1、git安装

2、创建版本库

a)、 $ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

b）、通过git init命令把这个目录变成Git可以管理的仓库：

$ git init

Initialized empty Git repository in /Users/michael/learngit/.git/

ls –ah会看到.git目录

3、把文件添加到版本库

第一步，用命令git add告诉Git，把文件添加到仓库：

$ git add readme.txt

第二步，用命令git commit告诉Git，把文件提交到仓库：

$ git commit -m "wrote a readme file"

[master (root-commit) cb926e7] wrote a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

相关命令：

git status：查看当前仓库状态

git diff：查看两个版本文件的区别

git log [--pretty=oneline]：查看历史提交记录

4、版本回退

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

$ git reset --hard HEAD^

HEAD is now at ea34578 add distributed

最新的那个版本append GPL已经看不到了！好比你从21世纪坐时光穿梭机来到了19世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着 往上找啊找啊，找到那个append GPL的commit id是3628164...，于是就可以指定回到未来的某个版本：

$ git reset --hard 3628164

HEAD is now at 3628164 append GPL

Git提供了一个命令git reflog用来记录你的每一次命令：

$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file

5、撤销修改

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

6、删除文件

git rm test.txt

git commit –m “remove test.txt”

7、添加远程库

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。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：



点“Add Key”，你就应该看到已经添加的Key：



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！

8、从远程库克隆

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 clone git@github.com:michaelliao/gitskills.git

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。

9、创建合并分支

在[版本回退](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000" \t "_blank)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：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev fec145a] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout master

Switched to branch 'master'

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：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

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。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was fec145a).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

10、解决分支合并冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

修改readme.txt最后一行，改为：

Creating a **new** branch is quick **AND** simple.

在feature1分支上提交：

$ git add readme.txt

$ git **commit** -m "AND simple"

[feature1 75a857c] **AND** simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 1 **commit**.

Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。

在master分支上把readme.txt文件的最后一行改为：

Creating a **new** branch is quick & simple.

提交：

$ git add readme.txt

$ git commit -m "& simple"

[master 400b400] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

现在，master分支和feature1分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict **in** readme.txt

Automatic merge failed; fix conflicts **and** **then** commit the result.

果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

$ git status

*# On branch master*

*# Your branch is ahead of 'origin/master' by 2 commits.*

*#*

*# Unmerged paths:*

*# (use "git add/rm <file>..." as appropriate to mark resolution)*

*#*

*# both modified: readme.txt*

*#*

no changes added to commit (use "git add" **and**/**or** "git commit -a")

我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

Creating a **new** branch is quick **and** simple.

再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master 59bc1cb] conflict fixed

现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：

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

\* 59bc1cb conflict fixed

|\

| \* 75a857c **AND** simple

\* | 400b400 & simple

|/

\* fec145a branch test

...

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 75a857c).

工作完成

11、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Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

修改readme.txt文件，并提交一个新的commit：

$ git add readme.txt

$ git **commit** -m "add merge"

[dev 6224937] **add** merge

1 file changed, 1 insertion(+)

现在，我们切换回master：

$ git checkout master

Switched to branch 'master'

准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

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

Merge made by the 'recursive' strategy.

readme.txt | 1 +

1 file changed, 1 insertion(+)

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。

合并后，我们用git log看看分支历史：

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

\* 7825a50 merge **with** no-ff

|\

| \* 6224937 add merge

|/

\* 59bc1cb conflict fixed

...

可以看到，不使用Fast forward模式，merge后就像这样：



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。

12、bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，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。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

$ git status

*# On branch dev*

*# Changes to be committed:*

*# (use "git reset HEAD <file>..." to unstage)*

*#*

*# new file: hello.py*

*#*

*# Changes not staged for commit:*

*# (use "git add <file>..." to update what will be committed)*

*# (use "git checkout -- <file>..." to discard changes in working directory)*

*#*

*# modified: readme.txt*

*#*

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

幸好，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：

$ git stash

Saved working directory **and** index state WIP on dev: 6224937 add merge

HEAD is now at 6224937 add merge

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。

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 6 commits.

$ git checkout -b issue-101

Switched to a new branch 'issue-101'

现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

$ git add readme.txt

$ git commit -m "fix bug 101"

[issue-101 cc17032] fix bug 101

1 file changed, 1 insertion(+), 1 deletion(-)

修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 2 commits.

$ git merge --no-ff -m "merged bug fix 101" issue-101

Merge made by the 'recursive' strategy.

readme.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

$ git branch -d issue-101

Deleted branch issue-101 (was cc17032).

太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到dev分支干活了！

$ git checkout dev

Switched to branch 'dev'

$ git status

*# On branch dev*

nothing to commit (working directory clean)

工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

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：

$ git stash pop

*# On branch dev*

*# Changes to be committed:*

*# (use "git reset HEAD <file>..." to unstage)*

*#*

*# new file: hello.py*

*#*

*# Changes not staged for commit:*

*# (use "git add <file>..." to update what will be committed)*

*# (use "git checkout -- <file>..." to discard changes in working directory)*

*#*

*# modified: readme.txt*

*#*

Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)

再用git stash list查看，就看不到任何stash内容了：

$ git stash list

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}

### 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。

13、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。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

$ git checkout -b feature-vulcan

Switched to a new branch 'feature-vulcan'

5分钟后，开发完毕：

$ git add vulcan.py

$ git status

*# On branch feature-vulcan*

*# Changes to be committed:*

*# (use "git reset HEAD <file>..." to unstage)*

*#*

*# new file: vulcan.py*

*#*

$ git commit -m "add feature vulcan"

[feature-vulcan 756d4af] add feature vulcan

1 file changed, 2 insertions(+)

create mode 100644 vulcan.py

切回dev，准备合并：

$ git checkout dev

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，

就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

$ git branch -d feature-vulcan

error: The branch 'feature-vulcan' is **not** fully merged.

If you are sure you want to delete it, run 'git branch -D feature-vulcan'.

销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 756d4af).

终于删除成功！

### 小结

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，可以通过git branch -D <name>强行删除。

14、多人协作

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

### 推送分支

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分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

### 抓取分支

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.git

Cloning into 'learngit'...

remote: Counting objects: 46, done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 46 (delta 16), reused 45 (delta 15)

Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.

Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git add hello.py

$ git **commit** -m "add coding: utf-8"

[dev bd6ae48] **add** coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev

**To** git@github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' **in** 'git push --help' **for** details.

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

$ git pull

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From github.com:michaelliao/learngit

fc38031..291bea8 dev -> origin/dev

There is no tracking information **for** the current branch.

Please specify which branch you want to merge **with**.

See git-pull(1) **for** details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream dev origin/<branch>

git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

$ git branch *--set-upstream dev origin/dev*

Branch dev **set** up **to** track remote branch dev **from** origin.

再pull：

$ git pull

Auto-merging hello.py

CONFLICT (content): Merge conflict **in** hello.py

Automatic merge failed; fix conflicts **and** **then** commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840202368c74be33fbd884e71b570f2cc3c0d1dcf000" \t "_blank)完全一样。解决后，提交，再push：

$ git **commit** -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

因此，多人协作的工作模式通常是这样：

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分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，一旦熟悉了，就非常简单。

### 小结

* 查看远程库信息，使用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致；
* 建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。

15、创建标签（tag）

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

$ git branch

\* dev

master

$ git checkout master

Switched to branch 'master'

然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.0

可以用命令git tag查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

$ git log --pretty=oneline --abbrev-commit

6a5819e merged bug fix 101

cc17032 fix bug 101

7825a50 merge with no-ff

6224937 add merge

59bc1cb conflict fixed

400b400 & simple

75a857c AND simple

fec145a branch test

d17efd8 remove test.txt

...

比方说要对add merge这次提交打标签，它对应的commit id是6224937，敲入命令：

$ git tag v0.9 6224937

再用命令git tag查看标签：

$ git tag

v0.9

v1.0

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

$ git **show** v0.9

**commit** 622493706ab447b6bb37e4e2a2f276a20fed2ab4

Author: Michael Liao <askxuefeng@gmail.com>

**Date**: Thu Aug 22 11:22:08 2013 +0800

**add** merge

...

可以看到，v0.9确实打在add merge这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v0.1 -m "version 0.1 released" 3628164

用命令git show <tagname>可以看到说明文字：

$ git show v0.1

tag v0.1

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

还可以通过-s用私钥签名一个标签：

$ git tag -s v0.2 -m "signed version 0.2 released" fec145a

签名采用PGP签名，因此，必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错：

gpg: signing failed: secret key **not** available

error: gpg failed to sign the data

error: unable to sign the tag

如果报错，请参考GnuPG帮助文档配置Key。

用命令git show <tagname>可以看到PGP签名信息：

$ git **show** v0.2

tag v0.2

Tagger: Michael Liao <askxuefeng@gmail.com>

**Date**: Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released

-----**BEGIN** PGP SIGNATURE-----

Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...

-----**END** PGP SIGNATURE-----

**commit** fec145accd63cdc9ed95a2f557ea0658a2a6537f

Author: Michael Liao <askxuefeng@gmail.com>

**Date**: Thu Aug 22 10:37:30 2013 +0800

branch test

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。验证签名的方法比较复杂，这里就不介绍了。

### 小结

* 命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；
* git tag -a <tagname> -m "blablabla..."可以指定标签信息；
* git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；
* 命令git tag可以查看所有标签。

16、操作标签

如果标签打错了，也可以删除：

$ git tag -d v0.1

Deleted tag 'v0.1' (was e078af9)

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

Total 0 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new tag] v1.0 -> 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

Counting objects: 1, done.

Writing objects: 100% (1/1), 554 bytes, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [**new** tag] v0.2 -> v0.2

\* [**new** tag] v0.9 -> v0.9

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

Deleted tag 'v0.9' (was 6224937)

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

To git@github.com:michaelliao/learngit.git

- [deleted] v0.9

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

### 小结

* 命令git push origin <tagname>可以推送一个本地标签；
* 命令git push origin --tags可以推送全部未推送过的本地标签；
* 命令git tag -d <tagname>可以删除一个本地标签；
* 命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

17、自定义Git-忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files ...，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，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。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：[https://github.com/github/gitignore](https://github.com/github/gitignore" \t "_blank)

忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

举个例子：

假设你在Windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有Desktop.ini文件，因此你需要忽略Windows自动生成的垃圾文件：

*# Windows:*

Thumbs.db

ehthumbs.db

Desktop.ini

然后，继续忽略Python编译产生的.pyc、.pyo、dist等文件或目录：

*# Python:*

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

加上你自己定义的文件，最终得到一个完整的.gitignore文件，内容如下：

*# 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。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个.gitignore文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。

有些时候，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命令检查：

$ 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。

小结

* 忽略某些文件时，需要编写.gitignore；
* .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

18、