

流行框架

目录

目录.....	2
第 1 章 工作流程.....	4
第 2 章 Shell 和 vi.....	4
2.1 什么是 shell.....	4
2.2 shell 分类.....	5
2.3 认识 bash 这个 shell.....	5
2.4 vi 编辑器.....	7
第 3 章 版本控制.....	11
3.1 关于版本控制.....	11
3.2 本地版本控制系统.....	11
3.3 集中式版本控制系统.....	12
3.4 分布式版本控制系统.....	13
第 4 章 Git.....	15
4.1 Git 工作原理.....	15
4.2 Git 安装.....	15
4.3 Git 本地仓库.....	16
4.3.1 Git 基础.....	16
4.3.2 Git 分支.....	21
4.4 Git 远程仓库.....	27
4.5 gitHub 和 gitLab.....	30

4.6 冲突解决.....	37
4.7 Git 高级.....	38
第 5 章 项目发布	41

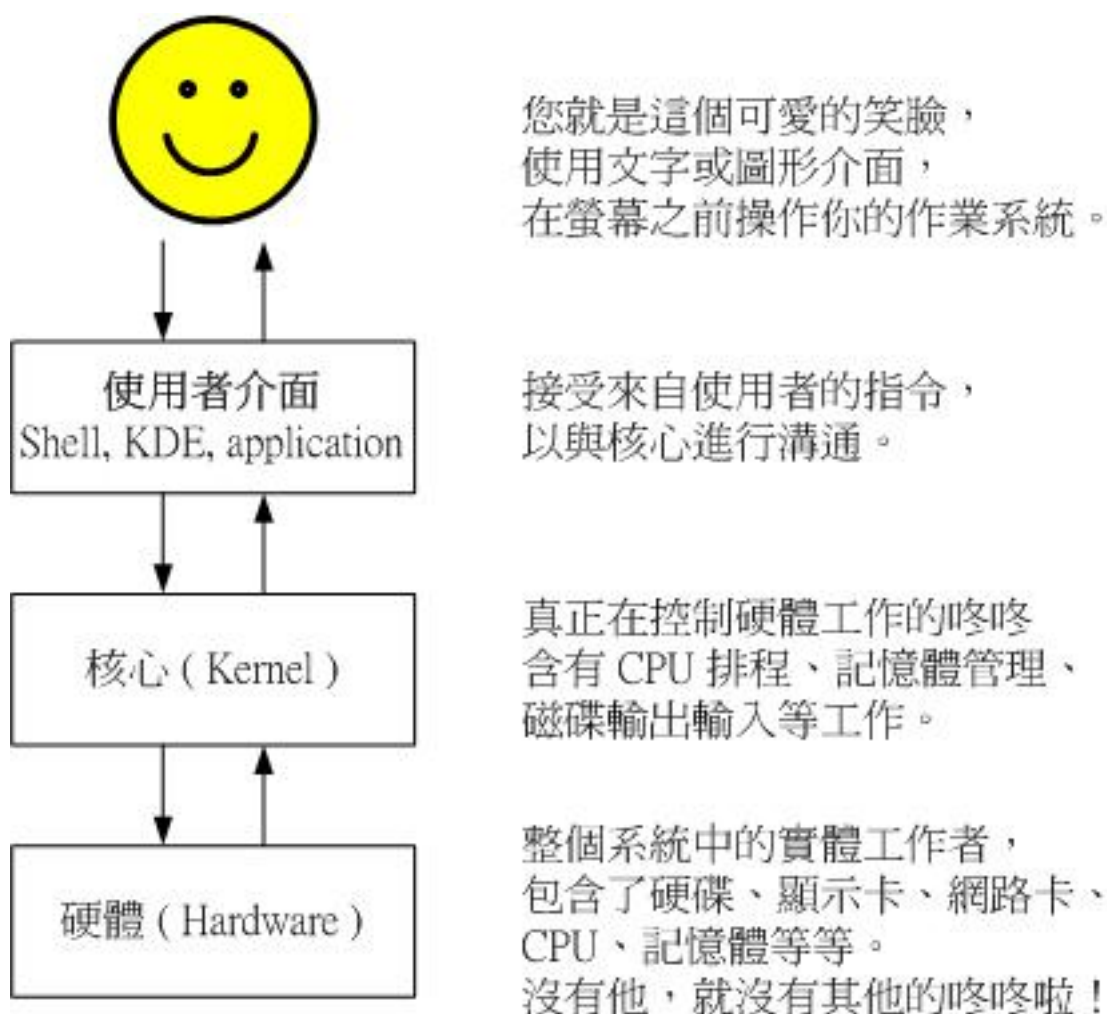
第 1 章 工作流程

通过下图了解实际开发过程中的工作流程

第 2 章 Shell 和 vi

2.1 什么是 shell

在计算机科学中，Shell 俗称壳，用来区别于 Kernel（核），是指“提供使用者使用界面”的软件（**命令解析器，也是一种编程语言**）。它类似于 DOS 下的 command 和后来的 cmd.exe。**它接收用户命令，然后调用相应的应用程序。**



2.2 shell 分类

1、图形界面 shell：通过提供友好的可视化界面，调用相应应用程序，如 windows 系列操作系统，Linux 系统上的图形化应用程序 GNOME、KDE 等。

2、命令行 shell：通过键盘输入特定命令的方式，调用相应的应用程序，如 windows 系统的 cmd.exe、Windows PowerShell，Linux 系统的 Bourne shell (sh)、**Bourne Again shell (bash)**等。

汇智网的 linux shell 教程

<http://www.hubwiz.com/class/56d906a0ecba4b4d31cd28ef>

linux 鸟哥私房菜

慕课网上的视频教程

2.3 认识 bash 这个 shell

各个 shell 的功能都差不多，Linux 默认使用 bash，所以我们主要学习 bash 的使用。

1、bash 命令格式

命令 [-options] [参数]，如：tar zxvf demo.tar.gz

查看帮助：命令 --help

2、bash 常见命令

pwd (**P**rint **W**orking **D**irectory) 查看当前目录

cd (**C**hange **D**irectory) 切换目录，如 cd /etc

ls (**L**ist) 查看当前目录下内容，如 `ls -al`

mkdir (**M**ake **D**irectory) 创建目录，如 `mkdir blog`

touch 创建文件，如 `touch index.html`

cat 查看文件全部内容，如 `cat index.html`

more less 查看文件，如 `more /etc/passwd`、`less /etc/passwd`

rm (**r**emove) 删除文件，如 `rm index.html`、`rm -rf blog`

rmdir (**R**emove **D**irectory) 删除文件夹，只能删除空文件夹，不常用（`rm -r temp/` 递归删除 `rm -rf temp/` 强行递归直接删除很危险）

mv (**m**ove) 移动文件或重命名，如 `mv index.html ./demo/index.html`

cp (**c**opy) 复制文件，`cp index.html ./demo/index.html`

head 查看文件前几行，如 `head -5 index.html`

tail 查看文件后几行 `-n -f`，如 `tail index.html`、`tail -f -n 5 index.html`（实时，错误日志）

tab 自动补全，连接两次会将所有匹配内容显示出来（筛选）

history 查看操作历史

ssh 远程登录，如 `ssh root@gitlab.study.com`

> 和 **>>** 重定向，如 `echo hello world! > README.md`，`>` 覆盖 `>>` 追加

wget 下载，如 `wget https://nodejs.org/dist/v4.4.0/node-v4.4.0.tar.gz`

tar 解压缩，如 `tar zxvf node-v4.4.0.tar.gz`

curl 网络请求，如 `curl http://www.baidu.com`

whoami 查看当前用户

| 管道符

grep 匹配内容，一般结合管道符使用

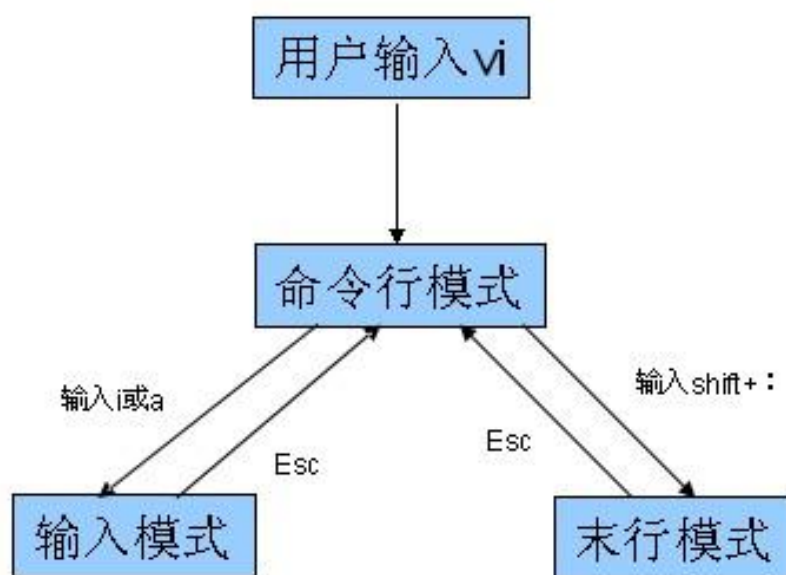
2.4 vi 编辑器

如同 Windows 下的记事本，vi 编辑器是 Linux 下的标配，通过它我们可以创建、编辑文件。它是一个随系统一起安装的文本编辑**软件**。

1、三种模式

vi 编辑器提供了 **3 种模式**，分别是命令模式、插入模式、底行模式，每种模式下用户所能进行的操作是不一样的。

3 种模式的切换如下图所示：



通过上图我们发现，输入模式是不能直接切换到末行模式的，必须要先切回到命令模式（按 ESC 键）

2、使用 vi 编辑器

- a) 打开/创建文件， vi 文件路径
- b) **底行模式** :w 保存，:w filename 另存为

- c) 底行模式 :q 退出
- d) 底行模式 :wq 保存并退出
- e) 底行模式 :e! 撤销更改，返回到上一次保存的状态
- f) 底行模式 :q! 不保存强制退出
- g) 底行模式 :set nu 设置行号
- h) 命令模式 ZZ（大写）保存并退出
- i) 命令模式 u 撤销操作，可多次使用
- j) 命令模式 dd 删除当前行
- k) 命令模式 yy 复制当前行
- l) 命令模式 p 粘贴内容
- m) 命令模式 ctrl+f 向前翻页
- n) 命令模式 ctrl+b 向后翻页
- o) 命令模式 i 进入编辑模式，当前光标处插入
- p) 命令模式 a 进入编辑模式，当前光标后插入
- q) 命令模式 A 进入编辑模式，光标移动到行尾
- r) 命令模式 o 进入编辑模式，当前行下面插入新行
- s) 命令模式 O 进入编辑模式，当前行上面插入新行

当我们处在编辑模式的情况下，和我们在 Windows 编辑器的使用相似。

2.5 SSH

SSH 是一种网络协议，用于计算机之间的加密登录。（防止传输过程中被捕获截取）

SSH 只是一种协议，存在多种实现，既有商业实现，也有开源实现。本文针对的实现是 **OpenSSH**，它是自由软件，应用非常广泛。

如果要在 Windows 系统中使用 SSH，会用到另一种软件 **PuTTY**，我们后面用到的 Git 客户内置集成了 SSH

```
ssh user@host
```

加密技术

对称加密算法在加密和解密时使用的是同一个密钥；而**非对称加密算法**需要两个**密钥**来进行加密和解密，这两个密钥是**公开密钥**（public key，简称公钥）和私有密钥（private key，简称私钥）。

工作原理

公钥和私钥是成对出现，可以通过 `ssh-keygen -t rsa` 来创建，既可以通过密钥来加密数据，也可以通过私钥来加密数据，如果是公钥进行的数据加密，只能与之相对应的私钥才可以解密，相反如果以私钥进行的数据加密，则只能与之对应的公钥才可以将数据进行解密，这样就可以提高信息传递的安全性。

免密码登录

我们可以将本地机器上的公钥保存到特定的远程计算机上，这样当我们再次登录访问这台远程计算机时就可以实现免密码登录了。

这部分具体实现细节，参照我的演示有个印象就可以了。

在 linux 上如何部署 node 环境

1、linux 上安装 node 环境的三种方式

<http://www.xitongzhijia.net/xtjc/20150202/36680.html>

(1) 编译好的文件安装

```
yum -y install gcc gcc-c++ openssl-devel
```

```
wget https://nodejs.org/dist/v4.4.4/node-v4.4.4-linux-x64.tar.xz
```

```
tar -zxvf node-v4.4.4-linux-x64.tar.xz
```

```
ln -s /download/node-v4.4.4-linux-x64.tar.xz/bin/node /usr/local/bin/node
```

```
ln -s /download/node-v4.4.4-linux-x64.tar.xz/bin/npm /usr/local/bin/npm
```

```
node -v
```

(2) 用第三方的工具安装

```
sudo yum install epel-release
```

```
sudo yum install nodejs
```

```
sudo yum install npm
```

用 pm2 部署 node 服务，只是 node 的一个模块包

Nginx 反向代理

<https://www.douban.com/note/314200231/>

```
npm install pm2 -g
```

第 3 章版本控制

3.1 关于版本控制

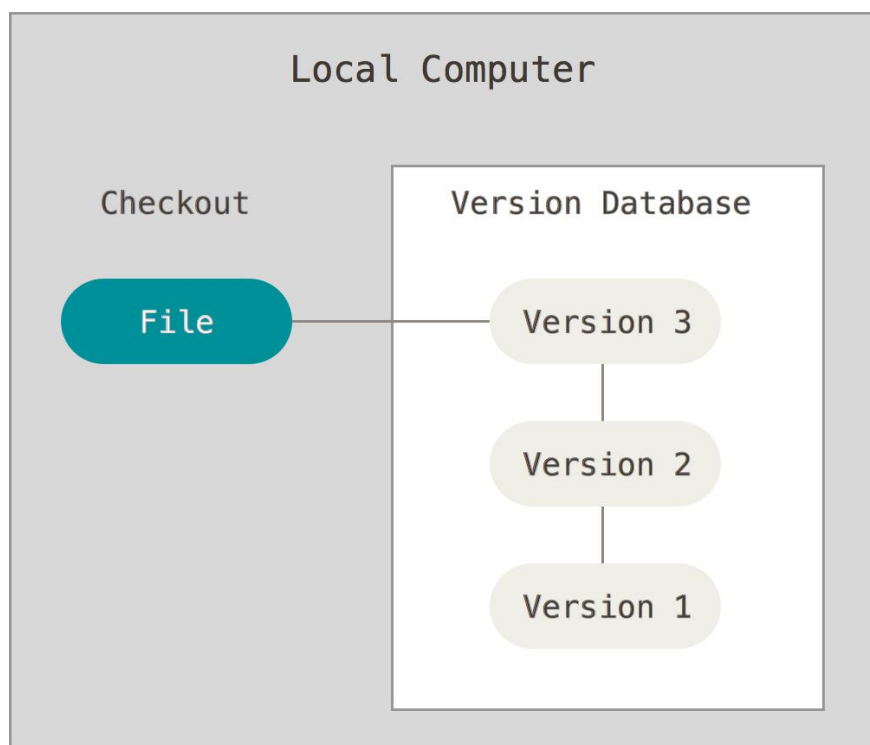
版本控制（Version Control Systems）是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。这个系统可以自动帮我们备份文件的每一次更改，并且可以非常方便的恢复到任意的备份（版本）状态。

举例：我们通常都是手动的重命名一个文件进行备份的，index.html 改成 index1.html 或者 index.html.bak 等形式，然后这种方式对于单个文件我们还能够管理，但是对于整个项目而言，就会成为噩梦了！！！我们不得不借助于软件来实现。

实现版本控制的软件有很多种类，大致可以分为本地版本控制系统、集中式版本控制系统、分布式版本控制系统。

3.2 本地版本控制系统

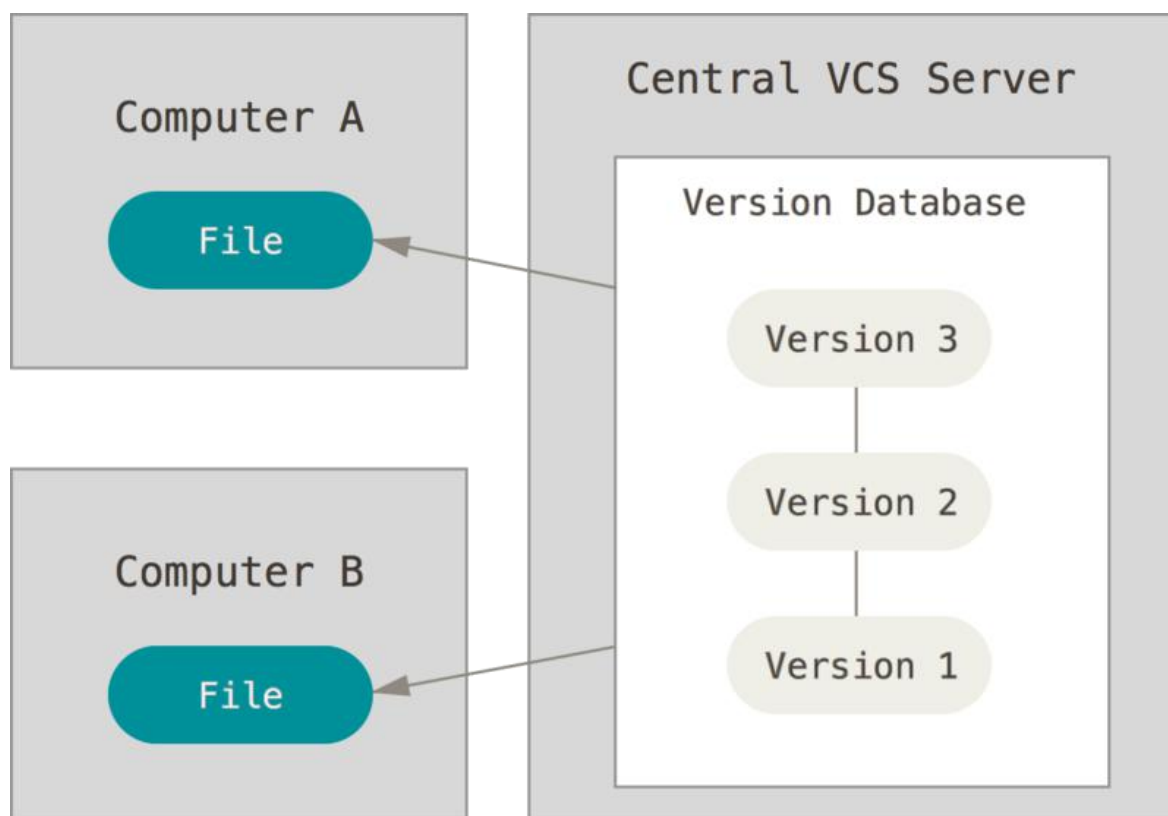
借助软件我们可以记录下文件的每一次修改，如下图所示，文件被修改后，记录下了 3 个版本，这样我们通过版本控制系统（软件）便可以非常方便的恢复到任意版本。



这种类型的版本控制系统，功能比较单一，比如很难实现多人协同开发，所以现在几乎很少使用了。

3.3 集中式版本控制系统

实际开发环境，一个项目通常是由多人协作共同完成的，如何让在不同系统上的开发者协同工作成了亟待解决的问题，集中式版本控制系统便应运而生了。它通过单一的集中管理的服务器，保存所有文件的修订版本，协同工作的开发者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。其代表为 SVN，如下图所示。

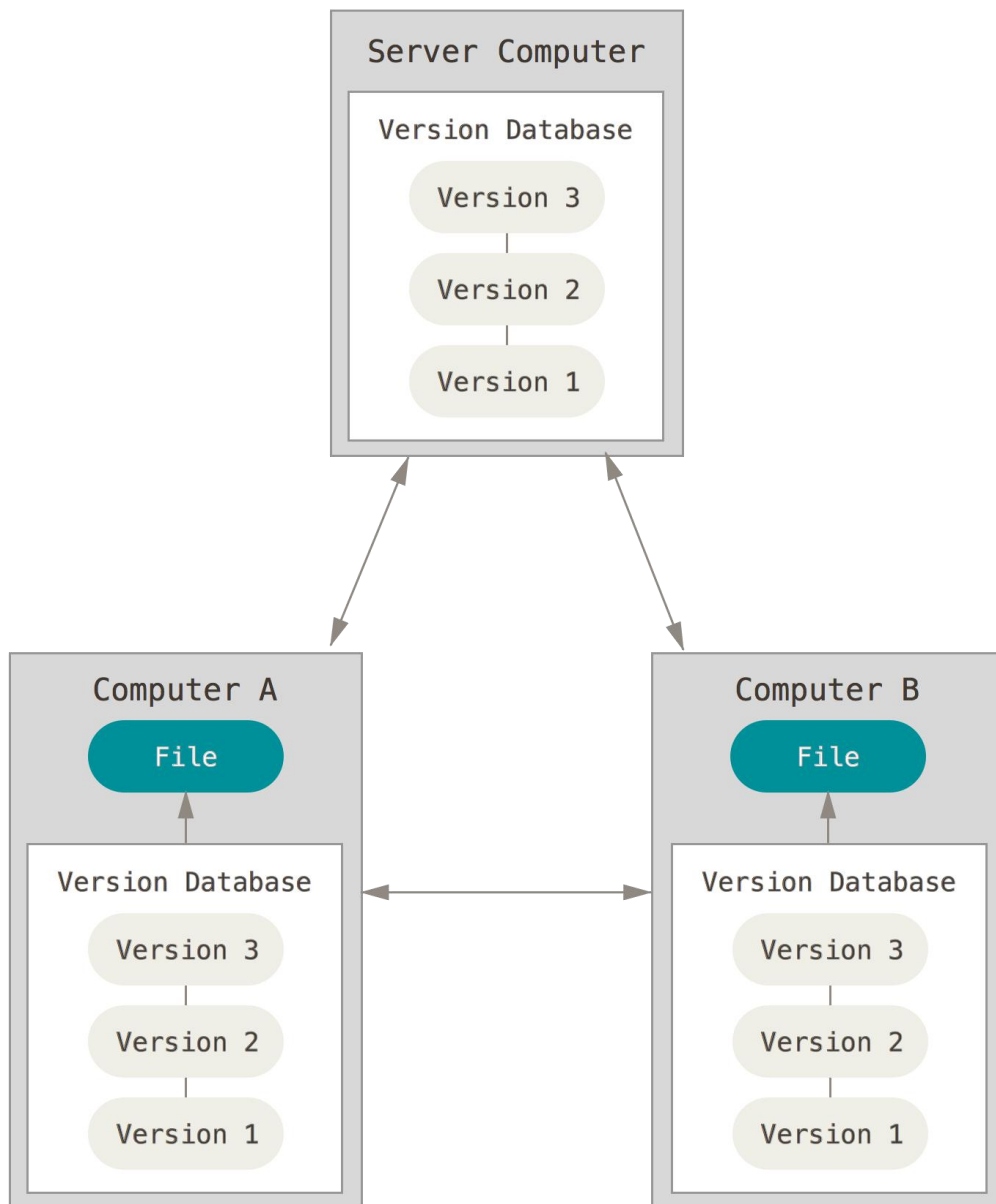


这种方式很好解决了多人协同开发的问题，但是也有一个弊端，如果集中管理的服务器出现故障，**将会导致数据（版本）丢失的风险**，另外协同开发者从集中服务器中更新数据时，**严重依赖网络**，如果网络不佳，或者在家就拿不到代码了，也给开发带来诸多不便。

3.4 分布式版本控制系统

分布式版本控制系统，则不需要中央服务器，每个协同开发者都拥有一个**完整的版本库**，这么一来，任何协同开发者用的服务器发生故障，事后都可以用其它协同开发者本地仓库恢复。

由于版本库在本地计算机，也便**不再受网络影响**了。如果要将本地的修改，推送给其它协同开发者，还需要一台**共享服务器**，所有开发者通过这台共享服务器提交和更新数据。如下图所示。



分布式版本控制系统弥补了前面两种版本控制系统的缺陷，成为了版本控制的首选方案。其代表就是 **Git**。

Git tfs svn vss

第 4 章 Git

4.1 Git 安装

Window 安装

<http://git-scm.com/download/win> 下载 Git 客户端软件，和普通软件安装方式一样。

Linux 安装

CentOS 发行版：sudo yum install git（现在都是集成的）

Ubuntu 发行版：sudo apt-get install git

Mac 安装

打开 Terminal 直接输入 git 命令，会自动提示，按提示引导安装即可。

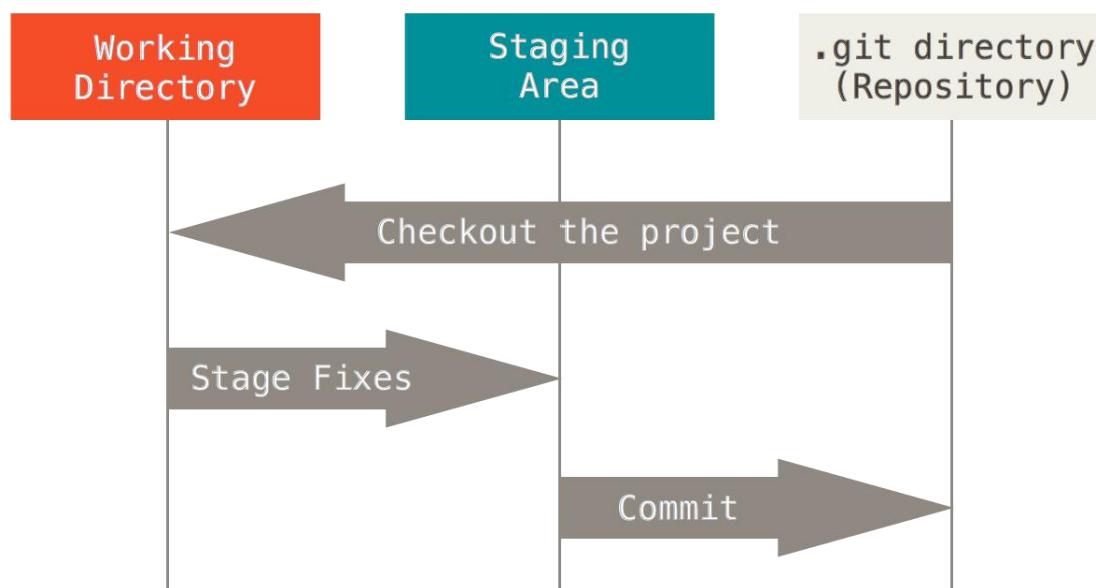
4.2 Git 工作原理

为了更好的学习 Git，我们必须了解 Git 管理我们文件的 **3 种状态**，分别是已提交（committed）、已修改（modified）和已暂存（staged），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。

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”（Index），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。

4.3 Git 本地仓库

4.3.1 Git 基础

命令行方式：任意目录（建议开发目录）右键 > Git Bash Here

1、配置用户

`git config --global user.name "自己的名字"`

`git config --global user.email "自己的邮箱地址"`

`--global` 配置当前用户所有仓库

`--list` 查看配置信息

`--system` 配置当前计算机上所有用户的所有仓库

2、初始化仓库

我们如果想要利用 git 进行版本控制，需要将现有项目初始化为一个仓库，或者将一个已有的使用 git 进行版本控制的仓库克隆到本地。

a) git init

```
Botue@Botue-PC MINGW64 /e/360
$ git init
Initialized empty Git repository in E:/360/.git/
```

git init 只是创建了一个名为.git 的隐藏目录，这个目录就是存储我们历史版本的仓库，ls -al 可以查看。

b) 假如公司已有了项目用了 Git，那我们就利用克隆

git clone 仓库地址

```
Botue@Botue-PC MINGW64 /e
$ git clone git@github.com:Botue/copy.git
Cloning into 'copy'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

执行完这个命令，会在当前目录下生成一个 copy 目录，这个便是已有一个项目。

3、查看文件状态

git status 可以检测当前仓库文件的状态

```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git status
On branch master

Initial commit

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

        copy.html
        css/
        images/
        index.html
        js/

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

4、添加文件到暂存区

git add 文件 “*” 或 -A 代表所有

```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git add -A
warning: LF will be replaced by CRLF in js/jquery.fullPage.min.js.
The file will have its original line endings in your working directory.

Botue@Botue-PC MINGW64 /e/360 (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   copy.html
        new file:   css/copy.css
        new file:   css/main.css
        new file:   images/1.jpg
```

将所有的文件添加到暂存区等待提交

我们可以再次查看仓库的当前状态

标明了当前的仓库状态

放到暂存区的文件被标记成了绿色，等待提交。

5、提交文件

git commit -m '备注信息'

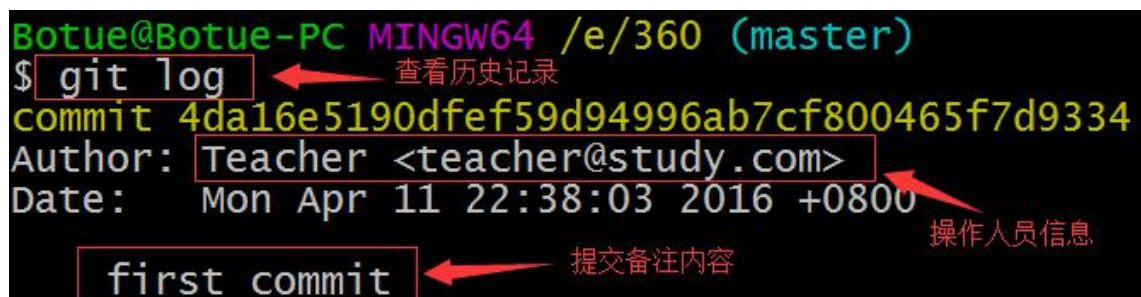
```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git commit -m 'first commit'
[master (root-commit) 4da16e5] first commit
warning: LF will be replaced by CRLF in js/jquery.fullPage.min.js.
The file will have its original line endings in your working directory.
41 files changed, 1002 insertions(+)
create mode 100644 copy.html
create mode 100644 css/copy.css
create mode 100644 css/main.css
create mode 100644 images/1.jpg
create mode 100644 images/2.jpg
create mode 100644 images/3.jpg
create mode 100644 images/4.jpg
create mode 100644 images/5.jpg
create mode 100644 images/circle.png
```

将所有更改提交到本地仓库

将暂存区被标记成绿色的文件，全部提交到本地仓库存储。

6、查看提交历史

git log



```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git log
commit 4da16e5190dfef59d94996ab7cf800465f7d9334
Author: Teacher <teacher@study.com>
Date: Mon Apr 11 22:38:03 2016 +0800

    first commit
```

Annotations in the image:

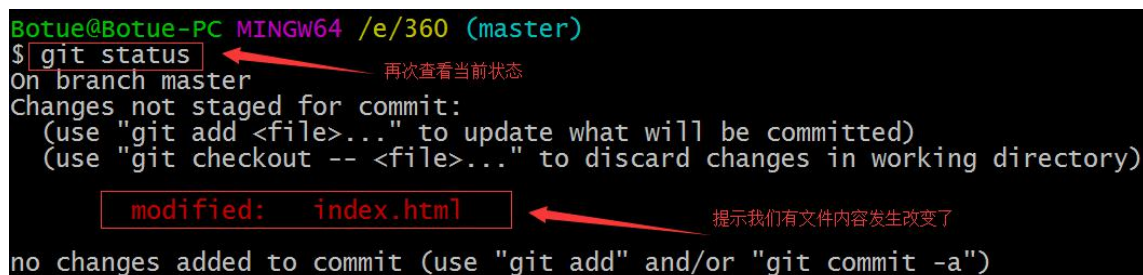
- 查看历史记录 (View history record) points to `git log`
- 提交备注内容 (Commit message content) points to `first commit`
- 操作人员信息 (Operator information) points to `Author: Teacher <teacher@study.com>`

我们可以查看到一次提交记录，4da16e5190 代表这次提交的唯一 ID，一般称为 SHA 值。傻？

这时我们对 `index.html` 文件做修改

7、再次检测仓库文件状态

git status



```
Botue@Botue-PC MINGW64 /e/360 (master)
$ 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:   index.html

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

Annotations in the image:

- 再次查看当前状态 (Check current status again) points to `git status`
- 提示我们有文件内容发生改变了 (Prompt that file content has changed) points to `modified: index.html`

被修改过的文件被标记成了红色，等待重新添加到暂存区。

8、重新添加暂存区然后提交

git add index.ht ml

git commit -m 'add some code'


```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git add index.html
Botue@Botue-PC MINGW64 /e/360 (master)
$ git status
on branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html
Botue@Botue-PC MINGW64 /e/360 (master)
$ git commit -m 'add some code'
[master 4d66e22] add some code
1 file changed, 1 insertion(+)

```

再次将文件添加到暂存区里

查看状态

放到了暂存区，颜色变成了绿色

一次新的提交，提交到本地仓库

9、再次查看历史

git log 可查到所有提交历史

```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git log
commit 4d66e22c5169a5307e219af3e49dfc721c79ff9d
Author: Teacher <teacher@study.com>
Date:   Mon Apr 11 22:45:35 2016 +0800

    add some code
commit 4da16e5190dfef59d94996ab7cf800465f7d9334
Author: Teacher <teacher@study.com>
Date:   Mon Apr 11 22:38:03 2016 +0800

    first commit

```

记录了第二次的提交信息

记录了第一次提交的信息

这时可以查看到两次提交历史。

这时关掉所有目录甚至关机！

10、恢复上一次提交的状态

git reset --hard 4da16e5190dfef 查看 index.html 发现回到了没有修改的状态

git log 再次查看发现提交历史只有一个了

```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git reset --hard 4da16e5190dfef
HEAD is now at 4da16e5 first commit

Botue@Botue-PC MINGW64 /e/360 (master)
$ git log
commit 4da16e5190dfef59d94996ab7cf800465f7d9334
Author: Teacher <teacher@study.com>
Date: Mon Apr 11 22:38:03 2016 +0800

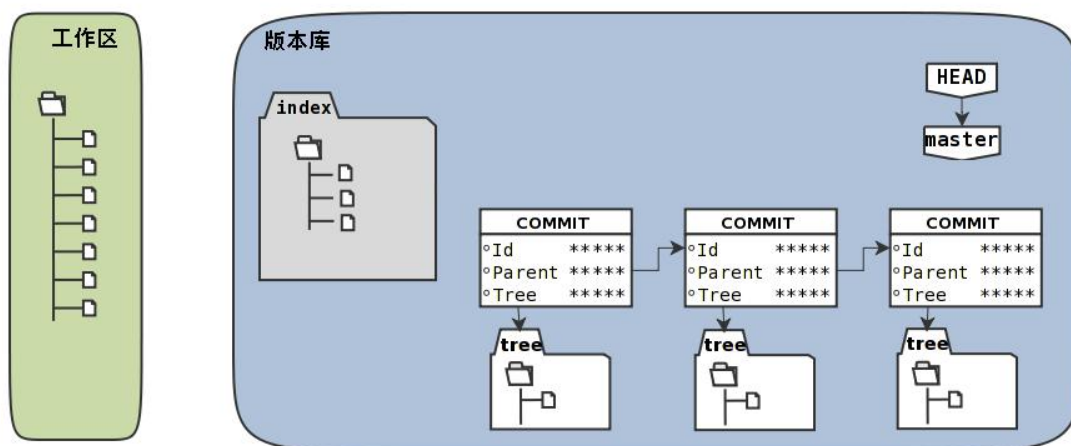
    first commit
```

回退到某个版本

历史记录也发生了改变

这时我们非常容易就回到了曾经的一个历史版本。

仓库示意图



4.3.2 Git 分支

在我们的现实开发中，需求往往是五花八门的，同时开发个需求的情况十分常见，比如当你正在专注开发一个功能时，突然有一个紧急的 BUG 需要你来修复，这个时候我们当然是希望在能够保存当前任务进度，再去修改这个 BUG，等这个 BUG 修复完成后再继续我们的任务。如何实现呢？

通过创建分支来解决实际开发中类似的问题。

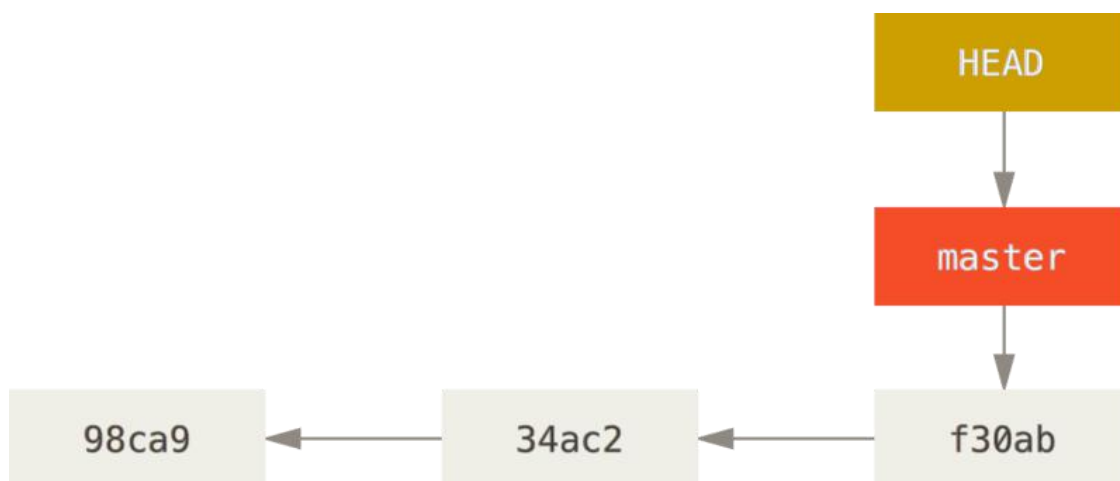
在 Git 的使用过程中一次提交便代表一个历史记录（版本），并且会生成一个唯一的字符串，如下图

```
$ git log
commit 4da16e5190dfef59d94996ab7cf800465f7d9334
Author: Teacher <teacher@study.com>
Date: Mon Apr 11 22:38:03 2016 +0800

first commit
```

唯一的不重复的字符串

通常利用这个串来代表某一个历史版本（实际使用只取前面几位就可以）如下图所示：



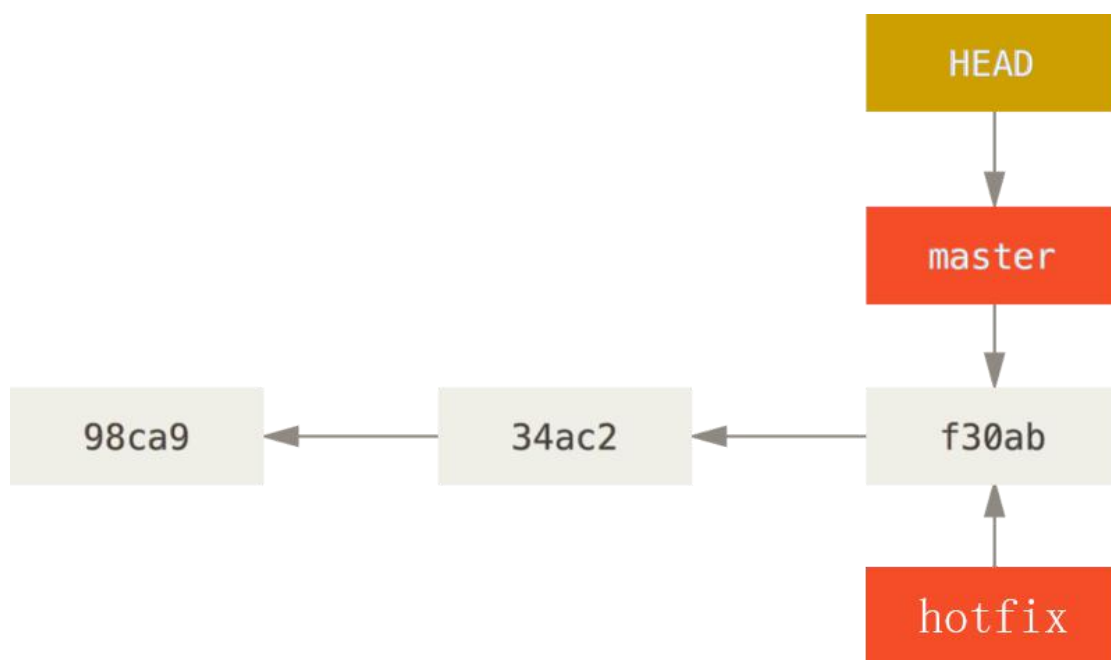
当我们在初始化仓库的时候，Git 会默认帮我们创建了一个 master 的分支，并且 HEAD 默认指向了 master 末端。

我们也可以创建自己的分支

1、创建分支

git branch hotfix

新的分支会在当前分支原有历史版本的结点上进行创建，我称其为子分支
如下图



新建的子分支会继承父分支的所有提交历史。

图 4-1

2、切换分支

Git branch 查看分支

git checkout hotfix

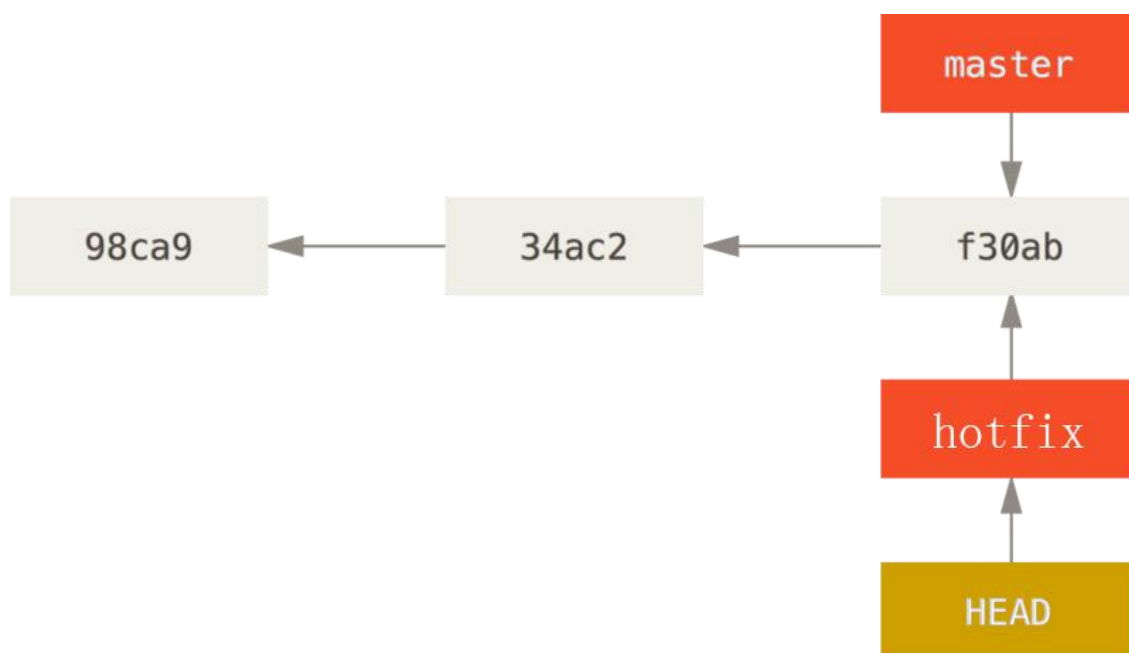


图 4-2

我们发现 HEAD 现在又指向了 hotfix 的末端。

3、再次提交操作

```
git add -A
```

```
git commit -m 'add some code for hotfix'
```

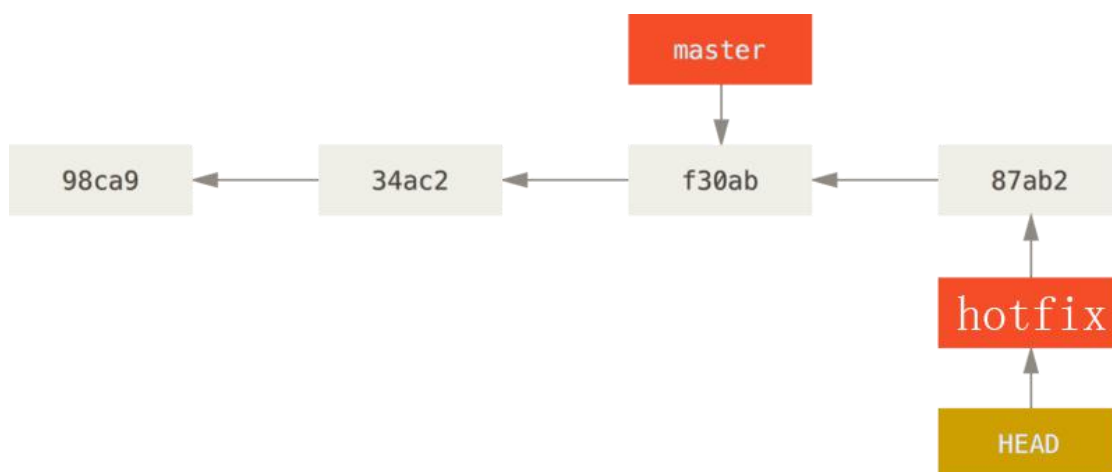


图 4-3

这次的提交历史版本就会记录在 hotfix 这个分支上了，并且 HEAD 伴随 hotfix 在移动。

4、当我们再次切回到 master 时

```
git checkout master
```

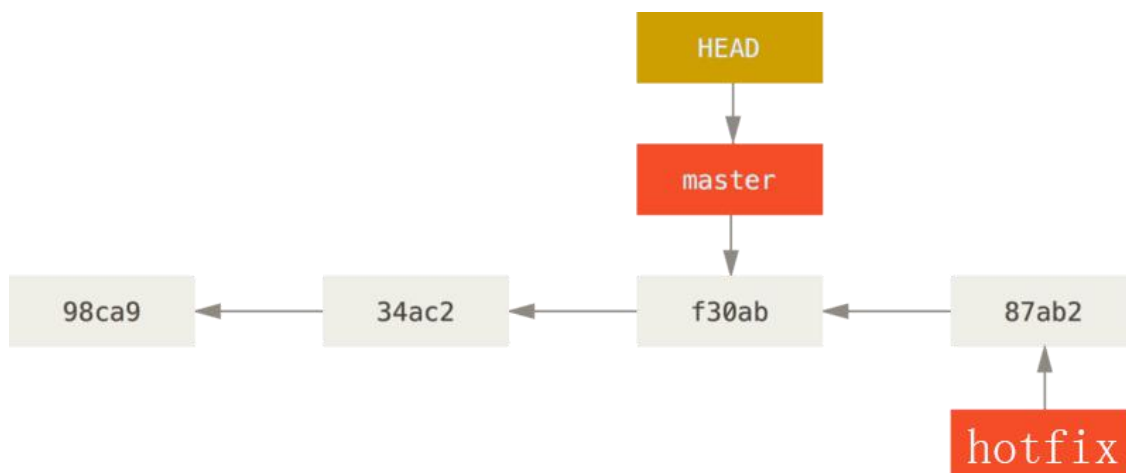


图 4-4

当我们切换回 master 后，HEAD 指向了 master 分支的末端，并且我们观察发现我们的文件内容还是原来的“模样”。

5、继续之前的开发

```
git add -A
```

```
git commit -m 'add some code for master'
```

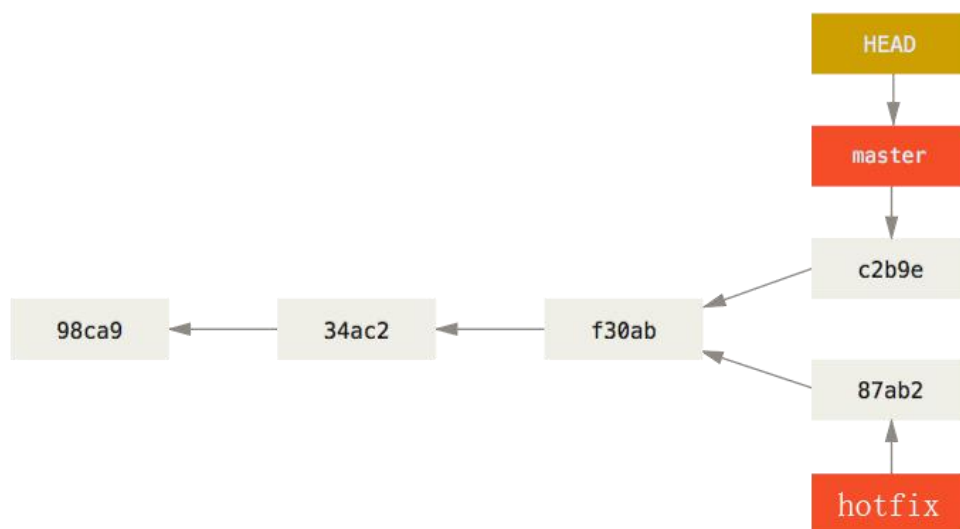


图 4-5

总结：当我们 `git checkout branchname` 时，HEAD 会自动指向对应分支的末端，工作目录中的源码也会随之发生改变。

这个时候我们就在 hotfix 这个分支上修复了这个 BUG，而我们原来在 master 分支上的操作并未受到影响。

思考一个问题：

现在 master 这个分支上是否包含了 hotfix 的修复呢？

实际上从图 4-5 可以看出这时的 master 分支并没有包含有 hotfix 的修复。

6、合并（融合）分支

`git checkout master`

`git merge hotfix` 注意使用位置

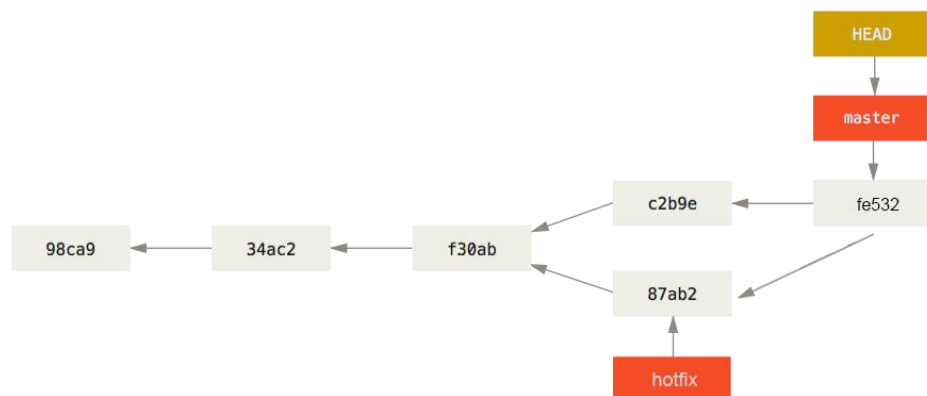


图 4-6

这时 master 会有两个父结点了，master 便包含了 hotfix 里的修复了

7、删除分支

`git branch -d hotfix`

这时用来修复 BUG 创建的 hotfix 分支已经没有用处了，我们可以将它删除。



图 4-7

4.4 Git 远程仓库

通过上面学习我们可以很好的管理本地版本控制了，可是如果我们下班回到家里突然来了灵感觉得有部分代码可以优化，如果能接着公司电脑上的代码继续写**该有多好呀！**另一种情形，假设项目比较大，不同的功能模块由不同的开发人员完成，不同模块儿之间又难免会依赖关系，这时如果我们的代码互相合并（融合）**该有多好呀！**所有模块开发完毕后，需要整合到一起，要能做到准确无误**该有多好呀！**

借助一个远程仓库，大家可以共享代码、历史版本等数据，便可以解决以上遇到的所有问题，在学习远程仓库前我们先来学习 `git clone path` 这个命令。

3、创建共享仓库

Git 要求共享仓库是一个以 **.git 结尾** 的目录。

`mkdir repo.git` 创建以 .git 结尾目录

`cd repo.git` 进入这个目录

git init --bare 初始化一个共享仓库，也叫裸仓库 **注意选项--bare**

```
Botue@Botue-PC MINGW64 /e
$ mkdir repo.git
Botue@Botue-PC MINGW64 /e
$ cd repo.git/
Botue@Botue-PC MINGW64 /e/repo.git
$ git init --bare
Initialized empty Git repository in E:/repo.git/
```

创建一个目录，准备当仓库用
进入到准备当仓库的目录中
初始化一个共享仓库，也叫裸仓库

这样我们就建好了一个共享的仓库，但这时这个仓库是一个**空的仓库**。

4、向共享仓库共享内容

进入到本地的仓库 360

cd 360

git push ../repo.git master

```
Botue@Botue-PC MINGW64 /e
$ cd 360
Botue@Botue-PC MINGW64 /e/360 (master)
$ git push ../repo.git master
Counting objects: 46, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (46/46), done.
Writing objects: 100% (46/46), 722.48 KiB | 0 bytes/s, done.
Total 46 (delta 4), reused 0 (delta 0)
To ../repo.git
* [new branch]      master -> master
```

进入到本地仓库
将本地仓库的内容推送到共享仓库中去

5、从共享仓库里取出内容

git clone ../repo.git demo

```
Botue@Botue-PC MINGW64 /e
$ git clone repo.git/ demo
Cloning into 'demo'...
done.
Botue@Botue-PC MINGW64 /e
$ ls
$RECYCLE.BIN/  copy/  demo/  repo.git/
360/          CSS3/  HTML5/  system volume information/
```

从远程仓库获取360项目，并存储到demo目录中
目录中的确多出来了一个demo目录

通过 repo.git 共享仓库，我们轻松得到了一个 360 的副本

6、通过 demo 仓库向 repo.git 共享内容

进入到 demo 里，我们做一些修改

```
cd demo
```

```
git push ../repo.git master
```

```
git push ../repo.git/ master
```

```
Botue@Botue-PC MINGW64 /e/demo (master)
$ git push ../repo.git master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 321 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To ../repo.git
    4da16e5..9664d8d  master -> master
```

通过 demo 仓库，将修改推送到共享仓库

7、在 360 仓库从 repo.git 获取共享的内容

```
cd 360
```

```
git pull ../repo.git master
```

```
Botue@Botue-PC MINGW64 /e/360 (master)
$ git pull ../repo.git/ master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ../repo
 * branch                master      -> FETCH_HEAD
Updating 4da16e5..9664d8d
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
```

从共享仓库中获取最新的修改

奇迹似乎发生了，我们轻松的将 demo 仓库里的内容，通过 repo.git 共享给了 360 仓库。

惊喜不断，问题也总是不断，我们发现我们这个共享的仓库只是放到了本地的，其它人是没有办法从我们这个共享仓库共享内容的！！！！

然而现实是，办法总是有的！！！！

我们把这个共享的仓库放到一台远程服务器上，问题不就解决了吗？

4.5 gitHub 和 gitLab

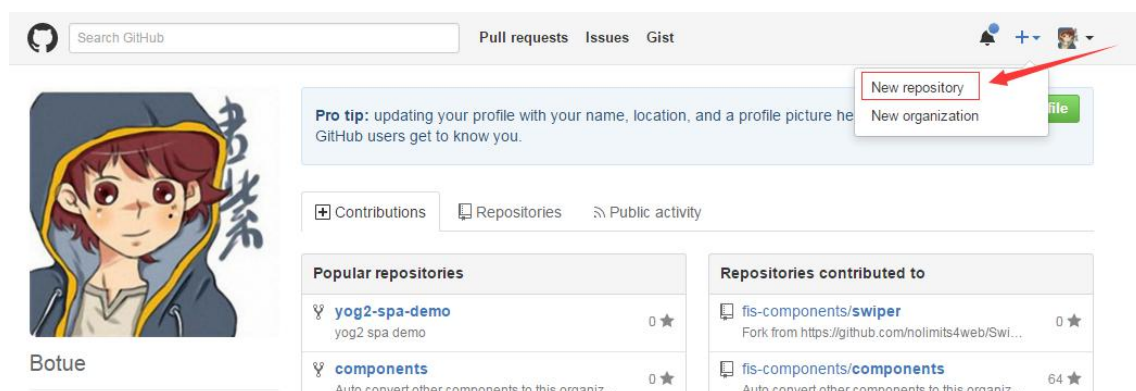
如果我们熟悉服务器的话，我们完全可以将上述的步骤在我们的远程服务器上进行操作，然后再做一些登录权限的设置，就可非常完美的搭建一个共享服务器了。其实为了更好的管理我们的仓库，一些第三方机构开发出了 **Web 版仓库管理程序**，通过 Web 界面形式管理仓库。

gitHub 关于它的名气与意义，大家可以自行查阅，我们这里介绍它的使用

1、注册账号并完善资料

自行注册略过

2、创建共享仓库



3、填写仓库资料

起个任意名字，中文我没有试过，请起个英文的吧！
但是自己不能拥有两个同名的仓库

做个自我介绍呗！

Great repository names are short and memorable. Need inspiration? How about **musical-waffle**?

创建仓库时需不需要初始化一结内容自行决定

仓库的性质，公开或私有的
公开的就是所有人都可以看到并得到你仓库的内容
私有的则不是谁都可以拿到你仓库内容
私有仓库是要交保护费的！！！！

忽略文件与此仓库遵循的开源协议
任意选择

Create repository

4、共享仓库

仓库地址，支持https和ssh两种协议
建议使用ssh协议

如果你是一个新的仓库，即本地没有任何仓库
选择这里的步骤

如果你本地已经有一个仓库了，选择这里的步骤

```
git remote add origin git@github.com:Botue/repo.git
git push -u origin master
```

远程地址特别长，我们可以给他起一个别名

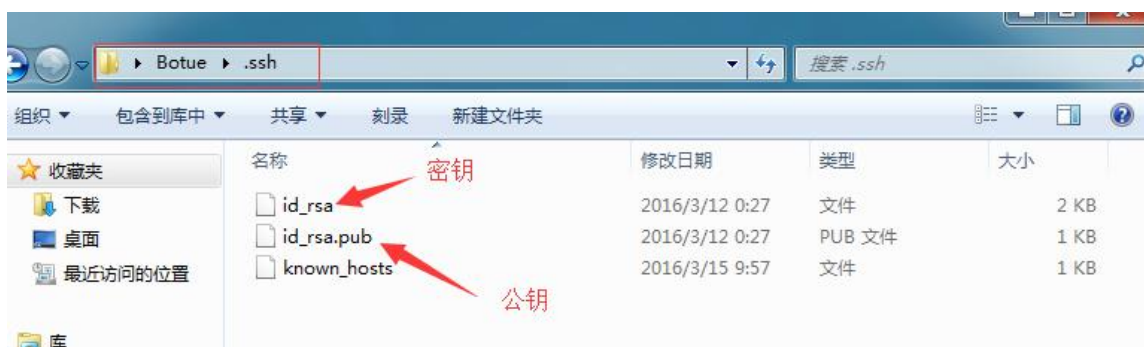
```
git remote add origin git@github.com:Botue/repo.git
```

这样 origin 就代表 git@github.com:Botue/repo.git

当我们通过 `git clone` 从共享仓库获内容时，会自动帮我们添加 `origin` 到对应的仓库地址，例如：`git clone git@github.com:Botue/repo.git` 会自动添加 `origin` 对应 `git@github.com:Botue/repo.git`

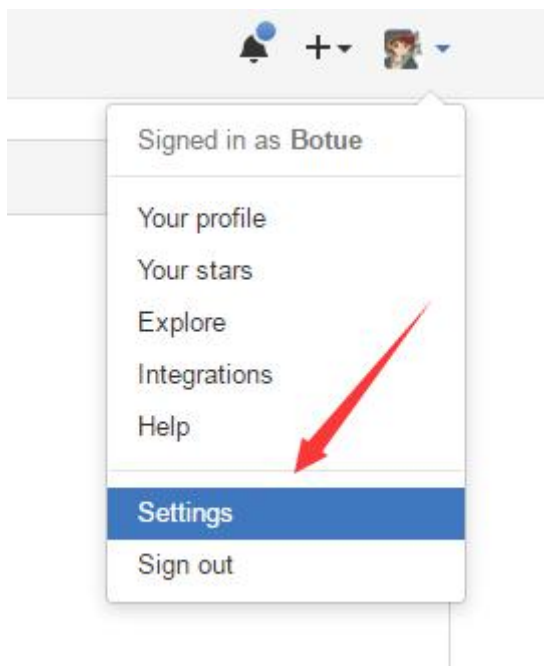
5、生成密钥

`ssh-keygen -t rsa` 然后一路回车，这里会在当前用户生成了一个 `.ssh` 的文件夹

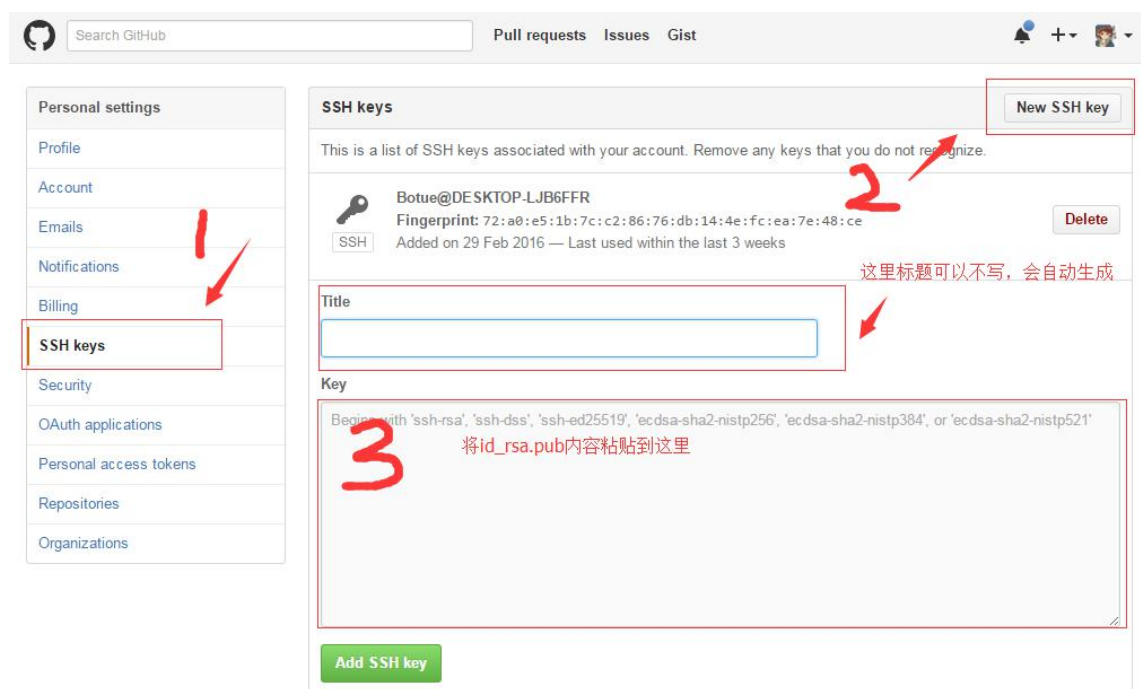


将 `id_rsa.pub` 公钥的内容复制

打开 `gitHub` 的个人中心



打到 `SSH keys`



到此我们便可以通过 **gitHub** 提供的 Web 界面来管理我们的仓库了。

我们发现通过 **gitHub** 管理仓库实在是太方便了，可是只能免费使用公开仓库，自己公司的代码当然不能公开了，可是私有仓库又是需要交“保护费”的，无耐国人还是比较喜欢免费的，网络界总是有很多雷锋的，比如 **gitLab!!!**

gitLab 也是一个可以通过 Web 界面管理仓库的网站程序，我们可以把它架设到公司自己的服务器上，实现仓库私有化，这也是大部分公司通常采用的方法，其使用方法与 **gitHub** 十分相似。

我将闲置电脑配置成了一台服务器，上面架设了 **gitLab** 程序，我们接下来的练习全部会在 **gitLab** 上进行演示。

省略很多内容.....

Gitpage

1、新建项目命名严格要求，固定的

用户名.+github.io

2、直接把项目放进去就可以访问了，不过只能放静态页，放 **node** 服务就不行了

Github 部署项目

1、注册 github 账号，并登陆，配置电脑公钥

(1) 生成公钥 `ssh-keygen -t rsa`

(2) github 上的公钥可以配置很多人的，也可以配置自己多台笔记本上的公钥

2、在 github 上建立共享仓储

(1) 配置仓储名称，是否公开，开源协议，就能得到一个 **ssh** 的服务器仓储地址

3、配置远端服务器地址

```
$ git remote add origin git@github.com:shiguqing/gitdemo.git
```

4、推送

Git push

4.6 命令汇总

`git config` 配置本地仓库

常用 `git config --global user.name`、`git config --global user.email`

`git config --list` 查看配置详情

`git init` 初始一个仓库，添加 `--bare` 可以初始化一个共享（裸）仓库

`git status` 可以查看当前仓库的状态

`git add` “文件” 将工作区中的文件添加到暂存区中，其中 `file` 可是一个单独的文件，也可以是一个目录、“*”、`-A`

`git commit -m '备注信息'` 将暂存区的文件，提交到本地仓库

`git log` 可以查看本地仓库的提交历史

`git branch` 查看分支

`git branch` “分支名称” 创建一个新的分支

`git checkout` “分支名称” 切换分支

`git checkout -b developeer` 他健并切到 `developer` 分支

`git merge` “分支名称” 合并分支

`git branch -d` “分支名称” 删除分支

`git clone` “仓库地址” 获取已有仓库的副本

`git push origin` “本地分支名称:远程分支名称”将本地分支推送至远程仓库，

`git push origin hotfix`（通常的写法）相当于

`git push origin hotfix:hotfix`

`git push origin hotfix:newfeature`

本地仓库分支名称和远程仓库分支名称一样的情况下可以简写成一个，即 `git push` “仓库地址” “分支名称”，如果远程仓库没有对应分支，将会自动创建

`git remote add` “主机名称” “远程仓库地址” 添加远程主机，即给远程主机起个别名，方便使用

`git remote` 可以查看已添加的远程主机

`git remote show` “主机名称” 可以查看远程主机的信息

4.7 GitLab 介绍

没错，Git 非常强大！

但是，如果我们的分支不加以规范管理，也有可能适得其反！

- 1、不要有太多的树杈（子分支）
- 2、要有一个“稳定分支”，即 master 分支不要轻易被修改
- 3、要有一个平行分支，保证 master 分支的稳定性

图形化 git 操作工具

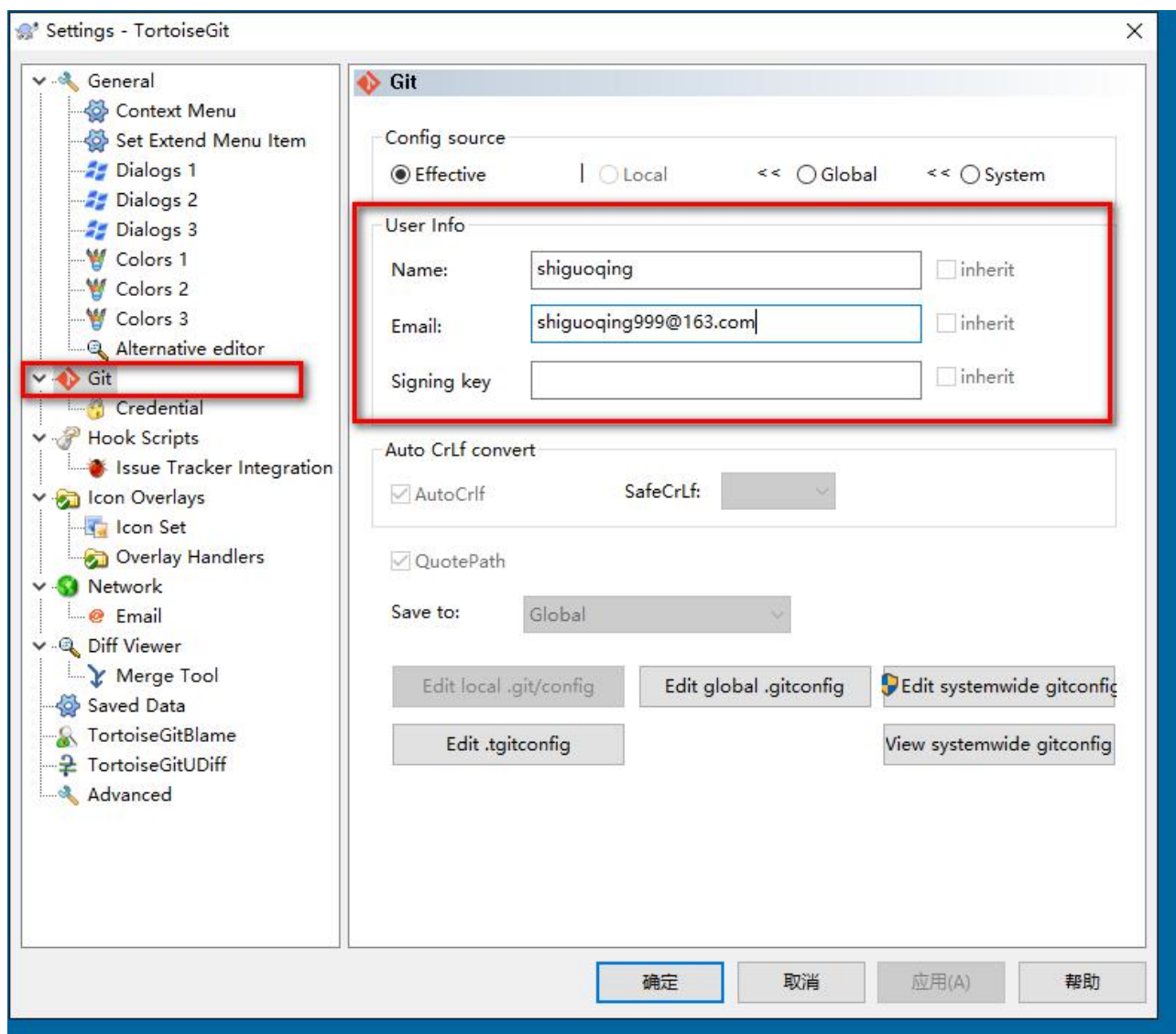
SourceTree, TortoiseGit, TortoiseSVN

1、安装

- （1）git
- （2）TortoiseGit 去百度搜索下载客户端

2、使用

- （1）配置用户名和邮箱，当然如果你曾经用 bash 的配置过，那么久就不需要了



2、创建本地仓储

4.8 冲突解决

假如两个开发同时改到同一文件的同一段内容会发生什么事情呢？

这时就会产生冲突了。

4.9 Git 高级（了解）

熟悉掌握以上操作，基本上是可以满足日常开的需求的，但是在解决一些特殊问题时，就又需要我们能够掌握更多的命令。

4.9.1 gitignore 忽略文件

在项目根目录下创建一个.gitignore 文件，可以将不希望提交的罗列在这个文件里，如项目的配置文件、node_modules 等

<https://github.com/github/gitignore>

<http://www.cnblogs.com/qwertWZ/archive/2013/03/26/2982231.html>

4.9.2 比较差异

当内容被修改，我们无法确定修改哪些内容时，可以通过 git diff 来进行差异比较。

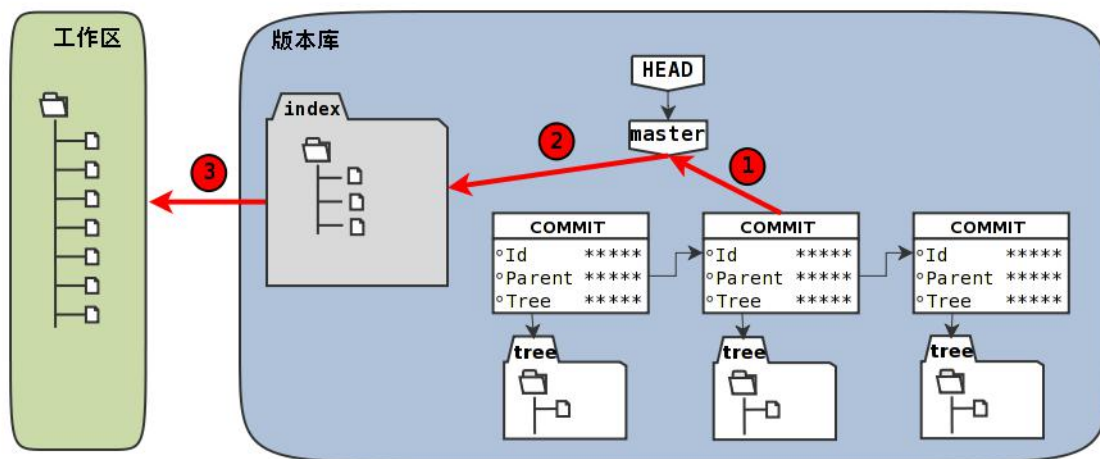
git difftool 比较的是工作区和暂存的差异

git difftool “SHA” 比较与特定提交的差异

git difftool “SHA” “SHA” 比较某两次提交的差异

git difftool 分支名称 比较与某个分支的差异

2、回滚（撤销）操作



HEAD 默认指向当前分支的“末端”，即最后的一次提交，但是我们通过 `git reset` 可以改变 HEAD 的指向。

看情况解释（稍微复杂一些，理解就好）

`git reset`

--hard 工作区会变、历史(HEAD)会变， 暂存区也变

--soft 只会变历史(HEAD)

--mixed（默认是这个选项）历史(HEAD)会变、暂存区也变，工作区不变

`git reflog` 回复 reset 的删除

`git checkout`

`git checkout SHA -- "某个文件"`，代表只是从 SHA 这个版中取出特定的文件，
和 `git reset` 是有区别的，`reset` 重写了历史，`checkout` 则没有。

4.9.3 更新仓库

在项目开发过程中，经常性的会遇到远程（共享）仓库和本地仓库不一致，我们可以通过 `git fetch` 命令来更新本地仓库，使本地仓库和远程（共享）仓库保持一致。

`git fetch` “远程主机”

或者

`git fetch` “远程主机” “分支名称”

我们要注意的，利用 `git fetch` 获取的更新会保存在本地仓库中，但是并没有体现到我们的工作目录中，需要我们再次利用 `git merge` 来将对应的分支合并（融合）到特定分支。如下

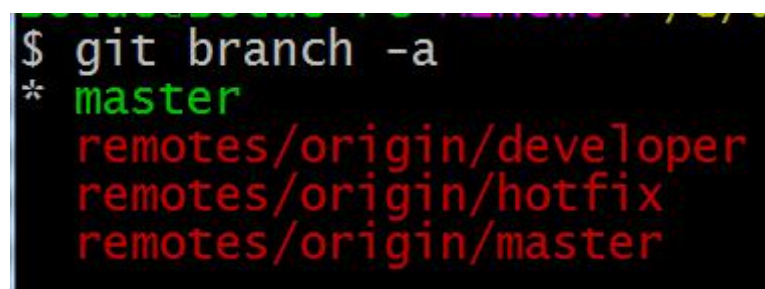
`git pull origin` 某个分支， 上操作相当于下面两步

`git fetch`

`git merge origin/某个分支`

问题：如何查看远程主机上总共有多少个分支？

`git branch -a` 便可以查看所有(本地+远程仓库)分支了



```
$ git branch -a
* master
remotes/origin/developer
remotes/origin/hotfix
remotes/origin/master
```


4.9.4 其它

删除远程分支 `git push origin --delete 分支名称`

删除远程分支 `git push origin :分支名称` 前面是空的本地分支

第 5 章项目发布

5.1 项目构建

当一个项目足够大的时候，我们会采按功能划分给不同的人员开发，进一步各个功能又会划分成不同的模块进行开发，这就会造成一个完整的项目实际上是由许许多多多的“代码版段”组成的；我们开发中又会用到 less、sass 等一些预处理程序，需要将这些预处理程序进行解析；为了减少请求需要将 css、javascript 进行合并；为了加速请求需要对 html、css、javascript、images 进行压缩；这一系列的任务完全靠手功完成几乎是不可能的，需要借助构建工具才可以实现。

所谓构建工具是指通过简单配置就可以帮我们实现合并、压缩、校验、预处理等一系列任务的软件工具。

常见的构建工具包括：Grunt、Gulp、F.I.S（百度出品）

5.2 Gulp

在了解了构建工具的用途，并且知道其只是一个软件后，接下来我们重点学习如何使用 Gulp 这个软件来工作。

5.2.1 Gulp 基础

Gulp 是基于 Nodejs 开发的一个构建工具。

1、全局安装

`npm install -g gulp`

2、本地安装（做为项目依赖）

在项目根目录下执行

`npm install gulp --save-dev`

3、创建任务清单 `gulpfile.js`

```
var gulp = require('gulp');

gulp.task('default', function() {
  // 将你的默认的任务代码放在这
});
```

4、定义任务

比如自动添加 CSS 私有前缀

本地安装 `gulp-autoprefixer --save-dev`

```
// 引入gulp包
var gulp = require('gulp');

// 引入自动前缀处理任务插件
var autoprefixer = require('gulp-autoprefixer');

// 配置任务
gulp.task('default', function () {
  gulp.src('./css/main.css')
    .pipe(autoprefixer({
      browsers: ['last 2 versions']
    }))
    .pipe(gulp.dest('./dist'));
});
```

5、执行任务

输入命令 gulp

```
Botue@Botue-PC MINGW64 /e/360
$ gulp
[16:10:14] Using gulpfile E:\360\gulpfile.js
[16:10:14] Starting 'default'...
[16:10:14] Finished 'default' after 12 ms
```

这样我们的 CSS 文件便会被自动的添加了浏览器私有前缀了

5.2.2 Gulp API

参见 <http://www.gulpjs.com.cn/docs/api/>

5.2.3 常用 Gulp 插件

Gulp-uglify

Gulp-autoprefixer

Gulp-htmlmin

Gulp-less

Gulp-minify-css

Gulp-imagemin

Gulp-concat

gulp-rev

gulp-rev-collector

gulp-clean

gulp 插件库:

<http://gulpjs.com/plugins/>

第 6 章gulp

1、工程化开发思想

2、gulp 简介

2.1、中文网

<http://www.gulpjs.com.cn/docs/api/>

2.2、gulp 优点

<http://www.hubwiz.com/class/562089cb1bc20c980538e25b>

3、快速开始 gulp

4、gulp 主要 api

5、gulp 常用插件介绍

6、高级（扩展）

自定义插件