Git 是啥?

Git 是分布式的版本库。

那么什么是分布式的版本库呢？说实话，第一次看到这句话时我的内心也是懵逼的。现在我们先不解释这个概念，总之你现在需要知道的就是：

Git 非常强大，非常好用，比 SVN 好使 1 万倍，是编写代码、修改 Bug、发布程序、持续集成，自动化运维、参与开源、居家旅行的必备神器。

Git 为什么这么牛掰呢 ？ 因为它是 Linus Torvalds 开发的。Linus Torvalds 是谁？Linus Torvalds 是这个世界最牛掰的程序员，他 21 岁时就开发出了 Linux。

等你认真读完这个手册，就会知道什么是分布式的版本库以及它相对于集中式版本库的好处。

下面我们正式开始进入 Git 的世界。

Get Started

安装完 Git,让我们开始第一次 Git 之旅。在终端运行下面的命令

```
$git config --global user.name "tianle"
$git config --global user.email "tianle@dangdang.com"
```

```
$git init git-demo
$cd git-demo
$echo "Git 学习" > README.md
$git add README.md
$git commit -m 'init repo and add README.md'
$git log
```

执行完以上命令，我们就已经建立了一个 Git 仓库，并且将文件 README.md 交由 Git 管理。是不是很简单!

如果你现在想把这个 README.md 分享给别人，就需要连接远程版本库，那么可以执行下面的命令。如果你比较自私不想分享你的成果，那么你现在就已经完成了第一个 Git 的"Hello World",可以跳到下一章了。

```
$git remote add origin git@github.com:christian-tl/git-demo.git
$git push origin master
```

这样你就将 README.md 推到了远程仓库的 master 分支，其他的小伙伴就可以看到了。他们不光能看到，还能修改完善这个文件。这就是工作的协作。

好了，现在我们愉快的结束了第一次 Git 之旅。你和 Git 已经不是那么陌生了。

git config 设置 user.name user.email 只需在首次使用 git 时设置，之后就无需再设置了。
如果是首次连接使用远程版本库的话，需要把当前用户的公钥传到远程版本库上，否则无法使用 ssh 协议

基本使用

由于 Git 是分布式的版本库，所以 Git 完全可以在没有网络的情况下使用，每个人的开发机都可以作为一个独立的版本库存在，开发者可以在自己的开发机管理代码版本，而不依赖于远程版本库。

所以，在使用 Git 时并不是必须有一个远程服务器的存在。但是在实际工作中的绝大部分时间里团队的成员之间都需要协作，提交（push）自己的修改和同步别人的代码(pull) .所以本文还是描述实际工作中的使用 Git 作为版本管理工具的协作方法。

团队开发中使用 Git 的基本流程：

- 克隆远程版本库
- 基于远程 develop 分支建立本地 develop 分支
- 基于 develop 分支建立本地特性分支 feature
- 在 feature 分支编写程序
- 切换到 develop 分支，合并 feature 的修改
- 把本地 develop 分支的修改推到远程 develop

上述流程基本上可以覆盖 90%的日常开发工作。

下面我们就通过一个简单的例子说明上述的流程，看看是如何用 Git 进行实际代码协作的。

- 克隆远程版本库

  ```
  $git clone git@github.com:christian-tl/git-demo.git
  ```

- 基于远程 develop 分支建立本地 develop 分支

  ```
  $cdgit-demo
  $git checkout -b develop origin/develop
  ```

- 基于 develop 分支建立本地特性分支 feature

  ```
  $git branch feature
  $git checkout feature
  ```

- 在 feature 分支编写程序

  ```
  $viREADME.md
  $via.txt
  $git add README.md a.txt
  $git commit -m'add a.txt , change README.md'
  ```

- 切换到 develop 分支，合并 feature 的修改

  ```
  $git checkout develop
  $git pull
  $git merge feature
  ```

- 把本地 develop 分支的修改推到远程 develop

  ```
  $git push
  ```

我们看到了一坨命令，是不是有点头晕眼花？是不是觉得 Git 很难用？没关系，你的想法是特别正常的。大部分人在最初接触 Git 时都想过放弃，觉得既然用 SVN 很简单为啥还要学着么难用的技术呢。实话实说 Git 的学习曲线确实很高，比 SVN/CVS 这种傻瓜式的版本控制工具要难很多。没办法，这个世界上没有什么完美的事情。我们要辩证的来看， 要想获得高收益就一定会付出一些代价。我们付出的代价就是多付出一点时间把 Git 的命令，对象存储原理弄明白，这对我们之后的整个开发流程都会有巨大的帮助。所以这个代价是值得的。

我们目前看到的命令是：***git clone***, ***git branch***, ***git checkout***, ***git add***, ***git commit***, ***git merge***, ***git pull***, ***git push***

这些都是 Git 最重要的命令，伴随着整个 Git 的使用过程。当然，在实际工作中不会像示例里这么简单，我们经常需要用 ***git status*** 查看工作区和暂存区状态,用 ***git log*** 查看提交版本，用 ***git diff*** 查看不同区域的差异，用 ***git reset*** 重置版本库，用 ***git rebase*** 合并提交与分支,用 ***cherry-pick*** 拣选提交，用 ***git fetch*** 从远程版本库下载，用 ***git tag*** 建立里程碑，等等。Git 有 170 多个命令，没有必要知道每个命令怎么用，也不可能做到。其实只要把上面列的命令会用了，工作中就基本够用了。后面的篇幅会详细介绍每个命令的用法。

在详细学习 Git 命令之前，我们要先了解一些 Git 的基本概念。这些概念非常的重要，它可以帮助我们理解 Git 的原理，这样我们才能更好的理解和使用命令，而不是对命令死记硬背。

Git 最重要的概念就是工作区，暂存区，版本库，Git 对象。

Git 目录

执行 **git init** 或 **git clone** 之后会生成一个目录， 我们的开发都是在这个目录下进行的。这里面有我们所有的代码文件，我们把这个目录叫做项目目录。

比如：**git clone** git@github.com:christian-tl/git-demo.git 或 git init git-demo 后 ， 生成的这个 git-demo 目录就是项目目录。

在项目目录下有一个 *Git 目录*，除了 Git 目录之外的都是*工作目录*

Git 目录

'*Git 目录*'是项目存储所有历史和元信息的目录 - 包括所有的对象 (***commits,trees,blobs,tags***).

每一个项目只能有一个'Git 目录'(这和 SVN,CVS 的每个子目录中都有此类目录相反)， Git 目录是项目的根目录下的一个名为 ***.git*** 的隐藏目录. 如果你查看这个目录的内容，你可以看到所有的重要文件。

还是回到前面 Get Started 的例子

```
$cd git-demo ; ll -al
drwxr-xr-x 9 christian staff 306 5 5 16:21 .git # 这个就是 Git 目录
-rw-r--r-- 1 christian staff 10 5 5 15:23 README.md # 这个是刚刚创建的文件
$cd .git ; tree -L 1
|-- HEAD # 记录当前处在哪个分支里
|-- config # 项目的配置信息，git config 命令会改动它
|-- description # 项目的描述信息
|-- hooks/ # 系统默认钩子脚本目录
|-- index # 索引文件
|-- logs/ # 各个 refs 的历史信息
|-- objects/ # Git 本地仓库的所有对象 (commits, trees, blobs, tags)
|-- refs/ # 标识每个分支指向了哪个提交(commit)。
```

这个.git 目录中还有几个其他的文件和目录，但都不是很重要。不用太关注。

工作目录

Git 的 '*工作目录*' 存储着你现在签出(checkout)来用来编辑的文件. 当你在项目的不同分支间切换时，工作目录里的文件经常会被替换和删除. 所有历史信息都保存在 'Git 目录'中； 工作目录只用来临时保存签出(checkout) 文件的地方，你可以编辑工作目录的文件直到下次提交(commit)为止.

'*工作目录*'包括在项目目录下，除了 **.git** 外的其他所有文件和目录

在我们的例子中对应关系如下。

项目目录：git-demo

Git 目录：git-demo/.git

工作目录：git-demo 下除了.git 目录之外的全部

.git 目录详解

对.git 目录先有个基本了解对后面的学习有很大帮助。

HEAD 文件

HEAD 文件就是一个只有一行信息的纯文本文件。这行内容记录的是当前头指针的引用，通常是指向一个分支的引用 ，有时也是一个提交(commit)的 SHA 值

```
$ cat .git/HEAD

ref: refs/heads/master #HEAD 文件的内容只有这一行，表明当前处于 master 分支
$ git checkout dd98199

Note: checking out 'dd98199'.


You are in 'detached HEAD' state.
...


$ cat .git/HEAD

dd981999876726a1d31110479807e71bba979c44 #这种情况是”头指针分离“模式，不处于任何分支下。HEAD 的值就是某一次 commit 的 SHA
```

config 文件

config 文件记录着项目的配置信息，也是一个普通的纯文本文件。git config 命令会改动它(当然也可以手工编辑)。
这个文件里面配置了当前这个版本库的基本属性信息，上游版本库信息，本地分支与上游的映射关系，命令别名等。
总之是一个很有用的文件。在你的.git 目录里看到的 config 文件内容基本上是下面的样子。

```
#基本配置
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = true
#上游版本库
[remote "origin"]
url = http://git.dangdang.com/stock/shopstock-update.git
fetch = +refs/heads/*:refs/remotes/origin/*
#本地分支与上游版本库分支的映射
[branch "master"]
remote = origin
merge = refs/heads/master
#当前仓库 Git 命令别名
[alias]
st = status
```

如果没有添加远程版本库，[remote "origin"]和[branch "master"]是不存在的；如果没有设置 alias 那么[alias]也是不存在的。

所以如果仅仅是 git init 之后的一个本地仓库，那么只有[core]配置项

hooks 目录

钩子(hooks)是一些在.git/hooks 目录的脚本，在被特定的事件触发后被调用。当 git init 命令被 调用后，一些非常有用的示例钩子脚本被拷到新仓库的 hooks 目录中；但是在默认情况下它们是不生效的。 把

这些钩 子文件的".sample"文件名后缀去掉就可以使它们生效。知道这个目录的用途就好，一般用不到。

index 文件

git 暂存区存放 index 文件中，所以我们把暂存区有时也叫作索引（index）。索引是一个二进制格式的文件，里面存放了与当前暂存内容相关的信息，包括暂存的文件名、文件内容的 SHA1 哈希串值和文件访问权限。暂存区是贯穿于整个 Git 使用流程的重要概念，所以 index 文件就很重要。由于是二进制所以我们无法查看具体内容，但是可以用 ***git ls-files --stage*** 命令查看暂存区里面的文件

```
$git ls-files --stage
100644 44601d12328ea8e04367337184dcccb85859610e 0 README.md
```

暂存区会在后面介绍，平时几乎不需要直接查看 index 文件，大家只需要知道 index 就是暂存区，非常重要就好。

objects 目录

Git 对象(blob,tree,commit,tag)都保存在 objects 目录里面，所以 objects 目录就是真正的仓库。objects 里面的目录结构组织的很有特点，是以 SHA 值的前 2 位作为目录，后 38 位作为这个目录下的文件名。

```
$tree objects/
objects/
```

```
├── 44
│   └── 601d12328ea8e04367337184dcccb85859610e
├── dd
│   └── 981999876726a1d31110479807e71bba979c44
├── e7
│   └── 77199b859e8e98db46e4897dc7076d07866042
├── info
└── pack
```

我们的工作目录里的所有文件，代码、库文件、图片等都会变成 git 对象存在这个 objects 目录下。每个文件都是一个二进制文件。可以通过 ***git cat-file -p*** SHA 值来查看文件的内容。

refs 目录

refs 目录下面是一些纯文本文件，分别记录着本地分支和远程分支的 ***SHA*** 哈希值。文件的数量取决于分支的数量。

```
$tree refs
refs
├── heads
│   ├── develop # 记录本地 develop 分支的 SHA 哈希值
│   └── master # 记录本地 master 分支的 SHA 哈希值
├── remotes
│   └── origin
│   ├── develop # 记录远程版本库 develop 分支的 SHA 哈希值
│   └── master # 记录远程版本库 master 分支的 SHA 哈希值
└── tags
└── v1.0 # 记录里程碑 V1.0 的 SHA 哈希值
```

回想前面介绍的 **HEAD 文件**， HEAD 文件的内容记录了当前处于哪个分支，值是 ref: refs/heads/master 。

而 refs/heads/master 文件 记录了 master 分支的最新提交的 SHA 哈希值 ，Git 就是通过 HEAD 文件和 refs/heads 下面的文件来判断当前分支及分支最新提交的。

```
$cat HEAD

ref: refs/heads/master # 说明当前处于 master 分支


$cat refs/heads/master

dd981999876726a1d31110479807e71bba979c44 # master 分支的最新提交 SHA 哈希值
```

logs 目录

logs 目录下面是几个纯文本文件，分别保存着 HEAD 文件和 refs 文件内容的历史变化。由于 HEAD 文件和 refs 文件的内容就是 SHA 值，所以 log 文件的内容就是这些 SHA 值的变化历史。

```
$tree logs
logs
├── HEAD
└── refs
├── heads
│ ├── develop
│ └── master
└── remotes
└── origin
├── develop
└── master
```

logs 目录的结构和 refs 几乎一样，只不过每个纯文本文件记录的 HEAD 文件和分支文件内容的变化日志,也就是 SHA 哈希值的变更日

志。在实际使用中，我们经常需要把代码整体回滚到一个历史状态，这是需要用到 ***git reflog*** 命令，这个命令其实就是读取的 logs 目录里的日志文件。就是这么简单。

```
$cat logs/HEAD
0000000000000000000000000000000000000000 dd981999876726a1d31110479807e71bba979c44 tianle <tianle@dangdang.com> 1493982048 +0800 commit (initial): init repo and add README.md
```

由于目前为止我们的例子只有一个提交(commit)，所以只有一个日志记录。

到目前为止 **.git 目录** 里的重要文件目录已经都介绍完了，大家掌握这些就可以了，还有一些其他的文件目录后续也会简单的提到。现在第一次看到这里，虽然我尽力写的简单，但是相信很多人看完一遍都会比较晕。比如提到了 Git 对象，那么什么是 Git 对象呢？我们接下来就介绍。等大家把后续的章节都看完一遍再回过头来 看这一章的时候会发现，原来真的不复杂很简单。

Git 对象存储及管理

SHA

所有用来表示项目历史信息的文件,是通过一个 40 个字符的“对象名”来索引的，对象名看起来像这样:

dd981999876726a1d31110479807e71bba979c44

你会在 Git 里到处看到这种“40 个字符”字符串。每一个“对象名”都是对“对象”内容做 SHA1 哈希计算得来的。

这样就意味着两个不同内容的对象不可能有相同的“对象名”。

这样做会有几个好处：

1. Git 只要比较对象名，就可以很快的判断两个对象是否相同。
2. 因为在每个仓库（repository）的“对象名”的计算方法都完全一样，如果同样的内容存在两个不同的仓库中，就会存在相同的“对象名”下,节省空间。
3. Git 还可以通过检查对象内容的 SHA1 的哈希值和“对象名”是否相同，来判断对象内容是否正确。

在 unix like 系统中，可以通过 ***sha1sum*** 命令对一个内容生成摘要。消息摘要算法主要有两种，分别是 **MD5** 和 **SHA**。SHA 又可以细分为 *SHA-1,SHA-256,SHA-384,SHA-512* 等算法，不同的算法会生成不同比特大小的哈希值。***sha1sum*** 这个从名字上可以看出，用的是 sha1 summary 算法。git 生成 SHA 哈希值用的就是 ***SHA1*** 算法。

```
$printf 'dangdang' | sha1sum
19dd09a3502f4d118893eaefbeab0dfc177e0b7a - #这就是 'dangdang'通过 SHA1 算法生成的摘要
```

SHA1 是一种密码学的信息摘要算法 ，有兴趣可以看一下我写的 JAVA SHA 算法的例子 ：

github.com/christian-tl 。

对象

每个对象(object) 包括三个部分：类型，大小和内容。大小就是指内容的大小，内容取决于对象的类型，Git 有四种类型的对象：" **blob**"、" **tree**"、 " **commit**" 和" **tag**"。

BLOB

用来存储文件数据，通常是一个文件。

TREE

“tree”有点像一个目录，它管理一些“tree”或是 “blob” （就像文件和子目录）

COMMIT

一个“commit”只指向一个"tree"，它用来标记项目某一个特定时间点的状态。它包括一些关于时间点的元数据，如时间戳、最近一次提交的作者、指向上次提交（commits）的指针等等。

TAG

一个“tag”是来标记某一个提交(commit) 的方法。

几乎所有的 Git 功能都是使用这四个简单的对象类型来完成的。它就像是在你本机的文件系统之上构建一个小的文件系统。这个小型的文件系统就是 **.git/objects** 目录。

与 SVN 的区别

Git 与你熟悉的大部分版本控制系统的差别是很大的。也许你熟悉 Subversion、CVS、Perforce、Mercurial 等等，他们使用 “增量文件系统” （Delta Storage systems），就是说它们存储每次提交(commit)之间的差异。Git 正好与之相反，它会把你的每次提交的文件的全部内容（snapshot）都会记录下来。这会是在使用 Git 时的一个很重要的理念。

为了更好的说明这 4 种对象类型，我们现在添加一些文件目录到当前的版本库中。

```
$mkdir -p script/shell script/perl ; echo '#!/bin/bash' >
script/shell/test1.sh ;echo '#!/usr/bin/perl' >
script/perl/test2.pl
$git add .
$git commit -m 'add shell and perl scprit.'
$git log
```

```
#我们看到现在有了两次 commit
commit e6361ed35aa40f5bae8bd52867885a2055d60ea2
Author: tianle <tianle@dangdang.com>
Date: Wed May 10 11:07:52 2017 +0800


add shell and perl scprit.


commit dd981999876726a1d31110479807e71bba979c44
Author: tianle <tianle@dangdang.com>
Date: Fri May 5 19:00:48 2017 +0800


init repo and add README.md
```

看一下现在的工作目录

```
$tree
.
├── README.md
└── script
├── perl
│ └── test2.pl
└── test1.sh


2 directories, 3 files
```

Blob 对象

blob 对象通常用来存储文件的内容。

可以使用 ***git show*** 或 ***git cat-file -p*** 命令来查看一个 blob 对象里的内容。在我们的例子中 README.md 文件对应的 blob 对象的 SHA1 哈希值是 **44601d12328ea8e04367337184dcccb85859610e**，我们可以通过下面的的命令来查看 blob 文件内容：

```
$ git show 44601d1
Git 学习
```

一个"blob 对象"就是一块二进制数据，它没有指向任何东西或有任何其它属性。

因为 blob 对象内容全部都是数据，如两个文件在一个目录树中有同样的数据内容，那么它们将会共享同一个 blob 对象，也就是说同样一份数据内容 git 只存储一个 blob 对象。Blob 对象和其所对应的文件所在路径、文件名是否改被更改都完全没有关系。

可以通过 git hash-object 命令生成文件的 SHA 哈希值,如果加上 **-w** 参数，会把这个文件生成 blob 对象并写入对象库。 **hash-object** 命令是个 Git 比较底层的命令，平时正常使用 Git 几乎用不到。

```
$git hash-object README.md
44601d12328ea8e04367337184dcccb85859610e
```

#这只会显示 READMD.md 文件 blob 对象的 SHA 值，并不会生成 blob 文件。
#git hash-object -w README.md ,则会真正把 README.md 生 blob 对象并写入对象库

总之，被 Git 管理的所有文件都会生成一个 blob 对象，

Icon 不需要写完整 40 位的 SHA 哈希值，只写前 7 位就可以

Tree 对象

一个 tree 对象有一串(bunch)指向 blob 对象或是其它 tree 对象的指针，它一般用来表示内容之间的目录层次关系。

git ls-tree 或 ***git cat-file -p*** 命令还可以用来查看 tree 对象，现在我们查看刚刚最新提交对应的 Tree 对象 我们可以像下面一样来查看它：

```
$git ls-tree HEAD^{tree}
100644 blob 44601d12328ea8e04367337184dcccb85859610e README.md
040000 tree 16a87dbed191bcfb19a4af9d0cc569f6448a01cc script
```

就如同你所见，一个 tree 对象包括一串(list)条目，每一个条目包括：mode、对象类型、SHA1 值 和名字(这串条目是按名字排序的)。它用来表示一个目录树的内容。

一个 tree 对象可以指向(reference): 一个包含文件内容的 blob 对象, 也可以是其它包含某个子目录内容的其它 tree 对象. Tree 对象、blob 对象和其它所有的对象一样，都用其内容的 SHA1 哈希值来命名的；只有当两个 tree 对象的内容完全相同（包括其所指向所有子对象）时，它的名字才会一样，反之亦然。这样就能让 Git 仅仅通过比较两个相关的 tree 对象的名字是否相同，来快速的判断其内容是否不同。tree 对象存储的是指针（tree 和 blob 的 SHA 哈希值），不存储真正的对象。tree 对象可以理解为就是一个目录，目录里包含子目录（tree 的 SHA 值）和文件（blob 的 SHA 值).而 SHA 值所对应的真正的对象文件存在 .git/objects 下面。

Icon 在 submodules 里，trees 对象也可以指向 commits 对象,本文不涉及 submodules 的相关内容，因为平时一般开发很少用到，如果确实需要，可以查询 Git 官方手册

Commit 对象

Commit 就是提交， "commit 对象"指向一个"tree 对象"，这个 tree 对象就是本次提交所对应的目录树,里面包括这次提交时工作区里面所有的目录和文件的指针，有时也叫做快照。"commit 对象"还带有相关的描述信息.

可以用 ***git log -1 --pretty=raw*** 或 ***git show -s --pretty=raw*** 或 ***git cat-file -p*** <commit>

```
$git cat-file -p HEAD
tree 8d384da6a7ebc4b88cc5fc5e45d609faf9b2cb29
parent dd981999876726a1d31110479807e71bba979c44
author tianle <tianle@dangdang.com> 1494385672 +0800
committer tianle <tianle@dangdang.com> 1494385672 +0800


add shell and perl scprit.
$git show -s --pretty=raw dd98199
commit dd981999876726a1d31110479807e71bba979c44
tree e777199b859e8e98db46e4897dc7076d07866042
author tianle <tianle@dangdang.com> 1493982048 +0800
committer tianle <tianle@dangdang.com> 1493982048 +0800


init repo and add README.md
```

提交(commit)由以下的部分组成:

一个 tree 对象: tree 对象的 SHA1 签名，代表着目录在某一时间点的内容.

父提交 (parent(s)): 提交(commit)的 SHA1 签名代表着当前提交前一步的项目历史. 上面的那个例子就只有一个父对象; 合并的提交(merge commits)可能会有不只一个父对象. 如果一个提交没有父对象，那么我们就叫它“根提交"(root commit)，它就代表着项目最初的一个版本(revision). 每个项目必须有至少有一个“根提交"(root commit)。Git 就是通过父提交把每个提交联系起来，也就是我们一般所说的提交历史。父提交就是当前提交上一版本。

作者 : 做了此次修改的人的名字， 还有修改日期.

提交者（committer): 实际创建提交(commit)的人的名字，同时也带有提交日期. TA 可能会和作者不是同一个人; 例如作者写一个补丁(patch)并把它用邮件发给提交者，由他来创建提交(commit).

提交说明： 用来描述此次提交.

一个提交(commit)本身并没有包括任何信息来说明其做了哪些修改; 所有的修改(changes)都是通过与父提交(parents)的内容比较而得出的。

一般用 git commit 来创建一个提交(commit), 这个提交(commit)的父对象一般是当前分支(current HEAD), 同时把存储在当前索引(index)的内容全部提交.

commit 是使用频率最高的对象，一般在使用 Git 时，我们直接接触的就是 commit。我们 *commit* 代码， *merge* 代码， *pull / push* 代码，重置版本库，查看历史，切换分支这些在开发流程中的基本操作都是直接和 commit 对象打交道。

对象模型

现在我们已经了解了 3 种主要对象类型(blob, tree 和 commit), 好现在就让我们大概了解一下它们怎么组合到一起的.

回忆一下现在项目的目录结构:

```
$tree
.

├── README.md
```

```
└── script

├── perl

│   └── test2.pl

└── test1.sh



2 directories, 3 files
```

在 Git 中它们的存储结构看起来就如下图:

Icon 每个目录都创建了 tree 对象，每个文件都创建了一个对应的 blob 对象 ．最后有一个 commit 对象 来指向根 tree 对象(root of trees)，这样我们就可以追踪项目每一项提交内容。除了第一个 commit,每个 commit 对象都有一个父 commit 对象,父 commit 就是上一次的提交(历史 history)，这样就形成了一条提交历史链。Git 就是通过这种方式组成了 git 版本库

Tag 对象

一个标签对象包括一个对象名，对象类型，标签名，标签创建人的名字("tagger")，还有一条可能包含有签名(signature)的消息. 你可以用 ***git cat-file -p*** 命令来查看这些信息。

现在我们的 git 对象库里还没有一个 tag 对象，我们先用 git tag -m <msg> <tagname> [<commit>] 命令创建一个 tag。

```
$git tag -m 'create tag from demo' v1.0 #基于当前 HEAD 建立一个 tag,所以 tag 指向的就是 HEAD 的引用
$git tag
v1.0
$git cat-file -p v1.0
object e6361ed35aa40f5bae8bd52867885a2055d60ea2
type commit
tag v1.0
tagger tianle <tianle@dangdang.com> 1494406971 +0800

create tag from demo
```

baa10bc4dc2ba6ff13e0bd0cdd50d8e9d36564a8

Tag 对象就是里程碑的作用，一般在我们正式发布代码是需要建立一个里程碑。

好了，至此我们就把 Git 的 4 种对象类型介绍完了。他们是 **blob**，**tree**，**commit**，**tag** 。这部分非常重要，理解了 Git 对象模型是理解 Git 使用流程和各种 Git 命令的基础。

要想会用 Git,这部分必须弄清楚，否则永远不会用，只能照猫画虎死记硬背命令，这样一旦命令的结果的预想的不一致，就懵了。

Git 官方文档对这部分写的很详细，看完上面的介绍再仔细看这个文档会有更深的认识 ： Git-内部原理-Git-对象

工作区、暂存区、版本库

Git 对于我们的代码管理分了 3 个区域，分别是工作区，暂存区和版本库。这是 Git 完全不同于 SVN 的地方。Git 之所以强大很大程度上就是因为它设计了 3 个区域，但同时也是因为这个设计让 Git 学习起来比较难，上手也比较难，比较难理解。凡事都是有利有弊。

- **工作区(Working Directory)：**

就是电脑上的一个目录，里面是正在开发的工程代码。执行 git init 命令后，生成的这个目录除了.git 目录，就是工作区。

- **暂存区(Stage / Index)：**

暂存区最不好理解的一个概念，可以先认为需要提交的文件要先放到暂存区才能提交。所以暂存区可以理解为“提交任务”。是代码提交到版本库前的一个缓冲区域。

暂存区其实就是 .git/index 文件

- **历史库(History)：**

  Git 版本库的概念和 SVN 完全不同。SVN 的版本库指的是远程中央仓库，而 Git 的版本库指的是本地仓库，这里面存放着文件的各个版本的数据。其实就是 .git/objects 目录。Git 对象都存在这个目录里

看过 Git 目录和 Git 对象两个章节后，再来理解这 3 个区域其实已经不是很难了。

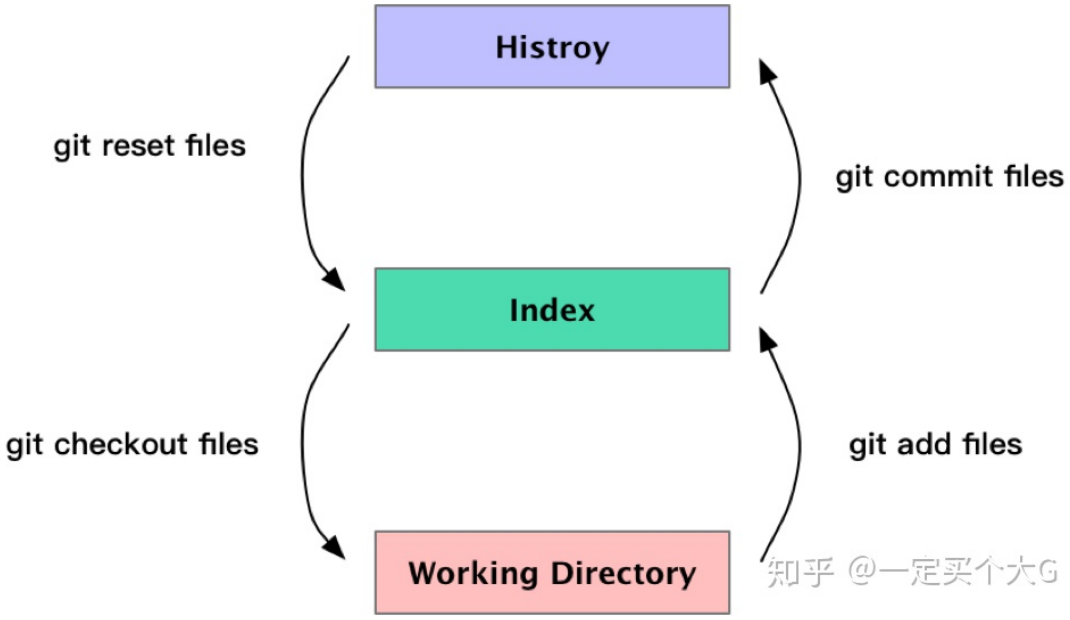
历史库，版本库，Git 仓库，History,叫法不同而已，其实指的是同一回事。就是执行了 git commit 后，生成了 commit 对象。现在我们知道 commit 对象包含了提交时间，提交人，提交说明以及提交时的目录树和父提交。那么上面这些都是构成版本库的要素。由于 Git 管理的所有对象文件都在 .git/objects 目录中，所以版本库概念可以具体形象的理解成 .git/objects 目录里面的对象文件。.git/objects 目录就是版本库，虽然这么说不是十分的准确，但是非常便于记忆和理解。

暂存区，Stage,Cached,Index ,叫法不同而已，其实指的是同一回事。暂存区就是 .git/index 这个二进制文件，记录着目录和文件的引用(SHA1 值)，而不是真正的文件对象。真正的文件对象存在于 .git/objects 目录里面。

工作区其实就没啥可说的了，就是真实的，看的见摸得到的，正在写的代码。就是你的 IDE 里面打开的这一堆东西。

再次加强一下记忆： 暂存区就是 **.git/index** 文件 ， 版本库就是 **.git/objects** 目录

下面的图例展示了 3 个区域的关系以及涉及到的主要命令 ： git add , git commit , git reset , git checkout



因为有了这 3 个区域，在使用 Git 时，文件经常处于不同的状态：

- 未被跟踪的文件（untracked file）
- 已被跟踪的文件（tracked file）
    - 被修改但未被暂存的文件（changed but not updated 或 modified）
    - 已暂存可以被提交的文件（changes to be committed 或 staged）
    - 自上次提交以来，未修改的文件(clean 或 unmodified)

好了，现在我们通过一张详细的图来形象的展现一下这 3 个区域。

在这个图中，我们可以看到部分 Git 命令是如何影响工作区和暂存区的：　　图中左侧为工作区，右侧为版本库。在版本库中标记为

"index" 的区域是暂存区（stage, index）。 图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下 当对工作区修改（或新增）的文件执行 ***git add*** 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的 SHA1 哈希值被记录在暂存区的文件索引中当执行提交操作时，暂存区的目录树写到版本库（对象库）中。

当执行 ***git status*** 命令扫描工作区改动的时候，先依据 .git/index 文件中记录的（工作区跟踪文件的）时间戳、长度等信息判断工作区文件是否改变。如果工作区的文件时间戳改变，说明文件的内容可能被改变了，需要打开文件，读取文件内容，和更改前的原始文件相比较（本地文件和与之对应的 object 库中的文件的内容进行对比），判断文件内容是否被更改。如果文件内容没有改变，则将该文件新的时间戳记录到 .git/index 文件中。因为判断文件是否更改，使用时间戳、文件长度等信息进行比较要比通过文件内容比较要快的多，所以 Git 这样的实现方式可以让工作区状态扫描更快速的执行，这也是 Git 高效的因素之一。

命令 ***git diff <file>*** 用来进行具体文件的变动对比，通常用来进行工作区与暂存区之间的对比，实质上是用 git objects 库中的快照与工作区文件的内容的对比。

git add <file> 把当前工作目录中的文件放入暂存区域。这么说是为了便于理解，其实不准确。

准确的说法是 git add files 做了两件事:

1. 将本地文件的时间戳、长度，当前文档对象的 id 等信息保存到一个树形目录中去（.git/index，即暂存区）
2. 将本地文件的内容做快照并保存到 Git 的对象库(.git/object) 。

从命令的角度来看，**git add** 可以分两条底层命令实现:

1. **git hash-object-w** *<file>*
2. **git update-index --add** *<file>*

```
$git hash-object a.txt
$git update-index --add a.txt
#以上两条命令等价于 git add a.txt
```

第一条命令，Git 将会根据新生成的文件产生一个长度为 40 的 SHA1 哈希字符串，并在.git/objects 目录下生成一个以该 SHA1 的前两个字符命名的子目录，然后在该子目录下，存储刚刚生成的一个新文件，新文件名称是 SHA1 的剩下的 38 个字符。

第二条命令将会更新.git/index 索引，使它指向新生成的 objects 目录下的文件。

这 2 条命令是为了说明 git add 文件到暂存区 Git 做了哪些事情，真正工作中大家只要记住把文件放到暂存区就是用 git add 命令就好。

我们再次加强一下理解，暂存区实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中(.git/index)，记录了文件名、文件的状态信息（时间戳、文件长度等），文件的内容并不存储其中，而是保存在 Git 对象库（.git/objects）中，文件索引建立了文件和对象库中对象实体之间的对应。

git commit <file> 命令就是生成一个新的提交，主要干了这么几件事：

1. 生成一个 commit 对象
2. 把暂存区的目录树写到版本库中，也就是生成一个 tree 对象，这个 tree 里面的引用和暂存区 index 里的引用一样
3. 跟新当前分支 ref 文件的引用，指向最新生成的这个 commit 的 SHA1 哈希值

git reset HEAD 命令，暂存区的目录树会被重写，被最新提交的目录树所替换，但是工作区不受影响。

“***git checkout .***” 或者 “***git checkout -- <file>***” 命令，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

“***git checkout HEAD .***” 或者 “***git checkout HEAD <file>***” 命令，会用最新提交的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

由于 add 和 commit 于 SVN 中的 add commit 命令名字相同，用惯了 SVN 刚接触 Git 时会很困惑，这里说明一下：

两个版本库中的 add 命令有点相似，都是让版本库 track 这个文件。只有被 add 的文件版本库才会管理它，这点是一样的。不同的是在 SVN 中，只有新文件才需要 add ,之后就不再需要 add 了。而 Git 是不仅仅新文件需要 add ,每次修改后都需要 add 。

两个版本库中的 commit 命令则完全不同。在 SVN 中 commit 是把代码提交到远程版本库，commit 之后别人就能看到这个提交了，需要联网。而 Git 中的 commit 是提交到本地版本库，不是到远程版本库，别人看不到这个提交，完全都是本地操作，不需要联网。所以说 SVN 是集中式的版本库，而 Git 是分布式的。

这些就是 Git 暂存区相关的知识，比较难理解，一开始看不懂的话也正常，我也不想写更多了，因为确实没什么可写的了，写更多那样只会更晕。理解这部分内容最好的办法就是动手做实验，每个命令多执行几次，而且要看每个命令执行后 对.git/index 和 .git/objects 目录的影响。放心的练习，不会因为执行几条 Git 命令就把电脑搞爆炸了。

最后总结几点：

- 理解 Git 暂存区和版本库的前提是理解和弄明白 Git 对象(blob,tree,commit)，否则只是看本文或网上各种帖子的描述，死记这几个命令的话，是很难理清这个逻辑的，即使当时有思路了，过几天也会忘。因为这是 Git 的原理，弄清原理后看似复杂难用的 git 其实就变得很简单。
- 一定不能只是看和想，这个东西不动手很难真的想明白，一定多动手试试。
- 在实际开发中请记住，只要对一个文件执行 add 命令了，不管之后这个文件是被删了还是从历史版本中彻底删除了，这个文件的内容一定可以找回来。原因不解释，等对暂存区和 Git 对象真的明白了，自然就清楚了。
- 初用 Git 时千万不要和 SVN 的命令和流程做对比，这两个版本库的设计哲学和实现完全不一样。千万不要想 Git 的这个命令对应着 SVN 的哪个命令，因为很难找到对应。

至此，Git 的原理已经介绍完了。接下来我们看看具体的使用。具体的使用就是命令，不同的命令干不同的活。

分支

分支是用来管理开发的，这点 Git 分支并没有什么不同，所以我们就不在这里过多解释什么是分支。我们现在要说的是 Git 是怎么实现的分支，为什么 Git 建分支这么快，这么方便？为什么 Git 分支对磁盘空间几乎没有任何影响，即使建了 200 个分支占用的空间也只不过不到 1M 的磁盘空间？在本章我们要关注的就是这个问题。

我们已经了解了 Git 目录结构和主要的文件，也清楚了 Git 的对象，现在是时候看一下 Git 版本库的整体结构了。

这个就是 Git 版本库的结构，组成这个结构的每个元素我们在前边都详细的描述过，大家仔细看下这个图就应该能够对 Git 整体有个清晰的认识了。

HEAD 文件记录了当前分支的引用文件名

refs/heads/<branchname> 文件记录了分支指向的提交 ID(commit SHA1)

```
$git branch
* master #当前处于 master 分支
```

```
$cat .git/HEAD
```

```
ref: refs/heads/master #指向了 master 分支，上面的 git branch 命令就是读取了这行记录才知道当前处于 master 分支
```

```
$cat .git/refs/heads/master
```

```
e6361ed35aa40f5bae8bd52867885a2055d60ea2 #说明 master 分支所指向的提交 ID
```

我们看到这两个文件都是普通的纯文本文件，且每个文件都只有一行记录。HEAD 文件只有 20 字节，分支文件有 41 字节大小。

Git 就是通过这两个文件表示一个分支，除此以外什么事都没干。分支文件在 **.git/refs/heads** 目录下与分支同名，每新建一个分支这个目录下就会生成一个同名的分支文件。

对于 master 分支，它的分支文件就是 **.git/refs/heads/master**；对于 develop 分支，它的分支文件就是 **.git/refs/heads/develop**,以此类推。

我们现在只有一个 master 分支，所以 **.git/refs/heads/** 目录下只有一个 master 文件。

通过 **git branch branchname <commit>** 命令可以新建一个分支。 提交 ID 可以省略，如果指定提交 ID 那么新分支就指向这个提交。如果省略提交 ID，那么新分支指向当前分支的头指针。

git branch develop e6361ed ,基于提交 e6361ed 创建新分支 develop

git branch feature, 这个命令等价于 git branch feature HEAD,就是给予当前分支的最新提交创建分支 feature

`$git branch #git branch` 命令可以查看当前所有本地分支，前面有 `*` 的代表当前分支
`develop`
`* master #`当前在 `master` 分支上

`$git branch develop #`基于 `master` 分支，创建一个名为 `develop` 的分支，这 2 个分支指向同一个 `commitID`

`$cat .git/refs/heads/develop #.git/refs/heads` 目录下会生成一个名为 `develop` 的文本文件，文件的内容就是分支指向的 `commitID`
`e6361ed35aa40f5bae8bd52867885a2055d60ea2`

`$git branch #git branch develop` 只新建分支但不切换分支，所以目前仍然在 `master` 分支上面
`develop`
`* master`

`$cat .git/HEAD`

`ref: refs/heads/master #HEAD` 文件指向的还是 `master` 的分支引用文件

```
$git checkout develop #git checkout 命令会切换分支
$git branch
* develop #当前分支已经切换到 develop 上
master
```

```
$cat .git/HEAD #HEAD 文件的内容也随之改变
ref: refs/heads/develop
```

```
$git checkout -b feature #git checkout -b 新建并切换分支， 等价于 git branch feature ; git checkout feature
$git branch
develop
* feature
master
$git checkout master
$git branch -D feature #git branch -D 删除一个分支
Deleted branch feature (was e6361ed).
```

HEAD 文件经常被称为头指针，就像一个游标，这个 HEAD 指向哪个分支当前就处于哪个分支。所以 **git checkout 分支名** 这个命令就是修改 HEAD 文件的内容。我们也可以手工修改 HEAD 文件，这和执行命令的效果是一样一样的。如果我们故意把 HEAD 文件清空，那么在这个项目目录里就无法识别这是个 git 工程了，有兴趣可以试试。

在项目目录中开发是，一般情况都是处于某一个分支中。有时，也会不处于任何分支下，这种情况叫做头指针分离。就是 HEAD 指向了一个提交号而不是分支文件

git checkout 提交号 命令会让项目处在头指针分离模式下

```
$git checkout e6361ed #detached HEAD，头指针分离模式
Note: checking out 'e6361ed'.

You are in 'detached HEAD' state. You
can look around, make experimental
changes and commit them, and you can discard any commits
you make in this
state without impacting any branches by performing another
checkout.

$cat .git/HEAD
e6361ed35aa40f5bae8bd52867885a2055d60ea2 #HEAD 指向了一个具体的提交号
```

在头指针分离模式下也可以继续正常 add commit 等操作，并没有特别大的区别，只是不处在任何分支下。我觉得头指针分离并没什么特别的意义，相当于一个人不属于任何组织，单干。

一般是在需要做一些修改尝试又不想建一个分支的情况下会直接 checkout 一个提交号进入头指针分离模式。但是 Git 建一个分支非常方便快捷，并不麻烦。所以，我们只需知道就行，平时很少用到。

Icongit init 初始化一个 Git 代码库之后，默认处于 master 分支下。HEAD 文件指向 ref: refs/heads/master 。但是并不会有 .git/refs/heads/master 文件，只有当第一次提交之后才会生成这个文件

现在我们了解到，Git 新建一个分支就是在**.git/refs/heads** 目录下生成一个 41 字节的分支文件，切换分支就是修改 **.git/HEAD** 文件的一行内容。不得不说 Git 的这个设计实在是太精妙了。

新建 100 个分支，就是新建 100 个分支文件而已，在分支间切换也只是修改一个 HEAD 文件而已。

所以

Git 建分支才会这样快

无论建多少个分支都不会使 Git 仓库变得冗余

分支间切换可以在瞬间完成

分支是所有版本管理工具的重要内容，所以 Git 高效的分支机制是他这么受欢迎的重要原因之一，也是 Git 的优势所在。

Git 分支比较简单好理解，没啥特别值得多说的。我这里主要想讨论的是对分支的管理

Git 分支管理

经过近 10 年的发展，在实践中大家总结出来了一个行之有效的 Git 分支管理模型。这个模型有一些变种，但都是基于 Vincent Driessen 在 2010 年提出的这个模型基础之上的。

这里我就介绍一下这个模型，英文好的同学可以直接看原文 nvie.com/posts/a-succes

如果我们的项目里只有一个分支，那么这个项目代码管理就是及其失败的。相当于只是把版本库当作一个代码分享工具，而不是开发流程管理工具。如果只有一个分支的话，那么这个项目团队一定是没有开发管理流程的。

因为一个分支确实无法 HOLD 住整个复杂的开发测试发布部署流程。比如，当你为一个项目开发新功能时线上出现 BUG 了，需要紧急修复，你改如何处理手头的代码呢？如果这个功能是几个人共同开发的，情况就更复杂了。比如，新加入的成员需要看线上稳定的程序代码，但是这时上面却是你提交了一半的新功能，如何保证代码取下来和线上的一致？再比如，你在给项目增加一个重要功能，另一个同事要增加别的不相关的功能，如何保证在最终合并前你们不互相影响？对于持续集成工具来说，如何从分支上拉取的代码保证是一定可以正常编译和运行的？ 例子太多了，实际开发工作有各种复杂的情况。

所以这就需要我们管理好，利用好分支。

基础分支

- **master**
- **develop**

一个项目的代码库至少要有 master 和 develop 这两个分支。团队成员从主分支(master)获得的都是处于可发布状态的代码，而从开发分支(develop)应该总能够获得最新开发进展的代码。

从 master 上获得的代码一定要保证是和线上运行的程序是一致的。

从 develop 上获得的应该是最新的稳定版本的代码。

除了基础分支外，我们还需要辅助分支。辅助分支大体包括如下几类：“管理功能开发”的分支、“帮助构建可发布代码”的分支、“可以便捷的修复发布版本关键 BUG”的分支。

辅助分支的最大特点就是“生命周期十分有限”，完成使命后即可被删除。

辅助分支

- **Feature branch**
- **Release branch**
- **Hotfix branch**

Feature branch

从 develop 分支检出，最终也会合并于 develop 分支。常用于开发一个独立的新功能，且其最终的结局必然只有两个，其一是合并入“develop”分支，其二是被抛弃。最典型的“Fearture branches”一定是存在于团队开发者那里，而不应该是“中心版本库”中。

通过下面的命令来解释这个流程

```
$ git checkout -b myfeature develop
#在 myfeature 上开发完代码之后，需要合并到 develop 分支上
$ git checkout develop
$ git merge myfeature
$ git branch -d myfeature
$ git push origin develop
```

Release branch

从 develop 分支检出，最终合并于“develop”或“master”分支。这类分支建议命名为“release-*”。通常负责“短期的发布前准备工作”、“小 bug 的修复工作”、“版本号等元信息的准备工作”。与此同时，“develop”分支又可以承接下一个新功能的开发工作了。在一段短时间内，在“Release branch”上，我们可以继续修复 bug。在此阶段，严禁新功能的并入，新功能应该是被合并到“develop”分支的。“Release branch”产生新提交的最好时机是“develop”分支已经基本到达预期的状态，至少希望新功能已经完全从“Feature branches”合并到“develop”分支了。

经过若干 bug 修复后，“Release branches”上的代码已经达到可发布状态，此时，需要完成三个动作：第一是将“Release branches”合并到“master”分支，第二是一定要为 master 上的这个新提交打 Tag（记录里程碑），第三是要将“Release branches”合并回“develop”分支。

通过下面的命令来解释这个流程

```
$ git checkout -b release-1.2 develop
#修改版本号等元信息的准备工作或者小 bug 的修复工作后 要合并到 master
$ git checkout master
$ git merge release-1.2
$ git tag -a 1.2 #发布前要建立里程碑
#如果有 bug 的修改，还需要合并到 develop
$ git checkout develop
$ git merge release-1.2
$ git branch -d release-1.2 #最后删除这个发布分支，它已经完成使命
```

Hotfix branch

从“master”检出，合并于“develop”和“master”，通常命名为“hotfix-*”

建议设立“Hotfix branches”的原因是：线上总是可能产生非预期的关键 BUG，希望避免“develop 分支”新功能的开发必须为 BUG 修复让路的情况。

BUG 修复后，需要将“Hotfix branches”合并回“master”分支，同时也需要合并回“develop”分支

通过下面的命令来解释这个流程

```
$ git checkout -b hotfix-1.2.1 master
#修复完 BUG 之后，要合并到 master
$ git checkout master
$ git merge hotfix-1.2.1
$ git tag -a 1.2.1 #修改线上 BUG 需要打标签


#修复完 BUG 之后，也要合并到 develop
$ git checkout develop
$ git merge hotfix-1.2.1


$ git branch -d hotfix-1.2.1 #最后 hotfix 的分支使命完成，删除之
```

这就是一个非常好的分支管理模型。

所以，在我们的 gitlab 上面我们一定至少要有 2 个分支

master 永远保持和线上代码同步，在上线部署时从这个分支拉去代码打包。如果我们的 DI 工具到时的功能完善，则 DI 工具直接从这个分支去代码打包发布

develop 我们的持续集成工具从每天从这个分支上取代码编译大包部署到测试环境(KVM,Docker)。

每次上线前都要建立里程碑 Tag

分支使用规范

这种分支模型的好处是：

1. 逻辑上明确各个分支的作用，在流程规范上杜绝了代码提交的混乱、代码合并的混乱、代码版本的混乱，可以更好的支持多人并行需求的开发过程。
2. 从流程上严格保证了未经测试的代码不会被发布到生产环境。
3. 不允许直接向 ***develop*** 分支提交代码，保证了 develop 分支的纯粹性，因为 develop 分支代表了最新最稳定的代码分支。同时，也可以把最后的代码评审收口到 develop，尽最大努力保证 develop 分支代码的可靠性。
4. ***qa*** 分支部署预发环境，尽最大努力把最后的风险控制在预发环境，保证生产环境的绝对可靠。
5. 在需求多并且交付频密，且多人协作的开发情况下，保证需求和需求之间、开发者和开发者之间互不影响。
6. 只有 ***master*** 分支可以部署生产环境，尽最大努力保证 master 分支的干净、纯粹、稳定和万无一失。

远程版本库

Git 版本库就是项目目录下的 .git 目录，里面包括所有的历史版本文件。如果这个.git 目录发生损坏或被删除那么整个版本库就永远丢失了,或者不小心把整个项目目录删了，那么所有的努力都白费了,我们使用版本控制工具就是为了保证代码不丢失。还有，我们总是需要写协

作分享我们的代码，那怎么分享自己的项目目录给别人呢？当然不是靠 QQ 或发邮件了。

克隆

git clone 命令可以克隆一个版本库，主要有 3 种形式：

用法 1 : git clone <repository> <directory>

用法 2 : git clone --bare <repository> <directory>

用法 3 : git clone --mirror <repository> <directory>

用法 1 会克隆一个 <repository>指向的版本库到 <directory> 目录，相当于 copy 了一个 repository 的副本，里面有着一样的工作区，一样的 .git 目录。差别是新克隆出来的这个版本库里的.git/config 文件会记录上游版本库 repository 的位置。

用法 2 克隆出来的版本库不包括工作区，直接就是版本库的内容，也就是不包括.git 目录而是直接就是.git 目录里面的内容。这样的版本库称为裸版本库。(通过 bare 名字就可以看出)

用法 3 和用法 2 类似，也是克隆出一个裸版本库。不过是可以通过 git fetch 命令与上游版本库 repository 持续同步。

```
$git clone git-demo A #clone 一个对等的版本库 A
Cloning into 'A'...
```

```
done.
$git clone --bare git-demo B #clone 一个裸的版本库 B
Cloning into bare repository 'B'...
done.
$git clone --mirror git-demo C #clone 一个裸的镜像版本库 C
Cloning into bare repository 'C'...
done.


$ls -a A #对等版本库 A 和 git-demo 有着同样的工作区，同时也有.git 目录
.git README.md script
$ls -a B #裸版本库 B 和 C 里面直接就是.git 目录里面的内容
HEAD config description hooks info objects packed-refs refs
$ls -a C
HEAD config description hooks info objects packed-refs refs


$cat A/.git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false #说明是非裸版本库
logallrefupdates = true
ignorecase = true
precomposeunicode = true
[remote "origin"] #当前版本库的上游版本库，名字为 origin
url = /Users/christian/work/tmp/git/git-demo #上游版本库的位置（URL）
fetch = +refs/heads/*:refs/remotes/origin/* #git fetch 时的默认引用表达式
```

```
[branch "master"] #本地 master 分支与上游版本库分支的映射，这样执行 git pull 时 相当于 git pull origin master
remote = origin
merge = refs/heads/master
```

```
$cat B/config
[core]
repositoryformatversion = 0
filemode = true
bare = true #说明是裸版本库
ignorecase = true
precomposeunicode = true
[remote "origin"]
url = /Users/christian/work/tmp/git/git-demo #上游版本库的位置（URL）
```

```
$cat C/config
[core]
repositoryformatversion = 0
filemode = true
bare = true #说明是裸版本库
ignorecase = true
precomposeunicode = true
[remote "origin"]
url = /Users/christian/work/tmp/git/git-demo
fetch = +refs/*:refs/*
```

`mirror = true #说明是--mirror ，可以执行 git fetch 命令和上游保持同步`

版本库之间的交互有 3 个命令：

git clone

git pull

git push

git fetch

有两点需要特别注意：

- **在非裸版本库中可以执行 git pull 和 git push 命令同步和推送代码，也可以从一个在非裸版本库 pull 代码，但是不能向在非裸版本库中 push，因为在非裸版本库有工作区，push 会导致工作区混乱。**
- **在裸版本库中不可以执行 git pull 和 git push 命令，但是可以从裸版本库 pull 和向裸版本库 push.**

Tips

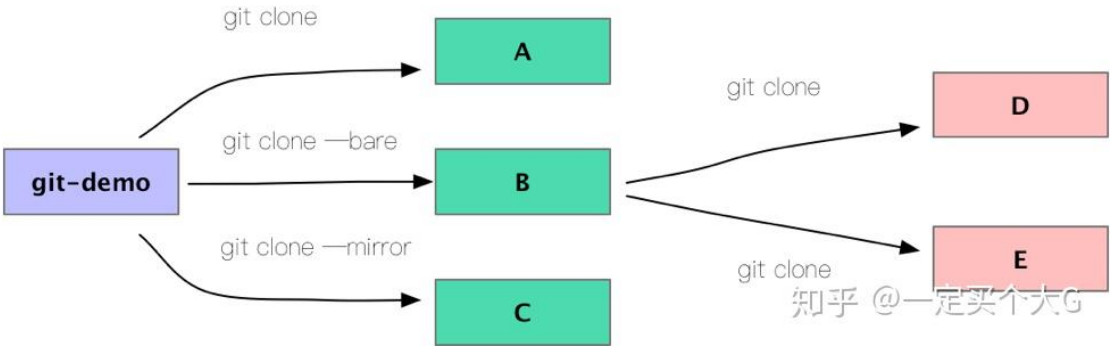
Git 的裸版本库约定以".git"结尾，如标准的裸版本库名 git-demo.git 。这里为了例子演示方便，取了好写好记的名字 A B C...

为了方便我们的实验，需要再 clone 两个版本库

```
$git clone B D #基于裸版本库 B clone 一个非裸版本库 D
Cloning into 'D'...
done.
$git clone B E #基于裸版本库 B clone 一个非裸版本库 E
Cloning into 'E'...
done.

#没看错，可以从一个裸版本库 clone 出一个非裸版本库。
$ls -a E #可以看到，E 有工作区，也有.git 目录
.git README.md script
```

现在有 6 个版本库，看起来的关系是这样的



下面我们通过实例，来看一下 git pull,git push,git fetch 这 3 个命令

```
$cd git-demo
$git commit --allow-empty -m 'c1' #为了方便测试，在 git-demo 里提交了一个空的 commit
[master 472f2ea] c1
$cd ../A
```

$git pull #在非裸版本库 A 中，把上游版本库 master 分支最新的提交 pull 下来

remote: Counting objects: 1, done.

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

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

From /Users/christian/work/tmp/git/git-demo

e6361ed..472f2ea master -> origin/master
Updating e6361ed..472f2ea
Fast-forward
$git log --oneline
472f2ea c1 #确实将 c1 提交 pull 下来了。
e6361ed add shell and perl scprit.
dd98199 init repo and add README.md

$git commit --allow-empty -m 'c2' #在 A 中也提交了一个新的 commit

[master eef952e] c2
$git push #想要推送到上游版本库，但是失败了，因为上游 git-demo 是个非裸版本库

Counting objects: 1, done.

Writing objects: 100% (1/1), 171 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)
remote: error: refusing to update checked out branch:
refs/heads/master
...

$cd ../B

$git pull #我们到版本库 B 中，也想 pull 上游的最新提交 c1,但是失败了，因为 B 没有工作区，是个裸版本库

fatal: This operation must be run in a work tree

$cd ../git-demo

$git push ../B master #我们到 git-demo 中把提交 push 到 B，说明可以向裸版本库 push 数据

Counting objects: 1, done.

```
Writing objects: 100% (1/1), 170 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To ../B
e6361ed..472f2ea master -> master
```

```
$cd ../C
$git fetch #我们到 mirror 镜像版本库中 fetch 上游 git-demo 的新提交
From /Users/christian/work/tmp/git/git-demo
e6361ed..472f2ea master -> master
```

通过上面的例子我们基本了解了 3 中不同类型版本库的区别，下面我们再看个好玩的例子来加深对 clone 版本库的理解

```
$cd ../D
$git pull #到非裸版本库 D 中 pull 上游裸版本库 B 的提交
$git log --oneline
472f2ea c1 #提交 c1 已经被 pull 下来，说明可以从裸版本库 pull 代码
e6361ed add shell and perl scprit.
dd98199 init repo and add README.md
```

```
$git commit --allow-empty -m 'c2' #我们在当前 D 中创建一个新 commit
'c2'
[master be5099d] c2
$git push origin master #把提交 push 到上游 B， 说明可以向裸版本库 push 代码
Counting objects: 1, done.
Writing objects: 100% (1/1), 170 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To /Users/christian/work/tmp/git/B
472f2ea..be5099d master -> master
```

```
$cd ../E
$git pull #到非裸版本库 E 中 pull 上游裸版本库 B 的提交
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
From /Users/christian/work/tmp/git/B
e6361ed..be5099d master -> origin/master
Updating e6361ed..be5099d
Fast-forward
$git log --oneline #我们看到了 D 中最新的提交 c2 , 这是从上游裸版本库 B 中 pull 下来的。
be5099d c2
472f2ea c1
e6361ed add shell and perl scprit.
dd98199 init repo and add README.md
```

上面例子的这个流程有没有觉得很眼熟 ？ 对，这就是我们平时的开发流程，从一个远程中央版本库拉取(pull)代码,本地修改，然后推送(push)回远程中央版本库。在这里，版本库 B 就是 远程中央版本库。

版本库 B 就是中央版本库。虽然它在本地而不是在"远程"，大家不要觉得奇怪，因为没有人说中央版本库必须在"远程"。

对于 Git 版本库来说，从广义上来讲，除了本身以外，其他的版本库都是远程版本库。每个版本库都是平等的，无非是有的版本库处于同一个本地磁盘，有的在网络上。根据所处的位置不同，Git 会采用不用的通信协议来进行交互。在本地就用本地协议，在网络上就用 SSH ,GIT,HTTP(S),FTP(S)等网络协议。不同的协议对使用来讲具体来说就是 URL 不同，其他的原理和使用方式没有任何不同。

在我们的例子中，因为使用的都是 /path/to/file ,这个是本地协议。在最开始的例子中，git clone git@github.com:christian-tl/git-demo.git ，这个用的就是 HTTP 协议。

不知道大家有没有注意到上面的例子中的这行命令：

`$git push ../B master #我们到 git-demo 中把提交 push 到 B，说明可以向裸版本库 push 数据`

我们在例子中在克隆出来的版本库里向上游 push,从上游 pull 代码。看起来天经地义，因为他们都有个上游可以追溯。但是虽然 B 是 git-demo 的下游，但是对于 git-demo 里说并不知道这个下游，因为在 git-demo 没有任何记录下游的信息。(想想也是，一个仓库可以 clone 出来 N 个新的仓库，无需也不可能记下来所有的仓库)。好啦，这不是我们要说的重点，重点想说的是 git-demo 可以向一个没有什么关联的仓库 push 。所以，向一个版本库 push 代码的完整命令可以写为：

git push 版本库 分支

只要 Git 可以找到这个版本库，就具备向它 push 的条件。这个***版本库***可以有 3 中形式：

1.版本库的路径

版本库的路径就是一个明确的 URL，可以指向本地也可以指向网络上的版本库

```
$cd D
#为了实验，我们在 gitlab 上新建一个名为 git-demo2 的空 project
$git push http://git.dangdang.com/tianle/git-demo2.git master #可以把版本库 D push 到 gitlab 上面的 git-demo2，虽然它们从未建立过任何联系
```

```
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (11/11), 805 bytes | 0 bytes/s, done.
Total 11 (delta 2), reused 0 (delta 0)
To http://git.dangdang.com/tianle/git-demo2.git
* [new branch] master -> master
```

```
$git push ../C master #版本库 C 和 D 也从未建立过关联，但仍然可以 push
Counting objects: 1, done.
Writing objects: 100% (1/1), 170 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To ../C
472f2ea..be5099d master -> master
```

```
$git init F
Initialized empty Git repository in /Users/christian/work/tmp/git/F/.git/
$cd F
$git pull http://git.dangdang.com/tianle/git-demo2.git master #可以直接从 git-demo2 pull 代码，而不是必需先 clone
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 11 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (11/11), done.
From http://git.dangdang.com/tianle/git-demo2
* branch master -> FETCH_HEAD
```

说明，只要 git 能找到命令中的这个版本库，就可以对其 push 和 pull 操作，并不一定需要 clone 来建立关联。

2.远程版本库的名字

我们可以为版本库指定一个名字，而不需要每次都写完整的 URL。如果我们把下面的配置加到 .git/config 配置文件中，就可以给版本库 D 注册一个远程版本库。

```
[remote "git-demo2"]
url = http://git.dangdang.com/tianle/git-demo2.git
fetch = +refs/heads/*:refs/remotes/git-demo2/*
$cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = true
[remote "origin"] #名为 origin 的远程版本库，URL ：/Users/christian/work/tmp/git/B
url = /Users/christian/work/tmp/git/B
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"] #本地 master 分支对应的远程版本库分支
remote = origin
merge = refs/heads/master
```

```
[remote "git-demo2"] #名为 origin 的远程版本库，
URL ： git.dangdang.com/tianle
url = http://git.dangdang.com/tianle/git-demo2.git
fetch = +refs/heads/*:refs/remotes/git-demo2/*


$git fetch git-demo2 #直接写名字，不需要写完整 URL
From http://git.dangdang.com/tianle/git-demo2
* [new branch] master -> git-demo2/master
```

3.省略版本库和分支名

如果 .git/config 文件中有类似下面的配置，那么 执行 ***git push*** 等价于 ***git push origin master*** ,***git pull*** 等价于 ***git push origin pull***

```
[remote "origin"] #名为 origin 的远程版本库，URL ：
/Users/christian/work/tmp/git/B
url = /Users/christian/work/tmp/git/B
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"] #本地 master 分支对应的远程版本库分支
remote = origin
merge = refs/heads/master
```

理论上只要能定位到一个版本库就可以 push/pull 但是**必须满足三个条件:**

1: 必需是裸版本库

2: 有这个版本库 push 和 pull 的权限

3: 满足快进式 Fast-forward 模式,否则无法 push 成功，pull 只会把代码 fetch 下来，但不会 merge

快进式(Fast-forward)

一般情况下，只允许快进式 push.所谓快进式推送，就是要推送的本地版本库的提交是建立在远程版本库相应分支的现有提交基础上的，即远程版本库相应分支的最新提交是本地版本库最新提交的祖先提交。非快进式（non-fast-forward）提交是不被允许的，因为这样每个人都随便提交的话会互相覆盖，会弄乱版本库，版本库无法保证一个完整的提交链。

Tip

广义上来说，当前版本库之外的版本库都是远程版本库，而上游版本库指的是通过 git clone 或 git remote add 所指向的那个版本库。所以上游版本库是远程版本库的一个子集。但是在实际工作中，几乎所有的版本库都是通过 git clone 而来，所以一般情况下远程版本库和上游版本库是同一个意思。

因为每个版本库都是平等的，没有一个严格主次之分，所以说，Git 是分布式的版本库，每个版本库都保存这所有的历史记录

远程版本库

我们现在已经了解了 Git 版本库之间的关系和交互方式，我们也应该清楚了 Git 是分布式的版本库，而不是向 SVN 这种集中式的版本库。各版本库之间没有主次之分，是平等的。

对于一个项目而言，你可以从你团队中任何一个人机器上的版本库上同步代码，也可以向他的版本库推送代码，所以在理论上来说，并不像 SVN 那样需要一台远程中央版本库。

但是在实际工作中，这比较难做到，因为首先不能保证个人的机器都不关机，而且最重要的原因是每个人版本库的提交都不一样，不能今天从 A 那同步代码，明天又向 B 提交代码。

所以就一定要在这个团队中固定一个人的版本库，大家都从他的版本库同步代码，也向他的版本库提交代码。这样大家的代码在能保证是同步更新的。那即使选定了一个固定的人但又不能保证他一直不关机。所以，我们需要找一个空闲的单独的稳定的服务器来做这个版本库，大家都从这里更新，向这个提交。这个版本库就是实际工作中的“远程版本库”。

因为这本版本库一定是在网络上，所以本机上的版本库和它交互就不能通过本地协议了，需要通过 HTTP HTTPS SSH GIT 等网络协议才行。

虽然这是一个远程版本库，但并不是一个主版本库，它和每个人机器上的版本库一样，没啥区别。只不过它在一台稳定的不断电的机器上，并且大家都通过它来协作而已。这个所谓的“远程”版本库就是一个普通的 git 版本库，只不过它恰巧在一个大家都通过网络可以访问到的机器上。

好了，不绕弯子了。我们实际工作中的这个远程版本库就是 git.dangdang.com

我们通过例子看一下是怎么和这个远程版本库交互的，其实这些例子和上面的例子没有任何不同，只是会更具体一些，也对实际工作更有意义。

其实说的还是 ***git clone***， ***git pull***， ***git push***， ***git fetch*** 这些命令，我们现在就来看看这几个命令是怎么应用于实际工作的。

我们要参与一个项目，首先我们要先从 gitlab 上 clone 一个工程到本地。***git clone URL 本地目录***

```
$git clone git@github.com:christian-tl/git-demo.git
$cd git-demo #工程目录名是 git-demo
```

这样我们就把 git-demo 工程克隆到本地的 git-demo 目录。当然我可以指定这个目录的名称

```
$git clone git@github.com:christian-tl/git-demo.git your_code
$cd your_code #工程目录名是 your_code
```

我们注意到远程版本库的后缀是'.git',根据前面介绍的约定 '.git'后缀的版本库是裸版本库。

所以，**远程版本库服务器上面的版本库都是裸版本库。**

克隆下来的版本库会默认把他所克隆的这个版本库注册为上游版本库,并且起名为 **origin** . origin 的英文就是‘起源’的意思，可以望文生义。上游版本库可以叫任何名字，只不过 origin 是 git 默认的上游版本库的名字，当需要写这个名字的地方却省略时，git 默认认为是 origin。除此以外，还做了一个本地 master 分支和远程版本库 master 分支的映射。

就因为有了这两个映射，所以在本地 master 分支上执行 ***git pull origin master*** 等价于 **git pull**， **git push origin master** 等价于 **git push**

```
$cat .git/config
[core]
repositoryformatversion = 0
filemode = true
```

```
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = true
#git clone 之后在本地版本库一定有下面的这 2 个配置节点
[remote "origin"] #注册上游版本库
url = git@github.com:christian-tl/git-demo.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"] #本地 master 分支到 origin 的 master 分支的映射
remote = origin
merge = refs/heads/master
```

和 ***git init*** 一样 **git clone** 之后版本库默认只有一个 master 分支，master 分支指向了 origin/master 分支相同的提交号 。

但是远程有 master 和 develop 两个分支，那我们如何得到 develop 分支呢？等等，是怎么知道远程版本库有 master 和 develop 的呢?

奥秘就存在于文件 **.git/packed-refs** 之中 ,git clone 之后 会把了远程版本库的分支和对应的提交号存在这个文件里

```
$cat .git/packed-refs #这个文件记录了远程版本库的分支和对应的提交号
472f2eaf555b622c4996d7cd17e2337c0c7fc448
refs/remotes/origin/develop
e6361ed35aa40f5bae8bd52867885a2055d60ea2
refs/remotes/origin/master
```

我们得到了这些 commitID 就自然找到了对应的 commit 对象，找到了 commit 对象也就找到了他的所有 tree,blob 对象。也就是得到了这个提交对应的所有文件。

那么这些对象在哪呢？当然是在 .git/objects 里了。这个在前面已经说了无数遍了。

git clone 命令会把远程版本库的 git 对象下载下来到 **.git/objects** 目录里，把分支引用保存到**.git/packed-refs** 文件中。得到了引用和 git 对象，就得到了整个版本库了。

如果这个手册是从头到尾学习的，那么到现在为止理解这个概念应该非常容易了。

现在我们关注下 .git/refs 目录

```
$tree .git/refs/
.git/refs/
├── heads
│ ├── develop
│ └── master
├── remotes
│ └── origin
│ ├── HEAD
└── tags
```

我们知道本地的分支文件在 .git/refs/heads 目录下。从这个目录结构看，聪明的我们一定会想到，远程的分支文件应该
在 .git/refs/heads/remotes/origin 下才对。

本地的分支文件在 .git/refs/heads 目录下，引用指向 .git/refs/objects 目录里的 git 对象，远程的分支在 .git/refs/heads/remotes/origin 下，引用也指向 .git/refs/objects 目录里的 git 对象的话，这样就特别好理解了。可惜，目前看 .git/refs/heads/remotes/origin 里面并没有 master 和 develop 两个分支文件。而是把分支引用保存在 .git/packed-refs 文件里。

其实这只是暂时的，当远程版本库有新提交，并且本地版本库执行了 git pull 或 git fetch 命令后，.git/refs/heads/remotes/origin 目录下会生成对应的分支文件。

其实在 git2.0 之前，git clone 之后就会有这 2 个分支文件，而不是 .git/packed-refs。为什么 git2.0 之后是这样的设计，我也不是很清楚，表示不理解。

好，我们现在知道了 git clone 的原理了。那我们就基于 origin/develop 来建一个本地 develop 分支

```
$git checkout -b develop origin/develop
```

我们来理解下这个命令。

我们知道 `git checkout -b <commit>` 命令可以建立并切换到一个分支。所以 `git checkout -b develop origin/develop` 就是建立 `develop` 分支并指向 `origin/develop` 的提交号。
`origin/develop` 就是远程 `develop` 分支，他的提交号保存在 `.git/packed-refs` 文件里。

```
也可以通过 git rev-parse 命令查看
$git rev-parse origin/develop
472f2eaf555b622c4996d7cd17e2337c0c7fc448
```

其实就是想说明，git clone 把远程版本库的 git 对象和分支引用下载到本地后，一切的操作和本地的任何操作是一样的。

唯一区别是，分支文件的目录不同罢了

```
本地分支文件保存在 .git/refs/heads 目录
远程分支文件保存在 .git/refs/heads/origin 目录
```

这下，明白了吗

```
让我们在看下现在 .git/config 文件的内容
$cat .git/config
...
[remote "origin"]
url = git@github.com:christian-tl/git-demo.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
[branch "develop"] #多了一个新的[branch]节点
remote = origin
merge = refs/heads/develop
git checkout -b develop origin/develop 命令会自动为新建的分支设置一个上游(upstream)对应分支.
```

好了，现在我们本地有两个分支 master 和 develop 分别对应着远程的 master,develop 分支。实际工作中就是这么做的，保持同样的名字是为了规范便于记忆，方便管理。

但是本地分支和远程分支的映射的名字并没有任何限制。做个简单实验说明一下

```
$git checkout -b abc123 origin/develop
Branch abc123 set up to track remote branch develop from origin.
Switched to a new branch 'abc123'
$cat .git/config
...
[remote "origin"]
url = git@github.com:christian-tl/git-demo.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
[branch "develop"]
remote = origin
merge = refs/heads/develop
[branch "abc123"] #多了一个新的映射
remote = origin
merge = refs/heads/develop
```

好了，到这里聪明的你已经不需要我在过多解释了。

git clone 之后本地就会有一个名为 origin 的上游，上游的分支信息保存在 .git/refs/remotes/origin 目录下 。

其实 git 版本库可以有 N 个上游版本库，这完全取决于实际的需要。

git remote add 上游名 URL 命令可以添加上游版本库。现在我们就用这个命令再添加一个上游版本库，随便起个名字比如叫 upstream2

```
$git remote add upstream2 http://git.dangdang.com/tianle/git-demo2.git
$cat .git/config
...
[remote "origin"]
url = git@github.com:christian-tl/git-demo.git
fetch = +refs/heads/*:refs/remotes/origin/*
#多了一个[remote]节点，名为 upstream2
[remote "upstream2"]
url = http://git.dangdang.com/tianle/git-demo2.git
fetch = +refs/heads/*:refs/remotes/upstream2/*
```

我们还可以修改、重命名和删除一个上游版本库

```
$git remote set-url upstream2 git@github.com:christian-tl/git-demo.git #修改 upstream2 上游版本库的 URL
$git remote rename upstream2 upstream1 #把 upstream2 重命名成 upstream1
$cat .git/config
...
[remote "origin"]
url = git@github.com:christian-tl/git-demo.git
fetch = +refs/heads/*:refs/remotes/origin/*
...
[remote "upstream1"] #名字变为了 upstream1，之前是 upstream2
url = git@github.com:christian-tl/git-demo.git #URL 变成了 git-demo.git ，之前是 git-demo2.git
fetch = +refs/heads/*:refs/remotes/upstream1/*
```

```
$git remote rm upstream1 #删除上游版本库 upstream1，.git/config 文件的[remote "upstream1"]节点会被删掉
```

总结一下，git remote 的相关命令就是修改 .git/config 文件而已，其实我们也可以手动编辑完成。

git fetch 命令可以把上游版本库的更新下载到本地版本库，这个更新包括 Git 对象和分支引用。

当上游版本库有新的提交后，上游版本库相关的分支文件会被更新，objects 目录里会多了最新提交所对应的 commit , tree,和 blob 对象。git fetch 命令就是负责把这些更新下载到本地版本库。

现在我们通过一个实验来直观的看一下 git fetch 干了什么，并且这个实验可以把前面的一些命令串起来。

实验步骤：

1. 克隆一份新版本库 demo-for-fetch,demo-for-fetch 向 develop 分支推送一个新的提交
2. 在本地老的版本库里 git fetch

```
#克隆一份新版本库 demo-for-fetch,demo-for-fetch 向 develop 分支推送一个新的提交
$git clone git@github.com:christian-tl/git-demo.git demo-for-fetch

$cd demo-for-fetch

$git checkout -b develop origin/develop
```

```
$echo '*.jar' >> .gitignore ; echo '*.war' >> .gitignore ; echo '*.ear' >> .gitignore; echo '*/target/*' >> .gitignore
$git add .
$git commit -m 'add .gitignore file'
$git push origin develop
$cat .git/refs/heads/develop
d83003a2e746416d37101acd6d8fbb8557aab63e #develop 分支的提交号是 d83003a
$cd git-demo
$git fetch
remote: Counting objects: 3, done. #下载了 3 个对象文件
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From http://git.dangdang.com/tianle/git-demo
472f2ea..d83003a develop -> origin/develop #更新分支文件 注意提交号 d83003a

$tree .git/refs/remotes/origin/
.git/refs/remotes/origin/
├── HEAD
└── develop #fetch 后 在 refs/remotes/origin 生成了一个分支文件

$cat .git/refs/remotes/origin/develop
d83003a2e746416d37101acd6d8fbb8557aab63e #origin/develop 分支的提交号是 d83003a
```

在我们的例子中远程版本库 master 并没有新的提交，所以 fetch 后本地 refs/remotes/origin 目录并没有生成一个 master 分支文件。有兴趣的话可以查看下 .git/objects 目录，看看不是不多了 3 个文件。

篇幅所限，不在这里罗列了。

git pull 命令等价于 ***git fetch + git merge***

git pull 命令先把远程版本库的更新下载到本地，然后和本地的分支合并 。

```
$git pull origin develop
#上面这条命令和用下面的两条命令的做的事情一样
$git fetch
$git merge origin/develop
```

所以 git pull 命令是分成了来给你个阶段完成的。阶段一从远程下载更新，需要网络。阶段二完全是正常分支间的合并操作，都是本地完成，不需要网络。

git pull 命令是工作中最重要，最常用的命令之一，但是真的没什么需要过多描述的，因为他是两个命令的组合，fetch 命令详细介绍完了，merge 命令会在后面详细介绍

git push 命令会把本地版本库的 git 对象上传到远程版本库，并用本地分支引用更新远程版本库对应分支的引用。

当然不是把本地的所有对象都上传，而只是新提交所对应的那些 git 对象。

在 push 之前，必需要获得远程版本库的最新提交信息，而这个提交必需是我们本地提交的父提交，这就是前面提到的 快进式(Fast-forward)，否则不能 push 成功。

在 push 操作时，必需要指定一个远程的分支，也就是要把提交推送到一个明确的分支。 所以在 push 之前，我们要先取得这个远程分支的提交并作为我们本地提交的祖先提交。

在这个祖先提交之后所有提交就是最新提交。push 就是把这些提交对应的 git 对象上传到远程版本库。

所以，在 push 之前，一定要先 pull 。git pull 会取得最新提交并和当前分支 merge,merge 之后最新提交就是本地提交的祖先提交了。

在为本地分支建立了一个上游分支之后可以通过 git status 查看本地分支领先了远程分支几个 commit

```
$git commit --allow-empty -m 'c2'
$git status
On branch develop
Your branch is ahead of 'origin/develop' by 1 commit. #仔细看这一行提示,现在 origin/develop 分支之后有了 1 次 commit
(use "git push" to publish your local commits)
nothing to commit, working tree clean
```

```
$git commit --allow-empty -m 'c3'
$git status
On branch develop
Your branch is ahead of 'origin/develop' by 2 commits. #仔细看这一行提示,现在 origin/develop 分支之后有了 2 次 commit
(use "git push" to publish your local commits)
nothing to commit, working tree clean
```

现在我们执行 git push origin develop 会成功吗？答案是 不一定。

由于我们的本地 develop 分支是基于 origin/develop 分支创建的，所以这之后的提交的祖先提交就是 origin/develop 分支所对应的提交。

如果在这个时间段内，远程版本库没有新的提交，那么就符合快进式提交条件，可以 push 成功。如果这个时间段内远程有新的提交，就不能 push 成功。

所以 ， 在每次 push 前 一定要先 pull 一下，同步一下远程版本库的最新信息。

```
#因为目前除了我没有人会更新 git-demo,所以我们的 push 当然时成功的啦
$git push origin develop
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 256 bytes | 0 bytes/s, done.
```

```
Total 2 (delta 1), reused 0 (delta 0)
remote:
remote: Create merge request for develop:
remote: http://git.dangdang.com/tianle/git-demo/merge_requests/new?merge_request%5Bsource_branch%5D=develop
remote:
To git@github.com:christian-tl/git-demo.git
d83003a..05d68dd develop -> develop
```

push 不光可以推送到远程已经存在的分支，也可以推送到不存在的分支，当推送到不存在分支时，就创建了这个远程分支。

push 到已经存在的分支时需要受到快进式(Fast-forward)的限制，但是 push 到不存在的分支时一定是成功的。

```
$git checkout develop
$git push origin develop:feature
* [new branch] develop -> feature #这样就在远程版本库创建了一个feature 新分支
```

所以 push 命令的比较完整的写法是 ***git push 上游版本库 本地分支:远程分支***，当本地分支于远程分支同名时只需 ***git push 上游版本库 本地分支***

```
$git push origin :feature # git push 上游版本库 :远程分支 可以删除远程分支
To git@github.com:christian-tl/git-demo.git
- [deleted] feature
```

常见问题简答

如何创建 Git 版本库?

git init 或 git clone

如何为当前本地版本库添加远程版本库?

git remote add origin URL

git checkout 和 git reset 的区别 ?

checkout 修改 HEAD 文件的内容，reset 修改 refs/heads 下面分支文件的内容。同时他们都可以替换工作区和暂存区的文件。

git merge 和 git rebase 的区别?

都可以合并分支，但是一般建议用 merge。因为 rebase 会改变已经生成的提交号，如果这个分支是大家协作的分支，那么你改了提交号之后会带来整个分支的混乱。

SHA 值是怎么生成的?

通过 SHA1 消息摘要算法，可以使用系统命令 sha1sum

如何产生公钥?

```
ssh-keygen -t rsa -C "tianle3@jd.com"
cat ~/.ssh/id_rsa.pub
```

如何产生 commit-ID？

(printf "commit %s\0" $(git --no-replace-objects cat-file commit HEAD | wc -c); git cat-file commit HEAD) | sha1sum