**创建版本库**

第一步、创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录。

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

**$ git init；**

第三步、用命令 git add 告诉Git，把文件添加到仓库：

**$ git add readme.txt**

第四步、用命令 git commit 告诉Git，把文件提交到仓库：

**$ git commit -m "wrote a readme file"**

简单解释一下 git commit 命令， -m 后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

总结：初始化一个Git仓库，使用 git init 命令。

添加文件到Git仓库，分两步：

• 第一步，使用命令 git add ，注意，可反复多次使用，添加多个⽂文件；

• 第二步，使用命令 git commit ，完成。

**Git穿梭**

要随时掌握工作区的状态，使⽤用 git status 命令。

• 如果 **git status** 告诉你有文件被修改过，用 **git diff** 可以查看修改内容。

HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使⽤用命

令 **git reset --hard commit\_id** 。

• 穿梭前，用 git log 可以查看提交历史，以便确定要**回退**到哪个版本。

• 要重返未来，用 git reflog 查看命令历史，以便确定要回到**未来**的哪个版本。

**工作区和暂存区**

**Git和其他版本控制系统如SVN的⼀一个不同之处就是有暂存区的概念**

工作区（Working Directory）：就是你在电脑里能看到的目录，比如learngit文件夹

就是一个工作区。

版本库（Repository）：工作区有一个隐藏目录“.git”，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还

有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD

**管理修改**

为什么Git比其他

版本控制系统设计得优秀，因为Git跟踪并管理的是修改，⽽而非文件。你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，

更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

**撤销修改**

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令

**Git checkout -- file** 。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令 **git reset HEAD file** ，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

删除：

命令 **git rm** 用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

**远程仓库**

要关联一个远程库，使用命令

**git remote add origin git@server-name:path/repo-name.git** ；

关联后，使用命令 git push -u origin master 第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令 git push origin master 推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联⺴网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了。

要克隆一个仓库，首先必须知道仓库的地址，然后使用 git clone 命令克隆。

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

**分支管理**

**Git鼓励大量使用分支：**

**查看分支：git branch**

**创建分支：git branch name**

**切换分支：git checkout name**

**创建+切换分支： git checkout -b name**

**合并某分支到当前分支： git merge name**

**删除分支： git branch -d name**

**解决冲突**

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用 **git log --graph** 命令可以看到分支合并图

**分支管理策略**

通常，合并分支时，如果可能，Git会用“**Fast forward**”模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁⽤用“Fast forward”模式，Git就会在merge时生成⼀一个新的commit，这样，从分支历史上就可以看出分支信息。

**分支策略**

在实际开发中，我们应该按照几个基本原则进⾏行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，⽐比如

1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上 **--no-ff** 参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，⽽而fast forward合并就看不出来曾经做过合并。

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场 **git stash** ⼀一下，然后去修复bug，修复后，再 git stash pop ，回到工作现场。

**Feature分支**

软件开发中，总有无穷无尽的新的功能要不断添加进来。

添加⼀个新功能时，你肯定不希望因为⼀些实验性质的代码，把主分支搞乱了，所以，每添加⼀个新功能，最好新建⼀个feature分支，在上面开发，完成后，合并，最后，删除该feature分支

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分⽀支，可以通过 **git branch -D name** 强⾏行删除。

**多人合作**

1. 首先，可以试图用 **git push origin branch-name** 推送自己的修改；

2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 **git pull** 试图合并；

3. 如果合并有冲突，则解决冲突，并在本地提交；

4. 没有冲突或者解决掉冲突后，再用 **git push origin branch-name** 推送就能成功！

如果 git pull 提示“no tracking information”，则说明本地分支和远程支的链接关系没有创建，用命令 **git branch --set-upstream branch-name origin/branch-name 。**

这就是多人协作的工作模式，⼀一旦熟悉了，就非常简单

**查看远程库信息，使⽤用 git remote -v** ；

**• 本地新建的分支如果不推送到远程，对其他人就是不可⻅见的；**

**• 从本地推送分支，使用 git push origin branch-name ，如果推送失败，先用git pull抓取远程的新提交；**

**• 在本地创建和远程分支对应的分支，使用**

**git checkout -b branch-name origin/branch-name ，本地和远程分支的名称最好⼀一致；**

**• 建⽴立本地分支和远程分支的关联，使用**

**git branch --set-upstream branch-name origin/branch-name ；**

**• 从远程抓取分支，使用 git pull ，如果有冲突，要先处理冲突。**

**让Git显⽰示颜⾊色，会让命令输出看起来更醒目：**

**$ git config --global color.ui true**