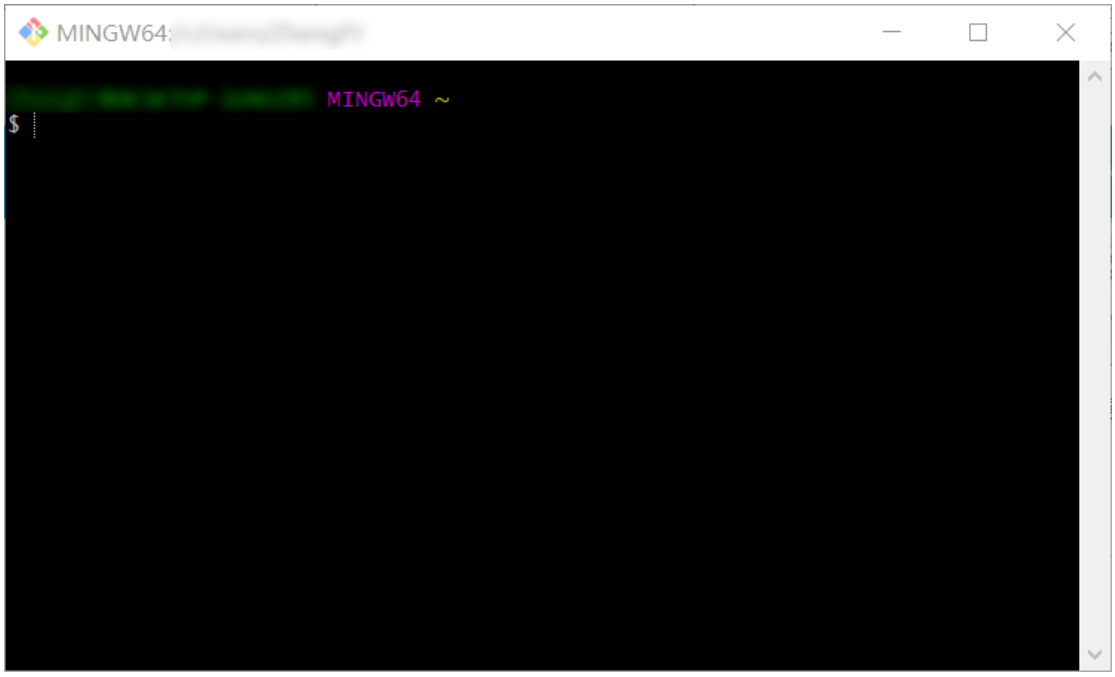
GIT 学习笔记

--（Window 版本）

--（廖雪峰官方网站 https://www.liaoxuefeng.com/学习笔记）

一、安装 Git

从 https://git-scm.com/downloads 下载相关的 windows 版本的 git 安装包，
自行安装即可
安装完成后，在开始菜单里找到“Git”→”Git Bash”，出现下图即表示安装成功。



安装完成后，还需要最后一步设置，在命令行输入：

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意 `git config` 命令的 `--global` 参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

二、创建版本库

在自己电脑的合适位置，创建版本库。我是创建在 e 盘的
用 cd /e 进入 E 盘
`mkdir` -->文件夹名
pwd -->可以显示文件夹的路径
`git init` -->可以把当前目录变成 Git 可以管理的仓库。

把仓库中隐藏的.git 文件夹删掉即可移除仓库。

三、创建文件

在文件库下创建文件 readme.txt，并在其中添加某些内容

`git status` -->可以让我们时刻掌握仓库当前的状态
`git diff readme.txt` -->看文件修改的信息

`git add readme.txt` -->添加文件
`git commit -m "add distributed"` -->提交文件，双引号内为修改的说明内容

四、版本回退

查看版本信息`git log`或者`git log --pretty=oneline`

首先，Git必须知道当前版本是哪个版本，在Git中，用`HEAD`表示当前版本，也就是最新的提交`3628164...882e1e0`（注意我的提交ID和你的肯定不一样），上一个版本就是`HEAD^`，上上一个版本就是`HEAD^^`，当然往上100个版本写100个`^`比较容易数不过来，所以写成`HEAD~100`。

现在，我们要把当前版本"append GPL"回退到上一个版本"add distributed"，就可以使用`git reset`命令：

`git reset --hard head^` --->(head^表示上一个版本，head~100 表示上的第 100 个版本，也可以写 id)

`cat readme.txt` -->可以查看文件的内容

回到以前版本后，再重返未来，`git reset --hard` 未来版本的 `id`（写前几个字符即可，不能太短，否则不起作用）

小结

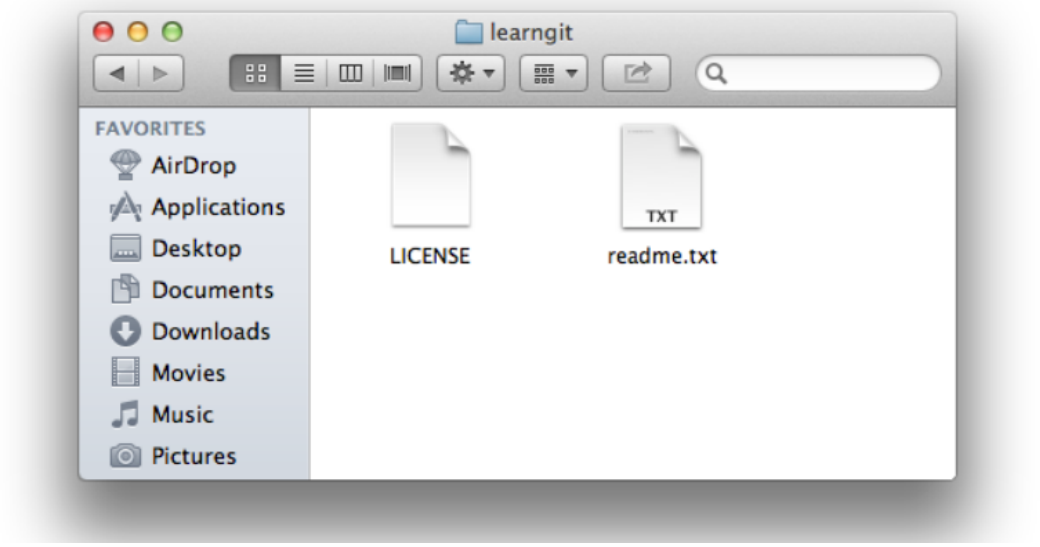
现在总结一下：

- `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。

五、工作区和暂存区

工作区（Working Directory）

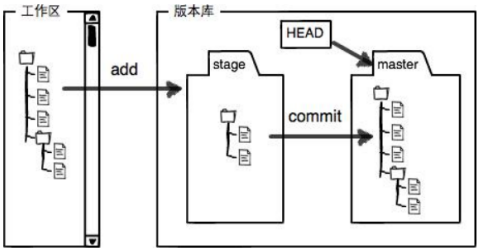
就是你在电脑里能看到的目录，比如我的`learngit`文件夹就是一个工作区：



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`。



分支和`HEAD`的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

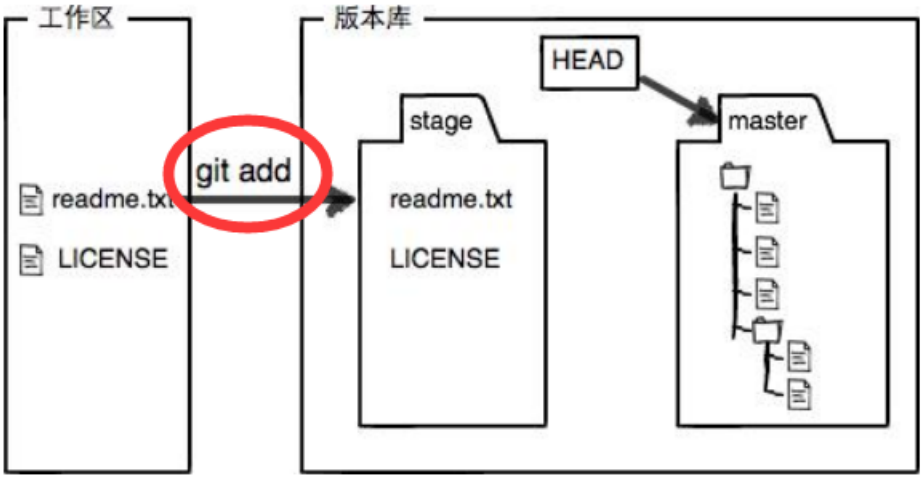
第一步是用`git add`把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用`git commit`提交更改，实际上就是把暂存区的所有内容提交到当前分支。

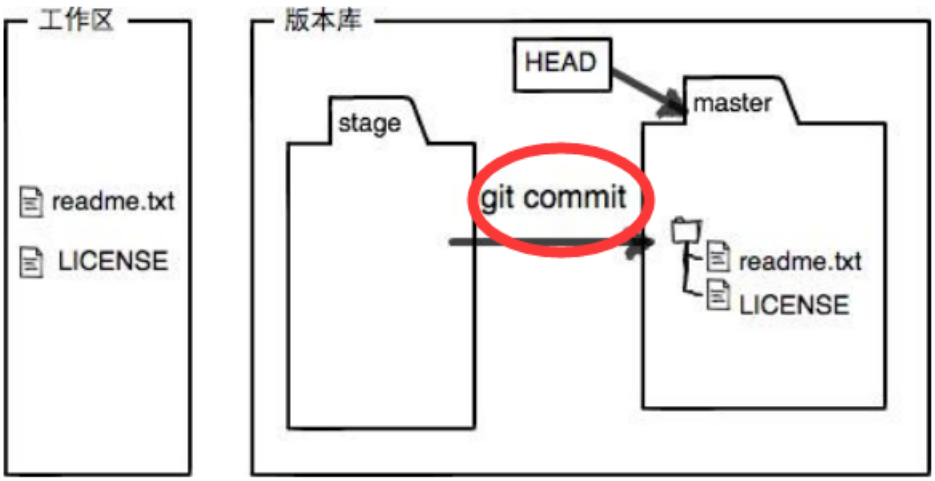
因为我们创建Git版本库时，Git自动为我们创建了唯一一个`master`分支，所以，现在，`git commit`就是往`master`分支上提交更改。

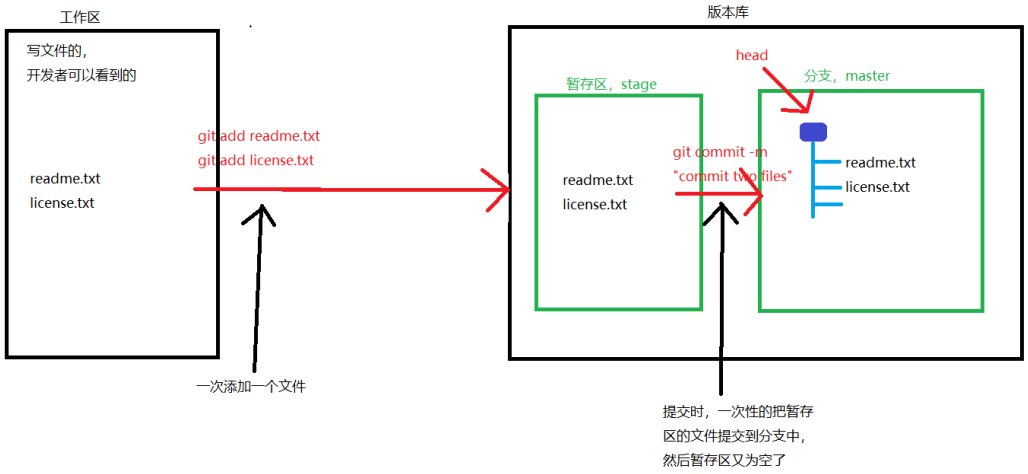
你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

现在，暂存区的状态就变成这样了：



现在版本库变成了这样，暂存区就没有任何内容了：



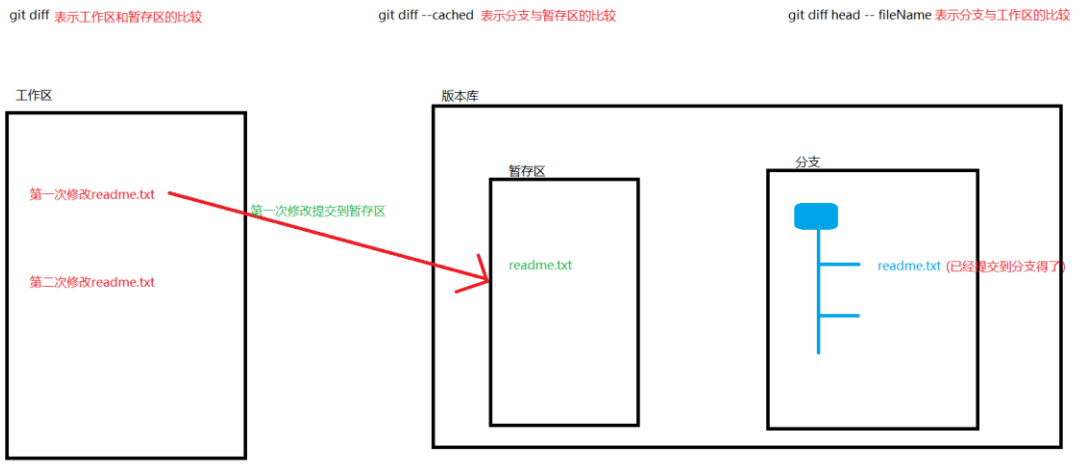


六、管理修改

Git 管理的是修改，而不是文件。
只有通过 add 命令添加到暂存区的文件通过 commit 命令才会真正的被提交

若有两次修改，第一次修改后，通过 add 添加到暂存区，此时又对文件进行第二次修改，但没有提交到暂存区，此刻若 commit 提交，则第一次的修改会被提交，第二次修改不会被提交。

所以可以得出结论：
git diff 是工作区和暂存区的对比
git diff -- cached 是暂存区和分支的对比
git diff HEAD -- readme.txt 工作区和分支的对比



七、撤销文件

`git status` -->可以查看当前版本状态
可以发现有其他的命令提示
`git checkout -- fileName` -->放弃工作区的修改

你可以发现，Git会告诉你，`git checkout -- file`可以丢弃工作区的修改：

```
$ git checkout -- readme.txt
```

命令`git checkout -- readme.txt`意思就是，把`readme.txt`文件在工作区的修改全部撤销，这里有两种情况：

一种是`readme.txt`自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是`readme.txt`已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次`git commit`或`git add`时的状态。

`git checkout -- file`命令中的`--`很重要，没有`--`，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到`git checkout`命令。

小结

又到了小结时间。

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。

八、删除文件

删除本地的文件：`rm fileName`

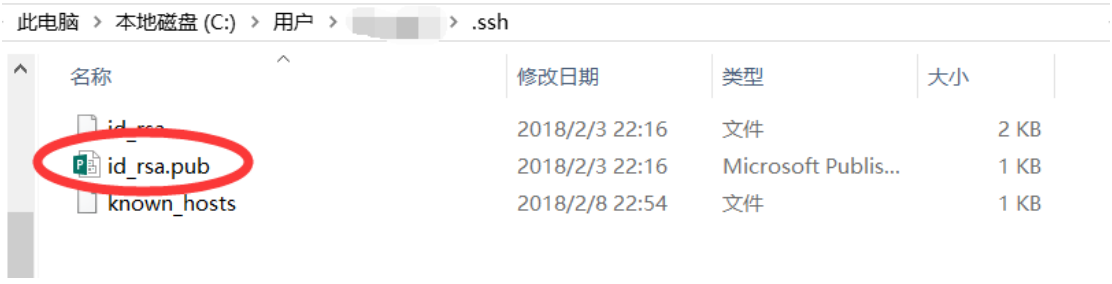
用 git status 命令可以查看那个文件被删除了

从版本库中删除文件：`git rm fileName`，然后`git commit`

误删后，可以从版本库中恢复到本地：`git checkout -- fileName`

九、远程仓库

在本地 git 上创建 SSH Key，在红色标记的文件中复制 ssh key



在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有`id_rsa`和`id_rsa.pub`这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

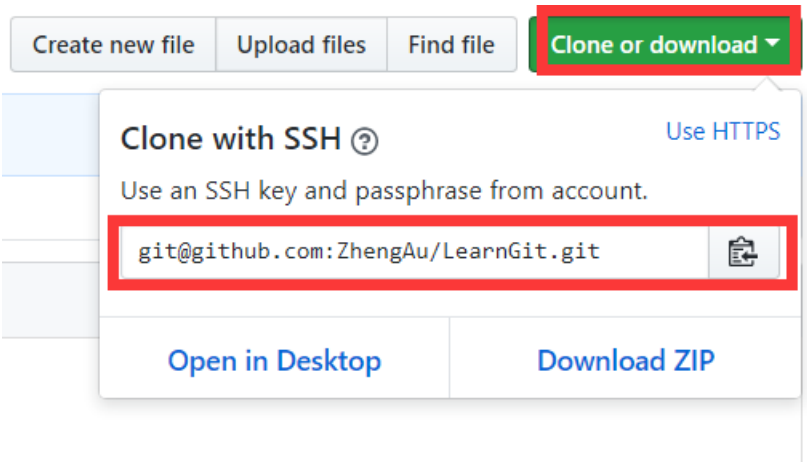
如果一切顺利的话，可以在用户主目录里找到`.ssh`目录，里面有`id_rsa`和`id_rsa.pub`两个文件，这两个就是SSH Key的秘钥对，`id_rsa`是私钥，不能泄露出去，`id_rsa.pub`是公钥，可以放心地告诉任何人。

注册 GitHub 账号，并添加 ssh key

在 GitHub 中创建一个空的 repository，不能创建.md 文件

在本地的要关联的仓库下运行命令：

```
git remote add origin git@github.com:ZhengAu/LearnGit.git
```



把本地库的内容推送到远程，用`git push`命令，实际上是把当前分支`master`推送到远程。

由于远程库是空的，我们第一次推送`master`分支时，加上了`-u`参数，Git不但会把本地的`master`分支内容推送的远程新的`master`分支，还会把本地的`master`分支和远程的`master`分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：

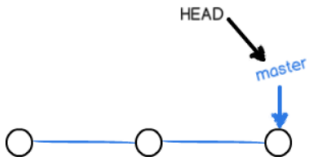
十、分支管理

10.1 创建与合并分支

理解 master、head、分支，其实就是指针

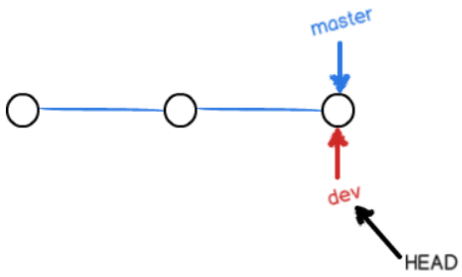
在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即`master`分支。`HEAD`严格来说不是指向提交，而是指向`master`，`master`才是指向提交的，所以，`HEAD`指向的就是当前分支。

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：



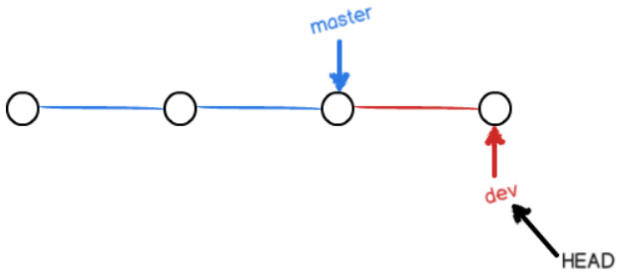
每次提交，`master`分支都会向前移动一步，这样，随着你不断提交，`master`分支的线也越来越长：

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：

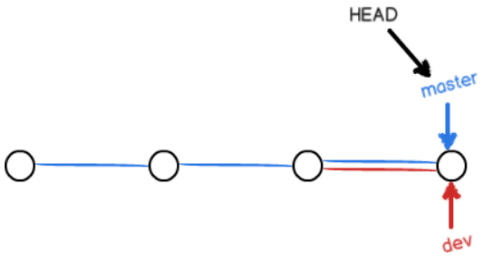


你看，Git创建一个分支很快，因为除了增加一个`dev`指针，改改`HEAD`的指向，工作区的文件都没有任何变化！

不过，从现在开始，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：

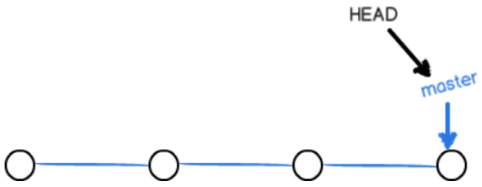


假如我们在`dev`上的工作完成了，就可以把`dev`合并到`master`上。Git怎么合并呢？最简单的方法，就是直接把`master`指指向`dev`的当前提交，就完成了合并：



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，我们就剩下了一条`master`分支：



命令

`git checkout -b branchName --→` -b 参数表示创建分支并切换的新分支

`git branch branchName  --→`创建分支
`git checkout branchName --→`切换分支
`git branch --→`查看当前分支(前面有*的表示当前分支)

`git merge branchName` -->在当前分支中合并另一个分支 branchName

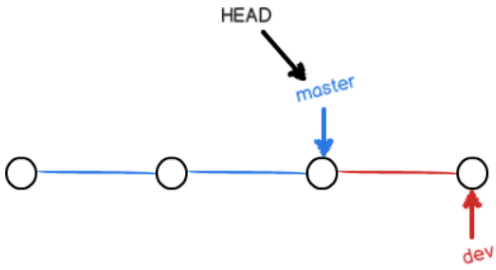
`git merge`命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和`dev`分支的最新提交是完全一样的。

注意到上面的`Fast-forward`信息，Git告诉我们，这次合并是“快进模式”，也就是直接把`master`指向`dev`的当前提交，所以合并速度非常快。

`Git branch -d branchName` -->删除分支

在当前分支中修改文件，并添加和提交，
此时切换到另一个分支，刚才修改的文件的内容不见了，因为在不同的分支上。需要合并两个分支后，刚才修改的内容在两个分支中才可见。

切换回`master`分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在`dev`分支上，而`master`分支此刻的提交点并没有变：



小结

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>`

10.2 解决冲突

Git用`<<<<<<<`，`=======`，`>>>>>>>`标记出不同分支的内容

小结

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。

10.3 分支管理策略

分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

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。

所以，团队合作的分支看起来就像这样：

小结

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`fast forward`合并就看不出来曾经做过合并。

10.4 Bug 分支

Git还提供了一个`stash`功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

你可以多次stash，恢复的时候，先用`git stash list`查看，然后恢复指定的stash，用命令：

```
$ git stash apply stash@{0}
```

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用`git stash apply`恢复，但是恢复后，stash内容并不删除，你需要用`git stash drop`来删除；

另一种方式是用`git stash pop`，恢复的同时把stash内容也删了：

小结

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`一下，然后去修复bug，修复后，再`git stash pop`，回到工作现场。

十一、Git 重命名操作

重命名：`git mv oldName newName`

查看状态：`git status -s`

提交：`git commit -a -m “”`

对于提交操作，需要使用 `-a` 标志，这使 `git commit` 自动检测修改的文件。