Git 工作方式

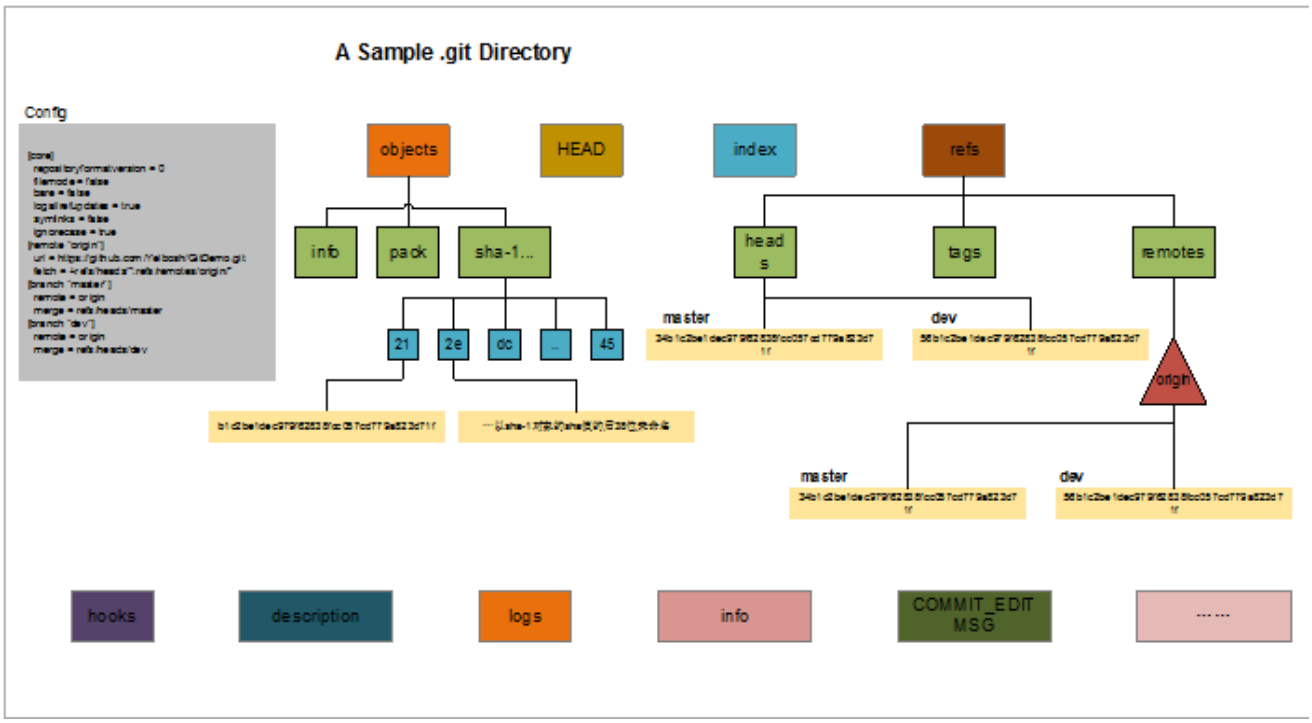
目录

Git 基本原理

本质上，Git 是一套内容寻址文件系统，而我们直接接触的 Git 界面是封装在其之上的一个应用层。应用层相关的命令（如git commit、 git push等）是通过底层相关的命令（如git hash-object、git update-index等）实现的。

Git 的目录结构

Git 在初始化的时候（`git init`）会生成一个`.git`隐藏文件夹，而 Git 进行版本控制所需要的文件全部放在这个文件夹中。`.git`目录中有很多的文件和文件夹，每一个文件都有各自的作用。.git 文件夹像是一本书，每一个版本的每一个变动都存储在这本书中，而且这本书还有一个目录，指明了不同的版本的变动内容存储在这本书的哪一页上。



.git 文件夹中的重要文件

config 文件

该文件主要记录针对该项目的一些配置信息，例如是否以 bare 方式初始化、remote 的信息等，通过`git remote add`命令增加的远程分支的信息就保存在这里

一个典型的 config 文件如下：

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = git@github.com:docxit/docxit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

objects 文件夹

该文件夹主要包含 git 对象。git 对象用于保存 Git 中的文件和一些操作，分为 BLOB、tree 和 commit 三种类型，而各个版本之间是通过版本树来组织的，比如当前的 HEAD 会指向某个 commit 对象，而该 commit 对象又会指向几个 BLOB 对象或者 tree 对象。

objects 文件夹中包含很多的子文件夹，其中 Git 对象保存在以其 sha-1 值的前两位为子文件夹、后 38 位为文件名的文件中；除此以外，Git 会定期对 Git 对象进行压缩和打包以节省磁盘空间占用，其中 `pack` 文件夹用于存储打包压缩的对象，而 `info` 文件夹用于从打包的文件中查找 git 对象

HEAD 文件

该文件指明了当前分支的结果，比如当前分支是 master，则该文件就会指向 master，存储分支在 refs 中的表示。一个典型的 HEAD 文件如下：

```
ref: refs/heads/dev
```

index 文件

该文件保存了暂存区域整个目录树的信息，内容包括它指向的文件的时间戳、文件名、sha1值等，记录从项目初始化到目前为止，项目仓库中所有文件最后一次修改时刻的时间戳以及对应的长度信息，因此随着加入仓库中的文件不断增多，index 文件也会不断增大。主要格式为：

```
Index魔数（DIRC） + 版本号 + 暂存的文件个数 + 每个文件的时间戳和长度
```

refs 文件夹

该文件夹存储指向数据（分支）的提交对象的指针。其中 `heads` 文件夹中每一个文件存储本地一个分支最近一次 commit （commit 对象）的 sha-1 值； `remotes` 文件夹则记录最后一次和每一个远程仓库的通信，Git 会把你最后一次推送到这个 remote 的每个分支的值都记录在这个文件夹中；tag 文件夹则是分支的别名。如在目录 `.git/refs/heads` 中，有两个文件 `dev` 和 `master`，内容均为

```
6f35f9052510ff162d0121c7238897a253669cad
```

他们指向同一个 commit 对象，该对象为 `.git/objects/6f/35f9052510ff162d0121c7238897a253669cad`。

logs 文件夹

logs 中的文件记录了本地仓库和远程仓库的每一个分支的提交记录，即所有的 commit 对象（包括时间、作者等信息）都会被记录在这个文件夹中，其中的 `HEAD` 文件可以是：

```
0000000000000000000000000000000000000000 ad189e3e153ef397c2fcc43abfbd2ef3c608ff75
yxyyyxxyy <yixinyu@mail.ustc.edu.cn> 1537025575 +0800 clone: from
git@github.com:docxit/docxit.git
ad189e3e153ef397c2fcc43abfbd2ef3c608ff75 1fbdfba6813d7f7f82f272861910f0b5f9852b1a
yxyyyxxyy <yixinyu@mail.ustc.edu.cn> 1537426369 +0800 commit: add research report
...
565f1b5cbdd7695a242d2e25604b4e003513d1ba b750a89bedb347cb43c4752a7f4941a787dae3ab
yxyyyxxyy <yixinyu@mail.ustc.edu.cn> 1539872347 +0800 reset: moving to HEAD^
6f35f9052510ff162d0121c7238897a253669cad 6f35f9052510ff162d0121c7238897a253669cad
yxyyyxxyy <yixinyu@mail.ustc.edu.cn> 1539917895 +0800 checkout: moving from master to
dev
```

info 文件夹

info 文件夹保存了一份不希望在 `.gitignore` 文件中管理的忽略模式的全局可执行文件，不是很常用。
`info/exclude` 初始为：

```
# git ls-files --others --exclude-from=.git/info/exclude
# Lines that start with '#' are comments.
# For a project mostly in C, the following would be a good set of
# exclude patterns (uncomment them if you want to use them):
# *.[oa]
# *~
```

COMMIT_EDITMSG 文件

COMMIT_EDITMSG 文件记录了最后一次提交时的注释信息，如：

```
fix tab bug
```

这是我最后一次 commit 提交的注释信息。

Git 对象

Git 是一套内容寻址文件系统，采用 HashTable 的方式进行寻址，即简单的存储键值对（key - value）。其中，key 就是文件（头+内容）的哈希值（采用 sha-1 的方式，40 位），value 就是经过压缩后的文件内容。Git 对象的类型包括 BLOB 对象、tree 对象、commit 对象。

Git 对象类型

BLOB 对象

全称为 binary
large object，可以存储几乎所有的文件类型，这种对象类型和数据库中的 BLOB 类型（经常用来在数据库中存储图片、视频等）是一样的，当作一种数据类型即可

tree 对象

用来组织 BLOB 对象的一种数据类型，可以把它想象成二叉树中的树节点，只不过 Git 中的树是多叉树

commit 对象

由 tree 对象衍生，每一个 commit 对象表示一次提交，在创建的过程中可以指定该 commit 对象的父节点，这样所有的 commit 操作便可以连接在一起，这些 commit 对象便组成了提交树，branch 是这个树中的某一个子树。

Git 对象存储方式

Git 将文件头与原始数据内容拼接起来，计算拼接后的新内容的 40 位的 sha-1 校验和，将该校验和的前2位作为 object 目录中的子目录的名称，后 38 位作为子目录中的文件名:

$$Key = sha1(file\_header + file\_content)$$

然后，Git 用 zlib 的方式对数据内容进行压缩，最后将用 zlib 压缩后的内容写入磁盘。

$$Value = zlib(file\_content)$$

文件头的格式为 "blob #{content.length}\0"，例如 "blob 16\000"。对于 tree 对象和 commit 对象，文件头的格式都是一样的，但是其文件数据却是有固定格式的。tree 对象类似于树中节点的定义，在 tree 对象中要包含对连接的 BLOB 对象的引用，而 commit 对象与 tree 对象类似，要包含提交的 tree 对象的引用。

对象暂存区

暂存区（也叫索引库），`.git/index` 文件记录该暂存区中的文件索引。`git add` 首先对每一个文件计算校验和（SHA-1 哈希字串），通过 `hash-object` 命令将需要暂存的文件进行 key-value 化转换成 Git 对象，并进行存储，即把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 类型的对象存储这些快照）；然后，通过 `update-index` 命令将这些对象加入到索引库进行暂存，即每个文件对应的当前版本的 key 也会加入到 index 文件中。这样便完成了Git文件的暂存操作。

在暂存操作中会用到的低层次命令有:

```
git hash-object          # 获取指定文件的 key，带上 -w 会将该对象的 value 进行存储
git update-index         # 将指定的 object 加入索引库，需要带上 -add 选项
git cat-file –p/-t key   # 获取指定key的对象信息，-p 打印详细信息，-t 打印对象的类型
```

我们执行

```
git init
echo "version 1" > version.txt
git hash-object -w version.txt  # 将该文件转换为 Git 对象并存储
```

这时 `hash-object` 命令会返回该 Git 对象的 Key 值，`.git/objects` 目录下多了一个前两位数字子目录，该目录中的文件名称为该 key 值的后 38 位。之后执行

```
git update-index --add --cacheinfo 100644 <Key值> version.txt    # 索引化
```

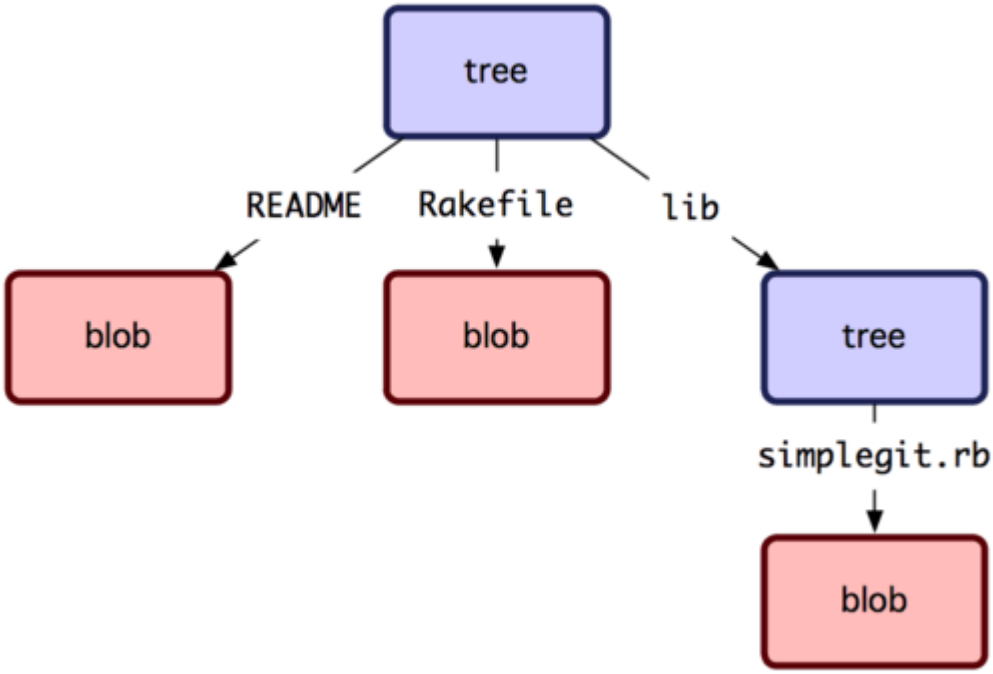
之后可以发现 .git 目录下多了 index 文件，并且在以后每次 `update-index` 命令执行之后，该 index 文件的内容都会发生变化。每次调用 `git add` 命令，都会把 add 的文件的索引信息（时间戳和大小）进行更新，而我们所使用的 `git status` 命令，则会把每一个文件的索引信息和上次提交的索引信息进行比较，如果发生了变化，就会显示出来。

tree 对象

一个单独的 tree 对象包含一条或多条 tree 记录，每一条记录含有一个指向 BLOB 对象或子 tree 对象的 sha-1 指针（也就是一个 40 位的 key 值），并附有该对象的权限模式
、类型和文件名信息。

```
git write-tree      # 根据索引库中的信息创建 tree 对象
```

commit 对象所对应的 tree 对象永远都是工作目录的根 tree 对象。每个子目录都对应一个 tree 对象，每个文件对应一个 BLOB 对象，因此整个工作目录对应一棵 Git 对象树，根节点就是 commit 对象所引用的 tree 节点，而每个子文件夹又分别对应一棵子树。所以任何一个文件的更改，都会导致其上层所有父对象的更改和重新存储。



commit 对象

每一次 commit 都对应一个 commit 对象，而一个 commit 对象对应一个 tree 对象。你现在有很多 tree 对象，它们指向了要跟踪的项目的不同快照，可是先前的问题依然存在：必须记住三个 SHA-1 值以获得这些快照。你也没有关于谁、何时以及为何保存了这些快照的信息。commit 对象保存了这些基本信息。创建 commit 对象需要使用如下命令：

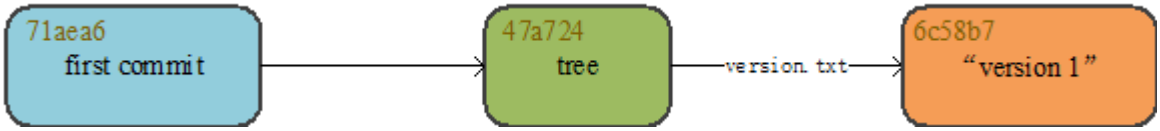
```
# 根据 tree 对象创建 commit 对象，-p 表示前继 commit 对象
git commit-tree key –p key2
```

第一次提交不需要带上 -p 选项来指明父节点。两个 commit 节点连接在一起，而不断的连接便构成了一棵树，即提交树。commit 对象中包含了与之关联的 tree 对象的 key 值，以及 author 和 committer 的信息。如：

```
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

所创建的所有对象的关系如下图所示：



Git 引用

我们需要一个文件来用一个简单的名字来记录不容易记住的 SHA-1 值，这样就可以用这些指针而不是原来的 SHA-1 值去检索了。可以使用

```
git update-ref refs/heads/<name> <key值>
```

来创建引用。在 `.git/refs/heads/` 目录下就会有相应名字的文件，内容为对应的 key 值。基本上 Git 中的一个分支其实就是一个指向某个工作版本一条 HEAD 记录的指针或引用。每当你执行 `git branch <分支名称>` 这样的命令，Git 基本上就是执行 `update-ref` 命令，把你现在所在分支中最后一次提交的 SHA-1 值，添加到你要创建的分支的引用。

HEAD

当执行 `git branch <分支名称>` 这条命令的时候，Git 要通过 HEAD 文件得到最后一次提交的 SHA-1 值。HEAD 文件是一个指向当前所在分支的引用标识符。这样的引用标识符并不包含 SHA-1 值，而是一个指向另外一个引用的指针。如果执行 `git checkout test`，Git 就会更新 `.git/HEAD` 文件的内容为 `ref: refs/heads/test`。这时执行 `git commit` 命令，它就创建了一个 commit 对象，把这个 commit 对象的父级设置为 HEAD 指向的引用的 SHA-1 值。

操作 HEAD 的低层次命令有：

```
git symbolic-ref HEAD                   # 读取 HEAD 的值
git symbolic-ref HEAD refs/heads/test   # 设置 HEAD 的值
```

注意，HEAD 不能设置成 refs 以外的形式

Tags

Tag 对象非常像一个 commit 对象——包含一个标签，一组数据，一个消息和一个指针。最主要的区别就是 Tag 对象指向一个 commit 而不是一个 tree。它就像是一个分支引用，但是不会变化——永远指向同一个 commit，仅仅是提供一个更加友好的名字。

Tag 有两种类型：annotated（可以标记任何 Git 对象） 和 lightweight （标记一个永远不变的 commit）。可以用类似下面这样的命令进行操作

```
git update-ref refs/tags/v1.0 <key值>    # 创建 lightweight tag
git tag -a v1.1 <key值> -m 'test tag'    # 创建 annotated tag
```

Remotes

Git 会把最后一次推送到 remote 的每个分支的值都记录在 `refs/remotes` 目录下。Remote 引用和分支主要区别在于他们是不能被 check out 的。Git 把他们当作是标记了这些分支在服务器上最后状态的一种书签。

Git 打包

Git 往磁盘保存对象时默认使用的格式叫松散对象 (loose object) 格式。Git 时不时地将这些对象打包至一个叫 packfile 的二进制文件以节省空间并提高效率。当仓库中有太多的松散对象，或是手工调用 `git gc` 命令，或推送至远程服务器时，Git 都会这样做。

Git 认为未被任何 commit 引用的 blob 是 "悬空" 的，不会将它们打包进 packfile 。剩下的文件是新创建的 packfile 以及一个索引。packfile 文件包含了刚才从文件系统中移除的所有对象。索引文件包含了 packfile 的偏移信息，这样就可以快速定位任意一个指定对象。

Git 打包对象时，会查找命名及尺寸相近的文件，并只保存文件不同版本之间的差异内容。`git verify-pack` 命令用于显示已打包的内容。