



# 看日记学 git

来源：<http://roclinux.cn/>  
整理：sirniu@gmail.com

## 目录

|                                       |                    |
|---------------------------------------|--------------------|
| <a href="#">看日记学 git.....</a>         | <a href="#">1</a>  |
| <a href="#">目录.....</a>               | <a href="#">2</a>  |
| <a href="#">《看日记学 git》之序.....</a>     | <a href="#">4</a>  |
| <a href="#">《看日记学 git》之一.....</a>     | <a href="#">5</a>  |
| <a href="#">《看日记学 git》之二.....</a>     | <a href="#">9</a>  |
| <a href="#">《看日记学 git》之三.....</a>     | <a href="#">11</a> |
| <a href="#">《看日记学 git》之四.....</a>     | <a href="#">15</a> |
| <a href="#">《看日记学 git》之五.....</a>     | <a href="#">19</a> |
| <a href="#">《看日记学 git》之六.....</a>     | <a href="#">20</a> |
| <a href="#">《看日记学 git》之七.....</a>     | <a href="#">28</a> |
| <a href="#">《看日记学 git》之八.....</a>     | <a href="#">29</a> |
| <a href="#">《看日记学 git》之九(总结).....</a> | <a href="#">31</a> |
| <a href="#">《看日记学 git》之十.....</a>     | <a href="#">34</a> |
| <a href="#">《看日记学 git》之十一.....</a>    | <a href="#">37</a> |
| <a href="#">《看日记学 git》之十二.....</a>    | <a href="#">41</a> |
| <a href="#">《看日记学 git》之十三.....</a>    | <a href="#">46</a> |
| <a href="#">《看日记学 git》之十四.....</a>    | <a href="#">47</a> |

|  |                     |
|--|---------------------|
| <a href="#">《看日记学 git》之十五.....</a>     | <a href="#">49</a>  |
| <a href="#">《看日记学 git》之十六.....</a>     | <a href="#">50</a>  |
| <a href="#">《看日记学 git》之十七.....</a>     | <a href="#">52</a>  |
| <a href="#">《看日记学 git》之十八(点睛).....</a> | <a href="#">54</a>  |
| <a href="#">《看日记学 git》之十九.....</a>     | <a href="#">59</a>  |
| <a href="#">《看日记学 git》之二十.....</a>     | <a href="#">63</a>  |
| <a href="#">《看日记学 git》之二十一.....</a>    | <a href="#">66</a>  |
| <a href="#">《看日记学 git》之二十二.....</a>    | <a href="#">82</a>  |
| <a href="#">《看日记学 git》之二十三.....</a>    | <a href="#">84</a>  |
| <a href="#">《看日记学 git》之二十四.....</a>    | <a href="#">91</a>  |
| <a href="#">《看日记学 git》之二十五.....</a>    | <a href="#">96</a>  |
| <a href="#">《看日记学 git》之二十六.....</a>    | <a href="#">102</a> |
| <a href="#">《看日记学 git》之二十七.....</a>    | <a href="#">103</a> |
| <a href="#">《看日记学 git》之二十八.....</a>    | <a href="#">109</a> |
| <a href="#">《看日记学 git》之二十九.....</a>    | <a href="#">110</a> |
| <a href="#">《看日记学 git》之三十.....</a>     | <a href="#">112</a> |
| <a href="#">《看日记学 git》之随笔笔记.....</a>   | <a href="#">113</a> |

## 《看日记学 git》之序

虽然《五分钟系列》才刚刚到第十集，但是由于项目的需要，版本控制系统的学习迫在眉睫~

虽然 rcs、cvs、svn 都是众多项目开发的主力，但是似乎 git 在众多经典项目中扮演的角色越来越重要。

git 相比较于其他 vcs(version control system)的最大优点就是“分布式”。这些特点一定会在后续的《看日记学 git》系列中提到：)

敬请期待...

rocrocket 目前的水平是这样的：曾经用过 cvs 来做项目，目前在使用 svn。完全没有接触过 git。

希望学习 git 的朋友们，可以随着我一起来了解、深入、精通 git。（也有可能 rocrocket 能力有限，永远无法到达精通的地步 不是谦虚^\_^）

也希望能和大家在 roclinux.cn 的《看日记学 git》系列文章中交流，大家多多留言！3x!

## 《看日记学 git》之一

今天是“git 第一天”，和我“从零开始”吧！

1

我用命令 `rpm -qa|grep -i git` 查看一下我的机器是否安装了 git：

```
[rocrocket@wupengchong ~]$ sudo su -  
[root@wupengchong ~]# rpm -qa|grep -i git  
libcapseo-0.2.0-0.1.20080603gita6ec446.fc9.i386  
libcaptury-0.3.0-0.1.20080323gitcca4e3c.fc9.i386  
xorg-x11-drv-nouveau-0.0.10-2.20080408git0991281.fc9.i386  
xorg-x11-drv-digitaledge-1.1.1-1.fc9.i386
```

此处 `grep` 命令使用 `-i` 是表示不区分大小写，这样的话无论是大写小写的 `git/GIT/GiT...` 都不会逃过 `grep` 的法眼。

结果发现了一些似乎使用 git 管理的开发项目，还有一些包含 git 字符的其他包，但就是没发现 git 包。看来我需要安装一个！

2

如果使用 fedora 的 `yum` 来安装 git，应该是相当简单。如果你是 ubuntu，那就更简单了，用 `apt-get` 就可以。如果是 gentoo 的话，我想可以用 `emerge` 吧。源码安装的话，大家可以参考 [git 官网](#) 看看。我当然毫无悬念的选择 `yum`：

```
[root@wupengchong ~]# yum list|grep ^git  
git.i386                                1.5.5.1-1.fc9          updates-newkey  
git-all.i386                           1.5.5.1-1.fc9          updates-newkey
```

|                 |               |                |
|-----------------|---------------|----------------|
| git-arch.i386   | 1.5.5.1-1.fc9 | updates-newkey |
| git-cvs.i386    | 1.5.5.1-1.fc9 | updates-newkey |
| git-daemon.i386 | 1.5.5.1-1.fc9 | updates-newkey |
| git-email.i386  | 1.5.5.1-1.fc9 | updates-newkey |
| git-gui.i386    | 1.5.5.1-1.fc9 | updates-newkey |
| git-svn.i386    | 1.5.5.1-1.fc9 | updates-newkey |
| gitk.i386       | 1.5.5.1-1.fc9 | updates-newkey |
| gitweb.i386     | 1.5.5.1-1.fc9 | updates-newkey |

此处的 grep 命令使用了 ^ 符号，表示后面的字符串需要在每行的最开头位置，于是我们得到了如上的一大串 git 软件包。首先我能肯定的 git.i386 是必然需要安装的，它一定是 git 的主包，但诸如 all、arch、cvs 等等这些包到底是起什么作用呢？我有些迷惑了。于是，不得不求助 baidu 了。

原来：

git 包是 git 的核心程序，它只包括最小的依赖关系，如果只安装 git 包的话，那么一些 git 的外围扩展将无法使用。

git-all 包包括了所有的集成在 git 上的外围扩展功能。安装了这个包，你就万事大吉了！

git-arch 包是为了兼容 arch 仓库的。大家或许听说过 linux 的发行版本 arch，但现在说的 arch 和这个发行版本似乎没有必然关系。git-arch 中的 arch 是 gnu arch，它是一个类似于 cvs、sccs、svn 的版本控制系统。git-arch 包用于支持从 gnu arch 仓库导入到 git 仓库。

git-cvs 包用于支持从 cvs 仓库导入到 git 仓库。

git-daemon 包用于支持用户使用 git:// 形式的命令来访问 git 仓库。

git-email 包支持使用 git 发送邮件。（这个功能用处在哪里呢？我也不太清楚。慢慢研究中... 如果哪位朋友会，在下面留个言）

git-gui 包是一个基于 Tk 的 git 程序，也就是说是有 GUI 的，可以用鼠标点击操作的 git。

git-svn 包用于支持从 svn 仓库导入到 git 仓库。

gitk 包是一个带有 Tcl/Tk GUI 的可以用来浏览 git 仓库历史信息的桌面程序。（如果你对 git 仓库这个词没有概念的话，不要急，后面会说到。现在我们的目的就是要把 git 安装上。）

gitweb 包则是一个成熟的基于 web 的用于 git 仓库管理的 web 程序。

### 3

知道了每一个包的作用后，我们就来安装吧！

```
[root@wupengchong ~]# yum install git-all.i386
```

```
Dependencies Resolved
```

| Package                      | Arch   | Version         | Repository     | Size  |
|------------------------------|--------|-----------------|----------------|-------|
| Installing:                  |        |                 |                |       |
| git-all                      | i386   | 1.5.5.1-1.fc9   | updates-newkey | 9.0 k |
| Installing for dependencies: |        |                 |                |       |
| cvsp                         | i386   | 2.1-6.fc9       | fedora         | 55 k  |
| emacs-common                 | i386   | 1:22.2-5.fc9    | updates-newkey | 19 M  |
| emacs-git                    | i386   | 1.5.5.1-1.fc9   | updates-newkey | 53 k  |
| git                          | i386   | 1.5.5.1-1.fc9   | updates-newkey | 3.5 M |
| git-arch                     | i386   | 1.5.5.1-1.fc9   | updates-newkey | 28 k  |
| git-cvs                      | i386   | 1.5.5.1-1.fc9   | updates-newkey | 69 k  |
| git-email                    | i386   | 1.5.5.1-1.fc9   | updates-newkey | 28 k  |
| git-gui                      | i386   | 1.5.5.1-1.fc9   | updates-newkey | 177 k |
| git-svn                      | i386   | 1.5.5.1-1.fc9   | updates-newkey | 71 k  |
| gitk                         | i386   | 1.5.5.1-1.fc9   | updates-newkey | 79 k  |
| perl-Error                   | noarch | 1:0.17012-2.fc9 | fedora         | 27 k  |

```
perl-Git                i386          1.5.5.1-1.fc9      updates-newkey       18 k
perl-TermReadKey         i386          2.30-6.fc9         fedora                33 k
subversion-perl          i386          1.4.6-7            fedora               893 k
tla                      i386          1.3.5-5.fc9        fedora               353 k
Transaction Summary
=====
Install      16 Package(s)
Update       0 Package(s)
Remove       0 Package(s)
Total download size: 24 M
Is this ok [y/N]:
```

上面列出了要安装的包和依赖，真是不小，24M！点 y 安装！

今天就到这里。今把 git 邀请到家了。先让它休息休息，明天继续~



## 《看日记学 git》之二

昨天将 git 安装到了机器了。今天起和大家一起学习下 git 到底是个什么东西。

查了一下百度百科和维基百科。

git 是一个版本控制系统。

官方的解释是：版本控制(Revision control)是一种软件工程技巧，籍以在开发的过程中，确保由不同人所编辑的同一档案都得到更新。

按我的白话文解释就是：一群志同道合的人身处祖国各地，希望来合作开发一个项目，假设这个项目是使用 c 语言写的（当然用任何语言都可以的）。那么大家怎么合作呢？用信件？效率太低。用邮件，不好实现多人沟通。用 google group 吧，可开发阶段中的源代码没法科学管理。用自建的网站吧，需要人力物力财力来支撑阿。

这个时候版本控制系统就派上用场了。它可以让一个团队里的不同的人在不同地点、不同时间开发和改进同一个项目，并且在大部分的时间里，版本控制系统会聪明的帮你把不同的人在不同地点不同时间修改的代码融合到项目中去。（当然在一些特殊的情况，还是需要人去决定到底哪些代码需要加入到项目中，这个在后面讨论不迟，先让大家对版本控制有一个好印象，呵呵）

知道了版本控制系统的优点之后，下面就要具体实践和体验了。建议你选用的版本控制系统包括：rcs, cvs, svn, git, Mercurial, Bazaar 等等。

当然 git, Mercurial 和 Bazaar 都是属于分布式版本控制系统。

下面是一些网友对于这些版本控制系统评论的只言片语：

- 1) svk 配合 svn 可以实现分布式的版本控制。

- 2) 我是从 SVN 转到 Git 下的。我想 Git 的优势是速度飞快，谁用谁知道！
  - 3) git 的确是最快的，bzi 慢的要死
  - 4) SVN 在 windows 下有 TortoiseSVN
  - 5) git 有 Windows 版本，在 google code 上的项目。<http://code.google.com/p/msysgit/>
  - 6) 大家可以试试国内提供的 git 服务。<http://www.githost.cn>
- 知道了版本控制系统的作用了，就要进入 “使用 git”了。  
今天就到这里。明天继续。

### 《看日记学 git》之三

昨天了解了一些关于版本控制系统的概念，知道了到底版本控制系统是用在哪些方面了。从今天我们开始重点关注 git 这个版本控制系统的应用了。

为了学习 git，我首先会登录到 google 查询 git 的官网地址，是 <http://git.or.cz/>。毕竟 git 官方网站是最权威的学习 git、下载 git 和更新 git 的地方嘛，其中的 Documentation 链接是用于用户入门和进阶的地方。

最先映入眼帘的是这样一句话：git – the stupid content tracker。呵呵 你可能会认为怎么可以把 stupid 用在 git 上呢？其实，stupid 在这里应该解释为“傻瓜式的”，而并非是你想象的那个意思^\_^（知你者，我也）。对的，git 就是一个傻瓜式的内容跟踪器。

其次，你会看到官网对于“学习路线的推介”：新手请浏览 [gittutorial\(7\)](#)，然后是 [Everyday Git](#)（包括了常用命令），接着是“man git-commandname”，cvs 用户需要看 [gitcvs-migration\(7\)](#)，需要更全面的了解 git 请看 [Git User's Manual](#)。

如果你希望用“男人”来获取帮助，那么基本格式是这样：man git-commandname

比如 man git-log 就是获取 git 日志方面的信息；man git-commit 就是获取“提交项目”方面的信息。如果你连 log 或者 commit 都不知道，那也没关系，先 man git 看看都有哪些命令就可以了。

我将按照官网的说明开始我的学习 - - “新手请浏览 [gittutorial\(7\)](#)”。

在 gittutorial(7)里将会涉及到如何导入一个新的项目。（当然如果你只是希望了解如何获取一个项目开发的代码，那么你完全可以只阅读 [The Git User's Manual](#) 的前两章节。）

1

首先你最好把自己介绍给 git 系统，比如自己的姓名阿、email 阿。命令是这样的：

```
git config -global user.name "Your Name"
git config -global user.email "you@example.com"
```

我照例执行：

```
[root@wupengchong ~]# git config -global user.name "roccrocket"
[root@wupengchong ~]# git config -global user.email "wupengchong@gmail.com"
```

2

下面是学习如何导入一个新的 git 项目。

现在我手头已经有了一个伙伴刚刚用 email 传给我的用 c 语言编写的项目（假设只有 main.c 一个文件），而且这个项目的全部代码和资源都放在 roccrocket 目录下，我将用下面的步骤来导入这个项目：（黑体字为重点）

```
[root@wupengchong git-study]# cd roccrocket/
[root@wupengchong roccrocket]# git init
Initialized empty Git repository in .git/
[root@wupengchong roccrocket]# ls -a
.  ..  .git main.c
[root@wupengchong roccrocket]# git add .
[root@wupengchong roccrocket]# git commit
Created initial commit df1d87d: This is the first git project.
1 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 main.c
```

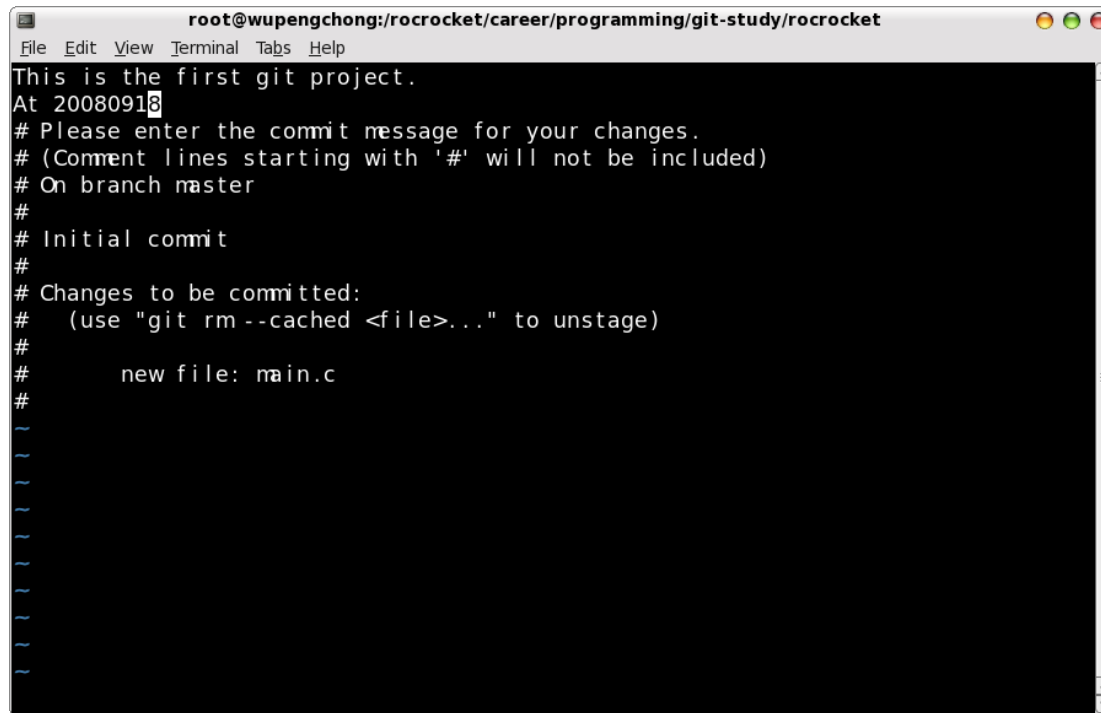
git init 命令用于初始化当前所在目录的这个项目，shell 返回的提示表明已经建立了一个.git 隐藏目录来保存这个项目的进展信息。我们可以用 ls -a 看到它。

git add .这个命令要求 git 给我目前的这个项目制作一个快照 snapshot（快照只是登记留名，快照不等于记录在案，git

管快照叫做索引 index)。快照一般会暂时存储在一个临时存储区域中。

git commit 用于将快照里登记的内容永久写入 git 仓库中，也就是开发者已经想好了要提交自己的开发成果了。

在输入 git commit 并按回车时会转到一个 vi 窗口，要求开发者输入这次提交的版本和开发信息。意思就是说这个项目目前的版本是多少，已经完成了哪些功能，还有哪些功能是需要以后完成的等信息（如果你不介意，当然也可以写上你的感情日记，也不会有人管你，只要你的开发伙伴可以忍受就好）。

A terminal window titled 'root@wupengchong:/rocrocket/career/programming/git-study/rocrocket' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal output shows the 'git commit' process. It starts with 'This is the first git project.' followed by 'At 20080918' where the cursor is. Then it prompts for a commit message: '# Please enter the commit message for your changes. # (Comment lines starting with \'#\' will not be included) # On branch master # # Initial commit # # Changes to be committed: # (use "git rm --cached <file>..." to unstage) # # new file: main.c'. Below this, there are several tilde '~' characters representing the commit message editor. The window has standard OS window controls (minimize, maximize, close) in the top right corner.

```
root@wupengchong:/rocrocket/career/programming/git-study/rocrocket
File Edit View Terminal Tabs Help
This is the first git project.
At 20080918
# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   main.c
#
~
~
~
~
~
~
~
~
~
```

git commit 的日志编辑

## 《看日记学 git》之四

这次我们来研究“改进代码之后怎么提交给 git”。

还记得在[之三](#)中我们项目的 main.c 吧，其中的内容其实就是一个 helloworld：

```
[root@wupengchong rocrocket]# cat -n main.c
1  #include<stdio.h>
2  int main()
3  {
4  printf("hello world!\n");
5  return 0;
6  }
```

这个时候，我来对 main.c 进行一些修改，在 printf 语句前加入一行：

```
printf("Version: 0.01\n");
```

于是程序变成了这样：

```
[root@wupengchong rocrocket]# cat -n main.c
1  #include<stdio.h>
2  int main()
3  {
4  printf("Version: 0.01\n");
5  printf("hello world!\n");
6  return 0;
7  }
```

接下来的两道工序主要是由开发者最后确认一下“自己的修改”：

```
[root@wupengchong rocrocket]# git diff -cached
[root@wupengchong rocrocket]#
```

这个 `git diff -cached` 是用来查看 index file 和仓库之间代码的区别的。由于我们目前只是在 working tree 里做了修改，还没有报告给 index file，所以使用此命令显然会输出空信息。而如果省略 `-cached` 选项的话，就是比较 working tree 和 index file 的区别，由于我们的确在 working tree 里做了修改，所以使用 `git diff` 后会输出修改信息。（可能有些读者不知道 working tree 是什么意思，其实很简单，通俗的说，它就是你的源代码文件，在这个例子里也就是 main.c 文件喽）

```
[root@wupengchong rocrocket]# git diff
diff -git a/main.c b/main.c
index 3a88d8c..e0fe92e 100644
--- a/main.c
+++ b/main.c
@@ -1,6 +1,7 @@
#include<stdio.h>
int main()
{
+printf("Version: 0.01\n");
printf("hello world!\n");
return 0;
}
```

（至于 `git diff` 的输出内容我们现在不必研究太深，只要知道这些信息表示的是修改后和修改前的不同之处就可以了）

使用 `git diff` 了解了不同之后，还可以使用 `git status` 命令来获取整体改动的信息：

```
[root@wupengchong rocrocket]# git status
# On branch master
```



```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   main.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

可以看到提示信息 “changed but not updated”，就是说 git 发现你有已经修改了但还未 git add 的内容。

如果 git 提示说 “Changes to be committed”，那就是表明 git 发现了你已经 git add 但还未 git commit 的内容。

如果 git 提示说 “Untracked files”，那么就是你增加了新文件或者在某个子目录下增加了新文件。

下面该进入提交阶段了。首先运行

```
[root@wupengchong rocrocket]# git add main.c
```

这句是要告诉 git，我已经修改了 main.c 文件，你（指 git）去检查一下。当然，如果你新增加了一个文件，比如 new.c，也需要在这里先执行 git add new.c 告诉 git。

提交我的工作：

```
[root@wupengchong rocrocket]# git commit
Created commit ecf78d1: This is the second version.
1 files changed, 1 insertions(+), 0 deletions(-)
```

至此，我的修改工作完成了，而且也顺利地提交给了 git。还是不放心的来查查：

```
[root@wupengchong rocrocket]# git log
commit ecf78d1b3603d0f5015e8b14bee69870db6459e1
Author: rocrocket <wupengchong@gmail.com>
Date: Thu Sep 18 15:39:47 2008 +0800
This is the second version.
Version 0.02
```

```
commit 3b1e328ad80caebe7fe2f4229e247d2ebe669cd8
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Sep 18 15:32:53 2008 +0800
This is the first git project.
At 20080916
```

用 git log 可以查看开发日志！看到黑体字了吧，Version0.02 就是我刚才在 git commit 时输入的新信息。这已经是我们项目的第二个开发版本了。（成就感油然而生）

总结一下

如果修改了项目代码，先 git add 你修改过的文件，再 git diff 并 git status 查看确认，然后 git commit 提交，然后输入你的开发日志，最后 git log 再次确认。

现在教给你一个偷懒方法，那就是 git commit -a，这个命令可以直接提交所有修改，省去了你 git add 和 git diff 和 git commit 的工序，可谓一条龙服务。

但是，此处有一点应该注意，那就是 git commit -a 无法把新增文件或文件夹加入进来，所以，如果你新增了文件或文件夹，那么就要老老实实的先 git add，再 git commit 喽。[此处非常感谢 [freeren](#) 的提醒]

对了，针对开发日志，要说一句：切记写开发日志的时候，第一行一定要是少于 50 字的开发概括信息，而且第二行务必是空行，第三行开始才可以开始细致描述开发信息。这是因为很多版本服务系统中的 email 机制都会选取 log 中的第一行为邮件题目。（你应该明白了吧:))

ps:我可能在此前也没有太注意关于日志写法的问题，今后也要避免错误:)

===

## 《看日记学 git》之五

接着上次的话题，先谈一下 git-add 命令。

在其他的版本控制系统中，add 命令往往是用在有新文件加入时。而在 git 中，add 命令似乎更简单了，只要是文件有改动，无论是这个文件被修改的还是被新加入的，都可以使用 add 来进行登记。

接着谈 log 命令。

最简单的查询开发日志的方法就是 git log。

但如果你觉得 git log 给出的信息太单薄了，可以使用 **git log -p**，这样 git 不但会给出开发日志，而且会显示每个开发版本的代码区别所在。

## 《看日记学 git》之六

上次的内容比较少，主要是想调节一下大家的心情，总是在大量的文字之间徘徊，身心俱惫。（呵呵，想偷懒有太多理由，其实主要原因是昨天参与了 SFD 北邮站的活动的组织工作，晚上回家实在没有精力钻研 git 了）

这部分主要关注：如何管理分支。

首先要树立这样一种思想，软件开发不是一线到底的，而是有许多曲折的路要走的。我们如何保证走上曲折的道路后能够回归正确的道路呢？可以利用 git 的分支功能。（当然，事情都有两面性，有时候误认为曲折的道路最后发现是正确的道路。呵呵 各种情况，git 也都全想到了）

还是接着我们之前的 main.c 的项目走。我想试着开发一个报时功能加入到 main.c 中，但我不保证这个功能一定能够实现。这个时候可以运行 git branch 命令来开启一个实验分支：

```
[root@wupengchong rocrocket]# git branch experimental
[root@wupengchong rocrocket]#
好了，分支建立完毕！
```

我来查看一下：

```
[root@wupengchong rocrocket]# git branch
Experimental
* master
```

看到喽，直接输入 git branch，不加任何后续参数，就表示让 git 列出所有已存在的分支。前面带“星号”的分支表示当前所在的分支。

好，我要进行报时功能的开发，当然我就要切换到 experimental 分支：

```
[root@wupengchong rocrocket]# git checkout experimental
Switched to branch "experimental"
```

好了，正如一小段英文所示，我们已经利用 checkout 命令成功切换到了 experimental 分支。好，现在就可以修改当前文件来开发我的报时功能了。（^\_^，我现在是走在曲折的路上了）

修改之后的 main.c 如下：

```
[root@wupengchong rocrocket]# cat -n main.c
1  #include<stdio.h>
2  #include<time.h>
3  int main()
4  {
5  time_t mytime;
6  struct tm *mylocaltime;
7  mytime=time(NULL);
8  mylocaltime=localtime(&mytime);
9  printf("Year:%d\n",mylocaltime->tm_year+1900);
10 printf("Month:%d\n",mylocaltime->tm_mon+1);
11 printf("Day:%d\n",mylocaltime->tm_mday);
12 printf("Hour:%d\n",mylocaltime->tm_hour);
13 printf("Min:%d\n",mylocaltime->tm_min);
14 printf("Second:%d\n",mylocaltime->tm_sec);
15 printf("Version: 0.02\n");
16 printf("Hello world!\n");
17 return 0;
```

```
18 }
```

黑体为新加的内容。好了，我的报时功能已经完成了。看来这个分支是可行的：)

运行此程序，运行结果如下：

```
[root@wupengchong rocrocket]# ./a.out
```

```
Year:2008
```

```
Month:9
```

```
Day:21
```

```
Hour:11
```

```
Min:17
```

```
Second:4
```

```
Version: 0.02
```

```
Hello world!
```

OK！运行也很完美。我可以完全的确认“这个走在分支上的项目”完全在正确的道路上。（话语有点拗口，希望你能明白）

下面的任务就是提交程序到分支项目：（注意虽然已经确认了分支的正确性，但还是不能着急报告给“主干道”，而还要先在分支上提交工作）

```
[root@wupengchong rocrocket]# git commit -a
```

```
Created commit 0dff98a: This is a branch.
```

```
1 files changed, 11 insertions(+), 0 deletions(-)
```

然后就可以切换到“主干道”了：

```
[root@wupengchong rocrocket]# git checkout master
```

```
Switched to branch "master"
```

（走在主干道上的你，无论使用 log 或是 status 命令都无法看到刚才在 experimental 分支所进行的工作。）

为了让 git 处理分支的本领展现的淋漓尽致，我们现在在主干道上再做一些改进。我们希望程序在最开始执行的时候输

出一行 “Welcome to roclinux.cn”。这行很简单，在主干道上完成后，main.c 的内容如下：

```
[root@wupengchong rocrocket]# cat -n main.c
1  #include<stdio.h>
2  int main()
3  {
4  printf("Welcome to roclinux.cn\n");
5  printf("Version: 0.02\n");
6  printf("Hello world!\n");
7  return 0;
8  }
```

然后在主干道上使用 git commit -a 提交！

好！我们清理一下思路。在 experimental 分支上有一个增加了报时功能的 main.c，而在主干道上有一个增加了 welcome 的 main.c。它们都进行了 git commit -a 命令。

下面，我们就来合并 “分支” 和 “主干道”（你猜会发生什么？）：

```
[root@wupengchong rocrocket]# git merge experimental
Auto-merged main.c
```

**CONFLICT** (content): Merge conflict in main.c

Automatic merge failed; fix conflicts and then commit the result.

报错了！因为我看到了 conflict 和 failed 这样的字眼。看来主干道上加入的 welcome 和分支干道产生了冲突。我们决定来修改主干道的 welcome 语句到文章的最后部位。主干道的 main.c 此时为：

```
[root@wupengchong rocrocket]# cat main.c
#include<stdio.h>
#include<time.h>
int main()
```

```
{
<<<<<<< HEAD:main.c
printf("Welcome to roclinux.cn\n");
time_t mytime;
struct tm *mylocaltime;
mytime=time(NULL);
mylocaltime=localtime(&mytime);
printf("Year:%d\n",mylocaltime->tm_year+1900);
printf("Month:%d\n",mylocaltime->tm_mon+1);
printf("Day:%d\n",mylocaltime->tm_mday);
printf("Hour:%d\n",mylocaltime->tm_hour);
printf("Min:%d\n",mylocaltime->tm_min);
printf("Second:%d\n",mylocaltime->tm_sec);
>>>>>> experimental:main.c
printf("Version: 0.02\n");
printf("Hello world!\n");
return 0;
}
```

请务必注意代码中三行红粗体字，显而易见这是 git 在告诉我们发生冲突的地点，中间的加黑的 “=====”表示两端冲突代码的分隔。可以看出 git 迷惑之处在于它不知道是把 welcome 这行放在前面还是把报时功能这段放在前面。呵呵 git 正在迷惑中...

现在轮到人类来帮助告诉 git 我们想要什么了，修改这段冲突代码直到你自己满意为止吧。

修改后的 main.c 如下：

```
[root@wupengchong rocrocket]# cat -n main.c
```



```
1  #include<stdio.h>
2  #include<time.h>
3  int main()
4  {
5  printf("Welcome to roclinux.cn\n");
6  time_t mytime;
7  struct tm *mylocaltime;
8  mytime=time(NULL);
9  mylocaltime=localtime(&mytime);
10 printf("Year:%d\n",mylocaltime->tm_year+1900);
11 printf("Month:%d\n",mylocaltime->tm_mon+1);
12 printf("Day:%d\n",mylocaltime->tm_mday);
13 printf("Hour:%d\n",mylocaltime->tm_hour);
14 printf("Min:%d\n",mylocaltime->tm_min);
15 printf("Second:%d\n",mylocaltime->tm_sec);
16 printf("Version: 0.02\n");
17 printf("Hello world!\n");
18 return 0;
19 }
```

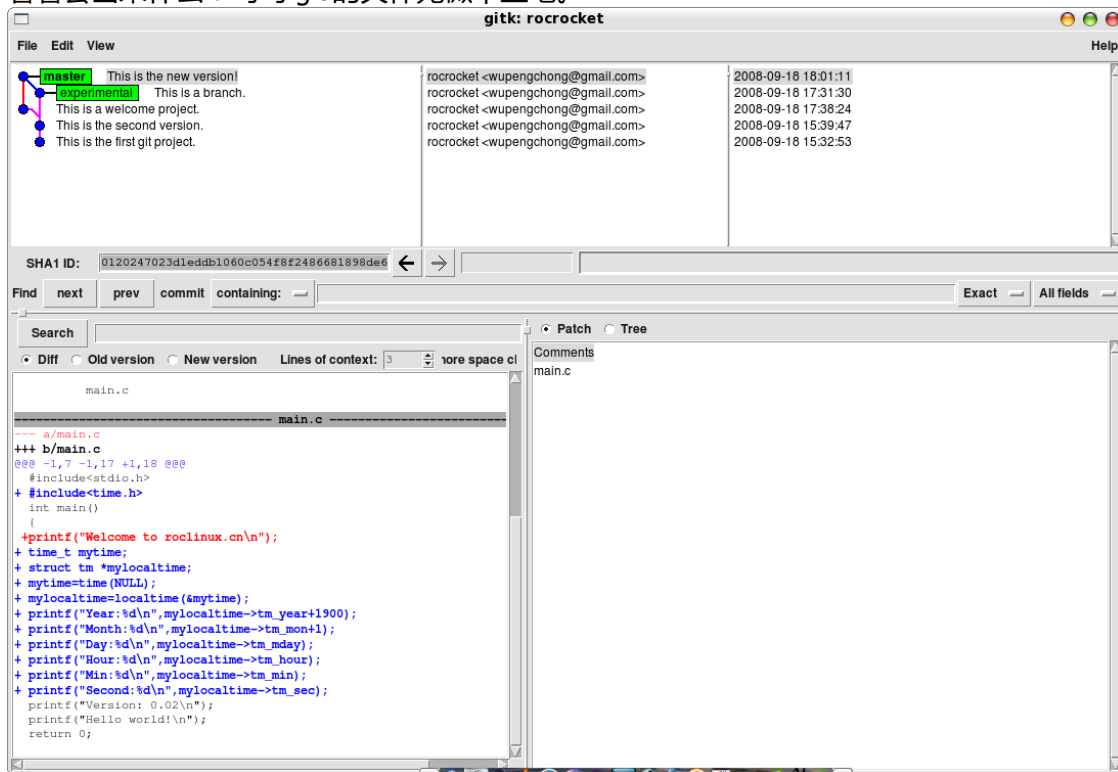
好，解决冲突后再次提交！

```
[root@wupengchong rocrocket]# git commit -a
Created commit 0120247: This is the new version!
```

好了，我们成功的完成了“将分支合并到主干道”的工作。下面轻松一下：

```
[root@wupengchong rocrocket]# gitk
```

看看会出来什么！呵呵 git 的关怀无微不至吧。



gitk 的开发分支的 gui

===

## 《看日记学 git》之七

继续分支话题。

上次学到了 gitk。是不是很爽。爽完之后，分支的任务也就完成了。可以删除分支了：

```
[root@wupengchong rocrocket]# git branch -d experimental
Deleted branch experimental.
```

在这里使用的是小写的-d，表示“在分支已经合并到主干后删除分支”。

如果使用大写的-D的话，则表示“不论如何都删除分支”，-D当然使用在“分支被证明失败”的情况下喽。

总结一下：今天学了 git branch 命令加大 D 和加小 d 的区别。

OK，今天比较项目比较忙，就到这里。：)

===

## 《看日记学 git》之八

这次我们将探讨如何在 git 里合作。

### 1

目前我已经拥有的项目放在/home/roccrocket/git-study/roccrocket 里面。此时我的一个同学小强(xiaoqiang)，他在这台用于开发的机器上也拥有一个登录帐号，他也希望贡献代码。怎么办呢？

小强可以这样获取我的工作成果：

```
xiaoqiang$ git clone /home/roccrocket/git-study/roccrocket myrepo
```

这样小强就克隆了我的工作成果到了他的家目录中的 myrepo 中。OK，小强可以对 myrepo 里的程序进行改进了。

在修改完毕之后，小强使用 git commit -a 来提交他的改进成果。此后，小强告诉我他已经完成了改进工作，让我去查看一下。

### 2

我为了肯定小强的劳动成果（也就是将他的改进合并到项目中来），我是这么干的：

```
cd /home/roccrocket/git-study/roccrocket  
git pull /home/xiaoqiang/myrepo master
```

这句话的意思是：将小强的工作的 master 合并到我的当前的分支上（通常是主干道）。当然，如果这期间，我也修改了项目，那么 git pull 的时候如果报告冲突，我也是需要自己来搞定的。

其实 pull 命令完成了两个动作，首先从远端分支获取 diff 信息，第二个动作就是将改变合并到本地分支中。

ps:由于考虑到后续的学习内容需要对前述内容有非常好的理解和掌握，因此《看日记学 git》系列之九将会复习并总结前八章的内容，大家温故知新以便继续后面 git 难点内容的学习。

## 《看日记学 git》之九(总结)

温故而知新，恩！本次复习之前的内容。

### 1

获得帮助可以使用类似 `man git-****` 的命令格式：

想获得关于 `commit` 命令的帮助，则 `man git-commit`

想获得关于 `pull` 命令的帮助，则 `man git-pull`

想获得关于 `merge` 命令的帮助，则 `man git-merge`

以此类推

### 2

任何人在使用 `git` 之前，都要提交简单的个人信息，以便 `git` 区分不同的提交者身份。

```
#git config --global user.name "your name"
#git config --global user.email yourname@example.com
```

### 3

想新开启一个项目，应该先建立一个目录，例如名为 `myproject`，然后所有的项目开发内容都在此目录下进行。

```
#cd myproject
#git init
#git add .
#git commit //这个步骤会自动进入编辑状态，要求提交者输入有关本次提交的“开发信息”
```

至此，一个新项目就诞生了，第一个开发信息（开发日志）也随之诞生。

### 4

如果改进了项目源代码，并且到了开发者认为“应该再次记录开发信息”的时候，则提交“工作成果”。

```
#git commit -a //这是一个偷懒的命令，相当于git add .; git commit;
```

但是，此处有一点应该注意，那就是 `git commit -a` 无法把新增文件或文件夹加入进来，所以，如果你新增了文件或文件夹，那么就要老老实实的先 `git add .`，再 `git commit` 喽。[此处非常感谢 [freeren](#) 的提醒]

## 5

想检查到目前为止对源码都做了哪些修改（相对于本次工作刚开始之时）：

```
#git diff //这个命令只在git add之前使用有效。如果已经add了，那么此命令输出为空
```

```
#git diff -cached //这个命令在git add之后在git commit之前有效。
```

```
#git status //这个命令在git commit之前有效，表示都有哪些文件发生了改动
```

## 6

想查看自项目开启到现在的所有开发日志

```
#git log
```

```
#git log -p //会输出非常详细的日志内容，包括了每次都做了哪些源码的修改
```

## 7

开启一个试验分支(experimental)，如果分支开发成功则合并到主分支（master），否则放弃该试验分支。

```
#git branch experimental //创建一个试验分支，名称叫experimental
```

```
#git branch //显示当前都有哪些分支，其中标注*为当前所在分支
```

```
#git checkout experimental //转移到experimental分支
```

（省略数小时在此分支上的开发过程）...

如果分支开发成功：

```
#git commit -a //在experimental分支改进完代码之后用commit在此分支中进行提交
```

```
#git checkout master //转移回master分支
```

```
#git merge experimental //经证实分支开发成功，将experimental分支合并到主分支
```

```
#git commit -a //彻底完成此次分支合并，即提交 master 分支
#git branch -d experimental //因为 experimental 分支已提交，所以可安全删除此分支
```

如果分支开发失败：

```
#git checkout master
#git branch -D experimental //由于分支被证明失败，因此使用-D 来放弃并删除该分支
```

8

随时查看图形化分支信息。

```
#gitk
```

9

当合作伙伴 bob 希望改进我 ( rocrocket ) 的工作成果，则：

```
bob$git clone /home/rocrocket/project myrepo //此命令用于克隆我的工作到 bob 的 myrepo 目录下。请注意，此命令有可能会因为/home/rocrocket 的目录权限问题而被拒绝，解决方法是 chmod o+rx /home/rocrocket。
```

( 省略 bob 数小时的开发过程 ) ...

```
bob$git commit -a //bob 提交自己的改进成果到自己的 git 仓库中，并口头告知我 ( rocrocket ) 他已经完成了工作。
```

我如果非常非常信任 bob 的开发能力：

```
$ cd /home/rocrocket/project
$ git pull /home/bob/myrepo //pull 命令的意思是从远端 git 仓库中取出 (git-fetch) 修改的代码，然后合并 (git-merge) 到我 ( rocrocket ) 的项目中去。读者要记住一个小技巧，那就是 "git pull ." 命令，它和 git merge 的功能是一样的，以后完全可以用 git pull . 来代替 git merge 哦！请注意，git-pull 命令有可能会因为/home/bob 的目录权限问题而被拒绝，解决方法是 chmod o+rx /home/bob。
```

如果我不是信任 bob 的开发能力：

```
$ cd /home/rocrocket/project
$ git fetch /home/bob/myrepo master:bobworks //此命令意思是提取出 bob 修改的代码内容，然后放到我 ( rocrocket ) 工作目录下的 bobworks 分支中。之所以要放到分支中，而不是 master 中，就是要我先仔仔细细看看 bob 的开发成果，如果我觉得满意，我再
```



```
merge 到 master 中，如果不满意，我完全可以直接 git branch -D 掉。  
$git whatchanged -p master..bobworks //用来查看 bob 都做了什么  
$git checkout master //切换到 master 分区  
$git pull . bobworks //如果我检查了 bob 的工作后很满意，就可以用 pull 来将 bobworks 分支合并到我的项目中了  
$git branch -D bobworks //如果我检查了 bob 的工作后很不满意，就可以用 -D 来放弃这个分支就可以了
```

过了几天，bob 如果想继续帮助我开发，他需要先同步一下我这几天的工作成果，只要在其当初 clone 的 myrepo 目录下执行 git pull 即可：

```
#git pull //不用加任何参数，因为当初 clone 的时候，git 已经记住了我（roccrocket）的工作目录，它会直接找到我的目录来取。
```

===

## 《看日记学 git》之十

本次将重点关注历史记录查询。

### 1

git 的历史记录是由一些列相关联的“commit”所组成的。每一次“commit”都会有一个唯一的名称。如下黑体字所示：

```
[roccrocket@wupengchong project]$ git log  
commit 5b888402aadd3cd41b3fe8c84a8658da07893b20  
Author: roccrocket <wupengchong@gmail.com>  
Date: Wed Sep 24 13:16:46 2008 +0800  
after pull from roccrocket  
Hello!!!!
```

### 2

我们可以使用 git show 再加上上述的 commit 名称来显式更详细的 commit 信息：

```
[roccrocket@wupengchong project]$ git show 5b888402aadd3cd41b3fe8c84a8658da07893b20
```

你完全可以用一个最短的且唯一的“名称前几个字符”来只待某次 commit：

```
[roccrocket@wupengchong project]$ git show 5b888 //只要能区别与其他名称就足够了
```

使用 git show 加分支名称，亦可以显示分支信息：

```
[roccrocket@wupengchong project]$git show master
```

```
[roccrocket@wupengchong project]$git show experimental
```

使用 HEAD 字段可以代表当前分支的头（也就是最近一次 commit）：

```
[roccrocket@wupengchong project]$git show HEAD
```

每一次 commit 都会有“parent commit”，可以使用^表示 parent：

```
[roccrocket@wupengchong project]$git show HEAD^ //查看 HEAD 的父母的信息
```

```
[roccrocket@wupengchong project]$git show HEAD^^ //查看 HEAD 的父母的父母的信息
```

```
[roccrocket@wupengchong project]$git show HEAD~4 //查看 HEAD 上溯 4 代的信息
```

要注意的是 git-merge 是会产生双父母的，这种情况这样处理：

```
[roccrocket@wupengchong project]$git show HEAD^1 //查看 HEAD 的第一个父母
```

```
[roccrocket@wupengchong project]$git show HEAD^2 //查看 HEAD 的第二个父母
```

3

你可以给复杂名称起个别名：

```
[roccrocket@wupengchong project]$ git tag V3 5b888 //以后可以用 V3 来代替复杂的名称 (5b888...)
```

```
[roccrocket@wupengchong project]$ git show V3
```

```
[roccrocket@wupengchong project]$ git branch stable V3 //建立一个基于 V3 的分支
```

4

可以用 git grep 帮助我们搜索：

```
[roccrocket@wupengchong project]$ git grep "print" V3 //在 V3 中搜索所有的包含 print 的行
```

```
[roccrocket@wupengchong project]$ git grep "print" //在所有的历史记录中搜索包含 print 的行
```

## 5

## 定位具体的历史记录

```
[roccrocket@wupengchong project]$ git log V3..V7 //显示 V3 之后直至 V7 的所有历史记录
[roccrocket@wupengchong project]$ git log V3.. //显示所有 V3 之后的历史记录。注意<since>..<until>中任何一个被省略
都将被默认设置为 HEAD。所以如果使用..<until>的话, git log 在大部分情况下会输出空的。
[roccrocket@wupengchong project]$ git log -since="2 weeks ago" //显示 2 周前到现在的所有历史记录。具体语法可查询
git-ref-parse 命令的帮助文件。
[roccrocket@wupengchong project]$ git log stable..experimental //将显示在 experimental 分支但不在 stable 分支的
历史记录
[roccrocket@wupengchong project]$ git log experimental..stable //将显示在 stable 分支但不在 experimental 分支的
历史记录
```

## 6

## 你最喜欢的 gitk 也可以定位具体的历史记录：

```
[roccrocket@wupengchong project]$ gitk -since="2 weeks ago" drivers/ //将在 GUI 中显示自 2 周前到现在为止的且位于
drivers 目录下的分支记录信息
```

===

到目前为止，我们针对最基本的 git 的操作已经学习完毕了。经过了这么多天的“看日记”，想必大家对这种学习形式有自己的认识和评价，希望大家写在下面的留言区，我会不断改进，为大家奉献更好的专题系列。：)

## 《看日记学 git》之十一

接下来的内容是以“之一~之十”的内容为基础的，内容将围绕对象数据库和索引文件展开，这是为了更好的理解 git 的工作机制和体系结构。

系列之十一将讲解“对象数据库 ( object database )”

### 1

为了讲清楚 object database 这个概念，我们先建立一个 git 仓库：

```
$ mkdir test-project
$ cd test-project
$ git init
$ echo 'Hi,roccrocket'>file.txt
$ git add .
$ git commit -a -m "initial commit" //此处的-m选项表示“后面的参数是本次提交的历史记录”
$ echo 'Hi,roccrocket!>file.txt
$ git commit -a -m "add emphasis"
$ git log //使用log命令查看一下
Created commit d85012f: add emphasis
1 files changed, 1 insertions(+), 1 deletions(-)
[roccrocket@wupengchong test-project]$ git log
commit d85012f4d17a774c7286f4d7d022f022d3b90073
Author: roccrocket <wupengchong@gmail.com>
```

```
Date:   Fri Sep 26 10:58:15 2008 +0800
add emphasis
commit 241e0a4d2a5644f92737b7fba8b9eb19dcb0c345
Author: rocrocket <wupengchong@gmail.com>
Date:   Fri Sep 26 10:57:13 2008 +0800
initial commit
```

你可曾想过到底这些跟在 commit 字符串后面的一大长串（共 40 位）十六进制数字是干什么用的？？

通过前十期的学习，我们知道这 40 位十六进制数是用来“标识一次 commit”的名称。其实，这 40 位十六进制数是一个 SHA1 哈希数（Secure Hash Algorithm），它可以保证每次 commit 生成的名称都是唯一的且可以永远有效的。

## 2

执行下面命令：

```
[rocrocket@wupengchong test-project]$ git cat-file -t 241e //cat-file 命令中-t 选项表示列出相应 ID 的对象类型；
241e 是刚才 commit 后得出的 SHA1 码
commit //可以看到此 ID 对应的对象类型为一次 commit
[rocrocket@wupengchong test-project]$ git cat-file commit 241e //此处的 commit 表示要查询的是一个对象类型为
commit 的对象，后面给出此对象的 ID
tree 9a327d5e3aa818b98ddaa7b5b369f5deb47dc9f6
author rocrocket <wupengchong@gmail.com> 1222397833 +0800
committer rocrocket <wupengchong@gmail.com> 1222397833 +0800
initial commit
```

你应该可以注意到命令输出结果中包含了 tree，tree 的 ID 表示了一个 BLOB 对象（二进制对象），此对象对应着一个文件或另一个 tree。你可以使用 ls-tree 命令来查询关于这个 tree 的更详细信息：

```
[rocrocket@wupengchong test-project]$ git ls-tree 9a327
100644 blob 7d4e0fa616551318405e8309817bcfecb7224cff file.txt
```

我们可以看到 9a327 这棵树上包括了一个 file.txt 文件，其 ID 为 7d4e0f

```
[roccrocket@wupengchong test-project]$ git cat-file -t 7d4e0f
blob
[roccrocket@wupengchong test-project]$ git cat-file blob 7d4e0f
Hi,roccrocket
```

可以看到 7d4e0f 对应的对象的类型是 blob，而其内容就是 “Hi,roccrocket”

### 3

所有的对象信息都存储在 .git/objects/ 目录下，使用 find 命令查看一下：

```
./objects
./objects/pack
./objects/24
./objects/24/1e0a4d2a5644f92737b7fba8b9eb19dcb0c345
./objects/fb
./objects/fb/4ec0f35525992eda56021f161f85f637837404
./objects/d8
./objects/d8/5012f4d17a774c7286f4d7d022f022d3b90073
./objects/7d
./objects/7d/4e0fa616551318405e8309817bcfecb7224cff
./objects/9a
./objects/9a/327d5e3aa818b98ddaa7b5b369f5deb47dc9f6
./objects/info
./objects/60
./objects/60/be00dabc677b196938758ef41457cc17b77b70
```

这些对象都是被压缩过的，其类型可以是 blob，tree，commit 或 tag。其中包括了对象长度、对象类型和对象的内容。

**4**

在.git目录下的HEAD文件比较特殊，查看.git/HEAD文件：

```
[rocrocket@wupengchong .git]$ cat HEAD
ref: refs/heads/master
[rocrocket@wupengchong .git]$ cd ..
[rocrocket@wupengchong test-project]$ git branch
* master
```

可以看到HEAD文件指示出当前所在的分支名称是master。其实更引起我们兴趣的是HEAD文件中所指示的这个路径“refs/heads/master”，我们随着HEAD的思路走下去：

```
[rocrocket@wupengchong test-project]$ cat .git/refs/heads/master
d85012f4d17a774c7286f4d7d022f022d3b90073
```

这个文件的内容给出了一个对象ID，继续寻根溯源：

```
[rocrocket@wupengchong test-project]$ git cat-file -t d850
commit
[rocrocket@wupengchong test-project]$ git cat-file commit d850
tree fb4ec0f35525992eda56021f161f85f637837404
parent 241e0a4d2a5644f92737b7fba8b9eb19dcb0c345
author rocrocket <wupengchong@gmail.com> 1222397895 +0800
committer rocrocket <wupengchong@gmail.com> 1222397895 +0800
add emphasis
```

谜底揭开了。HEAD所指向的原来是最后一次commit的信息，而且经测试可知parent指的是上一次commit的信息。

**5**

到目前为止，我们应该已经知道了对象数据库是如何管理历史记录的了：

commit对象会指向一个tree对象，即在历史记录中当前结点的tree目录的镜像；也会指向父母（parent）commit，

这是为了和之前的 commit 建立关联。

tree 对象用于显示一个目录的状态，tree 对象中包含了 blob 对象和子目录对象。

blob 对象包含的是文件的数据。

每个分支的 HEAD 会存储在.git/refs/heads 中。同时，当前所在分支的头部会存储在.git/HEAD 中。

## 《看日记学 git》之十二

这次重点讲解索引文件（index file）的作用。

我们在提交工作时，使用最多的命令就是 git commit -a 了，但是这个将提交你所做的所有工作。其实，如果你了解 commit 的工作机制，你会知道我们可以自定义提交哪些部分到哪些工作树中，其实自由度很大的。

1

还记得之前我们建立的 test-project 工作目录么。我们继续在这个目录下演示讲解。

```
[rocrocket@wupengchong test-project]$ echo "hello world,again">>file.txt
```

这次，我们不急着执行 commit 命令，而是先用 git diff 看看差别情况：

```
[rocrocket@wupengchong test-project]$ git diff
diff -git a/file.txt b/file.txt
index 60be00d..a559610 100644
- a/file.txt
+++ b/file.txt
@@ -1,2 @@
Hi,rocrocket!
+hello world,again
```



好了，我们可以看到 git 报告了我们刚才所做的修改。下面我们来 add 一下，然后再 git diff，看看 diff 有什么变化呢：

```
[rocrocket@wupengchong test-project]$ git add file.txt
```

```
[rocrocket@wupengchong test-project]$ git diff
```

```
[rocrocket@wupengchong test-project]$
```

大家可以看到在 add 之后的 git diff 的输出竟然为空了，但是此时我们尚未执行 commit 阿。如果这个时候你执行 git diff HEAD，你仍然会看到修改报告：

```
[rocrocket@wupengchong test-project]$ git diff HEAD
diff -git a/file.txt b/file.txt
index 60be00d..a559610 100644
- a/file.txt
+++ b/file.txt
@@ -1,2 @@
Hi,roccrocket!
+hello world,again
```

这就说明了一个问题：git diff 不是在和 HEAD 比，而是和另外一个“神秘”内容在比，而这个“神秘”内容就是“索引文件”！

索引文件（index file）就是.git/index 文件，它是二进制形式的文件。我们可以用 ls-files 来检查它的内容。

```
[rocrocket@wupengchong test-project]$ git ls-files -stage //一定要记住，此命令是用于查看 index file 的！！
```

```
100644 a55961026a22bdd4e938dcc90a4a83823a81f776 0      file.txt
```

```
[rocrocket@wupengchong test-project]$ git cat-file -t a5596
```

```
blob
```

```
[rocrocket@wupengchong test-project]$ git cat-file blob a5596
```

```
Hi,roccrocket!
```

```
hello world,again
```

很明显，我们可以看到其内容已经是改进后的代码了，怪不得 git-diff 会输出空呢！

我们的结论就是 git add 的作用就是创建一个 blob 文件来记录最新的修改代码，并且在 index file 里添加一个到此 blob 的链接。

2

如果在 git-diff 后面加上参数 HEAD，则 git-diff 会显示当前工作目录和最近一次提交之间的代码区别。

```
[rocrocket@wupengchong test-project]$ echo 'again?'>>file.txt
[rocrocket@wupengchong test-project]$ git diff HEAD
diff -git a/file.txt b/file.txt
index 60be00d..dfb67dc 100644
- a/file.txt
+++ b/file.txt
@@ -1 +1,3 @@
Hi,roccrocket!
+hello world,again
+again?
```

如果使用参数-cached，则会比较 index file 和最近一次提交之间的代码区别。

```
[rocrocket@wupengchong test-project]$ git diff -cached
diff -git a/file.txt b/file.txt
index 60be00d..a559610 100644
- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
Hi,roccrocket!
+hello world,again
```

按我的认识，更清楚且通俗的解释就是：git 维护的代码分成三部分，“当前工作目录” <-> “index file” <-> git 仓库。git commit 会将 index file 中的改变写到 git 仓库；git add 会将“当前工作目录”的改变写到“index file”；“commit -a”则会直接将“当前工作目录”的改动同时写到“index file”和“git 仓库”。

而 git diff 的使用稍微有些复杂，大家可以看看 Lee.MaRS 对于这个命令非常精彩的分析（蓝色字部分）：（在此非常感谢 Lee.MaRS）

将 Current working directory 记为 (1)

将 Index file 记为 (2)

将 Git repository 记为 (3)

他们之间的提交层次关系是 (1) -> (2) -> (3)

git add 完成的是 (1) -> (2)

git commit 完成的是 (2) -> (3)

git commit -a 两者的直接结合

从时间上看，可以认为 (1) 是最新的代码，(2) 比较旧，(3) 更旧

按时间排序就是 (1) <- (2) <- (3)

git diff 得到的是从 (2) 到 (1) 的变化

git diff -cached 得到的是从 (3) 到 (2) 的变化

git diff HEAD 得到的是从 (3) 到 (1) 的变化

3

status 命令会显示当前状态的一个简单总结：

```
[roccrocket@wupengchong test-project]$ git status
# On branch master
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
#
#   modified:   file.txt
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
#   modified:   file.txt
#
```

上面两行黑体字中的 **Changes to be committed** 表示在 index 和 commit 的区别状况。而 **Changed but not updated** 表示当前目录和 index 的区别状况。

ps: 下一步我们将学习 [Everyday Git](#).

===

## 《看日记学 git》之十三

在完成了入门学习（之一~之十）以及进阶学习（之十一~之十二）后，我们要深入学习各种 git 命令了。  
我们即将学习的内容将包括如下几个部分（扮演不同角色）：

- 1) Basic Repository Command：每个操纵 git 的人都需要掌握
- 2) Individual Developer (Standalone)：独立开发者需要掌握
- 3) Individual Developer (Participant)：合作开发者需要掌握
- 4) Integrator：项目汇总人需要掌握
- 5) Repository Administration：仓库管理员需要掌握

ps:从明天开始，将分别深入介绍。

===

## 《看日记学 git》之十四

我们从 basic repository 讲起。

1

我们都会使用这些命令来完成最基本的工作：

git-init 或 git-clone 来创建 git 仓库

git-fsck 来检查仓库的错误

git-gc 用来完成一些琐碎的工作，比如重组或删减。

2

我们来专门研究下 git-init 命令。

git-init 命令主要用于创建一个空的 git 仓库或者重新初始化一个已存在的仓库。

git-init 命令会创建一个.git 隐藏目录。在.git 目录下，会创建 objects, refs/heads, refs/tags 和模板文件，与此同时，指向 master 分支的 HEAD 文件也会被创建。

如果在环境变量中定义了\$GIT\_DIR 变量，那么 git-init 就不会在./.git 目录下创建这些文件，而是在\$GIT\_DIR 所指定的目录下创建。

环境变量\$GIT\_OBJECT\_DIRECTORY 是用来指示对象存储目录的路径的。如果设定了这个变量，则 SHA1 目录将被创建在哪里。否则，SHA1 目录会默认创建在\$GIT\_DIR/objects 中。

如果你突发奇想地在一个已经存在的仓库（也就是目录）中运行了 git-init，也是没有任何问题的，git-init（很识相地）不会把当前目录下的 git 仓库信息覆盖掉。其实，常常在需要载入新加模板时才需要“在一个已存在的仓库中运行 git-

init”。

有一点需要注意，那就是 git-init 和 git-init-db 是完全一样的。git 刚发布的时候是使用 git-init-db 的，但后来由于 init 所担任的工作越来越多，所以 db 这个词已经不能涵盖 init 的工作范围了，所以就去掉 db，只剩下 git-init 了。

===

## 《看日记学 git》之十五

git-clone 命令，主要负责克隆 git 仓库到另一个新目录中。

我们可以用如下的表示法来定义远程的一个 git 仓库：

```
rsync://host.xz/path/to/repo.git/  
http://host.xz/path/to/repo.git/  
https://host.xz/path/to/repo.git/  
git://host.xz/path/to/repo.git/  
git://host.xz/~user/path/to/repo.git/  
ssh://[user@]host.xz[:port]/path/to/repo.git/  
ssh://[user@]host.xz/path/to/repo.git/  
ssh://[user@]host.xz/~user/path/to/repo.git/  
ssh://[user@]host.xz/~ /path/to/repo.git
```

如果 git 仓库在本地，那么可以这样表示：

```
/path/to/repo.git/  
file:///path/to/repo.git/
```



## 《看日记学 git》之十六

本次我们将从一个独立开发者的视角来学习 git。

### 1

独立开发者的最大特点就是他们不需要和其他人来交换补丁，而且只在一个独立的固定的 git 仓库中工作。

下面这些命令将可以帮助你完成日常工作：

```
git-show-branch：可以显示你当前所在的分支以及提交记录。
git-log：显示提交日志
git-checkout 或者 git-branch：用于切换和创建分支
git-add：用于将修改内容加入到 index 文件中
git-diff 和 git-status：用于显示开发者所做的修改
git-commit：用于提交当前修改到 git 仓库。
git-reset 和 git-checkout：用于撤销某些修改
git-merge：用于合并两个分支
git-rebase：用于维护 topic 分支（此处我也不太懂，等完成 git 学习后转过头来会关注此问题）
git-tag：用于标记标签。
```

### 2

我们来举一个例子，模拟一下独立开发者使用 git 的情形。

首先使用一个 tarball 作为整个项目的初始点。

```
$ tar -xzvf mypro.tat.gz
$ cd mypro
```

```
$ git-init
$ git add .
$ git commit -m "important of mypro source tree."
$ git tag v2.43 //给这个 commit 起了一个简单的名字 v2.43
```

下面我们建立分支并继续开发：

```
$ git checkout -b alsa-audio //-b 用于建立一个新的分支，分支名称为 alsa-audio，并且转移到此分支
...(开发/编译/测试)
$ git checkout - curses/ux_audio_oss.c //用于取消对 curses/ux_audio_oss.c 文件的修改
$ git add curses/ux_audio_alsa.c //如果你在这一阶段的开发过程中增加了新文件，那么你应该用 git-add 告知 git 仓库，当然，
如果你只是修改或删除，那么使用 git-commit -a 就可以让 git 查觉到了。
...(开发/编译/测试)
$ git diff HEAD //查看一下我们即将 commit 的内容
$ git commit -a -s //提交
...(开发/编译/测试)
$ git reset -soft HEAD^ //回复到上一次 commit 的代码。-soft 选项表示不改动 index file 和 working tree 中的内容
...(开发/编译/测试)
$ git diff ORIG_HEAD //look at the changes since the premature commit we took back ( 此句不太懂 )
$ git commit -a -c ORIG_HEAD //重新提交，-c ORIG_HEAD 表示使用原有的提交信息
$ git checkout master
$ git merge alsa-audio
$ git log -since='3 days ago'
$ git log v2.43.. curses/ //查看自从 v2.43 以来的 curses 目录下的代码变化信息
```

===

## 《看日记学 git》之十七

上次是从一个独立开发者的视角来学习 git，这次要从一个合作开发者的角度来看问题，这个更加实用。

1

作为项目开发者的一员，学会和队友交流是一件很重要的事情。因此，我们不仅要掌握独立开发者所掌握的命令，还要掌握用于沟通的 git 命令。

git-clone：复制别人的 git 仓库到本地

git-pull 和 git-fetch：保持和别人的 git 仓库的同步更新

git-push：共享方式。等同于 CVS 方式下的共享。

git-format-patch：利用邮件形式提交补丁。等同于内核开发方式。

2

情景模拟：

```
$git clone git://git.kernel.org/pub/scm/.../torvalds/linux-2.6 my2.6
```

```
$cd my2.6
```

```
(开发...编译...测试...)
```

```
$git commit -a -s //-s 选项用于在 commit 信息后加上结束标志
```

```
$git format-patch origin //从本地分支生成 patch，用于 email 提交
```

```
$git pull //从 origin 取出更新并合并到当前分支
```

```
$git log -p ORIG_HEAD.. arch/i386 include/asm-i386
```

```
$git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git ALL //从特定 git 仓库取出变更并合并。
```

```
$git reset -hard ORIG_HEAD //恢复到上一次的内容  
$git gc //用垃圾回收机制清除由于 reset 而造成的垃圾代码  
$git fetch -tags //从 origin 取出 tags 并存储到 .git/refs/tags
```

3

学习到这里，一定会感觉有些迷糊了，原因是对于 git 的各种命令还不是很熟悉，所以无法和实际应用相结合。计划在下次进行一次大总结。之后再进入 “Integrator 和 admin 的讲解” 。:)

===

## 《看日记学 git》之十八(点睛)

这篇文章将总结之前文章的知识点，  
但并不会地毯式总结，简单的命令在这里不会重复，  
我们总结的将是疑难知识点。

我提炼出来的需要解决的疑难问题包括：

- 1 commit 和 commit -a 的区别
- 2 log -p 的中-p 的作用
- 3 merge 的用法及参数用法
- 4 fetch 的用法
- 5 pull 的用法
- 6 commit 信息详解
- 7 HEAD 的含义及相关用法，ORIG\_HEAD 的用法以及其他常量的使用方法
- 8 tag 的用法

下面我们就一一总结上述问题。

===

1

commit 和 commit -a 的区别

commit -a 相当于：

第一步：自动地 add 所有改动的代码，使得所有的开发代码都列于 index file 中

第二步：自动地删除那些在 index file 中但不在工作树中的文件

第三步：执行 commit 命令来提交

2

log -p 的中-p 的作用

git log：显示 commit 日志

git log -p：不仅显示 commit 日志，而且同时显示每次 commit 的代码改变。

3

merge 的用法及参数用法

git-merge 主要用于将两个或两个以上的开发分支进行合并。

**git merge branchname 用于将 branchname 分支合并到当前分支中。**(如果合并发生冲突，需要自己解决冲突)

当 merge 命令自身无法解决冲突的时候，它会将工作树置于一种特殊的状态，并且给用户提供冲突信息，以期用户可以自己解决这些问题。当然在这个时候，未发生冲突的代码已经被 git merge 登记在了 index file 里了。如果你这个时候使用 git diff，显示出来的只是发生冲突的代码信息。

在你解决了冲突之前，发生冲突的文件会一直在 index file 中被标记出来。这个时候，如果你使用 git commit 提交的话，git 会提示：filename.txt needs merge

在发生冲突的时候，如果你使用 git status 命令，那么会显示出发生冲突的具体信息。

在你解决了冲突之后，你可以使用如下步骤来提交：

第一步：git add filename.txt

第二步：git commit

如果你希望撤销一个分支到 merge 前的状态，那么使用如下命令：

\$ git reset --hard HEAD --no-hard 表示将 working tree 和 index file 都撤销到以前状态

在这先偷偷的告诉你, `-soft` 表示只撤销 commit, 而保留 working tree 和 index file 的信息, `-mixed` 会撤销 commit 和 index file, 只保留 working tree 的信息。OK, 如果你能记住 `-hard`、`-mixed` 和 `-soft` 的区别, 那最好, 如果记不住, 也不用自责啦, 以后还会讲到。

#### 4

fetch 的用法

git-fetch 用于从另一个 repository 下载 objects 和 refs。

命令格式为: `git fetch <options> <repository> <refspec>...`

其中 `<repository>` 表示远端的仓库路径。

其中 `<refspec>` 的标准格式应该为 `<src>:<dst>`, `<src>` 表示源的分支, 如果 `<dst>` 不为空, 则表示本地的分支; 如果为空, 则使用当前分支。

`git fetch /home/bob/myrepo master:bobworks` : 用于从 bob 的工作目录的 master 分支下载 objects 和 refs 到本地的 bobworks 分支中。

#### 5

pull 的用法

git-pull 的作用就是从另一个 repository 取出内容并合并到另一个 repository 中。

git pull 是 git fetch 和 git merge 命令的一个组合。

`git pull /home/bob/myrepo` 这个命令的意思是从此目录中取出内容并合并到当前分支中。

`git pull .` 就相当于 `git merge`。

#### 6

commit 信息详解

你使用 `git log` 可以看到每一次 commit 的信息, 大约如下格式:

```
[roccrocket@wupengchong project]$ git log
```

```
commit 5b888402aadd3cd41b3fe8c84a8658da07893b20
Author: rocrocket <wupengchong@gmail.com>
Date:   Wed Sep 24 13:16:46 2008 +0800
after pull from rocrocket
Hello!!!!
```

可以看到黑体部分为本次 commit 的 ID 号，你可以根据这个号码，使用 git show 来显示这次 commit 的更详细的信息，包括了提交时间、修改内容、git diff 信息等等。

## 7

### 常量的使用方法

HEAD：表示最近一次的 commit。

MERGE\_HEAD：如果是 merge 产生的 commit，那么它表示除 HEAD 之外的另一个父母分支。

FETCH\_HEAD：使用 git-fetch 获得的 object 和 ref 的信息都存储在这里，这些信息是为日后 git-merge 准备的。

HEAD^：表示 HEAD 父母的信息

HEAD^^：表示 HEAD 父母的父母的信息

HEAD~4：表示 HEAD 上溯四代的信息

HEAD^1：表示 HEAD 的第一个父母的信息

HEAD^2：表示 HEAD 的第二个父母的信息

COMMIT\_EDITMSG：最后一次 commit 时的提交信息。

## 8

### tag 的用法

主要作用是给某次 commit 起一个好记的名字：

```
[rocrocket@wupengchong project]$ git tag V3 5b888 //以后可以用 V3 来代替复杂的名称(5b888...)
```

```
[rocrocket@wupengchong project]$ git show V3
```



ps:我敢肯定，读者一定还有很多迷惑之处，但是到目前为止，作为一个初级用户，你简单地使用 git 已经没有任何问题了。我们如果想继续提高，需要的是对基本命令的更加熟练使用，以及对 git 原理的了解和把握。

===

## 《看日记学 git》之十九

今天，我们重点来探讨一下 git-diff 的用法。

我们知道在 git 提交环节，存在三大部分：working tree, index file, commit

**这三大部分中：**

working tree 就是你所工作所在的目录，每当你在代码中进行了修改，working tree 的状态就改变了。

index file 是索引文件，它是连接 working tree 和 commit 的桥梁，每当我们使用 git-add 命令来登记后，index file 的内容就改变了，此时 index file 就和 working tree 同步了。

commit 是最后的阶段，只有 commit 了，我们的代码才真正进入了 git 仓库。我们使用 git-commit 就是将 index file 里的内容提交到 commit 中。

**总结一下：**

git diff 是查看 working tree 与 index file 的差别的。

git diff -cached 是查看 index file 与 commit 的差别的。

git diff HEAD 是查看 working tree 和 commit 的差别的。（你一定没有忘记，HEAD 代表的是最近的一次 commit 的信息）

**为了更加清晰的阐释这个关系，我们来给出一个实例。**

首先在目录 test-git 下建立一个 c 文件，内容如下：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
```

```
int main(int argc, char *argv[])
{
    printf("hello.\n");
    printf("he was a student.\n");
    return 0;
}
```

然后 git init, git add . , git commit;

之后你将源代码修改为：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
int main(int argc, char *argv[])
{
    printf("hello.\n");
    printf("he was a student.\n");
    printf("he was born in finland.\n");
    return 0;
}
```

此时你 git add . , 但不用执行 git commit 命令。

然后你再将源代码改为：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
int main(int argc, char *argv[])
{
    printf("hello.\n");
    printf("he was a student.\n");
}
```

```
printf("he was born in finland.\n");  
printf("he is very clever!\n");  
return 0;  
}
```

这个时候，你执行如下三个命令，仔细查看，我相信你会发现它们三个的区别的！

```
$ git diff  
$ git diff -cached  
$ git diff HEAD
```

讲到这里，你基本上对 git diff 命令有了比较深入的了解了，现在你再使用 git status 看看输出结果，样子大概是这样：

```
[rocrocket@wupengchong test-git]$ git status  
# On branch master  
#  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified:   main.c  
#  
# Changed but not updated:  
#   (use "git add <file>..." to update what will be committed)  
#  
#       modified:   main.c  
#
```

很明显可以知道：

**Changes to be committed** 表示已经存在于 index file 里，但尚未提交。

**Changed but not updated** 表示在 working tree 已经做修改，但还没有使用 git add 登记到 index file 里。

## 《看日记学 git》之二十

今天，我们细细研究 git branch 和 git checkout branchname 这两个命令。

我们都知道 git branch 用于显示分支列表并且标识当前所在分支。这个命令比较简单。

git branch branchname 用于建立一个名字叫 branchname 的分支。但是你想过这个分支会根据什么来建么？是根据 working tree？还是根据 index file？还是 commit 呢？（不卖关子，答案告诉你，是...commit！）

下面给大家来个情景重现，来证实一下：git branch branchname 就是根据 commit 来建立的新分支。

首先在目录 test-git 下建立一个 c 文件，内容如下：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
int main(int argc,char *argv[])
{
printf("hello.\n");
printf("he was a student.\n");
return 0;
}
```

然后 git init, git add . , git commit;

之后你将源代码修改为：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
int main(int argc,char *argv[])
```

```
{  
printf("hello.\n");  
printf("he was a student.\n");  
printf("he was born in finland.\n");  
return 0;  
}
```

此时你 `git add`，但不用执行 `git commit` 命令。

然后你再将源代码改为：

```
[rocrocket@wupengchong test-git]$ cat main.c  
#include<stdio.h>  
int main(int argc,char *argv[])  
{  
printf("hello.\n");  
printf("he was a student.\n");  
printf("he was born in finland.\n");  
printf("he is very clever!\n");  
return 0;  
}
```

好了，这个时候，我们就营造了一个环境，在这个 git 仓库里，working tree、index file 和 commit 都是不同的。（可以用《看日记学 git》之十九中介绍的方法来证实这一点。）

此时，我们运行创建分支的命令：

```
[rocrocket@wupengchong test-git]$ git branch experimental
```

此命令用来创建一个 experimental 分支，然后用 `git checkout experimental` 转移到新分支中。

这时，你使用 `cat main.c` 可以发现，其内容是：

```
[rocrocket@wupengchong test-git]$ cat main.c
#include<stdio.h>
int main(int argc,char *argv[])
{
printf("hello.\n");
printf("he was a student.\n");
return 0;
}
```

看！这个内容显然是 master 分支中的 commit 里的。

OK，今天就强调这一个概念：) 记住哦，以后很有用的。

ps：如果你学有余力，我再告诉你一个信息。在你 git branch 一个新分支后，在目录.git/refs/heads 目录下会多出一个新的文件，对应于新分支的名称，用来记录新分支下对应的“最后一次 commit 的信息”。

ps:如果你学有余力，我还要告诉你一个信息。当你 git branch 一个新分支并 checkout 转移到这个新分支后，.git 目录下的 HEAD 文件会相应的改变，它的内容将对应着新分支下“最后一次 commit 的信息”。

===

## 《看日记学 git》之二十一

我们已经完全掌握了建立新分支的方法，但是建立了新分支，总要利用起来吧，通俗点说，就是“领导要管得住自己的下属”。

恩，读完本文，你将完全掌握如何利用别人的分支的啦！

我们会以实例的方式，来将分支技术娓娓道来:D 请大家跟上思路。

首先，先概括说一下整体思路：

- 1) 创建一个主分支，名称当然是默认的 master，写一个小程序 roc.c。
- 2) 之后，创建新分支，命名为 wukong（呵呵，你没看错，就是悟空），在 wukong 分支改进小程序 roc.c 代码并 commit。
- 3) 在 wukong 的基础上建立 bajie（对，八戒），再改进代码 roc.c，并在 bajie 分支 commit。
- 4) 切换到 wukong，悟空修改自己的代码，并故意造成和 bajie 的代码冲突。
- 5) 由悟空来操作，将 bajie 合并到 wukong 分支。（需要解决代码冲突）
- 6) 由我来操作，将 wukong 合并到 master 分支。
- 7) 实验完毕。可以删除两个分支了。

好了，大体思路就这么简单，也基本上可以涵盖所有的分支操作情况了。下面，我们就一步一步来开始实例体验吧！

**第一步：创建一个主分支，名称当然是默认的 master，写一个小程序。**

```
[roccrocket@wupengchong ~]$ mkdir git21
[roccrocket@wupengchong ~]$ cd git21/
[roccrocket@wupengchong git21]$ ls
[roccrocket@wupengchong git21]$ vi roc.c
[roccrocket@wupengchong git21]$ ls
```



```
roc.c
[rocrocket@wupengchong git21]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
5      return 0;
6  }
[rocrocket@wupengchong git21]$
```

然后进行初始化工作：

```
[rocrocket@wupengchong git21]$ git init
Initialized empty Git repository in .git/
[rocrocket@wupengchong git21]$ git add *
[rocrocket@wupengchong git21]$ git commit
Created initial commit d5d44dc: master:001
1 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 roc.c
[rocrocket@wupengchong git21]$
```

最后是例行检查：

```
[rocrocket@wupengchong git21]$ git diff
[rocrocket@wupengchong git21]$ git diff -cached
[rocrocket@wupengchong git21]$ git diff HEAD
[rocrocket@wupengchong git21]$ git log
commit d5d44dc938a263bdbbc1c9cb06de24c27adebc7
Author: roccrocket <wupengchong@gmail.com>
```

```
Date: Thu Oct 9 18:50:39 2008 +0800
master:001
[rocrocket@wupengchong git21]$ git status
# On branch master
nothing to commit (working directory clean)
[rocrocket@wupengchong git21]$ git branch
* master
[rocrocket@wupengchong git21]$
```

第二步：之后，创建新分支，命名为 wukong（你没看错，就是孙猴子，呵呵），在 wukong 分支改进小程序 roc.c 代码并 commit

```
[rocrocket@wupengchong git21]$ git branch wukong
[rocrocket@wupengchong git21]$ git branch
* master
wukong
[rocrocket@wupengchong git21]$ git checkout wukong
Switched to branch "wukong"
[rocrocket@wupengchong git21]$ git branch
master
* wukong
[rocrocket@wupengchong git21]$ cat .git/HEAD
ref: refs/heads/wukong
[rocrocket@wupengchong git21]$ ls
roc.c
[rocrocket@wupengchong git21]$ vi roc.c
```

编辑后 roc.c 的代码如下：

```
[rocrocket@wupengchong git21]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
5      printf("He is a youngster.\n");
6      return 0;
7  }
[rocrocket@wupengchong git21]$
```

下面进行 add 和例行检查：

```
[rocrocket@wupengchong git21]$ git add roc.c
[rocrocket@wupengchong git21]$ git diff
[rocrocket@wupengchong git21]$ git diff -cached
diff -git a/roc.c b/roc.c
index 66cb6f0..2fe0f42 100644
- a/roc.c
+++ b/roc.c
@@ -2,5 +2,6 @@
int main()
{
printf("Please guess who he is.\n");
+    printf("He is a youngster.\n");
return 0;
}
[rocrocket@wupengchong git21]$
```

下面在 wukong 分支提交此修改：

```
[rocrocket@wupengchong git21]$ git commit
Created commit lac0798: wukong:001
1 files changed, 1 insertions(+), 0 deletions(-)
[rocrocket@wupengchong git21]$ git log
commit lac0798784828577ecb808b03165facfb5bef3e3
Author: roccrocket <wupengchong@gmail.com>
Date: Thu Oct 9 19:07:57 2008 +0800
wukong:001
commit d5d44dc938a263bdbbc1c9cb06de24c27adebc7
Author: roccrocket <wupengchong@gmail.com>
Date: Thu Oct 9 18:50:39 2008 +0800
master:001
[rocrocket@wupengchong git21]$
```

好了，到此时，我们来查看一下 master 主分支和 wukong 分支的代码分别是什么情况：

```
[rocrocket@wupengchong git21]$ git branch
master
* wukong
[rocrocket@wupengchong git21]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Please guess who he is.\n");
printf("He is a youngster.\n");
return 0;
```

```
}
[rocrocket@wupengchong git21]$ git checkout master
Switched to branch "master"
[rocrocket@wupengchong git21]$ git branch
* master
wukong
[rocrocket@wupengchong git21]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Please guess who he is.\n");
return 0;
}
[rocrocket@wupengchong git21]$
```

显而易见，此时两个分支的代码是不一样的。:)继续我们的实例体验。

3 在 wukong 的基础上建立 bajie（对，八戒），再改进代码 roc.c，并在 bajie 分支 commit。

```
[rocrocket@wupengchong git21]$ git checkout wukong
Switched to branch "wukong"
[rocrocket@wupengchong git21]$ git branch
master
* wukong
[rocrocket@wupengchong git21]$ git branch bajie
[rocrocket@wupengchong git21]$ git branch
bajie
master
```

```
* wukong
[rocrocket@wupengchong git21]$ git checkout bajie
Switched to branch "bajie"
[rocrocket@wupengchong git21]$ git branch
* bajie
master
wukong
[rocrocket@wupengchong git21]$ vi roc.c
```

悟空成功建立了 bajie 分支，并 checkout 到 bajie 分支。

然后就是编辑 bajie 分支的 roc.c 了：

```
[rocrocket@wupengchong git21]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
5      printf("He is a youngster.\n");
6      printf("He is a male.\n");
7      return 0;
8  }
[rocrocket@wupengchong git21]$
```

可以看到黑体字，我们加了一句话：He is a male.（他是一个男性）。

好，八戒编辑完代码，可以提交了！

```
[rocrocket@wupengchong git21]$ git diff
diff -git a/roc.c b/roc.c
```

```
index 2fe0f42..05aaf78 100644
- a/roc.c
+++ b/roc.c
@@ -3,5 +3,6 @@ int main()
{
printf("Please guess who he is.\n");
printf("He is a youngster.\n");
+    printf("He is a male.\n");
return 0;
}
[rocrocket@wupengchong git21]$ git add roc.c
[rocrocket@wupengchong git21]$ git commit
Created commit 5a0f9ee: bajie:001
1 files changed, 1 insertions(+), 0 deletions(-)
[rocrocket@wupengchong git21]$ git log
commit 5a0f9ee71fe9ffe259b9b6cf842dd69baa95b307
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Oct 9 19:16:31 2008 +0800
bajie:001
commit 1ac0798784828577ecb808b03165facfb5bef3e3
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Oct 9 19:07:57 2008 +0800
wukong:001
commit d5d44dc938a263bdbbc1c9cb06de24c27adebc7
Author: rocrocket <wupengchong@gmail.com>
```

```
Date: Thu Oct 9 18:50:39 2008 +0800
master:001
[roccrocket@wupengchong git21]$ git branch
* bajie
master
wukong
[roccrocket@wupengchong git21]$
```

好了，已经在 bajie 分支提交成功了。（你应该注意到了，对于分支操作来说，不仅代码是向下继承，连 log 也是向下继承的。不过，log 和代码一样是不会自动向上复制的，只有 merge 了之后才可以看到下层分支的 Log 和代码）

4 切换到 wukong，悟空修改自己的代码，并故意造成和 bajie 的代码冲突。

```
[roccrocket@wupengchong git21]$ git branch
* bajie
master
wukong
[roccrocket@wupengchong git21]$ git checkout wukong
Switched to branch "wukong"
[roccrocket@wupengchong git21]$ git branch
bajie
master
* wukong
[roccrocket@wupengchong git21]$ vi roc.c
```

好了，成功切换到 wukong，开始在 roc.c 中加入代码吧！

```
[roccrocket@wupengchong git21]$ cat -n roc.c
1 #include<stdio.h>
```



```
2 int main()
3 {
4     printf("Please guess who he is.\n");
5     printf("He is a youngster.\n");
6     printf("He is a female.\n");
7     return 0;
8 }
[rocrocket@wupengchong git21]$
```

可以看到，呵呵，wukong 写了一句 He is a female. 很明显，这句话是捣乱的，也就是要故意引起冲突。因为在 bajie 分支里已经有 He is a male.这句话了。好，下面好戏来了，看看怎么解决这个冲突吧！

朋友要注意一点：要合并两个分支的话，两个分支都必须是 commit 的状态。否则 merge 可会报错的哦！来，看看 merge 报错时的样子：

```
[rocrocket@wupengchong git21]$ git merge bajie
Updating 1ac0798..5a0f9ee
roc.c: needs update
error: Entry 'roc.c' not uptodate. Cannot merge.
```

就是这样，告诉你 roc.c 需要升级，不能 merge。

下面，来将 wukong 修改的内容 commit 吧！

```
[rocrocket@wupengchong git21]$ git diff
diff -git a/roc.c b/roc.c
index 2fe0f42..b8066c1 100644
- a/roc.c
+++ b/roc.c
@@ -3,5 +3,6 @@ int main()
```

```
{
printf("Please guess who he is.\n");
printf("He is a youngster.\n");
+     printf("He is a female.\n");
return 0;
}
[rocrocket@wupengchong git21]$ git commit -a
Created commit 4f6ba4e: wukong:002
1 files changed, 1 insertions(+), 0 deletions(-)
[rocrocket@wupengchong git21]$ git log
commit 4f6ba4e96db6110405f615a5ea5d3119faf64d45
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Oct 9 23:35:31 2008 +0800
wukong:002
commit 1ac0798784828577ecb808b03165facfb5bef3e3
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Oct 9 19:07:57 2008 +0800
wukong:001
commit d5d44dc938a263bdbbc1c9cb06de24c27adebc7
Author: rocrocket <wupengchong@gmail.com>
Date:   Thu Oct 9 18:50:39 2008 +0800
master:001
```

5 由悟空来操作，将 bajie 合并到 wukong 分支。（需要解决代码冲突）

```
[rocrocket@wupengchong git21]$ git merge bajie
Auto-merged roc.c
```

```
CONFLICT (content): Merge conflict in roc.c
Automatic merge failed; fix conflicts and then commit the result.
[rocrocket@wupengchong git21]$
```

可以看到提示，是在抱怨 bajie 的代码和当前分支（wukong）的代码有冲突（conflict）。只好来解决一下了。  
可以看到在 merge 报冲突之后 roc.c 的内容是这样：

```
[rocrocket@wupengchong git21]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Please guess who he is.\n");
printf("He is a youngster.\n");
<<<<<< HEAD:roc.c
printf("He is a female.\n");
=====
printf("He is a male.\n");
>>>>>> bajie:roc.c
return 0;
}
```

显然是在等待人脑去判断代码是选择 male 还是 female。呵呵 那我们就来选择正确的，删除错误的，当然是选择 male 了！修改后的代码如下：

```
[rocrocket@wupengchong git21]$ cat roc.c
#include<stdio.h>
int main()
{
```

```
printf("Please guess who he is.\n");  
printf("He is a youngster.\n");  
printf("He is a male.\n");  
return 0;  
}
```

好，我们解决了冲突，这回可以放心提交了！（注意这回就不用 merge 了哦，直接 commit 就可以了！）

```
[rocrocket@wupengchong git21]$ git diff  
diff -cc roc.c  
index b8066c1,05aaf78..0000000  
- a/roc.c  
+++ b/roc.c  
[rocrocket@wupengchong git21]$ git add roc.c  
[rocrocket@wupengchong git21]$ git diff  
[rocrocket@wupengchong git21]$ git commit  
Created commit e1ca782: wukong:003  
[rocrocket@wupengchong git21]$
```

好了，commit 成功！

查看一下日志，即 git log 一下：

```
[rocrocket@wupengchong git21]$ git log  
commit e1ca782d95176e83d1013361ba88d0b4c2f51f50  
Merge: 4f6ba4e... 5a0f9ee...  
Author: roccrocket <wupengchong@gmail.com>  
Date: Thu Oct 9 23:43:06 2008 +0800  
wukong:003  
Merge branch 'bajie' into wukong
```

```
Conflicts:
roc.c
commit 4f6ba4e96db6110405f615a5ea5d3119faf64d45
Author: rocrocket <wupengchong@gmail.com>
Date: Thu Oct 9 23:35:31 2008 +0800
wukong:002
commit 5a0f9ee71fe9ffe259b9b6cf842dd69baa95b307
Author: rocrocket <wupengchong@gmail.com>
Date: Thu Oct 9 19:16:31 2008 +0800
bajie:001
commit 1ac0798784828577ecb808b03165facfb5bef3e3
Author: rocrocket <wupengchong@gmail.com>
Date: Thu Oct 9 19:07:57 2008 +0800
wukong:001
commit d5d44dc938a263bdbbc1c9cb06de24c27adebc7
Author: rocrocket <wupengchong@gmail.com>
Date: Thu Oct 9 18:50:39 2008 +0800
master:001
```

看到了么，log 好多阿！这回你可以推出一个结论，那就是：git merge 的不仅仅是代码，而且还包括 log！呵呵

**6 由我操作，将 wukong 合并到 master 分支。**

该到衣锦还乡的时候了！

悟空和八戒折腾了半天，是该回头找 master（师傅）了。

先切换到 master：

```
[rocrocket@wupengchong git21]$ git checkout master
```

```
Switched to branch "master"  
[rocrocket@wupengchong git21]$ git branch  
bajie  
* master  
wukong  
[rocrocket@wupengchong git21]$
```

然后就是 merge 了！

```
[rocrocket@wupengchong git21]$ git merge wukong  
Updating d5d44dc..elca782  
Fast forward  
roc.c | 2 ++  
1 files changed, 2 insertions(+), 0 deletions(-)  
[rocrocket@wupengchong git21]$
```

很好，成功了！看看你的 master 分支的 roc.c 代码，是不是已经把悟空和八戒的劳动成果吸纳过来了 呵呵 ~

```
[rocrocket@wupengchong git21]$ cat -n roc.c  
1  #include<stdio.h>  
2  int main()  
3  {  
4      printf("Please guess who he is.\n");  
5      printf("He is a youngster.\n");  
6      printf("He is a male.\n");  
7      return 0;  
8  }  
[rocrocket@wupengchong git21]$
```

很明显，youngster 和 male 都已经到来了。

7 实验完毕。可以删除两个分支了。

好了，实验完毕了，master（师傅）已经将悟空和八戒的劳动成果吸纳为己有了（呵呵），看来悟空和八戒已经可以自由的去玩了。

我们先来删除 bajie：

```
[rocrocket@wupengchong git21]$ git branch -d bajie
```

Deleted branch bajie.

很好，成功删除。我们使用了前面提到的-d 选项，这个选项用于在分支成功合并后删除分支。

还有一个选项叫-D，是大写的d，它主要用于在分支失败后删除分支。

我们用-D 删除 wukong：

```
[rocrocket@wupengchong git21]$ git branch -D wukong
```

Deleted branch wukong.

看，也成功了！这是因为这两个分支都已经成功 merge 了。

那么在什么时候-D 和-d 不同呢。下次重点讲解。：)

今天写了很多内容，超级累了，读者一步一步跟下来，一定会对 git 的工作原理有更深入的理解。欣慰欣慰~~ You can count on me!^\_^

===

## 《看日记学 git》之二十二

在之前的讲解中，涉及过 git branch 删除分支时 -D 和-d 的区别，但没有深入研究，感觉不解渴。

今天，就来分各种情况，好好研究研究-D 和-d 吧。（今天实验室组织去生存岛玩，很赞但很累。现在已经过了 12 点，凌晨坚持写完这篇博客... 困极了...）

前提要说清，我们有 master 主分支、wukong 分支共两个分支。

**情况一：wukong 分支处于已 commit 状态。master 也处于 commit 状态。然后在 master 主分支进行-D 和-d，看看结果。**

**结论：**

如果 master 分支执行了 merge wukong 操作，那么使用-d 和-D 是没有区别的。

如果 master 分支没有执行过 merge wukong 操作，且 wukong 做出了修改，也就是说 master 根本没有把 wukong 的工作吸纳进来，则使用-d 会显示：

```
[rocrocket@wupengchong git22]$ git branch -d wukong
error: The branch 'wukong' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D wukong'.
```

如果是用-D 当然会成功删除这个分支：

```
[rocrocket@wupengchong git22]$ git branch -D wukong
Deleted branch wukong.
```

**情况二：wukong 分支处于未 commit 状态。然后在 master 主分支进行-D 和-d，看看结果。**

**结论：**



这种情况更为糟糕，因为当分支未 commit 时，你甚至是无法转移到 master 主分支的，瞧：

```
[rocrocket@wupengchong git23]$ git checkout master
error: Entry 'roc.c' not up to date. Cannot merge.
```

无法切换到主分支，那么删除 wukong 分支也就无从谈起了。看来还是要老老实实的 commit wukong 分支阿。

情况三：wukong 分支处于已 commit 状态。master 处于未 commit 状态。然后在 master 主分支进行-D 和-d，看看结果。

**结论：**

如果 wukong 进行了改进，那么在 master 分支使用-d 来删除 wukong 分支时会失败：

```
[rocrocket@wupengchong git23]$ git branch -d wukong
error: The branch 'wukong' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D wukong'.
```

如果 wukong 进行了改进，那么使用-D 时会成功：

```
[rocrocket@wupengchong git23]$ git branch -D wukong
Deleted branch wukong.
```

看来-d 其实对于 master 是没有要求的，只是对 wukong 分支有要求，即必须是 commit 的。

**情况四：wukong 分支处于未 commit 状态。master 也处于未 commit 状态。然后在 master 主分支进行-D 和-d，看看结果。**

**结论：**

这个情况当然和情况二一样喽。

四种情况，涵盖了大部分-D 和-d 的情况。足够用了！

ps:这几天找工作，忙阿，投了 google, 不过都杳无音信了，还有 baidu,tencent,redhat, 继续努力吧 呵呵~  
晚安了！

===

## 《看日记学 git》之二十三

本次讲解“clone 和 pull 的艺术”。大家阅读完本文，就可以掌握如何和其他的合作开发者一起协同开发了！

1 以 rocrocket 用户来建立一个 git 仓库，在 master 主分支里建立 roc.c 文件，然后建立 wukong 分支，在其中改进 roc.c 文件。保证 wukong 分支为 commit 状态。然后再改进 master 分支的代码以使 master 分支的 roc.c 为未提交状态。

2 登录到 bob 用户，利用 clone 来获取 rocrocket 的信息

3 以 bob 为登录用户来改进 rocrocket 的 master 的代码，然后切换到 rocrocket 来 pull bob 修改的代码。

开始：

**1 以 rocrocket 用户来建立一个 git 仓库，在 master 主分支里建立 roc.c 文件，然后建立 wukong 分支，在其中改进 roc.c 文件。保证 wukong 分支为 commit 状态。然后再改进 master 分支的代码以使 master 分支的 roc.c 为未提交状态。**

```
[roccrocket@wupengchong git23]$ ls
roc.c
[roccrocket@wupengchong git23]$ git init
Initialized empty Git repository in .git/
[roccrocket@wupengchong git23]$ git commit -a
# On branch master
#
# Initial commit
#
# Untracked files:
```

```
# (use "git add <file>..." to include in what will be committed)
#
#      roc.c
nothing added to commit but untracked files present (use "git add" to track)
[rocrocket@wupengchong git23]$ git add .
[rocrocket@wupengchong git23]$ git commit
Created initial commit d3d0679: rocrocket:master:001
1 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 roc.c
[rocrocket@wupengchong git23]$
```

从这个初始化过程可以看出，git commit -a 命令只可以用在已经建立了 index file 之后，也就是在第一次初始化时，必须要使用 git add 命令来建立 index file，否则会报错。

然后建立 wukong 分支：

```
[rocrocket@wupengchong git23]$ git branch wukong
[rocrocket@wupengchong git23]$ git checkout wukong
Switched to branch "wukong"
[rocrocket@wupengchong git23]$ vi roc.c
[rocrocket@wupengchong git23]$ git commit -a
Created commit aad38ac: rocrocket:wukong:001
1 files changed, 1 insertions(+), 0 deletions(-)
[rocrocket@wupengchong git23]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
```

```
5     printf("He is born in beijing.\n");
6     return 0;
7 }
[roccrocket@wupengchong git23]$
```

黑体是在 master 的基础上新加入的一行代码。好了，wukong 分支已经处于 commit 状态了。现在回到 master，再改进下代码，并使得 master 处于未 commit 状态：

```
[roccrocket@wupengchong git23]$ git checkout master
Switched to branch "master"
[roccrocket@wupengchong git23]$ vi roc.c
[roccrocket@wupengchong git23]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
5      printf("He is crazy about linux.\n");
6      return 0;
7  }
[roccrocket@wupengchong git23]$ pwd
/roccrocket/PSB/home/git23
```

在 master 分支中加入了一句 He is crazy about linux. 并且我故意没有 git commit.

2 登录到 bob 用户，利用 clone 来获取 roccrocket 的信息。

```
[bob@wupengchong ~]$ whoami
bob
[bob@wupengchong ~]$ git clone /roccrocket/PSB/home/git23 bob23
```

```
Initialized empty Git repository in /roccrocket/PSB/bob/bob23/.git/
[ bob@wupengchong ~]$ ls
bob23
[ bob@wupengchong ~]$ cd bob23/
[ bob@wupengchong bob23]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("Please guess who he is.\n");
5      return 0;
6  }
[ bob@wupengchong bob23]$ git branch
* master
```

我们成功的在 bob 用户下将 roccrocket 的代码 clone 过来了，并放到了 bob 自己定义的 bob23 目录下。首先可以看到，我们 clone 到的是 roccrocket 的 master 分支的已 commit 的代码；而且可以看到当前的分支只有一个 master 主分支。而在 roccrocket 里的 wukong 分支并没有被 clone 过来。恩，不要沮丧和奇怪。git 是这样设计的：clone 的话，是 clone 远端的当前分支。通俗的说，远端当前处在哪个分支，你 clone 来的就是哪个分支。这下，你该知道如何 clone 到 roccrocket 的 wukong 分支了吧，对！就是让 roccrocket 也 checkout 到 wukong 分支，然后你再 clone 就 OK 了！这个时候你到 bob23 目录下再 git branch 会得到只有 wukong 一个分支。对滴，你要明确一点，不是任何 git 仓库都有 master 分支的哦~

By the way, 在执行 git checkout branchname 时，是必须保证当前本分支处于 commit 状态，否则 git 会提示：

```
[roccrocket@wupengchong git23]$ git checkout wukong
error: Entry 'roc.c' not uptodate. Cannot merge.
我们这个时候切换到 roccrocket，将未 commit 的代码 commit：
[roccrocket@wupengchong git23]$ git branch
```

```
* master
wukong
[roccrocket@wupengchong git23]$ git commit -a
Created commit fadfdb4: roccrocket:master:002
1 files changed, 1 insertions(+), 0 deletions(-)
```

3 以 bob 为登录用户来改进 roccrocket 的 master 的代码，然后切换到 roccrocket 来 pull bob 修改的代码。

```
[bob@wupengchong bob23]$ ls
roc.c
[bob@wupengchong bob23]$ git branch
* master
[bob@wupengchong bob23]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Please guess who he is.\n");
return 0;
}
[bob@wupengchong bob23]$ vi roc.c
[bob@wupengchong bob23]$ cat -n roc.c
1      #include<stdio.h>
2      int main()
3      {
4          printf("Please guess who he is.\n");
5          printf("His name is roc.\n");
6          return 0;
```

```
7    }
[ bob@wupengchong bob23]$ git commit -a
Created commit 7c1cd89: bob:master:001
1 files changed, 1 insertions(+), 0 deletions(-)
```

改进完毕并成功提交了。下面的任务就是转回到 rocrocket 来试着 pull 了！（pull 就是取回代码的命令）

```
[ rocrocket@wupengchong git23]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Please guess who he is.\n");
printf("He is crazy about linux.\n");
return 0;
}
[ rocrocket@wupengchong git23]$ git pull /rocrocket/PSB/bob/bob23
Unpacking objects: 100% (3/3), done.
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2)remote: , done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Auto-merged roc.c
CONFLICT (content): Merge conflict in roc.c
Automatic merge failed; fix conflicts and then commit the result.
[ rocrocket@wupengchong git23]$
```

可以看出 rocrocket 的代码和 bob 的代码是有冲突的，所以在我 git pull 远端仓库时，提示我 conflict。

这时，需要我自己解决冲突了。

```
[ rocrocket@wupengchong git23]$ vi roc.c
```

```
[rocrocket@wupengchong git23]$ cat -n roc.c
1   #include<stdio.h>
2   int main()
3   {
4       printf("Please guess who he is.\n");
5       printf("He is crazy about linux.\n");
6       printf("His name is roc.\n");
7       return 0;
8   }
[rocrocket@wupengchong git23]$
```

好，解决完冲突了，下面来 commit 吧！

```
[rocrocket@wupengchong git23]$ git commit -a
```

Created commit ed92cd2: new pull!

好了 一切都安静了，我们已经成功把 bob 的工作合并到 roccrocket 的工作之中了。

下次将重点讲解如何更好的获取别人的代码。

===



## 《看日记学 git》之二十四

今天重点讨论如何更好的从别人那里获取工作进展。你是否想过，在一个人员繁多的项目组里，你是很难深入了解每个人的编程能力的，那么你的员工编写完了代码，你是否会一下就 pull 到你的 master 主分支呢？如果你会，那么你胆子够大。不过大部分 project manager 不会这样做的，他们会先把代码取过来放到一个临时的地方，仔细考究员工编写的代码，认为代码合格后，才会放心的放到 master 中去。

我们即将学习和研究的就是这种 git 安全处理机制的操作方法。

首先在 rocrocket 的 git24 目录下建立一个新 roc.c 文件。然后在 bob 用户下 git clone 这个 git24 目录到本地的 bob24 目录下。

```
[bob@wupengchong ~]$ git clone /roccrocket/PSB/home/git24 bob24
Initialized empty Git repository in /roccrocket/PSB/bob/bob24/.git/
[bob@wupengchong ~]$ ls
bob24
[bob@wupengchong ~]$
```

好了，已经 clone 过来了！下面 bob 来改进代码：

```
[bob@wupengchong ~]$ cd bob24/
[bob@wupengchong bob24]$ ls
roc.c
[bob@wupengchong bob24]$ git branch
* master
```

```
[bob@wupengchong bob24]$ vi roc.c
[bob@wupengchong bob24]$ cat -n roc.c
1      #include<stdio.h>
2      int main()
3      {
4          printf("Hello,everyone!\n");
5          printf("Good bye!\n");
6      }
[bob@wupengchong bob24]$ git commit -a
Created commit 42b7069: bob:001
1 files changed, 1 insertions(+), 0 deletions(-)
[bob@wupengchong bob24]$
bob 已经改进了代码，并成功提交了！
```

下面就要到重头戏了，如何安全的处理员工的代码。

我们切换到 rocrocket ( 也就是 PM 的角色 ) ，

```
[rocrocket@wupengchong git24]$ git fetch /rocrocket/PSB/bob/bob24 master:bob
Unpacking objects: 100% (3/3), done.
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2)remote: , done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From /rocrocket/PSB/bob/bob24
* [new branch]      master      -> bob
[rocrocket@wupengchong git24]$
```

看到了吧，PM 用了一招，叫 fetch！用这个命令，可以将远端的代码先下载到本地的指定分支（例子中是 bob）中，而

不是像 git pull 那样直接合并到当前分支。

```
[rocrocket@wupengchong git24]$ git branch
bob
* master
[rocrocket@wupengchong git24]$ git checkout bob
Switched to branch "bob"
[rocrocket@wupengchong git24]$ ls
roc.c
[rocrocket@wupengchong git24]$ cat roc.c
#include<stdio.h>
int main()
{
printf("Hello,everyone!\n");
printf("Good bye!\n");
}
[rocrocket@wupengchong git24]$
```

我们切换到 bob 分支并开始检查代码。告诉你一个新命令，这个命令可以很方便的查看两个分支的代码区别，它就是 git-whatchanged !

```
[rocrocket@wupengchong git24]$ git-whatchanged -p master..bob
commit 42b706951ed78bad6ce716f61ca56fe63ad61067
Author: bob <bob@roclinux.cn>
Date:   Fri Oct 10 15:03:30 2008 +0800
bob:001
diff -git a/roc.c b/roc.c
index 33e540b..885aa26 100644
```

```
- a/roc.c
+++ b/roc.c
@@ -2,4 +2,5 @@
int main()
{
printf("Hello,everyone!\n");
+    printf("Good bye!\n");
}
[roccrocket@wupengchong git24]$
```

看，已经显示的很清楚了！经过 PM 的审核，可以放心地把代码加入到 master 中了！

```
[roccrocket@wupengchong git24]$ git checkout master
Switched to branch "master"
[roccrocket@wupengchong git24]$ git pull . bob
Updating dfa52c4..42b7069
Fast forward
roc.c |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
[roccrocket@wupengchong git24]$
```

我们仍然坚持使用了 git pull 命令来进行合并工作。其实此时，我们使用 merge 也未尝不可。

git-pull 的第一个参数表示仓库的位置，我们使用 “.” 表示在当前目录下。

git-pull 的第二个参数表示分支名称，我们使用 bob 表示了分支名称。

git-pull 默认就是将远端代码获取来并合并到当前分支中。

好了，任务完成！

最后总结一下：PM 使用 git fetch 将远端代码获取到一个本地新分支中，然后使用 git-whatchanged 来查看区别，最后

用 git pull 来将代码合并到本地分支中。

## 《看日记学 git》之二十五

这次重点讲解 git reset 这条命令。这个命令主要是用在恢复代码上。当你发现自己编写的一段已提交的代码是错误的，那么你就会用到 git reset 了！

**1 讲解 git reset --soft**

**2 讲解 git reset --hard**

**3 讲解 git reset --mixed**

**4 讲解 git reset**

**5 讲解 git reset --**

开始：

**1 讲解 git reset --soft**

```
[rocrocket@wupengchong ~]$ mkdir git25
[rocrocket@wupengchong ~]$ cd git25/
[rocrocket@wupengchong git25]$ vi roc.c
[rocrocket@wupengchong git25]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("He is a young man.\n");
5      return 0;
```

```
6 }
[roccrocket@wupengchong git25]$ git init
Initialized empty Git repository in .git/
[roccrocket@wupengchong git25]$ git add .
[roccrocket@wupengchong git25]$ git commit -m "1"
Created initial commit a2909db: 1
1 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 roc.c
[roccrocket@wupengchong git25]$ git log
commit a2909db30720fe7bb85724fdfad89b5fffd280b05
Author: roccrocket <wupengchong@gmail.com>
Date:   Wed Oct 15 08:46:20 2008 +0800
1
[roccrocket@wupengchong git25]$ vi roc.c
[roccrocket@wupengchong git25]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("He is a young man.\n");
5      printf("He is 24 years old.\n");
6      return 0;
7  }
[roccrocket@wupengchong git25]$ git commit -a -m "2"
Created commit 5fcb5ab: 2
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
[roccrocket@wupengchong git25]$ git log
commit 5fcb5abeee4dea39da8946bd39c6daea19977558
Author: roccrocket <wupengchong@gmail.com>
Date:   Wed Oct 15 08:47:10 2008 +0800
2
commit a2909db30720fe7bb85724fdfad89b5fffd280b05
Author: roccrocket <wupengchong@gmail.com>
Date:   Wed Oct 15 08:46:20 2008 +0800
1
[roccrocket@wupengchong git25]$
```

到目前为止，我们建立了一个 git 仓库，并进行了两次 commit。

此时，我发现我的 commit 是错误的，此人不是 24 岁，而是 25，我想撤销第二次 commit。看看用 `-soft` 会是什么效果：

```
[roccrocket@wupengchong git25]$ git reset -soft HEAD^
[roccrocket@wupengchong git25]$ git log
commit a2909db30720fe7bb85724fdfad89b5fffd280b05
Author: roccrocket <wupengchong@gmail.com>
Date:   Wed Oct 15 08:46:20 2008 +0800
1
[roccrocket@wupengchong git25]$ git diff
[roccrocket@wupengchong git25]$ git diff -cached
diff -git a/roc.c b/roc.c
index b089322..ee25766 100644
- a/roc.c
+++ b/roc.c
@@ -2,5 +2,6 @@
```



```
int main()
{
printf("He is a young man.\n");
+      printf("He is 24 years old.\n");
return 0;
}
[roccrocket@wupengchong git25]$ git diff HEAD
diff -git a/roc.c b/roc.c
index b089322..ee25766 100644
- a/roc.c
+++ b/roc.c
@@ -2,5 +2,6 @@
int main()
{
printf("He is a young man.\n");
+      printf("He is 24 years old.\n");
return 0;
}
[roccrocket@wupengchong git25]$ cat -n roc.c
1  #include<stdio.h>
2  int main()
3  {
4      printf("He is a young man.\n");
5      printf("He is 24 years old.\n");
6      return 0;
```

```
7 }  
[roccrocket@wupengchong git25]$
```

你如果一步一步按照我的设计运行了上面的命令，你会发现 git reset 之后，git diff 返回空，而 git diff --cached 和 git diff HEAD 会返回有效信息。这说明使用--soft 选项后，只回退了 commit 的信息，而不会回复到 index file 一级。哈哈，这就明了了！如果你想撤销 commit，并且只回退 commit 的信息，那么就用--soft 吧！

而且你可以观察到 git reset 的意思是“撤销到哪个位置”，。也就是说代码管理者需要在后面的参数中指定一个之前的 commit 位置。如上面提到的 HEAD^。

## 2 讲解 git reset --hard

有了--soft 的试验思路，我想你也应该知道如何测试--hard 了。有意思的工作留给你去自己完成吧。我只说结论：

--hard 会完全撤销一个 commit，彻底的回复到上一次 commit 的状态。连 working tree 的源代码也会完全倒退到上次 commit 之时的状态。所以使用--hard 后，git diff，git diff --cached 和 git diff HEAD 都会返回空。

有了这个--hard 好工具，你可以这样做：在当前的 current working tree 中修改了代码，你可以选择 git add 或者不 add，然后使用 git reset --hard HEAD 命令就可以恢复到修改之前的最初状态了。你修改的代码和 git add 的信息都会被丢弃。这个用法记住它，早晚你会用到它。但往往你会武断的认为 git reset 只能恢复到之前的 commit 状态，但你往往想不到 git reset 还可以恢复到当前的 HEAD 所指定的 commit 状态。

## 3 讲解 git reset --mixed

--mixed 选项会撤销最近的一次 commit，只保留 working tree 的源代码级的修改，而 index file 和 commit 都会回复到上一次 commit 的状态。所以使用--mixed 后，git diff 和 git diff HEAD 会有有效信息的输出，而 git diff --cached 会输出空。

## 4 讲解 git reset

我只需要告诉你--mixed 是 git reset 的默认选项。你应该知道了，git reset 和 git reset --mixed 效果是完全一样的。

## 5 讲解 git reset --

如果你想从 index file 中删除一个已登记的文件，那么就用这个命令。

比如，我想删除 index file 里登记的 roc.c，那么就 `$ git reset --roc.c` 就可以了！

这个功能似乎很奇怪，什么时候会用到呢？哦！是这样，如果你刚刚 `git add` 了一个文件到 index file 里，但是你突然发现这是错误的，那么就要用到 `git reset` 喽！

## 《看日记学 git》之二十六

在之二十五中讲到了 git reset 的用法，有个朋友问了我 N 个关于 reset 的问题。我认为可能是他对 working tree、index file 和 commit 的关系和区别还不是很明白（明显是没有仔细看前面的日记 呵呵）

下面总结一下 git reset 的各个选项吧：

- 1 git reset --soft 只撤销 commit，保留 working tree 和 index file。
- 2 git reset --hard 撤销 commit、index file 和 working tree，即撤销销毁最近一次的 commit
- 3 git reset --mixed 撤销 commit 和 index file，保留 working tree
- 4 git reset 和 git reset --mixed 完全一样
- 5 git reset --用于删除登记在 index file 里的某个文件。

## 《看日记学 git》之二十七

在经过数天的总结、复习之后，我们到了继续前进的时候了。

在本次讲解中，我们重点关注 git-show-branch 这个命令。它和 git branch 不一样，后者的功能是列出当前所有的分支。

我们先来看看这两个命令的不同之处：

```
[rocrocket@wupengchong clonetest]$ git show-branch
* [master] 1
! [wukong] 5
--
+ [wukong] 5
+ [wukong^] bajie
+ [wukong~2] 2
*+ [master] 1
[rocrocket@wupengchong clonetest]$ git branch
* master
wukong
[rocrocket@wupengchong clonetest]$
```

显而易见，git show-branch 会输出很多很晦涩的东西。好，下面我们就通过一个实例来搞定这个晦涩的命令！

首先新建一个 git 仓库，如下：

```
[rocrocket@wupengchong showbranch]$ cat roc.c
int main()
```

```
{  
printf("Firstly");  
return 0;  
}  
[rocrocket@wupengchong showbranch]$ git branch  
* master  
[rocrocket@wupengchong showbranch]$ git show-branch  
[master] Firstly  
[rocrocket@wupengchong showbranch]$ git log  
commit b11be45d63226eac8b89fa05119d4282cfd73df2  
Author: rocrocket ;  
Date:   Fri Nov 14 12:00:18 2008 +0800  
  
Firstly  
[rocrocket@wupengchong showbranch]$
```

可以看到，我只提交了一次 commit，在 git show-branch 时会显示分支名称和其开发日志的内容。（我的第一个 commit 的开发日志就是一个单词“Firstly”）

之后，我们修改一下 roc.c 文件再提交一次，开发日志设定为“Secondly”：

```
[rocrocket@wupengchong showbranch]$ cat roc.c  
int main()  
{  
printf("Firstly");  
printf("Secondly");  
return 0;  
}
```

```
[roccrocket@wupengchong showbranch]$ git log
commit 5102e1a2c812dfa6539d77f1ecede46faaa26c4f
Author: roccrocket ;
Date:   Fri Nov 14 12:04:54 2008 +0800
```

Secondly

```
commit b11be45d63226eac8b89fa05119d4282cfd73df2
Author: roccrocket ;
Date:   Fri Nov 14 12:00:18 2008 +0800
```

Firstly

```
[roccrocket@wupengchong showbranch]$ git show-branch
[master] Secondly
[roccrocket@wupengchong showbranch]$
```

如上可见，此时 git show-branch 输出的是 master 分支第二次 commit 的开发日志，而第一次的“ Firstly”并没有再出现了。

下面，我们新建一个分支，老规矩，叫做 wukong：

```
[roccrocket@wupengchong showbranch]$ git branch wukong
[roccrocket@wupengchong showbranch]$ git branch
* master
wukong
[roccrocket@wupengchong showbranch]$ git show-branch
* [master] Secondly
! [wukong] Secondly
```

```
--
*+ [master] Secondly
[roccrocket@wupengchong showbranch]$ git checkout wukong
Switched to branch "wukong"
[roccrocket@wupengchong showbranch]$ git-show-branch
! [master] Secondly
* [wukong] Secondly
--
+* [master] Secondly
[roccrocket@wupengchong showbranch]$
```

看，这时再 `git show-branch` 就出现了稍微复杂一点的输出。里面有\*和!，还有-，这都是什么乱七八糟的阿？！

如果你仔细观察了上面的输出信息，你会看到两次 `git-show-branch` 之后的!和\*调换了位置，对，这是因为我切换分支所造成的。所以可以断定!和\*是和当前分支有关的。

不卖关子了，告诉你：\*（星号）代表 HEAD 所指的分支，而其他分支则会标识为!（叹号）。这两个符号有些特别，你可能已经观察到了!和\*是有缩进区别的，这是因为!和\*不仅要标识本行是否为“当前分支”，而且还用来标识一列。比如说，!标识第一列，也就是说下面的输出中只要在第一列有符号的，都是在指示 master 分支的；而\*在第二列，只要下面的输出中有符号的，都表示在指示 wukong 分支。（OK,好好观察观察，自己做做实验，这个知识点不搞定，后面的内容可看不懂哦）

接下来，你会看到“-”符号，不要紧张，这个符号只是一个分隔符，用于更清晰的区分各个列用的，没有什么特殊意义的。呵呵，草木皆兵可不好...

我现在要告诉你，“-”符号上部内容主要用于显示分支列表，而“-”下部的内容为分支 commit 的关系。（看不懂没关系，看完全文再回来品品，保你能看懂）

了解了“-”上面的内容了，下面来研究“-”下面的内容，这里面内容比较花哨，又有空格，又有加号，其实还会有



减号呢！

开门见山：

+（加号）表示所在分支包含此行所标识的 commit

（空格）表示所在分支不包含此行所标识的 commit

-（减号）表示所在分支是经过 merge 得到的，而所在行的内容即是 merge 的基本信息。

\*（星号）表示如果需要在某列标识+（加号），且此列为当前分支所在列，那么则将+（加号）转变为\*（星号）。

我们来分析一个例子：

```
[rocrocket@wupengchong showbranch]$ git-show-branch
* [master] merge
! [wukong] wukong:Thirdly
--
- [master] merge
*+ [wukong] wukong:Thirdly
```

可以看到：

结论一：当前分支为 master 分支，而 wukong 分支并非当前分支。

结论二：master 分支的 HEAD 所指向的 commit 的开发日志的首行为 “merge”字符串，而 wukong 分支的 HEAD 所指 commit 的开发日志的首行信息为 “wukong: Thirdly”

结论三：根据 “-”符号可以知道，master 分支的 HEAD 的最近一次 commit 是由 merge 得到的，且此次 merge 后 commit 的信息为 “merge”字符串。你可能看到了在 “-”后面还有一个空格，这说明当前行所指的这次 commit 对 wukong 分支（第二列）是没有影响的，所以用 “空格”表示。

结论四：根据 “\*+”所在列可知，当前行的 commit 影响范围波及到了 master 分支和 wukong 分支，也就是说此次 commit 在两个分支都有效。这是为什么呢？很简单，这是 merge 的力量，这说明我在 wukong 分支的这一次 commit（其

开发日志的后行为 “wukong: Thirdly”) 被 merge 到了 master 分支。

ok~ 看懂这几段后, 你基本上已经掌握了 git-show-branch 命令了!

最后给你一个例子, 来分析分析, 看看到底发生了什么:

```
[rocrocket@wupengchong showbranch]$ git-show-branch
! [bajie] shaseng:Secondly
* [master] Fourth
! [shaseng] shaseng:Secondly
! [wukong] shaseng:Secondly
---
- [master] Fourth
+*++ [bajie] shaseng:Secondly
over~
```

## 《看日记学 git》之二十八

由于最近清空硬盘并安装了全新的 fedora 10，所以我使用 git 管理的项目也只能先备份然后转移到新系统中来。

有些人会问使用 git 管理的项目如何迁移呢？

其实备份及恢复过程非常简单，但是如果你不甚了解，可能 git 会给你一个小小的惊吓~~

举例来说：

我用于 git 管理的项目目录为 yaoming，那么备份的话，我只需要将整个 yaoming 目录拷贝到其他存储设备上即可，你使用 cp、rsync 或者 rcp 等，都随你。记得查看一下，其中的隐藏目录 .git 也一定要备份过去。

当恢复时，只需要将备份在其他设备上的 yaoming 目录拷贝回来，然后在 yaoming 目录下运行 git-init 即可，git 很聪明的，他知道你是想恢复一个仓库还是想新建一个空仓库。然后，就 OK 了，您可以继续项目开发了。

当然如果你不知道执行 git-init 这一步，信心满满的以为 git 仓库可以随便挪，那 git 的报错提示 “fatal: Not a git repository”，会让你以为你的 log、branch 等等都付之东流了呢！（其实只是虚惊一场~~）

ove~

## 《看日记学 git》之二十九

还是继续《看日记学 git》之二十八的话题。

故事是这样的：为了从 fedora 9 转移到 fedora 10，我把用 git 管理的项目拷贝到了 Fat32 格式的优盘里，然后等 fedora 10 安装好后，又将优盘里的 git 管理的项目拷贝到 fedora 10 中。

今天我 git commit 时，git 莫名其妙的提醒我：

```
[rocrocket@rocrocket Anycatch]$ git commit -a -F devlog/log-0.0.5
*
* You have some suspicious patch lines:
*
* In anycatch.c
* trailing whitespace (line 63)
anycatch.c:63:    struct in_addr inp;
* trailing whitespace (line 66)
anycatch.c:66:    sscanf(strline,"%s %u",str_ip,&_port);
* trailing whitespace (line 133)
anycatch.c:133:        read_IP_list(strline);
* trailing whitespace (line 135)
anycatch.c:135:        read_IP_sched(strline);
* In anycatch.h
* trailing whitespace (line 153)
```

```
anycatch.h:153:    IPAddress ip;  
* In devlog/template-0.0.1  
* trailing whitespace (line 6)  
devlog/template-0.0.1:6:DETAIL:
```

重点的一句我用粗体加黑了。

解决过程在此不表，关键说结论。

这个问题是由于文件系统差异导致的。解决办法是按照报错提示，将其所在行最后的空格去掉去掉即可。

当然，还有一种解决办法，那就是

编辑.git/hooks/pre-commit 文件，将其中的如下行注释掉也可以：

```
if (/\\s$/) {  
bad_line("trailing whitespace", $_);  
}
```

over~

## 《看日记学 git》之三十一

最近一直在专注于 FreeBSD 的资料收集整理和项目，所以 git 系列进展不多。今天中午休息时，发现了一个比较好的网站，属于 git 综述类型的，有兴趣可以看看，网址是 <http://zh-tw.whygitisbetterthanx.com> 而且本文还引出了一个网络 git 服务工具 GitHub，同样类型的工具还有一些，大家可以上 baidu/google 搜搜。GitHub 有免费服务类型，空间 100M，不限 repo 数目，不限合作者人数。 :) over~~

## 《看日记学 git》之随笔笔记

本文章就是我复习前三十集后的一个总结，把忘了的知识点写在这重新提一下。你也看看，正好复习复习:)

1 git clone 只会提取远程 git 仓库里已提交的信息，working tree 和 index file 里的信息，git clone 根本不会理会。如果你是经理，你让你的手下用 git clone path/to/your/repo newrepo 来将你的代码下载到他的本地就可以了，注意，他用此命令下载的是 path/to/your/repo 仓库中的当前分支的内容。当你的手下开发完他负责的那部分代码之后，通知你。你用 git fetch path/to/his/newrepo hisbranch:yourbranch 来将你手下的工作成果抓取到你本地的 yourbranch 分支上。待审核合格之后，你就可以用 git merge 来合并他的工作了。

2 想查看两个分支的区别，就用 git whatchanged -p branch1..branch2。注意，-p 表示以易理解的字符方式表示两者区别。如果不加-p 的话，git 会输出一些内部格式的区别信息，你根本看不懂:)

3 git pull 的用法至少有四种，一个是在 git clone 之后用 git pull 来同步；第二个用法是用 git pull .来替代 git merge 命令；第三个用法是 git pull /path/to/repo，将远程仓库的内容直接 merge 到当前分支中。第四种用法是在 git pull /path/to/repo 后面加上 <src>:<dst> 来指定源的分支和本地分支。

4 git 里有四种对象：commit 对象、tree 对象、blob 对象和 tag 对象

5 尽量不要用 git-push，因为他类似于 CVS 的管理模式，而 git 的模式特点就是分布式，所以常用 git pull 代替 git push。

6 git fetch <仓库路径> <分支信息>，用于从远程仓库获取代码。其中 <分支信息> 的格式是这样的 "<src>:<dst>"，<src> 表示源的分支，而 <dst> 表示本地分支。如果 <dst> 被省略，那么就默认为本地当前分支。

7 如果在当前分支有代码修改，但未 commit，那么不允许 git checkout 到其他分支。

8 想从 index file 中删除一个登记文件，用 `git reset -- filename`  
9 用 `git checkout -- filename` 来将此文件代码恢复到 index 中的状态。  
over~