Git为分布式版本控制系统（最快，最简单，最流行）

笔记本:	Git		
创建时间:	2018/10/9 16:38	**更新时间:**	2018/11/6 14:19
作者:	1607772973@qq.com		
URL:	https://www.liaoxuefeng.com/wiki/001373951630592960dd18361248578c67b8067c8c017b000/00137432...		

Git为分布式版本控制系统（最快，最简单，最流行）

集中式和分布式

集中式版本控制：版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。

集中式版本控制系统缺陷：必须联网才能工作

分布式版本控制系统没有“中央服务器”，每个电脑上都有完整的版本库，不需要联网

分布式版本控制系统如何多人协作：比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

分布式版本控制系统安全性比集中式版本高很多。

分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。
将一个文件放到Git仓库只需要两步
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) eaadf4e] wrote a readme file
1 file changed, 2 insertions(+)
create mode 100644 readme.txt
```

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。
git commit命令执行成功后会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：插入了两行内容（readme.txt有两行内容）。
为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

```
$ git add file1.txt
$ git add file2.txt file3.txt
```

```
$ git commit -m "add 3 files."
```

小结：
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：
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 status命令可以让我们时刻掌握仓库当前的状态
git diff 命令可以查看文件被修改前和被修改后的信息

```
[dyson@localhost learngit]$ 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.
```

小结：

- 要随时掌握工作区的状态，使用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

git log命令查看历史记录

```
[dyson@localhost learngit]$ git log
commit a9badc87ff98f1883943f89016851bff0217ae50
Author: dyson <1607772973@qq.com>
Date:   Tue Oct 9 03:27:53 2018 -0700

    add distributed

commit 9e798b345d617d903cdeedd9a429de45f675e5e8
Author: dyson <1607772973@qq.com>
Date:   Tue Oct 9 03:14:44 2018 -0700

    wrote a readme file

commit cc560c391c935c9aab4e85214330cbb29e1511cf
Author: CentOS 7 <dyson@localhost.localdomain>
Date:   Tue Oct 9 03:07:29 2018 -0700

    wrote a readme file
```

git log --pretty=oneline命令将历史记录简化

```
[dyson@localhost learngit]$ git log --pretty=oneline
a9badc87ff98f1883943f89016851bff0217ae50 add distributed
9e798b345d617d903cdeedd9a429de45f675e5e8 wrote a readme file
cc560c391c935c9aab4e85214330cbb29e1511cf wrote a readme file
```

友情提示的是，你看到的一大串类似1094adb...的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

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。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线。
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。

```
[dyson@localhost learngit]$ git reset --hard HEAD^
HEAD is now at 9e798b3 wrote a readme file
```

被还原后的内容

```
[dyson@localhost learngit]$ cat readme.txt
Git is a version control system.
Git is free software.
```

如果想再还原回去，只要上面的命令行窗口还没有关掉，就可以顺着往上找，找到add distributed 的commit id为a9bad... ,就可以再次回到未来的某个版本

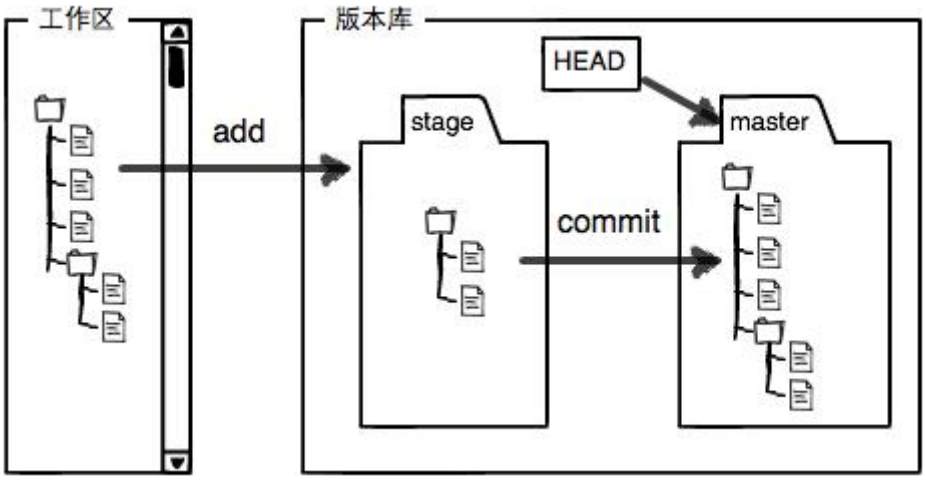
```
[dyson@localhost learngit]$ git reset --hard a9bad
HEAD is now at a9badc8 add distributed
```

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的版回退速度非常快，在Git内部有个指向当前版本的HEAd指针，当回退版本的时候，Git仅仅是把HEAD从指向wrote a readme file改为 add distributed
如果当年回退到某个版本时，关掉电脑，在当你想恢复到新版本时就可以使用git reflog命令（记录每一次命令）
小结：

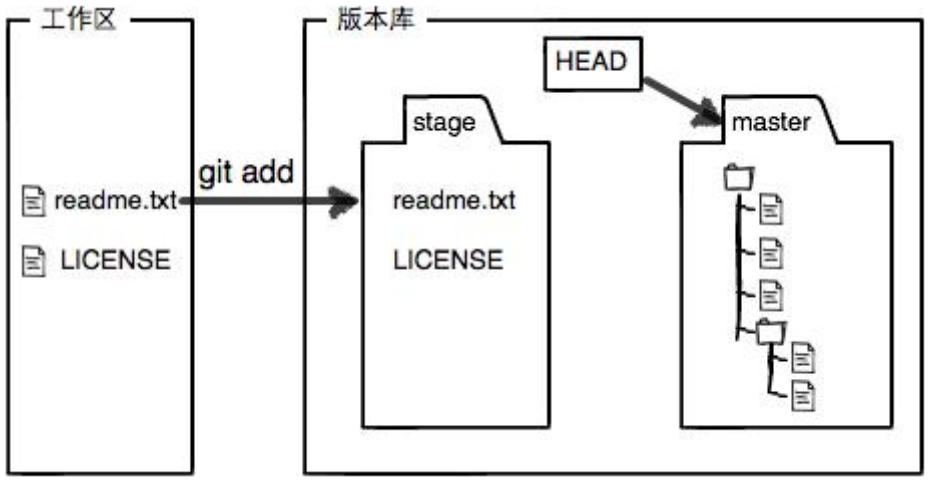
- HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit_id。
- 穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。
- 要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

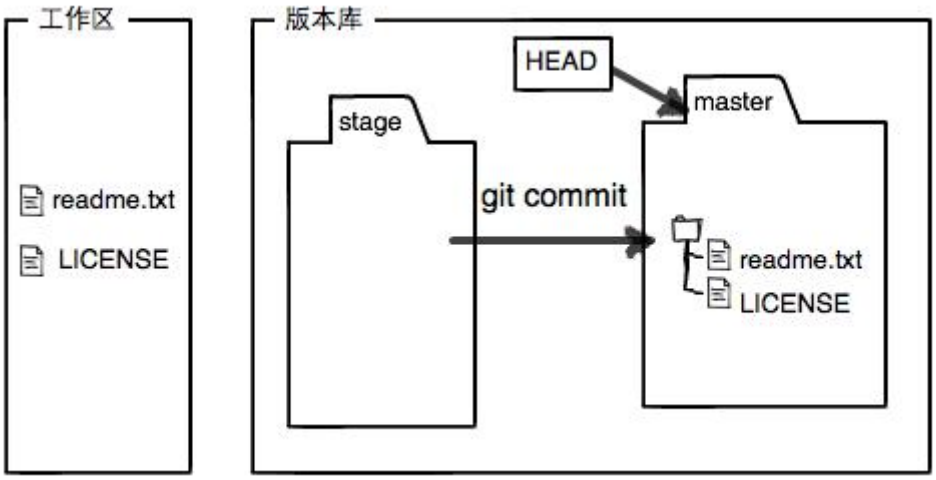
工作区和暂存区
工作区：相当于电脑中的目录，例如learngit文件夹就是一个工作区
版本库：工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

- 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



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。





管理修改

Git管理的是修改，当我们使用git add命令后，在工作区的第一次修改被放到暂存区，准备提交，但是当我们在提交前再次修改文件后提交，这时git commit只是将暂存区的修改提交，也就是将第一次修改提交，第二次修改并没有被提交

如何将两次修改都提交：第一次修改 -> git add -> 第二次修改 -> git add -> git commit
Gitg管理的·是修改而不是文件
Git是如何跟踪修改的，每次修改，如果不用git add到暂存区，那就不会加入到commit中。

撤销修改

git checkout -- <file>命令，可以将file文件在工作区的修改全部撤销，这里分为两种情况

- 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。

在第一种情况下的撤销修改直接使用git checkout --<file>
在第二种状态下的撤销修改

- 先使用git reset HEAD <file>命令将暂存区的修改撤销（unstage），重新放回工作区，这时暂存区是干净的，工作区有修改、
- 再使用git checkout --<file>将工作区所修改的内容撤销

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。一旦你把stupid boss提交推送到远程版本库，那你就真的惨了

删除文件

删除文件也是修改操作
当将一个文件提交到Git之后，使用rm <file>将文件删除后，Git知道你删除了文件，这时工作区和版本区就不一致，使用git status命令可以查看什么文件被删除了

```
[dyson@bogon learngit]$ 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:    readem.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

当将一个提交到暂存区的文件删除后，接下来你有两种选择

1. 确定要从版本库中删除该文件，然后使用git rm删除，再git commit，这样文件就从版本库中删除掉了

```
[dyson@bogon learngit]$ git rm test.txt
```

```
rm 'test.txt'
[dyson@bogon learngit]$ git commit --m "remove test.txt"
[master 5ecb52b] remove test.txt
1 file changed, 1 deletion(-)
delete mode 100644 test.txt
```

2. 当你发现删错了，由于版本库中还有备份，使用git checkout -- <file>命令k就可以将误删的文件恢复到最新版本

```
[dyson@bogon learngit]$ rm test.txt
[dyson@bogon learngit]$ ls
LICENSE  readme.txt
[dyson@bogon learngit]$ git checkout -- test.txt
[dyson@bogon learngit]$ ls
LICENSE  readme.txt  test.txt
```

3. 如果执行完git commit -m "remove test.txt"后就不能使用checkout恢复，因为commit之后版本库中的文件也没了，要使用git reset --hard HEAD^ 退回到上一个版本
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
版本库中，那么就不用担心误删，但是要注意的是，你只能恢复到最新版本，你会**丢失最近一次提交后所修改的内容**

远程仓库

添加远程库

小结：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
从远程库克隆
小结：
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
小结：Git的分支·是与众不同的，无论创建·，切换和删除分支，Git的速度很快就能完成
创建与合并分支
创建，合并和删除分支非常快，所以Git鼓励使用分支完成某个任务，合并后再删除分支，这和直接在master分支上工作效果一样，但过程更安全
小结：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

```
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git checkout -b featural
Switched to a new branch 'featural'
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (featural)
$ git add readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (featural)
$ git commit -m "AND simple"
[featural b6afaec] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (featural)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git add readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git commit -m "& simple"
[master 07ce66c] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git merge featural
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master|MERGING)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (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")
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master|MERGING)
$ cat readme.txt
Git is  version control system.
Git is free software.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick AND simple.
>>>>>>> featural
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master|MERGING)
$ vi readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master|MERGING)
$ git add readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master|MERGING)
$ git commit -m "comflict fixed"
[master 706647e] comflict fixed
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git log --graph --pretty=online --abbrev-commit
fatal: invalid --pretty format: online
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git log --graph --pretty=oneline --abbrev-commit
*   706647e (HEAD -> master) comflict fixed
```

```
|\
| * b6afaec (featural) AND simple
* | 07ce66c & simple
|/
* 87e2ca5 (origin/master) branch test
* c325896 wrote a readme file
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git branch -d featural
Deleted branch featural (was b6afaec).
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ cat readme.txt
Git is  version control system.
Git is free software.
Creating a new branch is quick and simple.
```

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 将显示数据简化

分支管理策略

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

```
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git checkout -b dev
Switched to a new branch 'dev'
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ vi readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git add readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git commit  -m "add mergee"
[dev 767e72c] add mergee
 1 file changed, 1 insertion(+)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git branch
  dev
* master
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git log --graph --pretty=oneline --abbrev-commit
```

```
*   79bc94d (HEAD -> master) merge with no-ff
|\
| * 767e72c (dev) add mergee
|/
*   706647e comflict fixed
|\
| * b6afaec AND simple
* | 07ce66c & simple
|/
* 87e2ca5 (origin/master) branch test
* c325896 wrote a readme file
```

分支策略

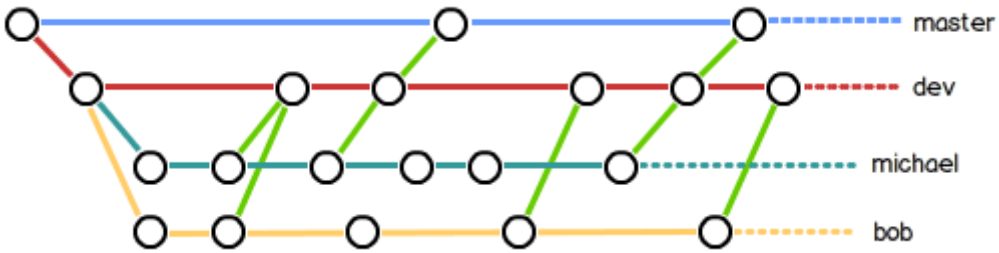
在实际开发中，我们应该按照几个基本原则进行分支管理：

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。

Bug分支

```
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git status
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git stash
Saved working directory and index state WIP on dev: 767e72c add mergee
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git status
On branch dev
nothing to commit, working tree clean
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git checkout -b issue-101
```

```
Switched to a new branch 'issue-101'
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (issue-101)
$ vi readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (issue-101)
$ git add readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (issue-101)
$ git commit -m "fix bug 101"
[issue-101 441d93d] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (issue-101)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ 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(-)
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git branch -d issue-101
Deleted branch issue-101 (was 441d93d).
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (master)
$ git checkout dev
Switched to branch 'dev'
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git status
On branch dev
nothing to commit, working tree clean
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git stash list
stash@{0}: WIP on dev: 767e72c add mergee
stash@{1}: WIP on dev: 767e72c add mergee
stash@{2}: WIP on master: 79bc94d merge with no-ff
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git stash pop
On branch dev
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
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (fd08046d6b86c0a0fd39ac2259a2518e49053f0e)
```

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: 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-101Switched 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 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 merge --no-ff -m "merged bug fix 101" issue-101Merge made by the 'recursive'
strategy.
readme.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

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 tree 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: 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}
```

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。

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 287773e] 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，但是这个包含机密资料的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).
```

终于删除成功！

```
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git checkout -b feature-vulcan
Switched to a new branch 'feature-vulcan'
M       readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ vi vulcan.py
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ git add vulcan.py
warning: LF will be replaced by CRLF in vulcan.py.
The file will have its original line endings in your working directory.
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ git status
On branch feature-vulcan
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   vulcan.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
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ cat vulcan.py
print('fly to vulcan')
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ git commit -m "add feature vulcan"
[feature-vulcan dd83e48] add feature vulcan
 1 file changed, 1 insertion(+)
 create mode 100644 vulcan.py
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (feature-vulcan)
$ git checkout dev
Switched to branch 'dev'
M       readme.txt
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ 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'.
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was dd83e48).
Dyson@DESKTOP-N8E2PHI MINGW64 ~/learngit (dev)
$ git status
On branch dev
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
no changes added to commit (use "git add" and/or "git commit -a")
```

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。

多人协作

推送分支：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，或者，用git remote -v显示更详细的信息

- 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

抓取分支：

- 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；
- 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>
- 从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

Rebase
小结：

- rebase操作可以把本地未push的分叉提交历史整理成直线；
- 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。

创建标签
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> 或 git tag <tagname> <commit id>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。**

操作标签
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。

当你要删除一个远程标签，首先从本地删除，然后再从远程删除

使用GitHub
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。

将Git内容推送到码云上步骤：
由于码云新建项目后悔自动创建一个readme文件，这样会造成远程仓库和本地库的文件bupipei，需要先pull下抓取最新的gitee库中文件，然后才能push本地库到远程

1. git pull gitee master
2. git push -u gitee master

git remote -v 查看远程库信息
git remote rm origin 删除已关联名为origin大远程库
git remote add github https://github.com/Dyson-x/javacode.git 关联远程库
这时远程库的名称不叫origin ，而叫github

git clone 将服务器的项目下载到本地
git三板斧：

1. git add
2. git commit
3. git push