返回 码云 首页(目录) 章节列表 ▾ Pro Git（中文版）

9 Git 内部原理

不管你是从前面的章节直接跳到了本章，还是读完了其余各章一直到这，你都将在本章见识 Git 的内部工作原理和实现方式。我个人发现学习这些内容对于理解 Git 的用处和强大是非常重要的，不过也有人认为这些内容对于初学者来说可能难以理解且过于复杂。正因如此我把这部分内容放在最后一章，你在学习过程中可以先阅读这部分，也可以晚点阅读这部分，这完全取决于你自己。

既然已经读到这了，就让我们开始吧。首先要弄明白一点，从根本上来讲 Git 是一套内容寻址 (content-addressable) 文件系统，在此之上提供了一个 VCS 用户界面。马上你就会学到这意味着什么。

早期的 Git (主要是 1.5 之前版本) 的用户界面要比现在复杂得多，这是因为它更侧重于成为文件系统而不是一套更精致的 VCS 。最近几年改进了 UI 从而使它跟其他任何系统一样清晰易用。即便如此，还是经常会有一些陈腔滥调提到早期 Git 的 UI 复杂又难学。

内容寻址文件系统层相当酷，在本章中我会先讲解这部分。随后你会学到传输机制和最终要使用的各种库管理任务。

9.1 底层命令 (Plumbing) 和高层命令 (Porcelain)

本书讲解了使用 `checkout`, `branch`, `remote` 等共约 30 个 Git 命令。然而由于 Git 一开始被设计成供 VCS 使用的工具集而不是一整套用户友好的 VCS，它还包含了许多底层命令，这些命令用于以 UNIX 风格使用或由脚本调用。这些命令一般被称为 "plumbing" 命令（底层命令），其他的更友好的命令则被称为 "porcelain" 命令（高层命令）。

本书前八章主要专门讨论高层命令。本章将主要讨论底层命令以理解 Git 的内部工作机制、演示 Git 如何及为何要以这种方式工作。这些命令主要不是用来从命令行手工使用的，更多的是用来为其他工具和自定义脚本服务的。

当你在一个新目录或已有目录内执行 `git init` 时，Git 会创建一个 `.git` 目录，几乎所有 Git 存储和操作的内容都位于该目录下。如果你要备份或复制一个库，基本上将这一目录拷贝至其他地方就可以了。本章基本上都讨论该目录下的内容。该目录结构如下：

```
$ ls
    HEAD
    branches/
    config
    description
    hooks/
    index
    info/
    objects/
    refs/
```

该目录下有可能还有其他文件，但这是一个全新的 `git init` 生成的库，所以默认情况下这些就是你能看到的结构。新版本的 Git 不再使用 `branches` 目录，`description` 文件仅供 GitWeb 程序使用，所以不用关心这些内容。`config` 文件包含了项目特有的配置选项，`info` 目录保存了一份不希望在

.gitignore 文件中管理的忽略模式 (ignored patterns) 的全局可执行文件。`hooks` 目录保存了第七章详细介绍了的客户端或服务端钩子脚本。

另外还有四个重要的文件或目录：`HEAD` 及 `index` 文件，`objects` 及 `refs` 目录。这些是 Git 的核心部分。`objects` 目录存储所有数据内容，`refs` 目录存储指向数据 (分支) 的提交对象的指针，`HEAD` 文件指向当前分支，`index` 文件保存了暂存区域信息。马上你将详细了解 Git 是如何操纵这些内容的。

9.2 Git 对象

Git 是一套内容寻址文件系统。很不错。不过这是什么意思呢？ 这种说法的意思是，Git 从核心上来看不过是简单地存储键值对（key-value）。它允许插入任意类型的内容，并会返回一个键值，通过该键值可以在任何时候再取出该内容。可以通过底层命令 `hash-object` 来示范这点，传一些数据给该命令，它会将数据保存在 `.git` 目录并返回表示这些数据的键值。首先初使化一个 Git 仓库并确认 `objects` 目录是空的：

```
$ mkdir test
    $ cd test
    $ git init
    Initialized empty Git repository in /tmp/test/.git/
    $ find .git/objects
    .git/objects
    .git/objects/info
    .git/objects/pack
    $ find .git/objects -type f
    $
```

Git 初始化了 `objects` 目录，同时在该目录下创建了 `pack` 和 `info` 子目录，但是该目录下没有其他常规文件。我们往这个 Git 数据库里存储一些文本：

```
$ echo 'test content' | git hash-object -w --stdin
    d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

参数 `-w` 指示 `hash-object` 命令存储 (数据) 对象，若不指定这个参数该命令仅仅返回键值。`--stdin` 指定从标准输入设备 (stdin) 来读取内容，若不指定这个参数则需指定一个要存储的文件的路径。该命令输出长度为 40 个字符的校验和。这是个 SHA-1 哈希值——其值为要存储的数据加上你马上会了解到的一种头信息的校验和。现在可以查看到 Git 已经存储了数据：

```
$ find .git/objects -type f
    .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

可以在 `objects` 目录下看到一个文件。这便是 Git 存储数据内容的方式——为每份内容生成一个文件，取得该内容与头信息的 SHA-1 校验和，创建以该校验和前两个字符为名称的子目录，并以 (校验和) 剩下 38 个字符为文件命名 (保存至子目录下)。

通过 `cat-file` 命令可以将数据内容取回。该命令是查看 Git 对象的瑞士军刀。传入 `-p` 参数可以让该命令输出数据内容的类型：

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
    test content
```

可以往 Git 中添加更多内容并取回了。也可以直接添加文件。比方说可以对一个文件进行简单的版本控制。首先，创建一个新文件，并把文件内容存储到数据库中：

```
$ echo 'version 1' > test.txt
    $ git hash-object -w test.txt
    83baae61804e65cc73a7201a7252750c76066a30
```

接着往该文件中写入一些新内容并再次保存：

```
$ echo 'version 2' > test.txt
    $ git hash-object -w test.txt
    1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

数据库中已经将文件的两个新版本连同一开始的内容保存下来了：

```
$ find .git/objects -type f
    .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
    .git/objects/83/baae61804e65cc73a7201a7252750c76066a30
    .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

再将文件恢复到第一个版本：

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
    $ cat test.txt
    version 1
```

或恢复到第二个版本：

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
    $ cat test.txt
    version 2
```

需要记住的是几个版本的文件 SHA-1 值可能与实际的值不同，其次，存储的并不是文件名而仅仅是文件内容。这种对象类型称为 blob 。通过传递 SHA-1 值给 `cat-file -t` 命令可以让 Git 返回任何对象的类型：

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
    blob
```

tree (树) 对象

接下去来看 tree 对象，tree 对象可以存储文件名，同时也允许存储一组文件。Git 以一种类似 UNIX 文件系统但更简单的方式来存储内容。所有内容以 tree 或 blob 对象存储，其中 tree 对象对应于 UNIX 中的目录，blob 对象则大致对应于 inodes 或文件内容。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 或子 tree 对象的 SHA-1 指针，并附有该对象的权限模式 (mode)、类型和文件名信息。以 simplegit 项目为例，最新的 tree 可能是这个样子：

```
$ git cat-file -p master^{tree}
    100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
    100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
    040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

`master^{tree}` 表示 `branch` 分支上最新提交指向的 tree 对象。请注意 `lib` 子目录并非一个 blob 对象，而是一个指向别一个 tree 对象的指针：

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
    100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

从概念上来讲，Git 保存的数据如图 9-1 所示。

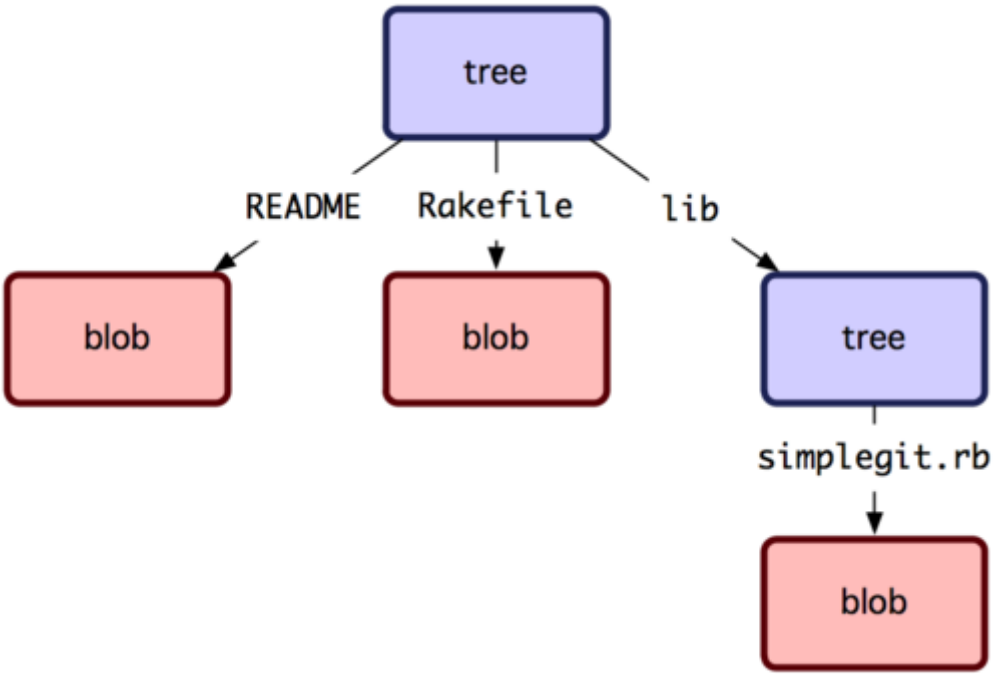

图 9-1. Git 对象模型的简化版

你可以自己创建 tree 。通常 Git 根据你的暂存区域或 index 来创建并写入一个 tree 。因此要创建一个 tree 对象的话首先要通过将一些文件暂存从而创建一个 index 。可以使用 plumbing 命令 `update-index` 为一个单独文件 —— test.txt 文件的第一个版本 —— 创建一个 index 。通过该命令人为的将 test.txt 文件的首个版本加入到了一个新的暂存区域中。由于该文件原先并不在暂存区域中 (甚至就连暂存区域也还没被创建出来呢) ，必须传入 `--add` 参数;由于要添加的文件并不在当前目录下而是在数据库中，必须传入 `--cacheinfo` 参数。同时指定了文件模式，SHA-1 值和文件名：

```
$ git update-index --add --cacheinfo 100644 \
    83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

在本例中，指定了文件模式为 `100644` ，表明这是一个普通文件。其他可用的模式有：`100755` 表示可执行文件，`120000` 表示符号链接。文件模式是从常规的 UNIX 文件模式中参考来的，但是没有那么灵活 —— 上述三种模式仅对 Git 中的文件 (blobs) 有效 (虽然也有其他模式用于目录和子模块)。

现在可以用 `write-tree` 命令将暂存区域的内容写到一个 tree 对象了。无需 `-w` 参数 —— 如果目标 tree 不存在，调用 `write-tree` 会自动根据 index 状态创建一个 tree 对象。

```
$ git write-tree
    d8329fc1cc938780ffdd9f94e0d364e0ea74f579
    $ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
    100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

可以这样验证这确实是一个 tree 对象：

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
    tree
```

再根据 test.txt 的第二个版本以及一个新文件创建一个新 tree 对象：

```
$ echo 'new file' > new.txt
    $ git update-index test.txt
    $ git update-index --add new.txt
```

这时暂存区域中包含了 test.txt 的新版本及一个新文件 new.txt 。创建 (写) 该 tree 对象 (将暂存区域或 index 状态写入到一个 tree 对象)，然后瞧瞧它的样子：

```
$ git write-tree
    0155eb4229851634a0f03eb265b69f5a2d56f341
    $ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
    100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
    100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

请注意该 tree 对象包含了两个文件记录，且 test.txt 的 SHA 值是早先值的 "第二版" (`1f7a7a`)。来点更有趣的，你将把第一个 tree 对象作为一个子目录加进该 tree 中。可以用 `read-tree` 命令将 tree 对象读到暂存区域中去。在这时，通过传一个 `--prefix` 参数给 `read-tree` ，将一个已有的 tree 对象作为一个子 tree 读到暂存区域中：

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
    $ git write-tree
    3c4e9cd789d88d8d89c1073707c3585e41b0e614
    $ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
    040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak
    100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
    100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

如果从刚写入的新 tree 对象创建一个工作目录，将得到位于工作目录顶级的两个文件和一个名为 `bak` 的子目录，该子目录包含了 test.txt 文件的第一个版本。可以将 Git 用来包含这些内容的数据想象成如图 9-2 所示的样子。

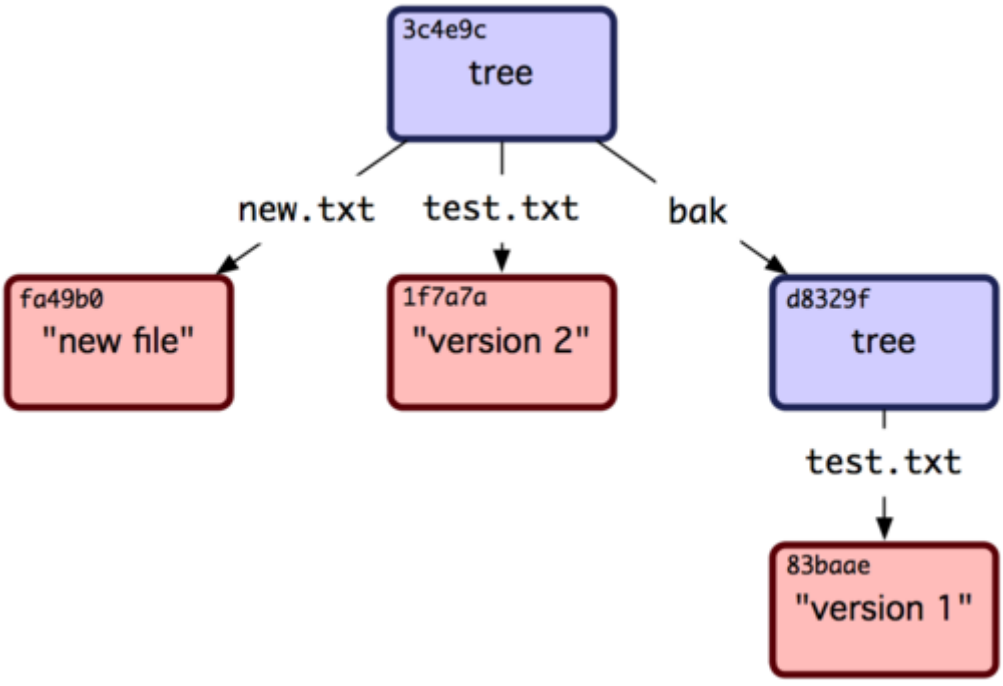

图 9-2. 当前 Git 数据的内容结构

commit (提交) 对象

你现在有三个 tree 对象，它们指向了你要跟踪的项目的不同快照，可是先前的问题依然存在：必须记往三个 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 对象为你保存了这些基本信息。

要创建一个 commit 对象，使用 `commit-tree` 命令，指定一个 tree 的 SHA-1，如果有任何前继提交对象，也可以指定。从你写的第一个 tree 开始：

```
$ echo 'first commit' | git commit-tree d8329f
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

通过 `cat-file` 查看这个新 commit 对象：

```
$ git cat-file -p fdf4fc3
    tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
    author Scott Chacon <schacon@gmail.com> 1243040974 -0700
    committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

    first commit
```

commit 对象有格式很简单：指明了该时间点项目快照的顶层树对象、作者/提交者信息（从 Git 设置的 `user.name` 和 `user.email`中获得)以及当前时间戳、一个空行，以及提交注释信息。

接着再写入另外两个 commit 对象，每一个都指定其之前的那个 commit 对象：

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
    cac0cab538b970a37ea1e769cbbde608743bc96d
    $ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
    1a410efbd13591db07496601ebc7a059dd55cfe9
```

每一个 commit 对象都指向了你创建的树对象快照。出乎意料的是，现在已经有了真实的 Git 历史了，所以如果运行 `git log` 命令并指定最后那个 commit 对象的 SHA-1 便可以查看历史：

```
$ git log --stat 1a410e
    commit 1a410efbd13591db07496601ebc7a059dd55cfe9
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri May 22 18:15:24 2009 -0700

    third commit

    bak/test.txt | 1 +
    1 files changed, 1 insertions(+), 0 deletions(-)

    commit cac0cab538b970a37ea1e769cbbde608743bc96d
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri May 22 18:14:29 2009 -0700

    second commit

    new.txt | 1 +
    test.txt | 2 +-
    2 files changed, 2 insertions(+), 1 deletions(-)

    commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri May 22 18:09:34 2009 -0700

    first commit

    test.txt | 1 +
    1 files changed, 1 insertions(+), 0 deletions(-)
```

真棒。你刚刚通过使用低级操作而不是那些普通命令创建了一个 Git 历史。这基本上就是运行 `git add` 和 `git commit` 命令时 Git 进行的工作 —保存修改了的文件的 blob，更新索引，创建 tree 对象，最后创建 commit 对象，这些 commit 对象指向了顶层 tree 对象以及先前的 commit 对象。这三类 Git 对象 — blob，tree 以及 tree — 都各自以文件的方式保存在 `.git/objects` 目录下。以下所列是目前为止样例中的所有对象，每个对象后面的注释里标明了它们保存的内容：

```
$ find .git/objects -type f
    .git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
    .git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
    .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
    .git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
    .git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
    .git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
    .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
    .git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
    .git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
    .git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

如果你按照以上描述进行了操作，可以得到如图 9-3 所示的对象图。

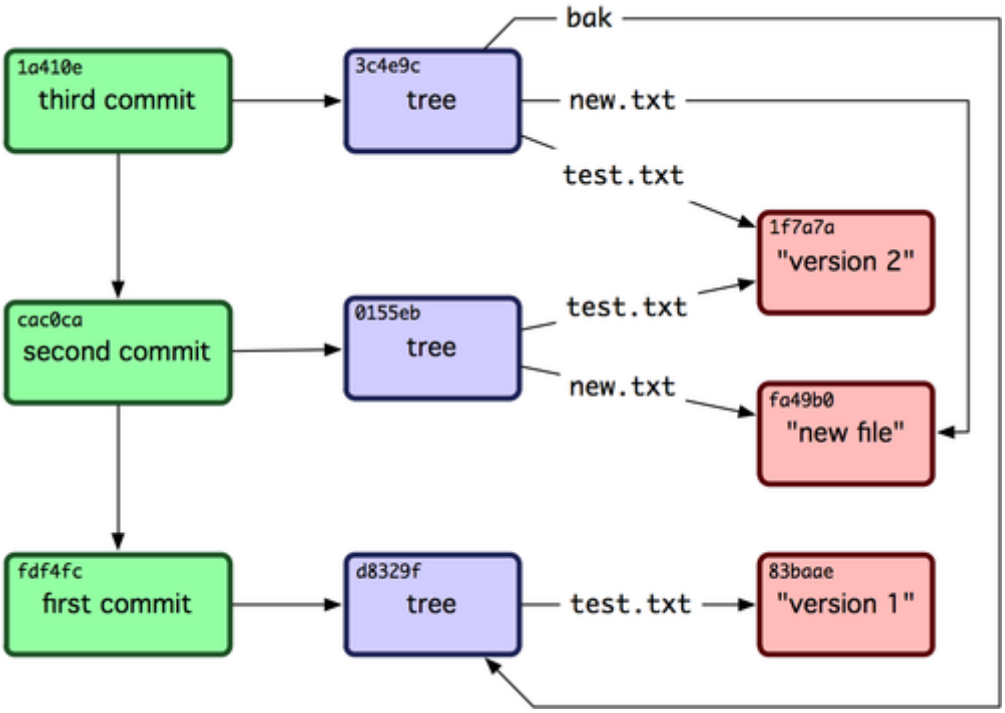

图 9-3. Git 目录下的所有对象

对象存储

之前我提到当存储数据内容时，同时会有一个文件头被存储起来。我们花些时间来看看 Git 是如何存储对象的。你将看来如何通过 Ruby 脚本语言存储一个 blob 对象 (这里以字符串 "what is up, doc?" 为例) 。使用 `irb` 命令进入 Ruby 交互式模式：

```
$ irb
    >> content = "what is up, doc?"
    => "what is up, doc?"
```

Git 以对象类型为起始内容构造一个文件头，本例中是一个 blob。然后添加一个空格，接着是数据内容的长度，最后是一个空字节 (null byte)：

```
>> header = "blob #{content.length}\0"
    => "blob 16\000"
```

Git 将文件头与原始数据内容拼接起来，并计算拼接后的新内容的 SHA-1 校验和。可以在 Ruby 中使用 `require` 语句导入 SHA1 digest 库，然后调用 `Digest::SHA1.hexdigest()` 方法计算字符串的 SHA-1 值：

```
>> store = header + content
    => "blob 16\000what is up, doc?"
    >> require 'digest/sha1'
    => true
    >> sha1 = Digest::SHA1.hexdigest(store)
    => "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git 用 zlib 对数据内容进行压缩，在 Ruby 中可以用 zlib 库来实现。首先需要导入该库，然后用 `Zlib::Deflate.deflate()` 对数据进行压缩：

```
>> require 'zlib'
    => true
    >> zlib_content = Zlib::Deflate.deflate(store)
    => "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

最后将用 zlib 压缩后的内容写入磁盘。需要指定保存对象的路径 (SHA-1 值的头两个字符作为子目录名称，剩余 38 个字符作为文件名保存至该子目录中)。在 Ruby 中，如果子目录不存在可以用 `FileUtils.mkdir_p()` 函数创建它。接着用 `File.open` 方法打开文件，并用 `write()` 方法将之前压缩的内容写入该文件：

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
    => ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
    >> require 'fileutils'
    => true
    >> FileUtils.mkdir_p(File.dirname(path))
    => ".git/objects/bd"
    >> File.open(path, 'w') { |f| f.write zlib_content }
    => 32
```

这就行了 —— 你已经创建了一个正确的 blob 对象。所有的 Git 对象都以这种方式存储，惟一的区别是类型不同 —— 除了字符串 blob，文件头起始内容还可以是 commit 或 tree 。不过虽然 blob 几乎可以是任意内容，commit 和 tree 的数据却是有固定格式的。

9.3 Git References

你可以执行像 `git log 1a410e` 这样的命令来查看完整的历史，但是这样你就要记得 `1a410e` 是你最后一次提交，这样才能在提交历史中找到这些对象。你需要一个文件来用一个简单的名字来记录这些 SHA-1 值，这样你就可以用这些指针而不是原来的 SHA-1 值去检索了。

在 Git 中，我们称之为“引用”（references 或者 refs，译者注）。你可以在 `.git/refs` 目录下面找到这些包含 SHA-1 值的文件。在这个项目里，这个目录还没不包含任何文件，但是包含这样一个简单的结构：

```
$ find .git/refs
    .git/refs
    .git/refs/heads
    .git/refs/tags
```

```
$ find .git/refs -type f
$
```

如果想要创建一个新的引用帮助你记住最后一次提交，技术上你可以这样做：

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

现在，你就可以在 Git 命令中使用你刚才创建的引用而不是 SHA-1 值：

```
$ git log --pretty=oneline master
    1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
    cac0cab538b970a37ea1e769cbbde608743bc96d second commit
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

当然，我们并不鼓励你直接修改这些引用文件。如果你确实需要更新一个引用，Git 提供了一个安全的命令 `update-ref`：

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

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。你可以用这条命令创建一个指向第二次提交的分支：

```
$ git update-ref refs/heads/test cac0ca
```

这样你的分支将会只包含那次提交以及之前的工作：

```
$ git log --pretty=oneline test
    cac0cab538b970a37ea1e769cbbde608743bc96d second commit
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

现在，你的 Git 数据库应该看起来像图 9-4 一样。

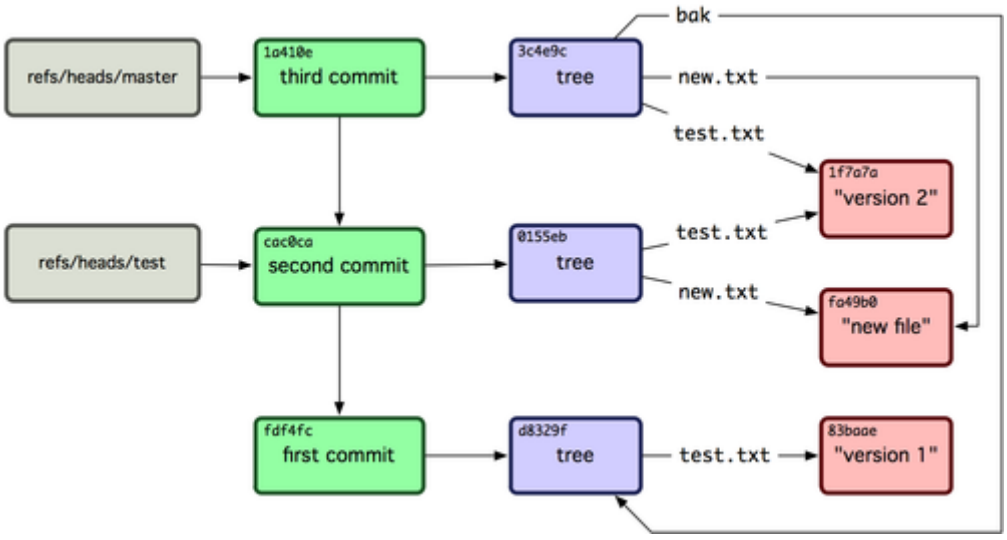

图 9-4. 包含分支引用的 Git 目录对象

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 标记

现在的问题是，当你执行 `git branch（分支名称）`这条命令的时候，Git 怎么知道最后一次提交的 SHA-1 值呢？答案就是 HEAD 文件。HEAD 文件是一个指向你当前所在分支的引用标识符。这样的引用标识符——它看起来并不像一个普通的引用——其实并不包含 SHA-1 值，而是一个指向另外一个引用的指针。如果你看一下这个文件，通常你将会看到这样的内容：

```
$ cat .git/HEAD
    ref: refs/heads/master
```

如果你执行 `git checkout test`，Git 就会更新这个文件，看起来像这样：

```
$ cat .git/HEAD
    ref: refs/heads/test
```

当你再执行 `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。

你也可以手动编辑这个文件，但是同样有一个更安全的方法可以这样做：`symbolic-ref`。你可以用下面这条命令读取 HEAD 的值：

```
$ git symbolic-ref HEAD
    refs/heads/master
```

你也可以设置 HEAD 的值：

```
$ git symbolic-ref HEAD refs/heads/test
    $ cat .git/HEAD
    ref: refs/heads/test
```

但是你不能设置成 refs 以外的形式：

```
$ git symbolic-ref HEAD test
    fatal: Refusing to point HEAD outside of refs/
```

Tags

你刚刚已经重温过了 Git 的三个主要对象类型，现在这是第四种。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。

正如我们在第二章所讨论的，Tag 有两种类型：annotated 和 lightweight 。你可以类似下面这样的命令建立一个 lightweight tag：

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

这就是 lightweight tag 的全部 —— 一个永远不会发生变化的分支。 annotated tag 要更复杂一点。如果你创建一个 annotated tag，Git 会创建一个 tag 对象，然后写入一个指向指向它而不是

直接指向 commit 的 reference。你可以这样创建一个 annotated tag（`-a` 参数表明这是一个 annotated tag）：

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

这是所创建对象的 SHA-1 值：

```
$ cat .git/refs/tags/v1.1
    9585191f37f7b0fb9444f35a9bf50de191beadc2
```

现在你可以运行 `cat-file` 命令检查这个 SHA-1 值：

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
    object 1a410efbd13591db07496601ebc7a059dd55cfe9
    type commit
    tag v1.1
    tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

    test tag
```

值得注意的是这个对象指向你所标记的 commit 对象的 SHA-1 值。同时需要注意的是它并不是必须要指向一个 commit 对象；你可以标记任何 Git 对象。例如，在 Git 的源代码里，管理者添加了一个 GPG 公钥（这是一个 blob 对象）对它做了一个标签。你就可以运行：

```
$ git cat-file blob junio-gpg-pub
```

来查看 Git 源代码仓库中的公钥. Linux kernel 也有一个不是指向 commit 对象的 tag —— 第一个 tag 是在导入源代码的时候创建的，它指向初始 tree（initial tree，译者注）。

Remotes

你将会看到的第四种 reference 是 remote reference（远程引用，译者注）。如果你添加了一个 remote 然后推送代码过去，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 `refs/remotes` 目录下。例如，你可以添加一个叫做 `origin` 的 remote 然后把你的 `master` 分支推送上去：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
    $ git push origin master
    Counting objects: 11, done.
    Compressing objects: 100% (5/5), done.
    Writing objects: 100% (7/7), 716 bytes, done.
    Total 7 (delta 2), reused 4 (delta 1)
    To git@github.com:schacon/simplegit-progit.git
    a11bef0..ca82a6d master -> master
```

然后查看 `refs/remotes/origin/master` 这个文件，你就会发现 `origin` remote 中的 `master` 分支就是你最后一次和服务器的通信。

```
$ cat .git/refs/remotes/origin/master
    ca82a6dff817ec66f44342007202690a93763949
```

Remote 应用和分支主要区别在于他们是不能被 check out 的。Git 把他们当作是标记这些了这些分支在服务器上最后状态的一种书签。

9.4 Packfiles

我们再来看一下 test Git 仓库。目前为止，有 11 个对象 —— 4 个 blob，3 个 tree，3 个 commit 以及一个 tag：

```
$ find .git/objects -type f
    .git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
    .git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
    .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
    .git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
    .git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
    .git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
    .git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
    .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
    .git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
    .git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
    .git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git 用 zlib 压缩文件内容，因此这些文件并没有占用太多空间，所有文件加起来总共仅用了 925 字节。接下去你会添加一些大文件以演示 Git 的一个很有意思的功能。将你之前用到过的 Grit 库中的 repo.rb 文件加进去 —— 这个源代码文件大小约为 12K：

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
    $ git add repo.rb
    $ git commit -m 'added repo.rb'
    [master 484a592] added repo.rb
    3 files changed, 459 insertions(+), 2 deletions(-)
    delete mode 100644 bak/test.txt
    create mode 100644 repo.rb
    rewrite test.txt (100%)
```

如果查看一下生成的 tree，可以看到 repo.rb 文件的 blob 对象的 SHA-1 值：

```
$ git cat-file -p master^{tree}
    100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
    100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e repo.rb
    100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

然后可以用 `git cat-file` 命令查看这个对象有多大：

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
    12898
```

稍微修改一下些文件，看会发生些什么：

```
$ echo '# testing' >> repo.rb
    $ git commit -am 'modified repo a bit'
    [master ab1afef] modified repo a bit
    1 files changed, 1 insertions(+), 0 deletions(-)
```

查看这个 commit 生成的 tree，可以看到一些有趣的东西：

```
$ git cat-file -p master^{tree}
    100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
    100644 blob 05408d195263d853f09dca71d55116663690c27c repo.rb
    100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

blob 对象与之前的已经不同了。这说明虽然只是往一个 400 行的文件最后加入了一行内容，Git 却用一个全新的对象来保存新的文件内容：

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
    12908
```

你的磁盘上有了两个几乎完全相同的 12K 的对象。如果 Git 只完整保存其中一个，并保存另一个对象的差异内容，岂不更好？

事实上 Git 可以那样做。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。手工调用 `git gc` 命令让 Git 将库中对象打包并看会发生些什么：

```
$ git gc
    Counting objects: 17, done.
    Delta compression using 2 threads.
    Compressing objects: 100% (13/13), done.
    Writing objects: 100% (17/17), done.
    Total 17 (delta 1), reused 10 (delta 0)
```

查看一下 objects 目录，会发现大部分对象都不在了，与此同时出现了两个新文件：

```
$ find .git/objects -type f
    .git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
    .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
    .git/objects/info/packs
    .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
    .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

仍保留着的几个对象是未被任何 commit 引用的 blob —— 在此例中是你之前创建的 "what is up, doc?" 和 "test content" 这两个示例 blob。你从没将他们添加至任何 commit，所以 Git 认为它们是 "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。有意思的是运行 `gc` 命令前磁盘上的对象大小约为 12K ，而这个新生成的 packfile 仅为 6K 大小。通过打包对象减少了一半磁盘使用空间。

Git 是如何做到这点的？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。可以查看一下 packfile ，观察它是如何节省空间的。`git verify-pack` 命令用于显示已打包的内容：

```
$ git verify-pack -v \
    .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
    0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 5400
    05408d195263d853f09dca71d55116663690c27c blob 12908 3478 874
    09f01cea547666f58d6a8d809583841a7c6f0130 tree 106 107 5086
    1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
    1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 5381
```

```
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag 136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob 7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree 106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

如果你还记得的话, `9bc1d` 这个 blob 是 repo.rb 文件的第一个版本，这个 blob 引用了 `05408` 这个 blob，即该文件的第二个版本。命令输出内容的第三列显示的是对象大小，可以看到 `05408` 占用了 12K 空间，而 `9bc1d` 仅为 7 字节。非常有趣的是第二个版本才是完整保存文件内容的对象，而第一个版本是以差异方式保存的 —— 这是因为大部分情况下需要快速访问文件的最新版本。

最妙的是可以随时进行重新打包。Git 自动定期对仓库进行重新打包以节省空间。当然也可以手工运行 `git gc` 命令来这么做。

9.5 The Refspec

这本书读到这里，你已经使用过一些简单的远程分支到本地引用的映射方式了，这种映射可以更为复杂。 假设你像这样添加了一项远程仓库：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

它在你的 `.git/config` 文件中添加了一节，指定了远程的名称 (`origin`), 远程仓库的URL地址，和用于获取操作的 Refspec:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec 的格式是一个可选的 `+` 号，接着是 `<src>:<dst>` 的格式，这里 `<src>` 是远端上的引用格式，`<dst>` 是将要记录在本地的引用格式。可选的 `+` 号告诉 Git 在即使不能快速演进的情况下，也去强制更新它。

缺省情况下 refspec 会被 `git remote add` 命令所自动生成， Git 会获取远端上 `refs/heads/` 下面的所有引用，并将它写入到本地的 `refs/remotes/origin/`. 所以，如果远端上有一个 `master` 分支，你在本地可以通过下面这种方式来访问它的历史记录：

```
$ git log origin/master
    $ git log remotes/origin/master
    $ git log refs/remotes/origin/master
```

它们全是等价的，因为 Git 把它们都扩展成 `refs/remotes/origin/master`.

如果你想让 Git 每次只拉取远程的 `master` 分支，而不是远程的所有分支，你可以把 fetch 这一行修改成这样：

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

这是 `git fetch` 操作对这个远端的缺省 refspec 值。而如果你只想做一次该操作，也可以在命令行上指定这个 refspec. 如可以这样拉取远程的 `master` 分支到本地的 `origin/mymaster` 分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

你也可以在命令行上指定多个 refspec. 像这样可以一次获取远程的多个分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster \
    topic:refs/remotes/origin/topic
    From git@github.com:schacon/simplegit
    ! [rejected] master -> origin/mymaster (non fast forward)
    * [new branch] topic -> origin/topic
```

在这个例子中， `master` 分支因为不是一个可以快速演进的引用而拉取操作被拒绝。你可以在 refspec 之前使用一个 `+` 号来重载这种行为。

你也可以在配置文件中指定多个 refspec. 如你想在每次获取时都获取 `master` 和 `experiment` 分支，就添加两行：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

但是这里不能使用部分通配符，像这样就是不合法的：

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

但无论如何，你可以使用命名空间来达到这个目的。如你有一个QA组，他们推送一系列分支，你想每次获取 `master` 分支和QA组的所有分支，你可以使用这样的配置段落：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

如果你的工作流很复杂，有QA组推送的分支、开发人员推送的分支、和集成人员推送的分支，并且他们在远程分支上协作，你可以采用这种方式为他们创建各自的命名空间。

推送 Refspec

采用命名空间的方式确实很棒，但QA组成员第1次是如何将他们的分支推送到 `qa/` 空间里面的呢？答案是你可以使用 refspec 来推送。

如果QA组成员想把他们的 `master` 分支推送到远程的 `qa/master` 分支上，可以这样运行：

```
$ git push origin master:refs/heads/qa/master
```

如果他们想让 Git 每次运行 `git push origin` 时都这样自动推送，他们可以在配置文件中添加 `push` 值：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

这样，就会让 `git push origin` 缺省就把本地的 `master` 分支推送到远程的 `qa/master` 分支上。

删除引用

你也可以使用 refspec 来删除远程的引用，是通过运行这样的命令：

```
$ git push origin :topic
```

因为 refspec 的格式是 `<src>:<dst>`, 通过把 `<src>` 部分留空的方式，这个意思是是把远程的 `topic` 分支变成空，也就是删除它。

9.6 传输协议

Git 可以以两种主要的方式跨越两个仓库传输数据：基于HTTP协议之上，和 `file://`, `ssh://`, 和 `git://` 等智能传输协议。这一节带你快速浏览这两种主要的协议操作过程。

哑协议

Git 基于HTTP之上传输通常被称为哑协议，这是因为它在服务端不需要有针对 Git 特有的代码。这个获取过程仅仅是一系列GET请求，客户端可以假定服务端的Git仓库中的布局。让我们以 simplegit 库来看看 `http-fetch` 的过程：

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

它做的第1件事情就是获取 `info/refs` 文件。这个文件是在服务端运行了 `update-server-info` 所生成的，这也解释了为什么在服务端要想使用HTTP传输，必须要开启 `post-receive` 钩子：

```
=> GET info/refs
    ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

现在你有一个远端引用和SHA值的列表。下一步是寻找HEAD引用，这样你就知道了在完成后，什么应该被检出到工作目录：

```
=> GET HEAD
    ref: refs/heads/master
```

这说明在完成获取后，需要检出 `master` 分支。 这时，已经可以开始漫游操作了。因为你的起点是在 `info/refs` 文件中所提到的 `ca82a6` commit 对象，你的开始操作就是获取它：

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
    (179 bytes of binary data)
```

然后你取回了这个对象 - 这在服务端是一个松散格式的对象，你使用的是静态的 HTTP GET 请求获取的。可以使用 zlib 解压缩它，去除其头部，查看它的 commmit 内容：

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
    tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
    parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    author Scott Chacon <schacon@gmail.com> 1205815931 -0700
    committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

    changed the version number
```

这样，就得到了两个需要进一步获取的对象 - `cfda3b` 是这个 commit 对象所对应的 tree 对象，和 `085bb3` 是它的父对象；

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    (179 bytes of data)
```

这样就取得了这它的下一步 commit 对象，再抓取 tree 对象：

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
    (404 - Not Found)
```

Oops - 看起来这个 tree 对象在服务端并不以松散格式对象存在，所以得到了404响应，代表在HTTP服务端没有找到该对象。这有好几个原因 - 这个对象可能在替代仓库里面，或者在打包文件里面， Git 会首先检查任何列出的替代仓库：

```
=> GET objects/info/http-alternates
    (empty file)
```

如果这返回了几个替代仓库列表，那么它会去那些地方检查松散格式对象和文件 - 这是一种在软件分叉之间共享对象以节省磁盘的好方法。然而，在这个例子中，没有替代仓库。所以你所需要的对象肯定在某个打包文件中。要检查服务端有哪些打包格式文件，你需要获取 `objects/info/packs` 文件，这里面包含有打包文件列表（是的，它也是被 `update-server-info` 所生成的）；

```
=> GET objects/info/packs
    P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

这里服务端只有一个打包文件，所以你要的对象显然就在里面。但是你可以先检查它的索引文件以确认。这在服务端有多个打包文件时也很有用，因为这样就可以先检查你所需要的对象空间是在哪一个打包文件里面了：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
    (4k of binary data)
```

现在你有了这个打包文件的索引，你可以看看你要的对象是否在里面 - 因为索引文件列出了这个打包文件所包含的所有对象的SHA值，和该对象存在于打包文件中的偏移量，所以你只需要简单地获取整个打包文件：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
    (13k of binary data)
```

现在你也有了这个 tree 对象，你可以继续在 commit 对象上漫游。它们全部都在这个你已经下载到的打包文件里面，所以你不用继续向服务端请求更多下载了。 在这完成之后，由于下载开始时已探明HEAD引用是指向 `master` 分支， Git 会将它检出到工作目录。

整个过程看起来就像这样：

```
$ git clone http://github.com/schacon/simplegit-progit.git
    Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
    got ca82a6dff817ec66f44342007202690a93763949
    walk ca82a6dff817ec66f44342007202690a93763949
    got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    Getting alternates list for http://github.com/schacon/simplegit-progit.git
    Getting pack list for http://github.com/schacon/simplegit-progit.git
    Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
    Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
    which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
    walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

智能协议

这个HTTP方法是很简单但效率不是很高。使用智能协议是传送数据的更常用的方法。这些协议在远端都有Git智能型进程在服务 - 它可以读出本地数据并计算出客户端所需要的，并生成合适的数据给它，这有两类传输数据的进程：一对用于上传数据和一对用于下载。

上传数据

为了上传数据至远端， Git 使用 `send-pack` 和 `receive-pack` 进程。这个 `send-pack` 进程运行在客户端上，它连接至远端运行的 `receive-pack` 进程。

举例来说，你在你的项目上运行了 `git push origin master`, 并且 `origin` 被定义为一个使用SSH协议的URL。 Git 会使用 `send-pack` 进程，它会启动一个基于SSH的连接到服务器。它尝试像这样透过SSH在服务端运行命令：

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
    005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
    003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
    0000
```

这里的 `git-receive-pack` 命令会立即对它所拥有的每一个引用响应一行 - 在这个例子中，只有 `master` 分支和它的SHA值。这里第1行也包含了服务端的能力列表（这里是 `report-status` 和 `delete-refs`）。

每一行以4字节的十六进制开始，用于指定整行的长度。你看到第1行以005b开始，这在十六进制中表示91，意味着第1行有91字节长。下一行以003e起始，表示有62字节长，所以需要读剩下的62字节。再下一行是0000开始，表示服务器已完成了引用列表过程。

现在它知道了服务端的状态，你的 `send-pack` 进程会判断哪些 commit 是它所拥有但服务端没有的。针对每个引用，这次推送都会告诉对端的 `receive-pack` 这个信息。举例说，如果你在更新 `master` 分支，并且增加 `experiment` 分支，这个 `send-pack` 将会是像这样：

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
    refs/heads/master report-status
    006700000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
```

```
refs/heads/experiment
    0000
```

这里的全'0'的SHA-1值表示之前没有过这个对象 - 因为你是在添加新的 experiment 引用。如果你在删除一个引用，你会看到相反的： 就是右边是全'0'。

Git 针对每个引用发送这样一行信息，就是旧的SHA值，新的SHA值，和将要更新的引用的名称。第1行还会包含有客户端的能力。下一步，客户端会发送一个所有那些服务端所没有的对象的一个打包文件。最后，服务端以成功(或者失败)来响应：

```
000Aunpack ok
```

下载数据

当你在下载数据时， `fetch-pack` 和 `upload-pack` 进程就起作用了。客户端启动 `fetch-pack` 进程，连接至远端的 `upload-pack` 进程，以协商后续数据传输过程。

在远端仓库有不同的方式启动 `upload-pack` 进程。你可以使用与 `receive-pack` 相同的透过SSH管道的方式，也可以通过 Git 后台来启动这个进程，它默认监听在9418号端口上。这里 `fetch-pack` 进程在连接后像这样向后台发送数据：

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

它也是以4字节指定后续字节长度的方式开始，然后是要运行的命令，和一个空字节，然后是服务端的主机名，再跟随一个最后的空字节。 Git 后台进程会检查这个命令是否可以运行，以及那个仓库是否存在，以及是否具有公开权限。如果所有检查都通过了，它会启动这个 `upload-pack` 进程并将客户端的请求移交给它。

如果你透过SSH使用获取功能， `fetch-pack` 会像这样运行：

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

不管哪种方式，在 `fetch-pack` 连接之后， `upload-pack` 都会以这种形式返回：

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag
    003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
    003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
    0000
```

这与 `receive-pack` 响应很类似，但是这里指的能力是不同的。而且它还会指出HEAD引用，让客户端可以检查是否是一份克隆。

在这里， `fetch-pack` 进程检查它自己所拥有的对象和所有它需要的对象，通过发送 "want" 和所需对象的SHA值，发送 "have" 和所有它已拥有的对象的SHA值。在列表完成时，再发送 "done" 通知 `upload-pack` 进程开始发送所需对象的打包文件。这个过程看起来像这样：

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
    0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    0000
    0009done
```

这是传输协议的一个很基础的例子，在更复杂的例子中，客户端可能会支持 `multi_ack` 或者 `side-band` 能力；但是这个例子中展示了智能协议的基本交互过程。

9.7 维护及数据恢复

你时不时的需要进行一些清理工作 —— 如减小一个仓库的大小，清理导入的库，或是恢复丢失的数据。本节将描述这类使用场景。

维护

Git 会不定时地自动运行称为 "auto gc" 的命令。大部分情况下该命令什么都不处理。不过要是存在太多松散对象 (loose object, 不在 packfile 中的对象) 或 packfile，Git 会进行调用 `git gc` 命令。`gc` 指垃圾收集 (garbage collect)，此命令会做很多工作：收集所有松散对象并将它们存入 packfile，合并这些 packfile 进一个大的 packfile，然后将不被任何 commit 引用并且已存在一段时间 (数月) 的对象删除。

可以手工运行 auto gc 命令：

```
$ git gc --auto
```

再次强调，这个命令一般什么都不干。如果有 7,000 个左右的松散对象或是 50 个以上的 packfile，Git 才会真正调用 gc 命令。可能通过修改配置中的 `gc.auto` 和 `gc.autopacklimit` 来调整这两个阈值。

`gc` 还会将所有引用 (references) 并入一个单独文件。假设仓库中包含以下分支和标签：

```
$ find .git/refs -type f
    .git/refs/heads/experiment
    .git/refs/heads/master
    .git/refs/tags/v1.0
    .git/refs/tags/v1.1
```

这时如果运行 `git gc`, `refs` 下的所有文件都会消失。Git 会将这些文件挪到 `.git/packed-refs` 文件中去以提高效率，该文件是这个样子的：

```
$ cat .git/packed-refs
    # pack-refs with: peeled
    cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
    ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
    cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
    9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
    ^1a410efbd13591db07496601ebc7a059dd55cfe9
```

当更新一个引用时，Git 不会修改这个文件，而是在 `refs/heads` 下写入一个新文件。当查找一个引用的 SHA 时，Git 首先在 `refs` 目录下查找，如果未找到则到 `packed-refs` 文件中去查找。因此如果在 `refs` 目录下找不到一个引用，该引用可能存到 `packed-refs` 文件中去了。

请留意文件最后以 `^` 开头的那一行。这表示该行上一行的那个标签是一个 annotated 标签，而该行正是那个标签所指向的 commit 。

数据恢复

在使用 Git 的过程中，有时会不小心丢失 commit 信息。这一般出现在以下情况下：强制删除了一个分支而后又想重新使用这个分支，hard-reset 了一个分支从而丢弃了分支的部分 commit。如果这真的发生了，有什么办法把丢失的 commit 找回来呢？

下面的示例演示了对 test 仓库主分支进行 hard-reset 到一个老版本的 commit 的操作，然后恢复丢失的 commit 。首先查看一下当前的仓库状态：

```
$ git log --pretty=oneline
    ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
    484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
    1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
    cac0cab538b970a37ea1e769cbbde608743bc96d second commit
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

接着将 `master` 分支移回至中间的一个 commit：

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
    HEAD is now at 1a410ef third commit
    $ git log --pretty=oneline
    1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
    cac0cab538b970a37ea1e769cbbde608743bc96d second commit
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

这样就丢弃了最新的两个 commit —— 包含这两个 commit 的分支不存在了。现在要做的是找出最新的那个 commit 的 SHA，然后添加一个指它它的分支。关键在于找出最新的 commit 的 SHA —— 你不大可能记住了这个 SHA，是吧？

通常最快捷的办法是使用 `git reflog` 工具。当你 (在一个仓库下) 工作时，Git 会在你每次修改了 HEAD 时悄悄地将改动记录下来。当你提交或修改分支时，reflog 就会更新。`git update-ref` 命令也可以更新 reflog，这是在本章前面的 "Git References" 部分我们使用该命令而不是手工将 SHA 值写入 ref 文件的理由。任何时间运行 `git reflog` 命令可以查看当前的状态：

```
$ git reflog
    1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
    ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

可以看到我们签出的两个 commit ，但没有更多的相关信息。运行 `git log -g` 会输出 reflog 的正常日志，从而显示更多有用信息：

```
$ git log -g
    commit 1a410efbd13591db07496601ebc7a059dd55cfe9
    Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
    Reflog message: updating HEAD
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri May 22 18:22:37 2009 -0700

    third commit

    commit ab1afef80fac8e34258ff41fc1b867c702daa24b
    Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
    Reflog message: updating HEAD
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri May 22 18:15:24 2009 -0700

    modified repo a bit
```

看起来弄丢了的 commit 是底下那个，这样在那个 commit 上创建一个新分支就能把它恢复过来。比方说，可以在那个 commit (ab1afef) 上创建一个名为 `recover-branch` 的分支：

```
$ git branch recover-branch ab1afef
    $ git log --pretty=oneline recover-branch
    ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
    484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
    1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
    cac0cab538b970a37ea1e769cbbde608743bc96d second commit
    fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

酷！这样有了一个跟原来 `master` 一样的 `recover-branch` 分支，最新的两个 commit 又找回来了。接着，假设引起 commit 丢失的原因并没有记录在 reflog 中 —— 可以通过删除 `recover-branch` 和 reflog 来模拟这种情况。这样最新的两个 commit 不会被任何东西引用到：

```
$ git branch -D recover-branch
    $ rm -Rf .git/logs/
```

因为 reflog 数据是保存在 `.git/logs/` 目录下的，这样就没有 reflog 了。现在要怎样恢复 commit 呢？办法之一是使用 `git fsck` 工具，该工具会检查仓库的数据完整性。如果指定 `--ful` 选项，该命令显示所有未被其他对象引用 (指向) 的所有对象：

```
$ git fsck --full
    dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
    dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
    dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
    dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

本例中，可以从 dangling commit 找到丢失了的 commit。用相同的方法就可以恢复它，即创建一个指向该 SHA 的分支。

移除对象

Git 有许多过人之处，不过有一个功能有时却会带来问题：`git clone` 会将包含每一个文件的所有历史版本的整个项目下载下来。如果项目包含的仅仅是源代码的话这并没有什么坏处，毕竟 Git 可以非常高效地压缩此类数据。不过如果有人在某个时刻往项目中添加了一个非常大的文件，那们即便他在后来的提交中将此文件删掉了，所有的签出都会下载这个大文件。因为历史记录中引用了这个文件，它会一直存在着。

当你将 Subversion 或 Perforce 仓库转换导入至 Git 时这会成为一个很严重的问题。在此类系统中，(签出时) 不会下载整个仓库历史，所以这种情形不大会有不良后果。如果你从其他系统导入了一个仓库，或是发觉一个仓库的尺寸远超出预计，可以用下面的方法找到并移除大 (尺寸) 对象。

警告：此方法会破坏提交历史。为了移除对一个大文件的引用，从最早包含该引用的 tree 对象开始之后的所有 commit 对象都会被重写。如果在刚导入一个仓库并在其他人在此基础上开始工作之前这么做，那没有什么问题 —— 否则你不得不通知所有协作者 (贡献者) 去衍合你新修改的 commit 。

为了演示这点，往 test 仓库中加入一个大文件，然后在下次提交时将它删除，接着找到并将这个文件从仓库中永久删除。首先，加一个大文件进去：

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
    $ git add git.tbz2
    $ git commit -am 'added git tarball'
    [master 6df7640] added git tarball
```

```
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

喔，你并不想往项目中加进一个这么大的 tar 包。最后还是去掉它：

```
$ git rm git.tbz2
    rm 'git.tbz2'
    $ git commit -m 'oops - removed large tarball'
    [master da3f30d] oops - removed large tarball
    1 files changed, 0 insertions(+), 0 deletions(-)
    delete mode 100644 git.tbz2
```

对仓库进行 `gc` 操作，并查看占用了空间：

```
$ git gc
    Counting objects: 21, done.
    Delta compression using 2 threads.
    Compressing objects: 100% (16/16), done.
    Writing objects: 100% (21/21), done.
    Total 21 (delta 3), reused 15 (delta 1)
```

可以运行 `count-objects` 以查看使用了多少空间：

```
$ git count-objects -v
    count: 4
    size: 16
    in-pack: 21
    packs: 1
    size-pack: 2016
    prune-packable: 0
    garbage: 0
```

`size-pack` 是以千字节为单位表示的 packfiles 的大小，因此已经使用了 2MB 。而在这次提交之前仅用了 2K 左右 —— 显然在这次提交时删除文件并没有真正将其从历史记录中删除。每当有人复制这个仓库去取得这个小项目时，都不得不复制所有 2MB 数据，而这仅仅因为你曾经不小心加了个大文件。当我们来解决这个问题。

首先要找出这个文件。在本例中，你知道是哪个文件。假设你并不知道这一点，要如何找出哪个 (些) 文件占用了这么多的空间？如果运行 `git gc`，所有对象会存入一个 packfile 文件；运行另一个底层命令 `git verify-pack` 以识别出大对象，对输出的第三列信息即文件大小进行排序，还可以将输出定向到 `tail` 命令，因为你只关心排在最后的那几个最大的文件：

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
    e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4667
    05408d195263d853f09dca71d55116663690c27c blob 12908 3478 1189
    7a9eb2fba2b1811321254ac360970fc169ba2330 blob 2056716 2056872 5401
```

最底下那个就是那个大文件：2MB 。要查看这到底是哪个文件，可以使用第 7 章中已经简单使用过的 `rev-list` 命令。若给 `rev-list` 命令传入 `--objects` 选项，它会列出所有 commit SHA 值，blob SHA 值及相应的文件路径。可以这样查看 blob 的文件名：

```
$ git rev-list --objects --all | grep 7a9eb2fb
    7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

接下来要将该文件从历史记录的所有 tree 中移除。很容易找出哪些 commit 修改了这个文件：

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

必须重写从 `6df76` 开始的所有 commit 才能将文件从 Git 历史中完全移除。这么做需要用到第 6 章中用过的 `filter-branch` 命令：

```
$ git filter-branch --index-filter \
'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

`--index-filter` 选项类似于第 6 章中使用的 `--tree-filter` 选项，但这里不是传入一个命令去修改磁盘上签出的文件，而是修改暂存区域或索引。不能用 `rm file` 命令来删除一个特定文件，而是必须用 `git rm --cached` 来删除它 —— 即从索引而不是磁盘删除它。这样做是出于速度考虑 —— 由于 Git 在运行你的 filter 之前无需将所有版本签出到磁盘上，这个操作会快得多。也可以用 `--tree-filter` 来完成相同的操作。`git rm` 的 `--ignore-unmatch` 选项指定当你试图删除的内容并不存在时不显示错误。最后，因为你清楚问题是从哪个 commit 开始的，使用 `filter-branch` 重写自 `6df7640` 这个 commit 开始的所有历史记录。不这么做的话会重写所有历史记录，花费不必要的更多时间。

现在历史记录中已经不包含对那个文件的引用了。不过 reflog 以及运行 `filter-branch` 时 Git 往 `.git/refs/original` 添加的一些 refs 中仍有对它的引用，因此需要将这些引用删除并对仓库进行 repack 操作。在进行 repack 前需要将所有对这些提交的引用去除：

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

看一下节省了多少空间。

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

repack 后仓库的大小减小到了 7K ，远小于之前的 2MB 。从 size 值可以看出大文件对象还在松散对象中，其实并没有消失，不过这没有关系，重要的是在再进行推送或复制，这个对象不会再传送出去。如果真的要完全把这个对象删除，可以运行 `git prune --expire` 命令。

9.8 总结

现在你应该对 Git 可以作什么相当了解了，并且在一定程度上也知道了 Git 是如何实现的。本章覆盖了许多 plumbing 命令 —— 这些命令比较底层，且比你在本书其他部分学到的 porcelain 命令要

来得简单。从底层了解 Git 的工作原理可以帮助你更好地理解为何 Git 实现了目前的这些功能，也使你能够针对你的工作流写出自己的工具和脚本。

Git 作为一套 content-addressable 的文件系统，是一个非常强大的工具，而不仅仅只是一个 VCS 供人使用。希望借助于你新学到的 Git 内部原理的知识，你可以实现自己的有趣的应用，并以更高级便利的方式使用 Git。

上一节首页（目录）| 返回 码云