目录

[第一节：创建版本库 3](#_Toc15457758)

[第二节：时光机穿梭 3](#_Toc15457759)

[第三节：版本回退 5](#_Toc15457760)

[第四节：工作区和暂存区 6](#_Toc15457761)

[第五节：管理修改 8](#_Toc15457762)

[第五节：撤销修改 8](#_Toc15457763)

[第六节：删除文件 9](#_Toc15457764)

[第六节：远程仓库 10](#_Toc15457765)

[第七节：从远程库克隆 10](#_Toc15457766)

[第八节：分支管理 11](#_Toc15457767)

[第九节：解决冲突 12](#_Toc15457768)

[第十节：分支管理策略 14](#_Toc15457769)

[第十一节：Bug分支 14](#_Toc15457770)

[第十二节：Feature分支 17](#_Toc15457771)

[第十三节：本地仓库上传到github仓库 17](#_Toc15457772)

[第十四节：克隆到本地的文件夹，查看git地址 17](#_Toc15457773)

（蓝色代表输入的代码，黄底蓝色代表命令行执行之后显示的代码）

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

版本控制系统：



集成式和分布式：

先说集中式版本控制系统，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最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？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。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

第一节：创建版本库

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。

1、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！

2、（1）创建版本库

$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

pwd命令用于显示当前目录。在我的Mac上，这个仓库位于/Users/michael/learngit。

（2）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/

3、把文件添加到版本库

编写一个readme.txt文件，一定要放到learngit目录下（子目录也行）内容如下：

Git is a version control system.

Git is free software.

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

总结：

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：

* 第一步，使用命令git add <file>，注意，可反复多次使用，添加多个文件；
* 第二步，使用命令git commit，完成。

第二节：时光机穿梭

1、修改readme.txt文件，改成如下内容：

Git is a distributed version control system.

Git is free software.

2、现在，运行git status命令看看结果：（命令可以让我们时刻掌握仓库当前的状态，）

$ git status

# On branch master

# 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 （readme.txt被修改过了）

#

no changes added to commit (use "git add" and/or "git commit -a") （但还没有准备提交的修改。）

3、具体修改了什么内容：用git diff这个命令看看：

$ 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. （我们在第一行添加了一个“distributed”单词。）

Git is free software.

4、提交到仓库

（1）git add

$ git add readme.txt

（2）样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：

git status

# On branch master

# Changes to be committed:

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

#

# modified: readme.txt

#

（3）git commit  
 $ git commit -m "add distributed"

[master ea34578] add distributed

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

（4）提交后，我们再用git status命令看看仓库的当前状态：

$ git status

# On branch master

nothing to commit (working directory clean)

总结：

* 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。

第三节：版本回退

1、修改readme.txt为

Git is a distributed version control system.

Git is free software distributed under the GPL.

2、尝试提交

$ git add readme.txt

$ git commit -m "append GPL"

[master 3628164] append GPL

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

3、回顾readme.txt文件一共有几个版本被提交到Git仓库里了：（历史记录）

$ git log

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

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

append GPL

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85 （一共有三个版本的readme.txt被修改）

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file  
git log命令显示从最近到最远的提交日志，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

3628164fb26d48395383f8f31179f24e0882e1e0 append GPL

ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed

cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file

4、把readme.txt回退到上一个版本

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

5、读取后退之后当前的readme.txt的内容

$ cat readme.txt

Git is a distributed version control system. （可以看到，现在的内容是我们第二次修改的内容）

Git is free software.

6、用git log再看看现在版本库的状态：

$ git log

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

（可以发现最新的那个版本append GPL已经看不到了，只剩下两个）

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

7、指定回到未来的某个版本

$ git reset --hard 3628164 （这个id的值与上面我们查到的git log后面的commit的前几个字母串）

HEAD is now at 3628164 append GPL

那么，如果出现找不到commit id怎么办呢？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 （这样，我们就可以找到append GPL的commit id了）

ea34578 HEAD@{2}: commit: add distributed

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

总结：

* 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。
* 用cat 文件名可以查看文档的内容

第四节：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。

1、实验先对readme.txt做个修改

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

2、然后，在工作区新增一个LICENSE文本文件（内容随便写）。先用git status查看一下状态：

$ git status

# On branch master

# 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 （readme.txt被修改了）

#

# Untracked files:

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

#

# LICENSE （LICENSE还从来没有被添加过，所以它的状态是Untracked。）

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

3、使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：

$ git status

# On branch master

# Changes to be committed:

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

# new file: LICENSE

# modified: readme.txt

#

4、commit

$ git commit -m "understand how stage works"

[master 27c9860] understand how stage works

2 files changed, 675 insertions(+)

create mode 100644 LICENSE

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

第五节：管理修改

Git跟踪并管理的是修改，而非文件。

如果，第一次你修改了readme.txt之后，git add 了，在这个时候，你有修改了readme.txt文件，但是没有git add ，直接git commit -m "hhhhhh"，提交之后查看状态，会发现第二次修改没有被提交。

提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：可以确定第二次修改的确没有被修改成功。

操作过程为：

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。

那怎么提交第二次修改呢？你可以继续git add再git commit，也可以别着急提交第一次修改，先git add第二次修改，再git commit，就相当于把两次修改合并后一块提交了：

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

现在，你又理解了Git是如何跟踪修改的，每次修改，如果不add到暂存区，那就不会加入到commit中。

第五节：撤销修改

一、没有add到暂存区：

1、例如现在的readme.txt 文档中有以下内容：

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.

My stupid boss still prefers SVN. （老板是傻子哈哈哈哈）

2、及时纠正

（1）先用git status查看一下当前的状态

$ git status

# On branch master

# 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)

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

# modified: readme.txt

#

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

（2）$ 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。

现在，看看readme.txt的文件内容：

$ cat readme.txt

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. （可以发现readme.txt的内容已经将最后一句话删除了）

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。

二、add到了缓存区怎么办呢？

git add readme.txt

git reset HEAD readme.txt

git status

git checkout -- readme.txt

git status

cat readme.txt

总结：

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](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)一节，不过前提是没有推送到远程库。

第六节：删除文件

在工作区建立文件：text.txt

删除文件：

rm text.txt

此时有两种情况：

（1）彻底删除，并不是误删

git rm text.txt

git commit -m "remove text.txt"

（2）误删，需要恢复

git checkout -- text.txt

总结：

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**。

第六节：远程仓库

1、添加远程库

$ git remote add origin git@github.com:michaelliao/learngit.git（是你自己的ssh）

2、把本地库的所有内容推送到远程库

$ git push -u origin master

Counting objects: 19, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (19/19), done.

Writing objects: 100% (19/19), 13.73 KiB, done.

Total 23 (delta 6), reused 0 (delta 0)

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

\* [new branch] master -> master

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

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。

从现在起，只要本地作了提交，就可以通过命令：

$ git push origin master

把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

总结：

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；

第七节：从远程库克隆

新建项目时候Initialize this repository with a README勾选

现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：

git clone git@github.com:michaelliao/gitskills.git （ssh网址）

loning into 'gitskills'...

emote: Counting objects: 3, done.

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

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

cd gitskills

ls

EADME.md

第八节：分支管理

[版本回退](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)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。

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：

实战：

1、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 （创建分支dev）

$ git checkout dev （切换到dev分支）

Switched to branch 'dev'

2、查看当前分支

$ git branch

\* dev （当前有两个分支，dev和master分支 git branch命令会列出所有分支，当前分支前面会标一个\*号。）

master

3、在dev分支上正常提交

如对readme.txt做个修改，加上一行：

$ git add readme.txt

$ git commit -m "branch test"

[dev fec145a] branch test

1 file changed, 1 insertion(+)

4、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：

5、把dev分支的工作成果合并到master分支上

$ git merge dev

Updating d17efd8..fec145a

Fast-forward (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 eadme.txt | 1 + 直接把master指向dev的当前提交，所以合并速度非常快。)

1 file changed, 1 insertion(+)

6、删除dev分支

$ git branch -d dev

Deleted branch dev (was fec145a).

7、查看branch分支

$ git branch

\* master

总结：

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>

第九节：解决冲突

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。

1、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'

2、修改readme.txt最后一行，改为：

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

3、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(-)

4、切换到master分支：

$ git checkout master

Switched to branch 'master'

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

5、在master分支上把readme.txt文件的最后一行改为：

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

6、提交

$ git add readme.txt

$ git commit -m "& simple"

[master 400b400] & simple

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

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 （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。）

CONFLICT (content): Merge conflict in readme.txt

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

7、查看readme.txt的内容：

cat readme.txt

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

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

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

$ git add readme.txt

$ git commit -m "conflict fixed"

[master 59bc1cb] conflict fixed

9、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 （用git log --graph命令可以看到分支合并图。）

\* 59bc1cb conflict fixed

|\

| \* 75a857c AND simple

\* | 400b400 & simple

|/

\* fec145a branch test

..

10、删除分支

$ git branch -d feature1

Deleted branch feature1 (was 75a857c).

第十节：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 （可以看到不使用Fast forward模式，merge后就像这样：）

|/

\* 59bc1cb conflict fixed

...

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。

第十一节：Bug分支

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。

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}

总结：

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。

第十二节：Feature分支

# 第十三节：本地仓库上传到github仓库

**在F盘下mygit2016仓库中有一个hello.txt文件，现在要讲hello.txt文件上传至github中**

一、文件保存至本地仓库

1、文件保存至本地仓库

鼠标右击选择 Git Bash Here

2、配置用户名和邮箱

git config --global user.name "xiaoduye..."

git config --global user.email "9187231@9838129"

3、添加和提交

git init

git add hello.txt

git commit -m "just atest"

提交成功

二、生成SSH密钥

1、 ssh-keygen -t rsa -C "13654922203@163.com" 按3个回车

2、在C:\Users\Administrator\.ssh目录下找到id\_rsa.pub复制里面所有内容

3、登录你的gihub账号，点击Your profile ------> 然后点击Edit profile ---->选择SSH并新建一个SSH Key ----->

其中Title中的名称可以任意填写，将C:\Users\Administrator\.ssh目录下id\_rsa.pub复制的所有内容粘贴到Key中，点击Add SSH Key，SSH密钥完成

三、远程提交

1、git remote add origin**git@github.com:chuaaqiCSDN/mygit.git**其中红色部分的URL时是gihub中的SSH

2、远程提交git push origin master

3、这时再看gihub中的mygit仓库，发现多了一个hello.txt文件

# 第十四节：克隆到本地的文件夹，查看git地址

命令：git remove -v