GitHub笔记

看Facebook或Twitter不一定能了解一个人的品性，而看GitHub能了解一个程序员的实力。

在开发者之间引发化学反应的Pull Request

Pull Request是指开发者在本地对源代码进行更改后，向GitHub中托管的Git仓库请求合并的功能。开发者可以在Pull Request上通过评论交流，例如“修正了BUG，可以合并一下吗？”以及“我试着做了这样一个新功能，可以合并一下吗?”等。

通过这个功能，开发者可以轻松更改源代码，并公开更改的细节，然后向仓库提交合并请求。而且，如果请求的更改与项目的初衷相违，也可以选择拒绝合并。

GitHub的Pull Request 不但能轻松查看源代码的前后差别，还可以对指定的一行代码进行评论。

任务管理和BUG报告可以通过Issue进行交互。如果想让特定用户来看，只要用“@用户名”的格式书写，对方便会接到通知（Notification），查看Issue。由于也提供了Wiki功能，开发者可以轻松创建文档，进行公开、共享。Wiki更新的历史记录也在Git中管理。

GitHub Flavored Markdown[^GFM]

还可以这样写

GitHub中可使用的描述方式不止“@用户名”

输入“@组织名”可以让属于该Organization的所有成员收到通知。

输入“@组织名/团队”可以让该团队的所有成员收到通知。

输入“#编号”，会连接到该仓库所对应的Issue编号。

输入“用户名/仓库名#编号”可以连接到指定仓库所对应的Issue编号。

能看到其他团队的软件

将其他团队的仓库添加至Watch中，就可以在News Feed查看该仓库的相关信息。

将隔壁团队正在开发的仓库添加到Watch中，就可以每天查看他们都在开发什么功能。一旦发现有用的功能或者库，可以立刻运用到自己的开发团队。如果能进一步交流，分割出共用的库，从而建立起新的仓库，便成了不同开发者团队间协作的美谈。

社会化编程

世界上任何人都可以比从前更加容易地获得源代码，将其自由更改并加以公开。GitHub的出现为软件开发者的世界带来了真正意义上的“民主”，让所有人都平等地拥有了更改源代码的权利。

GitHub最大特征是“面向人”

以往的仓库托管服务都是以项目为中心，每个项目就是一个信息封闭的世界。虽然能够知道一个仓库的管理者是谁，但这个管理者还在做哪些事，我们就不得而知了。

GitHub除项目外，把注意力集中在人身上。不但能阅览一个人公开的所有源代码，只要查看其控制面板中的News Feed，还能知道他对哪些仓库感兴趣，什么时候做过提交等。

GitHub提供的主要功能

> - Git仓库：公开仓库免费，不公开每月最低7美元
> - Organization：个人用个人账户，公司建议Organization账户。优点在于可以统一管理账户和权限。
> - Issue：将一个任务或问题分配给一个Issue进行追踪和管理的功能。在Git的提交信息中写上Issue的ID（例如“#7”），GitHub就会自动生成从Issue到对应提交的链接。另外，只要按照特定的格式描述提交信息，还可以关闭Issue。
> - Wiki：通过该功能，任何人都能随时对一篇文章进行更改并保存，因此可以多人共同完成一篇文章。该功能常用在开发文档或手册的编写中。Wiki页也是作为Git仓库进行管理的，改版的历史记录会被切实保存下来。由于其支持克隆至本地进行编辑，所以程序员使用时可以不必开启浏览器。
> - Pull Request：开发者向GitHub仓库推送更改或功能添加后，可以通过Pull Request功能向别人的仓库提出申请，请求对方合并。Pull Request送出后，目标仓库的管理者将能够查看Pull Request的内容及其中包含的代码更改。同时，GitHub还提供了对Pull Request和源代码前后差别进行讨论的功能。可以以行为单位对源代码添加评论，让程序员之间高效地交流。

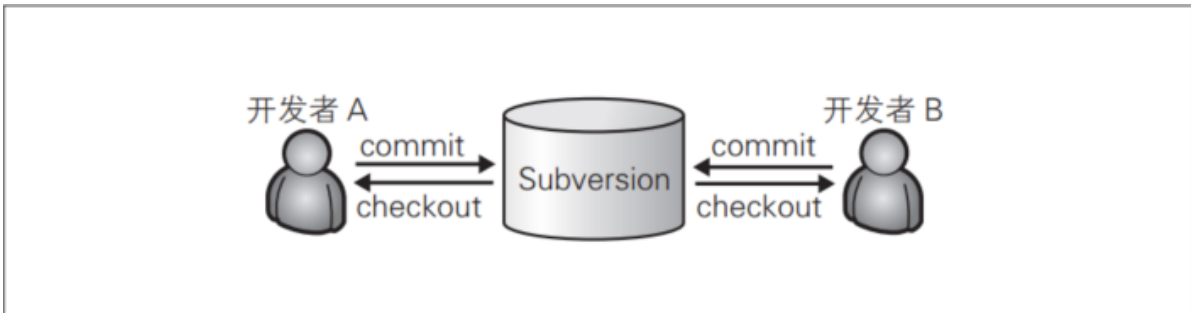
Git

Git属于分散型版本管理系统，是为版本管理而设计的软件。

版本管理就是管理更新的历史记录。它提供了软件开发过程中必不可少的功能，例如记录一款软件添加或更改源代码的过程，回滚到特定阶段，恢复误删除的文件等。

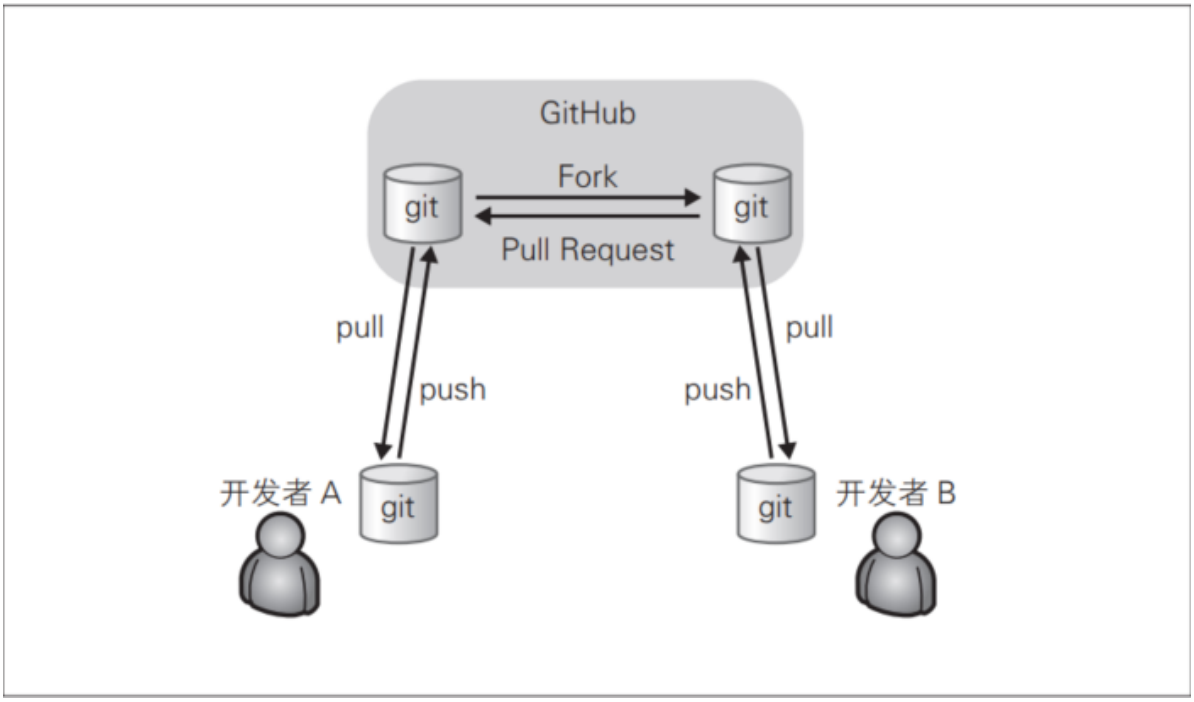
集中型与分散型版本管理系统

以Subversion为代表的集中型，将仓库集中存放在服务器中，所以只存在一个仓库。



集中型便于管理，但一旦开发者所处环境不能连接服务器，就无法获取最新代码。

以Git为代表的分散型，将仓库Fork给每一个用户。Fork就是将GitHub的某个特定仓库复制到自己的账户下。Fork出的仓库与原仓库是两个不同的仓库，开发者可以随意编辑。



分散型拥有多个仓库，相对而言稍显复杂。不过，由于本地的开发环境中就有仓库，所以开发者不必连接远程仓库就可以进行开发。所有仓库之间都可以进行push和pull。即便不通过Github，开发者A也可以直接向开发者B的仓库进行push和pull。

安装

Mac 预装Git

Linux 以软件包（Package）形式提供

Windows 需要装

Windows环境中，最简单快捷的方法是使用msysGit[^http://msysgit.github.io/]

设置环境变量，简单使用的话勾选Use Git Bash only

换行符的处理：GitHub中公开的代码大部分都是以Mac或Linux中的LF（Line Feed）换行。然而，Windows中是以CRLF（Carriage Return + Line Feed）换行的，所以在非对应的编辑器中将不能正常显示。Git可以通过设置自动转换这些换行符。使用Windows环境，选择推荐的“Checkout Windows-style,commit Unix-style line endings”选项，换行符在签出时会自动转换为CRLF，在提交时则会自动转换为LF。

Git Bash：照搬了许多Bash命令

初始设置

- 设置姓名和邮箱地址

```
$ git config --global user.name "Firstname Lastname"
$ git config --global user.email "your_email@example.com"
```

这个命令，会在“~/.gitconfig”中以如下形式输出设置文件。

```
[user]
  name = Firstname Lastname
  email = your_email@example.com
```

想更改这些信息时，可以去C:\Users\Administrator.gitconfig（PS：Administrator不同电脑可能有所区别）找到该文件直接编辑修改。这里设置的姓名和邮箱会用在Git提交日志中被公开，不要使用隐私信息。

- 提高命令输出的可读性

将color.ui设置为auto可以让命令的输出拥有更高的可读性。

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

“~/.gitconfig”中会增加下面一行。

```
[color]
  ui = auto
```

这样一来，各种命令的输出就会变得更容易分辨。

使用Github的前期准备

- 创建Github账户
- 设置头像：GitHub头像是通过Gravatar[^http://cn.gravatar.com/]服务显示的。只要使用创建GitHub账户时注册的邮箱在Gravatar上设置头像，GitHub头像就会变成你设置的样子。
- 设置SSH Key：GitHub上连接已有仓库时的认证，是通过使用了SSH的公开秘钥认证方式进行的。已经创建过秘钥，用现有秘钥进行设置。运行下面的命令创建SSH Key。

```
ssh-keygen -t rsa -C "your_email@example.com"
Generating public/private rsa key pair.
Enter file in wich to save the key
(/Users/your_user_directory/.ssh/id_rsa): 按回车
Enter passphrase (empty for no passphrase): 输入密码
Enter same passphrase again: 再次输入密码
```

然后会出现以下结果。

```
Your identification has been saved in /Users/your_user_directory/.ssh/id_rsa.
Your public key has been saved in /Users/your_user_directory/.ssh/id_rsa.pub.
The key fingerprint is:
fingerprint值 your_email@example.com
The key's randomart image is:
+--[ RSA 2048]----+
|     .+   +      |
|       = o O .   |
略
```

id_rsa是私有秘钥，id_rsa.pub是公开秘钥。

添加公开秘钥

在GitHub中添加公开秘钥，今后就可以用私有秘钥进行认证了。

点击右上角Account Settings，选择SSH Keys，在Title中输入适当的秘钥名称。Key部分粘贴id_rsa.pub文件里的内容。id_rsa.pub的内容可以用如下方法查看。

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa 公开秘钥的内容 your_email@example.com
```

添加成功后，邮箱会接到一封提示“公共秘钥添加完成”的右键。

完成以上设置后，就可以用手中的私人秘钥与GitHub进行认证和通信了。

```
$ ssh -T git@github.com
The authenticity of host 'github.com(207.97.227.239)' can't be established.
RSA key fingerprint is fingerprint值.
Are you sure you want to continue connecting (yes/no)? 输入yes
Hi hirocastest! You've successfully authenticated, but GitHub deos not
provide shell access.
```

创建仓库

点击右上角工具栏里的New repository，创建新仓库。

Description栏里可以设置仓库的说明。不是必须。

Public、Private 选Private可以创建非公开仓库，可以设置访问权限，但非公开仓库是收费的。

Initialize this repository with a README：勾选后，GitHub会自动初始化仓库并设置REDADME文件，让用户可以立刻clone这个仓库。如果想向GitHub添加手中已有的Git仓库，建议不要勾选，直接手动push。

Add .gitignore：下方左侧的下拉菜单非常方便，通过它可以在初始化时自动生成.gitignore文件。这个设定会帮我们把不需要在Git仓库中进行版本管理的文件记录在.gitignore文件中，省去了每次根据框架进行设置的麻烦。下拉菜单中包含了主要的语言及框架，选择今后将要使用的即可。

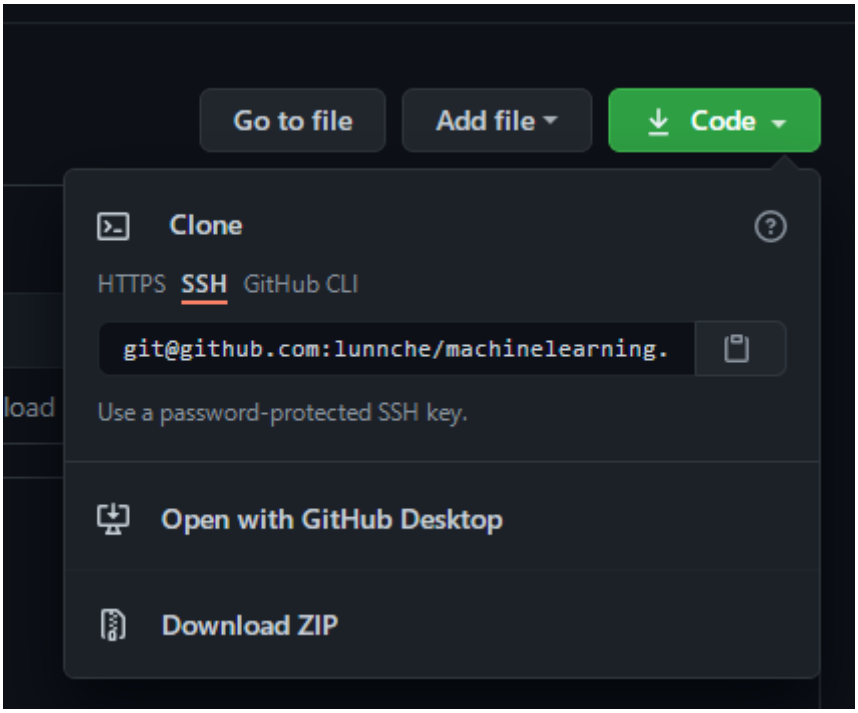
Add a license：右侧下拉菜单可以选择要添加的许可协议文件。如果这个仓库中包含的代码已经确定了许可协议，就在这里进行选择。随后将自动生成包含许可协议内容的LICENSE文件。

README.md：表明本仓库所包含软件的概要、使用流程、许可协议等信息

GitHub Flavored Markdown：在GitHub上进行交流时用到的Issue、评论、Wiki，都可以用Markdown语法表述。

clone已有仓库

将已有仓库clone到身边的开发环境中。clone路径如图：




```
$ git clone git@github.com:hirocastest/Hello-World.git
cloning into 'Hello-World'..
remote:Counting objects: 3,done.
remote:Total 3 (delta 0),reused 0 (delta 0)
receiving objects:100% (3/3),doen.

$ cd Hello-World
```

这里会要求输入GitHub上设置的公开秘钥的密码。认证成功后，仓库便会被clone至仓库名后的目录中。将想要公开的代码提交至这个仓库再push到GitHub的仓库中，代码便会被公开。

- 编写代码

编写一个hello_world.php文件，用来输出“Hello World!”。

```
<?php
  echo "Hello World";
?>
```

由于hello_world.php还没有添加至Git仓库，所以显示为Untracked files。

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       hello_world.php
nothing added to commit but untracked files present (use "git add" to track)
```

- 提交

将hello_world.php提交至仓库。这样，这个文件就进入了版本管理系统的管理之下。今后的更改管理都交由Git进行。

```
$ git add hello_world.php
$ git commit -m "Add hello world script by php"
[master d23b909] Add hello world script by php
 1 file changed, 3 insertions(+)
 create mode 100644 hello_world.php
```

通过git add命令将文件加入暂存区（在Index数据机构中记录文件提交之前的状态），再通过git commit命令提交。

添加成功后，可以通过git log命令查看提交日志。

```
$ git log
commit d23b909caad5d49a281490e6683ce3855-98a5da
Author:hirocastest <hohtsuka@gmail.com>
Date:  Tue May 1 14:36:58 2012 +0900

    Add hello world script by php
略
```

- 进行push

之后只要执行push，GitHub上的仓库就会被更新。

```
$ git push
Counting objects:4,done.
Delta compression using up to 4 threads.
Compressing objects:100%(2/2),done.
Writing objects:100% (3/3),328 bytes,done.
Total 3 (delta 0),reused 0 (delta 0)
To git@github.com:hirocastest/Hello-World.git
   46ff713..d23b909 master -> master
```

这样一来代码就在GitHub上公开了。

在GitHub网站上上传文件不能超过25m，用命令行方式通过git bash上传可以超过25m，据说100m以下就行，据说据说。。。

公开时的许可协议

GitHub上公开的源代码，著作者可选择合适的许可协议，如BSD许可协议、Apache许可协议等，不过大多数软件都使用MIT许可协议。

MIT许可协议具有以下特征：

> 被授权人权利：被授权人有权利使用、复制、修改、合并、出版发行、散步、再授权和/或贩售软件及软件的副本，及授予被供应人同等权利，唯服从以下义务。
>
> 被授权人义务：在软件和软件的所有副本中都必须包含以上版权声明和本许可声明。
>
> 其他重要特性：此许可协议并非属copyleft的自由软件许可协议条款，允许在自由及开放源代码软件或非自由软件（proprietary software）所使用。
>
> MIT的内容可依照程序著作权者的需求更改内容。此亦为MIT与BSD（The BSD license,3-clause BSD license）本质上不同处。
>
> MIT许可协议可与其他许可协议并存。另外，MIT条款也是自由软件基金会（FSF）所认可的自由软件许可协议条款，与GPL兼容。

实际使用时，只需将LICENSE文件加入仓库，并在README.md文件中声明使用了何种许可协议即可。

使用没有声明许可协议的软件时，以防万一最好直接联系著作者。

Git基本操作

- git init——初始化仓库

建立一个目录并初始化仓库。

```
$ mkdir git-tutorial
$ cd git-turtorial
$ git init
Initialized empty Git repository in /Users/hirocaster/github/github-book/git-
tutorial/.git/
```

执行了git init命令的目录下就会生成.git目录。这个.git目录里存储着管理当前目录内容所需的仓库数据。这个目录的内容称为“附属于该仓库的工作树”。文件的编辑等操作在工作树中进行，然后记录到仓库中，以此管理文件的历史快照。如果想将文件恢复到原先的状态，可以从仓库中调取之前的快照，在工作树中打开。

- git status——查看仓库的状态

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

结果显示了我们当前正处在master分支下。还显示了没有可提交的内容。提交（commit），是指“记录工作树中所有文件的当前状态”。

尚没有可提交的内容，就是说当前我们建立的这个仓库中还没有记录任何文件的任何状态。这里，我们建立README.md文件作为管理对象，为第一次提交做准备。

```
$ touch README.md
$ git status
# On branch master
#
# Initial commit
## Untracked files:#   (use "git add <file>.." to include in what will be
committed)#
#       README.md
nothing added to commit but untracked files present (use "git add" to track)
```

可以看到在Untracked files 中显示了README.md文件。类似地，只要对Git的工作树或仓库进行操作，git status命令的显示结果就会发生变化。

- git add —— 向暂存区中添加文件

如果只是用Git仓库的工作树创建了文件，那么该文件并不会被记入Git仓库的版本管理对象当中。因此用git status命令查看README.md文件时，它会显示在Untracked files里。

要想让文件成为Git仓库的管理对象，就需要用git add命令将其加入暂存区（Stage或Index）中。暂存区是提交前的一个临时区域。

```
$ git add README.md
$ git status
# on branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: README.md
#
```

将README.md文件加入暂存区后，git status命令的显示结果发生了变化。README.md文件显示在Changes to be committed中了。

- git commit——保存仓库的历史记录

git commit命令将当前暂存区中的文件实际保存到仓库的历史记录中。通过这些记录，我们就可以在工作树中复原文件。

记述一行提交信息：

```
$ git commit -m "First commit"
[master (root-commit) 9f129ba] First commit
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 README.md
```

-m参数后的“First commit”称作提交信息，是对这个提交的概述。

记述详细提交信息：

想记录更详细的提交信息，不加-m，直接执行git commit命令。执行后编辑器就会启动，显示如下结果。

```
# Please enter the commit message for your changes.Lines starting
# with '#' will be ignored,and an empty message aborts the commit.
# on branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README.md
#
```

在编辑器中记述提交信息的格式如下。

> 第一行：用一行文字简述提交的更改内容
>
> 第二行：空行
>
> 第三行以后：记述更改的原因和详细内容

只要按照上面的格式输入，今后便可以通过确认日志的命令或工具看到这些记录。

- 中止提交

如果在编辑器启动后想中止提交，将提交信息留空并直接关闭编辑器，提交就会中止。

- 查看提交后的状态

执行完git commit命令后再来查看当前状态。

```
$ git status
# On branch master
nothing to commit,working directory clean
```

当前工作树处于刚刚完成提交的最新状态，所以结果显示没有更改。

- git log——查看提交日志

git log命令可以查看以往仓库中提交的日志。包括可以查看什么人在什么时候进行了提交或合并，以及操作前后有怎样的差别。如下可看到刚才的commit命令被记录了。

```
$ git log

commit 9f129bae19b2c82c82fb4e98cde5890e52a6c546922
Author: hirocaster <hohtsuka@gmail.com>
Date:   Sun May 5 16:06:49 2013 +0900

    First commit
```

commit栏旁边显示的“9f129b……”是指向这个提交的哈希值。Git的其他命令中，在指向提交时会用到这个哈希值。

- 只显示提交信息的第一行

如果只想让程序显示第一行简述信息，可以在git log命令后加上 --pretty=short。这样开发人员就能更轻松地把握多个提交。

```
$ git log --pretty=short

commit 9f129bae19b2c82c82fb4e98cde5890e52a6c546922
Author: hirocaster <hohtsuka@gmail.com>

    First commit
```

- 只显示指定目录、文件的日志

在git log命令后加上目录名，便会只显示该目录下的日志。如果加的是文件名，就会只显示与该文件相关的日志。

```
$ git log README.md
```

- 显示文件的改动

如果想查看提交所带来的改动，可加上-p参数，文件的前后差别就会显示在提交信息之后。

```
$ git log -p
```

如，执行下面的命令，就可以只查看README.md文件的提交日志以及提交前后的差别。

```
$ git log -p README.md
```

- git diff——查看更改前后的差别

该命令可以查看工作树、暂存区、最新提交之间的差别。

例如，在刚提交的README.md中写点东西

```
# Git教程
```

上为用Markdown语法写下了一行题目。

- 查看工作树和暂存区的差别

执行git diff命令，查看当前工作树与暂存区的差别。

```
$ git diff

diff --git a/README.md b/README.md
index e69de29..cb5dc9f 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+# Git教程
```

由于尚未用git add命令向暂存区添加任何东西，所以程序只会显示工作树与最新提交状态之间的差别。

“+”号标出的是新添加的行，被删除的行用“-”号标出。

用git add命令将README.md文件加入暂存区。

```
$ git add README.md
```

- 查看工作树和最新提交的差别

如果现在执行git diff命令，由于工作树和暂存区的状态并无差别，结果什么都不会显示。要查看与最新提交的差别，执行以下命令。

```
$ git diff HEAD
diff --git a/README.md b/README.md
index e69de29..cb5dc9f 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+# Git教程
```

不妨养成一个好习惯：在执行git commit命令之前先执行git diff HEAD命令，查看本次提交与上次提交之间有什么差别，等确认完毕后再进行提交。这里的HEAD是指向当前分支中最新一次提交的指针。

由于我们刚刚确认过两个提交之间的差别，所以直接运行git commit命令。

```
$ git commit -m "Add index"
[master fd0cbf0] Add index
1 file changed, 1 insertion(+)
```

保险起见，查看一下提交日志，确认提交是否成功。

```
$ git log
commit fd0cbf0d4a24f84823-594d05cac1be82d33441d
Author: hirocaster <hohtsuka@gmail.com>
Date:   Sun May 5 16:10:15 2013 +0900

    Add index
commit 9f129bae19b2c8u2fb4e98cde5899e52a6436922
Author: hirocaster <hohtsuka@gmail.com>
Date:   Sun May 5 16:06:49 2013 +0900

    First commit
```

成功查到第二个提交

注意：以上新建工作树 git tutorial，在里面一通操作猛如虎后，想git push

报错：

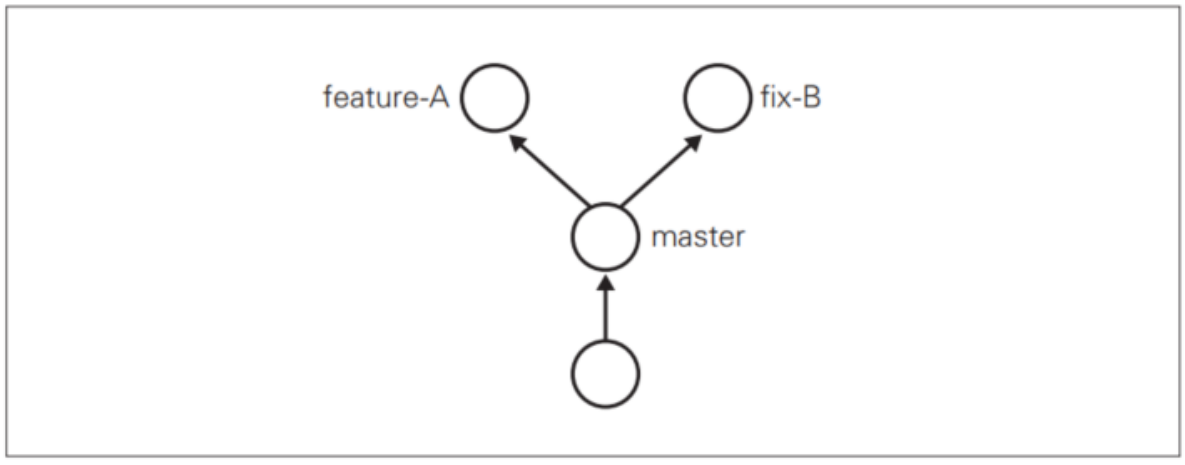
```
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository
using

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

为啥？因为GitHub里没有建新仓库与新工作树绑定，没地方push

分支的操作

并行开发时，同时存在多个最新代码状态。如图，从master分支创建feature-A分支和fix-B分支后，每个分支都拥有自己的最新代码。master 分支是Git默认创建的分支，因此基本上所有开发都是以这个分支为中心进行的。

图 4.1 从 master 分支创建 feature-A 分支和 fix-B 分支



不同分支，可以同时进行完全不同的作业。等该分支作业完成之后再与master分支合并。

- git branch——显示分支一览表

该命令将分支名列表显示，同时可以确认当前所在分支。

```
$ git branch
* master      标*号的为当前所在分支
```

假设我的文件结构/Users/user/machinelearning/git-tutorial，其中只有machinelearning工作树从GitHub克隆了仓库，那么：

```
user@DESKTOP-IOEF463 MINGW64 ~/machinelearning/git-tutorial (master)
$ pwd
/c/Users/user/machinelearning/git-tutorial

user@DESKTOP-IOEF463 MINGW64 ~/machinelearning/git-tutorial (master)
$ git branch
* master

user@DESKTOP-IOEF463 MINGW64 ~/machinelearning/git-tutorial (master)
$ cd

user@DESKTOP-IOEF463 MINGW64 ~ (master)
$ git branch
* master

user@DESKTOP-IOEF463 MINGW64 ~ (master)
$ cd machinelearning

user@DESKTOP-IOEF463 MINGW64 ~/machinelearning (main)
$ git branch
* main
```

如上，只有绑了仓库的才在main上，其他都在master上

- git checkout -b——创建、切换分支
- --- 切换到feature-A分支并进行提交

```
$ git checkout -b feature-A
Switched to a new branch 'feature-A'
```

实际上，连续执行下面两条命令也能收到同样效果。

```
$ git branch feature-A
$ git checkout feature-A
```

上述意为创建feature-A分支，并将当前分支切换为feature-A分支。此时修改代码、执行git add命令并提交的话代码就会提交至feature-A分支。像这样不断对一个分支进行提交操作，称为“培育分支”。

实操下，尝试在README.md文件中添加一行。

```
# Git 教程

 - feature-A
```

添加了feature-A这样一行字母，然后进行提交。

```
$ git add README.md
$ git commit -m "Add feature-A"
[feature-A 8a6c8b9] Add feature-A
 1 file changed, 2 insertions(+)
```

这一行就添加到feature-A分支总了。

- --- 切换到master分支

```
git checkout master
Switched to branch 'master'
```

查看发现，master分支上的README.md文件仍保持原状态，并没有被添加文字。

只要创建多个分支，就可以在不互相影响的情况下进行多个功能的开发。

- --- 切换回上一个分支

```
$ git checkout -
Switched to branch 'feature-A'
```

用“-"(连字符)代替分支名，就可以切换至上一个分支。

特性分支

Git与Subversion（SVN）等集中型版本管理系统个不同个，创建分支时不需要连接中央仓库，所以能够相对轻松地创建分支。因此，当今大部分工作流程中都用到了特性（Topic）分支。

特性分支是集中实现单一特性（主题），除此之外不进行任何作业的分支。

日常开发中，往往会创建数个特性分支，同时在此之外再保留一个随时可以发布软件的稳定分支。稳定分支的角色通常由master分支担当。

之前我们创建了feature-A分支，这一分支主要实现feature-A，除feature-A的实现之外不进行任何作业。

即便在开发过程中发现了BUG，也需要再创建新的分支，在新分支中进行修正。

主干分支

主干分支中并没有开发到一半的代码，可以随时供他人查看。

有时我们需要让这个主干分支总是配置在正式环境中，有时又需要用标签Tag等创建版本信息，同时管理多个版本发布。拥有多个版本发布时，主干分支也有多个。

- git merge——合并分支

若想把feature-A合并到主干分支master中，首先切换到master分支。

```
$ git checkout master
Switched to branch 'master'
```

然后合并feature-A分支。为了在历史记录中明确记录下本次分支合并，需要创建合并提交。因此，在合并时加上

--no--ff参数。

```
$ git merge --no-ff feature-A
```

随后编辑器会启动，用于录入合并提交的信息。

```
Merge branch 'feature-A'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merge an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored,and an empty message aborts
# the commit.
```

默认信息中已经包含了是从feature-A分支合并过来的相关内容，不必修改。输入保存退出后，看到：

```
Merge made by the 'recursive' strategy.
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

这样，feature-A分支的内容就合并到master分支中了。

太神奇了啊，通过Git 分支可以让你的文件夹里的东西在两种状态来回切换啊，仿佛世界线变动/穿梭任意门啊，想喊砸瓦鲁多啊手动无限狗头

- git log --graph——以图表形式查看分支

该命令能很清楚地看到特性分支（feature-A)提交的内容已被合并。除此以外，特性分支的创建以及合并也都清楚明了。

```
$ git log --graph

*   commit 83b0b94268675cb815ac6c8a5bc1965938c15f62
|\  Merge: fd0cbf0 8a6c8b9
| | Author:hirocaster <hohtsuka@gmail.com>
| | Date:  Sun May 5 16:37:57 2013 +0900
| |
| |     Merge branch 'feature-A'
| |
| * commit 8a6c8b97c8962cd44afb69c65f26d6e1a6c088d8
| / Author: hirocaster <hohtsuka@gmail.com>
|   Date:   Sun May 5 16:22:02 2013 +0900
|
|       Add feature-A
|
*   commit fd0cbf0d4a25f747230694d95cac1be72d33441d
|   Author: hirocaster <hohtsuka@gmail.com>
|   Date:   Sun May 5 16:10:15 2013 +0900
|
|      Add index
|
*   commit 9f129bae19b2c82fb4e98cde5890e52a6c546922
```

```
    Author: hirocaster <hohtsuka@gmail.com>
    Date:   Sun May 5 16:06:49 2013 +0900

        First commit
```

以上已学会如何在实现功能后进行提交，累积提交日志作为历史记录，借此不断培育一款软件

更改提交的操作

- git reset——回溯历史版本

Git 的另一特征是可以灵活操作历史版本。

尝试如下操作：回溯历史版本到创建feature-A之前，创建一个名为fix-B的特性分支。

- --- 回溯到创建feature-A分支前：

要让仓库的HEAD、暂存区、当前工作树回溯到指定状态，需要用到git rest --hard命令。只要提供目标时间点的哈希值，就可以完全恢复至该时间点的状态。

```
$ git reset --hard fd0cbf0d4a25f747230594d95cac1be72d33441d
HEAD is now at fd0cbf0 Add index
```

这样就成功回溯到特性分支（feature-A）创建之前的状态。

- --- 创建fix-B分支

```
$ git checkout -b fix-B
Switched to a new branch 'fix-B'
```

在README.md中添加一行

```
# Git教程

  - fix-B
```

然后直接提交README.md文件

```
$ git add README.md

$ git commit -m "Fix B"
[fix-B 4096d9e] Fix B
 1 fiel changed, 2 insertions(+)
```

接下来的目标是：尝试变成这样一种状态：主干分支合并feature-A分支的修改后，又合并了fix-B的修改。

- --- 推进至feature-A分支合并后的状态

首先恢复到feature-A分支合并后的状态。不妨称这一操作为“推进历史”

git log命令智能查到以当前状态为终点的历史日志。所以这里要使用git reflog命令，查看当前仓库的操作日志。

在日志中找出回溯历史之前的哈希值，通过git reset --hard命令恢复到回溯历史前的状态。

首先执行 git reflog命令，查看当前仓库执行过的操作的日志。

```
$ git reflog
4096d9e HEAD@{0}: commit: Fix B
fd0cbf0 HEAD@{1}: checkout: moving from master to fix-B
fd0cbf0 HEAD@{2}: reset: moving to fd0cbf0d4a25f747230694d95cac1be72d33441d
83b0b94 HEAD@{3}: merge feature-A: Merge made by the 'recursive' strategy.
fd0cbf0 HEAD@{4}: checkout: moving from feature-A to master
8a6c8b9 HEAD@{5}: checkout: moving from master to feature-A
fd0cbf0 HEAD@{6}: checkout: moving from feature-A to master
8a6c8b9 HEAD@{7}: commit: Add feature-A
fd0cbf0 HEAD@{8}: checkout: moving from master to feature-A
fd0cbf0 HEAD@{9}: commit: Add index
9f129ba HEAD@{10}: commit (initial): First commit
```

在日志中可以看到commit、checkout、reset、merge、等Git命令的执行记录。只要不进行Git的GC（Garbage Collection，垃圾回收），就可以通过日志随意调取近期的历史状态，就像给时间机器制定一个时间点，在过去未来中自由穿梭一般。即便开发者误执行了Git操作，基本也都可以用git reflog命令恢复到原先的状态。

从上面数第四行表示feature-A特性分支合并后的状态，对应哈希值为83b0b94（哈希值只要输入4位以上就可以执行）。我们将HEAD、暂存区、工作树恢复到这个时间点的状态。

```
$ git checkout master

$ git reset --hard 83b0b94
HEAD is now at 83b0b94 Merge branch 'feature-A'
```

- 消除冲突

现在只要合并 fix-B 分支，就可以得到我们想要的状态。

```
$ git merge --no-ff fix-B
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Recorded preimage for 'README.md'
Automatic merge failed; fix conflicts and then commit the result.
```

这时，系统告诉我们 README.md 文件发生了冲突（ Conflict）。系
统在合并 README.md 文件时， feature-A 分支更改的部分与本次想要合
并的 fix-B 分支更改的部分发生了冲突。
不解决冲突就无法完成合并，所以我们打开 README.md 文件，解
决这个冲突。

- 查看冲突部分并将其解决

用编辑器打开 README.md 文件，就会发现其内容变成了下面这个样子。

```
# Git教程

<<<<<<< HEAD
- feature-A
=======
- fix-B
>>>>>>> fix-B
```

=== 以上的部分是当前 HEAD 的内容，以下的部分是要合并的 fix-B 分支中的内容。我们在编辑器中将其改成想要的样子。

```
# Git教程
- feature-A
- fix-B
```

如上所示，本次修正让 feature-A 与 fix-B 的内容并存于文件之中。但是在实际的软件开发中，往往需要删除其中之一，所以在处理冲突时，务必要仔细分析冲突部分的内容后再行修改。

- 提交解决后的结果

冲突解决后，执行 git add命令与 git commit命令。

```
$ git add README.md
$ git commit -m "Fix conflict"
Recorded resolution for 'README.md'.
[master 6a97e48] Fix conflict
```

由于本次更改解决了冲突，所以提交信息记为 "Fix conflict"。

- git commit --amend——修改提交信息

要修改上一条提交信息，可以使用 git commit --amend命令。
我们将上一条提交信息记为了 "Fix conflict"，但它其实是 fix-B 分支的合并，解决合并时发生的冲突只是过程之一，这样标记实在不妥。
于是，我们要修改这条提交信息。

```
$ git commit --amend
```

执行上面的命令后，编辑器就会启动。

```
Fix conflict

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD^1 <file>..." to unstage)
#
# modified: README.md
#
```

编辑器中显示的内容如上所示，其中包含之前的提交信息。将提交信息的部分修改为 Merge branch 'fix-B'，然后保存文件，关闭编辑器。

```
[master 2e7db6f] Merge branch 'fix-B'
```

随后会显示上面这条结果。现在执行 git log --graph命令，可以看到提交日志中的相应内容也已经被修改。

```
$ git log --graph
* commit 2e7db6fb0b576e9946965ea680e4834ee889c9d8
|\ Merge: 83b0b94 4096d9e
| | Author: hirocaster <hohtsuka@gmail.com>
| | Date: Sun May 5 16:58:27 2013 +0900
| |
| | Merge branch 'fix-B'
| |
| * commit 4096d9e856995a1aafa982aabb52bfc0da656b74
| | Author: hirocaster <hohtsuka@gmail.com>
| | Date: Sun May 5 16:50:31 2013 +0900
| |
| | Fix B
| |
* | commit 83b0b94268675cb715ac6c8a5bc1965938c15f62
|\ \ Merge: fd0cbf0 8a6c8b9
| |/ Author: hirocaster <hohtsuka@gmail.com>
|/| Date: Sun May 5 16:37:57 2013 +0900
| |
| | Merge branch 'feature-A'
| |
| * commit 8a6c8b97c8962cd44afb69c65f26d6e1a6c088d8
|/ Author: hirocaster <hohtsuka@gmail.com>
| Date: Sun May 5 16:22:02 2013 +0900
|
| Add feature-A
|
* commit fd0cbf0d4a25f747230694d95cac1be72d33441d
| Author: hirocaster <hohtsuka@gmail.com>
| Date: Sun May 5 16:10:15 2013 +0900
|
| Add index
|
* commit 9f129bae19b2c82fb4e98cde5890e52a6c546922
Author: hirocaster <hohtsuka@gmail.com>
Date: Sun May 5 16:06:49 2013 +0900
First commit
```

- git rebase -i ——压缩历史

在合并特性分支之前，如果发现已提交的内容中有些许拼写错误等，不妨提交一个修改，然后将这个修改包含到前一个提交之中，压缩成一个历史记录。

- --- 创建feature-C分支

```
$ git checkout -b feature-C
Switched to a new branch 'feature-C'
```

作为 feature-C 的功能实现，我们在 README.md 文件中添加一行文字，并且故意留下拼写错误，以便之后修正。

```
# Git教程
- feature-A
- fix-B
- faeture-C
```

提交这部分内容。这个小小的变更就没必要先执行 git add命令再执行 git commit命令了，我们用 git commit -am命令来一次完成这两步操作。

```
$ git commit -am "Add feature-C"
[feature-C 7a34294] Add feature-C
1 file changed, 1 insertion(+)
```

- --- 修正拼写错误

修正README.md中错误后，

```
$ git diff
diff --git a/README.md b/README.md
index ad19aba..af647fd 100644
--- a/README.md
+++ b/README.md
@@ -2,4 +2,4 @@
   - feature-A
   - fix-B
-  - faeture-C
+  - feature-C
```

然后提交

```
$ git commit -am "Fix typo"
[feature-C 6fba227] Fix typo
1 file changed, 1 insertion(+), 1 deletion(-)
```

错字漏字等失误称作 typo，所以我们将提交信息记为 "Fix typo"。实际上，我们不希望在历史记录中看到这类提交，因为健全的历史记录并不需要它们。如果能在最初提交之前就发现并修正这些错误，也就不会出现这类提交了。

- --- 更改历史

因此，我们来更改历史。将 "Fix typo"修正的内容与之前一次的提交合并，在历史记录中合并为一次完美的提交。为此，我们要用到git rebase命令。

```
$ git rebase -i HEAD~2
```

用上述方式执行 git rebase命令，可以选定当前分支中包含HEAD（最新提交）在内的两个最新历史记录为对象，并在编辑器中打开

```
pick 7a34294 Add feature-C
pick 6fba227 Fix typo
# Rebase 2e7db6f..6fba227 onto 2e7db6f
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
```

```
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

将 6fba227 的 Fix typo 的历史记录压缩到 7a34294 的 Add feature-C里。按照下图所示，将 6fba227 左侧的 pick 部分删除，改写为 fixup。

```
pick 7a34294 Add feature-C
fixup 6fba227 Fix typo
```

保存编辑器里的内容，关闭编辑器。

```
[detached HEAD 5144Oc5] Add feature-C
1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/feature-C.
```

系统显示 rebase 成功。也就是以下面这两个提交作为对象，将 "Fix typo"的内容合并到了上一个提交 "Add feature-C"中，改写成了一个新的提交。
● 7a34294 Add feature-C
● 6fba227 Fix typo
现在再查看提交日志时会发现 Add feature-C 的哈希值已经不是7a34294 了，这证明提交已经被更改。

```
$ git log --graph

* commit 5144Oc55b23fa7fa50aedf20aa43c54138171137
| Author: hirocaster <hohtsuka@gmail.com>
| Date: Sun May 5 17:07:36 2013 +0900
|
| Add feature-C
|
* commit 2e7db6fb0b576e9946965ea680e4834ee889c9d8
|\ Merge: 83b0b94 4096d9e
| | Author: hirocaster <hohtsuka@gmail.com>
| | Date: Sun May 5 16:58:27 2013 +0900
| |
| | Merge branch 'fix-B'
| |
| * commit 4096d9e856995a1aafa982aabb52bfc0da656b74
| | Author: hirocaster <hohtsuka@gmail.com>
| | Date: Sun May 5 16:50:31 2013 +0900
| |
| | Fix B
| |
省略
```

这样一来， Fix typo 就从历史中被抹去，也就相当于 Add feature-C中从来没有出现过拼写错误。这算是一种良性的历史改写。

- --- 合并至master分支

feature-C 分支的使命告一段落，我们将它与 master 分支合并。

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff feature-C
Merge made by the 'recursive' strategy.
README.md | 1 +
1 file changed, 1 insertion(+)
```

master 分支整合了 feature-C 分支。开发进展顺利。

推送至远程仓库

让我们先在 GitHub 上创建一个仓库，并将其设置为本地仓库的远程仓库。

为防止与其他仓库混淆，仓库名与本地仓库保持一致，即 git-tutorial。创建时不要勾选 Initialize this repository with a README 选项。因为一旦勾选该选项， GitHub 一侧的仓库就会自动生成 README 文件，从创建之
初便与本地仓库失去了整合性。虽然到时也可以强制覆盖，但为防止这一情况发生还是建议不要勾选该选项，直接点击 Create repository 创建仓库。

git remote add——添加远程仓库

在 GitHub 上创建的仓库路径为“git@github.com:用户名 /git-tutorial.git”。现在我们用 git remote add 命令将它设置成本地仓库的远程仓库 。

```
$ git remote add origin git@github.com:github-book/git-tutorial.git
```

按照上述格式执行 git remote add命令之后， Git 会自动将git@github.com:github-book/git-tutorial.git远程仓库的名称设置为 origin（标识符）。

git push——推送至远程仓库

- --- 推送至master分支

如果想将当前分支下本地仓库中的内容推送给远程仓库，需要用到git push命令。现在假定我们在 master 分支下进行操作。

```
$ git push -u origin master
Counting objects: 20, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (20/20), 1.60 KiB, done.
Total 20 (delta 3), reused 0 (delta 0)
To git@github.com:github-book/git-tutorial.git
* [new branch] master -> master
Branch master set up to track remote branch master from origin.
```

像这样执行 git push命令，当前分支的内容就会被推送给远程仓库origin 的 master 分支。 -u参数可以在推送的同时，将 origin 仓库的 master 分支设置为本地仓库当前分支的 upstream（上游）。添加了这个参数，将来
运行 git pull命令从远程仓库获取内容时，本地仓库的这个分支就可以直接从 origin 的 master 分支获取内容，省去了另外添加参数的麻烦。执行该操作后，当前本地仓库 master 分支的内容将会被推送到 GitHub 的远程仓库中。在 GitHub 上也可以确认远程 master 分支的内容和本地 master 分支相同。

- --- 推送至master以外的分支

除了 master 分支之外，远程仓库也可以创建其他分支。举个例子，我们在本地仓库中创建 feature-D 分支，并将它以同名形式 push 至远程仓库。

```
$ git checkout -b feature-D
Switched to a new branch 'feature-D'
```

我们在本地仓库中创建了 feature-D 分支，现在将它 push 给远程仓库并保持分支名称不变。

```
$ git push -u origin feature-D
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:github-book/git-tutorial.git
* [new branch] feature-D -> feature-D
Branch feature-D set up to track remote branch feature-D from origin.
```

现在，在远程仓库的 GitHub 页面就可以查看到 feature-D 分支了。

从远程仓库获取

上一节中我们把在 GitHub 上新建的仓库设置成了远程仓库，并向
这个仓库 push 了 feature-D 分支。现在，所有能够访问这个远程仓库的
人都可以获取 feature-D 分支并加以修改。
本节中我们从实际开发者的角度出发，在另一个目录下新建一个本
地仓库，学习从远程仓库获取内容的相关操作。这就相当于我们刚刚执
行过 push 操作的目标仓库又有了另一名新开发者来共同开发。

git clone——获取远程仓库

- --- 获取远程仓库

首先我们换到其他目录下，将 GitHub 上的仓库 clone 到本地。注意不要与之前操作的仓库在同一目录下。

```
$ git clone git@github.com:github-book/git-tutorial.git
Cloning into 'git-tutorial'...
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 20 (delta 3), reused 20 (delta 3)
Receiving objects: 100% (20/20), done.
Resolving deltas: 100% (3/3), done.
$ cd git-tutorial
```

执行 git clone命令后我们会默认处于 master 分支下，同时系统
会自动将 origin 设置成该远程仓库的标识符。也就是说，当前本地仓库
的 master 分支与 GitHub 端远程仓库（ origin）的 master 分支在内容上是
完全相同的。

```
$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/feature-D
remotes/origin/master
```

我们用 git branch -a命令查看当前分支的相关信息。添加 -a
参数可以同时显示本地仓库和远程仓库的分支信息。
结果中显示了 remotes/origin/feature-D，证明我们的远程仓库中已经
有了 feature-D 分支。

- --- 获取远程的feature-D分支

```
$ git checkout -b feature-D origin/feature-D
Branch feature-D set up to track remote branch feature-D from origin.
Switched to a new branch 'feature-D'
```

-b 参数的后面是本地仓库中新建分支的名称。为了便于理解，我
们仍将其命名为 feature-D，让它与远程仓库的对应分支保持同名。新建
分支名称后面是获取来源的分支名称。例子中指定了 origin/feature-D，
就是说以名为 origin 的仓库（这里指 GitHub 端的仓库）的 feature-D 分
支为来源，在本地仓库中创建 feature-D 分支。

- --- 向本地的feature-D分支提交更改

现在假定我们是另一名开发者，要做一个新的提交。在 README.
md 文件中添加一行文字，查看更改。

```
$ git diff
diff --git a/README.md b/README.md
index af647fd..30378c9 100644
--- a/README.md
+++ b/README.md
@@ -3,3 +3,4 @@
- feature-A
- fix-B
- feature-C
+ - feature-D
```

按照之前学过的方式提交即可。

```
$ git commit -am "Add feature-D"
[feature-D ed9721e] Add feature-D
1 file changed, 1 insertion(+)
```

- --- 推送feature-D分支

```
$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 281 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:github-book/git-tutorial.git
ca0f98b..ed9721e feature-D -> feature-D
```

从远程仓库获取 feature-D 分支，在本地仓库中提交更改，再将 feature-D 分支推送回远程仓库，通过这一系列操作，就可以与其他开发者相互合作，共同培育 feature-D 分支，实现某些功能。

git pull——获取最新的远程仓库分支

现在我们放下刚刚操作的目录，回到原先的那个目录下。这边的本地仓库中只创建了 feature-D 分支，并没有在 feature-D 分支中进行任何提交。然而远程仓库的 feature-D 分支中已经有了我们刚刚推送的提交。这时我们就可以使用 git pull 命令，将本地的 feature-D 分支更新到最新状态。当前分支为 feature-D 分支。

```
$ git pull origin feature-D
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:github-book/git-tutorial
* branch feature-D -> FETCH_HEAD
First, rewinding head to replay your work on top of it...
Fast-forwarded feature-D to ed9721e686f8c588e55ec6b8071b669f411486b8.
```

GitHub 端远程仓库中的 feature-D 分支是最新状态，所以本地仓库中的 feature-D 分支就得到了更新。今后只需要像平常一样在本地进行提交再 push 给远程仓库，就可以与其他开发者同时在同一个分支中进行作业，不断给 feature-D 增加新功能。
如果两人同时修改了同一部分的源代码， push 时就很容易发生冲突。所以多名开发者在同一个分支中进行作业时，为减少冲突情况的发生，建议更频繁地进行 push 和 pull 操作。

进阶资料

以上的知识，足以应付日常开发中的大部分操作。高阶操作可参考以下资料

1 Pro Git http://git-scm.com/book/zh/v1 零基础Git学习资料（有中文）

2 LearnGitBranching http://pcottle.github.io/learnGitBranching/ 学习Git基本操作的网站。注重树形结构的学习方式非常适合初学者。（有中文）

3 tryGit http://try.github.io/ 在Web上一边操作一边学习Git的基本功能。（无中文）