Git总结

git config --global user.name "tancong_i"

git config --global user.email "tancong_i@didiglobal.com"

git init

使当前目录成为一个git仓库，建成之后在当前目录下就有了一个文件夹.git

把一个文件放到Git仓库需要两步

1. git add readme.txt. 告诉Git，把文件添加到仓库
2. git commit -m "write a readme file"　告诉Git，把文件提交到仓库

　　git commit命令执行成功后告诉你，`1 file changed`：一个文件被改动；`2 insertions`：插入了两行内容

为什么Git添加文件需要`add`、`commit`两步？因为commit一次可以提交很多文件

git status　仓库目前的状态

git diff readme.txt　当前文件readme.txt和上次提交的有什么差别

git log 查看git的历史记录，从最近到最远的提交日志

git的版本号是经过SHA1计算出来的一个很大的16进制数

`HEAD`表示当前版本

`HEAD^`上一个版本

`HEAD^^`上上一个版本

`HEAD~100`往上100个版本

git reset --hard HEAD^　回退到上一个版本

再回到未来的版本，只需要找到那个版本的版本号，不必须写全

git reset --hard 45d9

Git版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD移动

git reflog

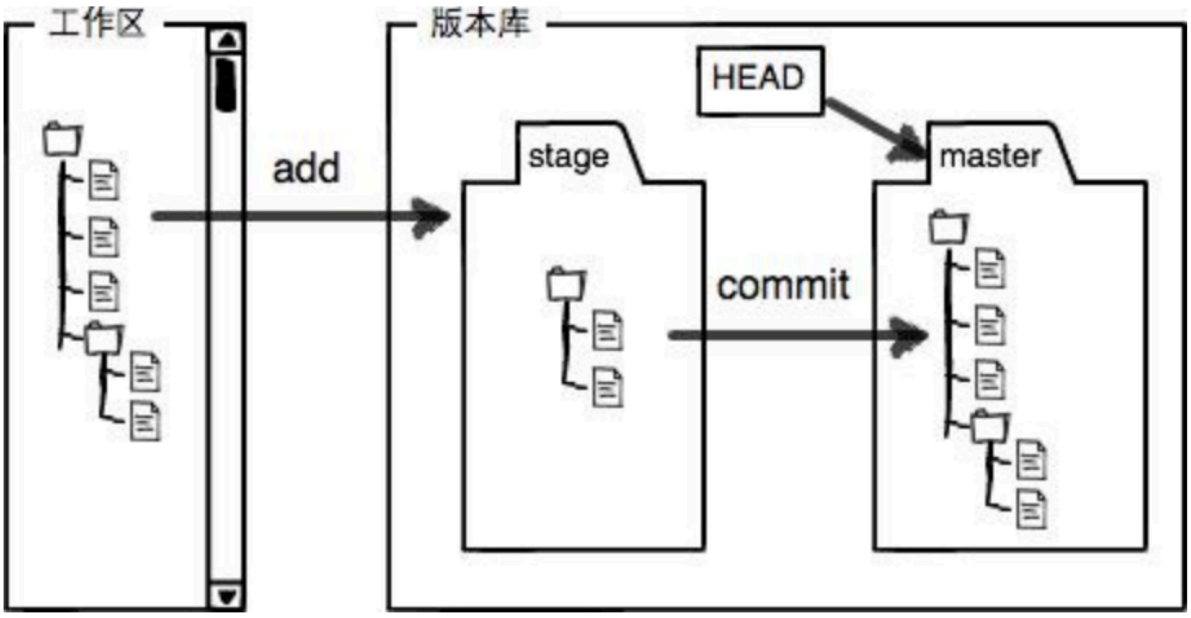
记录git的每一次命令

工作区

就是你在电脑里能看到的目录

版本库

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 add将文件添加到暂存区；

git commit 将暂存区的所有内容提交到当前分支

在创建Git版本库时，Git自动创建了一个唯一的`master`分枝；

需要提交的文件修改通通存放到暂存区，然后，一次性提交暂存区的所有修改。

git diff HEAD -- readme.txt

可以查看工作区和版本库里面最新版本的区别

git checkout -- file

丢掉工作区里面的内容

1. **readme.txt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**；

2. **readme.txt已经添加到赞存区，又做了修改，现在，撤销修改就回到添加到咱存区后的状态**；

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 reset HEAD

可以把暂存区的修改撤销掉（unstage），重新放回工作区

即，把已经存到暂存区的修改又返回到了工作区

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

git rm

先在工作区将对应的file删除

rm file

再从版本库中删掉该文件，并且git commit

如果误删了文件，但是版本库里面还有，可以很轻松地把误删的文件恢复到最新版本

git checkout -- test.txt

远程仓库

```
ssh-keygen -t rsa -C "tancong_i@didglobal.com"
```

产生.ssh文件夹及其下的两个文件：id_rsa和id_rsa.pub两个文件，一个私钥，一个公钥

将公钥放到Github上。

为什么GitHub需要公钥呢?

因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

将本地仓库和远程仓库关联

```
git remote add origin git@github.com:M000M/learngit.git
```

添加后，远程仓库的名字就是**origin**，这是Git的默认叫法

下一步就可以把本地仓库的所有内容推送到远程库上：

```
git push -u origin master
```

把本地仓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。

由于远程仓库是空的，我们第一推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送到远程新的master分支，还会把本地的master分支和远程的master分支关联起来，再以后的推送或则拉取时就可以简化命令。

从现在起，只要本地做了提交，就可以通过命令

```
git push origin master
```

把本地master分支的最新推送到GitHub，现在，你就拥有了真正的分布式版本库了。

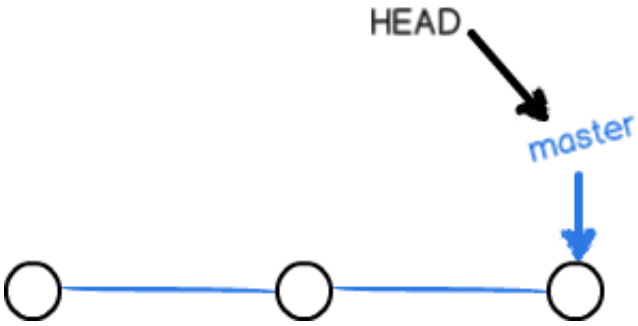
git支持多种协议，默认的git://使用SSH，但也可以使用https等其他协议。

使用https除了慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是再某些只开放http端口的公司内部就无法使用ssh协议只能用https。

Git分支管理

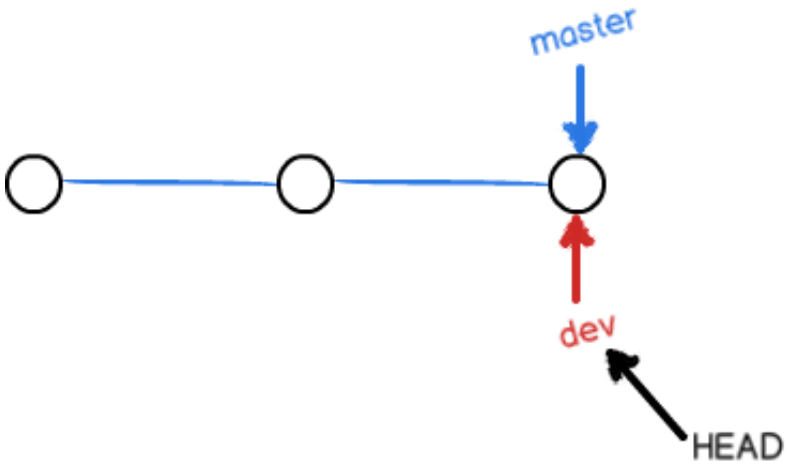
master才是只想提交的，HEAD只想的就是当前分支。

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。



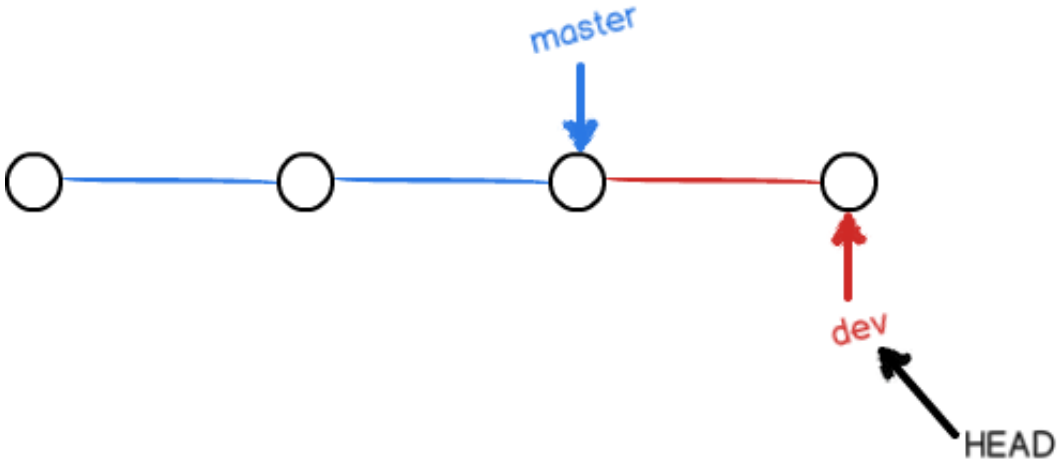
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

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，就表示当前分支再dev上：



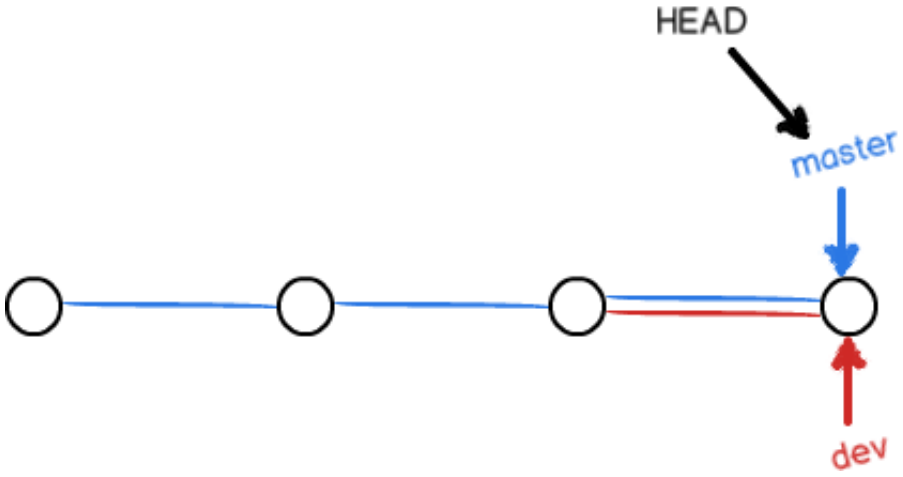
Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变换。

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



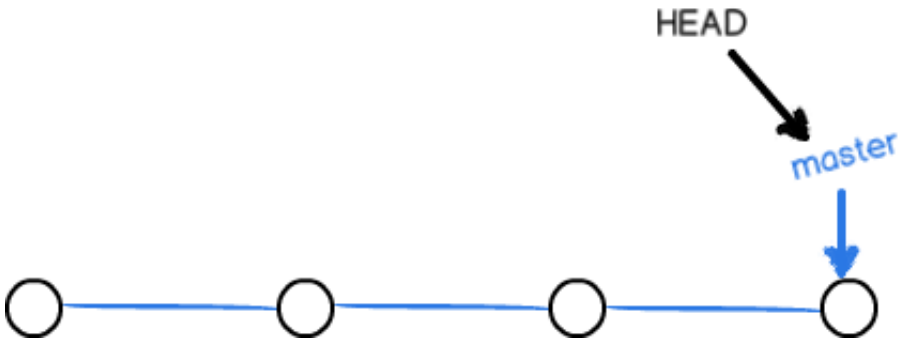
加入我们再dev上的工作完成了，就可以把dev分支合并到master上。

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，就是改改指针，工作区内容不变。

合并完成后，甚至可以删除dev分支。删除dev分支就是把dev指针删掉，删掉后，就剩下一条master分支；



git checkout -b dev

创建dev分支

相当于两条命令：

git branch dev

git checkout dev

git branch

查看当前分支

合并分支

例如将dev分支合并到master分支

当前分支切换到master上

git merge dev

就可以将dev分支合并到master分支上了

合并完成就可以删除分支了

git branch -d dev

删除dev分支

切换分支：

git checkout dev

或者

git switch dev

创建并切换分支：

git checkout -b dev

git switch -c dev 创建并切换到dev分支

创建分支：

git branch dev

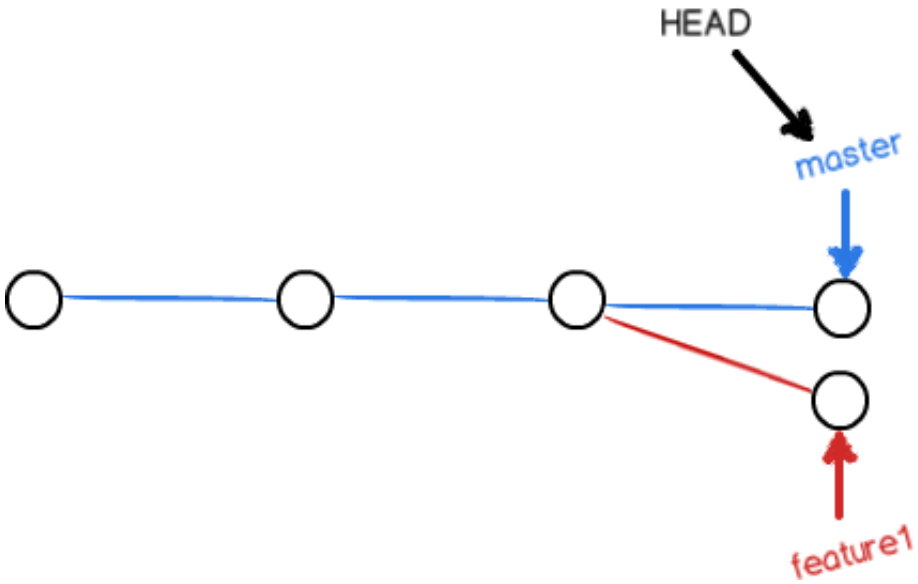
删除分支：

git branch -d

合并某分支到当前分支：

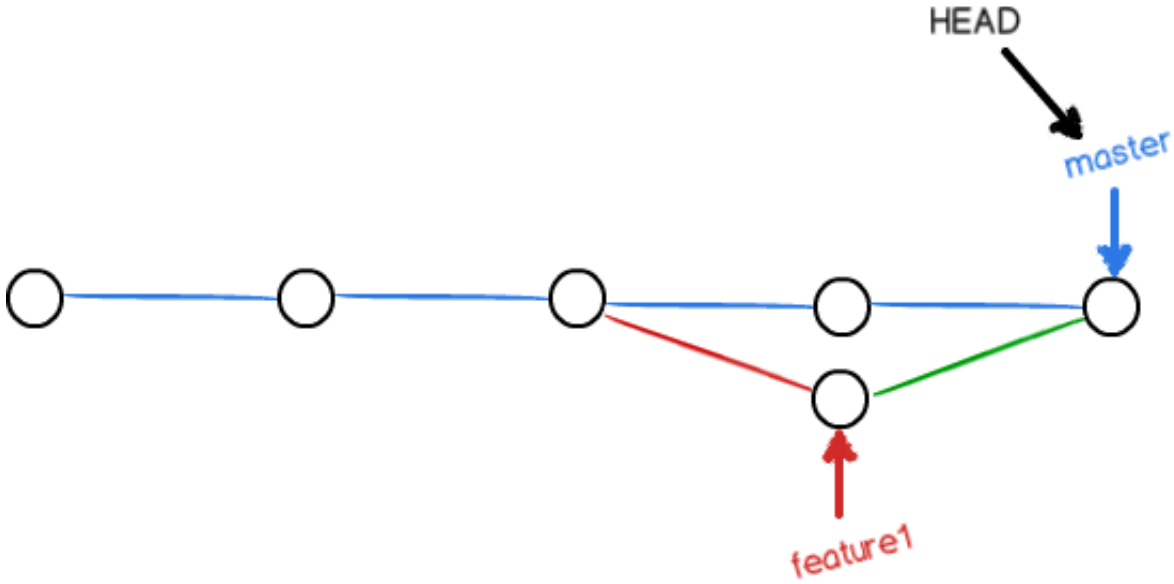
git merge s

合并分支冲突



git status查看冲突的文件

修改冲突再add和commit



可以查看

```
git log --graph --pretty=oneline --abbrev-commit
```

```
TestGit — -bash — 85×20
192:TestGit didi$ git log --graph --pretty=oneline --abbrev-commit
*   f8e76cb (HEAD -> master) conflict fixed
|\
| * 4f970dc (feature1) AND simple
* | a368bc5 & simple
|/
* 0658632 (dev) create a branch
* f4b59e2 (origin/master) remove test.txt
* 144d198 add test.txt
* 336e3ca add git tracks
* 45d9e81 append GPL
* e1e507f add distributed
* 897a879 write a readme file
192:TestGit didi$
```

通常，合并分支时，如果可能，Git会用Fast Farwarf模式，但这种模式下，删除分支后，会丢失分支信息。

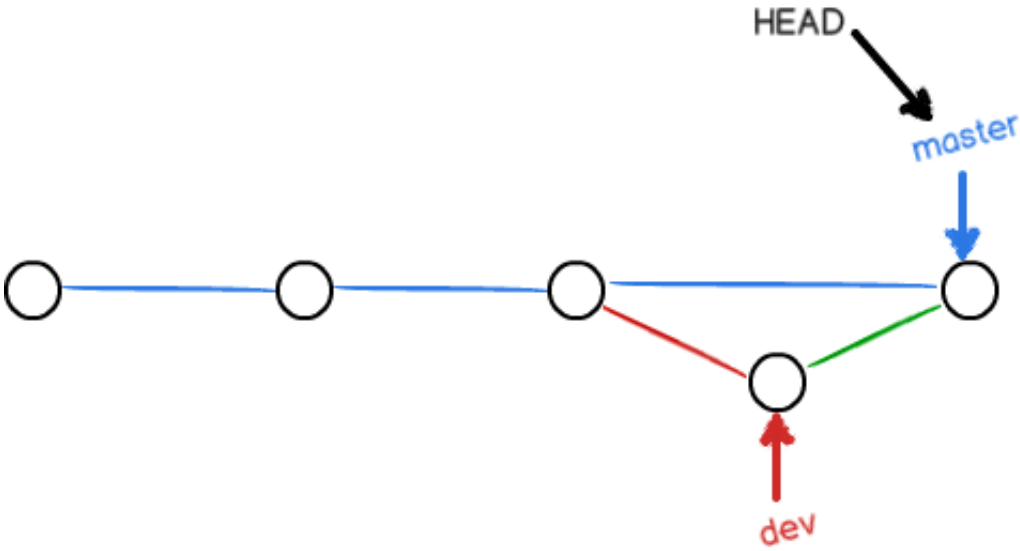
如果要强制禁用Fast Farward模式，Git就会在merge时产生一个新的commit，这样，从分支历史上就可以看出分支信息。

```
git merge --no-ff -m "merge with no-ff" dev
```

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

```
TestGit — -bash — 85×20
192:TestGit didi$ git log --graph --pretty=oneline --bbrev-commit
fatal: 未能识别的参数： --bbrev-commit
192:TestGit didi$ git log --graph --pretty=oneline --abbrev-commit
*   bd27d6f (HEAD -> master) merge with no-ff
|\
| * fddc318 (dev1) add merge
|/
*   f8e76cb conflict fixed
|\
| * 4f970dc (feature1) AND simple
* | a368bc5 & simple
|/
* 0658632 (dev) create a branch
* f4b59e2 (origin/master) remove test.txt
* 144d198 add test.txt
* 336e3ca add git tracks
* 45d9e81 append GPL
* e1e507f add distributed
* 897a879 write a readme file
192:TestGit didi$
```

不实用Fast Forward模式，merge后就是这样的：



分支策略

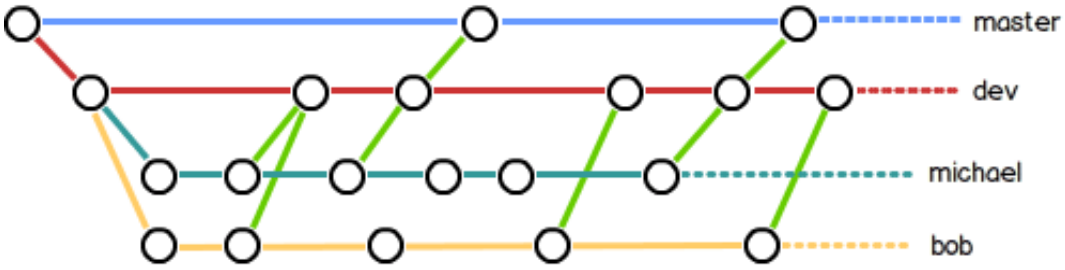
在实际开发中，我们应该按照几个基本原则进行分支管理：

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

干活都在dev分支上，也就是说，dev分支时不稳定的，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；

团队中的每个成员都在dev分支上干活，每个人都有自己的分支，是不是地在dev分支上合并就可以了。

如下图所示：



Bug分支

每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

git stash

把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug

首先去定需要在哪个分支上修复bug，嘉定需要在master分支上修复，就从master创建临时分支

修复完成后合并分支，并删除临时分支。

git stash list

查看存储的工作现场

Git将stash内容存在某个地方，但是需要恢复一下，有两个办法：

1. **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；
2. **git stash pop**，恢复的同时把stash内用也删了

将master上修复bug的提交复制到当前分支上

在master上修复了bug，但是在dev分支上也需要同步修复这个bug，我们只需要复制修复bug的提交到dev分支。

cherry-pick

```
git cherry-pick 4c805e2
```

Git自动给dev分支做了一次提交。

用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

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git stash pop，回到工作现场；

在master分支上修复bug，想要合并到当前dev分支，可以用git cherry-pick 命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有合并过的分支，可以通过git branch -D 强力删除

多人协作

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

```
git push origin master
```

如果要推送其他分支，比如dev，就改成：

```
git push origin dev
```

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢?

master分支是主分支，因此要时刻与远程同步；

dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

bug分支只用于在本地修复bug，就没必要推送到远程了，除非老板要看看你每周到底修复了几个bug；

feature分支是否推送到远程，取决于你是否和小伙伴合作在上面开发

git pull把最新的提交从origin/dev抓去下来，然后在本地habit，解决冲突，再推送

如果没有本地dev分支与远程origin/dev分支的链接，将失败

设置dev和origin/dev的链接：

```
git branch --set-upstream-to=origin/dev dev
```

再pull，如果 git pull有冲突，需要手动解决，解决后，提交，再push。

小结：

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分支的链接关系没有创建，用命令

```
git branch --set-upstream-to <branch-name> origin/<branch-name>
```

查看远程仓库信息，使用`git remote -v`;

git rebase

rebase操作可以把本地未push的分叉历史整理成直线；

rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比；

标签管理

```
git tag v1.0
```

可以用git tag查看所有标签

默认标签是打在最新提交的commit上的。

如果要给以前的版本打标签怎么办?

1. git log找出以前的提交版本；
2. `git tag v0.9 f52c633`

```
git show <tagname>
```

可以查看标签信息；

好可以创建带有说明的标签，-a指定标签名，-m指定说明文字

```
git tag -a v0.1 -m "version 0.1 released" 1094adb
```

删除标签

```
git tag -d v0.1
```

推送某个标签到远程

```
git push origin v1.0
```

或者，一次性推送全部尚未推送到远程的本地标签

```
git push origin --tags
```

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

```
git tag -d v0.9
```

然后，从远程删除。删除命令也是push，但是格式如下：

```
git push origin :refs/tags/v0.9
```

.gitignore文件

把需要git忽略写到.gitignore文件里面去

```
# Windows:
Thumbs.db
ehthumbs.db
Desktop.ini

# Python:
*.py[cod]
*.so
*.egg
*.egg-info
```

```
dist
build

# My configurations:
db.ini
deploy_key_rsa
```

最后一步就是把.gitignore也提交到Git，就完成了！当然检验.gitignore的标准是`git status`命令是不是说`working directory clean`

给Git命令陪着别名

搭建Git服务器

1. 安装git

   ```
   sudo apt-get install git
   ```

2. 创建一个git用户，用来运行git服务

   ```
   sudo adduser git
   ```

3. 创建证书登陆

   手机所有需要登陆的用户的弓腰，就是他们自己的id_rsa.pub文件，把所有公钥导入/home/git/.ssh/authorized_keys文件里，一行一个。

4. 初始化Git仓库

   先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

   ```
   sudo git init --bare sample.git
   ```

   Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git:

   ```
   sudo chown -R git:git sample.gir
   ```

5. 禁用shell登录

   出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

   ```
   git:x:1001:1001:,,,:/home/git:/bin/bash
   ```

改为

```
git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell
```

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

6. 克隆远程仓库：

   现在可以通过`git clone`命令克隆远程仓库了，在各自的电脑上运行

   ```
   git clone git@server:/srv/sample.git
   ```

管理公钥

如果团队很小，把每个人的公钥收集起来放到服务器的/home/git/.ssh/authorized_keys文件里就是可行的。如果团队有几百号人，就没法这么玩了，这是就可以通过Gitosis来管理公钥。

权限管理

Gitolite.