# Git

## Git基础概念

Git 是分布式版本控制和源代码管理系统，Git 最初是由Linus Torvalds设计开发的，用于管理Linux内核开发。Git 是根据GNU通用公共许可证版本2的条款分发的自由/免费软件。

Git官方网址是：https://git-scm.com/

### 关于版本控制

什么是“版本控制”？为什么要关心它呢？ 版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。在本书所展示的例子中，我们对保存着软件源代码的文件作版本控制，但实际上，可以对任何类型的文件进行版本控制。

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本(这或许是你非常渴望拥有的功能)，采用版本控制系统(VCS)是个明智的选择。 有了它就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。 使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改，删的删了，这也没有关系，你也照样可以很容易地就恢复到原先的样子。但额外增加的工作量却微乎其微。

### 本地版本控制系统

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。 这么做唯一的好处就是简单，但是特别容易犯错。 有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖意想外的文件。

为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。

Local Computer

Version Database

Checkout

File

Version3

Version2

Version1

其中最流行的一种叫做 RCS，现今许多计算机系统上都还看得到它的踪影。 甚至在流行的 Mac OS X 系统上安装了开发者工具包之后，也可以使用 rcs 命令。 它的工作原理是在硬盘上保存补丁集(补丁是指文件修订前后的变化)；通过应用所有的补丁，可以重新计算出各个版本的文件内容。

### 集中化的版本控制系统

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？ 于是，集中化的版本控制系统(Centralized Version Control Systems，简称 CVCS)应运而生。 这类系统，诸如 CVS、Subversion(SVN) 以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。多年以来，这已成为版本控制系统的标准做法。

Central VCS Server

Computer A

Version3

Version2

Version1

Version Database

File

Computer B

File

这种做法带来了许多好处，特别是相较于老式的本地 VCS 来说。 现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么。 而管理员也可以轻松掌控每个开发者的权限，并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易。

事分两面，有好有坏。 这么做最显而易见的缺点是中央服务器的单点故障。 如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。 如果中心数据库所在的磁盘发生损坏，又没有做恰当备份，毫无疑问您将丢失所有数据——包括项目的整个变更历史，只剩下人们在各自机器上保留的单独快照。本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

### 分布式版本控制系统

于是分布式版本控制系统(Distributed Version Control System，简称 DVCS)面世了。 在这类系统中，像 Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。 这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。如下图所示

Server Computer

Version Database

Version1

Version2

Version3

Version3

Version2

Version1

Version Database

Computer A

File

Computer A

File

Version3

Version2

Version1

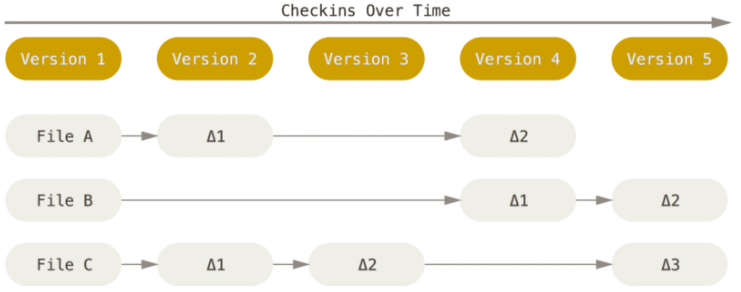
Version Database

更进一步，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。藉此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

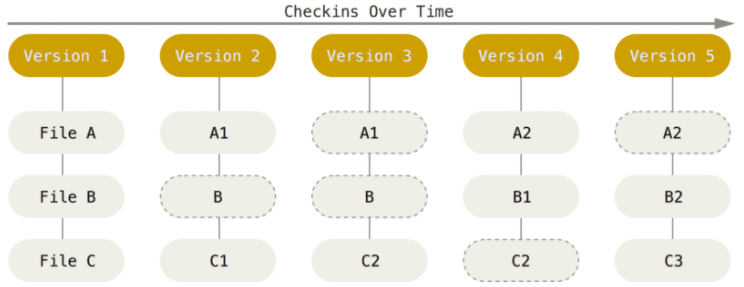
## Git基础和原理

**直接记录快照，而非差异比较**

Git 和其它版本控制系统(包括 Subversion 和近似工具)的主要差别在于 Git 对待数据的方法。 概念上来区分，其它大部分系统以文件变更列表的方式存储信息。 这类系统(CVS、Subversion、Perforce、Bazaar 等等)将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。存储每个文件与初始版本的差异，如下图所示



Git 不按照以上方式对待或保存数据。 反之，Git 更像是把数据看作是对小型文件系统的一组快照。 每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个 快照流。如下图所示



这是 Git 与几乎所有其它版本控制系统的重要区别。 因此 Git 重新考虑了以前每一代版本控制系统延续下来的诸多方面。 Git 更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不只是一个简单的 VCS。 稍后我们在 Git 分支讨论 Git 分支管理时，将探究这种方式对待数据所能获得的益处。

**近乎所有操作都是本地执行**

在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其它计算机的信息。 如果你习惯于所有操作都有网络延时开销的集中式版本控制系统，Git 在这方面会让你感到速度之神赐给了 Git 超凡的能量。 因为你在本地磁盘上就有项目的完整历史，所以大部分操作看起来瞬间完成。

举个例子，要浏览项目的历史，Git 不需外连到服务器去获取历史，然后再显示出来——它只需直接从本地数据库中读取。 你能立即看到项目历史。 如果想查看当前版本与一个月前的版本之间引入的修改，Git 会查找到一个月前的文件做一次本地的差异计算，而不是由远程服务器处理或从远程服务器拉回旧版本文件再来本地处理。

这也意味着你离线或者没有 VPN 时，几乎可以进行任何操作。 如你在飞机或火车上想做些工作，你能愉快地提交，直到有网络连接时再上传。 如你回家后 VPN 客户端不正常，你仍能工作。 使用其它系统，做到如此是不可能或很费力的。 比如，用 Perforce，你没有连接服务器时几乎不能做什么事；用 Subversion 和 CVS，你能修改文件，但不能向数据库提交修改(因为你的本地数据库离线了)。 这看起来不是大问题，但是你可能会惊喜地发现它带来的巨大的不同。

**Git 保证完整性**

Git 中所有数据在存储前都计算校验和，然后以校验和来引用。 这意味着不可能在 Git 不知情时更改任何文件内容或目录内容。 这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。 若你在传送过程中丢失信息或损坏文件，Git 就能发现。

Git 用以计算校验和的机制叫做 SHA-1 散列(hash，哈希)。 这是一个由 40 个十六进制字符(0-9 和 a-f)组成字符串，基于 Git 中文件的内容或目录结构计算出来。 SHA-1 哈希看起来是这样：



Git 中使用这种哈希值的情况很多，你将经常看到这种哈希值。 实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

**Git 一般只添加数据**

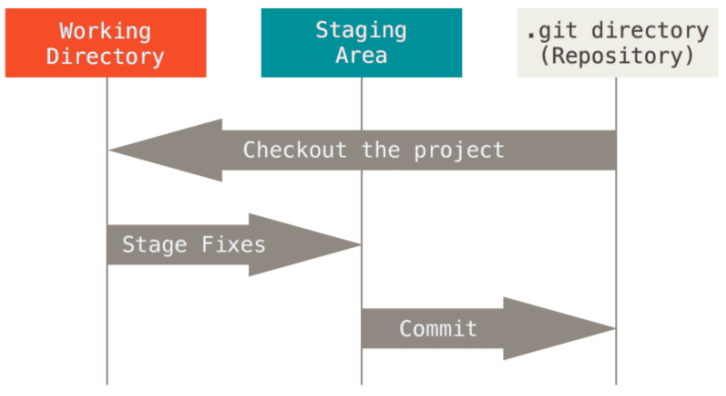
你执行的 Git 操作，几乎只往 Git 数据库中增加数据。 很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。 同别的 VCS 一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦你提交快照到 Git 中，就难以再丢失数据，特别是如果你定期的推送数据库到其它仓库的话。

这使得我们使用 Git 成为一个安心愉悦的过程，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。 更深度探讨 Git 如何保存数据及恢复丢失数据的话题，请参考撤消操作。

**三种状态**

Git 有三种状态，你的文件可能处于其中之一：已提交(committed)、已修改(modified)和已暂存(staged)。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。工作目录、暂存区域以及 Git 仓库如下图所示



Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。 这是 Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

工作目录是对项目的某个版本独立提取出来的内容。 这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。 有时候也被称作‘索引’，不过一般说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改文件。
2. 暂存文件，将文件的快照放入暂存区域。
3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

如果 Git 目录中保存着的特定版本文件，就属于已提交状态。 如果作了修改并已放入暂存区域，就属于已暂存状态。 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

## Git安装设置

### 在 Linux 上安装

如果你想在 Linux 上用二进制安装程序来安装 Git，可以使用发行版包含的基础软件包管理工具来安装。 如果以 Fedora 上为例，你可以使用 yum：

$ sudo yum install git

如果你在基于 Debian 的发行版上，请尝试用 apt-get

$ sudo apt-get install git

安装完成后，查看当前安装的 Git 的版本

yiibai@ubuntu:~$ git version 2.7.4

git version 2.7.4

### 在 Mac 上安装

在 Mac 上安装 Git 有多种方式。 最简单的方法是安装 Xcode Command Line Tools。 Mavericks (10.9) 或更高版本的系统中，在 Terminal 里尝试首次运行 git 命令即可。 如果没有安装过命令行开发者工具，将会提示你安装。

如果你想安装更新的版本，可以使用二进制安装程序。 官方维护的 OSX Git 安装程序可以在 Git 官方网站下载，网址为 http://git-scm.com/download/mac。



Git OS X 安装程序.

你也可以将它作为 GitHub for Mac 的一部分来安装。 它们的图形化 Git 工具有一个安装命令行工具的选项。 你可以从 GitHub for Mac 网站下载该工具，网址为 http://mac.github.com。

### 在 Windows 上安装

在 Windows 上安装 Git 也有几种安装方法。 官方版本可以在 Git 官方网站下载。 打开 http://git-scm.com/download/win ，下载会自动开始。 要注意这是一个名为 Git for Windows的项目(也叫做 msysGit)，和 Git 是分别独立的项目；



另一个简单的方法是安装 GitHub for Windows。 该安装程序包含图形化和命令行版本的 Git。 它也能支持 Powershell，提供了稳定的凭证缓存和健全的 CRLF 设置。 稍后我们会对这方面有更多了解，现在只要一句话就够了，这些都是你所需要的。 你可以在 GitHub for Windows 网站下载，网址为 http://windows.github.com

### 从源代码安装

有人觉得从源码安装 Git 更实用，因为你能得到最新的版本。 二进制安装程序倾向于有一些滞后，当然近几年 Git 已经成熟，这个差异不再显著。

注意：从源码安装 Git 依懒包会比较，可能比较费时。

如果你想从源码安装 Git，需要安装 Git 依赖的库：curl、zlib、openssl、expat，还有libiconv。 如果你的系统上有 yum (如 Fedora)或者 apt-get(如基于 Debian 的系统)，可以使用以下命令之一来安装最小化的依赖包来编译和安装 Git 的二进制版：

$ sudo yum install curl-devel expat-devel gettext-devel

openssl-devel zlib-devel

$ sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext

libz-dev libssl-dev

为了能够添加更多格式的文档(如 doc, html, info)，你需要安装以下的依赖包：

$ sudo yum install asciidoc xmlto docbook2x

$ sudo apt-get install asciidoc xmlto docbook2x

当你安装好所有的必要依赖，你可以继续从几个地方来取得最新发布版本的 tar 包。 你可以从 Kernel.org 网站获取，网址为 http://www.kernel.org/pub/software/scm/git，或从 GitHub 网站上的镜像来获得，网址为 http://github.com/git/git/releases。 通常在 GitHub 上的是最新版本，但 kernel.org 上包含有文件下载签名，如果你想验证下载正确性的话会用到。

接着，编译并安装：

$ tar -zxf git-2.0.0.tar.gz

$ cd git-2.0.0

$ make configure

$ ./configure --prefix=/usr

$ make all doc info

$ sudo make install install-doc install-html install-info

完成后，你可以使用 Git 来获取 Git 的升级：

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

## Git使用前配置

已经在系统上安装了 Git，那么现在要做几件事来配置/定制 Git 环境。每台计算机上只需要配置一次，程序升级时会保留配置信息。可以在任何时候再次通过运行命令来修改它们。

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位置：

1. /etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果使用带有 --system 选项的 git config 时，它会从此文件读写配置变量。
2. ~/.gitconfig 或 ~/.config/git/config 文件：只针对当前用户。 可以传递--global 选项让 Git 读写此文件。

当前使用仓库的 Git 目录中的 config 文件(就是 .git/config)：针对该仓库。

每一个级别覆盖上一级别的配置，所以 .git/config 的配置变量会覆盖 /etc/gitconfig 中的配置变量。

在 Windows 系统中，Git 会查找 $HOME 目录下(一般情况下是 C:\Users\$USER)的 .gitconfig 文件。 Git 同样也会寻找 /etc/gitconfig 文件，但只限于 MSys 的根目录下，即安装 Git 时所选的目标位置。

### 用户信息

当安装完 Git 应该做的第一件事就是设置用户名称与邮件地址。这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到每一次提交中，不可更改：

$ git config --global user.name "maxsu"

$ git config --global user.email maxsu@yiibai.com

再次强调，如果使用了 --global 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情， Git 都会使用这些信息。 当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行不使用 --global 选项的命令来配置。

很多 GUI 工具都会在第一次运行时帮助你配置这些信息。

### 文本编辑器

既然用户信息已经设置完毕，你可以配置默认文本编辑器了，当 Git 需要您输入信息时会调用它。 如果未配置，Git 会使用操作系统默认的文本编辑器，通常是 Vim。 如果你想使用不同的文本编辑器，例如 Emacs，可以这样做：

$ git config --global core.editor emacs

注：Vim 和 Emacs 是像 Linux 与 Mac 等基于 Unix 的系统上开发者经常使用的流行的文本编辑器。 如果你对这些编辑器都不是很了解或者你使用的是 Windows 系统，那么可能需要搜索如何在 Git 中配置你最常用的编辑器。 如果你不设置编辑器并且不知道 Vim 或 Emacs 是什么，当它们运行起来后你可能会被弄糊涂、不知所措。

### 检查配置信息

如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置。

yiibai@ubuntu:~$ git config --list

user.name=maxsu

user.email=maxsu@yiibai.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

上面的输出结果中，可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置(例如：/etc/gitconfig 与 ~/.gitconfig)。 这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

可以通过输入 git config <key>： 来检查 Git 的某一项配置。

yiibai@ubuntu:~$ git config user.name

maxsu

yiibai@ubuntu:~$ git config user.email

maxsu@yiibai.com

yiibai@ubuntu:~$

### 获取帮助

若在使用 Git 时需要获取帮助，有三种方法可以找到 Git 命令的使用手册：

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

例如，要想获得 config 命令的手册，执行

$ git help config

这些命令很棒，因为随时随地可以使用而无需联网。如果你觉得手册或者本书的内容还不够用，你可以尝试在 Freenode IRC 服务器( irc.freenode.net )的 #git 或 #github 频道寻求帮助。这些频道经常有上百人在线，他们都精通 Git 并且乐于助人。

## Git快速入门

### 远程仓库是什么

Repository(仓库)包含的内容 - Git的目标是管理一个工程，或者说是一些文件的集合，以跟踪它们的变化。Git使用Repository来存储这些信息。一个仓库主要包含以下内容(也包括其他内容)：

1. 许多commit objects
2. 到commit objects的指针，叫做heads
3. Git的仓库和工程存储在同一个目录下，在一个叫做.git的子目录中。

### 创建Repository(仓库)

在使用Repository(仓库)之前，我们首先需要创建仓库，创建仓库有很多种，这里常见的有如下几种：

自己搭建个 Git 服务器，安装如 GitLab 的Git版本管理系统

使用第三方托管平台，如国内的 http://git.oschina.net 和国外的 http://github.com/

这里使用第三方托管平台作为讲解，以 http://git.oschina.net 为例，大概需要通过以下几个步骤完成仓库的创建。

注册网站账号

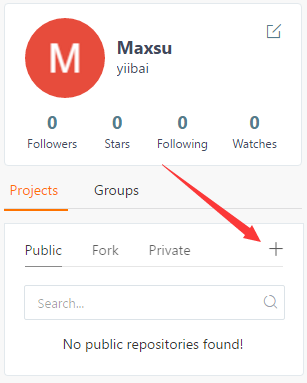
登录帐后，创建仓库

第一步：注册网站的帐号

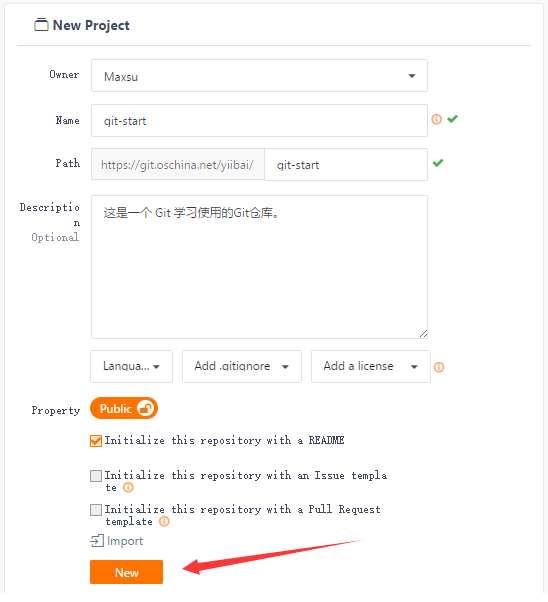
打开网址： http://git.oschina.net/signup ，写入一些必填项，然后提交，如下图所示



提交完成后，登录帐号还不能使用，还需要登录注册的邮箱验证帐号。验证邮箱验证帐号后，登录后默认的用户面板界面如下所示



点击红色箭头指向的”+“号，以创建一个仓库，如下所示



这样，一个公开的仓库就创建完成了。要记住上面图片创建的路径：http://git.oschina.net/yiibai/git-start.git

### 获取Git仓库

有两种取得 Git 项目仓库的方法。第一种是从一个服务器克隆一个现有的 Git 仓库。第二种是在现有项目或目录下导入所有文件到 Git 中

1. **克隆现有的仓库**

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令。 如果你对其它的 VCS 系统(比如说Subversion)很熟悉，请留心一下这里所使用的命令是”clone“而不是”checkout“。 这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。如果服务器的磁盘坏掉了，通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库。

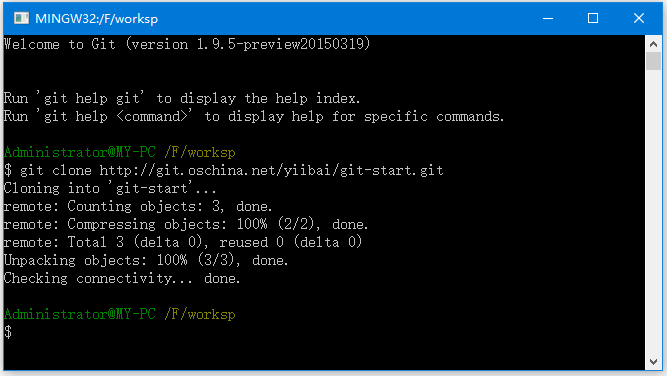
在安装了Git 的 Windows系统上，在一个目录(本示例是：F:\worksp)中，单击右键，在弹出的菜单中选择“Git Bash”，如下图中所示



克隆仓库的命令格式是 git clone [url] 。 比如，要克隆 Git 的上面创建的仓库 git-start.git，可以用下面的命令：

$ git clone http://git.oschina.net/yiibai/git-start.git

这会在当前目录下创建一个名为 “git-start.git” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。上面命令执行后，输出结果如下所示



如果想在克隆远程仓库的时候，自定义本地仓库的名字，可以使用如下命令：

$ git clone http://git.oschina.net/yiibai/git-start.git mygit-start

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 mygit-start。

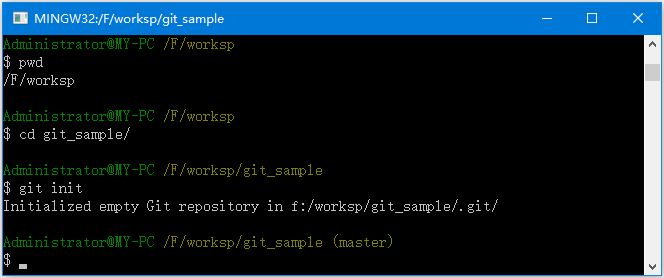
Git 支持多种数据传输协议。 上面的例子使用的是 http:// 协议，不过也可以使用 git:// 协议或者使用 SSH 传输协议，比如 user@server\_ip-or-host:path/to/repo.git 。在服务器上搭建 Git 将会介绍所有这些协议在服务器端如何配置使用，以及各种方式之间的利弊。

1. **在现有目录中初始化仓库**

如果不克隆现有的仓库，而是打算使用 Git 来对现有的项目进行管理。假设有一个项目的目录是：D:\worksp\git\_sample，只需要进入该项目的目录并输入：

$ git init

执行上面命令，输出结果如下



该命令将创建一个名为 .git 的子目录，这个子目录含有初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。 但是，在这个时候，我们仅仅是做了一个初始化的操作，项目里的文件还没有被跟踪。

如果是在一个已经存在文件的文件夹(而不是空文件夹)中初始化 Git 仓库来进行版本控制的话，应该开始跟踪这些文件并提交。可通过 git add 命令来实现对指定文件的跟踪，然后执行 git commit 提交，假设在目录 F:\worksp\git-start.git 中有一些代码需要跟踪(版本控制)，比如有一个 Python 代码文件叫作：hello.py 内容如下

#!/usr/bin/python3

#coding=utf-8

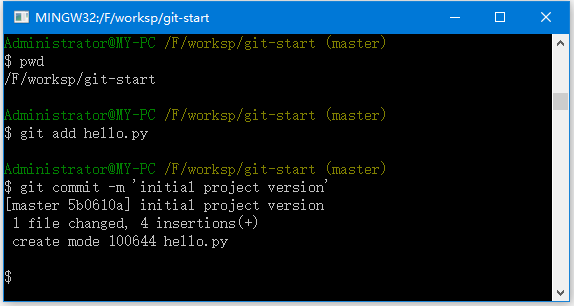
print ("This is my first Python Programming.")

可通过 git add 命令来实现对hello.py 文件的跟踪

$ git add hello.py

$ git commit -m 'initial project version'

上面命令执行结果如下



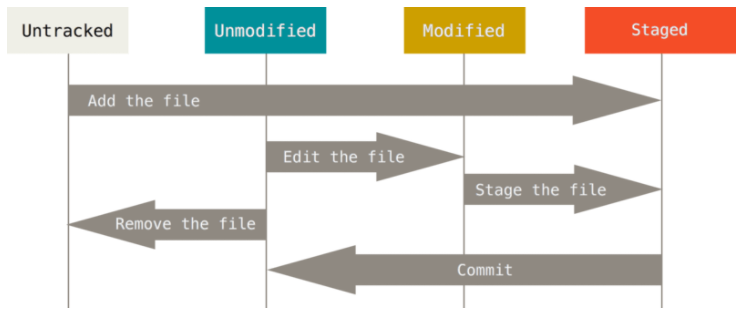
### 更新提交到仓库

1. **记录每次更新到仓库**

现在我们手上有了一个真实项目的 Git 仓库(如上面 clone 下来的 git-start.git)，并从这个仓库中取出了所有文件的工作拷贝。 接下来，对这些文件做些修改，在完成了一个阶段的目标之后，提交本次更新到仓库。

工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。 已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。 工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。 初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：

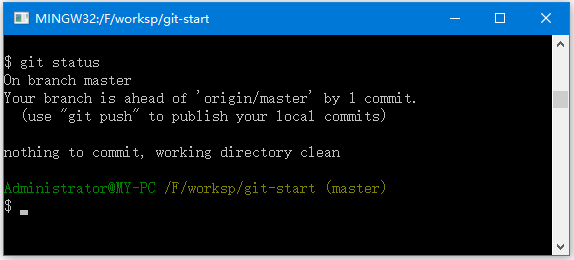


1. **检查当前文件状态**

要查看哪些文件处于什么状态，可以用 git status 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

$ git status

上面命令执行结果如下



这说明现在你的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。 此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。 最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。现在，分支名是 “master”, 这是默认的分支名。

现在，在项目下创建一个新的 mytext.txt 文件。 如果之前并不存在这个文件，使用 git status 命令，将看到一个新的未跟踪文件：

# 向 mytext.md 文件写入一点内容

$ echo 'This is my first Git control file ' > mytext.txt

$ git status

On branch master

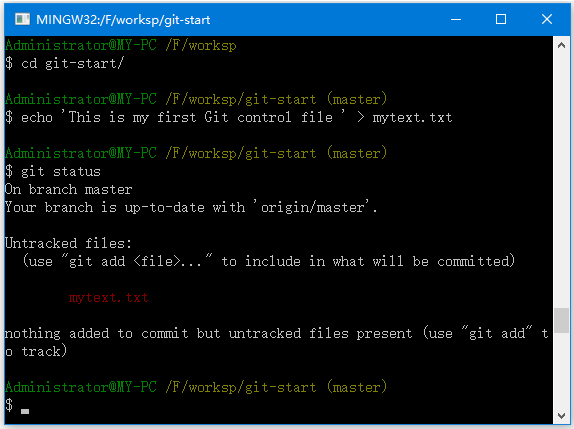
Untracked files:

(use "git add <file>..." to include in what will be committed)

mytext.txt

nothing added to commit but untracked files present (use "git add" to track)

上面命令执行结果如下



在状态报告中可以看到新建的 README 文件出现在 Untracked files 下面。 未跟踪的文件意味着 Git 在之前的快照(提交)中没有这些文件；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”， 这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。 不过现在的例子中，我们确实想要跟踪管理 README 这个文件。

1. **跟踪新文件**

使用命令 git add 开始跟踪一个文件。 所以，要跟踪 mytext.txt 文件，运行

$ git add mytext.txt

此时再运行 git status 命令，会看到 mytext.txt 文件已被跟踪，并处于暂存状态：

$ git status

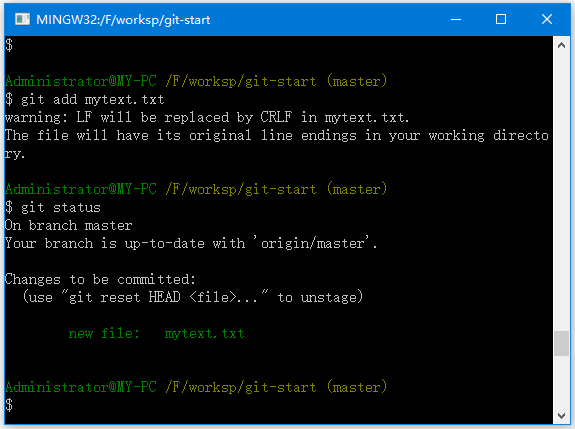
On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: mytext.txt

上面命令执行结果如下



只要在 Changes to be committed 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。git add 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

1. **暂存已修改文件**

现在我们来修改一个已被跟踪的文件。 如果修改了一个名为 README.md 的已被跟踪的文件，打开文件 README.md并编辑其中的内容，在文件的未尾加入一行内容：”这是暂存已修改文件示例”，然后运行 git status 命令，会看到下面内容：

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: mytext.txt

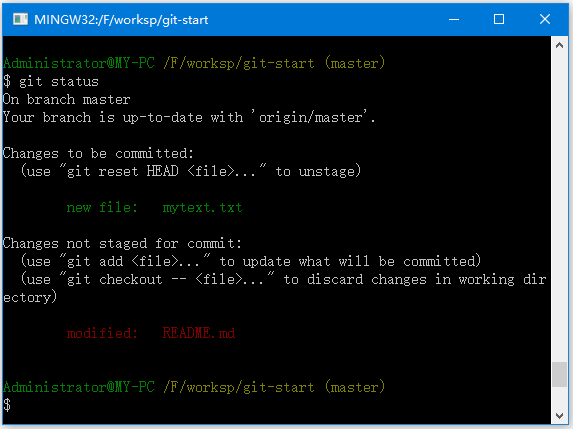
Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

执行上面命令得到以下结果



文件 README.md 出现在 Changes not staged for commit 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行 git add 命令。 这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。 将这个命令理解为“添加内容到下一次提交中”而不是“将一个文件添加到项目中”要更加合适。 现在让我们运行 git add 将”README.md“放到暂存区，然后再看看 git status 的输出：

$ git add README.md

Administrator@MY-PC /F/worksp/git-start (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README.md

new file: mytext.txt

Administrator@MY-PC /F/worksp/git-start (master)

$

现在两个文件都已暂存，下次提交时就会一并记录到仓库。 假设此时，想要在 README.md 里再加条注释， 重新编辑存盘后，准备好提交。不过且慢，先向 “README.md” 文件加入一点内容，再运行 git status ，如下所示

$ echo "Add new Line content 1002 " >> README.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: mytext.txt

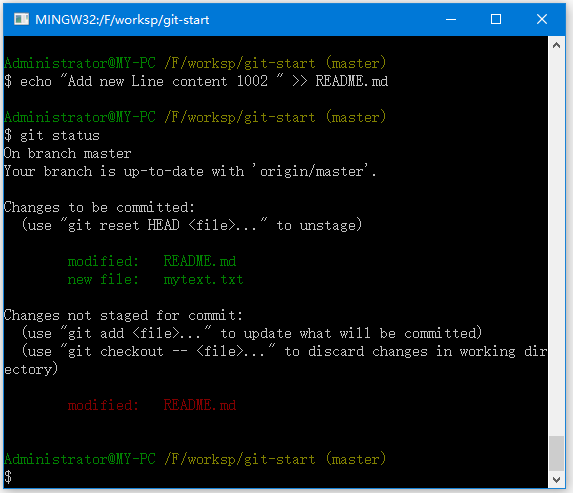
Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

执行上面命令，输出结果如下所示



怎么回事？ 现在 README.md 文件同时出现在暂存区和非暂存区。 这怎么可能呢？ 好吧，实际上 Git 只不过暂存了运行 git add 命令时的版本， 如果现在提交，README.md 的版本是最后一次运行 git add 命令时的那个版本，而不是运行 git commit 时，在工作目录中的当前版本。 所以，运行了 git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存起来：

$ git add README.md

Administrator@MY-PC /F/worksp/git-start (master)

$ git status

warning: LF will be replaced by CRLF in README.md.

The file will have its original line endings in your working directory.

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README.md

new file: mytext.txt

Administrator@MY-PC /F/worksp/git-start (master)

$

1. **状态简览**

git status 命令的输出十分详细，但其用语有些繁琐。 如果你使用 git status -s 命令或 git status --short 命令，将得到一种更为紧凑的格式输出。 运行 git status -s，状态报告输出如下：

$ git status -s

M README.md

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

新添加的未跟踪文件前面有 ?? 标记，新添加到暂存区中的文件前面有 A 标记，修改过的文件前面有 M 标记。 你可能注意到了 M 有两个可以出现的位置，出现在右边的 M 表示该文件被修改了但是还没放入暂存区，出现在靠左边的 M 表示该文件被修改了并放入了暂存区。 例如，上面的状态报告显示： README 文件在工作区被修改了但是还没有将修改后的文件放入暂存区,lib/simplegit.rb 文件被修改了并将修改后的文件放入了暂存区。 而 Rakefile 在工作区被修改并提交到暂存区后又在工作区中被修改了，所以在暂存区和工作区都有该文件被修改了的记录。

1. **忽略文件**

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。 通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。 在这种情况下，我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。 来看一个实际的例子：

$ cat .gitignore

\*.[oa]

\*~

第一行告诉 Git 忽略所有以 .o 或 .a 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的。 第二行告诉 Git 忽略所有以波浪符(~)结尾的文件，许多文本编辑软件(比如 Emacs)都用这样的文件名保存副本。 此外，你可能还需要忽略 log，tmp 或者 pid 目录，以及自动生成的文档等等。 要养成一开始就设置好 .gitignore 文件的习惯，以免将来误提交这类无用的文件。

文件 .gitignore 的格式规范如下：

所有空行或者以 ＃ 开头的行都会被 Git 忽略。

可以使用标准的 glob 模式匹配。

匹配模式可以以(/)开头防止递归。

匹配模式可以以(/)结尾指定目录。

要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号(!)取反。

所谓的 glob 模式是指 shell 所使用的简化了的正则表达式。 星号(\*)匹配零个或多个任意字符；[abc]匹配任何一个列在方括号中的字符(这个例子要么匹配一个字符 a，要么匹配一个字符 b，要么匹配一个字符 c)；问号(?)只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配(比如 [0-9] 表示匹配所有 0 到 9 的数字)。 使用两个星号(\*) 表示匹配任意中间目录，比如a/\*\*/z 可以匹配 a/z, a/b/z 或 a/b/c/z等。

下面再看一个 .gitignore 文件的例子：

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory

doc/\*\*/\*.pdf

1. **查看已暂存和未暂存的修改**

如果 git status 命令的输出对于你来说过于模糊，你想知道具体修改了什么地方，可以用 git diff 命令。 稍后我们会详细介绍 git diff，可能通常会用它来回答这两个问题：当前做的哪些更新还没有暂存？ 有哪些更新已经暂存起来准备好了下次提交？ 尽管 git status 已经通过在相应栏下列出文件名的方式回答了这个问题，git diff 将通过文件补丁的格式显示具体哪些行发生了改变。

假如再次修改 README.md 文件后暂存，然后编辑 READ.md 文件并在文件的最后追加一行内容：”this is another line 1003“ 之后先不暂存， 运行 git status 命令将会看到：

$ echo "this is another line 1003 " >> README.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README.md

new file: mytext.txt

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

Administrator@MY-PC /F/worksp/git-start (master)

$

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：

$ git diff

diff --git a/README.md b/README.md

index ea161e2..6679481 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,3 @@

Add new Line content 1001

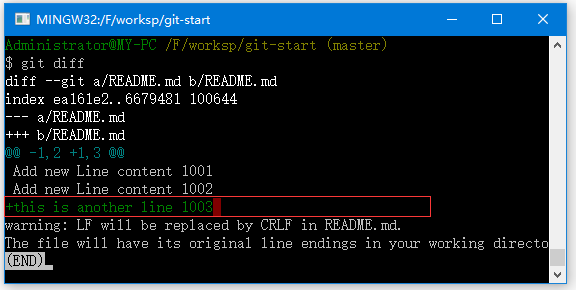
Add new Line content 1002

+this is another line 1003

warning: LF will be replaced by CRLF in README.md.

The file will have its original line endings in your working directo(END)

执行上面命令，得到以下输出结果 -



上面输出显示有加一行“+this is another line 1003”，前面带有一个加号：“+”。

请注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。 所以有时候你一下子暂存了所有更新过的文件后，运行 git diff 后却什么也没有，就是这个原因。

然后用 git diff --cached 查看已经暂存起来的变化：(--staged 和 --cached 是同义词)

$ git diff --cached

diff --git a/README.md b/README.md

index 2f88ca7..ea161e2 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,2 @@

-#git-start

-这是一个 Git 学习使用的Git仓库。

\ No newline at end of file

+Add new Line content 1001

+Add new Line content 1002

diff --git a/mytext.txt b/mytext.txt

new file mode 100644

index 0000000..1820ae1

--- /dev/null

+++ b/mytext.txt

@@ -0,0 +1 @@

+This is my first Git control file

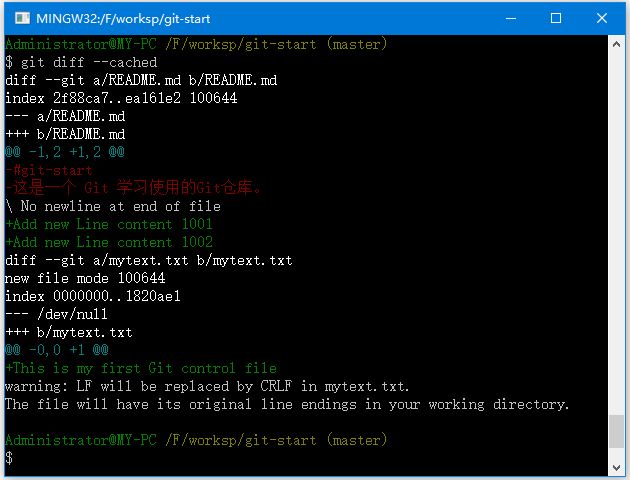
warning: LF will be replaced by CRLF in mytext.txt.

The file will have its original line endings in your working directory.

Administrator@MY-PC /F/worksp/git-start (master)

$

执行上面命令，得到如下输出结果



如上图中所示，分别对比了两个文件：README.md 和 mytext.txt，其中绿色的内容表示添加，红色的内容表示删除。

1. **提交更新**

现在的暂存区域已经准备妥当可以提交了。 在此之前，请一定要确认还有什么修改过的或新建的文件还没有 git add 过，否则提交的时候不会记录这些还没暂存起来的变化。 这些修改过的文件只保留在本地磁盘。 所以，每次准备提交前，先用 git status 看下，是不是都已暂存起来了，如果没有暂存起来则要先使用命令：git add .将所有文件暂存起来， 然后再运行提交命令 git commit

$ git status

$ git add .

$ git commit

这种方式会启动文本编辑器以便输入本次提交的说明。 (默认会启用 shell 的环境变量 $EDITOR 所指定的软件，一般都是 vim 或 emacs。使用 git config --global core.editor 命令设定你喜欢的编辑软件。)

编辑器会显示类似下面的文本信息(本例选用 Vim 的屏显方式展示)

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# new file: README

# modified: CONTRIBUTING.md

#

this is my commit info note.

~

~

".git/COMMIT\_EDITMSG" 9L, 283C

可以看到，默认的提交消息包含最后一次运行 git status 的输出，放在注释行里，另外开头还有一空行，供你输入提交说明。完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。 (如果想要更详细的对修改了哪些内容的提示，可以用 -v 选项，这会将你所做的改变的 diff 输出放到编辑器中从而使你知道本次提交具体做了哪些修改。) 退出编辑器时，Git 会丢掉注释行，用输入提交附带信息生成一次提交。如上面示例中，提交的备注信息是：“this is my commit info note.”。

另外，也可以在 commit 命令后添加 -m 选项，将提交信息与命令放在同一行，如下所示：

$ git commit -m "this is my commit info note."

[master 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README.md

现在已经创建了第一个提交！ 可以看到，提交后它会告诉你，当前是在哪个分支(master)提交的，本次提交的完整 SHA-1 校验和是什么(463dc4f)，以及在本次提交中，有多少文件修订过，多少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。 每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

1. **跳过使用暂存区域**

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。 Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤：

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

看到了吗？提交之前不再需要 git add 文件“README.md”了。

1. **移除文件**

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除(确切地说，是从暂存区域移除)，然后提交。 可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分(也就是 未暂存清单)看到：

$ rm mytext.txt

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: mytext.txt

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /F/worksp/git-start (master)

$

下一次提交时，该文件就不再纳入版本管理了。 如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f(注：即 force 的首字母)。 这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除(亦即从暂存区域移除)，但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 --cached 选项：

$ git rm --cached mytext.txt

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: mytext.txt

git rm 命令后面可以列出文件或者目录的名字，也可以使用 glob 模式。 比方说：

$ git rm log/\\*.log

注意到星号 \* 之前的反斜杠 \， 因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。 此命令删除 log/ 目录下扩展名为 .log 的所有文件。 类似的比如：

$ git rm \\*~

该命令为删除以 ~ 结尾的所有文件。

1. **移动文件**

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。 如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。 不过 Git 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 Git 的 mv 命令时一定会困惑不已。 要在 Git 中对文件改名，可以这么做：

$ git mv file\_from file\_to

它会恰如预期般正常工作。 实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说明：

$ git mv README.md README

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

其实，运行 git mv 就相当于运行了下面三条命令：

$ mv README.md README

$ git rm README.md

$ git add README

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。 不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

### 查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。 完成这个任务最简单而又有效的工具是 git log 命令。

接下来的例子会用我专门用于演示的 simplegit 项目， 运行下面的命令获取该项目源代码：

$ git clone http://github.com/yiibai/simplegit-progit

然后在此项目中运行 git log，应该会看到下面的输出：

$ git log

commit 0e72e2c0ab0c5bfbe34603e5fcca91a0b5c381ff

Author: your\_name <your\_email@mail.com>

Date: Thu Jul 6 23:49:46 2017 +0800

this is my comment

commit 85090b865d5cd7213e41a948e9f6f7466a950dbe

Author: Maxsu <769728683@qq.com>

Date: Thu Jul 6 17:34:41 2017 +0800

Initial commit

Administrator@MY-PC /F/worksp/git-start (master)

$

默认不用任何参数的话，git log 会按提交时间列出所有的更新，最近的更新排在最上面。 正如你所看到的，这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

git log 有许多选项可以帮助你搜寻你所要找的提交， 接下来我们介绍些最常用的。

一个常用的选项是 -p，用来显示每次提交的内容差异。 你也可以加上 -2 来仅显示最近两次提交：

$ git log -p -2

commit 0e72e2c0ab0c5bfbe34603e5fcca91a0b5c381ff

Author: your\_name <your\_email@mail.com>

Date: Thu Jul 6 23:49:46 2017 +0800

this is my comment

diff --git a/README.md b/README.md

index 2f88ca7..6679481 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,3 @@

commit 0e72e2c0ab0c5bfbe34603e5fcca91a0b5c381ff

Author: your\_name <your\_email@mail.com>

Date: Thu Jul 6 23:49:46 2017 +0800

this is my comment

diff --git a/README.md b/README.md

index 2f88ca7..6679481 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,3 @@

该选项除了显示基本信息之外，还附带了每次 commit 的变化。 当进行代码审查，或者快速浏览某个搭档提交的 commit 所带来的变化的时候，这个参数就非常有用了。 你也可以为 git log 附带一系列的总结性选项。 比如说，如果你想看到每次提交的简略的统计信息，可以使用 --stat 选项：

$ git log --stat

commit 0e72e2c0ab0c5bfbe34603e5fcca91a0b5c381ff

Author: your\_name <your\_email@mail.com>

Date: Thu Jul 6 23:49:46 2017 +0800

this is my comment

README.md | 5 +++--

mytext.txt | 1 +

2 files changed, 4 insertions(+), 2 deletions(-)

commit 85090b865d5cd7213e41a948e9f6f7466a950dbe

Author: Maxsu <769728683@qq.com>

Date: Thu Jul 6 17:34:41 2017 +0800

Initial commit

README.md | 2 ++

1 file changed, 2 insertions(+)

Administrator@MY-PC /F/worksp/git-start (master)

$

正如你所看到的，--stat 选项在每次提交的下面列出额所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。 在每次提交的最后还有一个总结。

另外一个常用的选项是 --pretty。 这个选项可以指定使用不同于默认格式的方式展示提交历史。 这个选项有一些内建的子选项供你使用。 比如用 oneline 将每个提交放在一行显示，查看的提交数很大时非常有用。 另外还有 short，full 和 fuller 可以用，展示的信息或多或少有些不同，请自己动手实践一下看看效果如何。

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

但最有意思的是 format，可以定制要显示的记录格式。 这样的输出对后期提取分析格外有用 — 因为你知道输出的格式不会随着 Git 的更新而发生改变：

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecessary test

a11bef0 - Scott Chacon, 6 years ago : first commit

git log --pretty=format 常用的选项 列出了常用的格式占位符写法及其代表的意义。

你一定感到奇怪 作者 和 提交者 之间究竟有何差别， 其实作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人。 所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者。 我们会在 分布式 Git 再详细介绍两者之间的细微差别。

当 oneline 或 format 与另一个 log 选项 --graph 结合使用时尤其有用。 这个选项添加了一些ASCII字符串来形象地展示你的分支、合并历史：

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

### 撤消操作

在任何一个阶段，你都有可能想要撤消某些操作。 这里，我们将会学习几个撤消你所做修改的基本工具。 注意，有些撤消操作是不可逆的。 这是在使用 Git 的过程中，会因为操作失误而导致之前的工作丢失的少有的几个地方之一。

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令尝试重新提交：

$ git commit --amend

这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改(例如，在上次提交后马上执行了此命令)，那么快照会保持不变，而你所修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。 编辑后保存会覆盖原来的提交信息。

例如，提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。

1. **取消暂存的文件**

接下来的两个小节演示如何操作暂存区域与工作目录中已修改的文件。 这些命令在修改文件状态的同时，也会提示如何撤消操作。 例如，你已经修改了两个文件并且想要将它们作为两次独立的修改提交，但是却意外地输入了 git add \* 暂存了它们两个。 如何只取消暂存两个中的一个呢？ git status 命令提示：

$ git add \*

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

deleted: mytext.txt

在 “Changes to be committed” 文字正下方，提示使用 git reset HEAD <file>... 来取消暂存。 所以，我们可以这样来取消暂存 mytext.txt 文件：

$ git reset HEAD mytext.txt

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: mytext.txt

1. **撤消对文件的修改**

如果并不想保留对 mytext.txt 文件的修改怎么办？ 该如何方便地撤消修改 - 将它还原成上次提交时的样子(或者刚克隆完的样子，或者刚把它放入工作目录时的样子)？ 幸运的是，git status 也告诉了你应该如何做。 在最后一个例子中，未暂存区域是这样：

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: mytext.txt

它非常清楚地告诉了如何撤消之前所做的修改。让我们来按照提示执行：

$ git checkout -- mytext.txt

Administrator@MY-PC /F/worksp/git-start (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Administrator@MY-PC /F/worksp/git-start (master)

$ ls

README mytext.txt

可以看到，mytext.txt文件又回来了。

如果仍然想保留对那个文件做出的修改，但是现在仍然需要撤消，我们将会在 Git 分支介绍保存进度与分支；这些通常是更好的做法。

记住，在 Git 中任何已提交的东西几乎总是可以恢复的。甚至那些被删除的分支中的提交或使用 --amend 选项覆盖的提交也可以恢复。然而，任何你未提交的东西丢失后很可能再也找不到了。

### 远程仓库的使用

前面所有讲解的内容都是一个人“自娱自乐”， Git这东西自己玩也没有多大意思，没有发挥出来Git最牛逼的地方。要使用Git在项目上多人协作那才有意思。

为了能在任意 Git 项目上协作，需要知道如何管理自己的远程仓库。远程仓库是指托管在因特网或其他网络中的你的项目的版本库。可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。 与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。 管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。 在本节中，我们将介绍一部分远程管理的技能。

1. **查看远程仓库**

如果想查看你已经配置的远程仓库服务器，可以运行 git remote 命令。 它会列出你指定的每一个远程服务器的简写。 如果已经克隆了自己的仓库，那么至少应该能看到 origin - 这是 Git 给你克隆的仓库服务器的默认名字：

$ git clone http://git.oschina.net/yiibai/git-start.git

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 157 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 74.35 KiB | 168.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd git-start

$ git remote

origin

也可以指定选项 -v，会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。

$ git remote -v

origin http://git.oschina.net/yiibai/git-start.git (fetch)

origin http://git.oschina.net/yiibai/git-start.git (push)

如果远程仓库不止一个，该命令会将它们全部列出。 例如，与几个协作者合作的，拥有多个远程仓库的仓库看起来像下面这样：

$ cd git-start

$ git remote -v

mydoor http://git.oschina.net/yiibai/git-start.git (fetch)

mydoor http://git.oschina.net/yiibai/git-start.git (push)

curry http://git.oschina.net/yiibai/git-start.git (fetch)

curry http://git.oschina.net/yiibai/git-start.git (push)

deepfun http://git.oschina.net/yiibai/git-start.git (fetch)

deepfun http://git.oschina.net/yiibai/git-start.git (push)

koke http://git.oschina.net/yiibai/git-start.git (fetch)

koke http://git.oschina.net/yiibai/git-start.git (push)

这样可以轻松拉取其中任何一个用户的贡献。 此外，大概还会有某些远程仓库的推送权限，虽然目前还不会在此介绍。

1. **添加远程仓库**

我在之前的章节中已经提到并展示了如何添加远程仓库的示例，不过这里将演示如何明确地做到这一点。 运行 git remote add <shortname> <url> 添加一个新的远程 Git 仓库，同时指定一个可以轻松引用的简写：

$ git remote

origin

$ git remote add gs http://git.oschina.net/yiibai/git-start.git

$ git remote -v

gs http://git.oschina.net/yiibai/git-start.git (fetch)

gs http://git.oschina.net/yiibai/git-start.git (push)

origin http://git.oschina.net/yiibai/git-start.git (fetch)

origin http://git.oschina.net/yiibai/git-start.git (push)

现在你可以在命令行中使用字符串 gs 来代替整个 URL。 例如，如果想拉取仓库中有但你没有的信息，可以运行git fetch gs

$ git fetch gs

From http://git.oschina.net/yiibai/git-start

\* [new branch] master -> gs/master

现在 master 分支可以在本地通过 gs/master 访问到 - 可以将它合并到自己的某个分支中，或者如果你想要查看它的话，可以检出一个指向该点的本地分支。

1. **从远程仓库中抓取与拉取**

就如刚才所见，从远程仓库中获得数据，可以执行：

$ git fetch [remote-name]

这个命令会访问远程仓库，从中拉取所有还没有的数据。执行完成后，将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

如果使用 clone 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。 所以，git fetch origin 会抓取克隆(或上一次抓取)后新推送的所有工作。 必须注意 git fetch 命令会将数据拉取到本地仓库 - 它并不会自动合并或修改当前的工作。 当准备好时必须手动将其合并入你的工作区。

如果你有一个分支设置为跟踪一个远程分支，可以使用 git pull 命令来自动的抓取然后合并远程分支到当前分支。 这对你来说可能是一个更简单或更舒服的工作流程；默认情况下，git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支(或不管是什么名字的默认分支)。 运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

1. **推送到远程仓库**

当想分享你的项目时，必须将其推送到上游。 这个命令很简单：git push [remote-name] [branch-name]。 当你想要将 master 分支推送到 origin 服务器时(再次说明，克隆时通常会自动帮你设置好那两个名字)，那么运行这个命令就可以将所做的备份到服务器：

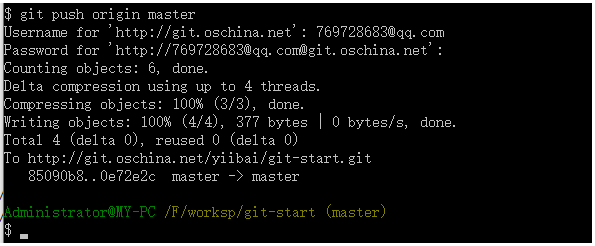
$ git push origin master

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。 当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。 你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。

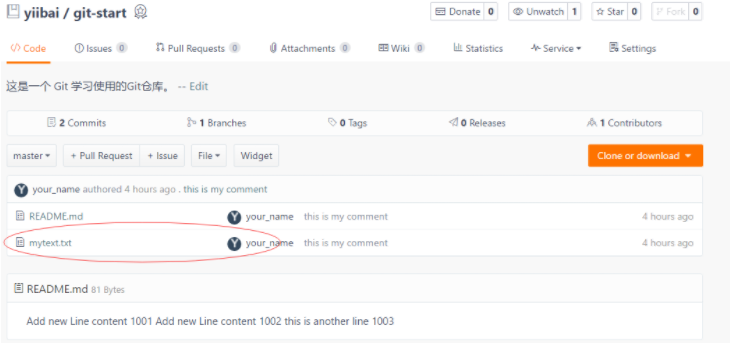
现在我们要把前所有添加和修改的内容添加到远程仓库，以便其协同的开发人员也可以获取到我们提交的内容。执行以下命令时，会要求我们输入在 http://git.oschina.net/ 注册的用户名和密码。

$ git push origin master

执行过程如下图中所示



现在登录 http://git.oschina.net/ 查看提交是成功(与本地电脑上的内容是否一致)，如下所示



此时文件：http://git.oschina.net/yiibai/git-start/blob/master/mytext.txt 的内容应该与 F:\worksp\git-start\mytext.txt 的内容完全一样。

1. **查看远程仓库**

如果想要查看某一个远程仓库的更多信息，可以使用 git remote show [remote-name] 命令。 如果想以一个特定的缩写名运行这个命令，例如 origin，会得到像下面类似的信息：

$ git remote show origin

\* remote origin

Fetch URL: http://git.oschina.net/yiibai/git-start.git

Push URL: http://git.oschina.net/yiibai/git-start.git

HEAD branch: master

Remote branch:

master tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (fast-forwardable)

它同样会列出远程仓库的 URL 与跟踪分支的信息。 这些信息非常有用，它告诉你正处于 master 分支，并且如果运行 git pull，就会抓取所有的远程引用，然后将远程 master 分支合并到本地 master 分支。 它也会列出拉取到的所有远程引用。

这是一个经常遇到的简单例子。 如果你是 Git 的重度使用者，那么还可以通过 git remote show 看到更多的信息。

$ git remote show origin

\* remote origin

Fetch URL: http://git.oschina.net/yiibai/git-start.git

Push URL: http://git.oschina.net/yiibai/git-start.git

HEAD branch: master

Remote branch:

master tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (fast-forwardable)

这个命令列出了当你在特定的分支上执行 git push 会自动地推送到哪一个远程分支。 它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当你执行 git pull 时哪些分支会自动合并

1. **远程仓库的移除与重命名**

如果想要重命名引用的名字可以运行 git remote rename 去修改一个远程仓库的简写名。 例如，想要将 gs 重命名为 newgs，可以用 git remote rename 这样做：

$ git remote rename gs newgs

$ git remote

origin

newgs

值得注意的是这同样也会修改你的远程分支名字。 那些过去引用 gs/master 的现在会引用 newgs/master。

如果因为一些原因想要移除一个远程仓库 - 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了 - 可以使用 git remote rm ：

$ git remote rm newgs

$ git remote

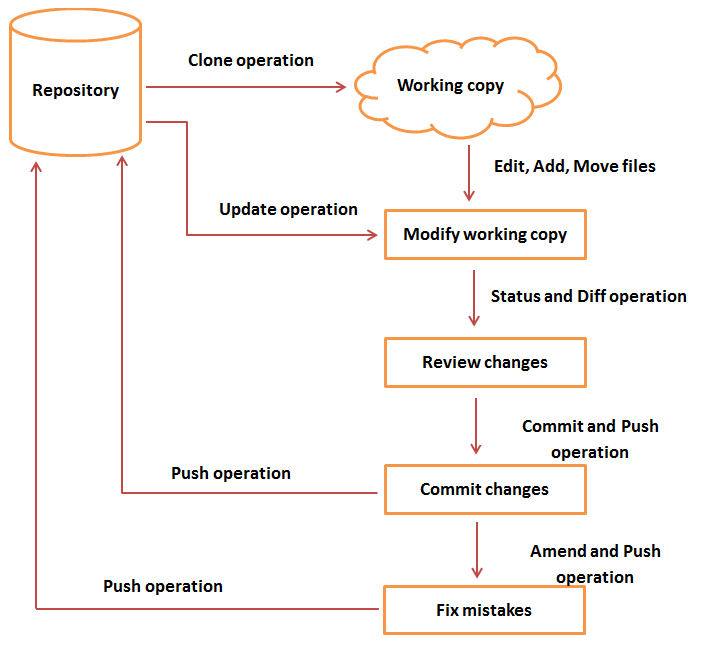
origin

## Git工作流程

Git的一般工作流程如下：

1. 将Git的一个存储库克隆为工作副本。
2. 可以通过添加/编辑文件修改工作副本。
3. 如有必要，还可以通过让其他开发人员一起来更改/更新工作副本。
4. 在提交之前查看更改。
5. 提交更改：如果一切正常，那么将您的更改推送到存储库。
6. 提交后，如果意识到某些错误并修改错误后，则将最后一个正确的修改提交并将推送到存储库。

下面显示的是工作流程的图示

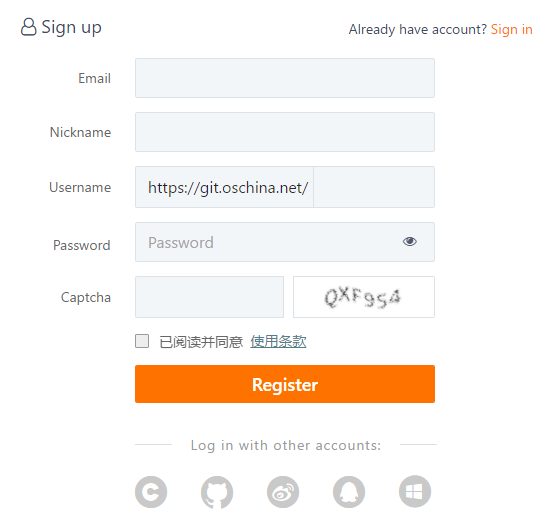


## Git创建存储库

在本章中，我们将看到如何创建一个远程Git仓库; 我们需要一个远程Git仓库来执行团队协作。我们将演示在 http://git.oschina.net/ 软件项目的托管平台上创建和初始化一个新的存储库。

**第一步：注册一个帐号**

打开网址：http://git.oschina.net/signup ， 写入用户名，邮件，密码等，注册一个帐号。如下所示

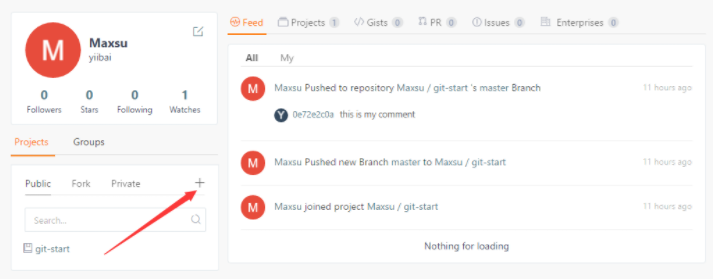


**第二步：验证账号邮箱**

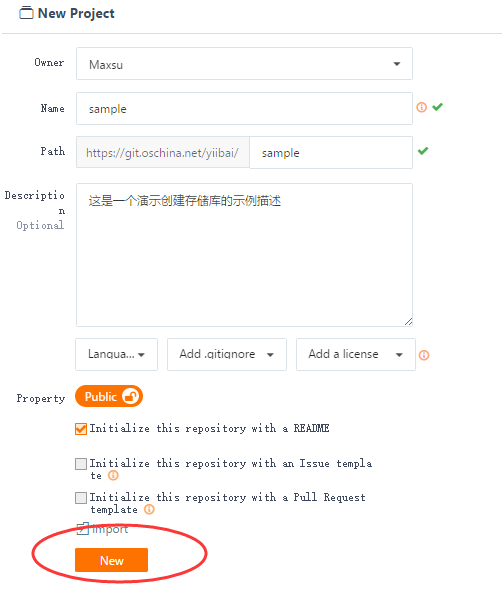
注册完成后，还不能马上使用，还需要登录注册的邮箱，验证完成后重新登录: http://git.oschina.net/login

**第三步：创建远程存储库**

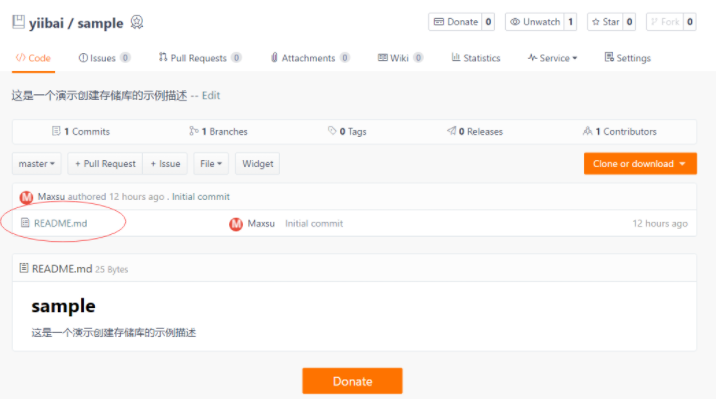
登录帐号成功后，在用户中心的左侧，找到“+”号的图标(下图中箭头指向)并点击，以创建一个存储库，如下图所示



在新界面中，填写一个必要的信息，如这里创建一个名称为：sample 的存储库，然后点击创建(New)，如下图所示



创建成功后，系统自动跳转到当前创建的存储库(project)下，在本示例中对应的URL是： http://git.oschina.net/yiibai/sample ，同时自动创建一个名称为：README的文件，写入的内容为上一步所写的描述：“这是一个演示创建存储库的示例描述”， 如下图所示



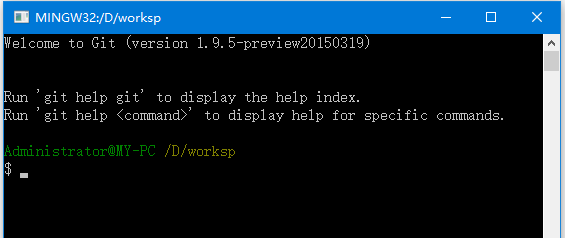
至此，一个新的存储库创建完成。在接下来的教程中，我们将基于这个存储库来演示各种 Git 命令的使用。

## Git克隆操作

进入一个即将用于存放存储库的目录，作为一个演示，这里使用的目录是：D:\worksp，在此目录中，点击右键，在弹出的菜单中选择：Git Bash，如下图所示



弹出一个 Git 的命令行工具界面，如下所示

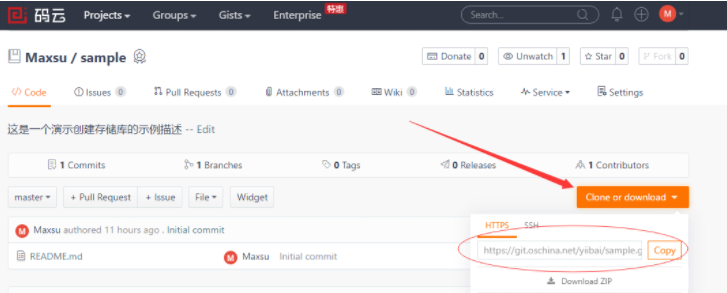


现在，使用以下命令来克隆远程存储库：sample，如下所示

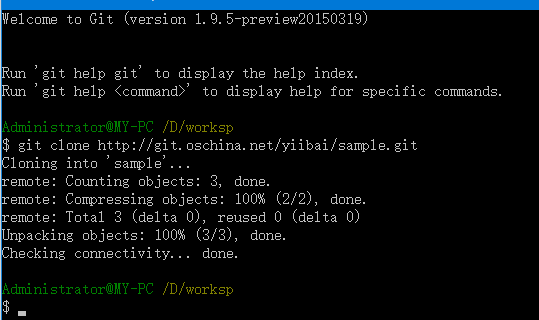
$ git clone http://git.oschina.net/yiibai/sample.git

注意：

上面的参数：http://git.oschina.net/yiibai/sample.git 从哪里来？ 您可直接访问： http://git.oschina.net/yiibai/sample ，找到 “Clone or download” 按钮并点击，在下拉的界面中选择 “http” 选项中的值复制就好。如下所示



执行上面的命令后，输出结果如下



现在整个远程存储库已经克隆完成，在目录 D:\worksp 中创建一个新的目录：sample，现在我们查看 sample 目录的文件内容，应该与远程存储库中的内容完全一样。

Administrator@MY-PC /D/worksp

$ cd sample/

$ ls

README.md

## Git执行变更操作

现在，在克隆存储库之后，我们开始学习 Git 基本的文件修改和版本管理操作。假设要使用 sample 这个存储库来协同管理一个Python的项目。首先创建一个Python的代码文件： main.py ，编写了一些代码完成并保存文件后，main.py 现在的内容如下所示：

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

假设上面代码编译并通过了测试，一切正常。 现在，我们可以安全地将这些更改添加到存储库。

查看当前工作区状态

$ git status -s

?? main.py

Git在文件名之前显示两个问号。因为到目前操作为止，这些文件还不是Git的一部分(Git还不能控制这些文件)，这就是Git不知道该怎么处理这些文件。 这就是为什么，Git在文件名之前显示问号。

现在，Git添加操作(git add命令)将文件添加到暂存区域。

$ git add main.py

在执行上命令后，已将文件添加到存储区域，git status命令将显示临时区域中存在的文件。

$ git status -s

A main.py

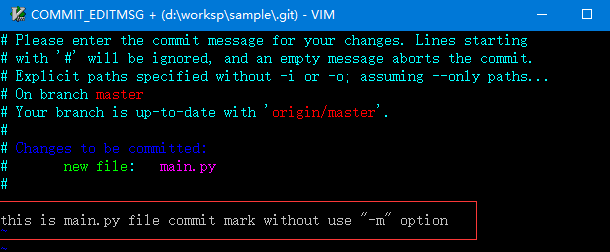
上面文件中，看到文件名称前面多了一个大写字母：A，表示该文件已添加到 Git 临时区域中了。

要提交更改，可使用了git commit命令，后跟-m选项。 如果忽略了-m选项。 Git将打开一个文本编辑器，我们可以在其中编写多行的提交备注消息。

在执行 git commit命令之前，一定要先执行 git add 命令。

$ git commit main.py

这里因为忽略了-m选项，所以会打开一个 vim 编辑器，可在引编辑器中写入提交说明备注信息，如下所示



如上图中，写了一条提交说明备注信息：“this is main.py file commit mark without use -m option”，退出 VIM 编辑器保存文件后， Git 会自动提交。当然，也可以直接使用以下命令一步完成。

git commit main.py -m "this is main.py file commit mark use -m option"

执行上面代码，结果如下

$ git commit main.py -m "this is main.py file commit mark use -m option"

[master 5eccf92] this is main.py file commit mark use -m option

1 file changed, 2 insertions(+), 1 deletion(-)

在提交查看日志详细信息后，运行git log命令。它将使用提交ID，提交作者，提交日期和提交的SHA-1哈希显示所有提交的信息。

$ git log

上述命令将产生以下结果：

$ git log

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

前面我们说过，要提交修改过的文件，首先使用 git add <file> ，然后再执行 git comit <file> -m "？？mark"，但是这两步也可以一步完成，如下所示

$ git commit -a -m "？？mark"

## Git查看更改

比如，我们查看提交详细信息后，需要修改代码，或添加更多的代码，或者对比提交结果。

下面使用git log命令查看日志详细信息。

$ git log

执行上面命令后，得到以下输出结果

$ git log

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

？？mark

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

Administrator@MY-PC /D/worksp/sample (master)

$

使用git show命令查看某一次提交详细信息。 git show命令采用SHA-1提交ID作为参数。

$ git show be24e214620fa072efa877e1967571731c465884

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

？？mark

diff --git a/main.py b/main.py

index 91a1389..657c8d0 100644

--- a/main.py

+++ b/main.py

@@ -3,3 +3,5 @@

print ("Life is short, you need Python !")

+# this is a comment line

+

上面显示的结果中，可以看到符号 “+“ ，表示添加的内容。如果有 “-”则表示删除的内容，现在我们打开 main.py ，把注释行去掉并定义一个变量。修改后的 main.py 的内容如下所示

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

a = 10

b = 20

然后使用命令：git stauts 查看当前工作区状态

$ git status

On branch master

Your branch is ahead of 'origin/master' by 3 commits.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

测试代码后，通过运行git diff命令来回顾他的更改

$ git diff

diff --git a/main.py b/main.py

index 95053b4..a4f953e 100644

--- a/main.py

+++ b/main.py

@@ -4,4 +4,6 @@

print ("Life is short, you need Python !")

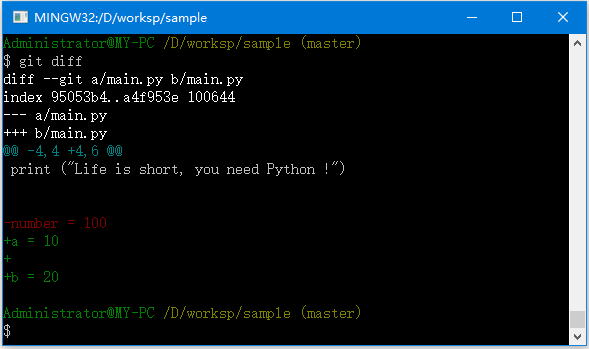
-number = 100

+a = 10

+

+b = 20

可以看到符号 “+“ (绿色)，表示添加的内容。如果有 “-”(红色)则表示删除的内容。执行的效果如下所示



现在使用以下命令将文件：main.py 添加到 git 暂存区，然后提交代码完成

$ git add main.py

$ git commit -m "define two var a & b "

最近修改的代码已提交完成。

## Git提交更改

在上一步中，我们已经修改了 main.py 文件中的代码，在代码中定义了两个变量并提交代码，但是要再次添加和修改main.py 文件中的代码，实现新功能：求两个变量相加值。修改提交的操作更改包含提交消息的最后一个提交; 它创建一个新的提交ID。

在修改操作之前，检查提交日志，如下命令所示

$ git log

commit d757c8e92ad6053db294100c77075865f829b7ac

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

define two var a & b

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

？？mark

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

下面我们打开文件：main.py 加入以下两行

c = a + b

print("The value of c is ", c)

更正操作提交新的更改，并查看提交日志。首先查看状态，如下命令

$ git status

On branch master

Your branch is ahead of 'origin/master' by 4 commits.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

添加文件并查看状态，如下命令

$ git add main.py

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 4 commits.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: main.py

提交更改的文件，如下所示

$ git commit --amend -m "add the sum of a & b "

[master 51de0f0] add the sum of a & b

1 file changed, 5 insertions(+), 1 deletion(-)

现在，使用 git log命令显示将显示新的提交消息与新的提交ID(51de0f02eb48ed6b84a732512f230028d866b1ea)，最近一次提交的放前面：

$ git log

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

？？mark

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

## Git推送（push）操作

在前面的文章中，都在要本地编写文件代码和提交，维护管制自己的文件版本，然后这种“自娱自乐”的方式，意义不是很大，在这里将介绍如何与其它的开发人员协同开发工作：每个开发人员都可以提交自己贡献的代码，并让其他人看到和修改。

要协同多人一起工作，可通过修改操作将代码文件最后一个确定版本提交，然后再推送变更。 推送(Push)操作将数据永久存储到Git仓库。成功的推动操作后，其他开发人员可以看到新提交的变化。

执行git log命令查看提交的详细信息。最后一次提交的代码的提交ID是：51de0f02eb48ed6b84a732512f230028d866b1ea，如下所示

$ git log

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

？？mark

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

在推送(push)操作之前，如想要检查文件代码变化，可使用git show命令指定提交ID来查看具体的变化。

$ git show 51de0f02eb48ed6b84a732512f230028d866b1ea

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

diff --git a/main.py b/main.py

index 657c8d0..25eb22b 100644

--- a/main.py

+++ b/main.py

@@ -3,5 +3,9 @@

print ("Life is short, you need Python !")

-# this is a comment line

+a = 10

+

+b = 20

+c = a + b

+print("The value of c is ", c)

\ No newline at end of file

注意：每一行代码前面的 -号和+号。-号表示删除，+号表示添加。如下

-# this is a comment line

+a = 10

+

+b = 20

+c = a + b

+print("The value of c is ", c)

如果对上面的提交修改没有疑义，则我们就可以将文件代码推送到远程存储库中，从而让其它开发人员可看查看和修改这些代码，现在就来看看怎么提交这些写好的代码，使用以下命令

$ git push origin master

上述命令将产生以下结果

$ git push origin master

Username for 'http://git.oschina.net': 76972883@qq.com <输入帐号>

Password for 'http://76972883@qq.com@git.oschina.net': <输入登录密码>

Counting objects: 13, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (12/12), 1.20 KiB | 0 bytes/s, done.

Total 12 (delta 3), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

290342c..51de0f0 master -> master

在上面命令中，需要您提提供( http://git.oschina.net )用户名和密码。

如上所示，现在代码已经成功地提交到了远程存储库( http://git.oschina.net )中了。要验证提交的结果，远程存储库中的内容是否是最后一次提交的信息，我们可以在另外一个空的目录中或在另外一台机器上使用 git clone 克隆出完整的文件代码，例如，在目录：E:\workspace 下执行以下命令

$ git clone http://git.oschina.net/yiibai/sample.git

Cloning into 'sample'...

remote: Counting objects: 15, done.

remote: Compressing objects: 100% (14/14), done.

remote: Total 15 (delta 3), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

Checking connectivity... done.

在执行上面命令后，打开文件： E:\workspace\sample\main.py ，其代码内容如下

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

a = 10

b = 20

c = a + b

print("The value of c is ", c)

可以看到此文件与最后一个版本的内容一样。

## Git更新操作

执行克隆操作，并得到了一个新的文件：main.py。想知道谁将这个文件修改变提交到存储库中，那么可以执行git log命令，为了更好的演示，开发人员minsu另外一台机(Ubuntu Linux)上，使用以下命令克隆

yiibai@ubuntu:~$ cd /home/yiibai/git

yiibai@ubuntu:~/git$ git clone http://git.oschina.net/yiibai/sample.git

Cloning into 'sample'...

remote: Counting objects: 15, done.

remote: Compressing objects: 100% (14/14), done.

remote: Total 15 (delta 3), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

Checking connectivity... done.

yiibai@ubuntu:~/git$

克隆操作将在当前工作目录中创建一个新的目录(sample)。 将目录更改(cd /home/yiibai/git/sample)为新创建的目录，并执行git log命令。

yiibai@ubuntu:~/git$ cd /home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ git log

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

commit be24e214620fa072efa877e1967571731c465884

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:58:16 2017 +0800

ï¼Ÿï¼Ÿmark

commit 5eccf92e28eae94ec5fce7c687f6f92bf32a6a8d

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:52:06 2017 +0800

this is main.py file commit mark use -m option

commit 6e5f31067466795c522b01692871f202c26ff948

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 18:42:43 2017 +0800

this is main.py file commit mark without use "-m" option

commit 290342c270bc90f861ccc3d83afa920169e3b07e

Author: Maxsu <769728683@qq.com>

Date: Fri Jul 7 16:55:12 2017 +0800

Initial commit

yiibai@ubuntu:~/git/sample$

观察日志后，可以看到maxsu添加了文件代码来实现两个变量a和b之和，现在假设minsu在文本编辑器中打开main.py，并修改优化代码。在示两个变量a和b之和，定义一个函数来实现两个变量之和。修改后，代码如下所示：

#!/usr/bin/python3

#coding=utf-8

a = 10

b = 20

def sum(a, b):

return (a+b)

c = sum(a, b)

print("The value of c is ", c)

使用 git diff 命令查看更改，如下所示

yiibai@ubuntu:~/git/sample$ git diff

diff --git a/main.py b/main.py

index 25eb22b..e84460d 100644

--- a/main.py

+++ b/main.py

@@ -7,5 +7,9 @@ print ("Life is short, you need Python !")

a = 10

b = 20

-c = a + b

-print("The value of c is ", c)

\ No newline at end of file

+

+def sum(a, b):

+ return (a+b)

+

+c = sum(a, b)

+print("The value of c is ", c)

yiibai@ubuntu:~/git/sample$

经过测试，代码没有问题，提交了上面的更改。

yiibai@ubuntu:~/git/sample$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

yiibai@ubuntu:~/git/sample$ git add main.py

yiibai@ubuntu:~/git/sample$ git commit -m "add a new function sum(a,b)"

[master 01c5462] add a new function sum(a,b)

1 file changed, 6 insertions(+), 2 deletions(-)

yiibai@ubuntu:~/git/sample$

再次查看提交记录信息

yiibai@ubuntu:~/git/sample$ git log

commit 01c54624879782e4657dd6c166ce8818f19e8251

Author: minsu <minsu@yiibai.com>

Date: Sun Jul 9 19:01:00 2017 -0700

add a new function sum(a,b)

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

....

经过上面添加和提交修改后，其它开发人员并不能看到代码中定义的 sum(a, b) 函数，还需要将这里提交的本地代码推送到远程存储库。使用以下命令

yiibai@ubuntu:~/git/sample$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':<你的帐号的密码>

Counting objects: 3, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 419 bytes | 0 bytes/s, done.

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

To http://git.oschina.net/yiibai/sample.git

51de0f0..01c5462 master -> master

yiibai@ubuntu:~/git/sample$

现在，代码已经成功提交到远程存储库中了，只要其它开发人员使用 git clone 或 git pull 就可以得到这些新提交的代码了。

下面我们将学习其它一些更为常用的 git 命令，经过下面的学习，你就可以使用这些基本的git操作命令来文件和代码管理和协同开发了。

* **添加新函数**

在开发人员B(minsu)修改文件main.py中的代码的同时，开发人员A(maxsu)在文件main.py中实现两个变量的乘积函数。 修改后，main.py文件如下所示：

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

a = 10

b = 20

c = a + b

print("The value of c is ", c)

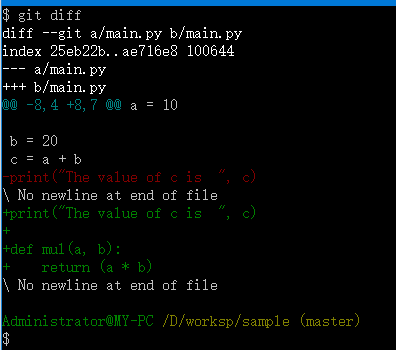
def mul(a, b):

return (a \* b)

现在修改完代码，运行以下命令

$ git diff

得到如下结果



现在添加并提交上面的代码

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

$ git add main.py

Administrator@MY-PC /D/worksp/sample (master)

$ git commit -m "add a mul(a, b) function"

[master e5f8dfa] add a mul(a, b) function

1 file changed, 4 insertions(+), 1 deletion(-)

现在查看上提交的日志信息，如下结果

$ git log

commit e5f8dfa9e7e89fea8813ab107e14b9b7412df2ae

Author: your\_name <your\_email@mail.com>

Date: Sun Jul 9 23:06:32 2017 +0800

add a mul(a, b) function

commit 51de0f02eb48ed6b84a732512f230028d866b1ea

Author: your\_name <your\_email@mail.com>

Date: Fri Jul 7 23:04:16 2017 +0800

add the sum of a & b

... ...

好了，现在要将上面的代码推送到远程存储库，使用以下命令

$ git push origin master

执行上面命令，结果如下

$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

To http://git.oschina.net/yiibai/sample.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'http://git.oschina.net/yiibai/sample.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

但Git推送失败。因为Git确定远程存储库和本地存储库不同步。为什么呢？因为在向文件main.py添加以下代码片段时

def mul(a, b):

return (a \* b)

另外一个开发人员B已经向远程存储库推送修改的代码，所以这里在向远程存储库推送代码时，发现上面的新的推送代码，现在这个要推送的代码与远程存储库中的代码不一致，如果强行推送上去，Git不知道应该以谁的为准了。

所以，必须先更新本地存储库，只有在经过此步骤之后，才能推送自己的改变。

* **获取最新更改**

执行git pull命令以将其本地存储库与远程存储库同步。

$ git pull

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From http://git.oschina.net/yiibai/sample

51de0f0..01c5462 master -> origin/master

Auto-merging main.py

CONFLICT (content): Merge conflict in main.py

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

现在打开 main.py 内容如下

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

a = 10

b = 20

<<<<<<< HEAD

c = a + b

print("The value of c is ", c)

def mul(a, b):

return (a \* b)

=======

def sum(a, b):

return (a+b)

c = sum(a, b)

print("The value of c is ", c)

>>>>>>> 01c54624879782e4657dd6c166ce8818f19e8251

拉取操作后，检查日志消息，并发现其它开发人员的提交的详细信息，提交ID为：01c54624879782e4657dd6c166ce8818f19e8251

打开 main.py 文件，修改其中的代码，修改完成后保文件，文件的代码如下所示

#!/usr/bin/python3

#coding=utf-8

print ("Life is short, you need Python !")

a = 10

b = 20

c = a + b

print("The value of c is ", c)

def mul(a, b):

return (a \* b)

def sum(a, b):

return (a+b)

c = sum(a, b)

print("The value of c is ", c)

再次添加提交，最后推送，如下命令

$ git add main.py

Administrator@MY-PC /D/worksp/sample (master|MERGING)

$ git commit -m "synchronized with the remote repository "

[master ef07ab5] synchronized with the remote repository

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(use "git push" to publish your local commits)

nothing to commit, working directory clean

Administrator@MY-PC /D/worksp/sample (master)

$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 10, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 657 bytes | 0 bytes/s, done.

Total 6 (delta 2), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

01c5462..ef07ab5 master -> master

现在，一个新的代码又提交并推送到远程存储库中了。

## Git隐藏(Stash)操作

假设您正在为产品新的功能编写/实现代码，当正在编写代码时，突然出现软件客户端升级。这时，您必须将新编写的功能代码保留几个小时然后去处理升级的问题。在这段时间内不能提交代码，也不能丢弃您的代码更改。 所以需要一些临时等待一段时间，您可以存储部分更改，然后再提交它。

在Git中，隐藏操作将使您能够修改跟踪文件，阶段更改，并将其保存在一系列未完成的更改中，并可以随时重新应用。

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

现在，要切换分支以进行客户升级，但不想提交一直在做的工作; 那么可以把当前工作的改变隐藏起来。 要将一个新的存根推到堆栈上，运行git stash命令。

$ git stash

Saved working directory and index state WIP on master: ef07ab5 synchronized with the remote repository

HEAD is now at ef07ab5 synchronized with the remote repository

现在，工作目录是干净的，所有更改都保存在堆栈中。 现在使用git status命令来查看当前工作区状态。

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

现在，可以安全地切换分支并在其他地方工作。通过使用git stash list命令来查看已存在更改的列表。

$ git stash list

stash@{0}: WIP on master: ef07ab5 synchronized with the remote repository

假设您已经解决了客户升级问题，想要重新开始新的功能的代码编写，查找上次没有写完成的代码，只需执行git stash pop命令即可从堆栈中删除更改并将其放置在当前工作目录中。

$ git status -s

Administrator@MY-PC /D/worksp/sample (master)

[jerry@CentOS project]$ git stash pop

上述命令将产生以下结果：

$ git stash pop

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (e713780380632c142ed5833a9087aca883a826fa)

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

可以看到，工作区中修改的文件(main.py)又显示了。现在我们就可以继续编写上次编写了未完成的代码。

## Git移动操作

移动操作将目录或文件从一个位置移动到另一个位置。例如，我们想要将源代码移动到src目录中。修改后的目录结构将显示如下：

Administrator@MY-PC /D/worksp/sample (master)

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ ls

README.md main.py

Administrator@MY-PC /D/worksp/sample (master)

$ mkdir src

Administrator@MY-PC /D/worksp/sample (master)

$ git mv main.py src/

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: main.py -> src/main.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: src/main.py

为了使这些更改永久性，必须将修改的目录结构推送到远程存储库，以便其他开发人员可以看到这些更改。

$ git add .

Administrator@MY-PC /D/worksp/sample (master)

$ git commit -m "Modified directory structure"

[master 186df84] Modified directory structure

1 file changed, 3 insertions(+)

rename main.py => src/main.py (78%)

Administrator@MY-PC /D/worksp/sample (master)

$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (4/4), 491 bytes | 0 bytes/s, done.

Total 4 (delta 0), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

ef07ab5..186df84 master -> master

在其它开发人员的本地存储库中，在执行git pull操作之前，它将显示旧的目录结构。在另外一台开发者机器上，执行以下命令

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ ls

main.py master README.md

yiibai@ubuntu:~/git/sample$

但是在执行git pull操作之后，目录结构将被更新。 现在，假设在另外一个开发人员(minsu)执行git pull操作之后,就可以看到目录中的src目录和文件了。

yiibai@ubuntu:~/git/sample$ git pull

remote: Counting objects: 10, done.

remote: Compressing objects: 100% (9/9), done.

remote: Total 10 (delta 2), reused 0 (delta 0)

Unpacking objects: 100% (10/10), done.

From http://git.oschina.net/yiibai/sample

01c5462..186df84 master -> origin/master

Updating 01c5462..186df84

Fast-forward

main.py => src/main.py | 11 +++++++++++

1 file changed, 11 insertions(+)

rename main.py => src/main.py (58%)

yiibai@ubuntu:~/git/sample$ ls

master README.md src

yiibai@ubuntu:~/git/sample$ ls src/

main.py

yiibai@ubuntu:~/git/sample$

## Git重命名操作

到目前为止，我们前面已经存建了一个 Python 的源代码文件，现在，要修改 main.py 文件的名称把它作为一个新的模块，假设这里要文件main.py的新名称为：module.py。

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ cd src/

Administrator@MY-PC /D/worksp/sample/src (master)

$ pwd

/D/worksp/sample/src

Administrator@MY-PC /D/worksp/sample/src (master)

$ ls

main.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ git mv main.py module.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ git status -s

R main.py -> module.py

Git在文件名之前显示R，表示文件已被重命名。

对于提交操作，需要使用-a标志，这使git commit自动检测修改的文件。

Administrator@MY-PC /D/worksp/sample/src (master)

$ git commit -a -m 'renamed main.py to module.py'

[master 6bdbf82] renamed main.py to module.py

1 file changed, 0 insertions(+), 0 deletions(-)

rename src/{main.py => module.py} (100%)

提交后，新文件将更改推送到远程存储库。

$ git push origin master

上述命令将产生以下结果：

$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.

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

To http://git.oschina.net/yiibai/sample.git

186df84..6bdbf82 master -> master

现在，其他开发人员可以通过使用git pull命令更新本地存储库来查看这些修改。

## Git删除操作

其他开发人员在更新他的本地存储库后，在src目录中找到一个module.py文件。查看提交消息后，了解到module.py文件是由maxsu添加的。

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ ls

README.md src

yiibai@ubuntu:~/git/sample$ ls src/

main.py

yiibai@ubuntu:~/git/sample$ git pull

Updating 186df84..6bdbf82

Fast-forward

src/{main.py => module.py} | 0

1 file changed, 0 insertions(+), 0 deletions(-)

rename src/{main.py => module.py} (100%)

yiibai@ubuntu:~/git/sample$ ls src/

module.py

yiibai@ubuntu:~/git/sample$ git log

commit 6bdbf8219c60d8da9ad352c23628600faaefbe13

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 20:34:28 2017 +0800

renamed main.py to module.py

......

现在，假设要对上面的项目中代码结构进行重构，代码文件：module.py 已经不再使用了，要将它删除，那么应该怎么做？请参考以下命令

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ ls

README.md src

yiibai@ubuntu:~/git/sample$ git rm src/module.py

rm 'src/module.py'

yiibai@ubuntu:~/git/sample$ git commit -a -m "remove/delete module.py"

[master 7d8162d] remove/delete module.py

1 file changed, 26 deletions(-)

delete mode 100644 src/module.py

yiibai@ubuntu:~/git/sample$

**验证删除结果**

在另外一台电脑上，执行以下命令更新当前工作区，查看 sample/src 目录中的文件是否还存在？

$ git pull

## Git修正错误

每个VCS都提供一个功能来修复错误，直到Git控制的某一点上。 Git提供了一个功能，可用于撤消对本地存储库所做的修改。

假设用户意外地对本地存储库进行了一些更改，然后想要撤消这些更改。 在这种情况下，恢复操作起着重要的作用。

* **恢复未提交的更改**

假设我们不小心修改了本地存储库中的一个文件，此时想撤销这些修改。为了处理这种情况，我们可以使用git checkout命令。可以使用此命令来还原文件的内容。

为了更好的演示，我们首先在 sample/src 目录下创建一个文件：string.py ，其代码如下所示

#!/usr/bin/python3

var1 = 'Hello World!'

var2 = "Python Programming"

print ("var1[0]: ", var1[0])

print ("var2[1:5]: ", var2[1:5]) # 切片加索引

并使用以下命令将此文件推送到远程存储库

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git add src/

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git add src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git commit -m "add new file string.py"

[master 44ea8e4] add new file string.py

1 file changed, 7 insertions(+)

create mode 100644 src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (4/4), 443 bytes | 0 bytes/s, done.

Total 4 (delta 0), reused 0 (delta 0)

现在，已经将string.py添加到远程存储库中了。

假设我们不小心/或者有心修改了本地存储库中的一个文件。但现在不想要这些修改的内容了，也就是说想要撤销修改。要处理这种情况，那么可以使用git checkout命令。可以使用此命令来还原文件的内容。

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: src/string.py

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /D/worksp/sample (master)

$ git checkout src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Administrator@MY-PC /D/worksp/sample (master)

$

此外，还可以使用git checkout命令从本地存储库获取已删除的文件。假设我们从本地存储库中删除一个文件，我们想要恢复这个文件。那么可以通过使用git checkout命令来实现这一点。

$ ls -l

total 1

-rw-r--r-- 1 Administ Administ 57 Jul 7 05:37 README.md

drwxr-xr-x 1 Administ Administ 0 Jul 10 21:16 src

Administrator@MY-PC /D/worksp/sample (master)

$ cd src/

Administrator@MY-PC /D/worksp/sample/src (master)

$ ls -l

total 1

-rwxr-xr-x 1 Administ Administ 156 Jul 10 21:16 string.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ rm string.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ ls -l

total 0

Administrator@MY-PC /D/worksp/sample/src (master)

$ git status -s

D string.py

Git在文件名前显示字母D， 这表示该文件已从本地存储库中删除。

$ git checkout string.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ ls -l

total 1

-rwxr-xr-x 1 Administ Administ 156 Jul 10 21:24 string.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

* **删除分段区域的更改**

我们已经看到，当执行添加操作时，文件将从本地存储库移动到暂存区域。 如果用户意外修改文件并将其添加到暂存区域，则可以使用git checkout命令恢复其更改。

在Git中，有一个HEAD指针总是指向最新的提交。 如果要从分段区域撤消更改，则可以使用git checkout命令，但是使用checkout命令，必须提供一个附加参数，即HEAD指针。 附加的提交指针参数指示git checkout命令重置工作树，并删除分段更改。

让我们假设从本地存储库修改一个文件。 如果查看此文件的状态，它将显示该文件已修改但未添加到暂存区域。

$ pwd

/D/worksp/sample/src

Administrator@MY-PC /D/worksp/sample/src (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: string.py

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /D/worksp/sample/src (master)

$ git add string.py

Git状态显示该文件存在于暂存区域，现在使用git checkout命令恢复该文件，并查看还原文件的状态。

$ git checkout head -- string.py

Administrator@MY-PC /D/worksp/sample/src (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

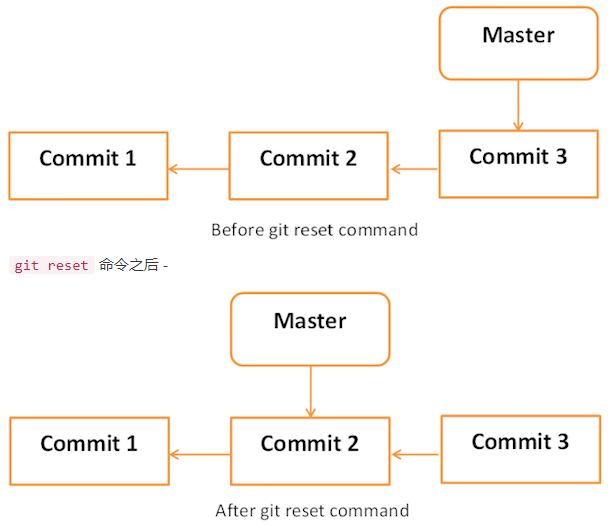
nothing to commit, working directory clean

* **用Git复位移动头指针**

经过少量更改后，可以决定删除这些更改。 git reset命令用于复位或恢复更改。 我们可以执行三种不同类型的复位操作。

下图显示了git reset命令的图示。

git reset命令之前



**soft选项**

每个分支都有一个HEAD指针，它指向最新的提交。 如果用--soft选项后跟提交ID的Git reset命令，那么它将仅重置HEAD指针而不会破坏任何东西。

.git/refs/heads/master文件存储HEAD指针的提交ID。 可使用git log -1命令验证它。

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ cat .git/refs/heads/master

44ea8e47307b47c9a80b44360e09f973e79312b0

现在，查看最新前两个的提交ID，最近一次ID将与上述提交ID一致。

$ git log -2

commit 44ea8e47307b47c9a80b44360e09f973e79312b0

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 21:09:35 2017 +0800

add new file string.py

commit 7d8162db36723b8523c56ad658a07808ae7fb64c

Author: minsu <minsu@yiibai.com>

Date: Mon Jul 10 17:51:11 2017 -0700

remove/delete module.py

Administrator@MY-PC /D/worksp/sample (master)

$

下面我们重置HEAD指针。

现在，只需将HEAD指针重新设置一个位置。现在查看.git/refs/heads/master文件的内容。

Administrator@MY-PC /D/worksp/sample (master)

$ cat .git/refs/heads/master

7d8162db36723b8523c56ad658a07808ae7fb64c

来自文件的提交ID已更改，现在通过查看提交消息进行验证。

$ git log -2

commit 7d8162db36723b8523c56ad658a07808ae7fb64c

Author: minsu <minsu@yiibai.com>

Date: Mon Jul 10 17:51:11 2017 -0700

remove/delete module.py

commit 6bdbf8219c60d8da9ad352c23628600faaefbe13

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 20:34:28 2017 +0800

renamed main.py to module.py

**mixed选项**

使用--mixed选项的Git重置将从尚未提交的暂存区域还原这些更改。它仅从暂存区域恢复更改。对文件的工作副本进行的实际更改不受影响。 默认Git复位等效于执行git reset - mixed。

**hard选项**

如果使用--hard选项与Git重置命令，它将清除分段区域; 它会将HEAD指针重置为特定提交ID的最新提交，并删除本地文件更改。

让我们查看提交ID。

Administrator@MY-PC /D/worksp/sample (master)

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git log -1

commit 7d8162db36723b8523c56ad658a07808ae7fb64c

Author: minsu <minsu@yiibai.com>

Date: Mon Jul 10 17:51:11 2017 -0700

remove/delete module.py

通过在文件开头添加单行注释来修改文件或者往文件里添加其它代码。

$ head -2 src/string.py

#!/usr/bin/python3

Administrator@MY-PC /D/worksp/sample (master)

$ git status -s

M src/string.py

将修改的文件添加到暂存区域，并使用git status命令进行验证。

$ git add src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch and 'origin/master' have diverged,

and have 1 and 1 different commit each, respectively.

(use "git pull" to merge the remote branch into yours)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: src/string.py

Git状态显示该文件存在于暂存区域中。 现在，重置HEAD与--hard选项。

$ git reset --hard 7d8162db36723b8523c56ad658a07808ae7fb64c

HEAD is now at 7d8162d remove/delete module.py

Administrator@MY-PC /D/worksp/sample (master)

git reset命令成功，这将从分段区域还原文件，并删除对文件所做的任何本地更改。

Administrator@MY-PC /D/worksp/sample (master)

$ git status -s

Git状态显示该文件已从暂存区域还原，当前恢复到了删除 moudle.py 时的版本了。

## Git标签操作

标签操作允许为存储库中的特定版本提供有意义的名称。 假设项目中有两个程序员：maxsu和minsu，他们决定标记项目代码，以便以后可以更容易访问这些代码。

### 创建标签

使用git tag命令来标记当前HEAD指针。在创建标签时需要提供-a选项的标签名称，并提供带-m选项的标签消息。

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git tag -a 'Release\_1\_0' -m 'Tagged basic string operation code' HEAD

如果要标记特定提交，则使用相应的COMMIT ID而不是HEAD指针。使用以下命令将标签推送到远程存储库。

$ git push origin tag Release\_1\_0

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 1, done.

Writing objects: 100% (1/1), 177 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

\* [new tag] Release\_1\_0 -> Release\_1\_0

### 查看标签

假设开发人员(maxsu)创建了标签。 现在，另外一个开发人员(minsu)就可以使用带有-l选项的git tag命令查看所有可用的标签。

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ git pull

remote: Counting objects: 1, done.

remote: Total 1 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (1/1), done.

From http://git.oschina.net/yiibai/sample

\* [new tag] Release\_1\_0 -> Release\_1\_0

Already up-to-date.

yiibai@ubuntu:~/git/sample$ git tag -l

Release\_1\_0

yiibai@ubuntu:~/git/sample$

可使用git show命令后跟其标签名来查看有关标签的更多详细信息。

$ git show Release\_1\_0

tag Release\_1\_0

Tagger: maxsu <your\_email@mail.com>

Date: Mon Jul 10 23:06:04 2017 +0800

Tagged basic string operation code

commit 44ea8e47307b47c9a80b44360e09f973e79312b0

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 21:09:35 2017 +0800

add new file string.py

diff --git a/src/string.py b/src/string.py

new file mode 100644

index 0000000..42fd1dd

--- /dev/null

+++ b/src/string.py

@@ -0,0 +1,7 @@

+#!/usr/bin/python3

+

+var1 = 'Hello World!'

+var2 = "Python Programming"

+

+print ("var1[0]: ", var1[0])

+print ("var2[1:5]: ", var2[1:5]) # 切片加索引

\ No newline at end of file

Administrator@MY-PC /D/worksp/sample (master)

$

### 删除标签

使用以下命令从本地以及远程存储库中删除标签，注意使用 git tag -d 中带有-d选项

$ git tag

Release\_1\_0

Administrator@MY-PC /D/worksp/sample (master)

$ git tag -d Release\_1\_0

Deleted tag 'Release\_1\_0' (was 600fa78)

Administrator@MY-PC /D/worksp/sample (master)

$ git push origin :Release\_1\_0

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

To http://git.oschina.net/yiibai/sample.git

- [deleted] Release\_1\_0

Administrator@MY-PC /D/worksp/sample (master)

$

## Git补丁操作

补丁是一个文本文件，其内容类似于git diff，但与代码一样，它也有关于提交的元数据; 例如提交ID，日期，提交消息等。我们可以从提交创建一个补丁，而其他人可以将它们应用到他们的存储库。

假设我们在项目实现了一个strcat函数。并将编写的代码的路径并发送给其他开发人员。 然后，其他开发人员可以将接收的补丁应用到自己的代码中。

我们使用git format-patch命令创建最新提交的修补程序。 如果要为特定提交创建修补程序，请在format-patch命令后面指定 COMMIT\_ID 。

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: src/string.py

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /D/worksp/sample (master)

$ git add src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git commit -m "Added my\_strcat function"

[master cea49f4] Added my\_strcat function

1 file changed, 4 insertions(+), 1 deletion(-)

Administrator@MY-PC /D/worksp/sample (master)

$ git format-patch -1

0001-Added-my\_strcat-function.patch

上述命令在当前工作目录中创建.patch文件。 其他开发人员可以使用这个补丁来修改他的文件。 Git分别提供两个命令：git am 和 git apply 来应用补丁。 git apply修改本地文件而不创建提交，而git am会修改文件并创建提交。

要应用补丁并创建提交，请使用以下命令：

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample/src

yiibai@ubuntu:~/git/sample$ git diff

yiibai@ubuntu:~/git/sample$ git status –s

yiibai@ubuntu:~/git/sample$ git apply 0001-Added-my\_strcat-function.patch

yiibai@ubuntu:~/git/sample$ git status -s

修补程序成功应用，现在我们可以使用git diff命令查看修改。

$ git diff

diff --git a/src/string.py b/src/string.py

index ab42b94..18f165f 100644

--- a/src/string.py

+++ b/src/string.py

@@ -6,4 +6,5 @@ var2 = "Python Programming"

print ("var1[0]: ", var1[0])

print ("var2[1:5]: ", var2[1:5]) # 切片 加索引

-

+def my\_strcat(str1, str2):

+ return (str1+str2)

## Git管理分支

分支操作允许创建另一路线/方向上开发。我们可以使用这个操作将开发过程分为两个不同的方向。 例如，我们发布了1.0版本的产品，可能需要创建一个分支，以便将2.0功能的开发与1.0版本中错误修复分开。

### 创建分支

我们可使用git branch <branch name>命令创建一个新的分支。可以从现有的分支创建一个新的分支。 也可以使用特定的提交或标签作为起点创建分支。 如果没有提供任何特定的提交ID，那么将以HEAD作为起点来创建分支。参考如下代码，创建一个分支：new\_branch

$ git branch new\_branch

Administrator@MY-PC /D/worksp/sample (master)

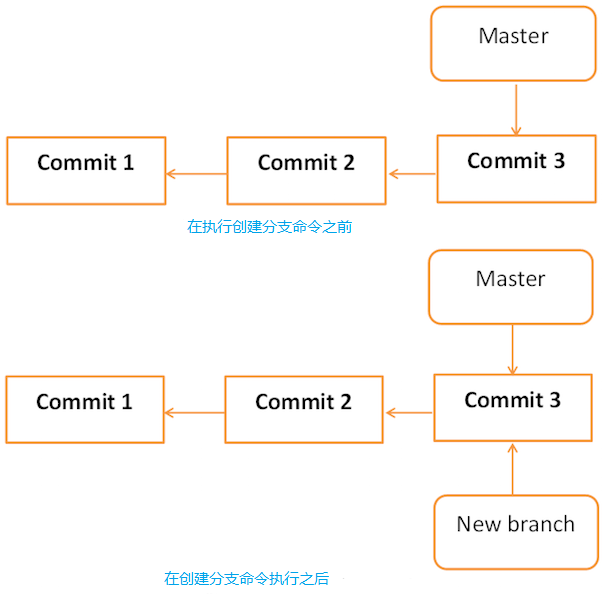
$ git branch

\* master

new\_branch

执行上命令后，它创建了一个新的分支：new\_branch; 使用git branch命令列出可用的分支。Git在当前签出分支之前显示一个星号。

创建分支操作的图示表示如下：



### 切换分支

使用git checkout命令在分支之间切换。

$ git checkout new\_branch

M src/string.py

Switched to branch 'new\_branch'

### 创建和切换分支的快捷方式

在上面的例子中，分别使用两个命令创建和切换分支。 Git为checkout命令提供-b选项; 此操作将创建一个新的分支，并立即切换到新分支。

$ git checkout -b test\_branch

M src/string.py

Switched to a new branch 'test\_branch'

Administrator@MY-PC /D/worksp/sample (test\_branch)

$ git branch

master

new\_branch

\* test\_branch

### 删除分支

可以通过向git branch命令提供-D选项来删除分支。 但在删除现有分支之前，请切换到其他分支。

如上面所示，目前在test\_branch分支，如要想删除该分支。需要先切换到其它分支再删除此分支，如下所示。

$ git branch

master

new\_branch

\* test\_branch

Administrator@MY-PC /D/worksp/sample (test\_branch)

$ git checkout master

M src/string.py

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 4 commits.

(use "git push" to publish your local commits)

Administrator@MY-PC /D/worksp/sample (master)

$ git branch -D test\_branch

Deleted branch test\_branch (was b759faf).

Administrator@MY-PC /D/worksp/sample (master)

当前剩下的分支如下

$ git branch

\* master

new\_branch

### 重命名分支

假设需要在项目中添加对宽字符的支持。并且已经创建了一个新的分支，但分支名称需要重新命名。那么可通过使用-m选项后跟旧的分支名称和新的分支名称来更改/重新命名分支名称。

$ git branch

\* master

new\_branch

Administrator@MY-PC /D/worksp/sample (master)

$ git branch -m new\_branch wchar\_support

现在，使用git branch命令显示新的分支名称。

$ git branch

\* master

wchar\_support

### 合并两个分支

实现一个函数来返回宽字符串的字符串长度。新的代码将显示如下

$ git branch

master

\* wchar\_support

$ pwd

/D/worksp/sample

Administrator@MY-PC /D/worksp/sample (master)

$ git diff

diff --git a/src/string.py b/src/string.py

index 18f165f..89e82b3 100644

--- a/src/string.py

+++ b/src/string.py

@@ -8,3 +8,9 @@ print ("var2[1:5]: ", var2[1:5]) # 切片 加索引

def my\_strcat(str1, str2):

return (str1+str2)

+

+a = '我'

+b = 'ab'

+ab = '我ab'

+

+print(len(a), len(b), len(ab), len('='))

\ No newline at end of file

Administrator@MY-PC /D/worksp/sample (master)

假设经过测试，代码没有问题，最后将其变更推送到新分行

$ git status

On branch master

Your branch is ahead of 'origin/master' by 5 commits.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: src/string.py

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /D/worksp/sample (master)

$ git add src/string.py

Administrator@MY-PC /D/worksp/sample (master)

$ git commit -m 'Added w\_strlen function to return string lenght of wchar\_t

> string'

[master 6bab70a] Added w\_strlen function to return string lenght of wchar\_t string

1 file changed, 6 insertions(+)

请注意，下面将把这些更改推送到新的分支，所以这里使用的分支名称为wchar\_support而不是master分支。

执行过程及结果如下所示

$ git push origin wchar\_support

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 18, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (15/15), 1.72 KiB | 0 bytes/s, done.

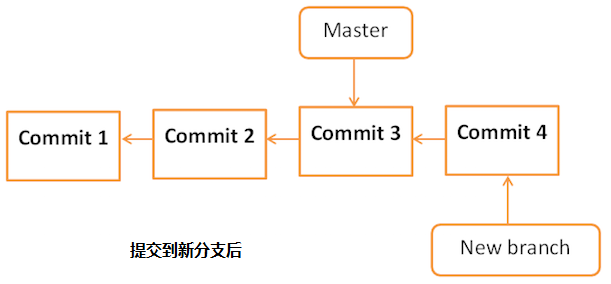
Total 15 (delta 3), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

\* [new branch] wchar\_support -> wchar\_support

Administrator@MY-PC /D/worksp/sample (master)

提交更改后，新分支将显示如下



如果其他开发人员很想知道，我们在私人分支上做什么，那么可从wchar\_support分支检查日志。

yiibai@ubuntu:~/git/sample$ git log origin/wchar\_support -2

输出结果如下

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ git pull

remote: Counting objects: 15, done.

remote: Compressing objects: 100% (12/12), done.

remote: Total 15 (delta 3), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

From http://git.oschina.net/yiibai/sample

\* [new branch] wchar\_support -> origin/wchar\_support

Already up-to-date.

yiibai@ubuntu:~/git/sample$ git log origin/wchar\_support -2

commit b759fafeb2a58bd1104f4142e4c0ababdadce01d

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 23:44:24 2017 +0800

fdasjkfdlaks

commit de08fcc70df3a31c788a2e926263b18498d2df09

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 23:40:00 2017 +0800

delete

yiibai@ubuntu:~/git/sample$

通过查看提交消息，其他开发人员(minsu)到有一个宽字符的相关计算函数，他希望在master分支中也要有相同的功能。不用重新执行代码编写同样的代码，而是通过将分支与主分支合并来执行代码的合并。下面来看看应该怎么做？

yiibai@ubuntu:~/git/sample$ git branch

\* master

yiibai@ubuntu:~/git/sample$ pwd

/home/yiibai/git/sample

yiibai@ubuntu:~/git/sample$ git merge origin/wchar\_support

Updating 44ea8e4..b759faf

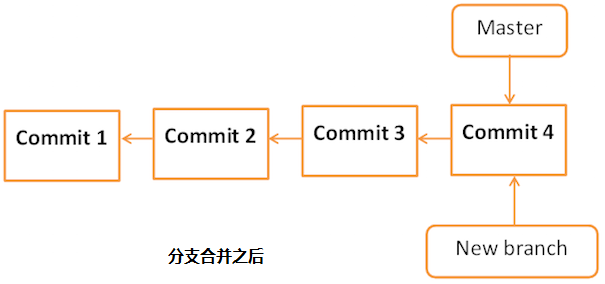
Fast-forward

src/string.py | 4 +++-

1 file changed, 3 insertions(+), 1 deletion(-)

yiibai@ubuntu:~/git/sample$

合并操作后，master分支显示如下



现在，分支wchar\_support已经和master分支合并了。 可以通过查看提交消息或者通过查看string.py文件中的修改来验证它。

yiibai@ubuntu:~/git/sample$ git branch

\* master

yiibai@ubuntu:~/git/sample$ cd src/

yiibai@ubuntu:~/git/sample/src$ git log -2

commit b759fafeb2a58bd1104f4142e4c0ababdadce01d

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 23:44:24 2017 +0800

fdasjkfdlaks

commit de08fcc70df3a31c788a2e926263b18498d2df09

Author: maxsu <your\_email@mail.com>

Date: Mon Jul 10 23:40:00 2017 +0800

上述命令将产生以下结果。

#!/usr/bin/python3

var1 = 'Hello World!'

var2 = "Python Programming"

print ("var1[0]: ", var1[0])

print ("var2[1:5]: ", var2[1:5]) # 切片 加索引

def my\_strcat(str1, str2):

return (str1+str2)

a = '我'

b = 'ab'

ab = '我ab'

print(len(a), len(b), len(ab), len('='))

测试后，就可将代码更改推送到master分支了。

$ git push origin master

Total 0 (delta 0), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

5776472..64192f9 master −> master

## Git处理冲突

假设要在wchar\_support分支中执行更改，修改wchar\_support分支中的代码。添加一个计算长度的函数：count\_len(obj)，代码变化如下

$ git branch

master

\* wchar\_support

Administrator@MY-PC /D/worksp/sample/src (wchar\_support)

$ git diff

diff --git a/src/string.py b/src/string.py

index ba6d584..4307fe2 100644

--- a/src/string.py

+++ b/src/string.py

@@ -13,4 +13,7 @@ a = '我' #

b = 'ab'

ab = '我ab'

-print(len(a), len(b), len(ab), len('='))

\ No newline at end of file

+print(len(a), len(b), len(ab), len('='))

+

+def count\_len(obj):

+ return len(obj)

\ No newline at end of file

假设验证代码后，没有问题就提交这些更改。

$ git status

On branch wchar\_support

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: string.py

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@MY-PC /D/worksp/sample/src (wchar\_support)

$ git add string.py

Administrator@MY-PC /D/worksp/sample/src (wchar\_support)

$ git commit -m "add new function: count\_len(obj)"

[wchar\_support 1cc9ddb] add new function: count\_len(obj)

1 file changed, 4 insertions(+), 1 deletion(-)

* **执行 master 分支变更**

同时在master分支中，另外一个开发人员(minsu)还会更改了内容，并将其更改推送到master分支。

yiibai@ubuntu:~/git/sample/src$ git diff

diff --git a/src/string.py b/src/string.py

index ba6d584..5eb2a5d 100644

--- a/src/string.py

+++ b/src/string.py

@@ -13,4 +13,6 @@ a = '我' #

b = 'ab'

ab = '我ab'

-print(len(a), len(b), len(ab), len('='))

\ No newline at end of file

+print(len(a), len(b), len(ab), len('='))

+def obj\_len(obj):

+ return len(obj)

yiibai@ubuntu:~/git/sample/src$

验证差异后，现在就提交更新内容。

yiibai@ubuntu:~/git/sample/src$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: string.py

no changes added to commit (use "git add" and/or "git commit -a")

yiibai@ubuntu:~/git/sample/src$ git add string.py

yiibai@ubuntu:~/git/sample/src$ git commit -m 'Changed function name from w\_strlen to my\_wc\_strlen'

[master 07cd5af] Changed function name from w\_strlen to my\_wc\_strlen

1 file changed, 3 insertions(+), 1 deletion(-)

yiibai@ubuntu:~/git/sample/src$ git push origin master

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 4, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (4/4), 398 bytes | 0 bytes/s, done.

Total 4 (delta 1), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

e7d1734..07cd5af master -> master

在wchar\_support分支上，我们已经实现了一个count\_len(obj)函数。假设经过测试后，提交并将其更改推送到wchar\_support分支。

* **出现冲突**

假设另外一个开发人员(minsu)想看看我们在wchar\_branch分支上做了什么，他试图从wchar\_support分支中拉出最新的变化，但是Git会中断这个操作，并显示以下错误消息。

yiibai@ubuntu:~/git/sample/src$ git pull origin wchar\_support

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 4 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (4/4), done.

From http://git.oschina.net/yiibai/sample

\* branch wchar\_support -> FETCH\_HEAD

e7d1734..1cc9ddb wchar\_support -> origin/wchar\_support

Auto-merging src/string.py

CONFLICT (content): Merge conflict in src/string.py

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

yiibai@ubuntu:~/git/sample/src$

* **解决冲突**

从错误消息中，很明显文件：src/string.py 中存在冲突。运行git diff命令查看更多细节。

yiibai@ubuntu:~/git/sample/src$ git diff

diff --cc src/string.py

index 5eb2a5d,4307fe2..0000000

--- a/src/string.py

+++ b/src/string.py

@@@ -14,5 -14,6 +14,11 @@@ b = 'ab

ab = 'æˆ‘ab'

print(len(a), len(b), len(ab), len('='))

++<<<<<<< HEAD

+def obj\_len(obj):

+ return len(obj)

++=======

+

+ def count\_len(obj):

- return len(obj)

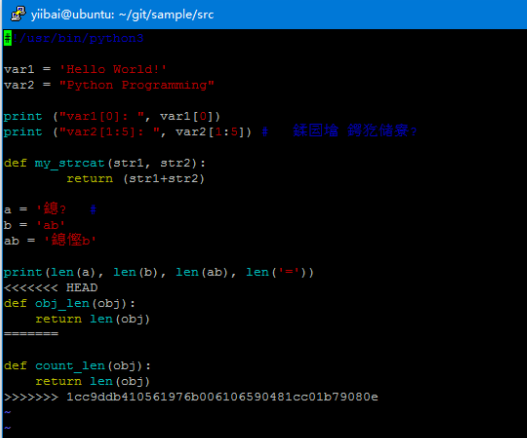
++ return len(obj)

++>>>>>>> 1cc9ddb410561976b006106590481cc01b79080e

yiibai@ubuntu:~/git/sample/src$

由于两个人同进修改了string.py中的代码，所以Git处于混乱状态，并且要求用户手动解决冲突。

假设maxsu决定保留修改的代码，并删除了自己定义的函数：obj\_len(obj)。删除冲突标记后(<<<<<<<<<<<<<<<< 和 >>>>>>>>>>>>>>>>>>>>的行)，现在冲突的代码如下所示



解决冲突需要修改代码后，如下所示



git diff将如下所示

yiibai@ubuntu:~/git/sample/src$ git diff

diff --cc src/string.py

index 5eb2a5d,4307fe2..0000000

--- a/src/string.py

+++ b/src/string.py

@@@ -14,5 -14,6 +14,7 @@@ b = 'ab

ab = 'æˆ‘ab'

print(len(a), len(b), len(ab), len('='))

+

-def count\_len(obj):

- return len(obj)

+def obj\_len(obj):

+ return len(obj)

++

yiibai@ubuntu:~/git/sample/src$

由于minsu已经修改了这些文件，所以必须首先提交这些修改，然后就可以提出这些修改。如下所示

yiibai@ubuntu:~/git/sample/src$ git add .

yiibai@ubuntu:~/git/sample/src$ git commit -a -m 'Resolved conflict'

[master 33f0406] Resolved conflict

yiibai@ubuntu:~/git/sample/src$ git pull origin wchar\_support

已经解决了冲突，现在执行git pull应该没有问题了。

## Git不同平台换行符问题

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

为了避免这些行结尾的差异的不必要提交，我们必须配置Git客户端写入与Git仓库使用相同的行结束符。

对于Windows系统，可以将Git客户端配置为将行结束符转换为CRLF格式，同时退出，并在提交操作时将其转换回LF格式。以下可根据您的需要来设置。

$ git config --global core.autocrlf true

对于GNU/Linux或Mac OS，我们可以配置Git客户端，以便在执行结帐操作时将线结束从CRLF转换为LF

yiibai@ubuntu:~$ git config --global core.autocrlf input

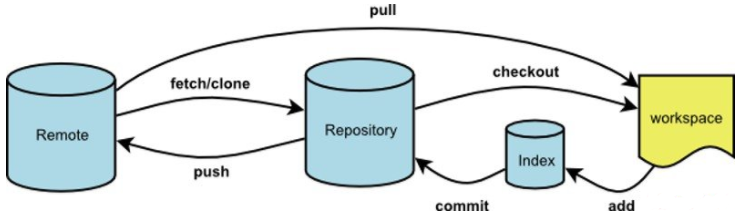
## Git远程操作详解

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。

1. git clone
2. git remote
3. git fetch
4. git pull
5. 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。



## Git常用命令

### git config命令

git config命令用于获取并设置存储库或全局选项。这些变量可以控制Git的外观和操作的各个方面。

**语法简介**

git config [<file-option>] [type] [--show-origin] [-z|--null] name [value [value\_regex]]

git config [<file-option>] [type] --add name value

git config [<file-option>] [type] --replace-all name value [value\_regex]

git config [<file-option>] [type] [--show-origin] [-z|--null] --get name [value\_regex]

git config [<file-option>] [type] [--show-origin] [-z|--null] --get-all name [value\_regex]

git config [<file-option>] [type] [--show-origin] [-z|--null] [--name-only] --get-regexp name\_regex [value\_regex]

git config [<file-option>] [type] [-z|--null] --get-urlmatch name URL

git config [<file-option>] --unset name [value\_regex]

git config [<file-option>] --unset-all name [value\_regex]

git config [<file-option>] --rename-section old\_name new\_name

git config [<file-option>] --remove-section name

git config [<file-option>] [--show-origin] [-z|--null] [--name-only] -l | --list

git config [<file-option>] --get-color name [default]

git config [<file-option>] --get-colorbool name [stdout-is-tty]

git config [<file-option>] -e | --edit

**描述**

可以使用此命令查询/设置/替换/取消设置选项。该名称实际上是由点(.)分隔键，该值将被转义。

可以使用--add选项将多行添加到选项。如果要更新或取消设置多行可能出现的选项，则需要给出POSIX正则表达式value\_regex。 只有与正则表达式匹配的现有值已更新或未设置。如果要处理与正则表达式不匹配的行，只需在前面添加一个感叹号。

类型说明符可以是--int或--bool，以使git config确保变量是给定类型，并将该值转换为规范形式(int是简单十进制数，“true”或 “false” 的布尔字符串表示)，或者--path，它进行一些路径扩展。如果没有类型说明符传递，则不会对该值执行检查或转换。

读取时，默认情况下从系统，全局和存储库本地配置文件读取这些值，而选项--system，--global，--local和--file <filename>可用于告知命令从只有那个位置设置和读取。

写入时，默认情况下将新值写入存储库本地配置文件，并且可以使用选项--system，--global，--file <filename>来告诉命令写入该位置(可以指定为 --local，但这是默认值)。

此错误后，此命令将失败，并显示非零状态。 一些退出代码是：

1. 部分或键无效(ret = 1)，
2. 没有提供部分或名称(ret = 2)，
3. 配置文件无效(ret = 3)，
4. 配置文件无法写入(ret = 4)，
5. 尝试取消设置不存在的选项(ret = 5)，
6. 尝试取消设置/设置多个行匹配的选项(ret = 5)或
7. 尝试使用无效的正则表达式(ret = 6)。

成功后，命令返回退出代码是：0

1. **配置文件的存储位置**

这些变量可以被存储在三个不同的位置：

1. /etc/gitconfig 文件：包含了适用于系统所有用户和所有库的值。如果你传递参数选项’—system’ 给 git config，它将明确的读和写这个文件。
2. ~/.gitconfig 文件 ：具体到你的用户。你可以通过传递—global 选项使Git 读或写这个特定的文件。
3. 位于git目录的config文件 (也就是 .git/config) ：无论当前在用的库是什么，特定指向该单一的库。每个级别重写前一个级别的值。因此，在.git/config中的值覆盖了在/etc/gitconfig中的同一个值。
4. **配置用户名和密码**

当安装Git后首先要做的事情是设置用户名称和e-mail地址。这是非常重要的，因为每次Git提交都会使用该信息。它被永远的嵌入到了你的提交中：

$ git config --global user.name "maxsu"

$ git config --global user.email "yiibai.com@gmail.com"

重申一遍，只需要做一次这个设置。如果传递了 --global 选项，因为Git将总是会使用该信息来处理在系统中所做的一切操作。如果希望在一个特定的项目中使用不同的名称或e-mail地址，可以在该项目中运行该命令而不要--global选项。

$ git config user.name "maxsu"

$ git config user.email "yiibai.com@gmail.com"

1. **配置编缉器**

可以配置默认的文本编辑器，Git在需要你输入一些消息时会使用该文本编辑器。缺省情况下，Git使用系统的缺省编辑器，这通常可能是 vi 或者 vim 。如果想使用一个不同的文本编辑器，例如：Emacs，可以按照如下操作：

$ git config --global core.editor emacs

1. **配置比较工具**

另外一个你可能需要配置的有用的选项是缺省的比较工具它用来解决合并时的冲突。例如，想使用 vimdiff 作为比较工具:

$ git config --global merge.tool vimdiff

Git可以接受 kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, 和 opendiff 作为有效的合并工具。也可以设置一个客户端的工具；

1. **检查配置**

如果想检查你的设置，可以使用 git config --list 命令来列出Git可以在该处找到的所有的设置:

$ git config --list

core.symlinks=false

core.autocrlf=true

color.diff=auto

color.status=auto

color.branch=auto

color.interactive=true

pack.packsizelimit=2g

help.format=html

http.sslcainfo=/bin/curl-ca-bundle.crt

sendemail.smtpserver=/bin/msmtp.exe

diff.astextplain.textconv=astextplain

rebase.autosquash=true

user.email=yiibai.com@gmail.com

user.name=minsu

user.author=author-maxsu

core.autocrlf=true

core.repositoryformatversion=0

core.filemode=false

core.bare=false

core.logallrefupdates=true

core.symlinks=false

core.ignorecase=true

core.hidedotfiles=dotGitOnly

gui.wmstate=normal

gui.geometry=799x475+304+80 287 214

user.name=maxsu

可能会看到一个关键字出现多次(如这里的:user.name就有两个值)，这是因为Git从不同的文件中(例如：/etc/gitconfig以及~/.gitconfig)读取相同的关键字。在这种情况下，对每个唯一的关键字，Git使用最后的那个值。

也可以查看Git认为的一个特定的关键字目前的值，使用如下命令 git config {key}:

$ git config user.name

maxsu

Administrator@MY-PC /C/Users/Administrator/Desktop (master)

$ git config user.email

your\_email@mail.com

**添加/删除配置项**

**添加配置项**

参数

1. -–add

格式: git config [–local|–global|–system] –add section.key value(默认是添加在 local 配置中)

注意add后面的 section, key, value 一项都不能少，否则添加失败。比如执行：

$ git config -–add site.name yiibai

**删除配置项**

命令参数

1. -–unset

格式：git config [–local|–global|–system] –unset section.key

相信有了前两个命令的使用基础，大家举一反三就知道改怎么用了，现在试试删除 local 配置中的site.name配置值 -

$ git config --local -–unset site.name

1. **获取帮助**

如果在使用Git时需要帮助，有三种方法可以获得任何git命令的手册页(manpage)帮助信息:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

例如，您想要看看有关 git config 如何使用，那么可以使用以下命令

$ git help config

### git help命令

git help命令显示有关Git的帮助信息。

1. **简介**

git help [-a|--all] [-g|--guide]

[-i|--info|-m|--man|-w|--web] [COMMAND|GUIDE]

1. **描述**

没有选项，没有给出任何命令或指导，git命令的概要和最常用的Git命令的列表打印在标准输出上。

如果给出--all或-a选项，则所有可用的命令都将打印在标准输出上。

如果给出了--guide或-g选项，那么在标准输出中也会列出有用的Git指南。

如果给出了命令或指南，则会提出该命令或指南的手册页。该程序默认用于此目的，但这可以被其他选项或配置变量覆盖。

注意，git --help ...与git help相同，因为前者被内部转换为后者。

要显示 git 手册页，请使用命令：git help git。

**注意关于git config —global**

请注意，所有这些配置变量应该使用--global标志设置，例如：

$ git config --global help.format web

$ git config --global web.browser firefox

1. **示例**

比如，想要查看 help 命令如何使用，可以使用以下命令

$ git help help

### git init命令

git init命令创建一个空的Git仓库或重新初始化一个现有仓库。

1. **简介**

git init [-q | --quiet] [--bare] [--template=<template\_directory>]

[--separate-git-dir <git dir>]

[--shared[=<permissions>]] [directory]

1. **描述**

该命令创建一个空的Git仓库 - 基本上是创建一个具有objects，refs/head，refs/tags和模板文件的.git目录。 还创建了引用主分支的HEAD初始的一个HEAD文件。

如果通过$GIT\_OBJECT\_DIRECTORY环境变量指定了对象存储目录，那么将在下面创建sha1目录，否则将使用默认的$GIT\_DIR/objects目录。

现有存储库中运行git init命令是安全的。 它不会覆盖已经存在的东西。 重新运行git init的主要原因是拾取新添加的模板(或者如果给出了--separate-git-dir，则将存储库移动到另一个地方)。

1. **示例**

为现有的代码库启动一个新的Git仓库

$ cd /path/to/my/codebase

$ git init #(1)

$ git add . #(2)

$ git commit . -m "a commit message" #(3)

1. 创建一个/path/to/my/codebase/.git目录。
2. 将所有现有文件添加到索引。
3. 将原始状态记录为历史的第一个提交。

### git add命令

git add命令将文件内容添加到索引(将修改添加到暂存区)。也就是将要提交的文件的信息添加到索引库中。

1. **简介**

git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]

[--edit | -e] [--[no-]all | --[no-]ignore-removal | [--update | -u]]

[--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing]

[--chmod=(+|-)x] [--] [<pathspec>…​]

1. **描述**

此命令将要提交的文件的信息添加到索引库中(将修改添加到暂存区)，以准备为下一次提交分段的内容。 它通常将现有路径的当前内容作为一个整体添加，但是通过一些选项，它也可以用于添加内容，只对所应用的工作树文件进行一些更改，或删除工作树中不存在的路径了。

“索引”保存工作树内容的快照，并且将该快照作为下一个提交的内容。 因此，在对工作树进行任何更改之后，并且在运行git commit命令之前，必须使用git add命令将任何新的或修改的文件添加到索引。

该命令可以在提交之前多次执行。它只在运行git add命令时添加指定文件的内容; 如果希望随后的更改包含在下一个提交中，那么必须再次运行git add将新的内容添加到索引。

git status命令可用于获取哪些文件具有为下一次提交分段的更改的摘要。

默认情况下，git add命令不会添加忽略的文件。 如果在命令行上显式指定了任何忽略的文件，git add命令都将失败，并显示一个忽略的文件列表。由Git执行的目录递归或文件名遍历所导致的忽略文件将被默认忽略。 git add命令可以用-f(force)选项添加被忽略的文件。

1. **示例**

以下是一些示例

添加documentation目录及其子目录下所有\*.txt文件的内容：

$ git add documentation/\*.txt

将所有 git-\*.sh 脚本内容添加：

$ git add git-\*.sh

因为这个例子让shell扩展星号(即明确列出文件)，所以它不考虑子目录中的文件，如：subdir/git-foo.sh 这样的文件不会被添加。

**基本用法**

git add <path>

通常是通过git add <path>的形式把<path>添加到索引库中，<path>可以是文件也可以是目录。

git不仅能判断出<path>中，修改(不包括已删除)的文件，还能判断出新添的文件，并把它们的信息添加到索引库中。

$ git add . # 将所有修改添加到暂存区

$ git add \* # Ant风格添加修改

$ git add \*Controller # 将以Controller结尾的文件的所有修改添加到暂存区

$ git add Hello\* # 将所有以Hello开头的文件的修改添加到暂存区 例如:HelloWorld.txt,Hello.java,HelloGit.txt ...

$ git add Hello? # 将以Hello开头后面只有一位的文件的修改提交到暂存区 例如:Hello1.txt,HelloA.java 如果是HelloGit.txt或者Hello.java是不会被添加的

git add -u [<path>]: 把<path>中所有跟踪文件中被修改过或已删除文件的信息添加到索引库。它不会处理那些不被跟踪的文件。省略<path>表示 . ,即当前目录。

git add -A: []表示把中所有跟踪文件中被修改过或已删除文件和所有未跟踪的文件信息添加到索引库。省略<path>表示 . ,即当前目录。

git add -i

我们可以通过git add -i [<path>]命令查看中被所有修改过或已删除文件但没有提交的文件，并通过其revert子命令可以查看<path>中所有未跟踪的文件，同时进入一个子命令系统。

比如：

$ git add -i

staged unstaged path

1: +0/-0 nothing branch/t.txt

2: +0/-0 nothing branch/t2.txt

3: unchanged +1/-0 readme.txt

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

这里的t.txt和t2.txt表示已经被执行了git add，待提交。即已经添加到索引库中。

readme.txt表示已经处于tracked下，它被修改了，但是还没有执行git add。即还没添加到索引库中。

### git clone命令

git clone命令将存储库克隆到新目录中。

1. **简介**

git clone [--template=<template\_directory>]

[-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]

[-o <name>] [-b <name>] [-u <upload-pack>] [--reference <repository>]

[--dissociate] [--separate-git-dir <git dir>]

[--depth <depth>] [--[no-]single-branch]

[--recurse-submodules] [--[no-]shallow-submodules]

[--jobs <n>] [--] <repository> [<directory>]

1. **描述**

将存储库克隆到新创建的目录中，为克隆的存储库中的每个分支创建远程跟踪分支(使用git branch -r可见)，并从克隆检出的存储库作为当前活动分支的初始分支。

在克隆之后，没有参数的普通git提取将更新所有远程跟踪分支，并且没有参数的git pull将另外将远程主分支合并到当前主分支(如果有的话)。

此默认配置通过在refs/remotes/origin下创建对远程分支头的引用，并通过初始化remote.origin.url和remote.origin.fetch配置变量来实现。

执行远程操作的第一步，通常是从远程主机克隆一个版本库，这时就要用到git clone命令。

$ git clone <版本库的网址>

比如，克隆jQuery的版本库

$ git clone http://github.com/jquery/jquery.git

该命令会在本地主机生成一个目录，与远程主机的版本库同名。如果要指定不同的目录名，可以将目录名作为git clone命令的第二个参数。

$ git clone <版本库的网址> <本地目录名>

git clone支持多种协议，除了HTTP(s)以外，还支持SSH、Git、本地文件协议等，下面是一些例子。

1. **示例**

以下是所支持协议的一些示例

$ git clone http[s]://example.com/path/to/repo.git

$ git clone http://git.oschina.net/yiibai/sample.git

$ git clone ssh://example.com/path/to/repo.git

$ git clone git://example.com/path/to/repo.git

$ git clone /opt/git/project.git

$ git clone file:///opt/git/project.git

$ git clone ftp[s]://example.com/path/to/repo.git

$ git clone rsync://example.com/path/to/repo.git

SSH协议还有另一种写法。

$ git clone [user@]example.com:path/to/repo.git

通常来说，Git协议下载速度最快，SSH协议用于需要用户认证的场合。

**应用场景示例**

从上游克隆下来：

$ git clone git://git.kernel.org/pub/scm/.../linux.git mydir

$ cd mydir

$ make # 执行代码或其它命令

在当前目录中使用克隆，而无需检出：

$ git clone -l -s -n . ../copy

$ cd ../copy

$ git show-branch

从现有本地目录借用从上游克隆：

$ git clone --reference /git/linux.git

git://git.kernel.org/pub/scm/.../linux.git

mydir

$ cd mydir

创建一个裸存储库以将您的更改发布给公众：

$ git clone --bare -l /home/proj/.git /pub/scm/proj.git

### git status命令

git status命令用于显示工作目录和暂存区的状态。使用此命令能看到那些修改被暂存到了, 哪些没有, 哪些文件没有被Git tracked到。git status不显示已经commit到项目历史中去的信息。看项目历史的信息要使用git log.

1. **简介**

git status [<options>…​] [--] [<pathspec>…​]

1. **描述**

显示索引文件和当前HEAD提交之间的差异，在工作树和索引文件之间有差异的路径以及工作树中没有被Git跟踪的路径。 第一个是通过运行git commit来提交的; 第二个和第三个是你可以通过在运行git commit之前运行git add来提交的。

git status相对来说是一个简单的命令，它简单的展示状态信息。输出的内容分为3个分类/组。

# On branch master

# Changes to be committed: (已经在stage区, 等待添加到HEAD中的文件)

# (use "git reset HEAD <file>..." to unstage)

#

#modified: hello.py

#

# Changes not staged for commit: (有修改, 但是没有被添加到stage区的文件)

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

#modified: main.py

#

# Untracked files:(没有tracked过的文件, 即从没有add过的文件)

# (use "git add <file>..." to include in what will be committed)

#

#hello.pyc

**忽略文件(untracked文件)**

没有tracked的文件分为两类. 一是已经被放在工作目录下但是还没有执行 git add 的, 另一类是一些编译了的程序文件(如.pyc, .obj, .exe等)。当这些不想add的文件一多起来, git status的输出简直没法看, 一大堆的状态信息怎么看?

基于这个原因。 Git让我们能在一个特殊的文件.gitignore中把要忽略的文件放在其中， 每一个想忽略的文件应该独占一行, \*这个符号可以作为通配符使用。例如在项目根目录下的.gitignore文件中加入下面内容能阻止.pyc和.tmp文件出现在git status中:

\*.pyc

\*.tmp

1. **示例**

以下是一些示例

在每次执行 git commit之前先使用git status检查文件状态是一个很好的习惯, 这样能防止你不小心提交了您不想提交的东西。 下面的例子展示 stage 前后的状态, 并最后提交一个快照.

# Edit hello.py

$ git status

# hello.py is listed under "Changes not staged for commit"

$ git add hello.py

$ git status

# hello.py is listed under "Changes to be committed"

$ git commit

$ git status

# nothing to commit (working directory clean)

第一个状态输出显示了这个文件没有被放到暂存区(staged)。git add将影响第二个git status的输出, 最后一个git status告诉我们没有什么能可以提交了，工作目录已经和最近的提交相匹配了。有些命令 (如, git merge) 要求工作目录是clean状态, 这样就不会不小心覆盖更新了。

git status命令可以列出当前目录所有还没有被git管理的文件和被git管理且被修改但还未提交(git commit)的文件。下面来看看如下一个示例

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: 2.txt

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: 1.txt

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# 1.log

上面输出结果中”Changes to be committed“中所列的内容是在索引中的内容，提交之后进入Git工作目录。

上面输出结果中“Changed but not updated”中所列的内容是在工作目录中的内容，git add之后将添加进入索引。

上面输出结果中“Untracked files”中所列的内容是尚未被Git跟踪的内容，git add之后进入添加进入索引。

通过git status -uno可以只列出所有已经被git管理的且被修改但没提交的文件。

### git diff命令

git diff命令用于显示提交和工作树等之间的更改。此命令比较的是工作目录中当前文件和暂存区域快照之间的差异,也就是修改之后还没有暂存起来的变化内容。

1. **简介**

git diff [options] [<commit>] [--] [<path>…​]

git diff [options] --cached [<commit>] [--] [<path>…​]

git diff [options] <commit> <commit> [--] [<path>…​]

git diff [options] <blob> <blob>

git diff [options] [--no-index] [--] <path> <path>

1. **描述**

在工作树和索引或树之间显示更改，索引和树之间的更改，两个树之间的更改，两个blob对象之间的更改或两个文件在磁盘上的更改。

为了防止异常情况发生，请注意，上述描述中的所有<commit>除了使用“..”符号的最后两种形式之外，都可以是任何<tree>

1. **示例**

以下是一些示例

git diff <file> # 比较当前文件和暂存区文件差异 git diff

git diff <id1><id1><id2> # 比较两次提交之间的差异

git diff <branch1> <branch2> # 在两个分支之间比较

git diff --staged # 比较暂存区和版本库差异

git diff --cached # 比较暂存区和版本库差异

git diff --stat # 仅仅比较统计信息

**检查工作树的几种方式**

$ git diff #(1)

$ git diff --cached #(2)

$ git diff HEAD #(3)

1. 工作树中的更改尚未分段进行下一次提交。
2. 索引和最后一次提交之间的变化; 查看已经git add ，但没有git commit 的改动。
3. 自上次提交以来工作树中的更改；如果运行“git commit -a”，查看将会提交什么。

查看尚未暂存的文件更新了哪些部分，不加参数直接输入

$ git diff

此命令比较的是工作目录(Working tree)和暂存区域快照(index)之间的差异

也就是修改之后还没有暂存起来的变化内容。

查看已经暂存起来的文件(staged)和上次提交时的快照之间(HEAD)的差异

$ git diff --cached

$ git diff --staged

显示的是下一次提交时会提交到HEAD的内容(不带-a情况下)

显示工作版本(Working tree)和HEAD的差别

$ git diff HEAD

直接将两个分支上最新的提交做diff

$ git diff topic master

$ #或

$ git diff topic..master

输出自topic和master分别开发以来，master分支上的变更。

$ git diff topic...master

查看简单的diff结果，可以加上--stat参数

$ git diff --stat

查看当前目录和另外一个分支(test)的差别

$ git diff test

显示当前目录和另一个叫’test‘分支的差别

$ git diff HEAD -- ./lib

显示当前目录下的lib目录和上次提交之间的差别(更准确的说是在当前分支下)

比较上次提交和上上次提交

$ git diff HEAD^ HEAD

比较两个历史版本之间的差异

$ git diff SHA1 SHA2

### git commit命令

git commit命令用于将更改记录(提交)到存储库。将索引的当前内容与描述更改的用户和日志消息一起存储在新的提交中。

1. **简介**

git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]

[--dry-run] [(-c | -C | --fixup | --squash) <commit>]

[-F <file> | -m <msg>] [--reset-author] [--allow-empty]

[--allow-empty-message] [--no-verify] [-e] [--author=<author>]

[--date=<date>] [--cleanup=<mode>] [--[no-]status]

[-i | -o] [-S[<keyid>]] [--] [<file>…​]

1. **描述**

git commit命令将索引的当前内容与描述更改的用户和日志消息一起存储在新的提交中。

要添加的内容可以通过以下几种方式指定：

1. 在使用git commit命令之前，通过使用git add对索引进行递增的“添加”更改(注意：修改后的文件的状态必须为“added”);
2. 通过使用git rm从工作树和索引中删除文件，再次使用git commit命令;
3. 通过将文件作为参数列出到git commit命令(不使用--interactive或--patch选项)，在这种情况下，提交将忽略索引中分段的更改，而是记录列出的文件的当前内容(必须已知到Git的内容) ;
4. 通过使用带有-a选项的git commit命令来自动从所有已知文件(即所有已经在索引中列出的文件)中添加“更改”，并自动从已从工作树中删除索引中的“rm”文件 ，然后执行实际提交;
5. 通过使用--interactive或--patch选项与git commit命令一起确定除了索引中的内容之外哪些文件或hunks应该是提交的一部分，然后才能完成操作。

--dry-run选项可用于通过提供相同的参数集(选项和路径)来获取上一个任何内容包含的下一个提交的摘要。

如果您提交，然后立即发现错误，可以使用 git reset 命令恢复。

1. **示例**

以下是一些示例

提交已经被git add进来的改动。

$ git add .

$ # 或者~

$ git add newfile.txt

$ git commit -m "the commit message" #

$ git commit -a # 会先把所有已经track的文件的改动`git add`进来，然后提交(有点像svn的一次提交,不用先暂存)。对于没有track的文件,还是需要执行`git add <file>` 命令。

$ git commit --amend # 增补提交，会使用与当前提交节点相同的父节点进行一次新的提交，旧的提交将会被取消。

录制自己的工作时，工作树中修改后的文件的内容将临时存储到使用git add命名为“索引”的暂存区域。 一个文件只能在索引中恢复，而不是在工作树中，使用git reset HEAD - <file>进行上一次提交的文件，这有效地恢复了git的添加，并阻止了对该文件的更改，以参与下一个提交在使用这些命令构建状态之后，git commit(没有任何pathname参数)用于记录到目前为止已经进行了什么更改。 这是命令的最基本形式。一个例子：

$ vi hello.c

$ git rm goodbye.c

$ git add hello.c

$ git commit

可以在每次更改后暂存文件，而不是在git commit中关注工作树中跟踪内容的文件的更改，可使用相应的git add和git rm。 也就是说，如果您的工作树中没有其他更改(hello.c文件内容不变)，则该示例与前面的示例相同：

$ vi hello.c

$ rm goodbye.c

$ git commit -a

命令git commit -a首先查看您的工作树，注意您已修改hello.c并删除了goodbye.c，并执行必要的git add和git rm。

在更改许多文件之后，可以通过给出git commit的路径名来更改记录更改的顺序。当给定路径名时，该命令提交只记录对命名路径所做的更改：

$ edit hello.c hello.h # 修改了这两个文件的内容

$ git add hello.c hello.h

$ edit Makefile

$ git commit Makefile

这提供了一个记录Makefile修改的提交。 在hello.c和hello.h中升级的更改不会包含在生成的提交中。然而，它们的变化并没有消失 - 他们仍然有更改，只是被阻止。 按照上述顺序执行：

$ git commit

这个第二个提交将按照预期记录更改为hello.c和hello.h。

合并后(由git merge或git pull发起)由于冲突而停止，干净合并的路径已经被暂存为提交，并且冲突的路径保持在未加载状态。 您必须首先检查哪些路径与git状态冲突，并在手工将其固定在工作树中之后，要像往常一样使用git add：

$ git status | grep unmerged

unmerged: hello.c

$ edit hello.c

$ git add hello.c

解决冲突和暂存结果后，git ls-files -u将停止提及冲突的路径。完成后，运行git commit最后记录合并：

$ git commit

### git reset命令

git reset命令用于将当前HEAD复位到指定状态。一般用于撤消之前的一些操作(如：git add,git commit等)。

1. **简介**

git reset [-q] [<tree-ish>] [--] <paths>…​

git reset (--patch | -p) [<tree-ish>] [--] [<paths>…​]

git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]

1. **描述**

在第一和第二种形式中，将条目从<tree-ish>复制到索引。 在第三种形式中，将当前分支头(HEAD)设置为<commit>，可选择修改索引和工作树进行匹配。所有形式的<tree-ish>/<commit>默认为 HEAD 。

这里的 HEAD 关键字指的是当前分支最末梢最新的一个提交。也就是版本库中该分支上的最新版本。

1. **示例**

以下是一些示例

在git的一般使用中，如果发现错误的将不想暂存的文件被git add进入索引之后，想回退取消，则可以使用命令：git reset HEAD <file>，同时git add完毕之后，git也会做相应的提示，比如：

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: test.py

git reset [--hard|soft|mixed|merge|keep] [<commit>或HEAD]：将当前的分支重设(reset)到指定的<commit>或者HEAD(默认，如果不显示指定<commit>，默认是HEAD，即最新的一次提交)，并且根据[mode]有可能更新索引和工作目录。mode的取值可以是hard、soft、mixed、merged、keep。下面来详细说明每种模式的意义和效果。

A). --hard：重设(reset) 索引和工作目录，自从<commit>以来在工作目录中的任何改变都被丢弃，并把HEAD指向<commit>。

下面是具体一个例子，假设有三个commit， 执行 git status结果如下:

commit3: add test3.c

commit2: add test2.c

commit1: add test1.c

执行git reset --hard HEAD~1命令后，

显示：HEAD is now at commit2，运行git log，如下所示

commit2: add test2.c

commit1: add test1.c

**应用场景**

下面列出一些git reset的典型的应用场景：

1. **回滚添加操作**

$ edit file1.c file2.c # (1)

$ git add file1.c file1.c # (1.1) 添加两个文件到暂存

$ mailx # (2)

$ git reset # (3)

$ git pull git://info.example.com/ nitfol # (4)

1. 编辑文件 file1.c, file2.c，做了些更改，并把更改添加到了暂存区。
2. 查看邮件，发现某人要您执行git pull，有一些改变需要合并下来。
3. 然而，您已经把暂存区搞乱了，因为暂存区同HEAD commit不匹配了，但是即将git pull下来的东西不会影响已经修改的file1.c 和 file2.c，因此可以revert这两个文件的改变。在revert后，那些改变应该依旧在工作目录中，因此执行git reset。
4. 然后，执行了git pull之后，自动合并，file1.c 和 file2.c这些改变依然在工作目录中。
5. **回滚最近一次提交**

$ git commit -a -m "这是提交的备注信息"

$ git reset --soft HEAD^ #(1)

$ edit code #(2) 编辑代码操作

$ git commit -a -c ORIG\_HEAD #(3)

1. 当提交了之后，又发现代码没有提交完整，或者想重新编辑一下提交的信息，可执行git reset --soft HEAD^，让工作目录还跟reset之前一样，不作任何改变。

HEAD^表示指向HEAD之前最近的一次提交。

1. 对工作目录下的文件做修改，比如：修改文件中的代码等。
2. 然后使用reset之前那次提交的注释、作者、日期等信息重新提交。注意，当执行git reset命令时，git会把老的HEAD拷贝到文件.git/ORIG\_HEAD中，在命令中可以使用ORIG\_HEAD引用这个提交。git commit 命令中 -a参数的意思是告诉git，自动把所有修改的和删除的文件都放进暂存区，未被git跟踪的新建的文件不受影响。commit命令中-c <commit> 或者 -C <commit>意思是拿已经提交的对象中的信息(作者，提交者，注释，时间戳等)提交，那么这条git commit 命令的意思就非常清晰了，把所有更改的文件加入暂存区，并使用上次的提交信息重新提交。
3. **回滚最近几次提交，并把这几次提交放到指定分支中**

回滚最近几次提交，并把这几次提交放到叫做topic/wip的分支上去。

$ git branch topic/wip (1)

$ git reset --hard HEAD~3 (2)

$ git checkout topic/wip (3)

1. 假设已经提交了一些代码，但是此时发现这些提交还不够成熟，不能进入master分支，希望在新的branch上暂存这些改动。因此执行了git branch命令在当前的HEAD上建立了新的叫做 topic/wip 的分支。
2. 然后回滚master分支上的最近三次提交。HEAD~3指向当前HEAD-3个提交，git reset --hard HEAD~3，即删除最近的三个提交(删除HEAD, HEAD^, HEAD~2)，将HEAD指向HEAD~3。
3. **永久删除最后几个提交**

$ git commit ## 执行一些提交

$ git reset --hard HEAD~3 (1)

最后三个提交(即HEAD, HEAD^和HEAD~2)提交有问题，想永久删除这三个提交。

1. **回滚merge和pull操作**

$ git pull (1)

Auto-merging nitfol

CONFLICT (content): Merge conflict in nitfol

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

$ git reset --hard (2)

$ git pull . topic/branch (3)

Updating from 41223... to 13134...

Fast-forward

$ git reset --hard ORIG\_HEAD (4)

`

1. 从origin拉取下来一些更新，但是产生了很多冲突，但您暂时没有这么多时间去解决这些冲突，因此决定稍候有空的时候再重新执行git pull操作。
2. 由于git pull操作产生了冲突，因此所有拉取下来的改变尚未提交，仍然再暂存区中，这种情况下git reset --hard 与 git reset --hard HEAD意思相同，即都是清除索引和工作区中被搞乱的东西。
3. 将topic/branch分支合并到当前的分支，这次没有产生冲突，并且合并后的更改自动提交。
4. 但是此时又发现将topic/branch合并过来为时尚早，因此决定退滚合并，执行git reset --hard ORIG\_HEAD回滚刚才的pull/merge操作。说明：前面讲过，执行git reset时，git会把reset之前的HEAD放入.git/ORIG\_HEAD文件中，命令行中使用ORIG\_HEAD引用这个提交。同样的，执行git pull和git merge操作时，git都会把执行操作前的HEAD放入ORIG\_HEAD中，以防回滚操作。
5. **在污染的工作区中回滚合并或者拉取**

$ git pull (1)

Auto-merging nitfol

Merge made by recursive.

nitfol | 20 +++++----

...

$ git reset --merge ORIG\_HEAD (2)

1. 即便你已经在本地更改了工作区中的一些东西，可安全的执行git pull操作，前提是要知道将要git pull下面的内容不会覆盖工作区中的内容。
2. git pull完后，发现这次拉取下来的修改不满意，想要回滚到git pull之前的状态，从前面的介绍知道，我们可以执行git reset --hard ORIG\_HEAD，但是这个命令有个副作用就是清空工作区，即丢弃本地未使用git add的那些改变。为了避免丢弃工作区中的内容，可以使用git reset --merge ORIG\_HEAD，注意其中的--hard 换成了 --merge，这样就可以避免在回滚时清除工作区。
3. **中断的工作流程处理**

在实际开发中经常出现这样的情形：你正在开发一个大的新功能(工作在分支：feature 中)，此时来了一个紧急的bug需要修复，但是目前在工作区中的内容还没有成型，还不足以提交，但是又必须切换的另外的分支去修改bug。请看下面的例子

$ git checkout feature ;# you were working in "feature" branch and

$ work work work ;# got interrupted

$ git commit -a -m "snapshot WIP" (1)

$ git checkout master

$ fix fix fix

$ git commit ;# commit with real log

$ git checkout feature

$ git reset --soft HEAD^ ;# go back to WIP state (2)

$ git reset (3)

1. 这次属于临时提交，因此随便添加一个临时注释即可。
2. 这次reset删除了WIP commit，并且把工作区设置成提交WIP快照之前的状态。
3. 此时，在索引中依然遗留着“snapshot WIP”提交时所做的未提交变化，git reset将会清理索引成为尚未提交”snapshot WIP“时的状态便于接下来继续工作。
4. **重置单独的一个文件**

假设你已经添加了一个文件进入索引，但是而后又不打算把这个文件提交，此时可以使用git reset把这个文件从索引中去除。

$ git reset -- frotz.c (1)

$ git commit -m "Commit files in index" (2)

$ git add frotz.c (3)

1. 把文件frotz.c从索引中去除，
2. 把索引中的文件提交
3. 再次把frotz.c加入索引
4. **保留工作区并丢弃一些之前的提交**

假设你正在编辑一些文件，并且已经提交，接着继续工作，但是现在你发现当前在工作区中的内容应该属于另一个分支，与之前的提交没有什么关系。此时，可以开启一个新的分支，并且保留着工作区中的内容。

$ git tag start

$ git checkout -b branch1

$ edit

$ git commit ... (1)

$ edit

$ git checkout -b branch2 (2)

$ git reset --keep start (3)

1. 这次是把在branch1中的改变提交了。
2. 此时发现，之前的提交不属于这个分支，此时新建了branch2分支，并切换到了branch2上。
3. 此时可以用reset --keep把在start之后的提交清除掉，但是保持工作区不变。

### git rm命令

git rm命令用于从工作区和索引中删除文件。

1. **简介**

git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch] [--quiet] [--] <file>…

1. **描述**

从索引中删除文件，或从工作树和索引中删除文件。 git rm不会从您的工作目录中删除文件。 (没有任何选项只能从工作树中删除文件，并将其保留在索引中;)要删除的文件必须与分支的提示相同，并且在索引中不能对其内容进行更新，尽管可以使用-f选项覆盖(默认行为)。 当给出--cached时，暂存区内容必须与分支的提示或磁盘上的文件相匹配，从而仅将文件从索引中删除。

使用 git rm 来删除文件，同时还会将这个删除操作记录下来；而使用 rm 来删除文件，仅仅是删除了物理文件，没有将其从 git 的记录中剔除。

直观的来讲，git rm 删除过的文件，执行 git commit -m "commit message or mark" 提交时，会自动将删除该文件的操作提交上去。

而对于用 rm 命令直接删除的文件，执行 git commit -m "commit message or mark"提交时，则不会将删除该文件的操作提交上去。不过不要紧，即使你已经通过 rm 将某个文件删除掉了，也可以再通过 git rm 命令重新将该文件从 git 的记录中删除掉，

这样的话，在执行 git commit -m "commit message or mark" 以后，也能将这个删除操作提交上去。

如果之前不小心用 rm 命令删除了一大批文件呢？如此时用 git rm 逐个地再删除一次就显得相当卵痛了。可如下的方式做提交： git commit -am "commit message or mark"

1. **示例**

以下是一些示例

在git中我们可以通过git rm命令把一个文件删除，并把它从git的仓库管理系统中移除。但是注意最后要执行git commit才真正提交到git仓库。

**示例1**

删除text1.txt文件，并把它从git的仓库管理系统中移除。

git rm text1.txt

**示例2**

删除文件夹：mydir，并把它从git的仓库管理系统中移除。

git rm -r mydir

**示例3**

$ git add 10.txt

$ git add -i

staged unstaged path

1: +0/-0 nothing 10.txt

2: +0/-0 nothing branch/t.txt

3: +0/-0 nothing branch/t2.txt

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now> 7

Bye.

$ git rm --cached 10.txt

rm '10.txt'

$ ls

10.txt 2 3.txt 5.txt readme.txt

$ git add -i

staged unstaged path

1: +0/-0 nothing branch/t.txt

2: +0/-0 nothing branch/t2.txt

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

在通过 git add 10.txt 命令把文件10.txt添加到索引库中后，又通过 git rm --cached 10.txt 把文件10.txt从git的索引库中移除,但是对文件10.txt本身并不进行任何操作。

另外对于已经被git rm删除掉(还没被提交)的文件或目录，如果想取消其操作的话，可以首先通过git add -i的子命令revert从索引库中把它们剔除，然后用git checkout <文件> 命令来达到取消的目。

**示例4**

$ git rm Documentation/\\*.txt

从Documentation目录及其任何子目录下的索引中删除所有.txt文件。

**示例5**

git rm -f git-\*.sh

因为这个例子让shell扩展星号(即显式列出文件)，它不会删除子目录中的文件，如：subdir/git-foo.sh文件不会被删除。

### git mv命令

git mv命令用于移动或重命名文件，目录或符号链接。

1. **简介**

git mv <options>… <args>…

1. **描述**

移动或重命名文件，目录或符号链接。

git mv [-v] [-f] [-n] [-k] <source> <destination>

git mv [-v] [-f] [-n] [-k] <source> ... <destination directory>

在第一种形式中，它将重命名<source>为<destination>，<source>必须存在，并且是文件，符号链接或目录。 在第二种形式中，最后一个参数必须是现有的目录; 给定的源(<source>)将被移动到这个目录中。

索引在成功完成后更新，但仍必须提交更改。

1. **示例**

以下是一些示例

把一个文件：text.txt 移动到 mydir，可以执行以下操作 -

$ git text.txt mydir

运行上面的 git mv 其实就相当于运行了3条命令：

$ mv test.txt mydir/

$ git rm test.txt

$ git add mydir

### git branch命令

git branch命令用于列出，创建或删除分支。

1. **简介**

git branch [--color[=<when>] | --no-color] [-r | -a]

[--list] [-v [--abbrev=<length> | --no-abbrev]]

[--column[=<options>] | --no-column] [--sort=<key>]

[(--merged | --no-merged) [<commit>]]

[--contains [<commit]] [--no-contains [<commit>]]

[--points-at <object>] [--format=<format>] [<pattern>…​]

git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]

git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]

git branch --unset-upstream [<branchname>]

git branch (-m | -M) [<oldbranch>] <newbranch>

git branch (-d | -D) [-r] <branchname>…

git branch --edit-description [<branchname>]

1. **描述**

如果给出了--list，或者如果没有非选项参数，则列出现有的分支; 当前分支将以星号突出显示。 选项-r导致远程跟踪分支被列出，而选项-a显示本地和远程分支。 如果给出了一个<pattern>，它将被用作一个shell通配符，将输出限制为匹配的分支。 如果给出多个模式，如果匹配任何模式，则显示分支。 请注意，提供<pattern>时，必须使用--list; 否则命令被解释为分支创建。

使用--contains，仅显示包含命名提交的分支(换句话说，提示提交的分支是指定的提交的后代)，--no-contains会反转它。 随着已经有了，只有分支合并到命名提交(即从提交提交可以提前提交的分支)将被列出。 使用--no合并只会将未合并到命名提交中的分支列出。 如果缺少<commit>参数，则默认为HEAD(即当前分支的提示)。

1. **示例**

以下是一些示例

1. **查看当前有哪些分支**

$ git branch

master

\* wchar\_support

上面显示结果中，当前有两个分支：master 和 wchar\_support，当前在 wchar\_support 分支上，它前面有个星号(\*)。

1. **新建一个分支**

下面命令将创建一个分支：dev2

$ git branch dev2

1. **切换到指定分支**

下面命令将切换到指定分支：dev2

$ git checkout dev2

$ # 再次查看分支

$ git branch

\* dev2

master

wchar\_support

1. **查看本地和远程分支**

$ git branch -a

\* dev2

master

wchar\_support

remotes/origin/HEAD -> origin/master

remotes/origin/master

remotes/origin/wchar\_support

1. **将更改添加到新建分支上**

$ git status

On branch dev2

Untracked files:

(use "git add <file>..." to include in what will be committed)

newfile.txt

nothing added to commit but untracked files present (use "git add" to track)

Administrator@MY-PC /D/worksp/sample (dev2)

$ git add newfile.txt

Administrator@MY-PC /D/worksp/sample (dev2)

$ git commit newfile.txt -m "commit a new file: newfile.txt"

[dev2 c5f8a25] commit a new file: newfile.txt

1 file changed, 2 insertions(+)

create mode 100644 newfile.txt

Administrator@MY-PC /D/worksp/sample (dev2)

$ git push origin dev2

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

Counting objects: 12, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (11/11), 965 bytes | 0 bytes/s, done.

Total 11 (delta 3), reused 0 (delta 0)

To http://git.oschina.net/yiibai/sample.git

\* [new branch] dev2 -> dev2

Administrator@MY-PC /D/worksp/sample (dev2)

$

1. **修改分支的名字**

$ git branch

\* dev2

master

wchar\_support

Administrator@MY-PC /D/worksp/sample (dev2)

$ git branch -m dev2 version.2

Administrator@MY-PC /D/worksp/sample (version.2)

$ git branch -r

origin/HEAD -> origin/master

origin/dev2

origin/master

origin/wchar\_support

Administrator@MY-PC /D/worksp/sample (version.2)

$ git branch

master

\* version.2

wchar\_support

1. **删除远程分支**

删除一个名称为：dev2 的远客

$ git branch

master

\* version.2

wchar\_support

Administrator@MY-PC /D/worksp/sample (version.2)

$ git push origin --delete dev2

Username for 'http://git.oschina.net': 769728683@qq.com

Password for 'http://769728683@qq.com@git.oschina.net':

To http://git.oschina.net/yiibai/sample.git

- [deleted] dev2

1. **合并某个分支到当前分支**

合并分支：version.2到当前分支(master)，如下

$ git branch

master

\* version.2

wchar\_support

Administrator@MY-PC /D/worksp/sample (version.2)

$ git checkout master

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

Administrator@MY-PC /D/worksp/sample (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Administrator@MY-PC /D/worksp/sample (master)

$ git merge version.2

Updating e7d1734..c5f8a25

Fast-forward

mydir/text.txt | 0

newfile.txt | 2 ++

src/string.py | 5 ++++-

3 files changed, 6 insertions(+), 1 deletion(-)

create mode 100644 mydir/text.txt

create mode 100644 newfile.txt

$

### git checkout命令

git checkout命令用于切换分支或恢复工作树文件。git checkout是git最常用的命令之一，同时也是一个很危险的命令，因为这条命令会重写工作区。

1. **使用语法**

git checkout [-q] [-f] [-m] [<branch>]

git checkout [-q] [-f] [-m] --detach [<branch>]

git checkout [-q] [-f] [-m] [--detach] <commit>

git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new\_branch>] [<start\_point>]

git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>…​

git checkout [-p|--patch] [<tree-ish>] [--] [<paths>…]

1. **描述**

更新工作树中的文件以匹配索引或指定树中的版本。如果没有给出路径 - git checkout还会更新HEAD，将指定的分支设置为当前分支。

1. **示例**

以下是一些示例

**示例-1**

以下顺序检查主分支，将Makefile还原为两个修订版本，错误地删除hello.c，并从索引中取回。

$ git checkout master #(1)

$ git checkout master~2 Makefile #(2)

$ rm -f hello.c

$ git checkout hello.c #(3)

1. 切换分支
2. 从另一个提交中取出文件
3. 从索引中恢复hello.c

如果想要检出索引中的所有C源文件，可以使用以下命令

$ git checkout -- '\*.c'

注意:\*.c是使用引号的。 文件hello.c也将被检出，即使它不再在工作树中，因为文件globbing用于匹配索引中的条目(而不是在shell的工作树中)。

如果有一个分支也命名为hello.c，这一步将被混淆为一个指令切换到该分支。应该写：

$ git checkout -- hello.c

**示例-2**

在错误的分支工作后，想切换到正确的分支，则使用：

$ git checkout mytopic

但是，您的“错误”分支和正确的“mytopic”分支可能会在在本地修改的文件中有所不同，在这种情况下，上述检出将会失败：

$ git checkout mytopic

error: You have local changes to 'frotz'; not switching branches.

可以将-m标志赋给命令，这将尝试三路合并：

$ git checkout -m mytopic

Auto-merging frotz

在这种三路合并之后，本地的修改没有在索引文件中注册，所以git diff会显示从新分支的提示之后所做的更改。

**示例-3**

当使用-m选项切换分支时发生合并冲突时，会看到如下所示：

$ git checkout -m mytopic

Auto-merging frotz

ERROR: Merge conflict in frotz

fatal: merge program failed

此时，git diff会显示上一个示例中干净合并的更改以及冲突文件中的更改。 编辑并解决冲突，并用常规方式用git add来标记它：

$ edit frotz # 编辑 frotz 文件中内容，然后重新添加

$ git add frotz

**其它示例**

git checkout的主要功能就是迁出一个分支的特定版本。默认是迁出分支的HEAD版本

一此用法示例：

$ git checkout master #//取出master版本的head。

$ git checkout tag\_name #//在当前分支上 取出 tag\_name 的版本

$ git checkout master file\_name #//放弃当前对文件file\_name的修改

$ git checkout commit\_id file\_name #//取文件file\_name的 在commit\_id是的版本。commit\_id为 git commit 时的sha值。

$ git checkout -b dev/1.5.4 origin/dev/1.5.4

# 从远程dev/1.5.4分支取得到本地分支/dev/1.5.4

$ git checkout -- hello.rb

#这条命令把hello.rb从HEAD中签出.

$ git checkout .

#这条命令把 当前目录所有修改的文件 从HEAD中签出并且把它恢复成未修改时的样子.

#注意：在使用 git checkout 时，如果其对应的文件被修改过，那么该修改会被覆盖掉。

### git merge命令

git merge命令用于将两个或两个以上的开发历史加入(合并)一起。

1. **使用语法**

git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]

[-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]

[--[no-]allow-unrelated-histories]

[--[no-]rerere-autoupdate] [-m <msg>] [<commit>…​]

git merge --abort

git merge --continue

1. **描述**

将来自命名提交的更改(从其历史从当前分支转移到当前分支之后)。 该命令由git pull用于合并来自另一个存储库的更改，可以手动使用将更改从一个分支合并到另一个分支。

1. **示例**

以下是一些示例

**示例-1**

合并分支fixes和enhancements在当前分支的顶部，使它们合并：

$ git merge fixes enhancements

**示例-2**

合并obsolete分支到当前分支，使用ours合并策略：

$ git merge -s ours obsolete

示例-3

将分支maint合并到当前分支中，但不要自动进行新的提交：

$ git merge --no-commit maint

当您想要对合并进行进一步更改时，可以使用此选项，或者想要自己编写合并提交消息。应该不要滥用这个选项来潜入到合并提交中。小修补程序，如版本名称将是可以接受的。

**示例-4**

将分支dev合并到当前分支中，自动进行新的提交：

$ git merge dev

### git mergetool命令

git mergetool命令用于运行合并冲突解决工具来解决合并冲突。

1. **使用语法**

git mergetool [--tool=<tool>] [-y | --[no-]prompt] [<file>…]

1. **描述**

git mergetool命令用于运行合并冲突解决工具来解决合并冲突。使用git mergetool运行合并实用程序来解决合并冲突。它通常在git合并后运行。

如果给出一个或多个<file>参数，则将运行合并工具程序来解决每个文件的差异(跳过那些没有冲突的文件)。 指定目录将包括该路径中的所有未解析文件。 如果没有指定<file>名称，git mergetool将在具有合并冲突的每个文件上运行合并工具程序。

1. **示例**

以下是一些示例

git设置 mergetool 可视化工具

可以设置BeyondCompare,DiffMerge等作为git的比较和合并的可视化工具,方便操作。

设置如下:

先下载并安装 BeyondCompare,DiffMerge 等，这里以 BeyondCompare 为例。

设置git配置,设置 BeyondCompare 的git命令如下:

#difftool 配置

git config --global diff.tool bc4

git config --global difftool.bc4.cmd "\"c:/program files (x86)/beyond compare 4/bcomp.exe\" \"$LOCAL\" \"$REMOTE\""

#mergeftool 配置

git config --global merge.tool bc4

git config --global mergetool.bc4.cmd "\"c:/program files (x86)/beyond compare 4/bcomp.exe\" \"$LOCAL\" \"$REMOTE\" \"$BASE\" \"$MERGED\""

git config --global mergetool.bc4.trustExitCode true

#让git mergetool不再生成备份文件(\*.orig)

git config --global mergetool.keepBackup false

使用方法如下:

1. diff使用方法:

git difftool HEAD // 比较当前修改情况

1. merge使用方法

git mergetool

### git log命令

git log命令用于显示提交日志信息。

1. **使用语法**

git log [<options>] [<revision range>] [[\--] <path>…]

1. **描述**

git log命令用于显示提交日志信息。

该命令采用适用于git rev-list命令的选项来控制显示的内容以及如何以及适用于git diff- \*命令的选项，以控制如何更改每个提交引入的内容。

1. **示例**

以下是一些示例

1. 显示整个提交历史记录，但跳过合并

$ git log --no-merges

commit c5f8a258babf5eec54edc794ff980d8340396592

Author: maxsu <your\_email@mail.com>

Date: Wed Jul 12 22:07:59 2017 +0800

commit a new file: newfile.txt

... ...

1. 显示自v2.6.12版以来所有提交更改include/scsi或drivers/scsi子目录中的任何文件的所有提交

$ git log master include/scsi drivers/scsi

1. 显示最近两周的更改文件gitk。 “--”是必要的，以避免与名为gitk的分支混淆

$ git log --since="2 weeks ago" -- gitk

1. 显示“test”分支中尚未在“release”分支中的提交，以及每个提交修改的路径列表

$ git log --name-status release..test

1. 显示更改builtin/rev-list.c的提交，包括在文件被赋予其现有名称之前发生的提交。

$ git log --follow builtin/rev-list.c

1. 显示在任何本地分支中的所有提交，但不包括任何远程跟踪分支机构的起始点(origin不具有)。

git log --branches --not --remotes=origin

1. 显示本地主服务器中的所有提交，但不显示任何远程存储库主分支。

git log master --not --remotes=\*/master

1. 显示历史，包括变化差异，但仅从“主分支”的角度来看，忽略来自合并分支的提交，并显示合并引入的变化的完全差异。只有当遵守在一个整合分支上合并所有主题分支的严格策略时，这才有意义。

git log -p -m --first-parent

1. 显示文件main.c中的函数main()随着时间的推移而演变。

git log -L '/int main/',/^}/:main.c

1. 将显示最近三次的提交。

git log -3

1. 根据提交ID查询日志

$ git log commit\_id 　　#查询ID(如：6bab70a08afdbf3f7faffaff9f5252a2e4e2d552)之前的记录，包含commit

$ git log commit1\_id commit2\_id #查询commit1与commit2之间的记录，包括commit1和commit2

$ git log commit1\_id..commit2\_id #同上，但是不包括commit1

其中，commit\_id可以是提交哈希值的简写模式，也可以使用HEAD代替。HEAD代表最后一次提交，HEAD^为最后一个提交的父提交，等同于HEAD～1，HEAD～2代表倒数第二次提交

--pretty按指定格式显示日志信息,可选项有：oneline,short,medium,full,fuller,email,raw以及format:,默认为medium，可以通过修改配置文件来指定默认的方式。

$ git log (--pretty=)oneline

常见的format选项：

#选项 #说明

%H 提交对象(commit)的完整哈希字串

%h 提交对象的简短哈希字串

%T 树对象(tree)的完整哈希字串

%t 树对象的简短哈希字串

%P 父对象(parent)的完整哈希字串

%p 父对象的简短哈希字串

%an 作者(author)的名字

%ae 作者的电子邮件地址

%ad 作者修订日期(可以用 -date= 选项定制格式)

%ar 作者修订日期，按多久以前的方式显示

%cn 提交者(committer)的名字

%ce 提交者的电子邮件地址

%cd 提交日期

%cr 提交日期，按多久以前的方式显示

%s 提交说明

注：作者是指最后一次修改文件的人；而提交者是指提交该文件的人。

$ git log --pretty=format:"%an %ae %ad %cn %ce %cd %cr %s" --graph

--mergs - 查看所有合并过的提交历史记录

--no-merges - 查看所有未被合并过的提交信息

--author=someonet - 查询指定作者的提交记录

$ git log --author=maxsu

--since，--affter - 仅显示指定时间之后的提交(不包含当前日期)

--until，--before - 仅显示指定时间之前的提交(包含当前日期)

$ git log --before={3,weeks,ago} --after={2018-04-18}

### git stash命令

git stash命令用于将更改储藏在脏工作目录中。

1. **使用语法**

git stash list [<options>]

git stash show [<stash>]

git stash drop [-q|--quiet] [<stash>]

git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]

git stash branch <branchname> [<stash>]

git stash save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]

[-u|--include-untracked] [-a|--all] [<message>]

git stash [push [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]

[-u|--include-untracked] [-a|--all] [-m|--message <message>]]

[--] [<pathspec>…​]]

git stash clear

git stash create [<message>]

git stash store [-m|--message <message>] [-q|--quiet] <commit>

1. **描述**

当要记录工作目录和索引的当前状态，但想要返回到干净的工作目录时，则使用git stash。 该命令保存本地修改，并恢复工作目录以匹配HEAD提交。

这个命令所储藏的修改可以使用git stash list列出，使用git stash show进行检查，并使用git stash apply恢复(可能在不同的提交之上)。调用没有任何参数的git stash相当于git stash save。 默认情况下，储藏列表为“分支名称上的WIP”，但您可以在创建一个消息时在命令行上给出更具描述性的消息。

创建的最新储藏存储在refs/stash中; 这个引用的反垃圾邮件中会发现较旧的垃圾邮件，并且可以使用通常的reflog语法命名(例如，stash@{0}是最近创建的垃圾邮件，stash@{1}是stash@{2.hours.ago}之前也是可能的)。也可以通过指定存储空间索引(例如整数n相当于储藏stash@{n})来引用锁存。

1. **示例**

以下是一些示例

1. **拉取到一棵肮脏的树**

当你处于某种状态的时候，你会发现有一些上游的变化可能与正在做的事情有关。当您的本地更改不会与上游的更改冲突时，简单的git pull将让您向前。

但是，有些情况下，本地更改与上游更改相冲突，git pull拒绝覆盖您的更改。 在这种情况下，您可以将更改隐藏起来，执行git pull，然后解压缩，如下所示：

git pull

...

file foobar not up to date, cannot merge.

$ git stash

$ git pull

$ git stash pop

1. **工作流中断**

当你处于某种状态的时候，比如你的老板进来，要求立即开会或处理非常紧急的事务。 传统上，应该提交一个临时分支来存储您的更改，并返回到原始(original)分支进行紧急修复，如下所示：

# ... hack hack hack ...

$ git checkout -b my\_wip

$ git commit -a -m "WIP"

$ git checkout master

$ edit emergency fix # 编辑内容

$ git commit -a -m "Fix in a hurry"

$ git checkout my\_wip

$ git reset --soft HEAD^

# ... continue hacking ...

上面过程可以使用git stash来简化上述操作，如下所示：

# ... hack hack hack ...

$ git stash

$ edit emergency fix

$ git commit -a -m "Fix in a hurry"

$ git stash pop

# ... continue hacking ...

1. **测试部分提交**

当要从工作树中的更改中提交两个或多个提交时，可以使用git stash save --keep-index，并且要在提交之前测试每个更改：

# ... hack hack hack ...

$ git add --patch foo # add just first part to the index

$ git stash save --keep-index # save all other changes to the stash

$ edit/build/test first part

$ git commit -m 'First part' # commit fully tested change

$ git stash pop # prepare to work on all other changes

# ... repeat above five steps until one commit remains ...

$ edit/build/test remaining parts

$ git commit foo -m 'Remaining parts'

1. **恢复被错误地清除/丢弃的垃圾**

如果你错误地删除或清除了垃圾，就不能通过正常的安全机制来恢复。 但是，您可以尝试以下命令来获取仍在存储库中但仍无法访问的隐藏列表：

git fsck --unreachable |

grep commit | cut -d\ -f3 |

xargs git log --merges --no-walk --grep=WIP

1. **储藏你的工作**

为了演示这一功能，可以进入你的项目，在一些文件上进行工作，有可能还暂存其中一个变更。如果运行git status，可以看到你的中间状态：

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

现在你想要切换分支，但是还不想提交正在进行中的工作；所以储藏这些变更为了往堆栈推送一个新的储藏，只要运行 git stash：

$ git stash

Saved working directory and index state \

"WIP on master: 049d078 added the index file"

HEAD is now at 049d078 added the index file

(To restore them type "git stash apply")

现在，工作目录就干净了：

$ git status

# On branch master

nothing to commit, working directory clean

这时，可以方便地切换到其他分支工作；变更都保存在栈上。要查看现有的储藏，可以使用 git stash list，如下所示

$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file\_size"

stash@{2}: WIP on master: 21d80a5 added number to log

在这个案例中，之前已经进行了两次储藏，所以你可以访问到三个不同的储藏。你可以重新应用你刚刚实施的储藏，所采用的命令就是之前在原始的 stash 命令的帮助输出里提示的：git stash apply。如果你想应用更早的储藏，可以通过名字指定它，像这样：git stash apply stash@{2}。如果不指明，Git 默认使用最近的储藏并尝试应用它：

$ git stash apply

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: index.html

# modified: lib/simplegit.rb

#

对文件的变更被重新应用，但是被暂存的文件没有重新被暂存。想那样的话，必须在运行 git stash apply 命令时带上一个 --index 的选项来告诉命令重新应用被暂存的变更。如果是这么做的，应该已经回到原来的位置：

$ git stash apply --index

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

apply 选项只尝试应用储藏的工作——储藏的内容仍然在栈上。要移除它，可以运行 git stash drop 再加上希望移除的储藏的名字：

$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file\_size"

stash@{2}: WIP on master: 21d80a5 added number to log

$ git stash drop stash@{0}

Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)

也可以运行 git stash pop 来重新应用储藏，同时立刻将其从堆栈中移走。

1. **取消储藏**

在某些情况下，可能想应用储藏的修改，在进行了一些其他的修改后，又要取消之前所应用储藏的修改。Git没有提供类似于 stash unapply 的命令，但是可以通过取消该储藏的补丁达到同样的效果：

$ git stash show -p stash@{0} | git apply -R

同样的，如果沒有指定具体的某个储藏，Git 会选择最近的储藏：

$ git stash show -p | git apply -R

可能会想要新建一个別名，在你的 Git 里增加一个 stash-unapply 命令，这样更有效率。例如：

$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'

$ git stash apply

$ #... work work work

$ git stash-unapply

1. **从储藏中创建分支**

如果储藏了一些工作，暂时不去理会，然后继续在你储藏工作的分支上工作，在重新应用工作时可能会碰到一些问题。如果尝试应用的变更是针对一个在那之后修改过的文件，会碰到一个归并冲突并且必须去化解它。如果你想用更方便的方法来重新检验储藏的变更，可以运行 git stash branch，这会创建一个新的分支，检出储藏工作时的所处的提交，重新应用你的工作，如果成功，将会丢弃储藏。

$ git stash branch testchanges

Switched to a new branch "testchanges"

# On branch testchanges

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)

### git tag命令

git tag命令用于创建，列出，删除或验证使用GPG签名的标签对象。同大多数 VCS 一样，Git 也可以对某一时间点上的版本打上标签。人们在发布某个软件版本(比如 v1.0 等等)的时候，经常这么做。本节我们一起来学习如何列出所有可用的标签，如何新建标签，以及各种不同类型标签之间的差别。

1. **使用语法**

git tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <file>]

<tagname> [<commit> | <object>]

git tag -d <tagname>…

git tag [-n[<num>]] -l [--contains <commit>] [--no-contains <commit>]

[--points-at <object>] [--column[=<options>] | --no-column]

[--create-reflog] [--sort=<key>] [--format=<format>]

[--[no-]merged [<commit>]] [<pattern>…]

git tag -v [--format=<format>] <tagname>…

1. **描述**

在refs/tags/中添加标签引用，除非提供了-d/-l/-v来删除，列出或验证标签。

tag 用于创建一个标签 用于在开发阶段，某个阶段的完成，创建一个版本，在开发中都会使用到, 可以创建一个tag来指向软件开发中的一个关键时期，比如版本号更新的时候可以建一个version1.0, version1.2之类的标签，这样在以后回顾的时候会比较方便。tag的使用很简单。

除非指定-f选项，否则不能创建已经存在的标签。

如果传递了-a，-s或-u <keyid>中的一个，该命令将创建一个标签对象，并且需要一个标签消息。 除非-m <msg>或-F <file>，否则将启动一个编辑器，供用户输入标签消息。

1. **示例**

以下是一些示例

1. **列显已有的标签**

列出现有标签的命令非常简单，直接运行 git tag 即可：

$ $ git tag

v1.0

v1.2

显示的标签按字母顺序排列，所以标签的先后并不表示重要程度的轻重。

我们可以用特定的搜索模式列出符合条件的标签。在 Git 自身项目仓库中，有着超过 240 个标签，如果只对 1.4.2 系列的版本感兴趣，可以运行下面的命令：

$ git tag -l 'v1.4.2.\*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

v1.4.2.4

1. **创建标签**

Git 使用的标签有两种类型：轻量级的(lightweight)和含附注的(annotated)。轻量级标签就像是个不会变化的分支，实际上它就是个指向特定提交对象的引用。而含附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 GNU Privacy Guard (GPG) 来签署或验证。一般我们都建议使用含附注型的标签，以便保留相关信息；当然，如果只是临时性加注标签，或者不需要旁注额外信息，用轻量级标签也没问题。

创建一个含附注类型的标签非常简单，用 -a (译注：取 annotated 的首字母)指定标签名字即可：

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

而 -m 选项则指定了对应的标签说明，Git 会将此说明一同保存在标签对象中。如果没有给出该选项，Git 会启动文本编辑软件供你输入标签说明。

可以使用 git show 命令查看相应标签的版本信息，并连同显示打标签时的提交对象。

$ git show v1.4

tag v1.4

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

我们可以看到在提交对象信息上面，列出了此标签的提交者和提交时间，以及相应的标签说明。

1. **签署标签**

如果你有自己的私钥，还可以用 GPG 来签署标签，只需要把之前的选项 -a 改为 -s (译注： 取 signed 的首字母)即可：

$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gee-mail.com>"

1024-bit DSA key, ID F721C45A, created 2009-02-09

现在再运行 git show 会看到对应的 GPG 签名也附在其内：

$ git show v1.5

tag v1.5

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN

Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

=WryJ

-----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

1. **删除标签**

很简单，比如想要名称删除名称为：v1.0的标签，可以执行以下操作：

$ git tag -d v1.0

1. **轻量级标签**

轻量级标签实际上就是一个保存着对应提交对象的校验和信息的文件。要创建这样的标签，一个 -a，-s 或 -m 选项都不用，直接给出标签名字即可：

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

现在运行 git show 查看此标签信息，就只有相应的提交对象摘要：

$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

1. **验证标签**

可以使用 git tag -v [tag-name] (译注：取 verify 的首字母)的方式验证已经签署的标签。此命令会调用 GPG 来验证签名，所以你需要有签署者的公钥，存放在 keyring 中，才能验证：

$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61

type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

若是没有签署者的公钥，会报告类似下面这样的错误：

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'

1. **后期加注标签**

甚至可以在后期对早先的某次提交加注标签。比如在下面展示的提交历史中：

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

我们忘了在提交 “updated rakefile” 后为此项目打上版本号 v1.2，没关系，现在也能做。只要在打标签的时候跟上对应提交对象的校验和(或前几位字符)即可：

$ git tag -a v1.2 9fceb02

现在，可以看到我们已经补上了标签：

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

1. **分享标签**

默认情况下，git push 并不会把标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。其命令格式如同推送分支，运行 git push origin [tagname] 即可：

$ git push origin v1.5

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

如果要一次推送所有本地新增的标签上去，可以使用 --tags 选项：

$ git push origin --tags

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v0.1 -> v0.1

\* [new tag] v1.2 -> v1.2

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

\* [new tag] v1.5 -> v1.5

现在，其他人克隆共享仓库或拉取数据同步后，也会看到这些标签。

### git fetch命令

git fetch命令用于从另一个存储库下载对象和引用。

1. **使用语法**

git fetch [<options>] [<repository> [<refspec>…]]

git fetch [<options>] <group>

git fetch --multiple [<options>] [(<repository> | <group>)…]

git fetch --all [<options>]

1. **描述**

从一个或多个其他存储库中获取分支和/或标签(统称为“引用”)以及完成其历史所必需的对象。 远程跟踪分支已更新(Git术语叫做commit)，需要将这些更新取回本地，这时就要用到git fetch命令。

默认情况下，还会获取指向正在获取的历史记录的任何标签; 效果是获取指向您感兴趣的分支的标签。可以使用--tags或--no-tags选项或通过配置远程.<name>.tagOpt 来更改此默认行为。 通过使用显式提取标签的refspec，可以获取不指向您感兴趣的分支的标签。

git fetch可以从单个命名的存储库或URL中获取，也可以从多个存储库中获取，如果给定了<group>，并且配置文件中有一个远程<group>条目。

获取的参考名称以及它们所指向的对象名称被写入到.git/FETCH\_HEAD中。 此信息可能由脚本或其他git命令使用，如git-pull。

1. **示例**

以下是一些示例

1. **更新远程跟踪分支**

$ git fetch origin

上述命令从远程refs/heads/命名空间复制所有分支，并将它们存储到本地的refs/remotes/ origin/命名空间中，除非使用分支.<name>.fetch选项来指定非默认的refspec。

1. **明确使用refspec**

$ git fetch origin +pu:pu maint:tmp

此更新(或根据需要创建)通过从远程存储库的分支(分别)pu和maint提取来分支本地存储库中的pu和tmp。

即使没有快进，pu分支将被更新，因为它的前缀是加号; tmp不会。

1. **在远程分支上窥视，无需在本地存储库中配置远程**

$ git fetch git://git.kernel.org/pub/scm/git/git.git maint

$ git log FETCH\_HEAD

第一个命令从 git://git.kernel.org/pub/scm/git/git.git 从存储库中获取maint分支，第二个命令使用FETCH\_HEAD来检查具有git-log的分支。

1. **将某个远程主机的更新**

$ git fetch <远程主机名>

要更新所有分支，命令可以简写为：

$ git fetch

上面命令将某个远程主机的更新，全部取回本地。默认情况下，git fetch取回所有分支的更新。如果只想取回特定分支的更新，可以指定分支名,如下所示

$ git fetch <远程主机名> <分支名>

比如，取回origin主机的master分支。

$ git fetch origin master

所取回的更新，在本地主机上要用”远程主机名/分支名”的形式读取。比如origin主机的master分支，就可以用origin/master读取。

git branch命令的-r选项，可以用来查看远程分支，-a选项查看所有分支。

$ git branch -r

origin/master

$ git branch -a

\* master

remotes/origin/master

上面命令表示，本地主机的当前分支是master，远程分支是origin/master。

取回远程主机的更新以后，可以在它的基础上，使用git checkout命令创建一个新的分支。

$ git checkout -b newBrach origin/master

上面命令表示，在origin/master的基础上，创建一个新分支:newBrach。

此外，也可以使用git merge命令或者git rebase命令，在本地分支上合并远程分支。

$ git merge origin/master

# 或者

$ git rebase origin/master

上面命令表示在当前分支上，合并origin/master。

### git pull命令

git pull命令用于从另一个存储库或本地分支获取并集成(整合)。git pull命令的作用是：取回远程主机某个分支的更新，再与本地的指定分支合并，它的完整格式稍稍有点复杂。

1. **使用语法**

git pull [options] [<repository> [<refspec>…]]

1. **描述**

将远程存储库中的更改合并到当前分支中。在默认模式下，git pull是git fetch后跟git merge FETCH\_HEAD的缩写。

更准确地说，git pull使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前分支中。 使用--rebase，它运行git rebase而不是git merge。

1. **示例**

以下是一些示例

$ git pull <远程主机名> <远程分支名>:<本地分支名>

比如，要取回origin主机的next分支，与本地的master分支合并，需要写成下面这样

$ git pull origin next:master

如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略。上面命令可以简写为：

$ git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

$ git fetch origin

$ git merge origin/next

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系(tracking)。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动”追踪”origin/master分支。

Git也允许手动建立追踪关系。

$ git branch --set-upstream master origin/next

上面命令指定master分支追踪origin/next分支。

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

$ git pull origin

上面命令表示，本地的当前分支自动与对应的origin主机”追踪分支”(remote-tracking branch)进行合并。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

$ git pull

上面命令表示，当前分支自动与唯一一个追踪分支进行合并。

如果合并需要采用rebase模式，可以使用–rebase选项。

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

**git fetch和git pull的区别**

1. git fetch：相当于是从远程获取最新版本到本地，不会自动合并。

$ git fetch origin master

$ git log -p master..origin/master

$ git merge origin/master

以上命令的含义：

首先从远程的origin的master主分支下载最新的版本到origin/master分支上

然后比较本地的master分支和origin/master分支的差别

最后进行合并

上述过程其实可以用以下更清晰的方式来进行：

$ git fetch origin master:tmp

$ git diff tmp

$ git merge tmp

1. git pull：相当于是从远程获取最新版本并merge到本地

git pull origin master

上述命令其实相当于git fetch 和 git merge

在实际使用中，git fetch更安全一些，因为在merge前，我们可以查看更新情况，然后再决定是否合并。

### git push命令

git push命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相似。

$ git push <远程主机名> <本地分支名>:<远程分支名>

1. **使用语法**

git push [--all | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--receive-pack=<git-receive-pack>]

[--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-v | --verbose]

[-u | --set-upstream] [--push-option=<string>]

[--[no-]signed|--sign=(true|false|if-asked)]

[--force-with-lease[=<refname>[:<expect>]]]

[--no-verify] [<repository> [<refspec>…]]

1. **描述**

使用本地引用更新远程引用，同时发送完成给定引用所需的对象。可以在每次推入存储库时，通过在那里设置挂钩触发一些事件。

当命令行不指定使用<repository>参数推送的位置时，将查询当前分支的branch.\*.remote配置以确定要在哪里推送。 如果配置丢失，则默认为origin。

1. **示例**

以下是一些示例

$ git push origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果master不存在，则会被新建。

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

$ git push origin :master

# 等同于

$ git push origin --delete master

上面命令表示删除origin主机的master分支。如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push -u origin master

上面命令将本地的master分支推送到origin主机，同时指定origin为默认主机，后面就可以不加任何参数使用git push了。

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。

$ git config --global push.default matching

# 或者

$ git config --global push.default simple

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用–all选项。

$ git push --all origin

上面命令表示，将所有本地分支都推送到origin主机。

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用–force选项。

$ git push --force origin

上面命令使用-–force选项，结果导致在远程主机产生一个”非直进式”的合并(non-fast-forward merge)。除非你很确定要这样做，否则应该尽量避免使用–-force选项。

最后，git push不会推送标签(tag)，除非使用–tags选项。

$ git push origin --tags

将当前分支推送到远程的同名的简单方法，如下

$ git push origin HEAD

将当前分支推送到源存储库中的远程引用匹配主机。 这种形式方便推送当前分支，而不考虑其本地名称。如下

$ git push origin HEAD:master

**其它示例**

1. 推送本地分支lbranch-1到新大远程分支rbranch-1：

$ git push origin lbranch-1:refs/rbranch-1

1. 推送lbranch-2到已有的rbranch-1，用于补充rbranch-1：

$ git checkout lbranch-2

$ git rebase rbranch-1

$ git push origin lbranch-2:refs/rbranch-1

1. 用本地分支lbranch-3覆盖远程分支rbranch-1：

$ git push -f origin lbranch-2:refs/rbranch-1

或者

$ git push origin :refs/rbranch-1 //删除远程的rbranch-1分支

$ git push origin lbranch-1:refs/rbranch-1

1. 查看push的结果

$ gitk rbranch-1

1. 推送tag

$ git push origin tag\_name

1. 删除远程标签

$ git push origin :tag\_name

### git remote命令

git remote命令管理一组跟踪的存储库。

要参与任何一个 Git 项目的协作，必须要了解该如何管理远程仓库。远程仓库是指托管在网络上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写。同他人协作开发某 个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。管理远程仓库的工作，包括添加远程库，移除废弃的远程库，管理各式远程库分支，定义是否跟踪这些分支等等。

1. **使用语法**

git remote [-v | --verbose]

git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>

git remote rename <old> <new>

git remote remove <name>

git remote set-head <name> (-a | --auto | -d | --delete | <branch>)

git remote set-branches [--add] <name> <branch>…​

git remote get-url [--push] [--all] <name>

git remote set-url [--push] <name> <newurl> [<oldurl>]

git remote set-url --add [--push] <name> <newurl>

git remote set-url --delete [--push] <name> <url>

git remote [-v | --verbose] show [-n] <name>…​

git remote prune [-n | --dry-run] <name>…​

git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…​]

1. **描述**

git remote命令管理一组跟踪的存储库。

1. **示例**

以下是一些示例

1. 查看当前的远程库

要查看当前配置有哪些远程仓库,可以用 git remote 命令,它会列出每个远程库的简短名字.在克隆完某个项目后,至少可以看到一个名为 origin 的远程库, git 默认使用这个名字来标识你所克隆的原始仓库:

$ git clone http://git.oschina.net/yiibai/sample.git

$ cd sample

git remote 不带参数，列出已经存在的远程分支

$ git remote

origin

git remote -v | --verbose 列出详细信息，在每一个名字后面列出其远程url

此时， -v 选项(译注:此为 –verbose 的简写,取首字母),显示对应的克隆地址:

$ git remote -v

origin http://git.oschina.net/yiibai/sample.git (fetch)

origin http://git.oschina.net/yiibai/sample.git (push)

Administrator@MY-PC /D/worksp/sample (master)

$ git remote --verbose

origin http://git.oschina.net/yiibai/sample.git (fetch)

origin http://git.oschina.net/yiibai/sample.git (push)

1. 添加一个新的远程，抓取，并从它检出一个分支

$ git remote

origin

$ git branch -r

origin/HEAD -> origin/master

origin/master

$ git remote add staging git://git.kernel.org/.../gregkh/staging.git

$ git remote

origin

staging

$ git fetch staging

...

From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging

\* [new branch] master -> staging/master

\* [new branch] staging-linus -> staging/staging-linus

\* [new branch] staging-next -> staging/staging-next

$ git branch -r

origin/HEAD -> origin/master

origin/master

staging/master

staging/staging-linus

staging/staging-next

$ git checkout -b staging staging/master

...

1. 添加远程仓库

要添加一个新的远程仓库,可以指定一个简单的名字,以便将来引用,运行 git remote add [shortname] [url]:

$ git remote

　　origin

$ git remote add pb http://git.oschina.net/yiibai/sample.git

$ git remote -v origin http://git.oschina.net/yiibai/sample.git

　　pb http://git.oschina.net/yiibai/sample2.git 现在可以用字串 pb 指代对应的仓库地址了.比如说,要抓取所有 Paul 有的,但本地仓库没有的信息,可以运行 git fetch pb:

$ git fetch pb

　　remote: Counting objects: 58, done.

　　remote: Compressing objects: 100% (41/41), done.

　　remote: Total 44 (delta 24), reused 1 (delta 0)

　　Unpacking objects: 100% (44/44), done.

　　From http://git.oschina.net/yiibai/sample2.git

　　\* [new branch] master -> pb/master

　　\* [new branch] ticgit -> pb/ticgit

1. 模仿 git clone，但只跟踪选定的分支

$ mkdir project.git

$ cd project.git

$ git init

$ git remote add -f -t master -m master origin git://example.com/git.git/

$ git merge origin

### git submodule命令

git submodule命令用于初始化，更新或检查子模块。

1. **使用语法**

git submodule [--quiet] add [<options>] [--] <repository> [<path>]

git submodule [--quiet] status [--cached] [--recursive] [--] [<path>…​]

git submodule [--quiet] init [--] [<path>…​]

git submodule [--quiet] deinit [-f|--force] (--all|[--] <path>…​)

git submodule [--quiet] update [<options>] [--] [<path>…​]

git submodule [--quiet] summary [<options>] [--] [<path>…​]

git submodule [--quiet] foreach [--recursive] <command>

git submodule [--quiet] sync [--recursive] [--] [<path>…​]

git submodule [--quiet] absorbgitdirs [--] [<path>…​]

1. **使用场景**

基于公司的项目会越来越多，常常需要提取一个公共的类库提供给多个项目使用，但是这个library怎么和git在一起方便管理呢？

我们需要解决下面几个问题：

1. 如何在git项目中导入library库?
2. library库在其他的项目中被修改了可以更新到远程的代码库中?
3. 其他项目如何获取到library库最新的提交?
4. 如何在clone的时候能够自动导入library库?

解决以上问题，可以考虑使用git的 submodule 来解决。

**submodule是什么?**

git submodule 是一个很好的多项目使用共同类库的工具，它允许类库项目做为repository,子项目做为一个单独的git项目存在父项目中，子项目可以有自己的独立的commit，push，pull。而父项目以Submodule的形式包含子项目，父项目可以指定子项目header，父项目中会的提交信息包含Submodule的信息，再clone父项目的时候可以把Submodule初始化。

1. **示例**

以下是一些示例

1. 添加子模块

在本例中，我们将会添加一个名为 “DbConnector” 的库。

$ git submodule add http://github.com/chaconinc/DbConnector

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

默认情况下，子模块会将子项目放到一个与仓库同名的目录中，本例中是 “DbConnector”。 如果你想要放到其他地方，那么可以在命令结尾添加一个不同的路径。

如果这时运行 git status，你会注意到几个东西。

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: .gitmodules

new file: DbConnector

首先应当注意到新的 .gitmodules 文件。 该置文件保存了项目 URL 与已经拉取的本地目录之间的映射：

$ cat .gitmodules

[submodule "DbConnector"]

path = DbConnector

url = http://github.com/chaconinc/DbConnector

如果有多个子模块，该文件中就会有多条记录。 要重点注意的是，该文件也像 .gitignore 文件一样受到(通过)版本控制。 它会和该项目的其他部分一同被拉取推送。 这就是克隆该项目的人知道去哪获得子模块的原因。

在 git status 输出中列出的另一个是项目文件夹记录。 如果你运行 git diff，会看到类似下面的信息：

$ git diff --cached DbConnector

diff --git a/DbConnector b/DbConnector

new file mode 160000

index 0000000..c3f01dc

--- /dev/null

+++ b/DbConnector

@@ -0,0 +1 @@

+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc

虽然 DbConnector 是工作目录中的一个子目录，但 Git 还是会将它视作一个子模块。当你不在那个目录中时，Git 并不会跟踪它的内容， 而是将它看作该仓库中的一个特殊提交。

如果你想看到更漂亮的差异输出，可以给 git diff 传递 --submodule 选项。

$ git diff --cached --submodule

diff --git a/.gitmodules b/.gitmodules

new file mode 100644

index 0000000..71fc376

--- /dev/null

+++ b/.gitmodules

@@ -0,0 +1,3 @@

+[submodule "DbConnector"]

+ path = DbConnector

+ url = http://github.com/chaconinc/DbConnector

Submodule DbConnector 0000000...c3f01dc (new submodule)

1. 克隆含有子模块的项目

接下来我们将会克隆一个含有子模块的项目。 当你在克隆这样的项目时，默认会包含该子模块目录，但其中还没有任何文件：

$ git clone http://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

$ cd MainProject

$ ls -la

total 16

drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .

drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..

drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git

-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules

drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector

-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile

drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src

$ cd DbConnector/

$ ls

$

其中有 DbConnector 目录，不过是空的。 你必须运行两个命令：git submodule init 用来初始化本地配置文件，而 git submodule update 则从该项目中抓取所有数据并检出父项目中列出的合适的提交。

$ git submodule init

Submodule 'DbConnector' (http://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

现在 DbConnector 子目录是处在和之前提交时相同的状态了。

不过还有更简单一点的方式。 如果给 git clone 命令传递 --recursive 选项，它就会自动初始化并更新仓库中的每一个子模块。

$ git clone --recursive http://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

Submodule 'DbConnector' (http://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

1. 删除Submodule

git 并不支持直接删除Submodule需要手动删除对应的文件:

cd pod-project

git rm --cached pod-library

rm -rf pod-library

rm .gitmodules

更改git的配置文件config:

vim .git/config

可以看到Submodule的配置信息：

[submodule "pod-library"]

url = git@github.com:jjz/pod-library.git

删除submodule相关的内容,然后提交到远程服务器:

git commit -a -m 'remove pod-library submodule'

### git show命令

git show命令用于显示各种类型的对象。

1. **使用语法**

git show [options] <object>…​

1. **描述**

显示一个或多个对象(blobs，树，标签和提交)。

对于提交，它显示日志消息和文本差异。 它还以git diff-tree --cc生成的特殊格式呈现合并提交。

对于标签，它显示标签消息和引用对象。

对于树，它显示的名称(相当于使用git ls-tree和--name-only选项)。

对于简单的blobs，它显示了普通的内容。

该命令采用适用于git diff-tree命令的选项来控制如何显示提交引入的更改。

1. **示例**

以下是一些示例

1. 显示标签v1.0.0，以及标签指向的对象

$ git show v1.0.0

1. 显示标签v1.0.0指向的树

$ git show v1.0.0^{tree}

1. 显示标签v1.0.0指向的提交的主题

$ git show -s --format=%s v1.0.0^{commit}

1. 显示 Documentation/README 文件的内容，它们是 next 分支的第10次最后一次提交的内容

$ git show next~10:Documentation/README

1. 将Makefile的内容连接到分支主控的头部

$ git show master:Makefile master:t/Makefile

### git shortlog命令

git shortlog命令用于汇总git日志输出。

1. **使用语法**

git log --pretty=short | git shortlog [<options>]

git shortlog [<options>] [<revision range>] [[\--] <path>…​]

1. **描述**

适当包含在发布公告中的格式汇总git日志输出。每个提交将按作者和标题分组。

另外，“[PATCH]”将从提交描述中删除。

如果在命令行上没有传递修订版本，并且标准输入不是终端或没有当前的分支，则git shortlog将输出从标准输入读取的日志的摘要，而不引用当前存储库。

**场景假设**

一个开发小组有10个程序员，他们用 Git 做版本控制，某一天程序员A push了当天的几个commit之后，突然在想“我在这个项目到底一共进行过多少次commit？谁比我commit更多？多多少？谁是组里面进行最多 commit的？谁是最少的？”

Git 非常人性化地支持这样一个命令：

$ git shortlog

这个命令会返回这个 git repository 底下每个用户进行 commit 的次数，以及每次 commit 的注释。

1. -s 参数省略每次 commit 的注释，仅仅返回一个简单的统计。
2. -n 参数按照 commit 数量从多到少的顺利对用户进行排序
3. **示例**

以下是一些示例

$ git shortlog -s -n

135 Tom Preston-Werner

15 Jack Danger Canty

10 Chris Van Pelt

7 Mark Reid

6 remi

3 Mikael Lind

3 Toby DiPasquale

2 Aristotle Pagaltzis

2 Basil Shkara

2 John Reilly

2 PJ Hyett

1 Marc Chung

1 Nick Gerakines

1 Nick Quaranto

1 Tom Kirchner

比如想要知道一个开源项目(例如 Graphiti )的 commit 统计

$ git shortlog -s -n

16 maxsu

5 your\_name

2 minsu

1 Maxsu

上面的结果表明， maxsu 确实是这个开源项目的主要开发者。

### git describe命令

git describe命令显示离当前提交最近的标签。

1. **使用语法**

git describe [--all] [--tags] [--contains] [--abbrev=<n>] [<commit-ish>…​]

git describe [--all] [--tags] [--contains] [--abbrev=<n>] --dirty[=<mark>]

1. **描述**

该命令查找从提交可访问的最新标记。 如果标签指向提交，则只显示标签。 否则，它将标记名称与标记对象之上的其他提交数量以及最近提交的缩写对象名称后缀。

默认情况下(不包括--all或--tags)git描述只显示注释标签。

1. **示例**

如果符合条件的tag指向最新提交则只是显示tag的名字，否则会有相关的后缀来描述该tag之后有多少次提交以及最新的提交commit id。不加任何参数的情况下，git describe 只会列出带有注释的tag

$ git describe --tags

tag1-2-g026498b

2:表示自打tag tag1 以来有2次提交(commit)

g026498b：g 为git的缩写，在多种管理工具并存的环境中很有用处；

### git rebase命令

git rebase命令在另一个分支基础之上重新应用，用于把一个分支的修改合并到当前分支。

1. **使用语法**

git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]

[<upstream> [<branch>]]

git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]

--root [<branch>]

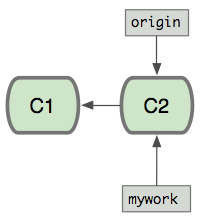
git rebase --continue | --skip | --abort | --quit | --edit-todo

1. **示例**

假设你现在基于远程分支”origin“，创建一个叫”mywork“的分支。

$ git checkout -b mywork origin

结果如下所示



现在我们在这个分支(mywork)做一些修改，然后生成两个提交(commit).

$ vi file.txt

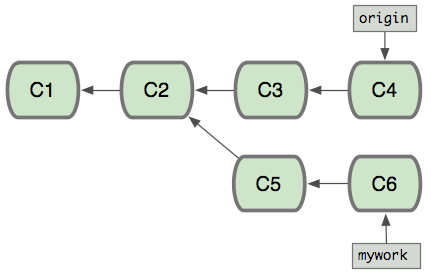
$ git commit

$ vi otherfile.txt

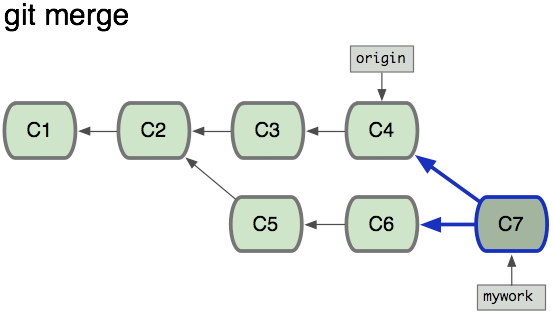
$ git commit

... ...

但是与此同时，有些人也在”origin“分支上做了一些修改并且做了提交了，这就意味着”origin“和”mywork“这两个分支各自”前进”了，它们之间”分叉”了。



在这里，你可以用”pull“命令把”origin“分支上的修改拉下来并且和你的修改合并； 结果看起来就像一个新的”合并的提交”(merge commit):

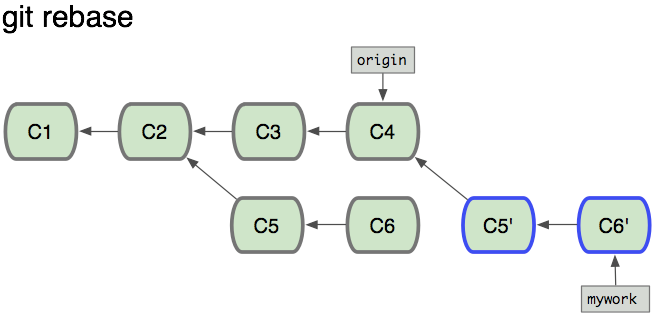


但是，如果你想让”mywork“分支历史看起来像没有经过任何合并一样，也可以用 git rebase，如下所示:

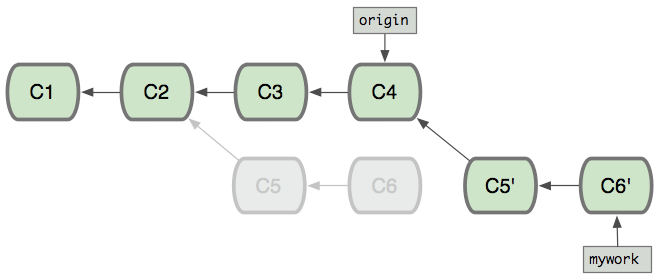
$ git checkout mywork

$ git rebase origin

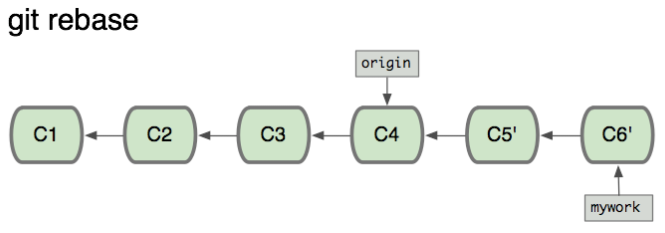
这些命令会把你的”mywork“分支里的每个提交(commit)取消掉，并且把它们临时 保存为补丁(patch)(这些补丁放到”.git/rebase“目录中),然后把”mywork“分支更新 到最新的”origin“分支，最后把保存的这些补丁应用到”mywork“分支上。

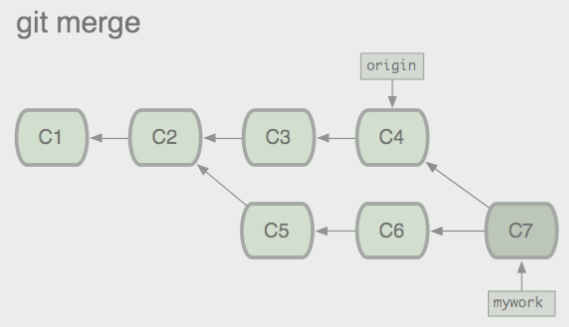


当’mywork‘分支更新之后，它会指向这些新创建的提交(commit),而那些老的提交会被丢弃。 如果运行垃圾收集命令(pruning garbage collection), 这些被丢弃的提交就会删除.



现在我们可以看一下用合并(merge)和用rebase所产生的历史的区别：





在rebase的过程中，也许会出现冲突(conflict)。在这种情况，Git会停止rebase并会让你去解决冲突；在解决完冲突后，用”git add“命令去更新这些内容的索引(index), 然后，你无需执行 git commit,只要执行:

$ git rebase --continue

这样git会继续应用(apply)余下的补丁。

在任何时候，可以用--abort参数来终止rebase的操作，并且”mywork“ 分支会回到rebase开始前的状态。

$ git rebase --abort

## Eclipse集成Git

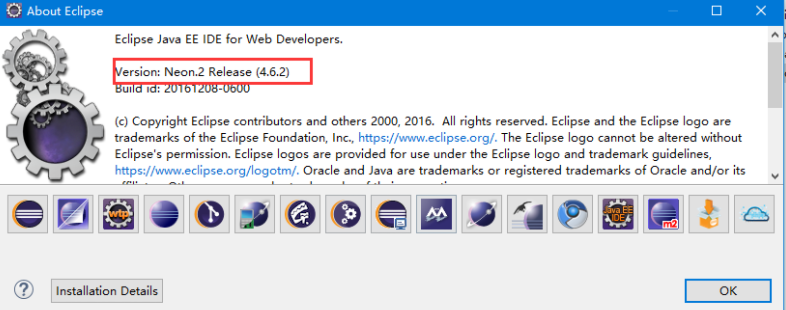
### Eclipse安装配置Git

1. 查看自己eclipse的版本

打开eclipse

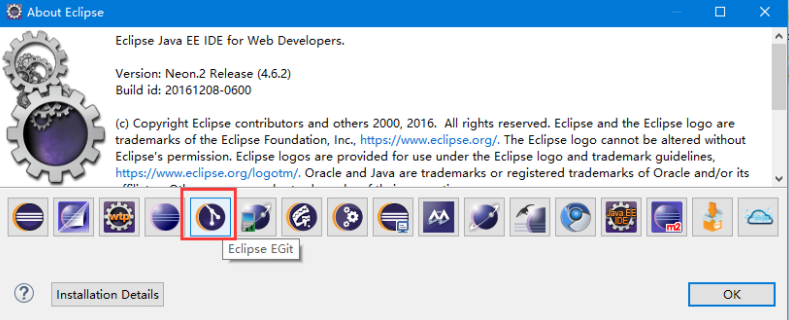
导航: help->AboutEclipse

如图：



1. 检查Eclipse中是否已安装Git插件

菜单栏Help -> AboutEclipse，查看是否有Git插件的图标。



现在的Eclipse下载后一般都提供Git插件了，若是没有Git插件,可以再eclipse 的market里面搜索egit插件安装，如果低版本的eclipse安装高版本的egit插件会报错，建议要么安装低版本的，要么去eclipse官网下载一个最新的eclipse for j2ee，不大，也就200多M。

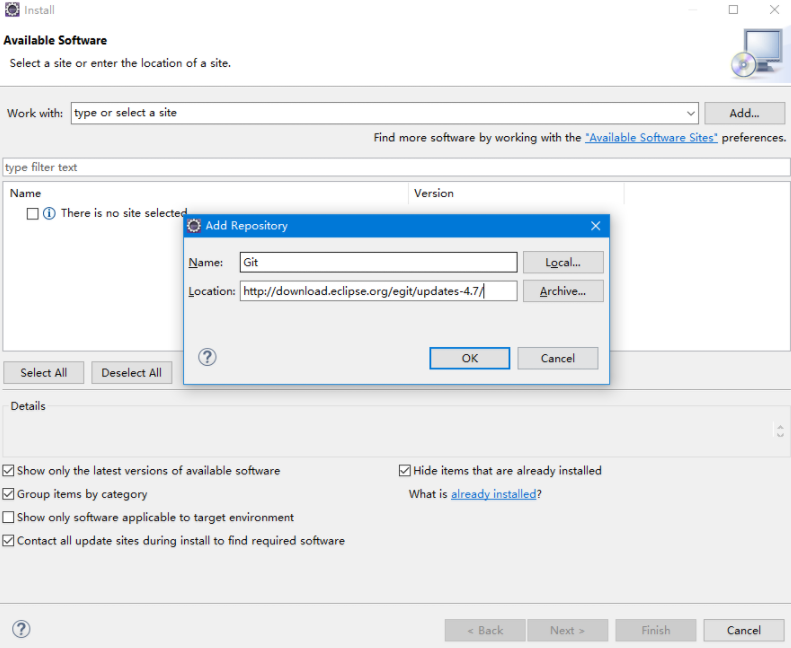
注意：一般market里面搜索到的EGit插件都是适合最新版本的eclipse，所以建议

先打开（http://wiki.eclipse.org/EGit/FAQ#Where\_can\_I\_find\_older\_releases\_of\_EGit.3F）

该网页提供了对应版本的EGit，自己选择相应的版本。

然后Help->Install new software->Add

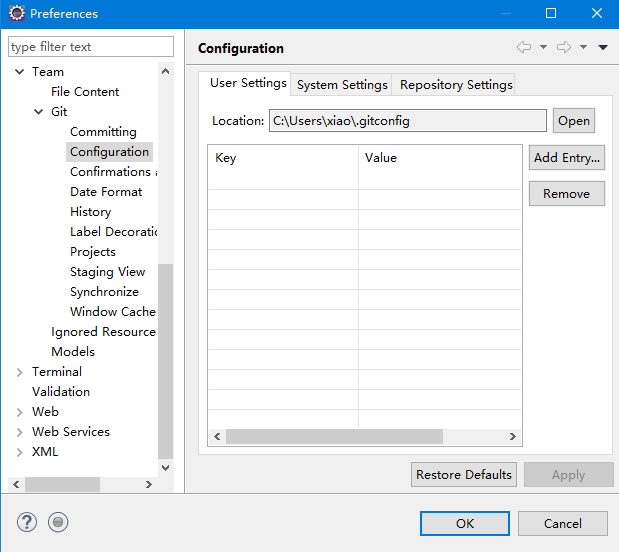
如图：



然后就一直点next就行，安装好之后会叫你重启Eclipse，点OK，git插件就安装好了。

1. **配置Git**

菜单栏Window ->Preferences -> Team -> Git -> Configuration，点击Add Entry…，添加name和email，在将项目代码commit到Git时需要用到name和email信息。在点击AddEntry…弹出的对话框中，Key填user.name，Value填姓名信息，这是添加name信息，在添加email信息时，Key填user.email，Value填email地址。 配置效果如图：



### 关联Gihub账号

1. **登录Github**

进入GitHub网页(http://github.com)，用GitHub帐号登录，若还没有GitHub账号则先注册。

登录GitHub后点击右下角的Newrepository按钮添加项目代码仓库。

输入Repository name（建议与要提交的项目名同名），Description（稍微描述一下此项目是干什么的），选Public（Private要收费，Public免费，不过相应的所发布的项目代码任何人都可以查看得到），最后点击Create repository，项目代码仓库创建完毕。

1. **创建Java项目**

建立Java项目。菜单栏File -> New -> Java Project，输入Project name，此处输入为Java，点击Finish完成。

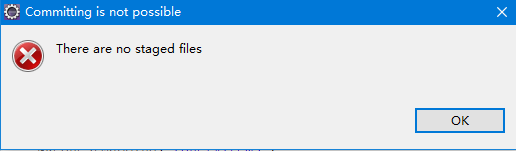
创建class文件。将鼠标移到Java项目src文件夹上，单击鼠标右键，New -> Class，填Package和Name，如果需要包含主方法，可以勾选上public static void main(String[] args)，点击Finish完成。

1. **关联Git**

编写相应的代码 后关联Git。鼠标选中要发布的项目，单击右键，Team -> Share Project，选Git，点击Next。勾选Use or create repository in parent folder ofproject，选中要发布的项目，点击Create Repository，点击Finish完成。

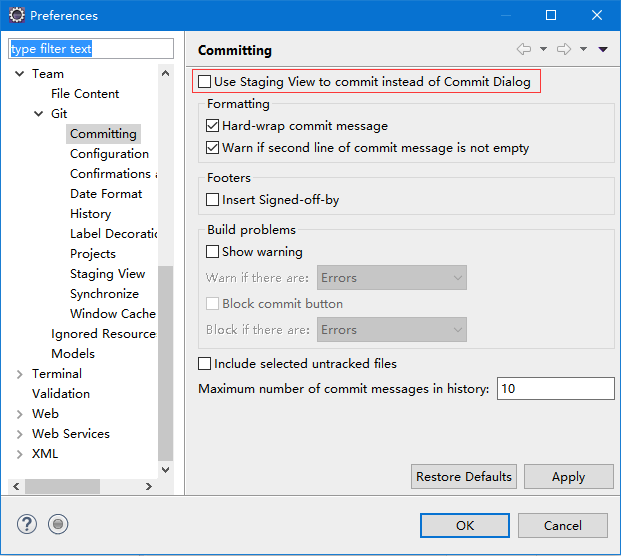
发布项目。鼠标选中要发布的项目，单击右键，Team -> Commit，填写Commitmessage，勾选全选，点击Commit提交。注意：此时，仅是提交在本地Git，还需要Push才能提交到GitHub中。

使用Ecplise git commit时出现"There are no stages files"



**解决方案**

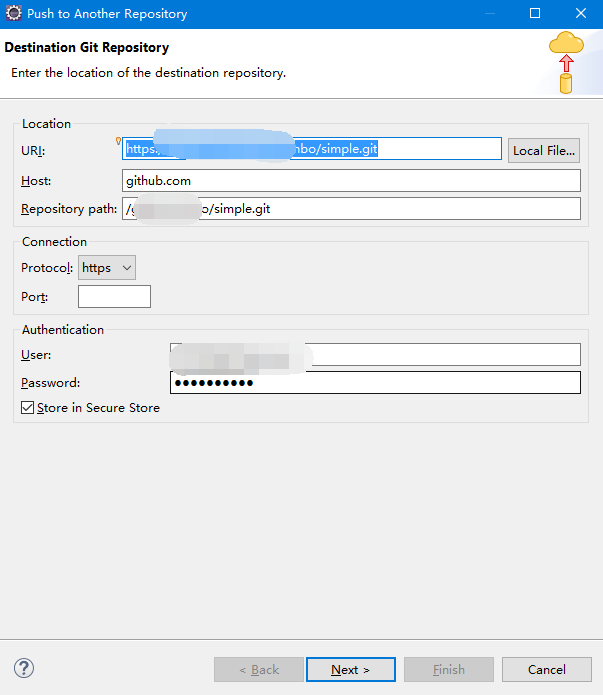
进入Window——Preferences——Team——Git——Committing，反选下图红圈部分：



1. **发布到Github**

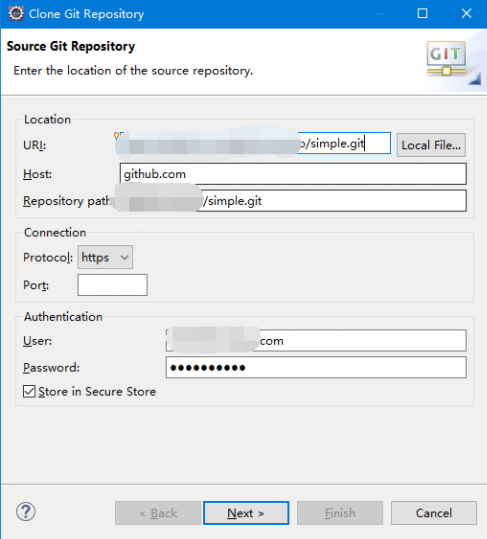
发布到GitHub。鼠标选中要发布的项目，单击右键，Team -> Remote -> Push，粘贴URI，填写User和Password，点击Next；点击Source ref的下拉框，选择mast，点击Add Spec，点击Finish，点击OK完成。

发布后的GitHub项目页面，可以很方便的看到提交的次数，每次提交的备注信息等，非常方便对代码的管理。



1. **Eclipse导入GitHub 项目**

从GitHub中取得项目代码。菜单栏Window -> Show View -> Git -> GitRepositories，点击OK；在Git Repositories视图中，点击图示按钮，进入Clone Git Repository对话框；填写URI，User，Password，点击Next；点击Next；选择要保存的路径，点击Finish。至此，项目代码就被下载到指定目录下了，需要使用的时候通过Eclipse Import进来即可。



# SVN

## SVN概述

Apache的Subversion通常缩写为 SVN，是一个开源许可下分发的软件版本和版本控制系统。Subversion由CollabNet公司创建于2000年，但现在它是Apache软件基金会的一个项目开发，并作为了丰富的开发人员和用户社区的一部分。

## SVN安装(Linux)

### SVN安装

Subversion是流行的开放源码的版本控制工具。 因为它是开放源码，它是一个互联网上免费提供。默认情况下，大多数的GNU/Linux发行，所以它可能是你的系统上已经安装了。要检查它是否安装或不使用下面的命令。

[jerry@CentOS ~]$ svn --version

如果没有安装Subversion客户端，然后命令将报告错误，否则它会显示安装的软件版本。

[jerry@CentOS ~]$ svn --version

-bash: svn: command not found

如果使用的是GNU/Linux基于RPM，然后使用yum命令进行安装。安装成功后，执行的svn - version命令。

[jerry@CentOS ~]$ su -

Password:

[root@CentOS ~]# yum install subversion

[jerry@CentOS ~]$ svn --version

svn, version 1.6.11 (r934486)

compiled Jun 23 2012, 00:44:03

如果使用的是基于Debian GNU/Linux的命令进行安装然后用apt

[jerry@Ubuntu]$ sudo apt-get update

[sudo] password for jerry:

[jerry@Ubuntu]$ sudo apt-get install subversion

[jerry@Ubuntu]$ svn --version

svn, version 1.7.5 (r1336830)

compiled Jun 21 2013, 22:11:49

### Apache安装

我们已经看到了如何在GNU/Linux上安装Subversion客户端。让我们来看看如何创建新的存储库，并允许用户访问。

我们必须在服务器上安装的Apache 的 httpd模块和svnadmin工具。

[jerry@CentOS ~]$ su -

Password:

[root@CentOS ~]# yum install mod\_dav\_svn subversion

mod\_dav\_svn 包允许使用HTTP访问到存储库，从Apache httpd 服务器和颠覆包安装svnadmin 工具。

Subversion版本读取它的配置从 /etc/httpd/conf.d/subversion.conf 文件.，增加配置后 subversion.conf 文件看起来像这样。

LoadModule dav\_svn\_module modules/mod\_dav\_svn.so

LoadModule authz\_svn\_module modules/mod\_authz\_svn.so

<Location /svn>

DAV svn

SVNParentPath /var/www/svn

AuthType Basic

AuthName "Authorization Realm"

AuthUserFile /etc/svn-users

Require valid-user

</Location>

让我们创建Subversion用户，并授予他们访问到存储库。 htpasswd的命令是用来创建和更新纯文本文件，用来存储​​用户HTTP基本身份验证的用户名和密码。 “-c”选项创建密码文件，如密码文件已经存在，它是覆盖。这就是为什么只有第一次使用'-c'选项。 '-m'选项启用MD5加密密码。

**用户设置**

让我们创建用户tom。

[root@CentOS ~]# htpasswd -cm /etc/svn-users tom

New password:

Re-type new password:

Adding password for user tom

让我们创建用户 jerry

[root@CentOS ~]# htpasswd -m /etc/svn-users jerry

New password:

Re-type new password:

Adding password for user jerry

[root@CentOS ~]#

创建Subversion版本父目录来存储所有的工作(详细风 /etc/httpd/conf.d/subversion.conf).

[root@CentOS ~]# mkdir /var/www/svn

[root@CentOS ~]# cd /var/www/svn/

**信息库设置**

创建项目库名为project\_repo。 svnadmin 命令将创建新的存储库，里面存储元数据和其他一些目录。

[root@CentOS svn]# svnadmin create project\_repo

[root@CentOS svn]# ls -l project\_repo

total 24

drwxr-xr-x. 2 root root 4096 Aug 4 22:30 conf

drwxr-sr-x. 6 root root 4096 Aug 4 22:30 db

-r--r--r--. 1 root root 2 Aug 4 22:30 format

drwxr-xr-x. 2 root root 4096 Aug 4 22:30 hooks

drwxr-xr-x. 2 root root 4096 Aug 4 22:30 locks

-rw-r--r--. 1 root root 229 Aug 4 22:30 README.txt

让我们改变存储库的用户和组所有权。

[root@CentOS svn]# chown -R apache.apache project\_repo/

检查是否启用SELinux或不使用SELinux状态工具。

[root@CentOS svn]# sestatus

SELinux status: enabled

SELinuxfs mount: /selinux

Current mode: enforcing

Mode from config file: enforcing

Policy version: 24

Policy from config file: targeted

对于我们的服务器上启用SELinux的，所以我们不得不改变SELinux的安全上下文。

[root@CentOS svn]# chcon -R -t httpd\_sys\_content\_t /var/www/svn/project\_repo/

为了让承诺通过HTTP执行以下命令。

[root@CentOS svn]# chcon -R -t httpd\_sys\_rw\_content\_t /var/www/svn/project\_repo/

重新启动Apache服务器，我们完成配置Apache服务器。

[root@CentOS svn]# service httpd restart

Stopping httpd: [FAILED]

Starting httpd: httpd: apr\_sockaddr\_info\_get() failed for CentOS

httpd: Could not reliably determine the server's fully qualified domain name, using 127.0.0.1 for ServerName

[ OK ]

[root@CentOS svn]# service httpd status

httpd (pid 1372) is running...

[root@CentOS svn]#

我们已经配置成功，现在Apache服务器配置存储库。库访问只有通过身份验证的用户，使用默认的授权文件，添加以下几行到 project\_repo/conf/svnserve.conf 文件

anon-access = none

authz-db = authz

每个Subversion项目通常有 trunk, tags 和 branches 目录项目的根目录下。

树干（trunk ）是一个目录，在所有主要的发展发生，通常是签出开发人员工作的项目。

标签目录是用来存储的项目命名的快照。当创建一个产能释放，团队将标记的代码，进入释放。

当想要追求发展的不同线路的分支目录。

让我们创建的树干（branches），标签和分支的目录结构，根据项目资料库。

[root@CentOS svn]# mkdir /tmp/svn-template

[root@CentOS svn]# mkdir /tmp/svn-template/trunk

[root@CentOS svn]# mkdir /tmp/svn-template/branches

[root@CentOS svn]# mkdir /tmp/svn-template/tags

从目录 /tmp/svn-template 模板导入到存储库。

[root@CentOS svn]# svn import -m 'Create trunk, branches, tags directory structure' /tmp/svn-template/

Adding /tmp/svn-template/trunk

Adding /tmp/svn-template/branches

Adding /tmp/svn-template/tags

Committed revision 1.

[root@CentOS svn]#

完成！ 我们已经创建成功，并允许访问Tom 和Jerry。从现在开始，他们可以执行所有支持的操作到版本库。

## SVN生命周期

### 建立资料库

该库是开发人员存储他们所有的工作一个中央地方。资源库不仅存储文件，而且还变更历史。这意味着它保持历史是谁做的变化，何时和为什么？

创建操作是用来创建一个新的资料库。大部分的时候只有一次操作完成。当创建一个新的存储库，VCS将期待你说的东西，比如，想让它被创建，它的名称应该是什么，以确定它。

### 检出

检出操作用于从资源库中创建一个工作副本。工作副本是一个私有工作场所的开发人员做他们的变化，后来就到存储库提交这些变化

### 更新

顾名思义更新操作是用来更新工作副本。此操作同步与存储库的工作副本。由于共享库是由所有的团队，其他开发人员可以提交他们的修改你的工作副本变成了旧的。

让我们假设Tom 和 Jerry 两个开发人员工作的一个项目。两者检出从库中的最新版本，并开始工作。此时他们的工作拷贝与存储库是完全同步的。杰里完成他的工作非常有效，并提交自己的修改到版本库。

现在，Tom 的工作副本是过时了。从资源库中，更新操作将Jerry 的最新变化，将Tom 的工作副本更新。

### 执行更改

检出后可以做各种操作进行更改。编辑是最常见的操作。一个可以编辑现有的文件，从文件中添加/删除的内容。

人们可以添加文件/目录。但随即这些文件/目录不会成为存储库中的一部分，而不是将它们添加到挂起的更改列表，并成为后提交操作的库中的一部分。

同样，一个可以删除的文件/目录。立即删除操作，删除文件，从工作副本，但实际删除文件添加到挂起的更改列表和操作提交到存储库后所做的更改。

重命名操作更改文件/目录的名称。用于移动从一个地方到另一个存储库树中的文件/目录移动操作。

### 回顾变化

当检出的工作副本或更新工作副本，然后工作副本完全同步与存储库。但是当工作副本的变化，它变得比库新。这是很好的做法，审查更改，然后再提交操作。

状态操作列出了已经取得的工作副本的修改。正如我之前提到的，只要做的工作副本的变化，所有这些变化成为挂起的更改列表的一部分。和状态操作用于查看挂起的更改列表。

状态操作，只提供了一系列的变化，但不是关于他们的详细资料。我们可以使用diff操作已作出的修改的工作副本中查看详细信息。

### 修正错误

让我们假设一个改变自己的工作副本，但现在他要扔掉这些变化。在这种情况下，恢复操作会有所帮助。

恢复操作撤销已作出的工作副本的修改。这是可能的一个或多个文件/目录恢复。也有可能整个工作副本恢复，在这种情况下，恢复操作会破坏挂起的更改列表，并会带来工作副本返回到原来的状态。

### 解决冲突

在合并时，可能会发生冲突。但是，什么是冲突？合并操作自动处理，可以做到安全的一切。一切被视为冲突。例如，如果“hello.c”文件被修改的分支，并在另一个分支删除？这种情况需要一个人作出决定。决心操作是用来帮助用户数字的东西，并告知VCS冲突应如何处理。

### 提交更改

提交操作是用来申请工作副本到版本库的变化。此操作修改库和其他开发人员可以通过更新他们的工作副本，看到这些变化。

之前提交一个挂起的更改列表添加文件/目录。这是改变的地方等待提交。有了提交，我们通常会提供一个日志消息来解释为什么有人做了修改。此日志消息的一部分的历史资料库。提交是一个原子操作。含义的成功，要么全部提交或回滚。用户永远不会看到一半完成提交。

## SVN检出处理

Subversion提供checkout命令检查从库中取出一个工作拷贝。下面的命令将创建新的目录名称project\_repo在当前工作目录。不要理会版本库URL，大部分时间它已经Subversion版本管理员提供适当的访问权限。

[tom@CentOS ~]$ svn checkout http://svn.server.com/svn/project\_repo --username=tom

上面的命令将产生以下结果

A project\_repo/trunk

A project\_repo/branches

A project\_repo/tags

Checked out revision 1.

每一个成功的检出操作后，修订号打印。如果你想查看更多的信息存储库。然后执行info命令。

[tom@CentOS trunk]$ pwd

/home/tom/project\_repo/trunk

[tom@CentOS trunk]$ svn info

上面的命令将产生以下结果

Path: .

URL: http://svn.server.com/svn/project\_repo/trunk

Repository Root: http://svn.server.com/svn/project\_repo

Repository UUID: 7ceef8cb-3799-40dd-a067-c216ec2e5247

Revision: 1

Node Kind: directory

Schedule: normal

Last Changed Author: jerry

Last Changed Rev: 0

Last Changed Date: 2013-08-24 18:15:52 +0530 (Sat, 24 Aug 2013)

[tom@CentOS trunk]$

## SVN执行变更

Jerry 检出存储库中的最新版本，并开始工作的一个项目。他创建trunk 目录array.cfile内。

[jerry@CentOS ~]$ cd project\_repo/trunk/

[jerry@CentOS trunk]$ cat array.c

上面的命令将产生以下结果

#include <stdio.h>

#define MAX 16

int main(void)

{

int i, n, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

printf("Enter the elements

");

for (i = 0; i < n; ++i)

scanf("%d", &arr[i]);

printf("Array has following elements

");

for (i = 0; i < n; ++i)

printf("|%d| ", arr[i]);

printf("

");

return 0;

}

他想要测试自己的代码之前提交

[jerry@CentOS trunk]$ make array

cc array.c -o array

[jerry@CentOS trunk]$ ./array

Enter the total number of elements: 5

Enter the elements

1

2

3

4

5

Array has following elements

|1| |2| |3| |4| |5|

他编译和测试自己的代码，一切都按预期工作，现在是时候提交修改。

[jerry@CentOS trunk]$ svn status

? array.c

? array

Subversion 显示？在前面的文件名，因为它不知道做什么用这些文件。

Jerry 提交之前需要将此文件添加到挂起的更改列表。

[jerry@CentOS trunk]$ svn add array.c

A array.c

让我们检查状态操作。Subversion版本之前显示A array.c的文件被成功添加到挂起的更改列表。

[jerry@CentOS trunk]$ svn status

? array

A array.c

array.c文件存储到存储库使用commit命令使用-m选项，然后提交信息。如果省略-m选项，Subversion会带来文本编辑器，在这里你可以输入多行消息。

[jerry@CentOS trunk]$ svn commit -m "Initial commit"

Adding trunk/array.c

Transmitting file data .

Committed revision 2.

现在array.c的文件成功添加到资料库和版本号加一。

## SVN审查变更

Jerry 已经将 array.c 文件已经添加到库中。Tom 也检出最新的代码，并开始运作。

[tom@CentOS ~]$ svn co http://svn.server.com/svn/project\_repo --username=tom

上面的命令将产生以下结果

A project\_repo/trunk

A project\_repo/trunk/array.c

A project\_repo/branches

A project\_repo/tags

Checked out revision 2.

但他是创立的，有人已经添加的代码。所以他好奇谁这样做，他看到更多的细节，使用下面的命令检查日志消息：

[tom@CentOS trunk]$ svn log

上面的命令将产生以下结果

------------------------------------------------------------------------

r2 | jerry | 2013-08-17 20:40:43 +0530 (Sat, 17 Aug 2013) | 1 line

Initial commit

------------------------------------------------------------------------

r1 | jerry | 2013-08-04 23:43:08 +0530 (Sun, 04 Aug 2013) | 1 line

Create trunk, branches, tags directory structure

------------------------------------------------------------------------

当Tom观察了杰里的代码。他立即注意到，一个bug已被放入。Jerry不检查数组溢出，这将导致严重的问题。因此，Tom 决定来解决这个问题。经过modificationarray.c会这个样子。

#include <stdio.h>

#define MAX 16

int main(void)

{

int i, n, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

/\* handle array overflow condition \*/

if (n > MAX) {

fprintf(stderr, "Number of elements must be less than %d

", MAX);

return 1;

}

printf("Enter the elements

");

for (i = 0; i < n; ++i)

scanf("%d", &arr[i]);

printf("Array has following elements

");

for (i = 0; i < n; ++i)

printf("|%d| ", arr[i]);

printf("

");

return 0;

}

Tom 要使用状态操作，看到挂起的更改列表。

[tom@CentOS trunk]$ svn status

M array.c

array.c 文件被修改，这就是为什么Subversion 文件名前显示中号信。接下来Tom编译和测试自己的代码和它的正常工作。在提交更改之前，他想通过审查所作的变化，他要仔细检查它。

[tom@CentOS trunk]$ svn diff

Index: array.c

===================================================================

--- array.c (revision 2)

+++ array.c (working copy)

@@ -9,6 +9,11 @@

printf("Enter the total number of elements: ");

scanf("%d", &n);

+ if (n > MAX) {

+ fprintf(stderr, "Number of elements must be less than %d

", MAX);

+ return 1;

+ }

+

printf("Enter the elements

");

for (i = 0; i < n; ++i)

Tom 补充array.c文件，这就是为什么颠覆显示+号新生产线前的几行。现在准备提交更改。

[tom@CentOS trunk]$ svn commit -m "Fix array overflow problem"

上面的命令将产生以下结果

Sending trunk/array.c

Transmitting file data .

Committed revision 3.

Tom 的变化成功提交到库中。

## SVN更新过程

Jerry 提交第一个版本的代码。但他认为，他应该写两个函数接受输入和显示阵列的内容。修改后array.c 的样子。

#include <stdio.h>

#define MAX 16

void accept\_input(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

scanf("%d", &arr[i]);

}

void display(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

printf("|%d| ", arr[i]);

printf("

");

}

int main(void)

{

int i, n, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

printf("Enter the elements

");

accept\_input(arr, n);

printf("Array has following elements

");

display(arr, n);

return 0;

}

Jerry 编译和测试自己的代码，并准备提交修改。在此之前，他希望使用以下命令查看更改。

[jerry@CentOS trunk]$ svn diff

上面的命令将产生以下结果

Index: array.c

===================================================================

--- array.c (revision 2)

+++ array.c (working copy)

@@ -2,6 +2,24 @@

#define MAX 16

+void accept\_input(int \*arr, int n)

+{

+ int i;

+

+ for (i = 0; i & n; ++i)

+ scanf("%d", &arr[i]);

+}

+

+void display(int \*arr, int n)

+{

+ int i;

+

+ for (i = 0; i < n; ++i)

+ printf("|%d| ", arr[i]);

+

+ printf("

");

+}

+

int main(void)

{

int i, n, arr[MAX];

@@ -10,15 +28,10 @@

scanf("%d", &n);

printf("Enter the elements

");

+ accept\_input(arr, n);

- for (i = 0; i < n; ++i)

- scanf("%d", &arr[i]);

-

printf("Array has following elements

");

- for (i = 0; i < n; ++i)

- printf("|%d| ", arr[i]);

-

- printf("

");

+ display(arr, n);

return 0;

}

对于新添加的行颠覆显示+符号， - 符号前行为删除行的显示。Jerry 尝试提交更改，使用下面的命令：

[jerry@CentOS trunk]$ svn commit -m "Add function to accept input and to display array contents"

上面的命令将产生以下结果

Sending trunk/array.c

svn: Commit failed (details follow):

svn: File or directory 'array.c' is out of date; try updating

svn: resource out of date; try updating

Subversion是不允许Jerry 提交的变化，因为Tome已经修改了库和Jerry的工作拷贝是过时了。为了避免覆盖对方的变化颠覆这个操作失败。Jerry必须更新的工作副本之前提交自己的修改。于是，他用update命令

[jerry@CentOS trunk]$ svn update

G array.c

Updated to revision 3.

Subversion 显示文件名前G 字母，这意味着此文件已被合并。

[jerry@CentOS trunk]$ svn diff

上面的命令将产生以下结果

Index: array.c

===================================================================

--- array.c (revision 3)

+++ array.c (working copy)

@@ -2,6 +2,24 @@

#define MAX 16

+void accept\_input(int \*arr, int n)

+{

+ int i;

+

+ for (i = 0; i < n; ++i)

+ scanf("%d", &arr[i]);

+}

+

+void display(int \*arr, int n)

+{

+ int i;

+

+ for (i = 0; i < n; ++i)

+ printf("|%d| ", arr[i]);

+

+ printf("

");

+}

+

int main(void)

{

int i, n, arr[MAX];

@@ -15,15 +33,10 @@

}

printf("Enter the elements

");

+ accept\_input(arr, n);

- for (i = 0; i < n; ++i)

- scanf("%d", &arr[i]);

-

printf("Array has following elements

");

- for (i = 0; i < n; ++i)

- printf("|%d| ", arr[i]);

-

- printf("

");

+ display(arr, n);

return 0;

}

Subversion是只显示Jerry 的变化。但array.c的文件合并，如果观察仔细Subversion的现在修订号为3，在以前的输出显示修订号为2。只要审查文件中的变化和目的是什么。

jerry@CentOS trunk]$ svn log

------------------------------------------------------------------------

r3 | tom | 2013-08-18 20:21:50 +0530 (Sun, 18 Aug 2013) | 1 line

Fix array overflow problem

------------------------------------------------------------------------

r2 | jerry | 2013-08-17 20:40:43 +0530 (Sat, 17 Aug 2013) | 1 line

Initial commit

------------------------------------------------------------------------

r1 | jerry | 2013-08-04 23:43:08 +0530 (Sun, 04 Aug 2013) | 1 line

Create trunk, branches, tags directory structure

------------------------------------------------------------------------

现在，Jerry's 工作副本与存储库同步，他可以放心地提交他的变化。

[jerry@CentOS trunk]$ svn commit -m "Add function to accept input and to display array contents"

Sending trunk/array.c

Transmitting file data .

Committed revision 4.

## SVN修正错误

假设Jerry 不小心修改array.c 文件编译错误。现在，他想扔掉的变化。在这种情况下，恢复操作会有所帮助。恢复操作将撤消任何地方到一个文件或目录的更改，并解决任何冲突的状态。

[jerry@CentOS trunk]$ svn status

上面的命令将产生以下结果

M array.c

让我们 make array 如下:

[jerry@CentOS trunk]$ make array

上面的命令将产生以下结果

cc array.c -o array

array.c: In function ‘main’:

array.c:26: error: ‘n’ undeclared (first use in this function)

array.c:26: error: (Each undeclared identifier is reported only once

array.c:26: error: for each function it appears in.)

array.c:34: error: ‘arr’ undeclared (first use in this function)

make: \*\*\* [array] Error 1

Jerry 对 array.c文件进行恢复操作

[jerry@CentOS trunk]$ svn revert array.c

Reverted 'array.c'

[jerry@CentOS trunk]$ svn status

[jerry@CentOS trunk]$

现在编译代码。

[jerry@CentOS trunk]$ make array

cc array.c -o array

恢复操作后，他的工作副本是回到它的原始状态。恢复操作可以恢复单个文件以及完整的目录。恢复目录用户 -r 选项，如下图所示

[jerry@CentOS project\_repo]$ pwd

/home/jerry/project\_repo

[jerry@CentOS project\_repo]$ svn revert -R trunk

到现在我们已经看到了如何恢复已向工作副本的变化。但是如果你想提交的修订版本恢复。版本控制系统的工具不允许从资源库中删除历史。我们只能追加的历史。它会发生，即使你删除文件，从资源库中。要撤销一个旧的版本，我们必须扭转作出任何改变旧的版本，然后提交一个新的版本。这就是所谓的反向合并。

让我们假设Jerry 添加代码线性搜索操作。经核实后，他提交了他的变化。

[jerry@CentOS trunk]$ svn diff

Index: array.c

===================================================================

--- array.c (revision 21)

+++ array.c (working copy)

@@ -2,6 +2,16 @@

#define MAX 16

+int linear\_search(int \*arr, int n, int key)

+{

+ int i;

+

+ for (i = 0; i < n; ++i)

+ if (arr[i] == key)

+ return i;

+ return -1;

+}

+

void bubble\_sort(int \*arr, int n)

{

int i, j, temp, flag = 1;

[jerry@CentOS trunk]$ svn status

? array

M array.c

[jerry@CentOS trunk]$ svn commit -m "Added code for linear search"

Sending trunk/array.c

Transmitting file data .

Committed revision 22.

Jerry 对 Tom 做什么很好奇，于是，他检查Subversion 的日志消息。

[jerry@CentOS trunk]$ svn log

上面的命令将产生以下结果

------------------------------------------------------------------------

r5 | tom | 2013-08-24 17:15:28 +0530 (Sat, 24 Aug 2013) | 1 line

Add binary search operation

------------------------------------------------------------------------

r4 | jerry | 2013-08-18 20:43:25 +0530 (Sun, 18 Aug 2013) | 1 line

Add function to accept input and to display array contents

查看日志消息后，Jerry意识到自己做了严重的错误。因为Tom已经实现二进制搜索操作，这是优于线性搜索。意味着他的代码是多余的，现在Jerry有他的变化恢复到以前的版本。所以先找到存储库的当前版本。目前库是在修订22，我们要恢复到以前的版本，即修订21。

[jerry@CentOS trunk]$ svn up

At revision 22.

[jerry@CentOS trunk]$ svn merge -r 22:21 array.c

--- Reverse-merging r22 into 'array.c':

U array.c

[jerry@CentOS trunk]$ svn commit -m "Reverted to revision 21"

Sending trunk/array.c

Transmitting file data .

Committed revision 23.

## SVN解决冲突

Tom 决定为他们的项目添加README文件。因此，他创造了README文件，并添加到该TODO列表。在加入这个文件库在修订版本6。

[tom@CentOS trunk]$ cat README

/\* TODO: Add contents in README file \*/

[tom@CentOS trunk]$ svn status

? README

[tom@CentOS trunk]$ svn add README

A README

[tom@CentOS trunk]$ svn commit -m "Added README file. Will update it's content in future."

Adding trunk/README

Transmitting file data .

Committed revision 6.

Jerry 检出最新的代码，这是修订版本6。他就立刻开始工作。经过几个小时的Tom更新README文件并提交了他的变化。修改README这个样子。

[tom@CentOS trunk]$ cat README

\* Supported operations:

1) Accept input

2) Display array elements

[tom@CentOS trunk]$ svn status

M README

[tom@CentOS trunk]$ svn commit -m "Added supported operation in README"

Sending trunk/README

Transmitting file data .

Committed revision 7.

现在，库是修订版本7 Jerry的工作拷贝是过时了。Jerry 还更新README文件，并尝试提交他的变化。

Jerry' 的README文件看起来像这样。

[jerry@CentOS trunk]$ cat README

\* File list

1) array.c Implementation of array operation.

2) README Instructions for user.

[jerry@CentOS trunk]$ svn status

M README

[jerry@CentOS trunk]$ svn commit -m "Updated README"

Sending trunk/README

svn: Commit failed (details follow):

svn: File or directory 'README' is out of date; try updating

svn: resource out of date; try updating

* **查看冲突**

这是好事。Subversion的检测README文件已经改变，因为最后一次更新。因此，Jerry 更新自己的工作副本

[jerry@CentOS trunk]$ svn up

Conflict discovered in 'README'.

Select: (p) postpone, (df) diff-full, (e) edit,

(mc) mine-conflict, (tc) theirs-conflict,

(s) show all options:

Subversion抱怨README文件有冲突，Subversion不知道如何解决这个问题。所以Jerry 选择 df 查看冲突。

[jerry@CentOS trunk]$ svn up

Conflict discovered in 'README'.

Select: (p) postpone, (df) diff-full, (e) edit,

(mc) mine-conflict, (tc) theirs-conflict,

(s) show all options: df

--- .svn/text-base/README.svn-base Sat Aug 24 18:07:13 2013

+++ .svn/tmp/README.tmp Sat Aug 24 18:13:03 2013

@@ -1 +1,11 @@

-/\* TODO: Add contents in README file \*/

+<<<<<<< .mine

+\* File list

+

+1) array.c Implementation of array operation.

+2) README Instructions for user.

+=======

+\* Supported operations:

+

+1) Accept input

+2) Display array elements

+>>>>>>> .r7

Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,

(mc) mine-conflict, (tc) theirs-conflict,

(s) show all options:

* **延缓冲突**

Jerry 选择推迟（p）选项，这样他就可以解决冲突。

Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,

(mc) mine-conflict, (tc) theirs-conflict,

(s) show all options: p

C README

Updated to revision 7.

Summary of conflicts:

Text conflicts: 1

在文本编辑器中打开README文件后，他意识到，Subversion会包含Tom 的代码和他冲突标记代码。

[jerry@CentOS trunk]$ cat README

<<<<<<< .min

\* File list

1) array.c Implementation of array operation.

2) README Instructions for user.

=======

\* Supported operations:

1) Accept input

2) Display array elements

>>>>>>> .r7

Jerry 希望 Tom 同他的变化一起修改，所以他只是删除了行包含冲突标记。

所以修改后的README文件会如下这个样子。

[jerry@CentOS trunk]$ cat README

\* File list

1) array.c Implementation of array operation.

2) README Instructions for user.

\* Supported operations:

1) Accept input

2) Display array elements

Jerry 解决冲突并提交，并让他重试。

[jerry@CentOS trunk]$ svn commit -m "Updated README"

svn: Commit failed (details follow):

svn: Aborting commit: '/home/jerry/project\_repo/trunk/README' remains in conflict

[jerry@CentOS trunk]$ svn status

? README.r6

? README.r7

? README.mine

C README

* **解决冲突**

这里是C字母表示有README文件冲突。Jerry 解决冲突，但没有告诉Subversion他解决冲突。他使用解决命令通知Subversion有关解决冲突

[jerry@CentOS trunk]$ svn resolve --accept=working README

Resolved conflicted state of 'README'

[jerry@CentOS trunk]$ svn status

M README

[jerry@CentOS trunk]$ svn commit -m "Updated README"

Sending trunk/README

Transmitting file data .

Committed revision 8.

## SVN标签

版本控制系统支持标签操作，通过使用一个有意义的名字可以给特定版本的代码。标签允许特定版本的代码来描述和令人难忘的名字。例如BASIC\_ARRAY\_OPERATIONS的是更令人难忘的修订版本4。

让我们来看看标签操作为例。Tom 决定创建一个标签，这样他就可以访问代码更容易。

[tom@CentOS project\_repo]$ svn copy --revision=4 trunk/ tags/basic\_array\_operations

上面的命令将产生以下结果

A tags/basic\_array\_operations/array.c

Updated to revision 4.

A tags/basic\_array\_operations

成功完成后，将创建新目录标签目录内。

[tom@CentOS project\_repo]$ ls -l tags/

total 4

drwxrwxr-x. 3 tom tom 4096 Aug 24 18:18 basic\_array\_operations

Tom 要提交之前要仔细检查它。状态操作显示该标签操作是成功的，所以他可以放心地提交他的变化。

[tom@CentOS project\_repo]$ svn status

A + tags/basic\_array\_operations

[tom@CentOS project\_repo]$ svn commit -m "Created tag for basic array operations"

Adding tags/basic\_array\_operations

Committed revision 5.

## SVN分支

分支操作创建另一条线路开发。它是有用的，当有人想开发过程叉成两个不同的方向。让我们假设发布的产品版本1.0中，可能要创建新的分支，所以可以保持独立，发展2.01.0 bug修复。

在本节中，我们将看到如何创建，遍历和合并分支。Jerry 已经沮丧，因为冲突，所以他决定创建新的私有分支。

[jerry@CentOS project\_repo]$ ls

branches tags trunk

[jerry@CentOS project\_repo]$ svn copy trunk branches/jerry\_branch

A branches/jerry\_branch

[jerry@CentOS project\_repo]$ svn status

A + branches/jerry\_branch

[jerry@CentOS project\_repo]$ svn commit -m "Jerry's private branch"

Adding branches/jerry\_branch

Adding branches/jerry\_branch/README

Committed revision 9.

[jerry@CentOS project\_repo]$

Jerry 是在他的私人分支工作。他补充说数组的排序操作。Jerry 修改后的代码看起来是这样的。

[jerry@CentOS project\_repo]$ cd branches/jerry\_branch/

[jerry@CentOS jerry\_branch]$ cat array.c

上面的命令将产生以下结果

#include <stdio.h>

#define MAX 16

void bubble\_sort(int \*arr, int n)

{

int i, j, temp, flag = 1;

for (i = 1; i < n && flag == 1; ++i) {

flag = 0;

for (j = 0; j < n - i; ++j) {

if (arr[j] > arr[j + 1]) {

flag = 1;

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

void accept\_input(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

scanf("%d", &arr[i]);

}

void display(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

printf("|%d| ", arr[i]);

printf("

");

}

int main(void)

{

int i, n, key, ret, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

/\* Error handling for array overflow \*/

if (n >MAX) {

fprintf(stderr, "Number of elements must be less than %d

", MAX);

return 1;

}

printf("Enter the elements

");

accept\_input(arr, n);

printf("Array has following elements

");

display(arr, n);

printf("Sorted data is

");

bubble\_sort(arr, n);

display(arr, n);

return 0;

}

Jerry 编译并测试自己的代码和准备提交他修改的变化。

[jerry@CentOS jerry\_branch]$ make array

cc array.c -o array

[jerry@CentOS jerry\_branch]$ ./array

上面的命令将产生以下结果

Enter the total number of elements: 5

Enter the elements

10

-4

2

7

9

Array has following elements

|10| |-4| |2| |7| |9|

Sorted data is

|-4| |2| |7| |9| |10|

[jerry@CentOS jerry\_branch]$ svn status

? array

M array.c

[jerry@CentOS jerry\_branch]$ svn commit -m "Added sort operation"

Sending jerry\_branch/array.c

Transmitting file data .

Committed revision 10.

同时通过躯干，Tom决定执行搜索操作。Tom搜索操作添加代码，他的代码看起来像这样。

[tom@CentOS trunk]$ svn diff

上面的命令将产生以下结果

Index: array.c

===================================================================

--- array.c (revision 10)

+++ array.c (working copy)

@@ -2,6 +2,27 @@

#define MAX 16

+int bin\_search(int \*arr, int n, int key)

+{

+ int low, high, mid;

+

+ low = 0;

+ high = n - 1;

+ mid = low + (high - low) / 2;

+

+ while (low <= high) {

+ if (arr[mid] == key)

+ return mid;

+ if (arr[mid] > key)

+ high = mid - 1;

+ else

+ low = mid + 1;

+ mid = low + (high - low) / 2;

+ }

+

+ return -1;

+}

+

void accept\_input(int \*arr, int n)

{

int i;

@@ -22,7 +43,7 @@

int main(void)

{

- int i, n, arr[MAX];

+ int i, n, ret, key, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

@@ -39,5 +60,16 @@

printf("Array has following elements

");

display(arr, n);

+ printf("Enter the element to be searched: ");

+ scanf("%d", &key);

+

+ ret = bin\_search(arr, n, key);

+ if (ret < 0) {

+ fprintf(stderr, "%d element not present in array

", key);

+ return 1;

+ }

+

+ printf("%d element found at location %d

", key, ret + 1);

+

return 0;

}

审查后，他提交了他修改的变化。

[tom@CentOS trunk]$ svn status

? array

M array.c

[tom@CentOS trunk]$ svn commit -m "Added search operation"

Sending trunk/array.c

Transmitting file data .

Committed revision 11.

但Tom 好奇为什么 Jerry 一直在做他的私人分支。

[tom@CentOS trunk]$ cd ../branches/

[tom@CentOS branches]$ svn up

A jerry\_branch

A jerry\_branch/array.c

A jerry\_branch/README

[tom@CentOS branches]$ svn log

------------------------------------------------------------------------

r9 | jerry | 2013-08-27 21:56:51 +0530 (Tue, 27 Aug 2013) | 1 line

Added sort operation

------------------------------------------------------------------------

查看Subversion的日志消息，Tom发现，Jerry实现的排序操作。Tom执行搜索操作使用二进制搜索算法，它总是希望数据的排序顺序。但是，如果用户提供的数据，未排序的顺序呢？在这种情况下的二进制搜索操作将失败。于是，他决定采用Jerry 的代码，搜索操作前对数据进行排序。于是，他会要求Subversion的从Jerry的分支到主干合并代码。

[tom@CentOS trunk]$ pwd

/home/tom/project\_repo/trunk

[tom@CentOS trunk]$ svn merge ../branches/jerry\_branch/

--- Merging r9 through r11 into '.':

U array.c

合并后array.c的这个样子。

[tom@CentOS trunk]$ cat array.c

上面的命令将产生以下结果

#include <stdio.h>

#define MAX 16

void bubble\_sort(int \*arr, int n)

{

int i, j, temp, flag = 1;

for (i = 1; i < n && flag == 1; ++i) {

flag = 0;

for (j = 0; j < n - i; ++j) {

if (arr[j] > arr[j + 1]) {

flag = 1;

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int bin\_search(int \*arr, int n, int key)

{

int low, high, mid;

low = 0;

high = n - 1;

mid = low + (high - low) / 2;

while (low <= high) {

if (arr[mid] == key)

return mid;

if (arr[mid] > key)

high = mid - 1;

else

low = mid + 1;

mid = low + (high - low) / 2;

}

return -1;

}

void accept\_input(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

scanf("%d", &arr[i]);

}

void display(int \*arr, int n)

{

int i;

for (i = 0; i < n; ++i)

printf("|%d| ", arr[i]);

printf("

");

}

int main(void)

{

int i, n, ret, key, arr[MAX];

printf("Enter the total number of elements: ");

scanf("%d", &n);

/\* Error handling for array overflow \*/

if (n > MAX) {

fprintf(stderr, "Number of elements must be less than %d

", MAX);

return 1;

}

printf("Enter the elements

");

accept\_input(arr, n);

printf("Array has following elements

");

display(arr, n);

printf("Sorted data is

");

bubble\_sort(arr, n);

display(arr, n);

printf("Enter the element to be searched: ");

scanf("%d", &key);

ret = bin\_search(arr, n, key);

if (ret < 0) {

fprintf(stderr, "%d element not present in array

", key);

return 1;

}

printf("%d element found at location %d

", key, ret + 1);

return 0;

}

Tom 测试完成后，提交他的修改到版本库。

[tom@CentOS trunk]$ make array

cc array.c -o array

[tom@CentOS trunk]$ ./array

Enter the total number of elements: 5

Enter the elements

10

-2

8

15

3

Array has following elements

|10| |-2| |8| |15| |3|

Sorted data is

|-2| |3| |8| |10| |15|

Enter the element to be searched: -2

-2 element found at location 1

[tom@CentOS trunk]$ svn commit -m "Merge changes from Jerry's code"

Sending trunk

Sending trunk/array.c

Transmitting file data .

Committed revision 12.

[tom@CentOS trunk]$