
版本控制系统入门书

Git 指南

最简洁的 Git 教程

李新星

2015-2-26

序

在学习中接触了 Git 这个版本控制工具，刚开始并不知道怎么利用这个工具帮助我来开发（虽然我知道它很强大），也是利用网上的一些资料入门，由于没有完全掌握这个版本控制工具，我把它只是用来当做备份代码的一个工具而已，当时并没有体会到利用 Git 帮助开发带来的便利。

后来在学校的图书馆偶然看到了一本介绍 Git 的书籍《Git 权威指南》，发现这本书写的非常不错，全面细致的介绍了 Git 使用的方法以及技巧。我在放寒假回家时借了这本书，打算利用一寒假的时间来系统的学习 Git。

放假回家学习 Git 非常顺利，我花了大约 2 周的时间便学完了 Git，多亏了这本书，讲解的非常全面，而且每个命令都有讲解，学习起来非常快。

我在学习的时候选择的操作系统是 Ubuntu，在上面安装 Git 非常方便，而且学习起来非常轻松。

学习完 Git 后我的最强力的感受是：我为什么没有早点学习它。Git 非常强大，我现在在开发中已经完全离不开 Git 了，Git 给开发带来了极大的便利，不仅可以很方便的解决版本控制的问题，也可以解决团队的协同开发问题，是开发的一个绝佳帮手。

由于 Git 如此多的好处，我推荐每个还没有系统学习 Git 的开发者都去学习下 Git，它会让开发变得简单。

说了这么多了，还没有说到写此书的意图啦。好，下面我就说说我写下此书的目的。

写此书的目的

Git 命令较多，可能很多初学者会记不住。我也是这样的，尽管已经系统的学习了上 Git，但是如果过个两三个月不使用 Git 的话，我也会记不清一些命令。

那么，写此书的目的来了，为了在我记不清一些命令时，我可以查阅这本书，便可以重温 Git 的命令，解决忘记的问题。由于单独写成此书，相比书本来说，更简单，在这本书中省去了一些关于 Git 命令的详细介绍，只保留必要的部分。这样使得此书的页数大大减少，省去了翻阅书籍的大量时间，提高了工作效率。同时，书中关于 Git 命令都有着详细的演示，部分演示配有截图，可以更好地阅读。

除了方便已经有 Git 基础的人员查阅外，这本书还适合没有基础，但想快速上手 Git 的人员阅读，这本书相比《Git 权威指南》、《Pro Git》来说，保留了所有 Git 的基础命令，并有适当的讲解，可以更快的学习 Git，利用 Git 开发。但是不推荐这么做，想要真正的利用 Git 这个强大的版本控制工具，还是去学习更详细更权威一点的书籍吧，但如果你喜欢看这本书，我也非常欢迎。

本书的组织

本书共有六篇。

第一篇主要介绍了版本控制系统的前世今生，详细讲解了 Git、SVN、CVS 之间的区别，同时介绍了 Git 的一些闪亮特性；最后详细介绍了 Git 在 Linux、windows、OS X 系统下的安装方法，没有安装好 Git 的用户可以详细的阅读并在自己的系统下安装 Git。

第二篇是 Git 入门。

主要介绍了 Git 的相关配置以及创建 Git 仓库、暂存、提交的命令、改变 Git 历史、查看提交日志等命令，在这一章，你会掌握 git 的基本命令，初步入门 git。

第三篇 Git 进阶。

在这一篇中，你将会接触到利用 git 来创建分支、打标记、使用 Git 远程版本库等知识，这些都是 git 的闪亮特性，学会这些，利用 Git 会变得更加有趣。

第四篇是使用 github。

我们原先创建版本库等各种操作都是在本地电脑上操作的，在这一篇中，我们会介绍如何利用 github 等平台备份代码、开发项目。

第五篇是 Git 其他应用。

在这一篇中，我会介绍一些使用 Git 的其他技巧。

第六篇 Git 常用命令。

至此，我们已经全部学完了 Git，在这一篇中，我会总结使用 Git 的全面命令，在开发中可以快速查阅。

适用读者

适用全部想学习 Git 的读者。

排版约定

本书中约定的排版格式如下：

- 命令输出级示例代码

\$ git --version

git version 1.9.1

在提示符**\$**后的字符为输入命令，输入命令为黑色粗体，而在输入命令后的非粗体字符为输出。

在线资源

本文全部发表在我的博客上：

官方博客：<http://blog.lxx1.com/>

<http://blog.lxx1.com/category/git>

本书下载地址: <http://www.lxx1.com/>

Github 下载地址: <https://github.com/xinxingli/git.git>

欢迎大家阅读、下载, 同时有什么疑问也可以同我交流。

李新星 (<http://blog.lxx1.com>)

2015 年 2 月 25 日星期三 20 时 1 分

目录

序	1
写此书的目的.....	2
本书的组织.....	3
适用读者.....	4
排版约定.....	4
在线资源.....	4
第一篇 Git 前言	8
1.1 版本控制系统前言.....	8
1.2 版本控制系统的历史.....	9
1.2.1 CVS——开启版本控制系统的大门.....	9
1.2.2 SVN——集中式版本控制系统	10
1.2.3 GIT——伟大的分布式版本控制系统	11
1.3 在你的电脑上安装 Git.....	13
1.3.1 Git 在 Linux 下安装	13
1.3.2 Git 在 OS X 下安装.....	14
1.3.3 Git 在 windows 下安装.....	14
第二篇 Git 入门.....	16
2.1 配置 Git.....	16
2.2 创建 Git 版本库.....	17
2.3 往 Git 版本库中提交代码	19
2.4 查看提交历史.....	21
2.5 使用 <code>git diff</code> 命令查看修改	23
2.6 使用 <code>git reset</code> 命令改变历史.....	27
第三篇 Git 进阶	35
3.1 远程仓库的使用.....	35
3.1.1 查看当前的远程库.....	35
3.1.2 添加远程仓库.....	37
3.1.3 从远程仓库抓取数据.....	39
3.1.4 推送数据到远程仓库.....	40
3.1.5 查看远程仓库信息.....	40
3.1.6 远程仓库的删除和重命名.....	43
3.2 打标签.....	44
3.2.1 列显已有的标签.....	44
3.2.2 新建标签.....	45
3.2.3 含附注的标签.....	46
3.2.4 签署标签.....	48
3.2.5 轻量级标签.....	49
3.2.7 后期加注标签.....	51
3.2.8 分享标签.....	54

3.3 Git 分支.....	55
3.3.1 何谓分支.....	56
3.3.2 新建分支.....	57
3.3.3 合并分支.....	58
3.3.4 删除分支.....	60
3.3.5 合并冲突.....	60
3.3.6 分支管理.....	62
3.4 远程分支.....	64
3.4.1 推送.....	64
3.4.2 跟踪分支.....	66
3.4.3 删除远程分支.....	67
3.5 小结.....	67
第四篇 使用 Github.....	69
4.1 建立账户.....	70
4.2 建立仓库.....	71
4.3 生成密钥.....	72
第五篇 Git 其他应用.....	76
5.1 使用 etckeeper 备份 Linux 下的/etc 目录。.....	76
第六篇 Git 常用命令.....	77
1.创建仓库.....	77
2.通过 git init 命令把这个目录变成 Git 可以管理的仓库：.....	77
3.用命令 git add 告诉 Git，把文件添加到仓库(实际上就是把文件修改添加到暂存区)：.....	77
4.用命令 git commit 告诉 Git，把文件提交到仓库(实际上就是把暂存区的所有内容提交到当前分支)：.....	77
5.随时掌握工作区的状态.....	77
6.查看文件被修改的内容.....	78
7.查看代码的历史版本号.....	78
8.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.....	78
9.查看命令历史，以便确定要回到未来的哪个版本.....	78
10.弄明白 Git 的工作区(当前分区)和暂存区.....	78
11.理解 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.....	78
12.撤销修改.....	78
13.删除文件.....	79
14.将本地仓库与 github 仓库关联起来.....	80
15.多人协作一个项目的时候，我们每个人可以通过从远程仓库克隆一份来作为己用。.....	80
16.创建分支并且切换到分支.....	80
17.解决冲突.....	81
18.Bug 修复.....	81
19.开发新功能.....	82
20.参与开源项目先要克隆一份到本地.....	82

第一篇 Git 前言

1.1 版本控制系统前言

如果你是一枚程序猿，或者是网站等等的开发人员，如果你正在为怎样提高团队开发效率，更好的解决在开发中遇到的 **BUG**，又或者为怎样管理备份代码而发愁，那么，你该去了解一下版本控制系统了。

什么是版本控制系统呐，举个例子吧，我在开发一个网站项目的时候，首先开发出了版本一，好的，将版本一部署到服务器上，部署好之后请客户验收，客户看过之后，提出了一些修改意见，然后要按照客户的意见修改网站功能，在修改首页的时候，为了防止丢失原来的首页，我将其另存为 `index.htm.backup`，而后在 `index.htm` 的基础上修改，修改完后存为 `index.htm`，后来发现缺少一个功能，我又开始修改首页，将 `index.htm` 存为 `index.htm.2`，开始加入新的功能，完成后保存，在改完后测试发现出现了 **bug**，又返回开始修改 **bug**，将 `index.htm` 存为 `index.htm.3`，又开始修改...，如此往复，最后去看 `index.htm`，发现已经是一团糟，更不记得哪一个版本引入了新功能，哪一个版本新引入了 **bug**。

造成这种现象的原因是在修改代码的时候没有使用版本控

制系统，造成代码重复保存、修改，出现混乱。上面举得只是个人在开发的时候遇到的问题，这仅仅只是其中的一方面，如果团队在项目开发，更要利用版本控制系统来协同开发、管理代码。

1.2 版本控制系统的历史

1.2.1 CVS——开启版本控制系统的大门

CVS (Concurrent Versions System)诞生于 1985 年，是由荷兰阿姆斯特丹 VU 大学的 Dick Grune 教授实现的。当时 Dick Grune 教授和两个学生一起共同开发一个项目，但是三人的工作时间无法协调到一起，迫切需要一个记录和协同开发的软件。于是，Dick Grune 教授通过脚本语言对 RCS（一个针对单独文件的版本管理软件）进行封装，设计出了历史上第一个被大规模使用的版本控制工具。

CVS 采用服务器/客户端架构设计，版本库位于服务器端，实际上就是一个 CVS 容器。每一个 RCS 文件以,v 作为文件名后缀，用于保存对应文件的每一次更改历史。RCS 文件中只保留一个版本的完全拷贝，其他历次更改则差异存储其中，使得存储非常有效率。

CVS 最大的优点就是简单，但他的缺点使得 CVS 不能适应现代的版本控制，其缺点主要有

- 服务器松散存储的文件使得其在建立里程碑及分支时效率不高，并且当服务器端存储的文件越多时，其效率越低。
- 分支和里程碑不可见，因为他们被分散的记录在服务器端的各个 RCS 文件中。
- 合并困难。
- 由于文件在服务器端是单独通过差异存储的，不能优化存储内容相同但文件名不同的文件。
- 不能对文件盒目录的重命名进行版本控制。

CVS 的成功激发了版本控制工具的大爆发，各式各样的版本控制系统如雨后春笋般爆发出来，这其中最典型的的就是 SVN。

1.2.2 SVN——集中式版本控制系统

Subversion, 由于其命令行工具为 `svn`，因此通常被称为 SVN，SVN 由 CollabNet 公司于 2000 年资助并开始开发，目的是创建一个更好用的版本控制系统以取代 CVS，到 2001 年，SVN 已经可以用于自己的版本控制了。

SVN 的每一次提交，都会在服务器端 `db/revs` 和 `db/revprops` 目录下各创建一个以顺序编号命名的文件。其中 `db/revs` 目录下

的文件记录了于上一次提交之间的差异（字母 **A** 表示新增，**M** 表示修改，**D** 表示删除）。在 `db/revprops` 目录下的同名文件则保存着提交日志、作者、提交时间等信息。这样做使得 **SVN** 拥有全局版本号，每提交一次，**SVN** 版本号都会自动加 1，使得使用 **SVN** 有了极大的便利；同时，在提交时文件名不受限制，因为服务器端不需要建立和客户端文件相似的文件名，于是，文件命名不受服务器操作系统字符集和大小写的限制。

SVN 在目录的轻量级拷贝、版本库授权以及冗余的拷贝方面的亮点，使得 **SVN** 成为开源社区和各大公司企业间的新宠。

但是，相比 **CVS**，**SVN** 本质上没有区别，都属于集中式版本控制系统。即一个项目对应一个版本库，所有成员只能通过网络向服务器上的版本库提交，这样除会出现单点故障外，在查看提交日志及提交数据的延迟，也带来了许多不便。

1.2.3 GIT——伟大的分布式版本控制系统

Linux 之父 Linus 是 **CVS** 和 **SVN** 的坚决反对者，在他管理维护 linux 代码时，一直以手工修补文件的方式管理代码，在 2002 年至 2005 年期间，Linus 选择了一款商业版本控制系统 `bitkeeper` 作为 linux 内核的代码管理工具，`bitkeeper` 不同于 **CVS** 和 **SVN**，是一款分布式的版本管理工具。

2005 年的一件事导致了 Git 的诞生，samba 的作者 Andrew Triggell 试图对 BitKeeper 进行反向工程，以开发一个能与 BitKeeper 交互的开源工具，这激怒了 BitKeeper 的所有公司，要求收回 linux 社区免费的 BitKeeper 使用权。迫不得已，Linus 最终决定自己开发一个分布式版本控制工具，于是，Git 诞生了。

Linus 以一个文件系统专家和内核设计者的视角对 Git 进行设计，其独特的设计让 Git 拥有非凡的性能和最为优化的存储能力。经过短短几年的发展，Git 得到广泛的认可，众多的开源项目纷纷迁移到 Git 上来，如 Git 和 linux 的内核外，还有 Perl、GNOME、KDE、Qt、ruby on Rails、Android、Debian，还有 github 上的上百万个项目。

如今，git 可以在 linux、Mac OS、windows 下运行，为每一个代码开发者带来了便利。

1.3 在你的电脑上安装 Git

Git 可以安装在 windows、Linux、OS X 等系统上，现在我在
这里介绍下如何在这三种系统平台下安装部署 git。

1.3.1 Git 在 Linux 下安装

要查看在 Linux 下是否已经安装了 Git，可以直接在你终端输入 git 命令，查看输出，这里是没有安装的输出：

```
git
```

程序“git”尚未安装。 您可以使用以下命令安装：

```
apt-get install git
```

这里我们使用的平台是 Ubuntu，要安装 Git 的命令是：

```
$ sudo apt-get install git -y
```

如果使用的是 centos，则安装的命令式：

```
$ sudo yum install git -y
```

只需要这一条简单的命令就可以安装 Git。

1.3.2 Git 在 OS X 下安装

MAC 下是默认安装 Git 的，只要在终端输入 `git` 命令即可查看是否安装，如果显示这样的话就是安装了：

```
lixinxingdeMac:~ lixinxing$
lixinxingdeMac:~ lixinxing$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
          [--exec-path<path>] [--html-path] [--man-path] [--info-path]
          [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
          [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
          <command> [<args>]

The most commonly used git commands are:
add      Add file contents to the index
bisect   Find by binary search the change that introduced a bug
branch   List, create, or delete branches
checkout Checkout a branch or paths to the working tree
clone    Clone a repository into a new directory
commit   Record changes to the repository
diff     Show changes between commits, commit and working tree, etc
fetch    Download objects and refs from another repository
grep     Print lines matching a pattern
init     Create an empty Git repository or reinitialize an existing one
log      Show commit logs
merge    Join two or more development histories together
mv       Move or rename a file, a directory, or a symlink
pull     Fetch from and integrate with another repository or a local branch
push     Update remote refs along with associated objects
rebase   Forward-port local commits to the updated upstream head
reset    Reset current HEAD to the specified state
rm       Remove files from the working tree and from the index
show     Show various types of objects
status   Show the working tree status
tag      Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
lixinxingdeMac:~ lixinxing$
```

blog.lxx1.com

如果没有安装，那么可以直接从 AppStore 安装 Xcode，Xcode 集成了 Git，不过默认没有安装，你需要运行 Xcode，选择菜单 “Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择 “Command Line Tools”，点“Install”就可以完成安装了。

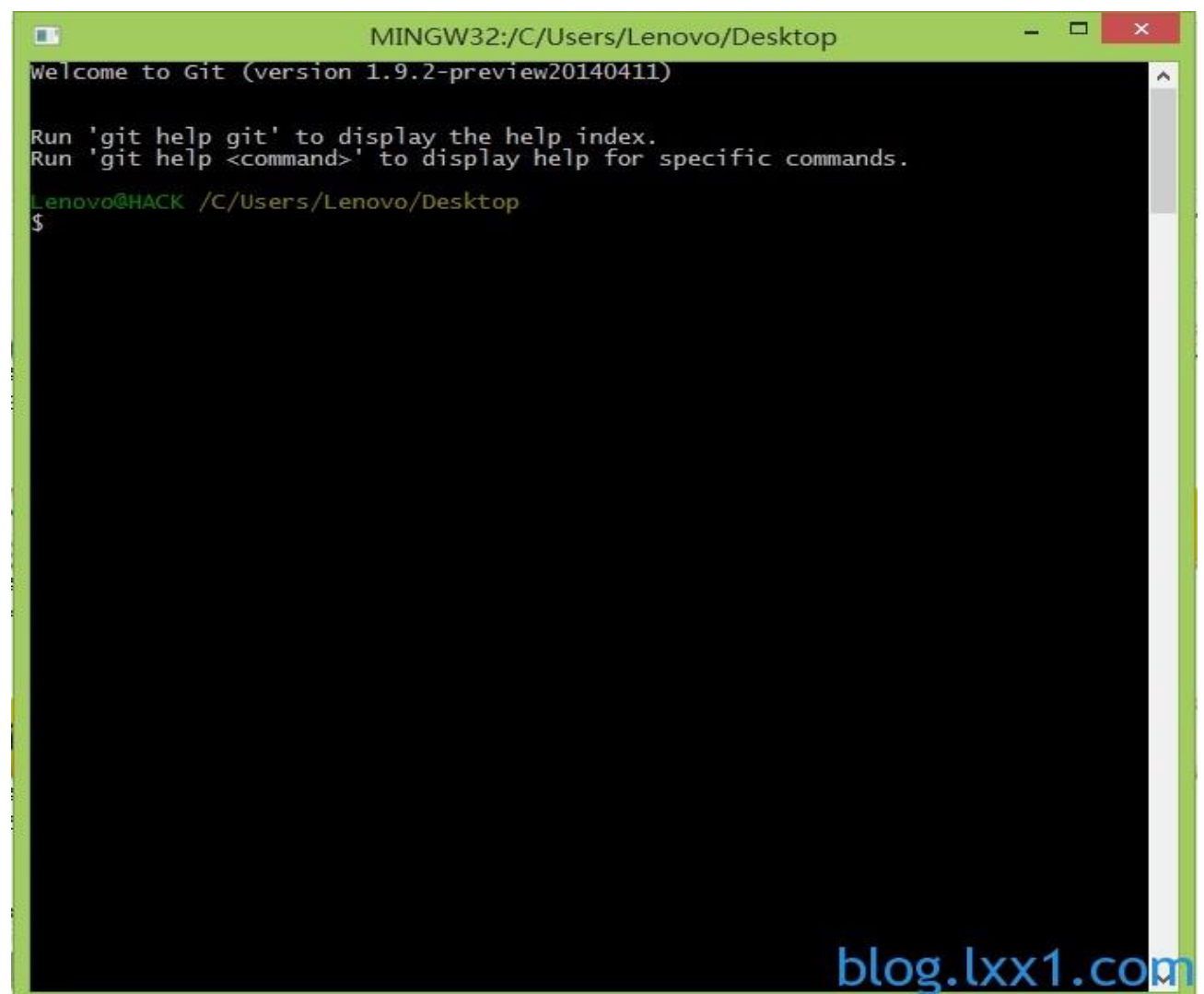
1.3.3 Git 在 windows 下安装

Windows 下要使用很多 Linux/Unix 的工具时，需要 Cygwin 这样的模拟环境，Git 也一样。在 windows 下安装一个名叫 msysgit

的程序，只需要下载一个单独的 exe 安装程序，其他什么也不用装，就可以在 windows 上使用 git。

msysgit 是 Windows 版的 Git，从 <http://msysgit.github.io/> 下载，然后按默认选项安装即可。

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！



```
MINGW32:/C:/Users/Lenovo/Desktop
Welcome to Git (version 1.9.2-preview20140411)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Lenovo@HACK /C:/Users/Lenovo/Desktop
$
```

blog.lxx1.com

第二篇 Git 入门

在上一篇中，我们介绍了版本控制工具的变更，以及相应的优点、缺点，相信看到这里，你已经知道了 Git 这款分布式版本控制系统的魅力，想进一步学习、利用这款最好的版本控制系统，那么在这一章，我们一起来学习 git。

在这一章，你会掌握 git 的基本命令，初步入门 git。

我假设你在电脑上已经将 Git 安装好了，就算没有，我相信你会搞定它的，如果遇到什么困难，可以去附录查看，我会详细的告诉你安装方式。

好的，接下来正式开始学习。

2.1 配置 Git

使用如下操作查看你的电脑上安装的 Git 版本：

```
$ git --version
```

```
git version 1.9.1
```

可以看到在这台电脑上已经成功安装了 git，并且版本是 1.9.1。

在正式使用 Git 前，需要设置一下 Git 的配置变量，配置会永久保存在 Git 的全局文件（/home/.gitconfig）或者系统文件（/etc/gitconfig）中。

- 1) 配置 Git 当前的用户名和邮件地址，这将在你提交版本库时用到。

```
$ git config --global user.name "li xinxing"
```

```
$ git config --global user.email lixinxing@lxx1.com
```

- 2) 配置一些 Git 别名，提高 Git 使用效率。

```
# sudo git config --system alias.st status
```

```
# sudo git config --system alias.ci commit
```

```
# sudo git config --system alias.co checkout
```

```
# sudo git config --system alias.br branch
```

- 3) 在 Git 命令输出中开启颜色显示

```
$ git config --global color.ui true
```

2.2 创建 Git 版本库

Git 的所有操作，都是通过 Git 一个命令完成的，下面演示如

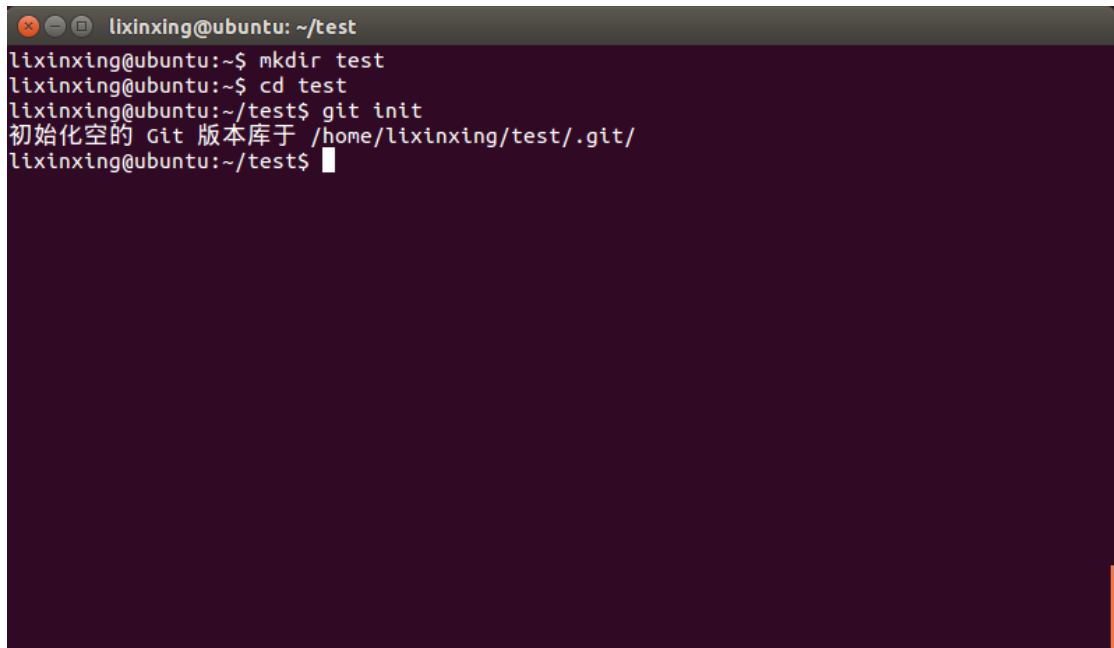
何创建 Git 版本库。

```
$ mkdir test
```

```
$ cd test
```

```
$ git init
```

初始化空的 Git 版本库于 `/home/lixinxing/test/.git/`



```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~$ mkdir test
lixinxing@ubuntu:~$ cd test
lixinxing@ubuntu:~/test$ git init
初始化空的 Git 版本库于 /home/lixinxing/test/.git/
lixinxing@ubuntu:~/test$
```

除在目录下使用 `git init` 命令创建版本库外，还可以在 `git init` 命令后面直接输入目录名称，自动完成目录的初始化。

```
$ git init /home/lixinxing/test2
```

初始化空的 **Git** 版本库于 `/home/lixinxing/test2/.git/`

查看 `git` 目录下所有文件：

```
$ ls -aF
```

```
./ ../ .git/
```

可以看到在 `git` 库下多了一个隐藏目录 **.git**，这个隐藏的 `.git` 目录就是 **Git** 版本库（又叫仓库，`repository`）。

`.git` 版本库所在目录为 `/home/lixinxing/test`，它被称为工作区。

2.3 往 **Git** 版本库中提交代码

下面提交代码到版本库

在工作区中新建文件 `welcome.txt`，内容就是一行“`hello world!`”。

```
$ echo "Hello World!" > welcome.txt
```

将 `welcome.txt` 添加到版本库：

```
$ git add welcome.txt
```

这里添加文件只是将添加到版本库的暂存区，要将其添加到版本库中，还需在执行一次提交命令：`git commit`。

需要注意的是，执行提交命令时需要加参数 `-m` 给出提交说明，不然 `git` 会自动打开编辑器，要求输入提交说明。

```
~/test$ git commit -m "initialized"
```

```
[master (根提交) b11d1c2] initialized
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 welcome.txt
```

在提交成功后，我们可以通过 `git status` 命令显示工作区文件状态

```
$ git status
```

```
位于分支 master
```

```
无文件要提交，干净的工作区
```

```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~/test$ git status
位于分支 master
无文件要提交，干净的工作区
lixinxing@ubuntu:~/test$
```

这样就提示在工作区的文件已经全部提交成功。

2.4 查看提交历史

提交后，可以通过 `git log` 命令查看提交日志（附加的 `-stat` 参数可以看到每次提交的文件变更统计）。

\$ git log

commit b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1

Author: li xinxing <lixinxing@lxx1.com>

Date: Mon Feb 16 11:33:03 2015 +0800

initialized

\$ git log --stat

commit b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1

Author: li xinxing <lixinxing@lxx1.com>

Date: Mon Feb 16 11:33:03 2015 +0800

initialized

welcome.txt | 1 +

1 file changed, 1 insertion(+)

```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~/test$ git log
commit b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1
Author: li xinxing <lixinxing@lxx1.com>
Date:   Mon Feb 16 11:33:03 2015 +0800

    initialized
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git log --stat
commit b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1
Author: li xinxing <lixinxing@lxx1.com>
Date:   Mon Feb 16 11:33:03 2015 +0800

    initialized

    welcome.txt | 1 +
    1 file changed, 1 insertion(+)
lixinxing@ubuntu:~/test$
```

我们可以看到 `welcome.txt` 文件已经提交到版本库中，如果我们修改 `welcome.txt` 文件，那么，怎样才能看到修改后的文件与版本库中的不同呐，我们可以通过非常强大的 `git diff` 命令来查看不同。

2.5 使用 `git diff` 命令查看修改

首先修改 `welcome.txt` 文件，在其后面追加一行。

```
$ echo "Nice to meet you." >> welcome.txt
```

现在将修改后的文件添加到暂存区。

```
$ git add welcome.txt
```

接着继续修改 `welcome.txt` 文件，在其后面追加一行。


```
$ echo "Bye-Bye." >> welcome.txt
```

现在，Git 工作区、暂存区、版本库中的 `welcome.txt` 文件都不一样了，我们可以通过 `git diff` 查看差异。

- 1) 不带任何参数的 `git diff` 命令显示工作区中的改动，即工作区中文件与提交任务（提交暂存区，`stage`）相比的差异。

```
$ git diff
```

```
diff --git a/welcome.txt b/welcome.txt
```

```
index 82d2e42..dbad484 100644
```

```
--- a/welcome.txt
```

```
+++ b/welcome.txt
```

```
@@ -1,2 +1,3 @@
```

```
Hello World!
```

```
Nice to meet you.
```

```
+Bye-Bye.
```

```
lixinxi...@ubuntu: ~/test
lixinxi...@ubuntu:~/test$ git diff
diff --git a/welcome.txt b/welcome.txt
index 82d2e42..dbad484 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello World!
 Nice to meet you.
+Bye-Bye.
lixinxi...@ubuntu:~/test$
```

- 2) 将工作区与 HEAD（当前分支）相比，会有更多的差异。

\$ git diff HEAD

```
diff --git a/welcome.txt b/welcome.txt
```

```
index 980a0d5..dbad484 100644
```

```
--- a/welcome.txt
```

```
+++ b/welcome.txt
```

```
@@ -1 +1,3 @@
```

```
Hello World!
```

+Nice to meet you.

+Bye-Bye.

```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~/test$ git diff HEAD
diff --git a/welcome.txt b/welcome.txt
index 980a0d5..dbad484 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,3 @@
 Hello World!
+Nice to meet you.
+Bye-Bye.
lixinxing@ubuntu:~/test$
```

- 3) 通过参数 `-cached` 或 `--staged` 调用 `git diff` 命令, 显示提交暂存区 (staged) 与版本库中文件的差异。

\$ git diff --staged

diff --git a/welcome.txt b/welcome.txt

index 980a0d5..82d2e42 100644

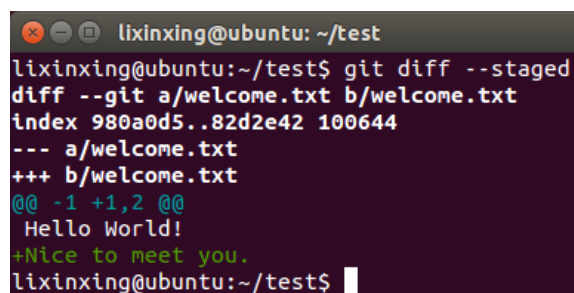
--- a/welcome.txt

+++ b/welcome.txt

```
@@ -1 +1,2 @@
```

```
Hello World!
```

```
+Nice to meet you.
```

A terminal window with a dark purple background. The prompt is 'lixinxing@ubuntu: ~/test'. The command 'git diff --staged' has been executed. The output shows a diff for 'a/welcome.txt' and 'b/welcome.txt' with index '980a0d5..82d2e42 100644'. The diff content is: '--- a/welcome.txt', '+++ b/welcome.txt', '@@ -1 +1,2 @@', 'Hello World!', '+Nice to meet you.'. The prompt is 'lixinxing@ubuntu:~/test\$' with a cursor.

```
lixinxing@ubuntu: ~/test$ git diff --staged
diff --git a/welcome.txt b/welcome.txt
index 980a0d5..82d2e42 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
Hello World!
+Nice to meet you.
lixinxing@ubuntu:~/test$
```

2.6 使用 git reset 命令改变历史

当有新的提交发生时，文件.git/refs/heads/master 内容就会发生改变，指向新的提交。

Git 提供了 git reset 命令，可以将“游标”.git/refs/heads/master 指向任意一个存在的提交 ID。下面示例人为的更改游标。

我们首先在工作区创建一个新文件，姑且为 new-commit.txt,

然后提交到版本库中。

```
$ touch new-commit.txt
```

```
$ git add new-commit.txt
```

```
$ git ci -m "add new-commit.txt"
```

```
[master 1924a0c] add new-commit.txt
```

```
2 files changed, 1 insertion(+)
```

```
create mode 100644 new-commit.txt
```

此时，工作区下有两个文件：

```
$ ls
```

```
new-commit.txt  welcome.txt
```

这时查看版本库引用空间下的 `master` 文件内容

```
$ cat .git/refs/heads/master
```

```
1924a0c69cc38e3ee2b45cfd41bdbdce3870f08f
```

可以看到，`master` 文件指向了新的提交 `1924a0c`。

使用 `git log` 查看提交日志，可以看到刚刚完成的提交：

```
$ git log --graph --oneline
```

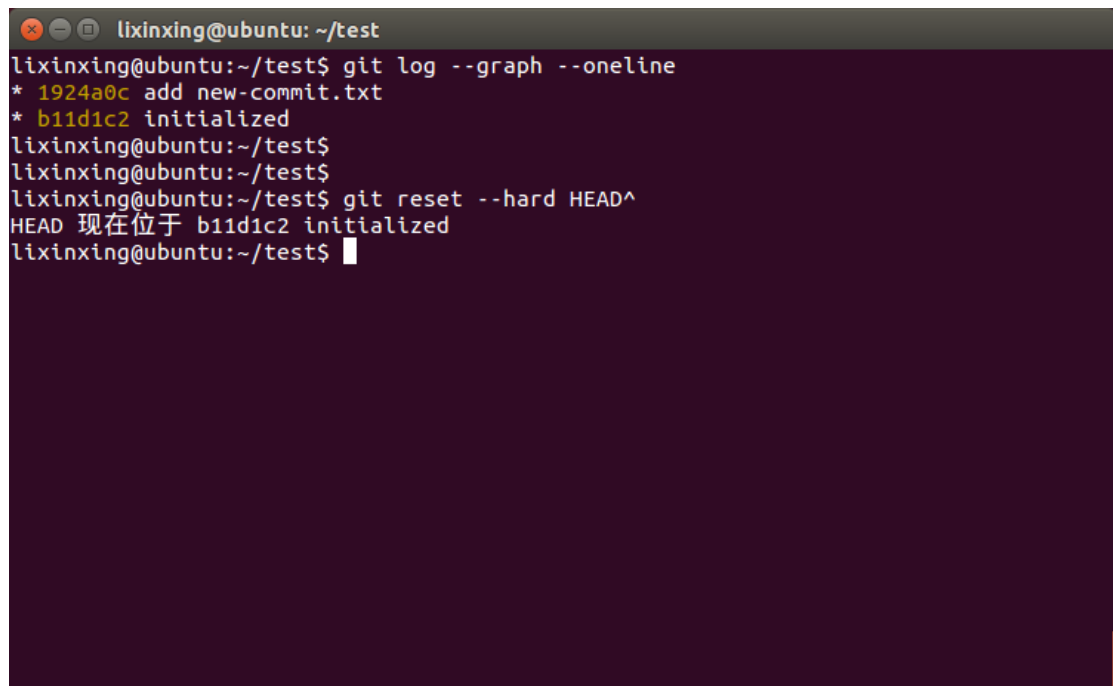
```
* 1924a0c add new-commit.txt
```

```
* b11d1c2 initialized
```

我们使用 `git reset` 命令更改提交

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

HEAD 现在位于 b11d1c2 initialized

A terminal window with a dark purple background and light green text. The window title is 'lixinxing@ubuntu: ~/test'. The terminal shows the following commands and output:

```
lixinxing@ubuntu:~/test$ git log --graph --oneline
* 1924a0c add new-commit.txt
* b11d1c2 initialized
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git reset --hard HEAD^
HEAD 现在位于 b11d1c2 initialized
lixinxing@ubuntu:~/test$
```

提示:

在命令中使用了 `--hard` 参数, 这会破坏工作区中未提交的改动, 使用前要慎重!

命令中使用的 `HEAD^` 代表版本库中的上一次提交。

符号`^`可以用于指代父提交。例如：

➤ `HEAD^` 代表版本库中最近一次提交的父提交。

➤ `HEAD^^` 代表 `HEAD^` 的父提交。

对于一个提交的多个父提交，可以在符号`^`后面用数字表示是第几个父提交。例如：

➤ `1924a0c^2` 表示提交 `1924a0c` 的多个父提交中的第二个父提交。

➤ `HEAD^1` 相当于 `HEAD^`。

➤ `HEAD^^2` 表示 `HEAD^` (HEAD 父提交) 的多个父提交中的第二个父提交。

在 `$ git reset --hard HEAD^` 命令后我们查看工作区文件：

```
$ ls
```

```
welcome.txt
```

可以看到 `new-commit.txt` 文件丢失了，查看引用文件内容：

```
$ cat .git/refs/heads/master
```

```
b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1
```

可以看到 `master` 分支的引用文件的指向更改为前一次提交的 ID 了。

我们可以通过 `git reflog` 命令操作分支 `master` 应用文件，使用 `show` 子命令显示此文件的内容。

```
$ git reflog show master | head -5
```

```
b11d1c2 master@{0}: reset: moving to HEAD^
```

```
1924a0c master@{1}: commit: add new-commit.txt
```

```
b11d1c2 master@{2}: commit (initial): initialized
```

`git reflog` 的输出将最新的更改放在了最前面显示，而且只显示每次更改的最终哈希值。最重要是 `git reflog` 命令的输出中还提供了一个方便易表达的表达式：`<refname>@{n}`，这个表达式的含义是引用`<refname>`之前第`{n}`此改变时的 SHA1 哈希值。

我们要想取消改变，重新找回 `new-commit.txt` 文件，可以使用命令

```
$ git reset --hard master@{1}
```

HEAD 现在位于 1924a0c add new-commit.txt

可以看到 new-commit.txt 也回来了，提交历史也回来了。

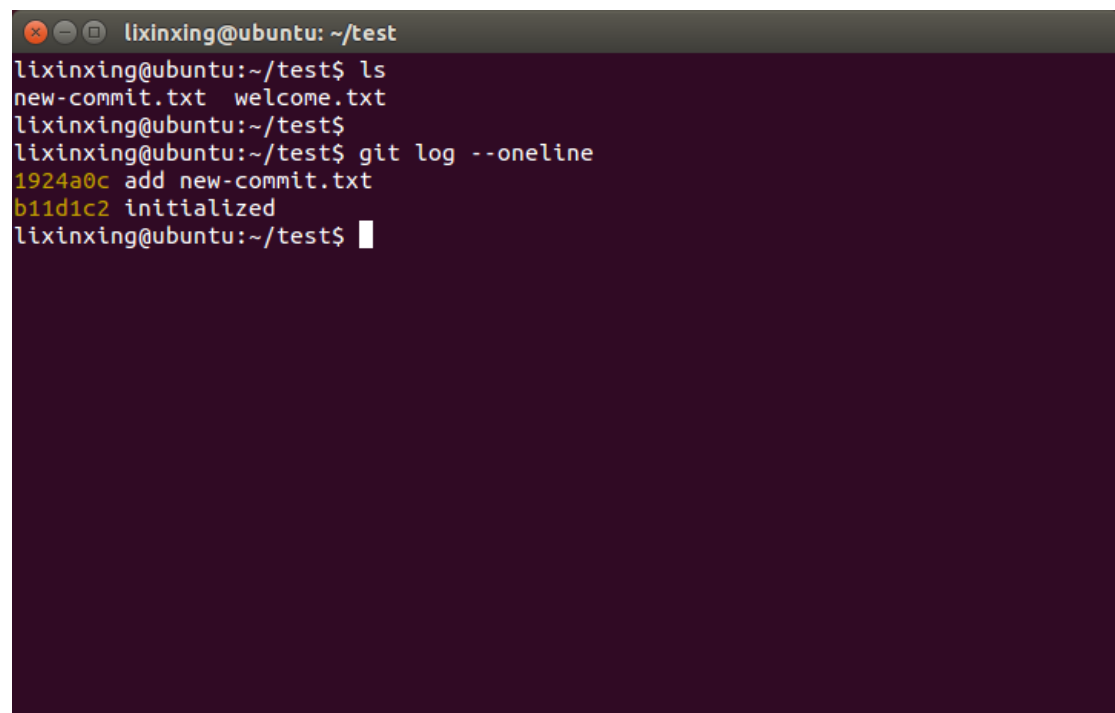
```
$ ls
```

```
new-commit.txt  welcome.txt
```

```
$ git log --oneline
```

```
1924a0c add new-commit.txt
```

```
b11d1c2 initialized
```

A terminal window with a dark purple background and light green text. The window title is 'lixinxing@ubuntu: ~/test'. The terminal shows the following commands and output:

```
lixinxing@ubuntu:~/test$ ls
new-commit.txt  welcome.txt
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git log --oneline
1924a0c add new-commit.txt
b11d1c2 initialized
lixinxing@ubuntu:~/test$
```

我们来看看 git reset 的几种常用命令。

➤ 命令：**git reset**

仅用 HEAD 指向的目录树重置暂存区，工作区不会受到影响。
即将之前用 `git add` 命令添加到暂存区的内容撤出暂存区，引用也没有改变。

➤ 命令：**`git reset HEAD`**

同上。

➤ 命令：**`git reset -- filename`**

仅将文件 `filename` 撤出暂存区，其他文件未收到影响。

➤ 命令：**`git reset HEAD -- filename`**

同上

➤ 命令：**`git reset -- soft HEAD^`**

工作区和暂存区不更改，但是引用向前回退一次。对最新提交的提交说明或提交的更改不满意时，撤销最新的提交以便重新提交。

➤ 命令：**`git reset HEAD^`**

工作区不改变，但是暂存区会退到上一次之前，引用也会回退一次。

➤ 命令：**git reset --hard HEAD^**

彻底撤销最近的提交，引用会回退到前一次，而且工作区和暂存区的内容都会回退到上一次提交时的状态。自上次提交以来的更改全部丢失。

第三篇 Git 进阶

在上一篇中，我们讲解了使用 `git` 的基本命名，掌握了如何创建 `git` 仓库、添加文件、将文件添加到暂存区、将文件提交、查看提交历史以及改变提交的内容等等。

在这一篇章中，我会教大家使用更加高级的 `git` 功能，使得更好的在开发中使用 `git` 这个伟大的版本控制工具。

3.1 远程仓库的使用

要参与任何一个 `Git` 项目的协作，必须要了解该如何管理远程仓库。远程仓库是指托管在网络上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写。同他人协作开发某个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。管理远程仓库的工作，包括添加远程库，移除废弃的远程库，管理各式远程库分支，定义是否跟踪这些分支，等等。本节我们将详细讨论远程库的管理和使用。

3.1.1 查看当前的远程库

要查看当前配置有哪些远程仓库，可以用 `git remote` 命令，它会列出每个远程库的简短名字。在克隆完某个项目后，至少可

以看到一个名为 `origin` 的远程库，Git 默认使用这个名字来标识你所克隆的原始仓库：

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository
in /private/tmp/ticgit/.git/ remote:
Counting objects: 595, done.
remote: Compressing objects:
100% (269/269), done. remote:
Total 595 (delta 255), reused 589
(delta 253) Receiving objects:
100% (595/595), 73.31 KiB | 1
KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd
ticgit
$ git
remote
origin
```

也可以加上 `-v` 选项（译注：此为 `—verbose` 的简写，取首字母），显示对应的克隆地址：

```
$ git remote -v
```

```
origin  
git://github.com/schacon/ticgit.git
```

如果有多个远程仓库，此命令将全部列出。比如在我的 **Grit** 项目中，可以看到：

```
$ cd grit  
  
$ git remote -v  
  
bakkdoor git://github.com/bakkdoor/grit.git cho45  
git://github.com/cho45/grit.git  
  
defunkt git://github.com/defunkt/grit.git koke  
git://github.com/koke/grit.git origin  
git@github.com:mojombo/grit.git
```

这样一来，我就可以非常轻松地从这些用户的仓库中，拉取他们的提交到本地。请注意，上面列出的地址只有 **origin** 用的是 SSH URL 链接，所以也只有这个仓库我能推送数据上去。

3.1.2 添加远程仓库

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用，运行 **git remote add [shortname] [url]**：

```
$ git remote origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin
git://github.com/schacon/ticgit.git
pb
git://github.com/paulboone/ticgit.git
```

现在可以用字符串 `pb` 指代对应的仓库地址了。比如说，要抓取所有 Paul 有的，但本地仓库没有的信息，可以运行 `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch] master -> pb/master * [new branch] ticgit
-> pb/ticgit
```

现在，Paul 的主干分支（`master`）已经完全可以在本地访问了，

对应的名字是 **pb/master**，你可以将它合并到自己的某个分支，或者切换到这个分支，看看有些什么有趣的更新。

3.1.3 从远程仓库抓取数据

正如之前所看到的，可以用下面的命令从远程仓库抓取数据到本地：

```
$ git fetch [remote-name]
```

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。（我们会在第三章详细讨论关于分支的概念和操作。）

如果是克隆了一个仓库，此命令会自动将远程仓库归于 **origin** 名下。所以，**git fetch origin** 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 **fetch** 以来别人提交的更新）。有一点很重要，需要记住，**fetch** 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支，只有当你确实准备好了，才能手工合并。

如果设置了某个分支用于跟踪某个远端仓库的分支（参见下节及第三章的内容），可以使用 **git pull** 命令自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下 **git clone** 命

令本质上就是自动创建了本地的 `master` 分支用于跟踪远程仓库中的 `master` 分支（假设远程仓库确实有 `master` 分支）。所以一般我们运行 `git pull`，目的都是要从原始克隆的远端仓库中抓取数据后，合并到工作目录中当前分支。

3.1.4 推送数据到远程仓库

项目进行到一个阶段，要同别人分享目前的成果，可以将本地仓库中的数据推送到远程仓库。实现这个任务的命令很简单：`git push [remote-name] [branch-name]`。如果要把本地的 `master` 分支推送到 `origin` 服务器上（再次说明下，克隆操作会自动使用默认的 `master` 和 `origin` 名字），可以运行下面的命令：

```
$ git push origin master
```

只有在所克隆的服务器上有写权限，或者同一时刻没有其他人在推数据，这条命令才会如期完成任务。如果你推数据前，已经有其他人推送了若干更新，那你的推送操作就会被驳回。你必须先把他们的更新抓取到本地，并到自己的项目中，然后才可以再次推送。有关推送数据到远程仓库的详细内容见第三章。

3.1.5 查看远程仓库信息

我们可以通过命令 `git remote show [remote-name]` 查看某个

远程仓库的详细信息，比如要看所克隆的 **origin** 仓库，可以运行：

```
$ git remote show origin

* remote origin

  URL: git://github.com/schacon/ticgit.git

  Remote branch merged with 'git
pull' while on branch master master

  Tracked
remote
branches
master ticgit
```

除了对应的克隆地址外，它还给出了许多额外的信息。它友善地告诉你如果是在 **master** 分支，就可以用 **git pull** 命令抓取数据合并到本地。另外还列出了所有处于跟踪状态中的远端分支。

实际使用过程中，**git remote show** 给出的信息可能会像这样：

```
$ git remote show origin
* remote origin

URL: git@github.com:defunkt/github.git

Remote branch merged with 'git
pull' while on branch issues issues

Remote branch merged with 'git
pull' while on branch master master

New remote branches (next fetch will
store in remotes/origin) caching

Stale tracking branches
(use 'git remote prune')

libwalker walker2

Tracked
remote
branches acl
apiv2
dashboard2
issues
master
postgres

Local branch pushed
with 'git push'
```

```
master:master
```

它告诉我们，运行 `git push` 时缺省推送的分支是什么（译注：最后两行）。它还显示了有哪些远端分支还没有同步到本地（译注：第六行的 `caching` 分支），哪些已同步到本地的远端分支在远端服务器上已被删除。

3.1.6 远程仓库的删除和重命名

在新版 Git 中可以用 `git remote rename` 命令修改某个远程仓库的简短名称，比如想把 `pb` 改成 `paul`，可以这么运行：

```
$ git remote rename pb paul  
  
$ git  
remote  
origin  
paul
```

注意，对远程仓库的重命名，也会使对应的分支名称发生变化，原来的 `pb/master` 分支现在成了 `paul/master`。

碰到远端仓库服务器迁移，或者原来的克隆镜像不再使用，又或者某个参与者不再贡献代码，那么需要移除对应的远端仓库，可以运行 `git remote rm` 命令：

```
$git remote rm paul  
  
$ git remote origin
```

3.2 打标签

同大多数 VCS 一样，Git 也可以对某一时间点上的版本打上标签。人们在发布某个软件版本（比如 `v1.0` 等等）的时候，经常这么做。本节我们一起来学习如何列出所有可用的标签，如何新建标签，以及各种不同类型标签之间的差别。

3.2.1 列显已有的标签

列出现有标签的命令非常简单，直接运行 `git tag` 即可：

```
$ git tag  
  
v0.1
```

v0.5

v0.8

v1.0

显示的标签按字母顺序排列，所以标签的先后并不表示重要程度的轻重。

我们可以用特定的搜索模式列出符合条件的标签。在 Git 自身项目仓库中，有着超过 240 个标签，如果你只对 1.4.2 系列的版本感兴趣，可以运行下面的命令：

```
$ git tag -l 'v1.4.2.*' v1.4.2.1 v1.4.2.2 v1.4.2.3 v1.4.2.4
```

3.2.2 新建标签

Git 使用的标签有两种类型：轻量级的（**lightweight**）和含附注的（**annotated**）。轻量级标签就像是个不会变化的分支，实际上它就是个指向特定提交对象的引用。而含附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 **GNU Privacy Guard (GPG)** 来签署或验证。一般我们都建议使用含附注型的标签，以便保留相关信息；当然，如果只是临时性加注标签，或者不需要旁注额外信息，用轻量级标签也没问题。

3.2.3 含附注的标签

创建一个含附注类型的标签非常简单，用 **-a** 指定标签名字即可：

```
$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v0.5

v0.8

v1.0

v1

.4
```

而 **-m** 选项则指定了对应的标签说明，Git 会将此说明一同保存在标签对象中。如果在此选项后没有给出具体的说明内容，Git 会启动文本编辑软件供你输入。

可以使用 **git show** 命令查看相应标签的版本信息，并连同显示打标签时的提交对象。

```
$ git show v1.4

tag v1.4

Tagger: li xinxing <lixinxing@lxx1.com>

Date:   Tue Feb 17 21:36:35 2015 +0800
```

```
my version 1.4
```

```
commit 1924a0c69cc38e3ee2b45cfd41bdbdce3870f08f
```

```
Author: li xinxing <lixinxing@lxx1.com>
```

```
Date: Mon Feb 16 21:48:26 2015 +0800
```

```
add new-commit.txt
```

```
Signed-off-by: li xinxing <lixinxing@lxx1.com>
```

```
diff --git a/new-commit.txt b/new-commit.txt
```

```
new file mode 100644
```

```
index 0000000..e69de29
```

```
diff --git a/welcome.txt b/welcome.txt
```

```
index 980a0d5..82d2e42 100644
```

```
--- a/welcome.txt
```

```
+++ b/welcome.txt
```

```
@@ -1 +1,2 @@
```

```
Hello World!
```

```
+Nice to meet you.
```

我们可以看到在提交对象信息上面，列出了此标签的提交者和提交时间，以及相应的标签说明。

3.2.4 签署标签

如果你有自己的私钥，还可以用 GPG 来签署标签，只需要把之前的 `-a` 改为 `-s`（译注：取 Signed 的首字母）即可：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the
secret key for user: "Scott Chacon
<schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created
2009-02-09
```

现在再运行 `git show` 会看到对应的 GPG 签名也附在其内：

```
$ git show v1.5 tag v1.5
Tagger: Scott Chacon
<schacon@gee-mail.com> Date: Mon Feb
9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP
SIGNATURE-----Version: GnuPG v1.4.8
(Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQF
```

```
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----

commit
15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>
Date:Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

稍后我们再学习如何验证已经签署的标签。

3.2.5 轻量级标签

轻量级标签实际上就是一个保存着对应提交对象的校验和信息的文件。要创建这样的标签，一个 **-a**，**-s** 或 **-m** 选项都不用，直接给出标签名字即可：

```
$ git
```

```
    t
```

```
    a
```

```
    g
```

```
    v
```

```
2
.
0

$ git tag
v0.1
v0.5
v0.8
v1.0
v1.4
v2.0
```

现在运行 `git show` 查看此标签信息，就只有相应的提交对象摘要：

```
$ git show v2.0
```

```
commit 1924a0c69cc38e3ee2b45cfd41bdbdce3870f08f
```

```
Author: li xinxing <lixinxing@lxx1.com>
```

```
Date: Mon Feb 16 21:48:26 2015 +0800
```

```
add new-commit.txt
```

```
Signed-off-by: li xinxing <lixinxing@lxx1.com>
```

```
diff --git a/new-commit.txt b/new-commit.txt
```

```
new file mode 100644
```

```
index 0000000..e69de29
```

```
diff --git a/welcome.txt b/welcome.txt
```

```
index 980a0d5..82d2e42 100644
```

```
--- a/welcome.txt
```

```
+++ b/welcome.txt
```

```
@@ -1 +1,2 @@
```

```
Hello World!
```

```
+Nice to meet you.
```

3.2.7 后期加注标签

你甚至可以在后期对早先的某次提交加注标签。比如在下面展示的

提交历史中：

```
$ git log --pretty=oneline  
  
1924a0c69cc38e3ee2b45cfd41bdbdce3870f08f  
add new-commit.txt  
  
b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1  
initialized
```

我们忘了在提交 “initialized” 后为此项目打上版本号 v0.9，没关系，现在也能做。只要在打标签的时候跟上对应提交对象的校验和（或前几位字符）即可：

```
$ git tag -a v0.9 b11d1c2 -m "my version v0.9"
```

可以看到我们已经补上了标签：

```
$ git tag  
  
v0.1  
  
v0.5  
  
v0.8  
  
v0.9  
  
v1.0
```

v1.4

v2.0

\$ git show v0.9

tag v0.9

Tagger: li xinxing <lixinxing@lxx1.com>

Date: Tue Feb 17 21:46:02 2015 +0800

my version v0.9

commit

b11d1c220a8e2bc21fc5a8bccd08650411e0bfc1

Author: li xinxing <lixinxing@lxx1.com>

Date: Mon Feb 16 11:33:03 2015 +0800

initialized

diff --git a/welcome.txt b/welcome.txt

new file mode 100644

index 0000000..980a0d5

--- /dev/null

+++ b/welcome.txt

@@ -0,0 +1 @@

```
+Hello World!
```

3.2.8 分享标签

默认情况下，`git push` 并不会把标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。

其命令格式如同推送分支，运行 `git push origin [tagname]` 即可：

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB,
done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag] v1.5 -> v1.5
```

如果要一次推送所有（本地新增的）标签上去，可以使用 `--tags` 选

项:

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38),
done.
Writing objects: 100% (44/44), 4.56 KiB,
done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git

* [new tag] v0.1 -> v0.1
* [new tag] v1.2 -> v1.2
* [new tag] v1.4 -> v1.4
* [new tag] v1.4-lw -> v1.4-lw
* [new tag] v1.5 -> v1.5
```

现在，其他人克隆共享仓库或拉取数据同步后，也会看到这些标签。

3.3 Git 分支

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的工作时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需

要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

有人把 Git 的分支模型称为“必杀技特性”，而正是因为它，将 Git 从版本控制系统家族里区分出来。

Git 有何特别之处呢？Git 的分支可谓是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。

3.3.1 何谓分支

为了理解 Git 分支的实现方式，我们需要回顾一下 Git 是如何储存数据的。Git 保存的不是文件差异或者变化量，而只是一系列文件快照。

在 Git 中提交时，会保存一个提交（commit）对象，它包含一个指向暂存内容快照的指针，作者和相关附属信息，以及一定数量（也可能没有）指向该提交对象直接祖先的指针：第一次提交是没有直接祖先的，普通提交有一个祖先，由两个或多个分支合并产生的提交则有多个祖先。

Git 中的分支，其实本质上仅仅是个指向 `commit` 对象的可变指针。Git 会使用 `master` 作为分支的默认名字。在若干次提交后，你其实已经有了一个指向最后一次提交对象的 `master` 分支，它在每次提交的时候都会自动向前移动。

3.3.2 新建分支

Git 新建分支的方法很简单，通过 `git branch` 命令即可创建于一个新的分支

```
$ git branch testing
```

Git 里有名为 `HEAD` 的特别指针，他记录着指向的分支，在新建完 `testing` 分支后，我们通过查看 `.git/refs/heads/testing` 文件的内容，发现与 `master` 分支引用文件的内容相同，这就说明，新建分支只是使得 `HEAD` 指针指向了 `testing` 分支。

```
$ cat .git/refs/heads/testing
```

```
1924a0c69cc38e3ee2b45cfd41bdbdce3870f08f
```

运行 `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 `master` 分支里工作。

```
$ git branch
```

```
* master
```

testing

要切换到其他分支，可以执行 `git checkout` 命令。我们现在在转换到新建的 `testing` 分支：

```
$ git checkout testing
```

这样 `HEAD` 就指向了 `testing` 分支。

```
$ git checkout testing
```

切换到分支 'testing'

```
$ git branch
```

```
master
```

```
* testing
```

3.3.3 合并分支

现在分支已经成功创建，我们可以在 `testing` 分支上开发新功能而不影响 `master` 分支，比如我们新建文件 `new-tools.txt` 并提交：

```
$ touch new-tools.txt
```

```
$ git add new-tools.txt
```

```
$ git commit -m "add new-tools.txt"
```

```
[testing 555be0e] add new-tools.txt
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 new-tools.txt
```

如果分支开发完成，我们可以用 `git merge` 命令来进行合并：

```
$ git checkout master
```

切换到分支 'master'

```
$ git merge testing
```

```
更新 1924a0c..555be0e
```

```
Fast-forward
```

```
new-tools.txt | 0
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 new-tools.txt
```

请注意，合并时出现了“Fast forward”（快进）提示。由于当前 `master` 分支所在的 `commit` 是要并入的 `testing` 分支的直接上游，Git 只需把指针直接右移。换句话说，如果顺着一个分支走下去可以到达另一个分支，那么 Git 在合并两者时，只会简单地把指针前移，因为没有什么分歧需要解决，所以这个过程叫做快进（Fast forward）。现在的目录变为当前 `master` 分支指向的 `commit` 所对应的快照。

3.3.4 删除分支

在 `testing` 分支成功合并到 `master` 分支后，现在 `testing` 分支与 `master` 分支指向相同，已经没什么用了，可以直接删除，使用 `git branch` 命令的 `-d` 选项表示删除分支：

```
$ git branch -d testing
```

已删除分支 `testing`（曾为 `555be0e`）。

现在只剩下 `master` 分支了：

```
$ git branch
```

```
* master
```

3.3.5 合并冲突

在分支合并时，如果没有冲突，则会基本合并，在合并后删除开发分支即可。

但经常在合并时会遇到冲突，如果你修改了两个待合并分支里同一个文件的同一部分，Git 就无法干净地把两者合到一起，在这种情况下，基本只能手动解决冲突。

```
$ git merge test
```

自动合并 `index.html`

冲突（添加/添加）：合并冲突于 `index.html`

自动合并失败，修正冲突然后提交修正的结果。

Git 作了合并，但没有提交，它会停下来等你解决冲突。要看看哪些文件在合并时发生冲突，可以用 `git status` 查阅：

```
$ git status index.html
```

位于分支 `master`

Git 指南

您有尚未合并的路径。

(解决冲突并运行 "git commit")

未合并的路径:

(使用 "git add <file>..." 标记解决方案)

双方添加: index.html

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")



```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~/test$ git status index.html
位于分支 master
您有尚未合并的路径。
(解决冲突并运行 "git commit")

未合并的路径:
(使用 "git add <file>..." 标记解决方案)

      双方添加:      index.html

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
lixinxing@ubuntu:~/test$
```

查看冲突文件, 可以看到冲突的地方已经用符号<<<<<<<、>>>>>>>、=====标出来了:

\$ cat index.html

<<<<<<< HEAD

<div id="footer">contact : email.support@github.com</div>

=====

<div id="footer"> please contact us at support@github.com

</div>

```
>>>>>> test
```

可以看到 ===== 隔开的上半部分，是 HEAD（即 master 分支，在运行 merge 命令时检出的分支）中的内容，下半部分是在 test 分支中的内容。解决冲突的办法无非是二者选其一或者由你亲自整合到一起。比如你可以通过把这段内容替换为下面这样来解决：

```
<div id="footer"> please contact us at support@github.com  
contact : email.support@github.com  
</div>
```

这个解决方案各采纳了两个分支中的一部分内容，而且我还删除了 <<<<<<, =====, 和>>>>>> 这些行。在解决了所有文件里的所有冲突后，运行 git add 将把它们标记为已解决（resolved）。因为一旦暂存，就表示冲突已经解决。

冲突解决后提交即可。

```
$ git add index.html
```

```
$ git commit -m "merge"
```

3.3.6 分支管理

git branch 命令不仅仅能创建和删除分支，如果不加任何参数，它会给出当前所有分支的清单。

```
$ git branch
```

```
* master
```

```
test
```

```
testing
```

git branch -merge 查看哪些分支已被并入当前分支。

```
$ git branch --merge
```

```
* master
```

```
test
```

git branch --no-merged 查看尚未合并的工作。

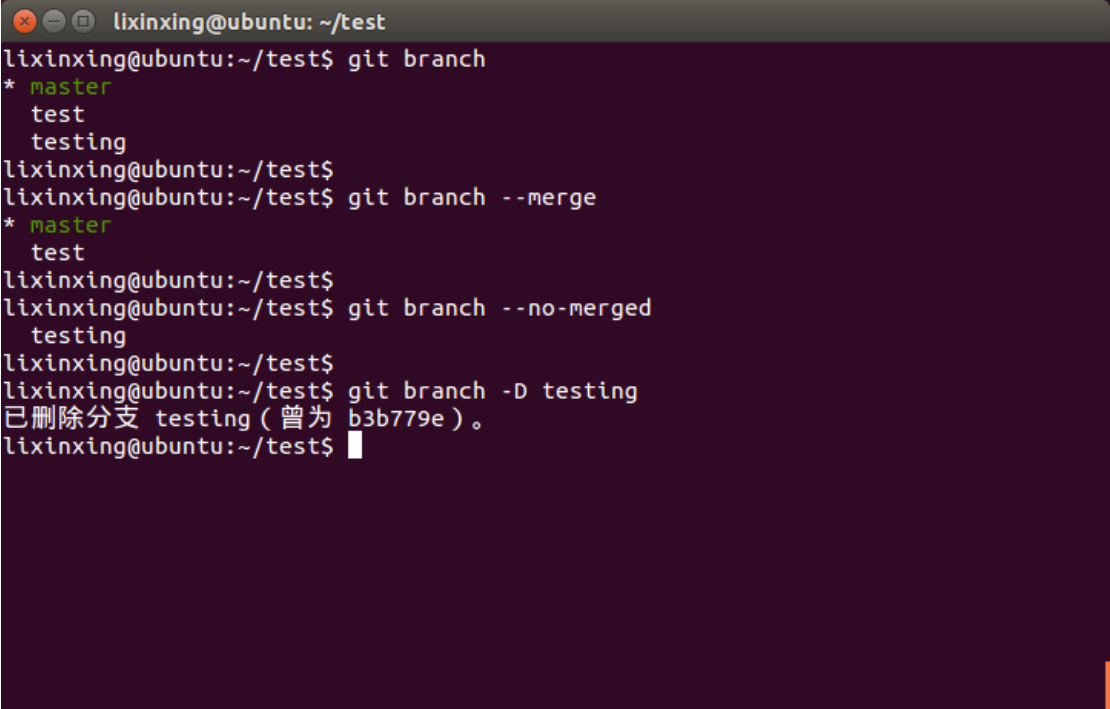
```
$ git branch --no-merged
```

```
testing
```

git branch -D testing 强制删除未合并的分支。

```
$ git branch -D testing
```

已删除分支 testing（曾为 b3b779e）。



```
lixinxing@ubuntu: ~/test
lixinxing@ubuntu:~/test$ git branch
* master
  test
  testing
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git branch --merge
* master
  test
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git branch --no-merged
testing
lixinxing@ubuntu:~/test$
lixinxing@ubuntu:~/test$ git branch -D testing
已删除分支 testing（曾为 b3b779e）。
lixinxing@ubuntu:~/test$
```


3.4 远程分支

3.4.1 推送

要想和其他人分享某个分支，你需要把它推送到一个你拥有写权限的远程仓库。你的本地分支不会被自动同步到你引入的远程分支中，除非你明确执行推送操作。换句话说，对于无意分享的，你尽可以保留为私人分支，而只推送那些协同工作的特性分支。

如果你有个叫 `serverfix` 的分支需要和他人一起开发，可以运行

`git push` (远程仓库名) (分支名):

```
$ git push origin serverfix Counting objects: 20, done.
```

```
Compressing objects: 100% (14/14), done.
```

```
Writing objects: 100% (15/15), 1.74 KiB, done.
```

```
Total 15 (delta 5), reused 0 (delta 0)
```

```
To git@github.com:schacon/simplegit.git
```

```
* [new branch] serverfix -> serverfix
```

接下来，当你的协作者再次从服务器上获取数据时，他们将

得到一个新的远程分支 `origin/serverfix`:

```
$ git fetch origin remote: Counting objects: 20, done. remote:
Compressing objects: 100% (14/14), done. remote: Total 15 (delta 5), reused 0
(delta 0) Unpacking objects: 100% (15/15), done.

From git@github.com:schacon/simplegit

* [new branch]    serverfix -> origin/serverfix
```

值得注意的是，在 `fetch` 操作抓来新的远程分支之后，你仍然无法在本地编辑该远程仓库。换句话说，在本例中，你不会有一个新的 `serverfix` 分支，有的只是一个你无法移动的 `origin/serverfix` 指针。

如果要把该内容合并到当前分支，可以运行 `git merge origin/serverfix`。如果想要一份自己的 `serverfix` 来开发，可以在远程分支的基础上分化出一个新的分支来：

```
$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch
refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"
```

这会切换到新建的 `serverfix` 本地分支，其内容同远程分支 `origin/serverfix` 一致，你可以在里面继续开发

了。

3.4.2 跟踪分支

从远程分支检出的本地分支，称为跟踪分支(tracking branch)。跟踪分支是一种和远程分支有直接联系的本地分支。在跟踪分支里输入 `git push`，Git 会自行推断应该向哪个服务器的哪个分支推送数据。反过来，在这些分支里运行 `git pull` 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

在克隆仓库时，Git 通常会自动创建一个 `master` 分支来跟踪 `origin/master`。这正是 `git push` 和 `git pull` 一开始就能正常工作的原因。当然，你可以随心所欲地设定为其它跟踪分支，比如 `origin` 上除了 `master` 之外的其它分支。刚才我们已经看到了这样的一个例子：`git checkout -b [分支名] [远程名]/[分支名]`。如果你有 1.6.2 以上版本的 Git，还可以用 `--track` 选项简化：

```
$ git checkout --track origin/serverfix

Branch serverfix set up to track remote
branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"
```

要为本地分支设定不同于远程分支的名字，只需在前个版本的命令里换个名字：

```
$ git checkout -b sf origin/serverfix
```

```
Branch sf set up to track remote branch  
refs/remotes/origin/serverfix.
```

```
Switched to a new branch "sf"
```

现在你的本地分支 `sf` 会自动向 `origin/serverfix` 推送和抓取数据了。

3.4.3 删除远程分支

如果不再需要某个远程分支了，比如搞定了某个特性并把它合并进了远程的 `master` 分支（或任何其他存放稳定代码的地方），可以用这个非常无厘头的语法来删除它：`git push [远程名]:[分支名]`。如果想在服务器上删除 `serverfix` 分支，运行下面的命令：

```
$ git push origin :serverfix
```

```
To git@github.com:schacon/simplegit.git
```

```
- [deleted] serverfix
```

3.5 小结

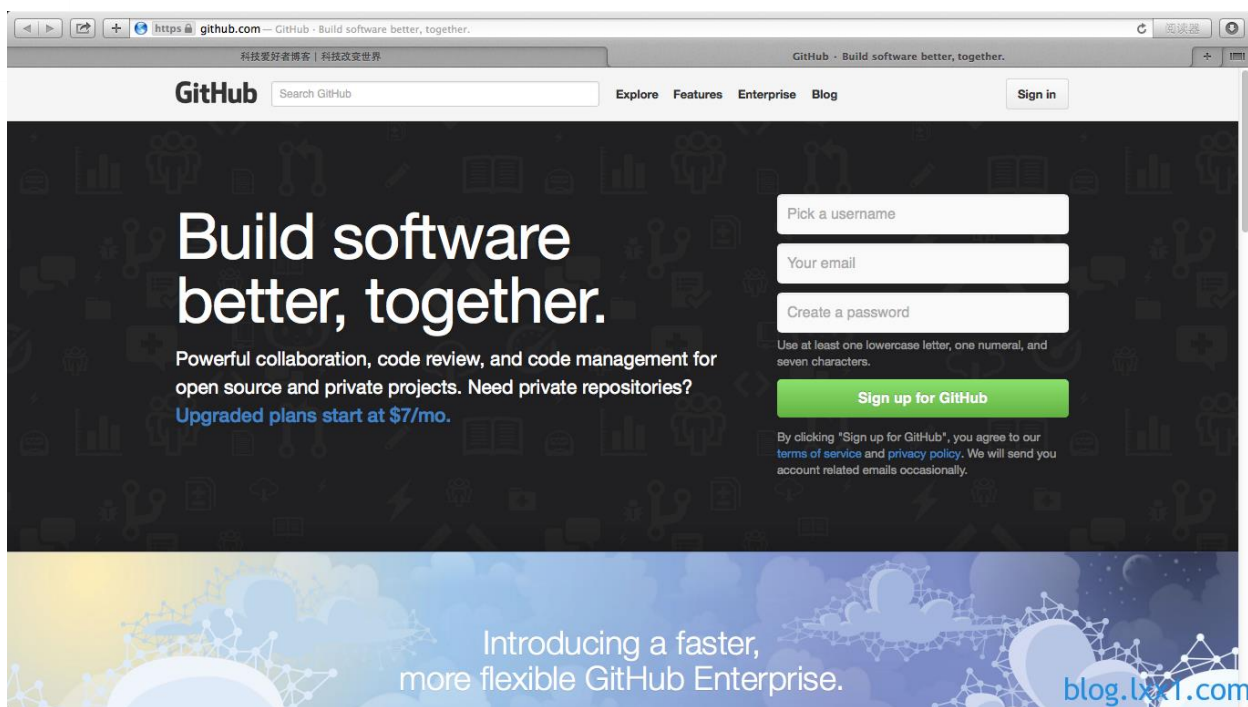
读到这里，你应该已经学会了如何创建分支并切换到新分支；在不同分支间转换；合并本地分支；把分支推送到共享服务器上，同世界分享；

第四篇 使用 Github

GitHub 和大多数的代码托管站点在处理项目命名空间的方式上略有不同。GitHub 的设计更侧重于用户，而不是全部基于项目。意谓本人在 GitHub 上托管一个 grit 项目的话，它将不会出现在 `github.com/grit`，而是在 `github.com/xixningli/grit`。不存在所谓某个项目的官方版本，所以假如第一作者放弃了某个项目，它可以无缝转移到其它用户的旗下。

GitHub 同时也是一个向使用私有仓库的用户收取费用的商业公司，不过所有人都可以快捷的得到一个免费账户并且在上面托管任意多的开源项目。我们将快速介绍一下该过程。

你需要访问 <http://github.com> 登陆自己的 github 账号或者建立账号，[这里我们演示如何建立账户并配置开发环境。](#)



4.1 建立账户

使用 github 服务的第一步是建立 github 账户，我们这里演示如何建立一个免费的 github 账号。

1.访问 Pricing and Signup （价格与注册）页面 <http://github.com/plans> 并点击 Free account （免费账户）的 “Sign Up（注册）”，进入注册页面。

2.这里要求选择一个系统中尚未存在的用户名，提供一个与之相连的电邮地址，以及一个密码。

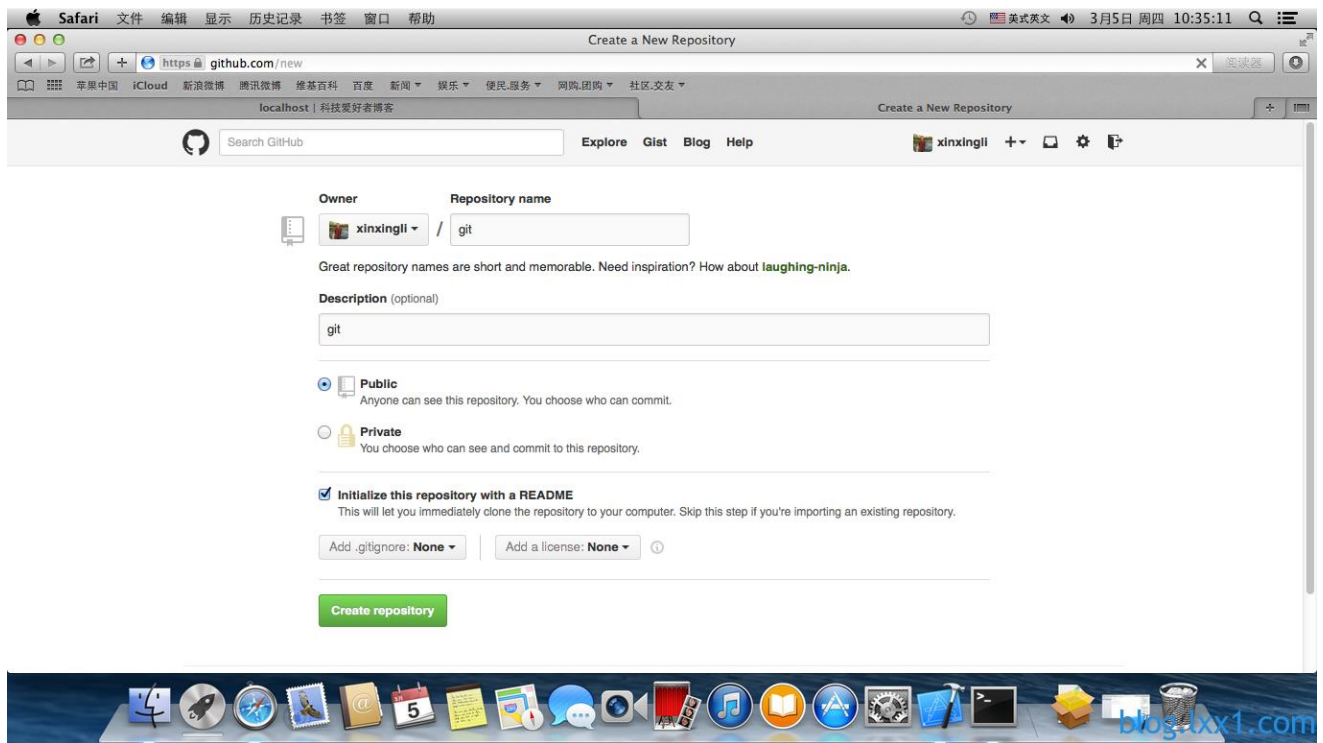
3.点击 “I agree, sign me up （同意条款，让我注册）” 按钮就能进入新用户的控制面板。接着可以建立仓库了。

4.2 建立仓库

点击用户面板上仓库旁边的“create a new one（新建）”连接。进入 **Create a New Repository**（新建仓库）表格。

唯一必做的仅仅是提供一个项目名称，当然也可以添加一点描述。搞定这些以后，点“**Create Repository**（建立仓库）”按钮。新仓库就建立起来了。

由于还没有提交代码，GitHub 会展示如何创建一个新项目，如何推送一个现存项目，以及如何从一个公共的 Subversion 仓库导入项目。



仓库建立后你就可以提交代码，如果要在本地端使用 SSH 提交代码，则需要在本地产生成 SSH 密钥，并将公钥添加到 github

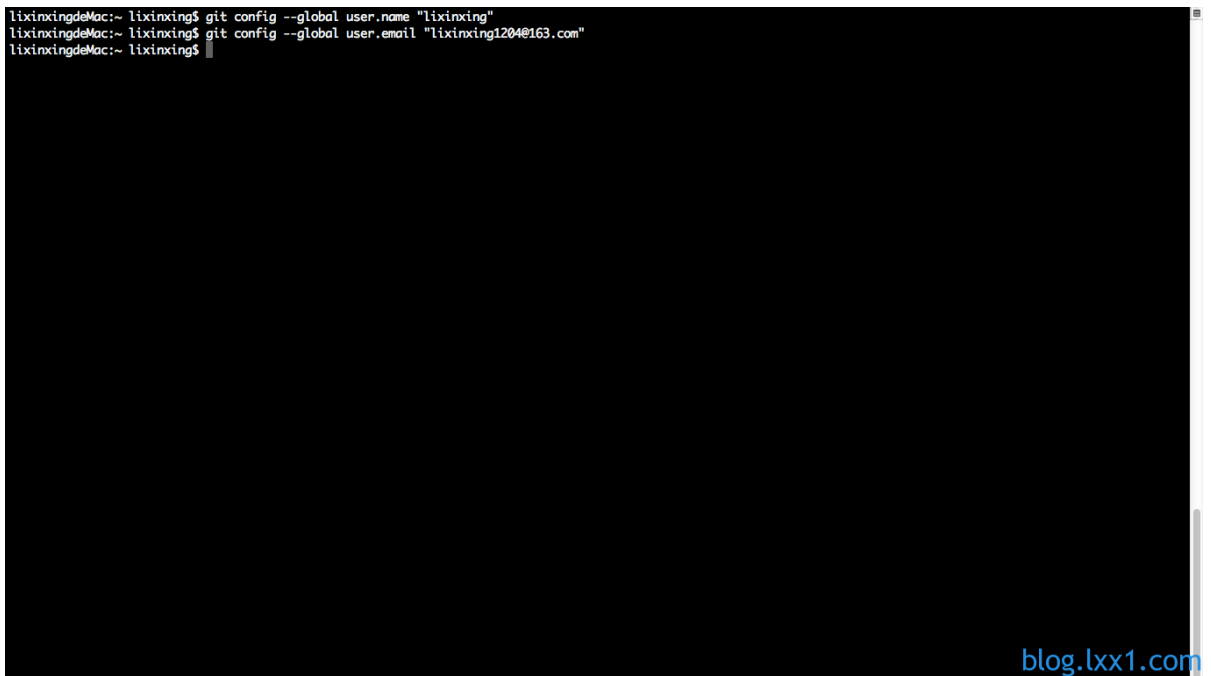
上，这里我演示下过程：

4.3 生成密钥

打开终端，首先配置 git 的用户名和邮箱：

```
$ git config --global user.name "lixinxing"
```

```
$ git config --global user.email "lixinxing1204@163.com"
```

A screenshot of a macOS terminal window. The prompt is 'lixinxingdeMac:~ lixinxing\$'. The first command entered is 'git config --global user.name "lixinxing"', followed by 'git config --global user.email "lixinxing1204@163.com"'. The terminal output shows the commands being executed. A blue watermark 'blog.lxx1.com' is visible in the bottom right corner of the terminal window.

```
lixinxingdeMac:~ lixinxing$ git config --global user.name "lixinxing"
lixinxingdeMac:~ lixinxing$ git config --global user.email "lixinxing1204@163.com"
lixinxingdeMac:~ lixinxing$
```

生成密钥：

```
$ ssh-keygen -t rsa -C "lixinxing1204@163.com"
```

提示：

Your identification has been saved in /user/lixinxing/.ssh/id_rsa.

Your public key has been saved in /user/lixinxing/.ssh/id_rsa.pub.

The key fingerprint is:

.....

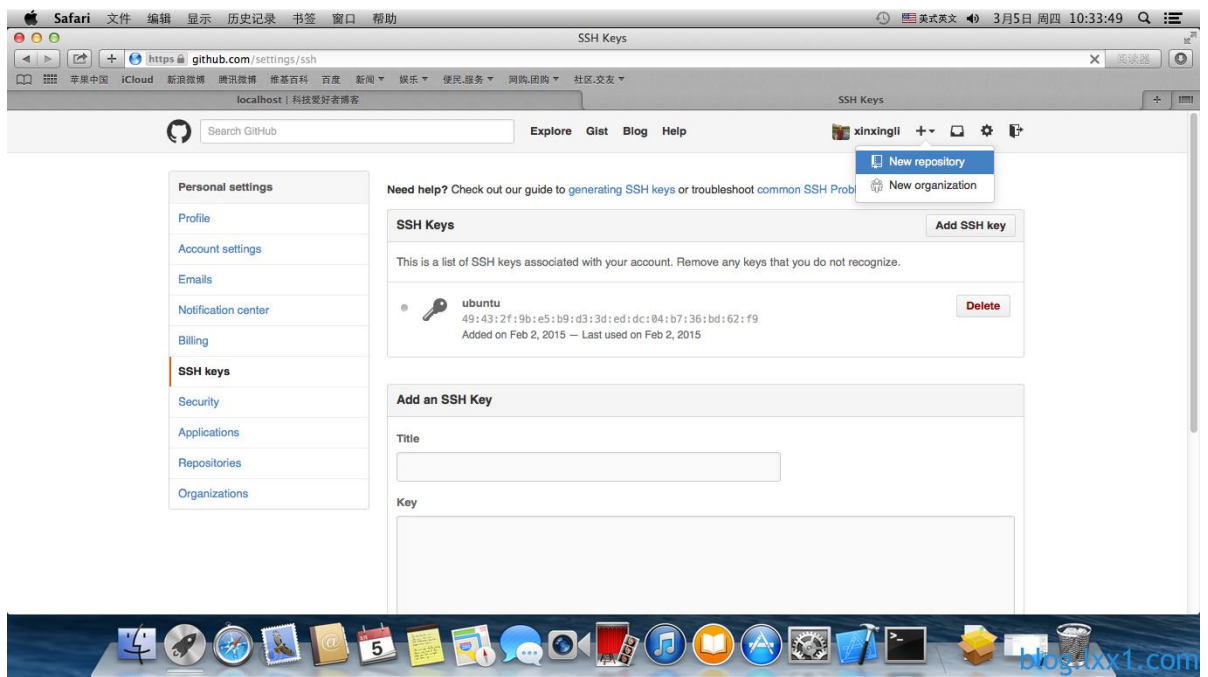
最后得到了两个文件：id_rsa 和 id_rsa.pub

```
lixixingdeMac:~ lixixing$ ssh-keygen -t rsa -C "lixixing1204@163.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/lixixing/.ssh/id_rsa):
Created directory '/Users/lixixing/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/lixixing/.ssh/id_rsa.
Your public key has been saved in /Users/lixixing/.ssh/id_rsa.pub.
The key fingerprint is:
27:b1:1e:8c:9c:65:64:7a:75:da:c2:29:78:f7:1a:72 lixixing1204@163.com
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      .               |
|      o =            |
|     o B *           |
|    . O = o          |
|   + S E .           |
|  . * o              |
|  .                  |
+-----+
lixixingdeMac:~ lixixing$
```

blog.lxx1.com

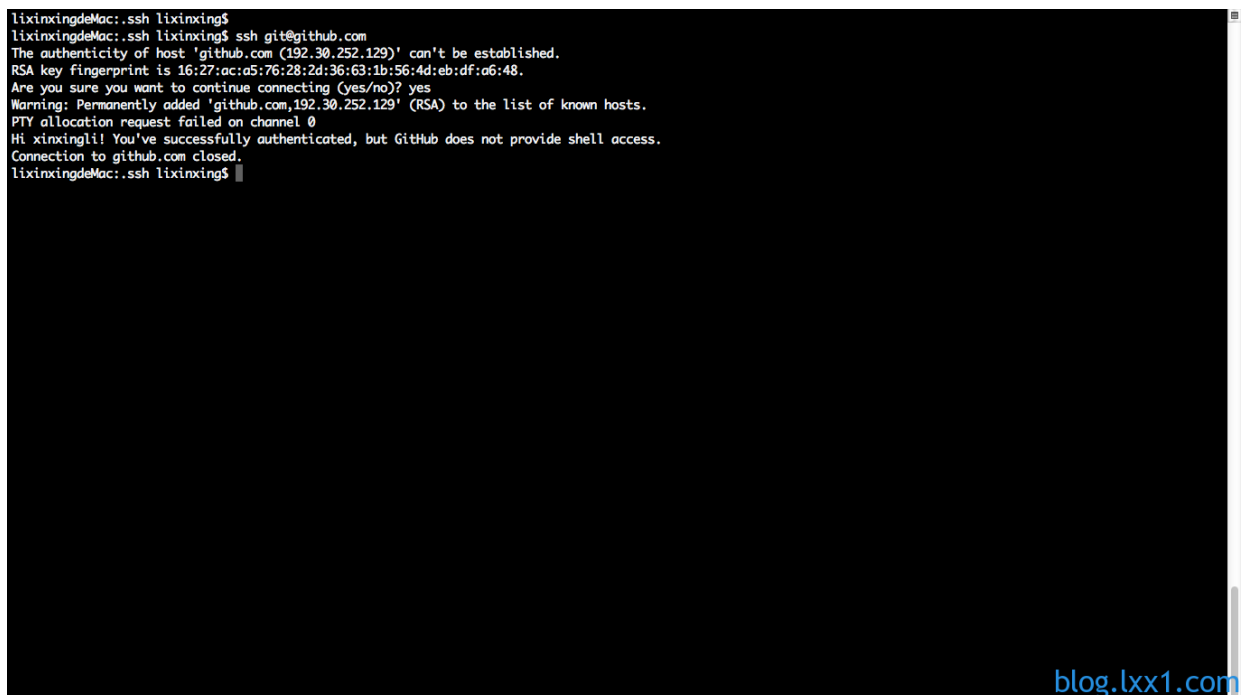
打开 id_rsa.pub 文件，将里面的一串字母添加到 github 的 ssh 公钥中。

Git 指南



测试:

输入命令 `$ssh git@github.com`



好的，开始使用 github 吧！

第五篇 Git 其他应用

在这一篇中，我们看看 Git 的其他应用。

5.1 使用 etckeeper 备份 Linux 下的/etc 目录。

/etc 目录下保存了 Linux 系统大量软件的配置信息，如果其中的文件被错误编辑或者删除，将会损失惨重，在这一节中，我们利用 keeper 这个工具来帮助实现/etc 目录的持续备份。

第六篇 Git 常用命令

1.创建仓库

```
mkdir git
```

```
cd git    ——创建/home/XXX/git 空目录
```

2.通过 git init 命令把这个目录变成 Git 可以管理的仓库:

```
git init ——初始化 Git 仓库
```

3.用命令 git add 告诉 Git，把文件添加到仓库(实际上就是把文件修改添加到暂存区):

```
git add filename
```

4.用命令 git commit 告诉 Git，把文件提交到仓库(实际上就是把暂存区的所有内容提交到当前分支):

```
git commit -m "有意义的附加说明"
```

5.随时掌握工作区的状态

```
git status
```

6.查看文件被修改的内容

```
git diff
```

7.查看代码的历史版本号

```
git log
```

```
git log --pretty=oneline
```

 ——要求版本信息只能在一行中显示

8.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

```
git reset --hard commit_id
```

或 `git reset --hard HEAD^` (`HEAD^^` 等等)

9.查看命令历史，以便确定要回到未来的哪个版本

```
git reflog
```

10.弄明白 Git 的工作区(当前分区)和暂存区

11.理解 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

12.撤销修改

命令 `git checkout -- filename` 意思就是，把 filename 文件在

工作区的修改全部撤销，这里有两种情况：

一种是 filename 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 filename 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。

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`。

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 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，版本回退，不过前提是没有推送到远程库。

13. 删除文件

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。

14.将本地仓库与 github 仓库关联起来

往里面添加文件:

```
touch README.md
```

```
git init
```

```
git add README.md
```

```
git commit -m "first commit"
```

```
git remote add origin git@github.com:xinxingli/git.git
```

```
git push -u origin master
```

将本地仓库同步 github 仓库:

```
git remote add origin git@github.com:xinxingli/git.git
```

```
git push -u origin master
```

然后, 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

15.多人协作一个项目的时候, 我们每个人可以通过从远程仓库克隆一份来作为己用。

```
git clone git@github.com:xinxingli/git.git
```

16.创建分支并且切换到分支

```
git checkout -b dev
```

```
Switched to a new branch 'dev'
```

等价于：

```
git branch dev  
git checkout dev  
Switched to branch 'dev'
```

查看分支：

```
git branch
```

将次分支合并到主分支上面：

```
git merge dev
```

删除分支：

```
git branch -d dev  
Deleted branch dev (was fec145a).
```

17.解决冲突

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。

18.Bug 修复

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，先把手头工作 `git stash` 一下，然后去修复 bug，修复后，再 `git stash pop`，回到工作现场

19.开发新功能

开发一个新功能，最好新建一个分支：

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D name` 强行删除。

20.参与开源项目先要克隆一份到本地

```
git clone git@github.com:xinxingli/git.git
```