# Git Learning



## Git本地基本

制定仓库名字

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改

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

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

注意：git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。在那个项目目录下运行没有 --global 选项的命令来配置。

检查配置信息

如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置。

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如：/etc/gitconfig 与~/.gitconfig）。 这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

你可以通过输入 git config <key>： 来检查 Git 的某一项配置

$ git config user.name

Jizx

建立仓库

1. 在现有目录中初始化仓库

通过git init命令把这个目录变成Git可以管理的仓库。

（如果你是在一个已经存在文件的文件夹（而不是空文件夹）中初始化 Git 仓库来进行版本控制的话，你应该开始跟踪这些文件并提交。 你可通过 git add 命令来实现对指定文件的跟踪，然后执行 git commit 提交）

1. 克隆现有的仓库

克隆仓库的命令格式是 git clone [url] 本地仓库名字 。可以不定义本地仓库名字，默认适应原仓库名字。比如，要克隆 Git 的可链接库 libgit2，可以用下面的命令：

$ git clone https://github.com/libgit2/libgit2 mylibgit2

这会在当前目录下创建一个名为 “mylibgit2” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。

提交文件

**第一步，**用命令git add告诉Git，把文件添加到暂存区：

$ git add readme.txt

add一个文件之后，如果又对文件进行了修改，需要重新add，不然commit只会保存最后一次git add的文件内容。

**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"

或者git commit，将会打开默认的文本编辑器（vim）进行文字输入。若实在不习惯 Vim，也可以设置为其它编辑器：

$ git config --global core.editor "notepad"

其中 notepad 可以替换为更好用的 wordpad、notepad++ 等（不过它们在命令行里无法直接访问，得先设置 PATH 变量）。

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."

**快速提交：**

Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，git commit -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤。

重新提交

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令尝试重新提交，命令执行后将暂存区中的文件提交，因此分为以下2种情况：

1. 修改提交信息：

$ git commit --amend

1. 添加忘记/漏add的文件，同时修改提交信息：

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit –amend

查看文件状态

要查看哪些文件处于什么状态，可以用 git status 命令。

使用 git status -s 命令或 git status --short 命令，你将得到一种更为紧凑的格式输出。

**$** git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

??标记：新添加的未跟踪文件

A 标记：新添加到暂存区中的文件

M 标记：修改过的文件，

出现在右边的 M 表示该文件被修改了但是还没放入暂存区

出现在左边的 M 表示该文件被修改了并放入了暂存区。

例如，上面的状态报告显示： README 文件在工作区被修改了但是还没有将修改后的文件放入暂存区,lib/simplegit.rb 文件被修改了并将修改后的文件放入了暂存区。 而 Rakefile 在工作区被修改并提交到暂存区后又在工作区中被修改了，所以在暂存区和工作区都有该文件被修改了的记录。



图 1文件状态周期

Untracked未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件（比如新建的文件）；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”。被追踪后处于3种状态：未修改-已修改-已暂存。将暂存的文件commit后就保存在git数据库中，文件就回到未修改状态咯。

接下来介绍如何对修改的文件进行版本控制。

查看文件具体变化的内容

查看尚未暂存的文件更新了哪些部分，输入 git diff

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容。

查看暂存区里将要添加到下次提交里的内容，可以用 git diff --cached 命令。

也就是查看暂存区里与已保存的文件的区别（Git 1.6.1 及更高版本还允许使用 git diff --staged，效果是相同的，但更好记些。）



应该是这样的关系？

忽略文件（不追踪文件）

在Git工作区的根目录下创建一个特殊的.gitignore文件，（window下直接新建不了，最好用命令行创建）使用命令$ cat .gitignore或者$ touch .gitignore，然后把要忽略的文件名或者目录填进去，Git就会自动忽略这些文件、文件夹。最后一步就是把.gitignore也提交到Git，就完成了！

注意：.gitignore文件只能作用于 **Untracked Files**，也就是那些从来没有被 Git 记录过的文件（自添加以后，从未 add 及 commit 过的文件）。

.gitignore 的格式规范如下：

1. 所有空行或者以 ＃ 开头的行都会被 Git 忽略。
2. 可以使用标准的 glob 模式匹配。
3. 匹配模式可以以（/）开头防止递归。
4. 匹配模式可以以（/）结尾指定目录。
5. 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）。 使用两个星号（\*) 表示匹配任意中间目录，比如a/\*\*/z 可以匹配 a/z, a/b/z 或 a/b/c/z等。

以下是一些例子：

# no .a files

忽略任何.a文件

\*.a

# but do track lib.a, even though you're ignoring .a files above 即使有忽略.a文件的规则，也不忽略lib.a

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

只忽略doc文件夹下的.txt,子目录下的不受影响

doc/\*.txt

# ignore all .pdf files in the doc/ directory

忽略doc文件夹下所有.pdf

doc/\*\*/\*.pdf

有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

$ git add App.class

The following paths are ignored by one of your .gitignore files:

App.class

Use -f if you really want to add them.

**强制添加：**

如果你确实想添加该文件，可以用-f强制添加到Git：

$ git add -f App.class

当然也可以在.gitignore里使用！排除这个文件。

**检查忽略规则：**

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：

$ git check-ignore -v App.class

.gitignore:3:\*.class App.class

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

以下是一些参考资料：

Create a local .gitignore

If you create a file in your repository named .gitignore, Git uses it to determine which files and directories to ignore, before you make a commit.

A .gitignore file should be committed into your repository, in order to share the ignore rules with any other users that clone the repository.

GitHub maintains an official list of recommended .gitignore files for many popular operating systems, environments, and languages [in the github/gitignore public repository](https://github.com/github/gitignore).

1. In Terminal, navigate to the location of your Git repository.
2. Enter touch .gitignore to create a .gitignore file.

The Octocat has [a Gist containing some good rules](https://gist.github.com/octocat/9257657) to add to this file.

If you already have a file checked in, and you want to ignore it, Git **will not** ignore the file if you add a rule later. In those cases, you must untrack the file first, by running the following command in your terminal:

git rm --cached

Create a global .gitignore

You can also create a global .gitignore file, which is a list of rules for ignoring files in every Git repositories on your computer. For example, you might create the file at ~/.gitignore\_global and add some rules to it.

1. Open Terminal.
2. Run the following command in your terminal:

git config --global core.excludesfile ~/.gitignore\_global

The Octocat has [a Gist containing some good rules](https://gist.github.com/octocat/9257657) to add to this file.

Explicit repository excludes

If you don't want to create a .gitignore file to share with others, you can create rules that **are not committed** with the repository. You can use this technique for locally-generated files that you don't expect other users to generate, such as files created by your editor.

Use your favorite text editor to open the file called .git/info/exclude within the root of your Git repository. Any rule you add here will not be checked in, and will only ignore files for your local repository.

1. In Terminal, navigate to the location of your Git repository.
2. Using your favorite text editor, open the file .git/info/exclude.

**对*.gitignore*文件的作用的理解**

.gitignore文件只能作用于 **Untracked Files**，也就是那些从来没有被 Git 记录过的文件（自添加以后，从未 add 及 commit 过的文件）。之所以你的规则不生效，是因为那些 .log 文件曾经被 Git 记录过，因此 .gitignore 对它们完全无效。这也正是开头那段简短答案所做的事情：

1. 从 Git 的数据库中删除对于该文件的追踪；
2. 把对应的规则写入 .gitignore，让忽略真正生效；
3. 提交＋推送。

只有这样做，所有的团队成员才会保持一致而不会有后遗症，也只有这样做，其他的团队成员根本不需要做额外的工作来维持对一个文件的改变忽略。

最后有一点需要注意的，git rm --cached 删除的是追踪状态，而不是物理文件；如果你真的是彻底不想要了，你也可以直接 rm＋忽略＋提交。

注意： .gitignore只能忽略那些原来没有被track的文件，如果某些文件已经被纳入了版本管理中，则修改.gitignore是无效的。

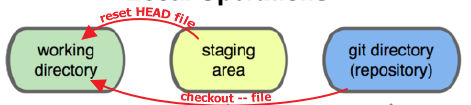
    ？？？有待研究。正确的做法是在每个clone下来的仓库中手动设置不要检查特定文件的更改情况。  
    git update-index --assume-unchanged PATH    在PATH处输入要忽略的文件。  
    另外 git 还提供了另一种 exclude 的方式来做同样的事情，不同的是 .gitignore 这个文件本身会提交到版本库中去。用来保存的是公共的需要排除的文件。而 .git/info/exclude 这里设置的则是你自己本地需要排除的文件。 他不会影响到其他人。也不会提交到版本库中去。  
    .gitignore 还有个有意思的小功能， 一个空的 .gitignore 文件 可以当作是一个 placeholder 。当你需要为项目创建一个空的 log 目录时， 这就变的很有用。 你可以创建一个 log 目录 在里面放置一个空的 .gitignore 文件。这样当你 clone 这个 repo 的时候 git 会自动的创建好一个空的 log 目录了。

GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>

撤销修改

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， 使用命令$ git checkout –– filename **.** git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

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 fileName，就回到了场景1，第二步按场景1操作。



版本回退

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。

$ git reset --hard HEAD^//回退到上一个版本

$ git reset --hard commit id//通过git log 查看commit id，即可回退到你需要的时间点

第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？在Git中，总是有后悔药可以吃的，Git提供了一个命令git reflog用来记录你的每一次命令，这样就可以找到你全部提交过的commit id。

删除文件、取消追踪

* 同时删除库中的文件与工作目录的文件：

使用命令git rm fileName删掉，并且git commit –m “someLog”

* 仅删除库中的文件（取消追踪/恢复未追踪状态）：

换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪,可以输入命令：git rm –cached filename/dir

另一种情况是在文件夹里误删了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：$ git checkout -- test.txt

文件重命名

要在 Git 中对文件改名，可以这么做：

$ git mv README.md README

其实，运行 git mv 就相当于运行了下面三条命令：

**$** mv README.md README

**$** git rm README.md

**$** git add README

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。 不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

版本历史

git log

//显示从最近到最远的提交日志

$ git log -p -2

//一个常用的选项是 -p，用来显示每次提交的内容差异。你也可以加上 -2 来仅显示最近两次提交

$ git log --decorate命令查看各个分支当前所指的对象。

$ git log --pretty=oneline

//另外一个常用的选项是 --pretty。 这个选项可以指定使用不同于默认格式的方式展示提交历史。 这个选项有一些内建的子选项供你使用。 比如用 oneline 将每个提交放在一行显示，查看的提交数很大时非常有用。 另外还有 short，full 和 fuller可以用（不过不怎么实用）。

$ git log --pretty=format:"%h - %an, %ar : %s"

//最有意思的是 format，可以定制要显示的记录格式

|  |  |
| --- | --- |
| Table 2-1. git log --pretty=format 常用的选项 | |
| 选项 | **说明** |
| %H | 提交对象（commit）的完整哈希字串 |
| %h | 提交对象的简短哈希字串 |
| %T | 树对象（tree）的完整哈希字串 |
| %t | 树对象的简短哈希字串 |
| %P | 父对象（parent）的完整哈希字串 |
| %p | 父对象的简短哈希字串 |
| %an | 作者（author）的名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 --date= 选项定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者(committer)的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期，按多久以前的方式显示 |
| %s | 提交说明 |

作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人

$ git log –graph

$ git log --graph --pretty=oneline --abbrev-commit//简洁视图版本

//当 oneline 或 format 与另一个 log 选项 --graph 结合使用时尤其有用。 这个选项添加了一些ASCII字符串来形象地展示你的分支、合并历史

$ git reflog//查看全部命令记录，以及HEAD指针

|  |  |
| --- | --- |
| Table 2-2. git log 的常用选项，可以一起使用 | |
| 选项 | **说明** |
| -p | 按补丁格式显示每个更新之间的差异。 |
| --stat | 显示每次更新的文件修改统计信息。 |
| --shortstat | 只显示 --stat 中最后的行数修改添加移除统计。 |
| --name-only | 仅在提交信息后显示已修改的文件清单。 |
| --name-status | 显示新增、修改、删除的文件清单。 |
| --abbrev-commit | 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。 |
| --relative-date | 使用较短的相对时间显示（比如，“2 weeks ago”）。 |
| --graph | 显示 ASCII 图形表示的分支合并历史。 |
| --pretty | 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。 |

|  |  |
| --- | --- |
| Table 2-3. 限制 git log 输出的选项 | |
| 选项 | **说明** |
| -(n) | 仅显示最近的 n 条提交 |
| --since, --after | 仅显示指定时间之后的提交。 |
| --until, --before | 仅显示指定时间之前的提交。 |
| --author | 仅显示指定作者相关的提交。 |
| --committer | 仅显示指定提交者相关的提交。 |
| --grep | 仅显示含指定关键字的提交 |
| -S | 仅显示添加或移除了某个关键字的提交 |

例子：

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" --before="2008-11-01"

## Git远程

配置ssh

1. 设置Git的user name和email：

$ git config --global user.name "jizx"

$ git config --global user.email [1822980003@qq..com](mailto:1822980003@qq..com)

1. 生成SSH密钥过程：
   1. 查看是否已经有了ssh密钥：cd ~/.ssh

如果没有密钥则不会有此文件夹，有则备份删除

* 1. 生存密钥：

$ ssh-keygen -t rsa -C “haiyan.xu.vip@gmail.com”

按3个回车，密码为空。

得到两个文件：id\_rsa和id\_rsa.pub，在(C:\Users\Administrator\.ssh)目录下

* 1. 添加密钥到ssh：ssh-add 文件名，需要之前输入密码（如果没有就跳过这一步）。
  2. 在github上添加ssh密钥，这要添加的是“id\_rsa.pub”里面的公钥。打开https://github.com/ ，登陆jizxgit，然后添加ssh，将id\_rsa.pub里的字符串全部复制粘贴到给github中
  3. 测试：ssh git@github.com

The authenticity of host ‘github.com (207.97.227.239)’ can’t be established.

RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added ‘github.com,207.97.227.239′ (RSA) to the list of known hosts.

ERROR: Hi tekkub! You’ve successfully authenticated, but GitHub does not provide shell access

Connection to github.com closed.

1. 开始使用github

1.获取源码：

$ git clone git@github.com:billyanyteen/github-services.git

2.这样你的机器上就有一个repo了。

3.git于svn所不同的是git是分布式的，没有服务器概念。所有的人的机器上都有一个repo，每次提交都是给自己机器的repo

仓库初始化：

git init

生成快照并存入项目索引：

git add

文件,还有git rm,git mv等等…

项目索引提交：

git commit

4.协作编程：

将本地repo于远程的origin的repo合并，

推送本地更新到远程：

git push origin master

更新远程更新到本地：

git pull origin master

补充：

添加远端repo：

$ git remote add upstream git://github.com/pjhyett/github-services.git

重命名远端repo：

$ git://github.com/pjhyett/github-services.git为“upstream”

从远程仓库克隆到本地

在某个目录下，打开命令行，运行以下命令，就可以把远程的仓库克隆到**当前目录**

//通用命令

$ git clone username@host:/path/to/repository

//如果远程服务器是github

$ git clone git@github.com:你的GitHub用户名/你的某一个仓库名.git

.git可以不用写

默认远程仓库别名未origin，如果要自定义，可以在克隆时运行命令：  
 git clone -o jizx git@github.com:你的GitHub用户名/你的某一个仓库名.git，那么你默认的远程分支名字将会是 jizx/master

**注意：**克隆后本地只有master分支，如果想要把别的分支如dev克隆下来，需要使用:

$ git checkout -b dev origin/dev

也就是说，当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：$ git checkout -b dev origin/dev

git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支（或不管是什么名字的默认分支）

从本地关联到远程仓库

新建一个文件夹，进入文件夹后使用git init命令，然后再使用git remote add 命令关联一个远程仓库，再使用git pull 命令就可以把管理仓库的所有文件复制到本文件夹中，同样可以修改文件并完成push。

//通用命令

$ git remote add remoteName git@server-name:path/仓库名.git

//github服务器命令

$ git remote add remoteName git@github.com:你的用户名/你的某一个仓库名.git

或者在一个文件夹下直接使用git clone命令，会把远程仓库整个复制到这个文件夹下，然后进入这个文件夹就可以修改文件并完成push。 这时默认远程的url别名为origin

查看远程库信息

用git remote

可以查看到默认的远程库：origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:jizxgit/learngit.git (fetch)

origin git@github.com:jizxgit/learngit.git (push)

remote详细用法

git remote [-v | --verbose]

git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>

git remote rename <old> <new>

git remote remove <name>

git remote rm <name>

git remote set-head <name> (-a | --auto | -d | --delete | <branch>)

git remote set-branches [--add] <name> <branch>…​

git remote get-url [--push] [--all] <name>

git remote set-url [--push] <name> <newurl> [<oldurl>]

git remote set-url --add [--push] <name> <newurl>

git remote set-url --delete [--push] <name> <url>

git remote [-v | --verbose] show [-n] <name>…​

git remote prune [-n | --dry-run] <name>…​

git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…​]

从本地推送到远程仓库

//第一次更新时

$ git push -u origin master

origin：远程服务器url别名 master：本地仓库的一个分支

//以后更新时

$ git push origin master

//如果要推送本地其他分支，比如dev，就改成：

$ git push origin dev

以上其实做了一定的简化，完整的命令是：$ git push origin 本地分支：远程分支  
如果并不想让远程仓库上的分支叫做 serverfix，可以运行 git push origin serverfix:awesomebranch 来将本地的 serverfix 分支推送到远程仓库上的 awesomebranch 分支。

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

* 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。

**冲突、推送失败：**

当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。 你必须先将他们的工作拉取pull下来并将其合并进你的工作后才能推送。

从远程仓库获取新数据

$ git fetch [remote-name]

这个命令会访问远程仓库，从中拉取所有你还没有的数据。 执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看，它并不会自动合并或修改你当前的工作。

$ git pull [remote-name]

如果你有一个分支设置为跟踪一个远程分支，可以使用 git pull 命令来自动的抓取然后合并远程分支到当前分支。git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支（或不管是什么名字的默认分支）。 运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

删除远程分支

假设你已经通过远程分支做完所有的工作了，也就是说你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 serverfix 分支，运行下面的命令：

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

## Git本地高级

分支操作

//创建新的分支，切换到dev分支

$ git branch dev

$ git checkout dev

等价于$ git checkout -b dev //创建并切换到新的分支dev

//复制远程分支到本地

$ git checkout -b dev origin/dev

dev是本地新建的分支名字，origin/dev是远程分支

//查看所有分支，以及当前所在分支，\*表示当前分支

$ git branch

//切换分支

$ git checkout dev

切换出去前，要保存好编辑过的文件，用add commit，或者保存现场

//保存现场

$ git stash

当在一个分支的开发工作未完成，却又要切换到另外一个分支进行开发的时候，并不是不想提交，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。除了commit原分支的代码改动的方法外，我觉得保存现场是一个更加便捷的选择，暂时冻结开发现场，等待其他分支完成后继续回来完成。

//查看现场

git stash list

//恢复现场

git stash apply，默认恢复最新的现场，但是恢复后，stash内容并不删除，你需要用git stash drop来删除

git stash pop，恢复的同时把stash内容也删了

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}

//删除合并后的分支

$ git branch -d dev

//丢弃一个没有被合并过的分支

git branch -D dev强行删除。

分支详解

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录的校验和，然后在 Git 仓库中这些校验和被保存为树对象。 随后，Git 便会创建一个提交对象（98ca9），它除了包含上面提到的那些信息外，还包含指向这个树对象（92ec2）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：一个提交对象（包含着指向前述树对象的指针和所有提交信息） 、一个树对象（记录着目录结构和 blob 对象索引）以及三个 blob 对象（保存着文件快照）。



图 commit操作解释

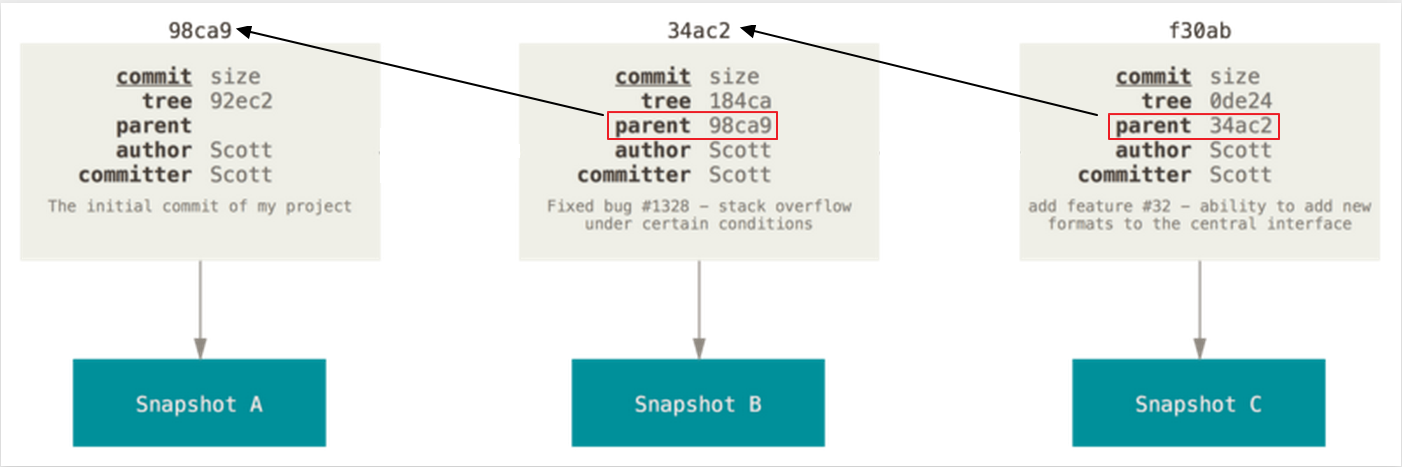


图 每个commit对象的父指针指向前一个commit对象

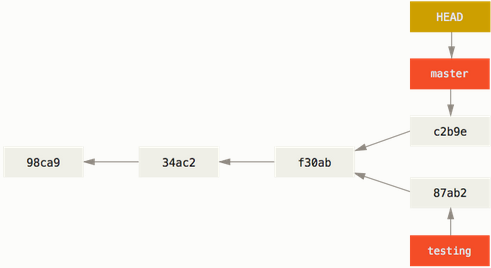
**分支创建：**

Git 的分支，其实本质上仅仅是指向提交对象的可变指针，Git 是怎么创建新分支的呢？ 很简单，它只是为你创建了一个可以移动的新的指针。

Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 HEAD 的特殊指针，它是一个指针，指向当前所在的本地分支，通过checkout切换分支。

经过修改testing分支上的文件，commit后新增了87ab2这个提交对象，这时切换回master分支，在master上修改，commit后又增加了c2b9e这个提交对象，之后就是相关的合并操作了。如下图：



git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph --all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况

分支合并

**快速合并**

$ git merge dev



//关闭快速合并，这样在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。

$ git merge --no-ff -m "merge with no-ff" dev



**合并冲突**

有时候合并操作不会如此顺利。 如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。此时 Git 做了部分合并工作，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。

master分支和feature1分支各自都分别对同一部分修改并有新的提交，就会产生冲突



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，可以直接查看readme.txt的内容：

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。



为了冲突解决，你可以选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容（可以是完全新的内容），并且 <<<<<<< , ======= , 和 >>>>>>> 这些行要被完全删除了。

在你解决了所有文件里的冲突之后，对每个文件使用 git add 命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决，然后commit保存入仓库。



如果你想使用图形化工具来解决冲突，你可以运行 git mergetool，该命令会为你启动一个合适的可视化合并工具，并带领你一步一步解决这些冲突：

$ git mergetool

//查看已合并的分支，从而删除没用的分支

$ git branch --merged

iss53

\* master

//查看未合并的分支

**$** git branch --no-merged

testing

**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>

跟踪分支

从一个远程跟踪分支检出一个本地分支，这个本地分支就叫做 “跟踪分支”（有时候也叫做 “上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

* 创建跟踪分支：git checkout -b [branch] [remotename]/[branch]  
  例如：

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

* **设置已有的本地分支**跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。

$ git branch -u origin/serverfix

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

可以使用 git branch -vv 。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

例如：

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets

master 1ae2a45 [origin/master] deploying index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing 5ea463a trying something new

可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 3落后 1，意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。  
这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：  
 git fetch --all; git branch –vv

标签

Git 可以给历史中的某一个提交打上标签，以示重要。

Git 使用两种主要类型的标签：**轻量标签**（lightweight）与**附注标签**（annotated）。

* 轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。
* 附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；通常建议创建附注标签，这样你可以拥有以上所有信息

标签创建

$ git tag//查看标签，**注意**，标签不是按时间顺序列出，而是按字母排序的。

$ git tag v1.0 //标签默认是打在最新提交的commit上的,标签名为v1.0

$ git tag -a v1.0 -m "version 1.0 released" commit\_id//可以创建带有说明的附注标签，用-a指定标签名，-m指定说明文字。

补打标签：

$ git tag v1.0 commit\_id//你也可以对过去的提交打标签，指定某个历史打标签，通过$ git log --pretty=oneline --abbrev-commit查找到对应要打标签的commit\_id。

$ git show <tagname>可以看到说明文字。

标签推送

默认情况下，git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上

$ git push origin <tagname>//推送某个标签到远程。

$ git push origin –-tags//一次性推送全部尚未推送到远程的本地标签.

标签删除

$ git tag -d v1.0//如果标签打错了，也可以删除。

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$ git tag -d v1.0  
Deleted tag 'v1.0' (was 6224937)

然后，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v1.0//可以删除一个远程标签。

标签切换

如果你想要工作目录与仓库中特定的标签版本完全一样，可以使用 git checkout -b [branchname] [tagname] 在特定的标签上创建一个新分支：

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

当然，如果在这之后又进行了一次提交，version2 分支会因为改动向前移动了，那么 version2 分支就会和 v2.0.0 标签稍微有些不同，这时就应该当心了。

使用GitHub

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页<https://github.com/twbs/bootstrap>，点“Fork”，将别人的仓库在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：git‍ clone ‍git@github.com:jizxGit/bootstrap.git

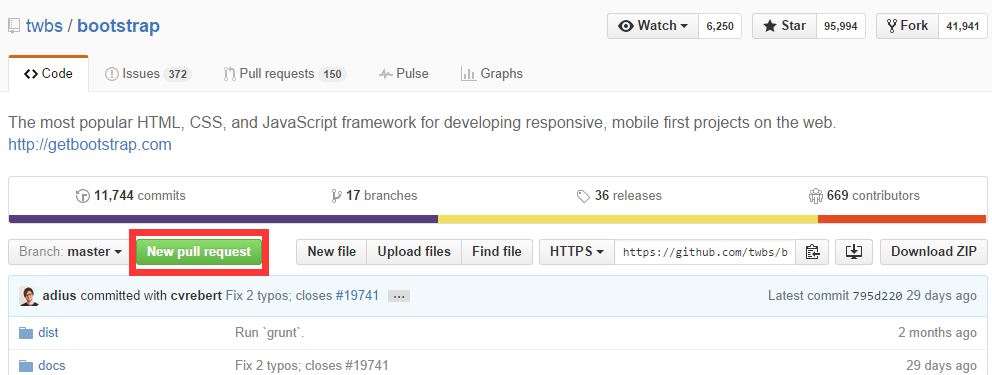
一定要从自己的账号下clone仓库，这样你才能推送修改，因为你在github添加了SSH。如果从bootstrap的作者的仓库地址git@github.com:twbs/bootstrap.git克隆，因为没有权限，你将不能推送修改。

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。



大文件更新Update

我们知道 Git 不仅仅是用来做代码版本管理的，很多其他领域的项目也会使用 Git。比如说我公司曾经一个客户的项目涉及到精密零件图纸文档的版本管理，他们也用 Git。有一种使用场景是**对一些体积庞大的文件进行修改**，但是每一次保存 Git 都要计算文件的变化并更新工作区，这在硬盘慢的时候延迟卡顿非常明显。

git update-index --assume-unchanged 的真正用法是这样的：

1. 你正在修改一个巨大的文件，你先对其 git update-index --assume-unchanged，这样 Git 暂时不会理睬你对文件做的修改；
2. 当你的工作告一段落决定可以提交的时候，重置改标识：git update-index --no-assume-unchanged，于是 Git 只需要做一次更新，这是完全可以接受的了；
3. 提交＋推送。

配置命令别名

* 配置别名有点像C语言里的宏，进行简单的替换。
* 替换单个词，告诉Git，以后st就表示status

$ git config --global alias.st status // alias：别名的意思

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

$ git config --global alias.co checkout

$ git config --global alias.ci commit

$ git config --global alias.br branch

* 替换多个词，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：  
  $ git config --global alias.unstage 'reset HEAD'
* 撤销别名，见配置文件

配置文件

Git的时候，加上--global是针对**当前用户（整个软件）**起作用的，如果不加，那只针对**当前的仓库（当前文件夹）**起作用。

* **当前仓库配置文件：**每个仓库的Git配置文件都放在.git/config文件中：

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ignorecase = true

precomposeunicode = true

[remote "origin"]

url = git@github.com:michaelliao/learngit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[alias]

last = log -1

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

* **全局配置文件**：放在用户主目录（window系统中的用户目录下）下的一个隐藏文件.gitconfig中：

$ cat .gitconfig

[alias]

co = checkout

ci = commit

br = branch

st = status

[user]

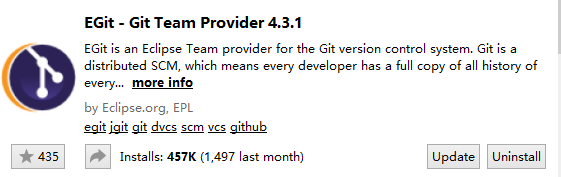
name = Your Name

email = your@email.com

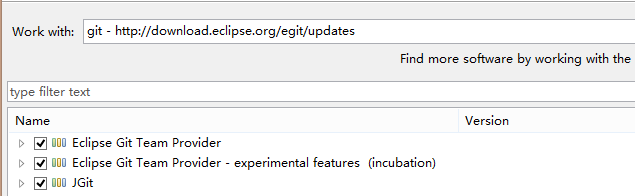
配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

eclipse使用git

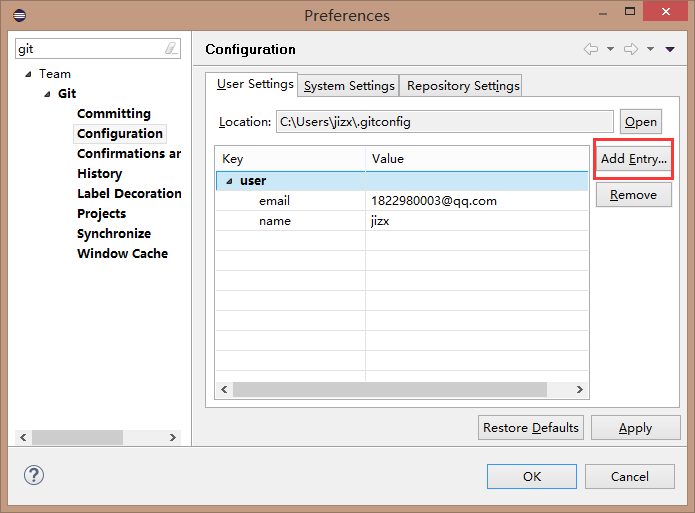
1. **首先需要有git插件**，一般eclipse自带git功能，如果没有，安装方法如下：

方法1：打开应用商店在help->marketplace中，搜egit，按照步骤安装

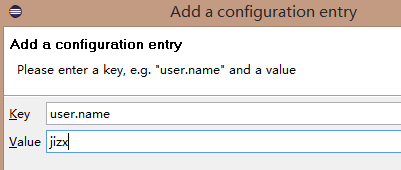
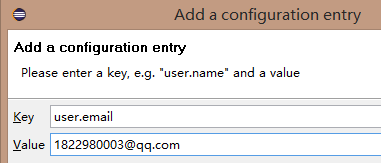
方法2：help->Install New Software中，输入git - <http://download.eclipse.org/egit/updates>勾选全部进行安装：



1. **配置本地用户信息**，在preference中搜索git，找到配置选项，如下图：



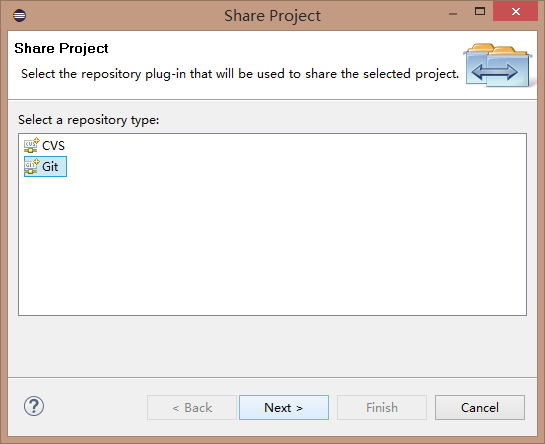
点击Add Entry，添加用户信息与邮箱，如下图：

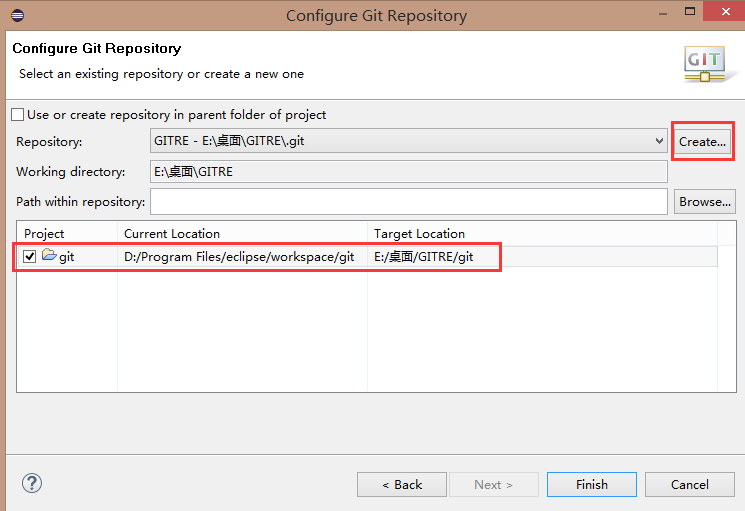
1. **创建本地仓库**

**方法1：**在java视图中，选中我们要使用git的项目，然后右键项目->Team->Share Project

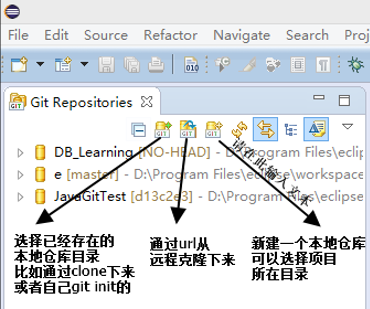
选择Git



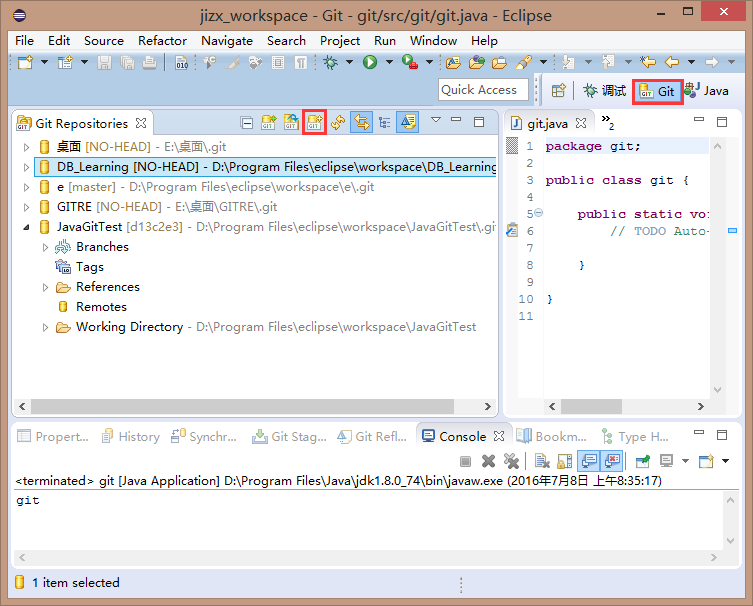
此时将打开Configure Git Repository对话框，点击create，选择目录来创建本地仓库，即保存项目的位置，然后完成。



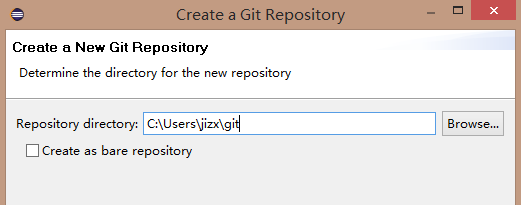
**方法2：**在git视图，如下图，可以看到如下3个图标，可以选择已经存在的仓库、从远程克隆下来、新建仓库。

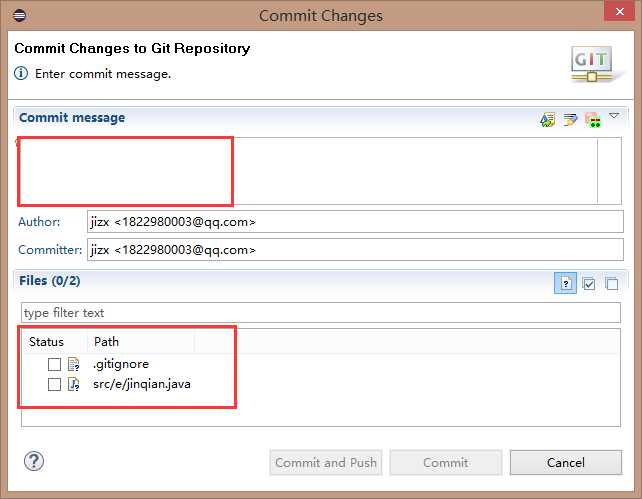


* **新建仓库步骤**：



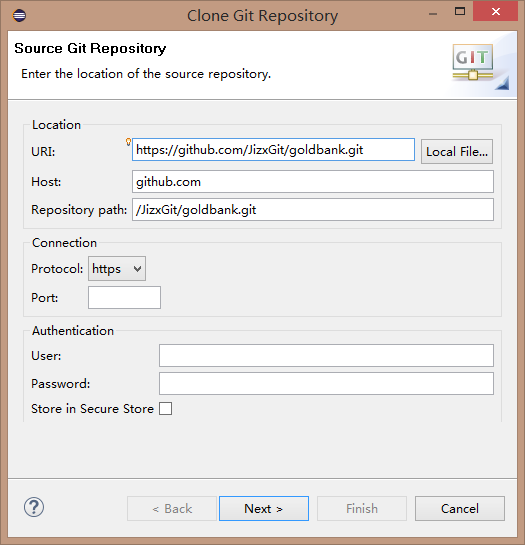
选择仓库保存的路径（可以直接选项项目的目录）



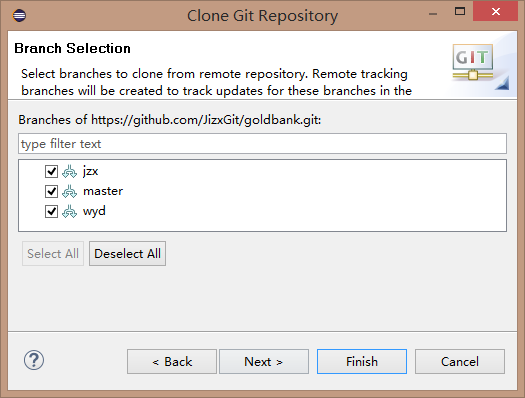
添加完成后即可在git repository出现最后修改代码后即可commit（shift+ctrl+3），选需要commit的文件，以及commit信息。如下图：

* **克隆github仓库**

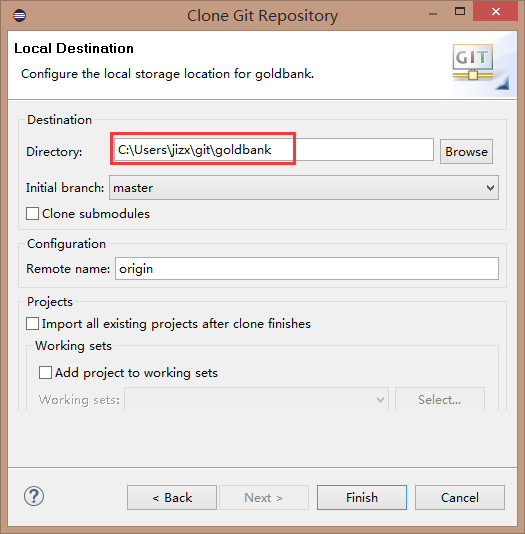
只需输入github上的URL即可，其他的会自动生成：



选择分支需要的分支：

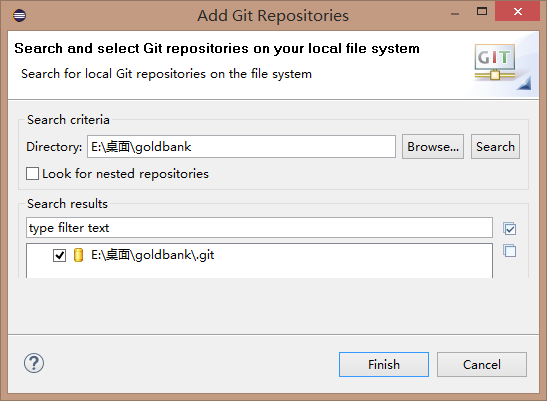


选择保存仓库的目录，初始分支，远程url的别名（默认为origin）等信息，就完成了



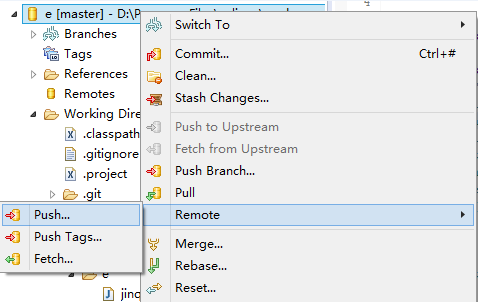
* **本地已存在的仓库**

选择本地仓库目录，如果目录中存在仓库，可自动发现，添加即可。

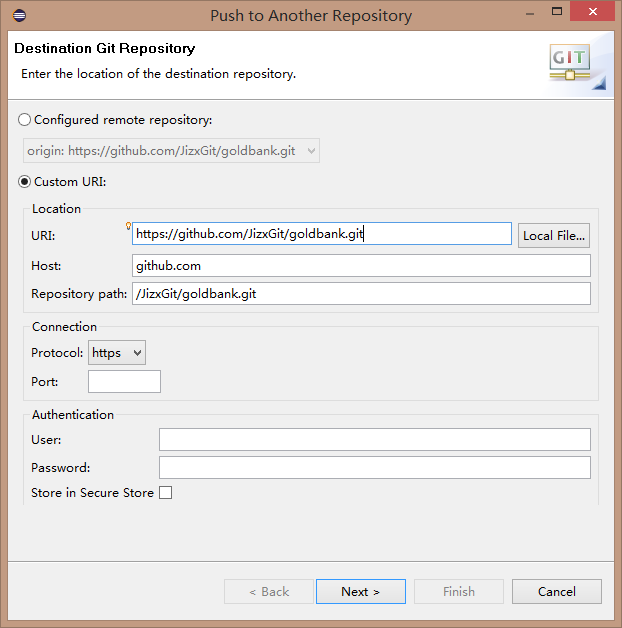


1. **本地仓库关联远程仓库**

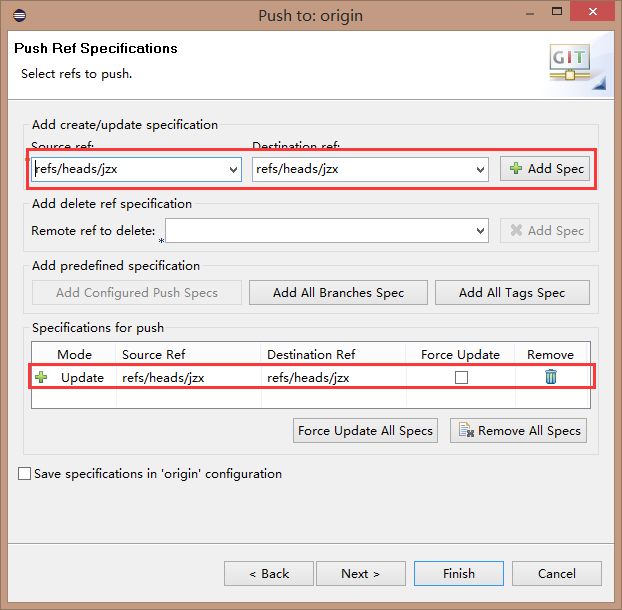
在Commit一次之后（不然下面操作会进行不了），选择项目，右键->team->remote->push（java视图）/右键->remote->push(Git视图) ，进行配置远程仓库url。



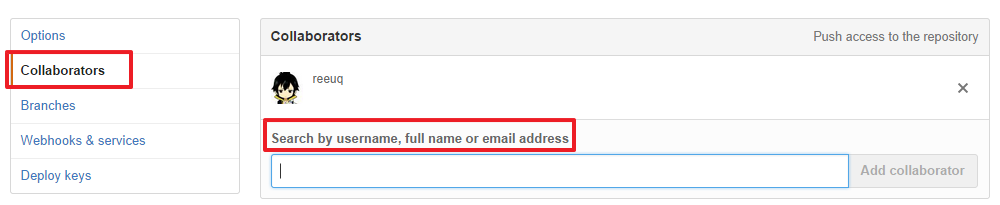
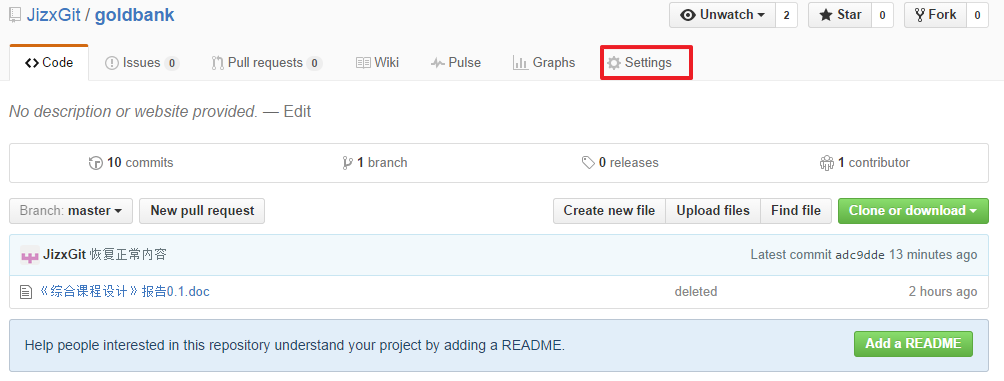
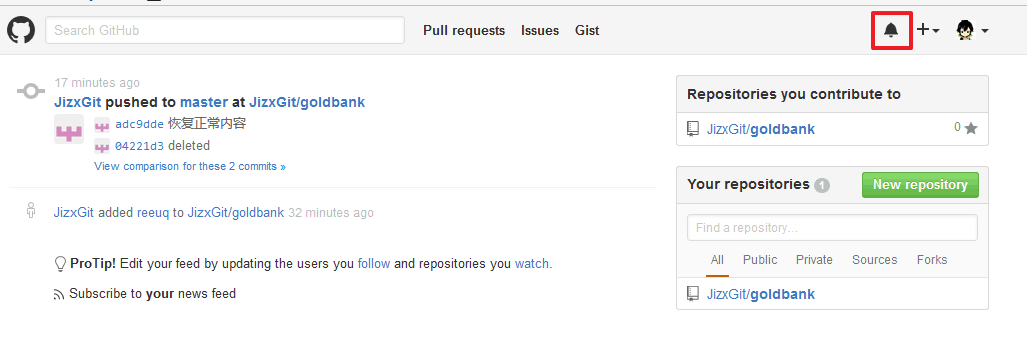
输入[https://github.com/用户名/仓库名.git](https://github.com/用户名/仓库名.git%20/) ，用户名，密码即可。



选择本地分支与远程分支，点击Add Spec，即可生成一个push方案，也就是把本地某个分支如何push到远程某个分支。



多人合作

1. 首先在github上创建一个需要与团队共享的仓库
2. 然后在仓库的Settings中添加队友的github帐户：
3. 添加完成后，对方会收到请求，如下图，需要对方同意请求，才算是加入到这个团队中，拥有push的权限。

**流程会像这样**：

A拥有repository，想要让B也能更新自己的repository，就把B加入collaborators，这样B要一起合做这份code的时候，先按照前一篇的流程设定B初始化的时候，注意clone URL 是 A 的 repo 的 clone URL，然后B就可以在自己的本机尽情修改code (用branch、commit、merge)最后在push的时候输入B自己的帐密，会更新A的repository(A仍就可以继续更新自己的repository)但B自己的github账户并不会有一份和A相同的repository(只会有A的repo连结)

## Git快捷键

全屏

ATL+ENTER

字体变化

CTRL+PLUS/MINUS/ZERO

右键

ALT+SPACE