



## 第 12 章

# 使用 Maven 构建 Web 应用

### 本章内容

- ☐ Web 项目的目录结构
- ☐ account-service
- ☐ account-web
- ☐ 使用 jetty-maven-plugin 进行测试
- ☐ 使用 Cargo 实现自动化部署
- ☐ 小结

到目前为止，本书讨论的只有打包类型为 JAR 或者 POM 的 Maven 项目。但在现今的互联网时代，我们创建的大部分应用程序都是 Web 应用，在 Java 的世界中，Web 项目的标准打包方式是 WAR。因此本章介绍一个 WAR 模块——account-web，它也来自于本书的账户注册服务背景案例。在介绍该模块之前，本章还会先实现 account-service。此外，还介绍如何借助 jetty-maven-plugin 来快速开发和测试 Web 模块，以及使用 Cargo 实现 Web 项目的自动化部署。

## 12.1 Web 项目的目录结构

我们都知道，基于 Java 的 Web 应用，其标准的打包方式是 WAR。WAR 与 JAR 类似，只不过它可以包含更多的内容，如 JSP 文件、Servlet、Java 类、web.xml 配置文件、依赖 JAR 包、静态 web 资源（如 HTML、CSS、JavaScript 文件）等。一个典型的 WAR 文件会有如下目录结构：

```
-war/
+ META-INF/
+ WEB-INF/
| + classes/
| | + ServletA.class
| | + config.properties
| | + ...
| |
| + lib/
| | + dom4j-1.4.1.jar
| | + mail-1.4.1.jar
| | + ...
| |
| + web.xml
|
+ img/
|
+ css/
|
+ js/
|
+ index.html
+ sample.jsp
```

一个 WAR 包下至少包含两个子目录：META-INF 和 WEB-INF。前者包含了一些打包元数据信息，我们一般不去关心；后者是 WAR 包的核心，WEB-INF 下必须包含一个 Web 资源表述文件 web.xml，它的子目录 classes 包含所有该 Web 项目的类，而另一个子目录 lib 则包含所有该 Web 项目的依赖 JAR 包，classes 和 lib 目录都会在运行的时候被加入到 Classpath 中。除了 META-INF 和 WEB-INF 外，一般的 WAR 包都会包含很多 Web 资源，例如你往往可以在 WAR 包的根目录下看到很多 html 或者 jsp 文件。此外，还能看到一些文件夹如 img、css 和 js，它们会包含对应的文件供页面使用。

同任何其他 Maven 项目一样，Maven 对 Web 项目的布局结构也有一个通用的约定。不过首先要记住的是，用户必须为 Web 项目显式指定打包方式为 war，如代码清单 12-1 所示。

代码清单 12-1 显式指定 Web 项目的打包方式为 war

```
<project>
...
<groupId>com.juvenxu.mvnbook</groupId>
<artifactId>sample-war</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
...
</project>
```

如果不显式地指定 packaging，Maven 会使用默认的 jar 打包方式，从而导致无法正确打包 Web 项目。

Web 项目的类及资源文件同一般 JAR 项目一样，默认位置都是 src/main/java/ 和 src/main/resources，测试类及测试资源文件的默认位置是 src/test/java/ 和 src/test/resources/。Web 项目比较特殊的地方在于：它还有一个 Web 资源目录，其默认位置是 src/main/webapp/。一个典型的 Web 项目的 Maven 目录结构如下：

```
+ project
|
+ pom.xml
|
+ src/
  + main/
    + java/
      || + ServletA.java
      || + ...
      ||
    + resources/
      || + config.properties
      || + ...
      ||
    + webapp/
      | + WEB-INF/
      | | + web.xml
      | |
      | + img/
      | |
      | + css/
      | |
      | + js/
      | +
      | + index.html
      | + sample.jsp
      |
  + test/
    + java/
    + resources/
```



在 `src/main/webapp/` 目录下, 必须包含一个子目录 `WEB-INF`, 该子目录还必须要包含 `web.xml` 文件。`src/main/webapp` 目录下的其他文件和目录包括 `html`、`jsp`、`css`、`JavaScript` 等, 它们与 `WAR` 包中的 `Web` 资源完全一致。

在使用 Maven 创建 Web 项目之前, 必须首先理解这种 Maven 项目结构和 `WAR` 包结构的对应关系。有一点需要注意的是, `WAR` 包中有一个 `lib` 目录包含所有依赖 JAR 包, 但 Maven 项目结构中并没有这样一个目录, 这是因为依赖都配置在 `POM` 中; Maven 在用 `WAR` 方式打包的时候会根据 `POM` 的配置从本地仓库复制相应的 JAR 文件。

## 12.2 account-service

本章将完成背景案例项目, 读者可以回顾第 4 章, 除了之前实现的 `account-email`、`account-persist` 和 `account-captcha` 之外, 该项目还包括 `account-service` 和 `account-web` 两个模块。其中, `account-service` 用来封装底层三个模块的细节, 并对外提供简单的接口, 而 `account-web` 仅包含一些涉及 `Web` 的相关内容, 如 `Servlet` 和 `JSP` 等。

### 12.2.1 account-service 的 POM

`account-service` 用来封装 `account-email`、`account-persist` 和 `account-captcha` 三个模块的细节, 因此它肯定需要依赖这三个模块。`account-service` 的 `POM` 内容如代码清单 12-2 所示。

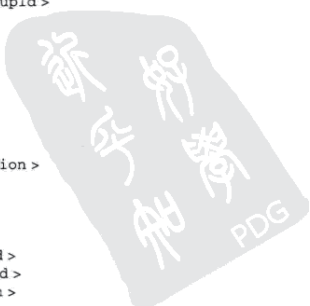
代码清单 12-2 account-service 的 POM

```
<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>
  <parent>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>

  <artifactId>account-service</artifactId>
  <name>Account Service</name>

  <properties>
    <greenmail.version>1.3.1b</greenmail.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-email</artifactId>
      <version>${project.version}</version>
    </dependency>
```



```

<dependency>
  <groupId> ${project.groupId} </groupId>
  <artifactId> account-persist </artifactId>
  <version> ${project.version} </version>
</dependency>
<dependency>
  <groupId> ${project.groupId} </groupId>
  <artifactId> account-captcha </artifactId>
  <version> ${project.version} </version>
</dependency>
<dependency>
  <groupId> junit </groupId>
  <artifactId> junit </artifactId>
</dependency>
<dependency>
  <groupId> com.icegreen </groupId>
  <artifactId> greenmail </artifactId>
  <version> ${greenmail.version} </version>
  <scope> test </scope>
</dependency>
</dependencies>

<build>
  <testResources>
    <testResource>
      <directory> src/test/resources </directory>
      <filtering> true </filtering>
    </testResource>
  </testResources>
</build>
</project>

```

与其他模块一样，account-service 继承自 account-parent，它依赖于 account-email、account-persist 和 account-captcha 三个模块。由于是同一项目中的其他模块，groupId 和 version 都完全一致，因此可以使用 Maven 属性 `${project.groupId}` 和 `${project.version}` 进行替换，这样可以在升级项目版本的时候减少更改的数量。项目的其他配置如 junit 和 greenmail 依赖，以及测试资源目录过滤配置，都是为了单元测试。前面的章节已经介绍过，这里不再赘述。

## 12.2.2 account-service 的主代码

account-service 的目的是封装下层细节，对外暴露尽可能简单的接口。先看一下这个接口是怎样的，见代码清单 12-3。

代码清单 12-3 AccountService.java

```

package com.juvenxu.mvnbook.account.service;

public interface AccountService
{
    String generateCaptchaKey()
        throws AccountServiceException;
}

```

```
byte[] generateCaptchaImage( String captchaKey )
    throws AccountServiceException;

void signUp( SignUpRequest signUpRequest )
    throws AccountServiceException;

void activate( String activationNumber )
    throws AccountServiceException;

void login( String id, String password )
    throws AccountServiceException;
}
```

正如 4.3.1 节介绍的那样，该接口提供 5 个方法。`generateCaptchaKey()` 用来生成一个验证码的唯一标识符。`generateCaptchaImage()` 根据这个标识符生成验证码图片，图片以字节流的方式返回。用户需要使用 `signUp()` 方法进行注册，注册信息使用 `SignUpRequest` 进行封装，这个 `SignUpRequest` 类是一个简单的 POJO，它包含了注册 ID、email、用户名、密码、验证码标识、验证码值等信息<sup>⑨</sup>。注册成功之后，用户会得到一个激活链接，该链接包含了一个激活码，这个时候用户需要使用 `activate()` 方法并传入激活码以激活账户。最后，`login()` 方法用来登录。

下面来看一下该接口的实现类 `AccountServiceImpl.java`。首先它需要使用 3 个底层模块的服务，如代码清单 12-4 所示。

代码清单 12-4 AccountServiceImpl.java 第 1 部分

```
public class AccountServiceImpl
    implements AccountService
{
    private AccountPersistService accountPersistService;

    private AccountEmailService accountEmailService;

    private AccountCaptchaService accountCaptchaService;

    public AccountPersistService getAccountPersistService()
    {
        return accountPersistService;
    }

    public void setAccountPersistService( AccountPersistService accountPersistService )
    {
        this.accountPersistService = accountPersistService;
    }
    ...
}
```

⑨ 由于篇幅的原因，这里不再给出源代码，有兴趣的读者可以自行下载并查看本书源码。

三个私有变量来自 account-persist、account-email 和 account-captcha 模块，它们都有各自的 get() 和 set() 方法，并且通过 Spring 注入。

AccountServiceImpl.java 借助 accountCaptchaService 实现验证码的标识符生成及验证码图片生成，如代码清单 12-5 所示。

代码清单 12-5 AccountServiceImpl.java 第 2 部分

---

```

public byte[] generateCaptchaImage( String captchaKey )
    throws AccountServiceException
{
    try
    {
        return accountCaptchaService.generateCaptchaImage( captchaKey );
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to generate Captcha Image.", e );
    }
}

public String generateCaptchaKey()
    throws AccountServiceException
{
    try
    {
        return accountCaptchaService.generateCaptchaKey();
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to generate Captcha key.", e );
    }
}

```

---

稍微复杂一点的是 signUp() 方法的实现，见代码清单 12-6。

代码清单 12-6 AccountServiceImpl.java 第 3 部分

---

```

private Map < String, String > activationMap = new HashMap < String, String > ();

public void signUp( SignUpRequest signUpRequest )
    throws AccountServiceException
{
    try
    {
        if (!signUpRequest.getPassword().equals(signUpRequest.getConfirmPassword()))
        {
            throw new AccountServiceException( "2 passwords do not match." );
        }

        if (!accountCaptchaService
            .validateCaptcha(signUpRequest.getCaptchaKey(),signUpRequest.
            getCaptchaValue()))
        {

```

---

```

        throw new AccountServiceException( "Incorrect Captcha. " );
    }

    Account account = new Account();
    account.setId( signUpRequest.getId() );
    account.setEmail( signUpRequest.getEmail() );
    account.setName( signUpRequest.getName() );
    account.setPassword( signUpRequest.getPassword() );
    account.setActivated( false );

    accountPersistService.createAccount( account );

    String activationId = RandomGenerator.getRandomString();

    activationMap.put( activationId, account.getId() );

    String link = signUpRequest.getActivateServiceUrl().endsWith( "/" ) ? signUpRequest.getActivateServiceUrl()
        + activationId : signUpRequest.getActivateServiceUrl() + "?key = " + activationId;

    accountEmailService.sendMail( account.getEmail(), "Please Activate Your Account", link );
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to validate captcha.", e );
    }
    catch ( AccountPersistException e )
    {
        throw new AccountServiceException( "Unable to create account.", e );
    }
    catch ( AccountEmailException e )
    {
        throw new AccountServiceException( "Unable to send activation mail.", e );
    }
}

```

signUp() 方法首先检查请求中的两个密码是否一致，接着使用 accountCaptchaService 检查验证码，下一步使用请求中的用户信息实例化一个 Account 对象，并使用 accountPersistService 将用户信息保存。下一步是生成一个随机的激活码并保存在临时的 activationMap 中，然后基于该激活码和请求中的服务器 URL 创建一个激活链接，并使用 accountEmailService 将该链接发送给用户。如果其中任何一步发生异常，signUp() 方法会创建一个一致的 AccountServiceException 对象，提供并抛出对应的异常提示信息。

最后再看一下相对简单的 activate() 和 login() 方法，见代码清单 12-7。

代码清单 12-7 AccountServiceImpl.java 第 4 部分

```

public void activate( String activationId )
    throws AccountServiceException
{

```



```

String accountId=activationMap.get(activationId);

if (accountId==null)
{
    throw new AccountServiceException("Invalid account activation ID.");
}

try
{
    Account account = accountPersistService.readAccount(accountId);
    account.setActivated(true);
    accountPersistService.updateAccount(account);
}
catch (AccountPersistException e)
{
    throw new AccountServiceException("Unable to activate account.");
}
}

public void login(String id, String password)
    throws AccountServiceException
{
    try
    {
        Account account = accountPersistService.readAccount(id);

        if (account == null)
        {
            throw new AccountServiceException("Account does not exist.");
        }

        if (!account.isActivated())
        {
            throw new AccountServiceException("Account is disabled.");
        }

        if (!account.getPassword().equals(password))
        {
            throw new AccountServiceException("Incorrect password.");
        }
    }
    catch (AccountPersistException e)
    {
        throw new AccountServiceException("Unable to log in.", e);
    }
}
}

```

activate() 方法仅仅是简单根据激活码从临时的 activationMap 中寻找对应的用户 ID，如果找到就更新账户状态为激活。login() 方法则是根据 ID 读取用户信息，检查其是否为激活，并比对密码，如果有任何错误则抛出异常。

除了上述代码之外，account-service 还包括一些 Spring 配置文件和单元测试代码，这里

就不再详细介绍。有兴趣的读者可以自行下载阅读。

## 12.3 account-web

account-web 是本书背景案例中唯一的 Web 模块，本书旨在用该模块来阐述如何使用 Maven 来构建一个 Maven 项目。由于 account-service 已经封装了所有下层细节，account-web 只需要在此基础上提供一些 Web 页面，并使用简单 Servlet 与后台实现交互控制。读者将会看到一个具体 Web 项目的 POM 是怎样的，也将能体会到让 Web 模块尽可能简洁带来的好处。

### 12.3.1 account-web 的 POM

除了使用打包方式 war 之外，Web 项目的 POM 与一般项目并没多大的区别。account-web 的 POM 代码见代码清单 12-8。

代码清单 12-8 account-web 的 POM

```
<?xml version="1.0"?>
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>

  <artifactId>account-web</artifactId>
  <packaging>war</packaging>
  <name>Account Web</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-service</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
```

新华书店  
PDG

```

        <scope>provided</scope>
      </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      </dependency>
    </dependencies>
  </project>

```

如上述代码所示，account-web 的 packaging 元素值为 war，表示这是一个 Web 项目，需要以 war 方式进行打包。account-web 依赖于 servlet-api 和 jsp-api 这两个几乎所有 Web 项目都要依赖的包，它们为 servlet 和 jsp 的编写提供支持。需要注意的是，这两个依赖的范围是 provided，表示它们最终不会被打包至 war 文件中，这是因为几乎所有 Web 容器都会提供这两个类库，如果 war 包中重复出现，就会导致潜在的依赖冲突问题。account-web 还依赖于 account-service 和 spring-web，其中前者为 Web 应用提供底层支持，后者为 Web 应用提供 Spring 的集成支持。

在一些 Web 项目中，读者可能会看到 finalName 元素的配置。该元素用来标识项目生成的主构件的名称，该元素的默认值已在超级 POM 中设定，值为 \${project.artifactId}-\${project.version}，因此代码清单 12-8 对应的主构件名称为 account-web-1.0.0-SNAPSHOT.war。不过，这样的名称显然不利于部署，不管是测试环境还是最终产品环境，我们都不想访问页面的时候输入冗长的地址，因此我们会需要名字更为简洁的 war 包。这时可以如下所示配置 finalName 元素：

```
<finalName>account</finalName>
```

经此配置后，项目生成的 war 包名称就会成为 account.war，更方便部署。

### 12.3.2 account-web 的主代码

account-web 的主代码包含了 2 个 JSP 页面和 4 个 Servlet，它们分别为：

- ☐ signup.jsp：账户注册页面。
- ☐ login.jsp：账户登录页面。
- ☐ CaptchaImageServlet：用来生成验证码图片的 Servlet。
- ☐ LoginServlet：处理账户注册请求的 Servlet。
- ☐ ActivateServlet：处理账户激活的 Servlet。
- ☐ LoginServlet：处理账户登录的 Servlet。

Servlet 的配置可以从 web.xml 中获得，该文件位于项目的 src/main/webapp/WEB-INF/ 目录。其内容见代码清单 12-9。

代码清单 12-9 account-web 的 web.xml

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

```

```

<web-app>
  <display-name>Sample Maven Project: Account Service</display-name>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
  </listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:/account-persist.xml
      classpath:/account-captcha.xml
      classpath:/account-email.xml
      classpath:/account-service.xml
    </param-value>
  </context-param>
  <servlet>
    <servlet-name>CaptchaImageServlet</servlet-name>
    <servlet-class>com.juvenxu.mvnbook.account.web.CaptchaImageServlet</serv-
let-class>
  </servlet>
  <servlet>
    <servlet-name>SignUpServlet</servlet-name>
    <servlet-class>com.juvenxu.mvnbook.account.web.SignUpServlet</servlet-
class>
  </servlet>
  <servlet>
    <servlet-name>ActivateServlet</servlet-name>
    <servlet-class>com.juvenxu.mvnbook.account.web.ActivateServlet</serv-
let-class>
  </servlet>
  <servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>com.juvenxu.mvnbook.account.web.LoginServlet</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CaptchaImageServlet</servlet-name>
    <url-pattern>/captcha_image</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>SignUpServlet</servlet-name>
    <url-pattern>/signup</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>ActivateServlet</servlet-name>
    <url-pattern>/activate</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
  </servlet-mapping>
</web-app>

```

web.xml 首先配置了该 Web 项目的显示名称，接着是一个名为 ContextLoaderListener 的

ServletListener。该 listener 来自 spring-web，它用来为 Web 项目启动 Spring 的 IoC 容器，从而实现 Bean 的注入。名为 contextConfigLocation 的 context-param 则用来指定 Spring 配置文件的位置。这里的值是四个模块的 Spring 配置 XML 文件，例如 classpath://account-persist.xml 表示从 classpath 的根路径读取名为 account-persist.xml 的文件。我们知道 account-persist.xml 文件在 account-persist 模块打包后的根路径下，这一 JAR 文件通过依赖的方式被引入到 account-web 的 classpath 下。

web.xml 中的其余部分是 Servlet，包括各个 Servlet 的名称、类名以及对应的 URL 模式。

下面来看一个位于 src/main/webapp/ 目录的 signup.jsp 文件，该文件用来呈现账户注册页面。其内容如代码清单 12-10 所示。

代码清单 12-10 signup.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="com.juvenxu.mvnbook.account.service.*,
    org.springframework.context.ApplicationContext,
    org.springframework.web.context.support.WebApplicationContextUtils"%>
<html>
<head>
<style type="text/css">
...
</style>
</head>
<body>

<%
ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext (
getServletContext () );
AccountService accountservice = (AccountService) context.getBean ( "accountSer-
vice" );
String captchaKey = accountservice.generateCaptchaKey ();
%>

<div class="text-field">

<h2>注册新账户</h2>
<form name="signup" action="signup" method="post">
    <label>账户 ID:</label> <input type="text" name="id"> </input> <br/>
    <label>Email:</label> <input type="text" name="email"> </input>
<br/>
    <label>显示名称:</label> <input type="text" name="name"> </input>
<br/>
    <label>密码:</label> <input type="password" name="password"> </input>
<br/>
    <label>确认密码:</label> <input type="password" name="confirm_password">
</input> <br/>
    <label>验证码:</label> <input type="text" name="captcha_value"> </in-
put> <br/>
    <input type="hidden" name="captcha_key" value="<% =captchaKey% ">"/>
    "/>
    </br>
```

```

        <button>确认并提交 </button>
    </form>
</div>

</body>
</html>

```

该 JSP 的主题是一个 name 为 signup 的 HTML FORM，其中包含了 ID、Email、名称、密码等字段，这与一般的 HTML 内容并无差别。不同的地方在于，该 JSP 文件引入了 Spring 的 ApplicationContext 类，并且用此类加载后台的 accountService，然后使用 accountService 先生成一个验证码的 key，再在 FORM 中使用该 key 调用 captcha\_image 对应的 Servlet 生成其标识的验证码图片。需要注意的是，上述代码中略去了 css 片段。

账户注册页面如图 12-1 所示。

上述 JSP 中使用到了/captcha\_image 这一资源获取验证码图片。根据 web.xml，我们知道该资源对应了 CaptchaImageServlet。下面看一下它的代码，见代码清单 12-11。

**注册新账户**

账户ID:	<input type="text" value="juven"/>
Email:	<input type="text" value="test@juvenxu.com"/>
显示名称:	<input type="text" value="Juven Xu"/>
密码:	<input type="password" value="•••••"/>
确认密码:	<input type="password" value="•••••"/>
验证码:	<input type="text" value="6c8x7"/>




图 12-1 账户注册页面

代码清单 12-11 CaptchaImageServlet.java

```

package com.juvenxu.mvnbook.account.web;

import java.io.IOException;
import ...

public class CaptchaImageServlet
    extends HttpServlet
{
    private ApplicationContext context;

    private static final long serialVersionUID = 5274323889605521606L;

```

```

@Override
public void init()
    throws ServletException
{
    super.init();
    context = WebApplicationContextUtils.getWebApplicationContext( getServ-
letContext() );
}

public void doGet( HttpServletRequest request, HttpServletResponse response)
    throws ServletException,
        IOException
{
    String key = request.getParameter( "key" );

    if ( key == null || key.length() == 0 )
    {
        response.sendError( 400, "No Captcha Key Found" );
    }
    else
    {
        AccountService service = (AccountService) context.getBean( "account-
Service" );

        try
        {
            response.setContentType( "image/jpeg" );
            OutputStream out = response.getOutputStream();
            out.write( service.generateCaptchaImage( key ) );
            out.close();
        }
        catch ( AccountServiceException e )
        {
            response.sendError( 404, e.getMessage() );
        }
    }
}
}

```

CaptchaImageServlet 在 init() 方法中初始化 Spring 的 ApplicationContext，这一 context 用来获取 Spring Bean。Servlet 的 doGet() 方法中首先检查 key 参数，如果为空，则返回 HTTP 400 错误，标识客户端的请求不合法；如果不为空，则载入 AccountService 实例。该类的 generateCaptchaImage() 方法能够产生一个验证码图片的字节流，我们将其设置成 image/jpeg 格式，并写入到 Servlet 相应的输出流中，客户端就能得到图 12-1 所示的验证码图片。

代码清单 12-10 中 FROM 的提交目标是 signup，其对应了 SignUpServlet。其内容如代码清单 12-12 所示。

代码清单 12-12 SignUpServlet.java

```

public class SignUpServlet
    extends HttpServlet
{

```

```

private static final long serialVersionUID = 4784742296013868199L;

private ApplicationContext context;

@Override
public void init ()
    throws ServletException
{
    super.init ();
    context = WebApplicationContextUtils.getWebApplicationContext ( getServ-
letContext () );
}

@Override
protected void doPost ( HttpServletRequest req, HttpServletResponse resp )
    throws ServletException,
        IOException
{
    String id = req.getParameter ( "id" );
    String email = req.getParameter ( "email" );
    String name = req.getParameter ( "name" );
    String password = req.getParameter ( "password" );
    String confirmPassword = req.getParameter ( "confirm_password" );
    String captchaKey = req.getParameter ( "captcha_key" );
    String captchaValue = req.getParameter ( "captcha_value" );

    if ( id == null || id.length () == 0 || email == null || email.length () == 0 ||
name == null
        || name.length () == 0 || password == null || password.length () == 0 || con-
firmPassword == null
        || confirmPassword.length () == 0 || captchaKey == null || captchaKey.length
() == 0 || captchaValue == null
        || captchaValue.length () == 0 )
    {
        resp.sendError ( 400, "Parameter Incomplete." );
        return;
    }

    AccountService service = (AccountService) context.getBean ( "accountSer-
vice" );

    SignUpRequest request = new SignUpRequest ();

    request.setId ( id );
    request.setEmail ( email );
    request.setName ( name );
    request.setPassword ( password );
    request.setConfirmPassword ( confirmPassword );
    request.setCaptchaKey ( captchaKey );
    request.setCaptchaValue ( captchaValue );

    request.setActivateServiceUrl ( getServletContext ().getRealPath ( "/" ) +
"activate" );

    try

```



```

    {
        service.signUp(request);
        resp.getWriter().print("Account is created, please check your mail box
for activation link.");
    }
    catch (AccountServiceException e)
    {
        resp.sendError(400, e.getMessage());
        return;
    }
}
}

```

SignUpServlet 的 doPost() 接受客户端的 HTTP POST 请求，首先它读取请求中的 id、name、email 等参数，然后验证这些参数的值是否为空，如果验证正确，则初始化一个 SignUpRequest 实例，其包含了注册账户所需要的各类数据。其中的 activateServiceUrl 表示服务应该基于什么地址发送账户激活链接邮件，这里的值是与 signup 平行的 activate 地址，这正是 ActivationServlet 的地址。SignUpServlet 使用 AccountService 注册账户，所有的细节都已经封装在 AccountService 中，如果注册成功，服务器打印一条简单的提示信息。

上面介绍了一个 JSP 和两个 Servlet，它们都非常简单。鉴于篇幅的原因，这里就不再详细解释另外几个 JSP 及 Servlet。感兴趣的读者可以自行下载本书的样例源码。

## 12.4 使用 jetty-maven-plugin 进行测试

在进行 Web 开发的时候，我们总是无法避免打开浏览器对应用进行测试，比如为了验证程序功能、验证页面布局，尤其是一些与页面相关的特性，手动部署到 Web 容器进行测试似乎是唯一的方法。近年来出现了很多自动化的 Web 测试技术如 Selenium，它能够录制 Web 操作，生成各种语言脚本，然后自动重复这些操作以进行测试。应该说，这类技术方法是未来的趋势，但无论如何，手动的、亲眼比对验证的测试是无法被完全替代的。测试 Web 页面的做法通常是将项目打包并部署到 Web 容器中，本节介绍如何使用 jetty-maven-plugin，以使这些步骤更为便捷。

在介绍 jetty-maven-plugin 之前，笔者要强调一点，虽然手动的 Web 页面测试是不可避免的，但这种方法绝不应该被滥用。现实中常见的情况是，很多程序员即使修改了一些较底层的代码（如数据库访问、业务逻辑），都会习惯性地打开浏览器测试整个应用，这往往是没有必要的。可以用单元测试覆盖的代码就不应该依赖于 Web 页面测试，且不说页面测试更加耗时耗力，这种方式还无法自动化，更别提重复性了。因此 Web 页面测试应该仅限于页面的层次，例如 JSP、CSS、JavaScript 的修改，其他代码修改（如数据访问），请编写单元测试。

传统的 Web 测试方法要求我们编译、测试、打包及部署，这往往会消耗数 10 秒至数分钟的时间，jetty-maven-plugin 能够帮助我们节省时间，它能够周期性地检查项目内容，发现

变更后自动更新到内置的 Jetty Web 容器中。换句话说，它帮我们省去了打包和部署的步骤。jetty-maven-plugin 默认就很好地支持了 Maven 的项目目录结构。在通常情况下，我们只需要直接在 IDE 中修改源码，IDE 能够执行自动编译，jetty-maven-plugin 发现编译后的文件变化后，自动将其更新到 Jetty 容器，这时就可以直接测试 Web 页面了。

使用 jetty-maven-plugin 十分简单。指定该插件的坐标，并且稍加配置即可，见代码清单 12-13。

代码清单 12-13 配置 jetty-maven-plugin

---

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>7.1.6.v20100715</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webAppConfig>
      <contextPath>/test</contextPath>
    </webAppConfig>
  </configuration>
</plugin>
```

---

jetty-maven-plugin 并不是官方的 Maven 插件，它的 groupId 是 org.mortbay.jetty，上述代码中使用了 Jetty 7 的最新版本。在该插件的配置中，scanIntervalSeconds 顾名思义表示该插件扫描项目变更的时间间隔，这里的配置是每隔 10 秒。需要注意的是，如果不进行配置，该元素的默认值是 0，表示不扫描，用户也就失去了所谓的自动化热部署的功能。上述代码中 webappConfig 元素下的 contextPath 表示项目部署后的 context path。例如这里的值为 /test，那么用户就可以通过 http://hostname:port/test/ 访问该应用。

下一步启动 jetty-maven-plugin。不过在这之前需要对 settings.xml 做个微小的修改。前面介绍过，默认情况下，只有 org.apache.maven.plugins 和 org.codehaus.mojo 两个 groupId 下的插件才支持简化的命令行调用，即可以运行 mvn help:system，但 mvn jetty:run 就不行了。因为 maven-help-plugin 的 groupId 是 org.apache.maven.plugins，而 jetty-maven-plugin 的 groupId 是 org.mortbay.jetty。为了能在命令行直接运行 mvn jetty:run，用户需要配置 settings.xml 如下：

```
<settings>
  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

现在可以运行如下命令启动 jetty-maven-plugin：

```
$ mvn jetty:run
```

jetty-maven-plugin 会启动 Jetty，并且默认监听本地的 8080 端口，并将当前项目部署到容器中，同时它还会根据用户配置扫描代码改动。

如果希望使用其他端口，可以添加 `jetty.port` 参数。例如：

```
$ mvn jetty:run -Djetty.port=9999
```

现在就可以打开浏览器通过地址 `http://localhost:9999/test/` 测试应用了。要停止 Jetty，只需要在命令行输入 `Ctrl + C` 即可。

启动 Jetty 之后，用户可以在 IDE 中修改各类文件，如 JSP、HTML、CSS、JavaScript 甚至是 Java 类。只要不是修改类名、方法名等较大的操作，`jetty-maven-plugin` 都能够扫描到变更并正确地将变化更新至 Web 容器中，这无疑在很大程度上帮助用户实现快速开发和测试。

上面的内容仅仅展示了 `jetty-maven-plugin` 最核心的配置点，如果有需要，还可以自定义 `web.xml` 的位置、项目 `class` 文件的位置、web 资源目录的位置等信息。用户还能够以 WAR 包的方式部署项目，甚至在 Maven 的生命周期中嵌入 `jetty-maven-plugin`。例如，先启动 Jetty 容器并部署项目，然后执行一些集成测试，最后停止容器。有兴趣进一步研究的读者可以访问该页面：[http://wiki.eclipse.org/Jetty/Feature/Jetty\\_Maven\\_Plugin](http://wiki.eclipse.org/Jetty/Feature/Jetty_Maven_Plugin)。

## 12.5 使用 Cargo 实现自动化部署

Cargo 是一组帮助用户操作 Web 容器的工具，它能够帮助用户实现自动化部署，而且它几乎支持所有的 Web 容器，如 Tomcat、JBoss、Jetty 和 Glassfish 等。Cargo 通过 `cargo-maven2-plugin` 提供了 Maven 集成，Maven 用户可以使用该插件将 Web 项目部署到 Web 容器中。虽然 `cargo-maven2-plugin` 和 `jetty-maven-plugin` 的功能看起来很相似，但它们的目的是不同的，`jetty-maven-plugin` 主要用来帮助日常的快速开发和测试，而 `cargo-maven2-plugin` 主要服务于自动化部署。例如专门的测试人员只需要一条简单的 Maven 命令，就可以构建项目并部署到 Web 容器中，然后进行功能测试。本节以 Tomcat 6 为例，介绍如何自动化地将 Web 应用部署至本地或远程 Web 容器中。

### 12.5.1 部署至本地 Web 容器

Cargo 支持两种本地部署的方式，分别为 `standalone` 模式和 `existing` 模式。在 `standalone` 模式中，Cargo 会从 Web 容器的安装目录复制一份配置到用户指定的目录，然后在此基础上部署应用，每次重新构建的时候，这个目录都会被清空，所有配置被重新生成。而在 `existing` 模式中，用户需要指定现有的 Web 容器配置目录，然后 Cargo 会直接使用这些配置并将应用部署到其对应的位置。代码清单 12-14 展示了 `standalone` 模式的配置样例。

代码清单 12-14 使用 `standalone` 模式部署应用至本地 Web 容器

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
```

```

        <containerId>tomcat6x</containerId>
        <home>D:\cmd\apache-tomcat-6.0.29</home>
    </container>
    <configuration>
        <type>standalone</type>
        <home>${project.build.directory}/tomcat6x</home>
    </configuration>
</configuration>
</plugin>

```

cargo-maven2-plugin 的 groupId 是 org.codehaus.cargo，这不属于官方的两个 Maven 插件 groupId，因此用户需要将其添加到 settings.xml 的 pluginGroup 元素中以方便命令行调用。

上述 cargo-maven2-plugin 的具体配置包括了 container 和 configuration 两个元素，configuration 的子元素 type 表示部署的模式（这里是 standalone）。与之对应的，configuration 的 home 子元素表示复制容器配置到什么位置，这里的值为 \${project.build.directory}/tomcat6x，表示构建输出目录，即 target/下的 tomcat6x 子目录。container 元素下的 containerId 表示容器的类型，home 元素表示容器的安装目录。基于该配置，Cargo 会从 D:\cmd\apache-tomcat-6.0.29 目录下复制配置到当前项目的 target/tomcat6x/目录下。

现在，要让 Cargo 启动 Tomcat 并部署应用，只需要运行：

```
$ mvn cargo:start
```

以 account-web 为例，现在就可以直接访问地址的账户注册页面<sup>⑤</sup>了。

默认情况下，Cargo 会让 Web 容器监听 8080 端口。可以通过修改 Cargo 的 cargo.servlet.port 属性来改变这一配置，如代码清单 12-15 所示。

代码清单 12-15 更改 Cargo 的 Servlet 监听端口

```

<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.0</version>
    <configuration>
        <container>
            <containerId>tomcat6x</containerId>
            <home>D:\cmd\apache-tomcat-6.0.29</home>
        </container>
        <configuration>
            <type>standalone</type>
            <home>${project.build.directory}/tomcat6x</home>
            <properties>
                <cargo.servlet.port>8081</cargo.servlet.port>
            </properties>
        </configuration>
    </configuration>
</plugin>

```

⑤ 地址为 <http://localhost:8080/account-web-1.0.0-SNAPSHOT/signup.jsp>。

要将应用直接部署到现有的 Web 容器下，需要配置 Cargo 使用 existing 模式，如代码清单 12-16 所示。

代码清单 12-16 使用 existing 模式部署应用至本地 Web 容器

---

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <home>D:\cmd\apache-tomcat-6.0.29</home>
    </container>
    <configuration>
      <type>existing</type>
      <home>D:\cmd\apache-tomcat-6.0.29</home>
    </configuration>
  </configuration>
</plugin>
```

---

上述代码中 configuration 元素的 type 子元素的值为 existing，而对应的 home 子元素表示现有的 Web 容器目录，基于该配置运行 mvn cargo: start 之后，便能够在 Tomcat 的 webapps 子目录看到被部署的 Maven 项目。

## 12.5.2 部署至远程 Web 容器

除了让 Cargo 直接管理本地 Web 容器然后部署应用之外，也可以让 Cargo 部署应用至远程的正在运行的 Web 容器中。当然，前提是拥有该容器的相应管理员权限。相关配置如代码清单 12-17 所示。

代码清单 12-17 部署应用至远程 Web 容器

---

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <type>remote</type>
    </container>
    <configuration>
      <type>runtime</type>
      <properties>
        <cargo.remote.username>admin</cargo.remote.username>
        <cargo.remote.password>admin123</cargo.remote.password>
        <cargo.tomcat.manager.url>http://localhost:9080/manager</car-
go.tomcat.manager.url>
      </properties>
    </configuration>
  </configuration>
```

---

```
</plugin>
```

对于远程部署的方式来说, container 元素的 type 子元素的值必须为 remote。如果不显式指定, Cargo 会使用默认值 installed, 并寻找对应的容器安装目录或者安装包, 对于远程部署方式来说, 安装目录或者安装包是不需要的。上述代码中 configuration 的 type 子元素值为 runtime, 表示既不使用独立的容器配置, 也不使用本地现有的容器配置, 而是依赖于一个已运行的容器。properties 元素用来声明一些容器热部署相关的配置。例如, 这里的 Tomcat 6 就需要提供用户名、密码以及管理地址。需要注意的是, 这部分配置元素对于所有容器来说不是一致的, 读者需要查阅对应的 Cargo 文档。

有了上述配置后, 就可以让 Cargo 部署应用了。运行命令如下:

```
$ mvn cargo:redeploy
```

如果容器中已经部署了当前应用, Cargo 会先将其卸载, 然后再重新部署。

由于自动化部署本身就不是简单的事情, 再加上 Cargo 要兼容各种不同类型的 Web 容器, 因此 cargo-maven2-plugin 的相关配置会显得相对复杂, 这个时候完善的文档就显得尤为重要。如果想进一步了解 Cargo, 可访问 <http://cargo.codehaus.org/Maven2+plugin>。

## 12.6 小结

本章介绍的是用 Maven 管理 Web 项目, 因此首先讨论了 Web 项目的基本结构, 然后分析实现了本书背景案例的最后两个模块: account-service 和 account-web, 其中后者是一个典型的 Web 模块。开发 Web 项目的时候, 大家往往会使用热部署来实现快速的开发和测试, jetty-maven-plugin 可以帮助实现这一目标。本章最后讨论的是自动化部署, 这一技术的主角是 Cargo, 有了它, 可以让 Maven 自动部署应用至本地和远程 Web 容器中。

