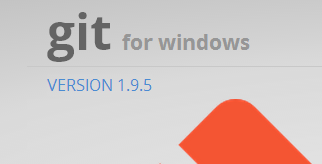
**安装Git**

如果你的系统是Linux的话，直接打开shell输入:

sudo apt-get install git

当然，大部分的系统估计都是Windows，这就需要我们到网上下载一个Git For Window了，可到下述网站下载：<https://git-for-windows.github.io/> 点击 Download，跳转到 Github ，下载对应安装包即可！



**打死不要用GUI的界面，只用命令行**

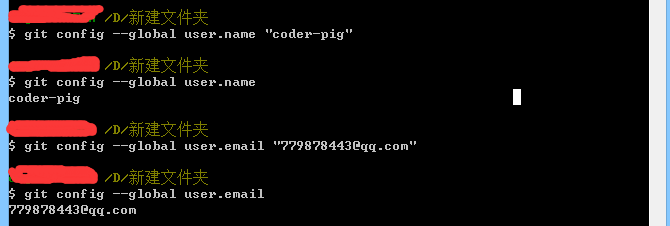
### 1.创建代码仓库

Step 1：先配置下我们的身份吧，这样在提交代码的时候Git就可以知道是谁提交的，命令如下：

git config --global user.name "XXX"

git congif --global user.email "XXX"

配置完成后，我们可以再次输入，不包括名称，可以看到我们已经配置成功了



第二步：在某个代码文件夹内部常见代码仓库。很简单，例如A是我们的文件夹，我们要建立A的代码仓库，那么，进去A内部，右键，git bash，继续git init，即可。输入ls –al，既可以看到.git文件夹，那就是我们创建的仓库，这个代码仓库其实是用来保存版本管理所需的一些信息，我们本地提交的代码都会提交到代码仓库中，于是乎我们可以选择还原到某个版本，当然，如果需要的话，我们还可以将保存在代码仓库中的代码推送那个到远程仓库中！比如GitHub!

**如果我们想删除代码仓库只需把这个文件夹删掉即可！**

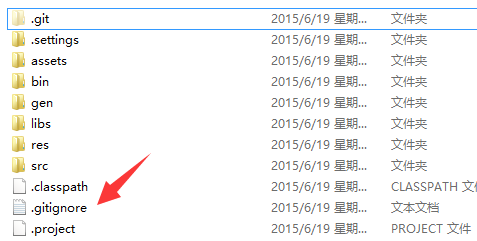
### 2.提交本地代码

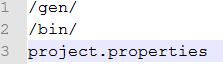
创建完代码仓库，接下来说下如何提交代码，**我们是先用add命令把要提交的内容都加进来，然后commit才是真的去执行提交操作！**命令例子如下，你可以一次次慢慢添加，当然也可以全部提交，直接git add .即可完成！ **注意，首先要有文件才能添加进去**

git add readme.txt

git commit -m "Wrote a readme file" // 捎带一句话

当然如果你可以add多个文件后再一次性commit，不过如果我们改动的文件很多的话，我们可以git add .一次添加全部，但有一些是几百年都不变一次的又或者自动生成的，比如lib，gen，bin文件夹等等，我们可以在代码仓库的根目录下创建一个名为.gitignore的文件，然后编辑里面的内容，把不需提交的文件忽略掉！





那么我们git add .的时候，这里的文件就不会add，另外可能你会觉的commit后面写-m "xxx"很麻烦，想偷懒，但还是写上吧！输入的是本次提交的一些声明，比如自己修改了些什么！就好像写代码的时候，你偷懒不写注释，过几天你连自己写的什么鬼都不知道...

### 3.查看修改内容

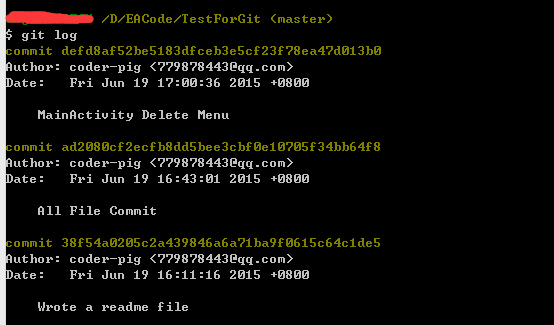
好吧，前面我们用git add提交了整个项目到本地仓库，接下来我们改点东西，然后使用git status可以查看 修改的部分，

### 4.查看提交记录

当然随着我们项目的深入，Commit的次数也会越来越多，可能你早已忘记每次提交都修改了什么内容， 没事，Git帮你记着呢，使用git log即可查看历史提交信息！键入

git log

回车：



我们取其中一小部分来分析：

commit defd8af52be5183dfceb3e5cf23f78ea47d013b0

Author: coder-pig <779878443@qq.com>

Date: Fri Jun 19 17:00:36 2015 +0800

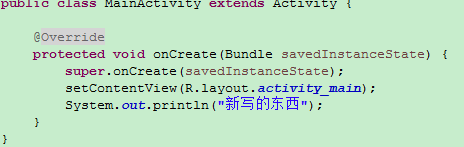
MainActivity Delete Menu

依次是：

* 此次提交对应的版本号
* 提交人：姓名 邮箱
* 提交的时间
* 提交版本修改的内容：就是我们commit -m "xxx"里的xxx

### 5.撤销未提交的修改

比如我们刚提交了一个版本，然后又乱七八糟地写了一堆东西，突然发现不小心误删了一些东西，然后ctrl + s保存了，这个时候是不是欲哭无泪，不过有Git，只需一个checkout命令即可撤销更改，当然是你还没add的情况，比如我们在MainActivity里随便添加一条语句，然后ctrl + s保存代码！



然后命令行键入：git diff：



嗯，这里可以看到我们改的内容，我们可以回去把这句代码删掉，但是如果改的有上千行你怎么改， 于是乎这个时候我们可以使用。这个是既没有add，也没有提交的情况。

git checkout src/com/jay/example/testforgit/MainActivity.java

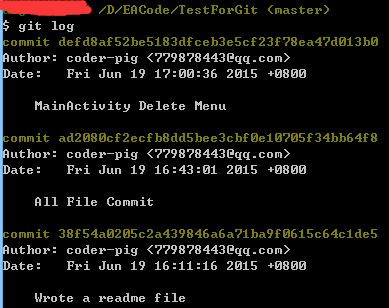
当然，如果我们已经add了的话，那么checkout是没任何作用的，我们要先取消添加才可以撤回提交，使用下述指令：

git reset HEAD src/com/jay/example/testforgit/MainActivity.java

git checkout src/com/jay/example/testforgit/MainActivity.java

### 6.版本回退

第五点我们教了大家撤销未提交的修改，但加入提交了，我们想回退到之前的某一个版本怎么办? 第四点中我们可以通过git log查看我们的提交记录，我们需要从这里获取一个版本号， 一般我们只需要前七位字符就够了；另外在Git中，用**HEAD**代表当前版本，上一个版本就是**HEAD^**， 再上一个版本就是**HEAD^^**依次类推！我们先Git Log看下版本历史先！



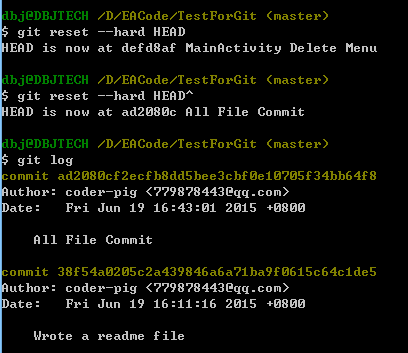
我们回到前一个提交的版本吧，依次键入下述指令：

git reset --hard HEAD

git reset --hard HEAD^

git log

这时看下我们的控制台：



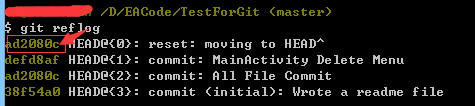
可以看到我们已经回退到了前一个版本了，当然你可以直接这样写：

git reset --hard ad2080c

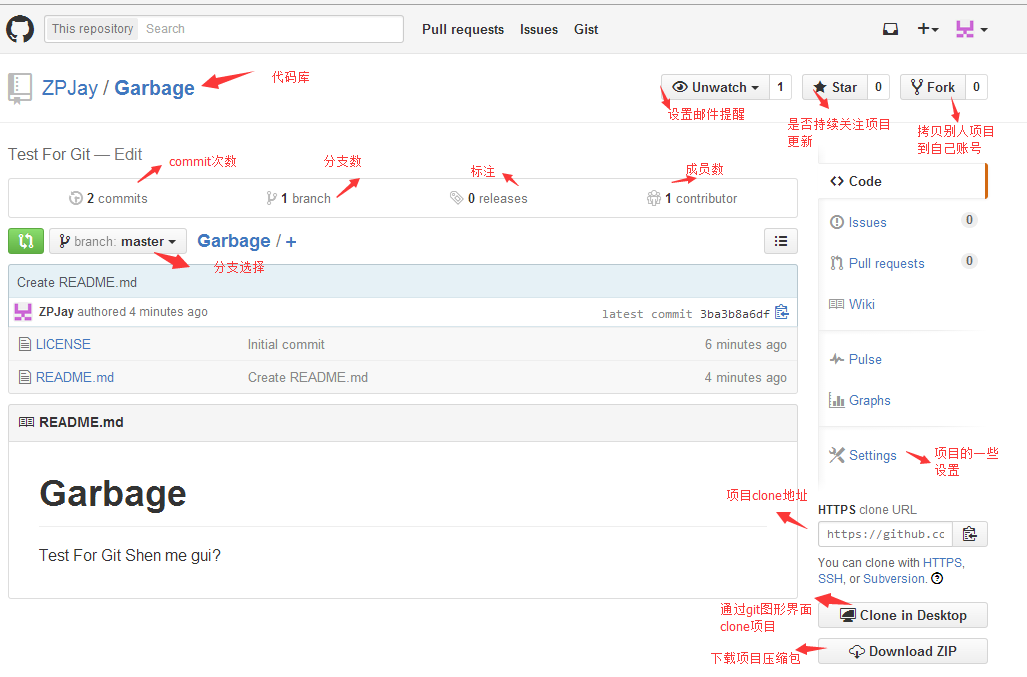
就是这么简单！回退后，你突然后悔了，想回退回新的那个版本， 可是遗憾的是，你键入git log却发现没有了最新的那个版本号，这怎么办呢... 没事，Git中给你提供了这颗"后悔药"，Git记录着你输入的每一条指令呢！键入：

git reflog

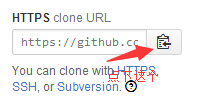
你会发现，版本号就在这里：



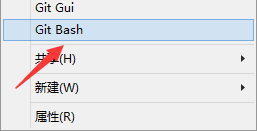
## Git之使用GitHub搭建远程仓库



## Clone代码库到本地

当然，你可以直接用图形化界面克隆，不过我还是喜欢通过命令行来Clone，先复制下Clone的地址 

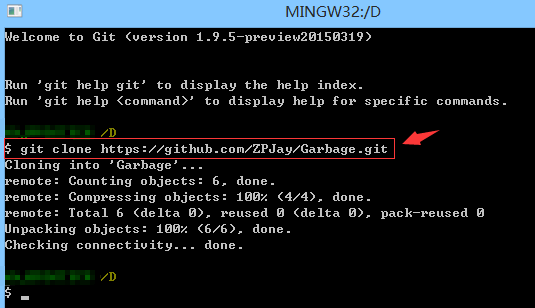
然后在某个地方，有键打开Git Bash：



键入：

git clone https://github.com/ZPJay/Garbage.git

然后可以看到我们的代码库就下载完成了：



#### 创建与合并分支

阅读: 655830

在[版本回退](http://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。

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：



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：



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：



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？

#### 创建与合并分支

阅读: 655830

在[版本回退](http://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。

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的代码到当前分支上来，也相当于当前分支的head指向的当前分支都去指向指定分支那里去了。。**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。

#### 创建与合并分支

阅读: 655830

在[版本回退](http://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。

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。

#### 创建与合并分支

阅读: 655830

在[版本回退](http://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。

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。

### 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>

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>