# [Using Mirrors for Repositories](http://maven.apache.org/guides/mini/guide-mirror-settings.html)

先说mirrors中mirror的作用，mirrors允许配置多个镜像mirror，mirror的作用一般是为了使用一个更快速度的镜像，主要是访问速度的考虑，或者突破屏蔽使用可以访问的镜像。

在maven中配置一个mirror时，mirror通常会设置成\*，还有可能是一个具体的值，还有可能是逗号隔开的几个值..

例如:

1.\*

2.repo3

3.repo1,repo2,\*,!repo3

4.external:\*

在上面4个例子中，第一个例子中\*的意思就是（根据mirrorOf和repository的id）匹配所有的库（repository），这样就是说如果你需要某个jar，他会从镜像地址去下载这个jar。不管你配置了多少个库，即使这些库的地址不一样，仍然会从镜像地址访问，这里就产生了问题。

**问题：**为什么我配置好了库（repository），但是下载不下来（或者下载地址不对）？

这个问题已经很明白了，就是镜像（mirror）的问题。

**问题：**如何解决上述问题呢？

这里请继续看第二个例子和第三个例子。

第二个例子只配置了一个repo3，这个镜像只能匹配id是repo3的库，该库会使用repo3的镜像访问。这里说**重点**了，即使**还**存在一个\*的，这里会先查找完全相同的存储库标识符匹配的镜像，如果找到了，就用匹配的，这里就是repo3，这点可以放心。如果找不到，那就去根据其他的mirrorOf查找去，查找到一个就使用。这里需要注意的是，如果存在多个重复的mirrorOf，只会用排名靠前的第一个（按照配置顺序来）。如果一个都没有，就会用maven中央库。

第三个例子是用逗号隔开的好几个，并且有一个特殊的!repo3，一般!的意思是非，这里的意思就是，就是排除repo3的情况。综合起来第三个的含义就是完全匹配repo1,repo2和其他全部库，排除repo3库。

到这里，第二个问题已经有了答案，给这个库配置单独的镜像，而且位置尽量在前面，id和mirrorOf一致。说到这两个一致，**还得说两个不能一致的情况**，就是mirror的mirrorOf不能和任何mirror的id一致，因为id在setting中唯一，mirrorOf要和库的id一致，所以势必不能和mirror的id一致，如果一致显然就起不到作用了（这里属于猜测，官方要求不能一致）。

第四个例子很特殊，含义就是匹配本地库之外的全部库。如果本地库存在，就用本地库的。

到这里，有关mirrorOf的内容说完了，记录下我解决问题的思路和原因以及解决办法，希望对你有所帮助。

# 手动导入仓库

mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14 -Dversion=10.2.0.4.0   
-Dpackaging=jar -Dfile=ojdbc14-10.2.0.4.0.jar

-Dfile = 这里要填我们自己的jar包的名字

<modelVersion>4.0.0</modelVersion>  
<groupId>com.oracle</groupId>  
<artifactId>ojdbc14</artifactId>  
<version>10.2.0.4.0</version>  
<package>jar</package>

# maven生命周期

Maven定义了三套生命周期：clean、default、site，每个生命周期都包含了一些阶段（phase）。三套生命周期相互独立，但各个生命周期中的phase却是有顺序的，且后面的phase依赖于前面的phase。执行某个phase时，其前面的phase会依顺序执行，但不会触发另外两套生命周期中的任何phase。

## clean生命周期

pre-clean ：执行清理前的工作；

clean ：清理上一次构建生成的所有文件；

post-clean ：执行清理后的工作

## default生命周期

default生命周期是最核心的，它包含了构建项目时真正需要执行的所有步骤。

1. validate 验证工程是否正确，所有需要的资源是否可用。
2. initialize
3. generate-sources
4. process-sources
5. generate-resources
6. process-resources ：复制和处理资源文件到target目录，准备打包；
7. compile ：编译项目的源代码；
8. process-classes
9. generate-test-sources
10. process-test-sources
11. generate-test-resources
12. process-test-resources
13. test-compile ：编译测试源代码；
14. process-test-classes
15. test ：运行测试代码；
16. prepare-package
17. package ：打包成jar或者war或者其他格式的分发包；
18. pre-integration-test
19. integration-test 如有需要，将包处理和发布到一个能够进行集成测试的环境。
20. post-integration-test
21. verify 运行所有检查，验证包是否有效且达到质量标准。
22. install ：将打好的包安装到本地仓库，供其他项目使用；
23. deploy ：将打好的包安装到远程仓库，供其他项目使用；

## site生命周期

pre-site

site ：生成项目的站点文档；

post-site

site-deploy ：发布生成的站点文档

# build 标签

<build>  
<defaultGoal>install</defaultGoal>  
<directory>${basedir}/target</directory>  
<finalName>${artifactId}-${version}</finalName>  
<filters>  
  <filter>filters/filter1.properties</filter>  
</filters>  
...  
</build>

* **defaultGoal**:不指定 goal 是默认用它，指定了用指定的
* **directory**: 默认 ${basedir}/target. 源码、测试，配置文件的输出根目录。
* **finalName**: This is the name of the bundled project when it is finally built (sans the file extension, for example: my-project-1.0.jar). It defaults to ${artifactId}-${version}. The term "finalName" is kind of a misnomer, however, as plugins that build the bundled project have every right to ignore/modify this name (but they usually do not). For example, if the maven-jar-plugin is configured to give a jar a classifier of test, then the actual jar defined above will be built as my-project-1.0-test.jar.
* **filter**: 指定 ${} 起作用的文件，没指定的文件里面的 ${} 不起作用

<build>  
  <sourceDirectory>${basedir}/src/main/java</sourceDirectory>  
  <scriptSourceDirectory>${basedir}/src/main/scripts</scriptSourceDirectory>  
  <testSourceDirectory>${basedir}/src/test/java</testSourceDirectory>  
  <outputDirectory>${basedir}/target/classes</outputDirectory>    
  directory指定根，这个指定class文件  
  <testOutputDirectory>${basedir}/target/test-classes</testOutputDirectory>  
  ...  
</build>

<build>  
  ...  
  <resources> 源码的配置文件  
    <resource>  
      <targetPath>META-INF/plexus</targetPath> 配置文件输出路径，class目录同级  
      <filtering>false</filtering> 是否将 ${} 替换  
      <directory>${basedir}/src/main/plexus</directory> 配置文件搜索的范围  
      <includes>  
        <include>configuration.xml</include> 在搜索范围中找符合条件的配置文件，不符合条件的配置文件不要  
      </includes>  
      <excludes>  
        <exclude>\*\*/\*.properties</exclude> 排除配置文件  
      </excludes>  
    </resource>  
  </resources>  
  <testResources> 测试的配置文件  
    ...  
  </testResources>  
  ...  
</build>

# [dependency](http://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html) 工具

## 自定义setttins

mvn命令行工具，默认只读取maven安装目录下得settings 和 .m 文件夹下的 settings 配置，如果改名字是识别不了的。如果自定义要加 –s 参数

**mvn dependency:list -s "C:\Users\Administrator\.m2\settings\_wuhan\_20180806.xml"**

## Analyze

* [dependency:analyze](http://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html) analyzes the dependencies of this project and determines which are: used and declared; used and undeclared; unused and declared.

使用了，声明了

使用了，没声明

没使用，声明了

## [analyze-dep-mgt](http://maven.apache.org/plugins/maven-dependency-plugin/analyze-dep-mgt-mojo.html)

analyzes your projects dependencies and lists mismatches between resolved dependencies and those listed in your dependency Management section.

找出 dependency tree 和 dependency Management 工具之间的差别，提示信息如下

[INFO] com.pt.poseidon:poseidon-ws:jar was excluded in DepMgt, but version 1.0.0 has been found in the dependency tree.

poseidon-ws 在 dependency Management中已经被排除了，但是当前工程中还是使用了1.0.0 的版本

## analyze-only

* [dependency:analyze-only](http://maven.apache.org/plugins/maven-dependency-plugin/analyze-only-mojo.html) is the same as analyze, but is meant to be bound in a pom. It does not fork the build and execute test-compile.

跟analyze命令类似，不做 build 、不执行 test-compile

## [build-classpath](http://maven.apache.org/plugins/maven-dependency-plugin/build-classpath-mojo.html)

列出全部依赖的jar的全路径

## go-offline

* [dependency:go-offline](http://maven.apache.org/plugins/maven-dependency-plugin/go-offline-mojo.html) tells Maven to resolve everything this project is dependent on (dependencies, plugins, reports) in preparation for going offline.

为下线做准备，列出全部依赖，准备下线

## list

* [dependency:list](http://maven.apache.org/plugins/maven-dependency-plugin/list-mojo.html) alias for resolve that lists the dependencies for this project.

以列表的形式，列出依赖jar 包

## resolve

* [dependency:resolve](http://maven.apache.org/plugins/maven-dependency-plugin/resolve-mojo.html) tells Maven to resolve all dependencies and displays the version. **JAVA 9 NOTE:** *will display the module name when running with Java 9.*

解析jar文件

## sources

* [dependency:sources](http://maven.apache.org/plugins/maven-dependency-plugin/sources-mojo.html) tells Maven to resolve all dependencies and their source attachments, and displays the version.

解析依赖关系，然后下载源文件

## tree

* [dependency:tree](http://maven.apache.org/plugins/maven-dependency-plugin/tree-mojo.html) displays the dependency tree for this project.
* 树形显示