第 10 章

使用 Maven 进行测试

本章内容

- account-captcha
- maven-surefire-plugin 简介
- 跳过测试
- 动态指定要运行的测试用例
- 包含与排除测试用例
- 测试报告
- 运行 TestNG 测试
- 重用测试代码
- 小结

随着敏捷开发模式的日益流行，软件开发人员也越来越认识到日常编程工作中单元测试的重要性。Maven 的重要职责之一就是自动运行单元测试，它通过 maven-surefire-plugin 与主流的单元测试框架 JUnit 3、JUnit 4 以及 TestNG 集成，并且能够自动生成丰富的结果报告。本章将介绍 Maven 关于测试的一些重要特性，但不会深入解释单元测试框架本身及相关技巧，重点是介绍如何通过 Maven 控制单元测试的运行。

除了测试之外，本章还会进一步丰富账户注册服务这一背景案例，引入其第 3 个模块：account-captcha。

10.1　account-captcha

在讨论 maven-surefire-plugin 之前，本章先介绍实现账户注册服务的 account-captcha 模块，该模块负责处理账户注册时验证码的 key 生成、图片生成以及验证等。读者可以回顾第 4 章的背景案例以获得更具体的需求信息。

10.1.1　account-captcha 的 POM

该模块的 POM（Project Object Model，项目对象模型）还是比较简单的，内容见代码清单 10-1。

代码清单 10-1　account-captcha 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-captcha</artifactId>
    <name>Account Captcha</name>

    <properties>
        <kaptcha.version>2.3</kaptcha.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.google.code.kaptcha</groupId>
            <artifactId>kaptcha</artifactId>
            <version>${kaptcha.version}</version>
            <classifier>jdk15</classifier>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>sonatype-forge</id>
            <name>Sonatype Forge</name>
            <url>http://repository.sonatype.org/content/groups/forge/</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
</project>
```

首先 POM 中的第一部分是父模块声明，如同 account-email、account-persist 一样，这里将父模块声明为 account-parent。紧接着是该项目本身的 artifactId 和名称，groupId 和 version 没有声明，将自动继承自父模块。再往下声明了一个 Maven 属性 kaptcha. version，该属性用在依赖声明中，account-captcha 的依赖除了 SpringFramework 和 JUnit 之外，还有一个 com. google. code. kaptcha：kaptcha。Kaptcha 是一个用来生成验证码（Captcha）的开源类库，account-captcha 将用它来生成注册账户时所需要的验证码图片，如果想要了解更多关于 Kaptcha 的信息，可以访问其项目主页：http://code. google. com/p/kaptcha/。

POM 中 SpringFramework 和 JUnit 的依赖配置都继承自父模块，这里不再赘述。Kaptcha 依赖声明中 version 使用了 Maven 属性，这在之前也已经见过。需要注意的是，Kaptcha 依赖还有一个 classifier 元素，其值为 jdk5，Kaptcha 针对 Java 1.5 和 Java 1.4 提供了不同的分发包，因此这里使用 classifier 来区分两个不同的构件。

POM 的最后声明了 Sonatype Forge 这一公共仓库，这是因为 Kaptcha 并没有上传的中央仓库，我们可以从 Sonatype Forge 仓库获得该构件。如果有自己的私服，就不需要在 POM 中声明该仓库了，可以代理 Sonatype Forge 仓库，或者直接将 Kaptcha 上传到自己的仓库中。

最后，不能忘记把 account-captcha 加入到聚合模块（也是父模块）account-parent 中，见代码清单 10-2。

代码清单 10-2　将 account-captcha 加入到聚合模块 account-parent

```
<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>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>Account Parent</name>
    <modules>
        <module>account-email</module>
        <module>account-persist</module>
        <module>account-captcha</module>
    </modules>
</project>
```

10.1.2　account-captcha 的主代码

account-captcha 需要提供的服务是生成随机的验证码主键，然后用户可以使用这个主键要求服务生成一个验证码图片，这个图片对应的值应该是随机的，最后用户用肉眼读取图片的值，并将验证码的主键与这个值交给服务进行验证。这一服务对应的接口可以定义，如代码清单 10-3 所示。

代码清单 10-3　AccountCaptchaService.java

```
package com.juvenxu.mvnbook.account.captcha;

import java.util.List;

public interface AccountCaptchaService
{
    String generateCaptchaKey()
        throws AccountCaptchaException;

    byte[] generateCaptchaImage( String captchaKey )
        throws AccountCaptchaException;

    boolean validateCaptcha( String captchaKey, String captchaValue )
        throws AccountCaptchaException;

    List<String> getPreDefinedTexts();

    void setPreDefinedTexts( List<String> preDefinedTexts );
}
```

很显然，generateCaptchaKey() 用来生成随机的验证码主键，generateCaptchaImage() 用来生成验证码图片，而 validateCaptcha() 用来验证用户反馈的主键和值。

该接口定义了额外的 getPreDefinedTexts() 和 setPreDefinedTexts() 方法，通过这一组方

法，用户可以预定义验证码图片的内容，同时也提高了可测试性。如果 AccountCaptchaService 永远生成随机的验证码图片，那么没有人工的参与就很难测试该功能。现在，服务允许传入一个文本列表，这样就可以基于这些文本生成验证码，那么我们也就能控制验证码图片的内容了。

为了能够生成随机的验证码主键，引入一个 RandomGenerator 类，见代码清单 10-4。

代码清单 10-4 RandomGenerator. java

```
package com.juvenxu.mvnbook.account.captcha;

import java.util.Random;

public class RandomGenerator
{
    private static String range = "0123456789abcdefghijklmnopqrstuvwxyz";

    public static synchronized String getRandomString()
    {
        Random random = new Random();

        StringBuffer result = new StringBuffer();

        for ( int i = 0; i < 8; i++ )
        {
            result.append( range.charAt( random.nextInt( range.length() ) ) );
        }

        return result.toString();
    }
}
```

RandomGenerator 类提供了一个静态且线程安全的 getRandomString( ) 方法。该方法生成一个长度为 8 的字符串，每个字符都是随机地从所有数字和字母中挑选，这里主要是使用了 java. util. Random 类，其 nextInt （int n） 方法会返回一个大于等于 0 且小于 n 的整数。代码中的字段 range 包含了所有的数字与字母，将其长度传给 nextInt( ) 方法后就能获得一个随机的下标，再调用 range. charAt( ) 就可以随机取得一个其包含的字符了。

现在看 AccountCaptchaService 的实现类 AccountCaptchaServiceImpl。首先需要初始化验证码图片生成器，见代码清单 10-5。

代码清单 10-5 AccountCaptchaServiceImpl. java 的 afterPropertySet( ) 方法

```
package com.juvenxu.mvnbook.account.captcha;

import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
```

```
import javax.imageio.ImageIO;

import org.springframework.beans.factory.InitializingBean;

import com.google.code.kaptcha.impl.DefaultKaptcha;
import com.google.code.kaptcha.util.Config;

public class AccountCaptchaServiceImpl
    implements AccountCaptchaService, InitializingBean
{
    private DefaultKaptcha producer;

    public void afterPropertiesSet()
        throws Exception
    {
        producer = new DefaultKaptcha();

        producer.setConfig(new Config(new Properties()));
    }
...
```

AccountCaptchaServiceImpl 实现了 SpringFramework 的 InitializingBean 接口，该接口定义了一个方法 afterPropertiesSet()，该方法会被 SpringFramework 初始化对象的时候调用。该代码清单中使用该方法初始化验证码生成器 producer，并且为 producer 提供了默认的配置。

接着 AccountCaptchaServiceImpl 需要实现 generateCaptchaKey() 方法，见代码清单 10-6。

代码清单 10-6　AccountCaptchaServiceImpl. java 的 generateCaptchaKey() 方法

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

private List<String> preDefinedTexts;

private int textCount = 0;

public String generateCaptchaKey()
{
    String key = RandomGenerator.getRandomString();

    String value = getCaptchaText();

    captchaMap.put(key, value);

    return key;
}

public List<String> getPreDefinedTexts()
{
    return preDefinedTexts;
}
```

```
public void setPreDefinedTexts( List < String > preDefinedTexts )
{
    this.preDefinedTexts = preDefinedTexts;
}

private String getCaptchaText()
{
    if ( preDefinedTexts != null && !preDefinedTexts.isEmpty() )
    {
        String text = preDefinedTexts.get( textCount );

        textCount = ( textCount + 1 ) % preDefinedTexts.size();

        return text;
    }
    else
    {
        return producer.createText();
    }
}
```

上述代码清单中的 generateCaptchaKey( ) 首先生成一个随机的验证码主键，每个主键将和一个验证码字符串相关联，然后这组关联会被存储到 captchaMap 中以备将来验证。主键的目的仅仅是标识验证码图片，其本身没有实际的意义。代码清单中的 getCaptchaText( ) 用来生成验证码字符串，当 preDefinedTexts 不存在或者为空的时候，就是用验证码图片生成器 producer 创建一个随机的字符串，当 preDefinedTexts 不为空的时候，就顺序地循环该字符串列表读取值。preDefinedTexts 有其对应的一组 get 和 set 方法，这样就能让用户预定义验证码字符串的值。

有了验证码图片的主键，AccountCaptchaServiceImpl 就需要实现 generateCaptchaImage( ) 方法来生成验证码图片，见代码清单 10-7。

代码清单 10-7　AccountCaptchaServiceImpl. java 的 generateCaptchaImage( ) 方法

```
public byte[] generateCaptchaImage( String captchaKey )

    throws AccountCaptchaException
{
    String text = captchaMap.get( captchaKey );

    if ( text == null )
    {
        throw new AccountCaptchaException( "Captch key'" + captchaKey + "'not
found!" );
    }

    BufferedImage image = producer.createImage( text );

    ByteArrayOutputStream out = new ByteArrayOutputStream();

    try
    {
```

```
            ImageIO.write( image, "jpg", out );
        }
        catch ( IOException e )
        {
            throw new AccountCaptchaException( "Failed to write captcha stream!", e );
        }

        return out.toByteArray();
    }
```

为了生成验证码图片，就必须先得到验证码字符串的值，代码清单中通过使用主键来查询 captchaMap 获得该值，如果值不存在，就抛出异常。有了验证码字符串的值之后，generateCaptchaImage( ) 方法就能通过 producer 来生成一个 BufferedImage，随后的代码将这个图片对象转换成 jpg 格式的字节数组并返回。有了该字节数组，用户就能随意地将其保存成文件，或者在网页上显示。

最后是简单的验证过程，见代码清单 10-8。

代码清单 10-8　AccountCaptchaServiceImpl. java 的 validateCaptcha( ) 方法

```
public boolean validateCaptcha( String captchaKey, String captchaValue )
    throws AccountCaptchaException
{
    String text = captchaMap.get( captchaKey );

    if ( text = = null )
    {
        throw new AccountCaptchaException( "Captch key'" + captchaKey + "'not
found!" );
    }

    if ( text.equals( captchaValue ) )
    {
        captchaMap.remove( captchaKey );

        return true;
    }
    else
    {
        return false;
    }
}
```

用户得到了验证码图片以及主键后，就会识别图片中所包含的字符串信息，然后将此验证码的值与主键一起反馈给 validateCaptcha( ) 方法以进行验证。validateCaptcha( ) 通过主键找到正确的验证码值，然后与用户提供的值进行比对，如果成功，则返回 true。

当然，还需要一个 SpringFramework 的配置文件，它在资源目录 src/main/resources/下，名为 account-captcha. xml，见代码清单 10-9。

代码清单 10-9 account-captcha.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="accountCaptchaService"
        class="com.juvenxu.mvnbook.account.captcha.AccountCaptchaServiceImpl">
    </bean>

</beans>
```

这是一个最简单的 SpringFramework 配置，它定义了一个 id 为 accountCaptchaService 的 bean，其实现为刚才讨论的 AccountCaptchaServiceImpl。

10.1.3 account-captcha 的测试代码

测试代码位于 src/test/java 目录，其包名也与主代码一致，为 com.juvenxu.mvnbook.account.captcha。首先看一下简单的 RandomGeneratorTest，见代码清单 10-10。

代码清单 10-10 RandomGeneratorTest.java

```
package com.juvenxu.mvnbook.account.captcha;

import static org.junit.Assert.assertFalse;

import java.util.HashSet;
import java.util.Set;

import org.junit.Test;

public class RandomGeneratorTest
{
    @Test
    public void testGetRandomString()
        throws Exception
    {
        Set<String> randoms = new HashSet<String>(100);

        for (int i = 0; i < 100; i++)
        {
            String random = RandomGenerator.getRandomString();

            assertFalse(randoms.contains(random));

            randoms.add(random);
        }
    }
}
```

该测试用例创建一个初始容量为 100 的集合 randoms，然后循环 100 次用 RandomGenerator 生成随机字符串并放入 randoms 中，同时每次循环都检查新生成的随机值是否已经包含在集合中。这样一个简单的检查能基本确定 RandomGenerator 生成值是否为随机的。

当然这个模块中最重要的测试应该在 AccountCaptchaService 上，见代码清单 10-11。

代码清单 10-11　AccountCaptchaServiceTest.java

```
package com.juvenxu.mvnbook.account.captcha;

import static org.junit.Assert.*;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AccountCaptchaServiceTest
{
    private AccountCaptchaService service;

    @Before
    public void prepare()
        throws Exception
    {
        ApplicationContext ctx = new ClassPathXmlApplicationContext( "account-
captcha.xml" );
        service = (AccountCaptchaService) ctx.getBean( "accountCaptchaService" );
    }

    @Test
    public void testGenerateCaptcha()
        throws Exception
    {
        String captchaKey = service.generateCaptchaKey();
        assertNotNull( captchaKey );

        byte[] captchaImage = service.generateCaptchaImage( captchaKey );
        assertTrue( captchaImage.length > 0 );

        File image = new File( "target/" + captchaKey + ".jpg" );
        OutputStream output = null;
        try
        {
            output = new FileOutputStream( image );
            output.write( captchaImage );
        }
```

```
        finally
        {
            if ( output != null )
            {
                output.close();
            }
        }
        assertTrue( image.exists() && image.length() > 0 );
    }

    @Test
    public void testValidateCaptchaCorrect()
        throws Exception
    {
        List<String> preDefinedTexts = new ArrayList<String>();
        preDefinedTexts.add( "12345" );
        preDefinedTexts.add( "abcde" );
        service.setPreDefinedTexts( preDefinedTexts );

        String captchaKey = service.generateCaptchaKey();
        service.generateCaptchaImage( captchaKey );
        assertTrue( service.validateCaptcha( captchaKey, "12345" ) );

        captchaKey = service.generateCaptchaKey();
        service.generateCaptchaImage( captchaKey );
        assertTrue( service.validateCaptcha( captchaKey, "abcde" ) );
    }

    @Test
    public void testValidateCaptchaIncorrect()
        throws Exception
    {
        List<String> preDefinedTexts = new ArrayList<String>();
        preDefinedTexts.add( "12345" );
        service.setPreDefinedTexts( preDefinedTexts );

        String captchaKey = service.generateCaptchaKey();
        service.generateCaptchaImage( captchaKey );
        assertFalse( service.validateCaptcha( captchaKey, "67890" ) );
    }
}
```

该测试类的 prepare( ) 方法使用@ Before 标注，在运行每个测试方法之前初始化 AccountCaptchaService 这个 bean。

testGenerateCaptcha( ) 用来测试验证码图片的生成。首先它获取一个验证码主键并检查其非空，然后使用该主键获得验证码图片，实际上是一个字节数组，并检查该字节数组的内容非空。紧接着该测试方法在项目的 target 目录下创建一个名为验证码主键的 jpg 格式文件，并将 AccountCaptchaService 返回的验证码图片字节数组内容写入到该 jpg 文件中，然后再检查文件存在且包含实际内容。运行该测试之后，就能在项目的 target 目录下找到一个名如 dhb022fc. jpg 的文件，打开是一个验证码图片，如图 10-1 所示。

图 10-1　AccountCaptchaServiceTest 生成的验证码图片

testValidateCaptchaCorrect()用来测试一个正确的 Captcha 验证流程。它首先预定义了两个 Captcha 的值放到服务中，然后依次生成验证码主键、验证码图片，并且使用主键和已知的值进行验证，确保服务正常工作。

最后的 testValidateCaptchaIncorrect()方法测试当用户反馈的 Captcha 值错误时发生的情景，它先预定义 Captcha 的值为“12345”，但最后验证是传入了“67890”，并检查 validateCaptcha()方法返回的值为 false。

现在运行测试，在项目目录下运行 **mvn test**，就会得到如下输出：

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Account Captcha 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] ...
[INFO]
[INFO] ---maven-surefire-plugin:2.4.3:test (default-test) @ account-captcha ---
[INFO] Surefire report directory: D:\code\ch-10\account-aggregator\account-captcha\target\surefire-reports

-------------------------------------------------------------------------------
TESTS
-------------------------------------------------------------------------------
Running com.juvenxu.mvnbook.account.captcha.RandomGeneratorTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.037 sec
Running com.juvenxu.mvnbook.account.captcha.AccountCaptchaServiceTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.016 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

这个简单的报告告诉我们，Maven 运行了两个测试类，其中第一个测试类 RandomGeneratorTest 包含 1 个测试，第二个测试类 AccountCaptchaServiceTest 包含 3 个测试，所有 4 个测试运行完毕后，没有任何失败和错误，也没有跳过任何测试。

报告中的 Failures、Errors、Skipped 信息来源于 JUnit 测试框架。Failures（失败）表示要测试的结果与预期值不一致，例如测试代码期望返回值为 true，但实际为 false；Errors（错误）表示测试代码或产品代码发生了未预期的错误，例如产品代码抛出了一个空指针错误，该错误又没有被测试代码捕捉到；Skipped 表示那些被标记为忽略的测试方法，在 JUnit 中用户可以使用@Ignore 注解标记忽略测试方法。

10.2 maven-surefire-plugin 简介

Maven 本身并不是一个单元测试框架，Java 世界中主流的单元测试框架为 JUnit（http://www.junit.org/）和 TestNG（http://testng.org/）。Maven 所做的只是在构建执行到特定生命周期阶段的时候，通过插件来执行 JUnit 或者 TestNG 的测试用例。这一插件就是 maven-surefire-plugin，可以称之为测试运行器（Test Runner），它能很好地兼容 JUnit 3、JUnit 4 以及 TestNG。

可以回顾一下 7.2.3 节介绍的 default 生命周期，其中的 test 阶段被定义为“使用单元测试框架运行测试”。我们知道，生命周期阶段需要绑定到某个插件的目标才能完成真正的工作，test 阶段正是与 maven-surefire-plugin 的 test 目标相绑定了，这是一个内置的绑定，具体可参考 7.4.1 节。

在默认情况下，maven-surefire-plugin 的 test 目标会自动执行测试源码路径（默认为 src/test/java/）下所有符合一组命名模式的测试类。这组模式为：

- **/Test*.java：任何子目录下所有命名以 Test 开头的 Java 类。
- **/*Test.java：任何子目录下所有命名以 Test 结尾的 Java 类。
- **/*TestCase.java：任何子目录下所有命名以 TestCase 结尾的 Java 类。

只要将测试类按上述模式命名，Maven 就能自动运行它们，用户也就不再需要定义测试集合（TestSuite）来聚合测试用例（TestCase）。关于模式需要注意的是，以 Tests 结尾的测试类是不会得以自动执行的。

当然，如果有需要，可以自己定义要运行测试类的模式，这一点将在 10.5 节详细描述。此外，maven-surefire-plugin 还支持更高级的 TestNG 测试集合 xml 文件，这一点将在 10.7 节详述。

当然，为了能够运行测试，Maven 需要在项目中引入测试框架的依赖，本书已经多次涉及了如何添加 JUnit 测试范围依赖，这里不再赘述，而关于如何引入 TestNG 依赖，可参看 10.7 节。

10.3 跳过测试

日常工作中，软件开发人员总有很多理由来跳过单元测试，“我敢保证这次改动不会导致任何测试失败”，“测试运行太耗时了，暂时跳过一下”，“有持续集成服务跑所有测试呢，我本地就不执行啦”。在大部分情况下，这些想法都是不对的，任何改动都要交给测试去验证，测试运行耗时过长应该考虑优化测试，更不要完全依赖持续集成服务来报告错误，测试错误应该尽早在尽小范围内发现，并及时修复。

不管怎样，我们总会要求 Maven 跳过测试，这很简单，在命令行加入参数 skipTests 就可以了。例如：

```
$ mvn package -DskipTests
```

Maven 输出会告诉你它跳过了测试：

```
[INFO] ---maven-compiler-plugin:2.0.2:testCompile (default-testCompile) @ ac-
count-captcha ---
[INFO] Compiling 2 source files to D:\\code\ch - 10\account-aggregator\account-
captcha\target\test-classes
[INFO]
[INFO] ---maven-surefire-plugin:2.4.3:test (default-test) @ account-captcha ---
[INFO] Tests are skipped.
```

当然，也可以在 POM 中配置 maven-surefire-plugin 插件来提供该属性，如代码清单 10-12 所示。但这是不推荐的做法，如果配置 POM 让项目长时间地跳过测试，则还要测试代码做什么呢？

代码清单 10-12　配置插件跳过测试运行

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.5</version>
    <configuration>
        <skipTests>true</skipTests>
    </configuration>
</plugin>
```

有时候用户不仅仅想跳过测试运行，还想临时性地跳过测试代码的编译，Maven 也允许你这么做，但记住这是不推荐的：

```
$ mvn package -Dmaven.test.skip=true
```

这时 Maven 的输出如下：

```
[INFO] ---maven-compiler-plugin:2.0.2:testCompile (default-testCompile) @ ac-
count-captcha ---
[INFO] Not compiling test sources
[INFO]
[INFO] ---maven-surefire-plugin:2.4.3:test (default-test) @ account-captcha ---
[INFO] Tests are skipped.
```

参数 maven.test.skip 同时控制了 maven-compiler-plugin 和 maven-surefire-plugin 两个插件的行为，测试代码编译跳过了，测试运行也跳过了。

对应于命令行参数 maven.test.skip 的 POM 配置如代码清单 10-13 所示，但这种方法也是不推荐使用的。

代码清单 10-13　配置插件跳过测试编译和运行

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.1</version>
    <configuration>
```

```
            <skip>true</skip>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.5</version>
        <configuration>
            <skip>true</skip>
        </configuration>
    </plugin>
```

实际上 maven-compiler-plugin 的 testCompile 目标和 maven-surefire-plugin 的 test 目标都提供了一个参数 skip 用来跳过测试编译和测试运行，而这个参数对应的命令行表达式为 maven. test. skip。

10.4 动态指定要运行的测试用例

反复运行单个测试用例是日常开发中很常见的行为。例如，项目代码中有一个失败的测试用例，开发人员就会想要再次运行这个测试以获得详细的错误报告，在修复该测试的过程中，开发人员也会反复运行它，以确认修复代码是正确的。如果仅仅为了一个失败的测试用例而反复运行所有测试，未免太浪费时间了，当项目中测试的数目比较大的时候，这种浪费尤为明显。

maven-surefire-plugin 提供了一个 test 参数让 Maven 用户能够在命令行指定要运行的测试用例。例如，如果只想运行 account-captcha 的 RandomGeneratorTest，就可以使用如下命令：

```
$ mvn test -Dtest=RandomGeneratorTest
```

这里 test 参数的值是测试用例的类名，这行命令的效果就是只有 RandomGeneratorTest 这一个测试类得到运行。

maven-surefire-plugin 的 test 参数还支持高级一些的赋值方式，能让用户更灵活地指定需要运行的测试用例。例如：

```
$ mvn test -Dtest=Random*Test
```

星号可以匹配零个或多个字符，上述命令会运行项目中所有类名以 Random 开头、Test 结尾的测试类。

除了星号匹配，还可以使用逗号指定多个测试用例：

```
$ mvn test -Dtest=RandomGeneratorTest,AccountCaptchaServiceTest
```

该命令的 test 参数值是两个测试类名，它们之间用逗号隔开，其效果就是告诉 Maven 只运行这两个测试类。

当然，也可以结合使用星号和逗号。例如：

```
$ mvn test -Dtest=Random*Test,AccountCaptchaServiceTest
```

需要注意的是，上述几种从命令行动态指定测试类的方法都应该只是临时使用，如果长时间只运行项目的某几个测试，那么测试就会慢慢失去其本来的意义。

test 参数的值必须匹配一个或者多个测试类，如果 maven-surefire-plugin 找不到任何匹配的测试类，就会报错并导致构建失败。例如下面的命令没有匹配任何测试类：

```
$ mvn test-Dtest
```

这样的命令会导致构建失败，输出如下：

```
[INFO] ---maven-surefire-plugin:2.4.3:test (default-test) @ account-captcha ---

[INFO] Surefire report directory: D:\code\ch-10\account-aggregator\account-cap-
tcha\target\surefire-reports

-------------------------------------------------------------------------
T E S T S
-------------------------------------------------------------------------
There are no tests to run.

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.747s
[INFO] Finished at: Sun Mar 28 17:00:27 CST 2010
[INFO] Final Memory: 2M/5M
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:
2.4.3:test (default-test) on project account-captcha: No tests were executed!
(Set-DfailIfNoTests = false to ignore this error.) - > [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
```

根据错误提示可以加上 -DfailIfNoTests = false，告诉 maven-surefire-plugin 即使没有任何测试也不要报错：

```
$ mvn test-Dtest-DfailIfNoTests = false
```

这样构建就能顺利执行完毕了。可以发现，实际上使用命令行参数-Dtest-DfailIfNoTests = false 是另外一种跳过测试的方法。

我们看到，使用 test 参数用户可以从命令行灵活地指定要运行的测试类。可惜的是，maven-surefire-plugin 并没有提供任何参数支持用户从命令行跳过指定的测试类，好在用户可以通过在 POM 中配置 maven-surefire-plugin 排除特定的测试类。

10.5　包含与排除测试用例

10.2 节介绍了一组命名模式，符合这一组模式的测试类将会自动执行。Maven 提倡约

定优于配置原则，因此用户应该尽量遵守这一组模式来为测试类命名。即便如此，maven-surefire-plugin 还是允许用户通过额外的配置来自定义包含一些其他测试类，或者排除一些符合默认命名模式的测试类。

例如，由于历史原因，有些项目所有测试类名称都以 Tests 结尾，这样的名字不符合默认的 3 种模式，因此不会被自动运行，用户可以通过代码清单 10-14 所示的配置让 Maven 自动运行这些测试。

代码清单 10-14 自动运行以 Tests 结尾的测试类

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <includes>
      <include>**/*Tests.java</include>
    </includes>
  </configuration>
</plugin>
```

上述代码清单中使用了 **/*Tests.java 来匹配所有以 Tests 结尾的 Java 类，两个星号 ** 用来匹配任意路径，一个星号 * 匹配除路径风格符外的 0 个或者多个字符。

类似地，也可以使用 excludes 元素排除一些符合默认命名模式的测试类，如代码清单 10-15 所示。

代码清单 10-15 排除运行测试类

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <excludes>
      <exclude>**/*ServiceTest.java</exclude>
      <exclude>**/TempDaoTest.java</exclude>
    </excludes>
  </configuration>
</plugin>
```

上述代码清单排除了所有以 ServiceTest 结尾的测试类，以及一个名为 TempDaoTest 的测试类。它们都符合默认的命名模式 **/*Test.java，不过，有了 excludes 配置后，maven-surefire-plugin 将不再自动运行它们。

10.6 测试报告

除了命令行输出，Maven 用户可以使用 maven-surefire-plugin 等插件以文件的形式生成更丰富的测试报告。

10.6.1　基本的测试报告

默认情况下，maven-surefire-plugin 会在项目的 target/surefire-reports 目录下生成两种格式的错误报告：

- 简单文本格式
- 与 JUnit 兼容的 XML 格式

例如，运行 10.1.3 节代码清单 10-10 中的 RandomGeneratorTest 后会得到一个名为 com.juvenxu.mvnbook.account.captcha.RandomGeneratorTest.txt 的简单文本测试报告和一个名为 TEST-com.juvenxu.mvnbook.account.captcha.RandomGeneratorTest.xml 的 XML 测试报告。前者的内容十分简单：

```
-------------------------------------------------------------------------------
Test set: com.juvenxu.mvnbook.account.captcha.RandomGeneratorTest
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec
```

这样的报告对于获得信息足够了，XML 格式的测试报告主要是为了支持工具的解析，如 Eclipse 的 JUnit 插件可以直接打开这样的报告，如图 10-2 所示。

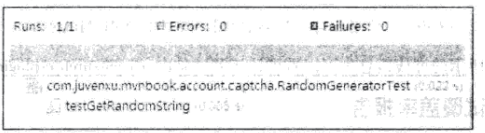

图 10-2　使用 Eclipse JUnit 插件打开成功的 XML 测试报告

由于这种 XML 格式已经成为了 Java 单元测试报告的事实标准，一些其他工具也能使用它们。例如，持续集成服务器 Hudson 就能使用这样的文件提供持续集成的测试报告。

以上展示了一些运行正确的测试报告，实际上，错误的报告更具价值。我们可以修改 10.1.3 节代码清单 10-11 中的 AccountCaptchaServiceTest 让一个测试失败，这时得到的简单文本报告会是这样：

```
-------------------------------------------------------------------------------
Test set: com.juvenxu.mvnbook.account.captcha.AccountCaptchaServiceTest
-------------------------------------------------------------------------------
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.932 sec <<<
FAILURE!
testValidateCaptchaCorrect (com.juvenxu.mvnbook.account.captcha.AccountCapt-
chaServiceTest) Time elapsed: 0.047 sec <<< FAILURE!
java.lang.AssertionError:
    at org.junit.Assert.fail(Assert.java:91)
    at org.junit.Assert.assertTrue(Assert.java:43)
    at org.junit.Assert.assertTrue(Assert.java:54)
```

```
    at com.juvenxu.mvnbook.account.captcha.AccountCaptchaServiceTest.testVal-
idateCaptchaCorrect(AccountCaptchaServiceTest.java:66)
...
```

报告说明了哪个测试方法失败、哪个断言失败以及具体的堆栈信息，用户可以据此快速地寻找失败原因。该测试的 XML 格式报告用 Eclipse JUnit 插件打开，如图 10-3 所示。

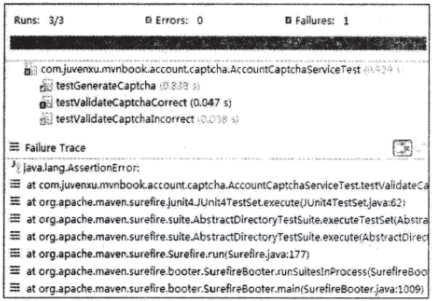

图 10-3　使用 Eclipse JUnit 插件打开失败的 XML 测试报告

从图 10-3 所示的堆栈信息中可以看到，该测试是由 maven-surefire-plugin 发起的。

10.6.2　测试覆盖率报告

测试覆盖率是衡量项目代码质量的一个重要的参考指标。Cobertura 是一个优秀的开源测试覆盖率统计工具（详见 http://cobertura.sourceforge.net/），Maven 通过 cobertura-maven-plugin 与之集成，用户可以使用简单的命令为 Maven 项目生成测试覆盖率报告。例如，可以在 account-captcha 目录下运行如下命令生成报告：

```
$ mvn cobertura:cobertura
```

接着打开项目目录 target/site/cobertura/下的 index.html 文件，就能看到如图 10-4 所示的测试覆盖率报告。

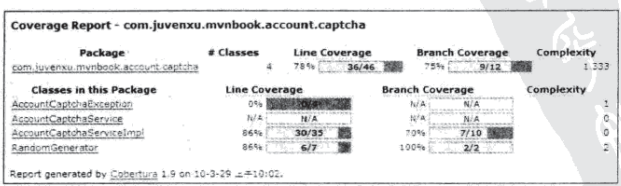

图 10-4　Cobertura 测试覆盖率报告

单击具体的类，还能看到精确到行的覆盖率报告，如图 10-5 所示。

```
74      public byte[] generateCaptchaImage( String captchaKey )
75          throws AccountCaptchaException
76      {
77 4        String text = captchaMap.get( captchaKey );
78
79 4        if ( text == null )
80          {
81 0            throw new AccountCaptchaException( "Captcha key " + captchaKey + " not found!" );
82          }
83
84 4        BufferedImage image = producer.createImage( text );
85
86 4        ByteArrayOutputStream out = new ByteArrayOutputStream();
87
88          try
89          {
90 4            ImageIO.write( image, "jpg", out );
91          }
92 0        catch ( IOException e )
93          {
94 0            throw new AccountCaptchaException( "Failed to write captcha stream", e );
95 4        }
96
97 4        return out.toByteArray();
98      }
```

图 10-5　具体到代码行的 Cobertura 测试覆盖率报告

10.7　运行 TestNG 测试

TestNG 是 Java 社区中除 JUnit 之外另一个流行的单元测试框架。NG 是 Next Generation 的缩写，译为“下一代”。TestNG 在 JUnit 的基础上增加了很多特性，读者可以访问其站点 http://testng.org/ 获取更多信息。值得一提的是，《Next Generation Java Testing》（Java 测试新技术，中文版已由机械工业出版社引进出版，书号为 978-7-111-24550-6）一书专门介绍 TestNG 和相关测试技巧。

使用 Maven 运行 TestNG 十分方便。以 10.1.3 节中的 account-captcha 测试代码为例，首先需要删除 POM 中的 JUnit 依赖，加入 TestNG 依赖，见代码清单 10-16。

代码清单 10-16　加入 TestNG 依赖

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>5.9</version>
    <scope>test</scope>
    <classifier>jdk15</classifier>
</dependency>
```

与 JUnit 类似，TestNG 的依赖范围应为 test。此外，TestNG 使用 classifier jdk15 和 jdk14 为不同的 Java 平台提供支持。

下一步需要将对 JUnit 的类库引用更改成对 TestNG 的类库引用。表 10-1 给出了常用类库的对应关系。

表 10-1 JUnit 和 TestNG 的常用类库对应关系

JUnit 类	TestNG 类	作 用
org. junit. Test	org. testng. annotations. Test	标注方法为测试方法
org. junit. Assert	org. testng. Assert	检查测试结果
org. junit. Before	org. testng. annotations. BeforeMethod	标注方法在每个测试方法之前运行
org. junit. After	org. testng. annotations. AfterMethod	标注方法在每个测试方法之后运行
org. junit. BeforeClass	org. testng. annotations. BeforeClass	标注方法在所有测试方法之前运行
org. junit. AfterClass	org. testng. annotations. AfterClass	标注方法在所有测试方法之后运行

将 JUnit 的类库引用改成 TestNG 之后，在命令行输入 **mvn test**，Maven 就会自动运行那些符合命名模式的测试类。这一点与运行 JUnit 测试没有区别。

TestNG 允许用户使用一个名为 testng. xml 的文件来配置想要运行的测试集合。例如，可以在 account-captcha 的项目根目录下创建一个 testng. xml 文件，配置只运行 RandomGeneratorTest，如代码清单 10-17 所示。

代码清单 10-17 TestNG 的 testng. xml

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

<suite name = "Suite1" verbose = "1" >
    <test name = "Regression1" >
        <classes >
          <class name = "com.juvenxu.mvnbook.account.captcha.RandomGeneratorTest"/>
        </classes >
    </test >
</suite >
```

同时再配置 maven-surefire-plugin 使用该 testng. xml，如代码清单 10-18 所示。

代码清单 10-18 配置 maven-surefire-plugin 使用 testng. xml

```
<plugin >
    <groupId >org.apache.maven.plugins </groupId >
    <artifactId >maven-surefire-plugin </artifactId >
    <version >2.5 </version >
    <configuration >
        <suiteXmlFiles >
            <suiteXmlFile >testng.xml </suiteXmlFile >
        </suiteXmlFiles >
    </configuration >
</plugin >
```

TestNG 较 JUnit 的一大优势在于它支持测试组的概念，如下的注解会将测试方法加入到两个测试组 util 和 medium 中：

```
@ Test( groups = { "util", "medium" } )
```

由于用户可以自由地标注方法所属的测试组，因此这种机制能让用户在方法级别对测试进行归类。这一点 JUnit 无法做到，它只能实现类级别的测试归类。

Maven 用户可以使用代码清单 10-19 所示的配置运行一个或者多个 TestNG 测试组。

代码清单 10-19　配置 maven-surefire-plugin 运行 TestNG 测试组

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.5</version>
    <configuration>
        <groups>util,medium</groups>
    </configuration>
</plugin>
```

由于篇幅所限，这里不再介绍更多 TestNG 的测试技术，感兴趣的读者请访问 TestNG 站点。

10.8　重用测试代码

优秀的程序员会像对待产品代码一样细心维护测试代码，尤其是那些供具体测试类继承的抽象类，它们能够简化测试代码的编写。还有一些根据具体项目环境对测试框架的扩展，也会被大范围地重用。

在命令行运行 **mvn package** 的时候，Maven 会将项目的主代码及资源文件打包，将其安装或部署到仓库之后，这些代码就能为他人使用，从而实现 Maven 项目级别的重用。默认的打包行为是不会包含测试代码的，因此在使用外部依赖的时候，其构件一般都不会包含测试代码。

然后，在项目内部重用某个模块的测试代码是很常见的需求，可能某个底层模块的测试代码中包含了一些常用的测试工具类，或者一些高质量的测试基类供继承。这个时候 Maven 用户就需要通过配置 maven-jar-plugin 将测试类打包，如代码清单 10-20 所示。

代码清单 10-20　打包测试代码

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.2</version>
    <executions>
        <execution>
            <goals>
                <goal>test-jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

maven-jar-plugin 有两个目标，分别是 jar 和 test-jar，前者通过 Maven 的内置绑定在 default 生命周期的 package 阶段运行，其行为就是对项目主代码进行打包，而后者并没有内置绑定，因此上述的插件配置显式声明该目标来打包测试代码。通过查询该插件的具体信息可以了解到，test-jar 的默认绑定生命周期阶段为 package，因此当运行 **mvn clean package** 后就会看到如下输出：

```
[INFO] ---maven-jar-plugin:2.2:jar (default-jar) @ account-captcha ---
[INFO] Building jar: D: \code \ch - 10 \account-aggregator \account-captcha \target \
account-captcha - 1.0.0 - SNAPSHOT.jar
[INFO]
[INFO] ---maven-jar-plugin:2.2:test-jar (default) @ account-captcha ---
[INFO] Building jar: D: \code \ch - 10 \account-aggregator \account-captcha \target \
account-captcha - 1.0.0 - SNAPSHOT - tests.jar
```

maven-jar-plugin 的两个目标都得以执行，分别打包了项目主代码和测试代码。

现在，就可以通过依赖声明使用这样的测试包构件了，如代码清单 10-21 所示。

代码清单 10-21　依赖测试包构件

```
<dependency>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-captcha</artifactId>
    <version>1.0.0 - SNAPSHOT</version>
    <type>test-jar</type>
    <scope>test</scope>
</dependency>
```

上述依赖声明中有一个特殊的元素 type，所有测试包构件都使用特殊的 test-jar 打包类型。需要注意的是，这一类型的依赖同样都使用 test 依赖范围。

10.9　小结

本章的主题是 Maven 与测试的集成，不过在讲述具体的测试技巧之前先实现了背景案例的 account-captcha 模块，这一模块的测试代码也成了本章其他内容良好的素材。maven-surefire-plugin 是 Maven 背后真正执行测试的插件，它有一组默认的文件名模式来匹配并自动运行测试类。用户还可以使用该插件来跳过测试、动态执行测试类、包含或排除测试等。maven-surefire-plugin 能生成基本的测试报告，除此之外还能使用 cobertura-maven-plugin 生成测试覆盖率报告。

除了主流的 JUnit 之外，本章还讲述了如何与 TestNG 集成，最后介绍了如何重用测试代码。