第5章

坐标和依赖

本章内容

- ❑ 何为 Maven 坐标
- ❑ 坐标详解
- ❑ account-email
- ❑ 依赖的配置
- ❑ 依赖范围
- ❑ 传递性依赖
- ❑ 依赖调解
- ❑ 可选依赖
- ❑ 最佳实践
- ❑ 小结

正如第 1 章所述，Maven 的一大功能是管理项目依赖。为了能自动化地解析任何一个 Java 构件，Maven 就必须将它们唯一标识，这就依赖管理的底层基础——坐标。本章将详细分析 Maven 坐标的作用，解释其每一个元素；在此基础上，再介绍如何配置 Maven，以及相关的经验和技巧，以帮助我们管理项目依赖。

5.1 何为 Maven 坐标

关于坐标（Coordinate），大家最熟悉的定义应该来自于平面几何。在一个平面坐标系中，坐标（x，y）表示该平面上与 x 轴距离为 y，与 y 轴距离为 x 的一点，任何一个坐标都能够唯一标识该平面中的一点。

在实际生活中，我们也可以将地址看成是一种坐标。省、市、区、街道等一系列信息同样可以唯一标识城市中的任一居住地址和工作地址。邮局和快递公司正是基于这样一种坐标进行日常工作的。

对应于平面中的点和城市中的地址，Maven 的世界中拥有数量非常巨大的构件，也就是平时用的一些 jar、war 等文件。在 Maven 为这些构件引入坐标概念之前，我们无法使用任何一种方式来唯一标识所有这些构件。因此，当需要用到 Spring Framework 依赖的时候，大家会去 Spring Framework 网站寻找，当需要用到 log4j 依赖的时候，大家又会去 Apache 网站寻找。又因为各个项目的网站风格迥异，大量的时间花费在了搜索、浏览网页等工作上面。没有统一的规范、统一的法则，该工作就无法自动化。重复地搜索、浏览网页和下载类似的 jar 文件，这本就应该交给机器来做。而机器工作必须基于预定义的规则，Maven 定义了这样一组规则：世界上任何一个构件都可以使用 Maven 坐标唯一标识，Maven 坐标的元素包括 groupId、artifactId、version、packaging、classifier。现在，只要我们提供正确的坐标元素，Maven 就能找到对应的构件。比如说，当需要使用 Java5 平台上 TestNG 的 5.8 版本时，就告诉 Maven："groupId = org. testng；artifactId = testng；version = 5.8；classifier = jdk15"，Maven 就会从仓库中寻找相应的构件供我们使用。也许你会奇怪，"Maven 是从哪里下载构件的呢？"答案其实很简单，Maven 内置了一个中央仓库的地址（http://repo1. maven. org/maven2），该中央仓库包含了世界上大部分流行的开源项目构件，Maven 会在需要的时候去那里下载。

在我们开发自己项目的时候，也需要为其定义适当的坐标，这是 Maven 强制要求的。在这个基础上，其他 Maven 项目才能引用该项目生成的构件，见图 5-1。

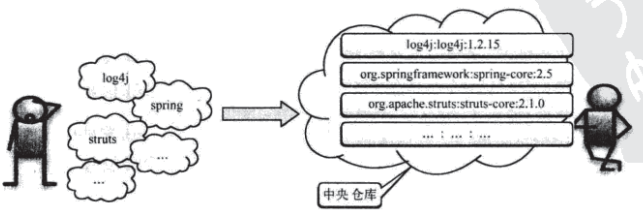

图 5-1 坐标为构件引入秩序

5.2 坐标详解

Maven 坐标为各种构件引入了秩序，任何一个构件都必须明确定义自己的坐标，而一组 Maven 坐标是通过一些元素定义的，它们是 groupId、artifactId、version、packaging、classifier。先看一组坐标定义，如下：

```
<groupId>org.sonatype.nexus</groupId>
<artifactId>nexus-indexer</artifactId>
<version>2.0.0</version>
<packaging>jar</packaging>
```

这是 nexus-indexer 的坐标定义，nexus-indexer 是一个对 Maven 仓库编纂索引并提供搜索功能的类库，它是 Nexus 项目的一个子模块。后面会详细介绍 Nexus。上述代码片段中，其坐标分别为 groupId：org.sonatype.nexus、artifactId：nexus-indexer、version：2.0.0、packaging：jar，没有 classifier。下面详细解释一下各个坐标元素：

- **groupId**：定义当前 Maven 项目隶属的实际项目。首先，Maven 项目和实际项目不一定是一对一的关系。比如 SpringFramework 这一实际项目，其对应的 Maven 项目会有很多，如 spring-core、spring-context 等。这是由于 Maven 中模块的概念，因此，一个实际项目往往会被划分成很多模块。其次，groupId 不应该对应项目隶属的组织或公司。原因很简单，一个组织下会有很多实际项目，如果 groupId 只定义到组织级别，而后面我们会看到，artifactId 只能定义 Maven 项目（模块），那么实际项目这个层将难以定义。最后，groupId 的表示方式与 Java 包名的表示方式类似，通常与域名反向一一对应。上例中，groupId 为 org.sonatype.nexus，org.sonatype 表示 Sonatype 公司建立的一个非盈利性组织，nexus 表示 Nexus 这一实际项目，该 groupId 与域名 nexus.sonatype.org 对应。
- **artifactId**：该元素定义实际项目中的一个 Maven 项目（模块），推荐的做法是使用实际项目名称作为 artifactId 的前缀。比如上例中的 artifactId 是 nexus-indexer，使用了实际项目名 nexus 作为前缀，这样做的好处是方便寻找实际构件。在默认情况下，Maven 生成的构件，其文件名会以 artifactId 作为开头，如 nexus-indexer-2.0.0.jar，使用实际项目名称作为前缀之后，就能方便从一个 lib 文件夹中找到某个项目的一组构件。考虑有 5 个项目，每个项目都有一个 core 模块，如果没有前缀，我们会看到很多 core-1.2.jar 这样的文件，加上实际项目名前缀之后，便能很容易区分 foo-core-1.2.jar、bar-core-1.2.jar……
- **version**：该元素定义 Maven 项目当前所处的版本，如上例中 nexus-indexer 的版本是 2.0.0。需要注意的是，Maven 定义了一套完成的版本规范，以及快照（SNAPSHOT）的概念。第 13 章会详细讨论版本管理内容。
- **packaging**：该元素定义 Maven 项目的打包方式。首先，打包方式通常与所生成构件的文件扩展名对应，如上例中 packaging 为 jar，最终的文件名为 nexus-indexer-

2.0.0.jar，而使用 war 打包方式的 Maven 项目，最终生成的构件会有一个 .war 文件，不过这不是绝对的。其次，打包方式会影响到构建的生命周期，比如 jar 打包和 war 打包会使用不同的命令。最后，当不定义 packaging 的时候，Maven 会使用默认值 jar。

- **classifier**：该元素用来帮助定义构建输出的一些附属构件。附属构件与主构件对应，如上例中的主构件是 nexus-indexer-2.0.0.jar，该项目可能还会通过使用一些插件生成如 nexus-indexer-2.0.0-javadoc.jar、nexus-indexer-2.0.0-sources.jar 这样一些附属构件，其包含了 Java 文档和源代码。这时候，javadoc 和 sources 就是这两个附属构件的 classifier。这样，附属构件也就拥有了自己唯一的坐标。还有一个关于 classifier 的典型例子是 TestNG，TestNG 的主构件是基于 Java 1.4 平台的，而它又提供了一个 classifier 为 jdk5 的附属构件。注意，不能直接定义项目的 classifier，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成。

上述 5 个元素中，groupId、artifactId、version 是必须定义的，packaging 是可选的（默认为 jar），而 classifier 是不能直接定义的。

同时，项目构件的文件名是与坐标相对应的，一般的规则为 artifactId-version［-classifier］.packaging，［-classifier］表示可选。比如上例 nexus-indexer 的主构件为 nexus-indexer-2.0.0.jar，附属构件有 nexus-indexer-2.0.0-javadoc.jar。这里还要强调的一点是，packaging 并非一定与构件扩展名对应，比如 packaging 为 maven-plugin 的构件扩展名为 jar。

此外，Maven 仓库的布局也是基于 Maven 坐标，这一点会在介绍 Maven 仓库的时候详细解释。

理解清楚城市中地址的定义方式后，邮递员就能够开始工作了；同样地，理解清楚 Maven 坐标之后，我们就能开始讨论 Maven 的依赖管理了。

5.3 account-email

在详细讨论 Maven 依赖之前，先稍微回顾一下上一章提到的背景案例。案例中有一个 email 模块负责发送账户激活的电子邮件，本节就详细阐述该模块的实现，包括 POM 配置、主代码和测试代码。由于该背景案例的实现是基于 Spring Framework，因此还会涉及相关的 Spring 配置。

5.3.1 account-email 的 POM

首先看一下该模块的 POM，见代码清单 5-1。

代码清单 5-1 · account-email 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>
<groupId>com.juvenxu.mvnbook.account</groupId>
<artifactId>account-email</artifactId>
<name>Account Email</name>
<version>1.0.0-SNAPSHOT</version>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>2.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>2.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>2.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>2.5.6</version>
    </dependency>
    <dependency>
        <groupId>javax.mail</groupId>
        <artifactId>mail</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.7</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.icegreen</groupId>
        <artifactId>greenmail</artifactId>
        <version>1.3.1b</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.5</source>
                <target>1.5</target>
            </configuration>
```

```
            </plugin>
        </plugins>
    </build>

</project>
```

先观察该项目模块的坐标，groupId：com. juvenxu. mvnbook. account；artifactId：account-email；version：1. 0. 0-SNAPSHOT。由于该模块属于账户注册服务项目的一部分，因此，其 groupId 对应了 account 项目。紧接着，该模块的 artifactId 仍然以 account 作为前缀，以方便区分其他项目的构建。最后，1. 0. 0-SNAPSHOT 表示该版本处于开发中，还不稳定。

再看 dependencies 元素，其包含了多个 dependency 子元素，这是 POM 中定义项目依赖的位置。以第一个依赖为例，其 groupId：artifactId：version 为 org. springframework：spring-core：2. 5. 6，这便是依赖的坐标，任何一个 Maven 项目都需要定义自己的坐标，当这个 Maven 项目成为其他 Maven 项目的依赖的时候，这组坐标就体现了其价值。本例中的 spring-core，以及后面的 spring-beans、spring-context、spring-context-support 是 Spring Framework 实现依赖注入等功能必要的构件，由于本书的关注点在于 Maven，只会涉及简单的 Spring Framework 的使用，不会详细解释 Spring Framework 的用法，如果读者有不清楚的地方，请参阅 Spring Framework 相关的文档。

在 spring-context-support 之后，有一个依赖为 javax. mail：mail：1. 4. 1，这是实现发送必须的类库。

紧接着的依赖为 junit：junit：4. 7，JUnit 是 Java 社区事实上的单元测试标准，详细信息请参阅 http://www. junit. org/，这个依赖特殊的地方在于一个值为 test 的 scope 子元素，scope 用来定义依赖范围。这里读者暂时只需要了解当依赖范围是 test 的时候，该依赖只会被加入到测试代码的 classpath 中。也就是说，对于项目主代码，该依赖是没有任何作用的。JUnit 是单元测试框架，只有在测试的时候才需要，因此使用该依赖范围。

随后的依赖是 com. icegreen：greenmail：1. 3. 1b，其依赖范围同样为 test。这时也许你已经猜到，该依赖同样只服务于测试目的，GreenMail 是开源的邮件服务测试套件，account-email 模块使用该套件来测试邮件的发送。关于 GreenMail 的详细信息可访问 http://www. icegreen. com/greenmail/。

最后，POM 中有一段关于 maven-compiler-plugin 的配置，其目的是开启 Java 5 的支持，第 3 章已经对该配置做过解释，这里不再赘述。

5. 3. 2 account-email 的主代码

account-email 项目 Java 主代码位于 src/main/java，资源文件（非 Java）位于 src/main/resources 目录下。

account-email 只有一个很简单的接口，见代码清单 5-2。

代码清单 5-2　AccountEmailService. java

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

public interface AccountEmailService
{
    void sendMail( String to, String subject, String htmlText )
        throws AccountEmailException;
}
```

sendMail( )方法用来发送 html 格式的邮件，to 为接收地址，subject 为邮件主题，html-Text 为邮件内容，如果发送邮件出错，则抛出 AccountEmailException 异常。

对应于该接口的实现见代码清单 5-3。

代码清单 5-3　AccountEmailServiceImpl. java

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

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

public class AccountEmailServiceImpl
    implements AccountEmailService
{
    private JavaMailSender javaMailSender;

    private String systemEmail;

    public void sendMail( String to, String subject, String htmlText )
        throws AccountEmailException
    {
        try
        {
            MimeMessage msg = javaMailSender.createMimeMessage();
            MimeMessageHelper msgHelper = new MimeMessageHelper( msg );

            msgHelper.setFrom( systemEmail );
            msgHelper.setTo( to );
            msgHelper.setSubject( subject );
            msgHelper.setText( htmlText, true );

            javaMailSender.send( msg );
        }
        catch ( MessagingException e )
        {
            throw new AccountEmailException( "Faild to send mail.", e );
        }
    }

    public JavaMailSender getJavaMailSender()
    {
```

```
        return javaMailSender;
    }

    public void setJavaMailSender( JavaMailSender javaMailSender )
    {
        this.javaMailSender = javaMailSender;
    }

    public String getSystemEmail()
    {
        return systemEmail;
    }

    public void setSystemEmail( String systemEmail )
    {
        this.systemEmail = systemEmail;
    }
}
```

首先，该 AccountEmailServiceImpl 类有一个私有字段 javaMailSender，该字段的类型 org. springframework. mail. javamail. JavaMailSender 是来自于 Spring Framework 的帮助简化邮件发送的工具类库，对应于该字段有一组 getter( )和 setter( )方法，它们用来帮助实现依赖注入。本节随后会讲述 Spring Framework 依赖注入相关的配置。

在 sendMail( )的方法实现中，首先使用 javaMailSender 创建一个 MimeMessage，该 msg 对应了将要发送的邮件。接着使用 MimeMessageHelper 帮助设置该邮件的发送地址、收件地址、主题以及内容，msgHelper. setText（htmlText, true ）中的 true 表示邮件的内容为 html 格式。最后，使用 javaMailSender 发送该邮件，如果发送出错，则捕捉 MessageException 异常，包装后再抛出该模块自己定义的 AccountEmailException 异常。

这段 Java 代码中没有邮件服务器配置信息，这得益于 Spring Framework 的依赖注入，这些配置都通过外部的配置注入到了 javaMailSender 中，相关配置信息都在 src/main/resources/account-email. xml 这个配置文件中，见代码清单 5-4。

代码清单 5-4　account-email. 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 = "propertyConfigurer"
    class = "org.springframework.beans.factory.config.PropertyPlaceholder-
Configurer" >
        <property name = "location" value = "classpath:service.properties" />
    </bean>

    <bean id = "javaMailSender" class = "org.springframework.mail.javamail.JavaMail-
SenderImpl" >
        <property name = "protocol" value = "${email.protocol}" />
```

```
        <property name="host" value="${email.host}" />
        <property name="port" value="${email.port}" />
        <property name="username" value="${email.username}" />
        <property name="password" value="${email.password}" />
        <property name="javaMailProperties">
            <props>
                <prop key="mail.${email.protocol}.auth">${email.auth}</prop>
            </props>
        </property>
    </bean>

    <bean id="accountEmailService"
        class="com.juvenxu.mvnbook.account.email.AccountEmailServiceImpl">
        <property name="javaMailSender" ref="javaMailSender" />
        <property name="systemEmail" value="${email.systemEmail}" />
    </bean>
</beans>
```

Spring Framework 会使用该 XML 配置创建 ApplicationContext，以实现依赖注入。该配置文件定义了一些 bean，基本对应了 Java 程序中的对象。首先解释下 id 为 propertyConfigurer 的 bean，其实现为 org.springframework.beans.factory.config.PropertyPlaceholderConfigurer，这是 Spring Framework 中用来帮助载入 properties 文件的组件。这里定义 location 的值为 classpath:account-email.properties，表示从 classpath 的根路径下载入名为 account-email.properties 文件中的属性。

接着定义 id 为 javaMailSender 的 bean，其实现为 org.springframework.mail.javamail.JavaMailSenderImpl，这里需要定义邮件服务器的一些配置，包括协议、端口、主机、用户名、密码，是否需要认证等属性。这段配置还使用了 Spring Framework 的属性引用，比如 host 的值为 ${email.host}，之前定义 propertyConfigurer 的作用就在于此。这么做可以将邮件服务器相关的配置分离到外部的 properties 文件中，比如可以定义这样一个 properties 文件，配置 javaMailSender 使用 gmail：

```
email.protocol=smtps
email.host=smtp.gmail.com
email.port=465
email.username=your-id@gmail.com
email.password=your-password
email.auth=true
email.systemEmail=your-id@juvenxu.com
```

这样，javaMailSender 实际使用的 protocol 就会成为 smtps，host 会成为 smtp.gmail.com，同理还有 port、username 等其他属性。

最后一个 bean 是 accountEmailService，对应了之前描述的 com.juvenxu.mvnbook.account.email.AccountEmailServiceImpl，配置中将另外一个 bean javaMailSender 注入，使其成为该类 javaMailSender 字段的值。

上述就是 Spring Framework 相关的配置，这里不再进一步深入，读者如果有不是很理解的地方，请查询 Spring Framework 相关文档。

5.3.3 account-email 的测试代码

测试相关的 Java 代码位于 src/test/java 目录，相关的资源文件则位于 src/test/resources 目录。

该模块需要测试的只有一个 AccountEmailService. sendMail( )接口。为此，需要配置并启动一个测试使用的邮件服务器，然后提供对应的 properties 配置文件供 Spring Framework 载入以配置程序。准备就绪之后，调用该接口发送邮件，然后检查邮件是否发送正确。最后，关闭测试邮件服务器，见代码清单 5-5。

代码清单 5-5 AccountEmailServiceTest. java

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

import static junit.framework.Assert.assertEquals;

import javax.mail.Message;

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

import com.icegreen.greenmail.util.GreenMail;
import com.icegreen.greenmail.util.GreenMailUtil;
import com.icegreen.greenmail.util.ServerSetup;

public class AccountEmailServiceTest
{
    private GreenMail greenMail;

    @ Before
    public void startMailServer()
        throws Exception
    {
        greenMail = new GreenMail( ServerSetup.SMTP );
        greenMail.setUser( "test@ juvenxu.com", "123456" );
        greenMail.start();
    }

    @ Test
    public void testSendMail()
        throws Exception
    {
        ApplicationContext ctx = new ClassPathXmlApplicationContext( "account-
email.xml" );
            AccountEmailService accountEmailService = (AccountEmailService)
ctx.getBean( "accountEmailService" );

        String subject = "Test Subject";
        String htmlText = "<h3>Test</h3>";
```

```
        accountEmailService.sendMail( "test@ juvenxu.com", "test2@ juvenxu.
com", subject, htmlText );

        greenMail.waitForIncomingEmail(2000, 1 );

        Message[] msgs = greenMail.getReceivedMessages();
        assertEquals(1, msgs.length );
        assertEquals( subject, msgs[0].getSubject() );
        assertEquals( htmlText, GreenMailUtil.getBody( msgs[0] ).trim() );
    }

    @ After
    public void stopMailServer()
        throws Exception
    {
        greenMail.stop();
    }
}
```

这里使用 GreenMail 作为测试邮件服务器，在 startMailServer( )中，基于 SMTP 协议初始化 GreenMail，然后创建一个邮件账户并启动邮件服务，该服务默认会监听 25 端口。如果你的机器已经有程序使用该端口，请配置自定义的 ServerSetup 实例使用其他端口。startMailServer( )方法使用了@ before 标注，表示该方法会先于测试方法（@ test）之前执行。

对应于 startMailServer( )，该测试还有一个 stopMailServer( )方法，标注@ After 表示执行测试方法之后会调用该方法，停止 GreenMail 的邮件服务。

代码的重点在于使用了@ Test 标注的 testSendMail( )方法，该方法首先会根据 classpath 路径中的 account-email. xml 配置创建一个 Spring Framework 的 ApplicationContext，然后从这个 ctx 中获取需要测试的 id 为 accountEmailService 的 bean，并转换成 AccountEmailService 接口，针对接口测试是一个单元测试的最佳实践。得到了 AccountEmailService 之后，就能调用其 sendMail( )方法发送电子邮件。当然，这个时候不能忘了邮件服务器的配置，其位于 src/test/resources/service. properties：

```
email.protocol = smtp
email.host = localhost
email.port = 25
email.username = test@ juvenxu.com
email.password = 123456
email.auth = true
email.systemEmail = your-id@ juvenxu.com
```

这段配置与之前 GreenMail 的配置对应，使用了 smtp 协议，使用本机的 25 端口，并有用户名、密码等认证配置。

回到测试方法中，邮件发送完毕后，再使用 GreenMail 进行检查。greenMail. waitForIncomingEmail（2000，1）表示接收一封邮件，最多等待 2 秒。由于 GreenMail 服务完全基于内存，实际情况下基本不会超过 2 秒。随后的几行代码读取收到的邮件，检查邮件的数目以及第一封邮件的主题和内容。

这时，可以运行 mvn clean test 执行测试，Maven 会编译主代码和测试代码，并执行测试，报告一个测试得以正确执行，构建成功。

5.3.4 构建 account-email

使用 mvn clean install 构建 account-email，Maven 会根据 POM 配置自动下载所需要的依赖构件，执行编译、测试、打包等工作，最后将项目生成的构件 account-email-1.0.0-SNAPSHOT.jar 安装到本地仓库中。这时，该模块就能供其他 Maven 项目使用了。

5.4 依赖的配置

5.3.1 节已经罗列了一些简单的依赖配置，读者可以看到依赖会有基本的 groupId、artifactId 和 version 等元素组成。其实一个依赖声明可以包含如下的一些元素：

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>...</groupId>
      <artifactId>...</artifactId>
      <version>...</version>
      <type>...</type>
      <scope>...</scope>
      <optional>...</optional>
      <exclusions>
        <exclusion>
          ...
        </exclusion>
        ...
      </exclusions>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

根元素 project 下的 dependencies 可以包含一个或者多个 dependency 元素，以声明一个或者多个项目依赖。每个依赖可以包含的元素有：

- **groupId**、**artifactId** 和 **version**：依赖的基本坐标，对于任何一个依赖来说，基本坐标是最重要的，Maven 根据坐标才能找到需要的依赖。
- **type**：依赖的类型，对应于项目坐标定义的 packaging。大部分情况下，该元素不必声明，其默认值为 jar。
- **scope**：依赖的范围，见 5.5 节。
- **optional**：标记依赖是否可选，见 5.8 节。
- **exclusions**：用来排除传递性依赖，见 5.9.1 节。

大部分依赖声明只包含基本坐标，然而在一些特殊情况下，其他元素至关重要。本章

下面的小节会对它们的原理和使用方式详细介绍。

5.5　依赖范围

上一节提到，JUnit 依赖的测试范围是 test，测试范围用元素 scope 表示。本节将详细解释什么是测试范围，以及各种测试范围的效果和用途。

首先需要知道，Maven 在编译项目主代码的时候需要使用一套 classpath。在上例中，编译项目主代码的时候需要用到 spring-core，该文件以依赖的方式被引入到 classpath 中。其次，Maven 在编译和执行测试的时候会使用另外一套 classpath。上例中的 JUnit 就是一个很好的例子，该文件也以依赖的方式引入到测试使用的 classpath 中，不同的是这里的依赖范围是 test。最后，实际运行 Maven 项目的时候，又会使用一套 classpath，上例中的 spring-core 需要在该 classpath 中，而 JUnit 则不需要。

依赖范围就是用来控制依赖与这三种 classpath（编译 classpath、测试 classpath、运行 classpath）的关系，Maven 有以下几种依赖范围：

- **compile**：编译依赖范围。如果没有指定，就会默认使用该依赖范围。使用此依赖范围的 Maven 依赖，对于编译、测试、运行三种 classpath 都有效。典型的例子是 spring-core，在编译、测试和运行的时候都需要使用该依赖。
- **test**：测试依赖范围。使用此依赖范围的 Maven 依赖，只对于测试 classpath 有效，在编译主代码或者运行项目的使用时将无法使用此类依赖。典型的例子是 JUnit，它只有在编译测试代码及运行测试的时候才需要。
- **provided**：已提供依赖范围。使用此依赖范围的 Maven 依赖，对于编译和测试 classpath 有效，但在运行时无效。典型的例子是 servlet-api，编译和测试项目的时候需要该依赖，但在运行项目的时候，由于容器已经提供，就不需要 Maven 重复地引入一遍。
- **runtime**：运行时依赖范围。使用此依赖范围的 Maven 依赖，对于测试和运行 classpath 有效，但在编译主代码时无效。典型的例子是 JDBC 驱动实现，项目主代码的编译只需要 JDK 提供的 JDBC 接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体 JDBC 驱动。
- **system**：系统依赖范围。该依赖与三种 classpath 的关系，和 provided 依赖范围完全一致。但是，使用 system 范围的依赖时必须通过 systemPath 元素显式地指定依赖文件的路径。由于此类依赖不是通过 Maven 仓库解析的，而且往往与本机系统绑定，可能造成构建的不可移植，因此应该谨慎使用。systemPath 元素可以引用环境变量，如：

```
<dependency>
  <groupId>javax.sql</groupId>
  <artifactId>jdbc-stdext</artifactId>
  <version>2.0</version>
```

```
    <scope>system</scope>
    <systemPath>${java.home}/lib/rt.jar</systemPath>
</dependency>
```

- **import（Maven 2.0.9 及以上）**：导入依赖范围。该依赖范围不会对三种 classpath 产生实际的影响，本书将在 8.3.3 节介绍 Maven 依赖和 dependencyManagement 的时候详细介绍此依赖范围。

上述除 import 以外的各种依赖范围与三种 classpath 的关系如表 5-1 所示。

表 5-1 依赖范围与 classpath 的关系

依赖范围（Scope）	对于编译 classpath 有效	对于测试 classpath 有效	对于运行时 classpath 有效	例 子
compile	Y	Y	Y	spring-core
test	—	Y	—	JUnit
provided	Y	Y	—	servlet-api
runtime	—	Y	Y	JDBC 驱动实现
system	Y	Y	—	本地的，Maven 仓库之外的类库文件

5.6 传递性依赖

5.6.1 何为传递性依赖

考虑一个基于 Spring Framework 的项目，如果不使用 Maven，那么在项目中就需要手动下载相关依赖。由于 Spring Framework 又会依赖于其他开源类库，因此实际中往往会下载一个很大的如 spring-framework-2.5.6-with-dependencies.zip 的包，这里包含了所有 Spring Framework 的 jar 包，以及所有它依赖的其他 jar 包。这么做往往就引入了很多不必要的依赖。另一种做法是只下载 spring-framework-2.5.6.zip 这样一个包，这里不包含其他相关依赖，到实际使用的时候，再根据出错信息，或者查询相关文档，加入需要的其他依赖。很显然，这也是一件非常麻烦的事情。

Maven 的传递性依赖机制可以很好地解决这一问题。以 account-email 项目为例，该项目有一个 org.springframework:spring-core:2.5.6 的依赖，而实际上 spring-core 也有它自己的依赖，我们可以直接访问位于中央仓库的该构件的 POM：http://repo1.maven.org/maven2/org/springframework/spring-core/2.5.6/spring-core-2.5.6.pom。该文件包含了一个 commons-logging 依赖，见代码清单 5-6。

代码清单 5-6 spring-core 的 commons-logging 依赖

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
```

```
    <version>1.1.1</version>
</dependency>
```

该依赖没有声明依赖范围，那么其依赖范围就是默认的 compile。同时回顾一下 account-email，spring-core 的依赖范围也是 compile。

account-mail 有一个 compile 范围的 spring-core 依赖，spring-core 有一个 compile 范围的 commons-logging 依赖，那么 commons-logging 就会成为 account-email 的 compile 范围依赖，commons-logging 是 account-email 的一个传递性依赖，如图 5-2 所示。

图 5-2　传递性依赖

有了传递性依赖机制，在使用 Spring Framework 的时候就不用去考虑它依赖了什么，也不用担心引入多余的依赖。Maven 会解析各个直接依赖的 POM，将那些必要的间接依赖，以传递性依赖的形式引入到当前的项目中。

5.6.2　传递性依赖和依赖范围

依赖范围不仅可以控制依赖与三种 classpath 的关系，还对传递性依赖产生影响。上面的例子中，account-email 对于 spring-core 的依赖范围是 compile，spring-core 对于 commons-logging 的依赖范围是 compile，那么 account-email 对于 commons-logging 这一传递性依赖的范围也就是 compile。假设 A 依赖于 B，B 依赖于 C，我们说 A 对于 B 是第一直接依赖，B 对于 C 是第二直接依赖，A 对于 C 是传递性依赖。第一直接依赖的范围和第二直接依赖的范围决定了传递性依赖的范围，如表 5-2 所示，最左边一行表示第一直接依赖范围，最上面一行表示第二直接依赖范围，中间的交叉单元格则表示传递性依赖范围。

表 5-2　依赖范围影响传递性依赖

	compile	test	provided	runtime
compile	compile	—	—	runtime
test	test	—	—	test
provided	provided	—	provided	provided
runtime	runtime	—	—	runtime

为了能够帮助读者更好地理解表 5-2，这里再举个例子。account-email 项目有一个 com. icegreen:greenmail:1. 3. 1b 的直接依赖，我们说这是第一直接依赖，其依赖范围是 test；而 greenmail 又有一个 javax. mail:mail:1. 4 的直接依赖，我们说这是第二直接依赖，其依赖范围是 compile。显然 javax. mail:mail:1. 4 是 account-email 的传递性依赖，对照表 5-2 可以知道，当第一直接依赖范围为 test，第二直接依赖范围是 compile 的时候，传递性依赖的范

围是 test，因此 javax. mail:mail:1. 4 是 account-email 的一个范围是 test 的传递性依赖。

仔细观察一下表 5-2，可以发现这样的规律：当第二直接依赖的范围是 compile 的时候，传递性依赖的范围与第一直接依赖的范围一致；当第二直接依赖的范围是 test 的时候，依赖不会得以传递；当第二直接依赖的范围是 provided 的时候，只传递第一直接依赖范围也为 provided 的依赖，且传递性依赖的范围同样为 provided；当第二直接依赖的范围是 runtime 的时候，传递性依赖的范围与第一直接依赖的范围一致，但 compile 例外，此时传递性依赖的范围为 runtime。

5.7 依赖调解

Maven 引入的传递性依赖机制，一方面大大简化和方便了依赖声明，另一方面，大部分情况下我们只需要关心项目的直接依赖是什么，而不用考虑这些直接依赖会引入什么传递性依赖。但有时候，当传递性依赖造成问题的时候，我们就需要清楚地知道该传递性依赖是从哪条依赖路径引入的。

例如，项目 A 有这样的依赖关系：A->B->C->X(1.0)、A->D->X(2.0)，X 是 A 的传递性依赖，但是两条依赖路径上有两个版本的 X，那么哪个 X 会被 Maven 解析使用呢？两个版本都被解析显然是不对的，因为那会造成依赖重复，因此必须选择一个。Maven 依赖调解（Dependency Mediation）的第一原则是：路径最近者优先。该例中 X(1.0)的路径长度为 3，而 X(2.0)的路径长度为 2，因此 X(2.0)会被解析使用。

依赖调解第一原则不能解决所有问题，比如这样的依赖关系：A->B->Y(1.0)、A->C->Y(2.0)，Y(1.0)和 Y(2.0)的依赖路径长度是一样的，都为 2。那么到底谁会被解析使用呢？在 Maven 2.0.8 及之前的版本中，这是不确定的，但是从 Maven 2.0.9 开始，为了尽可能避免构建的不确定性，Maven 定义了依赖调解的第二原则：第一声明者优先。在依赖路径长度相等的前提下，在 POM 中依赖声明的顺序决定了谁会被解析使用，顺序最靠前的那个依赖优胜。该例中，如果 B 的依赖声明在 C 之前，那么 Y（1.0）就会被解析使用。

5.8 可选依赖

假设有这样一个依赖关系，项目 A 依赖于项目 B，项目 B 依赖于项目 X 和 Y，B 对于 X 和 Y 的依赖都是可选依赖：A->B、B->X(可选)、B->Y(可选)。根据传递性依赖的定义，如果所有这三个依赖的范围都是 compile，那么 X、Y 就是 A 的 compile 范围传递性依赖。然而，由于这里 X、Y 是可选依赖，依赖将不会得以传递。换句话说，X、Y 将不会对 A 有任何影响，如图 5-3 所示。

为什么要使用可选依赖这一特性呢？可能项目 B 实现了两个特性，其中的特性一依赖于 X，特性二依赖于 Y，而且这两个特性是互斥的，用户不可能同时使用两个特性。比如 B 是一个持久层隔离工具包，它支持多种数据库，包括 MySQL、PostgreSQL 等。在构建这个

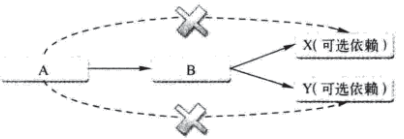

图 5-3　可选依赖

工具包的时候，需要这两种数据库的驱动程序，但在使用这个工具包的时候，只会依赖一种数据库。

项目 B 的依赖声明见代码清单 5-7。

代码清单 5-7　可选依赖的配置

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>project-b</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.10</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>8.4-701.jdbc3</version>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

上述 XML 代码片段中，使用 <optional> 元素表示 mysql-connector-java 和 postgresql 这两个依赖为可选依赖，它们只会对当前项目 B 产生影响，当其他项目依赖于 B 的时候，这两个依赖不会被传递。因此，当项目 A 依赖于项目 B 的时候，如果其实际使用基于 MySQL 数据库，那么在项目 A 中就需要显式地声明 mysql-connector-java 这一依赖，见代码清单 5-8。

代码清单 5-8　可选依赖不被传递

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>com.juvenxu.mvnbook</groupId>
```

```
    <artifactId>project-b</artifactId>
    <version>1.0.0</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.10</version>
  </dependency>
 </dependencies>
</project>
```

最后，关于可选依赖需要说明的一点是，在理想的情况下，是不应该使用可选依赖的。前面我们可以看到，使用可选依赖的原因是某一个项目实现了多个特性，在面向对象设计中，有个单一职责性原则，意指一个类应该只有一项职责，而不是糅合太多的功能。这个原则在规划 Maven 项目的时候也同样适用。在上面的例子中，更好的做法是为 MySQL 和 PostgreSQL 分别创建一个 Maven 项目，基于同样的 groupId 分配不同的 artifactId，如 com. juvenxu. mvnbook:project-b-mysql 和 com. juvenxu. mvnbook:project-b-postgresql，在各自的 POM 中声明对应的 JDBC 驱动依赖，而且不使用可选依赖，用户则根据需要选择使用 project-b-mysql 或者 project-b-postgresql。由于传递性依赖的作用，就不用再声明 JDBC 驱动依赖。

5.9 最佳实践

Maven 依赖涉及的知识点比较多，在理解了主要的功能和原理之后，最需要的当然就是前人的经验总结了，我们称之为最佳实践。本小节归纳了一些使用 Maven 依赖常见的技巧，方便用来避免和处理很多常见的问题。

5.9.1 排除依赖

传递性依赖会给项目隐式地引入很多依赖，这极大地简化了项目依赖的管理，但是有些时候这种特性也会带来问题。例如，当前项目有一个第三方依赖，而这个第三方依赖由于某些原因依赖了另外一个类库的 SNAPSHOT 版本，那么这个 SNAPSHOT 就会成为当前项目的传递性依赖，而 SNAPSHOT 的不稳定性会直接影响到当前的项目。这时就需要排除掉该 SNAPSHOT，并且在当前项目中声明该类库的某个正式发布的版本。还有一些情况，你可能也想要替换某个传递性依赖，比如 Sun JTA API，Hibernate 依赖于这个 JAR，但是由于版权的因素，该类库不在中央仓库中，而 Apache Geronimo 项目有一个对应的实现。这时你就可以排除 Sun JAT API，再声明 Geronimo 的 JTA API 实现，见代码清单 5-9。

代码清单 5-9 排除传递性依赖

```
<project>
  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.juvenxu.mvnbook</groupId>
<artifactId>project-a</artifactId>
<version>1.0.0</version>
<dependencies>
  <dependency>
    <groupId>com.juvenxu.mvnbook</groupId>
    <artifactId>project-b</artifactId>
    <version>1.0.0</version>
    <exclusions>
      <exclusion>
        <groupId>com.juvenxu.mvnbook</groupId>
        <artifactId>project-c</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>com.juvenxu.mvnbook</groupId>
    <artifactId>project-c</artifactId>
    <version>1.1.0</version>
  </dependency>
</dependencies>
</project>
```

上述代码中，项目 A 依赖于项目 B，但是由于一些原因，不想引入传递性依赖 C，而是自己显式地声明对于项目 C 1.1.0 版本的依赖。代码中使用 exclusions 元素声明排除依赖，exclusions 可以包含一个或者多个 exclusion 子元素，因此可以排除一个或者多个传递性依赖。需要注意的是，声明 exclusion 的时候只需要 groupId 和 artifactId，而不需要 version 元素，这是因为只需要 groupId 和 artifactId 就能唯一定位依赖图中的某个依赖。换句话说，Maven 解析后的依赖中，不可能出现 groupId 和 artifactId 相同，但是 version 不同的两个依赖，这一点在 5.6 节中已做过解释。该例的依赖解析逻辑如图 5-4 所示。

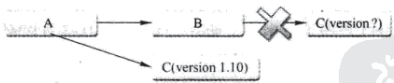

图 5-4 排除依赖

5.9.2 归类依赖

在 5.3.1 节中，有很多关于 Spring Framework 的依赖，它们分别是 org.springframework:spring-core:2.5.6、org.springframework:spring-beans:2.5.6、org.springframework:spring-context:2.5.6 和 org.springframework:spring-context-support:2.5.6，它们是来自同一项目的不同模块。因此，所有这些依赖的版本都是相同的，而且可以预见，如果将来需要升级 Spring Framework，这些依赖的版本会一起升级。这一情况在 Java 中似曾相识，考虑如下简单代码（见代码清单 5-10）。

代码清单 5-10 Java 中重复使用字面量

```
public double c( double r )
{
    return 2 * 3.14 * r;
}

public double s( double r )
{
    return 3.14 * r * r;
}
```

这两个简单的方程式计算圆的周长和面积，稍微有经验的程序员一眼就会看出一个问题，使用字面量（3.14）显然不合适，应该使用定义一个常量并在方法中使用，见代码清单 5-11。

代码清单 5-11 Java 中使用常量

```
public final double PI = 3.14;

public double c( double r )
{
    return 2 * PI * r;
}

public double s( double r )
{
    return PI * r * r;
}
```

使用常量不仅让代码变得更加简洁，更重要的是可以避免重复，在需要更改 PI 的值的时候，只需要修改一处，降低了错误发生的概率。

同理，对于 account-email 中这些 Spring Framework 来说，也应该在一个唯一的地方定义版本，并且在 dependency 声明中引用这一版本。这样，在升级 Spring Framework 的时候就只需要修改一处，实现方式见代码清单 5-12。

代码清单 5-12 使用 Maven 属性归类依赖

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.juven.mvnbook.account</groupId>
    <artifactId>account-email</artifactId>
    <name>Account Email</name>
    <version>1.0.0-SNAPSHOT</version>

    <properties>
      <springframework.version>2.5.6</springframework.version>
    </properties>

    <dependencies>
       <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>${springframework.version}</version>
    </dependency>
</dependencies>

</project>
```

这里简单用到了 Maven 属性（14.1 节会详细介绍 Maven 属性），首先使用 properties 元素定义 Maven 属性，该例中定义了一个 springframework.version 子元素，其值为 2.5.6。有了这个属性定义之后，Maven 运行的时候会将 POM 中的所有的 ${springframework.version} 替换成实际值 2.5.6。也就是说，可以使用美元符号和大括弧环绕的方式引用 Maven 属性。然后，将所有 Spring Framework 依赖的版本值用这一属性引用表示。这和在 Java 中用常量 PI 替换 3.14 是同样的道理，不同的只是语法。

5.9.3 优化依赖

在软件开发过程中，程序员会通过重构等方式不断地优化自己的代码，使其变得更简洁、更灵活。同理，程序员也应该能够对 Maven 项目的依赖了然于胸，并对其进行优化，如去除多余的依赖，显式地声明某些必要的依赖。

通过阅读本章前面的内容，读者应该能够了解到：Maven 会自动解析所有项目的直接依赖和传递性依赖，并且根据规则正确判断每个依赖的范围，对于一些依赖冲突，也能进行调节，以确保任何一个构件只有唯一的版本在依赖中存在。在这些工作之后，最后得到的那些依赖被称为已解析依赖（Resolved Dependency）。可以运行如下的命令查看当前项目的已解析依赖：

```
mvn dependency:list
```

在 account-email 项目中执行该命令，结果如图 5-5 所示。

图 5-5 显示了所有 account-email 的已解析依赖，同时，每个依赖的范围也得以明确标示。

在此基础上，还能进一步了解已解析依赖的信息。将直接在当前项目 POM 声明的依赖定义为顶层依赖，而这些顶层依赖的依赖则定义为第二层依赖，以此类推，有第三、第四

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Account Email
[INFO]    task-segment: [dependency:list]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:list (execution: default-cli)]
[INFO]
[INFO] The following files have been resolved:
[INFO]    aopalliance:aopalliance:jar:1.0:compile
[INFO]    com.icegreen:greenmail:jar:1.3.1b:test
[INFO]    commons-logging:commons-logging:jar:1.1.1:compile
[INFO]    javax.activation:activation:jar:1.1:compile
[INFO]    javax.mail:mail:jar:1.4.1:compile
[INFO]    junit:junit:jar:4.7:test
[INFO]    org.slf4j:slf4j-api:jar:1.3.1:test
[INFO]    org.springframework:spring-beans:jar:2.5.6:compile
[INFO]    org.springframework:spring-context:jar:2.5.6:compile
[INFO]    org.springframework:spring-context-support:jar:2.5.6:compile
[INFO]    org.springframework:spring-core:jar:2.5.6:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

图 5-5　已解析依赖列表

层依赖。当这些依赖经 Maven 解析后，就会构成一个依赖树，通过这棵依赖树就能很清楚地看到某个依赖是通过哪条传递路径引入的。可以运行如下命令查看当前项目的依赖树：

```
mvn dependency:tree
```

在 acccount-email 中执行该命令，效果如图 5-6 所示。

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Account Email
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree (execution: default-cli)]
[INFO] com.juven.mvnbook.account:account-email:jar:1.0.0-SNAPSHOT
[INFO] +- org.springframework:spring-core:jar:2.5.6:compile
[INFO] |  \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.springframework:spring-beans:jar:2.5.6:compile
[INFO] +- org.springframework:spring-context:jar:2.5.6:compile
[INFO] |  \- aopalliance:aopalliance:jar:1.0:compile
[INFO] +- org.springframework:spring-context-support:jar:2.5.6:compile
[INFO] +- javax.mail:mail:jar:1.4.1:compile
[INFO] |  \- javax.activation:activation:jar:1.1:compile
[INFO] +- junit:junit:jar:4.7:test
[INFO] \- com.icegreen:greenmail:jar:1.3.1b:test
[INFO]    \- org.slf4j:slf4j-api:jar:1.3.1:test
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

图 5-6　已解析依赖树

从图 5-6 中能够看到，虽然我们没有声明 org. slf4j:slf4j-api:1. 3 这一依赖，但它还是经过 com. icegreen:greenmail:1. 3 成为了当前项目的传递性依赖，而且其范围是 test。

使用 dependency:list 和 dependency:tree 可以帮助我们详细了解项目中所有依赖的具体信息，在此基础上，还有 dependency:analyze 工具可以帮助分析当前项目的依赖。

为了说明该工具的用途，先将 5. 3. 1 节中的 spring-context 这一依赖删除，然后构建项目，你会发现编译、测试和打包都不会有任何问题。通过分析依赖树，可以看到 spring-context 是 spring-context-support 的依赖，因此会得以传递到项目的 classspath 中。现在再运行如下命令：

```
mvn dependency:analyze
```

结果如图 5-7 所示。

```
[INFO] Preparing dependency:analyze
[INFO] [resources:resources (execution: default-resources)]
[WARNING] Using platform encoding (GB18030 actually) to copy filtered
[INFO] Copying 1 resource
[INFO] [compiler:compile (execution: default-compile)]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources (execution: default-testResources)]
[WARNING] Using platform encoding (GB18030 actually) to copy filtered
[INFO] Copying 1 resource
[INFO] [compiler:testCompile (execution: default-testCompile)]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze (execution: default-cli)]
[WARNING] Used undeclared dependencies found:
[WARNING]    org.springframework:spring-context:jar:2.5.6:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.springframework:spring-core:jar:2.5.6:compile
[WARNING]    org.springframework:spring-beans:jar:2.5.6:compile
[INFO] ------------------------------------------------------------
```

图 5-7　使用但未声明的依赖与声明但未使用的依赖

该结果中重要的是两个部分。首先是 Used undeclared dependencies，意指项目中使用到的，但是没有显式声明的依赖，这里是 spring-context。这种依赖意味着潜在的风险，当前项目直接在使用它们，例如有很多相关的 Java import 声明，而这种依赖是通过直接依赖传递进来的，当升级直接依赖的时候，相关传递性依赖的版本也可能发生变化，这种变化不易察觉，但是有可能导致当前项目出错。例如由于接口的改变，当前项目中的相关代码无法编译。这种隐藏的、潜在的威胁一旦出现，就往往需要耗费大量的时间来查明真相。因此，显式声明任何项目中直接用到的依赖。

结果中还有一个重要的部分是 Unused declared dependencies，意指项目中未使用的，但显式声明的依赖，这里有 spring-core 和 spring-beans。需要注意的是，对于这样一类依赖，我们不应该简单地直接删除其声明，而是应该仔细分析。由于 dependency:analyze 只会分析编译主代码和测试代码需要用到的依赖，一些执行测试和运行时需要的依赖它就发现不了。很显然，该例中的 spring-core 和 spring-beans 是运行 Spring Framework 项目必要的类库，因此

不应该删除依赖声明。当然，有时候确实能通过该信息找到一些没用的依赖，但一定要小心测试。

5.10 小结

本章主要介绍了 Maven 的两个核心概念：坐标和依赖。解释了坐标的由来，并详细阐述了各坐标元素的作用及定义方式。随后引入 account-email 这一实际的基于 Spring Framework 的模块，包括了 POM 定义、主代码和测试代码。在这一直观感受的基础上，再花了大篇幅介绍 Maven 依赖，包括依赖范围、传递性依赖、可选依赖等概念。最后，当然少不了关于依赖的一些最佳实践。通过阅读本章，读者应该已经能够透彻地了解 Maven 的依赖管理机制。下一章将会介绍 Maven 的另一个核心概念：仓库。