[【Git 第1课】 什么是Git？ 1](#_Toc374434586)

[【Git 第2课】 GitHub是什么？ 3](#_Toc374434587)

[【Git 第3课】 安装Git 6](#_Toc374434588)

[【Git 第4课】 创建Git仓库 7](#_Toc374434589)

[【Git 第5课】 Git工作流 8](#_Toc374434590)

[【Git 第6课】 文件状态 10](#_Toc374434591)

[【Git 第7课】 忽略文件 13](#_Toc374434592)

[【Git 第8课】 移除文件 15](#_Toc374434593)

[【Git 第9课】 撤销修改 16](#_Toc374434594)

[【Git 第10课】 版本回退 18](#_Toc374434595)

[【Git 第11课】 远程仓库 20](#_Toc374434596)

[【Git 第12课】 抓取/推送数据 22](#_Toc374434597)

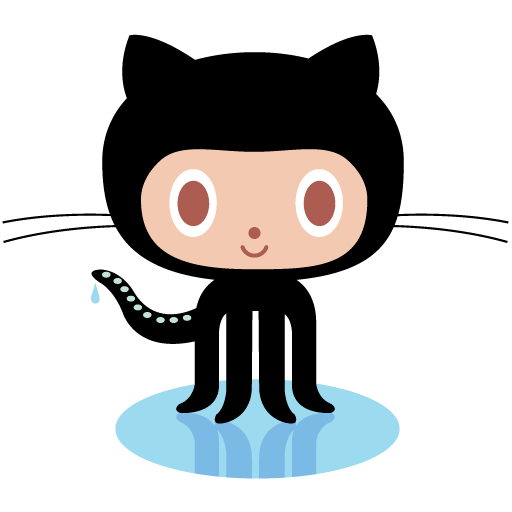
# 【Git 第1课】 什么是Git？

  
今天起，我会在微信推送文章里穿插一个新的系列：Git。和之前Python入门教程不同，这个系列需要有一点编程的经验。倒不是因为它很难，只是如果没有开发过稍大一点的项目，或者没有和别人合作开发过，你可能无法理解Git的用处，对其中涉及的内容也会感到不知所云。  
  
这个系列不会天天更新，平时仍然会有一些基础的编程知识，以及编程学习相关的文章推送。大家各取所需。刚刚入门的朋友也不用着急，按部就班地保持自己的节奏就好，学习是件长久的事情，欲速则不达。  
  
学习Git跟你使用的语言无关，只要你参与到程序开发中，就很有可能接触到Git，或者可以用Git帮助你进行开发。因此在这一点上，你不用再纠结于是该 选择C++、Java、Python……当然，Git也有同类产品可供选择，但对于目前的你，有足够多的理由去学习Git，哪怕以后你不再使用它。  
  
说了这么多，还没有进入今天的正题：Git到底是个什么东西？  
  
Git是一个版本控制系统（Version Control System）。  
  
那什么又是“版本控制系统”呢？  
  
想象一下这样的场景：你再写一个程序，这个程序对你来说有一点“大”，不是一个晚上就能解决的。比如我们之前的打飞机小游戏。于是编写的过程中，你需要保 存代码，下次接着写。另外，当你花了两个晚上，完成了最基本的功能后，第三天晚上，你又想尝试给你的游戏增加点新功能，比如你的飞机可以扔炸弹消灭屏幕上 的所有敌人。那么你就得在之前的代码上做改动。  
  
这时候，不幸的事情发生了！  
  
你发现你改动了一些代码之后，程序无法正常运行了。你想算了，还是恢复到之前那样吧。但是代码被改动的地方比较琐碎，你发现把印象中所有的改动都改了回去，程序一样无法运行。于是你就陷入了抓狂。  
  
即使你不是一个程序员，你也可能遭遇过类似的“悲剧”，比如误删了word文档中的一段话后手贱地点了保存，之后又想把那段找回来的时候才追悔莫及。人世间的事情大多如此吧……  
  
于是，吃过苦头的人会给自己留条后路，再保存文件的时候选择“另存为 Save As”而不是直接在原文件上保存。于是你的文件夹中就出现了诸如：  
report131106.doc  
myslide-updated.ppt  
loveletter\_old001.txt  
之类的文件。  
  
其实这就是一种最简陋的版本控制系统。  
  
本人当年上学的时候，第一次接触到版本控制这个概念时，就是用这种方式来解决的。我们从团队中挑选出一人（好吧，就是我）专门负责维护一台电脑上的一个文件夹。文件夹里分门别类地保存着项目的文档和代码的不同版本。  
  
现在回想起来，这是怎样一种勤劳而又愚昧的做法。你需要去小心每个版本不要放错了位置，每天开发完要把代码发给专人整理，甚至在宿舍断网的时候要用u盘去copy最新的代码，再把最新的代码交给需要进行修改的人……  
  
但其实早就不必如此。版本控制系统就是把你从这片苦海中拯救出来的工具。  
  
通过版本控制系统，可以有效地追踪文件的变化。当代码出错的时候，可以很容易地恢复到之前的状态。对于多人协作开发的项目来说，版本控制系统更是必不可 少。它保证每个人都能够获取最近更新的代码，每个人开发的代码可以更方便地增加到项目中，避免不同代码之前产生的冲突，减小沟通成本。  
  
其实它做的事情本质还是在某个地方，记录下所有文件的所有版本，只不过比在电脑上建很多文件夹的方法更合理、更便捷、更稳定。让你可以把精力集中在开发代码本身，而不用关注其他琐碎的问题。  
  
而Git就是时下最流行的版本控制系统。  
  
关于Git有哪些具体的功能和特性，在以后的课程中会具体分析。另外，说Git就不能不说到GitHub。它又是什么？请看下节课。

# 【Git 第2课】 GitHub是什么？

说Git就不能不说GitHub。  
  
以下摘录一段网上的文章片段，介绍一下GitHub。  
来自阳志平的网志《[如何高效利用GitHub](http://www.yangzhiping.com/tech/github.html)》  
  
**Q1：GitHub是什么**

**A1：一家公司**



位于旧金山，由[Chris Wanstrath](https://github.com/defunkt), [PJ Hyett](https://github.com/pjhyett) 与[Tom Preston-Werner](https://github.com/mojombo)三位开发者在2008年4月创办。迄今拥有59名全职员工，主要提供基于git的版本托管服务。

在此之前，它是由[Tom](https://github.com/mojombo)与[Chris](https://github.com/defunkt)在本地程序员聚会中，开始的一个用于托管git的项目。正如每个伟大的传奇都开始于一场冒险，Tom在这篇文章[我如何辞掉微软30万年薪邀约，创办GitHub](http://tom.preston-werner.com/2008/10/18/how-i-turned-down-300k.html)中谈到：

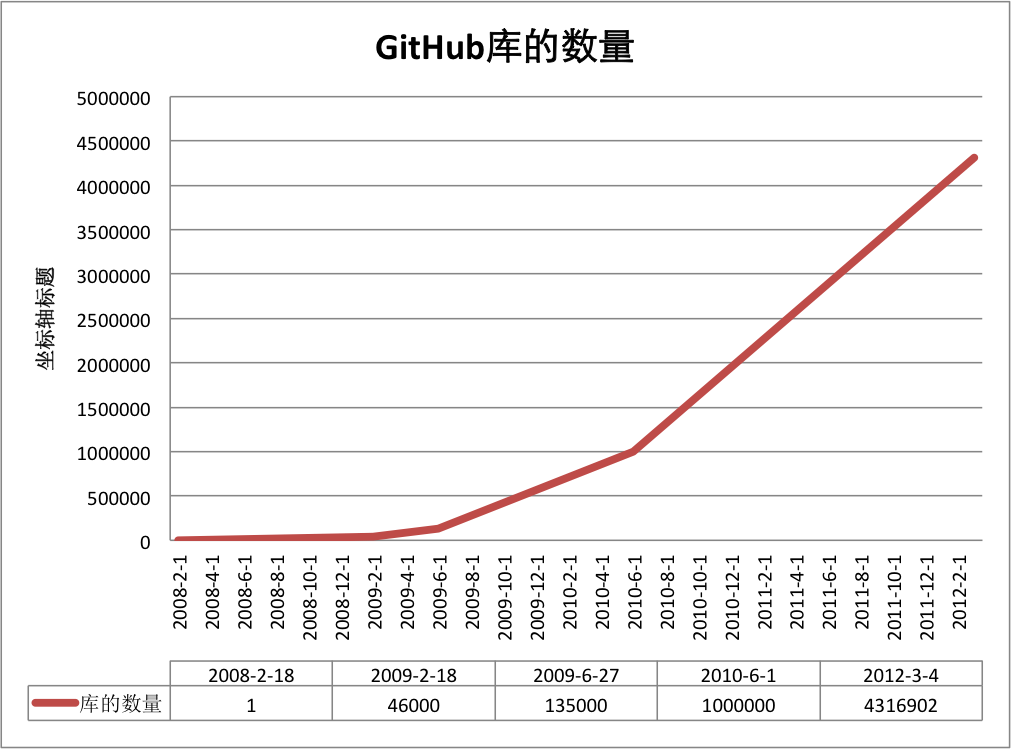
当我老去，回顾一生，我想说，“哇，那是一场冒险“；而不是，“哇，我真的很安稳。“

另一位创始人[Chris](https://github.com/defunkt)也详细描述了[GitHub初创的前因后果](https://gist.github.com/67060)，他说道：

Do whatever you want.

于是，在2008年4月10号这一天，GitHub正式成立。

目前看来，GitHub这场冒险已经胜出。根据来自[维基百科关于GitHub的描述](http://zh.wikipedia.org/wiki/GitHub)，我们可以形象地看出GitHub的增长速度：



今天，GitHub已是：

* 一个拥有143万开发者的社区。其中不乏Linux发明者[Torvalds](https://github.com/torvalds)这样的顶级黑客，以及Rails创始人[DHH](https://github.com/dhh)这样的年轻极客。
* 这个星球上最流行的开源托管服务。目前已托管431万git项目，不仅越来越多知名开源项目迁入GitHub，比如Ruby on Rails、jQuery、Ruby、Erlang/OTP；近三年流行的开源库往往在GitHub首发，例如：[BootStrap](https://github.com/twitter/bootstrap)、[Node.js](https://github.com/joyent/node)、[CoffeScript](https://github.com/jashkenas/coffee-script)等。
* alexa全球排名414的网站。

-----------------------------  
  
简单来说，GitHub就是一个基于Git的代码托管网站。由于其上聚集了大量的优秀代码和优秀程序员，使得它成为了一个极具价值的开发者社区。  
  
在GitHub上，你可以免费托管你的代码，进行版本控制，同别人协同开发。代码必须开源，付费用户可以选择私有项目。  
  
除此之外，你还可以通过GitHub来寻找牛人、参与开源项目，进而提升你的编程能力。你可以在上面建立你的个人博客。你在GitHub上的参与度和代码贡献，会在求职的时候为你加分不少。  
  
在后续关于Git的课程中，我们将会用到GitHub作为代码托管服务器。  
  
另外，国内也有一些Git托管网站，比如CSDN code，Git@OSC，gitcafe等，可以去尝试一下，尤其是如果需要私有项目的话。

# 【Git 第3课】 安装Git

|  |  |
| --- | --- |
| |  | | --- | | git-logo.png  要用Git，首先肯定要安装它。简要说一下Win，Mac和Linux三种平台的安装方法。  Windows  Git的官网(<http://git-scm.com>)提供了Windows下Git客户端的安装包。也可以去msysGit项目的页面上下载： <http://msysgit.github.com/>  安装好之后，你会得到一个Git的命令行工具Git-Bash和一个图形界面工具Git-GUI。在后面的教程里，会主要以命令行下Git的使用为主。掌握基本的命令之后，使用图形界面也就没什么问题了。  另外，GitHub提供了一个叫做GitHub For Windows的工具。这个工具也能实现Git的功能，并且提供了一个更便于管理GitHub上项目的图形化工具。也可以选择安装这个，地址是： <http://windows.github.com/> 不过它提供的是一个在线安装包，网络不好的时候就会安装失败。我自己安装了很多次才成功。  Mac OS  Git官网同样提供了Mac版的安装包，或者去这里下载： <http://code.google.com/p/git-osx-installer>  Mac上还有其他方法，对于已经在Mac上安装过MacPorts的用户，可以通过如下命令安装Git： sudo port install git-core +svn +doc +bash\_completion +gitweb  如果是安装过homebrew，可以直接使用如下命令安装： brew install git  Linux  Git最初是Linux的作者Linus Torvalds为了管理Linux内核开发而创造出来的。所以相对来说，Git更适用于Linux/Unix平台。  各类Linux系统的包管理工具中都提供Git的安装。如： Debian/Ubuntu $ apt-get install git  Fedora $ yum install git  Git的官网上提供了《Pro Git》一书，并且有中文翻译版本，这是一本关于Git由浅入深的详细教程，大家可以结合此书来学习Git。关于上述安装的内容，它里面也有说明。  当你安装好合适的Git客户端之后，可以通过Git命令行，到你电脑上的某一个文件夹中（关于命令行下的简单操作，可以查看第20课）。执行： git clone <https://github.com/crossin/CrossinClass.git>  然后就可以在文件夹中看到我建好的项目，里面暂时还没什么内容，只有一本咱们教室的朋友帮忙整理的一个python前60课合集的word文档。以后我会在里面陆续添加内容，大家也可以通过学习git的过程来同我一起添加和修改。 | |
|  |

# 【Git 第4课】 创建Git仓库

上次安装了 Git 之后，我让大家用命令：

1. git clone https://github.com/crossin/CrossinClass.git

把我在 GitHub 上建立的项目“复制”到本地。这就是创建 Git 仓库的一种方法：从现有的仓库“克隆”。  
  
Git 中所谓的“仓库”（repository），就是一个受 Git 版本控制的目录。仓库中文件的改动都将被 Git 所记录。  
  
通过 git clone 命令，可以从一个已有的 Git 仓库克隆出一个本地的镜像仓库。如果想**指定创建的本地文件夹名称**，可以在仓库的 url 地址后面加一个参数。如：

1. git clone **https**://github.com/crossin/CrossinClass.git MyClass

这里，**我们在数据传输是用了 https 协议**。除此之外，**Git 还支持 git:// 或 user@server:/path.git 的方式进行传输，暂时先不细**说。  
  
  
除了克隆已有仓库，还有一种方法就是创建一个新的 Git 仓库。

在**想要进行版本控制的目录下**执行：

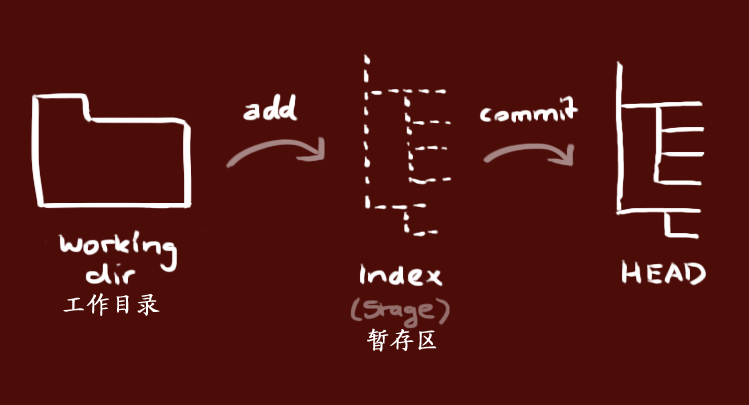
1. git init

新的仓库就被初始化了。这时目录下会多出一个 .git 的目录，它里面放了 Git 所需要的一些文件，我们也先不去管它。  
  
这时候，你已经有了一个新的 Git 仓库。往里面放一个 readme.txt，随便写点内容。然后在目录下执行：

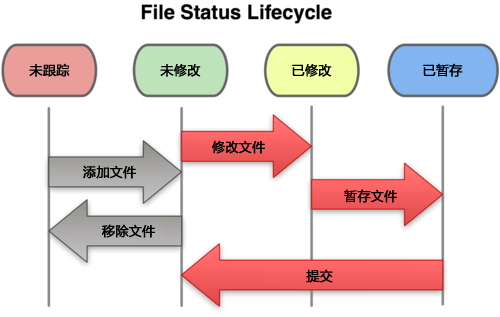
1. git add readme.txt
2. git commit -m 'my first commit'

OK。虽然你可能还很莫名，但你确实已经开始用 Git 管理这个 readme.txt 文件了。至于这两行命令的含义和这个文件的状态，请听下回分解。

# 【Git 第5课】 Git工作流

上次课中，我们创建了自己的Git仓库，并向其中添加了一个readme.txt的文件。在继续深入介绍Git的使用方法之前，有必要先来讲一讲Git的工作流。  
  
在你自建的**Git本地仓库中**，有三个区域：本地目录、暂存区、HEAD。要搞清Git的工作流就要理解这三个区域的作用。  
  
  
  
**工作目录**，就是受Git控制的文件夹，所有被跟踪记录的文件都包括在其中；  
  
**暂存区**（index或者stage），类似一个缓存区域，临时保存你做的改动；  
  
**HEAD**，指向最近一次提交后的结果。  
  
用一个比喻来描述一下这三个区域：假设你是一个学生，你需要写一门课的作业，作业有好几张试卷和几本练习题，写好之后把作业交给老师。  
  
你把所有作业放在桌上开始写，这里就是你的“工作目录”。当你写完一张试卷，把它放进你的书包里，它就进入了“暂存区”。等你把作业都写完了，放在书包里带到学校，全部交给了这门课的老师，他就是HEAD所指向的位置。

而对于**项目中的文件**来说，分为未跟踪（untracked）和已跟踪（tracked）两种状态。  
  
未跟踪的文件的改动不受到Git的版本控制。

而已跟踪的文件，即被纳入版本控制的文件，又分为未修改（unmodified）、已修改（modified）、已暂存（staged）三种状态。  
  
  
  
当在工作目录中**新加入一个文件**时，它**处于未跟踪状态**，这表示其没有纳入Git的版本控制。**通过 git add 命令可以将其加入跟踪，**并**同时**放入暂存区。  
  
一个已经被跟踪的文件，如果没有做过新的修改，就是未修改状态。一旦对其做了改动，就变成了已修改状态。通过 git add 命令可以将已修改的文件放入暂存区。  
  
初次克隆某个仓库时，工作目录中所有文件都是**已跟踪且未修改**的状态。

**git commit** 命令会**将暂存区中的文件提交至HEAD**所指向的分支。关于分支的概念，我们后面再说。当被commit之后，暂存区的文件将回到未修改状态。  
  
另外，也可以通过 git rm 将**已经跟踪的文件**从Git中移除。

再来套用前面的那个例子：作业里要交的试卷和练习题都处于已跟踪状态，那些不用交的就是未跟踪。当你写了某张试卷之后，它就变成了已修改。写完了把它放进 书包，它就是已暂存。等你把它提交给老师之后，假设老师瞬间就把它批改完并发还给你，那这时对你来说，它又重新成为了未修改。  
  
  
git status 命令是用来查看仓库中文件的状态。在命令行中，你的工作目录下运行此命令，可以看到输出文件状态信息。试试执行这条命令，然后修改readme.txt后再试试，看看有什么不同。下节课我们结合它来具体分析状态的变化。

# 【Git 第6课】 文件状态

在第4课的最后，我们用以下两条命令：  
  
git add readme.txt //跟踪文件  
git commit -m 'my first commit' //提交修改  
  
向我们新建的git仓库中添加了对readme.txt文件的版本控制。现在，我们要试图对仓库中的文件进行一些改动，同时观察它们的状态变化，以此实践上节课中说的git工作流。  
  
首先，在命令行的本地仓库工作目录下执行命令：  
  
git status //查看状态  
  
你会看到：  
  
# On branch master  
nothing to commit (working directory clean)  
  
很好，你的目录是干净的，没什么可提交的。说明所有的文件都是未修改的状态，没有未跟踪的文件，也没有修改过未提交的文件。输出的提示还告诉你，目前在branch master上。关于branch的概念我们将在后面的课程中介绍。  
  
接下来，随便用一个文本编辑器去修改一下readme.txt里的文字。另外，再向目录中新建一个test.py文件（也可以从别处copy一个过来）。  
  
再次执行：  
  
git status  
  
得到输出：  
  
# 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  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#        test.py  
no changes added to commit (use "git add" and/or "git commit -a")  
  
提示是比较清楚的：readme.txt被修改过了，还有**untracked**的test.py。提示里也说了，用"git add <file>..."可以添加要提交文件。git add的意思就是，把文件添加到暂存区，也就是我们说的暂存。  
  
那么我们就来用这条命令：  
  
git add readme.txt  
git add test.py  
  
然后继续：  
  
git status  
  
再来看结果：  
  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#        modified:   readme.txt  
#        new file:   test.py  
#  
  
**"Changes to be committed"里列出的就是已暂存的文件**，它们**将在commit时被提交。**  
执行：  
  
git commit -m 'test file status'  
  
将修改和增加的文件提交。-m**后面的参数是提交时的注释**。输出提示会告诉你这次提交做了哪些改动。  
  
最后再来执行一遍：  
  
git status  
  
可以看到又回到了最初未修改的状态。刚刚做的所有改动都已经被git所记录。  
  
通过命令：  
  
git log  
  
可以查看到之前提交的历史记录。  
  
  
在前面的过程中，可能会因为你所用编辑器的备份机制，在目录中产生一些奇怪的文件，比如readme.txt~。这类文件也会被列在未跟踪的列表中。下节课我们来说下如何避免这些我们不想关注的文件，以及其他一些在提交过程中的常用到的命令。

# 【Git 第7课】 忽略文件

在文件夹中，经常会有些“其他”文件，比如上节课最后提到的，编辑器产生~结尾的备份文件，或者一些临时文件。又可能，某些文件我们只是在本地使用，并不 想提交到远程的仓库中被别人看到。但如果我们不跟踪这些文件，Git会一直很执着地提醒我们工作目录中有一些未被跟踪的文件。如果你想清净一点，就可以通 过.gitignore文件忽略你不想看到的文件。  
  
要感受一下忽略文件的效果，首先你得有打算忽略的文件，所以我们强行创建一个叫做readme.txt.tmp的文件，假设它是readme.txt产生的临时文件。  
  
git status 看一下效果，同上节课中一样，在untracked files里列出了这个文件：  
  
# On branch master  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#        readme.txt.tmp  
nothing added to commit but untracked files present (use "git add" to track)  
  
然后我们在工作目录中新建一个.gitignore文件，在里面写上一行：  
  
\*tmp  
  
保存退出，再运行 git status：  
  
# On branch master  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#        .gitignore  
nothing added to commit but untracked files present (use "git add" to track)  
  
readme.txt.tmp文件消失了，untracked files里只剩下刚创建的.gitignore文件。  
  
来看一下《Pro Git》里对.gitignore格式规范的解释：  
  
所有空行或者以注释符号 ＃ 开头的行都会被 Git 忽略。  
可以使用标准的 glob 模式匹配。  
匹配模式最后跟反斜杠（/）说明要忽略的是目录。  
要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。  
  
所谓的 glob 模式是指 shell 所使用的简化了的正则表达式：  
星号（\*）匹配零个或多个任意字符；  
[abc] 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；  
问号（?）只匹配一个任意字符；  
如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）。  
  
所以我们前面写的 \*tmp，就是指忽略所有以tmp结尾的文件。  
  
再来看一个 .gitignore 文件的例子：  
  
# 此为注释 – 将被 Git 忽略  
# 忽略所有 .a 结尾的文件  
\*.a  
# 但 lib.a 除外  
!lib.a  
# 仅仅忽略项目根目录下的 TODO 文件  
# 不包括 subdir/TODO  
/TODO  
# 忽略 build/ 目录下的所有文件  
build/  
# 忽略 doc 目录下的所有 .txt 文件  
# 会忽略 doc/notes.txt   
# 但不包括 doc/server/arch.txt  
doc/\*.txt  
  
当然，你也可以把.gitignore文件本身从git中忽略。但我不建议这么做，而是所有协作开发者使用统一的规范，避免有人因为没写.gitignore而提交上去一些奇怪的文件。  
  
在项目一开始就把.gitignore设置好，不仅是为了好看，也是避免不必要的文件对代码产生干扰。比如有些需要本地生成的文件，如果放在远程仓库中被其他人下载使用，也可能会导致程序无法正常运行。

# 【Git 第8课】 移除文件

有进就有出，有借就得有还。有添加文件，自然也必须得有移除文件。当不想继续对某个文件进行版本控制的时候，就需要把它从 Git 中移除。  
  
从 Git 中移除一个文件，本质上做的事情就是把这个文件从暂存区中删除，然后提交。完成这个任务要用到命令 git rm。  
  
为了演示删除，我们先向工作目录中添加一个待删除的文件 to\_be\_deleted.txt，然后 git add，git commit 把它提交到仓库中。  
  
然后我们再将它从目录中删除。  
  
运行 git status 看下状态：  
  
# 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:    to\_be\_deleted.txt  
#  
no changes added to commit (use "git add" and/or "git commit -a")  
  
Git发现了一处文件改动：有个文件被删掉了。但这仅仅是从你的工作目录中删除了文件，而 Git 仍然在跟踪这个文件，并将会一直提示这个文件的删除状态。  
  
同 git add 添加文件至暂存区类似，用 git rm 命令把文件从暂存区中删除：  
  
git rm to\_be\_deleted.txt  
git status  
  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#        deleted:    to\_be\_deleted.txt  
#  
  
删除 to\_be\_deleted.txt 文件的操作已经被记录在了暂存区，换句话说就是，这个文件被从暂存区删除。接下来：  
  
git commit -m 'remove file'  
  
这样一来，这个曾经被跟踪的文件就从 Git 中移除了，以后的版本就没它什么事了。  
  
那么如果我们是手滑误删了文件呢？没关系，这也是我们使用版本控制的重要原因之一——恢复文件。下节课来讲。

# 【Git 第9课】 撤销修改

前面说了，如果你手滑删掉了一个文件，可以用 Git 帮你找回来。同样，如果一个文件被你改来改去面目全非，直到程序无法运行，你累感不爱想要回到开始的状态，Git 也可以帮你轻松搞定。

我们现在直接把 readme.txt 从文件夹中删除。看一下状态：

# 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:    readme.txt

#

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

注意其中有一句提示：

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

用 git checkout -- <file> 命令舍弃工作目录中的修改。注意 checkout 后面的 --，没有这两个减号就是另一条命令了，后面关于分支的时候会去说它。

那我们就来试一下：

git checkout -- readme.txt

看看文件夹中，消失的文件是不是又回来了？再看下 git status，也回到了没有产生修改的状态。

那么，如果一个修改后的文件已经被暂存了，要如何恢复到之前的状态呢？

我们来改一下 readme.txt，在文件中加点字，然后 git add 添加到暂存区。查看状态：

# On branch master

# Changes to be committed:

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

#

# modified:   readme.txt

#

这里 Git 又给出提示了：

use "git reset HEAD <file>..." to unstage

用 git reset HEAD <file> 命令撤销暂存。

git reset HEAD readme.txt

这条命令并不会更改 readme.txt 里的内容，修改仍然存在，但是文件的状态变回到已修改。

# 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

#

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

你可以修改了重新暂存，也可以用前面的方法把文件恢复。

再进一步，如果文件修改已经被 commit 了，要如何撤销？

一种情况是，你提交了之后发现还漏了几个地方没有改，或者提交的文件中有些小错误，想要撤销回来重新提交。那么你可以用 git commit --amend 来重新提交。

做完修改、暂存之后，运行

git commit --amend

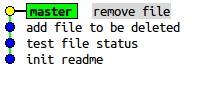
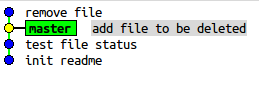
会开启文本编辑器让你修改上次的提交注释，或者通过 -m 参数直接指定。保存退出后，这一次的改动就会被直接加上上一次的提交里，不会产生新的 commit。

你 也许觉得，直接再提交一次不就好了。但难免有时候不想因为笔误产生过多提交。再说，程序员调试代码的时候总会有些恶趣味，诸如 print 'believe spring brother' 之类的调试语句，如果最后忘了删干净就提交了，肯定不想再做一次提交去删除。还是直接神不知鬼不觉地清理掉比较好。

还有另一种情况，就是整个上一次的提交你都不想要了，希望回退到上一个提交状态。这时候就需要用到版本回退了。

|  |
| --- |
|  |

# 【Git 第10课】 版本回退

上次说到，如果你已经提交了代码，但是又后悔了，想要放弃这次的改动，回到之前的版本。这时候就需要做版本回退。  
  
我们先在工作目录中运行 git log 命令，看看到目前为止的提交记录：  
  
commit 81e34d1d269b13bb23619d828495119492eabc5a  
Author: Crossin   
Date:   Thu Nov 28 17:27:34 2013 +0800  
  
    remove file  
  
commit 1edcbf15f45359768b3fa9cd99ddddde792912fe  
Author: Crossin   
Date:   Thu Nov 28 15:16:37 2013 +0800  
  
    add file to be deleted  
  
commit 15547cb5e6bb661a4b9934c1ea55fe0aeceb0682  
Author: Crossin   
Date:   Tue Nov 26 14:39:06 2013 +0800  
  
    test file status  
  
commit c1310cffd37c4b7fe765986035e54a3c27c2887f  
Author: Crossin   
Date:   Tue Nov 26 14:11:46 2013 +0800  
  
    init readme  
  
也许你的提交历史和我不一样，这没关系。  
  
还记得我们前面课里说过的 HEAD，它指向的是最新的提交。而上一次的提交就是 HEAD^，上上次是 HEAD^^，也可以写成 HEAD~2，以此类推。之前30次版本就是 HEAD~30，或者你愿意也可以打30个^。  
  
要回退上一个版本，只要：  
  
git reset --hard HEAD^  
  
--hard 表示放弃所有的本地改动，其他的可选模式我们先不管。  
  
输出提示：  
HEAD is now at 1edcbf1 add file to be deleted  
  
HEAD 已经被移动回“add file to be deleted”这个版本。  
  
如果想要移动到某个指定的提交，也可以直接使用提交id，就是 git log 里，commit 后面的一串字符（你的版本中的提交 id 和我是不一样的）。  
  
git reset --hard 15547  
  
id 不用输全，前几位就够，git 会自动去找匹配的提交。然后就可以看到：  
  
HEAD is now at 15547cb test file status  
  
  
顺便说一下，你可以使用 git 里自带的图形化客户端 gitk 来查看和管理版本变动。在命令行里输入 gitk，mac 上是 gitx，就可以打开（如果不幸没有打开，可能你需要另行安装以下）。  
  
回退前：  
  
回退后：  
  
注意，当用 git reset --hard 回退版本后，所有的本地未提交改动将被舍弃，而被回退掉的版本也不再出现在记录里（虽然还是可以被找回来）。所以在使用此操作时请慎重考虑。

# 【Git 第11课】 远程仓库

之前讲了很多课，都是在本地仓库里进行操作。而 Git 的作用一方面在于对文件进行版本控制，更重要的是便于多人协作开发。因此只有本地仓库的 Git 算不上真正的 Git。而 GitHub 就是创建远程仓库的好地方。  
  
通过 git remote 命令可以查看当前有哪些远程仓库。  
  
还记得我最开始安装成功后从 GitHub 上 clone 下的那个项目吗？在它的工作目录中输入：  
  
git remote  
  
你会看到显示：  
  
origin  
  
这就是这个项目的远程仓库。  
  
加上 -v 可以看到更详细的信息：  
  
origin        [git@github.com](mailto:git@github.com):crossin/CrossinClass.git (fetch)  
origin        [git@github.com](mailto:git@github.com):crossin/CrossinClass.git (push)  
  
克隆的源地址也被显示出来。  
  
当你是从远程仓库 clone 一个仓库到本地时，会自动将此地址加到你的远程仓库列表中，并命名为 origin。  
  
而如果是一个在本地 git init 创建的仓库，刚开始时 git remote 是看不到任何信息的，因为它没有添加任何远程仓库。  
  
想要添加一个远程仓库，可以用命令：  
  
git remote add [shortname]  [比如：  
  
git remote add myremote [url=mailto:git@github.com]git@github.com](http://crossin.me/):crossin/CrossinClass.git  
  
这样，在这个项目中就能 myremote 这个名字来引用添加的远程仓库。  
  
运行一下 git remote，看看是不是已经出现在了列表中。  
  
但此时此刻，你的工作目录下的文件并没有发生任何变化。只是相当于，你拿到了一个箱子的钥匙，而箱子里的东西还原封未动。对于远程仓库的操作，后面还有很多等待我们去折腾的。  
  
现在可以去 GitHub 注册个号，然后建一个自己的项目，clone 到本地。因为别人的项目是没有权限进行写操作的。接下来我们很快就会用到它。  
  
假如你要删除某个仓库，就使用 git remote rm 命令：  
  
git remote rm myremote  
  
就可以把 myremote 这个远程仓库从列表中清除了。

# 【Git 第12课】 抓取/推送数据

当添加了远程仓库之后，肯定不能只在列表中看到个名字就完事了。我们要通过远程仓库来存放数据。  
  
抓取数据的命令是：  
  
git fetch [remote-name]  
  
比如对于我们上次添加的 myremote 远程仓库，就是：  
  
git fetch myremote  
  
不过执行完此命令，你仍然看不到工作目录下的文件有任何变化。这是因为 git fetch 只是把远程的数据抓取到本地，而不会把改动合并到当前的分支上。  
  
如果你是直接从远程仓库 clone 了本地的仓库，那么你可以直接执行命令：  
  
git pull  
  
这样会把远程仓库抓取到本地，再将远端的 master 分支合并到本地的 master 分支上。所以当你之前 clone 下了代码，而其他开发者又更新了远端的代码后，可以通过 git pull 将本地更新到最新的版本。  
  
有关于分支的具体细节，留待后续课程中细说。  
  
  
当你在本地做了改动，并且 commit 了你的改动之后，需要将本地的数据更新到远程仓库中时，就需要用到命令：  
  
git push [remote-name] [branch-name]  
  
假设是要推送到默认的 origin 仓库中的 master 分支，那就运行：  
  
git push origin master  
  
注意，这要求你有对远程服务器上的写权限。另外，如果别人在你之前对远程仓库中提交了新的更新，你的推送就会被驳回。这时你需要先抓取他们的更新，合并到本地，再重新推送。