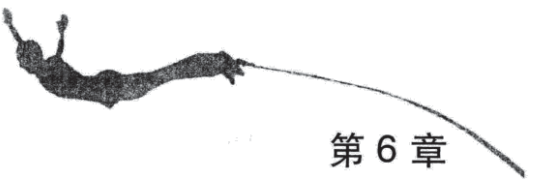

第6章

仓　库

本章内容

- ❑ 何为 Maven 仓库
- ❑ 仓库的布局
- ❑ 仓库的分类
- ❑ 远程仓库的配置
- ❑ 快照版本
- ❑ 从仓库解析依赖的机制
- ❑ 镜像
- ❑ 仓库搜索服务
- ❑ 小结

第 5 章详细介绍了 Maven 坐标和依赖，坐标和依赖是任何一个构件在 Maven 世界中的逻辑表示方式；而构件的物理表示方式是文件，Maven 通过仓库来统一管理这些文件。本章将详细介绍 Maven 仓库，在了解了 Maven 如何使用仓库之后，将能够更高效地使用 Maven。

6.1 何为 Maven 仓库

在 Maven 世界中，任何一个依赖、插件或者项目构建的输出，都可以称为构件。例如，依赖 log4j-1.2.15.jar 是一个构件，插件 maven-compiler-plugin-2.0.2.jar 是一个构件，第 5 章的 account-email 项目构建完成后的输出 account-email-1.0.0-SNAPSHOT.jar 也是一个构件。任何一个构件都有一组坐标唯一标识。

在一台工作站上，可能会有几十个 Maven 项目，所有项目都使用 maven-compiler-plugin，这些项目中的大部分都用到了 log4j，有一小部分用到了 Spring Framework，还有另外一小部分用到了 Struts2。在每个有需要的项目中都放置一份重复的 log4j 或者 struts2 显然不是最好的解决方案，这样做不仅造成了磁盘空间的浪费，而且也难于统一管理，文件的复制等操作也会降低构建的速度。而实际情况是，在不使用 Maven 的那些项目中，我们往往就能发现命名为 lib/的目录，各个项目 lib/目录下的内容存在大量的重复。

得益于坐标机制，任何 Maven 项目使用任何一个构件的方式都是完全相同的。在此基础上，Maven 可以在某个位置统一存储所有 Maven 项目共享的构件，这个统一的位置就是仓库。实际的 Maven 项目将不再各自存储其依赖文件，它们只需要声明这些依赖的坐标，在需要的时候（例如，编译项目的时候需要将依赖加入到 classpath 中），Maven 会自动根据坐标找到仓库中的构件，并使用它们。

为了实现重用，项目构建完毕后生成的构件也可以安装或者部署到仓库中，供其他项目使用。

6.2 仓库的布局

任何一个构件都有其唯一的坐标，根据这个坐标可以定义其在仓库中的唯一存储路径，这便是 Maven 的仓库布局方式。例如，log4j:log4j:1.2.15 这一依赖，其对应的仓库路径为 log4j/log4j/1.2.15/log4j-1.2.15.jar，细心的读者可以观察到，该路径与坐标的大致对应关系为 groupId/artifactId/version/artifactId-version.packaging。下面看一段 Maven 的源码，并结合具体的实例来理解 Maven 仓库的布局方式，见代码清单 6-1：

代码清单 6-1 Maven 处理仓库布局的源码

```
private static final char PATH_SEPARATOR = '/';

private static final char GROUP_SEPARATOR = '.';

private static final char ARTIFACT_SEPARATOR = '-';
```

```
public String pathOf( Artifact artifact )
{
    ArtifactHandler artifactHandler = artifact.getArtifactHandler();

    StringBuilder path = new StringBuilder( 128 );

    path.append( formatAsDirectory( artifact.getGroupId() ) ).append( PATH_SEPA-
RATOR );
    path.append( artifact.getArtifactId() ).append( PATH_SEPARATOR );
    path.append( artifact.getBaseVersion() ).append( PATH_SEPARATOR );
    path.append( artifact.getArtifactId() ).append( ARTIFACT_SEPARATOR ).append
( artifact.getVersion() );

    if ( artifact.hasClassifier() )
    {
        path.append( ARTIFACT_SEPARATOR ).append( artifact.getClassifier() );
    }

    if ( artifactHandler.getExtension() != null && artifactHandler.getExtension()
.length() > 0 )
    {
        path.append( GROUP_SEPARATOR ).append( artifactHandler.getExtension() );
    }

    return path.toString();
}

private String formatAsDirectory( String directory )
{
    return directory.replace( GROUP_SEPARATOR, PATH_SEPARATOR );
}
```

该 pathOf( ) 方法的目的是根据构件信息生成其在仓库中的路径。在阅读本段代码之前，可以先回顾一下第 5 章的相关内容。这里根据一个实际的例子来分析路径的生成，考虑这样一个构件：groupId = org. testng、artifactId = testng、version = 5. 8、classifier = jdk15、packaging = jar，其对应的路径按如下步骤生成：

1）基于构件的 groupId 准备路径，formatAsDirectory( ) 将 groupId 中的句点分隔符转换成路径分隔符。该例中，groupId org. testng 就会被转换成 org/testng，之后再加一个路径分隔符斜杠，那么，org. testng 就成为了 org/testng/。

2）基于构件的 artifactId 准备路径，也就是在前面的基础上加上 artifactId 以及一个路径分隔符。该例中的 artifactId 为 testng，那么，在这一步过后，路径就成为了 org/testng/testng/。

3）使用版本信息。在前面的基础上加上 version 和路径分隔符。该例中版本是 5. 8，那么路径就成为了 org/testng/tesgng/5. 8/。

4）依次加上 artifactId，构件分隔符连字号，以及 version，于是构建的路径就变成了 org/testng/testng/5. 8/testng-5. 8。读者可能会注意到，这里使用了 artifactId. getVersion( )，而上一步用的是 artifactId. getBaseVersion( )，baseVersion 主要是为 SNAPSHOT 版本服务的，例如 version 为 1. 0-SNAPSHOT 的构件，其 baseVersion 就是 1. 0。

5）如果构件有 classifier，就加上构件分隔符和 classifier。该例中构件的 classifier 是 jdk15，那么路径就变成 org/testng/testng/5.8/testng-5.8-jdk5。

6）检查构件的 extension，若 extension 存在，则加上句点分隔符和 extension。从代码中可以看到，extension 是从 artifactHandler 而非 artifact 获取，artifactHandler 是由项目的 packaging 决定的。因此，可以说，packaging 决定了构件的扩展名，该例的 packaging 是 jar，因此最终的路径为 org/testng/testng/5.8/testng-5.8-jdk5.jar。

到这里，应该感谢 Maven 开源社区，正是由于 Maven 的所有源代码都是开放的，我们才能仔细地深入到其内部工作的所有细节。

Maven 仓库是基于简单文件系统存储的，我们也理解了其存储方式，因此，当遇到一些与仓库相关的问题时，可以很方便地查找相关文件，方便定位问题。例如，当 Maven 无法获得项目声明的依赖时，可以查看该依赖对应的文件在仓库中是否存在，如果不存在，查看是否有其他版本可用，等等。

6.3 仓库的分类

对于 Maven 来说，仓库只分为两类：本地仓库和远程仓库。当 Maven 根据坐标寻找构件的时候，它首先会查看本地仓库，如果本地仓库存在此构件，则直接使用；如果本地仓库不存在此构件，或者需要查看是否有更新的构件版本，Maven 就会去远程仓库查找，发现需要的构件之后，下载到本地仓库再使用。如果本地仓库和远程仓库都没有需要的构件，Maven 就会报错。

在这个最基本分类的基础上，还有必要介绍一些特殊的远程仓库。中央仓库是 Maven 核心自带的远程仓库，它包含了绝大部分开源的构件。在默认配置下，当本地仓库没有 Maven 需要的构件的时候，它就会尝试从中央仓库下载。

私服是另一种特殊的远程仓库，为了节省带宽和时间，应该在局域网内架设一个私有的仓库服务器，用其代理所有外部的远程仓库。内部的项目还能部署到私服上供其他项目使用。

除了中央仓库和私服，还有很多其他公开的远程仓库，常见的有 Java.net Maven 库（http://download.java.net/maven/2/）和 JBoss Maven 库（http://repository.jboss.com/maven2/）等。

Maven 仓库的分类见图 6-1。

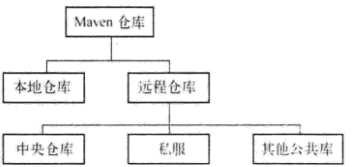

图 6-1 Maven 仓库的分类

6.3.1 本地仓库

一般来说，在 Maven 项目目录下，没有诸如 lib/这样用来存放依赖文件的目录。当 Maven 在执行编译或测试时，如果需要使用依赖文件，它总是基于坐标使用本地仓库的依赖文件。

默认情况下，不管是在 Windows 还是 Linux 上，每个用户在自己的用户目录下都有一个路径名为 .m2/repository/的仓库目录。例如，笔者的用户名是 juven，我在 Windows 机器上的本地仓库地址为 C:\Users\juven\.m2\repository\，而我在 Linux 上的本地仓库地址为/home/juven/.m2/repository/。注意，在 Linux 系统中，以点(.)开头的文件或目录默认是隐藏的，可以使用 **ls-a** 命令显示隐藏文件或目录。

有时候，因为某些原因（例如 C 盘空间不够），用户会想要自定义本地仓库目录地址。这时，可以编辑文件 ~/.m2/settings.xml，设置 localRepository 元素的值为想要的仓库地址。例如：

```
<settings>
    <localRepository>D:\java\repository\</localRepository>
</settings>
```

这样，该用户的本地仓库地址就被设置成了 D:\java\repository\。

需要注意的是，默认情况下，~/.m2/settings.xml 文件是不存在的，用户需要从 Maven 安装目录复制 $M2_HOME/conf/settings.xml 文件再进行编辑。本书始终推荐大家不要直接修改全局目录的 settings.xml 文件，具体原因已在第 2.7.2 节中阐述。

一个构件只有在本地仓库中之后，才能由其他 Maven 项目使用，那么构件如何进入到本地仓库中呢？最常见的是依赖 Maven 从远程仓库下载到本地仓库中。还有一种常见的情况是，将本地项目的构件安装到 Maven 仓库中。例如，本地有两个项目 A 和 B，两者都无法从远程仓库获得，而同时 A 又依赖于 B，为了能构建 A，B 就必须首先得以构建并安装到本地仓库中。

在某个项目中执行 **mvn clean install** 命令，就能看到如下输出：

```
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: D:\git-juven\maven-book\code\ch-5\account-email\target
\account-email-1.0.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing D:\git-juven\maven-book\code\ch-5\account-email\target\ac-
count-email-1.0.0-SNAPSHOT.jar to D:\java\repository\com\juven\mvnbook\account\
account-email\1.0.0-SNAPSHOT\account-email-1.0.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
```

Install 插件的 install 目标将项目的构建输出文件安装到本地仓库。在上述输出中，构建输出文件是 account-email-1.0.0-SNAPSHOT.jar，本地仓库地址是 D:\java\repository，Maven 使用 Install 插件将该文件复制到本地仓库中，具体的路径根据坐标计算获得。计算逻辑请参考 6.2 节。

6.3.2 远程仓库

安装好 Maven 后，如果不执行任何 Maven 命令，本地仓库目录是不存在的。当用户输入第一条 Maven 命令之后，Maven 才会创建本地仓库，然后根据配置和需要，从远程仓库下载构件至本地仓库。

这好比藏书。例如，我想要读《红楼梦》，会先检查自己的书房是否已经收藏了这本书，如果发现没有这本书，于是就跑去书店买一本回来，放到书房里。可能有一天我又想读一本英文版的《程序员修炼之道》，而书房里只有中文版，于是又去书店找，可发现书店没有，好在还有网上书店，于是从 Amazon 买了一本，几天后我收到了这本书，又放到了自己的书房。

本地仓库就好比书房，我需要读书的时候先从书房找，相应地，Maven 需要构件的时候先从本地仓库找。远程仓库就好比书店（包括实体书店、网上书店等），当我无法从自己的书房找到需要的书的时候，就会从书店购买后放到书房里。当 Maven 无法从本地仓库找到需要的构件的时候，就会从远程仓库下载构件至本地仓库。一般地，对于每个人来说，书房只有一个，但外面的书店有很多，类似地，对于 Maven 来说，每个用户只有一个本地仓库，但可以配置访问很多远程仓库。

6.3.3 中央仓库

由于最原始的本地仓库是空的，Maven 必须知道至少一个可用的远程仓库，才能在执行 Maven 命令的时候下载到需要的构件。中央仓库就是这样一个默认的远程仓库，Maven 的安装文件自带了中央仓库的配置。读者可以使用解压工具打开 jar 文件 $M2_HOME/lib/maven-model-builder-3.0.jar（在 Maven 2 中，jar 文件路径类似于 $M2_HOME/lib/maven-2.2.1-uber.jar），然后访问路径 org/apache/maven/model/pom-4.0.0.xml，可以看到如下的配置：

```
<repositories>
  <repository>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <url>http://repo1.maven.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

包含这段配置的文件是所有 Maven 项目都会继承的超级 POM，第 8 章会详细介绍继承及超级 POM。这段配置使用 id central 对中央仓库进行唯一标识，其名称为 Maven Repository Switchboard，它使用 default 仓库布局，也就是在第 6.2 节介绍的仓库布局。对于 Maven 1 的仓库，需要配置值为 legacy 的 layout，本书不会涉及 Maven 1。最后需要注意的是 snapshots 元素，其子元素 enabled 的值为 false，表示不从该中央仓库下载快照版本的构件（本章稍后

详细介绍快照版本)。

中央仓库包含了这个世界上绝大多数流行的开源 Java 构件，以及源码、作者信息、SCM、信息、许可证信息等，每个月这里都会接受全世界 Java 程序员大概 1 亿次的访问，它对全世界 Java 开发者的贡献由此可见一斑。由于中央仓库包含了超过 2000 个开源项目的构件，因此，一般来说，一个简单 Maven 项目所需要的依赖构件都能从中央仓库下载到。这也解释了为什么 Maven 能做到“开箱即用”。

6.3.4 私服

私服是一种特殊的远程仓库，它是架设在局域网内的仓库服务，私服代理广域网上的远程仓库，供局域网内的 Maven 用户使用。当 Maven 需要下载构件的时候，它从私服请求，如果私服上不存在该构件，则从外部的远程仓库下载，缓存在私服上之后，再为 Maven 的下载请求提供服务。此外，一些无法从外部仓库下载到的构件也能从本地上传到私服上供大家使用，如图 6-2 所示。

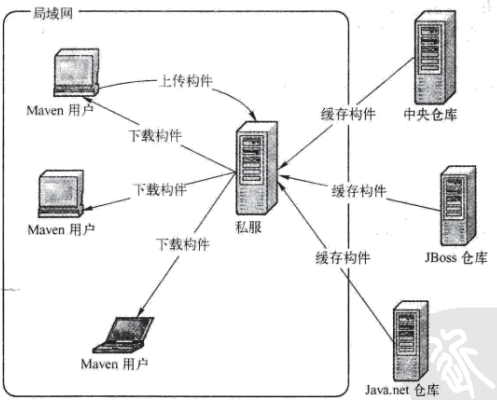

图 6-2 私服的用途

图 6-2 展示的是组织内部使用私服的情况。即使在一台直接连入 Internet 的个人机器上使用 Maven，也应该在本地建立私服。因为私服可以帮助你：

- **节省自己的外网带宽**。建立私服同样可以减少组织自己的开支，大量的对于外部仓库的重复请求会消耗很大的带宽，利用私服代理外部仓库之后，对外的重复构件下载便得以消除，即降低外网带宽的压力。
- **加速 Maven 构建**。不停地连接请求外部仓库是十分耗时的，但是 Maven 的一些内部

机制（如快照更新检查）要求 Maven 在执行构建的时候不停地检查远程仓库数据。因此，当项目配置了很多外部远程仓库的时候，构建的速度会被大大降低。使用私服可以很好地解决这一问题，当 Maven 只需要检查局域网内私服的数据时，构建的速度便能得到很大程度的提高。

- **部署第三方构件。**当某个构件无法从任何一个外部远程仓库获得，怎么办？这样的例子有很多，如组织内部生成的私有构件肯定无法从外部仓库获得、Oracle 的 JDBC 驱动由于版权因素不能发布到公共仓库中。建立私服之后，便可以将这些构件部署到这个内部的仓库中，供内部的 Maven 项目使用。
- **提高稳定性，增强控制。**Maven 构建高度依赖于远程仓库，因此，当 Internet 不稳定的时候，Maven 构建也会变得不稳定，甚至无法构建。使用私服后，即使暂时没有 Internet 连接，由于私服中已经缓存了大量构件，Maven 也仍然可以正常运行。此外，一些私服软件（如 Nexus）还提供了很多额外的功能，如权限管理、RELEASE/SNAPSHOT 区分等，管理员可以对仓库进行一些更高级的控制。
- **降低中央仓库的负荷。**运行并维护一个中央仓库不是一件容易的事情，服务数百万的请求，存储数 T 的数据，需要相当大的财力。使用私服可以避免很多对中央仓库重复的下载，想象一下，一个有数百位开发人员的公司，在不使用私服的情况下，一个构件往往会被重复下载数百次；建立私服之后，这几百次下载就只会发生在内网范围内，私服对于中央仓库只有一次下载。

建立私服是用好 Maven 十分关键的一步，第 9 章会专门介绍如何使用最流行的 Maven 私服软件——Nexus。

6.4 远程仓库的配置

在很多情况下，默认的中央仓库无法满足项目的需求，可能项目需要的构件存在于另外一个远程仓库中，如 JBoss Maven 仓库。这时，可以在 POM 中配置该仓库，见代码清单 6-2。

代码清单 6-2 配置 POM 使用 JBoss Maven 仓库

```
<project>
  ...
  <repositories>
    <repository>
      <id>jboss</id>
      <name>JBoss Repository</name>
      <url>http://repository.jboss.com/maven2/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
```

```
        <layout>default</layout>
    </repository>
  </repositories>
  ...
</project>
```

在 repositories 元素下，可以使用 repository 子元素声明一个或者多个远程仓库。该例中声明了一个 id 为 jboss，名称为 JBoss Repository 的仓库。任何一个仓库声明的 id 必须是唯一的，尤其需要注意的是，Maven 自带的中央仓库使用的 id 为 central，如果其他的仓库声明也使用该 id，就会覆盖中央仓库的配置。该配置中的 url 值指向了仓库的地址，一般来说，该地址都基于 http 协议，Maven 用户都可以在浏览器中打开仓库地址浏览构件。

该例配置中的 releases 和 snapshots 元素比较重要，它们用来控制 Maven 对于发布版构件和快照版构件的下载。关于快照版本，在第 6.5 节中会详细解释。这里需要注意的是 enabled 子元素，该例中 releases 的 enabled 值为 true，表示开启 JBoss 仓库的发布版本下载支持，而 snapshots 的 enabled 值为 false，表示关闭 JBoss 仓库的快照版本的下载支持。因此，根据该配置，Maven 只会从 JBoss 仓库下载发布版的构件，而不会下载快照版的构件。

该例中的 layout 元素值 default 表示仓库的布局是 Maven 2 及 Maven 3 的默认布局，而不是 Maven 1 的布局。

对于 releases 和 snapshots 来说，除了 enabled，它们还包含另外两个子元素 updatePolicy 和 checksumPolicy：

```
<snapshots>
    <enabled>true</enabled>
    <updatePolicy>daily</updatePolicy>
    <checksumPolicy>ignore</checksumPolicy>
</snapshots>
```

元素 updatePolicy 用来配置 Maven 从远程仓库检查更新的频率，默认的值是 daily，表示 Maven 每天检查一次。其他可用的值包括：never—从不检查更新；always—每次构建都检查更新；interval：X—每隔 X 分钟检查一次更新（X 为任意整数）。

元素 checksumPolicy 用来配置 Maven 检查检验和文件的策略。当构件被部署到 Maven 仓库中时，会同时部署对应的校验和文件。在下载构件的时候，Maven 会验证校验和文件，如果校验和验证失败，怎么办？当 checksumPolicy 的值为默认的 warn 时，Maven 会在执行构建时输出警告信息，其他可用的值包括：fail—Maven 遇到校验和错误就让构建失败；ignore—使 Maven 完全忽略校验和错误。

6.4.1 远程仓库的认证

大部分远程仓库无须认证就可以访问，但有时候出于安全方面的考虑，我们需要提供认证信息才能访问一些远程仓库。例如，组织内部有一个 Maven 仓库服务器，该服务器为每个项目都提供独立的 Maven 仓库，为了防止非法的仓库访问，管理员为每个仓库提供了一组用户名及密码。这时，为了能让 Maven 访问仓库内容，就需要配置认证信息。

配置认证信息和配置仓库信息不同，仓库信息可以直接配置在 POM 文件中，但是认证信息必须配置在 settings. xml 文件中。这是因为 POM 往往是被提交到代码仓库中供所有成员访问的，而 settings. xml 一般只放在本机。因此，在 settings. xml 中配置认证信息更为安全。

假设需要为一个 id 为 my-proj 的仓库配置认证信息，编辑 settings. xml 文件见代码清单 6-3：

代码清单 6-3　在 settings. xml 中配置仓库认证信息

```
<settings>
  ...
  <servers>
    <server>
      <id>my-proj</id>
      <username>repo-user</username>
      <password>repo-pwd</password>
    </server>
  </servers>
  ...
</settings>
```

Maven 使用 settings. xml 文件中并不显而易见的 servers 元素及其 server 子元素配置仓库认证信息。代码清单 6-3 中该仓库的认证用户名为 repo-user，认证密码为 repo-pwd。这里的关键是 id 元素，settings. xml 中 server 元素的 id 必须与 POM 中需要认证的 repository 元素的 id 完全一致。换句话说，正是这个 id 将认证信息与仓库配置联系在了一起。

6.4.2　部署至远程仓库

在第 6.3.4 节中提到，私服的一大作用是部署第三方构件，包括组织内部生成的构件以及一些无法从外部仓库直接获取的构件。无论是日常开发中生成的构件，还是正式版本发布的构件，都需要部署到仓库中，供其他团队成员使用。

Maven 除了能对项目进行编译、测试、打包之外，还能将项目生成的构建部署到仓库中。首先，需要编辑项目的 pom. xml 文件。配置 distributionManagement 元素见代码清单 6-4。

代码清单 6-4　在 POM 中配置构件部署地址

```
<project>
  ...
  <distributionManagement>
    <repository>
      <id>proj-releases</id>
      <name>Proj Release Repository</name>
      <url>http://192.168.1.100/content/repositories/proj-releases</url>
    </repository>
    <snapshotRepository>
      <id>proj-snapshots</id>
      <name>Proj Snapshot Repository</name>
      <url>http://192.168.1.100/content/repositories/proj-snapshots</url>
    </snapshotRepository>
```

```
    </distributionManagement>
    ...
</project>
```

distributionManagement 包含 repository 和 snapshotRepository 子元素，前者表示发布版本构件的仓库，后者表示快照版本的仓库。关于发布版本和快照版本，第 6.5 节会详细解释。这两个元素下都需要配置 id、name 和 url，id 为该远程仓库的唯一标识，name 是为了方便人阅读，关键的 url 表示该仓库的地址。

往远程仓库部署构件的时候，往往需要认证。配置认证的方式已在第 5.4 节中详细阐述，简而言之，就是需要在 settings.xml 中创建一个 server 元素，其 id 与仓库的 id 匹配，并配置正确的认证信息。不论从远程仓库下载构件，还是部署构件至远程仓库，当需要认证的时候，配置的方式是一样的。

配置正确后，在命令行运行 **mvn clean deploy**，Maven 就会将项目构建输出的构件部署到配置对应的远程仓库，如果项目当前的版本是快照版本，则部署到快照版本仓库地址，否则就部署到发布版本仓库地址。如下是部署一个快照版本的输出：

```
[INFO] --- maven-deploy-plugin:2.4:deploy (default-deploy) @ account-email ---
[INFO] Retrieving previous build number from proj-snapshots
Uploading:
http://192.168.1.100/content/repositories/proj-snapshots/com/juven/mvnbook/account/account-email/1.0.0-SNAPSHOT/account-email-1.0.0-20100103.150936-2.jar
6 KB uploaded at 727.8 KB/sec
[INFO] Retrieving previous metadata from proj-snapshots
[INFO] Uploading repository metadata for:'artifact com.juven.mvnbook.account:account-email'
[INFO] Uploading project information for account-email 1.0.0-20100103.150936-2
[INFO] Retrieving previous metadata from proj-snapshots
[INFO] Uploading repository metadata for:'snapshot com.juven.mvnbook.account:account-email:1.0.0-SNAPSHOT'
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

6.5 快照版本

在 Maven 的世界中，任何一个项目或者构件都必须有自己的版本。版本的值可能是 1.0.0、1.3-alpha-4、2.0、2.1-SNAPSHOT 或者 2.1-20091214.221414-13。其中，1.0.0、1.3-alpha-4 和 2.0 是稳定的发布版本，而 2.1-SNAPSHOT 和 2.1-20091214.221414-13 是不稳定的快照版本。

Maven 为什么要区分发布版和快照版呢？简单的 1.0.0、1.2、2.1 等不就够了吗？为什么还要有 2.1-SNAPSHOT，甚至是长长的 2.1-20091214.221414-13？试想一下这样的情况，小张在开发模块 A 的 2.1 版本，该版本还未正式发布，与模块 A 一同开发的还有模块 B，它由小张的同事季 MM 开发，B 的功能依赖于 A。在开发的过程中，小张需要经常将自己最

新的构建输出，交给季 MM，供她开发和集成调试，问题是，这个工作如何进行呢？

1. 方案一

让季 MM 自己签出模块 A 的源码进行构建。这种方法能够确保季 MM 得到模块 A 的最新构件，不过她不得不去构建模块 A。多了一些版本控制和 Maven 操作还不算，当构建 A 失败的时候，她会是一头雾水，最后不得不找小张解决。显然，这种方式是低效的。

2. 方案二

重复部署模块 A 的 2.1 版本供季 MM 下载。虽然小张能够保证仓库中的构件是最新的，但对于 Maven 来说，同样的版本和同样的坐标就意味着同样的构件。因此，如果季 MM 在本机的本地仓库包含了模块 A 的 2.1 版本构件，Maven 就不会再对照远程仓库进行更新。除非她每次执行 Maven 命令之前，清除本地仓库，但这种要求手工干预的做法显然也是不可取的。

3. 方案三

不停更新版本 2.1.1、2.1.2、2.1.3……。首先，小张和季 MM 两人都需要频繁地更改 POM，如果有更多的模块依赖于模块 A，就会涉及更多的 POM 更改；其次，大量的版本其实仅仅包含了微小的差异，有时候是对版本号的滥用。

Maven 的快照版本机制就是为了解决上述问题。在该例中，小张只需要将模块 A 的版本设定为 2.1-SNAPSHOT，然后发布到私服中，在发布的过程中，Maven 会自动为构件打上时间戳。比如 2.1-20091214.221414-13 就表示 2009 年 12 月 14 日 22 点 14 分 14 秒的第 13 次快照。有了该时间戳，Maven 就能随时找到仓库中该构件 2.1-SNAPSHOT 版本最新的文件。这时，季 MM 配置对于模块 A 的 2.1-SNAPSHOT 版本的依赖，当她构建模块 B 的时候，Maven 会自动从仓库中检查模块 A 的 2.1-SNAPSHOT 的最新构件，当发现有更新时便进行下载。默认情况下，Maven 每天检查一次更新（由仓库配置的 updatePolicy 控制，见第 6.4 节），用户也可以使用命令行-U 参数强制让 Maven 检查更新，如 **mvn clean install-U**。

基于快照版本机制，小张在构建成功之后才能将构件部署至仓库，而季 MM 可以完全不用考虑模块 A 的构建，并且她能确保随时得到模块 A 的最新可用的快照构件，而这一切都不需要额外的手工操作。

当项目经过完善的测试后需要发布的时候，就应该将快照版本更改为发布版本。例如，将 2.1-SNAPSHOT 更改为 2.1，表示该版本已经稳定，且只对应了唯一的构件。相比之下，2.1-SNAPSHOT 往往对应了大量的带有不同时间戳的构件，这也决定了其不稳定性。

快照版本只应该在组织内部的项目或模块间依赖使用，因为这时，组织对于这些快照版本的依赖具有完全的理解及控制权。项目不应该依赖于任何组织外部的快照版本依赖，由于快照版本的不稳定性，这样的依赖会造成潜在的危险。也就是说，即使项目构建今天是成功的，由于外部的快照版本依赖实际对应的构件随时可能变化，项目的构建就可能由于这些外部的不受控制的因素而失败。

6.6 从仓库解析依赖的机制

第5章详细介绍了Maven的依赖机制，本章又深入阐述了Maven仓库，这两者是如何具体联系到一起的呢？Maven是根据怎样的规则从仓库解析并使用依赖构件的呢？

当本地仓库没有依赖构件的时候，Maven会自动从远程仓库下载；当依赖版本为快照版本的时候，Maven会自动找到最新的快照。这背后的依赖解析机制可以概括如下：

1）当依赖的范围是system的时候，Maven直接从本地文件系统解析构件。

2）根据依赖坐标计算仓库路径后，尝试直接从本地仓库寻找构件，如果发现相应构件，则解析成功。

3）在本地仓库不存在相应构件的情况下，如果依赖的版本是显式的发布版本构件，如1.2、2.1-beta-1等，则遍历所有的远程仓库，发现后，下载并解析使用。

4）如果依赖的版本是RELEASE或者LATEST，则基于更新策略读取所有远程仓库的元数据groupId/artifactId/maven-metadata.xml，将其与本地仓库的对应元数据合并后，计算出RELEASE或者LATEST真实的值，然后基于这个真实的值检查本地和远程仓库，如步骤2）和3）。

5）如果依赖的版本是SNAPSHOT，则基于更新策略读取所有远程仓库的元数据groupId/artifactId/version/maven-metadata.xml，将其与本地仓库的对应元数据合并后，得到最新快照版本的值，然后基于该值检查本地仓库，或者从远程仓库下载。

6）如果最后解析得到的构件版本是时间戳格式的快照，如1.4.1-20091104.121450-121，则复制其时间戳格式的文件至非时间戳格式，如SNAPSHOT，并使用该非时间戳格式的构件。

当依赖的版本不明晰的时候，如RELEASE、LATEST和SNAPSHOT，Maven就需要基于更新远程仓库的更新策略来检查更新。在第6.4节提到的仓库配置中，有一些配置与此有关：首先是<releases><enabled>和<snapshots><enabled>，只有仓库开启了对于发布版本的支持时，才能访问该仓库的发布版本构件信息，对于快照版本也是同理；其次要注意的是<releases>和<snapshots>的子元素<updatePolicy>，该元素配置了检查更新的频率，每日检查更新、永远检查更新、从不检查更新、自定义时间间隔检查更新等。最后，用户还可以从命令行加入参数-U，强制检查更新，使用参数后，Maven就会忽略<updatePolicy>的配置。

当Maven检查完更新策略，并决定检查依赖更新的时候，就需要检查仓库元数据maven-metadata.xml。

回顾一下前面提到的RELEASE和LATEST版本，它们分别对应了仓库中存在的该构件的最新发布版本和最新版本（包含快照），而这两个“最新”是基于groupId/artifactId/maven-metadata.xml计算出来的，见代码清单6-5。

代码清单6-5 基于groupId和artifactId的maven-metadata.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>org.sonatype.nexus</groupId>
  <artifactId>nexus</artifactId>
```

```
    <versioning>
      <latest>1.4.2-SNAPSHOT</latest>
      <release>1.4.0</release>
      <versions>
        <version>1.3.5</version>
        <version>1.3.6</version>
        <version>1.4.0-SNAPSHOT</version>
        <version>1.4.0</version>
        <version>1.4.0.1-SNAPSHOT</version>
        <version>1.4.1-SNAPSHOT</version>
        <version>1.4.2-SNAPSHOT</version>
      </versions>
      <lastUpdated>20091214221557</lastUpdated>
    </versioning>
  </metadata>
```

该 XML 文件列出了仓库中存在的该构件所有可用的版本，同时 latest 元素指向了这些版本中最新的那个版本，该例中是 1.4.2-SNAPSHOT。而 release 元素指向了这些版本中最新的发布版本，该例中是 1.4.0。Maven 通过合并多个远程仓库及本地仓库的元数据，就能计算出基于所有仓库的 latest 和 release 分别是什么，然后再解析具体的构件。

需要注意的是，在依赖声明中使用 LATEST 和 RELEASE 是不推荐的做法，因为 Maven 随时都可能解析到不同的构件，可能今天 LATEST 是 1.3.6，明天就成为 1.4.0-SNAPSHOT 了，且 Maven 不会明确告诉用户这样的变化。当这种变化造成构建失败的时候，发现问题会变得比较困难。RELEASE 因为对应的是最新发布版构建，还相对可靠，LATEST 就非常不可靠了，为此，Maven 3 不再支持在插件配置中使用 LATEST 和 RELEASE。如果不设置插件版本，其效果就和 RELEASE 一样，Maven 只会解析最新的发布版本构件。不过即使这样，也还存在潜在的问题。例如，某个依赖的 1.1 版本与 1.2 版本可能发生一些接口的变化，从而导致当前 Maven 构建的失败。

当依赖的版本设为快照版本的时候，Maven 也需要检查更新，这时，Maven 会检查仓库元数据 groupId/artifactId/version/maven-metadata.xml，见代码清单 6-6。

代码清单 6-6　基于 groupId、artifactId 和 version 的 maven-metadata.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>org.sonatype.nexus</groupId>
  <artifactId>nexus</artifactId>
  <version>1.4.2-SNAPSHOT</version>
  <versioning>
    <snapshot>
      <timestamp>20091214.221414</timestamp>
      <buildNumber>13</buildNumber>
    </snapshot>
    <lastUpdated>20091214221558</lastUpdated>
  </versioning>
</metadata>
```

该 XML 文件的 snapshot 元素包含了 timestamp 和 buildNumber 两个子元素，分别代表了这一快照的时间戳和构建号，基于这两个元素可以得到该仓库中此快照的最新构件版本实

际为 1.4.2-20091214.221414-13。通过合并所有远程仓库和本地仓库的元数据，Maven 就能知道所有仓库中该构件的最新快照。

最后，仓库元数据并不是永远正确的，有时候当用户发现无法解析某些构件，或者解析得到错误构件的时候，就有可能是出现了仓库元数据错误，这时就需要手工地，或者使用工具（如 Nexus）对其进行修复。

6.7 镜像

如果仓库 X 可以提供仓库 Y 存储的所有内容，那么就可以认为 X 是 Y 的一个镜像。换句话说，任何一个可以从仓库 Y 获得的构件，都能够从它的镜像中获取。举个例子，http://maven.net.cn/content/groups/public/是中央仓库 http://repo1.maven.org/maven2/在中国的镜像，由于地理位置的因素，该镜像往往能够提供比中央仓库更快的服务。因此，可以配置 Maven 使用该镜像来替代中央仓库。编辑 settings.xml，见代码清单 6-7。

代码清单 6-7 配置中央仓库镜像

```
<settings>
  …
  <mirrors>
    <mirror>
      <id>maven.net.cn</id>
      <name>one of the central mirrors in China</name>
      <url>http://maven.net.cn/content/groups/public/</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  …
</settings>
```

该例中，<mirrorOf>的值为 central，表示该配置为中央仓库的镜像，任何对于中央仓库的请求都会转至该镜像，用户也可以使用同样的方法配置其他仓库的镜像。另外三个元素 id、name、url 与一般仓库配置无异，表示该镜像仓库的唯一标识符、名称以及地址。类似地，如果该镜像需要认证，也可以基于该 id 配置仓库认证。

关于镜像的一个更为常见的用法是结合私服。由于私服可以代理任何外部的公共仓库（包括中央仓库），因此，对于组织内部的 Maven 用户来说，使用一个私服地址就等于使用了所有需要的外部仓库，这可以将配置集中到私服，从而简化 Maven 本身的配置。在这种情况下，任何需要的构件都可以从私服获得，私服就是所有仓库的镜像。这时，可以配置这样的一个镜像，见代码清单 6-8。

代码清单 6-8 配置使用私服作为镜像

```
<settings>
  …
  <mirrors>
```

```
    <mirror>
      <id>internal-repository</id>
      <name>Internal Repository Manager</name>
      <url>http://192.168.1.100/maven2/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

该例中 <mirrorOf> 的值为星号，表示该配置是所有 Maven 仓库的镜像，任何对于远程仓库的请求都会被转至 http://192.168.1.100/maven2/。如果该镜像仓库需要认证，则配置一个 id 为 internal-repository 的 <server> 即可，详见第 5.4 节。

为了满足一些复杂的需求，Maven 还支持更高级的镜像配置：

- **<mirrorOf>*</mirrorOf>**：匹配所有远程仓库。
- **<mirrorOf>external:*</mirrorOf>**：匹配所有远程仓库，使用 localhost 的除外，使用 file://协议的除外。也就是说，匹配所有不在本机上的远程仓库。
- **<mirrorOf>repo1,repo2</mirrorOf>**：匹配仓库 repo1 和 repo2，使用逗号分隔多个远程仓库。
- **<mirrorOf>*,!repo1</mirrorOf>**：匹配所有远程仓库，repo1 除外，使用感叹号将仓库从匹配中排除。

需要注意的是，由于镜像仓库完全屏蔽了被镜像仓库，当镜像仓库不稳定或者停止服务的时候，Maven 仍将无法访问被镜像仓库，因而将无法下载构件。

6.8 仓库搜索服务

使用 Maven 进行日常开发的时候，一个常见的问题就是如何寻找需要的依赖，我们可能只知道需要使用类库的项目名称，但添加 Maven 依赖要求提供确切的 Maven 坐标。这时，就可以使用仓库搜索服务来根据关键字得到 Maven 坐标。本节介绍几个常用的、功能强大的公共 Maven 仓库搜索服务。

6.8.1 Sonatype Nexus

地址：http://repository.sonatype.org/

Nexus 是当前最流行的开源 Maven 仓库管理软件，本书后面会有专门的章节讲述如何使用 Nexus 假设私服。这里要介绍的是 Sonatype 架设的一个公共 Nexus 仓库实例。

Nexus 提供了关键字搜索、类名搜索、坐标搜索、校验和搜索等功能。搜索后，页面清晰地列出了结果构件的坐标及所属仓库。用户可以直接下载相应构件，还可以直接复制已经根据坐标自动生成的 XML 依赖声明，见图 6-3。

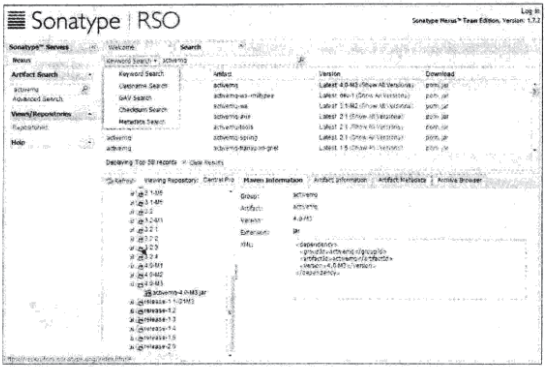

图 6-3 Sonatype Nexus 仓库搜索服务

6.8.2 Jarvana

地址：http://www.jarvana.com/jarvana/

Jarvana 提供了基于关键字、类名的搜索，构件下载、依赖声明片段等功能也一应俱全。值得一提的是，Jarvana 还支持浏览构件内部的内容。此外，Jarvana 还提供了便捷的 Java 文档浏览的功能。Jarvana 的搜索结果页面如图 6-4 所示。

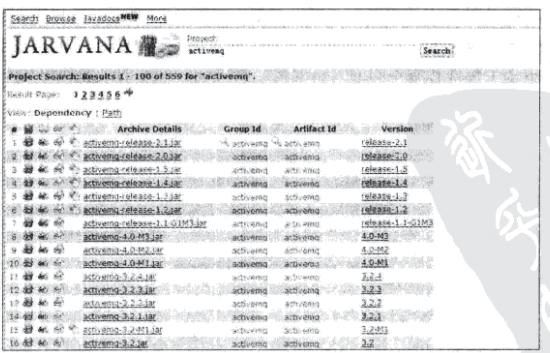

图 6-4 Jarvana 仓库搜索服务

6.8.3 MVNbrowser

地址：http://www.mvnbrowser.com

MVNbrowser 只提供关键字搜索的功能，除了提供基于坐标的依赖声明代码片段等基本功能之外，MVNbrowser 的一大特色就是，能够告诉用户该构件的依赖于其他哪些构件（Dependencies）以及该构件被哪些其他构件依赖（Referenced By），如图 6-5 所示。

图 6-5 MVNbrowser 仓库搜索服务

6.8.4 MVNrepository

地址：http://mvnrepository.com/

MVNrepository 的界面比较清新，它提供了基于关键字的搜索、依赖声明代码片段、构件下载、依赖与被依赖关系信息、构件所含包信息等功能。MVNrepository 还能提供一个简单的图表，显示某个构件各版本间的大小变化。MVNrepository 的页面如图 6-6 所示。

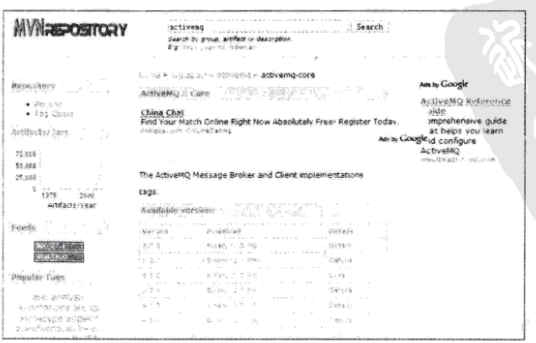

图 6-6 MVNrepository 仓库搜索服务

6.8.5 选择合适的仓库搜索服务

上述介绍的四个仓库搜索服务都代理了主流的 Maven 公共仓库，如 central、JBoss、Java. net 等。这些服务都提供了完备的搜索、浏览、下载等功能，区别只在于页面风格和额外功能。例如，Nexus 提供了其他三种服务所没有的基于校验和搜索的功能。用户可以根据喜好和特殊需要选择最合适自己的搜索服务，当然，也可以综合使用所有这些服务。

6.9 小结

本章深入阐述了仓库这一 Maven 核心概念。首先介绍了仓库的由来；接着直接剖析了一段 Maven 源码，介绍仓库的布局，以方便读者将仓库与实际文件联系起来；而仓库的分类这一部分则分别介绍了本地仓库、远程仓库、中央仓库以及私服等概念；基于这些概念，又详细介绍了仓库的配置；在此基础上，我们再深入仓库的内部工作机制，并同时解释了 Maven 中快照的概念。本章还解释了镜像的概念及用法。最后，本章介绍了一些常用的仓库搜索服务，以方便读者的日常开发工作。