

Git基本操作说明

一.前言

很多人都知道，Linux在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。Linux虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linux，然后由Linux本人通过手工方式合并代码！你也许会想，为什么Linux不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linux坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linux很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linux选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linux可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linux花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

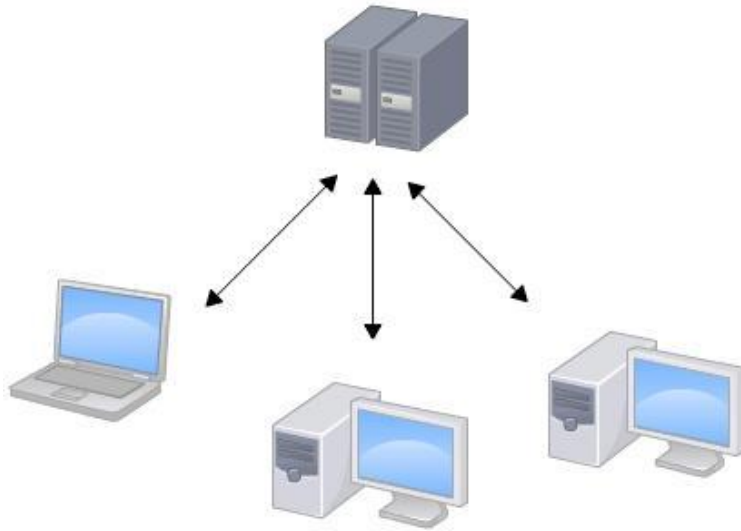
Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

二.集中式和分布式

Linux一直痛恨的CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

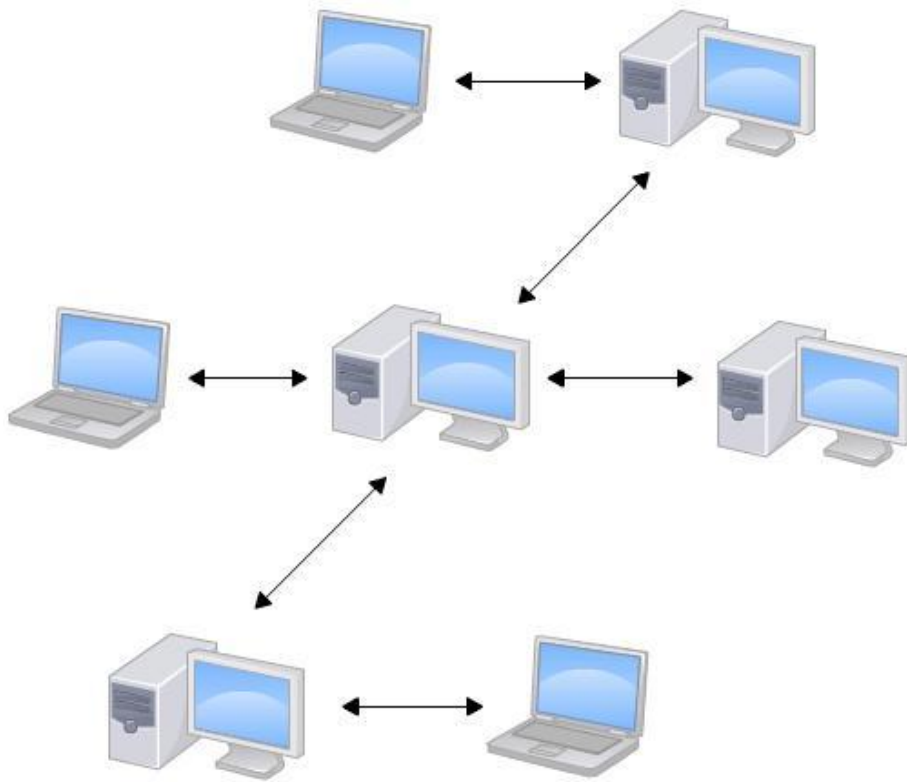
先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。



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。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，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。



当然，Git的优势不单是不必联网这么简单，后面我们还会看到Git极其强大的分支管理，把SVN等远远抛在了后面。

CVS作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人在用。由于CVS自身设计的问题，会造成提交文件不完整，版本库莫名其妙损坏的情况。同样是开源而且免费的SVN修正了CVS的一些稳定性问题，是目前用得最多的集中式版本库控制系统。除了免费的外，还有收费的集中式版本控制系统，比如IBM的ClearCase（以前是Rational公司的，被IBM收购了），特点是安装比Windows还大，运行比蜗牛还慢，能用ClearCase的一般是世界500强，他们有个共同的特点是财大气粗，或者人傻钱多。微软自己也有一个集中式版本控制系统叫VSS，集成在Visual Studio中。由于其反人类的设计，连微软自己都不好意思用了。

分布式版本控制系统除了Git以及促使Git诞生的BitKeeper外，还有类似Git的Mercurial和Bazaar等。这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是Git！

三.Git的安装

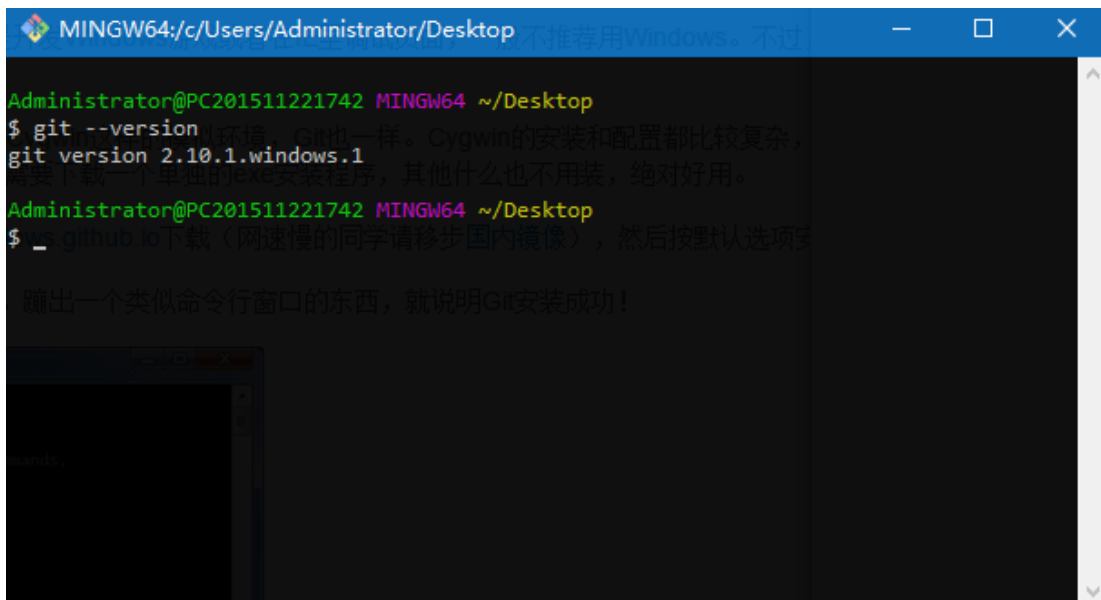
最早Git是在Linux上开发的，很长一段时间内，Git也只能在Linux和Unix系统上跑。不过，慢慢地有人把它移植到了Windows上。现在，Git可以在Linux、Unix、Mac和Windows这几大平台上正常运行了。

linux(Debian)系统

linux下的安装非常简单，Debian系统下，执行命令 `sudo apt-get install git`，输入密码即可。

Windows系统

windows系统下安装Git，首先下载安装包。下载地址：<https://git-scm.com/download/win>。然后按默认选项安装至完毕，完成后，一般可在开始菜单中找到Git->Git Bash，点击后弹出gitBash命令行，即表示安装成功。



安装成功后需要进行一步配置，命令如下：

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

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

即设置你的用户名，邮箱。用于分布式系统中的人员分辨。

四.基本操作

1.创建版本库

版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

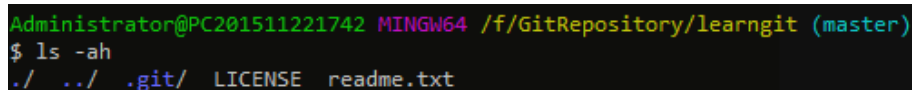
在要创建版本库的目录下运行gitBash，或者使用命令转到相应的目录下，执行

```
$ git init
```

Initialized empty Git repository in /Your/Directory/Name/.git/

即可在该目录下创建版本库，并且在该目录下会生成.git目录

(该目录为隐藏目录，使用ls -ah可以查看)



2.创建文件，添加到仓库

使用编辑器(vim或editplus等)，在刚才创建的仓库下（不是.git目录下）创建readme.txt文件作为测试。

使用如下命令添加文件

```
git add readme.txt --添加到仓库
```

```
git commit -m "添加文件" --提交到本地库
```

说明 git commit -m 中参数-m 后紧跟提交说明，并且是必须指定的。

add 可以多次使用(添加多个文件)

commit 可以一次把所有多次add的文件提交

3.查看当前库状态

要查看当前库的状态，可以使用git status命令

```
Administrator@PC201511221742 MINGW64 /f/GitRepository/learngit (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

干净的工作目录

```
Administrator@PC201511221742 MINGW64 /f/GitRepository/learngit (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/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
no changes added to commit (use "git add" and/or "git commit -a")
```

有文件修改，但是未提交的目录

如果要查看文件变动了那些部分，可以使用命令git diff，命令会显示出文件的变化信息

```
Administrator@PC201511221742 MINGW64 /f/GitRepository/learngit (master)
$ git diff
diff --git a/readme.txt b/readme.txt
index d021562..03b037a 100644
--- a/readme.txt
+++ b/readme.txt
@@ -2,4 +2,4 @@ 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 of files.
+hello
```

4.版本回退

要查看文件的修改历史，使用命令git log

```
Administrator@PC201511221742 MINGW64 /f/GitRepository/learngit (master)
$ git log
commit 25bbe8cc985d1e91e13a157307801947c7a1411d
Author: EricHuang <exsupel@163.com>
Date: Wed Oct 12 15:31:09 2016 +0800
    removes test.txt

commit 8f960ed4771d0b30d258f40541d3dbbbb747f9b2
Author: EricHuang <exsupel@163.com>
Date: Wed Oct 12 15:28:46 2016 +0800
    add test.txt

commit 29bf01f5638e60843384caf6982154d283082060
Author: EricHuang <exsupel@163.com>
Date: Wed Oct 12 15:02:14 2016 +0800
```

git log列出文件的修改历史

【参数--pretty=oneline 可以使列表更紧凑】

在Git中，用HEAD表示当前版本，也就是最新的提交25bbe8cc.....（注意这里提交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。

回退到上一个版本 使用命令

```
$ git reset --hard HEAD^
```

HEAD is now at 25bbe8cc xxxxxx

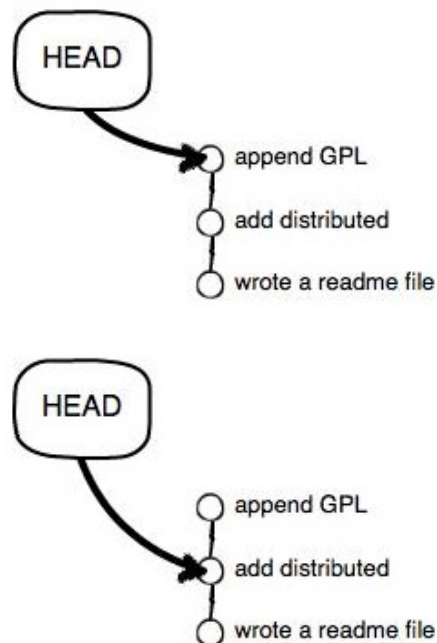
也可以指定commit 后面的版本号 转到某一个版本，如下命令

```
$ git reset --hard 8f960ed
```

HEAD is now at 8f960ed xxxxxx

(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的版本维护使用HEAD指针，版本的回退，只是改变HEAD指针的指向，所以变更当前版本速度非常快。



将HEAD指针从指向注释为append GPL的版本，转到注释为add distributed的版本

要查看历史版本记录，使用命令

```
$ git reflog
```

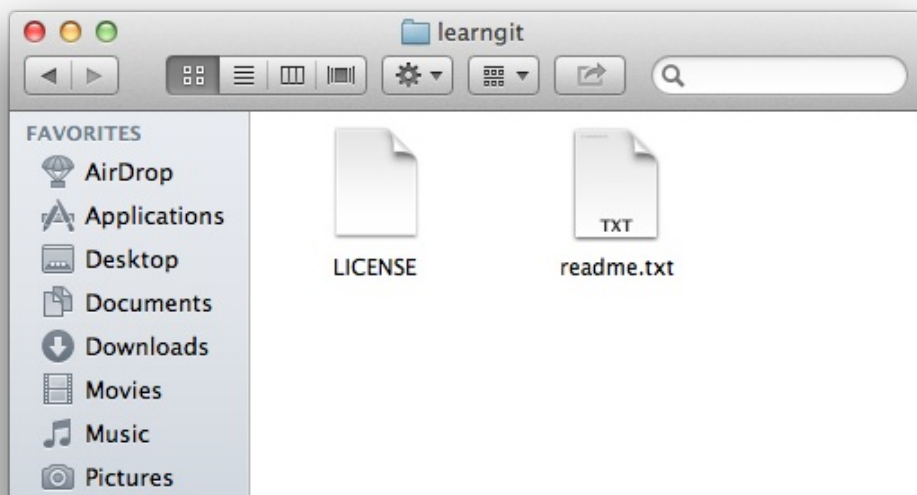
```
$ git reflog
ea34578 HEAD@{0}: reset: moving to HEAD^
3628164 HEAD@{1}: commit: append GPL
ea34578 HEAD@{2}: commit: add distributed
cb926e7 HEAD@{3}: commit (initial): wrote a readme file
```

会列出所有修改版本，使用git reset --hard xxxx即可回到指定版本。

5.基本概念

5.1工作区(Workign Directory)

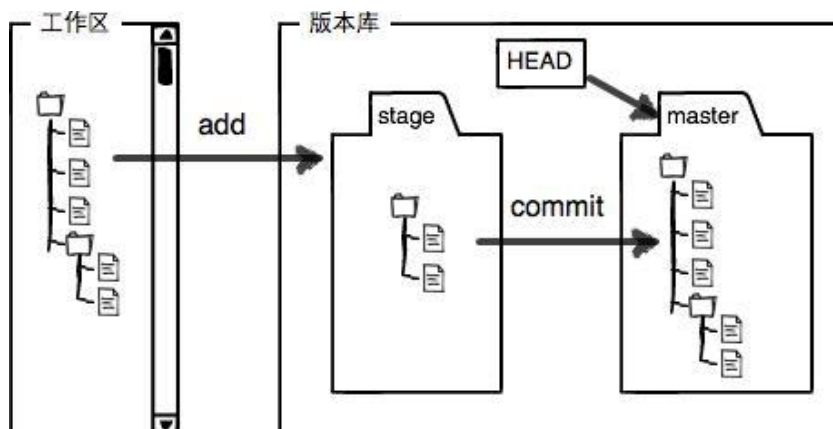
就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



5.2版本库(Repository)

工作区有一个隐藏的目录.git，这是Git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。



操作时:

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。

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。

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。

6.commit需要注意的地方

每一修改文件，都需要遵循以下步骤

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

即git add是每一步都需要的，第二次没有git add，commit只会提交第一次修改的部分。

7.撤销与删除

7.1 撤销

(未做add，未做commit之前)

git checkout -- xxx.xxx 撤销本次修改，作用是将该文件回到最近一次git commit或者git add时的状态，注意不要漏掉--，否则改命令就成了切换分支了。

(已add未commit之前)

git reset HEAD xxx.xxx

可以把暂存区的修改撤销掉 (unstage)，重新放回工作区，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。然后再执行

git checkout -- xxx.xxx 即可丢弃工作区的修改。

7.2删除文件

对于要删除的文件，直接在文件系统中删除该文件，或使用命令

\$ rm xxx.xxx

然后使用git status查看状态

```
$ 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")
```

会看到一次删除操作，要确认删除使用如下命令

git rm xxx.xxx

git commit -m "remove xxx.xxx"

删除并提交

```
$ git rm test.txt
rm 'test.txt'
$ git commit -m "remove test.txt"
[master d17efd8] remove test.txt
1 file changed, 1 deletion(-)
delete mode 100644 test.txt
```

如果是误删除文件，可以充版本库恢复，恢复命令为

\$git checkout -- xxx.xxx即可充版本库恢复后替换工作区目录

五.远程仓库

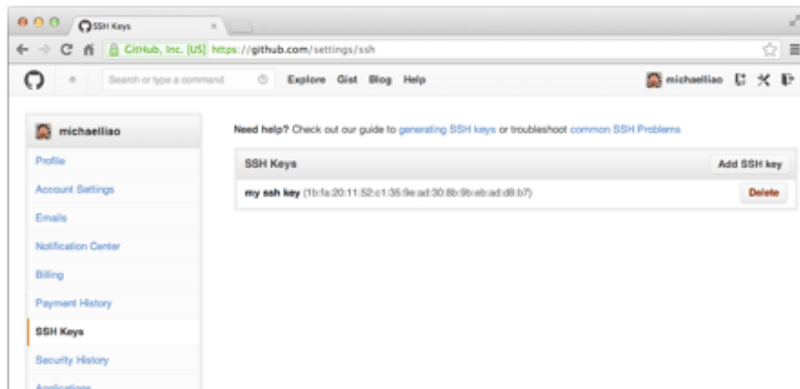
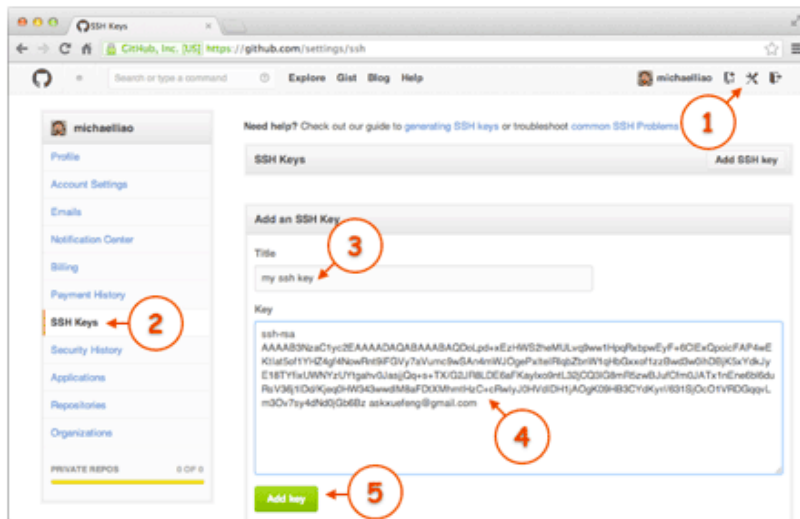
5.1 使用GitHub

首先登陆github并创建注册账号。这里使用SSH加密的方式与远程仓库通讯，接下来创建SSH Key

\$ ssh-keygen -t rsa -C "youremail@example.com"

使用默认参数即可，完成后在当前用户的C:\Users\Administrator\.ssh目录下可以看到id_rsa和id_rsa.pub两个文件，这两个就是SSH Key的密钥对，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。

登陆GitHub，填写SSH Key



为什么GitHub需要SSH Key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。

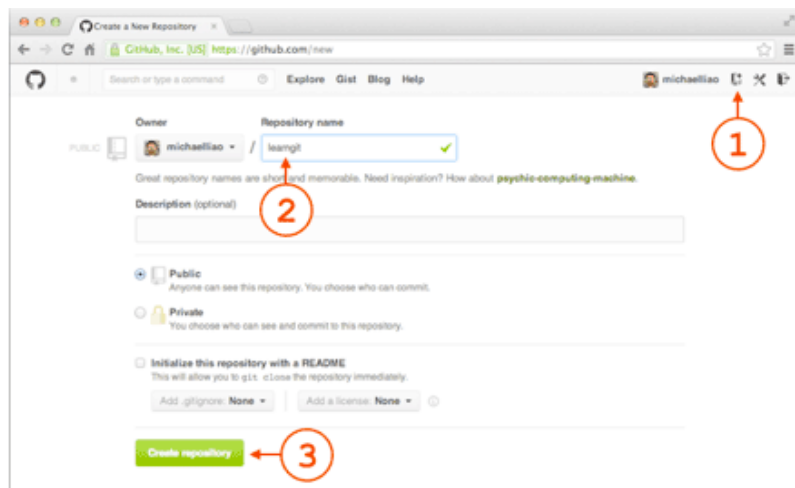
当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到（但只有你自己才能改）。所以，不要把敏感信息放进去。

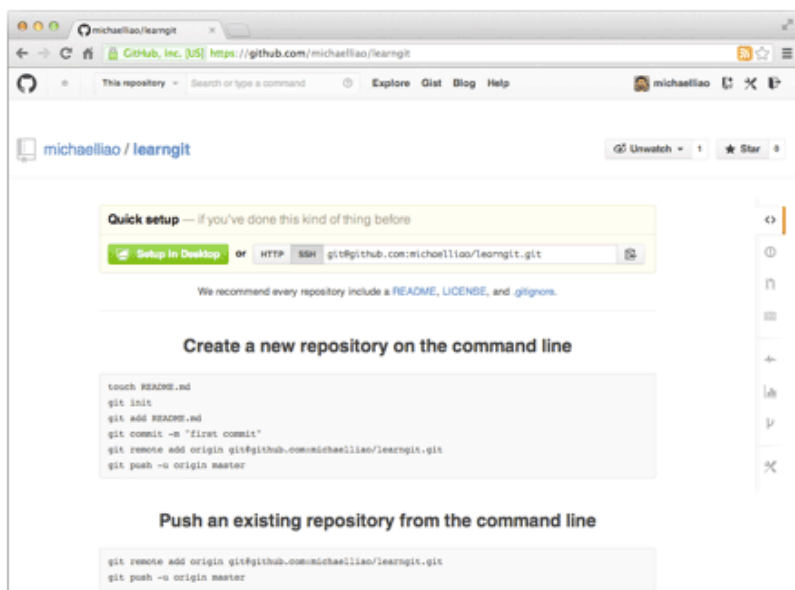
5.2添加远程库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

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

请千万注意，把上面的michaelliao替换成你自己的GitHub账户名，否则，你在本地关联的就是michaelliao的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在michaelliao的账户列表中。

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

下一步，就可以把本地库的所有内容推送到远程库上：

```
$ 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.
```

把本地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。由于远程库是空的，我们第一次推送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！

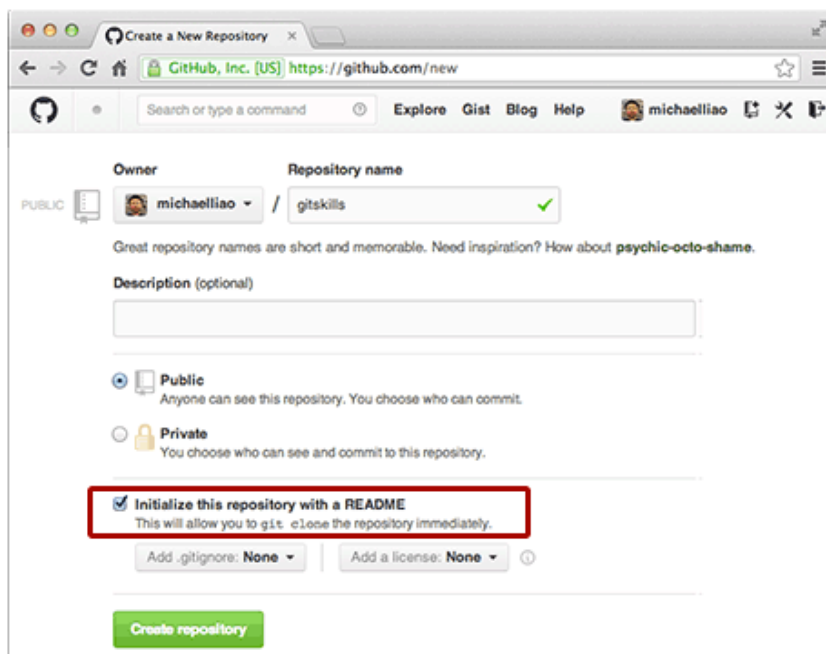
当你第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.  
RSA key fingerprint is xx.xx.xx.xx.xx.  
Are you sure you want to continue connecting (yes/no)?
```

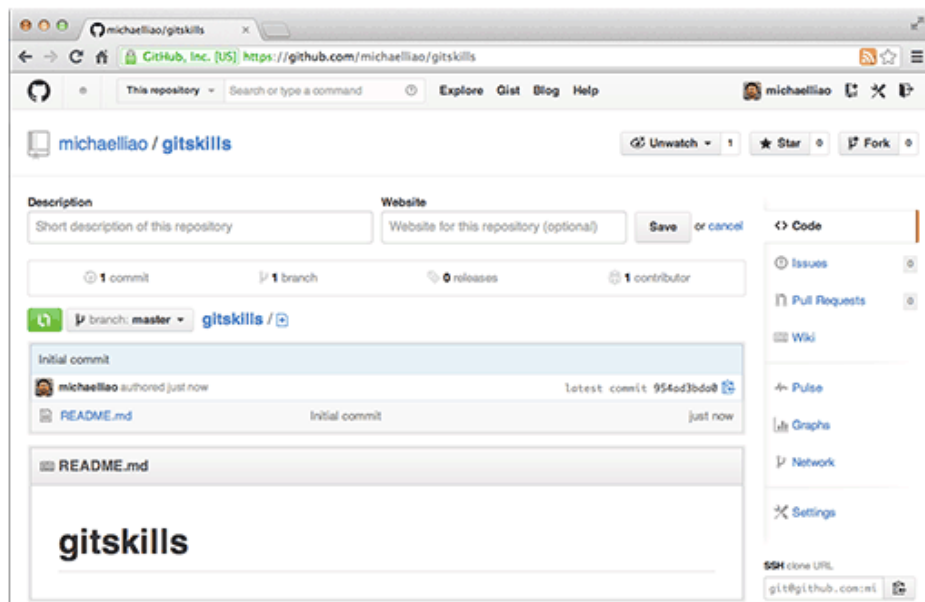
这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

5.3从远程仓库克隆

首先创建远程仓库，登陆GitHub，创建新的仓库，取名gitskills



可以勾选Initialize this repository with a README自动生成一份README.MD,完成后如下



接下来使用git clone命令

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

$ cd gitskills
$ ls
README.md
```

远程仓库就克隆到了本地。

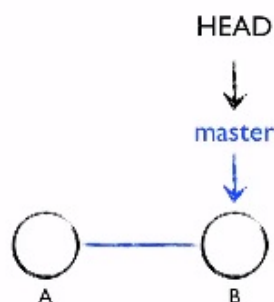
六.分支

在[版本回退](#)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：

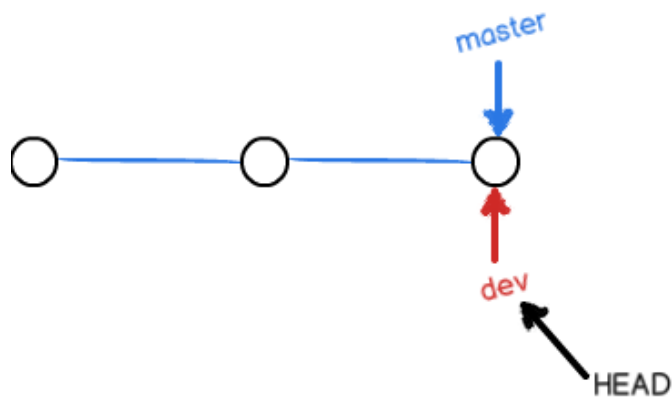
[图片下载失败]

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

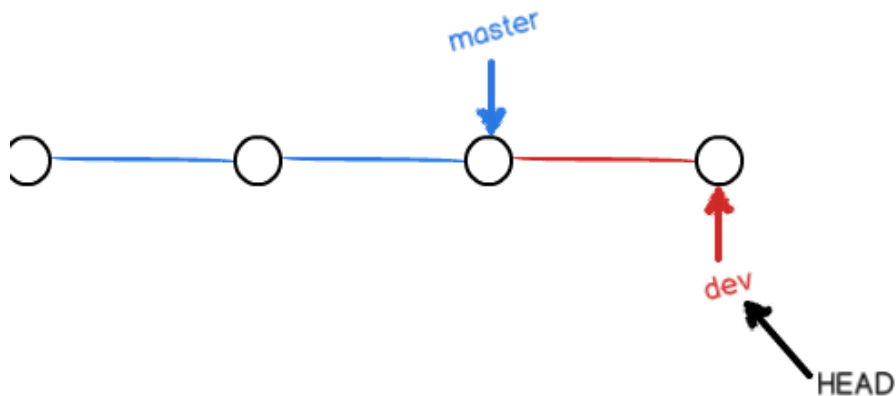


当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提

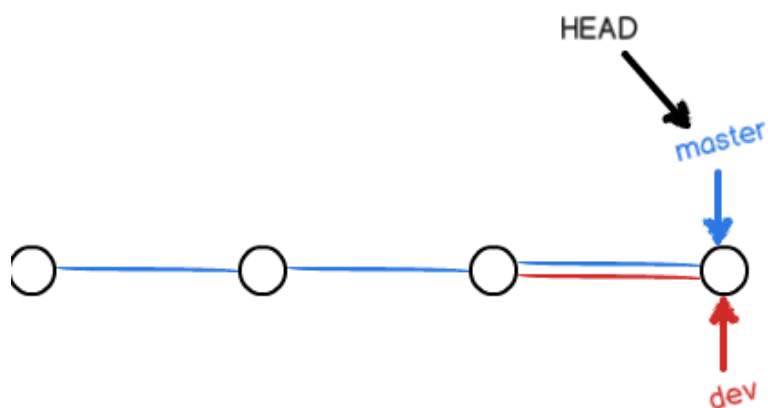
交，再把HEAD指向dev，就表示当前分支在dev上：



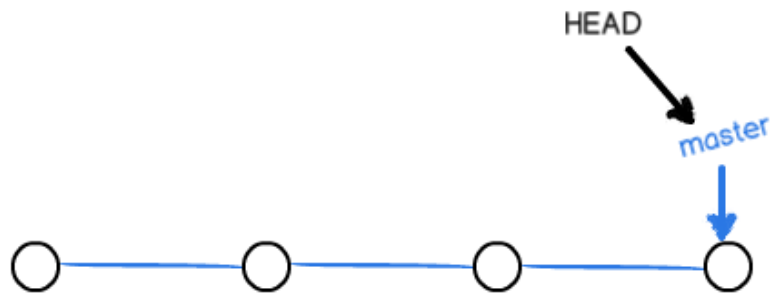
从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



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



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：



示例：

创建分支dev

```
$git checkout -b dev
```

Switched to a new branch 'dev'

(git checkout -b 相当于两条命令 git branch dev 和git checkout dev)

使用git branch查看当前所有分支

```
$git branch
```

```
$ git branch
* dev
  master
```

当前分支会带上*号

要合并dev到master分支上，依次执行如下命令

1.切换到master:: git checkout master

2.执行merge命令: git merge dev

3.如果有必要,删除dev: git branch -d dev

七.解决冲突

多人修改同一个分支的同一个文件时，就会遇到冲突，冲突必须先处理，然后才能继续提交

模拟一个冲突：

(master分支中有一个readme.txt文件，最后一行内容为Creating a **new** branch is quick)

```
$git checkout -b feature1
```

Switched to a new branch 'feature1'

修改readme最后一行的内容为

Creating a new branch is quick AND simple

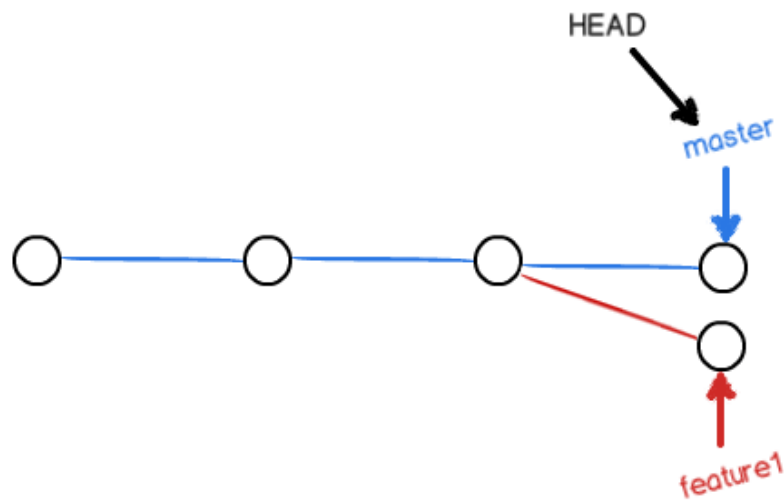
feature1分支 add并commit

切换到master分支，同样修改readme.txt最后一行为

Creating a new branch is quick & simple。

master分支 add并commit

现在两个分支的状态如下



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

尝试执行合并命令

```

$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
  
```

出现冲突，使用git status 查看状态

```

$ 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")
  
```

查看readme.txt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：

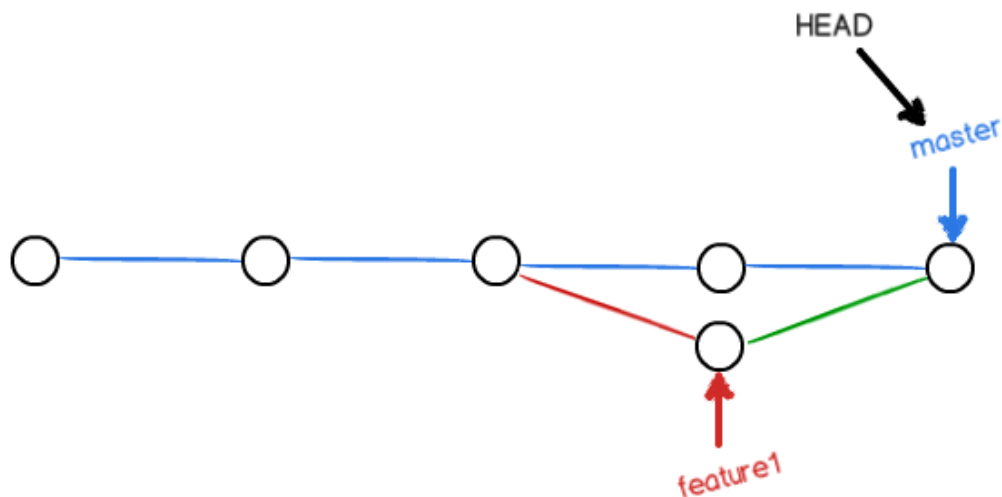
Creating a new branch is quci and simple.

再次提交

```

$ 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：



最后，删除feature1分支

```
$ git branch -d feature1
Deleted branch feature1 (was 75a857c).
```

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。用git log --graph命令可以看到分支合并图。

七.BUG分支

当工作在dev分支上时，需要临时修复一个bug，但是dev分支还没提交，又不能切换到bug分支上去工作，怎么办？

使用\$git stash命令

```
$ git stash
Saved working directory and index state WIP on dev: 6224937 add merge
HEAD is now at 6224937 add merge
```

Git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

再次使用git status查看状态，工作区已经是干净的了

从master分支上创建临时分支并解决bug，并和master分支合并后，dev怎么恢复呢？

使用git stash list查看

```
$ 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)
```

八.删除不要的分支

删除名为feature-vulcan的分支使用git branch -d feature命令

```
$ 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'.
```

但是git提示feature分支未合并，若要删除使用强制命令

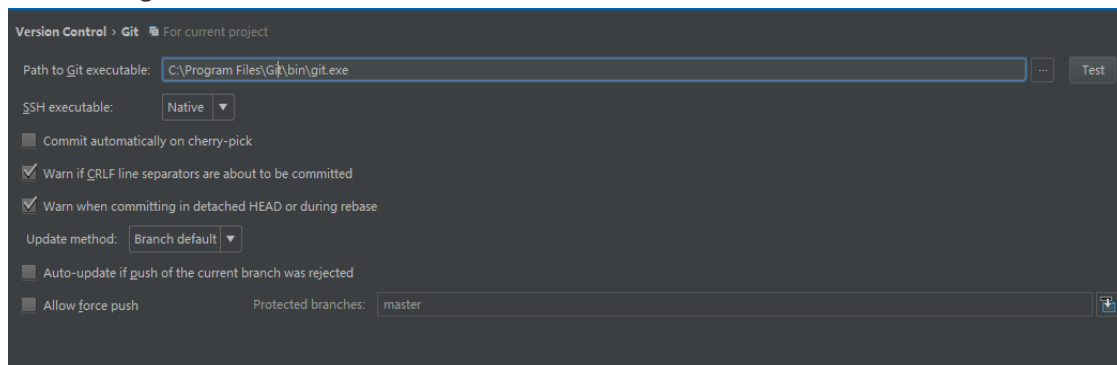
```
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 756d4af).
```

即可。

九.在IntelliJ Idea中使用 git(GitHub)

9.1 基础设置

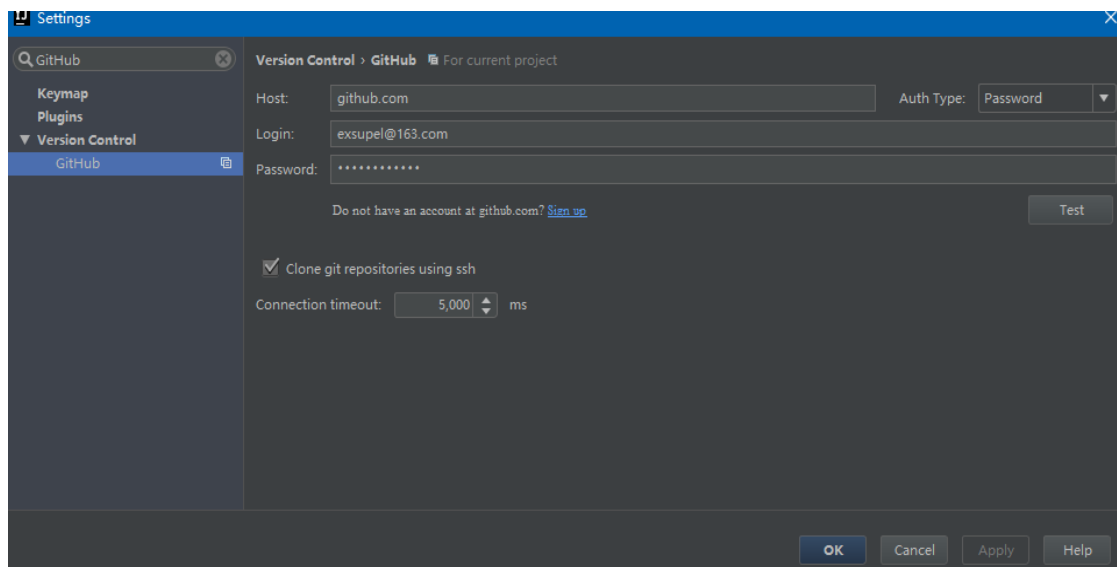
进入setting->Version Control->Git



Path to Git executable为你的git安装目录下的git.exe的路径，比如

C:\Program Files\Git\bin\git.exe

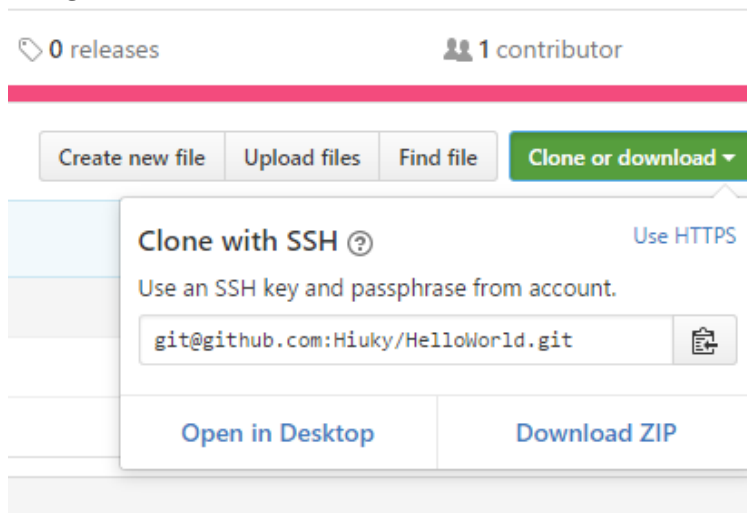
setting->Version Control->GitHub设置GitHub相关信息



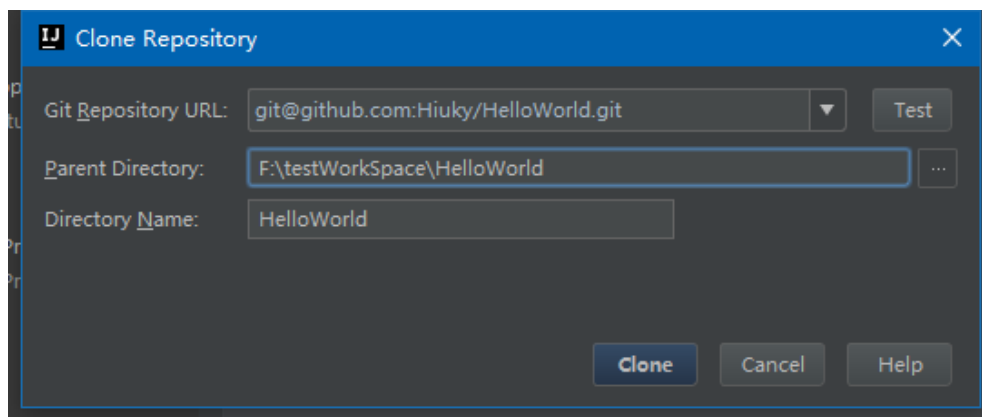
至此，配置完毕

9.2从GitHub克隆

在你的github项目上点击Clone or download复制SSH协议地址

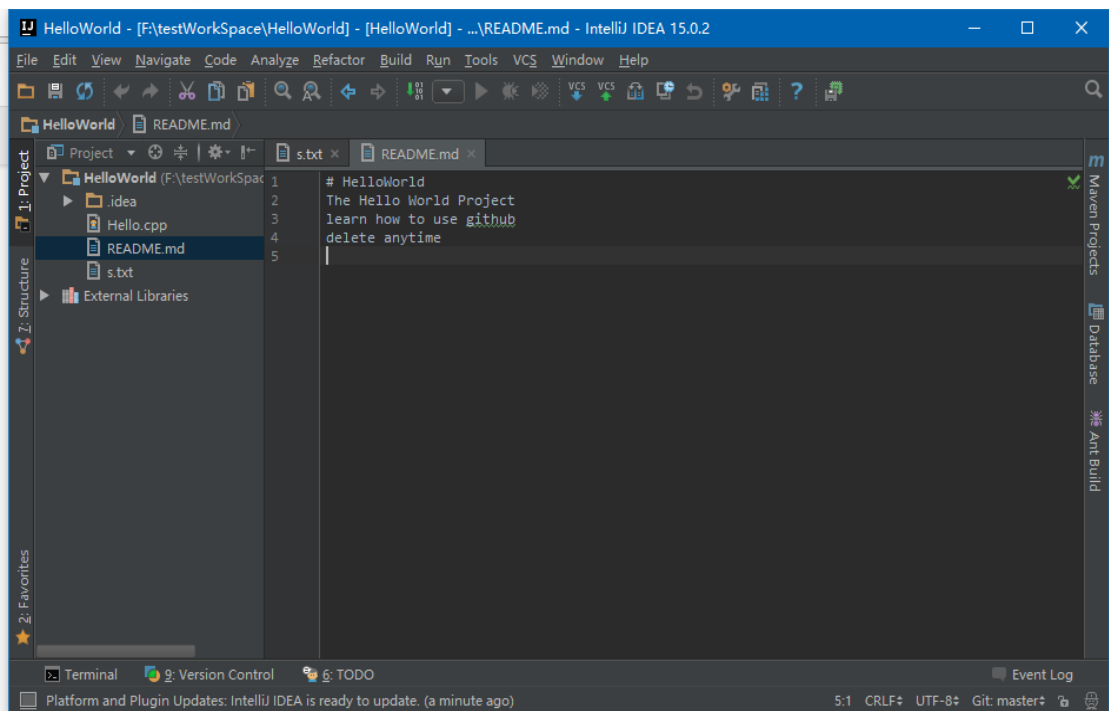


Idea从github clone



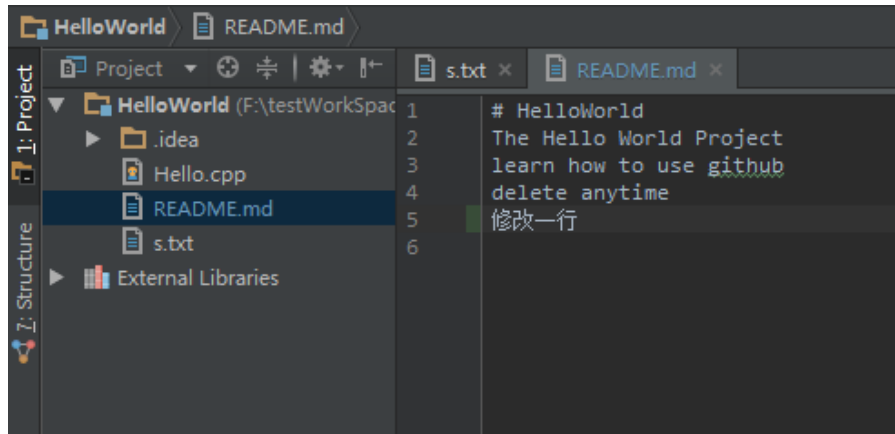
然后点击clone按钮，克隆项目到本地

然后打开该项目，即可看到和远程仓库一致的文件

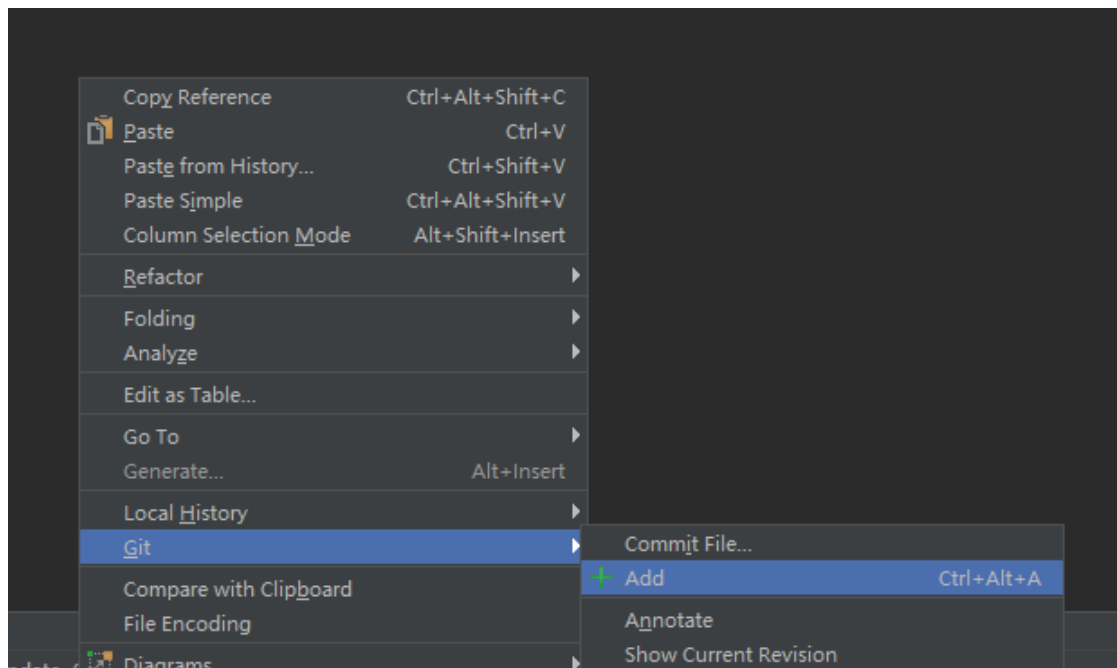


9.3修改add/commit

对文件进行修改后

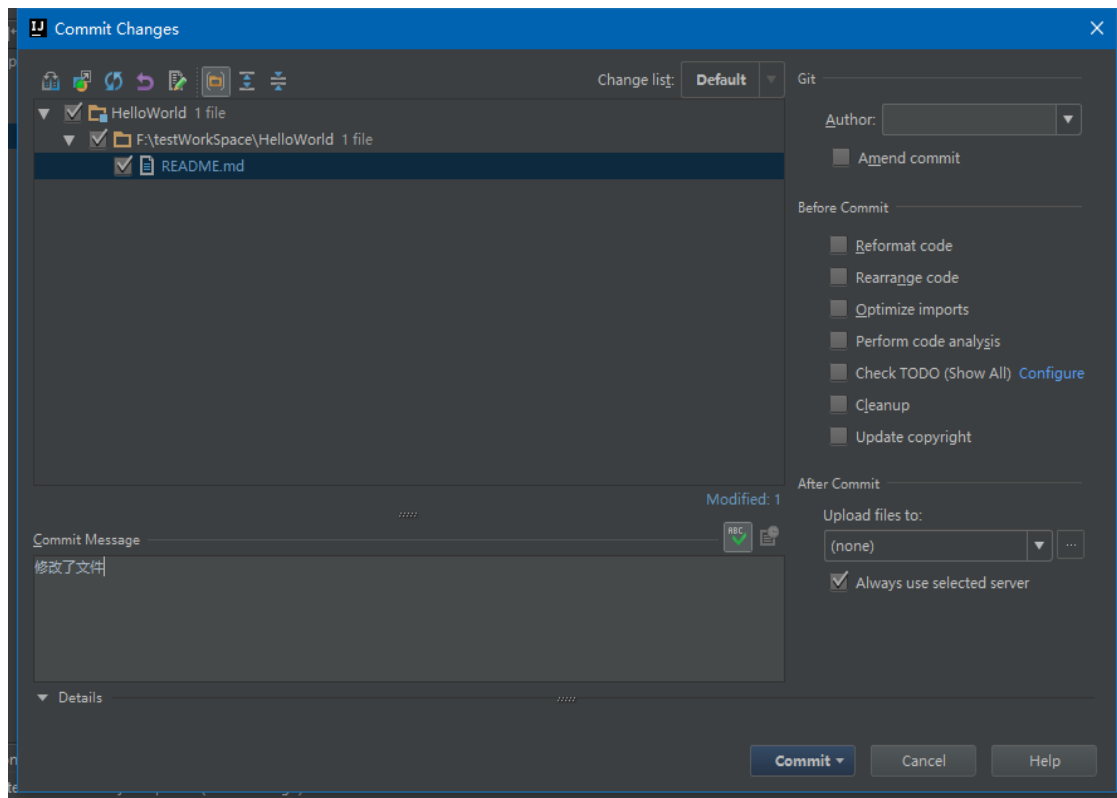


使用右键Git->Add

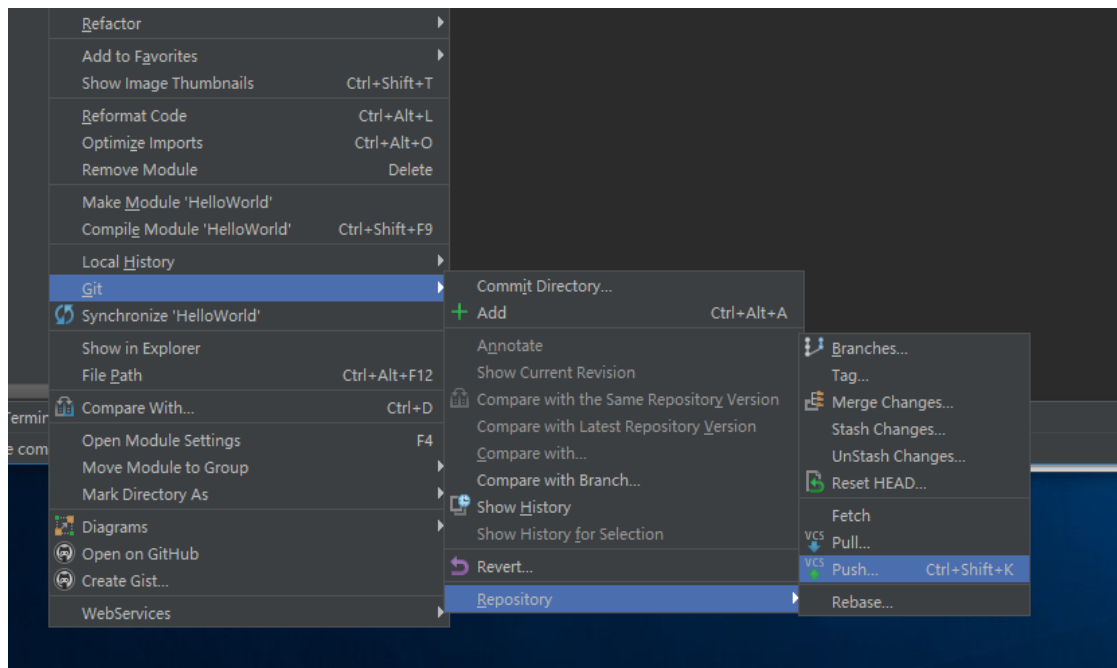


将文件添加到本地版本库

同样使用右键Git-Commit提交到本地版本库

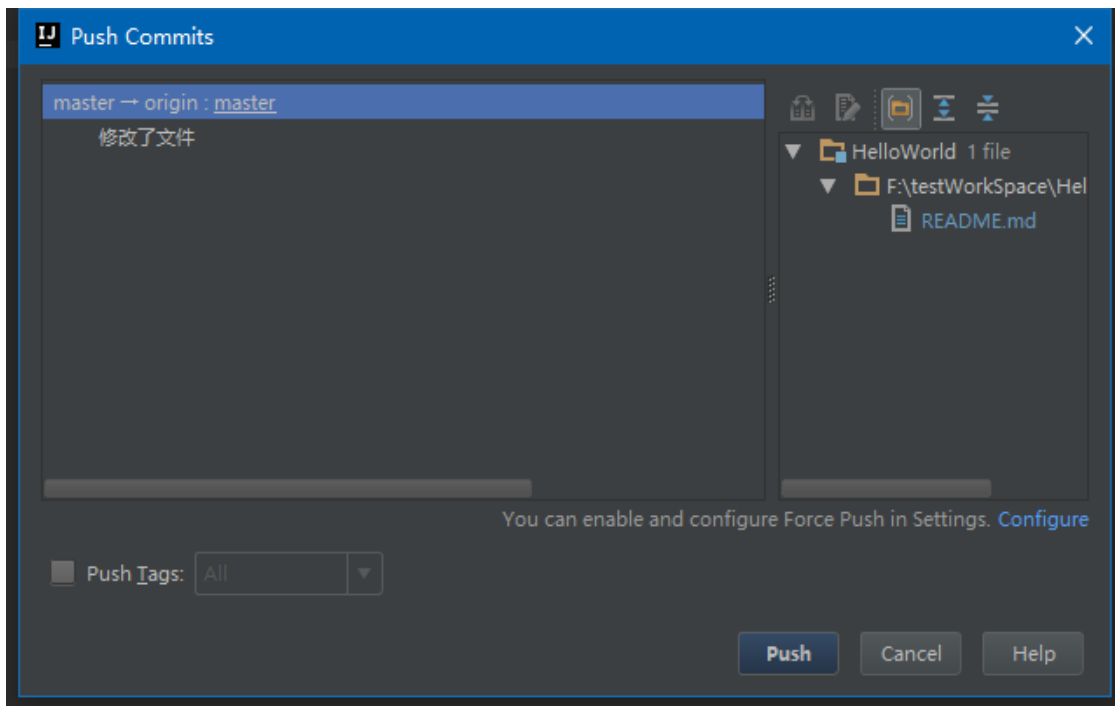


工作完成后，需要往远程仓库推送时，对项目使用右键



Git->Repository->Push

Push到指定分支即可



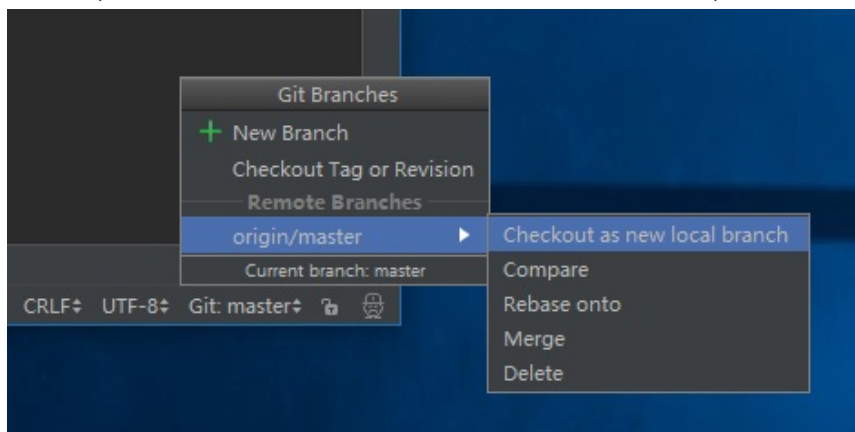
同时在GitHub上也同时看到更新

HelloWorld

The Hello World Project learn how to use github delete anytime 修改一行

9.4本地临时分支

新建本地临时分支(右下角显示的是当前分支，看到是在master分支上)

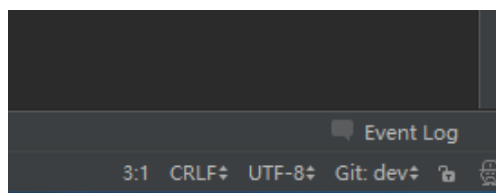


现在从master分支构建本地临时分支依次操作

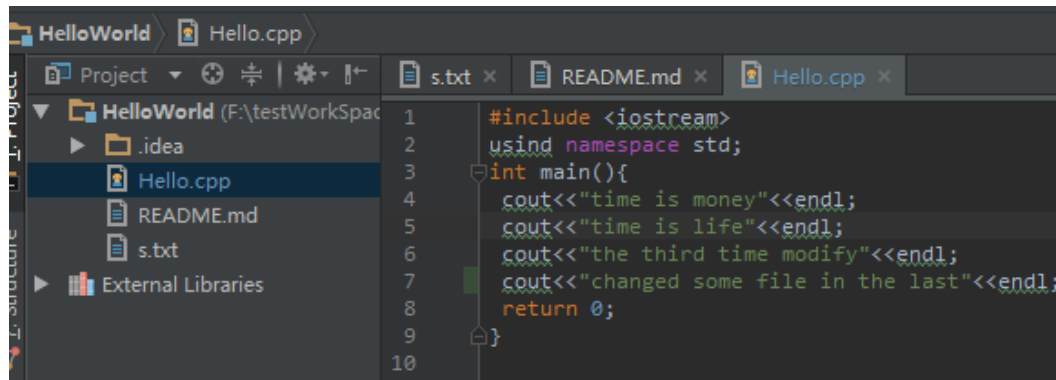
Git:master->origin/master->checkout as new local branch

这里我们起名dev

操作后，当前分支已经切换到了dev分支



在dev上修改文件

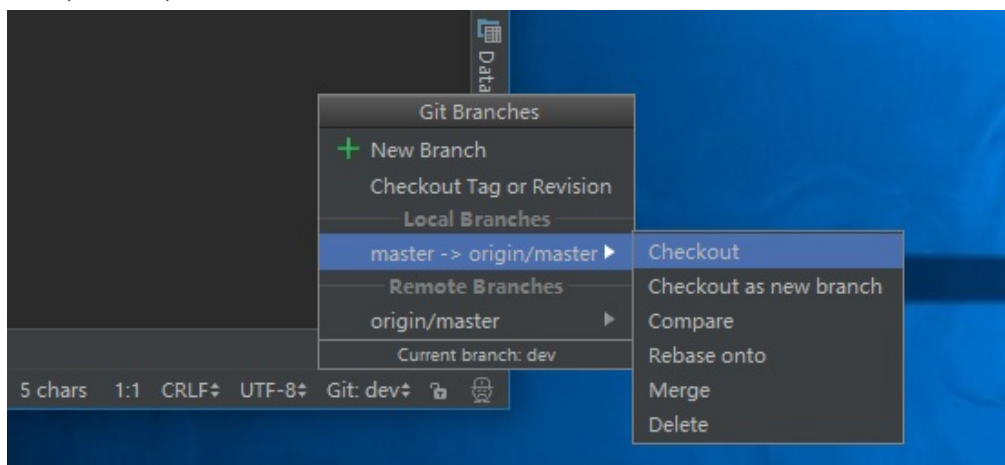


(蓝色表示有改动)

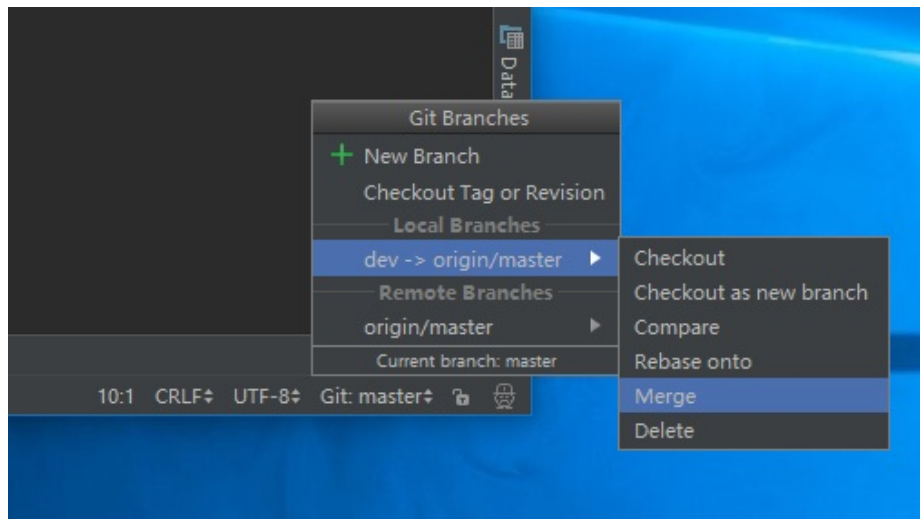
同样依次add 文件 commit 操作，或者继续工作然后再次add/commit操作

9.5 合并

再切换回(master)主分支

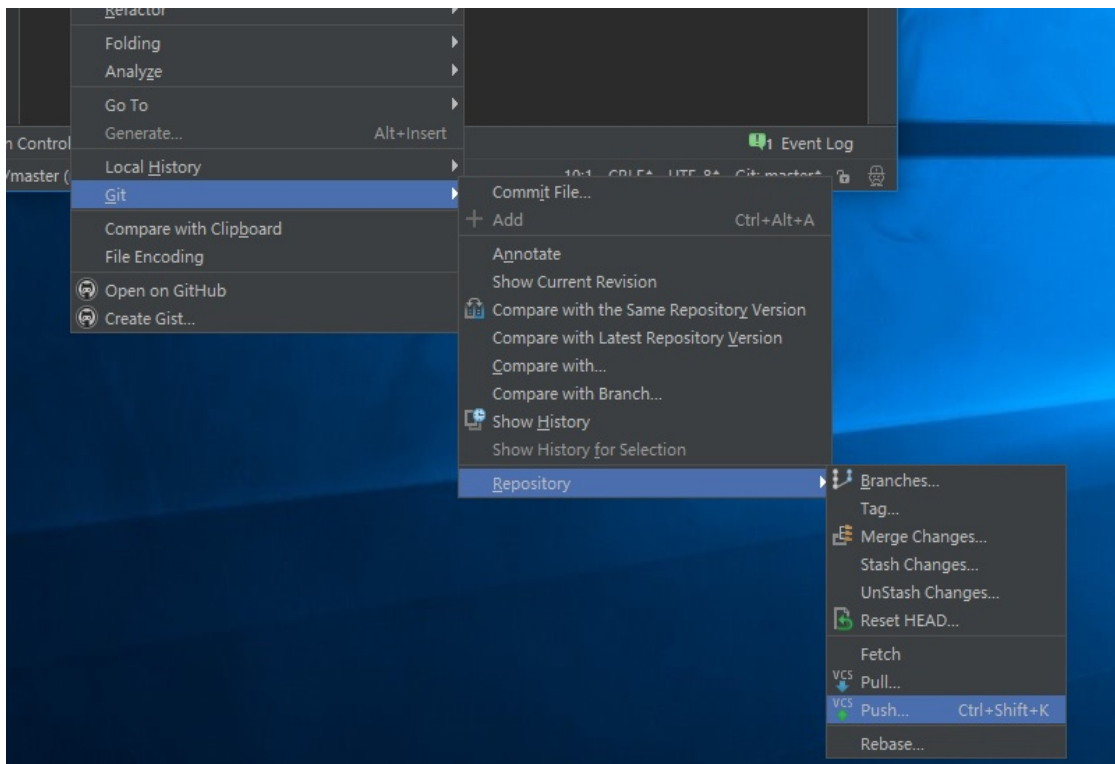


与dev分支合并



如果没有冲突，master分支将和dev分支内容一致

然后再次执行Push，将master分支推送到远程仓库



将master主分支推送到远程仓库

```

Fetching contributors...
Cannot retrieve contributors at this time

10 lines (9 sloc) | 209 Bytes
Raw Blame History

1 #include <iostream>
2 using namespace std;
3 int main(){
4     cout<<"time is money"<<endl;
5     cout<<"time is life"<<endl;
6     cout<<"the third time modify"<<endl;
7     cout<<"changed some file in the last"<<endl;
8     return 0;
9 }

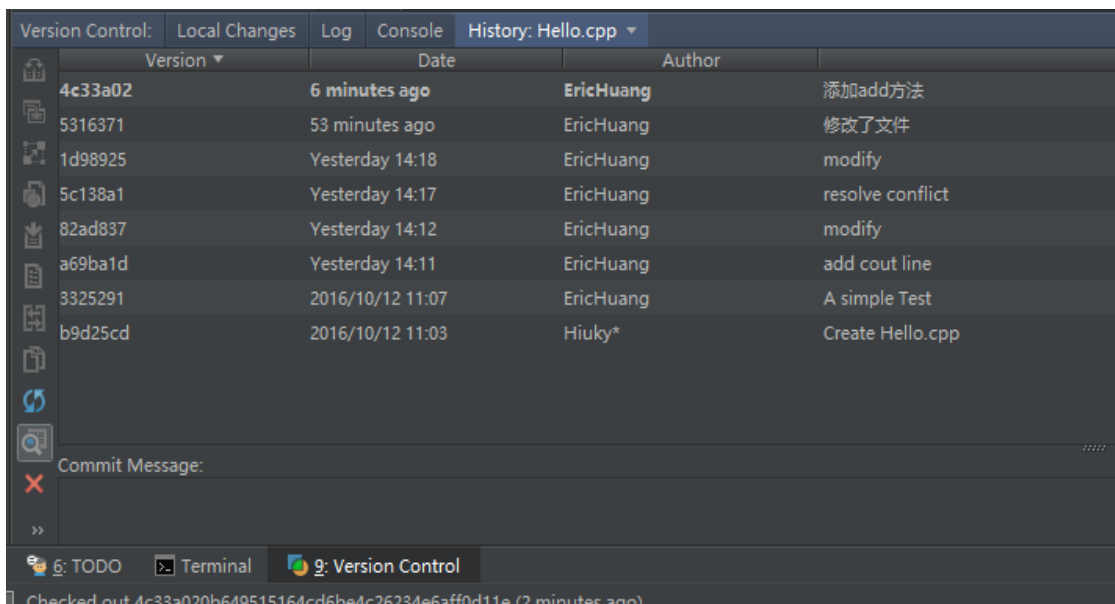
```

查看gitHub，发现文件已修改。

(通常，修改项目前应进行Git->Repository->Pull操作，将工作分支更新到最新版本)

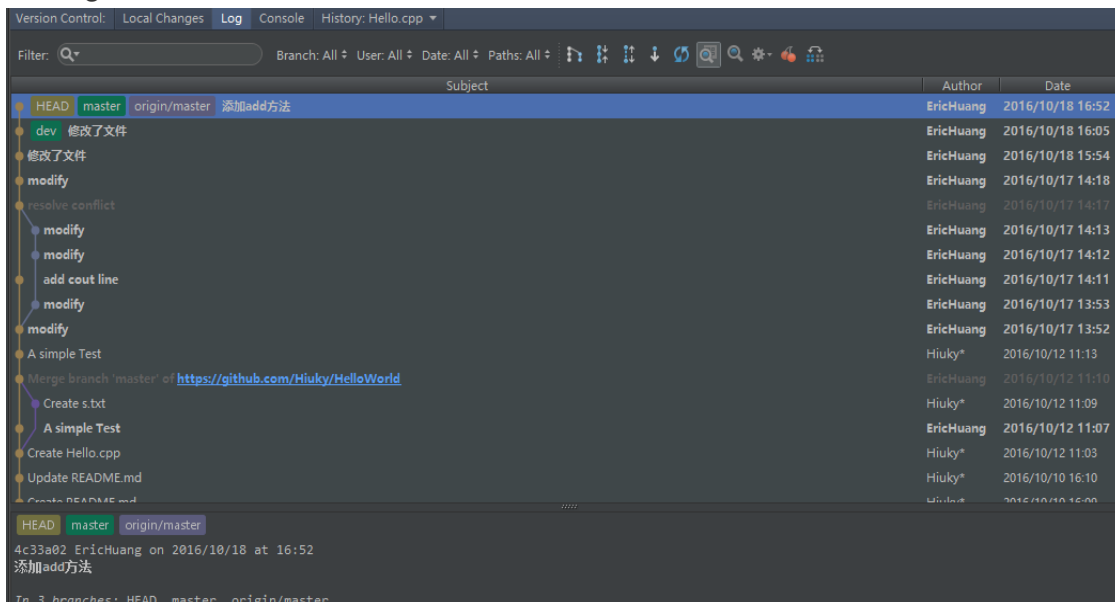
9.6 历史

查看版本历史，点击Version Control选项卡

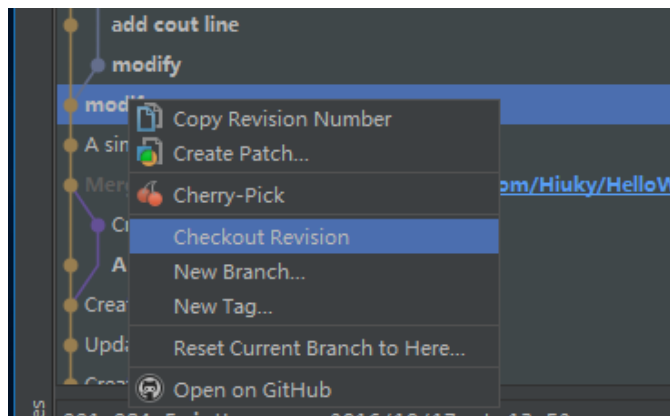


即可查看当前文件的历史操作信息

点击log列，可以查看提交日志和分支图形化详情



可以对任意节点右键操作Checkout Revision以回退到任意一个版本



十.附录

GitBash 常用命令一览

git add 添加

git commit -m "xxx" 注释提交

git log 提交日志

git log --pretty=oneline 提交日志在一行显示

git reset --hard HEAD~1 回退到上一个提交

git reset --hard 467dcab 回到某个commit

git reflog 查看命令历史

git checkout -- file 丢弃工作区修改

git rm 删除一个文件

暂存工作区

git stash 当前工作现场“储藏”起来

git stash pop 恢复并删除stash内容

git stash apply 恢复

git stash drop 删除stash内容

git stash list 列出stash列表

分支管理

git branch 查看分支

git branch <name> 创建分支

git checkout <name> 切换分支

git checkout -b <name> 创建+切换分支

git merge <name> 合并某分支到当前分支

git branch -d <name> 删除分支

git branch -D name 强行删除分支

git checkout -b dev origin/dev 创建远程分支到本地

git pull 从远程分支拉取

git branch --set-upstream dev origin/dev 指定本地dev分支与远程origin/dev分支的链接

创建分支，修改代码，合并并删除分支过程

git checkout master 切换到主分支

git checkout -b bug-001 创建bug-001分支

git add file 添加文件

git commit -m "fix bug 001" 提交文件并注释

git checkout master 切换到主分支

git merge --no-ff -m "merge bug fix 001" bug-001 bug-001分支合并到当前分支

git branch -d bug-001 删除bug分支

标签管理

git tag 列出标签

git tag v1 打一个新标签

git tag v1 6224937 给某次提交打tag

git tag -a v1 -m "tag desc" 6224937 指定某一标签打tag并注释

git show v1 看到文字说明

git tag -d v1 删除标签

git push origin tagname 推送标签到远程仓库

git push origin --tags 一次推送所有本地未推送标签

git push origin :refs/tags/tagname 删除远程标签

参考博客

<http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/>

GitHub Guide

<https://guides.github.com/activities/hello-world/>

Git Doc

<https://git-scm.com/doc>