

架构探险

——从零开始写 Java Web 框架

黄 勇 著

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书首先从一个简单的 Web 应用开始,让读者学会如何使用 IDEA、Maven、Git 等开发工具搭建 Java Web 应用;接着通过一个简单的应用场景,为该 Web 应用添加若干业务功能,从需求分析与系统设计开始,带领读者动手完成该 Web 应用,完善相关细节,并对已有代码进行优化;然后基于传统 Servlet 框架搭建一款轻量级 Java Web 框架,一切都是从零开始,逐个实现类加载器、Bean 容器、IoC 框架、MVC 框架,所涉及的代码也是整个框架的核心基础。为了使框架具备 AOP 特性,从代理技术讲到 AOP 技术,从 ThreadLocal 技术讲到事务控制技术。最后对框架进行优化与扩展,通过对现有框架的优化,使其可以提供更加完备的功能,并以扩展 Web 服务插件与安全控制插件为例,教会读者如何设计一款可扩展的 Web 应用框架。

本书适合具备 Java 基础知识,熟悉 Web 相关理论,并想成为架构师的程序员阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

架构探险:从零开始写 Java Web 框架/黄勇著. —北京:电子工业出版社,2015.8

ISBN 978-7-121-26829-8

I. ①架… II. ①黄… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 176473 号

责任编辑:陈晓猛

印 刷:北京天宇星印刷厂

装 订:北京天宇星印刷厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:22.75 字数:509.6 千字

版 次:2015 年 8 月第 1 版

印 次:2015 年 11 月第 3 次印刷

印 数:5001~8000 册 定价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

序

其实一开始黄勇找我为他的处女作写序的时候，我是拒绝的。因为你不能让我写，我马上就写。我要先看一下书，因为我不愿意写完后发现书很烂，然后读者来骂我乱推荐。

黄勇一直是开源中国非常活跃的会员，非常积极地回答各种问题和分享自己所擅长的知识。本身也是 **Smart Framework** 框架的作者，积分居然过千，要知道开源中国上超过 1000 积分的会员寥寥无几。我和黄勇并没有见过面，仅通过线上的信息大概觉得他是一个热情、虚怀若谷又非常接地气的技术牛人。

回归正题，2015 年是 Java 的 20 周年。Java 是一门让我们又爱又恨的编程语言，在编程语言界里算是一棵老树，关键是这棵老树还频频长出新枝。Java 8 的出现使其具备了各种流行的编程理念，而全新的 Java 9 也已经奠定了 Java 发展的里程碑并将于 2016 年下半年发布正式版本。说起又爱又恨，在 Java 最火热的那几年里，满世界的 SSH（Struts+Spring+Hibernate），所有人都在讨论 SSH 框架里的奇技淫巧，也出现了各种图书和培训教程。很多初学者认为 Java = SSH，学 Java 就是学 SSH，以至于很多人用 SSH 做了不少项目，但依然对 Java、HTTP 等基础知识一知半解、不甚了了。

在 Web 开发方面，Java 经历了这么几个阶段，从最开始使用大量 Servlet 来处理各种业务逻辑，然后出现了著名的 Struts 框架，大大简化了 Web 应用的开发以及配置，而后是 Hibernate 和 Spring 的出现，使这三者成为三驾马车，并一直流行到现在。

而如今，确切地说应该是最近几年，Java 用户开始回归理性。由于 SSH 在不断发展的同时，体积也变得越来越庞大，很多人在使用的过程中被各种配置、注解弄得头昏脑涨。而前些年 Ruby on Rails 框架以其“惯例优于配置”的理念让我们猛然清醒——原来 Web 的开发就应该这么简单。于是越来越多的 Java 开发者开始考虑轻量级框架解决方案。而黄勇的 Smart

Framework 就是这种轻量级解决方案之一。

Java 的世界从来不缺乏各种优秀的开源软件，理念成熟后大量的轻量级 Web 框架如雨后春笋般出现在我们眼前。在开源中国网站上你会发现 Java 的 Web 框架有超过 300 款之多。那么多的框架对初学者来说简直是噩梦。于是三年前我写了一篇文章《初学 Java Web 开发，请远离各种框架，从 Servlet 开发》，今天一看，这篇文章居然超过了 21 万阅读量。这篇文章是针对 Java 初学者的，因为很多人为了学习各种框架而疲于奔命，但却从来没有思考为什么同样是做开发的，自己要比别人更累。最根本原因在于方法不对，事倍功半！

在招聘 Java 开发人员时我最爱问的一个问题是：请解释一下 Session 的工作原理，从而来判断应聘者对基础知识的掌握情况。那么什么才是初学者学习 Web 开发的好方法呢？其实我在前面提到的文章里包含了详细的步骤，简单地说就是先要熟悉 Java EE 里关于 Servlet API 中的常用类和方法。在这个基础上再去学习某个框架的使用，最后是阅读 HTTP 协议的内容。想成为高手必须对 HTTP 协议有着深入的了解。一旦掌握了这些基础的内容，你会发现使用框架甚至是开发一个适合自己业务的框架是多么的容易。

而黄勇的这本书，虽然我只是看到书的目录以及前面两三章的内容，但相信这是一本对初学者非常好的书，没有华丽的词藻，实实在在地讲述了整个开发流程。这本书不是在教你怎么用 Smart Framework，而是展示了作者开发这个框架的整个心血历程，包括设计一个框架所用到的各种技术，还涉及了很多底层的 Java 技术，如类加载器、依赖注入、线程本地、事务管理和安全控制，等等。

不管是学习还是在实际的开发中，少问怎么做，多问问为什么要这么做。或许，这就是黄勇想要告诉大家的吧。

红薯

开源中国 (oschina.net) 创始人

2015 年 6 月 12 日

于北京到上海的 G123 列车上

前言

记得在 2013 年 9 月，那时我工作不太忙，利用自己的闲暇时间，写了一款名为 **Smart** 的轻量级 **Java Web** 框架，并发布到“开源中国”（oschina.net）社区网站上。当时引发了同行们的大量关注，随后纷纷出现了其他大量关于“轻量级 **Java Web** 框架”的开源项目。

当然也有人觉得此类项目没有多大意义，毕竟 **Spring** 等框架已经足够成熟了，功能完全能够满足我们日常的开发需求，为什么还要“重复发明轮子”呢？

对此问题，我是这样认为的：

1. 对于 **Spring** 框架而言，虽然它已经足够强大了，但也更加臃肿了，因为它提供的所有功能我们并非都需要。
2. 市面上有很多优秀的开源框架，我们不妨取其精华，自己动手开发一款适合自身开发需求的框架。
3. 国内开源环境与国外差距较大，我们需要从自身做起，才能带动身边更多的人一起投身到国内开源事业中去。

我并非是想让大家去造轮子，而是想把这个造轮子的过程描述出来，让大家在这个过程中有所收获。也许我的力量非常薄弱，但只要能涌现出一批热血的开源人，相信国内开源市场的前景一定会越来越好！

我认为自己的选择并没有错，于是我一边写代码，一边写博客，在短短的几个月里，写了上万行代码、上百篇博客（我的博客地址：<http://my.oschina.net/huangyong/blog>）。我想把自己在开发过程中遇到的问题与经验都记录下来，并且分享出去，让更多的人受益。

直到有一天，我接到了博文视点陈晓猛编辑的邀请，陈编辑想让我出一本书，选题让我自己来定。当时我既欢喜又发愁，欢喜是由于终于有机会出版自己的书了，发愁是由于自己的经

验与水平十分有限，难以写出好的作品，所以只能大胆尝试了。

在写这本书之前，我发现在国内介绍如何使用一些开源框架的书已经非常多了，但教会大家怎样从零开始写框架的书却非常少，因此我打算写一本这样的书——从零开始写 Java Web 框架。在写书前我备感压力，因为我只写过一点技术博客，还从未写过书。对于写书而言，我是非常缺乏经验的，总是担心自己写得不够好，要么写得太晦涩，要么写得太肤浅。反反复复逐字推敲，直到交稿的时候，我心里还是忐忑不安。

我在写这本书时下定了决心，面对的读者非常具体，他们必须具备一定的 Java 功底，尤其是 Java Web 方面的开发经验，他们都是希望步入架构师行列的程序员。所以，这本书的内容不能过于理论，而是需要将理论融入到实践中。全书以实现一款轻量级 Java Web 框架为主线，建议读者能亲手实践，这样才能学到更多有价值的东西。

本书内容

全书共 5 章，每章有先后顺序，建议读者按照章节顺序进行阅读。

第 1 章：从一个简单的 Web 应用开始，教会读者如何使用 IDEA、Maven、Git 等开发工具来搭建一个 Java Web 项目。

第 2 章：为 Web 应用添加业务功能，从需求分析与系统设计开始，带领读者动手开发一款简单的 Web 应用，并进行了相关细节完善与代码优化。

第 3 章：搭建轻量级 Java Web 框架，一切都是从零开始，逐个实现类加载器、Bean 容器、IOC 框架、MVC 框架，本章所涉及的代码也是整个框架的核心基础。

第 4 章：使框架具备 AOP 特性，从代理技术讲到 AOP 技术，从 ThreadLocal 技术讲到事务控制技术，通过阅读本章，读者可学会如何实现 AOP 框架，以及事务管理框架。

第 5 章：框架优化与功能扩展，通过对现有框架的优化，使框架提供更加完备的功能，并以扩展 Web 服务插件与安全控制插件为例，教会读者如何对框架进行扩展。

致谢

首先要感谢我的家人，特别是我的妻子，当占用大量休息时间写作的时候，她能够给予我极大的理解与支持，独自承担了所有的家务，以及照顾宝宝的生活，让我能够在没有任何打扰的环境下，全身心地投入到写作当中。如果没有她的帮助，这本书是无法完成的。

感谢开源中国社区网站，感谢创始人红薯，感谢曾经帮助我的朋友们，他们是：大漠真人、july、小菜的粉丝、ipandage、蛙牛、Liuzh_533、罗盛力、Bieber、哈库纳、悠悠然然、黄亿华、

Dead_knight、疑似一僧、杨唯浩、V 神、石头哥哥、乒乓狂魔、邹建芳、山哥、光石头（排名不分先后），没有你们的支持与鼓励，我是没有决心把开源做下去的，更不可能写出这本书。

感谢与我一起共事的同事们，让我体会到了团队的力量，在你们的身上我学到了很多宝贵的经验，也感谢你们为本书提供的宝贵建议。

感谢博文视点的编辑，这本书能够如期出版，离不开你们的敬业精神与一丝不苟的工作态度，我为你们点赞！

本书资源

本书下载资源请登录博文视点官网（www.broadview.com.cn/26829）中的“下载资源”进行下载。

读者俱乐部

欢迎加入本书读者俱乐部 QQ 群：179323031。

黄勇

2015 年 6 月于上海

目 录

第 1 章 从一个简单的 Web 应用开始	1
-----------------------------	---

正所谓“工欲善其事，必先利其器”，在正式开始设计并开发我们的轻量级 Java Web 框架之前，有必要首先掌握以下技能：

- 使用 IDEA 搭建并开发 Java 项目；
- 使用 Maven 自动化构建 Java 项目；
- 使用 Git 管理项目源代码。

1.1 使用 IDEA 创建 Maven 项目	3
1.1.1 创建 IDEA 项目	3
1.1.2 调整 Maven 配置	3
1.2 搭建 Web 项目框架	5
1.2.1 转为 Java Web 项目	5
1.2.2 添加 Java Web 的 Maven 依赖	6
1.3 编写一个简单的 Web 应用	10
1.3.1 编写 Servlet 类	10
1.3.2 编写 JSP 页面	11
1.4 让 Web 应用跑起来	12
1.4.1 在 IDEA 中配置 Tomcat	12

1.4.2	使用 Tomcat 的 Maven 插件	13
1.4.3	以 Debug 方式运行程序	13
1.5	将代码放入 Git 仓库中	14
1.5.1	编写.gitignore 文件	14
1.5.2	提交本地 Git 仓库	15
1.5.3	推送远程 Git 仓库	15
1.5.4	总结	16
第 2 章 为 Web 应用添加业务功能		17
我们在这个应用的基础上增加一些业务功能，您将学会更多有关项目实战的技能，具体包括：		
<ul style="list-style-type: none">• 如何进行需求分析；• 如何进行系统设计；• 如何编写应用程序。		
2.1	需求分析与系统设计	19
2.1.1	需求分析	19
2.1.2	系统设计	19
2.2	动手开发 Web 应用	21
2.2.1	创建数据库	22
2.2.2	准备开发环境	22
2.2.3	编写模型层	23
2.2.4	编写控制器层	25
2.2.5	编写服务层	27
2.2.6	编写单元测试	28
2.2.7	编写视图层	31
2.3	细节完善与代码优化	31
2.3.1	完善服务层	32
2.3.2	完善控制器层	59
2.3.3	完善视图层	60
2.4	总结	65

第 3 章 搭建轻量级 Java Web 框架 66

我们需要这样的框架，它足够轻量级、足够灵巧，不妨给它取一个优雅的名字——Smart Framework，本章我们就一起来实现这个框架。

您将通过本章的学习，掌握如下技能：

- 如何快速搭建开发框架；
- 如何加载并读取配置文件；
- 如何实现一个简单的 IOC 容器；
- 如何加载指定的类；
- 如何初始化框架。

3.1 确定目标.....	68
3.2 搭建开发环境.....	70
3.2.1 创建框架项目	70
3.2.2 创建示例项目	73
3.3 定义框架配置项.....	74
3.4 加载配置项.....	75
3.5 开发一个类加载器.....	78
3.6 实现 Bean 容器	87
3.7 实现依赖注入功能.....	90
3.8 加载 Controller	93
3.9 初始化框架.....	97
3.10 请求转发器.....	98
3.11 总结.....	109

第 4 章 使框架具备 AOP 特性 110

在本章中，您将学到大量有用的技术，具体包括：

- 如何理解并使用代理技术；
- 如何使用 Spring 提供的 AOP 技术；
- 如何使用动态代理技术实现 AOP 框架；
- 如何理解并使用 ThreadLocal 技术；
- 如何理解数据库事务管理机制；
- 如何使用 AOP 框架实现事务控制。

4.1 代理技术简介.....	112
4.1.1 什么是代理	112
4.1.2 JDK 动态代理	114
4.1.3 CGLib 动态代理.....	116
4.2 AOP 技术简介.....	118

4.2.1	什么是 AOP	118
4.2.2	写死代码	119
4.2.3	静态代理	120
4.2.4	JDK 动态代理	121
4.2.5	CGLib 动态代理	122
4.2.6	Spring AOP	124
4.2.7	Spring + AspectJ	136
4.3	开发 AOP 框架	142
4.3.1	定义切面注解	142
4.3.2	搭建代理框架	143
4.3.3	加载 AOP 框架	150
4.4	ThreadLocal 简介	158
4.4.1	什么是 ThreadLocal	158
4.4.2	自己实现 ThreadLocal	161
4.4.3	ThreadLocal 使用案例	163
4.5	事务管理简介	172
4.5.1	什么是事务	172
4.5.2	事务所面临的问题	173
4.5.3	Spring 的事务传播行为	175
4.6	实现事务控制特性	178
4.6.1	定义事务注解	178
4.6.2	提供事务相关操作	181
4.6.3	编写事务代理切面类	182
4.6.4	在框架中添加事务代理机制	184
4.7	总结	185
第 5 章 框架优化与功能扩展		186

本章将对现有框架进行优化，并提供一些扩展功能。通过本章的学习，您可以了解到：

- 如何优化 Action 参数；
- 如何实现文件上传功能；
- 如何与 Servlet API 完全解耦；
- 如何实现安全控制框架；
- 如何实现 Web 服务框架。

5.1	优化 Action 参数	188
5.1.1	明确 Action 参数优化目标	188
5.1.2	动手优化 Action 参数使用方式	188
5.2	提供文件上传特性	191
5.2.1	确定文件上传使用场景	191
5.2.2	实现文件上传功能	194
5.3	与 Servlet API 解耦	214
5.3.1	为何需要与 Servlet API 解耦	214
5.3.2	与 Servlet API 解耦的实现过程	215
5.4	安全控制框架——Shiro	219
5.4.1	什么是 Shiro	219
5.4.2	Hello Shiro	220
5.4.3	在 Web 开发中使用 Shiro	224
5.5	提供安全控制特性	230
5.5.1	为什么需要安全控制	230
5.5.2	如何使用安全控制框架	231
5.5.3	如何实现安全控制框架	242
5.6	Web 服务框架——CXF	261
5.6.1	什么是 CXF	261
5.6.2	使用 CXF 开发 SOAP 服务	262
5.6.3	基于 SOAP 的安全控制	278
5.6.4	使用 CXF 开发 REST 服务	291
5.7	提供 Web 服务特性	308
5.8	总结	329
附录 A	Maven 快速入门	330
附录 B	将构件发布到 Maven 中央仓库	342



第 1 章

从一个简单的 Web 应用开始

如果您已经掌握了 Java 的基础知识，理解了面向对象的基本概念，对 Java Web 开发有过一定的了解，比如会用 Servlet 与 JSP 开发一些简单的应用程序，那么本章的内容就是为这类您准备的。

现在我们打算使用业界最牛的 Java 开发工具 IntelliJ IDEA（简称 IDEA）开发一个简单的 Web 应用，该应用不包含任何业务功能，只是为了熟悉 Web 应用的开发过程与 IDEA 的使用方法。同时，我们也会使用业界最强大的项目构建工具 Maven 与代码版本控制工具 Git，利用这些工具让开发过程更加高效。

正所谓“工欲善其事，必先利其器”，在正式开始设计并开发我们的轻量级 Java Web 框架之前，有必要先掌握以下技能：

- 使用 IDEA 搭建并开发 Java 项目；
- 使用 Maven 自动化构建 Java 项目；
- 使用 Git 管理项目源代码。

假设您已经在 Windows 上安装了 IDEA、Maven、Git 这三个工具，现在要做的就是双击 IDEA 图标来启动它，如图 1-1 所示。



图 1-1 IDEA 启动界面

实际上 IDEA 有两个版本，一个是社区版（开源的个人版），另外一个是完全版（收费的企业版）。个人版提供的功能对于我们而言是基本够用的，可能对于 Web 开发来说有些不太方便，但足以满足我们基本的开发需求。不管使用 IDEA 的哪个版本，本书都是适用的。

下面我们就使用 IDEA 来创建一个基于 Maven 的 Java Web 项目。

如果此时不知道 Maven 是什么，或者听说过但不会使用它，那么强烈建议您通过“附录 A”来了解 Maven 是什么以及它的基本用法。

1.1 使用 IDEA 创建 Maven 项目

1.1.1 创建 IDEA 项目

我们无须单独下载 Maven，更不用将其集成到 IDEA 中，因为 IDEA 默认已经将其整合了。在 IDEA 中创建 Maven 项目非常简单，只需要按照以下步骤进行即可：

- (1) 单击“Create New Project”按钮，弹出 New Project 对话框。
- (2) 选择 Maven 选项，单击“Next”按钮。
- (3) 输入 GroupId (org.smart4j)、ArtifactId (chapter1)、Version (1.0.0)，单击“Next”按钮。
- (4) 输入 Project name (chapter1)、Project location (C:\chapter1)，单击“Finish”按钮。

需要说明的是，其中 GroupId 建议为网站域名的倒排方式，以便确保唯一性，类似于 Java 的包名。

说明：本书中出现的 GroupId (org.smart4j) 所对应的域名 smart4j.org 是我购买的个人域名，您可使用自己喜欢的任意 GroupId，当然也可以与我保持一致。

按照以上操作，只需片刻之间，IDEA 就创建了一个基于 Maven 的目录结构，如图 1-2 所示。

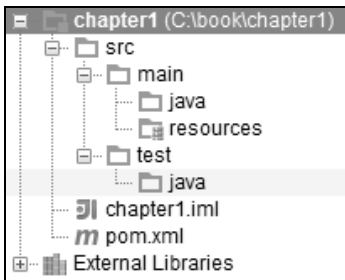


图 1-2 Maven 项目目录结构

1.1.2 调整 Maven 配置

当打开 Maven 项目配置文件 (pom.xml) 后，看到的应该是这样的配置（经笔者稍作整理）：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns=http://maven.apache.org/POM/4.0.0
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.smart4j</groupId>
  <artifactId>chapter1</artifactId>
  <version>1.0.0</version>

</project>
```

技巧：当调整 pom.xml 后，需单击右上角气泡中的 Import Changes，使 Maven 配置立即生效，此操作表示“手动生效”。也可以单击右上角气泡中的 Enable Auto-Import，一旦对 pom.xml 文件进行修改就会自动导入，此操作表示“自动生效”。不过建议还是使用前者，即“手动生效”方式，这样会更加可控一些，至少能够确定自己做过那些事情。

以上只是 pom.xml 的基本配置，下面需要为它添加一些常用配置。

首先，需要统一源代码的编码方式，否则使用 Maven 编译源代码的时候就会出现相关警告。一般情况下，我们都使用 UTF-8 进行编码，需要添加配置如下：

```
...
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
...
```

除了需要统一源代码编码方式以外，还需要统一源代码与编译输出的 JDK 版本。由于本书中使用 JDK 1.6 进行的开发，因此需要添加如下配置：

```
<build>
  <plugins>
    <!-- Compile -->
    <plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.3</version>
<configuration>
  <source>1.6</source>
  <target>1.6</target>
</configuration>
</plugin>
</plugins>
</build>
```

所有的 **plugin** 都需要添加到 `build/plugins` 标签下，Maven 插件或下文中所出现的 Maven 依赖都来自于 Maven 中央仓库，可以通过以下链接访问：

<http://search.maven.org/>

如果说上面添加的两个配置是必需的，那么下面这个配置应该就是可选的。如果在使用 Maven 打包时想跳过单元测试，那么可以添加如下插件：

```
<!-- Test -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.18.1</version>
  <configuration>
    <skipTests>true</skipTests>
  </configuration>
</plugin>
```

至此，一个 Maven 项目就搭建完毕了，下面我们就要在这个项目中添加有关 Java Web 的代码。

1.2 搭建 Web 项目框架

1.2.1 转为 Java Web 项目

目前，在 `pom.xml` 中还没有任何的 Maven 依赖（`dependency`），随后会添加一些 Java Web 所需的依赖，只不过在添加这些依赖之前，有必要先将这个 Maven 项目调整为 Web 项目结构。

提示：可以简单地将 Maven 依赖理解为 jar 包，只不过 Maven 依赖具备传递性，只需配置某个依赖，就能自动获取该依赖的相关依赖。

只需按照以下三步即可实现：

- (1) 在 main 目录下，添加 webapp 目录。
- (2) 在 webapp 目录下，添加 WEB-INF 目录。
- (3) 在 WEB-INF 目录下，添加 web.xml 文件。

此时，IDEA 给出一个提示，如图 1-3 所示。

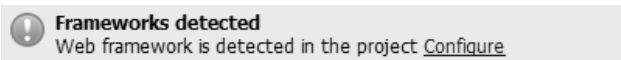


图 1-3 IDEA 框架检测提示

表示 IDEA 已经识别出目前我们使用了 Web 框架（即 Servlet 框架），需要进行一些配置才能使用。单击“Configure”按钮，会看到一个确认框，单击“OK”按钮即可将当前项目变为 Web 项目。

这里打算使用 Servlet 3.0 框架，所以在 web.xml 中添加如下代码：

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

</web-app>
```

实际上，使用 Servlet 3.0 框架是可以省略 web.xml 文件的，因为 Servlet 无须在 web.xml 里配置，只需通过 Java 注解的方式来配置即可，下面会描述具体的用法。

1.2.2 添加 Java Web 的 Maven 依赖

由于 Web 项目是需要打 war 包的，因此必须在 pom.xml 文件里设置 packaging 为 war（默认为 jar），配置如下：

```
<packaging>war</packaging>
```

接下来就需要添加 Java Web 所需的 Servlet、JSP、JSTL 等依赖了，添加如下配置：

```

<dependencies>
  <!-- Servlet -->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
  <!-- JSP -->
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.2</version>
    <scope>provided</scope>
  </dependency>
  <!-- JSTL -->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>

```

需要说明的是：

- (1) Maven 依赖的“三坐标”（groupId、artifactId、version）必须提供。
- (2) 如果某些依赖只需参与编译，而无须参与打包（例如，Tomcat 自带了 Servlet 与 JSP 所对应的 jar 包），可将其 scope 设置为 provided。
- (3) 如果某些依赖只是运行时需要，但无须参与编译（例如，JSTL 的 jar 包），可将其 scope 设置为 runtime。

为了便于阅读，以下是 pom.xml 的完整代码：

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

```

```
<modelVersion>4.0.0</modelVersion>

<groupId>org.smart4j</groupId>
<artifactId>chapter1</artifactId>
<version>1.0.0</version>
<packaging>war</packaging>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <!-- Servlet -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSP -->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.2</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
```

```
<plugins>
  <!-- Compile -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.3</version>
    <configuration>
      <source>1.6</source>
      <target>1.6</target>
    </configuration>
  </plugin>
  <!-- Test -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.18.1</version>
    <configuration>
      <skipTests>true</skipTests>
    </configuration>
  </plugin>
  <!-- Tomcat -->
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
      <path>/${project.artifactId}</path>
    </configuration>
  </plugin>
</plugins>
</build>

</project>
```

现在一个基于 Maven 的 Java Web 项目已搭建完毕，随后可进入具体的开发阶段。

学习任何技术，都需要从最容易入手的地方开始，下面我们利用一个简单的应用场景，一起学习 Servlet 3.0 框架的使用方法。

1.3 编写一个简单的 Web 应用

1.3.1 编写 Servlet 类

我们要做的是一件非常简单的事情：写一个 `HelloServlet`，接收 GET 类型的 `/hello` 请求，转发到 `/WEB-INF/jsp/hello.jsp` 页面，在 `hello.jsp` 页面上显示系统当前时间。

首先，需要在 `java` 目录下，创建一个名为 `org.smart4j.chapter1` 的包。

技巧：用鼠标选中 `java` 目录，按下 `Alt+Insert` 快捷键，选择 `Package` 选项，输入具体的包名，回车后即可创建包，用同样的方法也可以创建类或其他文件。

然后，在该包下创建一个 `HelloServlet` 的类，代码如下：

```
package org.smart4j.chapter1;

public class HelloServlet {
}
```

最后，我们需要为该类添加一些代码：

```
package org.smart4j.chapter1;

import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

    @Override
```

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String currentTime = dateFormat.format(new Date());
    req.setAttribute("currentTime", currentTime);
    req.getRequestDispatcher("/WEB-INF/jsp/hello.jsp").forward(req,
    resp);
}
}
```

对以上代码进行一些说明:

- (1) 继承 `HttpServlet`, 让它成为一个 `Servlet` 类。
- (2) 覆盖父类的 `doGet` 方法, 用于接收 GET 请求。
- (3) 在 `doGet` 方法中获取系统当前时间, 并将其放入 `HttpServletRequest` 对象中, 最后转发到 `/WEB-INF/jsp/hello.jsp` 页面。
- (4) 使用 `WebServlet` 注解并配置请求路径, 对外发布 `Servlet` 服务。

不需要在 `web.xml` 中添加任何的 `Servlet` 配置, 因为我们使用了 `Servlet 3.0` 规范提供的 `WebServlet` 注解。除此以外, 还有很多的注解或 API, 会让我们拥有一个“零配置”的 `web.xml`。

技巧:

(1) 在 `HelloServlet` 类中使用 `Alt+Insert` 快捷键, 选择“`Qverrides Methods...`”选项, 选择 `javax.servlet.http.HttpServlet` 中的 `doGet` 方法(可输入首字母进行快速查找), 最后单击“OK”按钮或回车, 可快速添加 `doGet` 方法模版。

(2) 将光标定位在某个位导入的类上(`HttpServlet`), 使用 `Alt + Insert` 快捷键, 可快速导入包名(`javax.servlet.annotation.WebServlet`)。

(3) 怎样在 IDEA 中使用“代码提示”快捷键? 我们需要在 `Settings`→`Appearance & Behavior` → `Keymap` 里设置, 路径为 `Main menu / Code / Completion / Basic`, 默认为 `Ctrl + 空格键`, 它与切换中英文输入法是有冲突的, 建议将其修改为自己最习惯的快捷键, 例如, `Alt + /` (与 `Eclipse` 相同)。

1.3.2 编写 JSP 页面

`HelloServlet` 已经完成了, 我们接着写一个 `hello.jsp`。在 `webapp` 目录下创建 `jsp` 目录, 并在该目录下创建 `hello.jsp`, 编写如下代码:

```
<%@ page pageEncoding="UTF-8" %>
<html>
<head>
    <title>Hello</title>
</head>
<body>

<h1>Hello!</h1>

<h2>当前时间: ${currentTime}</h2>

</body>
</html>
```

可见，我们使用了 JSTL 表达式来获取从 `HelloServlet` 里传递过来的 `currentTime` 请求属性。至此，所有的代码已编写完毕，我们即将让这个 Web 应用跑起来！

1.4 让 Web 应用跑起来

1.4.1 在 IDEA 中配置 Tomcat

首先，我们需要在 IDEA 里配置一个 Tomcat，详细过程如下：

(1) 单击 IDEA 的工具栏上的“Edit Configurations...”（在一个下拉框里，一眼就能看到它），弹出 Run/Debug Configurations 对话框。

(2) 单击左上角的“+”按钮（或使用 Alt+Insert 快捷键），选择 Tomcat Server→Local 选项。

(3) 输入 Tomcat 的 Name（例如：tomcat），取消勾选“After launch”选项。

(4) 单击 Application server 下拉框右侧的“Configure...”按钮，配置一个 Tomcat，配置完毕后，在下拉框中选择该 Tomcat。

(5) 切换到 Deployment 选项卡，单击右边的“+”按钮（或使用 Alt+Insert 快捷键），选择“Artifact...”选项，弹出 Select Artifact to Deploy 对话框。

(6) 选择 chapter1:war exploded，单击“OK”按钮或回车，关闭 Select Artifact to Deploy 对话框。

(7) 回到 Run/Debug Configurations 对话框，在 Application context 中输入 /chapter1。

(8) 切换回 Server 选项卡, 在 On frame deactivation 下拉框中选择“Update resources”选项, 单击“OK”按钮, 完成所有配置, 关闭 Run/Debug Configurations 对话框。

然后, 单击 IDEA 工具栏上的“Run”或“Debug”按钮, 启动 Tomcat 并部署 Web 应用。

技巧: 在开发阶段建议使用 Debug 方式运行 Tomcat, 这样在代码中所做的修改可进行自动部署(热部署), 只需使用 Ctrl + F9 键手工编译即可。不过有些情况下是无法进行热部署的, 例如, 修改了类名、方法名、成员变量名等。

1.4.2 使用 Tomcat 的 Maven 插件

如果我们当前使用的是 IDEA 的社区版, 该版本并没有提供集成 Tomcat 的功能, 那么我们怎么在 IDEA 中启动 Tomcat 并实现 Debug 功能呢?

我们可以使用 Tomcat 的 Maven 插件, 并在 IDEA 中以插件的方式启动 Tomcat。只需在 pom.xml 中添加如下配置:

```
<!-- Tomcat -->
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
  <configuration>
    <path>/${project.artifactId}</path>
  </configuration>
</plugin>
```

打开 IDEA 的 Maven 面板, 双击 tomcat7:run 命令, 即可运行 Maven 命令(mvn tomcat7:run), 如图 1-4 所示。

此时, 若我们尝试在 hello.jsp 中修改部分代码, 刷新浏览器, 将看到相应的调整, 但修改了 HelloServlet 并编译后, 并不能实现热部署, 此外, 也不能设置断点进行调试。

此方法仅用于运行 Maven 命令, 并不能与 IDEA 整合, 难道就没有更好的解决方案了吗?

一定要相信强大的 IDEA, 它是绝对不会让我们失望的!

1.4.3 以 Debug 方式运行程序

我们在 IDEA 社区版中添加一个 Maven 的 Configuration (具体操作与添加 Tomcat 类似),

在 Name 中输入 tomcat，在 Command line 中输入 tomcat7:run，回车后关闭对话框后，我们就可以单击“Run”或“Debug”按钮来启动 Tomcat 并部署应用了。

使用这种方法，不仅修改的 JSP 可立即生效，也可以实现类的热部署，而且还可以使用断点调试。

最后，打开我们最喜欢的浏览器，输入以下地址：<http://localhost:8080/chapter1/hello>。

此时，我们就会看到想要的效果，如图 1-5 所示。

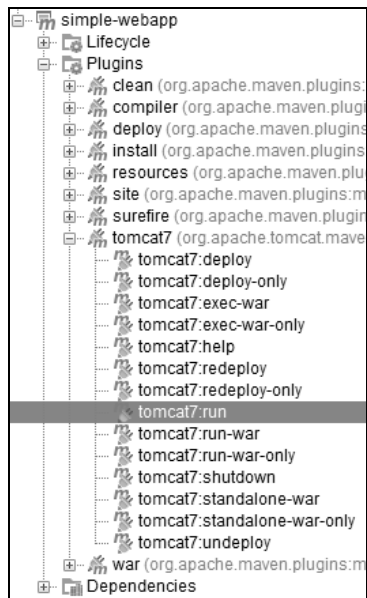


图 1-4 在 IDEA 中执行 Maven 命令

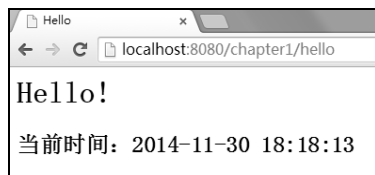


图 1-5 Hello 应用运行效果

1.5 将代码放入 Git 仓库中

1.5.1 编写.gitignore 文件

现在项目框架已经搭建完毕，我们有必要将代码放入 Git 仓库中进行版本控制管理。

有些文件是不需要放入 Git 中的，比如 Maven 生成的 target 目录、IDEA/Eclipse 的工程文件。

在项目的根目录（C:）下添加一个名为.gitignore 的文件，内容如下：

```
# Maven #
target/
```

```
# IDEA #  
.idea/  
*.iml  
  
# Eclipse #  
.settings/  
.metadata/  
.classpath  
.project  
Servers/
```

1.5.2 提交本地 Git 仓库

在 IDEA 中找到 VCS 菜单，单击“Import into Version Control/Create Git Repository...”菜单项，在弹出的对话框中选中项目的根目录，单击“OK”按钮关闭对话框。此时，IDEA 就为我们在本地创建了一个 Git 仓库。

选中项目根目录，在 VCS 菜单（或右键菜单）中单击“Git/Add”菜单项，可将除.gitignore 中忽略的所有文件添加到本地 Git 仓库中。当然，也可以使用 Ctrl+Alt+A 键来完成，只需选择需要提交的文件，然后使用该快捷键即可。若此时开启了 QQ，则需修改它的屏幕截图快捷键（建议修改为 Ctrl+Alt+Z），否则将引起快捷键冲突，导致 IDEA 无法使用该快捷键。

在 VCS 菜单（或右键菜单）中单击 Git/Commit Directory...菜单项，随后将弹出一个对话框，确认是否进行提交操作，输入 Commit Message 后，单击“Commit”按钮就可以提交代码到本地 Git 仓库了。当然，也可以单击工具栏中的“Commit Changes”按钮，或使用 Ctrl+K 键来做同样的事情。

提示：在提交代码时，建议勾选“Optimize imports”选项，它可优化 import 语句，去掉没有使用的包。

1.5.3 推送远程 Git 仓库

可以随时将 Git 本地仓库推送到远程仓库，只需在 IDEA 中的 VCS 菜单中单击 Git/Push 菜单项即可，或者使用 Ctrl+Shift+K 键。需要注意的是，在推送之前，必须将本地仓库与远程仓库建立一个连接。

我们可以免费使用 OSC（开源中国）提供的 Git 远程仓库，首先需要在 Git@OSC（<http://git.oschina.net/>）中创建一个项目，执行如下命令来完成推送：

```
git remote add origin <您的 Git 仓库地址>
git push -u origin master
```

现在就可以在 Git@OSC 中看到已推送的代码了。

关于“开源中国”开源中国成立于 2008 年 8 月，是目前国内最大的开源技术社区，拥有超过 200 万会员，有开源软件库、代码分享、资讯、协作翻译、讨论区和博客等几大频道内容，为 IT 开发者提供了一个发现、使用并交流开源技术的平台。2013 年，开源中国建立大型综合性的云开发平台——中国源，为中国广大开发者提供团队协作、源码托管、代码质量分析、代码评审、测试、代码演示平台等功能。

1.5.4 总结

在本章中，我们使用 IDEA 开发了一个简单的 Web 应用，学会了：

- （1）如何在 IDEA 中创建 Maven 项目。
- （2）如何将普通的 Maven 项目转为 Java Web 项目。
- （3）如何在 IDEA 中集成 Tomcat（提供了三种方法）。
- （4）如何将代码提交到本地 Git 仓库并推送到远程 Git 仓库。

我们现在已经熟悉了 Java Web 开发的常用工具与技巧，现在仅仅是一个 Servlet+JSP 而已，并没有与数据库打交道，我们在下一章里会编写更多的 Servlet 与 JSP，此外还会使用 JDBC 与数据库进行交互，一个真正的 Web 应用即将诞生。



第 2 章

为 Web 应用添加业务功能

经过我们上一章的努力，一个简单的 Web 应用基本完成了，但目前只能通过 Servlet 处理简单的请求，并没有实现具体的业务逻辑。

现在我们将在这个应用的基础上增加一些业务功能，您将学会更多有关项目实战的技能，具体包括：

- 如何进行需求分析；
- 如何进行系统设计；
- 如何编写应用程序。

此外，我们将使用一些代码重构的技巧，不断优化现有代码。好的程序不是一次性写出来的，而是不断地“改”出来的，这里的“改”就是重构。

我们现在就从需求分析开始。

2.1 需求分析与系统设计

2.1.1 需求分析

需求通常由用户或产品经理提出，例如以下需求描述：

- (1) 当用户进入“客户管理”模块时，可通过列表方式来查看所有客户。
- (2) 可通过“客户名称”关键字进行模糊查询。
- (3) 单击客户列表中的“客户名称”链接，可查看客户基本信息。
- (4) 单击“新增”按钮，进入“新增客户”界面，可新增客户基本信息。
- (5) 单击客户列表中的“编辑”按钮，进入“编辑客户”界面，可更新客户基本信息。
- (6) 单击客户列表中的“删除”按钮，可删除当前所选择的客户，需提示是否删除。

其中，客户基本信息包括：客户名称、联系人、电话号码、邮箱地址、备注。

以上这些信息告诉我们，用户期望我们提供一个“客户管理”模块，其中包含了对客户的新增、修改、删除等功能。

当理解了这些需求以后，我们就可以从“需求阶段”进入到“设计阶段”了。

说明：实际上，需求分析是一个很复杂的过程，内容已超出本书范围，请有兴趣的朋友自行学习。

下面的设计阶段是从需求到开发之间的桥梁，也就是说，如果设计做得不好，后面的开发一定会受到影响，甚至整个项目都会延期。既然设计阶段如此重要，那么我们应该怎样做设计呢？

2.1.2 系统设计

我们先从原始需求进行分析，找出需求中涉及的 Use Case（用例），然后设计表结构，画原型图，定义 URL 规范，就这样一步步地完成这些设计工作。

可见，设计过程是由粗到细、由表及里的，但需要注意的是，设计阶段不涉及具体的技术实现，因为后面才是真正的开发阶段。

下面我们就从设计用例开始吧！

1. 设计用例

从需求中，我们不难找到如下用例：

- 查询客户；
- 显示客户列表；
- 显示客户基本信息；
- 创建客户；
- 编辑客户；
- 删除客户。

最好可以用一张 UML 的“用例图”来描绘以上用例，这样效果会更好，因为这些用例一方面是为了指导后续的设计与开发，另一方面可以更加方便客户来确认需求。

2. 设计表结构

根据需求，我们可以很轻松地找到一个核心的业务实体，那就是“客户”，它对应于 customer 表，其结构如表 2-1 所示。

表 2-1 customer 表结构

字 段 名	数 据 类 型	是 否 非 空	字 段 描 述
id	BIGINT	√	ID（自增主键）
name	VARCHAR(255)	√	客户名称
contact	VARCHAR(255)	√	联系人
telephone	VARCHAR(255)	—	电话号码
email	VARCHAR(255)	—	邮箱地址
remark	TEXT	—	备注

关于表结构的设计，有如下建议：

- 建议表名与字段名均为小写，若多个单词可用“下划线”分割；
- 建议每张表都有唯一的主键字段，且字段名都为 id，可使用自增主键；
- 数据类型尽可能统一，不要出现太多的数据类型。

3. 设计界面原型

使用 Balsamiq Mockups 软件，我们可以快速地画出界面原型，如图 2-1 所示。

说明：Balsamiq Mockups 是一款绘制界面原型的商业软件，可以作为与用户交互的一个界面草图，也可以作为美工开发 HTML 的原型使用。



图 2-1 界面原型

当然，还有其他相关的用户界面，比如查看客户、创建客户、编辑客户等，这些都需要在设计文档中体现出来。

4. 设计 URL

通过界面之间的跳转与操作，我们可以分析出以下 URL，如表 2-2 所示。

表 2-2 URL 表

序 号	URL	描 述
1	GET:/customer	进入“客户列表”界面
2	POST:/customer_search	查询客户
3	GET:/customer_show?id={id}	进入“查看客户”界面
4	GET:/customer_create	进入“创建客户”界面
5	POST:/customer_create	创建客户
6	GET:/customer_edit?id={id}	进入“编辑客户”界面
7	PUT:/customer_edit?id={id}	编辑客户
8	DELETE:/customer_delete?id={id}	删除客户

注意：在设计阶段里一般不涉及任何的编码工作，写这些设计文档是为了让我们的思路更加清晰，也给我们后续的开发工作提供帮助。此外，这里定义的 URL 没有使用经典的 RESTful 风格，只是为了更容易地表达实际的业务含义。

当然，以上设计阶段还是远远不够的，我们还会进行其他设计过程，比如数据模型、业务流程等，由于本书篇幅有限，这里就不再描述了。

当设计阶段完成以后，也就意味着一份设计文档也完成了。随后，我们需要将关注点从设计阶段转移到开发阶段，所谓开发阶段就是需要我们动手编码的阶段。

相信您早已迫不及待了，我们马上就进入开发阶段了。

2.2 动手开发 Web 应用

开发阶段与设计阶段也有类似的地方，它也是由粗到细、由表及里的，千万不要一开始就

陷入到具体的实现细节中，我们不妨先搭建一个开发框架，然后用一系列的“TODO”来标明我们下一步要做的事情，也就是待办任务了，然后再一步步地去填充这些待办任务。

既然我们已经在设计阶段中定义了一系列的表结构，那么接下来要做的第一件事情就是创建数据库，这个数据库用于存放相应的业务数据表。

2.2.1 创建数据库

我们需要在 MySQL 中创建一个名为 demo 的数据库，编码方式最好统一为 UTF-8，以免编码不一致，从而导致中文乱码。

可使用如下方法创建数据库：

(1) 打开 MySQL 客户端工具 Navicat。

(2) 在 localhost 上单击右键，并选择“New Database...”菜单项，弹出 New Database 对话框。

(3) 输入 Database Name (demo)、Character set (utf8 -- UTF-8 Unicode)，单击“OK”按钮。

说明：Navicat 是一款商业的数据库客户端软件，我们可用它来创建数据库、设计表结构、查看并编辑表中的数据，以及执行 SQL 脚本等。当然，它还提供了很多数据库管理功能，包括用户与权限管理、备份与还原、导入与导出等，它也是笔者最喜欢的数据库客户端软件。

2.2.2 准备开发环境

在 IDEA 中创建一个新的 Maven 项目，名为 chapter2，它是一个 Web 项目，创建方法可以参考上一章的具体操作方法，这里就不再赘述了。

以下是生成的 pom.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.smart4j</groupId>
```

```

<artifactId>chapter2</artifactId>
<version>1.0.0</version>
<packaging>war</packaging>

</project>

```

稍后我们会在该文件中添加一些 **dependency**，让项目依赖一些有用的 **jar** 包。

现在我们在 **java** 目录下创建一个 **org.smart4j.chapter2** 包，并在正确的路径下创建一个 **web.xml** 文件。

当前的项目结构如图 2-2 所示。

为了使代码层次结构更加清晰，我们在 **org.smart4j.chapter2** 包下添加如图 2-3 所示的子包。

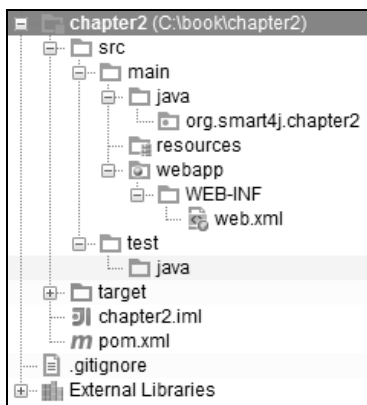


图 2-2 初始目录结构

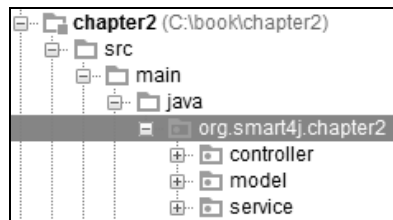


图 2-3 代码结构分层

需要说明的是，我们采用了 **MVC** 架构来搭建 **Web** 应用项目结构，**MVC** 即 **Model**（模型）、**View**（视图）、**Controller**（控制器），这是一种分层思想，它将应用程序分成若干层，让每一层都能做好自己的事情，责任更加清晰，这也是设计原则之“单一职责原则”所提倡的理念。

2.2.3 编写模型层

根据表结构的定义，我们创建如图 2-4 所示的模型类。

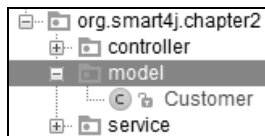


图 2-4 模型类

```
/**
 * 客户
 */
public class Customer {

    /**
     * ID
     */
    private long id;

    /**
     * 客户名称
     */
    private String name;

    /**
     * 联系人
     */
    private String contact;

    /**
     * 电话号码
     */
    private String telephone;

    /**
     * 邮箱地址
     */
    private String email;

    /**
     * 备注
     */
    private String remark;

    // 省略 getter/setter 方法
}
```

技巧：以上 getter/setter 方法可以使用 Alt+Insert 键自动生成，而且这个快捷键在很多场景下都会使用，比如创建类、创建包、创建任意的文件、实现接口的方法、覆盖父类的方法等。

模型类编写完毕后，我们顺便在 demo 数据库里创建一个名称为 customer 的表，当然也可以使用如下 SQL 语句：

```
CREATE TABLE `customer` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `contact` varchar(255) DEFAULT NULL,
  `telephone` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `remark` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

顺便在 customer 表中创建几条记录，执行如下 SQL 语句即可：

```
INSERT INTO `customer` VALUES ('1', 'customer1', 'Jack', '13512345678',
'jack@gmail.com', null);
INSERT INTO `customer` VALUES ('2', 'customer2', 'Rose', '13623456789',
'rose@gmail.com', null);
```

2.2.4 编写控制器层

由于我们使用了 Servlet 作为控制器层的实现技术，下面的工作就是根据 URL 的定义来编写具体的 Servlet，如图 2-5 所示。

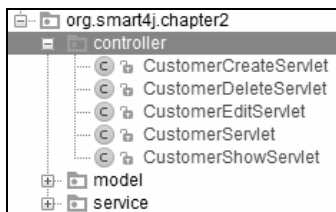


图 2-5 控制器层

Servlet 比较多，以 CustomerCreateServlet 为例进行描述：

```
package org.smart4j.chapter2.controller;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * 创建客户
 */
@WebServlet("/customer_create")
public class CustomerCreateServlet extends HttpServlet {

    /**
     * 进入 创建客户 界面
     */
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // TODO
    }

    /**
     * 处理 创建客户 请求
     */
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // TODO
    }
}
```

需要注意的是，该 Servlet 只有一个请求路径，但可以处理两种不同的请求类型。doGet 用于处理 GET 请求，而 doPost 用于处理 POST 请求，它们对应的请求路径都是/customer_create。

在 Servlet 中，请求类型有 GET、POST、PUT、DELETE，它们分别对应 doGet、doPost、doPut、doDelete 方法。

2.2.5 编写服务层

在标准的 MVC 架构中是没有服务层的，我们将该层作为衔接控制器层与数据库之间的桥梁，可以使用接口和实现类来表达，在简单情况下，无须使用接口，直接用类就可以了。并非在所有情况下都需要定义一个接口，要根据实际情况来做出选择。服务层如图 2-6 所示。

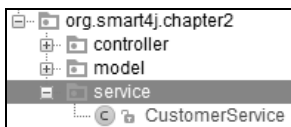


图 2-6 服务层

其中 CustomerService 代码如下：

```

package org.smart4j.chapter2.service;

import java.util.List;
import java.util.Map;
import org.smart4j.chapter2.model.Customer;

/**
 * 提供客户数据服务
 */
public class CustomerService {

    /**
     * 获取客户列表
     */
    public List<Customer> getCustomerList(String keyword) {
        // TODO
        return null;
    }

    /**
     * 获取客户
     */
    public Customer getCustomer(long id) {
        // TODO
        return null;
    }
}

```

```
    }

    /**
     * 创建客户
     */
    public boolean createCustomer(Map<String, Object> fieldMap) {
        // TODO
        return false;
    }

    /**
     * 更新客户
     */
    public boolean updateCustomer(long id, Map<String, Object> fieldMap) {
        // TODO
        return false;
    }

    /**
     * 删除客户
     */
    public boolean deleteCustomer(long id) {
        // TODO
        return false;
    }
}
```

2.2.6 编写单元测试

我们使用 JUnit 作为单元测试框架，需要在 pom.xml 中添加如下依赖：

```
<!-- JUnit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
```

在 `test/java` 目录下创建一个单元测试类 `org.smart4j.chapter2.test.CustomerServiceTest`，用于测试 `CustomerService` 类中的各个方法，如图 2-7 所示。

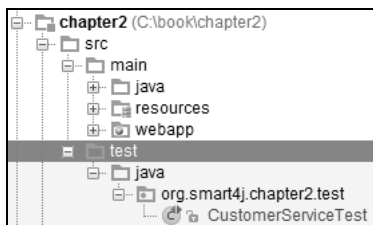


图 2-7 单元测试

```
package org.smart4j.chapter2.test;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.smart4j.chapter2.model.Customer;
import org.smart4j.chapter2.service.CustomerService;

/**
 * CustomerService 单元测试
 */
public class CustomerServiceTest {

    private final CustomerService customerService;

    public CustomerServiceTest() {
        customerService=new CustomerService();
    }

    @Before
    public void init() {
        // TODO 初始化数据库
    }

    @Test
```

```
public void getCustomerListTest() throws Exception {
    List<Customer> customerList=customerService.getCustomerList();
    Assert.assertEquals(2, customerList.size());
}

@Test
public void getCustomerTest() throws Exception {
    long id=1;
    Customer customer=customerService.getCustomer(id);
    Assert.assertNotNull(customer);
}

@Test
public void createCustomerTest() throws Exception {
    Map<String, Object> fieldMap=new HashMap<String, Object>();
    fieldMap.put("name", "customer100");
    fieldMap.put("contact", "John");
    fieldMap.put("telephone", "13512345678");
    boolean result=customerService.createCustomer(fieldMap);
    Assert.assertTrue(result);
}

@Test
public void updateCustomerTest() throws Exception {
    long id=1;
    Map<String, Object> fieldMap=new HashMap<String, Object>();
    fieldMap.put("contact", "Eric");
    boolean result=customerService.updateCustomer(id, fieldMap);
    Assert.assertTrue(result);
}

@Test
public void deleteCustomerTest() throws Exception {
    long id=1;
    boolean result=customerService.deleteCustomer(id);
    Assert.assertTrue(result);
}
}
```

2.2.7 编写视图层

使用 JSP 充当视图层，在 WEB-INF/view 目录下存放所有的 JSP 文件，如图 2-8 所示。

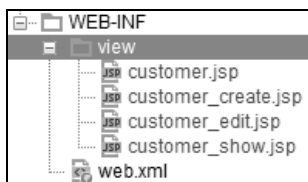


图 2-8 视图层

需要注意的是，推荐将 JSP 放到 WEB-INF 内部，而并非外部，因为用户无法通过浏览器地址栏直接请求放在内部的 JSP，必须通过 Servlet 程序进行转发（forward）或重定向（redirect）。

以“创建客户”为例，在 customer_create.jsp 中编写如下代码：

```
<%@ page pageEncoding="UTF-8" %>
<html>
<head>
  <title>客户管理 - 创建客户</title>
</head>
<body>

<h1>创建客户界面</h1>

<%-- TODO --%>

</body>
</html>
```

至此，一个简单的 Web 应用基本开发完毕，此时的应用程序只能算是一个“毛坯房”，我们还需要对它进行装修，让它更加实用。

2.3 细节完善与代码优化

我们根据一个业务场景，搭建了一个“客户管理”模块的代码框架，在代码中故意预留了许多“TODO”，这些就是将要逐步完善的细节，我们将分别完善服务层、控制器层、视图层，并且对代码不断地进行优化。

2.3.1 完善服务层

添加 SLF4J 依赖，用于提供日志 API，使用 Log4J 作为实现，配置如下：

```
<!-- SLF4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.7</version>
</dependency>
```

为了让 Log4J 起作用，必须在 main/resources 目录下创建一个名为 log4j.properties 的文件，内容如下：

```
log4j.rootLogger=ERROR,console,file

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%m%n

log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
log4j.appender.file.File=${user.home}/logs/book.log
log4j.appender.file.DatePattern='_'yyyyMMdd
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{HH:mm:ss,SSS} %p %c (%L) -
%m%n

log4j.logger.org.smart4j=DEBUG
```

我们将日志级别设置为 ERROR，并且提供了两种日志 appender，分别是 console 与 file。需要对这两种日志分别配置，然后指定只有 org.smart4j 包下的类才能输出 DEBUG 级别的日志。添加 MySQL 依赖，用于提供 JDBC 实现，配置如下：

```
<!-- MySQL -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.33</version>
  <scope>runtime</scope>
</dependency>
```

添加两个 Apache Commons 依赖，用于提供常用的工具类，配置如下：

```
<!-- Apache Commons Lang -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.3.2</version>
</dependency>
<!-- Apache Commons Collections -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-collections4</artifactId>
  <version>4.0</version>
</dependency>
```

打开 CustomerService，下面开始实现 getCustomerList 方法。

该方法需要获取 List，也就是所有的客户信息（此时并没有考虑分页，后续会增加此特性），我们需要执行一条 select 语句（Statement），获取相应的结果集（ResultSet），然后将其放入一个 List 中。不过在查询之前，我们必须先获取数据库连接（Connection）。

我们可使用 JDBC 轻松完成以上所有操作。为了让数据库信息可配置，我们在 classpath 下创建一个 config.properties 文件，该文件位于 src/main/resources 目录中。

注意：对于 Maven 目录结构而言，classpath 指的是 java 与 resources 这两个根目录。

config.properties 文件的内容如下：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/demo
jdbc.username=root
jdbc.password=root
```

既然属性文件已经准备好了，那么就需要有一个类来读取该文件，我们需要编写一个 PropsUtil 工具类来完成这件事情：

```
package org.smart4j.chapter2.util;

import java.io.FileNotFoundException;
import java.io.IOException;
```

```
import java.io.InputStream;
import java.util.Properties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 属性文件工具类
 */
public final class PropsUtil {

    private static final Logger LOGGER=LoggerFactory.getLogger(PropsUtil.
class);

    /**
     * 加载属性文件
     */
    public static Properties loadProps(String fileName) {
        Properties props=null;
        InputStream is=null;
        try {
            is=Thread.currentThread().getContextClassLoader().getResou-
rceAsStream(fileName);
            if (is == null) {
                throw new FileNotFoundException(fileName + " file is not found");
            }
            props=new Properties();
            props.load(is);
        } catch (IOException e) {
            LOGGER.error("load properties file failure", e);
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    LOGGER.error("close input stream failure", e);
                }
            }
        }
    }
}
```

```
        return props;
    }

    /**
     * 获取字符型属性（默认值为空字符串）
     */
    public static String getString(Properties props, String key) {
        return getString(props, key, "");
    }

    /**
     * 获取字符型属性（可指定默认值）
     */
    public static String getString(Properties props, String key, String
    defaultValue) {
        String value=defaultValue;
        if (props.containsKey(key)) {
            value=props.getProperty(key);
        }
        return value;
    }

    /**
     * 获取数值型属性（默认值为 0）
     */
    public static int getInt(Properties props, String key) {
        return getInt(props, key, 0);
    }

    // 获取数值型属性（可指定默认值）
    public static int getInt(Properties props, String key, int defaultValue) {
        int value=defaultValue;
        if (props.containsKey(key)) {
            value=CastUtil.castInt(props.getProperty(key));
        }
        return value;
    }
}
```

```
/**
 * 获取布尔型属性（默认值为 false）
 */
public static boolean getBoolean(Properties props, String key) {
    return getBoolean(props, key, false);
}

/**
 * 获取布尔型属性（可指定默认值）
 */
public static boolean getBoolean(Properties props, String key, Boolean
defaultValue) {
    boolean value=defaultValue;
    if (props.containsKey(key)) {
        value=CastUtil.castBoolean(props.getProperty(key));
    }
    return value;
}
}
```

其中，最关键的是 `loadProps` 方法，我们只需传递一个属性文件的名称，即可返回一个 `Properties` 对象，然后再根据 `getString`、`getInt`、`getBoolean` 这些方法由 `key` 获取指定类型的 `value`，同时也可指定 `defaultValue` 作为默认值。

在 `PropsUtil` 类中，我们用到了 `CastUtil` 类，该类是为处理一些数据转型操作而准备的，代码如下：

```
package org.smart4j.chapter2.util;

/**
 * 转型操作工具类
 */
public final class CastUtil {

    /**
     * 转为 String 型
     */
    public static String castString(Object obj) {
        return CastUtil.castString(obj, "");
    }
}
```

```
}

/**
 * 转为 String 型 (提供默认值)
 */
public static String castString(Object obj, String defaultValue) {
    return obj != null ? String.valueOf(obj) : defaultValue;
}

/**
 * 转为 double 型
 */
public static double castDouble(Object obj) {
    return CastUtil.castDouble(obj, 0);
}

/**
 * 转为 double 型 (提供默认值)
 */
public static double castDouble(Object obj, double defaultValue) {
    double doubleValue=defaultValue;
    if (obj != null) {
        String strValue=castString(obj);
        if (StringUtil.isNotEmpty(strValue)) {
            try {
                doubleValue=Double.parseDouble(strValue);
            } catch (NumberFormatException e) {
                doubleValue=defaultValue;
            }
        }
    }
    return doubleValue;
}

/**
 * 转为 long 型
 */
public static long castLong(Object obj) {
```

```
        return CastUtil.castLong(obj, 0);
    }

    /**
     * 转为 long 型（提供默认值）
     */
    public static long castLong(Object obj, long defaultValue) {
        long longValue=defaultValue;
        if (obj != null) {
            String strValue=castString(obj);
            if (StringUtil.isNotEmpty(strValue)) {
                try {
                    longValue=Long.parseLong(strValue);
                } catch (NumberFormatException e) {
                    longValue=defaultValue;
                }
            }
        }
        return longValue;
    }

    /**
     * 转为 int 型
     */
    public static int castInt(Object obj) {
        return CastUtil.castInt(obj, 0);
    }

    /**
     * 转为 int 型（提供默认值）
     */
    public static int castInt(Object obj, int defaultValue) {
        int intValue=defaultValue;
        if (obj != null) {
            String strValue=castString(obj);
            if (StringUtil.isNotEmpty(strValue)) {
                try {
                    intValue=Integer.parseInt(strValue);
                } catch (NumberFormatException e) {
                    intValue=defaultValue;
                }
            }
        }
        return intValue;
    }
}
```

```

        } catch (NumberFormatException e) {
            intValue=defaultValue;
        }
    }
}
return intValue;
}

/**
 * 转为 boolean 型
 */
public static boolean castBoolean(Object obj) {
    return CastUtil.castBoolean(obj, false);
}

/**
 * 转为 boolean 型 (提供默认值)
 */
public static boolean castBoolean(Object obj, boolean defaultValue) {
    boolean booleanValue=defaultValue;
    if (obj != null) {
        booleanValue=Boolean.parseBoolean(castString(obj));
    }
    return booleanValue;
}
}

```

在 `CastUtil` 类中，我们用到了 `StringUtil` 类，它用于提供一些字符串操作，代码如下：

```

package org.smart4j.chapter2.util;

import org.apache.commons.lang3.StringUtils;

/**
 * 字符串工具类
 */
public final class StringUtil {

    /**

```

```
    * 判断字符串是否为空
    */
    public static boolean isEmpty(String str) {
        if (str != null) {
            str=str.trim();
        }
        return StringUtils.isEmpty(str);
    }

    /**
    * 判断字符串是否非空
    */
    public static boolean isEmpty(String str) {
        return !isEmpty(str);
    }
}
```

以上我们只是对 Apache Commons 类库做了一个简单的封装。同理，也可以做一个 CollectionUtil，用于提供一些集合操作，代码如下：

```
package org.smart4j.chapter2.util;

import java.util.Collection;
import java.util.Map;
import org.apache.commons.collections4.CollectionUtils;
import org.apache.commons.collections4.MapUtils;

/**
 * 集合工具类
 */
public final class CollectionUtil {

    /**
    * 判断 Collection 是否为空
    */
    public static boolean isEmpty(Collection<?> collection) {
        return CollectionUtils.isEmpty(collection);
    }
}
```

```

/**
 * 判断 Collection 是否非空
 */
public static boolean isEmpty(Collection<?> collection) {
    return !isEmpty(collection);
}

/**
 * 判断 Map 是否为空
 */
public static boolean isEmpty(Map<?, ?> map) {
    return MapUtils.isEmpty(map);
}

/**
 * 判断 Map 是否非空
 */
public static boolean isEmpty(Map<?, ?> map) {
    return !isEmpty(map);
}
}

```

我们一口气写了四个工具类，每个工具类的分工各不相同，它们在后面还会经常用到，我们也会不断地完善这些工具类。

现在回到 **CustomerService**，我们需要在该类中执行数据库操作，也就是需要编写一些 JDBC 的代码，首先使用 **PropsUtil** 读取 **config.properties** 配置文件，获取与 JDBC 相关的配置项。

我们不妨在 **CustomerService** 中为这些配置项定义一些常量，并提供一个“静态代码块”来初始化这些常量，就像下面这样：

```

public class CustomerService {

    private static final String DRIVER;
    private static final String URL;
    private static final String USERNAME;
    private static final String PASSWORD;

    static {
        Properties conf=PropsUtil.loadProps("config.properties");
    }
}

```

```
DRIVER=conf.getProperty("jdbc.driver");
URL=conf.getProperty("jdbc.url");
USERNAME=conf.getProperty("jdbc.username");
PASSWORD=conf.getProperty("jdbc.password");

try {
    Class.forName(DRIVER);
} catch (ClassNotFoundException e) {
    LOGGER.error("can not load jdbc driver", e);
}
}
```

以 `getCustomerList` 为例，我们可以这样写 JDBC 代码：

```
/**
 * 获取客户列表
 */
public List<Customer> getCustomerList() {
    Connection conn=null;
    try {
        List<Customer> customerList=new ArrayList<Customer>();
        String sql="SELECT * FROM customer";
        conn=DriverManager.getConnection(URL, USERNAME, PASSWORD);
        PreparedStatement stmt=conn.prepareStatement(sql);
        ResultSet rs=stmt.executeQuery();
        while (rs.next()) {
            Customer customer=new Customer();
            customer.setId(rs.getLong("id"));
            customer.setName(rs.getString("name"));
            customer.setContact(rs.getString("contact"));
            customer.setTelephone(rs.getString("telephone"));
            customer.setEmail(rs.getString("email"));
            customer.setRemark(rs.getString("remark"));
            customerList.add(customer);
        }
        return customerList;
    } catch (SQLException e) {
        LOGGER.error("execute sql failure", e);
    }
}
```

```

        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    LOGGER.error("close connection failure", e);
                }
            }
        }
    }
}

```

运行一下 `getCustomerList` 方法的单元测试，如果不出意外的话，应该是“绿条”，表示可以测试通过，相反，如果出现“红条”就表示测试失败，我们可以查看控制台以了解具体的错误原因。

虽然以上代码可以运行，基本的功能算是可以实现了，但问题还是非常多，具体包括以下两个方面：

(1) 在 `CustomerService` 类中读取 `config.properties` 文件，这是不合理的，毕竟将来还有很多其他 `Service` 类需要做同样的事情，我们最好能将这些公共性的代码提取出来。

(2) 执行一条 `select` 语句需要编写一大堆代码，而且还必须使用 `try...catch...finally` 结构，开发效率明显不高。

用什么方法来解决以上这些问题呢？

我们先来解决第一个问题。

创建一个 `org.smart4j.chapter2.helper` 包，在该包中创建一个 `DatabaseHelper` 类，代码如下：

```

package org.smart4j.chapter2.helper;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.smart4j.chapter2.util.PropsUtil;

/**
 * 数据库操作助手类

```

```
*/
public final class DatabaseHelper {

    private static final Logger LOGGER=LoggerFactory.getLogger(Database-
    Helper.class);

    private static final String DRIVER;
    private static final String URL;
    private static final String USERNAME;
    private static final String PASSWORD;

    static {
        Properties conf=PropsUtil.loadProps("config.properties");
        DRIVER=conf.getProperty("jdbc.driver");
        URL=conf.getProperty("jdbc.url");
        USERNAME=conf.getProperty("jdbc.username");
        PASSWORD=conf.getProperty("jdbc.password");

        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException e) {
            LOGGER.error("can not load jdbc driver", e);
        }
    }

    /**
     * 获取数据库连接
     */
    public static Connection getConnection() {
        Connection conn=null;
        try {
            conn=DriverManager.getConnection(URL, USERNAME, PASSWORD);
        } catch (SQLException e) {
            LOGGER.error("get connection failure", e);
        }
        return conn;
    }
}
```

```

/**
 * 关闭数据库连接
 */
public static void closeConnection(Connection conn) {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            LOGGER.error("close connection failure", e);
        }
    }
}
}

```

可见，在 **DatabaseHelper** 中包含一些静态方法，用它们来封装数据库的相关操作，目前提供了“获取数据库连接”与“关闭数据库连接”两个助手方法。

现在 **CustomerService** 可以稍微简化一下了：

```

/**
 * 获取客户列表
 */
public List<Customer> getCustomerList() {
    Connection conn=null;
    try {
        List<Customer> customerList=new ArrayList<Customer>();
        String sql="SELECT * FROM customer";
        conn=DatabaseHelper.getConnection(); // <1>
        PreparedStatement stmt=conn.prepareStatement(sql);
        ResultSet rs=stmt.executeQuery();
        while (rs.next()) {
            Customer customer=new Customer();
            customer.setId(rs.getLong("id"));
            customer.setName(rs.getString("name"));
            customer.setContact(rs.getString("contact"));
            customer.setTelephone(rs.getString("telephone"));
            customer.setEmail(rs.getString("email"));
            customer.setRemark(rs.getString("remark"));
            customerList.add(customer);
        }
    }
}

```

```

        }
        return customerList;
    } catch (SQLException e) {
        LOGGER.error("execute sql failure", e);
    } finally {
        DatabaseHelper.closeConnection(conn); // <2>
    }
}

```

注意以上代码中<1>、<2>两处的修改。的确稍微简化了一些，至少每个 **Service** 类不会出现那些静态变量与代码块了，公共的代码放到了 **DatabaseHelper** 中，从而得到了重用。

我们再来解决第二个问题，实际上就是如何使代码变得更加精简。

著名的 **Apache Common** 项目中有一款名为 **DbUtils** 的类库，为我们提供了一个 **JDBC** 的封装，下面就用这款工具来解决该问题。

首先，需要在 **pom.xml** 中添加一个 **DbUtils** 的依赖：

```

<!-- Apache Commons DbUtils -->
<dependency>
    <groupId>commons-dbutils</groupId>
    <artifactId>commons-dbutils</artifactId>
    <version>1.6</version>
</dependency>

```

然后，在 **DatabaseHelper** 中添加如下代码：

```

import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanListHandler;

public final class DatabaseHelper {
    ...
    private static final QueryRunner QUERY_RUNNER=new QueryRunner();
    ...
    /**
     * 查询实体列表
     */
    public static <T> List<T> queryEntityList(Class<T> entityClass, String
sql, Object... params) {
        List<T> entityList;
        try {

```

```

        entityList=QUERY_RUNNER.query(conn, sql, new BeanListHandler<T>
(entityClass), params);
    } catch (SQLException e) {
        LOGGER.error("query entity list failure", e);
        throw new RuntimeException(e);
    } finally {
        closeConnection();
    }
    return entityList;
}

```

需要说明的是，使用 DbUtils 提供的 QueryRunner 对象可以面向实体（Entity）进行查询。实际上，DbUtils 首先执行 SQL 语句并返回一个 ResultSet，随后通过反射去创建并初始化实体对象。由于我们需要返回的是 List，因此可以使用 BeanListHandler。

最后，改写 CustomerService 的 getCustomerList 方法：

```

/**
 * 获取客户列表
 */
public List<Customer> getCustomerList() {
    Connection conn=DatabaseHelper.getConnection();
    try {
        String sql="SELECT * FROM customer";
        return DatabaseHelper.queryEntityList(Customer.class, conn, sql);
    } finally {
        DatabaseHelper.closeConnection(conn);
    }
}

```

现在的代码简单了很多，我们不再面对 PreparedStatement 与 ResultSet 了，只需使用 DatabaseHelper 就能执行数据库操作。我们每次都需要创建一个 Connection，然后进行数据库操作，最后关闭 Connection。

如何能让 Connection 对于开发人员完全透明呢？也就是说，如何隐藏掉创建与关闭 Connection 的代码呢？

为了确保一个线程中只有一个 Connection，我们可以使用 ThreadLocal 来存放本地线程变量。也就是说，将当前线程中的 Connection 放入 ThreadLocal 中存起来，这些 Connection 一定不会出现线程不安全问题，可以将 ThreadLocal 理解为一个隔离线程的容器。

提示：关于 ThreadLocal 核心原理与应用场景，请参考本书后续章节。

现在 DatabaseHelper 需要做一些调整了，具体代码如下：

```
...
public class DatabaseHelper {
    ...
    private static final ThreadLocal<Connection> CONNECTION_HOLDER=new
    ThreadLocal<Connection>();
    ...
    /**
     * 获取数据库连接
     */
    public static Connection getConnection() {
        Connection conn=CONNECTION_HOLDER.get(); // <1>
        if (conn == null) {
            try {
                conn=DriverManager.getConnection(URL, USERNAME, PASSWORD);
            } catch (SQLException e) {
                LOGGER.error("get connection failure", e);
                throw new RuntimeException(e);
            } finally {
                CONNECTION_HOLDER.set(conn); // <2>
            }
        }
        return conn;
    }

    /**
     * 关闭数据库连接
     */
    public static void closeConnection() {
        Connection conn=CONNECTION_HOLDER.get(); // <1>
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
```

```

        LOGGER.error("close connection failure", e);
        throw new RuntimeException(e);
    } finally {
        CONNECTION_HOLDER.remove(); // <3>
    }
}

/**
 * 查询实体列表
 */
public static <T> List<T> queryEntityList(Class<T> entityClass, String
sql, Object... params) {
    List<T> entityList;
    try {
        Connection conn=getConnection();
        entityList=QUERY_RUNNER.query(conn, sql, new BeanListHandler<T>
(entityClass), params);
    } catch (SQLException e) {
        LOGGER.error("query entity list failure", e);
        throw new RuntimeException(e);
    } finally {
        closeConnection();
    }
    return entityList;
}
}

```

当每次获取 `Connection` 时，首先在 `ThreadLocal` 中寻找（见<1>处），若不存在，则创建一个新的 `Connection`，并将其放入 `ThreadLocal` 中（见<2>处）。当 `Connection` 使用完毕后，需要移除 `ThreadLocal` 中持有的 `Connection`（见<3>处）。

现在 `CustomerService` 的 `getCustomerList` 方法看起来更加简单了：

```

/**
 * 获取客户列表
 */
public List<Customer> getCustomerList() {
    String sql="SELECT * FROM customer";

```

```
        return DatabaseHelper.queryEntityList(Customer.class, sql);
    }
```

使用同样的技巧，我们可以在 `DatabaseHelper` 中添加一个 `queryEntity` 方法，用户查询单个实体对象。

```
/**
 * 查询实体
 */
public static <T> T queryEntity(Class<T> entityClass, String sql, Object...
params) {
    T entity;
    try {
        Connection conn=getConnection();
        entity=QUERY_RUNNER.query(conn, sql, new BeanHandler<T>(entity-
Class), params);
    } catch (SQLException e) {
        LOGGER.error("query entity failure", e);
        throw new RuntimeException(e);
    } finally {
        closeConnection();
    }
    return entity;
}
```

需要注意的是，此时我们使用的是 `BeanHandler`，而不是 `BeanListHandler`。实际上，`DbUtils` 为我们提供了很多类似的 `Handler`，包括：

- `BeanHandler`——返回 `Bean` 对象；
- `BeanListHandler`——返回 `List` 对象；
- `BeanMapHandler`——返回 `Map` 对象；
- `ArrayHandler`——返回 `Object[]` 对象；
- `ArrayListHandler`——返回 `List` 对象；
- `MapHandler`——返回 `Map` 对象；
- `MapListHandler`——返回 `List` 对象；
- `ScalarHandler`——返回某列的值；
- `ColumnListHandler`——返回某列的值列表；

- **KeyedHandler**——返回 `Map` 对象，需要指定列名。

以上这些 `Handler` 都实现了 `ResultSetHandler`，层次结构如图 2-9 所示。

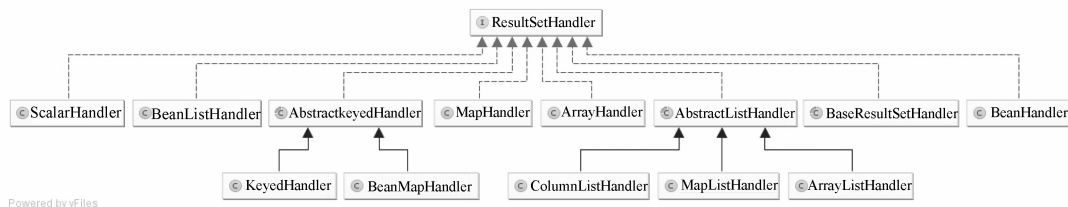


图 2-9 `ResultSetHandler` 层次结构

查询并不一定是基于单表的，将来有可能连接多表进行查询，因此我们有必要提供一个更为强大的查询方法，输入一个 SQL 与动态参数，输出一个 `List` 对象，其中的 `Map` 表示列名与列值的映射关系。

我们可以借助 `MapListHandler` 轻松实现：

```

/**
 * 执行查询语句
 */
public static List<Map<String, Object>> executeQuery(String sql, Object...
params) {
    List<Map<String, Object>> result;
    try {
        Connection conn=getConnection();
        result=QUERY_RUNNER.query(conn, sql, new MapListHandler(), params);
    } catch (Exception e) {
        LOGGER.error("execute query failure", e);
        throw new RuntimeException(e);
    }
    return result;
}

```

除了查询以外，还包括几种更新语句，例如 `update`、`insert`、`delete` 等，我们再提供一个通用的执行更新语句的方法：

```

/**
 * 执行更新语句（包括 update、insert、delete）
 */
public static int executeUpdate(String sql, Object... params) {

```

```
int rows=0;
try {
    Connection conn=getConnection();
    rows=QUERY_RUNNER.update(conn, sql, params);
} catch (SQLException e) {
    LOGGER.error("execute update failure", e);
    throw new RuntimeException(e);
} finally {
    closeConnection();
}
return rows;
}
```

该方法返回执行后受影响的行数，也就是说，更新了多少条记录。根据这个通用的方法，我们可以分别提供三种具体的数据库更新操作：

```
/**
 * 插入实体
 */
public static <T> boolean insertEntity(Class<T> entityClass, Map<String,
Object> fieldMap) {
    if (CollectionUtil.isEmpty(fieldMap)) {
        LOGGER.error("can not insert entity: fieldMap is empty");
        return false;
    }

    String sql="INSERT INTO " + getTableName(entityClass);
    StringBuilder columns=new StringBuilder("(");
    StringBuilder values=new StringBuilder("(");
    for (String fieldName : fieldMap.keySet()) {
        columns.append(fieldName).append(", ");
        values.append("?, ");
    }
    columns.replace(columns.lastIndexOf(", "), columns.length(), ")");
    values.replace(values.lastIndexOf(", "), values.length(), ")");
    sql += columns + " VALUES " + values;

    Object[] params=fieldMap.values().toArray();
```

```

        return executeUpdate(sql, params) == 1;
    }

    /**
     * 更新实体
     */
    public static <T> boolean updateEntity(Class<T> entityClass, long id,
        Map<String, Object> fieldMap) {
        if (CollectionUtil.isEmpty(fieldMap)) {
            LOGGER.error("can not update entity: fieldMap is empty");
            return false;
        }

        String sql="UPDATE " + getTableName(entityClass) + " SET ";
        StringBuilder columns=new StringBuilder();
        for (String fieldName : fieldMap.keySet()) {
            columns.append(fieldName).append("=?, ");
        }
        sql += columns.substring(0, columns.lastIndexOf(", ")) + " WHERE
        id=?";

        List<Object> paramList=new ArrayList<Object>();
        paramList.addAll(fieldMap.values());
        paramList.add(id);
        Object[] params=paramList.toArray();

        return executeUpdate(sql, params) == 1;
    }

    /**
     * 删除实体
     */
    public static <T> boolean deleteEntity(Class<T> entityClass, long id){
        String sql="DELETE FROM " + getTableName(entityClass) + " WHERE id=?";
        return executeUpdate(sql, id) == 1;
    }

    private static String getTableName(Class<?> entityClass) {

```

```
        return entityClass.getSimpleName();  
    }  
}
```

有了 `insertEntity`、`updateEntity`、`deleteEntity` 后，我们可以快速完成 `CustomerService` 中剩下的几个方法：

```
public class CustomerService {  
  
    /**  
     * 创建客户  
     */  
    public boolean createCustomer(Map<String, Object> fieldMap) {  
        return DatabaseHelper.insertEntity(Customer.class, fieldMap);  
    }  
  
    /**  
     * 更新客户  
     */  
    public boolean updateCustomer(long id, Map<String, Object> fieldMap) {  
        return DatabaseHelper.updateEntity(Customer.class, id, fieldMap);  
    }  
  
    /**  
     * 删除客户  
     */  
    public boolean deleteCustomer(long id) {  
        return DatabaseHelper.deleteEntity(Customer.class, id);  
    }  
}
```

现在 `CustomerService` 的实现相当精简，只需一两行代码即可完成单表的“增删改查”操作。

像上面这样，每次需要数据库连接时，就调用 `getConnection` 方法，在数据库操作完毕后，还需要调用 `closeConnection` 方法关闭数据库连接。虽然关闭这件事情已经被 `DatabaseHelper` 类给封装了，但考虑到如果频繁调用 `getConnection` 方式就会频繁创建数据库连接，这样做一定会造成大量的系统开销，毕竟数据库的连接数是有限的。因此，需要考虑一种解决方案，将这些数据库连接进行“池化”，也就是说，我们需要弄一个“数据库连接池”出来。Apache DBCP 是最好的数据库连接池之一，下面我们就使用这个数据库连接池。

首先，添加如下 Maven 依赖：

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.0.1</version>
</dependency>
```

然后，修改 DatabaseHelper 类的相关代码：

```
public final class DatabaseHelper {

    private static final ThreadLocal<Connection> CONNECTION_HOLDER;

    private static final QueryRunner QUERY_RUNNER;

    private static final BasicDataSource DATA_SOURCE;

    static {
        CONNECTION_HOLDER=new ThreadLocal<Connection>();

        QUERY_RUNNER=new QueryRunner();

        Properties conf=PropsUtil.loadProps("config.properties");
        String driver=conf.getProperty("jdbc.driver");
        String url=conf.getProperty("jdbc.url");
        String username=conf.getProperty("jdbc.username");
        String password=conf.getProperty("jdbc.password");

        DATA_SOURCE=new BasicDataSource();
        DATA_SOURCE.setDriverClassName(driver);
        DATA_SOURCE.setUrl(url);
        DATA_SOURCE.setUsername(username);
        DATA_SOURCE.setPassword(password);
    }

    /**
     * 获取数据库连接
     */
    public static Connection getConnection() {
        Connection conn=CONNECTION_HOLDER.get();
```

```
        if (conn == null) {
            try {
                conn=DATA_SOURCE.getConnection();
            } catch (SQLException e) {
                LOGGER.error("get connection failure", e);
                throw new RuntimeException(e);
            } finally {
                CONNECTION_HOLDER.set(conn);
            }
        }
        return conn;
    }
    ...
}
```

我们使用 Apache DBCP 的 `org.apache.commons.dbcp2.BasicDataSource` 来获取数据库连接，只需要保证该对象是静态的就行了。通过设置 `driver`、`url`、`username`、`password` 来初始化 `BasicDataSource`，并调用其 `getConnection` 方法即可获取数据库连接。

最后，删除以前的 `closeConnection` 方法，同时去掉所有 `closeConnection` 的 `finally` 代码块。

我们可以运行 `CustomerServiceTest` 单元测试类的相关测试方法来验证以上代码的正确性。现在的单元测试还存在一个非常严重的问题，比如执行完 `deleteCustomerTest` 方法，就不能继续执行 `getCustomerTest` 方法了，原因很简单，在测试之前数据库里的数据无法自动还原为初始状态。

JUnit 在调用每个 `@Test` 方法前，都会调用 `@Before` 方法，也就是我们在单元测试类里定义的 `init`，目前在该方法中预留了一个“TODO”，需要在这里初始化数据库，为我们准备一个便于测试的数据库环境。

此外，为了使测试数据库与开发数据库分离，也就是说，应该是两个数据库，只是表结构相同而已。我们需要为单元测试单独创建一个数据库，不妨命名为 `demo_test`，需要将 `demo` 数据库的 `customer` 表复制到 `demo_test` 数据库中。

在 Navicat 中进行如下具体操作：

(1) 进入 `demo` 数据库，选中 `customer` 表，使用 `Ctrl+C` 键复制表。

(2) 进入 `demo_test` 数据库，使用 `Ctrl+V` 键粘贴表，即可复制数据表。

(3) 在 `demo_test` 数据库中，选中 `customer` 表，单击右键，单击 `Truncate Table` 菜单项，即可清空表中所有的数据（`id` 将从 1 开始自增）。

随后，我们需要准备一个 `customer_init.sql` 文件，用于存放所有的 `insert` 语句，该文件位于 `test/resources/sql` 目录下，具体内容如下：

```
TRUNCATE customer;
INSERT INTO customer (name, contact, telephone, email, remark) VALUES
('customer1', 'Jack', '13512345678', 'jack@gmail.com', null);
INSERT INTO customer (name, contact, telephone, email, remark) VALUES
('customer2', 'Rose', '13623456789', 'rose@gmail.com', null);
```

我们需要先执行 `TRUNCATE` 语句，清空所有数据，然后再执行 `INSERT` 语句，插入相关数据。这里并不需要提供 `id` 列，因为该列是自增的。

技巧：默认情况下，IDEA 不会在 `test` 目录下自动创建 `resources` 目录，也就是说，我们需要手工创建，可使用 `Alt + Insert` 快捷键来完成。创建 `resources` 目录完毕后，需要右键单击该目录，单击 `Mark Directory As/Resources Root`，将该目录设置为 Maven 的测试资源目录。需要注意的是，`main/java`、`main/resources`、`test/java`、`test/resources` 这四个目录都是 `classpath` 的根目录，当运行单元测试时，遵循“就近原则”，即优先从 `test/java`、`test/resources` 加载类或读取文件。

目前 `test` 目录结构是这样的，如图 2-10 所示。

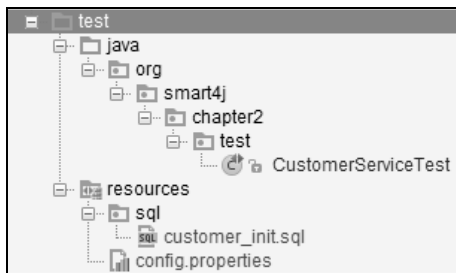


图 2-10 目录结构

我们既然已经为单元测试准备了独立的测试数据库，那么就有必要在 `test/resources` 目录下提供一个 `config.properties` 文件，用于提供测试数据库的相关配置：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/demo_test
jdbc.username=root
jdbc.password=root
```

注意这里使用的是 `demo_test`，而不是 `demo`。

下面我们利用 `DatabaseHelper` 的 `executeUpdate` 方法依次执行 `customer_init.sql` 中提供的 SQL 语句。

我们现在就来完成 `CustomerServiceTest` 的 `init` 方法：

```
@Before
public void init() throws Exception {
    String file="sql/customer_init.sql";
    InputStream is=Thread.currentThread().getContextClassLoader().getResourceAsStream(file);
    BufferedReader reader=new BufferedReader(new InputStreamReader(is));
    String sql;
    while ((sql=reader.readLine()) != null) {
        DatabaseHelper.executeUpdate(sql);
    }
}
```

从当前线程中获取线程上下文中的 `ClassLoader`，通过 `classpath` 下的 `sql/customer_init.sql` 获取一个 `InputStream` 对象，通过该输入流来创建 `BufferedReader` 对象，循环读取其中的每一行，并调用 `DatabaseHelper` 的 `executeUpdate` 方法来执行每条 SQL 语句。

没必要在所有单元测试类的 `init` 方法中都使用类似上面的代码，因此，有必要将这些代码做一个封装。不妨在 `DatabaseHelper` 类中提供一个 `executeSqlFile` 方法，只需提供 `filePath` 即可，就像下面这样：

```
/**
 * 执行 SQL 文件
 */
public static void executeSqlFile(String filePath) {
    InputStream is=Thread.currentThread().getContextClassLoader().getResourceAsStream(filePath);
    BufferedReader reader=new BufferedReader(new InputStreamReader(is));
    try {
        String sql;
        while ((sql=reader.readLine()) != null) {
            executeUpdate(sql);
        }
    } catch (Exception e) {
        LOGGER.error("execute sql file failure", e);
    }
}
```

```
        throw new RuntimeException(e);
    }
}
```

现在 `CustomerServiceTest` 的 `init` 方法更加简单了：

```
@Before
public void init() throws Exception {
    DatabaseHelper.executeSqlFile("sql/customer_init.sql");
}
```

运行所有测试方法，观察单元测试是否全部通过。

技巧：在 IDEA 中，将光标定位在测试方法外部，单击工具栏上的 `Run`（`Shift+F10`）或 `Debug`（`Shift+F9`）按钮，可执行所有的测试方法。如果只是将光标定位在某个测试方法内部，那么只能执行当前光标所在的方法。

至此，服务层开发完毕。这些服务会在控制器中得到应用，下一步我们就来完善控制器层。

2.3.2 完善控制器层

以 `CustomerServlet` 为例，目前的代码框架是这样的：

```
/**
 * 进入 客户列表 界面
 */
@WebServlet("/customer")
public class CustomerServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // TODO
    }
}
```

根据需求，当用户进入“客户列表”后，可看到所有的客户信息并以表格形式来展现。

我们要在 `CustomerServlet` 中创建一个 `CustomerService` 对象，并通过该对象的

getCustomerList 获取所有的客户列表。

很快我们就可以完成以下代码：

```
/**
 * 进入 客户列表 界面
 */
@WebServlet("/customer")
public class CustomerServlet extends HttpServlet {

    private CustomerService customerService;

    @Override
    public void init() throws ServletException {
        customerService=new CustomerService();
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        List<Customer> customerList=customerService.getCustomerList();
        req.setAttribute("customerList", customerList);
        req.getRequestDispatcher("/WEB-INF/view/customer.jsp").forward
            (req, resp);
    }
}
```

我们在 CustomerServlet 中定义一个 CustomerService 成员变量，并在 Servlet 的 init 方法中进行初始化。这样可以避免创建多个 CustomerService 实例，其实整个 Web 应用中只需要一个 CustomerService 实例（后续可使用“单例模式”进行优化）。

在 doGet 方法中，我们调用了 CustomerService 对象的 getCustomerList 方法来获取 List 对象，并将其放入请求属性中，最后通过请求对象重定向到 customer.jsp 视图。

2.3.3 完善视图层

打开/WEB-INF/view/customer.jsp 文件，完成如下代码：

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<c:set var="BASE" value="${pageContext.request.contextPath}"/>

<html>
<head>
    <title>客户管理</title>
</head>
<body>

<h1>客户列表</h1>

<table>
    <tr>
        <th>客户名称</th>
        <th>联系人</th>
        <th>电话号码</th>
        <th>邮箱地址</th>
        <th>操作</th>
    </tr>
    <c:forEach var="customer" items="${customerList}">
        <tr>
            <td>${customer.name}</td>
            <td>${customer.contact}</td>
            <td>${customer.telephone}</td>
            <td>${customer.email}</td>
            <td>
                <a href="${BASE}/customer_edit?id=${customer.id}">编辑</a>
                <a href="${BASE}/customer_delete?id=${customer.id}">删除</a>
            </td>
        </tr>
    </c:forEach>
</table>

</body>
</html>
```

技巧：在 IDEA 中查找文件有三种方法：使用 Ctrl+N 快捷键，根据 Java 类文件名进行查找；使用 Ctrl+Shift+N 快捷键，根据任意文件名进行查找；在 Project 目录树上，使用 Ctrl+Shift+F 快捷键在指定路径下查找文件。

部署应用到 Tomcat 中，在 IDEA 中启动 Tomcat，打开浏览器，输入地址：

http://localhost:8080/chapter2/customer，然后就会看到一个客户列表界面，如图 2-11 所示。

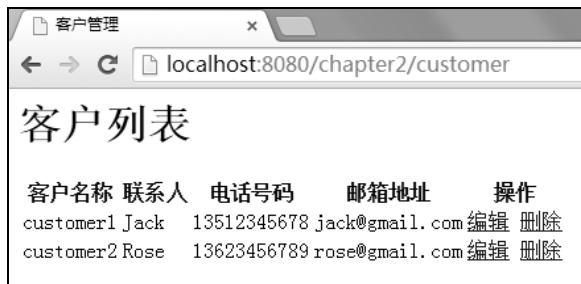


图 2-11 客户列表界面

现在 CustomerServlet 终于开发完毕了，但还有四个 Servlet 等待着我们去完成。

试想一下，随着业务需求的不断扩展，请求个数一定会不断增长，那么 Servlet 的数量一定会不断增多，势必会增加我们日后的维护工作量，得想个办法尽量减少 Servlet 的数量。

能否将同一个业务模块的 Servlet 合并到一个类中去呢？比如我们正在开发的这五个 Servlet: CustomerServlet、CustomerShowServlet、CustomerCreateServlet、CustomerEditServlet、CustomerDeleteServlet，合并到一个名为 CustomerController 类中。

也就是说，在 CustomerController 中包含了若干方法，每个方法都处理一种特定的请求，就像下面这样：

```
/**
 * 处理客户管理相关请求
 */
@Controller
public class CustomerController {

    @Inject
    private CustomerService customerService;

    /**
     * 进入 客户列表 界面
     */
    @Action("get:/customer")
    public View index(Param param) {
```

```
List<Customer> customerList=customerService.getCustomerList();
return new View("customer.jsp").addModel("customerList", customer-
List);
}

/**
 * 显示客户基本信息
 */
@RequestMapping("get:/customer_show")
public View show(Param param) {
    long id=param.getLong("id");
    Customer customer=customerService.getCustomer(id);
    return new View("customer_show.jsp").addModel("customer", cus-
tomer);
}

/**
 * 进入 创建客户 界面
 */
@RequestMapping("get:/customer_create")
public View create(Param param) {
    return new View("customer_create.jsp");
}

/**
 * 处理 创建客户 请求
 */
@RequestMapping("post:/customer_create")
public Data createSubmit(Param param) {
    Map<String, Object> fieldMap=param.getMap();
    boolean result=customerService.createCustomer(fieldMap);
    return new Data(result);
}

/**
 * 进入 编辑客户 界面
 */
@RequestMapping("get:/customer_edit")
```

```
public View edit(Param param) {
    long id=param.getLong("id");
    Customer customer=customerService.getCustomer(id);
    return new View("customer_edit.jsp").addModel("customer", customer);
}

/**
 * 处理 编辑客户 请求
 */
@RequestMapping("put:/customer_edit")
public Data editSubmit(Param param) {
    long id=param.getLong("id");
    Map<String, Object> fieldMap=param.getMap();
    boolean result=customerService.updateCustomer(id, fieldMap);
    return new Data(result);
}

/**
 * 处理 删除客户 请求
 */
@RequestMapping("delete:/customer_edit")
public Data delete(Param param) {
    long id=param.getLong("id");
    boolean result=customerService.deleteCustomer(id);
    return new Data(result);
}
}
```

(1) 在控制器类上，使用 **Controller** 注解，标明该类是一个控制器。

(2) 在成员变量上，使用 **Inject** 注解，自动创建该成员变量的实例，有一种流行的名称叫“注入（Inject）”。

(3) 在控制器类中包含若干方法（称为 **Action**），每个方法都会使用 **Get/Post/Put/Delete** 注解，用于指定“请求类型”与“请求路径”。

(4) 在 **Action** 的参数中，通过 **Param** 对象封装所有请求参数，可根据 **getLong/getFieldMap** 等方法获取具体类型或所有的请求参数。

(5) 在 **Action** 的返回值中，使用 **View** 封装一个 JSP 页面，使用 **Data** 封装一个 JSON 数据。

有了这个 `CustomerController` 以后, `Servlet` 数量一下子就降了下来, 而且还屏蔽了技术细节, 这种方式将 MVC 架构变得更加轻量级。

那么, 怎样开发这款轻量级 Web 框架呢? 在后面的章节里, 让我们一起探险吧。

2.4 总结

在本章中, 我们针对业务需求进行了简要的分析, 找出了需求中最核心的功能点, 并设计出了相关的用例, 根据业务实体确定了表结构, 通过界面原型设计了 URL 规范, 这些设计工作为后续的开发提供了指导。在开发阶段, 我们借助 MVC 架构, 将开发过程分解为八个步骤。我们对服务层进行了细化, 使用 `Apache Commons DbUtils` 类库开发了一个轻量级 JDBC 框架。此外, 还使用 `JUnit` 对每个服务方法进行了单元测试, 并提供了一种数据库初始化的解决方案, 让单元测试更加具备可测性。我们针对一个具体的业务需求, 从后端到前端进行了依次集成, 使数据在界面上得到了展现。

我们发现逐渐增长的 `Servlet` 会让我们的工作效率与维护成本大打折扣, 于是想到一个解决方案来减少 `Servlet` 的数量, 并简化 Web 应用的开发, 一款轻量级 Java Web 框架正在向我们挥手。



第 3 章

搭建轻量级 Java Web 框架

MVC (Model-View-Controller, 模型—视图—控制器) 是一种常用的设计模式, 可以使用这个模式将应用程序进行解耦。

在上一章中, 我们使用了一系列 Servlet 来充当 MVC 模式中的 Controller。这样做虽然可以实现基本的功能, 但 Servlet 的数量会随着业务功能的扩展而不断增加, 这不是我们所期望的。因此, 有必要减少 Servlet 的数量, 将某类业务交给 Controller 来处理, 它负责调用 Service 的相关方法, 并将返回值放入 Request 或 Response 中。此外, Service 不是通过 new 的方式来创建的, 而是通过一种名为“依赖注入”的方式, 让框架为我们来创建所需要的对象。

没错! 我们需要这样的框架, 它足够轻量级, 足够灵巧, 不妨给它取一个优雅的名字——Smart Framework, 本章我们就一起来实现这个框架。

读者将通过本章的学习, 您将掌握如下技能:

- 如何快速搭建开发框架;
- 如何加载并读取配置文件;
- 如何实现一个简单的 IOC 容器;
- 如何加载指定的类;
- 如何初始化框架。

我们首先需要确定一个目标, 然后搭建开发框架, 最后实现这个目标, 下面我们就一起开始吧!

3.1 确定目标

我们的目标是打造一个轻量级 MVC 框架，而 Controller 是 MVC 的核心。其实我们想要的是这样的 Controller 代码：

```
/**
 * 处理客户管理相关请求
 */
@Controller
public class CustomerController {

    @Inject
    private CustomerService customerService;

    /**
     * 进入 客户列表 界面
     */
    @Action("get:/customer")
    public View index(Param param) {
        List<Customer> customerList = customerService.getCustomerList();
        return new View("customer.jsp").addModel("customerList", customerList);
    }

    /**
     * 显示客户基本信息
     */
    @Action("get:/customer_show")
    public View show(Param param) {
        long id = param.getLong("id");
        Customer customer = customerService.getCustomer(id);
        return new View("customer_show.jsp").addModel("customer", customer);
    }

    /**
     * 进入 创建客户 界面
     */
}
```

```
@Action("get:/customer_create")
public View create(Param param) {
    return new View("customer_create.jsp");
}

/**
 * 处理 创建客户 请求
 */
@Action("post:/customer_create")
public Data createSubmit(Param param) {
    Map<String, Object> fieldMap = param.getMap();
    boolean result = customerService.createCustomer(fieldMap);
    return new Data(result);
}

/**
 * 进入 编辑客户 界面
 */
@Action("get:/customer_edit")
public View edit(Param param) {
    long id = param.getLong("id");
    Customer customer = customerService.getCustomer(id);
    return new View("customer_edit.jsp").addModel("customer", customer);
}

/**
 * 处理 编辑客户 请求
 */
@Action("put:/customer_edit")
public Data editSubmit(Param param) {
    long id = param.getLong("id");
    Map<String, Object> fieldMap = param.getMap();
    boolean result = customerService.updateCustomer(id, fieldMap);
    return new Data(result);
}

/**
 * 处理 删除客户 请求
```

```
*/  
@Action("delete:/customer_edit")  
public Data delete(Param param) {  
    long id = param.getLong("id");  
    boolean result = customerService.deleteCustomer(id);  
    return new Data(result);  
}  
}
```

通过 Controller 注解来定义 Controller 类，在该类中，可通过 Inject 注解定义一系列 Service 成员变量，这就是“依赖注入”。此外，有一系列被 Action 注解所定义的方法（简称 Action 方法），在这些 Action 方法中，调用了 Service 成员变量的方法来完成具体的业务逻辑。若返回 View 对象，则表示 JSP 页面；若返回 Data 对象，则表示一个 JSON 数据。

可见，Controller 代码非常清晰，一个 Controller 类包含了多个 Action 方法，可返回 View 或 Data 对象，分别对应 JSP 页面或 JSON 数据。

提示：在普通请求的情况下，可返回 JSP 页面；在 Ajax 请求的情况下，需要返回 JSON 数据。

现在 Controller 设计好了，这是我们想象中的样子，我们马上就要将想象变为现实。

下面我们就来搭建一个开发环境，实现这个 MVC 框架。

3.2 搭建开发环境

3.2.1 创建框架项目

首先，创建一个名为 smart-framework 的项目，它是一个普通的 Java 项目，在 pom.xml 中需要添加 Maven 三坐标：

```
<groupId>org.smart4j</groupId>  
<artifactId>smart-framework</artifactId>  
<version>1.0.0</version>
```

然后，需要为该项目添加相关的依赖。既然 Smart 是一款 Java Web 框架，那么一定会依赖 Servlet、JSP、JSTL 等。毫不犹豫，添加如下依赖：

```
<!-- Servlet -->
```

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>

<!-- JSP -->
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>

<!-- JSTL -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
  <scope>runtime</scope>
</dependency>
```

在框架中会大量使用日志输出，最流行的日志框架就是 **Log4J** 了，但它只是日志的一种具体实现，如果将来需要使用其他更好的日志框架，岂不是代码中很多日志输出的地方都需要修改？为了解决这个问题，我们使用一个名为 **SLF4J** 的日志框架，它实际上是日志框架的接口，而 **Log4J** 只是日志框架的一种实现而已。只需添加以下依赖，就能同时引入 **SLF4J** 与 **Log4J** 两个依赖。记住：**Maven** 依赖是有传递性的。

```
<!-- SLF4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.7</version>
</dependency>
```

由于我们使用了 **MySQL** 数据库，因此需要添加一个 **MySQL** 的 **Java** 驱动程序所对应的依赖：

```
<!-- MySQL -->
```

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.33</version>
  <scope>runtime</scope>
</dependency>
```

由于在 **Controller** 的 **Action** 方法返回值中是可以返回 JSON 数据的，因此需要选择一款 JSON 序列化工具，目前在功能、性能、稳定性各方面表现较好的 JSON 序列化工具就是 Jackson 了，我们就使用它。添加如下依赖：

```
<!-- Jackson -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.4.4</version>
</dependency>
```

还有一些常用的工具类，我们可以使用 Apache Commons 的以下两个依赖：

```
<!-- Apache Commons Lang -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.3.2</version>
</dependency>

<!-- Apache Commons Collections -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-collections4</artifactId>
  <version>4.0</version>
</dependency>
```

对于 JDBC 类库而言，毋庸置疑，我们选择了轻量级的 DbUtils，它也是 Apache Commons 的项目之一。添加如下依赖：

```
<!-- Apache Commons DbUtils -->
<dependency>
  <groupId>commons-dbutils</groupId>
```

```
<artifactId>commons-dbutils</artifactId>
<version>1.6</version>
</dependency>
```

最后，在框架中需要用到数据库连接池，我们选择了综合能力最强的连接池框架 DBCP，它同样是 Apache Commons 的项目之一。添加如下依赖：

```
<!-- Apache DBCP -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.0.1</version>
</dependency>
```

至此，Maven 依赖配置结束。

我们没有使用 Struts、Spring、Hibernate 这类框架，而是自行开发一个轻量级 MVC 框架，期望它能满足我们日常的工作需求，将来会不断扩展这个框架，让它变得更加强大。我们只使用了以上这些第三方开源项目，就是为了让框架的依赖尽可能少。

3.2.2 创建示例项目

除了 smart-framework 这个项目以外，我们有必要再创建一个使用该框架的项目，不妨命名为 chapter3 吧，它的代码大多数来自于 chapter2。

chapter3 项目是一个 Java Web 项目，它只需要依赖于 smart-framework 即可，详细的 pom.xml 代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.smart4j</groupId>
  <artifactId>chapter3</artifactId>
  <version>1.0.0</version>
```

```
<packaging>war</packaging>

<dependencies>
  <!-- Smart Framework -->
  <dependency>
    <groupId>org.smart4j</groupId>
    <artifactId>smart-framework</artifactId>
    <version>1.0.0</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <!-- Tomcat -->
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <path>/${project.artifactId}</path>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

开发环境准备完毕后，我们就可以实现具体的细节了。既然是一个框架，那么首先要考虑的问题就是配置，需要让配置尽可能地少，这样开发者的学习成本才会更低。

3.3 定义框架配置项

在 chapter3 项目的 src/main/resources 目录下，创建一个名为 smart.properties 的文件，内容如下：

```
smart.framework.jdbc.driver=com.mysql.jdbc.Driver
smart.framework.jdbc.url=jdbc:mysql://localhost:3306/demo
smart.framework.jdbc.username=root
```

```
smart.framework.jdbc.password=root

smart.framework.app.base_package=org.smart4j.chapter3
smart.framework.app.jsp_path=/WEB-INF/view/
smart.framework.app.asset_path=/asset/
```

前面几个配置项比较好理解，无非就是将数据库连接配置放到配置文件中，最后几个配置项有必要稍作解释：

- (1) `smart.framework.app.base_package` 表示 `chapter3` 项目的基础包名。
- (2) `smart.framework.app.jsp_path` 表示 JSP 的基础路径。
- (3) `smart.framework.app.asset_path` 表示静态资源文件的基础路径，比如 JS、CSS、图片等。

3.4 加载配置项

既然配置文件已经有了，那么如何根据配置项的名称来获取配置项的取值呢？这是框架需要做的事情。因此，我们在 `smart-framework` 项目中创建一个名为 `ConfigHelper` 的助手类，让它来读取 `smart.properties` 配置文件。

首先，我们需要创建一个名为 `ConfigConstant` 的常量类，让它来维护配置文件中相关的配置项名称，代码如下：

```
package org.smart4j.framework;

/**
 * 提供相关配置项常量
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface ConfigConstant {

    String CONFIG_FILE = "smart.properties";

    String JDBC_DRIVER = "smart.framework.jdbc.driver";
    String JDBC_URL = "smart.framework.jdbc.url";
    String JDBC_USERNAME = "smart.framework.jdbc.username";
```

```
String JDBC_PASSWORD = "smart.framework.jdbc.password";

String APP_BASE_PACKAGE = "smart.framework.app.base_package";
String APP_JSP_PATH = "smart.framework.app.jsp_path";
String APP_ASSET_PATH = "smart.framework.app.asset_path";
}
```

然后，我们只需要借助 **PropsUtil** 工具类就能轻松地实现 **ConfigHelper**，无非就是定义一些静态方法，让它们分别获取 **smart.properties** 配置文件中的配置项，代码如下：

```
package org.smart4j.framework.helper;

import java.util.Properties;
import org.smart4j.framework.ConfigConstant;
import org.smart4j.framework.util.PropsUtil;

/**
 * 属性文件助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class ConfigHelper {

    private static final Properties CONFIG_PROPS = PropsUtil.loadProps(
        ConfigConstant.CONFIG_FILE);

    /**
     * 获取 JDBC 驱动
     */
    public static String getJdbcDriver() {
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.JDBC_
            DRIVER);
    }

    /**
     * 获取 JDBC URL
     */
    public static String getJdbcUrl() {
```

```
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.JDBC_URL);
    }

    /**
     * 获取 JDBC 用户名
     */
    public static String getJdbcUsername() {
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.JDBC_
            USERNAME);
    }

    /**
     * 获取 JDBC 密码
     */
    public static String getJdbcPassword() {
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.JDBC_
            PASSWORD);
    }

    /**
     * 获取应用基础包名
     */
    public static String getAppBasePackage() {
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.APP_BASE_
            PACKAGE);
    }

    /**
     * 获取应用 JSP 路径
     */
    public static String getAppJspPath() {
        return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.APP_JSP_
            PATH, "/WEB-INF/view/");
    }

    /**
     * 获取应用静态资源路径
     */
```

```
public static String getAppAssetPath() {  
    return PropsUtil.getString(CONFIG_PROPS, ConfigConstant.APP_ASSET_  
        PATH, "/asset/");  
}  
}
```

可见，在 `ConfigHelper` 类中，为 `smart.framework.app.jsp_path` 与 `smart.framework.app.asset_path` 配置项提供了默认值。也就是说，在 `smart.properties` 配置文件中这两个配置项是可选的，如果不是特殊要求，可以修改这两个配置。换句话说，这两个配置项会以 `smart.properties` 配置文件中所配置的值为优先值。

3.5 开发一个类加载器

我们需要开发一个“类加载器”来加载该基础包下的所有类，比如使用了某注解的类，或者实现了某接口的类，又或者继承了某父类的所有子类等。

有必要写一个 `ClassUtil` 工具类，提供与类操作相关的方法，比如获取类加载器、加载类、获取指定包下的所有类等。代码如下：

```
package org.smart4j.framework.util;  
  
import java.io.File;  
import java.io.FileFilter;  
import java.net.JarURLConnection;  
import java.net.URL;  
import java.util.Enumeration;  
import java.util.HashSet;  
import java.util.Set;  
import java.util.jar.JarEntry;  
import java.util.jar.JarFile;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
/**  
 * 类操作工具类  
 *  
 * @author huangyong  
 * @since 1.0.0
```

```

    */
    public final class ClassUtil {

        private static final Logger LOGGER = LoggerFactory.getLogger(ClassUtil.class);

        /**
         * 获取类加载器
         */
        public static ClassLoader getClassLoader() {
            ...
        }

        /**
         * 加载类
         */
        public static Class<?> loadClass(String className, boolean isInitialized) {
            ...
        }

        /**
         * 获取指定包名下的所有类
         */
        public static Set<Class<?>> getClassSet(String packageName) {
            ...
        }
    }
}

```

获取类加载器实现起来最为简单，只需获取当前线程中的 `ClassLoader` 即可，该方法具体实现如下：

```

/**
 * 获取类加载器
 */
public static ClassLoader getClassLoader() {
    return Thread.currentThread().getContextClassLoader();
}

```

加载类需要提供类名与是否初始化的标志，这里提到的初始化指是否执行类的静态代码块，

该方法具体实现如下：

```
/**
 * 加载类
 */
public static Class<?> loadClass(String className, boolean isInitialized) {
    Class<?> cls;
    try {
        cls = Class.forName(className, isInitialized, getClassLoader());
    } catch (ClassNotFoundException e) {
        LOGGER.error("load class failure", e);
        throw new RuntimeException(e);
    }
    return cls;
}
```

为了提高加载类的性能，可将 `loadClass` 方法的 `isInitialized` 参数设置为 `false`。

最为复杂的是获取指定包名下的所有类，我们需要根据包名并将其转换为文件路径，读取 `class` 文件或 `jar` 包，获取指定的类名去加载类，该方法具体实现如下：

```
/**
 * 获取指定包名下的所有类
 */
public static Set<Class<?>> getClassSet(String packageName) {
    Set<Class<?>> classSet = new HashSet<Class<?>>();
    try {
        Enumeration<URL> urls = getClassLoader().getResources(packageName.
            replace(".", "/"));
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            if (url != null) {
                String protocol = url.getProtocol();
                if (protocol.equals("file")) {
                    String packagePath = url.getPath().replaceAll("%20", " ");
                    addClass(classSet, packagePath, packageName);
                } else if (protocol.equals("jar")) {
                    JarURLConnection jarURLConnection = (JarURLConnection)
                        url.openConnection();
                    if (jarURLConnection != null) {
```

```

        JarFile jarFile = jarURLConnection.getJarFile();
        if (jarFile != null) {
            Enumeration<JarEntry> jarEntries = jarFile.
                entries();
            while (jarEntries.hasMoreElements()) {
                JarEntry jarEntry = jarEntries.nextElement();
                String jarEntryName = jarEntry.getName();
                if (jarEntryName.endsWith(".class")) {
                    String className = jarEntryName.substring(0,
                        jarEntryName.lastIndexOf(".")).replaceAll
                            ("/", ".");
                    doAddClass(classSet, className);
                }
            }
        }
    }
}

} catch (Exception e) {
    LOGGER.error("get class set failure", e);
    throw new RuntimeException(e);
}

return classSet;
}

private static void addClass(Set<Class<?>> classSet, String packagePath,
    String packageName) {
    File[] files = new File(packagePath).listFiles(new FileFilter() {
        @Override
        public boolean accept(File file) {
            return (file.isFile() && file.getName().endsWith(".class")) ||
                file.isDirectory();
        }
    });
    for (File file : files) {
        String fileName = file.getName();
        if (file.isFile()) {

```

```

        String className = fileName.substring(0, fileName.
        lastIndexOf("."));
        if (StringUtil.isEmpty(packageName)) {
            className = packageName + "." + className;
        }
        doAddClass(classSet, className);
    } else {
        String subPackagePath = fileName;
        if (StringUtil.isEmpty(packagePath)) {
            subPackagePath = packagePath + "/" + subPackagePath;
        }
        String subPackageName = fileName;
        if (StringUtil.isEmpty(packageName)) {
            subPackageName = packageName + "." + subPackageName;
        }
        addClass(classSet, subPackagePath, subPackageName);
    }
}

private static void doAddClass(Set<Class<?>> classSet, String className) {
    Class<?> cls = loadClass(className, false);
    classSet.add(cls);
}

```

我们的目标是在控制器类上使用 **Controller** 注解，在控制器类的方法上使用 **Action** 注解，在服务类上使用 **Service** 注解，在控制器类中可使用 **Inject** 注解将服务类依赖注入进来。因此，我们需要自定义这 4 个注解类。

控制器注解代码如下：

```

package org.smart4j.framework.annotation;

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

/**

```

```

    * 控制器注解
    *
    * @author huangyong
    * @since 1.0.0
    */
    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Controller {
    }

```

Action 方法注解代码如下:

```

package org.smart4j.framework.annotation;

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

/**
 * Action 方法注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Action {

    /**
     * 请求类型与路径
     */
    String value();
}

```

服务类注解代码如下:

```

package org.smart4j.framework.annotation;

import java.lang.annotation.ElementType;

```

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

/**
 * 服务类注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Service {
}
```

依赖注入注解代码如下：

```
package org.smart4j.framework.annotation;

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

/**
 * 依赖注入注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Inject {
}
```

由于我们在 `smart.properties` 配置文件中指定了 `smart.framework.app.base_package`，它是整个应用的基础包名，通过 `ClassUtil` 加载的类都需要基于该基础包名。所以有必要提供一个 `ClassHelper` 助手类，让它分别获取应用包名下的所有类、应用包名下所有 `Service` 类、应用包名下所有 `Controller` 类。此外，我们可以将带有 `Controller` 注解与 `Service` 注解的类所产生的对

象, 理解为由 Smart 框架所管理的 Bean, 所以有必要在 ClassHelper 类中增加一个获取应用包名下所有 Bean 类的方法。ClassHelper 细节的代码如下:

```
package org.smart4j.framework.helper;

import java.util.HashSet;
import java.util.Set;
import org.smart4j.framework.annotation.Controller;
import org.smart4j.framework.annotation.Service;
import org.smart4j.framework.util.ClassUtil;

/**
 * 类操作助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class ClassHelper {

    /**
     * 定义类集合 (用于存放所加载的类)
     */
    private static final Set<Class<?>> CLASS_SET;

    static {
        String basePackage = ConfigHelper.getAppBasePackage();
        CLASS_SET = ClassUtil.getClassSet(basePackage);
    }

    /**
     * 获取应用包名下的所有类
     */
    public static Set<Class<?>> getClassSet() {
        return CLASS_SET;
    }

    /**
     * 获取应用包名下所有 Service 类
     */
}
```

```
    */
    public static Set<Class<?>> getServiceClassSet() {
        Set<Class<?>> classSet = new HashSet<Class<?>>();
        for (Class<?> cls : CLASS_SET) {
            if (cls.isAnnotationPresent(Service.class)) {
                classSet.add(cls);
            }
        }
        return classSet;
    }

    /**
     * 获取应用包名下所有 Controller 类
     */
    public static Set<Class<?>> getControllerClassSet() {
        Set<Class<?>> classSet = new HashSet<Class<?>>();
        for (Class<?> cls : CLASS_SET) {
            if (cls.isAnnotationPresent(Controller.class)) {
                classSet.add(cls);
            }
        }
        return classSet;
    }

    /**
     * 获取应用包名下所有 Bean 类（包括：Service、Controller 等）
     */
    public static Set<Class<?>> getBeanClassSet() {
        Set<Class<?>> beanClassSet = new HashSet<Class<?>>();
        beanClassSet.addAll(getServiceClassSet());
        beanClassSet.addAll(getControllerClassSet());
        return beanClassSet;
    }
}
```

就像上面这样，我们使用 `ClassHelper` 封装了 `ClassUtil`，并提供了一系列的助手方法，通过这些方法可以直接获取我们想要的类集合。在后面的开发中，我们会经常使用到 `ClassHelper`。

3.6 实现 Bean 容器

使用 `ClassHelper` 类可以获取所加载的类，但无法通过类来实例化对象。因此，需要提供一个反射工具类，让它封装 Java 反射相关的 API，对外提供更好用的工具方法。不妨将该类命名为 `ReflectionUtil`，代码如下：

```
package org.smart4j.framework.util;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 反射工具类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class ReflectionUtil {

    private static final Logger LOGGER = LoggerFactory.getLogger(
        (ReflectionUtil.class));

    /**
     * 创建实例
     */
    public static Object newInstance(Class<?> cls) {
        Object instance;
        try {
            instance = cls.newInstance();
        } catch (Exception e) {
            LOGGER.error("new instance failure", e);
            throw new RuntimeException(e);
        }
        return instance;
    }
}
```

```
/**
 * 调用方法
 */
public static Object invokeMethod(Object obj, Method method, Object...
args) {
    Object result;
    try {
        method.setAccessible(true);
        result = method.invoke(obj, args);
    } catch (Exception e) {
        LOGGER.error("invoke method failure", e);
        throw new RuntimeException(e);
    }
    return result;
}

/**
 * 设置成员变量的值
 */
public static void setField(Object obj, Field field, Object value) {
    try {
        field.setAccessible(true);
        field.set(obj, value);
    } catch (Exception e) {
        LOGGER.error("set field failure", e);
        throw new RuntimeException(e);
    }
}
}
```

我们需要获取所有被 Smart 框架管理的 Bean 类，此时需要调用 ClassHelper 类的 `getBeanClassSet` 方法，随后需要循环调用 ReflectionUtil 类的 `newInstance` 方法，根据类来实例化对象，最后将每次创建的对象存放在一个静态的 `Map<Class<?>, Object>` 中。我们需要随时获取该 Map，还需要通过该 Map 的 key（类名）去获取所对应的 value（Bean 对象）。BeanHelper 类代码如下：

```
package org.smart4j.framework.helper;

import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Set;
import org.smart4j.framework.util.ReflectionUtil;

/**
 * Bean 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class BeanHelper {

    /**
     * 定义 Bean 映射（用于存放 Bean 类与 Bean 实例的映射关系）
     */
    private static final Map<Class<?>, Object> BEAN_MAP = new HashMap<Class<?>, Object>();

    static {
        Set<Class<?>> beanClassSet = ClassHelper.getBeanClassSet();
        for (Class<?> beanClass : beanClassSet) {
            Object obj = ReflectionUtil.newInstance(beanClass);
            BEAN_MAP.put(beanClass, obj);
        }
    }

    /**
     * 获取 Bean 映射
     */
    public static Map<Class<?>, Object> getBeanMap() {
        return BEAN_MAP;
    }

    /**
     * 获取 Bean 实例
     */
    @SuppressWarnings("unchecked")
    public static <T> T getBean(Class<T> cls) {
```

```

        if (!BEAN_MAP.containsKey(cls)) {
            throw new RuntimeException("can not get bean by class: " + cls);
        }
        return (T) BEAN_MAP.get(cls);
    }
}

```

现在，BeanHelper 就相当于一个“Bean 容器”了，因为在 Bean Map 中存放了 Bean 类与 Bean 实例的映射关系，我们只需通过调用 `getBean` 方法，传入一个 Bean 类，就能获取 Bean 实例。

3.7 实现依赖注入功能

我们在 Controller 中定义 Service 成员变量，然后在 Controller 的 Action 方法中调用 Service 成员变量的方法。那么，如何实例化 Service 成员变量呢？

还记得之前定义的 Inject 注解吗？我们就用它来实现 Service 实例化。那么，谁来实例化呢？

不是开发者自己通过 `new` 的方式来实例化，而是通过框架自身来实例化，像这类实例化过程，称为 IoC（Inversion of Control，控制反转）。控制不是由开发者来决定的，而是反转给框架了。一般地，我们也将控制反转称为 DI（Dependency Injection，依赖注入），可以理解为将某个类需要依赖的成员注入到这个类中。那么，如何来实现依赖注入呢？

最简单的方式是，先通过 BeanHelper 获取所有 Bean Map（是一个 `Map<Class<?>, Object>` 结构，记录了类与对象的映射关系）。然后遍历这个映射关系，分别取出 Bean 类与 Bean 实例，进而通过反射获取类中所有的成员变量。继续遍历这些成员变量，在循环中判断当前成员变量是否带有 Inject 注解，若带有该注解，则从 Bean Map 中根据 Bean 类取出 Bean 实例。最后通过 `ReflectionUtil#setField` 方法来修改当前成员变量的值。

我们把以上这段逻辑写成一个名为 `IocHelper` 的类，让它来完成这件事情：

```

package org.smart4j.framework.helper;

import java.lang.reflect.Field;
import java.util.Map;
import org.smart4j.framework.annotation.Inject;
import org.smart4j.framework.util.ArrayUtil;
import org.smart4j.framework.util.CollectionUtil;
import org.smart4j.framework.util.ReflectionUtil;

```

```

/**
 * 依赖注入助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class IoHelper {

    static {
        // 获取所有的 Bean 类与 Bean 实例之间的映射关系（简称 Bean Map）
        Map<Class<?>, Object> beanMap = BeanHelper.getBeanMap();
        if (CollectionUtil.isEmpty(beanMap)) {
            // 遍历 Bean Map
            for (Map.Entry<Class<?>, Object> beanEntry : beanMap.entrySet()) {
                // 从 BeanMap 中获取 Bean 类与 Bean 实例
                Class<?> beanClass = beanEntry.getKey();
                Object beanInstance = beanEntry.getValue();
                // 获取 Bean 类定义的所有成员变量（简称 Bean Field）
                Field[] beanFields = beanClass.getDeclaredFields();
                if (ArrayUtil.isEmpty(beanFields)) {
                    // 遍历 Bean Field
                    for (Field beanField : beanFields) {
                        // 判断当前 Bean Field 是否带有 Inject 注解
                        if (beanField.isAnnotationPresent(Inject.class)) {
                            // 在 Bean Map 中获取 Bean Field 对应的实例
                            Class<?> beanFieldClass = beanField.getType();
                            Object beanFieldInstance = beanMap.get(beanFieldClass);
                            if (beanFieldInstance != null) {
                                // 通过反射初始化 BeanField 的值
                                ReflectionUtil.setField(beanInstance, beanField, beanFieldInstance);
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```
    }  
}
```

只需要在 `IocHelper` 的静态块中实现相关逻辑，就能完成 IoC 容器的初始化工作。那么，这个静态块在什么时候加载呢？

其实，当 `IocHelper` 这个类被加载的时候，就会加载它的静态块。后面我们需要找一个统一的地方来加载这个 `IocHelper`。

其中涉及了 `ArrayUtil` 类，它实际上是对数组的一些常用方法的封装，代码如下：

```
package org.smart4j.framework.util;  
  
import org.apache.commons.lang3.ArrayUtils;  
  
/**  
 * 数组工具类  
 *  
 * @author huangyong  
 * @since 1.0.0  
 */  
public final class ArrayUtil {  
  
    /**  
     * 判断数组是否非空  
     */  
    public static boolean isEmpty(Object[] array) {  
        return !ArrayUtils.isEmpty(array);  
    }  
  
    /**  
     * 判断数组是否为空  
     */  
    public static boolean isNotEmpty(Object[] array) {  
        return ArrayUtils.isEmpty(array);  
    }  
}
```

可见，一个简单的 IoC 框架只需十几行代码就能搞定，似乎比我们想象中的要简单许多。需要注意的是，此时在 IoC 框架中所管理的对象都是单例的，由于 IoC 框架底层还是从 `BeanHelper` 中获取 `Bean Map` 的，而 `Bean Map` 中的对象都是事先创建好并放入这个 `Bean` 容器

的，所有的对象都是单例的。

3.8 加载 Controller

我们需要创建一个 `ControllerHelper` 类，让它来处理如下逻辑：

通过 `ClassHelper`，我们可以获取所有定义了 `Controller` 注解的类，可以通过反射获取该类中所有带有 `Action` 注解的方法（简称“`Action` 方法”），获取 `Action` 注解中的请求表达式，进而获取请求方法与请求路径，封装一个请求对象（`Request`）与处理对象（`Handler`），最后将 `Request` 与 `Handler` 建立一个映射关系，放入一个 `Action Map` 中，并提供一个可根据请求方法与请求路径获取处理对象的方法。

首先，我们定义一个名为 `Request` 的类，代码如下：

```
package org.smart4j.framework.bean;

import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;

/**
 * 封装请求信息
 *
 * @author huangyong
 * @since 1.0.0
 */
public class Request {

    /**
     * 请求方法
     */
    private String requestMethod;

    /**
     * 请求路径
     */
    private String requestPath;

    public Request(String requestMethod, String requestPath) {
        this.requestMethod = requestMethod;
    }
}
```

```
        this.requestPath = requestPath;
    }

    public String getRequestMethod() {
        return requestMethod;
    }

    public String getRequestPath() {
        return requestPath;
    }

    @Override
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }

    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }
}
```

然后，编写一个名为 **Handler** 的类，代码如下：

```
package org.smart4j.framework.bean;

import java.lang.reflect.Method;

/**
 * 封装 Action 信息
 *
 * @author huangyong
 * @since 1.0.0
 */
public class Handler {

    /**
     * Controller 类
     */
}
```

```
private Class<?> controllerClass;

/**
 * Action 方法
 */
private Method actionMethod;

public Handler(Class<?> controllerClass, Method actionMethod) {
    this.controllerClass = controllerClass;
    this.actionMethod = actionMethod;
}

public Class<?> getControllerClass() {
    return controllerClass;
}

public Method getActionMethod() {
    return actionMethod;
}
}
```

最后，这是我们的 **ControllerHelper**，代码如下：

```
package org.smart4j.framework.helper;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.smart4j.framework.annotation.Action;
import org.smart4j.framework.bean.Handler;
import org.smart4j.framework.bean.Request;
import org.smart4j.framework.util.ArrayUtil;
import org.smart4j.framework.util.CollectionUtil;

/**
 * 控制器助手类
 *
 * @author huangyong
 */
```

```
* @since 1.0.0
*/
public final class ControllerHelper {

    /**
     * 用于存放请求与处理器的映射关系（简称 Action Map）
     */
    private static final Map<Request, Handler> ACTION_MAP = new HashMap
    <Request, Handler>();

    static {
        // 获取所有的 Controller 类
        Set<Class<?>> controllerClassSet = ClassHelper.getController-
        ClassSet();
        if (CollectionUtil.isEmpty(controllerClassSet)) {
            // 遍历这些 Controller 类
            for (Class<?> controllerClass : controllerClassSet) {
                // 获取 Controller 类中定义的方法
                Method[] methods = controllerClass.getDeclaredMethods();
                if (ArrayUtil.isEmpty(methods)) {
                    // 遍历这些 Controller 类中的方法
                    for (Method method : methods) {
                        // 判断当前方法是否带有 Action 注解
                        if (method.isAnnotationPresent(Action.class)) {
                            // 从 Action 注解中获取 URL 映射规则
                            Action action = method.getAnnotation(Action.
                            class);
                            String mapping = action.value();
                            // 验证 URL 映射规则
                            if (mapping.matches("\\w+://\\w*")) {
                                String[] array = mapping.split(":");
                                if (ArrayUtil.isEmpty(array) && array.length
                                == 2) {
                                    // 获取请求方法与请求路径
                                    String requestMethod = array[0];
                                    String requestPath = array[1];
                                    Request request = new Request(requestMethod,
                                    requestPath);
```

```

        Handler handler = new Handler(controller-
        Class, method);
        // 初始化 Action Map
        ACTION_MAP.put(request, handler);
    }
}
}
}
}
}
}

/**
 * 获取 Handler
 */
public static Handler getHandler(String requestMethod, String requestPath) {
    Request request = new Request(requestMethod, requestPath);
    return ACTION_MAP.get(request);
}
}
}

```

可见，我们在 `ControllerHelper` 中封装了一个 `Action Map`，通过它来存放 `Request` 与 `Handler` 之间的映射关系，然后通过 `ClassHelper` 来获取所有带有 `Controller` 注解的类，接着遍历这些 `Controller` 类，从 `Action` 注解中提取 `URL`，最后初始化 `Request` 与 `Handler` 之间的映射关系。

3.9 初始化框架

通过上面的过程，我们创建了 `ClassHelper`、`BeanHelper`、`IocHelper`、`ControllerHelper`，这四个 `Helper` 类需要通过一个入口程序来加载它们，实际上是加载它们的静态块。

这个加载程序就是 `HelperLoader`，代码如下：

```

package org.smart4j.framework;

import org.smart4j.framework.helper.BeanHelper;
import org.smart4j.framework.helper.ClassHelper;
import org.smart4j.framework.helper.ControllerHelper;
import org.smart4j.framework.helper.IocHelper;

```

```
import org.smart4j.framework.util.ClassUtil;

/**
 * 加载相应的 Helper 类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class HelperLoader {

    public static void init() {
        Class<?>[] classList = {
            ClassHelper.class,
            BeanHelper.class,
            IocHelper.class,
            ControllerHelper.class
        };
        for (Class<?> cls : classList) {
            ClassUtil.loadClass(cls.getName());
        }
    }
}
```

现在我们可以直接调用 `HelperLoader` 的 `init` 方法来加载这些 `Helper` 类了。实际上，当我们在第一次访问类时，就会加载其 `static` 块，这里只是为了让加载更加集中，所以才写了一个 `HelperLoader` 类。

3.10 请求转发器

以上过程都是在为这一步做准备，我们现在需要编写一个 `Servlet`，让它来处理所有的请求。从 `HttpServletRequest` 对象中获取请求方法与请求路径，通过 `ControllerHelper#getHandler` 方法来获取 `Handler` 对象。

当拿到 `Handler` 对象后，我们可以方便地获取 `Controller` 的类，进而通过 `BeanHelper.getBean` 方法获取 `Controller` 的实例对象。

随后可以从 `HttpServletRequest` 对象中获取所有请求参数，并将其初始化到一个名为 `Param` 的对象中，`Param` 类代码如下：

```
package org.smart4j.framework.bean;

import java.util.Map;
import org.smart4j.framework.util.CastUtil;

/**
 * 请求参数对象
 *
 * @author huangyong
 * @since 1.0.0
 */
public class Param {

    private Map<String, Object> paramMap;

    public Param(Map<String, Object> paramMap) {
        this.paramMap = paramMap;
    }

    /**
     * 根据参数名获取 long 型参数值
     */
    public long getLong(String name) {
        return CastUtil.castLong(paramMap.get(name));
    }

    /**
     * 获取所有字段信息
     */
    public Map<String, Object> getMap() {
        return paramMap;
    }
}
```

在 `Param` 类中，会有一系列的 `get` 方法，可通过参数名获取指定类型的参数值，也可以获取所有参数的 `Map` 结构。

还可从 `Handler` 对象中获取 `Action` 的方法返回值，该返回值可能有两种情况：

(1) 若返回值是 `View` 类型的视图对象，则返回一个 JSP 页面。

(2) 若返回值是 **Data** 类型的数据对象，则返回一个 JSON 数据。

我们需要根据以上两种情况来判断 **Action** 的返回值，并做不同的处理。

首先，看看 **View** 类，代码如下：

```
package org.smart4j.framework.bean;

import java.util.HashMap;
import java.util.Map;

/**
 * 返回视图对象
 *
 * @author huangyong
 * @since 1.0.0
 */
public class View {

    /**
     * 视图路径
     */
    private String path;

    /**
     * 模型数据
     */
    private Map<String, Object> model;

    public View(String path) {
        this.path = path;
        model = new HashMap<String, Object>();
    }

    public View addModel(String key, Object value) {
        model.put(key, value);
        return this;
    }

    public String getPath() {
```

```
        return path;
    }

    public Map<String, Object> getModel() {
        return model;
    }
}
```

由于视图中是可以包含模型数据的,因此在 View 中包括了视图路径和该视图所需的模型数据,该模型数据是一个 Map 类型的“键值对”,可在视图中根据模型的键名获取键值。

然后,看看 Data 类,代码如下:

```
package org.smart4j.framework.bean;

/**
 * 返回数据对象
 *
 * @author huangyong
 * @since 1.0.0
 */
public class Data {

    /**
     * 模型数据
     */
    private Object model;

    public Data(Object model) {
        this.model = model;
    }

    public Object getModel() {
        return model;
    }
}
```

返回的 Data 类型的数据封装了一个 Object 类型的模型数据,框架会将该对象写入 HttpServletResponse 对象中,从而直接输出至浏览器。

以下便是 MVC 框架中最核心的 DispatcherServlet 类,代码如下:

```
package org.smart4j.framework;

import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.Method;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.smart4j.framework.bean.Data;
import org.smart4j.framework.bean.Handler;
import org.smart4j.framework.bean.Param;
import org.smart4j.framework.bean.View;
import org.smart4j.framework.helper.BeanHelper;
import org.smart4j.framework.helper.ConfigHelper;
import org.smart4j.framework.helper.ControllerHelper;
import org.smart4j.framework.util.ArrayUtil;
import org.smart4j.framework.util.CodecUtil;
import org.smart4j.framework.util.JsonUtil;
import org.smart4j.framework.util.ReflectionUtil;
import org.smart4j.framework.util.StreamUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * 请求转发器
 *
 * @author huangyong
 * @since 1.0.0
 */
@WebServlet(urlPatterns = "/*", loadOnStartup = 0)
```

```
public class DispatcherServlet extends HttpServlet {

    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        // 初始化相关 Helper 类
        HelperLoader.init();
        // 获取 ServletContext 对象 (用于注册 Servlet)
        ServletContext servletContext = servletConfig.getServletContext();
        // 注册处理 JSP 的 Servlet
        ServletRegistration jspServlet = servletContext.getServlet-
            Registration("jsp");
        jspServlet.addMapping(ConfigHelper.getAppJspPath() + "*");
        // 注册处理静态资源的默认 Servlet
        ServletRegistration defaultServlet = servletContext.getServlet-
            Registration("default");
        defaultServlet.addMapping(ConfigHelper.getAppAssetPath() + "*");
    }

    @Override
    public void service(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        // 获取请求方法与请求路径
        String requestMethod = request.getMethod().toLowerCase();
        String requestPath = request.getPathInfo();
        // 获取 Action 处理器
        Handler handler = ControllerHelper.getHandler(requestMethod,
            requestPath);
        if (handler != null) {
            // 获取 Controller 类及其 Bean 实例
            Class<?> controllerClass = handler.getControllerClass();
            Object controllerBean = BeanHelper.getBean(controllerClass);
            // 创建请求参数对象
            Map<String, Object> paramMap = new HashMap<String, Object>();
            Enumeration<String> paramNames = request.getParameterNames();
            while (paramNames.hasMoreElements()) {
                String paramName = paramNames.nextElement();
                String paramValue = request.getParameter(paramName);
                paramMap.put(paramName, paramValue);
            }
        }
    }
}
```

```

    }
    String body = CodecUtil.decodeURL(StreamUtil.getString(
        request.getInputStream()));
    if (StringUtil.isNotEmpty(body)) {
        String[] params = StringUtil.splitString(body, "&");
        if (ArrayUtil.isNotEmpty(params)) {
            for (String param : params) {
                String[] array = StringUtil.splitString(param, "=");
                if (ArrayUtil.isNotEmpty(array) && array.length==2){
                    String paramName = array[0];
                    String paramValue = array[1];
                    paramMap.put(paramName, paramValue);
                }
            }
        }
    }
    Param param = new Param(paramMap);
    // 调用 Action 方法
    Method actionMethod = handler.getActionMethod();
    Object result = ReflectionUtil.invokeMethod(controllerBean,
        actionMethod, param);
    // 处理 Action 方法返回值
    if (result instanceof View) {
        // 返回 JSP 页面
        View view = (View) result;
        String path = view.getPath();
        if (StringUtil.isNotEmpty(path)) {
            if (path.startsWith("/")) {
                response.sendRedirect(request.getContextPath() + path);
            } else {
                Map<String, Object> model = view.getModel();
                for (Map.Entry<String, Object>entry:model.entrySet()){
                    request.setAttribute(entry.getKey(), entry.get-
                        Value());
                }
                request.getRequestDispatcher(ConfigHelper.getAppJsp-
                    Path() + path).forward(request, response);
            }
        }
    }
}

```

```

    }
    } else if (result instanceof Data) {
        // 返回 JSON 数据
        Data data = (Data) result;
        Object model = data.getModel();
        if (model != null) {
            response.setContentType("application/json");
            response.setCharacterEncoding("UTF-8");
            PrintWriter writer = response.getWriter();
            String json = JsonUtil.toJson(model);
            writer.write(json);
            writer.flush();
            writer.close();
        }
    }
}
}
}
}
}

```

在 `DispatcherServlet` 中用到了几个新的工具类。

其中，`StreamUtil` 类用于常用的流操作，代码如下：

```

package org.smart4j.framework.util;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 流操作工具类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class StreamUtil {

```

```
private static final Logger LOGGER = LoggerFactory.getLogger(StreamUtil.class);

/**
 * 从输入流中获取字符串
 */
public static String getString(InputStream is) {
    StringBuilder sb = new StringBuilder();
    try {
        BufferedReader reader = new BufferedReader(new InputStreamReader(is));
        String line;
        while ((line = reader.readLine()) != null) {
            sb.append(line);
        }
    } catch (Exception e) {
        LOGGER.error("get string failure", e);
        throw new RuntimeException(e);
    }
    return sb.toString();
}
```

CodecUtil 类用于编码与解码操作，代码如下：

```
package org.smart4j.framework.util;

import java.net.URLDecoder;
import java.net.URLEncoder;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 编码与解码操作工具类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class CodecUtil {
```

```
private static final Logger LOGGER = LoggerFactory.getLogger
    (CodecUtil.class);

/**
 * 将 URL 编码
 */
public static String encodeURL(String source) {
    String target;
    try {
        target = URLEncoder.encode(source, "UTF-8");
    } catch (Exception e) {
        LOGGER.error("encode url failure", e);
        throw new RuntimeException(e);
    }
    return target;
}

/**
 * 将 URL 解码
 */
public static String decodeURL(String source) {
    String target;
    try {
        target = URLDecoder.decode(source, "UTF-8");
    } catch (Exception e) {
        LOGGER.error("decode url failure", e);
        throw new RuntimeException(e);
    }
    return target;
}
}
```

JsonUtil 类用于处理 JSON 与 POJO 之间的转换，基于 Jackson 实现，代码如下：

```
package org.smart4j.framework.util;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

/**
 * JSON 工具类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class JsonUtil {

    private static final Logger LOGGER = LoggerFactory.getLogger(
        (JsonUtil.class));

    private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();

    /**
     * 将 POJO 转为 JSON
     */
    public static <T> String toJson(T obj) {
        String json;
        try {
            json = OBJECT_MAPPER.writeValueAsString(obj);
        } catch (Exception e) {
            LOGGER.error("convert POJO to JSON failure", e);
            throw new RuntimeException(e);
        }
        return json;
    }

    /**
     * 将 JSON 转为 POJO
     */
    public static <T> T fromJson(String json, Class<T> type) {
        T pojo;
        try {
            pojo = OBJECT_MAPPER.readValue(json, type);
        } catch (Exception e) {
            LOGGER.error("convert JSON to POJO failure", e);
        }
    }
}
```

```
        throw new RuntimeException(e);
    }
    return pojo;
}
}
```

至此，一款简单的 MVC 框架就开发完毕了，通过这个 `DispatcherServlet` 来处理所有的请求，根据请求信息从 `ControllerHelper` 中获取对应的 Action 方法，然后使用反射技术调用 Action 方法，同时需要具体的传入方法参数，最后拿到返回值并判断返回值的类型，进行相应的处理。

3.11 总结

在本章中，我们搭建了一个简单的 MVC 框架，定义了一系列注解：通过 `Controller` 注解来定义 `Controller` 类；通过 `Inejct` 注解来实现依赖注入；通过 `Action` 注解来定义 Action 方法。通过一系列的 `Helper` 类来初始化 MVC 框架；通过 `DispatcherServlet` 来处理所有的请求；根据请求方法与请求路径来调用具体的 Action 方法，判断 Action 方法的返回值，若为 `View` 类型，则跳转到 JSP 页面，若为 `Data` 类型，则返回 JSON 数据。

整个框架基本能跑起来了，但里面还存在大量需要优化的地方。此外，还有一些非常好的特性尚未提供，比如 AOP（Aspect Oriented Programming，面向方面编程）。我们可以使用这个特性来实现一些横向拦截操作，比如性能分析、日志收集、安全控制等，下一章我们将介绍如何实现这个特性。



第 4 章

使框架具备 AOP 特性

我们打算对方法调用进行性能监控,也就是说,在方法调用的时候统计出方法执行时间。

要实现这个功能,最容易想到的就是直接在方法的开头获取系统时间,然后在方法的结尾再次获取系统时间,最后把前后两次分别获取的系统时间做一个减法,即可获取方法执行所消耗的总时间。

这样做的确是可行的,但问题是我们的项目中有大量的方法,难道对每个方法都加上这些代码吗?这件事恐怕没人愿意去做。那么,我们不妨转换一下思路,能否不用修改现有代码,而是在另外一个地方做性能监控呢?AOP (Aspect Oriented Programming,面向方面编程)就是我们寻找的解决方案!

在 AOP 中,我们需要定义一个 Aspect (切面) 类来编写需要横切业务逻辑的代码,也就是上面提到的性能监控代码。此外,我们需要通过一个条件来匹配想要拦截的类,这个条件在 AOP 中称为 Pointcut (切点)。

我们还是用一个案例来实现这个思路:统计出执行每个 Controller 类的各个方法所消耗的时间。每个 Controller 类都带有 Controller 注解,也就是说,我们只需要拦截所有带有 Controller 注解的类就行了,切点很容易就能确定下来,剩下的就是做一个切面了。

在本章中,您将学到大量有用的技术,具体包括:

- 如何理解并使用代理技术;
- 如何使用 Spring 提供的 AOP 技术;
- 如何使用动态代理技术实现 AOP 框架;
- 如何理解并使用 ThreadLocal 技术;
- 如何理解数据库事务管理机制;
- 如何使用 AOP 框架实现事务控制。

下面,我们就从“代理技术”开始讲起吧!

4.1 代理技术简介

4.1.1 什么是代理

代理，或称为 Proxy。意思就是你不用去做，别人代替你去处理。比如说：赚钱方面，我就是我老婆的 Proxy；带小孩方面，我老婆就是我的 Proxy；家务事方面，没有 Proxy。

它在程序开发中起到了非常重要的作用，比如传说中的 AOP，就是针对代理的一种应用。此外，在设计模式中，还有一个“代理模式”。在公司里要上外网，要在浏览器里设置一个 HTTP 代理，可见代理无处不在。

凡事都要由浅入深，学习也不例外。先来看一个 Hello World：

```
public interface Hello {  
  
    void say(String name);  
}
```

上面是一个 Hello 接口，以下是实现类：

```
public class HelloImpl implements Hello {  
  
    @Override  
    public void say(String name) {  
        System.out.println("Hello! " + name);  
    }  
}
```

如果要在 println 方法前面和后面分别需要处理一些逻辑，怎么做呢？把这些逻辑写死在 say 方法里面吗？这么做肯定不够优雅，“菜鸟”一般这样干，作为一名资深的程序员，我们坚决不能这样做！

我们要用代理模式，写一个 HelloProxy 类，让它去调用 HelloImpl 的 say 方法，在调用的前后分别进行逻辑处理不就行了吗？代码如下：

```
public class HelloProxy implements Hello {  
  
    private Hello hello;
```

```
public HelloProxy() {  
    hello = new HelloImpl();  
}  
  
@Override  
public void say(String name) {  
    before();  
    hello.say(name);  
    after();  
}  
  
private void before() {  
    System.out.println("Before");  
}  
  
private void after() {  
    System.out.println("After");  
}  
}
```

用 `HelloProxy` 类实现了 `Hello` 接口(和 `HelloImpl` 实现相同的接口), 并且在构造方法中 `new` 出一个 `HelloImpl` 类的实例。这样一来, 我们就可以在 `HelloProxy` 的 `say` 方法里面去调用 `HelloImpl` 的 `say` 方法了。更重要的是, 我们还可以在调用的前后分别加上 `before` 与 `after` 方法, 在这两个方法里去实现那些前后逻辑。

用一个 `main` 方法来测试一下:

```
public static void main(String[] args) {  
    Hello helloProxy = new HelloProxy();  
    helloProxy.say("Jack");  
}
```

运行后, 打印结果如下:

```
Before  
Hello! Jack  
After
```

轻而易举, 我们就写出了这么优雅的代码(暗自高兴)。

我在一本设计模式的书上看到, 原来自己写的这个 `HelloProxy` 就是所谓的“代理模式”!

只能说，自己和 GoF 的距离又接近了一点。

4.1.2 JDK 动态代理

于是我疯狂地使用“代理模式”，项目中到处都有 XxxProxy 的声影。直到有一天，架构师看到了我的代码，他惊呆了！他对我说：“你怎么这么喜欢用静态代理呢？你就不会用动态代理吗？给我全都重构！”。

我表面上点了点头，说：“好的！”。其实我根本都不知道什么是“静态代理”，什么又是“动态代理”。我继续翻开那本垫桌脚的设计模式，深入地研究了一番，最后才明白，原来一直用的都是“静态代理”，怪不得架构师说到处都是 XxxProxy 类了。一定要将这些垃圾 Proxy 都重构为“动态代理”。

下面用 JDK 提供的动态代理方案写一个 DynamicProxy：

```
public class DynamicProxy implements InvocationHandler {

    private Object target;

    public DynamicProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        before();
        Object result = method.invoke(target, args);
        after();
        return result;
    }
    ...
}
```

在 DynamicProxy 类中，定义了一个 Object 类型的 target 变量，它就被代理的目标对象，通过构造函数来初始化（现在流行叫“注入”了，笔者觉得叫“射入”更加形象。构造方法初始化叫“正着射”，所以反射初始化就叫“反着射”，简称“反射”）。

言归正传，DynamicProxy 实现了 InvocationHandler 接口，那么必须实现该接口的 invoke

方法，参数不做解释，望文生义，它是 JRE 给我们“射”进来的。在该方法中，直接通过反射去 `invoke method`，在调用前后分别处理 `before` 与 `after`，最后将 `result` 返回。

写一个 `main` 方法看看实际怎么用：

```
public static void main(String[] args) {
    Hello hello = new HelloImpl();

    DynamicProxy dynamicProxy = new DynamicProxy(hello);

    Hello helloProxy = (Hello) Proxy.newProxyInstance(
        hello.getClass().getClassLoader(),
        hello.getClass().getInterfaces(),
        dynamicProxy
    );

    helloProxy.say("Jack");
}
```

意思就是用这个通用的 `DynamicProxy` 类去包装 `HelloImpl` 实例，然后再调用 JDK 给我们提供的 `Proxy` 类的工厂方法 `newProxyInstance` 去动态地创建一个 `Hello` 接口的代理类，最后调用这个代理类的 `say` 方法。

运行一下，结果和以前一样，动态代理成功了。其实，动态代理就是帮我们自动生成 `XxxProxy` 类的法宝。

要注意的是，`Proxy.newProxyInstance` 方法的参数实在是让人“醉了”！

- 参数 1: `ClassLoader`;
- 参数 2: 该实现类的所有接口;
- 参数 3: 动态代理对象。

调用完了还要来一个强制类型转换一下。

这一块一定要想办法封装一下，避免再次出现到处都是 `Proxy.newProxyInstance` 方法的情况。于是将这个 `DynamicProxy` 重构了一下：

```
public class DynamicProxy implements InvocationHandler {
    ...
    @SuppressWarnings("unchecked")
    public <T> T getProxy() {
        return (T) Proxy.newProxyInstance(
```

```

        target.getClass().getClassLoader(),
        target.getClass().getInterfaces(),
        this
    );
}
...
}

```

在 `DynamicProxy` 里添加了一个 `getProxy` 方法，无须传入任何参数，将刚才所说的那一块代码放在这个方法中，并且该方法返回一个泛型类型，就不会强制转换类型了。方法头上加 `@SuppressWarnings("unchecked")` 注解表示忽略编译时的警告（因为 `Proxy.newProxyInstance` 方法返回的是一个 `Object`，这里强制转换为 `T` 了，这是向下转型，IDE 中就会有警告，编译时也会出现提示）。

这下使用 `DynamicProxy` 就简单了：

```

public static void main(String[] args) {
    DynamicProxy dynamicProxy = new DynamicProxy(new HelloImpl());
    Hello helloProxy = dynamicProxy.getProxy();

    helloProxy.say("Jack");
}

```

确实简单，用 2 行代理就去掉了前面的 7 行代码（省了 5 行），经过一番代码重构后，总算学会使用动态代理了。

4.1.3 CGLib 动态代理

用了这个 `DynamicProxy` 以后，我觉得它还是非常好的，好的地方是，接口变了，这个动态代理类不用动。而静态代理就不一样了，接口变了，实现类还要动，代理类也要动。但动态代理并不是“万灵丹”，它也有搞不定的时候，比如要代理一个没有任何接口的类，它就没有用武之地了！

于是我又开始调研，能否代理没有接口的类呢？终于找到了这颗“银弹”！那就是 `CGLib` 这个类库。虽然它看起来不太起眼，但 `Spring`、`Hibernate` 这样高端的开源框架都用到了它。它是一个在运行期间动态生成字节码的工具，也就是动态生成代理类了。说起来好高深，实际用起来一点都不难。下面再写一个 `CGLibProxy`：

```

public class CGLibProxy implements MethodInterceptor {

```

```

public <T> T getProxy(Class<T> cls) {
    return (T) Enhancer.create(cls, this);
}

public Object intercept(Object obj, Method method, Object[] args,
    MethodProxy proxy) throws Throwable {
    before();
    Object result = proxy.invokeSuper(obj, args);
    after();
    return result;
}

...
}

```

需要实现 CGLib 给我们提供的 `MethodInterceptor` 实现类，并填充 `intercept` 方法。方法中最后一个 `MethodProxy` 类型的参数 `proxy` 值得注意。CGLib 给我们提供的是方法级别的代理，也可以理解为对方法的拦截（这不就是传说中的“方法拦截器”吗？）。这个功能对于我们程序员而言，如同雪中送炭。我们直接调用 `proxy` 的 `invokeSuper` 方法，将被代理的对象 `obj` 以及方法参数 `args` 传入其中即可。

与 `DynamicProxy` 类似，在 `CGLibProxy` 中也添加一个泛型的 `getProxy` 方法，便于我们可以快速地获取自动生成的代理对象。下面用一个 `main` 方法来描述：

```

public static void main(String[] args) {
    CGLibProxy cgLibProxy = new CGLibProxy();
    Hello helloProxy = cgLibProxy.getProxy(HelloImpl.class);
    helloProxy.say("Jack");
}

```

仍然通过 2 行代码就可以返回代理对象，与 JDK 动态代理不同的是，这里不需要任何的接口信息，对谁都可以生成动态代理对象（不管它是“矮穷挫”还是“高富帅”）。

由于我一向都追求完美，用 2 行代码返回代理对象还是有些多余的，不想总是去 `new` 这个 `CGLibProxy` 对象，最好 `new` 一次，以后随时拿随时用，于是想到了“单例模式”：

```

public class CGLibProxy implements MethodInterceptor {

    private static CGLibProxy instance = new CGLibProxy();

    private CGLibProxy() {

```

```
    }

    public static CGLibProxy getInstance() {
        return instance;
    }

    ...
}
```

加了以上几行代码就解决问题了。需要说明的是，这里有一个 `private` 的构造方法，就是为了限制外界不能再去 `new` 它了，换句话说，这个类被“阉割”了。

用一个 `main` 方法来证明我的简单主义思想：

```
public static void main(String[] args) {
    Hello helloProxy = CGLibProxy.getInstance().getProxy(HelloImpl.class);
    helloProxy.say("Jack");
}
```

没错，只需 1 行代码就可以获取代理对象了！

以上讲解了无代理、静态代理、JDK 动态代理、CGLib 动态代理，其实代理的世界远不止这么小，还有很多实际的应用场景。本节一开始谈到的 AOP 是一个最为典型的案例，所以有必要再进行继续下去。

4.2 AOP 技术简介

4.2.1 什么是 AOP

AOP (Aspect-Oriented Programming)，名字与 OOP 仅差一个字母，其实它是对 OOP 编程方式的一种补充，并非是取而代之。翻译过来就是“面向方面编程”，我更倾向于翻译为“面向切面编程”。它听起有些神秘，为什么呢？我们做的很重要的工作就是去写这个“切面”。那么什么是“切面”呢？

切面是 AOP 中的一个术语，表示从业务逻辑中分离出来的横切逻辑，比如性能监控、日志记录、权限控制等，这些功能都可从核心的业务逻辑代码中抽离出去。也就是说，通过 AOP 可以解决代码耦合问题，让职责更加单一。

需要澄清的是，其实很早以前就出现了 AOP 这个概念。最知名最强大的 Java 开源项目就是 AspectJ 了，它的前身是 AspectWerkz（该项目已经在 2005 年停止更新），这才是 AOP 的老

祖宗。直到后来，老罗（Rod Johnson，一个头发秃得和我老爸有一拼的天才），写了一个叫 Spring 的框架，从此“一炮走红”，成为了 Spring 之父。他在 Spring 的 IOC 框架基础上又实现了一套 AOP 框架，后来发现自己掉进了深渊里，在无法自拔的时候，网友建议他还是集成 AspectJ 吧，他在万般无奈之下，才接受了该建议。于是，我们现在用的最多的想必就是 Spring+AspectJ 这种 AOP 框架了。

那么 AOP 到底是什么？如何去使用它？本节将逐步揭开 AOP 神秘的面纱！

不过在开始讲解 AOP 之前，有必要先回忆一下这段代码。

4.2.2 写死代码

先来一个接口：

```
public interface Greeting {  
  
    void sayHello(String name);  
}
```

还有一个实现类：

```
public class GreetingImpl implements Greeting {  
  
    @Override  
    public void sayHello(String name) {  
        before();  
        System.out.println("Hello! " + name);  
        after();  
    }  
  
    private void before() {  
        System.out.println("Before");  
    }  
  
    private void after() {  
        System.out.println("After");  
    }  
}
```

before 与 after 方法写死在 sayHello 方法体中了，这样的代码的“味道”非常不好。如果哪位程序员大量写了这样的代码，肯定要被架构师骂个够呛。

比如我们要统计每个方法的执行时间，以对性能做出评估，那是不是要在每个方法的一头一尾都做点手脚呢？

再比如我们要写一个 JDBC 程序，那是不是也要在方法的开头去连接数据库，方法的末尾去关闭数据库连接呢？

这样的代码只会把程序员累死，把架构师气死！

一定要想办法对上面的代码进行重构，首先给出三个解决方案：

- 静态代理；
- JDK 动态代理；
- CGLib 动态代理。

下面分别对以上解决方案进行描述。

4.2.3 静态代理

最简单的解决方案就是使用静态代理模式了，我们单独为 `GreetingImpl` 这个类写一个代理类：

```
public class GreetingProxy implements Greeting {

    private GreetingImpl greetingImpl;

    public GreetingProxy(GreetingImpl greetingImpl) {
        this.greetingImpl = greetingImpl;
    }

    @Override
    public void sayHello(String name) {
        before();
        greetingImpl.sayHello(name);
        after();
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
```

```
        System.out.println("After");  
    }  
}
```

就用这个 `GreetingProxy` 去代理 `GreetingImpl`，下面看看客户端如何来调用：

```
public class Client {  
  
    public static void main(String[] args) {  
        Greeting greetingProxy = new GreetingProxy(new GreetingImpl());  
        greetingProxy.sayHello("Jack");  
    }  
}
```

这样写没错，但是有个问题，`XxxProxy` 这样的类会越来越多，如何才能将这些代理类尽可能减少呢？最好只有一个代理类。

这时我们就需要使用 JDK 提供的动态代理了。

4.2.4 JDK 动态代理

```
public class JDKDynamicProxy implements InvocationHandler {  
    private Object target;  
  
    public JDKDynamicProxy(Object target) {  
        this.target = target;  
    }  
  
    @SuppressWarnings("unchecked")  
    public <T> T getProxy() {  
        return (T) Proxy.newProxyInstance(  
            target.getClass().getClassLoader(),  
            target.getClass().getInterfaces(),  
            this  
        );  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws  
        Throwable {
```

```
        before();
        Object result = method.invoke(target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

客户端是这样调用的：

```
public class Client {

    public static void main(String[] args) {
        Greeting greeting = new JDKDynamicProxy(new GreetingImpl())
            .getProxy();
        greeting.sayHello("Jack");
    }
}
```

这样所有的代理类都合并到动态代理类中了，但这样做仍然存在一个问题：JDK 给我们提供的动态代理只能代理接口，而不能代理没有接口的类。有什么方法可以解决这个问题呢？

4.2.5 CGLib 动态代理

我们使用开源的 CGLib 类库可以代理没有接口的类，这样就弥补了 JDK 的不足。CGLib 动态代理类是这样的：

```
public class CGLibDynamicProxy implements MethodInterceptor {

    private static CGLibDynamicProxy instance = new CGLibDynamicProxy();

    private CGLibDynamicProxy() {
```

```
    }

    public static CGLibDynamicProxy getInstance() {
        return instance;
    }

    @SuppressWarnings("unchecked")
    public <T> T getProxy(Class<T> cls) {
        return (T) Enhancer.create(cls, this);
    }

    @Override
    public Object intercept(Object target, Method method, Object[] args,
        MethodProxy proxy) throws Throwable {
        before();
        Object result = proxy.invokeSuper(target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

以上代码中使用了 **Singleton** 模式，那么客户端调用也更加轻松了：

```
public class Client {

    public static void main(String[] args) {
        Greeting greeting = CGLibDynamicProxy.getInstance().getProxy(
            GreetingImpl.class);
        greeting.sayHello("Jack");
    }
}
```

到此为止，我们能做的都做了，问题似乎全部都解决了。但事情总不会那么完美，而我们一定要追求完美！

4.2.6 Spring AOP

老罗搞出了一个 AOP 框架，能否做到完美而优雅呢？

1. Spring AOP：前置增强、后置增强、环绕增强（编程式）

在 Spring AOP 的世界里，与 AOP 相关的术语实在太多，这往往也是我们的“拦路虎”，不管是看哪本图书或是技术文档，在开头都要将这些术语逐个灌输给我们。这完全是在吓唬人，其实没那么复杂，我们可以放轻松一点。

上面例子中提到的 `before` 方法，在 Spring AOP 里就叫 `Before Advice`（前置增强）。有些人将 `Advice` 直译为“通知”，这是不太合适的，因为它根本就没有“通知”的含义，而是对原有代码功能的一种“增强”。再者，CGLib 中也有一个 `Enhancer` 类，它就是一个增强类。

此外，像 `after` 这样的方法就叫 `After Advice`（后置增强），因为它放在后面来增强代码的功能。

如果能把 `before` 与 `after` 合并在一起，那就叫 `Around Advice`（环绕增强），就像汉堡一样，中间夹一根火腿（形象吗？）。

这三个概念是不是轻松地理解了呢？如果是，那就继续吧！

我们下面要做的就是去实现这些所谓的“增强类”，让它们横切到代码中，而不是将这些写死在代码中。

先来一个前置增强类：

```
public class GreetingBeforeAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws
        Throwable {
        System.out.println("Before");
    }
}
```

注意：这个类实现了 `org.springframework.aop.MethodBeforeAdvice` 接口，我们将需要增强的代码放入其中。

再来一个后置增强类：

```
public class GreetingAfterAdvice implements AfterReturningAdvice {

    @Override
    public void afterReturning(Object result, Method method, Object[] args,
        Object target) throws Throwable {
        System.out.println("After");
    }
}
```

类似地，这个类实现了 `org.springframework.aop.AfterReturningAdvice` 接口。

最后用一个客户端把它们集成起来，看看如何调用：

```
public class Client {

    public static void main(String[] args) {
        ProxyFactory proxyFactory = new ProxyFactory(); // 创建代理工厂
        proxyFactory.setTarget(new GreetingImpl()); // 射入目标类对象
        proxyFactory.addAdvice(new GreetingBeforeAdvice()); // 添加前置增强
        proxyFactory.addAdvice(new GreetingAfterAdvice()); // 添加后置增强

        Greeting greeting = (Greeting) proxyFactory.getProxy();
                                                // 从代理工厂中获取代理
        greeting.sayHello("Jack"); // 调用代理的方法
    }
}
```

仔细阅读以上代码及其注释就会发现，其实 Spring AOP 还是挺简单的。

当然，我们完全可以只定义一个增强类，让它同时实现 `MethodBeforeAdvice` 与 `AfterReturningAdvice` 这两个接口，代码如下：

```
public class GreetingBeforeAndAfterAdvice implements MethodBeforeAdvice,
    AfterReturningAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws
        Throwable {
        System.out.println("Before");
    }
}
```

```
    }

    @Override
    public void afterReturning(Object result, Method method, Object[] args,
        Object target) throws Throwable {
        System.out.println("After");
    }
}
```

这样我们只需要使用一行代码，就可以同时添加前置与后置增强：

```
proxyFactory.addAdvice(new GreetingBeforeAndAfterAdvice());
```

刚才有提到“环绕增强”，其实它可以把“前置增强”与“后置增强”的功能给合并起来，无须让我们同时实现以上两个接口。

```
public class GreetingAroundAdvice implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        before();
        Object result = invocation.proceed();
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

环绕增强类需要实现 `org.aopalliance.intercept.MethodInterceptor` 接口。注意，这个接口不是 Spring 提供的，它是 AOP 联盟（一个很“高大上”的技术联盟）写的，Spring 只是借用了它。

在客户端中同样也需要将该增强类的对象添加到代理工厂中：

```
proxyFactory.addAdvice(new GreetingAroundAdvice());
```

以上这就是 Spring AOP 的基本用法，但这只是“程式”而已。Spring AOP 如果只是这样，那就太弱了，它曾经也一度宣传用 Spring 配置文件的方式来定义 Bean 对象，把代码中的 new 操作全部解脱出来。

2. Spring AOP：前置增强、后置增强、环绕增强（声明式）

先看 Spring 配置文件是如何写的：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 扫描指定包（将带有 Component 注解的类自动定义为 Spring Bean） -->
    <context:component-scan base-package="aop.demo"/>

    <!-- 配置一个代理 -->
    <bean id="greetingProxy" class="org.springframework.aop.framework.
ProxyFactoryBean">
        <property name="interfaces" value="aop.Greeting"/>
        <!-- 需要代理的接口 -->
        <property name="target" ref="greetingImpl"/> <!-- 接口实现类 -->
        <property name="interceptorNames"> <!-- 拦截器名称（也就是增强类名称，
Spring Bean 的 id） -->

            <list>
                <value>greetingAroundAdvice</value>
            </list>
        </property>
    </bean>

</beans>
```

一定要阅读以上代码的注释，其实使用 ProxyFactoryBean 就可以取代前面的 ProxyFactory，其实它们是一回事儿。interceptorNames 改名为 adviceNames 或许会更容易让人理解，不就是往这个属性里面添加增强类吗？

此外，如果只有一个增强类，可以使用以下方法来简化：

```
...
<bean id="greetingProxy" class="org.springframework.aop.framework.
ProxyFactoryBean">
    <property name="interfaces" value="aop.Greeting"/>
    <property name="target" ref="greetingImpl"/>
    <property name="interceptorNames" value="greetingAroundAdvice"/>
    <!-- 注意这行配置 -->

</bean>
...
```

还需要注意的是，这里使用了 Spring 2.5+ 的“Bean 扫描”特性，这样我们就无须在 Spring 配置文件里不断地定义<bean id="xxx" class="xxx"/>了，从而解脱了我们的双手。

看看这是多么简单：

```
@Component
public class GreetingImpl implements Greeting {
    ...
}
@Component
public class GreetingAroundAdvice implements MethodInterceptor {
    ...
}
```

最后看看客户端：

```
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext
("aop/demo/spring.xml"); // 获取 Spring Context
        Greeting greeting = (Greeting) context.getBean("greetingProxy");
        // 从 Context 中根据 id 获取 Bean 对象（其实就是一个代理）
        greeting.sayHello("Jack"); // 调用代理的方法
    }
}
```

代码量确实少了，我们将配置性的代码放入配置文件，这样也有助于后期维护。更重要的是，代码只关注于业务逻辑，而将配置放入文件中，这是一条最佳实践！

除了上面提到的那三类增强以外，其实还有两类增强也需要了解一下，关键的时候要能想得到它们才行。

3. Spring AOP：抛出增强

程序报错，抛出异常了，一般的做法是打印到控制台或日志文件中，这样很多地方都得去处理，有没有一个一劳永逸的方法呢？那就是 **Throws Advice**（抛出增强）：

```
@Component
public class GreetingImpl implements Greeting {

    @Override
    public void sayHello(String name) {
        System.out.println("Hello! " + name);

        throw new RuntimeException("Error");
        // 故意抛出一个异常，看看异常信息能否被拦截到
    }
}
```

下面是抛出增强类的代码：

```
@Component
public class GreetingThrowAdvice implements ThrowsAdvice {

    public void afterThrowing(Method method, Object[] args, Object target,
        Exception e) {
        System.out.println("----- Throw Exception -----");
        System.out.println("Target Class: " + target.getClass().getName());
        System.out.println("Method Name: " + method.getName());
        System.out.println("Exception Message: " + e.getMessage());
        System.out.println("-----");
    }
}
```

抛出增强类需要实现 `org.springframework.aop.ThrowsAdvice` 接口，在接口方法中可获取方法、参数、目标对象、异常对象等信息。我们可以把这些信息统一写入到日志中，当然也可以持久化到数据库中。

这个功能确实太棒了！但还有一个更厉害的增强。如果某个类实现了 **A** 接口，但没有实现 **B** 接口，那么该类可以调用 **B** 接口的方法吗？如果没有看到下面的内容，一定不敢相信原来这是可行的！

4. Spring AOP: 引入增强

以上提到的都是对方法的增强, 那能否对类进行增强呢? 用 AOP 的行话来讲, 对方法的增强叫 **Weaving** (织入), 而对类的增强叫 **Introduction** (引入)。**Introduction Advice** (引入增强) 就是对类的功能增强, 它也是 **Spring AOP** 提供的最后一种增强。建议读者一开始千万不要去看《**Spring Reference**》, 否则一定会后悔的。因为当看了以下的代码示例后, 一定会彻底明白什么才是引入增强。

定义一个新接口 **Apology** (道歉):

```
public interface Apology {  
  
    void saySorry(String name);  
  
}
```

但我们不想在代码中让 **GreetingImpl** 直接去实现这个接口, 而想在程序运行的时候动态地实现它。因为假如实现了这个接口, 那么就一定要改写 **GreetingImpl** 这个类, 关键是我们不想改它, 或许在真实场景中, 这个类有 1 万行代码。于是, 我们需要借助 **Spring** 的引入增强。

```
@Component  
public class GreetingIntroAdvice extends DelegatingIntroductionInterceptor  
implements Apology {  
  
    @Override  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        return super.invoke(invocation);  
    }  
  
    @Override  
    public void saySorry(String name) {  
        System.out.println("Sorry! " + name);  
    }  
  
}
```

以上定义了一个引入增强类, 扩展了 **org.springframework.aop.support.DelegatingIntroductionInterceptor** 类, 同时也实现了新定义的 **Apology** 接口。在类中首先覆盖了父类的 **invoke()** 方法, 然后实现了 **Apology** 接口的方法。我们想用这个增强类去丰富 **GreetingImpl** 类的功能, 那么这个 **GreetingImpl** 类无须直接实现 **Apology** 接口, 就可以在程序运行的时候调用 **Apology** 接口的方法了。这简直是太神奇了。

看看是如何配置的：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="aop.demo"/>

    <bean id="greetingProxy" class="org.springframework.aop.framework.
ProxyFactoryBean">
        <property name="interfaces" value="aop.demo.Apology"/>
                                <!-- 需要动态实现的接口 -->
        <property name="target" ref="greetingImpl"/> <!-- 目标类 -->
        <property name="interceptorNames" value="greetingIntroAdvice"/>
                                <!-- 引入增强 -->
        <property name="proxyTargetClass" value="true"/>
                                <!-- 代理目标类（默认为 false，代理接口） -->
    </bean>

</beans>
```

需要注意 `proxyTargetClass` 属性，它表明是否代理目标类，默认为 `false`，也就是代理接口，此时 Spring 就用 JDK 动态代理；如果为 `true`，那么 Spring 就用 CGLib 动态代理。这简直太方便了。Spring 封装了这一切，让程序员不在关心那么多的细节。我们要向老罗同志致敬，他是我们心中永远的偶像！

当看完下面的客户端代码，您一定会完全明白以上的这一切：

```
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext
            ("aop/demo/spring.xml");
        GreetingImpl greetingImpl = (GreetingImpl) context.getBean("greetingProxy");
        // 注意：转型为目标类，而并非它的 Greeting 接口
    }
}
```

```

        greetingImpl.sayHello("Jack");

        Apology apology = (Apology) greetingImpl; // 将目标类强制向上转型为
            Apology 接口 (这是引入增强给我们带来的特性, 也就是“接口动态实现”功能)
        apology.saySorry("Jack");
    }
}

```

没想到 `saySorry` 方法原来是可以被 `greetingImpl` 对象来直接调用的, 只需将其强制转换为该接口即可。

沿着 Spring AOP 的方向, 老罗花了不少心思, 都是为了让使用 Spring 框架时不会感麻烦, 但事实却并非如此。那么, 后来老罗究竟对 Spring AOP 做了哪些改进呢?

5. Spring AOP: 切面

之前谈到的 AOP 框架其实可以将它理解为一个拦截器框架, 但这个拦截器似乎非常武断。比如说, 如果它拦截了一个类, 那么它就拦截了这个类中所有的方法。类似地, 当我们在使用动态代理的时候, 其实也遇到了这个问题。需要在代码中对所拦截的方法名加以判断, 才能过滤出我们需要拦截的方法, 这种做法确实不太优雅。在大量的真实项目中, 似乎我们只需要拦截特定的方法就行了, 没必要拦截所有的方法。于是, 老罗同志借助了 AOP 的一个很重要的工具——Advisor (切面), 来解决这个问题。它也是 AOP 中的核心, 是我们关注的重点。

也就是说, 我们可以通过切面, 将增强类与拦截匹配条件组合在一起, 然后将这个切面配置到 `ProxyFactory` 中, 从而生成代理。

这里提到这个“拦截匹配条件”在 AOP 中就叫作 `Pointcut` (切点), 其实说白了就是一个基于表达式的拦截条件。`Advisor` (切面) 封装了 `Advice` (增强) 与 `Pointcut` (切点)。

笔者我们在 `GreetingImpl` 类中故意增加了两个方法, 都以“good”开头。下面要做的就是拦截这两个新增的方法, 而对 `sayHello()` 方法不作拦截。

```

@Component
public class GreetingImpl implements Greeting {

    @Override
    public void sayHello(String name) {
        System.out.println("Hello! " + name);
    }

    public void goodMorning(String name) {
        System.out.println("Good Morning! " + name);
    }
}

```

```

    }

    public void goodNight(String name) {
        System.out.println("Good Night! " + name);
    }
}

```

在 Spring AOP 中，老罗已经给我们提供了许多切面类了，这些切面类个人感觉最好用的就是基于正则表达式的切面类：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <context:component-scan base-package="aop.demo"/>

    <!-- 配置一个切面 -->
    <bean id="greetingAdvisor" class="org.springframework.aop.support.
    RegexMethodPointcutAdvisor">
        <property name="advice" ref="greetingAroundAdvice"/> <!-- 增强 -->
        <property name="pattern" value="aop.demo.GreetingImpl.good.*"/>
                                <!-- 切点（正则表达式） -->
    </bean>

    <!-- 配置一个代理 -->
    <bean id="greetingProxy" class="org.springframework.aop.framework.
    ProxyFactoryBean">
        <property name="target" ref="greetingImpl"/> <!-- 目标类 -->
        <property name="interceptorNames" value="greetingAdvisor"/>
                                <!-- 切面 -->
        <property name="proxyTargetClass" value="true"/> <!-- 代理目标类 -->
    </bean>

</beans>

```

注意以上代理对象的配置中的 `interceptorNames`，它不再是一个增强，而是一个切面，因为已经将增强封装到该切面中了。此外，切面还定义了一个切点（正则表达式），其目的是为了只对满足切点匹配条件的方法进行拦截。

需要强调的是，这里的切点表达式是基于正则表达式的。示例中的 `aop.demo.GreetingImpl.good.*` 表达式后面的 `*` 表示匹配所有字符，翻译过来就是匹配 `aop.demo.GreetingImpl` 类中以

good 开头的方法。

除了 `RegexpMethodPointcutAdvisor` 以外，在 Spring AOP 中还提供了几个切面类，比如：

- `DefaultPointcutAdvisor`——默认切面（可扩展它来自定义切面）；
- `NameMatchMethodPointcutAdvisor`——根据方法名称进行匹配的切面；
- `StaticMethodMatcherPointcutAdvisor`——用于匹配静态方法的切面。

总的来说，让用户去配置一个或少数几个代理，似乎还可以接受，但随着项目的扩大，代理配置就会越来越多，配置的重复劳动就多了，麻烦不说，还很容易出错。能否让 Spring 框架为我们自动生成代理呢？

6. Spring AOP：自动代理（扫描 Bean 名称）

Spring AOP 提供了一个可根据 Bean 名称来自动生成代理的工具，它就是 `BeanNameAutoProxyCreator`。它是这样配置的：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    ...

    <bean class="org.springframework.aop.framework.autoproxy.
BeanNameAutoProxyCreator">
        <property name="beanNames" value="*Impl"/> <!-- 只为后缀是“Impl”的
                                Bean 生成代理 -->
        <property name="interceptorNames" value="greetingAroundAdvice"/>
                                <!-- 增强 -->
        <property name="optimize" value="true"/>
                                <!-- 是否对代理生成策略进行优化 -->
    </bean>

</beans>
```

以上使用 `BeanNameAutoProxyCreator` 只为后缀为“Impl”的 Bean 生成代理。需要注意的是，这个地方我们不能定义代理接口，也就是 `interfaces` 属性，因为我们根本就不知道这些 Bean 到底实现了多少接口。此时不能代理接口，而只能代理类。所以这里提供了一个新的配置项，它就是 `optimize`。若为 `true` 时，则可对代理生成策略进行优化（默认是 `false`）。也就是说，如果该类有接口，就代理接口（使用 JDK 动态代理）；如果没有接口，就代理类（使用 CGLib 动态代理）。并非像之前使用的 `proxyTargetClass` 属性那样，强制代理类，而不考虑代理接口的方式。可见 Spring AOP 确实为我们提供了很多很好的服务。

既然 CGLib 可以代理任何类了，那为什么还要用 JDK 的动态代理呢？

根据实际项目经验得知，CGLib 创建代理的速度比较慢，但创建代理后运行的速度却非常快，而 JDK 动态代理正好相反。如果在运行的时候不断地用 CGLib 去创建代理，系统的性能会大打折扣，所以建议一般在系统初始化的时候用 CGLib 去创建代理，并放入 Spring 的 ApplicationContext 中以备后用。

以上这个例子只能匹配目标类，而不能进一步匹配其中指定的方法，要匹配方法，就要考虑使用切面与切点了。Spring AOP 基于切面也提供了一个自动代理生成器：DefaultAdvisorAutoProxyCreator。

7. Spring AOP：自动代理（扫描切面配置）

为了匹配目标类中的指定方法，我们仍然需要在 Spring 中配置切面与切点：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    ...

    <bean id="greetingAdvisor" class="org.springframework.aop.support.
    RegexMethodPointcutAdvisor">
        <property name="pattern" value="aop.demo.GreetingImpl.good.*"/>
        <property name="advice" ref="greetingAroundAdvice"/>
    </bean>

    <bean class="org.springframework.aop.framework.autoproxy.Default-
    AdvisorAutoProxyCreator">
        <property name="optimize" value="true"/>
    </bean>

</beans>
```

这里无须再配置代理，因为代理将由 DefaultAdvisorAutoProxyCreator 自动生成。也就是说，这个类可以扫描所有的切面类，并为其自动生成代理。

看来不管怎样简化，老罗始终解决不了切面的配置这件繁重的手工劳动。在 Spring 配置文件中，仍然会存在大量的切面配置。然而在很多情况下，Spring AOP 所提供的切面类真的不太够用了，比如想拦截指定注解的方法，我们就必须扩展 DefaultPointcutAdvisor 类，自定义一个切面类，然后在 Spring 配置文件中对切面进行配置。

老罗的解决方案似乎已经掉进了切面类的深渊，这还真是所谓的“面向切面编程”了，最重要的是切面，最麻烦的也是切面。

必须要把切面配置给简化掉，Spring 才能有所突破！

4.2.7 Spring + AspectJ

神一样的老罗总算认识到了这一点，接受了网友们的建议，集成了 AspectJ，同时也保留了以上提到的切面与代理配置方式（为了兼容老的项目，更为了维护自己的面子）。将 Spring 与 AspectJ 集成与直接使用 AspectJ 是不同的，我们不需要定义 AspectJ 类（它扩展了 Java 语法的一种新的语言，还需要特定的编译器），只需要使用 AspectJ 切点表达式即可（它是比正则表达式更加友好的表现形式）。

1. Spring+AspectJ（基于注解：通过 AspectJ execution 表达式拦截方法）

下面以一个最简单的例子来实现之前提到的环绕增强。先定义一个 Aspect 切面类：

```
@Aspect
@Component
public class GreetingAspect {

    @Around("execution(* aop.demo.GreetingImpl.*(..))")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        before();
        Object result = pjp.proceed();
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

注意：类上面标注的 Aspect 注解表明该类是一个 Aspect（其实就是 Advisor）。该类无须实现任何的接口，只需定义一个方法（方法叫什么名字都无所谓），在方法上标注 Around 注解，在注解中使用 AspectJ 切点表达式。方法的参数中包括一个 ProceedingJoinPoint 对象，它在 AOP 中称为 Joinpoint（连接点），可以通过该对象获取方法的任何信息，例如，方法名、参数等。

下面重点来分析一下这个切点表达式：

```
execution(* aop.demo.GreetingImpl.*(..))
```

- execution()表示拦截方法，括号中可定义需要匹配的规则；
- 第一个“*”表示方法的返回值是任意的；
- 第二个“*”表示匹配该类中所有的方法；
- (..)表示方法的参数是任意的。

是不是比正则表达式的可读性更强呢？如果想匹配指定的方法，只需将第二个“*”改为指定的方法名称即可。

具体配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="aop.demo"/>

    <aop:aspectj-autoproxy proxy-target-class="true"/>

</beans>
```

两行配置就行了，不需要配置大量的代理，更不需要配置大量的切面，真是太棒了！需要注意的是 `proxy-target-class="true"` 属性，它的默认值是 `false`，默认只能代理接口（使用 JDK 动态代理），当为 `true` 时，才能代理目标类（使用 CGLib 动态代理）。

Spring 与 AspectJ 结合的威力远远不止这些，我们来点时尚的，拦截指定注解的方法怎么样？

2. Spring+AspectJ（基于注解：通过 AspectJ @annotation 表达式拦截方法）

为了拦截指定的注解的方法，我们首先需要来自定义一个注解：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Tag {
}
```

以上定义了一个 `Tag` 注解，此注解可标注在方法上，在运行时生效。

只需将前面的 `Aspect` 类的切点表达式稍作改动：

```
@Aspect
@Component
public class GreetingAspect {

    @Around("@annotation(aop.demo.Tag)")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        ...
    }
    ...
}
```

这次使用了 `@annotation()` 表达式，只需在括号内定义需要拦截的注解名称即可。

直接将 `Tag` 注解定义在想要拦截的方法上，就这么简单：

```
@Component
public class GreetingImpl implements Greeting {

    @Tag
    @Override
    public void sayHello(String name) {
        System.out.println("Hello! " + name);
    }
}
```

以上示例中只有一个方法，如果有多个方法，我们只想拦截其中的某一些时，这种解决方

案会更加有价值。

除了 `Around` 注解外，其实还有几个相关的注解，稍微归纳一下：

- `Before`——前置增强；
- `After`——后置增强；
- `Around`——环绕增强；
- `AfterThrowing`——抛出增强；
- `DeclareParents`——引入增强。

此外还有一个 `AfterReturning`（返回后增强），也可理解为 `Finally` 增强，相当于 `finally` 语句，它是在方法结束后执行的，也就是说，它比 `After` 还要晚一些。

最后一个 `DeclareParents` 竟然就是引入增强！为什么不叫 `Introduction` 呢？我也不知道为什么，但它干的活就是引入增强。

3. Spring + AspectJ（引入增强）

为了实现基于 `AspectJ` 的引入增强，我们同样需要定义一个 `Aspect` 类：

```
@Aspect
@Component
public class GreetingAspect {

    @DeclareParents(value = "aop.demo.GreetingImpl", defaultImpl =
        ApologyImpl.class)
    private Apology apology;
}
```

在 `Aspect` 类中定义一个需要引入增强的接口，它也就是运行时需要动态实现的接口。在这个接口上标注了 `DeclareParents` 注解，该注解有两个属性：

- `Value`——目标类；
- `defaultImpl`——引入接口的默认实现类。

我们只需要对引入的接口提供一个默认实现类即可完成引入增强：

```
public class ApologyImpl implements Apology {

    @Override
    public void saySorry(String name) {
        System.out.println("Sorry! " + name);
    }
}
```

以上这个实现会在运行时自动增强到 `GreetingImpl` 类中，也就是说，无须修改 `GreetingImpl` 类的代码，让它去实现 `Apology` 接口，我们单独为该接口提供一个实现类（`ApologyImpl`）来做 `GreetingImpl` 想做的事情。

还是用一个客户端来尝试一下：

```
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "aop/demo/spring.xml");
        Greeting greeting = (Greeting) context.getBean("greetingImpl");
        greeting.sayHello("Jack");

        Apology apology = (Apology) greeting; // 强制转型为 Apology 接口
        apology.saySorry("Jack");
    }
}
```

从 `Spring ApplicationContext` 中获取 `greetingImpl` 对象（其实是个代理对象），可转型为自己静态实现的接口 `Greeting`，也可转型为自己动态实现的接口 `Apology`，切换起来非常方便。

使用 `AspectJ` 的引入增强比原来的 `Spring AOP` 的引入增强更加方便了，而且还可面向接口编程（以前只能面向实现类），这也算一个非常巨大的突破。

这一切真的已经非常强大也非常灵活了，但仍然还是有用户不能尝试这些特性，因为他们还在使用 `JDK 1.4`（根本就没有注解这个东西），怎么办呢？没想到 `Spring AOP` 为那些遗留系统也考虑到了。

4. Spring+AspectJ（基于配置）

除了使用 `Aspect` 注解来定义切面类之外，`Spring AOP` 也提供了基于配置的方式来定义切面类：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="greetingImpl" class="aop.demo.GreetingImpl"/>

    <bean id="greetingAspect" class="aop.demo.GreetingAspect"/>

    <aop:config>
```

```

<aop:aspect ref="greetingAspect">
    <aop:around method="around" pointcut="execution(* aop.demo.
        GreetingImpl.*(..))"/>
</aop:aspect>
</aop:config>

</beans>

```

使用<aop:config>元素来进行 AOP 配置，在其子元素中配置切面，包括增强类型、目标方法、切点等信息。

无论用户是不能使用注解，还是不愿意使用注解，Spring AOP 都能提供全方位的服务。

我所知道的比较实用的 AOP 技术都在这里了，当然还有一些更为高级的特性，由于个人水平有限，这里就不再深入了。

还是依照惯例，给出一张高端大气上档次的思维导图，总结一下以上各个知识点，如图 4-1 所示。

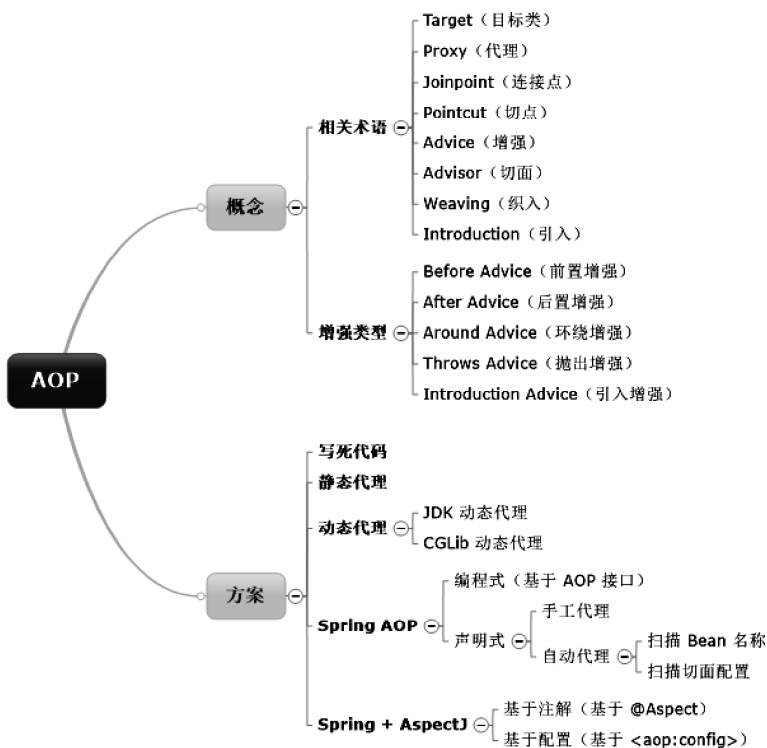


图 4-1 AOP 思维导图

表 4-1 总结了各类增强类型所对应的解决方案。

表 4-1 各类增强类型所对应的解决方案

增强类型	基于 AOP 接口	基于 AOP 注解	基于<aop:config>配置
Before Advice（前置增强）	MethodBeforeAdvice	@Before	<aop:before>
After Advice（后置增强）	AfterReturningAdvice	@After	<aop:after>
Around Advice（环绕增强）	MethodInterceptor	@Around	<aop:around>
Throws Advice（抛出增强）	ThrowsAdvice	@AfterThrowing	<aop:after-throwing>
Introduction Advice（引入增强）	DelegatingIntroductionInterceptor	@DeclareParents	<aop:declare-parents>

最后给出一张 UML 类图描述一下 Spring AOP 的整体架构，如图 4-2 所示。

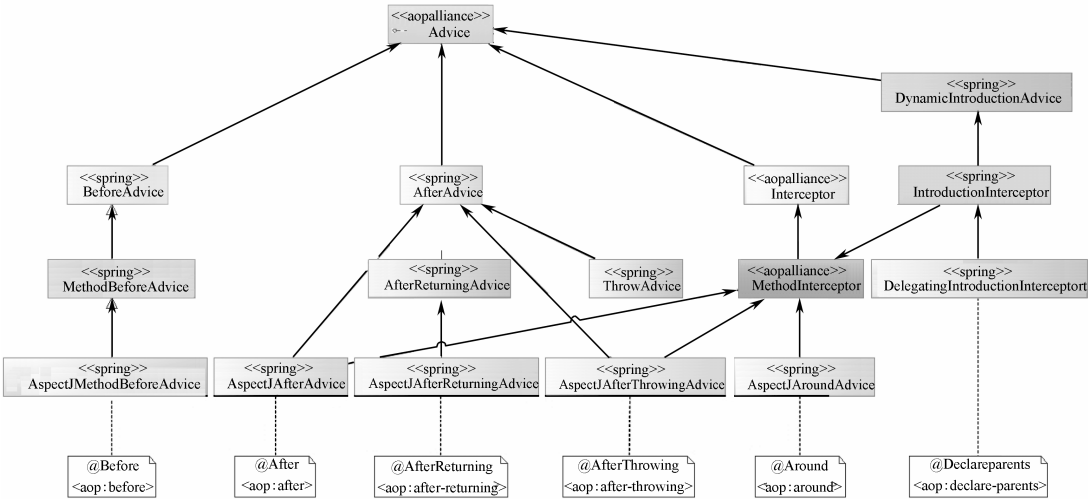


图 4-2 AOP 类图

4.3 开发 AOP 框架

我们打算借鉴 Spring AOP 的风格，写一个基于切面注解的 AOP 框架。在进行下面的步骤之前，请确保您已经学会使用动态代理技术了。

4.3.1 定义切面注解

在框架中添加一个名为 `Aspect` 的注解，代码如下：

```

package org.smart4j.framework.annotation;

import java.lang.annotation.Annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 切面注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Aspect {

    /**
     * 注解
     */
    Class<? extends Annotation> value();
}

```

通过`@Target(ElementType.TYPE)`来设置该注解只能应用在类上。该注解中包含一个名为`value`的属性，它是一个注解类，用来定义`Controller`这类注解。

在使用切面注解之前，我们需要先搭建一个代理框架。

4.3.2 搭建代理框架

继续在框架中添加一个名为`Proxy`的接口，代码如下：

```

package org.smart4j.framework.proxy;

/**
 * 代理接口
 *
 * @author huangyong

```

```
* @since 1.0.0
*/
public interface Proxy {

    /**
     * 执行链式代理
     */
    Object doProxy(ProxyChain proxyChain) throws Throwable;
}
```

这个 `Proxy` 接口中包括了一个 `doProxy` 方法，传入一个 `ProxyChain`，用于执行“链式代理”操作。

所谓链式代理，也就是说，可将多个代理通过一条链子串起来，一个个地去执行，执行顺序取决于添加到链上的先后顺序。

以下是 `ProxyChain` 的相关代码：

```
package org.smart4j.framework.proxy;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
import net.sf.cglib.proxy.MethodProxy;

/**
 * 代理链
 *
 * @author huangyong
 * @since 1.0.0
 */
public class ProxyChain {

    private final Class<?> targetClass;
    private final Object targetObject;
    private final Method targetMethod;
    private final MethodProxy methodProxy;
    private final Object[] methodParams;

    private List<Proxy> proxyList = new ArrayList<Proxy>();
}
```

```
private int proxyIndex = 0;

public ProxyChain(Class<?> targetClass, Object targetObject, Method
targetMethod, MethodProxy methodProxy, Object[] methodParams,
List<Proxy> proxyList) {
    this.targetClass = targetClass;
    this.targetObject = targetObject;
    this.targetMethod = targetMethod;
    this.methodProxy = methodProxy;
    this.methodParams = methodParams;
    this.proxyList = proxyList;
}

public Object[] getMethodParams() {
    return methodParams;
}

public Class<?> getTargetClass() {
    return targetClass;
}

public Method getTargetMethod() {
    return targetMethod;
}

public Object doProxyChain() throws Throwable {
    Object methodResult;
    if (proxyIndex < proxyList.size()) {
        methodResult = proxyList.get(proxyIndex++).doProxy(this);
    } else {
        methodResult = methodProxy.invokeSuper(targetObject, method-
        Params);
    }
    return methodResult;
}
}
```

在 ProxyChain 类中, 我们定义了一系列的成员变量, 包括 targetClass (目标类)、targetObject (目标对象)、targetMethod (目标方法)、methodProxy (方法代理)、methodParams (方法参数),

此外还包括 `proxyList`（代理列表）、`proxyIndex`（代理索引），这些成员变量在构造器中进行初始化，并提供了几个重要的获值方法。

需要注意的是 `MethodProxy` 这个类，它是 CGLib 开源项目为我们提供的一个方法代理对象，在 `doProxyChain` 方法中被使用。

需要稍作解释的是 `doProxyChain` 方法，在该方法中，我们通过 `proxyIndex` 来充当代理对象的计数器，若尚未达到 `proxyList` 的上限，则从 `proxyList` 中取出相应的 `Proxy` 对象，并调用其 `doProxy` 方法。在 `Proxy` 接口的实现中会提供相应的横切逻辑，并调用 `doProxyChain` 方法，随后将再次调用当前 `ProxyChain` 对象的 `doProxyChain` 方法，直到 `proxyIndex` 达到 `proxyList` 的上限为止，最后调用 `methodProxy` 的 `invokeSuper` 方法，执行目标对象的业务逻辑。

我们必须在 `pom.xml` 文件中添加 CGLib 的 Maven 依赖：

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.2.2</version>
</dependency>
```

现在我们需要写一个类，让它提供一个创建代理对象的方法，输入一个目标类和一组 `Proxy` 接口实现，输出一个代理对象，将该类命名为 `ProxyManager`，让它来创建所有的代理对象，代码如下：

```
package org.smart4j.framework.proxy;

import java.lang.reflect.Method;
import java.util.List;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

/**
 * 代理管理器
 *
 * @author huangyong
 * @since 1.0.0
 */
public class ProxyManager {

    @SuppressWarnings("unchecked")
```

```

public static <T> T createProxy(final Class<?> targetClass, final
    List<Proxy> proxyList) {
    return (T) Enhancer.create(targetClass, new MethodInterceptor() {
        @Override
        public Object intercept(Object targetObject, Method targetMethod,
            Object[] methodParams, MethodProxy methodProxy) throws Throwable {
            return new ProxyChain(targetClass, targetObject, targetMethod,
                methodProxy, methodParams, proxyList).doProxyChain();
        }
    });
}
}

```

我们使用 CGLib 提供的 `Enhancer#create` 方法来创建代理对象，将 `intercept` 的参数传入 `ProxyChain` 的构造器中即可。

谁来调用 `ProxyManager` 呢？当然是切面类了，因为在切面类中，需要在目标方法被调用的前后增加相应的逻辑。我们有必要写一个抽象类，让它提供一个模板方法，并在该抽象类的具体实现中扩展相应的抽象方法。我们不妨将该抽象类命名为 `AspectProxy`，代码如下：

```

package org.smart4j.framework.proxy;

import java.lang.reflect.Method;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 切面代理
 *
 * @author huangyong
 * @since 1.0.0
 */
public abstract class AspectProxy implements Proxy {

    private static final Logger logger = LoggerFactory.getLogger
        (AspectProxy.class);

    @Override
    public final Object doProxy(ProxyChain proxyChain) throws Throwable {

```

```
Object result = null;

Class<?> cls = proxyChain.getTargetClass();
Method method = proxyChain.getTargetMethod();
Object[] params = proxyChain.getMethodParams();

begin();
try {
    if (intercept(cls, method, params)) {
        before(cls, method, params);
        result = proxyChain.doProxyChain();
        after(cls, method, params, result);
    } else {
        result = proxyChain.doProxyChain();
    }
} catch (Exception e) {
    logger.error("proxy failure", e);
    error(cls, method, params, e);
    throw e;
} finally {
    end();
}

return result;
}

public void begin() {
}

public boolean intercept(Class<?> cls, Method method, Object[] params)
throws Throwable {
    return true;
}

public void before(Class<?> cls, Method method, Object[] params) throws
Throwable {
}
```

```

    public void after(Class<?> cls, Method method, Object[] params, Object
        result) throws Throwable {
    }

    public void error(Class<?> cls, Method method, Object[] params, Throwable
        e) {
    }

    public void end() {
    }
}

```

需要注意的是 `AspectProxy` 类中的 `doProxy` 方法，我们从 `proxyChain` 参数中获取了目标类、目标方法与方法参数，随后通过一个 `try...catch...finally` 代码块来实现调用框架，从框架中抽象出一系列的“钩子方法”，这些抽象方法可在 `AspectProxy` 的子类中有选择性地实现，就像下面这样：

```

package org.smart4j.chapter4.aspect;

import java.lang.reflect.Method;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.smart4j.framework.annotation.Aspect;
import org.smart4j.framework.annotation.Controller;
import org.smart4j.framework.proxy.AspectProxy;

/**
 * 拦截 Controller 所有方法
 *
 * @author huangyong
 * @since 1.0.0
 */
@Aspect(Controller.class)
public class ControllerAspect extends AspectProxy {

    private static final Logger LOGGER = LoggerFactory.getLogger(Con-
        trollerAspect.class);
}

```

```

        private long begin;

        @Override
        public void before(Class<?> cls, Method method, Object[] params) throws
            Throwable {
            LOGGER.debug("----- begin -----");
            LOGGER.debug(String.format("class: %s", cls.getName()));
            LOGGER.debug(String.format("method: %s", method.getName()));
            begin = System.currentTimeMillis();
        }

        @Override
        public void after(Class<?> cls, Method method, Object[] params, Object
            result) throws Throwable {
            LOGGER.debug(String.format("time: %dms", System.currentTimeMillis()
                - begin));
            LOGGER.debug("----- end -----");
        }
    }
}

```

我们只需实现 **before** 与 **after** 方法,就可以在目标方法执行前后添加其他需要执行的代码了。

那么这样就结束了吗?当然不是。我们需要在整个框架里使用 **ProxyManager** 来创建代理对象,并将该代理对象放入框架底层的 **Bean Map** 中,随后才能通过 **IOC** 将被代理的对象注入到其他对象中。

4.3.3 加载 AOP 框架

按照之前的套路,为了加载 AOP 框架,我们需要编写一个名为 **AopHelper** 的类,然后将其添加到 **HelperLoader** 类中。

在 **AopHelper** 中我们需要获取所有的目标类及其被拦截的切面类实例,并通过 **ProxyManager#createProxy** 方法来创建代理对象,最后将其放入 **Bean Map** 中。

首先,需要为 **BeanHelper** 类添加一个 **setBean** 方法,用于将 **Bean** 实例放入 **Bean Map** 中,代码如下:

```

public final class BeanHelper {
    ...
    /**

```

```

    * 设置 Bean 实例
    */
    public static void setBean(Class<?> cls, Object obj) {
        BEAN_MAP.put(cls, obj);
    }
    ...
}

```

然后，由于我们需要扩展 `AspectProxy` 抽象类的所有具体类，此外，还需要获取带有 `Aspect` 注解的所有类，因此需要在 `ClassHelper` 中添加以下两个方法：

```

public final class ClassHelper {
    ...
    /**
     * 获取应用包名下某父类（或接口）的所有子类（或实现类）
     */
    public static Set<Class<?>> getClassSetBySuper(Class<?> superClass) {
        Set<Class<?>> classSet = new HashSet<Class<?>>();
        for (Class<?> cls : CLASS_SET) {
            if (superClass.isAssignableFrom(cls) && !superClass.equals(cls)) {
                classSet.add(cls);
            }
        }
        return classSet;
    }

    /**
     * 获取应用包名下带有某注解的所有类
     */
    public static Set<Class<?>> getClassSetByAnnotation(Class<? Extends
Annotation> annotationClass) {
        Set<Class<?>> classSet = new HashSet<Class<?>>();
        for (Class<?> cls : CLASS_SET) {
            if (cls.isAnnotationPresent(annotationClass)) {
                classSet.add(cls);
            }
        }
        return classSet;
    }
}

```

```
...
}
```

有了以上两个工具方法以后，我们可以在 `AopHelper` 类中编写一个带有 `Aspect` 注解的所有类，不妨封装一个方法：

```
public final class AopHelper {
    ...
    private static Set<Class<?>> createTargetClassSet (Aspect aspect) throws
    Exception {
        Set<Class<?>> targetClassSet = new HashSet<Class<?>>();
        Class<? extends Annotation> annotation = aspect.value();
        if (annotation != null && !annotation.equals(Aspect.class)) {
            targetClassSet.addAll (ClassHelper.getClassSetByAnnotation
                (annotation));
        }
        return targetClassSet;
    }
    ...
}
```

获取 `Aspect` 注解中设置的注解类，若该注解类不是 `Aspect` 类，则可调用 `ClassHelper#getClassSetByAnnotation` 方法获取相关类，并把这些类放入目标类集合中，最终返回这个集合。

紧接着我们需要获取代理类及其目标类集合之间的映射关系，一个代理类可对应一个或多个目标类。需要强调的是，这里所说的代理类指的是切面类。通过以下代码获取这个映射关系：

```
public final class AopHelper {
    ...
    private static Map<Class<?>, Set<Class<?>>> createProxyMap() throws
    Exception {
        Map<Class<?>, Set<Class<?>>> proxyMap = new HashMap<Class<?>,
            Set<Class<?>>>();
        Set<Class<?>> proxyClassSet = ClassHelper.getClassSetBySuper (AspectProxy.class);
        for (Class<?> proxyClass : proxyClassSet) {
            if (proxyClass.isAnnotationPresent (Aspect.class)) {
                Aspect aspect = proxyClass.getAnnotation (Aspect.class);
                Set<Class<?>> targetClassSet = createTargetClassSet (aspect);
                proxyMap.put (proxyClass, targetClassSet);
            }
        }
    }
}
```

```

        }
    }
    return proxyMap;
}
...
}

```

代理类需要扩展 `AspectProxy` 抽象类，还需要带有 `Aspect` 注解，只有满足这两个条件，才能根据 `Aspect` 注解中所定义的注解属性去获取该注解所对应的目标类集合，然后才能建立代理类与目标类集合之间的映射关系，最终返回这个映射关系。

一旦获取了代理类与目标类集合之间的映射关系，就能根据这个关系分析出目标类与代理对象列表之间的映射关系，就像下面这样：

```

public final class AopHelper {
    ...
    private static Map<Class<?>, List<Proxy>> createTargetMap(Map<Class<?>,
        Set<Class<?>>> proxyMap) throws Exception {
        Map<Class<?>, List<Proxy>> targetMap = new HashMap<Class<?>,
            List<Proxy>>();
        for (Map.Entry<Class<?>, Set<Class<?>>> proxyEntry : proxyMap.
            entrySet()) {
            Class<?> proxyClass = proxyEntry.getKey();
            Set<Class<?>> targetClassSet = proxyEntry.getValue();
            for (Class<?> targetClass : targetClassSet) {
                Proxy proxy = (Proxy) proxyClass.newInstance();
                if (targetMap.containsKey(targetClass)) {
                    targetMap.get(targetClass).add(proxy);
                } else {
                    List<Proxy> proxyList = new ArrayList<Proxy>();
                    proxyList.add(proxy);
                    targetMap.put(targetClass, proxyList);
                }
            }
        }
        return targetMap;
    }
    ...
}

```

最后，在 `AopHelper` 中通过一个静态块来初始化整个 AOP 框架，代码如下：

```
public final class AopHelper {
    ...
    static {
        try {
            Map<Class<?>, Set<Class<?>>> proxyMap = createProxyMap();
            Map<Class<?>, List<Proxy>> targetMap = createTargetMap
                (proxyMap);
            for (Map.Entry<Class<?>, List<Proxy>> targetEntry : targetMap.
                entrySet()) {
                Class<?> targetClass = targetEntry.getKey();
                List<Proxy> proxyList = targetEntry.getValue();
                Object proxy = ProxyManager.createProxy(targetClass,
                    proxyList);
                BeanHelper.setBean(targetClass, proxy);
            }
        } catch (Exception e) {
            LOGGER.error("aop failure", e);
        }
    }
    ...
}
```

获取代理类及其目标类集合的映射关系，进一步获取目标类与代理对象列表的映射关系，进而遍历这个映射关系，从中获取目标类与代理对象列表，调用 `ProxyManager.createProxy` 方法获取代理对象，调用 `BeanHelper.setBean` 方法，将该代理对象重新放入 `Bean Map` 中。

下面是 `AopHelper` 类的所有代码：

```
package org.smart4j.framework.helper;

import java.lang.annotation.Annotation;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.slf4j.Logger;
```

```

import org.slf4j.LoggerFactory;
import org.smart4j.framework.annotation.Aspect;
import org.smart4j.framework.proxy.AspectProxy;
import org.smart4j.framework.proxy.Proxy;
import org.smart4j.framework.proxy.ProxyManager;

/**
 * 方法拦截助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class AopHelper {

    private static final Logger LOGGER = LoggerFactory.getLogger(
        (AopHelper.class);

    static {
        try {
            Map<Class<?>, Set<Class<?>>> proxyMap = createProxyMap();
            Map<Class<?>, List<Proxy>> targetMap = createTargetMap(
                proxyMap);
            for (Map.Entry<Class<?>, List<Proxy>> targetEntry : targetMap.
                entrySet()) {
                Class<?> targetClass = targetEntry.getKey();
                List<Proxy> proxyList = targetEntry.getValue();
                Object proxy = ProxyManager.createProxy(targetClass, proxy-
                    List);
                BeanHelper.setBean(targetClass, proxy);
            }
        } catch (Exception e) {
            LOGGER.error("aop failure", e);
        }
    }

    private static Map<Class<?>, Set<Class<?>>> createProxyMap() throws
        Exception {
        Map<Class<?>, Set<Class<?>>> proxyMap = new HashMap<Class<?>,
            Set<Class<?>>>();

```

```

        Set<Class<?>> proxyClassSet = ClassHelper.getClassSetBySuper
            (AspectProxy.class);
        for (Class<?> proxyClass : proxyClassSet) {
            if (proxyClass.isAnnotationPresent(Aspect.class)) {
                Aspect aspect = proxyClass.getAnnotation(Aspect.class);
                Set<Class<?>> targetClassSet = createTargetClassSet(aspect);
                proxyMap.put(proxyClass, targetClassSet);
            }
        }
        return proxyMap;
    }

    private static Set<Class<?>> createTargetClassSet(Aspect aspect) throws
        Exception {
        Set<Class<?>> targetClassSet = new HashSet<Class<?>>();
        Class<? extends Annotation> annotation = aspect.value();
        if (annotation != null && !annotation.equals(Aspect.class)) {
            targetClassSet.addAll(ClassHelper.getClassSetByAnnotation
                (annotation));
        }
        return targetClassSet;
    }

    private static Map<Class<?>, List<Proxy>> createTargetMap(Map<Class<?>,
        Set<Class<?>>> proxyMap) throws Exception {
        Map<Class<?>, List<Proxy>> targetMap = new HashMap<Class<?>, List
            <Proxy>>();
        for (Map.Entry<Class<?>, Set<Class<?>>> proxyEntry : proxyMap.
            entrySet()) {
            Class<?> proxyClass = proxyEntry.getKey();
            Set<Class<?>> targetClassSet = proxyEntry.getValue();
            for (Class<?> targetClass : targetClassSet) {
                Proxy proxy = (Proxy) proxyClass.newInstance();
                if (targetMap.containsKey(targetClass)) {
                    targetMap.get(targetClass).add(proxy);
                } else {
                    List<Proxy> proxyList = new ArrayList<Proxy>();
                    proxyList.add(proxy);

```

```

        targetMap.put(targetClass, proxyList);
    }
}
}
return targetMap;
}
}

```

别忘了将 `AopHelper` 添加到 `HelperLoader` 中进行初始化，代码如下：

```

package org.smart4j.framework;

import org.smart4j.framework.helper.AopHelper;
import org.smart4j.framework.helper.BeanHelper;
import org.smart4j.framework.helper.ClassHelper;
import org.smart4j.framework.helper.ControllerHelper;
import org.smart4j.framework.helper.IocHelper;
import org.smart4j.framework.util.ClassUtil;

/**
 * 加载相应的 Helper 类
 */
public final class HelperLoader {

    public static void init() {
        Class<?>[] classList = {
            ClassHelper.class,
            BeanHelper.class,
            AopHelper.class,
            IocHelper.class,
            ControllerHelper.class
        };
        for (Class<?> cls : classList) {
            ClassUtil.loadClass(cls.getName(), true);
        }
    }
}

```

需要注意的是，`AopHelper` 要在 `IocHelper` 之前加载，因为首先需要通过 `AopHelper` 获取代

理对象，然后才能通过 `IocHelper` 进行依赖注入。

至此，一个简单的 AOP 就算开发完毕了。

4.4 ThreadLocal 简介

4.4.1 什么是 ThreadLocal

`ThreadLocal` 直译为“线程本地”或“本地线程”，如果真的这么认为，那就错了！其实它就是一个容器，用于存放线程的局部变量，应该叫 `ThreadLocalVariable`（线程局部变量）才对，很不理解为什么当初 Sun 公司的工程师这样命名。

早在 JDK 1.2 的时代，`java.lang.ThreadLocal` 就诞生了，它是为了解决多线程并发问题而设计的，只不过设计得有些难用而已，所以至今没有得到广泛使用。

一个序号生成器的程序可能同时会有多个线程并发访问它，要保证每个线程得到的序号都是自增的，而不能相互干扰。

先定义一个接口：

```
public interface Sequence {  
  
    int getNumber();  
}
```

每次调用 `getNumber` 方法可获取一个序号，下次再调用时，序号会自增。

再做一个线程类：

```
public class ClientThread extends Thread {  
  
    private Sequence sequence;  
  
    public ClientThread(Sequence sequence) {  
        this.sequence = sequence;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 3; i++) {  
            System.out.println(Thread.currentThread().getName() + " => " +  
                sequence.getNumber());  
        }  
    }  
}
```

```
    }  
    }  
}
```

在线程中连续输出三次线程名与其对应的序列号。

我们不用 `ThreadLocal`，先做一个实现类：

```
public class SequenceA implements Sequence {  
  
    private static int number = 0;  
  
    public int getNumber() {  
        number = number + 1;  
        return number;  
    }  
  
    public static void main(String[] args) {  
        Sequence sequence = new SequenceA();  
  
        ClientThread thread1 = new ClientThread(sequence);  
        ClientThread thread2 = new ClientThread(sequence);  
        ClientThread thread3 = new ClientThread(sequence);  
  
        thread1.start();  
        thread2.start();  
        thread3.start();  
    }  
}
```

序列号初始值是 0，在 `main` 方法中模拟了三个线程，运行后结果如下：

```
Thread-0 => 1  
Thread-0 => 2  
Thread-0 => 3  
Thread-2 => 4  
Thread-2 => 5  
Thread-2 => 6  
Thread-1 => 7  
Thread-1 => 8  
Thread-1 => 9
```

由于线程启动顺序是随机的,所以并不是 0、1、2 这样的顺序,这个好理解。为什么当 Thread-0 输出了 1、2、3 之后,而 Thread-2 却输出了 4、5、6 呢?不应该也从 0 开始输出吗?

仔细分析才发现,线程之间共享的 `static` 变量无法保证对于不同线程而言是安全的,也就是说,此时无法保证“线程安全”。

那么如何才能做到“线程安全”呢?对应于这个案例,就是说不同的线程可拥有自己的 `static` 变量,如何实现呢?下面看看另外一个实现:

```
public class SequenceB implements Sequence {

    private static ThreadLocal<Integer> numberContainer = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    public int getNumber() {
        numberContainer.set(numberContainer.get() + 1);
        return numberContainer.get();
    }

    public static void main(String[] args) {
        Sequence sequence = new SequenceB();

        ClientThread thread1 = new ClientThread(sequence);
        ClientThread thread2 = new ClientThread(sequence);
        ClientThread thread3 = new ClientThread(sequence);

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

通过 `ThreadLocal` 封装了一个 `Integer` 类型的 `numberContainer` 静态成员变量,并且初始值是 0。再看 `getNumber` 方法,首先从 `numberContainer` 中 `get` 出当前的值,加 1,随后 `set` 到 `numberContainer` 中,最后在 `numberContainer` 中 `get` 出当前的值并返回。

是不是很绕？但是很强大！我们不妨把 `ThreadLocal` 看作一个容器，这样理解起来就简单了。所以，这里故意用 `Container` 这个单词作为后缀来命名 `ThreadLocal` 变量。

运行结果如下：

```
Thread-0 => 1
Thread-0 => 2
Thread-0 => 3
Thread-2 => 1
Thread-2 => 2
Thread-2 => 3
Thread-1 => 1
Thread-1 => 2
Thread-1 => 3
```

每个线程相互独立了，同样是 `static` 变量，对于不同的线程而言，它没有被共享，而是每个线程各一份，这样也就保证了线程安全。也就是说，`ThreadLocal` 为每一个线程提供了一个独立的副本。

搞清楚 `ThreadLocal` 的原理之后，有必要总结一下 `ThreadLocal` 的 API，其实很简单。

- `public void set(T value)`：将值放入线程局部变量中；
- `public T get()`：从线程局部变量中获取值；
- `public void remove()`：从线程局部变量中移除值（有助于 JVM 垃圾回收）；
- `protected T initialValue()`：返回线程局部变量中的初始值（默认为 `null`）。

为什么 `initialValue` 方法是 `protected` 的呢？就是为了提醒程序员，这个方法是要程序员来实现的，要给这个线程局部变量设置一个初始值。

4.4.2 自己实现 ThreadLocal

熟悉了原理与这些 API 之后，其实可以想想 `ThreadLocal` 里面不就是封装了一个 `Map` 吗？自己都可以写一个 `ThreadLocal` 了：

```
public class MyThreadLocal<T> {

    private Map<Thread, T> container = Collections.synchronizedMap(new
        HashMap<Thread, T>());

    public void set(T value) {
```

```

        container.put(Thread.currentThread(), value);
    }

    public T get() {
        Thread thread = Thread.currentThread();
        T value = container.get(thread);
        if (value == null && !container.containsKey(thread)) {
            value = initialValue();
            container.put(thread, value);
        }
        return value;
    }

    public void remove() {
        container.remove(Thread.currentThread());
    }

    protected T initialValue() {
        return null;
    }
}

```

以上完全“山寨”了一个 `ThreadLocal`，其中定义了一个同步 `Map`（为什么要这样？请自行思考），代码应该非常容易读懂。

下面用 `MyThreadLocal` 再来实现一次：

```

public class SequenceC implements Sequence {

    private static MyThreadLocal<Integer> numberContainer = new MyThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    public int getNumber() {
        numberContainer.set(numberContainer.get() + 1);
    }
}

```

```

        return numberContainer.get();
    }

    public static void main(String[] args) {
        Sequence sequence = new SequenceC();

        ClientThread thread1 = new ClientThread(sequence);
        ClientThread thread2 = new ClientThread(sequence);
        ClientThread thread3 = new ClientThread(sequence);

        thread1.start();
        thread2.start();
        thread3.start();
    }
}

```

以上代码其实就是将 `ThreadLocal` 替换成了 `MyThreadLocal`，仅此而已，运行效果和之前的一样，也是正确的。

其实 `ThreadLocal` 可以单独成为一种设计模式，就看程序员怎么理解了。

提示： 当在一个类中使用了 `static` 成员变量的时候，一定要多问问自己，这个 `static` 成员变量需要考虑“线程安全”吗？也就是说，多个线程需要独享自己的 `static` 成员变量吗？如果需要考虑，不妨请用 `ThreadLocal`。

4.4.3 ThreadLocal 使用案例

`ThreadLocal` 具体有哪些使用案例呢？

首先要说的就是通过 `ThreadLocal` 存放 JDBC Connection，以达到事务控制的能力。

记得在很久以前，用户提出过一个需求，需求很烦琐，就一句话：

当修改产品价格的时候，需要记录操作日志，什么时候做了什么事情。

想必这个案例，只要是做过应用系统的小伙伴都应该遇到过。不外乎数据库里就两张表：`product` 与 `log`，用两条 SQL 语句应该可以解决问题：

```

update product set price = ? where id = ?
insert into log (created, description) values (?, ?)

```

但要确保这两条 SQL 语句必须在同一个事务里进行提交，否则有可能 `update` 提交了，但 `insert` 却没有提交。如果这样的事情真的发生了，我们肯定会被用户指着鼻子狂骂：“为什么产品价格改了，却看不到什么时候改的呢？”。

当初我在接到这个需求以后的做法如下所述。

首先，写一个 `DBUtil` 的工具类，封装数据库的常用操作：

```
public class DBUtil {

    // 数据库配置
    private static final String driver = "com.mysql.jdbc.Driver";
    private static final String url = "jdbc:mysql://localhost:3306/demo";
    private static final String username = "root";
    private static final String password = "root";

    // 定义一个数据库连接
    private static Connection conn = null;

    // 获取连接
    public static Connection getConnection() {
        try {
            Class.forName(driver);
            conn = DriverManager.getConnection(url, username, password);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conn;
    }

    // 关闭连接
    public static void closeConnection() {
        try {
            if (conn != null) {
                conn.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

里面设置了一个 `static` 的 `Connection`，这下数据库连接就好操作了。

然后，定义一个接口，用于给逻辑层调用：

```
public interface ProductService {  
  
    void updateProductPrice(long productId, int price);  
}
```

根据用户提出的需求，我认为这个接口完全够用了。根据 `productId` 去更新对应 `Product` 的 `price`，然后再插入一条数据到 `log` 表中。

其实业务逻辑也不太复杂，于是快速地完成了 `ProductService` 接口的实现类：

```
public class ProductServiceImpl implements ProductService {  
  
    private static final String UPDATE_PRODUCT_SQL = "update product set price  
    = ? where id = ?";  
    private static final String INSERT_LOG_SQL = "insert into log (created,  
    description) values (?, ?)";  
  
    public void updateProductPrice(long productId, int price) {  
        try {  
            // 获取连接  
            Connection conn = DBUtil.getConnection();  
            conn.setAutoCommit(false);           // 关闭自动提交事务（开启事务）  
  
            // 执行操作  
            updateProduct(conn, UPDATE_PRODUCT_SQL, productId, price);  
                                                    // 更新产品  
            insertLog(conn, INSERT_LOG_SQL, "Create product."); // 插入日志  
  
            // 提交事务  
            conn.commit();  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            // 关闭连接  
            DBUtil.closeConnection();  
        }  
    }  
}
```

```
private void updateProduct(Connection conn, String updateProductSQL,
long productId, int productPrice) throws Exception {
    PreparedStatement pstmt = conn.prepareStatement(updateProductSQL);
    pstmt.setInt(1, productPrice);
    pstmt.setLong(2, productId);
    int rows = pstmt.executeUpdate();
    if (rows != 0) {
        System.out.println("Update product success!");
    }
}

private void insertLog(Connection conn, String insertLogSQL, String
logDescription) throws Exception {
    PreparedStatement pstmt = conn.prepareStatement(insertLogSQL);
    pstmt.setString(1, new SimpleDateFormat("yyyy-MM-dd HH:mm:ss
SSS").format(new Date()));
    pstmt.setString(2, logDescription);
    int rows = pstmt.executeUpdate();
    if (rows != 0) {
        System.out.println("Insert log success!");
    }
}
}
```

这里用到了 JDBC 的高级特性 **Transaction**。暗自庆幸了一番之后，我想是不是有必要写一个客户端来测试一下执行结果是不是我想要的呢？于是偷懒，直接在 **ProductService-Impl** 中增加了一个 **main** 方法：

```
public static void main(String[] args) {
    ProductService productService = new ProductServiceImpl();
    productService.updateProductPrice(1, 3000);
}
```

我想让 **productId** 为 1 的产品的价格修改为 3000。把程序跑了一遍，控制台输出如下：

```
Update product success!
Insert log success!
```

作为一名专业的程序员，为了万无一失，我一定要到数据库里再看看。没错！**product** 表对

应的记录更新了，log 表也插入了一条记录。这样就可以将 `ProductService` 接口交付给别人来调用了。

几个小时过去了，QA 妹妹开始对着我嚷：“那谁！我才模拟了 10 个请求，你这个接口怎么就挂了？报错说是数据库连接关闭了！”。

听到这样的叫声，我浑身打颤，立马关闭了小视频，赶紧打开 IDE，找到了这个 `ProductServiceImpl` 实现类。心里默念“好像没有 Bug 吧？”，但现在不敢给她任何回应，我确实有点怕她的叫声。

我突然想起，她是用工具模拟的，也就是模拟多个线程了！那我也可以模拟，于是写了一个线程类：

```
public class ClientThread extends Thread {

    private ProductService productService;

    public ClientThread(ProductService productService) {
        this.productService = productService;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
        productService.updateProductPrice(1, 3000);
    }

}
```

用这线程去调用 `ProductService` 的方法，看看是不是有问题。此时，还要再修改一下 main 方法：

```
// public static void main(String[] args) {
//     ProductService productService = new ProductServiceImpl();
//     productService.updateProductPrice(1, 3000);
// }

public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        ProductService productService = new ProductServiceImpl();
        ClientThread thread = new ClientThread(productService);
    }
}
```

```
        thread.start();  
    }  
}
```

同样我也模拟 10 个线程，运行结果如下：

```
Thread-1  
Thread-3  
Thread-5  
Thread-7  
Thread-9  
Thread-0  
Thread-2  
Thread-4  
Thread-6  
Thread-8  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
Update product success!  
Insert log success!  
com.mysql.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException: No  
operations allowed after connection closed.  
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
```

```

at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
at com.mysql.jdbc.Util.handleNewInstance(Util.java:411)
at com.mysql.jdbc.Util.getInstance(Util.java:386)
at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:1015)
at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:989)
at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:975)
at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:920)
at com.mysql.jdbc.ConnectionImpl.throwConnectionClosedException(ConnectionImpl.java:1304)
at com.mysql.jdbc.ConnectionImpl.checkClosed(ConnectionImpl.java:1296)
at com.mysql.jdbc.ConnectionImpl.commit(ConnectionImpl.java:1699)
at org.smart4j.sample.test.transaction.solution1.ProductServiceImpl.updateProductPrice(ProductServiceImpl.java:25)
at org.smart4j.sample.test.transaction.ClientThread.run(ClientThread.java:18)

```

没想到啊！竟然在多线程的环境下报错了，果然是数据库连接关闭了。怎么回事呢？我陷入了沉思中。在百度、Google，还有 OSC 上都查找了那句报错信息，解答实在是千奇百怪。

既然是跟 `Connection` 有关系，那就将主要精力放在检查 `Connection` 相关的代码上。是不是 `Connection` 不应该是 `static` 的呢？当初设计成 `static` 的主要目的是为了让 `DBUtil` 的 `static` 方法访问起来更加方便，用 `static` 变量来存放 `Connection` 也提高了性能。怎么办呢？

后来看到了 OSC 上非常火爆的一篇文章“`ThreadLocal` 那点事儿”，才终于明白了，原来要使每个线程都拥有自己的连接，而不是共享同一个连接，否则“线程 1”有可能会关闭“线程 2”的连接，所以“线程 2”就报错了。一定是这样！

于是赶紧将 `DBUtil` 重构了：

```

public class DBUtil {
    // 数据库配置
    private static final String driver = "com.mysql.jdbc.Driver";
    private static final String url = "jdbc:mysql://localhost:3306/demo";
    private static final String username = "root";
    private static final String password = "root";
}

```

```
// 定义一个用于放置数据库连接的局部线程变量（使每个线程都拥有自己的连接）
private static ThreadLocal<Connection> connContainer = new ThreadLocal<Connection>();

// 获取连接
public static Connection getConnection() {
    Connection conn = connContainer.get();
    try {
        if (conn == null) {
            Class.forName(driver);
            conn = DriverManager.getConnection(url, username, password);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        connContainer.set(conn);
    }
    return conn;
}

// 关闭连接
public static void closeConnection() {
    Connection conn = connContainer.get();
    try {
        if (conn != null) {
            conn.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        connContainer.remove();
    }
}
}
```

把 `Connection` 放到了 `ThreadLocal` 中，这样每个线程之间就隔离了，不会相互干扰了。

此外，在 `getConnection` 方法中，首先从 `ThreadLocal`（也就是 `connContainer`）中获取

Connection，如果没有，就通过 JDBC 来创建连接，最后再把创建好的连接放入这个 ThreadLocal 中。可以把 ThreadLocal 看作一个容器，一点不假。

同样也对 closeConnection 方法做了重构，先从容器中获取 Connection，拿到了就 close 掉，最后从容器中将其 remove 掉，以保持容器的清洁。

再次运行 main 方法：

```
Thread-0
Thread-2
Thread-4
Thread-6
Thread-8
Thread-1
Thread-3
Thread-5
Thread-7
Thread-9
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
Update product success!
Insert log success!
```

现在总算松了一口气！问题解决了，心里想着：“QA 妹妹，当她看到我修改了这个 Bug

后，应该会对我微笑吧？”。

注意：该示例仅用于说明 `TheadLocal` 的基本用法。在实际工作中，推荐使用连接池来管理数据库连接。示例中的代码仅作参考，使用前请酌情考虑。

4.5 事务管理简介

4.5.1 什么是事务

事务（Transaction）通俗的理解就是一件事情。从小父母就教育我们，做事情要有始有终，不能半途而废。事务也是这样，要么做完，要么不做，不要做一半留一半。也就是说，事务必须是一个不可分割的整体，就像我们在化学课里学到的原子，原子是构成物质的最小单位。于是，人们就归纳出事务的第一个特性：原子性（Atomicity）。

特别是在数据库领域，事务是一个非常重要的概念，除了原子性以外，它还有一个极其重要的特性，那就是一致性（Consistency）。也就是说，执行完数据库操作后，数据不会被破坏。打个比方，如果从 A 账户转账到 B 账户，不可能因为 A 账户扣了钱，而 B 账户没有加钱。如果出现了这类事情，用户一定会非常气愤。

当我们编写了一条 `update` 语句，提交到数据库的一刹那，有可能别人也提交了一条 `delete` 语句到数据库中。也许我们都是对同一条记录进行操作，可以想象，如果不稍加控制，就会出大麻烦。我们必须保证数据库操作之间是“隔离”的（线程之间有时也要做到隔离），彼此之间没有任何干扰，这就是隔离性（Isolation）。要想真正做到操作之间完全没有任何干扰是很难的，于是，数据库权威专家开始动脑筋了，“我们要制定一个规范，让各个数据库厂商都支持我们的规范！”，这个规范就是事务隔离级别（Transaction Isolation Level）。能定义出这么牛的规范真的挺不容易的，其实说白了就四个级别：

- `READ_UNCOMMITTED`;
- `READ_COMMITTED`;
- `REPEATABLE_READ`;
- `SERIALIZABLE`。

从上往下，级别越来越高，并发性越来越差，安全性越来越高。

当我们执行一条 `insert` 语句后，数据库必须要保证有一条数据永久地存放在磁盘中，这也算事务的一条特性，它就是持久性（Durability）。

归纳一下，以上一共提到了事务的 4 条特性，把它们的英文单词首字母合起来就是 ACID，这就是传说中的“事务 ACID 特性”。

真的是非常牛的特性。这 4 条特性是事务管理的基石，一定要透彻理解。此外还要明确，这 4 个家伙当中，谁才是“老大”？

其实想想也就清楚了：原子性是基础，隔离性是手段，持久性是目的，真正的“老大”就是一致性。所以说，这三个“小弟”都是跟着“一致性”这个“老大”混的，为它全心全意地服务。

4.5.2 事务所面临的问题

ACID 这 4 条特征当中，其实最难理解的倒不是一致性，而是隔离性。因为它是保证一致性的重要手段，它是工具，使用它不能有半点差池。怪不得数据库权威专家都来研究所谓的事务隔离级别。其实，定义这 4 个级别就是为了解决数据在高并发下产生的问题：

- Dirty Read（脏读）；
- Unrepeatable Read（不可重复读）；
- Phantom Read（幻读）。

首先看看“脏读”，我第一次看到“脏”这个字时，就想到了恶心、肮脏。数据怎么可能脏呢？其实脏数据也就是我们经常说的“垃圾数据”了。比如说，有两个事务，它们在并发执行（也就是竞争），如表 4-2 所示。

表 4-2 并发执行的两个事务

时 间	事务 A（存款）	事务 B（取款）
T1	开始事务	—
T2	—	开始事务
T3	—	查询余额（1000 元）
T4	—	取出 1000 元（余额 0 元）
T5	查询余额（0 元）	—
T6	—	撤销事务（余额恢复为 1000 元）
T7	存入 500 元（余额 500 元）	—
T8	提交事务	—

余额应该为 1500 元才对。请看 T5 时间点，事务 A 此时查询余额为 0 元，这个数据就是脏数据，它是事务 B 造成的，明显事务没有进行隔离，渗过来了，乱套了。

所以脏读这件事情是非常要不得的，一定要解决！让事务之间隔离起来才是硬道理。

那第 2 条不可重复读又怎么解释呢？还是用类似的例子来说明，如表 4-3 所示。

表 4-3 不可重复读示例

时 间	事务 A（存款）	事务 B（取款）
T1	开始事务	—
T2	—	开始事务
T3	—	查询余额（1000 元）
T4	查询余额（1000 元）	—
T5	—	取出 1000 元（余额 0 元）
T6	—	提交事务
T7	查询余额（0 元）	—

事务 A 其实除了查询了两次以外，其他什么事情都没有做，结果钱就从 1000 变成 0 了，这就是重复读了。其实这样也是合理的，毕竟事务 B 提交了事务，数据库将结果进行了持久化，所以事务 A 再次读取时自然就发生了变化。

这种现象基本上是可以理解的，但在有些“变态”的场景下却是不允许的。毕竟这种现象也是事务之间没有隔离所造成的，但我们对于这种问题似乎可以忽略。

最后一条是幻读。听起来很奇幻。Phantom 这个单词不就是“幽灵、鬼魂”吗？其意义就是鬼在读，不是人在读，或者说搞不清楚为什么，它就变了。下面举个例子，如表 4-4 所示。

表 4-4 幻读示例

时 间	事务 A（统计总存款）	事务 B（存款）
T1	开始事务	—
T2	—	开始事务
T3	统计总存款（10000 元）	—
T4	—	存入 100 元
T5	—	提交事务
T6	统计总存款（10100 元）	—

银行工作人员每次统计总存款时都看到不一样的结果。不过这确实也挺正常，总存款增多了，肯定是这个时候有人在存钱。但是如果银行系统真的这样设计，那算是完了。这种情况同样也是由于事务没有隔离造成的，但对于大多数应用系统而言，这似乎也是正常的。银行里的系统要求非常严密，统计的时候，甚至会将所有的其他操作给隔离开，这种隔离级别就算非常高了（估计要到 `SERIALIZABLE` 级别了）。

归纳一下，以上提到了事务并发所引起的与读取数据有关的问题，各用一句话来描述：

- 脏读——事务 A 读取了事务 B 未提交的数据，并在这个基础上又做了其他操作。

- 不可重复读——事务 A 读取了事务 B 已提交的更改数据。
- 幻读——事务 A 读取了事务 B 已提交的新增数据。

第一条是坚决抵制的，后两条在大多数情况下可不考虑。

这就是为什么必须要有事务隔离级别这个东西了，它就像一面墙一样，隔离不同的事务。不同的事务隔离级别能处理的事务并发问题如表 4-5 所示。

表 4-5 事务隔离级别

事务隔离级别	脏 读	不可重复读	幻 读
READ_UNCOMMITTED	允许	允许	允许
READ_COMMITTED	禁止	允许	允许
REPEATABLE_READ	禁止	禁止	允许
SERIALIZABLE	禁止	禁止	禁止

根据用户的实际需求参考这张表，然后确定事务隔离级别，应该不再是一件难事了。

JDBC 也提供了这四类事务隔离级别，但默认事务隔离级别对不同数据库产品而言却是不一样的。我们熟知的 MySQL 数据库的默认事务隔离级别就是 READ_COMMITTED，Oracle、SQL Server、DB2 等都有自己的默认值。READ_COMMITTED 已经可以解决绝大多数问题了，其他的就具体情况具体分析了。

若对其他数据库的默认事务隔离级别不太清楚，可以使用以下代码来获取：

```
DatabaseMetaData meta = DBUtil.getDataSource().getConnection().
getMetaData();
int defaultIsolation = meta.getDefaultTransactionIsolation();
```

提示：在 java.sql.Connection 类中可查看所有隔离级别。

我们知道 JDBC 只是连接 Java 程序与数据库的桥梁而已，那么数据库又是怎样隔离事务的呢？其实它就是“锁”这个东西。当插入数据时，就锁定表，这叫“锁表”；当更新数据时，就锁定行，这叫“锁行”。当然这个已经超出了我们现在所讨论的范围了。

4.5.3 Spring 的事务传播行为

除了 JDBC 给我们提供的事务隔离级别这种解决方案以外，还有哪些解决方案可以完善事务管理功能呢？

不妨看看 Spring 的解决方案，其实它是对 JDBC 的补充或扩展。它提供了一个非常重要的

功能——事务传播行为（Transaction Propagation Behavior）。

Spring 一共提供了 7 种事务传播行为，这 7 种行为一出现，真是震撼了整个码农界！

- PROPAGATION_REQUIRED;
- PROPAGATION_REQUIRES_NEW;
- PROPAGATION_NESTED;
- PROPAGATION_SUPPORTS;
- PROPAGATION_NOT_SUPPORTED;
- PROPAGATION_NEVER;
- PROPAGATION_MANDATORY。

我们可以这样理解，首先要明确事务从哪里来传播到哪里去？答案是从方法 A 传播到方法 B。Spring 解决的只是方法之间的事务传播，比如：

- 方法 A 有事务，方法 B 也有事务；
- 方法 A 有事务，方法 B 没有事务；
- 方法 A 没有事务，方法 B 有事务；
- 方法 A 没有事务，方法 B 也没有事务。

这样就是 4 种了，还有 3 种特殊情况。下面做一个分析。

假设事务从方法 A 传播到方法 B，用户需要面对方法 B，问自己一个问题：方法 A 有事务吗？

（1）如果没有，就新建一个事务；如果有，就加入当前事务。这就是 PROPAGATION_REQUIRED，它也是 Spring 提供的默认事务传播行为，适合绝大多数情况。

（2）如果没有，就新建一个事务；如果有，就将当前事务挂起。这就是 PROPAGATION_REQUIRES_NEW，意思就是创建了一个新事务，它和原来的事务没有任何关系了。

（3）如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。这就是 PROPAGATION_NESTED，也就是“嵌套事务”，所嵌套的子事务与主事务之间是有关联的（当主事务提交或回滚，子事务也会提交或回滚）。

（4）如果没有，就以非事务方式执行；如果有，就使用当前事务。这就是 PROPAGATION_SUPPORTS，这种方式非常随意，没有就没有，有就有，有点无所谓的态度，反正是支持的。

（5）如果没有，就以非事务方式执行；如果有，就将当前事务挂起。这就是 PROPAGATION_NOT_SUPPORTED，这种方式非常强硬，没有就没有，有也不支持，挂起来，不管它。

(6) 如果没有,就以非事务方式执行;如果有,就抛出异常。这就是 `PROPAGATION_NEVER`,这种方式更强硬,没有就没有,有了反而报错,它对大家宣称:我从不支持事务。

(7) 如果没有,就抛出异常;如果有,就使用当前事务。这就是 `PROPAGATION_MANDATORY`,这种方式可以说是最强硬的,没有事务直接就报错,它对全世界说:我必须要有事务。

看到上面这段解释,您是否已经感受到,自己的“任督二脉”有种被打通的感觉呢?如果还没通,就多读几遍吧,体会一下,就是读者自己的东西了,理解后,再去和身边的小伙伴分享吧。

需要注意的是 `PROPAGATION_NESTED`,不要被它的名字所欺骗——`Nested` (嵌套)。凡是在类似方法 A 调用方法 B 的时候,在方法 B 上使用了这种事务传播行为,都是错的。因为这是错误地以为 `PROPAGATION_NESTED` 就是为方法嵌套调用而准备的,其实默认的 `PROPAGATION_REQUIRED` 就可以做我们想要做的事情。

Spring 给我们带来了事务传播行为,这确实是一个非常强大而又实用的功能。除此以外,它也提供了一些小的附加功能,比如:

- 事务超时 (Transaction Timeout) ——为了解决事务时间太长,消耗太多资源的问题,所以故意给事务设置一个最大时长,如果超过了,就回滚事务。
- 只读事务 (Readonly Transaction) ——为了忽略那些不需要事务的方法,比如读取数据,这样可以有效地提高一些性能。

推荐使用 Spring 的注解式事务配置,而放弃 XML 式事务配置。因为注解实在是太优雅了,当然这一切都取决于程序员自身的情况。

在 Spring 配置文件中使用:

```
...
<tx:annotation-driven />
...
```

在需要事务的方法上使用:

```
@Transactional
public void xxx() {
    ...
}
```

可在 `Transactional` 注解中设置事务隔离级别、事务传播行为、事务超时时间、是否只读事务。

最后，有必要对本节的内容做一个总结，事务管理的思维导图如图 4-3 所示。

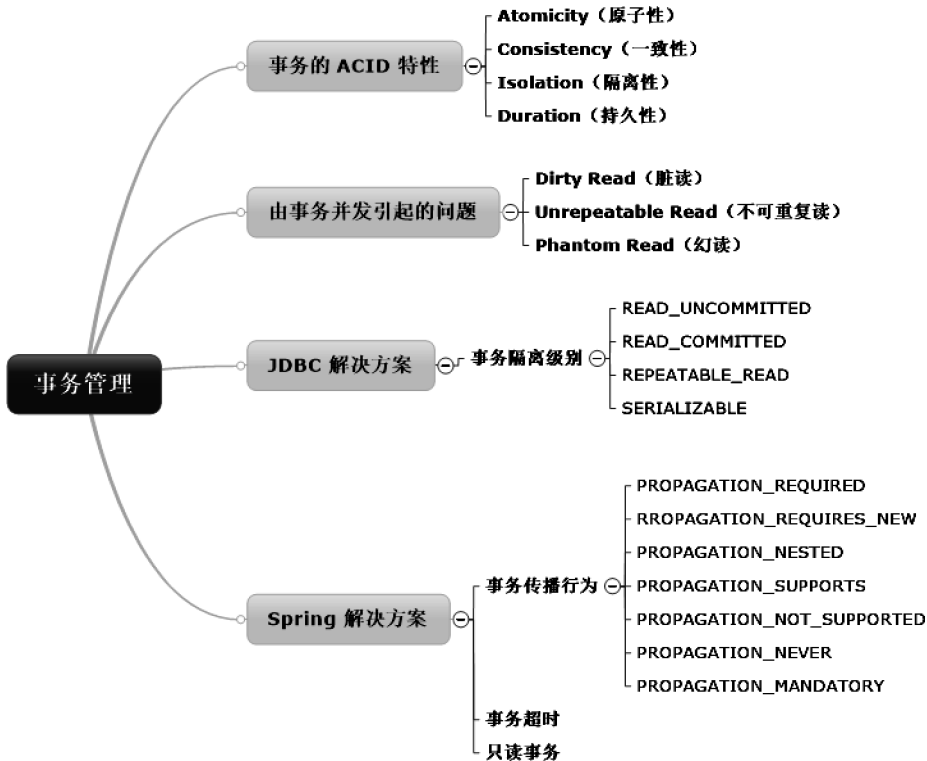


图 4-3 事务管理

4.6 实现事务控制特性

我们之前实现过一个 Service 注解，用于定义服务类，而在服务类中会包括若干方法，有些方法是具备事务性的，比如创建、修改、删除等。如何来保证这类方法具有事务性呢？我们可以利用这个 Proxy 框架来实现一个简单的事务控制特性。只需要开发者使用 Transaction 注解，将其定义在需要事务控制的方法上即可。

下面我们就来实现这个事务管理框架。

4.6.1 定义事务注解

```
package org.smart4j.framework.annotation;
```

```

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

/**
 * 定义需要事务控制的方法
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Transaction {
}

```

在 `Transaction` 注解中使用了 `@Target(ElementType.METHOD)`，说明该注解只能用于方法级别。也就是说，我们需要将该注解应用在每个具有事务性的方法上，比如：

```

package org.smart4j.chapter4.service;

import java.util.List;
import java.util.Map;
import org.smart4j.chapter4.model.Customer;
import org.smart4j.framework.annotation.Service;
import org.smart4j.framework.helper.DatabaseHelper;

/**
 * 提供客户数据服务
 */
@Service
public class CustomerService {

    /**
     * 获取客户列表
     */
    public List<Customer> getCustomerList() {
        String sql = "SELECT * FROM customer";
        return DatabaseHelper.queryEntityList(Customer.class, sql);
    }
}

```

```
    }

    /**
     * 获取客户
     */
    public Customer getCustomer(long id) {
        String sql = "SELECT * FROM customer WHERE id = ?";
        return DatabaseHelper.queryEntity(Customer.class, sql, id);
    }

    /**
     * 创建客户
     */
    @Transaction
    public boolean createCustomer(Map<String, Object> fieldMap) {
        return DatabaseHelper.insertEntity(Customer.class, fieldMap);
    }

    /**
     * 更新客户
     */
    @Transaction
    public boolean updateCustomer(long id, Map<String, Object> fieldMap) {
        return DatabaseHelper.updateEntity(Customer.class, id, fieldMap);
    }

    /**
     * 删除客户
     */
    @Transaction
    public boolean deleteCustomer(long id) {
        return DatabaseHelper.deleteEntity(Customer.class, id);
    }
}
```

以上 `createCustomer`、`updateCustomer`、`deleteCustomer` 方法都带有 `Transaction` 注解，表明它们都是具备事务性的。可以认为，凡是对数据库有变更的方法，都建议带上 `Transaction` 注解，这样就可以保证一旦方法中有一个更新操作失败了，整个方法都可以回滚。

4.6.2 提供事务相关操作

JDBC 提供了事务常用的操作，比如开启事务、提交事务、回滚事务等，我们可以将这些操作统一封装在 DatabaseHelper 中，就像下面这样：

```
/**
 * 数据库操作助手类
 */
public final class DatabaseHelper {
    ...
    /**
     * 开启事务
     */
    public static void beginTransaction() {
        Connection conn = getConnection();
        if (conn != null) {
            try {
                conn.setAutoCommit(false);
            } catch (SQLException e) {
                LOGGER.error("begin transaction failure", e);
                throw new RuntimeException(e);
            } finally {
                CONNECTION_HOLDER.set(conn);
            }
        }
    }

    /**
     * 提交事务
     */
    public static void commitTransaction() {
        Connection conn = getConnection();
        if (conn != null) {
            try {
                conn.commit();
                conn.close();
            } catch (SQLException e) {
                LOGGER.error("commit transaction failure", e);
            }
        }
    }
}
```

```

        throw new RuntimeException(e);
    } finally {
        CONNECTION_HOLDER.remove();
    }
}

/**
 * 回滚事务
 */
public static void rollbackTransaction() {
    Connection conn = getConnection();
    if (conn != null) {
        try {
            conn.rollback();
            conn.close();
        } catch (SQLException e) {
            LOGGER.error("rollback transaction failure", e);
            throw new RuntimeException(e);
        } finally {
            CONNECTION_HOLDER.remove();
        }
    }
}

...
}

```

需要注意的是，默认是自动提交事务的，所以需要将自动提交属性设置为 `false`。在开启事务完毕后，需要将 `Connection` 对象放入本地线程变量中。当事务提交或回滚后，需要移除本地线程变量中的 `Connection` 对象。

4.6.3 编写事务代理切面类

我们需要编写一个名为 `TransactionProxy` 的类，让它实现 `Proxy` 接口，在 `doProxy` 方法中完成事务控制的相关逻辑，代码如下：

```

package org.smart4j.framework.proxy;

import java.lang.reflect.Method;

```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.smart4j.framework.annotation.Transaction;
import org.smart4j.framework.helper.DatabaseHelper;

/**
 * 事务代理
 *
 * @author huangyong
 * @since 1.0.0
 */
public class TransactionProxy implements Proxy {

    private static final Logger LOGGER = LoggerFactory.getLogger(TransactionProxy.class);

    private static final ThreadLocal<Boolean> FLAG_HOLDER = new ThreadLocal<Boolean>() {
        @Override
        protected Boolean initialValue() {
            return false;
        }
    };

    @Override
    public Object doProxy(ProxyChain proxyChain) throws Throwable {
        Object result;
        boolean flag = FLAG_HOLDER.get();
        Method method = proxyChain.getTargetMethod();
        if (!flag && method.isAnnotationPresent(Transaction.class)) {
            FLAG_HOLDER.set(true);
            try {
                DatabaseHelper.beginTransaction();
                LOGGER.debug("begin transaction");
                result = proxyChain.doProxyChain();
                DatabaseHelper.commitTransaction();
                LOGGER.debug("commit transaction");
            } catch (Exception e) {
```

```

        DatabaseHelper.rollbackTransaction();
        LOGGER.debug("rollback transaction");
        throw e;
    } finally {
        FLAG_HOLDER.remove();
    }
} else {
    result = proxyChain.doProxyChain();
}
return result;
}
}

```

这里定义了一个名为 `FLAG_HOLDER` 的本地线程变量，它是一个标志，可以保证同一线程中事务控制相关逻辑只会执行一次。通过 `ProxyChain` 对象可获取目标方法，进而判断该方法是否带有 `Transaction` 注解。首先调用 `DatabaseHelper.beginTransaction` 方法开启事务，然后调用 `ProxyChain` 对象的 `doProxyChain` 方法执行目标方法，接着调用 `DatabaseHelper.commitTransaction` 提交事务，或者在异常处理中调用 `DatabaseHelper.rollbackTransaction` 方法回滚事务，最后别忘了移除 `FLAG_HOLDER` 本地线程变量中的标志。

4.6.4 在框架中添加事务代理机制

只需对 `AopHelper` 类的 `createProxyMap` 方法稍作调整，即可将事务代理机制添加到框架中。由于之前添加到 AOP 框架中的只是普通切面代理，现在需要将事务代理也添加进去。我们定义两个私有方法，一个用于添加普通切面代理，另一个用于添加事务代理，就像下面这样：

```

public final class AopHelper {
    ...
    private static Map<Class<?>, Set<Class<?>>>> createProxyMap() throws
    Exception {
        Map<Class<?>, Set<Class<?>>>> proxyMap = new HashMap<Class<?>,
        Set<Class<?>>>>();
        addAspectProxy(proxyMap);
        addTransactionProxy(proxyMap);
        return proxyMap;
    }

    private static void addAspectProxy(Map<Class<?>, Set<Class<?>>>>
    proxyMap) throws Exception {

```

```

        Set<Class<?>> proxyClassSet = ClassHelper.getClassSetBySuper
            (AspectProxy.class);
        for (Class<?> proxyClass : proxyClassSet) {
            if (proxyClass.isAnnotationPresent(Aspect.class)) {
                Aspect aspect = proxyClass.getAnnotation(Aspect.class);
                Set<Class<?>> targetClassSet = createTargetClassSet(aspect);
                proxyMap.put(proxyClass, targetClassSet);
            }
        }
    }

    private static void addTransactionProxy(Map<Class<?>, Set<Class<?>>>
        proxyMap) {
        Set<Class<?>> serviceClassSet = ClassHelper.getClassSetByAnnotation
            (Service.class);
        proxyMap.put(TransactionProxy.class, serviceClassSet);
    }
    ...
}

```

当运行应用程序后，就可看到事务代理切面中输出的日志信息了。到此为止，一个简单的事务控制框架就开发完毕了。

4.7 总结

在本章中，我们首先开发了一个 Proxy 框架，然后通过该框架实现了 AOP 特性，通过模版方法模式提供了一个抽象切面类 AspectProxy，底层实际上是用 CGLib 开发的一个动态代理框架。我们只需根据业务需求定义自己的切面类，扩展 AspectProxy 抽象类并定义 Aspect 注解，然后完成特定的钩子方法，即可将横切逻辑与业务逻辑相分离，这就是 AOP 要做的事情。最后，我们使用 Proxy 框架实现了一个简单的事务代理框架，可在 Service 类的方法中使用 Transaction 注解来定义需要进行事务控制的方法。

在下一章中，我们将对现有框架做一个优化，完善目前框架中需要提高的地方，再提供开发中一些常用的特性。



第 5 章

框架优化与功能扩展

目前的框架已具备 IOC、AOP、MVC 等特性，基本上一个简单的 Web 应用可以通过它来开发了。但或多或少还存在一些不足，需要优化的地方还有很多，此外，还有一些更强大的功能需要不断地补充进来。

本章将对现有框架进行优化，并提供一些扩展功能。通过本章的学习，您可以了解到：

- 如何优化 Action 参数；
- 如何实现文件上传功能；
- 如何与 Servlet API 完全解耦；
- 如何实现安全控制框架；
- 如何实现 Web 服务框架。

优化与扩展是永无止境的，但都是有一个开头的，我们首先从一件最简单的事情开始，不妨先对现有框架的 Action 参数做一个优化。

5.1 优化 Action 参数

5.1.1 明确 Action 参数优化目标

对于某些 Action 方法而言，其实根本就用不上 Param 参数，但框架需要我们必须提供一个 Param 参数，这样未免有些勉强。

比如以下代码示例：

```
@Controller
public class CustomerController {
    ...
    /**
     * 进入 客户列表 界面
     */
    @Action("get:/customer")
    public View index(Param param) {
        List<Customer> customerList = customerService.getCustomerList();
        return new View("customer.jsp").addModel("customerList", customer-
            List);
    }
    ...
}
```

这个 Action 方法根本就不需要 Param 参数，放在这里确实有些累赘，我们得想办法去掉这个参数，并且确保框架能够成功地调用 Action 方法。

当不需要请求参数时，如何做到省略 Action 方法的 Param 参数呢？这正是我们接下来要优化的地方。

5.1.2 动手优化 Action 参数使用方式

我们不得不修改框架代码来支持这个特性。先来看看现在框架是怎样调用 Action 方法的：

```
@WebServlet(urlPatterns = "/*", loadOnStartup = 0)
public class DispatcherServlet extends HttpServlet {
    ...
    public void service(HttpServletRequest request, HttpServletResponse
```

```

        response) throws ServletException, IOException {
            ...
            Param param = new Param(paramMap);

            Method actionMethod = handler.getActionMethod();
            Object result = ReflectionUtil.invokeMethod(controllerBean, action-
            Method, param);
            ...
        }
        ...
    }
}

```

在以上代码中，每次请求我们都创建了一个 **Param** 对象，并将其放入 **Action** 参数中。我们可以考虑这样做，当框架拿到 **Param** 对象后，判断一下该对象是否为 **null**，于是就有两种情况，可通过一个 **if...else...** 来处理这两种情况，就像下面这样：

```

@WebServlet(urlPatterns = "/*", loadOnStartup = 0)
public class DispatcherServlet extends HttpServlet {
    ...
    public void service(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        ...
        Param param = new Param(paramMap);

        Object result;
        Method actionMethod = handler.getActionMethod();
        if (param.isEmpty()) {
            result = ReflectionUtil.invokeMethod(controllerBean, action-
            Method);
        } else {
            result = ReflectionUtil.invokeMethod(controllerBean, action-
            Method, param);
        }
        ...
    }
    ...
}

```

当 **param.isEmpty()** 为 **true** 时，可以不将 **Param** 参数传入 **Action** 方法中，反之则将 **Param** 参

数传入 `Action` 方法中。

因此，我们需要为 `Param` 类添加一个 `isEmpty` 方法，就像下面这样：

```
public class Param {  
    ...  
    /**  
     * 验证参数是否为空  
     */  
    public boolean isEmpty() {  
        return CollectionUtil.isEmpty(paramMap);  
    }  
}
```

所谓验证参数是否为空，实际上是判断 `Param` 中的 `paramMap` 是否为空。

一旦做了这样的调整，我们就可以在 `Action` 方法中根据实际情况省略 `Param` 参数了，就像下面这样：

```
@Controller  
public class CustomerController {  
    ...  
    /**  
     * 进入 客户列表 界面  
     */  
    @Action("get:/customer")  
    public View index() {  
        List<Customer> customerList = customerService.getCustomerList();  
        return new View("customer.jsp").addModel("customerList", customerList);  
    }  
    ...  
}
```

至此，`Action` 参数优化完毕，我们可以根据实际情况自由选择是否在 `Action` 方法中使用 `Param` 参数。这样的框架更加灵活，从使用的角度来看也更加完美。

提示：框架的实现细节也许比较复杂，但框架的创建者一定要站在使用者的角度，尽可能地简化框架的使用方法。注重细节并不断优化，这是每个框架创建者的职责。

当然，目前框架中还有很多地方值得去优化，请读者自行思考并动手实践。

5.2 提供文件上传特性

5.2.1 确定文件上传使用场景

通常情况下，我们可通过一个 form（表单）来上传文件，就以下的“创建客户”为例来说明（对应的文件名是 customer_create.jsp），需要提供一个 form，并将其 enctype 属性设置为 multipart/form-data，表示以 form data 方式提交表单数据。

注意：enctype 的默认值为 application/x-www-form-urlencoded，表示以 url encoded 方式提交表单数据。

下面我们使用 jQuery 与 jQuery Form 插件快速编写一个基于 Ajax 的文件上传表单，代码如下：

```
<%@ page pageEncoding="UTF-8" %>
<html>
<head>
    <title>客户管理 - 创建客户</title>
</head>
<body>

<h1>创建客户界面</h1>

<form id="customer_form" enctype="multipart/form-data">
    <table>
        <tr>
            <td>客户名称: </td>
            <td>
                <input type="text" name="name" value="{customer.name}">
            </td>
        </tr>
        <tr>
            <td>联系人: </td>
```

```
        <td>
            <input type="text" name="contact" value="${customer.contact}">
        </td>
    </tr>
    <tr>
        <td>电话号码: </td>
        <td>
            <input type="text" name="telephone" value="${customer.tele-
            phone}">
        </td>
    </tr>
    <tr>
        <td>邮箱地址: </td>
        <td>
            <input type="text" name="email" value="${customer.email}">
        </td>
    </tr>
    <tr>
        <td>照片: </td>
        <td>
            <input type="file" name="photo" value="${customer.photo}">
        </td>
    </tr>
</table>
<button type="submit">保存</button>
</form>

<script src="${BASE}/asset/lib/jquery/jquery.min.js"></script>
<script src="${BASE}/asset/lib/jquery-form/jquery.form.min.js"></script>
<script>
    $(function() {
        $('#customer_form').ajaxForm({
            type: 'post',
            url: '${BASE}/customer_create',
            success: function(data) {
                if (data) {
                    location.href = '${BASE}/customer';
                }
            }
        });
    });
</script>
```

```

        }
    });
});
</script>

</body>
</html>

```

当表单提交时，请求会转发到 `CustomerController` 的 `createSubmit` 方法上。该方法带有一个 `Param` 参数，我们打算通过该参数来获取“表单字段的名值对映射”与“所上传的文件参数对象”，应该如何编写代码呢？以下是我们要实现的目标：

```

@Controller
public class CustomerController {
    ...
    /**
     * 处理 创建客户 请求
     */
    @Action("post:/customer_create")
    public Data createSubmit(Param param) {
        Map<String, Object> fieldMap = param.getFieldMap();
        FileParam fileParam = param.getFile("photo");
        boolean result = customerService.createCustomer(fieldMap, fileParam);
        return new Data(result);
    }
    ...
}

```

调用 `Param` 的 `getFieldMap` 方法来获取表单字段的名/值对映射（`Map fieldMap`），指定一个具体的文件字段名称（`"photo"`），并调用 `getFile` 方法即可获取对应的文件参数对象（`FileParam fileParam`）。随后，可调用 `customerService` 的 `createCustomer` 方法，将 `fieldMap` 与 `fileParam` 这两个参数传入。

`Controller` 层的代码就是这样简单，因为具体的业务逻辑都封装在 `Service` 层了。对于 `CustomerService` 而言，只需写几行代码，即可实现具体的业务逻辑，将输入参数存入数据库，同时将文件上传到服务器上。

```

@Service
public class CustomerService {
    ...
}

```

```
/**
 * 创建客户
 */
@Transactional
public boolean createCustomer(Map<String, Object> fieldMap, FileParam
fileParam) {
    boolean result = DatabaseHelper.insertEntity(Customer.class,
        fieldMap);
    if (result) {
        UploadHelper.uploadFile("/tmp/upload/", fileParam);
    }
    return result;
}
...
}
```

可见除了使用 `DatabaseHelper` 操作数据库，还可以通过 `UploadHelper` 将文件上传到指定的服务器目录中。

注意：实际上，完全可以通过代码来读取配置文件中定义的文件上传路径，此处只是为了简化，请注意。

我们把计划要完成的事情总结一下：

- (1) 改造 `Param` 结构，可通过它来获取已上传的文件参数（`FileParam`）。
- (2) 使用 `UploadHelper` 助手类来上传文件。

我们首先根据一个使用场景，知道了需要做什么事情，下面按步骤完成任务即可。

5.2.2 实现文件上传功能

我们不妨从 `FileParam` 开始，它实际上是一个用于封装文件参数的 `JavaBean`，代码如下：

```
package org.smart4j.framework.bean;

import java.io.InputStream;

/**
```

```
* 封装上传文件参数
*
* @author huangyong
* @since 1.0.0
*/
public class FileParam {

    private String fieldName;
    private String fileName;
    private long fileSize;
    private String contentType;
    private InputStream inputStream;

    public FileParam(String fieldName, String fileName, long fileSize,
String contentType, InputStream inputStream) {
        this.fieldName = fieldName;
        this.fileName = fileName;
        this.fileSize = fileSize;
        this.contentType = contentType;
        this.inputStream = inputStream;
    }

    public String getFieldName() {
        return fieldName;
    }

    public String getFileName() {
        return fileName;
    }

    public long getFileSize() {
        return fileSize;
    }

    public String getContentType() {
        return contentType;
    }
}
```

```
public InputStream getInputStream() {  
    return inputStream;  
}  
}
```

在 `FileParam` 类中定义了几个成员变量：

- (1) `fieldName` 表示文件表单的字段名。
- (2) `fileName` 表示上传文件的文件名。
- (3) `fileSize` 表示上传文件的文件大小。
- (4) `contentType` 表示上传文件的 `Content-Type`，可判断文件类型。
- (5) `inputStream` 表示上传文件的字节输入流。

除了文件参数 (`FileParam`)，我们还可以提供一个表单参数 (`FormParam`)，代码如下：

```
package org.smart4j.framework.bean;  
  
/**  
 * 封装表单参数  
 *  
 * @author huangyong  
 * @since 1.0.0  
 */  
public class FormParam {  
  
    private String fieldName;  
    private Object fieldValue;  
  
    public FormParam(String fieldName, Object fieldValue) {  
        this.fieldName = fieldName;  
        this.fieldValue = fieldValue;  
    }  
  
    public String getFieldName() {  
        return fieldName;  
    }  
  
    public Object getFieldValue() {
```

```
        return fieldValue;
    }
}
```

在一个表单中，所有的参数可分为两类：表单参数与文件参数。有必要将 **Param** 类做一个重构，让它封装这两类参数，并提供一系列的 **get** 方法，用于从该对象中获取指定的参数。以下是对 **Param** 类重构后的代码：

```
package org.smart4j.framework.bean;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.smart4j.framework.util.CastUtil;
import org.smart4j.framework.util.CollectionUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * 请求参数对象
 */
public class Param {

    private List<FormParam> formParamList;

    private List<FileParam> fileParamList;

    public Param(List<FormParam> formParamList) {
        this.formParamList = formParamList;
    }

    public Param(List<FormParam> formParamList, List<FileParam> fileParamList) {
        this.formParamList = formParamList;
        this.fileParamList = fileParamList;
    }

    /**
     * 获取请求参数映射
     */
}
```

```

    */
    public Map<String, Object> getFieldMap() {
        Map<String, Object> fieldMap = new HashMap<String, Object>();
        if (CollectionUtil.isEmpty(formParamList)) {
            for (FormParam formParam : formParamList) {
                String fieldName = formParam.getFieldName();
                Object fieldValue = formParam.getFieldValue();
                if (fieldMap.containsKey(fieldName)) {
                    fieldValue = fieldMap.get(fieldName) + StringUtil.SEPARATOR + fieldValue;
                }
                fieldMap.put(fieldName, fieldValue);
            }
        }
        return fieldMap;
    }

    /**
     * 获取上传文件映射
     */
    public Map<String, List<FileParam>> getFileMap() {
        Map<String, List<FileParam>> fileMap = new HashMap<String, List<FileParam>>();
        if (CollectionUtil.isEmpty(fileParamList)) {
            for (FileParam fileParam : fileParamList) {
                String fieldName = fileParam.getFieldName();
                List<FileParam> fileParamList;
                if (fileMap.containsKey(fieldName)) {
                    fileParamList = fileMap.get(fieldName);
                } else {
                    fileParamList = new ArrayList<FileParam>();
                }
                fileParamList.add(fileParam);
                fileMap.put(fieldName, fileParamList);
            }
        }
        return fileMap;
    }
}

```

```
/**
 * 获取所有上传文件
 */
public List<FileParam> getFileList(String fieldName) {
    return getFileMap().get(fieldName);
}

/**
 * 获取唯一上传文件
 */
public FileParam getFile(String fieldName) {
    List<FileParam> fileParamList = getFileList(fieldName);
    if (CollectionUtil.isEmpty(fileParamList) && fileParamList.
        size() == 1) {
        return fileParamList.get(0);
    }
    return null;
}

/**
 * 验证参数是否为空
 */
public boolean isEmpty() {
    return CollectionUtil.isEmpty(formParamList) && CollectionUtil.
        isEmpty(fileParamList);
}

/**
 * 根据参数名获取 String 型参数值
 */
public String getString(String name) {
    return CastUtil.castString(getFieldMap().get(name));
}

/**
 * 根据参数名获取 double 型参数值
 */
```

```

    public double getDouble(String name) {
        return CastUtil.castDouble(getFieldMap().get(name));
    }

    /**
     * 根据参数名获取 long 型参数值
     */
    public long getLong(String name) {
        return CastUtil.castLong(getFieldMap().get(name));
    }

    /**
     * 根据参数名获取 int 型参数值
     */
    public int getInt(String name) {
        return CastUtil.castInt(getFieldMap().get(name));
    }

    /**
     * 根据参数名获取 boolean 型参数值
     */
    public boolean getBoolean(String name) {
        return CastUtil.castBoolean(getFieldMap().get(name));
    }
}

```

可见 `Param` 包含了两个成员变量：`List formParamList` 与 `List fileParamList`；它们分别封装了表单参数与文件参数。随后提供了两个构造器，用于初始化 `Param` 对象，还提供了两个 `get` 方法，分别用于获取所有的表单参数与文件参数。返回值均为 `Map` 类型，其中 `Map` 表示请求参数映射，`Map<>` 表示上传文件映射。对于同名的请求参数，通过一个特殊的分隔符进行了处理，该分隔符定义在 `StringUtil` 类中，代码如下：

```

public final class StringUtil {

    /**
     * 字符串分隔符
     */
    public static final String SEPARATOR = String.valueOf((char) 29);
}

```

```
...
}
```

对于同名的上传文件，通过一个 List 进行了封装，可轻松实现多文件上传的需求。可通过 List `getFileList(String fieldName)` 方法获取所有上传文件，若只上传了一个文件，则可直接使用 `FileParam getFile(String fieldName)` 方法获取唯一上传文件。还提供了一个 `boolean isEmpty()` 方法，用于验证参数是否为空。最后，提供了一组根据参数名获取指定类型的方法，例如，`String getString(String name)`、`double getDouble(String name)` 等。

可借助 Apache Commons 提供的 `FileUpload` 类库实现文件上传特性，首先需要在 `pom.xml` 中添加如下依赖：

```
<dependency>
  <groupId>commons-fileupload</groupId>
  <artifactId>commons-fileupload</artifactId>
  <version>1.3.1</version>
</dependency>
```

接下来我们需要编写一个 `UploadHelper` 类来封装 Apache Commons `FileUpload` 的相关代码：

```
package org.smart4j.framework.helper;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.FileUploadException;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.smart4j.framework.bean.FileParam;
import org.smart4j.framework.bean.FormParam;
import org.smart4j.framework.bean.Param;
import org.smart4j.framework.util.CollectionUtil;
import org.smart4j.framework.util.FileUtil;
import org.smart4j.framework.util.StreamUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * 文件上传助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class UploadHelper {

    private static final Logger LOGGER = LoggerFactory.getLogger(Upload-
        Helper.class);

    /**
     * Apache Commons FileUpload 提供的 Servlet 文件上传对象
     */
    private static ServletFileUpload servletFileUpload;

    /**
     * 初始化
     */
    public static void init(ServletContext servletContext) {
        File repository = (File) servletContext.getAttribute("javax.
            servlet.context.tempdir");
        servletFileUpload = new ServletFileUpload(new DiskFileItemFactory
            (DiskFileItemFactory.DEFAULT_SIZE_THRESHOLD, repository));
        int uploadLimit = ConfigHelper.getAppUploadLimit();
        if (uploadLimit != 0) {
            servletFileUpload.setFileSizeMax(uploadLimit * 1024 * 1024);
        }
    }
}
```

```
/**
 * 判断请求是否为 multipart 类型
 */
public static boolean isMultipart(HttpServletRequest request) {
    return ServletFileUpload.isMultipartContent(request);
}

/**
 * 创建请求对象
 */
public static Param createParam(HttpServletRequest request) throws
IOException {
    List<FormParam> formParamList = new ArrayList<FormParam>();
    List<FileParam> fileParamList = new ArrayList<FileParam>();
    try {
        Map<String, List<FileItem>> fileItemListMap = servletFileUpload.
            parseParameterMap(request);
        if (CollectionUtil.isNotEmpty(fileItemListMap)) {
            for (Map.Entry<String, List<FileItem>> fileItemListEntry :
                fileItemListMap.entrySet()) {
                String fieldName = fileItemListEntry.getKey();
                List<FileItem> fileItemList = fileItemListEntry.get-
                    Value();
                if (CollectionUtil.isNotEmpty(fileItemList)) {
                    for (FileItem fileItem : fileItemList) {
                        if (fileItem.isFormField()) {
                            String fieldValue = fileItem.getString("UTF-
                                8");
                            formParamList.add(new FormParam(fieldName,
                                fieldValue));
                        } else {
                            String fileName = FileUtil.getRealFileName(new
                                String(fileItem.getName().getBytes(), "UTF-8"));
                            if (StringUtil.isNotEmpty(fileName)) {
                                long fileSize = fileItem.getSize();
                                String contentType = fileItem.getConten-
                                    tType();

```

```

        InputStream inputStream = fileItem.getInputStream();
        fileParamList.add(new FileParam(fieldName,
            fileName, fileSize, contentType, inputStream));
    }
}

}

}

}

} catch (FileUploadException e) {
    LOGGER.error("create param failure", e);
    throw new RuntimeException(e);
}
return new Param(formParamList, fileParamList);
}

/**
 * 上传文件
 */
public static void uploadFile(String basePath, FileParam fileParam) {
    try {
        if (fileParam != null) {
            String filePath = basePath + fileParam.getFileName();
            FileUtil.createFile(filePath);
            InputStream inputStream = new BufferedInputStream(fileParam.
                getInputStream());
            OutputStream outputStream = new BufferedOutputStream(new
                FileOutputStream(filePath));
            StreamUtil.copyStream(inputStream, outputStream);
        }
    } catch (Exception e) {
        LOGGER.error("upload file failure", e);
        throw new RuntimeException(e);
    }
}
}

```

```

/**
 * 批量上传文件
 */
public static void uploadFile(String basePath, List<FileParam> file-
ParamList) {
    try {
        if (CollectionUtil.isNotEmpty(fileParamList)) {
            for (FileParam fileParam : fileParamList) {
                uploadFile(basePath, fileParam);
            }
        }
    } catch (Exception e) {
        LOGGER.error("upload file failure", e);
        throw new RuntimeException(e);
    }
}
}

```

需要提供一个 `void init(ServletContext servletContext)` 方法，在该方法中初始化 `org.apache.commons.fileupload.servlet.ServletFileUpload` 对象。一般情况下，只需设置一个上传文件的临时目录与上传文件的最大限制；上传文件的临时目录可设置为应用服务器的临时目录，上传文件的最大限制可让用户自行配置。所以我们使用了 `ConfigHelper.getAppUploadLimit()` 来获取，可在 `smart.properties` 文件中进行配置。

首先，在 `ConfigConstant` 中添加一个配置常量 `APP_UPLOAD_LIMIT`：

```

public interface ConfigConstant {
    ...
    String APP_UPLOAD_LIMIT = "smart.framework.app.upload_limit";
}

```

这也就意味着，我们可在 `smart.properties` 文件中使用 `smart.framework.app.upload_limit` 配置项来设定上传文件的最大限制。

然后，在 `ConfigHelper` 中添加一个 `int getAppUploadLimit()` 方法，用于获取该配置的值。此时可设置该配置的初始值（为 10），也就是说，若不在 `smart.properties` 文件中提供该配置，则上传文件的最大限制是 10MB。

```

public final class ConfigHelper {
    ...
    /**

```

```

        * 获取应用文件上传限制
        */
        public static int getAppUploadLimit() {
            return PropsUtil.getInt(CONFIG_PROPS, ConfigConstant.APP_UPLOAD_
                LIMIT, 10);
        }
    }
}

```

在 `UploadHelper` 中提供了一个 `boolean isMultipart(HttpServletRequest request)` 方法，用于判断当前请求对象是否为 `multipart` 类型。只有在上传文件时对应的请求类型才是 `multipart` 类型，也就是说，可通过 `isMultipart` 方法来判断当前请求是否为文件上传请求。

接下来提供了一个非常重要的方法，可从当前请求中创建 `Param` 对象，它就是 `Param createParam(HttpServletRequest request)` 方法；其中我们使用了 `ServletFileUpload` 对象来解析请求参数，并通过遍历所有请求参数来初始化 `List formParamList` 与 `List fileParamList` 变量的值。在遍历请求参数时，需要对当前的 `org.apache.commons.fileupload.FileItem` 对象进行判断，若为普通表单字段（调用 `fileItem.isFormField()` 返回 `true`），则创建 `FormParam` 对象，并添加到 `formParamList` 对象中。否则即为文件上传字段，通过 `FileUtil` 提供的 `getRealFileName` 来获取上传文件的真实文件名，并从 `FileItem` 对象中构造 `FileParam` 对象，添加到 `fileParamList` 对象中。最后，通过 `formParamList` 与 `fileParamList` 来构造 `Param` 对象并返回。

`FileUtil` 代码如下：

```

package org.smart4j.framework.util;

import java.io.File;
import org.apache.commons.io.FileUtils;
import org.apache.commons.io.FilenameUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 文件操作工具类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class FileUtil {

```

```

private static final Logger LOGGER = LoggerFactory.getLogger(File-
Util.class);

/**
 * 获取真实文件名（自动去掉文件路径）
 */
public static String getRealFileName(String fileName) {
    return FilenameUtils.getName(fileName);
}

/**
 * 创建文件
 */
public static File createFile(String filePath) {
    File file;
    try {
        file = new File(filePath);
        File parentDir = file.getParentFile();
        if (!parentDir.exists()) {
            FileUtils.forceMkdir(parentDir);
        }
    } catch (Exception e) {
        LOGGER.error("create file failure", e);
        throw new RuntimeException(e);
    }
    return file;
}
}

```

最后还提供了两个用于上传文件的方法，一个用于上传单个文件，另一个用于批量上传。此时用到了 `StreamUtil` 工具类的 `copyStream` 方法，代码如下：

```

public final class StreamUtil {
    ...
    /**
     * 将输入流复制到输出流
     */
    public static void copyStream(InputStream inputStream, OutputStream
outputStream) {

```

```
        try {
            int length;
            byte[] buffer = new byte[4 * 1024];
            while((length=inputStream.read(buffer, 0, buffer.length))!=-1){
                outputStream.write(buffer, 0, length);
            }
            outputStream.flush();
        } catch (Exception e) {
            LOGGER.error("copy stream failure", e);
            throw new RuntimeException(e);
        } finally {
            try {
                inputStream.close();
                outputStream.close();
            } catch (Exception e) {
                LOGGER.error("close stream failure", e);
            }
        }
    }
}
```

现在 UploadHelper 已编写完毕，接下来需要找一个地方来调用 init 方法。整个 Web 框架的入口也就是 DispatcherServlet 的 init 方法了，所有我们需要在该方法中调用 UploadHelper 的 init 方法。

除了在 DispatcherServlet 的 init 方法中添加了一行代码，还需要对 service 代码进行一些重构。首先需要跳过/favicon.ico 请求，只处理普通的请求。然后需要判断请求对象是否为上传文件，针对两种不同的情况来创建 Param 对象，其中通过 UploadHelper 来创建的方式已在前面描述了。相应地，我们也对以前的代码进行封装，提供一个名为 RequestHelper 的类，并通过它的 createParam 方法来初始化 Param 对象。

RequestHelper 的代码如下：

```
package org.smart4j.framework.helper;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Enumuration;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
```

```

import org.smart4j.framework.bean.FormParam;
import org.smart4j.framework.bean.Param;
import org.smart4j.framework.util.ArrayUtil;
import org.smart4j.framework.util.CodecUtil;
import org.smart4j.framework.util.StreamUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * 请求助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class RequestHelper {

    /**
     * 创建请求对象
     */
    public static Param createParam(HttpServletRequest request) throws
        IOException {
        List<FormParam> formParamList = new ArrayList<FormParam>();
        formParamList.addAll(parseParameterNames(request));
        formParamList.addAll(parseInputStream(request));
        return new Param(formParamList);
    }

    private static List<FormParam> parseParameterNames(HttpServletRequest
        request) {
        List<FormParam> formParamList = new ArrayList<FormParam>();
        Enumeration<String> paramNames = request.getParameterNames();
        while (paramNames.hasMoreElements()) {
            String fieldName = paramNames.nextElement();
            String[] fieldValues = request.getParameterValues(fieldName);
            if (ArrayUtil.isNotEmpty(fieldValues)) {
                Object fieldValue;
                if (fieldValues.length == 1) {
                    fieldValue = fieldValues[0];
                } else {

```

```
        StringBuilder sb = new StringBuilder("");
        for (int i = 0; i < fieldValues.length; i++) {
            sb.append(fieldValues[i]);
            if (i != fieldValues.length - 1) {
                sb.append(StringUtil.SEPARATOR);
            }
        }
        fieldValue = sb.toString();
    }
    formParamList.add(new FormParam(fieldName, fieldValue));
}
}
return formParamList;
}

private static List<FormParam> parseInputStream(HttpServletRequest request
request) throws IOException {
    List<FormParam> formParamList = new ArrayList<FormParam>();
    String body = CodecUtil.decodeURL(StreamUtil.getString(request.
getInputStream()));
    if (StringUtil.isEmpty(body)) {
        String[] kvs = StringUtil.splitString(body, "&");
        if (ArrayUtil.isEmpty(kvs)) {
            for (String kv : kvs) {
                String[] array = StringUtil.splitString(kv, "=");
                if (ArrayUtil.isEmpty(array) && array.length == 2) {
                    String fieldName = array[0];
                    String fieldValue = array[1];
                    formParamList.add(new FormParam(fieldName, field-
Value));
                }
            }
        }
    }
    return formParamList;
}
}
```

可见以上代码逻辑并未变化，只是将以前放在 `DispatcherServlet` 中的相关代码搬到了 `RequestHelper` 中了。最后获取的 `View` 同样也分两种情况进行了处理，只是此时并未提供其他类来封装这些代码，而是直接在当前类中添加了两个私有方法 `handleViewResult` 与 `handleDataResult` 来封装代码。

以下是重构后的 `DispatcherServlet` 代码：

```
package org.smart4j.framework;

import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.Method;
import java.util.Map;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.smart4j.framework.bean.Data;
import org.smart4j.framework.bean.Handler;
import org.smart4j.framework.bean.Param;
import org.smart4j.framework.bean.View;
import org.smart4j.framework.helper.BeanHelper;
import org.smart4j.framework.helper.ConfigHelper;
import org.smart4j.framework.helper.ControllerHelper;
import org.smart4j.framework.helper.RequestHelper;
import org.smart4j.framework.helper.UploadHelper;
import org.smart4j.framework.util.JsonUtil;
import org.smart4j.framework.util.ReflectionUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * 请求转发器
 */
@WebServlet(urlPatterns = "/*", loadOnStartup = 0)
public class DispatcherServlet extends HttpServlet {
```

```
@Override
public void init(ServletConfig servletConfig) throws ServletException{
    HelperLoader.init();

    ServletContext servletContext = servletConfig.getServletContext();

    ServletRegistration jspServlet = servletContext.getServletRegistration("jsp");
    jspServlet.addMapping(ConfigHelper.getAppJspPath() + "*");

    ServletRegistration defaultServlet = servletContext.getServletRegistration("default");
    defaultServlet.addMapping(ConfigHelper.getAppAssetPath() + "*");

    UploadHelper.init(servletContext);
}

@Override
public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String requestMethod = request.getMethod().toLowerCase();
    String requestPath = request.getPathInfo();

    if (requestPath.equals("/favicon.ico")) {
        return;
    }

    Handler handler = ControllerHelper.getHandler(requestMethod, requestPath);
    if (handler != null) {
        Class<?> controllerClass = handler.getControllerClass();
        Object controllerBean = BeanHelper.getBean(controllerClass);

        Param param;
        if (UploadHelper.isMultipart(request)) {
            param = UploadHelper.createParam(request);
        } else {
```

```

        param = RequestHelper.createParam(request);
    }

    Object result;
    Method actionMethod = handler.getActionMethod();
    if (param.isEmpty()) {
        result = ReflectionUtil.invokeMethod(controllerBean, actionMethod);
    } else {
        result = ReflectionUtil.invokeMethod(controllerBean, actionMethod, param);
    }

    if (result instanceof View) {
        handleViewResult((View) result, request, response);
    } else if (result instanceof Data) {
        handleDataResult((Data) result, response);
    }
}

private void handleViewResult(View view, HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException {
    String path = view.getPath();
    if (StringUtil.isEmpty(path)) {
        if (path.startsWith("/")) {
            response.sendRedirect(request.getContextPath() + path);
        } else {
            Map<String, Object> model = view.getModel();
            for (Map.Entry<String, Object> entry : model.entrySet()) {
                request.setAttribute(entry.getKey(), entry.getValue());
            }
            request.getRequestDispatcher(ConfigHelper.getAppJspPath() +
                path).forward(request, response);
        }
    }
}
}

```

```
private void handleDataResult(Data data, HttpServletResponse response)
throws IOException {
    Object model = data.getModel();
    if (model != null) {
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        PrintWriter writer = response.getWriter();
        String json = JsonUtil.toJson(model);
        writer.write(json);
        writer.flush();
        writer.close();
    }
}
```

此时，一个简单的文件上传特性已基本具备，可以在框架中正常使用了。

5.3 与 Servlet API 解耦

5.3.1 为何需要与 Servlet API 解耦

目前在 Controller 中是无法调用 Servlet API 的，因为无法获取 Request 与 Response 这类对象，我们必须在 DispatcherServlet 中将对象传递给 Controller 的 Action 方法才能拿到这些对象，这显然会增加 Controller 对 Servlet API 的耦合。最好能让 Controller 完全不使用 Servlet API 就能操作 Request 与 Response 对象，有什么办法能够做到呢？

最容易拿到 Request 与 Response 对象的地方就是 DispatcherServlet 的 service 方法：

```
public class DispatcherServlet extends HttpServlet {

    @Override
    public void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        ...
    }
}
```

然而，我们又不想把 Request 与 Response 对象传递到 Controller 的 Action 方法中，所以我

们需要提供一个线程安全的对象，通过它来封装 Request 与 Response 对象，并提供一系列常用的 Servlet API，这样我们就可以在 Controller 中随时通过该对象来操作 Request 与 Response 对象的方法了。需要强调的是，这个对象一定是线程安全的，也就是说，每个请求线程独自拥有一份 Request 与 Response 对象，不同请求线程之间是隔离的。

5.3.2 与 Servlet API 解耦的实现过程

一个最简单的思路是，编写一个 ServletHelper 类，让它去封装 Request 与 Response 对象，提供常用的 ServletAPI 工具方法，并利用 ThreadLocal 技术来保证线程安全，代码应该是这样的：

```
import java.io.IOException;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Servlet 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class ServletHelper {

    private static final Logger LOGGER = LoggerFactory.getLogger(ServletHelper.class);

    /**
     * 使每个线程独自拥有一份 ServletHelper 实例
     */
    private static final ThreadLocal<ServletHelper> SERVLET_HELPER_HOLDER
        = new ThreadLocal<ServletHelper>();

    private HttpServletRequest request;
    private HttpServletResponse response;
```

```
private ServletHelper(HttpServletRequest request, HttpServletResponse
response) {
    this.request = request;
    this.response = response;
}

/**
 * 初始化
 */
public static void init(HttpServletRequest request, HttpServletResponse
response) {
    SERVLET_HELPER_HOLDER.set(new ServletHelper(request, response));
}

/**
 * 销毁
 */
public static void destroy() {
    SERVLET_HELPER_HOLDER.remove();
}

/**
 * 获取 Request 对象
 */
private static HttpServletRequest getRequest() {
    return SERVLET_HELPER_HOLDER.get().request;
}

/**
 * 获取 Response 对象
 */
private static HttpServletResponse getResponse() {
    return SERVLET_HELPER_HOLDER.get().response;
}

/**
 * 获取 Session 对象
```

```

    */
    private static HttpSession getSession() {
        return getRequest().getSession();
    }

    /**
     * 获取 ServletContext 对象
     */
    private static ServletContext getServletContext() {
        return getRequest().getServletContext();
    }
}

```

最重要的就是 `init` 与 `destroy` 方法，我们需要在恰当的地方调用它们，哪里是最恰当的地方呢？当然是上面提到的 `DispatcherServlet` 的 `service` 方法了。此外还提供了一系列私有的 `getter` 方法，因为我们下面需要封装几个常用的 `Servlet API` 工具方法：

```

...
    /**
     * 将属性放入 Request 中
     */
    public static void setRequestAttribute(String key, Object value) {
        getRequest().setAttribute(key, value);
    }

    /**
     * 从 Request 中获取属性
     */
    @SuppressWarnings("unchecked")
    public static <T> T getRequestAttribute(String key) {
        return (T) getRequest().getAttribute(key);
    }

    /**
     * 从 Request 中移除属性
     */
    public static void removeRequestAttribute(String key) {
        getRequest().removeAttribute(key);
    }
}

```

```
/**
 * 发送重定向响应
 */
public static void sendRedirect(String location) {
    try {
        getResponse().sendRedirect(getRequest().getContextPath() +
            location);
    } catch (IOException e) {
        LOGGER.error("redirect failure", e);
    }
}

/**
 * 将属性放入 Session 中
 */
public static void setSessionAttribute(String key, Object value) {
    getSession().setAttribute(key, value);
}

/**
 * 从 Session 中获取属性
 */
@SuppressWarnings("unchecked")
public static <T> T getSessionAttribute(String key) {
    return (T) getRequest().getSession().getAttribute(key);
}

/**
 * 从 Session 中移除属性
 */
public static void removeSessionAttribute(String key) {
    getRequest().getSession().removeAttribute(key);
}

/**
 * 使 Session 失效
 */
```

```

    public static void invalidateSession() {
        getRequest().getSession().invalidate();
    }
    ...

```

当然以上这些工具方法都是可以扩展的，要是我们认为比较常用的都可以封装起来。

现在 `ServletHelper` 已经开发完毕，是时候将其整合到 `DispatcherServlet` 中并初始化 `Request` 与 `Response` 对象了，实际上就是调用 `init` 与 `destroy` 方法，怎么调用呢？

我们需要借助一个 `try...finally...` 结构来实现。

```

public class DispatcherServlet extends HttpServlet {

    @Override
    public void service(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        ServletHelper.init(request, response);
        try {
            ...
        } finally {
            ServletHelper.destroy();
        }
    }
}

```

就是这么简单，现在就可以在 `Controller` 类中随时调用 `ServletHelper` 封装的 `Servlet API` 了；而且不仅仅可以在 `Controller` 类中调用，实际上在 `Service` 类中也可以调用。为什么呢？

因为所有的调用都来自于同一个请求线程。`DispatcherServlet` 是请求线程的入口，随后请求线程会先后来到 `Controller` 与 `Service` 中，我们只需使用 `ThreadLocal` 来确保 `ServletHelper` 对象中的 `Request` 与 `Response` 对象线程安全即可。

5.4 安全控制框架——Shiro

5.4.1 什么是 Shiro

前几天我遇见了一位美女，有种相逢恨晚的感觉。她皮肤白皙、气质优雅、楚楚动人，拥有苗条的身材，却又不失丰满之躯，她就是我朝思梦想的女神：Apache 组织下的名媛——Shiro

(希罗), 她是一款轻量级 Java 安全框架。

如果已经玩腻了丰满的 Spring Security, 想换换口味的話, 建议先与她约个会吧。

Apache Shiro 官网: <http://shiro.apache.org/>。

从官网上, 我们基本上可以了解到, 她提供的服务非常明确:

- (1) Authentication (认证)。
- (2) Authorization (授权)。
- (3) Session Management (会话管理)。
- (4) Cryptography (加密)。

首先, 她提供了 Authentication (认证) 服务, 也就是说, 通过她可以完成身份认证, 让她去判断用户是否为真实的会员。

其次, 她还提供了 Authorization (授权) 服务, 其实说白了就是“访问控制”服务, 也就是让她来识别用户是否可以做什么事情, 毕竟不同的用户是拥有不同的权限的。

更有特色的是, 她还提供了 Session Management (会话管理) 服务。这个就厉害了, 这并不是用户熟知的 HTTP Session, 而是一个独立的 Session 管理框架, 不管是否为 Web 应用, 都可以用这套框架。

最后(但并非最不重要), 她还提供了 Cryptography (加密) 服务, 封装了许多密码学算法, 琳琅满目, 应有尽有。

除了以上 4 个基本服务, 她也提供了很好的系统集成方案, 用户可以轻松将其运用到 Web 应用中, 可能这也是用户最关心的; 此外, 还可以集成第三方框架, 例如: Spring、Guice、CAS 等。

想必您已经了解了 Shiro 的主要功能, 那么如何才能真正拥有她呢? 不妨先主动跟她打声招呼吧: Hello Shiro!

5.4.2 Hello Shiro

如果您也使用 Maven 的话, 可以将以下配置复制到 pom.xml 文件中:

```
<!-- SLFJ -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.4</version>
</dependency>
```

```
<!-- Shiro -->
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-core</artifactId>
  <version>1.2.3</version>
</dependency>
```

需要说明的是，Shiro 依赖于 SLFJ 日志框架，而 SLFJ 只是一个接口，并没有提供具体的实现。您可以选择 Log4J 作为它的实现，正好 SLFJ 也提供了一个 slf4j-log4j12 的 Artifact，所以这里就可以用上了。

图 5.1 是 hello 项目的 Maven 依赖示意图。

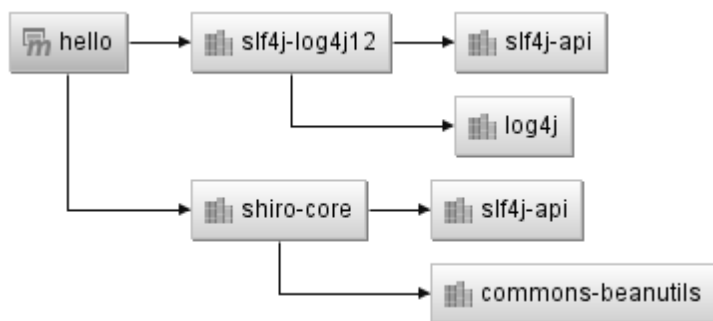


图 5-1 Shrio Maven 依赖

既然使用了 Log4J，那么就应该在 classpath 下提供一个 log4j.properties 文件：

```
log4j.rootLogger=INFO, console

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %c(%L) - %m%n
```

通过上面的配置将日志输出到控制台上，并配置了日志输出格式。

同样，既然使用了 Shiro，那么就应该在 classpath 下提供一个 shiro.ini 文件：

```
[users]
shiro = 201314
```

我们配置了一个用户名为 shiro，密码为 201314（爱你一生一世）的用户。当然，这里仅为演示，在实际项目中肯定不会把用户信息定义在配置文件中，除非这个项目的用户只有用户自己。

我们就用这个用户来见识一下 Shiro 的认证服务功能，下面写一个 main 方法试试：

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.AuthenticationException;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloShiro {

    private static final Logger logger = LoggerFactory.getLogger
        (HelloShiro.class);

    public static void main(String[] args) {
        // 初始化 SecurityManager
        Factory<SecurityManager> factory = new IniSecurityManagerFactory
            ("classpath:shiro.ini");
        SecurityManager securityManager = factory.getInstance();
        SecurityUtils.setSecurityManager(securityManager);

        // 获取当前用户
        Subject subject = SecurityUtils.getSubject();

        // 登录
        UsernamePasswordToken token = new UsernamePasswordToken("shiro",
            "201314");
        try {
            subject.login(token);
        } catch (AuthenticationException ae) {
            logger.info("登录失败！");
            return;
        }
        logger.info("登录成功！Hello " + subject.getPrincipal());
    }
}
```

```
// 注销
subject.logout();
}
}
```

我们分析一下这个 HelloShiro:

(1) 需要读取 classpath 下的 shiro.ini 配置文件, 并通过工厂类创建 SecurityManager 对象, 最终将其放入 SecurityUtils 中, 供 Shiro 框架随时获取。

(2) 同样通过 SecurityUtils 类获取 Subject 对象, 其实就是当前用户, 只不过在 Shiro 的世界里优雅地将其称为 Subject (主体)。

(3) 首先使用一个 Username 与 Password 来创建一个 UsernamePasswordToken 对象, 然后通过这个 Token 对象调用 Subject 对象的 login 方法, 让 Shiro 进行用户身份认证。

(4) 当登录失败时, 可以使用 AuthenticationException 来捕获这个异常; 当登录成功时, 可以调用 Subject 对象的 getPrincipal 方法来获取 Username, 此时 Shiro 已经创建了一个 Session。

(5) 最后还是通过 Subject 对象的 logout 方法来注销本次 Session。

只需要知道以上几个 Shiro 的核心成员的基本用法, Shiro 就是您的了。

其实, Shiro 的调用流程也不难理解, 如图 5-2 所示。

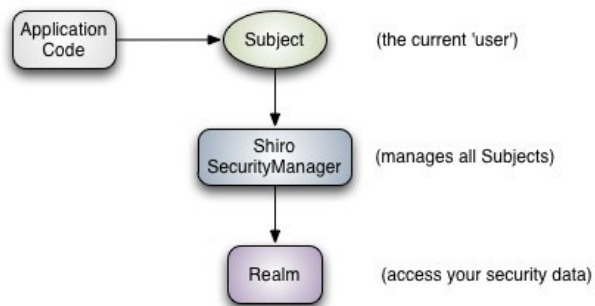


图 5-2 Shiro 的调用流程

通过 Subject 调用 SecurityManager, 通过 SecurityManager 调用 Realm。这个 Realm 感觉有点生僻, 其实就是提供用户信息的数据源。上面的例子在 shiro.ini 中配置的用户信息就是一种 Realm, 在 Shiro 中叫 IniRealm。除此以外, Shiro 还提供了其他几种 Realm: PropertiesRealm、JdbcRealm、JndiLdapRealm、ActiveDirectoryReam 等; 当然也可以定制 Realm 来满足业务需求。

不难发现, SecurityManager 才是 Shiro 的真正的核心, 只需通过 Subject 就可以操作 SecurityManager, 尤其是在 Web 应用中, 甚至都可以忘记 SecurityManager 的存在。

那么, 在 Web 中应该如何使用 Shiro 呢? 我们下节继续讨论。

5.4.3 在 Web 开发中使用 Shiro

我们可以直接在 Web 应用中使用 Shiro 官方提供的 Web 模块——shiro-web。

只需在 pom.xml 文件中增加如下配置：

```
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-web</artifactId>
  <version>1.2.3</version>
</dependency>
```

然后，在 web.xml 中添加一个 Listener 与一个 Filter：

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

  <listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoader-
      Listener</listener-class>
  </listener>

  <filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-
      class>
  </filter>

  <filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

实际上就是通过 EnvironmentLoaderListener 这个监听器来初始化 SecurityManager，并通过

ShiroFilter 来完成认证与授权。

可以使用以下数据表结构来存放用户及其权限的相关数据，这就是 RBAC 模型，如图 5-3 所示。

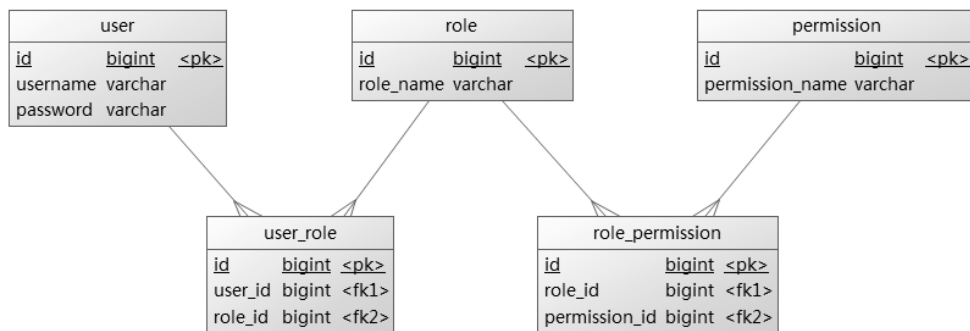


图 5-3 RBAC 模型

然后通过 Shiro 的 JdbcRealm 来进行认证与授权，只需在 shiro.ini 中做如下配置：

```

[main]
authc.loginUrl=/login

ds=org.apache.commons.dbcp.BasicDataSource
ds.driverClassName=com.mysql.jdbc.Driver
ds.url=jdbc:mysql://localhost:3306/sample
ds.username=root
ds.password=root

jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
jdbcRealm.dataSource=$ds
jdbcRealm.authenticationQuery=select password from user where username = ?
jdbcRealm.userRolesQuery=select r.role_name from user u, user_role ur, rol
e r where u.id = ur.user_id and r.id = ur.role_id and u.username=?
jdbcRealm.permissionsQuery=select p.permission_name from role r, role_
permission rp, permission p where r.id = rp.role_id and p.id=rp.permission_id
and r.role_name=?
jdbcRealm.permissionsLookupEnabled=true
securityManager.realms=$jdbcRealm

[urls]
  
```

```
/=anon
/space/**=authc
```

对以上配置解释如下：

首先，在[main]片段中，我们定义了一个“authc.loginUrl=/login”，用于配置当需要认证时需要跳转的 URL 地址，这里表示重定向到/login 请求，通过 Servlet 映射后可定位到 login 页面。

然后，定义了一个 DBCP 的 DataSource，用于获取 JDBC 数据库连接。

接着，定义 JdbcRealm 并指定 DataSource，通过配置以下几条 SQL 来完成认证与授权。

- authenticationQuery: 该 SQL 语句用于提供身份认证，即通过 username 查询 password。
- userRolesQuery: 该 SQL 语句用于提供基于角色的授权验证（属于粗粒度级别），即通过 username 查询 role_name。
- permissionQuery: 该 SQL 语句用于提供基于权限的授权验证（属于细粒度级别），即通过 role_name 查询 permission_name，此时需要开启 permissionsLookupEnabled 开关，默认是关闭的。

最后，在[urls]片段中，我们定义了一系列的 URL 过滤规则。Shiro 已经提供了一些默认的 Filter（过滤器），便于我们随时使用，当然也可以扩展其他过滤器。就本例配置而言，解释如下：

- /=anon——对于“/”请求（首页）可以匿名访问；
- /space/**=authc——对于以“/space/”开头的请求，均由 authc 过滤器处理，也就是完成身份认证操作。

还需要补充说明的是，在 permission 表中存放了所有的权限名，实际上是一个权限字符串，推荐使用“资源：操作”这种格式来命名，例如：product:view（查看产品权限）、product:edit（产品编辑权限）、product:delete（产品删除权限）等。

这些默认的过滤器如表 5-1 所示。

表 5-1 默认的过滤器

过滤器名称	功 能	配置项（及其默认值）
anon	确保只有未登录（匿名）的用户发送的请求才能通过	—
authc	确保只有已认证的用户发送的请求才能通过（若未认证，则跳转到登录页面）	authc.loginUrl = /login.jsp authc.successUrl = / authc.usernameParam = username authc.passwordParam = password authc.rememberMeParam = rememberMe authc.failureKeyAttribute = shiroLoginFailure

续表

过滤器名称	功 能	配置项（及其默认值）
authcBasic	提供 BasicHTTP 认证功能（在浏览器中弹出一个登录对话框）	authcBasic.applicationName=application
logout	接收结束会话的请求	logout.redirectUrl=/
noSessionCreation	提供 No Session 解决方案（若有 Session 就会报错）	—
perms	确保只有拥有特定权限的用户发送的请求才能通过	—
port	确保只有特定端口的请求才能通过	port=80
rest	提供 REST 解决方案（根据 REST URL 计算权限字符串）	—
roles	确保只有拥有特定角色的用户发送的请求才能通过	—
ssl	确保只有 HTTPS 的请求才能通过	—
user	确保只有已登录的用户发送的请求才能通过（包括：已认证或已记住）	—

可以在 index.jsp（首页）中判断该用户是游客还是已登录的用户：

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
<html>
<head>
    <title>首页</title>
</head>
<body>

<h1>首页</h1>

<shiro:guest>
    <p>身份：游客</p>
    <a href="<c:url value="/login"/>">登录</a>
    <a href="<c:url value="/register"/>">注册</a>
```

```
</shiro:guest>

<shiro:user>
  <p>身份: <shiro:principal/></p>
  <a href="<c:url value="/space"/>">空间</a>
  <a href="<c:url value="/logout"/>">退出</a>
</shiro:user>

</body>
</html>
```

需要使用 Shiro 提供的 JSP 标签:

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
```

随后就可以使用 shiro 标签的相关功能了, 如表 5-2 所示。

表 5-2 shiro 标签的相关功能

标 签	功 能
<shiro:guest>...</shiro:guest>	判断当前用户是否为游客
<shiro:user>...</shiro:user>	判断当前用户是否已登录（包括：已认证或已记住）
<shiro:authenticated>...</shiro:authenticated>	判断当前用户是否已认证通过（不包括已记住）
<shiro:notAuthenticated>...</shiro:notAuthenticated>	判断当前用户是否未认证通过
<shiro:principal />	获取当前用户的相关信息，例如：用户名
<shiro:hasRole name="foo">...</shiro:hasRole>	判断当前用户是否具有某种角色
<shiro:lacksRole name="foo">...</shiro:lacksRole>	判读当前用户是否缺少某种角色
<shiro:hasAnyRoles name="foo, bar">...</shiro:has-AnyRoles>	判断当前用户是否具有任意一种角色（foo 或 bar）
<shiro:hasPermission name="foo">...</shiro:hasPermission>	判断当前用户是否具有某种权限
<shiro:lacksPermission name="foo">...</shiro:lacksPermission>	判断当前用户是否缺少某种权限

如果 Shiro 再提供如下几个 JSP 标签那就完美了, 如表 5-3 所示。

表 5-3 JSP 标签

标 签	功 能
<shiro:hasAllRoles name="foo, bar">...</shiro:hasAllRoles>	判断当前用户是否同时具有每种角色（foo 与 bar）
<shiro:hasAnyPermission name="foo, bar">...</shiro:hasAnyPermission>	判断当前用户是否具有任意一种权限（foo 或 bar）
<shiro:hasAllPermission name="foo, bar">...</shiro:hasAllPermission>	判断当前用户是否同时具有每种权限（foo 与 bar）

实现以上这些标签并不是一件困难的事情，Shiro 最迷人的地方就是扩展了；有时候我们需要看源码并“依葫芦画瓢”，相信这一定是一件非常有趣的事情。

除了 JSP 标签，Shiro 还提供了 Java 注解，只需将这些注解定义在想要安全控制的方法上即可，如表 5-4 所示。

表 5-4 Java 注解

注 解	功 能
RequiresGuest	确保被标注的方法可被匿名用户访问
RequiresUser	确保被标注的方法只能被已登录的用户访问（包括：已认证或已记住）
RequiresAuthentication	确保被标注的方法只能被已认证的用户访问（不包括已记住）
RequiresRoles	确保被标注的方法仅被指定角色的用户访问
RequiresPermissions	确保被标注的方法仅被指定权限的用户访问

注意：当使用了 RequiresRoles 与 RequiresPermissions 注解，也就意味着把代码写死了；这样如果数据库里的 Role 或 Permission 更改了，代码也就无效了。这或许是 Shiro 的一点不完美的地方，不过瑕不掩瑜。

每次认证与授权都需要与数据库打交道，这会对性能产生一定的开销；关于这一点 Shiro 也为我们想到了，只需在[main]片段中增加如下配置即可：

```
cacheManager=org.apache.shiro.cache.MemoryConstrainedCacheManager
securityManager.cacheManager=$cacheManager
```

此时 Shiro 就会在内存中使用一个 Map 来缓存查询结果，从而减少了数据库的操作次数，提高了查询方面的性能。Shiro 也提供了 EhCache 的扩展，为缓存提供了更加“高大上”的解决方案。

目前在数据库里保存的是明文的密码，这样不太安全，如何将其加密呢？Shiro 同样提供了非常优雅的解决方案，只需在[main]片段下增加如下配置即可：

```
passwordMatcher=org.apache.shiro.authc.credential.PasswordMatcher
jdbcRealm.credentialsMatcher=$passwordMatcher
```

其实，Shiro 对密码的加密与解密提供了非常强大的支持，这里仅仅是一种最简单的情况。需要确保在创建密码的时候使用对应的加密算法，Shiro 给我们提供了 PasswordService 接口，可以这样来使用：

```
PasswordService passwordService = new DefaultPasswordService();
String encryptedPassword = passwordService.encryptPassword(plaintext-
Password);
```

只需将这个 encryptedPassword 存入数据库即可。

其实关于 Shiro 的那点事儿还有很多，不可能通过一节就能完全覆盖所有内容。我们可以从 Shiro 的官网上学到更多的特性，包括集成 EhCache、集成 Spring、集成 CAS 等。

5.5 提供安全控制特性

5.5.1 为什么需要安全控制

对于绝大多数应用系统而言，安全控制特性是绝对有必要的，比如以下需求：

(1) 我们需要提供一个登录界面，请用户进行身份认证，只有认证通过的用户才能访问应用系统。

(2) 如果未认证的用户尝试访问需要受保护的页面，那么系统就会将该请求重定向到登录界面。

(3) 当认证通过后，再次访问登录界面，将不会出现登录表单。

(4) 不同角色的用户可访问不同的操作，这就是所谓的“操作权限”特性。

我们已经学习了 Shiro 安全控制框架，并掌握了它的基本特性，接下来要做的事情就是如何应用它。也就是说，如何将 Shiro 整合到我们的 Smart 框架中。

这里说到的“整合”，不是简单地让开发者直接使用 Shiro 框架；而是将 Shiro 框架进行一些简单的封装，尽可能地屏蔽 Shiro 的存在。也就是说，提供更高级别的 API 与配置，目的是让开发者使用起来更加简单，使开发效率更加高效。

实际上安全控制相对于整个框架而言，是相当独立的，最好能够将其做成 Plugin（插件）

的形式，让开发者自由选择是否使用该插件。因此，将该插件命名为 **Smart Security Plugin**，它需要依赖 **Smart Framework** 与 **Shiro**。

下面我们就从开发者的角度，看看这款 **Smart Security** 插件是怎样使用的。然后再一起动手实现它。

5.5.2 如何使用安全控制框架

我们将要实现的安全控制框架的基本原则是：配置简单且使用方便。在配置上越少越好，在代码里只需要实现若干接口，并使用一些帮助类的 **API** 就能实现安全控制需求。

我们先从配置文件开始讲解。

1. 提供安全控制配置文件

只需在 **classpath** 下提供一份 **smart-security.ini** 配置文件，并在其中描述登录请求路径是什么，哪些请求路径可以匿名访问（也就是无须身份认证就能访问），哪些请求路径必须通过身份认证才能访问即可。配置看起来是这样的：

```
[main]
authc.loginUrl = /login

[urls]
/ = anon
/login = anon
/customer/** = authc
```

下面简单说明一下：

(1) 在 **main** 块中提供认证的相关属性，比如这里提供的登录 **URL** 地址是 **/login**。

(2) 在 **urls** 块中提供对具体 **URL** 的认证行为，对于匿名访问可使用 **anon**，对于认证访问可使用 **authc**。此外，在 **URL** 中还支持*通配符，表示匹配所有字符。

其实这个 **ini** 文件就是 **Shiro** 所需的 **shiro.ini** 配置文件，只是我们故意不让开发者知道底层是使用 **Shiro** 实现的，所以将文件名重命名为 **smart-security.ini** 了。这样做也就意味着我们需要定制 **Shiro** 框架，让它来识别 **smart-security.ini** 配置文件，而不是 **shiro.ini** 配置文件。我们后面就要一起动手实现这个特性。

除了配置，我们还需要实现安全控制框架提供的相关接口，并将我们的接口实现类配置在 **smart.properties** 文件中，因为安全控制框架需要读取这些配置项。

我们需要提供怎样的安全控制接口呢？

2. 实现安全控制接口

我们提供了一个名为 `SmartSecurity` 的接口，需要开发者实现三个方法：

- 根据用户名获取密码，在认证时需要调用。
- 根据用户名获取角色名集合，在授权时需要调用。
- 根据角色名获取操作名集合，在授权时需要调用。

以上三个方法包括了认证与授权两个方面，细节都在实现类中完成。此外还提供了第二种方式，如果开发者不想实现 `SmartSecurity` 接口，那么还可以在 `smart.properties` 文件中提供以下三个配置项：

- `smart.plugin.security.jdbc.authc_query`——根据用户名获取密码；
- `smart.plugin.security.jdbc.roles_query`——根据用户名获取角色名集合；
- `smart.plugin.security.jdbc.permissions_query`——根据角色名获取操作名集合。

以下是 `SmartSecurity` 接口的完整代码：

```
package org.smart4j.plugin.security;

/**
 * Smart Security 接口
 * <br/>
 * 可在应用中实现该接口，或者在 smart.properties 文件中提供以下基于 SQL 的配置项：
 * <ul>
 *   <li>smart.plugin.security.jdbc.authc_query: 根据用户名获取密码</li>
 *   <li>smart.plugin.security.jdbc.roles_query: 根据用户名获取角色名集合</li>
 *   <li>smart.plugin.security.jdbc.permissions_query: 根据角色名获取权限名集合</li>
 * </ul>
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface SmartSecurity {

    /**
     * 根据用户名获取密码
     *
     */
}
```

```

    * @param username 用户名
    * @return 密码
    */
    String getPassword(String username);

    /**
     * 根据用户名获取角色名集合
     *
     * @param username 用户名
     * @return 角色名集合
     */
    Set<String> getRoleNameSet(String username);

    /**
     * 根据角色名获取权限名集合
     *
     * @param roleName 角色名
     * @return 权限名集合
     */
    Set<String> getPermissionNameSet(String roleName);
}

```

紧接着，我们需要在应用中提供一个 **SmartSecurity** 的实现类，让它去完成相应的数据库操作，见如下 **AppSecurity.java** 代码：

```

package org.smart4j.chapter5;

import java.util.Set;
import org.smart4j.framework.helper.DatabaseHelper;
import org.smart4j.plugin.security.SmartSecurity;

/**
 * 应用安全控制
 *
 * @author huangyong
 * @since 1.0.0
 */
public class AppSecurity implements SmartSecurity {

```

```

    public String getPassword(String username) {
        String sql = "SELECT password FROM user WHERE username = ?";
        return DatabaseHelper.query(sql, username);
    }

    public Set<String> getRoleNameSet(String username) {
        String sql = "SELECT r.role_name FROM user u, user_role ur, role r
        WHERE u.id = ur.user_id AND r.id = ur.role_id AND u.username = ?";
        return DatabaseHelper.querySet(sql, username);
    }

    public Set<String> getPermissionNameSet(String roleName) {
        String sql = "SELECT p.permission_name FROM role r, role_permission
        rp, permission p WHERE r.id = rp.role_id AND p.id = rp.permission_id
        AND r.role_name = ?";
        return DatabaseHelper.querySet(sql, roleName);
    }
}

```

总结一下，我们在 `smart.properties` 文件中有两种配置方法：

- (1) 实现 `SmartSecurity` 接口，并在 `smart.properties` 文件中指定该接口的实现类。
- (2) 直接在 `smart.properties` 文件中提供相关 SQL 配置项，此时无须实现 `SmartSecurity` 接口。

对于第一种方式，我们可以这样配置：

```

smart.plugin.security.realms=custom
smart.plugin.security.custom.class="org.smart4j.chapter5.AppSecurity"

```

对于第二种方式，我们可以这样配置：

```

smart.plugin.security.realms=jdbc
smart.plugin.security.jdbc.authc_query="SELECT password FROM user WHERE
username = ?"
smart.plugin.security.jdbc.roles_query="SELECT r.role_name FROM user u,
user_role ur, role r WHERE u.id = ur.user_id AND r.id = ur.role_id AND
u.username = ?"
smart.plugin.security.jdbc.permissions_query="SELECT p.permission_name
FROM role r, role_permission rp, permission p WHERE r.id = rp.role_id AND
p.id = rp.permission_id AND r.role_name = ?"

```

后面我们将分别使用以上两种方式，到底哪种方式更好，完全取决于开发者自己。
完成以上配置后，我们需要提供一些简单的应用场景来验证我们的安全控制框架。

3. 实现几个典型的应用场景

产品经理提出了这样的需求：

(1) 在首页中需要提示用户的身份，如未登录，则显示游客，并提供“登录”按钮；若已登录，则显示该用户的用户名。

(2) 已登录的用户可查看系统的功能菜单，比如客户管理、订单管理等，并提供“注销”按钮。

(3) 当登录成功后，自动跳转到客户管理主页面；当登录失败后，需停留在登录页面。

(4) 对于未登录的用户，若在浏览器地址栏中输入需登录才能访问的页面地址，则自动跳转到登录页面。

(5) 对于已登录的用户，若再次访问登录页面，则自动跳转到客户管理主界面，无须再次出现登录表单。

从以上需求中，我们可以了解到，在首页与登录页面上都需要进行安全控制，用户所看到的界面或者页面跳转行为，完全取决于该用户是否登录。

因此，我们需要分别在首页与登录进行相应的安全控制，那么问题来了，用什么技术来实现这个需求呢？

我们可以用 JSP 来写页面，要实现以上需求，最简单的方法就是使用 JSP 标签技术了，下面我们一起来完成这些需求。

4. 在 JSP 中使用安全控制标签

我们先来看看首页应该怎么写，见如下 index.jsp 代码：

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="security" uri="/security" %>

<c:set var="BASE" value="${pageContext.request.contextPath}"/>

<html>
<head>
    <title>首页</title>
</head>
<body>
```

```

<h1>首页</h1>

<security:guest>
  <p>身份: 游客</p>
  <a href="${BASE}/login">登录</a>
</security:guest>

<security:user>
  <p>身份: <security:principal/></p>
  <ul>
    <li><a href="${BASE}/customer">客户管理</a></li>
  </ul>
  <a href="<c:url value="/logout"/>">注销</a>
</security:user>

</body>
</html>

```

首先, 声明一个自定义 JSP 标签:

```
<%@ taglib prefix="security" uri="/security" %>
```

然后, 使用如下标签对 JSP 页面相应的部分进行控制:

- (1) **security:guest**——用于验证当前用户是否为游客, 所谓游客就是未登录的用户。
- (2) **security:user**——用于验证当前用户是否为已登录的用户, 实际上已登录的用户包括两种, 刚刚登录过的和上次登录并记住的, 下面会进一步描述。
- (3) **security:principal**——用于获取当事人信息, 默认获取当前用户的用户名。

实际上, 以上这些标签都是 Shiro 框架提供的, 在 Smart 框架中只是对这些标签进行了封装或重命名, 后面会进一步描述相关实现细节。

类似地, 我们还需要在登录页面上进行安全控制, 见如下 login.jsp 代码:

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="security" uri="/security" %>

<c:set var="BASE" value="${pageContext.request.contextPath}"/>

```

```
<html>
<head>
  <title>登录</title>
</head>
<body>

<h1>登录</h1>

<security:guest>
  <form action="${BASE}/login" method="post">
    <table>
      <tr>
        <td>用户名: </td>
        <td><input type="text" name="username"></td>
      </tr>
      <tr>
        <td>密码: </td>
        <td><input type="password" name="password"></td>
      </tr>
      <tr>
        <td colspan="2">
          <button type="submit">登录</button>
        </td>
      </tr>
    </table>
  </form>
</security:guest>

<security:user>
  <c:redirect url="${BASE}/"/>
</security:user>

</body>
</html>
```

若为匿名用户，则显示登录表单；若为登录用户，则重定向到应用首页，实际上最终会重定向到客户管理页面。

除了能在 JSP 里使用框架提供的标签，我们还可以在 Java 类中使用框架提供的 API 来简化

我们的开发过程，那么具体有哪些 API 呢？

5. 使用安全控制 API

我们通过一个助手类来提供这些 API，在代码中可通过静态方法来调用这些 API，将这个助手类命名为 `SecurityHelper`，代码如下：

```
package org.smart4j.plugin.security;

/**
 * Security 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class SecurityHelper {

    /**
     * 登录
     */
    public static void login(String username, String password) throws
        AuthcException {
        ...
    }

    /**
     * 注销
     */
    public static void logout() {
        ...
    }
}
```

`SecurityHelper` 目前只有两个方法，一个用于登录，另一个用于注销。在登录时需要开发者提供用户名和密码这两个参数，并且必须处理 `AuthcException` 受检异常，也就是说，开发者必须通过 `try...catch...` 来处理这个 `AuthcException`。当登录失败时，需要做什么事情。

`AuthcException` 的代码也非常简单，直接继承 `Exception` 父类并重写相应的方法即可，代码如下：

```
package org.smart4j.plugin.security.exception;

/**
 * 认证异常（当非法访问时抛出）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class AuthcException extends Exception {

    public AuthcException() {
        super();
    }

    public AuthcException(String message) {
        super(message);
    }

    public AuthcException(String message, Throwable cause) {
        super(message, cause);
    }

    public AuthcException(Throwable cause) {
        super(cause);
    }
}
```

以上 `AuthcException` 是用于非法访问时抛出的异常，当权限无效时，也就是当前用户无权限访问某个操作时，应该抛出什么异常呢？我们需要再定义一个异常，命名为 `AuthzException`，代码如下：

```
package org.smart4j.plugin.security.exception;

/**
 * 授权异常（当权限无效时抛出）
 *
 * @author huangyong
 * @since 1.0.0
```

```
*/
public class AuthzException extends RuntimeException {

    public AuthzException() {
        super();
    }

    public AuthzException(String message) {
        super(message);
    }

    public AuthzException(String message, Throwable cause) {
        super(message, cause);
    }

    public AuthzException(Throwable cause) {
        super(cause);
    }
}
```

关于授权问题，我们稍后会进行描述。现在我们只是在解决认证问题，下面看看 Controller 是如何编写的，SystemController.java 代码如下：

```
package org.smart4j.chapter5.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.smart4j.framework.annotation.Action;
import org.smart4j.framework.annotation.Controller;
import org.smart4j.framework.bean.Param;
import org.smart4j.framework.bean.View;
import org.smart4j.framework.fault.AuthcException;
import org.smart4j.plugin.security.SecurityHelper;

/**
 * 处理系统请求
 */
@Controller
```

```
public class SystemController {

    private static final Logger LOGGER = LoggerFactory.getLogger(System-
        Controller.class);

    /**
     * 进入首页界面
     */
    @Action("get:/")
    public View index() {
        return new View("index.jsp");
    }

    /**
     * 进入登录界面
     */
    @Action("get:/login")
    public View login() {
        return new View("login.jsp");
    }

    /**
     * 提交登录表单
     */
    @Action("post:/login")
    public View loginSubmit(Param param) {
        String username = param.getString("username");
        String password = param.getString("password");
        try {
            SecurityHelper.login(username, password);
        } catch (AuthcException e) {
            LOGGER.error("login failure", e);
            return new View("/login");
        }
        return new View("/customer");
    }

    /**
```

```
    * 提交注销请求
    */
    @Action("get:/logout")
    public View logout() {
        SecurityHelper.logout();
        return new View("/");
    }
}
```

在 Controller 中使用了 SecurityHelper 提供的 API 来实现登录与注销功能，将所有与安全控制相关的事情都交给了 SecurityHelper 来处理，在 Controller 中只是处理请求，并调用相关组件的 API，根据调用结果来返回具体的 View，这就是 MVC 架构的精髓所在。绝对不要试图通过 Controller 来完成所有的事情，需要让每个组件的责任单一，这就是所谓的“单一职责原则”，该原则在面向对象的语言中使用非常广泛，我们需要深刻理解它。

以上仅从用户的角度描述了怎样使用 Smart Security 插件来提供安全控制特性，下面我们将深入到实现细节中，通过封装并扩展 Shiro 框架来打造一款轻量级的安全控制框架。

5.5.3 如何实现安全控制框架

我们创建一个名为 smart-plugin-security 的项目，它独立于 smart-framework，接下来要做的第一件事情就是添加相关的 Maven 依赖。

1. 添加相关的 Maven 依赖

```
<!-- Servlet -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>

<!-- JSP -->
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.2</version>
    <scope>provided</scope>
```

```
</dependency>

<!-- Smart Framework -->
<dependency>
    <groupId>org.smart4j</groupId>
    <artifactId>smart-framework</artifactId>
    <version>1.0.0</version>
</dependency>

<!-- Shiro -->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-web</artifactId>
    <version>1.2.3</version>
</dependency>
```

该项目依赖于 Servlet、JSP、Smart Framework 与 Shiro 构件，仅此四个构件就足够了，这些构件还会依赖其他构件，因为 Maven 依赖是具有传递性的。

2. 初始化插件

我们知道，当 Servlet 容器启动时会自动加载 Web 应用的 web.xml 文件与 classpath 中的类。而且要使用 Shiro 框架，需要在 web.xml 文件中添加一下配置项：

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener
</listener-class>
</listener>

<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-
class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

通过 `EnvironmentLoaderListener` 读取 `classpath` 中的 `shiro.ini` 文件，并加载其中的相关配置到内存中，以便 `ShiroFilter` 可随时获取。当从客户端发送请求时，该请求将被 `ShiroFilter` 拦截，获取请求中的 URL 与 `shiro.ini` 文件中的相关配置进行比较。一般情况下，我们需要拦截所有的请求，因此需要通过 `/*` 进行拦截。

我们的框架需要封装 `Shiro` 相关细节，也包括配置文件，因此，我们有必要通过 `Servlet 3.0` 规范提供的 `Servlet` 注册特性来省略 `web.xml` 文件的相关配置，并且将默认读取 `shiro.ini` 文件改为读取 `smart-security.ini` 文件。

此外，我们还需要在 `Web` 应用初始化的时候完成以上操作，其实 `Servlet` 框架已经为我们提供了支持，只需实现 `javax.servlet.ServletContainerInitializer` 接口，并在 `classpath` 下的 `META-INF/services/javax.servlet.ServletContainerInitializer` 文件中添加需要初始化的类即可。

我们提供一个名为 `SmartSecurityPlugin` 的类，让它去实现 `javax.servlet.ServletContainerInitializer` 接口。

然后需要在 `classpath` 下的 `META-INF/services/javax.servlet.ServletContainerInitializer` 文件中添加一行代码：

```
org.smart4j.plugin.security.SmartSecurityPlugin
```

必须通过以上配置，`Servlet` 容器才能读取 `jar` 包中的 `META-INF/services/javax.servlet.ServletContainerInitializer` 文件，并加载其中的 `SmartSecurityPlugin` 插件类。

那么，`SmartSecurityPlugin` 应该如何实现呢？具体代码如下：

```
package org.smart4j.plugin.security;

import java.util.Set;
import javax.servlet.FilterRegistration;
import javax.servlet.ServletContainerInitializer;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import org.apache.shiro.web.env.EnvironmentLoaderListener;

/**
 * Smart Security 插件
 *
 * @author huangyong
 * @since 1.0.0
 */
```

```

public class SmartSecurityPlugin implements ServletContainerInitializer {

    public void onStartup(Set<Class<?>> handlesTypes, ServletContext
        servletContext) throws ServletException {
        // 设置初始化参数
        servletContext.setInitParameter("shiroConfigLocations", "class-
            path:smart-security.ini");
        // 注册 Listener
        servletContext.addListener(EnvironmentLoaderListener.class);
        // 注册 Filter
        FilterRegistration.Dynamic smartSecurityFilter = servletContext.
            addFilter("SmartSecurityFilter", SmartSecurityFilter.class);
        smartSecurityFilter.addMappingForUrlPatterns(null, false, "/*");
    }
}

```

我们可通过 Shiro 提供的初始化参数来定制默认的 Shiro 配置文件名，通过 `ServletContext` 注册 `Listener` 与 `Filter`，只是这里注册的不是 `ShiroFilter`，而是 `SmartSecurityFilter`，它是前者的扩展。

那么，`SmartSecurityFilter` 究竟提供了哪些扩展呢？我们还是通过代码来解释：

```

package org.smart4j.plugin.security;

import java.util.LinkedHashSet;
import java.util.Set;
import org.apache.shiro.cache.CacheManager;
import org.apache.shiro.cache.MemoryConstrainedCacheManager;
import org.apache.shiro.mgt.CachingSecurityManager;
import org.apache.shiro.mgt.RealmSecurityManager;
import org.apache.shiro.realm.Realm;
import org.apache.shiro.web.mgt.WebSecurityManager;
import org.apache.shiro.web.servlet.ShiroFilter;
import org.smart4j.plugin.security.realm.SmartCustomRealm;
import org.smart4j.plugin.security.realm.SmartJdbcRealm;

/**
 * 安全过滤器
 */

```

```

    * @author huangyong
    * @since 1.0.0
    */
    public class SmartSecurityFilter extends ShiroFilter {

        @Override
        public void init() throws Exception {
            super.init();
            WebSecurityManager webSecurityManager = super.getSecurityManager();
            // 设置 Realm, 可同时支持多个 Realm, 并按照先后顺序用逗号分割
            setRealms(webSecurityManager);
            // 设置 Cache, 用于减少数据库查询次数, 降低 I/O 访问
            setCache(webSecurityManager);
        }

        private void setRealms(WebSecurityManager webSecurityManager) {
            // 读取 smart.plugin.security.realms 配置项
            String securityRealms = SecurityConfig.getRealms();
            if (securityRealms != null) {
                // 根据逗号进行拆分
                String[] securityRealmArray = securityRealms.split(",");
                if (securityRealmArray.length > 0) {
                    // 使 Realm 具备唯一性与顺序性
                    Set<Realm> realms = new LinkedHashSet<Realm>();
                    for (String securityRealm : securityRealmArray) {
                        if (securityRealm.equalsIgnoreCase(SecurityConstant.
                            REALMS_JDBC)) {
                            // 添加基于 JDBC 的 Realm, 需配置相关 SQL 查询语句
                            addJdbcRealm(realms);
                        } else if (securityRealm.equalsIgnoreCase(Security-
                            Constant.REALMS_CUSTOM)) {
                            // 添加基于定制化的 Realm, 需实现 SmartSecurity 接口
                            addCustomRealm(realms);
                        }
                    }
                }
                RealmSecurityManager realmSecurityManager = (RealmSecurity-
                    Manager) webSecurityManager;
                realmSecurityManager.setRealms(realms); // 设置 Realm
            }
        }
    }

```

```

        }
    }
}

private void addJdbcRealm(Set<Realm> realms) {
    // 添加自己实现的基于 JDBC 的 Ream
    SmartJdbcRealm smartJdbcRealm = new SmartJdbcRealm();
    realms.add(smartJdbcRealm);
}

private void addCustomRealm(Set<Realm> realms) {
    // 读取 smart.plugin.security.custom.class 配置项
    SmartSecurity smartSecurity = SecurityConfig.getSmartSecurity();
    // 添加自己实现的 Realm
    SmartCustomRealm smartCustomRealm = new SmartCustomRealm(smart-
        Security);
    realms.add(smartCustomRealm);
}

private void setCache(WebSecurityManager webSecurityManager) {
    // 读取 smart.plugin.security.cache 配置项
    if (SecurityConfig.isCache()) {
        CachingSecurityManager cachingSecurityManager = (CachingSecuri-
            tyManager) webSecurityManager;
        // 使用基于内存的 CacheManager
        CacheManager cacheManager = new MemoryConstrainedCacheManager();
        cachingSecurityManager.setCacheManager(cacheManager);
    }
}
}
}

```

以上 `SmartSecurityFilter` 是框架中最核心的对象，它继承了 `ShiroFilter`，具备父类的所有特性。有两个自定义的 `Realm`：`SmartJdbcRealm` 与 `SmartCustomRealm`，前者用于提供基于 SQL 配置的实现，后者用于提供更加灵活的 `SmartSecurity` 接口的实现，后面我们将分别介绍这两个 `Realm` 的具体实现。

3. 实现认证特性

下面我们对以上两种自定义 `Realm` 分别进行描述，我们先看比较简单的 `SmartJdbcRealm`：

```

package org.smart4j.plugin.security.realm;

import org.apache.shiro.realm.jdbc.JdbcRealm;
import org.smart4j.framework.helper.DatabaseHelper;
import org.smart4j.plugin.security.SecurityConfig;
import org.smart4j.plugin.security.password.Md5CredentialsMatcher;

/**
 * 基于 Smart 的 JDBC Realm (需要提供相关 smart.plugin.security.jdbc.* 配置项)
 *
 * @author huangyong
 * @since 1.0.0
 */
public class SmartJdbcRealm extends JdbcRealm {

    public SmartJdbcRealm() {
        super.setDataSource(DatabaseHelper.getDataSource());
        super.setAuthenticationQuery(SecurityConfig.getJdbcAuthcQuery());
        super.setUserRolesQuery(SecurityConfig.getJdbcRolesQuery());
        super.setPermissionsQuery(SecurityConfig.getJdbcPermissionsQuery());
        super.setPermissionsLookupEnabled(true);
        super.setCredentialsMatcher(new Md5CredentialsMatcher());
        // 使用 MD5 加密算法
    }
}

```

我们直接对 Shiro 提供的 JdbcRealm 进行了扩展, 通过 Smart 框架提供的 DatabaseHelper 助手类来获取 DataSource, 通过 Smart Security 插件提供的 SecurityConfig 常量类来获取相关 JDBC 配置项。我们需要开启 PermissionsLookup 开关, 开启后可连接 permission 表进行查询。最后, 我们通过自定义的 Md5CredentialsMatcher 对象来提供基于 MD5 的密码匹配机制。

Md5CredentialsMatcher 的代码如下:

```

package org.smart4j.plugin.security.password;

import org.apache.shiro.authc.AuthenticationInfo;
import org.apache.shiro.authc.AuthenticationToken;
import org.apache.shiro.authc.UsernamePasswordToken;

```

```

import org.apache.shiro.authc.credential.CredentialsMatcher;
import org.smart4j.framework.util.CodecUtil;

/**
 * MD5 密码匹配器
 *
 * @author huangyong
 * @since 1.0.0
 */
public class Md5CredentialsMatcher implements CredentialsMatcher {

    public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
        // 获取从表单提交过来的密码、明文，尚未通过 MD5 加密
        String submitted = String.valueOf(((UsernamePasswordToken) token).getPassword());
        // 获取数据库中存储的密码，已通过 MD5 加密
        String encrypted = String.valueOf(info.getCredentials());
        return CodecUtil.md5(submitted).equals(encrypted);
    }
}

```

只需实现 Shiro 提供的 `CredentialsMatcher` 接口即可完成该接口提供的 `doCredentialsMatch` 方法，该方法提供了两个参数。

(1) `AuthenticationToken`: 可通过该参数获取从表单提交过来的密码，该密码是明文，尚未通过 MD5 加密。

(2) `AuthenticationInfo`: 可通过该参数获取数据库中存储的密码，该密码已通过 MD5 加密。

最后通过 Smart 框架提供的 `CodecUtil` 工具类对表单提交过来的密码进行 MD5 加密，并将加密后的结果与数据库中的 MD5 密码进行比较，将比较的结果作为方法的返回值。

`CodecUtil` 的 `md5` 方法代码如下：

```

package org.smart4j.framework.util;

import org.apache.commons.codec.digest.DigestUtils;

/**
 * 编码与解码操作工具类

```

```

    */
    public final class CodecUtil {
        ...
        /**
         * MD5 加密
         */
        public static String md5(String source) {
            return DigestUtils.md5Hex(source);
        }
    }
}

```

下面我们再看看 SmartCustomRealm 的实现细节，代码如下：

```

package org.smart4j.plugin.security.realm;

import java.util.HashSet;
import java.util.Set;
import org.apache.shiro.authc.AuthenticationException;
import org.apache.shiro.authc.AuthenticationInfo;
import org.apache.shiro.authc.AuthenticationToken;
import org.apache.shiro.authc.SimpleAuthenticationInfo;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.authz.AuthorizationException;
import org.apache.shiro.authz.AuthorizationInfo;
import org.apache.shiro.authz.SimpleAuthorizationInfo;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;
import org.apache.shiro.subject.SimplePrincipalCollection;
import org.smart4j.plugin.security.SecurityConstant;
import org.smart4j.plugin.security.SmartSecurity;
import org.smart4j.plugin.security.password.Md5CredentialsMatcher;

/**
 * 基于 Smart 的自定义 Realm（需要实现 SmartSecurity 接口）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class SmartCustomRealm extends AuthorizingRealm {

```

```
private final SmartSecurity smartSecurity;

public SmartCustomRealm(SmartSecurity smartSecurity) {
    this.smartSecurity = smartSecurity;
    super.setName(SecurityConstant.REALMS_CUSTOM);
    super.setCredentialsMatcher(new Md5CredentialsMatcher());
    // 使用 MD5 加密算法
}

@Override
public AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
    if (token == null) {
        throw new AuthenticationException("parameter token is null");
    }

    // 通过 AuthenticationToken 对象获取从表单中提交过来的用户名
    String username = ((UsernamePasswordToken) token).getUsername();

    // 通过 SmartSecurity 接口并根据用户名获取数据库中存放的密码
    String password = smartSecurity.getPassword(username);

    // 将用户名与密码放入 AuthenticationInfo 对象中, 便于后续的认证操作
    SimpleAuthenticationInfo authenticationInfo = new SimpleAuthen-
ticationInfo();
    authenticationInfo.setPrincipals(new SimplePrincipalCollection
(username, super.getName()));
    authenticationInfo.setCredentials(password);
    return authenticationInfo;
}

@Override
public AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
    if (principals == null) {
        throw new AuthorizationException("parameter principals is null");
    }
}
```

```

    }

    // 获取已认证用户的用户名
    String username = (String) super.getAvailablePrincipal(principals);

    // 通过 SmartSecurity 接口并根据用户名获取角色名集合
    Set<String> roleNameSet = smartSecurity.getRoleNameSet(username);

    // 通过 SmartSecurity 接口并根据角色名获取与其对应的权限名集合
    Set<String> permissionNameSet = new HashSet<String>();
    if (roleNameSet != null && roleNameSet.size() > 0) {
        for (String roleName : roleNameSet) {
            Set<String> currentPermissionNameSet = smartSecurity.
                getPermissionNameSet(roleName);
            permissionNameSet.addAll(currentPermissionNameSet);
        }
    }

    // 将角色名集合与权限名集合放入 AuthorizationInfo 对象中,便于后续的授权操作
    SimpleAuthorizationInfo authorizationInfo = new SimpleAutho-
        rizationInfo();
    authorizationInfo.setRoles(roleNameSet);
    authorizationInfo.setStringPermissions(permissionNameSet);
    return authorizationInfo;
}
}

```

与 SmartJdbcRealm 不一样的地方是, SmartCustomRealm 继承自 AuthorizingRealm 父类, 覆盖了父类的 doGetAuthenticationInfo 与 doGetAuthorizationInfo 方法, 前者用于认证, 后者用于授权。

在 SecurityConfig 中提供了一系列静态方法, 用于获取 smart.properties 文件中的配置项, 代码如下:

```

package org.smart4j.plugin.security;

import org.smart4j.framework.helper.ConfigHelper;
import org.smart4j.framework.util.ReflectionUtil;

```

```

/**
 * 从配置文件中获取相关属性
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class SecurityConfig {

    public static String getRealms() {
        return ConfigHelper.getString(SecurityConstant.REALMS);
    }

    public static SmartSecurity getSmartSecurity() {
        String className = ConfigHelper.getString(SecurityConstant.SMART_
            SECURITY);
        return (SmartSecurity) ReflectionUtil.newInstance(className);
    }

    public static String getJdbcAuthcQuery() {
        return ConfigHelper.getString(SecurityConstant.JDBC_AUTHC_QUERY);
    }

    public static String getJdbcRolesQuery() {
        return ConfigHelper.getString(SecurityConstant.JDBC_ROLES_QUERY);
    }

    public static String getJdbcPermissionsQuery() {
        return ConfigHelper.getString(SecurityConstant.JDBC_PERMISSIONS_
            QUERY);
    }

    public static boolean isCacheable() {
        return ConfigHelper.getBoolean(SecurityConstant.CACHEABLE);
    }
}

```

将所有的常量统一放入 `SecurityConstant` 常量类中，具体代码如下：

```
package org.smart4j.plugin.security;
```

```

/**
 * 常量接口
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface SecurityConstant {

    String REALMS = "smart.plugin.security.realms";
    String REALMS_JDBC = "jdbc";
    String REALMS_CUSTOM = "custom";

    String SMART_SECURITY = "smart.plugin.security.custom.class";

    String JDBC_AUTHC_QUERY = "smart.plugin.security.jdbc.authc_query";
    String JDBC_ROLES_QUERY = "smart.plugin.security.jdbc.roles_query";
    String JDBC_PERMISSIONS_QUERY = "smart.plugin.security.jdbc.
permissions_query";

    String CACHE = "smart.plugin.security.cache";
}

```

到目前为止，认证部分已基本实现完毕，接下来我们看看之前提到的 `SecurityHelper` 是如何实现的。

4. 实现 `SecurityHelper`

`SecurityHelper` 提供了登录与注销两个方法，这些方法可以在 `Controller` 中被调用，我们只需简单封装 `Shiro` 的 API 就能实现该功能，具体代码如下：

```

package org.smart4j.plugin.security;

import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.AuthenticationException;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.subject.Subject;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.smart4j.plugin.security.exception.AuthcException;

```

```
/**
 * Security 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public final class SecurityHelper {

    private static final Logger LOGGER = LoggerFactory.getLogger(Security-
        Helper.class);

    /**
     * 登录
     */
    public static void login(String username, String password) throws Authc-
        Exception {
        Subject currentUser = SecurityUtils.getSubject();
        if (currentUser != null) {
            UsernamePasswordToken token = new UsernamePasswordToken(username,
                password);
            try {
                currentUser.login(token);
            } catch (AuthenticationException e) {
                LOGGER.error("login failure", e);
                throw new AuthcException(e);
            }
        }
    }

    /**
     * 注销
     */
    public static void logout() {
        Subject currentUser = SecurityUtils.getSubject();
        if (currentUser != null) {
            currentUser.logout();
        }
    }
}
```

```
    }
}
```

通过 Shiro 提供的 `SecurityUtils` 工具类，可随时获取当前登录的用户对象，在 Shiro 中被称为 `Subject`，我们通过调用 `Subject` 对象的 `login` 与 `logout` 方法分别完成登录与注销操作。可见，Shiro 已经帮我们屏蔽了许多技术细节，我们只需做简单的操作就能完全封装 Shiro 的 API。

下面我们再来看看如何通过 JSP 标签来实现页面安全控制。

5. 通过 JSP 标签实现页面安全控制

我们只需提供一个标签库定义文件（后缀为 `tld` 的文件），对 Shiro 提供的 JSP 标签类进行重新定义即可，以下是 `security.tld` 文件的代码片段：

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
  version="2.1">

  <description>Security JSP Tag</description>
  <tlib-version>1.1</tlib-version>
  <short-name>security</short-name>
  <uri>/security</uri>

  <tag>
    <description>判断当前用户是否已登录（包括：已认证与已记住）</description>
    <name>user</name>
    <tag-class>org.apache.shiro.web.tags.UserTag</tag-class>
    <body-content>JSP</body-content>
  </tag>

  <tag>
    <description>判断当前用户是否未登录（包括：未认证或未记住，即“访客”身份）
    </description>
    <name>guest</name>
    <tag-class>org.apache.shiro.web.tags.GuestTag</tag-class>
    <body-content>JSP</body-content>
```

```

    </tag>

    ...

</taglib>

```

需要将 `security.tld` 放入 `classpath` 的 `META-INF` 目录中，这样才能在 JSP 文件中定义如下标签库：

```
<%@ taglib prefix="security" uri="/security" %>
```

通过这样的封装，不仅可以屏蔽 Shiro 的底层实现，而且还能扩展 Shiro 缺乏的 JSP 标签，比如 Shiro 尚未提供的以下三个 JSP 标签。

(1) `security:hasAllRoles`：判断当前用户是否拥有其中所有的角色（逗号分隔，表示“与”的关系）

(2) `security:hasAnyPermissions`：判断当前用户是否拥有其中某一种权限（逗号分隔，表示“或”的关系）

(3) `security:hasAllPermissions`：判断当前用户是否拥有其中所有的权限（逗号分隔，表示“与”的关系）

以上 JSP 标签我们可在框架中自行实现，并定义在 `security.tld` 文件中，下面就以 `security:hasAllRoles` 为例进行描述。

首先，需要编写一个名为 `HasAllRolesTag` 的标签类：

```

package org.smart4j.plugin.security.tag;

import java.util.Arrays;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.web.tags.RoleTag;

/**
 * 判断当前用户是否拥有其中所有的角色（逗号分隔，表示“与”的关系）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class HasAllRolesTag extends RoleTag {

    private static final String ROLE_NAMES_DELIMITER = ",";

```

```

@Override
protected boolean showTagBody(String roleNames) {
    boolean hasAllRole = false;
    Subject subject = getSubject();
    if (subject != null) {
        if (subject.hasAllRoles(Arrays.asList(roleNames.split(ROLE_
            NAMES_DELIMITER)))) {
            hasAllRole = true;
        }
    }
    return hasAllRole;
}
}

```

然后，在 `security.tld` 文件中添加如下配置：

```

...
<tag>
    <description>判断当前用户是否拥有其中所有的角色（逗号分隔，表示“与”的关系）
    </description>
    <name>hasAllRoles</name>
    <tag-class>org.smart4j.plugin.security.tag.HasAllRolesTag</tag-
    class>
    <body-content>JSP</body-content>
    <attribute>
        <name>name</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
...

```

同理，您可以自行实现 `security:hasAnyPermissions` 与 `security:hasAllPermissions` 标签。

最后我们来看看如何通过自定义注解来实现授权特性。

6. 通过自定义注解实现授权特性

参考 Shiro 提供的 JSP 标签，我们可以提供以下注解：

- `User`——判断当前用户是否已登录（包括：已认证与已记住）；

- **Guest**——判断当前用户是否未登录（包括：未认证或未记住，即“访客”身份）；
- **Authenticated**——判断当前用户是否已认证；
- **HasRoles**——判断当前用户是否拥有某种角色；
- **HasPermissions**——判断当前用户是否拥有某种权限。

我们的目标是把这些注解定义在目标方法上，通过 AOP 的方式进行授权验证。

下面以 **User** 注解为例，描述如何实现授权特性。

首先，需要编写一个注解类：

```
package org.smart4j.plugin.security.annotation;

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

/**
 * 判断当前用户是否已登录（包括：已认证与已记住）
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface User {
}
```

然后，编写一个名为 **AuthzAnnotationAspect** 的切面类，实现前置增强机制，即覆盖 **AspectProxy** 抽象类的 **before** 方法。具体代码如下：

```
package org.smart4j.plugin.security.aspect;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.subject.PrincipalCollection;
import org.apache.shiro.subject.Subject;
import org.smart4j.framework.annotation.Aspect;
```

```

import org.smart4j.framework.annotation.Controller;
import org.smart4j.framework.proxy.AspectProxy;
import org.smart4j.plugin.security.annotation.User;
import org.smart4j.plugin.security.exception.AuthzException;

/**
 * 授权注解切面
 *
 * @author huangyong
 * @since 1.0.0
 */
@Aspect(Controller.class)
public class AuthzAnnotationAspect extends AspectProxy {

    /**
     * 定义一个基于授权功能的注解类数组
     */
    private static final Class[] ANNOTATION_CLASS_ARRAY = {
        User.class
    };

    @Override
    public void before(Class<?> cls, Method method, Object[] params) throws
        Throwable {
        // 从目标类与目标方法中获取相应的注解
        Annotation annotation = getAnnotation(cls, method);
        if (annotation != null) {
            // 判断授权注解的类型
            Class<?> annotationType = annotation.annotationType();
            if (annotationType.equals(User.class)) {
                handleUser();
            }
        }
    }

    @SuppressWarnings("unchecked")
    private Annotation getAnnotation(Class<?> cls, Method method) {
        // 遍历所有的授权注解

```

```

        for (Class<? extends Annotation> annotationClass : ANNOTATION_
            CLASS_ARRAY) {
            // 首先判断目标方法上是否带有授权注解
            if (method.isAnnotationPresent(annotationClass)) {
                return method.getAnnotation(annotationClass);
            }
            // 然后判断目标类上是否带有授权注解
            if (cls.isAnnotationPresent(annotationClass)) {
                return cls.getAnnotation(annotationClass);
            }
        }
        // 若目标方法与目标类上均未带有授权注解, 则返回空对象
        return null;
    }

    private void handleUser() {
        Subject currentUser = SecurityUtils.getSubject();
        PrincipalCollection principals = currentUser.getPrincipals();
        if (principals == null || principals.isEmpty()) {
            throw new AuthzException("当前用户尚未登录");
        }
    }
}

```

最后, 请自行修改 `AuthzAnnotationAspect` 类, 完成 `Guest`、`Authenticated`、`HasRoles` 与 `HasPermissions` 注解的相关实现。

至此, 一个简单的安全控制框架基本开发完毕, 但 `Shiro` 强大的功能不仅仅于此, 后期我们可以根据现实需求, 不断扩展 `Smart Security` 插件, 让它的功能更加强大。

5.6 Web 服务框架——CXF

5.6.1 什么是 CXF

CXF 是 Apache 旗下的一款非常优秀的 Web 服务 (WS) 开源框架, 具备轻量级的特性, 而且能无缝整合到 `Spring` 中, 只需简单配置与注解就能轻松使用它来发布 WS。

其实 CXF 是两个开源框架的整合, 它们分别是 `Celtix` 与 `XFire`, 前者是一款 ESB 框架, 后

者是一款 WS 框架。早在 2007 年 5 月，当 XFire 发展到了它的鼎盛时期（最终版本是 1.2.6），突然对业界宣布了一个令人震惊的消息：“XFire is now CXF”，随后 CXF 2.0 诞生了，直到 2014 年 5 月，CXF 3.0 降临了。真是 7 年磨一剑啊！CXF 终于长大了，相信在不久的将来，一定会取代 Java 界 WS 龙头老大 Axis 的江湖地位，貌似 Axis 自从 2012 年 4 月以后就没有升级了。

5.6.2 使用 CXF 开发 SOAP 服务

如何使用 CXF 开发基于 SOAP 的 WS 呢？

这就是本节重要的内容，重点是在 Web 容器中发布与调用 WS，这样也更加贴近我们实际工作的场景。

在 CXF 这个主角正式登台之前，先请出今天的配角 Oracle JAX-WS RI（本节简称 RI，即 Reference Implementation 的缩写），它是 Java 官方提供的 JAX-WS 规范的具体实现。

先让 RI 来跑跑龙套，看看如何使用 RI 发布 WS。

1. 使用 RI 发布 WS

第一步：整合 Tomcat 与 RI。

这一步稍微有一点点烦琐，不过也很容易做到。首先需要通过以下地址下载一份 RI 的程序包。

RI 下载地址：<https://jax-ws.java.net/2.2.8/>。

下载完毕后，只需解压即可，假设解压到 D:/Tool/jaxws-ri 目录下。随后需要对 Tomcat 的 config/catalina.properties 文件进行配置：

```
common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${catalina.home}/lib,${catalina.home}/lib/*.jar,D:/Tool/jaxws-ri/lib/*.jar
```

注意：以上配置中的最后一部分，其实就是在 Tomcat 中添加一系列关于 RI 的 jar 包。

看起来并不复杂，只是对现有的 Tomcat 有所改造而已，当然，将这些 jar 包全部放入自己应用的 WEB-INF/lib 目录中也是可行的。

第二步：编写 WS 接口及其实现。

接口部分：

```
package demo.ws.soap_jaxws;

import javax.xml.ws.WebService;
```

```

@WebService
public interface HelloService {

    String say(String name);
}
实现部分:
package demo.ws.soap_jaxws;

import javax.jws.WebService;

@WebService(
    serviceName = "HelloService",
    portName = "HelloServicePort",
    endpointInterface = "demo.ws.soap_jaxws.HelloService"
)
public class HelloServiceImpl implements HelloService {

    public String say(String name) {
        return "hello " + name;
    }
}

```

注意：接口与实现类上都标注 `javax.jws.WebService` 注解，可在实现类的注解中添加一些关于 WS 的相关信息，比如 `serviceName`、`portName` 等，当然这是可选的，只是为了让生成的 WSDL 的可读性更加强而已。

第三步：在 WEB-INF 下添加 `sun-jaxws.xml` 文件。

在 `sun-jaxws.xml` 文件里配置需要发布的 WS，其内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version=
"2.0">

    <endpoint name="HelloService"
        implementation="demo.ws.soap_jaxws.HelloServiceImpl"

```

```
url-pattern="/ws/soap/hello"/>

</endpoints>
```

这里仅发布一个 **endpoint**，并配置三个属性：**WS** 的名称、实现类、**URL** 模式，正是通过这个“**URL 模式**”来访问 **WSDL** 的。

第四步：部署应用并启动 Tomcat

当 **Tomcat** 启动成功后，会在控制台上看到如下信息：

```
2014-7-2 13:39:31 com.sun.xml.ws.transport.http.servlet.WSServletDelegate
<init>
信息：WSSERVLET14: JAX-WS servlet 正在初始化
2014-7-2 13:39:31 com.sun.xml.ws.transport.http.servlet.WSServlet-
ContextListener contextInitialized
信息：WSSERVLET12: JAX-WS 上下文监听程序正在初始化
```

随后打开浏览器，输入以下地址：

<http://localhost:8080/ws/soap/hello>

如果不出意外的话，现在应该可以看到如下界面了：

```
[RI 控制台][http://static.oschina.net/uploads/space/2014/0702/134909_
bimN_223750.png]
```

看起来这应该是一个 **WS** 控制台，方便我们查看发布了哪些 **WS**，单击上面的 **WSDL** 链接可查看具体信息。

看起来 **RI** 确实挺好的。不仅仅有一个控制台，而且还能与 **Tomcat** 无缝整合。但 **RI** 似乎与 **Spring** 的整合能力并不是太强，也许因为 **Oracle** 是 **EJB** 拥护者的原因吧。

那么，**CXF** 是否也具备 **RI** 这样的特性并且能够与 **Spring** 很好地集成呢？

CXF 不仅可以发布 **WS** 在任何的 **Web** 容器中，而且还提供了一个便于测试的 **Web** 环境，实际上它内置了一个 **Jetty**。

我们先看看如何启动 **Jetty** 来发布 **WS**，再演示如何在 **Spring** 容器中整合 **CXF**。

2. 使用 CXF 内置的 Jetty 来发布 WS

第一步：配置 **Maven** 依赖。

如果您是一位 **Maven** 用户，那么下面这段配置一定不会陌生：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>demo.ws</groupId>
  <artifactId>soap_cxf</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <cxf.version>3.0.4</cxf.version>
  </properties>

  <dependencies>
    <!-- CXF -->
    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-frontend-jaxws</artifactId>
      <version>${cxf.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-transports-http-jetty</artifactId>
      <version>${cxf.version}</version>
    </dependency>
  </dependencies>

</project>

```

如果还没有使用 **Maven**，那么就需要从以下地址下载 CXF 的相关 **jar** 包，并将其放入到应用中。

CXF 下载地址：<http://cxf.apache.org/download.html>。

第二步：写一个 **WS** 接口及其实现。

接口部分：

```
package demo.ws.soap_cxf;

import javax.jws.WebService;

@WebService
public interface HelloService {

    String say(String name);
}
```

实现部分:

```
package demo.ws.soap_cxf;

import javax.jws.WebService;

@WebService
public class HelloServiceImpl implements HelloService {

    public String say(String name) {
        return "hello " + name;
    }
}
```

这里简化了实现类上的 `WebService` 注解的配置, 让 CXF 自动为我们取默认值即可。

第三步: 写一个 `JaxWsServer` 类来发布 WS。

```
package demo.ws.soap_cxf;

import org.apache.cxf.jaxws.JaxWsServerFactoryBean;

public class JaxWsServer {

    public static void main(String[] args) {
        JaxWsServerFactoryBean factory = new JaxWsServerFactoryBean();
        factory.setAddress("http://localhost:8080/ws/soap/hello");
        factory.setServiceClass(HelloService.class);
        factory.setServiceBean(new HelloServiceImpl());
        factory.create();
    }
}
```

```
        System.out.println("soap ws is published");
    }
}
```

发布 WS 除了以上这种基于 JAX-WS 的方式以外, CXF 还提供了另一种选择——simple 方式。

通过 simple 方式发布 WS 的代码如下:

```
package demo.ws.soap_cxf;

import org.apache.cxf.frontend.ServerFactoryBean;

public class SimpleServer {

    public static void main(String[] args) {
        ServerFactoryBean factory = new ServerFactoryBean();
        factory.setAddress("http://localhost:8080/ws/soap/hello");
        factory.setServiceClass(HelloService.class);
        factory.setServiceBean(new HelloServiceImpl());
        factory.create();
        System.out.println("soap ws is published");
    }
}
```

注意: 以 simple 方式发布的 WS 不能通过 JAX-WS 方式来调用, 只能通过 simple 方式的客户端来调用, 下面会展示 simple 方式的客户端代码。

第四步: 运行 JaxWsServer 类。

当 JaxWsServer 启动后, 在控制台中会看到打印出来的一句提示。随后, 在浏览器中输入以下 WSDL 地址: `http://localhost:8080/ws/soap/hello?wsdl`。

注意: 通过 CXF 内置的 Jetty 发布的 WS, 仅能查看 WSDL, 却没有像 RI 那样的 WS 控制台。

可见这种方式非常容易测试与调试, 大大提高了我们的开发效率, 但这种方式并不适合于生产环境, 我们还是需要依靠 Tomcat 与 Spring。

那么，CXF 在实战中是如何集成在 Spring 容器中的呢？

3. 在 Web 容器中使用 Spring+CXF 发布 WS

Tomcat+Spring+CXF 这个场景应该更加接近我们的实际工作情况，开发过程也是非常自然的。

第一步：配置 Maven 依赖。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.ws</groupId>
    <artifactId>soap_spring_cxf</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <spring.version>4.0.5.RELEASE</spring.version>
        <cxf.version>3.0.4</cxf.version>
    </properties>

    <dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

```

    <!-- CXF -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>${cxf.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transport-http</artifactId>
        <version>${cxf.version}</version>
    </dependency>
</dependencies>

</project>

```

第二步：写一个 WS 接口及其实现。

接口部分：

```

package demo.ws.soap_spring_cxf;

import javax.jws.WebService;

@WebService
public interface HelloService {

    String say(String name);
}

```

实现部分：

```

package demo.ws.soap_spring_cxf;

import javax.jws.WebService;
import org.springframework.stereotype.Component;

@WebService
@Component
public class HelloServiceImpl implements HelloService {

    public String say(String name) {

```

```
        return "hello " + name;
    }
}
```

需要在实现类上添加 Spring 的 `org.springframework.stereotype.Component` 注解,这样才能被 Spring IoC 容器扫描到,认为它是一个 Spring Bean,可以根据 Bean ID(这里是 `helloServiceImpl`)来获取 Bean 实例。

第三步: 配置 `web.xml`。

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

    <!-- Spring -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoader-
            Listener</listener-class>
    </listener>

    <!-- CXF -->
    <servlet>
        <servlet-name>cxfr</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</ser-
            vlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>cxfr</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>

</web-app>
```

所有带有/ws 前缀的请求将会交给 CXFServlet 进行处理，也就是处理 WS 请求。目前主要使用了 Spring IoC 的特性，利用 ContextLoaderListener 来加载 Spring 配置文件，即这里定义的 spring.xml 文件。

第四步：配置 Spring。

配置 spring.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:component-scan base-package="demo.ws"/>

    <import resource="spring-cxf.xml"/>

</beans>
```

以上配置做了两件事情：

(1) 定义 IoC 容器扫描路径，即这里定义的 demo.ws，在这个包下面（包括所有子包）凡是带有 Component 注解的类都会扫描到 Spring IoC 容器中。

(2) 引入 spring-cxf.xml 文件，用于编写 CXF 相关配置。将配置文件分离是一种很好的开发方式。

第五步：配置 CXF。

配置 spring-cxf.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://cxf.apache.org/jaxws
```

```

    http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:server id="helloService" address="/soap/hello">
        <jaxws:serviceBean>
            <ref bean="helloServiceImpl"/>
        </jaxws:serviceBean>
    </jaxws:server>

</beans>

```

通过 CXF 提供的 Spring 命名空间即（jaxws:server）来发布 WS，其中最重要的是 address 属性，以及通过 jaxws:serviceBean 配置的 Spring Bean。

可见，在 Spring 中集成 CXF 比想象中更加简单，此外，还有一种更简单的配置方法，那就是使用 CXF 提供的 endpoint 方式，配置如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:endpoint id="helloService" implementor="#helloServiceImpl"
        address="/soap/hello"/>

</beans>

```

使用 jaxws:endpoint 可以简化 WS 发布的配置，与 jaxws:server 相比，确实是一种进步。

注意：这里的 implementor 属性值是 #helloServiceImpl，这是 CXF 特有的简写方式，并非 Spring 的规范，意思是通过 Spring 的 Bean ID 来获取 Bean 实例。

同样，也可以在 Spring 中使用 simple 方式来发布 WS，配置如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:simple="http://cxf.apache.org/simple"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://cxf.apache.org/simple
http://cxf.apache.org/schemas/simple.xsd">

<simple:server id="helloService" serviceClass="#helloService" address=
"/soap/hello">
  <simple:serviceBean>
    <ref bean="#helloServiceImpl"/>
  </simple:serviceBean>
</simple:server>

</beans>
```

可见，`simple:server` 与 `jaxws:server` 的配置方式类似，都需要配置一个 `serviceBean`。

比较以上这三种方式，我更加喜欢第二种，也就是 `endpoint` 方式，因为它够简单。

至于为什么 CXF 要提供如此之多的 WS 发布方式，个人认为，CXF 为了满足广大开发者的喜好，也是为了向前兼容，所以将这些方案全部保留下来了。

第六步：启动 Tomcat。

将应用部署到 Tomcat 中，在浏览器中输入以下地址可进入 CXF 控制台：

```
http://localhost:8080/ws
[CFX 控制台][http://static.oschina.net/uploads/space/2014/0702/144835_
r8rI_223750.png]
```

通过以上过程，可以看出 CXF 完全具备 RI 的易用性，并且与 Spring 有很好的可集成性，而且配置也非常简单。

同样通过以下地址可以查看 WSDL：

`http://localhost:8080/ws/soap/hello?wsdl`

注意：紧跟在 `/ws` 前缀后面的 `/soap/hello`，其实是在 `address="/soap/hello"` 中配置的。

现在已经成功地通过 CXF 对外发布了 WS，下面要做的事情就是用 WS 客户端来调用这些 `endpoint`。

可以不再使用 JDK 内置的 WS 客户端，也不必通过 WSDL 打包客户端 jar，因为 CXF 已经提供了多种 WS 客户端解决方案，可以根据需求自行选择。

4. CXF 提供的 WS 客户端

方案一：静态代理客户端。

```
package demo.ws.soap_cxf;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;

public class JaxWsClient {

    public static void main(String[] args) {
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
        factory.setAddress("http://localhost:8080/ws/soap/hello");
        factory.setServiceClass(HelloService.class);

        HelloService helloService = factory.create(HelloService.class);
        String result = helloService.say("world");
        System.out.println(result);
    }
}
```

这种方案需要自行通过 WSDL 打包客户端 jar，通过静态代理的方式来调用 WS。这种做法最为原始，下面的方案更有特色。

方案二：动态代理客户端。

```
package demo.ws.soap_cxf;

import org.apache.cxf.endpoint.Client;
import org.apache.cxf.jaxws.endpoint.dynamic.JaxWsDynamicClientFactory;

public class JaxWsDynamicClient {

    public static void main(String[] args) {
        JaxWsDynamicClientFactory factory = JaxWsDynamicClientFactory.
            newInstance();
        Client client = factory.createClient("http://localhost:8080/ws/
            soap/hello?wsdl");
    }
}
```

```
        try {
            Object[] results = client.invoke("say", "world");
            System.out.println(results[0]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

这种方案无须通过 WSDL 打包客户端 jar，底层实际上是通过 JDK 的动态代理特性完成的，CXF 实际上做了一个简单的封装。与 JDK 动态客户端不同的是，此时无须使用 HelloService 接口，可以说是货真价实的 WS 动态客户端。

方案三：通用动态代理客户端。

```
package demo.ws.soap_cxf;

import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.dynamic.DynamicClientFactory;

public class DynamicClient {

    public static void main(String[] args) {
        DynamicClientFactory factory = DynamicClientFactory.newInstance();
        Client client = factory.createClient("http://localhost:8080/ws/soap/hello?wsdl");

        try {
            Object[] results = client.invoke("say", "world");
            System.out.println(results[0]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

这种方案与“方案二”类似，但不同的是，它不仅用于调用 JAX-WS 方式发布的 WS，也用于使用 simple 方式发布的 WS，更加智能了。

方案四：基于 CXF simple 方式的客户端。

```
package demo.ws.soap_cxf;

import org.apache.cxf.frontend.ClientProxyFactoryBean;

public class SimpleClient {

    public static void main(String[] args) {
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
        factory.setAddress("http://localhost:8080/ws/soap/hello");
        factory.setServiceClass(HelloService.class);
        HelloService helloService = factory.create(HelloService.class);
        String result = helloService.say("world");
        System.out.println(result);
    }
}
```

这种方式仅用于调用 simple 方式发布的 WS，不能调用 JAX-WS 方式发布的 WS，这点需要注意。

方案五：基于 Spring 的客户端。

方法一：使用 JaxWsProxyFactoryBean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

    <bean id="factoryBean" class="org.apache.cxf.jaxws.JaxWsProxyFactory-
Bean">
        <property name="serviceClass" value="demo.ws.soap_spring_cxf.
HelloService"/>
        <property name="address" value="http://localhost:8080/ws/soap/
hello"/>
    </bean>

    <bean id="helloService" factory-bean="factoryBean" factory-method=
```

```

        "create"/>

</beans>

```

方法二：使用 `jaxws:client`（推荐）。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="helloService"
                 serviceClass="demo.ws.soap_spring_cxf.HelloService"
                 address="http://localhost:8080/ws/soap/hello"/>

</beans>

```

客户端代码：

```

package demo.ws.soap_spring_cxf;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            ("spring-client.xml");

        HelloService helloService = context.getBean("helloService",
            HelloService.class);
        String result = helloService.say("world");
        System.out.println(result);
    }
}

```

谈不上哪种方案更加优秀，建议根据实际场景选择最为合适的方案。

5.6.3 基于 SOAP 的安全控制

1. 什么是 WSDL

WSDL 的全称是 Web Services Description Language（Web 服务描述语言），用于描述 WS 的具体内容。

当成功发布一个 WS 后，就能在浏览器中通过一个地址来查看基于 WSDL 文档，它是一个基于 XML 的文档。一个典型的 WSDL 地址如下：<http://localhost:8080/ws/soap/hello?wsdl>。

注意：WSDL 地址必须带有一个 wsdl 参数。

在浏览器中会看到一个标准的 XML 文档，如图 5-4 所示。

```
▼<wSDL:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns:tns="http://soap.spring_cxf_wss4j.ws.demo/" xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
  xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="HelloServiceImplService"
  targetNamespace="http://soap.spring_cxf_wss4j.ws.demo/">
  <wSDL:types>...</wSDL:types>
  <wSDL:message name="say">...</wSDL:message>
  <wSDL:message name="sayResponse">...</wSDL:message>
  <wSDL:portType name="HelloService">...</wSDL:portType>
  <wSDL:binding name="HelloServiceImplServiceSoapBinding" type="tns:HelloService">...</wSDL:binding>
  <wSDL:service name="HelloServiceImplService">...</wSDL:service>
</wSDL:definitions>
```

图 5-4 WDSL 示例

其中，definitions 是 WSDL 的根节点，它包括两个重要的属性。

(1) name: WS 名称，默认为“WS 实现类+Service”，例如，HelloServiceImplService。

(2) targetNamespace: WS 目标命名空间，默认为“WS 实现类对应包名倒排后构成的地址”，例如，http://soap.spring_cxf.ws.demo/。

提示：可以在 javax.jws.WebService 注解中配置以上两个属性值，但这个配置一定要在 WS 实现类上进行，WS 接口类只需标注一个 WebService 注解即可。

在 definitions 这个根节点下，有五种类型的子节点：

- types 描述了 WS 中所涉及的数据类型；
- portType 定义了 WS 接口名称（endpointInterface）及其操作名称，以及每个操作的输入与输出消息；

- `message` 对相关消息进行了定义（供 `types` 与 `portType` 使用）；
- `binding` 提供了对 WS 的数据绑定方式；
- `service` 是 WS 名称及其端口名称（`portName`），以及对应的 WSDL 地址。

其中包括了两个重要信息：

- `portName` 是 WS 的端口名称，默认为“WS 实现类+Port”，如 `HelloServiceImplPort`；
- `endpointInterface` 是 WS 的接口名称，默认为“WS 实现类所实现的接口”，如 `HelloService`。

提示：可在 `javax.jws.WebService` 注解中配置 `portName` 与 `endpointInterface`，同样必须在 WS 实现类上配置。

2. 什么是 SOAP

如果说 WSDL 用于描述 WS 是什么，那么 SOAP 就用来表示 WS 里有什么。

其实 SOAP 就是一个信封（Envelope），在这个信封里包括两个部分，一是头（Header），二是体（Body）。用于传输的数据都放在 Body 中了，一些特殊的属性需要放在 Header 中（下面会看到）。

一般情况下，将需要传输的数据放入 Body 中，而 Header 是没有任何内容的，看起来整个 SOAP 消息如图 5-5 所示。

```
▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  ▼<soap:Body>
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
      <arg0>world</arg0>
    </ns2:say>
  </soap:Body>
</soap:Envelope>
```

图 5-5 SOAP 消息示例

可见 HTTP 请求的是 Request Header 与 Request Body，这正好与 SOAP 消息的结构有着异曲同工之妙！

看到这里，您或许会有很多疑问：

- （1）WS 不应该让任何人都可以调用，这样太不安全了，至少需要做一个身份认证吧？
- （2）为了避免第三方恶意程序监控 WS 调用过程，能否对 SOAP Body 中的数据进行加密呢？

(3) SOAP Header 中究竟可以存放什么东西呢？

没错！这就是我们下面要展开讨论的话题——基于 SOAP 的安全控制。

在 WS 领域有一个很强悍的解决方案，名为 WS-Security，它仅仅是一个规范，在 Java 业界里有一个很权威的实现，名为 WSS4J。

WSS4J 官网：<http://ws.apache.org/wss4j/>。

下面将一步步讲解如何使用 Spring+CXF+WSS4J 实现一个安全可靠的 WS 调用框架。

其实需要做的也就是两件事情：

- 认证 WS 请求；
- 加密 SOAP 消息。

如何对 WS 进行身份认证呢？可使用如下解决方案。

3. 基于用户令牌的身份认证

第一步：添加 CXF 提供的 WS-Security 的 Maven 依赖。

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-ws-security</artifactId>
  <version>${cxf.version}</version>
</dependency>
```

其实底层实现还是 WSS4J，CXF 只是对其做了一个封装而已。

第二步：完成服务端 CXF 相关配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cxf="http://cxf.apache.org/core"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd
    http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd">
```

```
<bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <constructor-arg>
    <map>
      <!-- 用户认证（明文密码） -->
      <entry key="action" value="UsernameToken"/>
      <entry key="passwordType" value="PasswordText"/>
      <entry key="passwordCallbackRef" value-ref="serverPassword-
        Callback"/>
    </map>
  </constructor-arg>
</bean>

<jaxws:endpoint id="helloService" implementor="#helloServiceImpl"
  address="/soap/hello">
  <jaxws:inInterceptors>
    <ref bean="wss4jInInterceptor"/>
  </jaxws:inInterceptors>
</jaxws:endpoint>

<cxf:bus>
  <cxf:features>
    <cxf:logging/>
  </cxf:features>
</cxf:bus>

</beans>
```

首先定义了一个基于 WSS4J 的拦截器（WSS4JInInterceptor），然后通过“<jaxws:inInterceptors>”将其配置到 helloService 上，最后使用 CXF 提供的 Bus 特性，只需要在 Bus 上配置一个 logging feature，就可以监控每次 WS 请求与响应的日志了。

注意：这个 WSS4JInInterceptor 是一个 InInterceptor，表示对输入的消息进行拦截，同样还有 OutInterceptor，表示对输出的消息进行拦截。由于以上是服务器端的配置，因此我们只需要配置 InInterceptor 即可，对于客户端而言，我们可以配置 OutInterceptor（下面会看到）。

有必要对以上配置中 WSS4JInInterceptor 的构造器参数做一个说明：

- `action = UsernameToken` 表示使用基于“用户名令牌”的方式进行身份认证；
- `passwordType = PasswordText` 表示密码以明文方式出现；
- `passwordCallbackRef = serverPasswordCallback` 需要提供一个用于密码验证的回调处理器（`CallbackHandler`）。

以下便是 `ServerPasswordCallback` 的具体实现：

```
package demo.ws.soap_spring_cxf_wss4j;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;
import org.springframework.stereotype.Component;

@Component
public class ServerPasswordCallback implements CallbackHandler {

    private static final Map<String, String> userMap = new HashMap<String, String>();

    static {
        userMap.put("client", "clientpass");
        userMap.put("server", "serverpass");
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        WSPasswordCallback callback = (WSPasswordCallback) callbacks[0];

        String clientUsername = callback.getIdentifier();
        String serverPassword = userMap.get(clientUsername);
```

```

        if (serverPassword != null) {
            callback.setPassword(serverPassword);
        }
    }
}

```

它实现了 `javax.security.auth.callback.CallbackHandler` 接口，这是 JDK 提供的用于安全认证的回调处理器接口。在代码中提供了两个用户，分别是 `client` 与 `server`，用户名与密码存放在 `userMap` 中。这里需要将 JDK 提供的 `javax.security.auth.callback.Callback` 转型为 WSS4J 提供的 `org.apache.wss4j.common.ext.WSPasswordCallback`，在 `handle` 方法中实现对客户端密码的验证，最终需要将密码放入 `callback` 对象中。

第三步：完成客户端 CXF 相关配置。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.0.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd">

    <context:component-scan base-package="demo.ws"/>

    <bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JOut-Interce-ptor">
        <constructor-arg>
            <map>
                <!-- 用户认证（明文密码） -->
                <entry key="action" value="UsernameToken"/>
                <entry key="user" value="client"/>
                <entry key="passwordType" value="PasswordText"/>
                <entry key="passwordCallbackRef" value-ref="clientPass-
                    wordCallback"/>
            </map>
        </constructor-arg>
    </bean>

```

```

        </map>
    </constructor-arg>
</bean>

<jaxws:client id="helloService"
    serviceClass="demo.ws.soap_spring_cxf_wss4j.HelloService"
    address="http://localhost:8080/ws/soap/hello">
    <jaxws:outInterceptors>
        <ref bean="wss4jOutInterceptor"/>
    </jaxws:outInterceptors>
</jaxws:client>

</beans>

```

注意：这里使用的是 WSS4JOutInterceptor，它是一个 OutInterceptor，使客户端对输出的消息进行拦截。

WSS4JOutInterceptor 的配置基本上与 WSS4JInInterceptor 大同小异，这里需要提供客户端的用户名（user=client），还需要提供一个客户端密码回调处理器（passwordCallbackRef=clientPasswordCallback），代码如下：

```

package demo.ws.soap_spring_cxf_wss4j;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;
import org.springframework.stereotype.Component;

@Component
public class ClientPasswordCallback implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        WSPasswordCallback callback = (WSPasswordCallback) callbacks[0];
    }
}

```

```

        callback.setPassword("clientpass");
    }
}

```

ClientPasswordCallback 设置客户端用户的密码，其他什么也不用做。客户端密码只能通过回调处理器的方式来提供，而不能在 Spring 中配置。

第四步：调用 WS 并观察控制台日志。

部署应用并启动 Tomcat，再次调用 WS，此时会在 Tomcat 控制台里的 Inbound Message 中看到如图 5-6 所示的 Payload：

```

▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
      ▼<wsse:UsernameToken wsu:Id="UsernameToken-a67bb246-b327-482c-a588-0eda911e1a62">
        <wsse:Username>client</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">clientpass</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </SOAP-ENV:Header>
  ▼<soap:Body>
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
      <arg0>world</arg0>
    </ns2:say>
  </soap:Body>
</soap:Envelope>

```

图 5-6 明文请求

可见，在 SOAP Header 中提供了 UsernameToken 的相关信息，但 Username 与 Password 都是明文的，SOAP Body 也是明文的，这显然不是最好的解决方案。

如果将 passwordType 由 PasswordText 改为 PasswordDigest（服务端与客户端都需要做同样的修改），那么就会看到一个加密过的密码，如图 5-7 所示。

```

▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
      ▼<wsse:UsernameToken wsu:Id="UsernameToken-f0480da4-0ed4-4c73-9cef-4e887aa14761">
        <wsse:Username>client</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest">DwtYauMulK6lkSYEXjfEy4rMjb0</wsse:Password>
        <wsse:Nonce EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary">n0RmielnlwzBbp7hKjeYkw==</wsse:Nonce>
        <wsu:Created>2014-07-03T16:22:36.825Z</wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </SOAP-ENV:Header>
  ▼<soap:Body>
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
      <arg0>world</arg0>
    </ns2:say>
  </soap:Body>
</soap:Envelope>

```

图 5-7 密文响应

除了这种基于用户名与密码的身份认证以外，还有一种更安全的身份认证方式，名为“数

字签名”。

4. 基于数字签名的身份认证

数字签名从字面上理解就是一种基于数字的签名方式。也就是说，当客户端发送 SOAP 消息时，需要对其进行“签名”来证实自己的身份，当服务端接收 SOAP 消息时，需要对其签名进行验证（简称“验签”）。

在客户端与服务端上都有各自的“密钥库”，这个密钥库里存放了“密钥对”，而密钥对实际上是由“公钥”与“私钥”组成的。当客户端发送 SOAP 消息时，需要使用自己的私钥进行签名，当客户端接收 SOAP 消息时，需要使用客户端提供的公钥进行验签。

因为有请求就有响应，所以客户端与服务端的消息调用实际上是双向的。也就是说，客户端与服务端的密钥库里所存放的信息是这样的：

- 客户端密钥库——客户端的私钥（用于签名）、服务端的公钥（用于验签）；
- 服务端密钥库——服务端的私钥（用于签名）、客户端的公钥（用于验签）。

记住一句话：使用自己的私钥进行签名，使用对方的公钥进行验签。

可见生成密钥库是我们要做的第一件事情。

第一步：生成密钥库。

现在需要创建一个名为 `keystore.bat` 的批处理文件，其内容如下：

```
@echo off

keytool -genkeypair -alias server -keyalg RSA -dname "cn=server" -keypass
serverpass -keystore server_store.jks -storepass storepass
keytool -exportcert -alias server -file server_key.rsa -keystore server_
store.jks -storepass storepass
keytool -importcert -alias server -file server_key.rsa -keystore client_
store.jks -storepass storepass -noprompt
del server_key.rsa

keytool -genkeypair -alias client -dname "cn=client" -keyalg RSA -keypass
clientpass -keystore client_store.jks -storepass storepass
keytool -exportcert -alias client -file client_key.rsa -keystore client_
store.jks -storepass storepass
keytool -importcert -alias client -file client_key.rsa -keystore server_
store.jks -storepass storepass -noprompt
del client_key.rsa
```

在以上这些命令中，使用了 JDK 提供的 `keytool` 命令行工具，关于该命令的使用方法，可参考 `keytool` 命令：<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>。

运行该批处理程序将生成两个文件：`server_store.jks` 与 `client_store.jks`，随后将 `server_store.jks` 放入服务端的 `classpath` 下，将 `client_store.jks` 放入客户端的 `classpath` 下。如果在本机运行，那么本机既是客户端又是服务端。

第二步：完成服务端 CXF 相关配置。

```
...
<bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.
WSS4JInInterceptor">
  <constructor-arg>
    <map>
      <!-- 验签（使用对方的公钥） -->
      <entry key="action" value="Signature"/>
      <entry key="signaturePropFile" value="server.properties"/>
    </map>
  </constructor-arg>
</bean>
...
```

其中 `action` 为 `Signature`，`server.properties` 内容如下：

```
org.apache.ws.security.crypto.provider=org.apache.wss4j.common.crypto.
Merlin
org.apache.ws.security.crypto.merlin.file=server_store.jks
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=storepass
```

第三步：完成客户端 CXF 相关配置。

```
...
<bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.
WSS4JOutInterceptor">
  <constructor-arg>
    <map>
      <!-- 签名（使用自己的私钥） -->
      <entry key="action" value="Signature"/>
      <entry key="signaturePropFile" value="client.properties"/>
      <entry key="signatureUser" value="client"/>
    </map>
  </constructor-arg>
</bean>
...
```

```

        <entry key="passwordCallbackRef" value-ref="clientPassword-
        Callback"/>
    </map>
</constructor-arg>
</bean>
...

```

其中 action 为 Signature, client.properties 内容如下:

```

org.apache.ws.security.crypto.provider=org.apache.wss4j.common.crypto.
Merlin
org.apache.ws.security.crypto.merlin.file=client_store.jks
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=storepass

```

此外, 客户端同样需要提供签名用户 (signatureUser) 与密码回调处理器 (passwordCallbackRef)。

第四步: 调用 WS 并观察控制台日志。

数字签名如图 5-8 所示。

```

▼<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ▼<SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    ▼<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      soap:mustUnderstand="1">
      ▼<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="SIG-ab21c91e-1dd4-49c5-94f7-606301991e0e">
        ▼<ds:SignedInfo>
          ▼<ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList="soap"/>
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          ▼<ds:Reference URI="#id-1cd390f4-4cee-4a5a-b44b-da8d7ed8258e">
            ▼<ds:Transforms>
              ▼<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList=""/>
              </ds:Transform>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>UED09/7CTPhZ4ME0h/levZF41EQ=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        ▼<ds:SignatureValue>
          MqdaRzR8LOx3B456B+PHqy/A4L7ozzyXsdde9PZNrI2VzIZrPwnkf6SMGedg2nuLPBKmWeFFN9JwnLR5p8KWZ398P/giNrJN/yZRUEFmUm05Q0g
        </ds:SignatureValue>
        ▼<ds:KeyInfo Id="KI-004a1cfb-aafe-45d9-a782-6862180f7cc6">
          ▼<wsse:SecurityTokenReference wsu:Id="STR-b2ad16a7-b52f-4815-954d-e6ee25a6d1b4">
            ▼<ds:X509Data>
              ▼<ds:X509IssuerSerial>
                <ds:X509IssuerName>CN=client</ds:X509IssuerName>
                <ds:X509SerialNumber>1404051942</ds:X509SerialNumber>
              </ds:X509IssuerSerial>
            </ds:X509Data>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </SOAP-ENV:Header>
  ▼<soap:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="id-1cd390f4-4cee-4a5a-b44b-da8d7ed8258e">
    ▼<ns2:say xmlns:ns2="http://soap_spring_cxf_wss4j.ws.demo/">
      <arg0>world</arg0>
    </ns2:say>
  </soap:Body>
</soap:Envelope>

```

图 5-8 数字签名

可见，数字签名确实是一种更为安全的身份认证方式，但无法对 SOAP Body 中的数据进行加密，数据仍然是“world”。

究竟怎样才能加密并解密 SOAP 消息中的数据呢？

5. SOAP 消息的加密与解密

WSS4J 除了提供签名与验签（Signature）这个特性以外，还提供了加密与解密（Encrypt）功能，只需要在服务端与客户端的配置中稍作修改即可。

服务端：

```
...
<bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.
WSS4JInInterceptor">
  <constructor-arg>
    <map>
      <!-- 验签 与 解密 -->
      <entry key="action" value="Signature Encrypt"/>
      <!-- 验签（使用对方的公钥） -->
      <entry key="signaturePropFile" value="server.properties"/>
      <!-- 解密（使用自己的私钥） -->
      <entry key="decryptionPropFile" value="server.properties"/>
      <entry key="passwordCallbackRef" value-ref="serverPassword-
      Callback"/>
    </map>
  </constructor-arg>
</bean>
...
```

客户端：

```
...
<bean id="wss4jOutInterceptor" class="org.apache.cxf.ws.security.wss4j.
WSS4JOutInterceptor">
  <constructor-arg>
    <map>
      <!-- 签名 与 加密 -->
      <entry key="action" value="Signature Encrypt"/>
      <!-- 签名（使用自己的私钥） -->
      <entry key="signaturePropFile" value="client.properties"/>
      <entry key="signatureUser" value="client"/>
    </map>
  </constructor-arg>
</bean>
...
```

```

        <entry key="passwordCallbackRef" value-ref="clientPassword-
        Callback"/>
        <!-- 加密（使用对方的公钥） -->
        <entry key="encryptionPropFile" value="client.properties"/>
        <entry key="encryptionUser" value="server"/>
    </map>
</constructor-arg>
</bean>
...

```

可见，客户端发送 SOAP 消息时进行签名（使用自己的私钥）与加密（使用对方的公钥），服务端接收 SOAP 消息时进行验签（使用对方的公钥）与解密（使用自己的私钥）。

现在看到的 SOAP 消息应该是这样的，如图 5-9 所示。

```

<?xml version='1.0' encoding='UTF-8'>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
      <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id="EK-4dffbad0-d34a-4f0f-984c-da6d4f81b6ae">...
      </xenc:EncryptedKey>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="SIG-4dd32bd3-6ff8d-4381-bc80-3312e5167221">...
      </ds:Signature>
    </wsse:Security>
  </SOAP-ENV:Header>
  <soap:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="id-38af1098-f9ef-4068-acf3-600de3e3defa">
    <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id="ED-af81b1ea-0dbb-4bc0-9a74-7b931453c6c1"
      Type="http://www.w3.org/2001/04/xmlenc#Content">
      <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <wsse:SecurityTokenReference xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
          wsu:Id="id-38af1098-f9ef-4068-acf3-600de3e3defa">
          <wsse:Reference URI="#EK-4dffbad0-d34a-4f0f-984c-da6d4f81b6ae">
            <wsse:SecurityTokenReference/>
          </wsse:Reference>
        </ds:KeyInfo>
      </xenc:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
          iBB1Y1FG5a/bSC//PuQAj07j9WjnDeGfrkSx7DHUCIF1UsYSnWb1jQmyCrR0tkUnkLX49Gf/4tx4tEcIQNyp+o2/ABhJXtBvKaDfYBTPl+ICR0M2olgc
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>

```

图 5-9 数字签名与消息加密

可见 SOAP 请求不仅签名了，而且还加密了，这样的通信更加安全可靠。

但是还存在一个问题，虽然 SOAP 请求已经很安全了，但 SOAP 响应却没有做任何安全控制，如图 5-10 所示。

```

<?xml version='1.0' encoding='UTF-8'>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:sayResponse xmlns:ns2="http://soap.spring_cxf_wss4j.ws.demo/">
      <return>hello world</return>
    </ns2:sayResponse>
  </soap:Body>
</soap:Envelope>

```

图 5-10 未做安全控制的响应

如何才能对 SOAP 响应进行签名与加密呢？您可以亲自动手试一试。

5.6.4 使用 CXF 开发 REST 服务

1. 什么是 REST

我们将视角集中在 REST 上，它是继 SOAP 以后，另一种广泛使用的 Web 服务。与 SOAP 不同，REST 并没有 WSDL 的概念，也没有叫“信封”的东西，因为 REST 主张用一种简单粗暴的方式来表达数据，传递的数据格式可以是 JSON 格式，也可以是 XML 格式，这完全由用户来决定。

REST 全称是 Representational State Transfer（表述性状态转移），它源自 Roy Fielding 博士在 2000 年写的一篇关于软件架构风格的论文，此文一出，震撼四方！许多知名互联网公司开始采用这种轻量级 Web 服务，我们习惯将其称为 RESTful Web Services，或简称 REST 服务。

那么 REST 到底是什么呢？

REST 本质是使用 URL 来访问资源的一种方式。众所周知，URL 就是我们平常使用的请求地址，其中包括两部分：请求方式与请求路径。比较常见的请求方式是 GET 与 POST，但在 REST 中又提出了其他几种其他类型的请求方式，汇总起来有六种：GET、POST、PUT、DELETE、HEAD、OPTIONS。尤其是前四种，正好与 CRUD（增删改查）四种操作相对应：GET（查）、POST（增）、PUT（改）、DELETE（删），这正是 REST 的奥妙所在。

实际上，REST 是一个“无状态”的架构模式，因为在任何时候都可以由客户端发出请求到服务端，最终返回自己想要的的数据。也就是说，服务端将内部资源发布为 REST 服务，客户端通过 URL 来访问这些资源，这不就是 SOA 所提倡的“面向服务”的思想吗？所以，REST 也被人们看作一种轻量级的 SOA 实现技术，因此在企业级应用与互联网应用中都得到了广泛使用。

在 Java 的世界里，有一个名为 JAX-RS 的规范，它就是用来实现 REST 服务的，目前已经发展到了 2.0 版本，也就是 JSR-339 规范，如果想深入研究 REST，请深入阅读此规范。

JSR-339 规范：<https://www.jcp.org/en/jsr/detail?id=339>。

JAX-RS 规范目前有以下几种比较流行的实现技术：

- Jersey——<https://jersey.java.net/>;
- Restlet——<http://restlet.com/>;
- RESTEasy——<http://resteasy.jboss.org/>;
- CXF——<http://cxf.apache.org/>。

本节以 CXF 为例，讲解如何使用 CXF 开发 REST 服务，此外还会将 Spring 与 CXF 做一个整合，让开发更加高效。

2. 使用 CXF 发布与调用 REST 服务

第一步：添加 Maven 依赖。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.ws</groupId>
    <artifactId>rest_cxf</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cxf.version>3.0.4</cxf.version>
        <jackson.version>2.4.1</jackson.version>
    </properties>

    <dependencies>
        <!-- CXF -->
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxrs</artifactId>
            <version>${cxf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transport-http-jetty</artifactId>
            <version>${cxf.version}</version>
        </dependency>
        <!-- Jackson -->
```

```

        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-json-provider</artifactId>
            <version>${jackson.version}</version>
        </dependency>
    </dependencies>

</project>

```

以上添加了 CXF 关于 REST 的依赖包，并使用了 Jackson 来实现 JSON 数据的转换。

第二步：定义一个 REST 服务接口。

```

package demo.ws.rest_cxf;

import java.util.List;
import java.util.Map;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

public interface ProductService {

    @GET
    @Path("/products")
    @Produces(MediaType.APPLICATION_JSON)
    List<Product> retrieveAllProducts();

    @GET
    @Path("/product/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    Product retrieveProductById(@PathParam("id") long id);
}

```

```

    @POST
    @Path("/products")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @Produces(MediaType.APPLICATION_JSON)
    List<Product> retrieveProductsByName(@FormParam("name") String name);

    @POST
    @Path("/product")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    Product createProduct(Product product);

    @PUT
    @Path("/product/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    Product updateProductById(@PathParam("id") long id, Map<String, Object>
    fieldMap);

    @DELETE
    @Path("/product/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    Product deleteProductById(@PathParam("id") long id);
}

```

以上 `ProductService` 接口中提供了一系列的方法，在每个方法上都使用了 JAX-RS 提供的注解，主要包括以下四类：

- (1) 请求方式注解，包括 `@GET`、`@POST`、`@PUT`、`@DELETE`；
- (2) 请求路径注解，包括 `@Path`，其中包括一个路径参数；
- (3) 数据格式注解，包括 `@Consumes`（输入）、`@Produces`（输出），可使用 `MediaType` 常量；
- (4) 相关参数注解，包括 `@PathParam`（路径参数）、`@FormParam`（表单参数），此外还有 `@QueryParam`（请求参数）。

针对 `updateProductById` 方法，简单解释一下：

该方法将被 `PUT:/product/{id}` 请求调用，请求路径中的 `id` 参数将映射到 `long id` 参数上，请求体中的数据将自动转换为 JSON 格式并映射到 `Map fieldMap` 参数上，返回的 `Product` 类型的数据将自动转换为 JSON 格式并返回到客户端。

注意：由于 Product 类与 ProductService 接口的实现类并不是本节的重点，因此省略了，本节结尾处会给出源码链接。

第三步：使用 CXF 发布 REST 服务。

```
package demo.ws.rest_cxf;

import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.ResourceProvider;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class Server {

    public static void main(String[] args) {
        // 添加 ResourceClass
        List<Class<?>> resourceClassList = new ArrayList<Class<?>>();
        resourceClassList.add(ProductServiceImpl.class);

        // 添加 ResourceProvider
        List<ResourceProvider> resourceProviderList = new ArrayList<ResourceProvider>();
        resourceProviderList.add(new SingletonResourceProvider(new ProductServiceImpl()));

        // 添加 Provider
        List<Object> providerList = new ArrayList<Object>();
        providerList.add(new JacksonJsonProvider());

        // 发布 REST 服务
        JAXRSServerFactoryBean factory = new JAXRSServerFactoryBean();
        factory.setAddress("http://localhost:8080/ws/rest");
        factory.setResourceClasses(resourceClassList);
        factory.setResourceProviders(resourceProviderList);
        factory.setProviders(providerList);
    }
}
```

```

        factory.create();
        System.out.println("rest ws is published");
    }
}

```

CXF 提供了一个名为 `org.apache.cxf.jaxrs.JAXRSServerFactoryBean` 的类，专门用于发布 REST 服务，只需为该类的实例对象指定四个属性即可。

(1) **Address:** REST 基础地址。

(2) **ResourceClasses:** 一个或一组相关的资源类，即接口对应的实现类（注意：REST 规范允许资源类没有接口）。

(3) **ResourceProviders:** 资源类对应的 Provider，此时使用 CXF 提供的 `org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider` 进行装饰。

(4) **Providers:** REST 服务所需的 Provider，此时使用 Jackson 提供的 `org.codehaus.jackson.jaxrs.JacksonJsonProvider`，用于实现 JSON 数据的序列化与反序列化。

运行以上 Server 类，将以 standalone 方式发布 REST 服务，下面我们通过客户端来调用已发布的 REST 服务。

第四步：使用 CXF 调用 REST 服务。

首先添加如下 Maven 依赖：

```

<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-client</artifactId>
  <version>${cxf.version}</version>
</dependency>

```

CXF 提供了三种 REST 客户端，下面将分别进行展示。

第一种：JAX-RS 1.0 时代的客户端。

```

package demo.ws.rest_cxf;

import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.jaxrs.client.JAXRSClientFactory;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class JAXRSClient {

```

```

public static void main(String[] args) {
    String baseAddress = "http://localhost:8080/ws/rest";

    List<Object> providerList = new ArrayList<Object>();
    providerList.add(new JacksonJsonProvider());

    ProductService productService = JAXRSClientFactory.create (base-
        Address, ProductService.class, providerList);
    List<Product> productList = productService.retrieveAllProducts();
    for (Product product : productList) {
        System.out.println(product);
    }
}
}

```

本质上是使用 CXF 提供的 `org.apache.cxf.jaxrs.client.JAXRSClientFactory` 工厂类来创建 `ProductService` 代理对象，通过代理对象调用目标对象上的方法。客户端同样也需要使用 `Provider`，此时仍然使用了 Jackson 提供的 `org.codehaus.jackson.jaxrs.JacksonJsonProvider`。

第二种：JAX-RS 2.0 时代的客户端。

```

package demo.ws.rest_cxf;

import java.util.List;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.MediaType;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class JAXRS20Client {

    public static void main(String[] args) {
        String baseAddress = "http://localhost:8080/ws/rest";

        JacksonJsonProvider jsonProvider = new JacksonJsonProvider();

        List productList = ClientBuilder.newClient()
            .register(jsonProvider)
            .target(baseAddress)

```

```

        .path("/products")
        .request(MediaType.APPLICATION_JSON)
        .get(List.class);
    for (Object product : productList) {
        System.out.println(product);
    }
}
}
}

```

在 JAX-RS 2.0 中提供了一个名为 `javax.ws.rs.client.ClientBuilder` 的工具类, 可用于创建客户端并调用 REST 服务, 显然这种方式比前一种要先进, 因为在代码中不再依赖 CXF API 了。

如果想返回带有泛型的 `List`, 那么可以使用以下代码片段:

```

List<Product> productList = ClientBuilder.newClient()
    .register(jsonProvider)
    .target(baseAddress)
    .path("/products")
    .request(MediaType.APPLICATION_JSON)
    .get(new GenericType<List<Product>>() {});
for (Product product : productList) {
    System.out.println(product);
}

```

第三种: 通用的 `WebClient` 客户端。

```

package demo.ws.rest_cxf;

import java.util.ArrayList;
import java.util.List;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import org.apache.cxf.jaxrs.client.WebClient;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class CXFWebClient {

    public static void main(String[] args) {
        String baseAddress = "http://localhost:8080/ws/rest";

        List<Object> providerList = new ArrayList<Object>();
    }
}

```

```

        providerList.add(new JacksonJsonProvider());

        List productList = WebClient.create(baseAddress, providerList)
            .path("/products")
            .accept(MediaType.APPLICATION_JSON)
            .get(List.class);
        for (Object product : productList) {
            System.out.println(product);
        }
    }
}

```

CXF 还提供了一种更为简洁的方式, 使用 `org.apache.cxf.jaxrs.client.WebClient` 来调用 REST 服务, 这种方式在代码层面上还是相当简洁的。

如果想返回带有泛型的 `List`, 那么可以使用以下代码片段:

```

List<Product> productList = WebClient.create(baseAddress, providerList)
    .path("/products")
    .accept(MediaType.APPLICATION_JSON)
    .get(new GenericType<List<Product>>() {});
for (Product product : productList) {
    System.out.println(product);
}

```

3. 使用 Spring+CXF 发布 REST 服务

第一步: 添加 Maven 依赖。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.ws</groupId>
    <artifactId>rest_spring_cxf</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

```

```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <spring.version>4.0.6.RELEASE</spring.version>
  <cxf.version>3.0.4</cxf.version>
  <jackson.version>2.4.1</jackson.version>
</properties>

<dependencies>
  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <!-- CXF -->
  <dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxrs</artifactId>
    <version>${cxf.version}</version>
  </dependency>
  <!-- Jackson -->
  <dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
    <version>${jackson.version}</version>
  </dependency>
</dependencies>

</project>

```

这里仅依赖 **Spring Web** 模块（无须 **MVC** 模块），此外就是 **CXF** 与 **Jackson** 了。

第二步：配置 web.xml。

```

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

```

```

        version="3.0">

    <!-- Spring -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- CXF -->
    <servlet>
        <servlet-name>cxfr</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>cxfr</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>

</web-app>

```

使用 Spring 提供的 ContextLoaderListener 去加载 Spring 配置文件 spring.xml；使用 CXF 提供的 CXFServlet 去处理前缀为/ws/的 REST 请求。

第三步：将接口的实现类发布为 SpringBean。

```

package demo.ws.rest_spring_cxf;

import org.springframework.stereotype.Component;

@Component
public class ProductServiceImpl implements ProductService {
    ...
}

```

使用 Spring 提供的 org.springframework.stereotype.Component 注解，将 ProductServiceImpl

发布为 Spring Bean，交给 Spring IOC 容器管理，无须再进行 Spring XML 配置。

第四步：配置 Spring。

以下是 spring.xml 的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.
xsd">

    <context:component-scan base-package="demo.ws"/>

    <import resource="spring-cxf.xml"/>

</beans>
```

在以上配置中扫描 demo.ws 这个基础包路径，Spring 可访问该包中的所有 Spring Bean，比如上面使用 Component 注解发布的 ProductServiceImpl。此外还加载了另一个配置文件 spring-cxf.xml，其中包括了关于 CXF 的相关配置。

以下是 spring-cxf.xml 的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://cxf.apache.org/jaxrs
http://cxf.apache.org/schemas/jaxrs.xsd">

    <jaxrs:server address="/rest">
        <jaxrs:serviceBeans>
            <ref bean="productServiceImpl"/>
        </jaxrs:serviceBeans>
    </jaxrs:server>
</beans>
```

```

        <jaxrs:providers>
            <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJson-
                Provider"/>
        </jaxrs:providers>
    </jaxrs:server>

</beans>

```

使用 CXF 提供的 Spring 命名空间来配置 Service Bean（即前面提到的 Resource Class）与 Provider。注意，这里配置了一个 address，属性为“/rest”，表示 REST 请求的相对路径，与 web.xml 中配置的“/ws/*”结合起来，最终的 REST 请求根路径是“/ws/rest”，在 ProductService 接口方法上 Path 注解所配置的路径只是一个相对路径。

第五步：调用 REST 服务。

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Demo</title>
    <link href="http://cdn.bootcss.com/bootstrap/3.1.1/css/bootstrap.min.
        css" rel="stylesheet">
</head>
<body>

<div class="container">
    <div class="page-header">
        <h1>Product</h1>
    </div>
    <div class="panel panel-default">
        <div class="panel-heading">Product List</div>
        <div class="panel-body">
            <div id="product"></div>
        </div>
    </div>
</div>

<script src="http://cdn.bootcss.com/jquery/2.1.1/jquery.min.js">
</script>

```

```

<script src="http://cdn.bootcss.com/bootstrap/3.1.1/js/bootstrap.min.
js"></script>
<script src="http://cdn.bootcss.com/handlebars.js/1.3.0/handlebars.min.
js"></script>

<script type="text/x-handlebars-template" id="product_table_template">
  { {#if data} }
    <table class="table table-hover" id="product_table">
      <thead>
        <tr>
          <th>ID</th>
          <th>Product Name</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>
        { {#data} }
          <tr data-id="{{id}}" data-name="{{name}}">
            <td>{{id}}</td>
            <td>{{name}}</td>
            <td>{{price}}</td>
          </tr>
        { {/data} }
      </tbody>
    </table>
  { {else} }
    <div class="alert alert-warning">Can not find any data!</div>
  { {/if} }
</script>

<script>
  $(function() {
    $.ajax({
      type: 'get',
      url: 'http://localhost:8080/ws/rest/products',
      dataType: 'json',
      success: function(data) {
        var template = $("#product_table_template").html();

```

```

        var render = Handlebars.compile(template);
        var html = render({
            data: data
        });
        $('#product').html(html);
    }
});
});
</script>

</body>
</html>

```

使用一个简单的 HTML 页面来调用 REST 服务，也就是说，前端发送 AJAX 请求来调用后端发布的 REST 服务。这里使用了 jQuery、Bootstrap、Handlebars.js 等技术。

4. 关于 AJAX 的跨域问题

如果服务端部署在 foo.com 域名下，而客户端部署在 bar.com 域名下，此时从 bar.com 发出一个 AJAX 的 REST 请求到 foo.com，就会出现报错：

```
No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

要想解决以上这个 AJAX 跨域问题，有以下两种解决方案。

方案一：使用 JSONP 解决 AJAX 跨域问题。

JSONP 的全称是 JSON with Padding，实际上是在需要返回的 JSON 数据外，用一个 JS 函数进行封装。

可以这样来理解，服务器返回一个 JS 函数，参数是一个 JSON 数据，例如：callback({您的 JSON 数据})。虽然 AJAX 不能跨域访问，但 JS 脚本是可以跨域执行的，因此客户端将执行这个 callback 函数，并获取其中的 JSON 数据。

如果需要返回的 JSON 数据是：

```
{ "id":2, "name": "ipad mini", "price":2500 }, { "id":1, "name": "iphone 5s", "price":5000 }
```

那么对应的 JSONP 格式是：

```
callback([ { "id":2, "name": "ipad mini", "price":2500 }, { "id":1, "name": "iphone 5s", "price":5000 } ] );
```

CXF 已经提供了对 JSONP 的支持，只需要通过简单的配置即可实现。

首先，添加 Maven 依赖：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-extension-providers</artifactId>
  <version>${cxf.version}</version>
</dependency>
```

然后，添加 CXF 配置：

```
<jaxrs:server address="/rest">
  <jaxrs:serviceBeans>
    <ref bean="productServiceImpl"/>
  </jaxrs:serviceBeans>
  <jaxrs:providers>
    <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
    <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpPreStream-
      Interceptor"/>
  </jaxrs:providers>
  <jaxrs:inInterceptors>
    <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpInInterce-
      ptor"/>
  </jaxrs:inInterceptors>
  <jaxrs:outInterceptors>
    <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpPostStream-
      Interceptor"/>
  </jaxrs:outInterceptors>
</jaxrs:server>
```

注意：JsonpPreStreamInterceptor 一定要放在<jaxrs:providers>中，而不是<jaxrs:inInterceptors>中。

最后，使用 jQuery 发送基于 JSONP 的 AJAX 请求：

```
$.ajax({
  type: 'get',
  url: 'http://localhost:8080/ws/rest/products',
  dataType: 'jsonp',
```

```
jsonp: '_jsonp',
jsonpCallback: 'callback',
success: function(data) {
    var template = $("#product_table_template").html();
    var render = Handlebars.compile(template);
    var html = render({
        data: data
    });
    $('#product').html(html);
}
});
```

以上代码中有三个选项需要加以说明：

(1) `dataType` 必须为 “jsonp”，表示返回的数据类型为 JSONP 格式。

(2) `jsonp` 表示 URL 中 JSONP 回调函数的参数名，CXF 默认接收的参数名是 “_jsonp”，也可以在 `JsonpInInterceptor` 中配置。

(3) `jsonpCallback` 表示回调函数的名称，若未指定，则由 jQuery 自动生成。

方案二：使用 CORS 解决 AJAX 跨域问题。

CORS 的全称是 Cross-Origin Resource Sharing(跨域资源共享)，它是 W3C 提出的一个 AJAX 跨域访问规范，可以从以下地址来了解此规范：

<http://www.w3.org/TR/cors/>。

相比 JSONP 而言，CORS 更为强大，因为它弥补了 JSONP 只能处理 GET 请求的局限性，但是只有较为先进的浏览器才能全面支持 CORS。

CXF 同样也提供了对 CORS 的支持，通过简单的配置就能实现。

首先，添加 Maven 依赖：

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-security-cors</artifactId>
  <version>${cxf.version}</version>
</dependency>
```

然后，添加 CXF 配置：

```
<jaxrs:server address="/rest">
  <jaxrs:serviceBeans>
    <ref bean="productServiceImpl"/>
```

```
</jaxrs:serviceBeans>
<jaxrs:providers>
    <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
    <bean class="org.apache.cxf.rs.security.cors.CrossOriginResource-
        SharingFilter">
        <property name="allowOrigins" value="http://localhost"/>
    </bean>
</jaxrs:providers>
</jaxrs:server>
```

在 `CrossOriginResourceSharingFilter` 中配置 `allowOrigins` 属性，将其设置为客户端的域名，示例中为 `http://localhost`，需根据实际情况进行设置。

最后，使用 jQuery 发送 AJAX 请求，就像在相同域名下访问一样，无须做任何配置。

注意：在 IE8 中使用 jQuery 发送 AJAX 请求时，需要配置 `$.support.cors = true`，才能开启 CORS 特性。

5.7 提供 Web 服务特性

1. Web 服务的种类

两个系统之间可使用 Web 服务进行通信，那么究竟有几种 Web 服务呢？

在企业应用中使用最频繁的 Web 服务莫过于 SOAP 服务了，客户端将消息放入到“信封”里，通过 HTTP 进行传输，服务端将“信封”解开并获取其中的消息。此外，客户端还可以根据 SOAP 服务发布的 WSDL 地址来生成客户端代码，正因为这样才能保证 Web 服务的跨平台性。在安全方面，SOAP 服务提供了强大的 Security 规范与实现，我们可以轻松地加密与签名所要发送的消息。

对于互联网应用而言，一般不会考虑使用 SOAP 服务，而是使用另一种更加轻量级的 REST 服务。在客户端只需要通过 HTTP 来表达请求路径与方法就能发送 REST 请求，在服务端将 HTTP 中的数据反序列化为我们所需的对象。REST 提供了多种 HTTP 请求方法，一般常用的包括：GET、POST、PUT 与 DELETE，这些方法正好对应于资源的 CRUD 操作，我们也是正因为这样才大规模地使用 REST 服务，并在此基础上搭建 OpenAPI 开放平台。需要指出的是，REST 服务的安全性问题需要我们自行实现，或者利用第三方安全控制框架（例如，OAuth），从而确保 REST 服务的安全性。

除了 SOAP、REST 这两种 Web 服务以外，还有其他种类的 Web 服务，比如 Hessian、

XML-RPC、JSON-RPC 等，这些技术更加轻量级，但在功能与安全方面不如 SOAP 与 REST。本节重点介绍 SOAP 与 REST 服务的使用方法，以及如何将这些技术作为插件的形式整合到 Smart 框架中，尽可能降低开发者使用 Web 服务的成本。

下面，我们先考虑支持 SOAP 服务，让它成为框架的一个插件，该插件命名为 Smart SOAP Plugin。

2. 支持 SOAP 服务

需要在服务端发布 Web 服务，当发布成功后，我们可以访问 WSDL 来验证 Web 服务是否发布成功，随后可通过 WSDL 地址生成客户端代码，这种方法属于静态生成方式，当然更简单的方法是通过 WSDL 动态生成客户端并发送 SOAP 请求。

我们先来看看开发者应该如何发布 SOAP 服务。

3. 发布 SOAP 服务

想要发布 SOAP 服务，必须先定义一个接口类。下面我们以 Customer 为例进行说明，通过 customerId 来获取 Customer 对象，代码如下：

```
package org.smart4j.chapter5.soap;

import org.smart4j.chapter5.model.Customer;

/**
 * 客户 SOAP 服务接口
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface CustomerSoapService {

    /**
     * 根据 客户 ID 获取客户对象
     *
     * @param customerId 客户 ID
     * @return 客户对象
     */
    Customer getCustomer(long customerId);
}
```

这是一个普通的 Java 接口，除了需要添加一些有必要的代码注释以外，没有任何需要注意的地方。下面我们来看看该接口的实现类，代码如下：

```
package org.smart4j.chapter5.soap;

import org.smart4j.chapter5.model.Customer;
import org.smart4j.chapter5.service.CustomerService;
import org.smart4j.framework.annotation.Inject;
import org.smart4j.framework.annotation.Service;
import org.smart4j.plugin.soap.Soap;

/**
 * 客户 SOAP 服务接口实现
 *
 * @author huangyong
 * @since 1.0.0
 */
@Soap
@Service
public class CustomerSoapServiceImpl implements CustomerSoapService {

    @Inject
    private CustomerService customerService;

    public Customer getCustomer(long customerId) {
        return customerService.getCustomer(customerId);
    }
}
```

实现类除了要实现 CustomerSoapService 接口以外，还需要同时定义 Soap 注解与 Service 注解。Soap 注解用于定义该类是需要发布为 SOAP 服务的，Service 注解用于将该类创建的对象交给 Smart IOC 容器进行管理，此时可通过 Inject 注解依赖注入 customerService 对象，然后在方法中调用 customerService 对象的 getCustomer 方法，过程非常简单。

可见，仅仅将一个 Soap 注解定义在接口类上，并将实现类定义为 Smart Bean，就能发布 Web 服务，那么，调用 SOAP 服务也是这般简单吗？

4. 调用 SOAP 服务

我们通过一个单元测试来调用 SOAP 服务，只需要利用一个名为 SoapHelper 的助手类就能

根据 WSDL 来创建接口代理对象，随后可调用该代理对象的任意方法，对开发者而言就像调用目标对象一样，代码如下：

```
package org.smart4j.chapter5.soap;

import org.junit.Assert;
import org.junit.Test;
import org.smart4j.chapter5.model.Customer;
import org.smart4j.plugin.soap.SoapHelper;

/**
 * 客户 SOAP 服务单元测试
 *
 * @author huangyong
 * @since 1.0.0
 */
public class CustomerSoapServiceTest {

    @Test
    public void getCustomerTest() {
        String wsdl = "http://localhost:8080/soap/CustomerSoapService";
        CustomerSoapService customerSoapService = SoapHelper.createClient(
            wsdl, CustomerSoapService.class);
        Customer customer = customerSoapService.getCustomer(1);
        Assert.assertNotNull(customer);
    }
}
```

在调用之前，最好先在浏览器中查看一下 WSDL 文档能否正常显示。需要注意的是，在浏览器中需要在 WSDL 地址后面加上?wsdl 请求参数，这样才能看到一份基于 XML 格式的 WSDL 文档，就像下面这样：

```
http://localhost:8080/soap/CustomerSoapService?wsdl
```

看来在客户端调用 SOAP 服务也是非常简单的，因为 WSDL 是已知信息，我们只需利用 SoapHelper 类来创建接口代理类，把接口代理类当成目标接口类一样来调用即可。但有一个前提条件，客户端必须能访问 SOAP 接口类，也就是说，需要提供包含 SOAP 接口类的 jar 包给客户端，或者由客户端自行实现一个同样的 SOAP 接口类。

5. 实现 SOAP 插件

第一步：添加 Maven 依赖。

SOAP 插件依赖于 Servlet、CXF 与 Smart 框架，需要添加以下 Maven 依赖到 pom.xml 文件中。具体的 Maven 依赖配置如下：

```
<!-- Servlet -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>

<!-- CXF -->
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transport-http</artifactId>
  <version>${version.cxf}</version>
</dependency>
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>${version.cxf}</version>
</dependency>

<!-- Smart Framework -->
<dependency>
  <groupId>org.smart4j</groupId>
  <artifactId>smart-framework</artifactId>
  <version>1.0.0</version>
</dependency>
```

其中，`${version.cxf}` 占位符需要在 pom.xml 里定义对应的属性，代码如下：

```
<properties>
  <version.cxf>3.0.4</version.cxf>
</properties>
```

第二步：定义 Soap 注解。

我们首先应该定义一个 Soap 注解，定义在需要发布 SOAP 服务的接口类上。

```
package org.smart4j.plugin.soap;

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

/**
 * SOAP 服务注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Soap {

    /**
     * 服务名
     */
    String value() default "";
}
```

Soap 注解只有一个 value 属性，用于描述 SOAP 服务的名称，若为空，则默认使用服务类名为 SOAP 的服务，比如，CustomerSoapService。

第三步：编写 SoapHelper 助手类。

接下来我们需要编写一个名为 SoapHelper 的助手类，用于发布 SOAP 服务，此外还可以创建 SOAP 客户端，也就是 SOAP 服务接口代理对象。当调用 Web 服务时，可开启调用日志，这样可以查看调用的过程，为我们的调试工作提供帮助。SoapHelper 的代码如下：

```
package org.smart4j.plugin.soap;

import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.frontend.ClientProxyFactoryBean;
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.interceptor.Interceptor;
```

```

import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;
import org.apache.cxf.message.Message;

/**
 * SOAP 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public class SoapHelper {

    private static final List<Interceptor<? extends Message>> inInterce-
    ptorList = new ArrayList<Interceptor<? extends Message>>();
    private static final List<Interceptor<? extends Message>> outInterce-
    ptorList = new ArrayList<Interceptor<? extends Message>>();

    static {
        // 添加 Logging Interceptor
        if (SoapConfig.isLog()) {
            LoggingInInterceptor loggingInInterceptor = new LoggingIn-
            Interceptor();
            inInterceptorList.add(loggingInInterceptor);
            LoggingOutInterceptor loggingOutInterceptor = new LoggingOut-
            Interceptor();
            outInterceptorList.add(loggingOutInterceptor);
        }
    }

    /**
     * 发布 SOAP 服务
     */
    public static void publishService(String wsdl, Class<?> interfaceClass,
    Object implementInstance) {
        ServerFactoryBean factory = new ServerFactoryBean();
        factory.setAddress(wsdl);
        factory.setServiceClass(interfaceClass);
        factory.setServiceBean(implementInstance);
    }
}

```

```

        factory.setInInterceptors(inInterceptorList);
        factory.setOutInterceptors(outInterceptorList);
        factory.create();
    }

    /**
     * 创建 SOAP 客户端
     */
    public static <T> T createClient(String wsdl, Class<? extends T>
        interfaceClass) {
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
        factory.setAddress(wsdl);
        factory.setServiceClass(interfaceClass);
        factory.setInInterceptors(inInterceptorList);
        factory.setOutInterceptors(outInterceptorList);
        return factory.create(interfaceClass);
    }
}

```

可见 SoapHelper 是基于 CXF 实现的，我们把 CXF 的相关细节进行了封装，对外提供了非常简单的 API 供开发者使用。首先，通过 static 块来初始化 CXF 提供的日志拦截器。然后，利用 CXF 提供的 ServerFactoryBean 来发布 SOAP 服务，利用 CXF 提供的 ClientProxyFactoryBean 来创建 SOAP 客户端，此时都需要设置相关日志拦截器。

其中，SoapConfig 是一个普通的工具类，用于读取 smart.properties 文件中的相关配置项，代码如下：

```

package org.smart4j.plugin.soap;

import org.smart4j.framework.helper.ConfigHelper;

/**
 * 从配置文件中获取相关属性
 *
 * @author huangyong
 * @since 1.0.0
 */
public class SoapConfig {

```

```

    public static boolean isLog() {
        return ConfigHelper.getBoolean(SoapConstant.LOG);
    }
}

```

目前只提供了一个配置项，用于获取是否需要记录日志的信息，该配置项定义在 `SoapConstant` 常量类中，代码如下：

```

package org.smart4j.plugin.soap;

/**
 * SOAP 插件常量
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface SoapConstant {

    String SERVLET_URL = "/soap/*";

    String LOG = "smart.plugin.soap.log";
}

```

此外，这里还定义了一个 `SERVLET_URL` 常量，表示发布 WSDL 的路径前缀，目前没有考虑将其进行配置，该常量马上就会使用到。

第四步：实现一个 `SoapServlet`。

最后我们需要编写一个 `SoapServlet` 类，让这个 `Servlet` 去拦截所有的 SOAP 请求，该请求路径的前缀就是 `SoapConstant.SERVLET_URL` 常量值。此外，我们只需要扩展 CXF 提供的 `CXFNonSpringServlet` 类，并在 `Servlet` 初始化的时候去初始化 CXF 总线与发布 SOAP 服务即可，因为所有的事情都在 `CXFNonSpringServlet` 中完成了。代码如下：

```

package org.smart4j.plugin.soap;

import java.util.Set;
import javax.servlet.ServletConfig;
import javax.servlet.annotation.WebServlet;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.servlet.CXFNonSpringServlet;

```

```

import org.smart4j.framework.helper.BeanHelper;
import org.smart4j.framework.helper.ClassHelper;
import org.smart4j.framework.util.CollectionUtil;
import org.smart4j.framework.util.StringUtil;

/**
 * SOAP Servlet
 *
 * @author huangyong
 * @since 1.0.0
 */
@WebServlet(urlPatterns = SoapConstant.SERVLET_URL, loadOnStartup = 0)
public class SoapServlet extends CXFNonSpringServlet {

    @Override
    protected void loadBus(ServletConfig sc) {
        // 初始化 CXF 总线
        super.loadBus(sc);
        Bus bus = getBus();
        BusFactory.setDefaultBus(bus);
        // 发布 SOAP 服务
        publishSoapService();
    }

    private void publishSoapService() {
        // 遍历所有标注了 SOAP 注解的类
        Set<Class<?>> soapClassSet = ClassHelper.getClassSetByAnnotation
            (Soap.class);
        if (CollectionUtil.isNotEmpty(soapClassSet)) {
            for (Class<?> soapClass : soapClassSet) {
                // 获取 SOAP 地址
                String address = getAddress(soapClass);
                // 获取 SOAP 类的接口
                Class<?> soapInterfaceClass = getSoapInterfaceClass
                    (soapClass);
                // 获取 SOAP 类的实例
                Object soapInstance = BeanHelper.getBean(soapClass);
                // 发布 SOAP 服务
            }
        }
    }
}

```

```

        SoapHelper.publishService(address, soapInterfaceClass,
                                   soapInstance);
    }
}

private Class<?> getSoapInterfaceClass(Class<?> soapClass) {
    // 获取 SOAP 实现类的第一个接口作为 SOAP 服务接口
    return soapClass.getInterfaces()[0];
}

private String getAddress(Class<?> soapClass) {
    String address;
    // 若 SOAP 注解的 value 属性不为空, 则获取当前值, 否则获取类名
    String soapValue = soapClass.getAnnotation(Soap.class).value();
    if (StringUtil.isEmpty(soapValue)) {
        address = soapValue;
    } else {
        address = getSoapInterfaceClass(soapClass).getSimpleName();
    }
    // 确保最前面只有一个 /
    if (!address.startsWith("/")) {
        address = "/" + address;
    }
    address = address.replaceAll("\\\\+", "/");
    return address;
}
}

```

我们在 `WebService` 注解中设置了 `loadOnStartup=0`, 表示该 Web 容器启动时会自动加载 Servlet, 即调用父类 `CXFNonSpringServlet` 的 `init` 方法。该方法中调用了子类 `SoapServlet` 的 `loadBus` 方法, 因此, 我们需要在 `loadBus` 方法里完成相关的初始化工作, 即初始化 CXF 总线与发布 SOAP 服务。

其他相关细节请参见代码中的注释, 需要注意的是, 当我们扫描出带有 `Soap` 注解的类时, 需要同时获取该类的接口, 而这个接口就是定义 SOAP 服务的接口。那么, 如果该实现类实现了多个接口怎么办? 我们在框架中默认只获取第一个接口作为 SOAP 服务接口。当然, 我们也可以为 `Soap` 注解扩展一个 `Class` 属性, 用于指定具体的 SOAP 服务接口类, 请自行实现。

至此，Smart SOAP Plugin 已基本完成。通过对 CXF 框架的封装，我们提供了一个基于注解的 SOAP 服务发布框架，同时也提供了一个便于使用的 SOAP 客户端工具。通过对 CXF 的学习，我们了解到它还可以发布并调用 REST 服务，下面我们再来实现另一个插件，用于发布并调用 REST 服务，该插件就是 Smart REST Plugin。

6. 支持 REST 服务

我们仍然按照开发 SOAP 插件的过程，先从如何使用开始，再到如何实现来讲解。

7. 发布 REST 服务

发布 REST 服务对于发布 SOAP 服务而言要相对简单一点，至少不需要单独定义一个服务接口。也就是说，只需要一个类就能将其发布为 REST 服务。我们可以直接通过 Java 原生提供的 JAX-RS API 来定义 REST 规则，并使用 Smart 提供的 Rest 注解与 Service 注解来发布 REST 服务。代码如下：

```
package org.smart4j.chapter5.rest;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.smart4j.chapter5.model.Customer;
import org.smart4j.chapter5.service.CustomerService;
import org.smart4j.framework.annotation.Inject;
import org.smart4j.framework.annotation.Service;
import org.smart4j.plugin.rest.Rest;

/**
 * 客户 REST 服务
 *
 * @author huangyong
 * @since 1.0.0
 */
@Rest
@Service
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
```

```
public class CustomerRestService {

    @Inject
    private CustomerService customerService;

    @GET
    @Path("/customer/{id}")
    public Customer getCustomer(@PathParam("id") long customerId) {
        return customerService.getCustomer(customerId);
    }
}
```

在 REST 类中使用 `Consumes` 注解定义输入的数据类型，并使用 `Produces` 定义输出的数据类型，可见输入与输出均为 JSON 格式。通过 `Path` 注解与 `PathParam` 注解来定义 REST 请求路径以及路径中的参数。

8. 调用 REST 服务

我们仍然通过一个单元测试来调用 REST 服务，查看测试是否通过，代码如下：

```
package org.smart4j.chapter5.soap;

import org.junit.Assert;
import org.junit.Test;
import org.smart4j.chapter5.model.Customer;
import org.smart4j.chapter5.rest.CustomerRestService;
import org.smart4j.plugin.rest.RestHelper;

/**
 * 客户 REST 服务单元测试
 *
 * @author huangyong
 * @since 1.0.0
 */
public class CustomerRestServiceTest {

    @Test
    public void getCustomerTest() {
        String wadl = "http://localhost:8080/rest/CustomerRestService";
```

```

        CustomerRestService customerRestService = RestHelper.createClient
            (wadl, CustomerRestService.class);
        Customer customer = customerRestService.getCustomer(1);
        Assert.assertNotNull(customer);
    }
}

```

这里不再使用 SOAP 服务的 WSDL 地址,而是使用 REST 服务的 WADL 地址,它是对 REST 服务的描述。

实际上,我们也可以在浏览器上直接输入以下地址来发送 REST 请求,查看 REST 服务所输出的 JSON 数据:

```
http://localhost:8080/rest/CustomerRestService/customer/1
```

9. 实现 REST 插件

第一步:添加 Maven 依赖。

与 SOAP 插件所需的 Maven 依赖相比,REST 插件需要 CXF 的 `cxf-rt-frontend-jaxrs` 构件。为了支持 JSONP,我们需要依赖 CXF 的 `cxf-rt-rs-extension-providers` 构件;为了支持 CORS 特性,我们需要依赖 CXF 的 `cxf-rt-rs-security-cors` 构件;最后,我们需要依赖 CXF 的 `cxf-rt-rs-client` 构件来开发 REST 客户端。为了确保 REST 请求能输出 JSON 数据,我们依赖了 `jackson-jaxrs-json-provider` 构件。具体的 Maven 依赖配置如下:

```

<!-- Servlet -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>

<!-- CXF -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxrs</artifactId>
    <version>${version.cxf}</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>

```

```

        <artifactId>cxfrt-rs-extension-providers</artifactId>
        <version>${version.cxf}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxfrt-rs-security-cors</artifactId>
        <version>${version.cxf}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxfrt-rs-client</artifactId>
        <version>${version.cxf}</version>
    </dependency>

    <!-- Jackson -->
    <dependency>
        <groupId>com.fasterxml.jackson.jaxrs</groupId>
        <artifactId>jackson-jaxrs-json-provider</artifactId>
        <version>2.5.2</version>
    </dependency>

    <!-- Smart Framework -->
    <dependency>
        <groupId>org.smart4j</groupId>
        <artifactId>smart-framework</artifactId>
        <version>1.0.0</version>
    </dependency>

```

其中，`${version.cxf}` 占位符需要在 `pom.xml` 里定义对应的属性，代码如下：

```

<properties>
    <version.cxf>3.0.4</version.cxf>
</properties>

```

第二步：定义 Rest 注解。

Rest 注解与 Soap 注解一样，只是名字不同而已，代码如下：

```

package org.smart4j.plugin.rest;

import java.lang.annotation.ElementType;

```

```

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

/**
 * REST 服务注解
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Rest {

    /**
     * 服务名
     */
    String value() default "";
}

```

第三步：编写 RestHelper 助手类。

RestHelper 与 SoapHelper 类似，只不过这里除了考虑日志以外，还支持 JSONP 与 CORS 特性，这些特性 CXF 都得到了很好的支持，代码如下：

```

package org.smart4j.plugin.rest;

import com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;
import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.interceptor.Interceptor;
import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.client.JAXRSClientFactory;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
import org.apache.cxf.jaxrs.provider.jsonp.JsonpInInterceptor;
import org.apache.cxf.jaxrs.provider.jsonp.JsonpPostStreamInterceptor;
import org.apache.cxf.jaxrs.provider.jsonp.JsonpPreStreamInterceptor;

```

```
import org.apache.cxf.message.Message;
import org.apache.cxf.rs.security.cors.CrossOriginResourceSharingFilter;
import org.smart4j.framework.helper.BeanHelper;

/**
 * REST 助手类
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RestHelper {

    private static final List<Object> providerList = new ArrayList<Object>();
    private static final List<Interceptor<? extends Message>> inInterce-
        ptorList = new ArrayList<Interceptor<? extends Message>>();
    private static final List<Interceptor<? extends Message>> outInterce-
        ptorList = new ArrayList<Interceptor<? extends Message>>();

    static {
        // 添加 JSON Provider
        JacksonJsonProvider jsonProvider = new JacksonJsonProvider();
        providerList.add(jsonProvider);
        // 添加 Logging Interceptor
        if (RestConfig.isLog()) {
            LoggingInInterceptor loggingInInterceptor = new LoggingIn-
                Interceptor();
            inInterceptorList.add(loggingInInterceptor);
            LoggingOutInterceptor loggingOutInterceptor = new LoggingOut-
                Interceptor();
            outInterceptorList.add(loggingOutInterceptor);
        }
        // 添加 JSONP Interceptor
        if (RestConfig.isJsonp()) {
            JsonpInInterceptor jsonpInInterceptor = new JsonpInInterce-
                ptor();
            jsonpInInterceptor.setCallbackParam(RestConfig.getJsonpFunc-
                tion());
            inInterceptorList.add(jsonpInInterceptor);
        }
    }
}
```

```

        JsonpPreStreamInterceptor jsonpPreStreamInterceptor = new
        JsonpPreStreamInterceptor();
        outInterceptorList.add(jsonpPreStreamInterceptor);
        JsonpPostStreamInterceptor jsonpPostStreamInterceptor = new
        JsonpPostStreamInterceptor();
        outInterceptorList.add(jsonpPostStreamInterceptor);
    }
    // 添加 CORS Provider
    if (RestConfig.isCors()) {
        CrossOriginResourceSharingFilter corsProvider = new CrossOrigin-
        ResourceSharingFilter();
        corsProvider.setAllowOrigins(RestConfig.getCorsOriginList());
        providerList.add(corsProvider);
    }
}

// 发布 REST 服务
public static void publishService(String wadl, Class<?> resourceClass) {
    JAXRSServerFactoryBean factory = new JAXRSServerFactoryBean();
    factory.setAddress(wadl);
    factory.setResourceClasses(resourceClass);
    factory.setResourceProvider(resourceClass, new SingletonResource-
    Provider(BeanHelper.getBean(resourceClass)));
    factory.setProviders(providerList);
    factory.setInInterceptors(inInterceptorList);
    factory.setOutInterceptors(outInterceptorList);
    factory.create();
}

// 创建 REST 客户端
public static <T> T createClient(String wadl, Class<? extends T>
resourceClass) {
    return JAXRSClientFactory.create(wadl, resourceClass, providerList);
}
}

```

其中，`RestConfig` 与 `SoapConfig` 类似，将相关配置项都集中在该类中，通过 `static` 方法获取。代码如下：

```
package org.smart4j.plugin.rest;

import java.util.Arrays;
import java.util.List;
import org.smart4j.framework.helper.ConfigHelper;
import org.smart4j.framework.util.StringUtil;

/**
 * 从配置文件中获取相关属性
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RestConfig {

    public static boolean isLog() {
        return ConfigHelper.getBoolean(RestConstant.LOG);
    }

    public static boolean isJsonp() {
        return ConfigHelper.getBoolean(RestConstant.JSONP);
    }

    public static String getJsonpFunction() {
        return ConfigHelper.getString(RestConstant.JSONP_FUNCTION);
    }

    public static boolean isCors() {
        return ConfigHelper.getBoolean(RestConstant.CORS);
    }

    public static List<String> getCorsOriginList() {
        String corsOrigin = ConfigHelper.getString(RestConstant.CORS_
            ORIGIN);
        return Arrays.asList(StringUtil.splitString(corsOrigin, ","));
    }
}
```

REST 插件相关的常量都放在 `RestConstant` 类中了，代码如下：

```
package org.smart4j.plugin.rest;

/**
 * REST 插件常量
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface RestConstant {

    String SERVLET_URL = "/rest/*";

    String LOG = "smart.plugin.rest.log";
    String JSONP = "smart.plugin.rest.jsonp";
    String JSONP_FUNCTION = "smart.plugin.rest.jsonp.function";
    String CORS = "smart.plugin.rest.cors";
    String CORS_ORIGIN = "smart.plugin.rest.cors.origin";
}
```

第四步：实现一个 `RestServlet`。

发布 REST 服务与发布 SOAP 服务基本类似，也需要继承 `CXFNonSpringServlet` 父类，并提供一个 `RestServlet`。不同的是，这里不需要通过实现类来获取接口，相对而言比较简单一些，代码如下：

```
package org.smart4j.plugin.rest;

import java.util.Set;
import javax.servlet.ServletConfig;
import javax.servlet.annotation.WebServlet;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.servlet.CXFNonSpringServlet;
import org.smart4j.framework.helper.ClassHelper;
import org.smart4j.framework.util.CollectionUtil;
import org.smart4j.framework.util.StringUtil;

/**
```

```
* REST Servlet
*
* @author huangyong
* @since 1.0.0
*/
@WebServlet(urlPatterns = RestConstant.SERVLET_URL, loadOnStartup = 0)
public class RestServlet extends CXFNonSpringServlet {

    @Override
    protected void loadBus(ServletConfig sc) {
        // 初始化 CXF 总线
        super.loadBus(sc);
        Bus bus = getBus();
        BusFactory.setDefaultBus(bus);
        // 发布 REST 服务
        publishRestService();
    }

    private void publishRestService() {
        // 遍历所有标注了 REST 注解的类
        Set<Class<?>> restClassSet = ClassHelper.getClassSetByAnnotation(
            Rest.class);
        if (CollectionUtil.isNotEmpty(restClassSet)) {
            for (Class<?> restClass : restClassSet) {
                // 获取 REST 地址
                String address = getAddress(restClass);
                // 发布 REST 服务
                RestHelper.publishService(address, restClass);
            }
        }
    }

    private String getAddress(Class<?> restClass) {
        String address;
        // 若 REST 注解的 value 属性不为空，则获取当前值，否则获取类名
        String value = restClass.getAnnotation(Rest.class).value();
        if (StringUtil.isEmpty(value)) {
            address = value;
        }
    }
}
```

```
    } else {  
        address = restClass.getSimpleName();  
    }  
    // 确保最前面只有一个 /  
    if (!address.startsWith("/")) {  
        address = "/" + address;  
    }  
    address = address.replaceAll("\\\\+", "/");  
    return address;  
}  
}
```

现在一个与 SOAP 插件类似的 REST 插件就开发完毕了,实现过程与 SOAP 插件完全一致。这里只是开发了一个框架原型,我们可以根据实际项目的需要,对该框架进行必要的扩展。

也许您会思考一个问题:能否将 SOAP 插件与 REST 插件合并为一个插件?这个就仁者见仁智者见智了,我认为还是有必要分开的,因为这是两种类型的 Web 服务,最好能够提供不同的插件,让开发者自行选择所需的插件,并不将一些用不到的特性包含到项目中。

5.8 总结

在本章中,我们对框架做出了一定的扩展,对 Action 方法参数进行了优化,若不存在任何请求参数,则可在 Action 方法中省略 Param 参数。提供了文件上传特性,并支持文件批量上传。使用 ServletHelper 来封装 Servlet API,在 Controller 或 Service 中可直接调用 ServletHelper 封装的 Servlet 常用 API。此外,我们也封装了 Shiro 框架,开发了一个简单的安全控制模块,并通过插件的方式集成到 Smart 框架中。同样也通过插件的方式封装了 CXF 框架,针对 SOAP 与 REST 服务开发了对应的 Smart 插件。

对于 Smart 框架现在只是一个开始,我们的目标是打造一款轻量级的 Web 开发框架,并提供大量的插件扩展机制,未来还有很长的路要走。

附录 A

Maven 快速入门

1. 什么是 Maven

Jason Van Zyl, 在 Java 十大风云人物排行榜上或许会看到他。

他就是 Maven 的创始人, 人们都尊称他为“Maven 他爸”。

毋庸置疑, Jason 也是一个秃顶。James Gosling、Rod Johnson、Gavin King, 都有这一个共性。

您曾经是否遇到这些问题:

我们要开发一个 Java 项目, 为了保证编译通过, 我们会到处去寻找 jar 包。当编译通过了, 在运行的时候, 却发现“ClassNotFoundException”, 难道还差 jar 包吗? 再去找找吧……

每个 Java 项目的目录结构都没有一个统一的标准, 配置文件到处都是, 单元测试代码到底应该放在哪里也没有一个权威的规范。

我们可以使用 Ant 作为项目构建工具, 它可以自动化地完成编译、测试、打包等任务, 确实为我们省了不少事, 但编写 Ant 的 XML 脚本绝非是一件轻松的事情。

有了 Maven, 以上这一切都不再是问题了。

Jason 就是 Java 开发规范的“救世主”。他给我们带来了一种全新的项目构建方式, 让我们的开发工作更加高效。

不仅如此, Jason 还是一名“野心家”, 他不仅希望每个 Java 开发者都能使用他定义的规范, 还要我们都从他家里去获取 jar 包(他家就是 Maven 中央仓库), 我们只需告诉他, 我们想要的 jar 包具体在什么位置即可(这个位置就是 Maven 坐标)。

看来 Jason 要做的是两件事情:

- 统一开发规范与工具;
- 统一管理 jar 包。

这两件事情他都做到了, 而且还做了更多的事情。

工欲善其事, 必先利其器。咱们也来玩玩 Maven 吧。不过先得去下载一个 Maven。

2. 安装 Maven

Maven 是 Apache 基金会的顶级项目, 一般情况下, 被 Apache 看中的都是精品。

我们可以从 <http://maven.apache.org/> 下载 Maven 开发包, 其实就是一个压缩包, 下载完毕后, 解压一下, 配置一下环境变量就可以用了。

假设我们刚刚下载了一个 apache-maven-3.1.1-bin.zip 文件, 现在将其解压到 D:/tool 目录下。我们将解压后的目录重命名为 Maven, 这样 Maven 的根目录就是 D:/tool/maven 了。

有两个环境变量可以配置:

- `M2_HOME = D:/tool/maven`
- `MAVEN_OPTS = -Xms128m -Xmx512m`

以上 `M2_HOME` 是必须要配置的，如果想让 `Maven` 跑得更快点，可以根据自己的情况来设置 `MAVEN_OPTS`。此外，还需修改 `Path` 环境变量，在末尾添加`%M2_HOME%\bin`。

现在我们可以打开 `cmd`，输入：

```
mvn -v
```

您一定会看到一些信息，`Maven` 安装成功。

在使用 `Maven` 之前，很有必要了解一下 `Maven` 到底是怎样管理 `jar` 包的，这就是 `Maven` 仓库要干的活了。

3. 了解 Maven 仓库

使用 `Maven` 给我们带来的最直接的帮助就是 `jar` 包得到了统一管理，那么这些 `jar` 包存放在哪里呢？它们就在本地仓库中，位于 `C:.m2` 目录下（当然也可以修改这个默认地址）。

实际上可将本地仓库理解为“缓存”，因为项目首先会从本地仓库中获取 `jar` 包，当无法获取指定 `jar` 包的时候，本地仓库会从远程仓库（或中央仓库）中下载 `jar` 包，并放入本地仓库中以备将来使用。这个远程仓库是 `Maven` 官方提供的，可通过 <http://search.maven.org/> 来访问。这样一来，本地仓库会随着项目的积累越来越大。通过图 A-1 可以清晰地表达项目、本地仓库、远程仓库之间的关系。

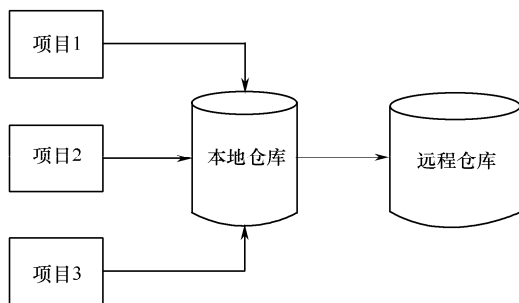


图 A-1 Maven 仓库

这个结构是否与 `Git` 的本地仓库与远程仓库有异曲同工之妙呢？

既然 `Maven` 安装了，那么本地仓库也就有了，下面我们就一起来创建一个 `Maven` 项目。

4. 创建 Maven 项目

我们创建一个 `Java Web` 项目，只需在 `cmd` 中输入：

```
mvn archetype:generate
```

随后 Maven 将下载 Archetype 插件及其所有的依赖插件，这些插件其实都是 jar 包，它们存放在 Maven 本地仓库中。

在 cmd 中，我们会看到几百个 Archetype（原型），可将它理解为项目模板，我们需要从中选择一个。

我们的目标是创建 Java Web 项目，所以可以选择 maven-archetype-webapp（可以在 cmd 中进行模糊搜索），随后 Maven 会与用户进行一些对话，Maven 想知道以下信息：

- 项目 Archetype Version（原型版本号）是什么——可选择 1.0 版本；
- 项目 groupId（组织名）是什么——可输入 org.smart4j；
- 项目 artifactId（构件名）是什么——可输入 smart-demo；
- 项目 version（版本号）是什么——可输入 1.0；
- 项目 package（包名）是什么——可输入 org.smart4j.demo。

以上这种方式称为 Interactive Mode（交互模式）。

如果您是一位高效人士，或许觉得这样的交互过于烦琐，那么也可以尝试仅使用一条命名来完成同样的事情：

```
mvn archetype:generate -DinteractiveMode=false -DarchetypeArtifactId=maven-archetype-webapp -DgroupId=org.smart4j -DartifactId=smart-demo -Dversion=1.0
```

以上这种方式称为 Batch Mode（批处理模式）。

当然，还有第三种选择，使用 IDE 来创建 Maven 项目，可以使用 Eclipse、NetBeans、IDEA 来创建 Maven 项目，操作过程应该是非常简单的。

也可以使用 IDEA 直接打开一个 Maven 项目，只需要选择 File→Open→pom.xml，那么就可以在 IDEA 中开发 Maven 项目了，如图 A-2 所示。

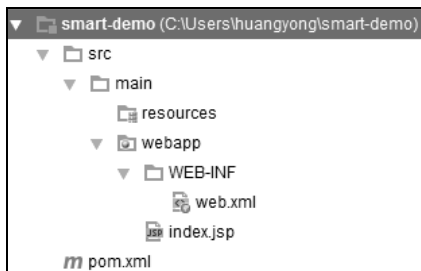


图 A-2 使用 IDEA 打开 Maven 项目

其实这个目录结构还不太完备，我们需要手工添加几个目录上去，最终的目录结构如图 A-3 所示。

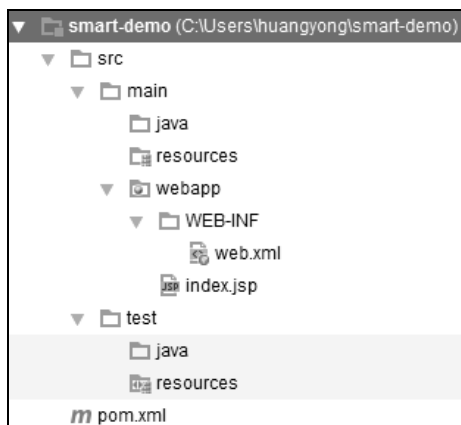


图 A-3 完备的 Maven 目录结构

我们手工创建了三个目录：

- `src/main/java`;
- `src/test/java`;
- `src/test/resources`。

有必要稍微解释一下这个 Maven 目录规范：

- `main` 目录下是项目的主要代码，`test` 目录下存放测试相关的代码。
- 编译输出后的代码会放在 `target` 目录下（该目录与 `src` 目录在同一级别下，这里没有显示出来）。
- `java` 目录下存放 Java 代码，`resources` 目录下存放配置文件。
- `webapp` 目录下存放 Web 应用相关代码。
- `pom.xml` 是 Maven 项目的配置文件。

其中 `pom.xml` 称为 Project Object Model（项目对象模型），它用于描述整个 Maven 项目，所以也称为 Maven 描述文件。

可见 `pom.xml` 才是理解 Maven 的关键点，很有必要看看它到底长什么样。

5. 理解 POM 结构

当打开自动生成的 `pom.xml` 时，或许会感觉到可读性不太好，有必要做一下格式化，经过整理后是这样的：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.smart4j</groupId>
  <artifactId>smart-demo</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>

  <name>smart-demo Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <finalName>smart-demo</finalName>
  </build>

</project>
```

从上往下简要说明一下：

- **modelVersion** 是 POM 的版本号，现在都是 4.0.0 的，必须得有，但不需要修改；
- **groupId**、**artifactId**、**version** 分别表示 Maven 项目的组织名、构件名、版本号，它们三个合起来就是 Maven 坐标，根据这个坐标可以在 Maven 仓库中对应唯一的 Maven 构件；
- **packaging** 表示该项目的打包方式，war 表示打包为 war 文件，默认为 jar，表示打包为

jar 文件;

- `name`、`url` 表示该项目的名称与 URL 地址，意义不大，可以省略;
- `dependencies` 定义该项目的依赖关系，其中每一个 `dependency` 对应一个 Maven 项目，可见 Maven 坐标再次出现，还多了一个 `scope`，表示作用域（下面会描述）;
- `build` 表示与构建相关的配置，这里的 `finalName` 表示最终构建后的名称 `smart-demo.war`，这里的 `finalName` 还可以使用另一种方式来定义（下面会描述）。

如果用树形图来查看 `pom.xml` 文件，那么会更加清晰，如图 A-4 所示。

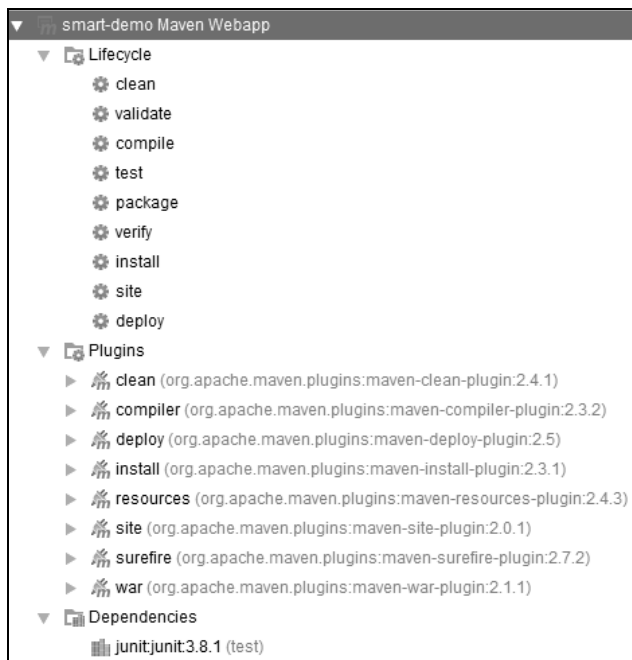


图 A-4 在 IDEA 中查看 POM 结构

除了项目的基本信息（Maven 坐标、打包方式等）外，每个 `pom.xml` 都应该包括：

- Lifecycle（生命周期）;
- Plugins（插件）;
- Dependencies（依赖）。

Lifecycle 是项目构建的生命周期，它包括 9 个 Phase（阶段）。

Maven 是一个核心加上多个插件的架构，而这些插件提供了一系列非常重要的功能，这些插件会在许多阶段里发挥重要作用，如表 A-1 所示。

表 A-1 Maven 中的插件

阶 段	插 件	作 用
clean	clean	清理自动生成的文件，也就是 target 目录
validate	由 Maven 核心负责	验证 Maven 描述文件是否有效
compile	compiler、resources	编译 Java 源码
test	compiler、surefire、resources	运行测试代码
package	war	项目打包，就是生成构件包，也就是打 war 包
verify	由 Maven 核心提供	验证构件包是否有效
install	install	将构件包安装到本地仓库
site	site	生成项目站点，就是一堆静态网页文件，包括 JavaDoc
deploy	deploy	将构件包部署到远程仓库

表 A-1 中所出现的插件名称实际上是插件的别名（或称为前缀），比如 `compiler` 实际上是 `org.apache.maven.plugins:maven-compiler-plugin:2.3.2`，这个才是 Maven 插件的完全名称。

每个插件又包括了一些列的 Goal（目标），以 `compiler` 插件为例，它包括以下目标：

- `compiler:help` 用于显示 `compiler` 插件的使用帮助；
- `compiler:compile` 用于编译 main 目录下的 Java 代码；
- `compiler:testCompile` 用于编译 test 目录下的 Java 代码。

可见插件目标才是具体干活的“人”，一个插件包括了一个或多个目标，一个阶段可由零个或多个插件来提供支持。

我们可以在 `pom.xml` 中定义一些系列的项目依赖（构件包），每个构件包都会有一个 Scope（作用域），它表示该构件包在什么时候起作用，包括以下五种：

- `compile`——默认作用域，在编译、测试、运行时有效；
- `test`——测试时有效；
- `runtime`——测试、运行时有效；
- `provided`——编译、测试时有效，但在运行时无效；
- `system`——与 `provided` 类似，但依赖于系统资源。

以上内容可用一张矩阵表格来表示，如表 A-2 所示。

表 A-2 作用域

作用域	编译时有效	测试时有效	运行时有效
compile	√	√	√
test	—	√	—
runtime	—	√	√
provided	√	√	—
system	√	√	—

如果想开发一个 Smart 应用，可参考如下 pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <smart.version>1.0.0</smart.version>
    </properties>

    <groupId>org.smart4j</groupId>
    <artifactId>smart-demo</artifactId>
    <version>1.0</version>
    <packaging>war</packaging>

    <dependencies>
        <!-- JUnit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
        <!-- MySQL -->
        <dependency>
            <groupId>mysql</groupId>
```

```

        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.25</version>
        <scope>runtime</scope>
    </dependency>
    <!-- Servlet -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
        <scope>runtime</scope>
    </dependency>
    <!-- Smart -->
    <dependency>
        <groupId>org.smart4j</groupId>
        <artifactId>smart-framework</artifactId>
        <version>${smart.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <!-- Compile -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.5.1</version>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
    </plugins>

```

```
<!-- Test -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.15</version>
  <configuration>
    <skipTests>true</skipTests>
  </configuration>
</plugin>
<!-- Package -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.4</version>
  <configuration>
    <warName>${project.artifactId}</warName>
  </configuration>
</plugin>
<!-- Tomcat -->
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
</plugin>
</plugins>
</build>

</project>
```

以上 pom.xml 大致解释一下：

- 我们可使用 **properties** 来定义一些配置属性，例如 **project.build.sourceEncoding**（项目构建源码编码方式），可设置为 UTF-8，防止中文乱码，也可定义相关构件包版本号，例如 **smart.version**，便于日后统一升级。
- 建议使用最新版本的 JUnit，通过 Archetype 自动生成的 JUnit 太老了（3.8.1），可改为最新版（4.11）。
- 因为没必要使用 MySQL 客户端的 API，它仅仅在运行时有效，所以我们将 MySQL 构件包的作用域设置为 **runtime**。

- 因为我们只想在代码中使用 Servlet API，而不想将它所对应的 jar 包放入 WEB-INF 的 lib 目录下，所以我们可设置 Servlet 构件包的作用域为 provided。
- 为了保证在 JDK 1.6 下运行，我们可配置 maven-compiler-plugin 插件，设置输入源码为 1.6，编译输出的字节码也为 1.6。
- 如果想跳过测试，可配置 maven-surefire-plugin 插件，将 skipTests 设置为 true。
- 如果想配置生成的 war 包为 artifactId，可修改 maven-war-plugin 插件，将 warName 修改为 \${project.artifactId}，这样就无须再配置 finalName 了。
- 如果想通过 Maven 将应用部署到 Tomcat 中，可使用 tomcat7-maven-plugin 插件，通过 mvn tomcat7:run-war 命令来运行 war 包。

6. 使用 Maven 命令

前面我们已经使用了几个 Maven 命令，例如 mvn archetype:generate、mvn tomcat7:run-war 等。其实，可使用两种不同的方式来执行 Maven 命令：

- 方式一——mvn<插件>:<目标>[参数];
- 方式二——mvn<阶段>。

现在我们接触到的都是第一种方式，而第二种方式才是我们日常中使用最频繁的，例如：

- mvn clean——清空输出目录（即 target 目录）；
- mvn compile——编译源代码；
- mvn package——生成构件包（一般为 jar 包或 war 包）；
- mvn install——将构件包安装到本地仓库；
- mvn deploy——将构件包部署到远程仓库。

执行 Maven 命令需要注意的是：必须在 Maven 项目的根目录处执行，也就是当前目录下一定存在一个名为 pom.xml 的文件。

Maven 使 Java 开发更加规范化与自动化，其实 Maven 那点事远远不止这些，如果掌握了以上这些基础知识，再去学习 Maven 的高级特性，一定会是一件非常轻松愉快的事情。

附录 B

将构件发布到 Maven 中央 仓库

说到中央仓库，不得不说 Sonatype 这家公司，因为中央仓库就是这家公司“砸钱”搞的，并且免费向全球所有的 Java 开发者提供构件托管服务，这对于我们而言，简直就是“福利”啊！

Sonatype 官网：<http://www.sonatype.org/>。

对于新手而言，第一次将构件发布到中央仓库，真的不是一件非常轻松的事情，所以非常有必要把些步骤总结出来，这样可以节省我们的时间，用来做更多重要的事情。

具体的操作步骤如下所述。

1. 第一步：注册一个 Sonatype 用户。

注册地址：<https://issues.sonatype.org/secure/Signup!default.jspx>。

这里的用户名与密码是非常重要的，后面会用到，一定要保存好。

此外，Sonatype 还提供了一个名为 OSS 的系统：<https://oss.sonatype.org>。

在 OSS 中可以查询到全世界已发布的构件，当然它还有另外一个作用，后面会提到。

2. 第二步：创建一个 Issue

Issue 地址：<https://issues.sonatype.org/secure/CreateIssue.jspa?issuetype=21&pid=10134>。

此时相当于提交了一个申请。其中最重要的信息就是 `groupId`，它一般是域名的倒排方式，而且最好能有自己独立的域名，笔者就购买了一个名为 `smart4j.org` 的域名，因此，`groupId` 就是 `org.smart4j` 了。

这样一来，Smart 在中央仓库里就可以申请到名为 `org.smart4j` 的 `groupId` 了。

3. 第三步：等待 Issue 审批通过

一般需要 1~2 天时间，需要耐心等待，审批通过后会发邮件通知。此外，在自己提交的 Issue 下面会看到 Sonatype 工作人员的回复。

4. 第四步：使用 GPG 生成密钥对

如果是 Windows 操作系统，需要下载 Gpg4win 软件来生成密钥对。建议下载 Gpg4win-Vanilla 版本，因为它仅包括 GnuPG，这个工具才是我们所需要的。

安装 GPG 软件后，打开命令行窗口，依次做以下操作：

(1) 查看是否安装成功。

```
gpg-version
```

能够显示 GPG 的版本信息，说明安装成功了。

(2) 生成密钥对。

```
gpg--gen-key
```

此时需要输入姓名、邮箱等字段,其他字段可使用默认值。此外,还需要输入一个 **Passphrase**,相当于一个密钥库的密码,一定不要忘了,也不要告诉别人,最好记下来,因为后面会用到。

(3) 查看公钥。

```
gpg--list-keys
```

输出如下信息:

```
C:/Users/huangyong/AppData/Roaming/gnupg/pubring.gpg
-----
pub   2048R/82DC852E 2014-12-12
uid           xxx <xxx@xxx.com>
sub   2048R/3ACA39AF 2014-12-12
```

可见这里的公钥的 ID 是 82DC852E,很明显是一个 16 进制的数字,马上就会用到。

(4) 将公钥发布到 PGP 密钥服务器。

```
gpg --keyserver hkp://pool.sks-keyservers.net --send-keys 82DC852E
```

可使用本地的私钥来对上传构件进行数字签名,而下载该构件的用户可通过上传的公钥来验证签名。也就是说,可以验证这个构件是否由本人上传的,因为有可能该构件被坏人给篡改了。

(5) 查询公钥是否发布成功。

```
gpg --keyserver hkp://pool.sks-keyservers.net --recv-keys 82DC852E
```

实际上就是从 key server 上通过公钥 ID 来接收公钥,此外,也可以到 sks-keyservers.net 上通过公钥 ID 去查询。

5. 第五步:修改 Maven 配置文件

需要修改的 Maven 配置文件包括 `setting.xml` (全局级别)与 `pom.xml` (项目级别)。

(1) `setting.xml`。

```
<settings>

...

<servers>
```

```
<server>
  <id>oss</id>
  <username>用户名</username>
  <password>密码</password>
</server>
</servers>

...

</settings>
```

使用自己注册的 Sonatype 账号的用户名与密码来配置以上 server 信息。

(2) pom.xml。

```
<project>

...

<name>smart</name>
<description>Smart is a lightweight Java Web Framework.</description>
<url>http://www.smart4j.org/</url>

<licenses>
  <license>
    <name>The Apache Software License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
  </license>
</licenses>

<developers>
  <developer>
    <name>xxx</name>
    <email>xxx@xxx.com</email>
  </developer>
</developers>
```

```
<scm>
  <connection>scm:git:git@git.oschina.net:huangyong/smart.git
</connection>
  <developerConnection>scm:git:git@git.oschina.net:huangyong/
smart.git</developerConnection>
  <url>git@git.oschina.net:huangyong/smart.git</url>
</scm>

...

<profiles>
  <profile>
    <id>release</id>
    <build>
      <plugins>
        <!-- Source -->
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-source-plugin</artifactId>
          <version>2.2.1</version>
          <executions>
            <execution>
              <phase>package</phase>
              <goals>
                <goal>jar-no-fork</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <!-- Javadoc -->
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-javadoc-plugin</artifactId>
          <version>2.9.1</version>
```

```
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>jar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    <!-- GPG -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-gpg-plugin</artifactId>
        <version>1.5</version>
        <executions>
            <execution>
                <phase>verify</phase>
                <goals>
                    <goal>sign</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
<distributionManagement>
    <snapshotRepository>
        <id>oss</id>
        <url>https://oss.sonatype.org/content/repositories/
            snapshots/</url>
    </snapshotRepository>
    <repository>
        <id>oss</id>
        <url>https://oss.sonatype.org/service/local/staging/
```

```
        deploy/maven2/</url>
    </repository>
</distributionManagement>
</profile>
</profiles>

...

</project>
```

注意：以上 pom.xml 必须包括：name、description、url、licenses、developers、scm 等基本信息。此外，使用了 Maven 的 profile 功能，只有在 release 的时候才源码包以及创建文档包、使用 GPG 进行数字签名。snapshotRepository 与 repository 中的 id 一定要与 setting.xml 中 server 的 id 保持一致。

6. 第六步：上传构件到 OSS 中

当执行以下 Maven 命令时，会自动弹出一个对话框，提示需要输入上面提到的 Passphrase，它就是通过 GPG 密钥对的密码，只有用户自己才知道。随后会看到大量的 upload 信息，而且速度比较慢，经常会 timeout，需要反复尝试。

```
mvn clean deploy -P release
```

注意：此时上传的构件并未正式发布到中央仓库中，只是部署到 OSS 中了，下面才是真正的发布。

7. 第七步：在 OSS 中发布构件

在 OSS 中，使用自己的 Sonatype 账号登录后，可在 Staging Repositories 中查看刚才已上传的构件，这些构件目前放在 Staging 仓库中，可进行模糊查询，快速定位到自己的构件。此时，该构件的状态为 Open，需要勾选它，然后单击 Close 按钮。接下来系统会自动验证该构件是否满足指定要求，当验证完毕后，状态会变为 Closed。最后，单击 Release 按钮来发布该构件。

8. 第八步：通知 Sonatype“构件已成功发布”

需要在曾经创建的 Issue 下面回复一条“构件已成功发布”的评论，这是为了通知 Sonatype 的工作人员为需要发布的构件做审批，发布后会关闭该 Issue。

9. 第九步：等待构件审批通过

没错，还是要等，也许又是 1~2 天。同样，当审批通过后，将会收到邮件通知。

10. 第十步：从中央仓库中搜索构件

最后就可以到中央仓库中搜索到自己发布的构件了。

中央仓库搜索网站：<http://search.maven.org/>。

至此，Smart 构件已成功发布到中央仓库，现在可在代码中直接配置 Smart 依赖了。

例如，依赖 Smart 框架，可以这样配置：

```
<dependency>
  <groupId>org.smart4j</groupId>
  <artifactId>smart-framework</artifactId>
  <version>1.0.0</version>
</dependency>
```

参考资料

Choosing your Coordinates

<https://docs.sonatype.org/display/Repository/Choosing+your+Coordinates>

Sonatype OSS Maven Repository Usage Guide

<https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide>

How To Generate PGP Signatures With Maven

<https://docs.sonatype.org/display/Repository/How+To+Generate+PGP+Signatures+With+Maven>
#HowToGeneratePGPSignaturesWithMaven-MavenGPGPlugin

