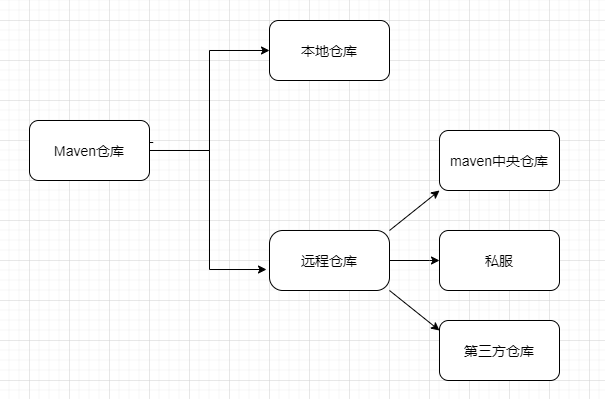
# Maven仓库

## Maven仓库的分类：

Maven有两种类型的仓库，一种是本地仓库，一种是远程仓库



【注】一个远程仓库中也分为很多种类型的库，如central库、snapshot库、release库等。

【注】第三方仓库一般就是中央仓库的镜像，是为了分担中央仓库的压力

Jar包寻找路径：本地仓库 → 私服 → 第三方仓库 → 中央仓库

## 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：

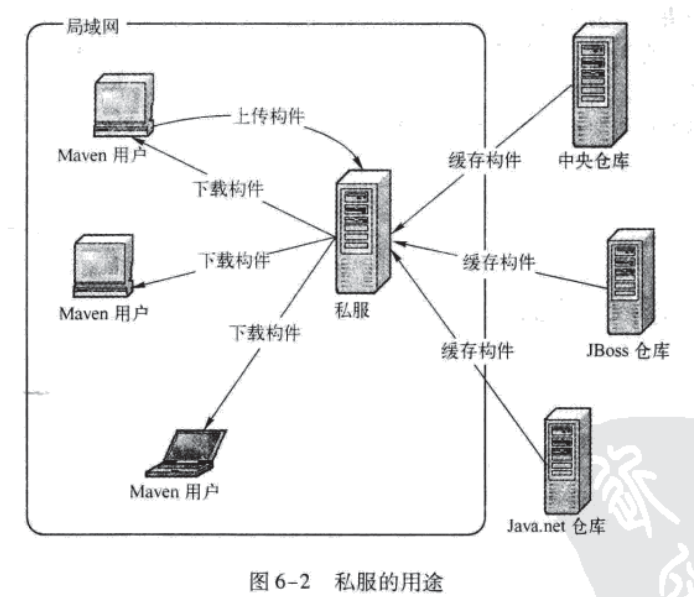


图6-2展示的是组织内部使用私服的情况。

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。因此私服可以帮助你：

* **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**：

不停的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的检查远程仓库数据**。因此，**当项目配置了很多外部远程仓库的时候，构建的速度会大大降低**。使用私服可以很好的解决这一问题，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，便可以讲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。使用私服之后，即使暂时没有Internet连接，由于私服中已经缓存了大量构建，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很容易的事情，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。

## 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：

|  |
| --- |
| <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>  </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了该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。这里需要注意到是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Maven2及Maven3的默认布局，而不是Maven1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> |

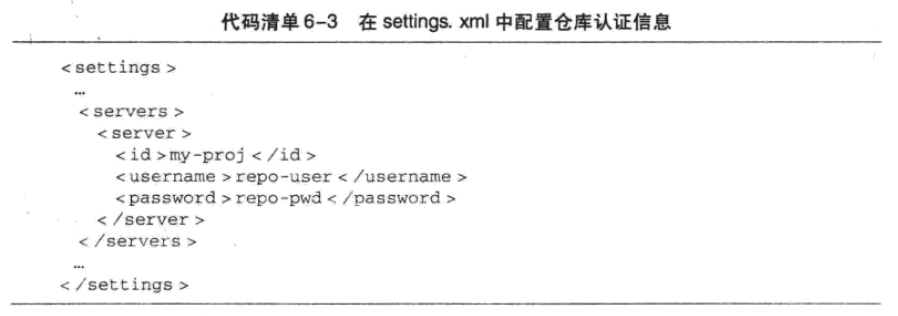
元素updatePolicy用来配置Maven从远程仓库检查更新的频率，默认值是daily，表示Maven每天检查一次。其他可用的包括：never——从不检查更新；always——每次构件都检查更新；interval:X——每隔X分钟检查一次更新（X为任意整数）。

元素checksumPolicy用来配置Maven检查校验和文件的策略。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，如果校验和校验失败，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——遇到校验和失败就让构建失败；ignore——使Maven完全忽略校验和文件。

## 远程仓库的认证：

大部分远程仓库无需认证就可以访问，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 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。



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。

## 部署至远程仓库：

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件部署到仓库中。首先，需要编辑项目的POM文件。配置distributionManagement元素见代码抢单6-4：

|  |
| --- |
| <project>  ...  <distributionManagement>  <repository>  <id>proj-releases</id>  <name>Project Release Repository</name>  <url>http://192.168.1.100/content/repositories/proj-releases</url>  </repository>  <snapshotRepository>  <id>proj-snapshots</id>  <name>Project 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。

这两个元素下都需要配置id、name、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。配置认证的方式在[远程仓库的认证]这一小节已经详细阐述，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。

## 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或者2.1-20091214.221414-13是不稳定的快照版本。

Maven为什么要区分发布版本和快照版本呢？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件输出，交给季MM，供她开发和集成测试，问题是，这个工作如何进行呢？

* **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失败时，它会是一头雾水，最后不得不找小张解决。显然，这种方式是低效的。

* **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。

* **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两人都需要频繁的更改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，当发现有最更新的构件时进行下载。默认情况下，Maven每天检查一次更新（由仓库的updatePolicy控制），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建部署至仓库，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。

|  |
| --- |
| 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。 |

## 从仓库解析依赖的机制：

之前详细介绍了Maven的依赖机制，本章有深入阐述了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？

当本地苍鹭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。这背后的依赖即系机制可以概括如下：

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素局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。

当依赖的版本不明确的时候，如RELEASE、LATEST和SNAPSHOT，Maven就需要

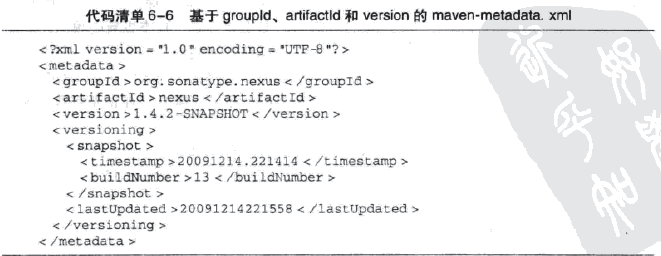
基于更新远程仓库的更新策略来检查更新。当Maven检查完更新策略，并决定检查依赖更新的时候，就需要检查仓库元数据maven-metadata.xml。

回顾一下前面提到的RELEASE和LATEST，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：



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，该例中时1.4.2-SNAPSHOT。而release元素指向了这些版本中最新的发布版本，该例中时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。

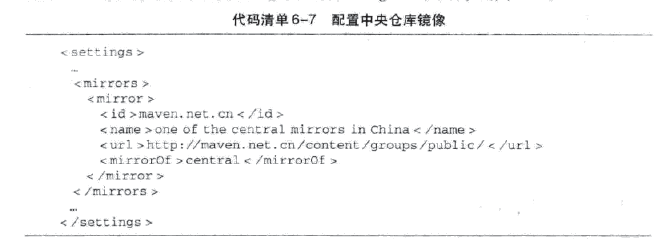
当依赖的版本设为快照版本的时候，Maven也需要检查更新，这时，Maven会检查仓库元数据groupId/artifactId/version/maven-metadata.xml，见代码清单6-6：



该XML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。

## 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：



该例中，<mirrorOf>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。类似的，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：



该例中<mirrorOf>的值为\*，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。

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。