用户手册

用户手册

GIT是一种快速分布式修正控制系统。

本手册是由有基本UNIX命令行技能的人阅读的，但以前没有Git的知识。

[被称为“仓库和分支”的部分](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_repositories-and-branches)和[“探索GIT历史”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_exploring-git-history)解释如何使用Git获取和研究一个项目——阅读这些章节来学习如何构建和测试软件项目的特定版本、搜索回归等。

需要做实际开发的人也要阅读。[“以Git发展”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_Developing-With-git)和[“与他人共享发展”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_sharing-development).

进一步章节涵盖更专门的主题。

综合参考文档可通过MAN页面获得，或[被称为“Git帮助（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-help(1))命令。例如，对于命令*git clone <repo>*，您可以使用：

$ man git-clone

或

$ git help clone

使用后者，您可以使用您所选择的手动查看器；[被称为“Git帮助（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-help(1))欲了解更多信息

也见[“Git快速引用”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_git-quick-start)对于Git命令的简要概述，没有任何解释。

最后，看到[称为“本手册的注释和待办事项列表”](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_todo)你可以帮助这个手册更完整。

储存库和分支机构

如何获得Git库

当你阅读本手册时，有一个Git储存库来进行实验是很有用的。

获得一个最好的方法是使用[“Git克隆（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-clone(1))命令下载现有存储库的副本。如果你还没有一个项目，这里有一些有趣的例子：

# Git itself (approx. 40MB download):  
$ git clone git://git.kernel.org/pub/scm/git/git.git  
# the Linux kernel (approx. 640MB download): Linux内核（大约640MB下载）：  
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git $Git克隆GIT://GIT.Keln.Org/PUB/SCM/LIUX/KNEL/GIT/TURVALDS/LIUX.GIT

对于大型项目，初始克隆可能耗时，但只需要克隆一次即可。

克隆命令创建一个以项目命名的新目录（*吉特*或*Linux*在上面的例子中）。在CD进入该目录后，您将看到它包含一个项目文件的副本，称为[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)，以及一个名为顶级目录的特殊目录*Git*，它包含有关项目历史的所有信息。

如何检查项目的不同版本

Git被认为是存储文件集合历史的工具。它将历史存储为项目内容的相关快照的压缩集合。在Git中，每个这样的版本被称为[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit).

这些快照不一定都是从最老到最新的一行排列的，相反，工作可以同时沿着平行的发展线进行。[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)，which may merge and diverge.

单个GIT存储库可以跟踪多个分支上的开发。它通过保存一个[头](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)它引用每个分支上的最新提交；[“Git分支（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-branch(1))命令显示您的分支主管名单：

$ git branch  
\* master

新克隆的存储库包含一个单独的分支头，默认情况下称为“主”，工作目录初始化为该分支头所指的项目状态。

大多数项目也使用[标签](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag). 标签，像头一样，是项目历史的引用，可以使用[被称为“Git标签（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-tag(1))命令：

$ git tag -l  
v2.6.11  
v2.6.11-tree  
v2.6.12  
v2.6.12-rc2  
v2.6.12-rc3  
v2.6.12-rc4  
v2.6.12-rc5  
v2.6.12-rc6  
v2.6.13  
...

标签预计总是指向同一版本的项目，而负责人预期随着发展的进步。

创建一个新的分支头指向这些版本中的一个并使用它检查[“Git结账（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-checkout(1))：

$ git checkout -b new v2.6.13

然后，工作目录反映了项目被标记为v2.613时的内容，以及[“Git分支（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-branch(1))显示两个分支，标有当前签出的分支的星号：

$ git branch  
  master  
\* new

如果您决定更愿意查看版本2.617，则可以将当前分支修改为指向V2.617，而不是

$ git reset --hard v2.6.17

请注意，如果当前分支头是历史上唯一指向某个特定点的引用，那么重置该分支可能会让您无法找到它指向的历史，因此请小心使用此命令。

了解历史：提交

项目历史的每一个变化都由一个提交来表示。这个[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))命令显示当前分支上的最新提交：

$ git show  
commit 17cf781661e6d38f737f15f53ab552f1e95960d7  
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>  
Date:   Tue Apr 19 14:11:06 2005 -0700  
  
    Remove duplicate getenv(DB\_ENVIRONMENT) call  
  
    Noted by Tony Luck.  
  
diff --git a/init-db.c b/init-db.c  
index 65898fa..b002dc6 100644  
--- a/init-db.c  
+++ b/init-db.c  
@@ -7,7 +7,7 @@  
  
 int main(int argc, char \*\*argv)  
 {  
-       char \*sha1\_dir = getenv(DB\_ENVIRONMENT), \*path;  
+       char \*sha1\_dir, \*path;  
        int len, i;  
  
        if (mkdir(".git", 0755) < 0) {

正如你所看到的，提交表明谁做了最新的改变，他们做了什么，为什么。

每个提交都有40个六位ID，有时称为“对象名称”或“SHA-1 ID”，在第一行上显示。*吉特秀*输出。通常可以用较短的名称来引用提交，例如标记或分支名称，但这个较长的名称也可以是有用的。最重要的是，它是这个提交的全局唯一名称：因此，如果您告诉别人对象名称（例如，在电子邮件中），那么您可以保证该名称将引用其存储库中与您的存储库中相同的提交（假设它们的存储库全部具有该提交）。由于对象名称在提交的内容上被计算为散列，所以您可以保证提交不会更改，也不会更改其名称。

事实上，在[“Git概念”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_git-concepts)我们将看到存储在GIT历史中的所有内容，包括文件数据和目录内容，都存储在一个对象中，该对象的名称是其内容的散列。

理解历史：提交、父母和可达性

每个提交（除了项目中的第一个提交）也有一个父提交，它显示在提交之前发生了什么。跟随父母的链条最终会带你回到项目的开始。

然而，提交不构成一个简单的列表；Git允许开发线发散，然后再汇聚，而将两条开发线重新汇聚的点称为“合并”。因此，表示合并的提交可以具有不止一个父级，每个父级表示最新的提交，这是导致该点的开发线之一。

最好的方法是如何使用[“GITK（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitk(1))命令；在Git存储库上运行GITK并查找合并提交将有助于理解Git如何组织历史。

下面，如果提交X是提交Y的祖先，我们可以说，提交X是“可达”的。等价地，你可以说Y是X的后代，或者有一个从提交Y到提交X的父母们链。

认识历史：历史图解

我们有时会用下面的图表来表示Git历史。提交被显示为“O”，它们之间的链接用-/和-\绘制的线条。时间从左到右：

o--o--o <-- Branch A

/

o--o--o <-- master

\

o--o--o <-- Branch B

如果我们需要讨论一个特定的提交，字符“O”可以用另一个字母或数字代替。

理解历史：一个分支是什么？

当我们需要精确的时候，我们将使用“分支”这个词来表示一条开发线，和“分支头”（或者仅仅是“头”）来表示对分支上的最近提交的引用。在上面的示例中，命名为“A”的分支头是指向一个特定提交的指针，但是我们指的是三个引线直到这一点，都是“分支A”的一部分。

然而，当没有混乱会导致，我们经常只使用术语“分支”既为分支机构和分支机构负责人。

分支操作

创建、删除和修改分支是简单快捷的，下面是命令的摘要：

git branch

列出所有分支

*git branch <branch>*

创建一个新的分支*<branch>*将历史中的同一点引用为当前分支。

*git branch <branch> <start-point>*

创建一个新的分支*<branch>*引用*<start-point>*可以使用任何您喜欢的方式指定，包括使用分支名称或标签名。

*git branch -d <branch>*

删除分支*<branch>*如果分支未完全合并在其上游分支中或包含在当前分支中，则此命令将以警告失败。

*git branch -D <branch>*

删除分支*<branch>*无论其合并状态如何

*git checkout <branch>*

制作当前的分支*<branch>*更新工作目录以反映引用的版本*<branch>*.

*git checkout -b <new> <start-point>*

创建新的分支*<new>*引用*<start-point>*并检查一下

特殊符号“头”总是可以用来指当前分支。事实上，Git使用一个名为*头部*在*Git*要记住哪个分支是当前的目录：

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

检查旧版本而不创建新分支

这个*GIT校验*命令通常期望分支头，但也将接受任意提交；例如，可以检查标记引用的提交：

$ git checkout v2.6.17  
Note: checking out 'v2.6.17'.  
  
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.  
  
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:  
  
  git checkout -b new\_branch\_name  
  
HEAD is now at 427abfa... Linux v2.6.17

然后，头指向提交的SHA-1而不是分支，Git分支表明您不再在分支上：

$ cat .git/HEAD  
427abfa28afedffadfca9dd8b067eb6d36bac53f  
$ git branch  
\* (detached from v2.6.17)  
  master

在这种情况下，我们说头部是“分离的”。

这是一个简单的方法来检查一个特定的版本，而不必为新的分支创建一个名称。如果你决定的话，你仍然可以为这个版本创建一个新的分支（或标签）。

从远程存储库检查分支

在您克隆时创建的“主”分支是您克隆的存储库中的头的副本。该存储库可能还拥有其他分支，而本地存储库保存跟踪这些远程分支中的每个分支的分支，称为远程跟踪分支，您可以使用*-r*选择权[“Git分支（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-branch(1))：

$ git branch -r  
  origin/HEAD  
  origin/html  
  origin/maint  
  origin/man  
  origin/master  
  origin/next  
  origin/pu  
  origin/todo

在这个例子中，“原点”被称为远程存储库，或者简称为“远程”。从我们的观点来看，这个存储库的分支被称为“远程分支”。上面列出的远程跟踪分支是基于克隆时间的远程分支创建的，将由*吉特取出*汉斯*GIT拉力*）*Git推*. 见[称为“用Git获取更新存储库”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_Updating-a-repository-With-git-fetch)详情

您可能希望在自己的分支上建立这些远程跟踪分支中的一个，就像您希望标记一样：

$ git checkout -b my-todo-copy origin/todo

你也可以退房*一切起源*直接检查或写一次性补丁。见[分离头](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_detached-head).

请注意，名称“原产地”只是Git默认使用的名称，指的是从您克隆的存储库。

命名分支、标签和其他引用

分支、远程跟踪分支和标签都是提交的引用。所有引用都以斜杠分隔路径名命名，从*参考文献*到目前为止，我们使用的名字实际上是速记：

* The branch *test* is short for *refs/heads/test*. 支路*测试*是短的*参考/测试/测试*.
* The tag *v2.6.18* is short for *refs/tags/v2.6.18*. 标签*v2.6.18*是短的*参考文献/标签/V2.618*.
* *origin/master* is short for *refs/remotes/origin/master*. *起源/大师*是短的*参考/遥控器/原点/主控器*.

例如，如果存在标签和具有相同名称的分支，则全名有时是有用的。

（新创建的参考文件实际上存储在*GIT/REF*目录下，按其名称给出的路径。然而，出于效率的原因，它们也可以打包在一个文件中；[被称为“Git PACK REFS（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pack-refs(1))）

作为另一个有用的快捷方式，存储库的“头”可以使用该存储库的名称来引用。因此，例如，“原点”通常是存储库“起源”中的头分支的捷径。

对于Git检查引用的完整路径列表，以及在使用具有相同的速记名称的多个引用时，它决定使用哪一个顺序，请参阅“指定修订”部分。[被称为“GIT修订版（7）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitrevisions(7)).

用Git获取更新存储库

在克隆一个存储库并提交一些自己的更改之后，您可能希望检查原始存储库进行更新。

这个*吉特取出*没有参数的命令将更新所有远程跟踪分支到原始存储库中找到的最新版本。它不会接触任何你自己的分支——甚至不是为克隆人创造的“大师”分支。

从其他存储库中获取分支

你也可以从你所克隆的库中追踪分支，使用[被称为“Git远程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-remote(1))：

$ git remote add staging git://git.kernel.org/.../gregkh/staging.git  
$ git fetch staging  
...  
From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging  
 \* [new branch]      master     -> staging/master  
 \* [new branch]      staging-linus -> staging/staging-linus  
 \* [new branch]      staging-next -> staging/staging-next

新的远程跟踪分支将根据您给出的速记名称存储。*Git远程添加*在这种情况下*分期*：

$ git branch -r  
  origin/HEAD -> origin/master  
  origin/master  
  staging/master  
  staging/staging-linus  
  staging/staging-next

如果你跑*git fetch <remote>*后来，远程跟踪分支被命名*<remote>*将更新

如果检查文件*.git/config*你会看到Git增加了一个新的节段：

$ cat .git/config  
...  
[remote "staging"]  
        url = git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging.git  
        fetch = +refs/heads/\*:refs/remotes/staging/\*  
...

这是导致Git跟踪远程分支的原因；您可以通过编辑修改或删除这些配置选项。*.git/config*使用文本编辑器。（参见“配置文件”部分）[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))详情

## Exploring Git history 探索GIT史

Git被认为是存储文件集合历史的工具。它通过存储文件层次结构的内容的压缩快照，以及表示这些快照之间的关系的“提交”来实现这一点。

GIT为探索项目的历史提供了极其灵活和快速的工具。

我们从一个专门的工具开始，用于发现将bug引入到项目中的承诺。

如何使用二等分回归

假设项目的2.618版本工作，但是“主”版本崩溃。有时，找到这种回归的原因最好的方法是通过项目的历史来进行一次蛮力搜索，以找到导致问题的特定提交。这个[“Git二等分（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-bisect(1))命令可以帮助你做到这一点：

$ git bisect start  
$ git bisect good v2.6.18  
$ git bisect bad master  
Bisecting: 3537 revisions left to test after this  
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try #6]

如果你跑*GIT分支*在这一点上，您将看到Git暂时将您移到“（没有分支）”。现在，头与任何分支分离，直接指向一个提交（带有提交ID 65934…），它可以从“主”而不是从V2.618到达。编译并测试它，看看它是否崩溃。假设它坠毁了。然后：

$ git bisect bad  
Bisecting: 1769 revisions left to test after this  
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings

检查一个旧版本。继续这样，在每一阶段告诉Git，它给你的版本是好还是坏，并且注意每次测试的修改次数大约减少一半。

经过大约13次测试（在这种情况下），它将输出有罪提交的提交ID。然后可以检查提交。[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))找出谁写的，然后用提交ID把bug报告发给他们。最后，运行

$ git bisect reset

让你回到从前的树枝上。

Note that the version which*二分查找*在每个点为你检查只是一个建议，如果你认为这是一个好主意，你可以尝试不同的版本。例如，偶尔你可能会犯下一桩不相关的事情；

$ git bisect visualize

它将运行GITK，并用一个称为“平分”的标记来标记它所选择的提交。选择一个安全外观提交，注意它的提交ID，并检查出来：

$ git reset --hard fb47ddb2db...

然后测试，运行*平分好*或*二分差*作为适当的人，继续

而不是*GIT二分可视化*然后*Git复位-硬FB47 DDB2DB…*您可能只想告诉Git您想跳过当前提交：

$ git bisect skip

然而，在这种情况下，Git可能最终无法在第一次跳过的提交和稍后的错误提交之间告诉第一个坏的。

如果你有一个测试脚本可以区分一个好的和一个坏的提交，那么也有办法使平分过程自动化。见[“Git二等分（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-bisect(1))有关此等的更多信息*二分查找*功能特色

命名约定

我们已经看到了几种命名提交方式：

* 40-hexdigit object name 40位数字对象名
* branch name: refers to the commit at the head of the given branch 分支名称：指给定分支头部的提交。
* tag name: refers to the commit pointed to by the given tag (we've seen branches and tags are special cases of [references](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_how-git-stores-references)). 标签名称：指由给定标签指向的提交（我们已经看到分支和标签是特殊情况）。[推荐信](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_how-git-stores-references)）
* HEAD: refers to the head of the current branch 头：指当前分支的头

还有更多；参见“指定修订”部分[被称为“GIT修订版（7）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitrevisions(7))手册页的名称清单的修订方法。一些例子：

$ git show fb47ddb2 # the first few characters of the object name  
                    # are usually enough to specify it uniquely  
$ git show HEAD^    # the parent of the HEAD commit  
$ git show HEAD^^   # the grandparent  
$ git show HEAD~4   # the great-great-grandparent

请记住，合并提交可能有多个父级；默认情况下，*^*和*~*遵循提交中列出的第一个父级，但您也可以选择：

$ git show HEAD^1   # show the first parent of HEAD  
$ git show HEAD^2   # show the second parent of HEAD

除了头部之外，还有几个其他特殊的名称：

合并（稍后将讨论）以及操作，例如*Git复位*更改当前签出的提交，通常将OrriHead设置为当前操作之前的值头。

这个*吉特取出*操作总是将最后提取的分支的头存储在FutCHyHead中。例如，如果您运行*吉特取出*不指定本地分支作为操作的目标

$ git fetch git://example.com/proj.git theirbranch

所取得的承诺仍将从费希海德获得。

当我们讨论合并时，我们还会看到特殊名称MelGeHead，它指的是我们正在合并到当前分支的另一个分支。

这个[被称为“Git Rev解析（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-rev-parse(1))命令是一个低级命令，偶尔用于将提交的名称转换为提交的对象名称是有用的：

$ git rev-parse origin  
e05db0fd4f31dde7005f075a84f96b360d05984b

创建标签

我们还可以创建一个标签来引用特定的提交；运行之后

$ git tag stable-1 1b2e1d63ff

你可以使用*稳定-1*参考提交1B2E1D63FF。

这将创建一个“轻量级”标签。如果您还希望包含标记的注释，并且可能以加密方式签名，则应该创建标记对象；[被称为“Git标签（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-tag(1))详情页

浏览修改

这个[被称为“Git log（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-log(1))命令可以显示提交列表。它本身显示了所有从父提交中获得的提交；但是您也可以提出更具体的请求：

$ git log v2.5..        # commits since (not reachable from) v2.5  
$ git log test..master  # commits reachable from master but not test  
$ git log master..test  # ...reachable from test but not master  
$ git log master...test # ...reachable from either test or master,  
                        #    but not both  
$ git log --since="2 weeks ago" # commits from the last 2 weeks  
$ git log Makefile      # commits which modify Makefile  
$ git log fs/           # ... which modify any file under fs/  
$ git log -S'foo()'     # commits which add or remove any file data  
                        # matching the string 'foo()'

当然，你可以把所有这些组合起来，下面的发现是从V2.5触碰到的。*生成文件*或任何文件下*FS*：

$ git log v2.5.. Makefile fs/

您也可以要求Git日志显示补丁：

$ git log -p

见*--pretty*期权在[被称为“Git log（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-log(1))更多页面显示选项

注意，Git日志从最新的提交开始，并且通过父本反向工作；然而，由于Git历史可以包含多条独立的开发线，因此提交的特定顺序可能有点武断。

产生差异

可以使用任意两个版本生成差异[被称为“Git DIFF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff(1))：

$ git diff master..test

这将产生两个分支尖端之间的差异。如果你想从他们的共同祖先中找到差异来测试，你可以用三个点代替两个点：

$ git diff master...test

有时，你想要的是一组补丁，为此你可以使用。[“Git格式修补程序（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-format-patch(1))：

$ git format-patch master..test

将生成一个带有补丁的文件，用于从测试中获得的每个提交，而不是从主机。

查看旧文件版本

您可以通过首先查看正确的修订来查看文件的旧版本。但有时在没有检查任何东西的情况下，能够查看单个文件的旧版本更方便；

$ git show v2.5:fs/locks.c

冒号之前可能是任何一个命名提交，并可能是任何路径的文件跟踪的GIT。

实例

计算分支的提交数

假设你想知道你做出了多少承诺*迈科*因为它偏离了*起源*：

$ git log --pretty=oneline origin..mybranch | wc -l

或者，你可能经常看到用低级命令完成这类事情。[被称为“GIT Rev列表（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-rev-list(1))，只是列出了所有提交的SHA-1：

$ git rev-list origin..mybranch | wc -l

检查两个分支是否指向同一历史

假设您想检查两个分支是否指向历史上的同一个点。

$ git diff origin..master

会告诉你在两个分支中项目的内容是否相同，但是在理论上，同样的项目内容有可能通过两条不同的历史路线到达。您可以比较对象名称：

$ git rev-list origin  
e05db0fd4f31dde7005f075a84f96b360d05984b  
$ git rev-list master  
e05db0fd4f31dde7005f075a84f96b360d05984b

或者你可以回忆起*…*运算符选择从一个引用或另一个而不是两个都可到达的所有提交；

$ git log origin...master

当两个分支相等时，将不返回任何提交。

Find first tagged version including a given fix

假设您知道提交E05dB0FD修复了一个特定的问题。您希望找到包含该修复的最早标记释放。

当然，可能有不止一个答案——如果在提交E05dB0FD之后分支了历史，那么可能会出现多个“最早”标记的版本。

您可以从E05dB0FD视觉检查提交。

$ gitk e05db0fd..

或者你可以使用[被称为“Git名称Rev（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-name-rev(1))，它将根据提交给提交人的后代之一的任何标签给提交一个名称：

$ git name-rev --tags e05db0fd  
e05db0fd tags/v1.5.0-rc1^0~23

这个[“Git描述（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-describe(1))命令相反，使用给定提交的标记来命名修订：

$ git describe e05db0fd  
v1.5.0-rc0-260-ge05db0f

但是，这有时可以帮助您猜出在给定提交之后哪些标记可能出现。

如果只想验证给定标记的版本是否包含给定的提交，则可以使用[“Git合并基（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-merge-base(1))：

$ git merge-base e05db0fd v1.5.0-rc1  
e05db0fd4f31dde7005f075a84f96b360d05984b

合并基命令找到给定提交的共同祖先，并且在其中一个是另一个后代的情况下总是返回一个或另一个；因此，上面的输出表明E05dB0FD实际上是V1.5.0 RC1的祖先。

或者，请注意

$ git log v1.5.0-rc1..e05db0fd

当且仅当V1.5.0 RC1包含E05dB0FD时，将产生空输出，因为它只输出不可从V1.5.0 RC1获得的提交。

作为另一种选择，[“Git Sshow分支（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show-branch(1))命令列出从其参数可从左侧显示的提交，该声明指示可提交哪些参数。所以，如果你运行类似的东西

$ git show-branch e05db0fd v1.5.0-rc0 v1.5.0-rc1 v1.5.0-rc2  
! [e05db0fd] Fix warnings in sha1\_file.c - use C99 printf format if  
available  
 ! [v1.5.0-rc0] GIT v1.5.0 preview  
  ! [v1.5.0-rc1] GIT v1.5.0-rc1  
   ! [v1.5.0-rc2] GIT v1.5.0-rc2  
...

然后一条线

+ ++ [e05db0fd] Fix warnings in sha1\_file.c - use C99 printf format if  
available

表明E05dB0FD可以从其本身，从V1.5.0 RC1和V1.5.0 RC2，而不是从V1.5.0 RC0。

显示给定分支唯一的提交

假设您希望看到所有从分支头可获得的提交。*硕士*但不是来自存储库中的任何其他头。

我们可以在这个存储库中列出所有的头[被称为“Git show REF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show-ref(1))：

$ git show-ref --heads  
bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial  
db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint  
a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master  
24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2  
1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes

我们可以得到分支头名称，并删除*硕士*在标准公用事业削减和GRIPP的帮助下：

$ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'  
refs/heads/core-tutorial  
refs/heads/maint  
refs/heads/tutorial-2  
refs/heads/tutorial-fixes

然后，我们可以要求看到所有可以从主人那里获得的契约，而不是从其他人那里得到：

$ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |  
                                grep -v '^refs/heads/master' )

显然，无休止的变化是可能的，例如，查看从某个头可以到达的所有提交，而不是从存储库中的任何标签访问：

$ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )

（见[被称为“GIT修订版（7）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitrevisions(7))用于解释提交选择语法，如*——不是*）

为软件发布创建变更日志和TARPOLL

这个[“Git档案（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-archive(1))命令可以从项目的任何版本创建一个TAR或zip存档；例如：

$ git archive -o latest.tar.gz --prefix=project/ HEAD

将使用头来生成一个GZIPAR TAR存档，其中每个文件名都在前面。*项目*. 输出文件格式从输出文件扩展名中推断出来，如果可能的话，请参见[“Git档案（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-archive(1))详情

超过1.7.7的Git版本不知道tar.gz格式，您需要使用GZIP显式：

$ git archive --format=tar --prefix=project/ HEAD | gzip >latest.tar.gz

如果你正在发布一个软件项目的新版本，你可能想同时制作一个变更日志以包含在发布公告中。

例如，莱纳斯TurvDS通过标记它们来运行新的内核版本，然后运行：

$ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7

其中发行脚本是一个shell脚本，看起来像：

#!/bin/sh  
stable="$1"  
last="$2"  
new="$3"  
echo "# git tag v$new"  
echo "git archive --prefix=linux-$new/ v$new | gzip -9 > ../linux-$new.tar.gz"  
echo "git diff v$stable v$new | gzip -9 > ../patch-$new.gz"  
echo "git log --no-merges v$new ^v$last > ../ChangeLog-$new"  
echo "git shortlog --no-merges v$new ^v$last > ../ShortLog"  
echo "git diff --stat --summary -M v$last v$new > ../diffstat-$new"

然后，在验证输出命令后，他就剪切并粘贴输出命令。

查找引用给定内容的文件的提交

有人递给你一个文件的副本，并询问哪一个提交修改了一个文件，使得它在提交之前或之后包含给定的内容。你可以用这个发现：

$  git log --raw --abbrev=40 --pretty=oneline |  
        grep -B 1 `git hash-object filename`

找出这项工作的原因留给了（高级）学生做练习。这个[被称为“Git log（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-log(1))，[被称为“Git微分树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff-tree(1))和[称为“Git哈希对象（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-hash-object(1))人页可能是有用的

用Git开发

把你的名字告诉Git

在创建任何提交之前，您应该将自己介绍给Git。最简单的方法就是使用。[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))：

$ git config --global user.name 'Your Name Comes Here'  
$ git config --global user.email 'you@yourdomain.example.com'

它会将以下内容添加到名为*GITCONFIG*在您的家庭目录：

[user]  
        name = Your Name Comes Here  
        email = you@yourdomain.example.com

参见“配置文件”部分[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))有关配置文件的详细信息。该文件是纯文本，因此您也可以用您最喜欢的编辑器编辑它。

创建新的仓库

从头开始创建一个新的存储库非常容易：

$ mkdir project  
$ cd project  
$ git init

如果你有一些初始内容（比如，一个塔球）：

$ tar xzvf project.tar.gz  
$ cd project  
$ git init  
$ git add . # include everything below ./ in the first commit:  
$ git commit

如何做出承诺

创建一个新的提交需要三个步骤：

1. 使用您最喜欢的编辑器对工作目录进行一些更改。
2. 告诉Git关于你的变化。
3. 使用在步骤2中告诉Git的内容创建提交。

在实践中，您可以根据需要重复和重复步骤1和2：为了跟踪在步骤3所需提交的内容，Git在特定的“索引”区域中维护树的内容的快照。

在开始时，索引的内容将与头部的内容相同。命令*高速缓存*这显示了头和索引之间的差异，因此在该点上不产生输出。

修改索引很容易：

若要使用新文件或已修改文件的内容更新索引，请使用

$ git add path/to/file

若要从索引中删除文件，请从工作树中删除文件，请使用

$ git rm path/to/file

每一步之后你都可以验证

$ git diff --cached

总是显示头部和索引文件之间的差异——这是您现在创建提交时所提交的内容

$ git diff

显示工作树和索引文件之间的区别。

注意*Git添加*总是将文件的当前内容添加到索引中，否则将忽略对同一文件的进一步更改。*Git添加*再次归档

当你准备好了，就跑

$ git commit

Git会提示您提交消息，然后创建新的提交。检查以确保它看起来像你所期望的

$ git show

作为一种特殊的捷径，

$ git commit -a

将更新所有已修改或删除的文件并创建提交的索引，一步到位。

许多命令对于跟踪你要提交的内容是有用的：

$ git diff --cached # difference between HEAD and the index; what  
                    # would be committed if you ran "commit" now.  
$ git diff          # difference between the index file and your  
                    # working directory; changes that would not  
                    # be included if you ran "commit" now.  
$ git diff HEAD     # difference between HEAD and working tree; what  
                    # would be committed if you ran "commit -a" now.  
$ git status        # a brief per-file summary of the above.

你也可以使用[“Git GUI（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-gui(1))要创建提交，查看索引和工作树文件中的更改，并单独选择包含在索引中的DIFF对象（通过右键单击DIF-HUK并选择“阶段HOKK提交”）。

创建良好的提交消息

虽然不是必需的，但是用一个简短的（少于50个字符的）行开始提交消息，总结变化，然后是一个空行，然后是一个更全面的描述，这是个好主意。提交到提交消息中的第一个空行的文本被视为提交标题，并且在整个Git中使用该标题。例如，[“Git格式修补程序（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-format-patch(1))将提交转换为电子邮件，它使用主题行上的标题和正文中的其他提交。

忽略文件

项目通常会生成您所做的文件。*不*想用Git跟踪。这通常包括由编辑器生成的生成过程或临时备份文件生成的文件。当然，*不*用Git跟踪文件只是一个问题。*不*打电话*Git添加*在他们身上。但是，如果这些未被追踪的文件放在周围，很快就会变得恼人；*加特*实际上是无用的，他们总是在输出中出现。*GIT状态*.

通过创建一个文件，可以告诉Git忽略某些文件。*吉蒂格诺*在工作目录的顶层，内容包括：

# Lines starting with '#' are considered comments.  
# Ignore any file named foo.txt.  
foo.txt  
# Ignore (generated) html files,  
\*.html  
# except foo.html which is maintained by hand.  
!foo.html  
# Ignore objects and archives.  
\*.[oa]

见[被称为“GigTiGORE（5）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitignore(5))对于语法的详细解释。您也可以将GiTigNoR文件放置在工作目录中的其他目录中，它们将应用于这些目录和它们的子目录。这个*吉蒂格诺*文件可以像任何其他文件一样添加到存储库中（只运行）*GITIGNORE*和*GIT提交*通常情况下，当排除模式（例如模式匹配生成输出文件）对克隆您的存储库的其他用户也有意义时，这是很方便的。

如果希望排除模式只影响某些存储库（而不是针对给定项目的每个存储库），则可以将它们放在存储库中的文件中。*.git/info/exclude*，或在由*排除文件*配置变量。一些Git命令也可以直接在命令行中排除模式。见[被称为“GigTiGORE（5）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitignore(5))细节问题

如何合并

你可以重新加入发展的两个分支。[“Git合并（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-merge(1))：

$ git merge branchname

合并分公司的发展branchname进入当前分支

通过合并所做的更改来进行合并。branchname这些变化是由你当前分支中的最新提交做出的，因为它们的历史分叉。当合并完成时，工作树将被重写，或者当合并导致冲突时由半合并结果覆盖。因此，如果您有未提交的更改触及与合并所影响的文件相同的文件，则Git将拒绝继续进行。大多数情况下，您希望在合并之前提交更改，如果不进行合并，则[被称为“Git StAsh（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-stash(1))在进行合并时可以将这些更改移除，然后重新应用它们。

如果更改是足够独立的，Git将自动完成合并并提交结果（或者在现有情况下重用现有的提交）。[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards)，见下文）。另一方面，如果存在冲突——例如，如果在远程分支和本地分支中以两种不同的方式修改同一文件——则警告您；输出可能看起来像这样：

$ git merge next  
 100% (4/4) done  
Auto-merged file.txt  
CONFLICT (content): Merge conflict in file.txt  
Automatic merge failed; fix conflicts and then commit the result.

冲突标记留在问题文件中，在手动解决冲突之后，可以在创建新文件时按内容更新索引并运行Git提交。

如果使用GITK检查所得的提交，您将看到它有两个父级，一个指向当前分支的顶部，另一个指向另一个分支的顶部。

解决合并

当合并未自动解决时，Git将索引和工作树置于一个特殊状态，该状态为您提供了帮助解决合并所需的所有信息。

在索引中特别标记有冲突的文件，所以在解决问题并更新索引之前，[“Git提交（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-commit(1))将失败：

$ git commit  
file.txt: needs merge

也，[被称为“Git状态（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-status(1))将这些文件列为“未合并”，冲突的文件将添加冲突标记，如下所示：

<<<<<<< HEAD:file.txt  
Hello world  
=======  
Goodbye  
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt

您所需要做的就是编辑文件来解决冲突，然后

$ git add file.txt  
$ git commit

请注意，提交消息已经为您填充了有关合并的一些信息。通常，您可以只使用默认的消息不变，但如果需要的话，您可以添加自己的注释。

以上是您需要知道的所有解决简单合并的方法。但是GIT还提供了更多信息来帮助解决冲突：

在合并过程中获得冲突解决帮助

Git能够自动合并的所有更改都已添加到索引文件中，因此[被称为“Git DIFF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff(1))表演只有冲突。你用的是不寻常的Syntax：

$ git diff  
diff --cc file.txt  
index 802992c,2b60207..0000000  
--- a/file.txt  
+++ b/file.txt  
@@@ -1,1 -1,1 +1,5 @@@  
++<<<<<<< HEAD:file.txt  
 +Hello world  
++=======  
+ Goodbye  
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt

回想一下，在我们解决这个冲突后提交的提交将有两个父代而不是通常的一个：一个父级将是当前分支的头，另一个将是另一个分支的末端，该分支在MelgEHead中临时存储。

在合并过程中，索引保存每个文件的三个版本。这三个“文件阶段”中的每一个都代表文件的不同版本：

$ git show :1:file.txt  # the file in a common ancestor of both branches  
$ git show :2:file.txt  # the version from HEAD.  
$ git show :3:file.txt  # the version from MERGE\_HEAD.

当你问[被称为“Git DIFF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff(1))为了显示冲突，它在工作树中与阶段2和3进行冲突的合并结果之间的三方差异，只显示内容来自两侧的亨利（混合）（换句话说，当Hunk的合并结果只来自阶段2时），该部分不冲突且未示出。相同的第3阶段）。

上面的差异显示了Fiel.Txt的工作树版本与阶段2和阶段3版本之间的差异。因此，不要逐行逐行*+*或*-*它现在使用两个列：第一列用于第一个父级和工作目录副本之间的差异，第二个列用于第二个父级和工作目录副本之间的差异。（参见“组合差异格式”部分）[被称为“Git DIFF文件（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff-files(1))有关格式的细节

在以明显的方式解决冲突之后（但在更新索引之前），差异将看起来像：

$ git diff  
diff --cc file.txt  
index 802992c,2b60207..0000000  
--- a/file.txt  
+++ b/file.txt  
@@@ -1,1 -1,1 +1,1 @@@  
- Hello world  
 -Goodbye  
++Goodbye world

这表明我们的解决版本从第一个父代中删除了“hello World”，从第二个父代中删除了“再见”，并添加了“再见世界”，这两个词以前都不存在。

一些特殊的Diff选项允许将工作目录与这些阶段中的任何一个相分离：

$ git diff -1 file.txt          # diff against stage 1  
$ git diff --base file.txt      # same as the above  
$ git diff -2 file.txt          # diff against stage 2  
$ git diff --ours file.txt      # same as the above  
$ git diff -3 file.txt          # diff against stage 3  
$ git diff --theirs file.txt    # same as the above.

这个[被称为“Git log（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-log(1))和[“GITK（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitk(1))命令还为合并提供特殊帮助：

$ git log --merge  
$ gitk --merge

这些将显示只存在于头上或MyGeHead上的所有提交，并触及未合并文件。

您也可以使用[被称为“Git CyrutoL（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-mergetool(1))这允许使用外部工具（如Emacs或KDIF3）合并未合并文件。

每次解决文件中的冲突并更新索引时：

$ git add file.txt

该文件的不同阶段将“崩溃”，之后*差异比较*（默认情况下）将不再显示该文件的差异。

撤消合并

如果你陷入困境，决定放弃，扔掉整个烂摊子，你总是可以回到预合并状态。

$ git reset --hard HEAD

或者，如果你已经提交了你想扔掉的合并，

$ git reset --hard ORIG\_HEAD

但是，在某些情况下，最后一个命令可能是危险的，如果提交本身可能已经合并到另一个分支中，则决不要丢弃已经提交的提交，因为这样做可能会混淆进一步合并。

快进合并

有一个特殊情况，没有提到上面，这是不同的对待。通常，合并会导致合并提交，其中有两个父节点，一个指向合并的两条开发线中的每一条。

但是，如果当前分支是另一个的祖先，那么当前分支中的每个提交都已经包含在另一个分支中，那么Git只是执行一个“快进”；当前分支的头向前移动，指向合并的分支的头部，而不创建任何新的提交。

固定错误

如果你把工作树弄乱了，但是还没有犯错误，你可以把整个工作树返回到最后提交的状态。

$ git reset --hard HEAD

如果你犯了一个你以后不希望犯的错误，有两个根本不同的方法来解决这个问题：

1. 您可以创建一个新的提交，撤销旧提交所做的任何操作。如果你的错误已经公开了，这是正确的。
2. 你可以回去修改旧的提交。如果你已经把历史公开了，你不应该这样做；Git通常不期望项目的“历史”发生改变，并且不能正确地执行一个有历史改变的分支的重复合并。

用新的承诺修复错误

创建一个新的提交以恢复较早的更改是非常容易的；[被称为“Git回复（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-revert(1))命令对错误提交的引用；例如，还原最近的提交：

$ git revert HEAD

这将创建一个新的提交，撤销头部的更改。您将有机会编辑新提交的提交消息。

您也可以恢复先前的更改，例如，下一个更改：

$ git revert HEAD^

在这种情况下，Git将尝试撤销旧的更改，同时保留完整的任何更改。如果最近的更改与要还原的更改重叠，那么您将被要求手动修复冲突，就像在[解决合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_resolving-a-merge).

通过改写历史来纠正错误

如果有问题的提交是最新的提交，并且尚未公开该提交，那么您可能只是[使用它销毁它*Git复位*](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_undoing-a-merge).

或者，你可以编辑工作目录并更新索引来修复错误，就像你要去做的那样。[创建新提交](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_how-to-make-a-commit)然后运行

$ git commit --amend

它将用新的提交替换旧的提交，合并您的更改，给您一个编辑旧提交消息的机会。

同样，您不应该对可能已经合并到另一个分支的提交进行此操作；[被称为“Git回复（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-revert(1))相反，在那种情况下

也有可能在历史上进一步取代提交，但这是一个有待进一步研究的主题。[另一章](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_cleaning-up-history).

检查文件的旧版本

在撤销以前坏更改的过程中，您可能会发现使用特定文件的旧版本是有用的。[“Git结账（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-checkout(1)). 我们已经用过*GIT校验*在切换分支之前，如果给定路径名，则具有完全不同的行为：命令

$ git checkout HEAD^ path/to/file

通过提交头中的内容替换路径/to/file，并更新匹配的索引。它不会改变树枝。

如果只想查看旧版本的文件，而不修改工作目录，就可以这样做。[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))：

$ git show HEAD^:path/to/file

这将显示文件的given版本。

暂时搁置正在进行的工作

当你正在处理一些复杂的事情时，你会发现一个不相关但又明显又微不足道的错误。在继续之前，您需要修复它。你可以使用[被称为“Git StAsh（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-stash(1))为了保存工作的当前状态，并且在修复bug之后（或者，可选地在不同分支上这样做之后再回来），不要隐藏正在进行中的工作更改。

$ git stash save "work in progress for foo feature"

这个命令可以将更改保存到stash并重置工作树和索引以匹配当前分支的尖端。然后你可以像往常一样修理。

... edit and test ...  
$ git commit -a -m "blorpl: typofix"

之后，你可以回到你正在从事的工作中去。 git stash pop：

$ git stash pop

确保良好性能

在大型存储库上，Git依赖于压缩来保持历史信息占用磁盘或内存中的太多空间。一些Git命令可以自动运行[“Git GC（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-gc(1))因此，你不必担心手动运行它。然而，压缩大型存储库可能需要一段时间，因此您可能需要调用*GC*明确地避免在不方便的时候自动压缩。

确保可靠性

检查仓库的腐败

这个[“Git FSCK（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fsck(1))命令在存储库上运行多个自我一致性检查，并报告任何问题。这可能需要一些时间。

$ git fsck  
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3  
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63  
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5  
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb  
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f  
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e  
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085  
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f  
...

您将看到悬空对象上的信息消息。它们是存储库中仍然存在的对象，但不再被任何分支引用，并且可以（或将）在一段时间后移除。*GC*. 你可以跑*GIT FSCK——不悬挂*来抑制这些消息，并仍然查看实际错误。

找回丢失的变化

副歌

假设你修改了一个分支[*重置-硬*](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fixing-mistakes)然后意识到分支是历史上唯一的参照物。

幸运的是，Git还保存一个日志，称为“ReFug”，每个分支的所有先前值。因此，在这种情况下，您仍然可以使用旧的历史，例如，

$ git log master@{1}

这列出了从以前版本的*硕士*分支头。此语法可以与任何接受提交的Git命令一起使用，而不只是使用*GIT日志*. 其他一些例子：

$ git show master@{2}           # See where the branch pointed 2,  
$ git show master@{3}           # 3, ... changes ago.  
$ gitk master@{yesterday}       # See where it pointed yesterday,  
$ gitk master@{"1 week ago"}    # ... or last week  
$ git log --walk-reflogs master # show reflog entries for master

为头部保留一个单独的重新漂浮物，所以

$ git show HEAD@{"1 week ago"}

将显示一个星期前的头指向，而不是一个星期前的当前分支指向什么。这可以让你看到你所签出的历史。

默认保存30天，然后修剪。见[被称为“Git ReFug（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-reflog(1))和[“Git GC（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-gc(1))学习如何控制剪枝，并查看“指定修订”部分。[被称为“GIT修订版（7）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitrevisions(7))详情

请注意，重浮历史与正常GIT历史非常不同。虽然在同一个项目上运行的每个存储库共享正常历史，但ReFoLG历史不共享：它只告诉您本地存储库中的分支随着时间的推移是如何变化的。

检查悬空物体

在某些情况下，ReFug可能无法拯救你。例如，假设你删除一个分支，然后意识到你需要它包含的历史。ReFug也被删除；但是，如果还没有修剪存储库，那么您仍然可以在悬吊对象中找到丢失的提交。*GIT FSCK*报告见[“悬挂物体”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_dangling-objects)细节问题

$ git fsck  
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3  
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63  
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5  
...

例如，你可以检查其中一个悬空的提交。

$ gitk 7281251ddd --not --all

它听起来像：它说你想看到由悬空提交描述的提交历史，而不是所有现有分支和标记所描述的历史。这样，你就可以从丢失的提交中得到准确的历史记录。（注意它可能不仅仅是一个提交：我们只报告“行的末尾”是悬空的，但是可能会有一个全面而复杂的提交历史被删除。）

如果您决定要返回历史，您可以总是创建指向它的新引用，例如，一个新分支：

$ git branch recovered-branch 7281251ddd

其他类型的悬吊对象（斑点和树木）也是可能的，悬吊对象可能出现在其他情况下。

与他人共享发展

用Git拉动获取更新

在克隆一个存储库并提交一些自己的更改之后，您可能希望检查原始存储库中的更新，并将它们合并到自己的工作中。

我们已经看到[如何使远程跟踪分支保持最新](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_Updating-a-repository-With-git-fetch)与[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1))以及如何合并两个分支。因此，可以将原始存储库的主分支中的更改合并为：

$ git fetch  
$ git merge origin/master

然而，[“Git拉（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pull(1))命令提供了一步完成这一过程的方法：

$ git pull origin master

事实上，如果你有*硕士*签出，然后这个分支已经配置了*Git克隆*从源库的头分支中获取更改。通常你可以用一个简单的方法来完成上面的事情。

$ git pull

此命令将从远程分支获取到远程跟踪分支的更改。origin/\*，并将默认分支合并到当前分支中。

更一般地说，从远程跟踪分支创建的分支将默认地从该分支拉动。请参阅*branch.<name>.remote*和*branch.<name>.merge*选项在[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))和讨论*歌曲*选择权[“Git结账（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-checkout(1))来学习如何控制这些默认值。

除了保存你的击键之外，*GIT拉力*也可以帮助您生成默认的提交消息，记录从中提取的分支和存储库。

（但请注意，在A的情况下，不会创建这样的提交。[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards)相反，您的分支将被更新以指向来自上游分支的最新提交。

这个*GIT拉力*也可以给出命令*.*作为“远程”存储库，在这种情况下，它只是合并在当前存储库中的一个分支中；因此命令

$ git pull . branch  
$ git merge branch

大致相当

向项目提交补丁

如果你只是有一些改变，最简单的提交方法可能只是把它们作为电子邮件中的补丁发送给他们：

首先，使用[“Git格式修补程序（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-format-patch(1)); for example:

$ git format-patch origin

将在当前目录中生成一系列编号的文件，其中一个用于当前分支中的每个补丁，而不是在*原点/头部*.

*Git格式补丁*可以包括一个最初的“求职信”。你可以在三条虚线之后插入个别补丁的注释。*格式化补丁*在提交消息之后，但在补丁本身之前的位置。如果你使用*吉特笔记*追踪你的求职信资料，*Git格式补丁——注释*将以类似的方式包含提交的注释。

然后，您可以将这些导入到邮件客户端并手工发送它们。但是，如果您有很多要立即发送，您可能更喜欢使用[“Git发送电子邮件（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-send-email(1))脚本来自动化进程。首先咨询项目的邮件列表，以确定其提交补丁的要求。

将补丁导入项目

Git还提供了一种叫做[“Git AM（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-am(1))（AM代表“应用邮箱”），用于导入这样一系列电子邮件补丁。只需将所有包含消息的补丁依次保存到单个邮箱文件中，*修补程序*然后运行

$ git am -3 patches.mbox

Git将按顺序应用每个补丁；如果发现任何冲突，它将停止，并且可以按照“[解决合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_resolving-a-merge)“”（The）*- 3*选项告诉Git执行一个合并；如果您希望它只是中止并使您的树和索引保持不变，您可以省略该选项。

一旦用冲突解决的结果更新索引，而不是创建新的提交，只需运行

$ git am --continue

Git将为您创建提交并继续应用邮箱中的其余补丁。

最终的结果将是一系列的提交，一个在原始邮箱中的每个补丁，每个作者从包含每个补丁的消息中获取作者和提交日志消息。

公共GIT库

向项目提交更改的另一种方法是告诉该项目的维护者从您的存储库中使用这些更改。[“Git拉（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pull(1)). 在“[获得更新*GIT拉力*](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_getting-updates-With-git-pull)“我们把它描述为从“主要”存储库获取更新的方法，但在另一个方向上也一样。

如果您和维护人员都在同一台机器上拥有帐户，那么您可以直接从对方的存储库中提取更改；接受库URL作为参数的命令也将接受本地目录名称：

$ git clone /path/to/repository  
$ git pull /path/to/other/repository

或SSH URL：

$ git clone ssh://yourhost/~you/repository

对于很少开发人员的项目，或者为了同步几个私有的存储库，这可能是您所需要的。

然而，更常见的方法是维护一个单独的公共存储库（通常在不同的主机上），以便其他人从中提取更改。这通常是比较方便的，并且允许你从公开可见的工作中清理出正在进行的私人工作。

您将继续在您的个人存储库中进行日常工作，但定期将“个人”存储库中的更改推到您的公共存储库中，从而允许其他开发人员从该存储库中退出。因此，在有一个公共存储库的另一个开发者的情况下，变化的趋势看起来是这样的：

you push

your personal repo ------------------> your public repo

^ |

| |

| you pull | they pull

| |

| |

| they push V

their public repo <------------------- their repo

我们将在下面的章节中解释如何做到这一点。

建立公共存储库

假设您的个人存储库在目录中*~/PROJ*. 我们首先创建一个新的克隆库并告诉*Git守护进程*它是公开的：

$ git clone --bare ~/proj proj.git  
$ touch proj.git/git-daemon-export-ok

所得的目录投影.git包含一个“裸露”的Git存储库，它只是*Git*目录，没有任何文件签出它周围。

下一步，复制*GIT*到您计划托管公共存储库的服务器。你可以使用SCP、RSyc或任何最方便的东西。

通过GIT协议导出Git存储库

这是首选方法

如果其他人管理服务器，他们应该告诉你把目录放在什么目录中，以及什么*GIT：/ /*它将出现在URL上。然后你可以跳转到“[将更改推送到公共存储库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_pushing-changes-to-a-public-repository)“在下面

否则，你需要做的就是开始。[被称为“Git守护进程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-daemon(1))它会在9418号端口听。默认情况下，它将允许访问任何看起来像Git目录的目录，并包含神奇文件Git守护进程导出OK。传递一些目录路径*Git守护进程*参数将进一步限制对这些路径的出口。

你也可以运行*Git守护进程*作为ITED服务；请参见[被称为“Git守护进程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-daemon(1))详情页。（请参阅示例部分）

通过HTTP导出Git存储库

GIT协议提供了更好的性能和可靠性，但是在设置了Web服务器的主机上，HTTP导出可能更容易建立。

您需要做的是将新创建的裸Git存储库放置在Web服务器导出的目录中，并进行一些调整，以向Web客户端提供所需的额外信息：

$ mv proj.git /home/you/public\_html/proj.git  
$ cd proj.git  
$ git --bare update-server-info  
$ mv hooks/post-update.sample hooks/post-update

（为了解释最后两行，请参阅[称为“Git Update Server信息（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-update-server-info(1))和[被称为“GITOOKS（5）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#githooks(5))）

警告之路*GIT*. 任何其他人都应该能够克隆或拔出该URL，例如使用命令行：

$ git clone http://yourserver.com/~you/proj.git

（参见链接：Woto/Git服务器通过HTTP.HTML[安装Git服务器通过HTTP），使用WebDAV进行更复杂的设置，也允许推送HTTP。）

将更改推送到公共存储库

注意上面提到的两种技术[HTTP协议](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_exporting-via-http)或[吉特](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_exporting-via-git)允许其他维护者获取您的最新更改，但它们不允许写入访问，您需要更新公共存储库中的私有存储库中创建的最新更改。

最简单的方法就是使用。[“Git Press（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-push(1))和SSH；更新远程分支名*硕士*用你的分支的最新状态命名*硕士*运行

$ git push ssh://yourserver.com/~you/proj.git master:master

或者只是

$ git push ssh://yourserver.com/~you/proj.git master

和一样*吉特取出*，*Git推*如果这不会导致[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards)请参阅下面一节以了解处理此情况的细节。

注意，A的目标*推*通常是[光秃秃的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_bare_repository)储存库。您还可以推到具有已签出的工作树的存储库，但默认情况下，推送更新当前签出的分支以防止混淆。见收据的描述。[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))详情

和一样*吉特取出*还可以设置配置选项来保存键入；因此，例如：

$ git remote add public-repo ssh://yourserver.com/~you/proj.git

将下列内容添加到*.git/config*：

[remote "public-repo"]  
        url = yourserver.com:proj.git  
        fetch = +refs/heads/\*:refs/remotes/example/\*

让你做同样的推

$ git push public-repo master

看到的解释*remote.<name>.url*，*branch.<name>.remote*和*remote.<name>.push*选项在[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))详情

推送失败时该怎么办？

如果推动不会导致[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards)在远程分支中，它会出错，例如：

error: remote 'refs/heads/master' is not an ancestor of  
 local  'refs/heads/master'.  
 Maybe you are not up-to-date and need to pull first?  
error: failed to push to 'ssh://yourserver.com/~you/proj.git'

这可能发生，例如，如果你：

* use *git reset --hard* to remove already-published commits, or 使用*重置-硬*删除已发布的提交，或
* use *git commit --amend* to replace already-published commits (as in [the section called “Fixing a mistake by rewriting history”](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fixing-a-mistake-by-rewriting-history)), or 使用*修改最后一次提交*替换已发布的提交（如[“改写历史错误”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fixing-a-mistake-by-rewriting-history)), or
* use *git rebase* to rebase any already-published commits (as in [the section called “Keeping a patch series up to date using git rebase”](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_using-git-rebase)). 使用*GIT重碱*重新设置任何已发布的提交（如[被称为“使用Git ReBASE保持补丁系列最新”的部分](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_using-git-rebase)）

你可以强迫*Git推*在前面用加号表示分支名称之前执行更新：

$ git push ssh://yourserver.com/~you/proj.git +master

Note the addition of the*+*标志或者，您可以使用*-f*强制远程更新的标志，如：

$ git push -f ssh://yourserver.com/~you/proj.git master

通常，当公共存储库中的分支头被修改时，它被修改为指向它之前指向的提交的后代。在这种情况下迫使你打破这种惯例。（见[“重写历史问题”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_problems-With-rewriting-history)）

不过，这是一种常见的做法，需要一种简单的方法来发布正在进行中的补丁系列，并且这是一个可接受的折衷方案，只要你警告其他开发人员，这就是你打算如何管理分支。

当其他人有权推到同一个仓库时，也有可能以这样的方式失败。在这种情况下，正确的解决方案是在第一次更新您的工作之后重试推送：要么通过拉拽，要么通过取回，然后用重新基；[下一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_setting-up-a-shared-repository)和[“GITCVS迁移（7）”一节](https://tortoisegit.org/docs/tortoisegit/Git-Tutorial.html#gitcvs-migration(7))更多

建立共享存储库

另一种协作方式是使用类似于CVS中常用的模型，其中几个具有特殊权限的开发人员都推到单个共享存储库。见[“GITCVS迁移（7）”一节](https://tortoisegit.org/docs/tortoisegit/Git-Tutorial.html#gitcvs-migration(7))有关如何设置这方面的说明。

然而，尽管Git对共享库的支持没有错，但通常不推荐这种操作模式，这是因为GIT支持的协作模式——通过交换补丁和从公共存储库中提取——比中央共享库有很多优点：

* Git的快速导入和合并补丁的能力允许单个维护者即使在非常高的速率下也能处理传入的更改。当它变得太多时，*GIT拉力*为维护人员提供一种简单的方法，将该任务委托给其他维护人员，同时允许对传入的更改进行可选的审查。
* 由于每个开发人员的存储库都具有相同的项目历史记录副本，所以没有存储库是特殊的，对于另一个开发人员来说，通过相互协议来接管项目的维护是微不足道的，或者因为维护人员变得无响应或难以使用。
* 缺少一个“承诺者”的中心群体，意味着对于谁是“在”谁是“出去”的决定不那么必要。

允许Web浏览库

GITWebCGI脚本为用户提供了一个简单的方法来浏览项目的修订、文件内容和日志，而不必安装Git。可选地启用RSS/Atom提要和责备/注释细节等功能。

这个[被称为“Git SimaWeb（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-instaweb(1))命令提供了一种使用GITWeb浏览存储库的简单方法。使用StimeWeb时的默认服务器是LytTPD。

在Git源树中查看GITWeb/安装文件[被称为“GITWeb（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitweb(1))详细说明使用CGI或Perl能力服务器建立永久性安装的说明。

如何获得最少历史的Git存储库

一[浅克隆](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_shallow_clone)由于其截断的历史，当一个人只感兴趣于一个项目的最近历史，从上游获得完整的历史是昂贵的。

一[浅克隆](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_shallow_clone)通过指定[“Git克隆（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-clone(1))*--深度*开关深度可以稍后改变。[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1))*--深度*交换机或完整历史恢复*——非浅层*.

内部合并[浅克隆](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_shallow_clone)只要在最近的历史中合并基础就行。否则，这将像合并无关的历史，可能会导致巨大的冲突。这种限制可能使这样的存储库不适合用于基于合并的工作流。

实例

维护Linux子系统维护者的主题分支

这描述了Tony Luck如何使用Git作为Linux内核IA64架构的维护者的角色。

他使用两个公共部门：

* 一个“测试”树，其中最初放置补丁，以便当与其他正在进行的开发结合时，它们可以得到一些曝光。这棵树对安得烈来说，只要他愿意就可以拉到MM。
* 一个“释放”树，其中测试的补丁被移动以进行最终的健全检查，并作为一种交通工具将其上游发送给莱纳斯（通过向他发送一个“请拔”请求）。

他还使用了一组临时分支（“主题分支”），每个分支包含补丁的逻辑分组。

为此，首先通过克隆莱纳斯的公共树来创建你的工作树：

$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git work  
$ cd work

莱纳斯的树将存储在名为Orth/MeST的远程跟踪分支中，并且可以使用[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1))您可以使用其他公共树跟踪[被称为“Git远程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-remote(1))建立“远程”[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1))使它们保持最新；[被称为“仓库和分支”的部分](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_repositories-and-branches).

现在创建您要工作的分支；这些开始于当前/主分支的提示，并且应该被设置（使用*歌曲*选择权[“Git分支（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-branch(1))）默认情况下合并莱纳斯中的更改。

$ git branch --track test origin/master  
$ git branch --track release origin/master

这些可以很容易地保持最新的使用。[“Git拉（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pull(1)).

$ git checkout test && git pull  
$ git checkout release && git pull

重要注意事项！如果在这些分支中有任何本地更改，则此合并将在历史中创建提交对象（没有本地更改，Git将简单地进行“快进”合并）。许多人不喜欢在Linux历史上创建的“噪音”，所以你应该避免在*发布*分支，因为这些嘈杂的承诺将成为永久历史的一部分，当你要求莱纳斯从发布分支拉。

一些配置变量（参见[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))可以很容易地将两个分支都推到公共树上。（见[“建立公共存储库”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_setting-up-a-public-repository)）

$ cat >> .git/config <<EOF  
[remote "mytree"]  
        url =  master.kernel.org:/pub/scm/linux/kernel/git/aegl/linux.git  
        push = release  
        push = test  
EOF

然后，可以使用测试树和发布树。[“Git Press（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-push(1))：

$ git push mytree

或者只使用一个测试和发布分支：

$ git push mytree test

或

$ git push mytree release

现在，从社区应用一些补丁。想想一个分支的短名称，以保存这个补丁（或相关补丁组），并从莱纳斯分支最近的稳定标签创建一个新的分支。为你的分支选择一个稳定的基础：1）帮助你：通过避免无关的和可能被轻微测试的变化2）帮助将来使用的bug猎人*二分查找*发现问题

$ git checkout -b speed-up-spinlocks v2.6.35

现在应用补丁（ES），运行一些测试，并提交更改。如果修补程序是多部分序列，则应将每个应用程序作为单独的提交应用到该分支。

$ ... patch ... test  ... commit [ ... patch ... test ... commit ]\*

当你对这种变化的状态感到满意时，你可以将它合并到“测试”分支中，准备将其公开：

$ git checkout test && git merge speed-up-spinlocks

你不太可能在这里发生任何冲突…但如果你在这一步上花了一段时间，也从上游拉了新版本，你可能会。

过了一段时间，经过了足够的时间，测试完成后，你可以把同一个分支拉到*发布*树准备上岸。在这里你可以看到将每个补丁（或补丁系列）保持在自己的分支中的价值。这意味着补丁可以移动到*发布*树的任何顺序

$ git checkout release && git merge speed-up-spinlocks

过了一会儿，你会有很多分支，尽管你为他们挑选的名字很好，但你可能会忘记他们的身份，或者他们的身份。要得到特定分支中的更改的提醒，请使用：

$ git log linux..branchname | git shortlog

若要查看它是否已合并到测试或发布分支，请使用：

$ git log test..branchname

或

$ git log release..branchname

（如果这个分支尚未合并，您将看到一些日志条目。如果已合并，则将没有输出。

一旦一个补丁完成了大循环（从测试转移到释放，然后由莱纳斯拉，最后回到你的地方）。*起源/大师*分支），不再需要这种改变的分支。当输出为：

$ git log origin..branchname

是空的。此时，可以删除分支：

$ git branch -d branchname

有些更改是如此微不足道，因此不必创建单独的分支，然后合并到每个测试和发布分支中。对于这些更改，只需直接应用于*发布*分支，然后合并到*测试*分支

把你的工作推到*MyTrice*，你可以使用[“GIT请求拉动（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-request-pull(1))准备一个“请拉”请求消息发送给莱纳斯：

$ git push mytree  
$ git request-pull origin mytree release

这里有一些脚本进一步简化了这一切。

==== update script ====  
# Update a branch in my Git tree.  If the branch to be updated  
# is origin, then pull from kernel.org.  Otherwise merge  
# origin/master branch into test|release branch  
  
case "$1" in  
test|release)  
        git checkout $1 && git pull . origin  
        ;;  
origin)  
        before=$(git rev-parse refs/remotes/origin/master)  
        git fetch origin  
        after=$(git rev-parse refs/remotes/origin/master)  
        if [ $before != $after ]  
        then  
                git log $before..$after | git shortlog  
        fi  
        ;;  
\*)  
        echo "usage: $0 origin|test|release" 1>&2  
        exit 1  
        ;;  
esac

==== merge script ====  
# Merge a branch into either the test or release branch  
  
pname=$0  
  
usage()  
{  
        echo "usage: $pname branch test|release" 1>&2  
        exit 1  
}  
  
git show-ref -q --verify -- refs/heads/"$1" || {  
        echo "Can't see branch <$1>" 1>&2  
        usage  
}  
  
case "$2" in  
test|release)  
        if [ $(git log $2..$1 | wc -c) -eq 0 ]  
        then  
                echo $1 already merged into $2 1>&2  
                exit 1  
        fi  
        git checkout $2 && git pull . $1  
        ;;  
\*)  
        usage  
        ;;  
esac

==== status script ====  
# report on status of my ia64 Git tree  
  
gb=$(tput setab 2)  
rb=$(tput setab 1)  
restore=$(tput setab 9)  
  
if [ `git rev-list test..release | wc -c` -gt 0 ]  
then  
        echo $rb Warning: commits in release that are not in test $restore  
        git log test..release  
fi  
  
for branch in `git show-ref --heads | sed 's|^.\*/||'`  
do  
        if [ $branch = test -o $branch = release ]  
        then  
                continue  
        fi  
  
        echo -n $gb ======= $branch ====== $restore " "  
        status=  
        for ref in test release origin/master  
        do  
                if [ `git rev-list $ref..$branch | wc -c` -gt 0 ]  
                then  
                        status=$status${ref:0:1}  
                fi  
        done  
        case $status in  
        trl)  
                echo $rb Need to pull into test $restore  
                ;;  
        rl)  
                echo "In test"  
                ;;  
        l)  
                echo "Waiting for linus"  
                ;;  
        "")  
                echo $rb All done $restore  
                ;;  
        \*)  
                echo $rb "<$status>" $restore  
                ;;  
        esac  
        git log origin/master..$branch | git shortlog  
done

改写历史，维护补丁系列

通常提交只是添加到一个项目，从来没有带走或更换。Git是用这个假设来设计的，并且违反它会导致Git的合并机器（例如）做错误的事情。

然而，有一个情况下，它可能是有用的违反这个假设。

创建完美补丁系列

假设您是一个大型项目的贡献者，您想添加一个复杂的特性，并将其呈现给其他开发人员，使他们能够轻松地阅读您的更改，验证它们是否正确，并理解为什么您进行了每次更改。

如果你把所有的更改都显示为一个补丁（或提交），他们可能会发现这太难消化了。

如果你把它们放在你工作的整个历史上，完成错误、改正和死胡同，他们可能会不知所措。

所以理想通常是产生一系列补丁，这样：

1. 每个补丁可以按顺序应用。
2. 每个补丁包括一个单一的逻辑更改，以及一个解释更改的消息。
3. 没有补丁引入回归：在应用了该系列的任何初始部分之后，生成的项目仍然编译和工作，并且没有以前没有的bug。
4. 完整的系列会产生与你自己相同的结局（很可能是弥赛亚！）开发过程做到了。

我们将介绍一些可以帮助你做到这一点的工具，解释如何使用它们，然后解释一些由于你改写历史而可能出现的问题。

使用Git ReBASE保持补丁系列最新

假设创建分支*我的作品*在远程跟踪分支上*起源*并在它上面创建一些提交：

$ git checkout -b mywork origin  
$ vi file.txt  
$ git commit  
$ vi otherfile.txt  
$ git commit  
...

您没有执行到MyWork的合并，所以它只是一个简单的线性序列补丁的顶部。*起源*：

o--o--O <-- origin

\

a--b--c <-- mywork

在上游工程中做了一些更有趣的工作，*起源*具有先进性：

o--o--O--o--o--o <-- origin

\

a--b--c <-- mywork

在这一点上，你可以使用*拉*将您的更改合并回；结果将创建一个新的合并提交，如下所示：

o--o--O--o--o--o <-- origin

\ \

a--b--c--m <-- mywork

但是，如果您希望将MyWork中的历史保留为没有任何合并的简单系列提交，您可以选择使用。[被称为“Git ReBASE（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-rebase(1))：

$ git checkout mywork  
$ git rebase origin

这将从MyWork中删除每一个提交，暂时将它们保存为补丁（在目录中命名）。*GIT/ReBASE应用程序*更新MyWork来指向原点的最新版本，然后将每个保存的补丁应用到新的MyWork中。结果看起来像：

o--o--O--o--o--o <-- origin

\

a'--b'--c' <-- mywork

在这个过程中，它可能会发现冲突。在这种情况下，它将停止并允许您修复冲突；在修复冲突之后，使用*Git添加*用这些内容更新索引，然后代替运行*GIT提交*只是运行

$ git rebase --continue

Git将继续应用其余的修补程序。

在任何时候你都可以使用*--abort*选项中止此进程并将MyWork返回到开始ReBASE之前的状态：

$ git rebase --abort

如果需要在分支中重新排序或编辑多个提交，则使用起来可能更容易。*GIT Rebase-I*它允许您重新排序和挤压提交，并在重新排序时标记它们以进行个别编辑。见[所谓“使用互动基础”](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_interactive-rebase)详情请参阅[称为“从补丁系列中重新排序或选择”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_reordering-patch-series)替代方案

重写单个提交

我们看到了[“改写历史错误”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fixing-a-mistake-by-rewriting-history)您可以使用最新的提交替换

$ git commit --amend

它将用新的提交替换旧的提交，合并您的更改，给您一个编辑旧提交消息的机会。这对于修复您上次提交中的键入或调整不良阶段提交的补丁内容非常有用。

如果你需要修改你的历史中更深的承诺，你可以使用[交互式ReBASE*编辑*指令](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_interactive-rebase).

从补丁系列中重新排序或选择

有时你想在你的历史中更深入地编辑一个提交。一种方法是使用*Git格式补丁*创建一系列补丁，然后将状态重置到补丁之前：

$ git format-patch origin  
$ git reset --hard origin

然后在重新应用补丁之前，根据需要修改、重新排序或删除补丁。[“Git AM（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-am(1))：

$ git am \*.patch

使用交互式数据库

还可以使用交互式ReBASE编辑修补程序系列。这是一样的[重新使用补丁序列*格式化补丁*](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_reordering-patch-series)所以，使用你最喜欢的界面。

将当前的头重新定位在最后一个要保留的提交中。例如，如果您想重新排序最后5个提交，请使用：

$ git rebase -i HEAD~5

这将打开编辑器，列出要执行您的ReBASE的步骤列表。

pick deadbee The oneline of this commit  
pick fa1afe1 The oneline of the next commit  
...  
  
# Rebase c0ffeee..deadbee onto c0ffeee  
#  
# Commands:  
#  p, pick = use commit  
#  r, reword = use commit, but edit the commit message  
#  e, edit = use commit, but stop for amending  
#  s, squash = use commit, but meld into previous commit  
#  f, fixup = like "squash", but discard this commit's log message  
#  x, exec = run command (the rest of the line) using shell  
#  
# These lines can be re-ordered; they are executed from top to bottom.  
#  
# If you remove a line here THAT COMMIT WILL BE LOST.  
#  
# However, if you remove everything, the rebase will be aborted.  
#  
# Note that empty commits are commented out

正如注释中所解释的，您可以通过编辑列表来重新排序提交、将它们挤在一起、编辑提交消息等。一旦您满意，保存列表并关闭编辑器，ReBASE将开始。

ReBASE将停止在何处*挑选*已被取代*编辑*或者当列表中的一个步骤无法机械地解决冲突时，需要你的帮助。当您完成编辑和/或解决冲突时，您可以继续使用*继续，继续*. 如果你认为事情变得太多毛，你就可以随时摆脱困境。*Git ReBase-中止*. 即使在ReBASE完成之后，仍然可以通过使用[重新浮起](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_reflogs).

有关过程和附加技巧的更详细的讨论，请参见“交互模式”部分。[被称为“Git ReBASE（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-rebase(1)).

其他工具

还有许多其他工具，如STGIT，其存在是为了维护补丁系列。这些超出了本手册的范围。

改写历史问题

重写分支历史的首要问题与合并有关。假设有人取出你的分支并将其合并到他们的分支中，结果如下：

o--o--O--o--o--o <-- origin

\ \

t--t--t--m <-- their branch:

然后假设修改最后三个提交：

o--o--o <-- new head of origin

/

o--o--O--o--o--o <-- old head of origin

如果我们在一个存储库中一起检查所有这些历史，它将看起来像：

o--o--o <-- new head of origin

/

o--o--O--o--o--o <-- old head of origin

\ \

t--t--t--m <-- their branch:

Git不知道新的头是旧的头的更新版本，它对待这种情况完全一样，如果两个开发人员独立地对旧的和新的头并行工作。在这一点上，如果有人试图将新的头合并到他们的分支中，Git将尝试将两个（旧的和新的）开发线合并在一起，而不是试图用新的替换旧的。结果可能是出乎意料的。

您仍然可以选择发布其历史被重写的分支，并且对于其他人来说，能够获取这些分支以便检查或测试它们是有用的，但是它们不应该试图将这样的分支拉进它们自己的工作中。

对于支持适当合并的真正分布式开发，发布的分支永远不应该被重写。

为什么划分合并提交比分割线性历史更困难？

这个[“Git二等分（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-bisect(1))命令正确处理包含合并提交的历史。然而，当它发现的提交是合并提交时，用户可能需要比平常更努力地找出为什么提交导致了问题。

想象一下这个历史：

---Z---o---X---...---o---A---C---D

\ /

o---o---Y---...---o---B

假设在上面的开发线上，在Z中存在的一个函数的意义在提交X上被改变。从Z提交导致函数的实现和Z中存在的所有调用站点以及它们添加的新调用站点的改变是一致的。A.没有虫子

假设在较低的开发线上，有人在提交Y上为该函数添加了一个新的调用站点。从Z到B的提交都假定该函数的旧语义，调用方和被调用方彼此一致。B也没有bug。

进一步假设两条开发线在C中干净地合并，因此不需要冲突解决。

然而，C上的代码被破坏了，因为在较低的开发线上添加的调用方还没有被转换为在上一行开发中引入的新语义。如果你知道D是坏的，Z是好的，那[“Git二等分（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-bisect(1))标识C作为罪魁祸首，你会如何发现问题是由于语义上的这种变化？

当A的结果*二分查找*是一个非合并提交，您通常应该能够通过检查该提交来发现问题。开发人员可以通过将它们的更改分解成小的独立提交来实现这一点。然而，在上述情况下，这将无济于事，因为从任何单个提交的检查中，问题并不明显；相反，需要开发的全局视图。更糟糕的是，问题函数中语义的变化可能只是上一行的变化中的一小部分。

另一方面，如果不是在C中合并，则将Z到B之间的历史重新建立在A的顶部，那么您将得到线性历史：

---Z---o---X--...---o---A---o---o---Y\*--...---o---B\*--D\*

在Z和D之间划线会碰到一个罪犯犯下Y \*，并且理解为什么Y \*被破解可能会更容易。

部分原因是，许多有经验的GIT用户，即使在处理合并的重项目时，也会在发布之前将历史线性化为最新的上游版本。

高级分行管理

撷取个别分支

代替使用[被称为“Git远程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-remote(1))还可以选择只更新一个分支，并在任意名称下本地存储：

$ git fetch origin todo:my-todo-work

第一个论点，*起源*只告诉Git从最初克隆的存储库中获取。第二个参数告诉Git获取命名的分支。*托多*从远程存储库中，并将其本地存储在名称下*我的工作*.

也可以从其他存储库中获取分支；

$ git fetch git://example.com/proj.git master:example-master

将创建一个新的分支*样板大师*并把它命名为分支*硕士*从给定URL的存储库中。如果已经有一个名为“示例母版”的分支，它将尝试[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards)由示例.com的主分支给出的提交。更详细地说：

快攻快攻

在前面的示例中，当更新现有分支时，*吉特取出*检查以确保远程分支上的最近提交是分支更新前的最新提交的后代，然后更新分支的副本以指向新提交。Git称此过程为[快进](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_fast-forwards).

快进看起来像这样：

o--o--o--o <-- old head of the branch

\

o--o--o <-- new head of the branch

在某些情况下，新的头是可能的。**不**其实是老脑袋的后裔。例如，开发商可能意识到她犯了严重的错误，并决定回溯，导致这样的情况：

o--o--o--o--a--b <-- old head of the branch

\

o--o--o <-- new head of the branch

在这种情况下，*吉特取出*将失败，并打印出警告。

在这种情况下，您仍然可以强制Git更新到新的头，如下节所述。然而，请注意，在上述情况下，这可能意味着丢失标记的提交。*一*和*乙*，除非您已经创建了一个指向自己的引用。

强制Git获取进行非快进更新

如果Git FETCH失败，因为分支的新头不是旧头的后裔，则可以强制更新：

$ git fetch git://example.com/proj.git +master:refs/remotes/example/master

Note the addition of the*+*标志或者，您可以使用*-f*标志强制更新所有获取的分支，如：

$ git fetch -f origin

请注意，前面提到的示例/主控版本可能会丢失，正如我们在上一节中看到的。

配置远程跟踪分支

我们在上面看到origin只是引用最初克隆的存储库的快捷方式。此信息存储在Git配置变量中，您可以使用[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))：

$ git config -l  
core.repositoryformatversion=0  
core.filemode=true  
core.logallrefupdates=true  
remote.origin.url=git://git.kernel.org/pub/scm/git/git.git  
remote.origin.fetch=+refs/heads/\*:refs/remotes/origin/\*  
branch.master.remote=origin  
branch.master.merge=refs/heads/master

如果还有其他经常使用的存储库，则可以创建类似的配置选项来保存键入；例如，

$ git remote add example git://example.com/proj.git

将下列内容添加到*.git/config*：

[remote "example"]  
        url = git://example.com/proj.git  
        fetch = +refs/heads/\*:refs/remotes/example/\*

还要注意，上面的配置可以通过直接编辑文件来执行。*.git/config*代替使用[被称为“Git远程（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-remote(1)).

在配置远程之后，下面的三个命令将做同样的事情：

$ git fetch git://example.com/proj.git +refs/heads/\*:refs/remotes/example/\*  
$ git fetch example +refs/heads/\*:refs/remotes/example/\*  
$ git fetch example

见[被称为“Git配置（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-config(1))有关上面提到的配置选项的更多细节[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1))有关ReScript语法的更多细节。

GIT概念

Git是建立在少量简单而有力的想法之上的。虽然有可能在不理解它们的情况下完成事情，但是如果你做的话，你会发现Git更直观。

我们从最重要的开始，[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)以及[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index).

对象数据库

我们已经看到[“理解历史：提交”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_understanding-commits)所有提交都存储在一个40位的“对象名”下。事实上，表示项目历史所需的所有信息都存储在具有此类名称的对象中。在每种情况下，通过取对象内容的SHA-1哈希来计算名称。SHA-1哈希是一个密码散列函数。这意味着我们不可能找到两个名字相同的不同对象。这具有许多优点；

* Git可以通过比较名称来快速确定两个对象是否相同。
* 由于对象名称在每个存储库中以相同的方式计算，因此存储在两个存储库中的相同内容将总是以相同的名称存储。
* Git可以通过读取对象的名称仍然是其内容的SHA-1哈希来检测读取对象时的错误。

（见[称为“对象存储格式”的部分](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_object-details)对于对象格式化和SHA-1计算的细节。

有四种不同类型的对象：“BLB”、“树”、“提交”和“TAG”。

* 一[“BLB”对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_blob_object)is used to store file data.
* 一[“树”对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)将一个或多个“BLUB”对象绑定到目录结构中。此外，树对象可以引用其他树对象，从而创建目录层次结构。
* 一[“提交”对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)将目录层次结构联系起来[有向无环图](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_DAG)每一个提交都包含在提交时指定一个目录层次结构的恰好一棵树的对象名称。此外，提交指的是“父”提交对象，它描述了我们如何到达目录层次结构的历史。
* 一[“标签”对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag_object)象征性地识别和可以用来签署其他对象。它包含另一个对象的对象名称和类型，一个符号名称（当然！）并且，可选地，签名。

对象类型更为详细：

提交对象

“提交”对象链接树的物理状态，描述我们如何到达那里以及为什么。使用*--pretty=raw*选择权[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))或[被称为“Git log（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-log(1))检查您最喜欢的提交：

$ git show -s --pretty=raw 2be7fcb476  
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4  
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf  
parent 257a84d9d02e90447b149af58b271c19405edb6a  
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400  
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700  
  
    Fix misspelling of 'suppress' in docs  
  
    Signed-off-by: Junio C Hamano <gitster@pobox.com>

正如您所看到的，提交的定义是：

* 树：树对象的SHA-1名称（如下所定义），表示某个时间点上目录的内容。
* 父（S）：某些项目提交的SHA-1名称，表示项目历史上的前一个步骤。上面的示例有一个父级；合并提交可能不止一个。没有父级的提交被称为“root”提交，并表示项目的初始修订。每个项目必须至少有一个根。一个项目也可以有多个根，虽然这并不常见（或者是一个好主意）。
* 作者：负责变更的人的姓名及其日期。
* 提交者：实际创建提交的人的名字，以及完成的日期。这可能与作者不同，例如，如果作者是编写补丁的人，并将其发送给使用它来创建提交的人。
* 如何描述这句话

请注意，提交本身不包含任何关于实际更改的信息；所有更改都是通过比较提交提交的树的内容与与其父母相关联的树来计算的。特别是，Git不尝试显式地记录文件重命名，尽管它可以识别在更改路径中存在相同文件数据的情况下表示重命名。（例如，见*-M*选择权[被称为“Git DIFF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff(1))）

提交通常是由[“Git提交（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-commit(1))创建一个提交，其父级通常是当前的头，其树是从当前存储在索引中的内容获取的。

树对象

Ever-凡尔赛蒂[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))命令也可以用来检查树对象，但是[被称为“Git LS树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-ls-tree(1))会给你更多的细节：

$ git ls-tree fb3a8bdd0ce  
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore  
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap  
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING  
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation  
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN  
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL  
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile  
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README  
...

正如您所看到的，树对象包含一个条目列表，每个条目都有一个模式、对象类型、SHA-1名称和名称，按名称排序。它表示单个目录树的内容。

对象类型可以是一个BLUB，代表文件或另一棵树的内容，代表子目录的内容。由于树和块，像所有其他对象一样，由它们的内容的SHA-1散列命名，所以两个树具有相同的SHA-1名称，当且仅当它们的内容（包括递归地，所有子目录的内容）相同时。这允许Git快速地确定两个相关树对象之间的差异，因为它可以忽略具有相同对象名称的任何条目。

（注意：在子模块的存在下，树也可以作为条目提交。见[被称为“子模块”的部分](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_submodules)for documentation.)

注意，文件都有模式644或755：Git实际上只关注可执行位。

污点对象

你可以使用[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))检查BLB的内容；例如，在条目中使用BLB*复印*从上面的树：

$ git show 6ff87c4664  
  
 Note that the only valid version of the GPL as far as this project  
 is concerned is \_this\_ particular version of the license (ie v2, not  
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.  
...

一个“BLUB”对象只不过是二进制数据块。它不指任何东西，也不具有任何属性。

由于BLB完全由其数据定义，如果目录树中的两个文件（或在多个不同版本的存储库中）具有相同的内容，则它们将共享相同的BLB对象。该对象完全独立于目录树中的位置，并且重命名文件不会更改文件关联的对象。

注意，可以使用任何树或BLB对象进行检查。[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))with the <revision>:<path> syntax. This can sometimes be useful for browsing the contents of a tree that is not currently checked out.

信任

如果您从一个源接收到一个BUB的SHA-1名称，并且它的内容来自另一个（可能不可信）源，那么只要SHA-1名称一致，您仍然可以相信这些内容是正确的。这是因为SHA-1的设计使得发现不同的内容产生相同的哈希是不可行的。

类似地，您只需要信任顶级树对象的SHA-1名称来信任它引用的整个目录的内容，并且如果从可信源接收到SHA-1提交的名称，那么您就可以很容易地验证通过该提交的父母们可以获得的所有历史记录，以及这些提交所引用的树的所有内容。

因此，在系统中引入一些真正的信任，你唯一需要做的就是数字签名。*一*特别注释，其中包含顶级提交的名称。您的数字签名显示了您信任提交的其他人，提交的历史的不变性告诉其他人他们可以信任整个历史。

换句话说，你可以很容易地通过发送一封电子邮件告诉一个完整的存档文件，它告诉人们顶部提交的名称（SHA-1哈希），并用GPG/PGP之类的数字签名电子邮件。

为了帮助这一点，Git还提供了标签对象…

标记对象

标记对象包含对象、对象类型、标记名称、创建标记的人（“标签”）的名称以及可以包含签名的消息，如可以看到的那样。[“Git CAT文件（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-cat-file(1))：

$ git cat-file tag v1.5.0  
object 437b1b20df4b356c9342dac8d38849f24ef44f27  
type commit  
tag v1.5.0  
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000  
  
GIT 1.5.0  
-----BEGIN PGP SIGNATURE-----  
Version: GnuPG v1.4.6 (GNU/Linux)  
  
iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui  
nLE/L9aUXdWeTFPron96DLA=  
=2E+0  
-----END PGP SIGNATURE-----

见[被称为“Git标签（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-tag(1))命令学习如何创建和验证标记对象。（注意[被称为“Git标签（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-tag(1))也可以用来创建“轻量级标签”，这些标签根本不是标签对象，而只是名称的简单引用。*参考文献/标签/*）

Git如何高效地存储对象：打包文件

新创建的对象最初创建在以对象的SHA-1哈希命名的文件中（存储在*git／对象*）

不幸的是，一旦项目有很多对象，这个系统就变得效率低下。试试这个老项目：

$ git count-objects  
6930 objects, 47620 kilobytes

第一个数是保存在单个文件中的对象数。第二个是那些“松散”物体占用的空间量。

通过将这些松散的对象移动到一个“打包文件”中，可以节省空间并使Git更快，它以高效的压缩格式存储一组对象；可以在Link：Teal/Poad格式中找到HTML文件包格式的详细信息。

把松散的物体放入一个包中，只需运行Git RePACK：

$ git repack  
Counting objects: 6020, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (6020/6020), done.  
Writing objects: 100% (6020/6020), done.  
Total 6020 (delta 4070), reused 0 (delta 0)

这将创建一个单独的“包文件”.git /Objs/Pask/包含所有当前解压缩的对象。然后你可以运行

$ git prune

移除包中包含的任何“松散”对象。这也将删除任何未引用的对象（例如，当您使用时）可以创建这些对象。*Git复位*删除提交。您可以通过查看*git／对象*目录或运行

$ git count-objects  
0 objects, 0 kilobytes

尽管对象文件已经消失，但是引用这些对象的任何命令都会像以前一样工作。

这个[“Git GC（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-gc(1))命令为您执行打包、修剪和更多操作，因此通常是您需要的唯一高级命令。

悬空对象

这个[“Git FSCK（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fsck(1))指挥有时会抱怨悬空物体。它们不是问题。

悬挂对象最常见的原因是你重新建立了一个分支，或者你从其他人那里撤回了一个分支。[“改写历史，维护补丁系列”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_cleaning-up-history). 在这种情况下，原来分支的老首领仍然存在，正如它指向的一切一样。分支指针本身没有，因为您用另一个指针替换它。

还有其他引起悬挂物体的情况。例如，一个“悬空斑点”可能出现，因为你做了一个*Git添加*一个文件，但是，在你真正提交它并使它成为更大图片的一部分之前，你改变了那个文件中的其他东西，并提交了那个文件。**更新的**事情——原来添加的旧状态最终不会被任何提交或树所指向，所以它现在是一个悬空的BLB对象。

类似地，当“递归”合并策略运行时，发现有交叉交叉合并，因此有不止一个合并基（这是相当不寻常的，但它确实发生了），它会生成一个临时的中途树（或者甚至更多，如果你有很多交叉交叉合并和多于两个合并基）作为临时的内部合并基，而且，这些都是真实的对象，但是最终结果不会指向它们，所以它们最终在你的存储库中“悬空”。

一般来说，悬空物体并不是什么值得担心的东西。它们甚至可以是非常有用的：如果你把某物拧起来，悬挂的物体可以是你如何恢复你的老树（比如说，你做了一个重新定位，并意识到你真的不想去——你可以看看你悬挂的物体，并决定把你的头重新设置成一些悬垂状态）。

对于提交，您可以只使用：

$ gitk <dangling-commit-sha-goes-here> --not --all

这要求从给定的提交中获得所有历史，而不是从任何分支、标签或其他引用。如果你决定它是你想要的东西，你总是可以创建一个新的引用，例如，

$ git branch recovered-branch <dangling-commit-sha-goes-here>

对于斑点和树木，你不能做同样的事情，但是你仍然可以检查它们。你可以做

$ git show <dangling-blob/tree-sha-goes-here>

为了显示什么内容的博客是（或，对于一棵树，基本上是什么）*LS*因为这个目录是），这可能会让你知道操作是什么留下了悬空物体。

通常，悬垂的斑点和树木不是很有趣。它们几乎都是一个半路MeGeBASE的结果（BLUB经常会有冲突标记，从它的合并中，如果你已经用手固定了冲突的合并），或者仅仅是因为你中断了一个。*吉特取出*用^ C之类的东西，离开*一些*在对象数据库中的新对象，但只是悬空和无用。

不管怎样，一旦你确信你对任何悬空状态不感兴趣，你就可以修剪所有不可触及的对象：

$ git prune

他们会消失的。（你应该只跑git prune在静态存储库上，这有点像文件系统FSCK恢复：在安装文件系统时，您不想这样做。*吉特梅干*在并发访问存储库的情况下不会造成任何伤害，但您可能会收到混乱或可怕的消息。

从仓库腐败中恢复

通过设计，GIT小心地对待它信任的数据。然而，即使在Git本身没有bug的情况下，硬件或操作系统错误仍然可能损坏数据。

对这些问题的第一个防御是备份。可以使用克隆备份Git目录，或者只使用CP、TAR或任何其他备份机制。

作为最后的手段，你可以搜索损坏的物体并试图用手替换它们。在尝试这个过程之前备份你的存储库，以防你在这个过程中损坏更多。

我们假设问题是单个缺失或损坏的斑点，这有时是一个可解的问题。（找回丢失的树木，特别是提交）**许多的**更难）

在启动之前，验证是否存在腐败，并找出其存在的原因。[“Git FSCK（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fsck(1))这可能是时间消耗

假设输出看起来像这样：

$ git fsck --full --no-dangling  
broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8  
              to    blob 4b9458b3786228369c63936db65827de3cc06200  
missing blob 4b9458b3786228369c63936db65827de3cc06200

现在你知道BLUB 4B945 8B3丢失了，并且树2D9263C6指向它。如果你能找到一个丢失的BLB对象的拷贝，可能在其他的存储库中，你可以把它移动到*GIT/Objs/4B/945 8B3…*然后完成。假设你不能，你仍然可以检查指向它的树。[被称为“Git LS树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-ls-tree(1))可能会输出一些类似的东西：

$ git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8  
100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore  
100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap  
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING  
...  
100644 blob 4b9458b3786228369c63936db65827de3cc06200    myfile  
...

现在你知道丢失的BLB是一个文件名的数据myfile. 你也可以识别这个目录——假设它在*一些目录*. 如果幸运的话，丢失的副本可能与你在工作树中签出的副本相同。*某些目录/MyFrm*你可以测试这是否正确[称为“Git哈希对象（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-hash-object(1))：

$ git hash-object -w somedirectory/myfile

它将创建和存储BoLB对象，其中包含某个目录/MyFrm的内容，并输出该对象的SHA-1。如果你非常幸运，它可能是4B948B3686236369C6936Db682De3cc06200，在这种情况下，你猜对了，并且损坏是固定的！

否则，你需要更多的信息。如何判断文件丢失的版本？

最简单的方法是：

$ git log --raw --all --full-history -- somedirectory/myfile

因为你要的是原始输出，现在你会得到类似的东西。

commit abc  
Author:  
Date:  
...  
:100644 100644 4b9458b... newsha... M somedirectory/myfile  
  
  
commit xyz  
Author:  
Date:  
  
...  
:100644 100644 oldsha... 4b9458b... M somedirectory/myfile

这告诉你，紧接着的文件版本是“NeXHA”，而前一个版本是“OLDHA”。您还知道从OLDHA到4B945 8B的更改以及从4B945 8B到NeXHA的更改提交消息。

如果您已经提交了足够小的更改，您现在可以在重构状态4B945 8B之间的内容方面有一个好的解决方案。

如果你能做到这一点，你现在可以重新创建丢失的对象。

$ git hash-object -w <recreated-file>

你的储存库又好了！

（顺便说一下，你本来可以忽略的。*FSCK*开始做一个

$ git log --raw --all

只是寻找失踪物体的SHA（4B945 8B…）。这取决于你——Git**有**很多信息，只是缺少一个特定的BLB版本。

指数

索引是二进制文件（通常保存在*GIT/指数*）包含路径名称的排序列表，每个都具有权限和BULB对象的SHA-1；[被称为“Git LS文件（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-ls-files(1))可以告诉你索引的内容：

$ git ls-files --stage  
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0       .gitignore  
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0       .mailmap  
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0       COPYING  
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0       Documentation/.gitignore  
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0       Documentation/Makefile  
...  
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0       xdiff/xtypes.h  
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0       xdiff/xutils.c  
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0       xdiff/xutils.h

请注意，在旧文档中，可以看到称为“当前目录缓存”的索引，或者只看到“缓存”。它有三个重要特性：

1. 该索引包含生成单个（唯一确定的）树对象所需的所有信息。

例如，运行[“Git提交（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-commit(1))从索引生成树对象，将其存储在对象数据库中，并将其用作与新提交相关联的树对象。

1. 该索引实现了定义的树对象和工作树之间的快速比较。

它通过为每个条目存储一些附加数据（例如最后修改时间）来实现这一点。此数据不显示在上面，并且不存储在创建的树对象中，但它可以用来快速确定工作目录中的哪些文件与索引中存储的文件不同，从而节省了Git，以便从这些文件中读取所有数据以查找更改。

1. 它可以有效地表示关于不同树对象之间的合并冲突的信息，允许每个路径名与涉及的树的足够信息相关联，从而可以创建它们之间的三路合并。

我们看到了[“在合并过程中获得冲突解决帮助”的一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_conflict-resolution)在合并过程中，索引可以存储单个文件的多个版本（称为“阶段”）。中的第三列[被称为“Git LS文件（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-ls-files(1))上面的输出是阶段号，对于合并冲突的文件，将接受0以外的值。

因此，索引是一个临时的分级区域，它被一棵树填满，你正在工作。

如果你完全把索引吹走，你通常没有丢失任何信息，只要你有它所描述的树的名字。

子模块

大型项目通常由较小的、独立的模块组成。例如，一个嵌入式Linux发行版的源树将包含分布中的每一个软件，并进行一些局部修改；电影播放器可能需要建立一个特定的、已知的解压缩库的工作版本；几个独立的程序可能都共享相同的构建脚本。

随着集中修订控制系统，这往往是通过将每个模块包括在一个单一的仓库。开发人员可以检查所有模块或只需要他们工作的模块。它们甚至可以在一个提交中修改多个模块中的文件，同时移动周围的事物或更新API和翻译。

Git不允许部分签出，因此在Git中复制该方法将迫使开发人员保留他们不感兴趣的模块的本地副本。提交一个巨大的结帐会比你预期的慢，因为Git必须扫描每个目录进行更改。如果模块有大量的本地历史，克隆将永远持续下去。

另外，分布式版本控制系统可以更好地与外部资源集成。在集中式模型中，外部项目的单个任意快照从其自己的版本控制导出，然后导入到供应商分支的本地修订控制。所有的历史都是隐藏的。通过分布式版本控制，您可以克隆整个外部历史，更容易跟踪开发和重新合并本地更改。

Git的子模块支持允许存储库作为子目录包含外部项目的签出。子模块支持自己的身份；子模块支持只存储子模块库位置和提交ID，所以克隆包含项目（“SujProject”）的其他开发人员可以很容易地克隆同一版本中的所有子模块。超级项目的部分检查是可能的：你可以告诉Git克隆所有的子模块中的一个或全部。

这个[被称为“Git子模块（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-submodule(1))命令自Git 1.5.3可用。使用Git 1.5.2的用户可以查找存储库中的子模块提交并手动检查它们；早期版本根本不会识别子模块。

要了解子模块如何支持工作，创建四个示例库，这些库可以稍后用作子模块：

$ mkdir ~/git  
$ cd ~/git  
$ for i in a b c d  
do  
        mkdir $i  
        cd $i  
        git init  
        echo "module $i" > $i.txt  
        git add $i.txt  
        git commit -m "Initial commit, submodule $i"  
        cd ..  
done

现在创建超级项目并添加所有子模块：

$ mkdir super  
$ cd super  
$ git init  
$ for i in a b c d  
do  
        git submodule add ~/git/$i $i  
done

笔记

如果您计划发布超级项目，请不要在这里使用本地URL！

查看什么文件*GIT子模块*创建：

$ ls -a  
.  ..  .git  .gitmodules  a  b  c  d

这个*git submodule add <repo> <path>*命令做了两件事：

* 它将子模块从*<repo>*到给定的*<path>*在当前目录下，默认情况下检查主分支。
* 它将子模块的克隆路径添加到[被称为“GITMas模块（5）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitmodules(5))文件，并将该文件添加到索引中，准备提交。
* 它将子模块的当前提交ID添加到索引中，准备提交。

提交超级项目：

$ git commit -m "Add submodules a, b, c and d."

Now clone the superproject:

$ cd ..  
$ git clone super cloned  
$ cd cloned

子模块目录在那里，但它们是空的：

$ ls -a a  
.  ..  
$ git submodule status  
-d266b9873ad50488163457f025db7cdd9683d88b a  
-e81d457da15309b4fef4249aba9b50187999670d b  
-c1536a972b9affea0f16e0680ba87332dc059146 c  
-d96249ff5d57de5de093e6baff9e0aafa5276a74 d

笔记

上面所示的提交对象名称对您来说是不同的，但它们应该匹配库的头提交对象名称。您可以通过运行检查*GIT LS-remote…/A*.

下拉子模块是一个两步过程。首次运行*初始化子模块*将子模块存储库URL添加到*.git/config*：

$ git submodule init

现在使用*GIT子模块更新*克隆存储库并检查超级项目中指定的提交：

$ git submodule update  
$ cd a  
$ ls -a  
.  ..  .git  a.txt

一个主要区别*GIT子模块更新*和*Git子模块添加*那是*GIT子模块更新*检查一个特定的提交，而不是分支的提示。这就像检查标签：头部被分离，所以你没有在树枝上工作。

$ git branch  
\* (detached from d266b98)  
  master

如果您想在子模块中进行更改，并且具有独立的头，则应该创建或签出分支，进行更改，在子模块中发布更改，然后更新超级项目以引用新提交：

$ git checkout master

或

$ git checkout -b fix-up

然后

$ echo "adding a line again" >> a.txt  
$ git commit -a -m "Updated the submodule from within the superproject."  
$ git push  
$ cd ..  
$ git diff  
diff --git a/a b/a  
index d266b98..261dfac 160000  
--- a/a  
+++ b/a  
@@ -1 +1 @@  
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b  
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24  
$ git add a  
$ git commit -m "Updated submodule a."  
$ git push

你必须跑*GIT子模块更新*之后*GIT拉力*如果你也想更新子模块。

Pitfalls with submodules

在将更改发布到引用它的超级项目之前，始终发布子模块更改。如果忘记发布子模块更改，其他人将无法克隆存储库：

$ cd ~/git/super/a  
$ echo i added another line to this file >> a.txt  
$ git commit -a -m "doing it wrong this time"  
$ cd ..  
$ git add a  
$ git commit -m "Updated submodule a again."  
$ git push  
$ cd ~/git/cloned  
$ git pull  
$ git submodule update  
error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file(s) known to git.  
Did you forget to 'git add'?  
Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a'

在旧的Git版本中，很容易忘记在子模块中提交新的或修改过的文件，这会导致类似的问题，因为不推动子模块的更改。从Git 1.7.0开始*GIT状态*和*差异比较*在超级项目中，子模块在包含新的或修改的文件时被修改，以防止意外提交这样的状态。*差异比较*也会添加一个*-dirty*在生成补丁输出时使用的工作树侧或与*——子模块*option:

$ git diff  
diff --git a/sub b/sub  
--- a/sub  
+++ b/sub  
@@ -1 +1 @@  
-Subproject commit 3f356705649b5d566d97ff843cf193359229a453  
+Subproject commit 3f356705649b5d566d97ff843cf193359229a453-dirty  
$ git diff --submodule  
Submodule sub 3f35670..3f35670-dirty:

你也不应该将子模块中的分支重绕在任何超级项目中的记录之外。

跑步是不安全的*GIT子模块更新*如果在子模块中做出了和提交的更改，而不首先检查分支。它们将被默默地覆盖：

$ cat a.txt  
module a  
$ echo line added from private2 >> a.txt  
$ git commit -a -m "line added inside private2"  
$ cd ..  
$ git submodule update  
Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'  
$ cd a  
$ cat a.txt  
module a

笔记

这些变化在子模块的重新浮点中仍然可见。

如果在子模块工作树中有未提交的更改，*GIT子模块更新*不会覆盖它们。取而代之的是，你得到了一个警告：不能从肮脏的树枝上切换。

低水平GIT操作

许多高级命令最初使用低级别Git命令的较小内核实现为shell脚本。当用Git做不寻常的事情时，这些方法仍然是有用的，或者只是一种理解它内部工作的方式。

对象访问与操作

这个[“Git CAT文件（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-cat-file(1))命令可以显示任何对象的内容，尽管更高级别[“Git show（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-show(1))通常更有用

这个[“git提交树（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-commit-tree(1))命令允许构建具有任意父和树的提交。

可以创建一棵树[被称为“Git写树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-write-tree(1))它的数据可以通过[被称为“Git LS树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-ls-tree(1)). 两棵树可以比较[被称为“Git微分树（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff-tree(1)).

创建一个标签[被称为“Git MKTAG（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-mktag(1))，签名可以通过[称为“Git验证标记（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-verify-tag(1))尽管使用起来通常更简单[被称为“Git标签（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-tag(1))两者兼而有之

工作流

高级操作，如[“Git提交（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-commit(1))，[“Git结账（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-checkout(1))和[被称为“Git Read（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-reset(1))通过在工作树、索引和对象数据库之间移动数据来工作。Git提供低级操作，它们单独执行这些步骤中的每一个。

一般来说，所有Git操作都在索引文件上工作。一些操作工作**纯粹地**在索引文件上（显示索引的当前状态），但大多数操作在索引文件和数据库或工作目录之间移动数据。因此有四个主要的组合：

工作目录>索引

这个[称为“Git更新索引（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-update-index(1))命令使用工作目录中的信息更新索引。通常通过指定要更新的文件名来更新索引信息，如下所示：

$ git update-index filename

但是，为了避免文件名GROUBIN等常见错误，命令通常不会添加完全新的条目或删除旧条目，即它通常只更新现有的缓存条目。

要告诉Git，是的，您确实知道某些文件不再存在，或者应该添加新文件，您应该使用*--删除*和*添加*分别标记

注意！一*--删除*旗行*不*意味着后续文件名必须被删除：如果文件仍然存在于目录结构中，则索引将以其新状态进行更新，而不会被删除。唯一的事*--删除*方法是更新索引将考虑删除的文件是一个有效的东西，如果文件真的不存在了，它将相应地更新索引。

作为特殊情况，您也可以这样做*Git更新索引——刷新*这将刷新每个索引的“STAT”信息以匹配当前的STAT信息。它将*不*更新对象状态本身，它只更新用于快速测试对象是否仍然匹配其旧的后备存储对象的字段。

先前介绍的[“Git ADD（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-add(1))只是一个包装[称为“Git更新索引（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-update-index(1)).

索引-对象数据库

用程序将当前索引文件写入“树”对象。

$ git write-tree

这不需要任何选项——它只将当前索引写入到描述该状态的树对象集合中，并返回最终树的名称。您可以使用该树在任何时候通过另一个方向重新生成索引：

object database → index

从对象数据库中读取一个“树”文件，并使用它填充（并且覆盖-如果您的索引包含以后可能要还原的任何未保存状态，请不要这样做！）您当前的索引。正常操作就是

$ git read-tree <SHA-1 of tree>

您的索引文件现在相当于前面保存的树。然而，这只是你的*指数*文件：您的工作目录内容没有被修改。

索引>工作目录

通过“签出”文件从索引更新工作目录。这不是一个非常常见的操作，因为通常你只需要更新你的文件，而不是写到你的工作目录，你会告诉索引文件你工作目录中的变化。*Git更新索引*）

但是，如果你决定跳转到一个新版本，或者查看别人的版本，或者只是恢复一个前一棵树，你就可以用读取树填充索引文件，然后你需要检查结果。

$ git checkout-index filename

或者，如果要检查所有索引，请使用*-a*.

注意！*GIT检验指数*通常拒绝重写旧文件，所以如果您已经签出了旧版本的树，则需要使用*-f*flag (*之前*这个*-a*标志或文件名*力*结帐

最后，有一些不完全从一种表示转移到另一种表示的零散部分：

把它们捆在一起

提交一棵树你已经实例化了*Git写树*您将创建一个“提交”对象，该对象引用该树及其背后的历史——最值得注意的是“父”在历史记录之前提交的对象。

通常，“提交”有一个父级：在进行某种更改之前，树的前一个状态。然而，有时它可以有两个或多个父提交，在这种情况下，我们称之为“合并”，因为这样的提交将两个或多个先前提交的其他状态提交到一起（“合并”）。

换句话说，当一个“树”代表工作目录的一个特定目录状态时，一个“提交”代表时间的状态，并解释我们是如何到达那里的。

您可以通过提交一个描述提交时状态的树和一个父列表来创建提交对象：

$ git commit-tree <tree> -p <parent> [(-p <parent2>)...]

然后给出STDIN提交的原因（通过管道或文件重定向，或者只是在TTY上键入）。

*GIT提交树*将返回表示提交的对象的名称，并将其保存以供以后使用。通常，你会犯下一个新的错误*头部*状态，而Git不关心在哪里保存关于该状态的注释，实际上我们倾向于将结果写入到指向的文件中。*.git/HEAD*这样我们就可以看到最后一个承诺的状态了。

这是一张图片，说明了各个部分是如何组合在一起的：

                     commit-tree  
                      commit obj  
                       +----+  
                       |    |  
                       |    |  
                       V    V  
                    +-----------+  
                    | Object DB |  
                    |  Backing  |  
                    |   Store   |  
                    +-----------+  
                       ^  
           write-tree  |     |  
             tree obj  |     |  
                       |     |  read-tree  
                       |     |  tree obj  
                             V  
                    +-----------+  
                    |   Index   |  
                    |  "cache"  |  
                    +-----------+  
         update-index  ^  
             blob obj  |     |  
                       |     |  
    checkout-index -u  |     |  checkout-index  
             stat      |     |  blob obj  
                             V  
                    +-----------+  
                    |  Working  |  
                    | Directory |  
                    +-----------+

检查数据

您可以使用各种辅助工具检查对象数据库和索引中所表示的数据。对于每个对象，您可以使用[“Git CAT文件（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-cat-file(1))检查对象的详细信息：

$ git cat-file -t <objectname>

显示对象的类型，一旦您有了类型（通常在您找到对象的地方隐式），您就可以使用

$ git cat-file blob|tree|commit|tag <objectname>

显示其内容。注意！树具有二进制内容，因此有一个特殊的助手来显示内容，称为*Git LS树*将二进制内容转换成更易于阅读的形式。

查看“提交”对象尤其有启发性，因为这些对象往往很小，很容易解释。特别是，如果您遵循了具有顶级提交名称的约定*.git/HEAD*你可以做到

$ git cat-file commit HEAD

看看上面的承诺是什么。

合并多棵树

Git可以帮助您执行三路合并，这可以通过多次重复合并过程而用于多种方式合并。通常情况下，你只做一个三路合并（调和两行历史）并提交结果，但是如果你愿意，你可以一次合并几个分支。

要执行三路合并，首先从两个要合并的提交开始，找到它们最接近的公共父级（第三提交），并比较对应于这三个提交的树。

若要获得合并的“基础”，请查找两个提交的公共父级：

$ git merge-base <commit1> <commit2>

这将打印基于它们的提交的名称。现在，您应该查找这些提交的树对象，您可以轻松地使用这些对象。

$ git cat-file commit <commitname> | head -1

因为树对象信息始终是提交对象中的第一行。

一旦你知道你要合并的三棵树（一棵原始树），又名普通树，以及两棵“结果”树，即你想合并的分支，你就可以在索引中进行“合并”读取。如果它必须扔掉旧的索引内容，这将是一个抱怨，所以你应该确保你已经提交了这些内容——事实上，你通常会对你的最后一个提交进行一个合并（这应该匹配你当前索引中的内容）。

若要合并，请执行

$ git read-tree -m -u <origtree> <yourtree> <targettree>

它将直接为索引文件中的所有微不足道的合并操作进行操作，您可以将结果写入*Git写树*.

合并多棵树，继续

遗憾的是，许多合并并不微不足道。如果有添加、移动或删除的文件，或者如果两个分支都修改了同一个文件，则会留下一个索引树，其中包含“合并条目”。这样的索引树可以*不是*将其写到树对象中，在编写结果之前，必须使用其他工具解决任何此类合并冲突。

您可以检查这些索引状态。*git ls文件——未合并*command. An example:

$ git read-tree -m $orig HEAD $target  
$ git ls-files --unmerged  
100644 263414f423d0e4d70dae8fe53fa34614ff3e2860 1       hello.c  
100644 06fa6a24256dc7e560efa5687fa84b51f0263c3a 2       hello.c  
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello.c

每一行*git ls文件——未合并*输出以BLB模式位开始，Bulb SHA-1，*舞台编号*, and the filename. The*舞台编号*Git的方法是说它来自哪个树：阶段1对应于*奥利格*树，第2阶段到*头部*树，第3阶段*美元目标*树

早些时候，我们说琐碎的合并是在内部完成的。*git读树-m*. 例如，如果文件没有从*奥利格*到*头部*或*美元目标*，或者如果文件从*奥利格*到*头部*和*奥利格*到*美元目标*同样的，显然最后的结果是什么*头部*. 上面的例子显示的是文件*你好，C*被改变*奥利格*到*头部*和*奥利格*到*美元目标*以不同的方式。你可以通过运行你喜欢的三路合并程序来解决这个问题。*diff3*，*合并*或者Git自己的合并文件，在BUB对象上，从这三个阶段自己，像这样：

$ git cat-file blob 263414f... >hello.c~1  
$ git cat-file blob 06fa6a2... >hello.c~2  
$ git cat-file blob cc44c73... >hello.c~3  
$ git merge-file hello.c~2 hello.c~1 hello.c~3

这会导致合并结果*你好，C~ 2*文件，以及冲突标记，如果存在冲突。在验证合并结果是有意义的之后，您可以告诉Git这个文件的最终合并结果是：

$ mv -f hello.c~2 hello.c  
$ git update-index hello.c

当路径处于“未合并”状态时，运行*Git更新索引*因为该路径告诉Git标记路径被解析。

以上是对Git合并在最低级别的描述，以帮助您理解引擎盖下概念上发生的事情。实际上，没有人，甚至Git自己也跑。*Git CAT文件*为此三次。有一个*Git合并索引*将阶段提取到临时文件并在其上调用“合并”脚本的程序：

$ git merge-index git-merge-one-file hello.c

这就是更高的层次*Git合并-S解决方案*实现

黑客攻击

本章涵盖了GIT实现的内部细节，可能只是GIT开发者需要理解的。

对象存储格式

所有对象都有静态确定的“类型”，它标识对象的格式（即它是如何使用的，以及它如何引用其他对象）。目前有四种不同的对象类型：“BLB”、“树”、“提交”和“TAG”。

不管对象类型如何，所有对象都具有以下特性：它们都用ZLIB放空，并且有一个头，不仅指定它们的类型，而且还提供对象中数据的大小信息。值得注意的是，用于命名对象的SHA-1哈希是原始数据的哈希加上这个头，所以*沙伊姆文件*与对象名称不匹配*文件*.

结果，对象的一般一致性总是可以独立于对象的内容或类型来测试：所有对象都可以通过验证（a）它们的哈希匹配文件的内容来验证，（b）对象成功地膨胀成形成序列的字节流。*<ascii type without space> + <space> + <ascii decimal size> + <byte\0> + <binary object data>*.

结构化对象可以进一步具有它们的结构和与其他对象的连通性。这通常是用*GIT FSCK*程序，它生成所有对象的完整依赖图，并验证它们的内部一致性（除了通过哈希验证它们的表面一致性）。

Git源代码的鸟瞰图

对于新开发人员来说，通过Git的源代码找到它们并不总是容易的。本节为您提供了一些指导，以显示从何处开始。

启动的好地方是初始提交的内容，包括：

$ git checkout e83c5163

最初的修订为Git今天所拥有的一切奠定了基础，但它足够小，可以一坐就可以阅读。

注意，自从修订以来，术语已经发生了变化。例如，该版本中的自述使用“变更集”这个词来描述我们现在所说的[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object).

此外，我们不再称它为“缓存”，而是“index”；然而，该文件仍然被称为“Cache”。*高速缓存*. 备注：现在没有太多理由改变它，尤其是因为它没有好的单名，因为它基本上是*这个*头文件，包括*全部*Git的C源

如果你掌握了初始提交中的想法，你应该检查一个更新的版本并略读。*高速缓存*，*对象H*和*提交*.

在早期，Git（在UNIX的传统中）是一组非常简单的程序，它们在脚本中使用，将一个输出连接到另一个脚本中。这对最初的开发很有帮助，因为它更容易测试新事物。然而，最近这些部件中的许多已经成为建设者，并且一些核心已经被“LISBITION”，即放入LBGIT。A用于性能、可移植性的原因，并避免代码复制。

现在，您知道索引是什么（并找到相应的数据结构）。*高速缓存*），并且只有两个对象类型（BLB、树、提交和标签）继承了它们的共同结构。*结构对象*，这是他们的第一个成员（因此，你可以铸造）。*（结构对象\*）提交*实现*相同的*作为*和提交->对象*，即获取对象名称和标志。

现在是一个很好的休息点让这个信息下降。

下一步：熟悉对象命名。阅读[“命名提交”一节](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_naming-commits). 有很多方法来命名一个对象（而不仅仅是修改！）所有这些都是在*S.*. 只需快速查看函数*GETHYSHA1（）*. 许多特殊的处理是通过函数来完成的。*GET-Sya1Basic（）*或者喜欢

这只是为了让你进入Git最易受歧视的部分：修订版沃克。

基本上，最初的版本*GIT日志*是shell脚本：

$ git-rev-list --pretty $(git-rev-parse --default HEAD "$@") | \  
        LESS=-S ${PAGER:-less}

这意味着什么？

*GIT rev-list*是修订版沃克的原始版本，*总是*打印STDUT的修订列表。它仍然是功能性的，并且需要，因为大多数新的Git命令都是从脚本开始使用的。*GIT rev-list*.

*Git Rev解析*不再重要了；它只是用来过滤与脚本调用的不同管道命令相关的选项。

大部分什么*GIT rev-list*确实包含在*C修订版*和*修订版*. 它封装了一个名为*新闻*，它控制着如何以及如何修改，以及更多。

原来的工作*Git Rev解析*现在由函数*setup\_revisions()*，它解析修订Walk的修订和常见命令行选项。此信息存储在结构中。*新闻*以后消费。您可以在调用后执行自己的命令行选项解析*setup\_revisions()*. 之后，你必须打电话*准备好了吗？*对于初始化，然后可以用函数逐个提交*get\_revision()*.

如果您感兴趣的是修订过程的更多细节，请看一下第一个实现*CMDYLY（）*呼叫*GIT显示V1.3.0～155 ^ 2～4*向下滚动到该函数（注意您不再需要调用）*StUpUpPar（）*直接）

如今，*GIT日志*是一个建设者，这意味着它是*包含的*在命令中*吉特*. 建站的源侧是

* 称为函数*cmd\_<bla>*，通常定义在*builtin/<bla.c>*（注意较早版本的Git使用它*builtin-<bla>.c*相反），并宣布*建筑业*.
* 中的一个条目*命令[]*数组在*C*和
* 一个条目*构建对象*在*生成文件*.

有时，在一个源文件中包含不止一个构建体。例如，*CMDYWORUNTEDD（）*和*CMDYLY（）*两者并存*构建/日志*，因为它们共享相当多的代码。在这种情况下，命令是*不*命名为*C*他们所居住的文件必须列入*建筑材料*在*生成文件*.

*GIT日志*在C中看起来比在原始脚本中更复杂，但是这允许更大的灵活性和性能。

这里再次停顿是一个很好的观点。

第三课是：学习代码。的确，这是了解GIT组织的最好方法（了解基本概念之后）。

所以，想想你感兴趣的东西，比如，“我怎么才能知道一个斑点，只要知道它的名字？”第一步是找到一个你可以做的Git命令。在这个例子中，它要么是*吉特秀*或*Git CAT文件*.

为了清晰起见，让我们留下来*Git CAT文件*，因为它

* 是水管，
* 甚至在最初的提交过程中（它只不过是经过了20次修订）而已。*cat-file.c*，重命名为*内置/CAT文件*当建造一个建筑，然后看到少于10个版本）。

所以，看看*内置/CAT文件*寻找*cmd\_cat\_file()*看看它做了什么

        git\_config(git\_default\_config);  
        if (argc != 3)  
                usage("git cat-file [-t|-s|-e|-p|<type>] <sha1>");  
        if (get\_sha1(argv[2], sha1))  
                die("Not a valid object name %s", argv[2]);

让我们跳过显而易见的细节，这里唯一有趣的部分就是*GETHYSHA1（）*. 它试图解释*AGV〔2〕*作为对象名称，如果它引用当前存储库中存在的对象，则将生成的SHA-1写入变量中。*沙哈*.

这里有两件事很有趣：

* *GETHYSHA1（）*返回0*成功*. 这可能会让一些新的GIT黑客吃惊，但在UNIX中，在不同错误的情况下返回不同的负数有着悠久的传统，而成功则有0。
* 变量*沙哈*在函数签名中*GETHYSHA1（）*是*无符号字符*，但实际上是一个指针*无符号字符〔20〕*. 此变量将包含给定提交的160位SHA-1。注意，每当SHA-1被传递为*无符号字符*它是二进制表示，而不是十六进制字符中的ASCII表示，它被传递为*char \**.

您将在代码中看到这两个东西。

现在，为肉：

        case 0:  
                buf = read\_object\_with\_reference(sha1, argv[1], &size, NULL);

这就是你如何阅读一个博客（实际上，不仅是一个博客，但任何类型的对象）。了解函数*Read对象的引用*实际工作，找到它的源代码（类似的东西）*GIT-GRIPRead对象Obj\*[GRP:：[AZ] ]*在Git存储库中，并读取源代码。

要知道如何使用结果，请继续阅读*cmd\_cat\_file()*：

        write\_or\_die(1, buf, size);

有时候，你不知道去哪里寻找一个特色。在许多这样的情况下，有助于搜索输出。*GIT日志*然后*吉特秀*相应的提交

Example: If you know that there was some test case for*Git束*但不记得在哪里（是的，你）*能够GIT-GRIP束T/*，但这并不能说明这一点！）：

$ git log --no-merges t/

在寻呼机中（*较少的*只需搜索“束”，返回几行，并看到它在提交1844 9AB0…现在只需复制该对象名称，并将其粘贴到命令行中

$ git show 18449ab0

沃伊拉

另一个例子：找出要做什么，以使一些脚本建立：

$ git log --no-merges --diff-filter=A builtin/\*.c

你知道，Git实际上是找出Git本身来源的最好工具！

GIT词汇表

alternate object database 替代对象数据库

通过交替机制，A[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)可以继承它的一部分[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)从另一个对象数据库，它被称为“备用”。

bare repository 裸仓库

裸仓库通常是适当命名的。[目录](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_directory)用一个*Git*后缀，在修订控制下没有任何文件的本地签出副本。也就是说，通常隐藏在Git中的所有管理和控制文件都存在。*Git*子目录直接存在于*知识库*目录，而不存在其他文件并签出。通常，公共存储库的发布者可以使用裸露的存储库。

blob object BLOB对象

非类型化的[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)E.G.The contents of a file.

Branch 分支

“分支”是一条积极的发展路线。最近的[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)在分支上称为分支的尖端。分支的顶端由分支引用。[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)随着分支的进一步发展，它向前移动。单一Git[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)可以跟踪任意数量的分支，但是你的[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)只与其中一个（“当前”或“签出”分支）关联，以及[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)指向那个分支

Cache 隐藏物

过时的：[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index).

Chain 链

对象列表，其中每个[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)在列表中包含对其继承人的引用（例如，A继承人）。[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)可能是其中之一[父母](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent)）

Changeset 变更集

BITKEST/CVSPS发言[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)“。由于Git不存储变化，而是状态，所以使用Git的“变更集”这个术语真的没有意义。

Checkout 结算

更新全部或部分的动作[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)用一个[树对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)或[斑点](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_blob_object)从[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)并更新[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index)和[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)如果整个工作树都指向了一个新的[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch).

cherry-picking 樱桃采摘

进入[单片机](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SCM)术语“Cyt Poice”意味着从一系列变化中选择一个变化子集（通常提交），并将它们记录为不同的代码库之上的一系列新的变化。在Git，这是由“Git樱桃挑选”命令来执行由现有的介绍所带来的变化。[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)并根据电流的尖端记录[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)作为一个新的提交

Clean 清洁

一[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)是干净的，如果它对应于[修订](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_revision)由当前引用[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head). 也见“[肮脏的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_dirty)“

Commit 提交

作为名词：Git历史中的单个点；项目的整个历史被表示为一组相互关联的提交。“提交”一词经常被GIT用于其他修订控制系统使用“修订”或“版本”的地方。也可用作短手[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object).

作为动词：通过创建表示当前状态的新提交来将项目状态的新快照存储在Git历史中的动作。[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index)前进[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)指向新提交

commit object 提交对象

安[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)包含特定信息的[修订](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_revision)，如[父母](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent)、提交人、作者、日期和[树对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)对应于顶部[目录](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_directory)of the stored revision.

commit-ish (also committish) 犯（也犯了罪）

一[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)或[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)这可以递归地提交给提交对象。下面是所有提交：提交对象，一个[标记对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag_object)指向一个提交对象，一个指向指向一个提交对象的标签对象的标签对象，等等。

core Git 核心Git

Git的基本数据结构和实用程序。只公开有限的源代码管理工具。

DAG 达格

有向无环图这个[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)形成有向无环图，因为它们有父（有向），并且提交对象的图是非循环的（没有）。[链](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_chain)以相同的方式开始和结束[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)）

dangling object 悬挂对象

安[不可达对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_unreachable_object)哪个不是[可达成的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_reachable)即使是从其他不可触及的物体上，悬垂物体也没有任何参考。[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)在[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository).

detached HEAD 分离头

通常情况下[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)stores the name of a[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)操作在历史磁头上的命令表示历史，导致头指向的分支尖端。然而，Git也允许你[退房](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_checkout)任意的[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)这不一定是任何分支的尖端。头部处于这种状态被称为“分离”。

注意在当前分支的历史上运行的命令（例如）。*GIT提交*建立一个新的历史之上）仍然工作，而头部分离。他们在更新历史的顶端更新头到点，而不影响任何分支。更新或查询信息的命令*关于*当前分支（例如）*Git分支--上游设置*设置当前分支的远程跟踪分支与之显然是不起作用的，因为在这个状态中没有（真实的）当前分支要询问。

Directory 目录

你用“LS”得到的列表：

Dirty 肮脏的

一[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)被称为“脏”，如果它包含了尚未修改的[坚信的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)迎合潮流[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch).

evil merge 坏的合并

邪恶的合并是[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)介绍没有出现在任何情况下的更改。[父母](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent).

fast-forward 快进

快进是一种特殊类型的快进。[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)你在哪里[修订](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_revision)你正在“融合”另一个[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)这些变化恰好是你所拥有的后代。在这样的情况下，你不会创造新的。[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)但是，只是更新他的修订。这将经常发生在[远程跟踪分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_remote_tracking_branch)遥远的[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository).

Fetch 取来

取A[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)获取分支机构的方法[主任裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head_ref)来自远方[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)找出本地对象缺少哪些对象[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)也要得到它们。也见[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1)).

file system 文件系统

莱纳斯ToValdDS最初设计的Git是一个用户空间文件系统，即保存文件和目录的基础设施。这确保了GIT的效率和速度。

GIT档案馆

同义词[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)（为拱人）

gitfile

普通文件*Git*在指向真实存储库目录的工作树的根上。

grafts 移植物

嫁接使两个不同的开发线通过记录虚假的祖先信息而结合在一起。这样你就可以让Git假装一套[父母](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent)一[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)与创建提交时记录的内容不同。通过配置*.git/info/grafts*文件

请注意，移植机制是过时的，并且可能导致在存储库之间传输对象的问题；[“Git替换（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-replace(1))一个更灵活、更健壮的系统来做同样的事情。

hash 搞砸

在Git的上下文中，同义词[对象名称](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_name).

head

一[命名参考文献](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)到[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)在尖端[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch). 磁头存储在文件中*$GITYDIRE/REFS/头/*目录，除非使用包装参考。（见[被称为“Git PACK REFS（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pack-refs(1))）

HEAD 头部

电流[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch). 更详细地说：你的[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)通常是由头部所指的树的状态导出的。头是对其中一个的引用[头](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)在您的存储库中，除了使用[分离头](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_detached_HEAD)在这种情况下，它直接引用任意提交。

head ref 主任裁判

同义词[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head).

hook 挂钩

在正常执行几个Git命令时，调用可选脚本，允许开发人员添加功能或检查。通常，钩子允许命令进行预验证和潜在中止，并允许在操作完成后发布通知。钩子脚本在*\*Git Uh Dir/Hooks\**目录，并通过简单删除*样品*文件名后缀。在早期版本的Git中，您必须使它们成为可执行的。

index 指数

具有STAT信息的文件的集合，其内容被存储为对象。该索引是您的存储版本。[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree). 事实上，它也可以包含一个第二个甚至第三个版本的工作树。[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge).

index entry 索引条目

关于特定文件的信息，存储在[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index). 索引条目可以被合并，如果[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)已启动，但尚未完成（即，如果索引包含该文件的多个版本）。

master

默认开发[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch). 每当你创建一个Git[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)创建一个名为“主”的分支，并成为活动分支。在大多数情况下，这包含局部开发，尽管这纯粹是按照惯例，而不是必需的。

master 合并

作为动词：带来另一个内容[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)（可能来自外部）[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)进入当前分支。在合并的分支来自不同的存储库的情况下，这是通过第一个来完成的。[取来](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_fetch)远程分支然后将结果合并到当前分支中。调用和合并操作的组合称为[拉](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_pull). 合并是由一个自动过程执行的，该过程识别分支自分支以来所做的更改，然后将所有这些更改应用到一起。在改变冲突的情况下，可能需要人工干预来完成合并。

作为名词：除非是名词 [fast-forward](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_fast_forward)成功的合并导致新的创建。 [commit](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)表示合并的结果，并具有[parents](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent)合并的技巧[branches](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch). 这个提交被称为“合并提交”，或者有时只是一个“合并”。

object 目标

Git中的存储单元。它是由[SHA-1](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SHA1)其内容。因此，对象不能被改变。

object database 对象数据库

存储一组“对象”和一个个体[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)由其识别[对象名称](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_name). 对象通常生活在*$GITYDIR/Objs/*.

object identifier 对象标识符

同义词[对象名称](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_name).

object name 对象名称

AN的唯一标识符[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object). 对象名称通常由40个字符的十六进制字符串表示。也俗称为[SHA-1](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SHA1).

object type 对象类型

标识符之一[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)“，”[树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)“，”[标签](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag_object)“或”[斑点](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_blob_object)" describing the type of an[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object).

Octopus 章鱼

到[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)不止两个[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch).

origin

默认上游[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository). 大多数项目至少有一个上游项目跟踪。默认情况下*起源*用于此目的。新的上游更新将被引入[远程跟踪分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_remote_tracking_branch)上游分支的命名原点/名称，您可以使用*分支机构*.

Pack 包装

一组被压缩成一个文件的对象（以节省空间或有效地传输它们）。

pack index

中的对象的标识符和其他信息的列表。[包装](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_pack)，以帮助有效地访问包的内容。

Pathspec 路径规格

用于限制Git命令中的路径的模式。

在“GIT-LS文件”、“Git LS树”、“Git添加”、“Git GRIP”、“Git DIFF”、“Git Debug”命令行以及许多其他操作限制了操作范围到树或工作树的某些子集上使用了PosiScript。请参阅每个命令的文档，以确定路径是否与当前目录或TopPrVE有关。PaseScript语法如下：

* 任何路径匹配自身
* 最后一个斜杠的路径标志表示一个目录前缀。该PaseScript的范围仅限于该子树。
* PaseScript的其余部分是路径名的其余部分的模式。相对于目录前缀的路径将使用fnMatl（3）与该模式匹配；*\**和*？可以*匹配目录分隔符

例如，文档/ \*.jpg将匹配文档子树中的所有.jpg文件，包括文档/查证1 /图1 .jpg。

从结肠开始的路径规范*：*具有特殊的意义。在简短的形式，领先的结肠*：*接着是零个或多个“魔法签名”字母（可选地由另一个冒号终止）。*：*），其余的是与路径匹配的模式。“魔法签名”由ASCII符号组成，这些符号既不是字母数字，也不是GLUB，正则表达式是特殊字符，也不是冒号。如果该模式以不属于“魔术签名”符号集而不是冒号的字符开头，则可以省略终止“魔术签名”的可选冒号。

在长形，领先的结肠*：*后面是一个开括号*（*一个逗号分隔的零或多个“魔术单词”列表，以及一个近旁括号。*）*其余的是与路径匹配的模式。

只有一个冒号的PosiScript意味着“没有路径规范”。此格式不应与其他路径相结合。

top

神奇的词*上衣*（魔术签名：*/*使模式与工作树的根匹配，即使在从子目录内运行命令时也是如此。

literal

模式中的通配符，如*\**或*？*被视为文字字符

icase

不区分大小写的匹配

glob

Git将该模式视为一个适合于消费的shell GLUB，由FnMead（3）与FnMyPATAODNEY标志：模式中的通配符将不匹配路径名中的//。例如，“文档/\*.html”匹配“文档/git .html”，而不是“文档/pPC/pc.html”或“工具/Prf/Debug /Prf.html”。

两个连续星号（“*\*\**“与完全路径名匹配的模式可能具有特殊含义：

* 领先的“*\*\**在所有目录后面都有斜线意味着匹配。例如：“*\*/FoO*“匹配文件或目录”*福*“在任何地方，同模式一样”*福*“。”*\*/Foo/bar*“匹配文件或目录”*酒吧*“直接在目录下的任何地方”*福*“
* 尾随*/\*\**“把里面的东西都配好。例如：“*abc/\*\**“匹配”目录“ABC”中的所有文件，相对于*吉蒂格诺*文件，具有无限深度
* 一个斜杠，然后是两个连续的星号，然后斜线匹配零或更多的目录。例如：“*a/\*\*/b*“火柴”*a/b*“，”*a/x/b*“，”*a/x/y/b*“等等
* 其他连续星号被认为是无效的。

GLUB魔法与文字魔法不相容。

Exclude 排除

在路径匹配任何非排除的PoScript之后，它将运行所有排除的PosiScript（魔术签名：*！*）如果匹配，则忽略该路径。

Parent 父母

一[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)包含一个（可能是空的）开发前线中逻辑前辈的列表，即它的父母。

pickaxe

术语[鹤嘴锄](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_pickaxe)是指帮助选择添加或删除给定文本字符串的更改的DeCype例程的选项。与*——鹤嘴锄*选项，它可以用来查看全部内容。[变更集](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_changeset)介绍或删除的，例如，特定的文本行。见[被称为“Git DIFF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-diff(1)).

Plumbing 水管

可爱的名字[核心Git](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_core_git).

porcelain

取决于程序和程序套件的可爱名称[核心Git](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_core_git)提出了对核心Git的高级访问。瓷器暴露更多[单片机](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SCM)接口比[水管](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_plumbing).

per-worktree ref 每个工作树参考

Refs that are per-[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)，而不是全球性的。这是目前唯一的[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)以及任何开始的参考文献*参考文献/二等分*但稍后可能会包括其他不寻常的参考文献。

Pseudoref 伪反射

伪ORF是一类文件*Git Udir美元*这些行为类似于REV，用于REV解析，但专门由GIT处理。Pseudorefs都有名字都是上限，并且总是从一条线组成[SHA-1](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SHA1)其次是空白。所以，头部不是伪反射，因为它有时是象征性的参考。它们可能会包含一些额外的数据。*梅吉尔黑德*和*樱桃头*就是例子。不像[每工作树参考文献](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_per_worktree_ref)这些文件不能是符号引用，而且从来没有重排。它们也不能通过正常的REF更新机制来更新。相反，它们是通过直接写入文件来更新的。但是，它们可以被读取，好像它们是参考文献一样。*解析GeigeHead*会工作

Pull 拉

拉A[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)意味着[取来](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_fetch)资讯科技与[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)IT也见[“Git拉（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-pull(1)).

Push 推

推一[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)获取分支机构的方法[主任裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head_ref)来自远方[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)找出它是否是分支的本地首长的直接祖先，在这种情况下，放置所有的对象，即[可达成的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_reachable)从本地磁头REF，从远程存储库中丢失，进入远程[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)如果远程，则更新远程磁头参考。[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)不是本地头的祖先，推动失败了。

Reachable 可达成的

一个给定的祖先[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)被认为是“可达到”的承诺。更一般地说，一[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)如果我们能从另一个到达另一个[链](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_chain)如下[标签](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag)无论他们贴什么标签，[承诺](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)对他们的父母或树木，以及[树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)树木或[斑点](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_blob_object)它们含有

rebase重碱

重新应用一系列的变化[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)到一个不同的基地，并重置[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)那个分支的结果

Ref 裁判

开始的名字*参考文献/*(e.g.*裁判/指挥/指挥*这指的是[对象名称](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_name)或另一个REF（后者称为[符号参考函数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_symref)）为了方便起见，有时使用REF作为对GIT命令的参数时缩写；参见[被称为“GIT修订版（7）”的部分](https://tortoisegit.org/docs/tortoisegit/git-misc.html#gitrevisions(7))详情。参考文献存储在[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository).

REF命名空间是分层的。不同的子层次用于不同的目的（例如*裁判/指挥部/*等级制度用于代表地方分支机构。

有一些特殊用途的参考文献没有开始。*参考文献/*. 最值得注意的例子是*头部*.

Reflog 重新浮起

RIFFLG显示了RIFE的局部“历史”，换句话说，它可以告诉你第三次修订的内容。*这*资源库是什么，当前状态是什么？*这*仓库，昨天晚上9点14分。见[被称为“Git ReFug（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-reflog(1))详情

refspec

“Refspec”is used by[取来](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_fetch)和[推](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_push)描述远程之间的映射[裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)本地参考文件

remote repository 远程存储库

一[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)用于跟踪同一项目，但驻留在其他地方。要与远程通信，请参见[取来](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_fetch)或[推](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_push).

remote-tracking branch 远程跟踪分支

一[裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)这是用来跟随另一个变化的。[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository). 它通常看起来像*REFS/ReloTe/Foo/bar*（指示跟踪一个名为*酒吧*在远程命名*福*），并与配置的获取的右侧匹配。[ReFSPEC](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_refspec). 远程跟踪分支不应包含直接修改或对其进行本地提交。

Repository 知识库

收藏[参考文献](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)连同一个[对象数据库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_database)包含所有对象的[可达成的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_reachable)从参考文献中，可能伴随着来自一个或多个元数据的数据。[瓷器](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_porcelain). 存储库可以通过其他数据库共享对象数据库。[交替机制](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_alternate_object_database).

Resolve 决定

手动修复一个自动故障的动作[合并](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_merge)落在后面

Revision 修订

同义词[犯罪](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)(the noun).

Rewind 重绕

抛弃部分发展，即指派[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)更早[修订](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_revision).

SCM

源代码管理（工具）

SHA-1

“安全散列算法1”；加密散列函数。在Git的上下文中用作同义词[对象名称](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_name).

shallow clone 浅克隆

主要是同义词[浅库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_shallow_repository)但是这个短语更清楚地说明了它是由运行产生的。*git clone --depth=...*命令

shallow repository 浅库

浅层[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)有一些不完整的历史[承诺](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit)有[父母](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_parent)换句话说，Git被告知假装这些承诺没有父母，即使他们被记录在[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)）当你感兴趣的只是一个项目的最近历史，即使在上游记录的真实历史更大时，这有时是有用的。浅库是通过给予*--深度*选择权[“Git克隆（1）”一节](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-clone(1))，它的历史可以进一步加深。[被称为“Git FETCH（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-fetch(1)).

Submodule 子模块

一[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)它将一个单独的项目的历史保存在另一个存储库中（后者称为另一个存储库）。[超级工程](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_superproject)）

Superproject 超级工程

一[知识库](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_repository)在其工作树中引用其他项目的库。[子模块](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_submodule). 超级项目知道包含子模块的提交对象的名称（但不保存副本）。

symref

符号引用：而不是包含[SHA-1](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_SHA1)ID本身，它是格式*参考文献：参考文献/部分/资料*当引用时，它递归引用此引用。[*头部*](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)是SyrRf的一个最好的例子。符号引用是用[被称为“Git符号REF（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git-symbolic-ref(1))命令

Tag 标签

一[裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)在下面*参考文献/标签/*指向任意类型对象的命名空间（通常标记指向[标签](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag_object)或A[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)). In contrast to a[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_head)标签不更新*犯罪*命令。Git标签与Lisp标签无关（称为[对象类型](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object_type)在Git的上下文中）。标记通常用于标记提交祖先中的特定点。[链](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_chain).

tag object 标记对象

安[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)包含一个[裁判](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_ref)指向另一个对象，它可以包含一个消息[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object). 它也可以包含一个（PGP）签名，在这种情况下，它被称为“签名标记对象”。

topic branch 主题分支

规则的Git[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)这是开发人员用来识别概念性开发线的。由于分支是非常简单和廉价的，所以通常希望有几个小分支，每个分支包含非常好定义的概念或小的增量但相关的变化。

Tree 树

要么是[工作树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_working_tree)，或者[树对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)相依[斑点](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_blob_object)树对象（即工作树的存储表示）。

tree object 树对象

安[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)包含文件名和模式的列表以及相关联的BLB和/或树对象的引用。一[树](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree)相当于[目录](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_directory).

tree-ish (also treeish) 树状的（树的）

一[树对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tree_object)或[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)可以递归地对树对象进行解引用。撤销A[提交对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit_object)生成对应于[修订](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_revision)顶部[目录](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_directory). 下面是所有的树[犯下](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_commit-ish)树对象[标记对象](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag_object)指向树对象、指向指向树对象的标记对象的标签对象等。

unmerged index 未合并索引

安[指数](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index)有什么不对劲[索引项](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_index_entry).

unreachable object 不可达对象

安[目标](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_object)哪个不是[可达成的](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_reachable)从A[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)，[标签](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_tag)或任何其他参考文献

upstream branch 上游分支

默认值[分支](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_branch)that is merged into the branch in question (or the branch in question is rebased onto). It is configured via branch.<name>.remote and branch.<name>.merge. If the upstream branch of*一*是*起源/ B*有时我们说“*一*跟踪*起源/ B*“

working tree 工作树

实际签出文件的树。工作树通常包含[头部](https://tortoisegit.org/docs/tortoisegit/apg.html#Git%20User%20Manual_def_HEAD)提交树，加上您所做但尚未提交的任何本地更改。

快速引用

这是对主要命令的快速总结；前面的章节解释了这些工作的细节。

创建新的仓库

From a tarball:

$ tar xzf project.tar.gz  
$ cd project  
$ git init  
Initialized empty Git repository in .git/  
$ git add .  
$ git commit

来自远程存储库：

$ git clone git://example.com/pub/project.git  
$ cd project

管理部门

$ git branch         # list all local branches in this repo  
$ git checkout test  # switch working directory to branch "test"  
$ git branch new     # create branch "new" starting at current HEAD  
$ git branch -d new  # delete branch "new"

代替将新分支建立在当前头部（默认）上，使用：

$ git branch new test    # branch named "test"  
$ git branch new v2.6.15 # tag named v2.6.15  
$ git branch new HEAD^   # commit before the most recent  
$ git branch new HEAD^^  # commit before that  
$ git branch new test~10 # ten commits before tip of branch "test"

同时创建并切换到一个新的分支：

$ git checkout -b new v2.6.15

从您所克隆的存储库中更新和检查分支：

$ git fetch             # update  
$ git branch -r         # list  
  origin/master  
  origin/next  
  ...  
$ git checkout -b masterwork origin/master

从一个不同的存储库中提取一个分支，并在其存储库中赋予它一个新名称：

$ git fetch git://example.com/project.git theirbranch:mybranch  
$ git fetch git://example.com/project.git v2.6.15:mybranch

保存你经常使用的知识库列表：

$ git remote add example git://example.com/project.git  
$ git remote                    # list remote repositories  
example  
origin  
$ git remote show example       # get details  
\* remote example  
  URL: git://example.com/project.git  
  Tracked remote branches  
    master  
    next  
    ...  
$ git fetch example             # update branches from example  
$ git branch -r                 # list all remote branches

探索历史

$ gitk                      # visualize and browse history  
$ git log                   # list all commits  
$ git log src/              # ...modifying src/  
$ git log v2.6.15..v2.6.16  # ...in v2.6.16, not in v2.6.15  
$ git log master..test      # ...in branch test, not in branch master  
$ git log test..master      # ...in branch master, but not in test  
$ git log test...master     # ...in one branch, not in both  
$ git log -S'foo()'         # ...where difference contain "foo()"  
$ git log --since="2 weeks ago"  
$ git log -p                # show patches as well  
$ git show                  # most recent commit  
$ git diff v2.6.15..v2.6.16 # diff between two tagged versions  
$ git diff v2.6.15..HEAD    # diff with current head  
$ git grep "foo()"          # search working directory for "foo()"  
$ git grep v2.6.15 "foo()"  # search old tree for "foo()"  
$ git show v2.6.15:a.txt    # look at old version of a.txt

搜索回归：

$ git bisect start  
$ git bisect bad                # current version is bad  
$ git bisect good v2.6.13-rc2   # last known good revision  
Bisecting: 675 revisions left to test after this  
                                # test here, then:  
$ git bisect good               # if this revision is good, or  
$ git bisect bad                # if this revision is bad.  
                                # repeat until done.

作出改变

确保GIT知道谁该受责备：

$ cat >>~/.gitconfig <<\EOF  
[user]  
        name = Your Name Comes Here  
        email = you@yourdomain.example.com  
EOF

选择要在下一个提交中包含的文件内容，然后提交提交：

$ git add a.txt    # updated file  
$ git add b.txt    # new file  
$ git rm c.txt     # old file  
$ git commit

或者，在一个步骤中准备和创建提交：

$ git commit d.txt # use latest content only of d.txt  
$ git commit -a    # use latest content of all tracked files

合并

$ git merge test   # merge branch "test" into the current branch  
$ git pull git://example.com/project.git master  
                   # fetch and merge in remote branch  
$ git pull . test  # equivalent to git merge test

分享你的改变

进口或出口修补程序：

$ git format-patch origin..HEAD # format a patch for each commit  
                                # in HEAD but not in origin  
$ git am mbox # import patches from the mailbox "mbox"

在不同的Git存储库中提取一个分支，然后合并到当前分支中：

$ git pull git://example.com/project.git theirbranch

在合并到当前分支之前将获取的分支存储到本地分支：

$ git pull git://example.com/project.git theirbranch:mybranch

在本地分支上创建提交之后，用您的约定更新远程分支：

$ git push ssh://example.com/project.git mybranch:theirbranch

当远程和本地分支都被命名为“test”：

$ git push ssh://example.com/project.git test

简短版本的经常使用的远程存储：

$ git remote add example ssh://example.com/project.git  
$ git push example test

知识库维护

Check for corruption:

$ git fsck

重新压缩，去除未使用的结壳：

$ git gc

本手册的注释和待办事项列表

这是一项正在进行中的工作。

基本要求：

* 它必须是可读的，从开始到结束，由一个智能的人掌握基本的UNIX命令行，但没有任何特殊知识的Git。如有必要，任何其他先决条件应在其出现时特别提及。
* 只要有可能，章节标题应该清楚地描述他们解释如何做的任务，在不需要更多知识的语言中：例如，“将补丁导入到项目中”，而不是“*吉特AM*命令“

想想如何创建一个清晰的章节依赖图，它可以让人们在不必阅读所有内容的情况下得到重要的话题。

扫描*文件/*for other stuff left out; in particular:

* 豪托
* 一些*技术*？
* 挂钩
* 命令列表[被称为“Git（1）”的部分](https://tortoisegit.org/docs/tortoisegit/git-command.html#git(1))

扫描电子邮件档案排除其他东西

扫描人页以查看是否有比本手册提供的更多背景。

添加更多的好例子。整本烹饪书的例子可能是个好主意，或许可以制作一个“高级示例”部分，作为章节章节的标准结尾？

在适当的情况下，包括对词汇表的交叉引用。

添加一个与其他版本控制系统一起工作的部分，包括CVS、SuffRead和刚刚发布的系列TayBar。

写一篇关于使用管道和书写脚本的章节。

交替，克隆-引用等

更多的是从存储库腐败中恢复。见：<http://marc.info/?l=git&m=117263864820799&w=2><http://marc.info/?l=git&m=117147855503798&w=2>