Git 允许你通过几种方法来指明特定的或者一定范围内的提交。了解它们并不是必需的，但是了解一下总没坏处。

单个修订版本

显然你可以使用给出的 SHA-1 值来指明一次提交，不过也有更加人性化的方法来做同样的事。本节概述了指明单个提交的诸多方法。

简短的SHA

Git 很聪明，它能够通过你提供的前几个字符来识别你想要的那次提交，只要你提供的那部分 SHA-1 不短于四个字符，并且没有歧义——也就是说，当前仓库中只有一个对象以这段 SHA-1 开头。

例如，想要查看一次指定的提交，假设你运行 git log 命令并找到你增加了功能的那次提交：

$ git log

commit 734713bc047d87bf7eac9674765ae793478c50d3

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jan 2 18:32:33 2009 -0800

fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Merge: 1c002dd... 35cfb2b...

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 14:58:32 2008 -0800

added some blame and merge stuff

假设是 1c002dd.... 。如果你想 git show 这次提交，下面的命令是等价的（假设简短的版本没有歧义）：

$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b

$ git show 1c002dd4b536e7479f

$ git show 1c002d

Git 可以为你的 SHA-1 值生成出简短且唯一的缩写。如果你传递 --abbrev-commit 给 git log 命令，输出结果里就会使用简短且唯一的值；它默认使用七个字符来表示，不过必要时为了避免 SHA-1 的歧义，会增加字符数：

$ git log --abbrev-commit --pretty=oneline

ca82a6d changed the version number

085bb3b removed unnecessary test code

a11bef0 first commit

通常在一个项目中，使用八到十个字符来避免 SHA-1 歧义已经足够了。最大的 Git 项目之一，Linux 内核，目前也只需要最长 40 个字符中的 12 个字符来保持唯一性。

关于 SHA-1 的简短说明

许多人可能会担心一个问题：在随机的偶然情况下，在他们的仓库里会出现两个具有相同 SHA-1 值的对象。那会怎么样呢？

如果你真的向仓库里提交了一个跟之前的某个对象具有相同 SHA-1 值的对象，Git 将会发现之前的那个对象已经存在在 Git 数据库中，并认为它已经被写入了。如果什么时候你想再次检出那个对象时，你会总是得到先前的那个对象的数据。

不过，你应该了解到，这种情况发生的概率是多么微小。SHA-1 摘要长度是 20 字节，也就是 160 位。为了保证有 50% 的概率出现一次冲突，需要 2^80 个随机哈希的对象（计算冲突机率的公式是 p = (n(n-1)/2) \* (1/2^160))。2^80 是 1.2 x 10^24，也就是一亿亿亿，那是地球上沙粒总数的 1200 倍。

现在举例说一下怎样才能产生一次 SHA-1 冲突。如果地球上 65 亿的人类都在编程，每人每秒都在产生等价于整个 Linux 内核历史（一百万个 Git 对象）的代码，并将之提交到一个巨大的 Git 仓库里面，那将花费 5 年的时间才会产生足够的对象，使其拥有 50% 的概率产生一次 SHA-1 对象冲突。这要比你编程团队的成员同一个晚上在互不相干的意外中被狼袭击并杀死的机率还要小。

分支引用

指明一次提交的最直接的方法要求有一个指向它的分支引用。这样，你就可以在任何需要一个提交对象或者 SHA-1 值的 Git 命令中使用该分支名称了。如果你想要显示一个分支的最后一次提交的对象，例如假设 topic1 分支指向ca82a6d，那么下面的命令是等价的：

$ git show ca82a6dff817ec66f44342007202690a93763949

$ git show topic1

如果你想知道某个分支指向哪个特定的 SHA，或者想看任何一个例子中被简写的 SHA-1，你可以使用一个叫做rev-parse 的 Git 探测工具。在第 9 章你可以看到关于探测工具的更多信息；简单来说，rev-parse 是为了底层操作而不是日常操作设计的。不过，有时你想看 Git 现在到底处于什么状态时，它可能会很有用。这里你可以对你的分支运执行 rev-parse。

$ git rev-parse topic1

ca82a6dff817ec66f44342007202690a93763949

引用日志里的简称

在你工作的同时，Git 在后台的工作之一就是保存一份引用日志——一份记录最近几个月你的 HEAD 和分支引用的日志。

你可以使用 git reflog 来查看引用日志：

$ git reflog

734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated

d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.

1c002dd HEAD@{2}: commit: added some blame and merge stuff

1c36188 HEAD@{3}: rebase -i (squash): updating HEAD

95df984 HEAD@{4}: commit: # This is a combination of two commits.

1c36188 HEAD@{5}: rebase -i (squash): updating HEAD

7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD

每次你的分支顶端因为某些原因被修改时，Git 就会为你将信息保存在这个临时历史记录里面。你也可以使用这份数据来指明更早的分支。如果你想查看仓库中 HEAD 在五次前的值，你可以使用引用日志的输出中的 @{n} 引用：

$ git show HEAD@{5}

你也可以使用这个语法来查看某个分支在一定时间前的位置。例如，想看你的 master 分支昨天在哪，你可以输入

$ git show master@{yesterday}

它就会显示昨天分支的顶端在哪。这项技术只对还在你引用日志里的数据有用，所以不能用来查看比几个月前还早的提交。

想要看类似于 git log 输出格式的引用日志信息，你可以运行 git log -g：

$ git log -g master

commit 734713bc047d87bf7eac9674765ae793478c50d3

Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)

Reflog message: commit: fixed refs handling, added gc auto, updated

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jan 2 18:32:33 2009 -0800

fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)

Reflog message: merge phedders/rdocs: Merge made by recursive.

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

需要注意的是，引用日志信息只存在于本地——这是一个记录你在你自己的仓库里做过什么的日志。其他人拷贝的仓库里的引用日志不会和你的相同；而你新克隆一个仓库的时候，引用日志是空的，因为你在仓库里还没有操作。git show HEAD@{2.months.ago} 这条命令只有在你克隆了一个项目至少两个月时才会有用——如果你是五分钟前克隆的仓库，那么它将不会有结果返回。

祖先引用

另一种指明某次提交的常用方法是通过它的祖先。如果你在引用最后加上一个 ^，Git 将其理解为此次提交的父提交。 假设你的工程历史是这样的：

$ git log --pretty=format:'%h %s' --graph

\* 734713b fixed refs handling, added gc auto, updated tests

\* d921970 Merge commit 'phedders/rdocs'

|\

| \* 35cfb2b Some rdoc changes

\* | 1c002dd added some blame and merge stuff

|/

\* 1c36188 ignore \*.gem

\* 9b29157 add open3\_detach to gemspec file list

那么，想看上一次提交，你可以使用 HEAD^，意思是“HEAD 的父提交”：

$ git show HEAD^

commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Merge: 1c002dd... 35cfb2b...

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

你也可以在 ^ 后添加一个数字——例如，d921970^2 意思是“d921970 的第二父提交”。这种语法只在合并提交时有用，因为合并提交可能有多个父提交。第一父提交是你合并时所在分支，而第二父提交是你所合并的分支：

$ git show d921970^

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 14:58:32 2008 -0800

added some blame and merge stuff

$ git show d921970^2

commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548

Author: Paul Hedderly <paul+git@mjr.org>

Date: Wed Dec 10 22:22:03 2008 +0000

Some rdoc changes

另外一个指明祖先提交的方法是 ~。这也是指向第一父提交，所以 HEAD~ 和 HEAD^ 是等价的。当你指定数字的时候就明显不一样了。HEAD~2 是指“第一父提交的第一父提交”，也就是“祖父提交”——它会根据你指定的次数检索第一父提交。例如，在上面列出的历史记录里面，HEAD~3 会是

$ git show HEAD~3

commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d

Author: Tom Preston-Werner <tom@mojombo.com>

Date: Fri Nov 7 13:47:59 2008 -0500

ignore \*.gem

也可以写成 HEAD^^^，同样是第一父提交的第一父提交的第一父提交：

$ git show HEAD^^^

commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d

Author: Tom Preston-Werner <tom@mojombo.com>

Date: Fri Nov 7 13:47:59 2008 -0500

ignore \*.gem

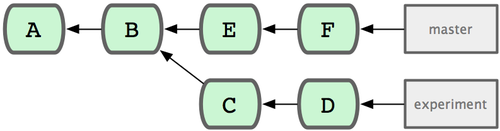
你也可以混合使用这些语法——你可以通过 HEAD~3^2 指明先前引用的第二父提交（假设它是一个合并提交）。

提交范围

现在你已经可以指明单次的提交，让我们来看看怎样指明一定范围的提交。这在你管理分支的时候尤显重要——如果你有很多分支，你可以指明范围来圈定一些问题的答案，比如：“这个分支上我有哪些工作还没合并到主分支的？”

双点

最常用的指明范围的方法是双点的语法。这种语法主要是让 Git 区分出可从一个分支中获得而不能从另一个分支中获得的提交。例如，假设你有类似于图 6-1 的提交历史。

  
图 6-1. 范围选择的提交历史实例

你想要查看你的试验分支上哪些没有被提交到主分支，那么你就可以使用 master..experiment 来让 Git 显示这些提交的日志——这句话的意思是“所有可从experiment分支中获得而不能从master分支中获得的提交”。为了使例子简单明了，我使用了图标中提交对象的字母来代替真实日志的输出，所以会显示：

$ git log master..experiment

D

C

另一方面，如果你想看相反的——所有在 master 而不在 experiment 中的分支——你可以交换分支的名字。experiment..master 显示所有可在 master 获得而在 experiment 中不能的提交：

$ git log experiment..master

F

E

这在你想保持 experiment 分支最新和预览你将合并的提交的时候特别有用。这个语法的另一种常见用途是查看你将把什么推送到远程：

$ git log origin/master..HEAD

这条命令显示任何在你当前分支上而不在远程origin 上的提交。如果你运行 git push 并且的你的当前分支正在跟踪 origin/master，被git log origin/master..HEAD 列出的提交就是将被传输到服务器上的提交。 你也可以留空语法中的一边来让 Git 来假定它是 HEAD。例如，输入 git log origin/master.. 将得到和上面的例子一样的结果—— Git 使用 HEAD 来代替不存在的一边。

多点

双点语法就像速记一样有用；但是你也许会想针对两个以上的分支来指明修订版本，比如查看哪些提交被包含在某些分支中的一个，但是不在你当前的分支上。Git允许你在引用前使用^字符或者--not指明你不希望提交被包含其中的分支。因此下面三个命令是等同的：

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

这样很好，因为它允许你在查询中指定多于两个的引用，而这是双点语法所做不到的。例如，如果你想查找所有从refA或refB包含的但是不被refC包含的提交，你可以输入下面中的一个

$ git log refA refB ^refC

$ git log refA refB --not refC

这建立了一个非常强大的修订版本查询系统，应该可以帮助你解决分支里包含了什么这个问题。

三点

最后一种主要的范围选择语法是三点语法，这个可以指定被两个引用中的一个包含但又不被两者同时包含的分支。回过头来看一下图6-1里所列的提交历史的例子。 如果你想查看master或者experiment中包含的但不是两者共有的引用，你可以运行

$ git log master...experiment

F

E

D

C

这个再次给出你普通的log输出但是只显示那四次提交的信息，按照传统的提交日期排列。

这种情形下，log命令的一个常用参数是--left-right，它会显示每个提交到底处于哪一侧的分支。这使得数据更加有用。

$ git log --left-right master...experiment

< F

< E

> D

> C

有了以上工具，让Git知道你要察看哪些提交就容易得多了。