

# 版本控制的前世和今生

除了茫然未知的宇宙，几乎任何事物都是从无到有，从简陋到完善。随着时间车轮的滚滚向前，历史被抛在身后逐渐远去，如同我们的现代社会，世界大同，到处都是忙碌和喧嚣，再也看不到已经远去的刀耕火种、男耕女织的慢生活岁月。

版本控制系统是一个另类。虽然其历史并不短暂，也有几十年，但是它的演进过程却一直在社会的各个角落重复着，而且惊人的相似。有的人从未使用甚至从未听说过版本控制系统，他和他的团队就像停留在黑暗的史前时代，任由数据自生自灭。有的人使用着有几十年历史的CVS或其改良版Subversion，让时间空耗在网络连接的等待中。再有就是以Git为代表的分布式版本控制系统，已经风靡整个开源社区，正等待你的靠近。

## 黑暗的史前时代

人们谈及远古，总爱以黑暗形容。黑暗实际上指的是秩序和工具的匮乏，而不是自然，如以自然环境而论，工业化和城市化对环境的破坏，现今才是最黑暗的年代。对软件开发来说也是如此，虽然遥远的C语言一统天下的日子，要比今天选Java，选.NET，还是选择脚本语言的多选题要简单得多，但是从工具和秩序上讲，过去的年代是黑暗的。

回顾一下我经历的版本控制的“史前时代”吧。在大学里，代码分散地拷贝在各个软盘中，最终我会被搞糊涂，不知道哪个软盘中的代码是最优的，因为最新并非最优，失败的重构会毁掉原来尚能运作的代码。在我工作的第一年，代码管理并未改观，还是以简单的目录拷贝进行数据的备份，三四个程序员利用文件服务器的目录共享进行协同，公共类库和头文件在操作过程中相互覆盖，痛苦不堪。很明显，那时我尚不知道版本控制系统为何物。我的版本控制史前时代一直延续到2000年，那时CVS已经诞生了14年，而我在那时对CVS还一无所知。

实际上，即便是在CVS出现之前的“史前时代”，也已经有了非常好用的源码比较和打补丁的工具：`:command:`diff`` 和 `:command:`patch``，他们今天生命力依然顽强。大名鼎鼎的Linus Torvalds（Linux之父）也对这两个工具偏爱有加，在1991-2002年之间，Linus一直顽固地使用`:command:`diff`` 和 `:command:`patch`` 管理着Linux的代码，即使不断有人提醒他CVS的存在[1]。

那么来看看`:command:`diff`` 和 `:command:`patch``，熟悉它们将对理解版本控制系统（差异存储），使用版本控制系统（代码比较和冲突合并）都有莫大的好处。

## 命令`:command:`diff`` 用于比较两个文本文件或目录的差异

先来构造两个文件[2]：

文件 <code>:file:`hello`</code>	文件 <code>:file:`world`</code>
应该杜绝文章中的错别子。	应该杜绝文章中的错别字。
但是无论使用 * 全拼，双拼 * 还是五笔	但是无论使用 * 全拼，双拼 * 还是五笔
是人就有可能犯错，软件更是如此。	是人就有可能犯错，软件更是如此。
犯了错，就要扣工资！	改正的成本可能会很高。
改正的成本可能会很高。	但是“只要眼球足够多，所有Bug都好捉”， 这就是开源的哲学之一。

对这两个文件执行`:command:`diff`` 命令，查看两个文件的差异。如下所示：

```
$ diff -u hello world | less -N
```

上面执行 :command:`diff` 命令的 -u 参数很重要，使得差异输出中带有上下文。管道后面带有 -N 参数的 :command:`less` 命令（按字母 q 退出）会在输出的每一行前面添加行号，便于对输出结果进行说明。

```
1 --- hello      2010-09-21 17:45:33.551610940 +0800
2 +++ world     2010-09-21 17:44:46.343610465 +0800
3 @@ -1,4 +1,4 @@
4 -应该杜绝文章中的错别子。
5 +应该杜绝文章中的错别字。
6
7 但是无论使用
8 * 全拼，双拼
9 @@ -6,6 +6,7 @@
10
11 是人就有可能犯错，软件更是如此。
12
13 -犯了错，就要扣工资！
14 -
15 改正的成本可能会很高。
16 +
17 +但是“只要眼球足够多，所有Bug都好捉”，
18 +这就是开源的哲学之一。
```

上面的差异文件，可以这么理解：

- 第1、2行作为差异文件的文件头，分别记录了用于比较的原始文件和目标文件的文件名及时间戳。第1行以三个减号 (---) 开始，记录原始文件的文件名和时间戳，而第2行以三个加号 (+++) 开始，记录的是目标文件的文件名及时间戳。
- 在比较内容中，以减号 (-) 开始的行是只出现在原始文件中的行，而在目标文件中不存在，即被删除的内容。例如：第4、13、14行。
- 在比较内容中，以加号 (+) 开始的行是只出现在目标文件中的行，而在原始文件中不存在，即新增加的内容。例如：第5、16-18行。
- 在比较内容中，以空格开始的行，是在原始文件和目标文件中都出现的行，用于上下文参考。例如：第6-8、10-12、15行。
- 第3-8行是第一个差异小节。每个差异小节以一行定位语句开始。第3行就是一条差异定位语句，其前后分别用两个@进行标识。
- 第3行定位语句中 -1,4 的含义是：本差异小节的内容相当于原始文件的从第1行开始的4行。不妨计算一下，第4、6、7、8行是原始文件中的内容，加起来刚好是4行。
- 第3行定位语句中 +1,4 的含义是：本差异小节的内容相当于目标文件的从第1行开始的4行。第5、6、7、8行是目标文件中的内容，加起来刚好是4行。
- 第9-18行是第二个差异小节。第9行是一条定位语句。
- 第9行定位语句中 -6,6 的含义是：本差异小节的内容相当于原始文件的从第6行开始的6行。统计一下，第10-15行是原始文件中的内容，加起来刚好是6行。
- 第9行定位语句中 +6,7 的含义是：本差异小节的内容相当于目标文件的从第6行开始的7行。第10-12、15-18行是目标文件中的内容，加起来刚好是7行。
- 命令 :command:`diff` 是基于行比较，所以即便只修改了一个字，也显示为一整行的修改（参见差异文件第4、5行）。Git 对 :command:`diff` 进行了扩展，提供一种逐词比较的差异比较方法，参见本书第2篇“11.4.4 差异比较：git diff”小节。

## 命令 :command:`patch` 相当于 :command:`diff` 的反向操作

有了原始文件 (:file:`hello`) 和差异文件 (:file:`diff.txt`)，若目标文件 (:file:`world`) 被删除或被覆盖，可以用下面的命令来恢复目标文件 (:file:`world`)：

```
$ cp hello world  
$ patch world < diff.txt
```

反之亦然。用目标文件（:file:`world`）和差异文件（:file:`diff.txt`）来恢复原始文件（:file:`hello`），使用如下操作：

```
$ cp world hello  
$ patch -R hello < diff.txt
```

命令:command:`diff` 和:command:`patch` 还可以对目录进行比较和恢复操作，这也就是 Linus 在 1991-2002 年用于维护 Linux 不同版本间差异的办法。可以用此命令，在没有版本控制系统的情况下，记录并保存改动前后的差异，还可以将差异文件注入版本控制系统（如果有的话）。

标准的:command:`diff` 和:command:`patch` 命令存在一个局限，就是不能对二进制文件进行处理。对二进制文件的修改或添加会在差异文件中缺失，进而丢失对二进制文件的改动或添加。Git 对差异文件格式提供了扩展支持，支持二进制文件的比较，解决了这个问题。这点可以参考本书第 7 篇“第 38 章 补丁中的二进制文件”的相关内容。

## CVS——开启版本控制大爆发

CVS (Concurrent Versions System) [3] 诞生于 1985 年，是由荷兰阿姆斯特丹 VU 大学的 Dick Grune 教授实现的。当时 Dick 教授和两个学生共同开发一个项目，但是三个人的工作时间无法协调到一起，迫切需要一个记录和协同代码开发的工具软件。于是 Dick 教授通过脚本语言对 RCS（一个针对单独文件的版本管理工具）进行封装，设计出有史以来第一个被大规模使用的版本控制工具。在 Dick 教授的网站上记录了 CVS 这段早期的历史。[4]

“在 1985 年一个糟糕的秋日里，我站在校汽车站等车回家，脑海里一直纠结着一件事 —— 如何处理 RCS 文件、用户文件（工作区）和 Entries 文件的复杂关系，有的文件可能会缺失、冲突、被删除，等等。我的头有些晕了，于是决定画一个大表，将复杂的关联画在其中看看出来的结果是什么样的……”

1986 年 Dick 教授通过新闻组发布了 CVS，1989 年由 Brian Berliner 将 CVS 用 C 语言重写。

从 CVS 的历史可以看出 CVS 不是设计出来的，而是被实际需要逼出来的，因此根据实用为上的原则，借用了已有的针对单一文件的多版本管理工具 RCS。CVS 采用客户端/服务器架构设计，版本库位于服务器端，实际上就是一个 RCS 文件容器。每一个 RCS 文件以 “,v” 作为文件名后缀，用于保存对应文件的历次更改历史。RCS 文件中只保留一个版本的完全拷贝，其他历次更改仅将差异存储其中，使得存储变得更加高效。我在 2008 年设计的一个 SVN 管理后台 pySvnManager [5]，实际上也采用了 RCS 作为保存 SVN 授权文件变更记录的“数据库”。

图 1-1 展示了 CVS 版本控制系统的工作原理，可以看到作为 RCS 文件容器的 CVS 版本库和工作区目录结构的一一对应关系。

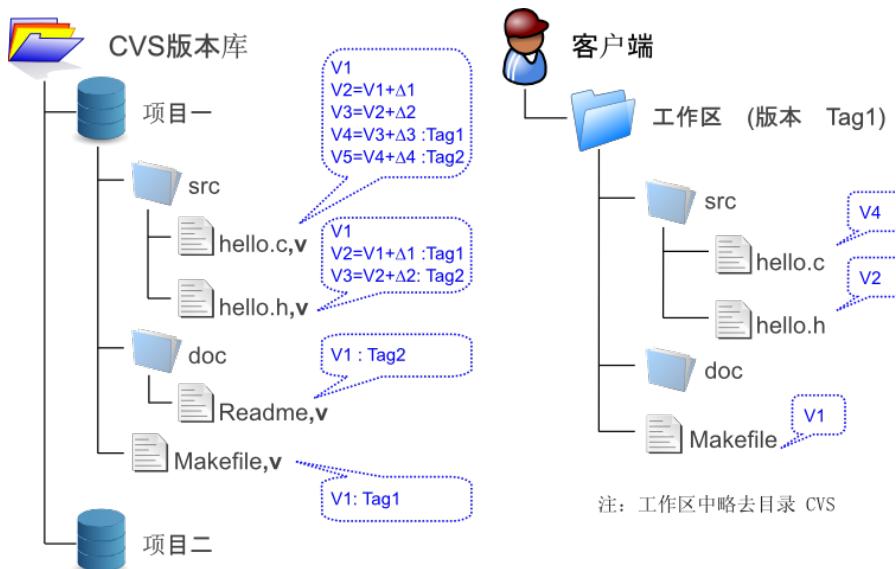


图1-1: CVS版本控制系统示意图

CVS的这种实现方式的最大好处就是简单。把版本库中随便一个目录拿出来就可以成为另外一个版本库。如果将版本库中的一个RCS文件重命名，工作区检出的文件名也相应地改变。这种低成本的服务器管理模式成为很多CVS粉丝至今不愿舍弃CVS的原因。

CVS的出现让软件工程师认识到了原来还可以这样协同工作。CVS成功地为后来的版本控制系统确立了标准，像提交（commit）、检入（checkin）、检出（checkout）、里程碑（tag或译为标签）、分支（branch）等概念早在CVS中就已经确立。CVS的命令行格式也被后来的版本控制系统竞相模仿。

在2001年，我正为使用CVS激动不已的时候，公司领导要求采用和美国研发部门同样的版本控制解决方案。于是，我的项目组率先进行了从CVS到该商业版本控制工具的迁移[6]。虽然商业版本控制工具有更漂亮的界面及更好的产品整合性，但是就版本控制本身而言，商业版本控制工具有着如下缺陷。

- 采用黑盒子式的版本库设计。让人捉摸不透的版本库设计，最大的目的可能就是阻止用户再迁移到其他平台。
- 缺乏版本库整理工具。如果有任何一个文件（如记录核弹起爆密码的文件）检入到版本库中，就没有办法再彻底移除它。
- 商业版本控制工具很难为个人提供版本控制解决方案，除非个人愿意花费高昂的许可证费用。
- 商业版本控制工具注定是小众软件，对新员工的培训成本不可忽视。

而上述商业版本控制系统的缺点，恰恰是CVS及其他开源版本控制系统的强项。但在经历了最初的成功之后，CVS也尽显疲态：

- 服务器端松散的RCS文件，导致在建立里程碑或分支时缺乏效率，服务器端文件越多，速度越慢。
- 分支和里程碑不可见，因为它们被分散地记录在服务器端的各个RCS文件中。
- 合并困难重重，因为缺乏对合并的追踪从而导致重复合并，引发严重冲突。
- 缺乏对原子提交的支持，会导致客户端向服务器端提交不完整的数据。
- 不能优化存储内容相同但文件名不同的文件，因为在服务器端每个文件都是单独进行差异存储的。
- 不能对文件和目录的重命名进行版本控制，虽然直接在服务器端修改RCS文件名可以让改名后的文件保持历史，但是这样做实际会破坏历史。
- 网络操作效率不高，修改的文件在提交时要通过网络传输完整的文件，这是因为本地缺乏文件的原始拷贝而不能在提交前计算出差异数据。

CVS的成功开启了版本控制系统的大爆发，各式各样的版本控制系统如雨后春笋般地诞生了。新的版本控制系统或多或少地解决了CVS版本控制系统存在的问题。在这些版本控制系

统中最典型的就是Subversion (SVN)。

## SVN——集中式版本控制集大成者

Subversion[7]，因其命令行工具名为:command:`svn`因此通常被简称为SVN。SVN由CollabNet公司于2000年资助并发起开发，目的是创建一个更好用的版本控制系统以取代CVS。前期SVN的开发使用CVS做版本控制，到了2001年，SVN已经可以用于自己的版本控制了[8]。SVN成熟的标志是其完成了后端存储上的变革，即从一开始的BDB（简单的关系型数据库）到FSFS（文件数据库）的转变[9]。FSFS相对于BDB具有更高的稳定性、免维护性，以及实现的可视性。图1-2展示了采用FSFS作为存储后端的SVN版本控制系统的工作原理。

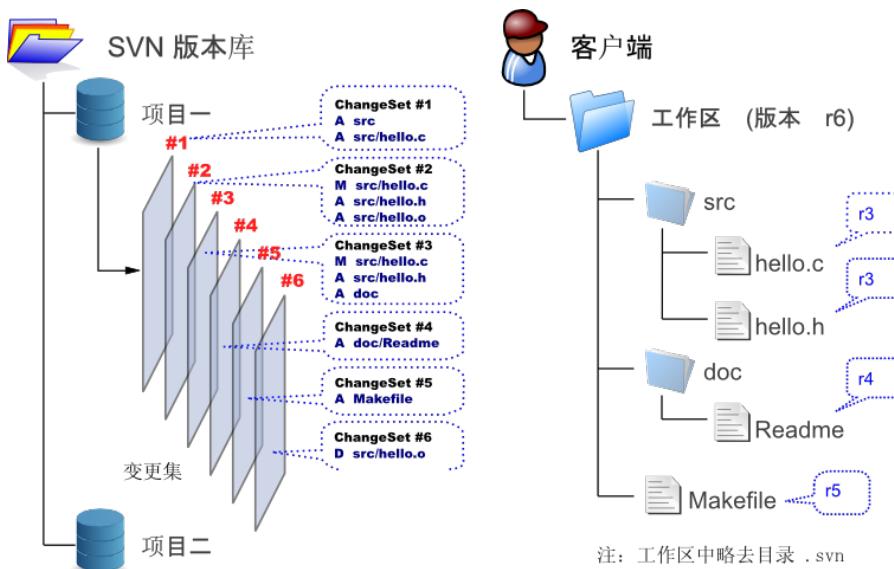


图1-2: SVN版本控制系统示意图

SVN的每一次提交，都会在服务器端的:file:`db/revs` 和:file:`db/revprops` 目录下各创建一个以顺序数字编号命名的文件。其中:file:`db/revs` 目录下的文件（即变更集文件）记录与上一个提交之间的差异（字母A表示新增，M表示修改，D表示删除）。

在:file:`db/revprops` 目录下的同名文件（没有在图1-2中体现）则保存着提交日志、作者、提交时间等信息（称作版本属性）。这样设计的好处有：

- 拥有全局版本号。每提交一次，SVN的版本号就会自动加一。这为SVN的使用提供了极大的便利。回想CVS时代，每个文件都拥有各自独立的版本号（RCS版本号），要想获得全局版本号，只能通过手工不断地建立里程碑（tag）来实现。
- 实现了原子提交。SVN不会像CVS那样出现部分文件被提交而其他文件由于故障没有被提交的状态。
- 文件名不受限制。因为服务器端不再需要建立和客户端文件相似的文件名，这样，文件的命名就不再受服务器操作系统的字符集及大小写的限制。
- 文件和目录重命名也得到了支持。

SVN最具特色的功能是轻量级拷贝，例如将目录:file:`trunk` 拷贝

为:file:`branches/v1.x` 的操作类似于创建符号链接（仅需在:file:`db/revs` 下的变更集文件中用特定的语法标注一下），是轻量级操作，可快速完成。利用轻量级拷贝，SVN在不同的名字空间下创建不同的目录实现里程碑和分支的创建，轻松地解决了CVS中存在的里程碑、分支创建速度慢又不可见的问题。使用SVN创建里程碑和分支只在眨眼之间。

SVN在版本库授权上也有改进，不再像CVS那样依赖操作系统本身对版本库目录和文件进行授权，而是采用授权文件的方式来实现。

SVN还有一个突破，就是在工作区跟踪目录(:file:`.svn` 目录)下为当前目录中的每一个文件都保存一份冗余的原始拷贝。这样做的好处一个是提高了网络的效率，在提交时仅传输变更差异，另外一个好处是部分操作不再需要网络连接，如本地修改的差异比较，以及

本地更改的回退等。

正是由于SVN的这些闪亮的功能，使得SVN成为继CVS之后诞生的诸多版本控制系统中的集大成者，成为开源社区一时的新宠，也成为当时各个企业版本控制的最佳选择之一。

但是SVN相对CVS在本质上并没有突破，都属于集中式版本控制系统，即一个项目只有唯一的一个版本库与之对应，所有的项目成员都通过网络向该服务器进行提交。单点故障是集中式版本控制的死穴，并由此带来数据备份和数据恢复的管理成本。此外集中式版本控制系统还存在着提交瓶颈。

所谓提交瓶颈就是单位时间内版本库允许的提交数量的限制。当提交非常密集时，会出现有的用户始终无法完成本地工作区的改动和服务器最新版本间的合并，其所做的改动无法提交的状况。为避免过早地出现提交瓶颈，SVN允许本地出现混杂版本（即工作区文件版本不一致，有的可能是最新版本，有的可能是历史版本），并可以针对部分目录、文件进行提交。这种非全量的提交方式会导致版本库中文件状态不可测，即本地提交前代码编译、运行是完好的，但被他人更新出来的版本存在bug。

集中式版本控制系统对分布式开发支持得不好，在局域网之外使用SVN，单是查看日志、提交数据等操作的延迟，就足以让基于广域网协同工作的团队抓狂了。

除了集中式版本控制系统固有的问题外，SVN的里程碑、分支的设计也被证明是一个错误，虽然这个错误使得SVN拥有了快速创建里程碑和分支的能力，但是这个错误导致了如下的更多问题。

- 项目文件在版本库中必须按照一定的目录结构进行部署，否则就可能无法建立里程碑和分支。

我在项目咨询过程中就见过很多团队，直接在版本库的根目录下创建项目文件。这样的版本库布局，在需要创建里程碑和分支时就无从下手了，因为根目录是不能拷贝到子目录中的。所以SVN的用户在创建版本库时必须遵守一个古怪的约定：先创建三个顶级目录:`:file:`/trunk``、`:file:`/tags``和`:file:`/branches``。

- 创建里程碑和分支会破坏精心设计的授权。

SVN的授权是基于目录的，分支和里程碑也被视为目录（和其他目录没有分别）。因此每次创建分支或里程碑时，就要将针对`:file:`/trunk``目录及其子目录的授权在新建的分支或里程碑上重建。随着分支和里程碑数量的增多，授权愈加复杂，维护也愈加困难。

- 虽然在SVN 1.5之后拥有了合并追踪功能，但仅适用于单向的合并追踪。

SVN的合并追踪信息并非由合并提交本身提供，而是通过记录在合并的目标目录之上、由独立于合并提交之外的属性提供的，是单边而非双边的。所以这种合并追踪方式仅适用于分支间的单向合并，对双向合并和复杂的多分支合并帮助不大。

2009年底，SVN由CollabNet公司交由Apache社区管理，至此SVN成为了Apache的一个子项目[\[10\]](#)。这对SVN到底意味着什么？是开发的停滞，还是新的开始，结果如何我们将拭目以待。

## Git——Linus的第二个伟大作品

Linux之父Linus是坚定的CVS反对者，他也同样地反对SVN。这就是为什么在1991-2002这十余年间，Linus宁可使用补丁文件和tar包的方式维护代码，也迟迟不愿使用CVS。2002年Linus顶着开源社区精英们的口诛笔伐，选择了一个商业版本控制系统BitKeeper作为Linux内核的代码管理工具[\[11\]](#)。和CVS/SVN不同，BitKeeper是属于分布式版本控制系统。

分布式版本控制系统最大的反传统之处在于，可以不需要集中式的版本库，每个人都工作在通过克隆操作建立的本地版本库中，也就是说每个人都拥有一个完整的版本库。分布式版本控制系统的几乎所有操作包括查看提交日志、提交、创建里程碑和分支、合并分支、

回退等都直接在本地完成而不需要网络连接。每个人都是本地版本库的主人，不再有谁能提交谁不能提交的限制，加之多样的协同工作模型（版本库间推送、拉回，及补丁文件传送等）让开源项目的参与度有爆发式增长。

2005年发生的一件事最终导致了Git的诞生。在2005年初Andrew Tridgell，即大名鼎鼎的Samba的作者，试图尝试对BitKeeper反向工程，以开发一个能与BitKeeper交互的开源工具。这激怒了BitKeeper软件的所有者BitMover公司，要求收回对Linux社区免费使用BitKeeper的授权 [12]。迫不得已，Linus选择了自己开发一个分布式版本控制工具以替代BitKeeper。以下是Git诞生大事记 [13]：

- 2005年4月3日，开始开发Git。
- 2005年4月6日，项目发布。
- 2005年4月7日，Git就可以作为自身的版本控制工具了。
- 2005年4月18日，发生第一个多分支合并。
- 2005年4月29日，Git的性能就已经达到了Linus的预期。
- 2005年6月16日，Linux核心2.6.12发布，那时Git已经在维护Linux核心的源代码了。

Linus以一个文件系统专家和内核设计者的视角对Git进行了设计，其独特的设计，让Git拥有非凡的性能和存储管理。完成原型设计后，在2005年7月26日，Linus功成身退，将Git的维护交给另外一个Git的主要贡献者Junio C Hamano [14]，直到现在。

最初的Git除了一些核心命令以外，其他的都用脚本语言开发，而且每个功能都作为一条独立的命令，例如克隆操作的命令`:command:`git-clone``，提交操作的命令`:command:`git-commit``。这导致Git拥有庞大的命令集，使用习惯也和其他版本控制系统格格不入。随着Git的开发者和使用者的增加，Git的使用界面也变得更友好。例如到1.5.4版本时，将一百多个独立的命令封装为一个`:command:`git``命令，使用习惯已经和其他版本控制工具非常一致了。

在Git出现之前，SVN曾是开源项目版本控制的毋庸置疑的首选，但是在Git诞生后的短短几年，开源项目中再一次出现了版本控制系统的大迁移，Git取代SVN成为当之无愧的版本控制之王。看看下面这些使用Git的项目吧，各个都耳熟能详：Linux kernel、Perl、Eclipse、Gnome、KDE、Qt、Ruby on Rails、Android、PostgreSQL、Debian、X.org，当然还有GitHub上的上百万个项目。

成为版本控制之王，Git当之无愧。

- 安全性强。

抵御了kernel.org在2011年的黑客事件。Git管理的每一个文件、目录、提交等都使用SHA1哈希值。

分布式。

- No delta，全量提交。
- 提交的父子关系和分支。
- DAG。提交

[1] Linus Torvalds于2007-05-03在Google的演讲：<http://www.youtube.com/watch?v=4XpnKHJAok8>

[2] 文件中特意留下的错别字（“字”误为“子”），是便于演示文件的差异比较。

[3] <http://www.nongnu.org/cvs/>

[4] <http://dickgrune.com/Programs/CVS.orig/#History>

[5]	<a href="http://pysvnmanager.sourceforge.net/">http://pysvnmanager.sourceforge.net/</a>
[6]	于是就有了这篇文章： <a href="http://www.worldhello.net/doc/cvs_vs_starteam/">http://www.worldhello.net/doc/cvs_vs_starteam/</a>
[7]	<a href="http://subversion.apache.org/">http://subversion.apache.org/</a>
[8]	<a href="http://svnbook.red-bean.com/en/1.5/svn.intro.whatis.html#svn.intro.history">http://svnbook.red-bean.com/en/1.5/svn.intro.whatis.html#svn.intro.history</a>
[9]	<a href="http://subversion.apache.org/docs/release-notes/1.2.html">http://subversion.apache.org/docs/release-notes/1.2.html</a>
[10]	<a href="http://en.wikipedia.org/wiki/Apache_Subversion">http://en.wikipedia.org/wiki/Apache_Subversion</a>
[11]	<a href="http://en.wikipedia.org/wiki/BitKeeper">http://en.wikipedia.org/wiki/BitKeeper</a>
[12]	<a href="http://en.wikipedia.org/wiki/Andrew_Tridgell">http://en.wikipedia.org/wiki/Andrew_Tridgell</a>
[13]	<a href="http://en.wikipedia.org/wiki/Git_%28software%29">http://en.wikipedia.org/wiki/Git_%28software%29</a>
[14]	<a href="http://marc.info/?l=git&amp;m=112243466603239">http://marc.info/?l=git&amp;m=112243466603239</a> 来源： <a href="https://github.com/gotgit/gotgit/blob/master/01-meet-git/010-scm-history.rst">https://github.com/gotgit/gotgit/blob/master/01-meet-git/010-scm-history.rst</a>

# 爱上Git的理由

本章通过一些典型应用展示Git作为版本控制系统的独特用法。对于不熟悉版本控制系统的读者，可以通过这些示例对版本控制拥有感性的认识。如果是有经验的读者，示例中的和SVN的对照可以让您体会到Git的神奇和强大。本章将列举Git的一些闪亮特性，期待能够让您爱上Git。

## 每日的工作备份

当我开始撰写本书时才明白写书真的是一个辛苦活。如何让辛苦的工作不会因为笔记本硬盘的意外损坏而丢失？如何防范灾害而不让一个篮子里的鸡蛋都毁于一旦？下面就介绍一下我在写本书时如何使用Git进行文稿备份的，请看图2-1。

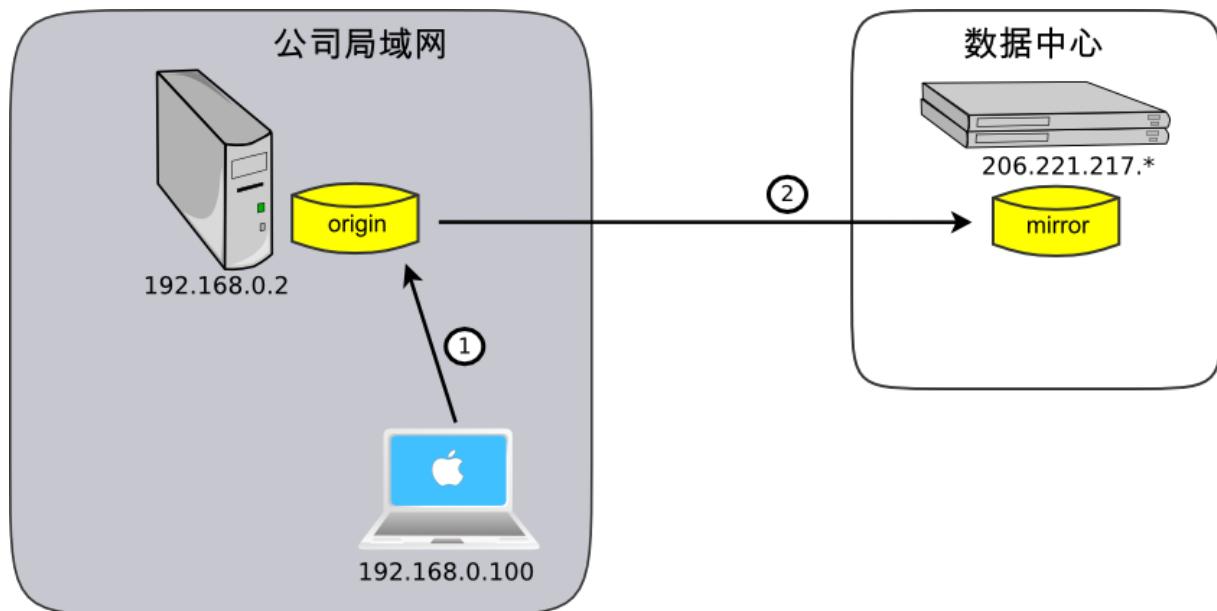


图2-1：利用Git做数据的备份

如图2-1，我的笔记本在公司局域网里的IP地址是192.168.0.100，公司的Git服务器的IP地址是192.168.0.2。公司使用动态IP上网因而没有固定的外网IP，但是公司在数据中心有托管服务器，拥有固定的IP地址，其中一台服务器用作Git服务器镜像。

我的写书习惯大概是这样：一般在写完一个小节，或是画完一张图，我会执行下面的命令提交一次。每一天平均提交3-5次。提交是在笔记本本地完成的，因此在图中没有表示出来。

```
$ git add -u      # 如果创建了新文件，可以执行 git add -i 命令。  
$ git commit
```

下班后，我会执行一次推送操作，将我在本地Git版本库中的提交同步到公司的Git服务器上。相当于图2-1中的步骤①。

```
$ git push
```

因为公司的Git服务器和异地数据中心的Git服务器建立了镜像，所以每当我向公司内网服务器推送的时候，就会自动触发从内网服务器到外网Git服务器的镜像操作。相当于图2-1中的步骤②，步骤②是自动执行的无须人工干预。图2-1中标记为mirror的版本库就是Git镜像版本库，该版本库只向用户提供只读访问服务，而不能对其进行写操作（推送）。

从图2-1中可以看出，我的每日工作保存有三个拷贝，一个在笔记本中，一个在公司内网的服务器上，还有一个在外网的镜像版本库中。鸡蛋分别装在了三个篮子里。

至于如何架设可以实时镜像的Git服务器，会在本书第5篇“第30章 Gitolite服务架设”中予以介绍。

## 异地协同工作

为了能够加快写书的进度，熬夜是必须的，这就出现了在公司和在家两地工作同步的问题。图2-2用于说明我是如何解决两地工作同步的问题的。

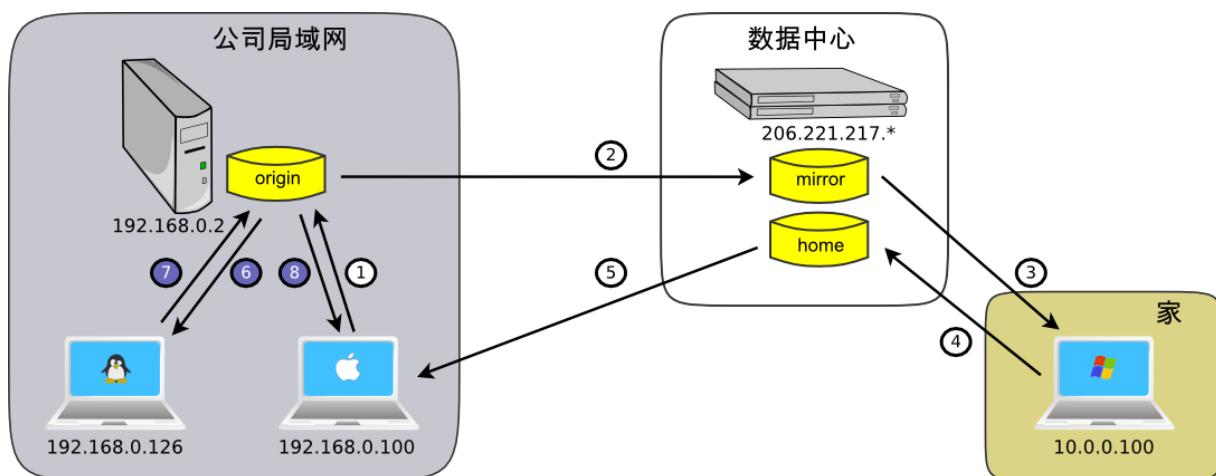


图2-2：利用Git实现异地工作协同

我在家里的电脑IP地址是10.0.0.100（家里也有一个小局域网）。如果在家里有时间工作的话，首先要做的就是图2-2中步骤③的操作：从mirror版本库同步数据到本地。只需要一条命令就好了：

```
$ git pull mirror master
```

然后在家里的电脑上编辑书稿并提交。当准备完成一天的工作时，就执行下面的命令，相当于图2-2中步骤④的操作：将在家中的提交推送到标记为home的版本库中。

```
$ git push home
```

为什么还要再引入另外一个名为home的版本库呢？使用mirror版本库不好么？不要忘了mirror版本库只是一个镜像库，不能提供写操作。

当一早到公司，开始动笔写书之前，先要执行图2-2中步骤⑤的操作，从home版本库将家里做的提交同步到公司的电脑中。

```
$ git pull home master
```

公司的小崔是我这本书的忠实读者，我每有新章节出来，他都会执行图2-2中步骤⑥的工作，从公司内网服务器获取我最新的文稿。

```
$ git pull
```

一旦发现文字错误，小崔会直接在文稿中修改，然后推送到公司的服务器上（图2-2中步骤⑦）。当然他的这个推送也会自动同步到外网的mirror版本库。

```
$ git push
```

而我只要执行：[command: `git pull`](#) 操作就可以获得小崔对我文稿的修订（图2-2中的步骤⑧）。采用这种工作方式，文稿竟然分布在5台电脑上拥有6个拷贝，真可谓狡兔三窟。不，比狡兔还要多三窟。

在本节中，出现在Git命令中的mirror和home是和工作区关联的远程版本库。关于如何注册和使用远程版本库，请参见本书第3篇“第19章 远程版本库”中的内容。

## 现场版本控制

所谓现场版本控制，就是在客户现场或在产品部署的现场，进行源代码的修改，并在修改过程中进行版本控制，以便在完成修改后能够将修改结果甚至修改过程一并带走，并能够将修改结果合并至项目对应的代码库中。

### SVN的解决方案

如果使用SVN进行版本控制，首先要将服务器上部署的产品代码目录变成SVN工作区，这个过程并不简单而且会显得很繁琐，最后将改动结果导出也非常不方便，具体操作过程如下。

1. 在其他位置建立一个SVN版本库。

```
$ svnadmin create /path/to/repos/project1
```

2. 在需要版本控制的目录下检出刚刚建立的空版本库。

```
$ svn checkout file:///path/to/repos/project1 .
```

3. 执行文件添加操作，然后执行提交操作。这个提交将是版本库中编号为1的提交。

```
$ svn add *
$ svn ci -m "initialized"
```

4. 然后开始在工作区中修改文件，提交。

```
$ svn ci
```

5. 如果对修改结果满意，可以通过创建补丁文件的方式将工作成果保存带走。但是SVN很难对每次提交逐一创建补丁，一般用下面的命令与最早的提交进行比较，以创建出一个大补丁文件。

```
$ svn diff -r1 > hacks.patch
```

上面用SVN将工作成果导出的过程存在一个致命的缺陷，就是SVN的补丁文件不支持二进制文件，因此采用补丁文件的方式有可能丢失数据，如新增或修改的图形文件会丢失。更为稳妥但也更为复杂的方式可能要用到：[command: `svnadmin`](#) 命令将版本库导出。命令如下：

```
$ svnadmin dump --incremental -r2:HEAD \
/path/to/repos/project1/ > hacks.dump
```

将：[command: `svnadmin`](#) 命令创建的导出文件恢复到版本库中也非常具有挑战性，这里就不再详细说明了。还是来看看Git在这种情况下的表现吧。

## Git的解决方案

Git对产品部署目录进行到工作区的转化相比SVN要更为简单，而且使用Git将提交历史导出也更为简练和实用，具体操作过程如下：

1. 现场版本库创建。直接在需要版本控制的目录下执行Git版本库初始化命令。

```
$ git init
```

2. 添加文件并提交。

```
$ git add -A  
$ git commit -m "initialized"
```

3. 为初始提交建立一个里程碑：“v1”。

```
$ git tag v1
```

4. 然后开始在工作区中工作——修改文件，提交。

```
$ git commit -a
```

5. 当对修改结果满意，想将工作成果保存带走时，可以通过下面的命令，将从v1开始的历次提交逐一导出为补丁文件。转换的补丁文件都包含一个数字前缀，并提取提交日志信息作为文件名，而且补丁文件还提供对二进制文件的支持。下面命令的输出摘自本书第3篇“第20章 补丁文件交互”中的实例。

```
$ git format-patch v1..HEAD  
0001-Fix-typo-help-to-help.patch  
0002-Add-I18N-support.patch  
0003-Translate-for-Chinese.patch
```

6. 通过邮件将补丁文件发出。当然也可以通过其他方式将补丁文件带走。

```
$ git send-email *.patch
```

Git创建的补丁文件使用了Git扩展格式，因此在导入时为了避免数据遗漏，要使用Git提供的命令而不能使用GNU patch命令。即使要导入的不是Git版本库，也可以使用Git命令，具体操作请参见本书第7篇“第38章 补丁中的二进制文件”中的相关内容。

## 避免引入辅助目录

很多版本控制系统，都要在工作区中引入辅助目录或文件，如SVN要在工作区的每一个子目录下都创建:`:file:`.svn``目录，CVS要在工作区的每一个子目录下都创建:`:file:`CVS``目录。

这些辅助目录如果出现在服务器上，尤其是Web服务器上是非常危险的，因为这些辅助目录下的`:file:`Entries``文件会暴露出目录下的文件列表，让管理员精心配置的禁止目录浏览的努力全部白费。

还有，SVN的`:file:`.svn``辅助目录下还存在文件的原始拷贝，在文件搜索时结果会加倍。如果您曾经在SVN的工作区用过`:command:`grep``命令进行内容查找，就会明白我指的是什么。

Git没有这个问题，不会在子目录下引入讨厌的辅助目录或文件

(`:file:`.gitignore``和`:file:`.gitattributes``文件不算)。当然Git还是要在工作区的顶级目录下创建名为`:file:`.git``的目录(版本库目录)，不过如果你认为唯一的一个`:file:`.git``目录也过于碍眼，可以将其放到工作区之外的任意目录。一旦这么做了，你在执行Git命令时，要通过命令行(`:command:`--git-dir``)或环境变量`:command:`GIT_DIR``为工作区指定版本库目录，甚至还要指定工作区目录。

Git还专门提供了一个`:command:`git grep``命令，这样在工作区根目录下执行查找时，目录`:file:`.git``也不会对搜索造成影响。

关于辅助目录的详细讨论请参见本书第2篇第4.2节中的内容。

## 重写提交说明

很多人可能如我一样，在敲下回车之后，才发现提交说明中出现了错别字，或忘记了写关联的Bug ID。这就需要重写提交说明。

### SVN的解决方案

SVN的提交说明默认是禁止更改的，因为SVN的提交说明属于不受版本控制的属性，一旦修改就不可恢复。我建议SVN的管理员只有在配置了版本库更改的外发邮件通知之后，再开放提交说明更改的功能。我发布于SourceForge上的pySvnManager项目，提供了SVN版本库图形化的钩子管理，会简化管理员的配置工作。

即使SVN管理员启用了允许更改提交说明的设置，修改提交说明也还是挺复杂的，看看下面的命令：

```
$ svn ps --revprop -r <REV> svn:log "new log message..."
```

### Git的解决方案

Git修改提交说明很简单，而且提交说明的修改也是被追踪的。Git修改最新提交的提交说明最为简单，使用一条名为修补提交的命令即可。

```
$ git commit --amend
```

这个命令如果不带“-m”参数，会进入提交说明编辑界面，修改原来的提交说明，直到满意为止。

如果要修改某个历史提交的提交说明，Git也可以实现，但要用到另外一个命令：变基命令。例如要修改<commit-id>所标识提交的提交说明，执行下面的命令，并在弹出的变基索引文件中修改相应提交前面的动作的关键字。

```
$ git rebase -i <commit-id>^
```

关于如何使用交互式变基操作更改历史提交的提交说明，请参见本书第2篇“第12章 改变历史”中的内容。

## 想吃后悔药

假如提交的数据中不小心包含了一个不应该检入的虚拟机文件——大约有1个GB！这时候，您会多么希望这个世界上有后悔药卖啊。

### SVN的解决方案

SVN遇到这个问题该怎么办呢？删除错误加入的大文件，再提交，这样的操作是不能解决问题的。虽然表面上去掉了这个文件，但是它依然存在于历史中。

管理员可能是受影响最大的人，因为他要负责管理服务器的磁盘空间占用及版本库的备份。实际上这个问题也只有管理员才能解决，所以你必须向管理员坦白，让他帮你在服务器端彻底删除错误引入的大文件。我要告诉你的是，对于管理员，这并不是一个简单的活。

1. SVN管理员要是没有历史备份的话，只能从头用`:command:`svnadmin dump``导出整个版本库。
2. 再用`:command:`svndumpfilter``命令过滤掉不应检入的大文件。
3. 然后用`:command:`svnadmin load``重建版本库。

上面的操作描述中省略了一些窍门，因为要把窍门说清楚的话，这本书就不是讲Git，而是讲SVN了。

### Git的解决方案

如果你用Git，一切就会非常简单，而且你也不必去乞求管理员，因为使用Git，每个人都是管理员。

如果是最新的提交引入了不该提交的大文件：`:file:`winxp.img``，操作起来会非常简单，

还是用修补提交命令。

```
$ git rm --cached winxp.img  
$ git commit --amend
```

如果是历史版本，例如是在`<commit-id>`所标识的提交中引入的文件，则需要使用变基操作。

```
$ git rebase -i <commit-id>^
```

执行交互式变基操作抛弃历史提交，版本库还不能立即瘦身，具体原因和解决方案请参见本书第2篇“第14章 Git库管理”中的内容。除了使用变基操作，Git还有更多的武器可以实现版本库的整理操作，具体请参见本书第6篇第35.4节的内容。

## 更好用的提交列表

正确的版本控制系统的使用方法是，一次提交只干一件事：完成一个新功能、修改了一个Bug、或是写完了一节的内容、或是添加了一幅图片，就执行一次提交。而不要在下班时才想起来要提交，那样的话版本控制系统就被降格为文件备份系统了。

但有时在同一个工作区中可能同时在做两件事情，一个是尚未完成的新功能，另外一个是解决刚刚发现的Bug。很多版本控制系统没有提交列表的概念，或者要在命令行指定要提交的文件，或者默认把所有修改内容全部提交，破坏了一个提交干一件事的原则。

### SVN的解决方案

SVN 1.5开始提供了变更列表（change list）的功能，通过引入一个新的命令：`:command:` svn changelist``来实现。但是我从来就没有用过，因为：

- 定义一个变更列表太麻烦。例如不支持将当前所有改动的文件加入列表，也不支持将工作区中的新文件全部加入列表。
- 一个文件不能同时属于两个变更列表。两次变更不许有文件交叉，这样的限制太牵强。
- 变更列表是一次性的，提交之后自动消失。这样的设计没有问题，但是相比定义列表时的繁琐，以及提交时必须指定列表的繁琐，使用变更列表未免得不偿失。
- 再有，因为Subversion的提交不能撤销，如果在提交时忘了提供变更列表名称以针对特定的变更列表进行提交，错误的提交内容将无法补救。

总之，SVN的变更列表尚不如鸡肋，食之无味，弃之不可惜。

### Git的解决方案

Git通过提交暂存区实现对提交内容的定制，非常完美地实现了对工作区的修改内容进行筛选提交：

- 执行`:command:`git add``命令将修改内容加入提交暂存区。执行`:command:`git add -u``命令可以将所有修改过的文件加入暂存区，执行`:command:`git add -A``命令可以将本地删除文件和新增文件都登记到提交暂存区，执行`:command:`git add -p``命令甚至可以对一个文件内的修改进行有选择性的添加。
- 一个修改后的文件被登记到提交暂存区后，可以继续修改，继续修改的内容不会被提交，除非再对此文件再执行一次`:command:`git add``命令。即一个修改的文件可以拥有两个版本，在提交暂存区中有一个版本，在工作区中有另外一个版本。
- 执行`:command:`git commit``命令提交，无须设定什么变更列表，直接将登记在暂存区中的内容提交。
- Git支持对提交的撤消，而且可以撤消任意多次。

只要使用Git，就会时刻在和隐形的提交列表打交道。本书第2篇“第5章 Git暂存区”会详细介绍Git的这一特性，相信你会爱上Git的这个特性。

## 更好的差异比较

Git对差异比较进行了扩展，支持对二进制文件的差异比较，这是对GNU的`:command:`diff``和`:command:`patch``命令的重要补充。还有Git的差异比较除了支持基于行的差异比较外，还支持在一行内逐字比较的方式，当向`:command:`git diff``命令传递`:command:`--word-diff``参数时，就会进行逐字比较。

在上面介绍了工作区的文件修改可能会有两个不同的版本，一个是在提交暂存区，一个是在工作区。因此在执行`:command:`git diff``命令时会遇到令Git新手费解的现象。

- 修改后的文件在执行`:command:`git diff``命令时会看到修改造成的差异。
- 修改后的文件通过`:command:`git add``命令提交到暂存区后，再执行`:command:`git diff``命令会看不到该文件的差异。
- 继续对此文件进行修改，再执行`:command:`git diff``命令，会看到新的修改显示在差异中，而看不到旧的修改。
- 执行`:command:`git diff --cached``命令才可以看到添加到暂存区中的文件所做出的修改。

Git差异比较的命令充满了魔法，本书第5章第5.3节会带您破解Git的diff魔法。一旦您习惯了，就会非常喜欢`:command:`git diff``的这个行为。

## 工作进度保存

如果工作区的修改尚未完成时，忽然有一个紧急的任务，需要从一个干净的工作区开始新的工作，或要切换到别的分支进行工作，那么如何保存当前尚未完成的工作进度呢？

## SVN的解决方案

如果版本库规模不大，最好重新检出一个新的工作区，在新的工作区进行工作。否则，可以执行下面的操作。

```
$ svn diff > /path/to/saved/patch.file  
$ svn revert -R  
$ svn switch <new_branch>
```

在新的分支中工作完毕后，再切换回当前分支，将补丁文件重新应用到工作区。

```
$ svn switch <original_branch>  
$ patch -p1 < /path/to/saved/patch.file
```

但是切记SVN的补丁文件不支持二进制文件，这种操作方法可能会丢失对二进制文件的更改！

## Git 的解决方案

Git提供了一个可以保存和恢复工作进度的命令:`git stash`。这个命令非常方便地解决了这个难题。

在切换到新的工作分支之前，执行`git stash`保存工作进度，工作区会变得非常干净，然后就可以切换到新的分支中了。

```
$ git stash  
$ git checkout <new_branch>
```

新的工作分支修改完毕后，再切换回当前分支，调用`git stash pop`命令则可恢复之前保存的工作进度。

```
$ git checkout <original_branch>  
$ git stash pop
```

本书第2篇“第9章 恢复进度”会为您揭开`git stash`命令的奥秘。

## 代理SVN提交实现移动式办公

使用像SVN一样的集中式版本控制系统，要求使用者和版本控制服务器之间要有网络连接，如果因为出差在外或在家办公访问不到版本控制服务器就无法提交。Git属于分布式版本控

制系统，不存在这样的问题。

当版本控制服务器无法实现从SVN到Git的迁移时，仍然可以使用Git进行工作。在这种情况下，Git作为客户端来操作SVN服务器，实现在移动办公状态下的版本提交（当然是在本地Git库中提交）。当能够连通SVN服务器时，一次性将移动办公状态下的本地提交同步给SVN服务器。整个过程对于SVN来说是透明的，没有人知道你是使用Git在进行提交。

使用Git来操作SVN版本控制服务器的一般工作流程为：

1. 访问SVN服务器，将SVN版本库克隆为一个本地的Git库，一个货真价实的Git库，不过其中包含针对SVN的扩展。

```
$ git svn clone <svn_repos_url>
```

2. 使用Git命令操作本地克隆的版本库，例如提交就使用`:command:`git commit``命令。
3. 当能够通过网络连接到SVN服务器，并想将本地提交同步给SVN服务器时，先获取SVN服务器上最新的提交，再执行变基操作，最后再将本地提交推送给SVN服务器。

```
$ git svn fetch  
$ git svn rebase  
$ git svn dcommit
```

本书第4篇“第26章 Git和SVN协同模型”中会详细介绍这一话题。

## 无处不在的分页器

虽然拥有图形化的客户端，但Git更有效率的操作方式还是命令行操作。使用命令行操作的好处一个是快，另外一个是防止鼠标手的出现。Git的命令行进行了大量的人性化设计，包括命令补全、彩色字符输出等，不过最具特色的还是无处不在的分页器。

在操作其他版本控制系统的命令行时，如果命令的输出超过了一屏，为了能够逐屏显示，需要在命令的后面加上一个管道符号将输出交给一个分页器。例如：

```
$ svn log | less
```

而Git则不用如此麻烦，因为常用的Git的命令都带有一个分页器，当一屏显示不下时启动分页器。分页器默认使用`:command:`less``命令（`:command:`less -FRSX``）进行分页。

因为`:command:`less``分页器在翻屏时使用了vi风格的热键，如果您不熟悉vi的话，可能会遇到麻烦。下面是在分页器中常用的热键：

- 字母q: 退出分页器。
- 字母h: 显示分页器帮助。
- 按空格下翻一页，按字母 b 上翻一页。
- 字母d和u: 分别代表向下翻动半页和向上翻动半页。
- 字母j和k: 分别代表向上翻一行和向下翻一行。
- 如果行太长被截断，可以用左箭头和右箭头使得窗口内容左右滚动。
- 输入/pattern: 向下寻找和pattern匹配的内容。
- 输入?pattern: 向上寻找和pattern匹配的内容。
- 字母n或N: 代表向前或向后继续寻找。
- 字母g: 跳到第一行；字母G: 跳到最后一行；输入数字再加字母g: 则跳转到对应的行。
- 输入!<command>: 可以执行Shell命令。

对于默认未提供分页器的Git命令，例如:`:command:`git status``命令，可以通过下面任一方法启用分页器：

- 在`:command:`git``和子命令（如`:command:`status``）之间插入参数-p或--paginate，为命令启用内建分页器。如：

```
$ git -p status
```

- 设置Git配置变量，设置完毕后运行相应的命令，将启用内建分页器。

```
$ git config --global pager.status true
```

Git命令的分页器支持带颜色的字符输出，对于太长的行则采用截断方式处理（可用左右方向键滚动）。如果不习惯分页器的长行截断模式而希望采用自动折行模式，可以通过下面任一方法进行设置：

- 通过设置LESS环境变量来实现。

```
$ export LESS=FRX
```

- 或者通过定义Git配置变量来改变分页器的默认行为。

```
$ git config --global core.pager 'less -+$LESS -FRX'
```

您有项目托管在sourceforge.net的CVS或SVN服务器上么？或者因为公司的SVN服务器部署在另外一个城市需要经过互联网才能访问？

使用传统的集中式版本控制服务器，如果遇到上面的情况——网络带宽没有保证，那么使用起来一定是慢得让人痛苦不堪。Git作为分布式版本控制系统彻底解决了这个问题，几乎所有的操作都在本地进行，而且还不是一般的快。

还有很多其他的分布式版本控制系统，如Hg、Bazaar等。和这些分布式版本控制系统相比，Git在速度上也有优势，这源自于Git独特的版本库设计。第2篇的相关章节会向您展示Git独特的版本库设计。

其他很多版本控制系统，当输入检出、更新或克隆等命令后，只能双手合十然后望眼欲穿，因为整个操作过程就像是一个黑洞，不知道什么时候才能够完成。而Git在版本库克隆及与版本库同步的时候，能够实时地显示完成的进度，这不但是非常人性化的设计，更体现了Git的智能。Git的智能协议源自于会话过程中在客户端和服务器端各自启用了一个会话的角色，按需传输以及获取进度。

来源：<https://github.com/gotgit/gotgit/blob/master/01-meet-git/020-love-git.rst>

# Linux下安装和使用Git

Git诞生于Linux平台并做为版本控制系统率先服务于Linux核心，因此在Linux安装Git是非常方便的。可以通过不同的方式在Linux上安装Git。一种方法是通过Linux发行版的包管理器安装已经编译好的二进制格式的Git软件包。另外一种方式就是从Git源码开始安装。

## 包管理器方式安装

用Linux发行版的包管理器安装Git，最为简单，但安装的Git可能不是最新的版本。还有一点要注意，就是Git软件包在有的Linux发行版中可能不叫做git，而叫做git-core。这是因为在Git之前就有一款叫做GNU交互工具（GNU Interactive Tools）的GNU软件在有的Linux发行版（Debian lenny）中已经占用了git的名称。为了以示区分，作为版本控制系统的Git，其软件包在这些平台就被命名为git-core。不过因为作为版本控制系统的Git太有名了，最终导致在一些Linux发行版的最新版本中，将GNU Interactive Tools软件包由git改名为gnuit，将git-core改名为git。所以在下面介绍的在不同的Linux发行版中安装Git时，会看到有git和git-core两个不同的名称。

- Ubuntu 10.10(maverick)或更新版本，Debian(squeeze)或更新版本：

```
$ sudo aptitude install git  
$ sudo aptitude install git-doc git-svn git-email gitk
```

其中git软件包包含了大部分Git命令，是必装的软件包。

软件包git-svn、git-email、gitk本来也是Git软件包的一部分，但是因为有着不一样的软件包依赖（如更多perl模组，tk等），所以单独作为软件包发布。

软件包git-doc则包含了Git的HTML格式文档，可以选择安装。如果安装了Git的HTML格式的文档，则可以通过执行：`command: `git help -w <sub-command>``命令，自动用Web浏览器打开相关子命令<sub-command>的HTML帮助。

- Ubuntu 10.04(lucid)或更老版本，Debian(lenny)或更老版本：

在老版本的Debian中，软件包git实际上是Gnu Interactive Tools，而非作为版本控制系统的Git。做为版本控制系统的Git在软件包git-core中。

```
$ sudo aptitude install git-core  
$ sudo aptitude install git-doc git-svn git-email gitk
```

- RHEL、Fedora、CentOS：

```
$ yum install git  
$ yum install git-svn git-email gitk
```

其他发行版安装Git的过程和上面介绍的方法向类似。Git软件包在这些发行版里或称为git，或称为git-core。

## 从源代码开始安装

访问Git的官方网站：<http://git-scm.com/>。下载Git源码包，例如：`:file:`git-1.7.3.5.tar.bz2``。

- 展开源码包，并进入到相应的目录中。

```
$ tar -jxvf git-1.7.3.5.tar.bz2  
$ cd git-1.7.3.5/
```

- 安装方法写在`:file:`INSTALL``文件当中，参照其中的指示完成安装。下面的命令将Git安装在`:file:`/usr/local/bin``中。

```
$ make prefix=/usr/local all  
$ sudo make prefix=/usr/local install
```

- 安装Git文档（可选）。

编译的文档主要是HTML格式文档，方便通过`:command:`git help -w <sub-command>``命令查看。实际上即使不安装Git文档，也可以使用man手册查看Git帮助，使用命令`:command:`git help <sub-command>``或者`:command:`git <sub-command> --help``。

编译文档依赖asciidoc，因此需要先安装asciidoc（如果尚未安装的话），然后编译文档。在编译文档时要花费很多时间，要有耐心。

```
$ make prefix=/usr/local doc info  
$ sudo make prefix=/usr/local \  
install-doc install-html install-info
```

安装完毕之后，就可以在`:file:`/usr/local/bin``下找到`:program:`git``命令。

## 从Git版本库进行安装

如果在本地克隆一个Git版本库，就可以用版本库同步的方式获取最新版本的Git，这样在下载不同版本的Git源代码时实际上采用了增量方式，会非常的节省时间和空间。当然使用这种方法的前提是已经用其他方法安装好了Git。

- 克隆Git版本库到本地。

```
$ git clone git://git.kernel.org/pub/scm/git/git.git  
$ cd git
```

- 如果本地已经克隆过一个Git版本库，直接在工作区中更新，以获得最新版本的Git。

```
$ git pull
```

- 执行清理工作，避免前一次编译的遗留文件造成影响。注意下面的操作将丢弃本地对Git代码的改动。

```
$ git clean -fdx  
$ git reset --hard
```

- 查看Git的里程碑，选择最新的版本进行安装。例如v1.7.3.5。

```
$ git tag  
...  
v1.7.3.5
```

- 检出该版本的代码。

```
$ git checkout v1.7.3.5
```

- 执行安装。例如安装到:file:`/usr/local` 目录下。

```
$ make prefix=/usr/local all doc info  
$ sudo make prefix=/usr/local install \  
install-doc install-html install-info
```

我在撰写本书的过程中，就通过Git版本库的方式安装，在:file:`/opt/git` 目录下安装了多个不同版本的Git，以测试Git的兼容性。使用类似下面的脚本，可以批量安装不同版本的Git。

```
#!/bin/sh

for ver in      \
    v1.5.0      \
    v1.7.3.5    \
    v1.7.4-rc1  \
; do
    echo "Begin install Git $ver.";
    git reset --hard
    git clean -fdx
    git checkout $ver || exit 1
    make prefix=/opt/git/$ver all && \
    sudo make prefix=/opt/git/$ver install || exit 1
    echo "Installed Git $ver."
done
```

## 命令补齐

Linux的shell环境（bash）通过:`:file:`bash-completion``软件包提供命令补齐功能，能够实现在录入命令参数时按一下或两下TAB键，实现参数的自动补齐或提示。例如输入:`:command:`git com``后按下TAB键，会自动补齐为`:command:`git commit``。

通过包管理器方式安装Git，一般都已经为Git配置好了自动补齐，但是如果是以源码编译方式安装Git，就需要为命令补齐多做些工作。

- 将Git源码包中的命令补齐脚本复制到`:file:`bash-completion``对应的目录中。

```
$ cp contrib/completion/git-completion.bash \
    /etc/bash_completion.d/
```

- 重新加载自动补齐脚本，使之在当前shell中生效。

```
$ . /etc/bash_completion
```

- 为了能够在终端开启时自动加载`:file:`bash_completion``脚本，需要在本地配置文件`:file:`~/.bash_profile``或全局文件`:file:`/etc/bashrc``文件中添加下面的内容。

```
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi
```

## 中文支持

Git的本地化做的并不完善，命令的输出以及命令的帮助还只能输出英文，也许在未来版本会使用gettext实现本地化，就像目前对git-gui命令所做的那样。

使用中文的用户最关心的问题还有：是否可以在提交说明中使用中文？是否可以使用中文文件名或者目录名？是否可以使用中文来命名分支或者里程碑？简单的说，可以在提交说明中使用中文，但是若使用非UTF-8字符集，则需要为Git做些设置。至于使用中文来命名文件、目录或引用，只有在使用UTF-8字符集的环境下才可以（Windows用户使用Cygwin），否则尽量避免使用。

### UTF-8字符集

Linux平台的中文用户一般会使用utf-8字符集，Git在utf-8字符集下可以工作的非常好。

- 在提交时，可以在提交说明中输入中文。
- 显示提交历史，能够正常显示提交说明中的中文字符。
- 可以添加中文文件名的文件，并可以在同样utf-8字符集的Linux环境中克隆及检出。
- 可以创建带有中文字符的里程碑名称。

但是默认设置下，带有中文文件名的文件，在工作区状态输出、查看历史更改概要、以及在补丁文件中，文件名不能正确显示为中文，而是用若干8进制编码来显示中文，如下：

```
$ git status -s  
?? "\350\257\264\346\230\216.txt"
```

通过设置变量:`:command:`core.quotePath``为`false`，就可以解决中文文件名在这些Git命令输出中的显示问题。

```
$ git config --global core.quotePath false  
$ git status -s  
?? 说明.txt
```

### GBK字符集

但如果Linux平台采用非UTF-8字符集，例如用zh\_CN.GBK字符集编码（某些Linux发行版），就要另外再做些工作了。

- 设置提交说明显示所使用的字符集为gbk，这样使用`:command:`git log``查看提交说明才能够正确显示其中的中文。

```
$ git config --global i18n.logOutputEncoding gbk
```

- 设置录入提交说明时所使用的字符集，以便在commit对象中对字符集正确标注。

Git在提交时并不会对提交说明进行从GBK字符集到UTF-8的转换，但是可以在提交说明中标注所使用的字符集，因此在非UTF-8字符集的平台录入中文，需要用下面指令设置录入提交说明的字符集，以便在commit对象中嵌入正确的编码说明。

```
$ git config --global i18n.commitEncoding gbk
```

来源: <https://github.com/gotgit/gotgit/blob/master/01-meet-git/035-install-on-linux.rst>

# Mac OS X下安装和使用Git

Mac OS X 被称为最人性化的操作系统，工作在Mac上是件非常惬意的事情，工作中怎能没有Git？

## 以二进制发布包的形式安装

Git在Mac OS X中也有好几种安装方法。最为简单的方式是安装:`:file:`.dmg``格式的安装包。

访问git-osx-installer的官方网站：<http://code.google.com/p/git-osx-installer/>，下载Git安装包。安装包带有`:file:`.dmg``扩展名，是苹果磁盘镜像（Apple Disk Image）格式的软件发布包。从官方网站上下载文件名类似`:file:`git-<version>-<arch>-leopard.dmg``的安装包文件，例如：`:file:`git-1.7.3.5-x86_64-leopard.dmg``是64位的安装包，`:file:`git-1.7.3.5-i386-leopard.dmg``是32位的安装包。建议选择64位的软件包，因为Mac OS X 10.6 雪豹（或更新版本）完美的兼容32位和64位（开机按住键盘数字3和2进入32位系统，按住6和4进入64位系统），即使在核心处于32位架构下，也可以放心的运行64位软件包。

苹果的`:file:`.dmg``格式的软件包实际上是一个磁盘映像，安装起来非常方便，点击该文件就直接挂载到Finder中，并打开，如图3-1所示。

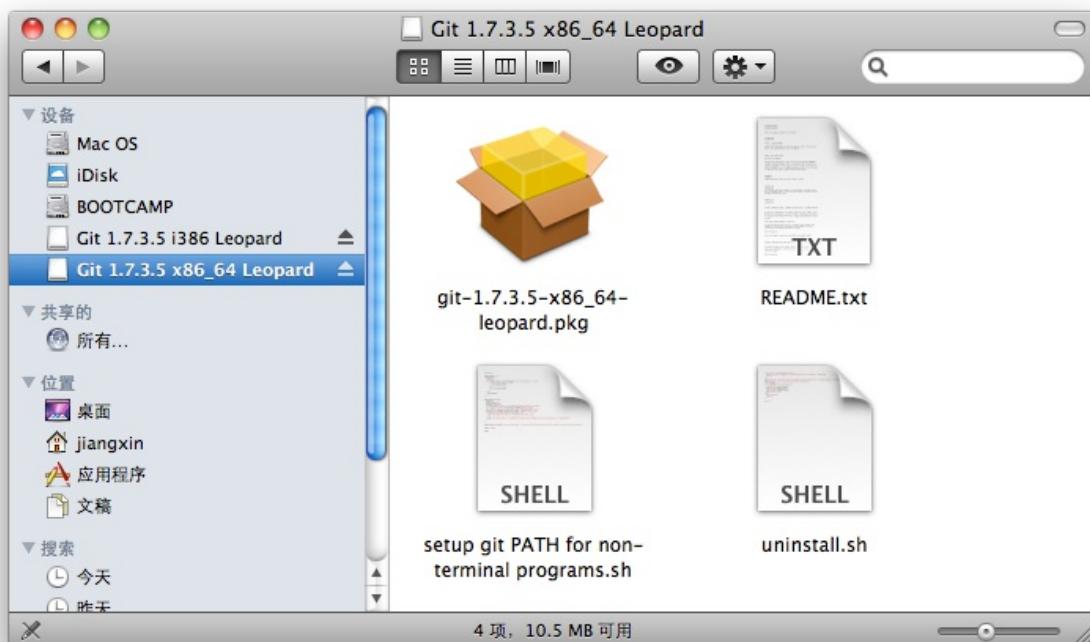


图3-1：在 Mac OS X 下打开 `.dmg` 格式磁盘镜像

其中带有一个正在解包图标的文件（扩展名为`:file:`.pkg``）是Git的安装程序，另外的两个脚本程序，一个用于应用的卸载（`:file:`uninstall.sh``），另外一个带有长长文件名的脚本可以在Git安装后执行的，为非终端应用注册Git的安装路径，因为Git部署在标准的系统路径之外`:file:`/usr/local/git/bin``。

点击扩展名为`:file:`.pkg``的安装程序，开始Git的安装，根据提示按步骤完成安装，如图3-2所示。



图3-2：在 Mac OS X 下安装 Git。

安装完毕，git会被安装到`:file:`/usr/local/git/bin``目录下。重启终端程序，才能让`:file:`/etc/paths.d/git``文件为PATH环境变量中添加的新路径注册生效。然后就可以在终端中直接运行`:program:`git``命令了。

## 安装Xcode

App Store安装Xcode

安装完毕，可以运行下面命令查看Xcode安装路径。

```
$ xcode-select --print-path  
/Applications/Xcode.app/Contents/Developer
```

路径并不在PATH中，可以运行:command:``xcrun``调用在Xcode路径中的Git工具。

```
$ xcrun git --version  
git version 1.7.9.6 (Apple Git-31.1)
```

为了方便在终端命令行下运行Git，可以用

```
$ cat /etc/paths.d/xcode  
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchai  
/Applications/Xcode.app/Contents/Developer/usr/bin
```

或者打开 Xcode, Preference → Downloads → Components → Command Line Tools (install)

Mac OS X基于Unix内核，因此也可以很方便的通过源码编译的方式进行安装，但是缺省安装的Mac OS X缺乏相应的开发工具，需要安装苹果提供的Xcode软件包。在Mac随机附送的光盘（Mac OS X Install DVD）的可选安装文件夹下就有Xcode的安装包（如图3-3所示），通过随机光盘安装Xcode可以省去了网络下载的麻烦，要知道Xcode有3GB以上。



图3-3：在Mac OS X下安装Xcode。

## 使用Homebrew安装Git

Mac OS X有好几个包管理器实现对一些开源软件在Mac OS X上的安装和升级进行管理。有传统的MacPort、Fink，还有更为简单易用的Homebrew。下面就介绍一下如何通过Homebrew包管理器，以源码包编译的方式安装Git。

Homebrew用Ruby语言开发，支持千余种开源软件在Mac OS X中的部署和管理。Homebrew项目托管在Github上，网址为：<https://github.com/mxcl/homebrew>。

首先是安装Homebrew，执行下面的命令：

```
$ ruby -e \
  "$(curl -fsSL https://gist.github.com/raw/323731/install_homebrew.rb)"
```

安装完成后，Homebrew的主程序安装在:`/usr/local/bin/brew`，在目录`/usr/local/Library/Formula/`下保存了所有Homebrew支持的软件的安装指引文件。

运行`brew`安装Git，使用下面的命令。

```
$ brew install git
```

使用Homebrew方式安装，Git被安装在`/usr/local/Cellar/git/1.7.3.5`，可执行程序自动在`/usr/local/bin`目录下创建符号连接，可以直接在终端程序中访问。

通过`brew list`命令可以查看安装的开源软件包。

```
$ brew list  
git
```

也可以查看某个软件包安装的详细路径和安装内容。

```
$ brew list git  
/usr/local/Cellar/git/1.7.3.5/bin/gitk  
...
```

## 从Git源码进行安装

如果需要安装历史版本的Git或是安装尚在开发中的未发布版本的Git，就需要从源码安装或通过克隆Git源码库进行安装。既然Homebrew就是通过源码编译方式安装Git的，那么也应该可以直接从源码进行安装，但是使用Homebrew安装Git和直接通过Git源码安装并不完全等同，例如Homebrew安装Git的过程中，是通过下载已经编译好的Git文档包进行安装，而非从头对文档进行编译。

直接通过源码安装Git包括文档，遇到主要的问题就是文档的编译，因为Git文档编译所需要的的相关工具没有在Xcode中提供。但是这些工具可以通过Homebrew进行安装。下面工具软件的安装过程可能会遇到一些小麻烦，不过大多可以通过参考命令输出予以解决。

```
$ brew install asciidoc  
$ brew install docbook2x  
$ brew install xmlto
```

当编译源码及文档的工具部署完全后，就可以通过源码编译Git。

```
$ make prefix=/usr/local all doc info  
$ sudo make prefix=/usr/local install \  
install-doc install-html install-info
```

## 命令自动补齐

Git通过bash-completion软件包实现命令补齐，在Mac OS X下可以通过Homebrew进行安装。

```
$ brew search completion  
bash-completion  
$ brew install bash-completion  
...  
Add the following lines to your `~/.bash_profile` file:  
if [ -f $(brew --prefix)/etc/bash_completion ]; then  
  . $(brew --prefix)/etc/bash_completion  
fi  
...
```

根据bash-completion安装过程中的提示，修改文件:`file: `~/.bash_profile``文件，并在其中加入如下内容，以便在终端加载时自动启用命令补齐。

```
if [ -f $(brew --prefix)/etc/bash_completion ]; then  
  . $(brew --prefix)/etc/bash_completion  
fi
```

将Git的命令补齐脚本拷贝到bash-completion对应的目录中。

```
$ cp contrib/completion/git-completion.bash \  
$(brew --prefix)/etc/bash_completion.d/
```

不用重启终端程序，只需要运行下面的命令，即可立即在当前的shell中加载命令补齐。

```
. $(brew --prefix)/etc/bash_completion
```

## 其他辅助工具的安装

本书中还会用到一些常用的GNU或其他开源软件，在Mac OS X下也可以通过Homebrew进行安装。这些软件包有：

- gnupg: 数字签名和加密工具。在为Git版本库建立签名里程碑时会用到。
- md5sum: 生成MD5或SHA1摘要。在研究Git版本库中的对象过程中会用到。
- cvs2svn: CVS版本库迁移到SVN或Git的工具。在版本库迁移时会用到。
- stgit: Git的补丁和提交管理工具。
- quilt: 一种补丁管理工具。在介绍Topgit时用到。

在Mac OS X下能够使用到的Git图形工具有除了Git软件包自带的:`:command:`gitk`` 和`:command:`git gui``之外，还可以安装GitX。下载地址：

- GitX的原始版本：<http://gitx.frim.nl/>。
- 或GitX的一个分支版本，提供增强的功能：<https://github.com/brotherbard/gitx/downloads>

Git的图形工具一般需要在本地克隆版本库的工作区中执行，为了能和Mac OS X有更好的整合，可以安装插件实现和Finder的整合。在git-osx-installer的官方网站：<http://code.google.com/p/git-osx-installer/>，有两个以`:file:`OpenInGitGui-`` 和`:file:`OpenInGitX-`` 为前缀的软件包，可以分别实现和`:command:`git gui`` 以及`:command:`gitx`` 的整合：在Finder中进入工作区目录，点击对应插件的图标，启动`:command:`git gui`` 或`:command:`gitx``。

## 中文支持

由于Mac OS X采用Unix内核，在中文支持上和Linux相近，请参照前面介绍Git在Linux下安装中3.1.5节相关内容。

来源：<https://github.com/gotgit/gotgit/blob/master/01-meet-git/040-install-on-mac.rst>

# Windows下安装和使用Git（Cygwin篇）

在Windows下安装和使用Git有两个不同的方案，通过安装msysGit或者通过安装Cygwin来使用Git。在这两种不同的方案下，Git的使用和在Linux下使用完全一致。再有一个就是基于msysGit的图形界面工具——TortoiseGit，也就是在CVS和SVN时代就已经广为人知的Tortoise系列软件的Git版本。TortoiseGit提供和资源管理器的整合，提供Git操作的图形化界面。

先介绍通过Cygwin来使用Git的原因，不是因为这是最便捷的方法，如果需要在Windows快速安装和使用Git，下节介绍的msysGit才是。之所以将Cygwin放在前面介绍是因为本书在介绍Git原理部分以及介绍其他Git相关软件时用到了大量的开源工具，这些开源工具在Cygwin下很容易获得，而msysGit的MSYS（Minimal SYSTEM，最简系统）则不能满足我们的需要。因此我建议使用Windows平台的读者在跟随本书学习Git的过程中，首选Cygwin，当完成Git的学习后，无论是msysGit或者TortoiseGit也都会应对自足。

Cygwin是一款伟大的软件，通过一个小小的DLL（cygwin1.dll）建立Linux和Windows系统调用及API之间的转换，实现了Linux下绝大多数软件到Windows的迁移。Cygwin通过cygwin1.dll所建立的中间层和诸如VMWare、VirtualBox等虚拟机软件完全不同，不会对系统资源进行独占。像VMWare等虚拟机，只要启动一个虚拟机（操作系统），即使不在其中执行任何命令，同样会占用大量的系统资源：内存、CPU时间等等。

Cygwin还提供了一个强大易用的包管理工具（setup.exe），实现了几千个开源软件包在Cygwin下便捷的安装和升级，Git就是Cygwin下支持的几千个开源软件中的一员。

我对Cygwin有着深厚的感情，Cygwin让我在Windows平台能用Linux的方式更有效率的做事，使用Linux风格的控制台替换Windows黑乎乎的、冰冷的、由`:command:`cmd.exe``提供的命令行。Cygwin帮助我逐渐摆脱对Windows的依赖，当我完全转换到Linux平台时，没有感到一丝的障碍。

## 安装Cygwin

安装Cygwin非常简单，访问其官方网站<http://www.cygwin.com/>，下载安装程序——一个只有几百KB的：`:program:`setup.exe``，即可开始安装。

安装过程会让用户选择安装模式，可以选择网络安装、仅下载，或者通过本地软件包缓存（在安装过程自动在本地目录下建立软件包缓存）进行安装。如果是第一次安装Cygwin，因为本地尚没有软件包缓存，当然只能选择从网络安装，如图3-4所示。

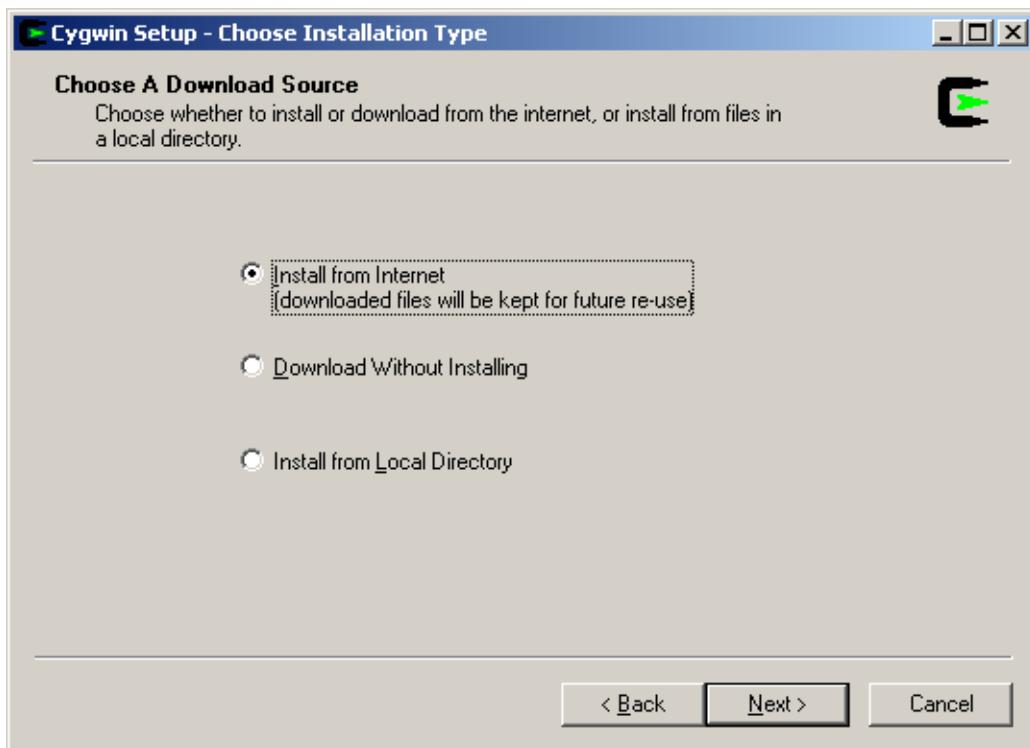


图3-4：选择安装模式

接下来，Cygwin询问安装目录，默认为:`file:`C:\\cygwin``，如图3-5所示。这个目录将作为Cygwin shell环境的根目录（根卷），Windows的各个盘符将挂载在根卷一个特殊目录之下。

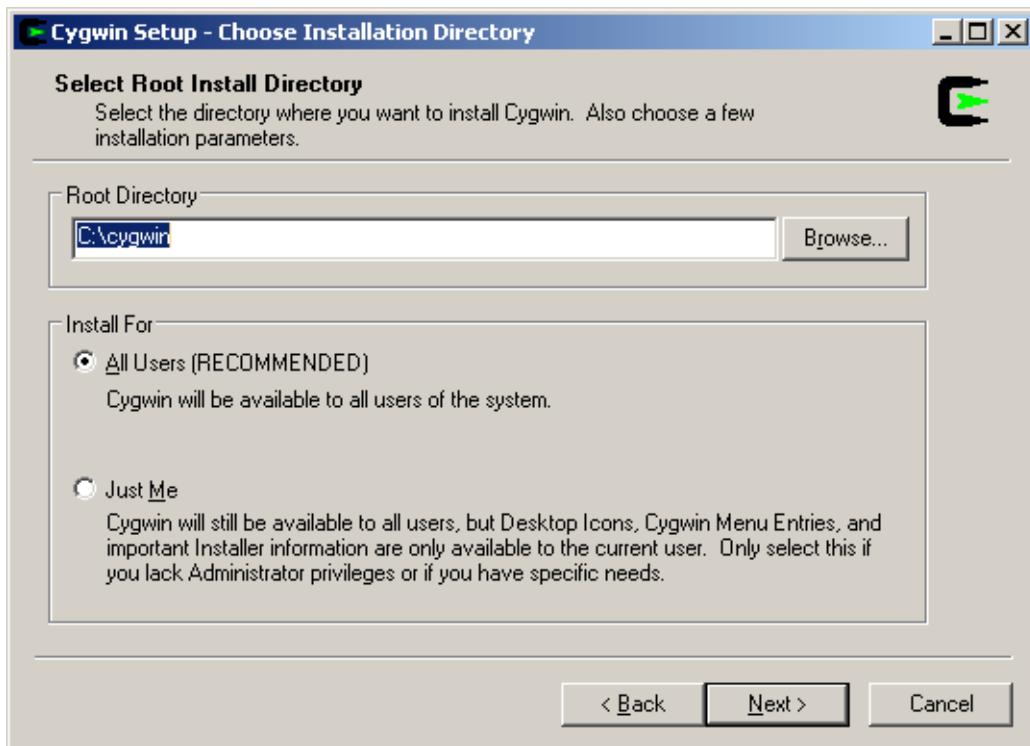


图3-5：选择安装目录

询问本地软件包缓存目录，默认是:`program:`setup.exe``所处的目录，如图3-6所示。

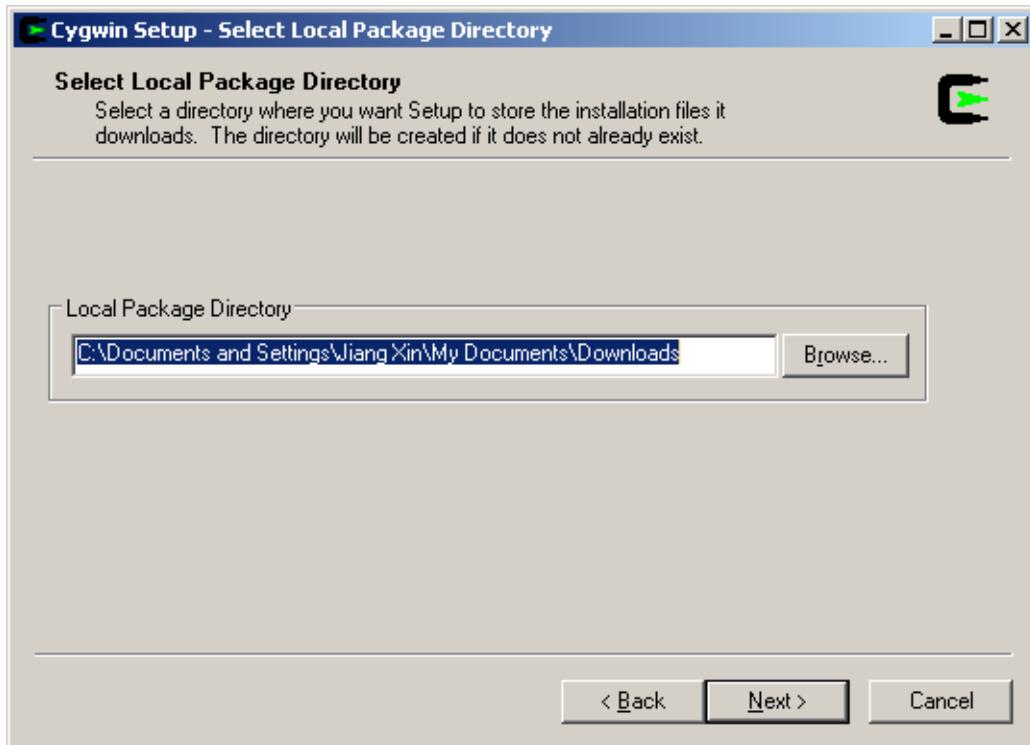


图3-6：选择本地软件包缓存目录

询问网络连接方式，是否使用代理等，如图3-7所示。默认会选择第一项：“直接网络连接”。如果一个团队有很多人要使用Cygwin，架设一个能够提供软件包缓存的HTTP代理服务器会节省大量的网络带宽和节省大把的时间。用Debian的apt-cacher-ng就可以非常简单的搭建一个软件包代理服务器。图3-7显示的就是我在公司内网安装Cygwin时使用了我们公司内网的服务器bj.ossxp.com做为HTTP代理的截图，端口设置为9999，因为这是apt-cacher-ng的默认端口。

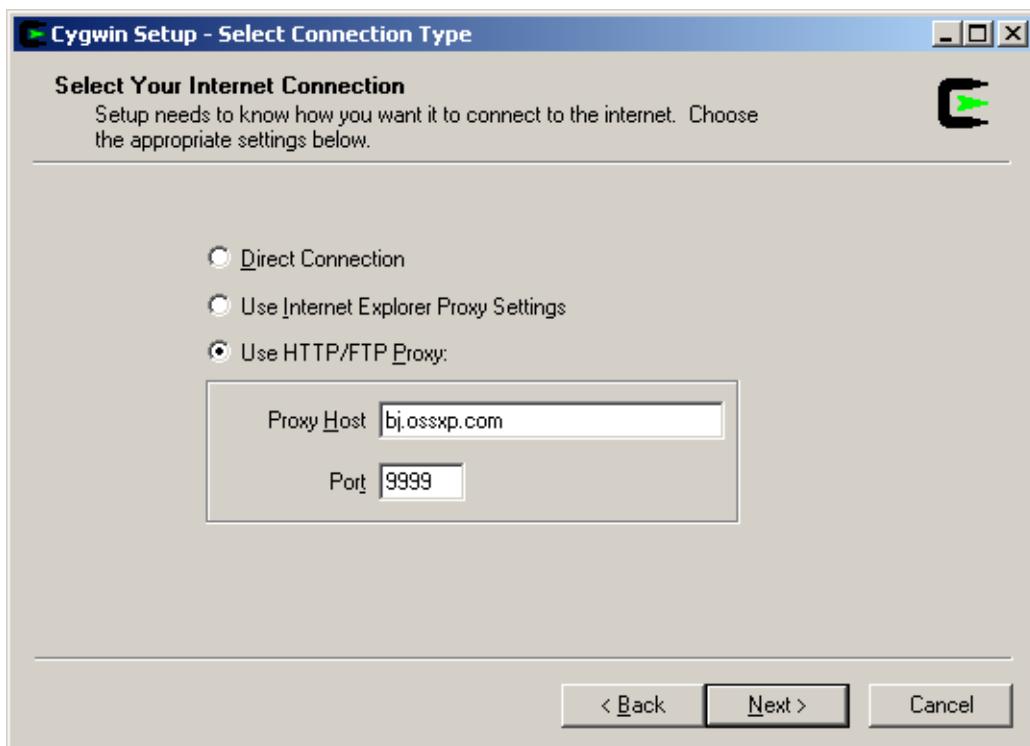


图3-7：是否使用代理下载Cygwin软件包

选择一个Cygwin源，如图3-8所示。如果在上一个步骤选择了使用HTTP代理服务器，就必须选择HTTP协议的Cygwin源。

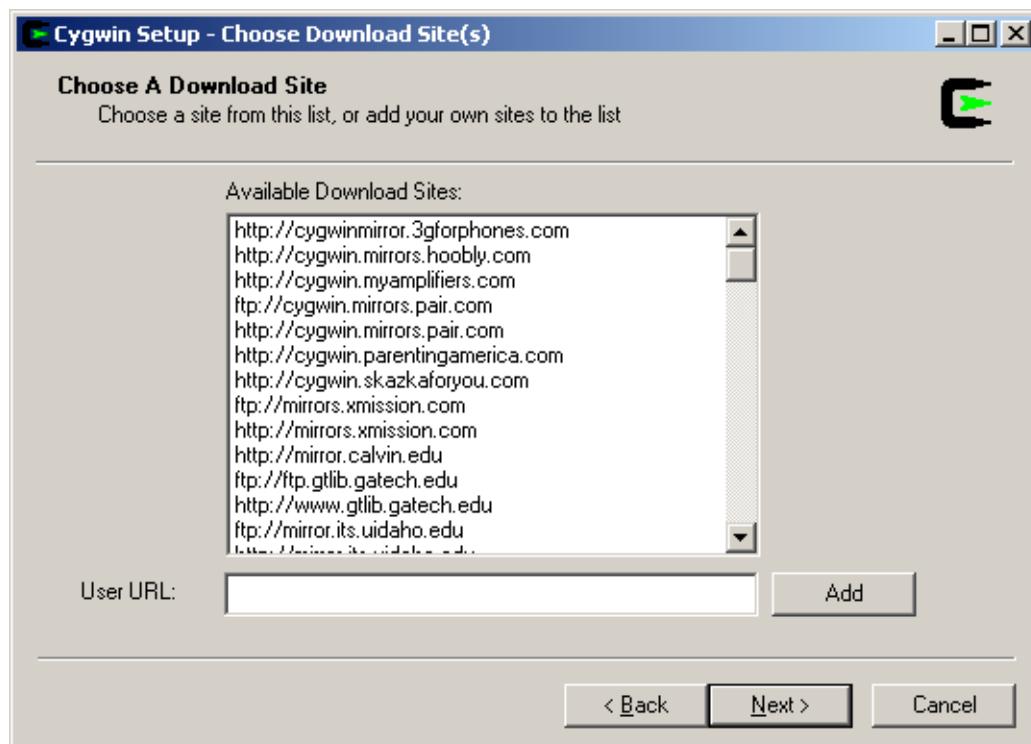


图3-8：选择Cygwin源

接下来就会从所选的Cygwin源下载软件包索引文件，然后显示软件包管理器界面，如图3-9所示。

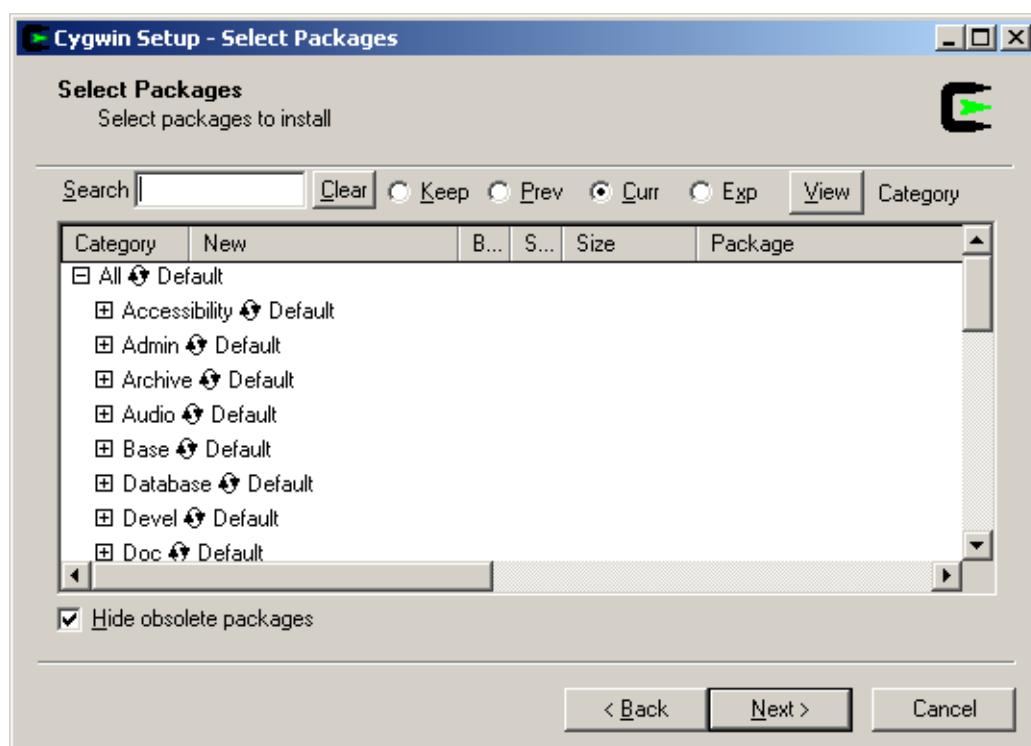


图3-9: Cygwin软件包管理器

Cygwin的软件包管理器非常强大和易用（如果习惯了其界面）。软件包归类于各个分组中，点击分组前的加号就可以展开分组。在展开的Admin分组中，如图3-10所示（这个截图不是首次安装Cygwin的截图），有的软件包如:program:`libattr1`已经安装过了，因为没有新版本而标记为“Keep”（保持）。至于没有安装过并且不准备安装的软件包则标记为“Skip”（跳过）。

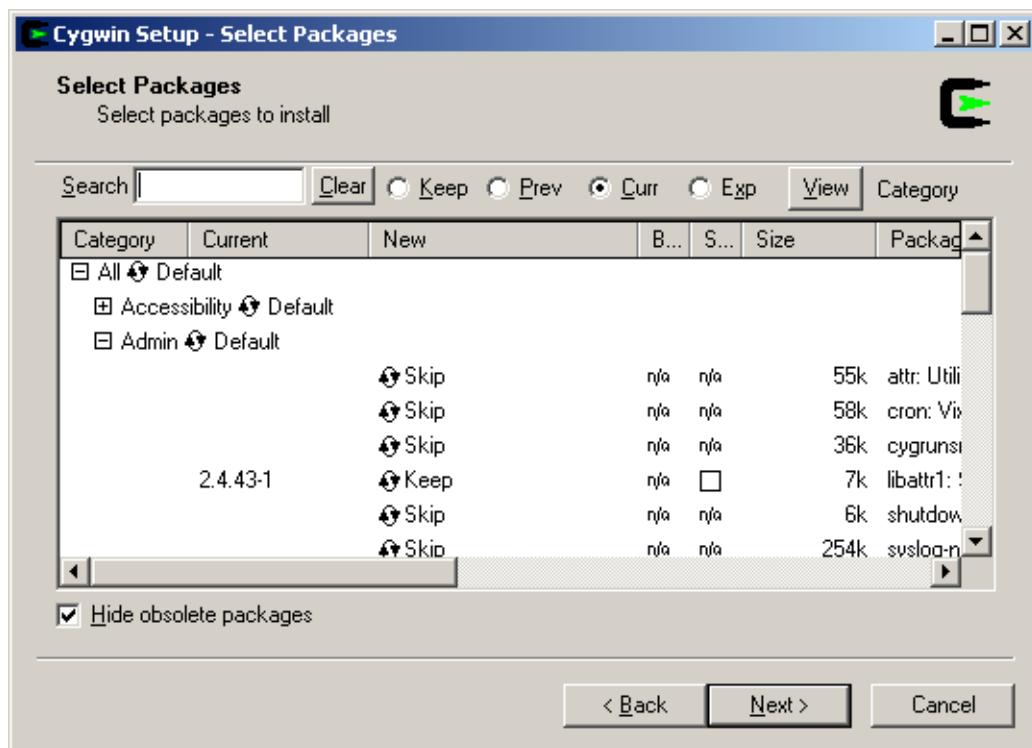


图3-10: Cygwin 软件包管理器展开分组

鼠标点击分组名称后面动作名称（文字“Default”），会进行软件包安装动作的切换。例如图3-11，将Admin分组的安装动作由“Default”（默认）切换为“Install”（安装），会看到Admin分组下的所有软件包都标记为安装（显示具体要安装的软件包版本号）。也可以通过鼠标点击，单独的为软件包进行安装动作的设定，可以强制重新安装、安装旧版本、或者不安装。

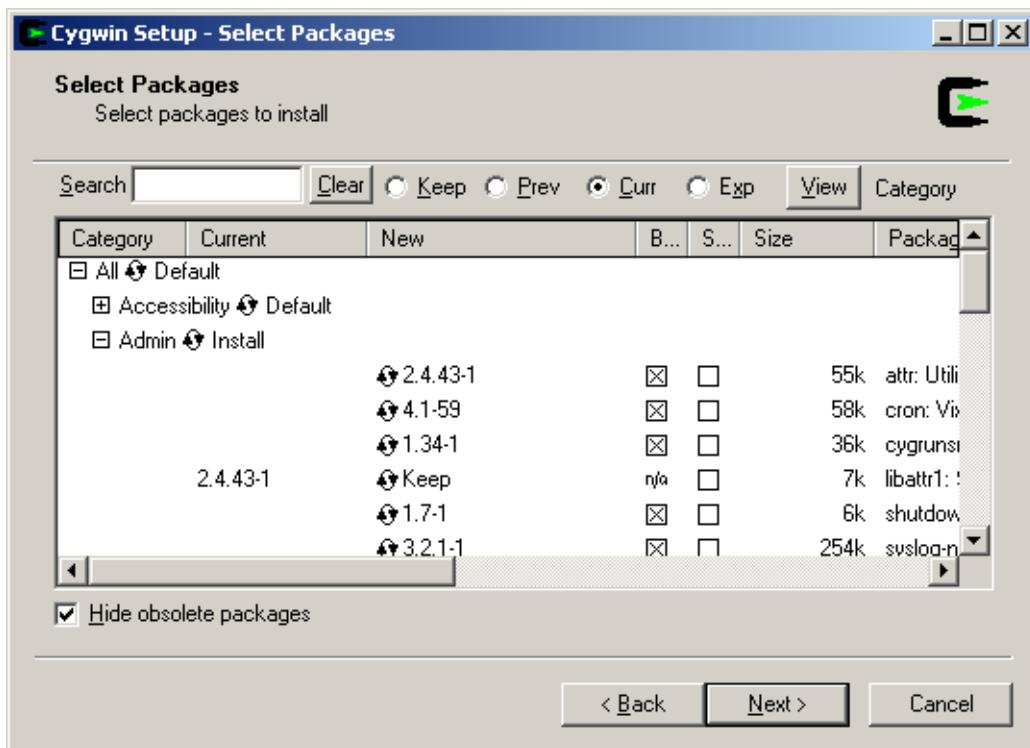


图3-11：Cygwin软件包管理器展开分组

当通过软件包管理器对要安装的软件包定制完毕后，点击下一步，开始下载软件包、安装软件包和软件包后处理，直至完成安装。根据选择的软件包的多少，网络情况以及是否架设有代理服务器，首次安装Cygwin的时间可能从几分钟到几个小时不等。

## 安装Git

默认安装的Cygwin没有安装Git软件包。如果在首次安装过程中忘记通过包管理器选择安装Git或其他相关软件包，可以在安装后再次运行Cygwin的安装程序：`program`setup.exe``。当再次进入Cygwin包管理器界面时，在搜索框中输入git。如图3-12所示。

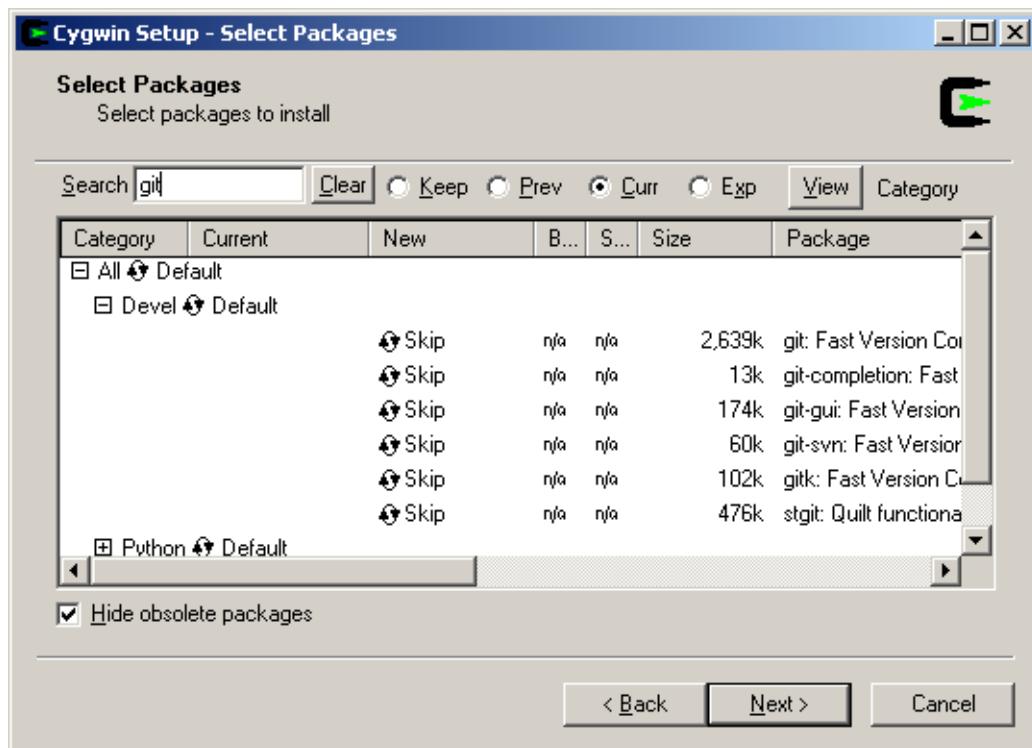


图3-12: Cygwin软件包管理器中搜索git

从图3-12中看出在Cygwin中包含了很多和Git相关的软件包，把这些Git相关的软件包都安装吧，如图3-13所示。

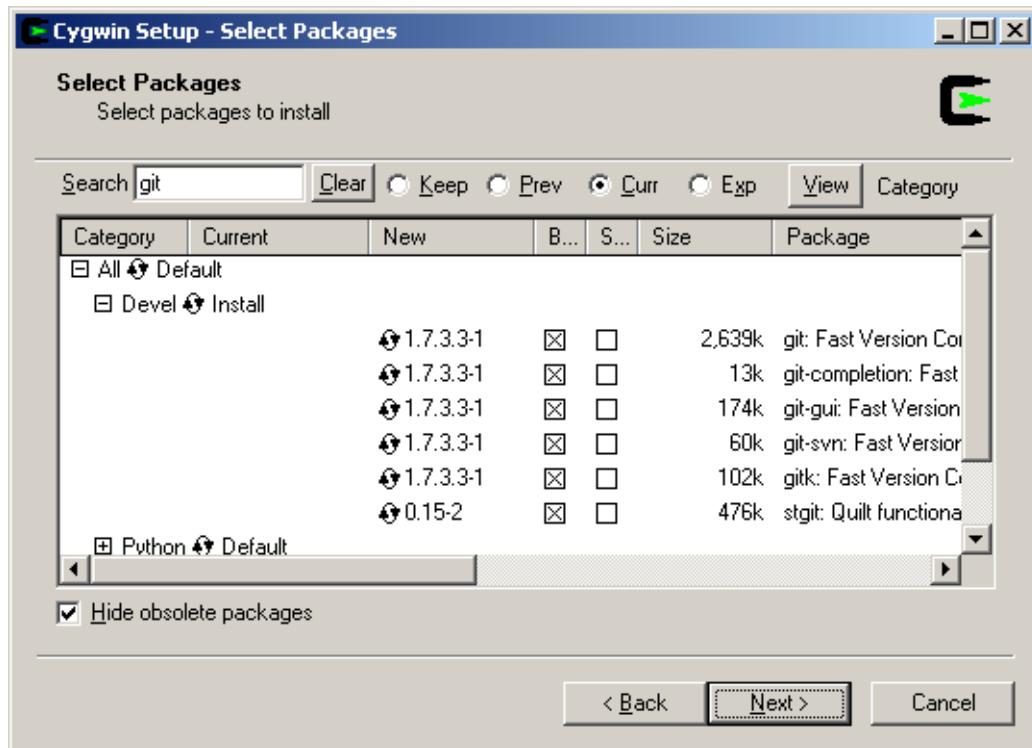


图3-13: Cygwin软件包管理器中安装git

需要安装的其他软件包：

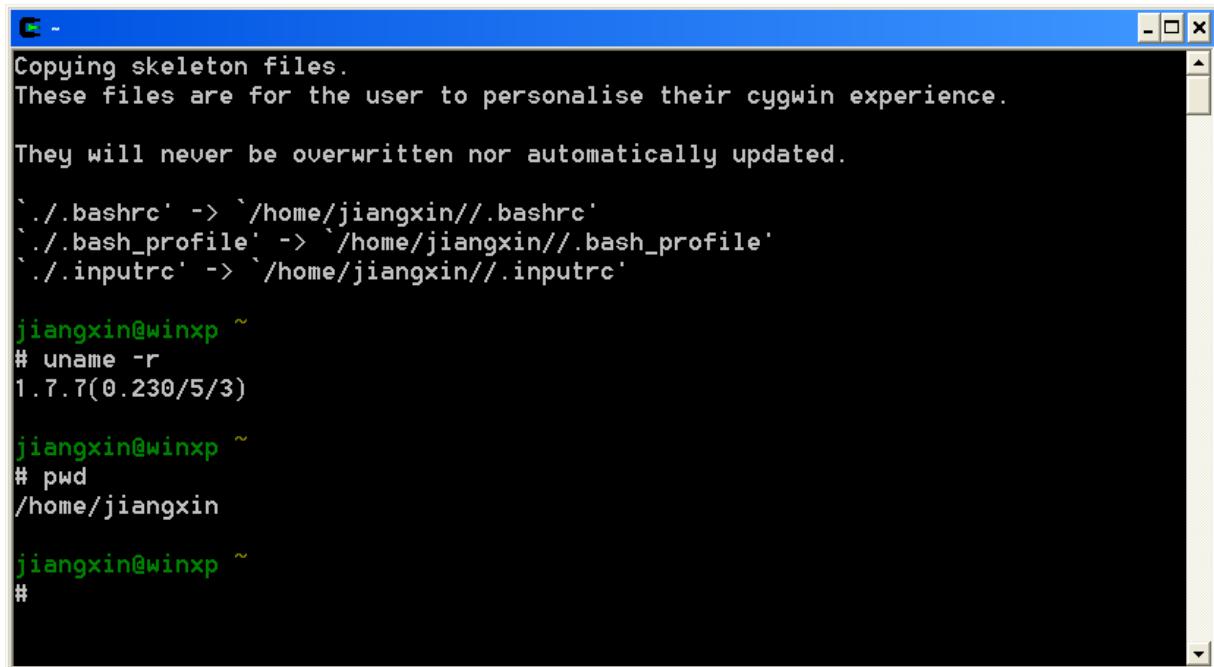
- **git-completion:** 提供Git命令自动补齐功能。安装该软件包会自动安装依赖的bash-

completion软件包。

- openssh: SSH客户端，提供Git访问ssh协议的版本库。
- vim: 是Git缺省的编辑器。

## Cygwin的配置和使用

运行Cygwin，就会进入shell环境中，见到熟悉的Linux提示符。如图3-14所示。



The screenshot shows a Cygwin terminal window with a blue title bar containing the letter 'C'. The window displays the following text:

```
Copying skeleton files.  
These files are for the user to personalise their cygwin experience.  
They will never be overwritten nor automatically updated.  
./.bashrc' -> `/home/jiangxin//.bashrc'  
./.bash_profile' -> `/home/jiangxin//.bash_profile'  
./.inputrc' -> `/home/jiangxin//.inputrc'  
  
jiangxin@winxp ~  
# uname -r  
1.7.7(0.230/5/3)  
  
jiangxin@winxp ~  
# pwd  
/home/jiangxin  
  
jiangxin@winxp ~  
#
```

图3-14: 运行 Cygwin

显示Cygwin中安装的软件包的版本，可以通过执行:`program: `cygcheck``命令来查看，例如查看cygwin软件包本身的版本：

```
$ cygcheck -c cygwin  
Cygwin Package Information  
Package          Version       Status  
cygwin           1.7.7-1     OK
```

## 如何访问Windows的磁符

刚刚接触Cygwin的用户遇到的第一个问题就是Cygwin如何访问Windows的各个磁盘目录，以及在Windows平台如何访问Cygwin中的目录？

执行:`program: `mount``命令，可以看到Windows下的盘符映射到:`file: `/cyg drive``特殊目录下。

```
$ mount
```

```
C:/cygwin/bin on /usr/bin type ntfs (binary,auto)
C:/cygwin/lib on /usr/lib type ntfs (binary,auto)
C:/cygwin on / type ntfs (binary,auto)
C: on /cygdrive/c type ntfs (binary,posix=0,user,noumount,auto)
D: on /cygdrive/d type ntfs (binary,posix=0,user,noumount,auto)
```

也就是说在Windows下的:`:file:`C:\\Windows``目录，在Cygwin以路径:`:file:`/cygdrive/c/Windows``进行访问。实际上Cygwin提供一个命令:`:program:`cygpath``实现Windows平台和Cygwin之间目录名称的变换。如下：

```
$ cygpath -u C:\\Windows
/cygdrive/c/Windows

$ cygpath -w ~/
C:\\cygwin\\home\\jiangxin\\
```

从上面的示例也可以看出，Cygwin下的用户主目录（即:`:file:`/home/jiangxin/``）相当于Windows下的:`:file:`C:\\cygwin\\home\\jiangxin\\``目录。

## 用户主目录不一致的问题

如果其他某些软件（如msysGit）为Windows设置了HOME环境变量，会影响到Cygwin中用户主目录的设置，甚至造成在Cygwin中不同命令有不同的用户主目录的设置。例如：Cygwin下Git的用户主目录设置为“/cygdrive/c/Documents and Settings/jiangxin”，而SSH客户端软件的主目录为“/home/jiangxin”，这会造成用户的困惑。

出现这种情况，是因为Cygwin确定用户主目录有几个原则，依照顺序确定主目录。首先查看系统的HOME环境变量，其次查看:`:file:`/etc/passwd``中为用户设置的主目录。有的软件遵照这个原则，而有些Cygwin应用如ssh，却没有使用HOME环境变量而直接使用:`:file:`/etc/passwd``中的的设置。要想避免在同一个Cygwin环境下有两个不同的用户主目录设置，可以采用下面两种方法。

- 方法1：修改Cygwin启动的批处理文件（如：`:file:`C:\\cygwin\\Cygwin.bat``），在批处理的开头添加如下的一行，就可以清除其他软件为Windows引入的HOME环境变量。

```
set HOME=
```

- 方法2：如果希望使用HOME环境变量指向的主目录，则通过手工编辑:`:file:`/etc/passwd``文件，将其中用户主目录修改成HOME环境变量所指向的目录。

## 命令行补齐忽略文件大小写

Windows的文件系统忽略文件名大小写，在Cygwin下最好对命令行补齐进行相关设置以忽略大小写，这样使用起来更方便。

编辑文件：`:file:`~/.inputrc``，在其中添加设置`set completion-ignore-case on`，或者取消已有相关设置前面的井号注释符。修改完毕后，再重新进入Cygwin，就可以实现文件名补齐对大小写的忽略。

## 忽略文件权限的可执行位

Linux、Unix、Mac OS X下的可执行文件在文件权限有特殊的设置（设置文件的可执行位），Git可以跟踪文件的可执行位，即在添加文件时会把文件的权限也记录其中。在Windows上，缺乏对文件可执行位的支持和需要，虽然Cygwin可以模拟Linux下的文件授权并对文件的可执行位进行支持，但一来为支持文件权限而调用Cygwin的`stat()`和`lstat()`函数会比Windows自身的Win32API要慢两倍，二来对于非跨平台项目也没有必要对文件权限位进行跟踪，还有其他Windows下的工具及操作可能会破坏文件的可执行位，导致Cygwin下的Git认为文件的权限更改需要重新提交。通过下面的配置，可以禁止Git对文件权限的跟踪：

```
$ git config --system core.fileMode false
```

在此模式下，当已添加到版本库中的文件其权限的可执行位改变时，该文件不会显示有改动。新增到版本库的文件，都以`100644`的权限添加（忽略可执行位），无论文件本身是否设置为可执行。

关于Cygwin的更多定制和帮助，参见网址：<http://www.cygwin.com/cygwin-ug-net/>。

## Cygwin下Git的中文支持

Cygwin当前版本1.7.x，对中文的支持非常好。无需任何配置就可以在Cygwin的窗口内输入中文，以及执行：`:program:`ls``命令显示中文文件名。这与我记忆中的6、7年前的Cygwin 1.5.x完全不一样了。老版本的Cygwin还需要做一些工作才能在控制台输入中文和显示中文，但是最新的Cygwin已经完全不需要了。反倒是后面要介绍的msysGit的shell环境仍然需要做出类似（老版本Cygwin）的改动才能够正常显示和输入中文。

Cygwin默认使用UTF-8字符集，并巧妙的和Windows系统的字符集之间进行转换。在Cygwin下执行：`:program:`locale``命令查看Cygwin下正在使用的字符集。

```
$ locale
LANG=C.UTF-8
LC_CTYPE="C.UTF-8"
LC_NUMERIC="C.UTF-8"
```

```
LC_TIME="C.UTF-8"
LC_COLLATE="C.UTF-8"
LC_MONETARY="C.UTF-8"
LC_MESSAGES="C.UTF-8"
LC_ALL=
```

正因如此，Cygwin下的Git对中文支持非常出色，虽然中文Windows本身使用GBK字符集，但是在Cygwin下Git的行为就如同工作在UTF-8字符集的Linux下，对中文的支持非常的好。

- 在提交时，可以在提交说明中输入中文。
- 显示提交历史，能够正常显示提交说明中的中文字符。
- 可以添加中文文件名的文件，并可以在使用utf-8字符集的Linux环境中克隆及检出。
- 可以创建带有中文字符的里程碑名称。

但是和Linux平台一样，在默认设置下，带有中文文件名的文件，在工作区状态输出、查看历史更改概要、以及在补丁文件中，文件名不能正确显示为中文，而是用若干8进制编码来显示中文，如下：

```
$ git status -s
?? "\350\257\264\346\230\216.txt"
```

通过设置变量core.quotepath为false，就可以解决中文文件名在这些Git命令输出中的显示问题。

```
$ git config --global core.quotepath false
$ git status -s
?? 说明.txt
```

## Cygwin下Git访问SSH服务

在本书第5篇第29章介绍的公钥认证方式访问Git服务，是Git写操作最重要的服务。公钥认证方式访问SSH协议的Git服务器时无需输入口令，而且更为安全。使用公钥认证就涉及到创建公钥/私钥对，以及在SSH连接时选择哪一个私钥的问题（如果建立有多个私钥）。

Cygwin下的openssh软件包提供的ssh命令和Linux下的没有什么区别，也提供ssh-keygen命令管理SSH公钥/私钥对。但是Cygwin当前的openssh（版本号：5.7p1-1）有一个Bug，偶尔在用Git克隆使用SSH协议的版本库时会中断，无法完成版本库克隆。如下：

```
$ git clone git@bj.ossxp.com:ossxp/gitbook.git
Cloning into gitbook...
remote: Counting objects: 3486, done.
remote: Compressing objects: 100% (1759/1759), done.
```

```
fatal: The remote end hung up unexpectedly MiB | 3.03 MiB/s
fatal: early EOFs: 75% (2615/3486), 13.97 MiB | 3.03 MiB/s
fatal: index-pack failed
```

如果读者也遇到同样的问题，建议使用PuTTY提供的`:command:`plink.exe``做为SSH客户端，替代存在问题的Cygwin自带的ssh命令。

## 安装PuTTY

PuTTY是Windows下一个开源软件，提供SSH客户端服务，还包括公钥管理相关工具。访问PuTTY的主页 (<http://www.chiark.greenend.org.uk/~sgtatham/putty/>)，下载并安装PuTTY。安装完毕会发现PuTTY软件包包含了好几个可执行程序，对于和Git整合，下面几个命令会用到。

- Plink: 即`:file:`plink.exe``，是命令行的SSH客户端，用于替代ssh命令。默认安装于`:file:`C:\Program Files\PuTTY\plink.exe``。
- PuTTYgen: 用于管理PuTTY格式的私钥，也可以用于将openssh格式的私钥转换为PuTTY格式的私钥。
- Pageant: 是SSH认证代理，运行于后台，负责为SSH连接提供私钥访问服务。

## PuTTY格式的私钥

PuTTY使用自定义格式的私钥文件（扩展名为`:file:`.ppk``），而不能直接使用openssh格式的私钥。即用openssh的ssh-keygen命令创建的私钥不能直接被PuTTY拿过来使用，必需经过转换。程序PuTTYgen可以实现私钥格式的转换。

运行PuTTYgen程序，如图3-15所示。



图3-15：运行PuTTYgen程序

PuTTYgen既可以重新创建私钥文件，也可以通过点击加载按钮（load）读取openssh格式的私钥文件，从而可以将其转换为PuTTY格式私钥。点击加载按钮，会弹出文件选择对话框，选择openssh格式的私钥文件（如文件：`file: `id_rsa``），如果转换成功，会显示如图3-16的界面。

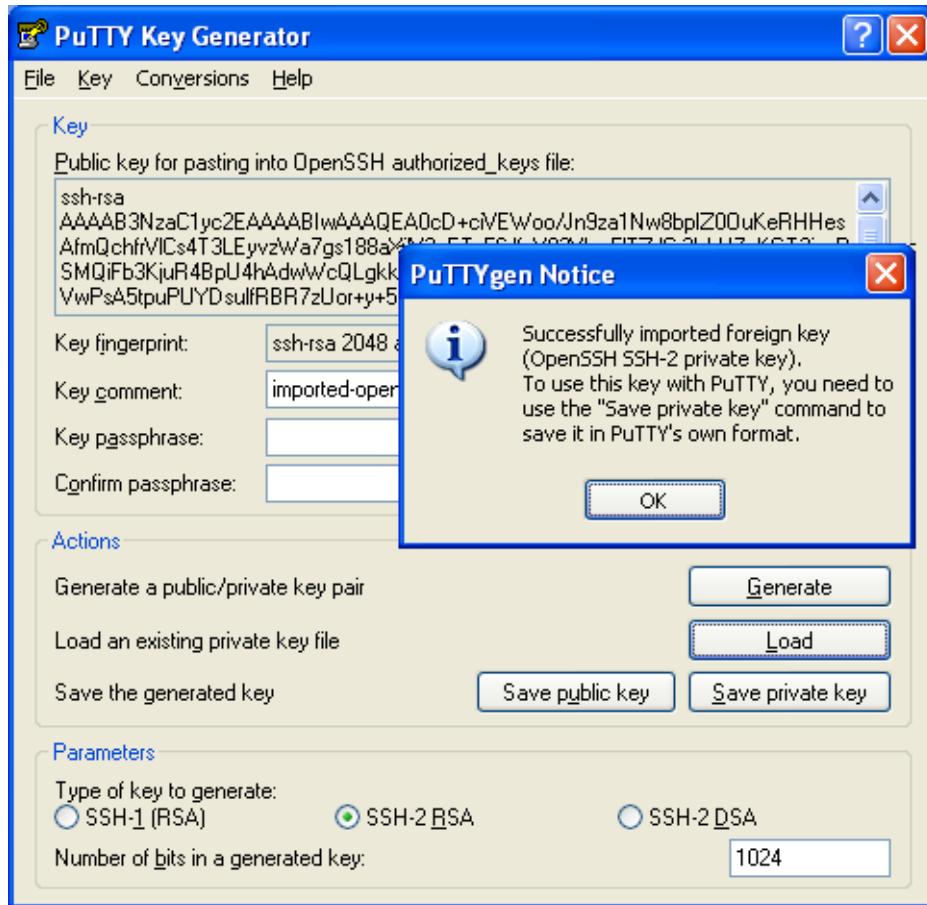


图3-16：PuTTYgen完成私钥加载

然后点击“Save private key”（保存私钥），就可以将私钥保存为PuTTY的`:file:`.ppk``格式的私钥。例如将私钥保存到文件`:file:`~/.ssh/jiangxin-cygwin.ppk``中。

## Git使用Pageant进行公钥认证

Git在使用命令行工具Plink (`:program:`plink.exe``) 做为SSH客户端访问SSH协议的版本库服务器时，如何选择公钥呢？使用Pageant是一个非常好的选择。Pageant是PuTTY软件包中为各个PuTTY应用提供私钥请求的代理软件，当Plink连接SSH服务器需要请求公钥认证时，Pageant就会提供给Plink相应的私钥。

运行Pageant，启动后显示为托盘区中的一个图标，在后台运行。当使用鼠标右键单击Pageant的图标，就会显示弹出菜单如图3-17所示。

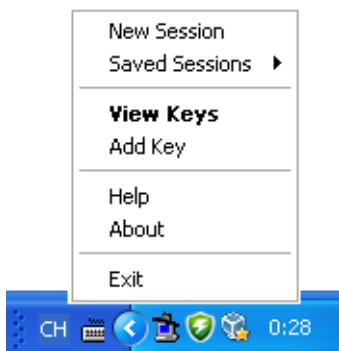


图3-17：Pageant的弹出菜单

点击弹出菜单中的“Add Key”（添加私钥）按钮，弹出文件选择框，选择扩展名为`:file:`.ppk``的PuTTY格式的公钥，即完成了Pageant的私钥准备工作。

接下来，还需要对Git进行设置，设置Git使用`:file:`plink.exe``做为SSH客户端，而不是缺省的`:program:`ssh``命令。通过设置GIT\_SSH环境变量即可实现。

```
$ export GIT_SSH=/cygdrive/c/Program\ Files/PuTTY/plink.exe
```

上面在设置GIT\_SSH环境变量的过程中，使用了Cygwin格式的路径，而非Windows格式，这是因为Git是在Cygwin的环境中调用`:program:`plink.exe``命令的，当然要使用Cygwin能够理解的路径。

然后就可以用Git访问SSH协议的Git服务器了。运行在后台的Pageant会在需要的时候为`:command:`plink.exe``提供私钥访问服务。但在首次连接一个使用SSH协议的Git服务器的时候，很可能会因为远程SSH服务器的公钥没有经过确认导致git命令执行失败。如下所示。

```
$ git clone git@bj.osssxp.com:osssxp/gitbook.git
Cloning into gitbook...
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
Connection abandoned.
fatal: The remote end hung up unexpectedly
```

这是因为首次连接一个SSH服务器时，要对其公钥进行确认（以防止被钓鱼），而运行于Git下的`:program:`plink.exe``没有机会从用户那里获取输入以建立对该SSH服务器公钥的信任，因此Git访问失败。解决办法非常简单，就是直接运行`:program:`plink.exe``连接一次远程SSH服务器，对公钥确认进行应答。如下：

```
$ /cygdrive/c/Program\ Files/PuTTY/plink.exe git@bj.ossxp.com
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.

The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
If you trust this host, enter "y" to add the key to
PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without
adding the key to the cache, enter "n".
If you do not trust this host, press Return to abandon the
connection.

Store key in cache? (y/n)
```

输入“y”，将公钥保存在信任链中，以后再次连接就不会弹出该确认应答了。当然执行Git命令，也就可以成功执行了。

## 使用自定义SSH脚本取代Pageant

使用Pageant还要在每次启动Pageant时手动选择私钥文件，比较的麻烦。实际上可以创建一个脚本对:program:`plink.exe`进行封装，在封装的脚本中指定私钥文件，这样就不必使用Pageant而实现公钥认证了。

例如：创建脚本:file:`~/bin/ssh-jiangxin`，文件内容如下了：

```
#!/bin/sh

/cygdrive/c/Program\ Files/PuTTY/plink.exe -i \
c:/cygwin/home/jiangxin/.ssh/jiangxin-cygwin.ppk $*
```

设置该脚本为可执行。

```
$ chmod a+x ~/bin/ssh-jiangxin
```

通过该脚本和远程SSH服务器连接，使用下面的命令：

```
$ ~/bin/ssh-jiangxin git@bj.ossxp.com
Using username "git".
Server refused to allocate pty
hello jiangxin, the gitolite version here is v1.5.5-9-g4c11bd8
the gitolite config gives you the following access:
    R          gistore-bj.ossxp.com/.*$
    R          gistore-ossxp.com/.*$
    C  R  W      ossxp/.*$
```

```
R W      test/repo1  
R W      test/repo2  
R W      test/repo3  
@R @W    test/repo4  
@C @R W   users/jiangxin/.+$
```

设置GIT\_SSH变量，使之指向新建立的脚本，然后就可以脱离Pageant来连接SSH协议的Git库了。

```
$ export GIT_SSH=~/bin/ssh-jiangxin
```

来源：<https://github.com/gotgit/gotgit/blob/master/01-meet-git/050-install-on-windows-cygwin.rst>

# Windows下安装和使用Git (msysGit篇)

运行在Cygwin下的Git不是直接使用Windows的系统调用，而是通过二传手:`file: `cygwin1.dll``来进行，虽然Cygwin的git命令能够在Windows下的`program: `cmd.exe``命令窗口中运行的非常好，但Cygwin下的Git并不能看作是Windows下的原生程序。相比Cygwin下的Git，msysGit是原生的Windows程序，msysGit下运行的Git直接通过Windows的系统调用运行。

msysGit的名字前面的四个字母来源于MSYS项目。MSYS项目源自于MinGW (Minimalist GNU for Windows, 最简GNU工具集)，通过增加了一个bash提供的shell环境以及其他相关工具软件，组成了一个最简系统 (Minimal SYStem)，简称MSYS。利用MinGW提供的工具，以及Git针对MinGW的一个分支版本，在Windows平台为Git编译出一个原生应用，结合MSYS就组成了msysGit。

## 安装msysGit

安装msysGit非常简单，访问msysGit的项目主页

(<http://code.google.com/p/msysgit/>)，下载msysGit。最简单的方式是下载名为`file: `Git-<VERSION>-preview<DATE>.exe``的软件包，如：`file: `Git-1.7.3.1-preview20101002.exe``。如果有时间和耐心，喜欢观察Git是如何在Windows上是编译为原生应用的，也可以下载带`file: `msysGit-fullinstall-``前缀的软件包。

点击下载的安装程序（如`file: `Git-1.7.3.1-preview20101002.exe``），开始安装，如图3-18。

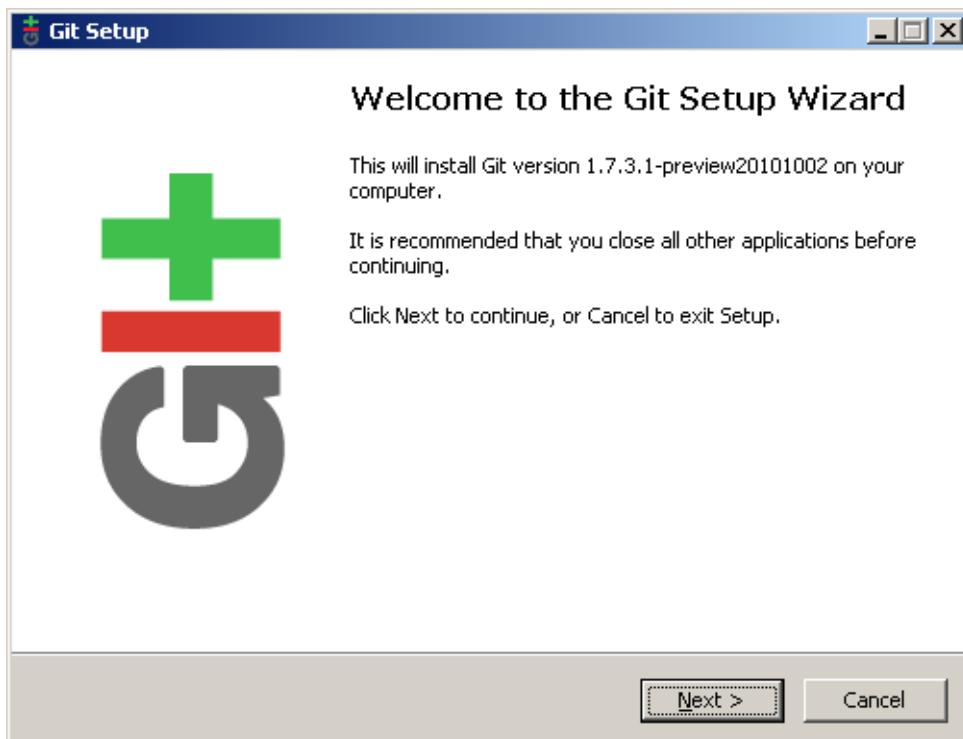


图3-18：启动msysGit安装

默认安装到:`file:`C:\\Program Files\\Git``目录中。

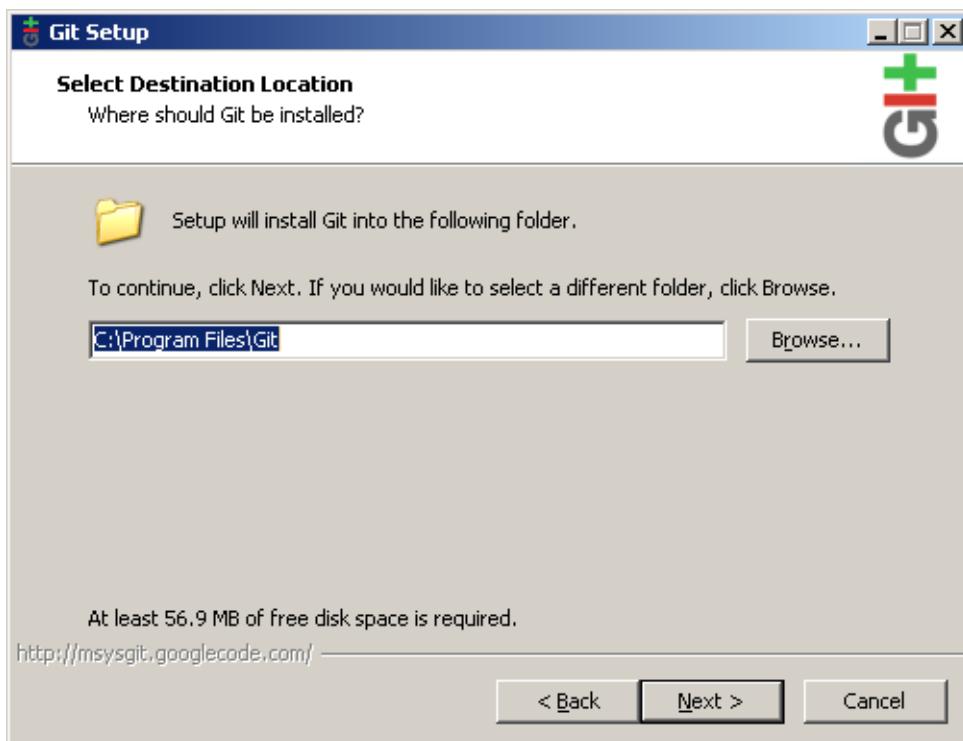


图3-19：选择msysGit的安装目录

在安装过程中会询问是否修改环境变量，如图3-20。默认选择“Use Git Bash Only”，即只在msysGit提供的shell环境（类似Cygwin）中使用Git，不修改环境变量。注意如果选择最后一项，会将msysGit所有的可执行程序全部加入Windows的PATH路径中，有的命令会覆盖Windows相同文件名的程序（如：`command: `find.exe`` 和 `command: `sort.exe``）。而且如

果选择最后一项，还会为Windows添加HOME环境变量，如果安装有Cygwin，Cygwin会受到msysGit引入的HOME环境变量的影响（参见前面3.3.3节的相关讨论）。



图3-20：是否修改系统的环境变量

还会询问换行符的转换方式，使用默认设置就好。参见图3-21。关于换行符转换，参见本书第8篇相关章节。

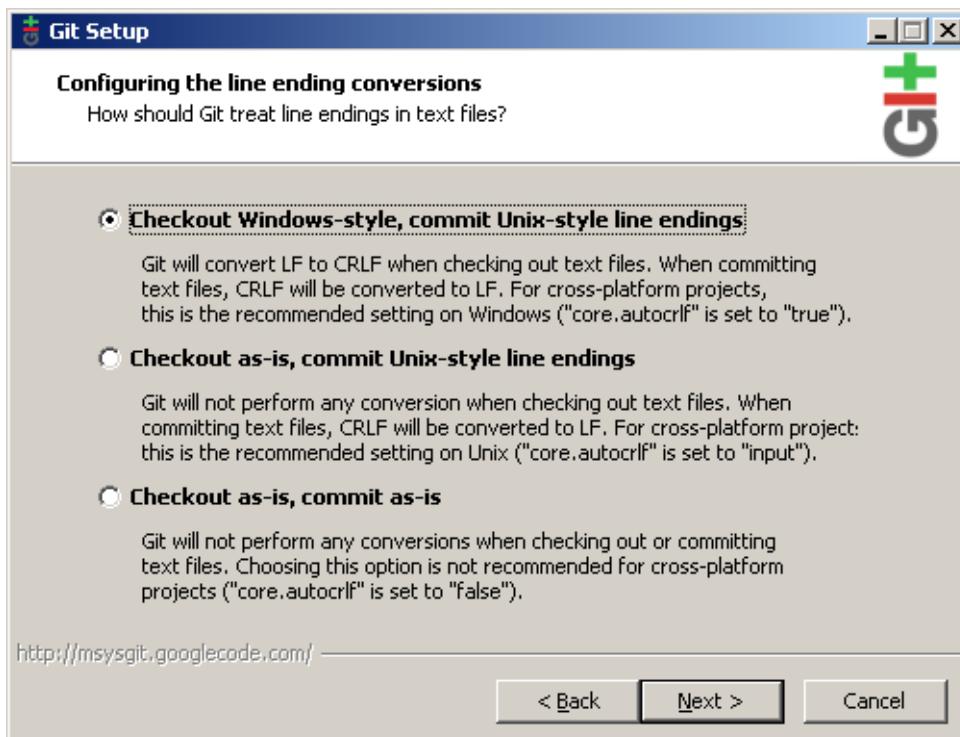


图3-21：换行符转换方式

根据提示，完成msysGit的安装。

## msysGit的配置和使用

完成msysGit的安装后，点击Git Bash图标，启动msysGit，如图3-22。会发现Git Bash 的界面和Cygwin的非常相像。

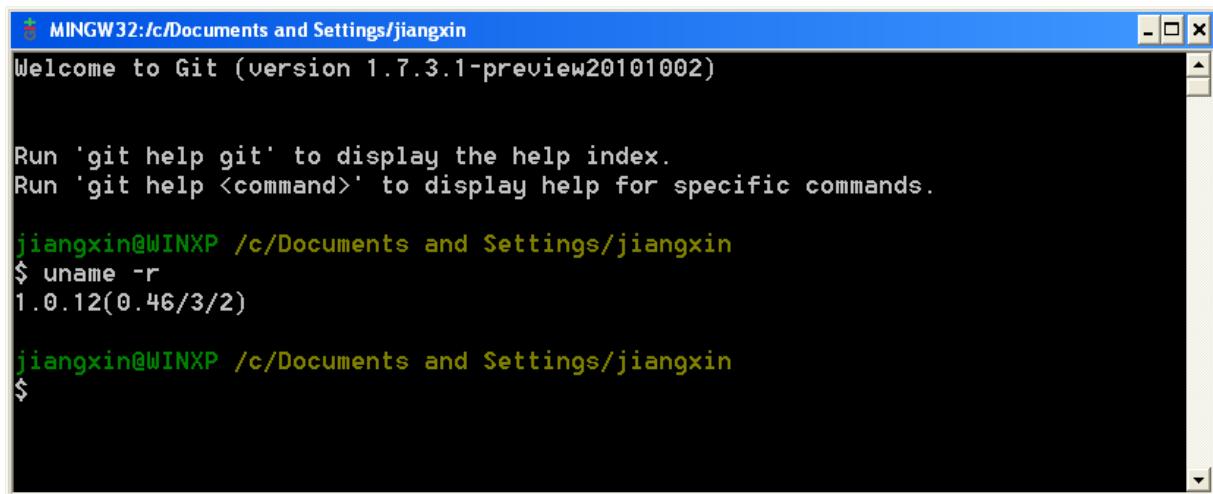


图3-22：启动Git Bash

## 如何访问Windows的磁符

在msysGit下访问Windows的各个盘符，要比Cygwin简单，直接通过:`:file:`/c``即可访问Windows的C:盘，用:`:file:`/d``访问Windows的D:盘。

```
$ ls -ld /c/Windows
drwxr-xr-x 233 jiangxin Administ          0 Jan 31 00:44 /c/Windows
```

至于msysGit的根目录，实际上就是msysGit安装的目录，如：“C:\Program Files\Git”。

## 命令行补齐和忽略文件大小写

msysGit缺省已经安装了Git的命令补齐功能，并且在对文件名命令补齐时忽略大小写。这是因为msysGit已经在配置文件:`:file:`/etc/inputrc``中包含了下列的设置：

```
set completion-ignore-case on
```

## msysGit的shell环境的中文支持

在介绍Cygwin的章节中曾经提到过，msysGit的shell环境的中文支持相当于老版本的Cygwin，需要配置才能够实现录入中文和显示中文。

## 中文录入问题

缺省安装的msysGit的shell环境无法输入中文。为了能在shell界面中输入中文，需要修改配置文件:`file: `/etc/inputrc``，增加或修改相关配置如下：

```
# disable/enable 8bit input
set meta-flag on
set input-meta on
set output-meta on
set convert-meta off
```

关闭Git Bash再重启，就可以在msysGit的shell环境中输入中文了。

```
$ echo 您好
您好
```

## 分页器中文输出问题

当对`file: `/etc/inputrc``进行正确的配置之后，能够在shell下输入中文，但是执行下面的命令会显示乱码。这显然是`program: `less``分页器命令导致的问题。

```
$ echo 您好 | less
<C4><FA><BA><C3>
```

通过管道符调用分页器命令`program: `less``后，原本的中文输出变成了乱码显示。这将会导致Git很多命令的输出都会出现中文乱码问题，因为Git大量的使用`program: `less``命令做为分页器。之所以`program: `less``命令出现乱码，是因为该命令没有把中文当作正常的字符，可以通过设置LESSCHARSET环境变量，将utf-8编码字符视为正规字符显示，则中文就能正常显示了。下面的操作，可以在`program: `less``分页器中正常显示中文。

```
$ export LESSCHARSET=utf-8
$ echo 您好 | less
您好
```

编辑配置文件`file: `/etc/profile``，将对环境变量LESSCHARSET的设置加入其中，以便msysGit的shell环境一启动即加载。

```
declare -x LESSCHARSET=utf-8
```

## ls命令对中文文件名的显示

最常用的显示目录和文件名列表的命令:`program: `ls``对中文文件名的显示有问题。下面的命令创建了一个中文文件名的文件，显示文件内容中的中文没有问题，但是显示文件名本身会显示为一串问号。

```
$ echo 您好 > 您好.txt  
  
$ cat *.txt  
您好  
  
$ ls *.txt  
????.txt
```

实际上只要在:`program: `ls``命令后添加参数:`command: `--show-control-chars``即可正确显示中文。

```
$ ls --show-control-chars *.txt  
您好.txt
```

为方便起见，可以为:`program: `ls``命令设置一个别名，这样就不必在输入:`program: `ls``命令时输入长长的参数了。

```
$ alias ls="ls --show-control-chars"  
  
$ ls *.txt  
您好.txt
```

将上面的:`command: `alias``命令添加到配置文件:`file: `/etc/profile``中，实现在每次运行Git Bash时自动加载。

## msysGit中Git的中文支持

非常遗憾的是msysGit中的Git对中文支持没有Cygwin中的Git做的那么好，msysGit中的Git对中文支持的程度，就相当于前面讨论过的Linux使用了GBK字符集时Git的情况。

- 未经配置的msysGit提交时，如果在提交说明中输入中文，从Linux平台或其他UTF-8字符集平台上查看提交说明显示乱码。
- 同样从Linux平台或者其他使用UTF-8字符集平台进行的提交，若提交说明包含中文，

在未经配置的msysGit中也显示乱码。

- 如果使用msysGit向版本库中添加带有中文文件名的文件，在Linux（或其他utf-8）平台检出文件名显示为乱码。反之亦然。
- 不能创建带有中文字符的引用（里程碑、分支等）。

如果希望版本库中出现使用中文文件名的文件，最好不要使用msysGit，而是使用Cygwin下的Git。而如果只是想在提交说明中使用中文，经过一定的设置msysGit还是可以实现的。

为了解决提交说明显示乱码问题，msysGit要为Git设置参数*i18n.logOutputEncoding*，将提交说明的输出编码设置为gbk。

```
$ git config --system i18n.logOutputEncoding gbk
```

Git在提交时并不会对提交说明进行从GBK字符集到UTF-8的转换，但是可以在提交说明中标注所使用的字符集，因此在非UTF-8字符集的平台录入中文，需要用下面指令设置录入提交说明的字符集，以便在commit对象中嵌入正确的编码说明。为了使msysGit提交时输入的中文说明能够在Linux或其他使用UTF-8编码的平台中正确显示，还必须对参数*i18n.commitEncoding*设置。

```
$ git config --system i18n.commitEncoding gbk
```

同样，为了能够让带有中文文件名的文件，在工作区状态输出、查看历史更改概要、以及在补丁文件中，能够正常显示，要为Git配置*core.quotePath*变量，将其设置为false。但是要注意在msysGit中添加中文文件名的文件，只能在msysGit环境中正确显示，而在其他环境（Linu、Mac OS X、Cygwin）中文件名会出现乱码。

```
$ git config --system core.quotePath false  
$ git status -s  
?? 说明.txt
```

注意：如果同时安装了Cygwin和msysGit（可能配置了相同的用户主目录），或者因为中文支持问题而需要单独为TortoiseGit准备一套msysGit时，为了保证不同的msysGit之间，以及和Cygwin之间的配置不会互相影响，而在配置Git环境时使用：[command: `--system`](#)参数。这是因为不同的msysGit安装以及Cygwin有着不同的系统配置文件，但是用户级配置文件位置却可能重合。

## 使用SSH协议

msysGit软件包包含的ssh命令和Linux下的没有什么区别，也提供ssh-keygen命令管理SSH公钥/私钥对。在使用msysGit的ssh命令时，没有遇到Cygwin中的ssh命令（版本号：

5. 7p1-1) 不稳定的问题，即msysGit下的ssh命令可以非常稳定的工作。

如果需要和Windows有更好的整合，希望使用图形化工具管理公钥，也可以使用PuTTY提供的plink.exe做为SSH客户端。关于如何使用PuTTY可以参见3. 3. 5节Cygwin和PuTTY整合的相关内容。

## TortoiseGit的安装和使用

TortoiseGit提供了Git和Windows资源管理器的整合，提供了Git的图形化操作界面。像其他Tortoise系列产品（TortoiseCVS、TortoiseSVN）一样，Git工作区的目录和文件的图标附加了标识版本控制状态的图像，可以非常直观的看到哪些文件被更改了需要提交。通过对右键菜单的扩展，可以非常方便的在资源管理器中操作Git版本库。

TortoiseGit是对msysGit命令行的封装，因此需要先安装msysGit。安装TortoiseGit非常简单，访问网站<http://code.google.com/p/tortoisegit/>，下载安装包，然后根据提示完成安装。

安装过程中会询问要使用的SSH客户端，如图3-23。缺省使用内置的TortoisePLink（来自PuTTY项目）做为SSH客户端。

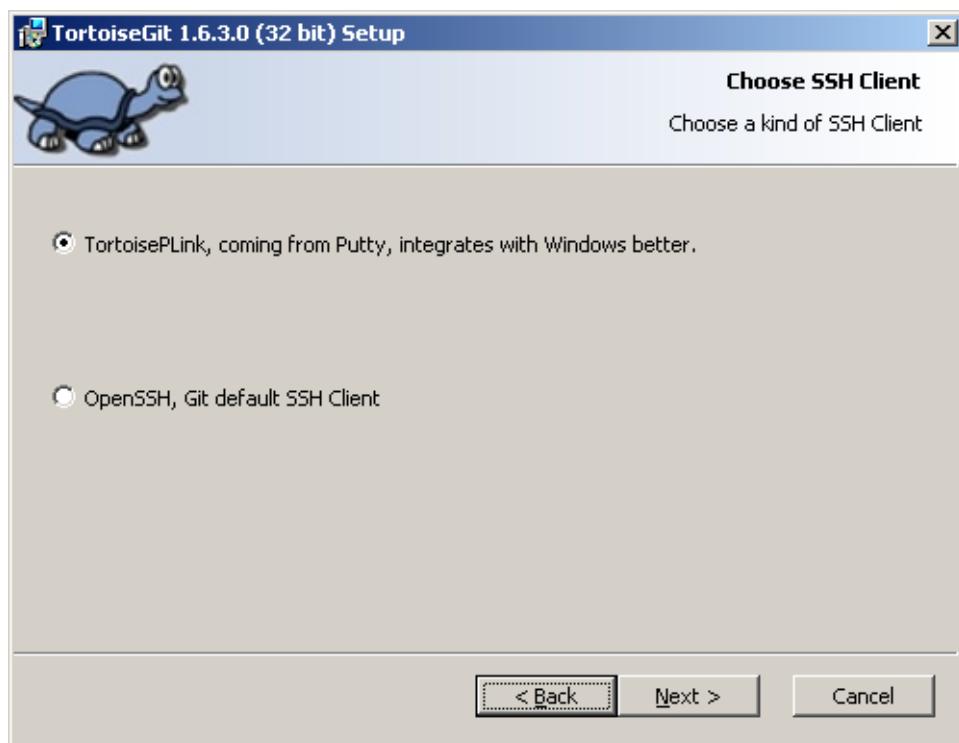


图3-23：启动Git Bash

TortoisePLink和TortoiseGit的整合性更好，可以直接通过对话框设置SSH私钥（PuTTY格式），而无需再到字符界面去配置SSH私钥和其他配置文件。如果安装过程中选择了OpenSSH，可以在安装完毕之后，通过TortoiseGit的设置对话框重新选择TortoisePLink做为缺省SSH客户端程序，如图3-24。

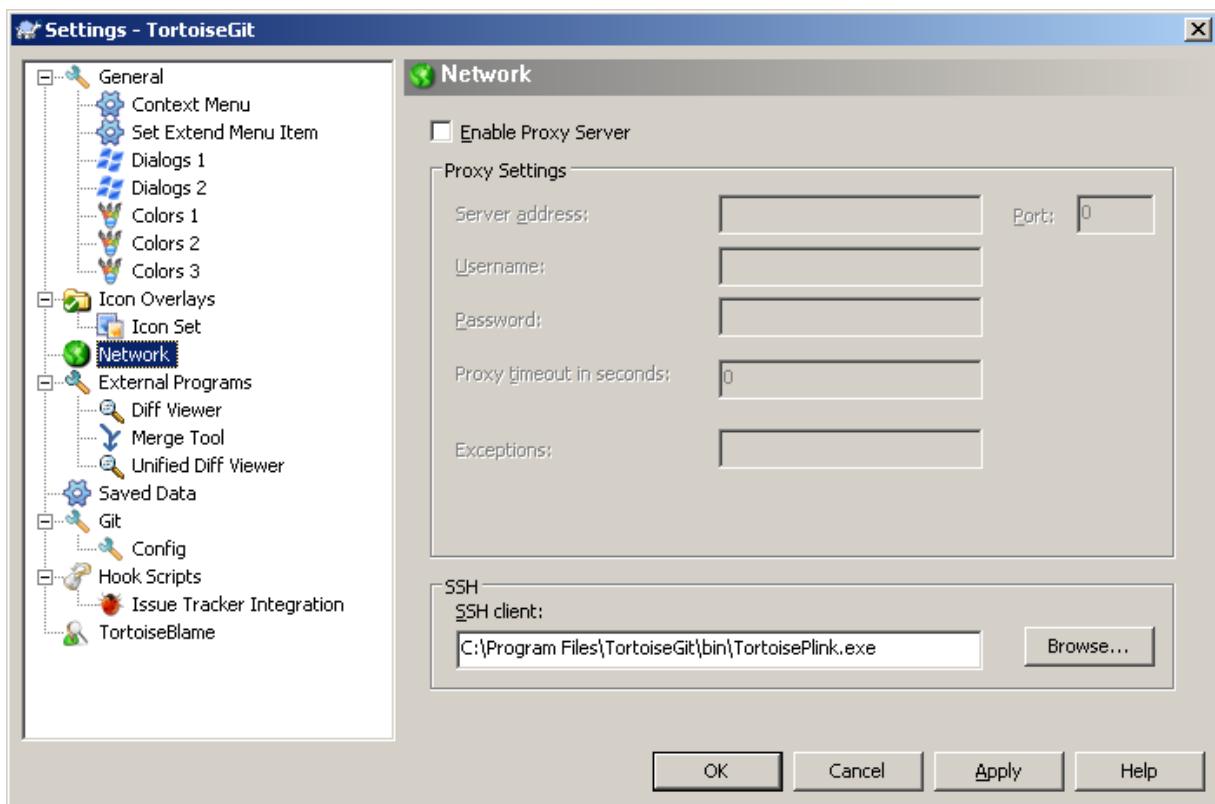


图3-24：配置缺省SSH客户端

当配置使用TortoisePLink做为缺省SSH客户端时，在执行克隆操作时，在操作界面中可以选择一个PuTTY格式的私钥文件进行认证，如图3-25。

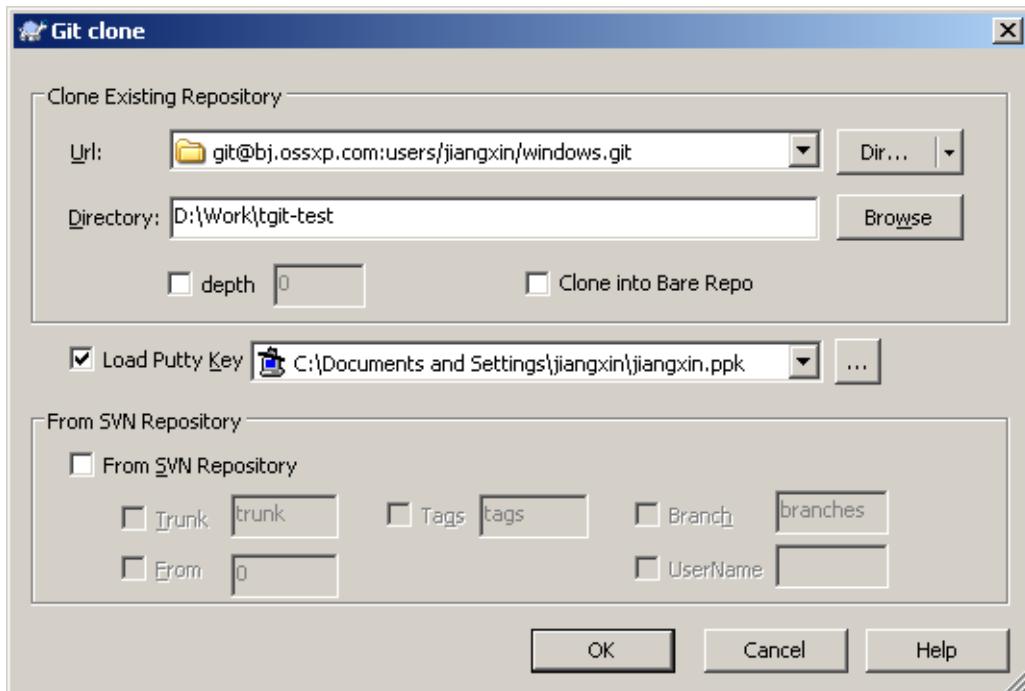


图3-25：克隆操作选择PuTTY格式私钥文件

如果连接一个服务器的SSH私钥需要更换，可以通过Git远程服务器配置界面对私钥文件进行重新设置。如图3-26。

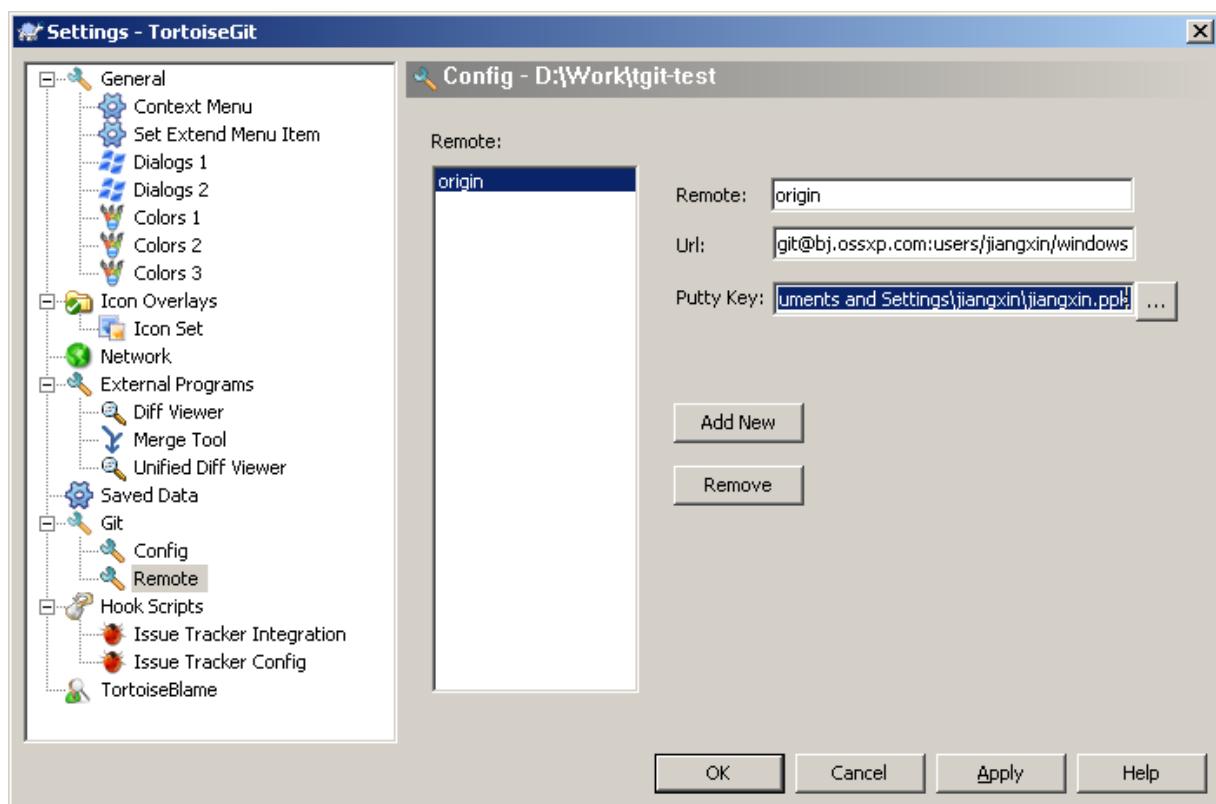


图3-26：更换连接远程SSH服务器的私钥

如果安装有多个msysGit拷贝，也可以通过TortoiseGit的配置界面进行选择，如图3-27。

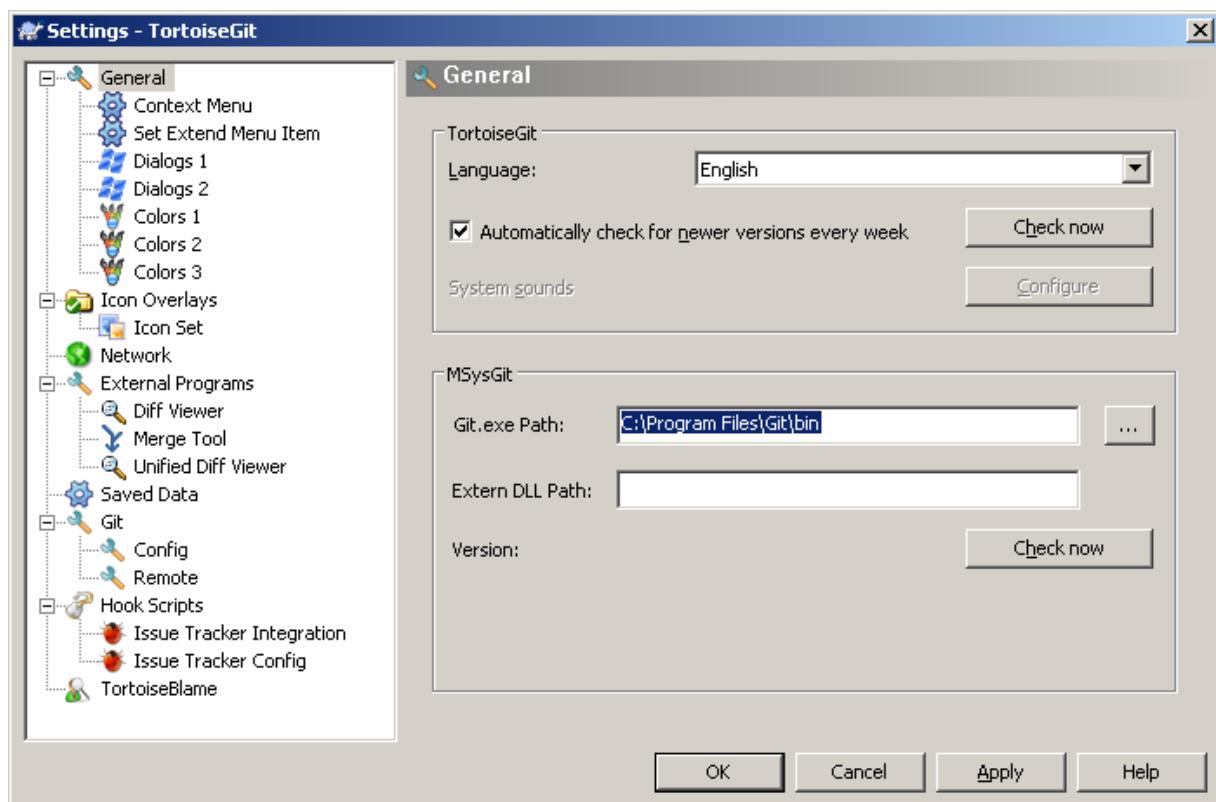


图3-27：配置msysGit的可执行程序位置

## TortoiseGit的中文支持

TortoiseGit虽然在底层调用了msysGit，但是TortoiseGit的中文支持和msysGit有区别，甚至前面介绍msysGit中文支持时所进行的配制会破坏TortoiseGit。

TortoiseGit在提交时，会将提交说明转换为UTF-8字符集，因此无须对`:command:`i18n.commitEncoding``变量进行设置。相反，如果设置了`:command:`i18n.commitEncoding``为gbk或其他，则在提交对象中会包含错误的编码设置，有可能为提交说明的显示带来麻烦。

TortoiseGit在显示提交说明时，认为所有的提交说明都是UTF-8编码，会转换为合适的Windows本地字符集显示，而无须设置`i18n.logOutputEncoding`变量。因为当前版本的TortoiseGit没有对提交对象中的encoding设置进行检查，因此使用GBK字符集的提交说明中的中文不能正常显示。

因此，如果需要同时使用msysGit的文字界面Git Bash以及TortoiseGit，并需要在提交说明中使用中文，可以安装两套msysGit，并确保TortoiseGit关联的msysGit没有对`i18n.commitEncoding`进行设置。

TortoiseGit对使用中文命名的文件和目录的支持和msysGit一样，都存在缺陷，因此应当避免在msysGit和TortoiseGit中添加用中文命名的文件和目录，如果确实需要，可以使用Cygwin。

来源：<https://github.com/gotgit/gotgit/blob/master/01-meet-git/060-install-on-windows-msysgit.rst>

# Git初始化

## 创建版本库及第一次提交

您当前使用的是1.5.6或更高版本的Git么？

```
$ git --version  
git version 1.7.11.2
```

Git是一个活跃的项目，仍在不断的进化之中，不同Git版本的功能不尽相同。本书对Git的介绍涵盖了1.5.6到1.7.11版本，这也是目前Git的主要版本。如果您使用的Git版本低于1.5.6，那么请升级到1.5.6或更高的版本。本书示例使用的是1.7.11.2版本的Git，我们会尽可能地指出那些低版本不兼容的命令及参数。

在开始Git之旅之前，我们需要设置一下Git的环境变量，这个设置是一次性的工作。即这些设置会在全局文件（用户主目录下的`:file:`.gitconfig``）或系统文件（`:file:`/etc/gitconfig``）中做永久的记录。

- 告诉Git当前用户的姓名和邮件地址，配置的用户名和邮件地址将在版本库提交时作为提交者的用户名和邮件地址。

注意下面的两条命令不要照抄照搬，而是用您自己的用户名和邮件地址代替这里的用户名和邮件地址，否则您的劳动成果（提交内容）可要算到作者的头上了。

```
$ git config --global user.name "Jiang Xin"  
$ git config --global user.email jiangxin@osxp.com
```

- 设置一些Git别名，以便可以使用更为简洁的子命令。

例如：输入`:command:`git ci``即相当于`:command:`git commit -s``[1]，输入`:command:`git st``即相当于`:command:`git -p status``[2]。

- 如果拥有系统管理员权限（可以执行`:command:`sudo``命令），希望注册的命令别名能够被所有用户使用，可以执行如下命令：

```
$ sudo git config --system alias.br branch  
$ sudo git config --system alias.ci "commit -s"  
$ sudo git config --system alias.co checkout  
$ sudo git config --system alias.st "-p status"
```

- 也可以运行下面的命令，只在本用户的全局配置中添加Git命令别名：

```
$ git config --global alias.st status  
$ git config --global alias.ci "commit -s"  
$ git config --global alias.co checkout  
$ git config --global alias.br branch
```

Git的所有操作，包括创建版本库等管理操作都用`:command:`git``一个命令即可完成，不像其他有的版本控制系统（如Subversion），一些涉及管理的操作要使用另外的命令（如`:command:`svnadmin``）。创建Git版本库，可以直接进入到包含数据（文件和子目录）的目录下，通过执行`:command:`git init``完成版本库的初始化。

下面就从一个空目录开始初始化版本库，这个版本库命名为“demo”，这个DEMO版本库将贯穿本篇始终。为了方便说明，使用了名为`:file:`/path/to/my/workspace``的目录作为个人的工作区根目录，您可以在磁盘中创建该目录并设置正确的权限。

首先建立一个新的工作目录，进入该目录后，执行:command:`git init`创建版本库。

```
$ cd /path/to/my/workspace  
$ mkdir demo  
$ cd demo  
$ git init  
初始化空的 Git 版本库于 /path/to/my/workspace/demo/.git/
```

实际上，如果Git的版本是1.6.5或更新的版本，可以在:command:`git init`命令的后面直接输入目录名称，自动完成目录的创建。

```
$ cd /path/to/my/workspace  
$ git init demo  
初始化空的 Git 版本库于 /path/to/my/workspace/demo/.git/  
$ cd demo
```

从上面版本库初始化后的输出中，可以看到执行:command:`git init`命令在工作区创建了隐藏目录:file:`.git`。

```
$ ls -aF  
./ ../.git/
```

这个隐藏的:file:`.git`目录就是Git版本库（又叫仓库，repository）。

:file:`.git`版本库目录所在的目录，即:file:`/path/to/my/workspace/demo`目录称工作区，目前工作区除了包含一个隐藏的file:.git版本库目录外空无一物。

下面为工作区中加点料：在工作区中创建一个文件:file:`welcome.txt`，内容就是一行“Hello.”。

```
$ echo "Hello." > welcome.txt
```

为了将这个新建立的文件添加到版本库，需要执行下面的命令：

```
$ git add welcome.txt
```

切记，到这里还没有完。Git和大部分其他版本控制系统都需要再执行一次提交操作，对于Git来说就是执行:command:`git commit`命令完成提交。在提交过程中需要输入提交说明，这个要求对于Git来说是强制性的，不像其他很多版本控制系统（如CVS、Subversion）允许空白的提交说明。在Git提交时，如果在命令行不提供提交说明（没有使用-m参数），Git会自动打开一个编辑器，要求您在其中输入提交说明，输入完毕保存退出。需要说明的是，读者要在一定程度上掌握vim或emacs这两种Linux下常用编辑器的编辑技巧，否则保存退出也会成为问题。

下面进行提交。为了说明方便，使用-m参数直接给出了提交说明。

```
$ git ci -m "initialized"  
[master (根提交) 7e749cc] initialized  
1 个文件被修改, 插入 1 行(+)  
create mode 100644 welcome.txt
```

从上面的命令及输出可以看出：

- 使用了Git命令别名，即:command:`git ci`相当于执行:command:`git commit`。在本节的一开始就进行了Git命令别名的设置。

- 通过-m参数设置提交说明为：“initialized”。该提交说明也显示在命令输出的第一行中。
- 命令输出的第一行还显示了当前处于名为master的分支上，提交ID为7e749cc[3]，且该提交是该分支的第一个提交，即根提交（root-commit）。根提交和其他提交的区别在于没有关联的父提交，这会在后面的章节中加以讨论。
- 命令输出的第二行开始显示本次提交所做修改的统计：修改了一个文件，包含一行的插入。

## 思考：为什么工作区下有一个:`file:.git`目录？

Git及其他分布式版本控制系统（如Mercurial/Hg、Bazaar）的一个显著特点是，版本库位于工作区的根目录下。对于Git来说，版本库位于工作区根目录下的`file:.git`目录中，且仅此一处，在工作区的子目录下则没有任何其他跟踪文件或目录。Git的这个设计要比CVS、Subversion这些传统的集中式版本控制工具来说方便多了。

看看版本控制系统前辈们是如何对工作区的跟踪进行设计的。通过其各自设计的优缺点，我们会更加深刻地体会到Git实现的必要和巧妙。

对于CVS，工作区的根目录及每一个子目录下都有一个`file:CVS`目录，`file:CVS`目录中包含几个配置文件，建立了对版本库的追踪。如`file:CVS`目录下的`Entries`文件记录了从版本库检出到工作区的文件的名称、版本和时间戳等，这样就可以通过对工作区文件时间戳的改变来判断文件是否更改。这样设计的好处是，可以将工作区移动到任何其他目录中，而工作区和版本控制服务器的映射关系保持不变，这样工作区依然能够正常工作。甚至还将工作区的某个子目录移动到其他位置，形成新的工作区，在新的工作区下仍然可以完成版本控制相关的操作。但是缺点也很多，例如工作区文件修改了，因为没有原始文件做比对，因此向服务器提交修改的时候只能对整个文件进行传输而不能仅传输文件的改动部分，导致从客户端到服务器的网络传输效率降低。还有一个风险是信息泄漏。例如Web服务器的目录下如果包含了`file:CVS`目录，黑客就可以通过扫描`file:CVS/Entries`文件得到目录下的文件列表，由此造成信息泄漏。

对于Subversion来说，工作区的根目录和每一个子目录下都有一个`file:.svn`目录。目录`file:.svn`中不但包含了类似CVS的跟踪目录下的配置文件，还包含了当前工作区下每一个文件的拷贝。多出文件的原始拷贝让某些svn命令可以脱离版本库执行，还可以在由客户端向服务器提交时，仅仅对文件改动的内容进行提交，因为改动的文件可以和原始拷贝进行差异比较。但是这么做的缺点除了像CVS因为引入`file:CVS`跟踪目录而造成的信息泄漏的风险外，还导致了加倍占用工作区的空间。再有一个不方便的地方就是，当在工作区目录下针对文件内容进行搜索的时候，会因为`file:.svn`目录下文件的原始拷贝，导致搜索的结果加倍，而出现混乱的搜索结果。

有的版本控制系统，在工作区根本就没有任何跟踪文件，例如，某款版本控制的商业软件（就不点名了），工作区就非常干净没有任何的配置文件和配置目录。但是这样的设计更加糟糕，因为它实际上是由服务器端建立的文件跟踪，在服务器端的数据库中保存了一个表格：哪台客户端，在哪个本地目录检出了哪个版本的版本库文件。这样做的后果是，如果客户端将工作区移动或改名会导致文件的跟踪状态丢失，出现文件状态未知的问题。客户端操作系统重装，也会导致文件跟踪状态丢失。

Git的这种设计，将版本库放在工作区根目录下，所有的版本控制操作（除了和其他远程版本库之间的互操作）都在本地即可完成，不像Subversion只有寥寥无几的几个命令才能脱离网络执行。而且Git也没有CVS和Subversion的安全泄漏问题（只要保护好`file:.git`目录），也没有Subversion在本地文件搜索时出现搜索结果混乱的问题，甚至Git还提供了一条`command:git grep`命令来更好地搜索工作区的文件内容。

例如作者在本书的Git库中执行下面的命令对版本库中的文件进行内容搜索：

```
$ git grep "工作区文件内容搜索"
02-git-solo/010-git-init.rst::command:`git grep` \ 命令来更好地搜索工作区的文件内
```

当工作区中包含了子目录，在子目录中执行Git命令时，如何定位版本库呢？

实际上，当在Git工作区目录下执行操作的时候，会对目录依次向上递归查找`:file:`.git``目录，找到的`:file:`.git``目录就是工作区对应的版本库，`:file:`.git``所在的目录就是工作区的根目录，文件`:file:`.git/index``记录了工作区文件的状态（实际上是暂存区的状态）。

例如在非Git工作区执行`:command:`git``命令，会因为找不到`:file:`.git``目录而报错。

```
$ cd /path/to/my/workspace/  
$ git status  
fatal: Not a git repository (or any of the parent directories): .git
```

如果跟踪一下执行`:command:`git status``命令时的磁盘访问[4]，会看到沿目录依次向上递归的过程。

```
$ strace -e 'trace=file' git status  
...  
getcwd("/path/to/my/workspace", 4096) = 14  
...  
access(".git/objects", X_OK) = -1 ENOENT (No such file or directory)  
access("./objects", X_OK) = -1 ENOENT (No such file or directory)  
...  
chdir("..") = 0  
...  
access(".git/objects", X_OK) = -1 ENOENT (No such file or directory)  
access("./objects", X_OK) = -1 ENOENT (No such file or directory)  
...  
chdir("..") = 0  
...  
access(".git/objects", X_OK) = -1 ENOENT (No such file or directory)  
access("./objects", X_OK) = -1 ENOENT (No such file or directory)  
fatal: Not a git repository (or any of the parent directories): .git
```

那么有什么办法知道Git版本库的位置，以及工作区的根目录在哪里呢？

当在工作区执行`:command:`git``命令时，上面查找版本库的操作总是默默地执行，就好像什么也没有发生的一样。如果希望显示工作区的根目录，Git有一个底层命令可以实现。

- 在工作区下建立目录`:file:`a/b/c``，进入到该目录中。

```
$ cd /path/to/my/workspace/demo/  
$ mkdir -p a/b/c  
$ cd /path/to/my/workspace/demo/a/b/c
```

- 显示版本库`:file:`.git``目录所在的位置。

```
$ git rev-parse --git-dir  
/path/to/my/workspace/demo/.git
```

- 显示工作区根目录。

```
$ git rev-parse --show-toplevel  
/path/to/my/workspace/demo
```

- 相对于工作区根目录的相对目录。

```
$ git rev-parse --show-prefix  
a/b/c/
```

- 显示从当前目录(cd)后退(up)到工作区的根的深度。

```
$ git rev-parse --show-cdup  
.../..../..
```

把版本库:file:`.git`目录放在工作区，是不是太不安全了？

从存储安全的角度上来讲，将版本库放在工作区目录下，有点“把鸡蛋装在一个篮子里”的味道。如果忘记了工作区中还有版本库，直接从工作区的根执行目录删除就会连版本库一并删除，这个风险的确是蛮高的。将版本库和工作区拆开似乎更加安全，但是不要忘了之前的讨论，将版本库和工作区拆开，就要引入其他机制以便实现版本库对工作区的追踪。

Git克隆可以降低因为版本库和工作区混杂在一起导致的版本库被破坏的风险。可以通过克隆版本库，在本机另外的磁盘/目录中建立Git克隆，并在工作区有改动提交时，手动或自动地执行向克隆版本库的推送(:file:`git push`)操作。如果使用网络协议，还可以实现在其他机器上建立克隆，这样就更安全了（双机备份）。对于使用Git做版本控制的团队，每个人都是一个备份，因此团队开发中的Git版本库更安全，管理员甚至根本无须顾虑版本库存储安全问题。

## 思考：:command:`git config`命令参数的区别？

在之前出现的:command:`git config`命令，有的使用了--global参数，有的使用了--system参数，这两个参数有什么区别么？执行下面的命令，您就明白:command:`git config`命令实际操作的文件了。

- 执行下面的命令，将打开:file:`/path/to/my/workspace/demo/.git/config`文件进行编辑。

```
$ cd /path/to/my/workspace/demo/  
$ git config -e
```

- 执行下面的命令，将打开:file:`/home/jiangxin/.gitconfig`（用户主目录下的:file:`.gitconfig`文件）全局配置文件进行编辑。

```
$ git config -e --global
```

- 执行下面的命令，将打开:file:`/etc/gitconfig`系统级配置文件进行编辑。

如果Git安装在:file:`/usr/local/bin`下，这个系统级的配置文件也可能是  
在:file:`/usr/local/etc/gitconfig`。

```
$ git config -e --system
```

Git的三个配置文件分别是版本库级别的配置文件、全局配置文件（用户主目录下）和系统级配置文件（:file:`/etc`目录下）。其中版本库级别配置文件的优先级最高，全局配置文件其次，系统级配置文件优先级最低。这样的优先级设置就可以让版本库:file:`.git`目录下的:file:`config`文件中的配置可以覆盖用户主目录下的Git环境配置。而用户主目录下的配置也可以覆盖系统的Git配置文件。

执行前面的三个:command:`git config`命令，会看到这三个级别配置文件的格式和内容，原来Git配置文件采用的是INI文件格式。示例如下：

```
$ cat /path/to/my/workspace/demo/.git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
```

命令`:command:`git config``可以用于读取和更改INI配置文件的内容。使用命令`:command:`git config <section>.<key>``，来读取INI配置文件中某个配置的键值。例如读取`[core]`小节的`bare`的属性值，可以用如下命令：

```
$ git config core.bare
false
```

如果想更改或设置INI文件中某个属性的值也非常简单，命令格式是：`:command:`git config <section>.<key> <value>``。可以用如下操作：

```
$ git config a.b something
$ git config x.y.z others
```

如果打开`:file:`.git/config``文件，会看到如下内容：

```
[a]
b = something

[x "y"]
z = others
```

对于类似`[x "y"]`一样的配置小节，会在本书第三篇介绍远程版本库的章节中经常遇到。

从上面的介绍中，可以看到使用`:command:`git config``命令可以非常方便地操作INI文件，实际上可以用`:command:`git config``命令操作任何其他的INI文件。

- 向配置文件`:file:`test.ini``中添加配置。

```
$ GIT_CONFIG=test.ini git config a.b.c.d "hello, world"
```

- 从配置文件`:file:`test.ini``中读取配置。

```
$ GIT_CONFIG=test.ini git config a.b.c.d
hello, world
```

后面介绍的git-svn软件，就使用这个技术读写git-svn专有的配置文件。

## 思考：是谁完成的提交？

在本章的一开始，先为Git设置了`user.name`和`user.email`全局环境变量，如果不设置会有什么结果呢？

执行下面的命令，删除Git全局配置文件中关于`user.name`和`user.email`的设置：

```
$ git config --unset --global user.name
$ git config --unset --global user.email
```

这下关于用户名和邮件的设置都被清空了，执行下面的命令将看不到输出。

```
$ git config user.name  
$ git config user.email
```

下面再尝试进行一次提交，看看提交的过程会有什么不同，以及提交之后显示的提交者是谁？

在下面的命令中使用了`--allow-empty`参数，这是因为没有对工作区的文件进行任何修改，Git默认不会执行提交，使用了`--allow-empty`参数后，允许执行空白提交。

```
$ cd /path/to/my/workspace/demo  
$ git commit --allow-empty -m "who does commit?"  
[master 252dc53] who does commit?  
Committer: JiangXin <jiangxin@hp.moon.osxp.com>  
Your name and email address were configured automatically based  
on your username and hostname. Please check that they are accurate.  
You can suppress this message by setting them explicitly:  
  
git config --global user.name "Your Name"  
git config --global user.email you@example.com  
  
If the identity used for this commit is wrong, you can fix it with:  
  
git commit --amend --author='Your Name <you@example.com>'
```

喔，因为没有设置`user.name`和`user.email`变量，提交输出乱得一塌糊涂。仔细看看上面执行`git commit`命令的输出，原来Git提供了详细的帮助指引来告诉如何设置必需的变量，以及如何修改之前提交中出现的错误的提交者信息。

看看此时版本库的提交日志，会看到有两次提交。

注意：下面的输出和您的输出肯定会有所不同，一个是提交时间会不一样，再有就是由40位十六进制数字组成的提交ID也不可能一样，甚至本书中凡是您亲自完成的提交，相关的40位魔幻般的数字ID都会不一样（原因会在后面的章节看到）。因此凡是涉及数字ID和作者示例不一致的时候，以读者自己的数字ID为准，作者提供的仅是示例和参考，切记切记。

```
$ git log --pretty=fuller  
commit 252dc539b5b5f9683edd54849c8e0a246e88979c  
Author: JiangXin <jiangxin@hp.moon.osxp.com>  
AuthorDate: Mon Nov 29 10:39:35 2010 +0800  
Commit: JiangXin <jiangxin@hp.moon.osxp.com>  
CommitDate: Mon Nov 29 10:39:35 2010 +0800  
  
    who does commit?  
  
commit 9e8a761ff9dd343a1380032884f488a2422c495a  
Author: Jiang Xin <jiangxin@osxp.com>  
AuthorDate: Sun Nov 28 12:48:26 2010 +0800  
Commit: Jiang Xin <jiangxin@osxp.com>  
CommitDate: Sun Nov 28 12:48:26 2010 +0800  
  
    initialized.
```

最早的提交（下面的提交），提交者的信息是由之前设置的环境变量`user.name`和`user.email`给出的。而最新的提交（上面第一个提交）因为删除了`user.name`和`user.email`，提交时Git对提交者的用户名和邮件地址做了大胆的猜测，这个猜测可能是错的。

为了保证提交时提交者和作者信息的正确性，重新恢复`user.name`和`user.email`的设置。记

住不要照抄照搬下面的命令，请使用您自己的用户名和邮件地址。

```
$ git config --global user.name "Jiang Xin"  
$ git config --global user.email jiangxin@ossp.com
```

然后执行下面的命令，重新修改最新的提交，改正作者和提交者的错误信息。

```
$ git commit --amend --allow-empty --reset-author
```

说明：

- 参数`--amend`是对刚刚的提交进行修补，这样就可以改正前面错误的提交（用户信息错误），而不会产生另外的新提交。
- 参数`--allow-empty`是因为要进行修补的提交实际上是一个空白提交，Git默认不允许空白提交。
- 参数`--reset-author`的含义是将Author（提交者）的ID重置，否则只会影响最新的Commit（提交者）的ID。这条命令也会重置AuthorDate信息。

通过日志，可以看到最新提交的作者和提交者的信息已经改正了。

```
$ git log --pretty=fuller  
commit a0c641e92b10d8bcc1ed1bf84ca80340fdefee6  
Author: Jiang Xin <jiangxin@ossp.com>  
AuthorDate: Mon Nov 29 11:00:06 2010 +0800  
Commit: Jiang Xin <jiangxin@ossp.com>  
CommitDate: Mon Nov 29 11:00:06 2010 +0800  
  
    who does commit?  
  
commit 9e8a761ff9dd343a1380032884f488a2422c495a  
Author: Jiang Xin <jiangxin@ossp.com>  
AuthorDate: Sun Nov 28 12:48:26 2010 +0800  
Commit: Jiang Xin <jiangxin@ossp.com>  
CommitDate: Sun Nov 28 12:48:26 2010 +0800  
  
initialized.
```

## 思考：随意设置提交者姓名，是否太不安全？

使用过CVS、Subversion等集中式版本控制系统的用户会知道，每次提交的时候须要认证，认证成功后，登录ID就作为提交者ID出现在版本库的提交日志中。很显然，对于CVS或Subversion这样的版本控制系统，很难冒充他人提交。那么像Git这样的分布式版本控制系统，可以随心所欲的设定提交者，这似乎太不安全了。

Git可以随意设置提交的用户名和邮件地址信息，这是分布式版本控制系统的特性使然，每个人都是自己版本库的主人，很难也没有必要进行身份认证从而使用经过认证的用户名作为提交的用户名。

在进行“独奏”的时候，还要为自己强制加上一个“指纹识别”实在是太没有必要了。但是团队合作时授权就成为必需了。不过一般来说，设置的Git服务器只会在个人向服务器版本库执行推送操作（推送其本地提交）的时候进行身份认证，并不对所推送的提交本身所包含的用户名作出检查。但Android项目是个例外。

Android项目为了更好的使用Git实现对代码的集中管理，开发了一套叫做Gerrit的审核服务器来管理Git提交，对提交者的邮件地址进行审核。例如下面的示例中在向Gerrit服务器推送的时候，提交中的提交者邮件地址为`jiangxin@ossp.com`，但是在Gerrit中注册用户时使用的邮件地址为`jiangxin@moon.ossp.com`。因为两者不匹配，从而导致推送失败。

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 ! [remote rejected] master -> master (you are not committer jiangxin@ossp.com)
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```

即使没有使用类似Gerrit的服务，作为提交者也不应该随意改变`user.name`和`user.email`的环境变量设置，因为当多人协同时这会给他造成迷惑，也会给一些项目管理软件造成麻烦。

例如Redmine是一款实现需求管理和缺陷跟踪的项目管理软件，可以和Git版本库实现整合。Git的提交可以直接关闭Redmine上的Bug，还有Git的提交可以反映出项目成员的工作进度。Redmine中的用户（项目成员）是用一个ID做标识，而Git的提交者则用一个包含用户名和邮件地址的字符串，如何将Redmine的用户和Git提交者相关联呢？Redmine提供了一个配置界面用于设置二者之间的关系，如图4-1所示。



图 4 1: Redmine中用户ID和Git提交者关联

显然如果在Git提交时随意变更提交者的姓名和邮件地址，会破坏Redmine软件中设置好的用户对应关系。

## 思考：命令别名是什么的？

在本章的一开始，通过对`alias.ci`等Git环境变量的设置，为Git设置了命令别名。命令别名可以帮助用户解决从其他版本控制系统迁移到Git后的使用习惯问题。像CVS和Subversion在提交的时候，一般习惯使用`ci`（check in）子命令，在检出的时候则习惯使用`co`（check out）子命令。如果Git不能提供对`ci`和`co`这类简洁命令的支持，对于拥有其他版本控制系统使用经验的用户来说，Git的用户体验就会打折扣。幸好聪明的Git提供了别名机制，可以满足用户特殊的使用习惯。

本章前面列出的四条别名设置指令，创建的是最常用的几个Git别名。实际上别名还可以包含命令参数。例如下面的别名设置指令：

```
$ git config --global alias.ci "commit -s"
```

如上设置后，当使用`git ci`命令提交的时候，会自动带上`-s`参数，这样会在提交的说明中自动添加上包含提交者姓名和邮件地址的签名人标识，类似于`Signed-off-by: User Name`

<email@address>。这对于一些项目（Git、Linux kernel、Android等）来说是必要甚至是必须的。

不过在本书会尽量避免使用别名命令，以免由于读者因为尚未设置别名而造成学习上的困惑。

## 备份本章的工作成果

执行下面的命令，算是对本章工作成果的备份。

```
$ cd /path/to/my/workspace  
$ git clone demo demo-step-1  
Cloning into demo-step-1...  
done.
```

[1] 命令:`git commit -s` 中的参数-s含义为在提交说明的最后添加“Signed-off-by:”签名。

[2] 命令:`git -p status` 中的参数-p含义是为:`git status` 命令的输出添加分页器。

[3] 大家实际操作中看到的ID肯定和这里写的不一样，具体原因会在后面的“6.1 Git对象库探秘”一节中予以介绍。如果碰巧您的操作显示出了同样的ID（78cde45），那么我建议您赶紧去买一张彩票。;)

[4] 示例中使用了Linux下的:`strace` 命令跟踪系统调用，在Mac OS X下则可使用:`sudo dtruss git status` 命令跟踪相关Git操作的系统调用。

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/010-git-init.rst>

## Git暂存区

上一章主要学习了三个命令：`:command:`git init``、`:command:`git add``和`:command:`git commit``，这三条命令可以说是版本库创建的三部曲。同时还通过对几个问题的思考，使读者能够了解Git版本库在工作区中的布局，Git三个等级的配置文件以及Git的别名命令等内容。

在上一章的实践中，DEMO版本库经历了两次提交，可以用`:command:`git log``查看提交日志（附加的`--stat`参数看到每次提交的文件变更统计）。

```
$ cd /path/to/my/workspace/demo
$ git log --stat
commit a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Mon Nov 29 11:00:06 2010 +0800

    who does commit?

commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Sun Nov 28 12:48:26 2010 +0800

    initialized.

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

可以看到第一次提交对文件`:file:`welcome.txt``有一行的变更，而第二次提交因为是使用了`--allow-empty`参数进行的一次空提交，所以提交说明中看不到任何对实质内容的修改。

下面仍在这个工作区，继续新的实践。通过新的实践学习Git一个最重要的概念：“暂存区”。

## 修改不能直接提交？

首先更改`:file:`welcome.txt``文件，在这个文件后面追加一行。可以使用下面的命令实现内容的追加。

```
$ echo "Nice to meet you." >> welcome.txt
```

这时可以通过执行`:command:`git diff``命令看到修改后的文件和版本库中文件的差异。  
(实际上这句话有问题，和本地比较的不是版本库中的文件，而是一个中间状态的文件)

```
$ git diff
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
Hello.
+Nice to meet you.
```

差异输出是不是很熟悉？在之前介绍版本库“黑暗的史前时代”的时候，曾经展示了`:command:`diff``命令的输出，两者的格式是一样的。

既然文件修改了，那么就提交吧。提交能够成功么？

```
$ git commit -m "Append a nice line."
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

提交成功了么？好像没有！

提交没有成功的证据：

- 先来看看提交日志，如果提交成功，应该有新的提交记录出现。

下面使用了精简输出来显示日志，以便更简洁和清晰的看到提交历史。在其中能够看出来版本库中只有两个提交，都是在上一章的实践中完成的。也就是说刚才针对修改文件的提交没有成功！

```
$ git log --pretty=oneline
a0c641e92b10d8bcc1ed1bf84ca80340fdefee6 who does commit?
9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

- 执行`:command:`git diff``可以看到和之前相同的差异输出，这也说明了之前的提交没有成功。
- 执行`:command:`git status``查看文件状态，也可以看到文件处于未提交的状态。  
而且`:command:`git status``命令的输出和`:command:`git commit``提交失败的输出信息一模一样！
- 对于习惯了像 CVS 和 Subversion 那样简练的状态输出的用户，可以在执行`:command:`git status``时附加上`-s`参数，显示精简格式的状态输出。

```
$ git status -s
M welcome.txt
```

提交为什么会失败呢？再回过头来仔细看看刚才`:command:`git commit``命令提交失败后的输出：

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

把它翻译成中文普通话：

```
# 位于您当前工作的分支 master 上
# 下列的修改还没有加入到提交任务（提交暂存区，stage）中，不会被提交；
#   （使用 "git add <file>..." 命令后，改动就会加入到提交任务中，
#     要在下一次提交操作时才被提交）
#   （使用 "git checkout -- <file>..." 命令，工作区当前您不打算提交的
#     修改会被彻底清除！！！）
#
```

```
#      已修改: welcome.txt
#
# 警告: 提交任务是空的嘛, 您不要再打扰我啦
#       (除非使用 "git add" 和/或 "git commit -a" 命令)
```

也就是说要对修改的:`:file:`welcome.txt``文件执行:`:command:`git add``命令, 将修改的文件添加到“提交任务”中, 然后才能提交!

这个行为真的很奇怪, 因为add操作对于其他版本控制系统来说是向版本库添加新文件用的, 修改的文件(已被版本控制跟踪的文件)在下次提交时会直接被提交。Git的这个古怪的行为会在下面的介绍中找到答案, 读者会逐渐习惯并喜欢Git的这个设计。

好了, 现在就将修改的文件“添加”到提交任务中吧:

```
$ git add welcome.txt
```

现在再执行一些Git命令, 看看当执行“添加”动作后, Git库发生了什么变化:

- 执行:`:command:`git diff``没有输出, 难道是被提交了? 可是只是执行了“添加”到提交任务的操作, 相当于一个“登记”的命令, 并没有执行提交哇?

```
$ git diff
```

- 这时如果和HEAD(当前版本库的头指针)或者master分支(当前工作分支)进行比较, 会发现有差异。这个差异才是正常的, 因为尚未真正提交么。

```
$ git diff HEAD
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
 Hello.
+Nice to meet you.
```

- 执行:`:command:`git status``命令, 状态输出和之前的不一样了。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   welcome.txt
#
```

再对新的Git状态输出做一回翻译:

```
$ git status
# 位于分支 master 上
# 下列的修改将被提交:
#   (如果你后悔了, 可以使用 "git reset HEAD <file>..." 命令
#    将下列改动撤出提交任务(提交暂存区, stage), 否则
#    执行提交命令可真的要提交喽)
#
#      已修改:   welcome.txt
#
```

不得不说, Git太人性化了, 它把各种情况下可以使用到的命令都告诉给用户了, 虽然这显得有点罗嗦。如果不要这么罗嗦, 可以用简洁方式显示状态:

```
$ git status -s  
M welcome.txt
```

上面精简的状态输出与执行:command:`git add`之前的精简状态输出相比，有细微的差别，发现了么？

- 虽然都是 M (Modified) 标识，但是位置不一样。在执行:command:`git add`命令之前，这个M位于第二列（第一列是一个空格），在执行完:command:`git add`之后，字符M位于第一列（第二列是空白）。
- 位于第一列的字符M的含义是：版本库中的文件和处于中间状态——提交任务（提交暂存区，即stage）中的文件相比有改动。
- 位于第二列的字符M的含义是：工作区当前的文件和处于中间状态——提交任务（提交暂存区，即stage）中的文件相比也有改动。

是不是还有一些不明白？为什么Git的状态输出中提示了那么多让人不解的命令？为什么存在一个提交任务的概念而又总是把它叫做暂存区（stage）？不要紧，马上就会专题讲述“暂存区”的概念。当了解了Git版本库的设计原理之后，理解相关Git命令就易如反掌了。

这时如果直接提交(:command:`git commit`)，加入提交任务的:file:`welcome.txt`文件的更改就被提交入库了。但是先不忙着执行提交，再进行一些操作，看看能否被彻底的搞糊涂。

- 继续修改一下:file:`welcome.txt`文件（在文件后面再追加一行）。

```
$ echo "Bye-Bye." >> welcome.txt
```

- 然后执行:command:`git status`，查看一下状态：

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified:   welcome.txt  
#  
# Changes not staged for commit:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working direct  
#  
#       modified:   welcome.txt  
#
```

状态输出中居然是之前出现的两种不同状态输出的魂附体。

- 如果显示精简的状态输出，也会看到前面两种精简输出的杂合体。

```
$ git status -s  
MM welcome.txt
```

上面的更为复杂的 Git 状态输出可以这么理解：不但版本库中最新提交的文件和处于中间状态——提交任务（提交暂存区，stage）中的文件相比有改动，而且工作区当前的文件和处于中间状态——提交任务（提交暂存区，stage）中的文件相比也有改动。

即现在:file:`welcome.txt`有三个不同的版本，一个在工作区，一个在等待提交的暂存区，还有一个是版本库中最新版本的:file:`welcome.txt`。通过不同的参数调

用:command:`git diff`命令可以看到不同版本库:file:`welcome.txt`文件的差异。

- 不带任何选项和参数调用:command:`git diff`显示工作区最新改动，即工作区和提交任务（提交暂存区，stage）中相比的差异。

```
$ git diff
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
Hello.
Nice to meet you.
+Bye-Bye.
```

- 将工作区和HEAD（当前工作分支）相比，会看到更多的差异。

```
$ git diff HEAD
diff --git a/welcome.txt b/welcome.txt
index 18832d3..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,3 @@
Hello.
+Nice to meet you.
+Bye-Bye.
```

- 通过参数--cached或者--staged参数调用:command:`git diff`命令，看到的是提交暂存区（提交任务，stage）和版本库中文件的差异。

```
$ git diff --cached
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
Hello.
+Nice to meet you.
```

好了现在是时候提交了。现在执行:command:`git commit`命令进行提交。

```
$ git commit -m "which version checked in?"
[master e695606] which version checked in?
 1 files changed, 1 insertions(+), 0 deletions(-)
```

这次提交终于成功了。如何证明提交成功了呢？

- 通过查看提交日志，看到了新的提交。

```
$ git log --pretty=oneline
e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
a0c641e92b10d8bcc1ed1bf84ca80340fdefee6 who does commit?
9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

- 查看精简的状态输出。

状态输出中文件名的前面出现了一个字母M，即只位于第二列的字母M。

```
$ git status -s
 M welcome.txt
```

那么第一列的M哪里去了？被提交了呗。即提交任务（提交暂存区，stage）中的内容被提交到版本库中，所以第一列因为提交暂存区（提交任务，stage）和版本库中的状态一致，所以显示一个空白。

提交的:`:file:`welcome.txt``是哪个版本呢？可以通过执行:`:command:`git diff``或者`:command:`git diff HEAD``命令查看差异。虽然命令`:command:`git diff``和`:command:`git diff HEAD``的比较过程并不相同（可以通过`:command:`strace``命令跟踪命令执行过程中的文件访问），但是会看到下面相同的差异输出结果。

```
$ git diff  
diff --git a/welcome.txt b/welcome.txt  
index fd3c069..51dbfd2 100644  
--- a/welcome.txt  
+++ b/welcome.txt  
@@ -1,2 +1,3 @@  
Hello.  
Nice to meet you.  
+Bye-Bye.
```

## 理解 Git 暂存区 (stage)

把上面的实践从头至尾走一遍，不知道读者的感想如何？

- “被眼花缭乱的Git魔法彻底搞糊涂了？”
- “Git为什么这么折磨人，修改的文件直接提交不就完了么？”
- “看不出Git这么做有什么好处？”

在上面的实践过程中，有意无意的透漏了“暂存区”的概念。为了避免用户被新概念吓坏，在暂存区出现的地方用同时使用了“提交任务”这一更易理解的概念，但是暂存区(stage，或称为index)才是其真正的名称。我认为Git暂存区(stage，或称为index)的设计是Git最成功的设计之一，也是最难理解的一个设计。

在版本库:`:file:`.git``目录下，有一个`:file:`index``文件，下面针对这个文件做一个有趣的试验。要说明的是：这个试验是用1.7.3版本的Git进行的，低版本的Git因为没有针对`:command:`git status``命令进行优化设计，需要使用`:command:`git diff``命令，才能看到`:file:`index``文件的日期戳变化。

首先执行`:command:`git checkout``命令（后面会介绍此命令），撤销工作区中`welcome.txt`文件尚未提交的修改。

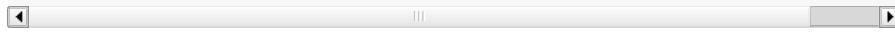
```
$ git checkout -- welcome.txt  
$ git status -s      # 执行 git diff，如果 git 版本号小于 1.7.3
```

通过状态输出，可以看到工作区已经没有改动了。查看一下`:command:`.git/index``文件，注意该文件的时间戳为：19:37:44。

```
$ ls --full-time .git/index  
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:37:44.625246224 +0800.git/in
```

再次执行`:command:`git status``命令，然后显示`:file:`.git/index``文件的时间戳为：19:37:44，和上面的一样。

```
$ git status -s      # 执行 git diff，如果 git 版本号小于 1.7.3  
$ ls --full-time .git/index  
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:37:44.625246224 +0800 .git/i
```



现在更改一下 welcome.txt 的时间戳，但是不改变它的内容。然后再执行:command:`git status` 命令，然后查看:file:`.git/index` 文件时间戳为: 19:42:06。

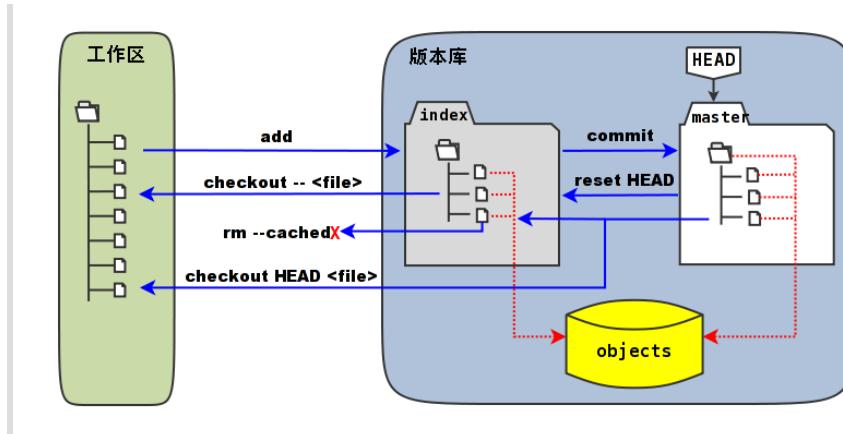
```
$ touch welcome.txt
$ git status -s      # 执行 git diff , 如果 git 版本号小于 1.7.3
$ ls --full-time .git/index
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:42:06.980243216 +0800 .git/i
```



看到了么，时间戳改变了！

这个试验说明当执行:command:`git status` 命令（或者:command:`git diff` 命令）扫描工作区改动的时候，先依据:file:`.git/index` 文件中记录的（工作区跟踪文件的）时间戳、长度等信息判断工作区文件是否改变。如果工作区的文件时间戳改变，说明文件的内容可能被改变了，需要打开文件，读取文件内容，和更改前的原始文件相比较，判断文件内容是否被更改。如果文件内容没有改变，则将该文件新的时间戳记录到:file:`.git/index` 文件中。因为判断文件是否更改，使用时间戳、文件长度等信息进行比较要比通过文件内容比较要快的多，所以Git这样的实现方式可以让工作区状态扫描更快速的执行，这也是Git高效的因素之一。

文件:file:`.git/index` 实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等）。文件的内容并不存储其中，而是保存在Git对象库:file:`.git/objects` 目录中，文件索引建立了文件和对象库中对象实体之间的对应。下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系。



工作区、版本库、暂存区原理图

在这个图中，可以看到部分Git命令是如何影响工作区和暂存区（stage，亦称index）的。下面就对这些命令进行简要的说明，而要彻底揭开这些命令的面纱要在接下来的几个章节。

- 图中左侧为工作区，右侧为版本库。在版本库中标记为index的区域是暂存区（stage，亦称index），标记为master的是master分支所代表的目录树。
- 图中可以看出此时HEAD实际是指向master分支的一个“游标”。所以图示的命令中出现HEAD的地方可以用master来替换。
- 图中的objects标识的区域为Git的对象库，实际位于:file:`.git/objects` 目录下，会在后面的章节重点介绍。
- 当对工作区修改（或新增）的文件执行:command:`git add` 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。
- 当执行提交操作（:command:`git commit`）时，暂存区的目录树写到版本库（对象库）中，master分支会做相应的更新。即master最新指向的目录树就是提交时原暂存

区的目录树。

- 当执行`:command:`git reset HEAD``命令时，暂存区的目录树会被重写，被master分支指向的目录树所替换，但是工作区不受影响。
- 当执行`:command:`git rm --cached <file>``命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行`:command:`git checkout .``或者`:command:`git checkout -- <file>``命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行`:command:`git checkout HEAD .``或者`:command:`git checkout HEAD <file>``命令时，会用HEAD指向的master分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

## Git Diff魔法

在本章的实践中展示了具有魔法效果的命令：`:command:`git diff``。在不同参数的作用下，`:command:`git diff``的输出并不相同。在理解了Git中的工作区、暂存区、和版本库最新版本（当前分支）分别是三个不同的目录树后，就非常好理解`:command:`git diff``魔法般的行为。

### 暂存区目录树的浏览

有什么办法能够像查看工作区一样的，直观的查看暂存区以及HEAD当中的目录树么？

对于HEAD（版本库中当前提交）指向的目录树，可以使用Git底层命令`:command:`git ls-tree``来查看。

```
$ git ls-tree -l HEAD
100644 blob fd3c069c1de4f4bc9b15940f490aeb48852f3c42      25    welcome.txt
```

其中：

- 使用`-l`参数，可以显示文件的大小。上面`:file:`welcome.txt``大小为25字节。
- 输出的`:file:`welcome.txt``文件条目从左至右，第一个字段是文件的属性(rw-r--r--)，第二个字段说明是Git对象库中的一个blob对象（文件），第三个字段则是该文件在对象库中对应的ID——一个40位的SHA1哈希值格式的ID（这个会在后面介绍），第四个字段是文件大小，第五个字段是文件名。

在浏览暂存区中的目录树之前，首先清除工作区当中的改动。通过`:command:`git clean -fd``命令清除当前工作区中没有加入版本库的文件和目录（非跟踪文件和目录），然后执行`:command:`git checkout .``命令，用暂存区内容刷新工作区。

```
$ cd /path/to/my/workspace/demo
$ git clean -fd
$ git checkout .
```

然后开始在工作区中做出一些修改（修改`:file:`welcome.txt``，增加一个子目录和文件），然后添加到暂存区。最后再对工作区做出修改。

```
$ echo "Bye-Bye." >> welcome.txt
$ mkdir -p a/b/c
$ echo "Hello." > a/b/c/hello.txt
$ git add .
$ echo "Bye-Bye." >> a/b/c/hello.txt
$ git status -s
AM a/b/c/hello.txt
M welcome.txt
```

上面的命令运行完毕后，通过精简的状态输出，可以看出工作区、暂存区、和版本库当前分支的最新版本（HEAD）各不相同。先来看看工作区中文件的大小：

```
$ find . -path ./git -prune -o -type f -printf "%-20p\t%s\n"
./welcome.txt      34
./a/b/c/hello.txt  16
```

要显示暂存区的目录树，可以使用`:command:`git ls-files``命令。

```
$ git ls-files -s
100644 18832d35117ef2f013c4009f5b2128dfa354f 0      a/b/c/hello.txt
100644 51dbfd25a804c30e9d8dc441740452534de8264b 0      welcome.txt
```

注意这个输出和之前使用`:command:`git ls-tree``命令输出不一样，如果想要使用`:command:`git ls-tree``命令，需要先将暂存区的目录树写入Git对象库（用`:command:`git write-tree``命令），然后在针对`:command:`git write-tree``命令写入的 tree 执行`:command:`git ls-tree``命令。

```
$ git write-tree
9431f4a3f3e1504e03659406faa9529f83cd56f8
$ git ls-tree -l 9431f4a
040000 tree 53583ee687fb2e913d18d508aefd512465b2092      -      a
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b      34      welcome.txt
```

从上面的命令可以看出：

- 到处都是40位的SHA1哈希值格式的ID，可以用于指代文件内容（blob），用于指代目录树（tree），还可以用于指代提交。但什么是SHA1哈希值ID，作用是什么，这些疑问暂时搁置，下一章再揭晓。
- 命令`:command:`git write-tree``的输出就是写入Git对象库中的Tree ID，这个ID将作为下一条命令的输入。
- 在`:command:`git ls-tree``命令中，没有把40位的ID写全，而是使用了前几位，实际上只要不和其他的对象ID冲突，可以随心所欲的使用缩写ID。
- 可以看到`:command:`git ls-tree``的输出显示的第一条是一个tree对象，即刚才创建的一级目录`:file:`a``。

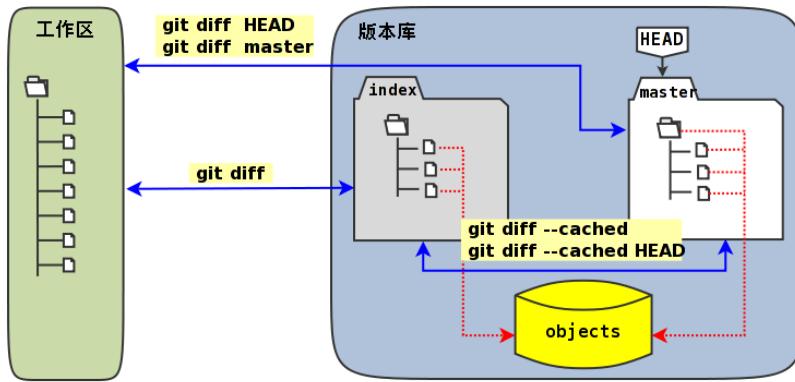
如果想要递归显示目录内容，则使用`-r`参数调用。使用参数`-t`可以把递归过程遇到的每棵树都显示出来，而不仅是显示最终的文件。下面执行递归操作显示目录树的内容。

```
$ git write-tree | xargs git ls-tree -l -r -t
040000 tree 53583ee687fb2e913d18d508aefd512465b2092      -      a
040000 tree 514d729095b7bc203cf336723af710d41b84867b      -      a/b
040000 tree deaec688e84302d4a0b98a1b78a434be1b22ca02      -      a/b/c
100644 blob 18832d35117ef2f013c4009f5b2128dfa354f      7      a/b/c/hello.t
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b      34      welcome.txt
```

好了现在工作区，暂存区和HEAD三个目录树的内容各不相同。下面的表格总结了不同文件在三个目录树中的文件大小。

文件名	工作区	暂存区	HEAD
welcome.txt	34 字节	34 字节	25 字节
a/b/c/hello.txt	16 字节	7 字节	0 字节

通过使用不同的参数调用`git diff`命令，可以对工作区、暂存区、HEAD中的内容两两比较。下面的这个图，展示了不同的`git diff`命令的作用范围。



通过上面的图，就不难理解下面`git diff`命令不同的输出结果了。

- 工作区和暂存区比较。

```
$ git diff
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
index 18832d3..e8577ea 100644
--- a/a/b/c/hello.txt
+++ b/a/b/c/hello.txt
@@ -1 +1,2 @@
Hello.
+Bye-Bye.
```

- 暂存区和HEAD比较。

```
$ git diff --cached
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
new file mode 100644
index 0000000..18832d3
--- /dev/null
+++ b/a/b/c/hello.txt
@@ -0,0 +1 @@
+Hello.

diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
Hello.
Nice to meet you.
+Bye-Bye.
```

- 工作区和HEAD比较。

```
$ git diff HEAD
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
new file mode 100644
index 0000000..e8577ea
--- /dev/null
+++ b/a/b/c/hello.txt
@@ -0,0 +1,2 @@
+Hello.
+Bye-Bye.

diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
```

```
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
Hello.
Nice to meet you.
+Bye-Bye.
```

## 不要使用:command:`git commit -a`

实际上Git的提交命令(:command:`git commit`)可以带上-a参数，对本地所有变更的文件执行提交操作，包括本地修改的文件，删除的文件，但不包括未被版本库跟踪的文件。

这个命令的确可以简化一些操作，减少用:command:`git add`命令标识变更文件的步骤，但是如果习惯了使用这个“偷懒”的提交命令，就会丢掉Git暂存区带给用户最大的好处：对提交内容进行控制的能力。

有的用户甚至通过别名设置功能，创建指向命令:command:`git commit -a`的别名ci，这更是不可取的行为，应严格禁止。在本书会很少看到使用:command:`git commit -a`命令。

## 搁置问题，暂存状态

查看一下当前工作区的状态。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#       modified:   welcome.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   a/b/c/hello.txt
#
```

在状态输出中Git体贴的告诉了用户如何将加入暂存区的文件从暂存区撤出以便让暂存区和HEAD一致（这样提交就不会发生），还告诉用户对于暂存区更新后在工作区所做的再一次的修改有两个选择：或者再次添加到暂存区，或者取消工作区新做出的改动。但是涉及到的命令现在理解还有些难度，一个是:command:`git reset`，一个是:command:`git checkout`。需要先解决什么是HEAD，什么是master分支以及Git对象存储的实现机制等问题，才可以更好的操作暂存区。

为此，我作出一个非常艰难的决定[1]：就是保存当前的工作进度，在研究了HEAD和master分支的机制之后，继续对暂存区的探索。命令:command:`git stash`就是用于保存当前工作进度的。

```
$ git stash
Saved working directory and index state WIP on master: e695606 which version
HEAD is now at e695606 which version checked in?
```

运行完:command:`git stash`之后，再查看工作区状态，会看见工作区尚未提交的改动（包括暂存区的改动）全都不见了。

```
$ git status  
# On branch master  
nothing to commit (working directory clean)
```

“I'll be back” —— 施瓦辛格, 《终结者》, 1984.

[1] 此句式模仿2010年11月份发生的“3Q大战”。参见: <http://zh.wikipedia.org/wiki/奇虎360与腾讯QQ争斗事件>。  
来源: <https://github.com/gotgit/gotgit/blob/master/02-git-solo/020-git-stage.rst>

# Git对象

在上一章学习了Git的一个最重要的概念：暂存区（stage，亦称index）。暂存区是一个介于工作区和版本库的中间状态，当执行提交时实际上是将暂存区的内容提交到版本库中，而且Git很多命令都会涉及到暂存区的概念，例如：`:command:`git diff`` 命令。

但是上一章还是留下了很多疑惑，例如什么是HEAD？什么是master？为什么它们二者（在上一章）可以相互替换使用？为什么Git中的很多对象像提交、树、文件内容等都用40位的SHA1哈希值来表示？这一章的内容将会揭开这些奥秘，并且还会画出一个更为精确的版本库结构图。

## Git对象库探秘

在前面刻意回避了对提交ID的说明，现在是时候来揭开由40位十六进制数字组成的“魔幻数字”的奥秘了。

通过查看日志的详尽输出，会惊讶的看到非常多的“魔幻数字”，这些“魔幻数字”实际上是SHA1哈希值。

```
$ git log -1 --pretty=raw
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800

    which version checked in?
```

一个提交中居然包含了三个SHA1哈希值表示的对象ID。

- commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86

这是本次提交的唯一标识。

- tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9

这是本次提交所对应的目录树。

- parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6

这是本地提交的父提交（上一次提交）。

研究Git对象ID的一个重量级武器就是：`:command:`git cat-file`` 命令。用下面的命令可以

查看一下这三个ID的类型。

```
$ git cat-file -t e695606  
commit  
$ git cat-file -t f58d  
tree  
$ git cat-file -t a0c6  
commit
```

在引用对象ID的时候，没有必要把整个40位ID写全，只需要从头开始的几位不冲突即可。

下面再用:command:`git cat-file`命令查看一下这几个对象的内容。

- commit对象e695606fc5e31b2ff9038a48a3d363f4c21a3d86

```
$ git cat-file -p e695606  
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9  
parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6  
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800  
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
```

which version checked in?

- tree对象f58da9a820e3fd9d84ab2ca2f1b467ac265038f9

```
$ git cat-file -p f58da9a  
100644 blob fd3c069c1de4f4bc9b15940f490aeb48852f3c42      welcome.txt
```

- commit对象a0c641e92b10d8bcc1ed1bf84ca80340fdefee6

```
$ git cat-file -p a0c641e  
tree 190d840dd3d8fa319bdec6b8112b0957be7ee769  
parent 9e8a761ff9dd343a1380032884f488a2422c495a  
author Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800  
committer Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800
```

who does commit?

在上面目录树(tree)对象中看到了一个新的类型的对象：blob对象。这个对象保存着文件:file:`welcome.txt`的内容。用:command:`git cat-file`研究一下。

- 该对象的类型为blob。

```
$ git cat-file -t fd3c069c1de4f4bc9b15940f490aeb48852f3c42
blob
```

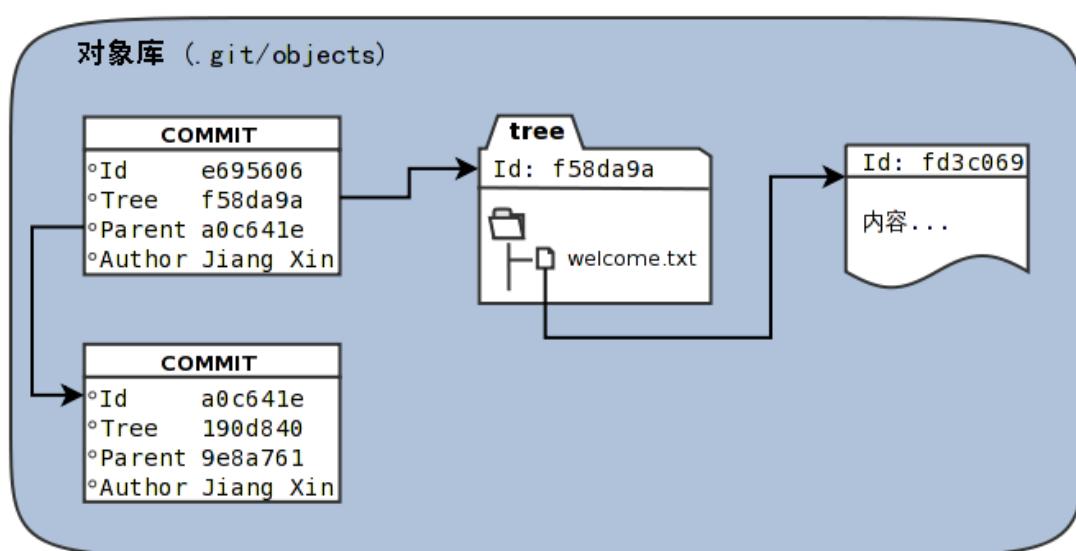
- 该对象的内容就是:`:file:`welcome.txt``文件的内容。

```
$ git cat-file -p fd3c069c1de4f4bc9b15940f490aeb48852f3c42
Hello.
Nice to meet you.
```

这些对象保存在哪里？当然是Git库中的:`:file:`objects``目录下了（ID的前两位作为目录名，后38位作为文件名）。用下面的命令可以看到这些对象在对象库中的实际位置。

```
$ for id in e695606 f58da9a a0c641e fd3c069; do \
  ls .git/objects/${id:0:2}/${id:2}*; done
.git/objects/e6/95606fc5e31b2ff9038a48a3d363f4c21a3d86
.git/objects/f5/8da9a820e3fd9d84ab2ca2f1b467ac265038f9
.git/objects/a0/c641e92b10d8bcc1ed1bf84ca80340fdefee6
.git/objects/fd/3c069c1de4f4bc9b15940f490aeb48852f3c42
```

下面的图示更加清楚的显示了Git对象库中各个对象之间的关系。



从上面的图示中很明显的看出提交（Commit）对象之间相互关联，通过相互之间的关联则很容易的识别出一条跟踪链。这条跟踪链可以在运行:`:command:`git log``命令时，通过使用`--graph`参数看到。下面的命令还使用了`--pretty=raw`参数以便显示每个提交对象的`parent`属性。

```
$ git log --pretty=raw --graph e695606
```

```
* commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
| tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
| parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
| author Jiang Xin <jiangxin@ossp.com> 1291022581 +0800
| committer Jiang Xin <jiangxin@ossp.com> 1291022581 +0800

|     which version checked in?

* commit a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
| tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
| parent 9e8a761ff9dd343a1380032884f488a2422c495a
| author Jiang Xin <jiangxin@ossp.com> 1290999606 +0800
| committer Jiang Xin <jiangxin@ossp.com> 1290999606 +0800

|     who does commit?

* commit 9e8a761ff9dd343a1380032884f488a2422c495a
tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
author Jiang Xin <jiangxin@ossp.com> 1290919706 +0800
committer Jiang Xin <jiangxin@ossp.com> 1290919706 +0800

initialized.
```

最后一个提交没有parent属性，所以跟踪链到此终结，这实际上就是提交的起点。

### 现在来看看HEAD和master的奥秘吧

因为在上一章的最后执行了:command:`git stash`将工作区和暂存区的改动全部封存起来，所以执行下面的命令会看到工作区和暂存区中没有改动。

```
$ git status -s -b
## master
```

说明：上面在显示工作区状态时，除了使用了-s参数以显示精简输出外，还使用了-b参数以便能够同时显示出当前工作分支的名称。这个-b参数是在Git 1.7.2以后加入的新的参数。

下面的:command:`git branch`是分支管理的主要命令，也可以显示当前的工作分支。

```
$ git branch
* master
```

在master分支名称前面出现一个星号表明这个分支是当前工作分支。至于为什么没有其他分支以及什么叫做分支，会在本书后面的章节揭晓。

现在连续执行下面的三个命令会看到相同的输出：

```
$ git log -1 HEAD
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
$ git log -1 master
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
$ git log -1 refs/heads/master
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
```

也就是说在当前版本库中，HEAD、master和refs/heads/master具有相同的指向。现在到版本库（:file:`.git` 目录）中一探它们的究竟。

```
$ find .git -name HEAD -o -name master
.git/HEAD
.git/logs/HEAD
.git/logs/refs/heads/master
.git/refs/heads/master
```

找到了四个文件，其中在:file:`.git/logs` 目录下的文件稍后再予以关注，现在把目光锁定在:file:`.git/HEAD` 和:file:`.git/refs/heads/master` 上。

显示一下:file:`.git/HEAD` 的内容：

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

把HEAD的内容翻译过来就是：“指向一个引用：refs/heads/master”。这个引用在哪里？当然是文件:file:`.git/refs/heads/master` 了。

看看文件:file:`.git/refs/heads/master` 的内容。

```
$ cat .git/refs/heads/master
```

```
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

显示的e695606...所指为何物？用:[command: `git cat-file`](#)命令进行查看。

- 显示SHA1哈希值指代的数据类型。

```
$ git cat-file -t e695606
commit
```

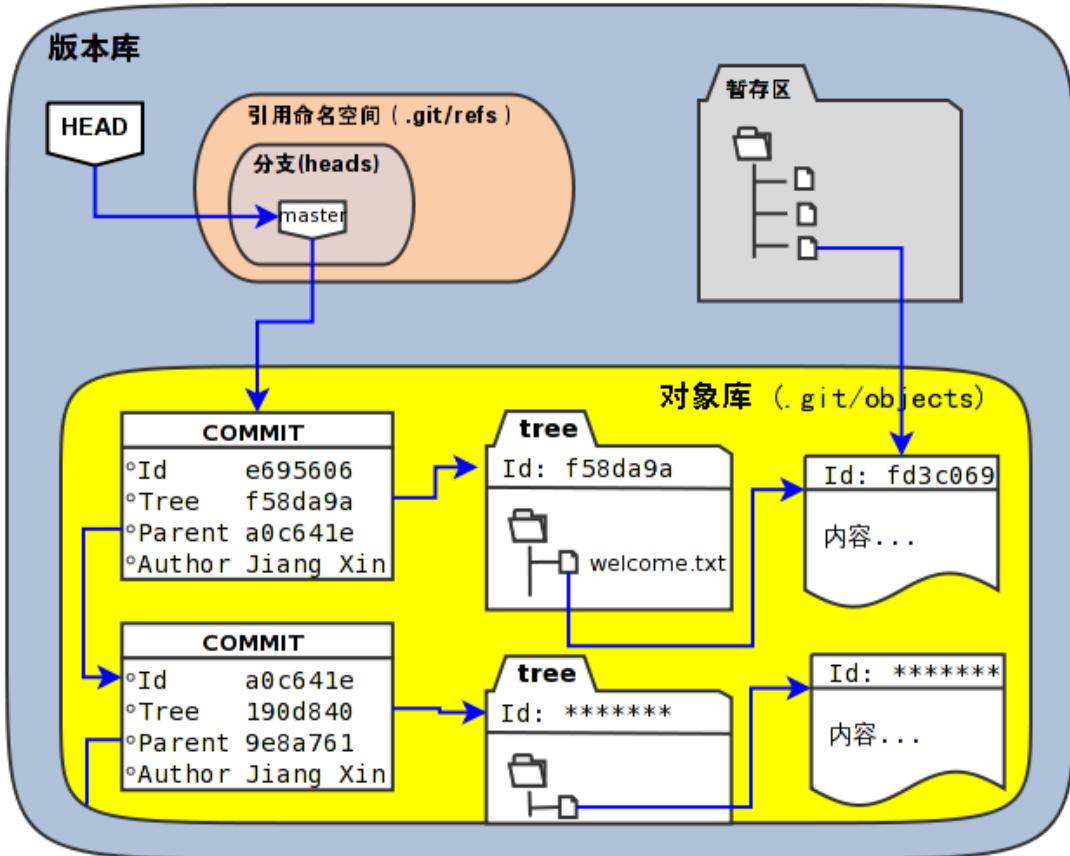
- 显示该提交的内容。

```
$ git cat-file -p e695606fc5e31b2ff9038a48a3d363f4c21a3d86
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800

which version checked in?
```

原来分支master指向的是一个提交ID（最新提交）。这样的分支实现是多么的巧妙啊：既然可以从任何提交开始建立一条历史跟踪链，那么用一个文件指向这个链条的最新提交，那么这个文件就可以用于追踪整个提交历史了。这个文件就是:[:file: ` .git/refs/heads/master`](#)文件。

下面看一个更接近于真实的版本库结构图：



目录:`.git/refs`是保存引用的命名空间，其中:`.git/refs/heads`目录下的引用又称为分支。对于分支既可以使用正规的长格式的表示法，如:`refs/heads/master`，也可以去掉前面的两级目录用`master`来表示。Git 有一个底层命令:`git rev-parse`可以用于显示引用对应的提交ID。

```
$ git rev-parse master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git rev-parse refs/heads/master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git rev-parse HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

可以看出它们都指向同一个对象。为什么这个对象是40位，而不是更少或者更多？这些ID是如何生成的呢？

## 问题：SHA1哈希值到底是什么，如何生成？

哈希(hash)是一种数据摘要算法(或称散列算法)，是信息安全领域当中重要的理论基石。该算法将任意长度的输入经过散列运算转换为固定长度的输出。固定长度的输出可以称为对应的输入的数字摘要或哈希值。例如SHA1摘要算法可以处理从零到一千多万个TB的

输入数据，输出为固定的160比特的数字摘要。两个不同内容的输入即使数据量非常大、差异非常小，两者的哈希值也会显著不同。比较著名的摘要算法有：MD5和SHA1。Linux下：`command: `shasum`` 命令可以用于生成摘要。

```
$ echo -n Git |sha1sum  
5819778898df55e3a762f0c5728b457970d72cae -
```

可以看出字符串Git的SHA1哈希值为40个十六进制的数字组成。那么能不能找出另外一个字符串使其SHA1哈希值和上面的哈希值一样呢？下面看看难度有多大。

每个十六进制的数字用于表示一个4位的二进制数字，因此40位的SHA1哈希值的输出为实为160bit。拿双色球博彩打一个比喻，要想制造相同的SHA1哈希值就相当于要选出32个“红色球”，每个红球有1到32个（5位的二进制数字）选择，而且红球之间可以重复。相比“双色球博彩”总共只需选出7颗球，SHA1“中奖”的难度就相当于要连续购买五期“双色球”并且必须每一期都要中一等奖。当然由于算法上的问题，制造冲突（相同数字摘要）的几率没有那么小，但是已经足够小，能够满足Git对不同对象的进行区分和标识了。即使有一天像发现了类似MD5摘要算法漏洞那样，发现了SHA1算法存在人为制造冲突的可能，那么Git可以使用更为安全的SHA-256或者SHA-512的摘要算法。

可是Git中的各种对象：提交（commit）、文件内容（blob）、目录树（tree）等（还有Tag）对象对应的SHA1哈希值是如何生成的呢？下面就来展示一下。

提交的SHA1哈希值生成方法。

- 看看HEAD对应的提交的内容。使用：`command: `git cat-file`` 命令。

```
$ git cat-file commit HEAD  
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9  
parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6  
author Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800  
committer Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800  
  
which version checked in?
```

- 提交信息中总共包含234个字符。

```
$ git cat-file commit HEAD | wc -c  
234
```

- 在提交信息的前面加上内容`commit 234<null>`（<null>为空字符），然后执行SHA1哈希算法。

```
$ ( printf "commit 234\000"; git cat-file commit HEAD ) | sha1sum  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86 -
```

- 上面命令得到的哈希值和用`:command:`git rev-parse``看到的是一样的。

```
$ git rev-parse HEAD  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

下面看一看文件内容的SHA1哈希值生成方法。

- 看看版本库中`:file:`welcome.txt``的内容。使用`:command:`git cat-file``命令。

```
$ git cat-file blob HEAD:welcome.txt  
Hello.  
Nice to meet you.
```

- 文件总共包含25字节的内容。

```
$ git cat-file blob HEAD:welcome.txt | wc -c  
25
```

- 在文件内容的前面加上`blob 25<null>`的内容，然后执行SHA1哈希算法。

```
$ ( printf "blob 25\000"; git cat-file blob HEAD:welcome.txt ) | sha1sum  
fd3c069c1de4f4bc9b15940f490aeb48852f3c42 -
```

- 上面命令得到的哈希值和用`:command:`git rev-parse``看到的是一样的。

```
$ git rev-parse HEAD:welcome.txt  
fd3c069c1de4f4bc9b15940f490aeb48852f3c42
```

最后再来看看树的SHA1哈希值的形成方法。

- HEAD对应的树的内容共包含39个字节。

```
$ git cat-file tree HEAD^{tree} | wc -c  
39
```

- 在树的内容的前面加上`tree 39<null>`的内容，然后执行SHA1哈希算法。

```
$ ( printf "tree 39\000"; git cat-file tree HEAD^{tree} ) | sha1sum  
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9 -
```

- 上面命令得到的哈希值和用`:command:`git rev-parse``看到的是相同的。

```
$ git rev-parse HEAD^{tree}  
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
```

在后面学习里程碑（Tag）的时候，会看到Tag对象（轻量级Tag除外）也是采用类似方法在对象库中存储的。

## 问题：为什么不用顺序的数字来表示提交？

到目前为止所进行的提交都是顺序提交，这可能让读者产生这么一个想法，为什么Git的提交不依据提交顺序对提交进行编号呢？可以把第一次提交定义为提交1，依次递增。尤其是对于拥有像Subversion等集中式版本控制系统使用经验的用户更会有这样的体会和想法。

集中式版本控制系统因为只有一个集中式的版本库，可以很容易的实现依次递增的全局唯一的提交号，像Subversion就是如此。Git作为分布式版本控制系统，开发可以是非线性的，每个人可以通过克隆版本库的方式工作在不同的本地版本库当中，在本地做的提交可以通过版本库之间的交互（推送/push和拉回/pull操作）而互相分发，如果提交采用本地唯一的数字编号，在提交分发的时候不可避免的造成冲突。这就要求提交的编号不能仅仅是本地局部有效，而是要“全球唯一”。Git的提交通过SHA1哈希值作为提交ID，的确做到了“全球唯一”。

Mercurial(Hg)是另外一个著名的分布式版本控制系统，它的提交ID非常有趣：同时使用了顺序的数字编号和“全球唯一”的SHA1哈希值。但实际上顺序的数字编号只是本地有效，对于克隆版本库来说没有意义，只有SHA1哈希值才是通用的编号。

```
$ hg log --limit 2  
修改集: 3009:2f1a3a7e8eb0  
标签: tip  
用户: Daniel Neuhäuser <dasdasich@gmail.com>  
日期: Wed Dec 01 23:13:31 2010 +0100  
摘要: "Fixed" the CombinedHTMLDiff test  
  
修改集: 3008:2fd3302ca7e5  
用户: Daniel Neuhäuser <dasdasich@gmail.com>  
日期: Wed Dec 01 22:54:54 2010 +0100  
摘要: #559 Add `html_permalink_text` confval
```

Hg的设计使得本地使用版本库更为方便，但是要在Git中做类似实现却很难，这是因为Git

相比Hg拥有真正的分支管理功能。在Git中会存在当前分支中看不到的其他分支的提交，如何进行提交编号的管理十分的复杂。

幸好Git提供很多方法可以方便的访问Git库中的对象。

- 采用部分的SHA1哈希值。不必写全40位的哈希值，只采用开头的部分，不和现有其他的冲突即可。
- 使用master代表分支master中最新的提交，使用全称refs/heads/master亦可。
- 使用HEAD代表版本库中最近的一次提交。
- 符号`^`可以用于指代父提交。例如：
  - HEAD^代表版本库中上一次提交，即最近一次提交的父提交。
  - HEAD^^则代表HEAD^的父提交。
- 对于一个提交有多个父提交，可以在符号^后面用数字表示是第几个父提交。例如：
  - a573106^2含义是提交a573106的多个父提交中的第二个父提交。
  - HEAD^1相当于HEAD^含义是HEAD多个父提交中的第一个。
  - HEAD^^2含义是HEAD^ (HEAD父提交) 的多个父提交中的第二个。
- 符号~<n>也可以用于指代祖先提交。下面两个表达式效果等同：

```
a573106~5  
a573106^^^^^
```

- 提交所对应的树对象，可以用类似如下的语法访问。

```
a573106^{tree}
```

- 某一提交对应的文件对象，可以用如下的语法访问。

```
a573106:path/to/file
```

- 暂存区中的文件对象，可以用如下的语法访问。

```
:path/to/file
```

读者可以使用:`command: `git rev-parse``命令在本地版本库中练习一下：

```
$ git rev-parse HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git cat-file -p e695
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcc1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1291022581 +0800

which version checked in?
$ git cat-file -p e695^
tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
parent 9e8a761ff9dd343a1380032884f488a2422c495a
author Jiang Xin <jiangxin@osssxp.com> 1290999606 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1290999606 +0800

who does commit?
$ git rev-parse e695^{tree}
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
$ git rev-parse e695^{^{tree}}
190d840dd3d8fa319bdec6b8112b0957be7ee769
```

在后面的介绍中，还会了解更多访问Git对象的技巧。例如使用tag和日期访问版本库对象。

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/030-head-master-commit-refs.rst>

# Git重置

在上一章了解了版本库中对象的存储方式以及分支master的实现。即master分支在版本库的引用目录（.git/refs）中体现为一个引用文件：file:`.git/refs/heads/master`，其内容就是分支中最新提交的提交ID。

```
$ cat .git/refs/heads/master  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

上一章还通过对提交本身数据结构的分析，看到提交可以通过到父提交的关联实现对提交历史的追溯。注意：下面的：command:`git log`命令中使用了--oneline参数，类似于--pretty=oneline，但是可以显示更短小的提交ID。参数--oneline在Git 1.6.3 及以后版本提供，老版本的Git可以使用参数--pretty=oneline --abbrev-commit替代。

```
$ git log --graph --oneline  
* e695606 which version checked in?  
* a0c641e who does commit?  
* 9e8a761 initialized.
```

那么是不是有新的提交发生的时候，代表master分支的引用文件的内容会改变呢？代表master分支的引用文件的内容可以人为的改变么？本章就来探讨用：command:`git reset`命令改变分支引用文件内容，即实现分支的重置。

## 分支游标master的探秘

先来看看当有新的提交发生的时候，文件：file:`.git/refs/heads/master`的内容如何改变。首先在工作区创建一个新文件，姑且叫做：file:`new-commit.txt`，然后提交到版本库中。

```
$ touch new-commit.txt  
$ git add new-commit.txt  
$ git commit -m "does master follow this new commit?"  
[master 4902dc3] does master follow this new commit?  
 0 files changed, 0 insertions(+), 0 deletions(-)  
  create mode 100644 new-commit.txt
```

此时工作目录下会有两个文件，其中文件：file:`new-commit.txt`是新增的。

```
$ ls
```

```
new-commit.txt  welcome.txt
```

来看看master分支指向的提交ID是否改变了。

- 先看看在版本库引用空间（.git/refs/目录）下的master文件内容的确更改了，指向了新的提交。

```
$ cat .git/refs/heads/master  
4902dc375672fbf52a226e0354100b75d4fe31e3
```

- 再用`:command:`git log``查看一下提交日志，可以看到刚刚完成的提交。

```
$ git log --graph --oneline  
* 4902dc3 does master follow this new commit?  
* e695606 which version checked in?  
* a0c641e who does commit?  
* 9e8a761 initialized.
```

引用`refs/heads/master`就好像是一个游标，在有新的提交发生的时候指向了新的提交。可是如果只可上、不可下，就不能称为“游标”。Git提供了`:command:`git reset``命令，可以将“游标”指向任意一个存在的提交ID。下面的示例就尝试人为的更改游标。（注意下面的命令中使用了`--hard`参数，会破坏工作区未提交的改动，慎用。）

```
$ git reset --hard HEAD^  
HEAD is now at e695606 which version checked in?
```

还记得上一章介绍的`HEAD^`代表了`HEAD`的父提交么？所以这条命令就相当于将`master`重置到上一个老的提交上。来看一下`master`文件的内容是否更改了。

```
$ cat .git/refs/heads/master  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

果然`master`分支的引用文件的指向更改为前一次提交的ID了。而且通过下面的命令可以看出新添加的文件`:file:`new-commit.txt``也丢失了。

```
$ ls  
welcome.txt
```

重置命令不仅仅可以重置到前一次提交，重置命令可以直接使用提交ID重置到任何一次提交。

- 通过:command:`git log`查询到最早的提交ID。

```
$ git log --graph --oneline
* e695606 which version checked in?
* a0c641e who does commit?
* 9e8a761 initialized.
```

- 然后重置到最早的一次提交。

```
$ git reset --hard 9e8a761
HEAD is now at 9e8a761 initialized.
```

- 重置后会发现:file:`welcome.txt`也回退到原始版本库，曾经的修改都丢失了。

```
$ cat welcome.txt
Hello.
```

使用重置命令很危险，会彻底的丢弃历史。那么还能够通过浏览提交历史的办法找到丢弃的提交ID，再使用重置命令恢复历史么？不可能！因为重置让提交历史也改变了。

```
$ git log
commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Sun Nov 28 12:48:26 2010 +0800

initialized.
```

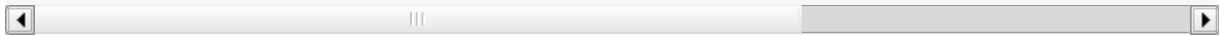
## 用reflog挽救错误的重置

如果没有记下重置前master分支指向的提交ID，想要重置回原来的提交真的是一件麻烦的事情（去对象库中一个一个地找）。幸好Git提供了一个挽救机制，通过:file:`.git/logs`目录下日志文件记录了分支的变更。默认非裸版本库（带有工作区）都提供分支日志功能，这是因为带有工作区的版本库都有如下设置：

```
$ git config core.logallrefupdates
true
```

查看一下master分支的日志文件:file:`.git/logs/refs/heads/master`中的内容。下面命令显示了该文件的最后几行。为了排版的需要，还将输出中的40位的SHA1提交ID缩短。

```
$ tail -5 .git/logs/refs/heads/master  
dca47ab a0c641e Jiang Xin <jiangxin@ossp.com> 1290999606 +0800 commit (am  
a0c641e e695606 Jiang Xin <jiangxin@ossp.com> 1291022581 +0800 commit: wh  
e695606 4902dc3 Jiang Xin <jiangxin@ossp.com> 1291435985 +0800 commit: do  
4902dc3 e695606 Jiang Xin <jiangxin@ossp.com> 1291436302 +0800 HEAD^: upd  
e695606 9e8a761 Jiang Xin <jiangxin@ossp.com> 1291436382 +0800 9e8a761: u
```



可以看出这个文件记录了master分支指向的变迁，最新的改变追加到文件的末尾因此最后出现。最后一行可以看出因为执行了:command:`git reset --hard`命令，指向的提交ID由e695606改变为9e8a761。

Git提供了一个:command:`git reflog`命令，对这个文件进行操作。使用show子命令可以显示此文件的内容。

```
$ git reflog show master | head -5  
9e8a761 master@{0}: 9e8a761: updating HEAD  
e695606 master@{1}: HEAD^: updating HEAD  
4902dc3 master@{2}: commit: does master follow this new commit?  
e695606 master@{3}: commit: which version checked in?  
a0c641e master@{4}: commit (amend): who does commit?
```

使用:command:`git reflog`的输出和直接查看日志文件最大的不同在于显示顺序的不同，即最新改变放在了最前面显示，而且只显示每次改变的最终的SHA1哈希值。还有个重要的区别在于使用:command:`git reflog`的输出中还提供一个方便易记的表达式: <refname>@{<n>}。这个表达式的含义是引用<refname>之前第<n>次改变时的SHA1哈希值。

那么将引用master切换到两次变更之前的值，可以使用下面的命令。

- 重置master为两次改变之前的值。

```
$ git reset --hard master@{2}  
HEAD is now at 4902dc3 does master follow this new commit?
```

- 重置后工作区中文件:file:`new-commit.txt`又回来了。

```
$ ls  
new-commit.txt  welcome.txt
```

- 提交历史也回来了。

```
$ git log --oneline  
4902dc3 does master follow this new commit?  
e695606 which version checked in?  
a0c641e who does commit?  
9e8a761 initialized.
```

此时如果再用:command:`git reflog`查看，会看到恢复master的操作也记录在日志中了。

```
$ git reflog show master | head -5  
4902dc3 master@{0}: master@{2}: updating HEAD  
9e8a761 master@{1}: 9e8a761: updating HEAD  
e695606 master@{2}: HEAD^: updating HEAD  
4902dc3 master@{3}: commit: does master follow this new commit?  
e695606 master@{4}: commit: which version checked in?
```

## 深入了解:command:`git reset`命令

重置命令(:command:`git reset`)是Git最常用的命令之一，也是最危险，最容易误用的命令。来看看:command:`git reset`命令的用法。

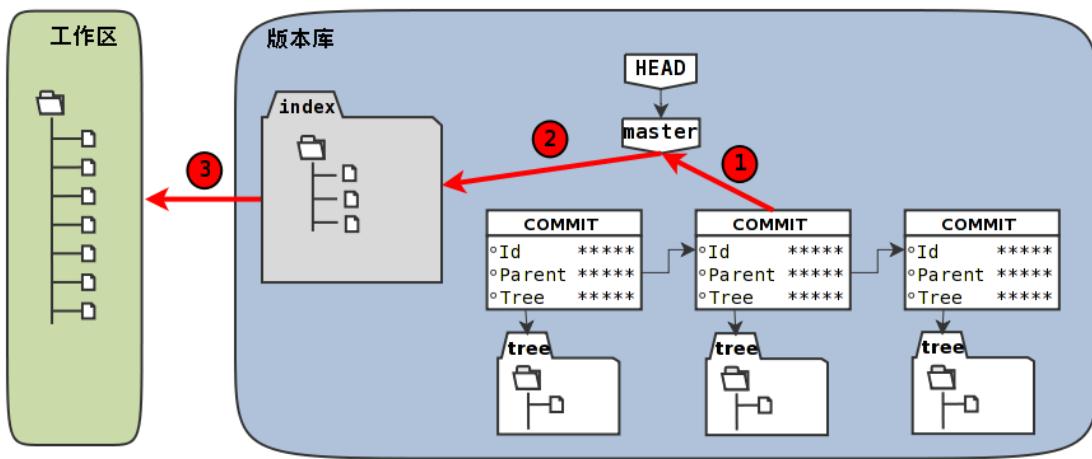
用法一： `git reset [-q] [<commit>] [--] <paths>...`  
用法二： `git reset [--soft | --mixed | --hard | --merge | --keep] [-q] [<commit>]`

上面列出了两个用法，其中<commit>都是可选项，可以使用引用或者提交ID，如果省略<commit>则相当于使用了HEAD的指向作为提交ID。

上面列出的两种用法的区别在于，第一种用法在命令中包含路径:file:`<paths>`。为了避免路径和引用（或者提交ID）同名而冲突，可以在:file:`<paths>`前用两个连续的短线（减号）作为分隔。

第一种用法（包含了路径:file:`<paths>`的用法）不会重置引用，更不会改变工作区，而是用指定提交状态(<commit>)下的文件(<paths>)替换掉暂存区中的文件。例如命令:command:`git reset HEAD <paths>`相当于取消之前执行的:command:`git add <paths>`命令时改变的暂存区。

第二种用法（不使用路径:file:`<paths>`的用法）则会重置引用。根据不同的选项，可以对暂存区或者工作区进行重置。参照下面的版本库模型图，来看一看不同的参数对第二种重置语法的影响。



命令格式: `git reset [--soft | --mixed | --hard] [<commit>]`

- 使用参数`--hard`, 如: `:command:`git reset --hard <commit>``。

会执行上图中的1、2、3全部的三个动作。即:

- 替换引用的指向。引用指向新的提交ID。
- 替换暂存区。替换后，暂存区的内容和引用指向的目录树一致。
- 替换工作区。替换后，工作区的内容变得和暂存区一致，也和HEAD所指向的目录树内容相同。

- 使用参数`--soft`, 如:`:command:`git reset --soft <commit>``。

会执行上图中的操作1。即只更改引用的指向，不改变暂存区和工作区。

- 使用参数`--mixed`或者不使用参数（缺省即为`--mixed`），如:`:command:`git reset <commit>``。

会执行上图中的操作1和操作2。即更改引用的指向以及重置暂存区，但是不改变工作区。

下面通过一些示例，看一下重置命令的不同用法。

- 命令: `:command:`git reset``

仅用HEAD指向的目录树重置暂存区，工作区不会受到影响，相当于将之前用`:command:`git add``命令更新到暂存区的内容撤出暂存区。引用也未改变，因为引用重置到HEAD相当于没有重置。

- 命令: `:command:`git reset HEAD``

同上。

- 命令: `:command:`git reset -- filename``

仅将文件`:file:`filename``撤出暂存区，暂存区中其他文件不改变。相当于对命令`:command:`git add filename``的反向操作。

- 命令: `:command:`git reset HEAD filename``

同上。

- 命令: `:command:`git reset --soft HEAD^``

工作区和暂存区不改变，但是引用向前回退一次。当对最新提交的提交说明或者提交的更改不满意时，撤销最新的提交以便重新提交。

在之前曾经介绍过一个修补提交命令`:command:`git commit --amend``，用于对最新的提交进行重新提交以修补错误的提交说明或者错误的提交文件。修补提交命令实际上相当于执行了下面两条命令。（注：文件`:file:`.git/COMMIT_EDITMSG``保存了上次的提交日志）

```
$ git reset --soft HEAD^
$ git commit -e -F .git/COMMIT_EDITMSG
```

- 命令: `:command:`git reset HEAD^``

工作区不改变，但是暂存区会回退到上一次提交之前，引用也会回退一次。

- 命令: `:command:`git reset --mixed HEAD^``

同上。

- 命令: `:command:`git reset --hard HEAD^``

彻底撤销最近的提交。引用回退到前一次，而且工作区和暂存区都会回退到上一次提交的状态。自上一次以来的提交全部丢失。

来源: <https://github.com/gotgit/gotgit/blob/master/02-git-solo/040-git-reset.rst>

# 恢复进度

在之前“Git暂存区”一章的结尾，曾经以终结者（The Terminator）的口吻说：“我会再回来”，会继续对暂存区的探索。经过了前面三章对Git对象、重置命令、检出命令的探索，现在已经拥有了足够多的武器，是时候“回归”了。

本章“回归”之后，再看Git状态输出中关于`:command:`git reset``或者`:command:`git checkout``的指示，有了前面几章的基础已经会觉得很亲切和易如反掌了。本章还会重点介绍“回归”使用的`:command:`git stash``命令。

## 继续暂存区未完成的实践

经过了前面的实践，现在DEMO版本库应该处于master分支上，看看是不是这样。

```
$ cd /path/to/my/workspace/demo
$ git status -sb          # Git 1.7.2 及以上版本才支持 -b 参数哦
## master
$ git log --graph --pretty=oneline --stat
*   2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
| \
| * acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode.
| | 0 files changed, 0 insertions(+), 0 deletions(-)
* | 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new comm
|/
|   0 files changed, 0 insertions(+), 0 deletions(-)
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
| welcome.txt | 1 +
| 1 files changed, 1 insertions(+), 0 deletions(-)
* a0c641e92b10d8bcc1ed1bf84ca80340fdfdefee6 who does commit?
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
  welcome.txt | 1 +
  1 files changed, 1 insertions(+), 0 deletions(-)
```

还记得在之前“Git暂存区”一章的结尾，是如何保存进度的么？翻回去看一下，用的是`:command:`git stash``命令。这个命令用于保存当前进度，也是恢复进度要用的命令。

查看保存的进度用命令`:command:`git stash list``。

```
$ git stash list
stash@{0}: WIP on master: e695606 which version checked in?
```

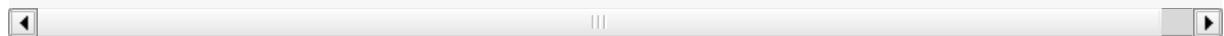
现在就来恢复进度。使用:[command: `git stash pop`](#)从最近保存的进度进行恢复。

```
$ git stash pop
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:    welcome.txt
#
Dropped refs/stash@{0} (c1bd56e2565abd64a0d63450fe42aba23b673cf3)
```



先不要管:[command: `git stash pop`](#)命令的输出，后面会专题介绍:[command: `git stash`](#)命令。通过查看工作区的状态，可以发现进度已经找回了（状态和进度保存前稍有不同）。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:    welcome.txt
#
```

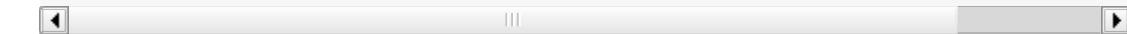


此时再看Git状态输出，是否别有一番感觉呢？有了前面三章的基础，现在可以游刃有余的应对各种情况了。

- 以当前暂存区状态进行提交，即只提交:[file: `a/b/c/hello.txt`](#)，不提交:[file: `welcome.txt`](#)。
  - 执行提交：

```
$ git commit -m "add new file: a/b/c/hello.txt, but leave welcome.txt"
```

```
[master 6610d05] add new file: a/b/c/hello.txt, but leave welcome.txt  
1 files changed, 2 insertions(+), 0 deletions(-)  
create mode 100644 a/b/c/hello.txt
```



- 查看提交后的状态:

```
$ git status -s  
M welcome.txt
```

- 反悔了，回到之前的状态。

- 用重置命令放弃最新的提交:

```
$ git reset --soft HEAD^
```

- 查看最新的提交日志，可以看到前面的提交被抛弃了。

```
$ git log -1 --pretty=oneline  
2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
```

- 工作区和暂存区的状态也都维持原来的状态。

```
$ git status -s  
A a/b/c/hello.txt  
M welcome.txt
```

- 想将:`:file:`welcome.txt``提交。

再简单不过了。

```
$ git add welcome.txt  
$ git status -s  
A a/b/c/hello.txt  
M welcome.txt
```

- 想将:`:file:`a/b/c/hello.txt``撤出暂存区。

也是用重置命令。

```
$ git reset HEAD a/b/c  
$ git status -s
```

```
M welcome.txt  
?? a/
```

- 想将剩下的文件（:file:`welcome.txt`）从暂存区撤出，就是说不想提交任何东西了。

还是使用重置命令，甚至可以不使用任何参数。

```
$ git reset  
Unstaged changes after reset:  
M      welcome.txt
```

- 想将本地工作区所有的修改清除。即清除:file:`welcome.txt`的改动，删除添加的目录:file:`a`即下面的子目录和文件。

- 清除:file:`welcome.txt`的改动用检出命令。

实际对于此例执行:command:`git checkout .`也可以。

```
$ git checkout -- welcome.txt
```

- 工作区显示还有一个多余的目录:file:`a`。

```
$ git status  
# On branch master  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       a/
```

- 删除本地多余的目录和文件，可以使用:command:`git clean`命令。先来测试运行以便看看哪些文件和目录会被删除，以免造成误删。

```
$ git clean -nd  
Would remove a/
```

- 真正开始强制删除多余的目录和文件。

```
$ git clean -fd  
Removing a/
```

- 整个世界清净了。

```
$ git status -s
```

## 使用:command:`git stash`

命令:command:`git stash`可以用于保存和恢复工作进度，掌握这个命令对于日常的工作会有很大的帮助。关于这个命令的最主要的用法实际上通过前面的演示已经了解了。

- 命令: :command:`git stash`

保存当前工作进度。会分别对暂存区和工作区的状态进行保存。

- 命令: :command:`git stash list`

显示进度列表。此命令显然暗示了:command:`git stash`可以多次保存工作进度，并且在恢复的时候进行选择。

- 命令: :command:`git stash pop [--index] [<stash>]`

如果不使用任何参数，会恢复最新保存的工作进度，并将恢复的工作进度从存储的工作进度列表中清除。

如果提供`<stash>`参数（来自于:command:`git stash list`显示的列表），则从该`<stash>`中恢复。恢复完毕也将从进度列表中删除`<stash>`。

选项`--index`除了恢复工作区的文件外，还尝试恢复暂存区。这也就是为什么在本章一开始恢复进度的时候显示的状态和保存进度前略有不同。

实际上还有几个用法也很有用。

- 命令: :command:`git stash [save [--patch] [-k|--[no-]keep-index] [-q|--quiet] [<message>]]`

- 这条命令实际上是第一条:command:`git stash`命令的完整版。即如果需要在保存工作进度的时候使用指定的说明，必须使用如下格式：

```
git stash save "message..."
```

- 使用参数`--patch`会显示工作区和HEAD的差异，通过对差异文件的编辑决定在进度中最终要保存的工作区的内容，通过编辑差异文件可以在进度中排除无关内容。
- 使用`-k`或者`--keep-index`参数，在保存进度后不会将暂存区重置。缺省会将暂存

区和工作区强制重置。

- 命令: :command:`git stash apply [--index] [<stash>]`

除了不删除恢复的进度之外，其余和:command:`git stash pop`命令一样。

- 命令: :command:`git stash drop [<stash>]`

删除一个存储的进度。缺省删除最新的进度。

- 命令: :command:`git stash clear`

删除所有存储的进度。

- 命令: :command:`git stash branch <branchname> [<stash>]`

基于进度创建分支。对了，还没有讲到分支呢。;)

## 探秘:command:`git stash`

了解一下:command:`git stash`的机理会有几个好处：当保存了多个进度的时候知道从哪个进度恢复；综合运用前面介绍的Git知识点；了解Git的源码，Git将不再神秘。

在执行:command:`git stash`命令时，Git实际调用了一个脚本文件实现相关功能，这个脚本的文件名就是:file:`git-stash`。看看:file:`git-stash`安装在哪里了。

```
$ git --exec-path  
/usr/lib/git-core
```

如果检查一下这个目录，会震惊的。

```
$ ls /usr/lib/git-core/  
git                  git-help          git-reflog  
git-add              git-http-backend  git-relink  
git-add-interactive  git-http-fetch   git-remote  
git-am               git-http-push    git-remote-ftp  
git-annotate         git-imap-send   git-remote-ftps  
git-apply             git-index-pack  git-remote-http  
.....  
... 省略40余行 ...  
.....
```

实际上在1.5.4之前的版本，Git会安装这些一百多个以:command:`git-<cmd>`格式命名的程序到可执行路径中。这样做的唯一好处就是不用借助任何扩展机制就可以实现命令行补

齐：即键入git-后，连续两次键入<Tab>键，就可以把这一百多个命令显示出来。这种方式随着Git子命令的增加越来越显得混乱，因此在1.5.4版本开始，不再提供:command:`git-<cmd>`格式的命令，而是用唯一的:command:`git`命令。而之前的名为:command:`git-<cmd>`的子命令则保存在非可执行目录下，由Git负责加载。

在后面的章节中偶尔会看到形如:command:`git-<cmd>`字样的名称，以及同时存在的:command:`git <cmd>`命令。可以这样理解：:command:`git-<cmd>`作为软件本身的名字，而其命令行为:command:`git <cmd>`。

最早很多Git命令都是用Shell或者Perl脚本语言开发的，在Git的发展中一些对运行效率要求高的命令用C语言改写。而:file:`git-stash`（至少在Git 1.7.3.2版本）还是使用Shell脚本开发的，研究它会比研究用C写的命令要简单的多。

```
$ file /usr/lib/git-core/git-stash  
/usr/lib/git-core/git-stash: POSIX shell script text executable
```

解析:file:`git-stash`脚本会比较枯燥，还是通过运行一些示例更好一些。

当前的进度保存列表是空的。

```
$ git stash list
```

下面在工作区中做一些改动。

```
$ echo Bye-Bye. >> welcome.txt  
$ echo hello. > hack-1.txt  
$ git add hack-1.txt  
$ git status -s  
A  hack-1.txt  
M welcome.txt
```

可见暂存区中已经添加了新增的:file:`hack-1.txt`，修改过的:file:`welcome.txt`并未添加到暂存区。执行:command:`git stash`保存一下工作进度。

```
$ git stash save "hack-1: hacked welcome.txt, newfile hack-1.txt"  
Saved working directory and index state On master: hack-1: hacked welcome.txt  
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

再来看工作区恢复了修改前的原貌（实际上用了git reset --hard HEAD命令），文件:file:`welcome.txt`的修改不见了，文件:file:`hack-1.txt`整个都不见了。

```
$ git status -s
$ ls
detached-commit.txt  new-commit.txt  welcome.txt
```

再做一个修改，并尝试保存进度。

```
$ echo fix. > hack-2.txt
$ git stash
No local changes to save
```

进度保存失败！可见本地没有被版本控制系统跟踪的文件并不能保存进度。因此本地新文件需要执行添加再执行：[command: `git stash`](#) 命令。

```
$ git add hack-2.txt
$ git stash
Saved working directory and index state WIP on master: 2b31c19 Merge commit '
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

不用看就知道工作区再次恢复原状。如果这时执行：[command: `git stash list`](#) 会看到有两次进度保存。

```
$ git stash list
stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'
stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1.txt
```

从上面的输出可以得出两个结论：

- 在用：[command: `git stash`](#) 命令保存进度时，提供说明更容易找到对应的进度文件。
- 每个进度的标识都是`stash@{<n>}`格式，像极了前面介绍的reflog的格式。

实际上，[:command: `git stash`](#) 的就是用到了前面介绍的引用和引用变更日志（reflog）来实现的。

```
$ ls -l .git/refs/stash .git/logs/refs/stash
-rw-r--r-- 1 jiangxin jiangxin 364 Dec  6 16:11 .git/logs/refs/stash
-rw-r--r-- 1 jiangxin jiangxin  41 Dec  6 16:11 .git/refs/stash
```

那么在“Git重置”一章中学习的reflog可以派上用场了。

```
$ git reflog show refs/stash
e5c0cdc refs/stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'
```

```
6cec9db refs/stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1
```

对照`:command:`git reflog``的结果和前面`:command:`git stash list``的结果，可以肯定用`:command:`git stash``保存进度，实际上会将进度保存在引用`refs/stash`所指向的提交中。多次的进度保存，实际上相当于引用`refs/stash`一次又一次的变化，而`refs/stash`引用的变化由`reflog`（即`:command:`.git/logs/refs/stash``）所记录下来。这个实现是多么的简单而巧妙啊。

新的一个疑问又出现了，如何在引用`refs/stash`中同时保存暂存区的进度和工作区中的进度呢？查看一下引用`refs/stash`的提交历史能够看出端倪。

```
$ git log --graph --pretty=raw refs/stash -2
*   commit e5c0cdc2dedc3e50e6b72a683d928e19a1d9de48
| \
|   tree 780c22449b7ff67e2820e09a6332c360ddc80578
|   | parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
|   | parent c5edbdc90addb06577ff60f644acd1542369194
|   | author Jiang Xin <jiangxin@ossp.com> 1291623066 +0800
|   | committer Jiang Xin <jiangxin@ossp.com> 1291623066 +0800
|
|   | WIP on master: 2b31c19 Merge commit 'acc2f69'
|
| * commit c5edbdc90addb06577ff60f644acd1542369194
| / tree 780c22449b7ff67e2820e09a6332c360ddc80578
|   parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
|   author Jiang Xin <jiangxin@ossp.com> 1291623066 +0800
|   committer Jiang Xin <jiangxin@ossp.com> 1291623066 +0800
|
|   index on master: 2b31c19 Merge commit 'acc2f69'
```

可以看到在提交关系图可以看到进度保存的最新提交是一个合并提交。最新的提交说明中有`WIP`字样（是`Work In Progress`的简称），说明代表了工作区进度。而最新提交的第二个父提交（上图中显示为第二个提交）有`index on master`字样，说明这个提交代表着暂存区的进度。

但是上图中的两个提交都指向了同一个树——`tree ``780c22449b7ff67e2820e09a6332c360ddc80578```，这是因为最后一次做进度保存时工作区相对暂存区没有改变，这让关于工作区和暂存区在引用`refs/stash`中的存储变得有些扑朔迷离。别忘了第一次进度保存工作区、暂存区和版本库都是不同的，可以用于验证关于`refs/stash`实现机制的判断。

第一次进度保存可以用`reflog`中的语法，即用`refs/stash@{1}`来访问，也可以用简称`stash@{1}`。下面就用第一次的进度保存来研究一下。

```
$ git log --graph --pretty=raw stash@{1} -3
* commit 6cec9db44af38d01abe7b5025a5190c56fd0cf49
| \
| tree 7250f186c6aa3e2d1456d7fa915e529601f21d71
| | parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
| | parent 4560d76c19112868a6a5692bf9379de09c0452b7
| | author Jiang Xin <jiangxin@osssxp.com> 1291622767 +0800
| | committer Jiang Xin <jiangxin@osssxp.com> 1291622767 +0800
| |
| |     On master: hack-1: hacked welcome.txt, newfile hack-1.txt
| |
| * commit 4560d76c19112868a6a5692bf9379de09c0452b7
| / tree 5d4dd328187e119448c9171f99cf2e507e91a6c6
| | parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
| | author Jiang Xin <jiangxin@osssxp.com> 1291622767 +0800
| | committer Jiang Xin <jiangxin@osssxp.com> 1291622767 +0800
| |
|     index on master: 2b31c19 Merge commit 'acc2f69'
|
* commit 2b31c199d5b81099d2ecd91619027ab63e8974ef
| \
| tree ab676f92936000457b01507e04f4058e855d4df0
| | parent 4902dc375672fbf52a226e0354100b75d4fe31e3
| | parent acc2f69cf6f0ae346732382c819080df75bb2191
| | author Jiang Xin <jiangxin@osssxp.com> 1291535485 +0800
| | committer Jiang Xin <jiangxin@osssxp.com> 1291535485 +0800
| |
| |     Merge commit 'acc2f69'
```

果然上面显示的三个提交对应的三棵树各不相同。查看一下差异。用“原基线”代表进度保存时版本库的状态，即提交`2b31c199`；用“原暂存区”代表进度保存时暂存区的状态，即提交`4560d76`；用“原工作区”代表进度保存时工作区的状态，即提交`6cec9db`。

- 原基线和原暂存区的差异比较。

```
$ git diff stash@{1}^2^ stash@{1}^2
diff --git a/hack-1.txt b/hack-1.txt
new file mode 100644
index 000000..25735f5
--- /dev/null
+++ b/hack-1.txt
@@ -0,0 +1 @@
+hello.
```

- 原暂存区和原工作区的差异比较。

```
$ git diff stash@{1}^2 stash@{1}
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
```

```
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
Hello.
Nice to meet you.
+Bye-Bye.
```

- 原基线和原工作区的差异比较。

```
$ git diff stash@{1}^1 stash@{1}
diff --git a/hack-1.txt b/hack-1.txt
new file mode 100644
index 0000000..25735f5
--- /dev/null
+++ b/hack-1.txt
@@ -0,0 +1 @@
+hello.
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
Hello.
Nice to meet you.
+Bye-Bye.
```

从stash@{1}来恢复进度。

```
$ git stash apply stash@{1}
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hack-1.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
```

显示进度列表，然后删除进度列表。

```
$ git stash list
```

```
stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'  
stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1.txt  
$ git stash clear
```

删除进度列表之后，会发现stash相关的引用和reflog也都不见了。

```
$ ls -l .git/refs/stash .git/logs/refs/stash  
ls: cannot access .git/refs/stash: No such file or directory  
ls: cannot access .git/logs/refs/stash: No such file or directory
```

通过上面的这些分析，有一定Shell编程基础的读者就可以尝试研究git-stash的代码了，可能会有新的发现。

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/060-git-stage-n-stash.rst>

# Git基本操作

之前的实践选取的示例都非常简单，基本上都是增加和修改文本文件，而现实情况要复杂的多，需要应对各种情况：文件删除，文件复制，文件移动，目录的组织，二进制文件，误删文件的恢复等等。

本章要用一个更为真实的例子：通过对Hello World程序源代码的版本控制，来介绍工作区中其他的一些常用操作。首先会删除之前历次实践在版本库中留下的“垃圾”数据，然后再在其中创建一些真实的代码，并对其进行版本控制。

## 先来合个影

马上就要和之前实践遗留的数据告别了，告别之前是不是要留个影呢？在Git里，“留影”用的命令叫做`:command:`tag``，更加专业的术语叫做“里程碑”（打tag，或打标签）。

```
$ cd /path/to/my/workspace/demo  
$ git tag -m "Say bye-bye to all previous practice." old_practice
```

在本章还不打算详细介绍里程碑的奥秘，只要知道里程碑无非也是一个引用，通过记录提交ID（或者创建Tag对象）来为当前版本库状态进行“留影”。

```
$ ls .git/refs/tags/old_practice  
.git/refs/tags/old_practice  
  
$ git rev-parse refs/tags/old_practice  
41bd4e2cce0f8baa9bb4cdda62927b408c846cd6
```

留过影之后，可以执行`:command:`git describe``命令显示当前版本库的最新提交的版本号。显示的时候会选取离该提交最近的里程碑作为“基础版本号”，后面附加标识距离“基础版本”的数字以及该提交的SHA1哈希值缩写。因为最新的提交上恰好被打了一个“里程碑”，所以用“里程碑”的名字显示为版本号。这个技术在后面的示例代码中被使用。

```
$ git describe  
old_practice
```

## 删除文件

看看版本库当前的状态，暂存区和工作区都包含修改。

```
$ git status -s
A  hack-1.txt
M welcome.txt
```

在这个暂存区和工作区都包含文件修改的情况下，使用删除命令更具有挑战性。删除命令有多种使用方法，有的方法很巧妙，而有的方法需要更多的输入。为了分别介绍不同的删除方法，还要使用上一章介绍的进度保存（:command:`git-stash`）命令。

- 保存进度。

```
$ git stash
Saved working directory and index state WIP on master: 2b31c19 Merge comm
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

- 再恢复进度。注意不要使用:command:`git stash pop`，而是使用:command:`git stash apply`，因为这个保存的进度要被多次用到。

```
$ git stash apply
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hack-1.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
#
#       modified:   welcome.txt
#
```

## 本地删除不是真的删除

当前工作区的文件有：

```
$ ls
detached-commit.txt
hack-1.txt
new-commit.txt
```

```
welcome.txt
```

直接在工作区删除这些文件，会如何呢？

```
$ rm *.txt
```

通过下面的命令，可以看到在暂存区（版本库）中文件仍在，并未删除。

```
$ git ls-files  
detached-commit.txt  
hack-1.txt  
new-commit.txt  
welcome.txt
```

通过文件的状态来看，文件只是在本地进行了删除，尚未加到暂存区（提交任务）中。也就是说：直接在工作区删除，对暂存区和版本库没有任何影响。

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       new file:   hack-1.txt  
#  
# Changed but not updated:  
#   (use "git add/rm <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working directory)  
#  
#       deleted:    detached-commit.txt  
#       deleted:    hack-1.txt  
#       deleted:    new-commit.txt  
#       deleted:    welcome.txt  
#
```

从Git状态输出可以看出，本地删除如果要反映在暂存区中应该用：[command: `git rm`](#) 命令，对于不想删除的文件执行：[command: `git checkout -- <file>`](#) 可以让文件在工作区重现。

## 执行：[command: `git rm`](#) 命令删除文件

好吧，按照上面状态输出的内容，将所有的文本文件删除。执行下面的命令。

```
$ git rm detached-commit.txt hack-1.txt new-commit.txt welcome.txt
rm 'detached-commit.txt'
rm 'hack-1.txt'
rm 'new-commit.txt'
rm 'welcome.txt'
```

再看一看状态：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    detached-commit.txt
#       deleted:    new-commit.txt
#       deleted:    welcome.txt
#
```

删除动作加入了暂存区。这时执行提交动作，就真正意义上执行了文件删除。

```
$ git commit -m "delete trash files. (using: git rm)"
[master 483493a] delete trash files. (using: git rm)
1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 detached-commit.txt
 delete mode 100644 new-commit.txt
 delete mode 100644 welcome.txt
```

不过不要担心，文件只是在版本库最新提交中删除了，在历史提交中尚在。可以通过下面命令查看历史版本的文件列表。

```
$ git ls-files --with-tree=HEAD^
detached-commit.txt
new-commit.txt
welcome.txt
```

也可以查看在历史版本中尚在的删除文件的内容。

```
$ git cat-file -p HEAD^:welcome.txt
Hello.
Nice to meet you.
```

命令：[command:](#) `git add -u` 快速标记删除

在前面执行:command:`git rm`命令时，一一写下了所有要删除的文件名，好长的命令啊！能不能简化些？实际上:command:`git add`可以，即使用-u参数调用:command:`git add`命令，含义是将本地有改动（包括添加和删除）的文件标记为删除。为了重现刚才的场景，先使用重置命令抛弃最新的提交，再使用进度恢复到之前的状态。

- 丢弃之前测试删除的试验性提交。

```
$ git reset --hard HEAD^  
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

- 恢复保存的进度。（参数-q使得命令进入安静模式）

```
$ git stash apply -q
```

然后删除本地文件，状态依然显示只在本地删除了文件，暂存区文件仍在。

```
$ rm *.txt  
$ git status -s  
D detached-commit.txt  
AD hack-1.txt  
D new-commit.txt  
D welcome.txt
```

执行:command:`git add -u`命令可以将（被版本库追踪的）本地文件的变更（修改、删除）全部记录到暂存区中。

```
$ git add -u
```

查看状态，可以看到工作区删除的文件全部被标记为下次提交时删除。

```
$ git status -s  
D detached-commit.txt  
D new-commit.txt  
D welcome.txt
```

执行提交，删除文件。

```
$ git commit -m "delete trash files. (using: git add -u)"  
[master 7161977] delete trash files. (using: git add -u)  
1 files changed, 0 insertions(+), 2 deletions(-)  
delete mode 100644 detached-commit.txt
```

```
delete mode 100644 new-commit.txt  
delete mode 100644 welcome.txt
```

## 恢复删除的文件

经过了上面的文件删除，工作区已经没有文件了。为了说明文件移动，现在恢复一个删除的文件。前面已经说过执行了文件删除并提交，只是在最新的提交中删除了文件，历史提交中文件仍然保留，可以从历史提交中提取文件。执行下面的命令可以从历史（前一次提交）中恢复:`:file:`welcome.txt``文件。

```
$ git cat-file -p HEAD~1:welcome.txt > welcome.txt
```

上面命令中出现的`HEAD~1`即相当于`HEAD^`都指的是`HEAD`的上一次提交。执行:`:command:`git add -A``命令会对工作区中所有改动以及新增文件添加到暂存区，也是一个常用的技巧。执行下面的命令后，将恢复过来的:`:file:`welcome.txt``文件添加回暂存区。

```
$ git add -A  
$ git status -s  
A  welcome.txt
```

执行提交操作，文件:`:file:`welcome.txt``又回来了。

```
$ git commit -m "restore file: welcome.txt"  
[master 63992f0] restore file: welcome.txt  
1 files changed, 2 insertions(+), 0 deletions(-)  
create mode 100644 welcome.txt
```

通过再次添加的方式恢复被删除的文件是最自然的恢复的方法。其他版本控制系统如CVS也采用同样的方法恢复删除的文件，但是有的版本控制系统如Subversion如果这样操作会有严重的副作用——文件变更历史被人为的割裂而且还会造成服务器存储空间的浪费。Git通过添加方式反删除文件没有副作用，这是因为在Git的版本库中相同内容的文件保存在一个blob对象中，而且即便是内容不同的blob对象在对象库打包整理过程中也会通过差异比较优化存储。

## 移动文件

通过将:`:file:`welcome.txt``改名为:`:file:`README``文件来测试一下在Git中如何移动文件。Git提供了:`:command:`git mv``命令完成改名操作。

```
$ git mv welcome.txt README
```

可以从当前的状态中看到改名的操作。

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       renamed:    welcome.txt -> README  
#
```

提交改名操作，在提交输出可以看到改名前后两个文件的相似度（百分比）。

```
$ git commit -m "改名测试"  
[master 7aa5ac1] 改名测试  
1 files changed, 0 insertions(+), 0 deletions(-)  
rename welcome.txt => README (100%)
```

可以不用:command:`git mv`命令实现改名

从提交日志中出现的文件相似度可以看出Git的改名实际上源自于Git对文件追踪的强大支持（文件内容作为blob对象保存在对象库中）。改名操作实际上相当于对旧文件执行删除，对新文件执行添加，即完全可以不使用:command:`git mv`操作，而是代之以:command:`git rm`和一个:command:`git add`操作。为了试验不使用:command:`git mv`命令是否可行，先撤销之前进行的提交。

- 撤销之前测试文件移动的提交。

```
$ git reset --hard HEAD^  
HEAD is now at 63992f0 restore file: welcome.txt
```

- 撤销之后:file:`welcome.txt`文件又回来了。

```
$ git status -s  
$ git ls-files  
welcome.txt
```

新的改名操作不使用:command:`git mv`命令，而是直接在本地改名（文件移动），将:file:`welcome.txt`改名为:file:`README`。

```
$ mv welcome.txt README
```

```
$ git status -s  
D welcome.txt  
?? README
```

为了考验一下Git的内容追踪能力，再修改一下改名后的 README 文件，即在文件末尾追加一行。

```
$ echo "Bye-Bye." >> README
```

可以使用前面介绍的:command:`git add -A`命令。相当于对修改文件执行:command:`git add`，对删除文件执行:command:`git rm`，而且对本地新增文件也执行:command:`git add`。

```
$ git add -A
```

查看状态，也可以看到文件重命名。

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       renamed:    welcome.txt -> README
```

执行提交。

```
$ git commit -m "README is from welcome.txt."  
[master c024f34] README is from welcome.txt.  
1 files changed, 1 insertions(+), 0 deletions(-)  
rename welcome.txt => README (73%)
```

这次提交中也看到了重命名操作，但是重命名相似度不是 100%，而是 73%。

## 一个显示版本号的Hello World

在本章一开始为纪念前面的实践留了一个影，叫做old\_practice。现在再次执行:command:`git describe`看一下现在的版本号。

```
$ git describe  
old_practice-3-gc024f34
```

就是说：当前工作区的版本是“留影”后的第三个版本，提交ID是c024f34。

下面的命令可以在提交日志中显示提交对应的里程碑（Tag）。其中参数--decorate可以在提交ID的旁边显示该提交关联的引用（里程碑或分支）。

```
$ git log --oneline --decorate -4
c024f34 (HEAD, master) README is from welcome.txt.
63992f0 restore file: welcome.txt
7161977 delete trash files. (using: git add -u)
2b31c19 (tag: old_practice) Merge commit 'acc2f69'
```

命令`:command:`git describe``的输出可以作为软件版本号，这个功能非常有用。因为这样可以很容易的实现将发布的软件包版本和版本库中的代码对应在一起，当发现软件包包含Bug时，可以最快、最准确的对应到代码上。

下面的Hello World程序就实现了这个功能。创建目录`:file:`src``，并在`:file:`src``目录下创建下面的三个文件：

- 文件：`:file:`src/main.c``

没错，下面的几行就是这个程序的主代码，和输出相关代码的就两行，一行显示“Hello, world.”，另外一行显示软件版本。在显示软件版本时用到了宏`_VERSION`，这个宏的来源参考下一个文件。

源代码：

```
#include "version.h"
#include <stdio.h>

int
main()
{
    printf( "Hello, world.\n" );
    printf( "version: %s.\n", _VERSION );
    return 0;
}
```

- 文件：`:file:`src/version.h.in``

没错，这个文件名的后缀是`:file:`.h.in``。这个文件其实是用于生成文件`:file:`version.h``的模板文件。在由此模板文件生成的`:file:`version.h``的过程中，宏`_VERSION`的值“`<version>`”会动态替换。

源代码:

```
#ifndef HELLO_WORLD_VERSION_H
#define HELLO_WORLD_VERSION_H

#define _VERSION "<version>"

#endif
```

- 文件: :file:`src/Makefile`

这个文件看起来很复杂，而且要注意所有缩进都是使用一个`<Tab>`键完成的缩进，千万不要错误的写成空格，因为这是:file:`Makefile`。这个文件除了定义如何由代码生成可执行文件:file:`hello`之外，还定义了如何将模板文件:file:`version.h.in`转换为:file:`version.h`。在转换过程中用:command:`git describe`命令的输出替换模板文件中的`<version>`字符串。

源代码:

```
OBJECTS = main.o
TARGET = hello

all: $(TARGET)

$(TARGET): $(OBJECTS)
    $(CC) -o $@ $^

main.o: | new_header
main.o: version.h

new_header:
    @sed -e "s/<version>/$(git describe)/g" \
        < version.h.in > version.h.tmp
    @if diff -q version.h.tmp version.h >/dev/null 2>&1; \
    then \
        rm version.h.tmp; \
    else \
        echo "version.h.in => version.h" ; \
        mv version.h.tmp version.h; \
    fi

clean:
    rm -f $(TARGET) $(OBJECTS) version.h

.PHONY: all clean
```

上述三个文件创建完毕之后，进入到`:file:`src``目录，试着运行一下。先执行`:command:`make``编译，再运行编译后的程序`:command:`hello``。

```
$ cd src
$ make
version.h.in => version.h
cc -c -o main.o main.c
cc -o hello main.o
$ ./hello
Hello, world.
version: old_practice-3-gc024f34.
```

## 使用`:command:`git add -i``选择性添加

刚刚创建的Hello World程序还没有添加到版本库中，在`:file:`src``目录下有下列文件：

```
$ cd /path/to/my/workspace/demo
$ ls src
hello  main.c  main.o  Makefile  version.h  version.h.in
```

这些文件中`:file:`hello``, `:file:`main.o``和`:file:`version.h``都是在编译时生成的程序，不应该加入到版本库中。那么选择性添加文件除了针对文件逐一使用`:command:`git add``命令外，还有什么办法么？通过使用`-i`参数调用`:command:`git add``就是一个办法，提供了一个交互式的界面。

执行`:command:`git add -i``命令，进入一个交互式界面，首先显示的是工作区状态。显然因为版本库进行了清理，所以显得很“干净”。

```
$ git add -i
      staged      unstaged path

*** Commands ***
1: status        2: update        3: revert        4: add untracked
5: patch        6: diff          7: quit          8: help
What now>
```

在交互式界面显示了命令列表，可以使用数字或者加亮显示的命令首字母，选择相应功能。对于此例需要将新文件加入到版本库，所以选择“4”。

```
What now> 4
1: src/Makefile
```

```
2: src/hello
3: src/main.c
4: src/main.o
5: src/version.h
6: src/version.h.in
Add untracked>>
```

当选择了“4”之后，就进入了“Add untracked”界面，显示了本地新增（尚不在版本库中）的文件列表，而且提示符也变了，由“What now>”变为“Add untracked>>”。依次输入1、3、6将源代码添加到版本库中。

- 输入“1”：

```
Add untracked>> 1
* 1: src/Makefile
2: src/hello
3: src/main.c
4: src/main.o
5: src/version.h
6: src/version.h.in
```

- 输入“3”：

```
Add untracked>> 3
* 1: src/Makefile
2: src/hello
* 3: src/main.c
4: src/main.o
5: src/version.h
6: src/version.h.in
```

- 输入“6”：

```
Add untracked>> 6
* 1: src/Makefile
2: src/hello
* 3: src/main.c
4: src/main.o
5: src/version.h
* 6: src/version.h.in
Add untracked>>
```

每次输入文件序号，对应的文件前面都添加一个星号，代表将此文件添加到暂存区。在提示符“Add untracked>>”处按回车键，完成文件添加，返回主界面。

```
Add untracked>>
added 3 paths

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now>
```

此时输入“1”查看状态，可以看到三个文件添加到暂存区中。

```
What now> 1
          staged      unstaged path
1: +20/-0      nothing src/Makefile
2: +10/-0      nothing src/main.c
3: +6/-0       nothing src/version.h.in

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
```

输入“7”退出交互界面。

查看文件状态，可以发现三个文件被添加到暂存区中。

```
$ git status -s
A  src/Makefile
A  src/main.c
A  src/version.h.in
?? src/hello
?? src/main.o
?? src/version.h
```

完成提交。

```
$ git commit -m "Hello world initialized."
[master d71ce92] Hello world initialized.
 3 files changed, 36 insertions(+), 0 deletions(-)
 create mode 100644 src/Makefile
 create mode 100644 src/main.c
 create mode 100644 src/version.h.in
```

## Hello world引发的新问题

进入`:file:`src``目录中，对Hello world执行编译。

```
$ cd /path/to/my/workspace/demo/src
$ make clean && make
rm -f hello main.o version.h
version.h.in => version.h
cc -c -o main.o main.c
cc -o hello main.o
```

运行编译后的程序，是不是对版本输出不满意呢？

```
$ ./hello
Hello, world.
version: old_practice-4-gd71ce92.
```

之所以显示长长的版本号，是因为使用了在本章最开始留的“影”。现在为Hello world留下一个新的“影”（一个新的里程碑）吧。

```
$ git tag -m "Set tag hello_1.0." hello_1.0
```

然后清除上次编译结果后，重新编译和运行，可以看到新的输出。

```
$ make clean && make
rm -f hello main.o version.h
version.h.in => version.h
cc -c -o main.o main.c
cc -o hello main.o
$ ./hello
Hello, world.
version: hello_1.0.
```

还不错，显示了新的版本号。此时在工作区查看状态，会发现工作区“不干净”。

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       hello
#       main.o
#       version.h
```

编译的目标文件和以及从模板生成的头文件出现在了Git的状态输出中，这些文件会对以后

的工作造成干扰。当写了新的源代码文件需要添加到版本库中时，因为这些干扰文件的存在，不得不一一将这些干扰文件排除在外。更为严重的是，如果不小心执行`:command:`git add .``或者`:command:`git add -A``命令会将编译的目标文件及其他临时文件加入版本库中，浪费存储空间不说甚至还会造成冲突。

Git提供了文件忽略功能，可以解决这个问题。

## 文件忽略

Git提供了文件忽略功能。当对工作区某个目录或者某些文件设置了忽略后，再执行`:command:`git status``查看状态时，被忽略的文件即使存在也不会显示为未跟踪状态，甚至根本感觉不到这些文件的存在。现在就针对Hello world程序目录试验一下。

```
$ cd /path/to/my/workspace/demo/src
$ git status -s
?? hello
?? main.o
?? version.h
```

可以看到`:file:`src``目录下编译的目标文件等显示为未跟踪，每一行开头的两个问号好像在向我们请求：“快把我们添加到版本库里吧”。

执行下面的命令可以在这个目下创建一个名为`:file:`.gitignore``的文件（注意文件的前面有个点），把这些要忽略的文件写在其中，文件名可以使用通配符。注意：第2行到第5行开头的右尖括号是`:command:`cat``命令的提示符，不是输入。

```
$ cat > .gitignore << EOF
> hello
> *.o
> *.h
> EOF
```

看看写好的`:file:`.gitignore``文件。每个要忽略的文件显示在一行。

```
$ cat .gitignore
hello
*.o
*.h
```

再来看看当前工作区的状态。

```
$ git status -s  
?? .gitignore
```

把`:file:`.gitignore``文件添加到版本库中吧。（如果不希望添加到库里，也不希望`:file:`.gitignore``文件带来干扰，可以在忽略文件中忽略自己。）

```
$ git add .gitignore  
$ git commit -m "ignore object files."  
[master b3af728] ignore object files.  
1 files changed, 3 insertions(+), 0 deletions(-)  
create mode 100644 src/.gitignore
```

`:file:`.gitignore``文件可以放在任何目录

文件`:file:`.gitignore``的作用范围是其所处的目录及其子目录，因此如果把刚刚创建的`:file:`.gitignore``移动到上一层目录（仍位于工作区内）也应该有效。

```
$ git mv .gitignore ..  
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       renamed:    .gitignore -> ../../.gitignore  
#
```

果然移动`:file:`.gitignore``文件到上层目录，`Hello world`程序目录下的目标文件依然被忽略着。

提交。

```
$ git commit -m "move .gitignore outside also works."  
[master 3488f2c] move .gitignore outside also works.  
1 files changed, 0 insertions(+), 0 deletions(-)  
rename src/.gitignore => .gitignore (100%)
```

**忽略文件有错误，后果很严重**

实际上面写的忽略文件不是非常好，为了忽略`:file:`version.h``，结果使用了通配符`*.h`会把源码目录下的有用的头文件也给忽略掉，导致应该添加到版本库的文件忘记添加。

在当前目录下创建一个新的头文件`:file:`hello.h``。

```
$ echo "/* test */" > hello.h
```

在工作区状态显示中看不到:`:file:`hello.h``文件。

```
$ git status  
# On branch master  
nothing to commit (working directory clean)
```

只有使用了`--ignored`参数，才会在状态显示中看到被忽略的文件。

```
$ git status --ignored -s  
!! hello  
!! hello.h  
!! main.o  
!! version.h
```

要添加:`:file:`hello.h``文件，使用`:command:`git add -A``和`:command:`git add .``都失效。无法用这两个命令将:`:file:`hello.h``添加到暂存区中。

```
$ git add -A  
$ git add .  
$ git st -s
```

只有在添加操作的命令行中明确的写入文件名，并且提供`-f`参数才能真正添加。

```
$ git add -f hello.h  
$ git commit -m "add hello.h"  
[master 48456ab] add hello.h  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 src/hello.h
```

忽略只对未跟踪文件有效，对于已加入版本库的文件无效

文件`:file:`hello.h``添加到版本库后，就不再受到`:file:`.gitignore``设置的文件忽略影响了，对`:file:`hello.h``的修改都会立刻被跟踪到。这是因为Git的文件忽略只是对未入库的文件起作用。

```
$ echo "/* end */" >> hello.h  
$ git status  
# On branch master  
# Changed but not updated:
```

```
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.h
#
no changes added to commit (use "git add" and/or "git commit -a")
```



偷懒式提交。（使用了-a参数提交，不用预先执行`:command:`git add`命令。）`

```
$ git commit -a -m "偷懒了，直接用 -a 参数直接提交。"
[master 613486c] 偷懒了，直接用 -a 参数直接提交。
 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 本地独享式忽略文件

文件`:file:`.gitignore``设置的文件忽略是共享式的。之所以称其为“共享式”，是因为`:file:`.gitignore``被添加到版本库后成为了版本库的一部分，当版本库共享给他人（克隆）或者把版本库推送（PUSH）到集中式的服务器（或他人的版本库），这个忽略文件就会出现在他人的工作区中，文件忽略在他人的工作区中同样生效。

与“共享式”忽略对应的是“独享式”忽略。独享式忽略就是不会因为版本库共享或者版本库之间的推送传递给他人的文件忽略。独享式忽略有两种方式：

- 一种是针对具体版本库的“独享式”忽略。即在版本库`:file:`.git``目录下的一个文件`:file:`.git/info/exclude``来设置文件忽略。
- 另外一种是全局的“独享式”忽略。即通过Git的配置变量`core.excludesfile`指定的一个忽略文件，其设置的忽略对所有文件均有效。

至于哪些情况需要通过向版本库中提交`:file:`.gitignore``文件设置共享式的文件忽略，哪些情况通过`:file:`.git/info/exclude``设置只对本地有效的独享式文件忽略，这取决于要设置的文件忽略是否具有普遍意义。如果文件忽略对于所有使用此版本库工作的人都有益，就通过在版本库相应的目录下创建一个`:file:`.gitignore``文件建立忽略，否则如果是需要忽略工作区中创建的一个试验目录或者试验性的文件，则使用本地忽略。

例如我的本地就设置着一个全局的独享的文件忽略列表（这个文件名可以随意设置）：

```
$ git config --global core.excludesfile /home/jiangxin/_gitignore
$ git config core.excludesfile
/home/jiangxin/_gitignore

$ cat /home/jiangxin/_gitignore
*~          # vim 临时文件
*.pyc       # python 的编译文件
```

```
.*.mmx      # 不是正则表达式哦，因为 FreeMind-MMX 的辅助文件以点开头
```

## Git忽略语法

Git的忽略文件的语法规则再多说几句。

- 忽略文件中的空行或者以井号 (#) 开始的行被忽略。
- 可以使用通配符，参见Linux手册：glob(7)。例如：星号 (\*) 代表任意多字符，问号 (?) 代表一个字符，方括号 ([abc]) 代表可选字符范围等。
- 如果名称的最前面是一个路径分隔符 (/)，表明要忽略的文件在此目录下，而非子目录的文件。
- 如果名称的最后面是一个路径分隔符 (/)，表明要忽略的是整个目录，同名文件不忽略，否则同名的文件和目录都忽略。
- 通过在名称的最前面添加一个感叹号 (!)，代表不忽略。

下面的文件忽略示例，包含了上述要点：

```
# 这是注释行 — 被忽略
*.a          # 忽略所有以 .a 为扩展名的文件。
!lib.a       # 但是 lib.a 文件或者目录不要忽略，即使前面设置了对 *.a 的忽略。
/TODO        # 只忽略根目录下的 TODO 文件，子目录的 TODO 文件不忽略。
build/       # 忽略所有 build/ 目录下的文件。
doc/*.txt    # 忽略文件如 doc/notes.txt，但是文件如 doc/server/arch.txt 不被忽略。
```

## 文件归档

如果使用压缩工具（tar、7zip、winzip、rar等）将工作区文件归档，一不小心会把版本库（:file:`.git` 目录）包含其中，甚至将工作区中的忽略文件、临时文件也包含其中。

Git提供了一个归档命令：:command:`git archive`，可以对任意提交对应的目录树建立归档。示例如下：

- 基于最新提交建立归档文件:file:`latest.zip`。

```
$ git archive -o latest.zip HEAD
```

- 只将目录:file:`src` 和:file:`doc` 建立到归档:file:`partial.tar` 中。

```
$ git archive -o partial.tar HEAD src doc
```

- 基于里程碑v1.0建立归档，并且为归档中文件添加目录前缀1.0。

```
$ git archive --format=tar --prefix=1.0/ v1.0 | gzip > foo-1.0.tar.gz
```

在建立归档时，如果使用树对象ID进行归档，则使用当前时间作为归档中文件的修改时间，而如果使用提交ID或里程碑等，则使用提交建立的时间作为归档中文件的修改时间。

如果使用tar格式建立归档，并且使用提交ID或里程碑ID，还会把提交ID记录在归档文件的文件头中。记录在文件头中的提交ID可以通过:command:`git tar-commit-id`命令获取。

如果希望在建立归档时忽略某些文件或目录，可以通过为相应文件或目录建立export-ignore属性加以实现。具体参见本书第8篇第41章“41.1 属性”一节。

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/070-git-basic.rst>

## 历史穿梭

经过了之前众多的实践，版本库中已经积累了很多次提交了，从下面的命令可以看出来有14次提交。

```
$ git rev-list HEAD | wc -l  
14
```

有很多工具可以研究和分析Git的历史提交，在前面的实践中已经用过很多相关的Git命令进行查看历史提交、查看文件的历史版本、进行差异比较等。本章除了对之前用到的相关Git命令作以总结外，还要再介绍几款图形化的客户端。

### 图形工具：gitk

gitk是最早实现的一个图形化的Git版本库浏览器软件，基于tcl/tk实现，因此gitk非常简洁，本身就是一个1万多行的tcl脚本写成的。gitk的代码已经和Git的代码放在同一个版本库中，gitk随Git一同发布，不用特别的安装即可运行。gitk可以显示提交的分支图，可以显示提交，文件，版本间差异等。

在版本库中调用gitk，就会浏览该版本库，显示其提交分支图。gitk可以像命令行工具一样使用不同的参数进行调用。

- 显示所有的分支。

```
$ gitk --all
```

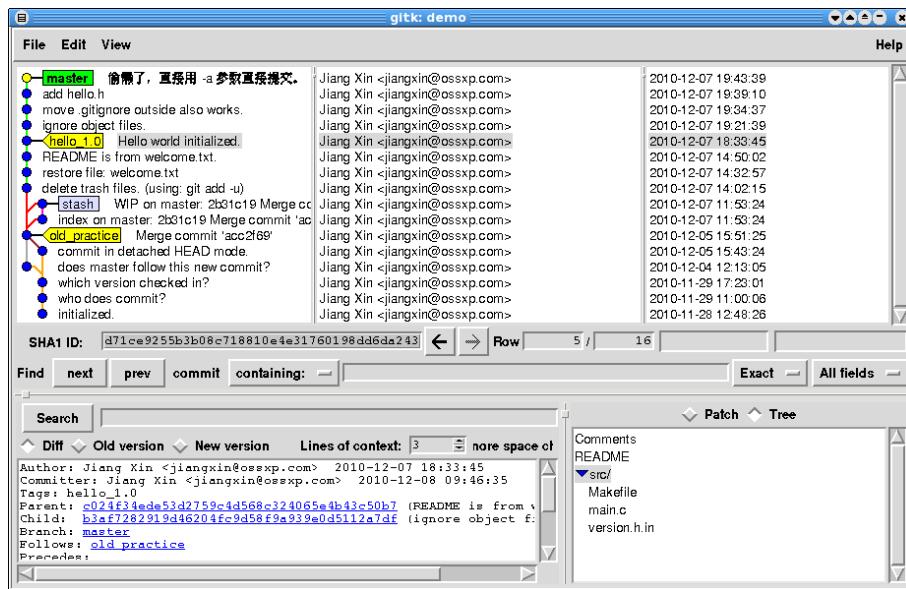
- 显示2周以来的提交。

```
$ gitk --since="2 weeks ago"
```

- 显示某个里程碑（v2.6.12）以来，针对某些目录和文件（:file:`include/scsi` 目录和:file:`drivers/scsi` 目录）的提交。

```
$ gitk v2.6.12.. include/scsi drivers/scsi
```

下面的图示就是在DEMO版本库中运行:command:`gitk --all`的显示。



在上图中可见不同颜色和形状区分的引用：

- 绿色的master分支。
- 黄色的hello\_1.0和old\_practice里程碑。
- 灰色的stash。

gitk使用tcl/tk开发，在显示上没有系统中原生图形应用那么漂亮的界面，甚至可以用丑陋来形容，下面介绍的gitg和qgit在易用性上比gitk进步了不少。

## 图形工具：gitg

gitg是使用GTK+图形库实现的一个Git版本库浏览器软件。Linux下最著名的Gnome桌面环境使用的就是GTK+，因此在Linux下gitg有着非常漂亮的原生的图形界面。gitg不但能够实现gitk的全部功能，即浏览提交历史和文件，还能帮助执行提交。

在Linux上安装gitg很简单，例如在Debian或Ubuntu上，直接运行下面的命令就可以进行安装。

```
$ sudo aptitude install gitg
```

安装完毕就可以在可执行路径中找到gitg。

```
$ which gitg
/usr/bin/gitg
```

为了演示gitg具备提交功能，先在工作区作出一些修改。

- 删除没有用到的:`file: `hello.h``文件。

```
$ cd /path/to/my/workspace/demo
$ rm src/hello.h
```

- 在:`file: `README``文件后面追加一行。

```
$ echo "Wait..." >> README
```

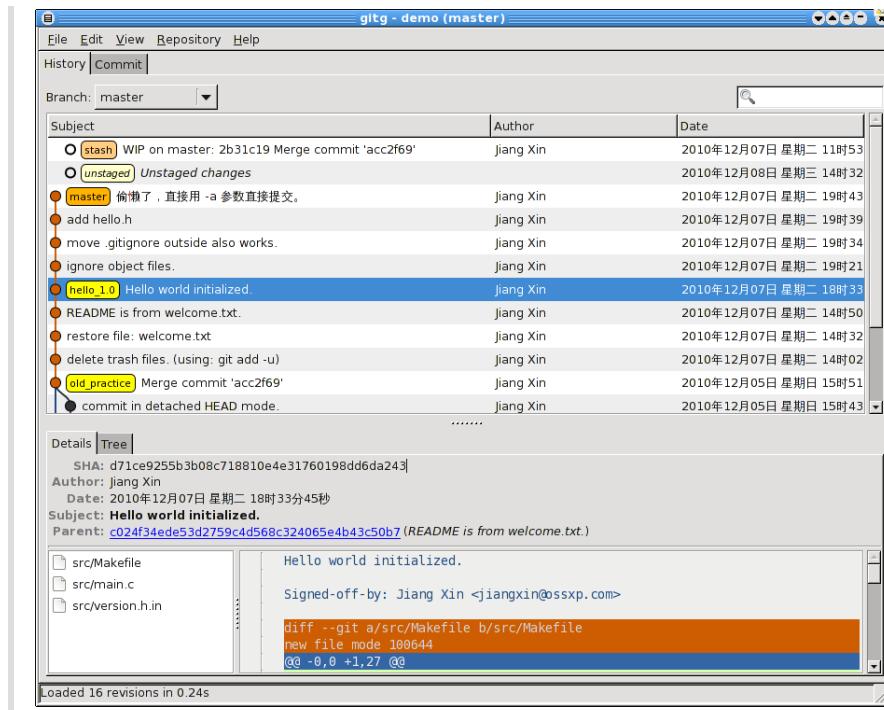
- 当前工作区的状态。

```
$ git status -s
M README
D src/hello.h
```

现在可以在工作区下执行gitg命令。

```
$ gitg &
```

下图就是gitg的缺省界面，显示了提交分支图，以及选中提交的提交信息和变更文件列表等。



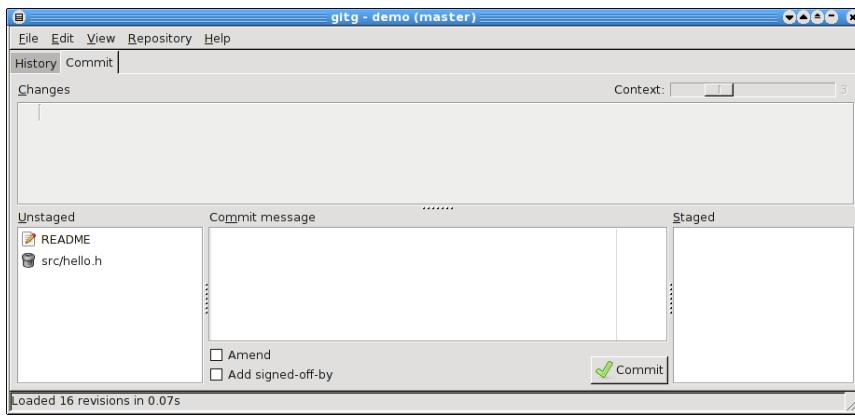
在上图中可以看见用不同颜色的标签显示的状态标识（包括引用）：

- 橙色的master分支。
- 黄色的hello\_1.0和old\_practice里程碑。
- 粉色的stash标签。
- 以及白色的显示工作区非暂存状态的标签。

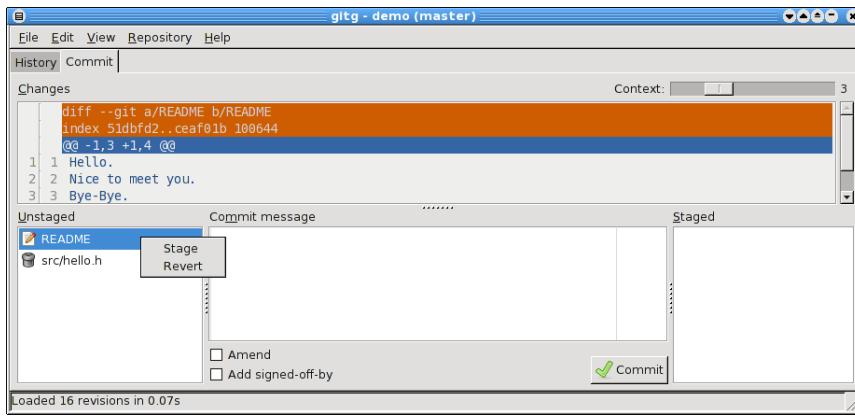
点击gitg下方窗口的标签“tree”，会显示此提交的目录树。



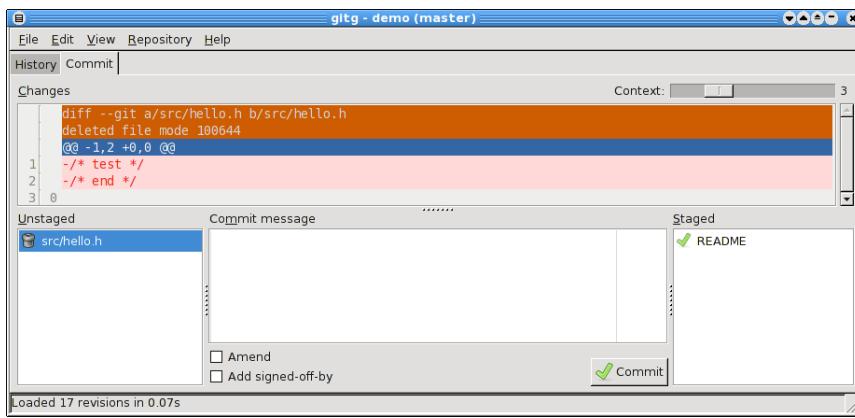
提交功能是gitg的一大特色。点击gitg顶部窗口的commit标签，显示下面的界面。



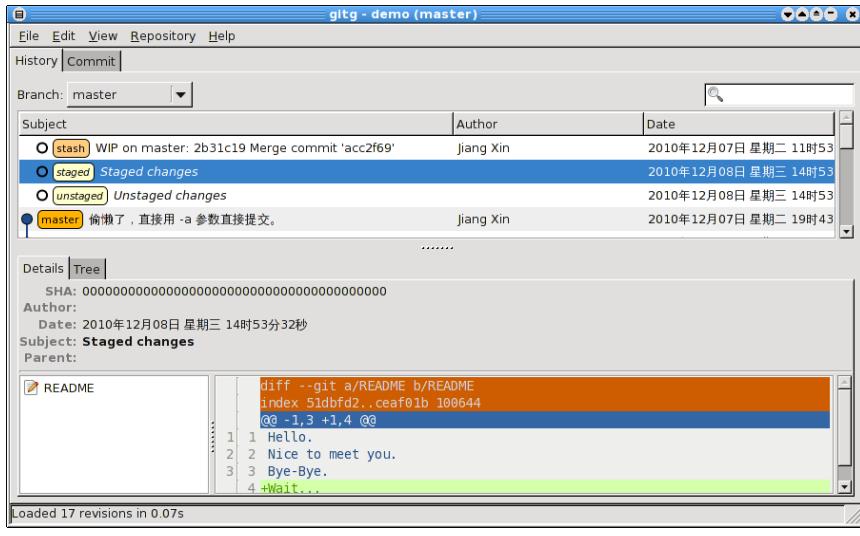
左下方窗口显示的是未更新到暂存区的本地改动。鼠标右击，在弹出菜单中选择“Stage”。



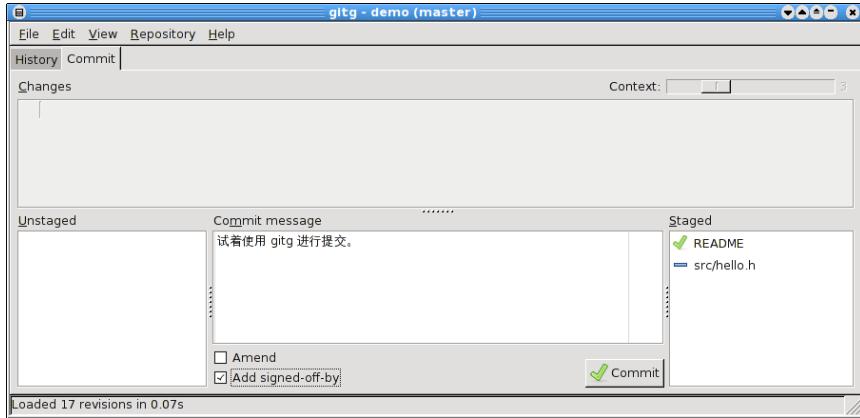
当把文件:`:file:`README``添加到暂存区后，可以看到:`:file:`README``文件出现在右下方的窗口中。



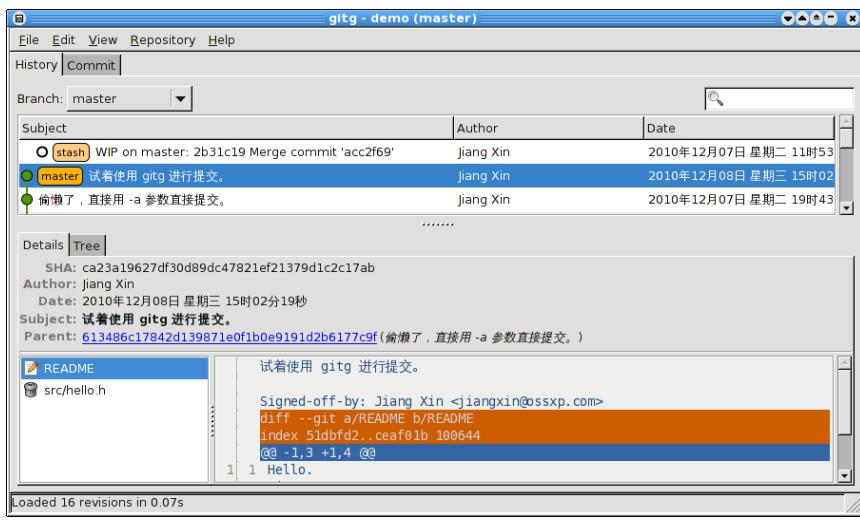
此时如果回到提交历史查看界面，可以看到在“stash”标签的下方，同时出现了“staged”和“unstaged”两个标签分别表示暂存区和工作区的状态。



当通过gitg的界面选择好要提交的文件（加入暂存区）之后，执行提交。



上图的提交说明对话框的下方有两个选项，当选择了“Add signed-off-by”选项后，在提交日志中会自动增加相应的说明文字。下图可以看到刚刚的提交已经显示在提交历史的最顶端，在提交说明中出现了Signed-off-by文字说明。



gitg还是一个比较新的项目，在本文撰写的时候，gitg才是0.0.6版本，相比下面要介绍的qgit还缺乏很多功能。例如gitg没有文件的blame（追溯）界面，也不能直接将文件检出，但是gitg整体的界面风格，以及易用的提交界面给人的印象非常深刻。

## 图形工具: qgit

前面介绍的gitg是基于GTK+这一Linux标准的图形库，那么也许有读者已经猜到qgit是使用Linux另外一个著名的图形库QT实现的Git版本库浏览器软件。QT的知名度不亚于GTK+，是著名的KDE桌面环境用到的图形库，也是蓄势待发准备和Android一较高低的MeeGo的UI核心。qgit目前的版本是2.3，相比前面介绍的gitg其经历的开发周期要长了不少，因此也提供了更多的功能。

在Linux上安装qgit很简单，例如在Debian或Ubuntu上，直接运行下面的命令就可以进行安装。

```
$ sudo aptitude install qgit
```

安装完毕就可以在可执行路径中找到qgit。

```
$ which qgit  
/usr/bin/qgit
```

qgit和gitg一样不但能够浏览提交历史和文件，还能帮助执行提交。为了测试提交，将在上一节所做的提交回滚。

- 使用重置命令回滚最后一次提交。

```
$ git reset HEAD^  
Unstaged changes after reset:  
M README  
M src/hello.h
```

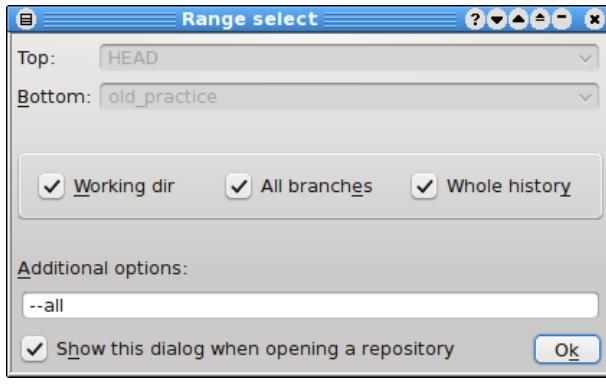
- 当前工作区的状态。

```
$ git status  
# On branch master  
# Changed but not updated:  
#   (use "git add/rm <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working direct  
#  
#       modified: README  
#       deleted: src/hello.h  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

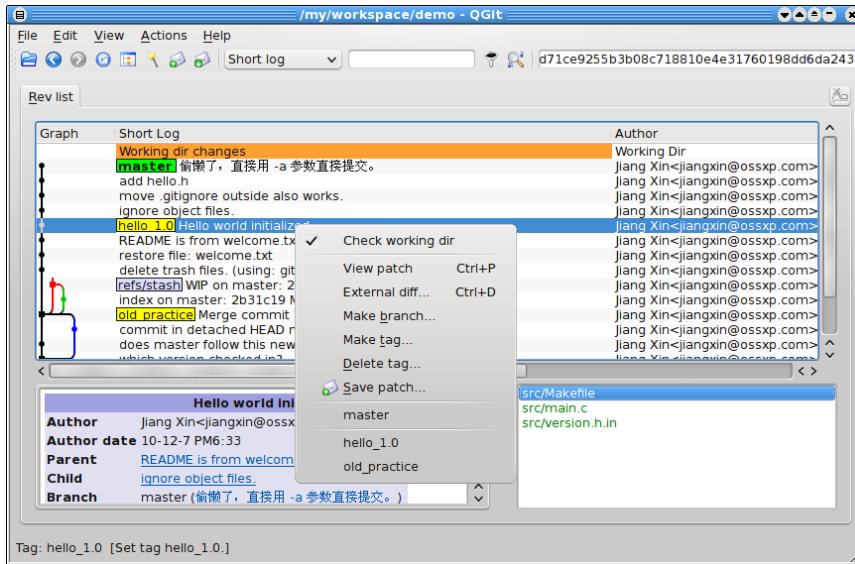
现在可以在工作区下执行qgit命令。

```
$ qgit &
```

启动qgit，首先弹出一个对话框，提示对显示的提交范围和分支范围进行选择。



对所有的选择打钩，显示下面的qgit的缺省界面。其中包括了提交分支图，以及选中提交的提交信息和变更文件列表等。



在上图中可以看见用不同颜色的标签显示的状态标识（包括引用）：

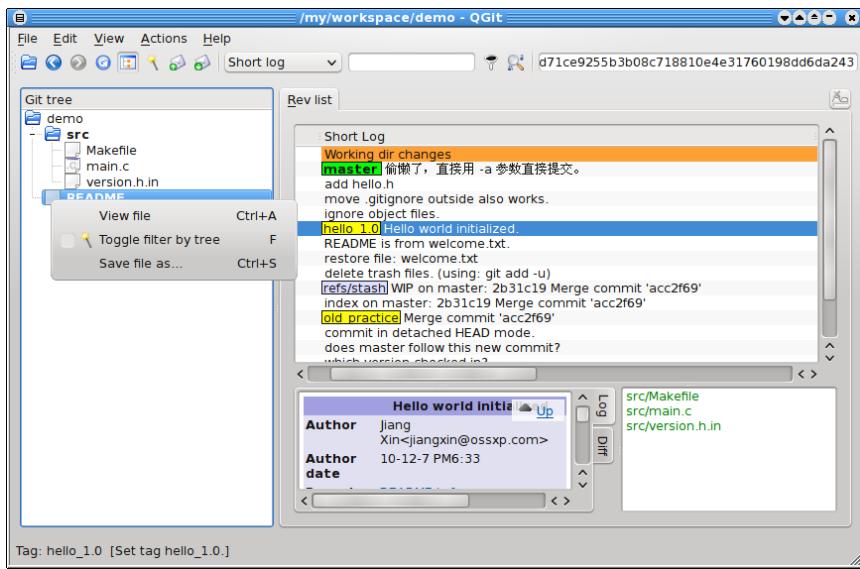
- 绿色的master分支。
- 黄色的hello\_1.0和old\_practice里程碑。
- 灰色的stash标签，显示在了创建时候的位置，并其包含的针对暂存区状态的提交也显示出来。
- 最顶端显示一行绿色背景的文件：工作区有改动。

qgit的右键菜单非常丰富，上图显示了鼠标右击提交时显示的弹出菜单，可以创建、切换标签或分支，可以将提交导出为补丁文件。

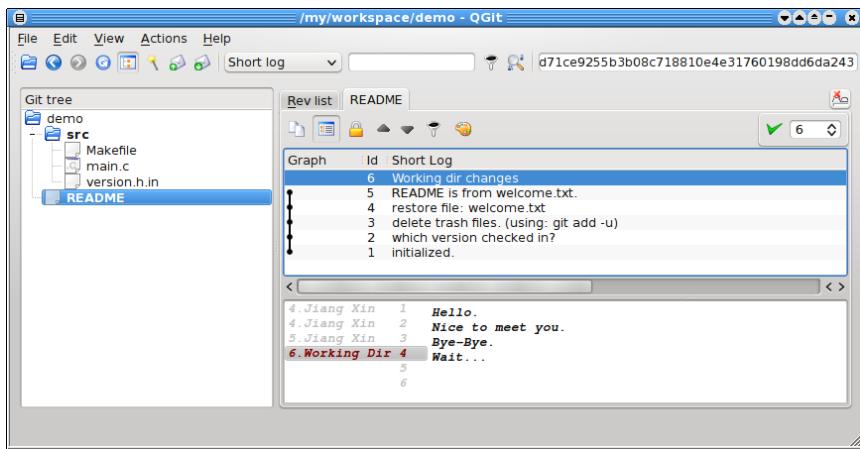
点击qgit右下方变更文件列表窗口，可以选择将文件检出或者直接查看。



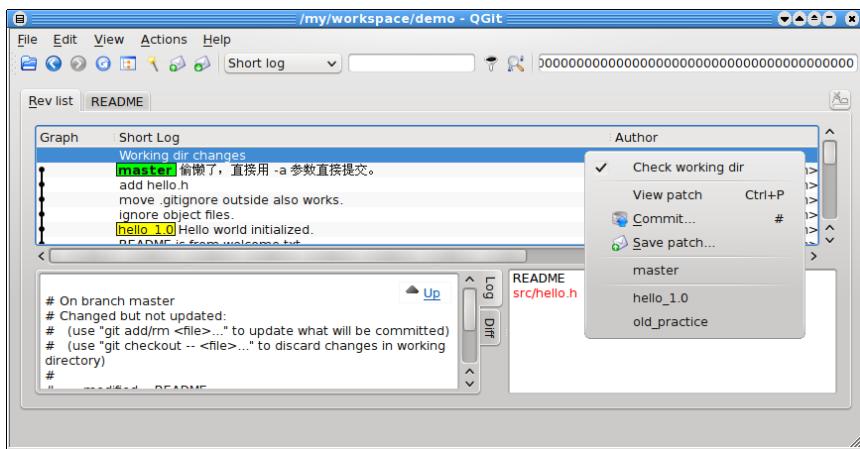
要想显示目录树，键入大写字母T，或者鼠标单击工具条上的图标 ，就会在左侧显示目录树窗口，如下。



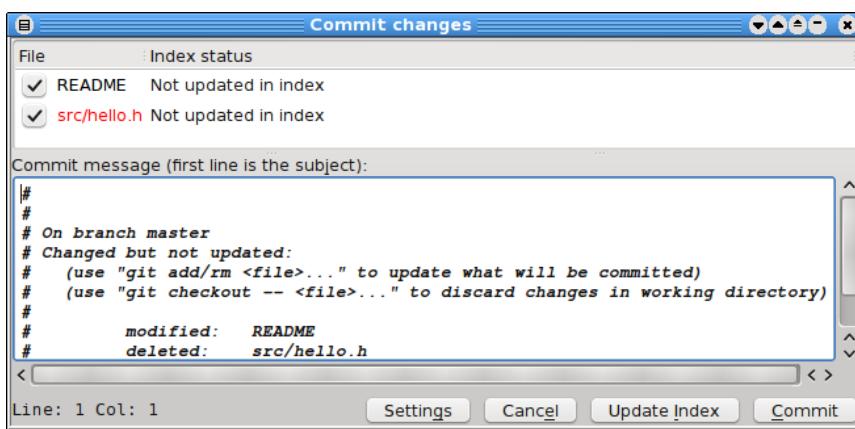
从上图也可以看到目录树的文件包含的右键菜单。当选择查看一个文件时，会显示此文件的追溯，即显示每一行是在哪个版本由谁修改的。追溯窗口见下图右下方窗口。



qgit也可以执行提交。选中qgit顶部窗口最上一行“Working dir changes”，鼠标右击，显示的弹出菜单包含了“Commit...”选项。

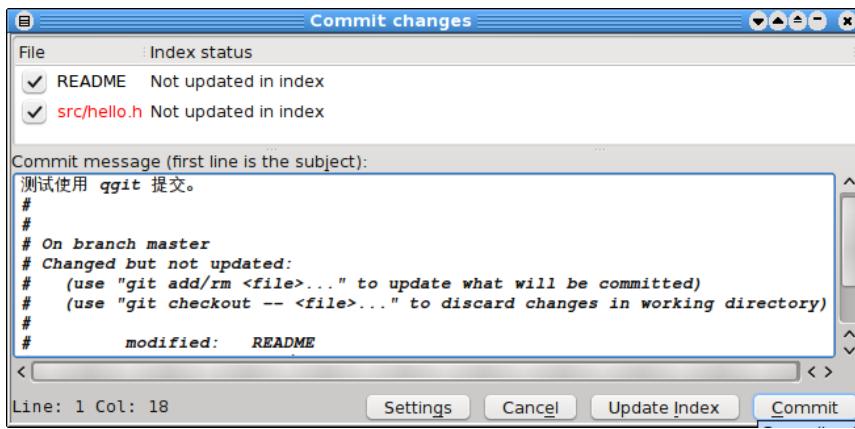


点击弹出菜单中的“Commit...”，显示下面的对话框。

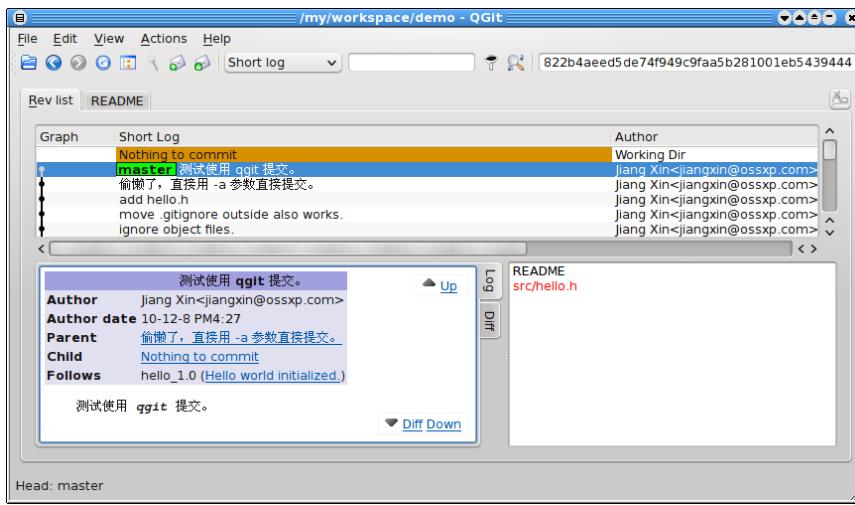


自动选中了所有的文件。上方窗口的选中文件目前状态是“Not updated in index”，就是说尚未添加到暂存区。

使用qgit做提交，只要选择好要提交的文件列表，即使未添加到暂存区，也可以直接提交。在下方的提交窗口写入提交日志，点击“Commit”按钮开始提交。



提交完毕返回qgit主界面，在显示的提交列表的最上方，原来显示的“Working dir changes”已经更新为“Nothing to commit”，并且可以看到刚刚的提交已经显示在提交历史的最顶端。



## 命令行工具

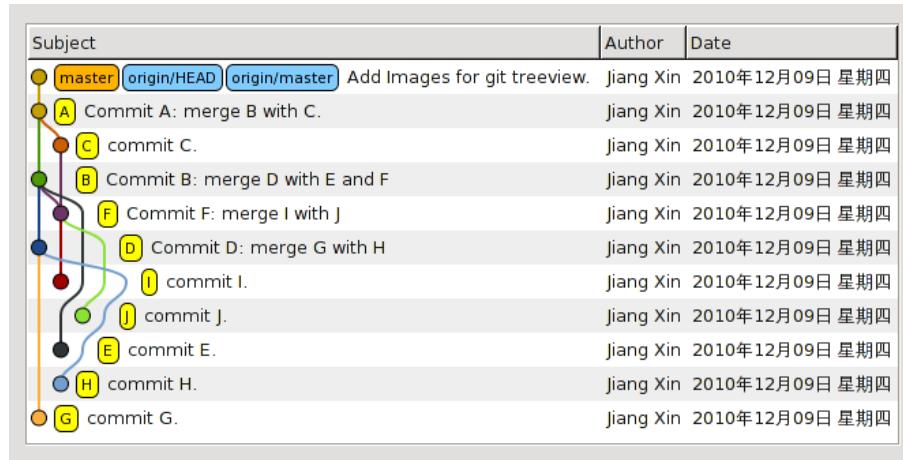
上面介绍的几款图形界面的Git版本库浏览器最大的特色就是更好看的提交关系图，还能非常方便的浏览历史提交的目录树，并从历史提交的目录树中提取文件等。这些操作对于Git

命令行同样可以完成。使用Git命令行探索版本库历史对于读者来说并不新鲜，因为在前几章的实践中已经用到了相关命令，展示了对历史记录的操作。本节对这些命令的部分要点进行强调和补充。

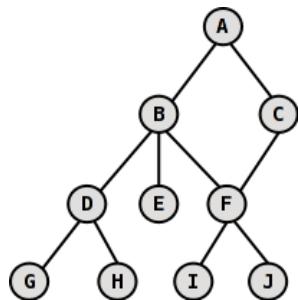
前面历次实践的提交基本上是线性的提交，研究起来没有挑战性。为了能够更加接近于实际又不失简洁，我构造了一个版本库，放在了Github上。可以通过如下操作在本地克隆这个示例版本库。

```
$ cd /path/to/my/workspace/
$ git clone git://github.com/ossxp-com/gitdemo-commit-tree.git
Cloning into gitdemo-commit-tree...
remote: Counting objects: 63, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 63 (delta 8), reused 0 (delta 0)
Receiving objects: 100% (63/63), 65.95 KiB, done.
Resolving deltas: 100% (8/8), done.
$ cd gitdemo-commit-tree
```

运行gitg命令，显示其提交关系图。



是不是有点“乱花渐欲迷人眼”的感觉。如果把提交用里程碑标识的圆圈来代表，稍加排列就会看到下面的更为直白的提交关系图。



Git的大部分命令可以使用提交版本作为参数（如：`:command:`git diff <commit-id>``），有的命令则使用一个版本范围作为参数（如：`:command:`git log <rev1>..<rev2>``）。Git的提交有着各式各样的表示法，提交范围也是一样，下面就通过两个命令`:command:`git rev-parse``和`:command:`git rev-list``分别研究一下Git的版本表示法和版本范围表示法。

## 版本表示法：`:command:`git rev-parse``

命令`:command:`git rev-parse``是Git的一个底层命令，其功能非常丰富（或者说杂乱），很多Git脚本或工具都会用到这条命令。

此命令的部分应用在“Git初始化”章节中就已经看到。例如可以显示Git版本库的位置（-

`--git-dir`，当前工作区目录的深度（`--show-cdup`），甚至可以用于被Git无关应用用于解析命令行参数（`--parseopt`）。

此命令可以显示当前版本库中的引用。

- 显示分支。

```
$ git rev-parse --symbolic --branches
```

- 显示里程碑。

```
$ git rev-parse --symbolic --tags
A
B
C
D
E
F
G
H
I
J
```

- 显示定义的所有引用。

其中`:file:`refs/remotes/``目录下的引用成为远程分支（或远程引用），在后面的章节会予以介绍。

```
$ git rev-parse --symbolic --glob=refs/*
refs/heads/master
refs/remotes/origin/HEAD
refs/remotes/origin/master
refs/tags/A
refs/tags/B
refs/tags/C
refs/tags/D
refs/tags/E
refs/tags/F
refs/tags/G
refs/tags/H
refs/tags/I
refs/tags/J
```

命令`:command:`git rev-parse``另外一个重要功能就是将一个Git对象表达式表示为对应的SHA1哈希值。针对本节开始克隆的版本库`gitdemo-commit-tree`，做如下操作。

- 显示HEAD对应的SHA1哈希值。

```
$ git rev-parse HEAD
6652a0dce6a5067732c00ef0a220810a7230655e
```

- 命令`:command:`git describe``的输出也可以显示为SHA1哈希值。

```
$ git describe
A-1-g6652a0d
$ git rev-parse A-1-g6652a0d
6652a0dce6a5067732c00ef0a220810a7230655e
```

- 可以同时显示多个表达式的SHA1哈希值。

下面的操作可以看出master和refs/heads/master都可以用于指代master分支。

```
$ git rev-parse master refs/heads/master  
6652a0dce6a5067732c00ef0a220810a7230655e  
6652a0dce6a5067732c00ef0a220810a7230655e
```

- 可以用哈希值的前几位指代整个哈希值。

```
$ git rev-parse 6652 6652a0d  
6652a0dce6a5067732c00ef0a220810a7230655e  
6652a0dce6a5067732c00ef0a220810a7230655e
```

- 里程碑的两种表示法均指向相同的对象。

里程碑对象不一定是提交，有可能是一个Tag对象。Tag对象包含说明或者签名，还包涵到对应提交的指向。

```
$ git rev-parse A refs/tags/A  
c9b03a208288aebdbfe8d84aeb984952a16da3f2  
c9b03a208288aebdbfe8d84aeb984952a16da3f2
```

- 里程碑A指向了一个Tag对象而非提交的时候，用下面的三个表示法都可以指向里程碑对应的提交。

实际上下面的语法也可以直接作用于轻量级里程碑（直接指向提交的里程碑）或者作用于提交本身。

```
$ git rev-parse A^{} A^0 A^{commit}  
81993234fc12a325d303eccea20f6fd629412712  
81993234fc12a325d303eccea20f6fd629412712  
81993234fc12a325d303eccea20f6fd629412712
```

- A的第一个父提交就是B所指向的提交。

回忆之前的介绍，`^`操作符代表着父提交。当一个提交有多个父提交时，可以通过在符号`^`后面跟上一个数字表示第几个父提交。`A^`就相当于`A^1`。而`B^0`代表了B所指向的一个Commit对象（因为B是Tag对象）。

```
$ git rev-parse A^ A^1 B^0  
776c5c9da9dcbb7e463c061d965ea47e73853b6e  
776c5c9da9dcbb7e463c061d965ea47e73853b6e  
776c5c9da9dcbb7e463c061d965ea47e73853b6e
```

- 更为复杂的表示法。

连续的`^`符号依次沿着父提交进行定位至某一祖先提交。`^`后面的数字代表该提交的第几个父提交。

```
$ git rev-parse A^^3^2 F^2 J^{}  
3252fcce40949a4a622a1ac012cb120d6b340ac8  
3252fcce40949a4a622a1ac012cb120d6b340ac8  
3252fcce40949a4a622a1ac012cb120d6b340ac8
```

- 记号`~<n>`就相当于连续`<n>`个符号`^`。

```
$ git rev-parse A~3 A^{tree} G^0
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
```

- 显示里程碑A对应的目录树。下面两种写法都可以。

```
$ git rev-parse A^{tree} A:
95ab9e7db14ca113d5548dc20a4872950e8e08c0
95ab9e7db14ca113d5548dc20a4872950e8e08c0
```

- 显示树里面的文件，下面两种表示法均可。

```
$ git rev-parse A^{tree}:src/Makefile A:src/Makefile
96554c5d4590dbde28183e9a6a3199d526eeb925
96554c5d4590dbde28183e9a6a3199d526eeb925
```

- 暂存区里的文件和HEAD中的文件相同。

```
$ git rev-parse :gitg.png HEAD:gitg.png
fc58966ccc1e5af24c2c9746196550241bc01c50
fc58966ccc1e5af24c2c9746196550241bc01c50
```

- 还可以通过在提交日志中查找字符串的方式显示提交。

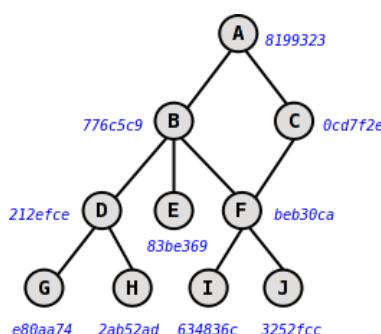
```
$ git rev-parse :/"Commit A"
81993234fc12a325d303eccea20f6fd629412712
```

- 再有就是reflog相关的语法，参见“Git重置”章节中关于reflog的介绍。

```
$ git rev-parse HEAD@{0} master@{0}
6652a0dce6a5067732c00ef0a220810a7230655e
6652a0dce6a5067732c00ef0a220810a7230655e
```

## 版本范围表示法: git rev-list

有的Git命令可以使用一个版本范围作为参数，命令`:command:`git rev-list``可以帮助研究Git的各种版本范围语法。



- 一个提交ID实际上就可以代表一个版本列表。含义是：该版本开始的所有历史提交。

```
$ git rev-list --oneline A
8199323 Commit A: merge B with C.
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
```

```
beb30ca Commit F: merge I with J  
212efce Commit D: merge G with H  
634836c commit I.  
3252fcc commit J.  
83be369 commit E.  
2ab52ad commit H.  
e80aa74 commit G.
```

- 两个或多个版本，相当于每个版本单独使用时指代的列表的并集。

```
$ git rev-list --oneline D F  
beb30ca Commit F: merge I with J  
212efce Commit D: merge G with H  
634836c commit I.  
3252fcc commit J.  
2ab52ad commit H.  
e80aa74 commit G.
```

- 在一个版本前面加上符号 (^) 含义是取反，即排除这个版本及其历史版本。

```
$ git rev-list --oneline ^G D  
212efce Commit D: merge G with H  
2ab52ad commit H.
```

- 和上面等价的“点点”表示法。使用两个点连接两个版本，如G..D，就相当于^G D。

```
$ git rev-list --oneline G..D  
212efce Commit D: merge G with H  
2ab52ad commit H.
```

- 版本取反，参数的顺序不重要，但是“点点”表示法前后的版本顺序很重要。

- 语法: ^B C

```
$ git rev-list --oneline ^B C  
0cd7f2e commit C.
```

- 语法: C ^B

```
$ git rev-list --oneline C ^B  
0cd7f2e commit C.
```

- 语法: B..C相当于^B C

```
$ git rev-list --oneline B..C  
0cd7f2e commit C.
```

- 语法: C..B相当于^C B

```
$ git rev-list --oneline C..B  
776c5c9 Commit B: merge D with E and F  
212efce Commit D: merge G with H  
83be369 commit E.  
2ab52ad commit H.  
e80aa74 commit G.
```

- 三点表示法的含义是两个版本共同能够访问到的除外。

B和C共同能够访问到的F、I、J排除在外。

```
$ git rev-list --oneline B...C
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
212efce Commit D: merge G with H
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 三点表示法，两个版本的前后顺序没有关系。

实际上r1...r2相当于r1 r2 --not \$(git merge-base --all r1 r2)，和顺序无关。

```
$ git rev-list --oneline C...B
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
212efce Commit D: merge G with H
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 某提交的历史提交，自身除外，用语法r1^@表示。

```
$ git rev-list --oneline B^@
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
634836c commit I.
3252fcc commit J.
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 提交本身不包括其历史提交，用语法r1^!表示。

```
$ git rev-list --oneline B^!
776c5c9 Commit B: merge D with E and F

$ git rev-list --oneline F^! D
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
2ab52ad commit H.
```

## 浏览日志：:command:`git log`

命令:command:`git log`是老朋友了，在前面的章节中曾经大量的出现，用于显示提交历史。

### 参数代表版本范围

当不使用任何参数调用，相当于使用了缺省的参数HEAD，即显示当前HEAD能够访问到的所有历史提交。还可以使用上面介绍的版本范围表示法，例如：

```
$ git log --oneline F^! D
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
2ab52ad commit H.
```

```
e80aa74 commit G.
```

### 分支图显示

通过`--graph`参数调用`:command:`git log``可以显示字符界面的提交关系图，而且不同的分支还可以用不同的颜色来表示。如果希望每次查看日志的时候都看到提交关系图，可以设置一个别名，用别名来调用。

```
$ git config --global alias.glog "log --graph"
```

定义别名之后，每次希望自动显示提交关系图，就可以使用别名命令：

```
$ git glog --oneline
* 6652a0d Add Images for git treeview.
*   8199323 Commit A: merge B with C.
| \
| * 0cd7f2e commit C.
| |
| |
*-. \   776c5c9 Commit B: merge D with E and F
| \ \ \
| | | /
| | * beb30ca Commit F: merge I with J
| | | \
| | | * 3252fcc commit J.
| | * 634836c commit I.
| * 83be369 commit E.
*   212efce Commit D: merge G with H
| \
| * 2ab52ad commit H.
* e80aa74 commit G.
```

### 显示最近的几条日志

可以使用参数`-n <n>`（`<n>`为数字），显示最近的`<n>`条日志。例如下面的命令显示最近的3条日志。

```
$ git log -3 --pretty=oneline
6652a0dce6a5067732c00ef0a220810a7230655e Add Images for git treeview.
81993234fc12a325d303eccea20f6fd629412712 Commit A: merge B with C.
0cd7f2ea245d90d414e502467ac749f36aa32cc4 commit C.
```

### 显示每次提交的具体改动

使用参数`-p`可以在显示日志的时候同时显示改动。

```
$ git log -p -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

        Add Images for git treeview.

        Signed-off-by: Jiang Xin <jiangxin@ossp.com>

diff --git a/gitg.png b/gitg.png
new file mode 100644
index 000000..fc58966
Binary files /dev/null and b/gitg.png differ
diff --git a/treeview.png b/treeview.png
```

```
new file mode 100644
index 000000..a756d12
Binary files /dev/null and b/treeview.png differ
```

因为是二进制文件改动，缺省不显示改动的内容。实际上Git的差异文件提供对二进制文件的支持，在后面“Git应用”章节予以专题介绍。

### 显示每次提交的变更概要

使用-p参数会让日志输出显得非常冗余，当不需要知道具体的改动而只想知道改动在哪些文件上，可以使用--stat参数。输出的变更概要像极了Linux的`:command:`diffstat`命令的输出。`

```
$ git log --stat --oneline I..C
0cd7f2e commit C.
 README | 1 +
 doc/C.txt | 1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
beb30ca Commit F: merge I with J
3252fcc commit J.
 README | 7 ++++++
 doc/J.txt | 1 +
 src/.gitignore | 3 ++
 src/Makefile | 27 ++++++++++++++++++++++++
 src/main.c | 10 ++++++++
 src/version.h.in | 6 ++++++
 6 files changed, 54 insertions(+), 0 deletions(-)
```

### 定制输出

Git的差异输出命令提供了很多输出模板提供选择，可以根据需要选择冗余显示或者精简显示。

- 参数`--pretty=raw`显示提交的原始数据。可以显示提交对应的树ID。

```
$ git log --pretty=raw -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
tree e33be9e8e7ca5f887c7d5601054f2f510e6744b8
parent 81993234fc12a325d303eccea20f6fd629412712
author Jiang Xin <jiangxin@ossxp.com> 1291882031 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291882892 +0800
```

Add Images for git treeview.

Signed-off-by: Jiang Xin <jiangxin@ossxp.com>

- 参数`--pretty=fuller`会同时显示作者和提交者，两者可以不同。

```
$ git log --pretty=fuller -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Thu Dec 9 16:07:11 2010 +0800
Commit: Jiang Xin <jiangxin@ossxp.com>
CommitDate: Thu Dec 9 16:21:32 2010 +0800
```

Add Images for git treeview.

Signed-off-by: Jiang Xin <jiangxin@ossxp.com>

- 参数`--pretty=oneline`显然会提供最精简的日志输出。也可以使用`--oneline`参数，效

果近似。

```
$ git log --pretty=oneline -1  
6652a0dce6a5067732c00ef0a220810a7230655e Add Images for git treeview.
```

如果只想查看、分析某一个提交，也可以使用`:command:`git show``或者`:command:`git cat-file``命令。

- 使用`:command:`git show``显示里程碑D及其提交：

```
$ git show D --stat  
tag D  
Tagger: Jiang Xin <jiangxin@ossp.com>  
Date: Thu Dec 9 14:24:52 2010 +0800  
  
create node D  
  
commit 212efce1548795a1edb08e3708a50989fc73cce  
Merge: e80aa74 2ab52ad  
Author: Jiang Xin <jiangxin@ossp.com>  
Date: Thu Dec 9 14:06:34 2010 +0800  
  
Commit D: merge G with H  
  
Signed-off-by: Jiang Xin <jiangxin@ossp.com>  
  
 README      |    2 ++
 doc/D.txt   |    1 +
 doc/H.txt   |    1 +
 3 files changed, 4 insertions(+), 0 deletions(-)
```

- 使用`:command:`git cat-file``显示里程碑D及其提交。

参数-p的含义是美观的输出（pretty）。

```
$ git cat-file -p D^0  
tree 1c22e90c6bf150ee1cde6cef476abbb921f491f  
parent e80aa7481beda65ae00e35afc4bc4b171f9b0ebf  
parent 2ab52ad2a30570109e71b56fa1780f0442059b3c  
author Jiang Xin <jiangxin@ossp.com> 1291874794 +0800  
committer Jiang Xin <jiangxin@ossp.com> 1291875877 +0800  
  
Commit D: merge G with H  
  
Signed-off-by: Jiang Xin <jiangxin@ossp.com>
```

## 差异比较：`:command:`git diff``

Git差异比较功能在前面的实践中也反复的接触过了，尤其是在介绍暂存区的相关章节重点介绍了`:command:`git diff``命令如何对工作区、暂存区、版本库进行比较。

- 比较里程碑B和里程碑A，用命令：`:command:`git diff B A``
- 比较工作区和里程碑A，用命令：`:command:`git diff A``
- 比较暂存区和里程碑A，用命令：`:command:`git diff --cached A``
- 比较工作区和暂存区，用命令：`:command:`git diff``
- 比较暂存区和HEAD，用命令：`:command:`git diff --cached``
- 比较工作区和HEAD，用命令：`:command:`git diff HEAD``

差异比较还可以使用路径参数，只显示不同版本间该路径下文件的差异。语法格式：

```
$ git diff <commit1> <commit2> -- <paths>
```

### 非Git目录/文件的差异比较

命令`:command:`git diff``还可以在Git版本库之外执行，对非Git目录进行比较，就像GNU的`:command:`diff``命令一样。之所以提供这个功能是因为Git差异比较命令更为强大，提供了对GNU差异比较的扩展支持。

```
$ git diff <path1> <path2>
```

### 扩展的差异语法

Git扩展了GNU的差异比较语法，提供了对重命名、二进制文件、文件权限变更的支持。在后面的“Git应用”辟专题介绍二进制文件的差异比较和补丁的应用。

#### 逐词比较，而非缺省的逐行比较

Git的差异比较缺省是逐行比较，分别显示改动前的行和改动后的行，到底改动哪里还需要仔细辨别。Git还提供一种逐词比较的输出，有的人会更喜欢。使用`--word-diff`参数可以显示逐词比较。

```
$ git diff --word-diff
diff --git a/src/book/02-use-git/080-git-history-travel.rst b/src/book/02-use
index f740203..2dd3e6f 100644
--- a/src/book/02-use-git/080-git-history-travel.rst
+++ b/src/book/02-use-git/080-git-history-travel.rst
@@ -681,7 +681,7 @@ Git的大部分命令可以使用提交版本作为参数（如: git diff）,
::
[-18:23:48 jiangxin@hp:~/gitwork/gitbook/src/book$-]{+$+} git log --stat --
0cd7f2e commit C.
 README | 1 +
 doc/C.txt | 1 +
```

上面的逐词差异显示是有颜色显示的：删除内容`[-...]`用红色表示，添加的内容`{+...+}`用绿色表示。

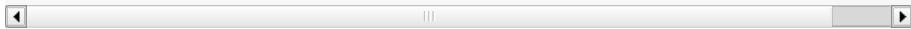
### 文件追溯：`:command:`git blame``

在软件开发过程中当发现Bug并定位到具体的代码时，Git的文件追溯命令可以指出是谁在什么时候，什么版本引入的此Bug。

当针对文件执行`:command:`git blame``命令，就会逐行显示文件，在每一行的行首显示此行最早是在什么版本引入的，由谁引入。

```
$ cd /path/to/my/workspace/gitdemo-commit-tree
$ git blame README
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 1) DEMO program for git-scm-bo
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 2)
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 3) Changes
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 4) =====
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 5)
81993234 (Jiang Xin 2010-12-09 14:30:15 +0800 6) * create node A.
0cd7f2ea (Jiang Xin 2010-12-09 14:29:09 +0800 7) * create node C.
776c5c9d (Jiang Xin 2010-12-09 14:27:31 +0800 8) * create node B.
```

```
beb30ca7 (Jiang Xin 2010-12-09 14:11:01 +0800 9) * create node F.
^3252fcc (Jiang Xin 2010-12-09 14:00:33 +0800 10) * create node J.
^634836c (Jiang Xin 2010-12-09 14:00:33 +0800 11) * create node I.
^83be369 (Jiang Xin 2010-12-09 14:00:33 +0800 12) * create node E.
212efce1 (Jiang Xin 2010-12-09 14:06:34 +0800 13) * create node D.
^2ab52ad (Jiang Xin 2010-12-09 14:00:33 +0800 14) * create node H.
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 15) * create node G.
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 16) * initialized.
```



只想查看某几行，使用-L n,m参数，如下：

```
$ git blame -L 6,+5 README
81993234 (Jiang Xin 2010-12-09 14:30:15 +0800 6) * create node A.
0cd7f2ea (Jiang Xin 2010-12-09 14:29:09 +0800 7) * create node C.
776c5c9d (Jiang Xin 2010-12-09 14:27:31 +0800 8) * create node B.
beb30ca7 (Jiang Xin 2010-12-09 14:11:01 +0800 9) * create node F.
^3252fcc (Jiang Xin 2010-12-09 14:00:33 +0800 10) * create node J.
```

## 二分查找：:command:`git bisect`

前面的文件追溯是建立在问题（Bug）已经定位（到代码上）的基础之上，然后才能通过错误的行（代码）找到人（提交者），打板子（教育或惩罚）。那么如何定位问题呢？Git的二分查找命令可以提供帮助。

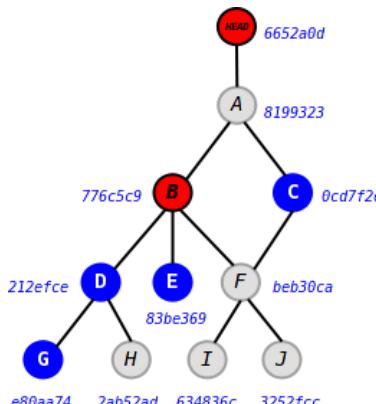
二分查找并不神秘，也不是万灵药，是建立在测试的基础之上的。实际上每个进行过软件测试的人都曾经使用过：“最新的版本出现Bug了，但是在给某某客户的版本却没有这个问题，所以问题肯定出在两者之间的某次代码提交上”。

Git提供的:command:`git bisect`命令是基于版本库的，自动化的问题查找和定位工作流程。取代传统软件测试中粗放式的、针对软件发布版本的、无法定位到代码的测试。

执行二分查找，在发现问题后，首先要找到一个正确的版本，如果所发现的问题从软件最早的版本就是错的，那么就没有必要执行二分查找了，还是老老实实的Debug吧。但是如果能够找到一个正确的版本，即在这个正确的版本上问题没有发生，那么就可以开始使用:command:`git bisect`命令在版本库中进行二分查找了：

1. 工作区切换到已知的“好版本”和“坏版本”的中间的一个版本。
2. 执行测试，问题重现，将版本库当前版本库为“坏版本”，如果问题没有重现，将当前版本标记为“好版本”。
3. 重复1-2，直至最终找到第一个导致问题出现的版本。

下面是示例版本库标记了提交ID后的示意图，在这个示例版本库中试验二分查找流程：首先标记最新提交（HEAD）是“坏的”，G提交是好的，然后通过查找最终定位到坏提交（B）。



在下面的试验中定义坏提交的依据很简单，如果在:file:`doc/`目录中包含文

件:`:file:`B.txt``，则此版本是“坏”的。（这个示例太简陋，不要见笑，聪明的读者可以直接通过:`:file:`doc/B.txt``文件就可追溯到B提交。）

下面开始通过手动测试（查找:`:file:`doc/B.txt``存在与否），借助Git二分查找定位“问题”版本。

- 首先确认工作在master分支。

```
$ cd /path/to/my/workspace/gitdemo-commit-tree/  
$ git checkout master  
Already on 'master'
```

- 开始二分查找。

```
$ git bisect start
```

- 已经当前版本是“坏提交”，因为存在文件:`:file:`doc/B.txt``。而G版本是“好提交”，因为不存在文件:`:file:`doc/B.txt``。

```
$ git cat-file -t master:doc/B.txt  
blob  
$ git cat-file -t G:doc/B.txt  
fatal: Not a valid object name G:doc/B.txt
```

- 将当前版本（HEAD）标记为“坏提交”，将G版本标记为“好提交”。

```
$ git bisect bad  
$ git bisect good G  
Bisecting: 5 revisions left to test after this (roughly 2 steps)  
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
```

- 自动定位到C提交。没有文件:`:file:`doc/B.txt``，也是一个好提交。

```
$ git describe  
C  
$ ls doc/B.txt  
ls: 无法访问doc/B.txt: 没有那个文件或目录
```

- 标记当前版本（C提交）为“好提交”。

```
$ git bisect good  
Bisecting: 3 revisions left to test after this (roughly 2 steps)  
[212efce1548795a1edb08e3708a50989fcfd73cce] Commit D: merge G with H
```

- 现在定位到D版本，这也是一个“好提交”。

```
$ git describe  
D  
$ ls doc/B.txt  
ls: 无法访问doc/B.txt: 没有那个文件或目录
```

- 标记当前版本（D提交）为“好提交”。

```
$ git bisect good  
Bisecting: 1 revision left to test after this (roughly 1 step)
```

```
[776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with E and F
```



- 现在定位到B版本，这是一个“坏提交”。

```
$ git bisect bad  
Bisecting: 0 revisions left to test after this (roughly 0 steps)  
[83be36956c007d7bffffe13805dd2081839fd3603] commit E.
```

- 现在定位到E版本，这是一个“好提交”。当标记E为好提交之后，输出显示已经成功定位到引入坏提交的最接近的版本。

```
$ git bisect good  
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
```

- 最终定位的坏提交用引用:`:file:`refs/bisect/bad``标识。可以如下方法切换到该版本。

```
$ git checkout bisect/bad  
Previous HEAD position was 83be369... commit E.  
HEAD is now at 776c5c9... Commit B: merge D with E and F
```

- 当对“Bug”定位和修复后，撤销二分查找在版本库中遗留的临时文件和引用。

撤销二分查找后，版本库切换回执行二分查找之前所在的分支。

```
$ git bisect reset  
Previous HEAD position was 776c5c9... Commit B: merge D with E and F  
Switched to branch 'master'
```

### 把“好提交”标记成了“坏提交”该怎么办？

在执行二分查找的过程中，一不小心就有可能犯错，将“好提交”标记为“坏提交”，或者相反。这将导致前面的查找过程也前功尽弃。Git的二分查找提供一个恢复查找进度的办法。

- 例如对E提交，本来是一个“好版本”却被错误的标记为“坏版本”。

```
$ git bisect bad  
83be36956c007d7bffffe13805dd2081839fd3603 is the first bad commit
```

- 用:`:command:`git bisect log``命令查看二分查找的日志记录。

把二分查找的日志保存在一个文件中。

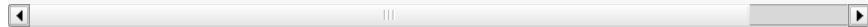
```
$ git bisect log > logfile
```

- 编辑这个文件，删除记录了错误动作的行。

以井号（#）开始的行是注释。

```
$ cat logfile  
# bad: [6652a0dce6a5067732c00ef0a220810a7230655e] Add Images for git tree  
# good: [e80aa7481beda65ae00e35afc4bc4b171f9b0ebf] commit G.  
git bisect start 'master' 'G'  
# good: [0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
```

```
git bisect good 0cd7f2ea245d90d414e502467ac749f36aa32cc4
# good: [212efce1548795a1edb08e3708a50989fc73cce] Commit D: merge G with
git bisect good 212efce1548795a1edb08e3708a50989fc73cce
# bad: [776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with
git bisect bad 776c5c9da9dcbb7e463c061d965ea47e73853b6e
```



- 结束上一次出错的二分查找。

```
$ git bisect reset
Previous HEAD position was 83be369... commit E.
Switched to branch 'master'
```

- 通过日志文件恢复进度。

```
$ git bisect replay logfile
We are not bisecting.
Bisection: 5 revisions left to test after this (roughly 2 steps)
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
Bisection: 0 revisions left to test after this (roughly 0 steps)
[83be36956c007d7bffffe13805dd2081839fd3603] commit E.
```

- 再一次回到了提交E，这一次不要标记错了。

```
$ git describe
E
$ git bisect good
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
```

## 二分查找使用自动化测试

Git的二分查找命令支持run子命令，可以运行一个自动化测试脚本。

- 如果脚本的退出码是0，正在测试的版本是一个“好版本”。
- 如果脚本的退出码是125，正在测试的版本被跳过。
- 如果脚本的退出码是1到127（125除外），正在测试的版本是一个“坏版本”。

对于本例写一个自动化测试太简单了，无非就是判断文件是否存在，存在返回错误码1，不存在返回错误码0。

测试脚本:[file: `good-or-bad.sh`](#)如下：

```
#!/bin/sh

[ -f doc/B.txt ] && exit 1
exit 0
```

用此自动脚本执行二分查找就非常简单了。

- 从已知的坏版本master和好版本G，开始新一轮的二分查找。

```
$ git bisect start master G
Bisection: 5 revisions left to test after this (roughly 2 steps)
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
```

- 自动化测试，使用脚本:[file: `good-or-bad.sh`](#)。

```
$ git bisect run sh good-or-bad.sh
```

```
running sh good-or-bad.sh
Bisection: 3 revisions left to test after this (roughly 2 steps)
[212efce1548795a1edb08e3708a50989fcfd73cce] Commit D: merge G with H
running sh good-or-bad.sh
Bisection: 1 revision left to test after this (roughly 1 step)
[776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with E and F
running sh good-or-bad.sh
Bisection: 0 revisions left to test after this (roughly 0 steps)
[83be36956c007d7bffffe13805dd2081839fd3603] commit E.
running sh good-or-bad.sh
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
bisect run success
```



- 定位到的“坏版本”是B。

```
$ git describe refs/bisect/bad
B
```

## 获取历史版本

提取历史提交中的文件无非就是下面表格中的操作，在之前的实践中多次用到，不再赘述。

动作	命令格式	示例
查看历史提交的目录树	git ls-tree <tree-ish> <paths>	<ul style="list-style-type: none"><li>• git ls-tree 776c5c9 README</li><li>• git ls-tree -r refs/tags/D doc</li></ul>
整个工作区切换到历史版本	git checkout <commit>	<ul style="list-style-type: none"><li>• git checkout HEAD^~</li></ul>
检出某文件的历史版本	git checkout <commit> -- <paths>	<ul style="list-style-type: none"><li>• git checkout refs/tags/D -- README</li><li>• git checkout 776c5c9 -- doc</li></ul>
检出某文件的历史版本到其他文件名	git show <commit>:<file> > new_name	<ul style="list-style-type: none"><li>• git show 887113d:README &gt; README.OLD</li></ul> <p>来源: <a href="https://github.com/gotgit/gotgit/blob/master/02-git-solo/080-git-history-travel.rst">https://github.com/gotgit/gotgit/blob/master/02-git-solo/080-git-history-travel.rst</a></p>

# 改变历史

我是《回到未来》的粉丝，偶尔会做梦，梦见穿梭到未来拿回一本2000-2050体育年鉴。操作Git可以体验到穿梭时空的感觉，因为Git像极了一个时光机器，不但允许你在历史中穿梭，而且能够改变历史。

在本章的最开始，介绍两种最简单和最常用的历史变更操作——“悔棋”操作，就是对刚刚进行的一次或几次提交进行修补。对于跳跃式的历史记录的变更，即仅对过去某一个或某几个提交作出改变，会在“回到未来”小节详细介绍。在“丢弃历史”小节会介绍一种版本库瘦身的方法，这可能会在某些特定的场合用到。

作为分布式版本控制系统，一旦版本库被多人共享，改变历史就可能是无法完成的任务。在本章的最后，介绍还原操作实现在不改变历史提交的情况下还原错误的改动。

## 悔棋

在日常的Git操作中，会经常出现这样的状况，输入`:command:`git commit``命令刚刚敲下回车键就后悔了：可能是提交说明中出现了错别字，或者有文件忘记提交，或者有的修改不应该提交，诸如此类。

像Subversion那样的集中式版本控制系统是“落子无悔”的系统，只能叹一口气责怪自己太不小心了。然后根据实际情况弥补：马上做一次新提交改正前面的错误；或者只能将错就错：错误的提交说明就让它一直错下去吧，因为大部分Subversion管理员不敢或者不会放开修改提交说明的功能导致无法对提交说明进行修改。

Git提供了“悔棋”的操作，甚至因为“单步悔棋”是如此经常的发生，以至于Git提供了一个简洁的操作——修补式提交，命令是：`:command:`git commit --amend``。

看看当前版本库最新的两次提交：

```
$ cd /path/to/my/workspace/demo
$ git log --stat -2
commit 822b4aeed5de74f949c9faa5b281001eb5439444
Author: Jiang Xin <jiangxin@osxp.com>
Date:   Wed Dec 8 16:27:41 2010 +0800
```

测试使用 qgit 提交。

```
README      |    1 +
src/hello.h |    2 --
2 files changed, 1 insertions(+), 2 deletions(-)
```

```
commit 613486c17842d139871e0f1b0e9191d2b6177c9f
Author: Jiang Xin <jiangxin@ossp.com>
Date: Tue Dec 7 19:43:39 2010 +0800
```

偷懒了，直接用 -a 参数直接提交。

```
src/hello.h | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

最新一次的提交是的确是在上一章使用qgit进行的提交，但这和提交内容无关，因此需要改掉这个提交的提交说明。使用下面的命令即可做到。

```
$ git commit --amend -m "Remove hello.h, which is useless."
[master 7857772] Remove hello.h, which is useless.
2 files changed, 1 insertions(+), 2 deletions(-)
delete mode 100644 src/hello.h
```

上面的命令使用了-m参数是为了演示的方便，实际上完全可以直接输入:`git commit --amend``，在弹出的提交说明编辑界面修改提交说明，然后保存退出完成修补提交。

下面再看看最近两次的提交说明，可以看到最新的提交说明更改了（包括提交的SHA1哈希值），而它的父提交（即前一次提交）没有改变。

```
$ git log --stat -2
commit 78577724305e3e20aa9f2757ac5531d037d612a6
Author: Jiang Xin <jiangxin@ossp.com>
Date: Wed Dec 8 16:27:41 2010 +0800
```

Remove hello.h, which is useless.

```
README      | 1 +
src/hello.h | 2 --
2 files changed, 1 insertions(+), 2 deletions(-)
```

```
commit 613486c17842d139871e0f1b0e9191d2b6177c9f
```

```
Author: Jiang Xin <jiangxin@ossp.com>
```

```
Date: Tue Dec 7 19:43:39 2010 +0800
```

偷懒了，直接用 -a 参数直接提交。

```
src/hello.h | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

如果最后一步操作不想删除文件:`file:`src/hello.h``，而只是想修改:`file:`README``，则

可以按照下面的方法进行修补操作。

- 还原删除的:file:`src/hello.h`文件。

```
$ git checkout HEAD^ -- src/hello.h
```

- 此时查看状态，会看到:file:`src/hello.h`被重新添加回暂存区。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   src/hello.h
#
```

- 执行修补提交，不过提交说明是不是也要更改呢，因为毕竟这次提交不会删除文件了。

```
$ git commit --amend -m "commit with --amend test."
[master 2b45206] commit with --amend test.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 再次查看最近两次提交，会发现最新的提交不再删除文件:file:`src/hello.h`了。

```
$ git log --stat -2
commit 2b452066ef6e92bceb999cf94fcce24afb652259
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Wed Dec 8 16:27:41 2010 +0800

    commit with --amend test.

 README |      1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

```
commit 613486c17842d139871e0f1b0e9191d2b6177c9f
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Tue Dec 7 19:43:39 2010 +0800
```

偷懒了，直接用 -a 参数直接提交。

```
src/hello.h |      1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 多步悔棋

Git能够提供悔棋的奥秘在于Git的重置命令。实际上上面介绍的单步悔棋也可以用重置命令来实现，只不过Git提供了一个更好用的更简洁的修补提交命令而已。多步悔棋顾名思义就是可以取消最新连续的多次提交，多次悔棋并非是所有分布式版本控制系统都具有的功能，像Mercurial/Hg只能对最新提交悔棋一次（除非使用MQ插件）。Git因为有了强大的重置命令，可以悔棋任意多次。

多步悔棋会在什么场合用到呢？软件开发中针对某个特性功能的开发就是一例。某个开发工程师领受某个特性开发的任务，于是在本地版本库进行了一系列开发、测试、修补、再测试的流程，最终特性功能开发完毕后可能在版本库中留下了多次提交。在将本地版本库改动推送（PUSH）到团队协同工作的核心版本库时，这个开发人员就想用多步悔棋的操作，将多个试验性的提及合为一个完整的提交。

以DEMO版本库为例，看看版本库最近的三次提交。

```
$ git log --stat --pretty=oneline -3
2b452066ef6e92bceb999cf94fcce24afb652259 commit with --amend test.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
613486c17842d139871e0f1b0e9191d2b6177c9f 偷懒了，直接用 -a 参数直接提交。
src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
48456abfaeab706a44880eabcd63ea14317c0be9 add hello.h
src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

想要将最近的两个提交压缩为一个，并把提交说明改为“modify hello.h”，可以使用如下方法进行操作。

- 使用`--soft`参数调用重置命令，回到最近两次提交之前。

```
$ git reset --soft HEAD^^
```

- 版本状态和最新日志。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#       modified:   src/hello.h
```

```
#  
$ git log -1  
commit 48456abfaeab706a44880eabcd63ea14317c0be9  
Author: Jiang Xin <jiangxin@osxp.com>  
Date:   Tue Dec 7 19:39:10 2010 +0800  
  
    add hello.h
```

- 执行提交操作，即完成最新两个提交压缩为一个提交的操作。

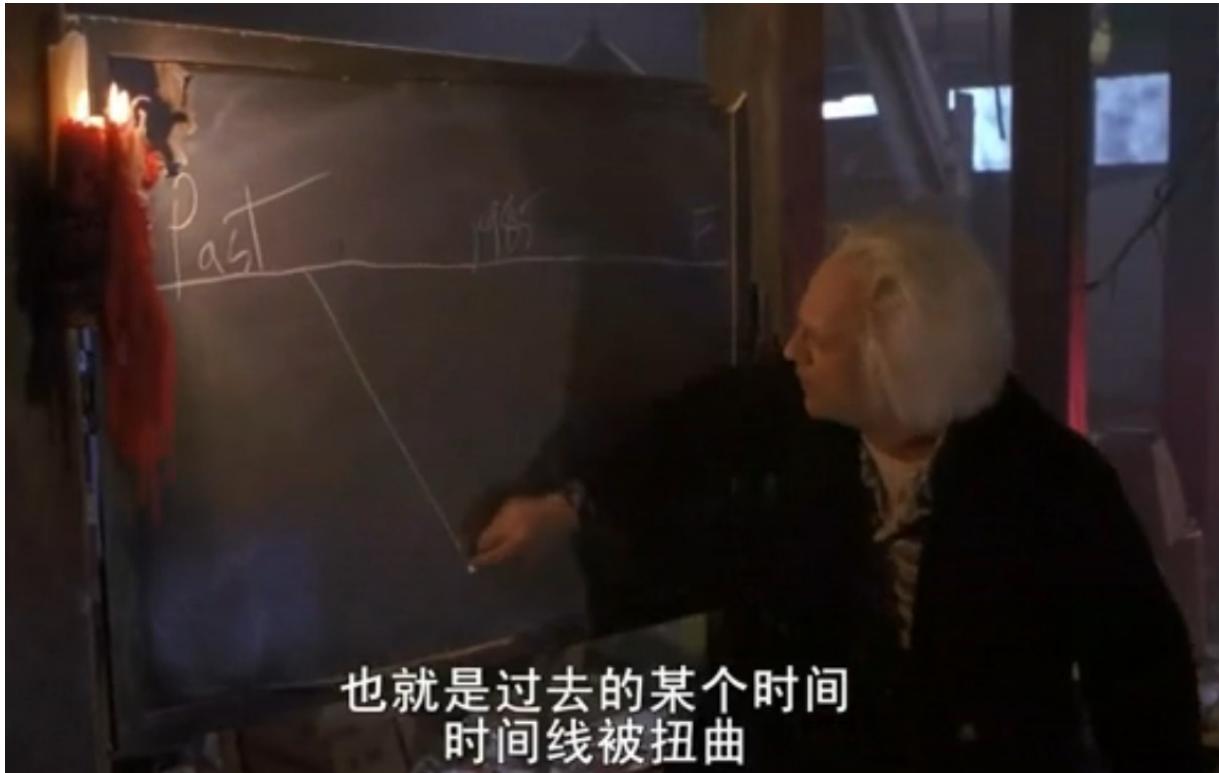
```
$ git commit -m "modify hello.h"  
[master b6f0b0a] modify hello.h  
 2 files changed, 2 insertions(+), 0 deletions(-)
```

- 看看提交日志，“多步悔棋”操作成功。

```
$ git log --stat --pretty=oneline -2  
b6f0b0a5237bc85de1863dbd1c05820f8736c76f modify hello.h  
  README | 1 +  
src/hello.h | 1 +  
 2 files changed, 2 insertions(+), 0 deletions(-)  
48456abfaeab706a44880eabcd63ea14317c0be9 add hello.h  
  src/hello.h | 1 +  
 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 回到未来

电影《回到未来》(Back to future)第二集，老毕福偷走时光车，到过去(1955年)给了小毕福一本书，导致未来大变。



布朗博士正在解释为何产生两个平行的未来

Git这一台“时光机”也有这样的能力，或者说也会具有这样的行为。当更改历史提交（SHA1哈希值变更），即使后续提交的内容和属性都一致，但是因为后续提交中有一个属性是父提交的SHA1哈希值，所以一个历史提交的改变会引起连锁变化，导致所有后续提交必然的发生变化，就会形成两条平行的时间线：一个是变更前的提交时间线，另外一条是更改历史后新的提交时间线。

把此次实践比喻做一次电影（回到未来）拍摄的话，舞台依然是之前的DEMO版本库，而剧本是这样的。

- 角色：最近的六次提交。分别依据提交顺序，编号为A、B、C、D、E、F。

```
$ git log --oneline -6
b6f0b0a modify hello.h          # F
48456ab add hello.h            # E
3488f2c move .gitignore outside also works. # D
b3af728 ignore object files.   # C
d71ce92 Hello world initialized. # B
c024f34 README is from welcome.txt. # A
```

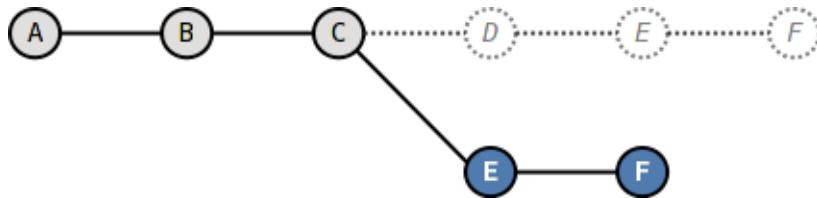
- 坏蛋：提交D。

即对`:file:`.gitignore``文件移动的提交不再需要，或者这个提交将和前一次提交(C)压缩为一个。

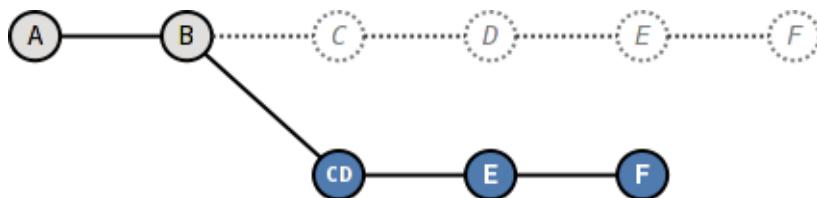
- 前奏：故事人物依次出场，坏蛋D在图中被特殊标记。



- 第一幕：抛弃提交D，将正确的提交E和F重新“嫁接”到提交C上，最终坏蛋被消灭。



- 第二幕：坏蛋D被C感化，融合为“CD”复合体，E和F重新“嫁接”到“CD”复合体上，最终大团圆结局。



- 道具：分别使用三辆不同的时光车来完成“回到未来”。

分别是：核能跑车，清洁能源飞车，蒸汽为动力的飞行火车。

## 时间旅行一

《回到未来-第一集》布朗博士设计的第一款时间旅行车是一辆跑车，使用核燃料：钚。与之对应，此次实践使用的工具也没有太出乎想象，用一条新的指令——拣选指令（`:command:`git cherry-pick``）实现提交在新的分支上“重放”。

拣选指令——`:command:`git cherry-pick``，其含义是从众多的提交中挑选出一个提交应用在当前的工作分支中。该命令需要提供一个提交ID作为参数，操作过程相当于将该提交导出为补丁文件，然后在当前HEAD上重放形成无论内容还是提交说明都一致的提交。

首先对版本库要“参演”的角色进行标记，使用尚未正式介绍的命令`:command:`git tag``（无非就是在特定命名空间建立的引用，用于对提交的标识）。

```
$ git tag F
$ git tag E HEAD^
$ git tag D HEAD^^
$ git tag C HEAD^^^
$ git tag B HEAD~4
$ git tag A HEAD~5
```

通过日志，可以看到被标记的6个提交。

```
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

### 现在演出第一幕：干掉坏蛋D

- 执行`git checkout`命令，暂时将HEAD头指针切换到C。`

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout C
Note: checking out 'C'.

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 b3af728... ignore object files.
```

- 执行拣选操作将E提交在当前HEAD上重放。

因为E和`master^`显然指向同一角色，因此可以用下面的语法。

```
$ git cherry-pick master^
[detached HEAD fa0b076] add hello.h
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 src/hello.h
```

- 执行拣选操作将F提交在当前HEAD上重放。

F和`master`也具有相同指向。

```
$ git cherry-pick master
```

```
[detached HEAD f677821] modify hello.h  
2 files changed, 2 insertions(+), 0 deletions(-)
```

- 通过日志可以看到坏蛋D已经不在了。

```
$ git log --oneline --decorate -6  
f677821 (HEAD) modify hello.h  
fa0b076 add hello.h  
b3af728 (tag: C) ignore object files.  
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.  
c024f34 (tag: A) README is from welcome.txt.  
63992f0 restore file: welcome.txt
```

- 通过日志还可以看出来，最新两次提交的原始创作日期（AuthorDate）和提交日期（CommitDate）不同。AuthorDate是拣选提交的原始更改时间，而CommitDate是拣选操作时的时间，因此拣选后的新提交的SHA1哈希值也不同于所拣选的原提交的SHA1哈希值。

```
$ git log --pretty=fuller --decorate -2  
commit f677821dfc15acc22ca41b48b8ebaab5ac2d2fea (HEAD)  
Author: Jiang Xin <jiangxin@osssxp.com>  
AuthorDate: Sun Dec 12 12:11:00 2010 +0800  
Commit: Jiang Xin <jiangxin@osssxp.com>  
CommitDate: Sun Dec 12 16:20:14 2010 +0800  
  
modify hello.h  
  
commit fa0b076de600a53e8703545c299090153c6328a8  
Author: Jiang Xin <jiangxin@osssxp.com>  
AuthorDate: Tue Dec 7 19:39:10 2010 +0800  
Commit: Jiang Xin <jiangxin@osssxp.com>  
CommitDate: Sun Dec 12 16:18:34 2010 +0800  
  
add hello.h
```

- 最重要的一步操作，就是要将master分支指向新的提交ID（f677821）上。

下面的切换操作使用了reflog的语法，即HEAD@{1}相当于切换回master分支前的HEAD指向，即f677821。

```
$ git checkout master  
Previous HEAD position was f677821... modify hello.h  
Switched to branch 'master'  
$ git reset --hard HEAD@{1}  
HEAD is now at f677821 modify hello.h
```

- 使用qgit查看版本库提交历史。

Graph	Short Log	Author	Author Date
	Nothing to commit	Working Dir	
	master modify hello.h	Jiang Xin<jiangxin...	10-12-12 PM12:11
•	add hello.h	Jiang Xin<jiangxin...	10-12-7 PM7:39
•	modify hello.h	Jiang Xin<jiangxin...	10-12-12 PM12:11
•	E add hello.h	Jiang Xin<jiangxin...	10-12-7 PM7:39
D	move .gitignore outside also works.	Jiang Xin<jiangxin...	10-12-7 PM7:34
C	ignore object files.	Jiang Xin<jiangxin...	10-12-7 PM7:21
B	hello_1.0 Hello world initialized.	Jiang Xin<jiangxin...	10-12-7 PM6:33
A	README is from welcome.txt.	Jiang Xin<jiangxin...	10-12-7 PM2:50
	restore file: welcome.txt	Jiang Xin<jiangxin...	10-12-7 PM2:32

## 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将master分支重新置回到提交F上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

布景完毕，大幕即将再次拉开。

## 现在演出第二幕：坏蛋D被感化，融入社会

- 执行`:command:`git checkout`命令，暂时将HEAD头指针切换到坏蛋D。`

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout D
Note: checking out 'D'.

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 3488f2c... move .gitignore outside also works.
```

- 悔棋两次，以便将C和D融合。

```
$ git reset --soft HEAD^^
```

- 执行提交，提交说明重用C提交的提交说明。

```
$ git commit -C C  
[detached HEAD 53e621c] ignore object files.  
 1 files changed, 3 insertions(+), 0 deletions(-)  
  create mode 100644 .gitignore
```

- 执行拣选操作将E提交在当前HEAD上重放。

```
$ git cherry-pick E  
[detached HEAD 1f99f82] add hello.h  
 1 files changed, 1 insertions(+), 0 deletions(-)  
  create mode 100644 src/hello.h
```

- 执行拣选操作将F提交在当前HEAD上重放。

```
$ git cherry-pick F  
[detached HEAD 2f13d3a] modify hello.h  
 2 files changed, 2 insertions(+), 0 deletions(-)
```

- 通过日志可以看到提交C和D被融合，所以在日志中看不到C的标签。

```
$ git log --oneline --decorate -6  
2f13d3a (HEAD) modify hello.h  
1f99f82 add hello.h  
53e621c ignore object files.  
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.  
c024f34 (tag: A) README is from welcome.txt.  
63992f0 restore file: welcome.txt
```

- 最重要的一步操作，就是要将master分支指向新的提交ID（2f13d3a）上。

下面的切换操作使用了reflog的语法，即HEAD@{1}相当于切换回master分支前的HEAD指向，即2f13d3a。

```
$ git checkout master
Previous HEAD position was 2f13d3a... modify hello.h
Switched to branch 'master'
$ git reset --hard HEAD@{1}
HEAD is now at 2f13d3a modify hello.h
```

- 使用gitk查看版本库提交历史。



## 别忘了后台的重新布景

为了接下来的时间旅行二能够顺利开始，需要重新布景，将master分支重新置回到提交F上。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

## 时间旅行二

《回到未来-第二集》布朗博士改进的时间旅行车使用了未来科技，是陆天两用的飞车，而且燃料不再依赖核物质，而是使用无所不在的生活垃圾。而此次实践使用的工具也进行了升级，采用强大的:command:`git rebase`命令。

命令:command:`git rebase`是对提交执行变基操作，即可以实现将指定范围的提交“嫁接”到另外一个提交之上。其常用的命令行格式有：

```
用法1: git rebase --onto <newbase> <since>      <till>
用法2: git rebase --onto <newbase> <since>
用法3: git rebase          <newbase>                  <till>
用法4: git rebase          <newbase>
用法5: git rebase -i ...
```

```
用法6: git rebase --continue  
用法7: git rebase --skip  
用法8: git rebase --abort
```

不要被上面的语法吓到，用法5会在下节（时间旅行三）中予以介绍，后三种用法则是变基运行过程被中断时可采用的命令——继续变基或终止等。

- 用法6是在变基遇到冲突而暂停后，当完成冲突解决后（添加到暂存区，不提交），恢复变基操作的时候使用。
- 用法7是在变基遇到冲突而暂停后，跳过当前提交的时候使用。
- 用法8是在变基遇到冲突后，终止变基操作，回到之前的分支时候使用。

而前四个用法如果把省略的参数补上（方括号内是省略掉的参数），看起来就都和用法1就一致了。

```
用法1: git rebase --onto <newbase> <since> <till>  
用法2: git rebase --onto <newbase> <since> [HEAD]  
用法3: git rebase [--onto] <newbase> [<newbase>] <till>  
用法4: git rebase [--onto] <newbase> [<newbase>] [HEAD]
```

下面就以归一化的`:command:`git rebase``命令格式来介绍其用法。

```
命令格式: git rebase --onto <newbase> <since> <till>
```

变基操作的过程：

- 首先会执行`:command:`git checkout``切换到`<till>`。  
因为会切换到`<till>`，因此如果`<till>`指向的不是一个分支（如`master`），则变基操作是在`detached HEAD`（分离头指针）状态进行的，当变基结束后，还要像在“时间旅行一”中那样，对`master`分支执行重置以实现把变基结果记录在分支中。
- 将`<since>..<till>`所标识的提交范围写到一个临时文件中。

还记得前面介绍的版本范围语法，`<since>..<till>`是指包括`<till>`的所有历史提交排除`<since>`以及`<since>`的历史提交后形成的版本范围。

- 当前分支强制重置（`git reset --hard`）到`<newbase>`。  
相当于执行：`:command:`git reset --hard <newbase>``。
- 从保存在临时文件中的提交列表中，一个一个将提交按照顺序重新提交到重置之后的分支上。

- 如果遇到提交已经在分支中包含，跳过该提交。
- 如果在提交过程遇到冲突，变基过程暂停。用户解决冲突后，执行`git rebase --continue`继续变基操作。或者执行`git rebase --skip`跳过此提交。或者执行`git rebase --abort`就此终止变基操作切换到变基前的分支上。

很显然为了执行将E和F提交跳过提价D，“嫁接”到C提交上。可以如此执行变基命令：

```
$ git rebase --onto C E^ F
```

因为E^等价于D，并且F和当前HEAD指向相同，因此可以这样操作：

```
$ git rebase --onto C D
```

有了对变基命令的理解，就可以开始新的“回到未来”之旅了。

确认舞台已经布置完毕。

```
$ git status -s -b
## master
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

现在演出第一幕：干掉坏蛋D

- 执行变基操作。

因为下面的变基操命令行使用了参数F。F是一个里程碑指向一个提交，而非master，会导致后面变基完成还需要对master分支执行重置。在第二幕中会使用master，会发现省事不少。

```
$ git rebase --onto C E^ F
First, rewinding head to replay your work on top of it...
Applying: add hello.h
Applying: modify hello.h
```

- 最后一步必需的操作，就是要将master分支指向变基后的提交上。

下面的切换操作使用了reflog的语法，即HEAD@{1}相当于切换回master分支前的HEAD指向，即3360440。

```
$ git checkout master
Previous HEAD position was 3360440... modify hello.h
Switched to branch 'master'
$ git reset --hard HEAD@{1}
HEAD is now at 3360440 modify hello.h
```

- 经过检查，操作完毕，收工。

```
$ git log --oneline --decorate -6
3360440 (HEAD, master) modify hello.h
1ef3803 add hello.h
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

## 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将master分支重新置回到提交F上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

布景完毕，大幕即将再次拉开。

## 现在演出第二幕：坏蛋D被感化，融入社会

- 执行`:command:`git checkout`命令`，暂时将HEAD头指针切换到坏蛋D。

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout D
Note: checking out 'D'.

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 3488f2c... move .gitignore outside also works.
```



- 悔棋两次，以便将C和D融合。

```
$ git reset --soft HEAD^^
```

- 执行提交，提交说明重用C提交的提交说明。

```
$ git commit -C C
[detached HEAD 2d020b6] ignore object files.
 1 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
```

- 记住这个提交ID：2d020b6。

用里程碑是最好的记忆提交ID的方法：

```
$ git tag newbase
$ git rev-parse newbase
2d020b62034b7a433f80396118bc3f66a60f296f
```

- 执行变基操作，将E和F提交“嫁接”到newbase上。

下面的变基操命令行没有像之前的操作使用使用了参数F，而是使用分支master。所以接下来的变基操作会直接修改master分支，而无须再进行对master的重置操作。

```
$ git rebase --onto newbase E^ master
First, rewinding head to replay your work on top of it...
Applying: add hello.h
Applying: modify hello.h
```

- 看看提交日志，看到提交C和提交D都不见了，代之以融合后的提交newbase。

还可以看到最新的提交除了和HEAD的指向一致，也和master分支的指向一致。

```
$ git log --oneline --decorate -6
2495dc1 (HEAD, master) modify hello.h
6349328 add hello.h
2d020b6 (tag: newbase) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

- 当前的确已经在master分支上了，操作全部完成。

```
$ git branch
* master
```

- 清理一下，然后收工。

前面的操作中为了方便创建了标识提交的新里程碑newbase，将这个里程碑现在没有什么用处了删除吧。

```
$ git tag -d newbase
Deleted tag 'newbase' (was 2d020b6)
```

### 别忘了后台的重新布景

为了接下来的时间旅行三能够顺利开始，需要重新布景，将master分支重新置回到提交F上。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

## 时间旅行三

《回到未来-第三集》铁匠布朗博士手工打造了可以时光旅行的飞行火车，使用蒸汽作为动力。这款时间旅行火车更大，更安全，更舒适，适合一家四口外加宠物的时空旅行。与之对应本次实践也将采用“手工打造”：交互式变基。

交互式变基就是在上一节介绍的变基命令的基础上，添加了-i参数，在变基的时候进入一个交互界面。使用了交互界面的变基操作，不仅仅是自动化变基转换为手动确认那么没有技术含量，而是充满了魔法。

执行交互式变基操作，会将<since>..<till>的提交悉数罗列在一个文件中，然后自动打开

一个编辑器来编辑这个文件。可以通过修改文件的内容（删除提交，修改提交的动作关键字）实现删除提交，压缩多个提交为一个提交，更改提交的顺序，更改历史提交的提交说明。

例如下面的界面就是针对当前DEMO版本库执行的交互式变基时编辑器打开的文件：

```
pick b3af728 ignore object files.
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h

# Rebase d71ce92..b6f0b0a onto d71ce92
#
# 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 <cmd>, exec <cmd> = Run a shell command <cmd>, and stop if it fails
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

从该文件可以看出：

- 开头的四行由上到下依次对应于提交C、D、E、F。
- 前四行缺省的动作都是pick，即应用此提交。
- 参考配置文件中的注释，可以通过修改动作名称，在变基的时候执行特定操作。
- 动作reword或者简写为r，含义是变基时应用此提交，但是在提交的时候允许用户修改提交说明。

这个功能在Git 1.6.6之后开始提供，对于修改历史提交的提交说明异常方便。老版本的Git还是使用edit动作吧。

- 动作edit或者简写为e，也会应用此提交，但是会在应用时停止，提示用户使用`:command:`git commit --amend``执行提交，以便对提交进行修补。

当用户执行`:command:`git commit --amend``完成提交后，还需要执行`:command:`git rebase --continue``继续变基操作。Git会对用户进行相应地提示。

实际上用户在变基暂停状态执行修补提交可以执行多次，相当于把一个提交分解为多个提交。而且edit动作也可以实现reword的动作，因此对于老版本的Git没有reword可

用，则可以使用此动作。

- 动作squash或者简写为s，该提交会与前面的提交压缩为一个。

- 动作fixup或者简写为f，类似squash动作，但是此提交的提交说明被丢弃。

这个功能在Git 1.7.0之后开始提供，老版本的Git还是使用squash动作吧。

- 可以通过修改配置文件中这四个提交的先后顺序，进而改变最终变基后提交的先后顺序。
- 可以对相应提交对应的行执行删除操作，这样该提交就不会被应用，进而在变基后的提交中被删除。

有了对交互式变基命令的理解，就可以开始新的“回到未来”之旅了。

确认舞台已经布置完毕。

```
$ git status -s -b
## master
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

现在演出第一幕：干掉坏蛋D

- 执行交互式变基操作。

```
$ git rebase -i D^
```

- 自动用编辑器修改文件。文件内容如下：

```
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h

# Rebase b3af728..b6f0b0a onto b3af728
#
# 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 <cmd>, exec <cmd> = Run a shell command <cmd>, and stop if it fails
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

- 将第一行删除，使得上面的配置文件看起来像是这样（省略井号开始的注释）：

```
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

- 保存退出。
- 变基自动开始，即刻完成。

显示下面的内容。

```
Successfully rebased and updated refs/heads/master.
```

- 看看日志。当前分支master已经完成变基，消灭了“坏蛋D”。

```
$ git log --oneline --decorate -6
78e5133 (HEAD, master) modify hello.h
11eea7e add hello.h
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

## 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将master分支重新置回到提交F上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

布景完毕，大幕即将再次拉开。

## 现在演出第二幕：坏蛋D被感化，融入社会

- 同样执行交互式变基操作，不过因为要将C和D压缩为一个，因此变基从C的父提交开始。

```
$ git rebase -i C^
```

- 自动用编辑器修改文件。文件内容如下（忽略井号开始的注释）：

```
pick b3af728 ignore object files.
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

- 修改第二行（提交D），将动作由pick修改为squash。

修改后的内容如下：

```
pick b3af728 ignore object files.
squash 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

- 保存退出。
- 自动开始变基操作，在执行到squash命令设定的提交时，进入提交前的日志编辑状态。

显示的待编辑日志如下。很明显C和D的提交说明显示在了一起。

```
# This is a combination of 2 commits.
# The first commit's message is:

ignore object files.

# This is the 2nd commit message:

move .gitignore outside also works.
```

- 保存退出，即完成squash动作标识的提交以及后续变基操作。
- 看看提交日志，看到提交C和提交D都不见了，代之以一个融合后的提交。

```
$ git log --oneline --decorate -6
c0c2a1a (HEAD, master) modify hello.h
c1e8b66 add hello.h
db512c0 ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

- 可以看到融合C和D的提交日志实际上是两者日志的融合。在前面单行显示的日志中看不出来。

```
$ git cat-file -p HEAD^^
tree 00239a5d0daf9824a23cbf104d30af66af984e27
parent d71ce9255b3b08c718810e4e31760198dd6da243
author Jiang Xin <jiangxin@osxp.com> 1291720899 +0800
committer Jiang Xin <jiangxin@osxp.com> 1292153393 +0800

ignore object files.

move .gitignore outside also works.
```

时光旅行结束了，多么神奇的Git啊。

## 丢弃历史

历史有的时候会成为负担。例如一个人使用的版本库有一天需要作为公共版本库多人共享，最早的历史可能不希望或者没有必要继续保持存在，需要一个抛弃部分早期历史提交的精简的版本库用于和他人共享。再比如用Git做文件备份，不希望备份的版本过多导致不必要的磁盘空间占用，同样会有精简版本的需要：只保留最近的100次提交，抛弃之前的历史提交。那么应该如何操作呢？

使用交互式变基当然可以完成这样的任务，但是如果历史版本库有成百上千个，把成百上千个版本的变基动作有pick修改为fixup可真的很费事，实际上Git有更简便的方法。

现在DEMO版本库有如下的提交记录：

```
$ git log --oneline --decorate
c0c2a1a (HEAD, master) modify hello.h
c1e8b66 add hello.h
db512c0 ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

```
7161977 delete trash files. (using: git add -u)
2b31c19 (tag: old_practice) Merge commit 'acc2f69'
acc2f69 commit in detached HEAD mode.
4902dc3 does master follow this new commit?
e695606 which version checked in?
a0c641e who does commit?
9e8a761 initialized.
```

如果希望把里程碑A (c024f34) 之前的历史提交历史全部清除可以如下进行操作。

- 查看里程碑A指向的目录树。

用`A^{tree}`语法访问里程碑A对应的目录树。

```
$ git cat-file -p A^{tree}
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b      README
```

- 使用`git commit-tree`命令直接从该目录树创建提交。

```
$ echo "Commit from tree of tag A." | git commit-tree A^{tree}
8f7f94ba6a9d94ecc1c223aa4b311670599e1f86
```

- 命令`git commit-tree`的输出是一个提交的SHA1哈希值。查看这个提交。

会发现这个提交没有历史提交，可以称之为孤儿提交。

```
$ git log 8f7f94ba6a9d94ecc1c223aa4b311670599e1f86
commit 8f7f94ba6a9d94ecc1c223aa4b311670599e1f86
Author: Jiang Xin <jiangxin@ossexp.com>
Date:   Mon Dec 13 14:17:17 2010 +0800

Commit from tree of tag A.
```

- 执行变基，将master分支从里程碑到最新的提交全部迁移到刚刚生成的孤儿提交上。

```
$ git rebase --onto 8f7f94ba6a9d94ecc1c223aa4b311670599e1f86 A master
First, rewinding head to replay your work on top of it...
Applying: Hello world initialized.
Applying: ignore object files.
Applying: add hello.h
Applying: modify hello.h
```

- 查看日志看到当前master分支的历史已经精简了。

```
$ git log --oneline --decorate
2584639 (HEAD, master) modify hello.h
30fe8b3 add hello.h
4dd8a65 ignore object files.
5f2cae1 Hello world initialized.
8f7f94b Commit from tree of tag A.
```

使用图形工具查看提交历史，会看到两棵树：最上面的一棵树是刚刚通过变基抛弃了大部分历史提交的新的master分支，下面的一棵树则是变基前的提交形成的。下面的一棵树之所以还能够看到，或者说还没有从版本库中彻底清除，是因为有部分提交仍带有里程碑标签。

Graph	Short Log	Author	Author Date
	master modify hello.h	Jiang Xin...	10-12-12 PM12:11
	add hello.h	Jiang Xin...	10-12-7 PM7:39
	ignore object files.	Jiang Xin...	10-12-7 PM7:21
	Hello world initialized.	Jiang Xin...	10-12-7 PM6:33
	Commit from tree of tag A.	Jiang Xin...	10-12-13 PM2:17
	F modify hello.h	Jiang Xin...	10-12-12 PM12:11
	E add hello.h	Jiang Xin...	10-12-7 PM7:39
	D move .gitignore outside also works.	Jiang Xin...	10-12-7 PM7:34
	C ignore object files.	Jiang Xin...	10-12-7 PM7:21
	B hello 1.0 Hello world initialized.	Jiang Xin...	10-12-7 PM6:33
	A README is from welcome.txt.	Jiang Xin...	10-12-7 PM2:50
	restore file: welcome.txt	Jiang Xin...	10-12-7 PM2:32
	delete trash files. (using: git add -u)	Jiang Xin...	10-12-7 PM2:02
	[refs/stash] WIP on master: 2b31c19 Merge commit 'acc2f69'	Jiang Xin...	10-12-7 AM11:53
	index on master: 2b31c19 Merge commit 'acc2f69'	Jiang Xin...	10-12-7 AM11:53
	old practice Merge commit 'acc2f69'	Jiang Xin...	10-12-5 PM3:51
	commit in detached HEAD mode.	Jiang Xin...	10-12-5 PM3:43
	does master follow this new commit?	Jiang Xin...	10-12-4 PM12:13
	which version checked in?	Jiang Xin...	10-11-29 PM5:23
	who does commit?	Jiang Xin...	10-11-29 AM11:00
	initialized.	Jiang Xin...	10-11-28 PM12:48

## 反转提交

前面介绍的操作都涉及到对历史的修改，这对于一个人使用Git没有问题，但是如果多人协同就会有问题了。多人协同使用Git，在本地版本库做的提交会通过多人之间的交互成为他人版本库的一部分，更改历史操作只能是针对自己的版本库，而无法去修改他人的版本库，正所谓“覆水难收”。在这种情况下要想修正一个错误历史提交的正确做法是反转提交，即重新做一次新的提交，相当于错误的历史提交的反向提交，修正错误的历史提交。

Git反向提交命令是:`:command:`git revert``，下面在DEMO版本库中实践一下。注意：Subversion的用户不要想当然的和`:command:`svn revert``命令对应，这两个版本控制系统中的revert命令的功能完全不相干。

当前DEMO版本库最新的提交包含如下改动：

```
$ git show HEAD
```

```
commit 25846394defe16eab103b92efdaab5e46cc3dc22
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Sun Dec 12 12:11:00 2010 +0800

    modify hello.h

diff --git a/README b/README
index 51dbfd2..ceaf01b 100644
--- a/README
+++ b/README
@@ -1,3 +1,4 @@
Hello.
Nice to meet you.
Bye-Bye.
+Wait...
diff --git a/src/hello.h b/src/hello.h
index 0043c3b..6e482c6 100644
--- a/src/hello.h
+++ b/src/hello.h
@@ -1 +1,2 @@
/* test */
+/* end */
```

在不改变这个提交的前提下对其修改进行撤销，就需要用到git revert反转提交。

```
$ git revert HEAD
```

运行该命令相当于将HEAD提交反向再提交一次，在提交说明编辑状态下暂停，显示如下（注释行被忽略）：

```
Revert "modify hello.h"

This reverts commit 25846394defe16eab103b92efdaab5e46cc3dc22.
```

可以在编辑器中修改提交说明，提交说明编辑完毕保存退出则完成反转提交。查看提交日志可以看到新的提交相当于所撤销提交的反向提交。

```
$ git log --stat -2
commit 6e6753add1601c4efa7857ab4c5b245e0e161314
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Mon Dec 13 15:19:12 2010 +0800

    Revert "modify hello.h"

This reverts commit 25846394defe16eab103b92efdaab5e46cc3dc22.
```

```
 README      |    1 -
src/hello.h |    1 -
2 files changed, 0 insertions(+), 2 deletions(-)

commit 25846394defe16eab103b92efdaab5e46cc3dc22
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Sun Dec 12 12:11:00 2010 +0800

modify hello.h

 README      |    1 +
src/hello.h |    1 +
2 files changed, 2 insertions(+), 0 deletions(-)
```

来源: <https://github.com/gotgit/gotgit/blob/master/02-git-solo/090-back-to-future.rst>

# Git克隆

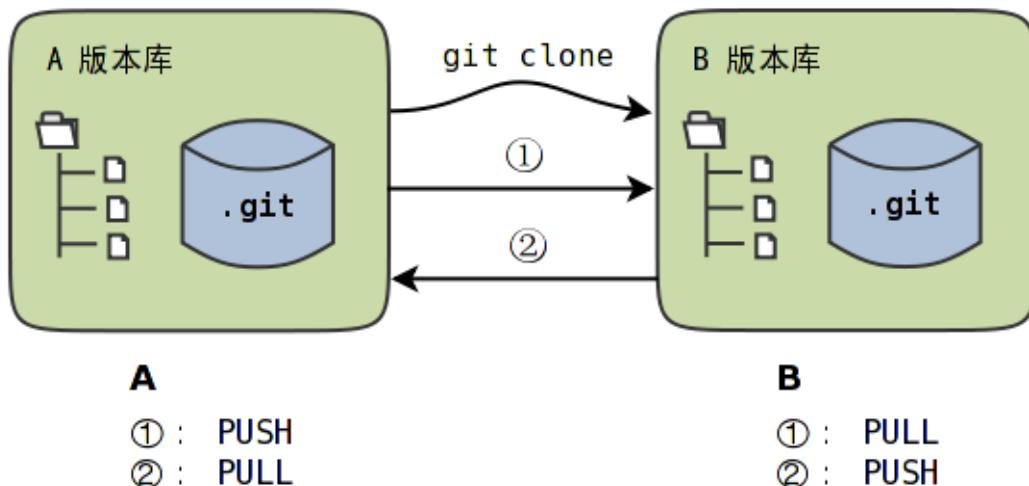
到现在为止，读者已经领略到Git的灵活性以及健壮性。Git可以通过重置随意撤销提交，可以通过变基操作更改历史，可以随意重组提交，还可以通过reflog的记录纠正错误的操作。但是再健壮的版本库设计，也抵挡不了存储介质的崩溃。还有一点就是不要忘了Git版本库是躲在工作区根目录下的:`.git`目录中，如果忘了这一点直接删除工作区，就会把版本库也同时删掉，悲剧就此发生。

“不要把鸡蛋装在一个篮子里”，是颠扑不破的安全法则。

在本章会学习到如何使用:`git clone`命令建立版本库克隆，以及如何使用:`git push`和`git pull`命令实现克隆之间的同步。

## 鸡蛋不装在一个篮子里

Git的版本库目录和工作区在一起，因此存在一损俱损的问题，即如果删除一个项目的工作区，同时也会把这个项目的版本库删除掉。一个项目仅在一个工作区中维护太危险了，如果有两个工作区就会好很多。



上图中一个项目使用了两个版本库进行维护，两个版本库之间通过拉回（PULL）和/或推送（PUSH）操作实现同步。

- 版本库A通过克隆操作创建克隆版本库B。
- 版本库A可以通过推送（PUSH）操作，将新提交传递给版本库B；
- 版本库A可以通过拉回（PULL）操作，将版本库B中的新提交拉回到自身（A）。
- 版本库B可以通过拉回（PULL）操作，将版本库A中的新提交拉回到自身（B）。
- 版本库B可以通过推送（PUSH）操作，将新提交传递给版本库A；

Git使用`git clone`命令实现版本库克隆，主要有如下三种用法：

```
用法1: git clone <repository> <directory>
用法2: git clone --bare    <repository> <repository.git>
用法3: git clone --mirror <repository> <repository.git>
```

这三种用法的区别如下:

- 用法1将`<repository>`指向的版本库创建一个克隆到`:file:`<directory>``目录。目录`:file:`<directory>``相当于克隆版本库的工作区，文件都会检出，版本库位于工作区下的`:file:`.git``目录中。
- 用法2和用法3创建的克隆版本库都不含工作区，直接就是版本库的内容，这样的版本库称为裸版本库。一般约定俗成裸版本库的目录名以`:file:`.git``为后缀，所以上面示例中将克隆出来的裸版本库目录名写做`:file:`<repository.git>``。
- 用法3区别于用法2之处在于用法3克隆出来的裸版本对上游版本库进行了注册，这样可以在裸版本库中使用`:command:`git fetch``命令和上游版本库进行持续同步。
- 用法3只在 1.6.0 或更新版本的Git才提供。

Git的PUSH和PULL命令的用法相似，使用下面的语法:

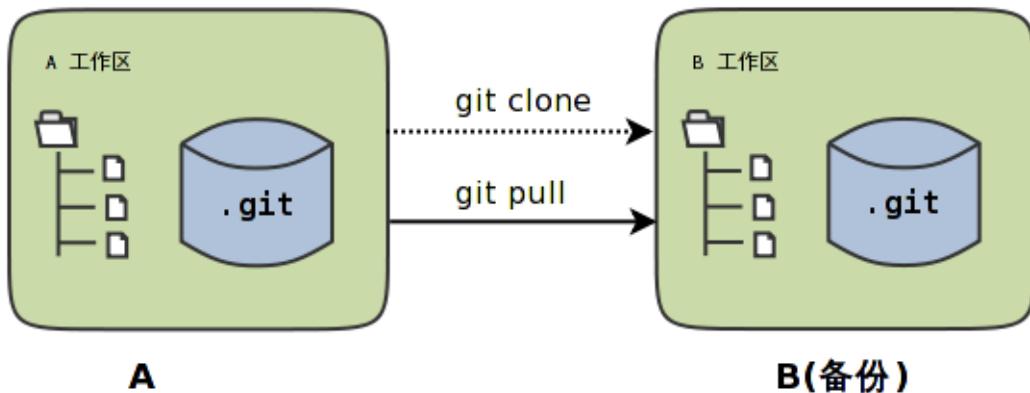
```
git push [<remote-repos> [<refspec>]]
git pull [<remote-repos> [<refspec>]]
```

其中方括号的含义是参数可以省略，`<remote-repos>`是远程版本库的地址或名称，`<refspec>`是引用表达式，暂时理解为引用即可。在后面的章节再具体介绍PUSH和PULL命令的细节。

下面就通过不同的Git命令组合，掌握版本库克隆和镜像的技巧。

## 对等工作区

不使用`--bare`或者`--mirror`创建出来的克隆包含工作区，这样就会产生两个包含工作区的版本库。这两个版本库是对等的，如下图。



这两个工作区本质上没有区别，但是往往提交是在一个版本（A）中进行的，另外一个（B）作为备份。对于这种对等工作区模式，版本库的同步只有一种可行的操作模式，就是在备份库（B）执行 git pull 命令从源版本库（A）拉回新的提交实现版本库同步。为什么不能从版本库A向版本库B执行 git push 的推送操作呢？看看下面的操作。

执行克隆命令，将版本库`:file:`/path/to/my/workspace/demo``克隆到`:file:`/path/to/my/workspace/demo-backup``。

```
$ git clone /path/to/my/workspace/demo /path/to/my/workspace/demo-backup
Cloning into /path/to/my/workspace/demo-backup...
done.
```

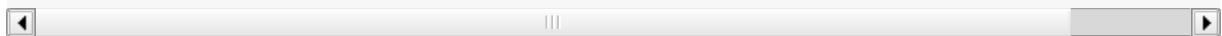
进入 demo 版本库，生成一些测试提交（使用`--allow-empty`参数可以生成空提交）。

```
$ cd /path/to/my/workspace/demo/
$ git commit --allow-empty -m "sync test 1"
[master 790e72a] sync test 1
$ git commit --allow-empty -m "sync test 2"
[master f86b7bf] sync test 2
```

能够在 demo 版本库向 demo-backup 版本库执行PUSH操作么？执行一下`:command:`git push``看一看。

```
$ git push /path/to/my/workspace/demo-backup
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 274 bytes, done.
Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
remote: error: refusing to update checked out branch: refs/heads/master
remote: error: By default, updating the current branch in a non-bare repository
remote: error: is denied, because it will make the index and work tree incons
```

```
remote: error: with what you pushed, and will require 'git reset --hard' to m
remote: error: the work tree to HEAD.
remote: error:
remote: error: You can set 'receive.denyCurrentBranch' configuration variable
remote: error: 'ignore' or 'warn' in the remote repository to allow pushing i
remote: error: its current branch; however, this is not recommended unless yo
remote: error: arranged to update its work tree to match what you pushed in s
remote: error: other way.
remote: error:
remote: error: To squelch this message and still keep the default behaviour,
remote: error: 'receive.denyCurrentBranch' configuration variable to 'refuse'
To /path/to/my/workspace/demo-backup
! [remote rejected] master -> master (branch is currently checked out)
error: failed to push some refs to '/path/to/my/workspace/demo-backup'
```



翻译成中文：

```
$ git push /path/to/my/workspace/demo-backup
```

...

对方说：错了：

拒绝更新已检出的分支 refs/heads/master 。  
缺省更新非裸版本库的当前分支是不被允许的，因为这将会导致  
暂存区和工作区与您推送至版本库的新提交不一致。这太古怪了。

如果您一意孤行，也不是不允许，但是您需要为我设置如下参数：

```
receive.denyCurrentBranch = ignore|warn
```

到 /path/to/my/workspace/demo-backup

```
! [对方拒绝] master -> master (分支当前已检出)
错误：部分引用的推送失败了，至 '/path/to/my/workspace/demo-backup'
```

从错误输出可以看出，虽然可以改变Git的缺省行为，允许向工作区推送已经检出的分支，但是这么做实在不高明。

为了实现同步，需要进入到备份版本库中，执行：[command: `git pull` 命令。](#)

```
$ git pull
From /path/to/my/workspace/demo
 6e6753a..f86b7bf  master      -> origin/master
Updating 6e6753a..f86b7bf
Fast-forward
```

在 demo-backup 版本库中查看提交日志，可以看到在 demo 版本库中的新提交已经被拉回

到 demo-backup 版本库中。

```
$ git log --oneline -2  
f86b7bf sync test 2  
790e72a sync test 1
```

为什么执行 `git pull` 拉回命令没有像执行 `git push` 命令那样提供那么多的参数呢？

这是因为在执行:[command:](#) `git clone` 操作后，克隆出来的demo-backup版本库中对源版本库（上游版本库）进行了注册，所以当在 demo-backup 版本库执行拉回操作，无须设置上游版本库的地址。

在 demo-backup 版本库中可以使用下面的命令查看对上游版本库的注册信息：

```
$ cd /path/to/my/workspace/demo-backup  
$ git remote -v  
origin  /path/to/my/workspace/demo (fetch)  
origin  /path/to/my/workspace/demo (push)
```

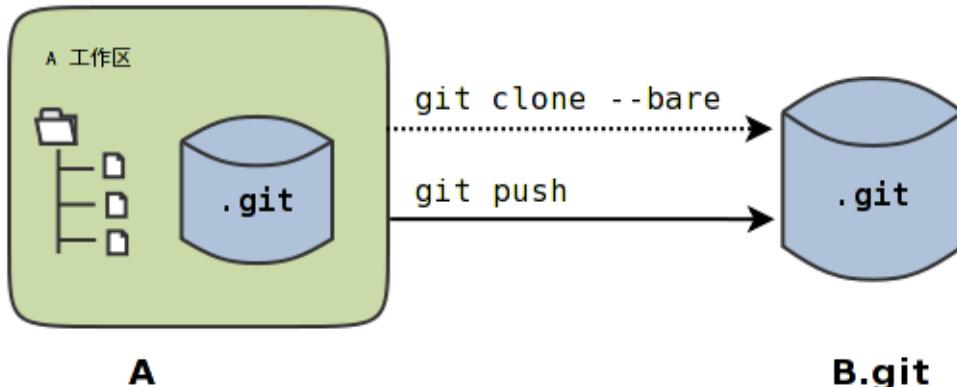
实际注册上游远程版本库的奥秘都在Git的配置文件中（略去无关的行）：

```
$ cat /path/to/my/workspace/demo-backup/.git/config  
...  
[remote "origin"]  
  fetch = +refs/heads/*:refs/remotes/origin/*  
  url = /path/to/my/workspace/demo  
[branch "master"]  
  remote = origin  
  merge = refs/heads/master
```

关于配置文件[remote]小节和[branch]小节的奥秘在后面的章节予以介绍。

## 克隆生成裸版本库

上一节在对等工作区模式下，工作区之间执行推送，可能会引发大段的错误输出，如果采用裸版本库则没有相应的问题。这是因为裸版本库没有工作区。没有工作区还有一个好处就是空间占用会更小。



使用`--bare`参数克隆demo版本库到`:file:`/path/to/repos/demo.git``，然后就可以从demo版本库向克隆的裸版本库执行推送操作了。（为了说明方便，使用了`:file:`/path/to/repos/``作为Git裸版本的根路径，在后面的章节中这个目录也作为Git服务器端版本库的根路径。可以在磁盘中以root账户创建该路径并设置正确的权限。）

```
$ git clone --bare /path/to/my/workspace/demo /path/to/repos/demo.git  
Cloning into bare repository /path/to/repos/demo.git...  
done.
```

克隆出来的`:file:`/path/to/repos/demo.git``目录就是版本库目录，不含工作区。

- 看看`:file:`/path/to/repos/demo.git``目录的内容。

```
$ ls -F /path/to/repos/demo.git  
branches/ config description HEAD hooks/ info/ objects/ packed-ref
```

- 还可以看到`demo.git`版本库`core.bare`的配置为`true`。

```
$ git --git-dir=/path/to/repos/demo.git config core.bare  
true
```

进入`demo`版本库，生成一些测试提交。

```
$ cd /path/to/my/workspace/demo/  
$ git commit --allow-empty -m "sync test 3"  
[master d4b42b7] sync test 3  
$ git commit --allow-empty -m "sync test 4"  
[master 0285742] sync test 4
```

在`demo`版本库向`demo-backup`版本库执行PUSH操作，还会有错误么？

- 不带参数执行:command:`git push`，因为未设定上游远程版本库，因此会报错：

```
$ git push  
fatal: No destination configured to push to.
```

- 在执行:command:`git push`时使用:file:`/path/to/repos/demo.git`作为参数。

推送成功。

```
$ git push /path/to/repos/demo.git  
Counting objects: 2, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 275 bytes, done.  
Total 2 (delta 1), reused 0 (delta 0)  
Unpacking objects: 100% (2/2), done.  
To /path/to/repos/demo.git  
f86b7bf..0285742 master -> master
```

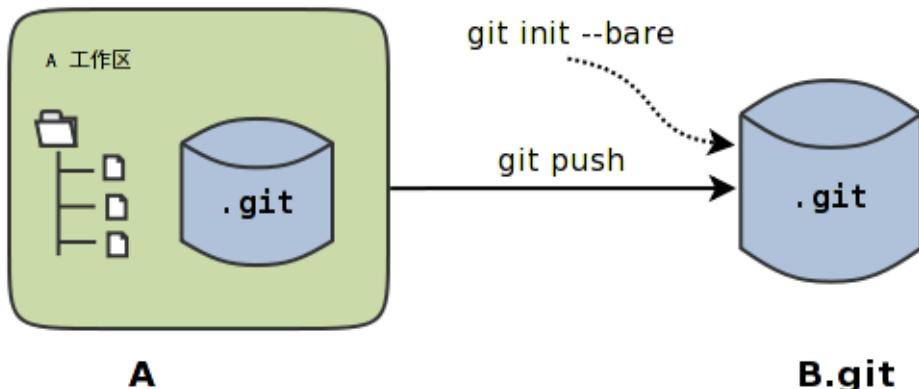
看看:file:`demo.git`版本库，是否已经完成了同步？

```
$ git log --oneline -2  
0285742 sync test 4  
d4b42b7 sync test 3
```

这种方式实现版本库本地镜像显然是更好的方法，因为可以直接在工作区修改、提交，然后执行:command:`git push`命令实现推送。稍有一点遗憾的是推送命令还需要加上裸版本库的路径。这个遗憾在后面介绍远程版本库的章节会给出解决方案。

## 创建生成裸版本库

裸版本库不但可以通过克隆的方式创建，还可以通过:command:`git init`命令以初始化的方式创建。之后的同步方式和上一节大同小异。



命令`:command:`git init``在“Git初始化”一章就已经用到了，是用于初始化一个版本库的。之前执行`:command:`git init``命令初始化的版本库是带工作区的，如何以裸版本库的方式初始化一个版本库呢？奥秘就在于`--bare`参数。

下面的命令会创建一个空的裸版本库于目录`:file:`/path/to/repos/demo-init.git``中。

```
$ git init --bare /path/to/repos/demo-init.git
Initialized empty Git repository in /path/to/repos/demo-init.git/
```

创建的果真是裸版本库么？

- 看看`:file:`/path/to/repos/demo-init.git``下的内容：

```
$ ls -F /path/to/repos/demo-init.git
branches/ config  description  HEAD  hooks/  info/  objects/  refs/
```

- 看看这个版本库的配置`core.bare`的值：

```
$ git --git-dir=/path/to/repos/demo-init.git config core.bare
true
```

可是空版本库没有内容啊，那就执行PUSH操作为其创建内容呗。

```
$ cd /path/to/my/workspace/demo
$ git push /path/to/repos/demo-init.git
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
fatal: The remote end hung up unexpectedly
error: failed to push some refs to '/path/to/repos/demo-init.git'
```

为什么出错了？翻译一下错误输出。

```
$ cd /path/to/my/workspace/demo
$ git push /path/to/repos/demo-init.git
没有指定要推送的引用，而且两个版本库也没有共同的引用。
所以什么也没有做。
可能您需要提供要推送的分支名，如 'master'。
严重错误：远程操作意外终止
错误：部分引用推送失败，至 '/path/to/repos/demo-init.git'
```

关于这个问题详细说明要在后面的章节介绍，这里先说一个省略版：因为：`/path/to/repos/demo-init.git` 版本库刚刚初始化完成，还没有任何提交更不要说分支了。当执行：`git push` 命令时，如果没有设定推送的分支，而且当前分支也没有注册到远程某个分支，将检查远程分支是否有和本地相同的分支名（如 `master`），如果有，则推送，否则报错。

所以需要把：`git push` 命令写的再完整一些。像下面这样操作，就可以完成向空的裸版本库的推送。

```
$ git push /path/to/repos/demo-init.git master:master
Counting objects: 26, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (26/26), 2.49 KiB, done.
Total 26 (delta 8), reused 0 (delta 0)
Unpacking objects: 100% (26/26), done.
To /path/to/repos/demo-init.git
 * [new branch]      master -> master
```

上面的：`git push` 命令也可以简写为：`git push /path/to/repos/demo-init.git master`。

推送成功了么？看看：`demo-init.git` 版本库中的提交。

```
$ git --git-dir=/path/to/repos/demo-init.git log --oneline -2
0285742 sync test 4
d4b42b7 sync test 3
```

好了继续在 `demo` 中执行几次提交。

```
$ cd /path/to/my/workspace/demo/
$ git commit --allow-empty -m "sync test 5"
[master 424aa67] sync test 5
$ git commit --allow-empty -m "sync test 6"
[master 70a5aa7] sync test 6
```

然后再向:`file: `demo-init.git``推送。注意这次使用的命令。

```
$ git push /path/to/repos/demo-init.git  
Counting objects: 2, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 273 bytes, done.  
Total 2 (delta 1), reused 0 (delta 0)  
Unpacking objects: 100% (2/2), done.  
To /path/to/repos/demo-init.git  
    0285742..70a5aa7  master -> master
```

为什么这次使用:`command: `git push``命令后面没有跟上分支名呢？这是因为远程版本库(`demo-init.git`)中已经不再是空版本库了，而且有名为`master`的分支。

通过下面的命令可以查看远程版本库的分支。

```
$ git ls-remote /path/to/repos/demo-init.git  
70a5aa7a7469076fd435a9e4f89c4657ba603ced      HEAD  
70a5aa7a7469076fd435a9e4f89c4657ba603ced      refs/heads/master
```

至此相信读者已经能够把鸡蛋放在不同的篮子中了，也对Git更加的喜爱了吧。

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/100-git-clone.rst>

# Git检出

在上一章学习了重置命令（:command:`git reset`）。重置命令的一个用途就是修改引用（如master）的游标。实际上在执行重置命令的时候没有使用任何参数对所要重置的分支名进行设置，这是因为重置命名实际上所针对的是头指针HEAD。之所以没有改变HEAD的内容是因为HEAD指向了一个引用refs/heads/master，所以重置命令体现为分支“游标”的变更，HEAD本身一直指向的是refs/heads/master，并没有在重置时改变。

如果HEAD的内容不能改变而一直都指向master分支，那么Git如此精妙的分支设计岂不浪费？如果HEAD要改变该如何改变呢？本章将学习检出命令（:command:`git checkout`），该命令的实质就是修改HEAD本身的指向，该命令不会影响分支“游标”（如master）。

## HEAD的重置即检出

HEAD可以理解为“头指针”，是当前工作区的“基础版本”，当执行提交时，HEAD指向的提交将作为新提交的父提交。看看当前HEAD的指向。

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

可以看出HEAD指向了分支 master。此时执行:command:`git branch`会看到当前处于 master分支。

```
$ git branch -v
* master 4902dc3 does master follow this new commit?
```

现在使用:command:`git checkout`命令检出该ID的父提交，看看会怎样。

```
$ git checkout 4902dc3^
Note: checking out '4902dc3^'.

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 e695606... which version checked in?
```

出现了大段的输出！翻译一下，Git肯定又是在提醒我们了。

```
$ git checkout 4902dc3^  
注意：正检出 '4902dc3^'.
```

您现在处于‘分离头指针’状态。您可以检查、测试和提交，而不影响任何分支。  
通过执行另外的一个 `checkout` 检出指令会丢弃在此状态下的修改和提交。

如果想保留在此状态下的修改和提交，使用 `-b` 参数调用 `checkout` 检出指令以  
创建新的跟踪分支。如：

```
git checkout -b new_branch_name
```

头指针现在指向 `e695606...` 提交说明为：`which version checked in?`

什么叫做“分离头指针”状态？查看一下此时HEAD的内容就明白了。

```
$ cat .git/HEAD  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

原来“分离头指针”状态指的就是HEAD头指针指向了一个具体的提交ID，而不是一个引用  
(分支)。

查看最新提交的reflog也可以看到当针对提交执行：`git checkout` 命令时，HEAD  
头指针被更改了：由指向master分支变成了指向一个提交ID。

```
$ git reflog -1  
e695606 HEAD@{0}: checkout: moving from master to 4902dc3^
```

注意上面的reflog是HEAD头指针的变迁记录，而非master分支。

查看一下HEAD和master对应的提交ID，会发现在它们指向的不一样。

```
$ git rev-parse HEAD master  
e695606fc5e31b2ff9038a48a3d363f4c21a3d86  
4902dc375672fbf52a226e0354100b75d4fe31e3
```

前一个是HEAD头指针的指向，后一个是master分支的指向。而且还可以看到执  
行：`git checkout` 命令并不像：`git reset` 命令，分支 (master) 的指  
向并没有改变，仍旧指向原有的提交ID。

现在版本库的HEAD是基于e695606提交的。再做一次提交，HEAD会如何变化呢？

- 先做一次修改：创建一个新文件：`file: `detached-commit.txt``，添加到暂存区中。

```
$ touch detached-commit.txt  
$ git add detached-commit.txt
```

- 看一下状态，会发现其中有：“当前不处于任何分支”的字样，显然这是因为HEAD处于“分离头指针”模式。

```
$ git status  
# Not currently on any branch.  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       new file:   detached-commit.txt  
#
```

- 执行提交。在提交输出中也会出现[detached HEAD ...]的标识，也是对用户的警示。

```
$ git commit -m "commit in detached HEAD mode."  
[detached HEAD acc2f69] commit in detached HEAD mode.  
 0 files changed, 0 insertions(+), 0 deletions(-)  
  create mode 100644 detached-commit.txt
```

- 此时头指针指向了新的提交。

```
$ cat .git/HEAD  
acc2f69cf6f0ae346732382c819080df75bb2191
```

- 再查看一下日志会发现新的提交是建立在之前的提交基础上的。

```
$ git log --graph --pretty=oneline  
* acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode.  
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?  
* a0c641e92b10d8bcc1ed1bf84ca80340fdefee6 who does commit?  
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

记下新的提交ID（acc2f69），然后以master分支名作为参数执行：`command: `git checkout``命令，会切换到master分支上。

- 切换到master分支。没有之前大段的文字警告。

```
$ git checkout master  
Previous HEAD position was acc2f69... commit in detached HEAD mode.  
Switched to branch 'master'
```

- 因为HEAD头指针重新指向了分支，而不是处于“断头模式”（分离头指针模式）。

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

- 切换之后，之前本地建立的新文件：[file: `detached-commit.txt`](#) 不见了。

```
$ ls  
new-commit.txt  welcome.txt
```

- 切换之后，刚才的提交日志也不见了。

```
$ git log --graph --pretty=oneline  
* 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new co  
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?  
* a0c641e92b10d8bcc1ed1bf84ca80340fdefee6 who does commit?  
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

刚才的提交在版本库的对象库中还存在么？看看刚才记下的提交ID。

```
$ git show acc2f69  
commit acc2f69cf6f0ae346732382c819080df75bb2191  
Author: Jiang Xin <jiangxin@ossp.com>  
Date:   Sun Dec 5 15:43:24 2010 +0800  
  
        commit in detached HEAD mode.  
  
diff --git a/detached-commit.txt b/detached-commit.txt  
new file mode 100644  
index 000000..e69de29
```

可以看出这个提交现在仍在版本库中。由于这个提交没有被任何分支跟踪到，因此并不能保证这个提交会永久存在。实际上当reflog中含有该提交的日志过期后，这个提交随时都会从版本库中彻底清除。

## 挽救分离头指针

在“分离头指针”模式下进行的测试提交除了使用提交ID（acc2f69）访问之外，不能通过master分支或其他引用访问到。如果这个提交是master分支所需要的，那么该如何处理呢？如果使用上一章介绍的`:command:`git reset``命令，的确可以将master分支重置到该测试提交acc2f69，但是如果那样就会丢掉master分支原先的提交4902dc3。使用合并操作`(:command:`git merge`)`可以实现两者的兼顾。

下面的操作会将提交acc2f69合并到master分支中来。

- 确认当前处于master分支。

```
$ git branch -v
* master 4902dc3 does master follow this new commit?
```

- 执行合并操作，将acc2f69提交合并到当前分支。

```
$ git merge acc2f69
Merge made by recursive.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 detached-commit.txt
```

- 工作区中多了一个`:file:`detached-commit.txt``文件。

```
$ ls
detached-commit.txt  new-commit.txt  welcome.txt
```

- 查看日志，会看到不一样的分支图。即在e695606提交开始出现了开发分支，而分支在最新的2b31c19提交发生了合并。

```
$ git log --graph --pretty=oneline
*   2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
|\ \
| * acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode
| * | 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new
|/
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
* a0c641e92b10d8bcc1ed1bf84ca80340fdefee6 who does commit?
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

- 仔细看看最新提交，会看到这个提交有两个父提交。这就是合并的奥秘。

```
$ git cat-file -p HEAD
tree ab676f92936000457b01507e04f4058e855d4df0
parent 4902dc375672fbf52a226e0354100b75d4fe31e3
parent acc2f69cf6f0ae346732382c819080df75bb2191
author Jiang Xin <jiangxin@osssxp.com> 1291535485 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1291535485 +0800

Merge commit 'acc2f69'
```

## 深入了解:command:`git checkout`命令

检出命令 (:command:`git checkout`) 是Git最常用的命令之一，同样也很危险，因为这条命令会重写工作区。

用法一: `git checkout [-q] [<commit>] [--] <paths>...`

用法二: `git checkout [<branch>]`

用法三: `git checkout [-m] [[-b|--orphan] <new_branch>] [<start_point>]`

上面列出的第一种用法和第二种用法的区别在于，第一种用法在命令中包含路径:file:`<paths>`。为了避免路径和引用（或者提交ID）同名而冲突，可以在:file:`<paths>`前用两个连续的短线（减号）作为分隔。

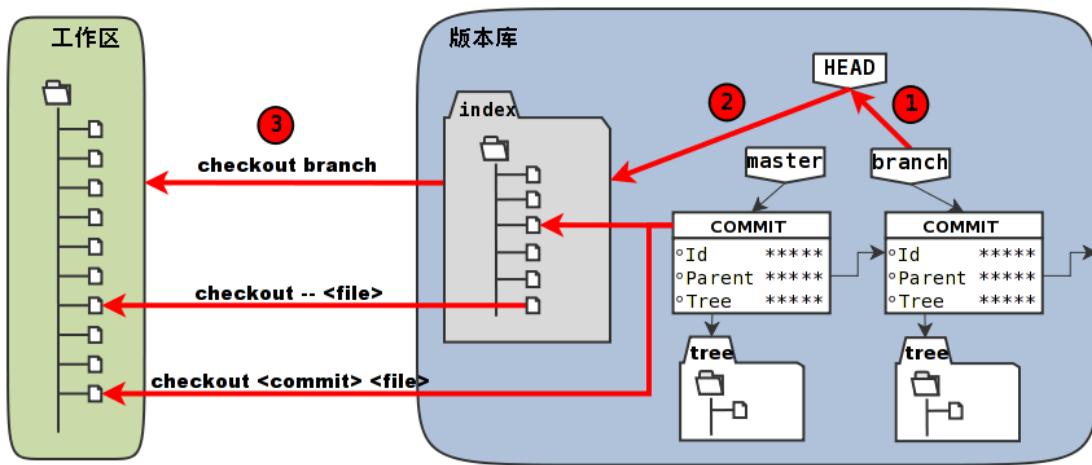
第一种用法的<commit>是可选项，如果省略则相当于从暂存区（index）进行检出。这和上一章的重置命令大不相同：重置的默认值是 HEAD，而检出的默认值是暂存区。因此重置一般用于重置暂存区（除非使用--hard参数，否则不重置工作区），而检出命令主要是覆盖工作区（如果<commit>不省略，也会替换暂存区中相应的文件）。

第一种用法（包含了路径:file:`<paths>`的用法）不会改变HEAD头指针，主要是用于指定版本的文件覆盖工作区中对应的文件。如果省略<commit>，会拿暂存区的文件覆盖工作区的文件，否则用指定提交中的文件覆盖暂存区和工作区中对应的文件。

第二种用法（不使用路径:file:`<paths>`的用法）则会改变HEAD头指针。之所以后面的参数写作<branch>，是因为只有HEAD切换到一个分支才可以对提交进行跟踪，否则仍然会进入“分离头指针”的状态。在“分离头指针”状态下的提交不能被引用关联到而可能会丢失。所以用法二最主要的作用就是切换到分支。如果省略<branch>则相当于对工作区进行状态检查。

第三种用法主要是创建和切换到新的分支（<new\_branch>），新的分支从<start\_point>指定的提交开始创建。新分支和我们熟悉的master分支没有什么实质的不同，都是在refs/heads命名空间下的引用。关于分支和:command:`git checkout`命令的这个用法会在后面的章节做具体的介绍。

下面的版本库模型图描述了:command:`git checkout` 实际完成的操作。



下面通过一些示例，具体的看一下检出命令的不同用法。

- 命令: :command:`git checkout branch`

检出branch分支。要完成如图的三个步骤，更新HEAD以指向branch分支，以branch指向的树更新暂存区和工作区。

- 命令: :command:`git checkout`

汇总显示工作区、暂存区与HEAD的差异。

- 命令: :command:`git checkout HEAD`

同上。

- 命令: :command:`git checkout -- filename`

用暂存区中:file:`filename`文件来覆盖工作区中的:file:`filename`文件。相当于取消自上次执行:command:`git add filename`以来（如果执行过）本地的修改。

这个命令很危险，因为对于本地的修改会悄无声息的覆盖，毫不留情。

- 命令: :command:`git checkout branch -- filename`

维持HEAD的指向不变。将branch所指向的提交中的:file:`filename`替换暂存区和工作区中相应的文件。注意会将暂存区和工作区中的:file:`filename`文件直接覆盖。

- 命令: :command:`git checkout -- .` 或写做 `git checkout .`

注意: :command:`git checkout` 命令后的参数为一个点（“.”）。这条命令最危险！会取消所有本地的修改（相对于暂存区）。相当于将暂存区的所有文件直接覆盖

本地文件，不给用户任何确认的机会！

来源：<https://github.com/gotgit/gotgit/blob/master/02-git-solo/050-git-checkout.rst>

# Git库管理

版本库管理？那不是管理员要干的事情么，怎么放在“Git独奏”这一部分了？

没有错，这是因为对于Git，每个用户都是自己版本库的管理员，所以在“Git独奏”的最后一章，来谈一谈Git版本库管理的问题。如果下面的问题您没有遇到或者不感兴趣，读者大可以放心的跳过这一章。

- 从网上克隆来的版本库，为什么对象库中找不到对象文件？而且引用目录里也看不到所有的引用文件？
- 不小心添加了一个大文件到Git库中，用重置命令丢弃了包含大文件的提交，可是版本库不见小，大文件仍在对象库中。
- 本地版本库的对象库里文件越来越多，这可能导致Git性能的降低。

## 对象和引用哪里去了？

从GitHub上克隆一个示例版本库，这个版本库在“历史穿梭”一章就已经克隆过一次了，现在要重新克隆一份。为了和原来的克隆相区别，克隆到另外的目录。执行下面的命令。

```
$ cd /path/to/my/workspace/
$ git clone git://github.com/ossxp-com/gitdemo-commit-tree.git i-am-admin
Cloning into i-am-admin...
remote: Counting objects: 65, done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 65 (delta 8), reused 0 (delta 0)
Receiving objects: 100% (65/65), 78.14 KiB | 42 KiB/s, done.
Resolving deltas: 100% (8/8), done.
```

进入克隆的版本库，使用`:command:`git show-ref``命令看看所含的引用。

```
$ cd /path/to/my/workspace/i-am-admin
$ git show-ref
6652a0dce6a5067732c00ef0a220810a7230655e refs/heads/master
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/HEAD
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/master
c9b03a208288aebdbfe8d84aeb984952a16da3f2 refs/tags/A
1a87782f8853c6e11aacba463af04b4fa8565713 refs/tags/B
9f8b51bc7dd98f7501ade526dd78c55ee4abb75f refs/tags/C
887113dc095238a0f4661400d33ea570e5edc37c refs/tags/D
6decd0ad3201ddb3f5b37c201387511059ac120c refs/tags/E
70cab20f099e0af3f870956a3fb6bda50a17864f refs/tags/F
96793e37c7f1c7b2ddf69b4c1e252763c11a711f refs/tags/G
```

```
476e74549047e2c5fbd616287a499cc6f07ebde0 refs/tags/H  
76945a15543c49735634d58169b349301d65524d refs/tags/I  
f199c10c3f1a54fa3f9542902b25b49d58efb35b refs/tags/J
```

其中以`refs/heads/`开头的是分支；以`refs/remotes/`开头的是远程版本库分支在本地的映射，会在后面章节介绍；以`refs/tags/`开头的是里程碑。按照之前的经验，在`:file:`.git/refs``目录下应该有这些引用所对应的文件才是。看看都在么？

```
$ find .git/refs/ -type f  
.git/refs/remotes/origin/HEAD  
.git/refs/heads/master
```

为什么才有两个文件？实际上当运行下面的命令后，引用目录下的文件会更少：

```
$ git pack-refs --all  
$ find .git/refs/ -type f  
.git/refs/remotes/origin/HEAD
```

那么本应该出现在`:file:`.git/refs/``目录下的引用文件都到哪里去了呢？答案是这些文件被打包了，放到一个文本文件`:file:`.git/packed-refs``中了。查看一下这个文件中的内容。

```
$ head -5 .git/packed-refs  
# pack-refs with: peeled  
6652a0dce6a5067732c00ef0a220810a7230655e refs/heads/master  
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/master  
c9b03a208288aebdbfe8d84aeb984952a16da3f2 refs/tags/A  
^81993234fc12a325d303eccea20f6fd629412712
```

再来看看Git的对象（commit、blob、tree、tag）在对象库中的存储。通过下面的命令，会发现对象库也不是原来熟悉的模样了。

```
$ find .git/objects/ -type f  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

对象库中只有两个文件，本应该一个一个独立保存的对象都不见了。读者应该能够猜到，所有的对象文件都被打包到这两个文件中了，其中以`:file:`.pack``结尾的文件是打包文件，以`:file:`.idx``结尾的是索引文件。打包文件和对应的索引文件只是扩展名不同，都保存于`:file:`.git/objects/pack/``目录下。Git对于以SHA1哈希值作为目录名和文件名保存的对象有一个术语，称为松散对象。松散对象打包后会提高访问效率，而且不同的对象

可以通过增量存储节省磁盘空间。

可以通过Git一个底层命令可以查看索引中包含的对象：

```
$ git show-index < .git/objects/pack/pack-*.idx | head -5
661 0cd7f2ea245d90d414e502467ac749f36aa32cc4 (0793420b)
63020 1026d9416d6fc8d34e1edfb2bc58adb8aa5a6763 (ed77ff72)
3936 15328fc6961390b4b10895f39bb042021edd07ea (13fb79ef)
3768 1a588ca36e25f58fbeae421c36d2c39e38e991ef (86e3b0bd)
2022 1a87782f8853c6e11aacba463af04b4fa8565713 (e269ed74)
```

为什么克隆远程版本库就可以产生对象库打包以及引用打包的效果呢？这是因为克隆远程版本库时，使用了“智能”的通讯协议，远程Git服务器将对象打包后传输给本地，形成本地版本库的对象库中的一个包含所有对象的包以及索引文件。无疑这样的传输方式——按需传输、打包传输，效率最高。

克隆之后的版本库在日常的提交中，产生的新的对象仍旧以松散对象存在，而不是以打包的形式，日积月累会在本地版本库的对象库中形成大量的松散文件。松散对象只是进行了压缩，而没有（打包文件才有的）增量存储的功能，会浪费磁盘空间，也会降低访问效率。更为严重的一些非正式的临时对象（暂存区操作中产生的临时对象）也以松散对象的形式保存在对象库中，造成磁盘空间的浪费。下一节就着手处理临时对象的问题。

## 暂存区操作引入的临时对象

暂存区操作有可能在对象库中产生临时对象，例如文件反复的修改和反复的向暂存区添加，或者添加到暂存区后不提交甚至直接撤销，就会产生垃圾数据占用磁盘空间。为了说明临时对象的问题，需要准备一个大的压缩文件，10MB即可。

在Linux上与内核匹配的:`:file:`initrd``文件（内核启动加载的内存盘）就是一个大的压缩文件，可以用于此节的示例。将大的压缩文件放在版本库外的一个位置上，因为这个文件会多次用到。

```
$ cp /boot/initrd.img-2.6.32-5-amd64 /tmp/bigfile
$ du -sh bigfile
11M    bigfile
```

将这个大的压缩文件复制到工作区中，拷贝两份。

```
$ cd /path/to/my/workspace/i-am-admin
$ cp /tmp/bigfile bigfile
$ cp /tmp/bigfile bigfile.dup
```

然后将工作区中两个内容完全一样的大文件加入暂存区。

```
$ git add bigfile bigfile.dup
```

查看一下磁盘空间占用：

- 工作区连同版本库共占用33MB。

```
$ du -sh .
33M   .
```

- 其中版本库只占用了11MB。版本库空间占用是工作区的一半。

如果再有谁说版本库空间占用一定比工作区大，可以用这个例子回击他。

```
$ du -sh .git/
11M   .git/
```

看看版本库中对象库内的文件，会发现多出了一个松散对象。之所以添加两个文件而只有一个松散对象，是因为Git对于文件的保存是将内容保存为blob对象中，和文件名无关，相同内容的不同文件会共享同一个blob对象。

```
$ find .git/objects/ -type f
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

如果不提交，想将文件撤出暂存区，则进行如下操作。

- 当前暂存区的状态。

```
$ git status -s
A  bigfile
A  bigfile.dup
```

- 将添加的文件撤出暂存区。

```
$ git reset HEAD
```

- 通过查看状态，看到文件被撤出暂存区了。

```
$ git status -s  
?? bigfile  
?? bigfile.dup
```

文件撤出暂存区后，在对象库中产生的blob松散对象仍然存在，通过查看版本库的磁盘占用就可以看出来。

```
$ du -sh .git/  
11M      .git/
```

Git提供了:command:`git fsck`命令，可以查看到版本库中包含的没有被任何引用关联松散对象。

```
$ git fsck  
dangling blob 2ebcd92d0dda2bad50c775dc662c6cb700477aff
```

标识为dangling的对象就是没有被任何引用直接或者间接关联到的对象。这个对象就是前面通过暂存区操作引入的大文件的内容。如何将这个文件从版本库中彻底删除呢？Git提供了一个清理的命令：

```
$ git prune
```

用:command:`git prune`清理之后，会发现：

- 用:command:`git fsck`查看，没有未被关联到的松散对象。

```
$ git fsck
```

- 版本库的空间占用也小了10MB，证明大的临时对象文件已经从版本库中删除了。

```
$ du -sh .git/  
236K      .git/
```

## 重置操作引入的对象

上一节用:command:`git prune`命令清除暂存区操作时引入的临时对象，但是如果是用重置命令抛弃的提交和文件就不会轻易的被清除。下面用同样的大文件提交到版本库中试验一下。

```
$ cd /path/to/my/workspace/i-am-admin  
$ cp /tmp/bigfile bigfile  
$ cp /tmp/bigfile bigfile.dup
```

将这两个大文件提交到版本库中。

- 添加到暂存区。

```
$ git add bigfile bigfile.dup
```

- 提交到版本库。

```
$ git commit -m "add bigfiles."  
[master 51519c7] add bigfiles.  
 2 files changed, 0 insertions(+), 0 deletions(-)  
  create mode 100644 bigfile  
  create mode 100644 bigfile.dup
```

- 查看版本库的空间占用。

```
$ du -sh .git/  
11M      .git/
```

做一个重置操作，抛弃刚刚针对两个大文件做的提交。

```
$ git reset --hard HEAD^
```

重置之后，看看版本库的变化。

- 版本库的空间占用没有变化，还是那么“庞大”。

```
$ du -sh .git/  
11M      .git/
```

- 查看对象库，看到三个松散对象。

```
$ find .git/objects/ -type f  
.git/objects/info/packs  
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff  
.git/objects/d9/38dee8fde4e5053b12406c66a19183a24238e1  
.git/objects/51/519c7d8d60e0f958e135e8b989a78e84122591
```

```
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

- 这三个松散对象分别对应于撤销的提交，目录树，以及大文件对应的blob对象。

```
$ git cat-file -t 51519c7  
commit  
$ git cat-file -t d938dee  
tree  
$ git cat-file -t 2ebcd92  
blob
```

向上一节一样，执行`:command:`git prune``命令，期待版本库空间占用会变小。可是：

- 版本库空间占用没有变化！

```
$ git prune  
$ du -sh .git/  
11M      .git/
```

- 执行`:command:`git fsck``也看不到未被关联到的对象。

```
$ git fsck
```

- 除非像下面这样执行。

```
$ git fsck --no-reflogs  
dangling commit 51519c7d8d60e0f958e135e8b989a78e84122591
```

还记得前面章节中介绍的reflog么？reflog是防止误操作的最后一道闸门。

```
$ git reflog  
6652a0d HEAD@{0}: HEAD^: updating HEAD  
51519c7 HEAD@{1}: commit: add bigfiles.
```

可以看到撤销的操作仍然记录在reflog中，正因如此Git认为撤销的提交和大文件都还被可以被追踪到，还在使用着，所以无法用`:command:`git prune``命令删除。

如果确认真的要丢弃不想要的对象，需要对版本库的reflog做过期操作，相当于将`:file:`.git/logs/``下的文件清空。

- 使用下面的reflog过期命令做不到让刚刚撤销的提交过期，因为reflog的过期操作缺

省只会让90天前的数据过期。

```
$ git reflog expire --all
$ git reflog
6652a0d HEAD@{0}: HEAD^: updating HEAD
51519c7 HEAD@{1}: commit: add bigfiles.
```

- 需要为`:command:`git reflog``命令提供`--expire=<date>`参数，强制`<date>`之前的记录全部过期。

```
$ git reflog expire --expire=now --all
$ git reflog
```

使用`now`作为时间参数，让`reflog`的全部记录都过期。没有了`reflog`，即回滚的添加大文件的提交从`reflog`中看不到后，该提交对应的`commit`对象、`tree`对象和`blob`对象就会成为未被关联的`dangling`对象，可以用`:command:`git prune``命令清理。下面可以看到清理后，版本库变小了。

```
$ git prune
$ du -sh .git/
244K    .git/
```

## Git管家：git gc

前面两节介绍的是比较极端的情况，实际操作中会很少用到`:command:`git prune``命令来清理版本库，而是会使用一个更为常用的命令`:command:`git gc``。命令`:command:`git gc``就好比Git版本库的管家，会对版本库进行一系列的优化动作。

- 对分散在`:file:`.git/refs``下的文件进行打包，打包到文件`:file:`.git/packed-refs``中。

如果没有将配置`:file:`gc.packrefs``关闭，就会执行命令：`:command:`git pack-refs --all --prune``实现对引用的打包。

- 丢弃90天前的`reflog`记录。

会运行使`reflog`过期命令：`:command:`git reflog expire --all``。因为采用了缺省参数调用，因此只会清空`reflog`中90天前的记录。

- 对松散对象进行打包。

运行`:command:`git repack``命令，凡是引用对象都被打在包里，未被关联的

对象仍旧以松散对象形式保存。

- 清除未被关联的对象。缺省只清除2周以前的未被关联的对象。

可以向`:command:`git gc``提供`--prune=<date>`参数，其中的时间参数传递给`:command:`git prune --expire <date>``，实现对指定日期之前的未被关联的松散对象进行清理。

- 其他清理。

如运行`:command:`git rerere gc``对合并冲突的历史记录进行过期操作。

从上面的描述中可见命令`:command:`git gc``完成了相当多的优化和清理工作，并且最大限度照顾了安全性的需要。例如像暂存区操作引入的没有关联的临时对象会最少保留2个星期，而因为重置而丢弃的提交和文件则会保留最少3个月。

下面就把前面的例子用`:command:`git gc``再执行一遍，不过这一次添加的两个大文件要稍有不同，以便看到`:command:`git gc``打包所实现的对象增量存储的效果。

复制两个大文件到工作区。

```
$ cp /tmp/bigfile bigfile  
$ cp /tmp/bigfile bigfile.dup
```

在文件`:file:`bigfile.dup``后面追加些内容，造成`:file:`bigfile``和`:file:`bigfile.dup``内容不同。

```
$ echo "hello world" >> bigfile.dup
```

将这两个稍有不同的文件提交到版本库。

```
$ git add bigfile bigfile.dup  
$ git commit -m "add bigfiles."  
[master c62fa4d] add bigfiles.  
2 files changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 bigfile  
create mode 100644 bigfile.dup
```

可以看到版本库中提交进来的两个不同的大文件是不同的对象。

```
$ git ls-tree HEAD | grep bigfile  
100644 blob 2ebcd92d0dda2bad50c775dc662c6cb700477aff      bigfile  
100644 blob 9e35f946a30c11c47ba1df351ca22866bc351e7b      bigfile.dup
```

做版本库重置，抛弃最新的提交，即抛弃添加两个大文件的提交。

```
$ git reset --hard HEAD^  
HEAD is now at 6652a0d Add Images for git treeview.
```

此时的版本库有多大呢，还是像之前添加两个相同的大文件时占用11MB空间么？

```
$ du -sh .git/  
22M      .git/
```

版本库空间占用居然扩大了一倍！这显然是因为两个大文件分开存储造成的。可以用下面的命令在对象库中查看对象的大小。

```
$ find .git/objects -type f -printf "%-20p\t%s\n"  
.git/objects/0c/844d2a072fd69e71638558216b69ebc57ddb64 233  
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff 11184682  
.git/objects/9e/35f946a30c11c47ba1df351ca22866bc351e7b 11184694  
.git/objects/c6/2fa4d6cb4c082fadfa45920b5149a23fd7272e 162  
.git/objects/info/packs 54  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx 2892  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack 80015
```

输出的每一行用空白分隔，前面是文件名，后面是以字节为单位的文件大小。从上面的输出可以看出来，打包文件很小，但是有两个大的文件各自占用了11MB左右的空间。

执行:command:`git gc`并不会删除任何对象，因为这些对象都还没有过期。但是会发现版本库的占用变小了。

- 执行:command:`git gc`对版本库进行整理。

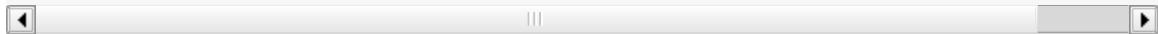
```
$ git gc  
Counting objects: 69, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (49/49), done.  
Writing objects: 100% (69/69), done.  
Total 69 (delta 11), reused 63 (delta 8)
```

- 版本库空间占用小了一半！

```
$ du -sh .git/  
11M      .git/
```

- 原来是因为对象库重新打包，两个大文件采用了增量存储使得版本库变小。

```
$ find .git/objects -type f -printf "%-20p\t%s\n" | sort
.git/objects/info/packs 54
.git/objects/pack/pack-7cae010c1b064406cd6c16d5a6ab2f446de4076c.idx 3004
.git/objects/pack/pack-7cae010c1b064406cd6c16d5a6ab2f446de4076c.pack 1126
```



如果想将抛弃的历史数据彻底丢弃，如下操作。

- 不再保留90天的reflog，而是将所有reflog全部即时过期。

```
$ git reflog expire --expire=now --all
```

- 通过`:command:`git fsck``可以看到有提交成为了未被关联的提交。

```
$ git fsck
dangling commit c62fa4d6cb4c082fadfa45920b5149a23fd7272e
```

- 这个未被关联的提交就是删除大文件的提交。

```
$ git show c62fa4d6cb4c082fadfa45920b5149a23fd7272e
commit c62fa4d6cb4c082fadfa45920b5149a23fd7272e
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 16 20:18:38 2010 +0800

    add bigfiles.

diff --git a/bigfile b/bigfile
new file mode 100644
index 000000..2ebcd92
Binary files /dev/null and b/bigfile differ
diff --git a/bigfile.dup b/bigfile.dup
new file mode 100644
index 000000..9e35f94
Binary files /dev/null and b/bigfile.dup differ
```

- 不带参数调用`:command:`git gc``虽然不会清除尚未过期（未到2周）的大文件，但是会将未被关联的对象从打包文件中移出，成为松散文件。

```
$ git gc
Counting objects: 65, done.
Delta compression using up to 2 threads.
```

```
Compressing objects: 100% (45/45), done.  
Writing objects: 100% (65/65), done.  
Total 65 (delta 8), reused 63 (delta 8)
```

- 未被关联的对象重新成为松散文件，所以`:file:`.git``版本库的空间占用又反弹了。

```
$ du -sh .git/  
22M      .git/  
$ find .git/objects -type f -printf "%-20p\t%s\n" | sort  
.git/objects/0c/844d2a072fd69e71638558216b69ebc57ddb64  233  
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff  11184682  
.git/objects/9e/35f946a30c11c47ba1df351ca22866bc351e7b  11184694  
.git/objects/c6/2fa4d6cb4c082fadfa45920b5149a23fd7272e  162  
.git/objects/info/packs 54  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx 2892  
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack 8001
```



- 实际上如果使用立即过期参数`--prune=now`调用`:command:`git gc``，就不用再等2周了，直接就可以完成对未关联的对象的清理。

```
$ git gc --prune=now  
Counting objects: 65, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (45/45), done.  
Writing objects: 100% (65/65), done.  
Total 65 (delta 8), reused 65 (delta 8)
```

- 清理过后，版本库的空间占用降了下来。

```
$ du -sh .git/  
240K      .git/
```

## Git管家的自动执行

对于老版本库的Git，会看到帮助手册中建议用户对版本库进行周期性的整理，以便获得更好的性能，尤其是对于规模比较大的项目，但是对于整理的周期都语焉不详。

实际上对于1.6.6及以后版本的Git已经基本上不需要手动执行`:command:`git gc``命令了，因为部分Git命令会自动调用`:command:`git gc --auto``命令，在版本库确实需要整理的情况下自动开始整理操作。

目前有如下Git命令会自动执行`:command:`git gc --auto``命令，实现对版本库的按需整

理。

- 执行命令:`git merge`进行合并操作后，对版本库进行按需整理。
- 执行命令:`git receive-pack`，即版本库接收其他版本库推送（push）的提交后，版本库会做按需整理操作。

当版本库接收到其他版本库的推送（push）请求时，会调用:`git receive-pack`命令以接收请求。在接收到推送的提交后，对版本库进行按需整理。

- 执行命令:`git rebase -i`进行交互式变基操作后，会对版本库进行按需整理。
- 执行命令:`git am`对mbox邮箱中通过邮件提交的补丁在版本库中进行应用的操作后，会对版本库做按需整理操作。

对于提供共享式“写操作”的Git版本库，可以免维护。所谓的共享式写操作，就是版本库作为一个裸版本库放在服务器上，团队成员可以通过推送（push）操作将提交推送到共享的裸版本中。每一次推送操作都会触发:`git gc --auto`命令，对版本库进行按需整理。

对于非独立工作的本地工作区，也可以免维护。因为和他人协同工作的本地工作区会经常执行:`git pull`操作从他人版本库或者从共享的版本库拉回新提交，执行:`git pull`操作会，会触发:`git merge`操作，因此也会对本地版本库进行按需整理。

Git管家命令使用`--auto`参数调用，会进行按需整理。因为版本库整理操作对于大的项目可能会非常费时，因此实际的整理并不会经常被触发，即有着非常苛刻的触发条件。想要观察到触发版本库整理操作是非常不容易的事情。

主要的触发条件是：松散对象只有超过一定的数量时才会执行。而且在统计松散对象数量时，为了降低在:`.git/objects/`目录下搜索松散对象对系统造成的负担，实际采取了取样搜索，即只会对对象库下一个子目录:`.git/objects/17`进行文件搜索。在缺省的配置下，只有该目录中对象数目超过27个，才会触发版本库的整理。至于为什么只在对象库下选择了一个子目录进行松散对象的搜索，这是因为SHA1哈希值是完全随机的，文件在由前两位哈希值组成的目录中差不多是平均分布的。至于为什么选择17，不知道对于作者Junio C Hamano有什么特殊意义，也许是向Linus Torvalds被评选为二十世纪最有影响力的100人中排名第17位而进行致敬。

可以通过配置:`gc.auto`的值，调整Git管家自动运行时触发版本库整理操作的频率，但是注意不要将:`gc.auto`设置为0，否则:`git gc --auto`命令永远不会触发版本库的整理。



# Git协议与工作协同

要想团队协作使用Git，就需要用到Git协议。

## Git支持的协议

首先来看看数据交换需要使用的协议。

Git提供了丰富的协议支持，包括：SSH、GIT、HTTP、HTTPS、FTP、FTPS、RSYNC及前面已经看到的本地协议等。各种不同协议的URL写法如表15-1所示。

表 15-1：Git支持的协议一览表

协议名称	语法格式	说明
SSH协议（1）	<code>ssh://[user@]example.com[:port]/path/to/repo.git/</code>	可在URL中设置用户名和端口。默认端口22。
SSH协议（2）	<code>[user@]example.com:path/to/repo.git/</code>	更为精简的SCP格式表示法，更简洁。但是非默认端口需要通过其他方式（如地址别名方式）设定。
GIT协议	<code>git://example.com[:port]/path/to/repo.git/</code>	最常用的只读协议。
HTTP[S]协议	<code>http[s]://example.com[:port]/path/to/repo.git/</code>	兼有智能协议和哑协议。
FTP[S]协议	<code>ftp[s]://example.com[:port]/path/to/repo.git/</code>	哑协议。
RSYNC协议	<code>rsync://example.com/path/to/repo.git/</code>	哑协议。
本地协议（1）	<code>file:///path/to/repo.git</code>	
本地协议（2）	<code>/path/to/repo.git</code>	和 <code>file://</code> 格式的本地协议类似。但有细微差别。 例如克隆时不支持浅克隆，且采用直接的硬连接实现克隆。

上面介绍的各种协议如果按照其聪明程度划分，可分为两类：智能协议和哑协议。

### 1. 智能协议

在通讯时使用智能协议，会在两个通讯的版本库的各自一端分别打开两个程序进行数据交换。使用智能协议最直观的印象就是在数据传输过程中会有清晰的进度显示，而且因为是按需传输所以传输量更小，速度更快。图15-1显示的就是在执行PULL和PUSH两个最常用的操作时，两个版本库各自启动辅助程序的情况。

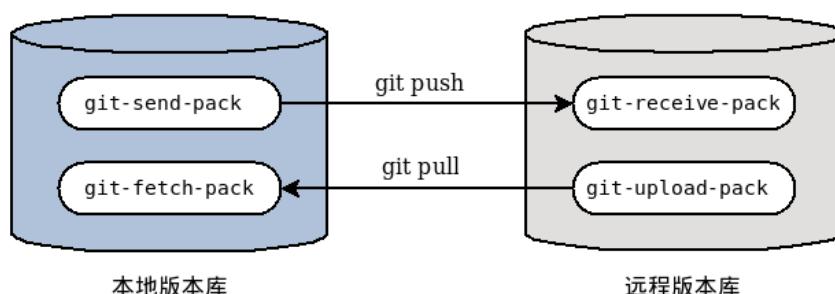


图 15-1: Git智能协议通讯示意图

上述协议中SSH、GIT及本地协议（file://）属于智能协议。HTTP协议需要特殊的配置（用git-[http-backend](#)配置CGI），并且客户端需要使用Git 1.6.6或更高的版本才能够使用智能协议。

## 2. 哑协议

和智能协议相对的是哑协议。使用哑协议在访问远程版本库的时候，远程版本库不会运行辅助程序，而是完全依靠客户端去主动“发现”。客户端需要访问文件:`file:~/.git/info/refs`获取当前版本库的引用列表，并根据引用对应的提交ID直接访问对象库目录下的文件。如果对象文件被打包而不以松散对象形式存在，则Git客户端还要去访问文件:`file:~/.git/objects/info/packs`以获得打包文件列表，并据此读取完整的打包文件，从打包文件中获取对象。由此可见哑协议的效率非常之低，甚至会因为要获取一个对象而去访问整个pack包。

使用哑协议最直观的感受是：传输速度非常慢，而且传输进度不可见，不知道什么时候才能够完成数据传输。上述协议中，像FTP、RSYNC都是哑协议，对于没有通过git-[http-backend](#)或类似CGI程序配置的HTTP服务器提供的也是哑协议。因为哑协议需要索引文件:`file:~/.git/info/refs`和:`file:~/.git/objects/info/packs`以获取引用和包列表，因此要在版本库的钩子脚本:`file:~post-update`中设置运行:`command:git update-server-info`以确保及时更新哑协议需要的索引文件。不过如果不使用哑协议，运行:`command:git update-server-info`就没有什么必要了。

以Git项目本身为例，看看如何使用不同的协议地址进行版本库克隆。

- Git协议（智能协议）：

```
git clone git://git.kernel.org/pub/scm/git/git.git
```

- HTTP(S)哑协议：

```
git clone http://www.kernel.org/pub/scm/git/git.git
```

- HTTP(S)智能协议：

使用Git 1.6.6或更高版本访问。

```
git clone https://github.com/git/git.git
```

## 多用户协同的本地模拟

在本篇（“Git和声”）的学习过程中，需要一个能够提供多人访问的版本库，显然要找到一个公共服务器，并且能让所有人都尽情发挥不太容易，但幸好可以使用本地协议来模拟。在后面的内容中，会经常使用本地协议地址`file:///path/to/repos/<project>.git`来代表对某一公共版本库的访问，您可以把`file://`格式的URL（比直接使用路径方式更逼真）想象为`git://`或者`http://`格式，并且想象它是在一台远程的服务器上，而非本机。

同样的，为了模拟多人的操作，也不再使用`file:~/path/to/my/workspace`作为工作区，而是分别使

用`file:~/path/to/user1/workspace`和`file:~/path/to/user2/workspace`等工作区来代表不同用户的工作环境。同样想象一

下`file:~/path/to/user1/`和`file:~/path/to/user2/`是在不同的主机上，并由不同的用户进行操作。

下面就来演示一个共享版本库的搭建过程，以及两个用户user1和user2在各自的工作区中

如何工作并进行数据交换的，具体过程如下。

- 创建一个共享的版本库，于:`:file:`/path/to/repos/shared.git``。

别忘了在第2篇的第13章“Git克隆”一章中介绍的，以裸版本库方式创建。

```
$ git init --bare /path/to/repos/shared.git
Initialized empty Git repository in /path/to/repos/shared.git/
```

- 用户user1克隆版本库。

克隆完成之后，在版本库级别设置`user.name`和`user.email`环境，以便和全局设置区分开，因为我们的模拟环境中所有用户都共享同一全局设置和系统设置。克隆及设置过程如下：

```
$ cd /path/to/user1/workspace
$ git clone file:///path/to/repos/shared.git project
Cloning into project...
warning: You appear to have cloned an empty repository.
$ cd project
$ git config user.name user1
$ git config user.email user1@sun.ossxp.com
```

- 用户user1创建初始数据并提交。

```
$ echo Hello. > README
$ git add README
$ git commit -m "initial commit."
[master (root-commit) 5174bf3] initial commit.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

- 用户user1将本地版本库的提交推送到上游。

在下面的推送指令中使用了`origin`别名，其实际指向就是`file:///path/to/repos/shared.git`。可以从`:file:`.git/config``配置文件中看到是如何实现对`origin`远程版本库注册的。关于远程版本库的内容要在第19章介绍。

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 210 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/shared.git
 * [new branch]      master -> master
```

- 用户user2克隆版本库。

```
$ cd /path/to/user2/workspace
$ git clone file:///path/to/repos/shared.git project
Cloning into project...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

- 同样在user2的本地版本库中，设置`user.name`和`user.email`环境，以区别全局环境设置。

```
$ cd /path/to/user2/workspace/project
```

```
$ git config user.name user2  
$ git config user.email user2@moon.osssp.com
```

- 用户user2的本地版本库现在拥有和user1用户同样的提交。

```
$ git log  
commit 5174bf33ab31a3999a6242fdcb1ec237e8f3f91a  
Author: user1 <user1@sun.osssp.com>  
Date:   Sun Dec 19 15:52:29 2010 +0800  
  
initial commit.
```

## 强制非快进式推送

现在用户user1和user2的工作区是相同的。思考一个问题：如果两人各自在本地版本库中进行独立的提交，然后再分别向共享版本库推送，会互相覆盖么？为了回答这个问题，进行下面的实践。

首先，用户user1先在本地版本库中进行提交，然后将本地的提交推送到“远程”共享版本库中。操作步骤如下：

- 用户user1创建:file:`team/user1.txt`文件。

假设这个项目约定：每个开发者在:file:`team`目录下写一个自述文件。于是用户user1创建文件:file:`team/user1.txt`。

```
$ cd /path/to/user1/workspace/project/  
$ mkdir team  
$ echo "I'm user1." > team/user1.txt  
$ git add team  
$ git commit -m "user1's profile."  
[master b4f3ae0] user1's profile.  
 1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 team/user1.txt
```

- 用户user1将本地提交推送到服务器上。

```
$ git push  
Counting objects: 5, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (4/4), 327 bytes, done.  
Total 4 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (4/4), done.  
To file:///path/to/repos/shared.git  
 5174bf3..b4f3ae0 master -> master
```

- 当前user1版本库中的日志

```
$ git log --oneline --graph  
* b4f3ae0 user1's profile.  
* 5174bf3 initial commit.
```

通过上面的操作步骤，可以看到用户user1成功的更新了“远程”共享版本库。如果用户user2在不知道用户user1所做的上述操作的前提下，在基于“远程”版本库旧的数据同步过来的本地版本库中进行改动，然后用户user2向“远程”共享版本库推送会有什么结果呢？用下面的操作验证一下。

- 用户user2创建:file:`team/user2.txt`文件。

```
$ cd /path/to/user2/workspace/project/
$ mkdir team
$ echo "I'm user1?" > team/user2.txt
$ git add team
$ git commit -m "user2's profile."
[master 8409e4c] user2's profile.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

- 用户user2将本地提交推送到服务器时出错。

```
$ git push
To file:///path/to/repos/shared.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

- 用户user2的推送失败了。错误日志翻译如下：

```
到版本库 file:///path/to/repos/shared.git
! [被拒绝]          master -> master (非快进)
错误：部分引用向 'file:///path/to/repos/shared.git' 推送失败
为防止您丢失历史，非快进式更新被拒绝。
在推送前请先合并远程改动，例如执行 'git pull'.
```

推送失败了。但这不是坏事情，反倒是一件好事情，因为这避免了用户提交的相互覆盖。Git通过对推送操作是否是“快进式”操作进行检查，从而保证用户的提交不会相互覆盖。一般情况下，推送只允许“快进式”推送。所谓快进式推送，就是要推送的本地版本库的提交是建立在远程版本库相应分支的现有提交基础上的，即远程版本库相应分支的最新提交是本地版本库最新提交的祖先提交。但现在用户user2执行的推送并非如此，是一个非快进式的推送。

- 此时用户user2本地版本库的最新提交及其历史提交可以用`:command:`git rev-list`命令显示，如下所示：`

```
$ git rev-list HEAD
8409e4c72388a18ea89eecb86d68384212c5233f
5174bf33ab31a3999a6242fdcb1ec237e8f3f91a
```

- 用`:command:`git ls-remote`命令显示远程版本库的引用对应的SHA1哈希值，会发现远程版本库所包含的最新提交的SHA1哈希值是b4f3ae0...，而不是本地最新提交的祖先提交。`

```
$ git ls-remote origin
b4f3ae0fcadce8c343f3cdc8a69c33cc98c98dfd      HEAD
b4f3ae0fcadce8c343f3cdc8a69c33cc98c98dfd      refs/heads/master
```

实际上当用户user2执行推送的时候，Git就是利用类似方法判断出当前的推送不是一个快进式推送，于是产生警告并终止。

那么如何才能成功推送呢？一个不见得正确的解决方案称为：**强制推送**。

在推送命令的后面使用-f参数可以进行强制推送，即使是非快进式的推送也会成功执行。用户user2执行强制推送，会强制刷新服务器中的版本。

```
$ git push -f
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (7/7), 503 bytes, done.
Total 7 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
 + b4f3ae0...8409e4c master -> master (forced update)
```

注意到了么，在强制推送的最后一行输出中显示了“强制更新（forced update）”字样。这样用户user1向版本库推送的提交由于用户user2的强制推送被覆盖了。实际上在这种情况下user1也可以强制的推送，从而用自己（user1）的提交再去覆盖用户user2的提交。这样的工作模式不是协同，而是战争！

#### 合理使用非快进式推送

在上面用户user2使用非快进式推送强制更新版本库，实际上是危险和错误的。滥用非快进式推送可能造成提交覆盖大战（战争是霸权的滥用）。正确地使用非快进式推送，应该是在不会造成提交覆盖“战争”的前提下，对历史提交进行修补。

下面的操作也许是一个使用非快进式推送更好的例子。

- 用户user2改正之前错误的录入。

细心的读者可能已经发现，用户user2在创建的个人描述文件中把自己的名字写错了。  
假设用户user2在刚刚完成向服务器的推送操作后也发现了这个错误，于是user2进行了下面的更改。

```
$ echo "I'm user2." > team/user2.txt
$ git diff
diff --git a/team/user2.txt b/team/user2.txt
index 27268e2..2dcb7b6 100644
--- a/team/user2.txt
+++ b/team/user2.txt
@@ -1 +1 @@
-I'm user1?
+I'm user2.
```

- 然后用户user2将修改好的文件提交到本地版本库中。

采用直接提交还是使用修补式提交，这是一个问题。因为前次提交已经被推送到共享版本库中，如果采用修补提交会造成前一次提交被新提交抹掉，从而在下次推送操作时造成非快进式推送。这时用户user2就要评估“战争”的风险：“我刚刚推送的提交，有没有可能被其他人获取了（通过`:command:`git pull``、`:command:`git fetch``或`:command:`git clone``操作）？”如果确认不会有他人获取，例如现在公司里只有user2自己一个人在加班，那么可以放心的进行修补操作。

```
$ git add -u
$ git commit --amend -m "user2's profile."
[master 6b1a7a0] user2's profile.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

- 采用强制推送，更新远程共享版本库中的提交。这个操作越早越好，在他人还没有来得及和服务器同步前将修补提交强制更新到服务器上。

```
$ git push -f
Counting objects: 5, done.
Delta compression using up to 2 threads.
```

```
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 331 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/shared.git
+ 8409e4c...6b1a7a0 master -> master (forced update)
```

## 合并后推送

理性的工作协同要避免非快进式推送。一旦向服务器推送后，如果发现错误，不要使用会更改历史的操作（变基、修补提交），而是采用不会改变历史提交的反转提交等操作。

如果在向服务器推送过程中遇到了非快进式推送的警告，应该进行如下操作才更为理性：  
执行`:command:`git pull``获取服务器端最新的提交并和本地提交进行合并，合并成功后再向服务器提交。

例如用户user1在推送时遇到了非快进式推送错误，可以通过如下操作将本地版本库的修改和远程版本库的最新提交进行合并。

- 用户user1发现推送遇到了非快进式推送。

```
$ cd /path/to/user1/workspace/project/
$ git push
To file:///path/to/repos/shared.git
! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```



- 执行`:command:`git pull``实现获取远程版本库的最新提交，以及实现获取到的远程版本库提交与本地提交的合并。

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
+ b4f3ae0...6b1a7a0 master      -> origin/master (forced update)
Merge made by recursive.
 team/user2.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

- 合并之后，看看版本库的提交关系图。

合并之后远程服务器中的最新提交`6b1a7a0`成为当前最新提交（合并提交）的父提交。  
如果再推送，则不再是快进式的了。

```
$ git log --graph --oneline
*   bccc620 Merge branch 'master' of file:///path/to/repos/shared
|\ 
| * 6b1a7a0 user2's profile.
* | b4f3ae0 user1's profile.
|/
* 5174bf3 initial commit.
```

- 执行推送，成功完成到远程版本库的推送。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 686 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
 6b1a7a0..bccc620 master -> master
```

## 禁止非快进式推送

非快进式推送如果被滥用，会成为项目的灾难：

- 团队成员之间的提交战争取代了本应的相互协作。
- 造成不必要的冲突，为他人造成麻烦。
- 在提交历史中引入包含修补提交前后两个版本的怪异的合并提交。

Git提供了至少两种方式对非快进式推送进行限制。一个是通过版本库的配置，另一个是通过版本库的钩子脚本。

将版本库的参数`receive.denyNonFastForwards`设置为`true`可以禁止任何用户进行非快进式推送。下面的示例中，可以看到针对一个已经预先设置为禁止非快进式推送的版本库执行非快进式推送操作，将会被禁止，即使使用强制推送操作。

- 更改服务器版本库`:file:`/path/to/repos/shared.git``的配置。

```
$ git --git-dir=/path/to/repos/shared.git config receive.denyNonFastForwa
```

- 在用户user1的工作区执行重置操作。

```
$ git reset --hard HEAD^1
$ git log --graph --oneline
* b4f3ae0 user1's profile.
* 5174bf3 initial commit.
```

- 用户user1使用强制推送也会失败。

在出错信息中看到服务器端拒绝执行：[remote rejected]。

```
$ git push -f
Total 0 (delta 0), reused 0 (delta 0)
remote: error: denying non-fast-forward refs/heads/master (you should pul
To file:///path/to/repos/shared.git
! [remote rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
```

另外一个方法是通过钩子脚本进行设置，可以仅对某些情况下的非快进式推送进行限制，而不是不分青红皂白地一概拒绝。例如：只对部分用户进行限制，而允许特定用户执行非快进式推送，或者允许某些分支可以进行强制提交而其他分支不可以。第5篇第30章会介绍Gitolite服务架设，通过授权文件（实际上通过版本库的`:file:`update``钩子脚本实现）对版本库非快进式推送做出更为精细的授权控制。

# 冲突解决

上一章介绍了Git协议，并且使用本地协议来模拟一个远程的版本库，以两个不同用户的身份检出该版本库，和该远程版本库进行交互——交换数据、协同工作。在上一章的协同中只遇到了一个小小的麻烦——非快进式推送，可以通过执行PULL（拉回）操作，成功完成合并后再推送。

但是在真实的运行环境中，用户间协同并不总是会一帆风顺，只要有合并就可能会有冲突。本章就重点介绍冲突解决机制。

## 拉回操作中的合并

为了降低难度，上一章的实践中用户user1执行`:command:`git pull``操作解决非快进式推送问题似乎非常的简单，就好像直接把共享版本库中最新提交直接拉回到本地，然后就可以推送了，其他好像什么都没有发生一样。真的是这样么？

- 用户user1向共享版本库推送时，因为user2强制推送已经改变了共享版本库中的提交状态，导致user1推送失败，如图16-1所示。

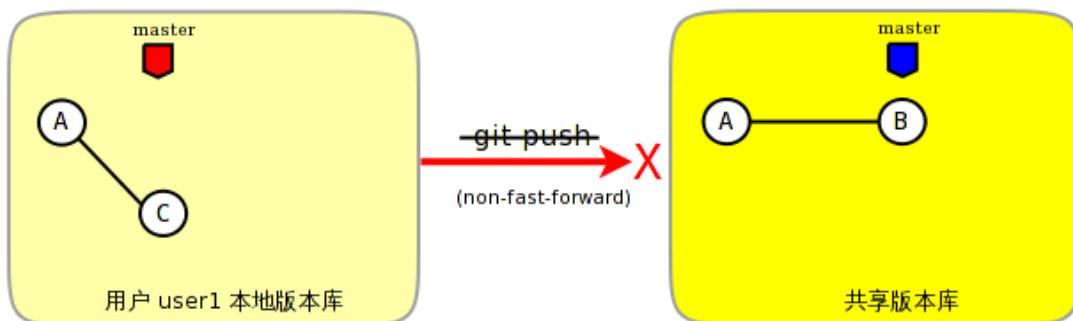


图 16-1：非快进式推送被禁止

- 用户user1执行PULL操作的第一阶段，将共享版本库master分支的最新提交拉回到本地，并更新到本地版本库特定的引用`refs/remotes/origin/master`（简称`origin/master`），如图16-2所示。

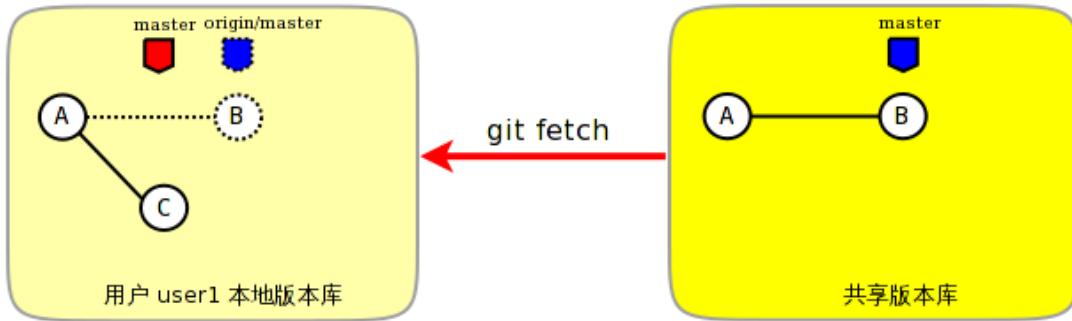


图 16-2: 执行获取操作

- 用户user1执行PULL操作的第二阶段，将本地分支master和共享版本库本地跟踪分支origin/master进行合并操作，如图16-3所示。

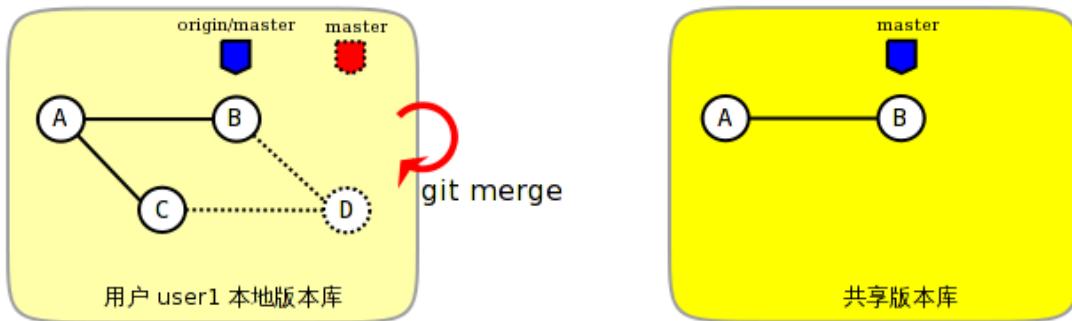


图 16-3: 执行合并操作

- 用户user1执行PUSH操作，将本地提交推送到共享版本库中，如图16-4所示。

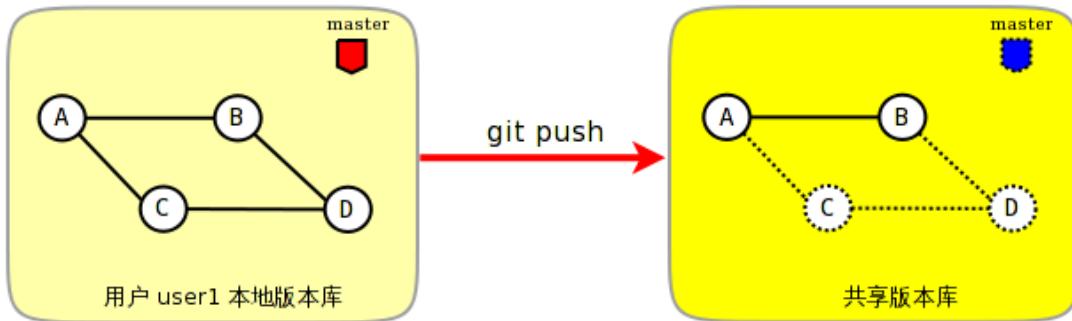


图 16-4: 执行推送操作

实际上拉回（PULL）操作是由两个步骤组成的，一个是获取（FETCH）操作，一个是合并（MERGE）操作，即：

```
git pull = git fetch + git merge
```

图16-2示意的获取（FETCH）操作看似很简单，实际上要到第19章介绍远程版本库的章节才能够讲明白，现在只需要根据图示将获取操作理解为将远程的共享版本库的对象（提交、里程碑、分支等）复制到本地即可。

合并（MERGE）操作是本章要介绍的重点。合并操作可以由拉回操作（`:command:`git pull``）隐式的执行，将其他版本库的提交和本地版本库的提交进行合并。还可以针对本版本库中的其他分支（将在第18章中介绍）进行显示的合并操作，将其他分支的提交和当前分支的提交进行合并。

合并操作的命令行格式如下：

```
git merge [选项...] <commit>...
```

合并操作的大多数情况，只须提供一个`<commit>`（提交ID或对应的引用：分支、里程碑等）作为参数。合并操作将`<commit>`对应的目录树和当前工作分支的目录树的内容进行合并，合并后的提交以当前分支的提交作为第一个父提交，以`<commit>`为第二个父提交。合并操作还支持将多个`<commit>`代表的分支和当前分支进行合并，过程类似。合并操作的选项很多，这会在本章及第24章“子树合并”中予以介绍。

默认情况下，合并后的结果会自动提交，但是如果提供`--no-commit`选项，则合并后的结果会放入暂存区，用户可以对合并结果进行检查、更改，然后手动提交。

合并操作并非总会成功，因为合并的不同提交可能同时修改了同一文件相同区域的内容，导致冲突。冲突会造成合并操作的中断，冲突的文件被标识，用户可以对标识为冲突的文件进行冲突解决操作，然后更新暂存区，再提交，最终完成合并操作。

根据合并操作是否遇到冲突，以及不同的冲突类型，可以分为以下几种情况：成功的自动合并、逻辑冲突、真正的冲突和树冲突。下面分别予以介绍。

## 合并一：自动合并

Git的合并操作非常智能，大多数情况下会自动完成合并。不管是修改不同的文件，还是修改相同的文件（文件的不同位置），或者文件名变更。

### 修改不同的文件

如果用户user1和user2各自的本地提交中修改了不同的文件，当一个用户将改动推送到服务器后，另外一个用户推送就遇到非快进式推送错误，需要先合并再推送。因两个用户修改了不同的文件，合并不会遇到麻烦。

在上一章的操作过程中，两个用户的本地版本库和共享版本库可能不一致，为确保版本库

状态的一致性以便下面的实践能够正常执行，分别在两个用户的本地版本库中执行下面的操作。

```
$ git pull  
$ git reset --hard origin/master
```

下面的实践中，两个用户分别修改不同的文件，其中一个用户要尝试合并操作将本地提交和另外一个用户的提交合并。

- 用户user1修改:file:`team/user1.txt`文件，提交并推送到共享服务器。

```
$ cd /path/to/user1/workspace/project/  
$ echo "hack by user1 at `date -R`" >> team/user1.txt  
$ git add -u  
$ git commit -m "update team/user1.txt"  
$ git push
```

- 用户user2修改:file:`team/user2.txt`文件，提交。

```
$ cd /path/to/user2/workspace/project/  
$ echo "hack by user2 at `date -R`" >> team/user2.txt  
$ git add -u  
$ git commit -m "update team/user2.txt"
```

- 用户user2在推送的时候，会遇到非快进式推进的错误而被终止。

```
$ git push  
To file:///path/to/repos/shared.git  
! [rejected]          master -> master (non-fast-forward)  
error: failed to push some refs to 'file:///path/to/repos/shared.git'  
To prevent you from losing history, non-fast-forward updates were rejected.  
Merge the remote changes (e.g. 'git pull') before pushing again. See the  
'Note about fast-forwards' section of 'git push --help' for details.
```

- 用户user2执行获取(:command:`git fetch`)操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用origin/master中。

```
$ git fetch  
remote: Counting objects: 7, done.  
remote: Compressing objects: 100% (4/4), done.  
remote: Total 4 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (4/4), done.
```

```
From file:///path/to/repos/shared  
 bccc620..25fce74 master -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge origin/master  
Merge made by recursive.  
 team/user1.txt | 1 +  
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push  
Counting objects: 12, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (7/7), done.  
Writing objects: 100% (7/7), 747 bytes, done.  
Total 7 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (7/7), done.  
To file:///path/to/repos/shared.git  
 25fce74..0855b86 master -> master
```

- 通过提交日志，可以看到成功合并的提交和其两个父提交的关系图。

```
$ git log -3 --graph --stat  
* commit 0855b86678d1cf86ccdd13adaaa6e735715d6a7e  
| \ Merge: f53acdf 25fce74  
| | Author: user2 <user2@moon.ossxp.com>  
| | Date: Sat Dec 25 23:00:55 2010 +0800  
| |  
| |     Merge remote branch 'origin/master'  
| |  
| * commit 25fce74b5e73b960c42e4a463d03d462919b674d  
| | Author: user1 <user1@sun.ossxp.com>  
| | Date: Sat Dec 25 22:54:53 2010 +0800  
| |  
| |     update team/user1.txt  
| |  
| |     team/user1.txt | 1 +  
| | 1 files changed, 1 insertions(+), 0 deletions(-)  
| |  
* | commit f53acdf6a76e0552b562f5aaa4d40ff19e8e2f77  
| / Author: user2 <user2@moon.ossxp.com>  
| | Date: Sat Dec 25 22:56:49 2010 +0800  
| |  
| |     update team/user2.txt
```

```
|  
|   team/user2.txt |      1 +  
| 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 修改相同文件的不同区域

当用户user1和user2在本地提交中修改相同的文件，但是修改的是文件的不同位置时，则两个用户的提交仍可成功合并。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区中修改:`:file:`README``文件，在文件的第一行插入内容，更改后的文件内容如下。

```
User1 hacked.  
Hello.
```

- 用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u  
$ git commit -m "User1 hack at the beginning."  
$ git push
```

- 用户user2在自己的工作区中修改:`:file:`README``文件，在文件的最后插入内容，更改后的文件内容如下。

```
Hello.  
User2 hacked.
```

- 用户user2对修改进行本地提交。

```
$ git add -u  
$ git commit -m "User2 hack at the end."
```

- 用户user2执行获取(`:command:`git fetch``)操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用origin/master中。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
  0855b86..07e9d08  master      -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Auto-merging README
Merge made by recursive.
 README |      1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 607 bytes, done.
Total 6 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
  07e9d08..2a67e6f  master -> master
```

- 如果追溯一下`:file:`README``文件每一行的来源，可以看到分别是user1和user2更改的最前和最后的一行。

```
$ git blame README
07e9d082 (user1 2010-12-25 23:12:17 +0800 1) User1 hacked.
^5174bf3 (user1 2010-12-19 15:52:29 +0800 2) Hello.
bb0c74fa (user2 2010-12-25 23:14:27 +0800 3) User2 hacked.
```

## 同时更改文件名和文件内容

如果一个用户将文件移动到其他目录（或修改文件名），另外一个用户针对重命名前的文件进行了修改，还能够实现自动合并么？这对于其他版本控制系统可能是一个难题，例如Subversion就不能很好地处理，还为此引入了一个“树冲突”的新名词。Git对于此类冲突能够很好地处理，可以自动解决冲突实现自动合并。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区中将文件:`:file:`README``进行重命名，本地提交并推送到共享版本库。

```
$ cd /path/to/user1/workspace/project/
$ mkdir doc
$ git mv README doc/README.txt
$ git commit -m "move document to doc/."
$ git push
```

- 用户user2在自己的工作区中修改:`:file:`README``文件，在文件的最后插入内容，并本地提交。

```
$ cd /path/to/user2/workspace/project/
$ echo "User2 hacked again." >> README
$ git add -u
$ git commit -m "User2 hack README again."
```

- 用户user2执行获取(`:command:`git fetch``)操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用`origin/master`中。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
  0855b86..07e9d08  master      -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Merge made by recursive.
 README => doc/README.txt |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename README => doc/README.txt (100%)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 636 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
  9c51cb9..f73db10  master -> master
```

- 使用-m参数可以查看合并操作所做出的修改。

```
$ git log -1 -m --stat
commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 887488eee19300c566c
Merge: 887488e 9c51cb9
Author: user2 <user2@moon.ossxp.com>
Date:   Sat Dec 25 23:36:57 2010 +0800

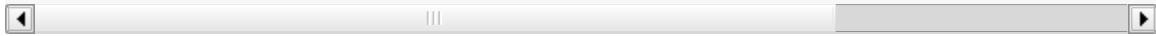
Merge remote branch 'refs/remotes/origin/master'

 README          |    4 -----
 doc/README.txt |    4 +++
 2 files changed, 4 insertions(+), 4 deletions(-)

commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 9c51cb91bfe12654e2d
Merge: 887488e 9c51cb9
Author: user2 <user2@moon.ossxp.com>
Date:   Sat Dec 25 23:36:57 2010 +0800

Merge remote branch 'refs/remotes/origin/master'

 doc/README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```



## 合并二：逻辑冲突

自动合并如果成功地执行，则大多数情况下就意味着完事大吉，但是在某些特殊情况下，合并后的结果虽然在Git看来是完美的合并，实际上却存在着逻辑冲突。

一个典型的逻辑冲突是一个用户修改了一个文件的文件名，而另外的用户在其他文件中引用旧的文件名，这样的合并虽然能够成功但是包含着逻辑冲突。例如：

- 一个C语言的项目中存在头文件:`file:`hello.h``，该头文件定义了一些函数声明。
- 用户user1将:`file:`hello.h``文件改名为:`file:`api.h``。

- 用户user2写了一个新的源码文件:`file: `foo.c``并在该文件中包含了:`file: `hello.h``文件。
- 两个用户的提交合并后，会因为源码文件:`file: `foo.c``找不到包含的:`file: `hello.h``文件而导致项目编译失败。

再举一个逻辑冲突的示例。假如一个用户修改了函数返回值而另外的用户使用旧的函数返回值，虽然成功合并但是存在逻辑冲突：

- 函数`compare(obj1, obj2)`用于比较两个对象`obj1`和`obj2`。返回`1`代表比较的两个对象相同，返回`0`代表比较的两个对象不同。
- 用户user1修改了该函数的返回值，返回`0`代表两个对象相同，返回`1`代表`obj1`大于`obj2`，返回`-1`则代表`obj1`小于`obj2`。
- 用户user2不知道user1对该函数的改动，仍以该函数原返回值判断两个对象的异同。
- 两个用户的提交合并后，不会出现编译错误，但是软件中会潜藏着重大的Bug。

上面的两个逻辑冲突的示例，尤其是最后一个非常难以捕捉。如果因此而贬低Git的自动合并，或者对每次自动合并的结果疑神疑鬼，进而花费大量精力去分析合并的结果，则是因噎废食、得不偿失。一个好的项目实践是每个开发人员都为自己的代码编写可运行的单元测试，项目每次编译时都要执行自动化测试，捕捉潜藏的Bug。在2010年OpenParty上的一个报告中，我介绍了如何在项目中引入单元测试及自动化集成，可以参考下面的链接：

- <http://www.beijing-open-party.org/topic/9>
- <http://wenku.baidu.com/view/63bf7d160b4e767f5acfcef6.html>

## 合并三：冲突解决

如果两个用户修改了同一文件的同一区域，则在合并的时候会遇到冲突导致合并过程中断。这是因为Git并不能越俎代庖的替用户做出决定，而是把决定权交给用户。在这种情况下，Git显示为合并冲突，等待用户对冲突做出抉择。

下面的实践非常简单，两个用户都修改:`file: `doc/README.txt``文件，在第二行“Hello.”的后面加上自己的名字。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区修改:`file: `doc/README.txt``文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.  
Hello, user1.  
User2 hacked.  
User2 hacked again.
```

- 用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u  
$ git commit -m "Say hello to user1."  
$ git push
```

- 用户user2在自己的工作区修改:file:`doc/README.txt`文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.  
Hello, user2.  
User2 hacked.  
User2 hacked again.
```

- 用户user2对修改进行本地提交。

```
$ git add -u  
$ git commit -m "Say hello to user2."
```

- 用户user2执行拉回操作，遇到冲突。

git pull操作相当于git fetch和git merge两个操作。

```
$ git pull  
remote: Counting objects: 7, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 4 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (4/4), done.  
From file:///path/to/repos/shared  
      f73db10..a123390  master      -> origin/master  
Auto-merging doc/README.txt  
CONFLICT (content): Merge conflict in doc/README.txt  
Automatic merge failed; fix conflicts and then commit the result.
```

执行:command:`git pull`时所做的合并操作由于遇到冲突导致中断。来看看处于合并冲突状态时工作区和暂存区的状态。

执行:command:`git status`命令，可以从状态输出中看到文件:file:`doc/README.txt`处

于未合并的状态，这个文件在两个不同的提交中都做了修改。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:    doc/README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

那么Git是如何记录合并过程及冲突的呢？实际上合并过程是通过:`:file:`.git``目录下的几个文件进行记录的：

- 文件:`:file:`.git/MERGE_HEAD``记录所合并的提交ID。
- 文件:`:file:`.git/MERGE_MSG``记录合并失败的信息。
- 文件:`:file:`.git/MERGE_MODE``标识合并状态。

版本库暂存区中则会记录冲突文件的多个不同版本。可以使用:`:command:`git ls-files``命令查看。

```
$ git ls-files -s
100644 ea501534d70a13b47b3b4b85c39ab487fa6471c2 1      doc/README.txt
100644 5611db505157d312e4f6fb1db2e2c5bac2a55432 2      doc/README.txt
100644 036dbc5c11b0a0cefc8247cf0e9a3e678f8de060 3      doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0      team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0      team/user2.txt
```

在上面的输出中，每一行分为四个字段，前两个分别是文件的属性和SHA1哈希值。第三个字段是暂存区编号。当合并冲突发生后，会用到0以上的暂存区编号。

- 编号为1的暂存区用于保存冲突文件修改之前的副本，即冲突双方共同的祖先版本。可以用`:1:<filename>`访问。

```
$ git show :1:doc/README.txt
User1 hacked.
Hello.
User2 hacked.
User2 hacked again.
```

- 编号为2的暂存区用于保存当前冲突文件在当前分支中修改的副本。可以

用:2:<filename>访问。

```
$ git show :2:doc/README.txt
User1 hacked.
Hello, user2.
User2 hacked.
User2 hacked again.
```

- 编号为3的暂存区用于保存当前冲突文件在合并版本（分支）中修改的副本。可以用:3:<filename>访问。

```
$ git show :3:doc/README.txt
User1 hacked.
Hello, user1.
User2 hacked.
User2 hacked again.
```

对暂存区中冲突文件的上述三个副本无须了解太多，这三个副本实际上是提供冲突解决工具，用于实现三向文件合并的。

工作区的版本则可能同时包含了成功的合并及冲突的合并，其中冲突的合并会用特殊的标记(<<<<< ===== >>>>>)进行标识。查看当前工作区中冲突的文件：

```
$ cat doc/README.txt
User1 hacked.
<<<<< HEAD
Hello, user2.
=====
Hello, user1.
>>>>> a123390b8936882bd53033a582ab540850b6b5fb
User2 hacked.
User2 hacked again.
```

特殊标识<<<<< (七个小于号) 和===== (七个等号) 之间的内容是当前分支所更改的内容。在特殊标识===== (七个等号) 和>>>>> (七个大于号) 之间的内容是所合并的版本更改的内容。

冲突解决的实质就是通过编辑操作，将冲突标识符所标识的冲突内容替换为合适的内容，并去掉冲突标识符。编辑完毕后执行:command:`git add`命令将文件添加到暂存区（标号0），然后再提交就完成了冲突解决。

当工作区处于合并冲突状态时，无法再执行提交操作。此时有两个选择：放弃合并操作，或者对合并冲突进行冲突解决操作。放弃合并操作非常简单，只须执行:command:`git

`reset``将暂存区重置即可。下面重点介绍如何进行冲突解决的操作。有两个方法进行冲突解决，一个是对少量冲突非常适合的手工编辑操作，另外一个是使用图形化冲突解决工具。

## 手工编辑完成冲突解决

先来看看不使用工具，直接手动编辑完成冲突解决。打开文件:`:file:`doc/README.txt``，将冲突标识符所标识的文字替换为`Hello, user1 and user2..`。修改后的文件内容如下：

```
User1 hacked.  
Hello, user1 and user2.  
User2 hacked.  
User2 hacked again.
```

然后添加到暂存区，并提交：

```
$ git add -u  
$ git commit -m "Merge completed: say hello to all users."
```

查看最近三次提交的日志，会看到最新的提交就是一个合并提交：

```
$ git log --oneline --graph -3  
* bd3ad1a Merge completed: say hello to all users.  
|\  
| * a123390 Say hello to user1.  
* | 60b10f3 Say hello to user2.  
|/  
|/
```

提交完成后，会看到`:file:`.git``目录下与合并相关的文件`:file:`.git/MERGE_HEAD``、`:file:`.git/MERGE_MSG``、`:file:`.git/MERGE_MODE``文件都自动删除了。

如果查看暂存区，会发现冲突文件在暂存区中的三个副本也都清除了（实际在对编辑完成的冲突文件执行`:command:`git add``后就已经清除了）。

```
$ git ls-files -s  
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0 doc/README.txt  
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0 team/user1.txt  
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0 team/user2.txt
```

## 图形工具完成冲突解决

上面介绍的通过手工编辑完成冲突解决并不复杂，对于简单的冲突是最快捷的解决方法。但是如果冲突的区域过多、过大，并且缺乏原始版本作为参照，冲突解决过程就会显得非常的不便，这种情况下使用图形工具就显得非常有优势。

还以上面的冲突解决为例介绍使用图形工具进行冲突解决的方法。为了制造一个冲突，首先把user2辛苦完成的冲突解决提交回滚，再执行合并进入冲突状态。

- 将冲突解决的提交回滚，强制重置到前一个版本。

```
$ git reset --hard HEAD^
```

- 这时查看状态，会显示当前工作分支的最新提交和共享版本库的master分支的最新提交出现了偏离。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
nothing to commit (working directory clean)
```

- 那么执行合并操作吧。冲突发生了。

```
$ git merge refs/remotes/origin/master
Auto-merging doc/README.txt
CONFLICT (content): Merge conflict in doc/README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

下面就演示使用图形工具如何解决冲突。使用图形工具进行冲突解决需要事先在操作系统中安装相关的工具软件，如：kdiff3、meld、tortoisemerge、araxis等。而启动图形工具进行冲突解决也非常简单，只须执行命令：`git mergetool` 即可。

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge
gvimdiff diffuse ecmerge p4merge araxis emerge vimdiff
Merging:
doc/README.txt

Normal merge conflict for 'doc/README.txt':
{local}: modified
{remote}: modified
Hit return to start merge resolution tool (kdiff3):
```

运行`:command:`git mergetool``命令后，会显示支持的图形工具列表，并提示用户选择可用的冲突解决工具。默认会选择系统中已经安装的工具软件，如`:command:`kdiff3``。直接按回车键，自动打开`:command:`kdiff3``进入冲突解决界面：

启动`:command:`kdiff3``后，如图16-5，上方三个窗口由左至右显示冲突文件的三个版本，分别是：

1. 暂存区1中的版本（共同祖先版本）。
2. 暂存区2中的版本（当前分支更改的版本）。
3. 暂存区3中的版本（他人更改的版本）。

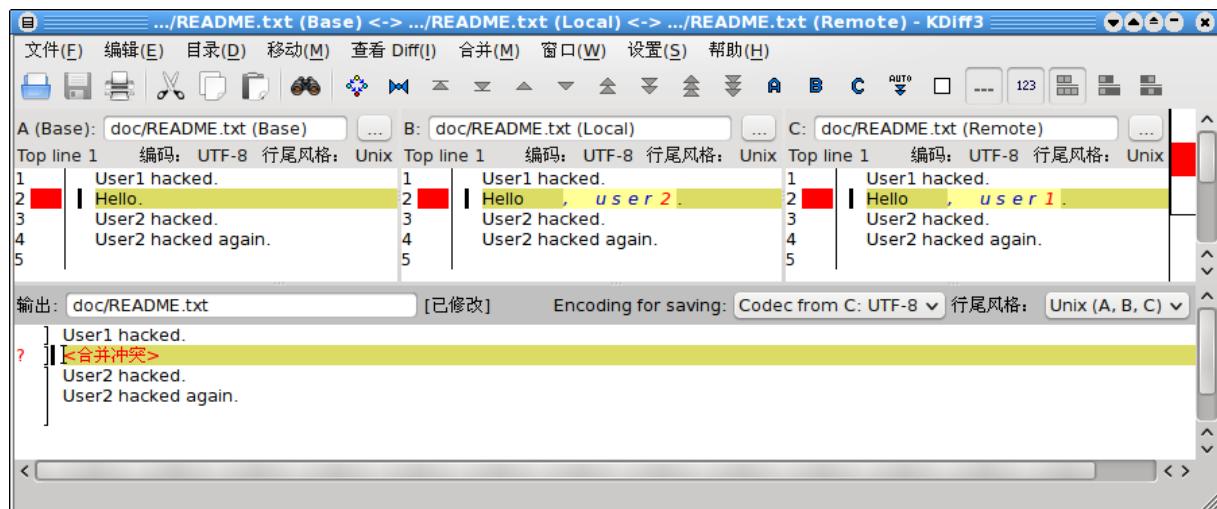


图 16-5: kdiff3 冲突解决界面

kdiff3下方的窗口是合并后文件的编辑窗口。如图16-6所示，点击标记为“合并冲突”的一行，在弹出菜单中出现A、B、C三个选项，分别代表从A、B、C三个窗口拷贝相关内容到当前位置。

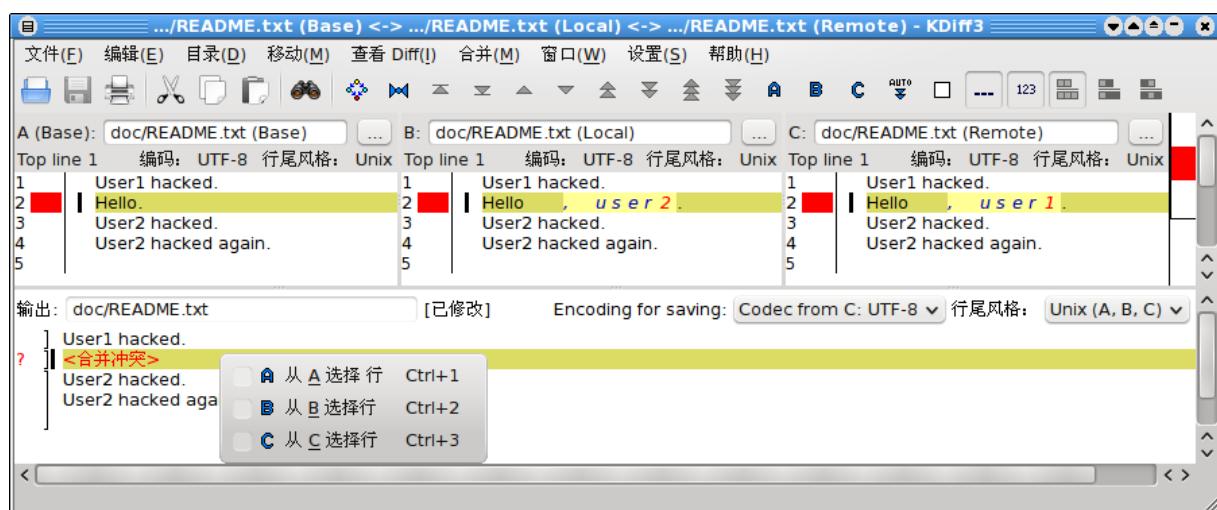


图 16-6: kdiff3 合并冲突行的弹出菜单

当通过图16-6显示的弹出菜单选择了B和C后，可以在图16-7中看到在合并窗口出现了标识B和C的行，分别代表user2和user1对该行的修改。

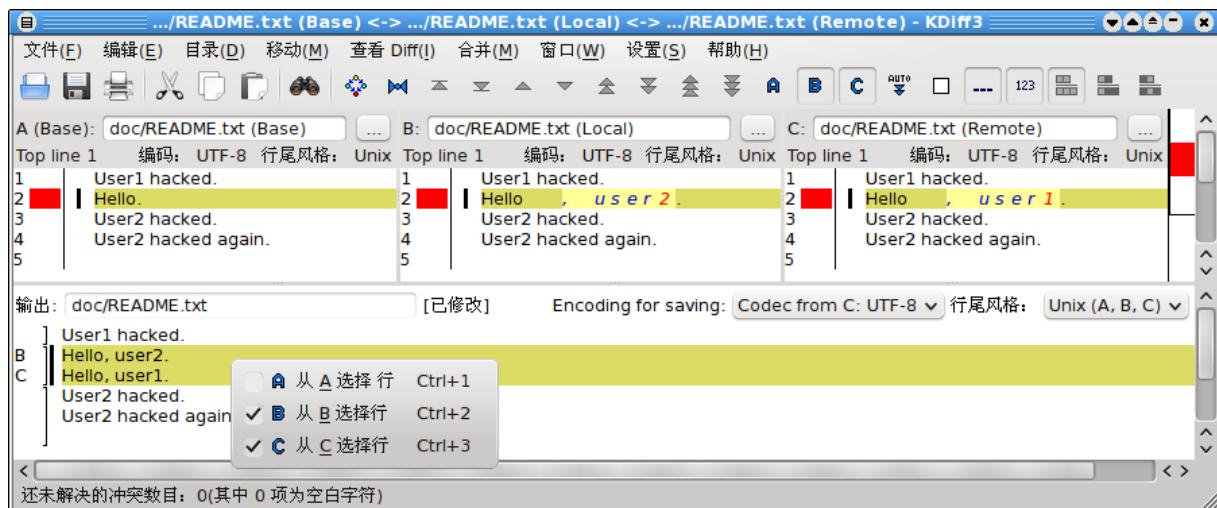


图 16-7: 在 kdiff3 冲突区域同时选取B和C的修改

在合并窗口进行编辑，将“Hello, user1.”修改为“Hello, user1 and user2.”，如图 16-8。修改后，可以看到该行的标识由c改变为m，含义是该行是经过手工修改的行。

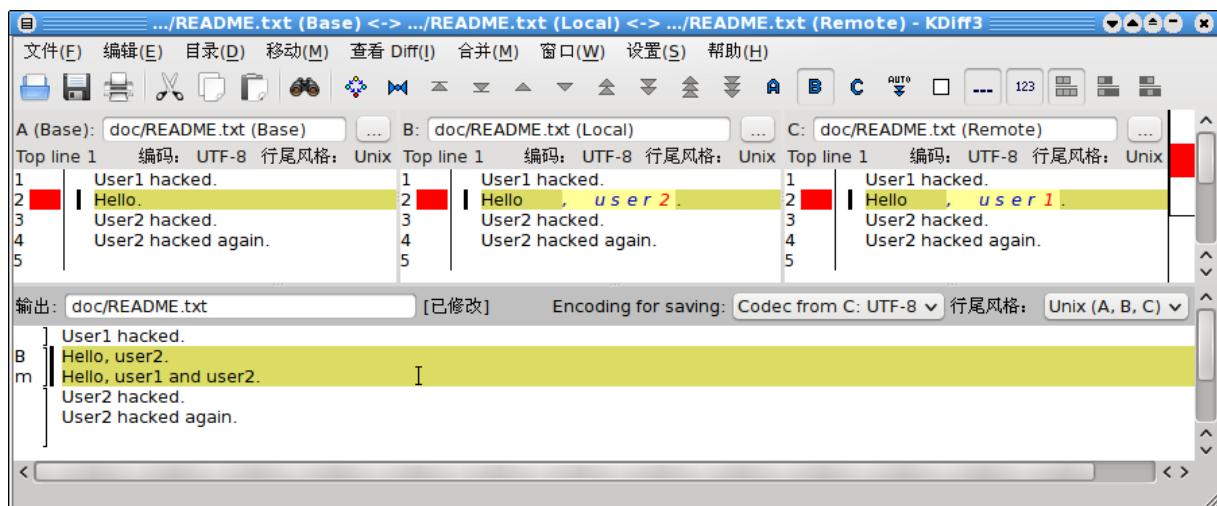


图 16-8: 在 kdiff3 的冲突区域编辑内容

在合并窗口删除标识为从B窗口引入的行“Hello, user2.”，如图16-9。保存退出即完成图形化冲突解决。

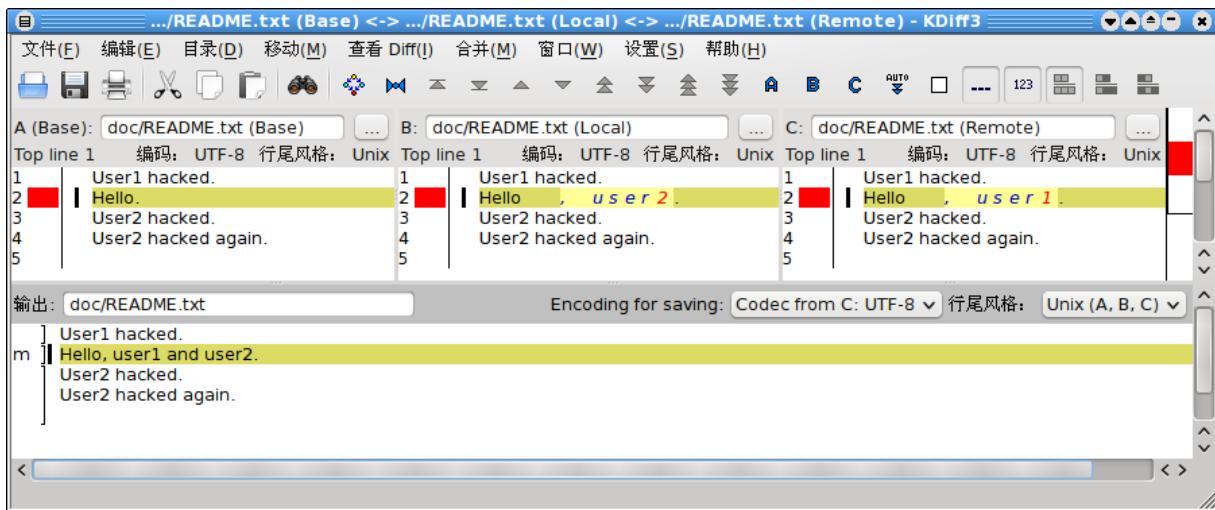


图 16-9：完成 kdiff3 冲突区域的编辑

图形工具保存退出后，显示工作区状态，会看到冲突已经解决。在工作区还会遗留一个以.orig结尾的合并前文件副本。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Changes to be committed:
#
#       modified:   doc/README.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       doc/README.txt.orig
```

查看暂存区会发现暂存区中的冲突文件的三个副本都已经清除。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0      doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0      team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0      team/user2.txt
```

执行提交和推送。

```
$ git commit -m "Say hello to all users."
[master 7f7bb5e] Say hello to all users.
$ git push
Counting objects: 14, done.
```

```
Delta compression using up to 2 threads.  
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (8/8), 712 bytes, done.  
Total 8 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (8/8), done.  
To file:///path/to/repos/shared.git  
 a123390..7f7bb5e master -> master
```

查看最近三次的提交日志，会看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3  
* 7f7bb5e Say hello to all users.  
|\  
| * a123390 Say hello to user1.  
* | 60b10f3 Say hello to user2.  
|/  
|/
```

## 合并四：树冲突

如果一个用户将某个文件改名，另外一个用户将同样的文件改为另外的名字，当这两个用户的提交进行合并操作时，Git显然无法替用户做出裁决，于是就产生了冲突。这种因为文件名修改造成的冲突，称为树冲突。这种树冲突的解决方式比较特别，因此专题介绍。

仍旧使用前面的版本库进行此次实践。为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

下面就分别以两个用户的身份执行提交，将同样的一个文件改为不同的文件名，制造一个树冲突。

- 用户user1将文件:`file:`doc/README.txt``改名为:`file:`readme.txt``，提交并推送到共享版本库。

```
$ cd /path/to/user1/workspace/project  
$ git mv doc/README.txt readme.txt  
$ git commit -m "rename doc/README.txt to readme.txt"  
[master 615c1ff] rename doc/README.txt to readme.txt  
 1 files changed, 0 insertions(+), 0 deletions(-)  
  rename doc/README.txt => readme.txt (100%)  
$ git push  
Counting objects: 3, done.  
Delta compression using up to 2 threads.
```

```
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 282 bytes, done.  
Total 2 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (2/2), done.  
To file:///path/to/repos/shared.git  
    7f7bb5e..615c1ff master -> master
```

- 用户user2将文件`doc/README.txt`改名为`README`，并做本地提交。

```
$ cd /path/to/user2/workspace/project  
$ git mv doc/README.txt README  
$ git commit -m "rename doc/README.txt to README"  
[master 20180eb] rename doc/README.txt to README  
 1 files changed, 0 insertions(+), 0 deletions(-)  
 rename doc/README.txt => README (100%)
```

- 用户user2执行`git pull`操作，遇到合并冲突。

```
$ git pull  
remote: Counting objects: 3, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 2 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (2/2), done.  
From file:///path/to/repos/shared  
    7f7bb5e..615c1ff master -> origin/master  
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HE  
Automatic merge failed; fix conflicts and then commit the result.
```

因为两个用户同时更改了同一文件的文件名并且改成了不同的名字，于是引发冲突。此时查看状态会看到：

```
$ git status  
# On branch master  
# Your branch and 'refs/remotes/origin/master' have diverged,  
# and have 1 and 1 different commit(s) each, respectively.  
#  
# Unmerged paths:  
#   (use "git add/rm <file>..." as appropriate to mark resolution)  
#  
#       added by us:          README  
#       both deleted:       doc/README.txt  
#       added by them:       readme.txt  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

此时查看一下用户user2本地版本库的暂存区，可以看到因为冲突在编号为1、2、3的暂存区出现了相同SHA1哈希值的对象，但是文件名各不相同。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 2      README
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 1      doc/README.txt
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 3      readme.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0      team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0      team/user2.txt
```

其中在暂存区1中是改名之前的`:file:`doc/README.txt``，在暂存区2中是用户user2改名后的文件名`:file:`README``，而暂存区3是其他用户（user1）改名后的文件`:file:`readme.txt``。

此时的工作区中存在两个相同的文件`:file:`README``和`:file:`readme.txt``分别是用户user2和user1对`:file:`doc/README.txt``重命名之后的文件。

```
$ ls -l readme.txt README
-rw-r--r-- 1 jiangxin jiangxin 72 12月 27 12:25 README
-rw-r--r-- 1 jiangxin jiangxin 72 12月 27 16:53 readme.txt
```

## 手工操作解决树冲突

这时user2应该和user1商量一下到底应该将该文件改成什么名字。如果双方最终确认应该采用user2重命名的名称，则user2应该进行下面的操作完成冲突解决。

- 删除文件`:file:`readme.txt``。

在执行`:command:`git rm``操作过程会弹出三条警告，说共有三个文件待合并。

```
$ git rm readme.txt
README: needs merge
doc/README.txt: needs merge
readme.txt: needs merge
rm 'readme.txt'
```

- 删除文件`:file:`doc/README.txt``。

执行删除过程，弹出的警告少了一条，因为前面的删除操作已经将一个冲突文件撤出暂存区了。

```
$ git rm doc/README.txt
```

```
README: needs merge  
doc/README.txt: needs merge  
rm 'doc/README.txt'
```

- 添加文件:file:` README`。

```
$ git add README
```

- 这时查看一下暂存区，会发现所有文件都在暂存区0中。

```
$ git ls-files -s  
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0 README  
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0 team/user1.txt  
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0 team/user2.txt
```

- 提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."  
[master e82187e] fixed tree conflict.
```

- 查看一下最近三次提交日志，看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3 -m --stat  
*   e82187e (from 615c1ff) fixed tree conflict.  
|\  
| | README      |    4 ++++  
| | readme.txt |    4 ----  
| | 2 files changed, 4 insertions(+), 4 deletions(-)  
* 615c1ff rename doc/README.txt to readme.txt  
| | doc/README.txt |    4 ----  
| | readme.txt  |    4 ++++  
| | 2 files changed, 4 insertions(+), 4 deletions(-)  
* | 20180eb rename doc/README.txt to README  
|/  
| README          |    4 ++++  
| doc/README.txt |    4 ----  
| 2 files changed, 4 insertions(+), 4 deletions(-)
```

## 交互式解决树冲突

树冲突虽然不能像文件冲突那样使用图形工具进行冲突解决，但还是可以用:command:`git mergetool`命令，通过交互式问答快速解决此类冲突。

首先将user2的工作区重置到前一次提交，再执行:command:`git merge`引发树冲突。

- 重置到前一次提交。

```
$ cd /path/to/user2/workspace/project  
$ git reset --hard HEAD^  
HEAD is now at 20180eb rename doc/README.txt to README  
$ git clean -fd
```

- 执行:command:`git merge`引发树冲突。

```
$ git merge refs/remotes/origin/master  
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HE  
Automatic merge failed; fix conflicts and then commit the result.  
$ git status -s  
AU README  
DD doc/README.txt  
UA readme.txt
```

上面操作所引发的树冲突，可以执行:command:`git mergetool`命令进行交互式冲突解决，会如下逐一提示用户进行选择。

- 执行:command:`git mergetool`命令。忽略其中的提示和警告。

```
$ git mergetool  
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge g  
Merging:  
doc/README.txt  
README  
readme.txt  
  
mv: 无法获取"doc/README.txt" 的文件状态(stat): 没有那个文件或目录  
cp: 无法获取"./doc/README.txt.BACKUP.13869.txt" 的文件状态(stat): 没有那个文件  
mv: 无法将".merge_file_I3gfzy" 移动至"./doc/README.txt.BASE.13869.txt": 没有
```

- 询问对文件:file:`doc/README.txt`的处理方式。输入d选择将该文件删除。

```
Deleted merge conflict for 'doc/README.txt':  
{local}: deleted  
{remote}: deleted  
Use (m)odified or (d)eleted file, or (a)bort? d
```

- 询问对文件:`:file:`README``的处理方式。输入c选择将该文件保留（创建）。

```
Deleted merge conflict for 'README':  
  {local}: created  
  {remote}: deleted  
Use (c)reated or (d)eleted file, or (a)abort? c
```

- 询问对文件:`:file:`readme.txt``的处理方式。输入d选择将该文件删除。

```
Deleted merge conflict for 'readme.txt':  
  {local}: deleted  
  {remote}: created  
Use (c)reated or (d)eleted file, or (a)abort? d
```

- 查看当前状态，只有一些尚未清理的临时文件，而冲突已经解决。

```
$ git status -s  
?? .merge_file_I3gfzy  
?? README.orig
```

- 提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."  
[master e070bc9] fixed tree conflict.
```

- 向共享服务器推送。

```
$ git push  
Counting objects: 5, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 457 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
To file:///path/to/repos/shared.git  
 615c1ff..e070bc9 master -> master
```

## 合并策略

Git合并操作支持很多合并策略，默认会选择最适合的合并策略。例如，和一个分支进行合并时会选择`recursive`合并策略，当和两个或两个以上的其他分支进行合并时采

用octopus合并策略。可以通过传递参数使用指定的合并策略，命令行如下：

```
git merge [-s <strategy>] [-X <strategy-option>] <commit>...
```

其中参数-s用于设定合并策略，参数-X用于为所选的合并策略提供附加的参数。

下面分别介绍不同的合并策略：

- **resolve**

该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略被认为是最安全、最快的合并策略。

- **recursive**

该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略是合并两个头指针时的默认合并策略。

当合并的头指针拥有一个以上的祖先的时候，会针对多个公共祖先创建一个合并的树，并以此作为三向合并的参照。这个合并策略被认为可以实现冲突的最小化，而且可以发现和处理由于重命名导致的合并冲突。

这个合并策略可以使用下列选项。

- **ours**

在遇到冲突的时候，选择我们的版本（当前分支的版本），而忽略他人的版本。如果他人的改动和本地改动不冲突，会将他人改动合并进来。

不要将此模式和后面介绍的单纯的**ours**合并策略相混淆。后面介绍的**ours**合并策略直接丢弃其他分支的变更，无论冲突与否。

- **theirs**

和**ours**选项相反，遇到冲突时选择他人的版本，丢弃我们的版本。

- **subtree[=path]**

这个选项使用子树合并策略，比下面介绍的**subtree**（子树合并）策略的定制能力更强。下面的**subtree**合并策略要对两个树的目录移动进行猜测，而**recursive**合并策略可以通过此参数直接对子树目录进行设置。

- **octopus**

可以合并两个以上的头指针，但是拒绝执行需要手动解决的复杂合并。主要的用途是

将多个主题分支合并到一起。这个合并策略是对三个及以上头指针进行合并时的默认合并策略。

- ours

可以合并任意数量的头指针，但是合并的结果总是使用当前分支的内容，丢弃其他分支的内容。

- subtree

这是一个经过调整的recursive策略。当合并树A和B时，如果B和A的一个子树相同，B首先进行调整以匹配A的树的结构，以免两棵树在同一级别进行合并。同时也针对两棵树的共同祖先进行调整。

关于子树合并在第4篇的第24章“子树合并”中详细介绍。

## 合并相关的设置

可以通过`:command:`git config``命令设置与合并相关的环境变量，对合并进行配置。下面是一些常用的设置。

- merge.conflictstyle

该变量定义冲突文件的显示风格，有两个可用的风格，默认的“merge”或“diff3”。

默认的“merge”风格使用标准的冲突分界符（<<<<<、=====、>>>>>）对冲突内容进行标识，其中的两个文字块分别是本地的修改和他人的修改。

如果使用“diff3”风格，则会在冲突中出现三个文字块，分别是：<<<<<和||||||之间的本地更改版本、||||||和=====之间的原始（共同祖先）版本和=====和>>>>>之间的他人更改的版本。例如：

```
User1 hacked.  
<<<<< HEAD  
Hello, user2.  
|||||| merged common ancestors  
Hello.  
=====  
Hello, user1.  
>>>>> a123390b8936882bd53033a582ab540850b6b5fb  
User2 hacked.  
User2 hacked again.
```

- merge.tool

执行`:command:`git mergetool``进行冲突解决时调用的图形化工具。变量`merge.tool`可以设置为如下内置支持的工具：“kdiff3”、“tkdiff”、“meld”、“xxdiff”、“emerge”、“vimdiff”、“gvimdiff”、“diffuse”、“ecmerge”、“tortoiseMerge”、“p4merge”、“araxis”和“opendiff”。

```
$ git config --global merge.tool kdiff3
```

如果将`merge.tool`设置为其他值，则使用自定义工具进行冲突解决。自定义工具需要通过`mergetool.<tool>.cmd`对自定义工具的命令行进行设置。

- `mergetool.<tool>.path`

如果`:command:`git mergetool``支持的冲突解决工具安装在特殊位置，可以使`mergetool.<tool>.path`对工具`<tool>`的安装位置进行设置。例如：

```
$ git config --global mergetool.kdiff3.path /path/to/kdiff3
```

- `mergetool.<tool>.cmd`

如果所用的冲突解决工具不在内置的工具列表中，还可以使`mergetool.<tool>.cmd`对自定义工具的命令行进行设置，同时要将`merge.tool`设置为`<tool>`。

自定义工具的命令行可以使用Shell变量。例如：

```
$ git config --global merge.tool mykdiff3
$ git config --global mergetool.mykdiff3.cmd '/usr/bin/kdiff3
-L1 "$MERGED (Base)" -L2 "$MERGED (Local)" -L3 "$MERGED (Remote)"
--auto -o "$MERGED" "$BASE" "$LOCAL" "$REMOTE"'
```

- `merge.log`

是否在合并提交的提交说明中包含合并提交的概要信息。默认为`false`。

来源：<https://github.com/gotgit/gotgit/blob/master/03-git-harmony/020-conflict.rst>

# Git里程碑

里程碑即Tag，是人为对提交进行的命名。这和Git的提交ID是否太长无关，使用任何数字版本号无论长短，都没有使用一个直观的表意的字符串来得方便。例如：用里程碑名称“v2.1”对应于软件的2.1发布版本就比使用提交ID要直观得多。

对于里程碑，实际上我们并不陌生，在第2篇的“第10章 Git基本操作”中，就介绍了使用里程碑来对工作进度“留影”纪念，并使用`:command:`git describe``命令显示里程碑和提交ID的组合来代表软件的版本号。本章将详细介绍里程碑的创建、删除和共享，还会介绍里程碑存在的三种不同形式：轻量级里程碑、带注释的里程碑和带签名的里程碑。

接下来的三章，将对一个使用Hello, World作为示例程序的版本库进行研究，这个版本库不需要我们从头建立，可以直接从Github上克隆。先使用下面的方法在本地创建一个镜像，用作本地用户的共享版本库。

- 进入本地版本库根目录下。

```
$ mkdir -p /path/to/repos/  
$ cd /path/to/repos/
```

- 从Github上镜像hello-world.git版本库。

如果Git是1.6.0或更新的版本，可以使用下面的命令建立版本库镜像。

```
$ git clone --mirror git://github.com/ossxp-com/hello-world.git
```

否则使用下面的命令建立版本库镜像。

```
$ git clone --bare \  
git://github.com/ossxp-com/hello-world.git \  
hello-world.git
```

完成上面操作后，就在本地建立了一个裸版本库`:file:`/path/to/repos/hello-world.git``。接下来用户user1和user2分别在各自工作区克隆这个裸版本库。使用如下命令即可：

```
$ git clone file:///path/to/repos/hello-world.git \  
/path/to/user1/workspace/hello-world  
$ git clone file:///path/to/repos/hello-world.git \  
/path/to/user2/workspace/hello-world
```

```
$ git --git-dir=/path/to/user1/workspace/hello-world/.git \
    config user.name user1
$ git --git-dir=/path/to/user1/workspace/hello-world/.git \
    config user.email user1@sun.osssp.com
$ git --git-dir=/path/to/user2/workspace/hello-world/.git \
    config user.name user2
$ git --git-dir=/path/to/user2/workspace/hello-world/.git \
    config user.email user2@moon.osssp.com
```

## 显示里程碑

里程碑可以使用`:command:`git tag``命令来显示，里程碑还可以在其他命令的输出中出现，下面分别对这些命令加以介绍。

### 1. 命令`:command:`git tag``

不带任何参数执行`:command:`git tag``命令，即可显示当前版本库的里程碑列表。

```
$ cd /path/to/user1/workspace/hello-world
$ git tag
jx/v1.0
jx/v1.0-i18n
jx/v1.1
jx/v1.2
jx/v1.3
jx/v2.0
jx/v2.1
jx/v2.2
jx/v2.3
```

里程碑创建的时候可能包含一个说明。在显示里程碑的时候同时显示说明，使用`-n<num>`参数，显示最多`<num>`行里程碑的说明。

```
$ git tag -n1
jx/v1.0      Version 1.0
jx/v1.0-i18n i18n support for v1.0
jx/v1.1      Version 1.1
jx/v1.2      Version 1.2: allow spaces in username.
jx/v1.3      Version 1.3: Hello world speaks in Chinese now.
jx/v2.0      Version 2.0
jx/v2.1      Version 2.1: fixed typo.
jx/v2.2      Version 2.2: allow spaces in username.
jx/v2.3      Version 2.3: Hello world speaks in Chinese now.
```

还可以使用通配符对显示进行过滤。只显示名称和通配符相符的里程碑。

```
$ git tag -l jx/v2*
jx/v2.0
jx/v2.1
jx/v2.2
jx/v2.3
```

## 2. 命令: command: `git log`

在查看日志时使用参数`--decorate`可以看到提交对应的里程碑及其他引用。

```
$ git log --oneline --decorate
3e6070e (HEAD, tag: jx/v1.0, origin/master, origin/HEAD, master) Show version
75346b3 Hello world initialized.
```

## 3. 命令: command: `git describe`

使用命令`:command:`git describe``将提交显示为一个易记的名称。这个易记的名称来自于建立在该提交上的里程碑，若该提交没有里程碑则使用该提交历史版本上的里程碑并加上可理解的寻址信息。

- 如果该提交恰好被打上一个里程碑，则显示该里程碑的名字。

```
$ git describe
jx/v1.0
$ git describe 384f1e0
jx/v2.2
```

- 若提交没有对应的里程碑，但是在其祖先版本上建有里程碑，则使用类似`<tag>-<num>-g<commit>`的格式显示。

其中`<tag>`是最接近的祖先提交的里程碑名字，`<num>`是该里程碑和提交之间的距离，`<commit>`是该提交的精简提交ID。

```
$ git describe 610e78fc95bf2324dc5595fa684e08e1089f5757
jx/v2.2-1-g610e78f
```

- 如果工作区对文件有修改，还可以通过后缀`-dirty`表示出来。

```
$ echo hacked >> README; git describe --dirty; git checkout -- README  
jx/v1.0-dirty
```

- 如果提交本身没有包含里程碑，可以通过传递`--always`参数显示精简提交ID，否则出错。

```
$ git describe master^ --always  
75346b3
```

命令`:command:`git describe``是非常有用的命令，可以将该命令的输出用作软件的版本号。在之前曾经演示过这个应用，马上还会看到。

## 4. 命令`:command:`git name-rev``

命令`:command:`git name-rev``和`:command:`git describe``类似，会显示提交ID及其对应的一个引用。默认优先使用分支名，除非使用`:command:`--tags``参数。还有一个显著的不同是，如果提交上没有相对应的引用，则会使用最新提交上的引用名称并加上向后回溯的符号`:command:`^~<num>``。

- 默认优先显示分支名。

```
$ git name-rev HEAD  
HEAD master
```

- 使用`--tags`优先使用里程碑。

之所以对应的里程碑引用名称后面加上后缀`^0`，是因为该引用指向的是一个tag对象而非提交。用`^0`后缀指向对应的提交。

```
$ git name-rev HEAD --tags  
HEAD tags/jx/v1.0^0
```

- 如果提交上没有对应的引用名称，则会使用新提交上的引用名称并加上后缀`:command:`^~<num>``。后缀的含义是第`<num>`个祖先提交。

```
$ git name-rev --tags 610e78fc95bf2324dc5595fa684e08e1089f5757  
610e78fc95bf2324dc5595fa684e08e1089f5757 tags/jx/v2.3~1
```

- 命令`:command:`git name-rev``可以对标准输入中的提交ID进行改写，使用管道符号对前一个命令的输出进行改写，会显示神奇的效果。

```
$ git log --pretty=oneline origin/helper/master | git name-rev --tags --s  
bb4fef88fee435bfac04b8389cf193d9c04105a6 (tags/jx/v2.3^0) Translate for C  
610e78fc95bf2324dc5595fa684e08e1089f5757 (tags/jx/v2.3~1) Add I18N suppor  
384f1e0d5106c9c6033311a608b91c69332fe0a8 (tags/jx/v2.2^0) Bugfix: allow s  
e5e62107f8f8d0a5358c3aff993cf874935bb7fb (tags/jx/v2.1^0) fixed typo: -he  
5d7657b2f1a8e595c01c812dd5b2f67ea133f456 (tags/jx/v2.0^0) Parse arguments  
3e6070eb2062746861b20e1e6235fed6f6d15609 (tags/jx/v1.0^0) Show version.  
75346b3283da5d8117f3fe66815f8aaaf5387321 (tags/jx/v1.0~1) Hello world ini
```



## 创建里程碑

创建里程碑依然是使用`:command:`git tag``命令。创建里程碑的用法有以下几种：

```
用法1: git tag <tagname> [<commit>]  
用法2: git tag -a <tagname> [<commit>]  
用法3: git tag -m <msg> <tagname> [<commit>]  
用法4: git tag -s <tagname> [<commit>]  
用法5: git tag -u <key-id> <tagname> [<commit>]
```

其中：

- 用法1是创建轻量级里程碑。
- 用法2和用法3相同，都是创建带说明的里程碑。其中用法3直接通过`-m`参数提供里程碑创建说明。
- 用法4和用法5相同，都是创建带GPG签名的里程碑。其中用法5用`-u`参数选择指定的私钥进行签名。
- 创建里程碑需要输入里程碑的名字`<tagname>`和一个可选的提交ID`<commit>`。如果没有提供提交ID，则基于头指针`HEAD`创建里程碑。

## 轻量级里程碑

轻量级里程碑最简单，创建时无须输入描述信息。我们来看看如何创建轻量级里程碑：

- 先创建一个空提交。

```
$ git commit --allow-empty -m "blank commit."  
[master 60a2f4f] blank commit.
```

- 在刚刚创建的空提交上创建一个轻量级里程碑，名为`mytag`。

省略了`<commit>`参数，相当于在`HEAD`上即最新的空提交上创建里程碑。

```
$ git tag mytag
```

- 查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my*
mytag
```

### 轻量级里程碑的奥秘

当创建了里程碑mytag后，会在版本库的:`:file:`.git/refs/tags``目录下创建一个新文件。

- 查看一下这个引用文件的内容，会发现是一个40位的SHA1哈希值。

```
$ cat .git/refs/tags/mytag
60a2f4f31e5ddd777c6ad37388fe6e5520734cb
```

- 用:`:command:`git cat-file``命令检查轻量级里程碑指向的对象。轻量级里程碑实际上指向的是一个提交。

```
$ git cat-file -t mytag
commit
```

- 查看该提交的内容，发现就是刚刚进行的空提交。

```
$ git cat-file -p mytag
tree 1d902fedc4eb732f17e50f111dcecb638f10313e
parent 3e6070eb2062746861b20e1e6235fed6f6d15609
author user1 <user1@sun.ossxp.com> 1293790794 +0800
committer user1 <user1@sun.ossxp.com> 1293790794 +0800

blank commit.
```

### 轻量级里程碑的缺点

轻量级里程碑的创建过程没有记录，因此无法知道是谁创建的里程碑，是何时创建的里程碑。在团队协同开发时，尽量不要采用此种偷懒的方式创建里程碑，而是采用后两种方式。

还有:`:command:`git describe``命令默认不使用轻量级里程碑生成版本描述字符串。

- 执行:`:command:`git describe``命令，发现生成的版本描述字符串，使用的是前一个版

本上的里程碑名称。

```
$ git describe  
jx/v1.0-1-g60a2f4f
```

- 使用`--tags`参数，也可以将轻量级里程碑用作版本描述符。

```
$ git describe --tags  
mytag
```

## 带说明的里程碑

带说明的里程碑，就是使用参数`-a`或者`-m <msg>`调用`:command:`git tag``命令，在创建里程碑的时候提供一个关于该里程碑的说明。下面来看看如何创建带说明的里程碑：

- 还是先创建一个空提交。

```
$ git commit --allow-empty -m "blank commit for annotated tag test."  
[master 8a9f3d1] blank commit for annotated tag test.
```

- 在刚刚创建的空提交上创建一个带说明的里程碑，名为`mytag2`。

下面的命令使用了`-m <msg>`参数在命令行给出了新建里程碑的说明。

```
$ git tag -m "My first annotated tag." mytag2
```

- 查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my* -n1  
mytag          blank commit.  
mytag2         My first annotated tag.
```

## 带说明里程碑的奥秘

当创建了带说明的里程碑`mytag2`后，会在版本库的`:file:`.git/refs/tags``目录下创建一个新的引用文件。

- 查看一下这个引用文件的内容：

```
$ cat .git/refs/tags/mytag2
```

```
149b6344e80fc190bda5621cd71df391d3dd465e
```

- 用`:command:`git cat-file``命令检查该里程碑（带说明的里程碑）指向的对象，会发现指向的不再是一个提交，而是一个 tag 对象。

```
$ git cat-file -t mytag2
tag
```

- 查看该提交的内容，会发现mytag2对象的内容不是之前我们熟悉的提交对象的内容，而是包含了创建里程碑时的说明，以及对应的提交ID等信息。

```
$ git cat-file -p mytag2
object 8a9f3d16ce2b4d39b5d694de10311207f289153f
type commit
tag mytag2
tagger user1 <user1@sun.ossxp.com> Sun Jan 2 14:10:07 2011 +0800

My first annotated tag.
```

由此可见使用带说明的里程碑，会在版本库中建立一个新的对象（tag对象），这个对象会记录创建里程碑的用户（tagger），创建里程碑的时间，以及为什么要创建里程碑。这就避免了轻量级里程碑因为匿名创建而无法追踪的缺点。

带说明的里程碑是一个tag对象，在版本库中以一个对象的方式存在，并用一个40位的SHA1哈希值来表示。这个哈希值的生成方法和前面介绍的commit对象、tree对象、blob对象一样。至此，Git对象库的四类对象我们就都已经研究到了。

```
$ git cat-file tag mytag2 | wc -c
148
$ (printf "tag 148\000"; git cat-file tag mytag2) | sha1sum
149b6344e80fc190bda5621cd71df391d3dd465e -
```

虽然mytag2本身是一个tag对象，但在很多Git命令中，可以直接将其视为一个提交。下面的`:command:`git log``命令，显示mytag2指向的提交日志。

```
$ git log -1 --pretty=oneline mytag2
8a9f3d16ce2b4d39b5d694de10311207f289153f blank commit for annotated tag test.
```

有时，需要得到里程碑指向的提交对象的SHA1哈希值。

- 直接用`:command:`git rev-parse``命令查看mytag2得到的是tag对象的ID，并非提交对

象的ID。

```
$ git rev-parse mytag2  
149b6344e80fc190bda5621cd71df391d3dd465e
```

- 使用下面几种不同的表示法，则可以获得mytag2对象所指向的提交对象的ID。

```
$ git rev-parse mytag2^{commit}  
8a9f3d16ce2b4d39b5d694de10311207f289153f  
$ git rev-parse mytag2^{}  
8a9f3d16ce2b4d39b5d694de10311207f289153f  
$ git rev-parse mytag2^0  
8a9f3d16ce2b4d39b5d694de10311207f289153f  
$ git rev-parse mytag2~0  
8a9f3d16ce2b4d39b5d694de10311207f289153f
```

## 带签名的里程碑

带签名的里程碑和上面介绍的带说明的里程碑本质上是一样的，都是在创建里程碑的时候在Git对象库中生成一个tag对象，只不过带签名的里程碑多做了一个工作：为里程碑对象添加GnuPG签名。

创建带签名的里程碑也非常简单，使用参数-s或-u <key-id>即可。还可以使用-m <msg>参数直接在命令行中提供里程碑的描述。创建带签名里程碑的一个前提是需要安装GnuPG，并且建立相应的公钥/私钥对。

GnuPG可以在各个平台上安装。

- 在Linux，如Debian/Ubuntu上安装，执行：

```
$ sudo aptitude install gnupg
```

- 在Mac OS X上，可以通过Homebrew安装：

```
$ brew install gnupg
```

- 在Windows上可以通过cygwin安装gnupg。

为了演示创建带签名的里程碑，还是事先创建一个空提交。

```
$ git commit --allow-empty -m "blank commit for GnuPG-signed tag test."  
[master ebcf6d6] blank commit for GnuPG-signed tag test.
```

直接在刚刚创建的空提交上创建一个带签名的里程碑mytag2很可能会失败。

```
$ git tag -s -m "My first GPG-signed tag." mytag3
gpg: “user1 <user1@sun.ossxp.com>”已跳过: 私钥不可用
gpg: signing failed: 私钥不可用
error: gpg failed to sign the tag
error: unable to sign the tag
```

之所以签名失败，是因为找不到签名可用的公钥/私钥对。使用下面的命令可以查看当前可用的GnuPG公钥。

```
$ gpg --list-keys
/home/jiangxin/.gnupg/pubring.gpg
-----
pub    1024D/FBC49D01 2006-12-21 [有效至: 2016-12-18]
uid          Jiang Xin <worldhello.net@gmail.com>
uid          Jiang Xin <jiangxin@ossxp.com>
sub    2048g/448713EB 2006-12-21 [有效至: 2016-12-18]
```

可以看到GnuPG的公钥链（pubring）中只包含了Jiang Xin用户的公钥，尚没有user1用户的公钥。

实际上在创建带签名的里程碑时，并非一定要使用邮件名匹配的公钥/私钥对进行签名，使用`-u <key-id>`参数调用就可以用指定的公钥/私钥对进行签名，对此例可以使用FBC49D01作为`<key-id>`。但如果沒有可用的公钥/私钥对，或者希望使用提交者本人的公钥/私钥对进行签名，就需要为提交者`:user1 <user1@sun.ossxp.com>`创建对应的公钥/私钥对。

使用命令`:command:`gpg --gen-key``来创建公钥/私钥对。

```
$ gpg --gen-key
```

按照提示一步一步操作即可。需要注意的有：

- 在创建公钥/私钥对时，会提示输入用户名，输入`User1`，提示输入邮件地址，输入`user1@sun.ossxp.com`，其他可以采用默认值。
- 在提示输入密码时，为了简单起见可以直接按下回车，即使用空口令。
- 在生成公钥私钥对过程中，会提示用户做一些随机操作以便产生更好的随机数，这时不停的晃动鼠标就可以了。

创建完毕，再查看一下公钥链。

```
$ gpg --list-keys  
/home/jiangxin/.gnupg/pubring.gpg  
-----  
pub 1024D/FBC49D01 2006-12-21 [有效至: 2016-12-18]  
uid Jiang Xin <worldhello.net@gmail.com>  
uid Jiang Xin <jiangxin@osssxp.com>  
sub 2048g/448713EB 2006-12-21 [有效至: 2016-12-18]  
  
pub 2048R/37379C67 2011-01-02  
uid User1 <user1@sun.osssxp.com>  
sub 2048R/2FCFB3E2 2011-01-02
```

很显然用户user1的公钥私钥对已经建立。现在就可以直接使用-s参数来创建带签名里程碑了。

```
$ git tag -s -m "My first GPG-signed tag." mytag3
```

查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my* -n1  
mytag      blank commit.  
mytag2     My first annotated tag.  
mytag3     My first GPG-signed tag.
```

和带说明的里程碑一样，在Git对象库中也建立了一个tag对象。查看该tag对象可以看到其中包含了GnuPG签名。

```
$ git cat-file tag mytag3  
object ebcf6d6b06545331df156687ca2940800a3c599d  
type commit  
tag mytag3  
tagger user1 <user1@sun.osssxp.com> 1293960936 +0800  
  
My first GPG-signed tag.  
-----BEGIN PGP SIGNATURE-----  
Version: GnuPG v1.4.10 (GNU/Linux)  
  
iQEcBAABAgAGBQJNIEboAAoJE09W1fg3N5xn42gH/jFDEKobqlupNKFvmkI1t9d6  
1ApDFUdcFMPWvxo/eq8VjcQyRcb1X1bGJj+pxXk455fDL1NWonaJa6HE6RLu868x  
CQIWqWelkCelfm05GE9FnPd2SmJsiDkTPZzINya1Hy1F5ZbrExH506JyCFk//FC2  
8zRApSbrsj3yAWMStW0fGqHKLuYq+sdepzGnnFnhhzkJhusMHUkTIfpLwaprhMsm  
1IIxKNm9i0Zf/tzq4a/R0N8NiFH1/9M95iV200I9PuuRWedV0tEPS6Onax2yT3JE  
I/w9gtIB0eb5uAz2Xrt5AUwt9JJTk5mmv2HBqWCq5wefxs/ub26iPmef35PwAgA=
```

```
-----END PGP SIGNATURE-----
```

要验证签名的有效性，如果直接使用gpg命令会比较麻烦，因为需要将这个文件拆分为两个，一个是不包含签名的里程碑内容，另外一个是签名本身。还好可以使用命令:`git tag -v`来验证里程碑签名的有效性。

```
$ git tag -v mytag3
object ebcf6d6b06545331df156687ca2940800a3c599d
type commit
tag mytag3
tagger user1 <user1@sun.ossexp.com> 1293960936 +0800

My first GPG-signed tag.
gpg: 于 2011年01月02日 星期日 17时35分36秒 CST 创建的签名，使用 RSA，钥匙号 37379C6
```

## 删除里程碑

如果里程碑建立在了错误的提交上，或者对里程碑的命名不满意，可以删除里程碑。删除里程碑使用命令:`git tag -d`，下面用命令删除里程碑mytag。

```
$ git tag -d mytag
Deleted tag 'mytag' (was 60a2f4f)
```

里程碑没有类似reflog的变更记录机制，一旦删除不易恢复，慎用。在删除里程碑mytag的命令输出中，会显示该里程碑所对应的提交ID，一旦发现删除错误，赶紧补救还来得及。下面的命令实现对里程碑mytag的重建。

```
$ git tag mytag 60a2f4f
```

### 为什么没有重命名里程碑的命令？

Git没有提供对里程碑直接重命名的命令，如果对里程碑名字不满意的话，可以删除旧的里程碑，然后重新用新的里程碑进行命名。

为什么没有提供重命名里程碑的命令呢？按理说只要将:`.git/refs/tags/`下的引用文件改名就可以了。这是因为里程碑的名字不但反映在:`.git/refs/tags/`引用目录下的文件名，而且对于带说明或签名的里程碑，里程碑的名字还反映在tag对象的内容中。尤其是带签名的里程碑，如果修改里程碑的名字，不但里程碑对象ID势必要变化，而且里程碑也要重新进行签名，这显然难以自动实现。

在第6篇第35章的“Git版本库整理”一节中会介绍使用:command:`git filter-branch`命令实现对里程碑自动重命名的方法，但是那个方法也不能毫发无损地实现对签名里程碑的重命名，被重命名的签名里程碑中的签名会被去除从而成为带说明的里程碑。

## 不要随意更改里程碑

里程碑建立后，如果需要修改，可以使用同样的里程碑名称重新建立，不过需要加上-f或--force参数强制覆盖已有的里程碑。

更改里程碑要慎重，一个原因是里程碑从概念上讲是对历史提交的一个标记，不应该随意变动。另外一个原因是里程碑一旦被他人同步，如果修改里程碑，已经同步该里程碑的用户不会自动更新，这就导致一个相同名称的里程碑在不同用户的版本库中的指向不同。下面就看看如何与他人共享里程碑。

## 共享里程碑

现在看看用户user1的工作区状态。可以看出现在的工作区相比上游有三个新的提交。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#
nothing to commit (working directory clean)
```

那么如果执行:command:`git push`命令向上游推送，会将本地创建的三个里程碑推送到上游么？通过下面的操作来试一试。

- 向上游推送。

```
$ git push
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 512 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/hello-world.git
  3e6070e..ebcf6d6  master -> master
```

- 通过执行:command:`git ls-remote`可以查看上游版本库的引用，会发现本地建立的三个里程碑，并没有推送到上游。

```
$ git ls-remote origin my*
```

创建的里程碑， 默认只在本地版本库中可见， 不会因为对分支执行推送而将里程碑也推送到远程版本库。这样的设计显然更为合理， 否则的话， 每个用户本地创建的里程碑都自动向上游推送， 那么上游的里程碑将有多么杂乱， 而且不同用户创建的相同名称的里程碑会互相覆盖。

### 那么如何共享里程碑呢？

如果用户确实需要将某些本地建立的里程碑推送到远程版本库， 需要在`:command:`git push``命令中明确地表示出来。下面在用户user1的工作区执行命令， 将mytag里程碑共享到上游版本库。

```
$ git push origin mytag
Total 0 (delta 0), reused 0 (delta 0)
To file:///path/to/repos/hello-world.git
 * [new tag]           mytag -> mytag
```

如果需要将本地建立的所有里程碑全部推送到远程版本库， 可以使用通配符。

```
$ git push origin refs/tags/*
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 687 bytes, done.
Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
To file:///path/to/repos/hello-world.git
 * [new tag]           mytag2 -> mytag2
 * [new tag]           mytag3 -> mytag3
```

再用命令`:command:`git ls-remote``查看上游版本库的引用， 会发现本地建立的三个里程碑， 已经能够在上游中看到了。

```
$ git ls-remote origin my*
60a2f4f31e5ddd777c6ad37388fe6e5520734cb      refs/tags/mytag
149b6344e80fc190bda5621cd71df391d3dd465e      refs/tags/mytag2
8a9f3d16ce2b4d39b5d694de10311207f289153f      refs/tags/mytag2^{}
5dc2fc52f2dc84987f511481cc6b71ec1b381f7      refs/tags/mytag3
ebcf6d6b06545331df156687ca2940800a3c599d      refs/tags/mytag3^{}
```

### 用户从版本库执行拉回操作， 会自动获取里程碑么？

用户 user2 的工作区中如果执行:command:`git fetch` 或:command:`git pull` 操作，能自动将用户 user1 推送到共享版本库中的里程碑获取到本地版本库么？下面实践一下。

- 进入user2的工作区。

```
$ cd /path/to/user2/workspace/hello-world/
```

- 执行:command:`git pull` 命令，从上游版本库获取提交。

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From file:///path/to/repos/hello-world
  3e6070e..ebcf6d6  master      -> origin/master
 * [new tag]        mytag3      -> mytag3
From file:///path/to/repos/hello-world
 * [new tag]        mytag       -> mytag
 * [new tag]        mytag2      -> mytag2
Updating 3e6070e..ebcf6d6
Fast-forward
```

- 可见执行:command:`git pull` 操作，能够在获取远程共享版本库的提交的同时，获取新的里程碑。下面的命令可以看到本地版本库中的里程碑。

```
$ git tag -n1 -l my*
mytag          blank commit.
mytag2         My first annotated tag.
mytag3         My first GPG-signed tag.
```

## 里程碑变更能够自动同步么？

里程碑可以被强制更新。当里程碑被改变后，已经获取到里程碑的版本库再次使用获取或拉回操作，能够自动更新里程碑么？答案是不能。可以看看下面的操作。

- 用户user2强制更新里程碑mytag2。

```
$ git tag -f -m "user2 update this annotated tag." mytag2 HEAD^
Updated tag 'mytag2' (was 149b634)
```

- 里程碑mytag2已经是不同的对象了。

```
$ git rev-parse mytag2  
0e6c780ff0fe06635394db9dac6fb494833df8df  
$ git cat-file -p mytag2  
object 8a9f3d16ce2b4d39b5d694de10311207f289153f  
type commit  
tag mytag2  
tagger user2 <user2@moon.ossxp.com> Mon Jan 3 01:14:18 2011 +0800  
  
user2 update this annotated tag.
```

- 为了更改远程共享服务器中的里程碑，同样需要显式推送。即在推送时写上要推送的里程碑名称。

```
$ git push origin mytag2  
Counting objects: 1, done.  
Writing objects: 100% (1/1), 171 bytes, done.  
Total 1 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (1/1), done.  
To file:///path/to/repos/hello-world.git  
149b634..0e6c780 mytag2 -> mytag2
```

- 切换到另外一个用户user1的工作区。

```
$ cd /path/to/user1/workspace/hello-world/
```

- 用户user1执行拉回操作，没有获取到新的里程碑。

```
$ git pull  
Already up-to-date.
```

- 用户user1必须显式地执行拉回操作。即要在:command:`git pull`的参数中使用引用表达式。

所谓引用表达式就是用冒号分隔的引用名称或通配符。用在这里代表用远程共享版本库的引用refs/tag/mytag2覆盖本地版本库的同名引用。

```
$ git pull origin refs/tags/mytag2:refs/tags/mytag2  
remote: Counting objects: 1, done.  
remote: Total 1 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (1/1), done.  
From file:///path/to/repos/hello-world  
- [tag update]      mytag2      -> mytag2  
Already up-to-date.
```

关于里程碑的共享和同步操作，看似很繁琐，但用心体会就会感觉到Git关于里程碑共享的设计是非常合理和人性化的：

- 里程碑共享，必须显式的推送。即在推送命令的参数中，标明要推送哪个里程碑。

显式推送是防止用户随意推送里程碑导致共享版本库中里程碑泛滥的方法。当然还可以参考第5篇“第30章Gitolite服务架设”的相关章节为共享版本库添加授权，只允许部分用户向服务器推送里程碑。

- 执行获取或拉回操作，自动从远程版本库获取新里程碑，并在本地版本库中创建。

获取或拉回操作，只会将获取的远程分支所包含的新里程碑同步到本地，而不会将远程版本库的其他分支中的里程碑获取到本地。这既方便了里程碑的取得，又防止本地里程碑因同步远程版本库而泛滥。

- 如果本地已有同名的里程碑，默认不会从上游同步里程碑，即使两者里程碑的指向是不同的。

理解这一点非常重要。这也要求里程碑一旦共享，就不要再修改。

## 删除远程版本库的里程碑

假如向远程版本库推送里程碑后，忽然发现里程碑创建在了错误的提交上，为了防止其他人获取到错误的里程碑，应该尽快将里程碑删除。

删除本地里程碑非常简单，使用`:command:`git tag -d <tagname>``就可以了，但是如何撤销已经推送到远程版本库的里程碑呢？需要登录到服务器上么？或者需要麻烦管理员么？不必！可以直接在本地版本库执行命令删除远程版本库的里程碑。

使用`:command:`git push``命令可以删除远程版本库中的里程碑。用法如下：

```
命令: git push <remote_url> :<tagname>
```

该命令的最后一个参数实际上是一个引用表达式，引用表达式一般的格式为`<ref>:<ref>`。该推送命令使用的引用表达式冒号前的引用被省略，其含义是将一个空值推送到远程版本库对应的引用中，亦即删除远程版本库中相关的引用。这个命令不但可以用于删除里程碑，在下一章还可以用它删除远程版本库中的分支。

下面演示在用户user1的工作区执行下面的命令删除远程共享版本库中的里程碑mytag2。

- 切换到用户user1工作区。

```
$ cd /path/to/user1/workspace/hello-world
```

- 执行推送操作删除远程共享版本库中的里程碑。

```
$ git push origin :mytag2
To file:///path/to/repos/hello-world.git
 - [deleted]           mytag2
```

- 查看远程共享库中的里程碑，发现mytag2的确已经被删除。

```
$ git ls-remote origin my*
60a2f4f31e5ddd777c6ad37388fe6e5520734cb      refs/tags/mytag
5dc2fc52f2dcb84987f511481cc6b71ec1b381f7      refs/tags/mytag3
ebcf6d6b06545331df156687ca2940800a3c599d      refs/tags/mytag3^{}
```

## 里程碑命名规范

在正式项目的版本库管理中，要为里程碑创建订立一些规则，诸如：

- 对创建里程碑进行权限控制，参考后面Git服务器架设的相关章节。
- 不能使用轻量级里程碑（只用于本地临时性里程碑），必须使用带说明的里程碑，甚至要求必须使用带签名的里程碑。
- 如果使用带签名的里程碑，可以考虑设置专用账户，使用专用的私钥创建签名。
- 里程碑的命名要使用统一的风格，并很容易和最终产品显示的版本号相对应。

Git的里程碑命名还有一些特殊的约定需要遵守。实际上，下面的这些约定对于下一章要介绍的分支及任何其他引用均适用：

- 不能以符号“-”开头。以免在命令行中被当成命令的选项。
- 可以包含路径分隔符“/”，但是路径分隔符不能位于最后。

使用路径分隔符创建tag实际上会在引用目录下创建子目录。例如名为demo/v1.2.1的里程碑，就会创建目录:`file:`.git/refs/tags/demo``并在该目录下创建引用文件v1.2.1。

- 不能出现两个连续的点“..”。因为两个连续的点被用于表示版本范围，当然更不能使用三个连续的点。
- 如果在里程碑命名中使用了路径分隔符“/”，就不能在任何一个分隔路径中以点“.”开头。

这是因为里程碑在用简写格式表达时，可能造成以一个点“.”开头。这样的引用名称在用作版本范围的最后一个版本时，本来两点操作符变成了三点操作符，从而造成歧义。

- 不能在里程碑名称的最后出现点“.”。否则作为第一个参数出现在表示版本范围的表达式中时，本来版本范围表达式可能用的是两点操作符，结果被误作三点操作符。
- 不能使用特殊字符，如：空格、波浪线“~”、脱字符“^”、冒号“:”、问号“?”、星号“\*”、方括号“[”，以及字符\\177（删除字符）或小于\\040（32）的Ascii码都不能使用。

这是因为波浪线“~”和脱字符“^”都用于表示一个提交的祖先提交。

冒号被用作引用表达式来分隔两个不同的引用，或者用于分隔引用代表的树对象和该目录树中的文件。

问号、星号和方括号在引用表达式中都被用作通配符。

- 不能以“.lock”为结尾。因为以“.lock”结尾的文件是里程碑操作过程中的临时文件。
- 不能包含“@{”字串。因为reflog采用“@{<num>}”作为语法的一部分。
- 不能包含反斜线“\”。因为反斜线用于命令行或shell脚本会造成意外。

## Linux中的里程碑

Linux项目无疑是使用Git版本库时间最久远，也是最重量级的项目。研究Linux项目本身的里程碑命名和管理，无疑会为自己的项目提供借鉴。

- 首先看看Linux中的里程碑命名。可以看到里程碑都是以字母v开头。

```
$ git ls-remote --tags \
  git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6-stable.g
  v2.6.36*
  25427f38d3b791d986812cb81c68df38e8249ef8      refs/tags/v2.6.36
  f6f94e2ab1b33f0082ac22d71f66385a60d8157f      refs/tags/v2.6.36^{}
  8ed88d401f908a594cd74a4f2513b0fabd32b699      refs/tags/v2.6.36-rc1
  da5cabf80e2433131bf0ed8993abc0f7ea618c73      refs/tags/v2.6.36-rc1^{}
  ...
  7619e63f48822b2c68d0e108677340573873fb93      refs/tags/v2.6.36-rc8
  cd07202cc8262e1669edff0d97715f3dd9260917      refs/tags/v2.6.36-rc8^{}
  9d389cb6dcae347cfcdadfc2a1ec5e66fc7a667ea      refs/tags/v2.6.36.1
  bf6ef02e53e18dd14798537e530e00b80435ee86      refs/tags/v2.6.36.1^{}
  ee7b38c91f3d718ea4035a331c24a56553e90960      refs/tags/v2.6.36.2
  a1346c99fc89f2b3d35c7d7e2e4aef8ea4124342      refs/tags/v2.6.36.2^{}
```

- 以`-rc<num>`为后缀的是先于正式版发布的预发布版本。

可以看出这个里程碑是一个带签名的里程碑。关于此里程碑的说明也是再简练不过了。

```
$ git show v2.6.36-rc1
tag v2.6.36-rc1
Tagger: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Aug 15 17:42:10 2010 -0700

Linux 2.6.36-rc1
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

iEYEABECAAYFAkxoiWgACgkQF3YsRnbihLtyKQCfQSIVcj2hvLj6IWgP9xK2FE7T
bPoAniJ1CjbwLxQBudRi71FvubqPLuVC
=iuls
-----END PGP SIGNATURE-----

commit da5cabf80e2433131bf0ed8993abc0f7ea618c73
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Aug 15 17:41:37 2010 -0700

Linux 2.6.36-rc1

diff --git a/Makefile b/Makefile
index 788111d..f3bdff8 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 6
-SUBLEVEL = 35
-EXTRAVERSION =
+SUBLEVEL = 36
+EXTRAVERSION = -rc1
NAME = Sheep on Meth

# *DOCUMENTATION*
```

- 正式发布版去掉了预发布版的后缀。

```
$ git show v2.6.36
tag v2.6.36
Tagger: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Wed Oct 20 13:31:18 2010 -0700
```

```
Linux 2.6.36
```

The latest and greatest, and totally bug-free. At least until 2.6.37 comes along and shoves it under a speeding train like some kind of a bully.

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.10 (GNU/Linux)

```
iEYEAECAYFAky/UcwACgkQF3YsRnbHLvg/ACffKjAb1fD6fpqcHbSijHHpbP3  
4SkAnR4x0y7iKhmfS50ZrVs0kFFTuBHG
```

=JD3z

-----END PGP SIGNATURE-----

```
commit f6f94e2ab1b33f0082ac22d71f66385a60d8157f
```

Author: Linus Torvalds <torvalds@linux-foundation.org>

Date: Wed Oct 20 13:30:22 2010 -0700

```
Linux 2.6.36
```

```
diff --git a/Makefile b/Makefile  
index 7583116..860c26a 100644
```

--- a/Makefile

+++ b/Makefile

@@ -1,7 +1,7 @@

VERSION = 2

PATCHLEVEL = 6

SUBLEVEL = 36

-EXTRAVERSION = -rc8

+EXTRAVERSION =

NAME = Flesh-Eating Bats with Fangs

```
# *DOCUMENTATION*
```

- 正式发布后的升级/修正版本是通过最后一位数字的变动体现的。

```
$ git show v2.6.36.1
```

tag v2.6.36.1

Tagger: Greg Kroah-Hartman <gregkh@suse.de>

Date: Mon Nov 22 11:04:17 2010 -0800

```
This is the 2.6.36.1 stable release
```

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v2.0.15 (GNU/Linux)

```
iEYEAECAYFAkzqvrIACgkQMUfUDdst+ym9VQCgmE1LK2eC/LE9HkscsxL1X62P
```

```
8F0AnRI28EHENLXC+FBPt+AFWoT9f1N8
```

=BX50

-----END PGP SIGNATURE-----

```
commit bf6ef02e53e18dd14798537e530e00b80435ee86
Author: Greg Kroah-Hartman <gregkh@suse.de>
Date:   Mon Nov 22 11:03:49 2010 -0800

    Linux 2.6.36.1

diff --git a/Makefile b/Makefile
index 860c26a..dafd22a 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 36
-EXTRAVERSION =
+EXTRAVERSION = .1
NAME = Flesh-Eating Bats with Fangs

# *DOCUMENTATION*
```

## Android项目

看看其他项目的里程碑命名，会发现不同项目关于里程碑的命名各不相同。但是对于同一个项目要在里程碑命名上遵照同一标准，并能够和软件版本号正确地对应。

Android项目是一个非常有特色的使用Git版本库的项目，在后面会用两章介绍Android项目为Git带来的两个新工具。看看Android项目的里程碑编号对自己版本库的管理有无启发。

- 看看Android项目中的里程碑命名，会发现其里程碑的命名格式为`android-<大版本号>_r<小版本号>`。

```
$ git ls-remote --tags \
  git://android.git.kernel.org/platform/manifest.git \
  android-2.2*
6a03ae8f564130cbb4a11acf49bd705df7c8df6      refs/tags/android-2.2.1_r
599e242dea48f84e2f26054b0d1721e489043440      refs/tags/android-2.2.1_r
656ba6fdbd243153af6ec31017de38641060bf1e      refs/tags/android-2.2_r1
27cd0e346d1f3420c5747e01d2cb35e9ffd025ea      refs/tags/android-2.2_r1^
f6b7c499be268f1613d8cd70f2a05c12e01bcb93      refs/tags/android-2.2_r1.
bd3e9923773006a0a5f782e1f21413034096c4b1      refs/tags/android-2.2_r1.
03618e01ec9bdd06fd8fe9afdbdcba4b84092c5      refs/tags/android-2.2_r1.
ba7111e1d6fd26ab150bafa029fd5eab8196dad1      refs/tags/android-2.2_r1.
e03485e978ce1662a1285837f37ed39eadaedb1d      refs/tags/android-2.2_r1.
7386d2d07956be6e4f49a7e83eafb12215e835d7      refs/tags/android-2.2_r1.
```

- 里程碑的创建过程中使用了专用帐号和GnuPG签名。

```
$ git show android-2.2_r1
tag android-2.2_r1
Tagger: The Android Open Source Project <initial-contribution@android.com>
Date:   Tue Jun 29 11:28:52 2010 -0700

Android 2.2 release 1
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBMKjtm6K0/gZqxDngRA1BUAJ9QwgFbUL592FgRZLTLLbzhKsSQ8ACffQu5
Mjxg5X9oc+7N1DfdU+pmOcI=
=0NG0
-----END PGP SIGNATURE-----

commit 27cd0e346d1f3420c5747e01d2cb35e9ffd025ea
Author: The Android Open Source Project <initial-contribution@android.com>
Date:   Tue Jun 29 11:27:23 2010 -0700

    Manifest for android-2.2_r1

diff --git a/default.xml b/default.xml
index 4f21453..aaa26e3 100644
--- a/default.xml
+++ b/default.xml
@@ -3,7 +3,7 @@
     <remote name="korg"
             fetch="git://android.git.kernel.org/"
             review="review.source.android.com" />
-    <default revision="froyo"
+    <default revision="refs/tags/android-2.2_r1"
                 remote="korg" />
...

```

来源: <https://github.com/gotgit/gotgit/blob/master/03-git-harmony/030-git-tag.rst>

704 lines (1226 sloc) | 74.5 KB

## Git分支

分支是我们的老朋友了，第2篇中的“第6章 Git对象库”、“第7章 Git重置”和“第8章 Git检出”等章节中，就已经从实现原理上理解了分支。您想必已经知道了分支master的存在方式无非就是在目录`:file:`.git/refs/heads``下的文件（或称引用）而已。也看到了分支master的指向如何随着提交而变化，如何通过`:command:`git reset``命令而重置，以及如何使用`:command:`git checkout``命令而检出。

之前的章节都只用到了一个分支：master分支，而在本章会接触到多个分支。会从应用的角度上介绍分支的几种不同类型：发布分支、特性分支和卖主分支。在本章可以学习到如何对多分支进行操作，如何创建分支，如何切换到其他分支，以及分支之间的合并、变基等。

## 代码管理之殇

分支是代码管理的利器。如果没有有效的分支管理，代码管理就适应不了复杂的开发过程和项目的需要。在实际的项目实践中，单一分支的单线开发模式还远远不够，因为：

- 成功的软件项目大多要经过多个开发周期，发布多个软件版本。每个已经发布的版本都可能发现bug，这就需要对历史版本进行更改。
- 有前瞻性的项目管理，新版本的开发往往是和当前版本同步进行的。如果两个版本的开发都混杂在master分支中，肯定会是一场灾难。
- 如果产品要针对不同的客户定制，肯定是希望客户越多越好。如果所有的客户定制都混杂在一个分支中，必定会带来混乱。如果使用多个分支管理不同的定制，但如果管理不善，分支之间定制功能的迁移就会成为头痛的问题。
- 即便是所有成员都在为同一个项目的同一个版本进行工作，每个人领受任务却不尽相同，有的任务开发周期会很长，有的任务需要对软件架构进行较大的修改，如果所有人都工作在同一分支中，就会因为过多过频的冲突导致效率低下。
- 敏捷开发（不管是极限编程XP还是Scrum或其他）是最有效的项目管理模式，其最有效的一个实践就是快速迭代、每晚编译。如果不能将项目的各个功能模块的开发通过分支进行隔离，在软件集成上就会遭遇困难。

## 发布分支

### 为什么bug没完没了？

在2006年我接触到一个项目团队，使用Subversion做版本控制。最为困扰项目经理的是刚刚修正产品的一个bug，马上又会接二连三地发现新的bug。在访谈开发人员，询问开发人员是如何修正bug的时候，开发人员的回答让我大吃一惊：“当发现产品出现bug的时候，我要中断当前的工作，把我正在开发的新功能的代码注释掉，然后再去修改bug，修改好就生成一个war包（Java开发网站项目）给运维部门，扔到网站上去。”

于是我就画了下面的一个图（图18-1），大致描述了这个团队进行bug修正的过程，从中可以很容易地看出问题的端倪。这个图对于Git甚至其他版本控制系统同样适用。

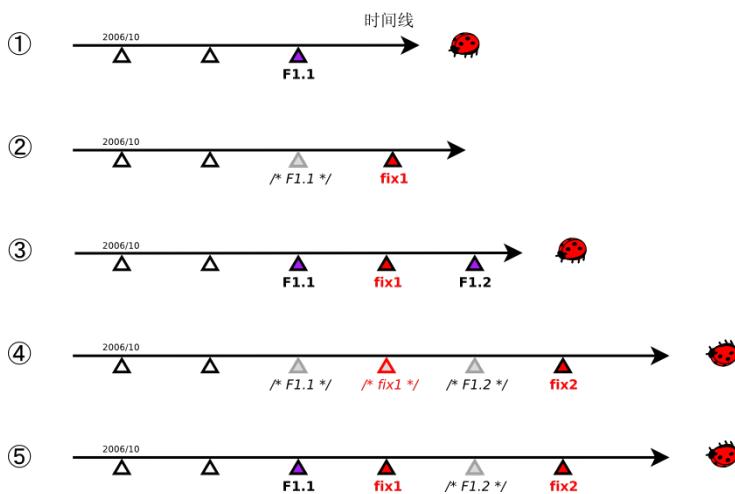


图 18-1：没有使用分支导致越改越多的bug

说明：

- 图18-1中的图示①，开发者针对功能1做了一个提交，编号“F1. 1”。这时客户报告产品出现了bug。
- 于是开发者匆忙地干了起来，图示②显示了该开发者修正bug的过程：将已经提交的针对功能1的代码“F1. 1”注释掉，然后提交一个修正bug的提交（编号：fix1）。
- 开发者编译出新的产品交给客户，接着开始功能1的开发。图示③显示了开发者针对功能1做出了一个新的提交“F1. 2”。
- 客户再次发现一个bug。开发者再次开始bug修正工作。
- 图示④和图示⑤显示了此工作模式下非常容易在修复一个bug的时候引入新的bug。
- 图示④的问题在于开发者注释功能1的代码时，不小心将“fix1”的代码也注释掉了，导致曾经修复的bug在新版本中重现。
- 图示⑤的问题在于开发者没有将功能1的代码剔出干净，导致在产品的 new 版本中引入了不完整和不需要的功能代码。用户可能看到一个新的但是不能使用的菜单项，甚至更糟。

使用版本控制系统的分支功能，可以避免对已发布的软件版本进行bug修正时引入新功能的代码，或者因误删其他bug修正代码导致已修复问题重现。在这种情况下创建的分支有一个专有的名称：bugfix分支或发布分支（Release Branch）。之所以称为发布分支，是因为在软件新版本发布后经常使用此技术进行软件维护，发布升级版本。

图18-2演示了如何使用发布分支应对bug修正的问题。

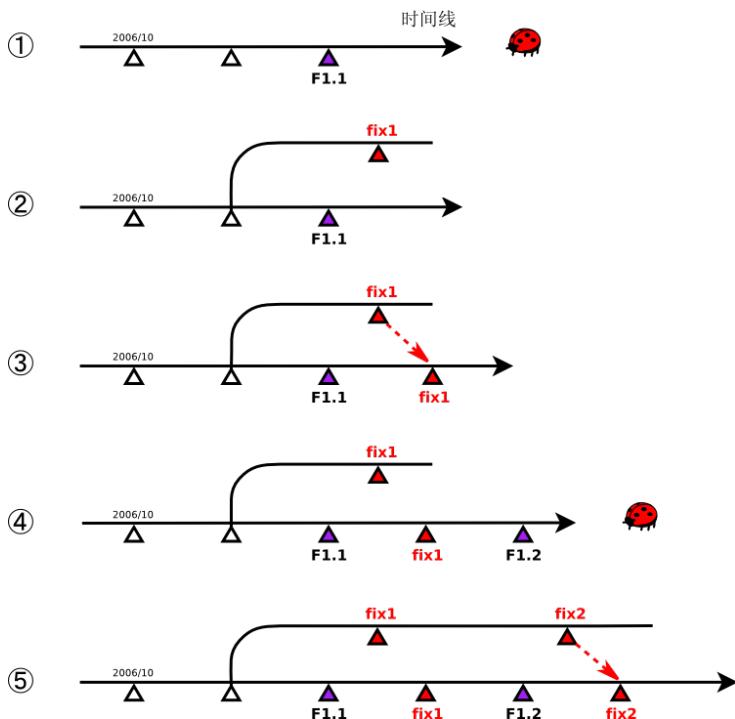


图 18-2：使用发布分支的bug修正过程

说明：

- 图18-2中的图示②，可以看到开发者创建了一个发布分支（bugfix分支），在分支中提交修正代码“fix1”。注意此分支是自上次软件发布时最后一次提交进行创建的，因此分支中没有包含开发者为新功能所做的提交“F1.1”，是一个“干净”的分支。
- 图示③可以看出从发布分支向主线做了一次合并，这是因为在主线上也同样存在该bug，需要在主线上也做出相应的更改。
- 图示④，开发者继续开发，针对功能1执行了一个新的提交，编号“F1.2”。这时，客户报告有新的bug。
- 继续在发布分支上进行bug修正，参考图示⑤。当修正完成（提交“fix2”）时，基于发布分支创建一个新的软件版本发给客户。不要忘了向主线合并，因为同样的bug可能在主线上也存在。

关于如何基于一个历史提交创建分支，以及如何在分支之间进行合并，在本章后面的内容中会详细介绍。

## 特性分支

为什么项目一再的拖延？

有这么一个软件项目，项目已经延期了可是还是看不到一点要完成的样子。最终老板变得有些不耐烦了，说道：“那么就砍掉一些功能吧”。项目经理听闻，一阵眩晕，因为项目经理知道自己负责的这个项目采用的是单一主线开发，要将一个功能从中撤销，工作量非常大，而且可能会牵涉到其他相关模块的变更。

图18-3就是这个项目的版本库示意图，显然这个项目的代码管理没有使用分支。

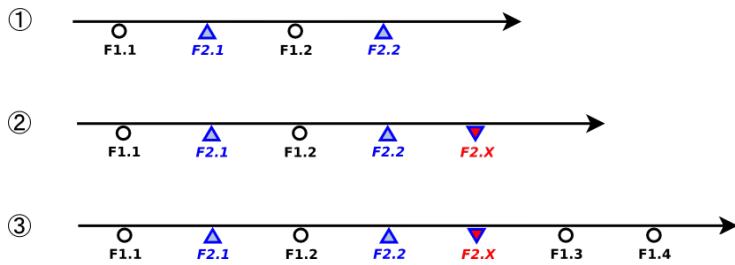


图 18-3: 没有使用分支导致项目拖延

说明:

- 图18-3中的图示①, 用圆圈代表功能1的历次提交, 用三角代替功能2的历次提交。因为所有开发者都在主线上工作, 所以提交混杂在一起。
- 当老板决定功能2不在这一版本的产品中发布, 延期到下一个版本时, 功能2的开发者做了一个(或者若干个)反向提交, 即图示②中的倒三角(代号为“F2.X”)标识的反向提交, 将功能2的所有历史提交全部撤销。
- 图示③表示除了功能2外的其他开发继续进行。

那么负责开发功能2的开发者干什么呢? 或者放一个长假, 或者在本地开发, 与版本库隔离, 即不向版本库提交, 直到延期的项目终于发布之后再将代码提交。这两种方法都是不可取的, 尤其是后一种隔离开发最危险, 如果因为病毒感染、文件误删、磁盘损坏, 就会导致全部工作损失殆尽。我的项目组就曾经遇到过这样的情况。

采用分支将某个功能或模块的开发与开发主线独立出来, 是解决类似问题的办法, 这种用途的分支被称为特性分支 (Feature Branch) 或主题分支 (Topic Branch)。图18-4就展示了如何使用特性分支帮助纠正要延期的项目, 协同多用户的开发。

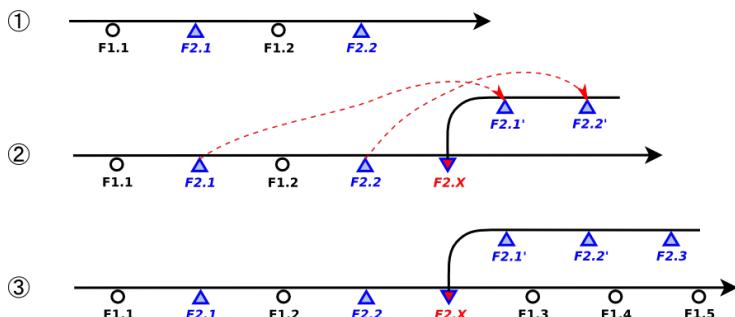


图 18-4: 使用特性分支协同多功能开发

说明:

- 图18-4中的图示①和前面的一样, 都是多个开发者的提交混杂在开发主线中。
- 图示②是当得知功能2不在此次产品发布中后, 功能2的开发者所做的操作。
  - 首先, 功能2的开发者提交一个(或若干个)反向提交, 将功能2的相关代码全部撤销。图中倒三角(代号为“F2.X”)的提交就是一个反向提交。
  - 接着, 功能2的开发者从反向提交开始创建一个特性分支。
  - 最后, 功能2的开发者将功能2的历史提交拣选到特性分支上。对于Git可以使用拣选命令:`git cherry-pick`。
- 图示③中可以看出包括功能2在内的所有功能和模块都继续提交, 但是提交的分支各不相同。功能2的开发者将代码提交到特性分支上, 其他开发者还提交到主线上。

那么在什么情况下使用特性分支呢? 试验性、探索性的功能开发应该为其建立特性分支。功能复杂、开发周期长(有可能在本次发布中取消)的模块应

该为其建立特性分支。会更改软件体系架构，破坏软件集成，或者容易导致冲突、影响他人开发进度的模块，应该为其建立特性分支。

在使用CVS或Subversion等版本控制系统建立分支时，或者因为太慢（CVS）或者因为授权原因需要找管理员进行操作，非常的不方便。Git的分支管理就方便多了，一是开发者可以在本地版本库中随心所欲地创建分支，二是管理员可以对共享版本库进行设置允许开发者创建特定名称的分支，这样开发者的本地分支可以推送到服务器实现数据的备份。关于Git服务器的分支授权参照本书第5篇的Gitolite服务器架设的相关章节。

## 卖主分支

有的项目要引用到第三方的代码模块并且需要对其进行定制，有的项目甚至整个就是基于某个开源项目进行的定制。如何有效地管理本地定制和第三方（上游）代码的变更就成为了一个难题。卖主分支（Vendor Branch）可以部分解决这个难题。

所谓卖主分支，就是在版本库中创建一个专门和上游代码进行同步的分支，一旦有上游代码发布就检入到卖主分支中。图18-5就是一个典型的卖主分支工作流程。

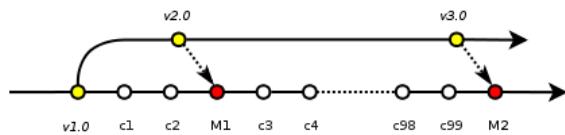


图 18-5：卖主分支工作流程

说明：

- 在主线检入上游软件版本1.0的代码。在图中标记为v1.0的提交即是。
- 然后在线上进行定制开发，c1、c2分别代表历次定制提交。
- 当上游有了新版本发布，例如2.0版本，就将上游新版本的源代码提交到卖主分支中。图中标记为v2.0的提交即是。
- 然后在线上合并卖主分支上的新提交，合并后的提交显示为M1。

如果定制较少，使用卖主分支可以工作得很好，但是如果定制的内容非常多，在合并的时候就会遇到非常多的冲突。定制的代码越多，混杂的越厉害，冲突解决就越困难。

本章的内容尚不能针对复杂的定制开发给出满意的版本控制解决方案，本书第4篇的“第22章 Topgit协同模型”会介绍一个针对复杂定制开发的更好的解决方案。

## 分支命令概述

在Git中分支管理使用命令：`git branch`。该命令的主要用法如下：

```
用法1: git branch
用法2: git branch <branchname>
用法3: git branch <branchname> <start-point>
用法4: git branch -d <branchname>
用法5: git branch -D <branchname>
用法6: git branch -m <oldbranch> <newbranch>
用法7: git branch -M <oldbranch> <newbranch>
```

说明：

- 用法1用于显示本地分支列表。当前分支在输出中会显示为特别的颜色，并用星号“\*”标识出来。

- 用法2和用法3用于创建分支。

用法2基于当前头指针（HEAD）指向的提交创建分支，新分支的分支名为`<branchname>`。

用法3基于提交`<start-point>`创建新分支，新分支的分支名为`<branchname>`。

- 用法4和用法5用于删除分支。

用法4在删除分支`<branchname>`时会检查所要删除的分支是否已经合并到其他分支中，否则拒绝删除。

用法5会强制删除分支`<branchname>`，即使该分支没有合并到任何一个分支中。

- 用法6和用法7用于重命名分支。

如果版本库中已经存在名为`<newbranch>`的分支，用法6拒绝执行重命名，而用法7会强制执行。

下面就通过hello-world项目演示Git的分支管理。

## Hello World开发计划

上一章从Github上检出的hello-world包含了一个C语言开发的应用，现在假设项目hello-world做产品发布，版本号定为1.0，则进行下面的里程碑操作。

- 为hello-world创建里程碑v1.0。

```
$ cd /path/to/user1/workspace/hello-world/  
$ git tag -m "Release 1.0" v1.0
```

- 将新建的里程碑推送到远程共享版本库。

```
$ git push origin refs/tags/v1.0  
Counting objects: 1, done.  
Writing objects: 100% (1/1), 158 bytes, done.  
Total 1 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (1/1), done.  
To file:///path/to/repos/hello-world.git  
* [new tag]           v1.0 -> v1.0
```

到现在为止还没有运行hello-world程序呢，现在就在开发者user1的工作区中运行一下。

- 进入`:file:`src``目录，编译程序。

```
$ cd src  
$ make  
version.h.in => version.h  
cc      -c -o main.o main.c  
cc -o hello main.o
```

- 使用参数`--help`运行hello程序，可以查看帮助信息。

说明：hello程序的帮助输出中有一个拼写错误，本应该是`--help`的地方写成了`-help`。这是有意为之。

```
$ ./hello --help  
Hello world example v1.0
```

```
Copyright Jiang Xin <jiangxin AT ossxp DOT com>, 2009.

Usage:
    hello
        say hello to the world.

    hello <username>
        say hi to the user.

    hello -h, --help
        this help screen.
```

- 不带参数运行，向全世界问候。

说明：最后一行显示版本为“v1.0”，这显然是来自于新建立的里程碑“v1.0”。

```
$ ./hello
Hello world.
(version: v1.0)
```

- 执行命令的时候，后面添加用户名作为参数，则向该用户问候。

说明：下面在运行hello的时候，显然出现了一个bug，即用户名中间如果出现了空格，输出的欢迎信息只包含了部分的用户名。这个bug也是有意为之。

```
$ ./hello Jiang Xin
Hi, Jiang.
(version: v1.0)
```

### 新版本开发计划

既然1.0版本已经发布了，现在是时候制订下一个版本2.0的开发计划了。计划如下：

- 多语种支持。

为hello-world添加多语种支持，使得软件运行的时候能够使用中文或其他本地化语言进行问候。

- 用getopt进行命令行解析。

对命令行参数解析框架进行改造，以便实现更灵活、更易扩展的命令行处理。在1.0版本中，程序内部解析命令行参数使用了简单的字符串比较，非常不灵活。从源文件：[file: `src/main.c`](#) 中可以看到当前实现的简陋和局限。

```
$ git grep -n argv
main.c:20:main(int argc, char **argv)
main.c:24:    } else if ( strcmp(argv[1],"-h") == 0 || 
main.c:25:        strcmp(argv[1],"--help") == 0 ) {
main.c:28:            printf ("Hi, %s.\n", argv[1]);
```

最终决定由开发者user2负责多语种支持的功能，由开发者user1负责用getopt进行命令行解析的功能。

## 基于特性分支的开发

有了前面“代码管理之殇”的铺垫，在领受任务之后，开发者user1和user2应该为自己负责的功能创建特性分支。

## 创建分支user1/getopt

开发者user1负责用getopt进行命令行解析的功能，因为这个功能用到getopt函数，于是将这个分支命名为user1/getopt。开发者 user1 使用:command:`git branch`命令创建该特性分支。

- 确保是在开发者user1的工作区中。

```
$ cd /path/to/user1/workspace/hello-world/
```

- 开发者user1基于当前HEAD创建分支user1/getopt。

```
$ git branch user1/getopt
```

- 使用:command:`git branch` 创建分支，并不会自动切换。查看当前分支可以看到仍然工作在master分支（用星号“\*”标识）中。

```
$ git branch
* master
  user1/getopt
```

- 执行:command:`git checkout` 命令切换到新分支上。

```
$ git checkout user1/getopt
Switched to branch 'user1/getopt'
```

- 再次查看分支列表，当前工作分支的标记符（星号）已经落在user1/getopt分支上。

```
$ git branch
  master
* user1/getopt
```

### 分支的奥秘

分支实际上是创建在目录:file:`.git/refs/heads`下的引用，版本库初始时创建的master分支就是在该目录下。在第2篇“Git重置”的章节中，已经介绍过master分支的实现，实际上这也是所有分支的实现方式。

- 查看一下目录:file:`.git/refs/heads` 目录下的引用。

可以在该目录下看到:file:`master` 文件，和一个:file:`user1` 目录。而在:file:`user1` 目录下是文件:file:`getopt`。

```
$ ls -F .git/refs/heads/
master  user1/
$ ls -F .git/refs/heads/user1/
getopt
```

- 引用文件:file:`.git/refs/heads/user1/getopt` 记录的是一个提交ID。

```
$ cat .git/refs/heads/user1/getopt
ebcf6d6b06545331df156687ca2940800a3c599d
```

- 因为分支user1/getopt是基于头指针HEAD创建的，因此当前该分支和master分支指向是一致的。

```
$ cat .git/refs/heads/master  
ebcf6d6b06545331df156687ca2940800a3c599d
```

- 当前的工作分支为user1/getopt，记录在头指针文件:`.git/HEAD`中。

切换分支命令:`git checkout`对文件:`.git/HEAD`的内容进行更新。可以参照第2篇“第8章 Git检出”的相关章节。

```
$ cat .git/HEAD  
ref: refs/heads/user1/getopt
```

## 创建分支user2/i18n

开发者user2要完成多语种支持的工作任务，于是决定将分支定名为user2/i18n。每一次创建分支通常都需要完成以下两个工作：

- 创建分支：执行:`git branch <branchname>`命令创建新分支。
- 切换分支：执行:`git checkout <branchname>`命令切换到新分支。

有没有简单的操作，在创建分支后立即切换到新分支上呢？有的，Git提供了这样一个命令，能够将上述两条命令所执行的操作一次性完成。用法如下：

```
用法: git checkout -b <new_branch> [<start_point>]
```

即检出命令:`git checkout`通过参数`-b <new_branch>`实现了创建分支和切换分支两个动作的合二为一。下面开发者user2就使用:`git checkout`命令来创建分支。

- 进入到开发者user2的工作目录，并和上游同步一次。

```
$ cd /path/to/user2/workspace/hello-world/  
$ git pull  
remote: Counting objects: 1, done.  
remote: Total 1 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (1/1), done.  
From file:///path/to/repos/hello-world  
* [new tag]           v1.0          -> v1.0  
Already up-to-date.
```

- 执行:`git checkout -b`命令，创建并切换到新分支user2/i18n上。

```
$ git checkout -b user2/i18n  
Switched to a new branch 'user2/i18n'
```

- 查看本地分支列表，会看到已经切换到user2/i18n分支上了。

```
$ git branch  
  master  
* user2/i18n
```

## 开发者 user1 完成功能开发

开发者user1开始在user1/getopt分支中工作，重构hello-world中的命令行参数解析的代码。重构时采用getopt\_long函数。

您可以试着更改，不过在hello-world中已经保存了一份改好的代码，可以直接检出。

- 确保是在user1的工作区中。

```
$ cd /path/to/user1/workspace/hello-world/
```

- 执行下面的命令，用里程碑jx/v2.0标记的内容（已实现用getopt进行命令行解析的功能）替换暂存区和工作区。

下面的`:command:`git checkout``命令的最后是一个点“.”，因此检出只更改了暂存区和工作区，而没有修改头指针。

```
$ cd /path/to/user1/workspace/hello-world/
$ git checkout jx/v2.0 -- .
```

- 查看状态，会看到分支仍保持为user1/getopt，但文件`:file:`src/main.c``被修改了。

```
$ git status
# On branch user1/getopt
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   src/main.c
#
```

- 比较暂存区和HEAD的文件差异，可以看到为实现用getopt进行命令行解析功能而对代码的改动。

```
$ git diff --cached
diff --git a/src/main.c b/src/main.c
index 6ee936f..fa5244a 100644
--- a/src/main.c
+++ b/src/main.c
@@ -1,4 +1,6 @@
 #include <stdio.h>
+#include <getopt.h>
+
 #include "version.h"

 int usage(int code)
@@ -19,15 +21,44 @@ int usage(int code)
 int
 main(int argc, char **argv)
 {
-    if (argc == 1) {
+    int c;
+    char *uname = NULL;
+
+    while (1) {
+        int option_index = 0;
+        static struct option long_options[] = {
+            {"help", 0, 0, 'h'},
+            {0, 0, 0, 0}
+    };
+
...}
```

- 开发者user1提交代码，完成开发任务。

```
$ git commit -m "Refactor: use getopt_long for arguments parsing"
[User1/getopt 0881ca3] Refactor: use getopt_long for arguments parsing
 1 files changed, 36 insertions(+), 5 deletions(-)
```

- 提交完成之后，可以看到这时user1/getopt分支和master分支的指向不同了。

```
$ git rev-parse user1/getopt master  
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf  
ebcf6d6b06545331df156687ca2940800a3c599d
```

- 编译运行hello-world。

注意输出中的版本号显示。

```
$ cd src  
$ make clean  
rm -f hello main.o version.h  
$ make  
version.h.in => version.h  
cc -c -o main.o main.c  
cc -o hello main.o  
$ ./hello  
Hello world.  
(version: v1.0-1-g0881ca3)
```

## 将user1/getopt分支合并到主线

既然开发者user1负责的功能开发完成了，那就合并到开发主线master上吧，这样测试团队（如果有的话）就可以基于开发主线master进行软件集成和测试了。

- 为将分支合并到主线，首先user1将工作区切换到主线，即master分支。

```
$ git checkout master  
Switched to branch 'master'
```

- 然后执行`:command:`git merge``命令以合并user1/getopt分支。

```
$ git merge user1/getopt  
Updating ebcf6d6..0881ca3  
Fast-forward  
src/main.c | 41 ++++++-----  
1 files changed, 36 insertions(+), 5 deletions(-)
```

- 本次合并非常的顺利，实际上合并后master分支和user1/getopt指向同一个提交。

这是因为合并前的master分支的提交就是user1/getopt分支的父提交，所以此次合并相当于分支master重置到user1/getopt分支。

```
$ git rev-parse user1/getopt master  
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf  
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
```

- 当前本地master分支比远程共享版本库的master分支领先一个提交。

可以从状态信息中看到本地分支和远程分支的跟踪关系。

```
$ git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 1 commit.  
#
```

```
nothing to commit (working directory clean)
```

- 执行推送操作，完成本地分支向远程分支的同步。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 689 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/hello-world.git
  ebcf6d6..0881ca3  master -> master
```

- 删除user1/getopt分支。

既然特性分支user1/getopt已经合并到主线上了，那么该分支已经完成了历史使命，可以放心地将其删除。

```
$ git branch -d user1/getopt
Deleted branch user1/getopt (was 0881ca3).
```

开发者user2对多语种支持功能有些犯愁，需要多花些时间，那么就先不等他了。

## 基于发布分支的开发

用户在使用1.0版的hello-world过程中发现了两个错误，报告给项目组。

- 第一个问题是：帮助信息中出现文字错误。本应该写为“--help”却写成了“-help”。
- 第二个问题是：当执行hello-world的程序，提供带空格的用户名时，问候语中显示的是不完整的用户名。

例如执行：`command: `./hello Jiang Xin``，本应该输出“Hi, Jiang Xin.”，却只输出了“Hi, Jiang.”。

为了能够及时修正1.0版本中存在的这两个bug，将这两个bug的修正工作分别交给两个开发者user1和user2完成。

- 开发者user1负责修改文字错误的bug。
- 开发者user2负责修改显示用户名不完整的bug。

现在版本库中master分支相比1.0发布时添加了新功能代码，即开发者user1推送的用getopt进行命令行解析相关代码。如果基于master分支对用户报告的两个bug进行修改，就会引入尚未经过测试、可能不稳定的新功能的代码。在此之前“代码管理之殇”中介绍的发布分支，恰恰适用于此场景。

## 创建发布分支

要想解决在1.0版本中发现的bug，就需要基于1.0发行版的代码创建发布分支。

- 软件hello-world的1.0发布版在版本库中有一个里程碑相对应。

```
$ cd /path/to/user1/workspace/hello-world/
$ git tag -n1 -l v*
v1.0          Release 1.0
```

- 基于里程碑v1.0创建发布分支hello-1.x。

注：使用了`:command:`git checkout``命令创建分支，最后一个参数v1.0是新分支hello-1.x创建的基准点。如果没有里程碑，使用提交ID也是一样的。

```
$ git checkout -b hello-1.x v1.0
Switched to a new branch 'hello-1.x'
```

- 用`:command:`git rev-parse``命令可以看到hello-1.x分支对应的提交ID和里程碑v1.0指向的提交一致，但是和master不一样。

提示：因为里程碑v1.0是一个包含提交说明的里程碑，因此为了显示其对应的提交ID，使用了特别的记法“`v1.0^{}{}`”。

```
$ git rev-parse hello-1.x v1.0^{} master
ebcf6d6b06545331df156687ca2940800a3c599d
ebcf6d6b06545331df156687ca2940800a3c599d
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
```

- 开发者user1将分支hello-1.x推送到远程共享版本库，因为开发者user2修改bug时也要用到该分支。

```
$ git push origin hello-1.x
Total 0 (delta 0), reused 0 (delta 0)
To file:///path/to/repos/hello-world.git
 * [new branch]      hello-1.x -> hello-1.x
```

- 开发者user2从远程共享版本库获取新的分支。

开发者user2执行`:command:`git fetch``命令，将远程共享版本库的新分支hello-1.x复制到本地引用origin/hello-1.x上。

```
$ cd /path/to/user2/workspace/hello-world/
$ git fetch
From file:///path/to/repos/hello-world
 * [new branch]      hello-1.x -> origin/hello-1.x
```

- 开发者user2切换到hello-1.x分支。

本地引用origin/hello-1.x称为远程分支，第19章将专题介绍。该远程分支不能直接检出，而是需要基于该远程分支创建本地分支。第19章会介绍一个更为简单的基于远程分支建立本地分支的方法，本例先用标准的方法建立分支。

```
$ git checkout -b hello-1.x origin/hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from
Switched to a new branch 'hello-1.x'
```

## 开发者user1工作在发布分支

开发者user1修改帮助信息中的文字错误。

- 编辑文件`:file:`src/main.c``，将“`-help`”字符串修改为“`--help`”。

```
$ cd /path/to/user1/workspace/hello-world/
$ vi src/main.c
...
```

- 开发者user1的改动可以从下面的差异比较中看到。

```
$ git diff
diff --git a/src/main.c b/src/main.c
index 6ee936f..e76f05e 100644
--- a/src/main.c
+++ b/src/main.c
@@ -11,7 +11,7 @@ int usage(int code)
        "      say hello to the world.\n\n"
        "      hello <username>\n"
        "      say hi to the user.\n\n"
-       "      hello -h, -help\n"
+       "      hello -h, --help\n"
        "      this help screen.\n\n", _VERSION);
    return code;
}
```

- 执行提交。

```
$ git add -u
$ git commit -m "Fix typo: -help to --help."
[hello-1.x b56bb51] Fix typo: -help to --help.
 1 files changed, 1 insertions(+), 1 deletions(-)
```

- 推送到远程共享版本库。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 349 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/hello-world.git
  ebcf6d6..b56bb51  hello-1.x -> hello-1.x
```

## 开发者user2工作在发布分支

开发者user2针对问候时用户名显示不全的bug进行更改。

- 进入开发者user2的工作区，并确保工作在hello-1.x分支中。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout hello-1.x
```

- 编辑文件:file:`src/main.c`，修改代码中的bug。

```
$ vi src/main.c
```

- 实际上在hello-world版本库中包含了我的一份修改，可以看看和您的更改是否一致。

下面的命令将我对此bug的修改保存为一个补丁文件。

```
$ git format-patch jx/v1.1..jx/v1.2
0001-Bugfix-allow-spaces-in-username.patch
```

- 应用我对此bug的改动补丁。

如果您已经自己完成了修改，可以先执行:command:`git stash`保存自己的修改进度，然后执行下面的命令应用补丁文件。当应用完补丁后，再执行:command:`git stash pop`将您的改动合并到工作区。如果我们的改动一致（英雄所见略同），将不会有冲突。

```
$ patch -p1 < 0001-Bugfix-allow-spaces-in-username.patch
patching file src/main.c
```

- 看看代码的改动吧。

```
$ git diff
diff --git a/src/main.c b/src/main.c
index 6ee936f..f0f404b 100644
--- a/src/main.c
+++ b/src/main.c
@@ -19,13 +19,20 @@ int usage(int code)
int
main(int argc, char **argv)
{
+    char **p = NULL;
+
    if (argc == 1) {
        printf ("Hello world.\n");
    } else if ( strcmp(argv[1],"-h") == 0 ||
                strcmp(argv[1],"--help") == 0 ) {
        return usage(0);
    } else {
-        printf ("Hi, %s.\n", argv[1]);
+        p = &argv[1];
+        printf ("Hi,");
+        do {
+            printf (" %s", *p);
+            } while (*(++p));
+        printf ("\n");
    }

    printf( "(version: %s)\n", _VERSION );
```

- 本地测试一下改进后的软件，看看bug是否已经被改正。如果运行结果能显示出完整的用户名，则bug成功修正。

```
$ cd src/
$ make
version.h.in => version.h
cc    -c -o main.o main.c
cc -o hello main.o
$ ./hello Jiang Xin
Hi, Jiang Xin.
(version: v1.0-dirty)
```

- 提交代码。

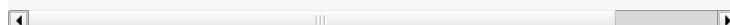
```
$ git add -u
$ git commit -m "Bugfix: allow spaces in username."
[hello-1.x e64f3a2] Bugfix: allow spaces in username.
 1 files changed, 8 insertions(+), 1 deletions(-)
```

## 开发者user2合并推送

开发者user2在本地版本库完成提交后，不要忘记向远程共享版本库进行推送。但在推送分支hello-1.x时开发者user2没有开发者user1那么幸运，因为此时远程共享版本库的hello-1.x分支已经被开发者user1推送过一次，因此开发者user2在推送过程中会遇到非快进式推送问题。

```
$ git push
To file:///path/to/repos/hello-world.git
! [rejected]      hello-1.x -> hello-1.x (non-fast-forward)
```

```
error: failed to push some refs to 'file:///path/to/repos/hello-wc'
To prevent you from losing history, non-fast-forward updates were
Merge the remote changes (e.g. 'git pull') before pushing again.
'Note about fast-forwards' section of 'git push --help' for detail
```



就像在“第15章 Git协议和工作协同”一章中介绍的那样，开发者user2需要执行一个拉回操作，将远程共享服务器的改动获取到本地并和本地提交进行合并。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/hello-world
  ebpcf6d6..b56bb51  hello-1.x  -> origin/hello-1.x
Auto-merging src/main.c
Merge made by recursive.
 src/main.c |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

通过显示分支图的方式查看日志，可以看到在执行`git pull`操作后发生了合并。

```
$ git log --graph --oneline
*   8cfffe5f Merge branch 'hello-1.x' of file:///path/to/repos/hell
|\ 
| * b56bb51 Fix typo: -help to --help.
| * e64f3a2 Bugfix: allow spaces in username.
|/
* ebpcf6d6 blank commit for GnuPG-signed tag test.
* 8a9f3d1 blank commit for annotated tag test.
* 60a2f4f blank commit.
* 3e6070e Show version.
* 75346b3 Hello world initialized.
```

现在开发者user2可以将合并后的本地版本库中的提交推送给远程共享版本库了。

```
$ git push
Counting objects: 14, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 814 bytes, done.
Total 8 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/hello-world.git
 b56bb51..8cfffe5f  hello-1.x -> hello-1.x
```

## 发布分支的提交合并到主线

当开发者user1和user2都相继在hello-1.x分支将相应的bug修改完后，就可以从hello-1.x分支中编译新的软件产品交给客户使用了。接下来别忘了在主线master分支也做出同样的更改，因为在hello-1.x分支修改的bug同样也存在于主线master分支中。

使用Git提供的拣选命令，就可以直接将发布分支上进行的bug修正合并到主线上。下面就以开发者user2的身份进行操作。

- 进入user2工作区并切换到master分支。

```
$ cd /path/to/user2/workspace/hello-world/
```

```
$ git checkout master
```

- 从远程共享版本库同步master分支。

同步后本地master分支包含了开发者user1提交的命令行参数解析重构的代码。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/hello-world
  ebcf6d6..0881ca3  master      -> origin/master
Updating ebcf6d6..0881ca3
Fast-forward
  src/main.c |  41 ++++++-----+
   1 files changed, 36 insertions(+), 5 deletions(-)
```

- 查看分支hello-1.x的日志，确认要拣选的提交ID。

从下面的日志可以看出分支hello-1.x的最新提交是一个合并提交，而要拣选的提交分别是其第一个父提交和第二个父提交，可以分别用hello-1.x^1和hello-1.x^2表示。

```
$ git log -3 --graph --oneline hello-1.x
*   8cfffe5f Merge branch 'hello-1.x' of file:///path/to/repos/
|\ \
| * b56bb51 Fix typo: -help to --help.
* | e64f3a2 Bugfix: allow spaces in username.
|/
```

- 执行拣选操作。先将开发者user2提交的修正代码拣选到当前分支（即主线）。

拣选操作遇到了冲突，见下面的命令输出。

```
$ git cherry-pick hello-1.x^1
Automatic cherry-pick failed. After resolving the conflicts,
mark the corrected paths with 'git add <paths>' or 'git rm <pa
and commit the result with:
```

```
git commit -c e64f3a216d346669b85807ffcfb23a21f9c5c187
```

- 拣选操作发生冲突，通过查看状态可以看到是在文件:`src/main.c`上发生了冲突。

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolut
#
#       both modified:    src/main.c
#
no changes added to commit (use "git add" and/or "git commit -
```

### 冲突发生的原因

为什么发生了冲突呢？这是因为拣选hello-1.x分支上的一个提交到master分

支时，因为两个甚至多个提交在重叠的位置更改代码所致。通过下面的命令可以看到到底是哪些提交引起的冲突。

```
$ git log master...hello-1.x^1
commit e64f3a216d346669b85807ffcfb23a21f9c5c187
Author: user2 <user2@moon.osxp.com>
Date:   Sun Jan 9 13:11:19 2011 +0800

    Bugfix: allow spaces in username.

commit 0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
Author: user1 <user1@sun.osxp.com>
Date:   Mon Jan 3 22:44:52 2011 +0800

    Refactor: use getopt_long for arguments parsing.
```

可以看出引发冲突的提交一个是当前工作分支master上的最新提交，即开发者user1的重构命令行参数解析的提交，而另外一个引发冲突的是要拣选的提交，即开发者user2针对用户名显示不全所做的错误修正提交。一定是因为这两个提交的更改发生了重叠导致了冲突的发生。下面就来解决冲突。

### 冲突解决

冲突解决可以使用图形界面工具，不过对于本例直接编辑冲突文件，手工进行冲突解决也很方便。打开文件:`file:`src/main.c``就可以看到发生冲突的区域都用特有的标记符标识出来，参见表18-1中左侧一列中的内容。

表 18-1：冲突解决前后对照

冲突文件 <code>src/main.c</code> 标识出的冲突内容	冲突解决后的内容对照
<pre> 21 int 22 main(int argc, char **argv) 23 { 24 &lt;&lt;&lt;&lt;&lt; HEAD 25     int c; 26     char *uname = NULL; 27 28     while (1) { 29         int option_index = 0; 30         static struct option long_options[] = { 31             {"help", 0, 0, 'h'}, 32             {0, 0, 0, 0} 33         }; 34 35         c = getopt_long(argc, argv, "h", 36                         long_options, &amp;option_index); 37         if (c == -1) 38             break; 39 40         switch (c) { 41         case 'h': 42             return usage(0); 43         default: 44             return usage(1); 45         } 46     } 47 48     if (optind &lt; argc) { 49         uname = argv[optind]; 50     } 51 52     if (uname == NULL) { 53 ====== 54     char **p = NULL; 55 56     if (argc == 1) { 57 &gt;&gt;&gt;&gt;&gt; e64f3a2... Bugfix: allow spaces in username. </pre>	<pre> 21 int 22 main(int argc, char **argv) 23 { 24     int c; 25     char **p = NULL; 26 27     while (1) { 28         int option_index = 0; 29         static struct option long_options = 30             {"help", 0, 0, 'h'}, 31             {0, 0, 0, 0} 32     }; 33 34     c = getopt_long(argc, argv, "h", 35                     long_options, &amp;option_index); 36     if (c == -1) 37         break; 38 39     switch (c) { 40     case 'h': 41         return usage(0); 42     default: 43         return usage(1); 44     } 45 } 46 47     if (optind &lt; argc) { 48         p = &amp;argv[optind]; 49     } 50 51     if (p == NULL    *p == NULL) { </pre>

```

58         printf ("Hello world.\n");
59     } else {
60 <<<<< HEAD
61         printf ("Hi, %s.\n", uname);
62 =====
63     p = &argv[1];
64     printf ("Hi,");
65     do {
66         printf (" %s", *p);
67     } while (*(++p));
68     printf ("\n");
69 >>>>> e64f3a2... Bugfix: allow spaces in username.
70 }
71
72     printf( "(version: %s)\n", _VERSION );
73     return 0;
74 }
```

```

52         printf ("Hello world.\n");
53     } else {
54         printf ("Hi,");
55     do {
56         printf (" %s", *p);
57     } while (*(++p));
58     printf ("\n");
59 }
60
61     printf( "(version: %s)\n", _VERSION );
62     return 0;
63 }
```

在文件:`src/main.c`冲突内容中，第25-52行及第61行是master分支中由开发者user1重构命令行解析时提交的内容，而第54-56行及第63-68行则是分支hello-1.x中由开发者user2提交的修正用户名显示不全的bug的相应代码。

表18-1右侧的一列则是冲突解决后的内容。为了和冲突前的内容相对照，重新进行了排版，并对差异内容进行加粗显示。您可以参照完成冲突解决。

将手动编辑完成的文件:`src/main.c`添加到暂存区才真正地完成了冲突解决。

```
$ git add src/main.c
```

因为是拣选操作，提交时最好重用所拣选提交的提交说明和作者信息，而且也省下了自己写提交说明的麻烦。使用下面的命令完成提交操作。

```
$ git commit -C hello-1.x^1
[master 10765a7] Bugfix: allow spaces in username.
 1 files changed, 8 insertions(+), 4 deletions(-)
```

接下来再将开发者 user1 在分支hello-1.x中的提交也拣选到当前分支。所拣选的提交非常简单，不过是修改了提交说明中的文字错误而已，拣选操作也不会引发异常，直接完成。

```
$ git cherry-pick hello-1.x^2
Finished one cherry-pick.
[master d81896e] Fix typo: -help to --help.
 Author: user1 <user1@sun.osxp.com>
 1 files changed, 1 insertions(+), 1 deletions(-)
```

现在通过日志可以看到master分支已经完成了对已知bug的修复。

```
$ git log -3 --graph --oneline
* d81896e Fix typo: -help to --help.
* 10765a7 Bugfix: allow spaces in username.
* 0881ca3 Refactor: use getopt_long for arguments parsing.
```

查看状态可以看到当前的工作分支相对于远程服务器有两个新提交。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
```

```
nothing to commit (working directory clean)
```

执行推送命令将本地master分支同步到远程共享版本库。

```
$ git push
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 802 bytes, done.
Total 8 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/hello-world.git
0881ca3..d81896e master -> master
```

## 分支变基

### 完成user2/i18n特性分支的开发

开发者user2针对多语种开发的工作任务还没有介绍呢，在最后就借着“实现”这个稍微复杂的功能来学习一下Git分支的变基操作。

- 进入user2的工作区，并切换到user2/i18n分支。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout user2/i18n
Switched to branch 'user2/i18n'
```

- 使用gettext为软件添加多语言支持。您可以尝试实现该功能。不过在hello-world中已经保存了一份实现该功能的代码（见里程碑jx/v1.0-i18n），可以直接拿过来用。
- 里程碑jx/v1.0-i18n最后的两个提交实现了多语言支持功能。

```
$ git log --oneline -2 --stat jx/v1.0-i18n
ade873c Translate for Chinese.
src/locale/zh_CN/LC_MESSAGES/helloworld.po | 30 ++++++
1 files changed, 23 insertions(+), 7 deletions(-)
0831248 Add I18N support.
src/Makefile | 21 ++++++
src/locale/helloworld.pot | 46 ++++++
src/locale/zh_CN/LC_MESSAGES/helloworld.po | 46 ++++++
src/main.c | 18 ++++++-
4 files changed, 125 insertions(+), 6 deletions(-)
```

- 可以通过拣选命令将这两个提交拣选到user2/i18n分支中，相当于在分支user2/i18n中实现了多语言支持的开发。

```
$ git cherry-pick jx/v1.0-i18n~1
...
$ git cherry-pick jx/v1.0-i18n
...
```

- 看看当前分拣选后的日志。

```
$ git log --oneline -2
7acb3e8 Translate for Chinese.
90d873b Add I18N support.
```

- 为了测试刚刚“开发”完成的多语言支持功能，先对源码执行编译。

```
$ cd src  
$ make  
version.h.in => version.h  
cc -c -o main.o main.c  
msgfmt -o locale/zh_CN/LC_MESSAGES/helloworld.mo locale/zh_CN/  
cc -o hello main.o
```

- 查看帮助信息，会发现帮助信息已经本地化。

注意：帮助信息中仍然有文字错误，`--help`误写为`-help`。

```
$ ./hello --help  
Hello world 示例 v1.0-2-g7acb3e8  
版权所有 蒋鑫 <jiangxin AT ossxp DOT com>, 2009  
  
用法：  
hello  
世界你好。  
  
hello <username>  
向用户问您好。  
  
hello -h, -help  
显示本帮助页。
```

- 不带用户名运行`hello`，也会输出中文。

```
$ ./hello  
世界你好。  
(version: v1.0-2-g7acb3e8)
```

- 带用户名运行`hello`，会向用户问候。

注意：程序仍然存在只显示部分用户名的问题。

```
$ ./hello Jiang Xin  
您好, Jiang.  
(version: v1.0-2-g7acb3e8)
```

- 推送分支`user2/i18n`到远程共享服务器。

推送该特性分支的目的并非是与他人在此分支上协同工作，主要只是为了进行数据备份。

```
$ git push origin user2/i18n  
Counting objects: 21, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (13/13), done.  
Writing objects: 100% (17/17), 2.91 KiB, done.  
Total 17 (delta 6), reused 1 (delta 0)  
Unpacking objects: 100% (17/17), done.  
To file:///path/to/repos/hello-world.git  
 * [new branch]      user2/i18n -> user2/i18n
```

## 分支`user2/i18n`变基

在测试刚刚完成的具有多语种支持功能的`hello-world`时，之前改正的两个bug又重现了。这并不奇怪，因为分支`user2/i18n`基于`master`分支创建的时候，这两个bug还没有发现呢，更不要说改正了。

在最早刚刚创建`user2/i18n`分支时，版本库的结构非常简单，如图18-6所

示。

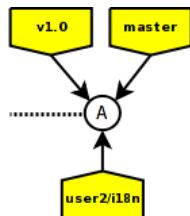


图 18-6：分支 user2/i18n 创建初始版本库分支状态

但是当前 master 分支中不但包含了对两个 bug 的修正，还包含了开发者 user1 调用 getopt 对命令行参数解析进行的代码重构。图 18-7 显示的是当前版本库 master 分支和 user2/i18n 分支的关系图。

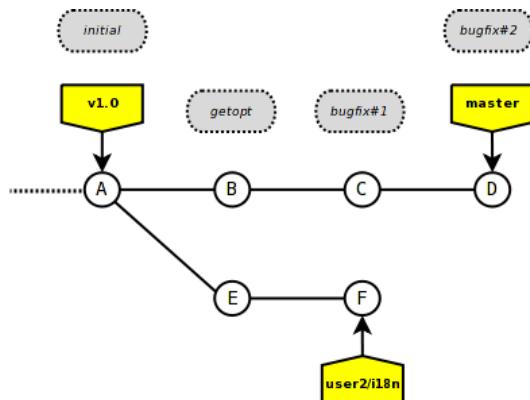


图 18-7：当前版本库分支示意图

开发者 user2 要将分支 user2/i18n 中的提交合并到主线 master 中，可以采用上一节介绍的分支合并操作。如果执行分支合并操作，版本库的状态将会如图 18-8 所示：

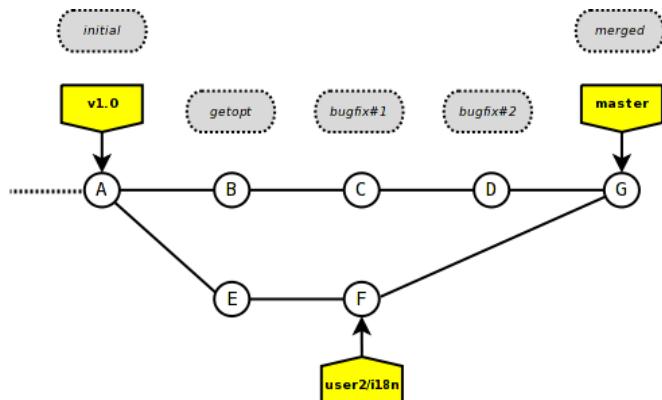


图 18-8：使用分支合并时版本库分支状态

这样操作有利有弊。有利的一面是开发者在 user2/i18n 分支中的提交不会发生改变，这一点对于提交已经被他人共享时很重要。再有因为 user2/i18n 分支是基于 v1.0 创建的，这样可以很容易将多语言支持功能添加到 1.0 版本的 hello-world 中。不过这些对于本项目来说都不重要。至于不利的一面，就是这样的合并操作会产生三个提交（包括一个合并提交），对于要对提交进行审核的项目团队来说增加了代码审核的负担。因此很多项目在特性分支合并到开发主线的时候，都不推荐使用合并操作，而是使用变基操作。如果执行变基操作，版本库相关分支的关系图如图 18-9 所示。

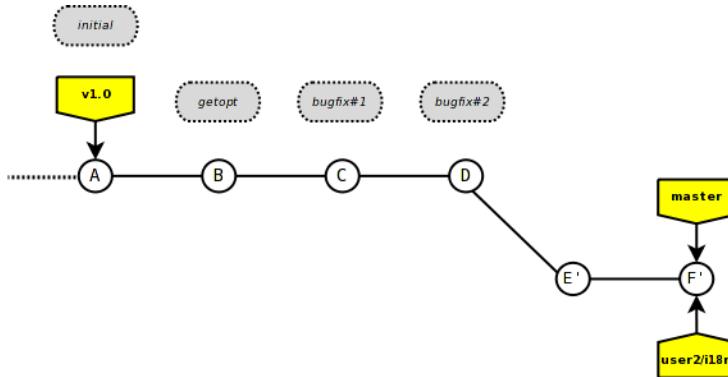


图 18-9：使用变基操作版本库分支状态

很显然，采用变基操作的分支关系图要比采用合并操作的简单多了，看起来更像是集中式版本控制系统特有的顺序提交。因为减少了一个提交，也会减轻代码审核的负担。

下面开发者user2就通过变基操作将特性分支user2/i18n合并到主线。

- 首先确保开发者user2的工作区位于分支user2/i18n上。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout user2/i18n
```

- 执行变基操作。

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add I18N support.
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Failed to merge in the changes.
Patch failed at 0001 Add I18N support.

When you have resolved this problem run "git rebase --continue"
If you would prefer to skip this patch, instead run "git rebase -s
To restore the original branch and stop rebasing run "git reb...
```

变基遇到了冲突，看来这回的麻烦可不小。冲突是在合并user2/i18n分支中的提交“Add I18N support”时遇到的。首先回顾一下变基的原理，参见第2篇“第12章 改变历史”相关章节。对于本例，在进行变基操作时会先切换到user2/i18n分支，并强制重置到master分支所指向的提交。然后再将原user2/i18n分支的提交一一拣选到新的user2/i18n分支上。运行下面的命令可以查看可能导致冲突的提交列表。

```
$ git rev-list --pretty=oneline user2/i18n^...master
d81896e60673771ef1873b27a33f52df75f70515 Fix typo: -help to --help
10765a7ef46981a73d57846669f6e17b73ac7e3 Bugfix: allow spaces in
90d873bb93cd7577b7638f1f391bd2ece3141b7a Add I18N support.
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf Refactor: use getopt_long
```

刚刚发生的冲突是在拣选提交“Add I18N support”时出现的，所以在冲突文件中标识为他人版本的是user2添加多语种支持功能的提交，而冲突文件中标识为自己版本的是修正两个bug的提交及开发者user1提交的重构命令行参数解析的提交。下面的两个表格（表18-2和表18-3）是文件src/main.c发生冲突的两个主要区域，表格的左侧一列是冲突文件中的内容，右侧一列则是冲突解决后的内容。为了方便参照进行了适当排版。

表 18-2: 变基冲突区域一解决前后对照

变基冲突区域一内容 (文件 src/main.c)	冲突解决后的内容
<pre> 12 int usage(int code) 13 { 14     printf(_("Hello world example %s\n" 15             "Copyright Jiang Xin &lt;jiangxin AT ossxp ...\\n" 16             "\\n" 17             "Usage:\\n" 18             "    hello\\n" 19             "        say hello to the world.\\n\\n" 20             "    hello &lt;username&gt;\\n" 21             "        say hi to the user.\\n\\n" 22 &lt;&lt;&lt;&lt;&lt; HEAD 23             "    hello -h, --help\\n" 24             "        this help screen.\\n\\n", _VERSION); 25       merged common ancestors 26             "    hello -h, -help\\n" 27             "        this help screen.\\n\\n", _VERSION); 28 ====== 29             "    hello -h, -help\\n" 30             "        this help screen.\\n\\n"), _VERSION); 31 &gt;&gt;&gt;&gt;&gt; Add I18N support. 32     return code; 33 }</pre>	<pre> 12 int usage(int code) 13 { 14     printf(_("Hello world example % 15             "Copyright Jiang Xin &lt;ji 16             "\\n" 17             "Usage:\\n" 18             "    hello\\n" 19             "        say hello t 20             "    hello &lt;username&gt;\\n" 21             "        say hi to t 22             "    hello -h, --help\\n" 23             "        this help s 24     return code; 25 }</pre>

表 18-3: 变基冲突区域二解决前后对照

变基冲突区域二内容 (文件 src/main.c)	冲突解决后的内容对照
<pre> 38 &lt;&lt;&lt;&lt;&lt; HEAD 39     int c; 40     char **p = NULL; 41 42     while (1) { 43         int option_index = 0; 44         static struct option long_options[] = { 45             {"help", 0, 0, 'h'}, 46             {0, 0, 0, 0} 47         }; 48 49         c = getopt_long(argc, argv, "h", 50                         long_options, &amp;option_index); 51         if (c == -1) 52             break; 53 54         switch (c) { 55         case 'h': 56             return usage(0); 57         default: 58             return usage(1); 59         } 60     } 61 62     if (optind &lt; argc) { 63         p = &amp;argv[optind]; 64     } 65 66     if (p == NULL    *p == NULL) { 67         printf ("Hello world.\\n"); 68       merged common ancestors 69     if (argc == 1) {</pre>	<pre> 30     int c; 31     char **p = NULL; 32 33     setlocale( LC_ALL, "" ); 34     bindtextdomain("helloworld","locale") 35     textdomain("helloworld"); 36 37     while (1) { 38         int option_index = 0; 39         static struct option long_options = 40             {"help", 0, 0, 'h'}, 41             {0, 0, 0, 0} 42     }; 43 44     c = getopt_long(argc, argv, "h", 45                     long_options, &amp;option_index); 46     if (c == -1) 47         break; 48 49     switch (c) { 50     case 'h': 51         return usage(0); 52     default: 53         return usage(1); 54     } 55 } 56 57     if (optind &lt; argc) { 58         p = &amp;argv[optind]; 59     } 60 61     if (p == NULL    *p == NULL) { 62         printf (_("Hello world.\\n")); 63     } 64 }</pre>

```

70         printf ("Hello world.\n");
71     } else if ( strcmp(argv[1],"-h") == 0 || 
72                 strcmp(argv[1],"--help") == 0 ) {
73         return usage(0);
74 ======
75     setlocale( LC_ALL, "" );
76     bindtextdomain("helloworld","locale");
77     textdomain("helloworld");
78
79     if (argc == 1) {
80         printf ( _("Hello world.\n") );
81     } else if ( strcmp(argv[1],"-h") == 0 || 
82                 strcmp(argv[1],"--help") == 0 ) {
83         return usage(0);
84 >>>>> Add I18N support.
85     } else {
86 <<<<< HEAD
87         printf ("Hi,");
88         do {
89             printf (" %s", *p);
90             } while (*(++p));
91         printf ("\n");
92 ||||| merged common ancestors
93         printf ("Hi, %s.\n", argv[1]);
94 ======
95         printf ( _("Hi, %s.\n"), argv[1]);
96 >>>>> Add I18N support.
97     }

```

```

63     } else {
64         printf ( _("Hi,"));
65         do {
66             printf (" %s", *p);
67             } while (*(++p));
68         printf ("\n");
69     }

```

将完成冲突解决的文件src/main.c加入暂存区。

```
$ git add -u
```

查看工作区状态。

```

$ git status
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   src/Makefile
#       new file:   src/locale/helloworld.pot
#       new file:   src/locale/zh_CN/LC_MESSAGES/helloworld.po
#       modified:   src/main.c
#

```

现在不要执行提交，而是继续变基操作。变基操作会自动完成对冲突解决的提交，并对分支中的其他提交继续执行变基，直至全部完成。

```

$ git rebase --continue
Applying: Add I18N support.
Applying: Translate for Chinese.

```

图18-10显示了版本库执行完变基后的状态。

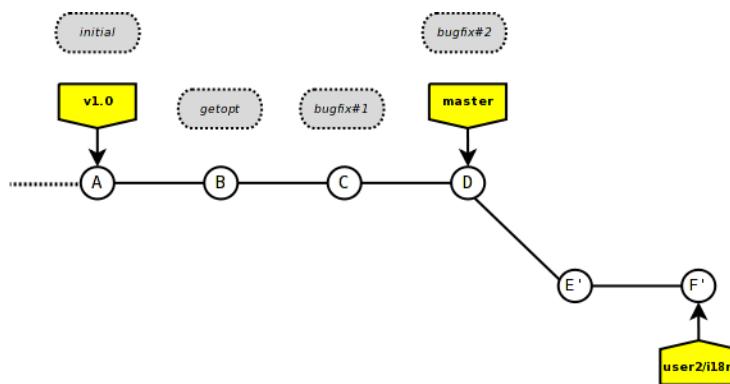


图 18-10: 变基操作完成后版本库分支状态

现在需要将user2/i18n分支的提交合并到主线master中。实际上不需要在master分支上再执行繁琐的合并操作，而是可以直接用推送操作——用本地的user2/i18n分支直接更新远程版本库的master分支。

```
$ git push origin user2/i18n:master
Counting objects: 21, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), 2.91 KiB, done.
Total 17 (delta 6), reused 1 (delta 0)
Unpacking objects: 100% (17/17), done.
To file:///path/to/repos/hello-world.git
```

仔细看看上面运行的`:command:`git push`命令`，终于看到了引用表达式中引号前后使用了不同名字的引用。含义是用本地的user2/i18n引用的内容（提交ID）更新远程共享版本库的master引用内容（提交ID）。

执行拉回操作，可以发现远程共享版本库的master分支的确被更新了。通过拉回操作本地的master分支也随之更新。

- 切换到master分支，会从提示信息中看到本地master分支落后远程共享版本库master分支两个提交。

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be
```

- 执行拉回操作，将本地master分支同步到和远程共享版本库相同的状态。

```
$ git pull
Updating d81896e..c4acab2
Fast-forward
src/Makefile               |   21 ++++++-
src/locale/helloworld.pot   |   46 ++++++++
src/locale/zh_CN/LC_MESSAGES/helloworld.po |   62 ++++++++
src/main.c                  |   18 +++++-
4 files changed, 141 insertions(+), 6 deletions(-)
create mode 100644 src/locale/helloworld.pot
create mode 100644 src/locale/zh_CN/LC_MESSAGES/helloworld.po
```

特性分支user2/i18n也完成了历史使命，可以删除了。因为之前user2/i18n已经推送到远程共享版本库，如果想要删除分支不要忘了也将远程分支同时删除。

- 删除本地版本库的user2/i18n分支。

```
$ git branch -d user2/i18n  
Deleted branch user2/i18n (was c4acab2).
```

- 删除远程共享版本库的user2/i18n分支。

```
$ git push origin :user2/i18n  
To file:///path/to/repos/hello-world.git  
- [deleted] user2/i18n
```

补充：实际上变基之后user2/i18n分支的本地化模板文件(helloworld.pot)和汉化文件(helloworld.po)都需要做出相应更新，否则hello-world的一些输出不能进行本地化。

- 更新模板需要删除文件:`file: `helloworld.pot``，再执行命令:`command: `make po``。
- 重新翻译中文字本地化文件，可以使用工具:`command: `lokalize``或者:`command: `kbabel``。

具体的操作过程就不再赘述了。

来源：<https://github.com/gotgit/gotgit/blob/master/03-git-harmony/040-git-branch.rst>

# 远程版本库

Git作为分布式版本库控制系统，每个人都是本地版本库的主人，可以在本地的版本库中随心所欲地创建分支和里程碑。当需要多人协作时，问题就出现了：

- 如何避免因为用户把所有的本地分支都推送到共享版本库，从而造成共享版本库上分支的混乱？
- 如何避免不同用户针对不同特性开发创建了相同名字的分支而造成分支名称的冲突？
- 如何避免用户随意在共享版本库中创建里程碑而导致里程碑名称上的混乱和冲突？
- 当用户向共享版本库及其他版本库推送时，每次都需要输入长长的版本库URL，太不方便了。
- 当用户需要经常从多个不同的他人版本库中获取提交时，有没有办法不要总是输入长长的版本库URL？
- 如果不带任何其他参数执行`:command:`git fetch``、`:command:`git pull``和`:command:`git push``到底是和哪个远程版本库及哪个分支进行交互？

本章介绍的`:command:`git remote``命令就是用于实现远程版本库的便捷访问，建立远程分支和本地分支的对应，使得`:command:`git fetch``、`:command:`git pull``和`:command:`git push``能够更为便捷地进行操作。

## 远程分支

上一章介绍Git分支的时候，每一个版本库最多只和一个上游版本库（远程共享版本库）进行交互，实际上Git允许一个版本库和任意多的版本库进行交互。首先执行下面的命令，基于hello-world.git版本库再创建几个新的版本库。

```
$ cd /path/to/repos/
$ git clone --bare hello-world.git hello-user1.git
Cloning into bare repository hello-user1.git...
done.
$ git clone --bare hello-world.git hello-user2.git
Cloning into bare repository hello-user2.git...
done.
```

现在有了三个共享版本库：`hello-world.git`、`hello-user1.git`和`hello-user2.git`。现在有一个疑问，如果一个本地版本库需要和上面三个版本库进行互操作，三个共享版本库都存在一个`master`分支，会不会互相干扰、冲突或覆盖呢？

先来看看`hello-world`远程共享版本库中包含的分支有哪些：

```
$ git ls-remote --heads file:///path/to/repos/hello-world.git
8cfffe5f135821e716117ee59bdd53139473bd1d8      refs/heads/hello-1.x
bb4fef88fee435bfa04b8389cf193d9c04105a6      refs/heads/helper/master
cf71ae3515e36a59c7f98b9db825fd0f2a318350      refs/heads/helper/v1.x
c4acab26ff1c1125f5e585ffa8284d27f8ceea55      refs/heads/master
```

原来远程共享版本库中有四个分支，其中hello-1.x分支是开发者user1创建的。现在重新克隆该版本库，如下：

```
$ cd /path/to/my/workspace/
$ git clone file:///path/to/repos/hello-world.git
...
$ cd /path/to/my/workspace/hello-world
```

执行`:command:`git branch``命令检查分支，会吃惊地看到只有一个分支master。

```
$ git branch
* master
```

那么远程版本库中的其他分支哪里去了？为什么本地只有一个分支呢？执行`:command:`git show-ref``命令可以看到全部的本地引用。

```
$ git show-ref
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/heads/master
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/remotes/origin/HEAD
8cfffe5f135821e716117ee59bdd53139473bd1d8 refs/remotes/origin/hello-1.x
bb4fef88fee435bfa04b8389cf193d9c04105a6 refs/remotes/origin/helper/master
cf71ae3515e36a59c7f98b9db825fd0f2a318350 refs/remotes/origin/helper/v1.x
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/remotes/origin/master
3171561b2c9c57024f7d748a1a5cf0d755a26054a refs/tags/jx/v1.0
aaaff5676a7c3ae7712af61dfb9ba05618c74bbab refs/tags/jx/v1.0-i18n
e153f83ee75d25408f7e2fd8236ab18c0abf0ec4 refs/tags/jx/v1.1
83f59c7a88c04ceb703e490a86dde9af41de8bcb refs/tags/jx/v1.2
1581768ec71166d540e662d90290cb6f82a43bb0 refs/tags/jx/v1.3
ccca267c98380ea7ffffb241f103d1e6f34d8bc01 refs/tags/jx/v2.0
8a5b9934aacdebb72341dcadbb2650cf626d83da refs/tags/jx/v2.1
89b74222363e8cbdf91aab30d005e697196bd964 refs/tags/jx/v2.2
0b4ec63aea44b96d498528dcf3e72e1255d79440 refs/tags/jx/v2.3
60a2f4f31e5ddd777c6ad37388fe6e5520734cb refs/tags/mytag
5dc2fc52f2dc84987f511481cc6b71ec1b381f7 refs/tags/mytag3
51713af444266d56821fe3302ab44352b8c3eb71 refs/tags/v1.0
```

从`:command:`git show-ref``的输出中发现了几个不寻常的引用，这些引用以`refs/remotes/origin/`为前缀，并且名称和远程版本库的分支名一一对应。这些引用实

际上就是从远程版本库的分支拷贝过来的，称为远程分支。

Git 的:command:`git branch`命令也能够查看这些远程分支，不过要加上-r参数：

```
$ git branch -r
origin/HEAD -> origin/master
origin/hello-1.x
origin/helper/master
origin/helper/v1.x
origin/master
```

Git这样的设计是非常巧妙的，在向远程版本库执行获取操作时，不是把远程版本库的分支原封不动地复制到本地版本库的分支中，而是复制到另外的命名空间。如在克隆一个版本库时，会将远程分支都复制到目录:file:`.git/refs/remotes/origin/`下。这样向不同的远程版本库执行获取操作，因为远程分支相互隔离，所以就避免了相互的覆盖。

那么克隆操作产生的远程分支为什么都有一个名为“origin/”的前缀呢？奥秘就在配置文件:file:`.git/config`中。下面的几行内容出自该配置文件，为了说明方便显示了行号。

```
6 [remote "origin"]
7   fetch = +refs/heads/*:refs/remotes/origin/*
8   url = file:///path/to/repos/hello-world.git
```

这个小节可以称为[remote]小节，该小节以origin为名注册了一个远程版本库。该版本库的URL地址由第8行给出，会发现这个URL地址就是执行:command:`git clone`命令时所用的地址。最具魔法的配置是第7行，这一行设置了执行:command:`git fetch origin`操作时使用的默认引用表达式。

- 该引用表达式以加号（+）开头，含义是强制进行引用的替换，即使即将进行的替换是非快进式的。
- 引用表达式中使用了通配符，冒号前面的含有通配符的引用指的是远程版本库的所有分支，冒号后面的引用含义是复制到本地的远程分支目录中。

正因为有了上面的[remote]配置小节，当执行:command:`git fetch origin`操作时，就相当于执行了下面的命令，将远程版本库的所有分支复制为本地的远程分支。

```
git fetch origin +refs/heads/*:refs/remotes/origin/*
```

远程分支不是真正意义上的分支，是类似于里程碑一样的引用。如果针对远程分支执行检出命令，会看到大段的错误警告。

```
$ git checkout origin/hello-1.x
```

```
Note: checking out 'origin/hello-1.x'.
```

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 8cfffe5f... Merge branch 'hello-1.x' of file:///path/to/repos/h
```



上面的大段的错误信息实际上告诉我们一件事，远程分支类似于里程碑，如果检出就会使得头指针HEAD处于分离头指针状态。实际上除了以refs/heads为前缀的引用之外，如果检出任何其他引用，都将使工作区处于分离头指针状态。如果对远程分支进行修改就需要创建新的本地分支。

## 分支追踪

为了能够在远程分支refs/remotes/origin/hello-1.x上进行工作，需要基于该远程分支创建本地分支。远程分支可以使用简写origin/hello-1.x。如果Git的版本是1.6.6或者更新的版本，可以使用下面的命令同时完成分支的创建和切换。

```
$ git checkout hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from origin.
Switched to a new branch 'hello-1.x'
```

如果Git的版本比较老，或注册了多个远程版本库，因此存在多个名为hello-1.x的远程分支，就不能使用上面简洁的分支创建和切换命令，而需要使用在上一章中学习到的分支创建命令，显式地从远程分支中创建本地分支。

```
$ git checkout -b hello-1.x origin/hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from origin.
Switched to a new branch 'hello-1.x'
```

在上面基于远程分支创建本地分支的过程中，命令输出的第一行说的是建立了本地分支和远程分支的跟踪。和远程分支建立跟踪后，本地分支就具有下列特征：

- 检查工作区状态时，会显示本地分支和被跟踪远程分支提交之间的关系。
- 当执行:command:`git pull`命令时，会和被跟踪的远程分支进行合并（或者变基），如果两者出现版本偏离的话。

- 当执行:command:`git push`命令时，会推送到远程版本库的同名分支中。

下面就在基于远程分支创建的本地跟踪分支中进行操作，看看本地分支是如何与远程分支建立关联的。

- 先将本地hello-1.x分支向后重置两个版本。

```
$ git reset --hard HEAD^^  
HEAD is now at ebcf6d6 blank commit for GnuPG-signed tag test.
```

- 然后查看状态，显示当前分支相比跟踪分支落后了3个版本。

之所以落后三个版本而非两个版本是因为hello-1.x的最新提交是一个合并提交，包含两个父提交，因此上面的重置命令丢弃掉三个提交。

```
$ git status  
# On branch hello-1.x  
# Your branch is behind 'origin/hello-1.x' by 3 commits, and can be fast-  
#  
nothing to commit (working directory clean)
```

- 执行:command:`git pull`命令，会自动与跟踪的远程分支进行合并，相当于找回最新的3个提交。

```
$ git pull  
Updating ebcf6d6..8cfffe5f  
Fast-forward  
src/main.c | 11 +++++++--  
1 files changed, 9 insertions(+), 2 deletions(-)
```

但是如果基于本地分支创建另外一个本地分支则没有分支跟踪的功能。下面就从本地的hello-1.x分支中创建hello-jx分支。

- 从hello-1.x分支中创建新的本地分支hello-jx。

下面的创建分支操作只有一行输出，看不到分支间建立跟踪的提示。

```
$ git checkout -b hello-jx hello-1.x  
Switched to a new branch 'hello-jx'
```

- 将hello-jx分支重置。

```
$ git reset --hard HEAD^^  
HEAD is now at ebcf6d6 blank commit for GnuPG-signed tag test.
```

- 检查状态看不到分支间的跟踪信息。

```
$ git status  
# On branch hello-jx  
nothing to commit (working directory clean)
```

- 执行:command:`git pull`命令会报错。

```
$ git pull  
You asked me to pull without telling me which branch you  
want to merge with, and 'branch.hello-jx.merge' in  
your configuration file does not tell me, either. Please  
specify which branch you want to use on the command line and  
try again (e.g. 'git pull <repository> <refsref>').  
See git-pull(1) for details.
```

If you often merge with the same branch, you may want to  
use something like the following in your configuration file:

```
[branch "hello-jx"]  
remote = <nickname>  
merge = <remote-ref>  
  
[remote "<nickname>"]  
url = <url>  
fetch = <refsref>
```

See git-config(1) for details.

- 将上面命令执行中的错误信息翻译过来，就是：

```
$ git pull  
您让我执行拉回操作，但是没有告诉我您希望与哪个远程分支进行合并，  
而且也没有通过配置 'branch.hello-jx.merge' 来告诉我。
```

请在命令行提供足够的参数，如 'git pull <repository> <refsref>'。  
或者如果您经常与同一个分支进行合并，可以和该分支建立跟踪。在配置  
中添加如下配置信息：

```
[branch "hello-jx"]  
remote = <nickname>  
merge = <remote-ref>
```

```
[remote "<nickname>"]
url = <url>
fetch = <refspec>
```

为什么用同样方法建立的分支hello-1.x和hello-jx，差距咋就那么大呢？奥秘就在于从远程分支创建本地分支，自动建立了分支间的跟踪，而从一个本地分支创建另外一个本地分支则没有。看看配置文件:`file: `.git/config``中是不是专门为分支hello-1.x创建了相应的配置信息？

```
9 [branch "master"]
10 remote = origin
11 merge = refs/heads/master
12 [branch "hello-1.x"]
13 remote = origin
14 merge = refs/heads/hello-1.x
```

其中第9-11行是针对master分支设置的分支间跟踪，是在版本库克隆的时候自动建立的。而第12-14行是前面基于远程分支创建本地分支时建立的。至于分支hello-jx则没有建立相关配置。

如果希望在基于一个本地分支创建另外一个本地分支时也能够使用分支间的跟踪功能，就要在创建分支时提供`--track`参数。下面实践一下。

- 删除之前创建的hello-jx分支。

```
$ git checkout master
Switched to branch 'master'
$ git branch -d hello-jx
Deleted branch hello-jx (was ebcf6d6).
```

- 使用参数`--track`重新基于hello-1.x创建hello-jx分支。

```
$ git checkout --track -b hello-jx hello-1.x
Branch hello-jx set up to track local branch hello-1.x.
Switched to a new branch 'hello-jx'
```

- 从Git库的配置文件中会看到为hello-jx分支设置的跟踪。

因为跟踪的是本版本库的本地分支，所以第16行设置的远程版本库的名字为一个点。

```
15 [branch "hello-jx"]
```

```
16 remote = .
17 merge = refs/heads/hello-1.x
```

## 远程版本库

名为origin的远程版本库是在版本库克隆时注册的，那么如何注册新的远程版本库呢？下面将版本库file:///path/to/repos/hello-user1.git以new-remote为名进行注册。

```
$ git remote add new-remote file:///path/to/repos/hello-user1.git
```

如果再打开版本库的配置文件:file:`.git/config`会看到新的配置。

```
12 [remote "new-remote"]
13   url = file:///path/to/repos/hello-user1.git
14   fetch = +refs/heads/*:refs/remotes/new-remote/*
```

执行:command:`git remote`命令，可以更为方便地显示已经注册的远程版本库。

```
$ git remote -v
new-remote      file:///path/to/repos/hello-user1.git (fetch)
new-remote      file:///path/to/repos/hello-user1.git (push)
origin        file:///path/to/repos/hello-world.git (fetch)
origin        file:///path/to/repos/hello-world.git (push)
```

现在执行:command:`git fetch`并不会从新注册的new-remote远程版本库获取，因为当前分支设置的默认远程版本库为origin。要想从new-remote远程版本库中获取，需要为:command:`git fetch`命令增加一个参数new-remote。

```
$ git fetch new-remote
From file:///path/to/repos/hello-user1
 * [new branch]      hello-1.x  -> new-remote/hello-1.x
 * [new branch]      helper/master -> new-remote/helper/master
 * [new branch]      helper/v1.x -> new-remote/helper/v1.x
 * [new branch]      master      -> new-remote/master
```

从上面的命令输出中可以看出，远程版本库的分支复制到本地版本库前缀为new-remote的远程分支中去了。用:command:`git branch -r`命令可以看到新增了几个远程分支。

```
$ git branch -r
new-remote/hello-1.x
```

```
new-remote/helper/master
new-remote/helper/v1.x
new-remote/master
origin/HEAD -> origin/master
origin/hello-1.x
origin/helper/master
origin/helper/v1.x
origin/master
```

## 更改远程版本库的地址

如果远程版本库的URL地址改变，需要更换，该如何处理呢？手工修改:`file: `.git/config``文件是一种方法，用`command: `git config``命令进行更改是第二种方法，还有一种方法是用`command: `git remote``命令，如下：

```
$ git remote set-url new-remote file:///path/to/repos/hello-user2.git
```

可以看到注册的远程版本库的URL地址已经更改。

```
$ git remote -v
new-remote      file:///path/to/repos/hello-user2.git (fetch)
new-remote      file:///path/to/repos/hello-user2.git (push)
origin        file:///path/to/repos/hello-world.git (fetch)
origin        file:///path/to/repos/hello-world.git (push)
```

从上面的输出中可以发现每一个远程版本库都有两个URL地址，分别是执行`command: `git fetch``和`command: `git push``命令时用到的URL地址。既然有两个地址，就意味着这两个地址可以不同，用下面的命令可以为推送操作设置单独的URL地址。

```
$ git remote set-url --push new-remote /path/to/repos/hello-user2.git
$ git remote -v
new-remote      file:///path/to/repos/hello-user2.git (fetch)
new-remote      /path/to/repos/hello-user2.git (push)
origin        file:///path/to/repos/hello-world.git (fetch)
origin        file:///path/to/repos/hello-world.git (push)
```

当单独为推送设置了URL后，配置文件`file: `.git/config``的对应[remote]小节也会增加一条新的名为`pushurl`的配置。如下：

```
12 [remote "new-remote"]
13   url = file:///path/to/repos/hello-user2.git
14   fetch = +refs/heads/*:refs/remotes/new-remote/*
15   pushurl = /path/to/repos/hello-user2.git
```

## 更改远程版本库的名称

如果对远程版本库的注册名称不满意，也可以进行修改。例如将new-remote名称修改为user2，使用下面的命令：

```
$ git remote rename new-remote user2
```

完成改名后，不但远程版本库的注册名称更改过来了，就连远程分支名称都会自动进行相应的更改。可以通过执行:command:`git remote`和:command:`git branch -r`命令查看。

```
$ git remote
origin
user2
$ git branch -r
origin/HEAD -> origin/master
origin/hello-1.x
origin/helper/master
origin/helper/v1.x
origin/master
user2/hello-1.x
user2/helper/master
user2/helper/v1.x
user2/master
```

## 远程版本库更新

当注册了多个远程版本库并希望获取所有远程版本库的更新时，Git提供了一个简单的命令。

```
$ git remote update
Fetching origin
Fetching user2
```

如果某个远程版本库不想在执行:command:`git remote update`时获得更新，可以通过参数关闭自动更新。例如下面的命令关闭远程版本库user2的自动更新。

```
$ git config remote.user2.skipDefaultUpdate true
$ git remote update
Fetching origin
```

## 删除远程版本库

如果想要删除注册的远程版本库，用`:command:`git remote``的`:command:`rm``子命令可以实现。例如删除注册的user2版本库。

```
$ git remote rm user2
```

## PUSH和PULL操作与远程版本库

在Git分支一章，已经介绍过对于新建立的本地分支（没有建立和远程分支的追踪），执行`:command:`git push``命令是不会被推送到远程版本库中，这样的设置是非常安全的，避免了因为误操作将本地分支创建到远程版本库中。当不带任何参数执行`:command:`git push``命令，实际的执行过程是：

- 如果为当前分支设置了`<remote>`，即由配置`branch.<branchname>.remote`给出了远程版本库代号，则不带参数执行`:command:`git push``相当于执行了`:command:`git push <remote>``。
- 如果没有为当前分支设置`<remote>`，则不带参数执行`:command:`git push``相当于执行了`:command:`git push origin``。
- 要推送的远程版本库的URL地址由`remote.<remote>.pushurl`给出。如果没有配置，则使用`remote.<remote>.url`配置的URL地址。
- 如果为注册的远程版本库设置了`push`参数，即通过`remote.<remote>.push`配置了一个引用表达式，则使用该引用表达式执行推送。
- 否则使用“`:`”作为引用表达式。该表达式的含义是同名分支推送，即对所有在远程版本库有同名分支的本地分支执行推送。

这也就是为什么在一个本地新建分支中执行`:command:`git push``推送操作不会推送也不会报错的原因，因为远程不存在同名分支，所以根本就没有对该分支执行推送，而推送的是其他分支（如果远程版本库有同名分支的话）。

在Git分支一章中就已经知道，如果需要在远程版本库中创建分支，则执行命令：`:command:`git push <remote> <new_branch>``。即通过将本地分支推送到远程版本库的方式在远程版本库中创建分支。但是在接下来的使用中会遇到麻烦：不能执行`:command:`git pull``操作（不带参数）将远程版本库中其他人推送的提交获取到本地。这是因为没有建立本地分支和远程分支的追踪，即没有设置`branch.<branchname>.remote`的值和`branch.<branchname>.merge`的值。

关于不带参数执行`:command:`git pull``命令解释如下：

- 如果为当前分支设置了`<remote>`，即由配置`branch.<branchname>.remote`给出了远程

版本库代号，则不带参数执行`:command:`git pull``相当于执行了`:command:`git pull <remote>``。

- 如果没有为当前分支设置`<remote>`，则不带参数执行`:command:`git pull``相当于执行了`:command:`git pull origin``。
- 要获取的远程版本库的URL地址由`remote.<remote>.url`给出。
- 如果为注册的远程版本库设置了`fetch`参数，即通过`remote.<remote>.fetch`配置了一个引用表达式，则使用该引用表达式执行获取操作。
- 接下来要确定合并的分支。如果设定了`branch.<branchname>.merge`，则对其设定的分支执行合并，否则报错退出。

在执行`:command:`git pull``操作的时候可以通过参数`--rebase`设置使用变基而非合并操作，将本地分支的改动变基到跟踪分支上。为了避免因为忘记使用`--rebase`参数导致分支的合并，可以执行如下命令进行设置。注意将`<branchname>`替换为对应的分支名称。

```
$ git config branch.<branchname>.rebase true
```

有了这个设置之后，如果是在`<branchname>`工作分支中执行`:command:`git pull``命令，在遇到冲突（本地和远程分支出现偏离）的情况下，会采用变基操作，而不是默认的合并操作。

如果为本地版本库设置参数`branch.autosetuprebase`，值为`true`，则在基于远程分支建立本地追踪分支时，会自动配置`branch.<branchname>.rebase`参数，在执行`:command:`git pull``命令时使用变基操作取代默认的合并操作。

## 里程碑和远程版本库

远程版本库中的里程碑同步到本地版本库，会使用同样的名称，而不会像分支那样移动到另外的命名空间（远程分支）中，这可能会给本地版本库中的里程碑带来混乱。当和多个远程版本库交互时，这个问题就更为严重。

前面的Git里程碑一章已经介绍了当执行`:command:`git push``命令推送时，默认不会将本地创建的里程碑带入远程版本库，这样可以避免远程版本库上里程碑的泛滥。但是执行`:command:`git fetch``命令从远程版本库获取分支的最新提交时，如果获取的提交上建有里程碑，这些里程碑会被获取到本地版本库。当删除注册的远程版本库时，远程分支会被删除，但是该远程版本库引入的里程碑不会被删除，日积月累本地版本库中的里程碑可能会变得愈加混乱。

可以在执行`:command:`git fetch``命令的时候，设置不获取里程碑只获取分支及提交。通过提供`-n`或`--no-tags`参数可以实现。示例如下：

```
$ git fetch --no-tags file:///path/to/repos/hello-world.git \
refs/heads/*:refs/remotes/hello-world/*
```

在注册远程版本库的时候，也可以使用`--no-tags`参数，避免将远程版本库的里程碑引入本地版本库。例如：

```
$ git remote add --no-tags hell-world \
file:///path/to/repos/hello-world.git
```

## 分支和里程碑的安全性

通过前面章节的探讨，会感觉到Git的使用真的是太方便、太灵活了，但是需要掌握的知识点和窍门也太多了。为了避免没有经验的用户在团队共享的Git版本库中误操作，就需要对版本库进行一些安全上的设置。本书第5篇Git服务器搭建的相关章节会具体介绍如何配置用户授权等版本库安全性设置。

实际上Git版本库本身也提供了一些安全机制避免对版本库的破坏。

- 用`reflog`记录对分支的操作历史。

默认创建的带工作区的版本库都会包含`core.logallrefupdates`为`true`的配置，这样在版本库中建立的每个分支都会创建对应的`reflog`。但是创建的裸版本库默认不包含这个设置，也就不会为每个分支设置`reflog`。如果团队的规模较小，可能因为分支误操作导致数据丢失，可以考虑为裸版本库添加`core.logallrefupdates`的相关配置。

- 关闭非快进式提交。

如果将配置`receive.denyNonFastForwards`设置为`true`，则禁止一切非快进式推送。但这个配置有些矫枉过正，更好的方法是搭建基于SSH协议的Git服务器，通过钩子脚本更灵活的进行配置。例如：允许来自某些用户的强制提交，而其他用户不能执行非快进式推送。

- 关闭分支删除功能。

如果将配置`receive.denyDeletes`设置为`true`，则禁止删除分支。同样更好的方法是通过架设基于SSH协议的Git服务器，配置分支删除的用户权限。

# 补丁文件交互

之前各个章节版本库间的交互都是通过`:command:`git push``和/或`:command:`git pull``命令实现的，这是Git最主要的交互模式，但并不是全部。使用补丁文件是另外一种交互方式，适用于参与者众多的大型项目进行分布式开发。例如Git项目本身的代码提交就主要由贡献者通过邮件传递补丁文件实现的。作者在写书过程中发现了Git的两个bug，就是以补丁形式通过邮件贡献给Git项目的，下面两个链接就是相关邮件的存档。

- 关于Git文档错误的bugfix:

<http://marc.info/?l=git&m=129248415230151>

- 关于`:command:`git-apply``的一个bugfix:

<http://article.gmane.org/gmane.comp.version-control.git/162100>

这种使用补丁文件进行提交的方式可以提高项目的参与度。因为任何人都可以参与项目的开发，只要会将提交转化为补丁，会发邮件即可。

## 创建补丁

Git提供了将提交批量转换为补丁文件的命令：`:command:`git format-patch``。该命令后面的参数是一个版本范围列表，会将包含在此列表中的提交一一转换为补丁文件，每个补丁文件包含一个序号并从提交说明中提取字符串作为文件名。

下面演示一下在user1工作区中，如何将master分支的最近3个提交转换为补丁文件。

- 进入user1工作区，切换到master分支。

```
$ cd /path/to/user1/workspace/hello-world/  
$ git checkout master  
$ git pull
```

- 执行下面的命令将最近三个提交转换为补丁文件。

```
$ git format-patch -s HEAD~3..HEAD  
0001-Fix-typo-help-to-help.patch  
0002-Add-I18N-support.patch  
0003-Translate-for-Chinese.patch
```

在上面的`:command:`git format-patch``命令中使用了`-s`参数，会在导出的补丁文件中添加

当前用户的签名。这个签名并非GnuPG式的数字签名，不过是将作者姓名添加到提交说明中而已，和在本书第2篇开头介绍的:command:`git commit -s`命令的效果相同。虽然签名很不起眼，但是对于以补丁方式提交数据却非常重要，因为以补丁方式提交可能因为合并冲突或其他原因使得最终提交的作者显示为管理员（提交者）的ID，在提交说明中加入原始作者的署名信息大概是作者唯一露脸的机会。如果在提交时忘了使用-s参数添加签名，可以在用:command:`git format-path`命令创建补丁文件的时候补救。

看一下补丁文件的文件头，在下面代码中的第7行可以看到新增的签名。

```
1 From d81896e60673771ef1873b27a33f52df75f70515 Mon Sep 17 00:00:00 2001
2 From: user1 <user1@sun.ossexp.com>
3 Date: Mon, 3 Jan 2011 23:48:56 +0800
4 Subject: [PATCH 1/3] Fix typo: -help to --help.
5
6
7 Signed-off-by: user1 <user1@sun.ossexp.com>
8 ---
9 src/main.c |    2 ++
10 1 files changed, 1 insertions(+), 1 deletions(-)
```

补丁文件有一个类似邮件一样的文件头（第1-4行），提交日志的第一行作为邮件标题（Subject），其余提交说明作为邮件内容（如果有的话），文件补丁用三个横线和提交说明分开。

实际上这些补丁文件可以直接拿来作为邮件发送给项目的负责人。Git提供了一个辅助邮件发送的命令:command:`git send-email`。下面用该命令将这三个补丁文件以邮件形式发送出去。

```
$ git send-email *.patch
0001-Fix-typo-help-to-help.patch
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
The following files are 8bit, but do not declare a Content-Transfer-Encoding.
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
Which 8bit encoding should I declare [UTF-8]?
Who should the emails appear to be from? [user1 <user1@sun.ossexp.com>]

Emails will be sent from: user1 <user1@sun.ossexp.com>
Who should the emails be sent to? jiangxin
Message-ID to be used as In-Reply-To for the first email?
...
Send this email? ([y]es|[n]o|[q]uit|[a]ll): a
...
```

命令:`git send-email` 提供交互式字符界面，输入正确的收件人地址，邮件就批量地发送出去了。

## 应用补丁

在前面通过:`git send-email` 命令发送邮件给`jiangxin`用户。现在使用 Linux 上的:`mail` 命令检查一下邮件。

```
$ mail
Mail version 8.1.2 01/15/2001. Type ? for help.
"/var/mail/jiangxin": 3 messages 3 unread
>N 1 user1@sun.osxp.c Thu Jan 13 18:02 38/1120 [PATCH 1/3] Fix typo: -h
 N 2 user1@sun.osxp.c Thu Jan 13 18:02 227/6207 =?UTF-8?q?=5BPATCH=202/3
 N 3 user1@sun.osxp.c Thu Jan 13 18:02 95/2893 =?UTF-8?q?=5BPATCH=203/3
&
```

如果邮件不止这三封，需要将三个包含补丁的邮件挑选出来保存到另外的文件中。在 `mail` 命令的提示符(&)下输入命令。

```
& s 1-3 user1-mail-archive
"user1-mail-archive" [New file]
& q
```

上面的操作在本地创建了一个由开发者`user1`的补丁邮件组成的归档文件`user1-mail-archive`，这个文件是mbox格式的，可以用:`mail` 命令打开。

```
$ mail -f user1-mail-archive
Mail version 8.1.2 01/15/2001. Type ? for help.
"user1-mail-archive": 3 messages
> 1 user1@sun.osxp.c Thu Jan 13 18:02 38/1121 [PATCH 1/3] Fix typo: -h
 2 user1@sun.osxp.c Thu Jan 13 18:02 227/6208 =?UTF-8?q?=5BPATCH=202/3
 3 user1@sun.osxp.c Thu Jan 13 18:02 95/2894 =?UTF-8?q?=5BPATCH=203/3
& q
```

保存在mbox中的邮件可以批量的应用在版本库中，使用:`git am` 命令。`am`是`apply email`的缩写。下面就演示一下如何应用补丁。

- 基于`HEAD~3`版本创建一个本地分支，以便在该分支下应用补丁。

```
$ git checkout -b user1 HEAD~3
```

```
Switched to a new branch 'user1'
```

- 将mbox文件user1-mail-archive中的补丁全部应用在当前分支上。

```
$ git am user1-mail-archive
Applying: Fix typo: -help to --help.
Applying: Add I18N support.
Applying: Translate for Chinese.
```

- 补丁成功应用上了，看看提交日志。

```
$ git log -3 --pretty=fuller
commit 2d9276af9df1a2fdb71d1e7c9ac6dff88b2920a1
Author: Jiang Xin <jiangxin@osssxp.com>
AuthorDate: Thu Jan 13 18:02:03 2011 +0800
Commit: user1 <user1@sun.osssxp.com>
CommitDate: Thu Jan 13 18:21:16 2011 +0800

    Translate for Chinese.

    Signed-off-by: Jiang Xin <jiangxin@osssxp.com>
    Signed-off-by: user1 <user1@sun.osssxp.com>

commit 41227f492ad37cdd99444a5f5cc0c27288f2bca4
Author: Jiang Xin <jiangxin@osssxp.com>
AuthorDate: Thu Jan 13 18:02:02 2011 +0800
Commit: user1 <user1@sun.osssxp.com>
CommitDate: Thu Jan 13 18:21:15 2011 +0800

    Add I18N support.

    Signed-off-by: Jiang Xin <jiangxin@osssxp.com>
    Signed-off-by: user1 <user1@sun.osssxp.com>

commit 4a3380fb7ae90039633dec84acc2aab85398efad
Author: user1 <user1@sun.osssxp.com>
AuthorDate: Thu Jan 13 18:02:01 2011 +0800
Commit: user1 <user1@sun.osssxp.com>
CommitDate: Thu Jan 13 18:21:15 2011 +0800

    Fix typo: -help to --help.

    Signed-off-by: user1 <user1@sun.osssxp.com>
```

从提交信息上可以看出：

- 提交的时间信息使用了邮件发送的时间。

- 作者（Author）的信息被保留，和补丁文件中的一致。
- 提交者（Commit）全都设置为user1，因为提交是在user1的工作区完成的。
- 提交说明中的签名信息被保留。实际上:command:`git am`命令也可以提供-s参数，在提交说明中附加执行命令用户的签名。

对于不习惯在控制台用:command:`mail`命令接收邮件的用户，可以通过邮件附件，U盘或其他方式获取:command:`git format-patch`生成的补丁文件，将补丁文件保存在本地，通过管道符调用:command:`git am`命令应用补丁。

```
$ ls *.patch
0001-Fix-typo-help-to-help.patch  0002-Add-I18N-support.patch  0003-Translate
$ cat *.patch | git am
Applying: Fix typo: -help to --help.
Applying: Add I18N support.
Applying: Translate for Chinese.
```



Git还提供一个命令:command:`git apply`，可以应用一般格式的补丁文件，但是不能执行提交，也不能保持补丁中的作者信息。实际上:command:`git apply`命令和GNU:command:`patch`命令类似，细微差别在本书第7篇第38章“补丁中的二进制文件”予以介绍。

## StGit和Quilt

一个复杂功能的开发一定是由多个提交来完成的，对于在以接收和应用补丁文件为开发模式的项目中，复杂的功能需要通过多个补丁文件来完成。补丁文件因为要经过审核才能被接受，因此针对一个功能的多个补丁文件一定要保证各个都是精品：补丁1用来完成一个功能点，补丁2用来完成第二个功能点，等等。一定不能出现这样的情况：补丁3用于修正补丁1的错误，补丁10改正了补丁7中的文字错误，等等。这样就带来补丁管理的难题。

实际上基于特性分支的开发又何尝不是如此？在将特性分支归并到开发主线前，要接受团队的评审，特性分支的开发者一定想将特性分支上的提交进行重整，把一些提交合并或者拆分。使用变基命令可以实现提交的重整，但是操作起来会比较困难，有什么好办法呢？

## StGit

Stacked Git (<http://www.procode.org/stgit/>) 简称StGit就是解决上述两个难题的答案。实际上StGit在设计上参考了一个著名的补丁管理工具Quilt，并且可以输出Quilt兼容的补丁列表。在本节的后半部分会介绍Quilt。

StGit是一个Python项目，安装起来还是很方便的。在Debian/Ubuntu下，可以直接通过包管理器安装：

```
$ sudo aptitude install stgit stgit-contrib
```

下面还是用hello-world版本库，进行StGit的实践。

- 首先检出hello-world版本库。

```
$ cd /path/to/my/workspace/  
$ git clone file:///path/to/repos/hello-world.git stgit-demo  
$ cd stgit-demo
```

- 在当前工作区初始化StGit。

```
$ stg init
```

- 现在补丁列表为空。

```
$ stg series
```

- 将最新的三个提交转换为StGit补丁。

```
$ stg uncommit -n 3  
Uncommittting 3 patches ...  
Now at patch "translate-for-chinese"  
done
```

- 现在补丁列表中有三个文件了。

第一列是补丁的状态符号。加号(+)代表该补丁已经应用在版本库中，大于号(>)用于标识当前的补丁。

```
$ stg ser  
+ fix-typo-help-to-help  
+ add-i18n-support  
> translate-for-chinese
```

- 现在查看master分支的日志，发现和之前没有两样。

```
$ git log -3 --oneline  
c4acab2 Translate for Chinese.
```

```
683448a Add I18N support.  
d81896e Fix typo: -help to --help.
```

- 执行StGit补丁出栈的命令，会将补丁撤出应用。使用-a参数会将所有补丁撤出应用。

```
$ stg pop  
Popped translate-for-chinese  
Now at patch "add-i18n-support"  
$ stg pop -a  
Popped add-i18n-support -- fix-typo-help-to-help  
No patch applied
```

- 再来看版本库的日志，会发现最新的三个提交都不见了。

```
$ git log -3 --oneline  
10765a7 Bugfix: allow spaces in username.  
0881ca3 Refactor: use getopt_long for arguments parsing.  
ebcf6d6 blank commit for GnuPG-signed tag test.
```

- 查看补丁列表的状态，会看到每个补丁前都用减号(-)标识。

```
$ stg ser  
- fix-typo-help-to-help  
- add-i18n-support  
- translate-for-chinese
```

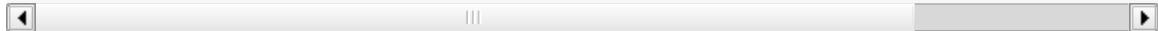
- 执行补丁入栈，即应用补丁，使用命令:`command: `stg push``或者:`command: `stg goto``命令，注意:`command: `stg push``命令和:`command: `git push``命令风马牛不相及。

```
$ stg push  
Pushing patch "fix-typo-help-to-help" ... done (unmodified)  
Now at patch "fix-typo-help-to-help"  
$ stg goto add-i18n-support  
Pushing patch "add-i18n-support" ... done (unmodified)  
Now at patch "add-i18n-support"
```

- 现在处于应用add-i18n-support补丁的状态。这个补丁有些问题，本地化语言模板有错误，我们来修改一下。

```
$ cd src/  
$ rm locale/helloworld.pot
```

```
$ make po  
xgettext -s -k_ -o locale/helloworld.pot main.c  
msgmerge locale/zh_CN/LC_MESSAGES/helloworld.po locale/helloworld.pot -o  
. 完成。  
mv locale/temp.po locale/zh_CN/LC_MESSAGES/helloworld.po
```



- 现在查看工作区，发现工作区有改动。

```
$ git status -s  
M locale/helloworld.pot  
M locale/zh_CN/LC_MESSAGES/helloworld.po
```

- 不要将改动添加暂存区，也不要提交，而是执行:command:``stg refresh``命令，更新补丁。

```
$ stg refresh  
Now at patch "add-i18n-support"
```

- 这时再查看工作区，发现本地修改不见了。

```
$ git status -s
```

- 执行:command:``stg show``会看到当前的补丁`add-i18n-support`已经更新。

```
$ stg show  
...
```

- 将最后一个补丁应用到版本库，遇到冲突。这是因为最后一个补丁是对中文本地化文件的翻译，因为翻译前的模板文件被更改了所以造成了冲突。

```
$ stg push  
Pushing patch "translate-for-chinese" ... done (conflict)  
Error: 1 merge conflict(s)  
    CONFLICT (content): Merge conflict in  
        src/locale/zh_CN/LC_MESSAGES/helloworld.po  
Now at patch "translate-for-chinese"
```

- 这个冲突文件很好解决，直接编辑冲突文件:file:``helloworld.po``即可。编辑好之后，注意一下第50行和第62行是否像下面写的一样。

```
50 "      hello -h, --help\n"
51 "          显示本帮助页。\\n"
...
61 msgid "Hi,"
62 msgstr "您好,"
```

- 执行:command:`git add`命令完成冲突解决。

```
$ git add locale/zh_CN/LC_MESSAGES/helloworld.po
```

- 不要提交，而是使用:command:`stg refresh`命令更新补丁，同时更新提交。

```
$ stg refresh
Now at patch "translate-for-chinese"
$ git status -s
```

- 看看修改后的程序，是不是都能显示中文了。

```
$ ./hello
世界你好。
(version: v1.0-5-g733c6ea)
$ ./hello Jiang Xin
您好，Jiang Xin.
(version: v1.0-5-g733c6ea)
$ ./hello -h
...
```

- 导出补丁，使用命令:command:`stg export`。导出的是Quilt格式的补丁集。

```
$ cd /path/to/my/workspace/stgit-demo/
$ stg export -d patches
Checking for changes in the working directory ... done
```

- 看看导出补丁的目标目录。

```
$ ls patches/
add-i18n-support  fix-typo-help-to-help  series  translate-for-chinese
```

- 其中文件:file:`series`是补丁文件的列表，列在前面的补丁先被应用。

```
$ cat patches/series
# This series applies on GIT commit d81896e60673771ef1873b27a33f52df75f70
```

```
fix-typo-help-to-help  
add-i18n-support  
translate-for-chinese
```



通过上面的演示可以看出StGit可以非常方便的对提交进行整理，整理提交时无需使用复杂的变基命令，而是采用：提交StGit化，修改文件，执行`:command:`stg refresh``的工作流程即可更新补丁和提交。StGit还可以将补丁导出为补丁文件，虽然导出的补丁文件没有像`:command:`git format-patch``那样加上代表顺序的数字前缀，但是用文件`:file:`series``标注了补丁文件的先后顺序。实际上可以在执行`:command:`stg export``时添加`-n`参数为补丁文件添加数字前缀。

StGit还有一些功能，如合并补丁/提交，插入新补丁/提交等，请参照StGit帮助，恕不一一举例。

## Quilt

Quilt是一款补丁列表管理软件，用Shell语言开发，安装也很简单，在Debian/Ubuntu上直接用下面的命令即可安装：

```
$ sudo aptitude install quilt
```

Quilt约定俗成将补丁集放在项目根目录下的子目录`:file:`patches``中，否则需要通过环境变量`QUILT_PATCHES`对路径进行设置。为了减少麻烦，在上面用`:command:`stg export``导出补丁的时候就导出到了`:file:`patches``目录下。

简单说一下Quilt的使用，会发现真的和StGit很像，实际上是先有的Quilt，后有的StGit。

- 重置到三个提交前的版本，否则应用补丁的时候会失败。还不要忘了删除`:file:`src/locale``目录。

```
$ git reset --hard HEAD~3  
$ rm -rf src/locale/
```

- 显示补丁列表

```
$ quilt series  
01-fix-typo-help-to-help  
02-add-i18n-support  
03-translate-for-chinese
```

- 应用一个补丁。

```
$ quilt push  
Applying patch 01-fix-typo-help-to-help  
patching file src/main.c
```

```
Now at patch 01-fix-typo-help-to-help
```

- 下一个补丁是什么？

```
$ quilt next  
02-add-i18n-support
```

- 应用全部补丁。

```
$ quilt push -a  
Applying patch 02-add-i18n-support  
patching file src/Makefile  
patching file src/locale/helloworld.pot  
patching file src/locale/zh_CN/LC_MESSAGES/helloworld.po  
patching file src/main.c  
  
Applying patch 03-translate-for-chinese  
patching file src/locale/zh_CN/LC_MESSAGES/helloworld.po
```

```
Now at patch 03-translate-for-chinese
```

Quilt的功能还有很多，请参照Quilt的联机帮助，恕不一一举例。

来源: <https://github.com/gotgit/gotgit/blob/master/03-git-harmony/060-git-offline.rst>

# 经典Git协同模型

## 集中式协同模型

可以像集中式版本控制系统那样使用Git，在一个大家都可以访问到的服务器上架设Git服务器，每个人从该服务器克隆代码，本地提交推送到服务器上。如图21-1所示。

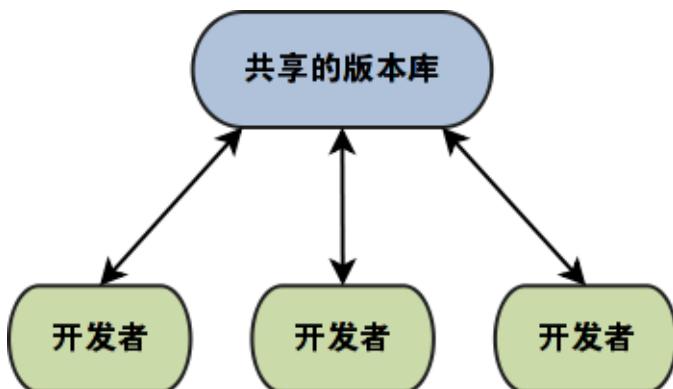


图21-1：集中式协同模型

回忆一下在使用Subversion等集中式版本控制系统时，对服务器管理上的要求：

- 只允许拥有帐号的用户访问版本库。
- 甚至只允许用户访问版本库中的某些路径，其他路径不能访问。
- 特定目录只允许特定用户执行写操作。
- 服务器可以通过钩子实现特殊功能，如对提交说明（commit log）的检查，数据镜像等。

对于这些需求，Git大部分都能支持，甚至能够做到更多：

- 能够设置谁能够访问版本库，谁不能访问版本库。
- 具有更为丰富的写操作授权。可以限制哪些分支不允许写，哪些路径不允许写。
- 可以设置谁可以创建新的分支。
- 可以设置谁可以创建新的版本库。
- 可以设置谁可以强制更新。
- 服务器端同样支持钩子脚本。

但是也要承认，在“读授权”上Git做不到很精细，这是分布式版本控制系统的机制使然。按模块分解Git版本库，并结合后面介绍的多版本库协同解决方案可以克服Git读授权的局限。

- Git不支持对版本库读取的精确授权，只能是非零即壹的授权。即或者能够读取一个版本库的全部，或者什么也读不到。

- 因为Git的提交是一个整体，提交中包含了完整目录树（tree）的哈希，因此完整性不容破坏。
- Git是分布式版本控制系统，如果允许不完整的克隆，那么本地就是截然不同的版本库，在向服务器推送的时候，会被拒绝或者产生新的分支。

### 用Gitolite架设集中式的Git服务器

对于集中式的工作模型的核心就是架设集中式的Git服务器，而且尽量能够满足前面提到的对授权和版本库管理上的需求。在本书第5篇介绍服务器部署的时候，会介绍用Gitolite架设Git服务器，可以实现集中式协同模型对版本库授权和管理上的要求。

## 使用集中协同模型

对于简单的代码修改，可以像传统集中式版本控制系统（Subversion）中那样工作，参照图21-2所示的工作流程图。

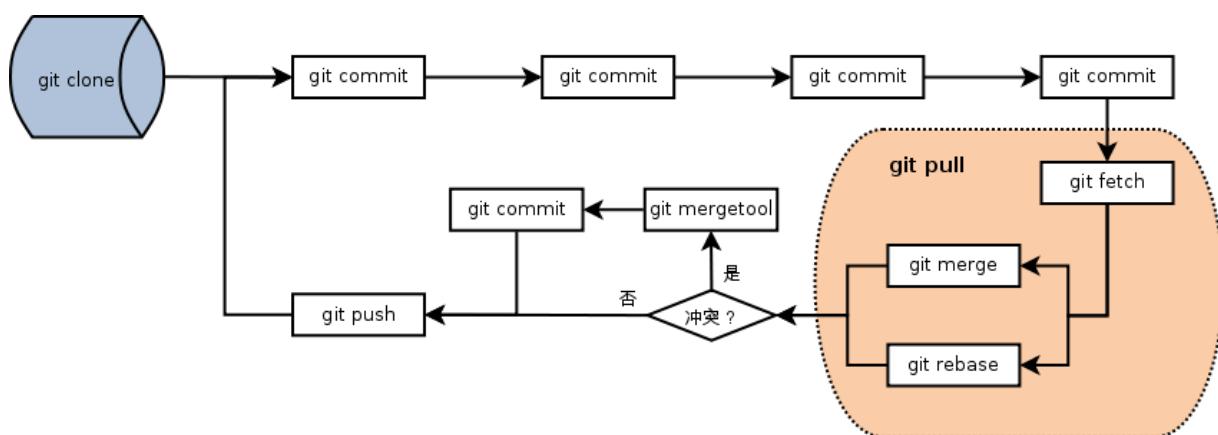


图21-2：集中式协同模型工作流1

但是对于复杂的修改（代码重构/增加复杂功能），这个工作模式就有些不合适了。

第一个问题是：很容易将不成熟代码带入共享的版本库，破坏共享版本库相应分支的代码稳定性。例如破坏编译、破坏每日集成。这是因为开发者克隆版本库后，直接工作在缺省的跟踪分支上，当不小心执行`:command:`git push`命令`，就会将自己的提交推送到服务器上。

为了避免上面的问题，开发者可能会延迟推送，例如在新功能定制的整个过程（一个月）只在本地提交，而不向服务器推送，这样产生更严重的问题：数据丢失。开发者可能因为操作系统感染病毒，或者不小心的目录删除，或者硬盘故障导致工作成果的彻底丢失，这对个人和团队来说都是灾难。

解决这个问题的方法也很简单，就是在本地创建本地分支（功能分支），并且同时在服务器端（共享版本库）也创建自己独享的功能分支。本地提交推送到共享版本库的自己独享的分支上。当开发完成之后，将功能分支合并到主线上，推送到共享版本库，完成开发。

当然如果该特性分支不再需要时需要作些清理工作。参见图21-3所示的工作流程图。

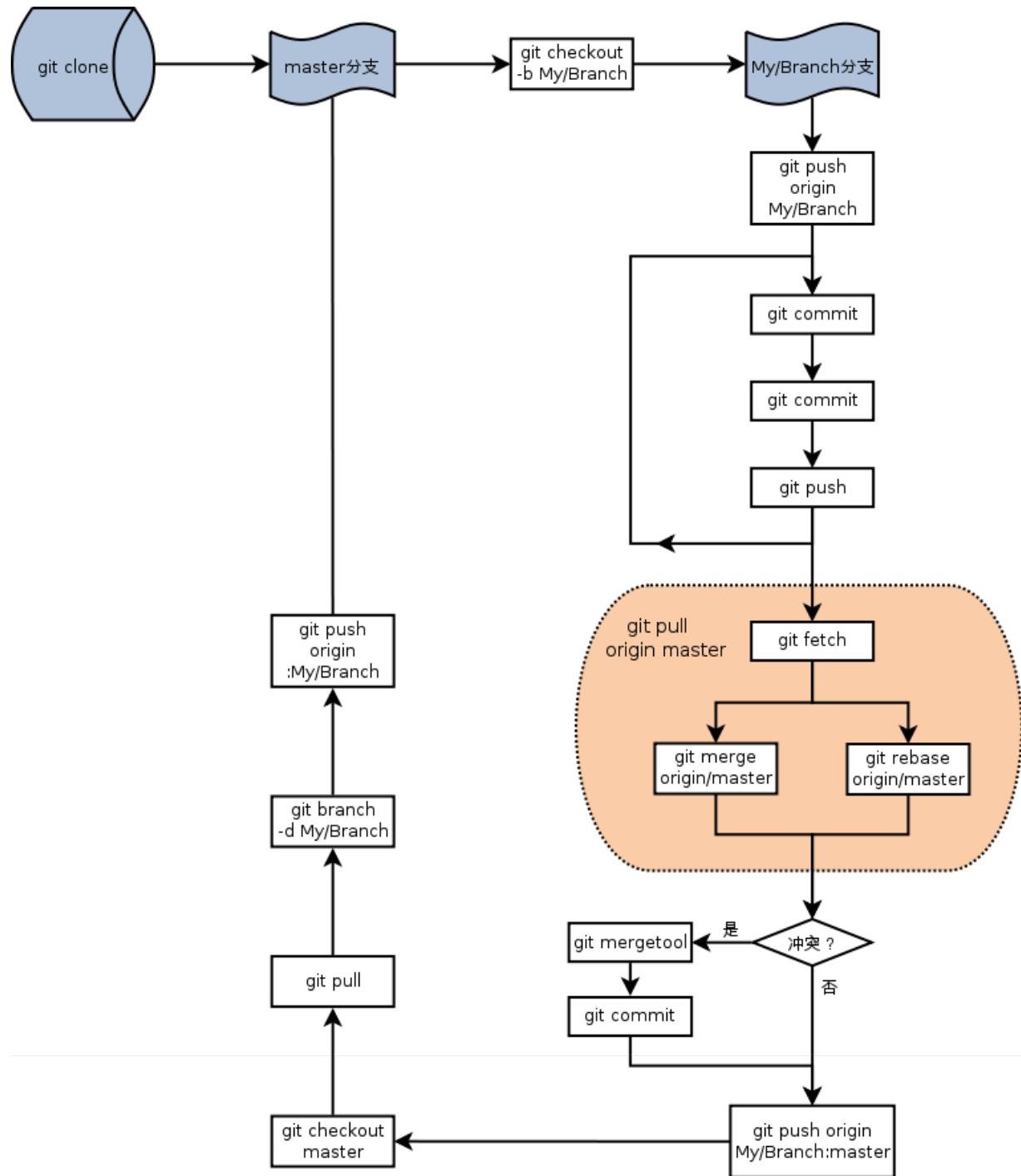


图21-3：集中式协同模型工作流2

## Gerrit特殊的集中协同模型

### 传统集中式协同模型的缺点

传统集中式协同模型的主要问题是在管理上：谁能够向版本库推送？可以信赖某人向版本库推送么？

对于在一个相对固定的团队内部使用集中式协同模型没有问题，因为大家彼此信赖，都熟

悉项目相关领域。但是对于公开的项目（开源项目）来说，采用集中式的协同模型，必然只能有部分核心人员具有“写”权限，很多有能力的参与者被拒之门外，这不利于项目的发展。因此集中式协同模型主要应用在公司范围内和商业软件开发中，而不会成为开源项目的首选。

### 强制代码审核的集中式协同模型

Android项目采用了独树一帜的集中式管理模型——通过Gerrit架设的审核服务器对提交进行强制审核。Android是由大约近200个Git版本库组成的庞大的项目，为了对庞大的版本库进行管理，Android项目开发了两个工具repo和Gerrit进行版本库的管理。其中Gerrit服务器为Android项目引入了特别的集中式协同模型。

Gerrit服务器通过SSH协议管理Git版本库，并实现了一个Web界面的评审工作流。任何注册用户都可以参与到项目中来，都可以推送Git提交到Gerrit管理下的Git版本库（通过Gerrit启动的特殊SSH端口）。Git推送不能直接推送到分支，而是推送到特殊的引用`refs/for/<branch-name>`，此提交会自动转换为形如`refs/changes/<nn>/<review-id>/<patch-set>`的补丁集，此补丁集在Gerrit的Web界面中显示为对应的评审任务。评审任务进入审核流程，当通过相关负责人的审核后，才被接受，被合并到正式的版本库中。

在本书的第5篇第32章“Gerrit代码审核服务器”中会详细介绍Gerrit代码审核服务器的部署和使用。

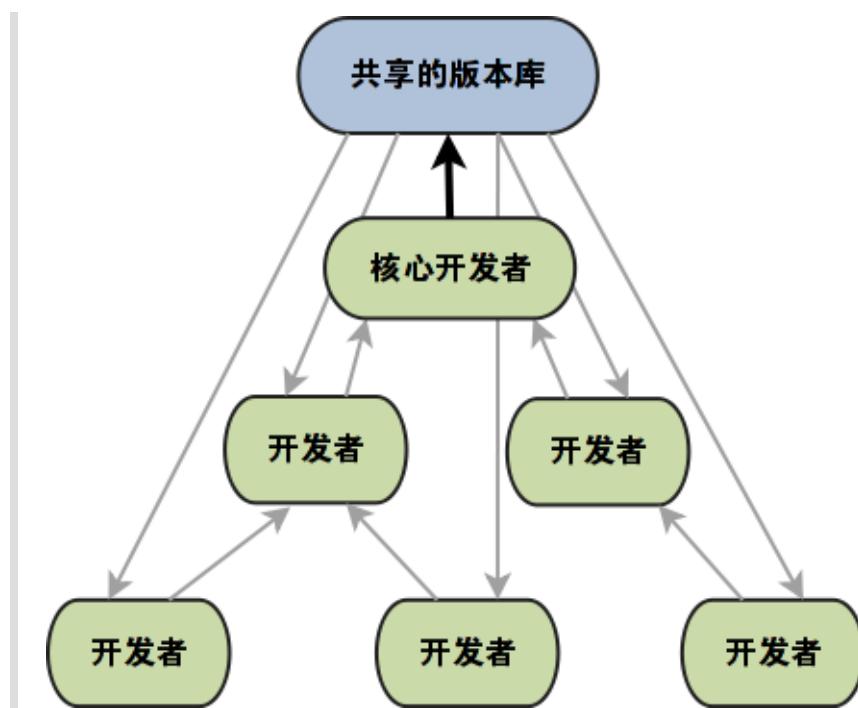
来源：<https://github.com/gotgit/gotgit/blob/master/04-git-model/010-central-model.rst>

# 金字塔式协同模型

自从分布式版本库控制系统（Mercurial/Hg、Bazaar、Git等）诞生之后，有越来越多的开源项目迁移了版本控制系统，例如从Subversion或CVS迁移到分布式版本控制系统。因为众多的开源项目逐渐认识到，集中式的版本控制管理方式阻止了更多的人参与项目的开发，对项目的发展不利。

集中式版本控制系统的最大问题是，如果没有在服务器端授权，就无法提交，也就无法保存自己的更改。开源项目虽然允许所有人访问代码库，但是不可能开放“写操作”授权给所有的人，否则代码质量无法控制（Gerrit审核服务器是例外）。于此相对照的是，在使用了分布式版本控制系统之后，任何人都可以在本地克隆一个和远程版本库一模一样的版本库，本地的版本库允许任何操作，这就极大的调动了开发者投入项目研究的积极性。

分布式的开发必然带来协同的问题，如何能够让一个素不相识的开发者将他的贡献提交到项目中？如何能够最大化的发动和汇聚全球智慧？开源社区逐渐发展出金字塔模型，而这也是必然之选。



图：金字塔式协同模型

金字塔模型的含义是，虽然理论上每个开发者的版本库都是平等的，但是会有一个公认的权威的版本库，这个版本库由一个或者多个核心开发者负责维护（具有推送的权限）。核心的开发人员负责审核其他贡献者的提交，审核可以通过邮件传递的补丁或者访问（PULL）贡献者开放的代码库进行。由此构成了由核心开发团队为顶层的所有贡献者共同参与的开发者金字塔。

Linux社区就是典型的金字塔结构。Linus Torvalds的版本库被公认为是官方的版本库，允

许核心成员的提交。其他贡献者的提交必须经过一个或多个核心成员的审核后，才能经由核心成员代为推送的到官方版本库。

采用这种金字塔式协同模型不需要复杂的Git服务器设置，只需要项目管理者提供一个让其他人只读访问的版本库。当然管理者要能够通过某种方法向该版本库推送，以便其他人能够通过该版本库获得更新。

## 贡献者开放只读版本库

因为不能直接向项目只读共享的版本库提交，为了能让项目的管理者获取自己的提交，贡献者需要提供项目管理者访问自己版本库的方法。建立一个自己所有的只读共享版本库是一个简单易行的方法。在第5篇搭建Git服务器的相关章节，会介绍几种快速搭建只读Git版本库的方法，包括：用HTTP智能协议搭建Git服务器，用Git协议搭建Git服务器。

贡献者在自己的只读共享版本库建立后，需要对自己贡献的提交进行一下检查和整理。

- 贡献的提交要处于一个单独的特性分支中，并且要为该特性分支取一个有意义的名字。

使用贡献者的名字以及简单的概括性文字是非常好的特性分支名。例如为自己的特性分支命名为jiangxin/fix-bug-xxx。

- 贡献的提交是否是基于上游对应分支的最新提交？如果不是需要变基到上游最新提交，以免产生合并。

项目的管理者会尽量避免不必要的合并，因此会要求贡献者的提交尽量基于项目的最新提交。使用下面的方式建立跟踪远程分支的本地分支，可以很简单的实现在执行：`command: `git pull`` 操作时使用变基操作取代合并操作。

```
$ git checkout -b jiangxin/fix-bug-xxx origin/master
$ git config branch.jiangxin/fix-bug-xxx.rebase true
hack, hack, hack
$ git pull
```

然后贡献者就可以向项目管理者发送通知邮件，告诉项目管理者有贡献的代码等待他/她的审核。邮件中大致包括以下内容：

- 为什么要修改项目的代码。
- 相应的修改是否经过了测试，或者提交中是否包含了单元测试。
- 自己版本库的访问地址。
- 特性分支名称。

## 以补丁方式贡献代码

使用补丁文件方式贡献代码也是开源项目常用的协同方式。Git项目本身就是采用该方式运作的。

- 每个用户先在本地版本库修改代码。
- 修改完成后，通过执行`:command:`git format-patch``命令将提交转换为补丁。
- 如果提交很多且比较杂乱，可以考虑使用StGit对提交进行重整。
- 调用`:command:`git send-email``命令或者通过图形界面的邮件客户端软件将补丁发到邮件列表以及项目维护者。
- 项目维护者认可贡献者提交的补丁后，执行`:command:`git am``命令应用补丁。

在第3篇第20章“补丁文件交互”已经详细介绍了该模式的工作流程，请参考相关章节。

来源：<https://github.com/gotgit/gotgit/blob/master/04-git-model/020-distribute-model.rst>

# Topgit协同模型

如果没有Topgit，就不会有此书。因为发现了Topgit，才让作者下定决心在公司大范围推广Git；因为Topgit，激发了作者对Git的好奇之心。

## 作者版本控制系统三个里程碑

从2005年开始作者专心于开源软件的研究、定制开发和整合，在这之后的几年，一直使用Subversion做版本控制。对于定制开发工作，Subversion有一种称为卖主分支（Vendor Branch）的模式。

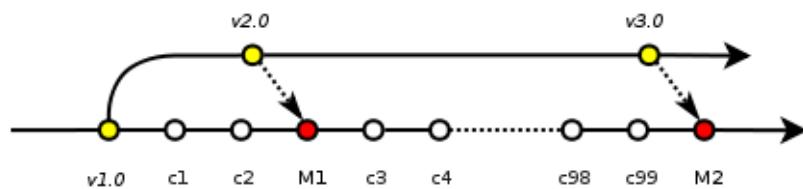


图22-1：卖主分支工作模式图

卖主分支的工作模式如图22-1所示：

- 图22-1由左至右，提交随着时间而递增。
- 主线trunk用于对定制开发的过程进行跟踪。
- 主线的第一个提交v1.0是导入上游（该开源软件官方版本库）发布的版本。
- 之后在v1.0提交之处建立分支，是为卖主分支（vendor branch）。
- 主线上依次进行了c1、c2两次提交，是基于v1.0进行的定制开发。
- 当上游有了新版本，提交到卖主分支上，即v2.0提交。和v1.0相比除了大量的文件更改外，还可能有文件增加和删除。
- 然后在主线上执行从卖主分支到主线的合并，即提交M1。因为此时主线上的改动相对少，合并v2.0并不太费事。
- 主线继续开发。可能同时有针对不同需求的定制开发，在主线上会有越来越多的提交，如上图从c3到c99近百次提交。
- 如果在卖主分支上导入上游的新版本v3.0，合并将会非常痛苦。因为主线上针对不同需求的定制开发已经混在一起！

实践证明，Subversion的卖主分支对于大规模的定制开发非常不适合。向上游新版本的迁移随着定制功能和提交的增多越来越困难。

在2008年，我们的版本库迁移到Mercurial（水银，又称为Hg），并工作在“Hg+MQ”模式下，我自以为找到了定制开发版本控制的终极解决方案，那时我们已被Subversion的卖主分支折磨的太久了。

Hg和Git一样也是一种分布式版本控制系统，MQ是Hg的一个扩展，可以实现提交和补丁两种模式之间的转换。Hg版本库上的提交可以通过`:command:`hg qimport``命令转化为补丁列表，也可以通过`:command:`hg qpush``、`:command:`hg qpop``等命令在补丁列表上游移（出栈和入栈），入栈的补丁转化为Hg版本库的提交，补丁出栈会从Hg版本库移走最新的提交。

使用“Hg+MQ”相比Subversion的卖主分支的好处在于：

- 针对不同需求的定制开发，其提交被限定在各自独立的补丁文件之中。

针对同一个需求的定制开发，无论多少次的更改都体现为补丁文件的变化，而补丁文件本身也是被版本控制的。

- 各个补丁之间是顺序依赖关系，形成一个Quilt格式的补丁列表。
- 迁移至上游新版本的过程是：先将所有补丁“出栈”，再将上游新版本提交到主线，然后依次将补丁“入栈”。

因为上游新版本的代码上下文改变等原因，补丁入栈可能会遇到冲突，只要在解决冲突完毕后，执行`:command:`hg qref``即可。

- 向上游新版本迁移过程的工作量降低了，是因为提交都按照定制的需求分类了（不同的补丁），每个补丁都可以视为一个功能分支。

但是当需要在定制开发上进行多人协作的时候，“Hg+MQ”弊病就显现了。因为“Hg+MQ”工作模式下，定制开发的成果是一个补丁库，在补丁库上进行协作难度非常大，当发生冲突的时候，补丁文件本身的冲突解决难度相当大。这就引发了我们第三次版本控制系统大迁移。

2009年，目光锁定在Topgit上。TopGit的项目名称是来自于Topic Git的简写，是基于Git用脚本语言开发的辅助工具，是用于管理多个Git的特性分支的工具。Topgit可以非常简单的实现“变基”——迁移至上游新版本。

Topgit的主要特点有：

- 上游代码库位于开发主线（如：`master`分支），每一个定制开发都对应于一条Git分支（`refs/heads/t/feature_name`）。
- 特性分支之间的依赖关系不像“Hg+MQ”简单的逐一依赖模式，而是可以任意设定分支之间的依赖。
- 特性分支和其依赖的分支可以转出为Quilt格式的补丁列表。
- 因为针对某一需求的定制开发在特定的分支中，可以多人协同参与，和正常的Git开发别无二致。

## Topgit原理

图22-2是一个近似的Topgit实现图（略去了重要的top-bases分支）。

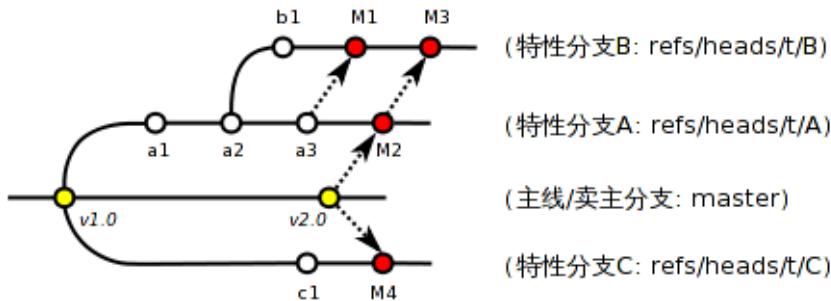


图22-2: Topgit特性分支关系图

在图22-2中，主线上的`v1.0`是上游的版本的一次提交。特性分支A和C都直接依赖主线`master`，而特性分支B则依赖特性分支A。提交`M1`是特定分支B因为特性分支A更新而做的一次迁移。提交`M2`和`M4`，则分别是特性分支A和C因为上游出现了新版本`v2.0`而做的迁移。当然特性分支B也要做相应的迁移，是为`M3`。

上述的描述非常粗糙，因为这样的设计很难实现特性分支导出为补丁文件。例如特性分支B的补丁，实际上应该是`M3`和`M2`之间的差异，而绝不是`M3`到`a2`之间的差异。Topgit为了能够实现分支导出为补丁，又为每个特性的开发引入了一个特殊的引用(`refs/top-bases/*`)，用于追踪分支依赖的“变基”。这些特性分支的基准分支也形成了复杂的分支关系图，如图22-3所示。

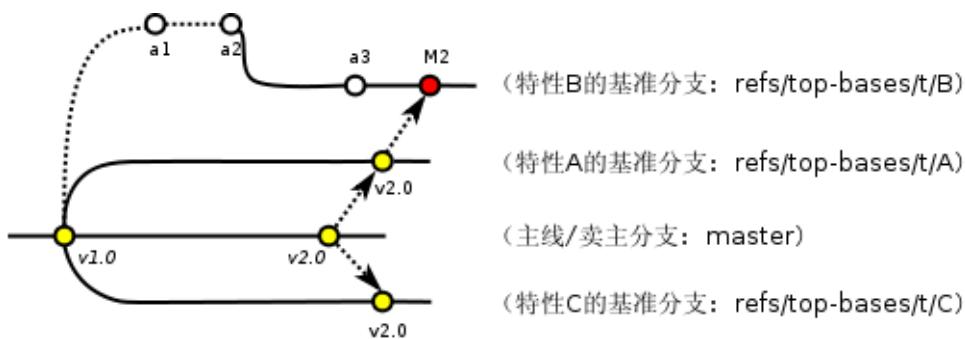


图22-3: Topgit特性分支的基准分支关系图

把图22-2和图22-3两张分支图重合，就可以获得各个特性分支在任一点的特性补丁文件。

上面的特性分支B还只是依赖一个分支，如果出现一个分支依赖多个特性分支的话，情况就会更加的复杂，更会体现出这种设计方案的精妙。

Topgit还在每个特性分支工作区的根目录引入两个文件，用以记录分支的依赖以及关于此分支的说明。

- 文件`:file:`.topdeps``记录该分支所依赖的分支列表。

该文件通过`:command:`tg create``命令在创建特性分支时自动创建，或者通

过:command:`tg depend add`命令来添加新依赖。

- 文件:file:`.topmsg`记录该分支的描述信息。

该文件通过:command:`tg create`命令在创建特性分支时创建，也可以手动编辑。

## Topgit的安装

Topgit的可执行命令只有一个:command:`tg`。其官方参考手册见：<http://repo.or.cz/w/topgit.git?a=blob;f=README>。

安装官方的Topgit版本，直接克隆官方的版本库，执行:command:`make`即可。

```
$ git clone git://repo.or.cz/topgit.git  
$ cd topgit  
$ make  
$ make install
```

缺省会把可执行文件:command:`tg`安装在:file:`\$HOME/bin`（用户主目录下的:file:`bin`目录）下，如果没有将:file:`~/bin`加入环境变量\$PATH中，可能无法执行:command:`tg`。如果具有root权限，也可以将:command:`tg`安装在系统目录中。

```
$ prefix=/usr make  
$ sudo prefix=/usr make install
```

作者对Topgit做了一些增强和改进，在后面的章节予以介绍。如果想安装改进的版本，需要预先安装:command:`quilt`补丁管理工具。然后进行如下操作。

```
$ git clone git://github.com/ossxp-com/topgit.git  
$ cd topgit  
$ QUILT_PATCHES=debian/patches quilt push -a  
$ prefix=/usr make  
$ sudo prefix=/usr make install
```

如果用的是Ubuntu或者Debian Linux操作系统，还可以这么安装。

- 先安装Debian/Ubuntu打包依赖的相关工具软件。

```
$ sudo aptitude install quilt debhelper build-essential fakeroot dpkg-dev
```

- 再调用:command:`dpkg-buildpackage`命令，编译出DEB包，再安装。

```
$ git clone git://github.com/ossxp-com/topgit.git  
$ cd topgit  
$ dpkg-buildpackage -b -rfakeroot  
$ sudo dpkg -i ../topgit_*.deb
```

- 安装完毕后，重新加载命令行补齐，可以更方便的使用`:command:`tg`命令。`

```
$ . /etc/bash_completion
```

## Topgit的使用

通过前面的原理部分，可以发现Topgit为管理特性分支，所引入的配置文件和基准分支都是和Git兼容的。

- 在`refs/top-bases/`命名空间下的引用，用于记录分支的变基历史。
- 在特性分支的工作区根目录引入两个文件`:file:`.topdeps``和`:file:`.topmsg``，用于记录分支依赖和说明。
- 引入新的钩子脚本`:file:`hooks/pre-commit``，用于在提交时检查分支依赖有没有发生循环等。

Topgit的命令行的一般格式为：

```
tg [global_option] <subcmd> [command_options...] [arguments...]
```

- 在子命令前为全局选项，目前可用全局选项只有`-r <remote>`。  
`-r <remote>`可选项，用于设定分支跟踪的远程服务器。默认为`origin`。
- 子命令后可以跟命令相关的可选选项，和参数。

### `:command:`tg help`命令`

`:command:`tg help`命令显示帮助信息。当在:command:`tg help`后面提供子命令名称，可以获得该子命令详细的帮助信息。`

### `:command:`tg create`命令`

`:command:`tg create`命令用于创建新的特性分支。用法：`

```
tg [...] create NAME [DEPS...] -r RNAME
```

其中：

- NAME是新的特性分支的分支名，必须提供。一般约定俗成，NAME以t/前缀开头，以标明此分支是一个Topgit特性分支。
- DEPS...是可选的一个或多个依赖分支名。如果不提供依赖分支名，则使用当前分支作为新的特性分支的依赖分支。
- -r RNAME选项，将远程分支作为依赖分支。不常用。

:command:`tg create`命令会创建新的特性分支refs/heads/NAME，跟踪变基分支refs/top-bases/NAME，并且在项目根目录下创建文件:file:`.topdeps`和:file:`.topmsg`。会提示用户编辑:file:`.topmsg`文件，输入详细的特性分支描述信息。

例如在一个示例版本库，分支master下输入命令：

```
$ tg create t/feature1
tg: Automatically marking dependency on master
tg: Creating t/feature1 base from master...
Switched to a new branch 't/feature1'
tg: Topic branch t/feature1 set up. Please fill .topmsg now and make initial
tg: To abort: git rm -f .top* && git checkout master && tg delete t/feature1
```

提示信息中以“tg:”开头的是Topgit产生的说明。其中提示用户编辑:file:`.topmsg`文件，然后执行一次提交完成Topgit特性分支的创建。

如果想撤销此次操作，删除项目根目录下的:file:`.top\*`文件，切换到master分支，然后执行:command:`tg delete t/feature1`命令删除t/feature1分支以及变基跟踪分支refs/top-bases/t/feature1。

输入:command:`git status`可以看到当前已经切换到t/feature1分支，并且Topgit已经创建了:file:`.topdeps`和:file:`.topmsg`文件，并已将这两个文件加入到暂存区。

```
$ git status
# On branch t/feature1
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .topdeps
#       new file:   .topmsg
#
$ cat .topdeps
master
```

打开`:file:`.topmsg``文件，会看到下面内容（前面增加了行号）：

```
1 From: Jiang Xin <jiangxin@ossp.com>
2 Subject: [PATCH] t/feature1
3
4 <patch description>
5
6 Signed-off-by: Jiang Xin <jiangxin@ossp.com>
```

其中第2行是关于该特性分支的简短描述，第4行是详细描述，可以写多行。

编辑完成，别忘了提交，提交之后才完成Topgit分支的创建。

```
$ git add -u
$ git commit -m "create tg branch t/feature1"
```

### 创建时指定依赖分支

如果这时想创建一个新的特性分支`t/feature2`，并且也是要依赖`master`，注意需要在命令行中提供`master`作为第二个参数，以设定依赖分支。因为当前所处的分支为`t/feature1`，如果不提供指定的依赖分支会自动依赖当前分子。

```
$ tg create t/feature2 master
$ git commit -m "create tg branch t/feature2"
```

下面的命令将创建`t/feature3`分支，该分支依赖`t/feature1`和`t/feature2`。

```
$ tg create t/feature3 t/feature1 t/feature2
$ git commit -m "create tg branch t/feature3"
```

## :command:`tg info` 命令

:command:`tg info` 命令用于显示当前分支或指定的Topgit分支的信息。用法：

```
tg [...] info [NAME]
```

其中`NAME`是可选的Topgit分支名。例如执行下面的命令会显示分支`t/feature3`的信息：

```
$ tg info
```

```
Topic Branch: t/feature3 (1/1 commit)
Subject: [PATCH] t/feature3
Base: 0fa79a5
Depends: t/feature1
          t/feature2
Up-to-date.
```

切换到t/feature1分支，做一些修改，并提交。

```
$ git checkout t/feature1
hack...
$ git commit -m "hacks in t/feature1."
```

然后再来看t/feature3的状态：

```
$ tg info t/feature3
Topic Branch: t/feature3 (1/1 commit)
Subject: [PATCH] t/feature3
Base: 0fa79a5
Depends: t/feature1
          t/feature2
Needs update from:
  t/feature1 (1/1 commit)
```

状态信息显示t/feature3不再是最新的状态(Up-to-date)，因为依赖的分支包含新的提交，而需要从t/feature1获取更新。

## :command:`tg update`命令

:command:`tg update`命令用于更新分支，即从依赖的分支或上游跟踪的分支获取最新的提交合并到当前分支。同时也更新在refs/top-bases/命名空间下的跟踪变基分支。

```
tg [...] update [NAME]
```

其中NAME是可选的Topgit分支名。下面就对需要更新的t/feature3分支执行:command:`tg update`命令。

```
$ git checkout t/feature3
$ tg update
tg: Updating base with t/feature1 changes...
Merge made by recursive.
 feature1 |      1 +
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 feature1
tg: Updating t/feature3 against new base...
Merge made by recursive.
feature1 |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 feature1
```

从上面的输出信息可以看出执行了两次分支合并操作，一次是针对`refs/top-bases/t/feature3`引用指向的跟踪变基分支，另外一次针对的是`refs/heads/t/feature3`特性分支。

执行`:command:`tg update``命令因为要涉及到分支的合并，因此并非每次都会成功。例如在`t/feature3`和`t/feature1`同时对同一个文件（如`feature1`）进行修改。然后在`t/feature3`中再执行`:command:`tg update``可能就会报错，进入冲突解决状态。

```
$ tg update t/feature3
tg: Updating base with t/feature1 changes...
Merge made by recursive.
feature1 |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
tg: Updating t/feature3 against new base...
Auto-merging feature1
CONFLICT (content): Merge conflict in feature1
Automatic merge failed; fix conflicts and then commit the result.
tg: Please commit merge resolution. No need to do anything else
tg: You can abort this operation using `git reset --hard` now
tg: and retry this merge later using `tg update`.
```

可以看出第一次对`refs/top-bases/t/feature3`引用指向的跟踪变基分支成功合并，但在对`t/feature3`特性分支进行合并时出错。

```
$ tg info
Topic Branch: t/feature3 (3/2 commits)
Subject: [PATCH] t/feature3
Base: 37dcb62
* Base is newer than head! Please run `tg update`.
Depends: t/feature1
          t/feature2
Up-to-date.

$ tg summary
      t/feature1                  [PATCH] t/feature1
  0      t/feature2                  [PATCH] t/feature2
>     B t/feature3                  [PATCH] t/feature3
```

```
$ git status
# On branch t/feature3
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:    feature1
#
no changes added to commit (use "git add" and/or "git commit -a")
```

通过`:command:`tg info``命令可以看出当前分支状态是Up-to-date，但是之前有提示：分支的基（Base）要比头（Head）新，请执行`:command:`tg update``命令。这时如果执行`:command:`tg summary``命令的话，可以看到t/feature3处于B（Break）状态。

用`:command:`git status``命令，可以看出因为两个分支同时修改了文件`:file:`feature1``导致冲突。

可以编辑`:file:`feature1``文件，或者调用冲突解决工具解决冲突，之后再提交，才真正完成此次`:command:`tg update``。

```
$ git mergetool
$ git commit -m "resolved conflict with t/feature1."

$ tg info
Topic Branch: t/feature3 (4/2 commits)
Subject: [PATCH] t/feature3
Base: 37dcb62
Depends: t/feature1
          t/feature2
Up-to-date.
```

## `:command:`tg summary` 命令`

`:command:`tg summary``命令用于显示Topgit管理的特性分支的列表及各个分支的状态。用法：

```
tg [...] summary [-t | --sort | --deps | --graphviz]
```

不带任何参数执行`:command:`tg summary``是最常用的Topgit命令。在介绍无参数的`:command:`tg summary``命令之前，先看看其他简单的用法。

使用-t参数只显示特性分支列表。

```
$ tg summary -t
```

```
t/feature1  
t/feature2  
t/feature3
```

使用`--deps`参数会显示Topgit特性分支，及其依赖的分支。

```
$ tg summary --deps  
t/feature1 master  
t/feature2 master  
t/feature3 t/feature1  
t/feature3 t/feature2
```

使用`--sort`参数按照分支依赖的顺序显示分支列表，除了Topgit分支外，依赖的非Topgit分支也会显示：

```
$ tg summary --sort  
t/feature3  
t/feature2  
t/feature1  
master
```

使用`--graphviz`会输出GraphViz格式文件，可以用于显示特性分支之间的关系。

```
$ tg summary --graphviz | dot -T png -o topgit.png
```

生成的特性分支关系图如图22-4所示。

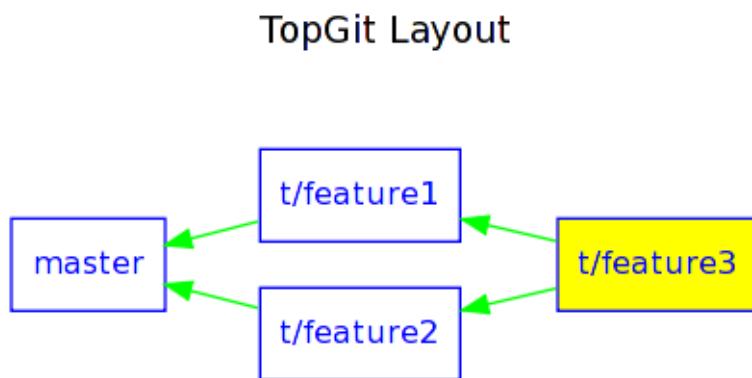


图22-4：Topgit特性分支依赖关系图

不带任何参数执行:`:command:`tg summary``会显示分支列表及状态。这是最常用的Topgit命令之一。

```
$ tg summary
      t/feature1          [PATCH] t/feature1
0      t/feature2          [PATCH] t/feature2
>      t/feature3          [PATCH] t/feature3
```

其中：

- 标记“>”：(t/feature3分支之前的大于号) 用于标记当前所处的特性分支。
- 标记“0”：(t/feature2分支前的数字0) 含义是该分支中没有提交，这一个建立后尚未使用或废弃的分支。
- 标记“D”：表明该分支处于过时(out-of-date)状态。可能是一个或多个依赖的分支包含了新的提交，尚未合并到此特性分支。可以用:`:command:`tg info``命令看出到底是由于哪个依赖分支的改动导致该特性分支处于过时状态。
- 标记“B”：之前演示中出现过，表明该分支处于Break状态，即可能由于冲突未解决或者其他原因导致该特性分支的基(base)相对该分支的头(head)不匹配。`refs/top-bases`下的跟踪变基分支迁移了，但是特性分支未完成迁移。
- 标记“!”：表明该特性分支所依赖的分支不存在。
- 标记“1”：表明该特性分支只存在于本地，不存在于远程跟踪服务器。
- 标记“r”：表明该特性分支既存在于本地，又存在于远程跟踪服务器，并且两者匹配。
- 标记“L”：表明该特性分支，本地的要被远程跟踪服务器要新。
- 标记“R”：表明该特性分支，远程跟踪服务器的要被本地的新。
- 如果没有出现“l/r/L/R”：表明该版本库尚未设置远程跟踪版本库(没有remote)。
- 一般带有标记“r”的是最常见的，也是最正常的。

下面通过:`:command:`tg remote``为测试版本库建立一个对应的远程跟踪版本库，然后就能在:`:command:`tg summary``的输出中看到标识符“l/r”等。

## `:command:`tg remote`命令`

`:command:`tg remote``命令用于为远程跟踪版本库设置Topgit的特性分支的关联，在和该远程版本库进行`fetch`、`pull`等操作时能够同步Topgit相关分支。

```
tg [...] remote [--populate] [REMOTE]
```

其中`REMOTE`为远程跟踪版本库的名称，如“`origin`”，会自动在该远程源的配置中增加`refs/top-bases`下引用的同步。下面的示例中前面用加号标记的行就是当执行:`:command:`tg remote origin``后增加的设置。

```
[remote "origin"]
  url = /path/to/repos/tgtest.git
  fetch = +refs/heads/*:refs/remotes/origin/*
+    fetch = +refs/top-bases/*:refs/remotes/origin/top-bases/*
```

如果使用`--populate`参数，除了会向上面那样设置缺省的Topgit远程版本库外，会自动执行`:command:`git fetch``命令，然后还会为新的Topgit特性分支在本地创建新的分支，以及其对应的跟踪分支。

当执行`:command:`tg``命令时，如果不使用`-r remote`全局参数，默认使用缺省的Topgit远程版本库。

下面为前面测试的版本库设置一个远程的跟踪版本库。

先创建一个裸版本库`tgtest.git`。

```
$ git init --bare /path/to/repos/tgtest.git
Initialized empty Git repository in /path/to/repos/tgtest.git/
```

然后在测试版本库中注册名为`origin`的远程版本库为刚刚创建的版本库。

```
$ git remote add origin /path/to/repos/tgtest.git
```

执行`:command:`git push``，将主线同步到远程的版本库。

```
$ git push origin master
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (7/7), 585 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To /path/to/repos/tgtest.git
 * [new branch]      master -> master
```

之后通过`:command:`tg remote``命令告诉Git这个远程版本库需要跟踪Topgit分支。

```
$ tg remote --populate origin
```

会在当前的版本库的`:file:`.git/config``文件中添加设置（以加号开头的行）：

```
[remote "origin"]
  url = /path/to/repos/tgtest.git
  fetch = +refs/heads/*:refs/remotes/origin/*
+    fetch = +refs/top-bases/*:refs/remotes/origin/top-bases/*
+[topgit]
+    remote = origin
```

这时再执行`:command:`tg summary``会看到分支前面都有标记“1”，即本地提交比远程版本库要新。

```
$ tg summary
1      t/feature1          [PATCH] t/feature1
01     t/feature2          [PATCH] t/feature2
> 1     t/feature3          [PATCH] t/feature3
```

将t/feature2的特性分支推送到远程版本库。

```
$ tg push t/feature2
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 457 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To /path/to/repos/tgtest.git
 * [new branch]      t/feature2 -> t/feature2
 * [new branch]      refs/top-bases/t/feature2 -> refs/top-bases/t/feature2
```

再来看看`:command:`tg summary``的输出，会看到t/feature2的标识变为“r”，即远程和本地相同步。

```
$ tg summary
1      t/feature1          [PATCH] t/feature1
0r     t/feature2          [PATCH] t/feature2
> 1     t/feature3          [PATCH] t/feature3
```

使用`:command:`tg push --all``(改进过的Topgit)，会将所有的topgit分支推送到远程版本库。之后再来看`:command:`tg summary``的输出。

```
$ tg summary
r      t/feature1          [PATCH] t/feature1
0r     t/feature2          [PATCH] t/feature2
> r     t/feature3          [PATCH] t/feature3
```

如果版本库设置了多个远程版本库，要针对每一个远程版本库执行`:command:`tg remote <REMOTE>``，但只能有一个远程的源用`--populate`参数调用`:command:`tg remote``将其设置为缺省的远程版本库。

### `:command:`tg push` 命令`

在前面`:command:`tg remote``的介绍中，已经看到了`:command:`tg push``命令。`:command:`tg push``命令用于将Topgit特性分支及对应的变基跟踪分支推送到远程版本库。用法：

```
tg [...] push [--dry-run] [--no-deps] [--tgish-only] [--all|branch*]
```

`:command:`tg push``命令后面的参数指定要推送给远程服务器的分支列表，如果省略则推送当前分支。改进的`:command:`tg push``可以不提供任何分支，只提供`--all`参数就可以将所有Topgit特性分支推送到远程版本库。

参数`--dry-run`是测试执行效果，不真正执行。参数`--no-deps`的含义是不推送依赖的分支，缺省推送。参数`--tgish-only`的含义是只推送Topgit特性分支，缺省指定的所有分支都进行推送。

### `:command:`tg depend` 命令`

`:command:`tg depend``命令目前仅实现了为当前的Topgit特性分支增加新的依赖。用法：

```
tg [...] depend add NAME
```

会将`NAME`加入到文件`:file:`.topdeps``中，并将`NAME`分支向该特性分支以及变基跟踪分支进行合并操作。虽然Topgit可以检查到分支的循环依赖，但还是要注意合理的设置分支的依赖，合并重复的依赖。

### `:command:`tg base` 命令`

`:command:`tg base``命令用于显示特性分支的基（base）当前的commit-id。

### `:command:`tg delete` 命令`

`:command:`tg delete``命令用于删除Topgit特性分支以及其对应的变基跟踪分支。用法：

```
tg [...] delete [-f] NAME
```

缺省只删除没有改动的分支，即标记为“0”的分支，除非使用-f参数。

目前此命令尚不能自动清除其分支中对删除分支的依赖，还需要手工调整:`:file:`.topdeps``文件，删除不存在分支的依赖。

## :command:`tg patch` 命令

:command:`tg patch` 命令通过比较特性分支及其变基跟踪分支的差异，显示该特性分支的补丁。用法：

```
tg [...] patch [-i | -w] [NAME]
```

其中参数-i显示暂存区和变基跟踪分支的差异。参数-w显示工作区和变基跟踪分支的差异。

:command:`tg patch` 命令存在的一个问题是在工作区的根执行才能够正确显示。这个缺陷已经在我改进的Topgit中被改正。

## :command:`tg export` 命令

:command:`tg export` 命令用于导出特性分支及其依赖，便于向上游贡献。可以导出Quilt格式的补丁列表，或者顺序提交到另外的分支中。用法：

```
tg [...] export ([--collapse] NEWBRANCH | [--all | -b BRANCH1,BRANCH2...]) --q
```

这个命令有三种导出方法。

- 将所有的Topgit特性分支压缩为一个提交到新的分支。

```
tg [...] export --collapse NEWBRAQNCH
```

- 将所有的Topgit特性分支按照线性顺序提交到一个新的分支中。

```
tg [...] export --linearize NEWBRANCH
```

- 将指定的Topgit分支（一个或多个）及其依赖分支转换为Quilt格式的补丁，保存到指定目录中。

```
tg [...] export -b BRANCH1,BRANCH2... --quilt DIRECTORY
```

在导出为Quilt格式补丁的时候，如果想将所有的分支导出，必须用**-b**参数将分支全部罗列（或者分支的依赖关系将所有分支囊括），这对于需要导出所有分支非常不方便。我改进的Topgit通过**--all**参数，实现导出所有分支。

## :command:`tg import` 命令

:command:`tg import` 命令将分支的提交转换为Topgit特性分支，每个分支称为一个特性分支，各个特性分支线性依赖。用法：

```
tg [...] import [-d BASE_BRANCH] {[ -p PREFIX] RANGE... | -s NAME COMMIT}
```

如果不使用**-d**参数，特性分支以当前分支为依赖。特性分支名称自动生成，使用约定俗成的*t/*作为前缀，也可以通过**-p**参数指定其他前缀。可以通过**-s**参数设定特性分支的名称。

## :command:`tg log` 命令

:command:`tg log` 命令显示特性分支的提交历史，并忽略合并引入的提交。

```
tg [...] log [NAME] [-- GIT LOG OPTIONS...]
```

:command:`tg log` 命令实际是对:command:`git log` 命令的封装。这个命令通过**--no-merges**和**--first-parent**参数调用:command:`git log`，虽然屏蔽了大量因和依赖分支合并而引入的依赖分支的提交日志，但是同时也屏蔽了合并到该特性分支的其他贡献者的提交。

## :command:`tg mail` 命令

:command:`tg mail` 命令将当前分支或指定特性分支的补丁以邮件型式外发。用法：

```
tg [...] mail [-s SEND_EMAIL_ARGS] [-r REFERENCE_MSGID] [NAME]
```

:command:`tg mail` 调用:command:`git send-email` 发送邮件，参数**-s**用于向该命令传递参数（需要用双引号括起来）。邮件中的目的地址从patch文件头中的*To*、*Cc*和*Bcc*等字段获取。参数**-r**引用回复邮件的id以便正确生成*in-reply-to*邮件头。

注意：此命令可能会发送多封邮件，可以通过如下设置在调用:command:`git send-email` 命令发送邮件时进行确认。

```
git config sendemail.confirm always
```

## :command:`tg graph` 命令

:command:`tg graph` 命令并非官方提供的命令，而是源自一个补丁，实现文本方式的 Topgit 分支图。当然这个文本分支图没有 :command:`tg summary --graphviz` 生成的那么漂亮。

## Topgit hacks

在 Topgit 的使用中陆续发现一些不合用的地方，于是便使用 Topgit 特性分支的方式来改进 Topgit 自身的代码。在群英汇博客上，介绍了这几个改进，参见：<http://blog.ossexp.com/tag/topgit/>。

下面就以此为例，介绍如何参与一个 Topgit 管理下的项目的开发。改进的 Topgit 版本库地址为：git://github.com/ossexp-com/topgit.git。

首先克隆该版本库。

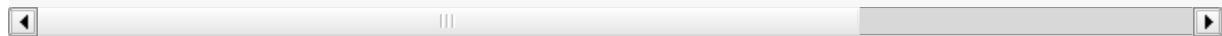
```
$ git clone git://github.com/ossexp-com/topgit.git  
$ cd topgit
```

查看远程分支。

```
$ git branch -r  
origin/HEAD -> origin/master  
origin/master  
origin/t.debian_locations  
origin/t/export_quilt_all  
origin/t/fast_tg_summary  
origin/t/graphviz_layout  
origin/t/tg_completion_bugfix  
origin/t/tg_graph_ascii_output  
origin/t/tg_patch_cdup  
origin/t/tg_push_all  
origin/tgmaster
```

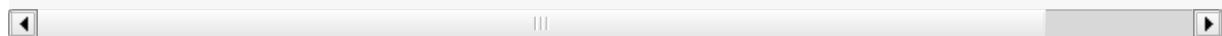
看到远程分支中出现了熟悉的以 t/ 为前缀的 Topgit 分支，说明这个版本库是一个 Topgit 管理的定制开发版本库。那么为了能够获取 Topgit 的变基跟踪分支，需要用 :command:`tg remote` 命令对缺省的 origin 远程版本库注册一下。

```
$ tg remote --populate origin
tg: Remote origin can now follow TopGit topic branches.
tg: Populating local topic branches from remote 'origin'...
From git://github.com/ossxp-com/topgit
* [new branch]      refs/top-bases/t/debian_locations -> origin/top-bases/t/
* [new branch]      refs/top-bases/t/export_quilt_all -> origin/top-bases/t/
* [new branch]      refs/top-bases/t/fast_tg_summary -> origin/top-bases/t/f
* [new branch]      refs/top-bases/t/graphviz_layout -> origin/top-bases/t/g
* [new branch]      refs/top-bases/t/tg_completion_bugfix -> origin/top-base
* [new branch]      refs/top-bases/t/tg_graph_ascii_output -> origin/top-bas
* [new branch]      refs/top-bases/t/tg_patch_cdup -> origin/top-bases/t/tg_
* [new branch]      refs/top-bases/t/tg_push_all -> origin/top-bases/t/tg_pu
tg: Adding branch t/debian_locations...
tg: Adding branch t/export_quilt_all...
tg: Adding branch t/fast_tg_summary...
tg: Adding branch t/graphviz_layout...
tg: Adding branch t/tg_completion_bugfix...
tg: Adding branch t/tg_graph_ascii_output...
tg: Adding branch t/tg_patch_cdup...
tg: Adding branch t/tg_push_all...
tg: The remote 'origin' is now the default source of topic branches.
```



执行:[command: `tg summary`](#)看一下本地Topgit特性分支状态。

```
$ tg summary
r ! t/debian_locations          [PATCH] make file locations Debian-co
r ! t/export_quilt_all           [PATCH] t/export_quilt_all
r ! t/fast_tg_summary            [PATCH] t/fast_tg_summary
r ! t/graphviz_layout             [PATCH] t/graphviz_layout
r ! t/tg_completion_bugfix       [PATCH] t/tg_completion_bugfix
r   t/tg_graph_ascii_output        [PATCH] t/tg_graph_ascii_output
r ! t/tg_patch_cdup               [PATCH] t/tg_patch_cdup
r ! t/tg_push_all                 [PATCH] t/tg_push_all
```



怎么？出现了感叹号？记得前面在介绍:[command: `tg summary`](#)命令的章节中提到过，感叹号的出现说明该特性分支依赖的分支丢失。用:[command: `tg info`](#)查看一下某个特性分支。

```
$ tg info t/export_quilt_all
Topic Branch: t/export_quilt_all (6/4 commits)
Subject: [PATCH] t/export_quilt_all
Base: 8b0f1f9
Remote Mate: origin/t/export_quilt_all
Depends: tgmaster
MISSING: tgmaster
```



```
Up-to-date.
```

原来该特性分支依赖tgmaster分支，而不是master分支。远程存在tgmaster分支而本地尚不存在。于是在本地建立tgmaster跟踪分支。

```
$ git checkout tgmaster
Branch tgmaster set up to track remote branch tgmaster from origin.
Switched to a new branch 'tgmaster'
```

这回:command:`tg summary` 的输出正常了。

```
$ tg summary
r      t.debian_locations          [PATCH] make file locations Debian-co
r      t/export_quilt_all           [PATCH] t/export_quilt_all
r      t/fast_tg_summary           [PATCH] t/fast_tg_summary
r      t/graphviz_layout           [PATCH] t/graphviz_layout
r      t/tg_completion_bugfix     [PATCH] t/tg_completion_bugfix
r      t/tg_graph_ascii_output     [PATCH] t/tg_graph_ascii_output
r      t/tg_patch_cdup             [PATCH] t/tg_patch_cdup
r      t/tg_push_all               [PATCH] t/tg_push_all
```

通过下面命令创建图形化的分支图。

```
$ tg summary --graphviz | dot -T png -o topgit.png
```

生成的特性分支关系图如图22-5所示。

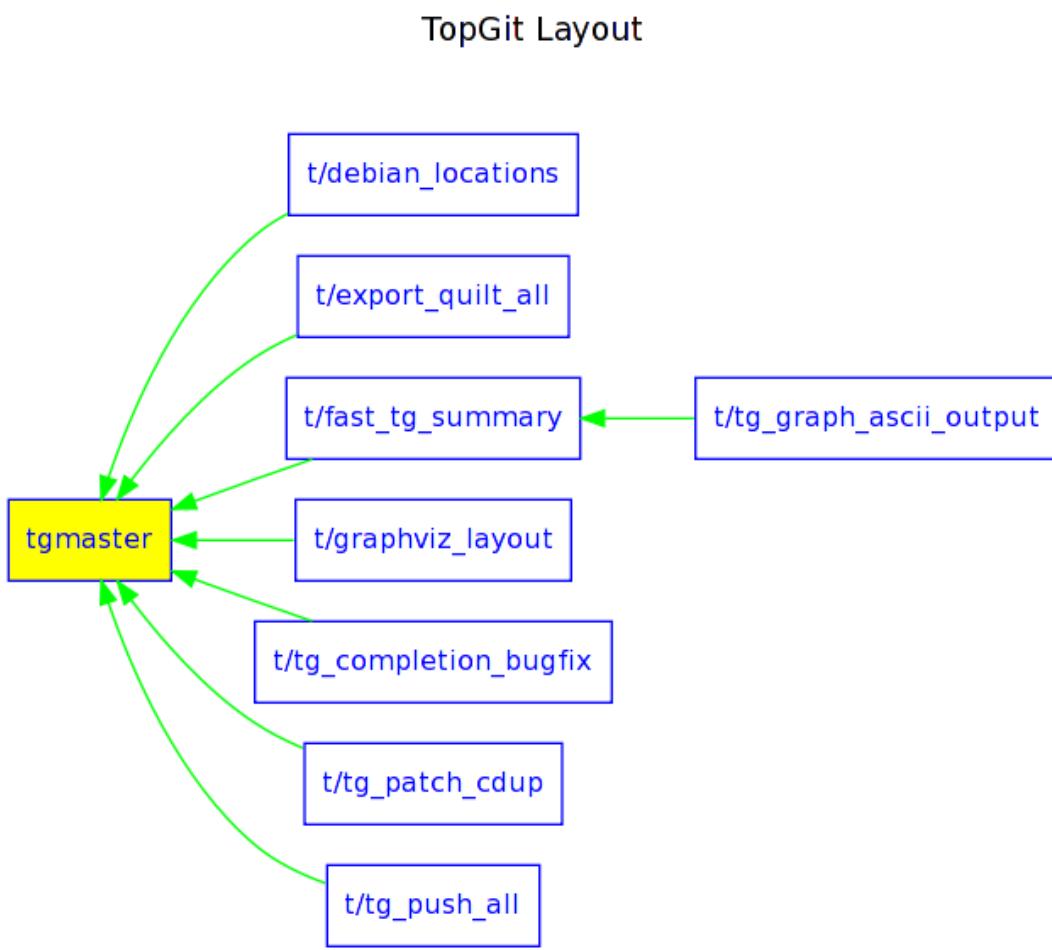


图22-5：Topgit改进项目的特性分支依赖关系图

其中：

- 特性分支`t/export_quilt_all`, 为`:command:`tg export --quilt``命令增加`--all`选项, 以便导出所有特性分支。
- 特性分支`t/fast_tg_summary`, 主要是改进`tg`命令补齐时分支的显示速度, 当特性分支接近上百个时差异非常明显。
- 特性分支`t/graphviz_layout`, 改进了分支的图形输出格式。
- 特性分支`t/tg_completion_bugfix`, 解决了命令补齐的一个 Bug。
- 特性分支`t/tg_graph_ascii_output`, 源自Bert Wesarg的贡献, 非常巧妙地实现了文本化的分支图显示, 展示了gvpr命令的强大功能。
- 特性分支`t/tg_patch_cdup`, 解决了在项目的子目录下无法执行`:command:`tg patch``的问题。
- 特性分支`t/tg_push_all`, 通过为`:command:`tg push``增加`--all`选项, 解决了当`tg`从0.7升级到0.8后, 无法批量向上游推送特性分支的问题。

下面展示一下如何跟踪上游的最新改动，并迁移到新的上游版本。分支tgmaster用于跟踪上游的Topgit分支，以t/开头的分支是对Topgit改进的特性分支，而master分支实际上是导出Topgit补丁文件并负责编译特定Linux平台发行包的分支。

把官方的Topgit分支以upstream的名称加入为新的远程版本库。

```
$ git remote add upstream git://repo.or.cz/topgit.git
```

然后将upstream远程版本的master分支合并到本地的tgmaster分支。

```
$ git pull upstream master:tgmaster
From git://repo.or.cz/topgit
 29ab4cf..8b0f1f9  master      -> tgmaster
```

此时再执行:command:`tg summary`会发现所有的Topgit分支都多了一个标记“D”，表明因为依赖分支的更新导致Topgit特性分支过时了。

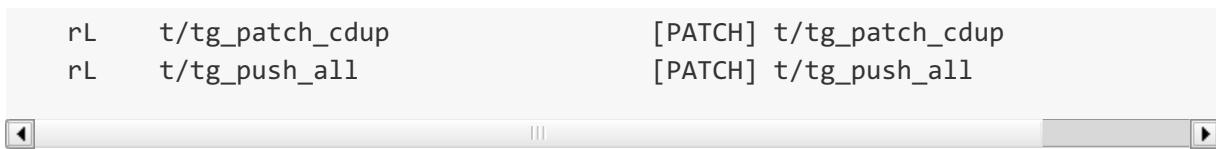
```
$ tg summary
r D  t.debian_locations          [PATCH] make file locations Debian-co
r D  t/export_quilt_all           [PATCH] t/export_quilt_all
r D  t/fast_tg_summary            [PATCH] t/fast_tg_summary
r D  t/graphviz_layout             [PATCH] t/graphviz_layout
r D  t/tg_completion_bugfix       [PATCH] t/tg_completion_bugfix
r D  t/tg_graph_ascii_output       [PATCH] t/tg_graph_ascii_output
r D  t/tg_patch_cdup               [PATCH] t/tg_patch_cdup
r D  t/tg_push_all                 [PATCH] t/tg_push_all
```

依次对各个分支执行:command:`tg update`，完成对更新的依赖分支的合并。

```
$ tg update t/export_quilt_all
...
```

对各个分支完成更新后，会发现:command:`tg summary`的输出中，标识过时的“D”标记变为“L”，即本地比远程服务器分支要新。

```
$ tg summary
rL  t.debian_locations          [PATCH] make file locations Debian-co
rL  t/export_quilt_all           [PATCH] t/export_quilt_all
rL  t/fast_tg_summary            [PATCH] t/fast_tg_summary
rL  t/graphviz_layout             [PATCH] t/graphviz_layout
rL  t/tg_completion_bugfix       [PATCH] t/tg_completion_bugfix
rL  t/tg_graph_ascii_output       [PATCH] t/tg_graph_ascii_output
```



执行:command:`tg push --all`就可以实现将所有Topgit特性分支推送到远程服务器上。当然需要具有提交权限才可以。

## Topgit使用中的注意事项

经常运行:command:`tg remote --populate`获取他人创建的特性分支

运行命令:command:`git fetch origin`和远程版本库(origin)同步，只能将他人创建的Topgit特性分支在本地以refs/remotes/origin/t/<branch-name>的名称保存，而不能自动在本地建立分支。

当版本库是使用Topgit维护的话，应该在和远程版本库同步的时候使用执行:command:`tg remote --populate origin`。这条命令会做两件事情：

- 自动调用:command:`git fetch origin`获取远程origin版本库的新的提交和引用。
- 检查:file:`refs/remotes/origin/top-bases/`下的所有引用，如果是新的、在本地(refs/top-bases/)尚不存在，说明有其他人创建了新的特性分支。Topgit会据此自动的在本地创建新的特性分支。

### 适时的调整特性分支的依赖关系

例如前面示例的Topgit库的依赖关系，各个分支可能的依赖文件内容如下。

- 分支t/feature1的:file:`.topdeps`文件

master

- 分支t/feature2的:file:`.topdeps`文件

master

- 分支t/feature3的:file:`.topdeps`文件

t/feature1  
t/feature2

如果分支t/feature3的:file:`.topdeps`文件是这样的，可能就会存在问题。

```
master  
t/feature1  
t/feature2
```

问题出在t/feature3依赖的其他分支已经依赖了master分支。虽然不会造成致命的影响，但是在特定情况下这种重复会造成不便。例如在master分支更新后，可能由于代码重构的比较厉害，在特性分支迁移时会造成冲突，如在t/feature1分支执行:command:`tg update`会遇到冲突，当辛苦完成冲突解决并提交后，在t/feature3执行:command:`tg update`时因为先依赖的是master分支，会先在master分支上对t/feature3分支进行变基，肯定会遇到和t/feature1相同的冲突，还要再重复地解决一次。

如果在:file:`.topdeps`文件中将对master分支的重复的依赖删除，就不会出现上面的重复进行冲突解决的问题了。

同样的道理，如果t/feature3的:file:`.topdeps`文件写成这样，效果也将不同：

```
t/feature2  
t/feature1
```

依赖的顺序不同会造成变基的顺序也不同，同样也会产生重复的冲突解决。因此当发现重复的冲突时，可以取消变基操作，修改特性分支的:file:`.topdeps`文件，调整文件内容（删除重复分支，调整分支顺序）并提交，然后在执行:command:`tg update`继续变基操作。

## Topgit特性分支的里程碑和分支管理

Topgit本身就是对特性分支进行管理的软件。Topgit的某个时刻的开发状态是所有Topgit管理下的分支（包括跟踪分支）整体的状态。如果需要对Topgit所有相关的分支进行跟踪管理该如何实现呢？

例如master主线由于提交了上游的新版本而改动，在对各个Topgit特性分支执行:command:`tg update`时，搞的一团糟，而又不小心执行了:command:`tg push --all`，这下无论本地和远程都处于混乱的状态。

使用里程碑(tags)来管理是不可能的，因为tag只能针对一个分支做标记而不能标记所有的分支。

使用克隆是唯一的方法。即针对不同的上游建立不同的Git库，通过不同的克隆实现针对不同上游版本特性分支开发的管理。例如一旦上游出现新版本，就从当前版本库建立一个克隆，或者用于保存当前上游版本的特性开发状态，或者用于新的上游版本的特性开发。

也许还可以通过其他方法实现，例如将Topgit所有相关分支都复制到一个特定的引用目录

中，如:`:file:`refs/top-tags/v1.0/``用于实现特性分支的里程碑记录。

来源：<https://github.com/gotgit/gotgit/blob/master/04-git-model/030-topgit-model.rst>

## 子模组协同模型

项目的版本库某些情况下需要引用其他版本库中的文件，例如公司积累了一套常用的函数库，被多个项目调用，显然这个函数库的代码不能直接放到某个项目的代码中，而是要独立为一个代码库，那么其他项目要调用公共的函数库，该如何处理呢？分别把公共函数库的文件拷贝到各自的项目中，会造成冗余，丢弃了公共函数库的维护历史，显然不是好的方法。本节要讨论的子模组协同模型，就是解决这个问题的一个方案。

熟悉Subversion的用户马上会想起`svn:externals`属性，可以实现对外部代码库的引用。

Git的子模组（submodule）是类似的一种实现。不过因为Git的特殊性，二者的区别还是满大的。参见表23-1。

表23-1： SVN和Git相似功能对照表

	<code>svn:externals</code>	<code>git submodule</code>
如何记录外部版本库地址？	目录的 <code>svn:externals</code> 属性	项目根目录下的 <code>:file:`.gitmodules`</code> 文件
缺省是否自动检出外部版本库？	是。在使用 <code>:command:` svn checkout`</code> 检出时若使用参数 <code>--ignore-externals</code> 可以忽略对外部版本库引用不检出。	否。缺省不克隆外部版本库。克隆要用 <code>:command:` git submodule init`</code> 、 <code>:command:` git submodule update`</code> 命令。
是否能部分引用外部版本库内容？	是。因为SVN支持部分检出。	否。必须克隆整个外部版本库。
是否可以指向分支而随之改变？	是。	否。固定于外部版本库某个提交。

## 创建子模组

在演示子模组的创建和使用之前，先作些准备工作。先尝试建立两个公共函数库（`libA.git`和`libB.git`）以及一个引用函数库的主版本库（`super.git`）。

```
$ git --git-dir=/path/to/repos/libA.git init --bare
$ git --git-dir=/path/to/repos/libB.git init --bare
$ git --git-dir=/path/to/repos/super.git init --bare
```

向两个公共的函数库中填充些数据。这就需要在工作区克隆两个函数库，提交数据，并推送。

- 克隆`libA.git`版本库，添加一些数据，然后提交并推送。

说明：示例中显示为`hack...`的地方，做一些改动（如创建新文件等），并将改动添加到暂存区。

```
$ git clone /path/to/repos/libA.git /path/to/my/workspace/libA
$ cd /path/to/my/workspace/libA
hack ...
$ git commit -m "add data for libA"
$ git push origin master
```

- 克隆libB.git版本库，添加一些数据，然后提交并推送。

```
$ git clone /path/to/repos/libB.git /path/to/my/workspace/libB  
$ cd /path/to/my/workspace/libB  
hack ...  
$ git commit -m "add data for libB"  
$ git push origin master
```

版本库super是准备在其中创建子模组的。super版本库刚刚初始化还未包含提交，master分支尚未有正确的引用。需要在super版本中至少创建一个提交。下面就克隆super版本库，在其中完成一个提交（空提交即可），并推送。

```
$ git clone /path/to/repos/super.git /path/to/my/workspace/super  
$ cd /path/to/my/workspace/super  
$ git commit --allow-empty -m "initialized."  
$ git push origin master
```

现在就可以在super版本库中使用:command:`git submodule add`命令添加子模组了。

```
$ git submodule add /path/to/repos/libA.git lib/lib_a  
$ git submodule add /path/to/repos/libB.git lib/lib_b
```

至此看一下super版本库工作区的目录结构。在根目录下多了一个:file:`.gitmodules`文件，并且两个函数库分别克隆到:file:`lib/lib\_a`目录和:file:`lib/lib\_b`目录下。

```
$ ls -aF  
./ ../.git/ .gitmodules lib/
```

看看:file:`.gitmodules`的内容：

```
$ cat .gitmodules  
[submodule "lib/lib_a"]  
  path = lib/lib_a  
  url = /path/to/repos/libA.git  
[submodule "lib/lib_b"]  
  path = lib/lib_b  
  url = /path/to/repos/libB.git
```

此时super的工作区尚未提交。

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       new file:   .gitmodules  
#       new file:   lib/lib_a  
#       new file:   lib/lib_b  
#
```

完成提交之后，子模组才算正式在super版本库中创立。运行:command:`git push`把建立了新模组的本地库推送到远程的版本库。

```
$ git commit -m "add modules in lib/lib_a and lib/lib_b."
```

```
$ git push
```

在提交过程中，发现作为子模组方式添加的版本库实际上并没有添加版本库的内容。实际上只是以gitlink方式[\[1\]](#)添加了一个链接。至于子模组的实际地址，是由文件`:file:`.gitmodules``中指定的。

可以通过查看补丁的方式，看到lib/lib\_a和lib/lib\_b子模组的存在方式（即gitlink）。

```
$ git show HEAD

commit 19bb54239dd7c11151e0dcb8b9389e146f055ba9
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Fri Oct 29 10:16:59 2010 +0800

    add modules in lib/lib_a and lib/lib_b.

diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 000000..60c7d1f
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,6 @@
+[submodule "lib/lib_a"]
+    path = lib/lib_a
+    url = /path/to/repos/libA.git
+[submodule "lib/lib_b"]
+    path = lib/lib_b
+    url = /path/to/repos/libB.git
diff --git a/lib/lib_a b/lib/lib_a
new file mode 160000
index 000000..126b181
--- /dev/null
+++ b/lib/lib_a
@@ -0,0 +1 @@
+Subproject commit 126b18153583d9bee4562f9af6b9706d2e104016
diff --git a/lib/lib_b b/lib/lib_b
new file mode 160000
index 000000..3b52a71
--- /dev/null
+++ b/lib/lib_b
@@ -0,0 +1 @@
+Subproject commit 3b52a710068edc070e3a386a6efcbdf28bf1bed5
```

## 克隆带子模组的版本库

之前的表23-1在对比Subversion的`svn:externals`子模组实现差异时，提到过克隆带子模组的Git库，并不能自动将子模组的版本库克隆出来。对于只关心项目本身数据，对项目引用的外部项目数据并不关心的用户，这个功能非常好，数据也没有冗余而且克隆的速度也更快。

下面在另外的位置克隆super版本库，会发现lib/lib\_a和lib/lib\_b并未克隆。

```
$ git clone /path/to/repos/super.git /path/to/my/workspace/super-clone
$ cd /path/to/my/workspace/super-clone
$ ls -aF
./ ../.git/.gitmodules lib/
$ find lib
lib
lib/lib_a
lib/lib_b
```

这时如果运行:command:`git submodule status`可以查看到子模组状态。

```
$ git submodule status  
-126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a  
-3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b
```

看到每个子模组的目录前面是40位的提交ID，在最前面是一个减号。减号的含义是该子模组尚未检出。

如果需要克隆出子模组型式引用的外部库，首先需要先执行:command:`git submodule init`。

```
$ git submodule init  
Submodule 'lib/lib_a' (/path/to/repos/libA.git) registered for path 'lib/lib_a'  
Submodule 'lib/lib_b' (/path/to/repos/libB.git) registered for path 'lib/lib_b'
```

执行:command:`git submodule init`实际上修改了:file:`.git/config`文件，对子模组进行了注册。文件:file:`.git/config`的修改示例如下（以加号开始的行代表新增的行）。

```
[core]  
repositoryformatversion = 0  
filemode = true  
bare = false  
logallrefupdates = true  
[remote "origin"]  
fetch = +refs/heads/*:refs/remotes/origin/*  
url = /path/to/repos/super.git  
[branch "master"]  
remote = origin  
merge = refs/heads/master  
+[submodule "lib/lib_a"]  
+ url = /path/to/repos/libA.git  
+[submodule "lib/lib_b"]  
+ url = /path/to/repos/libB.git
```

然后执行:command:`git submodule update`才完成子模组版本库的克隆。

```
$ git submodule update  
Initialized empty Git repository in /path/to/my/workspace/super-clone/lib/lib_a  
Submodule path 'lib/lib_a': checked out '126b18153583d9bee4562f9af6b9706d2e104016  
Initialized empty Git repository in /path/to/my/workspace/super-clone/lib/lib_b  
Submodule path 'lib/lib_b': checked out '3b52a710068edc070e3a386a6efcbdf28bf1bed5'
```

## 在子模组中修改和子模组的更新

执行:command:`git submodule update`更新出来的子模组，都以某个具体的提交版本进行检出。进入某个子模组目录，会发现其处于非跟踪状态（分离头指针状态）。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a  
$ git branch  
* (no branch)  
  master
```

显然这种情况下，如果修改lib/lib\_a下的文件，提交会丢失。下面介绍一下如何在检出的子模组中修改，以及更新子模组。

在子模组中切换到master分支（或其他想要修改的分支）后，再进行修改。

- 切换到master分支，然后在工作区做一些改动。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a  
$ git checkout master  
hack ...
```

- 执行提交。

```
$ git commit
```

- 查看状态，会看到相对于远程分支领先一个提交。

```
$ git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 1 commit.  
#  
nothing to commit (working directory clean)
```

在`:command:`git status``的状态输出中，可以看出新提交尚未推送到远程版本库。现在暂时不推送，看看在super版本库中执行`:command:`git submodule update``对子模组的影响。

- 先到super-clone版本库查看一下状态，可以看到子模组已修改，包含更新的提交。

```
$ cd /path/to/my/workspace/super-clone/  
$ git status  
# On branch master  
# Changed but not updated:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working direct  
#  
#       modified: lib/lib_a (new commits)  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

- 通过`:command:`git submodule status``命令可以看出lib/lib\_a子模组指向了新的提交ID（前面有一个加号），而lib/lib\_b模组状态正常（提交ID前是一个空格，不是加号也不是减号）。

```
$ git submodule status  
+5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9 lib/lib_a (heads/master)  
3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

- 这时如果不小心执行了一次`:command:`git submodule update``命令，会将lib/lib\_a重新切换到旧的指向。

```
$ git submodule update  
Submodule path 'lib/lib_a': checked out '126b18153583d9bee4562f9af6b9706d
```

- 执行`:command:`git submodule status``命令查看子模组状态，看到lib/lib\_a子模组被重置了。

```
$ git submodule status
126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

那么刚才在lib/lib\_a中的提交丢失了么？实际上因为已经提交到了master主线，因此提交没有丢失，但是如果有数据没有提交，就会造成未提交数据的丢失。

- 进到:file:`lib/lib\_a`目录，看到工作区再一次进入分离头指针状态。

```
$ cd lib/lib_a
$ git branch
* (no branch)
  master
```

- 重新检出master分支找回之前的提交。

```
$ git checkout master
Previous HEAD position was 126b181... add data for libA
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```

现在如果要将:file:`lib/lib\_a`目录下子模组的改动记录到父项目(super版本库)中，就需要在父项目中进行一次提交才能实现。

- 进入父项目根目录，查看状态。因为lib/lib\_a的提交已经恢复，因此再次显示为有改动。

```
$ cd /path/to/my/workspace/super-clone/
$ git status -s
 M lib/lib_a
```

- 查看差异比较，会看到指向子模组的gitlink有改动。

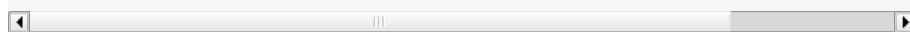
```
$ git diff
diff --git a/lib/lib_a b/lib/lib_a
index 126b181..5dea269 160000
--- a/lib/lib_a
+++ b/lib/lib_a
@@ -1 +1 @@
-Subproject commit 126b18153583d9bee4562f9af6b9706d2e104016
+Subproject commit 5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9
```

- 将gitlink的改动添加到暂存区，然后提交。

```
$ git add -u
$ git commit -m "submodule lib/lib_a upgrade to new version."
```

此时先不要忙着推送，因为如果此时执行:command:`git push`将super版本库推送到远程版本库，会引发一个问题。即推送后的远程super版本库的子模组lib/lib\_a指向了一个新的提交，而该提交还在本地的lib/lib\_a版本库(尚未向上游推送)，这会导致其他人克隆super版本库、更新模组时因为找不到该子模组版本库相应的提交而导致出错。下面就是这类错误的错误信息：

```
fatal: reference is not a tree: 5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9
Unable to checkout '5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9' in submodule pa
```



为了避免这种可能性的发生，最好先对lib/lib\_a中的新提交进行推送，然后再对super的子模组更新的提交进行推送。即：

- 先推送子模组。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a  
$ git push
```

- 再推送父版本库。

```
$ cd /path/to/my/workspace/super-clone/  
$ git push
```

## 隐性子模组

我在开发备份工具Gistore时遇到一个棘手的问题就是隐性子模组的问题。Gistore备份工具的原理是将要备份的目录都挂载（mount）在工作区中，然后执行`:command:`git add``。但是如果某个目录已经被Git化了，就只会以子模组方式将该目录添加进来，而不会添加该目录下的文件。对于一个备份工具来说，意味着备份没有成功。

例如当前super版本库下有两个子模组：

```
$ cd /path/to/my/workspace/super-clone/  
$ git submodule status  
126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)  
3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

然后创建一个新目录others，并把该目录用git初始化并做一次空的提交。

```
$ mkdir others  
$ cd others  
$ git init  
$ git commit --allow-empty -m initial  
[master (root-commit) 90364e1] initial
```

在others目录下创建一个文件`:file:`newfile``。

```
$ date > newfile
```

回到上一级目录，执行`:command:`git status``，看到有一个others目录没有加入版本库控制，这很自然。

```
$ cd ..  
$ git status  
# On branch master  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       others/  
nothing added to commit but untracked files present (use "git add" to track)
```

但是如果对others目录执行`:command:`git add``后，会发现奇怪的状态。

```
$ git add others
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   others
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   (commit or discard the untracked or modified content in submodules)
#
#       modified:   others (untracked content)
#
```

看看others目录的添加方式，就会发现others目录以gitlink方式添加到版本库中，而没有把该目录下的文件添加到版本库。

```
$ git diff --cached
diff --git a/others b/others
new file mode 160000
index 000000..90364e1
--- /dev/null
+++ b/others
@@ -0,0 +1 @@
+Subproject commit 90364e1331abc29cc63e994b4d2cfbf7c485e4ad
```

之所以:command:`git status`的显示中others出现两次，就是因为目录:file:`others`被当做子模组添加到父版本库中。因为others版本库本身“不干净”，存在尚未加入版本控制的文件，所以又在状态输出中显示子模组包含改动的提示信息。

执行提交，将others目录提交到版本库中。

```
$ git commit -m "add others as submodule."
```

执行:command:`git submodule status`命令，会报错。因为others作为子模组，没有在:file:`.gitmodules`文件中注册。

```
$ git submodule status
126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
No submodule mapping found in .gitmodules for path 'others'
```

那么如何在不破坏others版本库的前提下，把others目录下的文件加入版本库呢？即避免others以子模组形式添加入库。

- 先删除以gitlink形式入库的others子模组。

```
$ git rm --cached others
rm 'others'
```

- 查看当前状态。

```
$ git status
# On branch master
```

```
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       deleted:    others  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       others/
```

- 重新添加others目录，注意目录后面的斜线（即路径分隔符）非常重要。

```
$ git add others/
```

- 再次查看状态，看到others下的文件被添加到super版本库中。

```
$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       deleted:    others  
#       new file:   others/newfile  
#
```

- 执行提交。

```
$ git commit -m "add contents in others/."  
[master 1e0c418] add contents in others/.  
2 files changed, 1 insertions(+), 1 deletions(-)  
delete mode 160000 others  
create mode 100644 others/newfile
```

在上面的操作过程中，首先先删除了在库中的others子模组（使用--cached参数执行删除）；然后为了添加others目录下的文件，使用了others/（注意others后面的路径分割符“/”）。现在查看一下子模组的状态，会看到只有之前的两个子模组显示出来。

```
$ git submodule status  
126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)  
3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

## 子模组的管理问题

子模组最主要的一个问题是不能基于外部版本库的某一个分支进行创建，使得更新后，子模组处于非跟踪状态，不便于在子模组中进行对外部版本库进行改动。尤其对于授权或其他原因将一个版本库拆分为子模组后，使用和管理非常不方便。在第25章“Android式多版本库协同”可以看到管理多版本库的另外一个可行方案。

如果在局域网内维护的版本库所引用的子模组版本库在另外的服务器，甚至在互联网上，克隆子版本库就要浪费很多时间。而且如果子模组指向的版本库不在我们的掌控之内，一旦需要对其进行定制会因为提交无法向远程服务器推送而无法实现。在下一章即第24章“子树合并”中，会给出针对这个问题的解决方案。

# 子树合并

使用子树合并，同样可以实现在一个项目中引用其他项目的数据。但是和子模组方式不同的是，使用子树合并模式，外部的版本库整个复制到本版本库中并建立跟踪关联。使用子树合并模型，使得对源自外部版本库的数据的访问和本版本库数据的访问没有区别，也可以对其进行本地修改，并且能够以子树合并的方式将源自外部版本库的改动和本地的修改相合并。

## 引入外部版本库

为演示子树合并，需要至少准备两个版本库，一个是将被作为子目录引入的版本库util.git，另外一个是主版本库main.git。

```
$ git --git-dir=/path/to/repos/util.git init --bare  
$ git --git-dir=/path/to/repos/main.git init --bare
```

在本地检出这两个版本库：

```
$ git clone /path/to/repos/util.git  
$ git clone /path/to/repos/main.git
```

需要为这两个空版本库添加些数据。非常简单，每个版本库下只创建两个文件：`:file:`Makefile`` 和 `:file:`version``。当执行`:command:`make`` 命令时显示 `version` 文件的内容。对`:file:`version`` 文件多次提交以建立多个提交历史。别忘记在最后使用`:command:`git push origin master`` 将版本库推送到远程版本库中。

`:file:`Makefile`` 文件示例如下。注意第二行前面的空白是`<TAB>`字符，而非空格。

```
all:  
    @cat version
```

在之前尝试的`:command:`git fetch`` 命令都是获取同一项目的版本库的内容。其实命令`:command:`git fetch`` 从哪个项目获取数据并没有什么限制，因为Git的版本库不像Subversion那样用一个唯一的UUID标识让Subversion的版本库之间势同水火。当然也可以用`:command:`git pull`` 来获取其他版本库中的提交，但是那样将把两个项目的文件彻底混杂在一起。对于这个示例来说，因为两个项目具有同样的文件`:file:`Makefile`` 和 `:file:`version``，使用`:command:`git pull`` 将导致冲突。所以为了将不同项目的版本库引入，并在稍候以子树合并方式添加到一个子目录中，需要

用:command:`git fetch`命令从其他版本库获取数据。

- 为了便于以后对外部版本库的跟踪，在使用:command:`git fetch`前，先在main版本库中注册远程版本库util.git。

```
$ git remote add util /path/to/repos/util.git
```

- 查看注册的远程版本库。

```
$ git remote -v
origin  /path/to/repos/main.git/ (fetch)
origin  /path/to/repos/main.git/ (push)
util    /path/to/repos/util.git (fetch)
util    /path/to/repos/util.git (push)
```

- 执行:command:`git fetch`命令获取util.git版本库的提交。

```
$ git fetch util
```

- 查看分支，包括远程分支。

```
$ git branch -a
* master
  remotes/origin/master
  remotes/util/master
```

在不同的分支：master分支和remotes/util/master分支，文件:file:`version`的内容并不相同，因为来自不同的上游版本库。

- master分支中执行:command:`make`命令，显示的是main.git版本库中:file:`version`文件的内容。

```
$ make
main v2010.1
```

- 从util/master远程分支创建一个本地分支util-branch，并切换分支。

```
$ git checkout -b util-branch util/master
Branch util-branch set up to track remote branch master from util.
Switched to a new branch 'util-branch'
```

- 执行:command:`make`命令，显示的是util.git版本库中:file:`version`文件的内容。

```
$ make  
util v3.0
```

像这样在main.git中引入util.git显然不能满足需要，因为在main.git的本地克隆版本库中，master分支访问不到只有在util-branch分支中才出现的util版本库数据。这就需要做进一步的工作，将两个版本库的内容合并到一个分支中。即util-branch分支的数据作为子目录加入到master分支。

## 子目录方式合并外部版本库

下面就用git的底层命令:command:`git read-tree`、:command:`git write-tree`、:command:`git commit-tree`子命令实现将util-branch分支所包含的util.git版本库的目录树以子目录(:file:`lib/`)型式添加到master分支。

先来看看util-branch分支当前最新提交，记住最新提交所指向的目录树(tree)，即tree-id: 0c743e4。

```
$ git cat-file -p util-branch  
tree 0c743e49e11019678c8b345e667504cb789431ae  
parent f21f9c10cc248a4a28bf7790414baba483f1ec15  
author Jiang Xin <jiangxin@osssxp.com> 1288494998 +0800  
committer Jiang Xin <jiangxin@osssxp.com> 1288494998 +0800  
  
util v2.0 -> v3.0
```

查看tree 0c743e4所包含的内容，会看到两个文件: :file:`Makefile`和:file:`version`。

```
$ git cat-file -p 0c743e4  
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3      Makefile  
100644 blob bebe6b10eb9622597dd2b641efe8365c3638004e      version
```

切换到master分支，如下方式调用:command:`git read-tree`将util-branch分支的目录树读取到当前分支:file:`lib`目录下。

- 切换到master分支。

```
$ git checkout master
```

- 执行`:command:`git read-tree``命令，将分支util-branch读取到当前分支的一个子目录下。

```
$ git read-tree --prefix=lib util-branch
```

- 调用`:command:`git read-tree``只是更新了index，所以查看工作区状态，会看到`:file:`lib``目录下的两个文件在工作区中还不存在。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   lib/Makefile
#       new file:   lib/version
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
#
#       deleted:    lib/Makefile
#       deleted:    lib/version
#
```

- 执行检出命令，将`:file:`lib``目录下的文件更新出来。

```
$ git checkout -- lib
```

- 再次查看状态，会看到前面执行的`:command:`git read-tree``命令添加到暂存区中的文件。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   lib/Makefile
#       new file:   lib/version
#
```

现在还不能忙着提交，因为如果现在进行提交就体现不出来两个分支的合并关系。需要使用Git底层的命令进行数据提交。

- 调用`:command:`git write-tree``将暂存区的目录树保存下来。

要记住调用`:command:`git write-tree``后形成的新的tree-id: 2153518。

```
$ git write-tree  
2153518409d218609af40babedec6e8ef51616
```

- 执行`:command:`git cat-file``命令显示这棵树的内容，会注意到其中`:file:`lib``目录的tree-id和之前查看过的util-branch分支最新提交对应的tree-id一样都是0c743e4。

```
$ git cat-file -p 2153518409d218609af40babedec6e8ef51616  
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3      Makefile  
040000 tree 0c743e49e11019678c8b345e667504cb789431ae      lib  
100644 blob 638c7b7c6bdbde1d29e0b55b165f755c8c4332b5      version
```

- 要手工创建一个合并提交，即新的提交要有两个父提交。这两个父提交分别是master分支和util-branch分支的最新提交。用下面的命令显示两个提交的提交ID，并记下这两个提交ID。

```
$ git rev-parse HEAD  
911b1af2e0c95a2fc1306b8dea707064d5386c2e  
$ git rev-parse util-branch  
12408a149bfa78a4c2d4011f884aa2adb04f0934
```

- 执行`:command:`git commit-tree``命令手动创建提交。新提交的目录树来自上面`:command:`git write-tree``产生的目录树（tree-id为2153518），而新提交（合并提交）的两个父提交直接用上面`:command:`git rev-parse``显示的两个提交ID表示。

```
$ echo "subtree merge" | \  
  git commit-tree 2153518409d218609af40babedec6e8ef51616 \  
  -p 911b1af2e0c95a2fc1306b8dea707064d5386c2e \  
  -p 12408a149bfa78a4c2d4011f884aa2adb04f0934  
62ae6cc3f9280418bdb0fcf6c1e678905b1fe690
```

- 执行`:command:`git commit-tree``命令的输出是提交之后产生的新提交的提交ID。需要把当前的master分支重置到此提交ID。

```
$ git reset 62ae6cc3f9280418bdb0fcf6c1e678905b1fe690
```

- 查看一下提交日志及分支图，可以看到通过复杂的`git read-tree`、`git write-tree`和`git commit-tree`命令制造的合并提交，的确将两个不同版本库合并到一起了。

```
$ git log --graph --pretty=oneline
*   62ae6cc3f9280418bdb0fcf6c1e678905b1fe690 subtree merge
|\ \
| * 12408a149bfa78a4c2d4011f884aa2adb04f0934 util v2.0 -> v3.0
| * f21f9c10cc248a4a28bf7790414baba483f1ec15 util v1.0 -> v2.0
| * 76db0ad729db9fdc5be043f3b4ed94ddc945cd7f util v1.0
* 911b1af2e0c95a2fc1306b8dea707064d5386c2e main v2010.1
```

- 看看现在的master分支。

```
$ git cat-file -p HEAD
tree 2153518409d218609af40babedec6e8ef51616
parent 911b1af2e0c95a2fc1306b8dea707064d5386c2e
parent 12408a149bfa78a4c2d4011f884aa2adb04f0934
author Jiang Xin <jiangxin@osssxp.com> 1288498186 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1288498186 +0800

subtree merge
```

- 看看目录树。

```
$ git cat-file -p 2153518409d218609af40babedec6e8ef51616
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3      Makefile
040000 tree 0c743e49e11019678c8b345e667504cb789431ae      lib
100644 blob 638c7b7c6bdbde1d29e0b55b165f755c8c4332b5      version
```

整个过程非常繁琐，但是不要太过担心，只需要对原理了解清楚就可以了，因为在后面会介绍一个Git插件封装了复杂的子树合并操作。

## 利用子树合并跟踪上游改动

如果子树（`lib` 目录）的上游（即 `util.git`）包含了新的提交，如何将 `util.git` 的新提交合并过来呢？这就要用到名为 `subtree` 的合并策略。参见第3篇第16章第16.6小节“合并策略”中相关内容。

在执行子树合并之前，先切换到 `util-branch` 分支，获取远程版本库改动。

```
$ git checkout util-branch

$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /path/to/repos/util
  12408a1..5aba14f master      -> util/master
Updating 12408a1..5aba14f
Fast-forward
 version | 2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)

$ git checkout master
```

在切换回master分支后，如果这时执行:`git merge util-branch`，会将util-branch的数据直接合并到master分支的根目录下，而实际上是希望合并发生在`lib`目录中，这就需要如下方式进行调用，以subtree策略进行合并。

如果Git的版本小于1.7，直接使用subtree合并策略。

```
$ git merge -s subtree util-branch
```

如果Git的版本是1.7之后（含1.7）的版本，则可以使用缺省的recursive合并策略，通过参数`subtree=<prefix>`在合并时使用正确的子树进行匹配合并。避免了使用subtree合并策略时的猜测。

```
$ git merge -Xsubtree=lib util-branch
```

再来看看执行子树合并之后的分支图示。

```
$ git log --graph --pretty=oneline
*   f1a33e55eea04930a500c18a24a8bd009ecd9ac2 Merge branch 'util-branch'
| \
| * 5aba14fd347fc22cd8fb086c9f26a53276f15c9 util v3.1 -> v3.2
| * a6d53dfcf78e8a874e9132def5ef87a2b2febfa5 util v3.0 -> v3.1
* |   62ae6cc3f9280418bdb0fcf6c1e678905b1fe690 subtree merge
| \ \
| |
| * 12408a149bfa78a4c2d4011f884aa2adb04f0934 util v2.0 -> v3.0
| * f21f9c10cc248a4a28bf7790414babab483f1ec15 util v1.0 -> v2.0
| * 76db0ad729db9fdc5be043f3b4ed94ddc945cd7f util v1.0
* 911b1af2e0c95a2fc1306b8dea707064d5386c2e main v2010.1
```

## 子树拆分

既然可以将一个代码库通过子树合并方式作为子目录加入到另外一个版本库中，反之也可以将一个代码库的子目录独立出来转换为另外的版本库。不过这个反向过程非常复杂。要将一个版本库的子目录作为顶级目录导出到另外的项目，潜藏的条件是要导出历史的，因为如果不关心历史，直接文件拷贝重建项目就可以了。子树拆分的大致过程是：

1. 找到要导出的目录的提交历史，并反向排序。
2. 依次对每个提交执行下面的操作：
  3. 找出提交中导出目录对应的tree-id。
  4. 对该tree-id执行`:command:`git commit-tree``。
  5. 执行`:command:`git commit-tree``要保持提交信息还要重新设置提交的父提交(parent)。

手工执行这个操作复杂且易出错，可以用下节介绍的`git subtree`插件，或使用第6篇第35.4小节“Git版本库整理”中介绍的`:command:`git filter-branch``命令进行子目录过滤。

## git subtree插件

Git subtree插件用shell脚本开发，安装之后为Git提供了新的`:command:`git subtree``命令，支持前面介绍的子树合并和子树拆分。命令非常简单易用，将其他版本库以子树形式导入，再也不必和底层的Git命令打交道了。

Gitsubtree 插件的作者将代码库公布在Github上：<http://github.com/apenwarr/git-subtree/>。

安装Git subtree很简单：

```
$ git clone git://github.com/apenwarr/git-subtree.git
$ cd git-subtree
$ make doc
$ make test
$ sudo make install
```

### `:command:`git subtree add``

命令`:command:`git subtree add``相当于将其他版本库以子树方式加入到当前版本库。用法：

```
git subtree add [--squash] -P <prefix> <commit>
git subtree add [--squash] -P <prefix> <repository> <refspec>
```

其中可选的`--squash`含义为压缩为一个版本后再添加。

对于文章中的示例，为了将util.git合并到main.git的`:file:`lib``目录。可以直接这样调用：

```
$ git subtree add -P lib /path/to/repos/util.git master
```

不过推荐的方法还是先在本地建立util.git版本库的追踪分支。

```
$ git remote add util /path/to/repos/util.git
$ git fetch util
$ git checkout -b util-branch util/master
$ git subtree add -P lib util-branch
```

## :command:`git subtree merge`

命令`:command:`git subtree merge``相当于将子树对应的远程分支的更新重新合并到子树中，相当于完成了`:command:`git merge -s subtree``操作。用法：

```
git subtree merge [--squash] -P <prefix> <commit>
```

其中可选的`--squash`含义为压缩为一个版本后再合并。

对于文章中的示例，为了将util-branch分支包含的上游最新改动合并到master分支的`:file:`lib``目录。可以直接这样调用：

```
$ git subtree merge -P lib util-branch
```

## :command:`git subtree pull`

命令`:command:`git subtree pull``相当于先对子树对应的远程版本库执行一次`:command:`git fetch``操作，然后再执行`:command:`git subtree merge``。用法：

```
git subtree pull [--squash] -P <prefix> <repository> <refspec...>
```

对于文章中的示例，为了将util.git版本库的master分支包含的最新改动合并到master分支的:file:`lib`目录。可以直接这样调用：

```
$ git subtree pull -P lib /path/to/repos/util.git master
```

更喜欢用前面介绍的:command:`git subtree merge`命令，因为:command:`git subtree pull`存在版本库地址写错的风险。

### :command:`git subtree split`

命令:command:`git subtree split`相当将目录拆分为独立的分支，即子树拆分。拆分后形成的分支可以通过推送到新的版本库实现原版本库的目录独立为一个新的版本库。用法：

```
git subtree split -P <prefix> [--branch <branch>] [--onto ...] [--ignore-join
```

说明：

- 该命令的总是输出子树拆分后的最后一个commit-id。这样可以通过管道方式传递给其他命令，如:command:`git subtree push`命令。
- 参数--branch提供拆分后创建的分支名称。如果不提供，只能通过:command:`git subtree split`命令提供的提交ID得到拆分的结果。
- 参数--onto参数将目录拆分附加于已经存在的提交上。
- 参数--ignore-joins忽略对之前拆分历史的检查。
- 参数--rejoin会将拆分结果合并到当前分支，因为采用ours的合并策略，不会破坏当前分支。

### :command:`git subtree push`

命令:command:`git subtree push`先执行子树拆分，再将拆分的分支推送到远程服务器。用法：

```
git subtree push -P <prefix> <repository> <refspec...>
```

该命令的用法和:command:`git subtree split`类似，不再赘述。

# Android式多版本库协同

Android是谷歌（Google）开发的适合手持设备的操作系统，提供了当前最为吸引眼球的开源的手机操作平台，大有超越苹果（Apple.com）的专有的iOS的趋势。而Android的源代码就是使用Git进行维护的。Android项目在使用Git进行源代码管理上有两个伟大的创造，一个是用Python语言开发名为repo的命令行工具用于多版本库的管理，另外一个是用Java开发的名为Gerrit的代码审核服务器。本节重点介绍repo是如何管理多代码库的。

Android的源代码的Git库有160多个（截止至2010年10月）：

- Android的版本库管理工具repo:

```
git://android.git.kernel.org/tools/repo.git
```

- 保存GPS配置文件的版本库

```
git://android.git.kernel.org/device/common.git
```

- 160多个其他的版本库...

如果要是把160多个版本库都列在这里，恐怕各位的下巴会掉下来。那么为什么Android的版本库这么多呢？怎么管理这么复杂的版本库呢？

Android版本库众多的原因，主要原因是版本库太大以及Git不能部分检出。Android的版本库有接近2个GB之多。如果所有的东西都放在一个库中，而某个开发团队感兴趣的可能就是某个驱动，或者是某个应用，却要下载如此庞大的版本库，是有些说不过去。

好了，既然接受了Android有多达160多个版本库这一事实，那么Android是不是用之前介绍的“子模组”方式组织起来的呢？如果真的用“子模组”方式来管理这160个代码库，可能就需要如此管理：

- 建立一个索引版本库，在该版本库中，通过子模组方式，将一个一个的目录对应到160多个版本库。
- 当对此索引版本库执行克隆操作后，再执行`:command:`git submodule init``命令。
- 当执行`:command:`git submodule update``命令时，开始分别克隆这160多个版本库。
- 如果想修改某个版本库中的内容，需要进入到相应的子模组目录，执行切换分支的操作。因为子模组是以某个固定提交的状态存在的，是不能更改的，必须先切换到某个工作分支后，才能进行修改和提交。
- 如果要将所有的子模组都切换到某个分支（如master）进行修改，必须自己通过脚本对这160多个版本库一一切换。
- Android有多个版本：android-1.0、android-1.5、...、android-2.2\_r1.3、... 如何维护这么多的版本呢？也许索引库要通过分支和里程碑，和子模组的各个不同的提交

状态进行对应。但是由于子模组的状态只是一个提交ID，如何能够动态指定到分支，真的给不出答案。

幸好，上面只是假设。聪明的Android程序设计师一早就考虑到了Git子模组的局限性以及多版本库管理的问题，开发出了repo这一工具。

关于repo有这么一则小故事：Android之父安迪·鲁宾在回应乔布斯关于Android太开放导致开发维护更麻烦的言论时，在Twitter (<http://twitter.com/Arubin>) 上留了下面这段简短的话：

```
the definition of open: "mkdir android ; cd android ; repo init -u git://andr
```

是的，就是repo让Android的开发变得如此简单。

## 关于repo

Repo是Google开发的用于管理Android版本库的一个工具。Repo并不是用于取代Git，是用Python对Git进行了一定的封装，简化了对多个Git版本库的管理。对于repo管理的任何一个版本库，都还是需要使用Git命令进行操作。

repo的使用过程大致如下：

- 运行`:command:`repo init``命令，克隆Android的一个清单库。这个清单库和前面假设的“子模组”方式工作的索引库不同，是通过XML技术建立的版本库清单。
- 清单库中的`:file:`manifest.xml``文件，列出了160多个版本库的克隆方式。包括版本库的地址和工作区地址的对应关系，以及分支的对应关系。
- 运行`:command:`repo sync``命令，开始同步，即分别克隆这160多个版本库到本地的工作区中。
- 同时对160多个版本库执行切换分支操作，切换到某个分支。

## 安装repo

首先下载repo的引导脚本，可以使用wget、curl甚至浏览器从地址<http://android.git.kernel.org/repo>下载。把repo脚本设置为可执行，并复制到可执行的路径中。在Linux上可以用下面的指令将repo下载并复制到用户主目录的`:file:`bin``目录下。

```
$ curl http://android.git.kernel.org/repo > ~/bin/repo  
$ chmod a+x ~/bin/repo
```

为什么说下载的repo只是一个引导脚本（bootstrap）而不是直接称为repo呢？因为repo的大部分功能代码不在其中，下载的只是一个帮助完成整个repo程序的继续下载和加载工具。如果您是一个程序员，对repo的执行比较好奇，可以一起分析一下repo引导脚本。否则可以跳到下一节。

看看repo引导脚本的前几行（为方便描述，把注释和版权信息过滤掉了），会发现一个神奇的魔法：

```
1  #!/bin/sh
2
3  REPO_URL='git://android.git.kernel.org/tools/repo.git'
4  REPO_REV='stable'
5
6  magic='--calling-python-from-/bin/sh--'
7  """exec" python -E "$0" "$@" """#$magic"
8  if __name__ == '__main__':
9      import sys
10     if sys.argv[-1] == '#%s' % magic:
11         del sys.argv[-1]
12     del magic
```

Repo引导脚本是用什么语言开发的？这是一个问题。

- 第1行，有经验的Linux开发者会知道此脚本是用Shell脚本语言开发的。
- 第7行，是这个魔法的最神奇之处。既是一条合法的shell语句，又是一条合法的python语句。
- 第7行作为shell语句，执行:`:command:`exec``，用python调用本脚本，并替换本进程。三引号在这里相当于一个空字符串和一个单独的引号。
- 第7行作为python语句，三引号定义的是一个字符串，字符串后面是一个注释。
- 实际上第1行到第7行，即是合法的shell语句又是合法的python语句。从第8行开始后面都是python脚本了。
- Repo引导脚本无论是使用shell执行，或是用python执行，效果都相当于使用python执行此脚本。

Repo真正的位置在哪里？

在引导脚本repo的main函数，首先调用``` _FindRepo ```函数，从当前目录开始依次向上递归查找:`:file:`.repo/repo/main.py``文件。

```
def main(orig_args):
    main, dir = _FindRepo()
```

函数`_FindRepo`返回找到的`:file:`.repo/repo/main.py``脚本文件，以及包

含:`:file:`repo/main.py``的:`:file:`.repo``目录。如果找到:`:file:`.repo/repo/main.py``脚本，则把程序的控制权交给:`:file:`.repo/repo/main.py``脚本。（省略了在repo开发库中执行情况的判断）

在下载repo引导脚本后，没有初始化之前，当然不会存在:`:file:`.repo/repo/main.py``脚本，这时必须进行初始化操作。

## repo和清单库的初始化

下载并保存repo引导脚本后，建立一个工作目录，这个工作目录将作为Android的工作区目录。在工作目录中执行:`:command:`repo init -u <url>``完成repo完整的下载以及项目清单版本库(`manifest.git`)的下载。

```
$ mkdir working-directory-name  
$ cd working-directory-name  
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

命令:`:command:`repo init``要完成如下操作：

- 完成repo这一工具的完整下载，因为现在有的不过是repo的引导程序。

初始化操作会从Android的代码中克隆`repo.git`库，到当前目录下的:`:file:`.repo/repo``目录下。在完成`repo.git`克隆之后，`repo init`命令会将控制权交给工作区的:`:file:`.repo/repo/main.py``这个刚刚从`repo.git`库克隆来的脚本文件，继续进行初始化。

- 克隆 android 的清单库 `manifest.git`（地址来自于 `-u` 参数）。

克隆的清单库位于:`:file:`.repo/manifests.git``中，并本地克隆到:`:file:`.repo/manifests``。清单文件:`:file:`.repo/manifest.xml``是符号链接指向:`:file:`.repo/manifests/default.xml``。

- 提问用户的姓名和邮件地址，如果和Git缺省的用户名、邮件地址不同，则记录在:`:file:`.repo/manifests.git``库的 `config` 文件中。
- 命令:`:command:`repo init``还可以附带`--mirror`参数，以建立和上游Android的版本库一模一样的镜像。会在后面的章节介绍。

### 从哪里下载`repo.git`？

在repo引导脚本的前几行，定义了缺省的`repo.git`的版本库位置以及要检出的缺省分支。

```
REPO_URL='git://android.git.kernel.org/tools/repo.git'
```

```
REPO_REV='stable'
```

如果不想从缺省任务获取repo，或者不想获取稳定版（stable分支）的repo，可以在`:command:`repo init``命令中通过下面的参数覆盖缺省的设置，从指定的源地址克隆repo代码库。

- 参数`--repo-url`，用于设定repo的版本库地址。
- 参数`--repo-branch`，用于设定要检出的分支。
- 参数`--no-repo-verify`，设定不要对repo的里程碑签名进行严格的验证。

实际上，完成repo.git版本库的克隆，这个repo引导脚本就江郎才尽了，`init`命令的后续处理（以及其他命令）都交给刚刚克隆出来的`:file:`.repo/repo/main.py``来继续执行。

### 清单库是什么？从哪里下载？

清单库实际上只包含一个`:file:`default.xml``文件。这个XML文件定义了多个版本库和本地地址的映射关系，是repo工作的指引文件。所以在使用repo引导脚本进行初始化的时候，必须通过`-u`参数指定清单库的源地址。

清单库的下载，是通过`:command:`repo init``命令初始化时，用`-u`参数指定清单库的位置。例如repo针对Android代码库进行初始化时执行的命令：

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

Repo引导脚本的```init```命令可以使用下列和清单库相关的参数：

- 参数`-u (--manifest-url)`：设定清单库的Git服务器地址。
- 参数`-b (--manifest-branch)`：检出清单库特定分支。
- 参数`--mirror`：只在repo第一次初始化的时候使用，以和Android服务器同样的结构在本地建立镜像。
- 参数`-m (--manifest-name)`：当有多个清单文件，可以指定清单库的某个清单文件为有效的清单文件。缺省为`:file:`default.xml``。

Repo初始化命令（`repo init`）可以执行多次：

- 不带参数的执行`:command:`repo init``，从上游的清单库获取新的清单文件`:file:`default.xml``。
- 使用参数`-u (--manifest-url)`执行`:command:`repo init``，会重新设定上游的清单库地址，并重新同步。
- 使用参数`-b (--manifest-branch)`执行`:command:`repo init``，会使用清单库的不同分支，以便在使用`:command:`repo sync``时将项目同步到不同的里程碑。
- 但是不能使用`--mirror`命令，该命名只能在第一次初始化时执行。那么如何将已经按

照工作区模式同步的版本库转换为镜像模式呢？会在后面看到一个解决方案。

## 清单库和清单文件

当执行完毕`:command:`repo init``之后，工作目录内空空如也。实际上有一个`:file:`.repo``目录。在该目录下除了一个包含repo的实现的repo库克隆外，就是manifest库的克隆，以及一个符号链接链接到清单库中的`:file:`default.xml``文件。

```
$ ls -lF .repo/
drwxr-xr-x 3 jiangxin jiangxin 4096 2010-10-11 18:57 manifests/
drwxr-xr-x 8 jiangxin jiangxin 4096 2010-10-11 10:08 manifests.git/
lrwxrwxrwx 1 jiangxin jiangxin   21 2010-10-11 10:07 manifest.xml -> manifest
drwxr-xr-x 7 jiangxin jiangxin 4096 2010-10-11 10:07 repo/
```

在工作目录下的`:file:`.repo/manifest.xml``文件就是Android项目的众多版本库的清单文件。Repo命令的操作，都要参考这个清单文件。

打开清单文件，会看到如下内容：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <manifest>
3   <remote name="korg"
4     fetch="git://android.git.kernel.org/"
5     review="review.source.android.com" />
6   <default revision="master"
7     remote="korg" />
8
9   <project path="build" name="platform/build">
10    <copyfile src="core/root.mk" dest="Makefile" />
11  </project>
12
13  <project path="bionic" name="platform/bionic" />
14
15 ...
16
17 ...
181 </manifest>
```

这个文件不太复杂，是么？

- 这个XML的顶级元素是`manifest`，见第2行和第181行。
- 第3行通过一个`remote`元素，定义了名为`korg` (`kernel.org`缩写) 的源，其`Git`库的地址为`git://android.git.kernel.org/`，还定义了代码审核服务器的地址`review.source.android.com`。还可以定义更多的`remote`元素，这里只定义了一个。

- 第6行用于设置各个项目缺省的远程源地址（remote）为korg，缺省的分支为master。当然各个项目（project元素）可以定义自己的remote和revision覆盖该缺省配置。
- 第9行定义一个项目，该项目的远程版本库相对路径为：“platform/build”，在工作区克隆的位置为：“build”。
- 第10行，即project元素的子元素copyfile，定义了项目克隆后的一个附加动作：拷贝文件从:file:`core/root.mk`至:file:`Makefile`。
- 第13行后后续的100多行定义了其他160个项目，都是采用类似的project元素语法。name参数定义远程版本库的相对路径，path参数定义克隆到本地工作区的路径。
- 还可以出现manifest-server元素，其url属性定义了通过XMLRPC提供实时更新清单的服务器URL。只有当执行:command:`repo sync --smart-sync`的时候，才会检查该值，并用动态获取的manifest覆盖掉缺省的清单。

## 同步项目

在工作区，执行下面的命令，会参照:file:`.repo/manifest.xml`清单文件，将项目所有相关的版本库全部克隆出来。不过最好请在读完本节内容之后再尝试执行这条命令。

```
$ repo sync
```

对于Android，这个操作需要通过网络传递接近2个GB的内容，如果带宽不是很高的话，可能会花掉几个小时甚至是一天的时间。

也可以仅克隆感兴趣的项目，在:command:`repo sync`后面跟上项目的名称。项目的名称来自于:file:`.repo/manifest.xml`这个XML文件中project元素的name属性值。例如克隆platform/build项目：

```
$ repo sync platform/build
```

Repo有一个功能可以在这里展示。就是repo支持通过本地清单，对缺省的清单文件进行补充以及覆盖。即可以在:file:`.repo`目录下创建:file:`local\_manifest.xml`文件，其内容会和:file:`.repo/manifest.xml`文件的内容进行合并。

在工作目录下运行下面的命令，可以创建一个本地清单文件。这个本地定制的清单文件来自缺省文件，但是删除了remote元素和default元素，并将所有的project元素都重命名为remove-project元素。这实际相当于对原有的清单文件“取反”。

```
$ sed -e '/<remote/,+4 d' -e 's/<project/<remove-project/g' \
-e 's/</project>/<remove-project>/g' \
< .repo/manifest.xml > .repo/local_manifest.xml
```

用下面的这条命令可以看到repo运行时实际获取到的清单。这个清单来自于`:file:`.repo/manifest.xml``和`:file:`.repo/local_manifest.xml``两个文件的汇总。

```
$ repo manifest -o -
```

当执行`:command:`repo sync``命令时，实际上就是依据合并后的清单文件进行同步。如果清单中的项目被自定义清单全部“取反”，执行同步就不会同步任何项目，甚至会删除已经完成同步的项目。

本地定制的清单文件`:file:`local_manifest.xml``支持前面介绍的清单文件的所有语法，需要注意的是：

- 不能出现重复定义的`remote`元素。这就是为什么上面的脚本要删除来自`manifest.xml`的`remote`元素。
- 不能出现`default`元素，仅为全局仅能有一个。
- 不能出现重复的`project`定义（`name`属性不能相同），但是可以通过`remove-project`元素将缺省清单中定义的`project`删除再重新定义。

试着编辑`:file:`.repo/local_manifest.xml``，在其中再添加几个`project`元素，然后试着用`:command:`repo sync``命令进行同步。

## 建立Android代码库本地镜像

Android为企业提供一个新的市场，无论大企业，小企业都是处于同一个起跑线上。研究Android尤其是Android系统核心或者是驱动的开发，首先需要做的就是本地克隆建立一套Android版本库管理机制。因为Android的代码库是那么庞杂，如果一个开发团队每个人都去执行`:command:`repo init -u``，再执行`:command:`repo sync``从Android服务器克隆版本库的话，多大的网络带宽恐怕都不够用。唯一的办法是本地建立一个Android版本库的镜像。

建立本地镜像非常简单，就是在执行`:command:`repo init -u``初始化的时候，附带上`--mirror`参数。

```
$ mkdir android-mirror-dir  
$ cd android-mirror-dir  
$ repo init --mirror -u git://android.git.kernel.org/platform/manifest.git
```

之后执行`:command:`repo sync``就可以安装Android的Git服务器方式来组织版本库，创建一个Android版本库镜像。

实际上附带了`--mirror`参数执行`:command:`repo init -u``命令，会在克隆

的`:file:`.repo/manifests.git``下的`:file:`config``中记录配置信息：

```
[repo]
  mirror = true
```

## 从Android的工作区到代码库镜像

在初始化repo工作区时，如果使用不带`--mirror`参数的`:command:`repo init -u``，并完成代码同步后，如果再次执行`:command:`repo init``并附带了`--mirror`参数，repo 会报错退出：“fatal: --mirror not supported on existing client”。实际上`--mirror`参数只能对尚未初始化的repo工作区执行。

那么如果之前没有用镜像的方法同步Android版本库，难道要为创建代码库镜像再重新执行一次repo同步么？要知道重新同步一份Android版本库是非常慢的。我就遇到了这个问题。

不过既然有`:file:`manifest.xml``文件，完全可以对工作区进行反向操作，将工作区转换为镜像服务器的结构。下面就是一个示例脚本，这个脚本利用了已有的repo代码进行实现，所以看着很简洁。8-)

脚本`:file:`work2mirror.py``如下：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os, sys, shutil

cwd = os.path.abspath( os.path.dirname( __file__ ) )
repodir = os.path.join( cwd, '.repo' )
S_repo = 'repo'
TRASHDIR = 'old_work_tree'

if not os.path.exists( os.path.join(repodir, S_repo) ):
    print >> sys.stderr, "Must run under repo work_dir root."
    sys.exit(1)

sys.path.insert( 0, os.path.join(repodir, S_repo) )
from manifest_xml import XmlManifest

manifest = XmlManifest( repodir )

if manifest.IsMirror:
    print >> sys.stderr, "Already mirror, exit."
    sys.exit(1)

trash = os.path.join( cwd, TRASHDIR )
```

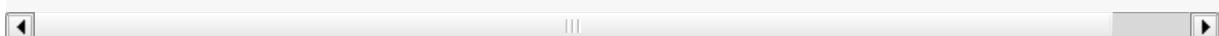
```
for project in manifest.projects.iterator():
    # 移动旧的版本库路径到镜像模式下新的版本库路径
    newgitdir = os.path.join( cwd, '%s.git' % project.name )
    if os.path.exists( project.gitdir ) and project.gitdir != newgitdir:
        if not os.path.exists( os.path.dirname(newgitdir) ):
            os.makedirs( os.path.dirname(newgitdir) )
        print "Rename %s to %s." % (project.gitdir, newgitdir)
        os.rename( project.gitdir, newgitdir )

    # 移动工作区到待删除目录
    if project.worktree and os.path.exists( project.worktree ):
        newworktree = os.path.join( trash, project.relpah )
        if not os.path.exists( os.path.dirname(newworktree) ):
            os.makedirs( os.path.dirname(newworktree) )
        print "Move old worktree %s to %s." % (project.worktree, newworktree)
        os.rename( project.worktree, newworktree )

    if os.path.exists ( os.path.join( newgitdir, 'config' ) ):
        # 修改版本库的配置
        os.chdir( newgitdir )
        os.system( "git config core.bare true" )
        os.system( "git config remote.korg.fetch '+refs/heads/*:refs/heads/*'" )

        # 删除 remotes 分支, 因为作为版本库镜像不需要 remote 分支
        if os.path.exists ( os.path.join( newgitdir, 'refs', 'remotes' ) ):
            print "Delete " + os.path.join( newgitdir, 'refs', 'remotes' )
            shutil.rmtree( os.path.join( newgitdir, 'refs', 'remotes' ) )

    # 设置 manifest 为镜像
    mp = manifest.manifestProject
    mp.config.SetString('repo.mirror', 'true')
```



使用方法很简单，只要将脚本放在Android工作区下，执行就可以了。执行完毕会将原有工作区的目录移动到`:file:`old_work_tree``子目录下，在确认原有工作区没有未提交的数据后，直接删除`:file:`old_work_tree``即可。

```
$ python work2mirror.py
```

### 创建新的清单库，或修改原有清单库

建立了Android代码库的本地镜像后，如果不对`manifest`清单版本库进行定制，在使用`:command:`repo sync``同步代码的时候，仍然使用Android官方的代码库同步代码，使得本地的镜像版本库形同虚设。解决办法是创建一个自己的`manifest`库，或者在原有清单库中建立一个分支加以修改。如果创建新的清单库，参考Android上游的`manifest`清单库进行创建。

## Repo的命令集

Repo命令实际上是Git命令的简单或者复杂的封装。每一个repo命令都对应于repo源码树中`:file:`subcmds``目录下的一个同名的Python脚本。每一个repo命令都可以通过下面的命令获得帮助。

```
$ repo help <command>
```

通过阅读代码，可以更加深入的了解repo命令的封装。

### :command:`repo init`命令

:command:`repo init`命令，主要完成检出清单版本库（`manifest.git`），以及配置Git用户的用户名和邮件地址的工作。

实际上，完全可以进入到`:file:`.repo/manifests``目录，用git命令操作清单库。  
对`manifests`的修改不会因为执行:command:`repo init`而丢失，除非是处于未跟踪状态。

### :command:`repo sync`命令

:command:`repo sync`命令用于参照清单文件克隆或者同步版本库。如果某个项目版本库尚不存在，则执行:command:`repo sync`命令相当于执行:command:`git clone`。如果项目版本库已经存在，则相当于执行下面的两个命令：

- `git remote update`

相当于对每一个remote源执行`fetch`操作。

- `git rebase origin/branch`

针对当前分支的跟踪分支，执行`rebase`操作。不采用`merge`而是采用`rebase`，目的是减少提交数量，方便评审(Gerrit)。

### :command:`repo start`命令

:command:`repo start`命令实际上是对:command:`git checkout -b`命令的封装。为指定的项目或者所有项目（若使用`--all`参数），以清单文件中为项目设定的分支或里程碑为基础，创建特性分支。特性分支的名称由命令的第一个参数指定。相当于执行:command:`git checkout -b`。

用法：

```
repo start <newbranchname> [--all | <project>...]
```

## :command:`repo status` 命令

:command:`repo status` 命令实际上是对:command:`git diff-index`、:command:`git diff-files` 命令的封装，同时显示暂存区的状态和本地文件修改的状态。

用法：

```
repo status [<project>...]
```

示例输出：

```
project repo/ branch devwork
-m      subcmds/status.py
...
```

上面示例输出显示了repo项目的devwork分支的修改状态。

- 每个小节的首行显示项目名称，以及所在分支名称。
- 之后显示该项目中文件变更状态。头两个字母显示变更状态，后面显示文件名或者其他变更信息。
- 第一个字母表示暂存区的文件修改状态。

其实是:command:`git-diff-index` 命令输出中的状态标识，并用大写显示。

- -：没有改变
- A：添加（不在HEAD中，在暂存区）
- M：修改（在HEAD中，在暂存区，内容不同）
- D：删除（在HEAD中，不在暂存区）
- R：重命名（不在HEAD中，在暂存区，路径修改）
- C：拷贝（不在HEAD中，在暂存区，从其他文件拷贝）
- T：文件状态改变（在HEAD中，在暂存区，内容相同）
- U：未合并，需要冲突解决

- 第二个字母表示工作区文件的更改状态。

其实是:command:`git-diff-files` 命令输出中的状态标识，并用小写显示。

- -: 新/未知 (不在暂存区, 在工作区)
  - m: 修改 (在暂存区, 在工作区, 被修改)
  - d: 删除 (在暂存区, 不在工作区)
- 两个表示状态的字母后面, 显示文件名信息。如果有文件重命名还会显示改变前后的文件名以及文件的相似度。

## :command:`repo checkout` 命令

:command:`repo checkout` 命令实际上是对:command:`git checkout` 命令的封装。检出之前由:command:`repo start` 创建的分支。

用法:

```
repo checkout <branchname> [<project>...]
```

## :command:`repo branches` 命令

:command:`repo branches` 命令读取各个项目的分支列表并汇总显示。该命令实际上是通过直接读取:file:`.git/refs` 目录下的引用来获取分支列表, 以及分支的发布状态等。

用法:

```
repo branches [<project>...]
```

输出示例:

```
*P nocolor          | in repo
repo2                |
```

- 第一个字段显示分支的状态: 是否是当前分支, 分支是否发布到代码审核服务器上?
- 第一个字母若显示星号(\*), 含义是此分支为当前分支
- 第二个字母若为大写字母P, 则含义是分支所有提交都发布到代码审核服务器上了。
- 第二个字母若为小写字母p, 则含义是只有部分提交被发布到代码审核服务器上。
- 若不显示P或者p, 则表明分支尚未发布。
- 第二个字段为分支名。

- 第三个字段为以竖线（|）开始的字符串，表示该分支存在于哪些项目中。

- | in all projects

该分支处于所有项目中。

- | in project1 project2

该分支只在特定项目中定义。如：project1、project2。

- | not in project1

该分支不存在于这些项目中。即除了project1项目外，其他项目都包含此分支。

## :command:`repo diff` 命令

:command:`repo diff` 命令实际上是对:command:`git diff` 命令的封装，用以分别显示各个项目工作区下的文件差异。

用法：

```
repo diff [<project>...]
```

## :command:`repo stage` 命令

:command:`repo stage` 命令实际上是对:command:`git add --interactive` 命令的封装，用以对各个项目工作区中的改动（修改、添加等）进行挑选以加入暂存区。

用法：

```
repo stage -i [<project>...]
```

## :command:`repo upload` 命令

:command:`repo upload` 命令相当于:command:`git push`，但是又有很大的不同。执行:command:`repo upload` 不是将版本库改动推送到克隆时的远程服务器，而是推送到代码审查服务器（由Gerrit软件架设）的特殊引用上，使用的是SSH协议（特殊端口）。代码审核服务器会对推送的提交进行特殊处理，将新的提交显示为一个待审核的修改集，并进入代码审查流程。只有当审核通过，才会合并到官方正式的版本库中。

用法：

```
repo upload [--re --cc] {[<project>]... | --replace <project>}
```

参数:

- |                                       |                               |
|---------------------------------------|-------------------------------|
| -h, --help                            | 显示帮助信息。                       |
| -t                                    | 发送本地分支名称到 Gerrit 代码审核服务器。     |
| --replace                             | 发送此分支的更新补丁集。注意使用该参数，只能指定一个项目。 |
| --re=REVIEWERS, --reviewers=REVIEWERS | 要求由指定的人员进行审核。                 |
| --cc=CC                               | 同时发送通知到如下邮件地址。                |

## 确定推送服务器的端口

分支改动的推送是发给代码审核服务器，而不是下载代码的服务器。使用的协议是SSH协议，但是使用的并非标准端口。如何确认代码审核服务器上提供的特殊SSH端口呢？

在执行:`:command:`repo upload``命令时，repo会通过访问代码审核Web服务器的`/ssh_info`的URL获取SSH服务端口，缺省29418。这个端口，就是`:command:`repo upload``发起推送的服务器的SSH服务端口。

## 修订集修改后重新传送

当已经通过`:command:`repo upload``命令在代码审查服务器上提交了一个修订集，会得到一个修订号。关于此次修订的相关讨论会发送到提交者的邮箱中。如果修订集有误没有通过审核，可以重新修改代码，再次向代码审核服务器上传修订集。

一个修订集修改后再次上传，如果修订集的ID不变是非常有用的，因为这样相关的修订集都在代码审核服务器的同一个界面中显示。

在执行`:command:`repo upload``时会弹出一个编辑界面，提示在方括号中输入修订集编号，否则会在代码审查服务器上创建新的ID。有一个办法可以不用手工输入修订集，如下：

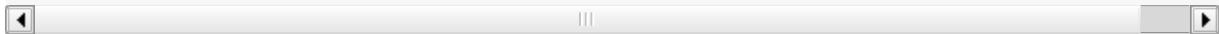
```
repo upload --replace project_name
```

当使用`--replace`参数后，repo会检查本地版本库名为`refs/published/branch_name`的特殊引用（上一次提交的修订），获得其对应的提交SHA1哈希值。然后在代码审核服务器的`refs/changes/`命名空间下的特殊引用中寻找和提交SHA1哈希值匹配的引用，找到的匹配引用其名称中就所包含有变更集ID，直接用此变更集ID作为新的变更集ID提交到代码审核服务器。

## Gerrit服务器魔法

`:command:`repo upload``命令执行推送，实际上会以类似如下的命令行格式进行调用：

```
git push --receive-pack='gerrit receive-pack --reviewer charlie@example.com'  
ssh://review.example.com:29418/project HEAD:refs/for/master
```



当Gerrit服务器接收到:command:`git push`请求后，会自动将对分支的提交转换为修订集，显示于Gerrit的提交审核界面中。Gerrit的魔法破解的关键点就在于:command:`git push`命令的--receive-pack参数。即提交交由:command:`gerrit-receive-pack`命令执行，进入非标准的Git处理流程，将提交转换为在refs/changes命名空间下的引用，而不在refs/for命名空间下创建引用。

## :command:`repo download`命令

:command:`repo download`命令主要用于代码审核者下载和评估贡献者提交的修订。贡献者的修订在Git版本库中以refs/changes/<changeid>/<patchset>引用方式命名（缺省的patchset为1），和其他Git引用一样，用:command:`git fetch`获取，该引用所指向的最新的提交就是贡献者待审核的修订。使用:command:`repo download`命令实际上就是用:command:`git fetch`获取到对应项目的refs/changes/<changeid>/patchset>引用，并自动切换到对应的引用上。

用法：

```
repo download {project change[/patchset]}...
```

## :command:`repo rebase`命令

:command:`repo rebase`命令实际上是对:command:`git rebase`命令的封装，该命令的参数也作为:command:`git rebase`命令的参数。但 -i 参数仅当对一个项执行时有效。

用法：

```
命令行：repo rebase {[<project>...] | -i <project>...}
```

参数：

-h, --help	显示帮助并退出
-i, --interactive	交互式的变基（仅对一个项目时有效）
-f, --force-rebase	向 git rebase 命令传递 --force-rebase 参数
--no-ff	向 git rebase 命令传递 -no-ff 参数
-q, --quiet	向 git rebase 命令传递 --quiet 参数
--autosquash	向 git rebase 命令传递 --autosquash 参数
--whitespace=WS	向 git rebase 命令传递 --whitespace=WS 参数

## :command:`repo prune`命令

:command:`repo prune`命令实际上是对:command:`git branch -d`命令的封装，该命令用于扫描项目的各个分支，并删除已经合并的分支。

用法：

```
repo prune [<project>...]
```

## :command:`repo abandon`命令

相比:command:`repo prune`命令，:command:`repo abandon`命令更具破坏性，因为:command:`repo abandon`是对:command:`git branch -D`命令的封装。该命令非常危险，直接删除分支，请慎用。

用法：

```
repo abandon <branchname> [<project>...]
```

## 其他命令

- :command:`repo grep`

相当于对:command:`git grep`的封装，用于在项目文件中进行内容查找。

- :command:`repo smartsync`

相当于用-s参数执行:command:`repo sync`。

- :command:`repo forall`

迭代器，可以对repo管理的项目进行迭代。

- :command:`repo manifest`

显示:file:`manifest`文件内容。

- :command:`repo version`

显示repo的版本号。

- :command:`repo selfupdate`

用于repo自身的更新。如果提供--repo-upgraded参数，还会更新各个项目的钩子脚本。

## Repo命令的工作流

图25-1是repo的工作流，每一个代码贡献都起始于`:command:`repo start``创建本地工作分支，最终都以`:command:`repo upload``命令将代码补丁发布于代码审核服务器。

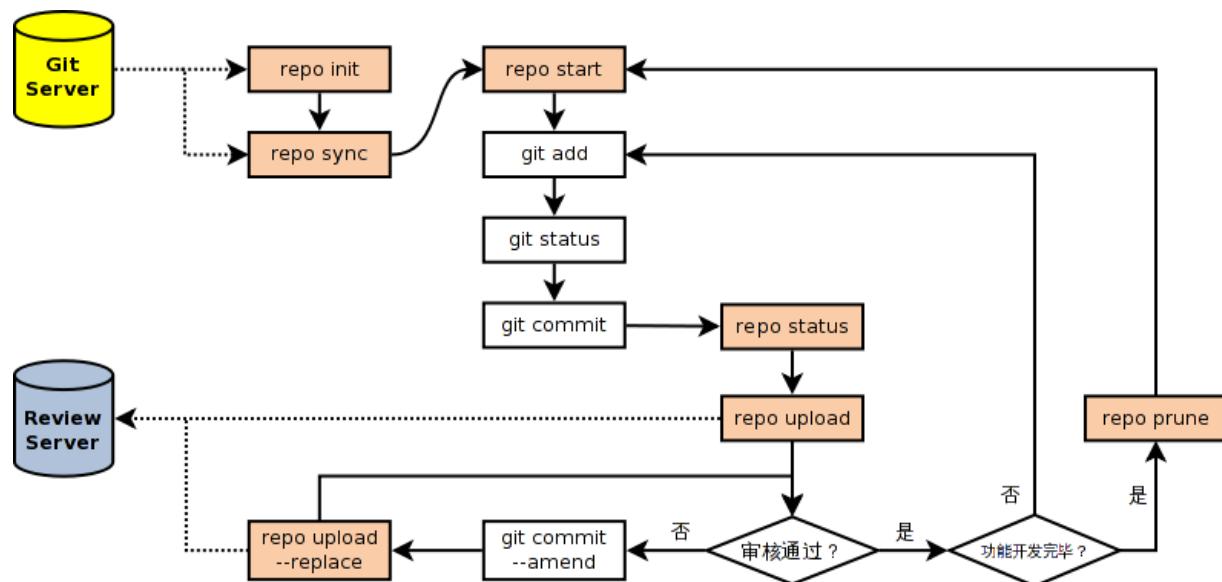


图25-1: repo工作流

## 好东西不能Android独享

通过前面的介绍能够体会到repo的精巧——repo巧妙的实现了多Git版本库的管理。因为repo使用了清单版本库，所以repo这一工具并没有被局限于Android项目，可以在任何项目中使用。下面就介绍三种repo的使用模式，将repo引入自己的（非Android）项目中，其中第三种repo使用模式是用作者改造后的repo实现脱离Gerrit服务器进行推送。

## Repo+Gerrit模式

Repo和Gerrit是Android代码管理的两大支柱。正如前面在repo工作流中介绍的，部分的repo命令从Git服务器读取，这个Git服务器可以是只读的版本库控制服务器，还有部分repo命令（`:command:`repo upload``、`:command:`repo download``）访问的则是代码审核服务器，其中`:command:`repo upload``命令还要向代码审核服务器进行`:command:`git push``操作。

在使用未经改动的repo来维护自己的项目（多个版本库组成）时，必须搭建Gerrit代码审核服务器。

搭建项目的版本控制系统环境的一般方法为：

- 用git-daemon或者http服务搭建Git服务器。具体搭建方法参见第5篇“搭建Git服务”

器”相关章节。

- 导入repo.git工具库。非必须，只是为了减少不必要的互联网操作。
- 还可以在内部http服务器维护一个定制的repo引导脚本。非必须。
- 建立Gerrit代码审核服务器。会在第5篇第32章“Gerrit代码审核服务器”中介绍Gerrit的安装和使用。
- 将相关的子项目代码库一一创建。
- 建立一个manifest.git清单库，其中remote元素的fetch属性指向只读Git服务器地址，review属性指向代码审核服务器地址。

示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <remote name="example"
        fetch="git://git.example.net/"
        review="review.example.net" />
    <default revision="master"
        remote="example" />
    ...

```

## Repo无审核模式

Gerrit代码审核服务器部署比较麻烦，更不要说因为Gerrit用户界面的学习和用户使用习惯的更改而带来的困难了。在一个固定的团队内部使用repo可能真的没有必要使用Gerrit，因为团队成员都应该熟悉Git的操作，团队成员的编程能力都可信，单元测试质量由提交者保证，集成测试由单独的测试团队进行，即团队拥有一套完整、成型的研发工作流，引入Gerrit并非必要。

脱离了Gerrit服务器，直接跟Git服务器打交道，repo可以工作么？是的，可以利用:command:`repo forall`迭代器实现多项目代码的PUSH，其中有如下关键点需要重点关注。

- :command:`repo start`命令创建本地分支时，需要使用和上游同样的分支名。

如果使用不同的分支名，上传时需要提供复杂的引用描述。下面的示例先通过:command:`repo manifest`命令确认上游清单库缺省的分支名为master，再使用该分支名(master)作为本地分支名执行:command:`repo start`。示例如下：

```
$ repo manifest -o - | grep default
<default remote="bj" revision="master"/>

$ repo start master --all
```

- 推送不能使用`:command:`repo upload``，而需要使用`:command:`git push``命令。

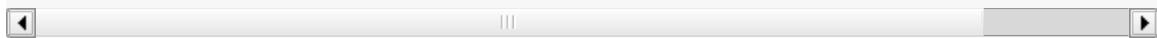
可以利用`:command:`repo forall``迭代器实现批命令方式执行。例如：

```
$ repo forall -c git push
```

- 如果清单库中的上游Git库地址用的是只读地址，需要为本地版本库一一更改上游版本库地址。

可以使用`forall`迭代器，批量为版本库设置`:command:`git push``时的版本库地址。下面的命令使用了环境变量`$REPO_PROJECT`是实现批量设置的关键。

```
$ repo forall -c \  
  'git remote set-url --push bj android@bj.ossxp.com:android/${REPO_PROJE
```



## 改进的Repo无审核模式

前面介绍的使用`:command:`repo forall``迭代器实现在无审核服务器情况下向上游推送提交，只是权宜之计，尤其是用`:command:`repo start``建立工作分支要求和上游一致，实在是有点强人所难。

我改造了repo，增加了两个新的命令`:command:`repo config``和`:command:`repo push``，让repo可以脱离Gerrit服务器直接向上游推送。代码托管在Github上：<http://github.com/ossxp-com/repo.git>。下面简单介绍一下如何使用改造之后的repo。

## 下载改造后的repo引导脚本

建议使用改造后的repo引导脚本替换原脚本，否则在执行`:command:`repo init``命令需要提供额外的`--no-repo-verify`参数，以及`--repo-url`和`--repo-branch`参数。

```
$ curl http://github.com/ossxp-com/repo/raw/master/repo > ~/bin/repo  
$ chmod a+x ~/bin/repo
```

## 用repo从Github上检出测试项目

如果安装了改造后的repo引导脚本，使用下面的命令初始化repo及清单库。

```
$ mkdir test
```

```
$ cd test  
$ repo init -u git://github.com/ossxp-com/manifest.git  
$ repo sync
```

如果用的是标准的（未经改造的）repo引导脚本，用下面的命令。

```
$ mkdir test  
$ cd test  
$ repo init --repo-url=git://github.com/ossxp-com/repo.git \  
--repo-branch=master --no-repo-verify \  
-u git://github.com/ossxp-com/manifest.git  
$ repo sync
```

当子项目代码全部同步完成后，执行make命令。可以看到各个子项目的版本以及清单库的版本。

```
$ make  
Version of test1: 1:0.2-dev  
Version of test2: 2:0.2  
Version of manifest: current
```

## 用`:command:`repo config``命令设置pushurl

现在如果进入到各个子项目目录，是无法成功执行`:command:`git push``命令的，因为上游Git库的地址是一个只读访问的URL，无法提供写服务。可以用新增的`:command:`repo config``命令设置当执行`:command:`git push``时的URL地址。

```
$ repo config repo.pushurl ssh://git@github.com/ossxp-com/
```

设置成功后，可以使用`:command:`repo config repo.pushurl``查看设置。

```
$ repo config repo.pushurl  
ssh://git@github.com/ossxp-com/
```

## 创建本地工作分支

使用下面的命令创建一个工作分支jiangxin。

```
$ repo start jiangxin --all
```

使用`:command:`repo branches``命令可以查看当前所有的子项目都属于jiangxin分支

```
$ repo branches
* jiangxin | in all projects
```

参照下面的方法修改test/test1子项目。对test/test2项目也作类似修改。

```
$ cd test/test1
$ echo "1:0.2-jiangxin" > version
$ git diff
diff --git a/version b/version
index 37c65f8..a58ac04 100644
--- a/version
+++ b/version
@@ -1 +1 @@
-1:0.2-dev
+1:0.2-jiangxin
$ repo status
# on branch jiangxin
project test/test1/                                branch jiangxin
-m      version
$ git add -u
$ git commit -m "0.2-dev -> 0.2-jiangxin"
```

执行:command:`make`命令，看看各个项目的改变。

```
$ make
Version of test1:    1:0.2-jiangxin
Version of test2:    2:0.2-jiangxin
Version of manifest: current
```

## PUSH到远程服务器

直接执行:command:`repo push`就可以将各个项目的改动进行推送。

```
$ repo push
```

如果有多个项目同时进行了改动，为了避免出错，会弹出编辑器显示因为包含改动而需要推送的项目列表。

```
# Uncomment the branches to upload:
#
# project test/test1/
# branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
```

```
#      4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
# branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#       86683ece 0.2-dev -> 0.2-jiangxin
```

每一行前面的井号是注释，会被忽略。将希望推送的分支前的注释去掉，就可以将该项目的分支执行推送动作。下面的操作中，把其中的两个分支的注释都去掉了，这两个项目当前分支的改动会推送到上游服务器。

```
# Uncomment the branches to upload:

#
# project test/test1/:
branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
#       4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#       86683ece 0.2-dev -> 0.2-jiangxin
```

保存退出（如果使用vi编辑器，输入`<ESC>:wq`执行保存退出）后，马上开始对选择的各个项目执行：`command: `git push``。

```
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 293 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test1.git
  27aee23..4f94123  jiangxin -> master
Counting objects: 5, done.
Writing objects: 100% (3/3), 261 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test2.git
  7f0841d..86683ec  jiangxin -> master

-----
[OK      ] test/test1/      jiangxin
[OK      ] test/test2/      jiangxin
```

从推送的命令输出可以看出来本地的工作分支`jiangxin`的改动被推送的远程服务器的`master`分支（本地工作分支跟踪的上游分支）。

再次执行:command:`repo push`，会显示没有项目需要推送。

```
$ repo push  
no branches ready for upload
```

## 在远程服务器创建新分支

如果想在服务器上创建一个新的分支，该如何操作呢？如下使用--new\_branch参数调用:command:`repo push`命令。

```
$ repo start feature1 --all  
$ repo push --new_branch
```

经过同样的编辑操作之后，自动调用:command:`git push`，在服务器上创建新分支feature1。

```
Total 0 (delta 0), reused 0 (delta 0)  
To ssh://git@github.com/ossxp-com/test1.git  
 * [new branch]      feature1 -> feature1  
Total 0 (delta 0), reused 0 (delta 0)  
To ssh://git@github.com/ossxp-com/test2.git  
 * [new branch]      feature1 -> feature1
```

```
-----  
[OK      ] test/test1/      feature1  
[OK      ] test/test2/      feature1
```

用:command:`git ls-remote`命令查看远程版本库的分支，会发现远程版本库中已经建立了新的分支。

```
$ git ls-remote git://github.com/ossxp-com/test1.git refs/heads/*  
4f9412399bf8093e880068477203351829a6b1fb      refs/heads/feature1  
4f9412399bf8093e880068477203351829a6b1fb      refs/heads/master  
b2b246b99ca504f141299ecdbadb23faf6918973      refs/heads/test-0.1
```

注意到feature1和master分支引用指向相同的SHA1哈希值，这是因为feature1分支是直接从master分支创建的。

## 通过不同的清单库版本，切换到不同分支

换用不同的清单库，需要建立新的工作区，并且在执行:command:`repo init`时，通过-b参数指定清单库的分支。

```
$ mkdir test-0.1  
$ cd test-0.1  
$ repo init -u git://github.com/ossxp-com/manifest.git -b test-0.1  
$ repo sync
```

当子项目代码全部同步完成后，执行：`make` 命令。可以看到各个子项目的版本以及清单库的版本不同于之前的输出。

```
$ make  
Version of test1: 1:0.1.4  
Version of test2: 2:0.1.3-dev  
Version of manifest: current-2-g12f9080
```

可以用：`repo manifest` 命令来查看清单库。

```
$ repo manifest -o -  
<?xml version="1.0" encoding="UTF-8"?>  
<manifest>  
  <remote fetch="git://github.com/ossxp-com/" name="github"/>  
  
  <default remote="github" revision="refs/heads/test-0.1"/>  
  
  <project name="test1" path="test/test1">  
    <copyfile dest="Makefile" src="root.mk"/>  
  </project>  
  <project name="test2" path="test/test2"/>  
</manifest>
```

仔细看上面的清单文件，可以注意到缺省的版本指向到`refs/heads/test-0.1`引用所指向的分支`test-0.1`。

如果在子项目中修改、提交，然后使用：`repo push` 会将改动推送的远程版本库的`test-0.1`分支中。

## 切换到清单库里程碑版本

执行如下命令，可以查看清单库包含的里程碑版本：

```
$ git ls-remote --tags git://github.com/ossxp-com/manifest.git  
43e5783a58b46e97270785aa967f09046734c6ab      refs/tags/current  
3a6a6da36840e716a14d52252e7b40e6ba6cbdea      refs/tags/current^{}  
4735d32613eb50a6c3472cc8087ebf79cc46e0c0      refs/tags/v0.1  
fb1a1b7302a893092ce8b356e83170eee5863f43      refs/tags/v0.1^{}
```

b23884d9964660c8dd34b343151aaf968a744400	refs/tags/v0.1.1
9c4c287069e29d21502472acac34f28896d7b5cc	refs/tags/v0.1.1^{}{}
127d9789cd4312ed279a7fa683c43eec73d2b28b	refs/tags/v0.1.2
47aaa83866f6d910a118a9a19c2ac3a2a5819b3e	refs/tags/v0.1.2^{}{}
af3abb7ed0a9ef7063e9d814510c527287c92ef6	refs/tags/v0.1.3
99c69bcfd7e2e7737cc62a7d95f39c6b9ffaf31a	refs/tags/v0.1.3^{}{}

可以从任意里程碑版本的清单库初始化整个项目。

```
$ mkdir v0.1.2
$ cd v0.1.2
$ repo init -u git://github.com/ossxp-com/manifest.git -b refs/tags/v0.1.2
$ repo sync
```

当子项目代码全部同步完成后，执行：`command: `make`` 命令。可以看到各个子项目的版本以及清单库的版本不同于之前的输出。

```
$ make
Version of test1: 1:0.1.2
Version of test2: 2:0.1.2
Version of manifest: v0.1.2
```

来源： <https://github.com/gotgit/gotgit/blob/master/04-git-model/060-android-model.rst>

# Git和SVN协同模型

在本篇的最后，将会在另外的一个角度上看Git版本库的协同。不是不同的用户在使用Git版本库时如何协同，也不是一个项目包含多个Git版本库时如何协同，而是当版本控制系统不是Git（如Subversion）时，如何能够继续使用Git的方式进行操作。

Subversion会在商业软件开发中占有一席之地，只要商业软件公司严格封闭源代码的策略不改变。对于熟悉了Git的用户，一定会对Subversion的那种一旦脱离网络、脱离服务器便寸步难行的工作模式厌烦透顶。实际上对Subversion的集中式版本控制的不满和改进在Git诞生之前就发生了，这就是SVK。

在2003年（Git诞生的前两年），台湾的高嘉良就开发了SVK，用分布式版本控制的方法操作SVN。其设计思想非常朴素，既然SVN的用户可以看到有访问权限数据的全部历史，那么也应该能够依据历史重建一个本地的SVN版本库，这样很多SVN操作都可以通过本地的SVN进行，从而脱离网络。当对本地版本库的修改感到满意后，通过本地SVN版本和服务器SVN版本库之间的双向同步，将改动归并到服务器上。这种工作方式真的非常酷。

不必为SVK的文档缺乏以及不再维护而感到惋惜，因为有更强的工具登场了，这就是git-svn。git-svn是Git软件包的一部分，用Perl语言开发。它的工作原理是：

- 将Subversion版本库在本地转换为一个Git库。
- 转换可以基于Subversion的某个目录，或者基于某个分支，或者整个Subversion代码库的所有分支和里程碑。
- 远程的Subversion版本库可以和本地的Git双向同步。Git本地库修改推送到远程Subversion版本库，反之亦然。

git-svn作为Git软件包的一部分，当Git从源码包进行安装时会默认安装，提供：`command: `git svn`` 命令。而几乎所有的Linux发行版都将git-svn作为一个独立的软件单独发布，因此需要单独安装。例如Debian和Ubuntu运行下面命令安装git-svn。

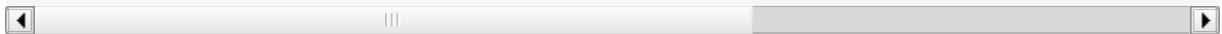
```
$ sudo aptitude install git-svn
```

将git-svn独立安装是因为git-svn软件包有着特殊的依赖，即依赖Subversion的Perl语言绑定接口，Debian/Ubuntu上由`libsvn-perl`软件包提供。

当git-svn正确安装后，就可以使用：`command: `git svn`` 命令了。但如果在执行：`command: `git svn --version`` 时遇到下面的错误，则说明Subversion的Perl语言绑定没有正确安装。

```
$ git svn --version
```

```
Can't locate loadable object for module SVN::_Core in @INC (@INC contains: ..  
BEGIN failed--compilation aborted at /usr/lib/perl5/SVN/Core.pm line 5.  
Compilation failed in require at /usr/lib/git-core/git-svn line 41.
```



遇到上面的情况，需要检查本机是否正确安装了Subversion以及Subversion的Perl语言绑定。

为了便于对git-svn的介绍和演示，需要有一个Subversion版本库，并且需要有提交权限以便演示用Git向Subversion进行提交。最好的办法是在本地创建一个Subversion版本库。

```
$ svnadmin create /path/to/svn/repos/demo  
  
$ svn co file:///path/to/svn/repos/demo svndemo  
取出版本 0  
  
$ cd svndemo  
  
$ mkdir trunk tags branches  
$ svn add *  
A           branches  
A           tags  
A           trunk  
  
$ svn ci -m "initialized."  
增加           branches  
增加           tags  
增加           trunk  
  
提交后的版本为 1。
```

再向Subversion开发主线trunk中添加些数据。

```
$ echo hello > trunk/README  
$ svn add trunk/README  
A           trunk/README  
$ svn ci -m "hello"  
增加           trunk/README  
传输文件数据。  
提交后的版本为 2。
```

建立分支：

```
$ svn up  
$ svn cp trunk branches/demo-1.0  
A           branches/demo-1.0
```

```
$ svn ci -m "new branch: demo-1.0"
增加           branches/demo-1.0
```

提交后的版本为 3。

建立里程碑：

```
$ svn cp -m "new tag: v1.0" trunk file:///path/to/svn/repos/demo/tags/v1.0
```

提交后的版本为 4。

## 使用git-svn的一般流程

使用git-svn的一般流程参见图26-1。

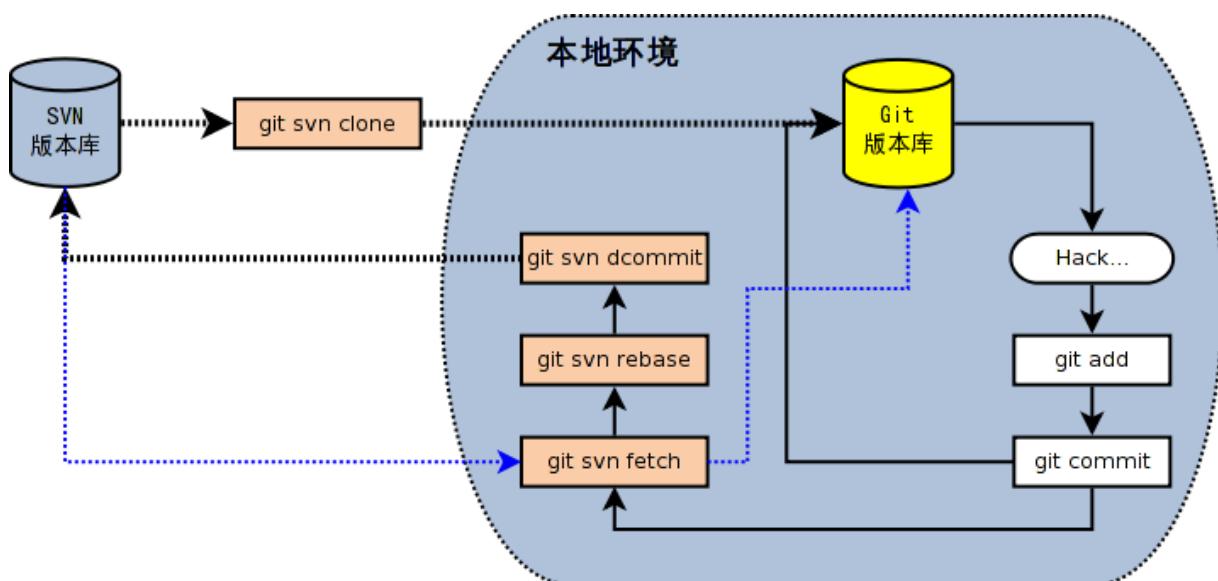
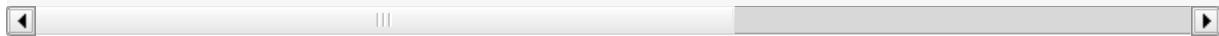


图26-1：git-svn工作流

首先用：`git svn clone` 命令对Subversion进行克隆，创建一个包含git-svn扩展的本地Git库。在下面的示例中，使用Subversion的本地协议（file:///）来访问之前创立的Subversion示例版本库，实际上git-svn可以使用任何Subversion可用的协议，并可以对远程版本库进行操作。

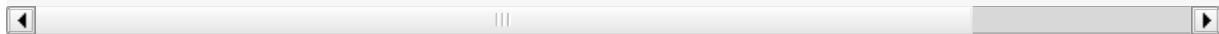
```
$ git svn clone -s file:///path/to/svn/repos/demo git-svn-demo
Initialized empty Git repository in /path/to/my/workspace/git-svn-demo/.git/
r1 = 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4 (refs/remotes/trunk)
      A      README
r2 = 1863f91b45def159a3ed2c4c4c9428c25213f956 (refs/remotes/trunk)
Found possible branch point: file:///path/to/svn/repos/demo/trunk => file:///
Found branch parent: (refs/remotes/demo-1.0) 1863f91b45def159a3ed2c4c4c9428c2
Following parent with do_switch
```

```
Successfully followed parent  
r3 = 1adcd5526976fe2a796d932ff92d6c41b7eedcc4 (refs/remotes/demo-1.0)  
Found possible branch point: file:///path/to svn/repos/demo/trunk => file:///  
Found branch parent: (refs/remotes/tags/v1.0) 1863f91b45def159a3ed2c4c4c9428c  
Following parent with do_switch  
Successfully followed parent  
r4 = c12aa40c494b495a846e73ab5a3c787ca1ad81e9 (refs/remotes/tags/v1.0)  
Checked out HEAD:  
    file:///path/to svn/repos/demo/trunk r2
```



从上面的输出可以看出，当执行了`:command:`git svn clone``之后，在本地工作目录创建了一个Git库（git-svn-demo），并将Subversion的每一个提交都转换为Git库中的提交。进入`:file:`git-svn-demo``目录，看看用git-svn克隆出来的版本库。

```
$ cd git-svn-demo/  
$ git branch -a  
* master  
  remotes/demo-1.0  
  remotes/tags/v1.0  
  remotes/trunk  
$ git log  
commit 1863f91b45def159a3ed2c4c4c9428c25213f956  
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>  
Date:   Mon Nov 1 05:49:41 2010 +0000  
  
hello  
  
git-svn-id: file:///path/to svn/repos/demo/trunk@2 f79726c4-f016-41bd-acd  
  
commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4  
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>  
Date:   Mon Nov 1 05:47:03 2010 +0000  
  
initialized.  
  
git-svn-id: file:///path/to svn/repos/demo/trunk@1 f79726c4-f016-41bd-acd
```



看到Subversion版本库的分支和里程碑都被克隆出来，并保存在`refs/remotes`下的引用中。在`:command:`git log``的输出中，可以看到Subversion的提交的确被转换为Git的提交。

下面就可以在Git库中进行修改，并在本地提交（用`:command:`git commit``命令）。

```
$ cat README  
hello
```

```
$ echo "I am fine." >> README
$ git add -u
$ git commit -m "my hack 1."
[master 55e5fd7] my hack 1.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ echo "Thank you." >> README
$ git add -u
$ git commit -m "my hack 2."
[master f1e00b5] my hack 2.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

对工作区中的:`:file:`README``文件修改了两次，并进行了本地的提交。查看这时的提交日志，会发现最新两个只在本地Subversion版本库的提交和之前Subversion 中的提交的不同。区别在于最新在Git中的提交没有用`git-svn-id:` 标签标记的行。

```
$ git log
commit f1e00b52209f6522dd8135d27e86370de552a7b6
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Nov 4 15:05:47 2010 +0800

my hack 2.

commit 55e5fd794e6208703aa999004ec2e422b3673ade
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Nov 4 15:05:32 2010 +0800

my hack 1.

commit 1863f91b45def159a3ed2c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

hello

git-svn-id: file:///path/to svn/repos/demo/trunk@2 f79726c4-f016-41bd-acd

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

initialized.

git-svn-id: file:///path/to svn/repos/demo/trunk@1 f79726c4-f016-41bd-acd
```

现在就可以向Subversion服务器推送改动了。但真实的环境中，往往在向服务器推 送时，已经有其他用户已经在服务器上进行了提交，而且往往更糟的是，先于我们的提交会造成

我们的提交冲突！现在就人为的制造一个冲突：使用`:command:` svn``命令在Subversion版本库中执行一次提交。

```
$ svn checkout file:///path/to/svn/repos/demo/trunk demo
A    demo/README
取出版本 4。
$ cd demo/
$ cat README
hello
$ echo "HELLO." > README
$ svn commit -m "hello -> HELLO."
正在发送      README
传输文件数据.
提交后的版本为 5.
```

好的，已经模拟了一个用户先于我们更改了Subversion版本库。现在回到用git-svn克隆的本地版本库，执行`:command:` git svn dcommit``操作，将Git中的提交推送的Subversion版本库中。

```
$ git svn dcommit
Committing to file:///path/to/svn/repos/demo/trunk ...
事务过时：文件 “/trunk/README” 已经过时 at /usr/lib/git-core/git-svn line 572
```

显然，由于Subversion版本库中包含了新的提交，导致执行`:command:` git svn dcommit``出错。这时需执行`:command:` git svn fetch``命令，以从Subversion版本库获取更新。

```
$ git svn fetch
M      README
r5 = fae6dab863ed2152f71bcb2348d476d47194fdd4 (refs/remotes/trunk)
$ git st
# On branch master
nothing to commit (working directory clean)
```

当获取了新的Subversion提交之后，需要执行`:command:` git svn rebase``将Git中未推送的提交通过变基（rebase）形成包含Subversion最新提交的线性提交。这是因为Subversion的提交都是线性的。

```
$ git svn rebase
First, rewinding head to replay your work on top of it...
Applying: my hack 1.
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
```

```
Auto-merging README
CONFLICT (content): Merge conflict in README
Failed to merge in the changes.
Patch failed at 0001 my hack 1.

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".

rebase refs/remotes/trunk: command returned error: 1
```

果不其然，变基时发生了冲突，这是因为Subversion中他人的修改和我们在Git库中的修改都改动了同一个文件，并且改动了相近的行。下面按照:command:`git rebase`冲突解决的一般步骤进行，直到成功完成变基操作。

先编辑:file:`README`文件，以解决冲突。

```
$ git status
# Not currently on any branch.
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      README
#
no changes added to commit (use "git add" and/or "git commit -a")

$ vi README
```

处于冲突状态的:file:`REAEUME`文件内容。

```
<<<<< HEAD
HELLO.
=====
hello
I am fine.
>>>>> my hack 1.
```

下面是修改后的内容。保存退出。

```
HELLO.
I am fine.
```

执行:command:`git add`命令解决冲突

```
$ git add README
```

调用:`git rebase --continue`完成变基操作。

```
$ git rebase --continue
Applying: my hack 1.
Applying: my hack 2.
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging README
```

看看变基之后的Git库日志:

```
$ git log
commit e382f2e99eca07bc3a92ece89f80a7a5457acfd8
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Thu Nov 4 15:05:47 2010 +0800

    my hack 2.

commit 6e7e0c7dccf5a072404a28f06ce0c83d77988b0b
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Thu Nov 4 15:05:32 2010 +0800

    my hack 1.

commit fae6dab863ed2152f71bcb2348d476d47194fdd4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:15:58 2010 +0000

    hello -> HELLO.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@5 f79726c4-f016-41bd-acd

commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

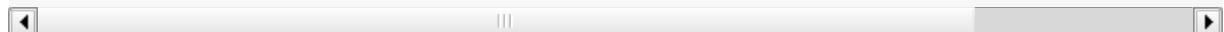
    hello

    git-svn-id: file:///path/to/svn/repos/demo/trunk@2 f79726c4-f016-41bd-acd

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.
```

```
git-svn-id: file:///path/to svn/repos/demo/trunk@1 f79726c4-f016-41bd-acd
```



当变基操作成功完成后，再执行：`git svn dcommit` 向Subversion推送Git库中的两个新提交。

```
$ git svn dcommit
Committing to file:///path/to svn/repos/demo/trunk ...
M README
Committed r6
M README
r6 = d0eb86bdfad4720e0a24edc49ec2b52e50473e83 (refs/remotes/trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
Unstaged changes after reset:
M README
M README
Committed r7
M README
r7 = 69f4aa56eb96230aedd7c643f65d03b618ccc9e5 (refs/remotes/trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

推送之后本地Git库中最新的两个提交的提交说明中也嵌入了`git-svn-id`:标签。这个标签的作用非常重要，在下一节会予以介绍。

```
$ git log -2
commit 69f4aa56eb96230aedd7c643f65d03b618ccc9e5
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:56:38 2010 +0000

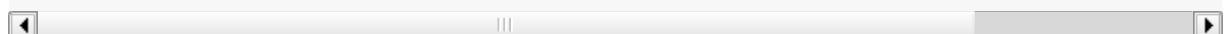
my hack 2.

git-svn-id: file:///path/to svn/repos/demo/trunk@7 f79726c4-f016-41bd-acd

commit d0eb86bdfad4720e0a24edc49ec2b52e50473e83
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:56:37 2010 +0000

my hack 1.

git-svn-id: file:///path/to svn/repos/demo/trunk@6 f79726c4-f016-41bd-acd
```



通过上面对git-svn的工作流程的介绍，相信读者已经能够体会到git-svn的强大。那么git-svn是怎么做到的呢？

git-svn只是在本地Git库中增加了一些附加的设置，特殊的引用，和引入附加的可重建的数据库实现对Subversion版本库的跟踪。

## Git库配置文件的扩展及分支映射

当执行:command:`git svn init`或者:command:`git svn clone`时，git-svn会通过在Git库的配置文件中增加一个小节，记录Subversion版本库的URL，以及Subversion分支/里程碑和本地Git库的引用之间的对应关系。

例如：当执行:command:`git svn clone -s file:///path/to svn/repos/demo`指令时，会在创建的本地Git库的配置文件:file:`.git/config`中引入下面新的配置：

```
[svn-remote "svn"]
  url = file:///path/to svn/repos/demo
  fetch = trunk:refs/remotes/trunk
  branches = branches/*:refs/remotes/*
  tags = tags/*:refs/remotes/tags/*
```

缺省svn-remote的名字为“svn”，所以新增的配置小节的名字为: [svn-remote "svn"]。在git-svn克隆时，可以使用--remote参数设置不同的svn-remote名称，但是并不建议使用。因为一旦使用--remote参数更改svn-remote名称，必须在git-svn的其他命令中都使用--remote参数，否则报告[svn-remote "svn"]配置小节未找到。

在该小节中主要的配置有：

- url = <URL>

设置Subversion版本库的地址

- fetch = <svn-path>:<git-refspec>

Subversion的开发主线和Git版本库引用的对应关系。

在上例中Subversion的:file:`trunk`目录对应于Git的refs/remotes/trunk引用。

- branches = <svn-path>:<git-refspec>

Subversion的开发分支和Git版本库引用的对应关系。可以包含多条branches的设置，以便将分散在不同目录下的分支汇总。

在上例中Subversion的`:file:`branches``子目录下一级子目录`(:file:`branches/\*`)`所代表的分支在Git的`refs/remotes/`名字空间下建立引用。

- `tags = <svn-path>:<git-refspec>`

Subversion的里程碑和Git版本库引用的对应关系。可以包含多条tags的设置，以便将分散在不同目录下的里程碑汇总。

在上例中Subversion的`tags`子目录下一级子目录`(tags/*)`所代表的里程碑在Git的`refs/remotes/tags`名字空间下建立引用。

可以看到Subversion的主线和分支缺省都直接被映射到`refs/remotes/`下。如`trunk`主线对应于`refs/remotes/trunk`，分支`demo-1.0`对应于`refs/remotes/demo-1.0`。Subversion的里程碑因为有可能和分支同名，因此被映射到`refs/remotes/tags/`之下，这样就里程碑和分支的映射放到不同目录下，不会互相影响。

## Git工作分支和Subversion如何对应？

Git缺省工作的分支是`master`，而看到上例中的Subversion主线在Git中对应的远程分支为`refs/remotes/trunk`。那么在执行`:command:`git svn rebase``时，`git-svn`是如何知道当前的HEAD对应的分支基于哪个Subversion跟踪分支进行变基？还有就是执行`:command:`git svn dcommit``时，当前的工作分支应该将改动推送到哪个Subversion分支中去呢？

很自然的会按照Git的方式进行思考，期望在`:file:`.git/config``配置文件中找到类似`[branch master]`之类的配置小节。实际上，在`git-svn`的Git库的配置文件中可能根本就不存在`[branch ...]`小节。那么`git-svn`是如何确定当前Git工作分支和远程Subversion版本库的分支建立对应的呢？

其实奥秘就在Git的日志中。当在工作区执行`:command:`git log``时，会看到包含`git-svn-id`标识的特殊日志。发现的最近的一个`git-svn-id`标识会确定当前分支提交的Subversion分支。

下面继续上一节的示例，先切换到分支，并将提交推送到Subversion的分支`demo-1.0`中。

首先在Git库中会看到有一个对应于Subversion分支的远程分支和一个对应于Subversion里程碑的远程引用。

```
$ git branch -r
  demo-1.0
  tags/v1.0
  trunk
```

然后基于远程分支demo-1.0建立本地工作分支myhack。

```
$ git checkout -b myhack refs/remotes/demo-1.0
Switched to a new branch 'myhack'
$ git branch
  master
* myhack
```

在myhack分支做一些改动，并提交。

```
$ echo "Git" >> README
$ git add -u
$ git commit -m "say hello to Git."
[myhack d391fd7] say hello to Git.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

下面看看Git的提交日志。

```
$ git log --first-parent
commit d391fd75c33f62307c3add1498987fa3eb70238e
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Fri Nov 5 09:40:21 2010 +0800

    say hello to Git.

commit 1adcd5526976fe2a796d932ff92d6c41b7eedcc4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:54:19 2010 +0000

    new branch: demo-1.0

    git-svn-id: file:///path/to svn/repos/demo/branches/demo-1.0@3 f79726c4-f

commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

    hello

    git-svn-id: file:///path/to svn/repos/demo/trunk@2 f79726c4-f016-41bd-acd

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.
```

```
git-svn-id: file:///path/to svn/repos/demo/trunk@1 f79726c4-f016-41bd-acd
```



III



看到了上述Git日志中出现的第一个git-svn-id:标识的内容为:

```
git-svn-id: file:///path/to svn/repos/demo/branches/demo-1.0@3 f79726c4-f016-
```



III



这就是说，当需要将Git提交推送给Subversion服务器时，需要推送到地址: file:///path/to svn/repos/demo/branches/demo-1.0。

执行:command:`git svn dcommit`，果然是推送到Subversion的demo-1.0分支。

```
$ git svn dcommit
Committing to file:///path/to svn/repos/demo/branches/demo-1.0 ...
M README
Committed r8
M README
r8 = a8b32d1b533d308bef59101c1f2c9a16baf91e48 (refs/remotes/demo-1.0)
No changes between current HEAD and refs/remotes/demo-1.0
Resetting to the latest refs/remotes/demo-1.0
```

## 其他辅助文件

在Git版本库中，git-svn在:file:`.git/svn`目录下保存了一些索引文件，便于git-svn更加快速的执行。

文件:file:`.git/svn/.metadata`文件是类似于:file:`.git/config`文件一样的INI文件，其中保存了版本库的URL，版本库UUID，分支和里程碑的最后获取的版本号等。

```
; This file is used internally by git-svn
; You should not have to edit it
[svn-remote "svn"]
reposRoot = file:///path/to svn/repos/demo
uuid = f79726c4-f016-41bd-acd5-6c9acb7664b2
branches-maxRev = 8
tags-maxRev = 8
```

在:file:`.git/svn/refs/remotes`目录下以各个分支和里程碑为名的各个子目录下都包含一个:file:`.rev\_map.<SVN-UUID>`的索引文件，这个文件用于记录Subversion的提交ID和Git的提交ID的映射。

目录:`.git/svn`的辅助文件由git-svn维护，不要手工修改否则会造成git-svn不能正常工作。

## 多样的git-svn克隆模式

在前面的git-svn示例中，使用`git svn clone`命令完成对远程版本库的克隆，实际上`git svn clone`相当于两条命令，即：

```
git svn clone = git svn init + git svn fetch
```

命令`git svn init`只完成两个工作。一个是在本地建立一个空的Git版本库，另外是修改`.git/config`文件，在其中建立Subversion和Git之间的分支映射关系。在实际使用中，我更喜欢使用`git svn init`命令，因为这样可以对Subversion和Git的分支映射进行手工修改。该命令的用法是：

用法：`git svn init [options] <subversion-url> [local-dir]`

可选的主要参数有：

```
--stdlayout, -s  
--trunk, -T <arg>  
--branches, --b=s@  
--tags, --t=s@  
--config-dir <arg>  
--ignore-paths <arg>  
--prefix <arg>  
--username <arg>
```

其中`--username`参数用于设定远程Subversion服务器认证时提供的用户名。参数`--prefix`用于设置在Git的`refs/remotes`下保存引用时使用的前缀。参数`--ignore-paths`后面跟一个正则表达式定义忽略的文件列表，这些文件将不予克隆。

最常用的参数是`-s`。该参数和前面演示的`git clone`命令中的一样，即使用标准的分支/里程碑部署方式克隆Subversion版本库。Subversion约定俗成使用`trunk`目录跟踪主线的开发，使用`branches`目录保存各个分支，使用`tags`目录来记录里程碑。

即命令：

```
$ git svn init -s file:///path/to/svn/repos/demo
```

和下面的命令等效：

```
$ git svn init -T trunk -b branches -t tags file:///path/to/svn/repos/demo
```

有的Subversion版本库的分支可能分散于不同的目录下，例如有的位于`:file:`branches``目录，有的位于`:file:`sandbox``目录，则可以用下面命令：

```
$ git svn init -T trunk -b branches -b sandbox -t tags file:///path/to/svn/re  
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
```

查看本地克隆版本库的配置文件：

```
$ cat git-svn-test/.git/config  
[core]  
    repositoryformatversion = 0  
    filemode = true  
    bare = false  
    logallrefupdates = true  
[svn-remote "svn"]  
    url = file:///path/to/svn/repos/demo  
    fetch = trunk:refs/remotes/trunk  
    branches = branches/*:refs/remotes/*  
    branches = sandbox/*:refs/remotes/*  
    tags = tags/*:refs/remotes/tags/*
```

可以看到在`[svn-remote "svn"]`小节中包含了两条`branches`配置，这就会实现将Subversion分散于不同目录的分支都克隆出来。如果担心Subversion的`:file:`branches``目录和`:file:`sandbox``目录下出现同名的分支导致在Git库的`refs/remotes/`下造成覆盖，可以在版本库尚未执行`:command:`git svn fetch``之前编辑`:file:`.git/config``文件，避免可能出现的覆盖。例如编辑后的`[svn-remote "svn"]`配置小节：

```
[svn-remote "svn"]  
    url = file:///path/to/svn/repos/demo  
    fetch = trunk:refs/remotes/trunk  
    branches = branches/*:refs/remotes/branches/*  
    branches = sandbox/*:refs/remotes/sandbox/*  
    tags = tags/*:refs/remotes/tags/*
```

如果项目的分支或里程碑非常多，也可以修改`[svn-remote "svn"]`配置小节中的版本号通配符，使得只获取部分分支或里程碑。例如下面的配置小节：

```
[svn-remote "svn"]
url = http://server.org/svn
fetch = trunk/src:refs/remotes/trunk
branches = branches/{red,green}/src:refs/remotes/branches/*
tags = tags/{1.0,2.0}/src:refs/remotes/tags/*
```

如果只关心Subversion的某个分支甚至某个子目录，而不关心其他分支或目录，那就更简单了，不带参数的执行:command:`git svn init`针对Subversion的某个具体路径执行初始化就可以了。

```
$ git svn init file:///path/to/svn/repos/demo/trunk
```

有的情况下，版本库太大，而且对历史不感兴趣，可以只克隆最近的部分提交。这时可以通过:command:`git svn fetch`命令的-r参数实现部分提交的克隆。

```
$ git svn init file:///path/to/svn/repos/demo/trunk git-svn-test
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
$ cd git-svn-test
$ git svn fetch -r 6:HEAD
A      README
r6 = 053b641b7edd2f1a59a007f27862d98fe5bcda57 (refs/remotes/git-svn)
M      README
r7 = 75c17ea61d8527334855a51e65ac98c981f545d7 (refs/remotes/git-svn)
Checked out HEAD:
file:///path/to/svn/repos/demo/trunk r7
```

当然也可以使用:command:`git svn clone`命令实现部分克隆：

```
$ git svn clone -r 6:HEAD \
    file:///path/to/svn/repos/demo/trunk git-svn-test
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
A      README
r6 = 053b641b7edd2f1a59a007f27862d98fe5bcda57 (refs/remotes/git-svn)
M      README
r7 = 75c17ea61d8527334855a51e65ac98c981f545d7 (refs/remotes/git-svn)
Checked out HEAD:
file:///path/to/svn/repos/demo/trunk r7
```

## 共享git-svn的克隆库

当一个Subversion版本库非常庞大而且和不在同一个局域网内，执行:command:`git svn

`clone``可能需要花费很多时间。为了避免因重复执行`:command:`git svn clone``导致时间上的浪费，可以将一个已经使用git-svn克隆出来的Git库共享，其他人基于此Git进行克隆，然后再用特殊的方法重建和Subversion的关联。还记得之前提到过，`:file:`.git/svn``目录下的辅助文件可以重建么？

例如通过工作区中已经存在的git-svn-demo执行克隆。

```
$ git clone git-svn-demo myclone
Initialized empty Git repository in /path/to/my/workspace/myclone/.git/
```

进入新的克隆中，会发现新的克隆缺乏跟踪Subversion分支的引用，即`refs/remotes/trunk`等。

```
$ cd myclone/
$ git br -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/myhack
```

这是因为Git克隆缺省不复制远程版本库的`refs/remotes/`下的引用。可以用`:command:`git fetch``命令获取`refs/remotes`的引用。

```
$ git fetch origin refs/remotes/*:refs/remotes/*
From /path/to/my/workspace/git-svn-demo
 * [new branch]      demo-1.0    -> demo-1.0
 * [new branch]      tags/v1.0   -> tags/v1.0
 * [new branch]      trunk       -> trunk
```

现在这个从git-svn库中克隆出来的版本库已经有了相同的Subversion跟踪分支，但是`:file:`.git/config``文件还缺乏相应的`[svn-remote "svn"]`配置。可以通过使用同样的`:command:`git svn init``命令实现。

```
$ pwd
/path/to/my/workspace/myclone

$ git svn init -s file:///path/to/svn/repos/demo

$ git config --get-regexp 'svn-remote.*'
svn-remote.svn.url file:///path/to/svn/repos/demo
svn-remote.svn.fetch trunk:refs/remotes/trunk
svn-remote.svn.branches branches/*:refs/remotes/*
svn-remote.svn.tags tags/*:refs/remotes/tags/*
```

但是克隆版本库相比用git-svn克隆的版本库还缺乏:`:file:`.git/svn``下的辅助文件。实际上可以用:`:command:`git svn rebase``命令重建，同时这条命令也可以变基到Subversion相应分支的最新提交上。

```
$ git svn rebase
Rebuilding .git/svn/refs/remotes/trunk/.rev_map.f79726c4-f016-41bd-acd5-6c9ac
r1 = 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
r2 = 1863f91b45def159a3ed2c4c4c9428c25213f956
r5 = fae6dab863ed2152f71bcb2348d476d47194fdd4
r6 = d0eb86bdfad4720e0a24edc49ec2b52e50473e83
r7 = 69f4aa56eb96230aedd7c643f65d03b618ccc9e5
Done rebuilding .git/svn/refs/remotes/trunk/.rev_map.f79726c4-f016-41bd-acd5-
Current branch master is up to date.
```

如果执行:`:command:`git svn fetch``则会对所有的分支都进行重建。

```
$ git svn fetch
Rebuilding .git/svn/refs/remotes/demo-1.0/.rev_map.f79726c4-f016-41bd-acd5-6c
r3 = 1acdcd5526976fe2a796d932ff92d6c41b7eedcc4
r8 = a8b32d1b533d308bef59101c1f2c9a16baf91e48
Done rebuilding .git/svn/refs/remotes/demo-1.0/.rev_map.f79726c4-f016-41bd-ac
Rebuilding .git/svn/refs/remotes/tags/v1.0/.rev_map.f79726c4-f016-41bd-acd5-6
r4 = c12aa40c494b495a846e73ab5a3c787ca1ad81e9
Done rebuilding .git/svn/refs/remotes/tags/v1.0/.rev_map.f79726c4-f016-41bd-a
```

至此，从git-svn克隆库二次克隆的Git库，已经和原生的git-svn库一样使用git-svn命令了。

## git-svn的局限

Subversion和Git的分支实现有着巨大的不同。Subversion的分支和里程碑，是用轻量级拷贝实现的，虽然创建分支和里程碑的速度也很快，但是很难维护。即使Subversion在1.5之后引入了`svn:mergeinfo`属性对合并过程进行标记，但是也不可能让Subversion的分支逻辑更清晰。git-svn无须利用`svn:mergeinfo`属性也可实现对Subversion合并的追踪，在合并的时候也不会对`svn:mergeinfo`属性进行更改，因此在使用git-svn操作时，如果在不同分支间进行合并，会导致 Subversion的`svn:mergeinfo`属性没有相应的更新，导致Subversion用户进行合并时因为重复合并导致冲突。

简而言之，在使用git-svn时尽量不要在不同的分支之间进行合并，而是尽量在一个分支下线性的提交。这种线性的提交会很好的推送到Subversion服务器中。

如果真的需要在不同的Subversion分支之间合并，尽量使用Subversion的客户端（svn 1.5 版本或以上）执行，因为这样可以正确的记录svn:mergeinfo属性。当Subversion完成分支合并后，在git-svn的克隆库中执行：`git svn rebase` 命令获取最新的Subversion提交并变基到相应的跟踪分支中。

来源： <https://github.com/gotgit/gotgit/blob/master/04-git-model/070-git-svn.rst>

## 使用HTTP协议

HTTP协议是版本控制工具普遍采用的协议，具有安全（HTTPS），方便（跨越防火墙）等优点。Git在 1.6.6版本之前对HTTP协议支持有限，是哑协议，访问效率低，但是在1.6.6之后，通过一个CGI实现了智能的HTTP协议支持。

### 哑传输协议

所谓的哑传输协议（dumb protocol）就是在Git服务器和Git客户端的会话过程中只使用了相关协议提供的基本传输功能，而未针对Git的传输特点进行相关优化设计。采用哑协议，Git客户端和服务器间的通讯缺乏效率，用户使用时最直观的体验之一就是在操作过程没有进度提示，应用会一直停在那里直到整个通讯过程处理完毕。

但是哑传输协议配置起来却非常的简单。通过哑HTTP协议提供Git服务，基本上就是把包含Git版本库的目录通过HTTP服务器共享出来。下面的Apache配置片段就将整个目录/path/to/repos共享，然后就可以通过地址http://<服务器名称>/git/<版本库名称>访问该共享目录下所有的Git版本库。

```
Alias /git /path/to/repos

<Directory /path/to/repos>
    Options FollowSymLinks
    AllowOverride None
    Order Allow,Deny
    Allow from all
</Directory>
```

如果以为直接把现存的Git版本库移动到该目录下就可以直接用HTTP协议访问，可就大错特错了。因为哑协议下的Git版本库需要包含两个特殊的文件，并且这两个文件要随Git版本库更新。例如将一个包含提交数据的裸版本库复制到路径/path/to/repos/myrepos.git中，然后使用下面命令克隆该版本库（服务器名称为server），可能会看到如下错误：

```
$ git clone http://server/git/myrepos.git
正克隆到 'myrepos'...
fatal: repository 'http://server/git/myrepos.git/' not found
```

这时，您仅需在Git版本库目录下执行git update-server-info命令即可在Git版本库中创建哑协议需要的相关文件。

```
$ git update-server-info
```

然后该Git版本库即可正常访问了。如下：

```
$ git clone http://server/git/myrepos.git
正克隆到 'myrepos'...
检查连接... 完成。
```

从上面的介绍中可以看出在使用哑HTTP协议时，服务器端运行git update-server-info的重要性。运行该命令会产生或更新两个文件：

- 文件:`file:.git/info/refs`：该文件中包含了版本库中所有的引用列表，每一个引用都指向正确的SHA1哈希值。
- 文件:`file:.git/objects/info/packs`：该文件记录Git对象库中打包文件列表。

正是通过这两个文件，Git客户端才检索到版本库的引用列表和对象库的包列表，从而实现对版本库的读取操作。

为支持哑HTTP协议，必须在版本库更新后及时更新上述两个文件。幸好Git版本库的钩子脚本:`:file:`post-update``可以帮助完成这个无聊的工作。在版本库的`hooks`目录下创建可执行脚本文件:`:file:`post-update``，内容如下：

```
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, rename this file to "post-update".
#
exec git update-server-info
```

哑HTTP协议也可以对版本库写操作提供支持，即允许客户端向服务器推送。这需要在Apache中为版本库设置WebDAV，并配置口令认证。例如下面的Apache配置片段：

```
Alias /git /path/to/repos

<Directory /path/to/repos>
    Options FollowSymLinks
    AllowOverride None
    Order Allow,Deny
    Allow from all

    # 启用 WebDAV
    DAV on

    # 简单口令认证
    AuthType Basic
    AuthName "Git Repository"
    AuthBasicProvider file
    # 该口令文件用 htpasswd 命令进行管理
    AuthUserFile /path/to/git-passwd
    Require valid-user

    # 基于主机IP的认证和基于口令的认证必须同时满足
    Satisfy All
</Directory>
```

配置了口令认证后，最好使用HTTPS协议访问服务器，以避免因为口令在网络中明文传输造成口令泄露。还可以在URL地址中加上用户名，以免在连接过程中的重复输入。下面的示例中以特定用户（如：`jiangxin`）身份访问版本库：

- 如果版本库尚未克隆，使用如下命令克隆：

```
$ git clone https://jiangxin@server/git/myrepo.git
```

- 如果已经克隆了版本库，可以执行下面命令修改远程origin版本库的URL地址：

```
$ cd myrepos
$ git remote set-url origin https://jiangxin@server/git/myrepo.git
$ git pull
```

第一次连接服务器，会提示输入口令。正确输入口令后，完成克隆或版本库的更新。试着在版本库中添加新的提交，然后执行`git push`推送到HTTP服务器。

如果推送失败，可能是WebDAV配置的问题，或者是版本库的文件、目录的权限不正确（需

要能够被执行Apache进程的用户可以读写）。一个诊断Apache的小窍门是查看和跟踪Apache的配置文件[\[1\]](#)。如下：

```
$ tail -f /var/www/error.log
```

## 智能HTTP协议

Git 1.6.6之后的版本，提供了针对HTTP协议的CGI程序：`file:`git-http-backend``，实现了智能的HTTP协议支持。同时也要求Git客户端的版本也不低于1.6.6。

查看文件：`file:`git-http-backend``的安装位置，可以用如下命令。

```
$ ls $(git --exec-path)/git-http-backend  
/usr/lib/git-core/git-http-backend
```

在Apache2中为Git配置智能HTTP协议如下。

```
SetEnv GIT_PROJECT_ROOT /var/www/git  
SetEnv GIT_HTTP_EXPORT_ALL  
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

说明：

- 第一行设置版本库的根目录为：`file:`/var/www/git``。
- 第二行设置所有版本库均可访问，无论是否在版本库中存在：`file:`git-daemon-export-ok``文件。

缺省只有在版本库目录中存在文件：`file:`git-daemon-export-ok``，该版本库才可以访问。这个文件是git-daemon服务的一个特性。

- 第三行，就是使用：`file:`git-http-backend``CGI脚本来相应客户端的请求。

当用地址`http://server/git/myrepo.git`访问时，即由此CGI提供服务。

## 写操作授权

上面的配置只能提供版本库的读取服务，若想提供基于HTTP协议的写操作，必须添加认证的配置指令。当用户通过认证后，才能对版本库进行写操作。

下面的Apache配置，在前面配置的基础上，为Git写操作提供授权：

```
<LocationMatch "^/git/.*git-receive-pack$">  
AuthType Basic  
AuthName "Git Access"  
AuthType Basic  
AuthBasicProvider file  
AuthUserFile /path/to/passwd/file  
...  
</LocationMatch>
```

## 读和写均需授权

如果需要对读操作也进行授权，那就更简单了，一个Location语句就够了。

```
<Location /git/private>  
AuthType Basic  
AuthName "Git Access"  
AuthType Basic
```

```
AuthBasicProvider file  
AuthUserFile /path/to/passwd/file  
...  
</Location>
```

### 对静态文件的直接访问

如果对静态文件的访问不经过CGI程序，直接由Apache提供服务，会提高访问性能。

下面的设置对Git版本库中的:`:file:`objects``目录下文件的访问，不经过CGI。

```
SetEnv GIT_PROJECT_ROOT /var/www/git  
  
AliasMatch ^/git/(.*objects/[0-9a-f]{2}/[0-9a-f]{38})$ /var/www/git  
AliasMatch ^/git/(.*objects/pack/pack-[0-9a-f]{40}.(pack|idx))$ /var/www/git  
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Git的智能HTTP服务彻底打破了以前哑传输协议给HTTP协议带来的恶劣印象，让HTTP协议成为Git服务的一个重要选项。但是在授权的管理上，智能HTTP服务仅仅依赖Apache自身的授权模型，相比后面要介绍的Gitosis和Gitolite，可管理性要弱的多。

- 创建版本库只能在服务器端进行，不能通过远程客户端进行。
- 配置认证和授权，也只能在服务器端进行，不能在客户端远程配置。
- 版本库的写操作授权只能进行非零即壹的授权，不能针对分支甚至路径进行授权。

需要企业级的版本库管理，还需要考虑后面介绍的基于SSH协议的Gitolite或Gitosis。

## Gitweb服务器

前面介绍的HTTP哑协议和智能HTTP协议服务架设，都可以用于提供Git版本库的读写服务，而本节介绍的Gitweb作为一个Web应用，只提供版本库的图形化浏览功能，而不能提供版本库本身的读写。

Gitweb是用Perl语言开发的CGI脚本，架设比较方便。Gitweb支持多个版本库，可以对版本库进行目录浏览（包括历史版本），可以查看文件内容，查看提交历史，提供搜索以及RSS feed支持。也可以提供目录文件的打包下载等。图27-1就是kernel.org上的Gitweb示例。

图27-1: Gitweb界面(kernel.org)

## Gitweb安装

各个Linux平台都会提供Gitweb软件包。如在Debian/Ubuntu上安装Gitweb:

```
$ sudo aptitude install gitweb
```

安装文件列表:

- 配置文件: `:file:`/etc/gitweb.conf``。
- Apache配置文件: `:file:`/etc/apache2/conf.d/gitweb``。默认设置用URL地址/`gitweb`来访问Gitweb服务。
- CGI脚本: `:file:`/usr/share/gitweb/index.cgi``。
- 其他附属文件: `:file:`/usr/share/gitweb/*``, 如: 图片和css等。

## Gitweb配置

编辑`:file:`/etc/gitweb.conf``, 更改Gitweb的默认设置。

- 版本库根目录的设置。

```
$projectroot = "/var/cache/git";
```

- 访问版本库多种协议的地址设置。

Gitweb可以为每个版本库显示访问的协议地址。可以在列表中填入多个地址。

```
@git_base_url_list = ("git://bj.osxp.com/git", "ssh://git@bj.osxp.com"
```

- 增加 actions 菜单

```
$feature{'actions'}{'default'} = [('git', 'git://bj.osxp.com/git/%n', 't
```

- 在首页上显示自定义信息

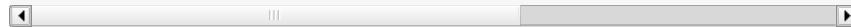
设定自定义HTML的文件名。

```
$home_text = "indextext.html";
```

在CGI脚本所在的目录下, 创建`:file:`indextext.html``文件。下面是我们公司(北京群英汇信息技术有限公司)内部gitweb自定义首页的内容。

```
<html>
<head>
</head>
<body>
<h2>群英汇 - git 代码库</h2>
<ul>
<li>点击版本库, 进入相应的版本库页面, 有 URL 指向一个 git://... 的检出链接</li>
<li>使用命令 git clone git://... 来克隆一个版本库</li>
<li>对于名称中含有 <i>-gitsvn</i> 字样的代码库, 是用 git-svn 从 svn 代码库镜
    <ul>
        <li>要将 git 库的远程分支(.git/ref/remotes/*) 也同步到本地!
            <pre>
                $ git config --add remote.origin.fetch '+refs/remotes/*:refs/remo
                $ git fetch
            </pre>
        </li>
        <li>如果需要克隆库和 Subversion 同步。用 git-svn 初始化代码库, 并使得相关
    </ul>
```

```
</li>
</ul>
</body>
</html>
```



- 版本库列表。

缺省扫描版本库根目录，查找版本库。如果版本库非常多，这个查找过程可能很耗时，可以提供一个文本文件包含版本库的列表，会加速Gitweb显示初始化。

```
# $projects_list = $projectroot;
$projects_list = "/home/git/gitosis/projects.list";
```

后面介绍的Gitosis和Gitolite都可以自动生成这么一个版本库列表，供Gitweb使用。

- Gitweb菜单定制。

- 在tree view文件的旁边显示追溯（blame）链接。

```
$feature{'blame'}{'default'} = [1];
$feature{'blame'}{'override'} = 1;
```

- 可以通过版本库的配置文件:`:file:`config``对版本库进行单独设置。

下面的设置覆盖Gitweb的全局设置，不对该项目显示blame菜单。

```
[gitweb]
blame = 0
```

- 为每个tree添加快照（snapshot）下载链接。

```
$feature{'pickaxe'}{'default'} = [1];
$feature{'pickaxe'}{'override'} = 1;
$feature{'snapshot'}{'default'} = ['zip', 'tgz'];
$feature{'snapshot'}{'override'} = 1;
```

## 版本库的Gitweb相关设置

可以通过Git版本库下的配置文件，定制版本库在Gitweb下的显示。

- 文件:`:file:`description``。

提供一行简短的git库描述。显示在版本库列表中。

也可以通过`:file:`config``配置文件中的`gitweb.description`进行设置。但是文件优先。

- 文件:`:file:`README.html``。

提供更详细的项目描述，显示在Gitweb项目页面中。

- 文件:`:file:`cloneurl``。

版本库访问的URL地址，一个一行。

- 文件:`:file:`config``。

通过`[gitweb]`小节的配置，覆盖Gitweb全局设置。

- gitweb.owner用于显示版本库的创建者。
- gitweb.description显示项目的简短描述，也可以通过:file:`description`文件来提供。（文件优先）
- gitweb.url显示项目的URL列表，也可以通过cloneurl文件来提供。（文件优先）

[1]

Apache日志文件的位置参见Apache配置文件中ErrorLog指令的设定。

来源: <https://github.com/gotgit/gotgit/blob/master/05-git-server/010-http.rst>

# 使用Git协议

Git协议是提供Git版本库只读服务的最为常用的协议，也是非常易用和易于配置的协议。该协议的缺点就是不能提供身份认证，而且一般也不提供写入服务。

## Git协议语法格式

Git协议的语法格式如下。

```
语法: git://<server>[:<port>]/path/to/repos.git/
```

说明:

- 端口为可选项，默认端口为9418。
- 版本库路径:`:file:`/path/to/repos.git``的根目录并不一定是系统的根目录，可以在:`:command:`git-daemon``启动时用参数`--base-path`指定根目录。如果:`:command:`git-daemon``没有设置根目录，则对应于系统的根目录。

## Git服务软件

Git服务由名为git-daemon的服务软件提供。虽然git-daemon也可以支持写操作，但因为git-daemon没有提供认证支持，因此没有人胆敢配置git-daemon提供匿名的写服务。使用git-daemon提供的Git版本库的只读服务，效率很高，而且是一种智能协议，在操作过程有进度显示，远比HTTP通讯协议方便（Git 1.6.6之后的版本已经支持智能HTTP通讯协议）。因此git-daemon很久以来，一直是Git版本库只读服务的首选。

Git软件包本身提供了git-daemon，因此只要安装了Git，一般就已经安装了git-daemon。默认git-daemon并没有运行，需要对其进行配置以服务方式运行。下面介绍两种不同的配置运行方式。

## 以inetd方式配置运行

最简单的方式，是以inetd服务方式运行git-daemon。在配置文件:`:file:`/etc/inetd.conf``中添加设置如下：

```
git stream tcp nowait nobody /usr/bin/git
    git daemon --inetd --verbose --export-all
        /gitroot/foo /gitroot/bar
```

说明：

- 以nobody用户身份执行:command:`git daemon`服务。
- 缺省:command:`git daemon`只对包含文件:file:`git-daemon-export-ok`的版本库提供服务。使用参数--export-all后，无论版本库是否存在标识文件:file:`git-daemon-export-ok`，都对版本库提供Git协议支持。
- 后面的两个参数是版本库。

也可以通过--base-path=<path>参数，设置版本库的根，对于这个目录下的所有版本库提供访问服务。例如下面的inetd配置：

```
git stream tcp nowait nobody /usr/bin/git
    git daemon --inetd --verbose --export-all
    --base-path=/var/cache /var/cache/git
```

## 以runit方式配置运行

runit是类似于sysvinit的服务管理进程，但是更简单。在Debian/Ubuntu上的软件包:command:`git-daemon-run`就是基于runit启动git-daemon服务。

- 安装:command:`git-daemon-run`：

```
$ sudo aptitude install git-daemon-run
```

- 配置:command:`git-daemon-run`：

缺省的服务配置文件：:file:`/etc/sv/git-daemon/run`。和之前的inetd运行方式相比，以独立的服务进程启动，相应速度更快。

```
#!/bin/sh
exec 2>&1
echo 'git-daemon starting.'
exec chpst -ugitdaemon \
    "$(git --exec-path)"/git-daemon --verbose --export-all --base-path=/var
```

缺省版本库中需要存在文件:file:`git-daemon-export-ok`，git-daemon才对此版本库提供服务。不过可以通过启动git-daemon时提供的参数--export-all，无论版本库是否存在标识文件:file:`git-daemon-export-ok`，都对版本库提供Git协议支持。

通过git-daemon提供的Git访问协议存在着局限性：

- 不支持认证。管理员大概可以做的只是配置防火墙，限制某个网段用户的使用。
- 只能提供匿名的版本库读取服务。因为写操作没有授权控制，因此一般不用来提供写操作。

来源： <https://github.com/gotgit/gotgit/blob/master/05-git-server/020-git.rst>

## 使用SSH协议

SSH协议用于为Git提供远程读写操作，是远程写操作的标准服务，在智能HTTP协议出现之前，甚至是写操作的唯一标准服务。

### SSH协议语法格式

对于拥有shell登录权限的用户帐号，可以用下面的语法访问Git版本库：

```
语法1: ssh://[<username>@]<server>[:<port>]/path/to/repos/myrepo.git  
语法2: [<username>@]<server>:/path/to/repos/myrepo.git
```

说明：

- SSH协议地址格式可以使用两种不同的写法，第一种是使用ssh://开头的SSH协议标准URL写法，另外一种是SCP格式的写法。

两种写法均可，SSH协议标准URL写法稍嫌复杂，但是对于非标准SSH端口（非22端口），可以通过URL给出端口号。

- <username>是服务器<server>上的用户帐号。

如果省略用户名，则缺省使用当前登录用户名（配置和使用了服务器别名除外）。

- <port>为SSH协议端口缺省为22。

端口只有在SSH协议标准URL写法可以给出，如果省略则使用缺省值22（配置和使用了服务器别名除外）。

- :file:`/path/to/repos/myrepo.git` 是服务器中版本库的绝对路径。若用相对路径则相对于username用户的主目录而言。

- 如果采用口令认证，不能像HTTPS协议那样可以在URL中同时给出登录名和口令，必须每次连接时输入。

- 如果采用公钥认证，则无须输入口令。

## 服务架设方式比较

SSH协议来实现Git服务，有如下方式：

- 其一是用标准的ssh帐号访问版本库。即用户帐号可以直接登录到服务器，获得shell。

对于这种使用标准SSH帐号方式，直接使用标准的SSH服务就可以了，无须赘述。

- 另外的方式是，所有用户都使用同一个专用的SSH帐号访问版本库。各个用户通过公钥认证的方式用此专用SSH帐号访问版本库。而用户在连接时使用的不同的公钥可以用于区分不同的用户身份。

Gitosis和Gitolite就是实现该方式的两个服务器软件。

标准SSH帐号和专用SSH帐号的区别见表29-1。

表29-1：不同SSH服务架设Git的对照

	标准SSH	Gitosis/Gitolite
--	-------	------------------

帐号	每个用户一个帐号	所有用户共用同一个帐号
认证方式	口令或公钥认证	公钥认证
用户是否能直接登录shell	是	否
安全性	差	好
管理员是否需要shell	是	否
版本库路径	相对路径或绝对路径	相对路径
授权方式	操作系统中用户组和目录权限	通过配置文件授权
对分支进行写授权	否	Gitolite
对路径进行写授权	否	Gitolite
架设难度	简单	复杂

实际上，标准SSH，也可以用公钥认证的方式实现所有用户共用同一个帐号。不过这类似于把一个公共帐号的登录口令同时告诉给多人。

- 在服务器端(server)创建一个公共帐号，例如：anonymous。
- 管理员收集需要访问git服务的用户公钥。  
如：`:file:`user1.pub``、`:file:`user2.pub``。
- 使用`:command:`ssh-copy-id``命令远程将各个git用户的公钥加入服务器(server)的公钥认证列表中。

```
$ ssh-copy-id -i user1.pub anonymous@server
$ ssh-copy-id -i user2.pub anonymous@server
```

如果直接在服务器上操作，则直接将文件追加到`:file:`authorized_keys``文件中。

```
$ cat /path/to/user1.pub >> ~anonymous/.ssh/authorized_keys
$ cat /path/to/user2.pub >> ~anonymous/.ssh/authorized_keys
```

- 在服务器端的anonymous用户主目录下建立git库，就可以实现多个用户利用同一个系统帐号(git)访问Git服务了。

这样做除了免除了逐一设置帐号，以及用户无需口令认证之外，标准SSH部署Git服务的缺点一个也不少，而且因为用户之间无法区分，更无法进行针对用户授权。

下面重点介绍一下SSH公钥认证，因为它们是后面介绍的Gitosis和Gitolite服务器软件的基础。

## 关于SSH公钥认证

关于公钥认证的原理，维基百科上的这个条目是一个很好的起点：[http://en.wikipedia.org/wiki/Public-key\\_cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography)。

如果用户的主目录下不存在`:file:`.ssh``目录，说明SSH公钥/私钥对尚未创建。可以用这个命令创建：

```
$ ssh-keygen
```

该命令会在用户主目录下创建`:file:`.ssh``目录，并在其中创建两个文件：

- `id rsa`

私钥文件。是基于RSA算法创建。该私钥文件要妥善保管，不要泄漏。

- `id_rsa.pub`

公钥文件。和`:file:`id_rsa``文件是一对儿，该文件作为公钥文件，可以公开。

创建了自己的公钥/私钥对后，就可以使用下面的命令，实现无口令登录远程服务器，即用公钥认证取代口令认证。

```
$ ssh-copy-id -i .ssh/id_rsa.pub <user>@<server>
```

说明：

- 该命令会提示输入用户`user`在`server`上的SSH登录口令。
- 当此命令执行成功后，再以`user`用户登录`server`远程主机时，不必输入口令直接登录。
- 该命令实际上将`:file:`.ssh/id_rsa.pub``公钥文件追加到远程主机`server`的`user`主目录下的`:file:`.ssh/authorized_keys``文件中。

检查公钥认证是否生效，运行SSH到远程主机，正常的话应该直接登录成功。如果要求输入口令则表明公钥认证配置存在问题。如果SSH登录存在问题，可以通过查看服务器端的`:file:`/var/log/auth.log``日志文件进行诊断。

## 关于SSH主机别名

在实际应用中，有时需要使用多套公钥/私钥对，例如：

- 使用默认的公钥访问git帐号，获取shell，进行管理员维护工作。
- 使用单独创建的公钥访问git帐号，执行git命令。
- 访问GitHub（免费的Git服务托管商）采用其他公钥。

首先要能够创建不同名称的公钥/私钥对。还是用`:command:`ssh-keygen``命令，如下：

```
$ ssh-keygen -f ~/.ssh/<filename>
```

注：

- 将`:file:`<filename>``替换为有意义的名称。
- 会在`:file:`~/.ssh``目录下创建指定的公钥/私钥对。文件`:file:`<filename>``是私钥，文件`:file:`<filename>.pub``是公钥。

将新生成的公钥添加到远程主机的`:file:`.ssh/authorized_keys``文件中，建立新的公钥认证。例如：

```
$ ssh-copy-id -i .ssh/<filename>.pub <user>@<server>
```

这样，就有两个公钥用于登录主机`server`，那么当执行下面的ssh登录指令，用到的是那个公钥呢？

```
$ ssh <user>@<server>
```

当然是默认公钥`:file:`~/.ssh/id_rsa.pub``。那么如何用新建的公钥连接`server`呢？

SSH的客户端配置文件`:file:`~/.ssh/config``可以通过创建主机别名，在连接主机时，使用特定的公钥。例如`:file:`~/.ssh/config``文件中的下列配置：

```
host bj
  user git
  hostname bj.ossexp.com
  port 22
  identityfile ~/.ssh/jiangxin
```

当执行

```
$ ssh bj
```

或者执行

```
$ git clone bj:path/to/repos/myrepo.git
```

含义为：

- 登录的SSH主机为bj.ossexp.com。
- 登录时使用的用户名为git。
- 认证时使用的公钥文件为:file:`~/.ssh/jiangxin.pub`。

来源：<https://github.com/gotgit/gotgit/blob/master/05-git-server/030-ssh.rst>

## Gitolite服务架设

Gitolite是一款Perl语言开发的Git服务管理工具，通过公钥对用户进行认证，并能够通过配置文件对写操作进行基于分支和路径的精细授权。Gitolite采用的是SSH协议并且使用SSH公钥认证，因此无论是管理员还是普通用户，都需要对SSH非常熟悉。在开始之前，请确认您已经通读过第29章“使用SSH协议”。

Gitolite的官方网址是：<http://github.com/sitaramc/gitolite>。从提交日志里可以看出作者是Sitaram Chamarty，最早的提交开始于2009年8月。作者是受到了Gitosis的启发，开发了这款功能更为强大和易于安装的软件。Gitolite的命名，作者的原意是Gitosis和lite的组合，不过因为Gitolite的功能越来越强大，已经超越了Gitosis，因此作者笑称Gitolite可以看作是Github-lite——轻量级的Github。

我是在2010年8月才发现Gitolite这个项目的，并尝试将公司基于Gitosis的管理系统迁移至Gitolite。在迁移和使用过程中，增加和改进了一些实现，如：通配符版本库的创建过程，对创建者的授权，版本库名称映射等。本文关于Gitolite的介绍也是基于我改进的版本[1]。

- 原作者的版本库地址：

<http://github.com/sitaramc/gitolite>

- 笔者改进后的Gitolite分支：

<http://github.com/ossxp-com/gitolite>

Gitolite的实现机制和使用特点概述如下：

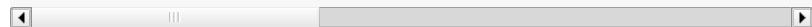
- Gitolite安装在服务器（server）某个帐号之下，例如git帐号。
- 管理员通过`:command:`git``命令检出名为`gitolite-admin`的版本库。

```
$ git clone git@server:gitolite-admin.git
```

- 管理员将所有Git用户的公钥保存在`gitolite-admin`库的`:file:`keydir``目录下，并编辑`:file:`conf/gitolite.conf``文件为用户授权。
- 当管理员提交对`gitolite-admin`库的修改并推送到服务器之后，服务器上`gitolite-admin`版本库的钩子脚本将执行相应的设置工作。

- 新用户的公钥自动追加到服务器端安装帐号主目录下的`:file:`.ssh/authorized_keys``文件中，并设置该用户的shell为`gitolite`的一条命令`:command:`gl-auth-command``。在`:file:`.ssh/authorized_keys``文件中增加的内容示例如下： [2]

```
command="/home/git/bin/gl-auth-command jiangxin",no-port-forwarding,n
```



- 更新服务器端的授权文件`:file:`~/.gitolite/conf/gitolite.conf``。

- 编译授权文件为`:file:`~/.gitolite/conf/gitolite.conf-compiled.pm``。

- 若用ssh命令登录服务器（以git用户登录）时，因为公钥认证的相关设置（使用`:command:`gl-auth-command``作为shell），不能进入shell环境，而是打印服务器端git库授权信息后马上退出。即用户不会通过git用户进入服务器的shell，也不会对系统的安全造成威胁。

```
$ ssh git@bj
hello jiangxin, the gitolite version here is v1.5.5-9-g4c11bd8
the gitolite config gives you the following access:
    R          gistore-bj.osxp.com/.*$
    C   R   W      ossxp/.*$
@C @R   W      users/jiangxin/.+$
Connection to bj closed.
```

- 用户可以用git命令访问授权的版本库。
- 若管理员授权，用户可以远程在服务器上创建新版本库。

下面介绍Gitolite的部署和使用。

## 安装Gitolite

安装Gitolite（2.1版本）对服务器的要求是：

- Git版本为1.6.6或以上。
- Unix或类Unix（Linux、MacOS等）操作系统。
- 服务器开启SSH服务。

和其他Unix上软件包一样Gitolite既可通过操作系统本身提供的二进制发布包方式安装，也可通过克隆Gitolite源码库从源代码安装Gitolite。

Note

老版本的Gitolite提供了一种从客户端发起安装的模式，但该安装模式需要管理员维护两套不同公钥/私钥对（一个公钥用于无口令登录服务器以安装和更新软件，另外一个公钥用于克隆和推送gitolite-admin版本库），稍嫌复杂，在2.1之后的Gitolite取消了这种安装模式。

## 安装之前

Gitolite搭建的Git服务器是以SSH公钥认证为基础的，无论是普通Git用户还是Gitolite的管理员都通过公钥认证访问Gitolite服务器。在Gitolite的安装过程中需要提供管理员公钥，以便在Gitolite安装完毕后管理员能够远程克隆gitolite-admin版本库（仅对管理员授权），对Gitolite服务器进行管理——添加新用户和为用户添加授权。

为此在安装Gitolite之前，管理员需要在客户端（用于远程管理Gitolite服务器的客户端）创建用于连接Gitolite服务器的SSH公钥（如果尚不存在的话），并把公钥文件拷贝到服务器上。

1. 在客户端创建SSH公钥/私钥对。

如果管理员在客户端尚未创建公钥/私钥对，使用下面的命令会在用户主目录下创建名为:`:file:`~/.ssh/id_rsa``的SSH私钥和名为:`:file:`~/.ssh/id_rsa.pub``的公钥文件：

```
$ ssh-keygen
```

2. 将公钥文件从客户端复制到服务器端，以便安装Gitolite时备用。

可以使用:`:command:`ftp``或U盘拷贝等方式从客户端向服务器端传送文件，不过用:`:command:`scp``命令是非常方便的，例如服务器地址为`server`，相应的拷贝命令为：

```
$ scp ~/.ssh/id_rsa.pub server:/tmp/admin.pub
```

## 以发布包形式安装

常见的Linux发行版都包含了Gitolite软件包，安装Gitolite使用如下命令：

- Debian/Ubuntu:

```
$ sudo aptitude install gitolite
```

- RedHat:

```
$ sudo yum install gitolite
```

安装完毕后会自动创建一个专用系统账号如gitolite。在Debian平台上创建的gitolite账号使用:`/var/lib/gitolite`作为用户主目录，而非`/home/gitolite`。

```
$ getent passwd gitolite  
gitolite:x:114:121:git repository hosting,,,:/var/lib/gitolite:/bin/bash
```

安装完毕，运行如下命令完成对Gitolite的配置：

1. 切换至新创建的gitolite用户账号。

```
$ sudo su - gitolite
```

2. 运行gl-setup命令，并以客户端复制过来的公钥文件路径作为参数。

```
$ gl-setup /tmp/admin.pub
```

Debian等平台会在安装过程中（或运行`sudo dpkg-reconfigure gitolite`命令时），开启配置界面要求用户输入Gitolite专用账号、Git版本库根目录、管理员公钥文件名，然后自动执行`gl-setup`完成设置。

## 从源代码开始安装

如果想在系统中部署多个Gitolite实例，希望部署最新的Gitolite版本，或者希望安装自己或他人对Gitolite的定制版本，就要采用从源代码进行Gitolite部署。

1. 创建专用系统账号。

首先需要在服务器上创建Gitolite专用帐号。因为所有用户都要通过此帐号访问Git版本库，为方便易记一般选择更为简练的git作为专用帐号名称。

```
$ sudo adduser --system --group --shell /bin/bash git
```

注意添加的用户要能够远程登录，若系统只允许特定用户组（如ssh用户组）的用户才可以通过SSH协议登录，就需要将新建的git用户添加到该特定的用户组中。执行下面的命令可以将git用户添加到ssh用户组。

```
$ sudo adduser git ssh
```

取消git用户的口令，以便只能通过公钥对git账号进行认证，增加系统安全性。

```
$ sudo passwd --delete git
```

2. 切换到新创建的用户账号，后续的安装都以该用户身份执行。

```
$ sudo su - git
```

3. 在服务器端下载Gitolite源码。一个更加“Git”的方式就是克隆Gitolite的版本库。

- 克隆官方的Gitolite版本库如下：

```
$ git clone git://github.com/sitaramc/gitolite.git
```

- 也可以克隆定制后的Gitolite版本库，如我在GitHub上基于Gitolite官方版本库建立的分支版本：

```
$ git clone git://github.com/ossxp-com/gitolite.git
```

4. 安装Gitolite。

运行源码目录中的`:command:`src/gl-system-install``执行安装。

```
$ cd gitolite  
$ src/gl-system-install
```

如果像上面那样不带参数的执行安装程序，会将Gitolite相关命令安装到`:file:`~/.bin``目录中，相当于执行：

```
$ src/gl-system-install $HOME/bin $HOME/share/gitolite/conf $HOME/share/g
```

5. 运行`:command:`gl-setup``完成设置。

若Gitolite安装到`:file:`~/.bin``目录下（即没有安装到系统目录下），需要设置PATH环境变量以便`:command:`gl-setup``能够正常运行。

```
$ export PATH=~/bin:$PATH
```

然后运行`:command:`gl-setup``命令，并以客户端复制过来的公钥文件路径作为参数。

```
$ ~/bin/gl-setup /tmp/admin.pub
```

## 管理Gitolite

### 管理员克隆gitolite-admin管理库

当Gitolite安装完成后，就会在服务器端版本库根目录下创建一个用于管理Gitolite的版本库。若以git用户安装，则该Git版本库的路径为：`:file:`~/.git/repositories/gitolite-admin.git``。

在客户端用`:command:`ssh``命令连接服务器server的git用户，如果公钥认证验证正确的话，Gitolite将此SSH会话的用户认证为admin用户，显示admin用户的权限。如下：

```
$ ssh -T git@server  
hello admin, this is gitolite v2.1-7-ge5c49b7 running on git 1.7.7.1  
the gitolite config gives you the following access:
```

```
R W      gitolite-admin  
@R_ @W_    testing
```

从上面命令的倒数第二行输出可以看出用户admin对版本库gitolite-admin拥有读写权限。

为了对Gitolite服务器进行管理，需要在客户端克隆gitolite-admin版本库，使用如下命令：

```
$ git clone git@server:gitolite-admin.git  
$ cd gitolite-admin/
```

在客户端克隆的:`:file:`gitolite-admin``目录下有两个子目录:`:file:`conf/``和`:file:`keydir/``，包含如下文件：

- 文件：`:file:`keydir/admin.pub``。

目录`:file:`keydir``下初始时只有一个用户公钥，即管理员admin的公钥。

- 文件：`:file:`conf/gitolite.conf``。

该文件为授权文件。初始内容为：

```
repo      gitolite-admin  
RW+      =      admin  
  
repo      testing  
RW+      =      @all
```

默认授权文件中只设置了两个版本库的授权：

- `gitolite-admin`

即本版本库。此版本库用于Gitolite管理，只有admin用户有读写和强制更新的权限。

- `testing`

默认设置的测试版本库。设置为任何人都可以读写及强制更新。

## 增加新用户

增加新用户，就是允许新用户能够通过其公钥访问Git服务。只要将新用户的公钥添加到gitolite-admin版本库的`:file:`keydir``目录下，即完成新用户的添加，具体操作过程如下。

1. 管理员从用户获取公钥，并将公钥按照`:file:`username.pub``格式进行重命名。
  - 用户可以通过邮件或其他方式将公钥传递给管理员，切记不要将私钥误传给管理员。如果发生私钥泄漏，马上重新生成新的公钥/私钥对，并将新的公钥传递给管理员，并申请将旧的公钥作废。
  - 用户从不同的客户端主机访问有着不同的公钥，如果希望使用同一个用户名进行授权，可以按照`:file:`username@host.pub``的方式命名公钥文件，和名为`:file:`username.pub``的公钥指向同一个用户`username`。
  - Gitolite也支持邮件地址格式的公钥，即形如`:file:`username@gmail.com.pub``的公钥。Gitolite能够很智能地区分是以邮件地址命名的公钥还是相同用户在不同主机上的公钥。如果是邮件地址命名的公钥，将以整个邮件地址作为用户名。
  - 还可以在`:file:`keydir``目录下创建子目录来管理用户公钥，同一用户的不同公钥可以用同一名称保存在不同子目录中。

- 管理员进入`gitolite-admin`本地克隆版本库中，复制新用户公钥到`:file:`keydir``目录。

```
$ cp /path/to/dev1.pub keydir/  
$ cp /path/to/dev2.pub keydir/  
$ cp /path/to/jiangxin.pub keydir/
```

- 执行`:command:`git add``命令，将公钥添加到版本库。

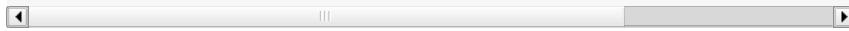
```
$ git add keydir
```

- 执行`:command:`git commit``，完成提交。

```
$ git commit -m "add user: jiangxin, dev1, dev2"
```

- 执行`:command:`git push``，同步到服务器，才真正完成新用户的添加。

```
$ git push  
Counting objects: 8, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (6/6), 1.38 KiB, done.  
Total 6 (delta 0), reused 0 (delta 0)  
remote: Already on 'master'  
remote:  
remote:           ***** WARNING *****  
remote:           the following users (pubkey files in parens) do not appear  
remote: dev1(dev1.pub),dev2(dev2.pub),jiangxin(jiangxin.pub)
```



在`:command:`git push``的输出中，以`remote`标识的输出是服务器端执行`:file:`post-update``钩子脚本的错误输出，用于提示新增的三个用户（公钥）在授权文件中没有被引用。接下来会介绍如何修改授权文件，以及如何为用户添加授权。

服务器端的`git`主目录下的`:file:`.ssh/authorized_keys``文件会随着新增用户公钥而更新，即添加三条新的记录。如下：

```
$ cat ~git/.ssh/authorized_keys  
# gitolite start  
command="/home/git/bin/gl-auth-command admin",no-port-forwarding,no-X11-forwar  
command="/home/git/bin/gl-auth-command dev1",no-port-forwarding,no-X11-forwar  
command="/home/git/bin/gl-auth-command dev2",no-port-forwarding,no-X11-forwar  
command="/home/git/bin/gl-auth-command jiangxin",no-port-forwarding,no-X11-fo  
# gitolite end
```

## 更改授权

新用户添加完毕，接下来需要为新用户添加授权，这个过程也比较简单，只需修改`:file:`conf/gitolite.conf``配置文件，提交并推送。具体操作过程如下：

- 管理员进入`:file:`gitolite-admin``本地克隆版本库中，编辑`:file:`conf/gitolite.conf``。

```
$ vi conf/gitolite.conf
```

- 授权指令比较复杂，先通过建立新用户组尝试一下更改授权文件。

考虑到之前增加了三个用户公钥，服务器端发出了用户尚未在授权文件中出现的警告。现在就在这个示例中解决这个问题。

- 可以在其中加入用户组@team1，将新添加的用户jiangxin、dev1、dev2都归属到这个组中。

只需要在:`:file:`conf/gitolite.conf``文件的文件头加入如下指令即可。用户名之间用空格分隔。

```
@team1 = dev1 dev2 jiangxin
```

- 编辑完毕退出。可以用:`:command:`git diff``命令查看改动：

还修改了版本库testing的授权，将@all用户组改为新建立的@team1用户组。

```
$ git diff
diff --git a/conf/gitolite.conf b/conf/gitolite.conf
index 6c5fdf8..f983a84 100644
--- a/conf/gitolite.conf
+++ b/conf/gitolite.conf
@@ -1,5 +1,7 @@
+@team1 = dev1 dev2 jiangxin
+
repo    gitolite-admin
RW+      =    admin

repo    testing
-        RW+      =    @all
+        RW+      =    @team1
```

3. 编辑结束，提交改动。

```
$ git add conf/gitolite.conf
$ git commit -q -m "new team @team1 auth for repo testing."
```

4. 执行:`:command:`git push``，同步到服务器，授权文件的更改才真正生效。

可以注意到，推送后的输出中没有了警告。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 398 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
To git@server:gitolite-admin.git
 bd81884..79b29e4  master -> master
```

## Gitolite授权详解

### 授权文件的基本语法

下面看一个不那么简单的授权文件。为方便描述添加了行号。

```
1  @manager = jiangxin wangsheng
2  @dev    = dev1 dev2 dev3
3
```

```

4 repo gitolite-admin
5     RW+             = jiangxin
6
7 repo ossxp/[a-z].+
8     C               = @manager
9     RW+             = CREATOR
10    RW              = WRITERS
11    R               = READERS @dev
12
13 repo testing
14    RW+             = @manager
15    RW     master   = @dev
16    RW     refs/tags/v[0-9] = dev1
17    -      refs/tags/ = @all

```

在上面的示例中，演示了很多授权指令：

- 第1行，定义了用户组@manager，包含两个用户jiangxin和wangsheng。
- 第2行，定义了用户组@dev，包含三个用户dev1、dev2和dev3。
- 第4-5行，定义了版本库gitolite-admin。指定只有超级用户jiangxin才能够访问，并拥有读(R)写(W)和强制更新(+)的权限。
- 第7行，通过正则表达式为一组版本库进行批量授权。即针对:file:`ossxp`目录下以小写字母开头的所有版本库进行授权。
- 第8行，用户组@manager中的用户可以创建版本库。即可以在:file:`ossxp`目录下创建以小写字母开头的版本库。
- 第9行，版本库的创建者拥有对所创建版本库的完全权限。版本库的创建者是通过:command:`git push`命令创建版本库的那一个人。
- 第10-11行，出现了两个特殊角色WRITERS和READERS，这两个角色不在本配置文件中定义，而是由版本库创建者使用Gitolite支持的setperms命令进行设置。
- 第11行，还设置了@dev用户组的用户对:file:`ossxp`目录下的版本库具有读取权限。
- 第13行开始，对:file:`testing`版本库进行授权。其中使用了对引用授权的语法。
- 第14行，用户组@manager对所有引用包括分支拥有读写、重置、添加和删除的授权，但里程碑除外，因为第17行定义了一条禁用规则。
- 第15行，用户组@dev可以读写master分支。（还包括名字以master开头的其他分支，如果说有的话。）
- 第16行，用户dev1可以创建里程碑（即以refs/tags/v[0-9]开始的引用）。
- 第17行，禁止所有人(@all)对以refs/tags/开头的引用进行写操作。实际上由于之前第14行和第16行建立的授权，用户组@manager的用户和用户dev1能够创建里程碑，而且用户组@manager还能删除里程碑。

下面针对授权指令进行详细的讲解。

## 定义用户组和版本库组

在:file:`conf/gitolite.conf`授权文件中，可以定义用户组或版本库组。组名称以@字符开头，可以包含一个或多个成员。成员之间用空格分开。

- 例如定义管理员组：

```
@admin = jiangxin wangsheng
```

- 组可以嵌套：

```
@staff = @admin @engineers tester1
```

除了作为用户组外，同样的语法也适用于版本库组。版本库组和用户组的定义没有任何区别，只是在版本库授权指令中处于不同的位置。即位于授权指令中的版本库位置代表版本

库组，位于授权指令中的用户位置代表用户组。

## 版本库ACL

一个版本库可以包含多条授权指令，这些授权指令组成了一个版本库的权限控制列表（ACL）。例如：

```
repo testing
  RW+      = jiangxin @admin
  RW      = @dev @test
  R      = @all
```

## 版本库

每一个版本库授权都以一条repo指令开始。指令repo后面是版本库列表，版本之间用空格分开，还可以包括版本库组。示例如下：

```
repo sandbox/test1 sandbox/test2 @test_repos
```

注意版本库名称不要添加.git后缀，在版本库创建或权限匹配过程中会自动添加.git后缀。用repo指令定义的版本库会自动在服务器上创建，但使用正则表达式定义的通配符版本库除外。

通配符版本库就是在repo指令定义的版本库名称中使用了正则表达式。通配符版本库针对的不是某一个版本库，而是匹配一组版本库，这些版本库可能已经存在或尚未创建。例如下面的repo指令定义了一组通配符版本库。

```
repo redmine/[a-zA-Z].+
```

通配符版本库匹配时会自动在版本库名称前面加上前缀^，在后面添加后缀\$。即通配符版本库对版本库名称进行完整匹配而非部分匹配，这一点和后面将要介绍的正则引用（refex）大不一样。

有时repo指令定义普通版本库和通配符版本库的界限并不是那么清晰，像下面这条repo指令：

```
repo ossxp/.+
```

因为点号(.)和加号(+)也可以作为普通字符出现在版本库名称中，这条指令会导致Gitolite创建:`file:`ossxp``目录，并在目录下创建名为:`file:`.+.git``的版本库。因此在定义通配符版本库时要尽量写得“复杂点”以免造成误判。

### Tip

我对Gitolite进行了一点改进，能够减少对诸如`ossxp/.+`通配符版本库误判的可能。并提供在定义通配符版本库时使用^前缀和\$后缀，以减少误判。如使用如下方式定义通配符版本库：`repo ^myrepo`。

## 授权指令

在repo指令之后是缩进的一条或多条授权指令。授权指令的语法如下：

```
<权限> [零个或多个正则表达式匹配的引用] = <user> [<user> ...]
```

每条指令必须指定一个权限，称为授权关键字。包括传统的授权关键字：C、R、RW和RW+，

以及将分支创建和分支删除分离出来的扩展授权关键字：RWC、RW+C、RWD、RW+D、RWCD、RW+CD。

传统的授权关键字包括：

- C

C代表创建版本库，仅在对通配符版本库进行授权时方可使用。用于设定谁可以创建名称与通配符匹配的版本库。

- R

R代表只读权限。

- RW

RW代表读写权限。如果在同一组（针对同一版本库）授权指令中没有出现代表创建分支的扩展授权关键字，则RW还包括创建分支的权限，而不仅是在分支中的读写。

- RW+

RW+除了具有读写权限外，还可以强制推送（执行非快进式推送）。如果在同一组授权指令中没有出现代表分支删除的扩展授权关键字，则RW+还同时包含了创建分支和删除分支的授权。

- -

-含义为禁用。因为禁用规则只在第二阶段授权生效[3]，所以一般只用于撤销特定用户对特定分支或整个版本库的写操作授权。

扩展的授权关键字将创建分支和删除分支的权限从传统授权关键字中分离出来，从而新增了六个授权关键字。在一个版本库的授权指令中一旦发现创建分支和/或删除分支的授权使用了下列新的扩展授权关键字后，原有的RW和RW+不再行使对创建分支和/或删除分支的授权。

- RWC

RWC代表读写授权、创建新引用（分支、里程碑等）的授权。

- RW+C

RW+C代表读写授权、强制推送和创建新引用的授权。

- RWD

RWD代表读写授权、删除引用的授权。

- RW+D

RW+D代表读写授权、强制推送和删除引用的授权。

- RWCD

RWCD代表读写授权、创建新引用和删除引用的授权。

- RW+CD

RW+CD代表读写授权、强制推送、创建新引用和删除引用的授权。

授权关键字后面（等号前面）是一个可选的正则引用（refex）或正则引用列表（用空格分隔）。

- 正则表达式格式的引用，简称正则引用（refex），在授权检查时对Git版本库的引用进行匹配。
- 如果在授权指令中省略正则引用，则意味着该授权指令对全部的引用都有效。
- 正则引用如果不以refs/开头，会自动添加refs/heads/作为前缀。
- 正则引用默认采用部分匹配策略，即如果不以\$结尾，则后面可以匹配任意字符，相当于添加.\*\$作为后缀。

授权关键字后面（等号前面）也可以包含一个以NAME/为前缀的表达式，但这个表达式并非引用，而是路径。支持基于路径的写操作授权。

授权指令以等号（=）为标记分为前后两段，等号后面的是用户列表。用户之间用空格分隔，并且可以使用用户组。

## Gitolite授权机制

Gitolite的授权实际分为两个阶段。第一个阶段称为前Git阶段，即在Git命令执行前，由SSH连接触发的`:command: `gl-auth-command``命令执行的授权检查。包括：

- 版本库的读。

如果用户拥有版本库或版本库的任意分支具有下列权限之一：R、RW、RW+（或其他扩展关键字），则整个版本库（包含所有分支）对用户均可读，否则版本库不可读取。

最让人迷惑的就是只为某用户分配了对某个分支的读授权（R），而该用户实际上能够读取版本库的任意分支。之所以Gitolite对读授权不能细化到分支甚至目录，只能针对版本库进行粗放的非零即壹的读操作授权，是因为读授权只在版本库授权的第一个阶段进行检查，而在此阶段还获取不到版本库的分支。

- 版本库的写。

版本库的写授权实际上要在两个阶段分别进行检查。本阶段，即第一阶段仅检查用户是否拥有下列权限之一：RW、RW+或C授权，具有这些授权则通过第一阶段的写权限检查。第二个阶段的授权检查由Git版本库的钩子脚本触发，能够实现基于分支和路径的写操作授权，以及对分支创建、删除和是否可强制更新进行授权检查，具体见第二阶段授权过程描述。

- 版本库的创建。

仅对正则表达式定义的通配符版本库有效。即拥有C授权的用户可以创建和相应的正则表达式匹配的版本库。创建版本库（尤其是通过执行`:command: `git push``命令创建版本库）不免要涉及到执行新创建的版本库的钩子脚本，所以需要为版本库设置一条创建者可读写的授权。如：

```
RW = CREATOR
```

Gitolite对授权的第二个阶段的检查，实际上是通过`:file:`update``钩子脚本进行的。因为版本库的读操作不执行`:file:`update``钩子，所以读操作只在授权的第一个阶段（前Git阶段）就完成了检查，授权的第二个阶段仅对写操作进行更为精细的授权检查。

- 钩子脚本`:file:`update``针对推送操作的各个分支进行逐一检查，因此第二个阶段可以进行针对分支写操作的精细授权。
- 在这个阶段可以获取到要更新的新、老引用的SHA1哈希值，因此可以判断出是否发生了非快进式推送、是否有新分支创建，以及是否发生了分支的删除，因此可以针对这些操作进行精细的授权。
- 基于路径的写授权也是在这个阶段进行的。

## 版本库授权案例

Gitolite的授权非常强大也很复杂，因此从版本库授权的实际案例来学习是非常行之有效的方式。

## 常规版本库授权

授权文件如下：

```
1 @admin = jiangxin
2 @dev   = dev1 dev2 badboy jiangxin
3 @test  = test1 test2
4
5 repo testing
6     RW+ = @admin
7     R  = @test
8     -  = badboy
9     RW = @dev test1
```

关于授权的说明：

- 用户jiangxin对版本库具有写的权限，并能够强制推送。

由于用户jiangxin属于用户组@admin，通过第6行授权指令而具有读写权限，以及强制推送、创建和删除引用的权限。

- 用户test1对版本库具有写的权限。

第7行定义了test1所属的用户组@test具有只读权限。第9行定义了test1用户具有读写权限。Gitolite的实现是对读权限和写权限分别进行判断并汇总（并集），从而test1用户具有读写权限。

- 用户badboy对版本库只具有读操作的权限，没有写操作权限。

第8行的指令以减号（-）开始，是一条禁用指令。禁用指令只在授权的第二阶段起作用，即只对写操作起作用，不会对badboy用户的读权限施加影响。在第9行的指令中，badboy所在的@dev组拥有读写权限。但禁用规则会对写操作起作用，导致badboy只有读操作权限，而没有写操作。

上面在Gitolite配置文件中对testing版本库进行的授权，当通过推送更新至Gitolite服务器上时，如果服务器端尚不存在一个名为testing的版本库，Gitolite会自动初始化一个空白的testing版本库。

## 通配符版本库授权

授权文件如下：

```
1 @administrators = jiangxin admin
2 @dev           = dev1 dev2 badboy
3 @test          = test1 test2
4
5 repo    sandbox/[a-z].+
6     C      = @administrators
7     RW+    = CREATOR
8     R      = @test
9     -      = badboy
10    RW     = @dev test1
```

这个授权文件的版本库名称中使用了正则表达式，匹配在:`:file:`sandbox``目录下的任意以小写字母开头的版本库。因为通配符版本库并非指代一个具体版本库，因而不会在服务器端自动创建，而是需要管理员手动创建。

创建和通配符匹配的版本库，Gitolite的原始实现是克隆即创建。例如管理员jiangxin创建名为sandbox/repos1.git版本库，执行下面命令：

```
jiangxin$ git clone git@server:sandbox/repos1.git
```

这种克隆即创建的方式很容易因为录入错误而导致意外创建错误的版本库。我改进的Gitolite需要通过推送来创建版本库。下面的示例通过推送操作（以jiangxin用户身份），远程创建版本库sandbox/repos1.git。

```
jiangxin$ git remote add origin git@server:sandbox/repos1.git  
jiangxin$ git push origin master
```

对创建完成的sandbox/repo1.git版本库进行授权检查，会发现：

- 用户jiangxin对版本库具有读写权限，而用户admin则不能读取sandbox/repo1.git版本库。

第6行的授权指令同时为用户jiangxin和admin赋予了创建与通配符相符的版本库的权限。但因为版本库sandbox/repo1.git是由jiangxin而非admin创建的，所以第7条的授权指令只为版本库的创建者jiangxin赋予了读写权限。

Gitolite通过在服务器端该版本库目录下创建一个名为:`:file:`gl-creater``的文件记录了版本库的创建者。

- 和之前的例子相同的是：

- 用户test1对版本库具有写的权限。
- 禁用指令让用户badboy对版本库仅具有只读权限。

如果采用接下来的示例中的版本库权限设置，版本库sandbox/repo1.git的创建者jiangxin还可以使用`:command:`setperms``命令为版本库添加授权。具体用法参见下面的示例。

## 每个人创建自己的版本库

授权文件如下：

```
1 @administrators = jiangxin admin  
2  
3 repo    users/CREATOR/[a-zA-Z].*  
4       C    =  @all  
5       RW+   =  CREATOR  
6       RW    =  WRITERS  
7       R     =  READERS @administrators
```

关于授权的说明：

- 第4条指令，设置用户可以在自己的名字空间(`:file:`/usrs/<userid>/``)下，自己创建版本库。例如下面就是用户dev1执行`:command:`git push``命令在Gitolite服务器上自己的名字空间下创建版本库。

```
dev1$ git push git@server:users/dev1/repos1.git master
```

- 第5条指令，设置版本库创建者对版本库具有完全权限。

即用户dev1拥有对其自建的users/dev1/repos1.git拥有最高权限。

- 第7条指令，让管理员组@administrators的用户对于:file:`users/`下用户自建的版本库拥有读取权限。

那么第6、7条授权指令中出现的WRITERS和READERS是如何定义的呢？实际上这两个变量可以看做是两个用户组，不过这两个用户组不是在Gitolite授权文件中设置，而是由版本库创建者执行:command:`ssh`命令创建的。

版本库users/dev1/repos1.git的创建者dev1可以通过:command:`ssh`命令连接服务器，使用:command:`setperms`命令为自己的版本库设置角色。命令setperms的唯一一个参数就是版本库名称。当执行命令时，会自动进入一个编辑界面，手动输入角色定义后，按下^D (Ctrl+D) 结束编辑。如下所示：

```
dev1$ ssh git@server setperms users/dev1/repos1.git
READERS dev2 dev3
WRITERS jiangxin
^D
```

即在输入setperms指令后，进入一个编辑界面，输入^D (Ctrl+D) 结束编辑。也可以将角色定义文件保存到文件中，用:command:`setperms`指令加载。如下：

```
dev1$ cat > perms << EOF
READERS dev2 dev3
WRITERS jiangxin
EOF

dev1$ ssh git@server setperms users/dev1/repos1.git < perms
New perms are:
READERS dev2 dev3
WRITERS jiangxin
```

当版本库创建者dev1对版本库users/dev1/repos1.git进行了如上设置后，Gitolite在进行授权检查时会将setperms设置的角色定义应用到授权文件中。故此版本库users/dev1/repos1.git中又补充了新的授权：

- 用户dev2和dev3具有读取权限。
- 用户jiangxin具有读写权限。

版本库users/dev1/repos1.git的建立者dev1可以使用:command:`getperms`查看自己版本库的角色设置。如下：

```
dev1$ ssh git@server getperms users/dev1/repos1.git
READERS dev2 dev3
WRITERS jiangxin
```

如果在用户自定义授权中需要使用READERS和WRITERS之外的角色，管理员可以通过修改:file:`gitolite.rc`文件中的变量\$GL\_WILDREPOS\_PERM\_CATS实现。该变量的默认设置如下：

```
$GL_WILDREPOS_PERM_CATS = "READERS WRITERS";
```

## 传统模式的引用授权

传统模式的引用授权指的是在授权指令中只采用R、RW和RW+的传统授权关键字，而不包括后面介绍的扩展授权指令。传统的授权指令没有把分支的创建和分支删除权限细分，而是和写操作及强制推送操作混杂在一起。

- 非快进式推送必须拥有上述关键字中的+方可授权。
- 创建引用必须拥有上述关键字中的W方可授权。
- 删除引用必须拥有上述关键字中的+方可授权。
- 如果没有在授权指令中提供引用相关的参数，相当于提供refs/.\*作为引用的参数，意味着对所有引用均有效。

授权文件：

```

1 @administrators = jiangxin admin
2 @dev            = dev1 dev2 badboy
3 @test           = test1 test2
4
5 repo    test/repo1
6      RW+          = @administrators
7      RW master refs/heads/feature/ = @dev
8      R             = @test

```

关于授权的说明：

- 第6行，对于版本库test/repo1，管理员组用户jiangxin和admin可以读写任意分支、强制推送，以及创建和删除引用。
- 第7行，用户组@dev除了对master和refs/heads/feature/开头的引用具有读写权限外，实际上可以读取所有引用。这是因为读取操作授权阶段无法获知引用。
- 第8行，用户组@test对版本库拥有只读授权。

## 扩展模式的引用授权

扩展模式的引用授权，指的是该版本库的授权指令出现了下列授权关键字中的一个或多个：RWC、RWD、RWCD、RW+C、RW+D、RW+CD，将分支的创建权限和删除权限从读写权限中分离出来，从而可对分支进行更为精细的权限控制。

- 非快进式推送必须拥有上述关键字中的+方可授权。
- 创建引用必须拥有上述关键字中的C方可授权。
- 删除引用必须拥有上述关键字中的D方可授权。

即引用的创建和删除使用了单独的授权关键字，和写权限和强制推送权限分开。

下面是一个采用扩展授权关键字的授权文件：

```

1 repo    test/repo2
2      RW+C = @administrators
3      RW+  = @dev
4      RW   = @test
5
6 repo    test/repo3
7      RW+CD = @administrators
8      RW+C  = @dev
9      RW    = @test

```

通过上面的配置文件，对于版本库test/repo2.git具有如下的授权：

- 第2行，用户组@administrators中的用户，具有创建和删除引用的权限，并且能强制推送。

其中创建引用来自授权关键字中的C，删除引用来自授权关键字中的+，因为该版本库授权指令中没有出现D，因而删除应用授权沿用传统授权关键字。

- 第3行，用户组@dev中的用户，不能创建引用，但可以删除引用，并且可以强制推送。

因为第2行授权关键字中字符C的出现，使得创建引用采用扩展授权关键字，因而用户组@dev不具有创建引用的权限。

- 第4行，用户组@test中的用户，拥有读写权限，但是不能创建引用，不能删除引用，也不能强制推送。

通过上面的配置文件，对于版本库test/repo3.git具有如下的授权：

- 第7行，用户组@administators中的用户，具有创建和删除引用的权限，并且能强制推送。

其中创建引用来自授权关键字中的c，删除引用来自授权关键字中的D。

- 第8行，用户组@dev中的用户，可以创建引用，并能够强制推送，但不能删除引用。

因为第7行授权关键字中字符C和D的出现，使得创建和删除引用都采用扩展授权关键字，因而用户组@dev不具有删除引用的权限。

- 第9行，用户组@test中的用户，可以推送到任何引用，但是不能创建引用，不能删除引用，也不能强制推送。

## 禁用规则的使用

授权文件片段：

```
1   RW      refs/tags/v[0-9]      =  jiangxin
2   -       refs/tags/v[0-9]      =  @dev
3   RW      refs/tags/         =  @dev
```

关于授权的说明：

- 用户jiangxin可以创建任何里程碑，包括以v加上数字开头的版本里程碑。
- 用户组@dev，只能创建除了版本里程碑（以v加上数字开头）之外的其他里程碑。
- 其中以-开头的授权指令建立禁用规则。禁用规则只在授权的第二阶段有效，因此不能限制用户的读取权限。

## 用户分支

前面我们介绍过通过CREATOR特殊关键字实现用户自建版本库的功能。与之类似，Gitolite还支持在一个版本库中用户自建分支的功能。

用户在版本库中自建分支用到的关键字是USER而非CREATOR。即当授权指令的引用表达式中出现的USER关键字时，在授权检查时会动态替换为用户ID。例如授权文件片段：

```
1   repo    test/repo4
2           RW+CD          = @administrators
3           RW+CD refs/heads/u/USER/ = @all
4           RW+    master      = @dev
```

关于授权的说明：

- 第2行，用户组@administrators中的用户，对所有引用具有读写、创建和删除的权限，并且能强制推送。
- 第3行，所有用户都可以创建以u/<userid>/（含自己用户ID）开头的分支。对自己名字空间下的引用具有完全权限。对于他人名字空间的引用只有读取权限，不能修改。
- 第4行，用户组@dev对master分支具有读写和强制更新的权限，但是不能删除。

## 对路径的写授权

Gitolite也实现了对路径的写操作的精细授权，并且非常巧妙的是实现此功能所增加的代码可以忽略不计。这是因为Gitolite把路径当作是特殊格式的引用的授权。

在授权文件中，如果一个版本库的授权指令中的正则引用字段出现了以NAME/开头的引用，则表明该授权指令是针对路径进行的写授权，并且该版本库要进行基于路径的写授权判断。

示例：

```
1 repo foo
2     RW          = @junior_devs @senior_devs
3
4     RW NAME/      = @senior_devs
5     - NAME/Makefile = @junior_devs
6     RW NAME/      = @junior_devs
```

关于授权的说明：

- 第2行，初级程序员@junior\_devs和高级程序员@senior\_devs可以对版本库foo进行读写操作。
- 第4行，设定高级程序员@senior\_devs对所有文件（NAME/）进行写操作。
- 第5行和第6行，设定初级程序员@junior\_devs对除了根目录的:file:`Makefile`文件外的其他文件具有写权限。

## 创建和导入版本库

Gitolite维护的版本库默认位于安装用户主目录下的repositories目录中，即如果安装用户为git，则版本库都创建在:file:`/home/git/repositories`目录之下。可以通过配置文件:file:`.gitolite.rc`修改默认的版本库的根路径。

```
$REPO_BASE="repositories";
```

有多种创建版本库的方式。一种是在授权文件中用repo指令设置版本库（未使用正则表达式的版本库）的授权，当对gitolite-admin版本库执行:command:`git push`操作时，自动在服务端创建新的版本库。另外一种方式是在授权文件中用正则表达式定义的通配符版本库，不会即时创建（也不可能被创建），而是被授权的用户在远程创建后推送到服务器上完成创建。

## 在配置文件中出现的版本库，即时生成

尝试在授权文件:file:`conf/gitolite.conf`中加入一段新的版本库授权指令，而这个版本库尚不存在。新添加到授权文件中的内容为：

```
repo testing2
  RW+        = @all
```

然后将授权文件的修改提交并推送到服务器，会看到授权文件中添加新授权的版本库testing2被自动创建。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 375 bytes, done.
```

```
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
remote: creating testing2...
remote: Initialized empty Git repository in /home/git/repositories/testing2.git
To gitadmin.bj:gitolite-admin.git
 278e54b..b6f05c1  master -> master
```



注意其中带remote标识的输出，可以看到版本库testing2.git被自动初始化了。

此外使用版本库组的语法（即用@创建的组，用作版本库），也会被自动创建。例如下面的授权文件片段设定了一个包含两个版本库的组@testing，当将新配置文件推送到服务器上时，会自动创建:file:`testing3.git`和:file:`testing4.git`。

```
@testing = testing3 testing4

repo @testing
  RW+           = @all
```

## 通配符版本库，管理员通过推送创建

通配符版本库是用正则表达式语法定义的版本库，所指的并非某一个版本库而是和正则表达式相匹配的一组版本库。要想使用通配符版本库，需要在服务器端Gitolite的安装用户（如git）主目录下，修改配置文件:file:`.gitolite.rc`，使其包含如下配置：

```
$GL_WILDREPOS = 1;
```

使用通配符版本库，可以对一组版本库进行授权，非常有效。但是版本库的创建则不像前面介绍的那样，不会在授权文件推送到服务器时创建，而是由拥有版本库创建授权（C）的用户手工进行创建。

对于用通配符设置的版本库，用C指令指定能够创建此版本库的管理员（拥有创建版本库的授权）。例如：

```
repo ossxp/[a-z].+
  C           = jiangxin
  RW          = dev1 dev2
```

用户jiangxin可以创建路径符合正则表达式ossxp/[a-z].+的版本库，用户dev1和dev2对版本库具有读写（但是没有强制更新）权限。

- 本地建库。

```
$ mkdir somerepo
$ cd somerepo
$ git init
$ git commit --allow-empty
```

- 使用:command:`git remote`指令设置远程版本库。

```
jiangxin$ git remote add origin git@server:ossxp/somerepo.git
```

- 运行:command:`git push`完成在服务器端版本库的创建。

```
jiangxin$ git push origin master
```

使用该方法创建版本库后，创建者jiangxin的用户ID将被记录在版本库目录下的`:file:`gl-creater``文件中。该帐号具有对该版本库最高的权限。该通配符版本库的授权指令中如果出现关键字CREATOR将会用创建者的用户ID替换。

实际上Gitolite的原始实现是通过克隆即可创建版本库。即当克隆一个不存在的、名称匹配通配符版本库的、且拥有创建权限(C)，Gitolite会自动在服务器端创建该版本库。但是我认为这不是一个好的实践，会经常因为在克隆时把URL写错，从而导致在服务器端创建垃圾版本库。因此我重新改造了Gitolite通配符版本库创建的实现方法，使用推送操作实现版本库的创建，而克隆一个不存在的版本库会报错、退出。

## 向Gitolite中导入版本库

在Gitolite搭建时，已经存在并使用的版本库需要导入到Gitolite中。如果只是简单地把这些裸版本库（以.git为后缀不带工作区的版本库）复制到Gitolite的版本库根目录下，针对这些版本库的授权可能不能正常工作。这是因为Gitolite管理的版本库都配置了特定的钩子脚本，以实现基于分支和/或路径的授权，直接拷贝到Gitolite中的版本库没有正确地设置钩子脚本。而且Gitolite还利用版本库中的`:file:`gl-creater``记录版本库创建者，用`:file:`gl-perms``记录版本库的自定义授权，而这些也是拷贝过来的版本库不具备的。

对于少量的版本库，直接修改`:file:`gitolite-admin``的授权文件、添加同名的版本库授权、提交并推送，就会在Gitolite服务器端完成同名版本库的初始化。然后在客户端进入到相应版本库的工作区，执行`:command:`git push``命令将原有版本库的各个分支和里程碑导入到Gitolite新建的版本库中。

```
$ git remote add origin git@server:<repo-name>.git  
$ git push --all origin  
$ git push --tags origin
```

如果要导入的版本库较多，逐一在客户端执行`:command:`git push``操作很繁琐。可以采用下面的方法。

- 确认要导入所有版本库都以裸版本库形式存在（以.git为后缀，无工作区）。
- 将要导入的裸版本库复制到Gitolite服务器的版本库根目录中。
- 在客户端修改`gitolite-admin`授权文件，为每个导入的版本库添加授权。
- 推送对`gitolite-admin`版本库的修改，相应版本库的钩子脚本会自动进行设置。

如果版本库非常多，就连在`gitolite-admin`的授权文件中添加版本库授权也是难事，还可以采用下面的办法：

- 确认要导入所有版本库都以裸版本库形式存在（以.git为后缀，无工作区）。
- 将要导入的裸版本库复制到Gitolite服务器的版本库根目录中。
- 在服务器端，为每个导入的裸版本库下添加文件`:file:`gl-creater``，内容为版本库创建者ID。
- 在服务器端运行`:command:`gl-setup``程序（无需提供公钥参数），参见Gitolite安装相应章节。
- 在客户端修改`gitolite-admin`授权文件，以通配符版本库形式为导入的版本库进行授权。

## 对Gitolite的改进

Gitolite托管在GitHub上，任何人都可以基于原作者Sitaramc的工作进行定制。我对Gitolite的定制版本在<http://github.com/ossxp-com/gitolite>，包含的扩展和改进有：

- 通配符版本库的创建方式和授权。

原来的实现是克隆即创建（克隆者需要被授予c的权限）。同时还要通过另外的授权语句为用户设置RW权限，否则创建者没有读和写权限。

新的实现是通过推送创建版本库（推送者需要被授予c权限）。不必再为创建者赋予RW等权限，创建者自动具有对版本库最高的授权。

- 避免通配符版本库的误判。

若将通配符版本库误判为普通版本库名称，会导致在服务器端创建错误的版本库。新的设计可以在通配符版本库的正则表达式之前添加^或之后添加\$字符避免误判。

- 改变默认配置。

默认安装即支持通配符版本库。

- 版本库重定向。

Gitosis的一个很重要的功能——版本库名称重定向，没有在Gitolite中实现。我为Gitolite增加了这个功能。

在Git服务器架设的初期，版本库的命名可能非常随意，例如redmine的版本库直接放在根下：`:file:`redmine-0.9.x.git``、`:file:`redmine-1.0.x.git``，…随着redmine项目越来越复杂，可能就需要将其放在子目录下进行管理，例如放到`:file:`ossxp/redmine/``目录下。只需要在Gitolite的授权文件中添加下面一行map语句，就可以实现版本库名称的重定向。使用旧地址的用户不必重新检出，可以继续使用。

```
map (redmine.*) = ossxp/redmine/$1
```

## Gitolite功能拓展

### 版本库镜像

Git版本库控制系统往往并不需要设计特别的容灾备份，因为每一个Git用户就是一个备份。但是下面的情况，就很有必要考虑容灾了。

- Git版本库的使用者很少（每个库可能只有一个用户）。
- 版本库克隆只限制在办公区并且服务器也在办公区内（所有鸡蛋都在一个篮子里）。
- Git版本库采用集中式的应用模型，需要建立双机热备（以便在故障出现时，实现快速的服务器切换）。

可以在两台或多台安装了Gitolite服务的服务器之间实现版本库的镜像。数据镜像的最小单位为版本库，对于任意一个Git版本库可以选择在其中一个服务器上建立主版本库（只能有一个主版本库），在其他服务器上建立的为镜像库。镜像库只接受来自主版本库的数据同步而不接受来自用户的推送。

### Gitolite服务器命名

首先要为每一台服务器架设Gitolite服务，并建议所有的服务器上Gitolite服务都架设在同一用户（如git）之下。如果Gitolite服务安装到不同的用户账号下，就必需通过文件`:file:`~/.ssh/config``建立SSH别名，以便能够使用正确的用户名连接服务器。

接下来为每个服务器设置一个名称，服务器之间数据镜像时就使用各自的名称进行连接。假设我们要配置的两个Gitolite服务器的其中一个名为server1，另一个名为server2。

打开server1上Gitolite的配置文件`:file:`~/.gitolite.rc``，进行如下设置：

```
$GL_HOSTNAME = 'server1';
```

```
$GL_GITCONFIG_KEYS = "gitolite.mirror.*";
```

- 设置\$GL\_HOSTNAME为本服务器的别名，如server1。
- 设置\$GL\_GITCONFIG\_KEYS以便允许在Gitolite授权文件中为版本库动态设置配置变量。

例如本例设置了\$GL\_GITCONFIG\_KEYS为gitolite.mirror.\*后，允许在gitolite-admin管理库的:file:`conf/gitolite.conf`中用config指令对版本库添加配置变量。

```
repo testing
  config gitolite.mirror.master      = "server1"
  config gitolite.mirror.slaves     = "server2 server3"
```

同样对server2进行设置，只不过将\$GL\_HOSTNAME设置为server2。

## 服务器之间的公钥认证

接下来每一个服务器为Gitolite的安装用户创建公钥/私钥对。

```
$ sudo su - git
$ ssh-keygen
```

然后把公钥拷贝到其他服务器上，并以本服务器名称命名。例如：

- server1上创建的公钥复制到server2上，命名为:file:`server1.pub`备用。
- server2上创建的公钥复制到server1上，命名为:file:`server2.pub`备用。

再运行:command:`gl-tool`设置其他服务器到本服务器上的公钥认证。例如在server1上执行命令：

```
$ gl-tool add-mirroring-peer server2.pub
```

当完成上述设置后，就可以从一个服务器发起到另外服务器的SSH连接，连接过程无需口令认证并显示相关信息。例如从server1发起到server2的连接如下：

```
$ ssh git@server2 info
Hello server1, I am server2
```

## 配置版本库镜像

做了前面的准备工作后，就可以开始启用版本库镜像了。下面通过一个示例介绍如何建立版本库镜像，将服务器server1上的版本库testing要镜像到服务器server2上。

首先要修改server1和server2的Gitolite管理库gitolite-admin，为testing版本库添加配置变量，如下：

```
repo    testing
  config gitolite.mirror.master = "server1"
  config gitolite.mirror.slaves = "server2"
```

两个服务器server1和server2都要做出同样的修改，提交改动并推送到服务器上。当推送完成，两个服务器上的testing版本库的:file:`config`就会被更新，包含类似如下的设置：

```
[gitolite "mirror"]
```

```
--  
master = server1  
slaves = server2
```

当向服务器server1的testing版本库推送新的提交时，就会自动同步到server2上。

```
$ git push git@server1:testing.git master  
[master c0b097a] test  
Counting objects: 1, done.  
Writing objects: 100% (1/1), 185 bytes, done.  
Total 1 (delta 0), reused 0 (delta 0)  
remote: (29781&) server1 === (testing) ===> server2  
To git@server1:testing.git  
d222699..c0b097a master -> master
```

如果需要将服务器server1上所有版本库，包括gitolite-admin版本库都同步到server2上，不必对版本库逐一设置，可以采用下面的简便方法。

修改server1和server2的Gitolite管理版本库gitolite-admin，在配置文件:`file: `conf/gitolite.conf``最开始插入如下设置。

```
repo @all  
config gitolite.mirror.master = "server1"  
config gitolite.mirror.slaves = "server2"
```

然后分别提交并推送。要说明的是gitolite-admin版本库此时尚未建立同步，直到服务器server1的gitolite-admin版本库推送新的提交，才开始gitolite-admin版本库的同步。

也可以在server1服务器端执行命令开始同步。例如：

```
$ gl-mirror-shell request-push gitolite-admin
```

Gitolite官方版本在版本库同步时有个局限，要求在镜像服务器上必需事先存在目标版本库并正确设置了gitolite.mirror.\*参数，才能同步成功。例如允许用户自行创建的通配符版本库，必需在主服务器上和镜像服务器上分别创建，之后版本库同步才能正常执行。我在GitHub上的Gitolite分支项目提交了一个补丁解决了这个问题。

关于Gitolite版本库镜像的更详悉资料，参见<http://sitaramc.github.com/gitolite/doc/mirroring.html>。

## Gitweb和Git daemon支持

Gitolite和git-daemon的整合很简单，就是由Gitolite创建的版本库会在版本库目录中创建一个空文件:`file: `git-daemon-export-ok``。

Gitolite和Gitweb的整合则提供了两个方面的内容。一个是可以设置版本库的描述信息，用于在Gitweb的项目列表页面中显示。另外一个是自动生成项目的列表文件供Gitweb参考，避免Gitweb使用低效率的目录递归搜索查找Git版本库列表。

可以在授权文件中设定版本库的描述信息，并在gitolite-admin管理库更新时创建到版本库的:`file: `description``文件中。

```
reponame = "one line of description"  
reponame "owner name" = "one line of description"
```

- 第1行，为名为reponame的版本库设定描述。
- 第2行，同时设定版本库的属主名称，以及一行版本库描述。

对于通配符版本库，使用这种方法则很不现实。Gitolite提供了SSH子命令供版本库的创建者使用。

```
$ ssh git@server setdesc path/to/repos.git  
$ ssh git@server getdesc path/to/repos.git
```

- 第一条指令用于设置版本库的描述信息。
- 第二条指令显示版本库的描述信息。

至于生成Gitweb所用的项目列表文件，默认创建在用户主目录下的`:file:`projects.list``文件中。对于所有启用Gitweb的[repo]小节所设定的版本库，以及通过版本库描述隐式声明的版本库都会加入到版本库列表中。

## 其他功能拓展和参考

Gitolite源码的`:file:`doc``目录包含用markdown标记语言编写的手册，可以直接在Github上查看。也可以使用markdown的文档编辑工具将`:file:`.mkd``文档转换为html文档。转换工具很多，有`:command:`rdiscount``、`:command:`Bluefeather``、`:command:`Maruku``、`:command:`BlueCloth2``，等等。

在这些参考文档中，用户可以发现Gitolite包含的更多的小功能或秘籍，包括：

- 版本库设置。

授权文件通过`:command:`git config``指令为版本库进行附加的设置。例如：

```
repo gitolite  
  config hooks.mailinglist = gitolite-commits@example.tld  
  config hooks.emailprefix = "[gitolite] "  
  config foo.bar = ""  
  config foo.baz =
```

- 多级管理员授权。

可以为不同的版本库设定管理员，操作`gitolite-admin`库的部分授权文件。具体参考：`:file:`doc/5-delegation.mkd``。

- 自定义钩子脚本。

因为Gitolite占用了几个钩子脚本，如果需要对同名钩子进行扩展，Gitolite提供了级联的钩子脚本，将定制放在级联的钩子脚本里。

例如：通过自定义`gitolite-admin`的`:file:`post-update.secondary``脚本，以实现无须登录服务器即可更改`:file:`.gitolite.rc``文件。具体参考：`:file:`doc/shell-games.mkd``。

关于钩子脚本的创建和维护，具体参考：`:file:`doc/hook-propagation.mkd``。

- 管理员自定义命令。

通过设置配置文件中的`$GL_ADC_PATH`变量，在远程执行该目录下的可执行脚本，如：`:command:`rmrepo``。

具体参考：`:file:`doc/admin-defined-commands.mkd``。

- 创建匿名的SSH认证。

允许匿名用户访问Gitolite提供的Git服务。即建立一个和Gitolite服务器端帐号同ID同主目录的用户，设置其的特定shell，并且允许口令为空。

具体参考: :file:`doc/mob-branches.mkd`。

- 可以通过名为@all的版本库进行全局的授权。

但是不能在@all版本库中对@all用户组进行授权。

- 版本库或用户非常之多（几千个）的时候，需要使用大配置文件模式。

因为Gitolite的授权文件要先编译才能生效，而编译文件的大小是和用户及版本库数量的乘积成正比的。选择大配置文件模式则不对用户组和版本库组进行扩展。

具体参考: :file:`doc/big-config.mkd`。

- 授权文件支持包含语句，可以将授权文件分成多个独立的单元。
- 执行外部命令，如:command:`rsync`。
- Subversion版本库支持。

如果在同一个服务器上以svn+ssh方式运行Subversion服务器，可以使用同一套公钥，同时为用户提供Git和Subversion服务。

- HTTP口令文件维护。通过名为htpasswd的SSH子命令实现。

[1]	对Gitolite的各项改动采用了Topgit特性分支进行维护，以便和上游最新代码同步更新。还要注意如果在Gitolite使用中发现问题，要区分是由上游软件引发的还是我的改动引起的，而不要把我的错误算在Sitaram头上。
[2]	公钥的内容为一整行，因排版需要做了换行处理。
[3]	可以为版本库设置配置变量gitolite-options.deny-repo在第一个授权阶段启用禁用规则检查。
[4]	参见第8部分41.2.2“Git模板”相关内容。 来源: <a href="https://github.com/gotgit/gotgit/blob/master/05-git-server/040-gitolite.rst">https://github.com/gotgit/gotgit/blob/master/05-git-server/040-gitolite.rst</a>

# Gitosis服务架设

Gitosis是Gitolite的鼻祖，同样也是一款基于SSH公钥认证的Git服务管理工具，但是功能要比之前介绍的Gitolite要弱的多。Gitosis由Python语言开发，对于偏爱Python不喜欢Perl的开发者（我就是其中之一），可以对Gitosis加以关注。

Gitosis的出现远早于Gitolite，作者Tommi Virtanen从2007年5月就开始了gitosis的开发，最后一次提交是在2009年9月，已经停止更新了。但是Gitosis依然有其生命力。

- 配置简洁，可以直接在服务器端编辑，成为某些服务定制的内置的无需管理的Git服务。

Gitosis的配置文件非常简单，直接保存于服务安装用户（如git）的主目录下：`:file:`.gitosis.conf`` 文件中，可以直接在服务器端创建和编辑。

Gitolite的授权文件需要复杂的编译，因此一般需要管理员克隆`gitolite-admin`库，远程编辑并推送至服务器。因此用Gitolite实现一个无需管理的Git服务难度要大很多。

- 支持版本库重定向。

版本库重定向一方面在版本库路径变更后保持旧的URL仍可工作，另一方面用在客户端用简洁的地址屏蔽服务器端复杂的地址。

例如我开发的一款备份工具（Gistore），版本库位于：`:file:`/etc/gistore/tasks/system/repo.git``（符号链接），客户端使用`system.git`即映射到复杂的服务器端地址。

这个功能我已经在定制的Gitolite中实现。

- Python语言开发，对于喜欢Python，不喜欢Perl的用户，可以选择Gitosis。
- 在Github上有很多Gitosis的克隆，我对gitosis的改动放在了github上：

<http://github.com/ossxp-com/gitosis>

Gitosis因为是Gitolite的鼻祖，因此下面的Gitosis实现机理，似曾相识：

- Gitosis安装在服务器（`server.name`）某个帐号之下，例如`git`帐号。
- 管理员通过Git命令检出名为`gitosis-admin`的版本库。

```
$ git clone git@server.name:gitosis-admin.git
```

- 管理员将git用户的公钥保存在gitosis-admin库的:`:file:`keydir``目录下，并编辑:`:file:`gitosis.conf``文件为用户授权。
- 当管理员对gitosis-admin库的修改提交并PUSH到服务器之后，服务器上gitosis-admin版本库的钩子脚本将执行相应的设置工作。
  - 新用户公钥自动追加到服务器端安装帐号的:`:file:`.ssh/authorized_keys``中，并设置该用户的shell为gitosis的一条命令:`:command:`gitosis-serve``。

```
command="gitosis-serve jiangxin",no-port-forwarding,no-X11-forwarding
```



- 更新服务器端的授权文件:`:file:`~/.gitosis.conf``。
- 用户可以用Git命令访问授权的版本库。
- 当管理员授权，用户可以远程在服务器上创建新版本库。

下面介绍Gitosis的部署和使用。在下面的示例中，约定：服务器的名称为server，Gitolite的安装帐号为git。

## 安装Gitosis

Gitosis的部署和使用可以直接参考源代码中的:`:file:`README.rst``。可以直接访问Github上我的gitosis克隆，因为Github能够直接将rst文件显示为网页。

参考：

```
http://github.com/ossxp-com/gitosis/blob/master/README.rst
```

## Gitosis的安装

Gitosis安装需要在服务器端执行。下面介绍直接从源代码进行安装，以便获得最新的改进。

Gitosis的官方Git库位于`git://eagain.net/gitosis.git`。我在Github上创建了一个分支：

```
http://github.com/ossxp-com/gitosis
```

- 使用git下载Gitosis的源代码。

```
$ git clone git://github.com/ossxp-com/gitosis.git
```

- 进入`:file:`gitosis``目录，执行安装。

```
$ cd gitosis  
$ sudo python setup.py install
```

- 可执行脚本安装在`:file:`/usr/local/bin``目录下。

```
$ ls /usr/local/bin/gitosis-*  
/usr/local/bin/gitosis-init  /usr/local/bin/gitosis-run-hook  /usr/local/
```

## 服务器端创建专用帐号

安装Gitosis，还需要在服务器端创建专用帐号，所有用户都通过此帐号访问Git库。一般为方便易记，选择git作为专用帐号名称。

```
$ sudo adduser --system --shell /bin/bash --disabled-password --group it
```

创建用户git，并设置用户的shell为可登录的shell，如`/bin/bash`，同时添加同名的用户组。

有的系统，只允许特定的用户组（如ssh用户组）的用户才可以通过SSH协议登录，这就需要将新建的git用户添加到ssh用户组中。

```
$ sudo adduser git ssh
```

## Gitosis服务初始化

Gitosis服务初始化，就是初始化一个gitosis-admin库，并为管理员分配权限，还要将Gitosis管理员的公钥添加到专用帐号的`:file:`~/.ssh/authorized_keys``文件中。

- 如果管理员在客户端没有公钥，使用下面命令创建。

```
$ ssh-keygen
```

- 管理员上传公钥到服务器。

```
$ scp ~/.ssh/id_rsa.pub server:/tmp
```

- 服务器端进行Gitosis服务初始化。

以git用户身份执行gitosis-init命令，并向其提供管理员公钥。

```
$ sudo su - git  
$ gitosis-init < /tmp/id_rsa.pub
```

- 确保gitosis-admin版本库的钩子脚本可执行。

```
$ sudo chmod a+x ~git/repositories/gitosis-admin.git/hooks/post-update
```

## 管理Gitosis

### 管理员克隆gitosis-admin管理库

当Gitosis安装完成后，在服务器端自动创建了一个用于Gitosis自身管理的Git库：gitosis-admin.git。

管理员在客户端克隆gitosis-admin.git库，注意要确保认证中使用正确的公钥：

```
$ git clone git@server:gitosis-admin.git  
$ cd gitosis-admin/  
  
$ ls -F  
gitosis.conf  keydir/  
  
$ ls keydir/  
jiangxin.pub
```

可以看出gitosis-admin目录下有一个陪孩子文件和一个目录：file:`keydir`。

- :file:`keydir/jiangxin.pub` 文件

:file:`keydir` 目录下初始时只有一个用户公钥，即管理员的公钥。管理员的用户名来自公钥文件末尾的用户名。

- :file:`gitosis.conf` 文件

该文件为授权文件。初始内容为：

```
1 [gitosis]
2
3 [group gitosis-admin]
4 writable = gitosis-admin
5 members = jiangxin
```

可以看到授权文件的语法完全不同于之前介绍的Gitolite的授权文件。整个授权文件是以用户组为核心，而非版本库为核心。

- 定义了一个用户组gitosis-admin。

第3行开始定义了一个用户组gitosis-admin。

- 第5行设定了该用户组包含的用户列表。

初始时只有一个用户，即管理员公钥所属的用户。

- 第4行设定了该用户组对那些版本库具有写操作。

这里配置对gitosis-admin版本库具有写操作。写操作自动包含了读操作。

## 增加新用户

增加新用户，就是允许新用户能够通过其公钥访问Git服务。只要将新用户的公钥添加到gitosis-admin版本库的:`:file:`keydir``目录下，即完成新用户的添加。

- 管理员从用户获取公钥，并将公钥按照`:file:`username.pub``格式进行重命名。

用户可以通过邮件或者其他方式将公钥传递给管理员，切记不要将私钥误传给管理员。如果发生私钥泄漏，马上重新生成新的公钥/私钥对，并将新的公钥传递给管理员，并申请将旧的公钥作废。

关于公钥名称，我引入了类似Gitolite的实现：

- 用户从不同的客户端主机访问有着不同的公钥，如果希望使用同一个用户名进行授权，可以按照`username@host.pub`方式命名公钥文件，和名为`username@pub`的公钥指向同一个用户`username`。
  - 也支持邮件地址格式的公钥，即形如`username@gmail.com.pub`的公钥。Gitosis能够很智能的区分是以邮件地址命名的公钥还是相同用户在不同主机上的公钥。如果是邮件地址命名的公钥，将以整个邮件地址作为用户名。
- 管理员进入gitosis-admin本地克隆版本库中，复制新用户公钥到`:file:`keydir``目录。

```
$ cp /path/to/dev1.pub keydir/
$ cp /path/to/dev2.pub keydir/
```

- 执行`:command:`git add``命令，将公钥添加入版本库。

```
$ git add keydir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   keydir/dev1.pub
#       new file:   keydir/dev2.pub
#
```

- 执行`:command:`git commit``，完成提交。

```
$ git commit -m "add user: dev1, dev2"
[master d7952a5] add user: dev1, dev2
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 keydir/dev1.pub
 create mode 100644 keydir/dev2.pub
```

- 执行`:command:`git push``，同步到服务器，才真正完成新用户的添加。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.03 KiB, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@server:gitosis-admin.git
 2482e1b..d7952a5  master -> master
```

如果这时查看服务器端`:file:`~git/.ssh/authorized_keys``文件，会发现新增的用户公钥也附加其中：

```
### autogenerated by gitosis, DO NOT EDIT
command="gitosis-serve jiangxin",no-port-forwarding,no-X11-forwarding,no-agent-fo
command="gitosis-serve dev1",no-port-forwarding,no-X11-forwarding,no-agent-fo
command="gitosis-serve dev2",no-port-forwarding,no-X11-forwarding,no-agent-fo
```

## 更改授权

新用户添加完毕，可能需要重新进行授权。更改授权的方法也非常简单，即修改:file:`gitosis.conf`配置文件，提交并推送。

首先管理员进入gitosis-admin本地克隆版本库中，编辑:file:`gitosis.conf`。

```
$ vi gitosis.conf
```

授权指令比较复杂，先通过建立一个新用户组并授权新版本库testing尝试一下更改授权文件。例如在:file:`gitosis.conf`中添加如下授权内容：

```
1 [group testing-admin]
2 members = jiangxin @gitosis-admin
3 admin = testing
4
5 [group testing-developer]
6 members = dev1 dev2
7 writable = testing
8
9 [group testing-reader]
10 members = @all
11 readonly = testing
```

- 上面的授权文件为版本库testing赋予了三个角色。分别是@testing-admin用户组，@testing-developer用户组和@testing-reader用户组。
- 第1行开始的testing-admin小节，定义了用户组@testing-admin。
- 第2行设定该用户组包含的用户有jiangxin，以及前面定义的@gitosis-admin用户组用户。
- 第3行用admin指令，设定该用户组用户可以创建版本库testing。

admin指令是笔者新增的授权指令，请确认安装的Gitosis包含笔者的改进。

- 第7行用writable授权指令，设定该@testing-developer用户组用户可以读写版本库testing。

笔者改进后的Gitosis也可以使用write作为writable指令的同义词指令。

- 第11行用readonly授权指令，设定该@testing-reader用户组用户（所有用户）可以只读访问版本库testing。

笔者改进后的Gitosis也可以使用read作为readonly指令的同义词指令。

编辑结束，提交改动。

```
$ git add gitosis.conf  
$ git commit -q -m "auth for repo testing."
```

执行:command:`git push`，同步到服务器，才真正完成授权文件的编辑。

```
$ git push
```

## Gitosis授权详解

### Gitosis缺省设置

在[gitosis]小节中定义Gitosis的缺省设置。如下：

```
1 [gitosis]  
2 repositories = /gitroot  
3 #loglevel=DEBUG  
4 gitweb = yes  
5 daemon = yes  
6 generate-files-in = /home/git/gitosis
```

其中：

- 第2行，设置版本库缺省的根目录是:file:`/gitroot` 目录。

否则缺省路径是安装用户主目录下的:file:`repositories` 目录。

- 第3行，如果打开注释，则版本库操作时显示Gitosis调试信息。
- 第4行，启用gitweb的整合。

可以通过[repo name]小节为版本库设置描述字段，用户显示在Gitweb中。

- 第5行，启用git-daemon的整合。  
即新创建的版本库中，创建文件:file:`git-daemon-export-ok`。
- 第6行，设置创建的项目列表文件（供gitweb使用）所在的目录。  
缺省即为安装用户的主目录下的:file:`gitosis` 目录。

## 管理版本库gitosis-admin

```
1 [group gitosis-admin]
2 write = gitosis-admin
3 members = jiangxin
4 repositories = /home/git
```

除了第4行，其他内容在前面都已经介绍过了，是Gitosis自身管理版本库的用户组设置。

第4行，重新设置了版本库的缺省根路径，覆盖缺省的[gitosis]小节中的缺省根路径。实际的gitosis-admin版本库的路径为:file:`/home/git/gitosis-admin.git`。

## 定义用户组和授权

下面的两个示例小节定义了两个用户组，并且用到了路径变换的指令。

```
1 [group ossxp-admin]
2 members = @gitosis-admin jiangxin
3 admin = ossxp/**
4 read = gistore/*
5 map admin redmine-* = ossxp/redmine/\1
6 map admin ossxp/redmine-* = ossxp/(redmine-.*):ossxp/redmine/\1
7 map admin ossxp/testlink-* = ossxp/(testlink-.*):ossxp/testlink/\1
8 map admin ossxp/docbones* = ossxp/(docbones.*):ossxp/docutils/\1
9
10 [group all]
11 read = ossxp/**
12 map read redmine-* = ossxp/redmine/\1
13 map read testlink-* = ossxp/testlink/\1
14 map read pysvnmanager-gitsvn = mirrors/pysvnmanager-gitsvn
15 map read ossxp/redmine-* = ossxp/(redmine-.*):ossxp/redmine/\1
16 map read ossxp/testlink-* = ossxp/(testlink-.*):ossxp/testlink/\1
17 map read ossxp/docbones* = ossxp/(docbones.*):ossxp/docutils/\1
18 repositories = /gitroot
```

在上面的示例中，演示了授权指令以及Gitosis特色的map指令。

- 第1行，定义了用户组@ossxp-admin。
- 第2行，设定该用户组包含用户jiangxin以及用户组@gitosis-admin的所有用户。
- 第3行，设定该用户组具有创建及读写与通配符ossxp/\*\*匹配的版本库。

两个星号匹配任意字符包括路径分隔符（/）。此功能属于笔者扩展的功能。

- 第4行，设定该用户组可以只读访问`gistore/*`匹配的版本库。

一个星号匹配任意字符包括路径分隔符（/）。此功能也属于笔者扩展的功能。

- 第5行，是Gitosis特有的版本库名称重定位功能。

即对`redmine-*`匹配的版本库，先经过名称重定位，在名称前面加上`ossxp/redmine`。其中`\1`代表匹配的整个版本库名称。

用户组`@ossxp-admin`的用户对于重定位后的版本库，具有`admin`（创建和读写）权限。

- 第6行，是我扩展的版本库名称重定位功能，支持正则表达式。

等号左边的名称进行通配符匹配，匹配后，再经过右侧的一对正则表达式进行转换（冒号前的用于匹配，冒号后的用于替换）。

- 第10行，使用了内置的`@all`用户组，因此不需要通过`members`设定用户，因为所有用户均属于该用户组。
- 第11行，设定所有用户均可以只读访问`ossxp/**`匹配的版本库。
- 第12-17行，对特定路径进行映射，并分配只读权限。
- 第18行，设置版本库的根路径为`:file:`/gitroot``，而非缺省的版本库根路径。

## Gitweb整合

Gitosis和Gitweb的整合，提供了两个方面的内容。一个是可以设置版本库的描述信息，用于在gitweb的项目列表页面显示。另外一个是自动生成项目的列表文件供Gitweb参考，避免Gitweb使用效率低的目录递归搜索查找Git版本库列表。

例如在`:file:`gitosis.conf``中下面的配置用于对`redmine-1.0.x`版本库的Gitweb整合进行设置。

```
1 [repo ossxp/redmine/redmine-1.0.x]
2 gitweb = yes
3 owner = Jiang Xin
4 description = Redmine 1.0.x 群英汇定制开发
```

- 第1行，`repo`小节用于设置版本库的Gitweb整合。

版本库的实际路径是用版本库缺省的根（即在`[gitosis]`小节中定义的或者缺省的）加上此小节中的版本库路径组合而成的。

- 第2行，启用Gitweb整合。如果省略，使用全局[gitosis]小节中gitweb的设置。
- 第3行，用于设置版本库的属主。
- 第4行，用于设置版本库的描述信息，显示在Gitweb的版本库列表中。

每一个repo小节所指向的版本库，如果启用了Gitweb选项，则版本库名称汇总到一个项目列表文件中。该项目列表文件缺省保存在:`:file:`~/.gitosis/projects.list``中。

## 创建新版本库

Gitosis维护的版本库位于安装用户主目录下的`:file:`repositories``目录中，即如果安装用户为git，则版本库都创建在`:file:`/home/git/repositories``目录之下。可以通过配置文件`:file:`gitosis.conf``修改缺省的版本库的根路径。

可以直接在服务器端创建，或者在客户端远程创建版本库。

### 克隆即创建，还是PUSH即创建？

在客户端远程创建版本库时，Gitosis的原始实现是对版本库具有`writable`（读写）权限的用户，对不存在的版本库执行克隆操作时，自动创建。但是我认为这不是一个好的实践，会经常因为克隆的URL写错，导致在服务器端创建垃圾版本库。笔者改进的实现如下：

- 增加了名为`admin`（或`init`）的授权指令，只有具有此授权的用户，才能够创建版本库。
- 只具有`writable`（或`write`）权限的用户，不能在服务器上创建版本库。
- 不通过克隆创建版本库，而是在对版本库进行PUSH的时候进行创建。当克隆一个不存在的版本库，会报错退出。

远程在服务器上创建版本库的方法如下：

- 首先，本地建库。

```
$ mkdir somerepo  
$ cd somerepo  
$ git init  
$ git commit --allow-empty
```

- 使用`:command:`git remote``指令添加远程的源。

```
$ git remote add origin git@server:osssxp/somerepo.git
```

- 运行`:command:`git push``完成在服务器端版本库的创建

```
$ git push origin master
```

## 轻量级管理的Git服务

轻量级管理的含义是不采用缺省的稍显复杂的管理模式（远程克隆gitosis-admin库，修改并PUSH的管理模式），而是直接在服务器端通过预先定制的配置文件提供Git服务。这种轻量级管理模式，对于为某些应用建立快速的Git库服务提供了便利。

例如在使用备份工具Gistore进行文件备份时，可以用Gitosis架设轻量级的Git服务，可以在远程使用Git命令进行双机甚至是异地备份。

首先创建一个专用帐号，并设置该用户只能执行:command:`gitosis-serve`命令。例如创建帐号gistore，通过修改:command:`/etc/ssh/sshd\_config`配置文件，实现限制该帐号登录的可执行命令。

```
Match user gistore
  ForceCommand gitosis-serve gistore
  X11Forwarding no
  AllowTcpForwarding no
  AllowAgentForwarding no
  PubkeyAuthentication yes
  #PasswordAuthentication no
```

上述配置信息告诉SSH服务器，凡是以gistore用户登录的帐号，强制执行Gitosis的命令。

然后，在该用户的主目录下创建一个配置文件:file:`.gitosis.conf`（注意文件名前面的点号），如下：

```
[gitosis]
repositories = /etc/gistore/tasks
gitweb = yes
daemon = no

[group gistore]
members = gistore
map readonly * = (.*):\1/repo
```

上述配置的含义是：

- 用户gistore才能够访问:file:`/etc/gistore/tasks`下的Git库。
- 版本库的名称要经过变换，例如system库会变换为实际路

径:`file:/etc/gistore/tasks/system/repo.git`。`

来源: <https://github.com/gotgit/gotgit/blob/master/05-git-server/050-gitosis.rst>

## Gerrit代码审核服务器

谷歌Android开源项目在Git的使用上有两个重要的创新，一个是为了多版本库协同而引入的repo，在前面第25章已经详细讨论过。另外一个重要的创新就是Gerrit——代码审核服务器。Gerrit为Git引入的代码审核是强制性的，就是说除非特别的授权设置，向Git版本库的推送（Push）必须要经过Gerrit服务器，修订必须经过代码审核的一套工作流之后，才可能经批准并纳入正式代码库中。

首先贡献者的代码通过Git命令（或repo封装）推送到Gerrit管理下的Git版本库，推送的提交转化为一个一个的代码审核任务，审核任务可以通过refs/changes/下的引用访问到。代码审核者可以通过Web界面查看审核任务、代码变更，通过Web界面做出通过代码审核或者打回等决定。测试者也可以通过refs/changes/之下的引用获取（fetch）修订对其进行测试，如果测试通过就可以将该评审任务设置为校验通过（verified）。最后经过了审核和校验的修订可以通过Gerrit界面中提交动作合并到版本库对应的分支中。

Android项目网站上有一个代码贡献流程图，详细的介绍了Gerrit代码审核服务器的工作流程。翻译后的工作流程图见图32-1。



图32-1：Gerrit代码审核工作流

## Gerrit的实现原理

Gerrit更准确的说应该称为Gerrit2。因为Android项目最早使用的评审服务器Gerrit不是今天这个样子，最早版本的Gerrit是用Python开发运行于Google App Engine上，是从Python之父Guido van Rossum开发的Rietveld分支而来。在这里要讨论的Gerrit实为Gerrit2，是用Java语言实现的。从这里（<http://code.google.com/p/gerrit/wiki/Background>）可以看到Gerrit更为详尽的发展历史。

## SSH协议的Git服务器

Gerrit本身基于SSH协议实现了一套Git服务器，这样就可以对Git数据推送进行更为精确的控制，为强制审核的实现建立了基础。

Gerrit提供的Git服务的端口并非标准的22端口，缺省是29418端口。可以访问Gerrit的Web界面得到这个端口。对Android项目的代码审核服务器，访问[https://review.source.android.com/ssh\\_info](https://review.source.android.com/ssh_info)就可以查看到Git服务的服务器域名和开放的端口。下面用：command: curl 命令查看网页的输出。

```
$ curl -L -k http://review.source.android.com/ssh_info
review.source.android.com 29418
```

## 特殊引用refs/for/<branch-name>和refs/changes/n<n>/<review-id>/m

Gerrit的Git服务器，禁止用户向refs/heads命名空间下的引用执行推送（除非特别的授权），即不允许用户直接向分支进行提交。为了让开发者能够向Git服务器提交修订，Gerrit的Git服务器只允许用户向特殊的引用refs/for/<branch-name>下执行推送，其中<branch-name>即为开发者的工作分支。向refs/for/<branch-name>命名空间下推送并不会在其中创建引用，而是为新的提交分配一个ID，称为review-id，并为该review-id的访问建立如下格式的引用refs/changes/n<n>/<review-id>/m，其中：

- review-id为Gerrit为评审任务顺序分配的全局唯一的号码。
- nn为review-id的后两位数，位数不足用零补齐。即nn为review-id除以100的余数。
- m为修订号，该review-id的首次提交修订号为1，如果该修订被打回，重新提交修订号会自增。

### Git库的钩子脚本:`:file:`hooks/commit-msg``

为了保证已经提交审核的修订通过审核入库后，被别的分支拣选（cherry-pick）后再推送至服务器时不会产生新的重复的评审任务，Gerrit设计了一套方法，即要求每个提交包含唯一的Change-Id，这个Change-Id因为出现在日志中，当执行拣选时也会保持，Gerrit一旦发现新的提交包含了已经处理过的change-Id，就不再为该修订创建新的评审任务和review-id，而直接将提交入库。

为了实现Git提交中包含唯一的Change-Id，Gerrit提供了一个钩子脚本，放在开发者本地Git库中（hooks/commit-msg）。这个钩子脚本在用户提交时自动在提交说明中创建以“Change-Id:”及包含`:command:`git hash-object``命令产生的哈希值的唯一标识。当Gerrit获取到用户向`refs/for/<branch-name>`推送的提交中包含“Change-Id: I...”的变更ID，如果该Change-Id之前没有见过，会创建一个新的评审任务并分配新的review-id，并在Gerrit的数据库中保存Change-Id和review-id的关联。

如果当用户的提交因为某种原因被要求打回重做，开发者修改之后重新推送到Gerrit时就要注意在提交说明中使用相同的“Change-Id”（使用--amend提交即可保持提交说明），以免创建新的评审任务，还要在推送时将当前分支推送到`refs/changes/nm/review-id/m`中。其中nn和review-id和之前提交的评审任务的修订相同，m则要人工选择一个新的修订号。

以上说起来很复杂，但是在实际操作中只要使用repo这一工具，就相对容易多了。

## 其余一切交给Web

Gerrit另外一个重要的组件就是Web服务器，通过Web服务器实现对整个评审工作流的控制。关于Gerrit工作流，参见在本章开头出现的Gerrit工作流程图。

感受一下Gerrit的魅力？直接访问Android项目的Gerrit网站：<https://review.source.android.com/>。会看到如图32-2的界面。



图32-2：Android项目代码审核网站

Android项目的评审网站，匿名即可访问。点击菜单中的“Merged”显示了已经通过评审合并到代码库中的审核任务。图32-3中显示的就是Andorid一个已经合并到代码库中的历史评审任务。



图32-3：Android项目的16993号评审

从图32-3可以看出：

- URL中显示的评审任务编号为16993。
- 该评审任务的Change-Id以字母I开头，包含了一个唯一的40位SHA1哈希。
- 整个评审任务有三个人参与，一个人进行了检查（verify），两个人进行了代码审核。
- 该评审任务的状态为已合并：“merged”。
- 该评审任务总共包含两个补丁集：Patch set 1和Patch set 2。
- 补丁集的下载方法是：`:command:`repo download platform/sdk 16993/2``。

如果使用repo命令获取补丁集是非常方便的，因为封装后的repo屏蔽掉了Gerrit的一些实

现细节，例如补丁集在Git库中的存在位置。如前所述，补丁集实际保存在refs/changes命名空间下。使用:command:`git ls-remote`命令，从Gerrit维护的代码库中可以看到补丁集对应的引用名称。

```
$ git ls-remote \
  ssh://review.source.android.com:29418/platform/sdk \
  refs/changes/93/16993*
5fb1e79b01166f5192f11c5f509cf51f06ab023d      refs/changes/93/16993/1
d342ef5b41f07c0202bc26e2bfff745b7c86d5a7      refs/changes/93/16993/2
```

接下来就来介绍一下Gerrit服务器的部署和使用方法。

## 架设Gerrit的服务器

### 下载war包

Gerrit是由Java开发的，封装为一个war包: :file:`gerrit.war`，安装非常简洁。如果需要从源码编译出war包，可以参照文档: <http://gerrit.googlecode.com/svn/documentation/2.1.5/dev-readme.html>。不过最简单的就是从Google Code上直接下载编译好的war包。

从下面的地址下载Gerrit的war包: <http://code.google.com/p/gerrit/downloads/list>。在下载页面会有一个文件名类似:file:`Gerrit-x.x.x.war`的war包，这个文件就是Gerrit的全部。示例中使用的是2.1.5.1版本，把下载的:file:`Gerrit-2.1.5.1.war`包重命名为Gerrit.war。下面的介绍就是基于这个版本。

### 数据库选择

Gerrit需要数据库来维护账户信息、跟踪评审任务等。目前支持的数据库类型有PostgreSQL、MySQL以及嵌入式的H2数据库。

选择使用默认的H2内置数据库是最简单的，因为这样无须任何设置。如果想使用更为熟悉的PostgreSQL或者MySQL，则预先建立数据库。

对于PostgreSQL，在数据库中创建一个用户gerrit，并创建一个数据库reviewdb。

```
createuser -A -D -P -E gerrit
createdb -E UTF-8 -O gerrit reviewdb
```

对于MySQL，在数据库中创建一个用户gerrit并为其设置口令（不要真如下面的将口令置为secret），并创建一个数据库reviewdb。

```
$ mysql -u root -p

mysql> CREATE USER 'gerrit'@'localhost' IDENTIFIED BY 'secret';
mysql> CREATE DATABASE reviewdb;
mysql> ALTER DATABASE reviewdb charset=latin1;
mysql> GRANT ALL ON reviewdb.* TO 'gerrit'@'localhost';
mysql> FLUSH PRIVILEGES;
```

### 以一个专用用户帐号执行安装

在系统中创建一个专用的用户帐号如: gerrit。以该用户身份执行安装，将Gerrit的配置文件、内置数据库、war包等都自动安装在该用户主目录下的特定目录中。

```
$ sudo adduser gerrit
$ sudo su gerrit
```

```
$ cd ~gerrit  
$ java -jar gerrit.war init -d review_site
```

在安装过程中会提问一系列问题。

- 创建相关目录。

缺省Gerrit在安装用户主目录下创建目录:`file:`review_site``并把相关文件安装在这个目录之下。Git版本库的根路径缺省位于此目录之下的:`file:`git``目录中。

```
*** Gerrit Code Review 2.1.5.1  
***  
  
Create '/home/gerrit/review_site' [Y/n]?  
  
*** Git Repositories  
***  
  
Location of Git repositories [git]:
```

- 选择数据库类型。

选择H2数据库是简单的选择，无须额外的配置。

```
*** SQL Database  
***  
  
Database server type [H2/?]:
```

- 设置Gerrit Web界面认证的类型。

缺省为openid，即使用任何支持OpenID的认证源（如Google、Yahoo）进行身份认证。此模式支持用户自建帐号，当用户通过OpenID认证源的认证后，Gerrit会自动从认证源获取相关属性如用户全名和邮件地址等信息创建帐号。Android项目的Gerrit服务器即采用此认证模式。

如果有可用的LDAP服务器，那么ldap或者ldap\_bind也是非常好的认证方式，可以直接使用LDAP中的已有帐号进行认证，不过此认证方式下Gerrit的自建帐号功能关闭。此安装示例选择的就是LDAP认证方式。

http认证也是可选的认证方式，此认证方式需要配置Apache的反向代理并在Apache中配置Web站点的口令认证，通过口令认证后Gerrit在创建帐号的过程中会询问用户的邮件地址并发送确认邮件。

```
*** User Authentication  
***  
  
Authentication method [OPENID/?]: ?  
Supported options are:  
  openid  
  http  
  http_ldap  
  ldap  
  ldap_bind  
  development_become_any_account  
Authentication method [OPENID/?]: ldap  
LDAP server [ldap://localhost]:  
LDAP username :  
Account BaseDN : dc=foo,dc=bar  
Group BaseDN [dc=foo,dc=bar]:
```

- 发送邮件设置。

缺省使用本机的SMTP发送邮件。

```
*** Email Delivery
***

SMTP server hostname      [localhost]:
SMTP server port          [(default)]:
SMTP encryption            [NONE/?]:
SMTP username               :
```

- Java相关设置。

使用OpenJava和Sun Java均可。Gerrit的war包要复制到:`file:`review_site/bin``目录中。

```
*** Container Process
***

Run as                  [gerrit]:
Java runtime             [/usr/lib/jvm/java-6-sun-1.6.0.21/jre]:
Copy gerrit.war to /home/gerrit/review_site/bin/gerrit.war [Y/n]?
Copying gerrit.war to /home/gerrit/review_site/bin/gerrit.war
```

- SSH服务相关设置。

Gerrit的基于SSH协议的Git服务非常重要，缺省的端口为29418。换做其他端口也无妨，因为repo可以自动探测到该端口。

```
*** SSH Daemon
***

Listen on address        [*]:
Listen on port            [29418]:

Gerrit Code Review is not shipped with Bouncy Castle Crypto v144
If available, Gerrit can take advantage of features
in the library, but will also function without it.
Download and install it now [Y/n]?
Downloading http://www.bouncycastle.org/download/bcprov-jdk16-144.jar ...
Checksum bcprov-jdk16-144.jar OK
Generating SSH host key ... rsa... dsa... done
```

- HTTP服务相关设置。

缺省启用内置的HTTP服务器，端口为8080，如果该端口被占用（如Tomcat），则需要更换为其他端口，否则服务启动失败。如下例就换做了8888端口。

```
*** HTTP Daemon
***

Behind reverse proxy      [y/N]? y
Proxy uses SSL (https://)  [y/N]? y
Subdirectory on proxy server  [/]: /gerrit
Listen on address          [*]:
Listen on port              [8081]:
Canonical URL               [https://localhost/gerrit]:
```

Initialized /home/gerrit/review\_site

### 启动Gerrit服务

Gerrit服务正确安装后，运行Gerrit启动脚本启动Gerrit服务。

```
$ /home/gerrit/review_site/bin/gerrit.sh start  
Starting Gerrit Code Review: OK
```

服务正确启动之后，会看到Gerrit服务打开两个端口，这两个端口是在Gerrit安装时指定的。您的输出和下面的示例可能略有不同。

```
$ sudo netstat -ltnp | grep -i gerrit  
tcp        0      0 0.0.0.0:8081          0.0.0.0:*          LISTEN  
tcp        0      0 0.0.0.0:29418         0.0.0.0:*          LISTEN
```

### 设置Gerrit服务开机自动启动

Gerrit服务的启动脚本支持start、stop、restart参数，可以作为init脚本开机自动执行。

```
$ sudo ln -snf \  
    /home/gerrit/review_site/bin/gerrit.sh \  
    /etc/init.d/gerrit.sh  
$ sudo ln -snf ../init.d/gerrit.sh /etc/rc2.d/S90gerrit  
$ sudo ln -snf ../init.d/gerrit.sh /etc/rc3.d/S90gerrit
```

服务自动启动脚本:`file:/etc/init.d/gerrit.sh`需要通过`file:/etc/default/gerritcodereview`提供一些缺省配置。以下面内容创建该文件。

```
GERRIT_SITE=/home/gerrit/review_site  
NO_START=0
```

### Gerrit认证方式的选择

如果是开放服务的Gerrit服务，使用OpenId认证是最好的方法，就像谷歌Android项目的代码审核服务器配置的那样。任何人只要在具有OpenId provider的网站上（如Google、Yahoo等）具有帐号，就可以直接通过OpenId注册，Gerrit会在用户登录OpenId provider网站成功后，自动获取（经过用户的确认）用户在OpenId provider站点上的部分注册信息（如用户全名或者邮件地址）在Gerrit上自动为用户创建帐号。

如果架设有LDAP服务器，并且用户帐号都在LDAP中进行管理，那么采用LDAP认证也是非常好的方法。登录时提供的用户名和口令通过LDAP服务器验证之后，Gerrit会自动从LDAP服务器中获取相应的字段属性，为用户创建帐号。创建的帐号的用户全名和邮件地址因为来自于LDAP，因此不能在Gerrit更改，但是用户可以注册新的邮件地址。我在配置LDAP认证时遇到了一个问题就是创建帐号的用户全名是空白，这是因为在LDAP相关的字段没有填写的原因。如果LDAP服务器使用的是OpenLDAP，Gerrit会从displayName字段获取用户全名，如果使用Active Directory则用givenName和sn字段的值拼接形成用户全名。

Gerrit还支持使用HTTP认证，这种认证方式需要架设Apache反向代理，在Apache中配置HTTP认证。当用户访问Gerrit网站首先需要通过Apache配置的HTTP Basic Auth认证，当Gerrit发现用户已经登录后，会要求用户确认邮件地址。当用户邮件地址确认后，再填写其他必须的字段完成帐号注册。HTTP认证方式的缺点除了在口令文件管理上需要管理员手工维护比较麻烦之外，还有一个缺点就是用户一旦登录成功后，想退出登录或者更换其他用户帐号登录变得非常麻烦，除非关闭浏览器。关于切换用户有一个小窍门：例如Gerrit登录URL为<https://server/gerrit/login/>，则用浏览器访

问<https://nobody:wrongpass@server/gerrit/login/>，即用错误的用户名和口令覆盖掉浏览器缓存的认证用户名和口令，这样就可以重新认证了。

在后面的Gerrit演示和介绍中，为了设置帐号的方便，使用了HTTP认证，因此下面再介绍一下HTTP认证的配置方法。

#### 配置Apache代理访问Gerrit

缺省Gerrit的Web服务端口为8080或者8081，通过Apache的反向代理就可以使用标准的80 (http) 或者443 (https) 来访问Gerrit的Web界面。

```
ProxyRequests Off
ProxyVia Off
ProxyPreserveHost On

<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>

ProxyPass /gerrit/ http://127.0.0.1:8081/gerrit/
```

如果要配置Gerrit的http认证，则还需要在上面的配置中插入Http Basic认证的设置。

```
<Location /gerrit/login/>
AuthType Basic
AuthName "Gerrit Code Review"
Require valid-user
AuthUserFile /home/gerrit/review_site/etc/gerrit.passwd
</Location>
```

在上面的配置中，指定了口令文件的位置：`:file:`/home/gerrit/review_site/etc/gerrit.passwd``。可以用`:command:`htpasswd``命令维护该口令文件。

```
$ touch /home/gerrit/review_site/etc/gerrit.passwd

$ htpasswd -m /home/gerrit/review_site/etc/gerrit.passwd jiangxin
New password:
Re-type new password:
Adding password for user jiangxin
```

至此为止，Gerrit服务安装完成。在正式使用Gerrit之前，先来研究一下Gerrit的配置文件，以免安装过程中遗漏或错误的设置影响使用。

## Gerrit的配置文件

Gerrit的配置文件保存在部署目录下的`:file:`etc/gerrit.conf``文件中。如果对安装时的配置不满意，可以手工修改配置文件，重启Gerrit服务即可。

全部采用缺省配置时的配置文件：

```
[gerrit]
basePath = git
canonicalWebUrl = http://localhost:8080/
[database]
type = H2
database = db/ReviewDB
[auth]
```

```
type = OPENID
[sendemail]
    smtpServer = localhost
[container]
    user = gerrit
    javaHome = /usr/lib/jvm/java-6-openjdk/jre
[sshd]
    listenAddress = *:29418
[httpd]
    listenUrl = http://*:8080/
[cache]
    directory = cache
```

如果采用LDAP认证，下面的配置文件片断配置了一个支持匿名绑定的LDAP服务器配置。

```
[auth]
type = LDAP
[ldap]
server = ldap://localhost
accountBase = dc=foo,dc=bar
groupBase = dc=foo,dc=bar
```

如果采用MySQL而非缺省的H2数据库，下面的配置文件显示了相关配置。

```
[database]
type = MYSQL
hostname = localhost
database = reviewdb
username = gerrit
```

LDAP绑定或者数据库连接的用户口令保存在`file:etc/secure.config`文件中。

```
[database]
password = secret
```

下面的配置将Web服务架设在Apache反向代理的后面。

```
[httpd]
listenUrl = proxy-https://*:8081/gerrit
```

## Gerrit的数据库访问

之所以要对数据库访问多说几句，是因为一些对Gerrit的设置往往在Web界面无法配置，需要直接修改数据库，而大部分用户在安装Gerrit时都会选用内置的H2数据库，如何操作H2数据库可能大部分用户并不了解。

实际上无论选择何种数据库，Gerrit都提供了两种数据库操作的命令行接口。第一种方法是在服务器端调用gerrit.war包中的命令入口，另外一种方法是远程SSH调用接口。

对于第一种方法，需要在服务器端执行，而且如果使用的是H2内置数据库还需要先将Gerrit服务停止。先以安装用户身份进入Gerrit部署目录下，在执行命令调用gerrit.war包，如下：

```
$ java -jar bin/gerrit.war gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))
```

```
Type '\h' for help. Type '\r' to clear the buffer.  
gerrit>
```

当出现“gerrit>”提示符时，就可以输入SQL语句操作数据库了。

第一种方式需要登录到服务器上，而且操作H2数据库时还要预先停止服务，显然很不方便。但是这种方法也有存在的必要，就是不需要认证，尤其是在管理员帐号尚未建立之前就可以查看和更改数据库。

当在Gerrit上注册了第一个帐号，即拥有了管理员帐号，正确为该帐号配置公钥之后，就可以访问Gerrit提供的SSH登录服务。Gerrit的SSH协议提供第二个访问数据库的接口。下面的命令就是用管理员公钥登录Gerrit的SSH服务器，操作数据库。虽然演示用的是本机地址（localhost），但是操作远程服务器也是可以的，只要拥有管理员授权。

```
$ ssh -p 29418 localhost gerrit gsql  
Welcome to Gerrit Code Review 2.1.5.1  
(H2 1.2.134 (2010-04-23))  
  
Type '\h' for help. Type '\r' to clear the buffer.  
gerrit>
```

即连接Gerrit的SSH服务，运行命令：`command: `gerrit gsql``。当连接上数据库管理接口后，便出现“gerrit>”提示符，在该提示符下可以输入SQL命令。下面的示例中使用的数据库后端为H2内置数据库。

可以输入`show tables`命令显示数据库列表。

```
gerrit> show tables;  
TABLE_NAME | TABLE_SCHEMA  
-----+-----  
ACCOUNTS | PUBLIC  
ACCOUNT AGREEMENTS | PUBLIC  
ACCOUNT DIFF PREFERENCES | PUBLIC  
ACCOUNT EXTERNAL IDS | PUBLIC  
ACCOUNT GROUPS | PUBLIC  
ACCOUNT GROUP AGREEMENTS | PUBLIC  
ACCOUNT GROUP MEMBERS | PUBLIC  
ACCOUNT GROUP MEMBERS AUDIT | PUBLIC  
ACCOUNT GROUP NAMES | PUBLIC  
ACCOUNT PATCH REVIEWS | PUBLIC  
ACCOUNT PROJECT WATCHES | PUBLIC  
ACCOUNT SSH KEYS | PUBLIC  
APPROVAL CATEGORIES | PUBLIC  
APPROVAL CATEGORY VALUES | PUBLIC  
CHANGES | PUBLIC  
CHANGE MESSAGES | PUBLIC  
CONTRIBUTOR AGREEMENTS | PUBLIC  
PATCH COMMENTS | PUBLIC  
PATCH SETS | PUBLIC  
PATCH SET ANCESTORS | PUBLIC  
PATCH SET APPROVALS | PUBLIC  
PROJECTS | PUBLIC  
REF RIGHTS | PUBLIC  
SCHEMA VERSION | PUBLIC  
STARRED CHANGES | PUBLIC  
SYSTEM CONFIG | PUBLIC  
TRACKING IDS | PUBLIC  
(27 rows; 65 ms)
```

输入show columns命令显示数据库的表结构。

```
gerrit> show columns from system_config;
FIELD          | TYPE      | NULL | KEY | DEFAULT
-----+-----+-----+-----+-----+
REGISTER_EMAIL_PRIVATE_KEY | VARCHAR(36) | NO   |   | ''
SITE_PATH       | VARCHAR(255) | YES  |   | NULL
ADMIN_GROUP_ID  | INTEGER(10)  | NO   |   | 0
ANONYMOUS_GROUP_ID | INTEGER(10) | NO   |   | 0
REGISTERED_GROUP_ID | INTEGER(10) | NO   |   | 0
WILD_PROJECT_NAME | VARCHAR(255) | NO   |   | ''
BATCH_USERS_GROUP_ID | INTEGER(10) | NO   |   | 0
SINGLETON       | VARCHAR(1)   | NO   | PRI | ''
```

(8 rows; 52 ms)

关于H2数据库更多的SQL语法，参考：<http://www.h2database.com/html/grammar.html>。

下面开始介绍Gerrit的使用。

## 立即注册为Gerrit管理员

第一个Gerrit账户自动成为权限最高的管理员，因此Gerrit安装完毕后的第一件事情就是立即注册或者登录，以便初始化管理员帐号。下面的示例是在本机（localhost）以HTTP认证方式架设的Gerrit审核服务器。当第一次访问的时候，会弹出非常眼熟的HTTP Basic Auth认证界面，如图32-4。



图32-4：Http Basic Auth 认证界面

输入正确的用户名和口令登录后，系统自动创建ID为1000000的帐号，该帐号是第一个注册的帐号，会自动被赋予管理员身份。因为使用的是HTTP认证，用户的邮件地址等个人信息尚未确定，因此登录后首先进入到个人信息设置界面。如图32-5。



图32-5：Gerrit第一次登录后的个人信息设置界面

在图32-5中可以看到在菜单中有“Admin”菜单项，说明当前登录的用户被赋予了管理员权限。在图32-5的联系方式确认对话框中有一个注册新邮件地址的按钮，点击该按钮弹出邮件地址录入对话框，如图32-6。



图32-6：输入个人的邮件地址

必须输入一个有效的邮件地址以便能够收到确认邮件。这个邮件地址非常重要，因为Git代码提交时在提交说明中出现的邮件地址需要和这个地址一致。当填写了邮件地址后，会收到一封确认邮件，点击邮件中的确认链接会重新进入到Gerrit帐号设置界面，如图32-7。



图32-7：邮件地址确认后进入Gerrit界面

在Full Name字段输入用户名，点击保存更改后，右上角显示的“Anonymous Coward”就会显示为登录用户的姓名和邮件地址。

接下来需要做的最重要的一件事就是配置公钥（如图32-8）。通过该公钥，注册用户可以通过SSH协议向Gerrit的Git服务器提交，如果具有管理员权限还能够远程管理Gerrit服务器。



图32-8：Gerrit的SSH公钥设置界面

在文本框中粘贴公钥。关于如何生成和管理公钥，参见第29章“使用SSH协议”相关内容。

点击“Add”按钮，完成公钥的添加。添加的公钥就会显示在列表中（如图32-9）。一个用户可以添加多个公钥。



图32-9：用户的公钥列表

点击左侧的“Groups”（用户组）菜单项，可以看到当前用户所属的分组，如图32-10。



图32-10：Gerrit用户所属的用户组

第一个注册的用户同时属于三个用户组，一个是管理员用户组（Administrators），另外两个分别是Anonymous Users（任何用户）和Registered Users（注册用户）。

## 管理员访问SSH的管理接口

当在Gerrit个人配置界面中设置了公钥之后，就可以连接Gerrit的SSH服务器执行命令，示例使用的是本机（localhost），其实远程IP地址一样可以。只是对于远程主机需要确认端口不要被防火墙拦截，Gerrit的SSH服务器使用特殊的端口，缺省是29418。

任何用户都可以通过SSH连接执行`:command:`gerrit ls-projects``命令查看项目列表。下面的命令没有输出，是因为项目尚未建立。

```
$ ssh -p 29418 localhost gerrit ls-projects
```

可以执行`:command:`scp``命令从Gerrit的SSH服务器中拷贝文件。

```
$ scp -P 29418 -p -r localhost:/ gerrit-files  
$ find gerrit-files -type f  
gerrit-files/bin/gerrit-cherry-pick  
gerrit-files/hooks/commit-msg
```

可以看出Gerrit服务器提供了两个文件可以通过`:command:`scp``下载，其中`:file:`commit-msg``脚本文件应该放在用户本地Git库的钩子目录中以便在生成的提交中包含唯一的Change-Id。在之前的Gerrit原理中介绍过。

除了普通用户可以执行的命令外，管理员还可以通过SSH连接执行Gerrit相关的管理命令。例如之前介绍的管理数据库：

```
$ ssh -p 29418 localhost gerrit gsql  
Welcome to Gerrit Code Review 2.1.5.1  
(H2 1.2.134 (2010-04-23))  
  
Type '\h' for help. Type '\r' to clear the buffer.  
  
gerrit>
```

此外管理员还可以通过SSH连接执行帐号创建、项目创建等管理操作，可以执行下面的命令查看帮助信息。

```
$ ssh -p 29418 localhost gerrit --help  
gerrit COMMAND [ARG ...] [--] [--help (-h)]  
  
-- : end of options  
--help (-h) : display this help text
```

```
Available commands of gerrit are:

approve
create-account
create-group
create-project
flush-caches
gsql
ls-projects
query
receive-pack
replicate
review
set-project-parent
show-caches
show-connections
show-queue
stream-events

See 'gerrit COMMAND --help' for more information.
```

更多的帮助信息，还可以参考Gerrit版本库中的帮助文件：[:file:`Documentation/cmd-index.html`](#)。

## 创建新项目

一个Gerrit项目对应于一个同名的Git库，同时拥有一套可定制的评审流程。创建一个新的Gerrit项目就会在对应的版本库根目录下创建Git库。管理员可以使用命令行创建新项目。

```
$ ssh -p 29418 localhost gerrit create-project --name new/project
```

当执行：[command:`gerrit ls-projects`](#) 命令，可以看到新项目创建已经成功创建。

```
$ ssh -p 29418 localhost gerrit ls-projects
new/project
```

在Gerrit的Web管理界面，也可以看到新项目已经建立，如图32-11。

图32-11：Gerrit中项目列表

在项目列表中可以看到除了新建的new/project项目之外还有一个名为“— All Projects —”的项目，其实它并非一个真实存在的项目，只是为了项目授权管理的方便。即在“— All Projects —” 中建立的项目授权能够被其他项目共享。

在服务器端也可以看到Gerrit部署中版本库根目录下已经有同名的Git版本库被创建。

```
$ ls -d /home/gerrit/review_site/git/new/project.git
/home/gerrit/review_site/git/new/project.git
```

这个新的版本库刚刚初始化，尚未包括任何数据。是否可以通过：[command:`git push`](#) 向该版本库推送一些初始数据呢？下面用Gerrit的SSH协议克隆该版本库，并尝试向其推送数据。

```
$ git clone ssh://localhost:29418/new/project.git myproject
Cloning into myproject...
warning: You appear to have cloned an empty repository.
```

```
$ cd myproject/  
  
$ echo hello > readme.txt  
  
$ git add readme.txt  
  
$ git commit -m "initialized."  
[master (root-commit) 15a549b] initialized.  
 1 files changed, 1 insertions(+), 0 deletions(-)  
 create mode 100644 readme.txt  
09:58:54 jiangxin@hp:~/tmp/myproject$ git push origin master  
Counting objects: 3, done.  
Writing objects: 100% (3/3), 222 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To ssh://localhost:29418/new/project.git  
! [remote rejected] master -> master (prohibited by Gerrit)  
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```

向Gerrit的Git版本库推送失败，远程Git服务器返回错误信息：“prohibited by Gerrit”。这是因为Gerrit缺省不允许直接向分支推送，而是需要向`refs/for/<branch-name>`的特殊引用进行推送以便将提交转换为评审任务。

但是如果希望将版本库的历史提交不经审核直接推送到Gerrit维护的Git版本库中可以么？是的，只要通过Gerrit的管理界面为该项目授权：允许某个用户组（如Administrators组）的用户可以向分支推送。（注意该授权在推送完毕后尽快撤销，以免被滥用）

Gerrit的界面对用户非常友好（如图32-12）。例如在添加授权的界面中，只要在用户组的输入框中输入前几个字母，就会弹出用户组列表供选择。



图32-12：添加授权的界面

添加授权完毕后，项目“new/project”的授权列表就会出现新增的为Administrators管理员添加的“+2: Create Branch”授权，如图32-13。



图32-13：添加授权后的授权列表

因为已经为管理员分配了直接向`refs/heads/*`引用推送的授权，这样就能够向Git版本库推送数据了。再执行一次推送任务，看看能否成功。

```
$ git push origin master  
Counting objects: 3, done.  
Writing objects: 100% (3/3), 222 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To ssh://localhost:29418/new/project.git  
! [remote rejected] master -> master (you are not committer jiangxin@osssxp.c  
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```

推送又失败了，但是服务器端返回的错误信息不同。上一次出错返回的是“prohibited by Gerrit”，而这一次返回的错误信息是“you are not committer”。

这是为什么呢？看看提交日志：

```
$ git log --pretty=full  
commit 15a549bac6bd03ad36e643984fed554406480b2c  
Author: Jiang Xin <jiangxin@osssxp.com>  
Commit: Jiang Xin <jiangxin@osssxp.com>  
  
initialized.
```

提交者（Commit）为“Jiang Xin <[jiangxin@osssxp.com](mailto:jiangxin@osssxp.com)>”，而Gerrit中注册的用户的邮件地址是“[jiangxin@moon.osssxp.com](mailto:jiangxin@moon.osssxp.com)”，两者之间的不一致，导致Gerrit再一次拒绝了提交。如果再到Gerrit看一下new/project的权限设置，会看到这样一条授权：

Category	Group Name	Reference Name	Permitted Range
Forge Identity	Registered Users	refs/*	+1: Forge Author Identity

这条授权的含义是提交中的Author字段不进行邮件地址是否注册的检查，但是要对Commit字段进行邮件地址检查。如果增加一个更高级别的“Forge Identity”授权，也可以忽略对Committer的邮件地址检查，但是尽量不要对授权进行非必须的改动，因为在提交的时候使用注册的邮件地址是一个非常好的实践。

下面就通过:command:`git config`命令修改提交时所用的邮件地址，和Gerrit注册时用的地址保持一致。然后用--amend参数重新执行提交以便让修改后的提交者邮件地址在提交中生效。

```
$ git config user.email jiangxin@moon.osssxp.com

$ git commit --amend -m initialized
[master 82c8fc3] initialized
Author: Jiang Xin <jiangxin@osssxp.com>
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 readme.txt

$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 233 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 * [new branch]      master -> master
```

看这次提交成功了！之所以成功，是因为提交者的邮件地址更改了。看看重新提交的日志，可以发现作者（Author）和提交者（Commit）的邮件地址的不同，Commit字段的邮件地址和注册时使用的邮件地址相同。

```
$ git log --pretty=full
commit 82c8fc3805d57cc0d17d58e1452e21428910fd2d
Author: Jiang Xin <jiangxin@osssxp.com>
Commit: Jiang Xin <jiangxin@moon.osssxp.com>

initialized
```

注意，版本库初始化完成之后，应尽快把为项目新增的“Push Branch”类型的授权删除，对新的提交强制使用Gerrit的评审流程。

## 从已有Git库创建项目

如果已经拥有很多版本库，希望从这些版本库创建Gerrit项目，如果像上面介绍的那样一个一个的创建项目，再执行:command:`git push`命令推送已经包含历史数据的版本库，将是十分麻烦的事情。那么有没有什么简单的办法呢？可以通过下面的步骤，实现多项目的快速创建。

首先将已有版本库创建到Gerrit的版本库根目录下。注意版本库名称将会成为项目名（除去.git后缀），而且创建（或克隆）的版本库应为裸版本库，即使用--bare参数创建。

例如在Gerrit的Git版本库根目录下创建名为hello.git的版本库。下面的示例中我偷了一下懒，直接从new/project克隆到hello.git。:)

```
$ git clone --mirror \
/home/gerrit/review_site/git/new/project.git \
/home/gerrit/review_site/git/hello.git
```

这时查看版本库列表，却看不到新建立的名为hello.git的Git库出现在项目列表中。

```
$ ssh -p 29418 localhost gerrit ls-projects
new/project
```

可以通过修改Gerrit数据库来注册新项目，即连接到Gerrit数据库，输入SQL插入语句。

```
$ ssh -p 29418 localhost gerrit gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))

Type '\h' for help. Type '\r' to clear the buffer.

gerrit> INSERT INTO projects
-> (use_contributor_agreements ,submit_type ,name)
-> VALUES
-> ('N' , 'M' , 'hello');
UPDATE 1; 1 ms
gerrit>
```

注意SQL语句中的项目名称是版本库名称除去.git后缀的部分。在数据库插入数据后，再来看项目列表就可以看到新注册的项目了。

```
$ ssh -p 29418 localhost gerrit ls-projects
hello
new/project
```

可以登录到Gerrit项目对新建立的项目进行相关设置。例如修改项目的说明，项目的提交策略，是否要求提交说明中必须包含“Signed-off-by”信息等，如图32-14。

图32-14：项目基本设置

这种通过修改数据库从已有版本库创建项目的方法适合大批量的项目创建。下面就对新建立的hello项目进行一次完整的Gerrit评审流程。

## 定义评审工作流

刚刚安装好的Gerrit的评审工作流并不完整，还不能正常的开展评审工作，需要对项目授权进行设置以定制适合的评审工作流。

缺省安装的Gerrit中只内置了四个用户组，如表32-1所示。

表32-1：Gerrit内置用户组

用户组	说明
Administrators	Gerrit 管理员
Anonymous Users	任何用户，登录或未登录

Non-Interactive Users	Gerrit 中执行批处理的用户
Registered Users	任何登录用户

未登录的用户只属于Anonymous Users，登录用户则同时拥有Anonymous Users和 Registered Users的权限。对于管理员则还拥有Administrators用户组权限。

查看全局（伪项目“-- All Projects --”）的初始权限设置。会看到如表32-2一样的授权表格。

表32-2: Gerrit授权表格

编号	类别	用户组名称	引用名称	权限范围
1	Code Review	Registered Users	refs/heads/*	-1: I would prefer that you didn't submit this
				+1: Looks good to me, but someone else must approve
2	Forge Identity	Registered Users	refs/*	+1: Forge Author Identity
3	Read Access	Administrators	refs/*	+1: Read access
4	Read Access	Anonymous Users	refs/*	+1: Read access
5	Read Access	Registered Users	refs/*	+2: Upload permission

对此表格中的授权解读如下：

- 对于匿名用户：根据第4条授权策略，匿名用户能够读取任意版本库。
- 对于注册用户：根据第5条授权策略，注册用户具有比第四条授权高一个等级的权限，即注册用户除了具有读取版本库权限外，还可以向版本库的refs/for/<branch-name>引用推送，产生评审任务的权限。

之所以这种可写的权限也放在“Read Access”类别中，是因为Git的写操作必须建立在拥有读权限之上，因此Gerrit将其与读取都放在“Read Access”归类之下，只不过更高一个级别。

- 对于注册用户：根据第2条授权策略，在向服务器推送提交的时候，忽略对提交中 Author 字段的邮件地址检查。这个在之前已经讨论过。
- 对于注册用户：根据第1条授权策略，注册用户具有代码审核的一般权限，即能够将评审任务设置为“+1”级别（看起来不错，但需要通过他人认可），或者将评审任务标记为“-1”，即评审任务没有通过不能提交。
- 对于管理员：根据第3条策略，管理员能够读取任意版本库。

上面的授权策略仅仅对评审流程进行了部分设置。如：提交能够进入评审流程，因为登录用户（注册用户）可以将提交以评审任务方式上传；注册用户可以将评审任务标记为“+1: 看起来不错，但需其他人认可”。但是没有人有权限可以将评审任务提交——合并到正式版本库中，即没人能够对评审任务做最终的确认及提交，因此评审流程是不完整的。

要想实现对评审最终确认的授权，有两种方法可以实现，一种是赋予特定用户Verified类别中的“+1: Verified”的授权，另外一个方法是赋予特定用户Code Review类别中更高级别的授权：“+2: Looks good to me, approved”。要想实现对经过确认的评审任务提交，还需要赋予特定用户Submit类别中的“+1: Submit”授权。

下面的示例中，创建两个新的用户组Reviewer和Verifier，并为其赋予相应的授权。

创建用户组，可以通过Web界面或者命令行。如果通过Web界面添加用户组，选择“Admin”菜单下的“Groups”子菜单，如图32-15。



图32-15: Gerrit用户组创建

输入用户组名称后，点击“Create Group”按钮。进入创建用户组后的设置页，如图32-16。



图32-16: Gerrit用户组设置页

注意到在用户设置页面中有一个Owners字段名称和用户组名称相同，实际上这是Gerrit关于用户组的一个特别的功能。一个用户组可以设置另外一个用户组为本用户组的Owners，属于Owners用户组的用户实际上相当于本用户组的管理者，可以添加用户、修改用户组名称等。不过一般最常用的设置是使用同名的用户组作为Owners。

在用户组设置页面的最下面，是用户组用户分配对话框，可以将用户分配到用户组中。注意Gerrit的用户组不能包含，即只能将用户分配到用户组中。

图32-17是添加了两个新用户组后的用户组列表：



图32-17: Gerrit用户组列表

接下来要为新的用户组授权，需要访问“Admin”菜单下的“Projects”子菜单，点击对应的项目进入权限编辑界面。为了简便起见，选择“— All Projects —”，对其授权的更改可以被所有其他的项目共享。图32-18是为Reviewer用户组建立授权过程的页面。



图32-18: 为Reviewer用户组建立授权

分别为两个新建立的用户组分配授权，如表32-3所示。编号从6开始，是因为这里补充的授权是建立在前面的缺省授权列表的基础上的。

表32-3：新用户组权限分配表

编号	类别	用户组名称	引用名称	权限范围
6	Code Review	Reviewer	refs/*	-2: Do not submit
				+2: Looks good to me, approved
7	Verified	Verifier	refs/*	-1: Fails
				+1: Verified
8	Submit	Verifier	refs/*	+1: Submit

这样，就为Gerrit所有的项目设定了可用的评审工作流。

## Gerrit评审工作流实战

分别再注册两个用户帐号dev1@moon.osssxp.com和dev2@moon.osssxp.com，两个用户分别属于Reviewer用户组和Verifier用户组。这样Gerrit部署中就拥有了三个用户帐号，用帐号jiangxin进行代码提交，用dev1帐号对任务进行代码审核，用dev2用户对审核任务进行最终的确认。

## 开发者在本地版本库中工作

Repo是Gerrit的最佳伴侣，凡是需要和Gerrit版本库交互的工作都封装在repo命令中。关于repo的用法在上一部分的repo多版本库协同的章节中已经详细介绍了。这里只介绍开发者如何只使用Git命令来和Gerrit服务器交互。这样也可以更深入的理解repo和gerrit整合的机制。

首先克隆Gerrit管理的版本库，使用Gerrit提供的运行于29418端口的SSH协议。

```
$ git clone ssh://localhost:29418/hello.git
Cloning into hello...
remote: Counting objects: 3, done
remote: Compressing objects: 100% (3/3)
Receiving objects: 100% (3/3), done.
```

然后拷贝Gerrit服务器提供的:`:file:`commit-msg``钩子脚本。

```
$ cd hello
$ scp -P 29418 -p localhost:/hooks/commit-msg .git/hooks/
```

别忘了修改Git配置中提交者的邮件地址，以便和Gerrit中注册的地址保持一致。不使用`--global`参数调用:`:command:`git config``可以只对本版本库的提交设定提交者邮件。

```
$ git config user.email jiangxin@moon.ossxp.com
```

然后修改:`:file:`readme.txt``文件，并提交。注意提交的时候使用了`-s`参数，目的是在提交说明中加入“Signed-off-by:”标记，这在Gerrit提交中可能是必须的。

```
$ echo "gerrit review test" >> readme.txt
$ git commit -a -s -m "readme.txt hacked."
[master c65ab49] readme.txt hacked.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

查看一下提交日志，会看到其中有特殊的标签。

```
$ git log --pretty=full -1
commit c65ab490f6d3dc36429b8f1363b6191357202f2e
Author: Jiang Xin <jiangxin@moon.ossxp.com>
Date:   Mon Nov 15 17:50:08 2010 +0800

readme.txt hacked.

Change-Id: Id7c9d88ebf5dac2d19a7e0896289de1ae6fb6a90
Signed-off-by: Jiang Xin <jiangxin@moon.ossxp.com>
```

提交说明中出现了“Change-Id:”标签，这个标签是由钩子脚本:`:file:`commit-msg``自动生成的。至于这个标签的含义，在前面Gerrit的实现原理中介绍过。

好了，准备把这个提交PUSH到服务器上吧。

## 开发者向审核服务器提交

由Gerrit控制的Git版本库不能直接提交，因为正确设置的Gerrit服务器，会拒绝用户直接向`refs/heads/*`推送。

```
$ git status
```

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)

$ git push
Counting objects: 5, done.
Writing objects: 100% (3/3), 332 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 ! [remote rejected] master -> master (prohibited by Gerrit)
error: failed to push some refs to 'ssh://localhost:29418/hello.git'
```

直接推送就会出现遇到“prohibited by Gerrit”的错误。

正确的做法是向特殊的引用推送，这样Gerrit会自动将新提交转换为评审任务。

```
$ git push origin HEAD:refs/for/master
Counting objects: 5, done.
Writing objects: 100% (3/3), 332 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 * [new branch]      HEAD -> refs/for/master
```

看到了么，向`refs/for/master`推送成功。

## 审核评审任务

以dev1用户登录Gerrit网站，点击“All”菜单下的“Open”标签，可以新提交到Gerrit状态为Open的评审任务，如图32-19。



图32-19：Gerrit评审任务列表

点击该评审任务，显示关于此评审任务的详细信息，如图32-20。



图32-20：Gerrit评审任务概述

从URL地址栏可以看到该评审任务的评审编号为1。目前该评审任务有一个补丁集（Patch Set 1），可以点击“Diff All Side-by-Side”查看变更集，以决定该提交是否应该被接受。作为测试，先让此次提交通过代码审核，于是以dev1用户身份点击“Review”按钮。

点击“Review”按钮后，弹出代码评审对话框，如图32-21。



图32-21：Gerrit任务评审对话框

选择“+2: Looks good to me, approved.”，点击按钮“Publish Comments”以通过评审。注意因为没有给dev1用户（Reviewer用户组）授予Submit权限，因此此时dev1还不能将此审核任务提交。

当dev1用户做出通过评审的决定后，代码提交者jiangxin会收到一封邮件，如图32-22。



图32-22：Gerrit通知邮件

## 评审任务没有通过测试

下面以dev2帐号登录Gerrit，查看处于打开状态的评审任务，如图32-23。会看到评审任务

1的代码评审已经通过，但是尚未进行测试检查（Verify）。于是dev2可以下载该补丁集，在本机进行测试。



图32-23: Gerrit评审任务显示

假设测试没有通过，dev2用户点击该评审任务的“Review”按钮，重置该任务的评审状态，如图32-24。



图32-24: Gerrit评审任务未通过

注意到图32-24中dev2用户的评审对话框有三个按钮，多出的“Publish and Submit”按钮是因为dev2拥有Submit授权。dev2用户在上面的对话框中，选择了“-1: Fails”，当点击“Publish Comments”按钮，该评审任务的评审记录被重置，同时提交者和其他评审参与者会收到通知邮件，如图32-25。



图32-25: Gerrit通知邮件：评审未通过

## 重新提交新的补丁集

提交者收到代码被打回的邮件，一定很难过。不过这恰恰说明了这个软件过程已经相当的完善，现在发现问题总比在集成测试时甚至被客户发现要好的多吧。

根据评审者和检验者的提示，开发者对代码进行重新修改。下面的bugfix过程仅仅是一个简单的示例，bugfix没有这么简单的，对么？;-)

```
$ echo "fixed" >> readme.txt
```

重新修改后，需要使用--amend参数进行提交，即使用前次提交的日志重新提交，这一点非常重要。因为这样就会对原提交说明中的“Change-Id:”标签予以原样保留，当再将新提交推送到服务器时，Gerrit不会为新提交生成新的评审任务编号而是会重用原有的任务编号，将新提交转化为老的评审任务的新的补丁集。

在执行:command:`git commit --amend`时，可以修改提交说明，但是注意不要删除Change-Id标签，更不能修改它。

```
$ git add -u
$ git commit --amend

readme.txt hacked with bugfix.

Change-Id: Id7c9d88ebf5dac2d19a7e0896289de1ae6fb6a90
Signed-off-by: Jiang Xin <jiangxin@moon.ossxp.com>

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
# modified:   readme.txt
#
```

提交成功后，执行:command:`git ls-remote`命令会看到Gerrit维护的Git库中只有一个评审任务（编号1），且该评审任务只有一个补丁集（Patch Set 1）。

```
$ git ls-remote origin
82c8fc3805d57cc0d17d58e1452e21428910fd2d      HEAD
c65ab490f6d3dc36429b8f1363b6191357202f2e      refs/changes/01/1/1
82c8fc3805d57cc0d17d58e1452e21428910fd2d      refs/heads/master
```

把修改后的提交推送到Gerrit管理下的Git版本库中。注意依旧推送到`refs/for/master`引用中。

```
$ git push origin HEAD:refs/for/master
Counting objects: 5, done.
Writing objects: 100% (3/3), 353 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 * [new branch]      HEAD -> refs/for/master
```

推送成功后，再执行：`git ls-remote` 命令，会看到唯一的评审任务（编号1）有了两个补丁集。

```
$ git ls-remote origin
82c8fc3805d57cc0d17d58e1452e21428910fd2d      HEAD
c65ab490f6d3dc36429b8f1363b6191357202f2e      refs/changes/01/1/1
1df9e8e05fcf97a46588488918a476abd1df8121      refs/changes/01/1/2
82c8fc3805d57cc0d17d58e1452e21428910fd2d      refs/heads/master
```

## 新修订集通过评审

当提交者重新针对评审任务进行提交时，原评审任务的审核者会收到通知邮件，提醒有新的补丁集等待评审，如图32-26。

图32-26：Gerrit通知邮件：新补丁集

登录Gerrit的Web界面，可以看到评审任务1有了新的补丁集，如图32-27。

图32-27：Gerrit新补丁集显示

再经过代码审核和测试，这次dev2用户决定让评审通过，点击了“Publish and Submit”按钮。Submit（提交）动作会将评审任务（`refs/changes/01/1/2`）合并到对应分支（`master`）。图32-28显示的是通过评审完成合并的评审任务1。

图32-28：Gerrit合并后的评审任务

## 从远程版本库更新

当dev1和dev2用户完成代码评审，提交者会收到多封通知邮件。这其中最让人激动的就是代码被接受并合并到开发主线（`master`）中（如图32-29），这是多么另开发者感到荣耀啊。

图32-29：Gerrit通知邮件：修订已合并

代码提交者执行：`git pull`，和Gerrit管理的版本库同步。

```
$ git ls-remote origin
1df9e8e05fcf97a46588488918a476abd1df8121      HEAD
```

```
c65ab490f6d3dc36429b8f1363b6191357202f2e      refs/changes/01/1/1
1df9e8e05fcf97a46588488918a476abd1df8121      refs/changes/01/1/2
1df9e8e05fcf97a46588488918a476abd1df8121      refs/heads/master

$ git pull
From ssh://localhost:29418/hello
 82c8fc3..1df9e8e  master    -> origin/master
 Already up-to-date.
```

## 更多Gerrit参考

Gerrit涉及到的内容非常庞杂，还有诸如和Gitweb、git-daemon整合，Gerrit界面定制等功能，恕不在此一一列举。可以直接参考Gerrit网站上的帮助。

参见：<http://gerrit.googlecode.com/svn/documentation/>。

来源：<https://github.com/gotgit/gotgit/blob/master/05-git-server/055-gerrit.rst>

# Git版本库托管

想不想在互联网上为自己的Git版本库建立一个克隆？这样再也不必为数据的安全担忧（异地备份），还可以和他人数据共享、协同工作？但是这样做会不会很贵呢？比如要购买域名、虚拟主机、搭建Git服务器什么的？

实际上这种服务（Git版本库托管服务）可以免费获得！GitHub、Gitorious、Bitbucket等都可以免费提供这些服务。

## Github

如果读者是按部就班式的阅读本书，那么可能早就注意到本书的很多示例版本库都是放在GitHub上的。GitHub提供了Git版本库托管服务，即包括收费商业支持，也提供免费的服务，很多开源项目把版本库直接放在了GitHub上，如：jQuery、curl、Ruby on Rails等。

注册一个GitHub帐号非常简单，访问GitHub网站：<https://github.com/>，点击菜单中的“Pricing and Signup”就可以看到GitHub的服务列表（如图33-1）。会看到其中有一个免费的服务：“Free for open source”，并且版本库托管的数量不受限制。当然免费的午餐是不管饱的，托管的空间只有300MB，而且不能创建私有版本库。

The screenshot shows the GitHub 'Plans & Pricing' page. At the top, it says 'Free for open source' with 'Unlimited public repositories and unlimited public collaborators'. A 'Create a free account' button is visible. Below are three paid plan options:

Micro	\$7/mo	Small	\$12/mo	Medium	\$22/mo
5 Private Repositories 1 Private Collaborator		10 Private Repositories 5 Private Collaborators		20 Private Repositories 10 Private Collaborators	
Unlimited public repositories		Unlimited public repositories		Unlimited public repositories	
Unlimited public collaborators		Unlimited public collaborators		Unlimited public collaborators	
<a href="#">Create an account</a>		<a href="#">Create an account</a>		<a href="#">Create an account</a>	

图33-1：GitHub服务价目表

点击按钮“Create a free account”，就可以创建一个免费的帐号。GitHub的用法和前面介绍的Gerrit Web界面的用法很类似，一旦帐号创建，应该马上为新建立的帐号设置公钥，以便能够用SSH协议读写自己帐号下创建的版本库，如图33-2。



图33-2：GitHub上配置公钥

创建仓库的操作非常简单，首先点击菜单上的“主页”（Dashboard），再点击右侧边栏上的“新建仓库”按钮就可以创建新的版本库了，如图33-3。

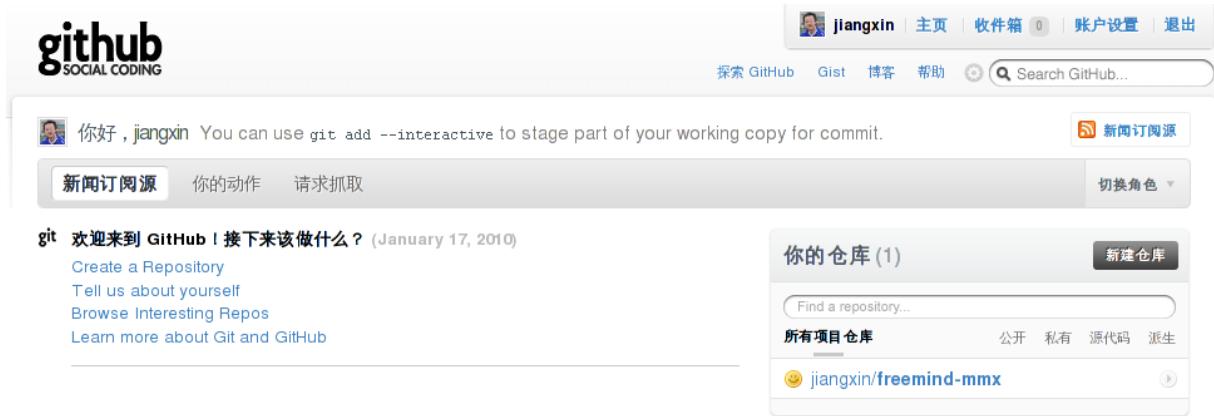


图33-3：GitHub页面上的新建版本库按钮

新建版本库会浪费本来就不多的托管空间，从GitHub上已有版本库派生（fork）一个克隆是一个好办法。首先通过GitHub搜索版本库名称，找到后点击“派生”按钮，就可以在自己的托管空间内建立相应版本库的克隆，如图33-4。



图33-4：自GitHub上的版本库派生

版本库建立后就可以用Git命令访问GitHub上托管的版本库了。GitHub提供三种协议可供访问，如图33-5。其中SSH协议和HTTP协议支持读写，Git-daemon提供只读访问。对于自己的版本库当然选择支持读写的服务方式了，其中SSH协议是首选。



图33-5：版本库访问URL

## Gitorious

Gitorious是另外一个Git版本库托管提供商，网址为`http://gitorious.org/`。最酷的是Gitorious本身的架站软件也是开源的，可以通过Gitorious上的Gitorious项目访问。如果读者熟悉Ruby on Rails，可以架设一个本地的Gitorious服务。

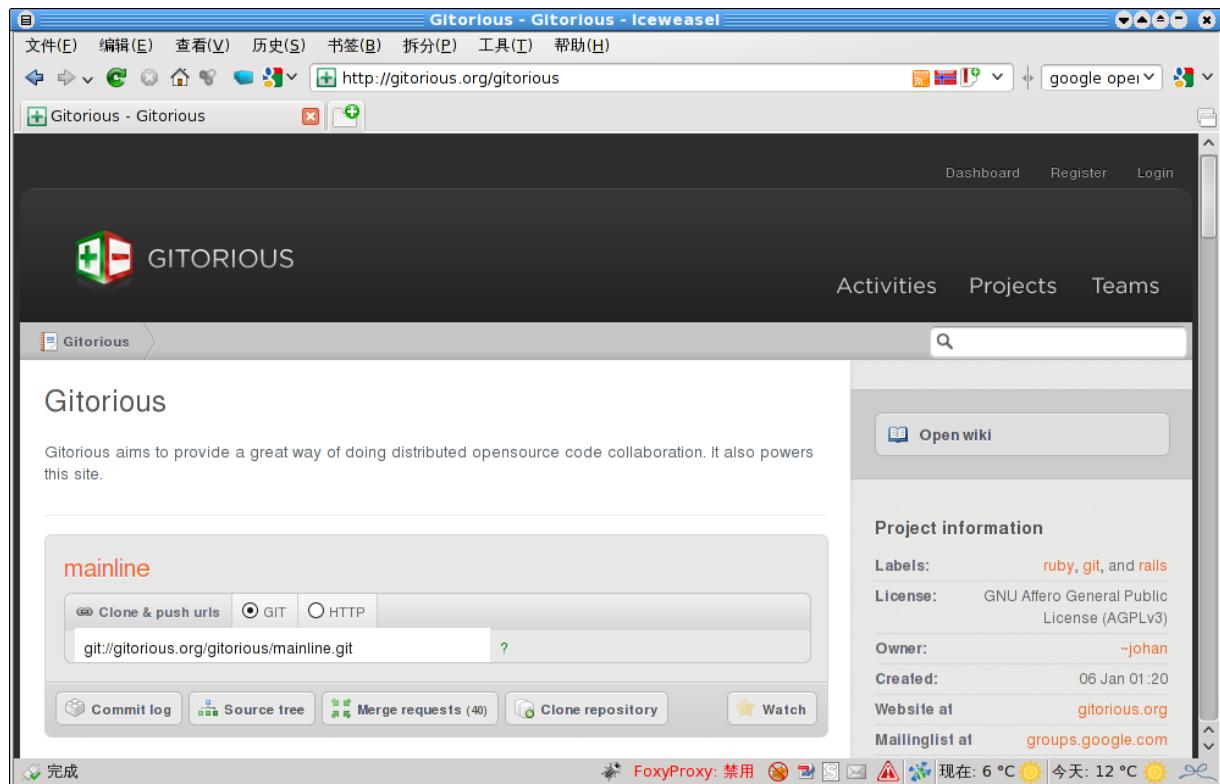


图33-6：Gitorious上的Gitorious项目

来源：<https://github.com/gotgit/gotgit/blob/master/05-git-server/060-github-like.rst>

## CVS版本库到Git的迁移

CVS是最早广泛使用的版本控制系统，因为其服务器端存储结构的简单直白，至今仍受到不少粉丝的钟爱。但是毕竟是几十年前的产物，因为设计上的原因导致缺乏现代版本控制系统的一些必须功能，如：没有原子提交，分支管理不便（慢），分支合并困难因为合并过程缺乏跟踪，不支持文件名/目录名的修改等等。很多CVS的用户都已经转换到Subversion这一更好的集中式版本控制系统了。如果还在使用CVS，那么可以考虑直接迁移到Git。

CVS到Git迁移可以使用cvs2svn软件包中的`cvs2git`命令。为什么该项目叫做cvs2svn而非cvs2git呢？这是因为该项目最早是为CVS版本库迁移到Subversion版本库服务的，只是最近才增加了CVS版本转换为Git版本库的功能。cvs2svn将CVS转换为Subversion版本库的过程一直以稳定著称，在cvs2svn 2.1版开始，增加了将CVS版本库转换为Git版本库的功能，无疑让这个工具更具生命力，也减少了之前CVS到Git库的转换环节。在推出cvs2git功能之前，通常的CVS到Git迁移路径是用cvs2svn将CVS版本库迁移到Subversion版本库，再用git-svn将Subversion版本库迁移到Git。

关于cvs2svn及cvs2git可以参考下面的链接：

- <http://cvs2svn.tigris.org/cvs2svn.html>
- <http://cvs2svn.tigris.org/cvs2git.html>

## 安装cvs2svn（含cvs2git）

### Linux下cvs2svn的安装

大部分Linux发行版都提供cvs2svn的发布包，可以直接用平台自带的cvs2svn软件包。cvs2svn在2.1版本之后开始引入了到Git库的转换，2.3.0版本有了独立的cvs2git转换脚本，cvs2git正在逐渐完善当中，因此尽量选择最新版本的cvs2svn。

例如在Debian或Ubuntu下，可以通过下面命令查看源里面的cvs2svn版本。

```
$ aptitude versions cvs2svn
p 2.1.1-1          stable           990
pi 2.3.0-2         testing,unstable 1001
```

可以看出Debian的Testing和Sid的仓库中才有2.3.0版本的cvs2svn。于是执行下面的命令安装在Testing版本才有的2.3.0-2版本的cvs2svn：

```
$ sudo aptitude cvs2svn/testing
```

如果对应的Linux发行版没有对应的版本也可以从源码开始安装。cvs2svn的官方版本库在<http://cvs2svn.tigris.org/svn/cvs2svn/trunk>，已经有人将cvs2svn项目转换为Git库。可以从Git库下载源码，安装cvs2svn。

- 下载cvs2svn源代码

```
$ git clone git://repo.or.cz/cvs2svn.git
```

- 进入cvs2svn源码目录，安装cvs2svn。

```
$ cd cvs2svn
$ sudo make install
```

- 安装用户手册。

```
$ sudo make man
```

cvs2svn对其他软件包的依赖:

- Python 2.4或以上版本 (Python 3.x暂不支持)。
- RCS: 如果在转换中使用了--use-rcs, 就需要安装RCS软件包。参见: <http://www.cs.purdue.edu/homes/trinkle/RCS/>。
- CVS: 如果在转换中使用了--use-cvs, 就需要安装CVS软件包。参见: <http://ccvs.cvshome.org/>。
- Git: 1.5.4.4或以上的版本。之前版本的Git的git fast-import 命令有Bug, 加载cvs2git导出文件有问题。

#### Mac OS X下cvs2svn的安装

Mac OS X下可以使用Homebrew安装cvs2svn。

- Mac OS X缺省安装的Python缺少cvs2svn依赖的gdbm模组, 先用Homebrew来重新安装python。

```
$ brew install python
```

- 安装cvs2svn

```
$ export PATH=/usr/local/bin:$PATH  
$ brew install cvs2svn
```

## 版本库转换（命令行参数模式）

转换CVS版本库的注意事项:

- 使用cvs2git对CVS版本库转换, 必须在CVS的服务器端执行, 即cvs2git必须能够通过文件系统直接访问CVS版本库中的:file:`, v`文件。
- 在转换前, 确保所有人的修改都已经提交到CVS版本库中。
- 在转换前, 停止CVS版本库的访问, 以免在转换过程中有新提交写入。
- 在转换前, 对原始版本库进行备份, 以免误操作对版本库造成永久的破坏。
- 在转换完成后, 永久停止CVS版本库的写入服务, 可以仅开放只读服务。

这是由于cvs2git是一次性操作, 不能对CVS后续提交执行增量式的到Git库转换, 因此当CVS版本库转换完毕后, 须停止CVS服务。

- 先做小规模的试验性转换。

转换CVS版本库切忌一上来就对整个版本库进行转换, 等到发现日志乱码、文件名乱码、提交者ID不完全后重新转换会浪费大量时间。

应该先选择CVS版本库中的部分文件和目录作为样本, 进行小规模的转换测试。

- 不要对包含:file:`CVSROOT`目录的版本库的根进行操作, 可以先对服务器目录布局进行调整。

如果转换直接针对包含:file:`CVSROOT`目录的版本库根目录进行操作, 会导致:file:`CVSROOT`目录下的文件及更改历史也被纳入到Git版本库, 这是不需要的。

#### 检查CVS版本库中的文件名乱码

CVS中保存的数据在服务器端直接和同名文件（文件多了一个“,v”后缀）相对应，当转换的CVS版本库是从其他平台（如Windows）拷贝过来的，就可能因为平台本身字符集不一致导致中文文件名包含乱码，在CVS版本库转换过程造成乱码。可以先对有问题的目录名和文件名进行重命名，转换为当前平台正确的编码。

### 小规模的转换试验

前面提到过，最好先进行小规模的转换试验，然后再对整个版本库进行转换。例如版本库是如下方式部署：`:file:`CVSROOT``为`/cvshome/user`，需要将之下

的`:file:`jiangxin/homepage/worldhello``转换为一个Git版本库。先检查一下版本库中的数据，找出典型的目录用于转换。

典型的数据是这样的：包含中文文件名，并且日志中包含中文。例如在版本库中，执行CVS查看日志命令，看到类似下面的输出。

```
RCS file: /cvshome/user/jiangxin/homepage/worldhello/archive/2003/.mhonarc.db
Working file: archive/2003/.mhonarc.db
head: 1.16
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 16;    selected revisions: 16
description:
-----
revision 1.16
date: 2004-09-21 15:56:30 +0800;  author: jiangxin;  state: Exp;  lines: +3 -
<D0><U+07B8><C4><D3>'<FE><B5><D8><A3><BB>
<D0><U+07B8><C4><CB><D1><CB><F7><D2><FD><C7>禁
-----
```

日志乱码是因为CVS并没有对日志的字符转换为统一的UTF-8字符集。此版本库之前用CVSNT维护，缺省字符集为GBK。那么就先对有乱码的这一个目录进行一下试验性的转换。

- 调用cvs2git执行转换，产生两个导出文件。这两个导出文件将作为Git版本库创建时的导入文件。

命令行用了两个`--encoding`参数设置编码，会依次进行尝试将日志中的非Ascii字符转换为UTF-8。

```
$ cvs2git --blobfile git-blob.dat --dumpfile git-dump.dat \
--encoding utf8 --encoding gbk --username cvs2git \
/cvshome/user/jiangxin/homepage/worldhello/archive/2003/
```

- 成功导出后，产生两个导出文件，一个保存各个文件的各个不同版本的数据内容，即在命令行指定的输出文件`:file:`git-blob.dat``。另外一个文件是上面命令行指定的`:file:`git-dump.dat``用于保存各个提交相关信息（提交者、提交时间、提交日志等）。

```
$ du -sh git*dat
9.8M    git-blob.dat
24K     git-dump.dat
```

可以看出保存文件内容的导出文件`(:file:`git-blob.dat`)`相对更大一些。

- 创建空的Git库，使用Git通用的数据迁移命令`:command:`git fast-import``将cvs2git的导出文件导入版本库中。

```
$ mkdir test
$ cd test
$ git init
$ cat ../git-blob.dat ../git-dump.dat | git fast-import
```

- 检查导出结果。

```
$ git reset HEAD
$ git checkout .
$ git log -1
commit 8334587cb241076bcd2e710b321e8e16b5e46bba
Author: jiangxin <>
Date:   Tue Sep 21 07:56:31 2004 +0000
```

修改邮件地址;  
修改搜索引擎;

很好，导出的Git库的日志，中文乱码问题已经解决。但是会发现提交日志中的作者（Author）字段信息不完整：缺乏邮件地址。这是因为CVS的提交者仅为用户登录ID，而Git的提交者信息还要包含邮件地址。cvs2git提供参数实现两种提交者ID的转换，不过需要通过配置文件予以指定，这就需要采用下面介绍的转换方法。

## 版本库转换（配置文件模式）

使用命令行参数调用cvs2git麻烦、可重用性差，而且可配置项有限。采用cvs2git配置文件模式运行不但能够简化cvs2git的命令行参数，而且能够提供更多的命令行无法提供的配置项，可以更精确的对CVS到Git版本库转换进行定制。

cvs2svn软件包提供了一个cvs2git的配置示例文件，见源码中的cvs2git-example.options[1]。将该示例文件在本地复制一份，对其进行更改。该文件是Python代码格式，以“#”（井号）开始的行是注释，文件缩进不要随意更改，因为缩进也是Python语法的一部分。可以考虑针对下列选项进行定制。

- 设置CVS版本库位置。

使用配置文件方式运行cvs2git，只能在配置文件中设置要转换的CVS版本库位置，而不能在命令行进行设置。具体说是在配置文件的最后面run\_options的set\_project方法中指定。

```
run_options.set_project(
    # CVS 版本库的位置（不是工作区，而是包含.vc 文件的版本库）
    # 可以是版本库下的子目录。
    r'/cvshome/user/jiangxin/homepage/worldhello/archive/2003/>,
```

- 导出文件的位置也在配置文件中预先设置好了，也不能再在命令行中设置。

- 导出CVS版本文件的内容至文件:`:file:`cvs2svn-tmp/git-blob.dat``。

缺省使用`:command:`cvs``命令做导出，最稳定。

```
ctx.revision_collector = GitRevisionCollector(
    'cvs2svn-tmp/git-blob.dat',

    #RCSRevisionReader(co_executable=r'co'),
    CVSRevisionReader(cvs_executable=r'cvs'),
)
```

- 另外一个导出文件的缺省位置：`:file:`cvs2svn-tmp/git-dump.dat``。

```

ctx.output_option = GitOutputOption(
    os.path.join(ctx.tmpdir, 'git-dump.dat'),

    # The blobs will be written via the revision recorder, so in
    # OutputPass we only have to emit references to the blob marks:
    GitRevisionMarkWriter(),

    # Optional map from CVS author names to git author names:
    author_transforms=author_transforms,
)

```

- 设置无提交用户信息时使用的用户名。这个用户名可以用接下来的用户映射转换为Git用户名。

```

ctx.username = 'cvs2svn'

```

- 建立CVS用户和Git用户之间的映射。Git用户名可以用Python的tuple语法(name, email)或者用字符串name <email>来表示。

```

author_transforms={
    'jiangxin' : ('Jiang Xin', 'jiangxin@ossp.com'),
    'dev1'      : u'开发者1 <dev1@ossp.com>',

    'cvs2svn'   : 'cvs2svn <admin@example.com>',
}

```

- 字符集编码。即如何转换日志中的用户名、提交说明以及文件名的编码。

对于可能在日志中出现中，必须做出下面类似设置。编码的顺序对输出也会有影响，一般将utf8放在gbk之前能保证当日志中同时出现两种编码时都能正常转换[\[2\]](#)。

```

ctx.cvs_author_decoder = CVSTextDecoder(
    [
        'utf8',
        'gbk',
    ],
    fallback_encoding='gbk'
)

ctx.cvs_log_decoder = CVSTextDecoder(
    [
        'utf8',
        'gbk',
    ],
    fallback_encoding='gbk'
)

ctx.cvs_filename_decoder = CVSTextDecoder(
    [
        'utf8',
        'gbk',
    ],
    #fallback_encoding='ascii'
)

```

- 是否忽略:`:file:`.cvsignore``文件？缺省保留:`:file:`.cvsignore``文件。

无论选择保留或是不保留，最好在转换后手工进行`:file:`.cvsignore``到`:file:`.gitignore``的转换。因为 cvs2git不能自动将`:file:`.cvsignore``文件转换为`:file:`.gitignore``文件。

```
ctx.keep_cvsignore = True
```

- 对文件换行符等的处理。下面的配置原本是针对CVS到Subversion的属性转换，但是也会影响到Git转换时的换行符设置。

维持默认值比较安全。

```
ctx.file_property_setters.extend([
    # 基于配置文件设置文件的 mime 类型
    #MimeMapper(r'/etc/mime.types', ignore_case=False),

    # 对于二进制文件 (-kb模式) 不设置 svn:eol-style 属性 (对于 Subversion 来说)
    CVSBinaryFileEOLStyleSetter(),

    # 如果文件是二进制，并且 svn:mime-type 没有设置，将其设置为 'application/octet-stream'
    CVSBinaryFileDefaultMimeTypeSetter(),

    # 如果希望根据文件的 mime 类型来判断文件的换行符，打开下面注释
    #EOLStyleFromMimeTypeSetter(),

    # 如果上面的规则没有为文件设置换行符类型，则为 svn:eol-style 设置缺省类型。
    # (二进制文件除外)
    # 缺省把文件视为二进制，不为其设置换行符类型，这样最安全。
    # 如果确认 CVS 的二进制文件都已经设置了 -kb 参数，或者使用上面的规则能够对
    # 文件类型做出正确判断，也可以使用下面参数为非二进制文件设置缺省换行符号。
    ## 'native': 服务器端文件的换行符保存为 LF，客户端根据需要自动转换。
    ## 'CRLF': 服务器端文件的换行符保存为 CRLF，客户端亦为 CRLF。
    ## 'CR': 服务器端文件的换行符保存为 CR，客户端亦为 CR。
    ## 'LF': 服务器端文件的换行符保存为 LF，客户端亦为 LF。
    DefaultEOLStyleSetter(None),

    # 如果文件没有设置 svn:eol-style，也不为其设置 svn:keywords 属性
    SVNBinaryFileKeywordsPropertySetter(),

    # 如果 svn:keywords 未色环只，基于文件的 CVS 模式进行设置。
    KeywordsPropertySetter(config.SVN_KEYWORDS_VALUE),

    # 设置文件的 svn:executable 属性，如果文件在 CVS 中标记为可执行文件。
    ExecutablePropertySetter(),
])
```

- 是否只迁移主线，忽略分支和里程碑？

缺省对所有分支和里程碑都进行转换。如果选择忽略分支和里程碑，将False修改为True。

```
ctx.trunk_only = False
```

- 分支和里程碑迁移及转换。

```
global_symbol_strategy_rules = [
    # 和正则表达式匹配的 CVS 标识，转换为 Git 的分支。
    #ForceBranchRegexpStrategyRule(r'branch.*'),

    # 和正则表达式匹配的 CVS 标识，转换为 Git 的里程碑。
    #ForceTagRegexpStrategyRule(r'tag.*'),

    # 忽略和正则表达式匹配的 CVS 标识，不进行（到Git分支/里程碑）转换。
    #ExcludeRegexpStrategyRule(r'unknown-*'),

    # 岐义的CVS标识的处理选项。
```

```

# 缺省根据使用频率自动确定转换为分支或里程碑。
HeuristicStrategyRule(),
# 或者全部转换为分支。
#AllBranchRule(),
# 或者全部转换为里程碑。
#AllTagRule(),

...

run_options.set_project()

...

# A list of symbol transformations that can be used to rename
# symbols in this project.
symbol_transforms=[
    # 是否需要重新命名里程碑? 第一个参数用于匹配, 第二个参数用于替换。
    #RegexpSymbolTransform(r'release-(\d+)_(\d+)',
    #                      r'release-\1.\2'),
    #RegexpSymbolTransform(r'release-(\d+)_(\d+)_(\d+)',
    #                      r'release-\1.\2.\3'),

```

### 使用配置文件的 cvs2git 转换过程

参照上面的方法, 从缺省的cvs2git配置文件定制, 在本地创建一个文件, 例如名为cvs2git.options文件。

- 使用cvs2git配置文件, 命令行大大简化了。

```
$ cvs2git --options cvs2git.options
```

- 成功导出后, 产生两个导出文件, 都保存在:`:file:`cvs2git-tmp``目录中。

一个保存各个文件的各个不同版本的数据内容, 即在命令行指定的输出文件:`:file:`git-blob.dat``。另外一个文件是上面命令行指定的:`:file:`git-dump.dat``用于保存各个提交相关信息(提交者、提交时间、提交日志等)。

可以看出保存文件内容的导出文件相对更大一些。

```
$ du -sh cvs2svn-tmp/*
9.8M    cvs2svn-tmp/git-blob.dat
24K      cvs2svn-tmp/git-dump.dat
```

- 创建空的Git库, 使用Git通用的数据迁移命令:`command: `git fast-import``将cvs2git的导出文件导入版本库中。

```
$ mkdir test
$ cd test
$ git init
$ cat ../cvs2svn-tmp/git-blob.dat \
  ../cvs2svn-tmp/git-dump.dat | git fast-import
```

- 检查导出结果。

```
$ git reset HEAD
$ git checkout .
$ git log -1
commit e3f12f57a77cbffcf62e19012507d041f1c9b03d
Author: Jiang Xin <jiangxin@osssxp.com>
Date:   Tue Sep 21 07:56:31 2004 +0000
```

修改邮件地址；  
修改搜索引擎；

可以看到，这一次的转换结果不但日志中的中文可以显示，而且提交者ID也转换成了Git的风格。

修改cvs2git.options中的CVS版本库地址，开始正式的转换过程。

## 迁移后版本库检查

完成迁移还不能算是大功告成，还需要进行细致的检验。

### 文件名和日志的中文

如果转换过程参考了前面的步骤和注意事项，文件名和版本库提交日志中的中文不应该出现乱码。

### 图片文件被破坏

最典型的错误就是转换后部分图片被破坏导致无法显示。这是怎么造成的呢？

CVS缺省将提交的文件以文本方式添加，除非用户在添加文件时使用了-kb参数。用命令行提交的用户经常会忘记，这就导致一些二进制文件（如图片文件）被以文本文件的方式添加到其中。文本文件在CVS检入和检出时会进行换行符转换，在服务器端换行符保存为LF，在Windows上检出时为CRLF。如果误做文本文件方式添加的图片中恰好出现CRLF，则在Windows上似乎没有问题（仍然是CRLF），但是CVS库转换成Git库后，图片文件在Windows上再检出时文件数据中原来CRLF被换成了LF，导致文件被破坏。

出现这种情况是CVS版本库使用和管理上出现了问题，应该在CVS版本库中对有问题的文件重新设置属性，标记为二进制文件。然后再进行CVS版本库到Git库的转换。

### :file:`.cvsignore` 文件的转换

CVS版本库中可能存在:file:`.cvsignore`文件用于设置文件忽略，相当于Git版本库中的:file:`.gitignore`。因为当前版本的cvs2git不能自动将:file:`.cvsignore`转换为:file:`.gitignore`，需要在版本库迁移后手工完成。CVS的:file:`.cvsignore`文件只对目录内文件有效，不会向下作用到子目录上，这一点和Git的:file:`.gitignore`相区别。还有不同就是:file:`.cvsignore`文件每一行用空格分割多个忽略，而Git每个忽略为单独的一行。

## 迁移后的测试

一个简单的检查方法是，在同一台机器上分别用CVS和Git检出（或克隆），然后比较本地的差异。要在不同的系统上（Windows, Linux）分别进行测试。

[1]	<a href="http://repo.or.cz/w/cvs2svn.git/blob/HEAD:/cvs2git-example.options">http://repo.or.cz/w/cvs2svn.git/blob/HEAD:/cvs2git-example.options</a>
[2]	部分中文的UTF8编码在GBK中存在古怪的对应 来源： <a href="https://github.com/gotgit/gotgit/blob/master/06-migrate/010-cvs.rst">https://github.com/gotgit/gotgit/blob/master/06-migrate/010-cvs.rst</a>

# SVN版本库到Git的迁移

Subversion版本库到Git版本库的转换，最好的方法就是git-svn。而git-svn的使用方法在前面“Git和SVN协同模型”一章已经详细介绍过。本章的内容将不再对git-svn的用法做过多的重复，只在这里强调一下版本库迁移时的注意事项，相关git-svn内容还请参照前面的内容。

在迁移之前要确认一个问题，Subversion转换到Git库之后，Subversion还继续使用么？意思是说还允许向Subversion提交么？

如果回答是，那么直接查看“Git和SVN协同模型”一章，用Git作为前端工具来操作Subversion版本库，而不要理会下面的内容。因为下面描述的迁移步骤针对的是一次性的Subversion到Git版本库的迁移。

**需要提交中出现git-svn-id标识么？**

如果一次性、永久性的将Subverison迁移到Git库，可以选择git-svn-id:标识不在转换后Git的提交日志中出现，这样转换后的Git库根本看不出来曾经用Subversion版本库维护过。

在git-svn的clone或者init子命令行中使用参数：`--no-metadata`。Git库的配置会自动配置`svn-remote.noMetadata`为1。之后执行：`command: `git svn fetch``时就不会在日志中产生`git-svn-id:`的标识。

**Subversion用户名到Git用户名的映射**

缺省转换后Git库的提交者ID为如下格式：`userid <userid@SVN-REPOS-UUID>`。即在邮件地址域名处以SVN版本库的UUID代替。可以在执行：`command: `git svn fetch``时通过下面的参数提供一个映射文件完成SVN用户名到Git用户名的转换。

```
-A<filename>, --authors-file=<filename>
```

即用`-A`或者`--authors-file`参数给出一个映射文件，这个文件帮助git-svn将Subversion用户名映射为Git用户名。此文件的每一行定义一个用户名映射，每一行的格式为：

```
loginname = User Name <user@example.com>
```

也可以通过下面的命令在Git库的：`file: `config``文件中设置，这样就不必在每次执行：`command: `git svn fetch``都带上这个参数。

```
$ git config svn.authorsfile /path/to/authersfile
```

当设定了用户映射文件后，如果在执行:command:`git svn fetch`是发现SVN的用户在该映射文件中没有定义，转换过程被中断。需要重新编辑用户映射文件，补充新的用户映射后，再重新执行git-svn命令。

**将Subversion分支和里程碑直接转换为Git分支和里程碑，不要放在``refs/remotes``下**

使用缺省参数执行SVN到Git的转换时，SVN的里程碑和分支转换到Git库的refs/remotes引用下。这会导致其他人从转换后的Git库克隆时，看不到Subversion原有的分支和里程碑。

当以缺省参数执行:command:`git svn init`时，Git的配置文件中会生成下面的配置：

```
[svn-remote "svn"]
fetch = trunk:refs/remotes/trunk
branches = branches/*:refs/remotes/*
tags = tags/*:refs/remotes/tags/*
```

可以直接编辑Git配置文件，将其内容调整如下：

```
[svn-remote "svn"]
fetch = trunk:refs/heads/master
branches = branches/*:refs/heads/*
tags = tags/*:refs/tags/*
```

之后再执行:command:`git svn fetch`后，就可以实现SVN的分支和里程碑正确的转换为Git库的里程碑。否则就需要将:file:`.git/refs/remots/`下的引用移动到:file:`.git/refs/heads`以及:file:`.git/refs/tags`下。

### 清除git-svn的中间文件

git-svn的中间文件位于目录:file:`.git/svn`下，删除此目录完成对git-svn转换数据库文件的清理。

来源：<https://github.com/gotgit/gotgit/blob/master/06-migrate/025-svn.rst>

# Hg版本库到Git的迁移

Mercurial（水银）是和Git同时代的、与之齐名的一款著名的分布式版本控制系统，也有相当多的使用者。就像水银又名汞，作为版本控制系统的Mercurial又称作Hg（水银元素符号）。Hg具有简单易用的优点，至少Hg提交的顺序递增的数字编号让Subversion用户感到更为亲切。Hg的开发语言除少部分因性能原因使用C语言外，大部分用Python语言开发完成，因而更易扩展，最终形成了Hg最具特色的插件系统。例如MQ就是Hg一个很有用的插件，通过Quilt式的补丁集实现对定制开发的特性分支的版本控制，当然StGit和Topgit也可以实现类似的功能。

但是Hg存在一些不足。例如服务器的存储效率不能和Git相比，服务器存储空间占用更大。Hg还不支持真正的分支，只能通过版本库克隆来进行分支开发。因为Hg不支持真正的分支，所以不能向git-svn那样完整的将Subversion版本库转换和互操作。Hg的速度相比Git要慢，尤其是网络操作没有像Git一样精确的进度显示。Hg提交只能回退一次，要想多次回退和整理版本库需要用到MQ插件。作为定制开发的利器“Hg+MQ”不适合多人协作开发而“Git+Topgit”更为适合。

不论是何原因想从Hg迁移到Git，用一个名为fast-export的转换工具可以很方便的实现。fast-export是一个用Python开发的命令行工具，可以将本地的Hg版本库迁移为Git版本库。其原理和CVS版本库迁移至Git时使用的cvs2git相仿，都是先从源版本库生成导出文件，再用Git的通用版本库转换工具:command:`git fast-import`导入到新建的Git版本库中。

安装fast-export非常简单，只要用Git克隆fast-export的版本库即可。

```
$ cd /path/to  
$ git clone git://repo.or.cz/fast-export.git
```

完成克隆后，会看到:file:`/path/to/fast-export`目录中有一个名为:file:`hg-fast-import.sh`的脚本文件，该文件封装了对相应Python脚本的调用。使用该脚本可以实现Hg版本库到Git版本库的迁移。

下面就演示一下Hg版本库到Git版本库的转换。

- 要转换的Hg版本库位于路径:file:`/path/to/hg/hello/.hg`下。

Hg不支持真正的分支，而且版本库中可能存在尚未合并的多个头指针。检查一下不要存在具有相同分支名但尚未合并的多个头指针，否则转换会失败。下面显示的该Hg版本库中具有两个具名分支r1.x和next，还有一个缺省未设置名称的头指针，因为分支名各不相同所以不会为转换过程造成麻烦。

```
$ hg heads
修改集:    7:afdd475caeee
分支:      r1.x
标签:      tip
父亲:      0:798a9568e10e
用户:      Jiang Xin <jiangxin@ossp.com>
日期:      Fri Jan 14 17:01:47 2011 +0800
描述:
start new branch: r1.x
```

```
修改集:    6:7f5a46201dda
分支:      next
用户:      Jiang Xin <jiangxin@ossp.com>
日期:      Fri Jan 14 17:01:04 2011 +0800
文件:      src/locale/zh_CN/LC_MESSAGES/helloworld.po
描述:
imported patch 100_locale_zh_cn.patch
```

```
修改集:    1:97f0a21021c6
用户:      Jiang Xin <worldhello.net AT gmail DOT com>
日期:      Sun Aug 23 23:53:05 2009 +0800
文件:      src/COPYRIGHT src/main.bak src/main.c
描述:
Fixed #6: import new upstream version hello-2.0.0
```

- 初始化一个Git版本库，该版本库就是迁移的目标版本库。

```
$ mkdir -p /path/to/my/workspace/hello
$ cd /path/to/my/workspace/hello
$ git init
Initialized empty Git repository in /path/to/my/workspace/hello/.git/
```

- 在刚刚完成初始化的Git工作区中调用:`hg-fast-export.sh`脚本完成版本库转换。

```
$ /path/to/fast-export/hg-fast-export.sh -r /path/to/hg/hello
```

- 转换完毕，执行:`git branch`会看到Hg版本库中的具名分支都转换为相应的分支，没有命名的缺省头指针转换为master分支。

```
$ git branch
* master
  next
```

r1.x

在转换后的Git版本库目录中，保存了几个用于记录版本库转换进度的状态文件（`:file:`.git/hg2git-*``），当在Git工作区不带任何参数执行`:file:`hg-fast-export.sh``命令时，会继续增量式的进行转换，将Hg版本库中的新提交迁移到Git版本库中。

如果使用了多个不同的Hg克隆版本库进行分支管理，就需要一一对Hg版本库进行转换，然后在对转换后的Git版本库进行合并。在合并Git版本库的时候可以参考下面的命令。

```
$ git remote add <name1> <path/to/repos/1>
$ git remote add <name2> <path/to/repos/2>
$ git remote update
$ git checkout -b <branch1> origin/<name1>/master
$ git checkout -b <branch2> origin/<name2>/master
```

来源：<https://github.com/gotgit/gotgit/blob/master/06-migrate/030-hg.rst>

# 通用版本库迁移

如果读者的版本控制工具在前面的迁移方案没有涉及到，也不要紧，因为很可能通过搜索引擎就能找到一款合适的迁移工具。如果找不到相应的工具，可能是您使用的版本控制工具太冷门，或者是一款不提供迁移接口的商业版本控制工具。这时您可以通过手工检入的方式或者针对Git提供的版本库导入接口：`git fast-import` 实现版本库导入。

手工检入的方式适合于只有少数几个提交或者对大部分提交历史不关心而只需要对少数里程碑版本执行导入。这种版本库迁移方式非常简单，相当于在完成Git版本库初始化后，在工作区重复执行：工作区文件清理，文件复制，执行：`git add -A` 添加到暂存区，执行：`git commit` 提交。

但是如果需要将版本库完整的历史全部迁移到新的Git版本库中，手工检入方法就不可取了，采用针对：`git fast-import` 编程是一个可以考虑的方法。Git提供了一个通用的版本库导入解决方案，即通过向命令：`git fast-import` 传递特定格式的字节流，就可以实现Git版本库的创建。工具：`git fast-import` 的导入文件格式设计的相对比较简单，当理解了其格式约定后，可以相对容易的开发出针对特定版本库的迁移工具。

下面就是一个简单的导入文件，为说明方便前面标注了行号。将这个文件保存为：`/path/to/file/dump1.dat`。

```
1 commit refs/heads/master
2 mark :1
3 committer User1 <user1@ossp.com> 1295312699 +0800
4 data <<EOF
5 My initial commit.
6 EOF
7 M 644 inline README
8 data <<EOF
9 Hello, world.
10 EOF
11 M 644 inline team/user1.txt
12 data <<EOF
13 I'm user1.
14 EOF
```

上面这段文字应该这样理解：

- 第1行以`commit`开头，标记一个提交的开始。该提交会创建（或更新）引用`refs/heads/master`。
- 第2行以`mark`开头，是一个标记指令，将这个提交用“`:1`”标示以方便后面的提交参

照。

- 第3行记录了这个提交的提交者是User1，邮件地址为<user1@osssxp.com>，提交时间则采用Unix时间格式。
- 第4-6行是该提交的提交说明，提交说明用data数据块的方式进行定义。
- 第4行在data语句后紧接着的<<EOF>>含义为data的内容到以EOF标记的行截止。这样的表示法称为“Here Documents”表示法。
- 第7行以字母M开头，含义是修改（或新建）了一个文件，文件名为:file:`README`，而文件的内容以inline的方式提供。
- 第8-10行则是以内联（inline）数据块的方式提供:file:`README`文件的内容。
- 第11行定义了该提交修改的第二个文件:file:`team/user1.txt`。该文件的内容也是以内联（inline）的方式给出。
- 第12-14行给出文件:file:`team/user1.txt`的内容。

下面初始化一个新的版本库，并通过导入文件:file:`/path/to/file/dump1.dat`的方式为版本库注入数据。

- 初始化版本库。

```
$ mkdir -p /path/to/my/workspace/import  
$ cd /path/to/my/workspace/import  
$ git init
```

- 调用:command:`git fast-import`命令。

```
$ git fast-import < /path/to/file/dump1.dat  
git-fast-import statistics:  
-----  
Alloc'd objects:      5000  
Total objects:        5 (          0 duplicates           )  
    blobs :        2 (          0 duplicates           0 deltas)  
    trees :        2 (          0 duplicates           0 deltas)  
    commits:       1 (          0 duplicates           0 deltas)  
    tags :         0 (          0 duplicates           0 deltas)  
Total branches:       1 (          1 loads     )  
    marks:        1024 (          1 unique     )  
    atoms:         3  
Memory total:        2344 KiB  
    pools:        2110 KiB  
    objects:       234 KiB  
-----  
pack_report: getpagesize() =        4096  
pack_report: core.packedGitWindowSize = 1073741824  
pack_report: core.packedGitLimit = 8589934592  
pack_report: pack_used_ctr =        1  
pack_report: pack_mmap_calls =        1
```

```
pack_report: pack_open_windows      =      1 /      1  
pack_report: pack_mapped          =    323 /    323  
-----
```

- 看看提交日志。

```
$ git log --pretty=fuller --stat  
commit 18f4310580ca915d7384b116fcb2e2ca0b833714  
Author: User1 <user1@osssxp.com>  
AuthorDate: Tue Jan 18 09:04:59 2011 +0800  
Commit: User1 <user1@osssxp.com>  
CommitDate: Tue Jan 18 09:04:59 2011 +0800  
  
My initial commit.  
  
 README      |      1 +  
team/user1.txt |      1 +  
 2 files changed, 2 insertions(+), 0 deletions(-)
```

再来看一个导入文件。将下面的内容保存到文件:[file: `/path/to/file/dump2.dat`](#)中。

```
1 blob  
2 mark :2  
3 data 25  
4 Hello, world.  
5 Hi, user2.  
6 blob  
7 mark :3  
8 data <<EOF  
9 I'm user2.  
10 EOF  
11 commit refs/heads/master  
12 mark :4  
13 committer User2 <user2@osssxp.com> 1295312799 +0800  
14 data <<EOF  
15 User2's test commit.  
16 EOF  
17 from :1  
18 M 644 :2 README  
19 M 644 :3 team/user2.txt
```

上面的内容标注了行号，注意不要把行号也代入文件中。其中：

- 第1-5行定义了编号为“:2”的文件内容。该文件的内容共有25字节，第3行开始的data文字块就通过在后面跟上一个表示文件长度的十进制数字界定了内容的起止。
- 第6-10行定义了编号为“:3”的文件内容。第8行界定该文件内容使用了“Here

Documents”的语法，该语法对于文本内容比较适合，使用内容长度标示内容起止对于二进制文件更为适合。

- 第11行开始定义了一个新的提交。
- 第12行设定该提交的编号为“:4”。
- 第17行以from开头，定义了该提交的父提交为编号为“:1”的提交，即在:file:`/path/to/file/dump1.dat`中定义的提交。
- 第18行和第19行设定了该提交更改的两个文件，这两个文件的内容不像之前的导出文件dump1.dat那样使用内联方式定义内容，而是采用引用方式引用前面定义的二进制数据流(blob)作为文件的内容。

如果以增量方式导入:file:`dump2.dat`会报错，因为在第17行引用的“:1”没有定义。

```
$ git fast-import < /path/to/file/dump2.dat
fatal: mark :1 not declared
fast-import: dumping crash report to .git/fast_import_crash_21772
```

如果将文件:file:`/path/to/file/dump2.dat`的第17行的引用修改为提交ID，是可以增量导入的。不过为了说明的方便，还是通过将两个导入文件一次性传递给:command:`git fast-import`创建一个新版本库。

- 初始化版本库import2。

```
$ mkdir -p /path/to/my/workspace/import2
$ cd /path/to/my/workspace/import2
$ git init
```

- 调用:command:`git fast-import`命令。

```
$ cat /path/to/file/dump1.dat \
  /path/to/file/dump2.dat | git fast-import
```

- 导入之后的日志显示：

```
$ git log --graph --stat
* commit 73a6f2742f9da7c1b4bb8748e018a2becad39dd6
| Author: User2 <user2@osxp.com>
| Date:   Tue Jan 18 09:06:39 2011 +0800
|
|       User2's test commit.
|
| README           |    1 +
| team/user2.txt |    1 +
```

```
| 2 files changed, 2 insertions(+), 0 deletions(-)
|
* commit 18f4310580ca915d7384b116fcb2e2ca0b833714
  Author: User1 <user1@osssxp.com>
  Date:   Tue Jan 18 09:04:59 2011 +0800

    My initial commit.

  README      |    1 +
  team/user1.txt |    1 +
  2 files changed, 2 insertions(+), 0 deletions(-)
```

下面再来看一个导入文件，在这个导入文件中，包含了合并提交以及创建里程碑。

```
1 blob
2 mark :5
3 data 25
4 Hello, world.
5 Hi, user1.
6 blob
7 mark :6
8 data 35
9 Hello, world.
10 Hi, user1 and user2.
11 commit refs/heads/master
12 mark :7
13 committer User1 <user1@osssxp.com> 1295312899 +0800
14 data <<EOF
15 Say hello to user1.
16 EOF
17 from :1
18 M 644 :5 README
19 commit refs/heads/master
20 mark :8
21 committer User2 <user2@osssxp.com> 1295312900 +0800
22 data <<EOF
23 Say hello to both users.
24 EOF
25 from :4
26 merge :7
27 M 644 :6 README
28 tag refs/tags/v1.0
29 from :8
30 tagger Jiang Xin <jiangxin@osssxp.com> 1295312901 +0800
31 data <<EOF
32 Version v1.0
33 EOF
```

将这个文件保存到`:file:`/path/to/file/dump3.dat``。下面针对该文件内容进行简要的说明：

- 第1–5行和第6–10行定义了两个blob对象，代表了两个对`:file:`README``文件的不同修改。
- 第11行开始定义了编号为“:7”的提交。从第17行可以看出该提交的父提交也是由`:file:`dump1.dat``导入的第一个提交。
- 第19行开始定义了编号为“:8``”的提交。该提交为一个合并提交，除了在第25行设定了第一个父提交外，还由第26行给出了第二个父提交。
- 第28行开始定义了一个里程碑。里程碑的名字为`refs/tags/v1.0`。第29行指定了该里程碑对应的提交。里程碑说明由第31–33行指令给出。
- 初始化版本库`import3`。

```
$ mkdir -p /path/to/my/workspace/import3  
$ cd /path/to/my/workspace/import3  
$ git init
```

- 调用`:command:`git fast-import``命令。

```
$ cat /path/to/file/dump1.dat /path/to/file/dump2.dat \  
/path/to/file/dump3.dat | git fast-import
```

- 查看创建的版本库的日志。

从日志中可以看出里程碑v1.0已经建立在最新的提交上了。

```
$ git log --oneline --graph --decorate  
*   a47790e (HEAD, tag: refs/tags/v1.0, master) Say hello to both users.  
|\\  
| * f486a44 Say hello to user1.  
* | 73a6f27 User2's test commit.  
|/  
* 18f4310 My initial commit.
```

理解了`git fast-import`的导入文件格式，针对特定的版本控制系统开发一个新的迁移工具不是坏事。Hg的迁移工具`:command:`fast-export``是一个很好的参照。

## Git版本库整理

Git提供了太多武器进行版本库的整理，可以将一个Git版本库改动换面成为另外的一个Git版本库。

- 使用交互式变基操作，将多个提交合并为一个。
- 使用StGit，合并提交以及更改提交。
- 借助变基操作，抛弃部分历史提交。
- 使用子树合并，将多个版本库整合在一起。
- 使用git-subtree插件，将版本库的一个目录拆分出来成为独立版本库的根目录。

但是有些版本库重整工作如果使用上面的工具会非常困难，而采用另外一个还没有用到的Git命令`:command:`git filter-branch``却可以做到事半功倍。看看使用这个新工具来实现下面的这几个任务是多么的简单和优美。

- 将版本库中某个文件彻底删除。即凡是有该文件的提交都一一作出修改，撤出该文件。

下面的命令并非最优实现，后面会介绍一个运行更快的命令。

```
$ git filter-branch --tree-filter 'rm -f filename' -- --all
```

- 更改历史提交中某一提交者的姓名及邮件地址。

```
$ git filter-branch --commit-filter '  
if [ "$GIT_AUTHOR_NAME" = "Xin Jiang" ]; then  
    GIT_AUTHOR_NAME="Jiang Xin"  
    GIT_AUTHOR_EMAIL="jiangxin@osxp.com"  
    GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"  
    GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"  
fi  
git commit-tree "$@";  
' HEAD
```

- 为没有包含签名的历史提交添加签名。

```
$ git filter-branch -f --msg-filter '  
signed=false  
while read line; do  
    if echo $line | grep -q Signed-off-by; then  
        signed=true  
    fi  
    echo $line  
done  
if ! $signed; then  
    echo ""  
    echo "Signed-off-by: YourName <your@email.address>"  
fi  
' HEAD
```

通过上面的例子，可以看出命令`:command:`git filter-branch``通过针对不同的过滤器提供可执行脚本，从不同的角度对Git版本库进行重构。该命令的用法：

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]  
[--index-filter <command>] [--parent-filter <command>]  
[--msg-filter <command>] [--commit-filter <command>]  
[--tag-name-filter <command>] [--subdirectory-filter <directory>]  
[--prune-empty]
```

```
[--original <namespace>] [-d <directory>] [-f | --force]
[--] [<rev-list options>...]
```

这条命令异常复杂，但是大部分参数是用于提供不同的接口，因此还是比较好理解的。

- 该命令最后的<rev-list>参数提供要修改的版本范围，如果省略则相当于HEAD指向的当前分支。也可以使用--all来指代所有引用，但是要在--all和前面的参数间使用分隔符--。
- 运行`:command:`git filter-branch``命令改写分支之后，被改写的分支会在`refs/original`中对原始引用做备份。对于在`refs/original`中已有备份的情况下，该命令拒绝执行，除非使用-f或--force参数。
- 其他需要接以<command>的参数都为`:command:`git filter-branch``提供相应的接口进行过滤，在下面会针对各个过滤器进行介绍。

## 环境变量过滤器

参数`--env-filter`用于设置一个环境过滤器。该过滤器用于修改环境变量，对特定的环境变量的修改会改变提交。下面的示例[1]可以用于修改作者/提交者的邮件地址。

```
$ git filter-branch --env-filter '
  an="$GIT_AUTHOR_NAME"
  am="$GIT_AUTHOR_EMAIL"
  cn="$GIT_COMMITTER_NAME"
  cm="$GIT_COMMITTER_EMAIL"

  if [ "$cn" = "Kanwei Li" ]; then
    cm="kanwei@gmail.com"
  fi
  if [ "$an" = "Kanwei Li" ]; then
    am="kanwei@gmail.com"
  fi

  export GIT_AUTHOR_EMAIL=$am
  export GIT_COMMITTER_EMAIL=$cm
'
```

这个示例和本节一开始介绍的更改作者/提交者信息的示例功能相同，但是使用了不同的过滤器，读者可以根据喜好选择。

## 树过滤器

参数`--tree-filter`用于设置树过滤器。树过滤器会将每个提交检出到特定目录中`(:file:`.git-rewrite`/ 目录或者用-d参数指定的目录)`，针对检出目录中文件的修改、添加、删除会改变提交。注意此过滤器忽略`:file:`.gitignore``，因此任何对检出目录的修改都会记录在新的提交中。之前介绍的文件删除就是一例。再比如对文件名的修改：

```
$ git filter-branch --tree-filter '
  [ -f oldfile ] && mv oldfile newfile || true
  ' -- --all
```

## 暂存区过滤器

树过滤器因为要将每个提交检出，因此非常费时，而参数`--index-filter`给出的暂存区过滤器则没有这个缺点。之前使用树过滤器删除文件的操作如果换做用暂存区过滤器实现运行的会更快。

```
$ git filter-branch --index-filter '  
  git rm --cached --ignore-unmatch filename  
  ' -- --all
```

其中参数`--ignore-unmatch`让`:command:`git rm``命令不至于因为暂存区中不存在`:file:`filename``文件而失败。

## 父节点过滤器

参数`--parent-filter`用于设置父节点过滤器。该过滤器用于修改提交的父节点。提交原始的父节点通过标准输入传入脚本，而脚本的输出将作为提交新的父节点。父节点参数的格式为：如果没有父节点（初始提交）则为空。如果有一个父节点，参数为`-p parent`。如果是合并提交则有多个父节点，参数为`-p parent1 -p parent2 -p parent3 ...`。

下面的命令将当前分支的初始提交嫁接到`<graft-id>`所指向的提交上。

```
$ git filter-branch --parent-filter 'sed "s/^$/-p <graft-id>/"' HEAD
```

如果不是将初始提交（没有父提交）而是任意的一个提交嫁接到另外的提交上，可以通过`GIT_COMMIT`环境变量对提交进行判断，更改其父节点以实现嫁接。

```
$ git filter-branch --parent-filter \  
  'test $GIT_COMMIT = <commit-id> && \  
  echo "-p <graft-id>" || cat  
  ' HEAD
```

关于嫁接，Git可以通过配置文件`:file:`.git/info/grafts``实现，而`:command:`git filter-branch``命令可以基于该配置文件对版本库实现永久性的更改。

```
$ echo "$commit-id $graft-id" >> .git/info/grafts  
$ git filter-branch $graft-id..HEAD
```

## 提交说明过滤器

参数`--msg-filter`用于设置提交说明过滤器。该过滤器用于改写提交说明。原始的提交说明作为标准输入传入脚本，而脚本的输出作为新的提交说明。

例如将使用git-svn从Subversion迁移过来的Git版本库，缺省情况下在提交说明中饱含`git-svn-id:`字样的说明，如果需要将其清除可以不必重新迁移，而是使用下面的命令重写提交说明。

```
$ git filter-branch --msg-filter 'sed -e "/^git-svn-id:/d"' -- --all
```

再如将最新的10个提交添加“Acked-by:”格式的签名。

```
$ git filter-branch --msg-filter '  
  cat &&  
  echo "Acked-by: Bugs Bunny <bunny@bugzilla.org>"  
  ' HEAD~10..HEAD
```

## 提交过滤器

参数`--commit-filter`用于设置树过滤器。提交过滤器所给出的脚本，在版本库重整过程的

每次提交时运行，取代缺省要执行的:command:`git commit-tree`命令。不过一般情况会在脚本中调用:command:`git commit-tree`命令。传递给脚本的参数格式为<TREE\_ID> [(-p <PARENT\_COMMIT\_ID>)...], 提交日志以标准输入的方式传递给脚本。脚本的输出是新提交的提交ID。作为扩展，如果脚本输出了多个提交ID，则这些提交ID作为子提交的多个父节点。

使用下面的命令，可以过滤掉空提交（合并提交除外）。

```
$ git filter-branch --commit-filter 'git_commit_non_empty_tree "$@"'
```

函数git\_commit\_non\_empty\_tree函数是在脚本:command:`git-filter-branch`中已经定义过的函数。可以打开文件:file:`\$(git --exec-path)/git-filter-branch`查看。

```
# if you run 'git_commit_non_empty_tree "$@"' in a commit filter,
# it will skip commits that leave the tree untouched, commit the other.
git_commit_non_empty_tree()
{
    if test $# = 3 && test "$1" = $(git rev-parse "$3^{tree}"); then
        map "$3"
    else
        git commit-tree "$@"
    fi
}
```

如果想某个用户的提交非空但是也想跳过，可以使用下面的命令：

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_NAME" = "badboy" ];
  then
      skip_commit "$@";
  else
      git commit-tree "$@";
  fi' HEAD
```

其中函数skip\_commit也是在:file:`git-filter-branch`脚本中已经定义好的。该函数的作用就是将传递给提交过滤器脚本的参数<tree\_id> -p parent1 -p parent2 ...进行处理，形成parent1 parent2的输出。参见Git命令脚本:file:`\$(git --exec-path)/git-filter-branch`中相关函数。

```
# if you run 'skip_commit "$@"' in a commit filter, it will print
# the (mapped) parents, effectively skipping the commit.
skip_commit()
{
    shift;
    while [ -n "$1" ];
    do
        shift;
        map "$1";
        shift;
    done;
}
```

## 里程碑名字过滤器

参数--tag-name-filter用于设置里程碑名字过滤器。该过滤器也是经常要用到的过滤器。上面介绍的各个过滤器都有可能改变提交ID，如果在原有的提交ID上建有里程碑，可能会随之更新但是会产生大量的警告日志，提示使用里程碑过滤器。里程碑过滤器脚本以原始里程碑名称作为标准输入，并把新里程碑名称作为标准输出。如果不打算变更里程碑名

称，而只是希望里程碑随提交而更新，可以在脚本中使用:command:`cat`命令。例如下面的命令中里程碑名字过滤器和目录树过滤器同时使用。

```
$ git filter-branch --tree-filter '
[ -f oldfile ] && mv oldfile newfile || true
' -- tag-name-filter 'cat' -- --all
```

在前面的里程碑一章曾经提到过:command:`git branch`命令没有提供里程碑重命名的功能，而使用里程碑名字过滤器可以实现里程碑的重命名。下面的示例会修改里程碑的名字，将前缀为old-prefix的里程碑改名为前缀为new-prefix的里程碑。

```
$ git filter-branch --tag-name-filter '
oldtag=`cat`'
newtag=${oldtag#old-prefix}'
if [ "$oldtag" != "$newtag" ]; then
    newtag="new-prefix$newtag"
fi
echo $newtag
'
```

注意因为签名里程碑重建后，因为签名不可能保持所以新里程碑会丢弃签名，成为一个普通的包含说明的里程碑。

## 子目录过滤器

参数--subdirectory-filter用于设置子目录过滤器。子目录过滤器可以将版本库的一个子目录提取为一个新版本库，并将该子目录作为版本库的根目录。例如从Subversion转换到Git版本库因为参数使用不当，将原Subversion的主线转换为Git版本库的一个目录:file:`trunk`。可以使用:command:`git filter-branch`命令的子目录过滤器将:file:`trunk`提取为版本库的根。

```
$ git filter-branch --subdirectory-filter trunk HEAD
```

[1]	摘自 <a href="http://kanwei.com/code/2009/03/29/fixing-git-email.html">http://kanwei.com/code/2009/03/29/fixing-git-email.html</a> 。 来源: <a href="https://github.com/gotgit/gotgit/blob/master/06-migrate/050-git-to-git.rst">https://github.com/gotgit/gotgit/blob/master/06-migrate/050-git-to-git.rst</a>
-----	---

## etckeeper

Linux/Unix的用户对:`/etc`目录都是再熟悉不过了，在这个最重要的目录中保存了大部分软件的配置信息，借以实现软件的配置以及整个系统的启动过程控制。对于Windows用户来说，可以把:`/etc`目录视为Windows中的注册表，只不过文件化了，可管理了。

这么重要的:`/etc`目录，如果其中的文件被错误编辑或者删除，将会损失惨重。`etckeeper`这个软件可以帮助实现:`/etc`目录的持续备份，借用分布式版本控制工具，如：`git`、`mercurial`、`bazaar`、`darcs`。

那么`etckeeper`是如何实现的呢？以`git`作为`etckeeper`的后端为例进行说明，其他的分布式版本控制系统大同小异。

- 将:`/etc`目录Git化。将会创建Git库于目录:`/etc/.git`中，`/etc`目录作为工作区。
- 与系统的包管理器，如Debian/Ubuntu的apt，Redhat上的yum等整合。一旦有软件包安装或删除，对:`/etc`目录下的改动执行提交操作。
- 除了能够记录:`/etc`目录中的文件内容，还可以记录文件属性等元信息。因为:`/etc`目录下的文件的权限设置往往是非常重要和致命的。
- 因为:`/etc`目录已经是一个版本库了，可以用git命令对:`/etc`下的文件进行操作：查看历史，回退到历史版本...
- 也可以将:`/etc`克隆到另外的主机中，实现双机备份。

## 安装etckeeper

安装`etckeeper`是一个最简单的活，因为`etckeeper`在主流的Linux发行版都有对应的安装包。使用相应Linux平台的包管理器（apt、yum）即可安装。

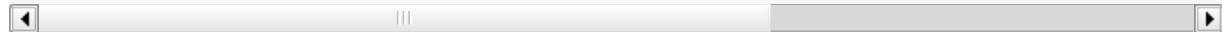
在Debian/Ubuntu上安装`etckeeper`如下：

```
$ sudo aptitude install etckeeper
```

安装`etckeeper`软件包，会自动安装上一个分布式版本控制系统工具，除非已经安装了。这是因为`etckeeper`需要使用一个分布式版本控制系统作为存储管理后端。在Debian/Ubuntu上会依据下面的优先级进行安装：`git` > `mercurial` > `bzr` > `darcs`。

在Debian/Ubuntu上，使用:`dpkg -s`命令查看`etckeeper`的软件包依赖，就会看到这个优先级。

```
$ dpkg -s etckeeper | grep "Depends"
Depends: git-core (>= 1:1.5.4) | git (>= 1:1.7) | mercurial | bzr (>= 1.4~) |
```



## 配置etckeeper

配置etckeeper首先要选择好某一分布式版本库控制工具，如Git，然后用相应的版本控制工具初始化:`/etc`目录，并做一次提交。

- 编辑配置文件:`/etc/etckeeper/etckeeper.conf`。

只要有下面一条配置就够了。告诉etckeeper使用git作为数据管理后端。

```
VCS="git"
```

- 初始化:`/etc`目录。即将其Git化。执行下面的命令（需要以root用户身份），会将:`/etc`目录Git化。

整个过程可能会比较慢，因为要对:`/etc`下的文件执行:`git add`，因为文件太多，会慢一些。

```
$ sudo etckeeper init
```

- 执行第一次提交。注意使用etckeeper命令而非git命令进行提交。

```
$ sudo etckeeper commit "this is the first etckeeper commit..."
```

整个过程可能会比较慢，主要是因为etckeeper要扫描:`/etc`下不属于root用户的文件以及特殊权限的文件并进行记录。这是为了弥补Git本身不能记录文件属主、权限信息等。

## 使用etckeeper

实际上由于etckeeper已经和系统的包管理工具进行了整合（如Debian/Ubuntu的apt，Redhat上的yum等），etckeeper可以免维护运行。即一旦有软件包安装或删除，对:`/etc`目录下的改动会自动执行提交操作。

当然也可以随时以root用户身份调用:`etckeeper commit`命令对:`/etc`目录的改动手动进行提交。

剩下的工作就交给Git了。可以在`:file:`/etc``目录执行`:command:`git log``、`:command:`git show``等操作。但要注意以root用户身份运行，因为`:file:`/etc/.git``目录的权限不允许普通用户操作。

# Gistore

当了解了etckeeper之后，读者可能会如我一样的提问到：“有没有像etckeeper一样的工具，但是能备份任意的文件和目录呢？”

我在Google上搜索类似的工具无果，终于决定动手开发一个，因为无论是我还是我的客户，都需要一个更好用的备份工具。这就是Gistore。

Gistore = Git + Store

2010年1月，我在公司的博客上发表了Gistore 0.1版本的消息，参见：<http://blog.ossxp.com/2010/01/406/>。并将Gistore的源代码托管在了Github上，参见：<http://github.com/ossxp-com/gistore>。

Gistore出现受到了etckeeper的启发，通过Gistore用户可以对全盘任何目录的数据纳入到备份中，定制非常简单和方便。特点有：

- 使用Git作为数据后端。数据回复和历史查看等均使用熟悉的Git命令。
- 每次备份即为一次Git提交，支持文件的添加/删除/修改/重命名等。
- 每次备份的日志自动生成，内容为此次修改的摘要信息。
- 支持备份回滚，可以设定保存备份历史的天数，让备份的空间占用维持在一个相对稳定的水平上。
- 支持跨卷备份。备份的数据源可以来自任何卷/目录或者文件。
- 备份源如果已经Git化，也能够备份。例如：`:file:`/etc`` 目录因为etckeeper被Git化，仍然可以对其用gistore进行备份。
- 多机异地备份非常简单，使用git克隆即可解决。可以采用git协议、http、或者更为安全的ssh协议。

说明：Gistore只能运行在Linux/Unix上，而且最好以root用户身份运行，以避免因为授权问题导致有的文件不能备份。

## Gistore的安装

### 从源码安装Gistore

从源代码安装Gistore，可以确保安装的是最新的版本。

- 先用git从Github上克隆代码库。

```
$ git clone git://github.com/ossxp-com/gistore.git
```

- 执行setup.py脚本完成安装

```
$ cd gistore  
$ sudo python setup.py install  
  
$ which gistore  
/usr/local/bin/gistore
```

## 用easy\_install安装

Gistore是用Python语言开发，已经在PYPI上注册：<http://pypi.python.org/pypi/gistore>。就像其他Python软件包一样，可以用easy\_install进行安装。

- 确保机器上已经安装了setuptools。

Setuptools的官方网站在<http://peak.telecommunity.com/DevCenter/setuptools>。几乎每个Linux发行版都有setuptools的软件包，因此可以直接用包管理器进行安装。

在Debian/Ubuntu上可以使用下面的命令安装setuptools：

```
$ sudo aptitude install python-setuptools  
  
$ which easy_install  
/usr/bin/easy_install
```

- 使用:command:`easy\_install`命令安装Gistore

```
$ sudo easy_install -U gistore
```

## Gistore的使用

先熟悉一下Gistore的术语：

- 备份库：通过:command:`gistore init`命令创建的，用于数据备份的数据仓库。备份库包含的数据有：
  - Git版本库相关目录和文件。如:file:`repo.git`目录（相当于:file:`.git`目录），:file:`.gitignore`文件等。
  - Gistore相关配置。如:file:`.gistore/config`文件。

- 备份项：可以为一个备份库指定任意多的备份项目。
  - 例如备份`:file:`/etc``目录，`:file:`/var/log``目录，`:file:`/boot/grub/menu.lst``文件等。
  - 备份项在备份库的`:file:`.gistore/config``文件中指定，如上述备份项在配置文件中写法：

```
[store /etc]
[store /var/log]
[store /boot/grub/menu.lst]
```

- 备份任务：在执行`:command:`gistore``命令时，可以指定一个任务或者多个任务。
  - 备份任务可以是对应的备份库的路径。可以使用绝对路径，也可以使用相对路径。
  - 如果不提供备份任务，缺省将当前目录作为备份库的所在。
  - 也可以使用一个任务别名来标识备份任务。
- 任务别名。
  - 在`:file:`/etc/gistore/tasks``目录中创建的备份库的符号链接的名称，作为这些备份库的任务别名。
  - 通过任务别名的机制，将可能分散在磁盘各处的备份库汇总一起，便于管理员定位备份库。
  - 将所有的别名显示出来，就是任务列表。

## 创建并初始化备份库

在使用Gistore开始备份之前，必须先初始化一个备份库。命令行格式如下：

```
gistore init [备份任务]
```

初始化备份库的示例如下：

- 将当前目录作为备份库进行初始化：

```
$ mkdir backup
$ cd backup
$ gistore init
```

- 将指定的目录作为备份库进行初始化：

```
$ sudo gistore init /backup/database
```

当一个备份库初始化完毕后，包含下列文件和目录：

- 目录:`:file:`repo.git``：存储备份的Git版本库。
- 文件:`:file:`.gistore/config``：Gistore配置文件。
- 目录:`:file:`logs``：Gistore运行的日志记录。
- 目录:`:file:`locks``：Gistore运行的文件锁目录。

## Gistore的配置文件

在每一个备份库的`:file:`.gistore``目录下的`:file:`config``文件是该备份库的配置文件，用于记录Gistore的备份项内容以及备份回滚设置等。

例如下面的配置内容：

```
1 # Global config for all sections
2 [main]
3 backend = git
4 backup_history = 200
5 backup_copies = 5
6 root_only = no
7 version = 2
8
9 [default]
10 keep_empty_dir = no
11 keep_perm = no
12
13 # Define your backup list below. Section name begin with 'store ' will be
14 # eg: [store /etc]
15 [store /opt/mailman/archives]
16 [store /opt/mailman/conf]
17 [store /opt/mailman/lists]
18 [store /opt/moin/conf]
19 [store /opt/moin/sites]
```

如何理解这个配置文件呢？

- 第2行到第7行的`[main]`小节用于Gistore的全局设置。
- 第3行设置了Gistore使用的SCM后端为Git，这是目前唯一可用的设置。
- 第4行设置了Gistore的每一个历史分支保存的最多的提交数目，缺省200个提交。当超过这个提交数目，进行备份回滚。

- 第5行设置了Gistore保存的历史分支数量，缺省5个历史分支。每当备份回滚时，会将备份主线保存到名为`gistore/1`的历史分支。
- 第6行设置非`root_only`模式。如果开启`root_only`模式，则只有root用户能够执行此备份库的备份。
- 第7行设置了Gistore备份库的版本格式。
- 第9行开始的`[default]`小节设置后面的备份项小节的缺省设置。在后面的`[store ...]`小节可以覆盖此缺省设置。
- 第10行设置是否保留空目录。暂未实现。
- 第11行设置是否保持文件属主和权限。暂未实现。
- 第15行到第19行是备份项小节，小节名称以`store`开始，后面的部分即为备份项的路径。

如`[store /etc]`的含义是：要对`:file:`/etc``目录进行备份。

## Gistore的备份项管理

当然可以直接编辑`:file:`.gistore/config``文件，通过添加或者删除`[store...]`小节的方式管理备份项。Gistore还提供了两个命令进行备份项的管理。

### 添加备份项

进入备份库目录，执行下面的命令，添加备份项`:file:`/some/dir``。注意备份项要使用全路径，即要以“/”开始。

```
$ gistore add /some/dir
```

### 删除备份项

进入备份库目录，执行下面的命令，策删除备份项`:file:`/some/dir``。

```
$ gistore rm /some/dir
```

### 查看备份项

进入备份库目录，执行`:command:`gistore status``命令，显示备份库的设置以及备份项列表。

```
$ gistore status
    Task name : system
    Directory : /data/backup/gistore/system
    Backend : git
Backup capability : 200 commits * 5 copies
    Backup list :
        /backup/databases (--)
        /backup/ldap (--)
        /data/backup/gistore/system/.gistore (--)
        /etc (AD)
        /opt/cosign/conf (--)
        /opt/cosign/factor (--)
        /opt/cosign/lib (--)
        /opt/gosa/conf (--)
        /opt/ossxp/conf (--)
        /opt/ossxp/ssl (--)
```

从备份库的状态输出，可以看到：

- 备份库的路径是:`:file:`/data/backup/gistore/system``。
- 备份库有一个任务别名为`system`。
- 备份的容量是`200*5`，如果按每天一次备份计算的话，总共保存`1000`天，差不多`3`年的数据备份。
- 在备份项列表，可以看到多达`10`项备份列表。

每个备份项后面的括号代表其备份选项，其中`:file:`/etc``的备份选项为`AD`。`A`代表记录并保持授权，`D`的含义是保持空目录。

## 执行备份任务

执行备份任务非常简单：

- 进入到备份库根目录下，执行：

```
$ sudo gistore commit
```

- 或者在命令行上指定备份库的路径。

```
$ sudo gistore ci /backup/database
```

说明：`ci`为`commit`命令的简称。

## 查看备份日志及数据

备份库中的:`:file:`repo.git``就是备份数据所在的Git库，这个Git库是一个不带工作区的裸库。可以对其执行:`:command:`git log``命令来查看备份日志。

因为并非采用通常:`:file:`.git``作为版本库名称，而且不带工作区，需要通过`--git-dir`参数制定版本库位置，如下：

```
$ git --git-dir=repo.git log
```

当然，也可以进入到:`:file:`repo.git``目录，执行:`:command:`git log``命令。

下面是我公司内的服务器每日备份的日志片断：

```
commit 9d16b5668c1a09f6fa0b0142c6d34f3cbb33072f
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Aug 5 04:00:23 2010 +0800

Changes summary: total= 423, A: 407, D: 1, M: 15
-----
A => etc/gistore/tasks/Makefile, opt/cosign/lib/share/locale/cosign.p
D => etc/gistore/tasks/default
M => .gistore/config, etc/gistore/tasks/gosa, etc/gistore/tasks/testl

commit 01b6bce2e4ee2f8cda57ceb3c4db0db9eb90bbcd
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Wed Aug 4 04:01:09 2010 +0800

Changes summary: total= 8, A: 7, M: 1
-----
A => backup/databases/blog_bj/blog_bj.sql, backup/databases/ossp/mys
M => .gistore/config

commit 15ef2e88f33dfa7dfb04ecbcdb9e6b2a7c4e6b00
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Tue Aug 3 16:59:12 2010 +0800

Changes summary: total= 2665, A: 2665
-----
A => .gistore/config, etc/apache2/sites-available/gems, etc/group-, e

commit 6883d5c2ca77caab9f9b2cf68dc6c27526731c8
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Tue Aug 3 16:55:49 2010 +0800

gistore root commit initialized.
```



从上面的日志可以看出：

- 备份发生在晚上4点钟左右。这是因为备份是晚上自动执行的。
- 最老的备份，即ID为6883d5c的提交，实际上是一个不包含任何数据的空备份，在数据发生回滚的时候，设置为回滚的起点。这个后面会提到。
- ID为15ef2e8的提交是一次手动提交。提交说明中可以看到添加了2665个文件。
- 最新的备份ID为9d16b56，其中既又文件添加（A），又有文件删除（D），还有文件变更（M），会随机选择各5个文件出现在提交日志中。

如果想查看详细的文件变更列表？

使用下面的命令：

```
$ git --git-dir=repo.git show --stat 9d16b56

commit 9d16b5668c1a09f6fa0b0142c6d34f3cbb33072f
Author: Jiang Xin <jiangxin@osxp.com>
Date:   Thu Aug 5 04:00:23 2010 +0800

Changes summary: total= 423, A: 407, D: 1, M: 15
-----
A => etc/gistore/tasks/Makefile, opt/cosign/lib/share/locale/cosign.p
D => etc/gistore/tasks/default
M => .gistore/config, etc/gistore/tasks/gosa, etc/gistore/tasks/test1

.gistore/config | 4 +
backup/databases/redmine/redmine.sql | 44 ++
etc/apache2/include/redmine/redmine.conf | 40 ++
etc/gistore/tasks/Makefile | 1 +
etc/gistore/tasks/default | 1 -
etc/gistore/tasks/gosa | 2 ++
...
opt/gosa/conf/sieve-spam.txt | 6 +
opt/gosa/conf/sieve-vacation.txt | 4 +
opt/osxp/conf/cron.d/osxp-backup | 8 ++
423 files changed, 30045 insertions(+), 51 deletions(-)
```



在备份库的`:file:`logs``目录下，还有一个备份过程的日志文件`:file:`logs/gitstore.log``。记录了每次备份的诊断信息，主要用于调试Gistore。

## 查看及恢复备份数据

所有的备份数据，实际上都在`:file:`repo.git``目录指向的Git库中维护。如何获取呢？

### 克隆方式检出

执行下面的命令，克隆裸版本库`repo.git`：

```
$ git clone repo.git data
```

进入`:file:`data``目录，就可以以Git的方式查看历史数据，以及恢复历史数据。当然恢复的历史数据还要拷贝到原始位置才能实现数据的恢复。

### 分离的版本库和工作区方式检出

还有一个稍微复杂的方法，就是既然版本库已经在`repo.git`了，可以直接利用它，避免克隆导致空间上的浪费，尤其是当备份库异常庞大的情况。

- 创建一个工作目录，如`:file:`export``。

```
$ mkdir export
```

- 设置环境变量，制定版本库和工作区的位置。注意使用绝对路径。

下面的命令中，用`:command:`pwd``命令获得当前工作路径，借以得到绝对路径。

```
$ export GIT_DIR=`pwd:file:`/repo.git  
$ export GIT_WORK_TREE=`pwd:file:`/export
```

- 然后就可以进入`:file:`export``目录，执行Git操作了。

```
$ git status  
$ git checkout .
```

### 为什么没有历史备份？

当针对`repo.git`执行`:command:`git log``的时候，满心期望能够看到备份的历史，但是看到的却只有孤零零的几个备份记录。不要着急，可能是备份回滚了。

参见下节的备份回滚，会找到如何获取更多历史备份的方法。

## 备份回滚及设置

我在开发Gistore时，最麻烦的就是备份历史的管理。如果不对备份历史进行回滚，必然会导致提交越来越多，备份空间占用越来越大，直至磁盘空间占慢。

最早的想法是使用:command:`git rebase`。即将准备丢弃的早期备份历史合并成为一个提交，后面的提交变基到合并提交之上，这样就实现了对历史提交的丢弃。但是这样的操作即费时，又比较复杂。忽然又一天灵机一动，为什么不用分支来实现对回滚数据的保留？至于备份主线（master分支）从一个新提交开始重建。

回滚后master分支如何从一个新提交开始呢？较早的实现是直接重置到一个空提交（gistore/0）上，但是这样会导致接下来的备份非常耗时。一个更好的办法是使用:command:`git commit-tree`命令，直接从回滚前的master分支创建新提交。在读者看到这本书的时候，我应该已经才用了新的实现。

具体的实现过程是：

- 首先在备份库初始化的时候，就会建立一个空的提交，并打上里程碑 Tag: gistore/0（新的实现这个步骤变得没有必要）。
- 每次备份，都提交在Git库的主线master上。
- 当Git库的master主线的提交数达到规定的阈值（缺省200），对gistore分支进行回滚，并基于当前master打上分支：gistore/1。
  - 如果设置了5个回滚分支，并且存在其他回滚分支，则分支依次向后回滚。
  - 删除gistore/5，gistore/4分支改名为gistore/5，等等，最后将gistore/1重命名为gistore/2。
  - 基于当前master建立分支gistore/1。
  - 将当前master以最新提交的树创建一个不含历史的提交，并重置到该提交。即 master分支抛弃所有的备份历史。
  - 在新的master分支进行一次新的备份。
- 当回滚发生后，对备份库的远程数据同步不会有什么影响，传输的数据量也仅是新增备份和上一次备份的差异。

### 如何找回历史备份？

通过上面介绍的Gistore回滚的实现方法，会知道当回滚发生后，主线master只包含两个提交。一个是上一次备份的数据，另外一个是最新的数据备份。似乎大部分备份历史被完全丢弃了。其实，可以从分支gistore/1中看到最近备份的历史，还可以从其他分支（如果有的话）会看到更老的历史。

查看回滚分支的提交历史：

```
$ git --git-dir=repo.git log gistore/1
```

通过日志找出要恢复的时间点和提交号，使用:command:`git checkout`即可检出历史版

本。

## 注册备份任务别名

因为Gistore可以在任何目录下创建备份任务，管理员很难定位当前到底存在多少个备份库，因此需要提供一个机制，让管理员能够看到系统中有哪些备份库。还有，就是在使用Gistore时若使用长长的备份库路径作为参数会显得非常笨拙。任务别名就是用来解决这些问题的。

任务别名实际上就是在备份库在目录:`/etc/gistore/tasks`下创建的符号连接。

为备份任务创建任务别名非常简单，只需要在:`/etc/gistore/tasks`目录中创建的备份库的符号链接，该符号链接的名称，作为这些备份库的任务别名。

```
$ sudo ln -s /home/jiangxin/Desktop/mybackup /etc/gistore/tasks/jx  
$ sudo ln -s /backup/database /etc/gistore/tasks/db
```

于是，就创建了两个任务别名，在以后执行备份时，可以简化备份命令：

```
$ sudo gistore commit jx  
$ sudo gistore commit db
```

查看一份完整备份列表也非常简单，执行:`gistore list`命令即可。

```
$ gistore list  
db      : /backup/database  
jx      : /home/jiangxin/Desktop/mybackup
```

当`gistore list`命令后面指定某个任务列表时，相当于执行`gistore status`命令，查看备份状态信息：

```
$ gistore list db
```

可以用一条命令对所有的任务别名执行备份：

```
$ gistore commit-all
```

## 自动备份：crontab

在:`/etc/cron.d/`目录下创建一个文件，如:`/etc/cron.d/gistore`，包含如

下内容：

```
## gistore backup
0    4 * * *      root /usr/bin/gistore commit-all
```

这样每天凌晨4点，就会以root用户身份执行:command:`gistore commit-all`命令。

为了执行相应的备份计划，需要将备份库在:file:`/etc/gistore/tasks`目录下创建符号链接。

## Gistore双机备份

Gistore备份库的主体就是repo.git，即一个Git库。可以通过架设一个Git服务器，远程主机通过克隆该备份库实现双机备份甚至是异地备份。而且最酷的是，整个数据同步的过程是可视的、快速的和无痛的，感谢伟大而又神奇的Git。

最好使用公钥认证的基于SSH的Git服务器架设，因为一是可以实现无口令的数据同步，二是增加安全性，因为备份数据中可能包含敏感数据。

还有可以直接利用现成的:file:`/etc/gistore/tasks`目录作为版本库的根。当然还需要在架设的Git服务器上，使用一个地址变换的小巧门。Gitosis服务器软件的地址变换魔法正好可以帮助实现。参见第31章第31.5节“轻量级管理的Git服务”。

# 补丁中的二进制文件

有的时候，需要将对代码的改动以补丁文件的方式进行传递，最终合并入版本库。例如直接在软件部署目录内进行改动，再将改动传送到开发平台。或者是因为在某个开源软件的官方版本库中没有提交权限，需要将自己的改动以补丁文件的方式提供给官方。

关于补丁文件的格式，补丁的生成和应用在第3篇第20章“补丁文件交互”当中已经进行了介绍，使用的是`:command:`git format-patch``和`:command:`git am``命令，但这两个命令仅对Git库有效。如果没有使用Git对改动进行版本控制，而仅仅是两个目录：一个改动前的目录和一个改动后的目录，大部分人会选择使用GNU的`:command:`diff``命令及`:command:`patch``命令实现补丁文件的生成和补丁的应用。

但是GNU的`:command:`diff``命令（包括很多版本控制系统，如SVN的`:command:`svn diff``命令）生成的差异输出有一个非常大的不足或者说漏洞。就是差异输出不支持二进制文件。如果生成了新的二进制文件（如图片），或者二进制文件发生了变化，在差异输出中无法体现，当这样的差异文件被导出，应用到代码树中，会发现二进制文件或二进制文件的改动丢失了！

Git突破了传统差异格式的限制，通过引入新的差异格式，实现了对二进制文件的支持。并且更为神奇的是，不必使用Git版本库对数据进行维护，可以直接对两个普通目录进行Git方式的差异比较和输出。

## Git版本库中二进制文件变更的支持

对Git工作区的修改进行差异比较（`:command:`git diff --binary``），可以输出二进制的补丁文件。包含二进制文件差异的补丁文件可以通过`:command:`git apply``命令应用到版本库中。可以通过下面的示例，看看Git的补丁文件是如何对二进制文件提供支持的。

- 首先建立一个空的Git版本库。

```
$ mkdir /tmp/test
$ cd /tmp/test
$ git init
initialized empty Git repository in /tmp/test/.git/

$ git ci --allow-empty -m initialized
[master (root-commit) 2ca650c] initialized
```

- 然后在工作区创建一个文本文件`:file:`readme.txt``，以及一个二进制文件`:file:`binary.data``。

二进制的数据读取自系统中的二进制文件:`:file:`/bin/ls``，当然可以用任何其他二进制文件代替。

```
$ echo hello > readme.txt
$ dd if=/bin/ls of=binary.data count=1 bs=32
记录了1+0 的读入
记录了1+0 的写出
32字节(32 B)已复制, 0.0001062 秒, 301 kB/秒
```

注：拷贝`:file:`/bin/ls``可执行文件（二进制）的前32个字节作为`:file:`binary.data``文件。

- 如果执行`:command:`git diff --cached``看到的是未扩展的差异格式。

```
$ git add .
$ git diff --cached
diff --git a/binary.data b/binary.data
new file mode 100644
index 0000000..dc2e37f
Binary files /dev/null and b/binary.data differ
diff --git a/readme.txt b/readme.txt
new file mode 100644
index 0000000..ce01362
--- /dev/null
+++ b/readme.txt
@@ -0,0 +1 @@
+hello
```

可以看到对于`:file:`binary.data``，此差异文件没有给出差异内容，而只是一行“`Binary files ... and ... differ`”。

- 再用`:command:`git diff --cached --binary``即增加了`--binary`参数试试。

```
$ git diff --cached --binary
diff --git a/binary.data b/binary.data
new file mode 100644
index 00000000000000000000000000000000..dc2e37f81e0fa88308bec48cd
GIT binary patch
literal 32
bcmb<-^>JfjWMqH=CI&kO5HCR00W1UnGBE;C

literal 0
HcmV?d00001

diff --git a/readme.txt b/readme.txt
new file mode 100644
```

```
index 0000000..ce01362  
--- /dev/null  
+++ b/readme.txt  
@@ -0,0 +1 @@  
+hello
```

看到了么，此差异文件给出了二进制文件:`:file:`binary.data``差异的内容，并且差异内容经过base85文本化了。

- 提交后，并用新的内容覆盖:`:file:`binary.data``文件。

```
$ git commit -m "new text file and binary file"  
[master 7ab2d01] new text file and binary file  
2 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 binary.data  
create mode 100644 readme.txt  
  
$ dd if=/bin/ls of=binary.data count=1 bs=64  
记录了1+0 的读入  
记录了1+0 的写出  
64字节(64 B)已复制, 0.00011264 秒, 568 kB/秒  
  
$ git commit -a -m "change binary.data."  
[master a79bcbe] change binary.data.  
1 files changed, 0 insertions(+), 0 deletions(-)
```

- 看看更改二进制文件的新差异格式。

```
$ git show HEAD --binary  
commit a79bcbe50c1d278db9c9db8e42d9bc5bc72bf031  
Author: Jiang Xin <jiangxin@osxp.com>  
Date:   Sun Oct 10 19:22:30 2010 +0800  
  
        change binary.data.  
  
diff --git a/binary.data b/binary.data  
index dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168  
GIT binary patch  
delta 37  
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;  
  
delta 4  
LcmZ=zn4kav0;B;E
```

- 更简单的，使用:`:command:`git format-patch``命令，直接将最近的两次提交导出为补

丁文件。

```
$ git format-patch HEAD^^  
0001-new-text-file-and-binary-file.patch  
0002-change-binary.data.patch
```

毫无疑问，这两个补丁文件都包含了对二进制文件的支持。

```
$ cat 0002-change-binary.data.patch  
From a79bcbe50c1d278db9c9db8e42d9bc5bc72bf031 Mon Sep 17 00:00:00 2001  
From: Jiang Xin <jiangxin@osxp.com>  
Date: Sun, 10 Oct 2010 19:22:30 +0800  
Subject: [PATCH 2/2] change binary.data.  
  
---  
binary.data | Bin 32 -> 64 bytes  
1 files changed, 0 insertions(+), 0 deletions(-)  
  
diff --git a/binary.data b/binary.data  
index dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168  
GIT binary patch  
delta 37  
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;  
  
delta 4  
LcmZ=zn4kav0;B;E  
  
--  
1.7.1
```

那么如何将补丁合并入代码树呢？

不能使用GNU:`:command:`patch``命令，因为前面曾经说过GNU的:`:command:`diff``和:`:command:`patch``不支持二进制文件的补丁。当然也不支持Git的新补丁格式。将Git格式的补丁应用到代码树，只能使用git命令，即:`:command:`git apply``命令。

接着前面的例子。首先将版本库重置到最近两次提交之前的状态，即丢弃最近的两次提交，然后将两个补丁都合并到代码树中。

- 重置版本库到两次提交之前。

```
$ git reset --hard HEAD^^  
HEAD is now at 2ca650c initialized
```

```
$ ls  
0001-new-text-file-and-binary-file.patch 0002-change-binary.data.patch
```

- 使用`:command:`git apply``应用补丁。

```
$ git apply 0001-new-text-file-and-binary-file.patch 0002-change-binary.d
```

- 可以看到64字节长度的`:file:`binary.data``又回来了。

```
$ ls -l  
总用量 16  
-rw-r--r-- 1 jiangxin jiangxin 754 10月 10 19:28 0001-new-text-file-and-b:  
-rw-r--r-- 1 jiangxin jiangxin 524 10月 10 19:28 0002-change-binary.data.|  
-rw-r--r-- 1 jiangxin jiangxin 64 10月 10 19:34 binary.data  
-rw-r--r-- 1 jiangxin jiangxin 6 10月 10 19:34 readme.txt
```

- 最后不要忘了提交。

```
$ git add readme.txt binary.data  
$ git commit -m "new text file and binary file from patch files."  
[master 7c1389f] new text file and binary file from patch files.  
 2 files changed, 1 insertions(+), 0 deletions(-)  
  create mode 100644 binary.data  
  create mode 100644 readme.txt
```

Git对补丁文件的扩展，实际上不只是增加了二进制文件的支持，还提供了对文件重命名（`rename from`和`rename to`指令），文件拷贝（`copy from`和`copy to`指令），文件删除（`deleted file`指令）以及文件权限（`new file mode`和`new mode`指令）的支持。

## 对非Git版本库中二进制文件变更的支持

不在Git版本库中的文件和目录可以比较生成Git格式的补丁文件么，以及可以执行应用补丁（`apply patch`）的操作么？

是的，Git的`diff`命令和`apply`命令支持对非Git版本库/工作区进行操作。但是当前Git最新版本(1.7.3)的`:command:`git apply``命令有一个bug，这个bug导致目前的`:command:`git apply``命令只能应用patch level（补丁文件前缀级别）为1的补丁。我已经将改正这个Bug的补丁文件提交到Git开发列表中，但有其他人先于我修正了这个Bug。不管最终是谁修正的，在新版本的Git中，这个问题应该已经解决。参见我发给Git邮件列表的相关讨论。

- <http://marc.info/?l=git&m=129058163119515&w=2>

下面的示例演示一下如何对非Git版本库使用`:command:`git diff`` 和`:command:`git patch`` 命令。首先准备两个目录，一个为`hello-1.0`目录，在其中创建一个文本文件以及一个二进制文件。

```
$ mkdir hello-1.0
$ echo hello > hello-1.0/readme.txt
$ dd if=/bin/ls of=hello-1.0/binary.dat count=1 bs=32
记录了1+0 的读入
记录了1+0 的写出
32字节(32 B)已复制, 0.0001026 秒, 312 kB/秒
```

另外一个`:file:`hello-2.0`` 目录，其中的文本文件和二进制文件都有所更改。

```
$ mkdir hello-2.0
$ printf "hello\nworld\n" > hello-2.0/readme.txt
$ dd if=/bin/ls of=hello-2.0/binary.dat count=1 bs=64
记录了1+0 的读入
记录了1+0 的写出
64字节(64 B)已复制, 0.0001022 秒, 626 kB/秒
```

然后执行`:command:`git diff`` 命令。命令中的`--no-index`参数对于不在版本库中的目录/文件进行比较时可以省略。其中还用了`--no-prefix`参数，这样就可以生成前缀级别(patch level) 为1的补丁文件。

```
$ git diff --no-index --binary --no-prefix \
    hello-1.0 hello-2.0 > patch.txt
$ cat patch.txt
diff --git hello-1.0/binary.dat hello-2.0/binary.dat
index dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168cb71
GIT binary patch
delta 37
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;

delta 4
LcmZ=zn4kav0;B;E

diff --git hello-1.0/readme.txt hello-2.0/readme.txt
index ce01362..94954ab 100644
--- hello-1.0/readme.txt
+++ hello-2.0/readme.txt
@@ -1 +1,2 @@
    hello
+world
```



进入到:`:file:`hello-1.0``目录，执行:`:command:`git apply``应用补丁，即使:`:file:`hello-1.0``不是一个Git库。

```
$ cd hello-1.0  
$ git apply ./patch.txt
```

会惊喜的发现:`:file:`hello-1.0``应用补丁后，已经变得和:`:file:`hello-2.0``一样了。

```
$ git diff --stat . ./.hello-2.0
```

命令:`:command:`git apply``也支持反向应用补丁。反向应用补丁后，`:file:`hello-1.0``中文件被还原，和:`:file:`hello-2.0``比较又可以看到差异了。

```
$ git apply -R ./patch.txt  
$ git diff --stat . ./.hello-2.0  
{. => ./.hello-2.0}/binary.dat | Bin 32 -> 64 bytes  
{. => ./.hello-2.0}/readme.txt | 1 +  
2 files changed, 1 insertions(+), 0 deletions(-)
```

## 其他工具对Git扩展补丁文件的支持

Git对二进制提供支持的扩展的补丁文件格式，已经成为补丁文件格式的新标准被其他一些应用软件所接受。例如Mercurial/Hg就提供了对Git扩展补丁格式的支持。

为:`:command:`hg diff``命令增加`--git`参数，实现Git扩展diff格式输出。

```
$ hg diff --git
```

Hg的MQ插件提供对Git补丁的支持。

```
$ cat .hg/patches/1.diff  
# HG changeset patch  
# User Jiang Xin <worldhello.net AT gmail DOT com>  
# Date 1286711219 -28800  
# Node ID ba66b7bca4baec41a7d29c5cae6bea6d868e2c4b  
# Parent 0b44094c755e181446c65c16a8b602034e65efd7  
new data  
  
diff --git a/binary.data b/binary.data
```

```
new file mode 100644
index 00000000000000000000000000000000..dc2e37f81e0fa88308bec48cd5195
GIT binary patch
literal 32
bc$}+u^>JfjWMqH=CI&kO5HCR00n7&gGBE;C
```



103 lines (72 sloc) | 5.67 KB

## 云存储

通过云存储，将个人数据备份在网络上是非常吸引人的服务，比较著名的公司或产品有dropbox、surgarsync、Live Mesh、Syncplicity等。这些产品的特点是能够和操作系统的shell整合，例如和Windows的资源管理器或者Linux上的nautilus，当本地有数据改动会自动的同步到远程的“云存储”上。用户可以在多个计算机或者手持设备上配置和同一个“云端”的帐号同步，从而实现在多个计算机或者多个手持设备上的数据同步。

## 现有云存储的问题

遗憾的是我并未使用过上述云存储服务，主要是支持Linux操作系统的云存储客户端比较少，或者即使有也因为网络的局限而无法访问。但是通过相关文档，还是可以了解到其实现的机理。

- 仅支持对部分历史数据的备份。

dropbox支持30天数据备份，surgarsync每个文件仅保留5个备份（付费用户），对于免费用户仅2个备份。

- 数据同步对网络带宽依赖比较高。
- “云端”被多个设备共享，冲突解决比较困难。

Surgarsync会将冲突的文件自动保存为多份，造成磁盘空间超出配额。其他有的产品在遇到冲突时停止同步，让用户决定选择哪个版本。

在Git书里介绍云存储，是因为上述云存储实现和Git有关么？不是。实际上通过上面各个云存储软件特性的介绍，有经验的Linux用户会感觉这些产品在数据同步时和Linux下的rsync、unison等数据同步工具非常类似，也许只是在服务器端增加了历史备份而已。

已经有用户尝试将云存储和Git结合使用，就是将Git库本身放在本机的云存储同步区（例如dropbox的Dropbox目录下），Git库被同步至云端。即用云存储作为二传手，实际上还是基于本地协议操作Git。这样实际会有问题的。

- 如果两台机器各自进行了提交，云存储同步一定会引发冲突，这种冲突是难以解决的。
- 云端对Git的每个文件都进行备份，包括执行:command:`git gc`命令

打包后丢弃掉的松散对象。这实际对于Git是不需要的，会浪费本来就有  
限的空间配额。

- 因为版本库周期性的执行: command: `git gc --auto` 会导致即使Git版本  
库的一个小提交也可能会触发大量的云存储数据传输。

## Git式云存储畅想

GitHub是Git风格的云存储，但缺乏像前提到的云存储提供的傻瓜式服务，只  
有Git用户才能真正利用好，这大大限制了Git在云存储领域的推广。下面是我  
的一个预言：一个结合了Git和傻瓜式云存储的网络存储服务终将诞生。新  
的傻瓜式云存储有下列特征：

### 差异同步传输

对用户体验最为关键的是网络传输，如果用Git可以在同步时实现仅对文件差  
异进行数据传输，会大大提高同步效率。之所以现有的在线备份系统实现不  
了“差异同步传输”，是因为没有在本地对上一次同步时的数据做备份，只  
能通过时间戳或者文件的哈希值判断文件是否改变，而无法得出文件修改前  
后的差异。

可以很容易的测试云存储软件的网络传输性能。准备一个大的压缩包（使同  
步时压缩传输可以忽略），测试一下同步时间。再在该文件后面追加几个字  
节，然后检查同步时间。比较前后两个时间，就可以看出同步是否实现了仅  
对差异的同步传输。

### 可预测的本地及云端存储空间的占用

要想实现前面提到的差异同步传输，就必须在本地保存上一次同步时文件的  
备份。Subversion是用一份冗余的本地拷贝实现的，这样本地存储是实际文  
件的两倍。Git在本地是完全版本库，占用空间逐渐增加变得不可预测。

使用Git实现云存储，就要解决在本地以及在服务器端空间占用不可预测的问  
题。对于服务器端，可以采用前面介绍的Gistore软件采用的重整版本库的方  
法，或者通过基于历史版本重建提交然后变基来实现提交数量的删减。对于  
客户端来说，只保留一个提交就够了，类似Subversion的文件原始拷贝，这  
就需要在客户端基于Git原理的重新实现。

### 更高效的云端存储效率

现有的云存储效率不高，很有可能因为冗余备份而导致存储超过配额，即使  
服务提供商的配额计算是以最后一个版本计算的，实际的磁盘占用还是很可  
观。

Git底层实现了一个对内容跟踪的文件系统，相同内容的文件即使文件名和目录不同，在Git看来都是一个对象并用一个文件存储（文件名是内容相关的SHA1哈希值）。因此Git方式实现的云存储在空间的节省上有先天的优势。

### 自动的冲突解决

冲突解决是和文件同步相关的，只有通过“差异同步传输”解决了同步的性能瓶颈，才能为冲突解决打下基础。先将冲突的各个版本都同步到本地，然后进行自动冲突解决，如果冲突无法自动解决，再提示用户手工解决冲突。还有，如果在手工冲突解决是引入类似kdiff3一样的工具，对用户更有吸引力。

### Git提交中引入特殊标识

如果使用变基或其他技术实现备份提交数量的删减，就会在云端的提交和本地数据合并上产生问题。可以通过为提交引入特殊的唯一性标识，不随着Git变基而改变，就像在Gerrit中的Change-Id标签一样。

我相信，基于Git的文件系统以及传输机理可以实现一个更好用的云存储服务。

# 跨平台操作Git

读者是在什么平台（操作系统）中使用Git呢？图40-1是网上一个Git调查结果的截图，从中可以看出排在前三位的是：Linux、Mac OS X和Windows。而Windows用户中又以使用msysGit的用户居多。

On which operating system(s) do you use Git? (Choice - Multiple answers)

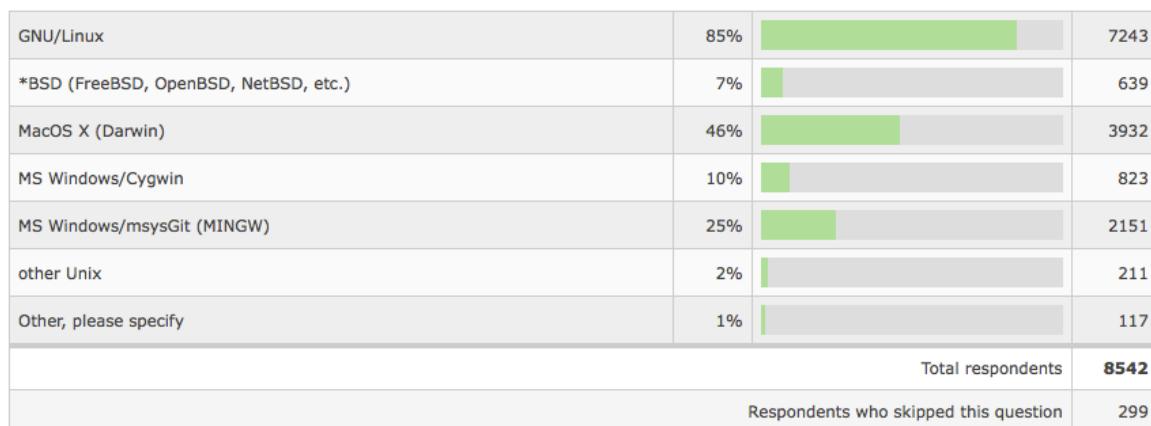


图40-1：Git用户操作系统使用分布图（摘

自：<http://www.survs.com/results/33Q00ZZE/MV653KSPI2>）

在如今手持设备争夺激烈的年代，进行软件开发工作在什么操作系统上已经变得不那么重要了，很多手持设备都提供可以运行在各种主流操作系统的虚拟器，因此一个项目团队的成员根据各自习惯，可能使用不同的操作系统。当一个团队中不同成员在不同平台中使用Git进行交互时，可能会遇到平台兼容性问题。

即使团队成员都在同一种操作系统上工作（如Windows），但是Git服务器可能架设在另外的平台上（如Linux），也同样会遇到平台兼容性问题。

# 字符集问题

在本书第1篇“第3章安装Git”中，就已经详细介绍了不同平台对本地字符集（如中文）的支持情况，本章再做以简单的概述。

Linux、Mac OS X以及Windows下的Cygwin缺省使用UTF-8字符集。Git运行在这些平台上，能够使用本地语言（如中文）写提交说明、命名文件，甚至使用本地语言命名分支和里程碑。在这些平台上唯一要做的就是对Git进行如下设置，以便使用了本地语言（如中文）命名文件时，能够在状态查看、差异比较时，正确显示文件名。

```
$ git config --global core.quotepath false
```

但是如果在Windows平台使用msysGit或者其他平台使用了非UTF-8字符集，要想使用本地语言撰写提交说明、命名文件名和目录名就非常具有挑战性了。例如对于使用GBK字符集的中文Windows，需要为Git进行如下设置，以便能够在提交说明中正确使用中文。

```
$ git config --system core.quotepath false
$ git config --system i18n.commitEncoding gbk
$ git config --system i18n.logOutputEncoding gbk
```

当像上面那样对i18n.commitEncoding进行设置后，如果执行提交，就会在提交对象中嵌入编码设置的指令。例如在Windows中使用msysGit执行一次提交，在Linux上使用`:command:`git cat-file``命令查看提交时会出现乱码，需要使用`:command:`iconv``命令对输出进行字符集转换才能正确查看该提交对象。下面输出的倒数第三行可以看到encoding gbk这条对字符集设置的指令。

```
$ git cat-file -p HEAD | iconv -f gbk -t utf-8
tree 00e814cda96ac016bcacabcf4c8a84156e304ac6
parent 52e6454db3d99c85b1b5a00ef987f8fc6d28c020
author Jiang Xin <jiangxin@osssxp.com> 1297241081 +0800
committer Jiang Xin <jiangxin@osssxp.com> 1297241081 +0800
encoding gbk
```

添加中文说明。

因为在提交对象中声明了正确的字符集，因此Linux下可以用`:command:`git log``命令正确显示msysGit生成的包含中文的提交说明。

但是对于非UTF-8字符集平台（如msysGit）下，使用本地字符（如中文）命名文件或目录，Git当前版本（1.7.4）的支持尚不完善。文件名和目录名实际上是写在树对象中

的，Git没能在创建树对象时，将本地字符转换为UTF-8字符进行保存，因而在跨平台时造成文件名乱码。例如下面的示例显示的是在Linux平台（UTF-8字符集）下查看由msysGit提交的包含中文文件的树对象。注意要在`:command:`git cat-file``命令的后面通过管道符号调用`:command:`iconv``命令进行字符转换，否则不能正确地显示中文。如果直接在Linux平台检出，检出的文件名显示为乱码。

```
$ git cat-file -p HEAD^{tree} | iconv -f gbk -t utf-8  
100644 blob 8c0b112f56b3b9897007031ea38c130b0b161d5a    说明.txt
```

# 文件名大小写问题

Linux、Solaris、BSD及其他类Unix操作系统使用的是大小写敏感的文件系统，而Windows和Mac OS X（默认安装）的文件系统则是大小写不敏感的文件系统。即用文件名:`:file:`README``、`:file:`readme``以及`:file:`Readme``（混合大小写）进行访问，在Linux等操作系统上访问的是不同的文件，而在Windows和Mac OS X上则指向同一个文件。换句话说，两个不同文件`:file:`README``和`:file:`readme``在Linux等操作系统上可以共存，而在Windows和Mac OS X上，这两个文件只能同时存在一个，另一个会被覆盖，因为在大小写不敏感的操作系统看来，这两个文件是同一个文件。

如果在Linux上为Git版本库添加了两个文件名仅大小写不同的文件（文件内容各不相同），如`:file:`README``和`:file:`readme``。当推送到服务器的共享版本库上，并在文件名大小写不敏感的操作系统中克隆或同步时，就会出现问题。例如在Windows和Mac OS X平台上执行`:command:`git clone``后，在本地工作区中出现两个同名文件中的一个，而另外一个文件被覆盖，这就会导致Git对工作区中文件造成误判，在提交时会导致文件内容被破坏。

当一个项目存在跨平台开发的情况时，为了避免这类问题的发生，在一个文件名大小写敏感的操作系统（如Linux）中克隆版本库后，应立即对版本库进行如下设置，让版本库的行为好似对文件大小写不敏感。

```
$ git config core.ignorecase true
```

Windows和Mac OS X在初始化版本库或者克隆一个版本库时，会自动在版本库中包含配置变量`core.ignorecase`为`true`的设置，除非版本库不是通过克隆而是直接从Linux上拷贝而来。

当版本库包含了`core.ignorecase`为`true`的配置后，文件名在添加时便被唯一确定。如果之后修改文件内容及修改文件名中字母的大小写，再提交亦不会改变文件名。如果对添加文件时设置的文件名的大小写不满意，需要对文件重命名，对于Linux来说非常简单。例如执行下面的命令就可以将`:file:`changelog``文件的文件名修改为`:file:`ChangeLog``。

```
$ git mv changelog ChangeLog  
$ git commit
```

但是对于Windows和Mac OS X，却不能这么操作，因为会拒绝这样的重命名操作：

```
$ git mv changelog ChangeLog  
fatal: destination exists, source=changelog, destination=ChangeLog
```

而需要像下面这样，先将文件重命名为另外的一个名称，再执行一次重命名改回正确的文件名，如下：

```
$ git mv changelog non-exist-filename  
$ git mv non-exist-filename ChangeLog  
$ git commit
```

# 换行符问题

每一个通用的版本控制系统，无论是CVS、Subversion、Git或是其他，都要面对换行符转换的问题。这是因为作为通用的版本控制系统要面对来自不同操作系统的文件，而不同的操作系统在处理文本文件时，可能使用不同的换行符。

## 不同的操作系统可能使用不同的换行符

文本文件的每一行结尾用一个或者两个特殊的ASCII字符进行标识，这个标识就是换行符。主要的换行符有三种：LF（Line Feed即换行，C语言等用“\\n”表示，相当于十六进制的`0x0A`），CR（Carriage Return即回车，C语言等用“\\r”表示，相当于十六进制的`0x0D`）和CRLF（即由两个字符“CR+LF”组成，即“\\r\\n”，相当于十六进制的`0x0D 0x0A`），分别用在不同的操作系统中。（以下内容摘

自<http://en.wikipedia.org/wiki/Newline>。）

- LF换行符：用于Multics、Unix、类Unix（如GNU/Linux、AIX、Xenix、Mac OS X、FreeBSD等）、BeOS、Amiga、RISC OS等操作系统中。
- CRLF换行符：用于DEC TOPS-10、RT-11和其他早期的非Unix，以及CP/M、MP/M、DOS（MS-DOS、PC-DOS等）、Atari TOS、OS/2、Microsoft Windows、Symbian OS、Palm OS等系统中。
- CR换行符：用于Commodore 8位机、TRS-80、苹果II家族、Mac OS 9及更早版本。

实际上，自从苹果的Mac OS从第10版转向Unix内核开始，依据不同的文本文件换行符，主流的操作系统可以划分为两大阵营，一个是微软Windows作为一方，使用CRLF作为换行符，另外一方包括Unix、类Unix（如Linux和Mac OS X等）使用LF作为换行符。分属不同阵营的操作系统之间交换文本文件会因为换行符的不同造成障碍，而对于使用版本控制系统，也同样会遇到换行符的麻烦。

- 编辑器不能识别换行符，可能会显示为特殊字符，如Linux上的编辑器显示的^M特殊字符，就是拜Windows的CRLF换行符所赐。或者丢弃换行符，如来自Linux的文本文件，在Windows上打开可能会因为识别不了换行符，导致所有的行合并。
- 版本库中的文件被来自不同操作系统的用户改来改去，在某一次提交中换行符为LF，在下一次提交中被替换为CRLF，这不但会在查看文件版本间差异时造成困惑（所有的行都显示为变更），还给版本库的存储带来不必要的冗余。
- 可能会在一个文件中引入混杂的换行符，即有的行是LF，而有的行是CRLF。无论在那个操作系统用编辑器打开这样的文件，都会或多或少感到困惑。
- 如果版本控制系统提供文本文件换行符的自动转换，在Windows平台进行版本库文件导出为源码包并发布，当该源码包被Linux用户下载，编译、运行可能会有问题，反之亦然。

### 文本文件和二进制文件的判别，是换行符转换的基础

几乎所有的版本库控制系统都采用这样的解决方案：对于文本文件，在版本库中保存时换行符使用LF，当从版本库检出到工作区时，则根据平台的不同或者用户的设置的不同，对文本文件的换行符进行转换（转换为LF、CR或CRLF）。

为什么换行符转换要特意强调文本文件呢？这是因为如果对二进制文件（程序或者数据）当中出现的换行符进行上述转换，会导致二进制文件被破坏。因此判别文件类型是文本文件还是二进制文件，是正确进行文件换行符转换的基础。

有的版本控制系统，如CVS，必须在添加文件时人为的设定文件类型（用-kb参数设定二进制文件），一旦用户忘记对二进制文件进行标记，就会造成二进制文件被破坏。这种破坏有时藏的比较深，例如在Linux上检出文件一切正常，因为版本库中被误判为文本文件的图形文件中所包含字符0x0A在Linux上检出没有改变，但是在Windows上检出会导致图形文件中的0x0A字符被转换为0x0D 0x0A两个字符，造成图片被破坏。

有的版本控制系统可以自动识别文本文件和二进制文件，但是识别算法存在问题。例如Subversion检查文件的前1024字节的内容，如果其中包含NULL字符（0x00），或者超过15%是非ASCII字符，则Subversion认定此文件为二进制文件（参见Subversion源代码：`file: `subversion/libsvn_subr/io.c`` 中的`svn_io_detect_mimetype2`函数）。这种算法会将包含大量中文的文本文件当作二进制文件，不进行换行符转换，也不能进行版本间的比较（除非强制执行）。

Git显然比Subversion更了解这个世界上文字的多样性，因此在判别二进制文件上没有多余的判别步骤，只对blob对象的前8000个字符进行检查，如果其中出现NULL字符（0x00）则当作二进制文件，否则为文本文件（参见Git源代码：`file: `xdiff-interface.c`` 中的`buffer_is_binary`函数）。Git还允许用户通过属性文件对文件类型进行设置，属性文件设置优先。

Git缺省并不开启文本文件的换行符转换，因为毕竟Git对文件是否是二进制文件所做的猜测存在误判的可能。如果用户通过属性文件或者其他方式显式的对文件类型进行了设置，则Git就会对文本文件开启换行符转换。

下面是一个属性文件的示例，为方便描述标以行号。

```
1 *.txt          text
2 *.vcproj       eol=crlf
3 *.sh           eol=lf
4 *.jpg          -text
5 *.jpeg         binary
```

包含了上面属性文件的版本库，会将.txt、.vcproj、.sh为扩展名的文件视为文本文件，在处理过程中会进行换行符转换，而将.jpg、.jpeg为扩展名的文件视为二进制文件，不进

行换行符转换。

### 依据属性文件进行换行符转换

关于属性文件，会在后面的章节详细介绍，现在可以将其理解为工作区目录下的`:file:`.gitattributes``文件，其文件匹配方法及该文件的作用范围和`:file:`.gitignore``文件非常类似。

像上面的属性文件示例中，第1行设置了扩展名为.txt的文件具有text属性，则所有扩展名为.txt的文件添加到版本库时，在版本库中创建的blob文件的换行符一律转换为LF。而当扩展名为.txt的文件检出到工作区时，则根据平台的不同使用不同的换行符，如在Linux上检出使用LF换行符，在Windows上检出使用CRLF换行符。

示例中的第2行设置扩展名为.vcproj的文件的属性eol的值为crlf，隐含着该文件属于文本文件的含义，当向版本库添加扩展名为.vcproj文件时，在版本库中创建的blob文件的换行符一律转换为LF。而当该类型的文件检出到工作区时，则一律使用CRLF作为换行符，不管是在Windows上检出，还是在Linux上检出。

同理示例中的第3行设置的扩展名为.sh的文件也会进行类似的换行符转换，区别在于该类型文件无论在哪个平台检出，都使用LF作为换行符。

向上面那样逐一为不同类型的文件设置换行符格式显得很麻烦，可以在属性文件中添加下面的设置，为所有文件开启自动文件类型判断。

```
* text=auto
```

当为所有文件设置了`text=auto`的属性后，Git就会在文件检入和检出时对文件是否是二进制进行判断，采用前面提到的方法：如果文件头部8000个字符中出现NULL字符则为二进制文件，否则为文本文件。如果判断文件是文本文件就会启用换行符转换。至于本地检出文件采用什么换行符格式，实际上是由`core.eol`配置变量进行设置的，不过因为`core.eol`没有设置时采用缺省值`native`，才使得工作区文本文件的检出采用操作系统默认的换行符格式。配置变量`core.eol`除了默认的`native`外，还可以使用`lf`和`crlf`，不过一般较少用到。

### 使用Git配置变量控制换行符转换

在Git 1.7.4之前，用属性文件的方式来设置文件的换行符转换，只能逐一为版本库进行设置，如果要为本地所有的版本库设定文件换行符转换就非常的麻烦。Git 1.7.4提供了全局可用的属性文件，实现了对换行符转换设定的全局控制，我们会在后面的章节加以介绍。现在介绍另外一个方法，即通过配置变量`core.autocrlf`来开启文本文件换行符转换的功能。例如执行下面的命令，对配置变量`core.autocrlf`进行设置：

```
$ git config --global core.autocrlf true
```

默认Git不对配置变量core.autocrlf进行设置，因此在也没有通过属性文件指定文件类型的情况下，Git不对文件进行换行符转换。但是将配置变量core.autocrlf设置为下列值时，会开启Git对文件类型的智能判别并对文本文件执行换行符转换。

- 设置配置变量core.autocrlf为true。

效果就相当于为版本库中所有文件设置了text=auto的属性。即通过Git对文件类型的自动判定，对文本文件进行换行符转换。在版本库的blob文件中使用LF作为换行符，而检出到工作区时无论是什么操作系统都使用CRLF为换行符。注意当设置了core.autocrlf为true时，会忽略core.eol的设置，工作区文件始终使用CRLF作为换行符，这对于Windows下的Git非常适合，但不适用于Linux等操作系统。

- 设置配置变量core.autocrlf为input。

同样开启文本文件的换行符转换，但只是在文件提交到版本库时，将新增入库的blob文件的换行符转换为LF。当从版本库检出文件到工作区，则不进行文件转换，即版本库中文件若是采用LF换行符，检出仍旧是LF作为换行符。这个设置对Linux等操作系统下的Git非常适合，但不适合于Windows。

### 配制``core.safecrlf``捕捉异常的换行符转换

无论是用户通过属性文件设定文件的类型，还是通过Git智能判别，都可能错误的将二进制文件识别为文本文件，在转换过程中造成文件的破坏。有一种情况下破坏最为严重，就是误判的文件中包含不一致的换行符（既有CRLF，又有LF），这就会导致保存到版本库中的blob对象无论通过何种转换方式都不能还原回原有的文件。

Git提供了名为core.safecrlf的配置变量，可以用于捕捉这种不可逆的换行符转换，提醒用户注意。将配置变量core.safecrlf设置为true时，如果发现存在不可逆换行符转换时，会报错退出，拒绝执行不可逆的换行符转换。如果将配置变量core.safecrlf设置为warn则允许不可逆的转换，但会发出警告。

# 属性

Git通过属性文件为版本库中的文件或目录添加属性。设置了属性的文件或目录，例如之前介绍换行符转换时设置了文本属性（text）的文件，在执行Git相关操作时会做特殊处理。

## 属性定义

属性文件是一个普通的文本文件，每一行对一个路径（可使用通配符）设置相应的属性。  
语法格式如下：

```
<pattern> <attr1> <attr2> ...
```

其中路径由可以使用通配符的<pattern>定义，属性可以设置一个或多个，不同的属性之间用空格分开。路径中通配符的用法和文件忽略（file:.gitignore）的语法格式相同，参见本书第2篇“第10.8节文件忽略”相关内容。下面以text属性为例，介绍属性的不同写法：

- text

直接通过属性名进行设置，相当于设置text属性的值为true。

对于设置了text属性的文件，不再需要Git对文件类型进行猜测，而直接判定为文本文件并进行相应的换行符转换。

- -text

在属性名前用减号标识，相当于设置text属性值为false。

对于设置了取反text属性的文件，直接判定为二进制文件，在文件检入和检出时不进行换行符转换。

- !text

在属性名前面添加感叹号，相当于该属性没有设置，即不等于true，也不等于false。

对于未定义text属性的文件，根据Git是否配置了core.autocrlf配置变量，决定是否进行换行符转换。因此对于text属性没有定义和进行取反text属性设置，两者存在差异。

- text=auto

属性除了上述true、false、未设置三个状态外，还可以对属性用相关的枚举值（预定

义的字符串) 进行设置。不同的属性值可能有不同的枚举值, 对于text属性可以设置为auto。

对于text属性设置为auto的文件, 文件类型实际上尚未确定, 需要Git读取文件内容进行智能判别, 判别为文本文件则进行换行符转换。显然当设置text属性为auto时, 并不等同于true。

## 属性文件及优先级

属性文件可以以:`:file:`.gitattributes``文件名保存在工作区目录中, 提交到版本库后就可以和其他用户共享项目文件的属性设置。属性文件也可以保存在工作区之外, 例如保存在文件:`:file:`.git/info/attributes``中, 仅对本版本库生效, 若保存在:`:file:`/etc/gitattributes``文件中则全局生效。在查询某个工作区某一文件的属性时, 在不同位置的属性文件具有不同的优先级, Git依据下列顺序依次访问属性文件。

- 文件:`:file:`.git/info/attributes``具有最高的优先级。
- 接下来检查工作区同一目录下的:`:file:`.gitattributes``, 并依次向上递归查找:`:file:`.gitattributes``文件, 直到工作区的根目录。
- 然后查询由Git的配置变量`core.attributesfile`指定的全局属性文件。
- 最后是系统属性文件, 即文件:`:file:`$(prefix)/etc/gitattributes``。不同的Git安装方式这个文件的位置可能不同, 但是该文件始终和Git的系统配置文件(可以通过:`command: `git config --system -e``命令打开进而知道位置)位于同一目录中。

注意只有在1.7.4或更新版本的Git才提供后两种(全局和系统级的)属性文件。可以通过下面的例子来理解属性文件的优先级和属性设置方法。

首先来看看某个版本库即系统中所包含的属性文件:

- 其一是位于版本库中的文件:`:file:`.git/info/attributes``, 内容如下:

```
a*      foo !bar -baz
```

- 其二是位于工作区子目录:`:file:`t``下的属性文件, 即:`:file:`t/.gitattributes``, 内容如下:

```
ab*      merge=filfre
abc      -foo -bar
*.c      frotz
```

- 再一个是位于工作区根目录下的属性文件:`:file:`.gitattributes``, 内容如下:

```
abc      foo bar baz
```

- 系统文件:`:file:`/etc/gitconfig``中包含如下配置，则每个用户主目录下的`:file:`.gitattributes``文件做为全局属性文件。

```
[core]
attributesfile = ~/.gitattributes
```

- 位于用户主目录下的属性文件，即文件`:file:`~/.gitattributes``的内容如下：

```
* text=auto
```

当查询工作区文件`:file:`t/abc``的属性时，根据属性文件的优先级，按照下列顺序进行检索：

- 先检查属性文件`:file:`.git/info/attributes``。显然该文件中唯一的一行就和文件`:file:`t/abc``匹配，因此文件`:file:`t/abc``的属性如下：

```
foo    : true
bar    : 未设置
baz    : false
```

- 再检查和文件`:file:`t/abc``同目录的属性文件`:file:`t/.gitattributes``。该属性文件的前两行和路径`:file:`t/abc``相匹配，但是因为第二行设置`foo`和`bar`属性已经由属性文件`:file:`.git/info/attributes``提供，因此第二行的设置不起作用。经过这一步，文件`:file:`t/abc``获得的属性为：

```
foo    : true
bar    : 未设置
baz    : false
merge : filfre
```

- 然后沿工作区当前目录向上遍历属性文件，找到工作区根目录下的属性文件`:file:`.gitattributes``，进行检查。因为该属性文件设置的属性已经由前面的属性文件提供，所以文件`:file:`t/abc``的属性和上面第2步的结果一样。
- 因为设置了`core.attributesfile`为`:file:`~/.gitattributes``文件，因此接下来查找用户主目录下文件即`:file:`.gitattributes``。该文件唯一的一行匹配所有文件，因此`:file:`t/abc``又被附加了新的属性值`text=auto`。最终文件`:file:`t/abc``的属性如下。

```
foo    : true
bar    : 未设置
baz    : false
merge  : filfre
text   : auto
```

## 常用属性介绍

### text

属性text用于显式的指定文件的类型：二进制（-text）、文本文件（text）或是开启文件类型的智能判别（text=auto）。对于文本文件，Git会对其进行换行符转换。本书第40章“40.3换行符问题”中已经详细介绍了属性text的用法，并且在本章“40.1.1 属性定义”的示例中对属性text的取值做了总结，在此不再赘述。

在“40.3换行符问题”一节，我们还知道可以通过在Git配置文件中设置core.autocrlf配置变量，来开启Git对文件类型的智能判别，并对文本文件开启换行符转换。那么Git的配置变量core.autocrlf和属性text有什么异同呢？

当设置了Git了配置变量core.autocrlf为true或者input后，相当于设置了属性text=auto。但是Git配置文件中的配置变量只能在本地进行设置并且只对本地版本库有效，不能通过共享版本库传递到其他用户的本地版本库中，因而core.autocrlf开启换行符转换不能跟其他用户共享，或者说不能将换行符转换策略设置为整个项目（版本库）的强制规范。属性文件则不同，可以被检入到版本库中并通过共享版本库传递给其他用户，因此可以通过在检入的:`:file:`.gitattributes``文件中设置text属性，或者干脆设置text=auto属性，强制同一项目的所有用户在提交文本文件时都要规范换行符。

建议所有存在跨平台开发可能的项目都在项目根目录中检入一个:`:file:`.gitattributes``文件，根据文件扩展名设置文件的text属性，或者设置即将介绍的eol属性。

### eol

属性eol用于设定文本文件的换行符格式。对于设置了eol属性的文件，如果没有设定text属性时，默认会设置text属性为true。属性eol的取值如下：

- eol=crlf

当文件检入版本库时，blob对象使用LF作为换行符。当检出到工作区时，使用CRLF作为换行符。

- eol=lf

当文件检入版本库时，blob对象使用LF作为换行符，检出的时候工作区文件也使用LF作为换行符。

除了通过属性设定换行符格式外，还可以在Git的配置文件通过core.eol配置变量来设定。两者的区别在于配置文件中的core.eol配置变量设置的换行符是一个默认值，没有通过eol属性指定换行符格式的文本文件会采用core.eol的设置。变量core.eol的值可以设定为lf、crlf和native。默认core.eol的取值为native，即采用操作系统标准的换行符格式。

下面的示例通过属性文件设置文件的换行符格式。

```
*.vcproj      eol=crlf
*.sh          eol=lf
```

扩展名为.vcproj的文件使用CRLF作为换行符，而扩展名为.sh的文件使用LF作为换行符。在版本库中检入类似的属性文件，会使得Git客户端无论在什么操作系统中都能够在工作区检出一致的换行符格式，这样无论是在Windows上还是在Linux上使用`:command:`git archive``命令将工作区文件打包，导出的文件都会保持正确的换行符格式。

## ident

属性ident开启文本文件中的关键字扩展，即关键字\$Id\$的自动扩展。当检出到工作区时，\$Id\$自动扩展为\$Id:，后面紧接着40位SHA1哈希值（相应blob对象的哈希值），然后以一个\$字符结尾。当文件检入时，要对内容中出现的以\$Id:开始，以\$结束的内容替换为\$Id\$再保存到blob对象中。

这个功能可以说是对CVS相应功能的模仿。自动扩展的内容使用的是blob的哈希值而非提交本身的哈希值，因此并无太大实际意义，不建议使用。如果希望在文本文件中扩展出提交者姓名、提交ID等更有实际意义的内容，可以参照后面介绍的属性export-subst。

## filter

属性filter为文件设置一个自定义转换过滤器，以便文件在检入版本库及检出到工作区时进行相应的转换。定义转换过滤器通过Git配置文件来完成，因此这个属性应该只在本地进行设置，而不要通过检入到版本库中的`:file:`.gitattributes``文件传递。

例如下面的属性文件设置了所有的C语言源文件在检入和检出的时候使用名为indent的代码格式化过滤器。

```
*.c      filter=indent
```

然后还要通过Git配置文件设定indent过滤器，示例如下：

```
[filter "indent"]
  clean = indent
  smudge = cat
```

定义过滤器只要设置两条命令，一条是名为clean的配置设定的命令，用于在文件检入时执行，另外一条是名为smudge的配置设定的命令，用于将文件检出到工作区时使用的命令。对于本例，在代码检入时执行`:command:`indent``命令对代码格式化后，再保存到版本库中。当检出到工作区执行`:command:`cat``命令，实际上相当于直接将blob对象复制到工作区。

## diff

和前面介绍的属性不同，属性diff不会对文件检入检出造成影响，而只是在查看文件历史变更时起作用。属性diff可以取值如下：

- diff

进行版本间比较时，以文本方式进行比较，即使文件看起来像是二进制文件（包含NULL字符），或者被设置为二进制文件（-text）。

- -diff

不以文本方式进行差异比较，而以二进制方式进行比较。因为默认查看版本间差异时只显示文本文件的差异不显示二进制文件差异，因此包含-diff属性设置的文件在差异比较时不显示内容上的差异。对于有些文本文件（如postscript文件）进行差异比较没有意义，可以对其设置-diff属性，避免在显示提交间差异时造成干扰。

- !diff

不设置diff属性，相当于在执行差异比较时要对文件内容进行智能判别，如果文件看起来像是文本文件，则显示文本格式的差异比较。

- diff=<driver>

设定一个外部的驱动用于文件的差异比较。例如对于Word文档的差异比较就可以通过这种方式进行配置。

Word文档属于二进制文件，默认不显示差异比较。在Linux上有一个名为antiword的应用软件可以将Word文档转换为文本文件显示，借助该软件就可以实现在Linux（包括Mac OS X）

上显示Word文件的版本间差异。

下面的Git配置就定义了一个名为antiword的适用于Word差异比较的驱动：

```
[diff "antiword"]
textconv=antiword
```

其中textconv属性用于设定一个文件转换命令行，这里设置为antiword，用于将Word文档转换为纯文本。

然后还需要设置属性，修改版本库下的`.git/info/attributes`文件就可以，新增属性设置如下：

```
*.doc diff=antiword
```

关于更多的差异比较外部驱动的设置，执行`git help --web attributes`参见相关的帮助。

## merge

属性`merge`用于为文件设置指定的合并策略，受影响的Git命令有：`:command:`git merge``、`:command:`git revert``和`:command:`git cherry-pick``等。属性`merge`可以取值如下：

- `merge`

使用内置的三向合并策略。

- `-merge`

将当前分支的文件版本设置为暂时的合并结果，并且声明合并发生了冲突，这实际上是二进制文件默认的合并方式。可以对文本文件设置该属性，使得在合并时的行为类似二进制文件。

- `!merge`

和定义了`merge`属性效果类似，使用内置的三向合并策略。然而当通过Git配置文件的`merge.default`配置变量设置了合并策略后，如果没有为文件设置`merge`属性，则使用`merge.default`设定的策略。

- `merge=<driver>`

使用指定的合并驱动执行三向文件合并。驱动可以是内置的三个驱动，也可以是用户通过Git配置文件自定义的驱动。

下面重点说一说通过枚举值来指定在合并时使用的内置驱动和自定义驱动。先来看看Git提供的三个内置驱动：

- merge=text

默认文本文件在进行三向合并时使用的驱动。会在合并后的文本文件中用特殊的标识<<<<<、=====和>>>>>来标记冲突的内容。

- merge=binary

默认二进制文件在进行三向合并时使用的驱动。会在工作区中保持当前分支中的版本不变，但是会通过在三个暂存区中进行冲突标识使得文件处于冲突状态。

- merge=union

在文本文件三向合并过程中，不使用冲突标志符标识冲突，而是将冲突双方的内容简单的罗列在文件中。用户应该对合并后的文件进行检查。请慎用此合并驱动。

用户还可以自定义驱动。例如Topgit就使用自定义合并驱动的方式来控制两个Topgit管理文件:`:file:`.topmsg``和`:file:`.topdeps``的合并行为。

Topgit会在版本库的配置文件`:file:`.git/info/config``中添加下面的设置定义一个名为`ours`的合并驱动。注意不要将此`ours`驱动和本书第3篇第16章“16.6 合并策略”一节中介绍的`ours`合并策略弄混淆。

```
[merge "ours"]
name = \"always keep ours\" merge driver
driver = touch %A
```

定义的合并驱动的名称由`merge.*.name`给出，合并时执行的命令则由配置`merge.*.driver`给出。本例中使用了命令`:command:`touch %A``，含义为对当前分支中的文件进行简单的触碰（更新文件时间戳），亦即合并冲突时采用本地版本，丢弃其他版本。

Topgit还会在版本库`:file:`.git/info/attributes``属性文件中包含下面的属性设置：

```
.topmsg merge=ours
.topdeps merge=ours
```

含义为对这两个Topgit管理文件，采用在Git配置文件中设定的`ours`合并驱动。Topgit之所以要这么实现是因为不同特性分支的管理文件之间并无关联，也不需要合并，在遇到冲突时只使用自己的版本即可。这对于Topgit要经常地执行变基和分支合并来说，设置这个策略可以简化管理，但是这个合并设置在特定情况下也存在不合理之处。例如两个用户工作

在同一分支上同时更改了:`:file:`.topmsg``文件以修改特性分支的描述，在合并时会覆盖对方的修改，这显然是不好的行为。但是权衡利弊，还是如此实现最好。

## whitespace

Git可以对文本文件中空白字符的使用是否规范做出检查，在文件差异比较时，将使用不当的空白字符用红色进行标记（开启`color.diff.whitespace`）。也可以在执行:`:command:`git apply``时通过参数`--whitespace=error`防止错误的空白字符应用到提交中。

Git默认开启对下面三类错误空白字符的检查。

- blank-at-eol

在行尾出现的空白字符（换行符之前）被视为误用。

- space-before-tab

在行首缩进中出现在TAB字符前面的空白字符视为误用。

- blank-at-eof

在文件末尾的空白行视为误用。

Git还支持对更多空白字符的误用做出检测，包括：

- indent-with-non-tab

用8个或者更多的空格进行缩进视为误用。

- tab-in-indent

在行首的缩进中使用TAB字符视为误用。显然这个设置和上面的`indent-with-non-tab`互斥。

- trailing-space

相当于同时启用`blank-at-eol`和`blank-at-eof`。

- cr-at-eol

将行尾的CR（回车）字符视为换行符的一部分。也就是说，在行尾前出现的CR字符不会引起`trailing-space`报错。

- tabwidth=<n>

设置一个TAB字符相当于几个空格，缺省为8个。

可以通过Git配置文件中的core.whitespace配置变量，设置开启更多的空白字符检查，将要开启的空白字符检查项用逗号分开即可。

如果希望对特定路径进行空白字符检查，则可以通过属性whitespace进行。属性whitespace可以有如下设置：

- whitespace

开启所有的空白字符误用检查。

- -whitespace

不对空白字符进行误用检查。

- !whitespace

使用core.whitespace配置变量的设置进行空白字符误用检查。

- whitespace=...

和core.whitespace的语法一样，用逗号分隔各个空白字符检查项。

## export-ignore

设置了该属性的文件和目录在执行:command:`git archive`时不予导出。

## export-subst

如果为文件设置了属性export-subst，则在使用:command:`git archive`导出项目文件时，会对相应文件内容中的占位符展开，然后再添加到归档中。注意如果在使用:command:`git archive`导出时使用树ID，而没有使用提交或者里程碑，则不会发生占位符展开。

占位符的格式为\$Format:PLACEHOLDERS\$，其中PLACEHOLDERS使用:command:`git log --pretty=format:`相同的参数（具体参见:command:`git help log`显示的帮助页）。例如：\$Format:%H\$将展开为提交的哈希值，\$Format:%an\$将展开为提交者姓名。

## delta

如果设置属性delta为false，则不对该路径指向的blob文件执行Delta压缩。

## encoding

设置文件所使用的字符集，以便使用GUI工具（如gitk和git-gui）能够正确显示文件内容。基于性能上的考虑，gitk默认不检查该属性，除非通过gitk的偏好设置启用“Support per-file encodings”。

如果没有为文件设置encoding属性，则使用git.encoding配置变量。

## binary

属性binary严格来说是一个宏，相当于-text -diff。即禁止换行符转换及禁止文本方式显示文件差异。

用户也可以自定义宏。自定义宏只能在工作区根目录中的:`:file:`.gitattributes``文件中添加，以内置的binary宏为例，相当于在属性文件中进行了如下的设置：

```
[attr]binary -diff -text
```

# 钩子和模板

## Git钩子

Git的钩子脚本位于版本库:`:file:`.git/hooks``目录下，当Git执行特定操作时会调用特定的钩子脚本。当版本库通过:`:command:`git init``或者:`:command:`git clone``创建时，会在`:file:`.git/hooks``目录下创建示例脚本，用户可以参照示例脚本的写法开发适合的钩子脚本。

钩子脚本要设置为可运行，并使用特定的名称。Git提供的示例脚本都带有`.sample`扩展名，是为了防止被意外运行。如果需要启用相应的钩子脚本，需要对其重命名（去掉`.sample`扩展名）。下面分别对可用的钩子脚本逐一介绍。

### applypatch-msg

该钩子脚本由:`:command:`git am``命令调用。在调用时向该脚本传递一个参数，即保存有提交说明的文件的文件名。如果该脚本运行失败（返回非零值），则`:command:`git am``命令在应用该补丁之前终止。

这个钩子脚本可以修改文件中保存的提交说明，以便对提交说明进行规范化以符合项目的标准（如果有的话）。如果提交说明不符合项目标准，脚本直接以非零值退出，拒绝提交。

Git提供的示例脚本:`:file:`applypatch-msg.sample``只是简单的调用:`:file:`commit-msg``钩子脚本（如果存在的话）。这样通过:`:command:`git am``命令应用补丁和执行:`:command:`git commit``一样都会执行:`:file:`commit-msg``脚本，因此如须定制，请更改`:file:`commit-msg``脚本。

### pre-applypatch

该钩子脚本由:`:command:`git am``命令调用。该脚本没有参数，在补丁应用后但尚未提交之前运行。如果该脚本运行失败（返回非零值），则已经应用补丁的工作区文件不会被提交。

这个脚本可以用于对应用补丁后的工作区进行测试，如果测试没有通过则拒绝提交。

Git提供的示例脚本:`:file:`pre-applypatch.sample``只是简单的调用:`:file:`pre-commit``钩子脚本（如果存在的话）。这样通过:`:command:`git am``命令应用补丁和执行:`:command:`git commit``一样都会执行:`:file:`pre-commit``脚本，因此如须定制，请更改`:file:`pre-commit``脚本。

## post-applypatch

该钩子脚本由:command:`git am`命令调用。该脚本没有参数，在补丁应用并且提交之后运行，因此该钩子脚本不会影响:command:`git am`的运行结果，可以用于发送通知。

## pre-commit

该钩子脚本由:command:`git commit`命令调用。可以向该脚本传递--no-verify参数，此外别无参数。该脚本在获取提交说明之前运行。如果该脚本运行失败（返回非零值），Git提交被终止。

该脚本主要用于对提交数据的检查，例如对文件名进行检查（是否使用了中文文件名），或者对文件内容进行检查（是否使用了不规范的空白字符）。

Git提供的示例脚本:file:`pre-commit.sample`禁止提交在路径中使用了非ASCII字符（如中文字符）的文件。如果确有使用的必要，可以在Git配置文件中设置配置变量hooks.allownonascii为true以允许在文件名中使用非ASCII字符。Git提供的该示例脚本也对不规范的空白字符进行检查，如果发现则终止提交。

Topgit为所管理的版本库设置了自己的:file:`pre-commit`脚本，检查工作的Topgit特性分支是否正确设置了两个Topgit管理文件:file:`.topdeps`和:file:`.topmsg`，以及定义的分支依赖是否存在着重复依赖和循环依赖等。

## prepare-commit-msg

该钩子脚本由:command:`git commit`命令调用，在默认的提交信息准备完成后但编辑器尚未启动之前运行。

该脚本有1到3个参数。第一个参数是包含提交说明的文件的文件名。第二个参数是提交说明的来源，可以是message（由-m或者-F参数提供），可以是template（如果使用了-t参数或由commit.template配置变量提供），或者是merge（如果提交是一个合并或存在:file:`.git/MERGE\_MSG`文件），或者是squash（如果存在:file:`.git/SQUASH\_MSG`文件），或者是commit并跟着一个提交SHA1哈希值（如果使用-c、-C或者--amend参数）。

如果该脚本运行失败（返回非零值），Git提交被终止。

该脚本用于对提交说明进行编辑，并且该脚本不会因为--no-verify参数被禁用。

Git提供的示例脚本:file:`prepare-commit-msg.sample`可以用于向提交说明中嵌入提交者签名，或者将来自merge的提交说明中的含有“Conflicts:”的行去掉。

## commit-msg

该钩子脚本由:command:`git commit`命令调用，可以通过传递--no-verify参数而禁用。该脚本有一个参数，即包含有提交说明的文件名。如果该脚本运行失败（返回非零值），Git提交被终止。

该脚本可以直接修改提交说明，可以用于对提交说明规范化以符合项目的标准（如果说有的话）。如果提交说明不符合标准，可以拒绝提交。

Git提供的示例脚本:file:`commit-msg.sample`检查提交说明中出现的相同的Signed-off-by行，如果发现重复签名即报错、终止提交。

Gerrit服务器需要每一个向其进行推送的Git版本库在本地使用Gerrit提供的:file:`commit-msg`钩子脚本，以便在创建的提交中包含形如“Change-Id: I...”的变更集标签。

## post-commit

该钩子脚本由:command:`git commit`命令调用，不带参数运行，在提交完成之后被触发执行。

该钩子脚本不会影响:command:`git commit`的运行结果，可以用于发送通知。

## pre-rebase

该钩子脚本由:command:`git rebase`命令调用，用于防止某个分支参与变基。

Git提供的示例脚本:file:`pre-rebase.sample`是针对Git项目自身情况而开发的，当一个功能分支已经合并到next分支后，禁止该分基进行变基操作。

## post-checkout

该钩子脚本由:command:`git checkout`命令调用，是在完成工作区更新之后触发执行。该钩子脚本有三个参数：前一个HEAD的引用，新HEAD的引用（可能和前一个一样也可能不一样），以及一个用于表示此次检出是否是分支检出的标识（分支检出为1，文件检出是0）。该钩子脚本不会影响:command:`git checkout`命令的结果。

除了由:command:`git checkout`命令调用外，该钩子脚本也在:command:`git clone`命令执行后被触发执行，除非在克隆时使用了禁止检出的--no-checkout (-n)参数。在由:command:`git clone`调用时，第一个参数给出的引用是空引用，则第二个和第三个参数都为1。

这个钩子一般用于版本库的有效性检查，自动显示和前一个HEAD的差异，或者设置工作区属性。

## post-merge

该钩子脚本由:command:`git merge`命令调用，当在本地版本库完成:command:`git pull`操作后触发执行。该钩子脚本有一个参数，标识合并是否是一个压缩合并。该钩子脚本不会影响:command:`git merge`命令的结果。如果合并因为冲突而失败，该脚本不会执行。

该钩子脚本可以与:file:`pre-commit`钩子脚本一起实现对工作区目录树属性（如权限/属主/ACL等）的保存和恢复。参见Git源码文件:file:`contrib/hooks/setgitperms.perl`中的示例。

## pre-receive

该钩子脚本由远程版本库的:command:`git receive-pack`命令调用。当从本地版本库完成一个推送之后，在远程服务器上开始批量更新引用之前，该钩子脚本被触发执行。该钩子脚本的退出状态决定了更新引用的成功与否。

该钩子脚本在接收(receive)操作中只执行一次。传递参数不通过命令行，而是通过标准输入进行传递。通过标准输入传递的每一行的语法格式为：

```
<old-value> <new-value> <ref-name>
```

<old-value>是引用更新前的老的对象ID，<new-value>是引用即将更新到的新对象ID，<ref-name>是引用的全名。当创建一个新引用时，<old-value>是40个0。

如果该钩子脚本以非零值退出，一个引用也不会更新。如果该脚本正常退出，每一个单独的引用的更新仍有可能被update钩子所阻止。

标准输出和标准错误都重定向到在另外一端执行的:command:`git send-pack`，所以可以直接通过:command:`echo`命令向用户传递信息。

## update

该钩子脚本由远程版本库的:command:`git receive-pack`命令调用。当从本地版本库完成一个推送之后，在远程服务器上更新引用时，该钩子脚本被触发执行。该钩子脚本的退出状态决定了更新引用的成功与否。

该钩子脚本在每一个引用更新的时候都会执行一次。该脚本有三个参数。

- 参数1：要更新的引用的名称。
- 参数2：引用中保存的旧对象名称。
- 参数3：将要保存到引用中的新对象名称。

正常退出（返回0）允许引用的更新，而以非零值退出禁止：`:command:`git-receive-pack`` 更新该引用。

该钩子脚本可以用于防止对某些引用的强制更新，因为该脚本可以通过检查新旧引用对象是否存在继承关系，从而提供更为细致的“非快进式推送”的授权。

该钩子脚本也可以用于记录（如用邮件）引用变更历史`old..new`。然而因为该脚本不知道整个的分支，所以可能会导致每一个引用发送一封邮件。因此如果要发送通知邮件，可能`post-receive`钩子脚本更适合。

另外，该脚本可以实现基于路径的授权。

标准输出和标准错误都重定向到在另外一端执行的：`:command:`git send-pack``，所以可以直接通过：`:command:`echo`` 命令向用户传递信息。

Git提供的示例脚本：`:file:`update.sample``展示了对多种危险的Git操作行为进行控制的可行性。

- 只有将配置变量`hooks.allowunannotated`设置为`true`才允许推送轻量级里程碑（不带说明的里程碑）。
- 只有将配置变量`hooks.allowdeletebranch`设置为`true`才允许删除分支。
- 如果将配置变量`hooks.denycreatebranch`设置为`true`则不允许创建新分支。
- 只有将配置变量`hooks.allowdeletetag`设置为`true`才允许删除里程碑。
- 只有将配置变量`hooks.allowmodifytag`设置为`true`才允许修改里程碑。

相比Git的示例脚本，Gitolite服务器为其管理的版本库设置的`update`钩子脚本更实用也更强大。Gitolite实现了用户认证，并通过检查授权文件，实现基于分支和路径的写操作授权，等等。具体参见本书第5篇“第30章Gitolite服务架设”相关内容。

## post-receive

该钩子脚本由远程版本库的：`:command:`git receive-pack`` 命令调用。当从本地版本库完成一个推送，并且在远程服务器上所有引用都更新完毕后，该钩子脚本被触发执行。

该钩子脚本在接收（receive）操作中只执行一次。该脚本不通过命令行传递参数，但是像`pre-receive`钩子脚本那样，通过标准输入以相同格式获取信息。

该钩子脚本不会影响`git-receive-pack`的结果，因为调用该脚本时工作已经完成。

该钩子脚本胜过`post-update`脚本之处在于可以获得所有引用的老的和新的值，以及引用的名称。

标准输出和标准错误都重定向到在另外一端执行的：`:command:`git send-pack``，所以可以

直接通过`:command:`echo``命令向用户传递信息。

Git提供的示例脚本`:file:`post-receive.sample``引入了`:file:`contrib/hooks``目录下的名为`:file:`post-receive-email``的示例脚本（默认被注释），以实现发送通知邮件的功能。

Gitolite服务器要对其管理的Git版本库设置`:file:`post-receive``钩子脚本，以实现当版本库有变更后进行到各个镜像版本库的数据传输。

## post-update

该钩子脚本由远程版本库的`:command:`git receive-pack``命令调用。当从本地版本库完成一个推送之后，即当所有引用都更新完毕后，在远程服务器上该钩子脚本被触发执行。

该脚本接收不定长的参数，每一个参数实际上就是已成功更新的引用名。

该钩子脚本不会影响`:command:`git-receive-pack``的结果，因此主要用于通知。

钩子脚本post-update虽然能够提供那些引用被更新了，但是该脚本不知道引用更新前后的对象SHA1哈希值，所以在这个脚本中不能记录形如old..new的引用变更范围。而钩子脚本post-receive知道更新引用前后的对象ID，因此更适合此种场合。

标准输出和标准错误都重定向到在另外一端执行的`:command:`git send-pack``，所以可以直接通过`:command:`echo``命令向用户传递信息。

Git提供的示例脚本post-update.sample会运行`:command:`git update-server-info``命令，以更新哑协议需要的索引文件。如果通过哑协议共享版本库，应该启用该钩子脚本。

## pre-auto-gc

该钩子脚本由`:command:`git gc --auto``命令调用，不带参数运行，如果以非零值退出会导致`:command:`git gc --auto``被中断。

## post-rewrite

该钩子脚本由一些重写提交的命令调用，如`:command:`git commit --amend``、`:command:`git rebase``，而`:command:`git-filter-branch``当前尚未调用该钩子脚本。

该脚本的第一个参数用于判断调用来自哪个命令，当前有`amend`和`rebase`两个取值，也可能将来会有其他更多命令相关参数传递。

该脚本通过标准输入接收一个重写提交列表，每一行输入的格式如下：

```
<old-sha1> <new-sha1> [<extra-info>]
```

前两个是旧的和新的对象SHA1哈希值。而`<extra-info>`参数是和调用命令相关的，而当前该参数为空。

## Git模板

当执行`:command:`git init``或`:command:`git clone``创建版本库时，会自动在版本库中创建钩子脚本（`:file:`.git/hooks/*``）、忽略文件（`:file:`.git/info/exclude``）及其他文件，实际上这些文件均拷贝自模板目录。如果需要本地版本库使用定制的钩子脚本等文件，直接在模板目录内创建（文件或符号链接）会事半功倍。

Git按照以下列顺序第一个确认的路径即为模板目录。

- 如果执行`:command:`git init``或`:command:`git clone``命令时，提供`--template=<DIR>`参数，则使用指定的目录作为模板目录。
- 由环境变量`$GIT_TEMPLATE_DIR`指定的模板目录。
- 由Git配置变量`init.templatedir`指定的模板目录。
- 缺省的模板目录，根据Git安装路径的不同可能位于不同的目录下。可以通过下面命令确认其实际位置：

```
$ ( cd $(git --html-path)/../../git-core/templates; pwd )
/usr/share/git-core/templates
```

如果在执行版本库初始化时传递了空的模板路径，则不会在版本库中创建钩子脚本 等文件。

```
$ git init --template= no-template
Initialized empty Git repository in /path/to/my/workspace/no-template/.git/
```

执行下面的命令，查看新创建的版本库`:file:`.git``目录下的文件。

```
$ ls -F no-template/.git/
HEAD      config    objects/ refs/
```

可以看到不使用模板目录创建的版本库下面的文件少的可怜。而通过对模板目录下的文件的定制，可以实现在建立的版本库中包含预先设置好的钩子脚本、忽略文件、属性文件等。这对于服务器或者对版本库操作有特殊要求的项目带来方便。



# 稀疏检出和浅克隆

## 稀疏检出

从1.7.0版本开始Git提供稀疏检出的功能。所谓稀疏检出就是本地版本库检出时不检出全部，只将指定的文件从本地版本库检出到工作区，而其他未指定的文件则不予检出（即使这些文件存在于工作区，其修改也会被忽略）。

要想实现稀疏检出的功能，必须同时设置core.sparseCheckout配置变量，并存在文件`:file:`.git/info/sparse-checkout``。即首先要设置Git配置变量core.sparseCheckout为true，然后编辑`:file:`.git/info/sparse-checkout``文件，将要检出的目录或文件的路径写入其中。其中文件`:file:`.git/info/sparse-checkout``的格式就和`:file:`.gitignore``文件格式一样，路径可以使用通配符。

稀疏检出是如何实现的呢？实际上Git在index（即暂存区）中为每个文件提供一个名为skip-worktree标志位，缺省这个标识位处于关闭状态。如果该标识位开启，则无论工作区对应的文件存在与否，或者是否被修改，Git都认为工作区该文件的版本是最新的、无变化。Git通过配置文件`:file:`.git/info/sparse-checkout``定义一个要检出的目录和/或文件列表，当前Git的`:command:`git read-tree``命令及其他基于合并的命令`(:command:`git merge` , :command:`git checkout` 等等)`能够根据该配置文件更新index中文件的skip-worktree标志位，实现版本库文件的稀疏检出。

先来在工作区`:file:`/path/to/my/workspace``中创建一个示例版本库sparse1，创建后的sparse1版本库中包含如下内容：

```
$ ls -F
doc1/ doc2/ doc3/
$ git ls-files -s -v
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc1/readme.txt
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc2/readme.txt
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc3/readme.txt
```

即版本库sparse1中包含三个目录`:file:`doc1``、`:file:`doc2``和`:file:`doc3``。命令`:command:`git ls-files``的`-s`参数用于显示对象的SHA1哈希值以及所处的暂存区编号。而`-v`参数则还会显示工作区文件的状态，每一行命令输出的第一个字符即是文件状态：字母H表示文件已被暂存，如果是字母S则表示该文件skip-worktree标志位已开启。

下面我们就来体验一下稀疏检出的功能。

- 修改版本库的Git配置变量core.sparseCheckout，将其设置为true。

```
$ git config core.sparseCheckout true
```

- 设置`:file:`.git/info/sparse-checkout``的内容，如下：

```
$ printf "doc1\ndoc3\n" > .git/info/sparse-checkout
$ cat .git/info/sparse-checkout
doc1
doc3
```

- 执行`:command:`git checkout``命令后，会发现工作区中`:file:`doc2``目录不见了。

```
$ git checkout
$ ls -F
doc1/ doc3/
```

- 这时如果用`:command:`git ls-files``命令查看，会发现`:file:`doc2``目录下的文件被设置了`skip-worktree`标志。

```
$ git ls-files -v
H doc1/readme.txt
S doc2/readme.txt
H doc3/readme.txt
```

- 修改`:file:`.git/info/sparse-checkout``的内容，如下：

```
$ printf "doc3\n" > .git/info/sparse-checkout
$ cat .git/info/sparse-checkout
doc3
```

- 执行`:command:`git checkout``命令后，会发现工作区中`:file:`doc1``目录也不见了。

```
$ git checkout
$ ls -F
doc3/
```

- 这时如果用`:command:`git ls-files``命令查看，会发现`:file:`doc1``和`:file:`doc2``目录下的文件都被设置了`skip-worktree`标志。

```
$ git ls-files -v
```

```
S doc1/readme.txt  
S doc2/readme.txt  
H doc3/readme.txt
```

- 修改:`:file:`.git/info/sparse-checkout``的内容，使之包含一个星号，即在工作区检出所有的内容。

```
$ printf "*\n" > .git/info/sparse-checkout  
$ cat .git/info/sparse-checkout  
*
```

- 执行:`:command:`git checkout``，会发现所有目录又都回来了。

```
$ git checkout  
$ ls -F  
doc1/ doc2/ doc3/
```

文件:`:file:`.git/info/sparse-checkout``的文件格式类似于:`:file:`.gitignore``的格式，也支持用感叹号实现反向操作。例如不检出目录:`:file:`doc2``下的文件，而检出其他文件，可以使用下面的语法（注意顺序不能写反）：

```
*
```

注意如果使用命令:`:command:`git checkout -- <file>...``，即不是切换分支而是用分支中的文件替换暂存区和工作区的话，则忽略`skip-worktree`标志。例如下面的操作中，虽然:`:file:`doc2``被设置为不检出，但是执行:`:command:`git checkout .``命令后，还是所有的目录都被检出了。

```
$ git checkout .  
$ ls -F  
doc1/ doc2/ doc3/  
$ git ls-files -v  
H doc1/readme.txt  
S doc2/readme.txt  
H doc3/readme.txt
```

如果修改:`:file:`doc2``目录下的文件，或者在:`:file:`doc2``目录下添加新文件，Git会视而不见。

```
$ echo hello >> doc2/readme.txt
```

```
$ git status  
# On branch master  
nothing to commit (working directory clean)
```

若此时通过取消`core.sparseCheckout`配置变量的设置而关闭稀疏检出，也不会改变目录`:file:`doc2``下的文件的`skip-worktree`标志。这种情况或者通过`:command:`git update-index --no-skip-worktree -- <file>...``来更改index中对应文件的`skip-worktree`标志，或者重新启用稀疏检出更改相应文件的检出状态。

在克隆一个版本库时只希望检出部分文件或目录，可以在执行克隆操作的时候使用`--no-checkout`或`-n`参数，不进行工作区文件的检出。例如下面的操作从前面示例的`sparse1`版本库克隆到`sparse2`中，不进行工作区文件的检出。

```
$ git clone -n sparse1 sparse2  
Cloning into sparse2...  
done.
```

检出完成后可以发现`sparse2`的工作区是空的，而且版本库中也不存在`:file:`index``文件。如果执行`:command:`git status``命令会看到所有文件都被标识为删除。

```
$ cd sparse2  
$ git status -s  
D doc1/readme.txt  
D doc2/readme.txt  
D doc3/readme.txt
```

如果希望通过稀疏检出的功能，只检出其中一个目录如`:file:`doc2``，可以用如下方法实现：

```
$ git config core.sparseCheckout true  
$ printf "doc2\n" > .git/info/sparse-checkout  
$ git checkout
```

之后看到工作区中检出了`:file:`doc2``目录，而其他文件被设置了`skip-worktree`标志。

```
$ ls -F  
doc2/  
$ git ls-files -v  
S doc1/readme.txt  
H doc2/readme.txt  
S doc3/readme.txt
```

## 浅克隆

上一节介绍的稀疏检出，可以部分检出版本库中的文件，但是版本库本身仍然包含所有的文件和历史。如果只对一个大的版本库的最近的部分历史提交感兴趣，而不想克隆整个版本库，稀疏检出是解决不了的，而是要采用本节介绍的浅克隆。

实现版本库的浅克隆的非常简单，只需要在执行:`git clone`或者`git fetch`操作时用`--depth <depth>`参数设定要获取的历史提交的深度（`<depth>`大于0），就会把源版本库分支上最近的`<depth> + 1`个历史提交作为新版本库的全部历史提交。

通过浅克隆方式克隆出来的版本库，每一个提交的SHA1哈希值和源版本库的相同，包括提交的根节点也是如此，但是Git通过特殊的实现，使得浅克隆的根节点提交看起来没有父提交。正因为浅克隆的提交对象的SHA1哈希值和源版本库一致，所以浅克隆版本库可以执行`git fetch`或者`git pull`从源版本库获取新的提交。但是浅克隆版本库也存在着很多限制，如：

- 不能从浅克隆版本库克隆出新的版本库。
- 其他版本库不能从浅克隆获取提交。
- 其他版本库不能推送提交到浅克隆版本库。
- 不要从浅克隆版本库推送提交至其他版本库，除非确认推送的目标版本库包含浅克隆版本库中缺失的全部历史提交，否则会造成目标版本库包含不完整的提交历史导致版本库无法操作。
- 在浅克隆版本库中执行合并操作时，如果所合并的提交出现在浅克隆历史中，则可以顺利合并，否则会出现大量的冲突，就好像和无关的历史进行合并一样。

由于浅克隆包含上述限制，因此浅克隆一般用于对远程版本库的查看和研究，如果在浅克隆版本库中进行了提交，最好通过`git format-patch`命令导出为补丁文件再应用到远程版本库中。

下面的操作使用`git clone`命令创建一个浅克隆。注意：源版本库如果是本地版本库要使用`file://`协议，若直接接使用本地路径则不会实现浅克隆。

```
$ git clone --depth 2 file:///path/to/repos/hello-world.git shallow1
```

然后进入到本地克隆目录中，会看到当前分支上只有3个提交。

```
$ git log --oneline  
c4acab2 Translate for Chinese.  
683448a Add I18N support.  
d81896e Fix typo: -help to --help.
```

查看提交的根节点d81896e，则会看到该提交实际上也包含父提交。

```
$ git cat-file -p HEAD^^  
tree f9d7f6b0af6f3ffffa74eb995f1d781d3c4876b25  
parent 10765a7ef46981a73d578466669f6e17b73ac7e3  
author user1 <user1@sun.ossxp.com> 1294069736 +0800  
committer user2 <user2@moon.ossxp.com> 1294591238 +0800  
  
Fix typo: -help to --help.
```

而查看该提交的父提交，Git会报错。

```
$ git log 10765a7ef46981a73d578466669f6e17b73ac7e3  
fatal: bad object 10765a7ef46981a73d578466669f6e17b73ac7e3
```

对于正常的Git版本库来说，如果对象库中一个提交丢失绝对是大问题，版本库不可能被正常使用。而浅克隆之所以看起来一切正常，是因为Git使用了类似嫁接（下一节即将介绍）的技术。

在浅克隆版本库中存在一个文件:[file: `.git/shallow`](#)，这个文件中罗列了应该被视为提交根节点的提交SHA1哈希值。查看这个文件会看到提交d81896e正在其中：

```
$ cat .git/shallow  
b56bb510a947651e4717b356587945151ac32166  
d81896e60673771ef1873b27a33f52df75f70515  
e64f3a216d346669b85807ffcfb23a21f9c5c187
```

列在[file: `.git/shallow`](#)文件中的提交会构建出对应的嫁接提交，使用类似嫁接文件[file: `.git/info/grafts`](#)（下节讨论）的机制，当Git访问这些对象时就好像这些对象是没有父提交的根节点一样。

# 嫁接和替换

## 提交嫁接

提交嫁接可以实现在本地版本库上将两条完全不同提交线（分支）嫁接（连接）到一起。对于一些项目将版本控制系统迁移到Git上，该技术会非常有帮助。例如Linux本身的源代码控制在转移到Git上时，尚没有任何工具可以将Linux的提交历史从旧的Bitkeeper版本控制系统中导出，后来Linux旧的代码通过bkcvn导入到Git中，如何将新旧两条开发线连接到一起呢？于是发明了提交嫁接，实现新旧两条开发线的合并，这样Linux开发者就可以在一个开发分支中由最新的提交追踪到原来位于Bitkeeper中的提交。（参考：<https://git.wiki.kernel.org/index.php/GraftPoint>）

提交嫁接是通过在版本库中创建:`:file:`.git/info/grafts``文件实现的。该文件每一行的格式为：

```
<commit sha1> <parent sha1> [<parent sha1>]*
```

用空格分开各个字段，其中第一个字段是一个提交的SHA1哈希值，而后面用空格分开的其他SHA1哈希值则作为该提交的父提交。当把一个提交线的根节点作为第一个字段，将第二个提交线顶节点作为第二个字段，就实现了两个提交线的嫁接，看起来像是一条提交线了。

在本书第6篇“第35.4节Git版本库整理”中介绍的:`:command:`git filter-branch``命令在对版本库整理时，如果发现存在:`:file:`.git/info/grafts``则会在物理上完成提交的嫁接，实现嫁接的永久生效。

## 提交替换

提交替换是在1.6.5或更新版本的Git提供的功能，和提交嫁接类似，不过提交替换不是用一个提交来伪装另外一个提交的父提交，而是直接替换另外的提交，实现在不影响其他提交的基础上实现对历史提交的修改。

提交替换是通过在特殊命名空间:`:file:`.git/refs/replace/``下定义引用实现的。引用的名称是要被替换掉的提交SHA1哈希值，而引用文件的内容（引用所指向的提交）就是用于替换的（正确的）提交SHA1哈希值。由于提交替换通过引用进行定义，因此可以在不同的版本库之间传递，而不像提交嫁接只能在本地版本库中使用。

Git提供:`:command:`git replace``命令来管理提交替换，用法如下：

```
用法1: git replace [-f] <object> <replacement>
用法2: git replace -d <object>...
用法3: git replace -l [<pattern>]
```

其中：

- 用法1用于创建提交替换，即在`:file:`.git/refs/replace``目录下创建名为`<object>`的引用，其内容为`<replacement>`。如果使用`-f`参数，还允许级联替换，用于替换的提交来可以是另外一个已经在`:file:`.git/refs/replace``中定义的替换。
- 用法2用于删除已经定义的替换。
- 用法3显示已经存在的提交替换。

提交替换可以被大部分Git命令理解，除了一些针对被替换的提交使用`--no-replace-objects`参数的命令。例如：

- 当提交`foo`被提交`bar`替换后，显示未被替换前的`foo`提交：

```
$ git --no-replace-objects cat-file commit foo
...foo 的内容...
```

- 不使用`--no-replace-objects`参数，则访问`foo`会显示替换后的`bar`提交：

```
$ git cat-file commit foo
...bar 的内容...
```

提交替换使用引用进行定义，因此可以通过`:command:`git fetch``和`:command:`git push``在版本库之间传递。但因为默认Git只同步里程碑和分支，因此需要在命令中显式的给出提交替换的引用表达式，如：

```
$ git fetch origin refs/replace/*
$ git push origin refs/replace/*
```

提交替换也可以实现两个分支的嫁接。例如要将分支A嫁接到B上，就相当于将分支A的根提交`<BRANCH_A_ROOT>`的父提交设置为分支B的最新提交`<BRANCH_B_CURRENT>`。可以先创建一个新提交`<BRANCH_A_NEW_ROOT>`，其父提交设置为`<BRANCH_B_CURRENT>`而提交的其他字段完全和`<BRANCH_A_ROOT>`一致。然后设置提交替换，用`<BRANCH_A_NEW_ROOT>`替换`<BRANCH_A_ROOT>`即可。

创建`<BRANCH_A_NEW_ROOT>`可以使用下面的命令，注意要用实际值替换下面命令中的`<BRANCH_A_ROOT>`和`<BRANCH_B_CURRENT>`。

```
$ git cat-file commit <BRANCH_A_ROOT> |  
  sed -e "/^tree / a \  
    parent $(git rev-parse <BRANCH_B_CURRENT>)" |  
  git hash-object -t commit -w --stdin
```

其中:`git cat-file commit`命令用于显示提交的原始信息, `sed`命令用于向原始提交中插入一条parent SHA1...的语句, 而命令:`git hash-object`是一个Git底层命令, 可以将来自标准输入的内容创建一个新的提交对象。

上面命令的输出即是`<BRANCH_A_NEW_ROOT>`的值。执行下面的替换命令, 完成两个分支的嫁接。

```
$ git replace <BRANCH_A_ROOT> <BRANCH_A_NEW_ROOT>
```

# Git评注

从1.6.6版本开始，Git提供了一个`:command:`git notes``命令可以为提交添加评注，实现  
在不改变提交对象的情况下在提交说明的后面附加评注。图41-1展示了  
Github (<https://github.com/ossxp-com/gitdemo-commit-tree/commit/6652a0dce6a5067732c00ef0a220810a7230655e>) 利用`:command:`git notes``实现的在提交显示界面中显示评注（如果存在的话）和添加评注的界面。

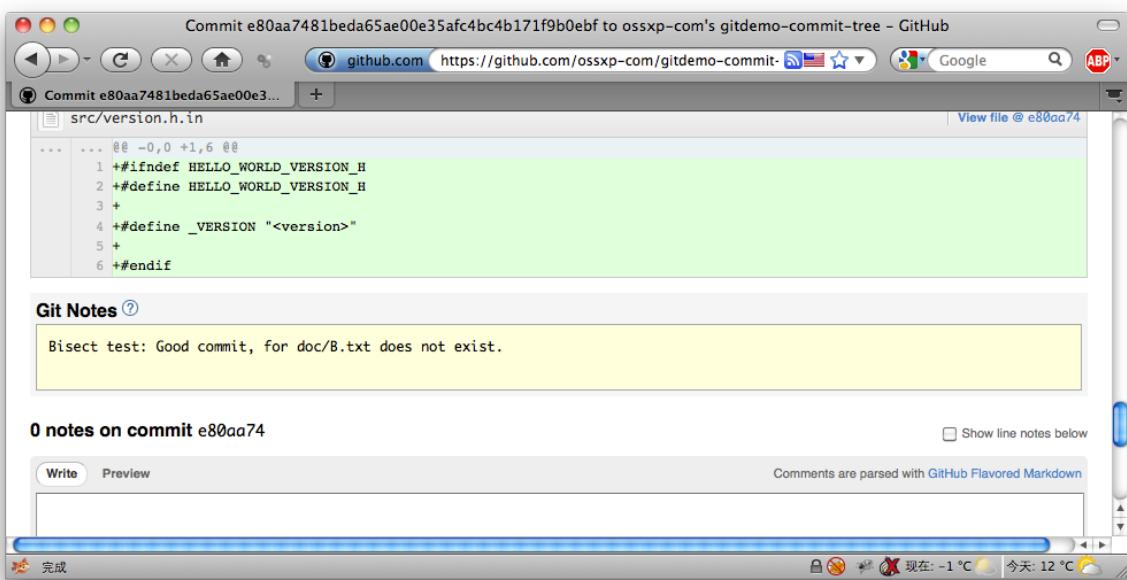


图41-1：Github上显示和添加评注

## 评注的奥秘

实际上Git评注可以针对任何对象，而且评注的内容也不限于文字，因为评注的内容是保存在Git对象库中的一个blob对象中。不过评注目前最主要的应用还是在提交说明后添加文字评注。

在第2篇“第11.4.6节二分查找”中用到的gitdemo-commit-tree版本库实际上就包含了提交评注，只不过之前尚未将评注获取到本地版本库而已。如果工作区中的gitdemo-commit-tree版本库已经不存在，可以使用下面的命令从GitHub上再克隆一个：

```
$ git clone -q git://github.com/ossxp-com/gitdemo-commit-tree.git
$ cd gitdemo-commit-tree
```

执行下面的命令，查看最后一次提交的提交说明：

```
$ git log -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

    Add Images for git treeview.

Signed-off-by: Jiang Xin <jiangxin@ossp.com>
```

下面为默认的origin远程版本库再添加一个引用获取表达式，以便在执行:`git fetch`命令时能够同步评注相关的引用。命令如下：`

```
$ git config --add remote.origin.fetch refs/notes/*:refs/notes/*
```

执行:`git fetch`会获取到一个新的引用refs/notes/commits，如下：`

```
$ git fetch
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From git://github.com/ossp-com/gitdemo-commit-tree
 * [new branch]      refs/notes/commits -> refs/notes/commits
```

当获取到新的评注相关的引用之后，再来查看最后一次提交的提交说明。下面的命令输出中提交说明的最后两行就是附加的提交评注。

```
$ git log -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

    Add Images for git treeview.

Signed-off-by: Jiang Xin <jiangxin@ossp.com>

Notes:
  Bisect test: Bad commit, for doc/B.txt exists.
```

附加的提交评注来自于哪里呢？显然应该和刚刚获取到的引用相关。查看一下获取到的最新引用，会发现引用`refs/notes/commits`指向的是一个提交对象。

```
$ git show-ref refs/notes/commits
```

```
6f01cdc5900489274119318ceb2330d6dc0cef1 refs/notes/commits
$ git cat-file -t refs/notes/commits
commit
```

既然新获取的评注引用是一个提交对象，那么就应该能够查看评注引用的提交日志：

```
$ git log --stat refs/notes/commits
commit 6f01cdc5900489274119318ceb2330d6dc0cef1
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Tue Feb 22 09:32:10 2011 +0800

    Notes added by 'git notes add'

6652a0dce6a5067732c00ef0a220810a7230655e |      1 +
1 files changed, 1 insertions(+), 0 deletions(-)

commit 9771e1076d2218922acc9800f23d5e78d5894a9f
Author: Jiang Xin <jiangxin@ossp.com>
Date:   Tue Feb 22 09:31:54 2011 +0800

    Notes added by 'git notes add'

e80aa7481beda65ae00e35afc4bc4b171f9b0ebf |      1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

从上面的评注引用的提交日志可以看出，存在两次提交，并且从提交说明可以看出是使用`:command:`git notes add``命令添加的。至于每次提交添加的文件却很让人困惑，所添加文件的文件名居然是40位的哈希值。

您当然可以通过`:command:`git checkout -b``命令检出该引用来研究其中所包含的文件，不过也可以运用我们已经学习到的Git命令直接对其进行研究。

- 用`:command:`git show``命令显示目录树。

```
$ git show -p refs/notes/commits^{tree}
tree refs/notes/commits^{tree}

6652a0dce6a5067732c00ef0a220810a7230655e
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
```

- 用`:command:`git ls-tree``命令查看文件大小及对应的blob对象的SHA1哈希值。

```
$ git ls-tree -l refs/notes/commits
100644 blob 80b1d249069959ce5d83d52ef7bd0507f774c2b0      47      6652a0dce
100644 blob e894f2164e77abf08d95d9bdad4cd51d00b47845      56      e80aa7481
```

- 文件名既然是一个40位的SHA1哈希值，那么文件名一定有意义，通过下面的命令可以看到文件名包含的40位哈希值实际对应于一个提交。

```
$ git cat-file -p 6652a0dce6a5067732c00ef0a220810a7230655e  
tree e33be9e8e7ca5f887c7d5601054f2f510e6744b8  
parent 81993234fc12a325d303eccea20f6fd629412712  
author Jiang Xin <jiangxin@ossp.com> 1291882031 +0800  
committer Jiang Xin <jiangxin@ossp.com> 1291882892 +0800  
  
Add Images for git treeview.  
  
Signed-off-by: Jiang Xin <jiangxin@ossp.com>
```

- 用`:command:`git cat-file``命令查看该文件的内容，可以看到其内容就是附加在相应提交上的评注。

```
$ git cat-file -p refs/notes/commits:6652a0dce6a5067732c00ef0a220810a7230  
Bisect test: Bad commit, for doc/B.txt exists.
```

综上所述，评注记录在一个blob对象中，并且以所评注对象的SHA1哈希值命名。因为对象SHA1哈希值的唯一性，所以可以将评注都放在同一个文件系统下而不会相互覆盖。针对这个包含所有评注的特殊的文件系统的更改被提交到一个特殊的引用`refs/notes/commits`当中。

## 评注相关命令

Git提供了`:command:`git notes``命令，对评注进行管理。如果执行`:command:`git notes list``或者像下面这样不带任何参数进行调用，会显示和上面`:command:`git ls-tree``类似的输出：

```
$ git notes  
80b1d249069959ce5d83d52ef7bd0507f774c2b0 6652a0dce6a5067732c00ef0a220810a7230  
e894f2164e77abf08d95d9bdad4cd51d00b47845 e80aa7481beda65ae00e35afc4bc4b171f9b
```

右边的一列是要评注的提交对象，而左边一列是附加在对应提交上的包含评注内容的blob对象。显示附加在某个提交上的评注可以使用`:command:`git notes show``命令。如下：

```
$ git notes show G^0
```

```
Bisect test: Good commit, for doc/B.txt does not exist.
```

注意上面的命令中使用`G^0`而非`G`, 是因为`G`是一个里程碑对象, 而评注是建立在由里程碑对象所指向的一个提交对象上。

添加评注可以使用下面的:`git notes add` 和:`git notes append` 子命令:

```
用法1: git notes add [-f] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>  
用法2: git notes append [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]
```

用法1是添加评注, 而用法2是在已有评注后面追加。两者的命令行格式和:`git commit` 非常类似, 可以用类似写提交说明的方法写提交评注。如果省略最后一个`<object>`参数, 则意味着向头指针HEAD添加评注。子命令:`git notes add` 中的参数`-f`意味着强制添加, 会覆盖对象已有的评注。

使用:`git notes copy` 子命令可以将一个对象的评注拷贝到另外一个对象上。

```
用法: git notes copy [-f] ( --stdin | <from-object> <to-object> )
```

修改评注可以使用下面的:`git notes edit` 子命令:

```
用法: git notes edit [<object>]
```

删除评注可以使用的:`git notes remote` 子命令, 而:`git notes prune` 则可以清除已经不存在的对象上的评注。用法如下:

```
用法1: git notes remove [<object>]  
用法2: git notes prune [-n | -v]
```

评注以文件形式保存在特殊的引用中, 如果该引用被共享并且同时有多人撰写评注时, 有可能出现该引用的合并冲突。可以用:`git notes merge` 命令来解决合并冲突。评注引用也可以使用其他的引用名称, 合并其他的评注引用也可以使用本命令。下面是:`git notes merge` 命令的语法格式, 具体操作参见:`git help notes` 帮助。

```
用法1: git notes merge [-v | -q] [-s <strategy> ] <notes_ref>  
用法2: git notes merge --commit [-v | -q]  
用法3: git notes merge --abort [-v | -q]
```

## 评注相关配置

默认提交评注保存在引用`refs/notes/commits`中，这个默认的设置可以通过`core.notesRef`配置变量修改。如须更改，要在`core.notesRef`配置变量中使用引用的全称而不能使用缩写。

在执行`:command:`git log``命令显示提交评注的时候，如果配置了`notes.displayRef`配置变量（可以使用通配符，并且可以配置多个），则在显示提交评注时，除了会参考`core.notesRef`设定的引用（或默认的`refs/notes/commits`引用）外，还会参考`notes.displayRef`指向的引用（一个或多个）来显示评注。

配置变量`notes.rewriteRef`用于配置哪个/哪些引用中的提交评注会随着提交的修改而复制到新的提交之上。这个配置变量可以使用多次，或者使用通配符，但该配置变量没有缺省值，因此为了使得提交评注能够随着提交的修改（修补提交、变基等）继续保持，必须对该配置变量进行设定。如：

```
$ git config --global notes.rewriteRef refs/notes/*
```

还有`notes.rewrite.amend`和`notes.rewrite.rebase`配置变量可以分别对两种提交修改模式（`amend`和`rebase`）是否启用评注复制进行设置，默认启用。配置变量`notes.rewriteMode`默认设置为`concatenate`，即提交评注复制到修改后的提交时，如果有评注则对评注进行合并操作。

## A. Git命令索引

每一个Git子命令都和特定目录下的一个名为:`:file:`git-<cmd>``的文件相对应，也就是在这个特定目录下存在的名为:`:file:`git-<cmd>``的可执行文件（有几个脚本文件被其他脚本包含提供相应的函数库，不能单独运行。如:`:file:`git-sh-setup``）可以用命令`:command:`git <cmd>``执行。这个特定的目录的位置可以用下面的命令查看：

```
$ git --exec-path
/usr/lib/git-core/
```

在这个目录下有150多个可执行文件，也就是说Git有非常多的子命令。在如此众多的子命令中，实际上常用的只有三分之一不到，其余的命令或者做为底层命令供其他命令及脚本调用，或者用于某些生僻场合，或者已经过时但出于兼容性的考虑而仍然健在。下面的表格分门别类的对所有Git命令做一概要性介绍，凡是在本书出现过的命令标以章节号和页码。

### A.1 常用的Git命令

命令	相关章节	页数	简要说明
git add	4.1; 10.2.3; 10.6	xxx; xxx; xxx	添加至暂存区
git add--interactive	10.6		交互式添加
git apply	20.2		应用补丁
git am	20.2		应用邮件格式补丁
git annotate	-	-	同义词，等同于 <code>git blame</code>
git archive	10.9		文件归档打包
git bisect	11.4.6		二分查找
git blame	11.4.5		文件逐行追溯
git branch	18.2		分支管理
git cat-file	6.1		版本库对象研究工具
git checkout	8.1; 18.4.2		检出到工作区、切换或创建分支
git cherry-pick	12.3.1		提交拣选
git citool	11.1		图形化提交，相当于 <code>git gui</code> 命令
git clean	5.3		清除工作区未跟踪文件
git clone	13.1; 41.3.2		克隆版本库
git commit	4.1; 4.4; 5.4		提交
git config	4.3		查询和修改配置
git describe	17.1		通过里程碑直观地显示提交ID
git diff	5.3		差异比较
git difftool	-	-	调用图形化差异比较工具
git fetch	19.1		获取远程版本库的提交
git format-patch	20.1		创建邮件格式的补丁文件。参见 <code>git am</code> 命令

git grep	4. 2		文件内容搜索定位工具
git gui	11. 1		基于Tcl/Tk的图形化工具，侧重提交等操作
git help	3. 1. 2		帮助
git init	4. 1; 13. 4		版本库初始化
git init-db*	-	-	同义词，等同于 git init
git log	11. 4. 3		显示提交日志
git merge	16. 1		分支合并
git mergetool	16. 4. 2		图形化冲突解决
git mv	10. 4		重命名
git pull	13. 1		拉回远程版本库的提交
git push	13. 1		推送至远程版本库
git rebase	12. 3. 2		分支变基
git rebase--interactive	12. 3. 3		交互式分支变基
git reflog	7. 2		分支等引用变更记录管理
git remote	19. 3		远程版本库管理
git repo-config*	-	-	同义词，等同于 git config
git reset	7. 1		重置改变分支“游标”指向
git rev-parse	4. 2; 11. 4. 1		将各种引用表示法转换为哈希值等
git revert	12. 5		反转提交
git rm	10. 2. 2		删除文件
git show	11. 4. 3		显示各种类型的对象
git stage*	-	-	同义词，等同于 git add
git stash	9. 2		保存和恢复进度
git status	5. 1		显示工作区文件状态
git tag	17. 1		里程碑管理

## A. 2 对象库操作相关命令

命令	相关章节	页数	简要说明
git commit-tree	12. 4		从树对象创建提交
git hash-object	41. 4. 2		从标准输入或文件计算哈希值或创建对象
git ls-files	5. 3; 18. 4. 2		显示工作区和暂存区文件
git ls-tree	5. 3		显示树对象包含的文件
git mktag	-	-	读取标准输入创建一个里程碑对象
git mktree	-	-	读取标准输入创建一个树对象
git read-tree	24. 2		读取树对象到暂存区
git update-index	41. 3. 1		工作区内容注册到暂存区及暂存区管理

git unpack-file	-	-	创建临时文件包含指定 blob 的内容
git write-tree	5. 3		从暂存区创建一个树对象

### A. 3 引用操作相关命令

命令	相关章节	页数	简要说明
git check-ref-format	17. 7		检查引用名称是否符合规范
git for-each-ref	-	-	引用迭代器，用于shell编程
git ls-remote	13. 4		显示远程版本库的引用
git name-rev	17. 1		将提交ID显示为友好名称
git peek-remote*	-	-	过时命令，请使用 git ls-remote
git rev-list	11. 4. 2		显示版本范围
git show-branch	-	-	显示分支列表及拓扑关系
git show-ref	14. 1		显示本地引用
git symbolic-ref	-	-	显示或者设置符号引用
git update-ref	-	-	更新引用的指向
git verify-tag	-	-	校验 GPG 签名的Tag

### A. 4 版本库管理相关命令

命令	相关章节	页数	简要说明
git count-objects	-	-	显示松散对象的数量和磁盘占用
git filter-branch	35. 4		版本库重构
git fsck	14. 2		对象库完整性检查
git fsck-objects*	-	-	同义词，等同于 git fsck
git gc	14. 4		版本库存储优化
git index-pack	-	-	从打包文件创建对应的索引文件
git lost-found*	-	-	过时，请使用 git fsck --lost-found 命令
git pack-objects	-	-	从标准输入读入对象ID，打包到文件
git pack-redundant	-	-	查找多余的 pack 文件
git pack-refs	14. 1		将引用打包到 .git/packed-refs 文件中
git prune	14. 2		从对象库删除过期对象
git prune-packed	-	-	将已经打包的松散对象删除
git relink	-	-	为本地版本库中相同的对象建立硬连接
git repack	14. 4		将版本库未打包的松散对象打包
git show-index	14. 1		读取包的索引文件，显示打包文件中的内容
git unpack-objects	-	-	从打包文件释放文件
git verify-pack	-	-	校验对象库打包文件

## A. 5 数据传输相关命令

命令	相关章节	页数	简要说明
git fetch-pack	15. 1		执行 git fetch 或 git pull 命令时在本地执行此命令，用于从其他版本库获取缺失的对象
git receive-pack	15. 1		执行 git push 命令时在远程执行的命令，用于接受推送的数据
git send-pack	15. 1		执行 git push 命令时在本地执行的命令，用于向其他版本库推送数据
git upload-archive	-	-	执行 git archive --remote 命令基于远程版本库创建归档时，远程版本库执行此命令传递归档
git upload-pack	15. 1		执行 git fetch 或 git pull 命令时在远程执行此命令，将对象打包、上传

## A. 6 邮件相关命令

命令	相关章节	页数	简要说明
git imap-send	-	-	将补丁通过 IMAP 发送
git mailinfo	-	-	从邮件导出提交说明和补丁
git mailsplit	-	-	将 mbox 或 Maildir 格式邮箱中邮件逐一提取为文件
git request-pull	21. 2. 1		创建包含提交间差异和执行PULL操作地址的信息
git send-email	20. 1		发送邮件

## A. 7 协议相关命令

命令	相关章节	页数	简要说明
git daemon	28. 2		实现Git协议
git http-backend	27. 2		实现HTTP协议的CGI程序，支持智能HTTP协议
git instaweb	27. 3. 4		即时启动浏览器通过 gitweb 浏览当前版本库
git shell	-	-	受限制的shell，提供仅执行Git命令的SSH访问
git update-server-info	15. 1		更新哑协议需要的辅助文件
git http-fetch	-	-	通过HTTP协议获取版本库
git http-push	-	-	通过HTTP/DAV协议推送
git remote-ext	-	-	由Git命令调用，通过外部命令提供扩展协议支持
git remote-fd	-	-	由Git命令调用，使用文件描述符作为协议接口
git remote-ftp	-	-	由Git命令调用，提供对FTP协议的支持
git remote-ftps	-	-	由Git命令调用，提供对FTPS协议的支持
git remote-http	-	-	由Git命令调用，提供对HTTP协议的支持
git remote-https	-	-	由Git命令调用，提供对HTTPS协议的支持

git remote-testgit	-	-	协议扩展示例脚本
--------------------	---	---	----------

## A. 8 版本库转换和交互相关命令

命令	相关章节	页数	简要说明
git archimport	-	-	导入Arch版本库到Git
git bundle	-	-	提交打包和解包，以便在不同版本库间传递
git cvsexportcommit	-	-	将Git的一个提交作为一个CVS检出
git cvsimport	-	-	导入CVS版本库到Git。或者使用 cvs2git
git cvsserver	-	-	Git的CVS协议模拟器，可供CVS命令访问Git版本库
git fast-export	-	-	将提交导出为 git-fast-import 格式
git fast-import	35. 3		其他版本库迁移至Git的通用工具
git svn	26. 1		Git 作为前端操作 Subversion

## A. 9 合并相关的辅助命令

命令	相关章节	页数	简要说明
git merge-base	11. 4. 2		供其他脚本调用，找到两个或多个提交最近的共同祖先
git merge-file	-	-	针对文件的两个不同版本执行三向文件合并
git merge-index	-	-	对index中的冲突文件调用指定的冲突解决工具
git merge-octopus	-	-	合并两个以上分支。参见 git merge 的 octopus 合并策略
git merge-one-file	-	-	由 git merge-index 调用的标准辅助程序
git merge-ours	-	-	合并使用本地版本，抛弃他人版本。参见 git merge 的 ours 合并策略
git merge-recursive	-	-	针对两个分支的三向合并。参见 git merge 的 recursive 合并策略
git merge-resolve	-	-	针对两个分支的三向合并。参见 git merge 的 resolve 合并策略
git merge-subtree	-	-	子树合并。参见 git merge 的 subtree 合并策略
git merge-tree	-	-	显式三向合并结果，不改变暂存区
git fmt-merge-msg	-	-	供执行合并操作的脚本调用，用于创建一个合并提交说明
git rerere	-	-	重用所记录的冲突解决方案

## A. 10 杂项

命令	相关章节	页数	简要说明
git bisect--helper	-	-	由 git bisect 命令调用，确认二分查找进度
git check-attr	41. 1. 2		显示某个文件是否设置了某个属性
git checkout-index	-	-	从暂存区拷贝文件至工作区
git cherry	-	-	查找没有合并到上游的提交
git diff-files	-	-	比较暂存区和工作区，相当于 git diff --raw
git diff-index	-	-	比较暂存区和版本库，相当于 git diff --cached --raw

git diff-tree	-	-	比较两个树对象，相当于 git diff --raw A B
git difftool--helper	-	-	由 git difftool 命令调用，默认要使用的差异比较工具
git get-tar-commit-id	10. 9		从 git archive 创建的 tar 包中提取提交ID
git gui--askpass	-	-	命令 git gui 的获取用户口令输入界面
git notes	41. 5		提交评论管理
git patch-id	-	-	补丁过滤行号和空白字符后生成补丁唯一ID
git quiltimport	20. 3. 2		将Quilt补丁列表应用到当前分支
git replace	41. 4. 2		提交替换
git shortlog	-	-	对 git log 的汇总输出，适合于产品发布说明
git strip-space	-	-	删除空行，供其他脚本调用
git submodule	23. 1		子模组管理
git tar-tree	-	-	过时命令，请使用 git archive
git var	-	-	显示 Git 环境变量
git web--browse	-	-	启动浏览器以查看目录或文件
git whatchanged	-	-	显示提交历史及每次提交的改动
git-mergetool--lib	-	-	包含于其他脚本中，提供合并/差异比较工具的选择和执行
git-parse-remote	-	-	包含于其他脚本中，提供操作远程版本库的函数
git-sh-setup	-	-	包含于其他脚本中，提供 shell 编程的函数库

# Git与CVS面对面

## 面对面访谈录

**Git:** 我的提交是原子提交。每次提交都对应于一个目录树（树对象）。因为我的提交ID是对目录树及相关的提交信息建立的一个SHA1哈希值，所以可以保证数据的完整性。

**CVS:** 我承认这是我的软肋，一次错误或冲突的提交会导致部分数据被提交，而部分数据没有提交，版本库完整性被破坏，所以人们才设计出来Subversion（SVN）来取代我。

**Git:** 我的分支和里程碑管理非常快捷。因为我的分支和里程碑就是一个记录提交ID的引用，你的呢？

**CVS:** 你怎么又提到别人的痛处了！我的分支和里程碑创建速度还是很快的，...嗯...，如果在版本库中只有几个文件的话。当然如果版本库的文件的很多，创建分支、里程碑创建就需要花费更多的时间。有些人对此忍无可忍，于是设计出SVN来取代我。

**Git:** 其实我不用里程碑都没有关系，因为每一个提交ID就对应于唯一的一个提交状态。

**CVS:** 这也是我做不到的。我没有全局版本号的概念，每一个文件都通过单独的版本号记录其变更历史，所以人们在使用我的时候必须经常地用里程碑（tag）对我的状态进行标识。还需要提醒一句，如果版本库中文件太多，创建里程碑是很耗时的，因为要一一打开每一个版本库中的文件，在其中记录里程碑和文件版本的关系。

**Git:** 我的工作区很干净。只在工作区的根目录下有一个`:file:`.git``目录，此外再无其他辅助目录或文件。

**CVS:** 我要在工作区的每一个目录下都放置一个CVS目录，这个目录下有一个`:file:`Entries``文件很重要，记录了对应工作区文件的检出版本以及时间戳等信息。这样做好处是可以将工作区移动到任何其他磁盘和目录，依然可以使用，甚至我可以将工作区的一个子目录拿出来，作为独立的工作区。

**Git:** 我也可以将工作区移动到其他磁盘，但是要保证工作区下的`:file:`.git``目录和工作区一同移动。也不可以只移动工作区下的一个目录到其他磁盘或目录，那样的话移出的目录就不能工作了。

**Git:** 我的网络传输效率很高。在和其他版本库交互时，对方会告诉我他有什么，我也知道我有什么，因为只对缺失对象的打包传输，所以效率很高而且能够显示传输进度。

**CVS:** 这一点我不行。因为我本地没有文件做对照，所以我在传输的时候不可能做到增量传输。

**Git:** 我甚至可以不需要网络，因为我在本地拥有完整的版本库，几乎所有操作都是在本地完成。

**CVS:** 我的操作处处需要网络，如果版本库是在网络中其他服务器上的话。如果网速比较慢，查看日志、查看历史版本都需要花费很长时间等待。

**CVS:** 你怎么没有更新（update）命令？还有你为什么老是要执行检出命令（checkout）？对我而言，检出命令只在工作区创建时一次完成的。

**Git:** 你的检出命令（checkout）是从远程版本库服务器获取数据完成本地工作区的创建，版本库仍然位于远程的服务器上。你的更新（update）命令执行的很慢对么？之所以你需要执行更新命令是因为你的版本库在远程啊。别忘了我的版本库是在本地，我的每一步操作工作区和版本库都是同步的，所以更新操作就没有存在的必要

了。而我的检出（checkout）操作是将本地版本库的数据检出到本地工作区，用于恢复本地丢失的文件或错误改动的文件，也用于切换不同的分支。我也有一个和你的更新（update）操作类似的比较耗时的网络操作命令叫做：command:`git fetch` 或：command:`git pull`，这两个操作是从别人的版本库获取他人改动。一般使用我（Git）做团队协作的时候，会部署一个集中共享的版本库，我就用这两个命令（command:`git fetch` 或 command:`git pull`）从共享的版本库执行拉回操作。也许你（CVS）会觉得 command:`git fetch` 或者 command:`git pull` 和你的 command:`cvs update` 命令更像吧。至于你的检出命令（command:`cvs checkout`），实际上和我克隆命令（command:`git clone`）很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

CVS：为什么你的检入命令（commit）命令执行的那么快？

Git：是的，我的检入命令飞一般就执行完了，也是因为版本库就在本地。也许你（CVS）会觉得我的推送命令（command:`git push`）和你的检入命令（command:`cvs commit`）更相像，其实这是一个误会。如果我不做本地提交，是不能通过推送命令（command:`git push`）将我的本地的提交共享给（推送给）其他版本库的。你（CVS）每一次提交都要和版本库进行网络通讯，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次 command:`git push`，将我本地版本库中新的提交推送给远程版本库。

CVS：我每一个文件都一个独立的版本号，你有么？

Git：每一个文件一个版本号？这有什么值得夸耀的？我听说你最早是用脚本对RCS系统进行封装实现的，所以你每个文件都有一个独立的版本控制，这让你变得很零碎。我听说某些商业版本控制系统也是这样，真糟糕。我的每一次提交都有一个全球唯一的版本号，这样不但在本地版本库中是唯一的，和其他人的版本库也不会有冲突。

CVS：我能一次检出一个目录，你好像不能吧？

Git：所以我有子模组，以及repo等第三方工具，可以帮助我把一个大的版本库拆开多个版本库组合来使用啊。

CVS：我能添加空目录，你好像不能吧！

Git：是的，我现在还不能记录空目录。但是用户可以在空目录下创建一个隐含文件，并将该隐含文件添加到版本库中，也就实现了空目录添加的功能。你，CVS，目录管理是你的软肋，你很难实现目录的重命名，而目录重命名对我来说是小菜一碟。

## CVS和Git命令对照

比较项目	CVS命令	Git命令
URL	:pserver:user@host:/path/to/cvsroot	git://host/path/to/repos.git
	/path/to/cvsroot	ssh://user@host/path/to/repos.git
		user@host:path/to/repos.git
		file:///path/to/repos.git
		/path/to/repos.git
版本库初始化	cvs -d <path> init	git init [--bare] <path>
导入数据	cvs -d <url> import -m ...	git clone; git add .; git commit
版本库检出	cvs -d <url> checkout [-d <path>]<module>	git clone <url> <path>

版本库分支检出	cvs -d <url> checkout -r <branch> <module>	git clone -b <branch> <url>
工作区更新	cvs update	git pull
更新至历史版本	cvs update -r <rev>	git checkout <commit>
更新到指定日期	cvs update -D <date>	git checkout HEAD@' {<date>}'
更新至最新提交	cvs update -A	git checkout master
切换至里程碑	cvs update -r <tag>	git checkout <tag>
切换至分支	cvs update -r <branch>	git checkout <branch>
还原文件/强制覆盖	cvs up -C <path>	git checkout -- <path>
添加文件	cvs add <TextFile>	git add <TextFile>
添加文件(二进制)	cvs add -kb <BinaryFile>	git add <BinaryFile>
删除文件	cvs remove -f <path>	git rm <path>
移动文件	mv <old> <new>; cvs rm <old>; cvs add <new>	git mv <old> <new>
反删除文件	cvs add <path>	git add <path>
工作区差异比较	cvs diff -u	git diff
		git diff --cached
		git diff HEAD
版本间差异比较	cvs diff -u -r <rev1> -r <rev2> <path>	git diff <commit1> <commit2> -- <path>
查看工作区状态	cvs -n up	git status
提交	cvs commit -m "<msg>"	git commit -a -m "<msg>" ; git push
显示提交日志	cvs log <path>   less	git log
逐行追溯	cvs annotate	git blame
显示里程碑/分支	cvs status -v	git tag
		git branch
		git show-ref
创建里程碑	cvs tag [-r <rev>] <tagnname> .	git tag [-m "<msg>"] <tagnname> [<commit>]
删除里程碑	cvs rtag -d <tagnname>	git tag -d <tagnname>
创建分支	cvs rtag -b -r <rev> -b <branch> <module>	git branch <branch> <commit>
		git checkout -b <branch> <commit>
删除分支	cvs rtag -d <branch>	git branch -d <branch>
导出项目文件	cvs -d <url> export -r <tag> <module>	git archive -o <output.tar> <tag> <path>
		git archive -o <output.tar> --remote=<url> <tag> <path>
分支合并	cvs update [-j <start>] -j <end>; cvs commit	git merge <branch>

显示文件列表	cvs ls	git ls-files
	cvs -d <url> rls -r <rev>	git ls-tree <commit>
更改提交说明	cvs admin -m <rev>:<msg> <path>	git commit --amend
撤消提交	cvs admin -o <range> <path>	git reset [ --soft   --hard ] HEAD^
杂项	.cvsignore 文件	.gitignore 文件
	参数 -kb 设置二进制模式	-text 属性
	参数 -kv 开启关键字扩展	export-subst 属性

# Git和SVN面对面

## 面对面访谈录

**Git:** 我的提交历史本身就是一幅美丽的图画——DAG (Directed Acyclic Graph, 有向非环图)，可以看到各个分支之间的合并关系。而你SVN，你的提交历史怎么是一条直线呢？要是在重症监护室看到你，还以为你挂掉了呢？

**SVN:** 我觉得挺好，至少我每次提交会有一个全局的版本号，而且我的版本号是递增的。你的版本号不是递增的吧！

**Git:** 你说的对，我的版本号不是一个简单递增的数字，而是一个长达40位的十六进制数字（哈希值），但是可以使用短格式，只要不冲突。虽然我的提交编号看起来似乎是无序的，但实际上我每一个提交都记录了父提交甚至是双亲或多亲提交，因此可以很容易的从任意一个提交开始建立一条指向历史提交的跟踪链。

**SVN:** 是啊，我的一个提交和前一个提交有时根本没有关系，例如一个提交是发生在主线:`/trunk`中的，下一个提交可能就发生在`/branches/1.3.x`分支中。你要知道要想画出一个像你那样的分支图，我要做多少工作么？我不容易呀。

**Git:** 我一直很奇怪，你的分支和里程碑怎么看起来和目录一样？我的分支和里程碑名字虽然看起来像是目录，实际上和工作区的目录完全没有关系，只是对提交ID的一个记号而已。

**SVN:** 我一开始觉得我用轻量级拷贝的方式实现分支和里程碑会很酷，也很快。但是我发现很多人在使用我的时候，直接在版本库的根目录下创建文件而不是把文件创建在`/trunk`目录下，这就导致这些人无法再创建分支和里程碑了，因为是无法将根目录拷贝到子目录的呀！

**Git:** 那么你是如何对分支合并进行跟踪的呢？我因为有了DAG的提交关系图，是很容易看出来分支之间的合并历史，但是你是怎么做到的呢？

**SVN:** 我用了一点小技巧，就是通过属性(`svn:mergeinfo`)记录了合并的分支名和版本范围，这样再合并的时候，我会根据相关属性确定是否要合并。但是如果经常在子目录下合并，有太多的`svn:mergeinfo`属性等待我检查，我会很困扰。还有我的这个功能是在1.5以后版本才提供的，因此老版本会破坏这个机制。

**SVN:** 对了，我的属性能干很多事哦，我甚至可以把我的照片作为属性附加在文件上。

**Git:** 这点我承认，你的属性非常强大。其实我也支持属性，只不过实现方式不同罢了。而且我可以通过评注的方式为任意对象（提交、文件、里程碑等）添加评注，也可以实现把照片做为评注附加在文件上，可是这个功能有什么实际用处么？

**SVN:** 我有轻量级拷贝，而我的分支和里程碑就是通过拷贝实现的，很强大哦。

**Git:** 我根本就不需要轻量级拷贝，因为我对文件的保存是和文件路径无关的，我只关心内容。所以相同内容的文件无论它们的文件名相差有多大，在我这里只保存一份。而你SVN，如果用户忘了用轻量级拷贝，版本库是不是负担很重啊。

**SVN:** 听说你不能针对目录授权，这可是我的强项，所以公司无论大小都在用我作为版本控制系统。

**Git:** 不要说你的授权了，简直是一团糟。虽然这本书的作者为你写了一个图形化的授权管理工具(<http://www.osxp.com/doc/pysvnmanager/user-guide/user-guide.html>)会有所改善，但是你糟糕的分支和里程碑的实现，会导致授权在新的分支和里程碑中要一一设置，工作量其大无比。虽然泛路径授权是一个解决方案，但是官方并没有提供啊。

Git: 说说我的授权吧。如果你认真的读过本书服务器架设的相关章节，你会为我能够提供的按照分支，以及按照路径进行写操作授权而击掌叫好的。当然我的读操作授权还不能做到很精细，但是可以将版本库拆分成若干小的版本库啊，再用参照本书介绍的各种多版本库协同模式，会找到一个适合的解决方案的啊。

Git: 我的工作区很干净。只在工作区的根目录下有一个:file:`.git`目录，此外再无其他。

SVN: 我要在工作区的每一个目录下都放置一个:file:`.svn`目录，这个目录在Linux下可是隐藏的哦。这个目录下不但有跟踪工作区文件状态的跟踪文件，而且还有每一个文件的原始拷贝呢。这样有的操作就可以脱离网络执行了，例如：差异比较，工作区文件的回滚。

Git: 嗯，你要是像我一样再保存多一点内容（整个版本库）就更好了。像你这样在每个工作区子目录下都有一个:file:`.svn`目录，而且每个:file:`.svn`目录下都有文件原始拷贝，在进行内容搜索的时候会搜索出来两份吧，太干扰了。而且你这么做和CVS一样有安全风险，造成本地文件名的信息泄漏，千万不要在Web服务器上用SVN检出哦。

Git: 我的操作可以不需要网络。因为我在本地拥有完整的版本库，几乎所有操作都是在本地完成。

SVN: 正如前面说到的，我有部分命令可以不需要网络，但是其他绝大多数命令还是要依赖网络的。

SVN: 你怎么没有更新(update)命令？还有你为什么老是要执行检出命令(checkout)？对我而言，检出命令只在工作区创建时一次完成的。

Git: 你的这个问题怎么和CVS问的一样。你的更新(update)命令执行的很慢对么？首先你要用检出命令(checkout)建立工作区，然后你要经常的执行更新(update)命令进行更新，否则容易造成你的更改和他人更改发生冲突。

Git: 之所以你需要更新是因为你的版本库在远程啊。别忘了我的版本库是在本地，我的每一步操作工作区和版本库都是同步的，所以更新操作就没有存在的必要了。而我的检出(checkout)操作一般是用户切换分支，或者从本地版本库检出丢失的文件或覆盖本地错误改动的文件时用到。如果我没记错的话，你切换分支用的是:command:`svn switch`命令对么？

Git: 实际上我也有一个比较耗时的网络操作命令叫做:command:`git fetch`或:command:`git pull`，这两个操作是从远程版本库获取他人改动。一般使用我(Git)做团队协作的时候，会部署一个集中共享的版本库，我就从这个共享的版本库执行拉回操作。也许你(SVN)会觉得:command:`git fetch`或:command:`git pull`和你的:command:`svn update`命令更像吧。至于你的检出命令(:command:`svn checkout`)，实际上和我克隆命令(:command:`git clone`)很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

SVN: 为什么你的检入命令(commit)命令执行的那么快？

Git: 是的，我的检入命令飞一般就执行完了，也是因为版本库就在本地。也许你(SVN)会觉得我的推送命令(:command:`git push`)和你的检入命令(:command:`svn commit`)更相像，其实这是一个误会。如果我不做本地提交，是不能通过推送命令(:command:`git push`)将我的本地的提交共享给(推送给)其他版本库的。你(SVN)每一次提交都要和版本库进行网络通讯，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次:command:`git push`，将我本地版本库中新的提交推送给远程版本库。

SVN: 我能一次检出一个目录，你好像不能吧？

**Git:** 所以我有子模组，以及repo等第三方工具，可以帮助我把一个大的版本库拆开  
多个版本库组合来使用啊。

**SVN:** 我能添加空目录，你好像不能吧！

**Git:** 是的，我现在还不能记录空目录，但是用户往往在空目录下创建一个隐含文件，并将该隐含文件添加到版本库中，也就实现了空目录添加的功能。

## SVN和Git命令对照

比较项目	SVN命令	Git命令
URL	svn://host/path/to/repos	git://host/path/to/repos.git
	https://host/path/to/repos	ssh://user@host/path/to/repos.git
	file:///path/to/repos	user@host:path/to/repos.git
		file:///path/to/repos.git
		/path/to/repos.git
版本库初始化	svnadmin create <path>	git init [--bare] <path>
导入数据	svn import <path> <url> -m ...	git clone; git add .; git commit
版本库检出	svn checkout <url/of/trunk> <path>	git clone <url> <path>
版本库分支检出	svn checkout <url/of/branches/name> <path>	git clone -b <branch> <url> <path>
工作区更新	svn update	git pull
更新至历史版本	svn update -r <rev>	git checkout <commit>
更新到指定日期	svn update -r {<date>}	git checkout HEAD@'{<date>}'
更新至最新提交	svn update -r HEAD	git checkout master
切换至里程碑	svn switch <url/of/tags/name>	git checkout <tag>
切换至分支	svn switch <url/of/branches/name>	git checkout <branch>
还原文件/强制覆盖	svn revert <path>	git checkout -- <path>
添加文件	svn add <path>	git add <path>
删除文件	svn rm <path>	git rm <path>
移动文件	svn mv <old> <new>	git mv <old> <new>
清除未跟踪文件	svn status   sed -e "s/^?//;"   xargs rm	git clean
清除工作锁定	svn clean	-
获取文件历史版本	svn cat -r<rev> <url/of/file>@<rev> > <output>	git show <commit>:<path> > <output>
反删除文件	svn cp -r<rev> <url/of/file>@<rev> <path>	git add <path>
工作区差异比较	svn diff	git diff
		git diff --cached
		git diff HEAD

版本间差异比较	<code>svn diff -r &lt;rev1&gt;:&lt;rev2&gt; &lt;path&gt;</code>	<code>git diff &lt;commit1&gt; &lt;commit2&gt; -- &lt;path&gt;</code>
查看工作区状态	<code>svn status</code>	<code>git status -s</code>
提交	<code>svn commit -m "&lt;msg&gt;"</code>	<code>git commit -a -m "&lt;msg&gt;" ; git push</code>
显示提交日志	<code>svn log   less</code>	<code>git log</code>
逐行追溯	<code>svn blame</code>	<code>git blame</code>
显示里程碑/分支	<code>svn ls &lt;url/of/tags/&gt;</code>	<code>git tag</code>
	<code>svn ls &lt;url/of/branches/&gt;</code>	<code>git branch</code>
		<code>git show-ref</code>
创建里程碑	<code>svn cp &lt;url/of/trunk/&gt; &lt;url/of/tags/name&gt;</code>	<code>git tag [-m "&lt;msg&gt;"] &lt;tagname&gt; [&lt;commit&gt;]</code>
删除里程碑	<code>svn rm &lt;url/of/tags/name&gt;</code>	<code>git tag -d &lt;tagname&gt;</code>
创建分支	<code>svn cp &lt;url/of/trunk/&gt; &lt;url/of/branches/name&gt;</code>	<code>git branch &lt;branch&gt; &lt;commit&gt;</code>
		<code>git checkout -b &lt;branch&gt; &lt;commit&gt;</code>
删除分支	<code>svn rm &lt;url/of/branches/name&gt;</code>	<code>git branch -d &lt;branch&gt;</code>
导出项目文件	<code>svn export -r &lt;rev&gt; &lt;path&gt; &lt;output/path&gt;</code>	<code>git archive -o &lt;output.tar&gt; &lt;commit&gt;</code>
	<code>svn export -r &lt;rev&gt; &lt;url&gt; &lt;output/path&gt;</code>	<code>git archive -o &lt;output.tar&gt; --remote=&lt;url&gt; &lt;commit&gt;</code>
反转提交	<code>svn merge -c -&lt;rev&gt;</code>	<code>git revert &lt;commit&gt;</code>
提交拣选	<code>svn merge -c &lt;rev&gt;</code>	<code>git cherry-pick &lt;commit&gt;</code>
分支合并	<code>svn merge &lt;url/of/branch&gt;</code>	<code>git merge &lt;branch&gt;</code>
冲突解决	<code>svn resolve --accept=&lt;ARG&gt; &lt;path&gt;</code>	<code>git mergetool</code>
	<code>svn resolved &lt;path&gt;</code>	<code>git add &lt;path&gt;</code>
显示文件列表	<code>svn ls</code>	<code>git ls-files</code>
	<code>svn ls &lt;url&gt; -r &lt;rev&gt;</code>	<code>git ls-tree &lt;commit&gt;</code>
更改提交说明	<code>svn ps --revprop -r&lt;rev&gt; svn:log "&lt;msg&gt;"</code>	<code>git commit --amend</code>
撤消提交	<code>svnadmin dump、svnadmin load 及 svndumpfilter</code>	<code>git reset [ --soft   --hard ] HEAD^</code>
属性	<code>svn:ignore</code>	<code>.gitignore</code> 文件
	<code>svn:mime-type</code>	<code>text</code> 属性
	<code>svn:eol-style</code>	<code>eol</code> 属性
	<code>svn:externals</code>	<code>git submodule</code> 命令
	<code>svn:keywords</code>	<code>export-subst</code> 属性

# Git和Hg面对面

## 面对面访谈录

Git: 你好Hg, 我发现我们真的很像。

Hg: 是啊, 人们把我们都归类为分布式版本控制工具, 所以我们之间的相似度, 要比和CVS、SVN的相似度高的多了。

Hg: 我是用Python和少部分的C语言实现的, 你呢?

Git: 我的核心当然使用C语言了, 因为Linus Torvalds最爱用C语言了。我的很多命令还使用了Shell脚本和Perl语言开发, Python用的很少。

Hg: 大量的使用C语言, 是你的性能比我高的原因么?

Git: 当然不是了, 你不也在核心模块使用C语言了么? 问题的关键在于我的对象库设计的非常优秀。你不要忘了我是谁发明的, 可是大名鼎鼎的Linux之父Linus Tolvars, 他对Linux文件系统可是再熟悉不过的了, 所以他能够以文件系统开发者的视角实现我的核心。

Git: 还有我在网络传输过程中非常的直观, 可以看到实时的进度显示, 这也是你所没有的。

Hg: 是啊, 非常的惭愧。当克隆一个比较大的Hg版本库时, 会出现假死状态, 用户不知道克隆操作的进展。对了, 你为什么能够实现实时的进度显示呢?

Git: 之所以我能够有这样的实现, 是因为我使用了“智能协议”。在网络传输的各自一端都启用了相应的辅助程序, 实现差异传输及传输进度的计算和显示。

Hg: 我有一个特点是SVN用户非常喜欢的, 就是我的顺序数字版本号。

Git: 你的顺序数字版本号只是在本地版本库中有效的。也就是说, 你不能像SVN那样将顺序数字版本号作为项目本身的版本号, 因为换做另外的一个版本库克隆, 那个数字版本号就会不一样了。

Hg: 我觉得你的暂存区(stage)的概念太古怪了。我提交的时候, 改动的文件会直接提交而不需要什么注册到暂存区的操作。

Git: 让读者来作评判吧。如果读者读过本书的第2篇, 一定会说Git的暂存区帅呆了。

Hg: 我只允许用户对最近的一次提交进行回滚撤销, 而你(Git)怎么能允许用户撤销任意多次历史提交呢? 那样安全么?

Git: 这就是的我对象库和引用设计的强大之处, 我可以使用:command:`git reset`命令将工作分支进行任意的重置, 丢弃任意多的历史。至于安全性, 我的重置命令有一个保险, 就是reflog, 我随时可以参照reflog的记录来弥补错误的重置。

Hg: 我们的:command:`revert`命令好像不同?

Git: 你Hg的:command:`hg revert`命令和SVN的:command:`svn revert`命令相似, 是取消本地修改, 用原始拷贝覆盖。你的这个操作在我这里是用:command:`git checkout`命令实现的。我也有一个:command:`git revert`命令, 但是这个命令是针对某个历史提交进行反向操作, 以取消该历史提交的改动的。

Hg: 我执行日志查看能够看到文本显示的分支图, 你呢?

Git: 我需要在日志显示时添加参数, 即用命令:command:`git log --graph`。我

支持通过建立别名实现简洁的调用，例如建立一个名为glog的别名。

**Git:** 我听说你Hg不支持分支？

**Hg:** 坦白的说，是的。我虽然也有分支命令，但是分支不是一个独立显示的提交空间，而是各个分支都显示在一起，相当于在版本库中同时拥有多个头，选择哪个分支就相当于把帽子戴在哪个头上面而已。所以我尽量要求我的用户使用克隆来当做分支。

**Git:** 这也就是为什么使用Hg作为项目的版本控制工具，要为每一个分支创建一个单独的版本库的原因吧。实际上我的每一个克隆的版本库也相当于独立的分支，但是因为我有强大的分支功能，因此很多用户还没有意识到。使用Topgit的用户就应该使用版本库克隆做为Topgit本身的分支管理。

**Git:** 还有，因为我对分支的完整支持，使得我可以和SVN很好的协同工作。我可以将整个SVN转换为本地的Git库，但是你Hg，显然只能每次转换一个分支。

**Hg:** 是的，我要向你多学习。

## Hg和Git命令对照

比较项目	Hg命令	Git命令
URL	<a href="http://host/path/to/repos">http://host/path/to/repos</a>	git://host/path/to/repos.git
	ssh://user@host/path/to/repos	ssh://user@host/path/to/repos.git
	file:///path/to/repos	user@host:/path/to/repos.git
	/path/to/repos	file:///path/to/repos.git
		/path/to/repos.git
配置	[ui] username = Firstname Lastname <mail@addr>	[user] name = Firstname Lastname email = mail@addr
版本库初始化	hg init <path>	git init [--bare] <path>
版本库克隆	hg clone <url> <path>	git clone <url> <path>
获取版本库更新	hg pull --update	git pull
更新至历史版本	hg update -r <rev>	git checkout <commit>
更新到指定日期	hg update -d <date>	git checkout HEAD@'<date>'
更新至最新提交	hg update	git checkout master
切换至里程碑	hg update -r <tag>	git checkout <tag>
切换至分支	hg update -r <branch>	git checkout <branch>
还原文件/强制覆盖	hg update -C <path>	git checkout -- <path>
添加文件	hg add <path>	git add <path>
删除文件	hg rm <path>	git rm <path>
添加及删除文件	hg addremove	git add -A
移动文件	hg mv <old> <new>	git mv <old> <new>
撤消添加、删除等操作	hg revert <path>	git reset -- <path>

清除未跟踪文件	hg clean	git clean -fd
获取文件历史版本	hg cat -r<rev> <path> > <output>	git show <commit>:<path> > <output>
反删除文件	hg add <path>	git add <path>
工作区差异比较	hg diff	git diff
		git diff --cached
		git diff HEAD
版本间差异比较	hg diff -r <rev1> -r <rev2> <path>	git diff <commit1> <commit2> -- <path>
查看工作区状态	hg status	git status -s
提交	hg commit -m "<msg>"	git commit -a -m "<msg>"
推送提交	hg push	git push
显示提交日志	hg log   less	git log
	hg glog   less	git log --graph
逐行追溯	hg annotate	git annotate, git blame
显示里程碑/分支	hg tags	git tag
	hg branches	git branch
	hg heads	git show-ref
创建里程碑	hg tag [-m "<msg>"] [-r <rev>] <tagname>	git tag [-m "<msg>"] <tagname> [<commit>]
删除里程碑	hg tag --remove <tagname>	git tag -d <tagname>
创建分支	hg branch <branch>	git branch <branch> <commit>
		git checkout -b <branch> <commit>
删除分支	hg commit --close-branch	git branch -d <branch>
导出项目文件	hg archive -r <rev> <output.tar.gz>	git archive -o <output.tar> <commit>
		git archive -o <output.tar> --remote=<url> <commit>
反转提交	hg backout <rev>	git revert <commit>
提交拣选	-	git cherry-pick <commit>
分支合并	hg merge <rev>	git merge <commit>
变基	hg rebase	git rebase
冲突解决	hg resolve --tool=<tool>	git mergetool
	hg resolve -m <path>	git add <path>
更改提交说明	Hg + MQ	git commit --amend
撤消最后一次提交	hg rollback	git reset [ --soft   --hard ] HEAD^
撤消多次提交	Hg + MQ	git reset [ --soft   --hard ] HEAD^<n>
撤消历史提交	Hg + MQ	git rebase -i <commit>^
启动Web浏览	hg serve	git instaweb
二分查找	hg bisect	git bisect

内容搜索	hg grep	git grep
提交导出补丁文件	hg export	git format-patch
工作区根目录	hg root	git rev-parse --show-toplevel
杂项	.hgignore 文件	.gitignore 文件
	pager 扩展	内置分页器
	color 扩展	color.* 配置变量
	mq 扩展	StGit, Topgit
	graphlog 扩展	git log --graph
	hgk 扩展	gitk

GotGit 是我在2010年底写的一本书《Git权威指南》的代号，五年的时间过去了，Git 走进了 2.x 版本的时代。短短几年 Git 爆发式的发展已经让这本书显得过时了。

- GitHub 引领了社区式编程的潮流，成为开源社区的代名词。我曾经应邀写了一个分析 GitHub 的电子书 《[GotGitHub](#)》，但是很难跟上 GitHub 的变化的节奏。
- Gitlab 开源项目诞生于 2011 年，从模仿 GitHub，到被别人拿来照搬，让各种社交式编程社区在中国四处开花。
- Gerrit 和 Gitlab 是公司构建企业内部 Git 仓库托管平台的主要选择，如何构建高可用、可扩展的平台成为很多大公司探索的课题。
- Git 曾经一度遇到外热内冷（外部开源社区热，企业内部遇冷）的情况。在2011到2013年我的主要客户还是使用 SVN，甚至我的 80% 时间用在 某些企业内部 SVN 平台的定制开发上。不过最近两年又几乎掉了一个个儿。但是在企业中推广 Git 还是很有挑战，一是 Windows 平台上的工具体验不佳，再一个是大的 Git 仓库（10GB以上）导致的用户体验差。业界出现了一些解决方案，例如 Git-LFS，在2015年底，我和我的小伙伴们为企业 内部的 Git 平台加入了 Git-LFS 功能。我也在思考和尝试更好的方案。
- Git 进入 2.x 之后，大的功能上的改变很少，不过小的改变很多，如果照着《Git权威指南》照搬，很多地方会遇到困难。
- 写书之前，我给 Git 只贡献了一两个提交，而这几年，我向Git社区贡献了近两百个提交，其中有一些是有点技术含量的新功能、Bugfix，也有相当多的中文本地化。2012年我成为 Git 社区本地化的负责人，关于Git社区的工作流程理解得也更为透彻。

早在三年前，出版社又开始和我约稿，包括热心网友的询问，我一直以 Git 2.0 尚未发布为挡箭牌。但是真的等到 Git 2.0 发布，我发现再也找不到大块儿的空余时间来专心写作了。这也就是为什么这本书现在已经很难在正规渠道买到了，那些拿到我亲笔签名的亲们，你们赚到了。 ;)

2015年底，在为华为做咨询顾问一年多之后，我决定接受新的挑战，加入[华为公司](#)，成为这个拥有着最大的开发者群体的世界级公司的一员。如何有效地在大公司内进行过知识的传递也是一个课题，我决定将这本书的书稿开源，惠及更多的开发者。

书稿由 reStructuredText 格式书写，要想在本地编译和浏览，需要安装相关工具链。我已经将相关工具链做成 Docker 镜像，使用方法参见 [jiangxin/docker-gotgit](#) 项目。