Java技术栈之工具使用

Maven简介与配置

1.什么是Maven

在Java项目开发中，项目的编译、测试、打包等是比较繁琐的，属于重复劳动的工作，浪费人力和时间成本。以往开发项目时，程序员往往需要花较多的精力在**引用jar包搭建项目环境**上，跨部门甚至跨人员之间的项目结构都有可能不一样。Maven的**仓库管理、依赖管理、继承和聚合**等特性为项目的构建提供了一整套完善的解决方案。

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

从官网的介绍中我们可以看到Apache Maven是一个**项目管理工具**，它基于**项目对象模型(POM)的概念**，通过一小段描述信息来管理项目的构建、报告和文档。

2.Maven的作用

Maven是**跨平台的项目管理工具**。主要服务于基于Java平台的项目构建，依赖管理和项目信息管理。

1.项目构建：

项目构建包括清理，…，编译，测试，报告，打包，部署…等步骤

2.理想的项目构建

高度自动化，跨平台，可重用的组件，标准化

3.传统方式管理jar包依赖的问题：

- jar包冲突
- jar包依赖
- jar包体积过大
- jar包在不同阶段无法个性化配置

4.使用maven方式管理jar包依赖的好处：

- 解决jar包冲突
- 解决jar包依赖问题
- jar包不用再每个项目保存，只需要放在仓库即可
- maven可以指定jar包的依赖范围

3.Maven安装与配置

1.Maven安装

1. 到Maven官网(http://maven.apache.org)下载软件

   注意：Maven软件依赖于Java，请先安装与配置好jdk，可参考JDK安装与环境变量配置

2. 直接解压apache-maven-3.6.0-bin.zip到需要安装的目录即可

3. 把Maven的bin目录配置到环境变量中

   新建系统变量

   变量名：M2_HOME

   变量值：C:\develop\Maven\apache-maven-3.6.0

4. 在命令提示符下输入mvn –version

```
Apache Maven 3.6.0 (97c98ec64a1fdfee7767ce5ffb20918da4f719f3; 2018-10-
25T02:41:47+08:00)
Maven home: C:\develop\Maven\apache-maven-3.6.0\bin\..
Java version: 1.8.0_191, vendor: Oracle Corporation, runtime:
C:\develop\Java\jdk1.8.0_191\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

如果你看到类似消息，说明 Apache Maven 在 Windows 上已安装成功。

2.Maven配置

1.配置本地仓库位置

Maven安装好之后默认配置了本地仓库，在%user%/.m2/respository目录，但是通常不会使用Maven的默认本地仓库，而是修改maven的本地仓库的地址，修改Maven目录的conf/settings.xml

```
<localRepository>C:\develop\Maven\apache-maven-
3.6.0\respository</localRepository>
```

2.配置阿里云镜像

为了更好的下载速度，我们会选用**国内镜像**，这里配置的是**阿里云镜像**，修改Maven目录的conf/settings.xml

```
<!-- 配置阿里云镜像 -->
<mirror>
    <!--该镜像的唯一标识符。id用来区分不同的mirror元素。 -->
    <id>nexus-aliyun</id>
    <!--*指的是访问任何仓库都使用我们的私服-->
    <mirrorOf>*</mirrorOf>
    <!--镜像名称-->
    <name>Nexus aliyun</name>
    <!--该镜像的URL。构建系统会优先考虑使用该URL，而非使用默认的服务器URL-->
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

3.设置Maven工程的默认jdk

创建Maven项目的时候，默认编译的jdk版本是1.7，但是我们需要使用的是**jdk1.8版本**，所以需要修改Maven目录的conf/settings.xml

```
<!-- 配置maven编译jdk版本 -->
<profile>
    <id>jdk-1.8</id>
    <activation>
        <activeByDefault>true</activeByDefault>
        <jdk>1.8</jdk>
    </activation>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
    </properties>
</profile>
```

4.Maven标准目录结构

若要使用Maven，那么项目的目录结构必须符合Maven的规范，其目录结构如下：

```
project
  src
    main  项目主代码和资源
      java  项目的Java源代码
      resources 项目的资源文件
      webapp  web项目资源文件(可无)
    test  单元测试代码和资源
      java  测试的Java源代码
      resources 测试的资源文件(可无)
  target    打包输出目录(可无)
    classes 编译输出目录
    test-classes  测试编译输出目录
  pom.xml
```

5.Maven的几个核心概念

1.POM

POM(Project Object Model)项目对象模型，一个项目所有的配置都放在POM文件中：定义项目的类型、名字、管理依赖关系，定制插件的行为等等。Maven通过pom.xml文件来管理依赖和管理项目的构建生命周期，而项目构建的生命周期是依靠一个个的插件完成的。

2.Maven仓库

Maven管理资源的位置。仓库里面包含**依赖**（jar包）和**插件**（plug-in）。Maven仓库分为本地仓库和远程仓库，而远程仓库又包括私服和中央仓库。

3.本地仓库

用户自己电脑上的仓库，直接从本地获取。

4.远程仓库

私服

私服是一种特殊的远程仓库，搭建在局域网内的仓库，私服代理广域网的仓库，提供给局域网内的用户使用，可用减少局域网内的用户与外界仓库的传输，每一个jar包只需要拉取一次就可以提供给局域网内所有的用户使用，并且也更加稳定。

中央仓库

Maven官方提供的远程仓库，里面拥有最全的jar包资源，Maven首先从本地仓库中寻找项目所需的jar包，若本地仓库没有，再到Maven的中央仓库下载所需jar包。地址是：http://repo1.maven.org/maven2/。

5.坐标

在Maven中，坐标是jar包的唯一标识，Maven通过坐标在仓库中找到项目所需的jar包。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.6.RELEASE</version>
</dependency>
```

```
groupId：公司或组织域名倒序
artifactId：模块名
version：版本号
packaging：项目的打包方式(pom/jar/war，默认jar)
groupId、artifactId、versioin简称GAV(Maven坐标)，是用来唯一标识jar包的。
```

最新最全的Maven依赖项版本查询网站：http://mvnrepository.com/

6.Maven工程的坐标与仓库中路径的关系

Maven坐标和仓库对应的映射关系：

```
[groupId][artifactId][version][artifactId]-[version].jar
```

对应本地仓库目录：

```
org\springframework\spring-core\4.3.4.RELEASE\spring-core-4.3.4.RELEASE.jar
```

7.依赖传递

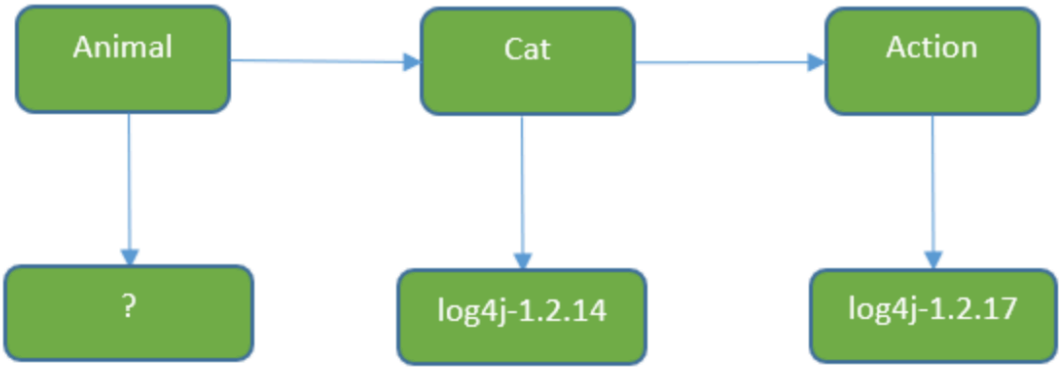
如果我们的项目引用了一个Jar包，而该Jar包又引用了其他Jar包，那么在默认情况下项目编译时，Maven会把直接引用和简洁引用的Jar包都下载到本地。

传递性依赖机制能够大大的简化依赖声明，而且大部分情况下我们只需要关心项目的直接依赖是什么，而不用考虑这些直接依赖会引入什么传递性依赖，但是当出现冲突了，则需要很清楚传递性依赖是从什么依赖路径引入的。

8.依赖冲突

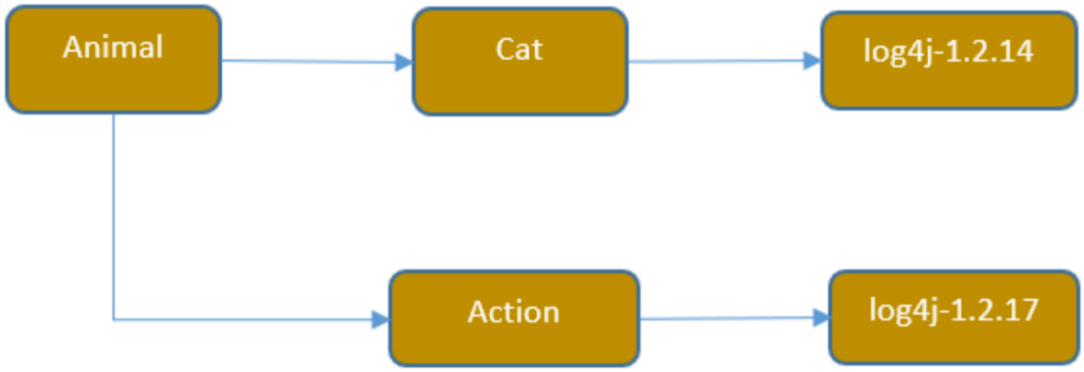
依赖的原则主要是为了解决模块之间jar包冲突问题。我们分两种情况说明一下：

1.路径最短者优先



例如：当我们工程Animal依赖Cat，Cat又依赖Action，每个单独工程中Action依赖log4j-1.2.17，Cat依赖log4j-1.2.14，那我们的Animal工程要依赖于哪个版本呢？其实Maven为我们提供了内置的原则，就是路径最短者优先，我们的Animal工程最终依赖的是log4j-1.2.14

2.路径相同先声明优先



例如：当我们工程Animal同时依赖Cat，又依赖Action，每个单独工程中Action依赖log4j-1.2.17，Cat依赖log4j-1.2.14，这时候依赖的路径是相同的，那我们的Animal工程最终依赖的是哪个版本呢?

3.统一管理依赖的版本

```
<properties>
  <!-- 在properties里面统一管理依赖的版本 -->
  <spring-boot.version>2.1.3.RELEASE</spring-boot.version>
  <spring-cloud-starter-alibaba.version>0.9.0.RELEASE</spring-cloud-starter-
alibaba.version>
```

```
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>${spring-boot.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
      <version>${spring-cloud-starter-alibaba.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
      <version>${spring-boot.version}</version>
    </dependency>
  </dependencies>
```

9.依赖范围

依赖范围就是控制依赖在不同阶段的作用。**不同的依赖会使用不同的classpath**，在Maven中依赖的域有这几个：import、provided、runtime、compile、system、test。默认取值为compile。

依赖范围	对编译classpath有效	对测试classpath有效	对运行classpath有效	说明	例子
compile	Y	Y	Y	编译范围，默认scope，在classpath中存在	spring-core
test	-	Y	-	测试范围，单元测试环境需要	junit
provided	Y	Y	-	已提供范围，比如容器提供servlet api	servlet-api
runtime	-	Y	Y	运行时范围，编译不需要，接口与实现分离	jdbc驱动
system	Y	Y	-	系统范围，自定义构件，指定systemPath	本地的，Maven仓库之外的类库

10.可选依赖和依赖排除

可选依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.6.RELEASE</version>
    <!-- 不可以向下传递依赖 -->
    <optional>true</optional>
</dependency>
在导入一个依赖的时候，是否需要把这个依赖向下传递
```

false：可以向下传递（默认值）

true：不可以向下传递

排除依赖

如果我们只想下载直接引用的Jar包，那么需要在pom.xml中做如下配置：(将需要排除的Jar包的坐标写在中)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- 去掉springboot默认配置的logback依赖 -->
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

11.聚合

什么是聚合?

将多个项目同时运行就称为聚合。聚合的作用，是为了简化构建项目的过程。一次性构建多个项目!

如何实现聚合?

只需在pom中作如下配置即可实现聚合

```
<modules>
  <module>nacos-config-example</module>
  <module>nacos-discovery-example</module>
  <module>nacos-gateway-example</module>
</modues>
```

12.继承

什么是继承?

在聚合多个项目时，如果这些被聚合的项目中需要引入相同的Jar，那么可以将这些Jar写入父pom中，各个子项目继承该pom即可。继承的作用，为了简化pom.xml配置，简化groupId，artifactId，version ，还可以锁定子工程依赖版本。

如何实现继承?

父pom配置：将需要继承的Jar包的坐标放入标签即可。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>28.0-jre</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

子pom配置：

```
<parent>
    <groupId>父pom所在项目的groupId</groupId>
    <artifactId>父pom所在项目的artifactId</artifactId>
    <version>父pom所在项目的版本号</version>
</parent>
```

13.生命周期

Maven有三套相互独立的生命周期，请注意这里说的是“三套”，而且“相互独立”，初学者容易将Maven的生命周期看成一个整体，其实不然。这三套生命周期分别是：

① Clean Lifecycle 在进行真正的构建之前进行一些清理工作。 Clean生命周期一共包含了三个阶段：

- pre-clean 执行一些需要在clean之前完成的工作 clean 移除所有上一次构建生成的文件 post-clean 执行一些需要在clean之后立刻完成的工作

② Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。

- validate generate-sources process-sources generate-resources process-resources 复制并处理资源文件，至目标目录，准备打包 compile 编译项目的源代码 process-classes generate-test-sources process-test-sources generate-test-resources process-test-resources 复制并处理资源文件，至目标测试目录 test-compile 编译测试源代码 process-test-classes 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 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 将包安装至本地仓库，以让其它项目依赖。 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

总结：不论你要执行生命周期的哪一个阶段，maven都是从这个生命周期的开始执行

插件：每个阶段都有插件（plugin）。插件的职责就是执行它对应的命令。

③ Site Lifecycle 生成项目报告，站点，发布站点。

- pre-site 执行一些需要在生成站点文档之前完成的工作 site 生成项目的站点文档 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 site-deploy 将生成的站点文档部署到特定的服务器上

6.Maven常用操作

1.手动安装Maven依赖

在使用Maven的依赖Oracle的驱动包时，会出现依赖错误的情况，原因是版权原因，Oracle官方屏蔽了依赖，那么要在本地使用其数据驱动包，要怎么做呢？去Oracle官网下载依赖然后安装到本地仓库

```
mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc7 -
Dversion=12.1.0.2 -Dpackaging=jar -Dfile=E:\ojdbc7.jar
```

-Dfile为jar包的位置，执行完maven命令，然后再引入依赖

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc7</artifactId>
    <version>12.1.0.2</version>
</dependency>
```

2.部署jar包到远程仓库

部署jar包到远程仓库主要包括两个部分：远程仓库认证，部署jar包到远程仓库

3.构建多模块Maven项目

dependencyManagement

在项目开发过程中，有时一个项目下面包含了几个子模块，在多模块的情况，POM的配置应该要注意写什么呢？我们通过一个例子来说明下。有这样一个工程，里面有A模块、B模块和C模块，A模块需要引入junit和log4j库，配置如下：

```
<dependency>
    <groupId>junit</groupId>
    <artifactid>junit</artifactId>
    <version>4.11</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactid>log4j</artifactId>
    <version>1.2.16</version>
</dependency>
```

此时B模块也需要引入这两个库，配置如下：

```
<dependency>
    <groupId>junit</groupId>
    <artifactid>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactid>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

会发现A模块和B模块对junit和log4j库依赖的版本是不同的，出现这种情况是十分危险的，因为依赖不同版本的库可能会造成很多未知的风险。怎么解决不同模块之间对同一个库的依赖版本一样呢？Maven提供了优雅的解决办法，使用继承机制以及dependencyManagement元素来解决这个问题。 如果你在父模块中配置dependencies，那么所有的子模块都自动继承，不仅达到了依赖一致的目的，还省了大段的代码，但这样来做会存在问题的。比如B模块需要spring-aop模块，但是C模块不需要spring-aop模块，如果用dependencies在父类中统一配置，C模块中也会包含有spring-aop模块，不符合我们的要求。但是用dependencyManagement就没有这样的问题。dependencyManagement只会影响现有依赖的配置，但不会引入依赖。 这样我们在父模块中的配置可以更改为如下所示：

```
<!-- dependencyManagement只会影响现有依赖的配置，但不会引入依赖。 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

这段配置不会给任何子模块引入依赖，如果某个子模块需要junit和log4j，只需要这样配置即可：

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
    </dependency>
</dependencies>
```

在多模块Maven项目中，使用dependencyManagement能够有效地帮我们维护依赖一致性。

pluginManagement

上面介绍了在多模块中对依赖库的管理，接下来介绍下对插件的管理。与dependencyManagement类似，我们可以使用pluginManagement元素管理插件。一个常见的用法就是我们希望项目所有模块的使用compiler插件的时候，都是用jdk1.8，以及指定Java源文件编码为UTF-8，这时可以在父模块的POM中如下配置pluginManagement：

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

这段配置会被应用到所有子模块的compiler插件中，因为Maven内置了compiler插件与生命周期的绑定，因此子模块不需要任何maven-compiler-plugin的配置了。

这段配置会被应用到所有子模块的compiler插件中，因为Maven内置了compiler插件与生命周期的绑定，因此子模块不需要任何maven-compiler-plugin的配置了。

4.Maven常用命令

命令	说明
mvn clean	清除原来的编译结果
mvn compile	编译
mvn test	运行测试代码，mvn test -Dtest=类名 //单独运行测试类
mvn package	打包项目，mvn package -Dmaven.test.skip=true //打包时不执行测试
mvn install	将项目打包并安装到本地仓库
mvn deploy	发布到本地仓库或者服务器

5.Maven常用插件

Maven本质上是一个插件框架，它的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来完成。下面说几个常用的插件：

maven-compiler-plugin(编译插件)

用来编译Java代码，在对Java代码进行编译的时候，可以指定使用哪个JDK版本来进行编译，配置如下所示：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
      <!-- 源代码使用jdk1.8支持的特性 -->
        <source>1.8</source>
        <!-- 使用jdk1.8编译目标代码 -->
        <target>1.8</target>
             <!-- 传递参数 -->
               <compilerArgs>
                    <arg>-parameters</arg>
                    <arg>-Xlint:unchecked</arg>
                    <arg>-Xlint:deprecation </arg>
              </compilerArgs>
    </configuration>
</plugin>
```

maven-resources-plugin(资源插件)

Maven区别对待Java代码和资源文件，maven-resources-plugin则用来处理资源文件。默认的主资源文件目录是src/main/resources，很多时候会需要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，配置如下所示：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>3.1.0</version>
    <executions>
        <execution>
          <!-- 与Maven编译生命周期绑定在一起 -->
            <phase>compile</phase>
        </execution>
    </executions>
</plugin>
```

maven-clean-plugin(清除插件)

主要作用就是清理构建目录下的全部内容，有些项目，构建时需要清理构建目录以外的文件，比如指定的库文件，这时候就需要配置来实现了，配置如下所示：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-clean-plugin</artifactId>
    <version>3.1.0</version>
    <configuration>
        <!--<skip>true</skip>-->
        <!--<failOnError>false</failOnError>-->
        <!--当配置true时,只清理filesets里的文件,构建目录中得文件不被清理.默认是flase.--
>
        <excludeDefaultDirectories>false</excludeDefaultDirectories>
        <filesets>
            <fileset>
                <!--要清理的目录位置-->
                <directory>${basedir}/logs</directory>
                <!--是否跟随符号链接（symbolic links)-->
                <followSymlinks>false</followSymlinks>
             </fileset>
        </filesets>
    </configuration>
</plugin>
```

maven-war-plugin(打包插件)

主要作用就是用来打包的，在打包的时候经常需要排除一些文件，就需要对warSourceExcludes进行配置了，配置如下所示：

```
<plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.2.3</version>
    <configuration>
        <warSourceExcludes>WEB-INF/lib/**</warSourceExcludes>
    </configuration>
</plugin>
```

IDEA使用与配置

参见博客:

[Intellij IDEA 使用教程] https://www.jianshu.com/p/572772f84105?utm_source=oschina-app

[IDEA常用配置和常用插件] https://blog.csdn.net/ThinkWon/article/details/101020481

常用插件

- Alibaba Java Coding Guidelines -阿里巴巴代码规范检查插件
- FindBugs-IDEA -检查潜在bug插件
- Free Mybatis plugin -Mybatis 辅助插件
- GsonFormat -将JSON字符串转换为内部类实体类插件
- Lombok plugin -简化实体类编写插件
- Maven Helper -Maven辅助插件
- SonarLint -代码质量检查插件
- Translation -翻译插件
- CodeGlance -代码地图
- .ignore -git忽略文件
- CamelCase -驼峰式转换插件
- String Manipulation -一款强大的字符串转换工具
- Key Promoter X -一款可以进行快捷键提示的插件
- AceJump -一款可以彻底摆脱鼠标的插件
- IDEA查看日志的插件
- ANSI Hignlighter - 高亮插件
- Ideolog – ide查看日志插件

Git使用与配置

1.版本控制概念

1.什么是版本控制

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。 除了项目源代码，你可以对任何类型的文件进行版本控制。

2.版本控制的作用

有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。

3.本地版本控制系统

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。 这么做唯一的好处就是简单，但是特别容易犯错。 有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖文件。为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。下图来源于Git官网。

4.集中式版本控制系统

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？ 于是，集中化的版本控制系统（Centralized Version Control Systems，简称 CVCS）应运而生。集中化的版本控制系统都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。

这么做虽然解决了本地版本控制系统无法让在不同系统上的开发者协同工作的诟病，但也还是存在下面的问题：

•**单点故障**：中央服务器宕机，则其他人无法使用；如果中心数据库磁盘损坏没有进行备份，你将丢失所有数据。本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

•**必须联网才能工作**：受网络状况、带宽影响。

5.分布式版本控制系统

于是分布式版本控制系统（Distributed Version Control System，简称 DVCS）面世了。 Git 就是一个典型的分布式版本控制系统。这类系统，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。 这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。

分布式版本控制系统可以不用联网就可以工作，因为每个人的电脑上都是完整的版本库，当你修改了某个文件后，你只需要将自己的修改推送给别人就可以了。但是，在实际使用分布式版本控制系统的时候，很少会直接进行推送修改，而是使用一台充当“中央服务器”的东西。这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。分布式版本控制系统的优势不单是不必联网这么简单，后面我们还会看到 Git 极其强大的分支管理等功能。

2.Git概述

Git是什么

Git是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理。Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git特点

优点：

- 适合分布式开发，强调个体；
- 公共服务器压力和数据量都不会太大；
- 速度快、灵活；
- 任意两个开发者之间可以很容易的解决冲突；
- 离线工作。

缺点：

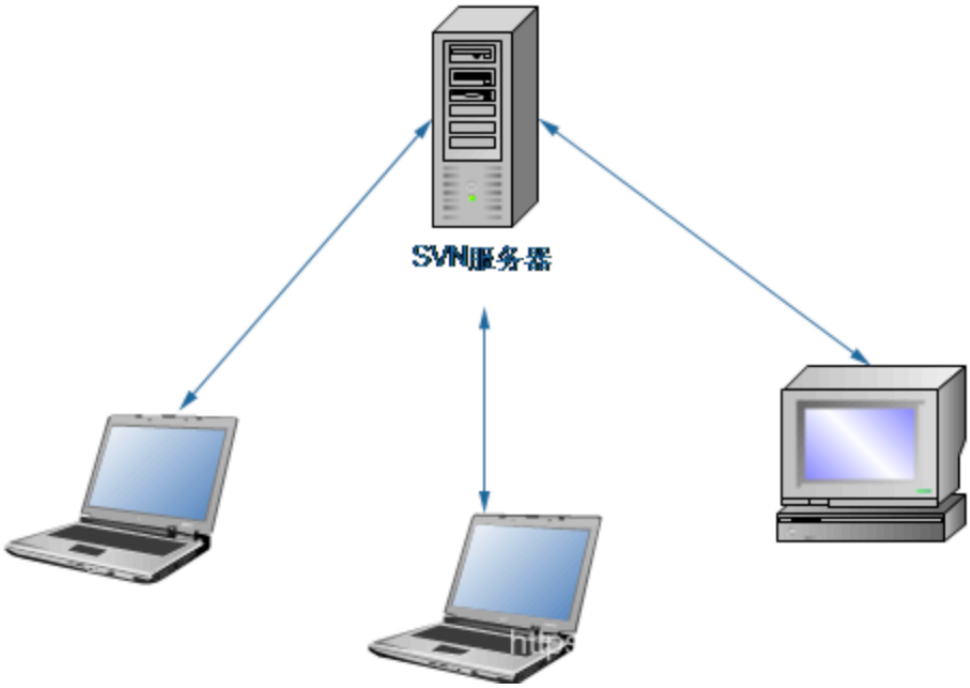
- 代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息；
- 权限控制不友好；如果需要对开发者限制各种权限的建议使用SVN。

3.Git与SVN的区别

SVN是**集中式版本控制系统**，而Git是**分布式版本控制系统**

1.SVN

SVN是集中式版本控制系统，版本库是集中放在中央服务器的，而干活的时候，用的都是自己的电脑，所以首先要从中央服务器哪里得到最新的版本，然后干活，干完后，需要把自己做完的活推送到中央服务器。集中式版本控制系统是必须联网才能工作，如果在局域网还可以，带宽够大，速度够快，如果在互联网下，如果网速慢的话，就郁闷了。下图就是标准的集中式版本控制工具管理方式：



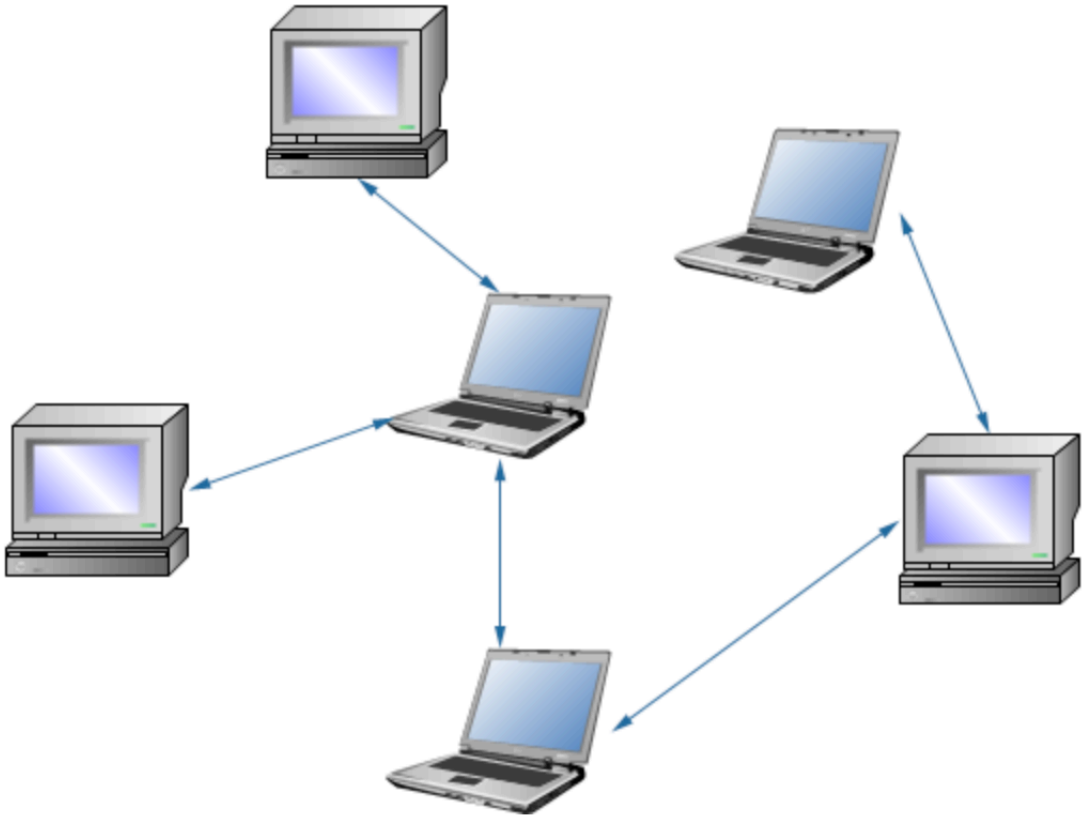
集中管理方式在一定程度上看到其他开发人员在干什么，而管理员也可以很轻松掌握每个人的开发权限。

但是相较于其优点而言，集中式版本控制工具缺点很明显：

- 服务器单点故障
- 容错性差

2.Git

Git是分布式版本控制系统，它没有中央服务器，每个人的电脑就是一个完整的版本库，这样工作的时候就不需要联网了，因为版本都是在自己的电脑上。既然每个人的电脑都有一个完整的版本库，那多个人如何协作呢？比如说自己在电脑上改了文件A，其他人也在电脑上改了文件A，这时，你们两之间只需把各自的修改**推送给对方**，就可以互相看到对方的修改了。下图就是分布式版本控制工具管理方式：



3.Git与SVN的区别

Git不仅仅是个版本控制系统，它也是个**内容管理系统**(CMS),工作管理系统等。如果你是一个具有使用SVN背景的人，你需要做一定的思想转换，来适应Git提供的一些概念和特征。 Git 与 SVN 区别点：

- Git是分布式的，SVN不是：这是Git和其它非分布式的版本控制系统，例如SVN，CVS等，最核心的区别。
- Git把内容按元数据方式存储，而SVN是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似.svn,.cvs等的文件夹里。
- Git分支和SVN的分支不同：分支在SVN中一点不特别，就是版本库中的另外的一个目录。
- Git没有一个全局的版本号，而SVN有：目前为止这是跟SVN相比GIT缺少的最大的一个特征。
- Git的内容完整性要优于SVN：Git的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。

4.Git 与其他版本管理系统的区别

Git 在保存和对待各种信息的时候与其它版本控制系统有很大差异，尽管操作起来的命令形式非常相近，理解这些差异将有助于防止你使用中的困惑。 Git与其他版本管理系统的主要差别：**对待数据的方式。**

Git采用的是直接记录快照的方式，而非差异比较。

大部分版本控制系统（CVS、Subversion、Perforce、Bazaar 等等）都是以文件变更列表的方式存储信息，这类系统将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。

具体原理理解起来其实很简单，每当我们提交更新一个文件之后，系统记录都会记录这个文件做了哪些更新，以增量符号Δ(Delta)表示。

我们怎样才能得到一个文件的最终版本呢?

很简单，高中数学的基本知识，我们只需要将这些原文件和这些增加进行相加就行了。

这种方式有什么问题呢?

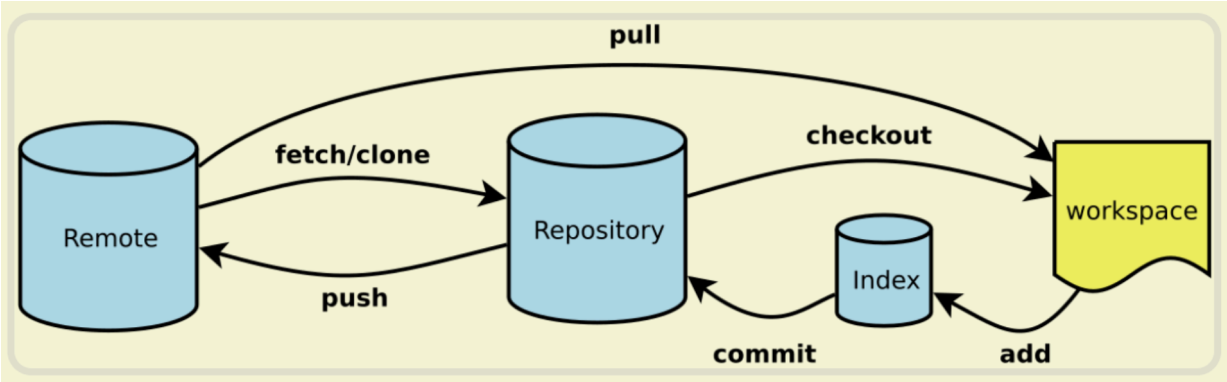
比如我们的增量特别特别多的话，如果我们要得到最终的文件是不是会耗费时间和性能。

Git 不按照以上方式对待或保存数据。 反之，Git 更像是把数据看作是对小型文件系统的一组快照。 每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个 快照流。

4.Git的核心概念

1.工作区、暂存区、版本库、远程仓库

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。



Workspace： 工作区，就是你平时存放项目代码的地方

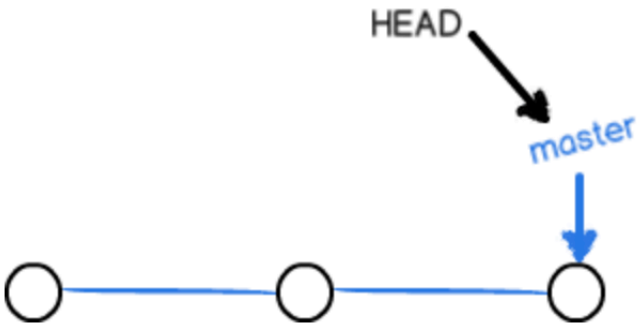
Index / Stage： 暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息

Repository： 仓库区（或版本库），就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中HEAD指向最新放入仓库的版本

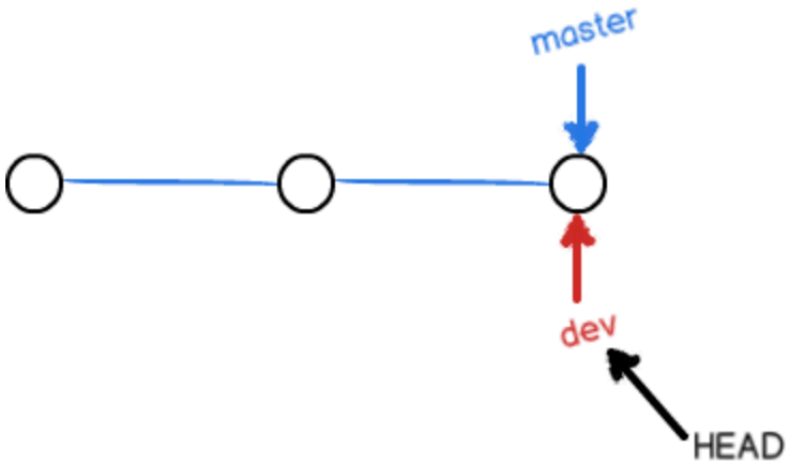
Remote： 远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换

2.分支

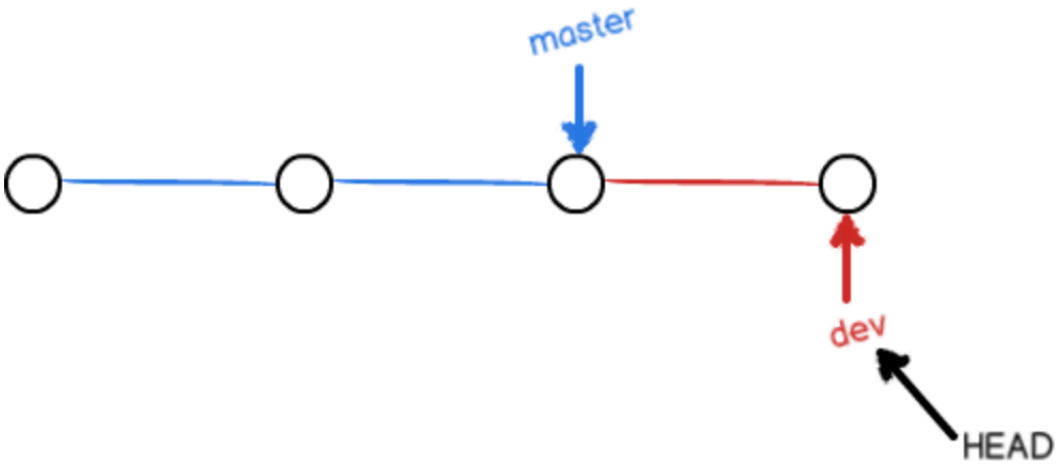
每次的提交Git都把它们**串成一条时间线**，这条时间线就是一个分支。截止到目前，只有**一条时间线**，在Git里这个分支叫主分支，即master分支。HEAD指针严格来说不是指向提交，而是指向master，master才是指向提交的。一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



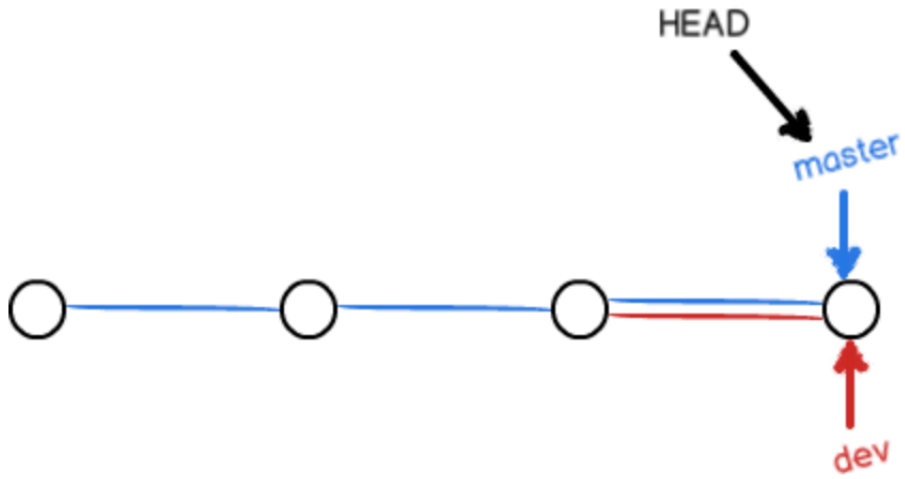
每次提交，master分支都会向前移动一步，这样随着不断提交，master分支的线也越来越长。当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



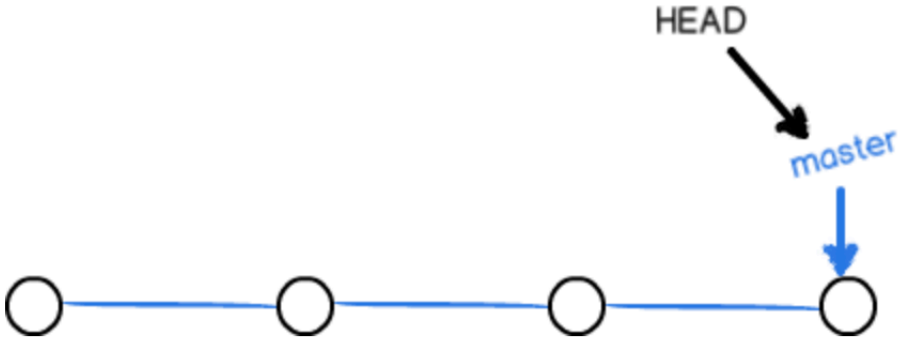
Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！不过切换到了dev分，对工作区的修改和提交就是**针对dev分支**了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



假如我们在dev上的工作完成了，就可以**把dev合并到master上**。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后就剩下了一条master分支：



3.远程仓库

远程仓库分为公有远程仓库和私有远程仓库。

公有远程仓库

本质和本地仓库无异，只是这个仓库①不在本地②大家可能都知道③需要将代码共享到远程仓库④可以被其他人克隆同步代码等。

一般情况下在企业中会有一个搭建在公司的远程仓库，可以让本公司内部的开发人员同步开发。而业界最富盛名的远程仓库则为github；它上面存放了非常多的开源组织、个人、企业等的开放源码库，任何都可以从上面获取源码。

私有远程仓库

远程仓库实际上和本地仓库一样，纯粹为了7x24小时开机并交换大家的修改。GitHub就是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。

GitHub远程仓库

在本地创建了一个Git仓库，又想让其他人来协作开发，此时就可以把本地仓库同步到远程仓库，同时还增加了本地仓库的一个备份。常用的远程仓库就是github：https://github.com/

Github支持两种同步方式“https”和“ssh”。如果使用https很简单基本不需要配置就可以使用，但是每次提交代码和下载代码时都需要输入用户名和密码。而且如果是公司配置的私有git服务器一般不提供https方式访问。

4.忽略文件

在工程中，并不是所有文件都需要保存到版本库中的，例如“target”目录下的文件就可以忽略。在Git工作区的根目录下创建一个**特殊的.gitignore文件**，然后把要忽略的文件名填进去，Git就会自动忽略这些文件或目录。

1.Git 忽略规则优先级

在 .gitingore 文件中，每一行指定一个忽略规则，Git 检查忽略规则的时候有多个来源，它的优先级如下（由高到低）：

```
从命令行中读取可用的忽略规则
当前目录定义的规则
父级目录定义的规则，依次递推
$GIT_DIR/info/exclude 文件中定义的规则
core.excludesfile中定义的全局规则
```

2.Git 忽略规则匹配语法

在 .gitignore 文件中，每一行的忽略规则的语法如下：

```
空格不匹配任意文件，可作为分隔符，可用反斜杠转义
开头的文件标识注释，可以使用反斜杠进行转义
! 开头的模式标识否定，该文件将会再次被包含，如果排除了该文件的父级目录，则使用 ！ 也不会再次被包含。可以使用反斜杠进行转义
/ 结束的模式只匹配文件夹以及在该文件夹路径下的内容，但是不匹配该文件
/ 开始的模式匹配项目跟目录
如果一个模式不包含斜杠，则它匹配相对于当前 .gitignore 文件路径的内容，如果该模式不在 .gitignore 文件中，则相对于项目根目录
** 匹配多级目录，可在开始，中间，结束
? 通用匹配单个字符
*通用匹配零个或多个字符
[] 通用匹配单个字符列表
```

常用匹配示例

```
bin/: 忽略当前路径下的bin文件夹，该文件夹下的所有内容都会被忽略，不忽略 bin 文件
/bin: 忽略根目录下的bin文件
/*.c: 忽略 cat.c，不忽略 build/cat.c
debug/*.obj: 忽略 debug/io.obj，不忽略 debug/common/io.obj 和 tools/debug/io.obj
**/foo: 忽略/foo，a/foo，a/b/foo等
a/**/b: 忽略a/b，a/x/b，a/x/y/b等
!/bin/run.sh: 不忽略 bin 目录下的 run.sh 文件
*.log: 忽略所有 .log 文件
config.php: 忽略当前路径的 config.php 文件
```

.gitignore规则不生效！！！！

.gitignore只能忽略那些原来没有被track的文件，如果某些文件已经被纳入了版本管理中，则修改.gitignore是**无效的**。解决方法就是**先把本地缓存删除（改变成未track状态），然后再提交**:

```
git rm -r --cached .
git add .
git commit -m 'update .gitignore'
```

你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

```
$ git add App.class
The following paths are ignored by one of your .gitignore files:
App.class
Use -f if you really want to add them.
```

如果你确实想添加该文件，可以用-f强制添加到Git：

```
$ git add -f App.class
```

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用**git check-ignore**命令：

```
$ git check-ignore -v App.class
.gitignore:3:*.class    App.class
```

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

Java项目中常用的.gitignore文件

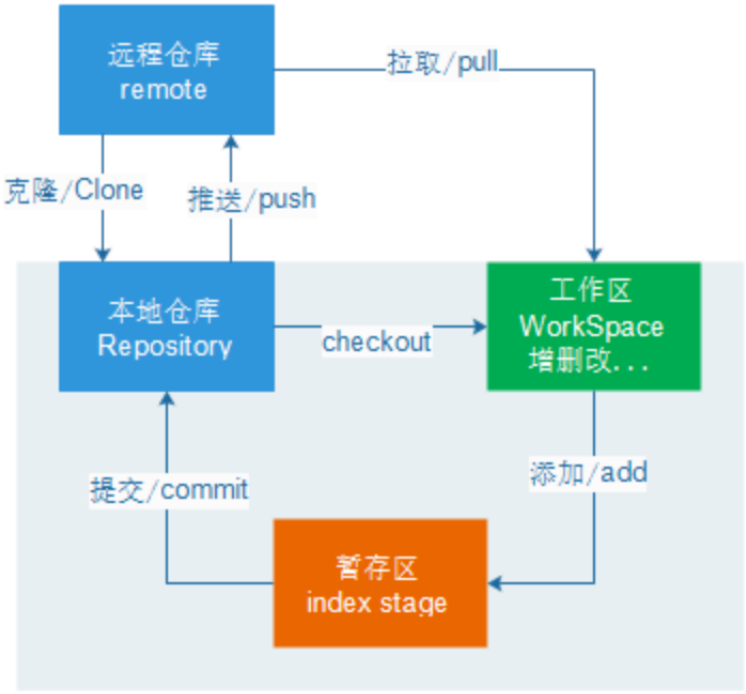
```
# Compiled class file
*.class

# Eclipse
.project
.classpath
.settings/

# Intellij
*.ipr
*.iml
*.iws
.idea/

# Maven
target/

# Gradle
```

```
build
.gradle

# Log file
*.log
log/

# out
**/out/

# Mac
.DS_Store

# others
*.jar
*.war
*.zip
*.tar
*.tar.gz
*.pid
*.orig
temp/
```

5.Git工作流程

- 从远程仓库中克隆 Git 资源作为本地仓库；
- 从本地仓库中checkout代码然后进行代码修改；
- 在提交本地仓库前先将代码提交到暂存区；
- 提交修改，提交到本地仓库；本地仓库中保存修改的各个历史版本；
- 在需要和团队成员共享代码时，可以将修改代码push到远程仓库。

Git 的工作流程图如下：



6.常用Git命令

经常使用 Git，但是很多命令还是记不住。但要熟练掌握，恐怕要记住40~60个命令，所以整理了一份常用Git命令清单。

1.配置用户名和邮箱

```
$ git --version   # 查看git的版本信息
$ git config --global user.name   # 获取当前登录的用户
$ git config --global user.email  # 获取当前登录用户的邮箱

# 如果刚没有获取到用户配置，则只能拉取代码，不能修改  要是使用git，你要告诉git是谁在使用
$ git config --global user.name 'userName'    # 设置git账户，userName为你的git账号，
$ git config --global user.email 'email'
# 获取Git配置信息，执行以下命令：
$ git config -list
```

2.配置https和ssh

推送时保存用户名和密码

```
# https提交保存用户名和密码
$ git config --global credential.helper store
# 生成公钥私钥，将公钥配置到GitHub，ssh提交就可以免输入用户名密码
# 三次回车即可生成 ssh key
$ ssh-keygen -t rsa
# 查看已生成的公钥
$ cat ~/.ssh/id_rsa.pub
```

3.推送到远程仓库

```
1. git init # 初始化仓库
2. git add .(文件name) # 添加文件到暂存区
3. git commit -m "first commit" # 添加文件到本地仓库并提交描述信息
4. git remote add origin 远程仓库地址 # 链接远程仓库，创建主分支
5. git pull origin master --allow-unrelated-histories # 把本地仓库的变化连接到远程仓库主分支
6. git push -u origin master # 把本地仓库的文件推送到远程仓库
```

4.新建

```
# 创建一个文件夹
$ mkdir GitRepositories     # 创建文件夹GitRepositories
$ cd GitRepositories        # 切换到GitRepositories目录下
# 在当前目录新建一个Git代码库
$ git init
# 新建一个目录，将其初始化为Git代码库
$ git init [project-name]
# 下载一个项目和它的整个代码历史
$ git clone [url]
```

5.增删

```
# 添加指定文件到暂存区
$ git add [file1][file2] ...
# 添加指定目录到暂存区，包括子目录
$ git add [dir]
# 添加当前目录的所有文件到暂存区
$ git add .
# 添加每个变化前，都会要求确认
# 对于同一个文件的多处变化，可以实现分次提交
$ git add -p
# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...
# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]
# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

6.提交

```
# 提交暂存区到仓库区
$ git commit -m [message]
# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] ... -m [message]
# 提交工作区自上次commit之后的变化，直接到仓库区
$ git commit -a
# 提交时显示所有diff信息
$ git commit -v
# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m [message]
# 重做上一次commit，并包括指定文件的新变化
$ git commit --amend [file1] [file2] ...
```

7.分支

```
# 列出所有本地分支
$ git branch
# 列出所有远程分支
$ git branch -r
# 列出所有本地分支和远程分支
$ git branch -a
# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]
# 新建一个分支，并切换到该分支
$ git checkout -b [branch]
# 新建一个分支，指向指定commit
$ git branch [branch] [commit]
# 新建一个分支，与指定的远程分支建立追踪关系
$ git branch --track [branch] [remote-branch]
# 切换到指定分支，并更新工作区
$ git checkout [branch-name]
# 切换到上一个分支
$ git checkout -
# 建立追踪关系，在现有分支与指定的远程分支之间
$ git branch --set-upstream [branch] [remote-branch]
# 合并指定分支到当前分支
$ git merge [branch]
# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]
# 删除分支
$ git branch -d [branch-name]
# 删除远程分支
$ git push origin --delete [branch-name]
$ git branch -dr [remote/branch]
```

8.同步

```
# 下载远程仓库的所有变动
$ git fetch [remote]
# 显示所有远程仓库
$ git remote -v
# 显示某个远程仓库的信息
$ git remote show [remote]
# 增加一个新的远程仓库，并命名
$ git remote add [shortname] [url]
# 取回远程仓库的变化，并与本地分支合并
$ git pull [remote] [branch]
# 上传本地指定分支到远程仓库
$ git push [remote] [branch]
# 强行推送当前分支到远程仓库，即使有冲突
$ git push [remote] --force
# 推送所有分支到远程仓库
$ git push [remote] --all
```

9.撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]
# 恢复某个commit的指定文件到暂存区和工作区
$ git checkout [commit] [file]
# 恢复暂存区的所有文件到工作区
$ git checkout .
# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
$ git reset [file]
# 重置暂存区与工作区，与上一次commit保持一致
$ git reset --hard
# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
$ git reset [commit]
# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
$ git reset --hard [commit]
# 重置当前HEAD为指定commit，但保持暂存区和工作区不变
$ git reset --keep [commit]
# 新建一个commit，用来撤销指定commit
# 后者的所有变化都将被前者抵消，并且应用到当前分支
$ git revert [commit]
# 暂时将未提交的变化移除，稍后再移入
$ git stash
$ git stash pop
```

10.查找

```
# 从当前目录的所有文件中查找文本内容：
$ git grep "Hello"
# 在某一版本中搜索文本：
$ git grep "Hello" v2.5
# 生成一个可供发布的压缩包
$ git archive
```