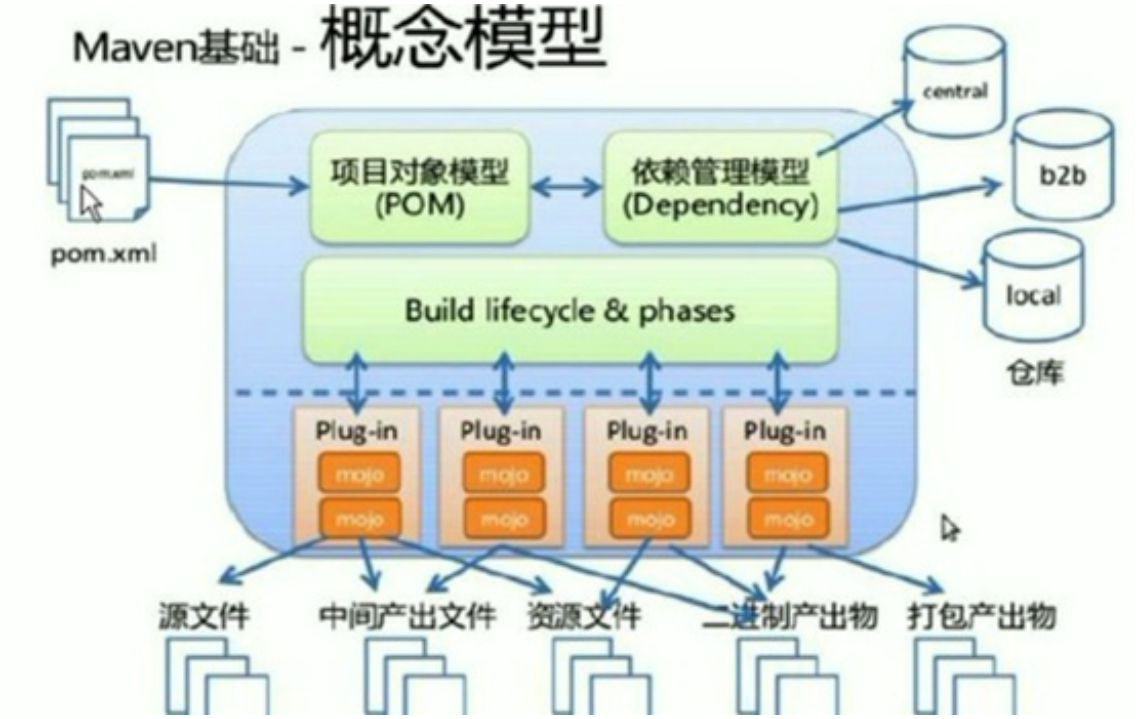
Maven介绍

1. maven概念模型

Maven 是Apache下的一个开源项目，它是一个创新的项目管理工具，它用于对Java项目进行项目构建、依赖管理及项目信息管理。

Maven 包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑



- 项目对象模型（Project Object Model）

  通过pom.xml文件定义项目的坐标、项目依赖、项目信息、插件目标、打包方式等。

- 依赖管理系统（Dependency Management System）

  通过定义项目所依赖组件的坐标有maven进行依赖管理。

比如：项目依赖spring-context, 通过在pom.xml中定义依赖即可将spring-context的jar包自动加入到工程：

pom.xml中定义依赖：

```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.2.4.RELEASE</version>
    </dependency>
```

- 一个项目生命周期（Project Lifecycle）

一个软件开发人员每天都在完成项目的生命周期：清理、编译、测试、部署，有的手工完成，有的通过Ant(项目构建工具)脚本自动完成，maven将项目生命周期抽象统一为：清理、初始化、编译、测试、报告、打包、部署、站点生成等。

Maven就是要保证一致的项目构建流程，通过执行一些简单的命令即可实现上面生命周期的各个过程。

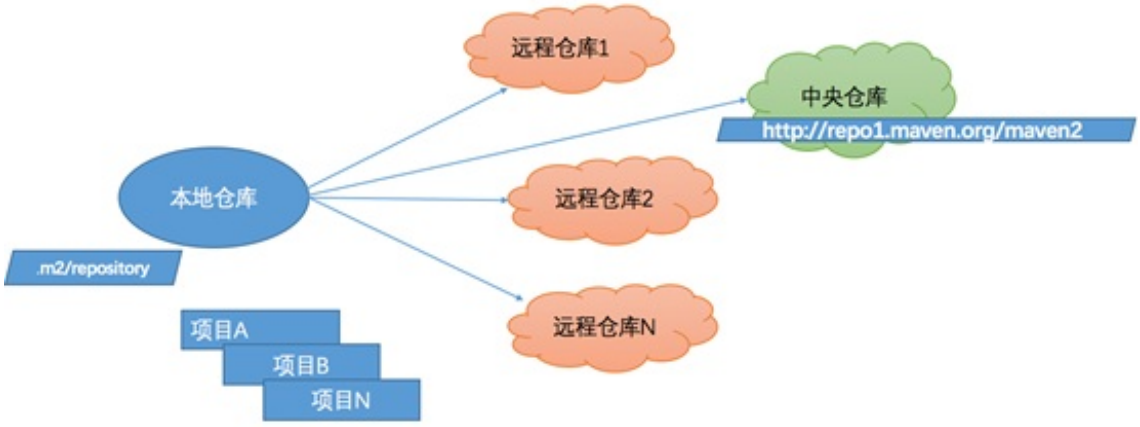
- 插件（plugin）目标（goal）

  maven管理项目生命周期过程都是基于插件完成的。

2.maven的仓库

maven工程需要配置仓库，本地的项目A项目B等通过maven从远程仓库（网络上的仓库）下载jar包并存在本地仓库，本地仓库就是本地文件夹，当第二次需要此jar包时则不需要从远程仓库下载，因为本地仓库已经存在了，可以将本地仓库理解为缓存，有了本地仓库就不用每次从远程仓库下载。

Maven仓库类型：



1. 本地仓库：用来存储远程仓库货中央仓库下载的插件和jar包，项目使用一些插件或jar包，优先从本地仓库查找。
2. 远程仓库：如果本地需要插件或者jar包，本地仓库没有，默认去远程仓库下载。
3. 中央仓库：在maven环境内部内置一个远程仓库地址http://repo1.maven.org/maven2,它是中央仓库，服务于整个互联网，它是由maven自己维护，里面有大量的常用类库，并包含该了世界上大部分流行的开源项目构件。

3.项目生命周期（了解）

Maven有三套项目独立的生命周期，请注意这里说的是“三套”，而且是“相互独立”，这三套生命周期分别是：

Clean Lifecycle 在进行真正的构建之前进行一些清理工作。

Default Lifecycle 构建的核心部分，编译、测试、打包、部署等等。

Site Lifecycle 生成项目报告、站点、发布站点。

4.Maven依赖管理

4.1 坐标管理

4.1.1 坐标定义

maven通过坐标定义每一个构建，在pom.xml中定义坐标：

groupId： 定义当前Maven项目名称

artifactId: 定义项目模块

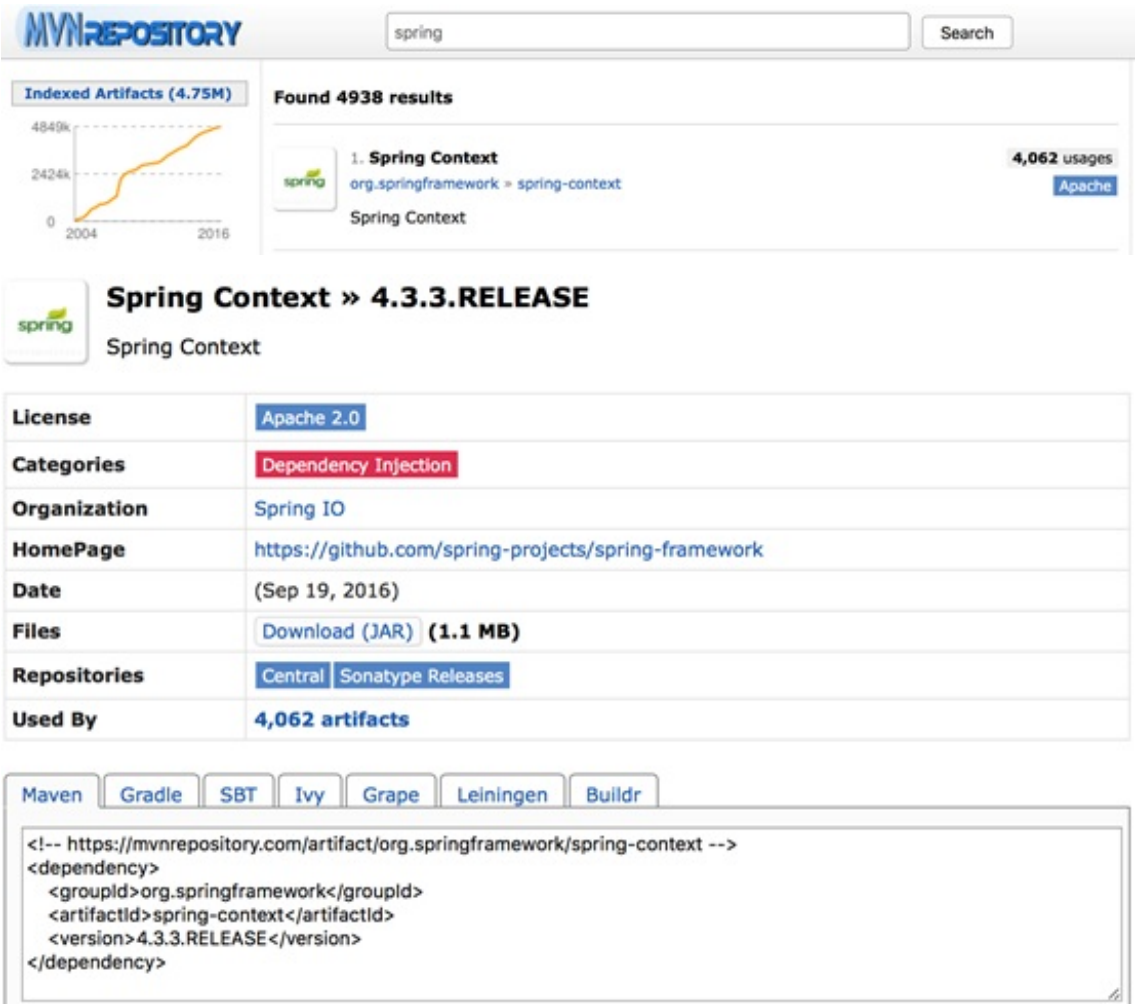
version: 定义当前项目的当前版本

4.1.2 查找坐标

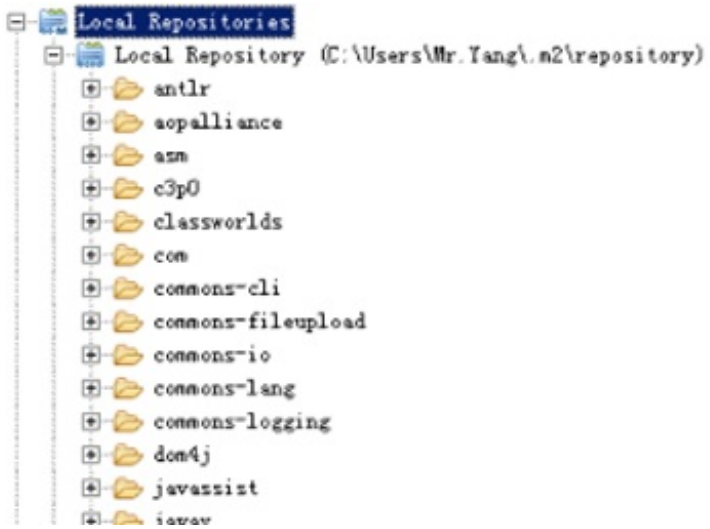
方法一： 使用网站搜索

http:// search.maven.org/

http:// mvnrespository.com/



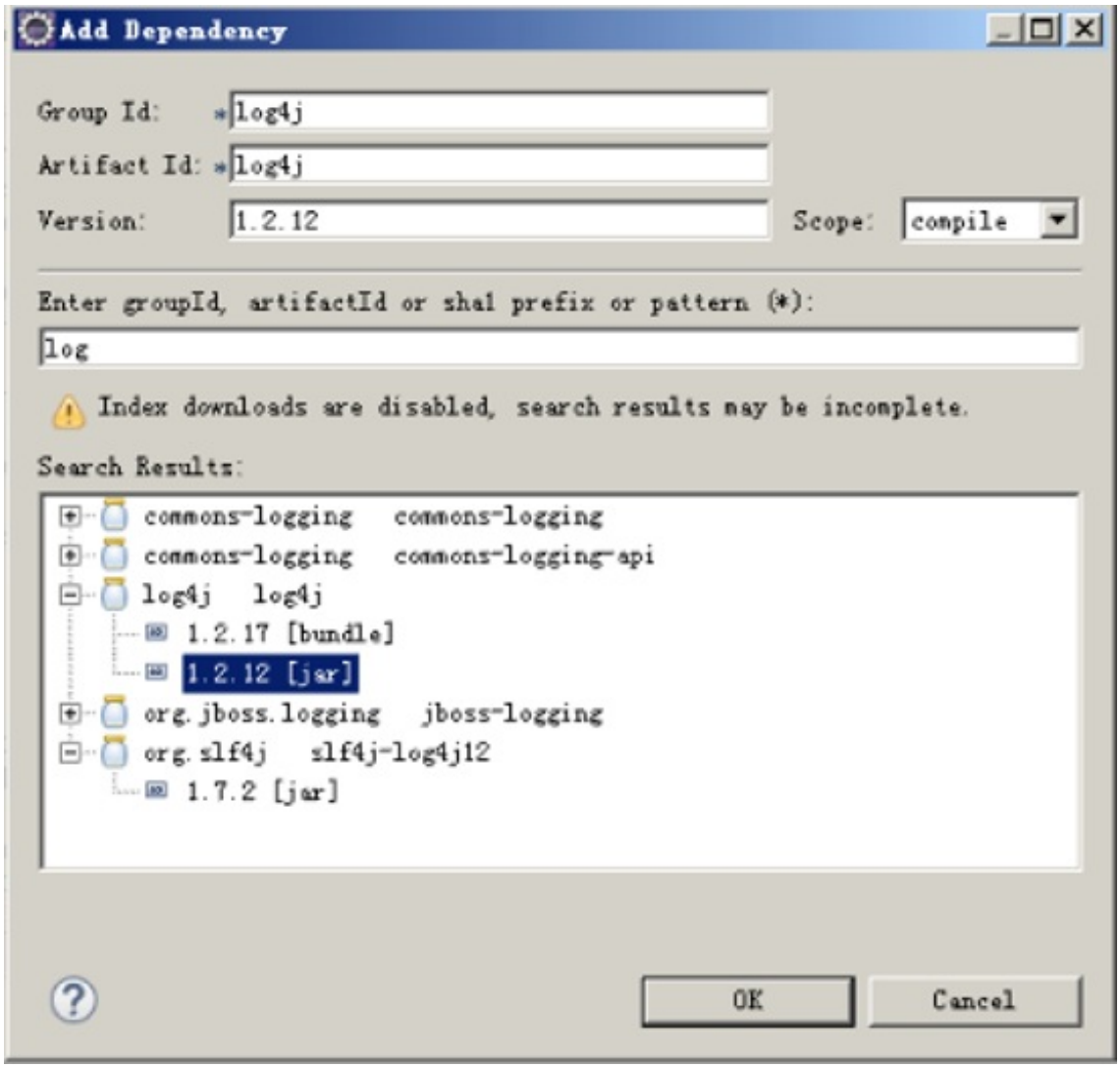
方法二：使用maven插件的索引功能（本地仓库）



4.2 添加依赖

4.2.1 通过坐标导入依赖

在pom.xml文件中右键



4.2.1 依赖范围scope

A依赖B，需要在A的pom.xml文件中添加B的坐标，添加坐标是需要制定依赖范围，依赖范围包括：

1) compile：当依赖的scope为compile的时候,那么当前这个依赖的包,会在编译的时候将这个依赖加入进来，并且在打包（mvn package）的时候也会将这个依赖加入进去意思就是：编译范围有效，在编译与打包时都会存储进去。在默认的情况下scope的范围是compile。

2） provided： 当依赖的scope为provided的时候，在编译和测试的时候有效，在执行（mvn package）进行打包成war包的时候不会加入，比如：servlet-api，因为servlet-api，tomcat等web服务器中已经存在，如果在打包进去，那么包之间就会冲突。

3） test: 当依赖的scope为test的时候，指的的是在测试范围有效，在编译与打包的时候都不会使用这个依赖。

4） runtime： 当依赖的scope为runtime的时候，在运行的时候才会依赖，在编译的时候不会依赖。

5）system: system范围依赖与provided类似，但是你必须显示的提供一个对于本地系统中jar文件的路径，需要制定systemPath磁盘路径，system依赖不推荐使用。

依赖范围	对于编译classpath有效	对于测试classpath有效	对于运行时classpath有效	例子
compile	Y	Y	Y	spring-core
test	-	Y	-	Junit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC驱动
system	Y	Y	-	本地的，Maven仓库之外的类库

总结：

- 默认引入的jar包 compile (默认范围，可以不写，编译、测试、运行都有效)
- jsp-api、servlet-api provided (编译、测试有效，运行时无效，避免和tomcat下的jar冲突)
- jdbc驱动jar包 runtime(测试、运行有效)
- junit test (测试有效)

4.3 传递依赖

4.3.1 什么是传递依赖

A依赖B、B依赖C，将B导入A后会自动导入C，C是A的传递依赖，如果C依赖D，则D也是A的传递依赖。

测试：加入struct2-spring-plugin的依赖如下：

```
<dependency>
    <groupId>org.apache.struts</groupId>
    <artifactId>struts2-spring-plugin</artifactId>
    <version>2.3.31</version>
</dependency>
```

最左边一列为直接依赖，理解为A依赖B的范围，最顶层一行为传递依赖，理解为B依赖C的范围，行与列的交叉即为A传递依赖C的范围。Struts2-spring-plugin依赖struts和spring, 基于传递依赖的原理，工程会自动添加struts和spring的依赖。

示例：

比如A对B有compile依赖，B对C有runtime依赖，那么根据变革所示A对C有runtime依赖。

总结：

每个单元格都对应列中最弱的那个，当P1依赖P2, P2依赖P3 …… ，最终P1传递依赖了Pn的时候，P1对Pn的依赖性取决于依赖链中最弱的一环。

4.4 依赖版本冲突解决

4.4.4.1 问题

当一个项目依赖的构件很多时，它们相互之间存在依赖，当你需要对依赖版本统一管理时，如果让maven自动处理可能并不能如你所愿，看示例：

同事加入以下依赖，观察依赖：

```
<!-- struts2-spring-plugin依赖spring-beans-3.0.5 -->
<dependency>
    <groupId>org.apache.struts</groupId>
    <artifactId>struts2-spring-plugin</artifactId>
    <version>2.3.31</version>
</dependency>

<!-- spring-context 依赖spring-beans-4.2.4-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.2.4.RELEASE</version>
</dependency>
```

struts依赖了spring-beans-3.0.5.jar，spring-context依赖了spring-beans-4.2.4.RELEASE.jar, 但是发现spring-beans-3.0.5加入到了工程中，而我们希望spring-beans-4.2.4加入到工程。

4.4.4.2 依赖调节原则

maven自动按照下边的原则调解：

- 第一声明者优先原则，在pom文件中谁先声明以谁为准。

测试： 将上面的struts-spring-plugin和spring-context顺序颠倒，系统将导入spring-beans-4.2.4。但是即使如此还是有3.0.5版本的jar, 如spring-web。

- 路径近者优先原则

比如：A->spring-beans-4.2.4，A->B->spring-beans-3.0.5，则spring-beans-4.2.4优先。

测试： 在工程中的pom中加入spring-beans-4.2.4的依赖，根据路径近者优先原则，系统将导入spring-beans-4.2.4 ：

```
<dependency>
        <groupId>org.springframework</groupId>
```

```
        <artifactId>spring-beans</artifactId>
        <version>4.2.4.RELEASE</version>
</dependency>
```

4.4.4.3 排除依赖

可以通过排除依赖方式辅助依赖调解：

比如struts2-spring-plugin中添加spring-beans:

```
<!-- struts2-spring-plugin依赖spring-beans-3.0.5 -->
       <dependency>
           <groupId>org.apache.struts</groupId>
           <artifactId>struts2-spring-plugin</artifactId>
           <version>2.3.31</version>
           <!-- 排除依赖 -->
           <exclusions>
              <exclusion>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-beans</artifactId>
              </exclusion>
           </exclusions>
       </dependency>
```

但是这样存在一个问题就是，如果某一个依赖里面过多，排除起来非常麻烦，且也不太确定其依赖了什么。

4.4.4 锁定版本（建议使用）

面对众多的依赖，有一种方法不用考虑依赖路径、声明优先等因素可以采用直接锁定版本的方法确定依赖构件的版本，此方法在企业开发中常用：

<!-- 锁定版本 spring 4.2.4-->

```
<dependencyManagement>
       <dependencies>
           <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-beans</artifactId>
               <version>4.2.4.RELEASE</version>
           </dependency>
           <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-core</artifactId>
               <version>4.2.4.RELEASE</version>
           </dependency>
           <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-web</artifactId>
```

```
            <version>4.2.4.RELEASE</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```