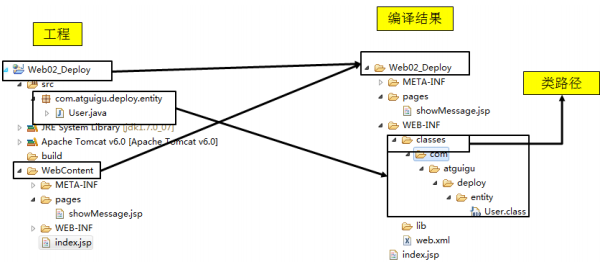
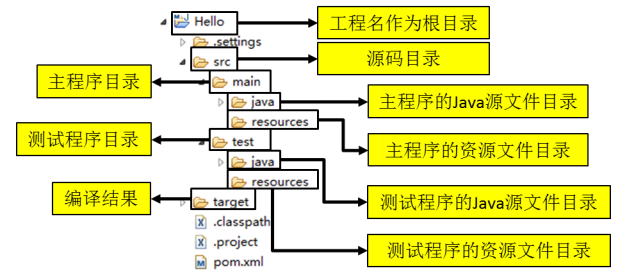
1. Web工程及其编译结果对比
   1. 
   2. WebContent为主，其下的WEB-INF里的classes为源码
2. 构建过程的主要环节：
   1. 清理：删除以前的编译结果，为重新编译做好准备。
   2. 编译：将 Java 源程序编译为字节码文件。
   3. 测试：针对项目中的关键点进行测试，确保项目在迭代开发过程中关键点的正确性。
   4. 报告：在每一次测试后以标准的格式记录和展示测试结果。
   5. 打包：将一个包含诸多文件的工程封装为一个压缩文件用于安装或部署。Java 工程对应 jar 包，Web 工程对应 war 包。
   6. 安装：在 Maven 环境下特指将打包的结果——jar 包或 war 包安装到本地仓库中。
   7. 部署：将打包的结果部署到远程仓库或将 war 包部署到服务器上运行。
3. 约定的目录结构：
   1. 
4. POM：Project Object Model，项目对象模型。将 Java 工程的相关信息封装为对象作为便于操作和管理的模型。
5. 坐标：
   1. groupid：公司或组织的域名倒序+当前项目名称
   2. artifactId：当前项目的模块名称
   3. version：当前模块的版本
   4. 将gav三个向量连接即可在仓库找到jar包
   5. 自己的maven工程需要执行mvn install安装进厂库
6. 依赖：A jar包用到B jar包即为依赖，使用标签dependency指定依赖

<dependency>

<groupId>com.atguigu.maven</groupId>

<artifactId>Hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>compile</scope>

</dependency>

* 1. 依赖范围使用标签scope标注，可选值：compile、test、provide

|  |  |  |  |
| --- | --- | --- | --- |
|  | Compile | Test | Provide |
| 主程序 | √ | × | √ |
| 测试 | √ | √ | √ |
| 参与部署 | √ | × | × |

* + 1. 依赖有效性总结
    2. 依赖传递性总结，A依赖B，B依赖C，A能否使用C？得看B依赖C的范围是不是Compile。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Maven工程 | | | 依赖范围 | 对A可见性 |
| A | B | C | Compile | √ |
| D | Test | × |
| E | Provide | × |

* 1. 排除依赖，使用标签exclusions，exclusion
  2. 统一管理所依赖的jar包版本，使用标签properties，自定义一个标签，在所依赖的jar包的标签version出使用${自定义标签}
  3. 依赖原则：
     1. 路径最短优先
     2. 路径相同时，先声明者优先

1. 仓库：本地仓库~/.m2，中央仓库，私服（局域网仓库）
2. 生命周期：Maven有三套互相独立的生命周期，互相独立，可以单独调用
   * 1. Clean Lifecycle：在进行真正的构建之前进行一些清理工作。
     2. Default Lifecycle：构建的核心部分，编译，测试，打包，安装，部署等等。
     3. Site Lifecycle：生成项目报告，站点，发布站点。
   1. 每套生命周期都由一组阶段组成，平时输入的例如mvn clean对应于一个特定的阶段，clean是Clean生命周期的一个阶段。有Clean生命周期，也有clean阶段。
   2. Clean生命周期：
      1. pre-clean 执行一些需要在 clean 之前完成的工作
      2. clean 移除所有上一次构建生成的文件
      3. post-clean 执行一些需要在 clean 之后立刻完成的工作
   3. Site生命周期：常用site，site-deploy
      1. pre-site 执行一些需要在生成站点文档之前完成的工作
      2. site 生成项目的站点文档
      3. post-site 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
      4. site-deploy 将生成的站点文档部署到特定的服务器上
   4. Default生命周期：
      1. validate
      2. generate-sources
      3. process-sources
      4. generate-resources
      5. process-resources 复制并处理资源文件，至目标目录，准备打包。
      6. compile 编译项目的源代码。
      7. process-classes
      8. generate-test-sources
      9. process-test-sources
      10. generate-test-resources
      11. process-test-resources 复制并处理资源文件，至目标测试目录。
      12. test-compile 编译测试源代码。
      13. process-test-classes
      14. test 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
      15. prepare-package
      16. package 接受编译好的代码，打包成可发布的格式，如 JAR。
      17. pre-integration-test
      18. integration-test
      19. post-integration-test
      20. verify
      21. install 将包安装至本地仓库，以让其它项目依赖。
      22. deploy 将最终的包复制到远程的仓库，以让其它开发人员与项目共享或部署到服务器上运行。
3. 生命周期与自动化构建：运行任何一个阶段的时候，他前面的所有阶段都会被运行。
4. 插件和目标：
   1. Maven 的核心仅仅定义了抽象的生命周期，具体的任务都是交由插件完成的。
   2. 每个插件都能实现多个功能，每个功能就是一个插件目标。
   3. Maven 的生命周期与插件目标相互绑定，以完成某个具体的构建任务。
   4. 例如：compile 就是插件 maven-compiler-plugin 的一个目标；pre-clean 是插件 maven-clean-plugin 的一个目标。
5. 继承：由于非compile范围的依赖信息不能在依赖链中船体，所以需要的工程只能单独配置，因此可以将这样的依赖信息同意提取到父工程模块中统一管理。以下为操作步骤。
   1. 创建父工程，打包方式处设置为pom
   2. 在子工程中引用父工程

<parent>

<!-- 父工程坐标 -->

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

<relativePath>从当前目录到父项目的 pom.xml 文件的相对路</relativePath>

</parent>

* + 1. 如果子工程的gv和父工程的重复，可以删除
  1. 在父工程中管理依赖：
     1. 使用标签dependencyManagement将标签dependencies和标签dependency包括。
  2. 子项目中重新指定需要依赖，删除版本号和范围。

1. 聚合：将多个工程拆分为模块后，需要手动逐个安装到仓库后依赖才能生效，修改后也需要手动clean操作，聚合后可以批量进行。
   1. 配置聚合：在总的聚合工程中使用标签modules/标签module组合，指定模块构成的相对路径即可。