maven全解

- maven pom
  - <distributionManagement>
    - 描述
      - 负责管理构件的发布
    - 子标签
      - repository
        - 描述
          - mvn deploy时，如果是正式版，会发布到的远程仓库的地址
        - 示例
          - <id>releases</id>
            <name>Nexus Release Repository</name>

            <url>http://nexus.abc.com/repository/maven-releases/</url>
      - downloadUrl
      - status
        - 当前Maven项目的状态，可用的状态如下所示。注意，该值是由Maven自动设置，永远不要人工设置。
      - snapshotRepository
        - mvn deploy时，如果jar包名以-SNAPSHOT结尾，则会推送到这个快照远程仓库地址
        - <id>snapshots</id>
          <name>Nexus Snapshot Repository</name>

          <url>http://nexus.fenqi.im:8081/repository/maven-snapshots/</url>
      - site
      - relocation
  - <dependencies>
    - 描述
      - 项目所需依赖的集合
    - 子标签
      - <dependency>
        - 描述
          - 项目所需的具体依赖
        - 子标签
          - <scope>
            - 描述

- 作用域
  - 可选值
    - system
      - 范围。该依赖与三种classpath的关系，和provided依赖范围完全一致。但是，使用system范围的依赖时必须通过systemPath元素显式地指定依赖文件的路径。由于此类依赖不是通过Maven仓库解析的，而且往往与本机系统绑定，可能造成构建的不可移植，因此应该谨慎使用。
    - test
      - 测试classpath可用
      - 例子
        - JUnit
    - runtime
      - 测试、运行时有效。编译时无效。
      - 例子
        - JDBC
          - 典型的例子是JDBC驱动实现，项目主代码的编译只需要JDK提供的JDBC接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体JDBC驱动。
    - compile
      - 默认值：编译，测试，运行三种classpath都可用
      - 例子
        - spring-core
    - provided
      - 编译，测试两种classpath有效，运行classpath不可用
      - 例子
        - servlet-api
          - 典型的例子是servlet-api，编译和测试项目的时候需要该依赖，但在运行项目的时候，由于容器已经提供，就不需要Maven重复地引入一遍。
      - provide='提供'，provided表示已经提供，由项目运行的容器提供，不需要打包的运行时classPath
    - import
      - 该依赖范围不会对三种classpath产生实际的影响
- <type>

- 相同jar包重复引入
  - 优先级/选择顺序
    - 第一原则
      - 路径最短原则
        - 1：A>B>C>X

- 2：D>C>X
- 选择2
- 第二原则
  - 先声明原则
    - 在依赖路径长度相等的前提下，在POM中依赖声明的顺序决定了谁会被解析使用，顺序最靠前的那个依赖优胜。
- 指定某个JAR包

- <pluginRepositories>
  - 描述
    - 配置远程插件仓库地址，一般情况下不用配置，默认的使用中央插件仓库地址：
  - 子标签
    - <pluginRepository>

- <build>
  - 描述
    - 子主题
  - 子标签
    - <pluginManagement>
      - 描述
        - 声明使用的plugin的属性。这里不会真正的引入该插件，只有在<plugins>里面引入的插件才会生效，如果<plugins>里面没有配置属性，则默认用这里配置的属性，如果<plugins>配置了属性则用配置的。
      - 子标签
        - <plugins>
          - 描述
            - 需要声明配置的插件集合
          - 子标签
            - <plugin>
              - 描述
                - 需要声明配置的具体标签
    - <plugins>
      - 描述
        - 需要引入的标签
      - 子标签
        - <plugin>
          - 描述
            - 需要引入的具体标签
            - 建议手动添加maven-compiler-plugin插件，并且配置jdk版本，避免不同环境的maven默认jdk不一致
          - 子标签
            - <version>

- 插件版本号，这里声明了插件版本号，如果在<pluginManagement>里面也声明了不一样的插件版本号，{以谁为准？}， 如果都不声明插件版本号，则会去本地插件仓库和声明的远程插件仓库获取最新的版本。{如果没有配置远程插件仓库，则会去默认的中央远程插件仓库获取，如果只配置了第三方远程插件仓库，是否也会去默认的中央远程插件仓库获取？}
- 推荐显示设置版本号
  - maven3默认解析最新的非快照版本插件,如果不设置版本号，那么该插件有新正式版发布的时候会被项目引用，仍然可能导致构建失败。
  - maven内置插件都设置了版本号
- <executions>
  - 描述
  - 子标签
    - <execution>
      - 描述
      - 子标签
        - <id>
          - ${TODO}
        - <goals>
          - 要执行的目标，目标必须已在插件里面定义
        - <phase>
          - 绑定到的生命周期
        - <configuration>
          - 配置供插件使用的变量
        - <inherited>
        - <project>

- <resources>
  - 描述
    - 标记哪些是资源文件，将项目主资源文件复制到主代码编译输出目录中
  - 子标签
    - <resource>
      - <directory>
        - 表示资源文件所在目录，相对根目录： 首位不能有/ ,末位用不用/都一样
      - <targetPath>
        - 表示资源文件打包后放在哪个目录，相对打包后的根目录： 首尾有没有/都一样
      - <includes>
        - <include>

- 需要打包成资源的文件路径，相对<directory>。 首位不能有/ 。
  如果有此标签，则只会打包<directory>路径下该子标签路径对应的文件。其他文件不会打包。
  可用*通配符表示多个文件或者文件夹。
- <excludes>
  - <exclude>
    - <directory>路径下需要被排除的文件的路径。 首位不能有/。
      可用*通配符。
      如果<includes>包含了某文件，而<excludes>也包含了该文件，则该文件会被排除。
      如果没有<includes>标签，则会打包除了<excludes>下文件的所有文件。
- <filtering>
  - 值为 true 或者 false: true表示会将资源文件里面占位符替换为对应的变量值

- <profiles>
  - 描述
    - 配置<profile>
  - 子标签
    - <profile>
      - 描述
        - 这里面可以配置各种标签，并且当该<profile>激活时，这里配置的标签会覆盖外面配置的相同标签。 比如配置<properties>会覆盖外面配置的<properties>
        - 这个属性有什么实用性： 1、可以根据不同环境配置不同的属性，比如数据库配置，生产环境和开发环境配置的属性不同。
          2、打包出的jar包名后缀根据不同环境配置，实现生产环境和开发环境jar包的隔离，不至于开发环境生成的包，被生产环境引用，导致jar包混乱。
        - 可以同时激活多个<profile>，多个用逗号隔开，逗号前后不能有空格。 如果多个<profile>有同名的配置，以后面的优先。
        - 激活方式： 1、通过命令激活
          2、通过settings激活
          以及下面<activation>的4个方式激活
        - profile种类：
          - pom.xml：很显然，pom.xml中声明的profile只对当前项目有效。· 用户settings.xml：用户目录下.m2/settings.xml中的profile对本机上该用户所有的Maven项目有效。·
            全局settings.xml：Maven安装目录下conf/settings.xml中的profile对本机上所有的Maven项目有效。
      - 子标签
        - <activation>

- 描述
  - 表示该<profile>的激活条件
- 子标签
  - 可选条件1：<activeByDefault>
    - 表示是否自动激活：true表示激活，false表示不激活
  - 可选条件2：<file>
    - <exists>
      - 只能存在一个<exists>，如果存在该路径配置的文件，则激活
    - <missing>
      - 只能存在一个<missing>，如果不存在该路径配置的文件，则激活
  - 可选条件3：<os>
    - 描述
      - 这里family的值包括Windows、UNIX和Mac等，而其他几项name、arch、version，用户可以通过查看环境中的系统属性os.name、os.arch、os.version获得。
    - 子标签
      - <os> <name></name>
        <version></version>
        <arch></arch>
        <family></family>
        </os>
  - 可选条件4：<property>
    - 描述
      - 当某系统属性存在的时候，自动激活profile
    - 子标签
      - <property> <name></name>
        <value></value>
        </property>

- <dependencyManagement>
  - 描述
    - 声明依赖信息，可被子项目继承。 这里面声明的依赖在当前项目不会被真正的引入，只有在<dependencies>里面声明的依赖才会引入。而且<dependencies>可以不用配置版本号和作用范围，默认使用这里配置的版本号和作用范围。
      在子项目也不会自动引入，需要子项目通过<dependency>显示引入,
      子项目显示引入时会继承父项目中声明的版本和作用范围，
      如果子项目显示声明了版本号作用范围，则以子项目的声明为准
    - 推荐在父项目里面使用此标签，而不是直接使用<dependencies>标签。 因为直接使用<dependencies>标签，所有的子项目都会继承<dependencies>里面的依赖；
      而使用此标签，子项目只需要声明依赖的groupId和articleId,即可引入相关依赖，不声明的

- 话，不会自动引入，而且所有子项目引入的依赖版本号和作用范围都一致，避免了版本冲突，减少了重复代码，减少了不必要的依赖。
- 子标签
  - <dependencies>
    - 描述
      - 要声明的依赖集合
      - 子标签
        - <dependency>
- <modules>
  - 描述
    - 用于声明模块，不可被子项目继承
  - 子标签
    - <module>
      - 模块
      - 值就是模块相对当前项目的路径
- <parent>
  - 描述
    - 设置父项目坐标和相对本项目的路径，不可被子项目继承
  - 子标签
    - <relativePath>
      - 描述：值就是相对于当前项目的相对路径，一般不用设置，默认当前项目的上级目录
    - <artifactId> <groupId>
      <version>
      - 父项目的坐标
- <properties>
  - 描述
    - 用于定义maven属性，可以在pom的任何地方通过${}的方式使用。为什么直接写在需要使用的地方？因为这样做可以消除重复，方便维护。有越多的地方使用该属性，意义越大。可以被子项目继承。
  - 子标签
    - 任意字符串可作为子标签，子标签名就是${}里面的名称
- 所有maven项目都继承了一个maven自带的超级pom
  - 超级pom在哪？：{mavenHome}/lib/maven-model-builder-{version}.jar里面的org\apache\maven\model\pom-4.0.0.xml
  - 超级pom有什么用？：定义了大量的默认配置

- maven 属性
  - 内置属性
    - ${basedir}
      - 项目根目录，即包含pom.xml文件的目录
    - ${version}

- 项目版本号
- pom属性
  - 描述
    - 用户可以使用该类属性引用POM文件中对应元素的值。 例如 ${project.artifactId}就对应了<project><artifactId>元素的值。
      这些属性都对应了一个POM元素，它们中一些属性的默认值都是在超级POM中定义的。
  - 常用的POM属性
    - ${project.build.sourceDirectory}
      - 项目的主源码目录，默认为src/main/java/。
    - ${project.build.testSourceDirectory}
      - 项目的测试源码目录，默认为src/test/java/。
    - ${project.build.directory}
      - 项目构建输出目录，默认为target/。
    - ${project.outputDirectory}
      - 项目测试代码编译输出目录，默认为target/test-classes/。
    - ${project.groupId}
      - 项目的groupId。
    - ${project.artifactId}
      - 项目的artifactId。
    - ${project.version}
      - 项目的version，与 ${version}等价。
    - ${project.build.finalName}
      - 项目打包输出文件的名称，默认为 ${project.artifactId}- ${project.version}。
- 自定义属性
  - 用户可以在POM的<properties>元素下自定义Maven属性。 <properties>
    <my.prop>hello</my.prop>
    </properties>
    然后在POM中其他地方使用 ${my.prop}的时候会被替换成hello。
- Settings属性
  - 与POM属性同理，用户使用以settings.开头的属性引用settings.xml文件中XML元素的值，如常用的 ${settings.localRepository}指向用户本地仓库的地址。
- Java系统属性
  - 所有Java系统属性都可以使用Maven属性引用，例如 ${user.home}指向了用户目录。用户可以使用mvn help：system查看所有的Java系统属性。
- 环境变量属性
  - 所有环境变量都可以使用以env.开头的Maven属性引用。例如 ${env.JAVA_HOME}指代了JAVA_HOME环境变量的值。 用户可以使用mvn help：system查看所有的环境变量

- plugin
  - Maven的核心仅仅定义了抽象的生命周期， 具体的任务是交由插件完成的，插件以独立的构件形式存在，

因此，Maven核心的分发包只有不到3MB的大小，

Maven会在需要的时候下载并使用插件。

每个插件都有一个或者多个目标，每个目标对应一个功能，

通用的写法是，用冒号隔开，冒号前面是插件名称，

冒号后面是插件目标。

例如：surefire:test　surefire是插件maven-surefire-plugin的前缀，

表示maven-surefire-plugin的test目标。

- 内置插件：不需要显示引入
  - maven-clean-plugin
    - 绑定clean生命周期的clean阶段
  - maven-compiler-plugin
    - 绑定default生命周期的compile阶段
  - maven-site-plugin
    - 绑定site生命周期的site阶段
  - maven-deploy-plugin
    - install
    - install-file
  - maven-install-plugin
  - maven-resources-plugin
  - maven-surefire-plugin
  - 根据pom.xml中packaging的值不同， 使用的内置插件不同
    - <packaging>jar</packaging> 或者不显示声明，默认值为jar
      - maven-jar-plugin
        - jar:jar
          - {实现原理？}
        - jar:test-jar
    - <packaging>war</packaging>
      - maven-war-plugin
- 其他插件
  - maven-surefire-plugin
    - 目标：运行应用程序的单元测试
  - maven-dependency-plugin

- 详细的插件列表，官方地址： http://maven.apache.org/plugins/index.html
- 官方插件下载地址： http://repo1.maven.org/maven2/org/apache/maven/plugins/
  - 需要手动下载吗？不需要
- 插件前缀
  - 每个插件可以定义插件前缀，方便通过mvn命令直接调用该插件，{多个插件的前缀可以一样？}
  - 插件前缀可以在元数据里面看： 元数据在哪：{groupId}/maven-metadata-central.xml
  - {插件前缀如何定义？}
- 插件仓库

- 本地仓库
  - 就是本地的jar包仓库
- 远程仓库
  - 默认：中央仓库
    - http://repo1.maven.org/maven2/org/apache/maven/plugins/
  - 不同公司有自己的内部仓库

- 编写maven插件
  - 步骤1：创建maven-plugin项目
    - pom加入依赖： <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>{version}</version>
      </dependency>
      - 目前最新版本： 3.6.0
    - pom加入依赖，可以使用注解的方式,配置Mojo： <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
      <version>3.6.0</version>
      </dependency>
  - 步骤2：编写插件目标
    - 编写类继承AbstractMojo类
    - 每个继承AbstractMojo的类都可以有一个目标，只能有一个插件目标，有了插件目标才使用该插件
    - 例子
      - /** * @goal print
        */
        public class BrightMojo extends AbstractMojo {
        - <plugin> <groupId>wishes.maven.plugins</groupId>
          <artifactId>bright-plugin</artifactId>
          <version>1.0-SNAPSHOT</version>
          <configuration>
          <proName>wishes.bright</proName>
          </configuration>
          <executions>
          <execution>
          <goals>
          <goal>print</goal>
          </goals>
          <phase>package</phase>
          </execution>

</executions>
</plugin>

- 步骤3：为目标提供配置点
  - 几个特定的注解，这些注解所在的属性在上一步的类里面
    - @parameter
      - 设置了一个可供使用插件方自由配置的参数，参数名称 为字段名。 可通过 expression 设置从系统属性中读取值。
        设置了expression的情况下，如果既设置<properties>环境变量，也设置plugin的插件变量，
        则以插件变量为准。
        - /** * @parameter expression ="${pn}"
          */
          private String proName="default";
          - 在使用方的pom里面配置： <properties>
            <pn>23</pn>
            </properties>
        - /** * @parameter
          */
          private String proName="default";
          - 在使用方的pom里面配置： <build>
            <plugins>
            <plugin>
            <groupId>com.wishes</groupId>
            <artifactId>hello-plugin</artifactId>
            <version>1.0-SNAPSHOT</version>
            <configuration>
            <proName>perfect</proName>
            </configuration>
            </plugin>
            </plugins>
            </build>
            - 此处配置覆盖<properties>的配置
    - @readonly
      - 表示该属性不能通过使用方配置，即使配置也不生效。
        - /** * @readonly
          */
          private String[] proNames={"1","2"};
      - 如果和@parameter一起使用，使用方配置的属性会生效。相当于@readonly无效。
    - @required
      - 该注解表示属性必须配置，如果没有配置则在执行插件目标的时候会报错。

- 步骤4：编写代码实现目标行为
- 步骤5：错误处理及日志
- 步骤6：测试插件

- 生命周期
  - 三大生命周期互相独立。 但是在某一生命周期内，各个阶段有先后顺序，
    执行某个阶段的时候，会先依次执行他前面的阶段，
    可以通过命令跳过某些阶段。{什么命令？}
    - clean
      - clean生命周期的目的是清理项目,它包含三个阶段
        - pre-clean
          - 执行一些清理前需要完成的工作{清理什么？}
        - clean
          - 清理上一次构建生成的文体
        - post-clean
          - 执行一些清理后需要完成的工作
    - default
      - default生命周期定义了真正构建时所需要执行的所有步骤，它是所有生命周期中最核心的部分
        - validate
        - initialize
        - generate-sources
        - process-sources
          - 处理项目主资源文件。一般来说，是对src/main/resources目录的内容进行变量替换等工作后，复制到项目输出的主classpath目录中。
        - generate-resources
        - process-resources
        - compile
          - 编译项目的主源码。一般来说，是编译src/main/java目录下的Java文件至项目输出的主classpath目录中。
        - process-classes
        - generate-test-sources
        - process-test-sources
          - 处理项目测试资源文件。一般来说，是对src/test/resources目录的内容进行变量替换等工作后，复制到项目输出的测试classpath目录中。
        - generate-test-resources
        - process-test-resources
        - test-compile
          - 编译项目的测试代码。一般来说，是编译src/test/java目录下的Java文件至项目输出的测试classpath目录中。
        - process-test-classes

- test
  - 使用单元测试框架运行测试，测试代码不会被打包或部署。
- prepare-package
- package
  - 接受编译好的代码，打包成可发布的格式，如JAR。
- pre-integration-test
- integration-test
- post-integration-test
- verify
- install
  - 将包安装到Maven本地仓库，供本地其他Maven项目使用。
- deploy
  - 将最终的包复制到远程仓库，供其他开发人员和Maven项目使用。

- site
  - site生命周期的目的是建立和发布项目站点，Maven能够基于POM所包含的信息， 自动生成一个友好的站点，方便团队交流和发布项目信息。
    - pre-site
      - 执行一些在生成项目站点之前需要完成的工作。
    - site
      - 生成项目站点文档。
    - post-site
      - 执行一些在生成项目站点之后需要完成的工作。
    - site-deploy
      - 将生成的项目站点发布到服务器上。

- maven mvn 指令
  - 由于Maven中主要的生命周期阶段并不多，而常用的Maven命令实际都是基于这些阶段简单组合而成的， 因此只要对Maven生命周期有一个基本的理解，读者就可以正确而熟练地使用Maven命令。
    maven命令 区分大小写，比如 mvn -P dev 不能写成 mvn -p dev
    - mvn clean
      - 清理项目构建结果：删除项目target目录
        - 该命令调用clean生命周期的clean阶段。实际执行的阶段为clean生命周期的pre-clean和clean阶段。
    - mvn validate
      - 验证工程是否正确，所有需要的资源是否可用：{?}
    - mvn compile
      - 编译项目：{? }
    - mvn test
      - 测试
        - 该命令调用default生命周期的test阶段。实际执行的阶段为default生命周期的validate、initialize等，直到test的所有阶段。

- mvn packge
  - 打包jar到target目录
- mvn install
  - 将jar包发布到本地缓存仓库
- mvn deploy
  - 将jar包发布到远程仓库
- -pl
  - --projects<arg>构建指定的模块，模块间用逗号分隔
  - mvn clean isntall -pl project-a,project-b
- -am
  - --also-make同时构建所列模块的依赖模块
  - 必须和-pl同时使用，例如：mvn clean install -pl projectA -am
- -amd
  - -also-make-dependents同时构建依赖于所列模块的模块
  - 必须和-pl同时使用，例如：mvn clean install -pl projectA -amd
- -rf
  - -resume-from<arg>就是从谁开始构建
  - 该命令不能单独使用，比如不能这样：mvn -rf projectA 可以这样写：mvn clean install -pl childrenb -am -rf childrena
    表示：构建指定的项目childrenb ，并且构建childrenb的依赖，并且从childrena开始构建，为什么从childrena开始构建？因为childrenb依赖了childrena，如果先构建childrenb则会报错。
    如果不写-rf也可以，maven会自动选择合适的顺序进行构建，写了就不能写错顺序。
- -P
  - 指定一个<profiles>下的<profile>
  - mvn clean package -Pdev
    - 表示使用<id>为dev的profile
  - 上面的命令也可以是：mvn clean package -P dev
  - 可以同时激活多个<profile>: mvn clean package -P dev,pro
    同时激活dev和pro两个<profile>。
    多个<profile>之间的逗号不能有空格。
- -D
  - 配置环境变量
  - mvn clean package -P dev,pro -D username=liujjj -Dpass=azs
- -U
  - 强制更新snapshot类型的插件或依赖库(否则maven一天只会更新一次snapshot依赖);
- -v
  - 查看maven安装信息：mvn -v
- -B
  - 以非交互的方式运行，禁用颜色输出，就是打印出来的信息没有彩色。 mvn clean package -pl user-member-api -B

- -b
  - 要使用的构建策略的ID
- -C
  - --strict-checksums 如果校验码不匹配的话,构建失败;

- maven 有哪些命令?
  - mvn -h 查询maven的命令帮助信息
    - maven命令格式： usage: mvn [options] [<goal(s)>] [<phase(s)>]
      mvn后面参数没有先后顺序
      - 例如： mvn clean install -Dmaven.test.skip=true -Dmaven.javadoc.skip=true -U
      - 例如：查询各个jar包的依赖关系 mvn dependeny:tree
        这里的dependency是插件的maven-dependency-plugin前缀
        - 为什么可以直接写前缀而不是写插件坐标， 因为写前缀简洁，
          但是不利于定位到是哪个具体插件
        - 使用前缀的命令，前缀对应的插件必须存在，否则报错。

- maven 快照版本和正式版本的区别
  - maven编译的时候，如果引用的jar包名称后缀带有-SNAPSHOT,则每次都会去maven仓库拉取最新的jar包，如果不是，则会根据版本号判断，如果pom里面引用的jar包的版本号和本地仓库里面的版本号一致，则不会去远程仓库拉取最新的jar包。所以一般在开发阶段，会把项目版本名称加上后缀-SNAPSHOT,这样执行mvn deploy 命令后会把jar包推送到远程仓库，而其他项目引入了这个jar包，每次都会拉取最新的版本，方便调试。
  - 快照版本命名规则
    - 1.0.0-SNAPSHOT=1.0.0-yyyyMMdd.HHmmss-times yyyyMMdd表示年月日
      HHmmss表示时分秒
      times表示当前快照版本是第几次生成的快照版本
  - 快照版本会保存多久
    - 不同版本的最新快照会永久保存
    - 相同版本的快照，较旧的会短期保存
  - 构建规则不同
    - 对于正式版本，如果 Maven 以前下载过指定的版本文件，比如说 data-service:1.0，Maven 将不会再从仓库下载新的可用的 1.0 文件。若要下载更新的代码，data-service 的版本需要升到1.1。 快照的情况下，每次 app-ui 团队构建他们的项目时，Maven 将自动获取最新的快照(data-service:1.0-SNAPSHOT)。

- 仓库
  - 分类
    - 本地仓库
      - 默认路径
        - windows
          - ./m2/repository/
        - linux
          - ./m2/repository/

- 远程仓库
  - 中央仓库
    - http://repo1.maven.org/maven2
  - 私服
    - 国内著名私服仓库
      - https://mvnrepository.com/
      - https://maven.aliyun.com/mvn/view
    - 公司局域网私服仓库
      - 大公司创建maven私服仓库的必要性
        - 节省外网宽带，几百人构建项目，使用外网会下载几百次，使用内网只需要从外部下载一次
        - 加速项目构建，内网下载jar包，速度快
        - 保护私有jar包，外部人员无法下载公司内部私服的jar包
        - 提高稳定性，增强控制，外网断网的情况下局域网内的私服仓库仍然可用
        - 降低中央仓库的负荷
    - 使用私服
    - 搭建私服
      - 选择
        - Nexus
      - 下载
        - https://www.sonatype.com/download-oss-sonatype
      - 例子
        - https://blog.csdn.net/a1965483733/article/details/77716347
      - 很多细节不写了，自己摸索
- jar加载顺序
  - 先从本地仓库找，本地仓库找不到再从远程仓库找。