# 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: 2012-12-06 |



# 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 |
| | | | |

# 目录

1 安装 Git....4
1.1 Windows 平台安装 Git....4
1.1.1 msysGit 的配置....4
1.2 Linux 平台安装 Git....4
1.2.1 包管理器方式安装....4
1.2.2 从源代码进行安装....5
2 配置 Git 的全局属性....5
3 通过 git 开发团队项目....5
3.1 创建服务器端代码仓库....5
3.2 创建 A 的代码仓库....6
3.3 创建 B 的代码仓库....6
3.4 A 修改代码....6
3.5 B 提取代码....8
3.6 B 修改代码....10
3.7 A，B 同时修改代码....11
3.7.1 A 修改代码....11
3.7.2 B 修改代码....11
3.7.3 A 提取代码....13
4 浏览提交历史....14
5 管理分支(branch)....16
6 Git 详解....20
6.1 Git 对象数据库....20
6.2 Git 提交记录的访问方法....28
6.2.1 通过 SHA1 访问....28
6.2.2 通过分支名访问....28
6.2.3 通过 HEAD 访问....28
6.2.4 通过标签访问....28
6.2.5 通过'^'访问....28
6.2.6 通过'^[n]'访问....29
6.2.7 通过'~'访问....29
6.2.8 通过'..'访问....29
6.2.9 通过'...'访问....30
7 Git 常用命令....30
7.1 初始化本地项目....30
7.2 下载远端项目....30
7.3 更新 index....30
7.4 查看工作目录状态....31
7.5 撤销 git add 操作....31

7.6 撤销本地操作.....31
7.7 提交修改.....31
7.8 浏览提交历史.....32
7.9 显示某个提交记录的内容.....32
7.10 显示版本树图形界面.....32
7.11 显示差异.....32
7.12 生成 patch.....33
7.13 显示本地分支.....33
7.14 创建本地分支.....33
7.15 删除本地分支.....33
7.16 强制删除本地分支.....33
7.17 显示远端分支.....34
7.18 切换分支.....34
7.19 显示远端仓库.....34
7.20 推送代码.....34
7.21 提取并更新代码.....34
7.22 提取代码.....34
7.23 合并分支.....34
7.24 调查 git 对象.....34
7.25 调查 tree 对象.....35
7.26 内容检索.....35
7.27 移动文件/目录.....35
7.28 删除文件/目录.....35
7.29 显示所有标签.....35
7.30 新建本地标签.....35
7.31 删除本地标签.....36
7.32 创建远端标签.....36
7.33 推送远端标签.....36
7.34 提取远端标签.....36
7.35 撤销某条提交记录.....36
7.36 获得某个文件的历史版本.....37
7.37 计算文件的 SHA1 字符串.....37
7.38 显示 git 配置.....37
7.39 获得 git 帮助.....37
8 附录.....37
8.1 词汇表.....37
8.2 引用.....38

# 1 安装 Git

## *1.1 Windows 平台安装 Git*

Windows 平台下图形版的 Git 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. git commit 不能提交中文注释

2. ls 命令现实中文文件名
   在安装目录下找到 /etc/profile，打开文件，在最末尾加一行：
   alias ls=”ls --show-control-chars”

## *1.2 Linux 平台安装 Git*

Linux 上安装 Git 是非常方便的。可以有两种不同的方式在 Linux 上安装 Git：一种方法是通过 Linux 发行版的包管理器安装

### 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

# 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"
```

# 3 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，应该如何在团队中分享代码呢？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。

## *3.1 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。

## 3.2 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。

## 3.3 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。

## 3.4 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

## *3.5 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

## *3.6 B 修改代码*

现在我们再模拟 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
```

# 3.7 A，B 同时修改代码

## 3.7.1 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
```

## 3.7.2 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
```

### 3.7.3 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(-)
```

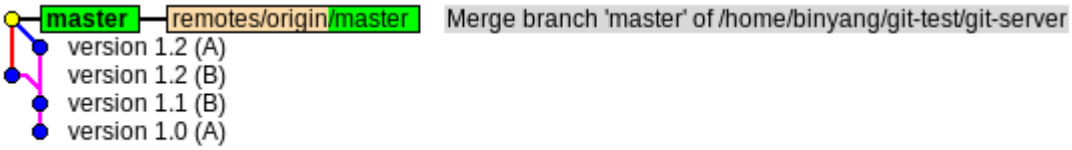
**注意：**

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
```




# 4 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 进制数字来学标识它，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 aa1e89f0ab6085f7cafc34cf970837b2fffdda2e
efiMerge: 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：

aa1e89f0ab6085f7cafc34cf970837b2fffdda2e

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 aa1e89f0ab6085f7cafc34cf970837b2fffdda2e
```

Git 会显示如下信息：

```
commit aa1e89f0ab6085f7cafc34cf970837b2fffdda2e
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
```

# 5 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 (modiffied)
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 (modiffied)
```

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 (modiffied)
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 分支众，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 (modiffied)
<<<<<<< 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 (modiffied)
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
```

这条命令以会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 中被成为"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>
```

# 6 Git 详解

在之前的培训中，我们了解了 git 的基本用法，也对 git 的系统文件有了一些初步的了解。在本章中我们将详细解读 git 的工作原理与各种系统文件。

## *6.1 Git 对象数据库*

在 git 中，文件，目录，提交记录都被作为对象管理。在本章中，我们将通过实际的例子学习 git 如何管理这些对象。

我们线创建一个新的本地项目，再模拟一些提交：

```
cd ~/git-test
mkdir git-details
cd git-details
git init
touch file1.txt
vi file1.txt
```

输入以下内容

```
version 1.0 (file1.txt)
```

```
mkdir test001
mkdir -p test002/test003
touch test001/file2.txt
vi test001/file2.txt
```

输入以下内容

```
version 1.0 (file2.txt)
```

```
touch test002/test003/file3.txt
vi test002/test003/file3.txt
```

输入以下内容

```
version 1.0 (file3.txt)
```

```
git add  *
git commit -a -m "version 1.0 (master)"
```

Git 会显示以下信息：

```
[master (root-commit) 0b5fc8a] version 1.0 (master)
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 test001/file2.txt
 create mode 100644 test002/test003/file3.txt
```

```
vi file1.txt
```

做如下修改：

```
version 1.0 (file1.txt)
version 1.1 (file1.txt)
```

```
vi test001/file2.txt
```

做如下修改：

```
version 1.0 (file2.txt)
version 1.1 (file2.txt)
```

```
vi test002/test003/file3.txt
```

做如下修改：

```
version 1.0 (file3.txt)
version 1.0 (file3.txt)
```

```
git commit -a -m "version 1.1 (master)"
```

Git 会显示以下信息：

```
[master ca16cbb] version 1.1 (master)
 3 files changed, 3 insertions(+), 0 deletions(-)
```

当提交的时候，系统显示 7 位 16 进制的数字（ca16cbb）是什么呢？

我们在之前的学习中了解到，在 git 中每一条提交记录都有唯一的一个 40 位 16 进制的数字来表示它，同时，只要不发生冲突，我们可以使用提交记录标识符的前导字符串来标识它。这里的 7 位 16 进制数字就是提交记录标示符的前导字符串。

在 git 中，每一条提交记录都有一个这样的标识符。提交记录的标识符是基于本次提交所包含的所有文件的内容，父节点的标示符，系统实践，用户名等综合信息所生成的一个 SHA1 字符串，这可以保证 git 永远不会生成重复的提交记录标识符（例如，两个不同的开发者即使在相同时间在各自的计算机上提交了相同的代码，git 也会为他们的提交记录生成不同的 SHA1 标识符，因此这两条提交记录也会被 git 理解为两个不同的"版本"）。

那么 git 的提交记录保存在什么位置呢？我们在之前的学习了解到在 git 中有 git/objects 这样一个目录。事实上，git 的提交记录就保存在这个目录中。

以 ca16cbb 这条提交记录为例，它的具体位置是：

```
.git/objects/ca/16cbb658f746519c571d6af3e76b554a85bd03
```

**注意：**

为了叙述方便，位于.git/objects 下的文件，我们成为对象。

我们来查看一下这个对象的内容：

```
cat .git/objects/ca/16cbb658f746519c571d6af3e76b554a85bd03
```

我们发现这个对象的内容是二进制的，因此我们不能理解它的内容。

我们可以通过以下的命令来调查它：

```
git cat-file -t ca16
```

Git 会显示以下信息：

```
commit
```

git cat-file -t <SHA1>命令会显示与<SHA1>对应的对象在 git 中的类型，因此 ca16 的类型是'commit'。这是合理的，因此 ca16 代表了一条提交记录(commit)。

进一步，我们可以查看这条提交记录的内容：

```
git cat-file commit ca16
```

Git 会显示以下信息：

```
tree 065d62c67af78a7cfbf472cb787a59111f0062b3
parent 0b5fc8a3a69609eb728247c17f80e034a8b26996
author yangbin <bin.yang@zerustech.com> 1324345351 +0800
committer yangbin <bin.yang@zerustech.com> 1324345351 +0800

version 1.1 (master)
```

git cat-file <TYPE> <SHA1>命令会将<SHA1>对应的对象作为<TYPE>类型显示。

这就是 ca16 这条提交记录的全部内容。我们不难理解 author，committer，以及结尾处的说明文字的意思。

那么第一行的"tree 065d62c67af78a7cfbf472cb787a59111f0062b3"是什么意思呢？

不难猜测，065d...这串 16 进制的数字是某个对象的标识符。那么 tree 又是什么呢？事实上 tree 是 065d62c67af78a7cfbf472cb787a59111f0062b3 这个对象的类型。

不难确定，这个对象确实存在：

.git/objects/ca/16cbb658f746519c571d6af3e76b554a85bd03，并且它的内容也是二进制的。

因此，现在我们得出以下几个结论与猜想：

1. 在 git 中每一条提交记录都具有一个唯一的标识符（结论）
2. 提交记录被作为二进制文件保存在.git/objects 目录下（结论）
3. git 中有多种类型的对象（结论）
4. 提交记录是 git 中的一种对象，它的类型为 commit（结论）
5. 每个对象都有唯一的 SHA1 字符串与之对应（猜想，因为我们还不知道 tree 对象的 SHA1 如何生成）
6. 提交记录只包含最基本的信息，它的具体内容保存在一个 tree 类型的对象内（猜想）

为了验证以上的猜想，我们来继续调查 tree 类型对象的内容。

我们可以通过以下命令来查看：

```
git ls-tree 065d
```

Git 会显示以下内容：

```
100644 blob f29f72ce40cc8d5ccf1af4c49b18a47efd24a29f    file1.txt
040000 tree bb421c14d19fe2fa53175e122273444b9a8cf5ec    test001
040000 tree 34f04b3083af687dbbaf004084f15b98a6ffb305    test002
```

不难发现，这与我们的 6 号猜想很接近了。因为，065d 这个 tree 对象的内容确实包含了 file1.txt，test001 目录与 test002 目录。但是我们需要进一步验证 file1.txt，test001 与 test002 的内容是否与我们提交的内容一致。因而我们可以继续调查：

```
git cat-file blob f29f
```

Git 会显示以下信息：

```
version 1.0 (file1.txt)
version 1.1 (file1.txt)
```

这正是我们所提交的 file1.txt 的代码。

```
git ls-tree bb42
```

Git 会显示以下信息：

```
100644 blob 1310d13d1352f121f8a0d5100bc2045ed3a55759    file2.txt
```

这正是我们所提交的 test001 目录中所包含的文件名。

```
git cat-file blob 1310
```

Git 会显示以下信息：

```
version 1.0 (file2.txt)
```

```
version 1.1 (file2.txt)
```

这正式我们所提交的内容

```
git ls-tree 34f0
```

Git 会显示以下信息：

```
040000 tree 243d9b620637586cc1df39a43e41d04b9063a45b    test003
```

这正是我们所提交的 test002 目录所包含的子目录名。

```
git ls-tree 243d
```

Git 会显示以下信息：

```
100644 blob d645356afc8e719a609f516090b1cf0c2dc300a3    file3.txt
```

这正是我们所提交的 test003 目录中所包含的文件名。

```
git cat-file blob d645
```

Git 会显示以下信息：

```
version 1.0 (file3.txt)
version 1.1 (file3.txt)
```

至此，我们已经验证了我们的 6 号猜想“提交记录只包含最基本的信息，它的具体内容保存在一个 tree 类型的对象内”。

进一步分析我们可以理解：git 中的 tree 对象对应提交的代码中的目录，blob 对象对应提交的代码中的文件。tree 对象可以包含 blob 对象（目录中的文件）与 tree 对象（目录中的子目录）。

那么对于 5 号猜想“每个对象都有唯一的 SHA1 字符串与之对应”，我们如何验证呢？

为此，我们可以做以下的测试：

```
cd ~/git-test/git-details/
git hash-object file1.txt
```

Git 会输出以下信息：

```
f29f72ce40cc8d5ccf1af4c49b18a47efd24a29f
```

将 file1.txt 复制为 test003/file2.txt

```
mkdir test003
cp file1.txt test003/file2.txt
git hash-object test003/file2.txt
```

Git 会输出以下信息：

```
f29f72ce40cc8d5ccf1af4c49b18a47efd24a29f
```

git hash-object <文件>命令会计算文件在 git 中的 SHA1 标识符。

以上的测试告诉我们，对于文件而言，无论文件的名称如何修改，位于哪个目录中，只要文件的内容不便，它的 SHA1 就是相同的。同样不难测试，只要文件的内容不同，他们的 SHA1 就是不同的。

因此，我们可以得出一个结论：在 git 中，blob 类型对象的 SHA1 只与文件的内容有关，相同的文件内容永远产生相同的 SHA1，而不同的文件内容永远产生不同的 SHA1。

**注意：**

这意味着如果我们将两个内容相同的文件提交到 git 中，他们实际上会被保存为同一个 blob 对象。

我们来验证这一点：

```
cd ~/git-test/git-details/
git add test003
git commit -a -m "version 1.2 (master)"
git log
```

Git 会输出以下内容：

```
commit 3898c15df2fd2330e84b6a2df821cf74827fb0a5
Author: yangbin <bin.yang@zerustech.com>
Date:   Wed Dec 21 10:11:24 2011 +0800

    version 1.2 (master)

commit ca16cbb658f746519c571d6af3e76b554a85bd03
Author: yangbin <bin.yang@zerustech.com>
Date:   Tue Dec 20 09:42:31 2011 +0800

    version 1.1 (master)

commit 0b5fc8a3a69609eb728247c17f80e034a8b26996
Author: yangbin <bin.yang@zerustech.com>
Date:   Tue Dec 20 09:33:30 2011 +0800

    version 1.0 (master)
```

```
git ls-tree 3898
```

Git 会显示以下信息：

```
100644 blob f29f72ce40cc8d5ccf1af4c49b18a47efd24a29f    file1.txt
040000 tree bb421c14d19fe2fa53175e122273444b9a8cf5ec    test001
040000 tree 34f04b3083af687dbbaf004084f15b98a6ffb305    test002
040000 tree 6ab4500313816ab92ec597b3017848df57769ae9    test003
```

```
git ls-tree 6ab4
```

Git 会显示以下信息：

```
100644 blob f29f72ce40cc8d5ccf1af4c49b18a47efd24a29f    file2.txt
```

我们可以发现：file.txt 与 test003/file2.txt 对应同一个 blob 对象。此外，我们还发现了以下不寻常的地方：

1. 我们在提交"version 1.2"的时候实际上只添加了"test003"对吗?为什么 git ls-tree 会显示 file1.txt,test001 和 test002 呢？

实际上一条提交记录是工作目录中所有文件的一个“快照”，对于那些没有修改的文件，“快照”直接引用.git/objects 下那些已经存在的对象，因此并不占用额外的磁盘空间，也不会增加额外的网络传输。

对于那些新增的或者修改过的文件，“快照”引用.git/objects 下新创建的对象。

2. 为什么 git ls-tree <提交记录的 SHA1>也可以显示正确的内容？

因为每条提交记录与唯一的一个 tree 对象关联，因此 git 允许我们直接对提交记录的 SHA1 使用 git ls-tree 命令。这当然是很便利的。

那么 tree 类型的 SHA1 是如何计算的呢？事实上，tree 类型的 SHA1 只与它包含的文件的 SHA1，文件名以及子目录的 SHA1 有关。（因为 git hash-object 不支持目录，我们可以通过调查提交记录来验证这一点）。

至此，我们已经验证了 5 号猜想“每个对象都有唯一的 SHA1 字符串与之对应”。

现在我们已经完全了解 git 是如何管理一条提交记录以及它所包含的内容了。那么 git 又是如何管理提交记录之间的关联关系呢？我们来做如下的调查：

```
cd ~/git-test/git-details/
git log
```

Git 会显示以下信息：

```
commit 3898c15df2fd2330e84b6a2df821cf74827fb0a5
Author: yangbin <bin.yang@zerustech.com>
Date:   Wed Dec 21 10:11:24 2011 +0800

    version 1.2 (master)

commit ca16cbb658f746519c571d6af3e76b554a85bd03
Author: yangbin <bin.yang@zerustech.com>
Date:   Tue Dec 20 09:42:31 2011 +0800

    version 1.1 (master)

commit 0b5fc8a3a69609eb728247c17f80e034a8b26996
Author: yangbin <bin.yang@zerustech.com>
Date:   Tue Dec 20 09:33:30 2011 +0800
```

```
    version 1.0 (master)
```

现在我们查看以下与"version 1.2(master)"对应的这条提交记录：

```
git cat-file commit 3898
```

Git 会显示以下信息：

```
tree 20a5c85b360d268ce7757de3268600a3a57ef59d
parent ca16cbb658f746519c571d6af3e76b554a85bd03
author yangbin <bin.yang@zerustech.com> 1324433484 +0800
committer yangbin <bin.yang@zerustech.com> 1324433484 +0800


version 1.2 (master)
```

parent ca16cbb658f746519c571d6af3e76b554a85bd03，说明它父节点的 SHA1 是 ca16，我们知道，在本例中 ca16 代表了"version 1.0(master)"这条提交记录。Git 正是通过 commit 对象的 parent 属性将独立的 commit 对象关联起来，从而构成了分支的版本树。

**注意：**

如果某个 commit 对象是 merge 操作（多个分支的合并），则它会有多个 parent 属性。

我们来对以上的分析总结以下：

- 在 git 中，提交记录，文件，目录都被作为对象保存在.git/objects 目录中
- 每个对象有一个惟一的 SHA1 标识符
- 每提交一次，git 就会在.git/objects 下创建一个新的 commit 类型的对象
  - *commit 类型对象的 SHA1 与本次提交的内容，系统时间，用户信息，父节点的 SHA1 等信息有关，因此 git 永远不会产生重复的 commit 类型对象的 SHA1。（这一点可以保证两个不同的用户即使在相同实践在各自的计算机上提交相同的内容，git 也会生成两个不同的“版本”）*
  - *commit 类型对象只包含关于本次提交的基本信息，提交的具体内容保存在另外一个 tree 类型的对象中*
  - *与 commit 类型对象直接关联的 tree 对象代表了代码的根目录*
- tree 对象可以包含 bolb 对象（目录中的文件）与 tree 对象（目录中的子目录）
- tree 对象的 SHA1 与目录中包含的文件的 SHA1，文件名以及子目录的 SHA1 有关。
- tree 对象的 SHA1 只与文件的内容有关，因此内容相同的文件共享同一个 blob 对象
- 每次提交，对于没有修改的文件或目录，commit 对象将直接引用现存的 blob 或 tree 对象
- 每次提交，对于新增/修改的内容，新的 blob 或 tree 对象会被创建并保存在.git/objects 目录中

下图演示了 git 如何管理各种对象：

我们通过图片展示了本例中"version 1.1"与"version 1.2"这两条提交记录中各种对象之间的关系。其中“实线”箭头表示关联到新创建的对象。而“虚线”箭头表示关联到已经存在的对象。

## *6.2 Git 提交记录的访问方法*

在 git 中的很多命令需要使用一条或多条提交记录(commit)作为参数，而最容易令初学者感到困惑的就是 git 灵活的提交记录访问方法。在本章我们简单总结一些最常用的访问方法：

### 6.2.1 通过 SHA1 访问

这是最原始的用法

例：

```
git show abcd
```

### 6.2.2 通过分支名访问

默认情况下，分支名与分支的末端（最后一条提交记录）等价

例：

```
git log master
```

### 6.2.3 通过 HEAD 访问

HEAD 等价于当前分支的末端。

例：

```
git log HEAD
```

### 6.2.4 通过标签访问

例：

```
git show version-1.10
```

### 6.2.5 通过'^'访问

例：

```
git show HEAD^
git show HEAD^^
git show master^
git show master^^
git show version-1.10^
git show version-1.10^^
```

### 6.2.6 通过'^[n]'访问

例：

```
#访问第一个父节点
git show HEAD^1

#访问第二个父节点
git show master^2

#访问 2 级祖先节点的第二个父节点
git show master^^^2
```

### 6.2.7 通过'~'访问

例：

```
git show HEAD~3
git show master~3
git show version-1.1.0~3
```

### 6.2.8 通过'..'访问

A..B 将筛选出这样的提交记录：包含在 B 所在的分支中，但是不包含在 A 所在的分支中。

例：

```
git log -p master..origin/master
git log -p master^^..master

#如果当前分支为 master，则以下命令与 git log -p master^^..master 等价
git log -p master^^..
```

### 6.2.9 通过'...'访问

A...B 将筛选出这样的提交记录：或包含在 A 所在的分支中，或包含在 B 所在的分支中，但是不同时包含在 A 与 B 所在的分支中。

例：

```
git log -p master...origin/master
git log -p master^^...master
git log -p version-1.1.0~3...master
```

# 7 Git 常用命令

现在，我们已经对 git 有了比较系统的了解。在本章中我们将总结一些常用的 git 命令：

## *7.1 初始化本地项目*

```
git init
```

## *7.2 下载远端项目*

git clone <远端仓库 URL> [本地路径]

例：

```
#将本地的 test001 项目下载为 test002 项目
git clone test001 test002

#将远端的 git-tutorial 项目下载为本地的同名项目
git clone git@git.gforge.zerstech.com:git-tutorial
```

## *7.3 更新 index*

将工作目录中的文件/目录的内容更新到 index 中

git add <路径>

例：

```
#将 file.txt 更新到 index 中
git add file.txt

#将所有内容更新到 index 中
git add *
```

## 7.4 查看工作目录状态

```
git status
```

## 7.5 撤销 git add 操作

如果将某个文件/目录更新到了 index 中，在执行 git commit 之前，如果希望将文件/目录从 index 中移除，则可以执行以下操作：

git reset HEAD <路径>

例：

```
git reset HEAD file.txt
```

**注意：**

如果代码已经通过 git commit 提交，则不能执行此操作。

## 7.6 撤销本地操作

如果在本地修改了某个文件之后，希望撤销自己的修改（将文件恢复为最近一次提交的状态），则可以使用以下操作：

git checkout -- <路径>

例：

```
git checkout -- file.txt
```

**注意：**

如果代码已经通过 git commit 提交，则不能执行此操作。

## 7.7 提交修改

git commit

git commit -a

git commit -a -m "提交说明"

例：

```
#提交 index 内容
git commit

#将修改更新到 index，然后提交。忽略新增的文件
git commit -a

#提交的同时输入提交说明
```

```
git commit -a -m "version 1.0"
```

## 7.8 浏览提交历史

git log <提交记录>

例：

```
#显示当前分支的所有提交历史
git log

#显示本地 master 与远端 master 之间的不同的提交记录
git log -p master..origin/master

#显示 master 三级父节点之前的提交历史
git log master~3..
```

## 7.9 显示某个提交记录的内容

git show <提交记录>：<文件>

例：

```
git show abcd
git show master
git show version-1.1.0
git show version-1.1.0:file.txt
git show version-1.1.0:test001/test002/file.txt
```

## 7.10 显示版本树图形界面

gitk <提交记录>

例：

```
gitk
gitk master..origin/master
gitk master..topic/a
gitk master^^..
```

## 7.11 显示差异

git diff <旧提交记录>..<新提交记录>

例：

```
git diff release..master

#如果当前分支是 master，则等价于 git diff release..master
git diff release

git diff master^^..master
```

## 7.12 生成 patch

git format-patch <旧提交记录>..<新提交记录>

例：

```
#生成 release 分支与 master 分支之间的 patch 文件
#这些 patch 可以用来将 master 分支的修改更新到 release 分支
git format-patch release..master

#如果当前分支是 master，则等价于 git format-patch release..master
git format-patch^^..master
```

## 7.13 显示本地分支

```
git branch
```

## 7.14 创建本地分支

git branch <分支名称>

例：

```
git branch topic/a
```

## 7.15 删除本地分支

git branch -d <分支名称>

例：

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

## 7.16 强制删除本地分支

git branch -D <分支名称>

例：

```
git branch -D topic/a
```

## 7.17 显示远端分支

```
git branch -r
```

## 7.18 切换分支

git checkout <分支名>

例：

```
git checkout topic/a
```

## 7.19 显示远端仓库

```
git remote
```

## 7.20 推送代码

```
git push 推送代码
```

## 7.21 提取并更新代码

```
git pull
```

## 7.22 提取代码

```
git fetch
```

## 7.23 合并分支

git merge <被合并分支>

例：

```
#将 topic/a 合并到当前分支
git merge topic/a
```

## 7.24 调查 git 对象

git cat-file -t <SHA1>

git cat-file <TYPE> <SHA1>

例：

```
git cat-file -t abcd
git cat-file commit abcd
git cat-file blob 1234
```

## 7.25 调查 tree 对象

git ls-tree <SHA1>

## 7.26 内容检索

git grep <关键字>

git grep <关键字> <提交记录>

git grep <关键字> <提交记录>：<文件>

例：

```
git grep "version"
git grep "version" v1.1
git grep "version" v1.1:file.txt
```

## 7.27 移动文件/目录

git mv <原始位置/名称> <新位置/名称>

例：

```
git mv file.txt test001/test002/file.txt
git mv file.txt file1.txt
```

## 7.28 删除文件/目录

git rm <路径>

例：

```
git rm file.txt
git rm test001
```

## 7.29 显示所有标签

```
git tag -n
```

## 7.30 新建本地标签

git tag <标签名> <提交记录>

例：

```
#在当前分支的末端添加标签
git tag "michael-release-1.1.0"

#在 release 分支的末端添加标签
git tag "michael-release-1.1.0" release

git tag "bug" master^^
```

## 7.31 删除本地标签

git tag -d <标签名>

## 7.32 创建远端标签

git tag -a <标签名> -m <标签说明> <提交记录>

例：

```
#在当前分支的末端创建一个远端标签
git tag -a release-1.0 -m "release 1.0"

#在 release 分支的末端创建一个远端标签
git tag -a release-1.0 -m "release 1.0" release
```

## 7.33 推送远端标签

git push --tags

## 7.34 提取远端标签

git fetch --tags

## 7.35 撤销某条提交记录

git revert <提交记录>

例：

```
#所有 abcd 提交的修改将被撤销，git 实际上将创建一条新的提交，旧的提交不会被真的改变
#如果有冲突，需要手动解决冲突
git revert abcd
```

## 7.36 获得某个文件的历史版本

git checkout <提交记录> <文件>

例：

```
git checkout master ^^ file.txt
git checkout v1.1 file.txt
```

## 7.37 计算文件的 SHA1 字符串

git hash-object <路径>

例：

```
git hash-object file.txt
```

## 7.38 显示 git 配置

git config -l

例：

```
#显示全局配置
git config --global -l

#显示本地配置
git config -l
```

## 7.39 获得 git 帮助

git help <命令>

例：

```
git help log
```

# 8 附录

## 8.1 词汇表

| 词汇 | 解释 |
| --- | --- |
| | |
| | |

*附录 1: 词汇表*

## 8.2 引用

| 引用 | 注释 |
| --- | --- |
| | |
| | |

*附录 2: 引用*