Git 自下而上方法 [译]

zhabgyuan1993

July 28, 2013

Contents

译者序

这本书讲解思路着实清晰。因此有了翻译本书的念头。被网友评选为10 大 Git 入门学习资料的本书不是只讲解较高层面的 Git 命令的书籍。这本书，短短 30 余页，确实是入门精华，拿这样的书作为自己初次翻译的尝试，或有不妥。译者水平有限，还望见谅，望大家指正。十分感谢诸位老师的悉心指导:

- Haix Zhang
- neo szm
- Ysyk Zheng

最后附上本书的英文原版链接，方便大家参阅。

作者序

当我追寻领悟 Git 之道时，自下向上的方法 —–而非单单关注它较高层面的命令使我受益匪浅。因为当从这个角度理解时，Git 是如此的简洁，我猜测其他人可能会对阅读我的发现有兴趣，也许能避免我在探索中经历过的一些困难。

我使用的是 Git 1.5.4.5，本书中所有示例都是基于此版本

License

引言

发发

欢迎来到 Git 世界，我希望这份文档可以帮助你理解这个强大的内容跟踪系统，同时揭示藏匿其中的简洁，而在外界看来这只是一系列让人发晕的命令选项。在我们深入之前，这里有一些需要首先提及的术语，因为他们将在本文中多次出现。

仓库 (repository)
仓库是提交 (commit) 的集合，每一个提交都是过去某个时间点工作目录状态的归档，无论它是你在主机上或其他人的主机。仓库还定义了 HEAD (参见下方)，用来标识当前工作目录源自的分支或提交。最后，它还包含了一系列的分支 (branch) 和标签 (tags)，用来识别特定的提交。

暂存区 (the index)
与其他你也许用过的类似的工具不同，Git 并非将工作目录中对文档的修改直接提交到仓库，而是将文档的修改首先记录在一种名为暂存区的东西中。可以将这种处理方式看作是在提交 (记录所有已确认的修改) 之前逐个地确认修改。一些人觉得称其为缓冲区域 (stage area) 而非索引文件 (index) 更有裨益。

工作目录 [3] (working tree) 工作目录是在你文件系统中任意一个拥有一个相关联的仓库的文件目录 (通常体现在工作目录中存在一个名为.git 的子文件目录)，包含工作目录里所有的文件和子目录。

提交 (commit)
提交是工作目录某个时刻的一次快照。当你完成一次提交，HEAD (参见下文) 原来指向的提交成为

了这次提交的父代提交，这就形成了“修订版本历史（revision histoty）”的概念。

分支（branch）
分支仅是某次提交的名称（稍后将详细阐述），或者称为索引（reference）。它的父代提交构成了分支的历史，因此有典型的提法“开发分支”（branch for devolopment）。

标签（tag）
标签，与分支类似，也可以作为某次提交的名称，但是它始终命名同一个提交，并且有自己的描述文字。

master
大多数仓库的开发主线都是在一个名为”master ”的分支上完成的。尽管它通常是默认的分支，但是与其它分支并无特别之处。

HEAD
HEAD 被仓库用来确定当前检出的对象：

- 如果检出的是某个分支，HEAD 就会符号链接至该分支，这意味着当执行下次提交时，分支名 (branch name)[4] 会被更新。

- 如果检出的是某个提交，HEAD 仅指向该提交。这样的 HEAD 被称为游离 HEAD (detached HEAD，例如，当你检出某个标签时，就会产生游离分支。

通常的操作流程是这样的：创建一个仓库后，你所有的工作都在工作目录中进行，当工作进行到某个重要阶段 —-修改了一个 Bug, 或结束了一天的工作，或所有代码都可以成功编译 —-你将这些修改写入到暂存区。当暂存区中包含所有你准备提交的内容后，你将它中的内容记录在仓库。

请记住这个总体上的描绘 [5]，因为接下来的章节将对上图中的每一个不同实体对于 Git 操作的重要性进行描述。

[3]: 此处翻译为工作目录, 避免与后文提及的树对象 (tree-object) 混淆。
[4]: 译者注：这里的分支名是指文件 .git/refs/heads/branchName , 它反映了一种从名称到提交的映射关系。当检出的对象为分支时，.git/HEAD 中的内容

为该文件，当检出某次提交时，.git/-HEAD 中的内容仅是某次提交的 id。

[5]: 实际上检出操作是从仓库中拷贝到到缓存区域再写入工作目录，而这一过程对于使用者是不可见的，故图示中将这一过程直接展示为从仓库到工作目录。

仓库：文件内容追踪

仓库：文件内容追

如前所述，Git 最为基本的功用在于它维护着一份工作目录内容的快照。Git 绝大多数的内部设计可以通过完成这个基本功用的角度理解。

Git 仓库的设计在许多方面都能看到 UNIX 系统的影子：一个基于根节点（ROOT）的文件系统，常由多个目录组成，其中多数目录拥有子目录或者存储了数据的文件。这些文件的元数据 (meta data) 既存在于目录中（它们的名称），也存在于索引节点 (index node/i-node) 中。索引节点包含了这些文件的诸如大小、类型、权限等信息。每个索引节点都

有其唯一的编码来标识其对应文件的内容。你可能通过多种不同的目录路径访问一个节点 (例如：hard link)，而只有这个索引节点在文件系统中真正拥有内容。

Git 在内部实现中同样有这样的结构，但有一两处重要的不同之处。其一：你的文件内容表示在 blob 中，blob 的角色相当与 Unix 文件系统中的叶节点，这种容纳叶节点的非常类似文件夹的东西被称为树 (tree)。就如文件系统中的索引节点通过一个由系统指定的独一无二的数区别，blob 的名字通过哈希函数 SHA-1 计算文件大小和内容得出6。实际上，如索引节点一样，他的名字只是一个随机的数字，但这样做带来的好处有两点额外的好处，其一确保文件的内容不会改变，其二，相同的内容总通过相同的 blob 表示，无论它所处的位置：跨提交，跨仓库，甚至跨整个互联网。如果有多个 tree (树) 引用同一个 blob，如同硬链接一样：只要仓库中拥有一个到某个 Blob 的链接，这个 Blob 就不会从该仓库中消失。

Git 中的 blob 与文件系统中文件的不同之处在于 blob 中不包含任何关于它存储内容的元信息。所有的元信息都存储在相关联的 tree 对象中。某个 tree 对象将一些数据内容视为一个创建于 2004 年的名

为 foo 的文件，而另一个树对象将相同的内容视为一个创建于 5 年后名为 bar 的文件。对于通常的文件系统而言，这两个拥有相同内容但具有不同元信息文件是相互独立的两个文件[14]。为何两者存在这种差异？因为文件系统需要支持文件的修改，而 Git 不必。Git 仓库中数据不可变的事实造成了与文件系统的差异，因此需要一个不同的设计。事实证明，而这样的设计允许更近凑的存储，因为所有具有相同内容的对象，不论所处的位置，都可以被共享。

```
touch foo bar # 创建两个空内容的文件
ls -i foo bar # 列出文件的节点信息, 此处应显示不同的索引号
diff for bar # 对比文件的内容，无区别
```

Blob 对象

既然我们有了大体上的描绘，现在来看一些实例。接下来我们将开始创建一个 Git 示例仓库，演示 Git 是如何自下而上工作的。阅读时也可以一起动手实践:

```
$ mkdir sample; cd sample
$ echo ' Hello world! ' > greeting
```

此处我创建了一个文件系统目录 sample , 里面包含一个 greeting 文件, 该文件的内容显而易见。尽管我还没有创建仓库，但可运行一些 Git 的命令来了解接下来是怎样实现的。首先，我想知道 Git 将 greeting 文件的内容存储于哪一个哈希标识符 (hash id) 之下

```
$ git hash-object greeting
af5626b4a114abcb82d63db7c8082c3c4756e51b
```

如果你在你的系统上运行上面的命令，我们将得到相同的哈希值。即便我们创建了两个不同的仓库来存储文件，在这两个仓库中存储 greeting 文件内容的 blob 对象也会拥有相同的哈希标识符。我甚至可以从你的仓库中 pull 某个提交，然后放入我的仓库中。

而 Git 将识别出我们追踪的是相同的文件内容 —-因此 Git 只会存储一个副本。这简直太酷了！下一步我们可以初始化一个仓库，把我们的文件 greeting 提交进去。我将一次完成这些步骤，过后再逐步操作一次这样你就可以了解它底层的实现了。

```
$ git init
$ git add greeting
$ git commit -m "Added my greeting"
```

此时应如我们期望的那样 blob 对象出现在仓库中，并且使用上文中已经确定的哈希标识符。方便起见，Git 要求只要位数足够在仓库中唯一确定一个对象的哈希标识符。通常 6-7 位足够了:

```
$ git cat-file -t af5626b
blob
$ git cat-file blob af5626b
Hello, world!
```

找到了！我甚至没有看哪个提交 (commit) 引用了它，或者是在哪个 tree 对象中，但单根据它的内容，

我便可以肯定它在那了，它将一直使用同一个标识符 (identifier)，不论仓库存在多长时间，文件存储在哪里，它的内容都是这样永久地存储下来。通过这种方式，Git 中的一个 blob 代表 Git 中基本的数据单元 (fundamental data unit)。实际上，整个 Git 系统就是关于 blob 的存储管理。

Blob 依附于 tree 对象

文件的内容存储于 blob，但是这些 blob 具有的属性很少，他们没有名字，没有结构 —-他们毕竟只是 “blob”[7]。为了让 Git 能够表示文件的结构，命名文件，Git 将 blob 作为叶节点依附到 tree 对象。仅查看 blob 本身的属性，无法知道一个 blob 依附于哪一个 tree 对象，因为这个 blob 可能有很多很多的拥有者。但我知道这个 blob 对象一定依附于与刚才完成的提交相关联的 tree 对象。

```
$ git ls-tree HEAD
100644 blob af5626b4a114abcb82d63db7c8082c3c47
```

找到了！第一次提交添加我的 greeting 文件至仓库中。这次提交包含一个 Git 的 tree 对象，该对象拥有一个叶节点：一个包含 greeting 文件内容的 blob 。虽然我可以通过传递参数 HEAD 给 ls-tree 命令 [15] 来查看包含 blob 的 tree 对象 (的内容), 但是我还未看到那次提交所引用的 tree 对象本身。这里有一些其它的命令来凸显两者的差异从而看到这个 tree 对象。

```
$ git rev-parse HEAD
588483b99a46342501d99e3f10630cfc1219ea32
# different on your system
$ git cat-file -t HEAD
commit
$ git cat-file commit HEAD
tree 0563f77d884e4f79ce95117e2d686d7d6e282887
author John Wiegley <johnw@newartisans.com> 12
committer John Wiegley <johnw@newartisans.com>
```

第一条命令解码别名 HEAD 为其引用的提交, 第二条命令确认它的类型，第三条命令显示提交所关联的 tree 对象的哈希标识符，与其他信息一同存储在

commit 对象中。我仓库中 commit 对象的哈希标识符是独一无二的 —-因为它包含了我的姓名和提交时间 —-但是你我之间 tree 对象的哈希标识符是相同的，在同样的命名之下它包含相同的 blob 对象。我们确认一下两者的确是同一个 tree 对象

```
$ git ls-tree 0563f77
100644 blob af5626b4a114abcb82d63db7c8082c3c47
```

现在你应该有这样的认识了：我的仓库含有一个提交，这个提交引用了一个 tree 对象，这个 tree 对象上依附有一个 blob，blob 含有我们想要记录的内容。我可以运行另外一些命令来确认确实是这样的情况。

```
$ find .git/objects -type f | sort
.git/objects/05/63f77d884e4f79ce95117e2d686d7
.git/objects/58/8483b99a46342501d99e3f10630cf
.git/objects/af/5626b4a114abcb82d63db7c8082c3
```

从输出结果中我们看到我的整个仓库含有三个对象，每一个对象都有一个在前面例子中已经出现过的哈希标识符。我们最后再来看看这些对象的类型，满足一下我们的好奇心。

```
$ git cat-file -t 588483b99a46342501d99e3f1063
commit
$ git cat-file -t 0563f77d884e4f79ce95117e2d68
tree
$ git cat-file -t af5626b4a114abcb82d63db7c808
blob
```

现在我可以通过 show 命令来查看这些对象明确的内容，但我决定把这个作为练习留给读者。

[7]:
[15]: 译者注：ls-tree 命令的可以传入的参数包括：提交，标签，tree 对象，ls-tree 命令通过 get_sha1 获取传入参数的哈希码，将哈希码传递给 parse_tree_indirect 函数获取其索引的 tree 对象。此处我们传递 HEAD 给 ls-tree 命令。

tree 对象如何形成

tree 对象如何形成
每一个提交都会关联一个 tree 对象，但 tree 对象是如何形成的？我们知道 blob 是填充你的文件的内容至 blob 中形成的 —-tree 对象拥有这些 blob—-但我们还不知道拥有这些 blob 的 tree 对象是如何形成的，也不知道 tree 对象是如何关联至提交。
我们在一个新的示例仓库中重新开始，但这次我们要手动完成一些事情，这样你就可以明确知道内部发生了什么。

```
$ rm -fr greeting .git
$ echo 'Hello, world!' > greeting
$ git init
$ git add greeting
```

这一切开始于你添加一个文件到暂存区。我们暂时将暂存区 (index) 视为你用来通过文件初始化 blob 对象的工具。当我添加 greeting 文件时，仓库中发生了一点改变。暂时无法以提交的形式观察到，但仍有一种途径让我说出发生了何种改变。

```
$ git log # 此命令失败，仓库中还没有提交
fatal: bad default revision 'HEAD'
$ git ls-files --stage # 列出暂存区中的
blob
100644 af5626b4a114abcb82d63db7c8082c3c4756e5
```

什么？还没有提交任何东西到仓库，但这里却有一个已经形成的对象。它有与文章开头处相同的哈希码，因此我知道它代表了 greeting 文件的内容。通过对这个哈希码使用 cat file -t 命令，我们可以确认它的类型确实是一个 blob 。实际上，这个 blob 和我们第一次创建示例仓库时的 blob 是相同的。相同的文件内容会产生相同的 blob(再次指出只是为了防止我对这一点强调的不够)。

这个 blob 还没有 tree 对象与之关联，也不存在任何提交。此时只有 .git/index 文件与之关联，现在我们创建一个 tree 对象使得我们的 blob 对象可以依附。

```
$ git write-tree # record the contents of the in
0563f77d884e4f79ce95117e2d686d7d6e282887
```

这个数字同样很眼熟：具有相同 blob(及自 tree 对象) 拥有相同的哈希码。我暂时还没有一个提交，但是仓库中有一个 tree 对象关联着 blob。底层命令 write-tree 的作用就是无论暂存区中的内容是什么，为了创建一个提交，将其攒一个新的 tree 对象。

我可以直接使用 tree 对象手动产生 commit 对象，这恰恰就是 commit-tree 命令的功能。

```
$ echo "Initial commit" | git commit-tree 0563f
5f1bc85745dcccce6121494fdd37658cb4ad441f
```

命令 commit-tree 将 tree 对象的哈希码作为参数，创建了一个提交来关联它。如果我希望新形成的提交对象拥有父代提交，则需要显式使用 -p 选项指定父代提交的哈希码。还需留意此处提交的哈希码与你系统上的不同：这是因为我的提交参考我的名字和提交日期，这两个细节总与你的不同。

我们的任务还没有完成，因为还没有注册这次提交为新的当前分支的 HEAD 。

```
$ echo 5f1bc85745dcccce6121494fdd37658cb4ad441
```

这条命令告诉 Git：名为 master 分支现在应该指向我们刚才的提交。另一种比较安全的方法是使用 update-ref 命令：

```
$ git update-ref refs/heads/master 5f1bc857
```

创建完 master 分支，我们必须将它与我们的工作目录关联起来，这通常会在你检出分支时发生:

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

这句命令将 HEAD 与 master 分支符号链接。这是极为重要的因为将来工作目录中的提交会自动更新 refs/heads/master 的值。
难以置信就是这么简单，现在可以使用 log 命令来查看我“打造”的提交。

```
$ git log
commit 5f1bc85745dcccce6121494fdd37658cb4ad44
Author: John Wiegley <johnw@newartisans.com>
Date:   Mon Apr 14 11:14:58 2008 -0400
    Initial commit
```

> 旁注：如果我没有设定 refs/heads/-master 指向新产生的提交，它将被视为“不可到达的 (unreachable)”, 因为没有任何东西索引它，它也不是可到达 (reachable) 的提交的父代提交。这种情形下，该提交会在某个时刻同它的 tree 对象, 它所有的 blob 被移除。(这些都是通过 gc 命令自动完成的，你几乎不需要手动执行它)。如我们上面所做的那样，通过将提交链接至 refs/-heads 中的一个名称，它成为了一个可到达的提交, 这样确保了它从现在开始被保留下来。

提交之美

提交之美

一些版本控制系统将分支 (branch) 定义为与“主线”或“主干”区分的“魔物”，另外一些则对这个名词争论不休，好像它与提交 (commit) 完全不同。

在 Git 的世界里，Branch 不作为一种独立的类型: 这里只有 Blob, 树对象，提交。16因为一个提交可能有一个或者多个父代提交，而这些父代提交又有父代提交，这就允许将某一个提交就可以看作一个分支，因为它拥有全部的修改至它本身的历史。

你可以任何时候运行 branch 命令检视最新的提交:

```
$ git branch -v
* master 5f1bc85 Initial commit
```

一个分支无非是一个被命名的对提交的引用，从这个角度来看标记和提交是相同的，唯一的不同是标记可以有自己的描述，而分支只是一个名字，标记是具有描述性的，不然怎么能叫标记呢。

实际上，我们一点也不需要使用别名 (alias)，例如，我可以通过哈希标识符索引仓库中的任何东西。下面是我直接定位和重置我现在的工作目录到某个特定的提交:

```
$ git reset --hard 5f1bc85
```

这里的 –hard 选项用来清除所有工作目录中的修改，不论他们有没有为了某次检入 (check in) 而记被录下来 (稍后详细讨论这个命令)。更为安全的方式应该使用 checkout 命令。:

```
$ git checkout 5f1bc85
```

两者的区别是，这种方式下工作目录中已经改变的文件会保留下来。如果对 checkout 命令添加 -f 选项，则它和 reset –hard 作用基本相同，除了 checkout 仅仅改变工作目录而 reset –hard 会改变当前分支的 HEAD 引用这一特定的 tree 对象。基于提交的系统还带来另外一个乐趣，那就是用一个词“提交”来表示其他最为复杂的版本控制术语。例如一个提交有多个父代提交，它被称为合并的提交 (merge commit)—-因为它将多个提交合并为一个，如果一个提交拥有多个子代提交，它被称为一系列分支的祖先, 等等。对 Git 而言，这些都没有什么区别: 对它而言，世界仅仅是提交构成的集合，每一个提交索引一个 tree 对象，这个 tree 对象索引其它的 tree 对象或 Blob, Blob 存储你的数据。比这些复杂的其他的仅仅就是命名的方式 (? device of nomenclature ?)

下面给出一幅图将这些片段拼凑起来：

爱看豆出品