

TutorialsPoint Git 教程

来源：易百教程

Git 教程

Git 诞生于一个极富纷争大举创新的年代。Linux 内核开源项目有着为数众多的参与者。绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991—2002 年间）。到 2002 年，整个项目组开始启用分布式版本控制系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了免费使用 BitKeeper 的权力。这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。他们对新的系统制订了若干目标：

- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许上千个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，它还有着令人难以置信的非线性分支管理系统（见第三章），可以应付各种复杂的项目开发需求。

Git 是一个分布式的版本控制和源代码管理系统，强调速度。[Git](#) 最初由 Linus Torvalds 设计和开发为 Linux 内核开发管理代码。Git 是 GNU 通用公共许可证版本 2 的条款下分发的免费软件。

本教程将教你如何使用 Git 在你的项目版本控制在分布式环境中的基于 Web 和非基于 Web 应用程序的开发工作。

读者

对于初学者来说已经准备本教程，帮助他们了解 Git 版本控制系统的基本功能。完成本教程后，可以把帮助你熟悉和使用 Git 版本控制系统。

前提条件

我们假设你要使用 Git 来处理各级 [Java](#) 和非 Java 项目。所以如果你有知识，开发的基于 Web 和非基于 Web 的应用程序的软件开发生命周期和知识，将有助于学习和使用 Git。

Git 基本概念 - Git 教程

版本控制系统 (VCS)

版本控制系统 (VCS) 是软件，帮助软件开发人员携手合作，他们的工作并保持完整的历史。

以下是 VCS 目标

1. 允许开发人员同步工作.
2. 不要覆盖对方的变化.
3. 维护历史的每一个版本.

以下是常见的 VCS

1. 集中式版本控制系统 (CVCS)
2. 分散式/分布式版本控制系统 (DVCS)

在这个教程，我们将介绍集中分布式的版本控制系统，尤其是 [Git](#)。Git 属于分布式版本控制系统。

分布式版本控制系统 (DVCS)

集中式版本控制系统采用中央服务器上存储的所有文件和实现团队协作。但是 CVCS 主要缺点是中央服务器的单点故障，即故障。不幸的是，如果中央服务器宕机一小时，然后在该时段没有人可以合作。即使在最坏的情况下，如果中央服务器的磁盘被损坏，并没有采取适当的备份，那么将失去整个项目的历史。

DVCS 客户不仅检出的最新快照目录，但他们也完全反映资源库。如果 SEVER 停机，然后从任何客户端库可以复制回服务器，以恢复它。每个结账是完整的版本库备份。Git 不会依赖中央服务器，这就是为什么可以执行许多操作，当处于脱机状态。可以提交修改，创建分支视图日志和执行其他操作，当处于脱机状态。只需要网络连接，发布您的更改，并采用最新变化。

Git 优势

Git 是 **GPL** 开源许可证下发布的。它可自由在互联网上。可以使用 **Git** 来管理项目无需支付一分钱。由于它是开源的，可以下载它的源代码，并根据要求进行更改。

由于大部分的操作都在本地执行，它给人带来巨大的好处，在速度方面。**Git** 不依赖于中央服务器，为什么每一个操作就没有必要与远程服务器进行交互。**Git** 核心部分是写在 **C** 中，从而避免了与其他高级语言的运行时开销。虽然 **Git** 中反映整个存储库，在客户端上的数据的大小是小的。这说明它是在客户端上的数据压缩和存储的效率有多高。

丢失数据的机会是非常罕见的，当有多个副本。存在于任何客户端的数据存储库中，因此它可以被用来在发生崩溃或磁盘损坏的镜像。

Git 使用常见的加密散列函数，称为安全的哈希算法（**SHA1**）命名，并确定在其数据库中的对象。每一个文件并提交检查总结和检索其校验在结账的时候。意思是说，这是不可能改变文件，日期，提交信息和且从 **Git** 数据库不知道 **Git** 任何其他数据。

在 **CVCS** 情况下，中心服务器需要足够强大，要求整个团队服务。对于较小的团队，这不是一个问题，但随着团队规模的增长，服务器的硬件限制可能成为一个性能瓶颈。在 **DVCS** 开发的情况下不与服务器进行交互，除非他们需要推或拉的变化。所有繁重发生在客户端上，所以服务器硬件可以是很简单的。

CVCS 使用廉价的复制机制，这意味着如果我们创建新的分支，它会复制到新分支的所有代码，所以它的耗时和效率不高。另外 **CVCS** 的分支的删除和合并是复杂和费时的。但是，使用 **Git** 分支管理是很简单的。只需要几秒钟，创建，删除和合并分支。

1. 自由和开放源码
2. 快速和小
3. 隐式备份
4. 安全
5. 不需要强大的硬件
6. 更简单的分支

DVCS 术语

本地资源库

每个 VCS 工具提供私有工作场所的工作副本。开发者在他的私人工作场所的变化，并提交这些更改后成为仓库的一部分。Git 的需要这一步，为他们提供的专用副本是整个仓库。用户可以执行许多操作，这个库中，如添加文件，删除文件，重命名文件，移动文件，提交改变，还有更多。

工作目录，暂存区域或索引

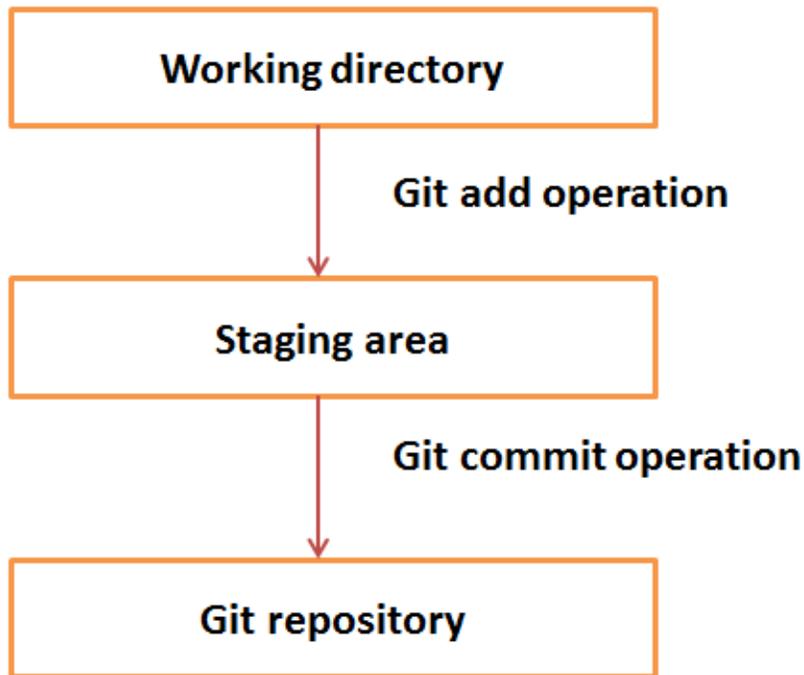
工作目录是地方文件检出。其他 CVCS 开发商一般不修改，并承诺他的变化，直接向版本库。但 Git 使用不同的策略。Git 不会跟踪每一个修改过的文件。每当提交操作，Git 在目前临时区域的文件。只有文件被认为是目前在临时区域提交，而不是所有修改过的文件。

让我们来看看 Git 的基本工作流程。

第 1 步：修改文件的工作目录。

第 2 步：将这些文件添加到暂存区

第 3 步：执行 commit 操作。这将文件从临时区域。推送操作后，它永久地存储更改的 Git 仓库



假设修改了两个文件，即“sort.c” and “search.c”，两种不同分别提交操作。可以添加一个文件分段区域，不提交。第一次提交后重复相同的步骤为另一个文件。

```
# First commit
[bash]$ git add sort.c

# adds file to the staging area
[bash]$ git commit -m "Added sort operation"

# Second commit
[bash]$ git add search.c

# adds file to the staging area
[bash]$ git commit -m "Added search operation"
```

BLOBS

BLOB 代表二进制大对象。为代表 blob 文件的每个版本。一个 blob 保存文件数据，但不包含任何有关文件的元数据。它是一个二进制文件，该文件它被命名为 **SHA1** 哈希 **Git** 数据库中。在 **Git** 中，文件未提及的名字。一切固定内容寻址。

Tree

树是一个对象，它表示一个目录。它拥有 blobs 以及其他子目录。一棵树是一个二进制文件，该文件存储 Blob 树，也被命名为树对象的 **SHA1** 哈希的引用。

Commit

提交持有的库的当前状态。**COMMIT** 命令同样由 **SHA1** 哈希的名字命名。可以考虑 commit 对象的链表节点。每个提交的对象有父 commit 对象的指针。从给定的承诺可以遍历寻找在父指针，查看历史记录提交。如果提交多个父承诺，这意味着特定的提交是由两个分支合并。

BRANCHES

分支用来创建另一条线的发展。默认情况下，**Git** 的主分支，这是一样躯干颠覆。平时要工作的新功能创建一个分支。功能完成后，它被合并回 **master** 分支，我们删除分支。每个分支所引用 **HEAD**，这点在分支的最新提交。每当做出了一个提交，**HEAD** 更新为最新提交。

TAGS

包括特定版本库中的标签分配一个有意义的名称。标签是非常相似的分支，但不同的是，标签是不可改变的。手段标记的一个分支，没有人打算修改。一旦标签被创建为特定的提交，即使创建一个新的提交，也不会被更新。通常开发人员创建标签的产品发布。

CLONE

克隆操作的库创建实例。克隆操作不仅检出的工作拷贝，但它也反映了完整的信息库。用户可以执行许多操作，这个本地仓库。网络介入是唯一的一次，当正在同步资料库实例。

PULL

Pull 操作复制的变化，本地的一个实例来从远程仓库。**Pull** 操作是用于两个存储库实例之间的同步。这是在 **Subversion** 更新操作一样。

PUSH

推动从本地存储库实例的远程操作副本的变化。这是用来储存到 **Git** 仓库中永久改变。这是在 **Subversion** 的提交操作相同。

HEAD

HEAD 指针总是指向分支的最新提交。每当你做出了一个提交，**HEAD** 更新为最新提交。**HEAD** 树枝存储在 `.git/refs/heads/` 目录中。

```
[CentOS]$ ls -l .git/refs/heads/  
master  
  
[CentOS]$ cat .git/refs/heads/master  
570837e7d58fa4bccd86cb575d884502188b0c49
```

REVISION

修订版本的源代码。在 **Git** 修订代表的提交。这些提交由 **SHA1** 安全哈希值确定。

URL

URL 代表的 **Git** 仓库的位置。 **Git** 的 **URL** 存储在配置文件中。

```
[tom@CentOS tom_repo]$ pwd  
/home/tom/tom_repo  
  
[tom@CentOS tom_repo]$ cat .git/config  
[core]  
repositoryformatversion = 0  
filemode = true  
bare = false  
logallrefupdates = true  
[remote "origin"]  
**url = gituser@git.server.com:project.git**  
fetch = +refs/heads/*:refs/remotes/origin/*
```

Git 环境设置（安装） - Git 教程

在使用 Git 之前，必须安装它，并做一些基本配置的变化。下面是步骤在 Ubuntu 和 CentOS [Linux](#) 安装 [Git](#) 客户端。

Git 客户端安装

如果使用的是 GNU/ Linux 发行版 Debian 基本 apt-get 命令就可以搞定一切。

```
[ubuntu ~]$ sudo apt-get install git-core
[sudo] password for ubuntu:

[ubuntu ~]$ git --version
git version 1.8.1.2
```

而且，如果使用的是基于 RPM 的 GNU/ Linux 发行版使用 yum 命令，如下：

```
[CentOS ~]$
su -
Password:

[CentOS ~]# yum -y install git-core

[CentOS ~]# git --version
git version 1.7.1
```

自定义 Git 环境

Git 提供 git 的配置工具，它允许设置配置变量。Git 会把所有的全局配置.gitconfig 文件位于主目录。要设置这些为全局配置值，添加 -global 选项，如果省略 -global 选项，那么配置是具体当前的 Git 存储库。

还可以设置系统范围内的配置。Git 存储这些值是在/etc/gitconfig 文件，其中包含的配置系统上的每一个用户和资源库。要设置这些值，必须有 root 权限，并使用 -system 选项。

上面的代码编译和执行时，它会产生以下结果：

设置用户名

此信息用于 Git 的每个提交。

```
[jerry@CentOS project]$ git config --global user.name "Jerry Mouse"
```

设置电子邮件 ID

此信息用于 Git 的每个提交。

```
[jerry@CentOS project]$ git config --global user.email  
"jerry@yiibai.com"
```

避免 **PULLING** 提交合并

先从远程资源库的最新变化，如果这些变化是不同的，默认情况下，Git 创建合并提交。我们可以通过以下设置来避免这种。

```
jerry@CentOS project]$ git config --global branch.autosetuprebase  
always
```

颜色高亮

下面的命令使颜色突出显示在控制台的 Git。

```
[jerry@CentOS project]$ git config --global color.ui true  
  
[jerry@CentOS project]$ git config --global color.status auto  
  
[jerry@CentOS project]$ git config --global color.branch auto
```

设置默认编辑器

默认情况下，Git 的使用系统默认取自 VISUAL 或 EDITOR 环境变量的编辑器。我们可以设定一个不同的使用 git 配置。

```
[jerry@CentOS project]$ git config --global core.editor vim
```

设置默认的合并工具

Git 不会提供一个默认的合并工具整合到工作树冲突的更改。我们可以设置默认的合并工具，通过启用以下设置。

```
[jerry@CentOS project]$ git config --global merge.tool vimdiff
```

列出 GIT 设置

为了验证自己的 Git 设置本地存储库使用 git 的 config-list 命令，如下所示。

```
[jerry@CentOS ~]$ git config --list
```

上面的命令会产生以下结果。

```
user.name=Jerry Mouse
user.email=jerry@yiibai.com
push.default=nothing
branch.autosetuprebase=always
color.ui=true
color.status=auto
color.branch=auto
core.editor=vim
merge.tool=vimdiff
```

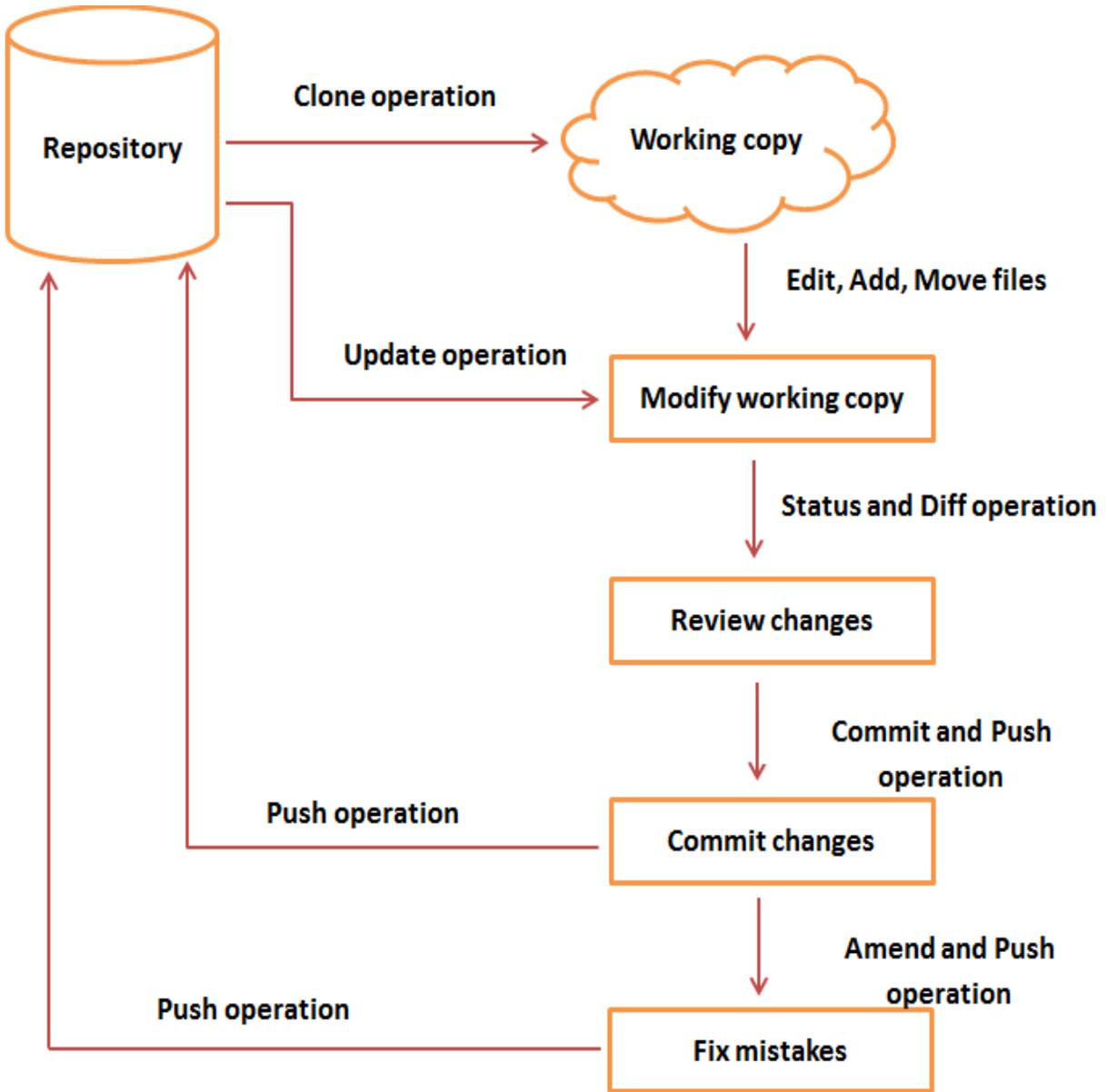
Git 生命周期 - Git 教程

在本章中，我们将讨论的 Git 的生命周期。在后面的章节中，我们将看到的 Git 命令为每个操作。

一般工作流程是这样的：

1. 克隆 Git 仓库作为工作副本。
2. 可以添加/编辑文件，修改工作副本。
3. 如果有必要，你还服用其他开发人员的变化，更新工作副本。
4. 审查前提交。
5. 提交修改。如果一切都很好，然后推到存储库的更改。
6. 提交之后，如果知道是什么错误，那么纠正最后一次提交，并推送修改到版本库。

以下是工作流程的图形表示。



Git 创建操作 - Git 教程

在本章中，我们将看到如何创建一个远程 Git 仓库，从现在开始，我们将会把它作为 Git 服务器。我们需要一个的 Git 服务器允许团队协作。

创建新用户

```
# add new group
[root@CentOS ~]# groupadd dev

# add new user
[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash
gituser

# change password
[root@CentOS ~]# passwd gituser
```

上面的命令会产生以下结果。

```
Changing password for user gituser.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

创建一个裸库

让我们初始化一个新的资料库使用 `init` 命令后面加上 `-bare` 选项。它初始化没有工作目录库。按照惯例裸库必须命名为 `.git`。

```
[gituser@CentOS ~]$ pwd
/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls
```

```
[gituser@CentOS project.git]$ git --bare init
Initialized empty Git repository in /home/gituser-m/project.git/
```

```
[gituser@CentOS project.git]$ ls
branches config description HEAD hooks info objects refs
```

生成公共/私有 RSA 密钥对

让我们遍历 Git 服务器端的配置过程中，使用 `ssh-keygen` 实用程序生成公共/私有 RSA 密钥对，我们将使用这些键进行用户认证。

打开一个终端并输入以下命令，直接按回车为每个输入。成功完成后，它会创建主目录 `.ssh` 目录内。

```
tom@CentOS ~]$ pwd
/home/tom
```

```
[tom@CentOS ~]$ ssh-keygen
```

上面的命令会产生以下结果。

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/tom/.ssh/id_rsa): **Press
Enter Only**
Created directory '/home/tom/.ssh'.
Enter passphrase (empty for no passphrase): **-----> Press
Enter Only**
Enter same passphrase again: **-----> Press
Enter Only**
Your identification has been saved in /home/tom/.ssh/id_rsa.
Your public key has been saved in /home/tom/.ssh/id_rsa.pub.
The key fingerprint is:
df:93:8c:a1:b8:b7:67:69:3a:1f:65:e8:0e:e9:25:a1 tom@CentOS
The key's randomart image is:
+--[ RSA 2048]-----+
| |
```

```
| |
| |
|
.
|
| Soo |
| o*B. |
| E = *. = |
| oo==. . |
| ..+0o
|
+-----+
```

ssh-keygen 已经产生了两个键，第一个是私有的（即 id_rsa），另一个是公共（即 id_rsa.pub 文件）。

> 注: 切勿与他人共享你的私钥。

添加键 `authorized_keys`

假设有两个开发项目即 Tom 和 Jerry 工作。两个用户生成公钥。让我们来看看如何使用这些密钥进行身份验证。

Tom 添加他的公钥服务器使用 `ssh-copy-id` 这个命令下面给出

```
[tom@CentOS ~]$ pwd
/home/tom

[tom@CentOS ~]$ ssh-copy-id -i ~/.ssh/id_rsa.pub
gituser@git.server.com
```

上面的命令会产生以下结果。

```
gituser@git.server.com's password:
Now try logging into the machine, with "ssh 'gituser@git.server.com'",
and check in:
.ssh/authorized_keys
```

```
to make sure we haven't added extra keys that you weren't expecting.
```

同样，Jerry 也增加了他的公共密钥服务器使用 `ssh-copy-id` 这个命令。

```
[jerry@CentOS ~]$ pwd
/home/jerry

[jerry@CentOS ~]$ ssh-copy-id -i ~/.ssh/id_rsa gituser@git.server.com
```

上面的命令会产生以下结果。

```
gituser@git.server.com's password:
Now try logging into the machine, with "ssh 'gituser@git.server.com'",
and check in:
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't expecting.
```

推修改到版本库

我们已经创建了裸库在服务器上，并允许两个用户访问。从现在 Tom 和 Jerry 可以把他们修改到版本库，将其添加为远程。

Git 的 `init` 命令创建 `.git` 目录来存储元数据的存储库。每次读取配置从 `.git/config` 文件。

Tom 创建一个新的目录，添加 `README` 文件作为初始提交并提交他的变化。提交后，他确认提交信息，运行 `git` 日志命令。

```
[tom@CentOS ~]$ pwd
/home/tom

[tom@CentOS ~]$ mkdir tom_repo

[tom@CentOS ~]$ cd tom_repo/

[tom@CentOS tom_repo]$ git init
Initialized empty Git repository in /home/tom/tom_repo/.git/
```

```
[tom@CentOS tom_repo]$ echo 'TODO: Add contents for README' > README

[tom@CentOS tom_repo]$ git status -s
?? README

[tom@CentOS tom_repo]$ git add .

[tom@CentOS tom_repo]$ git status -s
A README

[tom@CentOS tom_repo]$ git commit -m 'Initial commit'
```

上面的命令会产生以下结果。

```
[master (root-commit) 19ae206] Initial commit
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README
```

Tom 执行 git 的日志命令，检查日志消息。

```
[tom@CentOS tom_repo]$ git log
```

上面的命令会产生以下结果。

```
commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@yiibai.com>Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit</tom@yiibai.com>
```

Tom 提交了他的变化到本地资源库。现在是时候将更改到远程仓库。但在此之前，我们必须添加作为远程仓库，这是一个时间的操作。在此之后，他可以放心地推送到远程存储库的更改。

> **注:** 默认情况下, **Git** 的推到匹配的分支: 对于每一个分支退出本地端的远程端更新, 如果已经存在具有相同名称的一个分支。在我们的教程每次我推原点主分支的变化, 根据您的要求, 使用适当的分支名。

```
[tom@CentOS tom_repo]$ git remote add origin
gituser@git.server.com:project.git

[tom@CentOS tom_repo]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 242 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
* [new branch]
master -> master
```

现在更改成功提交到远程仓库。

Git 克隆操作 - Git 教程

我们有一个裸库 Git 服务器，Tom 也推了他的第一个版本。现在，Jerry 可以查看他的变化。克隆操作的远程存储库创建实例。

Jerry 在他的 home 目录，并创建新的目录，执行克隆操作。

```
[jerry@CentOS ~]$ mkdir jerry_repo  
  
[jerry@CentOS ~]$ cd jerry_repo/  
  
[jerry@CentOS jerry_repo]$ git clone  
gituser@git.server.com:project.git
```

上面的命令会产生以下结果。

```
Initialized empty Git repository in  
/home/jerry/jerry_repo/project/.git/  
remote: Counting objects: 3, done.  
Receiving objects: 100% (3/3), 241 bytes, done.  
remote: Total 3 (delta 0), reused 0 (delta 0)
```

Jerry 改变目录到新的本地存储库，并列出目录内容。

```
[jerry@CentOS jerry_repo]$ cd project/  
  
[jerry@CentOS jerry_repo]$ ls  
README
```

Git 执行更改 - Git 教程

Jerry 克隆库，他决定实现基本字符串操作。于是，他创建文件 `string.c`，在添加内容到 `string.c` 会这个样子。

```
#include <stdio.h>

int my_strlen(char *s)
{
    char *p = s;

    while (*p)
        ++p;

    return (p - s);
}

int main(void)
{
    int i;
    char *s[] = {
        "Git tutorials",
        "Tutorials Yiibai"
    };

    for (i = 0; i < 2; ++i)
        printf("string lenght of %s = %d\n", s[i], my_strlen(s[i]));

    return 0;
}
```

他编译和测试代码，一切工作正常。现在，他可以放心地添加这些修改到版本库。

Git 添加操作添加文件到暂存区。

```
[jerry@CentOS project]$ git status -s
?? string
?? string.c

[jerry@CentOS project]$ git add string.c
```

Git 是显示文件名前的问号。显然，这些文件不属于 Git，Git 不知道该怎么用这些文件。这就是为什么 Git 是文件名前显示问号。

Jerry 添加文件到存储区域，git 的状态命令将显示文件暂存区域。

```
[jerry@CentOS project]$ git status -s
A string.c
?? string
```

要提交更改他用 git 的 commit 命令 -m 选项。如果我们省略 -m 选项 git 会打开文本编辑器，在这里我们可以写多行提交信息。

```
[jerry@CentOS project]$ git commit -m 'Implemented my_strlen function'
```

上面的命令会产生以下结果。

```
[master cbe1249] Implemented my_strlen function
1 files changed, 24 insertions(+), 0 deletions(-)
create mode 100644 string.c
```

提交后查看日志信息，他使用 git 日志命令。它会显示提交 ID 所有提交的信息，提交作者，提交日期和提交的 SHA-1 散列。

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
```

```
Author: Jerry Mouse <jerry@yiibai.com>
```

```
Date: Wed Sep 11 08:05:26 2013 +0530
```

```
Implemented my_strlen function
```

```
commit 19ae20683fc460db7d127cf201a1429523b0e319
```

```
Author: Tom Cat <tom@yiibai.com>
```

```
Date: Wed Sep 11 07:32:56 2013 +0530
```

```
Initial commit
```

Git 审查更改 - Git 教程

但查看提交详细资料后，Jerry 实现字符串的长度不能为负数，所以他决定改变 `my_strlen` 函数的返回类型。

Jerry 使用 `git` 日志命令来查看日志信息。

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function
```

Jerry 使用 `git show` 命令查看提交的细节。Git 的 `show` 命令的 SHA-1 提交 ID 作为参数。

```
[jerry@CentOS project]$ git show
cbe1249b140dad24b2c35b15cc7e26a6f02d2277
```

上面的命令会产生以下结果。

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function

diff --git a/string.c b/string.c
new file mode 100644
index 0000000..187afb9
--- /dev/null
+++ b/string.c
```

```

@@ -0,0 +1,24 @@
+#include <stdio.h>
+
+int my_strlen(char *s)
+{
+
+char *p = s;
+
+
+while (*p)
+ ++p;
+ return (p - s );
+}
+

```

他改变了函数的返回类型 从 `int` 修改为 `size_t`。测试代码后，他查看其变化运行 `git diff` 命令。

```
[jerry@CentOS project]$ git diff
```

上面的命令会产生以下结果。

```

diff --git a/string.c b/string.c
index 187afb9..7da2992 100644
--- a/string.c
+++ b/string.c
@@ -1,6 +1,6 @@
#include <stdio.h>

-int my_strlen(char *s)
+size_t my_strlen(char *s)
{
char *p = s;
@@ -18,7 +18,7 @@ int main(void)
};
for (i = 0; i < 2; ++i)
- printf("string lenght of %s = %d

```

```
" , s[i], my_strlen(s[i]));  
+ printf("string lenght of %s = %lu  
" , s[i], my_strlen(s[i]));  
return 0;  
}
```

Git 的差异显示+号前行，这是新增加的，并显示符号被删除。

Git 提交更改 - Git 教程

Jerry 已经提交的更改，他想纠正他的最后一次提交，在这种情况下，git 的修改将帮助操作。最后提交修改操作的变化，包括提交信息，它创建新的提交 ID。

修改操作之前，他会检查提交日志。

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function

commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit
```

Jerry 提交了新的变化 - 修改操作，并查看提交日志。

```
[jerry@CentOS project]$ git status -s
M string.c
?? string

[jerry@CentOS project]$ git add string.c

[jerry@CentOS project]$ git status -s
M string.c
?? string

[jerry@CentOS project]$ git commit --amend -m 'Changed return type of
my_strlen to size_t'
```

```
[master d1e19d3] Changed return type of my_strlen to size_t
1 files changed, 24 insertions(+), 0 deletions(-)
create mode 100644 string.c
```

现在 git 的日志，将显示新的提交信息与新的提交 ID

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my_strlen to size_t

commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit
```

Git 推送操作 - Git 教程

Jerry 修改了他的最后一次提交的修改操作，他已经准备好将更改。推操作的数据永久存储的 Git 仓库。推操作成功后，其他开发人员可以看到 Jerry 的变化。

他执行的 git 日志命令来查看提交的细节。

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my_strlen to size_t
```

push 操作之前，他要审查他的变化，所以使用 git show 命令来查看他的变化。

```
[jerry@CentOS project]$ git show
d1e19d316224cddc437e3ed34ec3c931ad803958
```

上面的命令会产生以下结果。

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my_strlen to size_t

diff --git a/string.c b/string.c
new file mode 100644
index 0000000..7da2992
--- /dev/null
+++ b/string.c
@@ -0,0 +1,24 @@
```

```
+#include <stdio.h>
+
+size_t my_strlen(char *s)
+{
+
+char *p = s;
+
+
+while (*p)
+ ++p;
+ return (p -s );
+}
+
+int main(void)
+{
+ int i;
+ char *s[] = {
+ "Git tutorials",
+ "Tutorials Yiibai"
+
+ };
+
+
+
+ for (i = 0; i < 2; ++i)
+ printf("string lenght of %s = %lu
+ ", s[i], my_strlen(s[i]));
+
+
+ return 0;
+}
```

Jerry 为他的变化感到高兴，他是准备推他的变化。

```
[jerry@CentOS project]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 4, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 517 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
19ae206..d1e19d3 master -> master
```

Jerry 的变化成功地推到版本库，现在其他开发人员可以查看他的变化进行克隆或更新操作。

Git 更新操作 - Git 教程

修改现有函数

Tom 执行克隆操作后，看到新的文件 `string.c`，他想知道这个文件到存储库？目的是什么？于是，他执行 `git` 日志命令。

```
[tom@CentOS ~]$ git clone gituser@git.server.com:project.git
```

上面的命令会产生以下结果。

```
Initialized empty Git repository in /home/tom/project/.git/  
remote: Counting objects: 6, done.  
remote: Compressing objects: 100% (4/4), done.  
Receiving objects: 100% (6/6), 726 bytes, done.  
remote: Total 6 (delta 0), reused 0 (delta 0)
```

克隆操作将当前的工作目录内创建新的目录。他改变目录到新创建的目录和执行 `git` 日志命令。

```
[tom@CentOS ~]$ cd project/  
  
[tom@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958  
Author: Jerry Mouse <jerry@yiibai.com>  
Date: Wed Sep 11 08:05:26 2013 +0530  
  
Changed return type of my_strlen to size_t  
  
commit 19ae20683fc460db7d127cf201a1429523b0e319  
Author: Tom Cat <tom@yiibai.com>  
Date: Wed Sep 11 07:32:56 2013 +0530
```

```
Initial commit
```

查看记录后，他意识到，`string.c` 文件加入 Jerry 实现基本字符串操作。他是好奇 Jerry 的代码。于是他打开文本编辑 `string.c` 文件，并立即找到了一个漏洞。`my_strlen` 函数中 Jerry 没有使用常量指针。于是，他决定修改 Jerry 的代码。修改后的代码会这个样子。

```
[tom@CentOS project]$ git diff
```

上面的命令会产生以下结果。

```
diff --git a/string.c b/string.c
index 7da2992..32489eb 100644
--- a/string.c
+++ b/string.c
@@ -1,8 +1,8 @@
#include <stdio.h>-size_t my_strlen(char *s)
+size_t my_strlen(const char *s)
{
- char *p = s;
+ const char *p = s;
while (*p)
++p;</stdio.h>
```

经过测试，他提交他的变化。

```
[tom@CentOS project]$ git status -s
M string.c
?? string

[tom@CentOS project]$ git add string.c

[tom@CentOS project]$ git commit -m 'Changed char yiibaier to const char yiibaier'
[master cea2c00] Changed char yiibaier to const char yiibaier
1 files changed, 2 insertions(+), 2 deletions(-)
```

```
[tom@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit cea2c000f53ba99508c5959e3e12fff493b
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 08:32:07 2013 +0530

Changed char yiibaier to const char yiibaier

commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my_strlen to size_t

commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 07:32:56 2013 +0530
Initial commit
```

Tom 使用 `git push` 命令，推他的变化。

```
[tom@CentOS project]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 336 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
d1e19d3..cea2c00 master -> master
```

添加新函数

与此同时，Jerry 决定实现字符串比较函数。于是，他修改 `string.c` 文件。修改后的文件会这个样子。

```
[jerry@CentOS project]$ git diff
```

上面的命令会产生以下结果。

```
index 7da2992..bc864ed 100644
--- a/string.c
+++ b/string.c
30Git Tutorials
@@ -9,9 +9,20 @@ size_t my_strlen(char *s)
return (p - s );
}
+char *my_strcpy(char *t, char *s)
+{
+
+char *p = t;
+
+ while (*t++ = *s++)
+ ;
+
+return p;
+}
+
int main(void)
{
int i;
+
char p1[32];
char *s[] = {
"Git tutorials",
"Tutorials Yiibai"
@@ -20,5 +31,7 @@ int main(void)
for (i = 0; i < 2; ++i)
```

```
printf("string length of %s = %lu\n", s[i], my_strlen(s[i]));  
+  
printf("%s\n", my_strcpy(p1, "Hello, World !!!"));  
+  
return 0;  
}
```

经过测试，他准备推他的变化。

```
[jerry@CentOS project]$ git status -s  
M string.c  
?? string  
  
[jerry@CentOS project]$ git add string.c  
  
[jerry@CentOS project]$ git commit -m "Added my_strcpy function"  
[master e944e5a] Added my_strcpy function  
1 files changed, 13 insertions(+), 0 deletions(-)
```

push 操作之前，他验证通过查看日志信息提交。

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit e944e5aab74b26e7447d3281b225309e4e59efcd  
Author: Jerry Mouse <jerry@yiibai.com>  
Date: Wed Sep 11 08:41:42 2013 +0530  
  
Added my_strcpy function  
  
commit d1e19d316224cddc437e3ed34ec3c931ad803958  
Author: Jerry Mouse <jerry@yiibai.com>  
Date: Wed Sep 11 08:05:26 2013 +0530
```

```
Changed return type of my_strlen to size_t
```

```
commit 19ae20683fc460db7d127cf201a1429523b0e319
```

```
Author: Tom Cat <tom@yiibai.com>
```

```
Date: Wed Sep 11 07:32:56 2013 +0530
```

```
Initial commit
```

Jerry 为变化感到高兴，他想推他的变化

```
[jerry@CentOS project]$ git push origin master
```

上面的命令会产生以下结果。

```
To gituser@git.server.com:project.git
```

```
! [rejected]
```

```
master -> master (non-fast-forward)
```

```
error: failed to push some refs to
```

```
'gituser@git.server.com:project.git'
```

```
To prevent you from losing history, non-fast-forward updates were  
rejected
```

```
Merge the remote changes before pushing again. See the 'Note about  
fast-forwards' section of 'git push --help' for details.
```

但是，Git 是不允许 Jerry 推他的变化。因为 Git 确定该远程仓库和 Jerry 的本地资源库不同步。正因为如此，他可能会失去项目的历史。因此，为了避免这种混乱的 Git 对此操作失败。Jerry 必须首先更新它的本地存储库，然后再只有他才能把他自己的变化。

取最新变化

Jerry 执行 git pull 命令远程命令来同步自己的本地仓库。

```
[jerry@CentOS project]$ git pull
```

上面的命令会产生以下结果。

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From git.server.com:project
d1e19d3..cea2c00 master -> origin/master
First, rewinding head to replay your work on top of it...
Applying: Added my_strcpy function
```

抽取操作后 Jerry 检查日志消息，并找到 Tom 的提交详细，提交 ID 为 cea2c000f53ba99508c5959e3e12fff493ba6f69

```
[jerry@CentOS project]$ git log
```

上面的命令会产生以下结果。

```
commit e86f0621c2a3f68190bba633a9fe6c57c94f8e4f
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:41:42 2013 +0530

Added my_strcpy function

commit cea2c000f53ba99508c5959e3e12fff493ba6f69
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 08:32:07 2013 +0530

Changed char yiibaier to const char yiibaier

commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my_strlen to size_t

commit 19ae20683fc460db7d127cf201a1429523b0e319
```

```
Author: Tom Cat <tom@yiibai.com>  
Date: Wed Sep 11 07:32:56 2013 +0530  
Initial commit
```

现在，Jerry 的本地存储库是完全同步的远程仓库。所以，他可以放心地将他的变化。

```
[jerry@CentOS project]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 5, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 455 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
cea2c00..e86f062 master -> master
```

Git 藏匿操作 - Git 教程

假设您正在为您的产品实施的一项新功能。你的代码是在推进开发进度而客户不断升级需求突然来了。正因为如此，你必须保持放下你的新功能，工作几个小时。你不能提交你的部分代码，也不能扔掉你的变化。所以，你需要一些临时空间，在那里你可以存储你的部分修改，以便以后再提交。

在 Git 中，藏匿操作需要修改的跟踪文件和阶段的变化，并将其保存在栈上未完成的更改，可以在任何时候重新。

```
[jerry@CentOS project]$ git status -s
M string.c
?? string
```

现在要切换分支机构为客户不断升级，但你不想要提交你的工作，所以你会藏匿的变化。要推一个新的藏匿到您的堆栈，运行 `git stash` 命令

```
[jerry@CentOS project]$ git stash
Saved working directory and index state WIP on master: e86f062 Added
my_strcpy function
HEAD is now at e86f062 Added my_strcpy function
```

现在你的工作目录是干净的，所有的改变都保存在堆栈。让我们用 `git status` 命令验证。

```
[jerry@CentOS project]$ git status -s
?? string
```

现在可以安全地切换分支和做其他工作。我们可以看到的藏匿的变化列表通过使用 `git stash list` 命令。

```
[jerry@CentOS project]$ git stash list
stash@{0}: WIP on master: e86f062 Added my_strcpy function
```

假设你解决了客户不断升级和你要回到你的工作，已经做了一半的代码。只要执行 `git stash pop` 命令，它会从堆栈中删除的变化，并把它放在当前工作目录。

```
[jerry@CentOS project]$ git status -s
?? string

[jerry@CentOS project]$ git stash pop
```

上面的命令会产生以下结果。

```
# On branch master
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#
modified: string.c
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
#
string
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (36f79dfedae4ac20e2e8558830154bd6315e72d4)

[jerry@CentOS project]$ git status -s
M string.c
?? string</file></file></file>
```

Git 移动操作 - Git 教程

顾名思义移动(move)操作移动目录或文件从一个位置到另一个。Tom 决定移动到 src 目录下的源代码。因此，修改后的目录结构看起来会像这样。

```
[tom@CentOS project]$ pwd
/home/tom/project

[tom@CentOS project]$ ls
README string string.c

[tom@CentOS project]$ mkdir src

[tom@CentOS project]$ git mv string.c src/

[tom@CentOS project]$ git status -s
R string.c -> src/string.c
?? string
```

要进行这些永久性更改，以便其他开发人员可以看到这一点，我们必须修改的目录结构推到远程存储库。

```
[tom@CentOS project]$ git commit -m "Modified directory structure"

[master 7d9ea97] Modified directory structure
1 files changed, 0 insertions(+), 0 deletions(-)
rename string.c => src/string.c (100%)

[tom@CentOS project]$ git push origin master
Counting objects: 4, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
e86f062..7d9ea97 master -> master
```

在 Jerry 的本地资源库，抽取操作前，它会显示旧的目录结构。

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project
```

```
[jerry@CentOS project]$ ls
README string string.c
```

但是，抽取(pull)操作后的目录结构将得到更新。现在，Jerry 可以看到该目录内的 src 目录和文件。

```
[jerry@CentOS project]$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From git.server.com:project
e86f062..7d9ea97 master -> origin/master
First, rewinding head to replay your work on top of it...
Fast-forwarded master to 7d9ea97683da90bcdb87c28ec9b4f64160673c8a.
```

```
[jerry@CentOS project]$ ls
README src string
```

```
[jerry@CentOS project]$ ls src/
string.c
```

Git 重命名操作 - Git 教程

截至目前，Tome 和 Jerry 都使用手动命令来编译自己的项目。Jerry 决定为他们的项目创建 [Makefile](#)，并给予适当的名称来命名“string.c”文件。

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project

[jerry@CentOS project]$ ls
README src

[jerry@CentOS project]$ cd src/

[jerry@CentOS src]$ git add Makefile

[jerry@CentOS src]$ git mv string.c string_operations.c

[jerry@CentOS src]$ git status -s
A Makefile
R string.c -> string_operations.c
```

Git 是显示 R 在文件之前名称来指示文件已更名。

对于提交操作 Jerry 使用 `-a` 标志，这使得 git 提交自动检测修改过的文件。

```
[jerry@CentOS src]$ git commit -a -m 'Added Makefile and renamed
strings.c to
string_operations.c '

[master 94f7b26] Added Makefile and renamed strings.c to
string_operations.c
1 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/Makefile
rename src/{string.c => string_operations.c} (100%)
```

提交后，他推送了他的修改到版本库。

```
[jerry@CentOS src]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 6, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (4/4), 396 bytes, done.  
Total 4 (delta 0), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
7d9ea97..94f7b26 master -> master
```

现在，其他开发人员可以通过更新他们的本地资源库中的这些修改。

Git 删除操作 - Git 教程

Tom 更新了自己的本地存储库并进入 `src` 目录下找到编译后的二进制。查看提交信息后，他意识到，编译后的二进制是由 Jerry 加入的。

```
[tom@CentOS src]$ pwd
/home/tom/project/src

[tom@CentOS src]$ ls
Makefile string_operations string_operations.c

[tom@CentOS src]$ file string_operations
string_operations: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.18, not stripped

[tom@CentOS src]$ git log
commit 29af9d45947dc044e33d69b9141d8d2dad37cc62
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 10:16:25 2013 +0530

Added compiled binary
```

VCS 用于存储源代码，而不是只对可执行的二进制文件。因此，Tom 决定从资源库中删除此文件。对于进一步的操作，他使用 `git` 的 `rm` 命令。

```
[tom@CentOS src]$ ls
Makefile string_operations string_operations.c

[tom@CentOS src]$ git rm string_operations
rm 'src/string_operations'

[tom@CentOS src]$ git commit -a -m "Removed executable binary"

[master 5776472] Removed executable binary
1 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100755 src/string_operations
```

提交后，他推送了他的修改到版本库。

```
[tom@CentOS src]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 5, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 310 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
29af9d4..5776472 master -> master
```

Git 修正错误 - Git 教程

大部分的人都会犯错。所以每 VCS 提供了一个功能，修正错误，直到特定的点。Git 提供功能使用，我们可以撤销已作出的修改到本地资源库。

假设用户不小心做了一些更改，以他的本地的仓库，现在他要扔掉这些变化。在这种情况下，恢复操作中起着重要的作用。

恢复未提交的更改

让我们假设 Jerry 不小心修改文件从自己的本地仓库。但他想扔掉他的修改。要处理这种情况，我们可以使用 `git checkout` 命令。我们可以使用这个命令来恢复文件的内容。

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git checkout string_operations.c

[jerry@CentOS src]$ git status -s
```

甚至我们可以使用 `git checkout` 命令删除的文件从本地库。让我们假设 Tom 删除文件从本地存储库，我们希望这个文件。我们可以做到这一点，使用相同的命令。

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src

[tom@CentOS src]$ ls -l
Makefile
string_operations.c

[tom@CentOS src]$ rm string_operations.c

[tom@CentOS src]$ ls -l
```

```
Makefile
```

```
[tom@CentOS src]$ git status -s  
D string_operations.c
```

Git 是显示文件名前的字母 D。这标志着该文件已被删除，从本地资源库。

```
[tom@CentOS src]$ git checkout string_operations.c
```

```
[tom@CentOS src]$ ls -l  
Makefile  
string_operations.c
```

```
[tom@CentOS src]$ git status -s
```

> 注意：我们可以执行所有这些操作之前提交操作。

删除临时区域变化

我们已经看到，当我们执行加法运算;文件移动从本地存储库，参数区域。如果用户不小心修改一个文件，并把它添加到临时区域，但他马上意识到犯了错误。他希望恢复他的改变。我们可以处理这种情况下，通过使用 **git checkout** 命令。

在 Git 是一个 HEAD 指针始终指向最新提交。如果想撤销变更分阶段区域，那么可以使用 **git checkout** 命令，但 **checkout** 命令，必须提供额外的参数 HEAD 指针。额外提交指针参数指示的 **git checkout** 命令，重置工作树，还能够去除分阶段。

让我们假设 Tom 从自己的本地仓库修改一个文件。如果我们查看这个文件的状态，它会显示文件被修改，但不加入临时区域。

```
tom@CentOS src]$ pwd  
/home/tom/top_repo/project/src  
# Unmodified file  
  
[tom@CentOS src]$ git status -s
```

```
# Modify file and view it's status.  
[tom@CentOS src]$ git status -s  
M string_operations.c  
  
[tom@CentOS src]$ git add string_operations.c
```

Git 的状态显示该文件是在分期区域，现在它恢复使用 `git checkout` 命令和视图状态恢复的文件。

```
[tom@CentOS src]$ git checkout HEAD -- string_operations.c  
  
[tom@CentOS src]$ git status -s
```

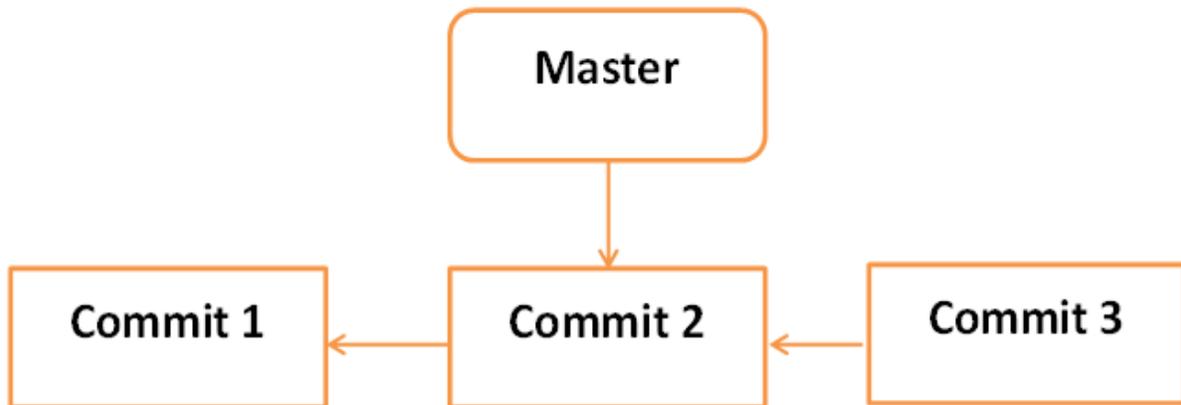
将 HEAD 指针与 GIT 复位

做一些改变后，可能会决定删除这些更改。Git 复位命令是用来重置或恢复一些变化。我们可以执行三种不同类型的复位操作。

下图显示图形表示 Git 复位命令。



Before git reset command



After git reset command

SOFT

每个分支都有 HEAD 指针指向最新提交。如果我们使用 `git --soft` 复位命令选项，随后提交 ID，然后将只有头指针复位，不破坏任何东西。

`.git/refs/heads/master` 文件存储的提交 ID 的 HEAD 指针。我们可以验证它通过使用 `git log -1` 命令。

```
[jerry@CentOS project]$ cat .git/refs/heads/master  
577647211ed44fe2ae479427a0668a4f12ed71a1
```

现在查看最新提交的 ID，这将配合上述提交 ID。

```
[jerry@CentOS project]$ git log -2
```

上面的命令会产生以下结果。

```
commit 577647211ed44fe2ae479427a0668a4f12ed71a1  
Author: Tom Cat <tom@yiibai.com>  
Date: Wed Sep 11 10:21:20 2013 +0530  
  
Removed executable binary  
  
commit 29af9d45947dc044e33d69b9141d8d2dad37cc62  
Author: Jerry Mouse <jerry@yiibai.com>  
Date: Wed Sep 11 10:16:25 2013 +0530  
  
Added compiled binary
```

让我们重设 HEAD 指针。

```
[jerry@CentOS project]$ git reset --soft HEAD~
```

现在我们只重设 HEAD 指针回到一个位置。让我们检查内容 `.git/refs/heads/master` 文件。

```
[jerry@CentOS project]$ cat .git/refs/heads/master
29af9d45947dc044e33d69b9141d8d2dad37cc62
```

从文件提交 ID 改变，现在验证通过查看提交的信息。

```
jerry@CentOS project]$ git log -2
```

上面的命令会产生以下结果。

```
commit 29af9d45947dc044e33d69b9141d8d2dad37cc62
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 10:16:25 2013 +0530

Added compiled binary

commit 94f7b26005f856f1a1b733ad438e97a0cd509c1a
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 10:08:01 2013 +0530

Added Makefile and renamed strings.c to string_operations.c
```

混合

Git 的复位 `--mixed` 选项从分段区域尚未提交还原更改。它仅恢复变化形成暂存区。实际所做的更改到工作副本的文件不受影响。默认的 Git 的复位相当于 `git` 的复位 `--mixed`。

有关更多详细信息，请参阅部分删除同一章临时区域变化。

HARD

如果使用 `-hard` 选项用 Git 复位命令，它会清除暂存区域，它会重设 HEAD 指针，以最后一次提交的具体提交 ID，也删除本地文件的变化。

让我们检查提交的 ID

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git log -1
```

上面的命令会产生以下结果。

```
commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary
```

Jerry 修改过的文件，在文件的开始，加入单行注释。

```
[jerry@CentOS src]$ head -2 string_operations.c
/* This line be removed by git reset operation */
#include <stdio.h>
```

他使用 `git status` 命令验证。

```
[jerry@CentOS src]$ git status -s
M string_operations.c
```

Jerry 修改后的文件添加到分期区域，并验证它与 `git` 的状态运行。

```
[jerry@CentOS src]$ git add string_operations.c
[jerry@CentOS src]$ git status
```

上面的命令会产生以下结果。

```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
```

```
#  
#  
modified: string_operations.c  
#
```

Git 的状态，显示该文件是在临时区域。现在重置 HEAD 用 `--hard` 选项。

```
[jerry@CentOS src]$ git reset --hard  
577647211ed44fe2ae479427a0668a4f12ed71a1  
  
HEAD is now at 5776472 Removed executable binary
```

Git 复位命令成功，这将恢复从分段区的文件，以及删除本地对文件所做的更改。

```
[jerry@CentOS src]$ git status -s
```

Git 状态显示，文件恢复从分段区。

```
[jerry@CentOS src]$ head -2 string_operations.c  
#include <stdio.h>
```

Head 命令还显示，复位操作删除局部变化。

Git 标签操作 - Git 教程

允许有意义的名称到一个特定的版本库中的标签操作。Tom 决定标记他们的项目代码，以便他们以后可以更容易访问。

创建标签

让我们标记当前 HEAD 使用 `git tag` 命令。他提供的标记名称前加上 `-a` 选项，使用 `-m` 选项，并提供标签信息。

```
tom@CentOS project]$ pwd
/home/tom/top_repo/project

[tom@CentOS project]$ git tag -a 'Release_1_0' -m 'Tagged basic string
operation code' HEAD
```

如果想标记特定提交然后使用适当的 COMMIT ID，而不是 HEAD 指针。Tom 使用下面的命令推到远程存储库中的标签。

```
[tom@CentOS project]$ git push origin tag Release_1_0
```

上面的命令会产生以下结果。

```
Counting objects: 1, done.
Writing objects: 100% (1/1), 183 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
* [new tag]
Release_1_0 -> Release_1_0
```

查看标签

Tom 创建标签。现在，Jerry 可以查看所有可用标签通过使用 `Git tag` 命令使用 `-l` 选项。

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git pull
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From git.server.com:project
* [new tag]
Release_1_0 -> Release_1_0
Current branch master is up to date.

[jerry@CentOS src]$ git tag -l
Release_1_0
```

Jerry 使用 Git 的 show 命令后跟标记名称的有关标签查看更多细节。

```
[jerry@CentOS src]$ git show Release_1_0
```

上面的命令会产生以下结果。

```
tag Release_1_0
Tagger: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 13:45:54 2013 +0530

Tagged basic string operation code

commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary

diff --git a/src/string_operations b/src/string_operations
deleted file mode 100755
index 654004b..0000000
```

```
Binary files a/src/string_operations and /dev/null differ
```

删除标签

Tom 使用下面的命令来删除标记从本地以及远程仓库。

```
[tom@CentOS project]$ git tag
Release_1_0

[tom@CentOS project]$ git tag -d Release_1_0
Deleted tag 'Release_1_0' (was 0f81ff4)
# Remove tag from remote repository.

[tom@CentOS project]$ git push origin :Release_1_0
To gituser@git.server.com:project.git
- [deleted]
Release_1_0
```

Git 补丁操作 - Git 教程

补丁是文本文件，其内容是相似于 **Git diff**，但随着代码，它也有元数据有关提交，如提交 ID，日期，提交信息等，我们可以创建补丁提交和其他人可以将它们应用到自己的资料库。

Jerry 为他们的项目实现 **strcat** 函数。Jerry 可以创建自己的代码路径发送到 Tom。那么他就可以收到 Jerry 的代码补丁。

杰里使用 **Git format-patch** 命令来创建最新提交的补丁。如果想创建补丁具体提交，然后使用 **COMMIT_ID** 和 **ormat-patch** 命令。

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git status -s
M string_operations.c
?? string_operations

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m "Added my_strcat function"

[master b4c7f09] Added my_strcat function
1 files changed, 13 insertions(+), 0 deletions(-)

[jerry@CentOS src]$ git format-patch -1
0001-Added-my_strcat-function.patch
```

上面的命令创建 **.patch** 文件里在当前工作目录。Tom 可以使用这个补丁修改他的文件。Git 提供两个命令来应用补丁调幅分别为：**git am** 和 **git apply**。Git **apply** 命令修改本地文件时，而无需创建提交，**git am** 命令修改文件，会一并创建提交。

适用于修补程序并创建提交使用下面的命令。

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src
```

```
[tom@CentOS src]$ git diff

[tom@CentOS src]$ git status -s

[tom@CentOS src]$ git apply 0001-Added-my_strcat-function.patch

[tom@CentOS src]$ git status -s
M string_operations.c
?? 0001-Added-my_strcat-function.patch
```

补丁得到成功应用，现在我们可以使用 `git diff` 命令查看修改。

```
[tom@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..f282fcf 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,5 +1,16 @@
#include <stdio.h>
+char *my_strcat(char *t, char *s)
diff --git a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..f282fcf 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,5 +1,16 @@
#include <stdio.h>
+char *my_strcat(char *t, char *s)
+{
+
+char *p = t;
+
+
+
```

```
while (*p)
++p;
+
while (*p++ = *s++)
+ ;
+ return t;
+}
+
size_t my_strlen(const char *s)
{
const char *p = s;
@@ -23,6 +34,7 @@ int main(void)
{
```

Git 管理分支 - Git 教程

分支操作可以创造另一条线的发展。对 **fork** 过程分为两个不同的方向发展，我们可以使用此操作。例如，我们发布了 **6.0** 版本的产品，我们可能要创建一个分支，使 **7.0** 功能的发展可以保持独立从 **6.0** bug 修复。

创建分支

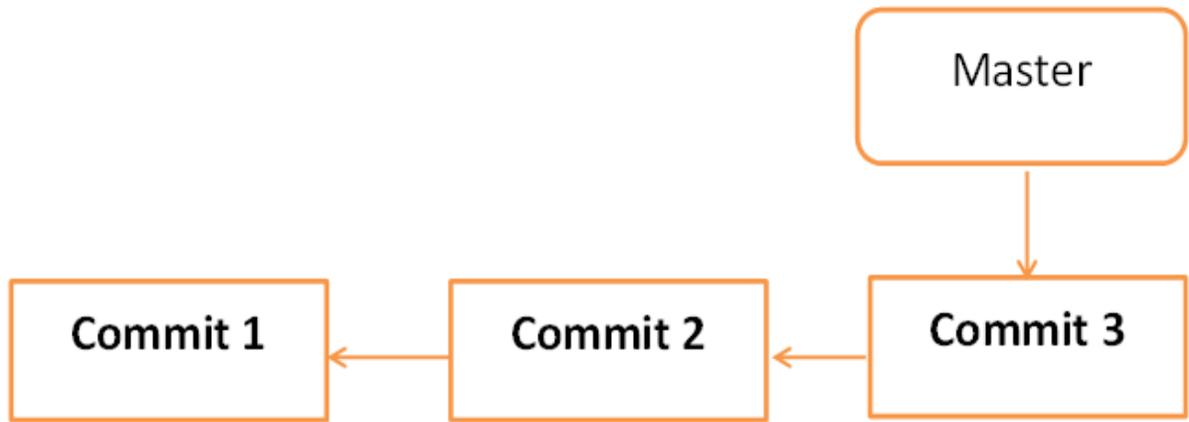
使用 **Git** 分支 `<branch name>` 命令创建新的分支。从现有的，我们可以创建一个新的分支。我们可以使用特定的提交或标签作为一个起点。如果没有提供任何具体的提交 **ID**，然后分支将 **HEAD** 创建作为一个起点。

```
[jerry@CentOS src]$ git branch new_branch
```

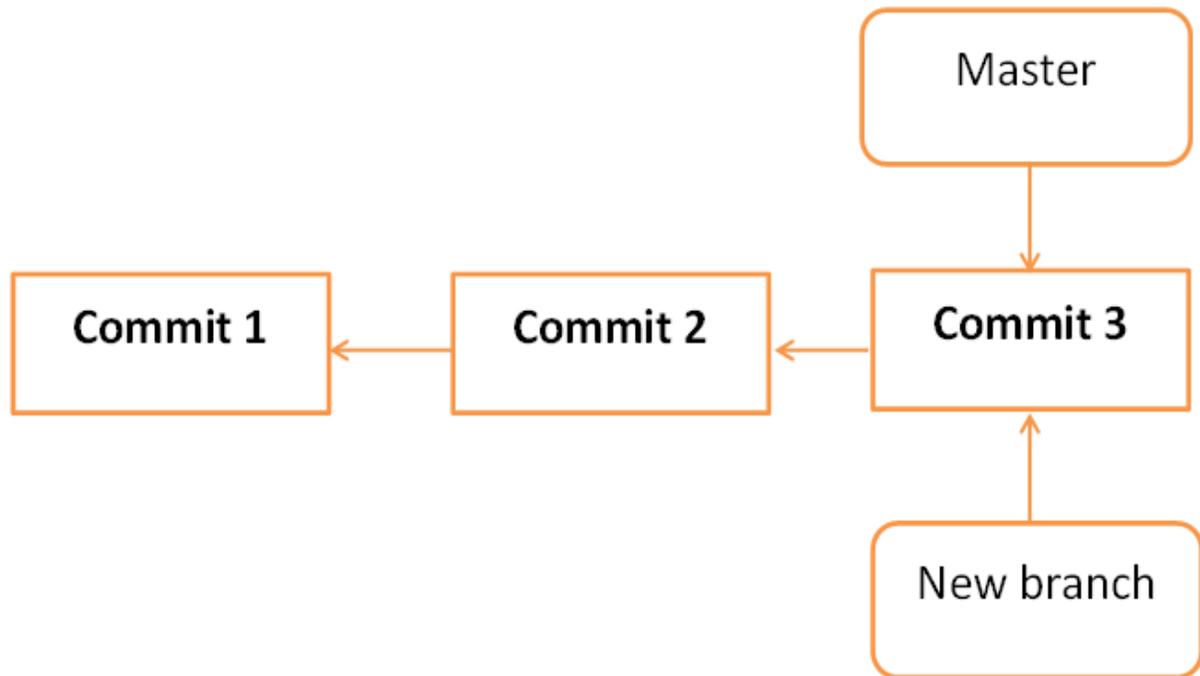
```
[jerry@CentOS src]$ git branch
* master
new_branch
```

创建新的分支，Tom 用 `git branch` 命令列出可用的分支。**Git** 会显示星号标记之前，当前检出的分支。

下面是创建分支操作的图形表示



Before create branch command



After create branch operation

切换分支

Jerry 使用 `git checkout` 命令到分支之间切换。

```
[jerry@CentOS src]$ git checkout new_branch
Switched to branch 'new_branch'
[jerry@CentOS src]$ git branch
master
* new_branch
```

创建和切换分支的快捷方式

在上面的例子中，我们使用了两个命令来创建和切换分支。Git 提供 `checkout` 命令 `-b` 选项，此操作将创建新的分支，并立即切换到新的分支。

```
[jerry@CentOS src]$ git checkout -b test_branch
Switched to a new branch 'test_branch'

[jerry@CentOS src]$ git branch
master
new_branch
* test_branch
```

删除分支

一个分支可以用 `git branch` 命令的 `-D` 选项被删除。但在此之前，删除现有的分支切换到其他分支。

Jerry 当前在 `test_branch` 想要删除该分支。于是，他分支和删除分支切换，如下图所示。

```
[jerry@CentOS src]$ git branch
master
new_branch
* test_branch

[jerry@CentOS src]$ git checkout master
```

```
Switched to branch 'master'
```

```
[jerry@CentOS src]$ git branch -D test_branch  
Deleted branch test_branch (was 5776472).
```

现在 Git 会显示只有两个分支。

```
[jerry@CentOS src]$ git branch  
* master  
new_branch
```

重命名分支

Jerry 决定添加宽字符支持他的字符串操作项目。他已经创建了一个新的分支，但分支名称是不恰当的。于是，他通过使用 `-m` 选项，其次是旧分支名称和新分支名称变更分支名称。

```
[jerry@CentOS src]$ git branch  
* master  
new_branch  
  
[jerry@CentOS src]$ git branch -m new_branch wchar_support
```

现在 `git branch` 命令将显示新分支名称。

```
[jerry@CentOS src]$ git branch  
* master  
wchar_support
```

合并两个分支

Jerry 实现函数返回字符串的长度为宽字符串。新代码将看起来像这样

```
[jerry@CentOS src]$ git branch  
master
```

```
* wchar_support

[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
t a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..8fb4b00 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,4 +1,14 @@
#include <stdio.h>
+#include <wchar.h>
+
+size_t w_strlen(const wchar_t *s)
+{
+
+const wchar_t *p = s;
+
+while (*p)
+ ++p;
+ return (p - s);
+}
```

测试后，他提交他的变化，并推到新的分支。

```
[jerry@CentOS src]$ git status -s
M string_operations.c
?? string_operations

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m 'Added w_strlen function to return
```

```
string length of wchar_t  
string'
```

```
[wchar_support 64192f9] Added w_strlen function to return string  
length of wchar_t string  
1 files changed, 10 insertions(+), 0 deletions(-)
```

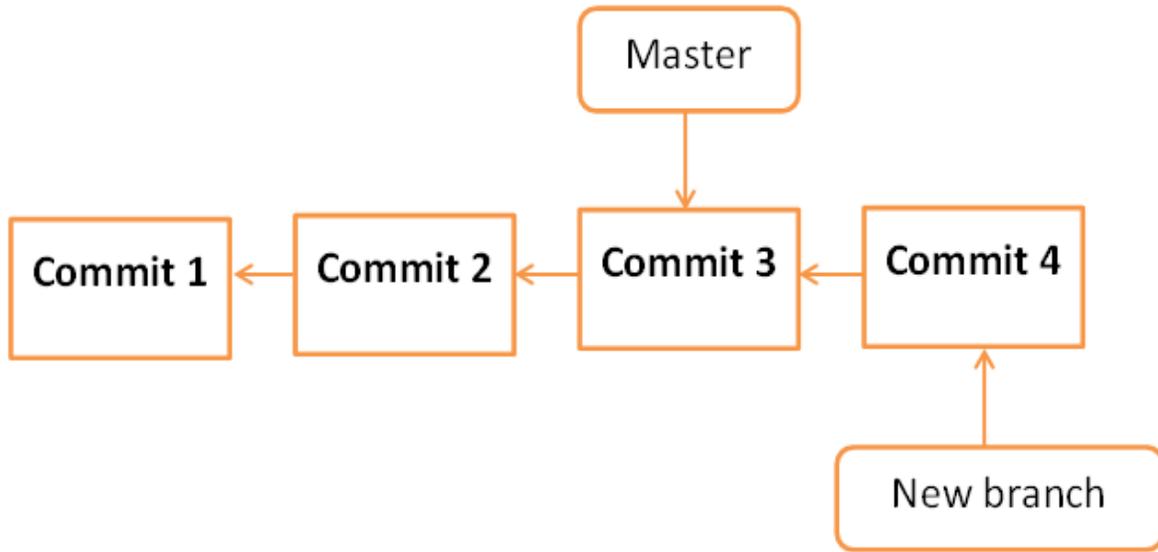
注意杰里推动这些变化的新分支，这就是为什么他用 `wchar_support` 分支的名称，而不是 `master` 分支。

```
[jerry@CentOS src]$ git push origin wchar_support **<-----  
Observer branch_name**
```

上面的命令会产生以下结果。

```
Counting objects: 7, done.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (4/4), 507 bytes, done.  
Total 4 (delta 1), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
* [new branch]  
wchar_support -> wchar_support
```

经过分支提交的变化，新分支会这个样子。



After commit in new branch

Tom 好奇 Jerry 在做什么在他的私人分支，这就是为什么他检查日志从 wchar_support 分支。

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src

[tom@CentOS src]$ git log origin/wchar_support -2
```

上面的命令会产生以下结果。

```
commit 64192f91d7cc2bcdf3bf946dd33ece63b74184a3
Author: Jerry Mouse <jerry@yiibai.com>
Date: Wed Sep 11 16:10:06 2013 +0530

Added w_strlen function to return string lenght of wchar_t string
```

```
commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@yiibai.com>
Date: Wed Sep 11 10:21:20 2013 +0530
```

```
Removed executable binary
```

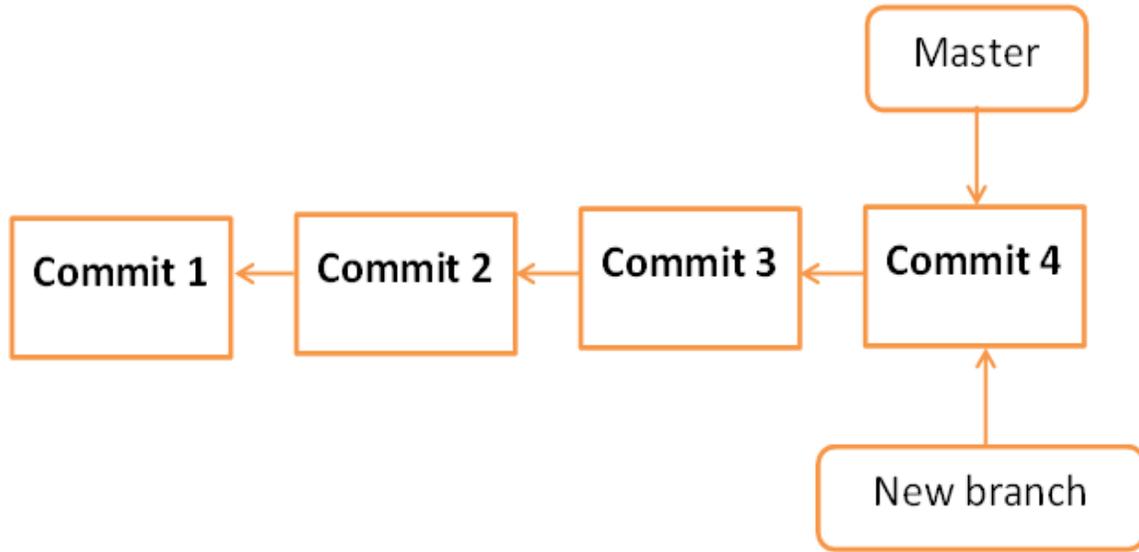
通过查看提交的信息，Tom 意识到 Jerry 实现宽字符 `strlen` 函数，他希望同样的功能集成到主分支。而不是重新实现他的分支合并到主分支，他决定采用杰里的代码。

```
[tom@CentOS project]$ git branch
* master

[tom@CentOS project]$ pwd
/home/tom/top_repo/project

[tom@CentOS project]$ git merge origin/wchar_support
Updating 5776472..64192f9
Fast-forward
src/string_operations.c | 10 ++++++++
1 files changed, 10 insertions(+), 0 deletions(-)
```

合并操作后的主分支会这个样子。



After branch merge

现在 `wchar_support` 分支合并到主分支中我们可以验证它的查看提交信息，通过查看修改成 `string_operation.c` 文件。

```
[tom@CentOS project]$ cd src/

[tom@CentOS src]$ git log -1

commit 64192f91d7cc2bcdf3bf946dd33ece63b74184a3
Author: Jerry Mouse <jerry@yiibai.com>Date: Wed Sep 11 16:10:06 2013
+0530

Added w_strlen function to return string length of wchar_t string

[tom@CentOS src]$ head -12 string_operations.c</jerry@yiibai.com>
```

上面的命令会产生以下结果。

```
#include <stdio.h>
#include <wchar.h>
size_t w_strlen(const wchar_t *s)
{
    const wchar_t *p = s;

    while (*p)
        ++p;

    return (p - s);
}
```

经过测试，他把他的代码更改到主分支。

```
[tom@CentOS src]$ git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
5776472..64192f9 master -> master
```

重订分支

Git 的 `rebase` 命令的一个分支合并的命令，但不同的是，它修改提交的顺序。

Git `merge` 命令，试图把从其他分支提交当前的本地分支的 `HEAD` 上。例如本地的分支已经提交 `A-> B-> C-> D` 和合并分支已提交 `A-> B-> X-> Y`，则 Git 合并并将当前转换像这样的本地分行 `A-> B-> C-> D-> X-> Y`

Git 的 `rebase` 命令试图找到当前的本地分支和合并分支之间的共同祖先。然后把修改提交的顺序，在当前的本地分支提交中的本地分支。例如，如果当地的分支已提交 `A-> B-> C-> D` 和合并分支已提交 `A-> B-> X-> Y`，Git 衍合的类似 `A-> B` 转换成当前的本地分支 `A->B->X->Y->C->D`

当多个开发人员在一个单一的远程资源库的工作，你不能在远程仓库提交修改订单。在这种情况下，可以使用变基操作把本地提交的远程仓库之上的提交，可以推送这些变化。

Git 冲突处理 - Git 教程

执行 wchar_support 分支变化

Jerry 工作在 wchar_support 分支。他改变了名称的功能和测试后，他提交他的变化。

```
[jerry@CentOS src]$ git branch
master
* wchar_support
[jerry@CentOS src]$ git diff
```

上面的命令产生以下结果

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 8fb4b00..01ff4e0 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,7 +1,7 @@
#include <stdio.h>
#include <wchar.h>
-size_t w_strlen(const wchar_t *s)
+size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;
```

验证代码后，他提交了他的变化。

```
[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m 'Changed function name'
[wchar_support 3789fe8] Changed function name
1 files changed, 1 insertions(+), 1 deletions(-)
```

```
[jerry@CentOS src]$ git push origin wchar_support
```

上面的命令会产生以下结果。

```
Counting objects: 7, done.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (4/4), 409 bytes, done.  
Total 4 (delta 1), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
64192f9..3789fe8 wchar_support -> wchar_support
```

在主分支进行更改

同时主分支 Tom 也改变了名称相同的功能，并将其更改到主分支。

```
[tom@CentOS src]$ git branch  
* master  
[tom@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
diff --git a/src/string_operations.c b/src/string_operations.c  
index 8fb4b00..52bec84 100644  
--- a/src/string_operations.c  
+++ b/src/string_operations.c  
@@ -1,7 +1,8 @@  
#include <stdio.h>  
#include <wchar.h>  
-size_t w_strlen(const wchar_t *s)  
+/* wide character strlen function */  
+size_t my_wc_strlen(const wchar_t *s)  
{  
const wchar_t *p = s;
```

验证差异后，他提交了他的变化。

```
[tom@CentOS src]$ git status -s
M string_operations.c

[tom@CentOS src]$ git add string_operations.c

[tom@CentOS src]$ git commit -m 'Changed function name from w_strlen
to my_wc_strlen'
[master ad4b530] Changed function name from w_strlen to my_wc_strlen
1 files changed, 2 insertions(+), 1 deletions(-)

[tom@CentOS src]$ git push origin master
```

上面的命令会产生以下结果。

```
Counting objects: 7, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 470 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
64192f9..ad4b530 master -> master
```

`strchr` 函数在分支 `Jerry` 实现 `wchar_support` 宽字符串。经过测试，他提交和推送其变化 `wchar_support` 分支。

```
[jerry@CentOS src]$ git branch
master
* wchar_support
[jerry@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 01ff4e0..163a779 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,6 +1,16 @@
```

```

#include <stdio.h>
#include <wchar.h>
wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
while (*ws) {
+
if (*ws == wc)
+
return ws;
+
++ws;
+ }
+ return NULL;
+}
+
size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;

```

验证变化后，他提交他的变化。

```

[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m 'Added strchr function for wide
character string'
[wchar_support 9d201a9] Added strchr function for wide character
string
1 files changed, 10 insertions(+), 0 deletions(-)

[jerry@CentOS src]$ git push origin wchar_support

```

上面的命令会产生以下结果。

```
Counting objects: 7, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 516 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
3789fe8..9d201a9 wchar_support -> wchar_support
```

对付冲突

Tom 想看, Jerry 在他的私人分支做什么? 这就是为什么他试图从 `wchar_support` 分支把最新的修改, 但 Git 放弃操作在得到错误消息后。

```
[tom@CentOS src]$ git pull origin wchar_support
```

上面的命令会产生以下结果。

```
remote: Counting objects: 11, done.
63Git Tutorials
remote: Compressing objects: 100% (8/8), done.
remote: Total 8 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From git.server.com:project
* branch
wchar_support -> FETCH_HEAD
Auto-merging src/string_operations.c
**CONFLICT (content): Merge conflict in src/string_operations.c**
Automatic merge failed; fix conflicts and then commit the result.
```

解决冲突

从错误消息很显然知道, 是有冲突的在 `src/string_operations.c`。他运行 `git diff` 命令查看进一步的细节。

```
[tom@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
diff --cc src/string_operations.c
index 52bec84,163a779..0000000
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@@ -1,8 -1,17 +1,22 @@@
#include <stdio.h>
#include <wchar.h>
++<<<<<<< HEAD
+/* wide character strlen fucntion */
+size_t my_wc_strlen(const wchar_t *s)
++=====
+ wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
+
+ while (*ws) {
+ if (*ws == wc)
+
+ return ws;
+
+ ++ws;
+ }
+ return NULL;
+}
+
+ size_t my_wstrlen(const wchar_t *s)
++>>>>>>>9d201a9c61bc4713f4095175f8954b642dae8f86
+ {
+ const wchar_t *p = s;
```

由于 Tom 和 Jerry 在相同的功能更改的名称，Git 不知道如何去做，因此这就是为什么它要求用户手动解决冲突。

Tom 决定保用 Jerry 建议的函数名，但他使用原来注释，因为这是他加入的。删除冲突标记混帐后差异会看起来像这样。

```
[tom@CentOS src]$ git diff
```

上面的命令会产生以下结果。

```
diff --cc src/string_operations.c
diff --cc src/string_operations.c
index 52bec84,163a779..0000000
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@@ -1,8 -1,17 +1,18 @@@
#include <stdio.h>
#include <wchar.h>
+ wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
+ while (*ws) {
+
+ if (*ws == wc)
+
+ return ws;
+
+ ++ws;
+ }
+ return NULL;
+}
+
+/* wide character strlen fucntion */
- size_t my_wc_strlen(const wchar_t *s)
+ size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;
```

Tom 修改过的文件，他先提交这些更改后，他就可以推送了。

```
[tom@CentOS src]$ git commit -a -m 'Resolved conflict'
[master 6b1ac36] Resolved conflict
```

```
[tom@CentOS src]$ git pull origin wchar_support.
```

Tom 已经解决冲突，现在推送操作将成功。

Git 不同的平台 - Git 教程

GNU/ [Linux](#) 和 Mac OS 使用换行符（LF）或新行作为行结束字符，而 Windows 使用换行和回车（LFCR）的组合来表示行结束字符。

为了避免不必要的提交，因为这些行结束的差异，Git 客户端配置写在同一行结束 Git 仓库。

对于 Windows 系统中，我们可以配置的 Git 客户端换行符转换为 CRLF 格式，同时检查了，并把它们转换回 LF 格式提交操作过程中。下面设置将不可少。

```
[tom@CentOS project]$ git config --global core.autocrlf true
```

对于 GNU/ Linux 或 Mac OS，我们可以配置 Git 客户端 CRLF 到 LF 换行符转换，同时进行检出（checkout）操作。

```
[tom@CentOS project]$ git config --global core.autocrlf input
```

GitHub 在线存储库 - Git 教程

GitHub 是使用 Git 的版本控制系统是一个基于网络的托管服务的软件开发项目。它也有其标准的 GUI 应用程序可供下载（在 Windows, Mac, GNU/Linux）的直接从服务的网站。但在这个环节中，我们将只能看到 CLI 部分。

创建 GitHub 的资料库

去到 github.com. 如果您已经 GitHub 的帐户，然后使用该帐户登录，或创建新的。从 github.com 网站按照以下步骤来创建新的存储库。

推送操作

Tom 决定使用 GitHub 上服务器。要开始新的项目，他将创建一个新的目录和一个文件里面。

```
[tom@CentOS]$ mkdir github_repo

[tom@CentOS]$ cd github_repo/

[tom@CentOS]$ vi hello.c

[tom@CentOS]$ make hello
cc hello.c -o hello

[tom@CentOS]$ ./hello
```

上面的命令会产生以下结果。

```
Hello, World !!!
```

在验证自己的代码后，他在初始化目录用 `git init` 命令在本地提交他的变化。

```
[tom@CentOS]$ git init
Initialized empty Git repository in /home/tom/github_repo/.git/

[tom@CentOS]$ git status -s
```

```
?? hello
?? hello.c

[tom@CentOS]$ git add hello.c

[tom@CentOS]$ git status -s
A hello.c
?? hello

[tom@CentOS]$ git commit -m 'Initial commit'
```

之后，他增加了 **GitHub** 的版本库 **URL** 作为一个远程的起源，并推他到远程仓库。

> 注：我们已经讨论了所有这些步骤在第 4 章下创建裸库部分。

```
[tom@CentOS]$ git remote add origin
https://github.com/kangralkar/testing_repo.git

[tom@CentOS]$ git push -u origin master
```

推送操作会询问 **GitHub** 的用户名和密码。验证成功后，操作会成功。

上面的命令会产生以下结果。

```
Username for 'https://github.com': kangralkar
Password for 'https://kangralkar@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 214 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/kangralkar/test_repo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

从现在开始，**Tom** 可以在 **GitHub** 库做任何更改。他可以使用本章讨论的所有命令在 **GitHub** 的仓库中。

Pull 操作

Tom 成功地把他的所有变化 **GitHub** 的库。现在其他开发人员可以查看这些更改进行克隆操作或更新他们的本地资源库。

Jerry 在他的 **home** 目录和克隆的 **GitHub** 库使用 **git clone** 命令创建新的目录。

```
[jerry@CentOS]$ pwd
/home/jerry

[jerry@CentOS]$ mkdir jerry_repo

[jerry@CentOS]$ git clone https://github.com/kangralkar/test_repo.git
```

上面的命令会产生以下结果。

```
Cloning into 'test_repo'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
```

他验证通过执行 **ls** 命令的目录内容。

```
[jerry@CentOS]$ ls
test_repo

[jerry@CentOS]$ ls test_repo/
hello.c
```

Git 远程操作详解 - Git 教程

Git 是目前最流行的版本管理系统，学会 Git 几乎成了开发者的必备技能。

Git 有很多优势，其中之一就是远程操作非常简便。本文详细介绍 5 个 Git 命令，它们的概念和用法，理解了这些内容，你就会完全掌握 Git 远程操作。

- [git clone](#)
- [git remote](#)
- [git fetch](#)
- [git pull](#)
- [git push](#)

本文针对初级用户，从最简单的讲起，但是需要读者对 Git 的基本用法有所了解。同时，本文覆盖了上面 5 个命令的几乎所有的常用用法，所以对于熟练用户也有参考价值。

