GIT使用入门

Part 1

第1章　基本原理

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，Bazzar等等。

当然git，Mercurial和Bazzar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，bzr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

那么，简单地说，Git

究竟是怎样的一个系统呢？请注意，接下来的内容非常重要，若是理解了 Git 的思想和基本的工作原理，用起来就会知其所以然，游刃有余。在开始学习 Git 的时候，请不要尝试把各种概念和其他的版本控制系统诸如 Subversion 和 Perforce 等相比拟，否则容易混淆每个操作的实际意义。Git 在保存和处理各种信息的时候，虽然操作起来的命令形式非常相近，但它与其他版本控制系统的做法颇为不同。理解这些差异将有助于你准确地使用 Git 提供的各种工具。

直接快照，而非比较差异

Git 和其他版本控制系统的主要差别在于，Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。这类系统（CVS，Subversion，Perforce，Bazaar 等等）每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容，请看图 1-4。

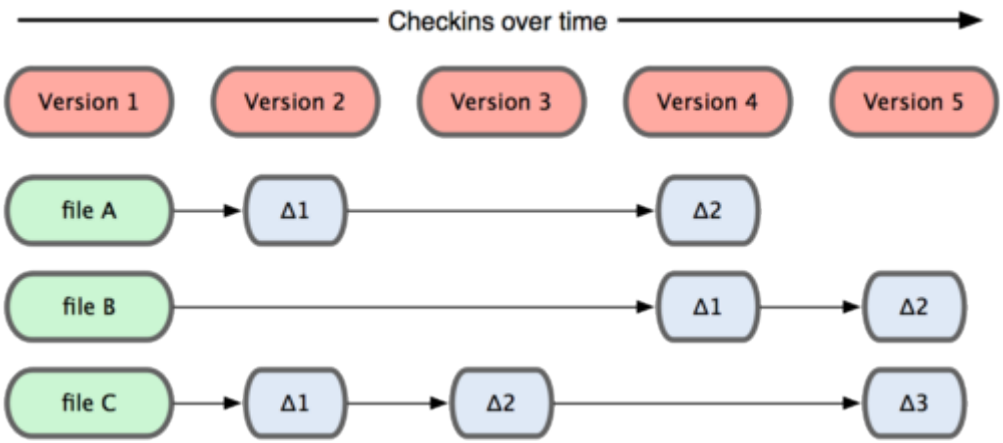


图 1-4. 其他系统在每个版本中记录着各个文件的具体差异

Git 并不保存这些前后变化的差异数据。实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新

时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引。为提高性能，若文件没有变化，Git

不会再次保存，而只对上次保存的快照作一连接。Git 的工作方式就像图 1-5 所示。

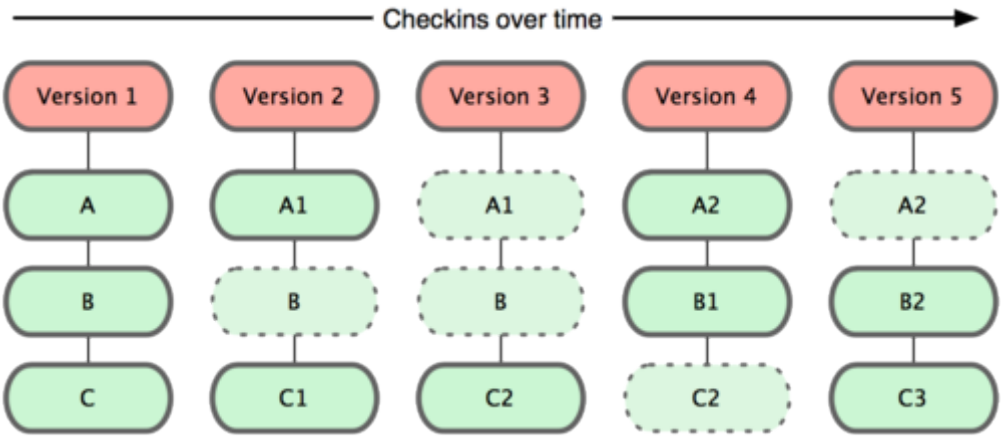


图 1-5. Git 保存每次更新时的文件快照

这是 Git

同其他系统的重要区别。它完全颠覆了传统版本控制的套路，并对各个环节的实现方式作了新的设计。Git

更像是个小型的文件系统，但它同时还提供了许多以此为基础的超强工具，而不只是一个简单的 VCS。稍后在第三章讨论 Git

分支管理的时候，我们会再看看这样的设计究竟会带来哪些好处。

近乎所有操作都可本地执行

在 Git 中的绝大多数操作都只需要访问本地文件和资源，不用连网。但如果用 CVCS 的话，差不多所有操作都需要连接网络。因为 Git

在本地磁盘上就保存着所有有关当前项目的历史更新，所以处理起来速度飞快。

举个例子，如果要浏览项目的历史更新摘要，Git

不用跑到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。如果想要看当前版本的文件和一个月前的版本之间有何差异，Git

会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。

用 CVCS 的话，没有网络或者断开 VPN 你就无法做任何事情。但用 Git

的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程的镜像仓库。同样，在回家的路上，不用连接 VPN

你也可以继续工作。换作其他版本控制系统，这么做几乎不可能，抑或非常麻烦。比如

Perforce，如果不连到服务器，几乎什么都做不了（译注：实际上手工修改文件权限改为可写之后是可以编辑文件的，只是其他开发者无法通过 Perforce

知道你正在对此文件进行修改。）；如果是 Subversion 或

CVS，虽然可以编辑文件，但无法提交更新，因为数据库在网络上。看上去好像这些都不是什么大问题，但在实际体验过之后，你就会惊喜地发现，这其实是会带来很大不同的。

时刻保持数据完整性

在保存到 Git

之前，所有数据都要进行内容的校验和（checksum）计算，并将此结果作为数据的唯一标识和索引。换句话说，不可能在你修改了文件或目录之后，Git

一无所知。这项特性作为 Git

的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git 都能立即察觉。

Git 使用 SHA-1

算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1

哈希值，作为指纹字符串。该字串由 40 个十六进制字符（0-9 及 a-

f）组成，看起来就像是:

24b9da6552252987aa493b52f8696cd6d3b00373

Git

的工作完全依赖于这类指纹字串，所以你会经常看到这样的哈希值。实际上，所有保存在 Git

数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

多数操作仅添加数据

常用的 Git

操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，要回退或重现都会非常困难。在别的 VCS

中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 Git

里，一旦提交快照之后就完全不用担心丢失数据，特别是在养成了定期推送至其他镜像仓库的习惯的话。

这种高可靠性令我们的开发工作安心不少，尽管去做各种试验性的尝试好了，再怎样也不会弄丢数据。至于 Git

内部究竟是如何保存和恢复数据的，我们会在第九章的"幕后细节"部分再作详述。

三种状态

好，现在请注意，接下来要讲的概念非常重要。对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。已提交表示该文件已经被安全地保存在本地数据库中了；已修改表示修改了某个文件，但还没有提交保存；已暂存表示把已修改的文件放在下次提交时要保存的清单中。

由此我们看到 Git 管理项目时，文件流转的三个工作区域：Git 的本地数据目录，工作目录以及暂存区域。

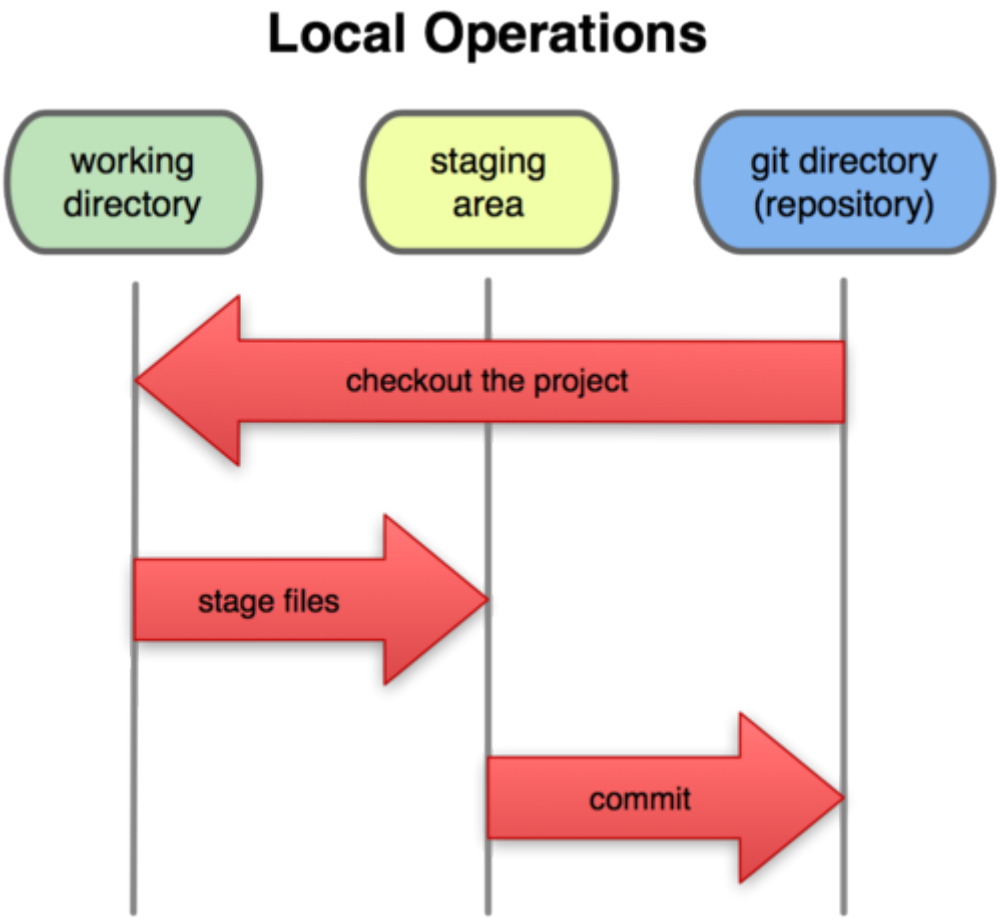


图 1-6. 工作目录，暂存区域和 git 目录

每个项目都有一个 git 目录，它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做工作目录。这些文件实际上都是从 git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

所谓的暂存区域只不过是个简单的文件，一般都放在 git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

基本的 Git 工作流程如下所示:

1. 在工作目录中修改某些文件。 2. 对这些修改了的文件作快照，并保存到暂存区域。 3. 提交更新，将保存在暂存区域的文件快照转储到 git 目录中。

所以，我们可以从文件所处的位置来判断状态：如果是 git 目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。到第二章的时候，我们会进一步了解个中细节，并学会如何善用这些状态，以及如何跳过暂存环节。

第二章 安装

安装 Git

是时候动动手了，不过在此之前得先安装好 Git。有许多安装方式，概括起来主要有两种，一种是通过编译源代码来安装；另一种是使用为特定平台预编译好的安装包。

从源代码安装

若是条件允许，从源代码安装有很多好处，至少可以安装最新的版本。Git 的每个版本都在不断尝试改进用户体验，所以能通过源代码自己编译安装最新版本就再好不过了。有些 Linux

版本自带的安装包更新起来并不及时，所以除非你在用最新的 distro 或者

backports，那么从源代码安装其实该算是最佳选择。

Git 的工作需要调用 curl，zlib，openssl，expat，libiconv

等库的代码，所以需要先安装这些依赖工具。在有 yum 的系统上（比如

Fedora）或者有 apt-get 的系统上（比如 Debian
体系的），可以用下面的命令安装:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

之后，从下面的 Git 官方站点下载最新版本源代码:

http://git-scm.com/download

然后编译并安装:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

现在已经可以用 git 命令了，用 git 把 Git
项目仓库克隆到本地，以便日后随时更新:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

在 Linux 上安装

如果要在 Linux 上安装预编译好的 Git

二进制安装包，可以直接用系统提供的包管理工具。在 Fedora 上用 yum 安装:

```
$ yum install git-core
```

在 Ubuntu 这类 Debian 体系的系统上，可以用 apt-get 安装:

```
$ apt-get instal git-core
```

在 Windows 上安装

在 Windows 上安装 Git 同样轻松，有个叫做 msysGit

的项目提供了安装包，可以从 Google Code 的页面上下载安装文件（.exe）:

http://code.google.com/p/msysgit

完成安装之后，就可以使用命令行的 git 工具（已经自带了 ssh

客户端）了，另外还有一个图形界面的 Git 项目管理工具。

Part 2

Git 基础

第三章 导入一个新项目

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 “rocrocket”

[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。

新建一个git目录:

$ git mkdir rocrocket

这时你可以去C:\Documents and Settings\Administrator\下查看,是不是多了一个rocrocket文件夹?

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rocrocket目录下(C:\Documents and Settings\Administrator\rocrocket)，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 rocrocket/

[root@wupengchong rocrocket]$ git init (rocrocket文件下多了一个.git文件夹)

Initialized empty Git repository in .git/

[root@wupengchong rocrocket]$ ls -a

. .. .git main.c

[root@wupengchong rocrocket]$ git add .

[root@wupengchong rocrocket]$ 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目前的进展信息。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）。

第四章 提交修改后的文件

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？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(这个命令貌似不行,有待讨论)，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:)

第五章　查看日志

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。

第六章　分支

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  (有待讨论)
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…

现在轮到我们人类来帮助告诉**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    }
```

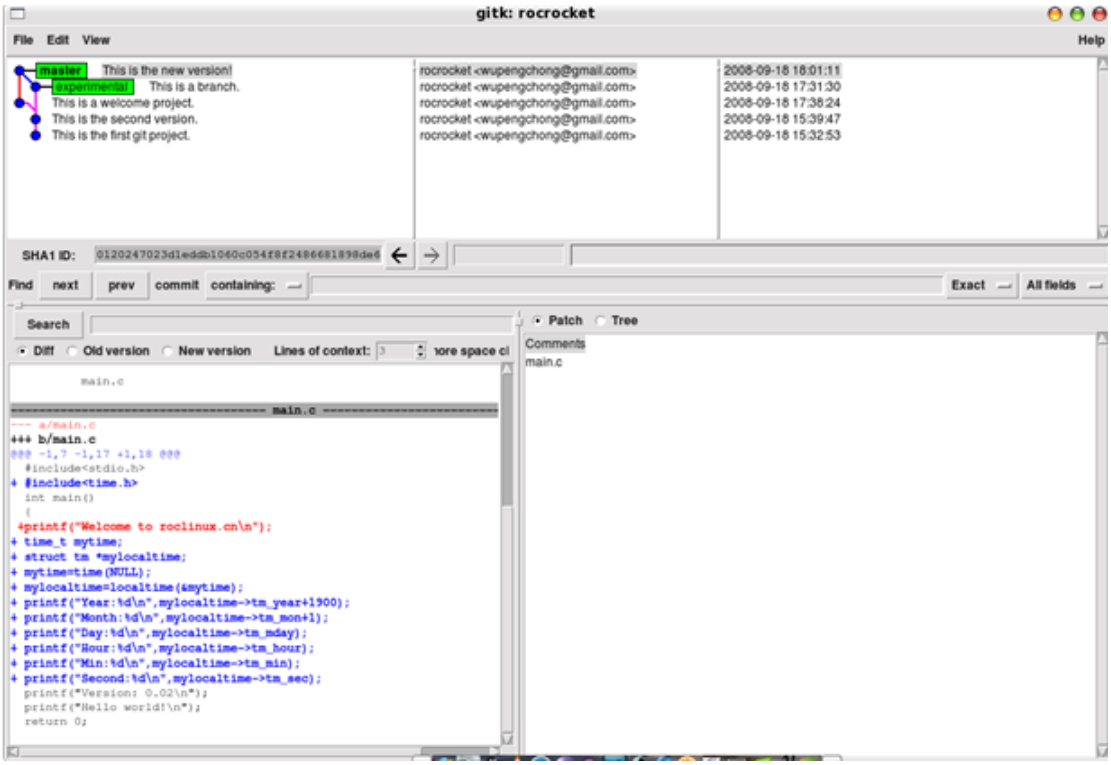
好，解决冲突后再次提交！

```
[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。



第七章　删除分支

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。

第八章　在GIT里面合作开发

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/rocrocket/git-
study/rocrocket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/rocrocket/git-study/rocrocket myrepo (有待商量)

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/rocrocket/git-study/rocrocket

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。

第九章　复习

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exerimental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。

如果我不是很信任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（rocrocket）的工作目录，它会直接找到我的目录来取。

第十章　历史记录查询

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:

[rocrocket@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!!!!!

2

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

[rocrocket@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:

[rocrocket@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:

[rocrocket@wupengchong project]$git show master

[rocrocket@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）:

[rocrocket@wupengchong project]$git show HEAD

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

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

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

[rocrocket@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:

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

[rocrocket@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:

[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

[rocrocket@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:

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

[rocrocket@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

[rocrocket@wupengchong project]$ git log V3..V7
//显示V3之后直至V7的所有历史记录

[rocrocket@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。

[rocrocket@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。

[rocrocket@wupengchong project]$ git log stable..experimental
//将显示在experimental分支但不在stable分支的历史记录

[rocrocket@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:

[rocrocket@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。