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 --full 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 commits 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。