Maven3 实战笔记 02 坐标和依赖

刘岩

Emai:suhuanzheng7784877@163.com

1. 项目需求

   加入我们现在有这样一个小小项目，就是做一个注册模块，让注册的人员记录可以插入到数据库中，还可以做账号的唯一性判断，注册成功后可以进行邮件提醒功能。书上功能很简单，其实重点不是功能，而是借由此示例说明 Maven 的特性。这一节咱们主要说明一下坐标与依赖的特性。其余的特性也皆由此案例中衍生出来。

2. 模块划分

   基本模块功能分为

   验证码生成：包括生成随即验证数字以及数字图片。

   发送邮件：注册成功后需要给注册的 Email 发一封 email，以便激活注册信息。

   激活账户：用于邮件激活用户之后，用户才可用。

   登录：一切成功后可以登录系统。

   系统比较简单了，但是我们也分模块进行开发。也好体现 Maven 模块强解耦合的管理思想。

3. 邮件模块的实现

   这里和书中的讲解顺序不太一样，我们先来开发功能模块，之后通过 Maven 的坐标和以来的概念来分析咱们的邮件模块。整个系统是 B/S 架构，采用 Spring 帮助我们进行类的管理。为了放大这个微型系统的规模，我们单独为这个邮件功能建立一个项目，模拟一个很大系统的一个模块。我们先看邮件模块的实现类吧，接口就不给出了

```
package com.liuyan.account.mail.impl;
```

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

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

import com.liuyan.account.mail.AccountEmailService;

public class AccountEmailServiceImpl implements AccountEmailService
{

    private JavaMailSender javaMailSender;

    public JavaMailSender getJavaMailSender() {
        return javaMailSender;
    }

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

    private String systemEmail;

    public String getSystemEmail() {
        return systemEmail;
    }

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

    @Override
    public void sendMail(String to, String subject, String message)
            throws MessagingException {

        MimeMessage msg = javaMailSender.createMimeMessage();
        MimeMessageHelper msgHelper = new MimeMessageHelper(msg);

        msgHelper.setFrom(systemEmail);
        msgHelper.setTo(to);
        msgHelper.setSubject(subject);
        msgHelper.setText(message);
        javaMailSender.send(msg);
```

```
    }

}
```

之后在包/src/test/java 编写测试用例

```
package com.liuyan.account.mail;

import javax.mail.MessagingException;

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.ServerSetup;

public class AccountEmailServiceTest {

    private GreenMail greenMail;

    @Before
    public void startTest() {
        greenMail = new GreenMail(ServerSetup.SMTP);
        greenMail.setUser("suhuanzheng7784877@163.com", "1111");
        greenMail.start();
    }

    @Test
    public void sendMail() throws MessagingException {

        ApplicationContext ctx = new ClassPathXmlApplicationContext(
                "applicationContext.xml");

        AccountEmailService accountEmailService =
(AccountEmailService) ctx
                .getBean("accountEmailService");

        String to = "suhuanzheng7784877@163.com";
```

```
        String subject = "测试";


        String message = "内容";

        accountEmailService.sendMail(to, subject, message);

    }

    @After
    public void stop() {
        greenMail.stop();
    }

}
```

单元测试也写完了，之后就是项目构建、打包了。

在这之前我们来看看 Spring 配置文件的内容

```
<?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
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"
    xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:p="http://www.springframework.org/schema/p">


    <!-- 加载Properties文件 -->

    <bean id="configurer"

    class="org.springframework.beans.factory.config.PropertyPlaceho
lderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:mail.properties</value>
            </list>
        </property>
    </bean>
```

```
    <bean id="javaMailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <property name="defaultEncoding" value="UTF-8" />
        <property name="host" value="${mail.host}" />
        <property name="username" value="${mail.username}" />
        <property name="password" value="${mail.password}" />
        <property name="javaMailProperties">
            <props>

                <!-- 设置认证开关 -->

                <prop key="mail.smtp.auth">true</prop>

                <!-- 启动调试开关 -->

                <prop key="mail.debug">true</prop>
            </props>
        </property>
    </bean>

    <bean id="accountEmailService"
class="com.liuyan.account.mail.impl.AccountEmailServiceImpl">
        <property name="javaMailSender"
ref="javaMailSender"></property>
        <property name="systemEmail"
value="suhuanzheng7784877@163.com"></property>
    </bean>
</beans>
```

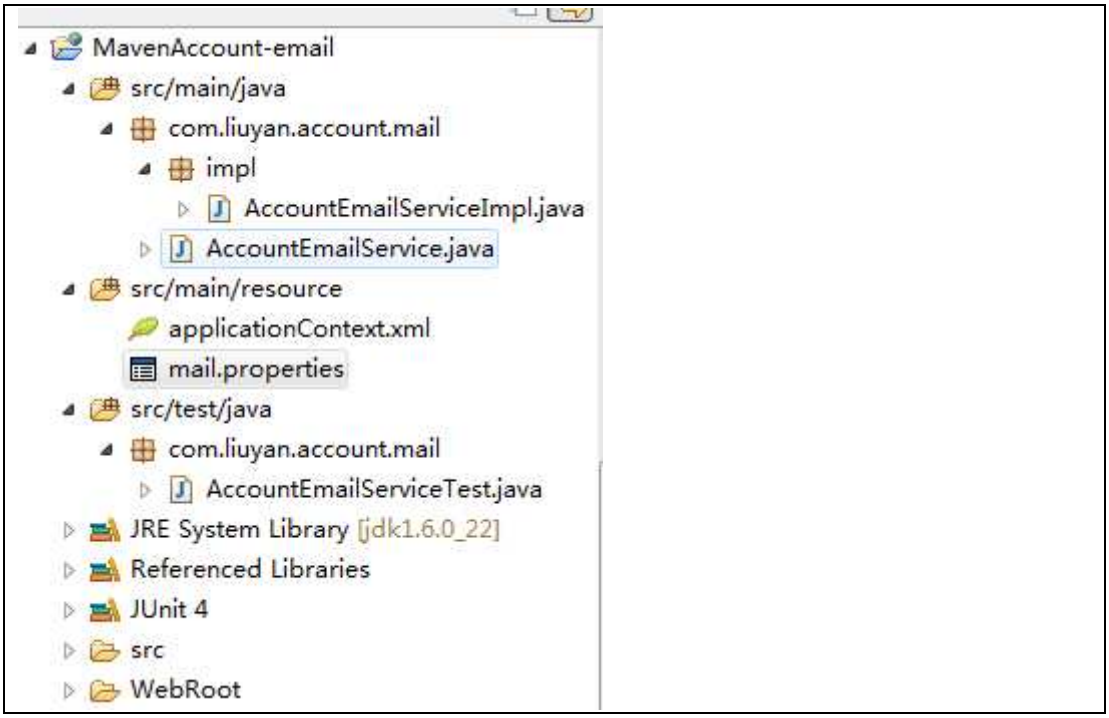
mail.properties 内容

```
mail.host=smtp.163.com
mail.username=suhuanzheng7784877@163.com
mail.password=111111
mail.from=suhuanzheng7784877@163.com
mail.to=suhuanzheng7784877@163.com
```

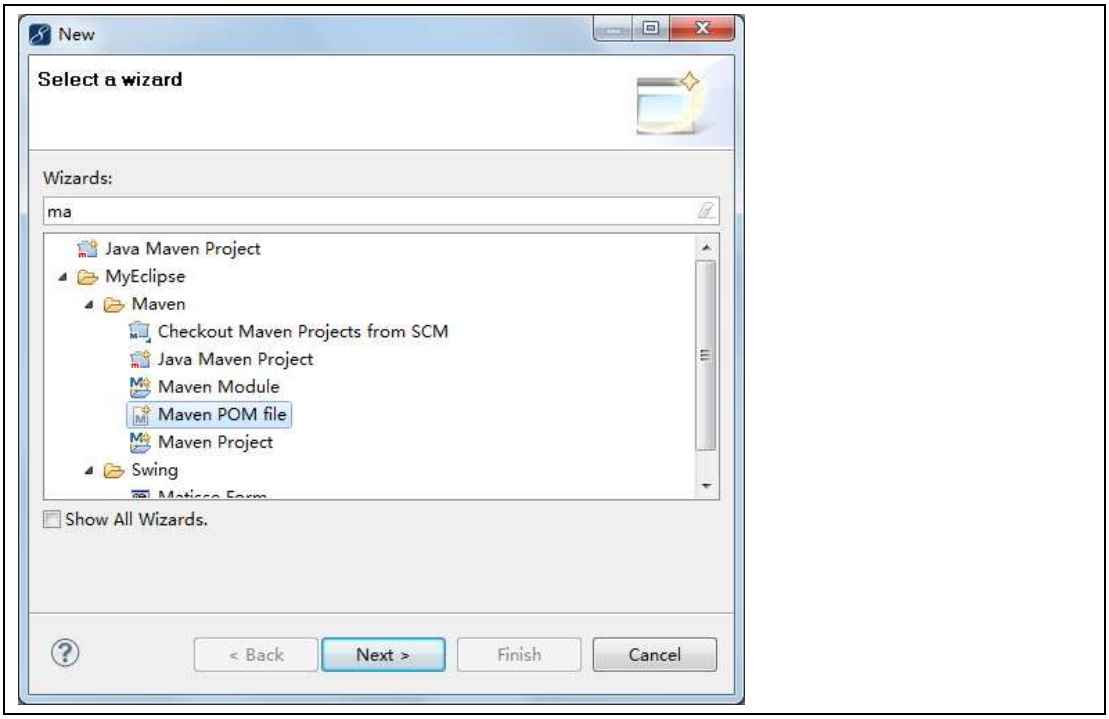
下面就需要 Maven 出场了。

4. Maven 的介入

其实上面的程序打包已经按照了 Maven 的规约了~我们再来看看这个项目模块的代码结构



源代码放到 src\main\java 下面，配置文件放到 src\main\resource 下面，单元测试代码在 src\test\java 下。现在我们来写项目描述文件 pom.xml。



通过 MyEclipse 插件很快写出 pom.xml 文件内容

```
<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.liuyan.account</groupId>
    <artifactId>MavenAccount-email</artifactId>
    <version>0.0.1-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>
        <resources>
            <resource>
                <directory>src/main/resource</directory>
            </resource>
        </resources>
    </build>
</project>
```

运行一下命令。

```
mvn clean test
```

自己的邮箱看看效果，多了一封邮件。Ok，再看本地仓库，唉~~~自己的 c 盘又添了不少东西。本地仓库会越来越大的。各位使用者要有心理准备，当然可以通过配置的方式转移本地仓库存储位置。

实际上是通过这个小例子说明坐标这个概念。在 Maven 中的坐标的意思就是各种构建引入的秩序，坐标是多维的。也就是 groupId、artifactId、version、packaging 这几个元素决定了其组建的唯一标识（classifier 没有看到，如果后面看到了再补充进来）。

前面也稍带提过了 groupId 是项目组的标识、artifactId 是模块标识、version 是版本号标识、packaging 是打包方式，默认是 jar 方式。如此看来这些元素描述了一个不能重复的标识，groupId+ artifactId+ version+”.”+packaging 在整个儿仓库中是不应该重复的。

```
    <groupId>com.liuyan.account</groupId>
    <artifactId>MavenAccount-email</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

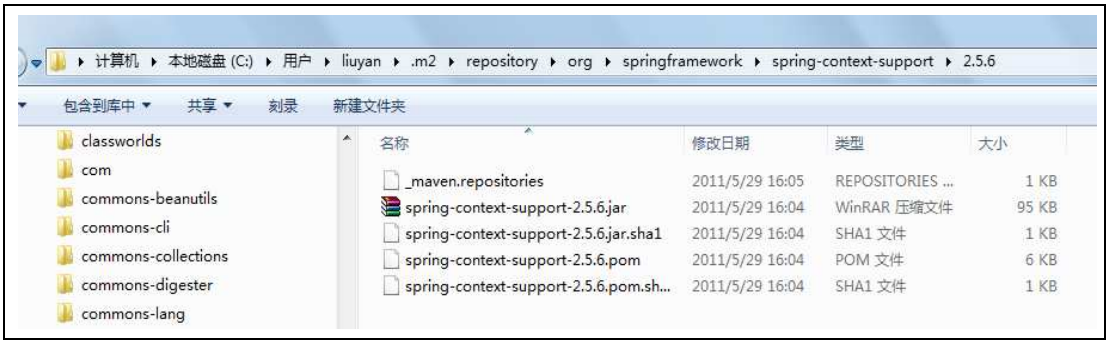
而其他项目需要你的模块为其服务的时候，也是根据坐标找到你的模块的。这就引出了下面依赖项的配置。这个模块依赖了 Spring，不怕，我们从 maven 仓库中去取依赖就可以了，下面的问题就是：我们需要哪些依赖？这些依赖的版本是什么？

通过已有组件的坐标，我们可以轻易从 Maven 中心库中获取依赖包

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
```

```
            <version>2.5.6</version>
        </dependency>
```

描述我们这个邮件模块依赖 org.springframework.spring-context-support.2.5.6.jar 这个东东。果然，运行后在本地仓库中找到了



项目依赖可以有以下几种配置范围

1：compile：默认值，这个代表在 Maven 项目周期的编译、测试、运行的 3 个 classpath 都生效，也就是说 Maven 自己在项目不同生命周期使用的 classpath 不一样，一旦配置了 compile，那么在以上三个周期会将依赖 jar 放到不同周期 classpath 中。

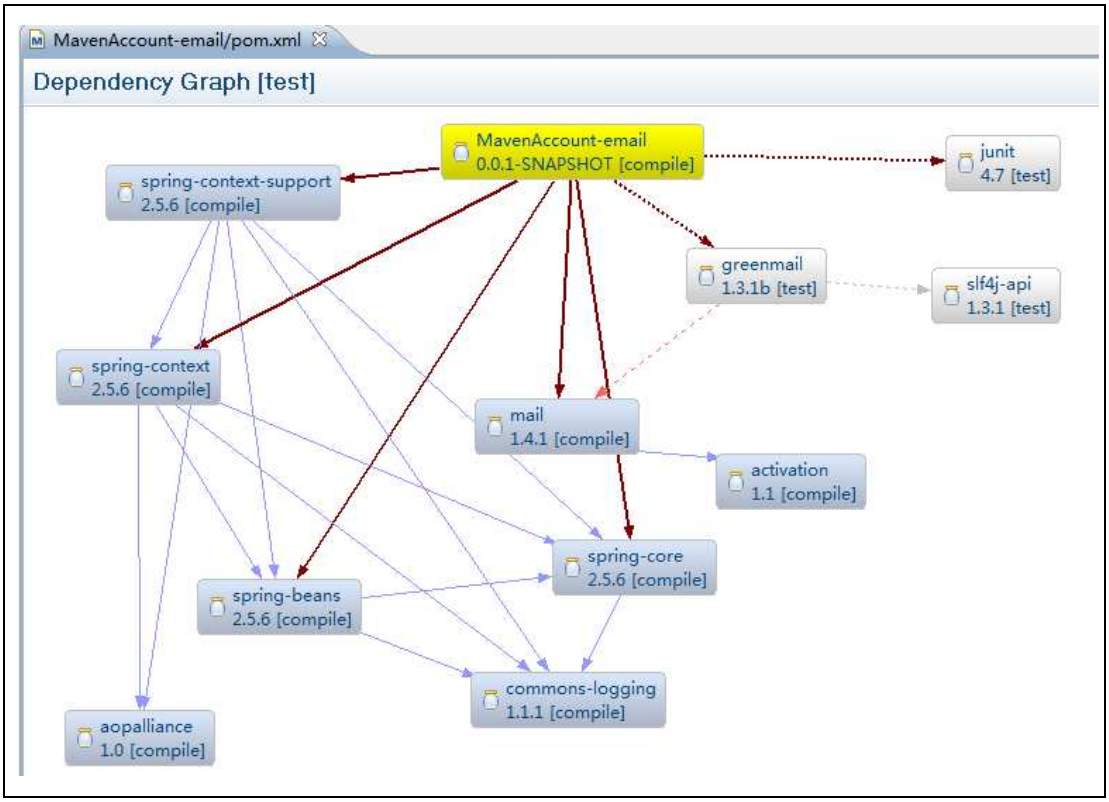
2：test：测试期间有效，运行期间不必引入，就像 junit 包，在运行期间就没必要引入了。

3：provided：在编译、测试期间有效，其余周期都没用到，比如 servlet-api.jar，web 容器提供了此规范 apijar 和实现。

4：runtime：只在运行、测试期间生效，以 JDK 的 JDBC 规范为代表，哦，还有 JavaEE 的 JPA 规范。

5：system：与本机系统环境绑定，在编译、测试期间有效，往往和本机环境绑定，移植性不太好。

依赖传播：单丝不成线，孤木不成林。一个项目需要依赖很多辅助包，我们的 spring 也不例外。下面是本项目的依赖关系图



咱们就拿 spring-core 来说事儿吧，本项目直接依赖于它，它呢也不是吃素的，也需要别人的支持，他需要 apache 的 commons-logging。那么就导致了如果我们想要使用 spring-core 的功能，必须得引入这个 commons-logging 包喽。没有 Maven 之前，我们一般都是先引入我们直接需要的包，之后根据错误信息加上我们自身的开发经验一个一个去网上下载我们核心包需要的其他包。造成的结果是，花了大量的时间在网上找包，好不容易将项目 run 起来了，执行一个操作后，又发现在运行的时候还需要别的 jar 包，如此根据错误信息往复在网上找相关 jar 包。最后发现自己一个很简单的项目怎么引入了那么多的包啊！

使用 Maven 管理项目的话，它会自动为您找这种传递性的依赖，因为在 Maven 中心仓库的项目描述中都已经严格阐述了各个开源项目的依赖关系。诚然，这种传递性依赖也存在范围的问题。我们就拿上面的举例子，本发送邮件模块项目依赖于 spring-core 依赖范围是 compile，而 spring-core 依赖于 commons-logging 也是 compile 范围。那么通过传递性，本项目对 commons-logging 的依赖也是 compile。看下表

直接依赖范围	compile	test	provided	runtime
compile	compile	-	-	runtime
test	test	-	-	test
provided	provide	-	provided	provided
runtime	runtime	-	-	runtime

这个表是嘛意思呢？就是说一个项目的第一直接依赖假如是 compile 范围，那么它的第二依赖包如果也是 compile，那么本项目和第二依赖包的依赖范围就是 compile。如果一个项目的第一直接依赖假如是 compile 范围，那么它的第二依赖包如果是 test，那么本项目和第二依赖包不存在任何依赖关系，也是，第一直接依赖包和第二依赖包是测试的时候才用到的。无需影响到其他使用者。

5. 依赖调节

假如现在有这么一个情况，项目 A->(依赖)项目 B->项目 C->X(1.0)，项目 A->项目 D->X(1.5)。那么项目 A 就不得不依赖于项目 X。那我们这个项目 A 到底是下载 X 项目的哪个版本呢？Maven3 对于这种情况有 2 个原则，**第一个就是路径优先原则，第二个就是在配置文件 pom.xml 中谁先配置在前面谁解析使用。**

项目路径就是指依赖的层级比如 A->(依赖)项目 B->项目 C->X(1.0)就是 X 是 A 的第三层依赖，A->项目 D->X(1.5)则是 X 是 A 的第二层依赖。那么根据原则 1，X1.5 版本会被优先选择使用。

6. 可选依赖

当一个项目出现了依赖可选——optional 为 true 的时候说明，只有当前这个项目依赖于此可选依赖，而别的项目需要此项目的时候，此项目的可选依赖并不会像其他依赖类型似地，可选依赖不会传递给别的项目。在别的项目需要相关的可选依赖的时候还需要在

pom.xml 文件中显示的进行声明。其实可选依赖并不倡导，可选依赖就意味着此模块的职能比较复杂，不单一。一般替使用者完成了不该完成的功能。有点违背了 Java 设计模式的职能单一原则。

7. 依赖最佳化实现

1)：归类依赖：就拿咱们这个邮件模块来说，用到了 Spring2.5.6，其中用到了 Spring 项目的不同模块，现在开源的项目越来越讲究模块化，模块职能单一化。所以看到往往一个开源项目有很多 jar，用到那个模块引入哪个 jar，Spring、Hibernate 都是这么做的。那么假如我们现在的项目要升级版本，Spring2.5.6 已经不能满足了，要升级到 3.0 版本，怎么办？一个个去改以来的版本号？恩，也是个办法，不过有点笨拙。有另一个办法，不错，就跟 Spring 配置文件引入的那个资源文件原理一样，声明一个常量信息，所有用到的地方都用这个常量信息就够了，如下是我们修改后的配置

```
<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.liuyan.account</groupId>
    <artifactId>MavenAccount-email</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <springversion>2.5.6</springversion>
        <junitversion>2.5.6</junitversion>
    </properties>

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

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${springversion}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${springversion}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${springversion}</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>
        <resources>
            <resource>
                <directory>src/main/resource</directory>
            </resource>
        </resources>
    </build>
</project>
```

2) : 排除依赖 : 假如现在有这么一个场景项目 A->项目 B-项目 C,而 A 与 C 就形成了间接依赖 , 而 C 呢却又老是不稳定。那么构建项目 A 的时候如果下载了项目 C 的非稳定版本

是很不安全的。那么不妨这么做，在项目 A 的依赖中肯定是要配置项目 B 的，在配置项目 B 的同时 ,强制让项目 B 排斥项目 C ,让 A 再加个依赖就是 A 用着较为稳定的版本 C。

配置如下

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${springversion}</version>
            <exclusions>
                <exclusion>
                    <groupId>org.apache</groupId>
                    <artifactId>logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache</groupId>
            <artifactId>logging</artifactId>
            <version>1.1.0</version>
        </dependency>
```

使用 IDE 工具可以显示依赖图，当然可以通过命令

```
//显示依赖项

mvn dependency:list



//显示依赖树

mvn dependency:tree



//依赖项分析-找出那些依赖项没用

mvn dependency:analyze
```

8. 总结

我们这次开发了一个模块介绍了一下 Maven 的构建特性和 jar 包依赖特性。之后结合 IDE

阐述了 Maven 的 pom.xml 文件的一些内容——坐标。之后介绍了一下项目依赖的概念。