

**Git学习笔记**

[文档副标题]



[日期]

[公司名称]

[公司地址]

目录

[标题1](#_Toc110788539)

[目录](#_Toc110788539)

[键入章标题(第 1 级)1](#_Toc110788539)

[键入章标题(第 2 级)2](#_Toc110788539)

[键入章标题(第 3 级)3](#_Toc110788539)

[键入章标题(第 1 级)4](#_Toc110788539)

[键入章标题(第 2 级)5](#_Toc110788539)

[键入章标题(第 3 级)6](#_Toc110788539)

[2](#_Toc110788539)

[标题2 2](#_Toc110788540)

[标题3 2](#_Toc110788541)

[标题4 2](#_Toc110788542)

[标题五 2](#_Toc110788543)

[标题6 2](#_Toc110788544)

# 标题1

## 标题2

## 标题3

#### 标题4

#### 标题五

###### 标题6

# Git安装及基本操作

### 1.在Windows上安装Git

首先，从Git官网直接下载安装程序。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

在命令行输入：

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

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。

### 2.Git版本控制系统

#### 2.1工作区 (Working Directory)

就是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：

图形用户界面, 文本, 应用程序

描述已自动生成

图2.1.1

#### 2.2暂存区 (Stage/Index)和分支 (Branch)

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（Repository）。

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。

在Git中，用HEAD表示当前版本，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。

图示, 示意图

描述已自动生成

图2.2.1

所以，git add命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行git commit就可以一次性把暂存区的所有修改提交到分支。提交后，暂存区就没有任何内容了。

### 3.Git仓库操作

#### 3.1初始化一个Git仓库

使用git init命令。

$ git init

Initialized empty Git repository in /D/Git/learnGit/.git/

#### 3.2添加或修改文件

添加文件到Git仓库，分两步：

1.使用命令git add <file>，注意，可反复多次使用，添加多个文件；

2.使用命令git commit -m “<message>”，完成。

$ git add readme.txt

$ git commit -m "wrote a readme file"

[master (root-commit) eaadf4e] wrote a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

git commit命令中，-m后面输入的是本次提交的说明，可以输入任意内容，这样就能从历史记录里方便地找到改动记录。

commit可以一次提交很多文件，所以可以多次add不同的文件，比如：

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

#### 3.3复制文件

git mv [file] [path]

git rm [files]

这两个操作会修改工作区内容，同时将操作记录提交到暂存区。

#### 3.4删除文件

#### 3.5 移动或者重命名

#### 3.6 压缩和解压缩

# 二．Git版本控制

### 1.查看版本状态

* 要随时掌握工作区的状态，使用git status命令。

Changes not staged for commit:#只在工作区进行了修改

Untracked files:#只在工作区未被添加

Changes to be committed:#已经由工作区添加至暂存区但是未提交

nothing to commit, working tree clean #已经完成提交。（没有需要提交的修改，而且工作目录是干净的。）

* 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。（在工作区文件和暂存区文件的差别）

$ git diff readme.txt

diff --git a/readme.txt b/readme.txt

index 46d49bf..9247db6 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,2 +1,2 @@

-Git **is** a version control system.

+Git **is** a distributed version control system.

Git **is** free software

用git diff HEAD -- <file name>命令可以查看工作区和版本库里面最新版本的区别：

$ git diff HEAD -- readme.txt

diff --git a/readme.txt b/readme.txt

index 76d770f..a9c5755 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,4 +1,4 @@

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.

+Git tracks changes of files.

### 2.版本回退

在Git中，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把之前的工作成果全部丢失。

现在，我们回顾一下readme.txt文件一共有几个版本被提交到Git仓库里了：

版本1：wrote a readme file

Git **is** a version control system.

Git **is** free software.

版本2：add distributed

Git **is** a distributed version control system.

Git **is** free software.

版本3：append GPL

Git **is** a distributed version control system.

Git **is** free software distributed under the GPL.

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 re**set** --hard HEAD^

HEAD **is** now **at** e475afc **add** distributed

在Git中，用HEAD表示当前版本，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。

在Git中，我们用git log命令查看历史记录，git log命令显示从最近到最远的提交日志：

$ git log

commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (HEAD -> master)

Author: Michael Liao <askxuefeng@gmail.com>

Date: Fri May 18 21:06:15 2018 +0800

append GPL

commit e475afc93c209a690c39c13a46716e8fa000c366

Author: Michael Liao <askxuefeng@gmail.com>

Date: Fri May 18 21:03:36 2018 +0800

add distributed

commit eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Fri May 18 20:59:18 2018 +0800

wrote a readme file

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：

$ git log --pretty=oneline

1094adb7b9b3807259d8cb349e7df1d4d6477073 (HEAD -> master) append GPL

e475afc93c209a690c39c13a46716e8fa000c366 add distributed

eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0 wrote a readme file

其中显示的一大串类似1094adb...的数字是commit id（版本号）。是一个SHA1计算出来的一个非常大的数字，用十六进制表示。每提交一个新版本，实际上Git就会把它们自动串成一条时间线。

Git提供了一个命令git reflog用来记录的每一次命令：

$ git reflog

e475afc HEAD@{1}: reset: moving to HEAD^

1094adb (HEAD -> master) HEAD@{2}: commit: append GPL

e475afc HEAD@{3}: commit: add distributed

eaadf4e HEAD@{4}: commit (initial): wrote a readme file

在Git中，如果不慎误退回了旧版本，还可以用版本号指定回到未来的某个版本。

$ git re**set** --hard 1094a

HEAD **is** now **at** 83b0afe append GPL

版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

Git的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从指向append GPL：

┌────┐

│HEAD│

└────┘

│

└──> ○ append GPL

│

○ add distributed

│

○ wrote a readme file

改为指向add distributed：

┌────┐

│HEAD│

└────┘

│

│ ○ append GPL

│ │

└──> ○ add distributed

│

○ wrote a readme file

然后顺便把工作区的文件更新了。所以让HEAD指向哪个版本号，就把当前版本定位在哪。

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。

### 3.管理修改

第一次修改 -> git add -> 第二次修改 -> git commit

Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

第一次修改 -> git add -> 第二次修改 -> git add -> git commit

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。

每次修改，如果不用git add到暂存区，那就不会加入到commit中。

### 4.撤销修改

1.撤销工作区修改

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。

注：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。

2.撤销暂存区修改

如果已经git add到暂存区了，但在commit之前，发现了问题想要撤销。

用命令git reset HEAD <file>可以把暂存区的修改撤销掉（unstage），重新放回工作区：

$ git re**set** HEAD readme.txt

Unstaged changes after reset:

M readme.txt

3.撤销版本库修改

如果改错了东西，还从暂存区提交到了版本库。可以通过版本回退来退到上一个版本。不过，这是有条件的，就是还没有把自己的本地版本库推送到远程。

使用git reset命令：

$ git re**set** --hard HEAD^

小结：

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。

### 5.删除文件

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

$ rm test.txt

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ git status

On branch master

Changes not staged for commit:

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

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

deleted: test.txt

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

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

$ git rm test.txt

rm 'test.txt'

$ git **commit** -m "remove test.txt"

[master d46f35e] remove test.txt

1 file changed, 1 deletion(-)

**delete** mode 100644 test.txt

现在，文件就从版本库中被删除了

注：小提示：先手动删除文件，然后使用git rm <file>和git add<file>效果是一样的。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

注意：从来没有被添加到版本库就被删除的文件，是无法恢复的！

小结：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**。

# 三．远程仓库

### 1.添加远程仓库

1.要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git；关联一个远程库时必须给远程库指定一个名字，origin是默认习惯命名；

$ git remote add origin https://github.com.NineTailsFoxes/learnGit.git

2.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；

$ git push -u origin master

Counting objects: 20, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.

Total 20 (delta 5), reused 0 (delta 0)

remote: Resolving deltas: 100% (5/5), done.

To github.com:michaelliao/learngit.git

\* [new branch] master -> master

Branch 'master' **set** up **to** track remote branch 'master' **from** 'origin'.

3.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；

$ git push origin master

4.删除远程库

如果添加的时候地址写错了，或者就是想删除远程库，可以用git remote rm <name>命令。使用前，建议先用git remote -v查看远程库信息：

$ git remote -v

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

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

然后，根据名字删除，比如删除origin：

$ git remote rm origin

此处的“删除”其实是解除了本地和远程的绑定关系，并不是物理上删除了远程库。远程库本身并没有任何改动。要真正删除远程库，需要登录到GitHub，在后台页面找到删除按钮再删除。

### 2.从远程仓库克隆

要克隆一个仓库，首先必须知道仓库的地址

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

然后使用git clone 仓库地址(repository address)命令克隆。

$ git clone https://github.com.NineTailsFoxes/gitskills.git

例如：

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

Cloning into 'gitskills'...

remote: Counting objects: 3, done.

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

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

Git支持多种协议，包括https，但ssh协议速度最快。

### 3.使用Github

如何参与一个开源项目呢？比如Dan Gisselquist的axidmacheck项目，点“Fork”就在自己的账号下克隆了一个axidmacheck仓库，

图形用户界面, 文本, 网站

描述已自动生成

然后，从自己的账号下clone：

$ git clone https://github.com/NineTailsFoxes/AXI\_DMA\_Check.git

Cloning into 'AXI\_DMA\_Check'...

remote: Enumerating objects: 165, done.

remote: Counting objects: 100% (165/165), done.

remote: Compressing objects: 100% (80/80), done.

remote: Total 165 (delta 104), reused 144 (delta 83), pack-reused 0

Receiving objects: 100% (165/165), 222.28 KiB | 406.00 KiB/s, done.

Resolving deltas: 100% (104/104), done.

一定要从自己的账号下clone仓库，这样你才能推送修改。如果从作者的仓库地址克隆，因为没有权限，你将不能推送修改。

axidmacheck的官方仓库ZipCPU/axidmacheck、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：

┌─ GitHub ────────────────────────────────────┐

│ │

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ZipCPU/axidmacheck│────>│ my/axidmacheck│ │

│ └─────────────────┘ └─────────────────┘ │

│ ▲ │

└──────────────────────────────────┼──────────┘

▼

┌─────────────────┐

│ local/axidmacheck │

└─────────────────┘

如果你想修复axidmacheck的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望axidmacheck的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

### 小结

* 在GitHub上，可以任意Fork开源仓库；
* 自己拥有Fork后的仓库的读写权限；
* 可以推送pull request给官方仓库来贡献代码。

### 4.使用Gitee

# 四．分支管理

### 1.创建与合并分支

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：

图片包含 游戏机, 物体, 钟表

描述已自动生成

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：

图片包含 物体, 钟表, 男人, 游戏机

描述已自动生成

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：

图片包含 物体, 游戏机, 钟表

描述已自动生成

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：

图示

中度可信度描述已自动生成

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。

现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d46f35e..b17d20e

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge+指定分支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 b17d20e).

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。

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。

下面实战一下，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*

\* e1e9c68 (HEAD -> master) merge with no-ff

|\

| \* f52c633 (dev) add merge

|/

\* cf810e4 conflict fixed

...

可以看到，不使用Fast forward模式，merge后就像这样：

图片包含 游戏机, 物体, 钟表

描述已自动生成

### 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；

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：

图表, 折线图

描述已自动生成

实际上，切换分支这个动作，用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支：

创建并切换到新的dev分支，可以使用：

$ git switch -c dev

直接切换到已有的master分支，可以使用：

$ git switch master

### 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>或者git switch <name>

创建+切换分支：git checkout -b <name>或者git switch -c <name>

合并某分支到当前分支：git merge <name>

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。

删除分支：git branch -d <name>

熵湮 2022/8/1 18:24:14

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。

实际上，切换分支这个动作，用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支：

### 2.解决冲突

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

图表

中度可信度描述已自动生成

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.

(**use** "git push" to publish your local commits)

You have unmerged paths.

(fix conflicts **and** run "git commit")

(**use** "git merge --abort" to abort the merge)

Unmerged paths:

(**use** "git add <file>..." 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 cf810e4] conflict fixed

现在，master分支和feature1分支变成了下图所示：

图片包含 游戏机, 物体, 钟表

描述已自动生成

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

\* cf810e4 (HEAD -> master) conflict fixed

|\

| \* 14096d0 (feature1) **AND** simple

\* | 5dc6824 & simple

|/

\* b17d20e branch test

\* d46f35e (origin/master) remove test.txt

\* b84166e add test.txt

\* 519219b git tracks changes

\* e43a48b understand how stage works

\* 1094adb append GPL

\* e475afc add distributed

\* eaadf4e wrote a readme file

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 14096d0).

工作完成。

### 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。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

用git log --graph命令可以看到分支合并图。

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

### 3.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，但是当前正在dev上进行的工作还没有提交。对于这种情况，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: f52c633 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.

(use "git push" to publish your local 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 4c805e2] 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 switch master

Switched to branch 'master'

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

(use "git push" to publish your local 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(-)

用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: f52c633 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} (5d677e2ee266f39ea296182fb2354265b91b3b2a)

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}

同样的bug，要在dev上修复，我们只需要把4c805e2 fix bug 101这个提交所做的修改“复制”到dev分支。注意：我们只想复制4c805e2 fix bug 101这个提交所做的修改，并不是把整个master分支merge过来。

为了方便操作，Git专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支：

$ git branch

\* dev

master

$ git cherry-pick 4c805e2

[master 1d4b803] fix bug 101

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

Git自动给dev分支做了一次提交，注意这次提交的commit是1d4b803，它并不同于master的4c805e2，因为这两个commit只是改动相同，但确实是两个不同的commit。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，再git stash pop，回到工作现场；

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

问题：

$ git cherry-pick 1d21199

error: your local changes would be overwritten by cherry-pick.

hint: commit your changes or stash them to proceed.

fatal: cherry-pick failed

如果dev分支上有内容修改了，需要先提交，而后合并bug分支就没问题了

### 4.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，可以通过git branch -D <name>强行删除。

销毁feature-vulcan分支：

$ 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，需要使用大写的-D参数。。

现在我们强行删除：

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 287773e).

删除成功！

### 5.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。

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-to <branch-name> origin/<branch-name>。

### 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。

# 五．标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像，但是分支可以移动，标签不能移动）。创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

引入tag之后：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

### 1.创建标签

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

…

f52c633 add merge

1094adb append GPL

…

比方说要对add merge这次提交打标签，它对应的commit id是f52c633，敲入命令：

$ git tag v0.9 f52c633

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** f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)

Author: Michael Liao <askxuefeng@gmail.com>

**Date**: Fri May 18 21:56:54 2018 +0800

**add** merge

diff --git a/readme.txt b/readme.txt

...

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" 1094adb

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: Fri May 18 22:48:43 2018 +0800

version 0.1 released

commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (tag: v0.1)

Author: Michael Liao <askxuefeng@gmail.com>

Date: Fri May 18 21:06:15 2018 +0800

append GPL

diff --git a/readme.txt b/readme.txt

...

注意：标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签

### 小结

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

### 2.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 f15b0dd)

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 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

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

To github.com:michaelliao/learngit.git

\* [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 f52c633)

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 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。

# 六．自定义Git