开发工具使用总结

|  |
| --- |
| *Git*使用篇 |
| 设置用户名和邮箱地址：这是在*git*安装后要做的第一件事，否则不能进行*commit, push*等操作。 |
| 学会使用命令帮助手册：*git*自带帮助手册，里面有命令的详细用法，使用*git help <verb>* 或 *git <verb> --help* 查看。 |
| 将远程仓库克隆到本地：使用*git clone*命令，该命令同时会创建一个本地分支，由远程仓库的当前活跃分支检出。 |
| 将本地目录转换为仓库：使用*git init*命令，*git*仓库的本质是*.git*子目录，该子目录里存放着仓库的所有信息。 |
| 总结：安装了*git*之后，应立即配置用户名和邮箱地址，在本地引入*git*仓库，并知道如何使用命令对版本库进行操作。 |
|  |
| 了解什么是未跟踪文件：未跟踪文件通常是临时文件或不希望进入仓库的文件，通常我们会进行三种操作： |
| * 希望未跟踪文件在提交时直接排除，避免在提交时干扰我们选择文件，该功能通过*.gitignore*文件实现； |
| * 希望在清理工作区的时候，能快速删除全部的未跟踪文件，该功能由*git clean -df*命令实现； |
| * 希望一个已暂存文件，重新回归到未跟踪状态，该功能由*git rm –cached*命令实现； |
| 了解什么是已跟踪文件：已跟踪文件指已纳入版本管理，其状态又可以被分为已修改、已暂存和已提交。 |
| * 已修改：文件修改后未暂存，未提交； |
| * 已暂存：对未跟踪或已修改文件进行了暂存操作； |
| * 已提交：文件真正进入了仓库，称为了某个版本中的一员； |
| 了解什么是暂存区：暂存区位于工作目录和*git*仓库之间，未跟踪或已修改文件可以先放入暂存区，由暂存区统一提交（主要是 |
| 方便命令行操作，使用图形化界面通常会忽略暂存区）。 |
|  |
| 如何查看文件所处状态：*git status*命令，负责查看文件所处状态，能查看“未跟踪”、“已修改”或“已暂存”三种状态。 |
| 如何将文件放入暂存区：*git add*命令可以将“未跟踪”和“已修改”的文件放入暂存区，放入后文件状态变更为“已暂存”。 |
| 如何将文件提交到仓库：*git commit*命令，负责提交已暂存的文件，当然也可以越过暂存区，直接提交未跟踪和已修改的文件。 |
|  |
| 分支的底层原理是什么：分支本质上是一个指针，它永远指向这条分支的最后一个提交对象上。 |
| 分支的操作命令有哪些：*git branch*负责创建、删除和重命名分支；*git checkout*负责切换分支；*git merge*和*git rebase* |
| 负责合并分支；*git reset*负责移动分支指针。 |
| 分支切换时如何实现的：有一个*HEAD*指针，*HEAD*指针指向分支指针（指向谁，谁就是当前分支），切换分支的本质就是在改变 |
| *HEAD*指针的指向。 |
|  |
| *git checkout*命令有哪些用法： |
| * 从一个分支切换到另一个分支上（要求目标分支已存在） |
| * 创建新分支，并切换到新分支上（相当于*git branch + git checkout*在同时使用） |
| * 对一个远程分支使用*git checkout*，会创建对应的本地分支，并切换到该本地分支上（会自动创建追踪关系） |
|  |
| *git merge*如何对两个分支进行合并的： |
| * 若两个分支在同一条分支线上，当前分支只是进度落后，则执行快进（*fast-forward*），将当前分支直接向前移动 |
| * 若两个分支处在两条不同的分支线上，则将两条分支线合并，在合并处会产生一个新的提交对象，将当前分支移动到新的 |
| 提交对象上。（这种合并，可能会有代码冲突，需要手动解决冲突后才能继续合并） |
| *git rebase*如何对两个分支进行合并的： |
| * 相当于将当前分支移动到目标分支的后面 |
| * 优点是：没有多分支分叉问题，提交历史为一条直线 |
| * 缺点是：若发生冲突，需要连锁地解决后续产生的冲突 |
|  |
| 如果管理远程仓库：一个本地仓库上可以设置多个远程仓库，由*git remote*命令进行查看、新增、移除和重命名等操作。 |
| 如何管理远程分支：远程分支在本地以 *<remote>/<branch>* 的形式命名，在本地是无法通过*git reset*移动远程分支指针的。 |
| * 运行*git fetch*命令，会从远程仓库的拉取更新，并移动远程分支指针； |
| 如何在远程分支上进行工作： |
| *1、*对远程分支使用*git checkout*命令，检出对应的本地分支（会自动创建跟踪关系） |
| *2、*对本地分支使用*git pull*命令，会先更新远程分支（*git fetch*），然后合并远程分支（*git merge*） |
| *3、*对本地分支使用*git push*命令，会先向远程仓库推送本地分支上的更新，推送成功后更新远程分支指针 |
| 注：为了简化*git pull*和*git push*命令，需要为本地分支和远程分支建立跟踪关系（*pull*和*push*的跟踪关系相互独立） |
| 如何创建和删除远程仓库上的分支： |
| * 创建：需先有本地分支，然后使用*git push -u*命令向远程仓库推送（*-u*可以建立跟踪关系） |
| * 删除：使用*git pull --delete*命令即可在远程仓库上删除分支 |
| 附注：执行*git pull*命令，默认使用*git merge*进行合并，可以替换为*git rebase*合并；执行*git push*命令，要求本地必 |
| 须领先于远程仓库，否则不能推送 |
|  |
| 如果管理标签：在项目开发过程中，可以为某一个提交上设置标签，标签分为轻量标签和附注标签两种，由*git tag*命令实现。 |
| * 轻量标签：只是对提交对象的引用，如果提交对象被删除，轻量标签也会随着失效； |
| * 附注标签：是一个完整的对象，如果提交对象被删除，仍能使用附注标签检出所有的文件。 |
| * 附注标签使用场景：为项目设置版本号 |
| * 轻量标签使用场景：当临时使用时，例如给某个提交增加额外的解释说明，当后续不用时再删除掉； |
| 附注：对标签进行操作，仅本地生效，意味着使用*git push*推送代码时，标签不会被一同推向云端（当标签需要推送云端时，需 |
| 要手动推送，推送标签同样使用*git push*命令）。 |
| 附注：可以对标签执行*git checkout*命令，检出后在该标签上继续开发。 |
|  |
| 想要查看提交的历史记录：使用*git log*命令 |
| 将长命令用短命令替换掉：使用*git config –global alias*对原生命令起一个别名 |
|  |
| 常见的工作流有哪些： |
| * 分支工作流： |
| *1、*在远程仓库上建立属于自己的开发分支 |
| *2、*执行*git pull*，从远程仓库的公共开发分支上拉取最新的提交 |
| *3、*执行*git push*，将本地提交推送到远程仓库的个人开发分支上 |
| 4*、*使用*Merge Request*，在远程仓库上，将个人开发分支上的更新，合并到公共开发分支上 |
| * *Fork*工作流： |
| *1、fork*公共远程仓库，建立属于自己的远程仓库 |
| *2、*执行*git pull*，从公共远程仓库的开发分支上拉取最新的提交 |
| *3、*执行*git push*，将本地提交推送到个人远程仓库的开发分支上 |
| 4*、*使用*Merge Request*，将个人远程仓库上的更新，合并到公共远程仓库上 |
|  |
| 为什么推荐使用*SSH*而不是*HTTP*：*SSH*使用秘钥作为凭证（管理简单），*HTTP*使用账号和密码作为凭证（管理复杂） |
| 当前工作目录有修改，想要切换分支，该怎么做：使用*git stash*命令能将所有修改暂存，然后工作目录会被重置 |
| 如果快速清理未跟踪文件：使用*git clean -df*命令，但被*.gitignore*文件忽略的未跟踪文件不会被删除 |
| 如果快速清理未跟踪文件，包括被*.gitignore*文件忽略的未跟踪文件：使用*git clean -df -X*命令 |
| 对于未提交过的文件，希望忽略后续更改：将该文件加入*.gitignore*文件（*.gitignore*对已跟踪的文件，是不生效的） |
| 对于已提交过的文件，希望忽略后续更改：对这些文件使用*git update-index --assume-unchanged*命令 |
| 如何修改最后一次提交：使用*git commit -amend*命令，相当于用当前提交替换掉最后一次提交 |
| 想要删除掉一些错误的提交记录，且重置工作区的文件变更：使用*git reset -hard*命令 |
| 想要删除掉一些错误的提交记录，且保留工作区的文件变更：使用*git reset -soft*命令 |
| 使用*git reset*导致本地提交记录丢失，如何找回：使用*git reflog*查阅分支日志，寻找*reset* 操作前的最后一次提交，然后 |
| 再用*git reset*将分支指针重置到那次提交上。（如果时间久了，*git*会自动清理无效的提交对象，会导致无法恢复） |
| 想要对文件进行回滚，重置为某个提交的版本：对想要回滚的文件使用*git checkout*命令 |
| 附注：使用*git checkout*命令后，若*HEAD*不在分支指针上，就会出现分离头指针，这时后续 *commit* 无意义 |
| 想要撤销某次提交引发的文件变更：对那次提交使用*git revert*命令,会新建一个相反的提交，来抵消那次提交造成的影响 |
| 想将某个提交从一个分支复制到另一个分支：使用*git cherry-pick*命令 |
| 想要查看文件上每一行的历史修改记录：使用*git blame*命令 |
| 如果没有代码提交权限，该怎么做：生成补丁文件（*git format-patch*）,交由有提交权限的人打入补丁（*git apply*）后提交 |
|  |

|  |
| --- |
| *Maven*使用篇 |
| 定义模块和声明依赖时需要坐标，坐标由三个标签构成： |
| * *groupId*：一般由组织反向域名和项目名构成，例如*org.sonatype.nexus* |
| * *artifactId*：一般由项目名、中划线和模块名构成，例如*nexus-indexer* |
| * *version*：当前项目所处的版本 |
| 附注：对于子模块而言，*groupId*和*version*可由父模块继承，所以在定义子模块时通常不会设置这两个标签。 |
|  |
| 模块的打包格式有哪些，默认的打包格式有什么：打包格式用*packaging*标签定义，常见格式有*jar*和*war*，默认为*jar* |
|  |
| 什么时候需要声明依赖范围，常见类型有哪些：项目有编译、测试和运行三个阶段，如果一个依赖在某个阶段不需要，则应该使用 |
| *scope*标签排除掉。 |
| * *Juit*等测试组件，在运行阶段不需要，所以应该将*scope*声明为*test*，意指仅在编译和测试阶段引入 |
| * *sevlet-api*等组件，特点是运行环境已经提供了，所以应该将*scope*声明为*provided*，意指运行环境已提供，仅在编 |
| 译和和测试阶段引入（运行时不应重复引入，避免运行时版本冲突） |
| * *JDBC*驱动实现等组件，特点是接口和实现是分模块开发的，项目在编译阶段仅需要提供接口，不需要提供实现，所以应该 |
| 将接口的依赖包声明为*compile*，实现的依赖包声明为*runtime*，*runtime*意指在测试和运行阶段引入，编译阶段不引入 |
| 附注：依赖的默认范围为*compile*，指在编译、测试和运行三个阶段都引入该依赖。 |
| 附注：在运行时引入一个依赖，意指该依赖会被打入最终生成的*jar*包或*war*包中。 |
|  |
| 什么依赖冲突，又该如何解决依赖冲突：当前模块不仅会引入直接依赖，还会引入所有的间接依赖，间接依赖有可能会导致某个依 |
| 赖有重复且版本号不一致，这种冲突就是依赖冲突。面对依赖冲突，我们只能保留其中一个版本并删除其他版本，对直接依赖使用 |
| *exclude*标签即来完成对间接依赖的删除 |
|  |
| 如果项目间接引入了某个依赖，该依赖的版本不符合要求，该如何解决：使用*exclude*标签排除掉该依赖，并在模块中重新引入, |
| 在引入时声明需要的版本即可。 |
|  |
| 如果当前模块下有一个依赖，该依赖仅作用于当前模块本身，若其他模块引入当前模块，不希望这个依赖被间接传递过去：在当前 |
| 模块引入这个依赖时，可以添加*<optional>true</optional>*标签，添加后依赖就不会向上传递 |
|  |
| 项目开发是如何为依赖注入版本的：在*properties*标签中声明版本号，在依赖的*version*标签中使用*${}*引用代表版本号的变量 |
|  |
| 通过坐标，可以快速定位依赖在仓库中的位置，查找径路为*groupId/artifactId/version/artifactId-version.packaging* |
|  |
| 如何配置依赖的下载和上传仓库和镜像仓库： |
| * 下载仓库：即可以配置在*settings.xml*里，也可以配置在*pom.xml*里，配置在它们的*profile*标签下 |
| 下载仓库分为依赖下载仓库和构建下载仓库，分别配置在*repository*和*pluginRepository*标签下，下载仓库不区分发 |
| 布版本和快照版本； |
| * 上传仓库：只能配置在*pom.xml*里的*profile*标签下 |
| 上传仓库分为发布版本仓库和快照版本仓库，分别配置在*repository*和*snapshotRepository*标签下，上传仓库不区分 |
| 依赖仓库和插件仓库； |
| * 镜像仓库：只能配置在*settings.xml*里的*mirror*标签下 |
| 镜像仓库用于屏蔽特定的下载仓库，当访问被镜像的仓库后，会直接跳转到镜像仓库的*url*地址 |
| 配置额外的仓库（下载、上传、镜像），通常需要配置该仓库的访问凭证，例如用户名和密码，访问凭证为确保安全只能配置在 |
| *settings.xml*文件里，使用*server*标签进行配置 |
|  |
| 什么是快照版本，快照版本和发布版本有什么区别： |
| * 快照版本（*SNAPSHOT*）：是开发测试阶段输出的不稳定版本，临时性的，版本内容会随着后续开发不断更新； |
| * 发布版本（*RELEASE*）：是对外提供的较为稳定的版本，版本内容不会再更新，即使有功能迭代或漏洞修复，也只会在下 |
| 一个版本中进行； |
|  |
| *Maven*在每次编译时，是否都会从远程仓库下载依赖，对依赖是如何解析下载的： |
| * 对于显示版本，例如*1.2、2.1-beta-1*，如果仓库里有，则直接使用，不会再访问远程仓库 |
| * 对于*RELEASE*、*LATEST*、*xxx-SNAPSHOT*版本，会基于更新策略，从远程仓库中下载最新版本 |
| *RELEASE*指最新的发布版本，*LATEST*指最新的发布版本或快照版本，*xxx-SNAPSHOT*指当前快照版本 |
| 仓库在存放快照版本时，会带上时间戳，在实际使用时根据时间戳拿最新的快照版本并转为非时间戳格式 |
| 所谓的更新策略，指在配置远程仓库时，会使用*updatePolicy*标签，配置检查依赖或插件的更新频率 |
|  |
| 为什么有些插件需要指定版本，有些则不需要：核心插件，*Maven*已经设定了版本，如果声明时不指定版本，就会使用该版本；对 |
| 于其他插件，如果不指定版本，就会使用*RELEASE*版本（但*RELEASE*本身属于不稳定版本） |
|  |
| *Maven*有三套相互独立的生命周期，分别为*clean*、*default*和*site*，且每个生命周期又有多个阶段 |
| *Maven*可通过命令行调用生命周期的某个阶段，对项目进行不同程度的构建，常用的命令及对应的阶段有 |
| * *maven clean：*负责清理上一次构建生成的*target*目录 |
| * *maven compile：*编译项目的主源码 |
| * *maven test：*执行项目的测试用例 |
| * *maven package：*对项目进行打包 |
| * *maven install：*将包安装到本地仓库 |
| * *maven deploy：*将包推送到远程仓库 |
| *clean*属于*clean*生命周期，而*compile*、*test*、*package*、*install*和*deploy*都属于*default*生命周期 |
| * 同一个生命周期内，后面的阶段依赖于前面的阶段，意味着执行*package*阶段，会先执行*compile*和*test*阶段 |
| * 不同的生命周期相互独立，意味着想要同时执行*clean*和*package*阶段，执行的命令应该是*mvn* *clean* *package* |
| 声明周期是与插件的目标绑定的，调用生命周期，本质上是执行插件的目标，例如 |
| * 插件目标*maven-compiler-plugin:compile*与生命周期*compile*绑定 |
| * 插件目标*maven-surefire-plugin:test*与生命周期*test*绑定 |
| * 插件目标*maven-jar-plugin:jar*与生命周期*package*绑定 |
| 插件的目标除了通过调用生命周期的被动执行，还有命令行的主动执行，例如*mvn dependency:tree* |
|  |
| 使用*Maven*构建项目时，文件的布局应该遵循以下阅约定： |
| * *src/main/java：*存放项目主源码 |
| * *src/main/resources：*存放主源码的配置文件 |
| * *src/main/webapp：*存放*web*资源，例如*WEB-INF、html、css、js、img*等文件 |
| * *src/test/java：*存放单元测试源码 |
| * *src/test/resources：*存放单元测试源码的配置文件 |
| * *pom.xml：*项目的配置信息 |
| * *target：*存放项目的输出结果 |
| 模块中包的目录结构，通常会与*groupId*对应，例如*org.sonatype.nexus*对应包为*org/sonatype/nexus* |
|  |
| 模块中的依赖和插件，怎么在版本和配置上做统一管理： |
| * 版本统一管理：用*repositories*标签自定义变量，将所有依赖和插件的版本声明在一起，实际使用时用*${}*引用 |
| * 配置统一管理：用*dependencyManagement*和*pluginMangement*下完成依赖和插件的配置，在引入依赖和插件时只需要 |
| 声明*groupId*和*artifactId*即可 |
| *dependencyManagement*和*pluginMangement*标签只有声明的作用，并不会引入这些依赖或插件的包，只有在*dependencies*和 |
| *plugins*标签下声明，才会真正将包引入到模块中 |
|  |
| 什么是模块的继承，继承能带来什么好处：通过继承自同一父模块，可以对所有子模块的做统一管理 |
| * 通过继承*properties*、*dependencyManagement*、*pluginMangement、dependencies*、*pulgins*标签，实现对依赖和 |
| 插件的统一管理 |
| * 通过继承*repositories, distributionManagement*标签，实现对仓库的统一配置管理 |
| * 通过继承*groupId、version*，所有模块只需要声明*artifactId*，能够自动统一*groupId*和*version* |
|  |
| 什么是模块的聚合，聚合能带来什么好处：聚合被用于快速构建项目（在父模块上执行构建命令，会作用到所有子模块上） |
| * 在模块被聚合后，可以通过*-pl*、*-am*、*-amd*、*-rf*等选项，手动选择需要的子模块进行构建，而非构建所有子模块 |
|  |
| 怎么搭建父子模块： |
| *1、*首先搭建父模块，父模块仅有一个*pom.xml*文件，且包方式为*pom*，用*module*标签声明子模块所在位置 |
| *2、*然后搭建子模块，父模块用*parent*标签声明父模块的坐标，使用*relativePath*子标签声明父模块所在位置 |
| 备注：*relativePath*子标签在*parent*标签下，默认值为*../pom.xm*，在采用层级目录结构时可以不用配置。 |
|  |
| *SpringBoot*的配置原理：子模块会优先从*relativePath*中定位父模块的位置，如果找不到，会从仓库中检索。 |
| *SpringCloud*的配置原理：如果当前工程已经有父工程了，可以在*dependencyManagement*标签中引入依赖，并将依赖的*scope*标签声明为*import*，该操作会将依赖中的*dependencyManagement*与当前模块的*dependencyManagement*进行合并。 |
|  |
| 测试类和测试方法的命名标准：测试类通常以*Test*结尾，测试方法通常以*test*开头 |
| *test*阶段在*compile*之后，*package*之前，为了快速打包和运行代码，通常需要跳过测试，跳过的方式有哪些 |
| * 命令行构建时增加参数，例如*mvn package -DskipTests*（临时跳过） |
| * 在*IDE*上构建时，操作界面会提供跳过测试的按钮（临时跳过） |
| * 配置*maven-surefire-plugin*插件，无论采用何种方式构建都会跳过测试阶段 |
| *maven-surefire-plugin*插件还可以在测试阶段，指定包含或指定排除测试用例，输出测试结果、代码覆盖率等测试报表 |
|  |
| 在*maven*上如何进行多环境配置： |
| * 设置多个*profile*标签，每个*profile*都可以定义一组*properties*属性 |
| * 在*activation*标签中定义默认激活的*profile*，或者在构建时使用*-P*选项指定*profile*激活 |
| * 在项目资源文件中使用*${key}*引用自定义属性，并配置插件开启资源过滤，资源过滤可以在编译前将资源文件中的占位符 |
| 替换为对应的属性值 |
| *maven*的*${key}*不仅可以引用自定义属性，还可以引用到系统环境变量和各类项目属性 |
|  |
| *Maven install*在什么场景会用到：多模块存在依赖关系，首先应将被依赖模块安装到本地仓库中 |
| 运行一个服务，前置包括哪些阶段：服务在运行前，会先经过编译、测试、打包这三个阶段 |
| 子模块如何替换父模块中已声明的依赖：在子模块中重新引入即可，子模块中声明的依赖，其优先级高于父模块 |
| 默认打包格式是不可执行的，如何创建可执行*jar*包：可执行*jar*包要求在包中的*META-INF/MANIFEST.MF*中指定*Main-Class* |
| （程序入口），可执行*jar*包可以通过以下插件实现 |
| * *maven-jar-plugin + maven-dependency-plugin* |
| * *maven-shade-plugin* |
| * *maven-assembly-plugin*(推荐) |

对方法进行提炼

如果需要花时间浏览一段代码才能弄清楚它到底在干什么，那么就应该将其提炼到一个函数中。

有许多短函数，甚至是一行的方法委托 方法的参数不应该超过4个

|  |
| --- |
| if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd)) |
| charge = quantity \* plan.summerRate; |
| else |
| charge = quantity \* plan.regularRate + plan.regularServiceCharge; |
|  |
| 如果条件逻辑需要花费一段时间才能理解，建议提炼条件判断和条件分支为函数 |
| function summer() { |
| return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd); |
| } |
|  |
| function summerCharge() { |
| return quantity \* plan.summerRate; |
| } |
|  |
| function regularCharge() { |
| return quantity \* plan.regularRate + plan.regularServiceCharge; |
| } |
|  |
| if (summer()) |
| charge = summerCharge(); |
| else |
| charge = regularCharge(); |
|  |
| 如果*if-else*仅对变量进行赋值，建议使用三元运算符进行简化 |
| charge = summer() ? summerCharge() : regularCharge(); |

|  |
| --- |
| if (anEmployee.seniority < 2) return 0; |
| if (anEmployee.monthsDisabled > 12) return 0 |
| if (anEmployee.isPartTime) return 0; |
|  |
| if (anEmployee.onVacation) |
| if (anEmployee.seniority > 10) |
| return 1; |
|  |
| 如果多条分支的处理逻辑相同，建议合并条件表达式并提炼为函数，如果检查条件的确彼此独立，则不应该合并 |
|  |
| 平行的分支，在合并时使用逻辑或进行合并 |
| if (isNotEligibleForDisability()) return 0; |
|  |
| function isNotEligibleForDisability() { |
| return ((anEmployee.seniority < 2) |
| || (anEmployee.monthsDisabled > 12) |
| || (anEmployee.isPartTime)); |
|  |
| 嵌套的分支，在合并时使用逻辑与进行合并 |
| if (isEligableForVacation()) return 1; |
|  |
| function isEligableForVacation() { |
| return (anEmployee.onVacation |
| && (anEmployee.seniority > 10)); |
| } |

|  |
| --- |
| function getPayAmount() { |
| let result; |
| if (isDead) |
| result = deadAmount(); |
| else { |
| if (isSeparated) |
| result = separatedAmount(); |
| else { |
| if (isRetired) |
| result = retiredAmount(); |
| else |
| result = normalPayAmount(); |
| } |
| } |
| return result; |
| } |
|  |
| 条件分支可以分为两类，一类是两个条件分支都属于正常，另一类是：一个条件分支是正常行为，另一个分支是异常情况。 |
| 异常分支可以使函数结束调用，具体表现为无任何执行语句，或执行结束立即返回，或执行过程抛出异常 |
| 当出现异常分支建议使用卫语句重构整个条件逻辑，重构后有分支相同的卫语句则合并条件表达式 |
|  |
| function getPayAmount() { |
| if (isDead) return deadAmount(); |
| if (isSeparated) return separatedAmount(); |
| if (isRetired) return retiredAmount(); |
| return normalPayAmount(); |
| } |

额

如果表示是不是，用is+...。  
如果是形容词就直接拿来用，可以加is也可以不加。比如isYoung、isSimple、isNaive可以直接写成young、simple、naive，还有各种...able的词前面也没必要加is。但是如果这个形容词有常用的做动词的含义，那就要加is，比如empty这个词可以作动词表示清空的意思，那么表示是否为空就写成isEmpty而不是empty。

2. 表示有没有，用has+...。  
3. 表示能不能，用can+...。  
4. 不要用flag。不要用flag。不要用flag。flag应该搭配的是enum而不是boolean。

变量命名实践

抽象层次上，导致仅仅的委托调用