

目录

1 工具使用	1
1.1 shell	1
1.2 git 基本命令	2
1.3 Maven 使用初步	3
1.4 Maven 常用插件	5

§1 工具使用

§1.1 shell

标准输入 `stdin`，标准输出 `stdout`，标准错误 `stderr`，其文件描述符分别为 `0,1,2`。> 默认为将标准输出（`stdout`）重定向到其它地方，`2>&1` 表示将标准错误输出（`stderr`）重定向到标准输出（`stdout`）。`&>file` 表示把标准输出（`stdout`）和标准错误（`stderr`）都输出到文件 `file` 中，因此，`2>1` 表示将 `stderr` 重定向到文件 `1` 中，而不是 `stdout`。

`let` 命令的替代表示形式是：((算术表达式))。例如，`let 'j=i*6+2'` 等价于 `((j=i*6+2))`。当表达式中有 Shell 的特殊字符时，必须用双引号或单引号将其括起来。例如，`let ``val=a|b```。如果不括起来，Shell 会把命令行 `let val=a|b` 中的 `'|'` 看作管道符号，将其左右两边看成不同的命令，因此，将无法正确执行。

linux 中除了常见的读（`r`）、写（`w`）、执行（`x`）权限以外，还有 3 个特殊的权限，分别是 `setuid`、`setgid` 和 `stick bit`。

```
[root@MyLinux ~]# ls -l /usr/bin/passwd /etc/passwd
-rw-r--r-- 1 root root 1549 08-19 13:54 /etc/passwd
-rwsr-xr-x 1 root root 22984 2007-01-07 /usr/bin/passwd
```

`/etc/passwd` 文件存放的各个用户的账号与密码信息，`/usr/bin/passwd` 是执行修改和查看此文件的程序，但从权限上看，`/etc/passwd` 仅有 `root` 权限的写（`w`）权，可实际上每个用户都可以通过 `/usr/bin/passwd` 命令去修改这个文件，于是这里就涉及了 linux 里的特殊权限 `setuid`，正如 `-rwsr-xr-x` 中的 `s`，`setuid` 就是：让普通用户拥有可以执行“只有 `root` 权限才能执行”的特殊权限，`setgid` 同理是让普通用户拥有“组用户”才能执行的特殊权限”。

`/tmp` 目录是所有用户共有的临时文件夹，所有用户都拥有读写权限，这就必然出现一个问题，A 用户在 `/tmp` 里创建了文件 `a`，此时 B 用户看了不爽，在 `/tmp` 里把它给删了（因为拥有读写权限），那肯定是不行的。实际上不会发生这种情况，因为有特殊权限 `stick bit`（粘贴位）权限，`drwxrwxrwt` 中的最后一个 `t` 的意思是：除非目录的 `owner` 和 `root` 用户才有权限删除它，除此之外其它用户不能删除和修改这个目录。也就是说，`/tmp` 目录中，只有文件的拥有者和 `root` 才能对其修改和删除，其他用户则不行，避免了上面所说的问题。

如何设置以上特殊权限，如下所示，`suid` 的二进制串为 `100`，换算十进制为 `4`，`guid` 的二进制串为 `010`，`stick bit` 二进制串为 `001`，换算成 `1`。在一些文件设置了特殊权限后，字母不是小写的 `s` 或者 `t`，而是大写的 `S`

和 T, 那代表此文件的特殊权限没有生效, 是因为尚未赋予它对应用户的 x 权限。

setuid: chmod u+s xxx	setuid:chmod 4755 xxx
setgid: chmod g+s xxx	setgid:chmod 2755 xxx
stick bit : chmod o+t xxx	stick bit:chmod 1755 xxx

§1.2 git 基本命令

`git add` 命令主要用于把要提交的文件的信息添加到索引库中, 当我们使用 `git commit` 时, `git` 将依据索引库中的内容来进行文件的提交。`git add <path>` 表示 add to index only files created or modified and not those deleted, 因此 `git add .` 添加的文件不包括已经删除了的文件, `<path>` 可以是文件也可以是目录。`git` 不仅能判断出 `<path>` 中, 修改 (不包括已删除) 了的文件, 还能判断出新建的文件, 并把它们的信息添加到索引库中。

`git add -u [<path>]` 表示 add to index only files modified or deleted and not those created, 即把 `<path>` 中所有 tracked 文件中被修改过或已删除文件的信息添加到索引库, 它不会处理 untracked 的文件。`git add -A [<path>]` 表示把 `<path>` 中所有 tracked 文件中被修改过或已删除文件和所有 untracked 的文件信息添加到索引库。省略 `<path>` 表示 '.', 即当前目录。使用 `git add .` 后, 如果打算撤销这次 add, 则使用命令: `git rm -r --cached .`, 打算忽略整个目录时, 可以添加 `.gitignore` 文件, 表示忽略当前目录下的 `mapreduce/ESMapReduce/lib/` 目录。

```
*.class
# Package Files #
mapreduce/ESMapReduce/lib/
*.jar
*.war
*.ear
```

`git` 提交环节, 存在三大部分: working tree, index file, commit. working tree 是工作所在的目录, 每当在代码中进行了修改, working tree 的状态就改变了。index file 是索引文件, 它是连接 working tree 和 commit 的桥梁, 每当我们使用 `git add` 命令后, index file 的内容就改变了, 此时 index file 和 working tree 完成了同步。commit 是代码的一次提交, 只有完成提交, 代码才真正地进入 `git` 仓库, 使用 `git commit` 就是将 index file 里的内容提交到 commit 中。因此, `git diff` 是查看 working tree 与 index file 的差别的, `git diff --cached` 是查看 index file 与 commit 的差别的, `git diff HEAD` 是查看 working tree 和 commit 的差别的 (注意, HEAD 代表最近一次 commit)。

§1.3 Maven 使用初步

(1) 配置 maven 环境，最主要的是设置环境变量：M2_HOME，将其设置为 maven 安装目录，例子目录为：/usr/share/maven；

(2) 修改仓库位置，仓库用于存放平时项目开发依赖的所有 jar 包。例子仓库路径：/opt/maven/repo，为设置仓库路径，必须修改 \$M2_HOME/conf 目录下的 setting.xml 文件。

```
<localRepository>/opt/maven/repo</localRepository>
```

在 shell 中输入并执行 mvn help:system，如果没有错误，在仓库路径下应该多了些文件，这些文件是从 maven 的中央仓库下载到本地仓库的。

(3) 创建 maven 项目，通过 maven 命令行方式创建一个项目，命令为：mvn archetype:create -DgroupId=com.mvn.test -DartifactId=hello -DpackageName=com.mvn.test -Dversion=1.0

由于第一次构建项目，所有依赖的 jar 包都要从 maven 的中央仓库下载，所以需要时间等待。做完这一步后，在工程根目录下应该有个 pom.xml 文件，其中 groupId, artifactId 和 version 比较常用。

- project: pom.xml 文件的顶层元素；
- modelVersion: 指明 POM 使用的对象模型的版本，这个值很少改动。
- groupId: 指明创建项目的小组的唯一标识。GroupId 是项目的关键标识，此标识以组织的完全限定名来定义。如 org.apache.maven.plugins 是所有 Maven 插件项目指定的 groupId。
- artifactId: 指明此项目产生的主要产品的基本名称。项目的主要产品通常为一个 jar 包，源代码包通常使用 artifactId 作为最后名称的一部分。典型的产品名称使用这个格式：<artifactId>-<version>.<extension>(比如：myapp-1.0.jar)。
- version: 项目产品的版本号。Maven 帮助你管理版本，可以经常看到 SNAPSHOT 这个版本，表明项目处于开发阶段。
- name: 项目的显示名称，通常用于 maven 产生的文档中。
- url: 指定项目站点，通常用于 maven 产生的文档中。
- description: 描述此项目，通常用于 maven 产生的文档中。

(4) 项目 hello 已经创建完成，但它并不是 eclipse 所需要的项目目录格式，需要把它构建成 eclipse 可以导入的项目。进入到刚创建的项目目录 (~/.workspace/hello)，执行：mvn clean (告诉 maven 清理输出目录

target), 然后执行 `mvn compile` (告诉 maven 编译项目 main 部分的代码, 或者两步合成一步 `mvn clean compile`), 此次仍会下载 jar 包到仓库中。编译后, 项目的目录结构并不可以直接导入到 Eclipse, 需执行命令: `mvn eclipse:eclipse`, 命令执行完成后就能 import 到 Eclipse。

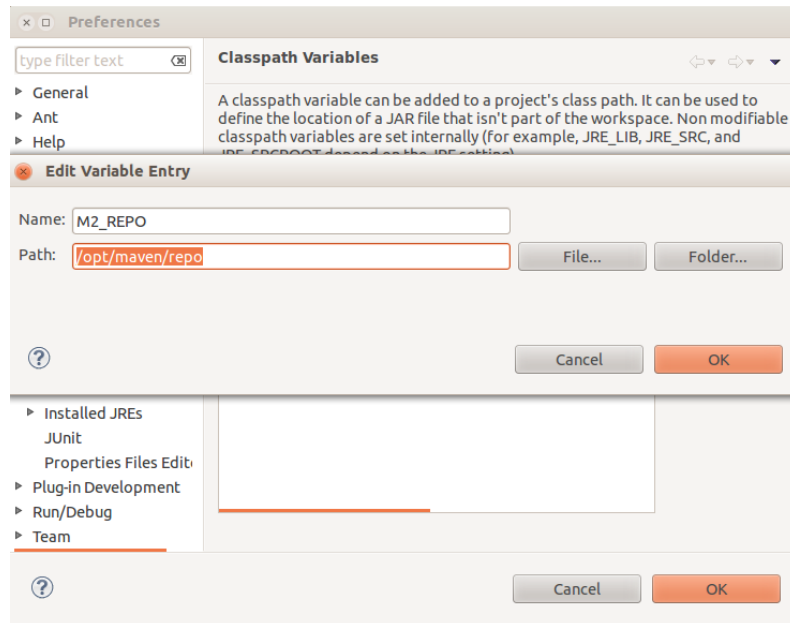


图 1-1 Eclipse 配置 Maven 仓库路径

- (5) 打开 eclipse, 在其中配置 maven 仓库路径, 配置路径为: Window-->Perferences-->java-->Build Path-->Classpath Variables, 新建变量 (M2_REPO) 的类路径, 如图 1-1 示。
- (6) 包的更新与下载, 如果发现 junit 版本比较旧, 想换成新版本, 修改项目下的 `pom.xml` 文件为

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

就可改变 junit 的版本号, 在以后的 maven 操作中, maven 会自动下载依赖的 jar 包。Maven 中央仓库地址为 <http://search.maven.org>, 假如想下载 struts 的 jar 包, 可在 url 内搜索 struts。

- (7) 某些 jar 包可能位于远程机器上, 因此需要配置 maven 仓库, 配置仓库代码如下, 其中的 id 属性无特别含义, 仅用于标识仓库。当下载完某个 jar 包后, 其在本地仓库的相对路径形如 <groupId>/<artifactId>/<version>/*.jar, 例如: /opt/maven/repo/org/ansj/tree_split/1.0.1/tree_split-1.0.1.jar。

```
<repositories>
  <repository>
    <id>ansj-maven-repo</id>
    <url>https://raw.github.com/ansjsun/mvn-repo/gh-pages</url>
  </repository>
  <repository>
    <id>remote-maven-repo</id>
    <url>http://search.maven.org</url>
  </repository>
</repositories>
```

- (8) 有时候 Maven 访问很慢, 可在 settings.xml 里重新设置中央仓库的镜像, 中央仓库的默认地址为: <http://repo.maven.apache.org/maven2/>

```
<mirrors>
  <mirror>
    <id>ibiblio</id>
    <mirrorOf>central</mirrorOf>
    <name>Human Readable Name for this Mirror.</name>
    <url>http://mirrors.ibiblio.org/maven2</url>
  </mirror>
</mirrors>
```

打算执行 maven 工程下某个类时, 例如 test 目录下的某个类, 执行命令如下: `mvn exec:java -X -Dexec.mainClass="org.ansj.liubo.test.test" -Dexec.classpathScope=test`, 其中的 classpathScope=test 告诉 maven 打算执行 test 类, 而非工程的主类。执行 test 部分的某些类之前, 必须执行 `mvn test-compile`, 以编译 test 部分的代码, 如果没有, 执行过程会报错。当准备向执行类添加参数时, 使用如下命令。

```
mvn exec:java -Dexec.mainClass="org.ansj.liubo.test.test"
-Dexec.args="/mnt/f/tmp/content.txt /mnt/f/tmp/result3.txt"
-Dexec.classpathScope=test
```

§1.4 Maven 常用插件

Maven 本质上是一个插件框架, 其核心并不执行任何具体的构建任务, 所有这些任务都交给插件完成, 例如编译源代码由 maven-compiler-plugin 完成。进

一步说, 每个任务对应一个插件, 每个插件会有一个或者多个目标 (goal), 例如 `maven-compiler-plugin` 的 `compile` 目标用来编译位于 `src/main/java/` 目录下的主源码, 而 `testCompile` 目标则用来编译位于 `src/test/java/` 目录下的 `code`。

用户可通过两种方式调用 Maven 插件目标。第一种方式是将插件目标与生命周期阶段 (lifecycle phase) 绑定, 这样用户在命令行只输入了生命周期阶段, 例如 Maven 默认将 `maven-compiler-plugin` 的 `compile` 目标与 `compile` 生命周期阶段绑定, 因此命令 `mvn compile` 实际上先定位到 `compile` 这一生命周期阶段, 然后再根据绑定关系调用 `maven-compiler-plugin` 的 `compile` 目标。第二种方式是直接在命令行指定要执行的插件目标, 例如 `mvn archetype:generate` 就表示调用 `maven-archetype-plugin` 的 `generate` 目标, 这种带冒号的调用方式与生命周期无关。

Maven 有两个插件列表, 第一个列表的 GroupId 为 `org.apache.maven.plugins`, 这里的插件最为成熟, 具体地址为: <http://maven.apache.org/plugins/index.html>。第二个列表的 GroupId 为 `org.codehaus.mojo`, 这里的插件没有那么成熟, 但也十分有用, 地址为: <http://mojo.codehaus.org/plugins.html>。

为使项目结构更为清晰, Maven 区别对待 Java 代码文件和资源文件, `maven-compiler-plugin` 用来编译 java 代码, `maven-resources-plugin` 则用来处理 resource 文件。默认的资源文件目录是 `src/main/resources`。很多用户会添加额外的资源文件目录, 这个时候就可以通过配置 `maven-resources-plugin` 来实现。此外, 资源文件过滤也是 Maven 的一大特性, 可以在资源文件中使用 `$propertyName` 形式的 Maven 属性, 然后配置 `maven-resources-plugin` 以开启对资源文件的过滤, 之后就可以通过命令行或者 Profile, 针对不同环境传入不同的属性值, 以实现灵活构建。

由于历史原因, Maven2/3 中用于执行测试的插件不是 `maven-test-plugin`, 而是 `maven-surefire-plugin`。其实大部分时间内, 只要测试类遵循通用的命令约定 (以 `Test` 结尾、以 `TestCase` 结尾、或者以 `Test` 开头), 就几乎不用知晓该插件是否存在。然而在当你想要跳过测试、排除某些测试类、或者使用一些 `Test` 特性时, 了解 `maven-surefire-plugin` 的一些配置选项就很有用了。例如 `mvn test -Dtest=FooTest` 这样一条命令的效果是仅运行 `FooTest` 测试类, 这是通过控制 `maven-surefire-plugin` 的 `test` 参数实现的。

Maven 默认只允许指定一个主 Java 代码目录和一个测试 Java 代码目录, 虽然这是一个应当尽量遵守的约定, 但偶尔用户还是希望能够指定多个源码目录 (例如为了应对遗留项目), `build-helper-maven-plugin` 的 `add-source` 目标就服务于这个目的, 通常它被绑定到默认生命周期的 `generate-sources` 阶段以添加额外的源码目录。这种做法是不推荐的, 因为它破坏了 Maven 的约定, 而

且可能会遇到其他严格遵守约定的插件工具无法正确识别额外的 `source` 目录。`build-helper-maven-plugin` 的另一个非常有用的目标是 `attach-artifact`，使用该目标你可以以 `classifier` 的形式选取部分项目文件生成附属构件，并同时 `install` 到本地仓库，也可以 `deploy` 到远程仓库。

`exec-maven-plugin` 很好理解，顾名思义，它能让你运行任何本地的系统程序，在某些特定情况下，运行一个 Maven 外部的程序可能就是最简单的问题解决方案，这就是 `exec:exec` 的用途，当然，该插件还允许你配置相关的程序运行参数。除了 `exec` 目标之外，`exec-maven-plugin` 还提供了一个 `java` 目标，该目标要求你提供一个 `mainClass` 参数，然后它能够利用当前项目的依赖作为 `classpath`，在同一个 JVM 中运行该 `mainClass`。有时候，为了简单的演示一个命令行 Java 程序，可以在 `pom.xml` 中配置好 `exec-maven-plugin` 的相关运行参数，然后直接在命令运行 `mvn exec:java` 以查看运行效果。

进行 Web 开发时，打开浏览器对应用进行手动的测试几乎是无法避免的，这种测试方法通常就是将项目打包成 `war` 文件，然后部署到 Web 容器中，再启动容器进行验证，这显然十分耗时。为了帮助开发者节省时间，`jetty-maven-plugin` 应运而生，它完全兼容 Maven 项目的目录结构，能够周期性地检查源文件，一旦发现变更后自动更新到内置的 Jetty Web 容器中。做一些基本配置后（例如 Web 应用的 `contextPath` 和自动扫描变更的时间间隔），只要执行 `mvn jetty:run`，然后在 IDE 中修改代码，代码经 IDE 自动编译后产生变更，再由 `jetty-maven-plugin` 侦测到后将更新写入到 Jetty 容器，这时就可以直接测试 Web 页面。需要注意的是，`jetty-maven-plugin` 并不是宿主于 Apache 或 Codehaus 的官方插件，因此使用的时候需要额外的配置 `settings.xml` 的 `pluginGroups` 元素，将 `org.mortbay.jetty` 这个 `pluginGroup` 加入。

很多 Maven 用户遇到这样一个问题，当项目包含大量模块的时候，集体更新版本就变成一件烦人的事情，到底有没有自动化工具能帮助完成这件事情呢？（当然你可以使用 `sed` 之类的文本操作工具）答案是肯定的，`versions-maven-plugin` 提供了很多目标帮助你管理 Maven 项目的各种版本信息。例如最常用的命令 `mvn versions:set -DnewVersion=1.1-SNAPSHOT` 就能帮助你把所有模块的版本更新到 `1.1-SNAPSHOT`。该插件还提供了其他一些很有用的目标，`display-dependency-updates` 能告诉你项目依赖有哪些可用的更新，类似的 `display-plugin-updates` 能告诉你可用的插件更新，`use-latest-versions` 能自动帮你将所有依赖升级到最新版本。最后，如果对所做的更改满意，则可以使用 `mvn versions:commit` 提交，不满意的话也可以使用 `mvn versions:revert` 进行撤销操作。