

Git 使用手册

杨彬

- git-use-v1.0 -

提交至: 杨彬
所有者: 杨彬
作者: 杨彬
编辑者: 杨彬
版本: 1.0
日期: 2014-01-08

© 2014 版权所有（杨彬保留所有权力）

本文件的内容涉及商业机密，未经允许，不得泄露给任何第三方。

文件修改履历

版本	修改日	修改人	注释
1.0	2012/08/24	杨彬	初稿
1.1	2014/01/04	杨彬	增加 windows GUI 配置
1.2	2014/01/06	杨彬	

目录

1 安装 Git.....	3
1.1 Windows 平台安装 Git.....	3
1.1.1 MsysGit 的配置.....	3
1.2 Linux 平台安装 Git.....	3
1.2.1 包管理器方式安装.....	3
1.2.2 从源代码进行安装.....	4
1.2.2.1 Centos 6.4 安装 Git.....	4
2 配置 Git 的全局属性.....	4
3 管理分支(branch).....	5
4 通过 git 开发团队项目.....	8
4.1 创建服务器端代码仓库.....	8
4.2 创建 A 的代码仓库.....	9
4.3 创建 B 的代码仓库.....	10
4.4 A 修改代码.....	10
4.5 B 提取代码.....	11
4.6 B 修改代码.....	13
4.7 A, B 同时修改代码.....	14
4.7.1 A 修改代码.....	14
4.7.2 B 修改代码.....	14
4.7.3 A 提取代码.....	16
5 浏览提交历史.....	17
6 附录.....	20
6.1 词汇表.....	20
6.2 引用.....	20

1 安装 Git

1.1 Windows 平台安装 Git

Windows 平台下使用比较广泛的 Git 图形工具叫做 msysGit。

下载地址为 <http://code.google.com/p/msysgit/downloads/list>

下载名为 Git-<版本号>-preview<日期>.exe 的软件包，如 Git-1.7.11-preview20120710.exe。

点击安装程序开始安装，不需要特别修改，一切默认就可以。

1.1.1 MsysGit 的配置

1. ls 不能显示中文目录

在 /etc/git-completion.bash 中增加一行：

```
alias ls='ls --show-control-chars --color=auto'
```

2. git log 中文乱码、git status 输出中文会显示为 UNICODE 编码

在 /etc/gitconfig 中添加如下配置：

```
[gui]
    encoding = utf-8

[i18n]
    logOutputEncoding = utf-8
    commitEncoding = utf-8

[svn]
    pathnameencoding = utf-8

[core]
    editor = vim
    quotePath = false
    fileMode = false
```

1.2 Linux 平台安装 Git

Linux 上安装 Git 有两种不同的方式：

- 1) 通过 Linux 发行版的包管理器安装
- 2) 通过源码包安装

1.2.1 包管理器方式安装

Ubuntu 10.10 以上版本、Debian (squeeze) 系统安装：

```
sudo apt-get install git
```

```
sudo apt-get install git-doc git-svn git-email git-gui gitk
```

Ubuntu 10.04 以下版本、Debian (lenny)以下版本：

```
sudo apt-get install git-core
```

```
sudo apt-get install git-doc git-svn git-email git-gui gitk
```

Fedora、CentOS 系统安装：

```
yum install git
```

```
yum install git-svn git-email git-gui gitk
```

其他发行版安装 Git 的过程类似。Git 软件包在这写发行版里称为 git，或 git-core。

1.2.2 从源代码进行安装

1.2.2.1 Centos 6.4 安装 Git

安装 Git 依赖包：

```
yum -y install zlib-devel openssl-devel perl cpio expat-devel gettext-devel gcc
```

```
yum -y install autoconf
```

```
yum -y install make
```

获取最新 Git 源码包：

```
wget http://www.codemonkey.org.uk/projects/git-snapshots/git/git-latest.tar.gz
```

解压并开始安装 Git:

```
tar xzvf git-latest.tar.gz
```

```
cd git-<版本号>
```

```
autoconf
```

```
./configure --with-curl=/usr/local
```

```
make
```

```
sudo make install
```

安装完毕后，查看 Git 版本号信息：

```
git --version
```

显示如下：

```
git version 1.8.4.GIT
```

2 配置 Git 的全局属性

在开始 git 学习之前，我们需要首先配置 git 的全局属性：

```
git config --global user.name "<your name>"
```

```
git config --global user.email <your email address>
```

例如:

```
git config --global user.name "yang bin"
```

```
git config --global user.email "yangbin@vungu.com"
```

打开配置文件:

```
cd ~
```

```
vi .gitconfig
```

添加以下配置:

```
[gui]
    encoding = utf-8

[i18n]
    logOutputEncoding = utf-8
    commitEncoding = utf-8

[svn]
    pathnameencoding = utf-8

[core]
    editor = vim
    quotePath = false
    filemode = false
```

3 管理分支(branch)

在 git 中, 一个代码仓库可以维护多个开发的分支(branch)。可以通过以下命令创建一个叫做"topic/a"的分支:

```
git branch topic/a
```

如果你运行

```
git branch
```

你会得到一个所有分支的列表:

```
* master
topic/a
```

我们可以看到"topic/a"是我们创建的分支, 而"master"是系统自动创建的默认分支。'*'标志出你当前所在的分支。

输入以下命令:

```
git checkout topic/a
```

此时, 我们切换到了 topic/a 分支。

现在编辑 file.txt, 提交修改, 再切换回 master 分支:

```
vi file.txt
```

做如下修改:

```
version 1.0
version 1.1 (modified)
version 1.2 (topic/a)
```

```
git commit -a -m "version 1.2 (topic/a)"
git checkout master
cat file.txt
```

系统显示如下信息：

```
version 1.0
version 1.1 (modified)
```

我们发现在 topic/a 中所做的修改不见了。这是为什么呢？原因在于我们刚才的修改是在 topic/a 分支上所做的，而我们现在已经切换回了 master 分支。

此时，我们可以在 master 分支上对 file.txt 做一个不同的修改：

```
vi file.txt
```

做如下修改：

```
version 1.0
version 1.1 (modified)
version 1.2 (master)
```

提交修改：

```
git commit -a -m "version 1.2 (master)"
```

此时，两个分支(master 与 topic/a)产生了分歧：每个分支包含不同的修改。如果需要将 topic/a 的代码合并到 master 分支中，我们可以执行以下命令：

```
git merge topic/a
```

如果两个分支的代码没有冲突，则 merge 会自动完成。如果合并的过程中出现冲突，则合并过程会中断。

在本例中存在代码的冲突，因此 git 会显示如下信息：

```
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

此时，我们可以先检查冲突的细节：

```
git status
```

Git 会显示以下信息：

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
```

```
# both modified: file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

我们发现 file.txt 发生了冲突。

此时，我们应该编辑 file.txt 并手工解决冲突：

```
vi file.txt
```

此时，我们应该编辑 file.txt 并手工解决冲突：

```
vi file.txt
```

此时 file.txt 的内容如下：

```
version 1.0
version 1.1 (modified)
<<<<<<< HEAD
version 1.2 (master)
=====
version 1.2 (topic/a)
>>>>>>> topic/a
```

其中，"<<<<<<< HEAD"与=====之间得内容为 master 分支中的内容=====与>>>>>>>topic/a 之间的内容为 topic/a 分支中的内容

我们对 file.txt 做如下修改：

```
version 1.0
version 1.1 (modified)
version 1.2 (master)
version 1.2 (topic/a)
```

完成修改之后将 file.txt 添加到 index 中并提交

```
git commit -a
```

此时系统会自动生成提交提示信息，通常我们不需要修改这个信息。

```
Merge branch 'topic/a'
```

```
Conflicts:
```

```
file.txt
```

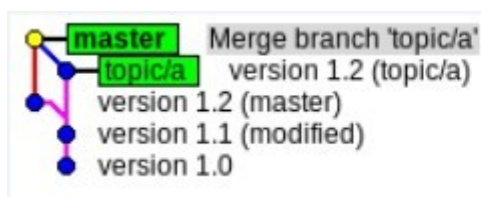
完成之后，git 会显示如下信息：

```
[master 8f76a0e] Merge branch 'topic/a'
```

我们还可以通过 gitk 命令查看历史记录：

```
gitk
```

这条命令会以显示历史记录的图示：

**注意：**

在本例中，因为发生了冲突，我们手动解决了冲突，并最终执行 `git commit` 来手动提交了更新，因此此时产生了一条新的提交记录（合并记录）。如果没有发生冲突，`git merge` 仍然会自动提交一条新的提交记录（合并记录）。

但是有一种特殊情况：如果 A 是 `topic/a` 的祖先节点（A 包含在 `topic/a` 中），那么 `git merge topic/a` 不会产生新的提交记录，此时，git 将直接将 A 移动到 `topic/a` 的位置。这种行为在 git 中被称为“fast forward”。

例如，如果当前的分支情况如下：

```

O -> O -> O -> O -> O (master)
      |
      O -> O -> O (A)*
            |
            t -> t1 -> t2 -> (topic/a)
  
```

当我们执行 `git merge A topic/a` 的时候，git 不会产生新的提交记录，而是直接将 A 移动到 `topic/a` 的末端位置：

```

O -> O -> O -> O -> O (master)
      |
      O -> O -> O
            |
            t -> t1 -> t2 -> (topic/a,A*)
  
```

此时，`topic/a` 分支已经完成了历史使命，因此我们可以将它删除：

```
git branch -d topic/a
```

Git 会显示以下信息：

```
Deleted branch topic/a (was c9a1949).
```

`git branch -d` 命令会首先检查将要删除的分支是否已经被合并到了其他分支中，如果没有，系统会报错。如果你只是用某个分支测试一些想法，之后发现想法不可行而希望将这个分支强制删除，则可以使用以下这条命令：

```
git branch -D <分支名称>
```


4 通过 git 开发团队项目

以上的例子中，你可能会产生一个疑问：所有的代码都是我一个人在开发，而且代码也是提交到本地，那么如果是一个团队协作开发一个项目，应该如何团队中分享代码呢？在本章中我们将讨论这种实践。

首先我们来作以下假设：

- 团队中有两位开发者：A 与 B
- A 与 B 都在自己的计算机上修改代码
- A 与 B 通过服务器上的代码仓库分享代码

出于学习目的，我们将在同一台计算机上模拟这个架构。

4.1 创建服务器端代码仓库

创建一个 git 用户并设置密码为 git：

```
adduser git
password git
```

切换到 git 用户并进入用户跟目录

```
su git
cd
```

创建 git-test 代码仓库

```
git --bare init git-test
```

Git 将输出以下信息：

```
Initialized empty Git repository in /home/git/git-test/
```

我们将把 git-test 作为我们的 git 服务器端代码仓库使用。

4.2 创建 A 的代码仓库

克隆命令：

```
git clone <user@host:目录> [<目录>]
```

克隆 A：

```
git clone git@218.246.35.250:/home/git/git-test A
```

输入 git 用户的密码

如果代码仓库是一个空的仓库，则会提示一个警告：

```
warning: You appear to have cloned an empty repository.
```

此时，我们将服务器端的 git-test 仓库下载到了本地的 A 目录中。

检查远端仓库的配置：

```
cd A
git remote
```

Git 将会显示以下信息

```
origin
```

git remote 命令的作用是显示本地所关联的远端仓库。因此，以上的输出意味着本地关联了一个叫做 origin 的远端仓库。那么 origin 的地址是什么呢？我们可以进一步检查远端仓库的配置：

```
cat .git/config
```

可以看到如下内容：

```
...
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = git@218.246.35.250:/home/git/git-test
...
```

origin 其实是一个远端仓库的别名，它实际的地址是有 .git/config 中的 url 来决定的。因此在 A 的仓库中，origin 这个别名实际上指向了 'git@218.246.35.250:/home/git/git-test' 这个服务器端的代码仓库。

4.3 创建 B 的代码仓库

克隆 B：

```
git clone git@218.246.35.250:/home/git/git-test B
```

此时，我们将服务器端的 git-server 仓库下载到了本地的 B 目录中。

4.4 A 修改代码

切换到 A 的目录下面：

```
cd A
vi file.txt
```

输入以下内容：

```
version 1.0 (A)
```

```
git add file.txt
```

```
git commit -a -m "version 1.0 (A)"
```

此时 file.txt 被提交到本地的代码仓库中。

git commit -a 是把当前修改过的或者已经删除的文件提交到代码仓库。

现在我们将本地的修改推送到服务器端的代码仓库：

```
git push
```

此时，git 将显示以下错误：

```
No refs in common and none specified; doing nothing.
```

Perhaps you should specify a branch such as 'master'.

fatal: The remote end hung up unexpectedly

error: failed to push some refs to 'git@218.246.35.250:/home/git/git-test'

这是因为我们第一次将本地的修改推送到服务器端，此时服务器端还不存在 master 分支。

我们需要通过以下的命令来重新推送：

```
git push origin master
```

Git 会显示以下信息：

```
Counting objects: 3, done.
```

```
Writing objects: 100% (3/3), 220 bytes, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
```

```
To git@218.246.35.250:/home/git/git-test
```

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

推送成功。

git push 命令的作用是将本地仓库中的分支推送到远端仓库并与远端分支合并，它的用法如下：

git push <远端仓库> <远端分支>或 git push

当使用 git push <远端仓库> <远端分支>的时候，git 将当前的本地分支推送到<远端仓库>中的<远端分支>例如：

```
git push origin master
```

意味着将当前分支推送到 origin 中的 master 分支。

当使用 git push 的时候，git 会使用哪个<远端仓库>与<远端分支>呢？

首先，我们再次查看.git/config 文件

```
cat .git/config
```

```
[branch "master"]
```

```
    remote = origin
```

```
...
```

我们发现对于本地的 master 分支，它的 remote 为 origin。这意味着，当推送本地的 master 分支的时候，git 会向 origin 仓库推送。但是要推送到 origin 仓库中的哪个分支呢？答案是：git 默认会推送到同名的分支中去，也就是说 git 会将本地的 master 分支推送到远端的 origin/master 中去。

因此，如果本地的分支为 topic/a，那么 git push 等价于 git push origin topic/a。

注意：

在大多数情况下，我们只需要使用 git push

4.5 B 提取代码

假设 A 推送代码之后通知 B 将服务器端的代码更新到本地，那么 B 可以通过以下操作来提取代码：

```
cd B
git pull
```

Git 会显示以下信息：

```
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@218.246.35.250:/home/git/git-test
* [new branch]    master -> origin/master
```

此时我们发现，在 B 的目录中出现了 file.txt，而且 file.txt 的内容正是 A 推送的内容。

```
cat file.txt
version 1.0(A)
```

git pull 命令的作用是将远端仓库的远端分支下载到本地，并将下载的远端分支合并到本地分支中。它的用法如下：

git pull <远端仓库> <远端分支>或 git pull

当使用 **git pull <远端仓库> <远端分支>** 的时候，git 将<远端仓库>/<远端分支>提取到本地，然后与当前分支合并。例如：

```
git pull origin master
```

意味着 git 会：1) 将 origin/master 下载到本地，2) 再将 origin/master 与当前分支合并

当使用 **git pull** 的时候，git 会从使用哪个<远端仓库>与<远端分支>呢？首先，我们再次查看.git/config

```
cat .git/config

[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = git@218.246.35.250:/home/git/git-test
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

remote = origin,意味着当前分支(master)会从 origin 远端仓库提取分支。

fetch = +refs/heads/*:refs/remotes/origin/*, 意味着会将远端仓库中所有位于 refs/heads 目录下的分支下载到本地的 refs/remotes/origin 目录下面

merge = refs/heads/master, 意味着 git 会将远端仓库中 refs/heads 目录下的 master 分支与当前分支合并。

因此，在默认情况下，当使用 git pull 时，git 会：1) 从当前分支 remote 属性指定的远端仓库下载所有的分支 2) 将当前分支 merge 属性所指定的远端分支合并到当前分支。

在本例中，git pull 等价于 git pull origin master。

注意：

在大多数情况下，我们只需要使用 git pull

我们可以用 git branch 命令显示所有的本地分支

```
git branch
* master
```

我们可以用 git remote 命令显示所有的远端仓库。

```
git remote
origin
```

git branch -r 来显示所有下载到本地的远端分支。

```
git branch -r
origin/master
```

现在，我们再来查看.git/config 文件

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = git@218.246.35.250:/home/git/git-test
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

[remote "origin"] – 表示此处配置一个远端仓库，本地名称为 origin

fetch = +refs/heads/*:refs/remotes/origin/* - 表示在本地执行 git pull origin 的时候，git 会先将远端仓库的 refs/heads/* (相对于远端仓库而言，这是它的本地分支) 下载到本地的 refs/remotes/origin/* 中。

url = git@218.246.35.250:/home/git/git-test - 表示 origin 远端仓库的内容来源于 git@218.246.35.250:/home/git/git-test

[branch "master"] - 表示此处配置一个本地分支，分支名称为 master

remote = origin - 表示本地的 master 分支与 origin 远端仓库关联，因此：git push 将向 origin.url/refs/heads 推送，而 git pull 将从 origin.url/refs/heads 下载

merge = refs/heads/master - 表示在 master 分支执行 git pull 的时候，git 会将 origin.url/heads/master 与本地的 master 合并。我们知道，因为 git pull 会首先将 origin.url/refs/heads/master 下载到本地的 refs/remotes/origin/heads/master,因此 git 实际上是将 refs/remotes/origin/heads/master 与本地的 master 分支合并

4.6 B 修改代码

现在我们来模拟 B 修改代码的场景：

```
cd B
vi file.txt
```

做如下修改：

```
version 1.0 (A)
version 1.1 (B)
```

```
git commit -a -m "B v1.1"
```

Git 会显示以下信息：

```
[master b0e2f52] version 1.1 (B)
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在我们将本地的修改推送到服务器端的代码仓库：

```
git push
```

Git 会显示以下信息：

```
Counting objects: 5, done.
Writing objects: 100% (3/3), 271 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To git@218.246.35.250:/home/git/git-test
d4738f5..b0e2f52 master -> master
```

4.7 A, B 同时修改代码

4.7.1 A 修改代码

```
cd A
vi file.txt
```

做如下修改：

```
version 1.0 (A)
version 1.1 (B)
version 1.2 (A)
```

```
git commit -a -m "version 1.2 (A)"
git push
```

4.7.2 B 修改代码

```
cd B
vi file.txt
```

做如下修改：

```
version 1.0 (A)
version 1.1 (B)
version 1.2 (B)
```

```
git commit -a -m "version 1.2 (B)"
git push
```

Git 会现实以下错误信息：

```
To /home/binyang/git-test/git-server
! [rejected]      master -> master (non-fast forward)
error: failed to push some refs to '/home/binyang/git-test/git-server'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again.  See the 'non-fast forward'
section of 'git push --help' for details.
```

这个错误意味着，git-server 中的 refs/heads/master 的版本比本地的 refs/heads/master 的版本新，因此不允许推送。

此时，我们应该尝试提取服务器上最新的代码并将代码合并到本地的分支中：

```
git pull

remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@218.246.35.250:/home/git/git-test
   b0e2f52..4e626b3  master    -> origin/master
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

我们发现在合并代码的时候，发生了冲突"CONFLICT"，因此合并被终止了。

此时，我们首先应该查看哪些文件发生了冲突：

```
git status

# On branch master
# Your branch and 'origin/master' have diverged,
```

```
# and have 1 and 1 different commit(s) each, respectively.
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       unmerged:   file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

我们发现 file.txt 发生了冲突，因此必须手动编辑 file.txt 并解决冲突：

```
vi file.txt
```

内容如下：

```
version 1.0 (A)
version 1.1 (B)
<<<<<<< HEAD
version 1.2 (B)
=====
version 1.2 (A)
>>>>>> 4e626b314b1f79ebc38c9121aa27dea4a2df4166
```

将代码修改如下：

```
version 1.0 (A)
version 1.1 (B)
version 1.2 (B)
version 1.2 (A)
```

将修改加入 index 并提交

```
git add file.txt
git commit
```

Git 会显示如下信息：

```
[master aa1e89f] Merge branch 'master' of git@218.246.35.250:/home/git/git-test
```

将修改推送到远端仓库

```
git push
```

Git 会显示如下信息：

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 593 bytes, done.
```



```
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To /home/binyang/git-test/git-server
 4e626b3..aa1e89f master -> master
```

4.7.3 A 提取代码

```
cd ~/git-test/A
git pull
```

Git 会显示如下信息：


```
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/binyang/git-test/git-server
 4e626b3..aa1e89f master -> origin/master
Updating 4e626b3..aa1e89f
Fast forward
 file.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

注意：

更新合并成功，因为 B 已经解决了冲突并将解决后的版本推送到了服务器。

运行 gitk 查看版本树：

```
gitk
```



Merge branch 'master' of /home/binyang/git-test/git-server

5 浏览提交历史

Git 的历史记录是由一系列彼此关联的提交(commit)记录构成的。我们已经学习了通过 git log 或者 gitk 可以显示这些提交记录。此外，与传统的版本管理工具不同，git 不是通过版本号来表示提交记录的。相反 Git 为每一个提交记录分配一个 40 位的 16 进制数字来标识它，不难理解，这串数字一定是唯一的。

在任何时刻，你都可以通过以下命令查看提交的历史记录

```
git log
```

如果你希望同时显示每一步的修改内容，可以使用：

```
git log -p
```

其中 p 是"patch"的缩写，意思是输出补丁。

很多时候，显示历史记录的概要内容就可以帮助我们大概了解每一次提交所做的改动：

```
git log --stat --summary
```

例如：

```
cd ~/git-test/A
```

```
git log
```

Git 会显示以下内容：

```
commit aa1e89f0ab6085f7cafc34cf970837b2ffdda2e
Merge: b8f4754 4e626b3
Author: yangbin <bin.yang@zerustech.com>
Date: Tue Nov 29 16:50:29 2011 +0800

    Merge branch 'master' of /home/binyang/git-test/git-server

Conflicts:
    file.txt

commit b8f475421d4403be97c71465ee5a9fe83bcb063d
Author: yangbin <bin.yang@zerustech.com>
Date: Tue Nov 29 16:43:34 2011 +0800

    version 1.2 (B)
...
```

我们看到这里显示了 2 条提交(commit)记录：它们的标示符分别为：

aa1e89f0ab6085f7cafc34cf970837b2ffdda2e

与

b8f475421d4403be97c71465ee5a9fe83bcb063d

我们可以通过 git show 命令来查看某个提交的具体内容：

```
git show aa1e89f0ab6085f7cafc34cf970837b2ffdda2e
```

Git 会显示如下信息：

```
commit aa1e89f0ab6085f7cafc34cf970837b2ffdda2e
```

```
Merge: b8f4754 4e626b3
Author: yangbin <bin.yang@zerustech.com>
Date: Tue Nov 29 16:50:29 2011 +0800

Merge branch 'master' of /home/binyang/git-test/git-server
```

```
Conflicts:
    file.txt
```

```
diff --cc file.txt
index ebf14d7,74d44f2..20f8106
--- a/file.txt
+++ b/file.txt
@@@ -1,3 -1,3 +1,4 @@@
    version 1.0 (A)
    version 1.1 (B)
+version 1.2 (B)
+ version 1.2 (A)
```

除了使用上述的字符串访问提交记录外，我们还可以使用以下方法：

- 使用字符串前导部分，只要这部分字符串足够长不会与其他的提交记录重复：

```
git show aa1e
```

- 使用 HEAD 变量：

```
git show HEAD
```

#HEAD 是一个系统的变量（我们称之为分支末端），它的值是当前分支中最后一条提交记录的标示符
#因此，默认情况下，git show HEAD 等价于 git show

- 使用分支名：

```
git show master
```

#在 git 中，分支名等价于分支最后一条提交记录的标示符

我们可以通过一下方式来访问提交记录的祖先节点：

```
#显示 HEAD 的父节点
```

```
git show HEAD^
```

```
#显示 HEAD 的 2 级祖先节点
```

```
git show HEAD^^
```

```
#显示 HEAD 的 4 级祖先节点
```

```
git show HEAD~4
```

我们也可以为某条提交记录添加一个标签(tag)

```
git tag B-v1.1 1071
```

此后，我们可以通过 B-v1.0 来访问 1071...这条提交记录

例如：

```
#显示 B-v1.1 与当前分支顶端之间的差异
```

```
git diff B-v1.1 HEAD
```

```
#基于 B-v1.1 创建一个名称为 topic/a 的分支
```

```
git branch topic/a B-v1.1
```

6 附录

6.1 词汇表

词汇	解释

附录 1: 词汇表

6.2 引用

引用	注释
Git 安装	http://blog.haohtml.com/archives/10093

附录 2: 引用