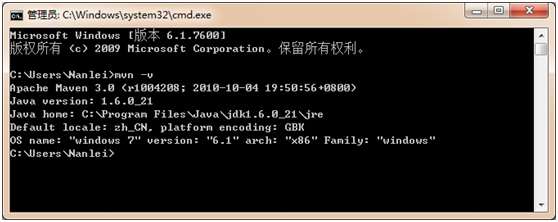
[**用Maven构建Java Web开发环境（Jetty容器）之一**](http://sarin.iteye.com/blog/784275)

**博客分类：**

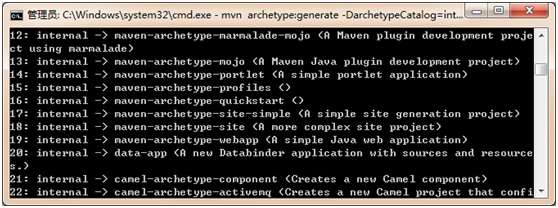
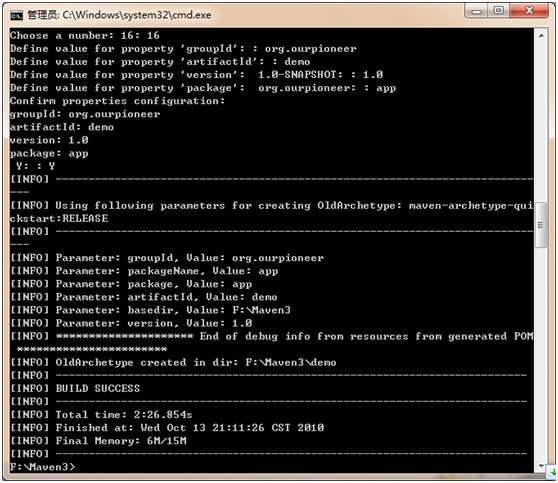
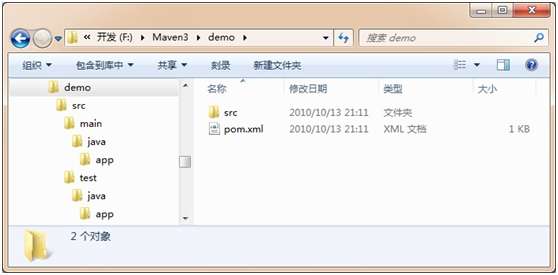
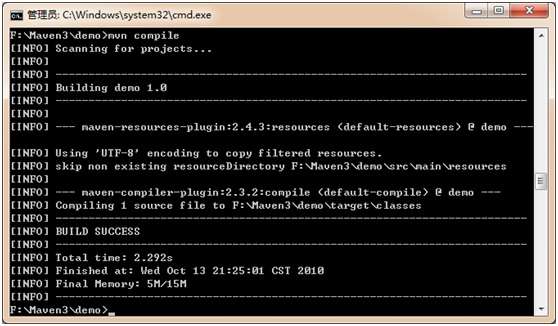
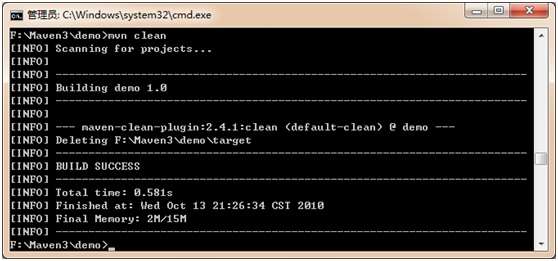
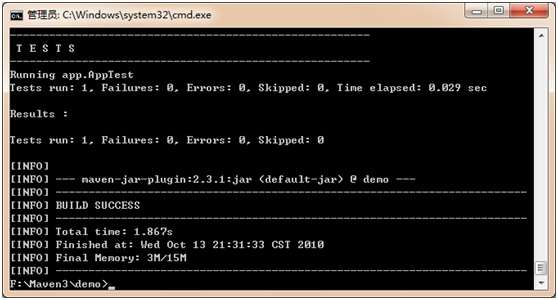
* [积少成多](http://sarin.iteye.com/category/93571)

[maven](http://www.iteye.com/blogs/tag/maven)[Web](http://www.iteye.com/blogs/tag/Web)[Java](http://www.iteye.com/blogs/tag/Java)[项目管理](http://www.iteye.com/blogs/tag/%E9%A1%B9%E7%9B%AE%E7%AE%A1%E7%90%86)[应用服务器](http://www.iteye.com/blogs/tag/%E5%BA%94%E7%94%A8%E6%9C%8D%E5%8A%A1%E5%99%A8)

    Maven是构建Java应用的优秀管理工具，是Apache软件基金会的顶级项目。其具有开发目录构建和管理，统一管理第三方依赖，编译，测试，打包，发布等功能。最大的特点要数第三方依赖管理了，因为其它功能都能从Ant中找到踪影。以往每个项目独立管理依赖，使得每个项目打包后都是巨大无比的，而真正的程序文件却没有多少，Maven接管第三方依赖后，我们的应用就只有程序文件了，没有了jar包轻巧很多，而统一的管理，使得我们可以随意获取jar资源。   
    初学者使用Maven总是感觉很难上手，其中也有一部分人是被Maven下载资源那疯狂的做法吓退的，这就是Maven的精巧之处。从Apache官方网站下载的Maven工具仅仅2M大小，其中是不包含我们要使用的依赖的，那么就要从网络中获取这些依赖，就是那个疯狂下载的过程。下面我们从头开始构建Maven环境，直到运行Jetty容器执行Web应用，这一切都很简单。   
    下载Maven的开发包之后，解压到一个固定的文件夹，建议不要使用汉字并在磁盘的根目录下，这是比较好的实践。那么本文以Maven 3为例来说明，Maven的位置是F:\Maven 3目录。解压后就要添加系统环境变量，这里要说明，因为Maven也是Java应用，所以JDK就必须要提前安装并设置环境遍历。将JAVA\_HOME和M3\_HOME单独设置，防止其使用时不能用Path变量下发现所需路径。下面是JAVA\_HOME和M3\_HOME的配置位置，这个因人而异，但是最好固定不要随意修改。   
   
   
    在Path中，这样来添加即可，非常简单。   
   
    至此Maven已经安装完成了，下面我们来验证安装，在CMD中输入mvn –v   
   
    可以看到，已经识别mvn命令了。做点简单的设置，就是第三方依赖下载的时候的存放位置，默认是在C盘用户目录的当前用户文件夹下（如C:\用户\Admin，这里是Windows 7的命名方式），在这里Maven会创建.m2文件夹，这里面就会存放下载的类库。当然我不想这些类库放在C盘下，不是特别安全，我想让它们存放在Maven安装目录下，那么就可以这样进行：在Maven安装目录的conf下打开settings.xml，这是对Maven进行设置的XML文件，这里我们简单设置第三方依赖的位置，如下设置：

**Xml代码  [收藏代码](javascript:void())**

1. **<localRepository>**F:/Maven3/repository**</localRepository>**

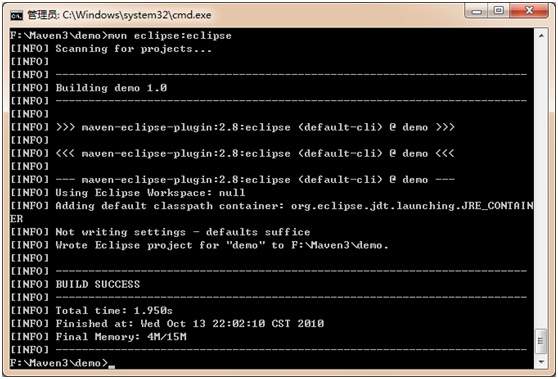
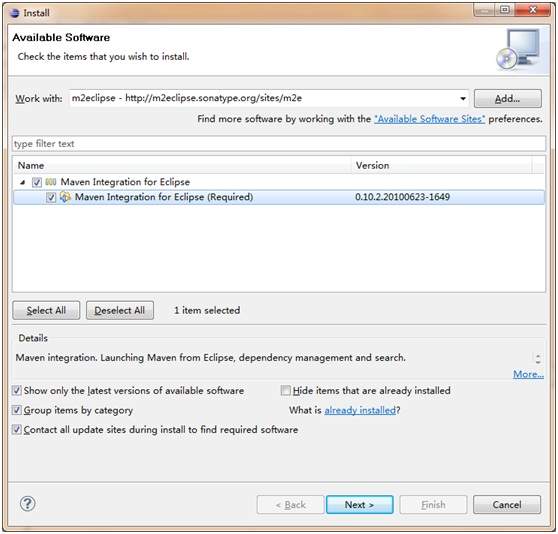
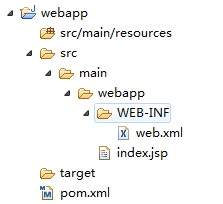
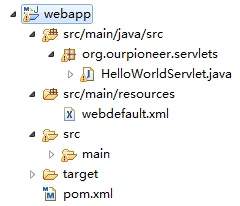
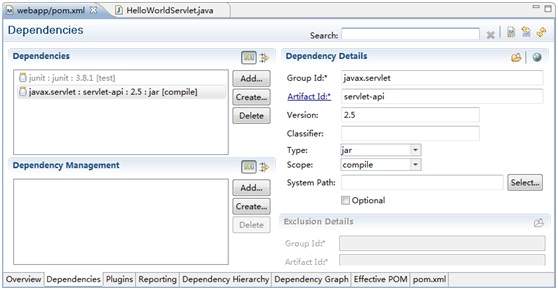
    这样，那些疯狂下载的依赖就在这个位置下了，要留给足够大的空间，因为依赖下载很可能占用很大的一块空间。为什么？下载后去看看它的真实面目就知道了。   
    下面我们就来创建应用目录，使用mvn archetype:generate命令一步一步根据提示，选择项目类型，添加groupId和artifactId就可以创建项目了，我们来看一下，这里执行mvn archetype:generate -DarchetypeCatalog=internal命令，要手工设置一下项目类型分类属性，可能因为类型太多导致屏幕显示不全，那么我们就简单来看几个常用的就行，就是internal类型。执行该命令后，在控制台我们看到有如下的内容。（切记这个操作位置就是生成项目目录的位置）   
   
    这里我们先选择16是maven-archetype-quickstart，就是基本的JavaSE项目结构的构建，19是maven-archetype-webapp，用来创建简单的Java Web应用，我们先选择16来看看。   
   
    选择16之后，Maven提示输出groupId，就是我们的项目命名组成之一，没什么过多解释的东西，artifactId用来标记项目名称，它们两个是联合使用的。而package定义的是包名，我们在文件夹内就发现了demo命名的项目文件夹。看看这个目录的结构，这是Maven为我们创建的，就是一个标准的格式：   
   
    根目录下有一个pom.xml，这就是项目的Maven配置文件，暂时不介绍其强大的功能，仅仅简单看一下实际应用，这是基础的目录结构，Maven为我们创建了一个Hello World的小程序，我们来运行它。想要运行这个程序，那么就要编译，mvn的编译命令是mvn compile，这样就为我们编译整个项目，这个操作要在demo目录下完成。   
   
    回头看看demo文件夹，多出一个target目录，这就是我们的类路径了，class文件就在下面。这个操作的逆过程是mvn clean，顾名思义，就是清理，清理生成的可执行部分，仅留下源码部分，如下：   
   
    Maven执行了Deleting操作，删除了target目录。如果想执行程序，是需要class文件或jar文件的，前面的mvn compile命令仅仅是编译，而没有对应用进行打包发布，而mvn package命令可以将应用打包（jar或war等），那么我们就来为项目打包发布：   
   
    这个过程，Maven执行了很多操作，有日志可以细看，其中还有测试过程，这是JUnit的测试。其实我们创建项目时，Maven为我们引入的第一个依赖就是JUnit。下面我们来执行jar文件，运行程序。（当然执行class是一样的效果）   
   
    这样就得到执行效果了，Hello World!   
（未完待续，下一篇将结合Eclipse以图片方式来说明Web项目的配置和运行）   
    下一篇地址：[第二部分](http://sarin.iteye.com/blog/784697)

[**用Maven构建Java Web开发环境（Jetty容器）之二**](http://sarin.iteye.com/blog/784697)

**博客分类：**

* [积少成多](http://sarin.iteye.com/category/93571)

[maven](http://www.iteye.com/blogs/tag/maven)[Web](http://www.iteye.com/blogs/tag/Web)[Java](http://www.iteye.com/blogs/tag/Java)[Eclipse](http://www.iteye.com/blogs/tag/Eclipse)[Servlet](http://www.iteye.com/blogs/tag/Servlet)

    本文接上一篇[第一部分](http://sarin.iteye.com/blog/784275)继续来介绍。   
    目前为止我们还是手工命令行方式执行程序的，没有和IDE结合，其实Maven天生就对Eclipse做了集成，我们使用mvn eclipse:eclipse就得到了一个Eclipse的项目结构，在Eclipse中使用import功能就能直接导入到IDE中了。我们来看一下这个过程：   
   
    此时的demo就是Eclipse项目格式的了，出现了.project和.classpath文件。我们在Eclipse中引入这个项目，此时的Eclipse没有安装Maven插件，不能自动运行Maven命令，我们来安装Maven的Eclipse插件M2E。   
   
    在Eclipse的Install New Software中直接选择安装即可，非常简单。下面我们来创建Web项目并导入Eclipse中，在Jetty容器中运行程序。首先执行mvn archetype:generate命令创建。   
   
    可以看到，刚创建的web项目结构包含了resources目录，而没有java代码目录，我们需要手工创建，在Eclipse中创建source folder，路径为src/main/java/src，现在我们得到如下一个项目结构，新建一个Servlet用于测试。   
   
    此时，项目中没有Servlet的依赖，需要添加，我们使用m2eclipse插件来直接添加依赖，如下所示：   
   
    相应的XML为：

**Xml代码  [收藏代码](javascript:void())**

1. **<dependency>**
2. **<groupId>**javax.servlet**</groupId>**
3. **<artifactId>**servlet-api**</artifactId>**
4. **<version>**2.5**</version>**
5. **<type>**jar**</type>**
6. **<scope>**compile**</scope>**
7. **</dependency>**

    下面就可以编写Servlet了，很简单，就输出HelloWorld吧。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.ourpioneer.servlets;
3. **import** java.io.IOException;
4. **import** java.io.PrintWriter;
6. **import** javax.servlet.ServletException;
7. **import** javax.servlet.http.HttpServlet;
8. **import** javax.servlet.http.HttpServletRequest;
9. **import** javax.servlet.http.HttpServletResponse;
11. **public** **class** HelloWorldServlet **extends** HttpServlet {
13. **protected** **void** doGet(HttpServletRequest request,
14. HttpServletResponse response) **throws** ServletException, IOException {
15. **this**.process(request, response);
16. }
18. **protected** **void** doPost(HttpServletRequest request,
19. HttpServletResponse response) **throws** ServletException, IOException {
20. **this**.process(request, response);
21. }
23. **private** **void** process(HttpServletRequest request,
24. HttpServletResponse response) **throws** ServletException, IOException {
25. response.setContentType("text/html;charset=utf-8");
26. response.setCharacterEncoding("utf-8");
27. PrintWriter out = response.getWriter();
28. String title="Webapp Demo";
29. out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">");
30. out.println("<html xmlns=\"http://www.w3.org/1999/xhtml\">");
31. out.println("<head>");
32. out.println("<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8\" />");
33. out.println("<title>" + title + "</title>");
34. out.println("<body>");
35. out.println("<h1>Hello World!</h1>");
36. out.println("</body>");
37. out.println("</html>");
38. }
39. }

    然后不能忘了在web.xml中配置这个Servlet，这里是Servlet 2.5的规范，不是Servlet 3，不能用注解。这也很简单。

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"**>**
7. **<display-name>**Archetype Created Web Application**</display-name>**
9. **<servlet>**
10. **<servlet-name>**helloworld**</servlet-name>**
11. **<servlet-class>**org.ourpioneer.servlets.HelloWorldServlet**</servlet-class>**
12. **</servlet>**
14. **<servlet-mapping>**
15. **<servlet-name>**helloworld**</servlet-name>**
16. **<url-pattern>**/helloworld**</url-pattern>**
17. **</servlet-mapping>**
18. **</web-app>**

    程序都有了，剩下就是运行了，Maven既然天生和Jetty是一对儿，这里我们就使用Jetty吧，在Maven中配置Jetty，首先是webdefault.xml要准备好，它是配置Jetty的，这个可以从Jetty的包中找到，并复制到resources下，这里多说一点，默认Jetty运行时是锁定JS/CSS等静态文件的，如果想在Jetty运行时也能修改它们，要在webdefault.xml中修改如下设置：

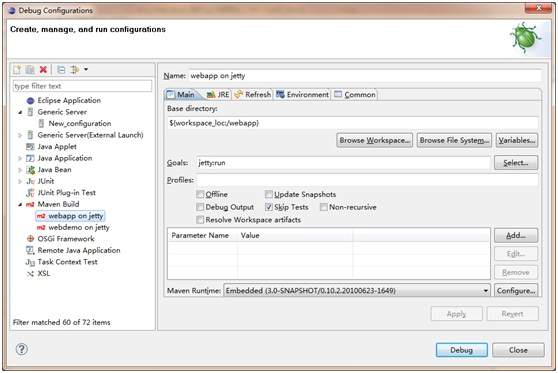
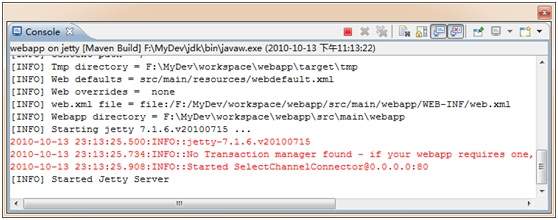
**Xml代码  [收藏代码](javascript:void())**

1. **<init-param>**
2. **<param-name>**useFileMappedBuffer**</param-name>**
3. **<param-value>**false**</param-value>**
4. **</init-param>**

    Jetty也准备了，运行命令是jetty:run，这要在Maven中设置，那么需要在pom.xml中加入Jetty的插件的设置信息。这里直接贴出其整体构建信息。

**Xml代码  [收藏代码](javascript:void())**

1. **<build>**
2. **<finalName>**webapp**</finalName>**
3. **<sourceDirectory>**src/main/java/src**</sourceDirectory>**
4. **<testSourceDirectory>**src/test**</testSourceDirectory>**
5. **<plugins>**
6. **<plugin>**
7. **<groupId>**org.apache.maven.plugins**</groupId>**
8. **<artifactId>**maven-compiler-plugin**</artifactId>**
9. **<version>**2.0.2**</version>**
10. **<configuration>**
11. **<source>**1.6**</source>**
12. **<target>**1.6**</target>**
13. **<encoding>**utf-8**</encoding>**
14. **</configuration>**
15. **</plugin>**
16. **<plugin>**
17. **<groupId>**org.apache.maven.plugins**</groupId>**
18. **<artifactId>**maven-resources-plugin**</artifactId>**
19. **<configuration>**
20. **<encoding>**UTF-8**</encoding>**
21. **</configuration>**
22. **</plugin>**
23. **<plugin>**
24. **<groupId>**org.mortbay.jetty**</groupId>**
25. **<artifactId>**jetty-maven-plugin**</artifactId>**
26. **<version>**7.1.6.v20100715**</version>**
27. **<configuration>**
28. **<stopKey>**stop**</stopKey>**
29. **<stopPort>**5599**</stopPort>**
30. **<webAppConfig>**
31. **<contextPath>**/**</contextPath>**
32. **<defaultsDescriptor>**src/main/resources/webdefault.xml**</defaultsDescriptor>**
33. **</webAppConfig>**
34. **<scanIntervalSeconds>**0**</scanIntervalSeconds>**
35. **<connectors>**
36. **<connector** implementation="org.eclipse.jetty.server.nio.SelectChannelConnector"**>**
37. **<port>**80**</port>**
38. **<maxIdleTime>**60000**</maxIdleTime>**
39. **</connector>**
40. **</connectors>**
41. **</configuration>**
42. **</plugin>**
43. **<plugin>**
44. **<groupId>**org.apache.maven.plugins**</groupId>**
45. **<artifactId>**maven-eclipse-plugin**</artifactId>**
46. **<version>**2.7**</version>**
47. **<configuration>**
48. **<addVersionToProjectName>**false**</addVersionToProjectName>**
49. **<useProjectReferences>**false**</useProjectReferences>**
50. **<encoding>**UTF-8**</encoding>**
51. **<wtpmanifest>**false**</wtpmanifest>**
52. **<wtpapplicationxml>**true**</wtpapplicationxml>**
53. **<wtpversion>**1.5**</wtpversion>**
54. **<additionalBuildcommands>**
55. **<buildcommand>**org.eclipse.jdt.core.javabuilder**</buildcommand>**
56. **<buildcommand>**org.eclipse.wst.common.project.facet.core.builder**</buildcommand>**
57. **<buildcommand>**org.eclipse.wst.validation.validationbuilder**</buildcommand>**
58. **</additionalBuildcommands>**
60. **<additionalProjectnatures>**
61. **<nature>**org.springframework.ide.eclipse.core.springnature**</nature>**
62. **<nature>**org.maven.ide.eclipse.maven2Nature**</nature>**
63. **<nature>**org.eclipse.wst.common.project.facet.core.nature**</nature>**
64. **<nature>**org.eclipse.jdt.core.javanature**</nature>**
65. **<nature>**org.eclipse.wst.common.modulecore.ModuleCoreNature**</nature>**
66. **</additionalProjectnatures>**
67. **<classpathContainers>**
68. **<classpathContainer>**org.eclipse.jdt.launching.JRE\_CONTAINER**</classpathContainer>**
69. **</classpathContainers>**
70. **</configuration>**
71. **</plugin>**
72. **<plugin>**
73. **<groupId>**org.apache.maven.plugins**</groupId>**
74. **<artifactId>**maven-war-plugin**</artifactId>**
75. **<version>**2.1-beta-1**</version>**
76. **<configuration>**
77. **<warName>**webapp**</warName>**
78. **</configuration>**
79. **</plugin>**
80. **</plugins>**
81. **</build>**

    此时，更新一下Maven依赖，它们就都自动下载到本地了，到这个过程结束，我们就可以在Eclipse中配置Debug运行了。配置很简单，如下。   
   
    这是Debug模式运行，Run模式下是一样的，用Debug模式可以在Eclipse中断点运行程序，非常便于调试。下面我们就让它跑起来吧。运行命令是jetty:run，Base directory配置是：${workspace\_loc:/应用名}，启动调试，看到如下信息，Jetty就成功启动了。   
   
    这里我们使用了80端口，配置方式在pom.xml中，上面的代码已经体现了。在浏览器中访问地址如下：http://localhost/helloworld，之后，我们就看到了效果。   
   
    本文系作者本人的实践和探索，希望对使用者有用，欢迎交流。