值类型等进行调试和测试。因此，掌握客户端的调试工具是开发优秀的 REST 服务的前提。接下来，将介绍这一领域常用的 REST 请求工具，以使读者更进一步地熟悉 REST 开发和调试。

2.6.1 命令行调试工具 cURL

cURL 是非常易用、强大的基于 URL 标准（RFC 3986）的命令行工具，通过命令行即可完成多种协议（比如 HTTP）的请求，并可以将请求的响应信息输出在终端 / 控制台上，因此对于调试和测试 REST 请求非常方便。在 2.5.5 节的流程分析中，我们已经使用过 cRUL，本节将进一步介绍这个工具的使用。

小白讲堂

下面是“维基百科"对 cURL 的解释。

cURL 是一个利用 URL 语法在命令行下工作的文件传输工具，1997 年首次发行。因为它支持文件上传和下载，所以是综合传输工具。但按照传统，习惯称 cURL 为下载工具。cURL 还包含了用于程序开发的 libcurl。

cURL 支持的通信协议有 FTP、FTPS、HTTP、HTTPS、TFTP、SFTP、Gopher、SCP、Telnet、DICT、FILE、LDAP、LDAPS、IMAP、POP3、SMTP 和 RTSP。

libcurl 支持的平台有 Solaris、NetBSD、FreeBSD、OpenBSD、Darwin、HP-UX、IRIX、AIX、Tru64、Linux、UnixWare、HURD、Windows、Symbian、Amiga、OS/2、BeOS、Mac OS X、Ultrix、QNX、BlackBerry Tablet OS、OpenVMS、RISC OS、Novell NetWare、DOS 等。

---

(1) 最简单的 cURL 命令

首先利用上一节示例所提供的 REST 服务，通过一个 REST 请求来掌握 cURl 的使用。启动上节开发的 REST 服务，然后在命令行中输入如下一行命令。

```
curl -H "Accept: application/json"
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/book?id=1
```

在上述命令中，curl 默认发送的是 HTTP GET 请求方法，-H 参数中定义了请求头信息，本例将通知服务器只接收 application/json 类型的表述。命令的最后是 HTTP 请求的地址信息，这个资源地址表示要获取图书 ID 为 1 的图书信息。在执行该命令后，REST 服务会给出响应，示例如下。

```
{"bookId":1,"bookName":"Java Restful Web Service 使用指南 ","publisher":"cmpbook"}
```

从结果可以看出，这是个 JSON 格式的数据，键值对描述了 ID 为 1 的图书信息。

(2) 获取区间信息的命令

第一个 cURL 示例请求的地址是一个唯一确定的图书资源信息，接下来的例子中的地址信息是一个区间，而不是唯一地址。

```
curl http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/{1,2}
```

从 cURL 命令的地址中可以看到，我们要获取的是一个区间信息，{1，2} 代表图书的 ID 是 1 和 2。因为我们没有在 cURL 命令中使用 -H 来定义请求头信息，因此返回实体默认的媒体格式是 XML 格式的表述。服务器的响应信息如下所示。

```
[1/2]: http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/
books/1 --><stdout>
--_curl_--http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/
books/1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="1"
bookName="Java Restful Web Service 使用指南 " publisher="cmpbook"/>
[2/2]: http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/
books/2 --><stdout>
--_curl_--http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/
books/2
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="2"
bookName="JSF2 和 RichFaces4 使用指南 " publisher="phei"/>
```

从上述结果可以看到，返回的结果集包含两条 XML 格式的数据，分别是 ID 为 1 和 2 的图书信息。

在 cURL 的帮助手册中，对使用一组 URL 作为请求地址的示例如下所示。

```
http://site.{one,two,three}.com
ftp://ftp.numericals.com/file[1-100].txt
ftp://ftp.numericals.com/file[001-100].txt (with leading zeros)
ftp://ftp.letters.com/file[a-z].txt
http://www.numericals.com/file[1-100:10].txt
http://www.letters.com/file[a-z:2].txt
http://any.org/archive[1996-1999]/vol[1-4]/part{a,b,c}.html
```

cURL 的请求不仅局限于 HTTP 请求，还包括 FTP 等协议。资源路径可以包含区间信息，小括号是开区间，中括号是闭区间，大括号是指定数值的集合。

(3) 请求写操作的命令

cURL 默认的请求方法是 GET，但不局限于只读请求。我们还可以使用 cURL 提交 XML 数据到服务器，完成写操作，示例如下。

```
curl -v -X POST -H "Content-Type:application/xml" -H "Accept: application/xml"
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books --data-binary
"<book bookName='JAX-RS2'/>"
curl -v -X PUT -H "Content-Type:application/xml" -H "Accept: application/xml"
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/2 -d
"<book bookName='JAX-RS2' publisher='CMP'/>"
```

cURL 命令的参数 -X 用于显式声明 HTTP 请求方法，示例中分别使用了 POST 和 PUT，参数 --data-binary（或者使用缩写形式 -d）定义提交的数据内容。这里需要注意的是，XML 格式的数据，其外部使用的是双引号，其内部使用的是单引号。

同样，对于提交 JSON 数据到服务器的示例如下。

```
curl -v -X PUT -H "Content-Type:application/json"
http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/3 -d
"{\"bookName\":\"JAX-RS2\",\"publisher\":\"CMP\"}"
```

JSON 数据只使用双引号，内部没有使用单引号。如果 JSON 内部使用单引号，服务器会对单独字符进行解析，因此，必须使用转义字符和双引号的组合来表示 JSON 数据内部的双引号。

在提交写请求后，可以使用基本的 cURL 请求对其进行验证，示例如下。

```
curl http://localhost:8080/simple-service-webapp-spring-jpa-jquery/webapi/books/3
```

因为上述的 PUT 请求定义了图书 ID 等于 3，并更新了该图书信息，所以验证过程只需请求 ID 为 3 的图书信息。

最后要注意的地方是，如果 REST 请求的服务器需要走代理，在 cURL 的请求命令中应使用参数 --proxy，示例如下。

```
curl --proxy <[protocol://][user@password]proxyhost[:port]>
```

cURL 的优点是简单方便，缺点是没有图形化界面。下面将介绍几款基于浏览器的扩展插件，作为 REST 客户端调试工具使用。

2.6.2 基于浏览器的图形化调试插件

cURL 功能强大，易于在自动化脚本中使用，但 cURL 的每个请求都要通过输入文字来完成，且没有图形界面，并不适于所有读者。下面将介绍几种基于浏览器的图形化调试插件，以方便读者在开发和测试 REST 服务时选择使用。

基于 Chrome 浏览器的 REST 插件有很多，本节将介绍其中的三种。每种的测试均使用照片存储社交网站 Flickr 公布的 REST API。对 Flickr 网站的 REST 请求需要用户 ID，读者可以通过访问 http://www.flickr.com/services/api/explore/flickr.people.getInfo 页面，获取用户在

Flickr 账户的有效值。

（1）Simple REST Client 插件

Simple REST Client 插件是基于 Chrome 浏览器的扩展，安装该插件后 Chrome 窗口的右上方会出现该插件的图标，以方便用户使用。该项目的地址是：https://github.com/jeremys/Simple-Rest-Client-Chrome-Extension，插件的下载地址是：http://chrome.google.com/extensions/detail/fhjcajmcbmldlhcimfajhfbgofnpcjmb。Simple REST Client 插件的界面如图 2-12 所示。

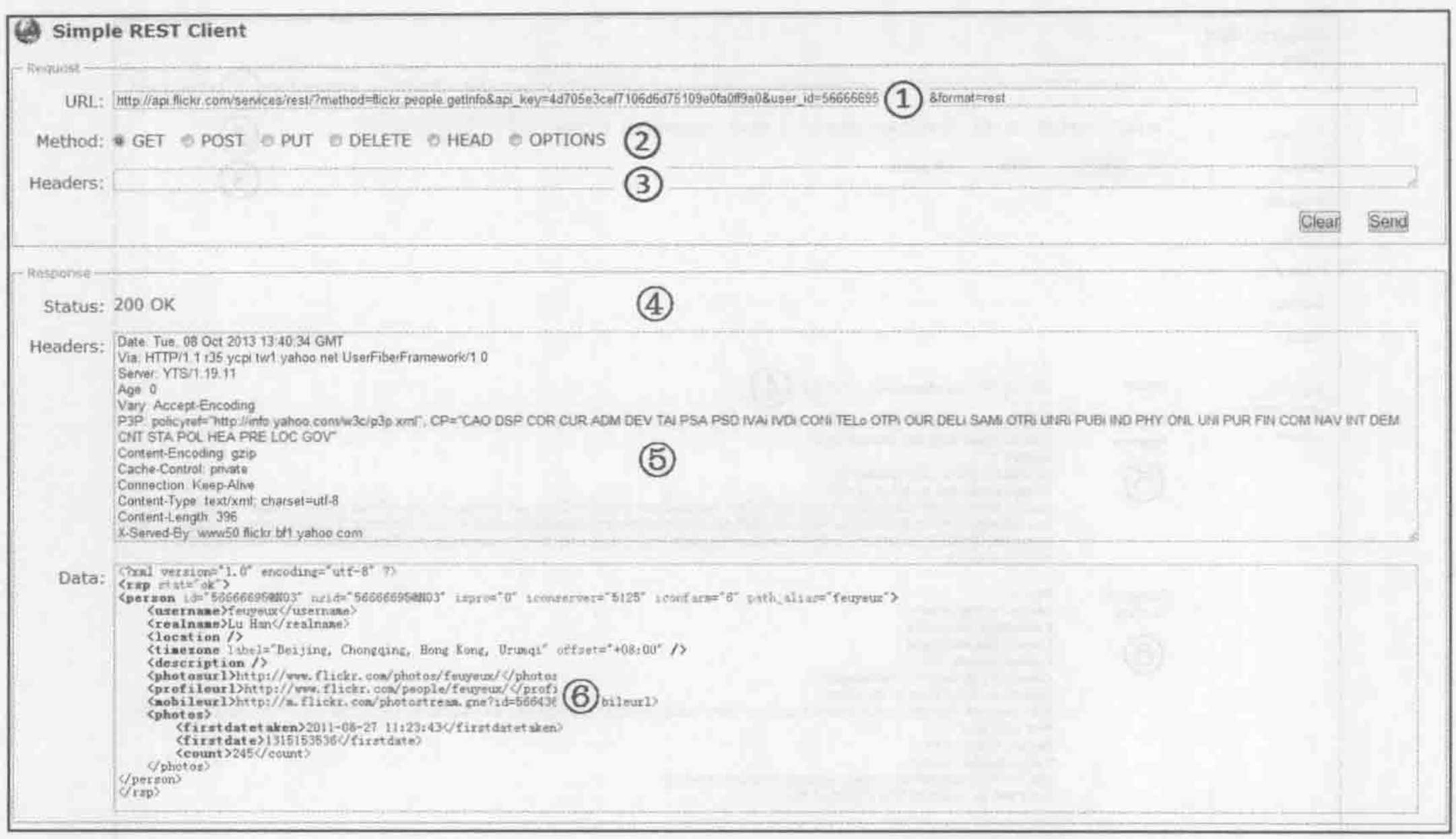


图 2-12　Simple REST Client 插件示意图

Simple REST Client 插件的特点是简单易用，其界面分为请求信息录入和响应信息展示上下两部分。录入部分包括 URL、HTTP 请求方法和请求头三部分，见图 2-12 中上方的数字标识 1 ~ 3。其中，HTTP 请求方法支持 HTTP 的标准方法 GET、POST、PUT、DELETE、HEAD 和 OPTIONS，Headers 部分需要完全手工输入。响应信息部分包括响应状态、响应头和响应实体三部分。其中，Headers 部分展示 HTTP 请求交互的响应头信息，Data 中展示的是响应实体信息，语法高亮显示，见图 2-12 中下方的数字标识 4 ~ 6。

总体上说 Simple REST Client 插件，虽小却功能比较齐全，但相比后面要讲的插件仍不够强大。

（2）Advance REST Client 插件

Advance REST Client 可以看做 Simple REST Client 的增强版。该项目的地址是：https://code.google.com/p/chrome-rest-client，插件的下载地址是：https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo。Advance REST Client 插件界面见

图 2-13 所示。

Advance REST Client 提供更为丰富的功能，除了 Simple REST Client 插件具备的输入和输出（见图 2-13 中的数字标识），Advance REST Client 还支持带参数的请求和提交表单等更完整的请求功能。数据格式上，支持原生的格式（Raw）、XMl 格式的响应（Response）信息。Advance REST Client 支持对请求地址的保存和对最近使用地址的记忆，如果调试中需要多次测试同一个资源地址，那么可以将其保存下来供以后使用；而多个这样的地址也可以按照项目分别保存，方便区分使用。

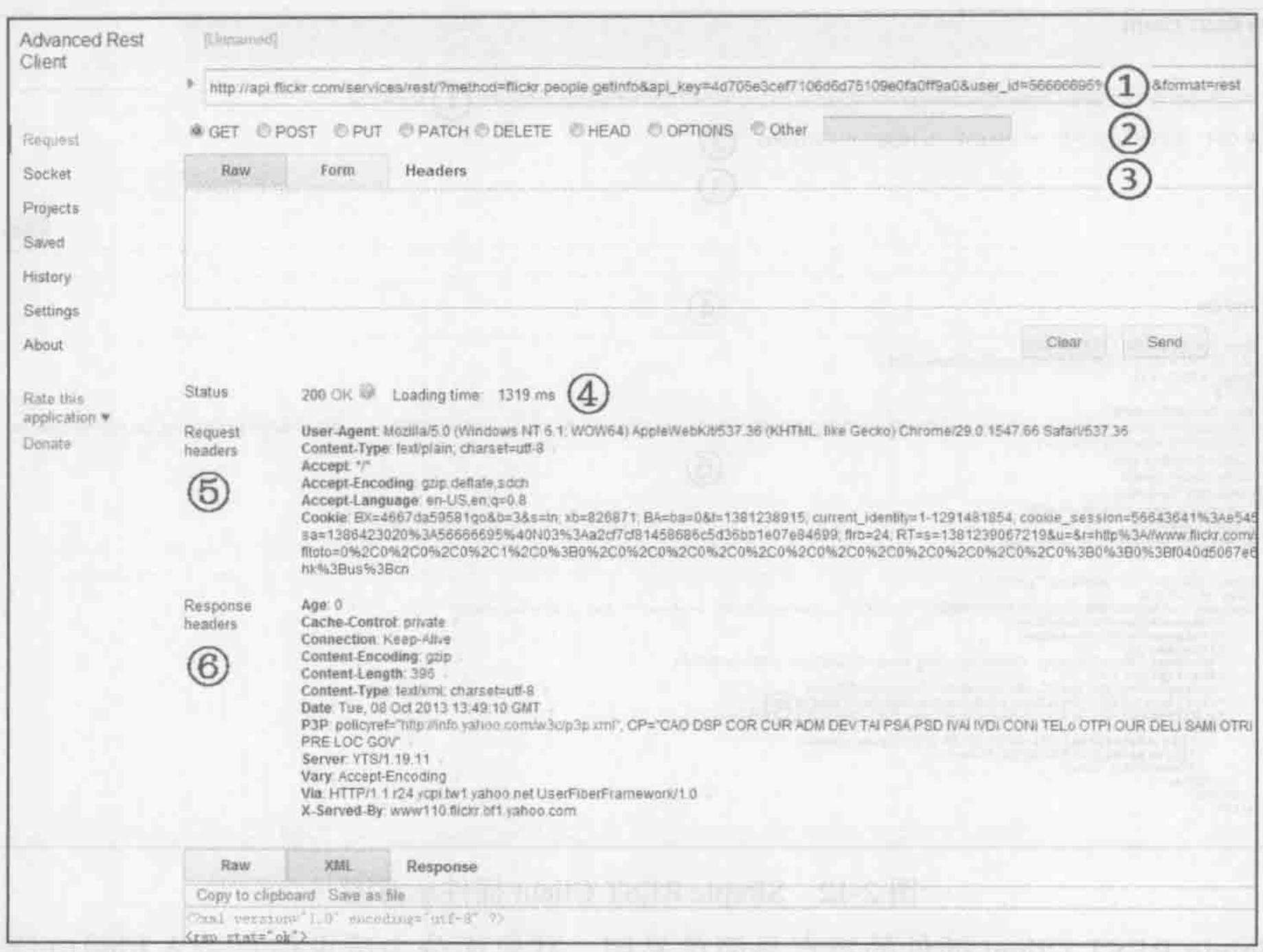


图 2-13 Advance REST Client 插件示意图

（3）Postman-REST Client 插件

Postman-REST Client 是基于 Simple REST Client 源代码编写的专门针对 REST 的插件。该项目的地址是：https://github.com/a85/POSTMan-Chrome-Extension，插件的下载地址是：http://www.getpostman.com。Postman-REST Client 插件界面如图 2-14 所示。

Postman-REST Client 提供的功能更多，除了 Advance REST Client 具有的输入和输出（见图 2-14 的数字标识部分），还可以发起基于安全的请求。请求方法不仅包括 HTTP 的标准方法，还包括 WebDAV 标准的方法。其响应信息的展示和支持的格式也更丰富。如果读者希望深入和细致地调试 REST 服务，Postman-REST Client 要比其他插件更加胜任。可以说三个插件的复杂度和功能性是递增的，使用哪一个要看读者的需求。类似的 Chrome 插件很多，读者如果有兴趣，可以通过 https://chrome.google.com/webstore/category/extensions 这个链接

访问 Chrome 的网上商店，搜索更多的 Chrome 插件。

（4）Firefox 插件

相对于 Chrome 浏览器，Firefox 的 REST 插件功能类似，其中常用的插件有 REST-Easy 和 RESTClient。REST-Easy 的项目地址是：https://github.com/nathan-osman/REST-Easy，RESTClient 的项目地址是：http://restclient.net。

类似的 Firefox 插件比较多，读者如果有兴趣，可以通过在 Firefox 浏览器中输入 about:addons 进入 Firefox 的扩展，搜索更多的相关插件。

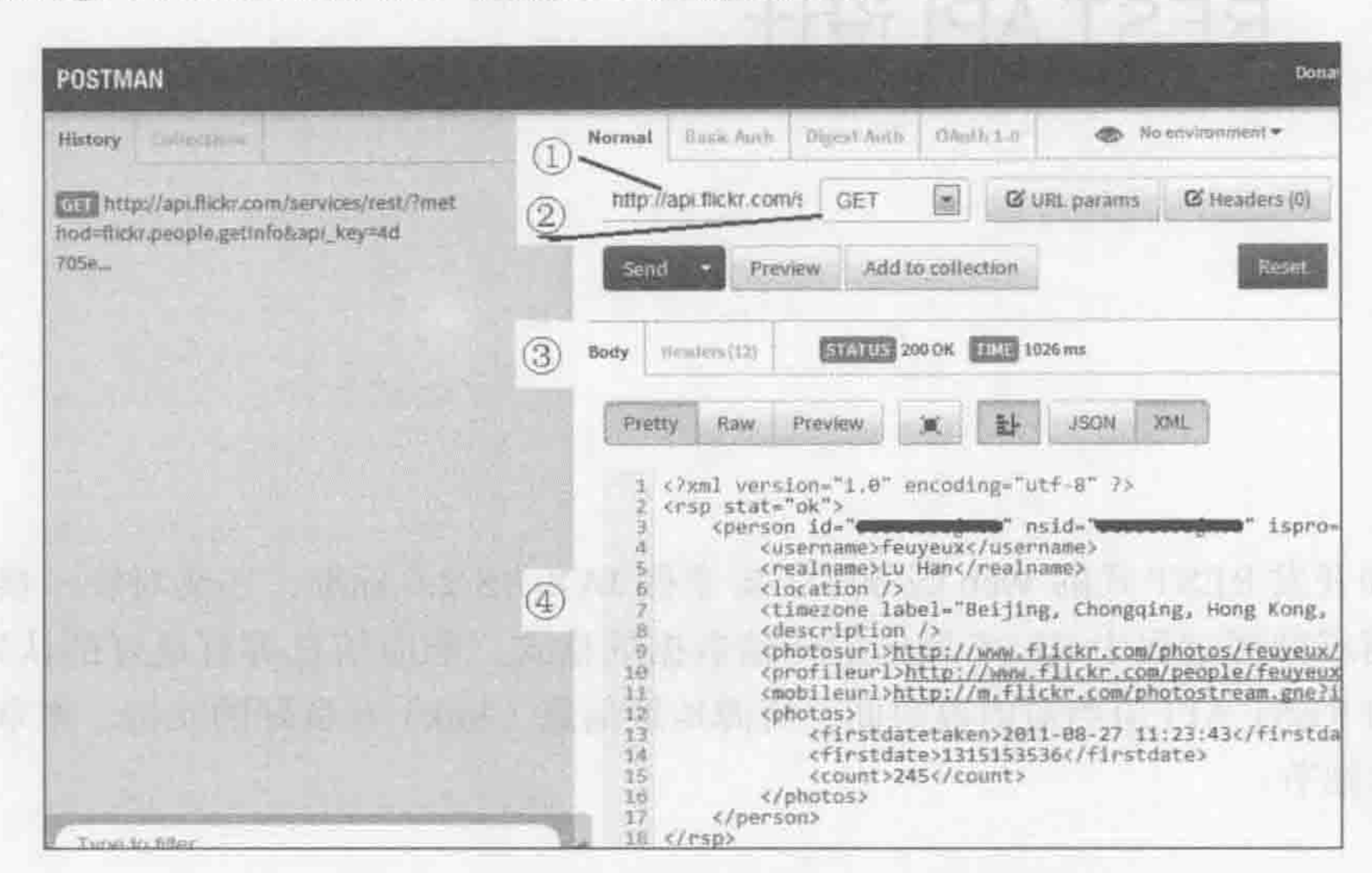


图 2-14　Postman-REST Client 插件示意图

2.7　本章小结

本章全面讲述了如何使用 JAX-RS 2.0 的参考实现 Jersey 2.x 来开发 REST 服务，并提供了一个完整的 REST 服务应用示例和流程分析。同时，介绍了 REST 开发过程中的调试工具。其中，2.1 节分析了基于 Maven 构建的 Jersey 官方的 Java SE 例子 simple-service 并基于示例实现了自定义资源的地址。2.2 节分析了基于 Servlet 环境的 Jersey 官方示例 simple-service-webapp，并演示如何在 Maven 插件、Servlet 容器和 Java EE 容器中运行该示例。2.3 节讲述了 JAX-RS 2.0 的 Application 和 Servlet 的 4 种存在类型。2.4 节讲述了 WADL 以及 Jersey 对 WADL 的支持。2.5 节是完整示例和 Jersey 处理 REST 请求的流程分析。2.6 节介绍了 REST 请求的 3 组调试工具。

通过本章的讲述，我们了解了 Jersey 开发的全貌，接下来的第 3 章我们要提高一个层次，掌握如何设计 REST API，以使我们的 REST 服务更标准和健壮。

Chapter 3 第 3 章

REST API 设计

设计和开发 REST 式的 Web 服务除了要掌握 JAX-RS 2.0 标准，还要对统一接口、资源定位以及请求处理过程中 REST 风格的传输数据的格式、响应信息等有良好的认知。此外，设计良好的 REST API 应当对内容协商、资源地址信息（link）有良好的支持。本章将详细讨论这些技术细节。

3.1 REST 统一接口

REST 式的 Web 服务和 RPC 式的 Web 服务在接口定义上的区别是，REST 使用 HTTP 的通用方法作为统一接口的标准词汇。REST 式的 Web 服务所提供的方法信息都在 HTTP 方法里，而 RPC 式的 Web 服务所提供的方法信息在 SOAP/HTTP 信封里（其封装的格式通常是 HTTP 或者是 SOAP），每一个 RPC 式的 Web 服务都会公布一套符合自己商业逻辑的方法词汇。

阅读指南

3.1 节示例所在目录是 jax-rs2-guide\sample\3\simple-service-3。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。

相关包：com.example.annotation.method。

每一种 HTTP 请求方法都可以从安全性和幂等性两方面考虑，这对正确理解 HTTP 请求方法和设计统一接口具有决定性的意义。换句话说，要定义严谨的 REST 统一接口，就需要真正理解 HTTP 方法的安全性和幂等性。

安全性代表安全的 REST 接口，是指外系统对该接口的访问，不会使服务器端资源的状

态发生改变。幂等性（idempotence）是指外系统对同一 REST 接口的多次访问，得到的资源状态是相同的。

阅读指南　这里讨论的安全性对应的英文是 Safety 而不是 Security，系统安全可参考第 6 章。

以下，将从 REST 统一接口的定义角度，逐个讲述 HTTP 方法。

3.1.1　GET 方法

REST 使用 HTTP 的 GET 方法获取服务提供的资源。GET 方法是只读的，那么它是幂等和安全的吗？答案马上揭晓。

1. 幂等性和安全性

HTTP 的 GET 方法用于读取资源。GET 方法是幂等的，因为读取同一个资源，总是得到相同的数据。GET 方法也是安全的，因为读取资源不会对其状态做改动。JAX-RS 2.0 指出了 @GET 注解对资源方法的定义，使得该方法用于处理 GET 请求。

值得注意的是，虽然 GET 方法的特性是幂等和安全的，但这不意味着任何一个定义为处理 GET 请求的方法都是幂等和安全的。换句话说，设计不良的 API 有可能违背 GET 的特性，将一个不该是 GET 的方法定义为之。

举个例子，在系统 B 中设计一个 REST 的 API，在客户端调用时读取系统 A 中 x 类型的数据，然后将 A.x 与系统 B 内的 y 类型数据做比较，如果两个集合的内容、最后更新时间上有不同，需要执行同步数据，即将 A.x 追加或者更新到 B.y 中。最后，将同步结果信息返回给向系统 B 发起请求的客户端，如图 3-1 所示。

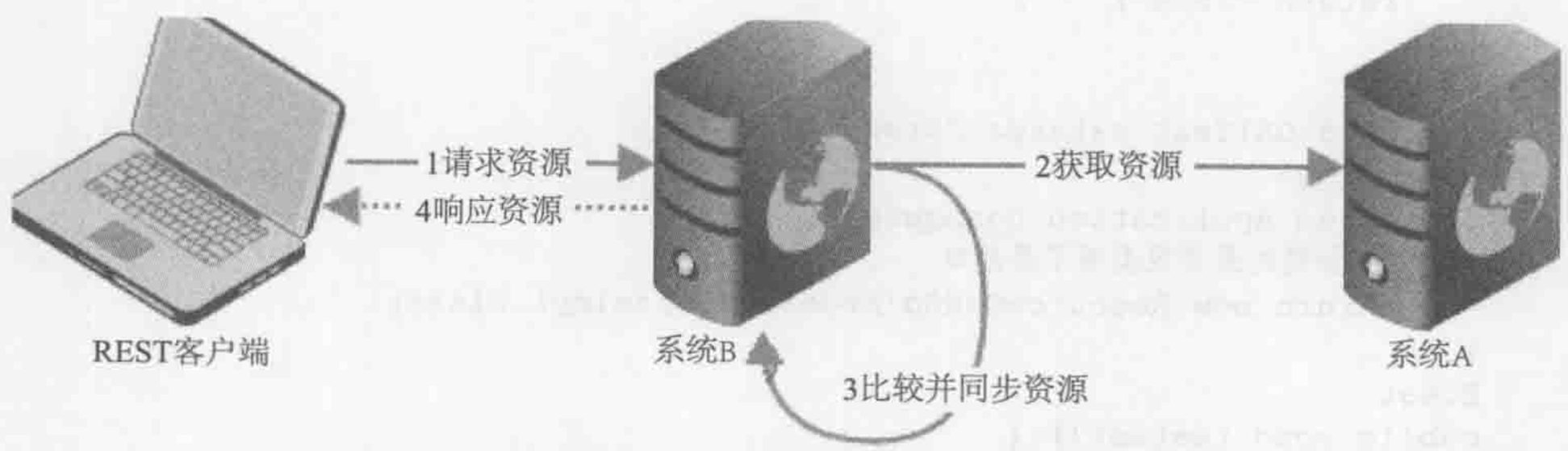


图 3-1　请求资源示意图

观察图 3-1 左侧部分，这是一个获取同步信息的 API，因此这个 API 的设计应该使用 GET 请求方法。但是，稍加分析后即可知道该场景并不具备使用 GET 的基本条件。因为同步过程中对系统 B 内的资源有写操作的可能，因此不具备安全性；而写的内容又不是每次相同，因此不具有幂等性。所以，这个例子应该定义的正确的请求方法是 POST。

2. 资源方法命名

不妨一起探讨一下上面这个同步信息的 API 该如何命名？既然是同步功能，那就以 sync 一类的字根作为前缀，这样所有的同步 API 都具有相同的开头，字迹也很工整。遗憾的是，这样的设计并不符合 REST 风格。笔者的理解是，从字面上看有两个问题：第一，sync 字根具有非名词性的含义，从 ROA 角度上看，sync 是 RPC 风格的命名：动词、自定义方法名称。第二，这样命名后，资源名称从一个主语变成了宾语，从 ROA 角度上看，面向的不再是资源，而是要执行的动作。

因此，标准的命名方式应该是单数的同步操作以资源名称命名，批量的同步操作以资源名称的复数名称命名。比如这个 API 是用于同步设备的，那么命名可以使用 device 和 devices。如果担心与普通查询业务资源地址混淆，那么可以在资源路径中增加查询或者路径参数，比如 device/id=1&source=a_b、device/b/a/ 等。

3. 抽象层注解资源

JAX-RS 2.0 的 HTTP 方法注解可以定义在接口和 POJO 中，置于接口中的方法名上更具抽象性和通用性。示例代码如下。

```
@Path("book")
public interface BookResource {
// 关注点 1: GET 注解从抽象类上移到接口
@GET
    public String getWeight();
}
public class EBookResourceImpl implements BookResource {
    // 关注点 2: 实现类无须 GET 注解
    @Override
    public String getWeight() {
        return "150M";
    }
}
public class GETTest extends JerseyTest {
    @Override
    protected Application configure() {
// 关注点 3: 加载的是实现类而不是接口
        return new ResourceConfig(EBookResourceImpl.class);
    }
    @Test
    public void testGet() {
        Response response = target("book").request().get();
        Assert.assertEquals("150M", response.readEntity(String.class));
    }
}
```

在这段代码中，资源接口 BookResource 定义了一个 GET 方法 getWeight()，这个方法上使用了 HTTP 方法注解 @GET，见关注点 1。资源接口 BookResource 的实现类 EBookResourceImpl 实现了 getWeight() 方法，但没有再次使用 @GET 注解。也就是说，在接

口中抽象地定义了资源的请求方法类型后，其全部实现类都无须再定义。这使得编码更整洁和抽象，见关注点 2。

最后，需要注意的是，在测试类 GETTest 中注册的是实现类 EBookResourceImpl 类型而不是接口 BookResource 类型，见关注点 3。

另外，我们一并介绍 HEAD 方法和 OPTIONS 方法。

HEAD 方法和 GET 方法相似，只是服务器端的返回值不包括 HTTP 实体。因此，HEAD 方法是安全的和幂等的。JAX-RS 2.0 定义了 @HEAD 注解来定义相关资源方法。

OPTIONS 方法和 GET 方法相似，是安全的和幂等的。OPTIONS 用于读取资源所支持的（Allow）所有 HTTP 请求方法。JAX-RS 2.0 定义了 @OPTIONS 注解来定义相关资源方法。

3.1.2 PUT 方法

PUT 方法是一种写操作的 HTTP 请求。REST 使用 HTTP 的 PUT 方法更新或添加资源。下面讲解一下 PUT 方法的作用和操作时的媒体类型。

1. 更新资源

因为 REST 只是风格，不是技术规范 / 标准，所以有些实现 REST 的细节没有明确的定义，这对实践而言，不可避免会产生某些误解。比如在创建和更新某个资源的时候，开发者比较迷茫的是何时该用 HTTP 的 PUT 方法，何时该使用 POST 方法。

为了解决这一问题，我们首先应该知道 PUT 方法的特性。PUT 方法是幂等的，即多次插入或者更新同一份数据，在服务器端对资源状态所产生的改变是相同的。PUT 方法不是安全的，有写动作的 HTTP 方法都不是安全的。

我们知道，使用同一份数据向服务器请求更新某一资源，得到的结果应该总是相同的，因此对于更新操作，使用 PUT 是没有疑问的。可能读者会想到最后更新时间字段每次提交会不同，但那已经不是同一份数据了。

2. 添加资源

而创建操作通常每次得到的结果是不同的，因为服务器端的业务层逻辑通常要求数据的主键字段要么来自于业务平台自增一个逻辑值，要么来自于数据库的主键自增。因此，相同的数据每一次提交到服务器端，都会为数据添加一个新的主键值，也就是创建一个主键值不同的新资源（如果没有业务或者外键冲突）。所以，创建操作通常应当设计为 POST 方法的 API。唯有一种场景应当使用 PUT 方法来设计 API，即客户端在发起创建请求时，在同一份数据中总可以提供唯一的主键值，服务器不会对其进行修改，这样的创建请求确保了幂等性，不应再使用 POST 方法。

JAX-RS 2.0 定义了 @PUT 注解来定义相关资源方法，示例代码如下。

```
@Path("book")
```

```
public interface BookResource {
// 关注点 1: PUT 方法
    @PUT
// 关注点 2: 资源方法定义了 Produces 注解和 Consumes 注解
    @Produces (MediaType.TEXT_PLAIN)
    @Consumes (MediaType.APPLICATION_XML)
    public String newBook (Book book) ;
}
public class PutTest extends JerseyTest {
    public static AtomicLong clientBookSequence = new AtomicLong();
    @Test
    public void testNew() {
        final Book newBook = new Book (clientBookSequence.incrementAndGet(),
        "book-" + System.nanoTime()) ;
        MediaType contentTypeMediaType = MediaType.APPLICATION_XML_TYPE;
        MediaType acceptMediaType = MediaType.TEXT_PLAIN_TYPE;
        final Entity<Book> bookEntity = Entity.entity(newBook, contentTypeMediaType);
        final String lastUpdate = target ("book") .request (acceptMediaType)
        .put (bookEntity, String.class) ;
        // 关注点 3: 资源方法定义了 Produces 注解和 Consumes 注解
        Assert.assertNotNull (lastUpdate) ;
        LOGGER.debug (lastUpdate) ;
    }
```

在这段代码中，资源接口 BookResource 使用 @PUT 注解定义了 newBook() 方法，即该方法用于处理相对资源路径为“book”的 PUT 请求，见关注点 1。单元测试类 PutTest 对其功能性进行了验证，对 lastUpdate 使用非空断言，lastUpdate 是更新方法 newBook() 的返回实体的值，代表最后更新时间戳，见关注点 3。

我们注意到，在 newBook() 方法上，同时定义了 @Produces（MediaType.TEXT_PLAIN）注解和 @Consumes（MediaType.APPLICATION_XML）注解，见关注点 2，下面我们来介绍一下与关注点 2 相关的媒体类型知识。

3. 媒体类型

PUT 方法执行写操作的非安全的 HTTP 方法，需要考虑请求实体媒体类型和响应实体媒体类型。请求实体媒体类型使用 HTTP 头的 Content Type 定义，响应实体媒体类型使用 HTTP 头的 Accept 定义。

在服务器端，@Consumes（MediaType.APPLICATION_XML）定义了服务器端要消费的媒体类型，即消费客户端请求实体的媒体类型。@Produces（MediaType.TEXT_PLAIN）定义了服务器端生产的媒体类型，即服务器产生的响应实体的媒体类型。

客户端在提交非安全性 HTTP 请求方法前，在 Entity 类的实例中，定义该 Entity 实例的媒体类型，即客户端请求实体的媒体类型。Request() 方法用于定义可接收的 HTTP 方法的返回媒体类型，即服务器的响应实体的媒体类型。

测试资源方法 newBook()，将得到如下所示的请求头信息，从中可以看到请求媒体类型。

```
public final static String TEXT_PLAIN = "text/plain";
public final static String APPLICATION_XML = "application/xml";
public final static MediaType TEXT_PLAIN_TYPE = new MediaType("text", "plain");
public final static MediaType APPLICATION_XML_TYPE = new MediaType("application", "xml");
1 > PUT http://localhost:9998/book
1 > Accept: text/plain
1 > Content-Type: application/xml
```

在这段代码中，javax.ws.rs.core.MediaType 类是 JAX-RS 2.0 提供的媒体类型定义类，其中定义了包括示例中使用的 MediaType.TEXT_PLAIN，其值为“text/plain”。在 MediaType 类中，对应的响应实体媒体类型定义为 Accept: text/plain；MediaType.APPLICATION_XML 值为“application/xml”，对应的请求实体媒体类型定义为 Content-Type: application/xml。

3.1.3 DELETE 方法

DELETE 方法是幂等的，即多次删除同一份数据（通常请求中传递的参数是数据的主键值），在服务器端产生的改变是相同的。JAX-RS 2.0 定义了 @DELETE 注解来定义相关资源方法。下面来看看具体示例。

执行删除的资源方法，其返回值可以定义为 void，即该方法没有返回值。之所以在删除资源的场景中可以采用这样的方式定义，是因为删除的前提是对该资源信息已经充分了解，没有必要再将其从服务器上传递回来。示例代码如下。

```
@Path("book")
public interface BookResource {
    @DELETE
    public void delete(@QueryParam("bookId") final long bookId);
}
```

在这段代码中，无返回值的资源方法 delete() 返回的响应实体为空，HTTP 状态码为 204。该定义可以参考 Jersey 的源代码中的 Response 类，示例代码如下。

```
package javax.ws.rs.core;
public abstract class Response {
    public interface StatusType {
    public enum Status implements StatusType {
        NO_CONTENT(204, "No Content"),
    ...
    }
```

接下来是删除资源方法的单元测试，示例代码如下。

```
public class DeleteTest extends JerseyTest {
    @Test
    public void testGet() {
        final Response response =
        target("book").queryParam("bookId", "9527").request().delete();
        int status = response.getStatus();
        LOGGER.debug(status);
        Assert.assertEquals(Response.Status.NO_CONTENT.getStatusCode(), status);
    }
}
```

在这段代码中，对 REST 请求的测试断言不是针对删除资源的实体，而是响应中 HTTP 状态码。也就是说，删除资源方法的返回值类型可以定义为 void，业务逻辑更关注删除操作的结果状态。

3.1.4 POST 方法

POST 方法是一种写操作的 HTTP 请求。RPC 的所有写操作均使用 POST 方法，而 REST 只使用 HTTP 的 POST 方法添加资源。

(1) 既不幂等也不安全

定义为 POST 的 REST 接口用于写数据。POST 方法的特性是既不幂等也不安全。因为请求会改变服务器端资源的状态，因此不是安全的；每次请求对服务器端资源状态的改变并不是相同的，因此不是幂等的。

(2) 两种分类

REST 中使用的 POST 可以称为 POST(a)，即用于创建、添加资源的 HTTP 方法。这是相对于 RPC 式的 Web 服务中对 POST 的使用而言的。

在 RPC 中使用的 POST 可以称为 POST（p），即通过重载的 POST 用于处理某种操作。服务器接收 POST（p）的请求后，不是直接处理 POST 请求，真正的方法信息位于信封头或实体主体里，因此需要先解析出执行方法。

JAX-RS 2.0 定义了 @POST 注解来定义相关资源方法，示例代码如下。

```
@Path("book")
public interface BookResource {
// 关注点 1: POST 方法
    @POST
    @Produces(MediaType.APPLICATION_XML)
    @Consumes(MediaType.APPLICATION_XML)
    public Book createBook(Book book);

public class PostTest extends JerseyTest {
    @Test
    public void testCreate() {
        final Book newBook = new Book("book-" + System.nanoTime());
        MediaType contentTypeMediaType = MediaType.APPLICATION_XML_TYPE;
        MediaType acceptMediaType = MediaType.APPLICATION_XML_TYPE;
        final Entity<Book> bookEntity = Entity.entity(newBook, contentTypeMediaType);
        final Book book =
        target("book").request(acceptMediaType).post(bookEntity, Book.class);
        // 关注点 2: 测试 POST 方法的断言
        Assert.assertNotNull(book.getBookId());
        LOGGER.debug("Server Id="+book.getBookId());
    }
```

在这段代码中，资源接口 BookResource 定义了 createBook() 方法，该方法使用 @POST 注解表示该方法处理“book”路径下的 POST 请求，见关注点 1。在测试方法 testCreate() 中，关注请求结果实体的主键是否为空。这是因为在 POST 请求提交的添加资源操作中，主

键的设置是在服务器端完成的，因此客户端成功请求添加资源后，应关注服务器端返回的实体结果是否有主键信息，见关注点 2。

到此，我们完成了对 HTTP 的基本方法的讲述。除了 HTTP 定义的标准方法，还存在来自其他协议中的 HTTP 方法。接下来，我们一起探讨这些方法对 REST 服务的影响。

3.1.5 WebDAV 扩展方法

WebDAV（Web-based Distributed Authoring and Versioning，基于 Web 的分布式创作与版本控制）是 IETF 的 RFC 4918 规范（RFC 2518 规范的替代规范，地址在 http://tools.ietf.org/html/rfc4918），是对 HTTP 1.1 的一组扩展，该协议允许用户以协作方式编辑和管理远程 Web 服务器上的文件。WebDAV 在 HTTP 方法的基础上，增加了如下方法。

- PROPFIND 方法：用于从 Web 资源中查询存储为 XML 格式的属性数据，或者重载为从一个远程系统中查询目录结构的数据。
- PROPPATCH 方法：用于原子地更改和删除一个资源的多个属性。
- MKCOL 方法：用于创建目录。
- COPY 方法：用于将资源从一个 URI 资源地址复制到另一个 URI 资源地址。
- MOVE 方法：用于将资源从一个 URI 资源地址移动到另一个 URI 资源地址。
- LOCK 方法：用于锁定一个资源。WebDAV 支持共享锁和独占锁。
- UNLOCK 方法：用于解锁一个资源。

虽然 WebDAV 对 HTTP 方法做出了功能性的扩展，使之提供更强大的服务，但是从 ROA 角度讲，WebDAV 破坏了统一接口的原则，因为 WebDAV 在 HTTP 标准方法的基础上增加了特殊的方法名称。因此，对是否应该在 REST 式的 Web 服务中支持 WebDAV，业内的观点并不一致。

笔者的观点是如果遵从 ROA，那么就不使用 HTTP 标准方法之外的方法。如果业务需求确实超出了标准方法所及，那么可以使用如下注解实现对 WebDAV 的支持。

JAX-RS 2.0 规范没有对 WebDAV 提供支持的描述，但是 JAX-RS 2.0 定义了 @HttpMethod 注解来定义相关的资源方法。在 Jersey 应用中，可以使用 @HttpMethod 注解定义 HTTP 标准方法之外的方法名称来支持 WebDAV，示例代码如下。

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(value = "MOVE")
@Documented
public @interface MOVE {
}
```

这段代码是 @MOVE 注解的定义，使用 @HttpMethod 注解定义了名为 MOVE 的 HTTP 扩展方法。有了扩展方法注解，我们就可以在资源类中定义新的方法来支持扩展方法的请求了。示例代码如下。

```
@Path("book")
public interface BookResource {
    @MOVE
    public boolean moveBooks(Books books);
}
```

在这段代码中，资源类 BookResource 定义了 moveBooks() 方法，该方法使用 @MOVE 注解定义，表示用于处理“book”路径下的 MOVE 请求。下面我们来看看相关的测试代码。

```
public class HttpMethodTest extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig resourceConfig = new ResourceConfig(EBookResourceImpl.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig clientConfig) {
    //关注点 1: 定义 Grizzly 连接器
   clientConfig.connectorProvider(new GrizzlyConnectorProvider());
   super.configureClient(clientConfig);
    }
    @Test
    public void testWebDav() {
//关注点 2: HTTP MOVE 请求
        final Response response = target("book").request().method("MOVE");
        Boolean result = response.readEntity(Boolean.class);
//关注点 3: Move 方法测试断言
        Assert.assertEquals(Boolean.TRUE, result);
    }
}
```

在这段代码中，测试方法 testWebDav() 在请求中定义了 MOVE 请求，见关注点 2。断言是针对 MOVE 方法的返回值，见关注点 3。可以看出，使用 Jersey 实现对 WebDav 的支持并不困难。

需要注意的是，Jersey 默认的连接器只支持 HTTP 标准方法，因此要使用 HTTP 的扩展方法就不能直接使用默认的连接器，这里使用了 Grizzly 连接器。对应的代码行是：clientConfig.connectorProvider(new GrizzlyConnectorProvider())，即为客户端配置实例提供 Grizzly 连接器，见关注点 1。这行代码是 Jersey 2.5+ 后的写法，Jersey 2.5 之前的写法是 clientConfig.connector(new GrizzlyConnector(clientConfig))。从中可以看出，Jersey 在不断优化，包括 API。这一好处是活跃的社区为用户带来越来越便捷、高效的使用体验，缺点是破坏了向下兼容性。

到此，我们全面掌握了 HTTP 方法在 REST 统一接口定义中的作用和实现。明白了 REST 接口该使用什么样的请求方法非常重要，这决定了其性质。但是这还不够，一个接口如何被请求唯一定位还需要深入掌握 REST 的资源定位。接下来一节将详述资源定位的细节。

3.2 REST 资源定位

REST 使用 URI 实现资源定位，从这个角度上讲，对外提供 REST 式的 Web 服务的接口就是公布一系列的 URI 及其参数，这使得 REST 的实践过程简单到了极致。但是 URI 形式上的简单并不意味着我们可以将 URI 的定义随意使用，正所谓“不以规矩，不成方圆”。

在设计 REST 式的 Web 服务过程中，资源地址的设计是非常严谨的，如果设计不好，不仅 REST 接口的风格无法统一，使系统的扩展性和易用性降低，也很难实现资源准确地被定位。

资源地址的设计过程是面向资源的，资源名称应是准确描述该资源的名词，资源地址应具有直观的描述性。比如一个班级的资源地址可以是：学校 / 学院 / 学级 / 班级。值得注意的是，一个 URI 资源地址唯一对应一个资源，但是一个资源可以拥有多个 URI 资源地址。比如，Jersey 最新版本的文档地址和 Jersey 2.9 版本的文档地址指向同一个资源（笔者撰稿时）。

阅读指南

3.2 节示例所在目录是 jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.annotation.param。

3.2.1 资源地址设计

资源地址的设计对整个 REST 式的 Web 服务至关重要，涉及系统的可用性、可维护性和可扩展性等诸多方面的表现。因此，本节关注如何对资源地址进行设计。

1. 资源路径概览

资源地址的路径变量是用来表达逻辑上的层次结构的，资源和子资源的形式是自左至右、斜杠分割的名词。它们的关系可以是从整体到局部，比如学校到班级，洲际到乡镇；也可以是从一般到具体，比如一个生物的“门、纲、目、科、属、种”的资源路径。资源地址具体可以分为 5 个部分，见表 3-1。

表 3-1 资源地址路径分解

scheme://host:port/path?queryString	
scheme	协议名称。通常是 HTTP 和 HTTPS
host	（DNS）主机名称或者 IP 地址
port	端口
path	资源地址，使用“/”符号来分隔逻辑上的层次结构
?	用来分隔资源地址和查询字符串符号
queryString	查询字符串，方法作用域信息 使用“&”符号来分隔查询条件 使用逗号分隔有次序的作用域信息 使用分号分隔无次序的作用域信息

一个典型的 URI 如表 3-1 所示，包括协议名称、主机名称、服务端口、资源地址和查询字符串等 5 个部分。其中资源地址部分，根据具体部署的不同或有差别，如图 3-2 所示。

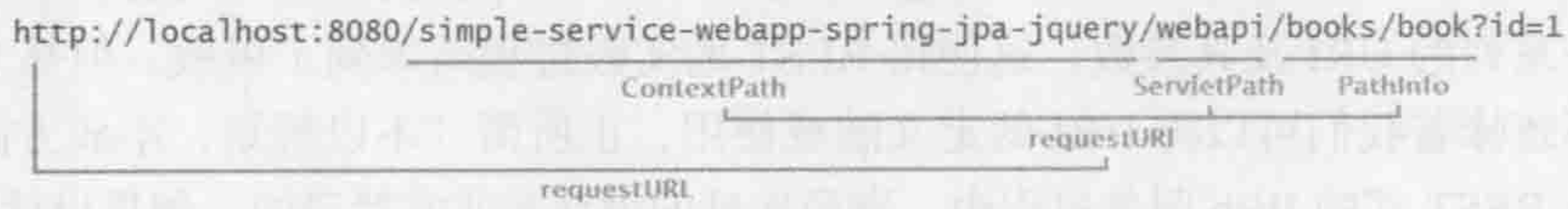


图 3-2 资源地址示例

如图 3-2 所示，通常使用 ContextPath、ServletPath 和 PathInfo 来细分资源地址。ContextPath 是上下文名称，通常和部署服务器的配置或者 REST 服务的 web.xml 配置有关。ServletPath 是 Servlet 的名称，与 REST 服务中定义的 @ApplicationPath 注解或者 web.xml 的配置有关。JAX-RS 2.0 定义了 @Path 注解来定义资源地址。PathInfo 是资源路径信息，与资源类、子类以及类中的方法中定义的 @Path 注解有关。

现在我们对资源地址的层次结构有了初步了解，此时需要思考一个问题：资源地址是否可以唯一定位一个资源？

答案是否定的。资源地址相同，但 HTTP 方法不同的两个方法是两个不同的 REST 接口。HTTP 方法和资源地址结合在一起才可以完成对资源的定位。细心的读者也许从 3.1 节的示例中已经看出端倪。示例中，GET 方法用于读取 / 检索、查询 / 过滤一个资源，PUT 方法用于修改 / 更新资源、创建客户端维护主键信息的资源，DELETE 方法用于删除资源，POST 方法用于创建资源。但这些方法的资源地址是相同的，都是“book”。

当上述的标准的 HTTP 方法无法满足业务需求时，比如对于图书资源，除了基本的 CRUD 之外，若需要公布像借阅、折旧、电子版下载等实际生活中的更新操作的接口时，仅公布一个 PUT 方法就不够用了。这些操作是动词性的，无法简单地使用一个 book 名词就能定位。在路径变量难以准确描述的情况下，一种方案是可以考虑使用动词作为查询参数；另一种方案是可以在 REST 设计过程中引入 RPC 风格的 POST 方法，辅助完成复杂业务的接口设计，这就是 REST 和 RPC 混合型的 Web 服务了。

2. 资源地址和作用域

在路径变量里可以使用标点符号以辅助增强逻辑清晰性。这些辅助符号用在表 3-2 中的查询字符串，作为资源地址的查询变量，用来表达算法的输入，实现对方法的作用域的约束。下面来逐一讲述这些对资源地址设计至关重要的符号。

1）“问号”（?）是用来分隔资源地址和查询字符串的，“与”（&）符号是用来分隔查询条件的参数的。示例代码如下。

```
GET /books?start=0&size=10
```

这行代码的作用是查询图书列表，开始行参数为 0，条目参数为 10，即从第 0 行开始取 10 条信息并返回该图书列表。

2）“逗号”（,）是用来分隔有次序的作用域信息。需要注意的是，逗号分隔逻辑上的顺

序信息，这种顺序可以是约定俗成的，比如先写经度后写纬度；也可以是系统约定的，比如“月日年”的顺序等。举例来说，按时间区间查询图书，日期信息在资源地址中是采用“月年”顺序，示例如下。

```
GET /books/01,2002-12,2014
```

这行代码的作用是查询 2002 年 1 月 ~ 2014 年 12 月这个时间段（出版）的图书。这个例子中还使用了“连字符”(-)，有时候也可以使用“下划线”(_) 来做逻辑上的辅助分隔。

3）“分号”(;) 是用来分隔无次序的作用域信息。通常这些信息是逻辑上并列存在的，比如并列的查询条件，示例如下所示。

```
GET /books/restful;program=java;type=web
```

这行代码的作用是查询满足图书内容为 RESTful 的、使用的编程语言是 Java 的、讲述的类型是 Web 的图书列表。这样的逻辑没有顺序，互换顺序的查询条件不会影响资源的表述。

基于上述理论，列出常用的资源地址设计示例，见表 3-2。

表 3-2 资源地址设计

功能	资源地址
添加 / 创建	POST /books PUT /books/{id}
删除	DELETE /books/{id}
修改 / 更新	PUT /books/{id}
查询全部	GET /books HTTP1.1
主键查询	GET /books/{id} HTTP1.1 GET /books?id=12345678
分页作用域查询	GET /books?start=0&size=10 GET /books/01,2002-12,2014 GET /books/restful;program=java;type=web GET /books?limit=100&sort=bookname

如果读者可以轻松领会表 3-2 列出的这些典型的 REST 接口和资源定位的设计，就可以放手去做实现的事情了，否则建议回顾本节内容。接下来，我们完成从设计到实现的跨越，看看 JAX-RS 2.0 标准是如何通过注解来支持资源定位的，并使用 Jersey 完成上述设计的实践。

3.2.2 @QueryParam 注解

查询条件决定了方法的作用域，查询参数组成了查询条件。JAX-RS 2.0 定义了 @QueryParam 注解来定义查询参数，本节使用 @QueryParam 演示 3 个 REST 查询接口的实现示例，见表 3-3。

表 3-3 @QueryParam 示例列表

接口描述	资源地址
分页查询列表数据	/query-resource/yijings?start=24&size=10

（续）

接口描述	资源地址
排序并分页查询列表数据	/query-resource/sorted-yijings?limit=5&sort=pronounce
查询单项数据	/query-resource/yijing?id=8

（1）分页查询

分页查询是使用 @QueryParam 解析参数的基本示例。实现代码如下所示。

```
public Yijings getByPaging(@QueryParam("start") final int start,
@QueryParam("size") final int size) {// 关注点 1: 资源方法入参
...
  int listSize = globalList.size();
  final int max = size > listSize ? listSize : size;
  // 关注点 2: 分页迭代逻辑
  for(int i = 0, index = start; i < max; i++) {
      final Yijing yijing = globalList.get(index + i);
      // 关注点 3: 添加 Link 以保证 REST 的连通性
      final URI location = ub.clone().queryParam("id", yijing.getSequence()).build();
      final Link link =
      new Link("detail", location.toASCIIString(), MediaType.APPLICATION_XML);
      links.add(link);
      yijings.add(yijing);
  }
  result.setLinks(links);
  result.setGuas(yijings);
  return result;
}
```

在这段代码中，getByPaging() 方法的输入参数包含了两个使用 @QueryParam 注解定义的查询参数，分别是起始条目参数“start”和条目数量参数“size”，参数的类型是整型，见关注点 1。在查询的迭代中使用这两个参数获取图书列表，见关注点 2。在迭代中，每个图书资源条目的 URI 都存储在返回值中，以保证资源的联通性，见关注点 3。该 URI 被封装到 Link 实例中，在单项查询时使用。

另外，参数的定义使用了 final，符合 Checkstyle 的编程风格，即输入参数只作为逻辑算法的依据使用，其本身不会在这个过程中被修改。也许这种不变的变量对提高执行效率并没有多少影响，但长久来看效果还是不错的。推荐 Java 开发者在 REST 开发中引入 SonarQube 平台或者单纯使用 Checkstyle 工具对静态代码进行质量检测，以帮助我们改进代码的质量。

（2）排序查询

排序查询是在解析参数的基础上，额外处理结果集顺序的示例，代码如下。

```
public Yijings getByOrder(@QueryParam("limit") final int limit,
@QueryParam("sort") final String sortName) {// 关注点 1: 资源方法入参
...
Collections.sort(list, new Comparator<Yijing>() {
  @Override
  // 关注点 2: 排序中的比较算法
  public int compare(final Yijing o1, final Yijing o2) {
```

```
        switch (sortName) {
            case "sequence":
                return o1.getSequence().compareTo(o2.getSequence());
            case "name":
                return o1.getName().compareTo(o2.getName());
            case "pronounce":
                return o1.getPronounce().compareTo(o2.getPronounce());
        }
        return 0;
    }
});
```

在这段代码中，limit 参数的用途同分页查询示例，而 sortName 参数则用于排序，见关注点 1。排序接口需要额外解析 sortName 传递的排序字段，并将其作为数据库查询语句中的排序参数使用。这里实现了 Comparator 接口的 compare() 方法来完成根据不同字段对集合的排序，见关注点 2。

(3) 单项查询

客户端在获得结果集的基础上，根据表述中链接信息，向服务器发起单项查询的示例，代码如下所示。

```
public Yijing getByQuery(@QueryParam("id") final int seqId) {
  return ParamCache.find("" + seqId);
}
```

在这段代码中，使用 @QueryParam 定义了“id”参数，该参数来自分页查询中返回的 URI 信息。

3.2.3 @PathParam 注解

JAX-RS 2.0 定义了 @PathParam 注解来定义路径参数——每个参数对应一个子资源。本节使用 @PathParam 完成表 3-4 所示的 REST 查询接口。

表 3-4 @PathParam 示例列表

接口描述	资源地址
基本路径参数	/path-resource/Eric
结合查询参数	/path-resource/Eric?hometown=Buenos Aires
带有标点符号的资源路径	/path-resource/199-1999
	/path-resource/01,2012-12,2014
子资源变长的资源路径	/path-resource/Asia/China/northeast/liaoning/shenyang/huanggu
	/path-resource/q/restful;program=java;type=web /path-resource/q2/restful;program=java;type=web

1. @Path 注解

JAX-RS 2.0 定义了 @Path 注解来定义资源路径，@Path 接收一个 value 参数，来解析资源路径地址。该参数除了前面示例中的 books 这种静态定义的方式外，也可以使用动态

变量的方式，其的格式为：{参数名称：正则表达式}。这个接口的功能和查询参数实现的 /query-resource/yijings?start=24&size=10 相似，也是用于分页查询，其资源地址形如：/path-resource/199-1999。参考示例如下。

```
@GET
@Path ("{from:\\d+}-{to:\\d+}")
public String getByCondition (@PathParam ("from") final Integer from,
@PathParam ("to") final Integer to) {
...
}
```

在这段代码中，使用 @PathParam 注解定义的两个参数 from 和 to 用以定义查询区间，正则表达式部分是 \\d+，表示数字。两个参数中间的“连接符”(-) 是路径的格式信息。

稍显复杂的例子是：/path-resource/01,2012-12,2014，引入了逗号（,）作为有顺序的日期分隔符号，那么对应的正则表达式为：

```
@Path ("{beginMonth:\\d+},{beginYear:\\d+}-{endMonth:\\d+},{endYear:\\d+}")
```

2. 正则表达式

正则表达式的内容超出了本书的范围，这里只简述示例中用到的正则表达式。刚才的例子中的 \\d+，代表参数应为数字并且至少出现一次。第一个反斜杠是 Java 中的转义字符，第二个反斜杠是正则表达式的起始，加号（+）是至少出现一次的意思，星号（*）则代表出现至少零次，句号（.）是匹配任何字符，d 是匹配数字，w 是匹配数字和字母。

示例中使用的正则表达式见表 3-5 所示，读者掌握所说明的路径含义即可，我们的目的是学习 REST API 设计，而非正则表达式本身。

表 3-5 正则表达式示例

正则表达式	含义
[a-zA-Z][a-zA-Z_0-9]*	以字母开头，后面是零到多个“字母 _ 数字”格式的字符组合
{region:.+}/{district:\\w+}	region 变量至少包含一个任意字符 district 变量至少包含一个为数字或者字母的字符

3. 路径配合查询

查询参数和路径参数在一个接口中配合使用，可以更快捷地完成资源定位，这很像战场上的多兵种协同作战。前述的图书资源的复杂设计就需要两者结合来完成，示例代码如下。

```
@Path("{user: [a-zA-Z][a-zA-Z_0-9]*}")
@Produces(MediaType.TEXT_PLAIN)
public String getUserInfo(@PathParam("user") final String user,
@DefaultValue("Shen Yang")@QueryParam("hometown") final String hometown) {
```

```
    return user + ":" + hometown;
  }
```

在这段代码中，路径参数 user 中使用了通配符，方法参数中同时使用 @PathParam 注解和 @QueryParam，定义了 user 和 hometown 两个参数。以资源地址：/path-resource/Eric?hometown=Buenos Aires 为例，REST 容器会将该请求匹配到 getUserInfo() 方法，其中 Eric 是路径变量 user 的值，Buenos Aires 作为查询变量 hometown 的值。

4. 路径区间

路径区间（PathSegment）是对资源地址更灵活的支持，使资源类的一个方法可以支持更广泛的资源地址的请求。我们以下面定义的资源地址列表来了解 PathSegment。

```
/path-resource/Asia/China/northeast/liaoning/shenyang/huanggu
/path-resource/China/northeast/liaoning/shenyang/tiexi
/path-resource/China/shenyang/huanggu
```

如上所示的资源地址中含有固定子资源（shenyang）和动态子资源两部分。对于动态匹配变长的子资源资源地址，PathSegment 类型的参数结合正则表达式将发挥很大的作用。示例代码如下。

```
@GET
@Path("{region:.+}/shenyang/{district:\\w+}")
public String getByAddress(@PathParam("region") final List<PathSegment> region,
@PathParam("district") final String district) {
  final StringBuilder result = new StringBuilder();
  for (final PathSegment pathSegment : region) {
    result.append(pathSegment.getPath()).append("-");
  }
  result.append("shenyang-" + district);
...
}
```

在这段代码中，getByAddress() 方法用来匹配表的这些资源地址。该方法的 region 变量是 PathSegment 类型的数组，以匹配至少出现一个字符的正则表达式（.+）。PathSegment 如其名字所示，是路径的片段，是子资源的集合。遍历 PathSegment 集合，对于每一个 PathSegment 实例，可以通过调用其 getPath 方法获取子资源名称。

对于查询参数动态给定的场景，可以定义 PathSegment 作为参数类型，通过 getMatrixParameters() 方法获取 MultivaluedMap 类型的查询参数信息，即可将参数条件作为一个整体解析。示例代码如下。

```
@Path("q/{condition}")
public String getByCondition3(@PathParam("condition") final PathSegment condition) {
...
  final MultivaluedMap<String, String> matrixParameters = condition.getMatrixParameters();
  final Iterator<Entry<String, List<String>>>
iterator = matrixParameters.entrySet().iterator();
```

```
    while (iterator.hasNext()) {
        final Entry<String, List<String>> entry = iterator.next();
        conds.append(entry.getKey()).append("=");
        conds.append(entry.getValue()).append(" ");
    }
    return conds.toString();
}
```

在这段代码中，getByCondition3() 方法只有一个 PathSegment 类型的参数 condition，该参数包含了查询条件中携带的全部参数列表。举例来说，资源地址为“path-resource/q/restful;program=java;type=web”的请求可以匹配 getByCondition3() 方法，其中，Multivalued Map 类型的实例 matrixParameters 的值为“[program=[java], type=[web]]”。

5. @MatrixParam 注解

上例中，通过编程方式，调用 PathSegment 类的 getMatrixParameters() 方法来获取查询参数信息。还有一种方式是通过 @MatrixParam 注解来逐一定义参数，即通过声明方式来获取。示例代码如下。

```
@Path("q2/{condition}")
public String getByCondition4(@PathParam("condition")
final PathSegment condition, @MatrixParam("program") final String program,
  @MatrixParam("type") final String type) {
  return condition.getPath() + " program=[" + program + "] type=[" + type + "]";
}
```

在这段代码中，使用 @MatrixParam 注解分别定义了“program”和“type”两个参数。与上例相比，这段代码更加清晰地表达了可接收的参数名称和类型，缺点是缺乏对请求资源地址更灵活的支持。

3.2.4 @FormParam 注解

JAX-RS 2.0 定义了 @FormParam 注解来定义表单参数，相应的 REST 方法用以处理请求实体媒体类型为 Content-Type: application/x-www-form-urlencoded 的请求。示例代码如下。

```
@Path("form-resource")
public class FormResource {
@POST
public String newPassword(
        @DefaultValue("feuyeux") @FormParam(FormResource.USER) final String user,
        @Encoded @FormParam(FormResource.PW) final String password,
        @Encoded @FormParam(FormResource.NPW) final String newPassword,
        @FormParam(FormResource.VNPW) final String verification) {
...
}
```

在这段代码的 newPassword() 方法中，@FormParam 注解定义了 user 等 4 个参数，这些参数是容器从请求中获取并匹配的。相关的客户端测试如图 3-3 所示。

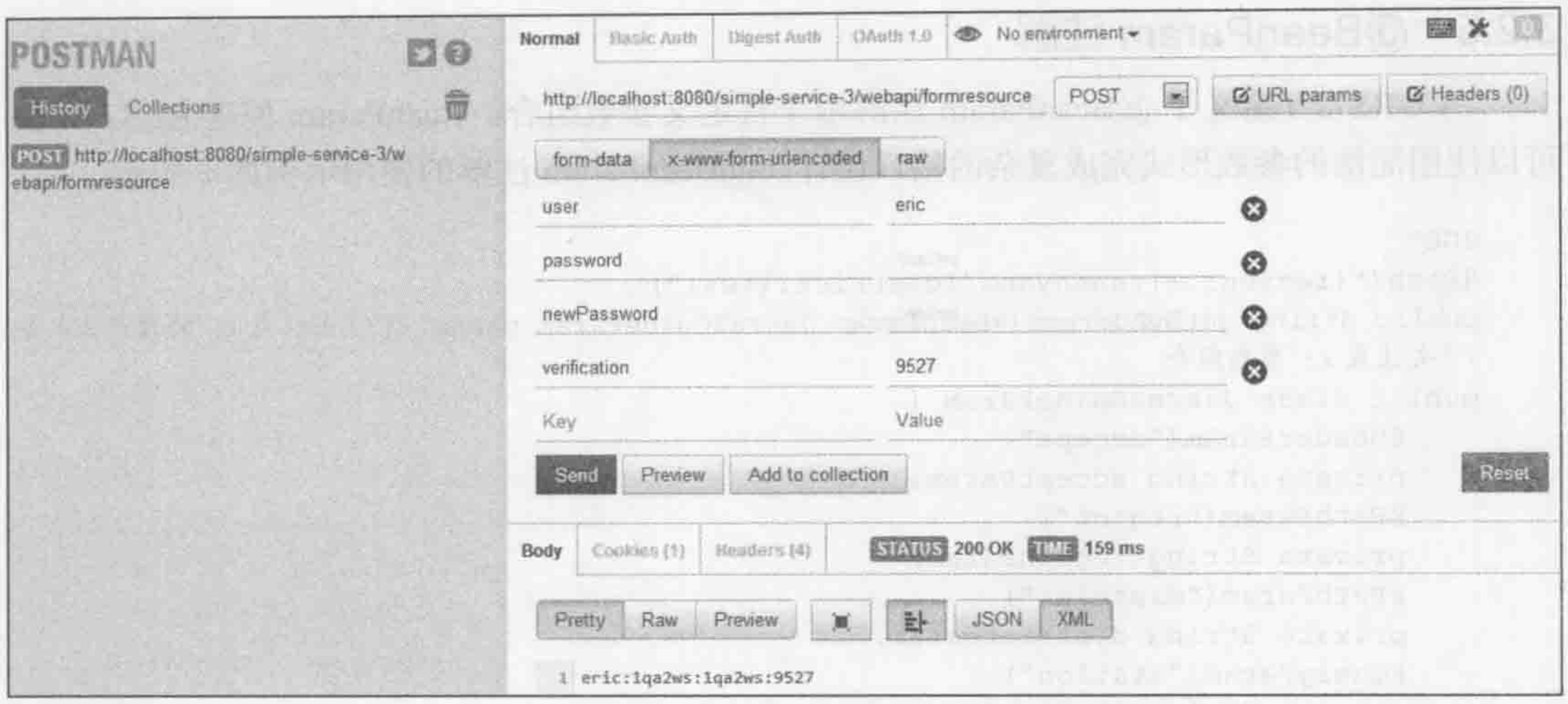


图 3-3 表单示例

图 3-3 所示的客户端工具是 Postman（详见 2.6 节），使用 Posman 定义的基本表单信息与 newPassword() 方法一致。

newPassword() 方法的测试代码片段如下。

```
@Test
public void testPost2() {
  final Form form = new Form();
  form.param(FormResource.USER, "feuyeux");
  form.param(FormResource.PW, "北京");
  form.param(FormResource.NPW, "上海");
  form.param(FormResource.VNPW, "上海");
  final String result = target("form-resource").request().
  post(Entity.entity(form, MediaType.APPLICATION_FORM_URLENCODED_TYPE), String.class);
  FormTest.LOGGER.debug(result);
  Assert.assertEquals("encoded should let it to disable decoding",
"feuyeux:%E5%8C%97%E4%BA%AC:%E4%B8%8A%E6%B5%B7:上海", result);
}
```

在这段代码中，Form 类实例是请求实体，请求实体的类型为 MediaType.APPLICATION_FORM_URLENCODED_TYPE，即 application/x-www-form-urlencoded。

这里还需要注意的是 @Encoded 注解和 @DefaultValue 注解的使用。

JAX-RS 2.0 定义了 @Encoded 注解用以标识禁用自动解码。示例的测试结果中“%E4%B8%8A%E6%B5%B7”是 newPassword() 的参数值“上海”的编码值，当对 newPassword() 使用 @Encoded 注解时，REST 方法得到的参数值就不会被解码，如果将其直接返回，那么客户端得到的值就会是处于编码状态的字符串。

JAX-RS 2.0 定义了 @DefaultValue 注解用以为客户端没有为其提供值的参数提供默认值。本例的 user 参数的默认值为 feuyeux。

3.2.5 @BeanParam 注解

JAX-RS 2.0 定义了 @BeanParam 注解用于自定义参数组合，BeanParam 使得 REST 方法可以使用简洁的参数形式完成复杂的接口设计。@BeanParam 注解的使用示例如下所示。

```
@GET
@Path("{region:.+}/shenyang/{district:\\w+}")
public String getByAddress(@BeanParam Jaxrs2GuideParam param) {// 关注点 1: 资源方法入参
// 关注点 2: 参数组合
public class Jaxrs2GuideParam {
    @HeaderParam("accept")
    private String acceptParam;
    @PathParam("region")
    private String regionParam;
    @PathParam("district")
    private String districtParam;
    @QueryParam("station")
    private String stationParam;
    @QueryParam("vehicle")
    private String vehicleParam;
public void testBeanParam() {
...
  final WebTarget queryTarget = target(path).path("China").path("northeast")
  .path("shenyang").path("tiexi")
  .queryParam("station", "Workers Village").queryParam("vehicle", "bus");
  result = queryTarget.request().get().readEntity(String.class);
  // 关注点 3: 查询结果断言
  Assert.assertEquals("China/northeast:tiexi:Workers Village:bus", result);
}
// 关注点 4: 复杂的查询请求
http://localhost:9998/ctx-resource/China/shenyang/tiexi?station=Workers+Village&vehicle=bus
```

在这段代码中，getByAddress() 方法只有了一个使用 @BeanParam 注解定义的 Jaxrs2-GuideParam 类型的参数，见关注点 1。Jaxrs2GuideParam 类定义了一系列 REST 方法会用到的参数类型，包括示例中使用的查询参数“station”和路径参数“region”等。从而使得 getByAddress() 方法可以匹配更为复杂的资源路径，见关注点 2。在变长子资源的例子基础上，增加了查询条件，但从测试方法 testBeanParam() 发起的请求的资源地址 (见关注点 4) 可以看出，这是一个较为复杂的查询请求，其中路径部分包括 China/shenyang/tiexi，查询条件包括 station=Workers+Village 和 vehicle=bus。这些条件均在 Jaxrs2GuideParam 类中可以匹配，因此从关注点 3 的测试断言中可以看出，该请求的响应的预期结果是“China/northeast:tiexi:Workers Village:bus”。

3.2.6 @CookieParam 注解

JAX-RS 2.0 定义了 @CookieParam 注解用以匹配 Cookie 中的键值对信息，示例代码如下。

```
@GET
public String getHeaderParams(@CookieParam("longitude") final String longitude,
```

```
        @CookieParam("latitude") final String latitude,
        @CookieParam("population") final double population,
        @CookieParam("area") final int area) {// 关注点 1: 资源方法入参
        return longitude + "," + latitude + " population=" + population + ",area=" + area;
@Test
public void testContexts() {
  final Builder request = target(path).request();
  request.cookie("longitude", "123.38");
  request.cookie("latitude", "41.8");
  request.cookie("population", "822.8");
  request.cookie("area", "12948");
  result = request.get().readEntity(String.class);
  // 关注点 2: 测试结果断言
  Assert.assertEquals("123.38,41.8 population=822.8,area=12948", result);
}
```

在这段代码中，getHeaderParams() 方法包含 4 个使用 @CookieParam 注解定义的参数，用于匹配 Cookie 的字段，见关注点 1。在测试方法 testContexts() 中，客户端 Builder 实例填充了相应的 cookie 键值对信息，其断言是对 cookie 字段值的验证，见关注点 2。

3.2.7 @Context 注解

JAX-RS 2.0 定义了 @Context 注解来解析上下文参数，JAX-RS 2.0 中有多种元素可以通过 @Context 注解作为上下文参数使用。示例代码如下。

```
public String getByAddress(
        @Context final Application application,
        @Context final Request request,
        @Context final javax.ws.rs.ext.Providers provider,
        @Context final UriInfo uriInfo,
        @Context final HttpHeaders headers){
...
}
```

在这段代码中，分别定义了 Application、Request、Providers、UriInfo 和 HttpHeaders 等 5 种类型的上下文实例。从这些实例中可以获取请求过程中的重要参数信息，示例代码如下。

```
final MultivaluedMap<String, String> pathMap = uriInfo.getPathParameters();
final MultivaluedMap<String, String> queryMap = uriInfo.getQueryParameters();
final List<PathSegment> segmentList = uriInfo.getPathSegments();
final MultivaluedMap<String, String> headerMap = headers.getRequestHeaders();
```

在这段代码中，UriInfo 类是路径信息的上下文，从中可以获取路径参数集合 getPathParameters() 和查询参数集合 getQueryParameters()。类似的，我们可以从 HttpHeaders 类中获取头信息集合 getRequestHeaders()。这些业务逻辑处理中常用的辅助信息的获取，通过 @Context 注解定义方法的参数或者类的字段来实现。

到此，我们已经对统一接口和资源定位的设计和实现讲述完毕。但是，设计 REST 接口还需要在此基础上，掌握请求实体和响应实体的传输格式。接下来，让我们看看 Jersey 都支持哪些类型的传输格式。

3.3 REST 传输格式

本节要考虑的就是如何设计表述，即传输过程中数据采用什么样的数据格式。通常，REST 接口会以 XML 和 JSON 作为主要的传输格式，这两种格式数据的处理是本节的重点。那么 Jersey 是否还支持其他的数据格式呢？答案是肯定的，让我们逐一掌握各种类型的实现。

3.3.1 基本类型

Java 的基本类型又叫原生类型，包括 4 种整型（byte、short、int、long）、两种浮点类型（float、double）、Unicode 编码的字符（char）和布尔类型（boolean）。

阅读指南

3.3 节的前 4 小节示例所在目录是：jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.response。

Jersey 支持全部的基本类型，还支持与之相关的引用类型。前述示例已经呈现了整型（int）等 Java 的基本类型的参数，本例将展示字节数组类型作为请求实体类型、字符串作为响应实体类型的示例，代码如下。

```
@POST
@Path("b")
public String postBytes(final byte[] bs) {// 关注点 1: 测试方法入参
  for (final byte b : bs) {
      LOGGER.debug(b);
  }
  return "byte[]:" + new String(bs);
}
@Test
public void testBytes() {
  final String message = "TEST STRING";
  final Builder request = target(path).path("b").request();
  final Response response = request.post(
  Entity.entity(message, MediaType.TEXT_PLAIN_TYPE), Response.class);
  result = response.readEntity(String.class);
  // 关注点 2: 测试断言
  Assert.assertEquals("byte[]:" + message, result);
}
```

在这段代码中，资源方法 postBytes() 的输入参数是 byte[] 类型，输出参数是 String 类型，见关注点 1。单元测试方法 testBytes() 的断言是对字符串“TEST STRING”的验证，见关注点 2。

3.3.2 文件类型

Jersey 支持传输 File 类型的数据，以方便客户端直接传递 File 类实例给服务器端。文件

类型的请求，默认使用的媒体类型是 Content-Type: text/html。示例代码如下。

```
@POST
@Path("f")
//关注点 1: 测试方法入参
public File postFile(final File f) throws FileNotFoundException, IOException {
//关注点 2: try-with-resources
  try (BufferedReader br = new BufferedReader(new FileReader(f))) {
      String s;
      do {
          s = br.readLine();
          LOGGER.debug(s);
      } while (s != null);
      return f;
  }
}
@Test
public void testFile() throws FileNotFoundException, IOException {
//关注点 3: 获取文件全路径
  final URL resource = getClass().getClassLoader().getResource("gua.txt");
//关注点 4: 构建 File 实例
  final String file = resource.getFile();
  final File f = new File(file);
  final Builder request = target(path).path("f").request();
//关注点 5: 提交 POST 请求
  Entity<File> e = Entity.entity(f, MediaType.TEXT_PLAIN_TYPE);
  final Response response = request.post(e, Response.class);
  File result = response.readEntity(File.class);
  try (BufferedReader br = new BufferedReader(new FileReader(result))) {
      String s;
      do {
          s = br.readLine();//关注点 6: 逐行读取文件
          LOGGER.debug(s);
      } while (s != null);
  }
}
```

在这段代码中，资源方法 postFile() 的输入参数类型和返回值类型都是 File 类型，见关注点 1。服务器端对 File 实例进行解析，最后将该资源释放，即 try-with-resources，见关注点 2。在测试方法 testFile() 中，构建了 File 类型的“gua.txt”文件的实例，见关注点 3，作为请求实体提交，见关注点 4，并对响应实体进行逐行读取的校验，见关注点 5。

需要注意的是，由于我们使用的是 Maven 构建的项目，测试文件位于测试目录的 resources 目录，其相对路径为 /simple-service-3/src/test/resources/gua.txt，获取该文件的语句为 getClass().getClassLoader().getResource("gua.txt")，见关注点 6。

另外，文件的资源释放使用了 JDK 7 的 try-with-resources 语法，见关注点 2。

3.3.3 InputStream 类型

Jersey 支持 Java 的两大读写模式，即字节流和字符流。本示例展示字节流作为 REST 方

法参数，代码如下。

```
@POST
@Path("bio")
//关注点 1: 资源方法入参
public String postStream(final InputStream is) throws FileNotFoundException, IOException {
  //关注点 2: try-with-resources
  try (BufferedReader br = new BufferedReader(new InputStreamReader(is))) {
       StringBuilder result = new StringBuilder();
       String s = br.readLine();
       while (s != null) {
            result.append(s).append("\n");
            LOGGER.debug(s);
            s = br.readLine();
       }
       return result.toString();//关注点 3: 资源方法返回值
  }
}
@Test
public void testStream() {
//关注点 4: 获取文件全路径
  final InputStream resource = getClass().getClassLoader().getResourceAsStream("gua.txt");
  final Builder request = target(path).path("bio").request();
  Entity<InputStream> e = Entity.entity(resource, MediaType.TEXT_PLAIN_TYPE);
  final Response response = request.post(e, Response.class);
  result = response.readEntity(String.class);
  //关注点 5: 输出返回值内容
  LOGGER.debug(result);
}
```

在这段代码中，资源方法 postStream() 的输入参数类型是 InputStream，见关注点 1，服务器端从中读取字节流，并最终释放该资源，见关注点 2，返回值是 String 类型，内容是字节流信息，见关注点 3。测试方法 testStream() 构建了“gua.txt”文件内容的字节流，作为请求实体提交，见关注点 4。响应实体预期为 String 类型的“gua.txt”文件内容信息，见关注点 5。

3.3.4 Reader 类型

本示例展示另一种 Java 读写模式，以字符流作为 REST 方法参数，代码如下。

```
@POST
@Path("cio")
//关注点 1: 资源方法入参
public String postChars(final Reader r) throws FileNotFoundException, IOException {
  //关注点 2: try-with-resources
  try (BufferedReader br = new BufferedReader(r)) {
       String s = br.readLine();
       if (s == null) {
            throw new Jaxrs2GuideNotFoundException("NOT FOUND FROM READER");
       }
       while (s != null) {
```

```
            LOGGER.debug(s);
            s = br.readLine();
        }
        return "reader";
    }
}
@Test
public void testReader() {
    // 关注点 3: 构建并提交 Reader 实例
    ClassLoader classLoader = getClass().getClassLoader();
    final Reader resource =
    new InputStreamReader(classLoader.getResourceAsStream("gua.txt"));
    final Builder request = target(path).path("cio").request();
    Entity<Reader> e = Entity.entity(resource, MediaType.TEXT_PLAIN_TYPE);
    final Response response = request.post(e, Response.class);
    result = response.readEntity(String.class);
    // 关注点 4: 输出返回值内容
    LOGGER.debug(result);
}
```

在这段代码中，资源方法 postChars() 的输入参数类型是 Reader，见关注点 1，服务器端从中读取字符流，并最终释放该资源，见关注点 2，返回值是 String 类型。测试方法 testReader() 构建了“gua.txt”文件内容的 Reader 实例，将字符流作为请求实体提交，见关注点 3。响应实体预期为 String 类型的“gua.txt”文件内容信息，见关注点 4。

3.3.5 XML 类型

XML 类型是使用比较广泛的数据类型。Jersey 对 XML 类型的数据处理，支持 Java 领域的两大标准，即 JAXP（Java API for XML Processing，JSR-206）和 JAXB（Java Architecture for XML Binding，JSR-222）。

阅读指南

本小节示例所在目录是：jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.media.xml。

1. JAXP 标准

- JAXP 包含了 DOM、SAX 和 StAX 三种解析 XML 的技术标准。
- DOM 是面向文档解析的技术，要求将 XML 数据全部加载到内存，映射为树和结点模型以实现解析。
- SAX 是事件驱动的流解析技术，通过监听注册事件，触发回调方法以实现解析。
- StAX 是拉式流解析技术，相对于 SAX 的事件驱动推送技术，拉式解析使得读取过程可以主动推进当前 XML 位置的指针而不是被动获得解析中的 XML 数据。

对应的，JAXP 定义了三种标准类型的输入接口 Source（DOMSource、SAXSource、

StreamSource）和输出接口 Result（DOMResult、SAXResult、StreamResult）。Jersey 可以使用 JAXP 的输入类型作为 REST 方法的参数，示例代码如下。

```
@POST
@Path("stream")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public StreamSource getStreamSource(
javax.xml.transform.stream.StreamSource streamSource) {
  //关注点 1: 资源方法入参
  return streamSource;
}
@POST
@Path("sax")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
//关注点 2: 支持 SAX 技术
public SAXSource getSAXSource(javax.xml.transform.sax.SAXSource saxSource) {
  return saxSource;
}
@POST
@Path("dom")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
//关注点 3: 支持 DOM 技术
public DOMSource getDOMSource(javax.xml.transform.dom.DOMSource domSource) {
  return domSource;
}
@POST
@Path("doc")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
//关注点 4: 支持 DOM 技术
public Document getDocument(org.w3c.dom.Document document) {
  return document;
}
```

在这段代码中，资源方法 getStreamSource() 使用 StAX 拉式流解析技术支持输入 / 输出类型为 StreamSource 的请求，见关注点 1。getSAXSource() 方法使用 SAX 是事件驱动的流解析技术支持输入 / 输出类型为 SAXSource 的请求，见关注点 2。getDOMSource() 方法和 getDocument() 方法使用 DOM 面向文档解析的技术支持输入 / 输出类型分别为 DOMSource 和 Document 的请求，见关注点 3 和关注点 4。

2. JAXB 标准

JAXP 的缺点是需要编码解析 XML，这增加了开发成本，但对业务逻辑的实现并没有实质的贡献。JAXB 只需要在 POJO 中定义相关的注解（早期人们使用 XML 配置文件来做这件事），使其和 XML 的 schema 对应，无须对 XML 进行程序式解析，弥补了 JAXP 的这一缺点，因此本书推荐使用 JAXB 作为 XML 解析的技术。

JAXB 通过序列化和反序列化实现了 XML 数据和 POJO 对象的自动转换过程。在运行时，JAXB 通过编组（marshall）过程将 POJO 序列化成 XML 格式的数据，通过解编（unmarshall）过程将 XML 格式的数据反序列化为 Java 对象。JAXB 的注解位于 javax.xml.bind.annotation 包中，详情可以访问 JAXB 的参考实现网址 https://jaxb.java.net/tutorial。

需要指出的是，从理论上讲，JAXB 解析 XML 的性能不如 JAXP，但使用 JAXB 的开发效率很高。笔者所在的开发团队即使用 JAXB 解析 XML，从实践体会而言，笔者并不支持 JAXB 影响系统运行性能这样的观点。因为计算机执行的瓶颈在 IO，而无论使用哪种技术解析，XML 数据本身是一样的，区别仅在于解析手段。而 REST 风格以及敏捷思想的宗旨就是简单——开发过程简单化、执行逻辑简单化，因此，如果连 XML 数据都趋于简单，JAXP 带来的性能优势就可以基本忽略不计了。综合考量，实现起来更简单的 JAXB 更适合做 REST 开发。

Jersey 支持使用 JAXBElement 作为 REST 方法参数的形式，也支持直接使用 POJO 作为 REST 方法参数的形式，后一种更为常用。示例代码如下。

```
@POST
@Path("jaxb")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(JAXBElement<Book> bookElement) {
  Book book = bookElement.getValue();
  LOGGER.debug(book.getBookName());
  return book;
}
@POST
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(Book book) {
  LOGGER.debug(book.getBookName());
  return book;
}
```

以上 JAXP 和 JAXB 的测试如下所示，其传输内容是相同的，不同在于服务器端的 REST 方法定义的解析类型和返回值类型。

```
1 > Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="100"
bookName="TEST BOOK"/>
2 < Content-Length: 79
2 < Content-Type: text/html
<?xml version="1.0" encoding="UTF-8"?><book bookId="100" bookName="TEST BOOK"/>
```

从测试结果可以看到，POJO 类的字段是作为 XML 的属性组织起来的，详见如下的图书实体类定义。

```
@XmlRootElement
public class Book implements Serializable {
```

```
//关注点 1: JAXB 属性注解
    @XmlAttribute(name = "bookId")
    public Long getBookId() {
        return bookId;
    }
    @XmlAttribute(name = "bookName")
    public String getBookName() {
        return bookName;
    }
    @XmlAttribute(name = "publisher")
    public String getPublisher() {
        return publisher;
    }
}
```

（1）property 和 element

本例的 POJO 类 Book 的字段都定义为 XML 的属性（property）来组织，POJO 的字段也可以作为元素（element）组织，见关注点 1。如何定义通常取决于对接系统的设计。需要注意的是，如果 REST 请求的传输数据量很大，并且无须和外系统对接的场景，建议使用属性来组织 XML，这样可以极大地减小 XML 格式的数据包的规模。

（2）XML_SECURITY_DISABLE

Jersey 默认设置了 XMLConstants.FEATURE_SECURE_PROCESSING（http://javax.xml.XMLConstants/feature/secure-processing）属性，当属性或者元素过多时，会报“well-formedness error”这样的警告信息。如果业务逻辑确实需要设计一个烦琐的 POJO，可以通过设置 MessageProperties.XML_SECURITY_DISABLE 参数值为 TRUE 来屏蔽。服务器端和客户端示例代码如下。

```
@ApplicationPath("/*")
public class XXXResourceConfig extends ResourceConfig {
    public XXXResourceConfig() {
        packages("xxx.yyy.zzz");
        property(MessageProperties.XML_SECURITY_DISABLE, Boolean.TRUE);
    }
}
ClientConfig config = new ClientConfig();
config.property(MessageProperties.XML_SECURITY_DISABLE, Boolean.TRUE);
```

3.3.6 JSON 类型

JSON 类型已经成为 AJAX 技术中数据传输的实际标准。Jersey 提供了 4 种处理 JSON 数据的媒体包。表 3-6 展示了 4 种技术对 3 种解析流派 [基于 POJO 的 JSON 绑定、基于 JAXB 的 JSON 绑定以及低级的（逐字的）JSON 解析和处理] 的支持情况。MOXy 和 Jackson 的处理方式相同，它们都不支持以 JSON 对象方式解析 JSON 数据，而是以绑定方式解析。Jettison 支持以 JSON 对象方式解析 JSON 数据，同时支持 JAXB 方式的绑定。JSON-P 只支持 JSON 对象方式解析这种方式。

表 3-6 Jersey 对 JSON 的处理方式列表

解析方式 \ JSON 支持包	MOXy	JSON-P	Jackson	Jettison
POJO-based JSON Binding	Yes	No	Yes	No
JAXB-based JSON Binding	Yes	No	Yes	Yes
Low-level JSON parsing & processing	No	Yes	No	Yes

下面将介绍 MOXy、JSON-P、Jackson 和 Jettison 这 4 种 Jersey 支持的 JSON 处理技术在 REST 式的 Web 服务开发中的使用情况。

1. 使用 MOXy 处理 JSON

MOXy 是 EclipseLink 项目的一个模块，其官方网站为：http://www.eclipse.org/eclipselink/moxy.php，宣称 EclipseLink 的 MOXy 组件是使用 JAXB 和 SDO 作为 XML 绑定的技术基础。MOXy 实现了 JSR 222 标准（JAXB 2.2）和 JSR 235 标准（SDO 2.1.1），这使得使用 MOXy 的 Java 开发者能够高效地完成 Java 类和 XML 的绑定，所要花费时间的只是使用注解来定义它们之间的对应关系。同时，MOXy 实现了 JSR 353 标准（Java API for Processing JSON 1.0），以 JAXB 为基础，来实现对 JSR 353 的支持。下面开始讲述使用 MOXy 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

本小节示例所在目录是：jax-rs2-guide\sample\3\1simple-service-moxy。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/1simple-service-moxy

（1）定义依赖

MOXy 是 Jersey 默认的 JSON 解析方式，可以在项目中添加 MOXy 的依赖包来使用 MOXy。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-moxy</artifactId>
</dependency>
```

（2）定义 Application

如果使用 Servlet 3，则可以不定义 web.xml 配置，否则需参考 2.3 节的讲述。

MOXy 的 Feature 接口实现类是 MoxyJsonFeature，默认情况下，Jersey 对其自动探测，无须在 Applicaion 类或其子类显式注册该类。如果不希望出现 Jersey 的这种默认行为，可以通过设置如下属性来禁用自动探测：CommonProperties.MOXY_JSON_FEATURE_DISABLE，两端禁用；ServerProperties.MOXY_JSON_FEATURE_DISABLE，服务器端禁用；ClientProperties.MOXY_JSON_FEATURE_DISABLE，客户端禁用。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        /*
        property(
        org.glassfish.jersey.CommonProperties.MOXY_JSON_FEATURE_DISABLE, true);
        */
    }
}
```

(3) 定义资源类

接下来，我们定义一个图书资源类 BookResource，并在其中实现表述媒体类型为 JSON 的资源方法 getBooks()。支持 JSON 格式的表述的资源类定义如下。

```
@Path("books")
//关注点 1: @Produces 注解和 @Consumes 注解上移到接口
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
    private static final Logger LOGGER = Logger.getLogger(BookResource.class);
    private static final HashMap<Long, Book> memoryBase;
    static {
        memoryBase = com.google.common.collect.Maps.newHashMap();
        memoryBase.put(1L, new Book(1L, "Java Restful Web Services 使用指南"));
        memoryBase.put(2L, new Book(2L, "Java EE 7 精髓"));
    }
    @GET
    //关注点 2: 实现类方法无须再定义 @Produces 注解和 @Consumes 注解
    public Books getBooks() {
        final List<Book> bookList = new ArrayList<>();
        final Set<Map.Entry<Long, Book>> entries = BookResource.memoryBase.entrySet();
        final Iterator<Entry<Long, Book>> iterator = entries.iterator();
        while (iterator.hasNext()) {
            final Entry<Long, Book> cursor = iterator.next();
            BookResource.LOGGER.debug(cursor.getKey());
            bookList.add(cursor.getValue());
        }
        final Books books = new Books(bookList);
        BookResource.LOGGER.debug(books);
        return books;
    }
```

在这段代码中，需要注意的是，资源类 BookResource 上定义了 @Consumes（MediaType.APPLICATION_JSON）和 @Produces（MediaType.APPLICATION_JSON），代表其支持的所有资源方法都使用 MediaType.APPLICATION_JSON 类型作为请求和响应的数据类型，见关注点 1。因此，getBooks() 方法上无须再定义 @Consumes 和 @Produces，见关注点 2。

(4) 单元测试

JSON 处理的单元测试主要关注请求的响应中 JSON 数据的可用性、完整性和一致性。在本章使用的单元测试中，验证 JSON 处理无误的标准是测试的返回值是一个 Java 类型的实体类实例，整个请求处理过程中没有异常发生。测试代码如下。

```
public class JsonTest extends JerseyTest {
    private final static Logger LOGGER = Logger.getLogger(JsonTest.class);
    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testGettingBooks() {
        //关注点 1: 在请求中定义媒体类型为 JSON
        Books books = target("books").request(MediaType.APPLICATION_JSON_TYPE).
        get(Books.class);
        for (Book book : books.getBookList()) {
            LOGGER.debug(book.getBookName());
        }
    }
}
```

在这段代码中，测试方法 testGettingBooks() 定义了请求资源的数据类型为 MediaType.APPLICATION_JSON_TYPE 来匹配服务器端提供的 REST API，其作用是定义请求的媒体类型为 JSON 格式，见关注点 1。

(5) 集成测试

除了单元测试，我们使用 cURL 来做集成测试。启动本示例，输入如下所示的命令，将返回 JSON 格式的数据。

```
curl http://localhost:8080/simple-service-moxy/api/books
curl -H "Content-Type: application/json" http://localhost:8080/simple-service-moxy/api/books
{"bookList":{"book":[{"bookId":1,"bookName":"Java Restful Web Services 使用指南"},
{"bookId":2,"bookName":"Java EE 7 精髓"}]}}
```

2. 使用 JSON-P 处理 JSON

JSON-P 的全称是 Java API for JSON Processing（Java 的 JSON 处理 API），而不是 JSON with Padding（JSONP），两者只是名称相仿，用途却大不相同。JSON-P 是 JSR 353 标准规范，用于统一 Java 处理 JSON 格式数据的 API，其生产和消费的 JSON 数据以流的形式，类似 StAX 处理 XML，并为 JSON 数据建立 Java 对象模型，类似 DOM。而 JSONP 是用于异步请求中传递脚本的回调函数来解决跨域问题的。下面开始讲述使用 JSON-P 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南 本小节示例所在目录是：
jax-rs2-guide\sample\3\4simple-service-jsonp
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/4simple-service-jsonp

（1）定义依赖

使用 JSON-P 方式处理 JSON 类型的数据，需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-processing</artifactId>
</dependency>
```

（2）定义 Application

使用 JSON-P 的应用，默认不需要在其 application 中注册 JsonProcessingFeature，除非使用了如下设置，分别用于在服务器和客户端两侧去活 JSON-P 功能、在服务器端去活 JSON-P 功能、在客户端去活 JSON-P 功能：

- CommonProperties.JSON_PROCESSING_FEATURE_DISABLE
- ServerProperties.JSON_PROCESSING_FEATURE_DISABLE
- ClientProperties.JSON_PROCESSING_FEATURE_DISABLE

JsonGenerator.PRETTY_PRINTING 属性用于格式化 JSON 数据的输出，当属性值为 TRUE 时，MessageBodyReader<T> 和 MessageBodyWriter<T> 实例会对 JSON 数据进行额外处理，使得 JSON 数据可以格式化打印。该属性的设置在 application 中，示例代码如下，见关注点 1。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
//关注点 1: 配置 JSON 格式化输出
        property(JsonGenerator.PRETTY_PRINTING, true);
    }
}
```

（3）定义资源类

资源类 BookResource 同上例一样定义了类级别的 @Consumes 和 @Produces，媒体格式为 MediaType.APPLICATION_JSON。资源类 BookResource 的示例代码如下。

```
@Path("books")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
...
    static {
        memoryBase = com.google.common.collect.Maps.newHashMap();
        //关注点 1: 构建 JsonObjectBuilder 实例
```

```
        JsonObjectBuilder jsonObjectBuilder = Json.createObjectBuilder();
        //关注点 2: 构建 JsonObject 实例
        JsonObject newBook1 = jsonObjectBuilder.add("bookId", 1).
...
    }
    @GET
    public JsonArray getBooks() {
        //关注点 3: 构建 JsonArrayBuilder 实例
        final JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
        final Set<Map.Entry<Long, JsonObject>> entries =
          BookResource.memoryBase.entrySet();
        final Iterator<Entry<Long, JsonObject>> iterator = entries.iterator();
        while (iterator.hasNext()) {
...
        }
        //关注点 4: 构建 JsonArray 实例
        JsonArray result = arrayBuilder.build();
        return result;
    }
```

在这段代码中，JsonObjectBuilder 用于构造 JSON 对象，见关注点 1；JsonObject 是 JSON-P 定义的 JSON 对象类，见关注点 2；JsonArrayBuilder 用于构造 JSON 数组对象，见关注点 3；JsonArray 是 JSON 数组类，见关注点 4。

(4) 单元测试

JSON-P 示例的单元测试需要关注 JSON-P 定义的 JSON 类型，测试验收标准在前一小节 MOXy 的单元测试中已经讲述，示例代码如下。

```
public class JsonTest extends JerseyTest {
    private final static Logger LOGGER = Logger.getLogger(JsonTest.class);
    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testGettingBooks() {
        //关注点 1: 请求的响应类型为 JsonArray
        JsonArray books = target("books").request(MediaType.APPLICATION_JSON_TYPE).
        get(JsonArray.class);
        for (JsonValue jsonValue : books) {
            //关注点 2: 强制转换 JsonValue 为 JsonObject
            JsonObject book = (JsonObject) jsonValue;
            LOGGER.debug(book.getString("bookName"));//关注点 3: 打印输出测试结果
        }
    }
}
```

在这段代码片段中，JsonArray 是 getBooks() 方法的返回类型，get 请求发出后，对应的方法是 getBooks() 方法，见关注点 1。JsonValue 类型是一种抽象化的 JSON 数据类型，此处

类型强制转化为 JsonObject，见关注点 2。getString() 方法是将 JsonObject 对象的某个字段以字符串类型返回，见关注点 3。

(5) 集成测试

使用 cURL 对本示例进行集成测试的结果如下所示。JSON 数据结果可以格式化打印输出。

```
curl http://localhost:8080/simple-service-jsonp/api/books
[
    {
        "bookId":1,
        "bookName":"JSF 和 Richfaces 使用指南 ",
        "publisher":" 电子 • 博文 "
    },
    {
        "bookId":2,
        "bookName":"Java Restful Web Services 使用指南 ",
        "publisher":" 机工 • 华章 "
    }
]
curl http://localhost:8080/simple-service-jsonp/api/books/book?id=2
{
    "bookId":2,
    "bookName":"Java Restful Web Services 使用指南 ",
    "publisher":" 机工 • 华章 "
}
curl -H "Content-Type: application/json" -X POST
-d "{\"bookName\":\"abc\",\"publisher\":\"me\"}"
http://localhost:8080/simple-service-jsonp/api/books
{
    "bookId":12447994371104,
    "bookName":"abc",
    "publisher":"me"
}
```

3. 使用 Jackson 处理 JSON

Jackson 是一种目前比较流行的 JSON 支持技术，其源代码托管于 GitHub，地址是：https://github.com/FasterXML/jackson。Jackson 提供了 3 种 JSON 解析方式。

- 第一种是基于流式 API 的增量式解析 / 生成 JSON 的方式，读写 JSON 内容的过程是通过离散事件触发的，其底层基于 StAX API 读取 JSON 使用 org.codehaus.jackson.JsonParser，写入 JSON 使用 org.codehaus.jackson.JsonGenerator。
- 第二种是基于树形结构的内存模型，提供一种不变式的 JsonNode 内存树模型，类似 DOM 树。
- 第三种是基于数据绑定的方式，org.codehaus.jackson.map.ObjectMapper 解析，使用 JAXB 的注解。

下面开始讲述使用 Jackson 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南 本小节示例所在目录是：
jax-rs2-guide\sample\3\2simple-service-jackson
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/2simple-service-jackson

（1）定义依赖

使用 Jackson 方式处理 JSON 类型的数据，需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId>
</dependency>
```

（2）定义 Application

使用 Jackson 的应用，需要在其 application 中注册 JacksonFeature。同时，如果有必要根据不同的实体类做详细的解析，则可以注册 ContextResolver 的实现类，示例代码如下。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        register(JacksonFeature.class);
        // 关注点 1: 注册 ContextResolver 的实现类 JsonContextProvider
        register(JsonContextProvider.class);
    }
}
```

在这段代码中，注册了 ContextResolver 的实现类 JsonContextProvider，用于提供 JSON 数据的上下文，见关注点 1。有关 ContextResolver 的详细信息，可参考 4.2 节。

（3）定义 POJO

本例定义了 3 种不同方式的 POJO，以演示 Jackson 处理 JSON 的多种方式，分别是 JsonBook、JsonHybridBook 和 JsonNoJaxbBook。

第一种方式是仅用 JAXB 注解的普通的 POJO，示例类 JsonBook 如下。

```
@XmlRootElement
@XmlType(propOrder = {"bookId", "bookName", "chapters"})
public class JsonBook {
    private String[] chapters;
    private String bookId;
    private String bookName;
    public JsonBook() {
        bookId = "1";
        bookName = "Java Restful Web Services 使用指南";
        chapters = new String[0];
    }
    public String getBookId() {
        return bookId;
    }
```

```
    public void setBookId(String bookId) {
        this.bookId = bookId;
    }
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
    public String[] getChapters() {
        return chapters;
    }
    public void setChapters(String[] chapters) {
        this.chapters = chapters;
    }
}
```

第二种方式是将JAXB的注解和Jackson提供的注解混合使用的POJO，示例类JsonHybridBook如下。

```
@XmlRootElement //关注点1：使用JAXB注解
public class JsonHybridBook {
//关注点2：使用Jackson注解
@JsonProperty("bookId")
private String bookId;
    @JsonProperty("bookName")
    private String bookName;
    public JsonHybridBook() {
        bookId = "2";
        bookName = "Java Restful Web Services使用指南";
    }
}
```

在这段代码中，分别使用了JAXB的注解javax.xml.bind.annotation.XmlRootElement，见关注点1，以及Jackson的注解org.codehaus.jackson.annotate.JsonProperty，见关注点2，分别定义XML根元素和XML属性。

第三种方式是不使用任何注解的POJO，示例类JsonNoJaxbBook如下。

```
public class JsonNoJaxbBook {
    private String[] chapters;
    private String bookId;
    private String bookName;
    public JsonNoJaxbBook() {
        bookId = "3";
        bookName = "Java Restful Web Services使用指南";
        chapters = new String[0];
    }
    public String[] getChapters() {
        return chapters;
    }
    public void setChapters(String[] chapters) {
```

```
        this.chapters = chapters;
    }
    public String getBookId() {
        return bookId;
    }
    public void setBookId(String bookId) {
        this.bookId = bookId;
    }
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
}
```

这样的三种 POJO 如何使用 Jackson 来处理呢？我们继续往下看。

（4）定义资源类

资源类 BookResource 用于演示 Jackson 对上述三种不同 POJO 的支持，示例代码如下。

```
@Path("books")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {
    @Path("/emptybook")
    @GET
//关注点 1：支持第一种方式的 POJO 类型
    public JsonBook getEmptyArrayBook() {
return new JsonBook();
    }
    @Path("/hybirdbook")
    @GET
//关注点 2：支持第二种方式的 POJO 类型
    public JsonHybridBook getHybirdBook() {
return new JsonHybridBook();
    }
    @Path("/nojaxbbook")
    @GET
//关注点 3：支持第三种方式的 POJO 类型
    public JsonNoJaxbBook getNoJaxbBook() {
return new JsonNoJaxbBook();
    }
...
```

在这段代码中，资源类 BookResource 定义了路径不同的 3 个 GET 方法，返回类型分别对应上述的 3 种 POJO，见关注点 1 ~ 关注点 3。有了这样的资源类，我们就可以向其发送 GET 请求，并获取不同类型的 JSON 数据，以研究 Jackson 是如何支持这 3 种 POJO 的 JSON 转换。

（5）上下文解析实现类

JsonContextProvider 是 ContextResolver（上下文解析器）的实现类，其作用是根据上下

文提供的 POJO 类型，分别提供两种解析方式。第一种是默认的方式，第二种是混合使用 Jackson 和 JAXB。两种解析方式的示例代码如下。

```
@Provider
public class JsonContextProvider implements ContextResolver<ObjectMapper> {
    final ObjectMapper d;
    final ObjectMapper c;
    public JsonContextProvider() {
        //关注点 1: 实例化 ObjectMapper
        d = createDefaultMapper();
        c = createCombinedMapper();
    }
    private static ObjectMapper createCombinedMapper() {
        Pair ps = createIntrospector();
        ObjectMapper result = new ObjectMapper();
        result.setDeserializationConfig(
        result.getDeserializationConfig().withAnnotationIntrospector(ps));
        result.setSerializationConfig(
        result.getSerializationConfig().withAnnotationIntrospector(ps));
        return result;
    }
    private static ObjectMapper createDefaultMapper() {
        ObjectMapper result = new ObjectMapper();
        result.configure(Feature.INDENT_OUTPUT, true);
        return result;
    }
    private static Pair createIntrospector() {
        AnnotationIntrospector p = new JacksonAnnotationIntrospector();
        AnnotationIntrospector s = new JaxbAnnotationIntrospector();
        return new Pair(p, s);
    }
    @Override    public ObjectMapper getContext(Class<?> type) {
        //关注点 2: 判断 POJO 类型返回相应的 ObjectMapper 实例
        if (type == JsonHybridBook.class) {
            return c;
        } else {
            return d;
        }
    }
}
```

在这段代码中，JsonContextProvider 定义并实例化了两种类型 ObjectMapper，见关注点 1，在实现接口方法 getContext() 中，通过判断当前 POJO 的类型，返回两种 ObjectMapper 实例之一，见关注点 2。通过这样的实现，当流程获取 JSON 上下文时，即可使用 Jackson 依赖包完成对相关 POJO 的处理。

（6）单元测试

单元测试类 BookResourceTest 的目的是对支持上述三种 POJO 的资源地址发起请求并测试结果，示例代码如下。

```
public class BookResourceTest extends JerseyTest {
    private static final Logger LOGGER = Logger.getLogger(BookResourceTest.class);
    WebTarget booksTarget = target("books");
    @Override
    protected ResourceConfig configure() {
        //关注点 1: 服务器端配置
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        ResourceConfig resourceConfig = new ResourceConfig(BookResource.class);
        //关注点 2: 注册 JacksonFeature
        resourceConfig.register(JacksonFeature.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig config) {
        //关注点 3: 注册 JacksonFeature
        config.register(new JacksonFeature());
        config.register(JsonContextProvider.class);
    }
    @Test
    //关注点 4: 测试出参为 JsonBook 类型的资源方法
    public void testEmptyArray() {
        JsonBook book = booksTarget.path("emptybook").request(MediaType.APPLICATION_JSON).
        get(JsonBook.class);
        LOGGER.debug(book);
    }
    @Test
    //关注点 5: 测试出参为 JsonHybridBook 类型的资源方法
    public void testHybrid() {
        JsonHybridBook book = booksTarget.path("hybirdbook").
        request(MediaType.APPLICATION_JSON).
        get(JsonHybridBook.class);
        LOGGER.debug(book);
    }
    @Test
    //关注点 6: 测试出参为 JsonNoJaxbBook 类型的资源方法
    public void testNoJaxb() {
        JsonNoJaxbBook book = booksTarget.path("nojaxbbook").
        request(MediaType.APPLICATION_JSON).
        get(JsonNoJaxbBook.class);
        LOGGER.debug(book);
    }
...
```

在这段代码中，首先要在服务器端注册支持 Jackson 功能，见关注点 2，同时在客户端也要注册支持 Jackson 功能并注册 JsonContextProvider，见关注点 3。该测试类包含了用于测试 3 种类型 POJO 的测试用例，见关注点 4 ~ 关注点 6。注意，configure() 方法是覆盖测试服务器实例行为，configureClient() 方法是覆盖测试客户端实例行为，见关注点 1。

（7）集成测试

使用 cURL 对本例进行集成测试，结果如下所示。

```
curl http://localhost:8080/simple-service-jackson/api/books
{
  "bookList" : [ {
    "bookId" : 1,
    "bookName" : "Java Restful Web Services使用指南",
    "publisher" : null
  }, {
    "bookId" : 2,
    "bookName" : "Java EE 7 精髓",
    "publisher" : null
  } ]
}
curl http://localhost:8080/simple-service-jackson/api/books/emptybook
{
  "chapters" : [ ],
  "bookId" : "1",
  "bookName" : "Java Restful Web Services使用指南"
}
curl http://localhost:8080/simple-service-jackson/api/books/hybirdbook
{"bookId":"2","bookName":"Java Restful Web Services使用指南"}
curl http://localhost:8080/simple-service-jackson/api/books/nojaxbbook
{
  "chapters" : [ ],
  "bookId" : "3",
  "bookName" : "Java Restful Web Services使用指南"
}
```

4. 使用 Jettison 处理 JSON

Jettison 是一种使用 StAX 来解析 JSON 的实现。项目地址是：http://jettison.codehaus.org。Jettison 项目起初用于为 CXF 提供基于 JSON 的 Web 服务，在 XStream 的 Java 对象的序列化中也使用了 Jettison。Jettison 支持两种 JSON 映射到 XML 的方式。Jersey 默认使用 Mapped 方式，另一种叫做 BadgerFish。

下面开始讲述使用 Jettison 实现在 REST 应用中解析 JSON 的完整过程。

阅读指南

本小节示例所在目录是：

jax-rs2-guide\sample\3\3simple-service-jettison

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/3simple-service-jettison

---

(1) 定义依赖

使用 Jettison 方式处理 JSON 类型的数据，需要在项目的 Maven 配置中声明如下依赖。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jettison</artifactId>
</dependency>
```

(2) 定义 Application

使用 Jettison 的应用，需要在其 application 中注册 JettisonFeature。同时，如果有必要根据不同的实体类做详细的解析，那么可以注册 ContextResolver 的实现类，示例代码如下。

```
@ApplicationPath("/api/*")
public class JsonResourceConfig extends ResourceConfig {
    public JsonResourceConfig() {
        register(BookResource.class);
        // 关注点 1: 注册 JettisonFeature 和 ContextResolver 的实现类 JsonContextResolver
        register(JettisonFeature.class);
        register(JsonContextResolver.class);
    }
}
```

在这段代码中，注册了 Jettison 功能 JettisonFeature 和 ContextResolver 的实现类 JsonContext Resolver，以便使用 Jettison 处理 JSON，见关注点 1。

(3) 定义 POJO

本例定义了两个类名不同但内容相同的 POJO（JsonBook 和 JsonBook2），用以演示 Jettison 对 JSON 数据以 JETTISON_MAPPED（default notation）和 BADGERFISH 两种不同方式处理的情况。

```
@XmlRootElement
public class JsonBook {
    private String bookId;
    private String bookName;
    public JsonBook() {
        bookId = "1";
        bookName = "Java Restful Web Services 使用指南";
    }
    @XmlElement
    public String getBookId() {
        return bookId;
    }
    public void setBookId(String bookId) {
        this.bookId = bookId;
    }
    @XmlElement
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
}
```

(4) 定义资源类

资源类 BookResource 为两种 JSON 方式提供了资源地址，示例代码如下。

```
@Path("books")
```

```
public class BookResource {
    private static final Logger LOGGER = Logger.getLogger(BookResource.class);
    private static final HashMap<Long, Book> memoryBase;
    static {
        memoryBase = com.google.common.collect.Maps.newHashMap();
        memoryBase.put(1L, new Book(1L, "Java Restful Web Services 使用指南"));
        memoryBase.put(2L, new Book(2L, "Java EE 7 精髓"));
    }
    @Path("/jsonbook")
    @GET
    // 关注点 1: 返回类型为 JsonBook 的 GET 方法
    public JsonBook getBook() {
        final JsonBook book = new JsonBook();
        BookResource.LOGGER.debug(book);
        return book;
    }
    @Path("/jsonbook2")
    @GET
    // 关注点 2: 返回类型为 JsonBook2 的 GET 方法
    public JsonBook2 getBook2() {
        final JsonBook2 book = new JsonBook2();
        BookResource.LOGGER.debug(book);
        return book;
    }
...
}
```

在这段代码中，资源类 BookResource 定义了路径不同的两个 GET 方法，返回类型分别是 JsonBook 和 JsonBook2，分别见关注点 1 和关注点 2。有了这样的资源类，我们就可以向其发送 GET 请求，并获取不同类型的 JSON 数据，以研究 Jettison 是如何处理 JETTISON_MAPPED 和 BADGERFISH 两种不同格式的 JSON 数据的。

（5）上下文解析实现类

JsonContextResolver 实现了 ContextResolver 接口，示例代码如下。

```
@Provider
public class JsonContextResolver implements ContextResolver<JAXBContext> {
    private final JAXBContext context1;
    private final JAXBContext context2;
    @SuppressWarnings("rawtypes")
    public JsonContextResolver() throws Exception {
        Class[] clz = new Class[]{JsonBook.class, JsonBook2.class, Books.class, Book.class};
        // 关注点 1: 实例化 JettisonJaxbContext
        this.context1 = new JettisonJaxbContext(JettisonConfig.DEFAULT, clz);
        this.context2 = new JettisonJaxbContext(JettisonConfig.badgerFish().build(), clz);
    }
    @Override
    public JAXBContext getContext(Class<?> objectType) {
        // 关注点 2: 判断 POJO 类型返回相应的 JAXBContext 实例
        if (objectType == JsonBook2.class) {
            return context2;
```

```
        } else {
            return context1;
        }
    }
}
```

在这段代码中，JsonContextResolver 定义了两种 JAXBContext，分别使用 Mapped 方式和 BadgerFish 方式，见关注点 1。这两种方式的参数信息来自 Jettison 依赖包的 JettisonConfig 类。在实现接口方法 getContext() 中，根据不同的 POJO 类型，返回两种 JAXBContext 实例之一，见关注点 2。通过这样的实现，当流程获取 JSON 上下文时，即可使用 Jettison 依赖包完成对相关 POJO 的处理。

（6）单元测试

单元测试类 BookResourceTest 的目的是对支持上述两种 JSON 方式的资源地址发起请求并测试结果，示例代码如下。

```
public class BookResourceTest extends JerseyTest {
    private static final Logger LOGGER = Logger.getLogger(BookResourceTest.class);
    @Override
    protected ResourceConfig configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        ResourceConfig resourceConfig = new ResourceConfig(BookResource.class);
        // 关注点 1: 注册 JettisonFeature 和 JsonContextResolver
        resourceConfig.register(JettisonFeature.class);
        resourceConfig.register(JsonContextResolver.class);
        return resourceConfig;
    }
    @Override
    protected void configureClient(ClientConfig config) {
        // 关注点 2: 注册 JettisonFeature 和 JsonContextResolver
        config.register(new JettisonFeature()).register(JsonContextResolver.class);
    }
    @Test
    public void testJsonBook() {
        // 关注点 3: 测试返回类型为 JsonBook 的 GET 方法
        JsonBook book = target("books").path("jsonbook").
        request(MediaType.APPLICATION_JSON).
        get(JsonBook.class);
        LOGGER.debug(book);
        //{"jsonBook":{"bookId":1,"bookName":"abc"}}
    }
    @Test
    public void testJsonBook2() {
        // 关注点 4: 测试返回类型为 JsonBook2 的 GET 方法
        JsonBook2 book = target("books").path("jsonbook2").
        request(MediaType.APPLICATION_JSON).
        get(JsonBook2.class);
        LOGGER.debug(book);
        //{"jsonBook2":{"bookId":{"$":"1"},"bookName":{"$":"abc"}}}
    }
...
}
```

在这段代码中，首先要在服务器和客户端两侧注册 Jettison 功能和 JsonContextResolver，见关注点 1 和关注点 2。该测试类包含了用于测试两种格式 JSON 数据的测试用例，见关注点 3 和 4。

（7）集成测试

使用 cURL 对本例进行集成测试，结果如下所示。可以看到 Mapped 和 BadgerFish 两种方式的 JSON 数据内容不同。

```
curl http://localhost:8080/simple-service-jettison/api/books
{"books":{"bookList":{"book":[{"@bookId":"1","@bookName":"Java Restful Web
Services 使用指南 "},{"@bookId":"2","@bookName":"Java EE 7 精髓 "}]}}}
Jettison mapped notation
curl http://localhost:8080/simple-service-jettison/api/books/jsonbook
{"jsonBook":{"bookId":1,"bookName":"Java Restful Web Services 使用指南 "}}
Badgerfish notation
curl http://localhost:8080/simple-service-jettison/api/books/jsonbook2
{"jsonBook2":{"bookId":{"$":"1"},"bookName":{"$":"Java Restful Web Services 使用指南 "}}}
```

最后简要介绍一下 Atom 类型。

Atom 是一种基于 XML 的文档格式，该格式的标准定义在 IETF RFC 4287（Atom Syndication Format，即 Atom 联合格式），其推出的目的是用来替换 RSS。AtomPub 是基于 Atom 的发布协议，定义在 IETF RFC 5023（Atom Publishing Protocol）。

Jersey 2 没有直接引入支持 Atom 格式的媒体包，但 Jersey 1.x 中包含 jersey-atom 包。这说明 Jersey 的基本架构对支持基于 XML 类型的数据并不是问题，这种可插拔的媒体包支持对于 Jersey 本身更具灵活性，对使用 Jersey 的 REST 服务更具可扩展性。

3.4 REST 连通性

REST 的一个重要的特性就是连通性。Web Link 和 HATEOAS 以不同方式实现了 REST 式服务的连通性。

- Web Link 定义在 IETF RFC 5988（Web Linking），是通过在 HTTP 头中定义链接信息，以描述当前页面与链接页面之间的关系。Web Link 是一种过渡型链接（Transitional Links）。JAX-RS 2.0 引入了 javax.ws.rs.core.Link 类，用来处理 Web Link 的表述。
- HATEOAS（Hypermedia as the Engine of Application State，超媒体作为应用程序状态引擎）。HATEOAS 的形式是包含链接信息的超媒体文档。HATEOAS 的核心是“引擎”，该引擎的目的是通过请求的响应实体将超媒体信息返回给客户端，超媒体信息可以告诉用户，如果接下来选择去往某个链接（或者链接列表中的某个链接），应用的状态就会如超媒体描述的那样发生转变。HATEOAS 是一种结构型链接（Structural Links）。Jersey 2 中可以使用 XML 实现 HATEOAS 的结构要求。

下面讲述 Jersey 中是如何实现 Web Link 和 HATEOAS 这两种 REST 连通性实践方式的。

阅读指南

3.4 节示例所在目录是 jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.link。

3.4.1 过渡型链接

Web Link 通过使用 HTTP 的头信息来传递操作链接，在 Jersey 中使用 javax.ws.rs.core.Link 类可以非常简洁地实现支持 Web Link 的资源类，示例代码如下。

```
@Path("weblink-resource")
public class WebLinkResource {
    @Context
    UriInfo uriInfo;
    @POST
    @Produces(MediaType.APPLICATION_XML)
    @Consumes(
    { MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML, MediaType.TEXT_XML })
    public Response saveBook(final Book book) {
        final long newId = System.nanoTime();
        book.setBookId(newId);
        LinkCache.map.put(newId, book);
        // 关注点 1: 通过 uriInfo 实例获取资源路径
        final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
        final URI location = ub.path("" + newId).build();
        // 关注点 2: 通过模板获取资源路径
        final String uriTemplate = "http://{host}:{port}/{path}/{param}";
        final URI location2 = UriBuilder.fromUri(uriTemplate)
        .resolveTemplate("host", "localhost").resolveTemplate("port", "9998")
        .resolveTemplate("path", "weblink-resource")
        .resolveTemplate("param", newId).build();
        // 关注点 3: 通过模板方法获取资源路径
        final UriBuilder ub3 = uriInfo.getAbsolutePathBuilder();
        final URI location3 = ub3.scheme("http").host("localhost").port(9998)
        .path("weblink-resource").path("" + newId).build();
        // 关注点 4: 为响应实例添加路径信息
        return Response.created(location).link(location2, "view1")
        .link(location3, "view2").entity(book).build();
    }
}
```

在这段代码中，使用了 3 种方式构建 URI 实例。第一种方式是通过调用 uriInfo 实例的 getAbsolutePathBuilder() 方法可以获取当前请求的绝对路径，然后基于此路径添加资源 ID 信息，见关注点 1。第二种方式是为 UriBuilder 提供路径模板，然后链式调用 resolveTemplate() 方法传递并解析模板参数，最后通过 UriBuilder 的 build() 方法生成 URI 实例，见关注点 2。第三种方式和第二种类似，不同的是模板信息被具体方法替代，见关注点 3。最后，这 3 个与 Link 相关的 URI 实例由 Response 构建，作为返回值响应给客户端，见关注点 4。

3.4.2 结构型链接

HATEOAS 用以代替聚集数据并避免描述膨胀，通常使用 Atom 格式在实体字段中提供链接信息。本例使用 XML 格式来支持 HATEOAS，折中的设计是在 POJO 中额外定义一个链接字段。支持 HATEOAS 的资源类示例如下。

```
@Path("hateoas-resource")
public class HATEOASResource {
@Context
    UriInfo uriInfo;
    @POST
    @Produces({ MediaType.APPLICATION_XML })
    @Consumes({ MediaType.APPLICATION_XML })
    public BookWrapper saveBook(final Book book) {
        final long newId = System.nanoTime();
        book.setBookId(newId);
        LinkCache.map.put(newId, book);
        //关注点 1: 通过uriInfo 实例获取资源路径
        final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
        final URI uri = ub.path("" + newId).build();
        BookWrapper b = new BookWrapper();
        b.setBook(book);
        //关注点 2: 将资源路径赋给资源实体
        b.setLink(uri.toString());
        return b;
    }
}
```

在这段代码中，URI 实例由上下文 uriInfo 中获取的绝对路径和资源 ID 组成，见关注点 1，该链接信息被赋值到 POJO 实例的 link 属性中，以实现 HATEOAS，见关注点 2。

宅人坑事

REST 连通性的实践手段非常多，推荐读者从成熟的产品中学习其设计。如果有可能，这里推荐 Jenkins 和 RallyDev 两个敏捷开发中常用的平台，它们提供了比较“舒适”的连通性设计。比如在 RallyDev 中，为一个测试用例结果添加测试用例属性（该属性是必须输入项），其内容并不是对应测试用例的实例，而是该测试用例的引用地址字符串。这样的设计不但减少了网络传输的负载，还方便在调试和维护时排错。

3.5 REST 响应处理

REST 的响应处理结果应包括响应头中 HTTP 状态码，响应实体中媒体参数类型和返回值类型，以及异常情况处理。JAX-RS 2.0 支持 4 种返回值类型的响应，分别是无返回值、返回 Response 类实例、返回 GenericEntity 类实例和返回自定义类实例。下面，逐一讲述这 4 种返回值类型。

阅读指南

3.5 节示例所在目录是 jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.response。

3.5.1 返回类型

（1）void

在返回值类型是 void 的响应中，其响应实体为空，HTTP 状态码为 204。在前面的 DELETE 方法讲述中已经介绍过，再来看一下这种类型的资源方法。

```
@DELETE
@Path("{s}")
// 关注点 1：无返回值的 DELETE 方法
public void delete(@PathParam("s") final String s) {
  LOGGER.debug(s);
}
```

因为 delete() 方法无须返回更多的关于资源表述的信息，所以该方法没有返回值，即返回值类型为 void，见关注点 1。

（2）Response

在返回值类型为 Response 的响应中，响应实体为 Response 类的 entity() 方法定义的实体类实例。如果该内容为空，则 HTTP 状态码为 204，否则 HTTP 状态码为 200 OK。示例代码如下。

```
@POST
@Path("c")
public Response get(final String s) {
  LOGGER.debug(s);
  //Response.noContent().build();
// 关注点 1：构建无返回值的响应实例
  return Response.ok().entity("char[]:" + s).build();
}
```

在这段代码中，Response 首先定义了 HTTP 的状态码为 ok，然后填充实体信息，最后调用 build() 方法构建 Response 实例，见关注点 1。

（3）GenericEntity

通用实体类型作为返回值的情况并不常用。其形式是构造一个统一的实体实例并将其返回，实体实例作为第一个参数，该实体类型作为第二个参数。示例代码如下。

```
@POST
@Path("b")
public String get(final byte[] bs) {
  for (final byte b : bs) {
      LOGGER.debug(b);
  }
```

```
    return "byte[]:" + new String(bs);
  }
  /*
  public GenericEntity<String> get(final byte[] bs) {
    for (final byte b : bs) {
          LOGGER.debug(b);
    }
  // 关注点 1: 构建 GenericEntity 实例
    return new GenericEntity<String>("byte[]:" + new String(bs), String.class);
  }
  */
```

在这段代码中，GenericEntity 的第一个参数是由 byte 数组实例作为参数构建的字符串实例，第二个参数是字符串类，见关注点 1。

(4) 自定义类型

JDK 中的类（比如 File、String 等）都可以作为返回值类型，更常用的是返回自定义的 POJO 类型，前述多个例子就是这样做的。下面再来看一个示例。

```
@POST
@Path("f")
// 关注点 1: GET 方法的返回类型为 File
public File get(final File f) throws FileNotFoundException, IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(f))) {
        String s;
        do {
              s = br.readLine();
              LOGGER.debug(s);
        } while (s != null);
        return f;
  }
}
@POST
@Consumes({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
@Produces(MediaType.APPLICATION_XML)
// 关注点 2: POST 方法的返回值是自定义类 Book
public Book getEntity(Book book) {
  LOGGER.debug(book.getBookName());
  return book;
}
@POST
@Consumes({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
@Produces(MediaType.APPLICATION_XML)
// 关注点 3: POST 方法的返回值是自定义类 Book
public Book getEntity(JAXBElement<Book> bookElement) {
  Book book = bookElement.getValue();
  LOGGER.debug(book.getBookName());
  return book;
}
```

在这段代码中，返回值类型有来自 JDK 的 File 类型，见关注点 1，也有自定义的 POJO 类型，见关注点 2 和关注点 3。

3.5.2 处理异常

实现 REST 的资源方法时应使其具有良好的异常处理能力，这包括异常的定义和错误状态码的正确返回。

1. 处理状态码

首先通过表 3-7 了解一下 REST 中常用的 HTTP 状态码，我们应当在处理异常的同时，为 REST 请求的客户端提供对应的状态码。

表 3-7 HTTP 常用状态码列表

状态码	含义
200 OK	服务器正常响应
201 Created	创建新实体，响应头 Location 指定访问该实体的 URL
202 Accepted	服务器接收请求，处理尚未完成。可用于异步处理机制
204 No Content	服务器正常响应，但响应实体为空
301 Moved Permanently	请求资源的地址发生永久变动，响应头 Location 指定新的 URL
302 Found	请求资源的地址发生临时变动
304 Not Modified	客户端缓存资源依然有效
400 Bad Request	请求信息出现语法错误
401 Unauthorized	请求资源无法授权给未验证用户
403 Forbidden	请求资源未授权当前用户
404 Not Found	请求资源不存在
405 Method Not Allowed	请求方法不匹配
406 Not Acceptable	请求资源的媒体类型不匹配
500 Internal Server Error	服务器内部错误，意外终止响应
501 Not Implemented	服务器不支持当前请求

JAX-RS 2.0 规定的 REST 式的 Web 服务的基本异常类型为运行时异常 WebApplicationException 类。该类包含 3 个主要的子类，分别对应：

- HTTP 状态码为 3xx 的重定向类 RedirectionException。
- HTTP 状态码为 4xx 的请求错误类 ClientErrorException。
- HTTP 状态码为 5xx 的服务器错误类 ServerErrorException。

它们各自的子类对照 HTTP 状态码再细分，比如常见的 HTTP 状态码 404 错误，对应的错误类为 NotFoundException，可参考图 3-4。

除了 Jersey 提供的标准异常类型，我们也可以根据业务需要自定义相关的业务异常类，示例如下。

```
// 关注点 1: 定义 WebApplicationException 接口实现类
public class Jaxrs2GuideNotFoundException extends WebApplicationException {
    public Jaxrs2GuideNotFoundException() {
```

```
// 关注点 2: 定义 HTTP 状态
        super(javax.ws.rs.core.Response.Status.NOT_FOUND);
    }
    public Jaxrs2GuideNotFoundException(String message) {
        super(message);
    }
}
```

在这段代码中，Jaxrs2GuideNotFoundException 类继承自 JAX-RS 2.0 的 WebApplicationException 类，见关注点 1。其默认构造子提供了 HTTP 状态码，其值为 Response.Status.NOT_FOUND，见关注点 2。

图 3-4 Jersey 定义的异常类型

2. ExceptionMapper

Jersey 框架为我们提供了更为通用的异常处理方式。通过实现 ExceptionMapper 接口并使用 @Provider 注解将其定义为一个 Provider，可以实现通用的异常的面向切面处理，而非针对某一个资源方法的异常处理，示例如下。

```
@Provider
public class EntityNotFoundMapper
implements ExceptionMapper<Jaxrs2GuideNotFoundException>{
// 关注点 1: 定义 ExceptionMapper 接口实现类
    @Override
    public Response toResponse(final Jaxrs2GuideNotFoundException ex) {
// 关注点 2: 拦截并返回新的响应实例
        return Response.status(404).entity(ex.getMessage()).type("text/plain").build();
```

```
    }
}
```

在这段代码中，EntityNotFoundMapper 实现了 ExceptionMapper 接口，并提供了泛型类型为前面刚定义的 Jaxrs2GuideNotFoundException 类，见关注点 1。当响应中发生了 Jaxrs2GuideNotFoundException 类型的异常时，响应流程就会被拦截并补充 HTTP 状态码和异常消息，以文本作为媒体格式返回给客户端，见关注点 2。

3.6 REST 内容协商

阅读指南

3.6 节示例所在目录是 jax-rs2-guide\sample\3\simple-service-3。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/3/simple-service-3。
相关包：com.example.conneg。

一个资源可以有不同格式的表述，表述（即响应实体）的内容是人类可识别的信息，服务器很难使用一种表述来适应所有用户。conneg（HTTP Content Negotiation，内容协商）是指在服务器提供的多种表述中，为特定的请求选择最好的一种表述的处理过程。那么什么是最好，又怎样做到最好呢？服务器和客户端 / 浏览器之间往复通信来协商用于交换数据的内容格式等信息，达成一致即为最好。内容协商定义在 RFC 2616 的第 12 节（http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html）。

客户端 / 浏览器通过使用 HTTP Accept、Accept-Charset、Accept-Language 和 Accept-Encoding 头来定义接收头的信息，将其所期待的格式或 MIME 类型告知服务器，服务器根据协商算法，返回客户端 / 浏览器可接收的数据信息。内容协商不只是数据格式协商，还包括语言、编码、字符集等信息。Accept 用于数据类型协商，Accept-Language 用于语言协商，Accept-Charset 用于字符集协商，Accept-Encoding 用于压缩算法协商。

JAX-RS 2.0 对内容协商的支持，是通过 @Produces 实现的，对于其他协商，没有从架构上提供支持，可以通过编码从请求头中获取信息并处理。

3.6.1 @Produces 注解

注解 @Produces 用于定义方法的响应实体的数据类型。可以定义一个或多个，同时可以为每种类型定义质量因素（quality factor）。质量因素是取值范围从 0 ~ 1 的小数值。如果不定义质量因素，那么该类型的质量因素默认为 1。我们将结合示例深入了解 @Produces 注解对媒体类型的影响，代码如下。

```
@Path("conneg-resource")
public class ConnegResource {
    @GET
```

```
    @Path("{id}")
//关注点 1: 媒体类型为 XML
    @Produces(MediaType.APPLICATION_XML)
    public Book getJaxbBook(@PathParam("id") final Long bookId) {
        return new Book(bookId);
    }
    @GET
    @Path("{id}")
//关注点 2: 媒体类型为 JSON
    @Produces(MediaType.APPLICATION_JSON)
    public Book getJsonBook(@PathParam("id") final Long bookId) {
        return new Book(bookId);
    }
...
}
```

在这段代码中，getJaxbBook() 和 getJsonBook() 是同等质量因素、资源地址相同的两个 GET 方法，一个定义响应实体格式为 XML，另一个定义响应实体格式为 JSON，分别见关注点 1 和关注点 2。那么对同一个资源的访问，JAX-RS 2.0 该如何选择处理方法呢？如果请求中明确定义可接收的数据类型为两者之一，那么处理方法应该是定义相应数据类型的那个方法。如果两者都定义了，那么处理方法应该是质量因素高的那个方法。如果两者都定义，而且数据类型的质量因素是相等的，或者没有定义 Accept，那么 XML 的方法会被优先选择。

客户端明确表述格式为 XML，Jersey 通过内容协商，会选择 getJaxbBook() 作为相应的资源方法来处理该请求。其测试代码如下所示。

```
WebTarget path = target("conneg-resource").path("123");
Builder request = path.request(MediaType.APPLICATION_XML_TYPE);
Book book = request.get(Book.class);
1 > GET http://localhost:9998/conneg-resource/123
1 > Accept: application/xml
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="123"/>
```

接下来，测试一个稍微复杂的内容协商。客户端明确表述格式的质量因素 JSON 高于 XML，Jersey 会选择资源方法 getJsonBook() 来处理请求。其测试代码如下所示。

```
WebTarget path = target("conneg-resource").path("123");
Builder request = path.request();
request.header("Accept", "application/xml;q=0.1,application/json;q=0.2");
Book book = request.get(Book.class);
1 > GET http://localhost:9998/conneg-resource/123
1 > Accept: application/xml;q=0.1,application/json;q=0.2
2 < Content-Type: application/json
{"bookId":123}
```

现在我们清楚了两个同等方法的场景，再来看一下一个方法中多种数据类型的场景，示例代码如下。

```
@GET
@Produces({ "application/json; qs=.9", "application/xml; qs=.5" })
```

```
@Path("book/{id}")
public Book getBook(@PathParam("id") final Long bookId) {
  return new Book(bookId);
}
```

在这段代码中，getBook() 方法定义了 XML 和 JSON 两种表述数据类型，XML 的质量因素是 0.5（0 可以省略），JSON 的是 0.9。

因此，我们可以推断，如果客户端请求中明确接收的数据类型是两者之一，那么响应实体使用指定类型。如果没有定义或者两者都定义且 JSON 的质量因素大于或者等于 XML 的质量因素，则返回 JSON 类型。还有一种用例是，两者都定义但 JSON 的质量因素小于 XML 的质量因素，该如何处理请求方法呢？答案是内容协商的结果按照客户端的喜好选择响应实体的数据类型，即选择 XML 格式。

其测试代码如下所示，客户端明确表述格式 XML 优于 JSON。虽然服务器端定义的资源方法中 JSON 的质量因素高，但 Jersey 会根据客户端的喜好，选择了 XML 格式作为表述的格式返回。

```
WebTarget path = target("conneg-resource").path("book").path("123");
Builder request = path.request();
request.header("Accept", "application/xml;q=0.7,application/json;q=0.2");
Book book = request.get(Book.class);
1 > GET http://localhost:9998/conneg-resource/book/123
1 > Accept: application/xml;q=0.7,application/json;q=0.2
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book bookId="123"/>
```

3.6.2 @Consumes 注解

注解 @Consumes 用于定义方法的请求实体的数据类型。和 @Produces 不同的是，@Consumes 的数据类型的定义只用于 JAX-RS 2.0 匹配请求处理的方法，不做内容协商使用。如果匹配不到，那么服务器会返回 HTTP 状态码 415（Unsupported Media Type）。其示例代码如下。

```
@POST
// 关注点 1: @Consumes 注解定义了 XML 和 JSON 两种格式
@Consumes({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
@Produces(MediaType.APPLICATION_XML)
public Book getEntity(Book book) {
  LOGGER.debug(book.getBookName());
  return book;
}
final Builder request = target(path).request();
// 关注点 2
final Book result = request.post(
Entity.entity(book, MediaType.APPLICATION_XML), Book.class);
```

在这段代码中，getEntity() 方法定义了 @Consumes 媒体类型为 XML 格式和 JSON 格式，

见关注点 1。那么，在客户端请求中，如果请求实体的数据类型定义为两者之一，那么该方法会被选择为处理请求的方法，否则查找是否有定义为相应数据类型的方法，如果没有抛出 javax.ws.rs.NotSupportedException 异常，则使用该方法处理请求，见关注点 2。

3.7 本章小结

本章是 REST 理论和 Jersey 实践的核心章节，详细讲述了 HTTP 方法与 REST API 的统一接口设计、URI 的 REST 风格设计，并逐个讲述了 JAX-RS 2.0 定义的注解如何支持资源定位，还讲述了 Jersey 对各种表述类型的支持和实现、Jersey 对 REST 连通性的两种实现、REST 资源方法对响应的处理，以及 Jersey 对内容协商的支持和实现。接下来，我们走进 JAX-RS 定义的各种 Providers，更深入地了解过滤、拦截等功能的实现。

第 4 章 Chapter 4

REST 请求处理

设计良好的 REST API 除了要符合关于统一接口和资源定位等要求，还要详细考虑通用的请求处理过程中，每个步骤的特殊处理，并设计出符合业务规范的处理流程。本章将深入 REST 请求处理过程中的扩展点，并讲述如何对其实现。

阅读指南

本章示例所在目录是 jax-rs2-guide\sample\4\jaxrs2-handle。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/4/jaxrs2-handle。

4.1 REST 和 AOP

AOP 对增强 REST 式的 Web 服务的功能性、安全性和可扩展性等方面都具有深远意义，因此，完整的 REST 风格的框架都从容器级别支持 AOP 式的开发。Jersey 可以在不依赖于 Spring 等 AOP 支持框架的情况下，天然地支持 AOP。

小白讲堂

AOP（Aspect Oriented Programming，面向切面编程）的典型应用场景有权限管理、日志记录、统计记录、事务以及异常处理等。其实现原理是代理被调用的方法，在其被执行的方法前后，增加额外业务功能。AOP 的实现机制是通过注解或者 XML 配置，依据这些配置，动态生成字节码（bytecode），使调用代码对应的字节码被环绕注入新的功能；或者使用 Java 的动态代理机制，完成对被调用方法的增强。

Jersey 的 AOP 功能来自于 GlassFish 项目集的 HK2 项目（参见 1.4 节）。Jersey 通用包 jersey-common 依赖 HK2（轻量级 DI 架构），包括 hk2-api 和 hk2-locator。

其中 hk2-locator 依赖于 javax.inject 包、asm-all-repackaged 包和 cglib 包。从这些包名不难看出，hk2-locator 是个致力于 AOP 方向的包。javax.inject 包出自 Java 依赖注入规范（JSR-330），cglib 是 Spring 用户熟识的动态代码生成工具，是 Spring 两种 AOP 实现方式的一种，其底层依赖于 ASM。ASM 来自于开源软件国际联盟 OW2（http://www.ow2.org），OW2 的前身是来自法国的 ObjectWeb 和来自中国的 OrientWare 两个中间件开源组织。

Jersey 提供的 REST 过滤器和拦截器为开发者提供了很贴心的切面扩展点，开发者无须像在 Spring 中为了针对某个类的方法进行 AOP 扩展而写配置文件。在 Jersey 中，只要实现相应扩展点的接口，即可实现 REST 请求流程中特定事件点的拦截、扩展，其他工作由底层的 HK2 帮我们做。典型的应用包括请求和响应的过滤和读写拦截。

4.2 Providers 详解

javax.ws.rs.ext.Providers 是 JAX-RS 2.0 定义的一种辅助接口，其实现类用于辅助 REST 框架完成过滤和读写拦截等功能。使用注解 @Provider 来标注这些实现类，可以被 JAX-RS 2.0 的运行时自动探测、加载。Provider 实例可以通过 @Context 注解被依赖注入到其他实例中。Providers 接口定义了 4 个方法，分别用来获取 MessageBodyReader、MessageBodyWriter、ExceptionMapper 和 ContextResolver 实例。

4.2.1 实体 Providers

在 3.3 节我们讲述了 Jersey 支持的各种传输格式。Jersey 之所以可以支持那么多种表述的类型，即响应实体的传输格式，是因为其底层实体 Providers 具备的对不同格式的处理能力。如图 4-1 所示，Jersey 内部提供了非常丰富的 MessageBodyReader 接口和 MessageBodyWriter 接口实现类，用于处理不同格式的表述，比如字节数组、XML、文件和流等。本节将介绍读写实体的工具类 MessageBodyReader 和 MessageBodyWriter，并为我们业务所用。

1. MessageBodyReader

消息体读处理器接口 MessageBodyReader<T> 用于将传输流转换为 Java 类型的对象。MessageBodyReader 接口定义了一个泛型，接口的实现类为这个泛型定义一个具体类型，该类型即是该实现类所支持的转换类型。实现类被业务系统启用有两种方式，一种是使用注解 @Provider 定义实现类，业务系统在启动时自动探测并加载，另一种是通过编码注册到 Application 类或其子类中，业务系统在启动时，加载 Application 类或其子类时一并加载。

（1）isReadable

MessageBodyReader<T> 接口定义了两个方法。第一个方法 isReadable() 用来判断实现类

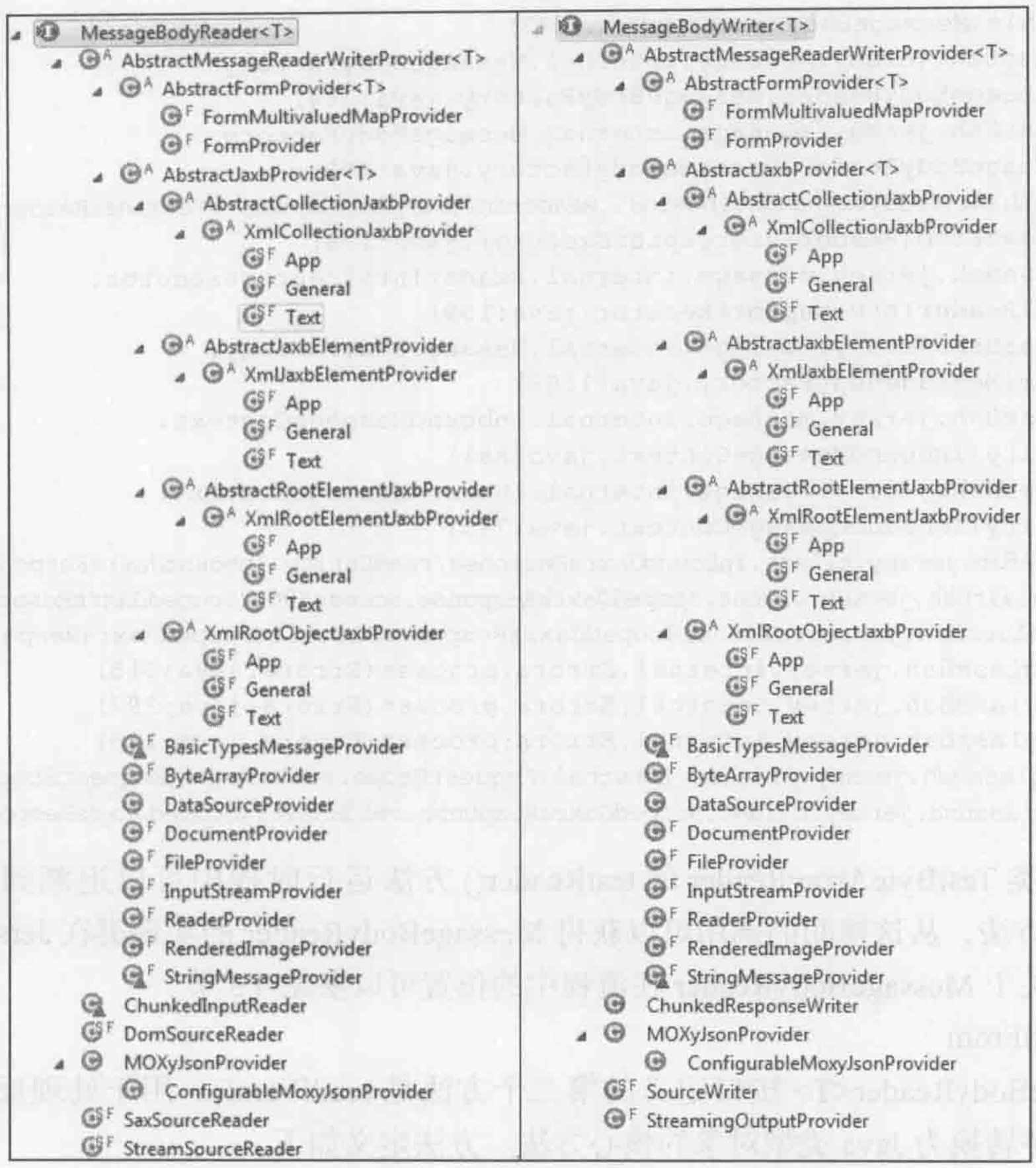


图 4-1 MessageBodyReader 和 MessageBodyWriter 的实现类

是否支持将当前请求的数据类型反序列化。以读取字节数组实体实现类 ByteArrayProvider 为例，其覆盖方法会判断当前类型是否是字节数组类型，示例代码如下所示。

```
//isReadable 接口方法
public boolean isReadable(Class<?> type, Type genericType,Annotation[]
annotations, MediaType mediaType);
//isReadable 方法的实现
@Override
public boolean isReadable(Class<?> type, Type genericType, Annotation[]
annotations, MediaType mediaType) {
    return type == byte[].class;
}
// 测试方法运行时栈
org.glassfish.jersey.message.internal.ByteArrayProvider.
isReadable(ByteArrayProvider.java:66)
org.glassfish.jersey.message.internal.MessageBodyFactory$MbrModel.
isReadable(MessageBodyFactory.java:250)
org.glassfish.jersey.message.internal.MessageBodyFactory$MbrModel.
```

```
isReadable(MessageBodyFactory.java:243)
org.glassfish.jersey.message.internal.MessageBodyFactory
._getMessageBodyReader(MessageBodyFactory.java:744)
org.glassfish.jersey.message.internal.MessageBodyFactory
.getMessageBodyReader(MessageBodyFactory.java:659)
org.glassfish.jersey.message.internal.ReaderInterceptorExecutor$TerminalReaderInterceptor.
aroundReadFrom(ReaderInterceptorExecutor.java:194)
org.glassfish.jersey.message.internal.ReaderInterceptorExecutor.
proceed(ReaderInterceptorExecutor.java:139)
org.glassfish.jersey.message.internal.MessageBodyFactory.
readFrom(MessageBodyFactory.java:1109)
org.glassfish.jersey.message.internal.InboundMessageContext.
readEntity(InboundMessageContext.java:851)
org.glassfish.jersey.message.internal.InboundMessageContext.
readEntity(InboundMessageContext.java:785)
org.glassfish.jersey.client.InboundJaxrsResponse.readEntity(InboundJaxrsResponse.java:96)
   org.glassfish.jersey.client.ScopedJaxrsResponse.access$001(ScopedJaxrsResponse.java:56)
   org.glassfish.jersey.client.ScopedJaxrsResponse$1.call(ScopedJaxrsResponse.java:77)
   org.glassfish.jersey.internal.Errors.process(Errors.java:315)
   org.glassfish.jersey.internal.Errors.process(Errors.java:297)
   org.glassfish.jersey.internal.Errors.process(Errors.java:228)
   org.glassfish.jersey.process.internal.RequestScope.runInScope(RequestScope.java:397)
   org.glassfish.jersey.client.ScopedJaxrsResponse.readEntity(ScopedJaxrsResponse.java:74)
```

从测试类 TestByteArrayReader 的 testReader() 方法运行时栈中可以追溯到 Response 的 readEntity() 方法，从这期间的调用可以获得 MessageBodyReader 的实现类在 Jersey 架构体系中的位置。关于 MessageBodyReader 在流程中的位置可以参见 4.3 节。

（2）readFrom

MessageBodyReader<T> 接口定义的第二个方法是 readFrom()，用于处理反序列化，是处理读取流并转换为 Java 类型对象的核心方法。方法定义如下。

```
public T readFrom(Class<T> type, Type genericType,Annotation[] annotations,
MediaType mediaType,MultivaluedMap<String, String> httpHeaders,
InputStream entityStream) throws java.io.IOException, javax.ws.rs.WebApplicationException;
```

继续读取字节数组实体实现类 ByteArrayProvider 的研究，该方法的覆写包含两个内容。第一是将实体输入流写入字节数组输出流，第二是将该流以字节数组的形式返回。运行单元测试类 TestByteArrayReader，并在 ByteArrayProvider 类中设置断点，可以监控到具体的行为，可参考图 4-2。

2. MessageBodyWriter

消息体写处理器接口 MessageBodyWriter<T> 用于将 Java 类型的对象转换为流，是序列化的过程，是 MessageBodyReader<T> 接口实现的反序列化的逆过程。两个接口的设计原理是相同的。对应的，MessageBodyWriter<T> 定义了两个方法，即 isWriteable() 和 writeTo()。其实，解析一种传输类型的 Provider 类通常会同时实现 MessageBodyReader 和 MessageBodyWriter 这两个接口，比如上面提及的 ByteArrayProvider 类。

为了更全面地掌握实体 Provider 实现类序列化过程的分析，我们选用较为复杂的 MOXyJsonProvider 类做例子。MOXyJsonProvider 类是第 3 章讲述的 4 种 JSON 支持技术中 MOXy 的支持类，来自 EclipseLink 项目。MOXyJsonProvider 类实现了上述的两个接口，并定义了生产和消费的媒体类型。

```
@Produces({"application/octet-stream", "*/*"})
@Consumes({"application/octet-stream", "*/*"})
public final class ByteArrayProvider extends AbstractMessageReaderWriterProvider<byte[]> {

    @Override
    public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations, MediaType mediaType) {
        return type == byte[].class;
    }

    @Override
    public byte[] readFrom(
            Class<byte[]> type,
            Type genericType,
            Annotation annotations[],
            MediaType mediaType,
            MultivaluedMap<String, String> httpHeaders,
            InputStream entityStream) throws IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        writeTo(entityStream, out);
        return out.toByteArray();
    }
Variables
  entityStream = {org.glassfish.jersey.message.internal.EntityInputStream@2819}
  out = {java.io.ByteArrayOutputStream@2820}"Hello"
    buf = {byte[32]@3330}
    count = 5
```

图 4-2　readFrom() 方法示例

（1）isWriteable

isWriteable() 方法用于检测实现类是否支持序列化当前请求的类型，如果不可写，Jersey 框架会放弃使用这个实现类来处理当前的表述。在分析 ByteArrayProvider 类的可读实现时，该方法只使用到了第一个参数 type，而在 MOXyJsonProvider 类的可写检测的覆写中，同时用到了最后一个参数 mediaType 来校验请求的媒体类型是否是 JSON 类型。示例代码如下所示。

```
public boolean isWriteable(Class<?> type,
Type genericType,Annotation[] annotations, MediaType mediaType);
package org.eclipse.persistence.jaxb.rs
@Produces({MediaType.APPLICATION_JSON, MediaType.WILDCARD, "application/x-javascript"})
@Consumes({MediaType.APPLICATION_JSON, MediaType.WILDCARD})
public class MOXyJsonProvider
implements MessageBodyReader<Object>, MessageBodyWriter<Object>{
public boolean isWriteable(Class<?> type,
Type genericType, Annotation[] annotations, MediaType mediaType) {
//关注点 1: 对 type 的校验
  if(type == JSONWithPadding.class && APPLICATION_XJAVASCRIPT
   .equals(mediaType.toString())) {
```

```
        return true;
    }
    //媒体类型不是JSON类型，直接返回false
    if(!supportsMediaType(mediaType)) {
        return false;//类型是字节数组或者字符串类型，直接返回false
    } else if(CoreClassConstants.APBYTE == type || CoreClassConstants.STRING == type) {
        return false;
    } else if(File.class.isAssignableFrom(type)) {
        return false;
    } else if(DataSource.class.isAssignableFrom(type)) {
        return false;
    } else if(StreamingOutput.class.isAssignableFrom(type)) {
        return false;
    } else if(Object.class == type) {
        return false;
    } else if(JAXBElement.class.isAssignableFrom(type)) {
        Class domainClass = getDomainClass(genericType);
        return isWriteable(domainClass, null, annotations, mediaType)
        || domainClass == String.class;
    } else if(Collection.class.isAssignableFrom(type)) {
        Class domainClass = getDomainClass(genericType);
        return isWriteable(domainClass, null, annotations, mediaType)
        || domainClass == String.class;
    } else {
        return true;
    }
}
```

在这段代码中，在关注点 1 处，MOXyJsonProvider 类对第一个参数 type 类型的校验条件繁多，从字节数组到 Collection 类的子类，只要匹配就返回 false，也就是说，MOXyJsonProvider 类不负责对这些类型进行序列化操作。

（2）writeTo

writeTo() 方法是将请求对象写入流的序列化过程。MOXyJsonProvider 类的 writeTo 方法的覆写内容有 30 多行，基本都是对 Marshaller 实例的配置，关键的一行是执行 Marshaller 实例的 marshal 一行，示例代码如下。

```
//writeTo接口方法:
public void writeTo(T t, Class<?> type, Type genericType, Annotation[] annotations,
MediaType mediaType,MultivaluedMap<String, Object> httpHeaders,
OutputStream entityStream)throws java.io.IOException, javax.ws.rs.WebApplicationException;
//MOXyJsonProvider类的writeTo方法:
...
marshaller.marshal(object, entityStream);
```

其对应的测试栈信息如图 4-3 所示。

如图 4-3 所示，从测试类 TestMoxyWriter 的 testMoxyWriter 方法的栈信息中，可以看到 Book 实例被序列化到 CommittingOutputStream 流实例中，实现了写入的过程。

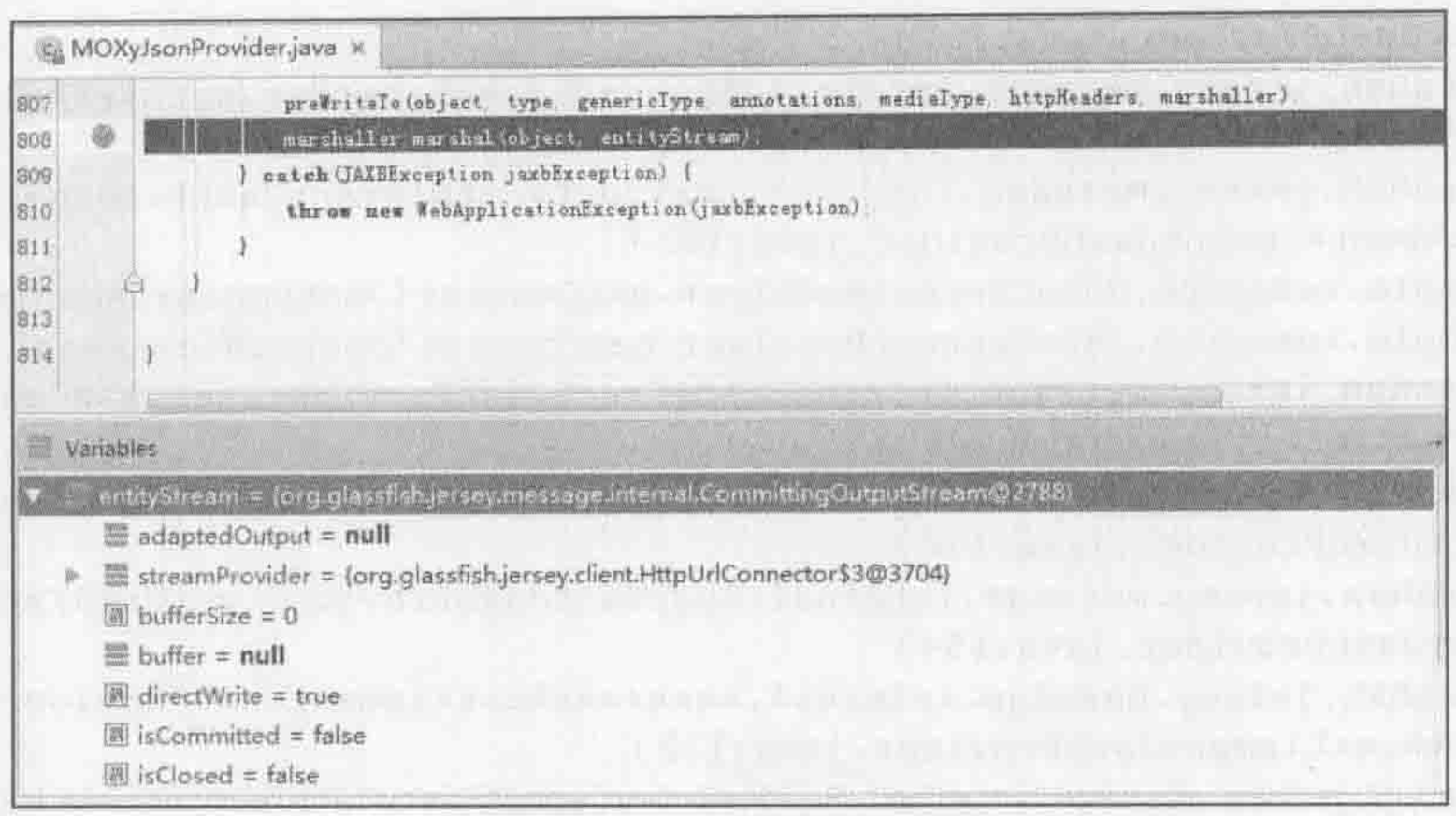


图 4-3　writeTo 方法 () 示例

3. MessageBodyWorkers

从图 4-1 所示的 MessageBodyReader 和 MessageBodyWriter 的实现类示意图中可以看出，实体读写接口的实现类非常多，选择哪个实现类作为当前请求的读写处理器的算法是非常繁重的工作，MessageBodyWorkers 接口旨在抽象这一遴选工作，其实现类可以通过 @Context 依赖注入到使用 MessageBodyWorkers 的类中。MessageBodyFactory 是 MessageBodyWorkers 接口的实现类。在上述的 isReadable 示例栈中，可以发现它的“身影”。

宅人坑事

实体读写接口定义在 JAX-RS 2.0 的 javax.ws.rs.ext 包中，MessageBodyWorkers 定义在 Jersey 核心包 jersey-common 的 org.glassfish.jersey.message 包中，前者是规范定义的接口，后者是参考实现定义的接口，不要混淆。

4.2.2　上下文 Providers

除了处理实体的 Provider，处理上下文的 Provider 也非常重要。ContextResolver<T> 接口即是用于提供资源类和其他 Provider 上下文信息的接口。ContextResolver<T> 定义了一个方法 getContext()，输入参数是表述对象的类型，输出是上下文泛型。

我们来回顾一下 3.3 节讲述的 JsonContextResolver 类，其是 ContextResolver 的实现类，根据不同的表述类型，提供不同的 JAXB 上下文类 JettisonJaxbContext 实例。栈信息如下。

```
T getContext(Class<?> type);
com.example.resource.JsonContextResolver.getContext(JsonContextResolver.java:26)
com.example.resource.JsonContextResolver.getContext(JsonContextResolver.java:12)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getJAXBContext
(AbstractJaxbProvider.java:244)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getMarshaller
```

```
(AbstractJaxbProvider.java:217)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getMarshaller
(AbstractJaxbProvider.java:184)
org.glassfish.jersey.message.internal.AbstractRootElementJaxbProvider.writeTo
(AbstractRootElementJaxbProvider.java:162)
com.example.resource.JsonContextResolver.getContext(JsonContextResolver.java:26)
com.example.resource.JsonContextResolver.getContext(JsonContextResolver.java:12)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getJAXBContext
(AbstractJaxbProvider.java:244)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getUnmarshaller
(AbstractJaxbProvider.java:178)
org.glassfish.jersey.message.internal.AbstractJaxbProvider.getUnmarshaller
(AbstractJaxbProvider.java:154)
org.glassfish.jersey.message.internal.AbstractRootElementJaxbProvider.readFrom
(AbstractRootElementJaxbProvider.java:122)
org.glassfish.jersey.message.internal.ReaderInterceptorExecutor$TerminalReaderInterceptor.
invokeReadFrom(ReaderInterceptorExecutor.java:239)
org.glassfish.jersey.message.internal.ReaderInterceptorExecutor$TerminalReaderInterceptor.
aroundReadFrom(ReaderInterceptorExecutor.java:211)
org.glassfish.jersey.message.internal.ReaderInterceptorExecutor.proceed
(ReaderInterceptorExecutor.java:139)
org.glassfish.jersey.message.internal.MessageBodyFactory.readFrom
(MessageBodyFactory.java:1109)
```

从栈的输出中可以观察到序列化过程，从 writeTo 到 getContext，以及在这一过程中，从 readFrom 到 getContext 的栈信息。

除了上述两种类型的 Provider，JAX-RS 2.0 还定义了处理异常的 Provider，可参考 3.5.2 节。

4.3 REST 请求流程

在 REST 请求处理的扩展点上，我们已经讲述了实体处理的 Provider 接口，以及上下文处理和异常处理。本章还将讲述两种在面向切面编程中非常重要的、特殊的 Provider：过滤器（4.4 节）和拦截器（4.5 节）。在进入这个主题之前，我们需要对 REST 请求处理的流程这条线有明确的认识，这样一来，才会知道这些点都处于流程的什么位置。只有这样才能清楚地实现对扩展点的开发和调试。

如图 4-4 所示，请求流程中存在 3 种角色，分别是用户、REST 客户端和 REST 服务器。请求始于请求的发送，止于调用 Response 类的 readEntity() 方法，获取响应实体。

1）用户提交请求数据，客户端接收请求，进入第一个扩展点："客户端请求过滤器 ClientRequestFilter 实现类"的 filter() 方法。

2）请求过滤处理完毕后，流程进入第二个扩展点："客户端写拦截器 WriterInterceptor 实现类"的 aroundWriteTo() 方法，实现对客户端序列化操作的拦截。

3）"客户端消息体写处理器 MessageBodyWriter"执行序列化，流程从客户端过渡到服务器端。

4）服务器接收请求，流程进入第三个扩展点："服务器前置请求过滤器 ContainerRequestFilter 实现类"的 filter() 方法。

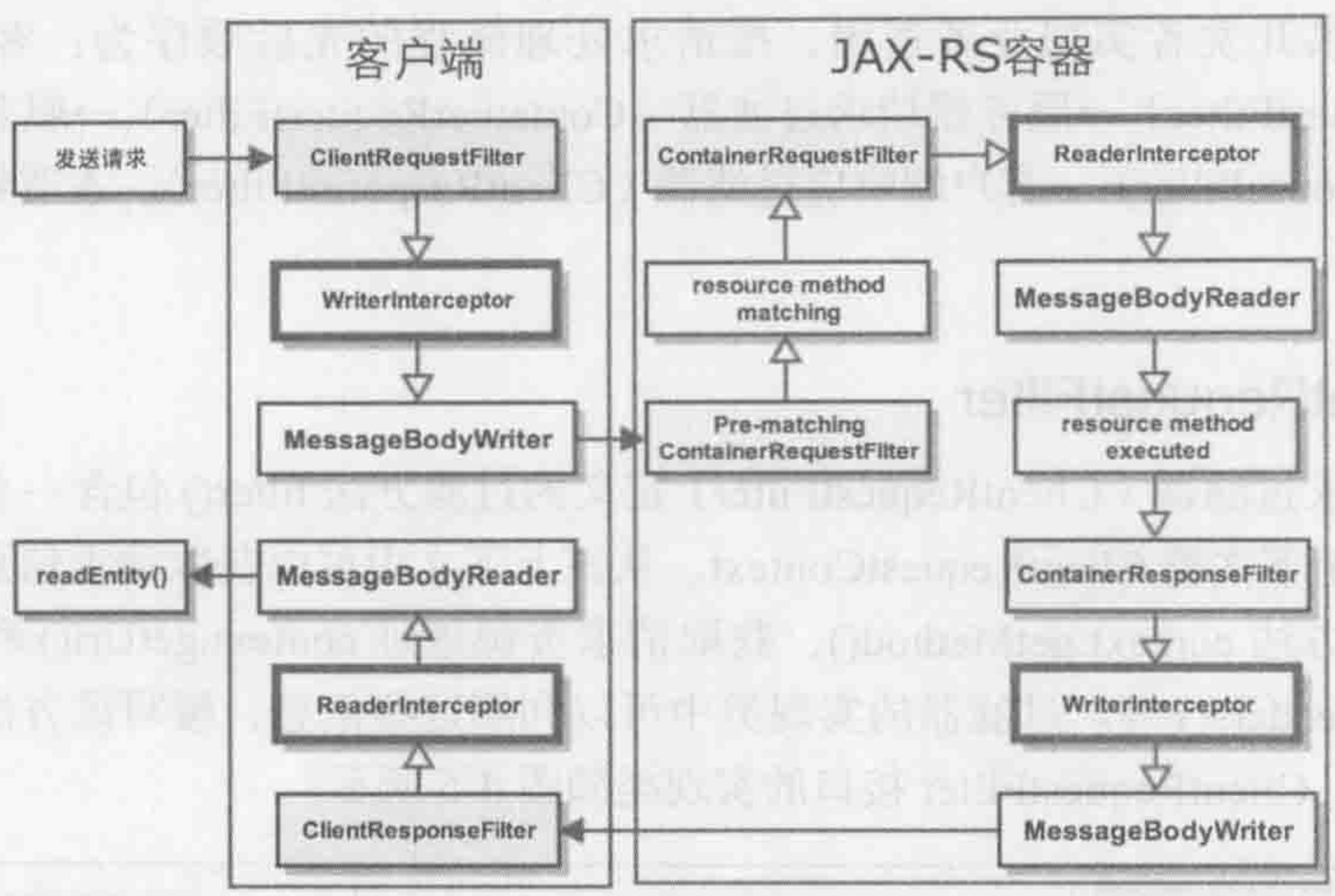


图 4-4　Jersey 的 REST 请求处理流程

5）过滤处理完毕后，服务器根据请求匹配资源方法，如果匹配到相应的资源方法，流程进入第四个扩展点："服务器后置请求过滤器 ContainerRequestFilter 实现类"的 filter() 方法。

6）后置请求过滤处理完毕后，流程进入第五个扩展点："服务器读拦截器 ReaderInterceptor 实现类"的 aroundReadFrom() 方法，拦截服务器端反序列化操作。

7）"服务器消息体读处理器 MessageBodyReader"完成对客户端数据流的反序列化。服务器执行匹配的资源方法。

8）REST 请求资源的处理完毕后，流程进入第六个扩展点："服务器响应过滤器 ContainerResponseFilter 实现类"的 filter() 方法。

9）过滤处理完毕后，流程进入第七个扩展点："服务器写拦截器 WriterInterceptor 实现类"的 aroundWriteTo() 方法，实现对服务器端序列化到客户端这个操作的拦截。

10）"服务器消息体写处理器 MessageBodyWriter"执行序列化，流程返回到客户端一侧。

11）客户端接收响应，流程进入第八个扩展点："客户端响应过滤器 ClientResponseFilter 实现类"的 filter() 方法。

12）过滤处理完毕后，客户端响应实例 response 返回到用户一侧，用户执行 response.readEntity()，流程进入第九个扩展点："客户端读拦截器 ReaderInterceptor 实现类"的 aroundReadFrom() 方法，对客户端反序列化进行拦截。

13）"客户端消息体读处理器 MessageBodyReader"执行反序列化，将 Java 类型的对象最终作为 readEntity() 方法的返回值。

到此，一次 REST 请求处理的完整流程完毕。在这个期间，如果出现异常或资源不匹配等情况，则会从出错点结束流程。

4.4 REST 过滤器

从上一节的流程讲述中，我们了解 JAX-RS 2.0 定义的 4 种过滤器扩展点（extension point）接口，供开发者实现业务逻辑，按请求处理流程的先后顺序为：客户端请求过滤器（ClientRequestFilter）→服务器请求过滤器（ContainerRequestFilter）→服务器响应过滤器（ContainerResponseFilter）→客户端响应过滤器（ClientResponseFilter）。本节将全面讲述 4 种过滤器的使用。

4.4.1 ClientRequestFilter

客户端请求过滤器（ClientRequestFilter）定义的过滤方法 filter() 包含一个输入参数，是客户端请求的上下文类 ClientRequestContext。从该上下文中可以获取请求信息，典型的示例包括获取请求方法 context.getMethod()，获取请求资源地址 context.getUri() 和获取请求头信息 context.getHeaders() 等。过滤器的实现类中可以利用这些信息，覆写该方法以实现该类特有的过滤功能。ClientRequestFilter 接口的实现类如图 4-5 所示。

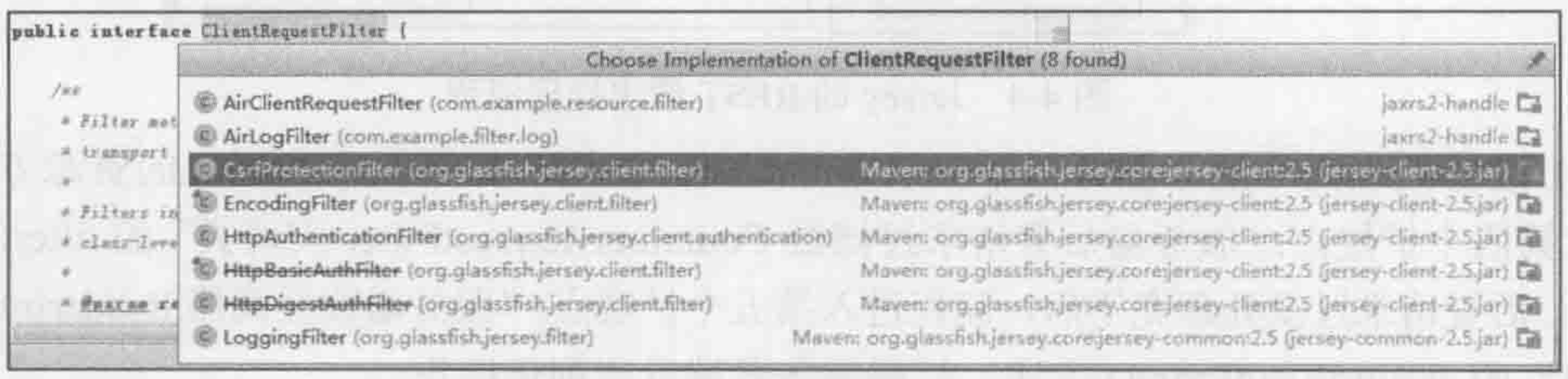


图 4-5 ClientRequestFilter 接口的实现类

图 4-5 展示了 ClientRequestFilter 接口的实现类，包括 Jersey 内部提供的实现类和本书示例代码中的实现类。我们选择 HTTP 基本认证过滤器类 HttpBasicAuthFilter 作为例子，来感受上面的讲述（HTTP 基本认证这个知识点可参考 6.1 节）。示例代码如下所示。

```
@Override
public void filter(ClientRequestContext rc) throws IOException {
  if(!rc.getHeaders().containsKey(HttpHeaders.AUTHORIZATION)) {
      rc.getHeaders().add(HttpHeaders.AUTHORIZATION, authentication);
  }
}
```

在这段代码中，HTTP 基本认证过滤器类在 filter() 方法中，判断请求头信息中是否包含“Authorization”，如果不包含，则添加请求头“Authorization”为 authentication，authentication 的内容是“Basic”+ Base64.encodeAsString（usernamePassword）。这样以来，经过 HTTP 基本认证过滤器类过滤处理后，可以确保请求头信息中包含“Authorization”。

```
@Override
public void filter(ClientRequestContext rc) throws IOException {
  if(!rc.getHeaders().containsKey(HttpHeaders.AUTHORIZATION)) {
      rc.getHeaders().add(HttpHeaders.AUTHORIZATION, authentication);
```

```
    }
}
```

4.4.2 ContainerRequestFilter

针对过滤切面，服务器请求过滤器接口 ContainerRequestFilter 的实现类可以定义为预处理和后处理。默认情况下，采用后处理方式，即先执行容器接收请求操作，当服务器接收并处理请求后，流程才进入过滤器实现类的 filter() 方法。而预处理是在服务器处理接收到的请求之前就执行过滤。如果希望实现一个预处理的过滤器实现类，需要在类名上定义注解 @PreMatching。

服务器请求过滤器定义的过滤方法 filter() 包含一个输入参数，即容器请求上下文类 ContainerRequestContext。ContainerRequestFilter 接口的实现类如图 4-6 所示。

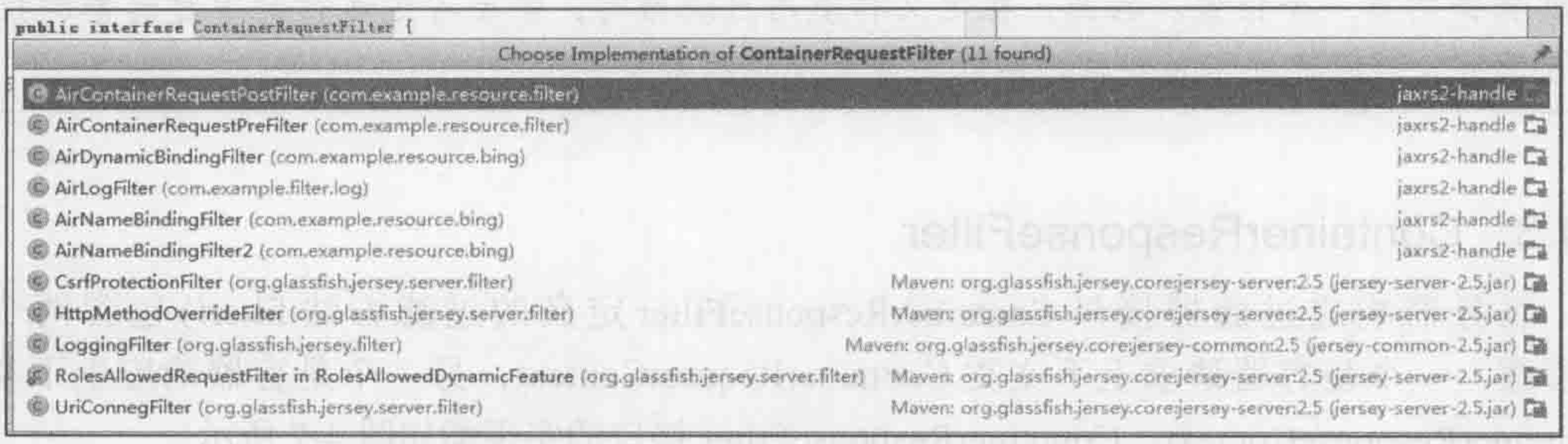


图 4-6 ContainerRequestFilter 接口的实现类

图 4-6 展示了 ContainerRequestFilter 接口的实现类，我们以 CsrfProtectionFilter 为例来说明（CSRF 这个知识点可参考 6.5 节）。示例代码如下所示。

```
@Priority(Priorities.AUTHENTICATION)
public class CsrfProtectionFilter implements ContainerRequestFilter {
public static final String HEADER_NAME = "X-Requested-By";
//关注点 1: 忽略方法集合
private static final Set<String> METHODS_TO_IGNORE;
static {
        HashSet<String> mti = new HashSet<String>();
        mti.add("GET");
        mti.add("OPTIONS");
        mti.add("HEAD");
        METHODS_TO_IGNORE = Collections.unmodifiableSet(mti);
    }
    @Override
    public void filter(ContainerRequestContext rc) throws IOException {
    //关注点 2: 判断方法名称是否符合条件
        if(!METHODS_TO_IGNORE.contains(rc.getMethod())
            && !rc.getHeaders().containsKey(HEADER_NAME)) {
            throw new BadRequestException();
        }
    }
```

在这段代码中，CsrfProtectionFilter 定义了一个特殊的头信息“X-Requested-By”和 CSRF 忽略监控的方法集合，见关注点 1。在过滤器的 filter() 方法中，首先从上下文中获取头信息（使用 rc.getHeaders()）和请求方法信息（使用 rc.getMethod()），然后判断头信息是否包含“X-Requested-By”，方法信息是否是安全的请求方法，即“GET”、“OPTIONS”或“HEAD”。如果两个条件都不成立，那么过滤器会抛出一个运行时异常 BadRequestException，见关注点 2。

通过 CsrfProtectionFilter 过滤器，可以确保请求是 CSRF 安全的。

宅人坑事

CsrfProtectionFilter 类使用了注解 @Priority（Priorities.AUTHENTICATION）来定义该类，明确了该过滤器具有最高的优先级。同时，以注解的文字告诉开发者，需要将其放在过滤器链的第一个位置。因此，在定义和使用过滤器时，需要考虑运行时中过滤器的执行先后顺序，否则无法实现过滤器的功能或者使流程混乱。

4.4.3 ContainerResponseFilter

服务器响应过滤器接口 ContainerResponseFilter 定义的过滤方法 filter() 包含两个输入参数，一个是容器请求上下文类 ContainerRequestContext，另一个是容器响应上下文类 ContainerResponseContext。ContainerResponseFilter 接口的实现类如图 4-7 所示。

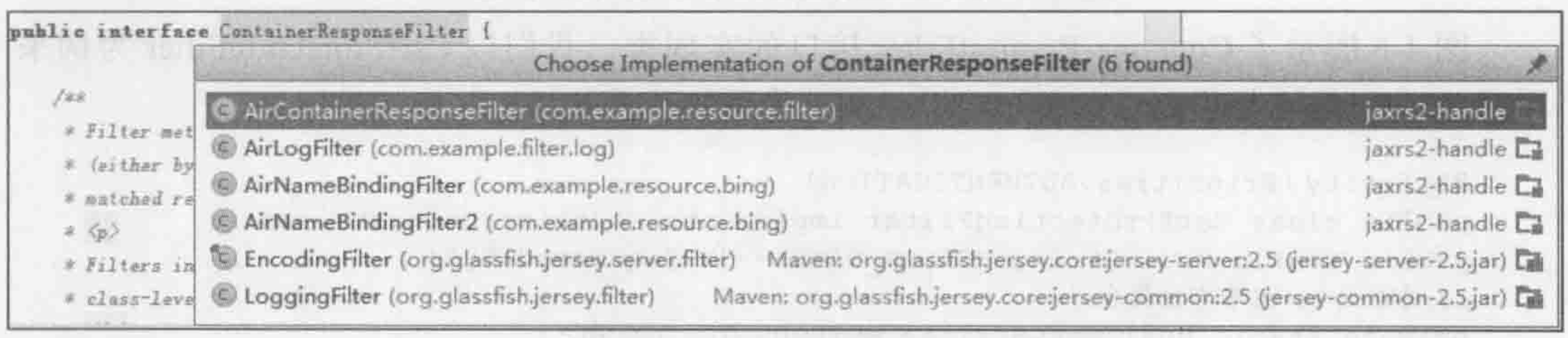


图 4-7 ContainerResponseFilter 接口的实现类

图 4-8 展示了 ContainerResponseFilter 接口的实现类，我们以 EncodingFilter 为例来说明。该过滤器的作用是完成内容协商中编码匹配的工作（内容协商这个知识点可参考 3.6 节），示例代码如下所示。

```
@Priority(Priorities.HEADER_DECORATOR)
public final class EncodingFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext request,
    ContainerResponseContext response) throws IOException {
...
        List<String> varyHeader =
        ((ContainerResponse) response).getStringHeaders().get(HttpHeaders.VARY);
        //关注点 1: Vary 头信息
        if (varyHeader == null || !varyHeader.contains(HttpHeaders.ACCEPT_ENCODING)) {
```

```
        response.getHeaders().add(HttpHeaders.VARY, HttpHeaders.ACCEPT_ENCODING);
    }
    if (response.getHeaders().getFirst(HttpHeaders.CONTENT_ENCODING) != null) {
        return;
    }
    List<String> acceptEncoding = request.getHeaders()
    .get(HttpHeaders.ACCEPT_ENCODING);
    ...
    // 关注点 1: Content-Encoding 头信息
    if (!IDENTITY_ENCODING.equals(contentEncoding)) {
            response.getHeaders().putSingle(HttpHeaders.CONTENT_ENCODING,contentEncoding);
        }
    }
```

EncodingFilter 过滤器的 filter() 方法通过对请求头信息“Accept-Encoding”的分析，先后为响应头信息“Vary”和“Content-Encoding”赋值，以实现编码部分的内容协商，见关注点 1。

4.4.4 ClientResponseFilter

客户端响应过滤器 ClientResponseFilter 定义的过滤方法 filter() 包含两个输入参数，一个是客户端请求的上下文类 ClientRequestContext，另一个是客户端响应的上下文类 ClientResponseContext。ClientResponseFilter 接口的实现类如图 4-8 所示。

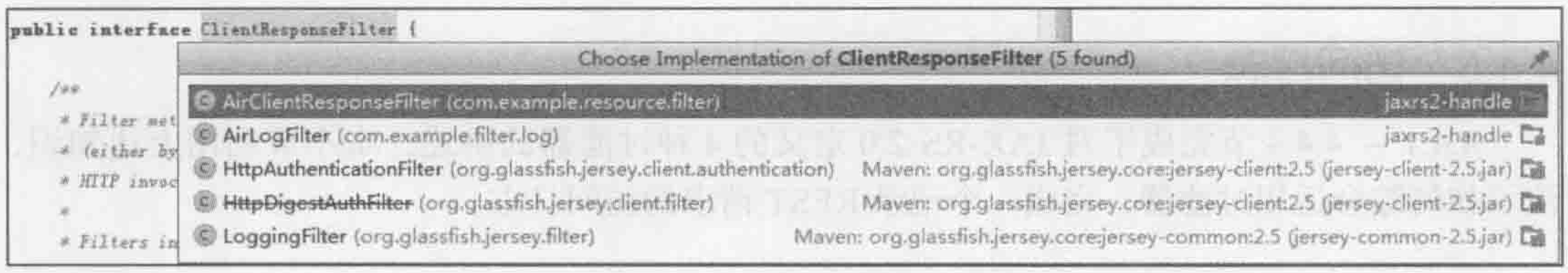


图 4-8 ClientResponseFilter 接口的实现类

图 4-8 展示了 ClientResponseFilter 接口的实现类，包括 Jersey 内部提供的实现类和本书示例代码中的实现类。我们以 HTTP 摘要认证过滤器类 HttpDigestAuthFilter 为例进行演示（HTTP 摘要认证这个知识点可参考 6.1 节）。示例代码如下所示。

```
@Override
public void filter(ClientRequestContext requestContext, ClientResponseContext
responseContext) throws IOException {
...
  if (Response.Status.fromStatusCode(responseContext.getStatus())
  == Status.UNAUTHORIZED) {
...
        Client client = requestContext.getClient();
        String method = requestContext.getMethod();
        MediaType mediaType = requestContext.getMediaType();
        URI lUri = requestContext.getUri();
        WebTarget resourceTarget = client.target(lUri);
        Invocation.Builder builder = resourceTarget.request(mediaType);
```

```
        builder.headers(requestContext.getHeaders());
        //关注点 1: 增加 "WWW-Authenticate" 头信息
        builder.header(HEADER_DIGEST_SCHEME, digestScheme);
        Invocation invocation = builder.build(method);
        //关注点 2: 自动重新发送一次携带 "WWW-Authenticate" 头信息的请求
        Response nextResponse = invocation.invoke();
        if (nextResponse == null) {
            return;
        }
        if (nextResponse.hasEntity()) {
            responseContext.setEntityStream(nextResponse.readEntity(InputStream.class));
        }
        MultivaluedMap<String, String> headers = responseContext.getHeaders();
        headers.clear();
        headers.putAll(nextResponse.getStringHeaders());
        responseContext.setStatus(nextResponse.getStatus());
        }
    }
```

在这段代码中，HttpDigestAuthFilter 的 filter() 方法校验响应信息，如果服务器的响应信息中，HTTP 状态码是 UNAUTHORIZED（401，“Unauthorized”），该过滤器会尝试添加摘要认证的头信息“WWW-Authenticate”，见关注点 1。如果“WWW-Authenticate”信息存在，则会自动重新发送一次携带该信息的请求，见关注点 2。这样以来，经过 HTTP 摘要认证过滤器类过滤处理后，如果存在“WWW-Authenticate”头信息，可以简化手动交互的过程。

4.4.5 访问日志

4.4.1 ~ 4.4.4 节完成了对 JAX-RS 2.0 定义的 4 种过滤器的讲述，本小节利用上述知识，演示如何综合运用过滤器，完成一个记录 REST 请求的访问日志。

1. 访问日志实现类

访问日志类 AirLogFilter 实现了上述的 4 种过滤器，旨在记录服务器和客户端的请求和响应中的运行时信息。AirLogFilter 类定义如下所示。

```
@PreMatching
public class AirLogFilter implements ContainerRequestFilter, ClientRequestFilter,
ContainerResponseFilter, ClientResponseFilter {
...
}
```

AirLogFilter 为每一种过滤器接口定义的 filter() 方法提供了实现。在客户端请求过滤中，输出请求资源地址信息和请求头信息；在容器请求过滤中，输出请求方法、请求资源地址信息和请求头信息；在容器响应过滤中，输出 HTTP 状态码和请求头信息；在客户端响应过滤中，输出 HTTP 状态码和请求头信息。4 个阶段的 filter 示例代码如下所示。

```
@Override
public void filter(ClientRequestContext context) throws IOException {
```

```
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    //关注点 1: 获取请求方法和地址
    printRequestLine(CLIENT_REQUEST, b, id, context.getMethod(), context.getUri());
    //关注点 2: 获取请求头信息
    printPrefixedHeaders(CLIENT_REQUEST, b, id,
    HeadersFactory.asStringHeaders(context.getHeaders()));
    LOGGER.info(b.toString());
  }
```

在这段代码中，Filter 类实现了客户端请求过滤。从客户端请求上下文实例中，可以获取到请求方法和请求地址信息，见关注点 1。同样，头信息也可以从中获取，见关注点 2。

```
  @Override
  public void filter(ClientRequestContext requestContext,
  ClientResponseContext responseContext) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    //关注点 1: 获取响应状态
    printResponseLine(CLIENT_RESPONSE, b, id, responseContext.getStatus());
    //关注点 2: 获取响应头信息
    printPrefixedHeaders(CLIENT_RESPONSE, b, id, responseContext.getHeaders());
    LOGGER.info(b.toString());
  }
```

在这段代码中，Filter 类实现了客户端响应过滤。从客户端响应上下文实例中，可以获取响应状态信息和响应头信息，分别见关注点 1 和关注点 2。

```
  @Override
  public void filter(ContainerRequestContext context) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    //关注点 1: 获取容器请求方法和请求地址信息
    printRequestLine(SERVER_REQUEST, b, id, context.getMethod(),
    context.getUriInfo().getRequestUri());
    //关注点 2: 获取请求头信息
    printPrefixedHeaders(SERVER_REQUEST, b, id, context.getHeaders());
    LOGGER.info(b.toString());
  }
```

在这段代码中，Filter 类实现了容器请求过滤。从容器请求上下文实例中，可以获取请求方法和请求资源地址信息，见关注点 1。同样，可以从中获取请求头信息，见关注点 2。

```
  @Override
  public void filter(ContainerRequestContext requestContext,
  ContainerResponseContext responseContext) throws IOException {
    long id = logSequence.incrementAndGet();
    StringBuilder b = new StringBuilder();
    //关注点 1: 获取容器响应状态
    printResponseLine(SERVER_RESPONSE, b, id, responseContext.getStatus());
    //关注点 2: 获取容器响应头信息
    printPrefixedHeaders(SERVER_RESPONSE, b, id,
```

```
        HeadersFactory.asStringHeaders(responseContext.getHeaders()));
        LOGGER.info(b.toString());
    }
```

在这段代码中，Filter 类实现了容器响应过滤。从容器响应上下文实例中，可以获取容器的响应状态信息和响应头信息，分别见关注点 1 和关注点 2。

2. 单元测试类

访问日志的单元测试示例如下所示。

```
public class TIResourceJtfTest extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        return config.register(com.example.filter.log.AirLogFilter.class);
    }
    @Override
    protected void configureClient(ClientConfig config) {
        config.register(new AirLogFilter());
    }
```

在这段代码中，为了使访问日志类生效，需要测试类 TIResourceJtfTest 在 Jersey 测试框架的服务器端和客户端，分别注册服务日志类 AirLogFilter。

单元测试的结果如下所示，在 4 种过滤器中分别打印了该阶段的日志信息。

```
main - 1 * AirLog - Request received on thread main
1 / GET http://localhost:9998/books/1
1 / Accept: application/json
Grizzly-worker(1) - 2 * AirLog - Request received on thread Grizzly-worker(1)
2 > GET http://localhost:9998/books/1
2 > accept: application/json
. . .
Grizzly-worker(1) - 3 * AirLog - Response received on thread Grizzly-worker(1)
3 < 200
3 < Content-Type: application/json
main - 4 * AirLog - Response received on thread main
4 \ 200
4 \ Content-Type: application/json
. . .
```

4.5 REST 拦截器

拦截器和过滤器的相同之处在于都是一种在请求—响应模型中用做切面处理的 Provider。两者的不同除了功能上的差异（一个用于过滤消息，另一个用于拦截处理）之外，形式上也不同。拦截器通常读写成对，而且没有服务器端和客户端的区分。Jersey 提供的拦截器类如图 4-9 所示。

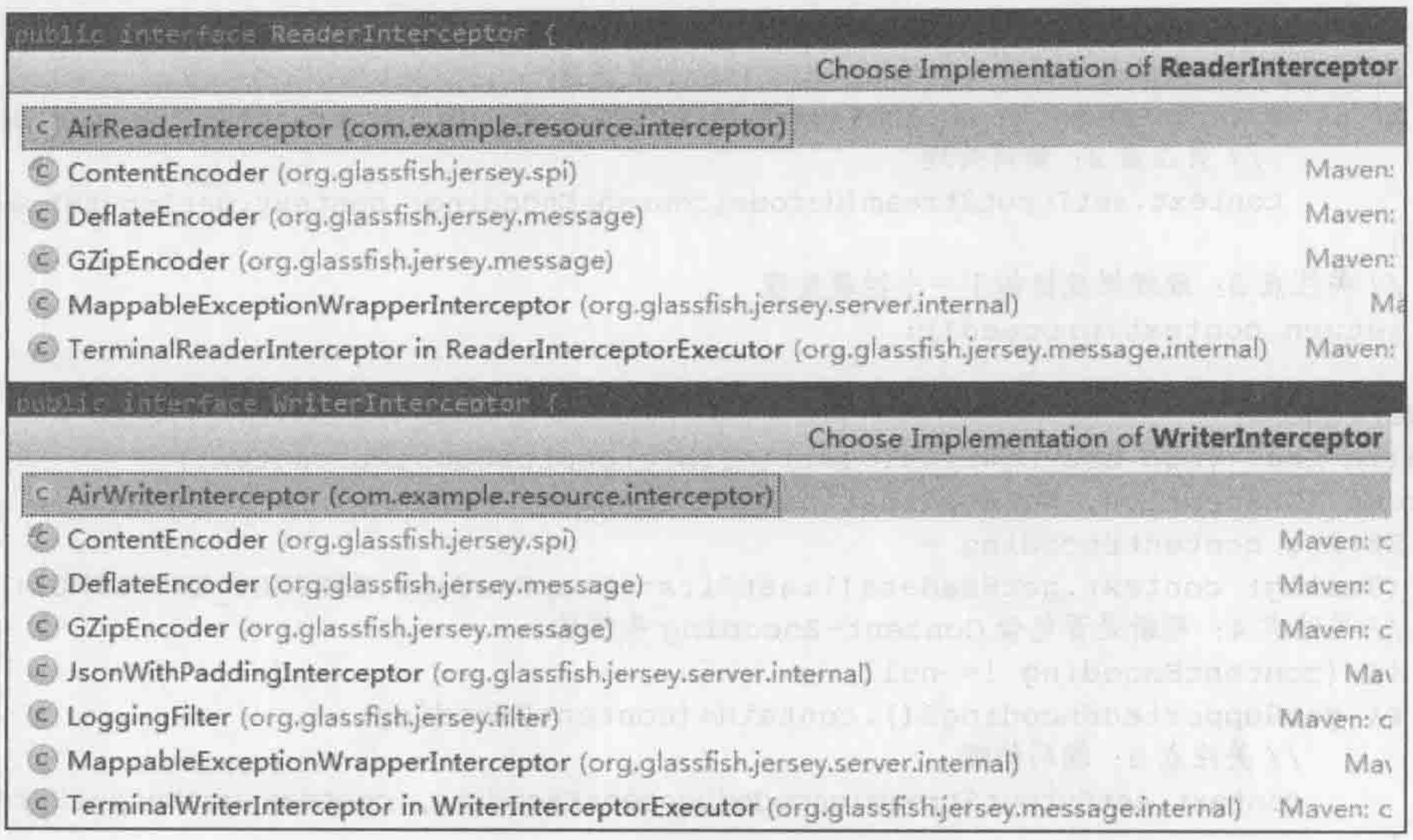


图 4-9 读写拦截器的实现类

如图 4-9 所示，Jersey 内部实现了几个典型应用的拦截器，它们是成对出现的。比如 GZiPEncoder 同时实现了读 / 写拦截器，以实现使用 GZip 压缩格式压缩消息体的功能。

（1）ReaderInterceptor

读拦截器接口 ReaderInterceptor 定义的拦截方法是 aroundReadFrom()，该方法包含一个输入参数，即读拦截器的上下文接口 ReaderInterceptorContext，从中可以获取头信息、输入流以及父接口 InterceptorContext 提供的媒体类型等上下文信息。接口方法示例如下。

```
public Object aroundReadFrom(ReaderInterceptorContext context) throws
java.io.IOException, javax.ws.rs.WebApplicationException;
```

（2）WriterInterceptor

写拦截器接口 WriterInterceptor 定义的拦截方法是 aroundWriteTo()，该方法包含一个输入参数，即写拦截器上下文接口 WriterInterceptorContext，从中可以获取头信息、输出流以及父接口 InterceptorContext 提供的媒体类型等上下文信息。接口方法示例如下所示。

```
void aroundWriteTo(WriterInterceptorContext context)throws
java.io.IOException, javax.ws.rs.WebApplicationException;
```

（3）编解码约束拦截器

编解码约束拦截器类 ContentEncoder 是一个位于 org.glassfish.jersey.spi 包中的拦截器，SPI 包下的工具是可插拔的。ContentEncoder 拦截器用于约束序列化和反序列化过程中的编解码的内容协商，示例代码如下所示。

```
@Override
public final Object aroundReadFrom(ReaderInterceptorContext context)
throws IOException, WebApplicationException {
  String contentEncoding = context.getHeaders()
```

```
        .getFirst(HttpHeaders.CONTENT_ENCODING);
        // 关注点 1: 判断是否包含 Content-Encoding 头信息
        if (contentEncoding != null && getSupportedEncodings().contains(contentEncoding)) {
                // 关注点 2: 解码处理
            context.setInputStream(decode(contentEncoding, context.getInputStream()));
        }
        // 关注点 3: 继续拦截链的下一个拦截处理
        return context.proceed();
    }
    @Override
    public final void aroundWriteTo(WriterInterceptorContext context)
    throws IOException, WebApplicationException {
        String contentEncoding =
        (String) context.getHeaders().getFirst(HttpHeaders.CONTENT_ENCODING);
        // 关注点 4: 判断是否包含 Content-Encoding 头信息
        if (contentEncoding != null
        && getSupportedEncodings().contains(contentEncoding)){
            // 关注点 5: 编码处理
            context.setOutputStream(encode(contentEncoding, context.getOutputStream()));
        }
        // 关注点 6: 继续拦截链的下一个拦截处理
        context.proceed();
    }
```

在这段代码中，分别给出了 ContentEncoder 拦截器的读、写拦截处理，读取阶段进行解码，写入阶段进行编码，分别见关注点 2 和关注点 5。只有当头信息包含“Content-Encoding”信息时，编解码才被执行，见关注点 1 和关注点 4。上下文的 proceed() 方法用于执行拦截器链的下一个拦截器，见关注点 3 和关注点 6。

4.6 绑定机制

在我们了解了面向切面的 Providers 的功能后，需要掌握它们是如何加载的，以及它们的作用域。这些容器级别的 Providers，通常使用编码的方式注册到 Application 中，但这不是唯一的办法。本节将详细讨论 Providers 的绑定机制。

默认情况下，过滤器和拦截器都是全局绑定的，也就是说，下列其中之一的过滤器或拦截器是全局有效的：

- 通过手动注册到 Application 或者 Configuration。
- 注解为 @Provider，被自动探测。

下面介绍其他的绑定机制。

1. 名称绑定

过滤器或拦截器可以使用特定的注解来指定其作用范围，这种特定的注解称为名称绑定。

(1) *名称绑定注解*

使用 @NameBinding 注解可以定义一个运行时的自定义注解，该注解用于定义类级别名

称和类的方法名，示例代码如下所示。

```
@NameBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface AirLog {
}
```

在这段代码中，自定义注解 AirLog 使用了 @NameBinding，在运行时该注解将解析为一个名称绑定的注解。

（2）绑定 Provider

在定义了 @AirLog 注解后，即可在 Provider 中使用该注解，示例代码如下所示。

```
//关注点 1: 使用自定义注解 @AirLog
@AirLog
@Priority(Priorities.USER)
public class AirNameBindingFilter implements ContainerRequestFilter,
ContainerResponseFilter {
    private static final Logger LOGGER = Logger.getLogger(AirNameBindingFilter.class);
    public AirNameBindingFilter() {
        LOGGER.info("Air-NameBinding-Filter initialized");
    }
    @Override
    //关注点 2: filter 实现访问日志
    public void filter(final ContainerRequestContext containerRequest)
    throws IOException {
        LOGGER.debug("Air-NameBinding-ContainerRequestFilter invoked:" +
        containerRequest.getMethod());
        LOGGER.debug(containerRequest.getUriInfo().getRequestUri());
    }
    @Override
    //关注点 3: filter 实现访问日志
    public void filter(ContainerRequestContext containerRequest,
    ContainerResponseContext responseContext) throws IOException {
        LOGGER.debug("Air-NameBinding-ContainerResponseFilter invoked:" +
        containerRequest.getMethod());
        LOGGER.debug("status=" + responseContext.getStatus());
    }
}
```

在这段代码中，过滤器类 AirNameBindingFilter 使用了自定义注解 @AirLog，这样 AirNameBindingFilter 类就实现了名称绑定，见关注点 1。该类实现了容器的请求和响应过滤器接口，功能是记录访问日志，见关注点 2 和关注点 3。

（3）绑定方法

接下来，我们在资源方法级别使用自定义注解 @AirLog，来实现在资源类的指定方法上启用 AirNameBindingFilter 过滤器，示例代码如下所示。

```
@Path("books")
public class BookResource {
```

```
//关注点 1: 绑定方法
@AirLog
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Books getBooks() {
...
        return books;
    }
    @Path("{bookId:[0-9]*}")
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Book getBookByPath(@PathParam("bookId") final Long bookId) {
        final Book book = BookResource.memoryBase.get(bookId);
        return book;
    }
```

在这段代码中，资源类 BookResource 包含多个方法，我们只在 getBooks() 方法上使用了注解 @AirLog，而其他方法并没有绑定，见关注点 1。

(4) 单元测试类

接下来，通过单元测试来校验名称绑定的设计和实现是否正确，示例代码如下所示。

```
public class TestNamingBinding extends JerseyTest {
    @Override
    protected Application configure() {
    //关注点 1: AirAopConfig 内部注册了 AirNameBindingFilter
    return new AirAopConfig();
    }
    @Test
    //关注点 2: 测试 getBookByPath() 方法
    public void testPathGetJSON() {
        final WebTarget pathTarget = target(BASE_URI).path("1");
        final Invocation.Builder invocationBuilder =
         pathTarget.request(MediaType.APPLICATION_JSON_TYPE);
        final Book result = invocationBuilder.get(Book.class);
        Assert.assertNotNull(result.getBookId());
    }
    @Test
    //关注点 3: 测试 getBooks() 方法
    public void testGetAll() {
        final Invocation.Builder invocationBuilder = target(BASE_URI).request();
        final Books result = invocationBuilder.get(Books.class);
        Assert.assertNotNull(result.getBookList());
    }
}
```

在这段代码中，测试类 TestNamingBinding 通过 AirAopConfig() 注册了 AirNameBindingFilter 过滤器，见关注点 1。该类包含两个测试方法，分别测试资源类 BookResource 的 getBook ByPath() 和 getBooks() 方法，分别见关注点 2 和关注点 3。

我们可以从终端打印的信息来检验名称绑定的运行结果，示例如下。

```
Air-NameBinding-ContainerRequestFilter invoked:GET
http://localhost:9998/books/
Air-NameBinding-ContainerResponseFilter invoked:GET
status=200
```

从上述测试结果中可以看到，只有在 testGetAll() 方法输出的日志中输出了 AirNameBindingFilter 类中定义的日志信息。这和预期的“只有使用注解 @AirLog 定义的方法，才会在请求流程中启用相应的 Provider”一致。

2. 动态绑定

名称绑定需要通过自定义的注解名称来绑定 Provider 和扩展点方法或者类，相比而言，动态绑定无须新增注解，而是使用编码的方式，实现动态特征接口 javax.ws.rs.container.DynamicFeature，定义扩展点方法的名称、请求方法类型等匹配信息。在运行期，一旦 Provider 匹配当前处理类或方法，面向切面的 Provider 方法即被触发。

（1）定义绑定 Provider

AirDynamicFeature 类实现了 DynamicFeature 接口，示例代码如下。

```
public class AirDynamicFeature implements DynamicFeature {
    @Override
    public void configure(final ResourceInfo resourceInfo, final FeatureContext context) {
        boolean classMatched =
        BookResource.class.isAssignableFrom(resourceInfo.getResourceClass());
        boolean methodNameMatched =
        resourceInfo.getResourceMethod().getName().contains("getBookBy");
        boolean methodTypeMatched =
        resourceInfo.getResourceMethod().isAnnotationPresent(POST.class);
        //关注点 1: 匹配成功才注册 AirDynamicBindingFilter
        if (classMatched &&(methodNameMatched || methodTypeMatched)) {
            context.register(AirDynamicBindingFilter.class);
        }
    }
}
public class AirDynamicBindingFilter implements ContainerRequestFilter {
    @Override
    public void filter(final ContainerRequestContext requestContext) throws IOException {
        AirDynamicBindingFilter.LOGGER.debug("Air-Dynamic-Binding-Filter invoked");
    }
}
```

在这段代码中，在 AirDynamicFeature 的配置方法中，启用了如下匹配规则。

- 类匹配：对 BookResource 类及其子类的匹配。
- 方法名称匹配：方法名包含 getBookBy 的匹配。
- 请求方法类型匹配：与 POST 方法的匹配。

只有当匹配成功时，才会注册 AirDynamicBindingFilter。对于 Provider 的实现类，并没有特殊的要求。

(2) 单元测试类

测试类 TestDynamicBinding 注册了动态绑定特征实现类 AirDynamicFeature，示例代码如下所示。

```
public class TestDynamicBinding extends JerseyTest {
    @Override
    protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        config.register(AirDynamicFeature.class);
        return config;
    }
    @Test
    public void testPost() {
        final Book newBook = new Book("Java Restful Web Service 使用指南 -" + System.nanoTime());
        final Entity<Book> bookEntity =
        Entity.entity(newBook, MediaType.APPLICATION_JSON_TYPE);
        final Book savedBook =target(BASE_URI).request(MediaType.APPLICATION_JSON_TYPE)
        .post(bookEntity, Book.class);
        Assert.assertNotNull(savedBook.getBookId());
    }
}
```

运行测试方法，AirDynamicBindingFilter 的日志信息如预期输出，示例代码如下所示。

```
Air-Dynamic-Binding-Filter initialized
Air-Dynamic-Binding-Filter invoked
```

4.7 优先级

不同扩展点的 Provider 在请求处理流程中的顺序在 4.3 节已经阐述，对于同一个扩展点的多个 Provider 的执行先后顺序是靠优先级排序的。

优先级的定义使用注解 @Priority，优先级的值是一个整型值，常量定义在 javax.ws.rs.Priorities 类中。对于 ContainerRequest、PreMatchContainerRequest、ClientRequest 和读写拦截器，该数值采用升序策略，即数值越小，优先级越高；对于 ContainerResponse 和 ClientResponse，该数值采用降序策略，即数值越大，优先级越高。示例代码如下所示。

```
@Priority(Priorities.USER)
public class AirNameBindingFilter
@Priority(Priorities.USER + 1)
public class AirNameBindingFilter2
```

在这段代码中，AirNameBindingFilter2 的 Priority 数值大于 AirNameBindingFilter，在请求过程中，AirNameBindingFilter 优先执行，而在响应过程中，AirNameBindingFilter2 优先执行。单元测试代码如下所示。

```
public class TestPriority extends JerseyTest {
```

```
    @Override
        protected Application configure() {
        ResourceConfig config = new ResourceConfig(BookResource.class);
        config.register(AirNameBindingFilter.class);
        config.register(AirNameBindingFilter2.class);
        return config;
    }
    @Test
    public void testGetAll() {
        final Invocation.Builder invocationBuilder = target(BASE_URI).request();
            final Books result = invocationBuilder.get(Books.class);
            Assert.assertNotNull(result.getBookList());
        }
    }
```

单元测试结果如下所示，在处理请求阶段，AirNameBindingFilter2 执行在后；在响应阶段，AirNameBindingFilter2 执行在先。

```
Air-NameBinding-ContainerRequestFilter invoked:GET
http://localhost:9998/books/
Air-NameBinding-ContainerRequestFilter2  Priority+1 invoked
Air-NameBinding-ContainerResponseFilter2 Priority+1 invoked
Air-NameBinding-ContainerResponseFilter invoked:GET
status=200
```

4.8 本章小结

本章详述了 JAX-RS 2.0 定义的 Provider 及其两个特殊类型：过滤器和拦截器，并详细讲述了 JAX-RS 2.0 对 REST 请求的处理过程中各个扩展点对应的 Provider。随后讲述了 Provider 的绑定机制和优先级机制。下一章，我们开始介绍 REST 的客户端。

Chapter 3 第 5 章

REST 客户端

在 Java API for RESTful Web Services（JAX-RS）1.x 的定义中，没有关于 REST 客户端的定义。但是，为了方便开发者使用，JAX-RS 的多个实现项目（包括 Jersey、CXF 等）都自行弥补了 REST 客户端的实现。Jersey 1.x 的客户端实现底层采用的是 Apache Http Client。JAX-RS 2.0 标准化了 Client API（客户端 API）。Jersey 2.x 的客户端包 jersey-client 实现了 JAX-RS 2.0 的客户端 API，并对其进行了可插拔的（pluggable）扩展。客户端 API 通过 HTTP 请求 Web 资源，其设计宗旨是使客户端 API 符合统一接口和 REST 架构风格，同时客户端 API 应该易于使用，而且服务器端在概念和扩展点上保持一致。与 Apache HTTP Client 和 HttpURLConnection 相比，客户端 API 是具备对 REST 的感知的高层 API，可以和 Providers 集成，返回值直接对应高层的业务类实例，而不是 JAXB 对象或者更为低层的数据类型。

有关 REST 客户端的实例贯穿每一章的实例中。因为本书提倡单元测试，所以每个实例中的单元测试皆使用了 REST 客户端。比如有关安全性的客户端实例可以参考相关章节的代码，本章实例主要针对通用接口和连接器实践。

阅读指南

本章示例所在目录是 jax-rs2-guide\sample\5\jaxrs2-client。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/5/jaxrs2-client。

5.1 客户端接口

REST 客户端主要包括三个接口：javax.ws.rs.client.Client、javax.ws.rs.client.WebTarget

和 javax.ws.rs.client.Invocation。接下来我们逐一讲述。

5.1.1 Client 接口

Client 接口是 REST 客户端的基本接口，用于和 REST 服务器通信。Client 被定义为一种重量级的对象，其内部要管理客户端通信底层实现所需的各种对象，比如连接器、解析器等。因此，不推荐在应用中产生大量的 Client 实例，这一点需要开发者注意。为了提示读者注意，摘出源代码中的注释如下。

```
It is therefore advised to construct only a small number of Clientinstances in the application.
```

这句话的意思是在应用中尽量少地构造 Client 实例。另外，该接口要求其实例要有关闭连接的保障，否则会造成内存泄漏。

Jersey 对 JAX-RS 2.0 的 Client 接口的实现类是 org.glassfish.jersey.client.JerseyClient。创建一个 Client 实例是通过 ClientBuilder 构造的，通常使用一个 ClientConfig 实例作为参数，该实例的构建过程是 23 种设计模式中构造模式的实践。在传入 ClientConfig 实例前，通常会注册（register）相关的 Provider 和 Feature，也可以设定自定义属性（键值对），但这些编码都是可选的。最简单的形式是没有参数的构造，示例如下。

```
Client client = ClientBuilder.newClient();
```

通常情况下，在客户端需要加载指定的资源、注册某些 Provider 或者预定义某些属性，以支持更复杂的业务逻辑。我们在 ClientConfig 实例中完成了这些配置后，将其作为参数在 Client 构建时传入，示例代码如下。

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.register(MyProvider.class);// 关注点 1: 注册 Provider
clientConfig.register(MyFeature.class);// 关注点 2: 注册 Feature
clientConfig.register(new AnotherClientFilter());// 关注点 3: 注册 Filter
Client client = ClientBuilder.newClient(clientConfig);
client.property("MyProperty", "MyValue");// 关注点 4: 注册属性
```

在这段代码中，在构造 Client 实例前，预先配置了 ClientConfig 实例，见关注点 1 ~ 关注点 3。在 Client 实例中，通过 property() 方法同样可以配置相关的属性，见关注点 4。

在配置完毕后，可以通过 Client 的 getConfiguration() 方法获取当前 Client 实例的配置信息，示例代码如下。

```
private void checkConfig() {
  Configuration newConfiguration = client.getConfiguration();// 关注点 1: 获取 client 配置
  Map<String, Object> properties = newConfiguration.getProperties();// 关注点 2: 获取配置属性
  Iterator<Entry<String, Object>> iterator = properties.entrySet().iterator();
  while(iterator.hasNext()) {// 关注点 3: 迭代属性
      Entry<String, Object> next = iterator.next();
      LOGGER.debug(next.getKey() + ":" + next.getValue());
  }
}
```

在这段代码中，在关注点 1 处获取了 client 配置，该配置包含了当前客户端实例的属性键值对信息，见关注点 2。通过迭代该键值对可以获取全部的属性信息，见关注点 3。

Client 接口还提供了对客户端的安全连接和异步的支持。为了不重复讲述，可参见本书第 6 章和第 8 章的相关内容。

5.1.2 WebTarget 接口

WebTarget 接口是为 REST 客户端实现资源定位的接口。通过 WebTarget 接口，可以定义请求资源的具体地址、查询参数和媒体类型等信息。Jersey 对 JAX-RS 2.0 的 WebTarget 接口的实现类是 org.glassfish.jersey.client.JerseyWebTarget。

WebTarget 对象接收配置参数的方法定义有很浓厚的 DSL（Domain Specific Language）“味道”，通过方法链即可完成对一个 WebTarget 实例的配置。但是，值得注意的是，如果不将方法链写成一行，而是要分开来写，需要知道 WebTarget 接口所采用的方法链模式是一个不变式（immutable）。也就是说，其返回值是一个新的 WebTarget 对象，我们无法像使用 StringBuilder 的 append() 方法（该方法返回的是 this）那样设置 WebTarget 对象，而是必须接收设置方法的返回值，作为后续流程的句柄。

如下做法是不合适的，WebTarget 初始化一行后面的方法调用并不对该 WebTarget 实例产生作用。

```
WebTarget webTarget = client.target(BASE_URI);
webTarget.path("books");// 关注点 1: 设置的返回值被丢弃了
webTarget.path("book");
webTarget.queryParam("bookId", "1");
final Invocation.Builder invocationBuilder = webTarget.request(MediaType.APPLICATION_XML);
http://localhost:9527/client/
```

在这段代码中，WebTarget 实例每次设置都将产生一个新的 WebTarget 实例，但不幸的是，该实例在下一行没有被使用，原来的 WebTarget 实例并没有使用到这一配置。见关注点 1。

正确的用法如下，每次使用一个新的 WebTarget 都接收 WebTarget 的方法调用，最终使用最后一个实例作为请求句柄。

```
WebTarget webTarget = client.target(BASE_URI);
WebTarget pathTarget = webTarget.path("books");
WebTarget pathTarget2 = pathTarget.path("book");
WebTarget queryTarget = pathTarget2.queryParam("bookId", "1");
final Invocation.Builder invocationBuilder =
queryTarget.request(MediaType.APPLICATION_XML);
http://localhost:9527/client/books/book?bookId=1
```

5.1.3 Invocation 接口

Invocation 接口是在完成资源定位配置后，向 REST 服务端发起请求的接口。请求包括同步和异步两种方式，由 Invocation 接口内部的 Builder 接口定义，Builder 接口继承了同步

接口 SyncInvoker。SyncInvoker 中定义了标准的 HTTP 请求方法。Jersey 对 JAX-RS 2.0 的 Invocation 接口的实现类是 org.glassfish.jersey.client.JerseyInvocation。

Invocation.Builder 接口实例分别执行了 GET 请求和 POST 请求来提交查询和创建。默认情况下，HTTP 方法调用的返回类型是 Response 类型，同时也支持泛型类型的返回值。

```
final Invocation.Builder invocationBuilder =
queryTarget.request(MediaType.APPLICATION_XML);
final Book book = invocationBuilder.get(Book.class);
Invocation.Builder invocationBuilder = target().path(AtupApi.USER_PATH).request();
Response response = invocationBuilder.post(userEntity);
```

5.2 资源释放

作为 REST 框架，JAX-RS 2.0 不希望开发者编码实现对客户端实例的资源管理，Response 实例的 readEntity() 在返回响应实体的同时，即完成了对客户端资源的释放。因此，开发者无须担心连接、释放等资源管理细节。也许开发者的编码没有使用 Response 的 readEntity()，而是直接返回泛型类型的对象，然而其底层还是使用了这个方法。我们可以从图 5-1 所示的调用栈示意图中一探究竟。

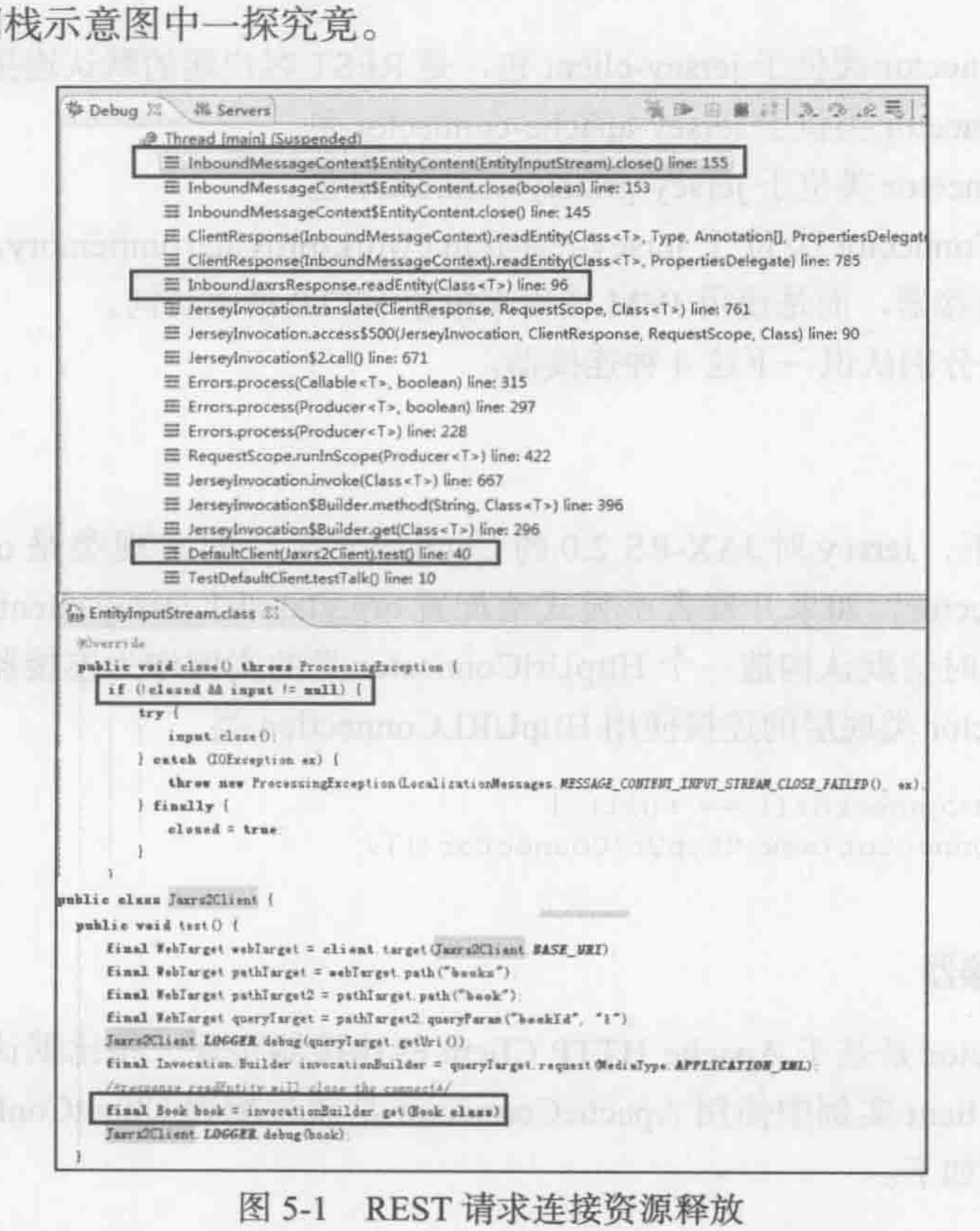


图 5-1 REST 请求连接资源释放

如图 5-1 所示，测试代码使用 Invocation.Builder 接口实例执行 GET 请求，根据 Debug 栈信息，可以发现 readEntity() 方法的身影，对应显示 line 为 96 的一行。再往下的栈进入了输入流类中，对应显示 line 为 155 的一行。在该类的 close() 方法中，可以看到最终关闭了流对象，资源得以释放。

5.3 连接器

Connector 接口是 REST 客户端底层连接器接口，Jersey 为 Connector 接口提供了 4 个实现，如图 5-2 所示。

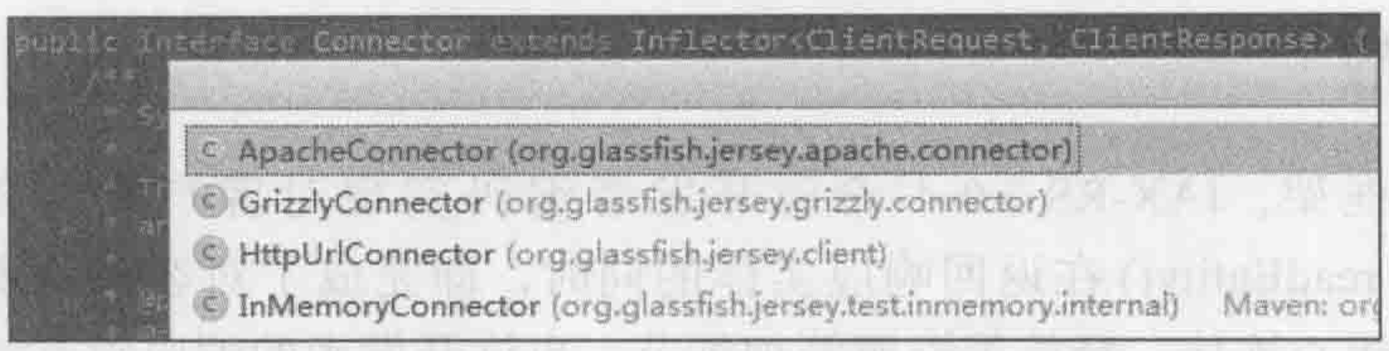


图 5-2 Connector 接口实现类

其中：

- HttpUrlConnector 类位于 jersey-client 包，是 REST 客户端的默认连接器。
- ApacheConnector 类位于 jersey-apache-connector 包。
- GrizzlyConnector 类位于 jersey-grizzly-connector 包。
- InMemoryConnector 类位于 jersey-test-framework-provider-inmemory，不是一种真实的 HTTP 连接器，而是使用 JVM 调用来模拟 HTTP 请求访问。

接下来，我们分别认识一下这 4 种连接器。

1. 默认连接器

在默认情况下，Jersey 对 JAX-RS 2.0 的 Connector 接口的实现类是 org.glassfish.jersey.client.HttpUrlConnector。如果开发者不显式地配置 org.glassfish.jersey.client.ClientConfig，那么在 Client 初始化时会默认构造一个 HttpUrlConnector 类的实例作为连接器，源代码示例如下。HttpUrlConnector 类底层的连接使用 HttpURLConnection 类。

```
if (state.getConnector() == null) {
  state.setConnector(new HttpUrlConnector());
}
```

2. Apache 连接器

ApacheConnector 是基于 Apache HTTP Client 的连接器实现，相比默认的连接器功能更完整、强大。在 Client 实例中使用 ApacheConnector 是通过配置 ClientConfig 的 connector 来实现的，示例代码如下。

```
final ApacheConnector connector = new ApacheConnector(clientConfig);
clientConfig.connector(connector);
client = ClientBuilder.newClient(clientConfig);
```

在这段代码中，ClientConfig 实例在 Client 加载前，完成了对 connector 的配置，该 connector 是一个 ApacheConnector 实例。ClientConfig 实例还可以配置和 Apache 连接器相关的属性，示例代码如下。

```
// 关注点 1：代理服务器配置
clientConfig.property(ApacheClientProperties.PROXY_URI, "http://192.168.0.100");
clientConfig.property(ApacheClientProperties.PROXY_USERNAME, "erichan");
clientConfig.property(ApacheClientProperties.PROXY_PASSWORD , "***");
// 关注点 2：连接超时设置
clientConfig.property(ClientProperties.CONNECT_TIMEOUT, 1000);
clientConfig.property(ClientProperties.READ_TIMEOUT, 2000);
```

在这段代码中，分别设置了代理服务器和超时时间。代理服务器的设置包括代理服务器地址、用户名和密码（代码中以星号代替），见关注点 1。连接超时设置包括物理连接超时时间设置和读取数据超时设置，见关注点 2。详情随后讲述。

ApacheConnector 内部使用 DefaultHttpClient 作为底层的连接。DefaultHttpClient 内部定义了 org.apache.http.conn.ClientConnectionManager 接口和 org.apache.http.params.HttpParams 接口，分别通过配置属性 ApacheClientProperties.CONNECTION_MANAGER 和 ApacheClientProperties.HTTP_PARAMS 来定义，默认情况下都是 null。通过定义 ClientConnectionManager 接口的实例，可以改变客户端实例加载的策略，以提高客户端连接处理的性能，可参考本节“HTTP 连接池”相关内容。

（1）代理服务器设置

通过 jersey-apache-connector 包提供的属性值 ApacheClientProperties.PROXY_URI，可以在 Client 实例中使用 HTTP 代理服务器，常量 ApacheClientProperties.PROXY_URI 的值是 jersey.config.apache.client.proxyUri，用于配置代理服务器地址，常量 ApacheClientProperties.PROXY_USERNAME 和 ApacheClientProperties.PROXY_PASSWORD 分别用于配置代理服务器的用户名和密码。

（2）超时设置

默认情况下，连接超时和读取超时是不设置的，就是说客户端一直尝试连接或者读取，直到成功或被服务器主动断开为止。

通过设置 ClientProperties.CONNECT_TIMEOUT，可以定义客户端尝试连接的最长时间，单位是毫秒（ms），当超出这个时间后，客户端会主动放弃连接并抛出超时异常 ConnectTimeoutException。

读取超时是指连接和资源定位成功后，客户端接收服务器响应消息的最长时间。通过设置 ClientProperties.READ_TIMEOUT，可以定义客户端读取超时时间，单位是毫秒，当超出这个时间后，客户端会主动关闭连接，并抛出 SocketTimeoutException 异常。

3. Grizzly 连接器

GrizzlyConnector 是 Grizzly 提供的连接器实现，其内部使用异步处理客户端 com.ning.http.client.AsyncHttpClient 类作为底层的连接，该类来自于 AsyncHttpClient 项目（地址是 https://github.com/AsyncHttpClient）。该项目的 2.0 版本将 Maven 的坐标信息进行了更改：groupId 从 com.ning.http.client 改为 org.asynchttpclient，com.ning.http.client 包名改为 org.asynchttpclient。读者在引用 Grizzly 连接器时应当注意所用版本的 Maven 坐标是否符合上述定义。

AsyncHttpClient 使用 GrizzlyAsyncHttpProvider 处理 HTTP 请求异步事件，GrizzlyAsyncHttpProvider 是 AsyncHandler 接口的实现类。

在 Client 实例中使用 GrizzlyConnector 是通过配置 ClientConfig 的 connector 来实现的，示例代码如下。

```
final GrizzlyConnector connector = new GrizzlyConnector(clientConfig);
clientConfig.connector(connector);
client = ClientBuilder.newClient(clientConfig);
```

在这段代码中，ClientConfig 在 Client 加载前，配置 connector 为 GrizzlyConnector 实例。

4. HTTP 连接池

在讲述完连接器后，我们回到 Client 是重型组件这个问题。Client 的内部不只是简单地包含一个 HTTP 连接器的设计，而是携带了诸如资源状态等信息，因此在系统中频繁地创建 Client 实例会影响总体的性能。一种常见的解决方案是使用 HTTP 连接池来管理连接，而不是每次请求都创建一个 Client 实例。

这里使用 ApacheConnector 来实现 HTTP 连接池，示例代码如下。

```
final ClientConfig clientConfig = new ClientConfig();
final SchemeRegistry registry = new SchemeRegistry();
registry.register(new Scheme("http", 8080, PlainSocketFactory.getSocketFactory()));
final PoolingClientConnectionManager cm = new PoolingClientConnectionManager(registry);
//关注点 1: 配置连接池管理实例
cm.setMaxTotal(20000);
cm.setDefaultMaxPerRoute(10000);
//关注点 2: 将连接池管理实例配置为 ClientConfig 的属性值
clientConfig.property(ApacheClientProperties.CONNECTION_MANAGER, cm);
//关注点 3: 传递客户端配置实例给连接器
final ApacheConnector connector = new ApacheConnector(clientConfig);
clientConfig.connector(connector);
//关注点 4: 客户端配置实例携带了 Apache 连接器信息
client = ClientBuilder.newClient(clientConfig);
```

在这段代码中，首先定义了连接池管理实例，该实例通过 setMaxTotal() 方法设置了最大连接数，通过 setDefaultMaxPerRoute() 方法设置了每条路由的默认最大连接数，见关注点 1。ClientConfig 的属性 ApacheClientProperties.CONNECTION_MANAGER 用于定义连接池管理实例，其属性值为 ClientConnectionManager 接口的实现类 PoolingClientConnectionManager 的实例，该类是线程安全的，支持多线程并发操作，见关注点 2。ApacheConnector 的

初始化需要设置 ClientConfig 实例，该实例在加载到连接器之前可以将 REST 客户端参数配置好，在连接器初始化后将其配置到自身的 connector 属性中，见关注点 3。最后，ClientBuilder 使用携带了连接器信息的 ClientConfig 实例创建的 REST 客户端实例，即可实现对 ApacheConnector 连接器的使用，见关注点 4。

5.4 封装 Client

通常，REST 式的 Web 服务会按模块分别提供独立的 Web 服务，而模块之间的调用通过 Web 服务的 REST API 来实现。因此，每个模块对其他模块的调用即是客户端请求，不必在每次请求时重复编写构造客户端实例的代码。为此，抽象出 Client 到公共模块是非常有必要的。封装 Client 可以减少代码冗余，也为统一 Client 构建策略（避免配置不统一带来的混乱）提供了可能。

如下代码片段来自第 11 章综合实例的 atup-core 模块。rest() 方法封装了 HTTP 请求，参数依次是 HTTP 请求方法名、请求资源地址、请求头参数、请求查询参数、请求媒体类型、返回值类型、请求实体信息。

```
public T rest(String method, String requestUrl, Set<AtupRequestParam>headParams,
Set<AtupRequestParam>queryParams,MediaTyperequestDataType,
Class<T>returnType, T requestData) {
  //关注点 1: 构造 Client
  if (clientConfig == null) {
      clientConfig = new ClientConfig();
  }
  Client client = ClientBuilder.newClient(clientConfig);
  if (!CollectionUtils.isEmpty(clientRegisters)) {
      for (Class<?>clazz : clientRegisters) {
          client.register(clazz);
      }
  }
  //关注点 2: 构造 WebTarget
  WebTarget webTarget = client.target(requestUrl);
  if (!CollectionUtils.isEmpty(queryParams)) {
      for (AtupRequestParam atupRequestParam : queryParams) {
          webTarget = webTarget.queryParam(atupRequestParam.getKey(),
          atupRequestParam.getValue());
      }
  }
  //关注点 3: 构造 Invocation.Builder
  Invocation.Builder invocationBuilder = webTarget.request(requestDataType);
  if (!CollectionUtils.isEmpty(headParams)) {
      for (AtupRequestParam atupRequestParam : headParams) {
          invocationBuilder.header(atupRequestParam.getKey(),
          atupRequestParam.getValue());
      }
  }
  //关注点 4: 发起请求和结果处理
```

```
    javax.ws.rs.core.Response response;
    Entity<T> entity;
    switch (method) {
        case GET:
                response = invocationBuilder.get();
                return response.readEntity(returnType);
        case DELETE:
                response = invocationBuilder.delete();
                return response.readEntity(returnType);
        case PUT:
                entity = Entity.entity(requestData, requestDataType);
                response = invocationBuilder.put(entity);
                return response.readEntity(returnType);
        case POST:
                entity = Entity.entity(requestData, requestDataType);
                response = invocationBuilder.post(entity);
                return response.readEntity(returnType);
        default:
                return null;
    }
}
```

在这段代码中，rest() 方法可以分为 4 个部分，第一部分是构造 Client，见关注点 1；第二部分是构造 WebTarget，见关注点 2；第三部分是构造 Invocation.Builder，见关注点 3；最后一部分是发起请求和结果处理，见关注点 4。

5.5 本章小结

本章讲述了 JAX-RS 2.0 定义的 REST 客户端及其初始化，主要的接口包括 Client、WebTarget 和 Invocation。客户端的实现需要注意的是客户端的实例数量不宜过多，可以根据情况使用 5.3 节讲述的连接器来避免每次请求实例化一个客户端的情况。最后，5.4 节演示了 REST 客户端的初始化代码块的封装和复用。下一章，我们开始讲述 REST 的安全内容。

第二篇 *Part 2*

全面掌握——JAX-RS 2.0 进阶

- 第 6 章　REST 安全
- 第 7 章　JREST 测试
- 第 8 章　REST 推送与异步通信
- 第 9 章　Jersey1.x 迁移
- 第 10 章　JAX-RS 调优

Chapter 6 第 6 章

REST 安全

REST 式的 Web 服务提供了统一接口和资源定位，简化了 Web 服务接口的设计和实现，降低了 Web 服务的复杂度。但是，与此同时，易于识别和理解的 REST 接口也存在着易于破解的危险。如果破坏者通过对 Web 服务资源地址的猜解，获取了删除某一资源的接口并对该 Web 服务进行攻击，那么很容易造成系统数据的破坏。因此，REST 式的 Web 服务的安全性至关重要。本章将对 REST 的安全性进行全面的阐述。

简而言之，实现安全的过程就是系统实现 AAA 的过程。AAA 是指认证（Authentication）、授权（Authorization）和账户（Accounting）。

什么是认证呢？认证就是系统识别访问者身份的过程。举个例子，比如交警对司机进行交通检查，“认证”就是交警确认驾驶者是该车的准驾司机的过程。如果是双向认证，还包括驾驶者对交警身份确认的过程。什么是授权呢？还用这个小例子，授权就是当交警确认并记住（系统是通过 Cookie、加密的 URI 或者 Session）驾驶者的司机身份后，对该准驾类型的司机授予的权限进行监控的过程。比如小型客车的司机在驾驶一辆大型货车，那么该驾驶者没有权限这样做，交警将对其依法处置（系统拒绝访问者访问，返回 HTTP 状态码 403）。

那么，认证是如何实现的呢？简单的认证方式是通过用户名-密码（也称口令）进行验证。访问者在服务器提出身份质询后做出反馈，即（加密）提交用户名和密码，服务器端接收该信息并对其进行验证。用户名-密码验证方式的实现比较简单、易行，但很容易被盗取和破解（即使经过加密处理）。

除了用户名-密码验证方式，还可以通过数字证书进行验证，但验证过程相对复杂。还以交通检查为例，过程如下。

首先，驾驶者出示驾照（数字证书，访问者身份的凭据）。交警拿到驾照后，首先核实驾照的真伪（数字证书是否是伪造的），判断的依据是核对驾照上是否有交警支队的印章 [CA

（Certificate Authority，权威的数字证书认证机构）签发的数字证书]，还要判断这个印章的真伪（CA 是否在对方的信任证书列表中。这个验证过程可能包括对 CA 身份的验证，比如交警支队的身份由交通部门的印章来确认，直至追溯到双方都确信的 CA 出示的根证书。本例中，交警信任交警支队，所以交警支队的印章可以看做双方都信任的根证书）。接着，交警要查看证书的有效期等信息，保证证书合法、有效。到此，交警确认了驾照的合法性（服务器对客户端的数字证书本身的校验完毕）。

最后，交警会查看驾照上对驾驶者身份等信息的描述，来确认其身份和准驾车类型（这个步骤相当于服务器从证书中获取访问者的身份信息）。认证过程中，使用了诸多安全技术，比如加密算法、摘要算法、公钥 / 私钥对、证书生成-导入-导出-吊销等。授权的实现可以由容器级别的配置完成，或者通过程序级别的注解和编码完成。

在了解了认证和授权的过程后，我们从简述理论到实践，逐一讲述 REST 中可以采用的认证和授权办法。

阅读指南

本章示例所在目录是 jax-rs2-guide\sample\6\security-rest。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/6/security-rest。

6.1 身份认证

HTTP 认证规范（RFC 2617）定义了两种 HTTP 身份认证方式：HTTP Basic（基本认证）和 HTTP Digest（摘要认证）。HTTP 认证是一种无状态的认证方式，服务器容器不会为这样的登录匹配 Session，身份信息随浏览器关闭而消失。Java 平台包含了 HTTP 的两种身份认证，此外还定义了 HTTP+HTML form-based authentication（表单认证）和证书认证。

基本认证、摘要认证和表单认证都是基于“用户名-密码”的认证机制，而证书认证是基于证书的认证机制。基本认证和表单认证的请求过程需要提交用户的密码信息，而且对服务器的合法性缺乏判断依据，摘要认证和证书认证的请求过程将对服务器的合法性进行验证而且不直接提交用户的密码。

下面将对这 4 种认证方式逐一讲述。

6.1.1 基本认证

HTTP 的基本认证最初定义在 HTTP 1.0 规范（RFC 1945）中，后续最新的定义包含于 HTTP 1.1 规范（RFC 2616）和 HTTP 认证规范（RFC 2617）。HTTP 基本认证是指通过 Web 浏览器或其他客户端在发送请求时，提供用户名和密码作为身份凭证的一种登录验证方式。在请求发送之前，用户名和密码字符串通过一个冒号合并，形如：Username:Password，合并后的字符串经过 Base64 算法进行编码。

例如，提供的用户名是 eric、密码是 han，合并后为 eric:han，经过 Base64 算法进行编

码后，字符串为 ZXJpYzpoYW4=。

浏览器或客户端最终将经过 Base64 编码的字符串提交给服务器。Base64 编码的目的并不是实现安全与隐私，而是为了将用户名和密码中的与 HTTP 不兼容的字符转换为兼容的字符集。

HTTP 基本认证的请求过程是一个质询 / 回应（challenge/response）的对话流程，即客户端请求一个需要身份认证的资源路径，但是没有提供用户名和密码。服务器端响应 HTTP 状态码 401（Unauthorized）的应答，并提供一个认证域（Basic realm）。客户端增加认证消息头，内容为前述的 Base64 加密字符串，形式为 Authorization: Basic base64encode（Username:Password），并再次发起请求。服务器端接收请求并处理，如果用户凭据非法或无效，服务器可能再次返回 HTTP 状态码 401，客户端可以再次提示用户输入密码。质询 / 回应会话流程是可以省略的，即在客户端第一次请求中就发送认证消息头。

6.1.2 摘要认证

摘要认证最初定义在 RFC 2069（HTTP 的一个扩展：摘要访问认证）中，以服务器生成的随机数来维护安全性。RFC 2069 中定义的认证响应由 HA1、HA2、A1 及 A2 组成。其中，nonce 是服务器端随机数。后续最新的定义包含于 HTTP 认证规范（RFC 2617，是 RFC 2069 的替代方案）。RFC 2617 引入了一系列安全增强的选项，包括 QoP（Quality of Protection，保护质量）、请求计数器 nc（nonceCount）和 cnonce（clientNonce，客户端随机数）。这些增强项有效地防止了通信过程中的明文攻击。RFC 2617 中对 A2、HA2 和 response 做了如下增强定义，见表 6-1。

表 6-1 RFC 2069 和 RFC 2617 摘要认证参数列表

参数组合	说明
A1=Username:Realm:Password A2=method:digestURI HA1=MD5（A1） HA2=MD5（A2） response=MD5（HA1:nonce:HA2）	qop=auth/NONE 时，A2=method:digestURI qop=auth-int 时，A2=method:digestURI:MD5（entityBody） qop=NONE 时，response=MD5（HA1:nonce:HA2） qop=auth/auth-int 时，response=MD5（HA1:nonce:nonceCount:clientNonce:qop:HA2）

HTTP 摘要认证同样遵循请求过程的质询 / 回应对话流程。具体描述参考上文中的基本认证。

6.1.3 表单认证

表单认证是基于 HTTP，使用 HTML 的 Form 标签提交表单的认证形式。用户登录页面定义在 web.xml 文件的 form-login-page 字段中，在没有被认证前，访问者对资源地址的访问会被引导到该页面。访问者提交身份信息后，服务器接收并处理请求，如果认证通过，将重定向到 welcome-file 字段定义的页面，如果失败，将重定向到 form-error-page 字段定义的页面。相对于 HTTP 认证，表单认证允许开发者指定并设计具有良好用户体验的登录页面、登

录成功页面和登录失败页面。

Java 平台处理表单认证的实现是在客户端页面（比如 HTML+AJAX、JSP、JSF 等）中，定义一个名称指定为 j_security_check 的请求方法，该方法传递两个名称固定的字段。用户名字段为 j_username，密码字段为 j_password。服务器端会根据上述固定名称对提交的表单进行解析和验证，示例代码如下。

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

6.1.4 证书认证

证书认证是通过数字证书认证身份的方式。这一方式从技术角度可以拆分为证书管理和通信协议两个环节。证书是静态的文件，为基于 SSL/TLS 协议的通信过程所用。

1. 证书

证书是包含公钥和密钥属主信息的文件。加密是为了将信息保密且完整地传递出去，加密算法是实现这一过程的手段。持有加密算法的文件即为密钥，密钥分为公钥和私钥，在加密、解密环节，公钥用于加密，私钥用于解密。

当甲持有乙的证书后，甲就可以从该证书中获取乙的公钥。使用该公钥对数据进行加密，然后发送给乙，乙使用私有的私钥将数据解密。在这一过程中，信息的保密性和完整性得到了保障，但安全性中的端点认证无法通过加密 / 解密解决。也就是说，甲无法确认接收者就是乙。

证书认证解决了端点认证的两个问题。

第一个问题是验证接收者是乙。在签名环节，私钥用于签名，公钥用于校验。乙的公钥就像乙手写签名的快照一样，只有乙本人的签名（乙的私钥）才与之能匹配。

第二个问题是验证证书的合法性。通信过程中，对方提交的证书说他是乙，那么甲如何相信证书及其所言呢？这需要使用或间接使用另一个权威证书来鉴定该证书的合法性，读者可以回忆一下前面说过的交警支队需要交通部门来证明其真伪的例子，这就是证书链模型。该模型最简版本为甲乙互相信任，这种情况下无须 CA 证书参与进来。最复杂的版本就是甲乙只相信最权威的 CA 认证机构，而他们持有的认证证书却都不是直接来自那个机构签发的。通过证书链逐级上溯，最终，甲通过信任 CA 来认可当前证书是为乙颁发的，这样一来，只要乙的私钥签名在经过证书公钥校验后结果一致，甲就可以相信接收者就是乙。

证书的签发格式遵循 X.509 标准。X.509 是由国际电信联盟（ITU-T）制定的数字证书标准。证书的管理可以通过 OpenSSL（http://www.openssl.org）和 Java 平台的 keytool 工具来实现，本章实例部分将使用 keytool 来实现证书的管理。

2. SSL/TLS

SSL（Secure Socket Layer，安全套接层协议）是在传输通信协议（TCP/IP）上实现的一种安全协议，采用公钥技术保证数据的保密性和完整性。HTTPS 是运行在 SSL 协议之上的 HTTP，其本质还是 SSL 通信。IETF 组织将 SSL 标准化后成为 TLS（Transport Layer Security，传输层安全协议）（RFC 2246）。SSLv2 已经过时，TLS 1.0 非常近似 SSLv3，本书余文使用 TLS 代替 SSL/TLS 的书写方式。

TLS 包含三个基本阶段：①对等协商密钥算法；②传输基于非对称密钥加密的数据和基于 PKI（Public Key Infrastructure，公钥基础设施）证书的身份认证；③传输基于对称密钥加密的数据。

3. JSR 219

在 Java 领域，证书管理、对 TLS 的支持定义在 JSR 219 中。JSR 219 规范（Foundation Profile 1.1，Java 平台对安全领略支持的规范）包括 3 个标准：JCE（Java Cryptography Extension）、JSSE（Java Secure Sockets Extension）、JAAS（Java Authentication and Authorization Service）。

- JCE 定义了 Java 对加密 / 解密、密钥生成和协商，以及 MAC（Message Authentication Code，消息认证码）算法的实现规范。
- JSSE 定义了 Java 对安全传输和握手机制的实现规范，包含了对 SSL/TLS 的支持。
- JAAS 是 Java 平台认证和授权的标准。JAAS 建立在 PAM（Pluggable Authentication Module，可插入的认证模块）安全体系结构之上，一方面，使业务逻辑和安全校验解耦；另一方面，提供了 23 种设计模式中责任链模式的校验过程，使模块化的校验机制可插拔地无缝集成于业务平台中。

6.2 资源授权

通过上文的讲述，我们对 REST 应用中可以采用的 4 种身份认证方式有了初步了解。在有了用户的身份信息后，服务器要做的事情就是识别该用户对 REST 应用中资源的访问权限。授权管理包括容器管理和应用管理。

容器管理权限提供无编码的、可配置的全局的管理方式，零编码、方便统一管理。应用管理权限提供细粒度的权限管理，通过编码使具体业务的权限分配更灵活。接下来我们分别讲述两种授权管理的实现。

6.2.1 容器管理权限

容器管理权限是指容器通过启动时从其配置文件中加载角色-权限信息，在运行时对用户身份进行认证，并对其角色的资源访问权限进行管控的方式。

1. 基于 Realm 的身份认证

Java 平台的 Servlet 容器和 Java EE 容器根据规范各自都提供了认证和授权的实现。本书以 Tomcat 为例展示容器管理权限功能。

在 Tomcat 中，使用 Realm 作为用户身份识别的基本单位，Realm 这个概念类似 UNIX 下的 groups，Tomcat 在运行期根据其中定义的 role 与访问资源的匹配，控制当前用户的访问权限。Realm 的配置通常在服务器配置文件 conf/server.xml 中定义格式。在 Tomcat 配置文件中对 Realm 的定义示例如下。

```
<Realm className=" 实现 org.apache.catalina.Realm 接口的类全名 " ... 该实现的其他属性 .../>
```

Realm 项可以内置于任何一个容器项中，这将影响该 realm 的作用域（scope）。

- 内置于 Engine 项：该 Realm 将作用于所有 hosts 中的 Web 应用，除非在 Host 项或者 Context 项中被覆盖。
- 内置于 Host 项：该 Realm 将作用于所有 virtual host 中的 Web 应用，除非在 Context 项中被覆盖。
- 内置于 Context 项：该 Realm 将只作用于该 Web 应用。

Tomcat 提供了 6 个标准的 Realm 认证插件：JDBCRealm、DataSourceRealm、UserDatabaseRealm、JAASRealm、MemoryRealm（通过读取 XML 格式内存集合对象，获取认证信息。这是一个过时的 Realm，目前被 UserDatabaseRealm 取代）和 JNDIRealm（通过 JNDI 访问 LDAP 服务器，获取认证信息）。本章 6.3 节将通过 5 个示例来展示如何结合各种身份认证和 Tomcat 提供的 Realm 实现 REST 应用中的认证和授权。

2. 角色-权限配置

配置文件中定义的角色-权限信息是使用 XML 完成的。虽然在很多架构或者工具的发展过程中，已经将其 XML 格式的配置文件转为注解方式，但是权限配置使用 XML 定义的方式，层次更清晰。容器管理权限的配置讲述如下。

（1）配置 security-constraint 元素

security-constraint 元素用来定义安全约束，包括可访问的资源、访问者身份和数据传输类型的约束。

1）web-resource-collection：web-resource-collection 元素用来标识需要限制访问的资源子集。在 web-resource-collection 元素中，可以定义 url-pattern（使用资源路径通配符定义的资源集合）和 http-method（HTTP 方法）。如果不存在 HTTP 方法，就将安全约束应用于所有的方法。

2）auth-constraint：auth-constraint 元素用于指定可以访问该资源集合的用户角色。如果没有指定 auth-constraint 元素，就将安全约束应用于所有角色。role-name 元素包含安全角色的名称。

举例说明，在如下定义中，user 角色可以访问 /webapi/* 资源集合中所有的 GET 方法。

```
<security-constraint>
  <web-resource-collection>
      <web-resource-name>restful resources</web-resource-name>
      <url-pattern>/webapi/*</url-pattern>
      <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
      <role-name>user</role-name>
  </auth-constraint>
  <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

3）user-data-constraint：user-data-constraint 元素用来标识在客户端和 Web 容器之间传输数据的方式。transport-guarantee 元素必须具有如下的某个值：NONE，意味着应用不需要传输保证；INTEGRAL，意味着服务器和客户端之间的数据必须以某种方式发送，而且在传送中不能改变；CONFIDENTIAL，意味着传输的数据必须是加密的数据。通常，TLS 使用 INTEGRAL 或 CONFIDENTIAL。

（2）配置 login-config 元素

login-config 元素用来指定所使用的验证方法、领域名和表单验证机制所需的特性。

1）auth-method：auth-method 指定验证方法。它的值为以下 4 种中的一个：BASIC、DIGEST、FORM 或 CLIENT-CERT。

2）realm-name：realm-name 指定验证中使用的领域名。

3）form-login-config：form-login-config 指定基于表单认证的登录中使用的登录页面和登录失败页面。如果没有使用基于表单的验证，则忽略这些元素。form-login-page 用于指定显示登录页面的资源路径，form-error-page 则用于指定用户登录失败时显示出错页面的资源路径。

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

（3）security-role 元素

security-role 元素指定用于安全约束中的安全角色的声明。role-name 指定角色名称。

```
<security-role>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <role-name>user</role-name>
</security-role>
```

6.2.2 应用管理权限

相对于容器管理权限，应用管理权限是指容器中的每个应用各自通过编码和注解完成对资源路径访问的权限管理。

（1）JSR 250

JSR 250（Common Annotations for the Java Platform 1.1，Java 平台通用注解标准）提供了安全访问注解，包名为 javax.annotation.security。应用代码通过使用安全访问注解对资源路径进行定义，实现权限管理。

注解 @PermitAll、@DenyAll 和 @RolesAllowed 用来定义方法级别的访问权限。使用 @DenyAll 注解定义的方法不允许任何角色访问。使用 @PermitAll 注解定义的方法允许所有角色访问。使用 @RolesAllowed 注解定义的方法允许指定的角色访问。如果注解在类级别，适用于类中所有的方法。如果注解在方法级别，则方法级别的权限注解优先于 / 覆盖类级别的权限注解。

注解 @DeclareRoles 和 @RunAs 用来定义类级别的访问权限。

注解 @RunAs 定义的方法允许执行的角色必须存在于安全领域中，并映射到用户或组。注解 @DeclareRoles 用来定义允许执行该类的安全角色。@DeclareRoles（“admin”）相当于如下的 XML 格式的定义。

```
<security-role>
  <role-name>admin</role-name>
</security-role>
```

（2）JAX-RS 2.0

JAX-RS 2.0 规范定义了一个安全上下文接口 SecurityContext，通过该接口的实现可以在运行时获取当前用户的身份信息，从而通过编码方式，动态地为当前用户提供资源访问的权限。

Jersey 2.x 提供了 DynamicFeature 接口的实现类 RolesAllowedDynamicFeature，REST 服务或应用通过注册该类即可使用 JSR 250 的注解和 JAX-RS 2.0 的 SecurityContext 动态管理资源访问权限。其工作原理是 REST 应用使用注册的 RolesAllowedDynamicFeature 从 SecurityContext.isUserInRole() 获取用户角色并和类以及方法中使用的 JSR 250 注解匹配，如果失败，即返回没有权限的错误（HTTP 状态码 403）给客户端。

6.3 认证与授权实现

前面两节分别讲述了认证和授权的方式，本节将结合不同的认证和授权方式，为读者呈现 5 种实现。

阅读指导

6.3 节示例所在目录是：jax-rs2-guide\sample\6\security-rest。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/6/security-rest

6.3.1 基本认证与 JDBCRealm

本示例使用 HTTP 基本认证，结合 Tomcat 提供的 JDBCRealm 实现认证与授权。本例所示的场景为以 webapi/ 开头的资源路径需要用户登录访问。其中获取图书列表的只读资源路径（http://localhost:8080/security-rest/webapi/books）可以被 admin 角色和 user 角色访问，其他以 webapi/ 开头的资源路径只有 admin 角色可以访问。

小白讲堂

Realm 的概念在 6.2.1 节已经介绍过，有关 Tomcat 的 Realm 详情可参考官方文档，地址是：http://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html。

（1）创建 Realm 所需的数据表

JDBCRealm 通过 JDBC 访问关系型数据库获取认证信息，首先我们从数据库脚本入手，逐步完成认证与授权的实现。本例使用 MySQL 数据库，脚本信息如下。

```
// 导出数据库脚本的命令为: mysqldump simpleservicebook -uroot -p security.sql
// 导入数据库脚本的命令为: mysql -uroot -p < security.sql
DROP DATABASE IF EXISTS `simple_service_book`;
CREATE DATABASE `simple_service_book`;
USE `simple_service_book`;
CREATE TABLE `simple_book` (
  `BOOKID` int(11) NOT NULL AUTO_INCREMENT,
  `BOOKNAME` varchar(128) DEFAULT NULL,
  `PUBLISHER` varchar(128) DEFAULT NULL,
  PRIMARY KEY(`BOOKID`),
  UNIQUE KEY `BOOKID`(`BOOKID`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `simple_book` WRITE;
INSERT INTO `simple_book` VALUES(1,'Java Restful Web Service 使用指南','cmpbook'),
(2,'JSF2 和 RichFaces4 使用指南','phei');
UNLOCK TABLES;
CREATE TABLE `user_roles`(
  `user_name` varchar(15) NOT NULL,
  `role_name` varchar(15) NOT NULL,
  PRIMARY KEY(`user_name`,`role_name`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `user_roles` WRITE;
INSERT INTO `user_roles` VALUES('caroline','user'),('eric','admin');
UNLOCK TABLES;
CREATE TABLE `users`(
  `user_name` varchar(15) NOT NULL,
  `user_pass` varchar(15) NOT NULL,
  PRIMARY KEY(`user_name`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `users` WRITE;
INSERT INTO `users` VALUES('caroline','zhang'),('eric','han');
UNLOCK TABLES;
```

这段脚本创建了名为 simple_service_book 的数据库，该数据库包含名为 simple_book 的表用来展示 REST 请求资源，另外，名为 user_roles 的表和名为 users 的表用来展示 JDBCRealm 认证和授权。本例包含两个用户，一个是角色为 admin 的 eric 用户，另一个是角色为 user 的 caroline 用户。认证通过后，根据两个用户具有的身份不同，将会有不同的权限。

（2）配置 JDBCRealm

根据 Tomcat 官方网站提供的文档（http://tomcat.apache.org/tomcat-7.0-doc/config/realm.html），服务器配置文件 $CATALINA_BASE/conf/server.xml 设置如下所示。

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
driverName="org.gjt.mm.mysql.Driver"
connectionURL="jdbc:mysql://localhost:3306/simple_service_book?user=root&amp;password=root"
userTable="users"
userNameCol="user_name"
userCredCol="user_pass"
userRoleTable="user_roles"
roleNameCol="role_name"/>
```

在这段脚本中，用户表为 users，用户名字段为 user_name，密码字段为 user_pass；角色表为 user_roles，角色名字段为 role_name。

$CATALINA_BASE 路径可以是 Tomcat 的物理路径，也可以是 IDE 集成环境中的相对路径。更灵活的设置方式是使用后者，即在 IDE 集成环境中对服务器配置，以方便切换不同配置和断点调试。以 Eclipse 为例，在 Servers 实例的内置 Tomcat 中，配置 server.xml 文件，如图 6-1 所示。

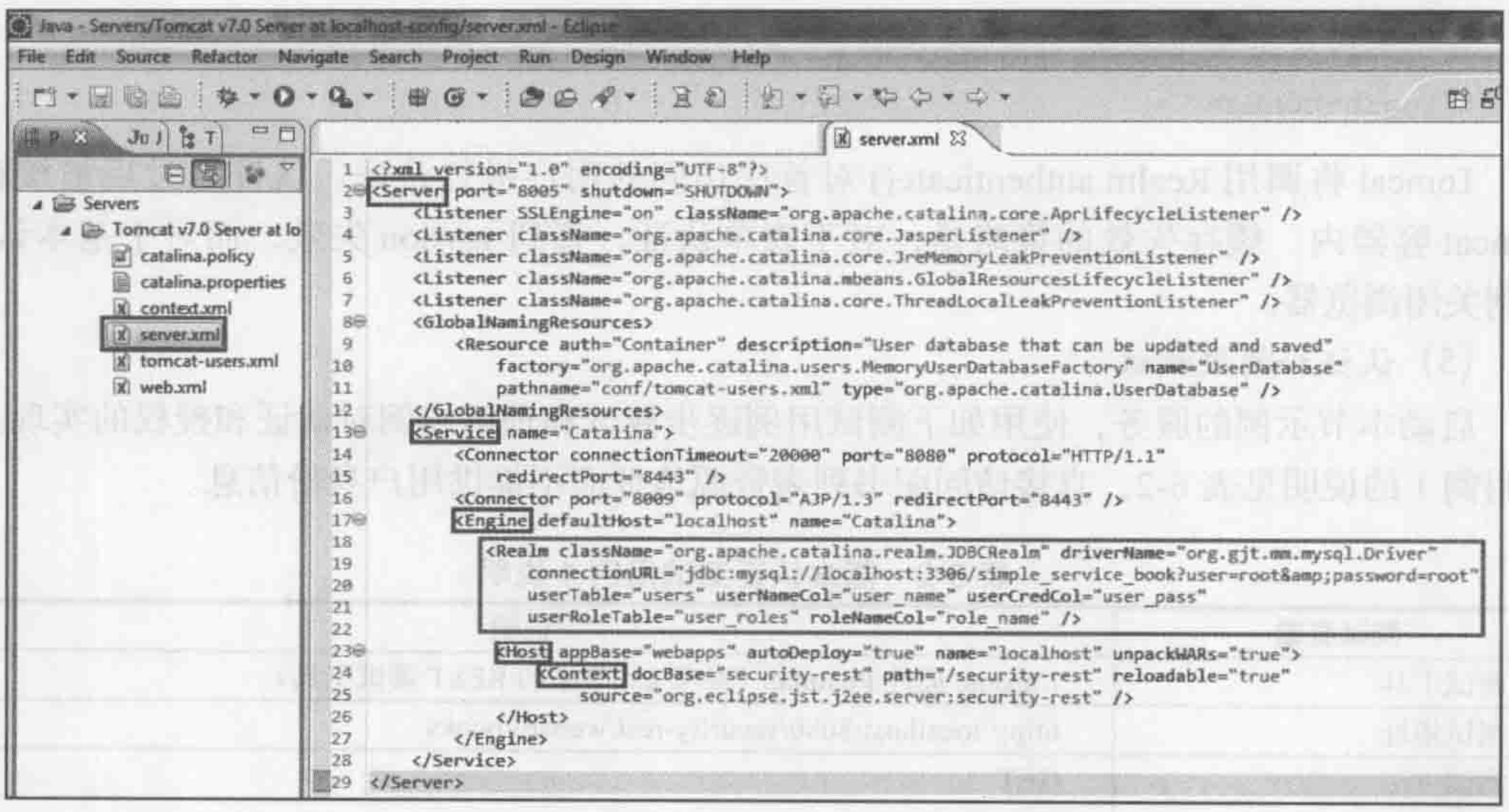


图 6-1 Eclipse 中定义 Tomcat 配置示意图

（3）数据库驱动

复制 MySQL 的 JDBC 驱动到 $CATALINA_HOME/lib 目录。使用 Maven 的项目可以

从本地仓库获取，否则可以网上搜索。本地仓库地址为 M2_REPO/mysql/mysql-connector-java/5.1.25/mysql-connector-java-5.1.25.jar（仓库地址路径示例：M2_REPO=C:\Users\hanl\.m2\repository）。

（4）配置应用

Web 配置文件 /security-rest/src/main/webapp/WEB-INF/web.xml 摘要如下，逻辑如前所述。

```
<security-constraint>
  <web-resource-collection>
      <url-pattern>/webapi/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <http-method>UPDATE</http-method>
      <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
      <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
      <url-pattern>/webapi/*</url-pattern>
      <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
      <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

Tomcat 将调用 Realm.authenticate() 对首次访问的用户进行认证。认证通过后被缓存在 Tomcat 容器内。缓存失效的策略是，对于表单认证，直到 session 失效，而对于基本认证，直到关闭浏览器。

（5）认证和授权测试

启动本节示例的服务，使用如下测试用例逐步测试和理解本例对认证和授权的实现。测试用例 1 的说明见表 6-2，直接访问图书列表资源地址而不提供用户身份信息。

表 6-2 基本认证测试用例 1 说明

测试要素	说明
测试工具	Chrome 插件 Postman（参考 2.6 节中的 REST 调试工具）
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	GET
测试用户	无
测试结果	401 Unauthorized

服务器返回 HTTP 状态码 401，如图 6-2 所示，进入质询 / 回应的对话流程，流程参见

后续测试用例。

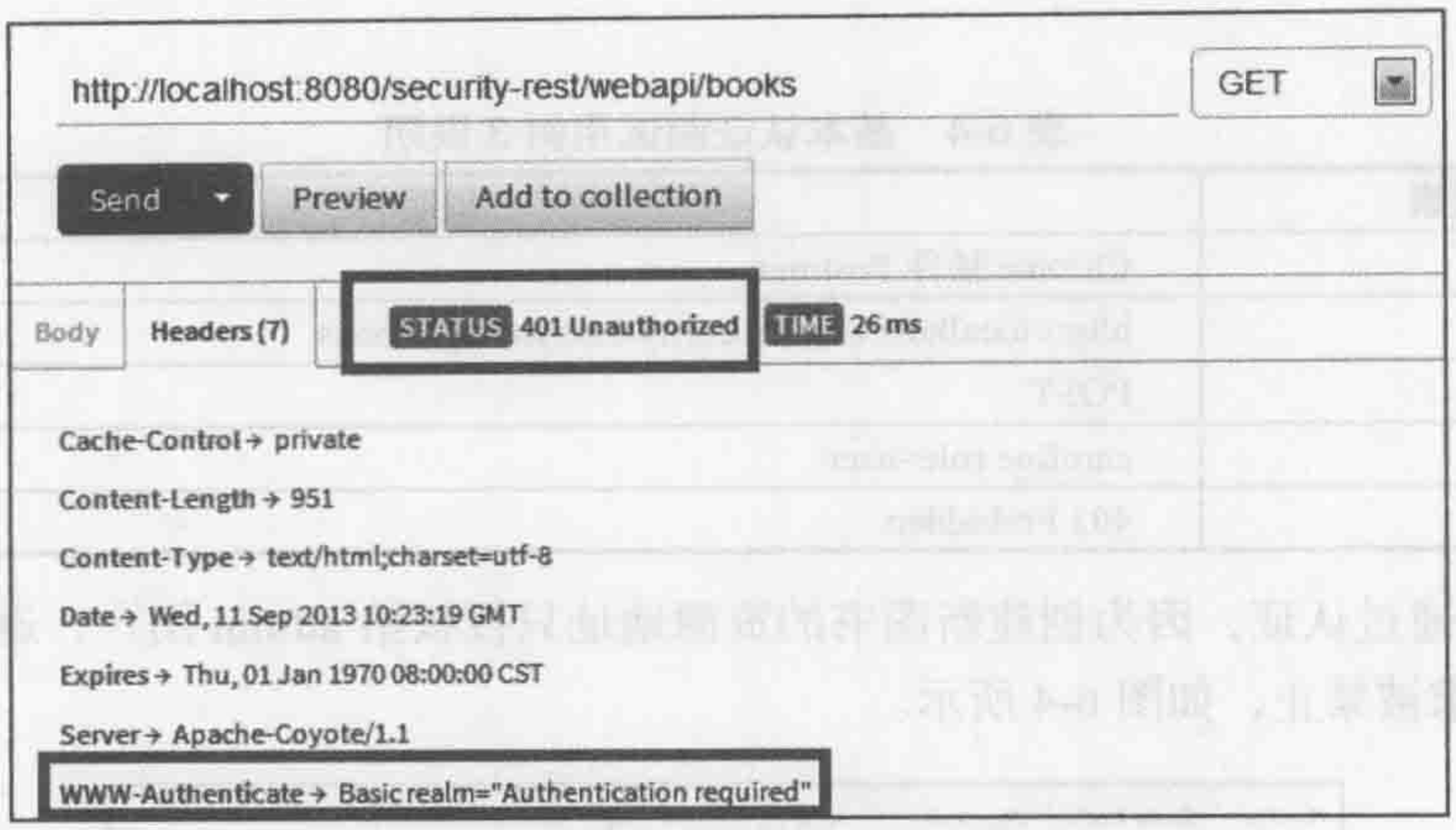


图 6-2　基本认证测试用例 1 结果

测试用例 2 的说明见表 6-3，使用基本认证并设置用户为 caroline 来访问图书列表资源地址。

表 6-3　基本认证测试用例 2 说明

测试要素	说明
测试工具	Chrome 插件 Postman
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	GET
测试用户	caroline role=user
测试结果	200 OK

服务器通过认证，因为 GET 方法只读资源授权给 user 用户，返回 HTTP 状态码为 200 和图书列表资源表述，如图 6-3 所示。

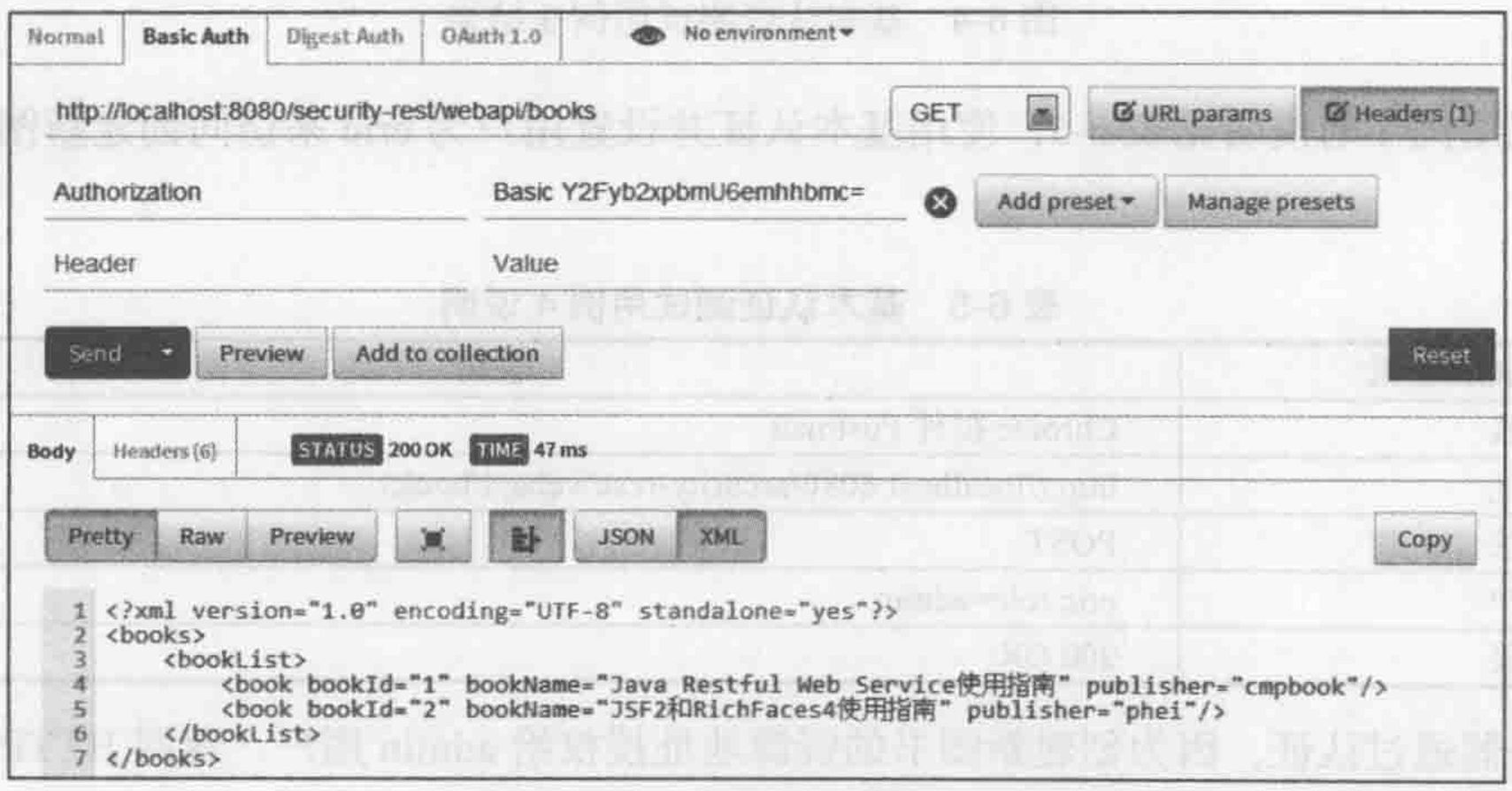


图 6-3　基本认证测试用例 2 结果

测试用例 3 的说明见表 6-4，使用基本认证并设置用户为 caroline 来访问创建新图书的资源地址。

表 6-4 基本认证测试用例 3 说明

测试要素	说明
测试工具	Chrome 插件 Postman
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	POST
测试用户	caroline role=user
测试结果	403 Forbidden

服务器未通过认证，因为创建新图书的资源地址只授权给 admin 用户，返回 HTTP 状态码为 403，请求被禁止，如图 6-4 所示。

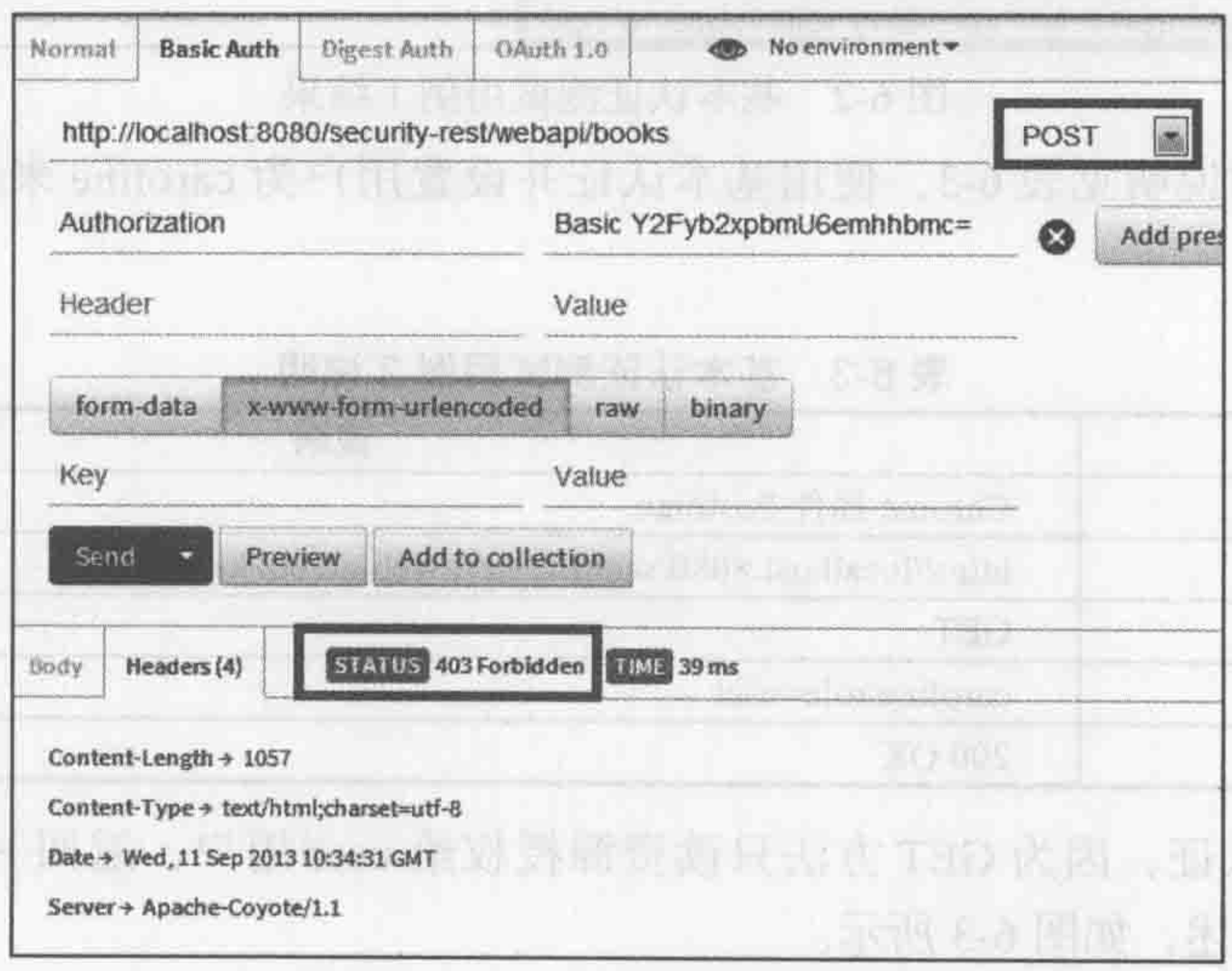


图 6-4 基本认证测试用例 3 结果

测试用例 4 的说明见表 6-5，使用基本认证并设置用户为 eric 来访问创建新图书的资源地址。

表 6-5 基本认证测试用例 4 说明

测试要素	说明
测试工具	Chrome 插件 Postman
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	POST
测试用户	eric role=admin
测试结果	200 OK

服务器通过认证，因为创建新图书的资源地址授权给 admin 用户，返回 HTTP 状态码为 200 和创建完毕的新图书资源表述，如图 6-5 所示。

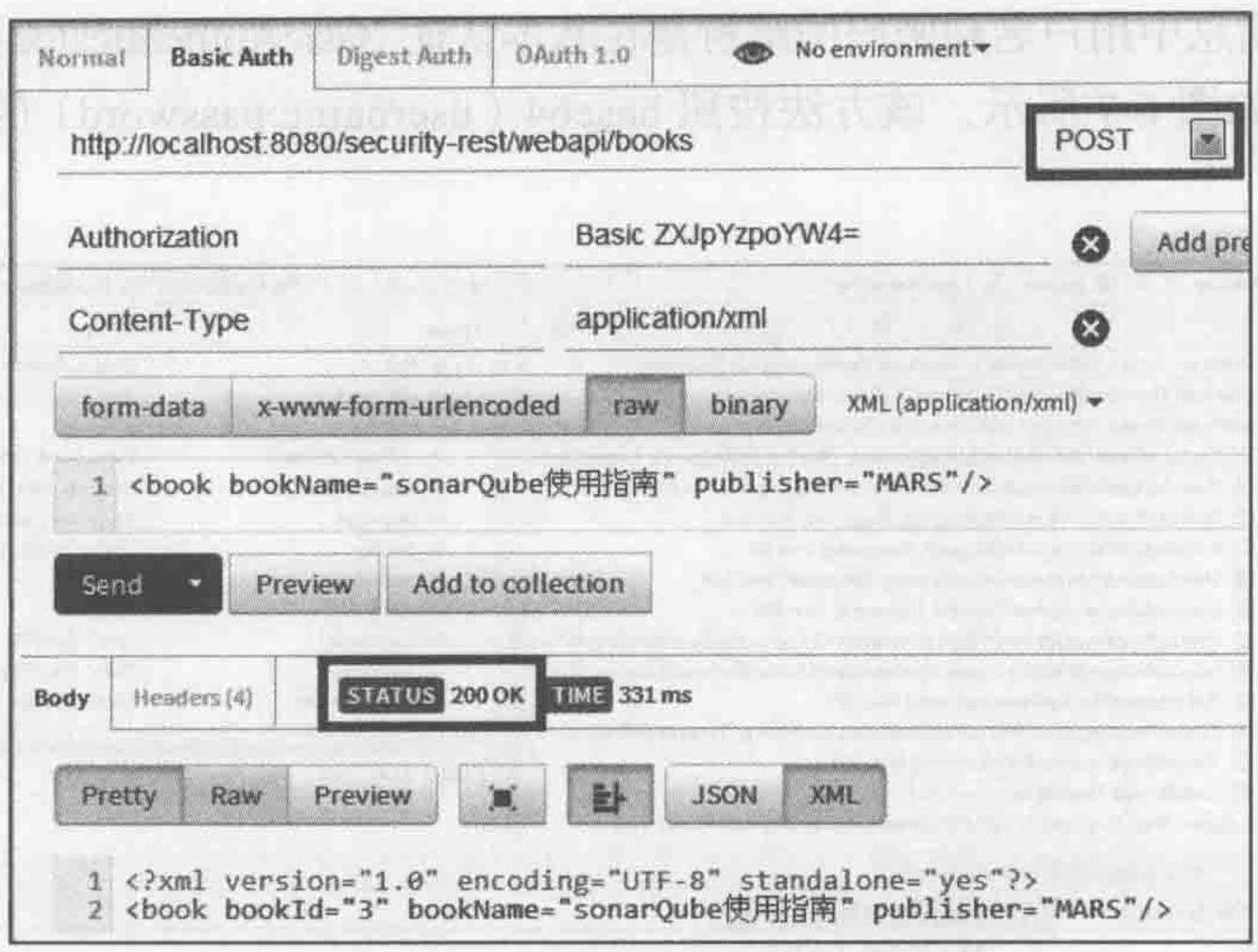


图 6-5 基本认证测试用例 4 结果

到此，我们通过 4 个测试用例对示例的场景进行了覆盖性的功能测试，本示例如预期完成了既定功能。最后，通过断点调试了解一下 Tomcat 是如何使用 JDBCRealm 完成 HTTP 基本认证功能的。

(6) 基本认证

阅读指导 本例使用的 Tomcat 版本为 7.0.42，Maven 仓库源代码地址为 C:\Users\hanl.m2\repository\org\apache\tomcat\tomcat-catalina\7.0.42\tomcat-catalina-7.0.42-sources.jar。

以 Debug 的方式在 IDE 中启动本示例，并使用上述的“测试用例 4”向服务器发出请求。如图 6-6 所示，通过断点的栈信息可以看到，JDBCRealm 类的 authenticate() 方法最终处理客户端请求信息和数据库身份信息的匹配工作。

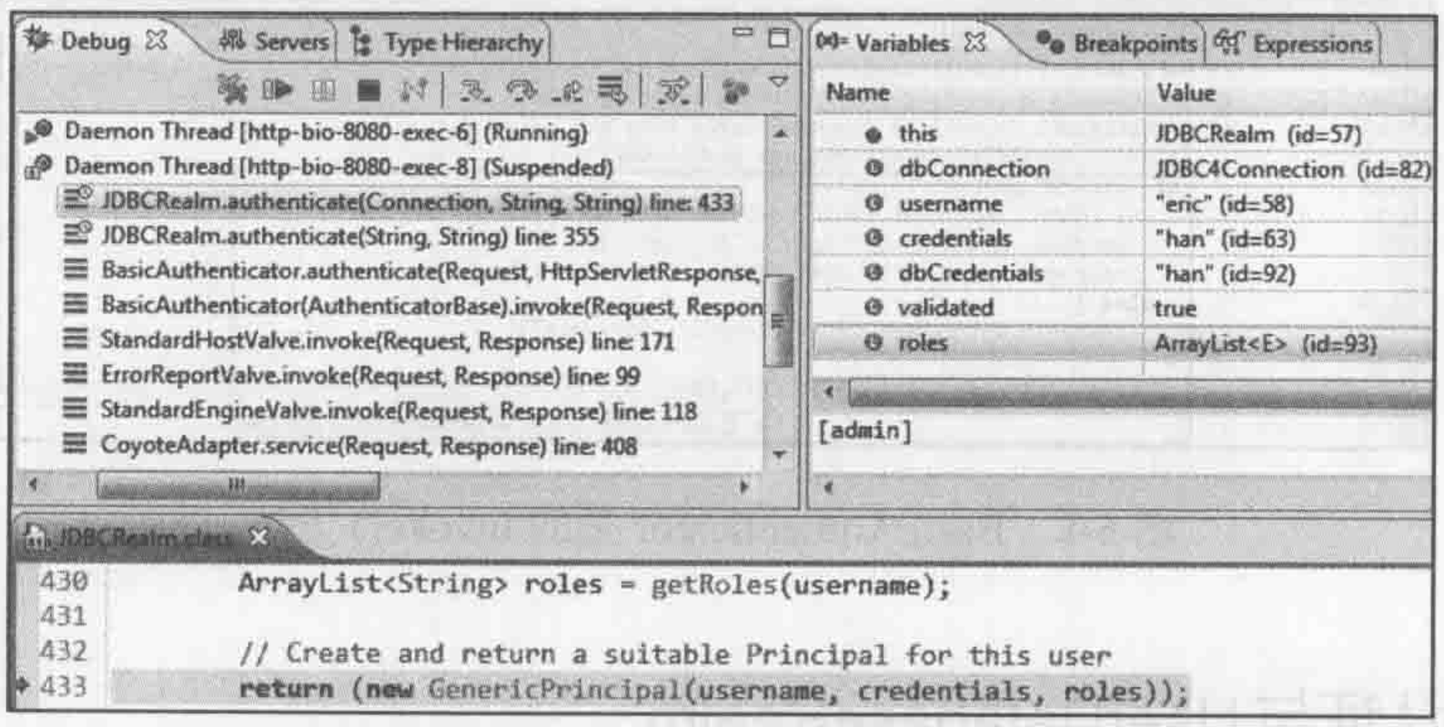


图 6-6 JDBCRealm 类处理基本认证

客户端请求信息中用户名和密码的解析是在基本认证 BasicAuthenticator 类的 authenticate() 方法中完成的，如图 6-7 所示，该方法按照 base64（username:password）的格式对字符串进行了解码。

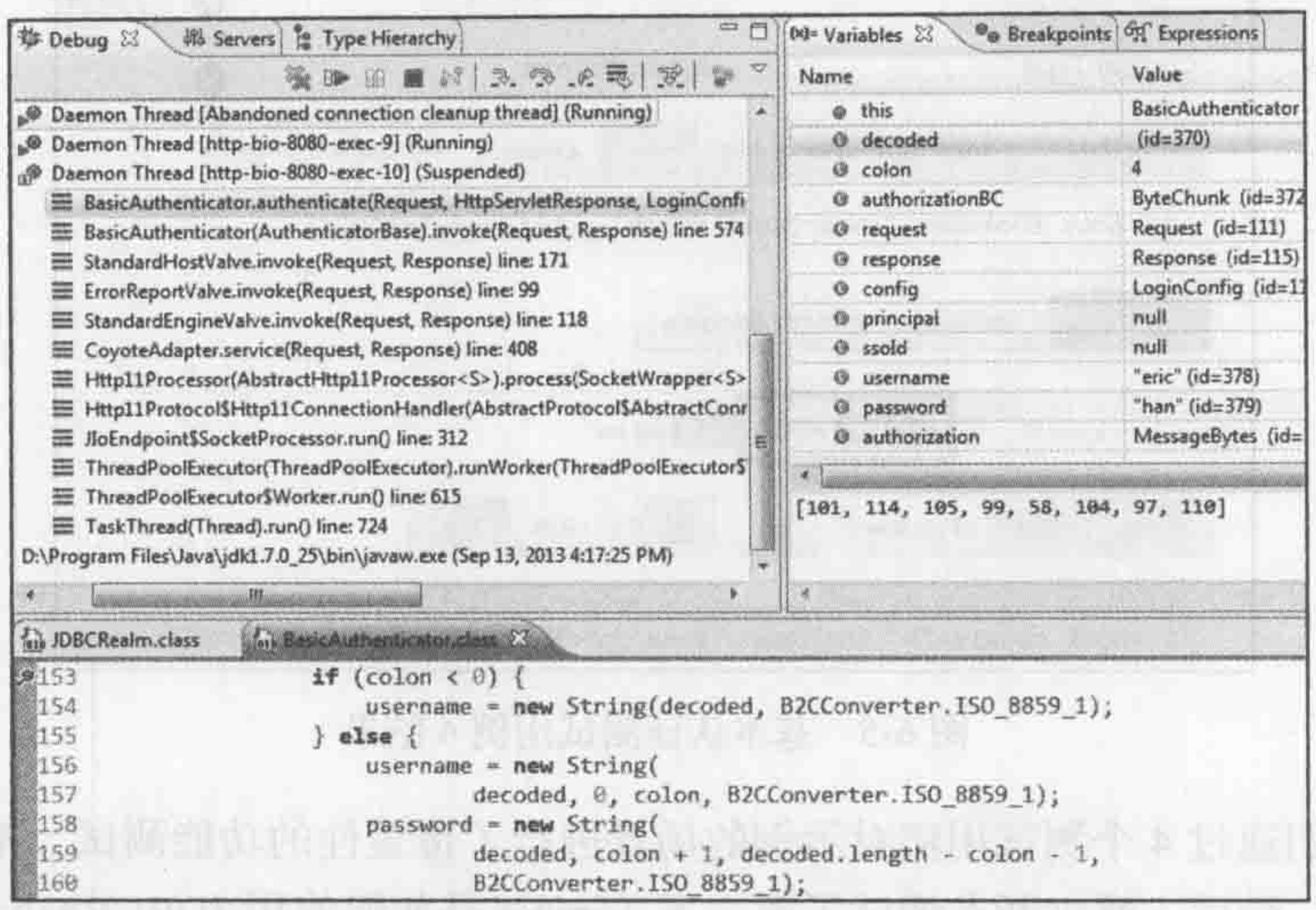


图 6-7 BasicAuthenticator 类的 authenticate() 方法

最后，如图 6-8 所示，BasicAuthenticator 类的 invoke() 方法对响应头进行了修改。

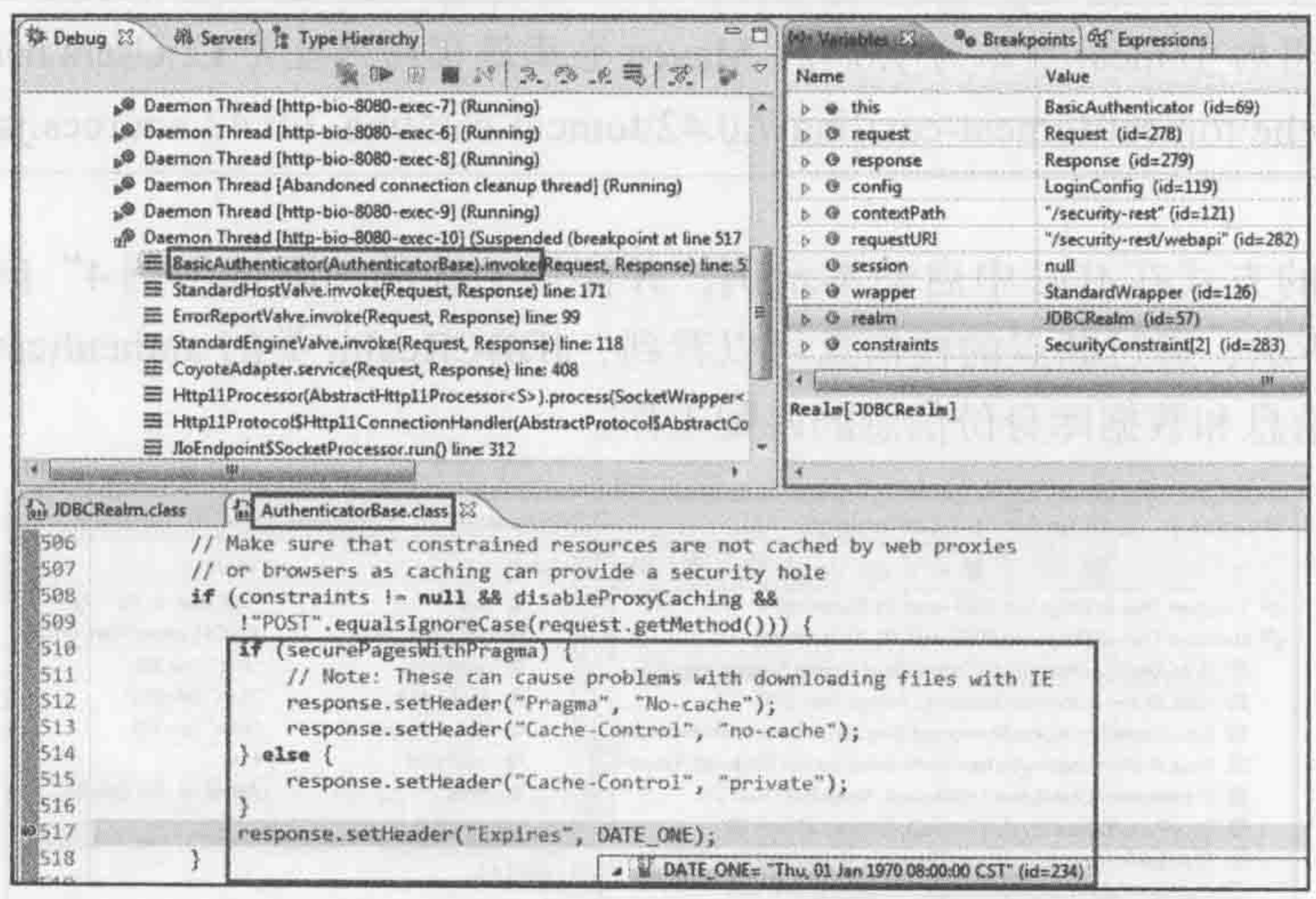


图 6-8 BasicAuthenticator 类的 invoke() 方法

6.3.2 摘要认证与 UserDatabaseRealm

本示例使用 HTTP 摘要认证结合 Tomcat 提供的 UserDatabaseRealm 实现认证与授权。为

了简化示例逻辑，本例所示的授权场景和前例基本相同，不同的是，UserDatabaseRealm 通过读取 XML 格式的 JNDI 资源（默认使用 conf/tomcat-users.xml），获取认证信息。因此本例不需要数据库的支持。

（1）配置 UserDatabaseRealm

同前例，服务器配置文件 $CATALINA_BASE/conf/server.xml 添加 Realm，其资源通过名称匹配在 Resource 中定义的 XML 文件。

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="tom" />
<Resource auth="Container" description="User database that can be updated and
saved" factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
name="tom" pathname="conf/tomcat-users.xml" type="org.apache.catalina.UserDatabase" />
```

同前例，Eclipse 内置的 Tomcat 配置如图 6-9 所示。

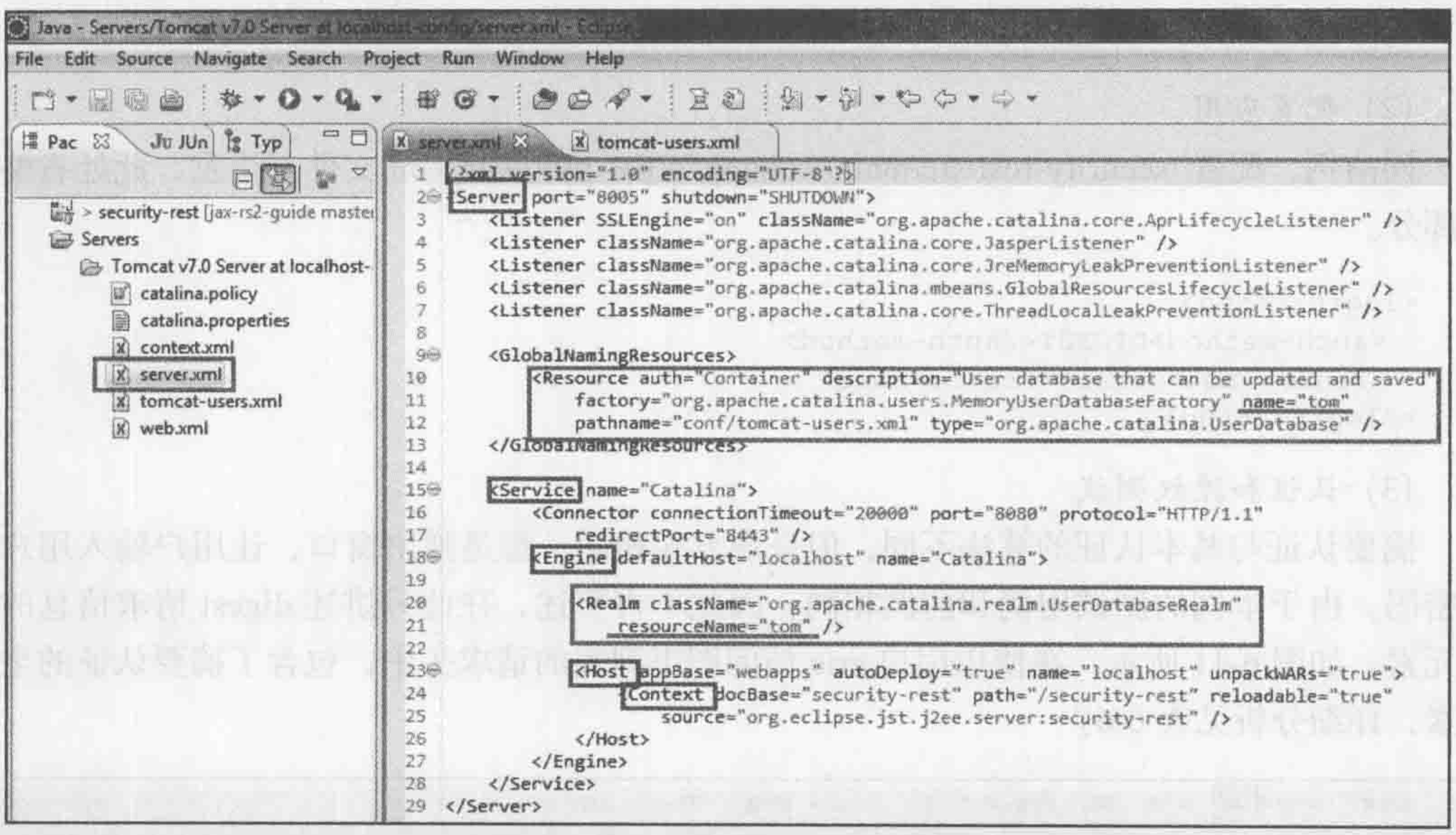


图 6-9 在 Eclipse 中为 Tomcat 插件配置 Realm

Resource 中定义的 XML 文件为 $CATALINA_BASE/conf/tomcat-users.xml，按照前述的逻辑，定义如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
  <role rolename="admin" />
  <role rolename="user" />
  <user name="eric" password="han" roles="admin" />
  <user name="caroline" password="zhang" roles="user" />
</tomcat-users>
```

Eclipse 内置 Tomcat tomcat-users 配置如图 6-10 所示。

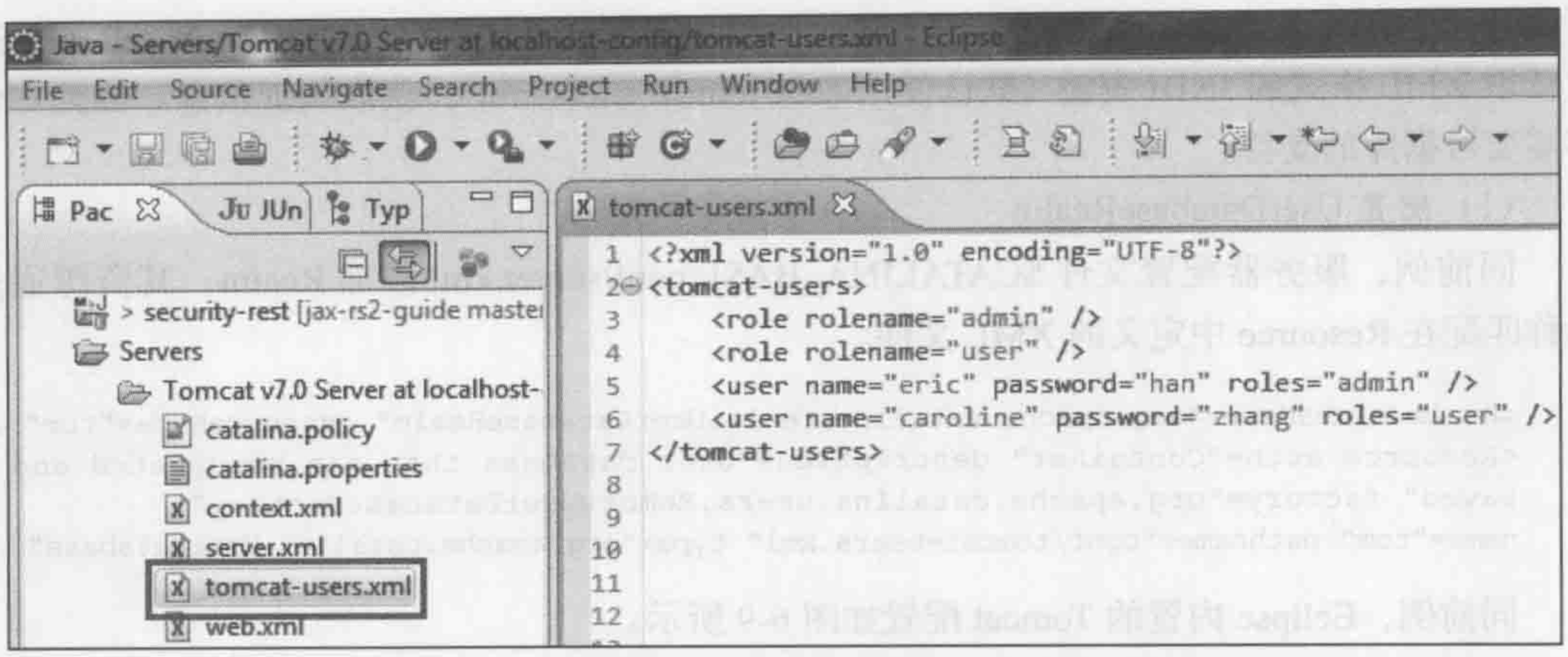


图 6-10 在 Eclipse 中为 Tomcat 插件配置用户权限

(2) 配置应用

同前例，配置 /security-rest/src/main/webapp/WEB-INF/web.xml 文件。注意，此处省略相同部分。

```
<login-config>
  <auth-method>DIGEST</auth-method>
  <realm-name>drealm</realm-name>
</login-config>
```

(3) 认证和授权测试

摘要认证与基本认证的算法不同，但呈现方式相同，都是弹出窗口，让用户输入用户名和密码。由于本例的测试用例和前例相同，因此不再赘述，在此只讲述 digest 请求信息的相关元素。如图 6-11 所示，在使用用户 eric 访问图书列表的请求头中，包含了摘要认证的全部元素，详细分析见表 6-6。

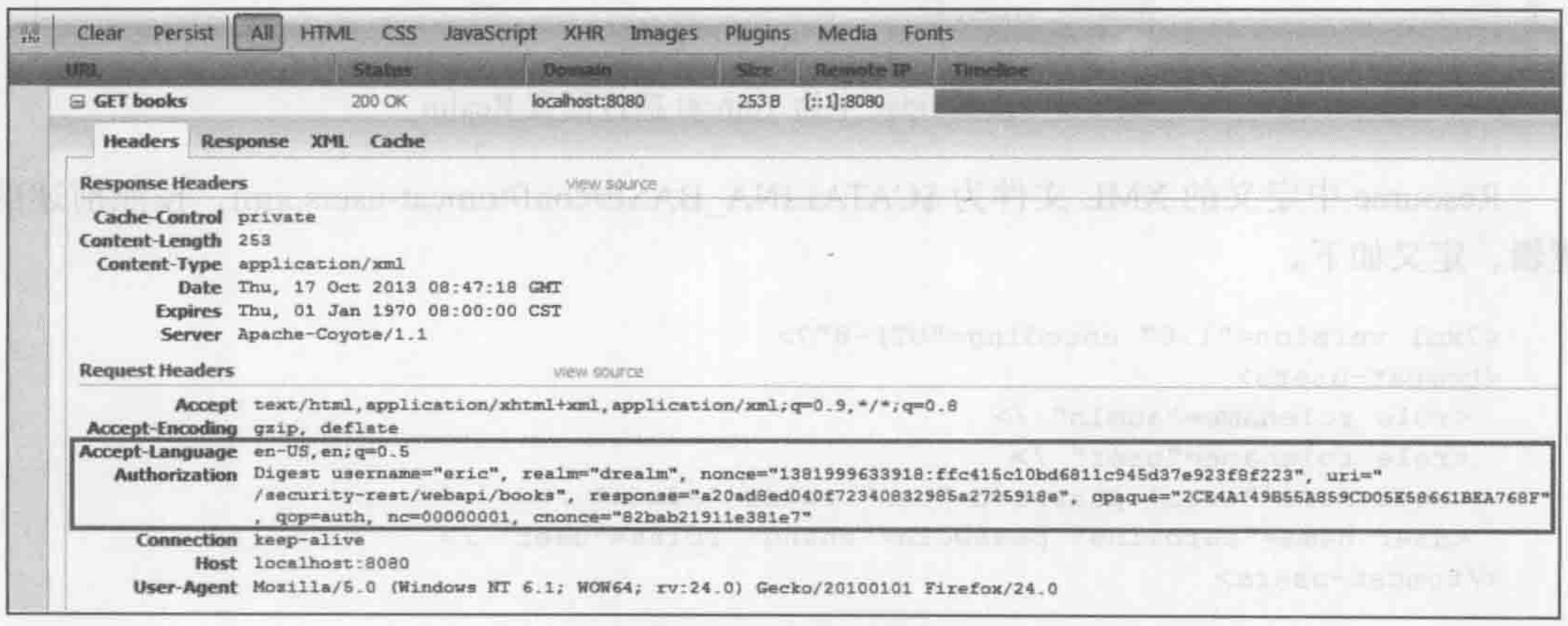


图 6-11 摘要认证头信息

表 6-6 摘要认证参数列表

认证条目	简述
username="eric"	用户名
realm="drealm"	Realm 名称
nonce="1381999633918:ffc415c10bd6811c945d37e923f8f223"	服务器端随机数
uri="/security-rest/webapi/books"	REST 服务资源地址
response="a20ad8ed040f72340832985a2725918e"	MD5（HA1:nonce:HA2）计算值
opaque="2CE4A149B55A859CD05E58661BEA768F"	服务器端质询响应信息
qop=auth	保护质量
nc=00000001	客户端请求计数器
cnonce="82bab21911e381e7"	客户端随机数

（4）摘要认证

如图 6-12 所示，通过断点栈可以观察到，UserDatabaseRealm 类的 authenticate() 方法完成了摘要认证的算法，具体为 md5（md5（username:realm:password）:nonce:nc:cnonce:qop:md5（httpmethod:uri））。可参考摘要认证类 DigestAuthenticator。

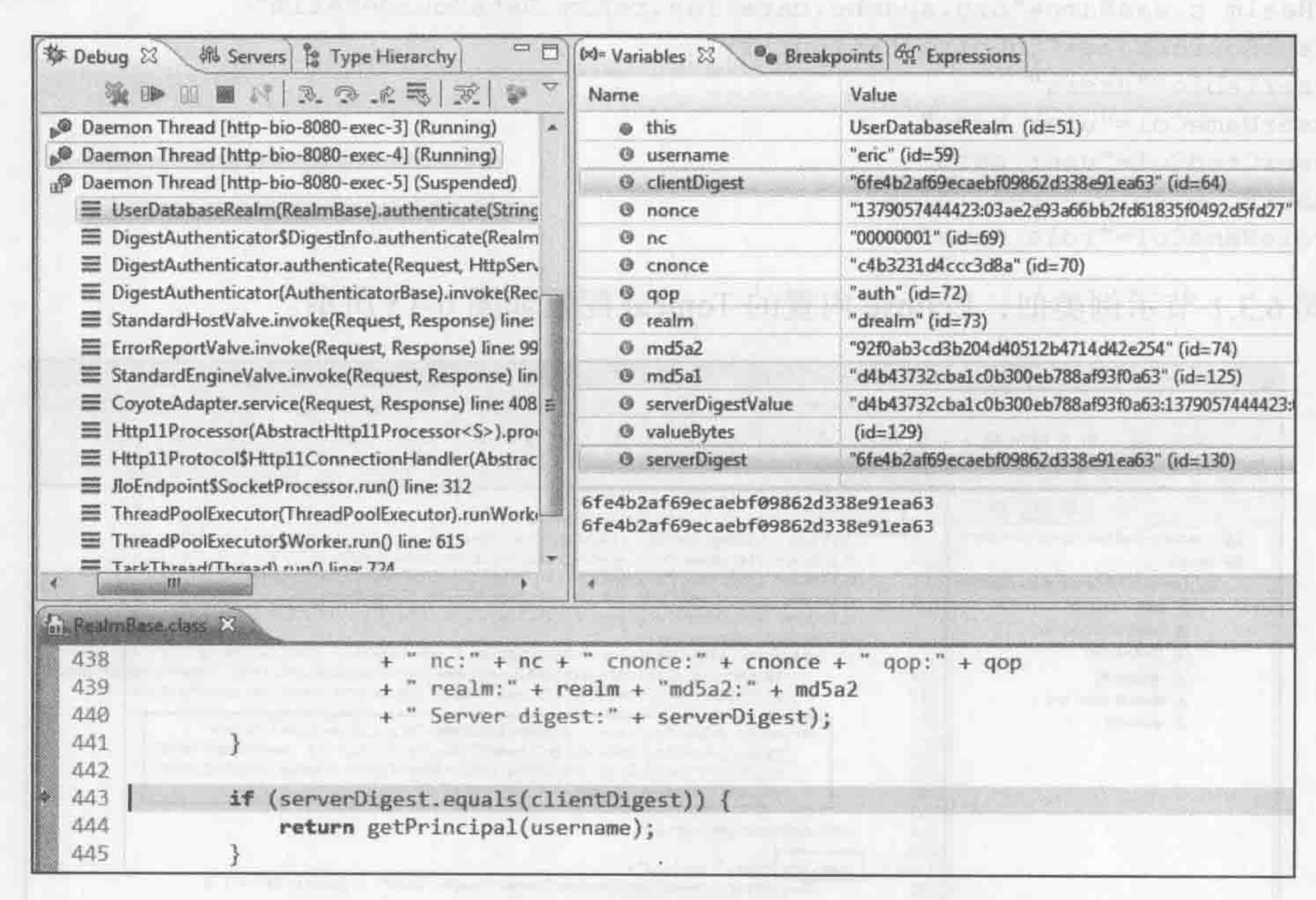


图 6-12 UserDatabaseRealm 类的 authenticate() 方法

6.3.3 表单认证与 DataSourceRealm

本示例使用表单认证，并结合 Tomcat 提供的 DataSourceRealm 类实现认证与授权。为了简化示例逻辑，本例所示的授权场景和前例相同。DataSourceRealm 通过 JNDI 访问关系型数据库，获取认证信息。DataSourceRealm 与 UserDatabaseRealm 相比，数据库连接使用容器

提供的全局数据源（Data Source），而不是在 Realm 中自行配置。

我们首先需要创建 Realm 所需的数据表，同 6.3.1 节的数据库脚本，使用“mysql -uroot -p < security.sql”命令导入。

（1）配置 DataSourceRealm

同前例，服务器配置文件 $CATALINA_BASE/conf/server.xml 添加 Realm，首先配置容器的全局数据源。

```
<Resource auth="Container"
driverClassName="org.gjt.mm.mysql.Driver"
type="javax.sql.DataSource"
name="jdbc/rest-security"
username="root" password="root"
url="jdbc:mysql://localhost:3306/simple_service_book"
validationQuery="select 1 from users"/>
```

然后，设置 DataSourceRealm 并使用该数据源。

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
dataSourceName="jdbc/rest-security"
userTable="users"
userNameCol="user_name"
userCredCol="user_pass"
userRoleTable="user_roles"
roleNameCol="role_name"/>
```

同 6.3.1 节示例类似，Eclipse 内置的 Tomcat 配置如图 6-13 所示。

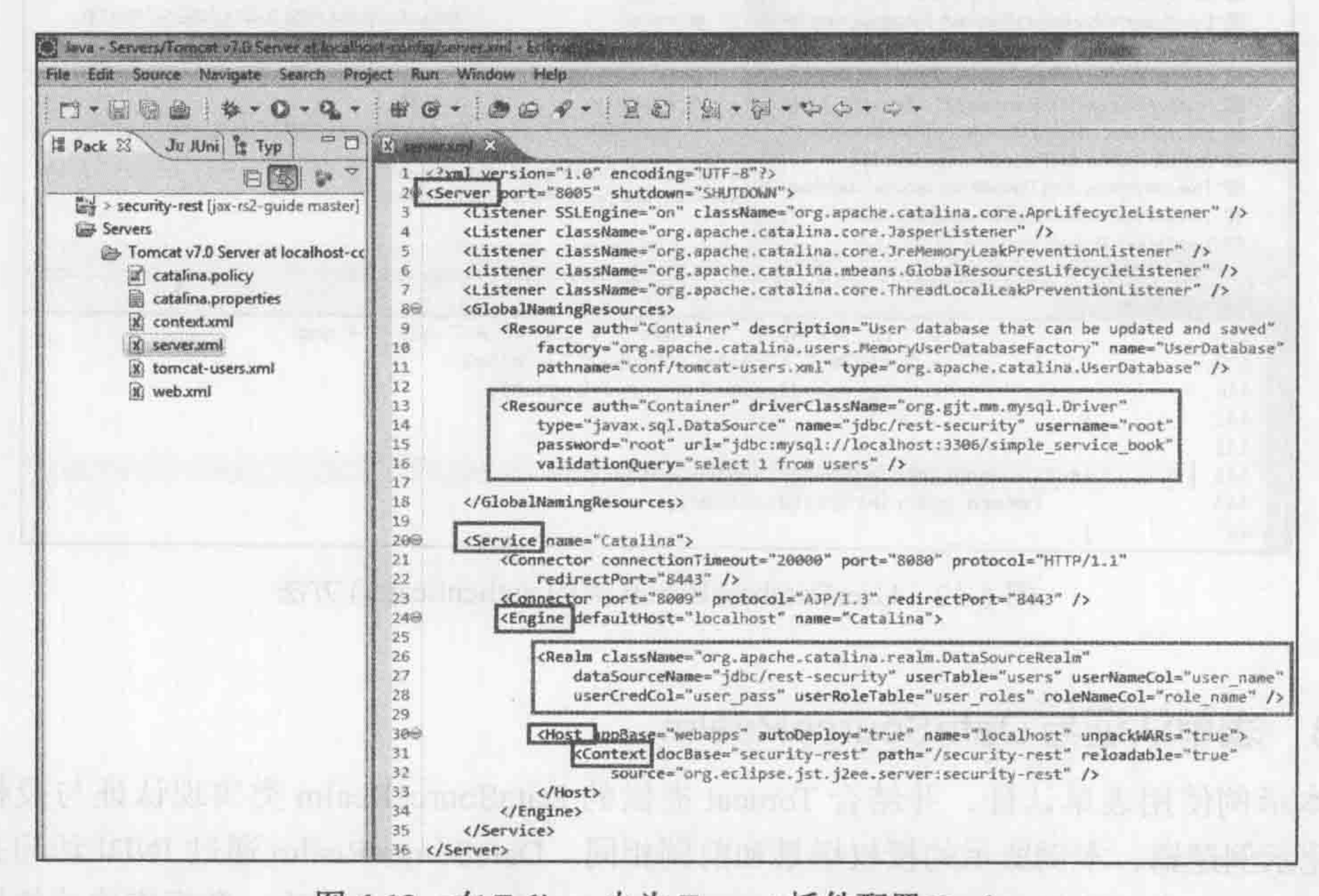


图 6-13 在 Eclipse 中为 Tomcat 插件配置 Realm

(2) 数据库驱动

同 6.3.1 节示例，复制 MySQL 的 JDBC 驱动到 $CATALINA_HOME/lib 目录。

(3) 配置应用

同 6.3.1 节示例，配置 /security-rest/src/main/webapp/WEB-INF/web.xml 文件。注意此处省略相同部分。

```
<resource-ref>
  <description>MySQL DB Connection Pool</description>
  <res-ref-name>jdbc/rest-security</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
<welcome-file-list>
  <welcome-file>/index.html</welcome-file>
</welcome-file-list>
```

(4) 登录页面

创建并编辑登录页面 /security-rest/src/main/webapp/login.html。

```
<form action="j_security_check">
  <div>
      <span>User Name</span>
      <input id="j_username" name="j_username" type="text" />
  </div>
  <div style="margin-top:10px;margin-bottom:10px;">
      <span>Pass Word</span>
      <input id="j_password" name="j_password" type="password" />
  </div>
  <input type="submit" value="Sign In" />
</form>
```

如图 6-14 所示，启动服务后，访问 http://localhost:8080/security-rest/webapi/books，页面将跳转至登录页面。

下面是详细的测试用例。

(5) 认证和授权测试

测试用例 1（见表 6-7），访问地址为 http://localhost:8080/security-rest/，页面跳转至登录页面。

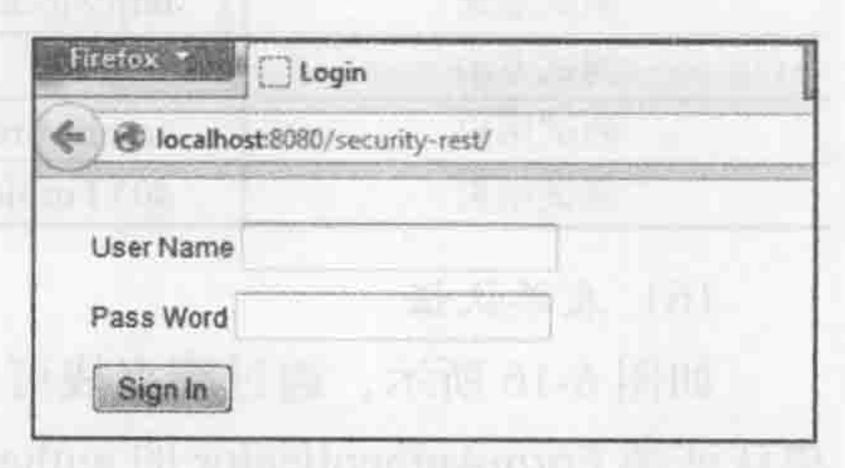


图 6-14 登录页面

表 6-7 摘要认证测试用例 1

测试要素	说明
测试地址	http://localhost:8080/security-rest/
测试方法	FORM jsecuritycheck
测试用户	eric role=admin
测试结果	302 Found

当用户信息提交后，服务器处理登录信息返回 HTTP 状态码 302（Found），重定向请求，然后页面跳转到首页 index.html，返回 200 OK，如图 6-15 所示。

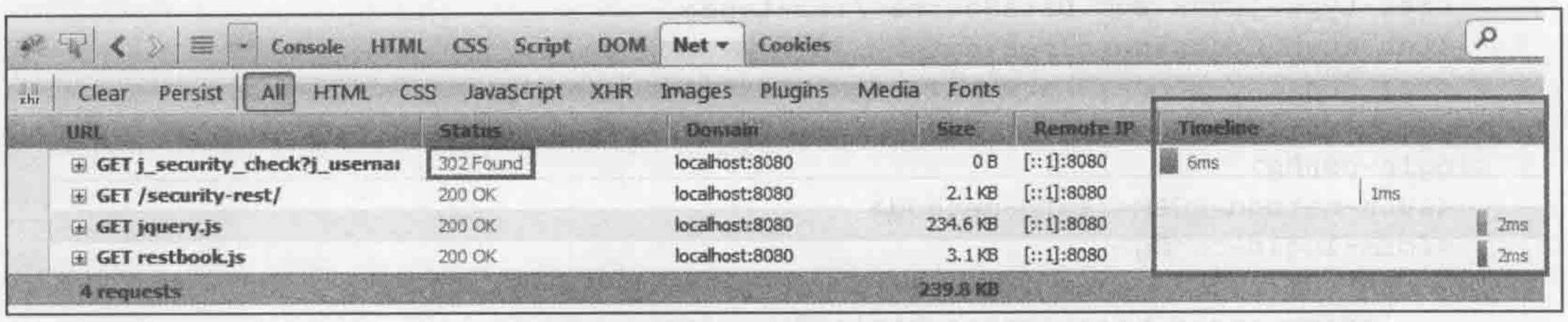


图 6-15 摘要认证测试用例 1 结果

测试用例 2（见表 6-8），使用 admin 角色登录后，测试是否具有访问创建新图书的资源地址的权限。结果返回 HTTP 状态码 200 OK。

表 6-8 摘要认证测试用例 2

测试要素	说明
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	POST
测试用户	eric role=admin
测试结果	200 OK

测试用例 3（见表 6-9）使用 user 角色登录后，测试是否具有访问创建新图书的资源地址的权限。结果返回 HTTP 状态码 403 Forbidden。

表 6-9 摘要认证测试用例 3

测试要素	说明
测试地址	http://localhost:8080/security-rest/webapi/books
测试方法	POST
测试用户	caroline role=user
测试结果	403 Forbidden

（6）表单认证

如图 6-16 所示，通过断点栈可以观察到，DataSourceRealm 类的 authenticate() 方法被表单认证类 FormAuthenticator 的 authenticate() 调用，匹配请求信息和数据库认证信息。

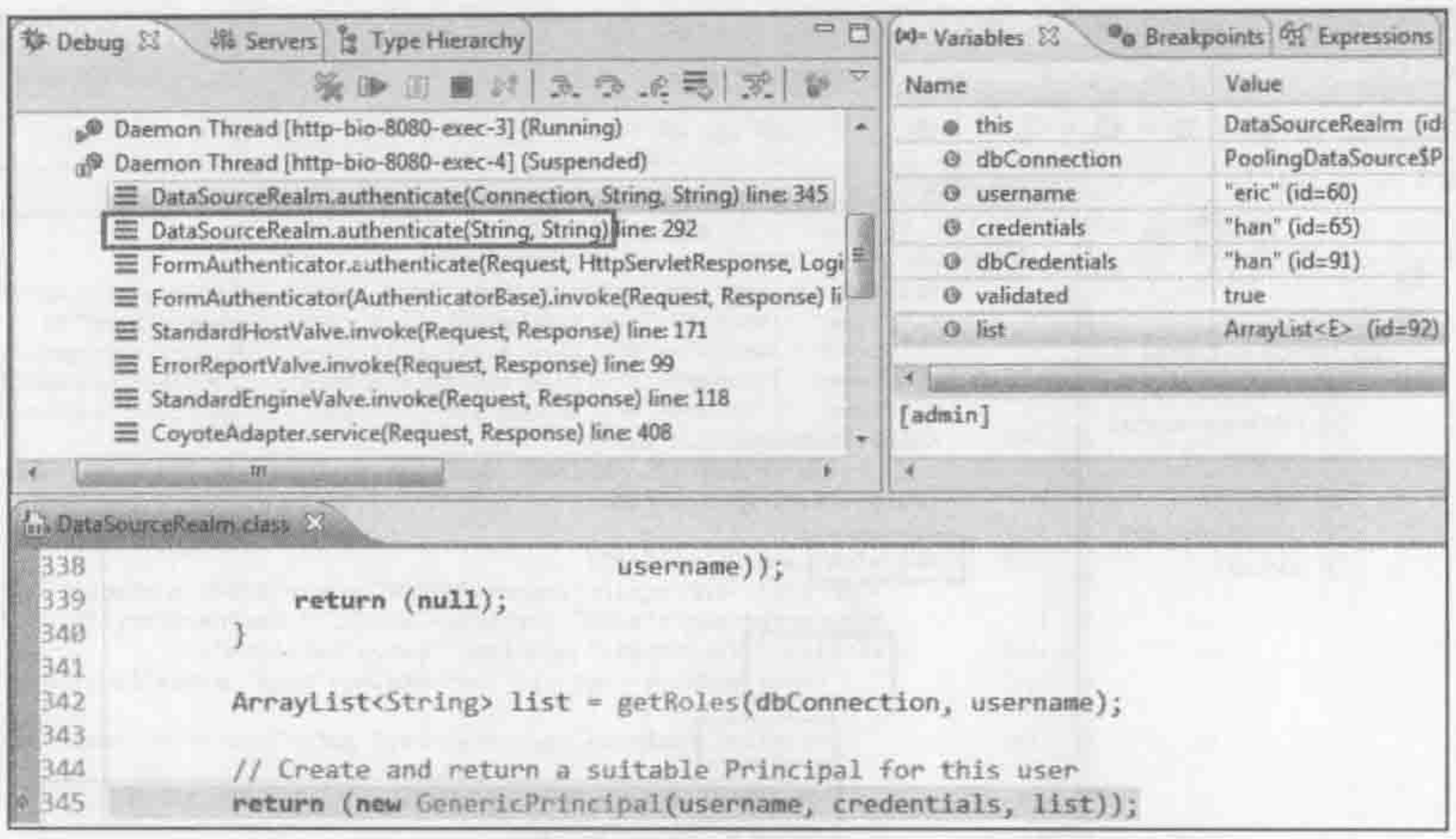


图 6-16 DataSourceRealm 类的 authenticate() 方法

6.3.4 表单认证与 JAASRealm

JAAS（Java Authentication & Authorization Service，Java 认证和授权服务）是 Java 平台认证和授权的标准规范（JSR 196 标准），JAAS 只能使用表单认证方式获取用户登录信息。Tomcat 提供了基于 JAAS 的 JAASRealm，本示例将使用表单认证和 JAASRealm 展示一个 JAAS 认证授权的实现过程。

（1）创建 Realm 所需的数据表

同 6.3.1 节的数据库脚本，使用“mysql -uroot -p < security.sql”命令导入。

（2）配置 JAASRealm

配置 $CATALINA_BASE/conf/server.xml，添加 JAASRealm 的定义。

```
<Context docBase="security-rest" ……>
<Realm className="org.apache.catalina.realm.JAASRealm"
    appName="RestJaasRealm"
    roleClassNames="com.example.jaas.RestRolePrincipal"
    userClassNames="com.example.jaas.RestUserPrincipal"/>
</Context>
```

Realm 定义在 context 中，否则会导致角色 POJO 字段 roleClassNames 和用户 POJO 字段 userClassNames 中定义的类找不到。appName 定义的名字和 JAAS 配置文件 restJaas.conf 中定义的必须一致。

在 Eclipse 中，其内置的 Tomcat JAASRealm 配置如图 6-17 所示。

（3）JAAS 配置文件

实现 JAAS 需要为 JAAS 框架提供一个配置文件，本例使用 /security-rest/src/main/resources/restJaas.conf。

```
RestJaasRealm{ com.example.jaas.RestLoginModule required; };
```

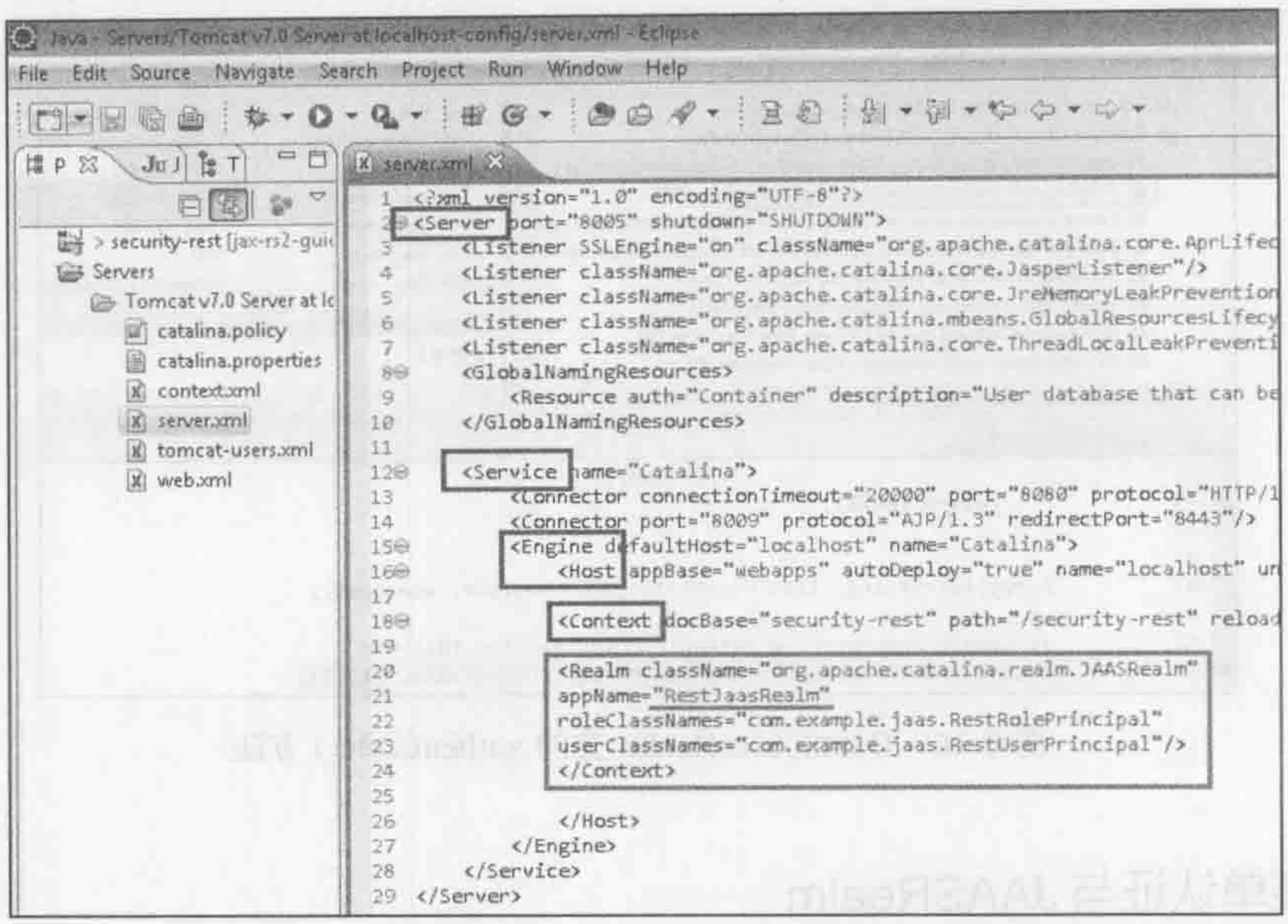


图 6-17 在 Eclipse 中为 Tomcat 插件配置 JAASRealm

名称 RestJaasRealm 与图 6-17 中的 appName 必须一致。其中第一个参数是登录模块的类全名，第二个参数的取值见表 6-10。

表 6-10 JAAS 配置行第二个参数取值和说明

参数值	说明
Required	该模块必须认证用户，如果认证失败，那么使用其他登录模块认证
Requisite	如果认证失败，将终止认证
Sufficient	如果认证成功，即获得登录认证；如果认证失败，使用其他登录模块认证
Optional	认证将继续下去，即使该模块认证成功

(4) JAAS 实现类

JAAS 的实现类目录为 jax-rs2-guide\sample\6\security-rest\src\main\java\com\example\jaas，其中定义了 4 个 JAAS 实现类，见表 6-11。

表 6-11 JAAS 实现类列表

实现类作用	实现类文件名
LoginModule 实现类	RestLoginModule.java
LoginModule 实现类的数据库操作类	RestLoginDao.java
Role 接口 POJO 类	RestRolePrincipal.java
User 接口 POJO 类	RestUserPrincipal.java

(5) 配置 JVM 启动参数

前面设置的 JAAS 配置文件需要加入启动参数中，以便 JAAS 框架识别。

```
-Djava.security.auth.login.config="D:\+aries\github\jax-rs2-guide\sample\6\
security-rest\src\main\resources\restJaas.conf"
```

Eclipse 内置 Tomcat 虚拟机参数设置如图 6-18 所示。

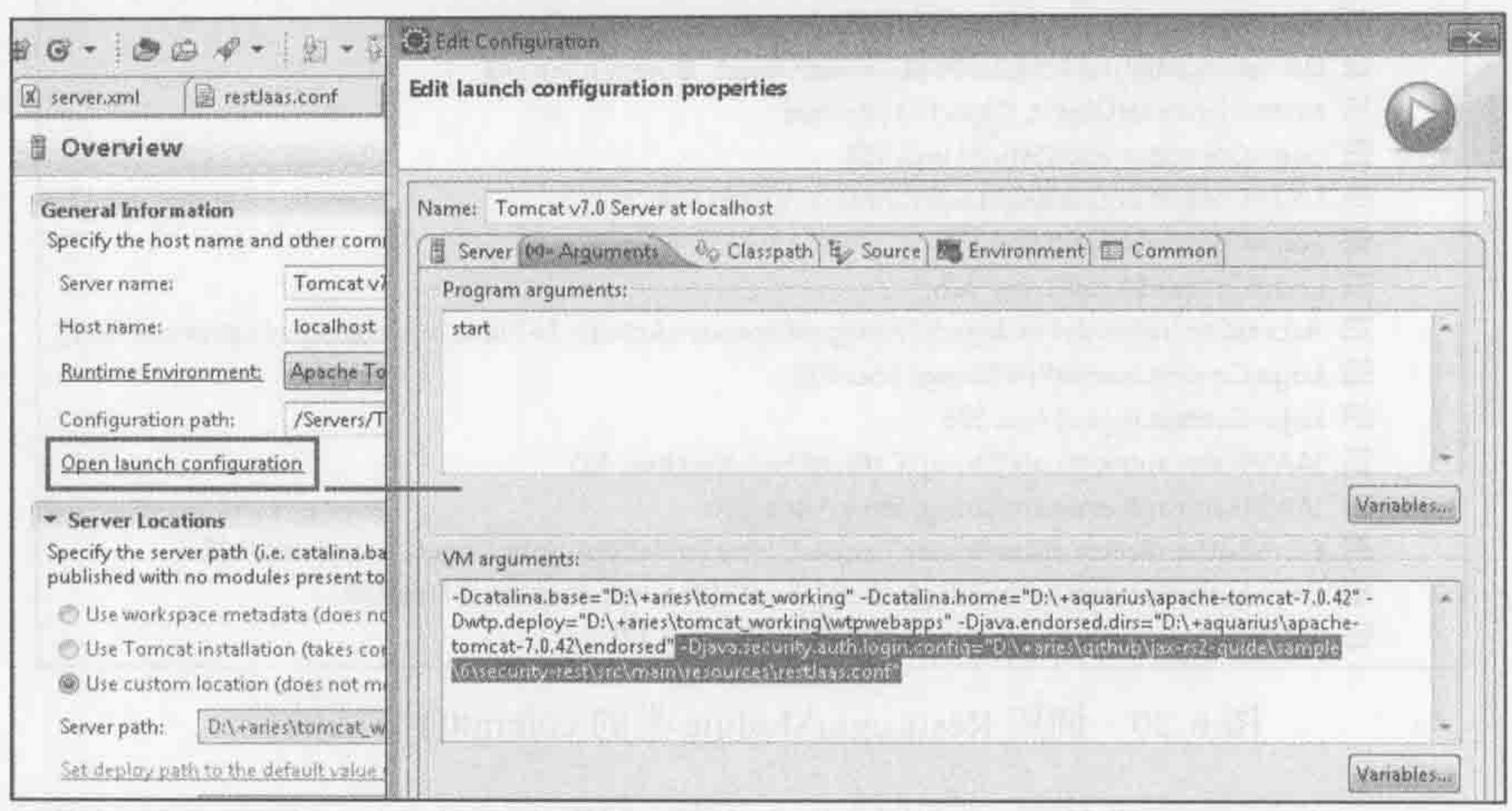


图 6-18 在 Eclipse 中为 Tomcat 插件配置支持 JAAS 的启动参数

(6) JAAS 流程

在 JAASRealm.authenticate 认证方法中，有以下两个主要对象。

- CallbackHandler：持有登录信息的回调。
- LoginContext：通过配置文件感知 LoginModule 实现类的上下文。

认证分为两个步骤：

第一步，验证登录信息合法性对应 RestLoginModule 类的 login() 方法，如图 6-19 所示。

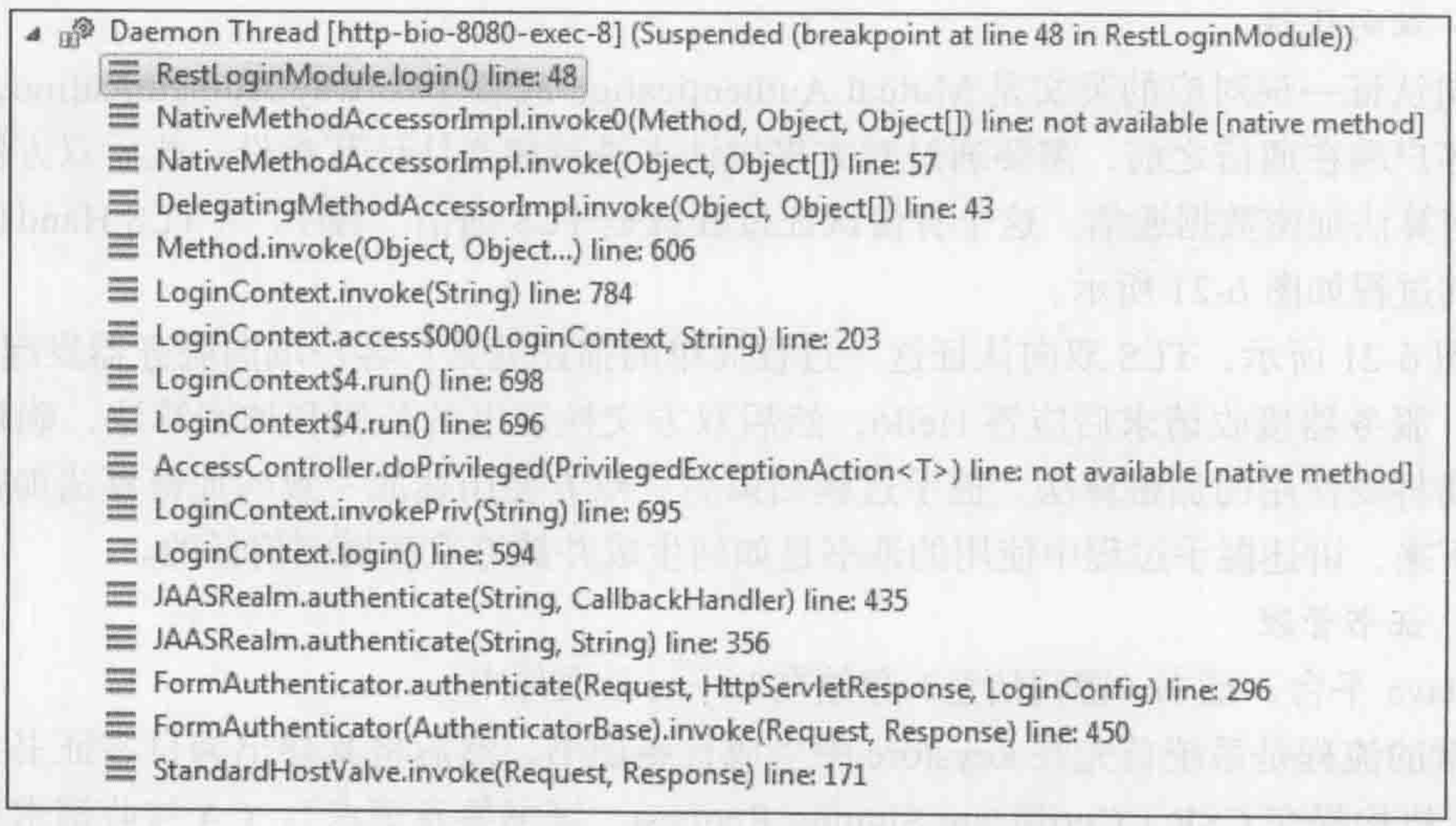


图 6-19 调用 RestLoginModule 类的 login() 方法的栈

第二步，获取登录身份信息对应 RestLoginModule 类的 commit() 方法，如图 6-20 所示。

```
Daemon Thread [http-bio-8080-exec-8] (Suspended (breakpoint at line 79 in RestLoginModule))
    RestLoginModule.commit() line: 79
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
    Method.invoke(Object, Object...) line: 606
    LoginContext.invoke(String) line: 784
    LoginContext.access$000(LoginContext, String) line: 203
    LoginContext$4.run() line: 698
    LoginContext$4.run() line: 696
    AccessController.doPrivileged(PrivilegedExceptionAction<T>) line: not available [native method]
    LoginContext.invokePriv(String) line: 695
    LoginContext.login() line: 595
    JAASRealm.authenticate(String, CallbackHandler) line: 435
    JAASRealm.authenticate(String, String) line: 356
    FormAuthenticator.authenticate(Request, HttpServletResponse, LoginConfig) line: 296
    FormAuthenticator(AuthenticatorBase).invoke(Request, Response) line: 450
    StandardHostValve.invoke(Request, Response) line: 171
```

图 6-20 调用 RestLoginModule 类的 commit() 方法的栈

到此，JAAS 在 REST 式的 Web 服务中的应用就讲述完毕了，希望读者对此有清楚的认识。还要说明的是，JAAS 这种验证实现有些重型，对于轻量级的 REST 应用以及多模块分布式的 Web 服务而言，JAAS 验证的实现并不令人满意。

6.3.5 证书认证与 UserDatabaseRealm

6.1 节介绍了证书认证的技术背景，证书认证包括对客户端（有时是对服务器）的单项认证和双向认证。本节结合 Tomcat 提供的 UserDatabaseRealm，采用双向认证来演示证书认证的实现。

（1）双向认证

双向认证一词对应的英文是 Mutual Authentication 或者 Two way Authentication，是指服务器和客户端在通信之前，需要通过对方提供证书进行相互认证其身份，此后双方使用协商好的加密算法加密数据通信。这个身份认证过程就是 TLS 通信“握手”（TLS Handshake）的过程。其过程如图 6-21 所示。

如图 6-21 所示，TLS 双向认证这一过程简单的描述就是：客户端向服务器发起 Hello 请求会话，服务器接收请求后应答 Hello，然后双方交换证书的公钥和加密算法，确认对方身份并协商将要使用的加密算法。握手过程结束后，双方使用达成一致的加密算法加密数据通信。接下来，讲述握手过程中使用的证书是如何生成并被签发和授权信任的。

（2）证书管理

在 Java 平台，证书（密钥信息）存储在 keystore 文件中。

通常的流程是系统首先在 keystore 中生成自签证书，然后将其导出为自签证书文件，接着向 CA 机构提交 CSR（Certificate Signing Request，证书签发请求）。CA 接收请求后，使用

私钥签发该证书，并将 CA 签发的证书和 CA 证书（包含 CA 公钥）返回。系统将 CA 证书导入 keystore，即信任 CA，此后 TLS 通信过程中，凡是该 CA 签发的证书，该系统皆信任，如图 6-22 所示。CA 签发的证书在该系统与外系统通信时，作为该系统身份认证证书提交。

简化的流程是省略 CA 签发证书这个流程，双方的自签证书可以作为通信时身份认证的证书，即在通信前，外系统已经导入该系统的自签证书，对其信任。

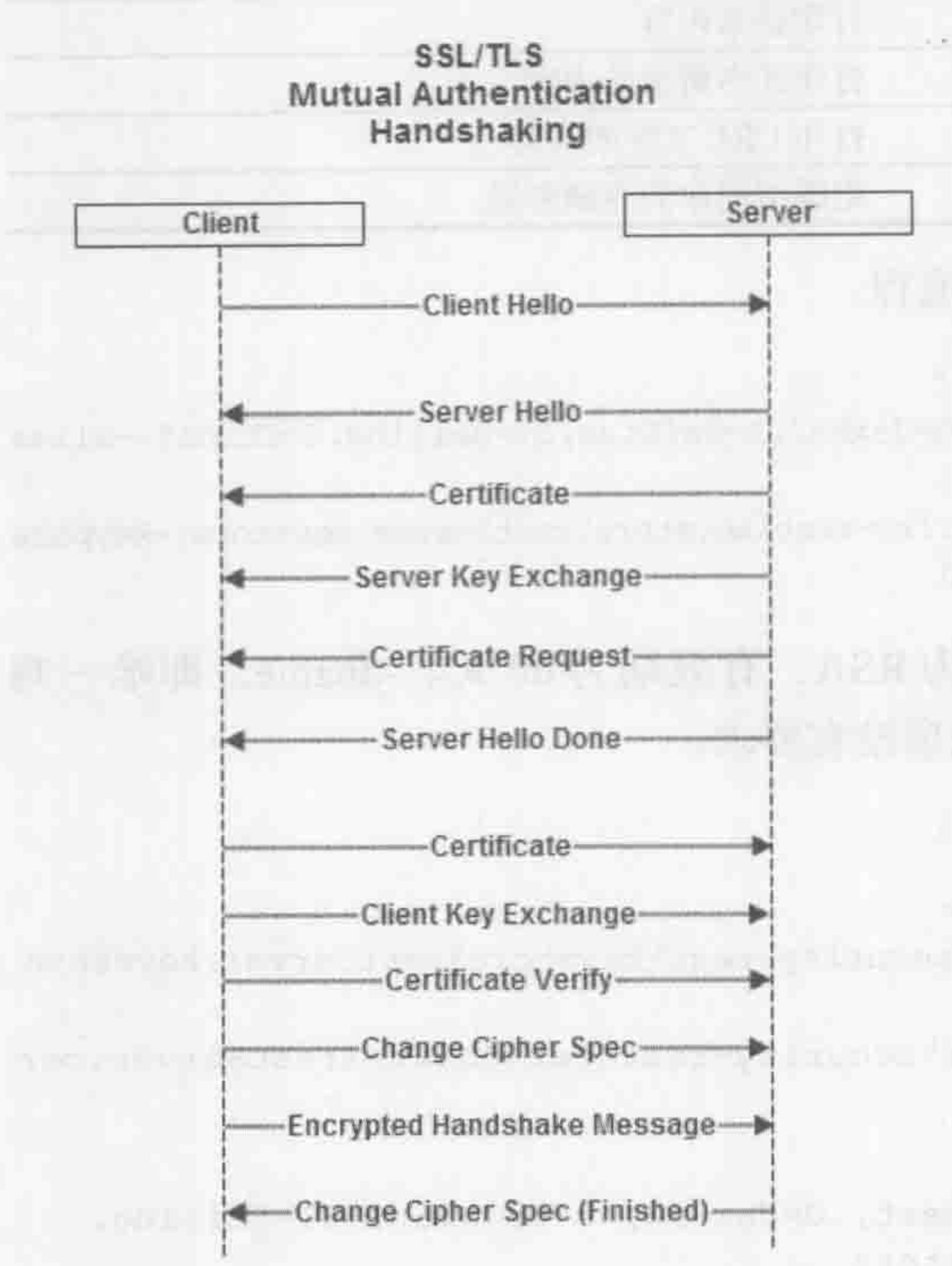


图 6-21 TLS 双向认证握手流程

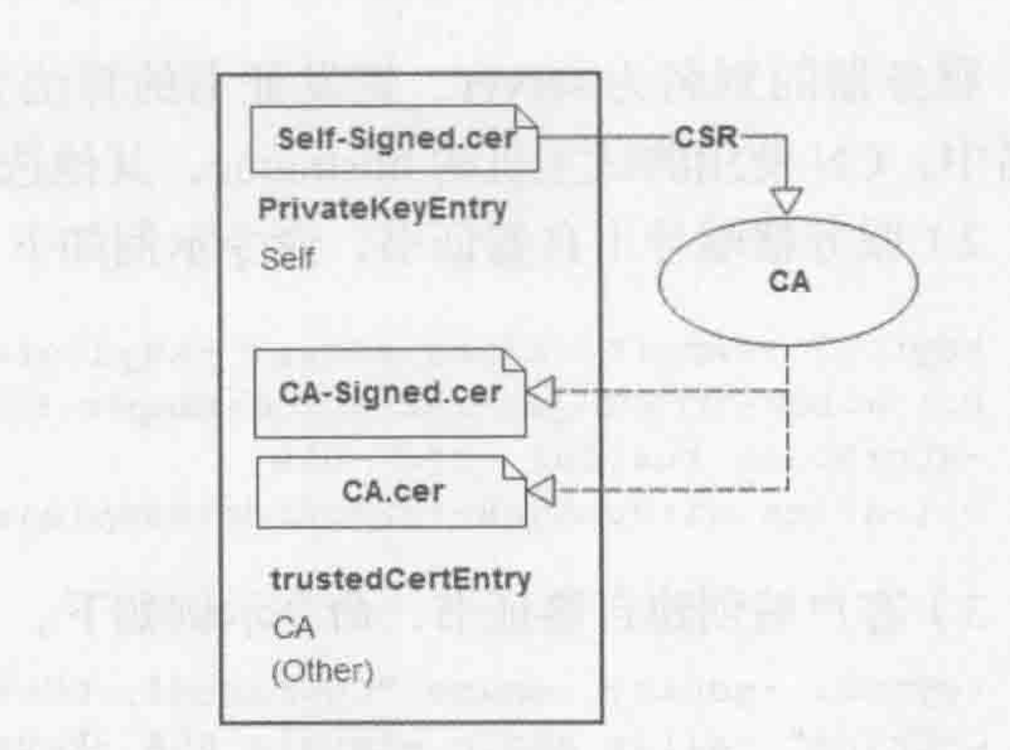


图 6-22 证书签发流程

（3）密钥和证书管理工具

JDK 提供了密钥和证书管理工具 keytool，该工具实现的功能和 JCE 规范提供的 API 类似。通过“keytool help”命令可以得到表 6-12 所示的参数信息。

表 6-12 keytool 工具参数列表

参数	功能
-certreq	生成证书请求
-changealias	更改条目的别名
-delete	删除条目
-exportcert	导出证书
-genkeypair	生成密钥对
-genseckey	生成密钥
-gencert	根据证书请求生成证书

（续）

参数	功能
-importcert	导入证书或证书链
-importkeystore	从其他密钥库导入一个或所有条目
-keypasswd	更改条目的密钥密码
-list	列出密钥库中的条目
-printcert	打印证书内容
-printcertreq	打印证书请求的内容
-printcrl	打印 CRL 文件的内容
-storepasswd	更改密钥库的存储密码

通过 keytool，我们可以完成下列证书管理流程。

1）服务器端创建自签证书，命令示例如下。

```
keytool -genkey -dname "CN=mars64,OU=Rest,O=JaxRs2,L=HaiDian,ST=Beijing,C=China" -alias
server -keyalg RSA -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.keystore -keypass
restful -storepass restful -validity 60
```

服务器的别名为 server，签发证书的算法为 RSA，有效期为 60 天。-dname，即唯一判别名中，CN 使用测试主机的 hostname，其他选项没有要求。

2）服务器端导出自签证书，命令示例如下。

```
keytool -export -alias server -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.keystore
-storepass restful -rfc -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restServer.cer
```

3）客户端创建自签证书，命令示例如下。

```
keytool -genkey -dname "CN=mars64, OU=Rest, O=JaxRs2, L=HaiDian, ST=Beijing,
C=China" -alias admin -keyalg RSA -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restAdminClient.
keystore -keypass restful -storepass restful -validity 60
keytool -genkey -dname "CN=mars64, OU=Rest, O=JaxRs2, L=HuangGu, ST=Shenyang,
C=China" -alias user -keyalg RSA -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restUserClient.
keystore -keypass restful -storepass restful -validity 60
```

由于我们要演示管理员和普通用户两种角色，因此，客户端生成了两个证书并分别保存在独立的 keystore 文件中，以便客户端程序分别加载。

4）客户端导入服务器证书，命令示例如下。

```
keytool -importcert -noprompt -trustcacerts -alias server -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restServer.cer -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restAdminClient.
keystore -storepass restful -keypass restful
keytool -importcert -noprompt -trustcacerts -alias server -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restServer.cer -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restUserClient.
keystore -storepass restful -keypass restful
```

将服务器导出的证书分别导入客户端的两个 keystore 中。

使用 -list 查看导入后 keystore 的信息，可以看到该 keystore 中存储的私钥和信任的公钥列表，以别名来区分，命令示例如下。

```
keytool -list -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restUserClient.
keystore -storepass restful
```

5）客户端导出自签证书，命令示例如下。

```
keytool -export -alias admin -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restAdminClient.
keystore -storepass restful -rfc -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restAdminClient.cer
keytool -export -alias user -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restUserClient.
keystore -storepass restful -rfc -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restUserClient.cer
```

6）服务器端导入客户端证书，命令示例如下。

```
keytool -importcert -noprompt -trustcacerts -alias admin -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restAdminClient.cer
-keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.
keystore -storepass restful -keypass restful
keytool -importcert -noprompt -trustcacerts -alias user -file
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\certificate\restUserClient.
cer -keystore
D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.
keystore -storepass restful -keypass restful
```

7）服务器端权限配置，命令示例如下。

```
<user username="CN=mars64, OU=Rest, O=JaxRs2, L=HaiDian, ST=Beijing, C=China"
password="null" roles="admin"/>
<user username="CN=mars64, OU=Rest, O=JaxRs2, L=HuangGu, ST=Shenyang, C=China"
password="null" roles="user"/>
```

证书管理完毕后，待服务器和客户端启动加载。接下来，需要对服务器端的权限进行配置。根据上述客户端的两个角色的唯一判别名，将它们的角色分别定义为 admin 和 user。

（4）配置服务器 SSL

Tomcat 服务器支持两种证书服务，一种是 APR（Apache Portable Runtime），另一种是 JSSE。本例使用 JSSE 方式。

```
<Connector clientAuth="true" port="8443" minSpareThreads="5"
maxSpareThreads="75" enableLookups="true" disableUploadTimeout="true"
acceptCount="100" maxThreads="200" scheme="https" secure="true"
SSLEnabled="true"
keystoreFile=
"D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.keystore"
keystoreType="JKS" keystorePass="restful"
```

```
truststoreFile=
"D:\-aries\github\jax-rs2-guide\sample\6\security-rest\keystore\restServer.keystore"
truststoreType="JKS" truststorePass="restful" SSLVerifyClient="require"
SSLEngine="on" sslProtocol="TLS" />
```

(5) 配置应用

本例的 web.xml 配置参考 6.3.3 节的示例，唯一不同之处是登录配置，如下所示。

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

(6) 认证和授权测试

浏览器动态加载证书提交请求并不常见，本示例的测试参见下文的客户端实现。

通过上述内容，相信读者对在 REST 中实现认证和授权有了深入的了解，并结合自己的项目有了初步的选择。第 11 章的综合示例会给出一种客户端“记忆”当前用户信息的方案，目的是解决 REST 服务分模块部署的场景问题。因为在该场景下，某个模块“记忆”的用户信息无法与其他模块分享。该方案并没有使用 Cookie，具体是如何实现的，参见第 11 章。

6.4 JAX-RS 2.0 实现

6.3 节通过几个示例展示了单独使用容器进行认证和授权的过程。本节将展示结合上述的容器认证，通过编码进行授权的流程。

1.Application 类

Application 类是容器中 REST 应用的入口，安全相关的配置要在这里完成注册，示例代码如下。

```
@ApplicationPath("/webapi/*")
public class AirResourceConfig extends ResourceConfig {
    public AirResourceConfig() {
        super(RolesAllowedDynamicFeature.class, BookResource.class);
    }
}
```

在这段代码中，Application 类 AirResourceConfig 注册了 RolesAllowedDynamicFeature 以支持 6.2 节讲述的 JSR 250 注解，还注册了资源类 BookResource 以实现 REST 服务。

2. 资源类

本例中，与其他章节相比，与授权相关的资源类 BookResource 的实现加入了 JSR 250 的注解和 JAX-RS 2.0 的 SecurityContext 接口，示例代码如下。

```
@Path("books")
```

```
public class BookResource {
    @RolesAllowed(value={"admin"})// 关注点 1: 根据角色授权访问
    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public Books getBooks(@Context final SecurityContext sc) {
        logMe(sc);// 关注点 2: 在方法中获取安全上下文
        final Books books = bookService.getBooks();
        BookResource.LOGGER.debug(books);
        return books;
    }
    private void logMe(final SecurityContext sc) {
        try {
            BookResource.LOGGER.info("User=" + sc.getUserPrincipal().getName());
            BookResource.LOGGER.info("User Role?=" + sc.isUserInRole("user"));
            BookResource.LOGGER.info("Auth way=" + sc.getAuthenticationScheme());
        } catch (final Exception e) {
            LOGGER.debug("Cannot print credential info."+e);
        }
    }
User=eric
User Role?=false
Auth way=BASIC
```

在这段代码中，GET 方法 getBooks() 使用了 @RolesAllowed（value={“admin”}）注解，授权访问此方法的角色为 admin，见关注点 1。此时，虽然容器的配置中定义了 user 角色可以访问这个 GET 方法，但是应用级别的权限优先级高于容器级别，在此定义的具有更细粒度的权限约束致使 user 角色不能访问 getBooks() 方法。getBooks() 方法定义了 SecurityContext 实例作为参数，SecurityContext 实例携带了用户角色的相关信息，运行时可以从中获取当前 getBooks() 方法访问者的角色信息，见关注点 2。

3. 资源测试类

Jersey 2.x 提供了基本认证连接过滤器（HttpBasicAuthFilter）、摘要认证连接过滤器（HttpDigestAuthFilter），而证书认证是通过设置 JerseyClient 内部的 SSLContext 字段实现的。Jersey 2.5 开始将 HttpBasicAuthFilter 和 HttpDigestAuthFilter 标记为 deprecated，使用 HttpAuthenticationFeature 统一配置认证。

（1）基本认证测试

对于基本认证的客户端测试，分别测试普通连接和注册 HttpBasicAuthFilter 或者 HttpAuthenticationFeature 的连接。没有提供基本认证的连接会得到服务器返回的 HTTP 状态码 401，表示没有通过认证的错误，对应的预计异常类是 NotAuthorizedException。在 ClientConfig 实例中注册 HttpBasicAuthFilter 实例的连接会通过认证并取得访问 GET 方法的权限，从服务器获得 Books 类型的数据。示例代码如下。

```
@Test(expected = javax.ws.rs.NotAuthorizedException.class)
public void testGetAll() {
  final ClientConfig cc = new ClientConfig();
```

```
    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
  }
  @Test
  public void testGetAll2() {
    final ClientConfig cc = new ClientConfig();
    //Jersey2.5-: cc.register(new HttpBasicAuthFilter("caroline", "zhang"));
    HttpAuthenticationFeature feature =
  HttpAuthenticationFeature.basicBuilder().nonPreemptive()
  .credentials("caroline", "zhang").build();
    cc.register(feature);
    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
  }
```

(2) 摘要认证测试

摘要认证的客户端测试和基本认证测试过程类似。

对于基本认证的客户端测试，分别测试普通连接和注册 HttpDigestAuthFilter 或者 HttpAuthenticationFeature 的连接。没有提供摘要认证的连接会得到服务器返回的 HTTP 状态码 401，表示没有通过认证的错误，对应的预计异常类是 NotAuthorizedException。在 ClientConfig 实例中注册 HttpDigestAuthFilter 实例的连接会通过认证并取得访问 GET 方法的权限，从服务器获得 Books 类型的数据。示例代码如下。

```
  @Test
  public void testGetAll2() {
    final ClientConfig cc = new ClientConfig();
    //2.5-: cc.register(new HttpDigestAuthFilter("caroline", "zhang"));
    HttpAuthenticationFeature feature =
  HttpAuthenticationFeature.digest("caroline", "zhang");
    cc.register(feature);
    final Client client = ClientBuilder.newClient(cc);
    final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
    invocationBuilder.get(Books.class);
  }
```

(3) 证书认证测试

对于证书认证的客户端测试，需要用到 6.3.5 节关于证书认证的知识。首先设置 keystore 到 SslConfigurator 类的实例中，SslConfigurator 实例通过本地证书信息生成 javax.net.ssl.SSLContext 类的实例，接着 SSLContext 实例赋值于 REST 客户端，作为 SSL 上下文字段，此时 REST 客户端向服务器发起握手流程，即可实现上文所述的双向认证。授权逻辑可参考本章前言内容。示例代码如下。

```
  private Client buildSecureClient(boolean admin) {
    String keystore;

    if (admin) {
```

```
    keystore = "restAdminClient.keystore";
} else {
    keystore = "restUserClient.keystore";
}
final SslConfigurator sslConfig = SslConfigurator.newInstance().trustStoreFile(keystore)
.trustStorePassword("restful").keyStoreFile(keystore).keyPassword("restful");
final SSLContext sslContext = sslConfig.createSSLContext();
final Client client = ClientBuilder.newBuilder().sslContext(sslContext).build();
return client;
}
@Test
public void testGetAll() {
  final Client client = buildSecureClient(false);
  final Invocation.Builder invocationBuilder = client.target(BASE_URI).request();
  invocationBuilder.get(Books.class);
}
@Test(expected = javax.ws.rs.ForbiddenException.class)
public void testPost() {
...
  buildSecureClient(false).target(BASE_URI)
.request(MediaType.APPLICATION_JSON_TYPE).post(bookEntity, Book.class);
}
```

6.5 其他安全考虑

在完成了通用的认证和授权的实现后，我们的 REST 服务还要考虑很多安全方面的问题。完整的安全策略推荐阅读专业领域的书籍，在此简单介绍一下 AJAX 的跨域安全和 OAuth。

（1）AJAX 跨域安全

和其他 Java EE 标准一样，JAX-RS 2.0 并没有给出对 XSS（Cross Site Script，跨站脚本攻击）和 CSRF(Cross-Site Request Forgery，跨站点请求伪造）等攻击手段的防范标准。因此，一方面需要开发者在编码中防止出现代码级别的漏洞，另一方面需要开发者实现规避攻击风险的代码。举例来说，可以借助 JAX-RS 2.0 定义 Providers 接口来实现过滤器，Jersey 2.x 提供了 CSRF 防御的过滤器实现类 CsrfProtectionFilter 可供参考。对于第 4 章示例代码包中提供的关于 CORS（Cross-Origin Resource Sharing，跨域资源共享）的实例是基于“HTTP 头不会被脚本篡改”这一理论的，在联合部署不同应用的 REST 服务的场景中，CORS 是必要的解决方案。

（2）OAuth 和 OAuth2

虽然 OAuth 和 OAuth2 是当今 Web 服务（尤其是社交网络）的流行的认证协议，但是 JAX-RS 2.0 并没有给出相关定义。Jersey 2.x 对这两种协议的实现处于开发状态，在本书发稿时，Jersey 2.x 提供了 OAuth 和 OAuth2 规范的客户端实现和 OAuth 协议的服务器端支持包，尚未提供 OAuth2 的服务器端支持包。读者可以参考 Jersey 源代码中 security 目录的进展情况。Jersey 2.x 的 OAuth2 客户端支持包 oauth2-client 的依赖定义示例如下。

```
<dependency>
  <groupId>org.glassfish.jersey.security</groupId>
  <artifactId>oauth2-client</artifactId>
  <version>${project.version}</version>
</dependency>
```

6.6 本章小结

本章全面讲述了与 REST 安全相关的理论和实践。其中 6.1 节介绍了 HTTP 基本认证、HTTP 摘要认证、表单认证和证书认证；6.2 节介绍了容器和应用两种权限管理方式；6.3 节结合 Tomcat 的 Realm 机制，分别演示了 4 种认证的实践；6.4 节演示了使用 JAX-RS 2.0 定义的安全注解，通过编码实现权限管理；6.5 节简述了其他安全技术。下一章，我们开始讲述 REST 的测试。

第 7 章 Chapter 7

REST 测试

本章讲述如何使用 Jersey 提供的测试框架实现对基于 JAX-RS 2.0 的 Web 服务进行自动化测试。

自动化测试是软件质量保证的必要手段，是减少开发团队重复劳动的利器，同时也是敏捷开发的重要环节。项目源代码的代码覆盖率即指测试代码中对源代码的公有方法的覆盖情况，是 CI（Continuous Integration，持续集成）和 CD（Continuous Delivery，持续交付）中的重要指标之一。

使用 JUnit 结合 Jersey 的测试框架，能够轻松地实现单元测试、CI 测试和系统测试。Jersey 提供了 4 种内置容器，测试过程对外置容器没有天然的依赖。

阅读指导 第 7 章没有单独的示例项目，因为测试存在于各个项目中，可参考 2.5 节的示例项目：jax-rs2-guide\sample\2\5simple-service-webapp-spring-jpa-jquery。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/2/5simple-service-webapp-spring-jpa-jquery。

7.1 Jersey 测试框架

Jersey 核心测试框架包 jersey-test-framework-core 提供了测试环境中容器和客户端实例，使开发者不必编写管理容器和客户端的样板代码。

在没有 Jersey 核心测试框架支持的测试类中，开发者需要在编写测试用例的同时，维护测试容器和客户端代码。setUp() 方法中的代码都是样板代码，这些样板代码会在使用 Jersey

核心测试框架的测试类中消失。没有使用 Jersey 测试框架的测试，示例代码如下。

```
public class TIMyResourceTest {
    public static final String BASE_URI = "http://localhost:8080/webapi/";
    private HttpServer server;
    private WebTarget target;
    @Before
    public void setUp() throws Exception {
        final ResourceConfigrc = new ResourceConfig().packages("com.example");
        final URI uri = URI.create(BASE_URI);
        server = GrizzlyHttpServerFactory.createHttpServer(uri, rc);
        server.start();
        final ClientConfig cc = new ClientConfig();
        final Client client = ClientBuilder.newClient(cc);
        target = client.target(BASE_URI).path("books");
    }
    @After
    public void tearDown() throws Exception {
        if (server != null) {
            server.stop();
        }
    }
    @Test
    public void testQueryGetXML() {
        final WebTarget queryTarget = target.path("/book")
        .queryParam("id", Integer.valueOf(1));
        final Invocation.Builder invocationBuilder =
        queryTarget.request(MediaType.APPLICATION_XML_TYPE);
        final Response response = invocationBuilder.get();
    ...
    }
```

使用了 Jersey 核心测试框架后，代码量减少了很多，示例代码如下。

```
public class TIMyResourceJTFTest extends JerseyTest {
    private static final String BASEURI = "books/";
    @Override
    protected Application configure() {
        return new ResourceConfig(BookResource.class);
    }
    @Test
    public void testQueryGetXML() {
        final WebTarget queryTarget = target(BASEURI + "book")
        .queryParam("id", Integer.valueOf(1));
        final Invocation.Builder invocationBuilder =
        queryTarget.request(MediaType.APPLICATION_XML_TYPE);
        final Response response = invocationBuilder.get();
    ...
    }
```

在这段代码中，TIMyResourceJTFTest 类继承了 Jersey 核心测试框架类 JerseyTest，其测试代码中只有覆盖 configure() 方法是与测试用例无关的代码。在测试用例中可以直接使用

target() 方法从测试框架中获取 WebTarget 类的实例，余下的代码都是测试用例本身的内容。

在项目中使用 Jersey 核心测试框架包时，需要在 pom.xml 文件中定义依赖。Jersey 核心测试框架提供了 4 种内置容器，默认使用 Grizzly 2 容器。因此，定义 Jersey 核心测试框架依赖的同时，需要定义测试框架 Grizzly 2 容器包 jersey-test-framework-provider-grizzly2。否则，运行时会报 TestContainerException 异常。

Jersey 测试框架核心的依赖定义如下：

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework</groupId>
  <artifactId>jersey-test-framework-core</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

下面是 4 种内置容器的定义。

1）Jersey 内置容器 Grizzly 2 依赖定义如下：

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架 Grizzly 2 容器所在的模块为 jersey-test-framework-provider-grizzly2。Grizzly 轻量级 HTTP 容器是 Jersey 测试框架的默认容器。

2）Jersey 内置容器 inmemory 依赖定义如下：

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-inmemory</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架内存容器所在的模块为 jersey-test-framework-provider-inmemory，内存容器不是一个真正的容器。测试框架直接调用应用的 API，因此没有真正的网络通信。内存容器不支持真正的容器特性和 Servlet，但是非常适合作为单元测试阶段的容器。

3）Jersey 内置容器 JDK-HTTP 依赖定义如下：

```
<dependency>
  <groupId>org.glassfish.jersey.test-framework.providers</groupId>
  <artifactId>jersey-test-framework-provider-jdk-http</artifactId>
  <version>${jersey.version}</version>
</dependency>
```

测试框架 JDK 版 HTTP 容器所在的模块为 jersey-test-framework-provider-jdk-http，JDK 版 HTTP 容器是 JDK。

4）Jersey 内置容器 simple-HTTP 依赖定义如下：

```
<dependency>
```

```
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-simple</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

测试框架简单版 HTTP 容器所在的模块为 jersey-test-framework-provider-simple，集成 Jersey 的轻量级 HTTP 容器。

在测试类中使用 4 种内置容器非常方便，只需要在覆盖 configure() 方法中定义容器工厂名称即可。TestProperties.CONTAINER_FACTORY（jersey.config.test.container.factory）参数的值设置为 org.glassfish.jersey.test.simple.SimpleTestContainerFactory。

```
public class TIMyResourceJTFTest extends JerseyTest {
    static final String CONTAINER_GRIZZLY =
    "org.glassfish.jersey.test.grizzly.GrizzlyTestContainerFactory";
    static final String CONTAINER_MEMORY =
    "org.glassfish.jersey.test.inmemory.InMemoryTestContainerFactory";
    static final String CONTAINER_JDK =
    "org.glassfish.jersey.test.jdkhttp.JdkHttpServerTestContainerFactory";
    static final String CONTAINER_SIMPLE =
    "org.glassfish.jersey.test.simple.SimpleTestContainerFactory";
    @Override
    protected Application configure() {
        set(TestProperties.CONTAINER_FACTORY, CONTAINER_SIMPLE);
        return new ResourceConfig(BookResource.class);
    }
    ...
}
```

7.2 单元测试

简单来说，单元测试就是对源代码中公有方法的功能性测试。本节不会对单元测试的基本概念和测试方法进行详述，而是着重讲述单元测试的难点。

7.2.1 集成 Spring 的单元测试

第一个难点是如何在单元测试中使用 Spring 容器。Spring 提供了相关的解决方案，可以在测试类的注解中指定 SpringJUnit4ClassRunner 来实现，下面开始详细描述。

1. 依赖注入

对于使用依赖注入的源代码，单元测试最常见的问题是被测试类 A 依赖注入了 B 类，运行时 B 的实例总是空指针。出现这个问题是因为单元测试的上下文中没有依赖注入容器的环境。以 Spring 作为依赖注入容器的项目为例，BookService 类中依赖注入了 BookDao 类的实例，那么测试代码中，BookService 类的实例就不能直接通过构造函数创建，否则这个 BookService 实例就不是 Spring 容器管理的 Bean，导致的结果就是 BookDao 实例为空指针。示例代码如下。

```
public class BookService {
  @Autowired
  privateBookDaobookDao;
public Book saveBook(final Book book) {
return bookDao.store(book);
}
...
}
```

这种情况下，我们可以在测试类中使用 @RunWith 注解定义 SpringJUnit4ClassRunner 类。与 Jersey 测试框架帮助用户管理测试服务器和客户端类似，spring-test 包提供的 SpringJUnit4ClassRunner 类可以帮用户管理 Spring 容器。通过 @ContextConfiguration 注解定义的 Spring 配置文件，SpringJUnit4ClassRunner 实例在测试类启动后，帮助用户建立起了 Spring 容器环境，这样就可以让容器帮助用户生成 BookService 类的实例，这一过程中，BookDao 类的实例也创建好并存在于 Spring 容器中了。单元测试代码使用 @Autowired 定义了一个 BookService 字段，其余部分只和测试用例有关。示例代码如下。

```
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class TUMyServiceTest {
    @Autowired
    private BookService bookService;
    @Test
    public void testGetAndSave() {
        final Book result = bookService.getBook(1L);
...
}
```

2. 事务

在使用 JPA 的项目时，对 DAO 层源代码的测试中，一个常见的“怪现象”是明明存储一个实体类已经通过，数据库中却没有这条记录。这是因为 JPA 需要显式地提交事务才会将数据真正写入数据库。在集成 Spring 的项目中，通常使用 @Transactional 注解存储方法，即可在运行时由 Spring 容器提交事务，如下所示。

```
@Repository
public class BookDao {
    @Transactional
    public Book store(final Book entity) {
        return entityManager.merge(entity);
    }
...
}
```

但是，项目中的业务逻辑不总会允许每一个实体的存储都原子地提交事务，而是当 A、B、C 都成功保存后才会提交事务，否则回滚事务。因此，@Transactional 注解会从 DAO 层移至 Service 层。这种情况下，对 DAO 层的单元测试再次陷入到“怪现象”的窘境中。有两种情况可以忽略测试数据的持久化，一种是使用内存数据库（通常，单元测试应该这样做），

另一种是测试逻辑成功即可，测试数据没有意义。

如果存储测试数据是我们的单元测试所必需的，那么只有在测试类中寻找实现事务显式提交的办法。这种情况下，我们可以使用 @TransactionConfiguration 注解和 @Transactional 注解，如下所示。

```
@ContextConfiguration(locations = { "classpath:context.xml",
"classpath:context-persistence.xml", "classpath:context-transaction.xml" })
@TransactionConfiguration(transactionManager="xxxTransactionManager",
                          defaultRollback=false)
@Transactional
@RunWith(SpringJUnit4ClassRunner.class)
public class TITestXxxDao {
...
}
```

7.2.2 异步测试

单元测试中第二个难点是对实现异步功能的源代码进行测试。一个常见的现象是被测试的线程还在进行中，而测试主线程已经结束。JUnit 框架同步测试的扩展可以从编码和架构上解决。

从编码上解决异步使用第 8 章将要讲述的 Java 并发技术。例如，在主线程中使用 Future 和同步器来阻塞主线程，直到被测试的线程结束（代码的实现可参考第 8 章相关示例）。也可以在测试用例中创建线程或使用线程池来测试异步程序。异步测试代码的编写相对难度大些，但可以快速解决对功能模块的单元测试，需要读者仔细揣摩。这里要说明的是，异步单元测试用例通常比较耗时，因此在全量执行单元测试时应当排除这部分用例。一个通用的做法是使用 Maven 定制 profile 来隔离不同的测试用例。

架构上可以选择 JUnit 的第三方扩展包来支持多线程测试，这超出了本书的范围，读者可参考 JUnit 扩展的相关资料。从实际开发和单元测试经验上看，架构层面的解决方案更重量级，适合团队规定使用的场景；编码解决更轻量和灵活，但不具有通用性。

阅读指导 异步测试的示例可参考第 8 章的示例项目：jax-rs2-guide\sample\8\asyn-rest。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/8/asyn-rest。

7.3 集成测试

集成测试是系统运行时的功能测试，是对组件、模块之间联合工作的测试。REST 应用的集成测试就是对系统公开的资源地址的功能性测试。运行时测试依赖于测试服务器。根据笔者经验，测试服务器的建立与启动可以分为 3 类。

第一类是使用如上所述的内置 HTTP 容器。这种方式通过 JUnit 测试类直接定义测试服务器或者继承 Jersey 测试框架类 JerseyTest，使其在执行测试前加载配置并启动服务，在测

试用例完成后停止服务。

第二类是使用 Maven 插件启动外置服务器。这种方式通过在项目的 Maven 配置文件 pom.xml 中定义测试服务器，使其在 Maven 生命周期的 pre-integration-test 阶段启动，并在 post-integration-test 阶段停止，可参考第 2 章关于 Servlet 容器应用一节。

以上两种方式都无须部署（deploy）到测试服务器。第三类是将项目打包部署到和生产环境相同的集成测试环境，其优点是可以模拟真实场景，以完成对系统功能性和非功能性的测试，而缺点是打包部署占用了集成测试的时间。典型的 Servlet 容器是 Tomcat，Java EE 容器是 GlassFish，可参考第 2 章关于 Servlet 容器应用一节。

7.4 日志增强

在测试日志中，Jersey 测试框架提供了输出通信过程中的传输日志信息。在测试代码中启用这一功能是通过覆盖 JerseyTest 类的 configure() 方法并加入关注点 1 这一行，示例代码如下。

```
public class TIMyResourceJTFTest extends JerseyTest {
@Override
protected Application configure() {
  enable(TestProperties.LOG_TRAFFIC);// 关注点 1: 启用传输日志
  enable(TestProperties.DUMP_ENTITY);
  //set(TestProperties.CONTAINER_FACTORY, CONTAINER_SIMPLE);
  return new ResourceConfig(BookResource.class);
}
...
}
```

这样一来，测试过程中输出的信息将包括传输日志，可参考下面的输出。在启用传输日志后，可以启用实体日志来输出传输过程中的实体信息，可参见下面代码中最后一段所示的 XML 格式的数据。

```
1 > GET http://localhost:9998/books/
2 < 200
2 < Date: Tue, 22 Oct 2013 03:24:46 GMT
2 < Content-Length: 496
2 < Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><books><bookList><book bookId="1"
bookName="Java Restful Web Service 使用指南 " publisher="cmpbook"/><book bookId="2"
bookName="JSF2 和 RichFaces4 使用指南 " publisher="phei"/></bookList></books>
```

7.5 本章小结

本章详细讲述了 Jersey 的测试框架及其在单元测试和集成测试中的使用。单元测试主要考虑依赖注入和事务这两个技术点如何和生产环境保持一致。集成测试主要考虑如何部署测试环境，使之尽量贴近生产环境。最后介绍了 Jersey 测试框架对日志信息的增强。下一章，我们开始讲述 REST 的服务器端推送技术和异步通信。

Chapter 8 第 8 章

REST 推送与异步通信

本章分为两部分，前半部分讲述 REST 通信中有关服务器推送事件的技术，后半部分讲述 REST 的异步通信和 JAX-RS 2.0 定义的异步请求处理规范。

8.1 服务器-浏览器通信

服务器-浏览器通信也可称为服务器端推送技术，是一种当服务器端的业务数据、资源状态发生改变时，服务器可以主动将这一信息通知给相关的浏览器的通信技术。如果服务器与客户端使用 TCP/IP 协议建立连接，这样的 Socket 通信并无特别之处，一旦连接建立，在这样的双向通信链路中随时都可以发送通知。然而，REST 通信是基于 HTTP 的通信，而 HTTP 是无状态的通信协议，每一次请求-响应都是基于一个新建立的 HTTP 连接，这就使服务器主动通知浏览器成为一个难点。这是因为在基于请求-响应模式下，服务器的角色是被动应答，无法主动通知浏览器；另一个原因是，每次请求后 HTTP 连接断开，服务器无法再获取客户端的地址，也就是无法将通知发送给浏览器端。

为了解决这一问题，人们尝试过很多基于请求-响应模式的变通办法，总结起来有两种途径。第一种是由浏览器周期性地主动去服务器获取，使用“拉”的方式模拟服务器的“推”；要么在请求处理结束后，服务器返回响应但不关闭这条连接，直到服务器将业务数据的变化响应给浏览器，最后将连接关闭。第二种方式并没有破坏请求-响应模式，而是将响应分成了两个阶段。这两种方式都存在着让开发者头疼的问题，处理不好会给产品日后的维护带来问题。

先回到 REST 中，这些 HTTP“通病”会在后面结合着 REST 实践给出解决方案。和 MVC 一样，服务器-浏览器通信在 JAX-RS 2.0 中并没有相关规范的定义，但是和 MVC 与 REST 平行的关系不同的是，SSE（Server-Sent Events，服务器端推送事件）这项技术对实现

REST 式的 Web 服务的功能性和性能颇具影响。因此，本书将其收录并作为一个小节讲述，本章将基于上述提出的两种解决方案，详述如何在项目中实现这一通信技术。

8.1.1 Polling 技术

服务器-浏览器通信技术的第一种解决方案是客户端轮询技术，即 Polling。

1. 简述

客户端轮询技术（Polling）相对其他方案，最原始、易行，即浏览器周期性地主动访问服务器的特定地址，以获取服务器端数据状态的变化。如图 8-1 所示，通常来说，在浏览器端使用 JavaScript 脚本启动一个定时任务，该任务向服务器发送请求并获取资源状态。如果服务器端特定数据发生变化，那么会将变化信息响应给客户端，客户端使用响应的数据渲染界面，为用户做出及时的反馈。

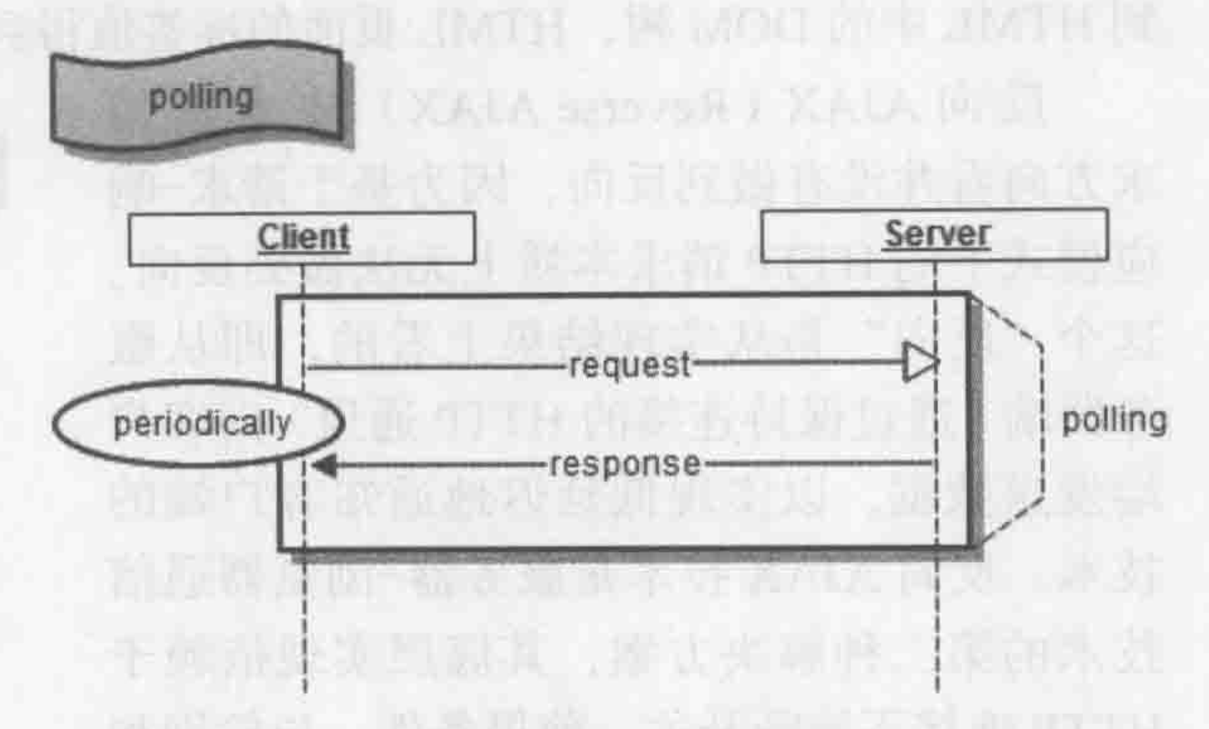


图 8-1 客户端轮询示意图

2. 优点与缺点

优点：客户端轮询技术易于实现。不需要为此在服务器端或者浏览器端额外使用任何第三方的库，开发者容易理解，使用现有技术和工具就可以实现。客户端轮询技术对设计没有注入性污染。选择技术架构和设计、实现业务逻辑时，这种方式可以即插即拔，不会污染业务平台中结构性的代码。

缺点：客户端轮询技术的缺点是显而易见的，轮询中的每次请求-响应过程都需要建立新的 HTTP 连接并在结束时关闭该连接。这就造成两大问题。

第一，如果服务器端的业务数据在两次定时任务发起的请求过程中没有变化，后一次请求的做功实际为负数——浪费了服务器端的带宽，而且没有获得有效负载。

第二，也是最让开发者纠结的事情，即浏览器端的定时器间隔时间参数的设置。由于需要及时获取服务器端的业务数据的状态，这个定时间隔参数设置不宜过长，但是过短又会频发第一个问题。因此，间隔时间的设置是个比较尴尬的问题，因为在编码和调试阶段定义并运行完好的参数，很难和生产环境吻合，甚至开发阶段有可能疏漏或无法覆盖全部生产环境中的业务场景。还有一个让人难受的地方是，这种请求的代码很难抽象出来，因为不同业务的定时间隔都是一个独立的经验值。

8.1.2 Comet 技术

Comet 是反向 AJAX 的技术集，包括长轮询（Long Polling）和流（Streaming）两种技

术实现。

1. 简述

什么是反向 AJAX 呢？要了解反向 AJAX，要先从 AJAX 说起。

AJAX 技术是指从浏览器端向服务器端发起的异步请求，已经广为人知，8.1.1 节所述的 Polling 就是 AJAX 的实践。概括地说，浏览器发起的请求是通过脚本实现的，页面并没有提交或者跳转，请求由服务器处理并返回响应数据后，浏览器处理响应数据并将这一变化渲染到 HTML 中的 DOM 树，HTML 页面的标签值得到了更新，实现了页面的局部刷新。

反向 AJAX（Reverse AJAX）技术从请求方向看并没有做到反向，因为基于请求-响应模式下的 HTTP 请求本质上无法做到反向。这个“反向”是从实现结果上看的，即从服务器端（通过保持连接的 HTTP 通道）向客户端发送数据，以实现低延迟地通知客户端的技术。反向 AJAX 技术是服务器-浏览器通信技术的第二种解决方案，其底层实现依赖于 HTTP 连接不能断开这一前提条件。长轮询和流技术是反向 AJAX 的两种技术手段，通信原理相同。长轮询示意图如图 8-2 所示。

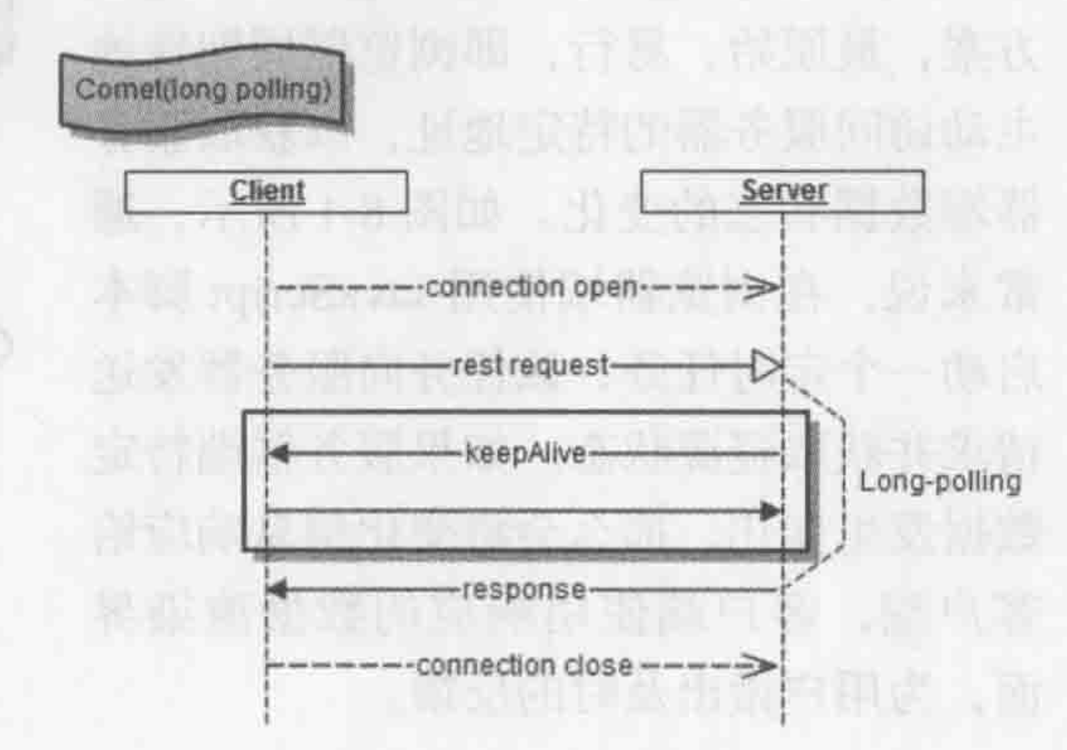


图 8-2 长轮询示意图

如图 8-2 所示，长轮询通过 keepAlive 使 HTTP 连接得以保持。为什么要保持连接呢？因为在请求发出后的一定时间内，服务器一直没有做出响应，该连接会因连接超时而断开。Comet 利用 HTTP 1.1 的 keepAlive 持久性连接技术，在浏览器发出请求后，通过 keepAlive 保持服务器向浏览器做出响应的通信。这样一来，就解决了连接超时断开的问题。那么，连接的关闭就只有两种情况，一种是浏览器主动断开，另一种是服务器端特定数据发生变化，并将这一信息响应给浏览器，主动断开连接来完成请求-响应模式的一次请求。

相比 Polling，实现 Comet 要困难得多，服务器端和浏览器端都需要第三方的库来支持这一技术。Atmosphere 库和 CometD 库是实现 Comet 技术的第三方工具包，Jersey 并没有提供支持 Comet 实现的包，本书也不会对 Comet 的实现做进一步讨论，但读者要清楚 Comet 技术是这个领域的一个选择。

2. 优点与缺点

优点：反向 AJAX 的技术解决了 Polling 低效地消耗服务器的网络带宽和系统负载的问题。同时，由于服务器主动向浏览器发送数据，因此有很好的低延迟性。

缺点：Comet 需要服务器端额外的技术实现来支持，同时需要在服务器和浏览器两端引入第三方工具包，实现相对复杂。

8.1.3 SSE 技术

SSE 是 HTML5 技术集的一部分，定义了服务器推送技术的标准规范。

1. 简述

SSE 规范的地址是 http://dev.w3.org/html5/eventsource。其核心是基于 EventSource 接口的事件监听机制，包括 onopen、onmessage 和 onerror 三个事件监听器。SSE 服务器端响应数据的媒体类型（Content-Type）是 text/event-stream。Jersey 的媒体库提供了对 SSE 的支持，可参见 8.2 节相关内容。

2. 优点与缺点

优点：SSE 是标准规范——HTML5 标准之一，具备编程语言的无关性。首先，SSE 支持跨语言的开发，无论具体使用什么语言和框架，只要按照以 EventSource 接口为中心，完成事件监听机制即可实现 SSE。其次，SSE 支持跨语言的调用，这点很好理解，正是基于标准接口和标准事件监听，七层协议上的 HTTP 包很容易被各系统彼此阅读。SSE 的代码实现和交互逻辑相对简单，在 Java EE 生态环境中，得到了 Jersey 提供的支持。

缺点：由于请求-响应模型的限制，SSE 和 Comet 一样，是一种从服务器端到浏览器端的单向通信，浏览器无法在同一条连接上做出二次请求或者对服务器的响应做出“响应”，这一缺点无法支持复杂的交互需求。另外，SSE 标准和 Jersey-sse 支持包都还在不断完善中（Jersey 提供持续升级的 SSE 支持包，但这并不意味着当前版本不稳定）。

8.1.4 WebSocket 技术

WebSocket 是 HTML5 技术集的一部分，它提供了一个双向的、在一条 TCP 信道中的客户端和服务器之间全双工的通信。

1. 简述

WebSocket 消除了所有与 HTTP 连接的无状态特性相关的限制。Java EE 7 已经支持 WebSocket，其参考实现是 GlassFish 项目的 Tyrus 子项目（项目地址是 https://tyrus.java.net）。本书不会对 WebSocket 的实现做进一步讲述，推荐读者阅读来自 Oracle 的布道者 Arun Gupta 所著的《Java EE 7 Essentials》一书的第 7 章内容。

2. 优点与缺点

优点：WebSocket 是标准规范——HTML5 标准之一，逐渐开始流行。其功能强大、性能突出：双向、双工通信。

缺点：相对于 SSE 的实现，WebSocket 较为复杂。

8.2 SSE 详述

从 8.1 节对上述 4 种服务器端推送技术的分析中，我们清楚了服务器-浏览器通信是什么和怎么做。我们的中心思想是如何提高基于 Jersey 的应用的性能，因此，本节结合 REST 式的 Web 服务，就服务器端推送事件（SSE）技术进行详细讲述。

阅读指导 8.2 节示例所在目录是 jax-rs2-guide\sample\8\sse。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/8/sse。

8.2.1 Java 并发

在 REST 的异步调用和测试中，会经常使用 JDK 1.6+ 提供的 Future 和 Synchronization（同步器）来保证异步、并发线程能够按照既定的顺序执行。因此，掌握 JDK 的异步执行接口 Future 和同步器的使用是十分必要的。本节，将对 JDK 1.6+ 提供的 Future 和 4 种同步器逐一讲述。

1. Future

Future 接口是 JDK 1.5 提供的用于处理异步线程的接口。该接口定义了一个泛型类型，该类型是 get() 方法的返回类型。get() 方法用于执行一个实现了 Callable 接口的线程，并获取其执行结果，作为 get() 的返回值。

（1）Callable

Callable 接口是 JDK 1.5 提供的用于执行任务的线程接口，是 Runnable 接口的增强（两者没有直接的依赖关系）。Runnable 接口定义的 run() 用于执行线程任务，但没有返回值；Callable 接口定义了一个泛型类型，该类型是接口方法 call() 的返回类型。call() 方法执行线程任务后，返回泛型类型的对象。

（2）FutureTask

FutureTask 类的字面意思是“未来的任务”，是一个结合了 Runnable 接口和 Future 接口的实现类，如图 8-3 所示。

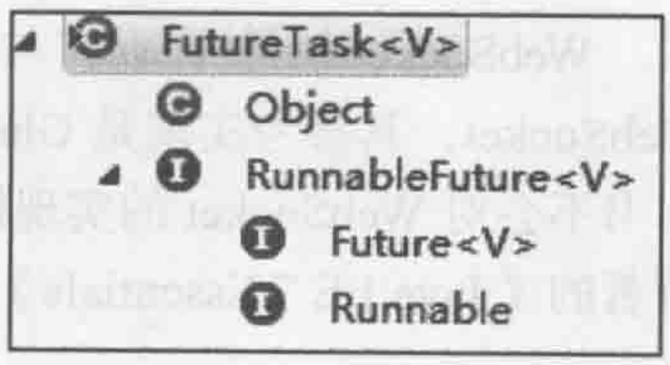


图 8-3 FutureTask 继承关系示意图

FutureTask 用于处理可取消的异步任务。FutureTask 的 get() 方法用于获取任务执行结果，如果任务尚未完成，则阻塞 get() 方法直至任务完成。FutureTask 的 run() 方法用于执行任务，任务一旦完成就不能重新开始或取消。它有三个状态：等待（waiting to run）、运行（running）和完成（completed）。

2. 同步器

同步器用于在多线程之间同步状态，JDK 根据不同的使用场景，提供了如下 4 种同步器。

（1）Semaphore 同步器

Semaphore 的字面意思是“信号”，通过内部计数器控制一组线程在同一时间对某一特定资源的访问，或者控制一组线程对执行某个操作的权限。Semaphore 的 acquire() 方法会使计数器数值递减，如果计数器归零，则请求线程被阻塞，直到计数器重新可以递减到零。Semaphore 的 release() 方法会使计数器数值递增，从而可能释放一个正在阻塞的线程。Semaphore 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。

（2）CountDownLatch 同步器

CountDownLatch 从字面上可以理解为一个带有 Count（计数器）的 Latch（门闩），在初始化后执行 countDown() 操作来控制多个线程的同步。CountDownLatch 通过内部的计数器控制所有线程处于等待，直到其计数器归零后，所有线程一起执行后续任务。在所有线程都能通过之前，每个线程通过 CountDownLatch 的 await() 方法（可以设置等待超时时间）等待。CountDownLatch 的计数器无法被重置。如果需要重置计数，那么可以考虑使用 CyclicBarrier。

CountDownLatch 的应用场景举例如下：

1）在一个资源初始化之前，使用该资源的所有活动都处于等待。

2）在释放一个服务之前，完成所有依赖于该服务的任务。

3）在开始多人游戏之前，确保所有参与者的终端都连接完毕。

（3）CyclicBarrier 同步器

CyclicBarrier 的字面意思是循环使用的 Barrier（路障）。CyclicBarrier 通过内部的计数器控制所有线程互相等待，直到到达某个公共路障点（Common Barrier Point）。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 Barrier 在释放等待线程后可以重用，所以称它为循环使用的 Barrier。

CountDownLatch 用来等待事件，当计数器递减归零时，被 CountDownLatch 的 await() 方法阻塞的线程继续执行后续任务；CyclicBarrier 用来等待其他线程，当 await() 方法使计数器递增的数量到达设定数量后，被 CyclicBarrier 的 await() 方法阻塞的线程继续执行后续任务。

（4）Phaser 同步器

Phaser 是 JDK 1.7 提供的同步器，其作用和 CountDownLatch、CyclicBarrier 类似，但 Phaser 的使用方式更为灵活。Phaser 使用 register（注册）方法来递增计数器，使用 arriveAndDeregister() 方法来递减计数器，使用 arriveAndAwaitAdvance() 方法来阻塞线程，该阻塞在 Phaser 的计数器归零时唤醒线程继续执行后续任务。

另外，JDK 提供 Exchanger（交换器）类用于两个线程共享资源并允许它们交换执行任务。Exchanger 在 REST 的场景中并不常用。

3. 并发数据结构

JDK 提供了包括数组、链表、队列和键值对等丰富的线程安全的并发数据结构。在每个数据结构内部，通过细粒度的锁或原子操作，确保内部的数据在多线程同时引用并操作时，

逻辑上的正确性。由于本书的内容和篇幅所限，无法列举 JDK 提供的全部并发数据结构，在此只简述本书示例中涉及的队列接口 BlockingQueue 的并发特性。BlockingQueue 接口操作见表 8-1。

表 8-1 lockingQueue 接口操作

操作	抛出异常	特殊值	阻塞	超时
入队	add(e)	offer(e)	put(e)	offer(e, time, unit)
出队	remove()	poll()	take()	poll(time, unit)
检测	element()	peek()		

如表 8-1 所示，“抛出异常”列，在并发场景中，当前线程的操作无法执行时直接抛出异常。“特殊值”列，当操作无法执行时返回 null 或者 false。“阻塞”列，当操作无法执行时会阻塞调用线程，直至操作执行完毕。“超时”列，当操作无法执行时，会在超时时间内阻塞调用线程，超时后放弃操作。

8.2.2 SSE 流程

Jersey 的 SSE 支持包 jersey-media-sse 基于 HTML5 的 SSE 规范，提供了一套支持 SSE 规范的完整的 API。Maven 依赖包定义如下。

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-sse</artifactId>
</dependency>
```

Jersey 的 SSE 支持包提供两种通信模式，即发布-订阅模式和广播模式。前者是一种端到端的通信，后者是多播通信。接下来我们分别讲述这两种模式。

1. 发布-订阅模式

发布-订阅模式的 API 和典型的工作流程描述如图 8-4 所示。

根据 SSE 标准规范定义的 EventSource 接口，Jersey SSE 定义了 EventListener 接口及其实现类 EventSource。SSE 的实现流程描述如下：

第一步，在客户端创建 EventSource 实例并覆盖 onEvent() 方法。onEvent() 方法用来处理服务器端推送事件，输入参数为 InboundEvent（进站事件）。EventSource 构造函数的输入参数为一个 WebTarget 端点，指向服务器端的资源路径为 sse 的 GET 方法，该方法使用了注解 @Produces（SseFeature.SERVER_SENT_EVENTS）。在 EventSource 实例化过程中，如图 8-4 中①所示，会从这个端点向服务器发出请求并指明接收数据的媒体类型为 SseFeature.SERVER_SENT_EVENTS，即 Accept 头信息声明为 text/event-stream 类型。

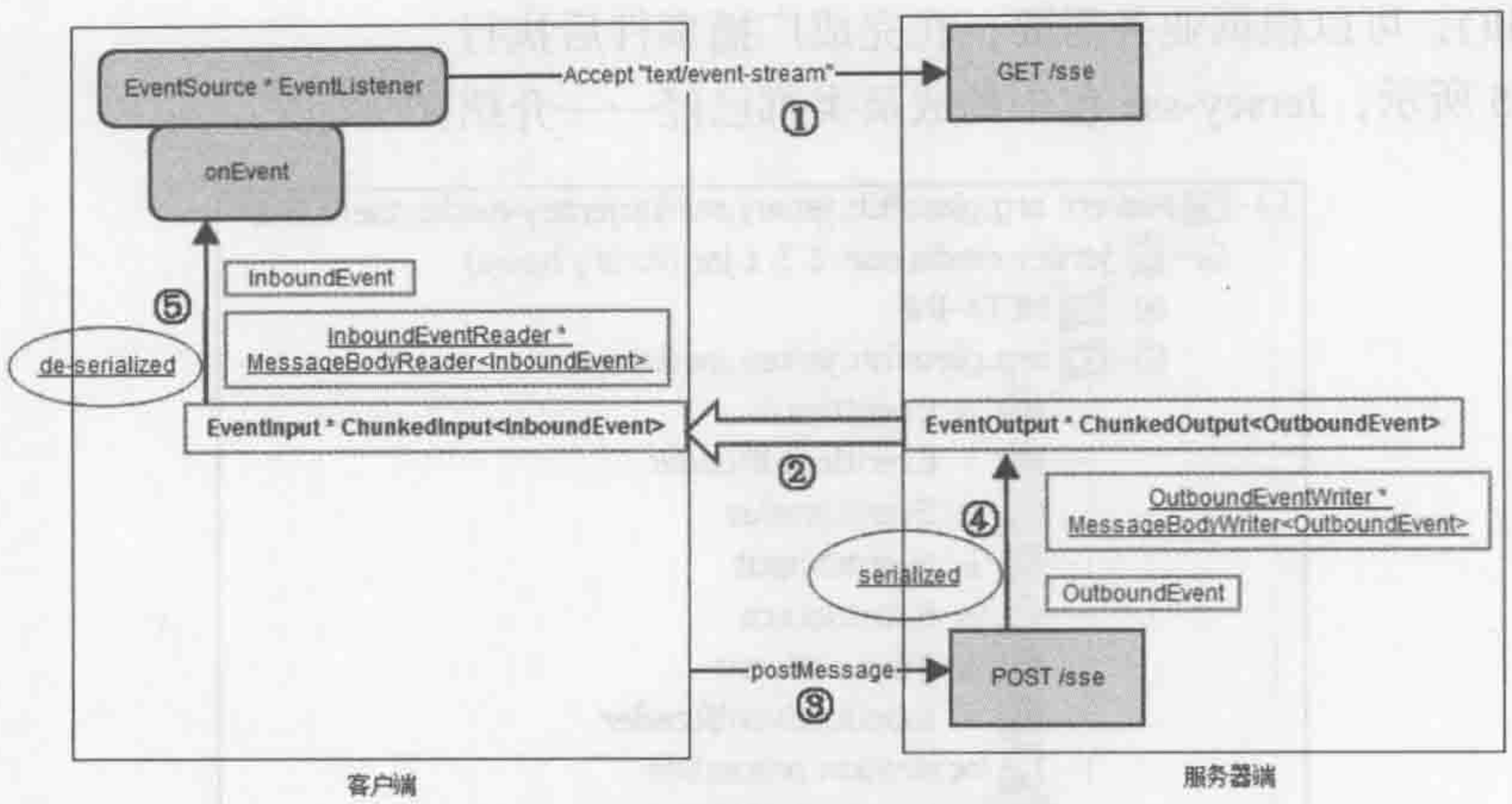


图 8-4　Jersey 实现 SSE 流程示意图

第二步，请求被服务器接收后，开启 SSE 事件通信通道，方向是从服务器端向客户端，并返回响应给客户端。此时，HTTP 保持连接，并没有关闭，如图 8-4 中②所示。

SSE 事件通信通道的客户端一端建立 EventInput 信道，用于读取 InboundEvent（进站事件）。EventInput 类继承自 ChunkedInput，ChunkedInput 允许将数据在一条信道中分次传输；EventInput 类的泛型类型为 InboundEvent，即上述的 onEvent() 方法待处理的类型。当进站事件到达时，MessageBodyReader 的 SSE 实现类 InboundEventReader 将解析数据并反序列化为 InboundEvent 类型的数据。

SSE 事件通信通道的服务器一端建立 EventOutput 信道，用于写入 OutboundEvent（出站事件）。EventOutput 类继承自 ChunkedOutput，ChunkedOutput 允许在发送出站事件后，HTTP 连接通过 HTTP 1.1 的 Keep-Alive 保持连接，出站事件写入 HTTP 响应头后，EventOutput 信道等待更多的服务器推送事件。出站事件的序列化写入，由 MessageBodyWriter 的 SSE 实现类 OutboundEventWriter 完成。

接下来的三个步骤是在这个信道上完成的。第三步如图 8-4 中③所示，客户端向服务器发送 POST 请求。第四步如图 8-4 中④所示，服务器端接收后会向 EventOutput 信道写入数据。最后，如图 8-4 中⑤所示，客户端监听到信道中有数据到达，将读取并处理推送事件。

到此，服务器推送事件的流程执行完一遍。信道的连接可以由服务器主动关闭，或者由客户端请求关闭。关闭时机可以由业务灵活控制。

2. 广播模式

通过上述的发布-订阅模式，我们对 Jersey 的 SSE 支持包和通信机制有了比较全面的了解。广播模式与发布-订阅模式相比，客户端的实现相同，服务器端推送事件的写入从端到端的 EventInput 信道换成了多点广播类 SseBroadcaster。SseBroadcaster 类继承自 Broadcaster 类，泛型类型为 OutboundEvent（出站事件）。SseBroadcaster 类提供了一次关闭多点信道的

方法 closeAll()，可以根据业务需要，在完成广播事件后执行。

如图 8-5 所示，Jersey-sse 包中的成员类都已经一一介绍。

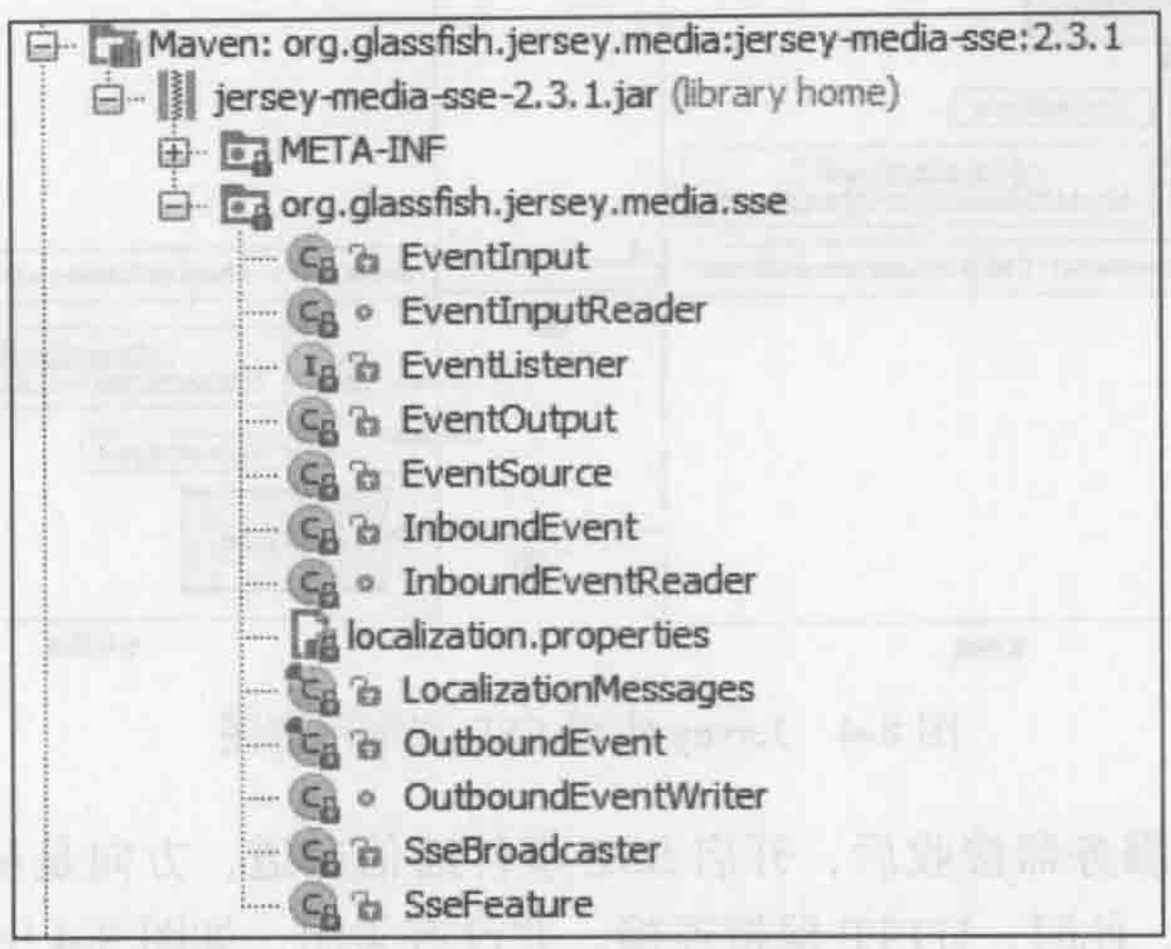


图 8-5 Jersey 的 SSE 实现包示意图

到此，Jersey 的 SSE 支持包讲述完毕。接下来将讲述如何使用 Jersey 的 SSE 支持包来实践上述两种模式。

8.2.3 SSE 实现

在我们的 REST 应用中使用 Jersey 实现 SSE 的过程可以概括为两个环节，第一个环节是基于本章上述理论开发新的或维护已有的：REST 入口类 Application、资源类以及单元测试类；第二个环节是集成测试，目的是验证资源方法的功能。

阅读指导

8.2.3 节示例所在目录是 :jax-rs2-guide\sample\8\sse

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/8/sse

1. Application 类

支持 SSE 的 REST 应用入口类 AirResourceConfig 是 Application 类的子类，示例代码如下。

```
@ApplicationPath("/event/*")// 关注点 1: 为 SSE 定义资源路径
public class AirResourceConfig extends ResourceConfig {
    public AirResourceConfig() {
// 关注点 2: 注册 Feature 和资源类
        super(SseFeature.class, AirSsePubSubResource.class,
AirSseBroadcastResource.class);
    }
}
```

在这段代码中，定义了SSE服务的根资源路径为/event/*，见关注点1。手动注册了SseFeature类（在Jersey 2.8版本之后可以自动探测），用以标识该服务具备处理SSE的特征。同时注册了发布-订阅资源类AirSsePubSubResource和广播资源类AirSseBroadcastResource，见关注点2。

2. 发布-订阅资源类

资源类AirSsePubSubResource用于支持发布-订阅模式的SSE，实现了发布-订阅模式中阐述的流程，示例代码如下。

```
@Path("pubsub")// 关注点 1: 为发布-订阅模式定义资源路径
public class AirSsePubSubResource {
    private static EventOutput eventOutput = new EventOutput();
    @GET// 关注点 2: 提供 SSE 事件输出通道的资源方法
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput publishMessage() throws IOException {
        return eventOutput;
    }
    @POST// 关注点 3: 执行业务逻辑和写入 SSE 通道的资源方法
    public void saveMessage(String message) throws IOException {
       LOG.info("post message=" + message);
       eventOutput.write(new OutboundEvent.Builder()
       .id(System.nanoTime() + "")
       .name("post message")
       .data(String.class, message).build());
    }
}
```

在这段代码中，资源地址定义了发布-订阅模式的资源地址为“pubsub”，见关注点1。GET方法用于公布SSE通信通道，POST方法用于处理业务和写入SSE事件，两者逻辑上是先后被调用的关系。publishMessage()方法公布了推送事件输出通道的接口，返回值是前面讲述的EventOutput类型的推送事件输出信道，见关注点2。POST方法saveMessage()用于接收客户端提交的信息，并将其写入推送事件输出信道，根据HTML5的SSE规范，出站事件OutboundEvent的数据结构包含三个主要信息：id、name和data，见关注点3。

3. 发布-订阅测试类

SseBroadcaseTest类继承了第7章讲述的Jersey的测试框架类JerseyTest，省去了样板测试代码，本例只需注册SSE特征类SseFeature和发布-订阅测试类AirSsePubSubResource。示例代码如下。

```
public class SsePubSubTest extends JerseyTest {
...
@Test
public void testEventSource() throws InterruptedException, URISyntaxException {
...
```

```
final CountDownLatch latch = new CountDownLatch(testCount);
//关注点 1: 创建 EventSource 实例
final EventSource eventSource = new EventSource(target().path(ROOT_PATH)) {
    private inti;
    @Override
    public void onEvent(InboundEvent inboundEvent) {
        try {
            //关注点 2: SSE 输入信道监听逻辑
            LOG.info("Received: " + inboundEvent.getId() + ":"
            + inboundEvent.getName() + ":" + new String(inboundEvent.getRawData()));
            Assert.assertEquals(messagePrefix + i++,
            inboundEvent.readData(String.class));
            latch.countDown();
        } catch (ProcessingException e) {
            e.printStackTrace();
        }
    }
};
for (inti = 0; i<testCount; i++) {
    target().path(ROOT_PATH).request().post(Entity.text(messagePrefix + i));
}
try {
    latch.await();
} finally {
    eventSource.close();
    }
}
```

在这段代码中，用于测试发布-订阅模式功能和流程的方法 testEventSource() 是一个集成测试方法，模拟了客户端监听并处理 SSE 事件的流程。在该测试方法中，首先创建 EventSource 实例，并通过该实例请求服务器端 GET 方法以获得 SSE 事件输出信道，见关注点 1。在 EventSource 实现中，onEvent() 方法中做了三件事：第一是打印输出服务器推送事件的内容，内容包括前面讲述的 SSE 规范定义的数据结构：id、name 和 data；第二是使用相等断言测试服务器端返回数据是否符合预期；第三是调用同步器 CountDownLatch 实例的 countDown() 方法（原因见下文），见关注点 2。

测试代码中使用 CountDownLatch 的原因是在测试流程中，通过 GET 订阅服务器 SSE 事件通知和通过 POST 发送数据两个异步操作，而我们期待的顺序是在执行完 POST 代码块后，前面代码块中定义的监听和处理推送事件的 onEvent() 方法才被执行，即前者先被执行但前者的回调后被触发。为此，如 8.2.1 节所述，测试流程在 POST 代码块后，调用了 CountDownLatch 实例的 await() 方法，使得测试主线程被阻塞而不是执行完，以便等待回调被执行。所以，onEvent() 方法的第三步是调用 CountDownLatch 实例的 countDown() 方法来递减其内部计数器，确保主线程最终被唤醒。

4. 广播资源类

资源类 AirSseBroadcastResource 用于支持广播模式的 SSE，示例代码如下。

```
@Path("broadcast")//关注点 1: 为广播模式定义资源路径
public class AirSseBroadcastResource {
    private static final BlockingQueue<BroadcastProcess>processQueue =
    new LinkedBlockingQueue<>(1);
    @Path("book")
    @POST
    public Boolean postBook(@DefaultValue("0") @QueryParam("total")int total,
    String bookName) {
        //关注点 2: 调用 BroadcastProcess 实例实现广播
        final BroadcastProcess broadcastProcess = new BroadcastProcess(total, bookName);
        processQueue.add(broadcastProcess);
        Executors.newSingleThreadExecutor().execute(broadcastProcess);
        return true;
    }
//关注点 3: 广播处理线程类
static class BroadcastProcess implements Runnable {
...
public void run() {
...
        OutboundEvent.Builder eventBuilder = new OutboundEvent.Builder()
        .mediaType(MediaType.TEXT_PLAIN_TYPE);
        OutboundEvent event = eventBuilder.id(processId + "")
        .name("New Book Name")
        .data(String.class, bookName).build();
        broadcaster.broadcast(event);
...
}
```

在这段代码中，资源地址定义了广播模式的资源地址为“broadcast”，见关注点 1。实现广播的核心代码块位于内部线程类 BroadcastProcess 中，该线程类使用广播类 SseBroadcaster 实例将 SSE 事件广播出去，见关注点 3。

POST 方法 postBook() 资源地址是 broadcast/book 用于接收客户端提交的收听广播的客户端数量和最新图书信息。一旦接收数据，该方法会动态生成一个线程类 BroadcastProcess 的实例，并尝试将其缓存到 BlockingQueue 实例中。如 8.2.1 节所述，BlockingQueue 的 add() 方法会在尝试入队失败时，直接抛出异常，意味着广播类只缓存最新的一条图书资源信息，直接失败的处理可以避免客户端请求线程阻塞 / 挂起，见关注点 2。缓存队列会被 GET 方法消费，由于篇幅所限，因此没有展示全部代码，读者可以从源代码中查阅。

广播测试类 SseBroadcaseTest 与前一测试非常类似，不再讲述相同部分。需要额外说明的是，在每一个客户端的 GET 请求中，生成 EventSource 类实例的过程是通过 EventSource 类的内部 Builder 完成的，并没有自动打开 SSE 信道，因此在注册监听、覆盖 onEvent() 方法后，显式调用 open() 方法，读者可在源代码中查阅相关内容。

5. 集成测试

集成测试更为普遍的方式是使用 cURL 命令直接对资源地址进行访问，即时检测资源方法的功能，或者使用 Shell 脚本编写 cURL 请求。本例将使用 cURL 命令分别对上述的两种模式的实现进行集成测试。

(1) 测试发布-订阅

启动示例服务，然后使用两个终端 / 控制台测试发布-订阅流程。在第一个终端中输入如下 cURL 命令，然后等待其输出。这一步请求了 GET 方法并建立了 HTTP 连接。

```
curl -H "Accept: text/event-stream" --url http://localhost:8080/sse/event/pubsub
```

在第二个终端中依次输入如下 cURL 命令，以模拟多次提交 POST 请求的流程。

```
curl -X POST --data "Java1.5" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.6" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.7" --url http://localhost:8080/sse/event/pubsub
curl -X POST --data "Java1.8" --url http://localhost:8080/sse/event/pubsub
```

第一个终端应输出如下信息，展示了服务器端在处理每一次 POST 请求时，向事件输出通道写入出站事件。

```
event: post message
id: 22127543063654
data: Java1.5
event: post message
id: 22134553708520
data: Java1.6
event: post message
id: 22141048454287
data: Java1.7
event: post message
id: 22149255904866
data: Java1.8
```

(2) 测试广播

启动服务，然后根据上述测试场景的逻辑，按照图 8-6 所示，使用三个终端 / 控制台测试广播流程。

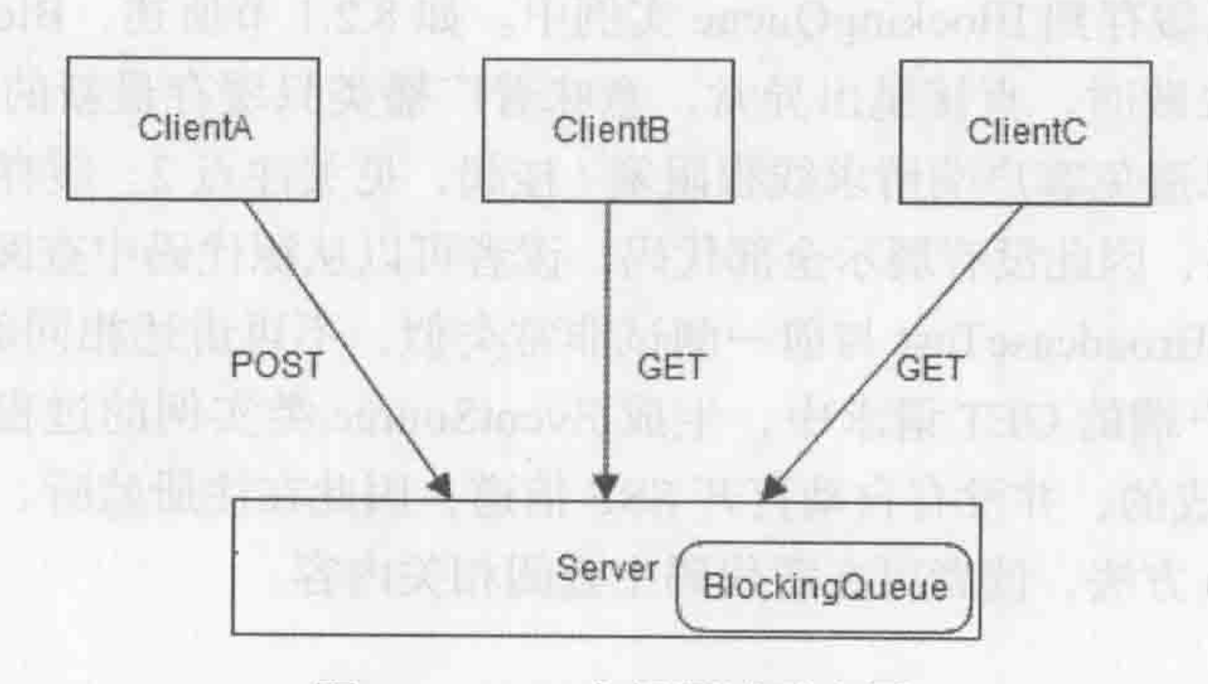


图 8-6 SSE 广播测试示意图

在图 8-6 所示的第一终端 ClientA 中输入如下 cURL 命令，其中，数据为 jax-rs2-guide，参数 total=2，当两个客户端连接到服务器后，向全部客户端发起广播。

```
curl -X POST --data "jax-rs2-guide" http://localhost:8080/sse/event/broadcast/book?total=2
```

接下来，在第二个终端 ClientB 输入如下 cURL 命令，终端会出现阻塞，不去管它，继续第三个终端 ClientC 的命令输入。

```
curl -H "Accept: text/event-stream" http://localhost:8080/sse/event/broadcast/book?clientId=1
```

在第三个终端输入如下 cURL 命令后，第二和第三个终端会同时收到服务器的广播消息。

```
curl -H "Accept: text/event-stream" http://localhost:8080/sse/event/broadcast/book?clientId=2
```

事件消息中的 data 信息即是 POST 提交的内容。

```
event: New Book Name
id: 33035275182692
data: jax-rs2-guide
```

8.3 异步通信

JAX-RS 2.0 定义了 REST 的异步通信 API，这是本章的另外一个主题。REST 的异步通信是提高 REST 式的 Web 服务的性能、执行效率不可或缺的技术，同时异步通信可以增强开发过程中业务实现的功能性和灵活性。本章前述的服务器推送技术属于异步通信的一种场景，即服务器主动向客户端推送数据和资源状态的变化。此外，还有一种异步通信的场景是当客户端提交请求后，服务器通过异步的方式接收并处理请求，此时客户端线程可以自由选择是否阻塞线程等待处理结果。该场景适用于处理信息较大、处理时间较长的情况，和服务器推送技术既有相似之处又有不同。

在 JAX-RS 2.0 出现之前，REST 的异步通信实现方式并没有统一的标准。在进入 JAX-RS 2.0 的异步规范这个主题之前，我们先来简单了解一下 REST 应用中都出现了哪些异步解决方案，从而在宏观上掌握异步通信的机制。

阅读指导

8.3 节示例所在目录是 jax-rs2-guide\sample\8\asyn-rest。

源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/8/asyn-rest。

1. Polling 异步通信

Polling 异步通信方案使用的技术结合了前述的 HATEOAS 和 Web Link，以及 Polling 技术，如图 8-7 所示。

服务器会在接收请求后立即（以 HATEOAS 或者 Web Link 技术）返回给客户端一个查询处理结果的资源地址，并结束这一次的请求-响应流程，HTTP 连接关闭，HTTP 状态码为

202（注意：不是 HTTP 状态 200 OK，202 代表服务器已接收请求但尚未处理）。客户端通过轮询机制，向新的 REST 地址发起请求并获得该处理的进度状态（完成状态为 HTTP 状态码 100，如果请求过期或者资源地址错误，则 HTTP 状态码为 404，即找不到），并最终在获取处理完毕信息后结束轮询。

这种解决方案比起同步处理的优点是客户端可以即时得到服务器的反馈，并在获得最终结果之前，有机会处理后续业务。另外，Polling 技术不需要对服务器和客户端使用额外的第三方支持包，易于实现。缺点在前面关于 Polling 一节已经阐述过。

2. Web Hook 异步通信

Web Hook 解决方案是指在客户端发送请求时，将一个回调地址同时发送给服务器，服务器接收响应后，异步处理请求并对本次请求即刻做出响应，客户端随即处理其他业务并监听回调。服务器端在响应客户端后，继续以异步的方式处理刚才的请求，在处理完毕后，通过回调地址通知客户端处理结果，如图 8-8 所示。

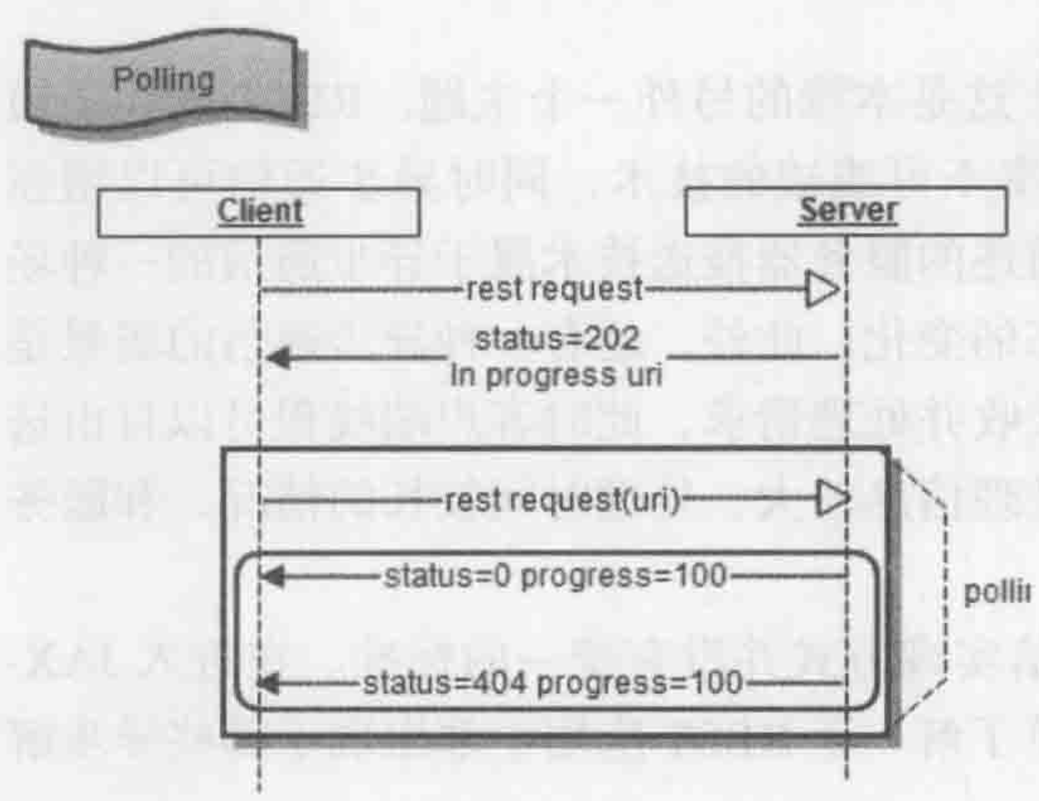


图 8-7 客户端轮询实现异步通信示意图

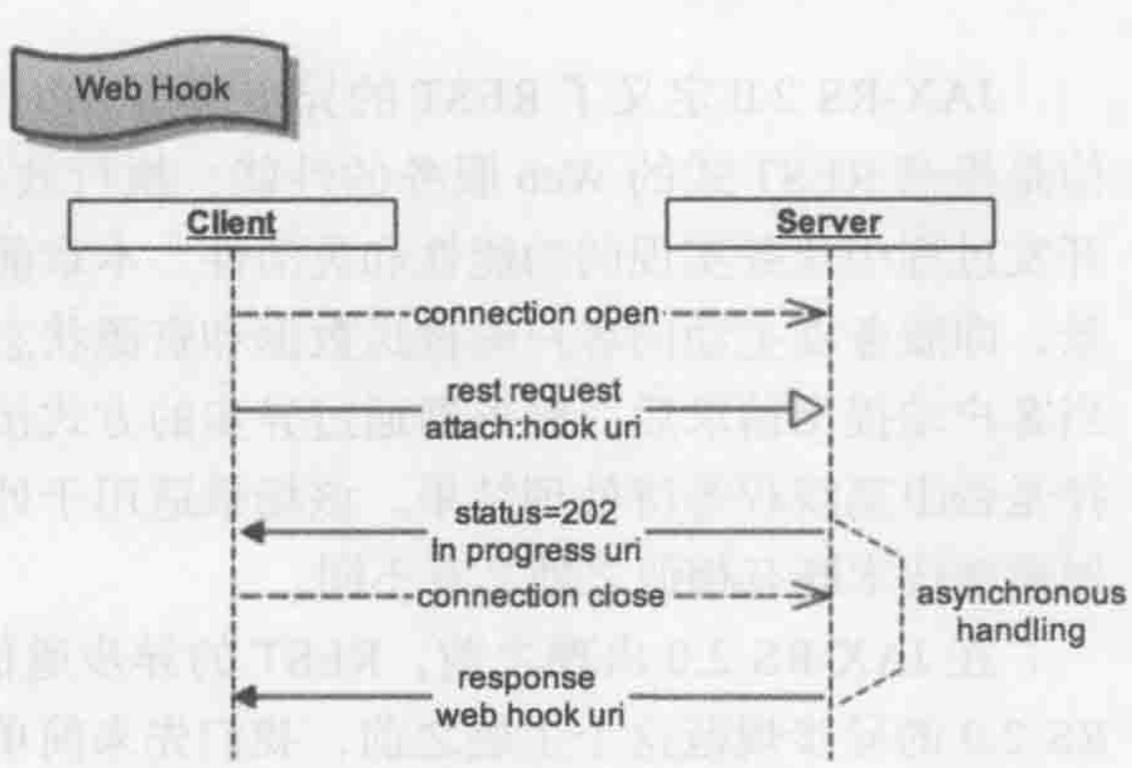


图 8-8 Web Hook 实现异步通信示意图

Web Hook 解决方案的优点是覆盖了 Polling 方案并且没有 Polling 方案中无效的轮询负载。但是，这种方案无法在浏览器作为客户端的场景中实施，因为浏览器无法提供一个回调地址给服务器。因此，该方案适用于另外一个服务器作为客户端的场景。另外，和后面的解决方案比起来，该方案还是多出了一次服务器回调客户端的 HTTP 连接。

3. Comet 异步通信

Comet 技术在服务器推送业务中的处理流程同样适用于这里的异步通信，客户端发送请求后，可以继续执行后续业务并监听服务器返回的处理结果的通知，如图 8-9 所示。Comet 解决方案的优点是异步通信可以在一次请求-响应的模型中完成。缺点在 8.1 节已经阐述过。

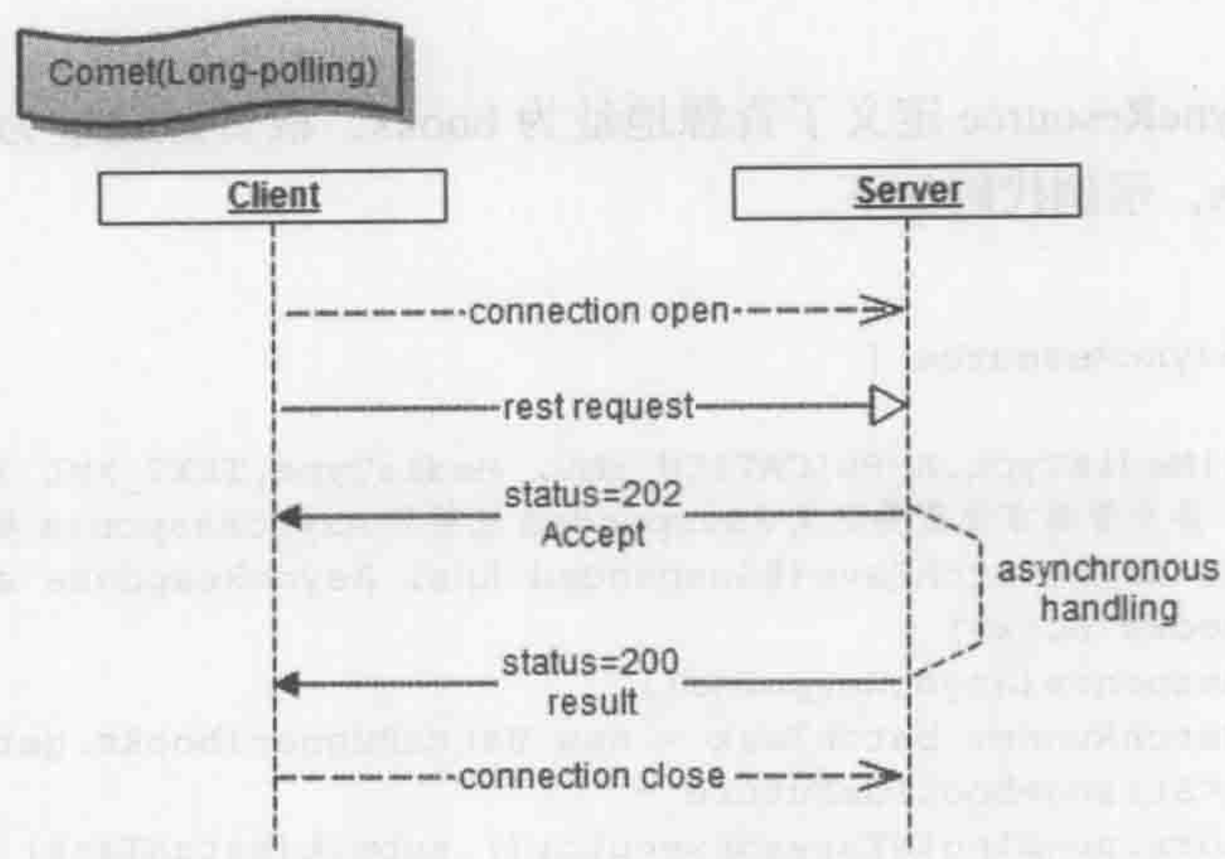


图 8-9 Comet 实现异步通信示意图

4. HTML5 异步通信

HTML5 包含 SSE 和 Web Socket。SSE 用于服务器推送事件，但并不仅限于这样使用，上文的相关示例中已经展示了处理异步通信的能力。Web Socket 无疑非常适用于当下讨论的场景。由于 JAX-RS 2.0 和 Jersey 并没有相关的描述，因此本书不涉及 Web Socket 的讲述，如果读者有兴趣，那么可以参考前面推荐的资料，学习其如何实现。

8.4 JAX-RS 2.0 实现异步通信

JAX-RS 2.0 的异步处理是通过两个线程实现的，其中一个线程用于处理客户端的请求，另一个线程是为此次请求新生成对，用于处理具体业务。在后一个线程处理开始前，前一个线程可以响应客户端请求正在执行，然后进入挂起状态，保持连接。后一个线程执行完毕后，唤醒前一个线程。JAX-RS 2.0 的异步实现过程中，线程的管理是由容器实现的，这是 Java EE 7 中 JSR 236 规范定义的功能，读者可以参见《Java EE 7 Essentials》一书的第 10 章。可以说，Java 领域的并发处理，Java SE 5.0 是一个里程碑，在 Java EE 中与之对应的就是 7.0 版本了。在容器级别有了并发的支持，客户端等待服务器的响应就可以由一个 Future 实现，感觉上就像 Java SE 开发中，等待同一个 JVM 的另一个线程一样。

了解了 JAX-RS 2.0 异步处理的流程后，接下来我们进入实践。

8.4.1 服务端实现

在 JAX-RS 的服务器端，实现异步通信包括两个技术点，一个是资源方法中对 AsyncResponse 的使用，另一个是对异步通信的 CompletionCallback 和 TimeoutHandler 接口的实现，本节将分别讲述。

1. 异步资源类

异步资源类 AsyncResource 定义了资源地址为 books，根资源地址为 rest-async，资源路径为 rest-async/books，示例代码如下。

```
@Path("books")
public class AsyncResource {
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.TEXT_XML })
    //关注点 1: 异步资源方法需要定义 @Suspended 注解和 AsyncResource 参数
    public void asyncBatchSave(@Suspended final AsyncResponse asyncResponse,
        final Books books) {
        configResponse(asyncResponse);
        final BatchRunner batchTask = new BatchRunner(books.getBookList());
        Future<String>bookIdsFuture =
        Executors.newSingleThreadExecutor().submit(batchTask);
        String ids;
        try {
            ids = bookIdsFuture.get();
        //关注点 2: 请求响应线程被唤醒
        asyncResponse.resume(ids);
        } catch (InterruptedException | ExecutionException e) {
            LOGGER.error(e.getMessage());
        }
    }
```

在这段代码中，POST 方法 asyncBatchSave() 用于处理批量保存业务（模拟一个耗时的处理场景）。该方法包含两个参数，第一参数是异步响应类 AsyncResponse 的实例（习惯上，位于前面的参数是上下文环境变量参数，业务参数置后），使用注解 @Suspended 来标识，见关注点 1。在执行过程中，请求处理线程被挂起，直到异步请求处理结束，异步响应实例的 resume() 方法被调用，请求处理线程才被唤醒，将 resume() 方法的参数作为返回值响应给客户端，见关注点 2。

2. 超时和回调

JAX-RS 2.0 定义了处理完成回调接口 CompletionCallback。当请求处理完成时，CompletionCallback 的实例会被回调。实现 onComplete() 方法可以监听请求处理完成事件并实现相关业务流程。CompletionCallback 的实现可以作为 AsyncResource 的 register() 方法的参数来配置，这样配置后，AsyncResource 实例会在 resume() 被调用后执行回调方法 onComplete()，示例代码如下。

```
asyncResponse.register(new CompletionCallback() {
  @Override
  public void onComplete(Throwable throwable) {
      if (throwable == null) {
          LOGGER.info("CompletionCallback-onComplete: OK");
      } else {
```

```
            LOGGER.info("CompletionCallback-onComplete: ERROR: " + throwable.getMessage());
        }
    }
});
```

JAX-RS 2.0 定义了连接断开回调接口 ConnectionCallback。当请求—响应模型的连接断开时，ConnectionCallback 的实例会被回调。实现 onDisconnect() 方法可以监听连接断开事件并实现相关业务，比如主动唤醒 AsyncResource 实例并设置 HTTP 状态码为 410、客户端请求资源不可用（Response.Status.GONE）来完成响应。ConnectionCallback 的实现可以作为 AsyncResource 的 register() 方法的参数来配置，示例代码如下。

```
asyncResponse.register(new ConnectionCallback() {
  @Override
  public void onDisconnect(AsyncResponse disconnected) {
      disconnected.resume(Response.status(Response.Status.GONE)
      .entity("disconnect!").build());
  }
});
```

JAX-RS 2.0 定义了超时处理器接口 TimeoutHandler 专门处理异步响应类超时，当预期的超时时间到达后，TimeoutHandler 的实例会被调用。实现 handleTimeout() 方法可以监听超时事件并处理相关业务，比如主动唤醒 AsyncResource 实例并设置 HTTP 状态码为 503、服务器端服务不可用（Response.Status.SERVICE_UNAVAILABLE）来完成响应。TimeoutHandler 的实现可以作为 AsyncResource 的 setTimeoutHandler() 方法的参数来配置。AsyncResource 的 setTimeout() 方法用于设置超时时间，默认情况下 AsyncResource 永不超时。示例代码如下。

```
asyncResponse.setTimeoutHandler(new TimeoutHandler() {
  @Override
  public void handleTimeout(AsyncResponse asyncResponse) {
      asyncResponse.resume(Response.status(Response.Status.SERVICE_UNAVAILABLE)
      .entity("Operation time out.").build());
  }
});
asyncResponse.setTimeout(TIMEOUT, TimeUnit.SECONDS);
```

8.4.2 客户端实现和测试

相应的，JAX-RX 为客户端提供了用于执行异步请求的 API。开发者使用这样的方法可以实现对服务器端异步的请求，同时也可以使用客户端的异步调用实现对服务器端异步通信的测试。

1. 异步测试类

异步测试类 TIAsyncJFTTest 继承了 Jersey 测试框架类 JerseyTest，异步测试没有额外的

配置，示例代码如下。

```
public class TIAsyncJFTTest extends JerseyTest {
@Test
public void testAsyncBatchSave() throws InterruptedException, ExecutionException {
  List<Book>bookList = new ArrayList<>(COUNT);
  try {
...
// 关注点 1: 请求方法的调用使用 AsyncInvoker 实例
        final AsyncInvokerasync = request.async();
        final Future<String>responseFuture = async.post(booksEntity, String.class);
        String result = null;
        try {
// 关注点 2: 异步获取服务器的最终响应
         result = responseFuture.get(AsyncResource.TIMEOUT + 1, TimeUnit.SECONDS);
      } catch (TimeoutException e) {
...
}
```

在这段代码中，客户端使用 AsyncInvoker 接口的 post() 方法提交异步请求，见关注点 1。该方法返回 Future 接口的实例，客户端线程可以以非阻塞的方式处理其他业务流程，然后调用 Future 的 get() 方法获取服务器处理结果，见关注点 2。

2. 异步回调

可以在 AsyncInvoker 接口的 post() 方法中，定义一个 InvocationCallback 接口的实例，实现 REST 调用的回调处理，示例代码如下。

```
final Future<String>responseFuture = async.post(booksEntity,
  new InvocationCallback<String>() {
  @Override
// 关注点 1: 服务器请求处理成功的回调
  public void completed(String result) {
      LOGGER.debug("On Completed: " + result);
  }

  @Override
// 关注点 2: 服务器请求处理失败的回调
  public void failed(Throwable throwable) {
      LOGGER.debug("On Failed: " + throwable.getMessage());
      throwable.printStackTrace();
  }
});
String result = null;
try {
  result = responseFuture.get(AsyncResource.TIMEOUT + 1, TimeUnit.SECONDS);
...
```

在这段代码中，completed() 方法用于监听并处理 REST 调用成功事件，见关注点 1。failed() 方法用于监听并处理 REST 调用失败事件，见关注点 2。

3. 集成测试

启动示例服务，然后在终端 / 控制台输入如下 cURL 命令来测试 REST 异步处理。示例中的这个资源方法模拟实现了一个耗时的图书处理过程，预期的返回值为两个图书资源的 ID。

```
curl -X POST -H "Content-Type:application/xml"
http://localhost:8080/asyn-rest/rest-async/books --data-binary "<books><bookList><book
bookName='jax-rs2-guide'/><book bookName='jsf-richfaces4-guide'/></bookList></books>"
63 64
```

8.5　本章小结

本章对推送和异步进行了讲述。其中前半部分的两节分别讲述了 REST 通信中服务器推送事件的 4 种技术和 Jersey 对服务器端推送事件（SSE）技术的支持。后半部分的两节分别讲述 REST 的异步通信的 4 种技术和 JAX-RS 2.0 的异步请求处理实现。下一章，我们开始讲述 Jersey 1.x 到 Jersey 2.x 的迁移。

Chapter 9 第 9 章

Jersey 1.x 迁移

Jersey 1.x 作为 JAX-RS 的参考实现，当前版本为 1.18（可参考 https://jersey.java.net/download.html）。相信有相当数量的开发者首先接触的是 Jersey 1.x，因此无论 Java EE 容器版本的升级还是对 JAX-RS 2.0 规范的支持，掌握从 Jersey 1.x 迁移到 Jersey 2.x 这一过程的要点是非常必要的。本章将讲述迁移过程中需要注意的事项。

9.1 变更 Maven 依赖定义

首先需要注意的是，Jersey 1.x 使用的包名是 Java 开发者亲切和熟识的 com.sun，而 Jersey 2.x 使用的是 org.glassfish。抛开这一变化的商业目的，我们作为开发者，从 2.x 的命名中更容易获知其归属于 GlassFish 项目集。因此，在迁移项目的过程中，依赖定义不是简单地将版本号修改即可，而是要全面排查 com.sun 这个包名关键字是否被 org.glassfish 替换掉。

除了项目名称坐标的变更，扩展包的命名也发生了变化。Jersey 1.x 对扩展包的命名使用的是 com.sun.jersey.contribs，而 Jersey 2.x 使用的是 org.glassfish.jersey.ext。

以上所述的变更包含如下所示的对 Jersey 1.x 的依赖定义，供读者参考。

```
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-servlet</artifactId>
</dependency>
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-client</artifactId>
</dependency>
<dependency>
```

```
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-apache-client</artifactId>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-spring</artifactId>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey.jersey-test-framework</groupId>
    <artifactId>jersey-test-framework-external</artifactId>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
  </dependency>
```

9.2 客户端迁移

客户端的代码部分迁移点比较多，这是因为在 JAX-RS 中没有提供可以遵循的标准。Jersey 1.x 的客户端的实现没有标准依据，自行进行了实现，而 Jersey 2.x 遵循的是 JAX-RS 2.0 才引入的客户端接口标准。听起来这似乎很难对应，但要说明的是，Jersey 作为 JAX-RS 的参考实现，为这个标准规范提供了反向的参考，在 JAX-RS 2.0 的客户端定义中，借鉴了 Jersey 1.x 的 API，因此代码迁移还是有据可循的。主要（并不是全部）的迁移包括 Client 接口、WebTarget 接口和 QueryParam 三个部分。

9.2.1 Client 接口迁移

Client 接口是 JAX-RS 2.0 中定义的。在 Jersey 1.x 中使用 Apache 的 HTTP 客户端来实例化。而 Jersey 2.x 中使用 JAX-RS 2.0 定义的 ClientBuilder 来实例化，示例代码如下。

```
//Jersey 1.x
DefaultApacheHttpClientConfig config = new DefaultApacheHttpClientConfig();
client = ApacheHttpClient.create(config);
//Jersey 2.x
ResourceConfig config=new ResourceConfig();
client=ClientBuilder.newClient(config);
```

（1）属性设置

客户端配置在 Jersey 2.x 版本中更直接，即使用 property() 方法。而 Jersey 1.x 中要先获取 features，然后将属性的键值对赋给 features，示例代码如下。

```
//Jersey 1.x
config.getFeatures().put(FeaturesAndProperties.FEATURE_DISABLE_XML_SECURITY, true);
//Jersey 2.x
config.property(MessageProperties.XML_SECURITY_DISABLE, Boolean.TRUE);
```

(2) 设置超时

Jersey 2.x 使用 ClientProperties 类为属性键定义常量，Jersey 1.x 使用的是配置类本身。超时配置分为连接超时和读取超时，前者用于控制通信建立连接的时间，后者用于控制请求—响应过程中，数据从服务器端下行到客户端的时间，示例代码如下。

```
//Jersey 1.x
config.getProperties().put(ClientConfig.PROPERTY_CONNECT_TIMEOUT, this.connectionTimeout);
config.getProperties().put(ClientConfig.PROPERTY_READ_TIMEOUT, this.readTimeout);
//Jersey 2.x
config.property(ClientProperties.CONNECT_TIMEOUT, this.connectionTimeout);
config.property(ClientProperties.READ_TIMEOUT, this.readTimeout);
```

(3) 配置代理服务器

Jersey 2.x 的代理服务器的配置在 2.5 版本之前依然需要依赖 Apache 的 HTTP 客户端。这种配置在 Jersey 2.5 版本中已经去掉，以实现其配置的完整性，使 Jersey 默认的客户端（Jersey 客户端的代理还可以在 Apache 连接器中设置）功能更趋完善，迁移示例代码如下。

```
//Jersey 1.x
config.getProperties().put(ApacheHttpClientConfig.PROPERTY_PROXY_URI, proxyUrl);
//Jersey 2.5-
config.property(ApacheHttpClientConfig.PROXY_URI, proxyUrl);
//Jersey 2.5+
config.property(ClientConfig.PROXY_URI, proxyUrl);
```

9.2.2 WebTarget 接口迁移

WebTarget 这个概念是在 JAX-RS 2.0 提出的，在 Jersey 1.x 中对应的是 WebResource。两者的功能类似，实例化的实现上区别不是很大。两者的主要区别只是名字，显然规范中的名字更容易理解其含义，而 Jersey 1.x 中的容易和资源类混淆。因此，这部分的迁移比较容易。迁移示例代码如下。

```
//Jersey 1.x
WebResource createWebResource(final String url, final String username, final String
password) throws HttpConnectionException {
  final Client client = createClient(true);
  client.addFilter(new HTTPBasicAuthFilter(username, password));
  final WebResource webResource = client.resource(url);
  return webResource;
}
//Jersey 2.x
WebTarget createWebResource(final String url, final String username, final String
password) throws HttpConnectionException  {
  final Client client =createClient(true);
  WebTarget target=client.target(url);
  target.register(new HttpBasicAuthFilter(username, password));
  return target;
}
```

JAX-RS 2.0 规定了请求中媒体类型定义的方法名称为 request，在 Jersey 1.x 中对应的是 accept() 方法。另外，写操作在 JAX-RS 2.0 一律接收 Entity 类实例，这在 Jersey 1.x 中没有规定。迁移示例代码如下。

```
//Jersey 1.x
result = webResource.accept(MediaType.APPLICATION_XML).get(this.returnClass);
result = webResource.accept(MediaType.APPLICATION_XML).post(this.returnClass, postXml);
//Jersey 2.x
result = webTarget.request(MediaType.APPLICATION_XML).get(this.returnClass);
Entity<String> postXmlEntity=Entity.entity(postXml,MediaType.APPLICATION_XML);
result = webResource.request(MediaType.APPLICATION_XML)
.post(postXmlEntity,this.returnClass);
```

9.2.3 QueryParam

JAX-RS 2.0 为 WebTarget 定义了 queryParam() 来设置查询条件键值对。在 Jersey 1.x 中使用的是键值对 Map。迁移示例代码如下。

```
//Jersey 1.x
final MultivaluedMap<String, String> queryParams = new MultivaluedMapImpl();
queryParams.add("start", indexStart);
queryParams.add("pagesize", pageSize);
final WebResource webResourceWithParameters = webResource.queryParams(queryParams);
//Jersey 2.x
WebTarget queryWebTarget =queryTarget.queryParam("start", indexStart);
queryWebTarget=queryWebTarget.queryParam("pagesize", pageSize);
```

需要注意的是，WebTarget 的链式赋值并不改变其本身，这在 5.1 节的讲述中已经说明。因此，在循环代码块中不能像 Jersey 1.x 为键值对 Map 赋值那样使用，而应当接收每次赋值的结果作为下一次赋值的句柄。迁移示例代码如下。

```
//Jersey 1.x
for (final GenericQueryParameter param : parameters) {
        queryParams.add(param.getParameterName(), param.getParameterValue());
    }
webResourceWithParameters = webResource.queryParams(queryParams);
//Jersey 2.x
for (final GenericQueryParameter param : parameters) {
        queryWebTarget = queryWebTarget.queryParam(param.getParameterName(),
param.getParameterValue());
}
```

9.3 服务器端迁移

在服务器端，代码部分的迁移点并不是很多。配置上，web.xml 文件中定义的 Servlet 名称，需要修改为相应的包名和配置项名称，这在第 2 章的示例中已经详述。关于配置部分可参考第 10 章。需要注意的是，Jersey 2.0 引入了 HK2 作为依赖注入框架。但是，实际的项目

中更多的是和 Spring 集成，因此下面简述在迁移中集成 Spring 的变化。

由于 Jersey 1.x 集成 Spring 的最高版本是 Spring 2.6.5，因此无法使用 Spring 3.x 的特性。Jersey 2.x 引入 Spring 3.x 作为集成版本，其对比配置的示例代码如下。

```
//Jersey 1.x
<servlet>
  <servlet-name>Jersey Spring Web Application</servlet-name>
   <servlet-class>com.sun.jersey.spi.spring.container.servlet.SpringServlet</servlet-class>
   <init-param>
      <param-name>com.sun.jersey.config.feature.DisableXmlSecurity</param-name>
      <param-value>true</param-value>
  </init-param>
  <init-param> <param-name>com.sun.jersey.config.feature.logging.DisableEntitylogging</param-name>
      <param-value>false</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey Spring Web Application</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>

//Jersey 2.x
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/Java EE" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE http://java.sun.com/
xml/ns/Java EE/web-app_3_0.xsd"
  version="3.0">
  <display-name>simple-service</display-name>
  <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:applicationContext.xml</param-value>
  </context-param>
</web-app>
```

本章没有讲述 Jersey 2.x 小版本之间的迁移，因为在 Jersey 快速的发展中，其内部 API 在不断优化中，详见 https://jersey.java.net/documentation/latest/user-guide.html#migration。本书的示例代码会根据最新版本的 API 进行调整，并在 GitHub 上作为分支提供给读者。

9.4 本章小结

本章讲述了 Jersey 1.x 迁移到 Jersey 2.x 的细节，这些内容来自笔者的实际工作经验。9.1 节列举了依赖包的变化，9.2 节对 REST 客户端接口的变化进行了阐述并给出了实例，9.3 节简述了服务器端的变化。下一章，我们开始讲述 JAX-RS 的调优内容。

第 10 章 Chapter 10

JAX-RS 调优

基于 JAX-RS 的项目是部署在 Servlet 容器或者 Java EE 容器的 Web 服务，因此要提高这类项目的性能，除了在编码阶段需要注意代码的质量，在配置部署阶段还要关注网络负载、Jersey 参数配置以及对 Java 虚拟机调优，在运行阶段还要对运行平台的操作系统、服务器软件进行监控和调优。本章将讲述如何为基于 JAX-RS 的项目进行调优。

阅读指南 第 10 章示例所在目录是 jax-rs2-guide\sample\10\simple-service10。
源代码地址：https://github.com/feuyeux/jax-rs2-guide/tree/master/sample/10/simple-service10。

10.1 使用缓存优化负载

系统的网络吞吐能力是其性能的重要指标，我们可以使用操作系统自带的工具进行监控。但是，监控只能判断网络的负载情况，而降低网络负载是提高网络吞吐能力的一个有效的手段。使用动静分离和缓存是降低系统和网络负载的手段之一。本节将讲述基于 JAX-RS 的项目的缓存实践。

10.1.1 缓存协商

Jersey 支持使用通用的 HTTP 头进行缓存协商。在 HTTP 头字段中，Expires、Cache-Control 和 Last-Modified 作为缓存过期时间的判断依据。ETag 用于标识一次请求。If-Modified-Since 和 If-None-Match 是缓存控制的两种处理标识。下面分别讲述它们的使用。

1）Expires：用于记录 HTTP 缓存过期时间头信息。Expires 简单易用，服务器据此判断

请求是否应该刷新。但是 Expires 的标识比较粗糙，其缺点是当服务器的时区和本地时间与客户端上的不一致时，会影响缓存的准确性。

2）Cache-Control：用于控制 HTTP 缓存属性的头信息。常用的属性包括：

- Public：响应可缓存在任何缓存区；
- Private：响应只能缓存在私有缓存区，不能被共享缓存处理；
- no-cache：响应不能缓存（HTTP/1.0 用 Pragma 的 no-cache 替换）；
- max-age：缓存最大时间（以秒为单位）。和 Expires 共存时，优先使用 max-age。和 Expires 对比，max-age 使用的是相对时间，因此不存在 Expries 的问题。

3）Last-Modified：页面文件最后的修改时间，精确到秒。

4）If-Modified-Since：是指自从某个时间点开始服务器提供了最新数据。当服务器从客户端的请求中读出这个标识后，结合 Cache-Control 中定义的缓存参数，判断是否需要刷新数据来响应当前请求。如图 10-1 所示，客户端请求一个 REST 资源，服务器返回最新数据并指定 Cache-Control 的 max-age 为 1200 秒，Last-Modified 为当前时间。之后客户端再次发起对该资源的请求，服务器根据 Cache-Control 的 max-age 和 Last-Modified 计算是否需要刷新数据。如果需要，即执行业务处理并返回最新的数据，HTTP 状态码为 200 OK；否则直接返回缓存数据，HTTP 状态码为 304 Not Modified。

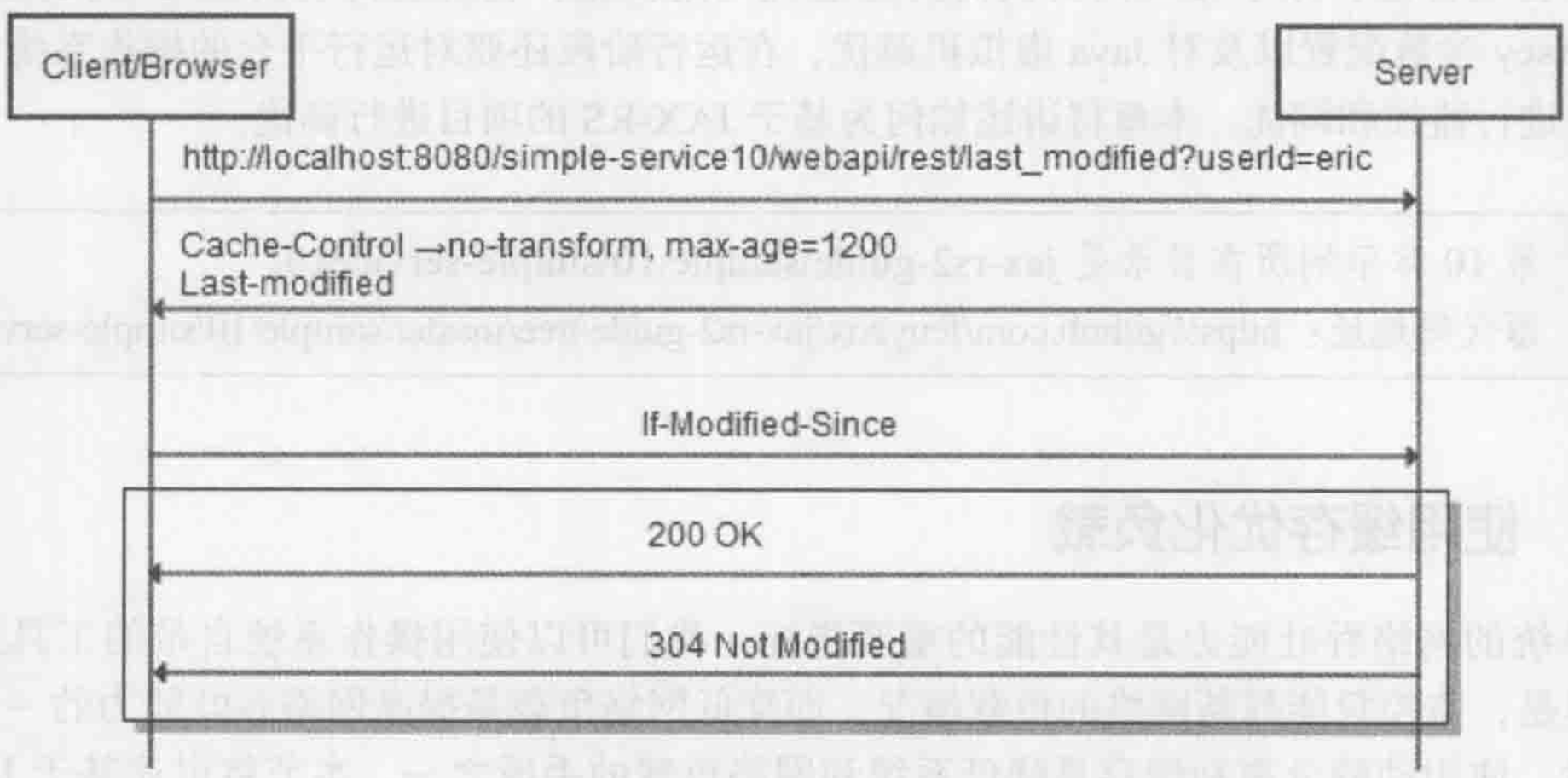


图 10-1 If-Modified-Since 流程示意图

5）ETag：服务器端的散列值，每次请求处理都是唯一的。

6）If-None-Match：是指请求的 ETag 是否匹配。如图 10-2 所示，当服务器从客户端的请求中读出这个标识后，会与当前 ETag 对比，如果符合，即返回缓存数据，HTTP 状态码为 304 Not Modified；否则返回最新数据，HTTP 状态码为 200 OK。

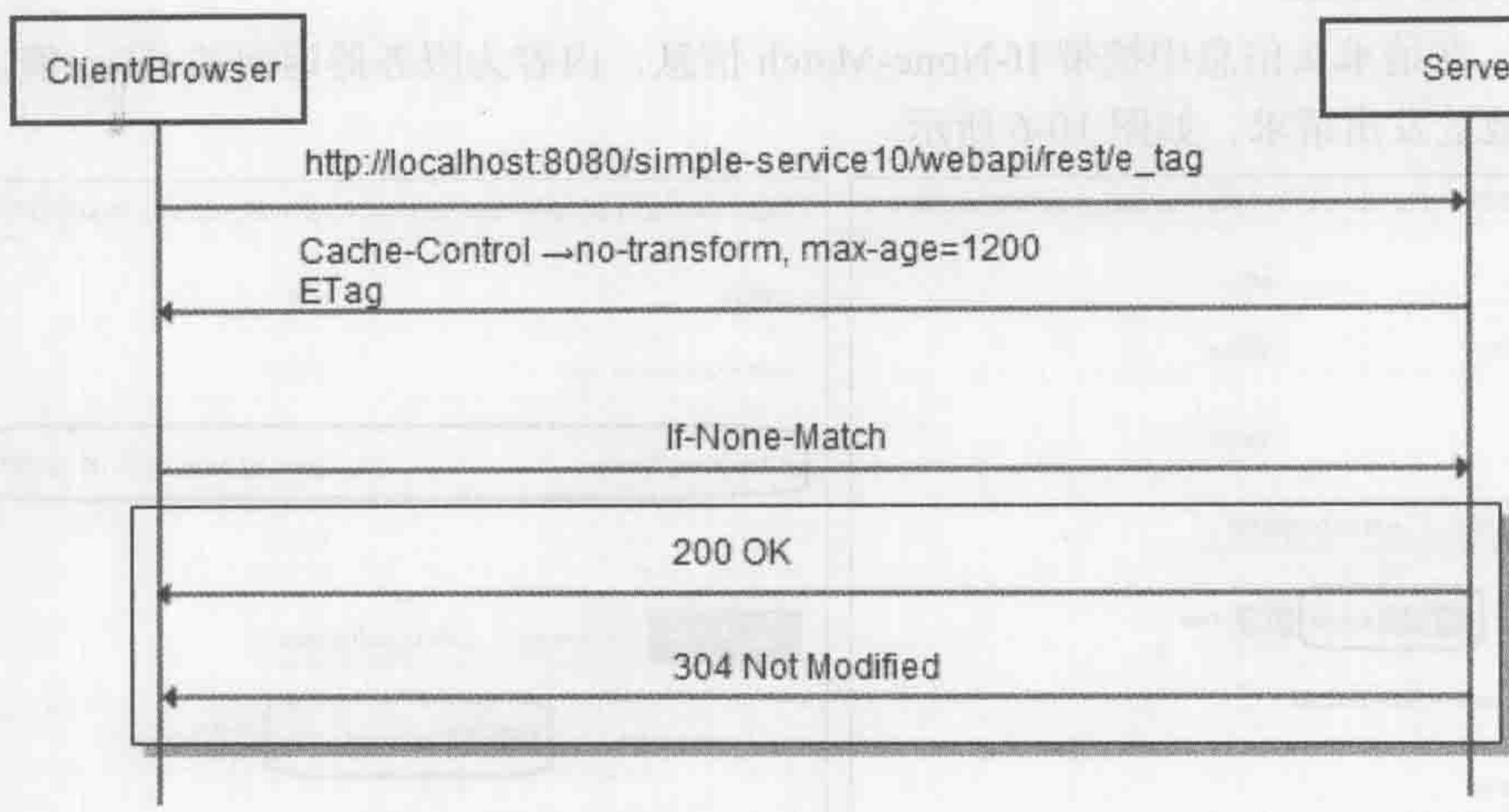


图 10-2 If-None-Match 流程示意图

10.1.2 条件 GET

条件 GET 是使 If-Modified-Since 或者 If-None-Match 头信息对 GET 请求进行缓存处理的一种方案。本节使用本章提供的示例项目演示如何实现对条件 GET 方法的支持。

（1）Last-Modified

使用 Chrome 浏览器的 Postman 插件向支持 If-Modified-Since 条件 GET 请求的 REST 资源发起请求，地址为：http://localhost:8080/simple-service10/webapi/rest/last_modified，如图 10-3 所示。

如图 10-3 所示，返回头信息有 6 个字段，包含 Cache-Control 和 Last-Modified 信息，同时 HTTP 状态码为 200 OK。

（2）If-Modified-Since

接下来，在请求头信息中携带 If-Modified-Since 信息，内容为服务器返回的 Last-Modified 值。立即再次向同一地址发出请求，如图 10-4 所示。

如图 10-4 所示，服务器响应的 HTTP 状态码是 304 Not Modified，因为最后修改时间与服务器当前时间之差没有达到最大缓存的时间 1200 秒，因此直接返回缓存数据。

（3）ETag

使用 Chrome 浏览器的 Postman 插件向支持 If-None-Match 条件 GET 请求的 REST 资源发起请求，地址为：http://localhost:8080/simple-service10/webapi/rest/e_tag，如图 10-5 所示。

如图 10-5 所示，返回头信息有 6 个字段，包含 Cache-Control 和 ETag 信息，同时 HTTP 状态码为 200 OK。

（4）If-None-Match

接下来，在请求头信息中携带 If-None-Match 信息，内容为服务器返回的 ETag 值。立即再次向同一地址发出请求，如图 10-6 所示。

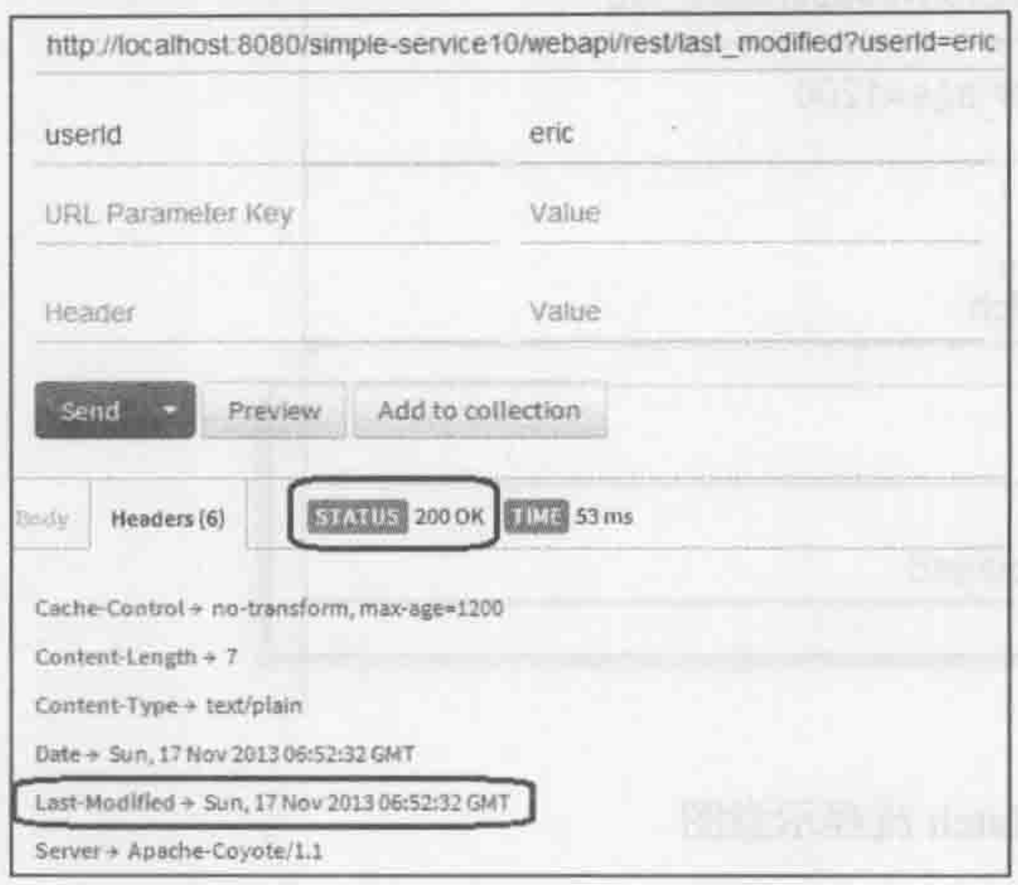


图 10-3　响应头字段 Last-Modified

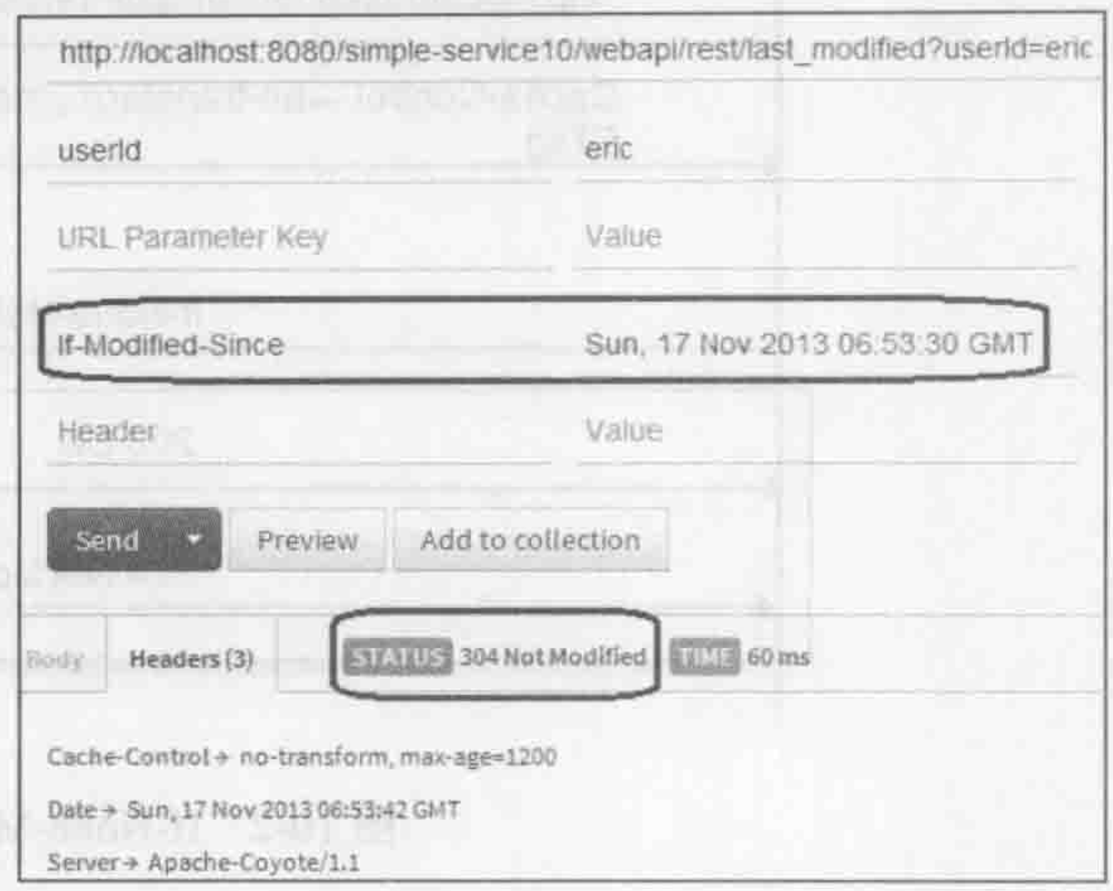


图 10-4　请求头字段 If-Modified-Since

图 10-5　响应头字段 ETag

图 10-6　请求头字段 If-None-Match

如图 10-6 所示，服务器响应的 HTTP 状态码是 304 Not Modified，因为最后修改时间与服务器当前时间之差没有达到最大缓存的时间 1200 秒，因此直接返回缓存数据。

通过上述的实验，我们从浏览器端了解了 HTTP 条件 GET 的原理及其在 REST 服务中的实践。接下来，我们走进服务器端，讲述 REST 缓存的代码实现。

10.1.3　REST 缓存实践

Jersey 提供了对条件 GET 支持的方法，本节将从实践角度进行讲述。

（1）CacheControl

JAX-RS 提供了 CacheControl 类与上述的 HTTP 头信息 Cache-Control 对应。在 REST 请求处理的方法中，可以通过构造 CacheControl 实例，并为其设置参数值，使其作为 HTTP 响应头信息中的 Cache-Control 信息，示例代码如下。

```
CacheControl cacheControl = new CacheControl();
cacheControl.setMaxAge(1200);
cacheControl.setMustRevalidate(true);
```

在这段代码中，Cache-Control 信息包含了 max-age 和 must-revalidate，分别设置为 20 分钟（1200 秒）和强制客户端不缓存数据。

（2）EntityTag

JAX-RS 提供了 EntityTag 类与上述的 HTTP 头信息 ETag 对应。EntityTag 实例用来设置 HTTP 头信息 ETag，示例代码如下。

```
EntityTag tag = new EntityTag(userId.hashCode() + "");
```

在这段代码中，使用当前用户 ID 的 hashcode 作为 ETage 的值。

（3）ResponseBuilder

在完成 HTTP 头信息的设置后，可以填充给 Response 实例，作为响应头信息，示例代码如下。

```
Response.ok(getIt()).cacheControl(cacheControl).build();
Response.ok(getIt()).cacheControl(cacheControl).lastModified(date).build();
Response.ok(getIt()).cacheControl(cacheControl).tag(tag).build();
```

在这段代码中，静态方法 Response.ok() 的返回类型是 ResponseBuilder，通过使用 ResponseBuilder 可以为 Response 实例填充 HTTP 头信息。本例示范了不同 HTTP 头信息的填充，分别是 Cache-Control 信息、Last-Modified 信息和 ETag 信息。

（4）evaluatePreconditions() 方法

Jersey 对 If-Modified-Since 和 If-None-Match 的支持是通过 Request 接口的 evaluate-Preconditions() 方法实现的。

If-Modified-Since 的实现是传入 Date 类型的实例，代表最后修改时间，输出是 ResponseBuilder 实例，如果不为空，即代表缓存尚未失效，否则重新获取数据并构造 Cache-Control 和 lastModified 响应头信息，示例代码如下。

```
Date lastModified = map.get(userId);
Response.ResponseBuilder rb = null;
if (lastModified != null) {
  rb = request.evaluatePreconditions(lastModified);
}
if (rb != null) {
  return rb.cacheControl(cacheControl).build();
} else {
```

```
    Date date = new Date();
    map.put(userId, date);
    return Response.ok(getIt()).cacheControl(cacheControl).lastModified(date).build();
}
```

If-None-Match 的实现是传入 EntityTag 实例，代表当前 ETag 值，输出是 ResponseBuilder 实例，如果不为空，即代表缓存尚未失效，否则重新获取数据并构造 Cache-Control 和 ETag 响应头信息，示例代码如下。

```
EntityTag tag = new EntityTag(userId.hashCode() + "");
rb = request.evaluatePreconditions(tag);
if (rb != null) {
  return rb.cacheControl(cacheControl).build();
} else {
  return Response.ok(getIt()).cacheControl(cacheControl).tag(tag).build();
}
```

10.1.4 ab 测试

ab 是 Apache 的 HTTP 服务器 benchmark 工具，可以对条件 GET 进行吞吐量测试。使用 ab 工具模拟 100 个客户端进行 1000 次的请求来测试资源地址 http://192.168.0.163:8080/simple-service10/webapi/rest，测试报告如下所示。

```
//ab - Apache HTTP server benchmarking tool
//ab  [ -c concurrency ] [ -H custom-header ] [ -n requests ] [http[s]://]hostname[:port]/path
ab -n1000 -c100 http://192.168.0.163:8080/simple-service10/webapi/rest
Requests per second:    131.13 [#/sec] (mean)
Time per request:       762.574 [ms] (mean)
ab -n1000 -c100 -H 'If-Modifed-Since:'
http://192.168.0.163:8080/simple-service10/webapi/rest/last_modified?userId=eric
Requests per second:    158.23 [#/sec] (mean)
Time per request:       632.006 [ms] (mean)
```

如上述测试报告所示，-n 参数表示请求数，-c 参数表示并发客户端数。当没有使用缓存策略时，100 个客户端进行 1000 次请求得到的每秒请求次数大约是 131 次，每个请求的处理时间是 762.574ms。同样，测试参数对支持条件 GET 的资源地址进行测试，得到的每秒请求次数大约是 158 次，每个请求的处理时间是 632.006ms。

10.2 使用版本号优化服务

在 REST 服务的资源地址中加入版本号，可以在 REST 服务接口升级后方便地实现对旧接口的长期支持。但是否应该在 REST 资源中使用版本号，REST 领域的声音并不一致。本节站在使用版本号为开发和运维人员带来效率提升的角度，讲述 REST 服务中的版本号。

10.2.1 何时使用版本号

REST 的版本号（version）并不是必须存在的，我们从 B/S 结构的网站和应用软件两个应用场景讨论在 REST 服务中何时该使用版本号信息。

1. 网站

网站的特点是“永远处于 Beta 版”，发版是开发团队和运维团队的日常工作。因此，确保与外系统之间的 Web 服务始终有效是网站灰度测试不可或缺的环节。如果我们的 Web 服务升级，那么必须在上线前确保外系统的接入没有问题。

最直接的办法就是提供带有版本号的地址，在上线新版本时通知对方修改其 Web 服务接入地址。这个场景下，外系统如果接入旧版本地址，可以继续使用旧版本功能，省去了因升级导致的不兼容问题。此后，外系统的开发人员可以视情况将其接口方法升级到新版本，然后接入新版地址。

如果对方由于某种原因无法修改接入地址，我们的系统应该保留旧的资源方法并提供兼容性解决方案。例如，通过在访问者身份信息和请求头信息中额外定义的版本信息等方式，相应地为其提供新版服务。

其实，在 REST 式的 Web 服务中使用版本号并不是必要的。如果必须使用，笔者推荐将最新版本的资源地址和无版本号的资源地址一并使用，这样做的好处是，如果接入端不使用带有版本号的地址，那么其接入的永远是最新版本的资源地址，而无须因为服务端的频繁升级带来无谓的修改。

2. 应用软件

如果我们开发的 REST 式的 Web 服务是企业管理平台一类的软件，而且出现带版本号的资源地址，那么该版本号应该和软件版本匹配，退一步讲，即使从来都没有被修改过，version 信息也应该相应地升级。其实，在软件中使用版本号的必要性很低，维护版本的价值本身就不大。笔者经历过一个对版本号依赖很强的产品，该产品历经多年开发，有多个小版本，其中多数是为支持不同的客户开发的，小版本之间缺乏共性，因此在 Web 服务中定义了版本号。在这个例子中，版本号的确有存在的必要，但排除历史原因，因此一个优秀的设计是可以减少日后不必要的维护成本的。

10.2.2 如何使用版本号

纵然笔者以浅薄的认识站在尽量不去使用版本号的行列，但当需要使用版本号的需求摆在面前时，头等大事还是考虑如何在 REST 服务中实现版本号。

1. 资源定义

REST 服务对版本号信息的支持有两种方式，即通过资源地址（URL）或者 HTTP 头信

息来实现，下面分别讲述这两种实现。

（1）URL

通过资源地址实现的方式实现 REST 的版本信息，其 URL 形式如下所示。

```
http://localhost:8080/simple-service10/webapi/rest/v1.0
http://localhost:8080/simple-service10/webapi/rest/v2.0
```

在服务的 URL 中，版本号信息作为资源路径的一部分加入到 context 与资源名称之间。上面定义了 1.0 版本和 2.0 版本。相应的资源方法的实现并不复杂，只需额外增加一个 @Path 注解，示例代码如下。

```
@GET
@Path("v2.0")
@Produces(MediaType.TEXT_PLAIN)
public String getIt2() {
...
```

对应的测试代码分别测试了两个版本的资源地址的可用性，示例代码如下。

```
@Test
public void testVersion() {
  Response response =target("rest").path("v1.0").request().get();
  Link link =response.getLink("currentVersion") ;
  System.out.println(link);

  Response response2 =target("rest").path("v2.0").request().get();
  String result =response2.readEntity(String.class);
  System.out.println(result);
}
```

（2）HEAD

通过头信息支持版本号需要在 HTTP 头信息（HEAD）中指定一个特殊的属性，本示例使用 X-API-Version 作为版本信息的标识，示例代码如下。

```
@GET
@Path("head-version")
@Produces(MediaType.TEXT_PLAIN)
public String getIt3(@Context final HttpHeaders headers) {
  String version = headers.getRequestHeaders().get("X-API-Version").get(0);
  if (version.equals("2"))
      return getIt2();
...
```

在这段代码中，资源方法 getIt3() 首先从请求头信息中获取 X-API-Version 属性信息，该属性值代表请求的 REST 版本信息。

对应的测试代码通过定义请求头信息中的 X-API-Version 值对不同版本的资源地址进行访问，示例代码如下。

```
@Test
public void testHeadVersion() {
  Response response =target("rest").path("head-version").request()
.header("X-API-Version","2").get();
  String result =response.readEntity(String.class);
  System.out.println(result);
}
1 > GET http://localhost:9998/rest/head-version
1  > X-API-Version: 2
```

2. 过期定义

一个 REST 服务经历长期的开发后，也许需要对早期的 Web 服务资源地址进行去化（即逐渐淘汰）。在正式废弃旧版本的资源地址之前，应该提供相当一段时间，比如间隔几个版本的兼容性支持的时间，并在 API 文档中给出最后支持时间，以便外系统或客户端有充分的升级准备时间。一种友好的通知版本变更的方式是通过响应头 Link，将新版本的资源地址响应给客户端，示例代码如下。

```
@GET
@Path ("v1.0")
@Produces (MediaType.TEXT_PLAIN)
public Response getIt1 (@Context UriInfo uriInfo) {
  final UriBuilder ub = uriInfo.getAbsolutePathBuilder();
  final URI uri = ub.replacePath ("rest/v2.0") .build();
  return Response.accepted().link (uri, "currentVersion") .build();
}
```

在这段代码中，响应信息提供了 link 信息，其 rel 定义为 currentVersion，代表该资源的当前版本，uri 的值是当前版本的资源地址。

10.3 使用参数配置优化服务

Jersey 提供了丰富的配置参数，用于为 REST 服务特定的需求提供定制。本节将分 3 部分讲述 Jersey 的配置：通用部分、服务器端、客户端。

10.3.1 通用配置

Jersey 提供的通用配置类 org.glassfish.jersey.CommonProperties 位于 jersey-common 包内。在应用中通过修改通用配置的值，可以改变 Jersey 服务器端和客户端两侧的默认行为。例如，禁用不必要的自动探测功能，以提高系统的整体性能。表 10-1 提供了 Jersey 通用配置参数参考。

表 10-1 Jersey 通用配置参数列表

常量名称及参数名称	说明
FEATURE_AUTO_DISCOVERY_DISABLE jersey.config.disableAutoDiscovery	全局禁用自动探测特征。默认值 =false
JSON_PROCESSING_FEATURE_DISABLE jersey.config.disableJsonProcessing	禁用对 Json Processing（JSR-353）特征的支持。默认值 =false
METAINF_SERVICES_LOOKUP_DISABLE jersey.config.disableMetainfServicesLookup	禁用自动加载 META-INF/services 目录下的 SPI。默认值 =false
MOXY_JSON_FEATURE_DISABLE jersey.config.disableMoxyJson	禁用对 MOXyJson 特征的支持。默认值 =false
OUTBOUND_CONTENT_LENGTH_BUFFER jersey.config.contentLength.buffer	设置响应实体缓存的大小和响应头 Content-Length 的值。默认值 =8192

在 REST 应用中修改通用配置类 org.glassfish.jersey.CommonProperties 的值非常简单，示例代码如下。

```
public class AirApplication extends ResourceConfig {
    public AirApplication() {
        property(CommonProperties.FEATURE_AUTO_DISCOVERY_DISABLE,true);
        property(CommonProperties.JSON_PROCESSING_FEATURE_DISABLE,true);
        property(CommonProperties.METAINF_SERVICES_LOOKUP_DISABLE,true);
        property(CommonProperties.MOXY_JSON_FEATURE_DISABLE,true);
        property(CommonProperties.OUTBOUND_CONTENT_LENGTH_BUFFER,20480);
        packages("com.example.resource");
    }
}
```

在这段代码中，分别配置了禁用自动探测 feature、支持 JSR-353、自动加载 SPI 和支持 MOXy 的 JSON，并配置了响应实体缓存的大小为 20KB。

10.3.2 服务器端配置

Jersey 提供的服务器端配置类 org.glassfish.jersey.server.ServerProperties 位于 jersey-server 包内。在应用中通过修改通用配置的值，可以改变 Jersey 服务器端的默认行为。如禁用不必要的自动探测功能，可以提高系统的整体性能。Jersey 服务器端配置参数见表 10-2。

表 10-2 Jersey 服务器端配置参数列表

常量名称及参数名称	说明
APPLICATION_NAMEjersey.config.server.application.name	应用名称，用于在 JMX 监控中的唯一标识。无默认值
BV_FEATURE_DISABLEjersey.config.beanValidation.disable.server	禁用对 Bean Validation 的支持。默认值 =false
BV_DISABLE_VALIDATE_ON_EXECUTABLE_OVERRIDE_CHECKjersey.config.beanValidation .disable.validateOnExecutableCheck.server	禁用对注解 @ValidateOnExecution 的校验。默认值 =false
BV_SEND_ERROR_IN_RESPONSEjersey.config.beanValidation .enableOutputValidationErrorEntity.server	启用发送校验错误信息到客户端。默认值 =false

（续）

常量名称及参数名称	说明
FEATURE_AUTO_DISCOVERY_DISABLEjersey.config.disableAutoDiscovery.server	禁用服务器端对 feature 的自动探测。默认值=false
HTTP_METHOD_OVERRIDEjersey.config.server.httpMethodOverride	定义 HTTP 方法覆盖用于过滤器类 HttpMethodOverrideFilter
JSON_PROCESSING_FEATURE_DISABLE jersey.config.disableJsonProcessing.server	禁用服务器端对 Json Processing（JSR-353） feature 的支持。默认值 =false
LANGUAGE_MAPPINGSjersey.config.server.languageMappings	定义 URI 扩展语言匹配。用于过滤器类 UriConnegFilter
MEDIA_TYPE_MAPPINGSjersey.config.server.mediaTypeMappings	定义 URI 扩展媒体匹配。用于过滤器类 UriConnegFilter
METAINF_SERVICES_LOOKUP_DISABLEjersey.config.disableMetainfServicesLookup.server	禁用服务器端对 META-INF/services 下 SPI 的自动加载。默认值 =false
MOXY_JSON_FEATURE_DISABLEjersey.config.disableMoxyJson.server	禁用服务器端对 MOXyJson feature 的支持。默认值 =false
MONITORING_STATISTICS_ENABLEDjersey.config.server .monitoring.statistics.enabled	启用监控统计。默认值 =false
MONITORING_STATISTICS_MBEANS_ENABLED jersey.config.server.monitoring.statistics.mbeans.enabled	启用 JMX Mbeans 监控统计。默认值 =false
OUTBOUND_CONTENT_LENGTH_BUFFER（Jersey 2.2+）jersey.config.contentLength.buffer.server	设置响应实体缓存的大小和响应头 Content-Length 的值。默认值 =8192
PROVIDER_CLASSNAMESjersey.config.server.provider.classnames	指定初始的 Resource 和 Provider 类名
PROVIDER_CLASSPATHjersey.config.server.provider.classpath	指定初始的 Resource 和 Provider 类路径
PROVIDER_PACKAGESjersey.config.server.provider.packages	指定初始的 Resource 和 Provider 类所在的包名
PROVIDER_SCANNING_RECURSIVEjersey.config.server .provider.scanning.recursive	设置包扫描递归策略。默认值 =true
RESOURCE_VALIDATION_DISABLEjersey.config.server .resource.validation.disable	禁用 Resource 验证。默认值 =false
RESOURCE_VALIDATION_IGNORE_ERRORSjersey.config.server.resource.validation.ignoreErrors	启用忽略资源模型校验错误。默认值 =false
WADL_FEATURE_DISABLEjersey.config.server.wadl.disableWadl	禁用 WADL 生成。默认值 =false
WADL_GENERATOR_CONFIGjersey.config.server.wadl.generatorConfig	定义一个生成 WADL 的 provides，WadlGenerator 类的实例

10.3.3 客户端配置

Jersey 提供的客户端配置类 org.glassfish.jersey.client.ClientProperties 位于 jersey-client 包

内。在应用中通过修改通用配置的值，可以改变 Jersey 客户端的默认行为。如禁用不必要的自动探测功能，可以提高系统的整体性能。Jersey 客户端配置参数参见表 10-3。

表 10-3 Jersey 客户端配置参数列表

常量名称及参数名称	说明
ASYNC_THREADPOOL_SIZEjersey.config.client.async.threadPoolSize	设定 Grizzly Connector 中异步线程池大小。默认不设置
CHUNKED_ENCODING_SIZEjersey.config.client.chunkedEncodingSize	Chunked encoding 大小。默认不设置
CONNECT_TIMEOUTjersey.config.client.connectTimeout	连接超时，单位毫秒（ms）。默认值 =0，永不超时
FEATURE_AUTO_DISCOVERY_DISABLEjersey.config.disableAutoDiscovery.client	禁用客户端自动探测特征。默认值 =false
FOLLOW_REDIRECTSjersey.config.client.followRedirects	定义客户端自动重定向到 3xx 指定的 URI。默认值 =true
HTTP_URL_CONNECTION_SET_METHOD_WORKAROUNDjersey.config.client.httpUrlConnection.setMethodWorkaround	定义客户端设置不支持的 HTTP 方法到 HttpURLConnection。默认值 =false
JSON_PROCESSING_FEATURE_DISABLEjersey.config.disableJsonProcessing.client	禁用对 Json Processing（JSR-353）特征的支持。默认值 =false
METAINF_SERVICES_LOOKUP_DISABLEjersey.config.disableMetainfServicesLookup.client	禁用客户端对 META-INF/services 下 SPI 的自动加载。默认值 =false
MOXY_JSON_FEATURE_DISABLEjersey.config.disableMoxyJson.client	禁用客户端对 MOXyJson 特征的支持。默认值 =false
OUTBOUND_CONTENT_LENGTH_BUFFER（Jersey 2.2+）jersey.config.contentLength.buffer.client	设置响应实体缓存的大小和响应头 Content-Length 的值。默认值 =8192
READ_TIMEOUTjersey.config.client.readTimeout	读超时，单位毫秒（ms）。默认值 =0，永不超时
SUPPRESS_HTTP_COMPLIANCE_VALIDATION（Jersey 2.2+）jersey.config.client.suppressHttpComplianceValidation	启用忽略客户端请求兼容性校验错误。默认值 =false
USE_ENCODINGjersey.config.client.useEncoding	指定 Content-Encoding 的值，指定值类型为 EncodingFilter。默认不指定

10.4 Java 虚拟机调优

基于 JAX-RS 的项目，通常要关注服务器和开发环境的 Java 虚拟机配置。日常的关注点通常是内存溢出和内存泄漏。为了更好地解决 Java 虚拟机这个不是项目代码问题的问题，这里将从简述虚拟机开始。

10.4.1 虚拟机概述

在虚拟机加载类之前，Java 类的源代码首先被构建工具（如 Maven 等）通过调用编译器 javac 编译成 .class 字节码文件。虚拟机启动时，使用 ClassLoader 以双亲委派机制装载类

的字节码文件，链接到内存，然后对类进行初始化（或者滞后初始化）。此时，Java 类就进入了虚拟机的内存，我们的关注点转到虚拟机的内存模型。虚拟机内存模型示意图如图 10-7 所示。

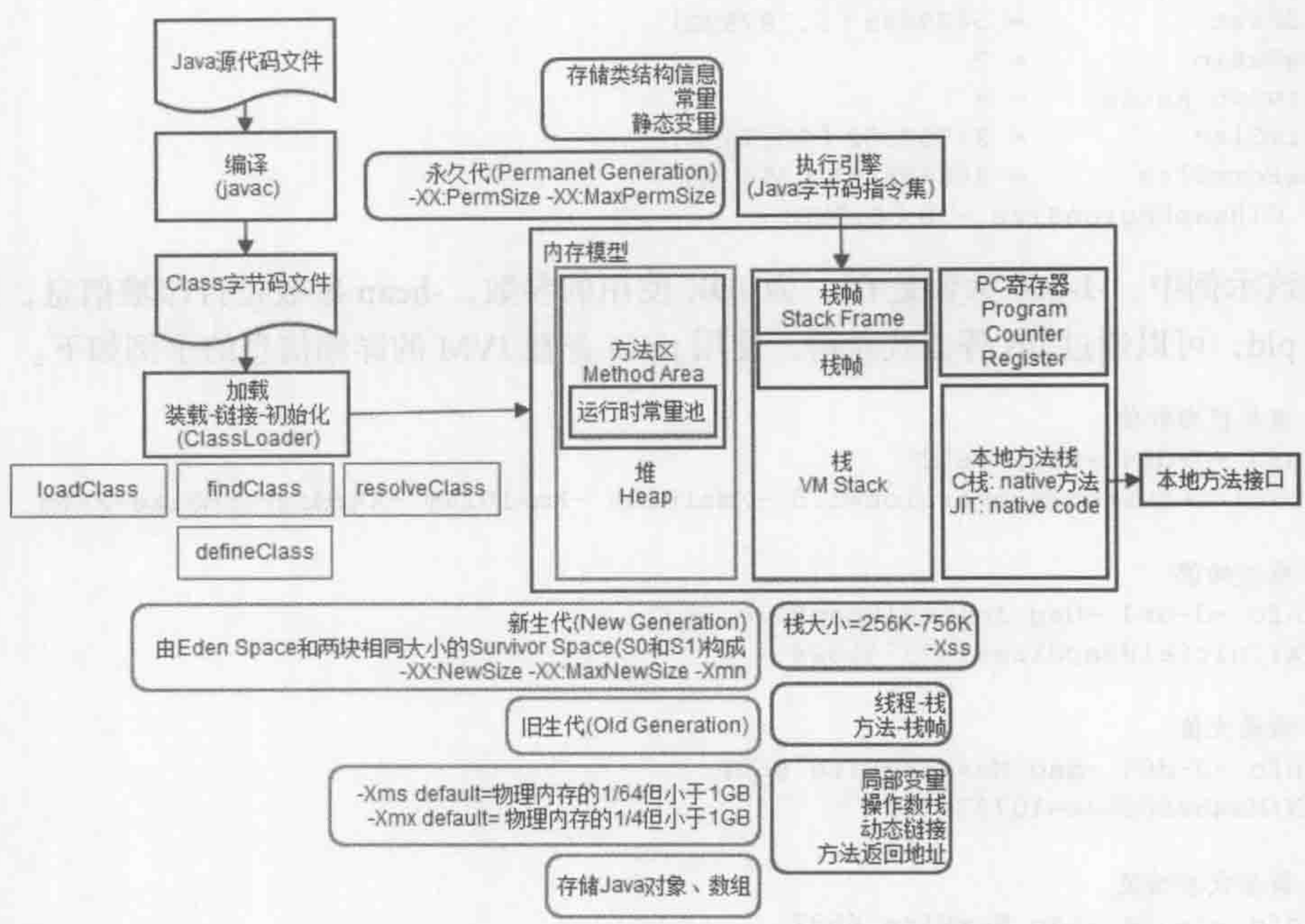


图 10-7 Java 虚拟机示意图

如图 10-7 所示，Java 虚拟机大体上可分为堆和栈两部分。其中，堆用于存储对象和数组，堆内的资源是线程共享的。堆细分为新生代、旧生代和永久代，SUN 的虚拟机将永久代和方法区等同（永久代在 JDK 8 中被驱除出 JVM）。方法区内还包括常量池和运行时常量池。方法区用于存储类信息以及常量和静态变量等信息。垃圾回收器对堆和方法区的回收频率是不同的，方法区内的资源通常不会被频繁回收。非严格来说，虚拟机栈、本地方法栈以及 PC 寄存器都可以视为栈。虚拟机栈与运行期的线程对应，每启动一个线程就会在寄存器中注册一个指针，指向内存中为其开辟的虚拟机栈。虚拟机栈内会为线程的方法开辟栈帧，栈帧用于存储局部变量、操作数栈、方法返回值等信息。本地方法栈用于调用操作系统相关的本地方法接口，本地方法是指使用 native 关键字修饰、使用 C 语言实现的方法，因此本地方法栈又称 C 栈。本地方法栈同时存储 JIT 编译的方法，这类方法由于在虚拟机栈中被频繁调用到一定次数，因此被 JIT 编译成本地方法，以提高性能。

可以使用 JDK 自带的工具 jmap 和 jinfo 来查看项目的虚拟机配置，示例代码如下。

```
jmap -J-d64 -heap 6632

Heap Configuration:
```

```
MinHeapFreeRatio = 40
MaxHeapFreeRatio = 70
MaxHeapSize      = 1073741824 (1024.0MB)
NewSize          = 1310720 (1.25MB)
MaxNewSize       = 17592186044415 MB
OldSize          = 5439488 (5.1875MB)
NewRatio         = 2
SurvivorRatio    = 8
PermSize         = 21757952 (20.75MB)
MaxPermSize      = 268435456 (256.0MB)
G1HeapRegionSize = 0 (0.0MB)
```

在该示例中，-J-d64 参数是 64，为 JDK 使用的参数，-heap 参数是打印堆信息，最后的参数是 pid，可以通过 jps 等方式获得。使用 jinfo 查看 JVM 的详细信息的示例如下。

```
// 虚拟机参数值
jinfo -J-d64 -flags 6632
-Dosgi.requiredJavaVersion=1.5 -Xms1024m -Xmx1024m -XX:MaxPermSize=256m

// 堆初始值
jinfo -J-d64 -flag InitialHeapSize 6632
-XX:InitialHeapSize=1073741824

// 堆最大值
jinfo -J-d64 -flag MaxHeapSize 6632
-XX:MaxHeapSize=1073741824

// 新生代初始值
jinfo -J-d64 -flag NewSize 6632
-XX:NewSize=1310720

// 新生代最大值
jinfo -J-d64 -flag MaxNewSize 6632
-XX:MaxNewSize=18446744073709486080

// 永久代初始值
jinfo -J-d64 -flag PermSize 6632
-XX:PermSize=21757952

// 永久代最大值
jinfo -J-d64 -flag MaxPermSize 6632
-XX:MaxPermSize=268435456
```

在该示例中，“-flags”打印显式定义的虚拟机参数值列表，“-flag 参数名”打印某项参数值。

在日常开发和调试中遇到的栈溢出时，可以调整虚拟机栈参数 -Xss，但如果虚拟机栈的大小不是根本问题，比如递归方法中不适当的编码产生死循环而导致的栈溢出，那么还需要监控和跟踪代码来发现和解决问题。

对于处理堆的异常或者错误，通常可归结为内存溢出和内存泄漏这两个方面，浅析如下。

10.4.2 内存溢出与内存泄漏

内存溢出和内存泄漏的表象都是系统提供的内存空间不足以支撑应用的运行，但两者的实质是不同的。前者可以理解为进入虚拟机的资源总量比出去的资源总量多、占用内存的速率要快于垃圾回收的速率，这有可能是正常的业务逻辑，也有可能是设计上的缺陷；后者可以理解为进入虚拟机的某些资源存在引用并且无法被垃圾回收，虚拟机失去了对这部分资源的回收能力。简述如下。

1. 内存溢出

内存溢出的原因是堆的大小无法承载内部的对象和数组了。这个问题简单地说，可以从两个方面解决。一是在代码中减少不必要的实例构造，这个途径会直接提高项目的性能。二是调整堆大小，-Xms 是设置堆的最小空间的参数，-Xmx 是设置堆的最大空间的参数。对于新生代，空间的设置参数为 -XX:NewSize 和 -XX:MaxNewSize，当 -Xms 和 -Xmx 的值相同时，可以使用参数 -Xmn 来简化。对于永生代（SUN 虚拟机对应的是方法区）的设置，初始空间参数为 -XX:PermSize，最大空间参数为 -XX:MaxPermSize。注意，虚拟机一旦启动，堆的总大小就被固定，无法动态调整。

2. 内存泄漏

内存泄漏的原因是垃圾回收器无法对存在无效引用的对象进行回收导致的内存溢出。内存泄漏没有办法通过设置虚拟机参数来解决，因为导致这一问题的原因来自编码阶段，比如资源在使用完毕后没有被释放等。解决这一问题的办法是改进代码或者变更依赖包（项目的依赖包并不总是让人放心的），发现问题的途径不一而足，通常是通过分析工具来排查泄漏点。常用的分析工具见表 10-4。

表 10-4 Java 分析工具列表

工具名称	获取路径
jvisualvm（Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool）	JAVA_HOME/bin
JMC（Oracle Java Mission Control）	JAVA_HOME/bin（JDK 7u40+）
MAT（Memory Analyzer）	http://www.eclipse.org/mat
JProfiler	http://www.ej-technologies.com/products/jprofiler/overview.html
YourKit Profiler	http://www.yourkit.com

分析过程包括实时监控和 dump 快照两种，前一种通过工具本身提供的 GUI 界面即可实现，后一种可以通过 JDK 自带的工具 jmap 来 dump 当前虚拟机快照，然后将快照文件（本例是 Ubuntu 系统下的 /var/local/my.mem.bin）导入分析工具，排查内存泄漏点。

在 Server 上执行：

```
jmap -dump:format=b,file=mem.bin<pid>
```

在做分析的机器执行：

```
sudo scp root@server:/home/erichan/mem.bin /var/local/my.mem.bin
```

另外，也可以通过 psi-probe 这样的实时工具对运行期的服务器环境进行监控。下面给出 psi-probe 的安装示例。

1）下载 ZIP 格式的 psi-probe 工具包。

```
// 下载
sudowget https://psi-probe.googlecode.com/files/probe-2.3.3.zip -P /opt
```

2）将其中的 war 包部署到 Tomcat 的应用路径下。

```
// 安装
sudo unzip /opt/probe-2.3.3.zip -d /opt/
cp /opt/probe.war /opt/apache-tomcat-7.0.42/webapps/
```

3）启动 Tomcat，然后访问该工具的 URL，这里的 context 是 probe。

```
// 测试
http://localhost:8080/probe/logs/index.htm
```

由于篇幅所限，因此虚拟机的部分到此结束，读者可以找来专门讲述 Java 虚拟机的书来学习。

10.5 本章小结

本章讲述了使用 Java 进行 REST 开发中需要注意的性能和优化。10.1 节讲述了 HTTP 通信过程中的缓存机制和实践，10.2 节讲述了何时使用以及如何优化 REST API 的版本信息，10.3 节讲述了 Jersey 提供的参数配置，10.4 节简单介绍了 Java 虚拟机的情况、如何优化其配置以及基本的 JVM 问题解决办法。下一章，我们开始讲述实现 JAX-RS 的综合示例，即基于 REST 式的 Web 服务项目：统一自动化测试平台。

第三篇 *Part 3*

实践分享——JAX-RS 2.0 综合

■ 第 11 章　统一自动化测试平台

Chapter 11 第 11 章

统一自动化测试平台

本书的最后一章是一个综合性的 REST 式 Web 服务实例，讲述如何使用 Jersey 开发一个基于 JAX-RS 2.0 的 REST 式的 Web 服务。同时，在这个过程中，将全面展示前面 10 章所述的知识点，并与读者分享敏捷开发、持续集成和持续部署等项目管理实践。我们要开发的 Web 服务是一个统一自动化测试平台，命名为 ATUP（Automation Test Unified Platform），从无到有，为读者呈现一个完整的应用。

阅读指南

ATUP 源代码地址：https://github.com/feuyeux/jax-rs2-atup

11.1 ATUP 的定义

我们的示例分为 3 个部分：需求的定义、设计和交付（编码、测试、部署）。首先我们从需求开始。

统一自动化测试平台需满足如下特征：

- ATUP 应确保测试项目和测试用例的无关性和统一性，对不同的测试平台、测试系统和测试语言应使用统一的输入和输出。
- ATUP 应支持测试的类型包括自动化测试和手动测试。测试的输入和输出方式统一，手动应和自动化拥有一致的行为。
- ATUP 对测试设备的管理应做到使测试设备处于满负荷工作，同时对测试设备状态进行监控，并提供告警机制。
- ATUP 平台应支持对测试者的身份管理和权限控制。

11.1.1　需求仓库

基于上述定义，我们根据 Scrum 项目管理方式，首先收集用户的需求，着手建立 Backlog（需求仓库）。为了演示敏捷开发生命周期，我们将 ATUP 的实现在两个迭代周期内完成，如图 11-1 所示。

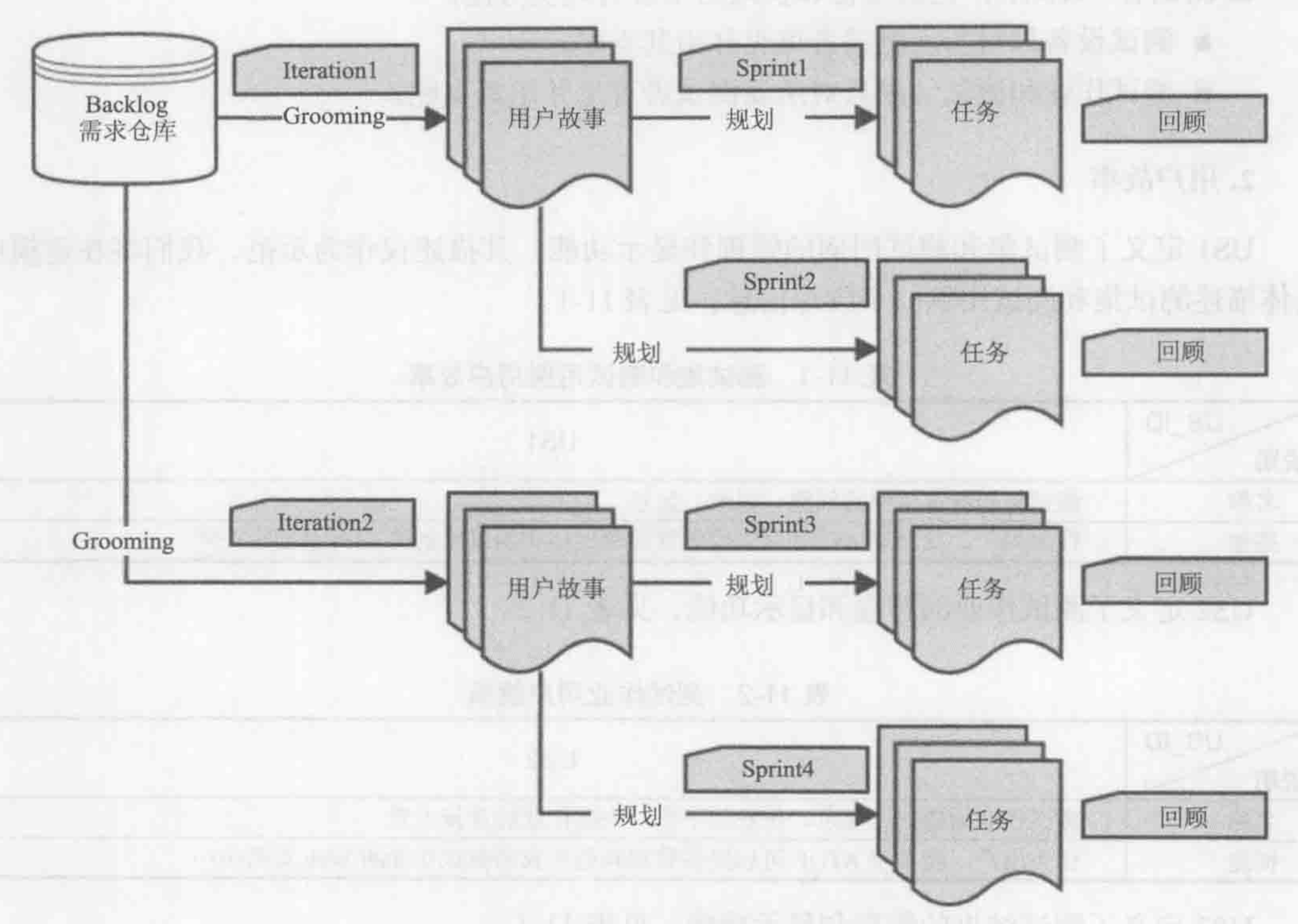


图 11-1　Scrum 开发管理流程

如图 11-1 所示，Backlog（需求仓库）是实现 ATUP 的源头，每个迭代（Iteration）开始的步骤是开 Grooming 会议，会议中会选择仓库中优先级高的需求并定义相关的用户故事（User Story）。

用户故事是用户需求的单位，用以标识和描述用户需求。在定义用户故事的过程中，产品负责人（Product Owner，PO）作为开发团队的代表和用户一起明确需求的内容、可行性等信息。在明确用户故事后，PO 作为用户的代表和开发团队一起在 Grooming 阶段明确需求信息，并定义用户故事的验收标准和优先级别等信息。

下面，我们模拟设定 ATUP 项目的 Backlog 和 User Story。

1. Backlog 列表

ATUP 项目的需求仓库包含如下条目。

- 测试用例（Test Case）：它是测试项目的基本单位。测试集（Test Suite）是测试用例的集合，一个测试集中的测试用例是一个统一的整体。测试者可以自由组合测试用例到测试集。

- 测试作业（Test Job）：它是运行时的测试用例，测试设备总是先执行高优先级的作业。
- 测试结果（Test Result）：它是完成时的测试用例，用来记录测试作业的执行情况。
- 测试设备（Test Device）：它是运行测试的基本单位，测试平台监控设备的状态和在其上的测试用例的分配。
- 测试者（Tester）：它的身份和权限由平台管理员分配。
  - 测试设备只对所属测试者可见并由其支配。
  - 测试作业和测试结果只对所属测试者可见并由其支配。

2. 用户故事

US1 定义了测试集和测试用例的管理和显示功能。其描述仅作为示范，我们将在建模中具体描述测试集和测试用例的字段等信息，见表 11-1。

表 11-1 测试集和测试用例用户故事

US_ID / 说明	US1
名称	测试集和测试用例的创建、修改、显示
描述	作为用户，我希望 ATUP 可以提供管理测试集和测试用例的 Web 页面……[⊖]

US2 定义了测试作业的管理和显示功能，见表 11-2。

表 11-2 测试作业用户故事

US_ID / 说明	US2
名称	测试作业的建立、显示，作业的下发，失败作业的重新下发
描述	作为用户，我希望 ATUP 可以提供管理和监控我的测试作业的 Web 页面……

US3 定义了测试结果的管理和显示功能，见表 11-3。

表 11-3 测试结果用户故事

US_ID / 说明	US3
名称	测试结果的回写、显示
描述	作为用户，我希望 ATUP 可以提供我的测试结果显示和统计的 Web 页面……

US4 定义了测试设备的管理和显示功能，见表 11-4。

表 11-4 测试设备用户故事

US_ID / 说明	US4
名称	测试设备的建立、显示、管理
描述	作为用户，我希望 ATUP 可以提供管理和监控测试设备的 Web 页面……

⊖ “……”部分请读者自行补充完整。

US5 定义了用户的管理和显示功能，见表 11-5。

表 11-5　平台用户管理用户故事

说明 \ US_ID	US5
名称	测试者的创建、修改、显示
描述	作为平台管理员，我希望 ATUP 可以提供管理 ATUP 用户的 Web 页面……

11.1.2　需求分析

对于我们熟知的瀑布模型中的需求分析，Scrum 框架的 Grooming 阶段与之最为接近。在这个阶段，PO 会代表客户或者同客户一起和开发团队讨论即将到来的迭代中要完成的用户故事。用户故事的优先级包括最高到最低的若干级别，通常低于中级的用户故事不会进入 Grooming 阶段，因为每个迭代都很短小，无法在容纳高优先级用户故事的同时，做低优先级的事情。

举例来说，在 Grooming 阶段，US1 被定义为最高优先级，因为测试集和测试用例的管理和显示功能是实现测试平台的基础功能。US1 经过详细讨论后，定义了 4 个验收标准，见表 11-6。

表 11-6　测试集和测试用例用户故事分析

说明 \ US_ID	US1
名称	测试集和测试用例的创建、修改、显示
验收标准	实现测试集的显示功能 实现测试集的增加、删除、修改、查询功能，测试集非空时不能删除 实现测试用例显示功能 实现测试用例的增加、删除、修改、查询功能
优先级别	最高

US2 的优先级也是最高的，因为测试作业是动态的测试用例，是从下发到完成测试的中间状态，所以在测试平台的功能开发中非常紧急而重要。US2 定义了三个验收标准分别用于验收测试作业的定义、监控和容错，见表 11-7。

表 11-7　测试作业用户故事分析

说明 \ US_ID	US2
名称	测试作业的建立、显示，作业的下发，失败作业的重新下发
验收标准	实现用户根据所属的测试集，定义测试作业类型。测试作业类型包括即时下发、延时下发和周期下发 3 种类型，测试作业优先级包括高、中等和低 3 个级别 实现用户对所属测试作业列表的监控功能 实现用户对失败的所属测试作业重新下发功能
优先级别	最高

US3 的优先级是高。虽然测试结果是用户最终要得到的输出，而前述的两个用户故事都是输入和过程环节，但在整个系统的开发进程中，测试结果是重要但不紧急的需求，其基于的前提更为紧急。测试结果的验收标准包括记录、查询和统计，见表 11-8。

表 11-8 测试结果用户故事分析

US_ID / 说明	US3
名称	测试结果的回写、显示
验收标准	一个测试作业一旦下发就必须有一个测试结果，无论是否成功或者超时 实现用户对所属测试结果的查询功能 实现用户对所属测试结果的统计功能
优先级别	高

US4 的优先级是最高，因为测试作业的下发要依赖于测试设备。验收标准包括用户和管理员对设备的配置和监控，见表 11-9。

表 11-9 测试设备用户故事分析

US_ID / 说明	US4
名称	测试设备的建立、显示、管理
验收标准	实现用户对所属设备的配置和监控功能 实现管理员对设备的定义与分配功能 测试设备的实时状态显示无误。只有测试队列中没有相关的测试作业，测试设备才会空闲
优先级别	最高

US5 的优先级是高，因为测试者的管理、注册、登录，以及权限分配等功能都是重要而不紧急的。作为示例，这部分被弱化了，其实该用户故事的验收标准可以列举出很多，这里 US5 以一条验收标准来代指完成上述 US1 ~ US4 的几个功能点，见表 11-10。

表 11-10 平台用户管理用户故事分析

US_ID / 说明	US5
名称	测试者的创建、修改、显示
验收标准	实现管理员对 ATUP 用户的维护
优先级别	高

11.1.3 迭代规划

作为开发者，首先会想到作为一个从无到有的项目，有许多基础工作要做，比如环境搭建、技术调研、原型测试，以及通用的核心代码编写。但作为用户，他只关心所需的功能是否实现，运行是否稳定、高效。因此，开发者以“小步快跑”的迭代开发来交付功能给用户是最终目的，从而，我们的基础工作应该以每次够用就好，以不断重构完善的方式来推动。

1. 用户故事与迭代

Iteration 就是一次迭代，是开发周期的基本单位，通常为 2 ~ 3 周。每个 Iteration 的权重与开发者人数、每日工作时等信息相关，这里假定参考值为 15，那么在做计划时，累计的用户故事的权重达到参考值，就不再加新的用户故事到当前的 Iteration 中。作为示例，我们将 ATUP 系统的开发分为两个迭代完成，每个迭代实现的用户故事权重之和为 12，见表 11-11 ~ 表 11-15。

表 11-11　用户故事的权重 1

US_ID	US1
名称	测试集和测试用例的创建、修改、显示
迭代编号	Iteration-1
权重	6

表 11-12　用户权重故事 2

US_ID	US2
名称	测试作业的建立、显示，作业的下发，失败作业的重新下发
迭代编号	Iteration-1
权重	6

表 11-13　用户权重故事 3

US_ID	US3
名称	测试结果的回写、显示
迭代编号	Iteration-2
权重	3

表 11-14　用户权重故事 4

US_ID	US4
名称	测试设备的建立、显示、管理
迭代编号	Iteration-2
权重	6

表 11-15　用户权重故事 5

US_ID	US5
名称	测试者的创建、修改、显示
迭代编号	Iteration-2
权重	3

2. 规划迭代任务

Planning 是在 Iteration 定义好用户故事后，对其进行任务定义的规划过程。任务会被分配到即将到来的 Sprint 中。Sprint 是开发周期的最小单位，通常一次迭代包含 1 ~ 2 个 Sprint。每个 Sprint 开始前都会有一次 Planning 来细化用户故事，粒度精确到可执行 / 可编码

的任务。

一个任务是一个用户故事可执行的基本单位，是对验收标准的分步实现。可以在任务开始前指定负责人，也可以在敏捷开发过程中动态分配。任务的状态从定义到完成可以有定义、处理中、完成和被阻塞等多种状态。每个任务包括一到多个的测试集和测试用例。没错，我们的自动化测试平台的开发也需要 CI 构建和自动化测试，因为自动化测试是履行敏捷开发的重要组成部分。作为示例，我们只列举测试集和测试用例建模一个任务的定义，见表 11-16。

表 11-16 测试集和测试用例建模任务

用户故事	US1: 测试集和测试用例的创建、修改、显示
TA_ID	TA1
名称	测试集和测试用例建模
描述	定义用户、测试集和测试用例的数据库模型 定义用户、测试集和测试用例对应的 Java 类 JPA 实现数据库 CRUD
工时评估	12 小时（2 人 / 日）
任务状态	定义
负责人	×××

11.2 ATUP 的设计

在完成用户故事的定义后，开发团队要基于本节讲述的敏捷流程来对需求做概要和详细设计，这里其实还包括了技术选型和部署拓扑设计。

11.2.1 开发和部署环境

我们根据用户故事和自身开发特征，选用三组开发工具和平台：开发、测试和部署运行，见表 11-17。

表 11-17 ATUP 环境与工具

项目	工具	版本
开发工具	JDK	1.7.0_45
	IntelliJ IDEA	13
静态页面部署	Nginx	
REST 式的 Web 服务部署	Tomcat	7.0.42
数据库	MySQL	5.6
构建工具	Maven	3.1.1
版本控制	Git	1.8.1
集成测试（CI）平台	Jenkins	
代码质量管理平台	SonarQube	

（续）

项目	工具	版本
开发环境（Development）		
开发平台	Ubuntu Desktop	13.10
测试环境（Staging）		
测试平台	Ubuntu Server	13.10
产品环境（Production）		
测试平台	Ubuntu Server	13.10

1. IntelliJ IDEA 配置

IDEA 并非 ATUP 唯一可以使用的 IDE，但是其智能和便捷的体验相对其他 IDE 较好，因此在本书最后一章，笔者还是推荐将完整示例运行在 IDEA 上。开发服务器使用 Tomcat 7，这个争议要少很多，毕竟目前 Servlet 容器没有比 Tomcat 更流行的了。在 IDEA 中配置使用 Tomcat 和在其他 IDE 中差别不大，如图 11-2 所示。

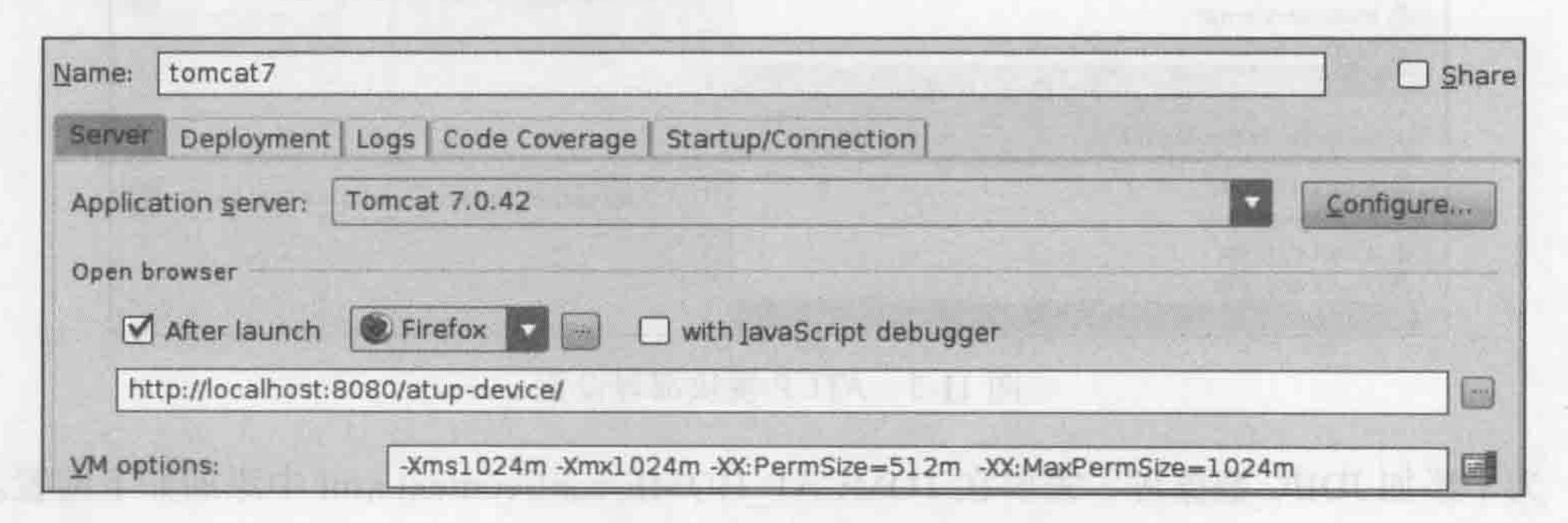


图 11-2　IDEA 内置 Tomcat 配置

如图 11-2 所示，在服务器配置页面，为开发服务器配置了虚拟机参数（见图 11-2 所示下方的“VM options”），这步的目的是扩大 JVM 堆的大小，以承载更多的对象实例化。否则，会在调试阶段出现堆内存溢出的问题。虚拟机配置的参考参数如下，可根据本机情况调整。

```
-Xms1024m -Xmx1024m -XX:PermSize=512m -XX:MaxPermSize=1024m
```

在部署页面，需要指定各模块的 context 名称和加载顺序，如图 11-3 所示。

如图 11-3 所示，ATUP 包含 5 个 war 包，分别是 REST 交互页面模块 atup-page、设备管理模块 atup-device、用户管理模块 atup-user、用例管理模块 atup-case，以及测试平台模块 atup-test-station。如果做一键发布，模块部署顺序可参考图 11-3。需要指出的是，atup-core 是 jar 包，是这 5 个 war 包的底层支持模块，无须部署。

2. Tomcat 配置

完成 IDE 配置后，为了运行 ATUP 示例，我们需要修改 Tomcat 的配置。这里只需要设置两个 Tomcat 配置：一个是在 Tomcat 的上下文中添加 JDBC 的数据源（必选），另一个是设

置 JVM 的参数以支持更大的堆存储空间（可选）。

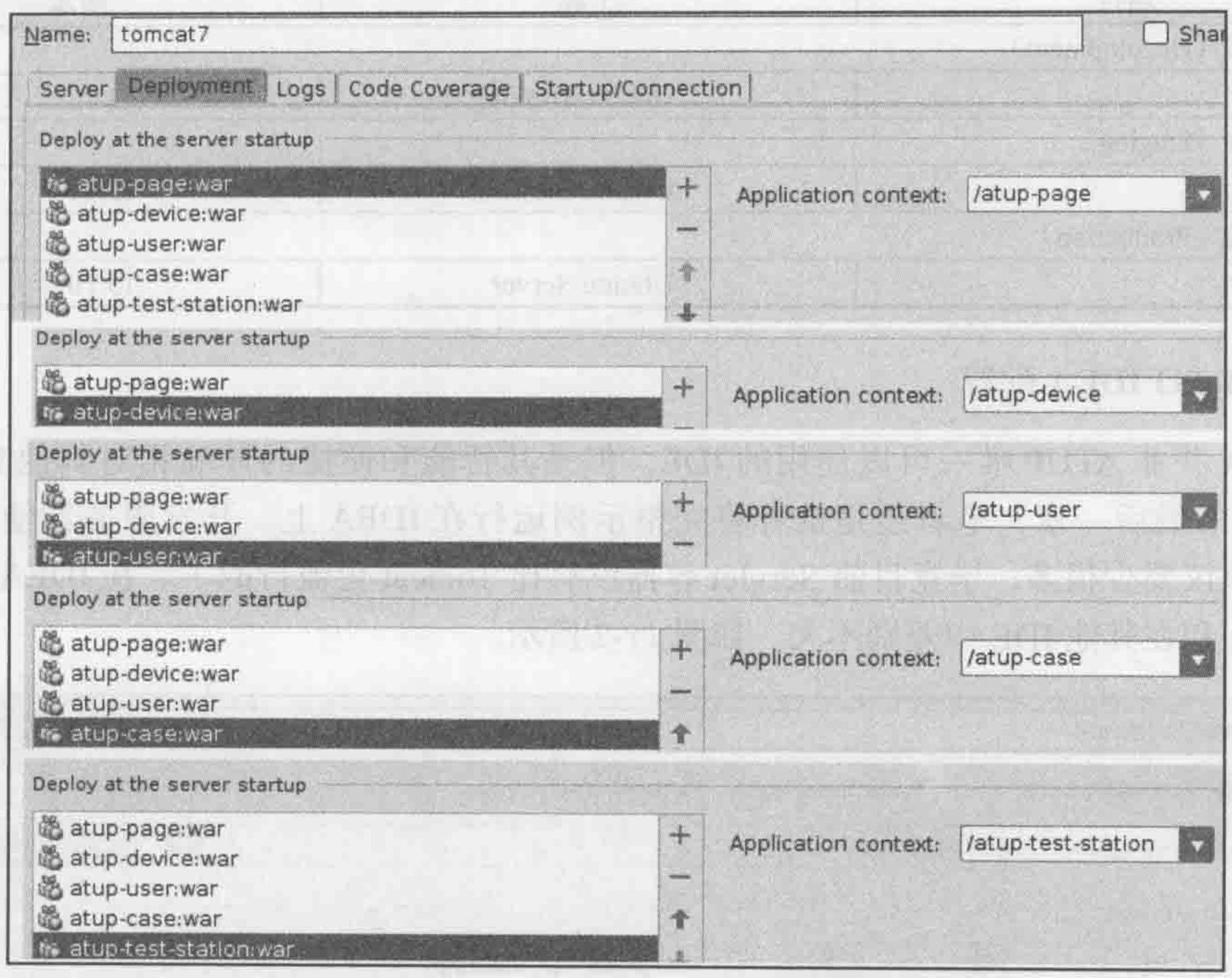


图 11-3 ATUP 模块部署设置

为了添加 JDBC 数据源，需要在 TOMCAT_HOME/conf/context.xml 中添加如下配置。

```
<Context>
...
<Resource name="jdbc/AtupDataSource"
   auth="Container"
   type="javax.sql.DataSource"
   driverClassName="org.gjt.mm.mysql.Driver"
   url="jdbc:mysql://localhost:3306/jaxrs2_atup"
   username="root"
   password="root"
   maxActive="20"
   maxIdle="10"
   maxWait="-1" />
</Context>
```

在这段代码中，定义了 MySQL 数据库驱动、数据 schema 为 jaxrs2_atup，以及用户名和密码是 root 等。

为了设定 JVM 参数，需要修改 TOMCAT_HOME/bin/catalina.sh 文件（Windows 平台是 catalina.bat 文件），添加如下配置。

```
JAVA_OPTS=' -server -Xms1024m -Xmx1024m -XX:PermSize=128m -XX:MaxNewSize=256m
```

```
-XX:MaxPermSize=256m'
```

11.2.2　模块定义和拓扑

ATUP 是使用 Maven 管理的多模块项目。父模块负责定义子模块构建顺序、依赖包版本定义、依赖包仓库定义及构建、Profile 等多模块通用配置。ATUP 父模块的 Maven 坐标信息如下所示。

```
<groupId>org.feuyeux.jaxrs2.atup</groupId>
<artifactId>atup</artifactId>
<packaging>pom</packaging>
<version>0.0.2-SNAPSHOT</version>
<name>ATUP Project Parent</name>
```

ATUP 父模块定义了 6 个子模块，示例代码如下所示。

```
<modules>
   <module>atup-core</module>
   <module>atup-device</module>
   <module>atup-user</module>
   <module>atup-case</module>
   <module>atup-test-station</module>
   <module>atup-page</module>
</modules>
```

在这段代码中，atup-core 是核心功能模块，负责提供 REST 式的 Web 服务中的基础和通用的代码实现。atup-device 是测试设备模块。atup-user 是用户管理模块。atup-case 模块用于实现测试集、测试用例、测试作业和测试结果。atup-page 模块是 ATUP 系统的用户界面模块，是纯 HTML 实现的。严格来说 atup-test-station 模块，不是 ATUP 系统的组成部分，只是为了演示 ATUP 系统和测试平台的交互，以及测试 ATUP 平台的测试作业和测试结果的工作状态。

ATUP 的依赖包版本信息定义如下所示。

```
<properties>
   <JDK.version>1.7</JDK.version>
   <jersey.version>2.9</jersey.version>
   <junit.version>4.11</junit.version>
   <log4j.version>2.0-beta9</log4j.version>
   <spring.version>3.2.5.RELEASE</spring.version>
   <eclipselink.version>2.5.0</eclipselink.version>
   <jquery.version>2.0.3</jquery.version>
```

在这段代码中，定义了 JDK 版本为 1.7、Jersey 版本为 2.9、单元测试工具 JUnit 版本为 2.4.1、日志工具 Log4j 版本为 2.0-beta9、IoC 和事务管理容器 Spring 版本为 3.2.5.RELEASE、JPA 参考实现 EclipseLink 版本为 2.5.0，以及静态页面脚本库 jQuery 版本为 2.0.3。

ATUP 采用动静分离的部署方式。静态页面模块 atup-page 部署在 Nginx 服务器上，为不同身份的用户提供访问并控制访问权限（后面讲述原因）。用户、设备和测试用例分别部署在

独立的 Tomcat 服务器上，以便分担压力（作为演示项目，部署都是单点的，演示项目实际上支持大用户的并发访问比较困难）。作为演示项目，数据库服务器只部署一台，为全部模块提供数据。测试平台模块 atup-test-station 单独部署，作为演示也使用 Tomcat 服务器——对于 ATUP 而言，只要提供 REST 式的 Web 服务即可，并不关心其实现。ATUP 的拓扑如图 11-4 所示。

如图 11-4 所示，ATUP 分别部署在 Nginx、Tomcat 上，数据库部署在 MySQL 上，测试平台（又称执行机）通常部署在 Linux 系统，可以是 Tomcat 上的 Servlet 应用（比如 Jenkins 的 Slave），也可以是 Python、Ruby、Perl、TCL 等设备通信的脚本环境。4 种角色的用户将通过 Nginx 进入 ATUP 系统。

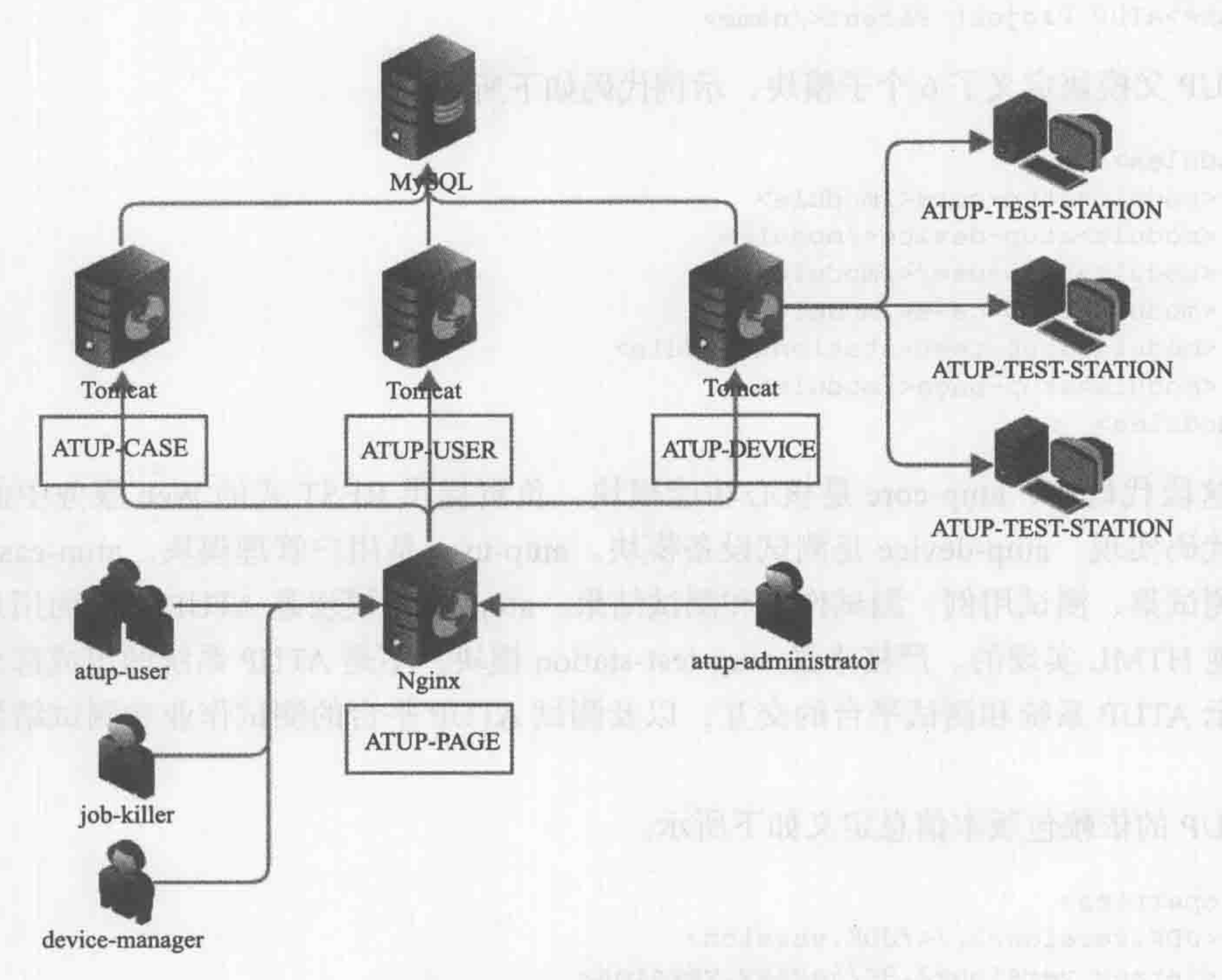


图 11-4 ATUP 部署拓扑图

11.2.3 持续集成流程

从狭义上讲，持续集成流程是对应用集成测试持续进行的流程，流程的核心是对集成本身的持续；从广义上讲，已不是这个名词本身的含义，包括了对部署和交付的持续。因为如果持续集成做得好，那么可以做到通过持续进行的集成测试后的产品是可以直接部署上线的，上线的产品是可以满足用户需求的。所以，持续发布和持续交付是对持续集成的延伸和肯定。下面我们结合这一理论完成对 ATUP 持续集成的设计。

1. 持续集成

ATUP 的持续集成伴随着敏捷开发的进程进行。每个功能点开发完毕，必须进行单元测试，确保代码的功能性没有问题，然后向版本控制服务器提交代码。每次向版本控制服务器提交都会触发一次 CI 构建，包括编译、基础功能测试和打包。在这个过程中，会对项目进行单元测试和集成测试。同时，CI 服务器 Jenkins 会使用代码覆盖率工具检查当前版本中代码被测试的覆盖情况，Jenkins 还会触发代码质量管理平台 SonarQube 以检验当前版本代码的质量。例如，在每天夜间，Jenkins 会周期性地启动 Nightly 构建，包括编译、全量测试和打包。在这个过程中，会对项目额外进行系统测试。在每个开发迭代结束前，会对这个迭代的开发进行功能性和性能测试。

2. 持续发布和交付

ATUP 的持续发布通常会发生在每个迭代结束前。在测试通过后，DevOps 人员会触发 Jenkins 的自动部署作业，将构建好的当前项目部署到 Staging 环境。然后，在 Staging 环境中，开发团队需要对其进行 GUI 自动化测试和手动测试以校验其功能性，同时，启动自动化测试脚本来对其进行压力测试。当 Staging 环境中的项目测试通过后，DevOps 人员会将这个版本的项目部署到 Production 环境。ATUP 的持续发布模型如图 11-5 所示。

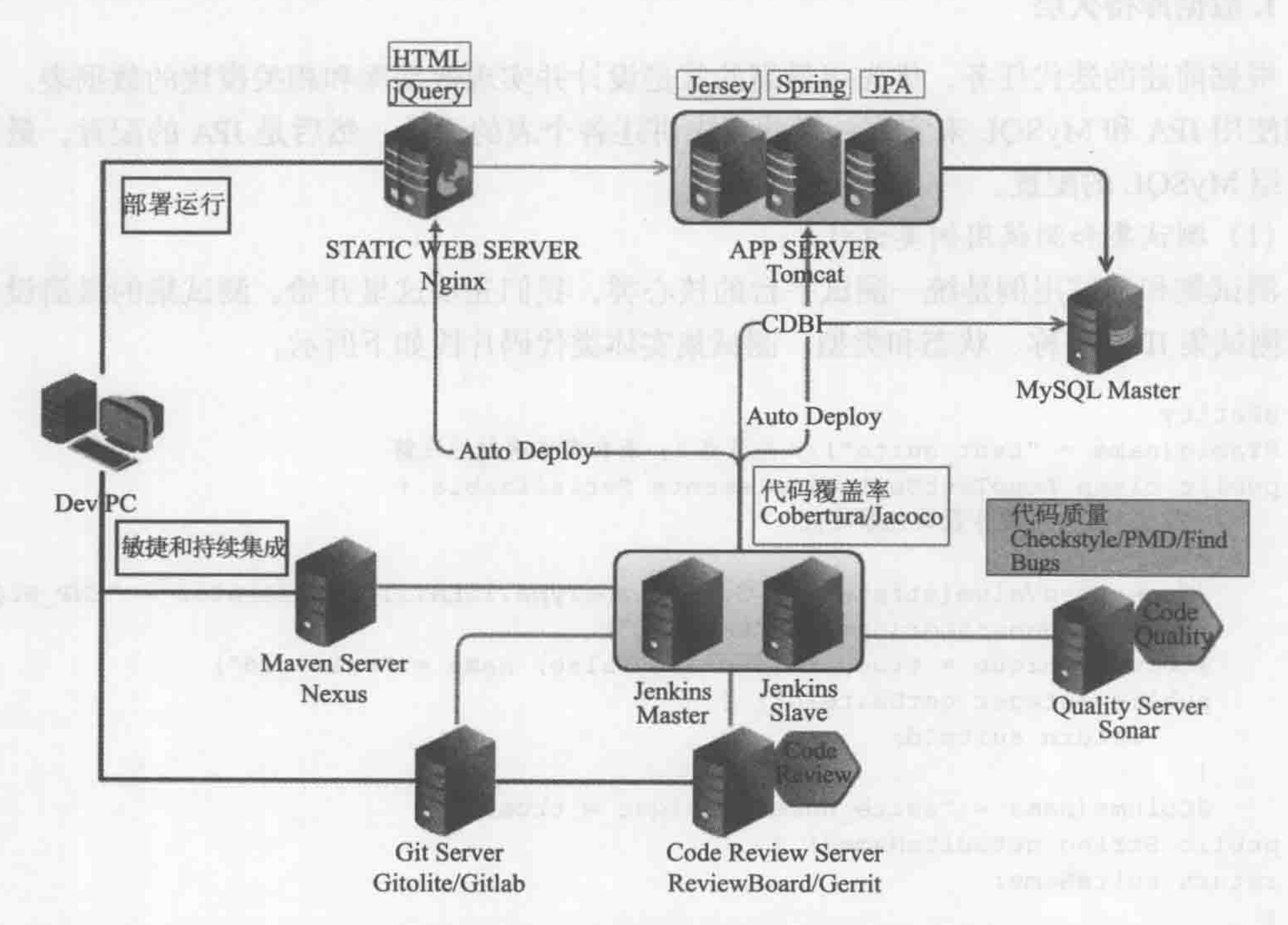


图 11-5　ATUP 持续发布示意图

在图 11-5 中，根据服务器用途可以划分为上下两类，图上部的机器用于部署 ATUP 应

用，包括静态应用服务器、应用服务器和数据库服务器；图下部的机器用于开发管理，包括Maven 私服、Git 私服、持续集成服务器、代码评审和代码质量计算服务器。

ATUP 的持续交付流程发生在迭代结束时，在这个阶段会对这次迭代中定义的每个用户故事进行验收测试。当用户接收后，这次的交付流程即宣告完毕。

11.3 ATUP 的实现

ATUP 的实现包括三个部分。首先实现的是通用的核心模块，包括数据库配置与访问、REST 客户端的封装、用户验证和权限、作业分派以及持续发布的统一配置。接下来是用户模块、设备模块、用例模块和静态页面模块等模块功能的实现。最后实现各模块的持续交付，使其成为一个产品。

11.3.1 Sprint1 核心功能

本节讲述 ATUP 的核心模块 atup-core 的实现是 Sprint1 的工作内容，将在本节作为Sprint1 的实现示例来讲。每个步骤会根据前述的用户故事和迭代规划中的优先级来进行。

1. 数据库持久层

根据前述的迭代任务，优先级最高的就是设计并实现数据库和相关模块的数据表。由于我们使用 JPA 和 MySQL 来实现，因此首先讲述各个表的设计，然后是 JPA 的配置，最后简要介绍 MySQL 的配置。

（1）测试集和测试用例类设计

测试集和测试用例是统一测试平台的核心类，我们先从这里开始。测试集的最简设计应包括测试集 ID、名称、状态和类型。测试集实体类代码片段如下所示。

```
@Entity
@Table(name = "test_suite")//关注点 1：表和实体映射的注解
public class AtupTestSuite implements Serializable {
   //关注点 2：主键字段的注解定义
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator(name = "EMP_SEQ")
    @Column(unique = true, nullable = false, name = "suite_id")
    public Integer getSuiteId() {
        return suiteId;
    }
    @Column(name = "suite_name", unique = true)
public String getSuiteName() {
return suiteName;
}
    @Column(name = "suite_type")
public Integer getSuiteType() {
```

```
return suiteType;
}
    @Column(name = "suite_status")
public Integer getSuiteStatus() {
return suiteStatus;
    }
}
```

在这段代码中，@Table 注解将类对应到 test_suite 表，见关注点 1；@Column 将类的属性对应到 test_suite 表的字段，其中主键字段为 suite_id，unique = true 代表唯一，nullable = false 代表非空。该字段根据数据库的 sequence 自增，见关注点 2。

此外，为了方便查询，我们定义了 JPA 的名称查询 findBySuiteStatus 和 findBySuiteName，分别根据测试集状态和测试集名称返回该表的数据集，定义如下所示。

```
@NamedQueries({
@NamedQuery(name = "findBySuiteStatus",
query = "SELECT testSuite FROM AtupTestSuite testSuite
WHERE testSuite.suiteStatus= :suiteStatus"),
@NamedQuery(name = "findBySuiteName",
query = "SELECT testSuite FROM AtupTestSuite testSuite
WHERE testSuite.suiteName= :suiteName")})
```

接下来是测试集的数据表 test_suite 的 DDL，示例代码如下所示。

```
CREATE TABLE `test_suite`(
  `suite_id` int(11) NOT NULL AUTO_INCREMENT,
  `suite_name` varchar(255) DEFAULT NULL,
  `suite_status` int(11) DEFAULT NULL,
  `suite_type` int(11) DEFAULT NULL,
  PRIMARY KEY(`suite_id`),
  UNIQUE KEY `suite_id`(`suite_id`),
  UNIQUE KEY `suite_name`(`suite_name`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

在这段代码中，定义了最简的 4 个字段：主键、测试集名称、测试集状态和测试集类型。为了简便，我们没有为测试集的各种状态和类型单独建立表。测试集的状态包括：可用（1）和禁用（0）。测试集的类型包括：普通测试集、性能测试集和边缘测试集，示例如下所示。

- NORMAL_SUITE = 1
- PERFORMANCE_SUITE = 2
- EDGE_SUITE = 3

详情可参见 org.feuyeux.jaxrs2.atup.core.constant.AtupParam 接口。

为了更方便演示 ATUP，我们为测试集表创建初始测试集，内容如下所示。

```
INSERT INTO `jaxrs2_atup`.`test_suite`
(`suite_id`,`suite_name`,`suite_status`,`suite_type`)
VALUES
(1,'Performance Test',1,2),
(2,'GUI Test',1,1),
```

```
(3,'Unit Test',1,1),
(4,'Integration Test',1,1),
(5,'System Test',1,1);
```

讲述完测试集，接下来介绍测试用例。包含于测试集的测试用例，其最简信息包括测试用例 ID、名称、内容、状态以及所属的测试集。此外，我们为其添加了创建时间和最后修改时间。测试用例实体类代码片段如下所示。

```
@Entity
@Table (name = "test_case")
public class AtupTestCase implements Serializable {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator (name = "EMP_SEQ")
    @Column (unique = true, nullable = false, name = "case_id")
public Integer getCaseId() {
return caseId;
    }
    @Column (name = "case_name", unique = true)
public String getCaseName() {
return caseName;
    }
    @ManyToOne //关注点 1: 测试用例和测试集的多对一关系注解
    @JoinColumn (name = "suiteId")
public AtupTestSuite getSuite() {
return suite;
    }
    @Column (name = "case_body")
    @XmlAttribute
public String getCaseBody() {
return caseBody;
    }
    @Column (name = "create_time")
    @XmlJavaTypeAdapter (JaxbDateSerializer.class)
public Date getCreateTime() {
return createTime;
    }
    @Column (name = "update_time")
    @XmlJavaTypeAdapter (JaxbDateSerializer.class)
public Date getUpdateTime() {
return updateTime;
    }
    @Column (name = "case_status")
public Integer getCaseStatus() {
return caseStatus;
    }
```

这段代码与测试集类似，额外要说明的是，测试用例和测试集是多对一（ManyToOne）的关系，即多个测试用例属于一个测试集，见关注点 1。

此外，为了方便查询，我们定义了 JPA 的名称查询 findByStatus 和 findByName，分别根据测试用例状态和名称返回该表的数据集，定义如下所示。

```
@NamedQueries({
@NamedQuery(name = "findByStatus",
query = "SELECT testCase FROM AtupTestCase testCase WHERE testCase.caseStatus= :caseStatus"),
@NamedQuery(name = "findByName",
query = "SELECT testCase FROM AtupTestCase testCase WHERE testCase.caseName= :caseName")})
```

接下来是测试用例的数据表 test_case 的 DDL，示例代码如下所示。

```
CREATE TABLE `test_case` (
  `case_id` int(11) NOT NULL AUTO_INCREMENT,
  `case_name` varchar(255) DEFAULT NULL,
  `case_body` varchar(255) DEFAULT NULL,
  `case_status` int(11) DEFAULT NULL,
  `create_time` datetime DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  `suiteId` int(11) DEFAULT NULL,
  PRIMARY KEY (`case_id`),
  UNIQUE KEY `case_id` (`case_id`),
  UNIQUE KEY `case_name` (`case_name`),
  KEY `FK_test_case_suiteId` (`suiteId`),
  CONSTRAINT `FK_test_case_suiteId` FOREIGN KEY (`suiteId`)
  REFERENCES `test_suite` (`suite_id`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对于这段代码要额外说明的是，测试状态包括：正常（0）和禁用（1），详见 org.feuyeux.jaxrs2.atup.core.constant.AtupParam 接口。

（2）用户类设计

用户类用于 ATUP 系统标识当前操作者的身份，其最简信息包括用户 ID、用户名和密码。此外，我们添加了用户角色和用户状态。该类的代码片段如下所示。

```
@Entity
@Table(name = "atup_user")
public class AtupUser implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator(name = "EMP_SEQ")
    @Column(unique = true, nullable = false, name = "user_id")
public Integer getUserId() {
return userId;
    }
    @Column(name = "user_role")
public Integer getUserRole() {
return userRole;
    }
    @Column(name = "user_name", unique = true)
public String getUserName() {
return userName;
    }
    @Column(name = "user_pwd")
public String getPassWord() {
return passWord;
```

```
    }
    @Column (name = "user_status")
public Integer getStatus() {
return status;
    }
}
```

与测试集实体类相似，@Table 注解将类对应到 atup_user 表，@Column 将类的属性对应到 atup_user 表的字段，其中主键字段 user_id 根据数据库的 sequence 自增。

为了方便查询，我们定义了 JPA 的名称查询 findByUserName，根据用户名返回状态为正常的用户数据，定义如下所示。

```
@NamedQueries({
@NamedQuery(name = "findByUserName",
query = "SELECT atupUser FROM AtupUser atupUser
WHERE atupUser.userName= :userName and atupUser.status=0")
})
```

用户实体类对应的数据表 atup_user 的 DDL 如下所示。

```
CREATE TABLE `atup_user` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(255) DEFAULT NULL,
  `user_pwd` varchar(255) DEFAULT NULL,
  `user_role` int(11) DEFAULT NULL,
  `user_status` int(11) DEFAULT NULL,
  PRIMARY KEY (`user_id`),
  UNIQUE KEY `user_id` (`user_id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

在这段代码中，为了简便，我们没有为用户的权限和各种状态单独建立表。这里简述如下，用户角色包括：系统管理员（1）、作业管理员（2）、设备管理员（3）和普通用户（4）。用户状态包括：正常（0）和停用（1）。详见 org.feuyeux.jaxrs2.atup.core.constant.AtupParam 接口。

为了更方便演示 ATUP，我们为该表创建初始用户，见表 11-18。

表 11-18 ATUP 初始用户列表

用户角色描述	用户名 / 密码
ATUP 系统管理员	atupAdmin/jaxman
ATUP 作业管理员	atupJobKiller/jaxman
ATUP 设备管理员	atupDeviceKeeper/jaxman
ATUP 普通用户	xer/ xer

表 11-14 对应的数据库初始化脚本如下。

```
INSERT INTO `jaxrs2_atup`.`atup_user`
(`user_id`,`user_name`,`user_pwd`,`user_role`,`user_status`)
VALUES (1,'atupAdmin','650b81ee61374ef758f6e6b8ab68426e',1,0),
```

```
(2,'atupJobKiller','650b81ee61374ef758f6e6b8ab68426e',2,0),
(3,'atupDeviceKeeper','650b81ee61374ef758f6e6b8ab68426e',3,0),
(4,' xer' ,' faa709c5035aea00f9efb278f2ad5df0' ,4,0);
```

从这段初始化脚本中可以看出，ATUP 系统对用户密码进行了 MD5 加密。这个处理过程是在 atup-page 完成的，即客户端加密。使用的算法来自 https://github.com/blueimp/JavaScript-MD5，相关脚本文件如下所示。

```
https://raw.github.com/blueimp/JavaScript-MD5/master/js/md5.min.js
```

（3）设备类设计

设备类定义了测试设备的基本信息，包括设备 ID、主机 IP、名称、状态和类型，此外还有设备所属用户、添加时间和更新时间。其实体类代码片段如下所示。

```
@Entity
@Table (name = "atup_device")
public class AtupDevice implements Serializable {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator (name = "EMP_SEQ")
    @Column (unique = true, nullable = false, name = "device_id")
public Integer getDeviceId() {
return deviceId;
    }
    @ManyToOne //关注点 1: 设备和用户的多对一关系注解
    @JoinColumn (name = "userId")
public AtupUser getUser() {
return user;
    }
    @Column (name = "device_host", unique = true)
public String getDeviceHost() {
return deviceHost;
    }
    @Column (name = "device_name", unique = true)
public String getDeviceName() {
return deviceName;
    }
    @Column (name = "device_status")
public Integer getDeviceStatus() {
return deviceStatus;
    }
    @Column (name = "device_type")
public Integer getDeviceType() {
return deviceType;
    }
    @Column (name = "create_time")
public Date getCreateTime() {
return createTime;
    }
    @Column (name = "update_time")
public Date getUpdateTime() {
```

```
    return updateTime;
    }
}
```

与测试集实体类类似，@Table 注解将类对应到 atup_device 表，@Column 将类的属性对应到 atup_device 表的字段，其中主键字段 device_id 根据数据库的 sequence 自增。额外要说明的是，设备和用户之间的关系是多对一，见关注点 1。

为了方便查询，我们定义了 JPA 的名称查询 findByDeviceIp 和 findDevicesByUser，分别根据设备主机 IP 和名称返回该表的数据集，定义如下所示。

```
@NamedQueries({
@NamedQuery(name = "findByDeviceIp",
query = "SELECT atupDevice FROM AtupDevice atupDevice
WHERE atupDevice.deviceHost= :deviceHost"),
@NamedQuery(name = "findDevicesByUser",
query = "SELECT atupDevice FROM AtupDevice atupDevice
WHERE atupDevice.user.userId= :userId")})
```

设备实体类对应的数据表 atup_device 的 DDL 如下所示。

```
CREATE TABLE `atup_device` (
  `device_id` int(11) NOT NULL AUTO_INCREMENT,
  `device_host` varchar(255) DEFAULT NULL,
  `device_name` varchar(255) DEFAULT NULL,
  `device_type` int(11) DEFAULT NULL,
  `device_status` int(11) DEFAULT NULL,
  `userId` int(11) DEFAULT NULL,
  `create_time` datetime DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  PRIMARY KEY (`device_id`),
  UNIQUE KEY `device_id` (`device_id`),
  UNIQUE KEY `device_host` (`device_host`),
  UNIQUE KEY `device_name` (`device_name`),
  KEY `FK_atup_device_userId` (`userId`),
  CONSTRAINT `FK_atup_device_userId` FOREIGN KEY (`userId`)
REFERENCES `atup_user` (`user_id`)
)
```

在这段代码中，设备状态字段 device_status 的值包括：空闲（0）、忙碌（1）和宕机（2），设备类型字段 device_type 的值包括：快速测试（1）和慢速测试（2）。快速测试主要包括冒烟测试、SANITY 测试、回归测试等测试时间不是很长的测试；慢速测试主要包括性能测试、系统测试等全面或者耗时的测试。详情可参见 org.feuyeux.jaxrs2.atup.core.constant.AtupParam 接口。

（4）测试结果设计

测试结果是测试作业根据测试用例在指定设备上运行的测试结果，包括如下最简信息：测试结果 ID、内容、状态、测试用例、测试者、测试设备，以及生成时间和更新时间。其实体类代码片段如下所示。

```
@Entity
@Table (name = "test_result")
public class AtupTestResult implements Serializable {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY, generator = "EMP_SEQ")
    @SequenceGenerator (name = "EMP_SEQ")
    @Column (unique = true, nullable = false, name = "result_id")
public Integer getResultId() {
return resultId;
    }
    @ManyToOne //关注点 1: 测试结果和测试用例的多对一注解
    @JoinColumn (name = "caseId")
public AtupTestCase getTestCase() {
return testCase;
    }
    @ManyToOne//关注点 2: 测试结果和用户的多对一注解
    @JoinColumn (name = "userId")
public AtupUser getUser() {
return user;
    }
    @Column (name = "result_status")
public Integer getResultStatus() {
return resultStatus;
    }
    @Column (name = "result_body")
public String getResultBody() {
return resultBody;
    }
    @Column (name = "create_time")
public Date getCreateTime() {
return createTime;
    }
    @Column (name = "update_time")
public Date getUpdateTime() {
return updateTime;
    }
    @ManyToOne//关注点 3: 测试结果和测试设备的多对一注解
    @JoinColumn (name = "deviceId")
public AtupDevice getDevice() {
return device;
    }
}
```

在这段代码中，@Table 注解将类对应到 test_result 表，@Column 将类的属性对应到 test_result 表的字段，其中主键字段 result_id 根据数据库的 sequence 自增。测试结果和测试用例、设备、用户之间的关系都是多对一，见关注点 1 ~ 关注点 3。

为了方便查询，我们定义了 JPA 的名称查询 findResultByTestCase、findResultByStatus 和 findResultByUser，分别根据测试用例、测试结果状态和测试者返回该表的数据集，定义如下所示。

```
@NamedQueries({
@NamedQuery(name = "findResultByTestCase",
query = "SELECT testResult FROM AtupTestResult testResult
WHERE testResult.testCase= :testCase"),
@NamedQuery(name = "findResultByStatus",
query = "SELECT testResult FROM AtupTestResult testResult
WHERE testResult.resultStatus= :resultStatus"),
@NamedQuery(name = "findResultByUser",
query = "SELECT testResult FROM AtupTestResult testResult
WHERE testResult.user.userId= :userId")})
```

测试结果实体类对应的数据表 test_result 的 DDL 如下所示。

```
CREATE TABLE `test_result` (
  `result_id` int(11) NOT NULL AUTO_INCREMENT,
  `result_body` varchar(255) DEFAULT NULL,
  `result_status` int(11) DEFAULT NULL,
  `caseId` int(11) DEFAULT NULL,
  `deviceId` int(11) DEFAULT NULL,
  `userId` int(11) DEFAULT NULL,
  `create_time` datetime DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  PRIMARY KEY(`result_id`),
  UNIQUE KEY `result_id`(`result_id`),
  KEY `FK_test_result_caseId`(`caseId`),
  KEY `FK_test_result_userId`(`userId`),
  KEY `FK_test_result_deviceId`(`deviceId`),
  CONSTRAINT `FK_test_result_caseId` FOREIGN KEY(`caseId`)
  REFERENCES `test_case`(`case_id`),
  CONSTRAINT `FK_test_result_deviceId` FOREIGN KEY(`deviceId`)
  REFERENCES `atup_device`(`device_id`),
  CONSTRAINT `FK_test_result_userId` FOREIGN KEY(`userId`)
  REFERENCES `atup_user`(`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

在这段代码中，测试结果状态字段 result_status 包括：成功（0）、失败（1）和未知错误（2），详情可参见 org.feuyeux.jaxrs2.atup.core.constant.AtupParam 接口。

（5）数据源配置

JPA 数据源的配置根据事务类型 transactionType 的不同分为 RESOURCE_LOCAL 和 JTA 两种方式。

阅读指南

本章使用的 JPA 实现是 EclipseLink，有关配置的详细信息可参考如下地址。

JPA 配置列表：http://wiki.eclipse.org/EclipseLink/Examples/JPA。

Tomcat 容器配置：http://wiki.eclipse.org/EclipseLink/Examples/JPA/Tomcat_Web_Tutorial。

EclipseLink 缓存配置：http://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching。

RESOURCE_LOCAL 实例如下，配置文件 src/main/resources/META-INF/persistence.xml

中包括实体类列表和数据源信息。

```
<persistence-unit name="jpaMysql2" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupDevice</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestCase</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestResult</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestSuite</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupUser</class>
  <properties>
     <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
     <property name="javax.persistence.jdbc.url"
     value="jdbc:mysql://localhost:3306/jaxrs2_atup" />
     <property name="javax.persistence.jdbc.user" value="root" />
     <property name="javax.persistence.jdbc.password" value="root" />
     <property name="eclipselink.ddl-generation" value="create-tables" />
  </properties>
</persistence-unit>
```

JTA 实例如下，配置文件 src/main/resources/META-INF/persistence.xml 中包括实体类列表，其数据源配置是容器级别的。本章使用 Tomcat 作为容器，可参见 11.2.1 节。

```
<persistence-unit name="jpaMysql" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <jta-data-source>java:/comp/env/jdbc/AtupDataSource</jta-data-source>
  <non-jta-data-source>java:/comp/env/jdbc/AtupDataSource</non-jta-data-source>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupDevice</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestCase</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestResult</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupTestSuite</class>
  <class>org.feuyeux.jaxrs2.atup.core.domain.AtupUser</class>
  <properties>
      <property name="eclipselink.ddl-generation" value="create-tables" />
  </properties>
</persistence-unit>
```

(6) Ubuntu 系统下 MySQL 简明参考

这里以 Ubuntu 系统下的 MySQL 的安装和配置作为示例。如果服务器存在旧版本，那么可以先使用如下命令卸载旧版本。

```
sudo apt-get remove --purge mysql-server mysql-client mysql-common
sudo apt-get autoremove
sudo apt-get autoclean
```

使用 wget 下载 64 位 Debian 安装包。

```
wget http://cdn.mysql.com/Downloads/MySQL-5.6/mysql-5.6.15-debian6.0-x86_64.deb
```

安装 Debian 安装包命令如下。注意，其默认安装路径是 /opt/mysql/。

```
sudo apt-get install libaio-dev
sudo dpkg -i mysql-5.6.15-debian6.0-x86_64.deb
```

安装完毕后，可参考如下步骤配置 MySQL 服务器，详细信息可参考：http://dev.mysql.com/doc/refman/5.6/en/binary-installation.html。

```
sudo groupadd mysql
sudo useradd -r -g mysql mysql
cd /opt/mysql/server-5.6/
sudo mkdir data
sudo chown mysql:mysql data
sudo mkdir log
sudo chown mysql:mysql log
sudo cp /opt/mysql/server-5.6/share/english/errmsg.sys /usr/share/mysql/errmsg.sys
sudo chown mysql:mysql /usr/share/mysql/errmsg.sys
sudo scripts/mysql_install_db --user=mysql --no-defaults
sudo cp support-files/my-default.cnf /etc/mysql/my.cnf
sudo cp support-files/mysql.server /etc/init.d/mysql.server
sudo cp support-files/mysql-log-rotate /etc/logrotate.d/mysql.server
```

完成上述配置后，使用“sudo nano /etc/mysql/my.cnf”命令开始对配置文件进行编辑，参考配置如下所示。

```
[client]
port            = 3306
socket          = /tmp/mysql56.sock
[mysqld_safe]
port            = 3306
socket          = /tmp/mysql56.sock
nice            = 0
[mysqld]
user            = mysql
pid-file        = /tmp/mysql56.pid
port            = 3306
socket          = /tmp/mysql56.sock
basedir         = /opt/mysql/server-5.6
datadir         = /opt/mysql/server-5.6/data
tmpdir          = /tmp
lc-messages-dir = /usr/share/mysql
#bind-address           = 127.0.0.1
key_buffer              = 16M
max_allowed_packet      = 16M
thread_stack            = 192K
thread_cache_size       = 8
myisam-recover          = BACKUP
query_cache_limit       = 1M
query_cache_size        = 16M
log_error= /opt/mysql/server-5.6/log/error.log
expire_logs_days        = 10
max_binlog_size         = 100M
[mysqldump]
quick
quote-names
max_allowed_packet      = 16M
[mysql]
```

```
#no-auto-rehash # faster start of mysql but no tab completition
[isamchk]
key_buffer              = 16M
sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
```

最后，启动 MySQL 服务，并登录验证。

```
sudo service mysql.server start
mysqladmin -u root password 'root'
```

2. 数据库访问层

设计好数据库并配置好连接后，接下来的工作是对数据库的访问。我们分别讲述通用访问类和一个具体的访问类，以示说明。

（1）AtupDao 类

AtupDao 类是 ATUP 系统的通用数据库访问层类，因此其命名中接收一个泛型，示例代码如下所示。

```
package org.feuyeux.jaxrs2.atup.core.dao;
//关注点 1: 泛型定义
public class AtupDao<T> {
    //关注点 2: 日志实例
    private final Logger log = LogManager.getLogger(AtupDao.class.getName());
    @PersistenceContext
    EntityManager entityManager;//关注点 3: 实体管理类实例
    private Class<T> entityClass; //关注点 4: 特化实体类的类型
    @SuppressWarnings("unchecked")
public AtupDao() {
final Type genericSuperclass = getClass().getGenericSuperclass();
      if(genericSuperclass instanceof ParameterizedType) {
          final ParameterizedType parameterizedType =
          (ParameterizedType) genericSuperclass;
final Type[] actualTypeArguments = parameterizedType.getActualTypeArguments();
          this.entityClass =(Class<T>) actualTypeArguments[0];
      }
    }
```

在这段代码中，泛型 T 的特化使用上文提到的 5 个实体类，见关注点 1。AtupDao 类包含 3 个属性，第一个属性是日志类实例，见关注点 2，ATUP 全局使用了 Log4j 2 作为日志记录工具，可参考下文相关内容；第二个属性是实体管理类实例，见关注点 3，类似 Hibernate 的 Session 实例，该属性使用了 @PersistenceContext 注解，表示该实例是一个持久化上下文，由容器管理。容器提供 EntityManagerFactory 的实例来维护这个属性的生命周期；第三个属性是指定特化的实体类类型，见关注点 4，在构造子中通过反射机制动态地为其赋值。这个属性的作用是为查询提供特化类型参数。

AtupDao 类为子类提供了通用的数据库访问实现，比如分页查询、插入、更新、删除等通用操作，示例代码片段如下所示。

```
    private List<T> findAll(final boolean isPaging, final int firstResult,
    final int maxResults) {
  final CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        final CriteriaQuery<T> cq = cb.createQuery(entityClass);
        final TypedQuery<T> q = entityManager.createQuery(cq);
        if(isPaging) {
            q.setMaxResults(maxResults);
            q.setFirstResult(firstResult);
        }
    return q.getResultList();
    }
    @Transactional //关注点1: 容器管理事务注解
    public T save(final T entity) {
        entityManager.persist(entity);
    //关注点2: 为调用者提供插入后的实体类实例
    entityManager.flush();
    return entity;
    }
```

在这段代码中，写操作使用了 @Transactional 注解，表示该方法执行事务提交，见关注点 1。

JPA 的插入接口没有返回值，这对于 REST 接口为客户端提供返回值来说，非常不方便。一个变通的做法是使用 EntityManager 的 flush() 方法更新内存中的实体类实例，然后将其返回。此时，该实体类实例已经包含数据库为之生成的主键信息，见关注点 2。

宅人坑事

将 @Transactional 注解标记在数据访问层的这种处理是很有争议的，因为一个业务层的原子操作通常不止一张表的一次写操作，如果是这样，那么这个 @Transactional 注解应当标记在业务层的相关方法上。但是，在能确保该操作在业务中为原子操作的前提下，本例的这种做法有其细粒度的优势。

（2）AtupDeviceDao 类

AtupDao 的子类分别对应一个实体类，子类之间的关系是平级的。这里以设备访问类 AtupDeviceDao 为例，其他子类的实现类似。该类集成了 AtupDao 并提供泛型的特化类型为 AtupDevice。示例代码如下所示。

```
@Repository //关注点1: Spring 数据访问层组件注解
public class AtupDeviceDao extends AtupDao<AtupDevice> {
//关注点2: 按用户查询设备列表
    public List<AtupDevice> findDevicesByUser(final Integer userId) {
        return entityManager.createNamedQuery("findDevicesByUser",
        AtupDevice.class).setParameter("userId", userId).getResultList();
    }
```

在这段代码中，AtupDeviceDao 类使用了 @Repository 注解，表示该类是 Spring 的组件类，其实例由 Spring 容器管理，见关注点 1。

对于实体类特殊的访问方法，定义在 AtupDao 的子类中。这里以 JPA 名称查询 findDevicesByUser 为例，在 AtupDeviceDao 中，定义了 findDevicesByUser() 方法，见关注点 2。该方法调用 EntityManager 的 createNamedQuery() 方法创建一个名称查询，参数是 AtupDevice 中定义的名称查询名称 findDevicesByUser 和 AtupDevice 类型。findDevicesByUser 接收的参数信息是用户 ID，最后调用查询接口的 getResultList() 方法返回结果集。

3. 日志记录

ATUP 模块使用开源日志工具 Log4j 的增强版本 Log4j 2，这个版本采用异步 I/O，增强了吞吐能力，详见其官方网站：http://logging.apache.org/log4j/2.x/。其默认配置文件的路径和名称是：src\main\resources\log4j2.xml，示例如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="off">
  <Appenders> // 关注点 1: Appender 的配置
// 关注点 2: 日志输出路径
<File name="file" fileName="${sys:user.home}/atup_log/atup_case.log">
<PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>// 关注点 3: 日志输出格式
        </PatternLayout>
    </File>
    <Console name="STDOUT" target="SYSTEM_OUT">
        <PatternLayout pattern="%c{1.} %m%n"/>
    </Console>
</Appenders>
<Loggers>
    <Root level="DEBUG"> // 关注点 4: 日志输出级别
        <AppenderRef ref="file" level="DEBUG"/>
        <AppenderRef ref="STDOUT" level="DEBUG"/>
    </Root>
 </Loggers>
</Configuration>
```

在这段代码中，ATUP 开发中使用了两个 Appender，分别输出日志到文件和控制台，见关注点 1。动态变量 ${sys:user.home} 指向用户根目录，见关注点 2。对于 Windows 平台，示例如 C:\Users\Administrator，对于 Ubuntu 等 Linux 平台，示例如 /home/eric。日志输出格式见关注点 3。在匹配模式参数中，%d 为日期，%p 为日志级别，%c 为类名，数字指定打印类名长度，%t 为线程，%ex 是异常，%n 是换行标记，%m 是日志信息。输出级别是 DEBUG，见关注点 4。

4. Spring 配置

上述的数据库访问层已经用到 Spring 的配置，这里简述如下。ATUP 模块使用 Spring 的默认配置文件路径和名称：src/main/resources/applicationContext.xml。示例代码如下。

```
// 关注点 1: 自动扫描并加载的类所在包定义
```

```
<context:component-scan base-package="org.feuyeux.jaxrs2.atup"/>
<!-- JPA Transaction manager JPA -->
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  //关注点 2: 实体管理工厂和持久化单元定义
  <property name="persistenceUnitName" value="jpaMysql"/></bean>
<!-- TransAction Manager JPA -->
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  //关注点 3: 事务管理和实体管理工厂定义
  <property name="entityManagerFactory" ref="entityManagerFactory"/></bean>
  <tx:annotation-driven transaction-manager="transactionManager"/>//关注点 4: 事务管理定义
```

在这段代码中，关注点 1 定义了容器扫描，并自动加载其中的类的所属包名 org.feuyeux.jaxrs2.atup。关注点 2 定义了实体管理工厂接口 EntityManagerFactory 实例，使用的持久层配置是 jpaMysql，该配置参见前述的数据库配置。关注点 3 是配置事务管理实例 TransactionManager，使 EntityManagerFactory 实例生产的 EntityManager 接受容器级的事务的管理。关注点 4 启用注解方式的事务管理配置。

5. 封装 REST 客户端

REST 客户端的封装是为了满足 ATUP 模块之间的访问以及静态页面对 ATUP 模块提供的 REST 服务的访问。其中，服务器端模块之间的访问封装在核心模块 atup-core 的 AtupRequest 类中，页面的访问封装在通用脚本文件 atupRest 中。

（1）核心模块 atup-core

AtupRequest 类是客户端发起 REST 请求的封装类，包括对使用各种 HTTP 通用方法的封装、连接器和连接配置的封装等，示例代码片段如下。

```
public class AtupRequest<R, E> {//关注点 1: 请求包装类的泛型定义
public AtupRequest() {
    }
    public AtupRequest(final ClientConfig clientConfig) {
        this.clientConfig = clientConfig;
    }
    public void setClientRegisters(final Set<Class<?>> clientRegisters) {
        this.clientRegisters = clientRegisters;
    }
    public void proxy(String proxyUri, String proxyUserName, String proxyPassword) {
        if(this.clientConfig == null) {
            this.clientConfig = new ClientConfig();
        }
        this.clientConfig.property(ClientProperties.PROXY_URI, proxyUri);
        this.clientConfig.property(ClientProperties.PROXY_USERNAME, proxyUserName);
        this.clientConfig.property(ClientProperties.PROXY_PASSWORD, proxyPassword);
    }

    //timeout
    public void timeout(int connectTimeout, int readTimeout) {
        if(this.clientConfig == null) {
```

```
        this.clientConfig = new ClientConfig();
    }
    if (connectTimeout > 0) {
        this.clientConfig.property(ClientProperties.CONNECT_TIMEOUT, connectTimeout);
    }
    if (readTimeout > 0) {
        this.clientConfig.property(ClientProperties.READ_TIMEOUT, readTimeout);
    }
}
```

宅人坑事

lientProperties.CONNECT_TIMEOUT 和 ClientProperties.READ_TIMEOUT 两个属性接收的参数必须可以转化为 Integer 类型。默认值是 0，代表无限期，单位是毫秒。

在这段代码中，AtupRequest 定义了两个泛型参数：R 和 E，分别是返回值类型和请求实体类型，见关注点 1。该类主要处理两件事情，第一是封装客户端配置参数，主要包括对 ClientConfig 实例和 clientRegisters 的赋值。前者包括安全、超时等参数，后者包括 Provider 和资源类等注册类信息。第二是将配置参数赋值给 Client 实例，并处理请求方法和请求参数。

如下面代码片段所示，请求参数分别是请求方法名称、请求资源地址、请求头参数集合、请求参数集合、请求媒体类型、请求实体和返回类型。

```
public E rest(final String method, final String requestUrl,
final Set<AtupRequestParam> headParams, final Set<AtupRequestParam> queryParams,
final MediaType requestDataType, final R requestData, final Class<E> returnType) {
  if (clientConfig == null) {
      clientConfig = new ClientConfig();
  }
  final Client client = ClientBuilder.newClient(clientConfig);
  if (!CollectionUtils.isEmpty(clientRegisters)) {
      for (final Class<?> clazz : clientRegisters) {
          client.register(clazz);
      }
  }
  WebTarget webTarget = client.target(requestUrl);
  if (!CollectionUtils.isEmpty(queryParams)) {
       for (final AtupRequestParam atupRequestParam : queryParams) {
           webTarget = webTarget.queryParam(atupRequestParam.getKey(),
            atupRequestParam.getValue());
       }
  }
  final Invocation.Builder invocationBuilder = webTarget.request();
  if (!CollectionUtils.isEmpty(headParams)) {
       for (final AtupRequestParam atupRequestParam : headParams) {
           invocationBuilder.header(atupRequestParam.getKey(),
            atupRequestParam.getValue());
       }
  }
  javax.ws.rs.core.Response response = null;
```

```
        Entity<R> entity;
        switch (method) {
            case GET:
                response = invocationBuilder.get();
                break;
            case DELETE:
                response = invocationBuilder.delete();
                break;
            case PUT:
                entity = Entity.entity(requestData, requestDataType);
                response = invocationBuilder.put(entity);
                break;
            case POST:
                entity = Entity.entity(requestData, requestDataType);
                response = invocationBuilder.post(entity);
                break;
        }
        if (response != null) {
            return response.readEntity(returnType);
        } else {
            return null;
        }
    }
```

(2) 通用脚本 atupRest

atupRest.js 封装了浏览器端请求 REST 资源地址的函数，代码片段如下所示。

```
function restGet(restUrl, httpMethod, callback) {
    rest(restUrl, httpMethod, "", "application/json", "json", callback);
}
function restSet(restUrl, httpMethod, entity, callback) {
    rest(restUrl, httpMethod, entity, "application/json", "json", callback);
}
//关注点 1: rest() 方法定义
function rest(restUrl, httpMethod, entity, contentType, dataType, callback) {
    var resultLine = jQuery('#resultDiv');
    resultLine.html(LOADING);//关注点 2: 设定为加载状态
    var userId = storage.getItem("userId");//关注点 3: HTML5 的 storage 信息
    var userRole = storage.getItem("userRole");
    var request = jQuery.ajax({type: httpMethod, url: restUrl,
    headers: {'Atup-User': userId,'Atup-UserRole': userRole},
    data: entity, contentType: contentType, dataType: dataType,
crossDomain: true});
//关注点 4: 成功的回调
    request.done(function(data) {
    try {
            if (data === null || data === undefined) {
                resultLine.html(NO_RESULT);//关注点 5: 设定为无结果状态
            } else if (data.statusCode && data.statusCode != normal_status) {
                resultLine.html("Error:" + data.errorInfo);//关注点 6: 设定为错误状态
            } else if (callback != null) {
                resultLine.html("");//关注点 7: 清空设定状态
```

```
                callback(data);
            }
        } catch(e) {
            resultLine.html(e);// 关注点 8: 输出错误信息
        }
    });
// 关注点 9: 失败的回调
request.fail(function(textStatus, errorThrown) {
        resultLine.html(errorThrown + " status=" + textStatus.status + " text=" +
        textStatus.statusText);
    });
    resultLine.append(" DONE!");// 关注点 10: 设定为完成状态
}
```

在这段代码中，方法的参数分别是 REST 资源地址、请求方法名称、请求实体、请求头 contentType 类型、返回值类型和回调函数，见关注点 1。

ATUP 的纯 HTML 页面上都包含一个命名为 resultDiv 的 DIV 标签，用于动态显示 REST 请求的处理进程，在请求开始之前显示为“loading...”，请求处理结束后显示为“DONE!”，发生异常情况时，显示异常信息，分别见关注点 2、关注点 5 ~关注点 8 和关注点 10。

请求头包含 HTML5 的 storage 保存的当前用户 ID 和用户角色，见关注点 3，这部分在下文阐述。

请求是通过 jQuery 的 ajax() 函数实现的，需要额外说明的是，ATUP 各模块独立部署，因此需要设置跨域请求值 crossDomain 为 true。ajax() 函数提供了两个回调方法——done() 和 fail()，分别用于处理请求成功和请求异常，见关注点 4 和关注点 9。后者的处理是将错误信息显示在 resultDiv 上，前者要判断服务器端的处理是否有返回值，如果没有，data 为空，将没有返回值这一信息显示在 resultDiv 上；如果正常处理中包含错误代码（statusCode），处理同异常情况；否则调用参数中的回调函数。

调用 atupRest.js 封装的函数是 HTML 页面处理 REST 请求的唯一方式，这保证了统一的代码风格和处理方式，对安全、性能等环节做到了统一管理。这里给出一个调用示例，示例代码如下所示。

```
function updateUser() {
    var userName = jQuery.trim(jQuery("#userName").val());
    var password = jQuery("#password").val();
    var hashPassword = md5(password);
    var userRole = jQuery("#role").val();
    var putData = JSON.stringify(
    {userName: userName, passWord: hashPassword, userRole: userRole});
    restSet(HOST + ATUP_USER_BASE_URI + USER_PATH, PUT_METHOD, putData, renderUpdate);
}
function renderUpdate(data) {
    jQuery("#usersDiv").html(
    "<div><span style='width:100px;display:inline-block;'>User ID</span>");
...
```

```
    jQuery("#usersDiv").append(data.userId);
...
    jQuery("#usersDiv").append(data.userName);
...
    jQuery("#usersDiv").append("</span></div>");
}
```

在这段代码中，更新用户信息的 REST 请求调用了封装的方法 restSet()，并提供回调函数 renderUpdate()。updateUser() 方法首先从页面收集用户信息，然后将其转换为 JSON 格式的数据并调用 atupRest.js 中的 rest() 方法，回调函数 renderUpdate() 用于在页面上更新处理结果信息。

6. 测试作业分派

测试作业的创建过程可以理解为当前用户根据测试用例选择测试设备并提交到系统，系统将其排队的过程。测试作业的下发过程可以理解为系统根据优先级提取排队作业，如果其运行测试的设备状态空闲，即下发该作业，否则放弃出队操作。测试作业的下发可以并行执行。

前述的用户故事中定义了 3 种类型的作业。作为演示，本例所示的测试作业只涉及用户手动创建、系统排队即时下发的作业，没有涉及系统周期自动创建作业和系统定时下发作业。

（1）设备资源探测

如前所述，设备的状态包括：空闲（0）、忙碌（1）和死机（2）。只有其处于空闲时，才可以接收新的测试作业。因此，首先演示设备探测的处理。测试探测结果和下发时对设备状态的查询这一写一读两个线程使用数据库来同步。

设备探测是周期性的动作，可以使用系统的 Crontab 周期地触发 cURL 脚本来实现，cURL 访问一个一次性更新设备状态的 REST 资源地址。该资源地址也可以通过 Spring 配置的 Quartz 工具来调用。

另一种方式是通过编码完成周期性的动作，并公布其资源地址。可以在系统启动时触发，也可以由外部调用 cURL 脚本来触发。

本例使用后一种方式，示例代码如下，参考类为 org.feuyeux.jaxrs2.atup.device.service.StationDetectService。

```
detectTask.scheduleWithFixedDelay(new Runnable() {// 关注点 1: 定义周期性执行线程
    @Override
    public void run() {
        try {
            List<AtupDevice> deviceList = dao.findAll();
            for(final AtupDevice atupDevice : deviceList) {// 关注点 2: 遍历并探测设备
                final String detectPath = AtupApi.PROTOCOL +
                atupDevice.getDeviceHost() + ":" +
                AtupApi.SERVICE_PORT + AtupApi.SERVICE_PATH;
                final AtupRequest<String, Integer> request = new AtupRequest<>();
                // 关注点 3: 请求超时设置
                request.timeout(AtupVariable.DETECT_CONNECT_TIMEOUT, 0);
                try {
```

```
                final Integer result = request.rest(
                AtupRequest.GET, detectPath, Integer.class);
                log.debug("detecting " + atupDevice.getDeviceHost() + " :" + result);
                if (!result.equals(atupDevice.getDeviceStatus())) {
                    atupDevice.setDeviceStatus(result);
                    dao.update(atupDevice);
                }
            } catch (final Exception e) {
                log.error(e);
                if (!AtupParam.DEVICE_ERROR.equals(atupDevice.getDeviceStatus())) {
                    atupDevice.setDeviceStatus(AtupParam.DEVICE_ERROR);
                    dao.update(atupDevice);
                }
            }
        }
    } catch (final Exception e) {
        log.error(e);
    }
}
}, 0, AtupVariable.DETECT_INTERVAL, TimeUnit.SECONDS);// 关注点 4: 周期性任务的间隔设定
```

在这段代码中，detectTask 是 ScheduledExecutorService 接口的实现类，其方法 schedule WithFixedDelay() 用于接受一个周期执行的任务，其中定义了一个执行线程，见关注点 1。该线程用于遍历设备列表并探测其状态，见关注点 2。REST 请求连接超时设定为 AtupVariable. DETECT_CONNECT_TIMEOUT，类型为整型，见关注点 3。周期性执行的线程设定间隔为 AtupVariable.DETECT_INTERVAL，类型为长整型，单位是秒，见关注点 4。

可以通过以下命令触发探测流程：

```
curl -X POST http://localhost:8080/atup-device/rest-api/devices/status
```

（2）优先队列

在测试作业从生成到下发的过程中，测试作业按照优先级顺序缓存在优先队列中。ATUP 定义了 3 种优先级，分别是最高（HIGH = 0）、普通（MEDIUM = 1）和最低（LOW = 2）。优先队列的设计和实现有多种办法，ATUP 使用堆来实现。JDK 提供的 java.util. PriorityQueue<E> 内部使用数组实现了小根堆的数据结构，ATUP 选用 PriorityQueue 作为优先队列的基础。

小白讲堂

堆是一种经过排序的完全二叉树，其中任一非终端结点的数据值均不大于（或不小于）其左叶子和右叶子结点的值。最大堆和最小堆是二叉堆的两种形式。最大堆：根结点的键值是所有堆结点键值中最大者。最小堆：根结点的键值是所有堆结点键值中最小者。

有了优先队列后，要为其提供比较方法。PriorityQueue 内部支持 Comparator 接口和 Comparable 接口两种实现，优先使用 Comparator 接口的实现。Comparator 接口是一种策略模式的实现，即被比较的实体类本身无须具备比较能力，该类的比较类实现 Comparator 接口，并将其泛型类型定义为该实体类。Comparable 接口要求被比较的实体具备比较能力。两

者各有优缺点，前者简化了实体类的设计，后者简化了类的设计。测试作业类的设计实现了 Comparable 接口，示例代码如下所示。

```
@XmlRootElement
public class AtupTestJobInfo extends AtupInfo implements Comparable<AtupTestJobInfo> {
private Integer jobId;
private Integer userId;
private String deviceIp;
private Integer caseId;
private Integer priority;
private Long generatedTime;
...
    @Override
    public int compareTo(AtupTestJobInfo other) {
        int result = this.priority.compareTo(other.priority);
        if(result == 0) {
            return this.generatedTime.compareTo(other.generatedTime);
        } else {
            return result;
        }
    }
}
```

在这段代码中，首先比较测试作业的优先级，按照 Integer 的比较规则，正（负）数 1 为当前测试作业优先级高（低）。如果为 0，两者优先级相同，接下来比较测试作业生成时的时间，取纳秒数（nano time），这是个 Long 型数值，相等几率非常低。

有了理论依托，我们来设想一个场景，以检验代码实现是否正确。如图 11-6 所示，按照创建的时间先后列出了 5 个待下发的作业 J1 ~ J5，对应的测试用例编号为 C1 ~ C5，测试服务器包括 192.168.1.180 和 192.168.1.181 两台虚拟机，以及本地宿主机。作业优先级分别为高（H）、中（M）、低（L）。

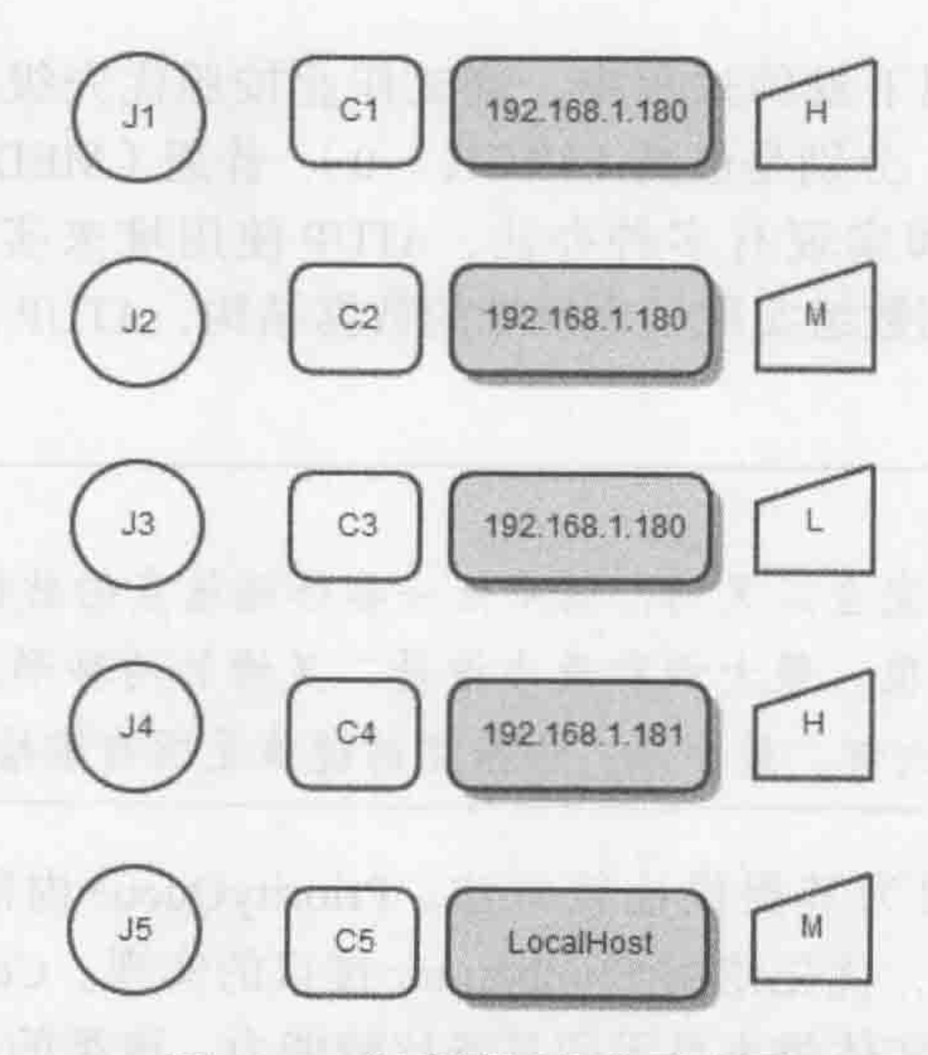


图 11-6 创建测试作业示意图

我们期待的堆结构如图 11-6 所示，优先级最高的是 J1 和 J4，而前者先于后者创建，因此排在前面，随后是 J2 和 J5，最后是 J3。并且，我们期待优先级最高的两个项出堆后，逻辑排列顺序是 J2-J5-J3，如图 11-7 所示。

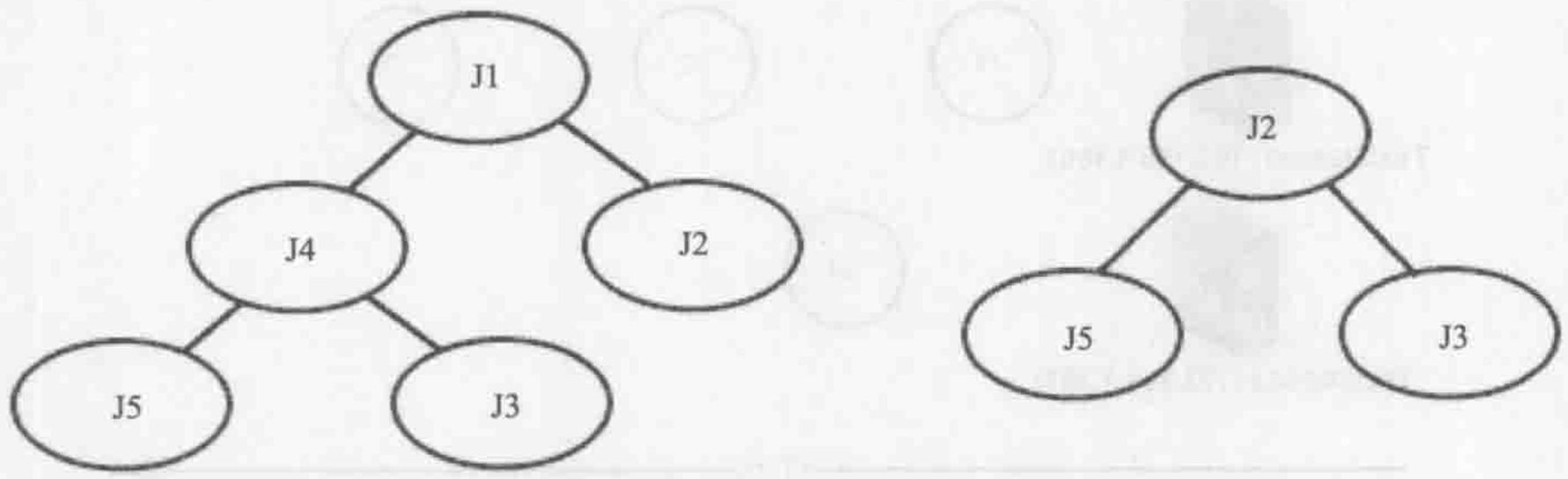


图 11-7　测试作业小根堆示意图

验证部分可以参考 JobLaunchService 类，该类是封装和调用测试作业队列 PriorityQueue 实例的业务类，包括增加和删除测试作业、查看当前待下发作业信息和下发作业等功能。其中，获取当前队列的待下发作业信息的实质是对 PriorityQueue 实例内部数组的遍历，但这个说法其实是不正确的，因为堆的内部结构和算法非严格一致，即当前出队的作业的优先级取值永远最小（代表优先级最高），但在其内部数组中的顺序不是照此安排的。可参见 PriorityQueue 类的 siftUp() 方法和 siftDown() 方法。因此，查看当前待下发作业信息的实现要额外对获取的数组进行一次排序，这样获取的列表顺序才和优先级逻辑一致。示例代码如下所示。

```
public AtupTestJobListInfo getJobs() {
        AtupTestJobInfo[] jobs = jobQueue.toArray(new AtupTestJobInfo[jobQueue.size()]);
        if (jobs.length > 0) {
            Arrays.sort(jobs);
        }
        return new AtupTestJobListInfo(Arrays.asList(jobs));
    }
```

（3）并发下发

按照设备状态和优先队列的讲述，我们可以设想测试作业是按照如图 11-8 所示的顺序下发并返回测试结果的。

但是，图 11-8 的这个设想基本上是错误的，除非我们使用串行的下发方式，即一个接一个地下发。但这在实际生产中是不可能的，我们需要谨慎地考虑并发问题。使用 PriorityQueue 类非常方便地实现堆结构的时候，切不可太过随意，因为这个类不是线程安全的，并且不建议在多线程中使用。一个很典型的并发问题是下发操作会对同一个作业重复下发或者同时向一个设备下发多个作业导致设备忙碌，测试作业有丢失的可能。另外，优先队列的插入操作和删除操作是通过页面完成的，和下发出队操作不在一个线程中，也需要考虑。

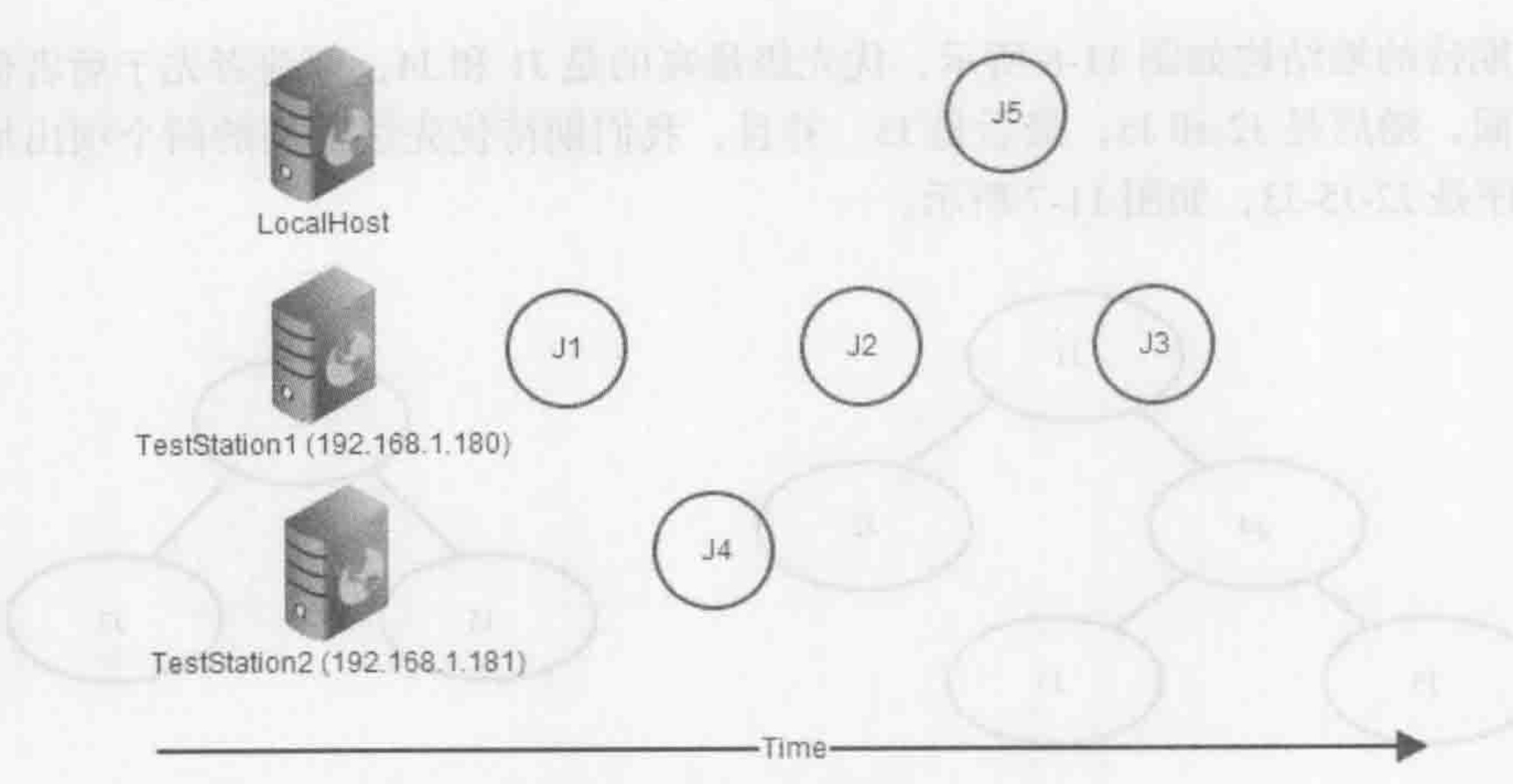


图 11-8 测试作业下发顺序示意图

解决并发问题的核心是锁的使用和锁粒度的掌握。如果粒度过大，会导致串行，甚至出现互锁导致的死锁。粒度过小，就无法保证业务完整地按顺序执行。在下发方法中，应当针对设备 IP 锁定，保证同一设备测试作业的串行（测试设备运行测试后状态为忙碌，应当等待其空闲时再下发新的测试作业）。代码示例如下所示。

```
synchronized(jobQueue) {// 关注点 1: 锁 jobQueue
  jobInfo = jobQueue.poll();
  deviceIp = jobInfo.getDeviceIp();
...
}
if(jobInfo != null) {
  synchronized(lockMap.get(deviceIp)) {
      if(deviceIp.matches(IP_PATTERN)
      || deviceIp.toUpperCase().equals("LOCALHOST")) {
          final AtupDevice testDevice = fetchDeviceFromDB(deviceIp);
          Integer deviceStatus = testDevice.getDeviceStatus();
          if(AtupParam.DEVICE_IDLE.equals(deviceStatus)) {
              final AtupTestCase testCase = fetchTestCaseFromDB(jobInfo.getCaseId());
              launchTest(jobInfo.getUserId(), testDevice, testCase);
              return true;
          }
      }
  }
}
```

在这段代码中，首先是控制锁力度，见关注点 1，这里只将与作业队列操作相关的部分锁住，在没有涉及优先队列操作的地方无须将锁粒度定为 jobQueue。另外，如果当前测试作业出堆后，没有满足条件或者异常导致下发失败，应当将其重新入堆，确保逻辑上事务的原子性。详见 org.feuyeux.jaxrs2.atup.cases.service.JobLaunchService 类。

（4）异步响应

本部分是对第 8 章的实践，原理可参考该章。业务逻辑是下发过程中，ATUP CASE 服务器无须等待 ATUPTest Station 测试服务器运行测试即可返回下发结果给客户端，待 ATUPTest Station 测试完毕，异步返回测试结果，ATUP CASE 服务器将测试用例的测试结果写入数据库。详见 org.feuyeux.jaxrs2.atup.cases.service.JobLaunchService 类。

（5）作业监控

测试作业的监控对测试者和作业管理员都非常有价值，其核心价值是时效性和准确性。这部分的实现是通过 JavaScript 脚本的轮询请求资源列表的 REST 资源地址实现的，代码示例如下所示。

```
function loadDevices() {
...
    setInterval(restCall, 10000);
}
function restCall() {
    restGet(HOST + ATUP_CASE_BASE_URI + TEST_JOB_PATH, GET_METHOD, renderGetAll);
}
```

7. 持续发布

持续发布的两个关键参数是服务器地址和发布环境的配置。前者是对 Nginx 和 Tomcat 服务器的定位，后者是对前述 3 套环境中相关参数的赋值。现在的持续发布形式是一键式的，即动态设置参数以匹配响应的部署环境，因此我们需要定位一个环节来动态地为上述两组参数赋值。构建工具和集成测试平台都可以做这件事情。ATUP 选用的是构建工具 Maven。本节将列举几个关键点，来演示如何实现动态配置。

（1）服务器地址

服务器是指 ATUP 各个模块部署的机器。在不同的环境中，机器的地址是不同的。例如，在开发环境中，各模块都部署在本地，因此服务器地址就是 localhost，端口（通常情况下）就是 8080。集成测试环境中，采用了动静分离；生产环境中，各个模块单独部署。因此，服务器的地址不能硬编码，也不能由某个配置文件实现。

ATUP 采用的解决方案是为每一种环境提供独立的配置，比如集成测试环境的配置文件为 atupCommon_staging.js，在 Maven 构建过程中，如果目标部署环境是 CI 的，就将其替换为 atupCommon.js。这样运行期就是用这套环境的服务器地址。这里要用到的 Maven 插件是 maven-antrun-plugin，相关配置在 jax-rs2-atup\atup-page\pom.xml 中，参考代码如下所示。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.7</version>
  <executions>
      <execution>
          <phase>initialize</phase>
```

```
            <configuration>
                <target>
                    <copy file="src/main/webapp/js/atupCommon_staging.js"
                    tofile="${project.build.directory}/${project.artifactId}-
                    ${project.version}/js/atupCommon.js" overwrite="true" />
                </target>
            </configuration>
            <goals>
               <goal>run</goal>
            </goals>
        </execution>
      </executions>
    </plugin>
```

jax-rs2-atup\atup-page\src\main\webapp\js\atupCommon_staging.js 的内容参考如下代码。

```
var ATUP_CASE_HOST = "http://10.11.72.57:8080/";
var ATUP_DEVICE_HOST = "http://10.11.72.57:8080/";
var ATUP_USER_HOST = "http://10.11.72.57:8080/";
var ATUP_PAGE_HOST = "http://10.11.72.57/";
var ATUP_CASE_BASE_URI = 'atup-case/rest-api';
var ATUP_DEVICE_BASE_URI = 'atup-device/rest-api';
var ATUP_USER_BASE_URI = 'atup-user/rest-api';
var ATUP_PAGE_BASE_URI = 'atup-page/';
var ATUP_CASE_URI = ATUP_CASE_HOST+ATUP_CASE_BASE_URI;
var ATUP_DEVICE_URI = ATUP_DEVICE_HOST+ATUP_DEVICE_BASE_URI;
var ATUP_USER_URI = ATUP_USER_HOST+ATUP_USER_BASE_URI;
var ATUP_PAGE_URI =ATUP_PAGE_HOST +ATUP_PAGE_BASE_URI;
```

本例是比较简单的多个模块部署在同一台主机上的情况，分模块（独立）部署见 11.3.4 节。

（2）动态模块热部署

Maven 插件 tomcat7-maven-plugin 支持 Tomcat 的热部署，在构建成功后执行，实现远程复制 war 包并热部署该应用。示例参考如下，在 jax-rs2-atup\pom.xml 中，定义 Tomcat 服务器地址和端口。

```
<properties>
  <tomcat.server.ip>192.168.1.180</tomcat.server.ip>
  <tomcat.local.port>8080</tomcat.local.port>
</properties>
```

在插件使用的配置中，提供上述参数，以实现对远程服务器的访问，示例代码如下所示。

```
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>${tomcat7-maven-plugin.version}</version>
  <configuration>
      <url>http://${tomcat.server.ip}:${tomcat.local.port}/manager/text</url>
      <server>Atup_Tomcat_Staging</server>
  </configuration>
</plugin>
```

在子模块的配置中，比如 jax-rs2-atup\atup-device\pom.xml，具体定义触发该插件执行的阶段（pre-integration-test）和复制 war 的名称和路径，默认为 target/${project.build.finalName}.war 和 /${project.build.finalName}。示例代码如下所示。

```
<profile>
<id>CI</id>
<build>
  <plugins>
      <plugin>
          <groupId>org.apache.tomcat.maven</groupId>
          <artifactId>tomcat7-maven-plugin</artifactId>
          <executions>
              <execution>
                  <phase>pre-integration-test</phase>
                  <configuration>
                      <warFile>target/${project.build.finalName}.war</warFile>
                      <path>/${project.build.finalName}</path>
              </configuration>
              <goals>
                  <goal>redeploy</goal>
              </goals>
          </execution>
   </executions>
  </plugin>
</plugins>
</build>
</profile>
```

构建 ATUP 的主机需要添加 Maven 配置中的服务器的用户信息。.m2/settings.xml 文件中的参考配置如下所示。

```
<servers>
  <server>
      <id>Atup_Tomcat_Staging</id>
      <username>admin</username>
      <password>admin</password>
  </server>
</servers>
```

同时，在 Tomcat 中需要建立该用户信息。TOMCAT_HOME/conf/tomcat-users.xml 文件中的参考配置如下所示。

```
<?xml version=' 1.0' encoding=' utf-8' ?>
<tomcat-users>
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<user username="admin" password="admin" roles="manager-script,manager-gui"/>
</tomcat-users>
```

执行动态模块的部署，可以使用 Maven 的构建命令，示例如下所示。

```
mvn clean install -DskipTests -Dtomcat.server.ip=192.168.1.181 -PCI
```

其中，-D 后跟随构建中用到的变量键值对，-P 后跟随 Maven 配置中 profile 名称。

(3) 静态模块热部署

因为在静态服务器中没有虚拟机的概念，所以静态模块部署的实质是文件复制。基本思路是使用 Maven 的 Ant 插件调用远程复制命令 scp，示例代码如下所示。

```
<profile>
  <id>CI</id>
  <build>
      <plugins>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-antrun-plugin</artifactId>
              <version>1.7</version>
              <executions>
                  <execution>
                      <id>deploy package</id>
                      <phase>package</phase>
                      <configuration>
                          <target>
                              <scp todir="${u}:${p}@${h}:/${d}"
                              trust="true" failonerror="true">
                              <fileset
                              dir="${project.build.directory}/
                              ${project.artifactId}-${project.version}" />
                              </scp>
                          </target>
                      </configuration>
                      <goals>
                          <goal>run</goal>
                      </goals>
                  </execution>
              </executions>
              <dependencies>
                      <dependency>
                          <groupId>org.apache.ant</groupId>
                          <artifactId>ant-jsch</artifactId>
                          <version>1.9.2</version>
                      </dependency>
                      <dependency>
                          <groupId>com.jcraft</groupId>
                          <artifactId>jsch</artifactId>
                          <version>0.1.50</version>
                      </dependency>
                  </dependencies>
              </plugin>
          </plugins>
      </build>
      <properties>
          <h>${nginx.server.ip}</h>
```

```
            <u>erichan</u>
            <p>han</p>
            <d>/usr/share/nginx/html/atup-page</d>
  </properties>
</profile>
```

执行静态部署的基本命令如下。

```
mvn clean install -DskipTests -PCI
```

基本命令很难满足多样的部署环境，因此，执行命令中为参数赋值是更可取的办法，这样做的好处是，避免每次切换部署环境时，修改 Maven 配置文件。命令参考示例如下所示。

```
mvn clean install -DskipTests -Du=eric -Dp=han -Dd=/home/eric/nginx/atup-page/ -PCI
```

（4）数据库部署

Maven 插件 sql-maven-plugin 为构建期间操作远程数据库提供了可能。在 ATUP 到根配置文件 jax-rs2-atup\pom.xml 中定义 MySQL 服务器的访问信息，参考代码如下。

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId>
  <version>${sql-maven-plugin.version}</version>
  <dependencies>
      <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>${mysql-connector.version}</version>
      </dependency>
  </dependencies>
  <configuration>
      <driver>com.mysql.jdbc.Driver</driver>
      <url>jdbc:mysql://${mysql.server.ip}:3306</url>
      <username>root</username>
      <password>root</password>
  </configuration>
</plugin>
```

子模块的配置文件 modules/pom.xml 中定义了触发插件执行的阶段（pre-integration-test）和触发操作（执行见表脚本 atup-user.sql）。

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId>
  <executions>
      <execution>
         <id>create-db</id>
         <phase>pre-integration-test</phase>
         <goals>
              <goal>execute</goal>
         </goals>
```

```
            <configuration>
                <srcFiles>
                    <srcFile>../document/atup-ddl/atup-user.sql</srcFile>
                </srcFiles>
            </configuration>
        </execution>
    </executions>
</plugin>
```

数据库的部署命令参考如下。

```
mvn clean install -DskipTests -Dmysql.server.ip=10.11.72.57 -PCI
```

数据库的部署通常是在 CI 环境中使用，因为开发环境使用本地数据库，其测试数据不宜每次都删除。单元测试通常使用内存数据库，无须 MySQL。生产环境中，部署环节通常不应该对数据库进行减法操作，加法操作也相对谨慎。

（5）发布版本号

发布版本号是每次构建和部署的唯一标识，在 CI 测试阶段可以有效落实缺陷所处的版本，对于 UI 测试尤为重要。如何为每次构建和部署生成唯一标识呢？这里使用了 Maven 的 replacer 插件，动态将构建时间赋值给指定的 token。根配置 jax-rs2-atup\pom.xml 参考如下，这里定义了根目录路径。

```
<properties>
<main.basedir>${project.basedir}</main.basedir>
</properties>
```

子模块配置 jax-rs2-atup\atup-page\pom.xml 参考如下，定义了根目录路径和覆盖 token 的实现。

```
<properties>
  <main.basedir>${project.parent.basedir}</main.basedir>
</properties>
...
<profile>
  <id>CI</id>
  <build>
      <plugins>
          <plugin>
              <groupId>com.google.code.maven-replacer-plugin</groupId>
              <artifactId>replacer</artifactId>
              <version>1.5.2</version>
              <executions>
                  <execution>
                      <phase>validate</phase>
                      <goals>
                          <goal>replace</goal>
                      </goals>
                  </execution>
              </executions>
              <configuration>
                  <file>
```

```
                ${main.basedir}/atup-page/src/main/
                webapp/js/index_buildtime.js
                </file>
                <outputFile>
                ${main.basedir}/atup-page/src/main/webapp/js/index.js
                </outputFile>
                <replacements>
                    <replacement>
                        <token>@buildtime@</token>
                        <value>${maven.build.timestamp}</value>
                    </replacement>
                </replacements>
            </configuration>
        </plugin>
    </plugins>
  </build>
</profile>
```

jax-rs2-atup\atup-page\src\main\webapp\js\index_buildtime.js 中定义了名为 buildtime 的 token，在执行期替换为构建时间 ${maven.build.timestamp} 变量，然后将该文件内容覆盖，输出到 ${main.basedir}/atup-page/src/main/webapp/js/index.js 文件，这个文件是最终运行期使用的脚本文件。

```
function markVersion() {
    jQuery('#buildDiv').html("Build Time: @buildtime@");
}
```

(6) 数据库连接

如果不使用 JNDI 配置，那么数据库的配置信息就会公布在程序中，这样一方面不符合安全的要求，另一方面很难支持多种环境的部署。这里使用 Maven 的 properties-maven-plugin 插件实现为配置文件中的变量动态赋值。

在 JPA 的配置文件 jax-rs2-atup\atup-user\src\main\resources\META-INF\persistence.xml 的 JDBC 配置中启用参数。

```
<!-- Connection JDBC -->
<properties>
<property name="javax.persistence.jdbc.driver" value="${db.driver}"/>
<property name="javax.persistence.jdbc.url" value="${db.url}"/>
<property name="javax.persistence.jdbc.user" value="${db.username}"/>
<property name="javax.persistence.jdbc.password" value="${db.password}"/>
</properties>
```

创建参数键值对配置文件 jax-rs2-atup\atup-user\src\main\resources\section\staging.properties。

```
db.server.address=10.11.72.54
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://${db.server.address}:3306/jaxrs2_atup
db.username=root
db.password=root
```

编辑 Mavne 配置文件 jax-rs2-atup\atup-user\pom.xml，执行插件的 read-project-properties，将其目标定义为上述的键值对文件，这样构建后的 JPA 配置中就会将参数键替换成参数值。

```
<profile>
  <id>CI</id>
  <build>
     <plugins>
          <plugin>
               <groupId>org.codehaus.mojo</groupId>
               <artifactId>properties-maven-plugin</artifactId>
               <version>1.0-alpha-2</version>
               <executions>
                  <execution>
                      <phase>initialize</phase>
                      <goals>
                          <goal>read-project-properties</goal>
                      </goals>
                      <configuration>
                          <files>
                              <file>
                              src/main/resources/section/staging.properties

                              </file>
                          </files>
                      </configuration>
                  </execution>
               </executions>
          </plugin>
     </plugins>
  </build>
</profile>
```

(7) 单元测试和集成测试

单元测试代码使用 TU 前缀，对方法功能进行测试。集成测试前缀是 TI。示例如下所示。

```
@ContextConfiguration(locations = {"classpath:applicationContext2.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TUAtupUserDao {
    private final Logger log = LogManager.getLogger(TUAtupUserDao.class.getName());
    @Autowired
    private AtupUserDao dao;
    @Test
    public void testCreateUser() {
        final AtupUser user = CreateUser.buildUser();
        final AtupUser newUser = dao.save(user);
        log.info(newUser);
        Assert.assertEquals(user.getUserName(), newUser.getUserName());
    }
```

这段代码使用了 applicationContext2.xml 文件作为 Spring 的环境配置，和运行环境的主要区别是没有使用容器配置的 JNDI 作为 JPA 的数据集，而是采用本地配置 jpaMysql2。

在 Maven 的配置中，为测试建立名为 TI 的 profile，参考代码如下。

```
<profile>
  <id>TI</id>
  <build>
      <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>${maven-failsafe-plugin.version}</version>
                <configuration>
                    <includes>
                        <include>**/**/**/**/**/TU**.java</include>
                        <include>**/**/**/**/**/TI**.java</include>
                    </includes>
                </configuration>
                <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
      </plugins>
  </build>
</profile>
```

11.3.2　Sprint2 模块功能

在完成基础功能后，我们进入下一个 Sprint 来实现各个模块的功能。下面逐一讲述实现用户模块、设备模块和用例模块过程中的要点。

1. 用户模块

ATUP 采用动静分离的多模块单机部署拓扑，默认情况下多模块之间的会话信息无法共享。因此，本示例采用的策略是在用户登录后，静态服务器上的脚本将用户的角色信息保存在浏览器端，使用的是 HTML5 的 Web Storage 技术。

小白讲堂

HTML5 Web Storage 相比 Cookie 更安全和快速。数据以键值对形式存储，支持存储较大的数据，而不影响性能。这比 Cookie 只有 4kB 容量的限制要强大许多。Web Storage 包括两种类型，其中 localStorage 用于存储不过期的数据，在浏览器关闭后，数据依然有效；sessionStorage 用于存储会话生命周期的数据。Firefox、Chrome、Internet Explorer 8+、Safari 和 Opera 浏览器都支持 Web Storage。

---

(1) 注册和登录

ATUP 的注册页面是 atup-page/src/main/webapp/signUp.html，对应的脚本文件是 atup-page/src/main/webapp/js/users/user.js，用于管理员创建普通用户。代码片段如下，页面部分的处理方式遵循上文讲述的 REST 请求。

```
<input id="createButton" type="button" value="Create" onclick="createUser();"/>
function createUser() {
    var userName = jQuery.trim(jQuery("#userName0").val());
    var password = jQuery("#password0").val();
    var hashPassword = md5(password);
    var postData = JSON.stringify(
    {userName: userName, passWord: hashPassword, userRole: 4, status: 0});
    restSet(HOST + ATUP_USER_BASE_URI + USER_PATH, POST_METHOD, postData, renderCreate);
}
```

ATUP 注册用户登录页面是 atup-page/src/main/webapp/signIn.html，对应的脚本文件是 atup-page/src/main/webapp/js/users/user.js；首页是 atup-page/src/main/webapp/index.html，对应的脚本文件是 atup-page/src/main/webapp/js/index_buildtime.js。登录页面用于请求服务器验证当前用户信息，并保存用户 ID 到 HTML5 的 sessionStorage 中，然后根据用户权限动态生成首页信息。示例代码片段如下所示。

```
// 关注点 1: 登录按钮
<input id="signInButton" type="button" value="Sign In" onclick="signIn();"/>

// 关注点 2: 登录方法
function signIn() {
    var userName = jQuery("#userName").val();
    var password = jQuery.trim(jQuery("#password").val());
    var hashPassword = md5(password);
    var url = HOST + ATUP_USER_BASE_URI + SIGNIN_PATH
    + "?user=" + userName + "&password=" + hashPassword;
    restGet(url, GET_METHOD, renderSignIn);
}
// 关注点 3: 登录回调方法
function renderSignIn(data) {
    if(data.userId != null && data.userName != null && data.userRole != null) {
        storage.setItem("userId", data.userId);
        storage.setItem("userName", data.userName);
        storage.setItem("userRole", data.userRole);
        window.location.href = "index.html";
    }
// 关注点 4: 按角色渲染页面方法
function loadLinkByRole() {
    var userRole = storage.getItem("userRole");
    if(userRole == ROLE_ADMIN) {
        jQuery('#userDiv').html(
            "<div><h5>Users</h5></div>" +
                "<a href='users/users.html'>User List</a><br/>" +
                "<a href='users/user.html'>Manage User</a><br/>");
```

```
    } else if (userRole == ROLE_JOB_KILLER) {
...
    } else if (userRole == ROLE_DEVICE_MANAGER) {
...
    } else if (userRole == ROLE_USER) {
...
    }
}
```

在这段代码中，分别定义了登录按钮、登录方法、登录回调方法和按角色渲染页面方法，分别见关注点 1 ~ 关注点 4。这里要说明的是，REST 请求最终进入 rest() 方法后所用到的用户 ID 和用户角色信息是在用户登录成功的回调函数 renderSignIn() 中保存的。另外，用户名称用于页面显示。

(2) 认证和授权

静态页面对用户授权是通过动态生成符合当前角色的 HTML 内容实现的，而认证是通过页面加载时调用脚本函数完成的，示例代码如下所示。

```
function checkSignIn() {
    var storageUserId = storage.getItem("userId");
    if (storageUserId == null) {
        window.location.href = HOST + ATUP_PAGE_BASE_URI + "signIn.html";
    } else {
        var user = storage.getItem("userName");
        var welcomeDiv = jQuery('#topDiv').html("Welcome " + user);
    }
}
```

在第 3 章中，我们讲述过使用 @Context 注解获取请求头信息，createUser() 方法中可以得到脚本函数 rest() 提供的用户 ID 和用户角色信息。由头信息得到当前用户的角色，然后根据角色决定当前资源是否可以被其使用。如果没有用户角色信息或者用户角色不符合要求，那么将遵循第 3 章所述的处理响应的要求，直接返回错误信息。

服务器端的认证是通过对请求头信息处理来实现的，授权是通过编码完成的，示例代码如下所示。

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public javax.ws.rs.core.Response createUser(
  @Context final HttpHeaders headers, final AtupUserInfo userInfo) {
  Response response = checkRole(headers, AtupParam.USER_ADMIN);
  if (response == null) {
      return createUser(userInfo);
  } else {
      return response;
  }
}
public Response checkRole(HttpHeaders headers, Integer role) {
  Integer userId = Integer.valueOf(
```

```
headers.getRequestHeader(AtupApi.ATUP_USER_HEAD).get(0);
Integer userRole = Integer.valueOf(
headers.getRequestHeader(AtupApi.ATUP_USER_ROLE_HEAD).get(0);
if(userId == null) {
      final AtupUserInfo result =
      new AtupUserInfo("No user info found.", AtupErrorCode.UNAUTHORIZED_ERROR);
      return Response.status(
      Response.Status.INTERNAL_SERVER_ERROR).entity(result).build();
  } else {
      final AtupUser user = service.getUser(userId);
      if(!user.getUserRole().equals(role) || !userRole.equals(role)) {
            final AtupUserInfo result =
            new AtupUserInfo("No permission for this request.",
            AtupErrorCode.FORBIDDEN_ERROR);
            return Response.status(
            Response.Status.INTERNAL_SERVER_ERROR).entity(result).build();
      } else {
            return null;
      }
  }
}
```

(3) Web Storage 与多 Tab

通常，浏览器端存储让开发者忌惮的是多 Tab 对会话的影响，这里要明确的是，sessionStorage 的作用域是当前 Tab。如果新打开一个 Tab 并复制前一个 Tab 登录后访问的地址，新 Tab 页面会跳转到登录页面，这正是我们期待的。使用 Chrome 浏览器试验如下。

在第一个 Tab 中，启用开发者工具（快捷键是 <F12>）查看 sessionStorage，因为该页面已经登录，所以会有当前会话的用户信息，如图 11-9 所示。

打开第二个 Tab，输入 http://localhost:8080/atup-page，页面自动跳转到登录页面，查看该 Tab 页面的 sessionStorage，发现内容为空，如图 11-10 所示。

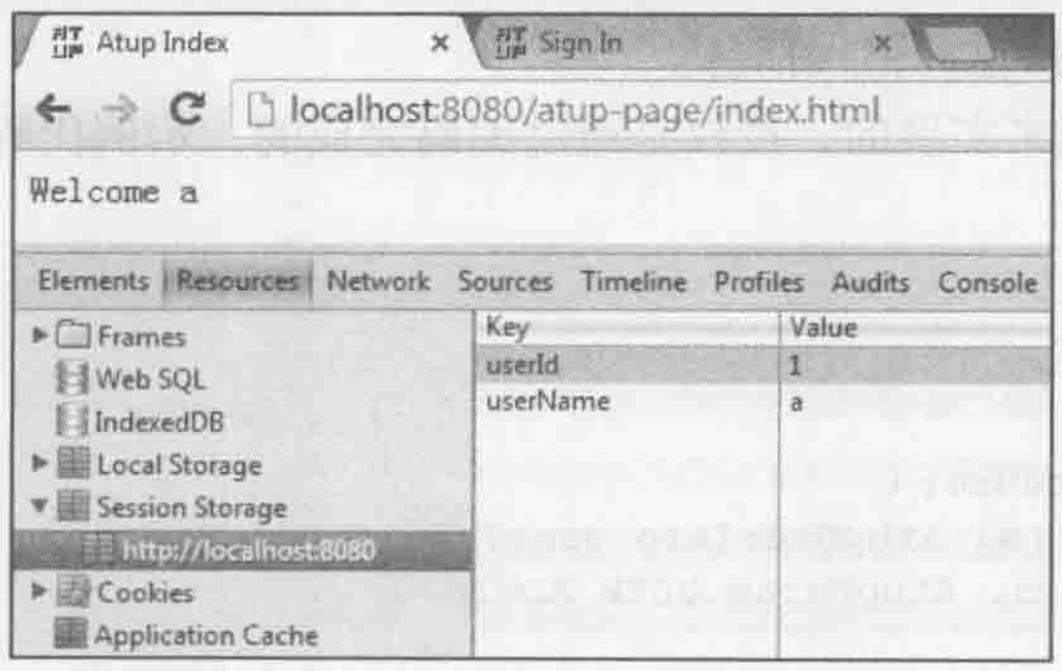


图 11-9 Chrome 开发者工具中的 Session Storage 示意图 1

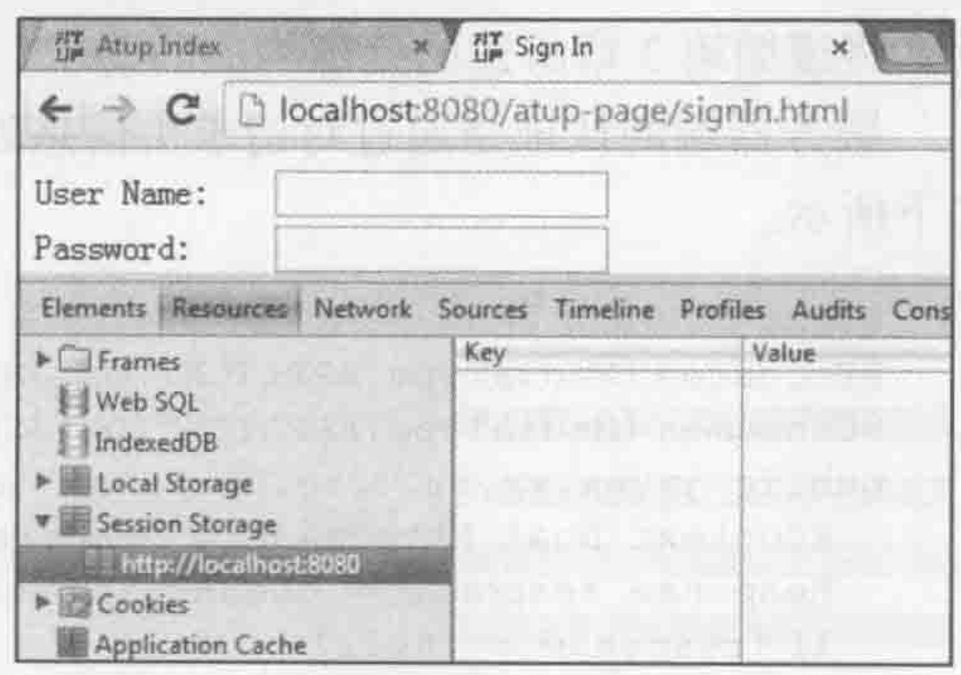


图 11-10 Chrome 开发者工具中的 Session Storage 示意图 2

(4) 跨域访问

因为 ATUP 是动静分离，但模块分别部署，所以另一个要考虑的问题是跨域访问。因

此，需要在 ATUP 的核心模块中实现跨域设置。AtupCrossDomainFilter 类分别实现了 ContainerRequestFilter 和 ContainerResponseFilter 接口，而且在 filter() 方法的实现中，对响应头进行跨域相关的头字段进行填充。示例代码如下所示。

```
@Override
public void filter(final ContainerRequestContext requestContext,
final ContainerResponseContext responseContext) throws IOException {
  responseContext.getHeaders().add("Access-Control-Allow-Origin", "*");
  responseContext.getHeaders().add("Access-Control-Allow-Headers",
  "origin, content-type, accept, authorization, Atup-User, Atup-UserRole");
  responseContext.getHeaders().add("Access-Control-Allow-Credentials", "true");
  responseContext.getHeaders().add("Access-Control-Allow-Methods",
  "GET, POST, PUT, DELETE, OPTIONS, HEAD");
  responseContext.getHeaders().add("Access-Control-Max-Age", "1209600");
}
```

模块的 * Application 中需要注册该类，示例代码如下所示。

```
register(org.feuyeux.jaxrs2.atup.core.util.AtupCrossDomainFilter.class);
```

同时，Nginx 的配置也要考虑跨域访问。

```
add_header 'Access-Control-Allow-Origin' '*';
add_header 'Access-Control-Allow-Credentials' 'true';
add_header 'Access-Control-Allow-Headers'
'origin,Content-Type,Accept,Atup-User,Atup-UserRole';
add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, HEAD';
add_header 'Access-Control-Max-Age' '1209600';
```

2. 设备模块

设备模块的实现需要考虑两个细节，即纯 HTML 页面间参数的传递和页面定时刷新设备状态。

(1) 设备管理页面传参

在设备管理脚本 jax-rs2-atup\atup-page\src\main\webapp\js\devices\device.js 中，首先会处理待管理设备的传入参数，示例代码如下所示。

```
if(query) {
   jQuery("#deviceIp").val(getValue(query, "deviceIp"));
   jQuery("#deviceName").val(getValue(query, "deviceName"));
   jQuery("#deviceType").val(getValue(query, "deviceType"));
   jQuery("#deviceStatus").val(getValue(query, "deviceStatus"));
}
```

这样做的好处是，方便更新设备的所有者。

(2) 设备列表页面轮询

创建设备时，有一个小的技巧就是将设备状态设为 ERROR，这样做的目的在于让后台探测线程去决定一个设备的正确状态。11.3.1 节的测试作业分派中讲述了探测的实现。

设备列表页面定时对列出的所有者拥有的设备进行状态的探测，设备管理员角色可以查看全部设备的状态等信息。

3. 用例模块

用例模块的重点是将静态的测试用例和与之相关的运行期测试作业之间有机地结合，并在测试设备上完成对测试用例的测试。

(1) 测试用例页面的测试集列表

测试用例必须属于某个测试集，因此其管理页面在加载时会将已有测试集以 select 列表标签组织，示例代码如下所示。

```
function initial() {
    restGet (ATUP_CASE_URI + TEST_SUITE_PATH
    + "/suites?start=0&size=100", GET_METHOD, renderSuiteList0);
}
function renderSuiteList0 (data) {
    var html = renderSuites (data);
    jQuery ("#suiteList0") .empty().append (html);
}
function renderSuites (data) {
var list = data.suiteList;
var html = "";
    jQuery.each (list, function (i, suite) {
html += "<option value=\"";
html += suite.suiteId + "\">";
html += suite.suiteName + "</option>";
    });
return html;
}
```

(2) 测试作业页面的设备列表

测试作业的创建依赖于测试设备，原理同上面的测试集列表，示例代码如下所示。

```
function initial() {
    restGet (ATUP_DEVICE_URI + DEVICE_PATH, GET_METHOD, renderDeviceList);
}
function renderDeviceList (data) {
var list = data.deviceList;
var html = "";
    jQuery.each (list, function (i, device) {
html += "<option value=\"";
html += device.deviceHost + "\">";
html += device.deviceName + "</option>";
    });
    jQuery ("#deviceList") .empty().append (html);
}
```

(3) 测试作业列表页面轮询

测试作业列表页面定时从作业队列中查询待处理作业情况，以便监控作业下发线程的工作状态，示例代码如下所示。

```
function loadJobs() {
...
    setInterval(restCall, 10000);
}
function renderGetAll(data) {
var list = data.jobs;
    if(list == null || list.length == 0) {
        jQuery("#jobsDiv").html("No job is arranging.");
    } else {
        var userRole = storage.getItem("userRole");
        if(ROLE_JOB_KILLER == userRole) {
            jQuery("#jobsDiv").html(
                "<div align='left'>" + SPAN_BEGIN
                    + "Job Id</span>" + SPAN_BEGIN
                    + "Case Id</span>" + SPAN_BEGIN2
                    + "Device Ip</span>" + SPAN_BEGIN
                    + "User Id</span>" + SPAN_BEGIN2
                    + "Job Priority</span>" + SPAN_BEGIN
                    + "Create Time</span>" + SPAN_BEGIN
                    + "Manage</span></div>");
            jQuery.each(list, function(i, jobInfo) {
            var line = "<div align='left'>" + SPAN_BEGIN
                    + jobInfo.jobId + "</span>" + SPAN_BEGIN
                    + jobInfo.caseId + "</span>" + SPAN_BEGIN2
                    + jobInfo.deviceIp + "</span>" + SPAN_BEGIN
                    + jobInfo.userId + "</span>" + SPAN_BEGIN2
                    + jobInfo.priority + "</span>" + SPAN_BEGIN
                    + jobInfo.generatedTime + "</span>" + SPAN_BEGIN
                    + "<input type='button' value='REMOVE'
                   onclick='removeJob(" + jobInfo.jobId + ");'/></span></div>";
    jQuery("#jobsDiv").append(line);
            });
        } else {
            jQuery("#jobsDiv").html(
                "<div align='left'>" + SPAN_BEGIN
                    + "Job Id</span>" + SPAN_BEGIN
                    + "Case Id</span>" + SPAN_BEGIN
                    + "Device Ip</span>" + SPAN_BEGIN
                    + "User Id</span>" + SPAN_BEGIN2
                    + "Job Priority</span>" + SPAN_BEGIN
                    + "Create Time</span></div>");
            jQuery.each(list, function(i, jobInfo) {
var line = "<div align='left'>" + SPAN_BEGIN
                    + jobInfo.jobId + "</span>" + SPAN_BEGIN
                    + jobInfo.caseId + "</span>" + SPAN_BEGIN
                    + jobInfo.deviceIp + "</span>" + SPAN_BEGIN
                    + jobInfo.userId + "</span>" + SPAN_BEGIN2
                    + jobInfo.priority + "</span>" + SPAN_BEGIN
                    + jobInfo.generatedTime + "</span></div>";
                jQuery("#jobsDiv").append(line);
            });
        }
    }
}
```

11.3.3 Iteration1 的演示和回顾

在关注如何使用 Jersey 完成 REST 式的 Web 服务项目 ATUP 功能的同时，我们还在一个敏捷的管理进程中。本节将对 Sprint1 所完成的核心功能和 Sprint2 所完成的模块功能进行 Scrum 的迭代收尾工作——为客户演示完成的用户故事，并做本次 Sprint 的回顾。

1. 单元和集成测试

在用户验收之前，每次提交都会执行单元测试和集成测试，相关配置见 11.3.1 节中的持续发布的“单元测试和集成测试”部分。这里对 ATUP 进行整体单元测试的校验。

```
mvn clean install -PTI
```

测试结果如下所示。

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] ATUP Project Parent ............................ SUCCESS [0.444s]
[INFO] ATUP Core ...................................... SUCCESS [3.151s]
[INFO] ATUP User ...................................... SUCCESS [12.699s]
[INFO] ATUP Device .................................... SUCCESS [7.269s]
[INFO] ATUP Case ...................................... SUCCESS [29.669s]
[INFO] ATUP Page ...................................... SUCCESS [9.077s]
[INFO] ATUP Test Station .............................. SUCCESS [2.349s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:04.929s
```

2. 验收自动化部署

本节将对自动化部署的 ATUP 进行全面的验收测试，包括部署、配置的测试，模块功能点的测试和基础功能的测试。

(1) 验收动态模块

动态模块的部署校验方法是查看 Tomcat 服务器的 webapps 目录，成功的部署应包括 ATUP 的 4 个 REST 模块。

```
ll /opt/tomcat7.0.34/webapps/
drwxrwxr-x  4 eric eric     4096 Mar  1 07:49 atup-case/
-rw-rw-r--  1 eric eric 26273260 Mar  1 07:49 atup-case.war
drwxrwxr-x  4 eric eric     4096 Mar  1 07:49 atup-device/
-rw-rw-r--  1 eric eric 26260468 Mar  1 07:49 atup-device.war
drwxrwxr-x  4 eric eric     4096 Mar  1 07:50 atup-test-station/
-rw-rw-r--  1 eric eric 26255950 Mar  1 07:50 atup-test-station.war
drwxrwxr-x  4 eric eric     4096 Mar  1 07:48 atup-user/
-rw-rw-r--  1 eric eric 26258968 Mar  1 07:48 atup-user.war
```

(2) 验收数据库

通过对数据库服务器的检测实现数据库的校验，最简脚本示例如下。

```
mysql -uroot -proot
mysql> show databases;
mysql> show tables;
+-----------------------+
| Tables_in_jaxrs2_atup |
+-----------------------+
| atup_device           |
| atup_user             |
| test_case             |
| test_result           |
| test_suite            |
+-----------------------+
5 rows in set (0.00 sec)
```

(3) 验收静态模块

静态模块的部署校验是对 Nginx 服务器的 atup-page 模块对应的目录进行检查，成功的部署应包含如下目录和文件。

```
ll ~/nginx/atup-page
drwxr-xr-x 2 eric eric 4096 Feb 27 22:48 cases/
drwxr-xr-x 2 eric eric 4096 Feb 27 22:48 devices/
-rw-r--r-- 1 eric eric  318 Mar  2 00:20 favicon.ico
-rw-r--r-- 1 eric eric 1160 Mar  2 00:20 index.html
drwxr-xr-x 2 eric eric 4096 Feb 27 22:48 jobs/
drwxr-xr-x 7 eric eric 4096 Feb 27 22:48 js/
-rw-r--r-- 1 eric eric 1282 Mar  2 00:20 signIn.html
-rw-r--r-- 1 eric eric 1170 Mar  2 00:20 signUp.html
-rw-r--r-- 1 eric eric  639 Mar  2 00:20 storageTest.html
drwxr-xr-x 2 eric eric 4096 Feb 27 22:48 users/
drwxr-xr-x 3 eric eric 4096 Feb 27 22:48 WEB-INF/
drwxr-xr-x 3 eric eric 4096 Mar  1 23:50 webjars/
```

(4) 验收 REST 接口

一旦 Tomcat 服务器启动，REST 接口就可以通过 WADL 进行校验，Jersey 默认提供了 WADL 的生成。

```
//REST 服务 WADL
curl http://192.168.1.180:8080/atup-user/rest-api/application.wadl
curl http://192.168.1.180:8080/atup-device/rest-api/application.wadl
curl http://192.168.1.180:8080/atup-case/rest-api/application.wadl
```

(5) 验收登录功能

尝试使用用户名 xer 和密码 xer 登录，如果成功，即表明登录没有问题了，进而可以校验普通用户可以查看的页面和权限。

(6) 验收添加测试设备

使用用户名 atupDevicekeeper 和密码 jaxman 登录，然后进入如下页面，添加设备。

```
http://192.168.1.160/atup-page/devices/device.html
```

设备创建后，其状态为 ERROR，只有探测线程与该物理设备成功通信，该设备的状态

才会更新为 IDLE。下面是对这个功能的校验。

(7) 校验测试设备状态

首先，通过访问如下页面可以观察设备列表中各个设备的状态信息。该页面应该定时刷新。

```
http://192.168.1.160/atup-page/devices/devices.html
```

如果该页面定时刷新，并得到期待的状态，那么证明周期检测设备状态的线程工作是正常的，即如下脚本被正确执行。

```
curl -X POST http://192.168.1.180:8080/atup-device/rest-api/devices/status
```

(8) 验收更新测试设备所有者

用户 atupDeviceKeeper 的身份是设备管理员，普通用户要创建测试作业，就要持有测试设备。因此，atupDeviceKeeper 在检测设备正常后，应分配设备给用户。首先进入设备列表页面，选取待分派的设备。

```
http://192.168.1.160/atup-page/devices/devices.html
```

选取操作会访问如下地址，并携带相关参数。进入设备管理页面，将设备分配下去。

```
http://192.168.1.160/atup-page/devices/device.html?deviceIp=192.168.1.180&deviceName
=&deviceType=0&deviceStatus=0&userId=4
```

(9) 验收创建测试用例

用户 xer 得到分配的设备，即可开始创建作业的流程。首先是创建测试用例，地址如下。

```
http://192.168.1.160/atup-page/cases/testCase.html
```

(10) 验收创建测试作业

成功创建测试用例后，可以通过访问测试用例列表来校验，参考地址如下。

```
http://192.168.1.160/atup-page/cases/testCases.html
```

在列表页面中，单击相关测试用例，即可创建测试作业，参考地址如下。

```
http://192.168.1.160/atup-page/jobs/jobsQueue.html?caseId=1&caseName=abc
```

(11) 校验测试作业是否正确下发

创建测试作业后，可以通过待下发测试作业列表观察，参考地址如下。

```
http://192.168.1.160/atup-page/jobs/runningJobs.html
```

如果测试作业在列出后的下 n 个刷新后消失，即认为测试作业已经下发。同时，证明周期下发作业的线程工作正常。对应的脚本如下。

```
curl -X POST http://192.168.1.180:8080/atup-case/rest-api/testjobs/jobs?count=5
```

(12) 验收测试结果

测试作业下发后，执行完毕会返回测试结果。通过如下地址检测测试作业的结果。如果已经列出，证明整个测试作业流程结束。

```
http://192.168.1.160/atup-page/cases/testResults.html
```

(13) 验收服务器日志

在测试流程的同时，我们可以通过服务器上各个模块的独立日志文件来观察运行情况。这一检测也是对 ATUP 中 Log4j 2 的使用的校验。相关日志文件如下。

```
ls /home/eric/atup_log
atup_case.log  atup_device.log  atup_user.log
```

静态模块的日志可以参考 Nginx 的访问日志和错误日志来检测。

```
ls /usr/local/nginx/logs
error.log access.log
```

11.3.4　Sprint 3 持续交付

在完成基础功能和模块功能后，ATUP 的开发工作基本已经完成，新的 Sprint 的重点放到了持续集成测试上。当然，这期间通常会夹杂着小功能的开发和缺陷的修复，因此，在这一过程中，小步、快速地进行持续集成便是重点。

1. 持续集成

ATUP 的 Maven 配置完成后，应使用 Jenkins 的 job 来触发而不是在终端中手动执行。一个完整的持续集成（CI）和持续交付（CD）流程如图 11-11 所示。

图 11-10 展示了整个持续交付的过程。持续交付是持续集成的延伸和最终目的，在图 11-11 中，Nexus 是 Maven 私有仓库的一种，Gitolite 是 Git 的私有服务器。CI 流程的第一个关键点是 Git 的 hook 脚本。当开发者提交代码到 Gitolite 服务器后，会触发 post receive 脚本，该脚本中定义了对 Jenkins 指定 job 的访问。在 Jenkins 一端，配置了 Maven 命令，即前面使用的 Maven 命令。这期间，代码覆盖工具 Cobertura 和代码质量检查工具 sonarQube 会被 Jenkins 的 job 调用，生成相关报告。最后，构建成功后会执行前面讲过的部署操作。

在简要了解了 CI 流程后，回头观察，CI 流程的发起是开发者的提交操作。因此，Sprint3 的工作比起前面的方式，在敏捷和自动化测试上都更好。这里不再深入研究如何实现各个服务器的搭建和配置，读者可自行实现，以深入理解和掌握相关原理和细节。

2. 分模块部署

分模块部署只是对前面工作的细化，并没有新的知识点。单独部署 atup-user 模块到 192.168.1.181，atup-case 模块到 192.168.1.182，atup-device 模块到 192.168.1.183，atup-page 模块到 192.168.1.161。修改 jax-rs2-atup\atup-page\src\main\webapp\js\atupCommon_staging.js 文件，参考代码如下。

```
var ATUP_USER_HOST = "http://192.168.1.181:8080/";
var ATUP_CASE_HOST = "http://192.168.1.182:8080/";
var ATUP_DEVICE_HOST = "http://192.168.1.183:8080/";
var ATUP_PAGE_HOST = "http://192.168.1.161/";
```

部署时，提供动态的服务器地址参数，示例代码如下。

```
mvn clean install -DskipTests -Dmysql.server.ip=192.168.1.181 -PCI
```

3. 灰度发布

CI 起始于开发者的提交带来的好处是，新的功能和改变可以很快上线，让用户第一时间验收。通过在代码中使用 switch 和动态指定服务器地址可以使指定服务器为指定人群开放指定的功能。这对系统的稳定性和伸缩性带来了更多的支持。

ATUP 采用多 staging 的方式提供不同用户组不同功能的部署模式，这里不再展开介绍。

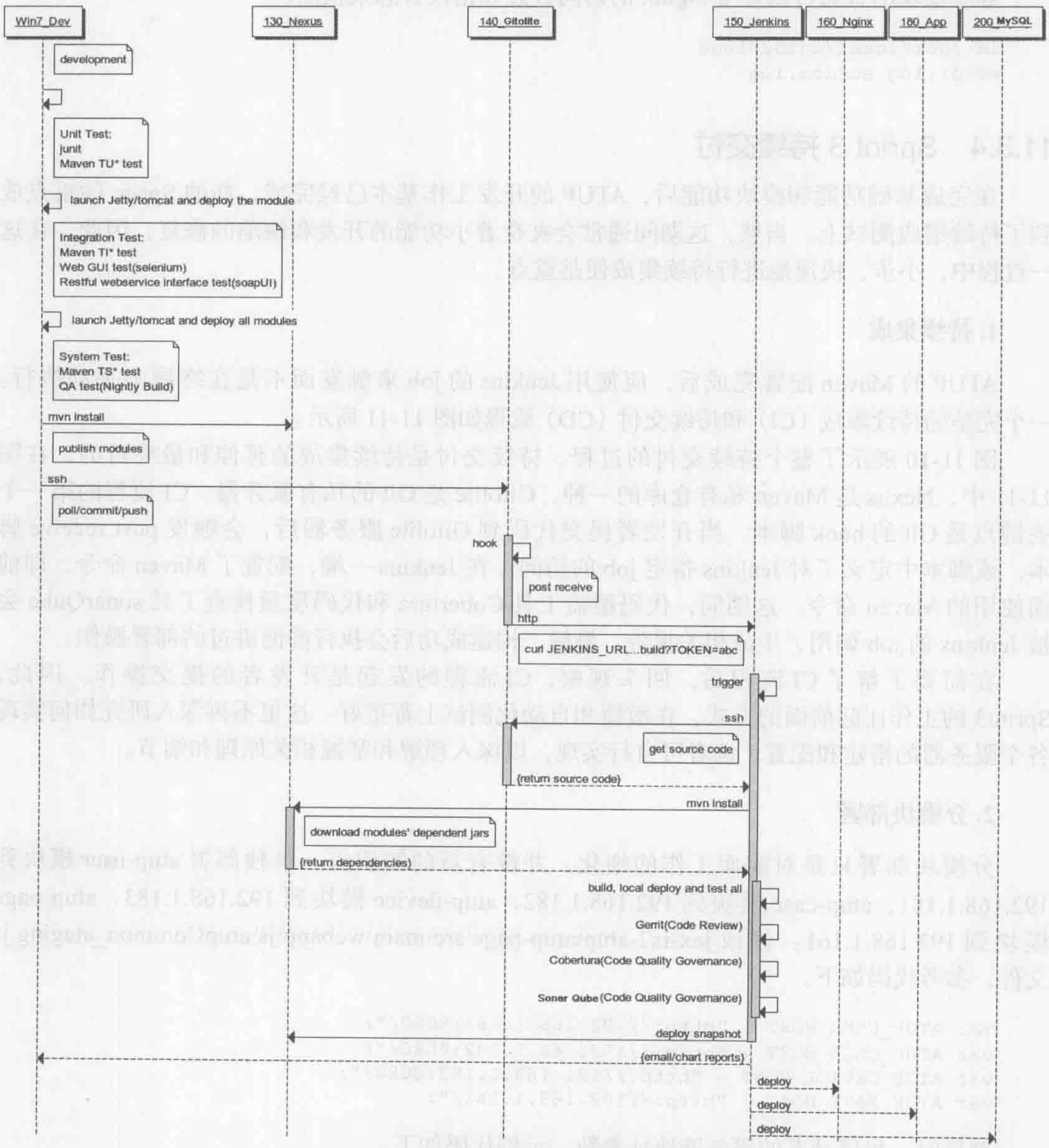


图 11-11 持续交付流程示意图

4. Iteration2 的演示和回顾

相较于 Iteration1 的验收测试，Iteration2 给出了更自动化的方式。因为前者的部署是通过 Maven 命令执行的构建和部署，而后者是通过 CI 流程 / 工具完成的一键式部署。

Iteration2 的验收包括 CI 流程中各个服务器的配置和基本功能的验证、CI 流程的完整性、CI 结果的完整性等。这超出了 ATUP 功能本身的讲述范围。

11.3.5　交付和总结

经过了三个 Sprint 的敏捷开发，我们演示了开发一个 REST 式的 Web 服务的全过程。虽然，ATUP 是不太完善的，但它是可用的。敏捷带给我们的思想是简单，我们从简单的设计和实现入手，经过不断迭代，持续重构和完善既有产品及其设计。如果再扩展去谈敏捷，就超出了本书的范围。下面将就此展望一下 ATUP 的下一个迭代。

（1）下一个迭代

针对代码覆盖率和代码质量，开发团队内部比谁都清楚该如何提高。与此同时，来自客户的新功能和已有功能的改进对 ATUP 既有代码的重构和实现会提出新的挑战。这些都是未来迭代需要实践的。

（2）重构

代码重构是敏捷实践不可或缺的环节。作为开发者，我们必须知道哪里应该重构。除了不断阅读 Java 编程规范、重构和设计模式等经典著作，我们也应（借助如下工具）从实践中总结出自己的设计和重构经验。

1）sonarQube：为用户提供评分和改进方案是这款质量保证平台最有价值的地方之一。开发者可以从中自我提高。

2）Profiler：JVM 的 Profiler 工具是 ATUP 出现性能问题时需要考虑使用的，这样可以尽快定位实现过程中“制造”的问题。

3）Probe：Probe 这类运行时服务器监控工具对辅助了解运行时资源使用情况非常有帮助。如果 ATUP 系统不能良好运行，则可以阶段性地考虑使用，以排查问题。

11.4　本章小结

本章讲述了一个完整的 REST 式的 Web 服务的实现全过程，结合 Scrum 敏捷实践，对 ATUP 平台的需求、设计和编码、测试、发布进行了详细的讲述。此外，讲述了 REST 开发的大环境，包括小步迭代中使用的持续集成 (CI) 流程及工具、重构和螺旋式提高代码质量的心得分享。

Web 简史

从 REST 和 JAX-RS 的角度来看 Web 开发的发展时，我们可以从 C/S（Client/Server）时期开始简单回顾，即可更清楚地明白 REST 在 Web 领域为何成为一种趋势，以及如何与那些息息相关的技术集协同工作的。

从胖客户端到瘦客户端

胖客户端指的是 C/S（Client/Server）架构中的 Client。C/S 时期是以 Windows 为代表的桌面应用时代，也是 VB、Delphi 盛行的时期。Java 领域的 Applet 是这个时期的后起之秀，并成为引领 C/S 时期转向 B/S（Browser/Server）时期的“旗手”。相对于今天的浏览器（客户端），C/S 时期的客户端称为胖客户端，因为 Client 维护着用户的会话信息和交互信息，对于数据校验、缓存等功能“亲历亲为”，界面的用户体验比后来的 B/S 结构更具优势，使服务器专注于处理数据。

从 C/S 到 B/S 的转型是网络时代崛起的标志和必然趋势。胖客户端明显的缺点是它必须安装到客户端本地。虽然 Applet 没有一个明显的安装客户端的过程，但也需要本地 Java 运行环境（JRE）的支持。对于开发和维护者来说，客户端的多版本共存和升级就是“无尽”的任务。对于用户来说，最不情愿的事情就是被迫升级本地的客户端版本，而且每当切换操作系统，就要重新安装一次客户端。至于跨平台使用是一种奢望，胖客户端对操作系统有依赖，甚至会陷入 DLL 窘境，但这一问题在浏览器作为客户端的 B/S 时期荡然无存。

因此，B/S 应运而生。B/S 无须安装，打开浏览器，输入网址就可以进入应用。B/S 的出现更重要的原因是实现那个时期的一个愿望：那个时期的服务器端编程已经非常强大，我们

为什么要让客户端做那么多的事情呢？把它交给服务器吧。在 Java 领域，以 Servlet 和 JSP 为基础的瘦客户端开发逐渐成熟，Struts 带来了 MVC，Spring 带来了 IoC 和 AOP，Hibernate 带来了 O/R-Mapping，“瘦小”的前端也引入了模板、标签等技术。B/S 蓬勃发展，并且促使 Java 的标准随之而动，Struts 和 Spring 促成了 JSF 的诞生，Spring 改变了 EJB 在 3.0 之后的命运，Hibernate 促成了 Java 领域的数据库访问新标准 JPA。Java 社区和标准组织不断出台的、相互协作、相互促进的框架和技术使得瘦客户端走上历史的顶峰。开发过程变得富有哲理和艺术气息，分层开发有利于解耦，易于梳理和定位异常和缺陷。我们可以想象，从分层逻辑中收益的绝不只是开发工程师，还有质量保证工程师、运维工程师，甚至整个研发团队。

从瘦客户端到富客户端

时至今日，B/S 依然流行，但 B/S 结构并没有停留在开始的瘦客户端阶段。把工作都交给服务器是瘦客户端的初衷，同时，这样的想法也是一把“双刃剑”。由于 HTTP 是无状态的，Session 成为了服务器必要的“累赘”，用户的每一次交互都要交由服务器处理，甚至是验证输入框信息这么小的一个交互，包括这期间的时延等带来的用户体验非常糟糕。瘦客户端，从另一个角度看是一个“穷客户端”，除了处理提交和展示服务器返回的信息以外几乎什么也没有。时代的选择是让客户端“富”起来。

AJAX 是这个阶段比较成功的代表。异步请求和 DOM 的动态改写带来了页面的局部刷新，Dojo 和 jQuery 为代表的 Java Scrpit 库为客户端带来了丰富的组件和皮肤风格，良好的用户体验和更为轻松愉快的开发过程让“富”起来的客户端“活力四射”。客户端开发的发展不是再度发“胖”而走向倒退，而是在坚持 B/S 结构的同时，使之功能强大起来。

AJAX 的出现其实是一件水到渠成的事情。因为在这期间，许多技术严谨、不拘一格的开发者在为 JavaScript 贡献着开源 JavaScript 库，这些优秀的 JS 库不仅是工具包，其中包含了许多引领时代发展的思想和技术。虽然富客户端的美学因素让开发十分惬意，但跨浏览器开发中，样式和事件的调试和测试是极其专业的事情，再不是一个进行过后台开发就能上来摆弄前端的时代了。AJAX 的出现成就了许多专业的前端开发者，从这个时期开始，前端开发的技术含量逐日增加，前端开发工程师也开始被业内更多地认可和尊重。AJAX 在很大程度上会推动 Web 开发的未来，包括单页面技术、平庸客户端。

另一个代表是 JSF。JSF 是 Java 领域的一个标准，采用浏览器端的事件驱动的机制，它吸收了 Spring 和 AJAX 的精华，是 Java 领域中最具 ASP.NET 风范的技术框架。其全栈式的开发框架和组件的生命周期模式使得富客户端开发变得快速和一致。需要提及的是，XHTML 在 JSF 领域得到了重视和推动。JSF 的缺点是设计上不够轻量，入门的起点较高，其广泛性也远远不如 Spring 等。

还有一组代表是 JavaFX、Flex，以及微软公司的 Silverlight 这样的技术。富客户端这个时期并没有结束，但同属 RIA（Rich Internet Applications）时期的 Flash，由于市场的原因，对开发团队和开发者来说，也许会考虑其投资回报率和风险。

从富客户端到“平庸”的客户端

“平庸”的纯 HTML 网页可以做的事情主要是通过 HTML 标签来展示数据。纯 HTML+AJAX 的功能是发出 HTTP 请求并局部刷新页面，展示服务器返回的数据。这样的客户端在 REST 出现之前，似乎太有局限性，我们必须引入脚本语言（ASP、PHP、JSP 等）来“壮大”客户端。而有了 REST，这种平庸的客户端就可以完成更多的业务逻辑功能。因为 REST 式的 Web 服务公布的就是简单的 HTTP 请求。这看似变弱的变化，有什么意义呢？

平庸的客户端带来的是一个 Web 发展的新时期。这个时期，客户端不仅是平台无关的，而且是语言无关的，这样的客户端可以实现对任何一种编程语言实现的 REST 服务器的请求。REST 服务提供者的 DEMO 包里再不需要区分 PHP、JSP、ASP.NET 版本，只需要提供 HTML 版本，甚至是 cURL 脚本或者 Shell 脚本。

解耦是平庸客户端最明显的优势。松散的耦合使开发难度降低，系统的水平扩展能力增强。从代码耦合上看，纯 HTML 的客户端开发并不关心服务器端使用的是哪一种编程语言，唯一需要和服务器端交互的是 REST 接口，其开发成本大大降低。从部署上看，纯 HTML 的客户端是静态页面，可以部署在以 Nginx 为代表的 HTTP 服务器上，有效地降低了应用服务器的负载。富客户端在这方面是束手无策的，动态脚本不仅有性能问题，而且在部署过程中，很难实现和服务器端代码分离。面对敏捷和持续集成、持续部署和持续交付，纯 HTML 的客户端可以轻松地以独立的模块作为被测试和部署的基本单位。

平庸客户端带来的思考是，前端为何走向平庸（包括美术风格上的扁平）。Web 发展过程中，始终不是孤立的，在 C/S 时期，移动设备有 Java ME；在 MVC 流行的时期，设计者和开发者在开发 Web 的同时，Wap 版本占据着一席之地；从移动 2.5+G 时代开始，iOS 和 Android 占据的社交网络、电子商务等领域的市场配额逐步增加，“短平快”成为趋势。我们所关注的不局限于 Web 页面如何解读复杂业务，而是关注服务器和客户端之间如何快速通信、快速响应。AJAX 不是终点，它虽然解决了客户端异步请求问题，但网络 IO 依然是阻塞的，下一个主题会是什么呢？相信会覆盖到 HTML5 以及与之并行的服务器推送技术和服务器异步、非阻塞 IO。与此同时，平庸的客户端也会出现新的特性，尤其是作为配合服务器新特性的展示层，比如 Web Socket 以及与之并行的 Server Push Taglib、JS 库的 Comet 支持。

在基于 Jersey 的实现中，REST 风格的应用已经具备了异步处理请求、非阻塞式读写、动静分离等特征。同时，可以很好地与 Web Socket、SSE 等 HTML5 特征协同工作。

参考资料 Reference

[1] Roy Thomas Fielding. 架构风格与基于网络的软件架构设计 [C] 李锟、廖志刚、刘丹、杨光，译 . 尔湾：加州大学欧文分校，2000.

[2] Roy Thomas Fielding. 架构风格与基于网络应用软件的架构设计（中文修订版）[J/OL]. 李锟，译 . http://www.infoq.com/cn/minibooks/web-based-apps-archit-design.

[3] JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services[J/OL]. https://www.jcp.org/en/jsr/detail?id=339.

[4] Jersey 2.10.1 User Guide[S]. https://jersey.java.net/documentation/latest.

[5] Network Workng Group、Hypertext Transfer Protocol——HTTP/1.1[S]. http://www.ietf.org/rfc/rfc2616.txt.

[6] Leonard Richardson , Sam Ruby. RESTful Web Services[M]. O' Reilly Media, Inc，2007.

[7] Subbu Allamaraju. RESTful Web Services Cookbook[M]. O' Reilly Media, Inc，2010.

[8] Network Working Group. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIS), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations[S]. http://www.ietf.org/rfc/rfc3305.txt.

Afterword 后记

作为技术人员，你的头衔可能是技术总监、架构师、技术主管，或者高级开发工程师、开发工程师等，请别嫌弃作为程序员的这个身份。我喜欢学着《喜剧之王》里尹天仇的口吻说，其实我是一个程序员。因为，程序员这个身份是上述头衔去除名和利之后的本质。

诚然，技术总监可以是从系统工程师、质量保证工程师提升上来的，他们懂工程，懂质量保障，但是一名合格的 CTO，他必须懂开发。所以，我坚信他是自己人。

身为程序员，首先应该具备的素质就是持续学习。因为“码”这个瞬间动词需要“厚积薄发”这个持续的形容词来做基础，知识的积累和梳理需要你不间断地学习、实践和思考。另一个素质是“站队”和甄别。“站队”（选择某种技术）需要有一定的积累，你会知道什么技术是有发展的，因为技术层出不穷，你不可能都去学习，一要根据兴趣，二要看清未来的发展趋势。甄别是要懂得学什么，是学原理和规范还是学工具和实例。我曾经目睹某位 Android 的工程师在不了解 Java 的设计模式的情况下，误读内部架构的设计。我们要做好程序员，一定要“知其然，知其所以然”，否则很难成长，纵然你很努力。

选择本书的读者一定是在持续学习的人，你站在了 Java 和 REST 的“队伍”里。你第一步要做的是知识的积累，即看完这本书，然后是梳理，按照你对 JAX-RS 的理解完成书里的例子，同时做笔记。建议在掌握本书内容后，去做两件事：翻看一下 JSR 339，它是 Java API for RESTful Web Services 的技术规范；再查看一下 Jersey 的源代码，它是 JSR 339 规范的实现，你可以从中了解到如何使用 Java 的技术来实现一个规范。本书涉及 jQuery、Dojo 等工具，如果你懂得甄别的话，那么对工具的学习应该是够用就好。

Java
RESTful Web Service实战

传统的Web服务是基于RPC风格的，其实现技术主要包含SOAP、WS标准栈等。RPC风格的Web服务用在分布式的、开放的环境中会带来一些问题，如技术架构复杂、可伸缩性差等，REST风格的Web服务足以解决上述问题。本书收纳了作者近3年的RESTful实践经验，将REST理论与Java实现相结合，循序渐进地将使用Java开发REST式的Web服务中总结的知识点和经验呈现给读者。各个章节中的知识点都精心设计了相应的示例代码，便于读者更好地理解和更及时地实践。

笔者从敏捷角度为读者呈现了一个完整和相对复杂的REST式的Web服务实例，相信这个实例能让读者更好地融会贯通，同时，可以对敏捷开发和自动化测试有新的认识。

多年前我在开发 Sonatype Nexus 的时候首次接触并熟悉了 REST 风格的 Web 服务，这种风格使得我们的设计简单、规范且易测，Nexus 的大部分功能都可以通过一行简单的 cURL 命令来验证，单凭这一点就足以让我喜爱上 REST 风格。现如今，开发 REST 风格的 Web 服务可简单多啦，尤其是如果你遵循 JAX-RS 2.0 标准并借助 Jersey 优秀的实现，编写 REST 风格 Web 服务的难度就大大降低了。这本书全面生动地阐述了 JAX-RS 2.0 标准，不仅覆盖了 API、请求处理、安全等核心内容，更有异步通信、调优等高级主题，无疑是学习 Java REST 风格的 Web 服务的绝佳参考。韩陆技术功底扎实，在写作过程中潜心阅读了大量的 Jersey 源码，这也让这本书的质量得到了进一步保障，相信本书定会助你在 REST 的道路上更快、更稳地前行。

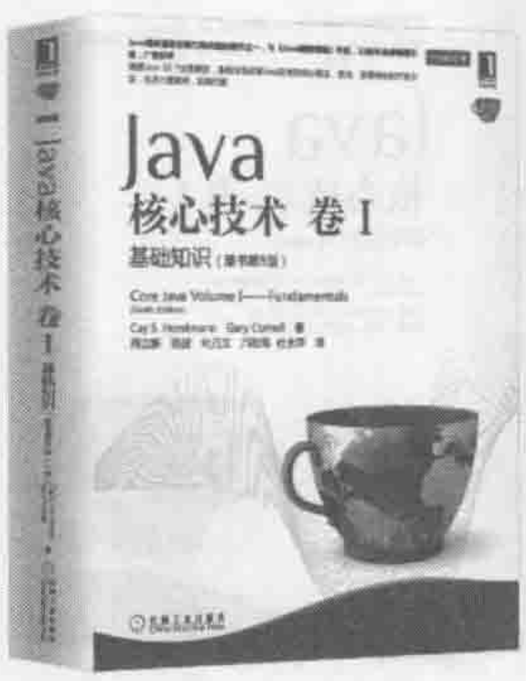
——许晓斌，《Maven实战》作者

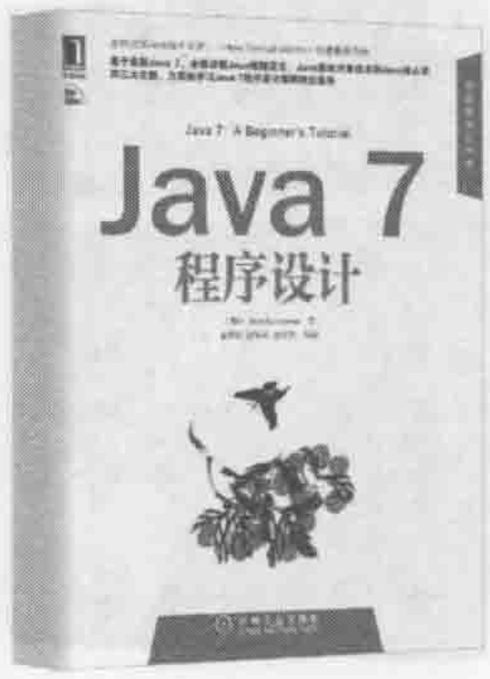
投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

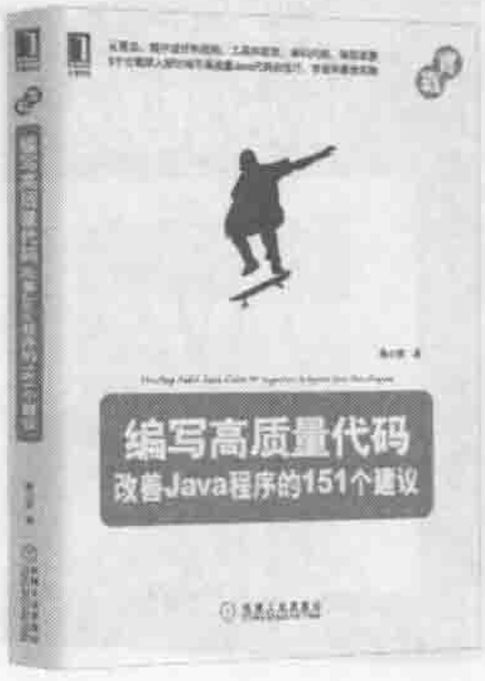
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

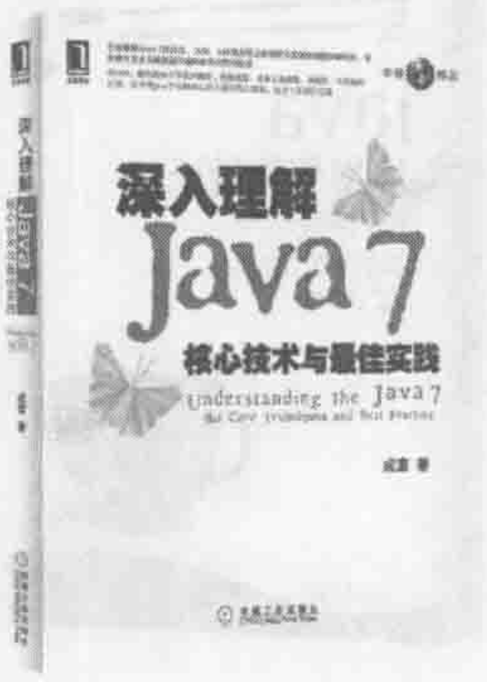
上架指导：计算机/程序设计/Java
ISBN 978-7-111-47888-1
9 787111 478881 >
定价：69.00元

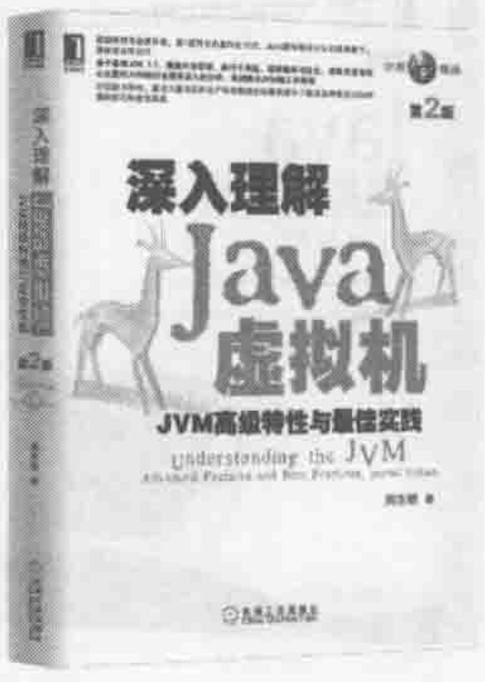
推荐阅读

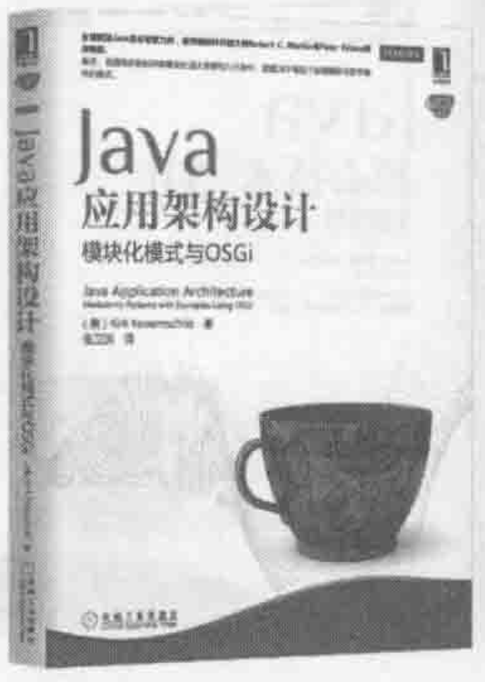

Java
核心技术 卷I
基础知识
Core Java Volume I——Fundamentals

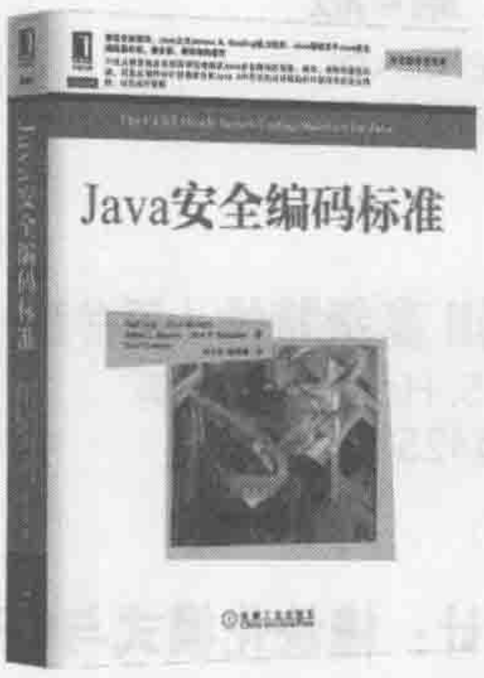

Java 7
程序设计

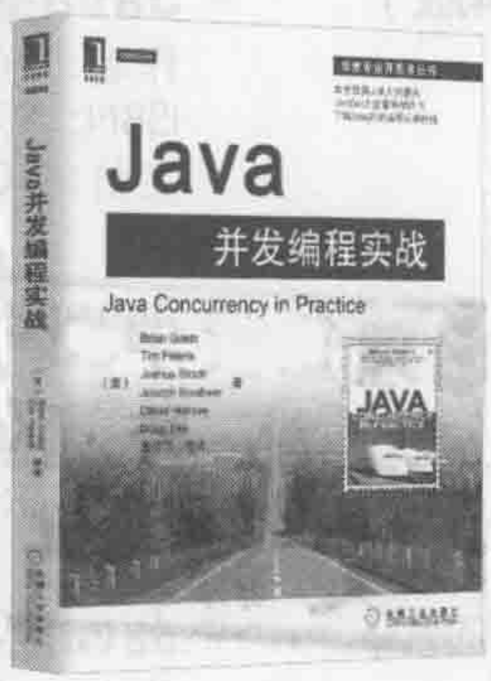

编写高质量代码
改善Java程序的151个建议


深入理解
Java 7
核心技术与最佳实践
Understanding the Java 7


深入理解
Java
虚拟机
JVM高级特性与最佳实践
Understanding the JVM

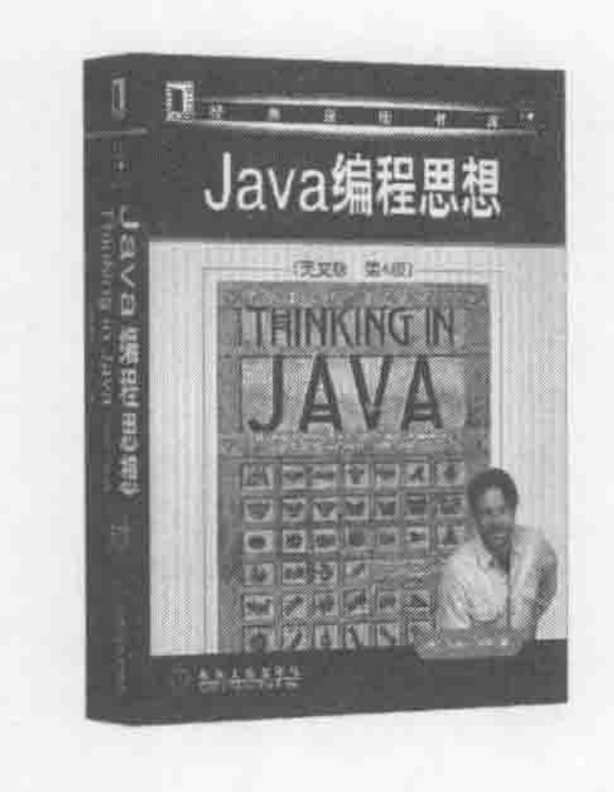

Java
应用架构设计
模块化模式与OSGi
Java Application Architecture

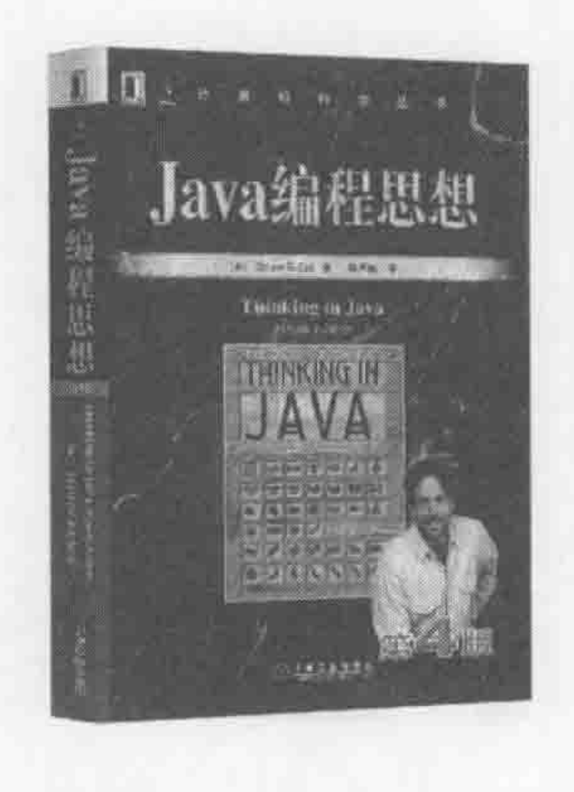

Java安全编码标准

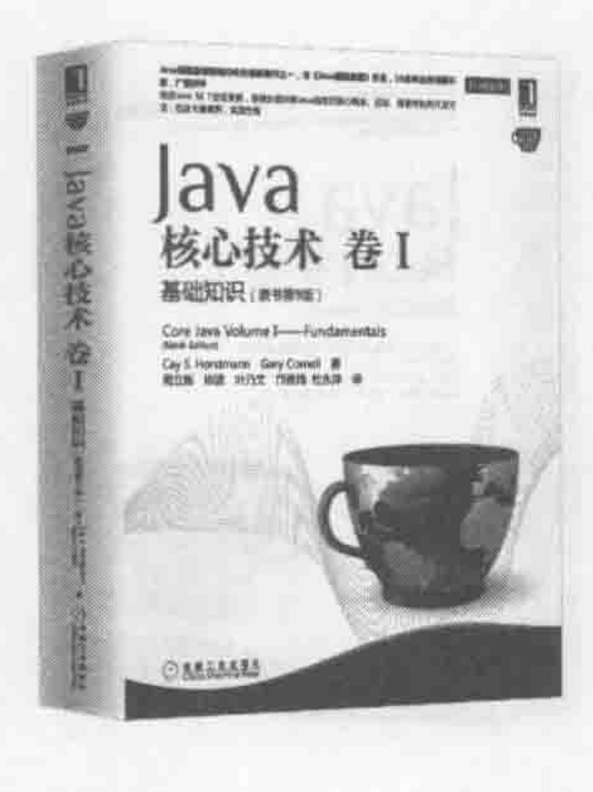

Java
并发编程实战
Java Concurrency in Practice

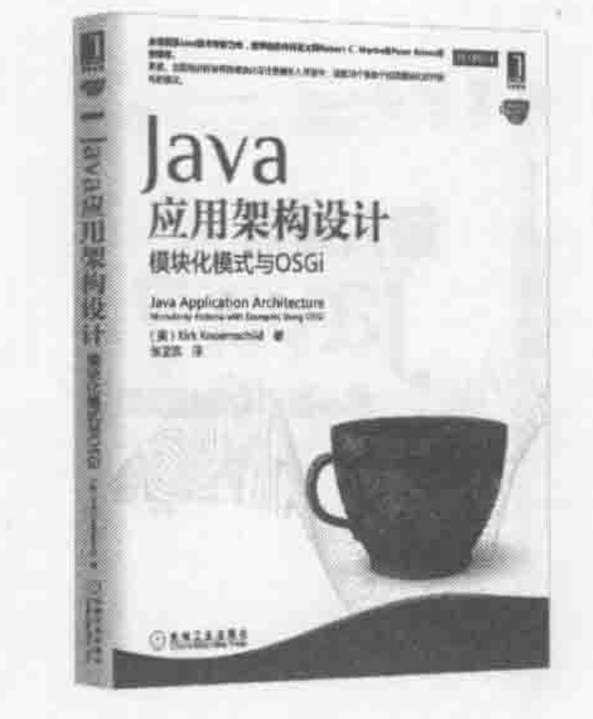
Java虚拟机并发编程

推荐阅读









- **Java编程思想（英文版·第4版）**
  作者：（美）Bruce Eckel
  ISBN：978-7-111-21250-8
  定价：79.00元

- **Java编程思想 第4版**
  作者：（美）Bruce Eckel
  ISBN：978-7-111-21382-6
  定价：108.00元

- **Java核心技术 卷I 基础知识（原书第9版）**
  作者：（美）Cay S. Horstmann 等
  ISBN：978-7-111-44514-2
  定价：119.00元

- **Java核心技术 卷II 高级特性（原书第9版）**
  作者：（美）Cay S. Horstmann 等
  ISBN：978-7-111-44250-9
  定价：139.00元

- **Java虚拟机规范（Java SE 7版）**
  作者：（美）Tim Lindholm 等
  ISBN：978-7-111-44515-9
  定价：69.00元

- **Java应用架构设计：模块化模式与OSGi**
  作者：（美）Kirk Knoernschild
  ISBN：978-7-111-43768-0
  定价：69.00元