



**HAPPINESS**

## GIT 学习教程



DEZEMING FAMILY

DEZEMING





Copyright © 2022-01-10 Dezeming Family

**Copying prohibited**

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, without the prior written permission of the publisher.

Art. No 0

ISBN 000-00-0000-00-0

Edition 0.0

Cover design by Dezeming Family

Published by Dezeming

Printed in China



# 目录

0.1	本书前言	5
<b>1</b>	<b>初识 Git</b>	<b>6</b>
1.1	为什么要使用 Git	6
1.2	分布式版本控制与 Git	7
1.3	Git 的安装	8
<b>2</b>	<b>Git 的本地基础使用方法</b>	<b>10</b>
2.1	版本库的创建	10
2.2	提交文件到版本库	11
2.3	版本回退	13
2.4	Git 的工作区与暂存区	15
2.5	修改的撤销	16
2.6	小结	17
<b>3</b>	<b>Git 的高阶使用方法</b>	<b>18</b>
3.1	自定义 Git	18
3.2	忽略特殊文件	19
3.3	Git 可视化工具	21

3.4	分支的创建、合并与冲突	21
3.5	保存与恢复现场	27
3.6	小结	29
4	远程仓库与多人协作 .....	30
4.1	远程仓库与 Github	30
4.2	添加到远程库与从远程库 clone	31
4.3	多人协作	33
4.4	rebase	36
4.5	标签管理	38
4.6	参与 Github 开源项目	39
4.7	搭建 Git 服务器	39
4.8	小结	40
	Literature .....	40



# 前言及简介

*DezemingFamily* 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 [<https://dezeming.top/>] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

## 0.1 本书前言

---

在编写程序时，往往需要多次开发，并且保留一些历史版本，以及对不同的需求扩展到新的版本，于是，版本控制软件就变得非常重要。有了版本控制软件，我们就不再需要手动记录一些文件的增删改查，而是可以随时回溯到历史版本。

或许有人会觉得 Git 只是一个简单的工具，在 Github 上学习一下基本操作大致就可以了，我想这是很多人的误区——人们不会觉得 C 语言非常容易，这是基本的编程工具。但人们喜欢习惯性地忽略一些其他“非语言类”的代码构建工具，例如 Makefile，Git 以及 IDE 的调试器等。本文参考了 [1] 和 [3] 的讲解结构，并丰富和完善了讲解顺序和讲解内容。在讲解顺序上，我们先讲解本地 Git 的初级使用方法，然后讲解 Git 的更高阶使用，之后过渡到远程 Git 与多人协作。内容上，我们通过可视化工具来更好地理解和解释 Git 的操作和效果。

当然，仅仅依靠一本书是不可能把 Git 学会的，而且类比一下编程，哪怕你已经熟悉了一个编程语言，当遇到一些 Bug 时可能还是需要去 Google 或百度上找资料，Git 也是如此，实际情况中我们常常会遇到很多复杂的情况，比如一些错误的操作，此时就需要查阅资料和论坛来解决了。





# 1. 初识 Git

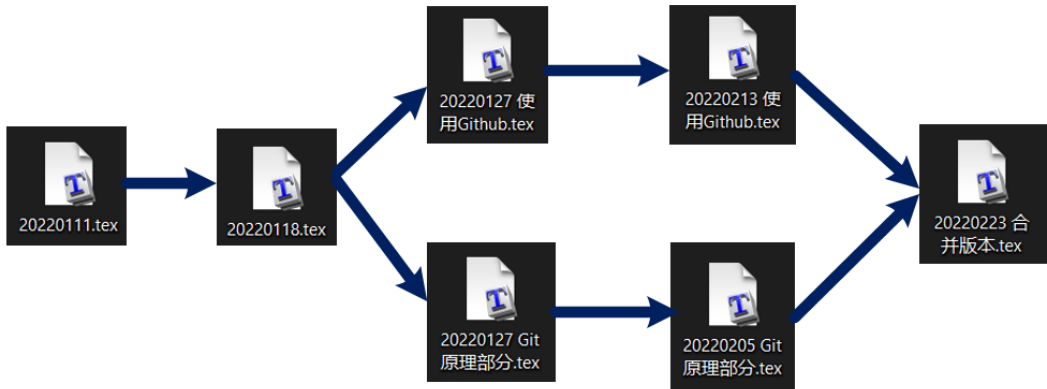
1.1	为什么要使用 Git	6
1.2	分布式版本控制与 Git	7
1.3	Git 的安装	8

本章对 *Git* 进行一些基础入门的介绍，包括 *Git* 的作用和基本概念。

## 1.1 为什么要使用 Git

比如，你想写一本书，每写一部分内容，你就保存为一个版本。有时候，你觉得写了一些废话，但是觉得以后可能会用到，这时你就舍不得删掉，而是把这个版本的一些概要记录下来（比如，该版本写了一些关于“为什么要使用 Git”的内容）。日久天长，你会发现自己已经保存了海量的版本文件。

而有些时候，你的写作出现了岔路：你在一个版本里写 Github 的使用方法，而你在另一个版本里写 Git 的原理，两个版本都各自发展为了一个版本体系，直到有一天，你想把它们合并起来：



而如果你在这些各自的分支版本上都进行了一些增删操作，合并就变得很困难。有时候，你的工作并不止自己完成，还要与别人一起合作，这时，版本的改动和相互之间协作就变得更困难了。

Git 就是一个可以帮你做好版本控制的软件系统，它会自动帮你记录文件的改动（自动记录很重要，而不是你自己手动把每一行改动都记录下来，但我们可以自己写一个简单的说明），还可以让同事们一起协作编辑。

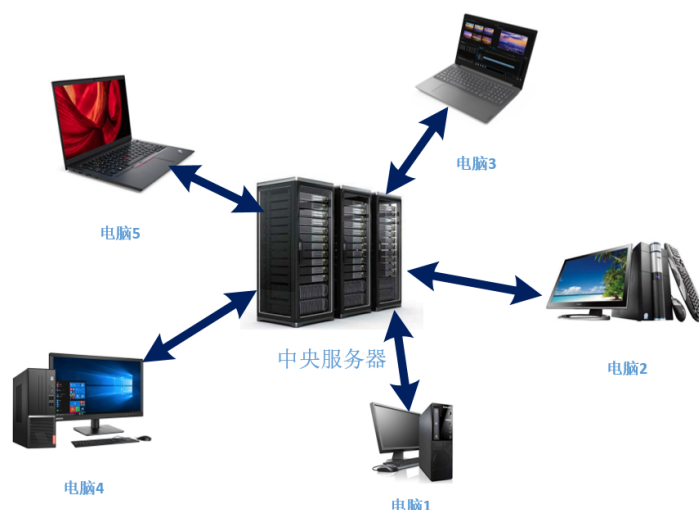
## 1.2 分布式版本控制与 Git

分步式版本控制（distributed revision control，又称为分散式版本控制）[1]，又称去中心化版本控制（decentralized version control），是一种版本控制的方式，它允许软件开发者可以共同参与一个软件开发过程，但是不必在相同的网络系统下工作（可以分开单独进行，独立开发出新的版本）。

分布式版本控制的作法是在每个开发者电脑中复制一份完整的代码库以及完整历史。因此在无法连接网络链路，仍可以进行软件的分支及合并，可以加速大部份的作业，增加此情形可以进行的工作，而且系统的代码库可以在多家电脑上备份，不需靠单一位置的备份。而多个位置的代码库再通过其他机制来达到同步。

Git 是一个用 C 语言开发的分散式版本控制软件，著名的 Github 就是使用 Git 做版本控制的代码托管平台。Git 是 Linux 的创始人 Linus 发布的，曾经的 Linux 在 BitKeeper（也是分布式版本控制器）上托管（有免费使用权），但后来由于一些 Linux 的极客们尝试破解 BitKeeper，所以使得 BitKeeper 的公司 BitMover 收回了它的免费使用权。之后，Linus 用两周重写了一个新的版本控制系统，也就是现在的 Git。

有分布式就有集中式，早期的 SVN 和 CVS 都属于集中式版本控制系统 [2]，有一个单一的服务器来管理，这个服务器保存了所有文件的修订和版本，开发人员连接到服务器上以后，就可以取出文件以及提交更新。我们想要更改和修订时，就需要给联网到服务器：



而分布式版本控制理论上并没有中央服务器，如果使用客户端提取文件则需要把整个代码仓库镜像完整拷贝下来，每个人都保存一个完整的版本库。如果某一个服务器出现了故障，就可以使用任何一个本地仓库来恢复。在实际操作时，Git 也会拟定一个中央仓库，而当中央仓库出现错误时，则开发者可以新建一个新的中央仓库，然后将本地仓库同步到新的中央仓库上：



对于代码的托管平台，基于 Git 的托管平台主要有 Github 和 GitLab。Github 免费提供公共仓库，但是需要开源，如果不想开源则需要付费；另一个基于 Git 的公共仓库是 GitLab，可以免费创建私人仓库。

### 1.3 Git 的安装

由于我现在手里暂时没有安装了 Linux 系统的机器，所以暂时只介绍在 Windows 系统下的安装。以后会补充上 Linux 下的安装和使用。

在 [4] 进行下载，如下图，直接点击“Click here to download”下载即可。注意下面的两个其他的 Git 下载选项，Standalone Installer 表示安装版，而 Portable (“thumbdrive edition”) 是绿色版，直接解压就能用。我们一般都是使用安装版，安装好以后，右键鼠标就能出现 Git 选项。

[Click here to download](#) the latest (2.34.1) 64-bit version of Git for Windows. This is the most recent maintained build. It was released about 2 months ago, on 2021-11-25.

#### Other Git for Windows downloads

##### Standalone Installer

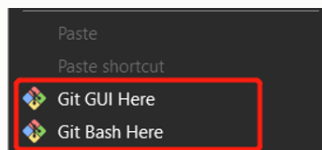
32-bit Git for Windows Setup.

64-bit Git for Windows Setup.

##### Portable (“thumbdrive edition”)

32-bit Git for Windows Portable.

64-bit Git for Windows Portable.

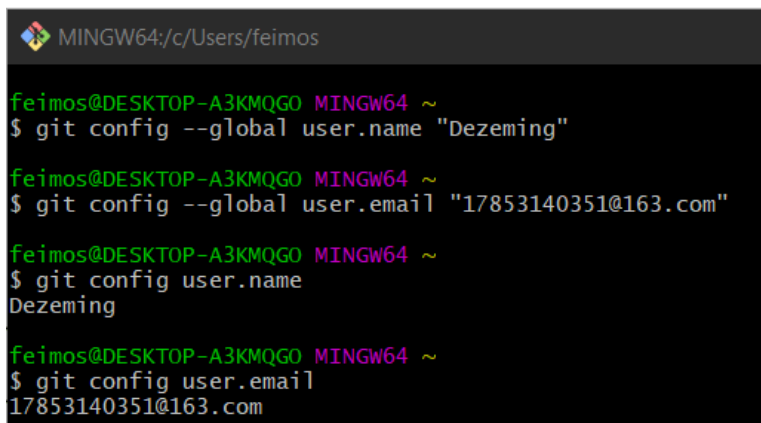


安装好之后我们就可以在菜单里打开 Git Bash。我们需要进行设置，告诉 Git 你的 Email 地址和用户名，这是因为 Git 是分布式版本控制系统，每个用户都要进行备注。我们一般会使用一个 -global 参数，表示机器上所有的 Git 仓库都会设置为当前状态（由当前用户管理），也可以对不同的 Git 仓库设置不同的用户名和 Email。我们实际操作一下，在 Git Bash 里输入：

```
$ git config --global user.name "Dezeming"
$ git config --global user.email "12345678@163.com"
```

还可以输入命令来查看自己的用户名和邮箱：



A terminal window with a dark background and light-colored text. The title bar at the top reads 'MINGW64:/c/Users/feimos'. The terminal shows four lines of commands and their outputs. The first line is a command to set the global user name. The second line is a command to set the global user email. The third line is a command to view the user name, followed by its output. The fourth line is a command to view the user email, followed by its output.

```
MINGW64:/c/Users/feimos  
feimos@DESKTOP-A3KMQGO MINGW64 ~  
$ git config --global user.name "Dezeming"  
feimos@DESKTOP-A3KMQGO MINGW64 ~  
$ git config --global user.email "17853140351@163.com"  
feimos@DESKTOP-A3KMQGO MINGW64 ~  
$ git config user.name  
Dezeming  
feimos@DESKTOP-A3KMQGO MINGW64 ~  
$ git config user.email  
17853140351@163.com
```

我们设置了用户名和邮箱以后，当把本地代码提交到远程仓库中时，远程仓库就会记录是谁提交的。比如远程仓库是 Github，如果我们的邮箱是 Github 里真实存在的，则 commits 就会显示你的邮箱对应的账号，否则就只会显示你的用户名。那么问题来了，如果有人故意要冒充别人怎么办呢？比如有人知道你的注册邮箱，那么他可能会恶意用你的邮箱来对 Github 社区捣乱，这时可以通过 GPG 签名的方法来有效制止这种现象，我们暂时先不深入讲解这些内容，何况本人对里面的很多细节和原理也不甚了解。



## 2. Git 的本地基础使用方法

2.1	版本库的创建	10
2.2	提交文件到版本库	11
2.3	版本回退	13
2.4	Git 的工作区与暂存区	15
2.5	修改的撤销	16
2.6	小结	17

本章介绍如何使用本地 *Git* 来控制版本，介绍一些基础功能。

### 2.1 版本库的创建

版本库一般被称为代码仓库，如果你以前从 Github 上下载过代码，在 Github 页面应该经常看到这个词：repository。一个仓库收录了里面所有文件的创建、更改和删除，并记录了全部的历史，以及可以进行还原。

使用 `pwd` 命令可以查看当前文件夹：

```
$ pwd
```

然后我们使用下面一系列命令，在你的 D 盘开发目录下创建一个 `gitTest` 目录（目录名尽可能不要用中文），并进入：

```
$ cd d:
$ cd developer
$ mkdir gitTest
$ cd gitTest
```

之后，使用 `git init` 命令，就可以创建一个 Git 管理的仓库：

```
feimos@DESKTOP-A3KMQGD MINGW64 /d/developer/gitTest
$ git init
Initialized empty Git repository in D:/Developer/gitTest/.git/
```

可以看到自动生成了一个 `.git` 目录（是一个隐藏目录），注意该目录是用来管理和跟踪记录版本的，不要删除或任意修改。

## 2.2 提交文件到版本库

所有的版本控制系统只能跟踪文本文件的改动,例如.txt,.tex 以及任意的代码文件,例如.cpp。使用文本文件,Latex 就可以告诉你你每次的修改,比如在某一行添加了一行代码,或者某一行修改了一点语法之类的。对于图片、pdf 以及 word 文件,它没有办法跟踪文件的变化,但是可以记录它文件大小的改动。

我们一般会使用 UTF-8 编码或者 Unicode 编码的文本文件,这里一般要求使用 UTF-8 编码,不会起冲突(各大平台都会默认支持 UTF-8)。在 windows 系统上编写程序时,根据 [3] 所述微软会在记事本前面加 0xEFBBBF 这些字符,导致容易出错,而在我的 window10 系统中我使用磁盘调试工具没有看到这些字符,可能是微软取消了。

现在我们编写一个 txt 文件(建议使用 VSCode 或者 Notepad++),可以叫做 readme.txt,然后写上几句话:

```
Now let us start our project.  
This is the first day.
```

然后把文件移动到当前目录下(刚才创建的 gitTest 目录下),然后告诉 Git 要把该文件添加到仓库:

```
$ git add readme.txt
```

上面的命令没有任何显示就说明成功添加。你可以一次性提交很多文件,但是你可能会考虑,如果你每提交一次文件,Git 都会帮你进行记录,岂不是太费时间和存储了嘛?其实 Git 的跟踪记录并不是自动的,因此你可以一次性改动很多文件,然后输入命令来跟踪记录。

使用 git commit 来对前面提交到内容进行一个说明,并且启动跟踪记录(-m 表示输入的内容是本次提交的说明):

```
$ git commit -m "start our project"
```

等输入完以后,Git 就会告诉你相比于上一次,本次的提交有哪些改动的内容:

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)  
$ git commit -m "start our project"  
[master (root-commit) f22324e] start our project  
1 file changed, 2 insertions(+)  
create mode 100644 readme.txt
```

我们在 readme.txt 文件中进行一些修改,并添加一行:

```
Now let us start our project.  
This is the 1st day.  
I am happy.
```

之后使用 git status 命令来查看,我们可以看到,根据输出信息显示,虽然修改了某个文件,但是并没有添加,也没通知仓库去跟踪修改情况:

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)  
$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git restore <file>..." to discard changes in working directory)  
        modified:   readme.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")
```



我们还可以进一步查看之前修改了哪些内容（比如你某次修改以后时间久远，你已经不记得自己改了哪些东西），使用 `git diff` 即可：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git diff
diff --git a/readme.txt b/readme.txt
index e5bbbc1..73d2c50 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,3 @@
- Now let us start our project.
- This is the first day.
\ No newline at end of file
+ This is the 1st day.
+ I am happy.
\ No newline at end of file
```

对于输出内容，`-` 表示变动前的文件，`+++` 表示变动后的文件。

我们再次调用 `git add` 添加文件，以及调用 `git status` 来查看状态，确认在主分支中已经提交了修改：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git add readme.txt

feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.txt
```

之后我们就可以使用 `git commit` 命令进行提交，然后可以再使用 `git status` 来查看状态。如果当前没有新的内容，则 `git status` 就会显示当前工作树是干净的：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git commit -m "change first to 1st, and add a line"
[master f4b63cf] change first to 1st, and add a line
1 file changed, 2 insertions(+), 1 deletion(-)

feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git status
On branch master
nothing to commit, working tree clean
```

我们再修改两次，并把每次都添加和提交：

第一次：

文件内容修改为：

Now let us start our project.

This is the 1st day.

Now I need food.

I am happy.

输入：

```
$ git add readme.txt
```

```
$ git commit -m "add I need food"
```

第二次：

文件内容修改为：

Now let us start our project.

Now I need food.

```
I am happy.
输入:
$ git add readme.txt
$ git commit -m "delete 1st day"
```

现在，我们已经更新了多个版本，接下来我们就要进行一些其他的操作了。

另外，为了以后的操作更方便，这里把每次重新打开 Git Bash 后需要执行的命令列一下：

```
$ git config --global user.name "Dezeming"
$ git config --global user.email "12345678@163.com"
$ cd d:
$ cd developer
$ cd gitTest
```

## 2.3 版本回退

### 回退到前面的版本

使用 `git log` 命令，就可以查看从最近到最早的提交日志：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git log
commit e1e57c65ae516a432586f20ca06cf14b6db503c6 (HEAD -> master)
Author: Dezeming <17853140351@163.com>
Date: Thu Jan 13 20:42:14 2022 +0800

    delete 1st day

commit 18cdccb82616e8eaac6012b55852a20b651d2436
Author: Dezeming <17853140351@163.com>
Date: Thu Jan 13 20:38:57 2022 +0800

    add I need food

commit f4b63cfa565115dda167c2446fba6786a43dc91f
Author: Dezeming <17853140351@163.com>
Date: Thu Jan 13 09:50:01 2022 +0800

    change first to 1st, and add a line

commit f22324e05c2eb4262126a403a3bf382ed1fb2e3f
Author: Dezeming <17853140351@163.com>
Date: Wed Jan 12 09:02:12 2022 +0800

    start our project
```

上面的输出中，commit 后面跟着的编码是 commit id，也就是版本号。对于分布式版本控制系统而言，由于需要多人协作，共同对一个版本库做贡献，所以没法使用简单的递增数字来编号，例如 1,2,3，因此，需要根据你的用户账号等信息计算出一个非常大的数字来避免冲突。

使用 `git log -1` 就可以查看最近一次提交的版本日志。

我们现在希望将当前版本回退一个版本，即回退到“add I need food”版，使用 `git reset` 命令即可。若要回退到前一个版本，则调用：

```
$ git reset --hard HEAD^
```

其中，`HEAD^` 表示上一个版本，`HEAD^^` 则表示上上个版本。如果想要回退到 10 个版本之前，则调用 `HEAD~10`。这里的 `--hard`（注意前面是两个-）参数的功能涉及工作区与暂存区，在当前的

实验阶段，我们先使用带-hard 的 reset 命令。

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git reset --hard HEAD^
HEAD is now at 18cdccb add I need food
```

再次调用 git log 命令，就可以看到提交日志只剩下了三个，最新的提交内容已经被删除了。

## 恢复到回退之前的版本

我们再次调用：

```
$ git reset --hard HEAD^
```

来回退到上一个版本，然后查看 log，发现只剩两个版本了。如果此时还想恢复到之前的“add I need food”版本应该如何？幸好此时我们还没有关闭 Git Bash 窗口，我们查看那个版本的 id 是：

```
commit 18cdccb82616e8eaac6012b55852a20b651d2436
```

然后调用 reset 命令：

```
$ git reset --hard 18cdc
```

这样就可以恢复到回退之前的版本了，我们只需要 id 的前几个位就可以，Git 会帮我们自动找到（除非前几位相同，则需要更多位来区分不同的版本 id）。

我们可以再尝试调用：

```
$ git reset --hard HEAD^
```

再次回退到倒数第二次 commit，然后调用：

```
$ git reset --hard e1e57
```

此时就可以恢复到最新一次我们 commit 的结果。注意此时调用 log 就会发现前面的 18cdc 也会被恢复，而不是从倒数第二个版本直接到最新的版本。

注意，Git 的版本恢复和回退其实就是操作 HEAD 指针指向不同的版本，所以速度非常快。现在你可能要问了，如果有一次回退之后关掉了 Git Bash 了，但是又想恢复到没有回退之前的样子，应该怎么办呢？其实很简单，调用：

```
$ git reflog
```

就可以查看之前的全部 commit 和 reset 命令，就能找到对应的版本 id，也就可以恢复到任意版本了：

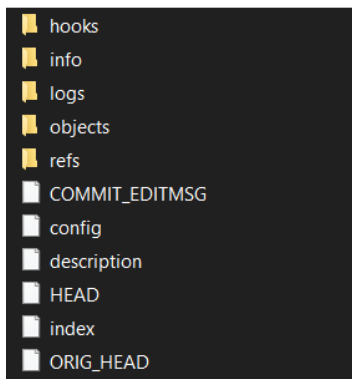
```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git reflog
e1e57c6 (HEAD -> master) HEAD@{0}: reset: moving to e1e57
f4b63cf HEAD@{1}: reset: moving to HEAD^
18cdccb HEAD@{2}: reset: moving to 18cdc
f4b63cf HEAD@{3}: reset: moving to HEAD^
18cdccb HEAD@{4}: reset: moving to HEAD^
e1e57c6 (HEAD -> master) HEAD@{5}: commit: delete 1st day
18cdccb HEAD@{6}: commit: add I need food
f4b63cf HEAD@{7}: commit: change first to 1st, and add a line
f22324e HEAD@{8}: commit (initial): start our project
```



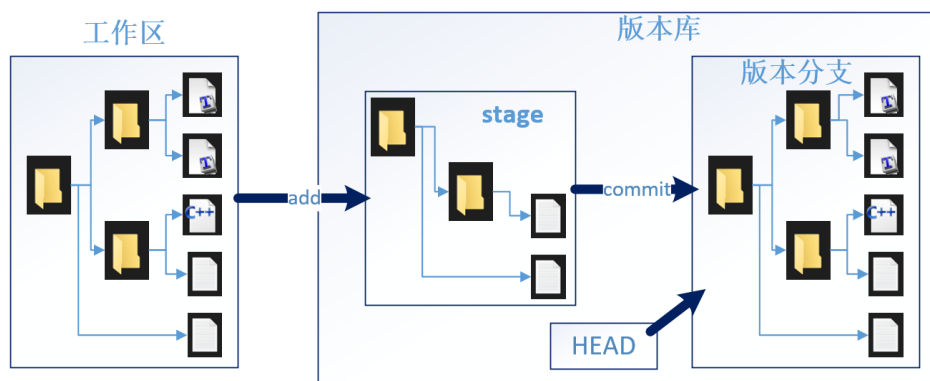
## 2.4 Git 的工作区与暂存区

所谓工作区 (working tree, 也叫工作树), 就是我们 Git 进入的目录, 在本文中是 D:/Developer/gitTest。

前面说过, 该工作区里的 .git 隐藏目录保存了 Git 的跟踪目录。Git 版本库里有很多内容, 如下图。但我们没必要去分析这些文件, 而是可以借助一些命令来帮助理解。



一开始, Git 会为我们创建一个主分支 master, 以及指向该主分支的指针 HEAD。当我们调用 git add 命令时, 相当于把文件添加到了暂存区, 而没有提交到主分支。只有当调用 git commit 命令以后, 才会把暂存区的内容提交到当前的分支上。



这个暂存区一般被称为 stage (也叫 index)。git add 命令把所有需要提交的修改内容都放到暂存区里, 然后执行 git commit 就可以把暂存区所有的修改都提交到分支中。

如果我们修改了文件, 然后执行 git add 添加了该文件, 然后再次修改了该文件, 还没有执行 git add 就直接执行了 git commit, 则 Git 并不会把第二次修改进行提交, 这是因为 Git 的管理只针对放入暂存区的修改。

### git reset 的参数

git reset 可以有三种参数 [6]: -soft, -mixed (默认方法), -hard。

假设我们在上一次 commit 之后, 在工作区的某个总共 8 行的文件中的末尾添加了两行代码, 然后再次进行了 commit。然后又添加了两行代码, 并且 add 到了暂存区, 但没有 commit。当前工作区的该文件一共 12 行代码。

执行 -soft 参数的 git reset 不会修改 stage (或者叫 index) 和工作树, 而是仅仅将指针 HEAD 指定到某个版本。此时 stage 里仍然保存了最近一次添加的修改。工作树中也仍然是你最新修改

的内容。所以此时工作树中的该文件还是 12 行，而且 stage 中的该文件也仍然是 12 行。

-mixed 参数会修改 stage，但是不会修改工作区。也就是说，如果你在工作区进行了一些修改，但是还没有 add 到 stage 里，执行 reset 以后，stage 会被更新到 reset 的目标版本，但是工作树仍保持你新修改的内容。此时工作树中的文件还是 12 行，但是 stage 中的该文件是 8 行。

-hard 就是全部都修改的参数，它会把工作树更新到你 reset 的目标版本，因此，如果你没有把新修改的内容 add 并 commit，则你新修改的东西就会全部消失——它可能会让你失去最近一段时间的工作进度。此时工作树和 stage 中的该文件都是 8 行。

## 2.5 修改的撤销

有时候我们修改了一些文件，但是我们想恢复到修改前的样子，此时可能有两种情况，一是我们修改前已经 add 并且 commit 过了；二是我们修改前只是 add 过，但是没有 commit。

我们当前的工作区只有一个 readme.txt 文件，当前文件有三行（如果不是这样，请修改文件并进行 add 和 commit，保证当前文件有三行，并且调用 git status 可以看到工作树是干净的）：

```
Now let us start our project.  
Now I need food.  
I am happy.
```

现在我们修改一下该文件，把中间那一行改为：

```
Now let us start our project.  
Now I don't need food.  
I am happy.
```

此时我们并没有 add，也没有 commit。假如日久天长，我们突然不想提交修改，而是想恢复到最后一次提交的样子，则可以调用 checkout 命令（这里的-参数很重要，不使用-的 checkout 命令有着其他的作用，这里先不进行介绍）：

```
$ git checkout — readme.txt
```

然后再次打开 readme.txt 文件就是：

```
Now let us start our project.  
Now I need food.  
I am happy.
```

我们再次在第二行加上 don't，然后 add 到暂存区，但是并不 commit。然后再把文件修改为（在最后添加了 Haha）：

```
Now let us start our project.  
Now I don't need food.  
I am happy. Haha.
```

然后再调用：

```
$ git checkout — readme.txt
```

打开 `readme.txt`，可以看到此时文件变成了：

```
Now let us start our project.  
Now I don't need food.  
I am happy.
```

因此，`checkout` 可以将工作区恢复到最近一次 `commit` 或者 `add` 时的状态。

如果我们已经 `add` 了某文件，但是想恢复到最后一次 `commit` 时该文件的状态，则使用下面的语句即可：

```
$ git reset HEAD readme.txt  
$ git checkout — readme.txt
```

第一句命令首先将暂存区的内容撤销修改（变为 `unstage`），但不会改变工作区，此时暂存区就变干净了。第二句就是把上次 `commit` 的内容恢复到工作区。

## 删除文件

文件能添加自然也能删除。

如果我们删除了某个被 `commit` 过的文件，例如 `readme.txt`，调用 `git status`，Git 显示你删除了文件。如果你想在 Git 版本库中也删除，则可以调用下面的命令来告诉 Git 你确实想删除该文件，并 `commit`：

```
$ git rm readme.txt  
$ git commit -m "delete readme.txt"
```

如果你只是误删了，则可以使用 `checkout` 命令来恢复（其实就是使用版本库里面的文件恢复到工作区）：

```
$ git checkout — readme.txt
```

## 2.6 小结

本章主要讲解了如何提交文件以及版本进退，命令也都比较简单。下一章我们会讲解更复杂的 Git 使用方法，难点在于分支的控制，但只要自己操作两三遍，就会觉得整个流程其实并没有那么复杂。



## 3. Git 的高阶使用方法

3.1	自定义 Git	18
3.2	忽略特殊文件	19
3.3	Git 可视化工具	21
3.4	分支的创建、合并与冲突	21
3.5	保存与恢复现场	27
3.6	小结	29

本章介绍更高阶的 *Git* 使用，包括分支的管理和冲突解决。

### 3.1 自定义 Git

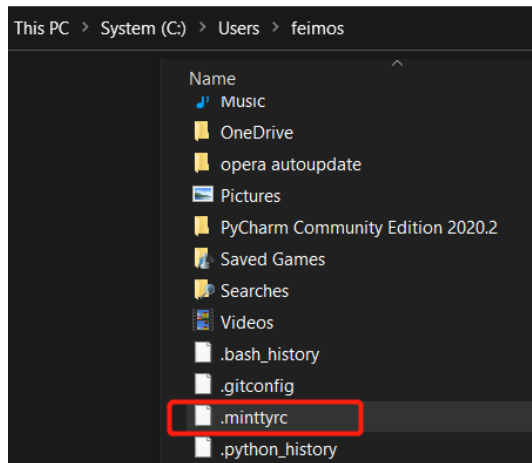
#### 界面

在 Linux 系统中我们可以调用 `config` 命令来让 Git 的颜色更丰富一些（在 Windows 中的颜色显示已经很丰富了）：

```
$ git config --global color.ui true
```

我们想把界面改得更好看一点，纯黑色太压抑，同时还想换一下字体。

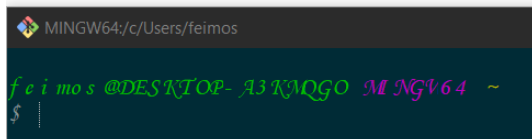
在 Windows 中，新建一个 Git 版本库默认在 `C:/user/<user names>` 文件夹下，在该文件夹有一个 `.minttyrc` 文件，如果没有你可以自己创建一个：



然后输入你想要的字体和背景色：

```
Font=华文隶书
FontHeight=14
ForegroundColor=131,148,150
BackgroundColor=0,43,54
```

这样重新打开 Git Bash 以后就会显示为：



## 别名

如果有的命令比较长，可以使用别名，我们列举一下 [3] 的例子。

使用 st 表示 status：

```
$ git config --global alias.st status
# 此时，git st 等同于 git status
```

撤销修改：

```
$ git config --global alias.unstage 'reset HEAD'
# 此时，git unstage readme.txt 等同于 git reset HEAD readme.txt
```

这些--global 参数表示针对当前用户起作用，否则就只对当前的仓库起作用。

config 命令会修改配置文件，对于当前仓库的配置文件，在当前仓库的.git/config 文件中，里面有 [alias] 信息。而当前用户的配置文件在 C/user/<user names> 文件夹下的.gitconfig 文件中。配置时可以直接修改配置文件。

## 3.2 忽略特殊文件

### 建立.gitignore 文件

我们在当前的 Git 工作目录下新建一个文件，叫 debunk.txt，用于吐槽这个项目的各种不切实际的要求。你肯定不希望这个吐槽文件被提交 commit，因此你不会将它提交，但是你调用 git status 命令时会出现 Untracked files：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        debunk.txt

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

如果你的文件项目很庞大，不需要被追踪的文件也有很多，则每次修改查看状态时都会出现海量的 Untracked files 信息，非常杂乱。Git 给出了一种方法，即创建.gitignore 文件，然后把想要忽略追踪的文件名填写进去，Git 就会自动忽略这些文件。

我们从 [7] 中可以看到一些预设的.gitignore 配置文件，对于不同的编程语言有不同的文件，这是为什么呢？因为我们在编译时，肯定会生成一些你不想要追踪的内容，比如编译产生的中间文件，而不同的编程语言产生的中间文件是不同的。当然，生成的可执行文件我们也不想被跟踪。甚至有时候，我们希望对一整个文件夹的内容都不进行跟踪（该文件夹是系统生成的包含调试信息的文件夹）。于是我们可以把这些内容都放到.gitignore 文件里。

我们新建两个文件 t1.a 和 t2.a，然后新建一个文件夹，叫做 build（模仿编译生成的文件夹），在该文件夹中新建一个 t1.b 和一个 t2.d 文件，我们希望这些新建的文件和 debunk.txt 一样都不要被追踪。对于我们当前那么简单的工程来说，还没必要使用 [7] 中的配置文件，我们完全可以自己写一个.gitignore 文件，：

```
build
debunk.txt
*.a
```

此时运行 git status 命令就会显示这些未被追踪的文件（甚至包括了.gitignore 文件）：

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  .gitignore
  Personal experience/
  t1.a
  t2.a
```

此时我们还需要将.gitignore 文件提交上去：

```
$ git add .gitignore
$ git commit -m "commit .gitignore file"
```

此时再运行 git status 命令就会发现工作树是干净的了。

## 一些注意事项

如果我们提交的.gitignore 文件中有 \*.a，则后续我们就没法再把.a 文件添加进来了：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git add t3.a
The following paths are ignored by one of your .gitignore files:
t3.a
hint: Use -f if you really want to add them.
hint: Turn this message off by running
hint: "git config advice.addIgnoredFile false"
```

如果我们想把该文件添加进来，可以按照上面的提示使用 -f 来强制添加：

```
$ git add -f t3.a
```

或许我们认为.gitignore 文件写的有问题，可以用下面的命令来检查：

```
$ git check-ignore -v t3.a
```

输出会显示是你定义的哪条规则把该文件忽略了：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git check-ignore -v t3.a
.gitignore:5:*.a      t3.a
```

我们也可以在.gitignore 文件中写入下面一行，来表示虽然我们要求忽略所有.a 文件，但是我们不要求忽略 t3.a：

```
!t3.a
```

有时候我们希望忽略所有以. 开头的文件,同时不希望忽略.gitignore 文件,因此可以在.gitignore 文件中这么写:

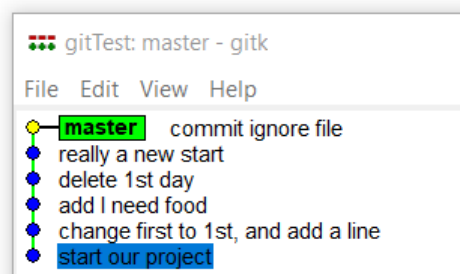
```
.*  
!.gitignore
```

### 3.3 Git 可视化工具

我们在命令行里操作了那么久，有时候感觉很不方便。或许我们需要一个可视化工具来更直观地查看文件的提交和改动（当然也是为了下一节讲分支的时候更方便）。

本质来说，Git 可视化工具是给那些不想详细了解 Git 的人准备的，可以很快上手，但我们可以将其作为一个辅助工具。在 Windows 下安装 Git 会自动安装一个叫做 Git GUI 的可视化工具。我们点击运行，然后打开现在已经存在的仓库。

我们可以在菜单栏中的 Repository 项中把 commit 的历史进行可视化：



由于我们并没有创建分支和合并，所以 commit 历史看起来很简单。等我们下一节学习了分支与合并时，可以再来试试这个工具。

### 3.4 分支的创建、合并与冲突

分支很好理解，无论是多人协作还是单人写作，经常都需要进行分支。

比如，你的主干功能是一个渲染器。有一天，你突发奇想，需要上面添加一个后处理功能，但是需要对图像输出过程进行修改；同时，你还想增加一些交互功能，则需要对渲染中间步骤进行一些修改。同时进行修改并不是那么容易，或者这两个功能是由不同的人在同一时间线操作的，因此我们就需要创建两个分支。多人协作更是如此，尽管我们还没有涉及多人协作，但我们知道，我们在构建一个大项目的时候，不可能完全是顺序进行的，很可能是多个人同步进行，我们总不能等一个人完成了他的功能再下一个人去工作吧。

Git 的分支管理非常快（SVN 非常慢，导致大家不喜欢使用上面的分支功能，想了解 SVN 可以去参考 [8]），删除、合并等操作瞬间就能完成。

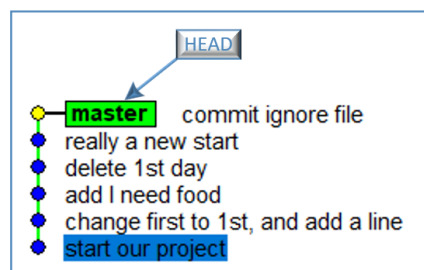


## 分支的创建

本节我们先考虑最简单的情况，就是我们新建了一个分支，然后在该分支上修改，但不修改主分支，之后把新分支上修改的内容合并到主分支上。

有人可能会问，这不是一种脱了裤子放屁的行为。其实如果不考虑后面介绍的多分支合并，这种行为的意义却不是那么大。但有时候也有用，比如，你是某个项目的发起者，你的项目主干是 master，你这周没有参与到这个项目的修改，但是别人修改了一些内容，你觉得还不错，于是你就可以把别人修改的内容合并到你的主分支中。这其实应该并不能算是多人协作，但确实也能避免很多问题（你希望你对这个项目有足够的掌控力）。

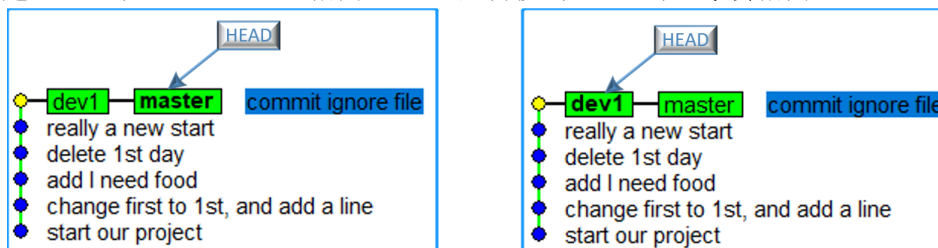
我们一开始创建一个仓库时，Git 就会创建一个主分支 master。HEAD 指向 master，而 master 指向最近的 commit。



此时我们创建一个分支，名叫 dev1，并将 HEAD 切换到 dev1 分支：

```
$ git branch dev1
$ git checkout dev1
```

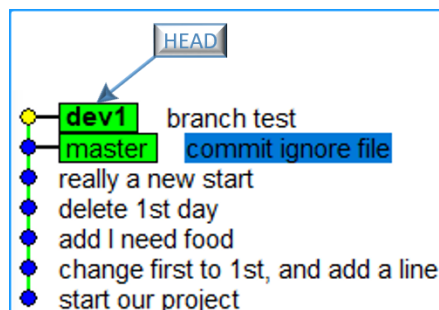
刚创建 dev1 时，HEAD 还是指向 master；转移到 dev1 时，才会指向 dev1。



上面两行代码可以用一行来代替（-b 表示创建，-c 表示切换。建议使用 switch 命令）：

```
# 方法一：
$ git checkout -b dev1
# 方法二：
$ git switch -c dev1
```

使用 git branch 命令可以查看当前分支。我们现在修改一下 readme.txt，增加或者删除一些内容，然后 add 并 commit。之后分支情况如下：



我们可以感受到，这里的分支创建其实就是移动指针，所以非常快。

## 合并到主分支

现在我们已经对新的分支的内容修改完了，想合并到主分支上，此时我们先切换到主分支上：

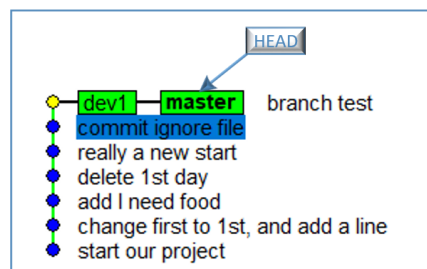
```
# 方法一：
$ git checkout master
# 方法二：
$ git switch master
```

然后把 dev 分支的内容合并到 master 分支上：

```
$ git merge dev1
```

```
Feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git merge dev1
Updating e66699d..5855afd
Fast-forward
 readme.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

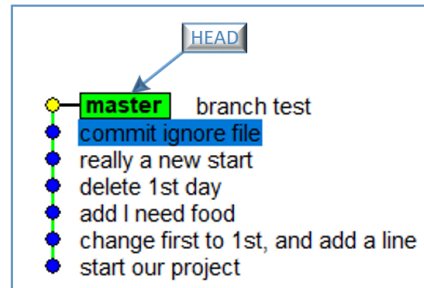
显示 Fast-forward 表示快进模式（也就是直接把 master 指向 dev1 的最新 commit），此时分支示意图为：



合并完以后我们可以删除 branch：

```
$ git branch -d dev1
```

此时就只剩下 master 分支了：



## 解决分支冲突

我们考虑创建分支 dev1 以后，master 和 dev1 都对 readme.txt 进行了修改，此时想要合并应该怎么办呢？

我们先通过 reset 命令取消 branch test 分支：

```
$ git reset --hard HEAD^
```

设此时我们的 readme.txt 文件表示如下：

```
Now let us start our project.
Now I don't need food.
I am happy.
```

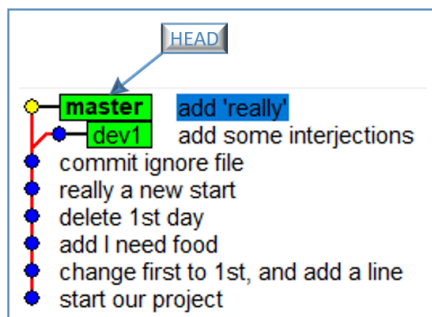
我们按照下面的顺序创建和提交一个新的分支：

```
# 创建并转到新分支上
$ git switch -c dev1
# 在文件readme.txt最后一行末尾加个 Lalala.
.....
# 添加和提交
$ git add readme.txt
$ git commit -m "add some interjections"
```

然后按照下面的顺序转到 master 分支并修改提交：

```
# 转到master分支上
$ git switch master
# 在文件readme.txt第二行的 I 和 don't 之间加个 really.
.....
# 添加和提交
$ git add readme.txt
$ git commit -m "add 'really'"
```

现在整个 commit 历史如图：



此时再调用 merge 就无法再快速合并，Git 会尽量把文件进行合并，但是两个分支都修改的文件则会冲突，我们先看看效果：

```
Feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git merge dev1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

显示 readme.txt 文件冲突，我们可以调用 git status 更详细地查看效果。由于冲突，所以合并失败，此时我们需要手动合并冲突的文件。

我们此时如果打开 readme.txt 文件，该文件就会显示不同分支的内容：

```
Now let us start our project.
<<<<<< HEAD
Now I really don't need food.
I am happy.
=====
Now I don't need food.
I am happy. Lalala.
>>>>>> dev1
```

其中，< 和 = 之间的内容表示 HEAD 指向的分支（也就是 master 分支，即我们要合并到的分支）修改的内容，= 和 > 之间的内容表示 dev1 分支（也就是我们要用来合并的分支）修改的内容。我们手动修改一下这个文件：

```
Now let us start our project.
Now I really don't need food.
I am happy. Lalala.
```

修改完以后就可以 add 到暂存区了（当 we 有多个冲突文件时，我们就要全都手动修改并且 add）：

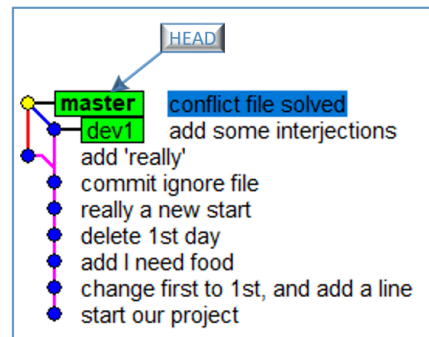
```
$ git add readme.txt
```

全都 add 完以后就可以 commit 了：

```
$ git commit -m "conflict_file_solved"
```

此时的 commit 历史图为：





除了可视化工具，分支合并情况也可以用 `git log` 查看。

简单提一句（虽然是废话，但可能可以更好地理解分支之间的关系），由于你的 HEAD 现在是指向 master 分支的，所以虽然是合并了 master 和 dev1 分支，但其实是合并到了 master 上。如果此时你运行 `reset` 命令：

```
$ git reset --hard HEAD^
```

则会沿着 master 分支往前回返，而不会回返到 dev1 分支上。

最后，我们可以像之前一样删除 dev1 分支。

对于没有冲突的合并，默认是 Fast forward 模式的快速合并方法。但是我们有时候不想直接快速合并，而是两个分支产生一个新的 commit（比如当前 master 是一个稳定的发布版本，如今我们对一个 dev1 分支增加了新的功能，确认功能稳定以后，打算合并到 master 上作为新的版本来发布，我们希望保留这个 dev1 分支开发的历史，而不是直接合并到一条线上）。我们在合并时使用 `-no-ff` 参数就可以禁用 Fast forward：

```
$ git merge --no-ff -m "merge with no-ff" dev1
```

注意，因为这次 merge 是要产生一个新的 commit，而不是仅仅简单的把 dev1 合并到 master 上，所以需要使用 `-m` 参数。

## 合并的重要策略

在多人协作时，我们可以把所有的工作和合并都在一个 dev 分支上，比如小红开的 dev1 分支和小明开的 dev2 分支，都往 dev 分支上合并，等一段时间程序稳定下来以后，就把 dev 分支合并到 master 主分支上，作为一个稳定的版本来发布。在这种工作模式下，master 只用来合并那些已经测试稳定的版本，而一些中间过程的合并版本会被合并到 dev 上。

如何掌握 Git 的使用并不困难，但是要在一个大型项目中进行尝试，才能更深刻地感受到 Git 的强大功能。

最后提一句，有时候我们在某个分支上（比如 dev2）开发的程序是一个实验性的代码，写着写着发现根本不合理，我们就想废除这个分支。如果没有合并就废除，Git 会不允许，此时可以用 `-D` 来强行删除该分支：

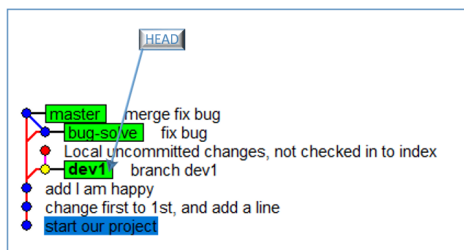
```
$ git branch -D dev2
```

### 3.5 保存与恢复现场

考虑在我们还没有完成并提交当前工作时，当前已经提交的最新版本的发现了一个 Bug，你需要立即去着手解决。此时，你需要保存一下当前工作的内容。假设我们最新提交的文件如下：

```
Now let us start our project.
Now I don't need food.
I am happy.
```

此时 commit 历史如图：



我们我们创建一个新的分支 dev1，然后在分支 dev1 上工作，再 readme.txt 加入第四行：

```
But I need to sleep ,
```

我们还没写完，但 Bug 修复的任务来了，于是我们调用 git stash 命令：

```
$ git stash
输出: Saved working directory and index state WIP on dev1: 4188d9f
      commit ignore file
```

现在再使用 git status 工作区就是干净的。然后我们转到主分支并创建新的分支：

```
$ git switch master
$ git switch -c bug-solve
```

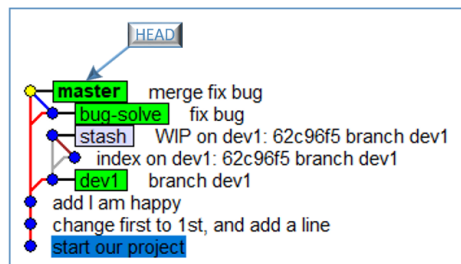
把 readme.txt 的第二行修改为：

```
Now I don't need food , but I need drink.
```

然后添加并提交，合并到 master：

```
$ git add readme.txt
$ git commit -m "fix bug"
$ git switch master
$ git merge --no-ff -m "merge fix bug" bug-solve
```

此时我们就修复完了 Bug。



现在我们希望回到之前保存的现场去工作，首先回到 dev1 分支：

```
$ git switch dev1
```

调用 `stash list` 命令查看保存的现场：

```
$ git stash list
```

```
输出: stash@{0}: WIP on master: 4188d9f commit ignore file
```

有两种恢复方法：

```
# 方法一：使用apply恢复，并使用drop删除stash
```

```
$ git stash apply
```

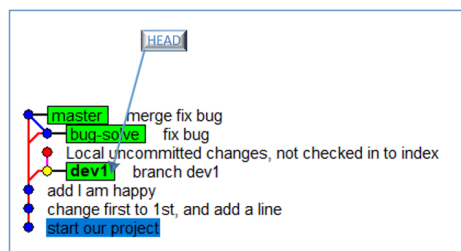
```
$ git stash drop
```

```
# 方法二：使用pop恢复并删除
```

```
$ git stash pop
```

如果你要恢复的现场不在栈顶，则可以根据序号来恢复：

```
$ git stash apply stash@{0}
```



但是现在恢复的现场也是存在之前的 Bug 的，我们可以手动重新在当前分支上修改 Bug，但是这样有时候太麻烦。或许我们也可以通过 `merge` 进行合并，但此时我们仅仅希望拷贝 `bug-solve` 分支（在我这里 ID 为 `3c701s4`）所做的修改，并不是想直接把 `master` 给 `merge` 到当前 `dev1` 分支上，Git 命令 `cherry-pick` 便是如此，它可以避免重复劳动：

```
$ git cherry-pick 3c701s4
```

```
输出: error: Your local changes to the following files would be
      overwritten by merge: readme.txt
```

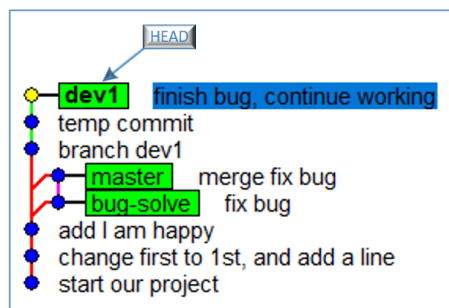
```
Please commit your changes or stash them before you merge.
```

```
Aborting
```

```
fatal: cherry-pick failed
```

当然，由于我们在 `bug-solve` 和 `dev1` 都修改了 `readme.txt`，所以造成了无法自动改动过来，需要先 `commit`。如果 `bug-solve` 修改的文件和 `dev1` 修改的文件并不同（比如 `dev1` 新建了一个文件去实现别的功能），则 `cherry-pick` 就把我们指定的 `commit` 给复制到当前分支，并自动在当前的分支上 `commit` 一次。

如果我们把当前 `dev1` 的 `readme.txt` 给 `add` 并 `commit` 了（备注可以叫“暂时提交”，`temp commit`），则再次调用 `cherry-pick` 命令就会报告 `CONFLICT` 冲突，至于怎么处理冲突，我想你之前已经掌握地很好了。最终合并完以后的结果如下：



## 3.6 小结

本章讲述的操作确实不少，不过最核心的部分还是分支操作管理。其实用到的命令不会很多，只是分支会带来理解上的复杂性（人们一般不会喜欢一堆分支的复杂结构），但只要熟悉了 `merge` 的冲突和冲突解决，其实其他过程都是很直观的。

有些时候我们可能会考虑，`commit` 历史太多则会比较混乱，那么有什么办法可以删除一些 `commit` 历史呢？答案是有的，我们在下一章讲解多人协作后再进行讲解。



# 4. 远程仓库与多人协作

4.1	远程仓库与 Github	30
4.2	添加到远程库与从远程库 clone	31
4.3	多人协作	33
4.4	rebase	36
4.5	标签管理	38
4.6	参与 Github 开源项目	39
4.7	搭建 Git 服务器	39
4.8	小结	40

本章介绍如何使用远程仓库，以及如何进行多人协作。

## 4.1 远程仓库与 Github

本地 Git 的基本操作流程已经讲解完毕，讲道理，如果不考虑多人协作，使用 Git 还不如不用，毕竟 Git 当年就是为了多人一起开发 Linux 系统而开发的。Git 真正强大的功能是可以使用远程仓库与多人协作。

同一个 Git 仓库可以分布到不同的计算机上（其实就是拷贝），在实际情况中，有一个电脑会充当服务器，保持 24 小时开机，其他人都可以从这个服务器中拷贝仓库到自己的电脑，然后再提交到服务器中，也可以从服务器中拉取其他人的提交结果。

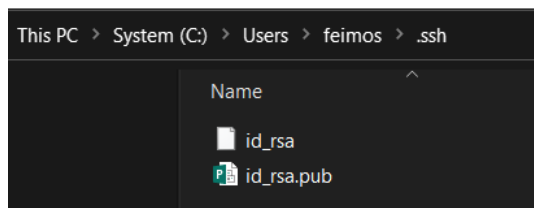
这个中央服务器我们可以自己搭建（方法也不比本地新建 Git 仓库难多少），但在自己构建服务器之前，我们需要学习和掌握一个重要的中央服务器网站——Github[9]。凡是从事计算机工程和计算机科学的人，应该都从 Github 上下载过别人的程序，我们会讲解如何使用 Github 更重要的功能。

我们需要先去 [9] 注册一个账号。由于本地 Git 仓库与 Github 之间通过 SSH 加密，所以需要设置。我们打开 Git Bash，不用切换目录，直接执行：

```
$ ssh-keygen -t rsa -C "12345678@163.com"
```

之后会有一些设置，我们无需手动设置，直接一路回车即可：一开始直接回车会默认在 id\_rsa 上生成 ssh key，之后再按回车表示不设置密码（一般个人项目不需要那么严格），之后重复密码时也直接回车。

之后我们在用户的主目录里找到.ssh 目录，里面有两个文件：



其中 `id_rsa` 是私钥文件，不能泄露给别人；`id_rsa.pub` 表示公钥 (public)，可以告诉别人。我们把公钥里面的字符串复制下来（包括前面的 `ssh-rsa` 这几个字符）。

登陆 Github，打开 settings 的 SSH Keys 页面（由于 Github 网站也会一直变化，不同时期的 Github 网页页面也有所不同），然后新建一个 SSH Key，并把复制的内容粘贴上去，并起一个名字（例如，homePC。Github 支持多个 SSH Keys，这样你就可以在不同的电脑上推送你的修改了，而中央服务器都会识别为是你的修改）。通过 SSH 协议，Github 就可以知道当有新的内容被提交上来时，是你自己的提交的，而不是别人在冒充你。

## 4.2 添加到远程库与从远程库 clone

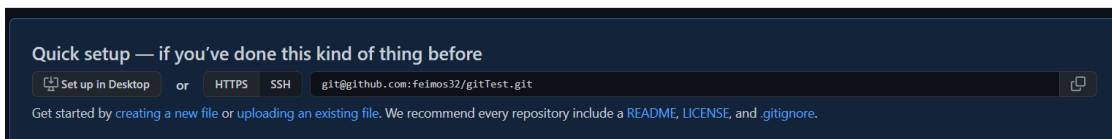
现在我们已经有了远程库，我们希望把本地的内容推送到远程库上，这样远程库就可以作为备份，又可以让其他人通过该仓库来协同工作。

由于 Github 网页界面时不时会发生一些改变，所以这里我们不再粘贴网页操作图片，而当你已经掌握了 Git 的本地用法之后，Github 的使用其实就会变得很容易。

### 添加到远程库

我们在 Github 网站上点击 Create a new repository。

我们可以直接从 Github 上新建一个 Git 仓库：取一个名字，跟我们本地的 Git 仓库名一样，叫 `gitTest` 就行。我们可以选择一些初始化项，例如添加 README file，以及添加忽略文件 `.gitignore`，我们先不考虑这些初始化项，而是直接创建一个空白的仓库，然后能看到如下界面：



我们可以看到 SSH，这样就能把本地的 Git 仓库关联到远程仓库。登录到本地 Git，然后执行命令（注意改成你的用户名）：

```
$ git remote add origin git@github.com:你的用户名/gitTest.git
```

如果想要取消关联可以调用：

```
$ git remote remove origin
$ git remote rm origin
```

如果想查看本地 Git 关联了哪些远程 Git，可以调用：

```
$ git remote -v
```

origin 是我们给远程库取的名字，在 Git 中属于默认的叫法。之后我们就可以把本地库的内容推送到远程库上了（注意要保证你此时的用户名和邮箱和远程 git 一致），在本地 Git Bash 执行命令（-u 表示远程 Git 还没有内容，我们把本地的 master 分支推送给远程 master 分支）：

```
$ git push -u origin master
```

如果顺利的话，就会把本地内容同步到 Github 上。

## 报错与解决

执行 push 命令时可能会报错如下：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git push -u origin master
ssh: connect to host github.com port 22: Connection timed out
fatal: could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

这时有多种可能，比如你的公钥私钥不匹配、你当前的邮箱和 Github 注册的邮箱不一致等。你需要首先把前面的步骤重新进行一次（根据你的邮箱生成密钥，添加公钥到 Github 上等），如果还是报当前错误，则可能是连接的问题。执行下面的命令来检查：

```
$ ssh -T git@github.com
```

出现连接超时，说明就是连接的问题：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ ssh -T git@github.com
ssh: connect to host github.com port 22: Connection timed out
```

我们在主目录的.ssh 目录下新建一个 config 文件 [10]，然后输入以下内容并保存：

```
Host github.com
User 你注册github的邮箱
Hostname ssh.github.com
PreferredAuthentications publickey
IdentityFile ~/.ssh/id_rsa
Port 443
```

之后再保存，然后继续执行上面的命令应该就没有问题了：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ ssh -T git@github.com
The authenticity of host '[ssh.github.com]:443 ([140.82.113.36]:443)' can't be e
stablished.
ECDSA key fingerprint is SHA256:p2QAMXNIC1TjYweIottrvc98/R1BUFWu3/LiyKgufQM.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '[ssh.github.com]:443,[140.82.113.36]:443' (ECDSA) to
the list of known hosts.
Hi feimos32! You've successfully authenticated, but GitHub does not provide shel
l access.
```

此时我们就可以再次尝试提交文件：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/gitTest (master)
$ git push -u origin master
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 12 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (9/9), 741 bytes | 370.00 KiB/s, done.
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To github.com:feimos32/gitTest.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

显示 successfully 就说明成功提交。

这里只列举了可能出现的一种状况，实际情况五花八门，说不定你会遇到其他更难解决的问题，查了不少网上的资料也很难找到解决方案，这个时候只能耐心去分析和测试。就像我第一次推送 Git 时没有遇到任何问题，但是这次在实验室 PC 上推送就遇到了上述问题。

## 从远程库 clone 到本地库

我们使用 cd 命令进入 Developer 文件夹（就是放你的 Git 仓库的文件夹），然后调用如下命令：

```
$ git clone git@github.com:feimos32/gitTest.git
```

就会把远程 Git 的仓库拷贝到当前文件夹下，之后你就可以自由操作了。

## 4.3 多人协作

多人协作我们换一种讲解顺序，先描述整个工作流程，然后再详细地介绍命令。

我们在本地工作后，会把本地的提交推送到远程 Git 上，推送时需要指定推送的分支。哪些分支需要推送，哪些不需要是需要考量的，例如 bug 分支在修复好 bug 之后就合并到主分支上了，因此 bug 分支没有必要推送。

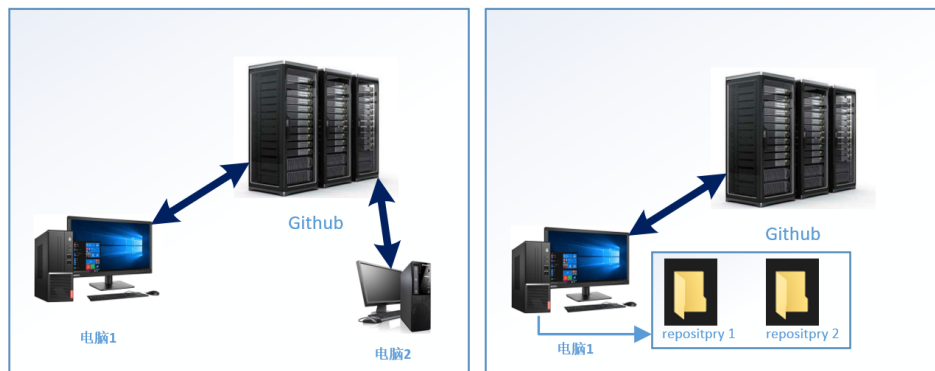
在多人协作时，master 分支用来存放稳定的版本，dev 分支用来存放一些团队工作和修改的结果。现在假设远程服务器的 SSH keys 会包含你们团队所有人的账号（你们这样就可以都能直接参与到整个项目的维护）。远程 clone 时，一般默认只会 clone 远程 Git 的 master 分支，然后需要自己创建远程 origin 的 dev 分支到本地，然后修改这个 dev 分支，并把 dev 分支 push 到远程。

如果有两个人都提交了对同一个文件的修改，就会造成冲突，此时我们需要后提交的人把一个提交的人的 commit 给 pull 到自己本地，然后解决冲突，再 push 到远程 Git 上去。

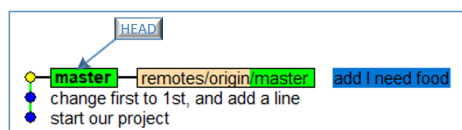
## 准备工作

现在开始进行实验。因为一个团队都能自由提交，这与你一个人在本地建立两个仓库来模拟两个人其实是同样的，所以我们不用两个电脑来进行模拟：





实验前的准备工作：我们 Github 的 gitTest 上只有一个 master 分支，与 Github 的 origin 仓库关联，上面有一个 readme.txt 文件：



在本地 Git 仓库创建 dev 分支，修改并 commit，然后推送到 Github 上：

```
$ git switch -c dev
# 在readme.txt上添加一些内容。
.....
$ git add readme.txt
$ git commit -m "add some words"
$ git push origin dev
```

刷新 Github 网页，然后我们就在 Github 网页看到我们的 gitTest 仓库有了两个分支。

## clone 分支

我们在自己的电脑上建立 myGit1 和 myGit2 两个文件夹。

在 Git Bash 中先进入 myGit1，然后把 Github 仓库 clone 到 myGit1 中；然后在 Git Bash 中进入 myGit2，把 Github 仓库 clone 到 myGit2 中。最后，Git Bash 进入 myGit1/gitTest：

```
# clone 的命令
$ git clone git@github.com:feimos32/gitTest.git
```

此时调用 git branch 并看不到主分支。我们创建远程 origin 的 dev 分支到本地：

```
$ git checkout -b dev origin/dev
输出：Switched to a new branch 'dev'
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

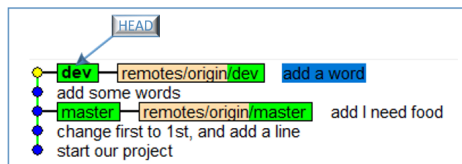
这样就把本地 dev 分支与远程 dev 分支进行了连接。我们再次修改一下 readme.txt 并 commit，然后 push 到远程服务器上：

```
# 在readme.txt上添加一个单词。
.....
```

```
$ git add readme.txt
$ git commit -m "add_a_word"
$ git push origin dev
```

可以在 Github 网页刷新以后查看新分支的最新修改。

此时，myGit1/gitTest 本地仓库的内容就是：



## 多人协作

现在，该到了使用 myGit2 目录下的仓库的时刻了。myGit2 模拟的是另一个团队队员，他在你修改提交到远程 Git 之前就 clone 了 dev。

现在你操作：

# 在 readme.txt 的末尾添加新的一行。

```
.....
$ git add readme.txt
$ git commit -m "add_a_line"
$ git push origin dev
```

可惜推送失败：

```
feimos@DESKTOP-A3KMQGO MINGW64 /d/developer/myGit2/gitTest (dev)
$ git push origin dev
warning: Permanently added the ECDSA host key for IP address '[140.82.113
.35]:443' to the list of known hosts.
To github.com:feimos32/gitTest.git
 ! [rejected]        dev -> dev (fetch first)
error: failed to push some refs to 'github.com:feimos32/gitTest.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository push
ing
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
```

原因是远程的进行了更新，你需要调用 pull 命令来把远程的更新合并到自己本地的工程上，然后再提交：

```
git pull
```

但是此时出现了冲突（如果没有其他错误的话），这是因为我们都修改了 readme.txt，我们需要手动合并这些冲突。这个过程想必你已经非常熟悉了：

# 解决冲突

```
.....
$ git add readme.txt
$ git commit -m "solve_conflict"
$ git push origin dev
```

现在我们可以看到 Github 网页上的代码就是我们提交的版本。

如果我们调用 `git pull` 来将远程 Git 的更新 `pull` 到本地时显示输出：

```
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.
git pull <remote> <branch>
If you wish to set tracking information for this branch you can do so
with:
git branch --set-upstream-to=origin/<branch> dev
```

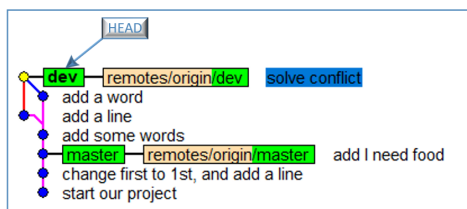
意思是它并不知道你的 `dev` 分支是与远程 Git 的哪个分支相连接的，所以根据提示，我们需要这样操作：

```
git branch --set-upstream-to=origin/dev dev
```

这样就能建立本地分支与远程分支的关联。

## 4.4 rebase

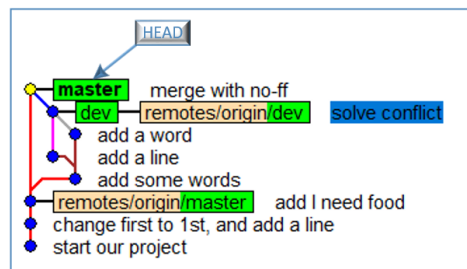
我承认这是本书最难的一节，因为它也是与分支处理有关的（一遇到分支就很麻烦）。我们当前在 `myGit2` 目录下的 `gitTest` 的 `commit` 历史如下：



我们把 `dev` 合并到 `master` 上，并推送到远程服务器：

```
$ git switch master
$ git merge --no-ff -m "merge with no-ff" dev
$ git push origin master
```

得到当前的合并结果：



时间一久，你可能会觉得整个提交历史非常混乱，你希望整个提交历史是一条干净的直线，这个操作叫做 `rebase`。`rebase` 可以删除一些提交历史，也可以在 `push` 之前进行把提交历史变干净。我们继续开始进行实验。

## 实验过程

把 myGit2 目录下的 gitTest 复制到 myGit1 目录，作为我们实验的开始。我们先在 myGit2/gitTest 目录里进入 dev 分支，然后修改和提交一些内容，并 push 到远程服务器：

```
$ git switch dev
# 在readme.txt上删除一个单词
.....
$ git add readme.txt
$ git commit -m "delete a word"
$ git push origin dev
```

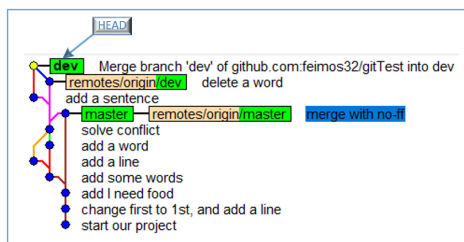
然后我们的 Git Bash 进入 myGit1/gitTest，修改并提交：

```
$ git switch dev
# 在readme.txt上加一句话
.....
$ git add readme.txt
$ git commit -m "add a sentence"
$ git push origin dev
```

此时会提示有新的版本被提交到 dev 了，我们需要先更新本地的内容。老规矩，pull 一下以后解决冲突：

```
$ git pull
```

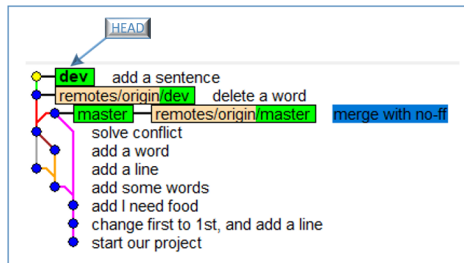
此时本地 myGit1/gitTest 的提交历史就是：



我们调用 git rebase 命令：

```
$ git rebase
```

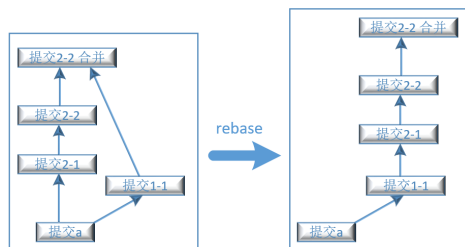
我们再次查看提交历史，就可以看到下面的情况：



也就是说这个分叉被取消了，我们可以把这么一条干净的提交线 push 到远程。

## 原理讲解

rebase 其实就是变基，过程如下：



这样就能将整个提交历史变为一条干净的线。

rebase 也可以删除一些历史 commit 等其他功能，这些内容就不在这里进行讲解了，展开讲会占用比较多的篇幅。我们或许会在一些进阶教程中更深入地讲解。

## 4.5 标签管理

我们有时候想去使用某个版本来测试，我们知道，定位一个版本的 id 就是 commit ID，但是它是一堆字符串，很长，也很难记住。Git 可以让我们自定义一些标签，一个标签和一个版本关联起来，想恢复到某一个版本时直接使用标签即可。

我们进入 master 分支，调用命令来打一个标签：

```
# 打标签
$ git tag 标签 你需要打标签的commit ID
# 例如：
$ git tag v1.0 6r3c37
# 查看标签
$ git tag
```

如果没有给出你要打标签的 commit ID，就会给当前 HEAD 指向的分支的当前 commit 打上标签。调用 show 命令就可以查看标签：

```
$ git show v1.0
```

我们可以给标签做一个说明（-a 表示版本，-m 表示说明）：

```
$ git tag -a v1.0 -m "a tag test" 6r3c37
```

调用 git log 命令回顾所有的 commit 历史，并把所有 commit 历史排成一行，这样就能看到所有提交历史的 ID。

```
$ git log --pretty=oneline --abbrev-commit
```

一些标签的操作：

```
# 删除标签
$ git tag -d v1.0
# 推送到远程
```



```
$ git push origin v1.0
# 把全部标签推送到远程
$ git push origin --tags
```

删除已经推送到远程的标签：

```
# 在本地删除标签
$ git tag -d v1.0
# 在远程删除标签
$ git push origin :refs/tags/v1.0
```

## 4.6 参与 Github 开源项目

在本节的实验中你需要两个 Github 账号，一个用来建立开源项目，另一个作为你的账号，来参与到这个开源项目中来。

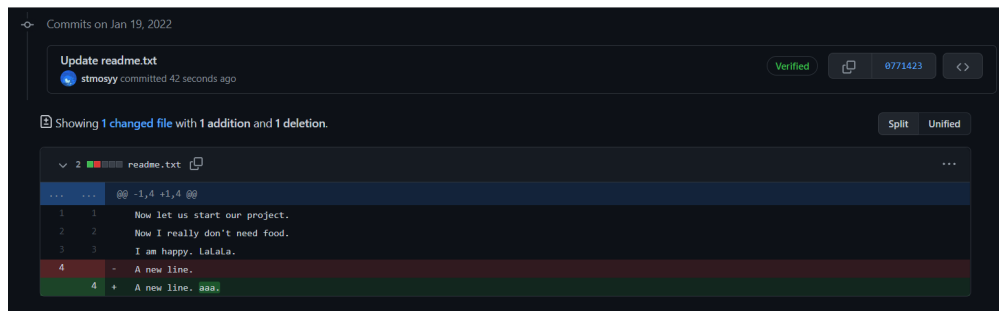
我们前面注册的账号 A 发布的内容可以作为一个开源项目，现在我们登录到另一个账号 B。然后找到账号 A 的开源项目 gitTest，然后点击 Fork，这样就 Fork 到自己这里了。我们 clone 仓库一定要从自己的账号下 clone，否则没法推送修改。

我们换一台电脑，配置账号 B 的本地 Git 环境，包括 SSH key。我们在 Git Bash 上进入某个目录，把仓库 clone 到本地：

```
$ git clone git@github.com:账号B的用户名/gitTest.git
```

然后修改并推送到 Github 账号 B 下的 gitTest。

如果你修改了很多内容，决定希望开源项目官方来接受你的修改，则可以发起一个 pull request。然后点击 create a pull request，之后就可以 pull 自己的修改给项目发起人：



然后你再登陆账号 A 去接受这个 pull 就可以了。

## 4.7 搭建 Git 服务器

一般的大型公司和实验室都会自己搭建 Git 服务器或者 SVN 服务器（我所在的实验室搭建的是 SVN），我们或许希望自己搭建一个 Git 服务器。

关于这方面的内容的网站博客其实有很多，这里暂时不进行介绍。一方面是因为你自己使用的话没必要建立 Git 服务器（或许你甚至都不需要用 Git），另一方面是因为注册购买服务器或者自己找个电脑作为服务器都各自不同，整体这一套内容写下来也不容易。

## 4.8 小结

---

学习 Git 的使用和学习编程语言有很多相似之处——实验出真理。但我们也没必要像编程语言一样去不断训练自己的 Git 使用熟练度，当我们开始使用 Git 去管理自己的项目时，一定会遇到各种各样的问题，在解决这些问题时，就会愈加熟练，也会产生更深的理解。



# Bibliography

- [1] <https://en.wikipedia.org/wiki/Git>
- [2] <https://www.cnblogs.com/leeyongbard/p/9777498.html>
- [3] <https://www.liaoxuefeng.com/wiki/896043488029600>
- [4] <https://git-scm.com/downloads>
- [5] [https://blog.csdn.net/weixin\\_37909391/article/details/84641899](https://blog.csdn.net/weixin_37909391/article/details/84641899)
- [6] <https://www.jianshu.com/p/c6927e80a01d>
- [7] <https://github.com/github/gitignore>
- [8] <https://www.runoob.com/svn/svn-tutorial.html>
- [9] <https://github.com/>
- [10] <https://blog.csdn.net/nightwishh/article/details/99647545>