开源分布式版本控制工具 —— Git 之

皮 光明 和 谢 炯
2013 年 7 月 22 日发布

背景

Git 是一个开源的分布式版本控制软件。在英式英语中，Git 指一个愚笨或者不开心的人，恐怕 Torvalds 当时的自嘲心理不无关系吧。2002 年之前，Linux 内核维护工作的绝大部分时间都浪 务上。启用版本控制工具 BitKeeper 管理 Linux 内核成了当务之急。不过，BitKeeper 毕竟是一 使用之后，Linux 社区不得不寻求它的替代品，以便继续托管 Linux 内核源代码。2005 年，迫 了一套开源版本控制工具，并命名为 Git。

自诞生以来，Git 就以其开源、简单、快捷、分布式、高效等特点，应付了类似 Linux 内核源代 今，Git 已经非常成熟，被广泛接受与使用，越来越多的项目都迁移到 Git 仓库中进行管理。以 80% 的 Eclipse 基金会项目已经完全使用 Git 管理，CVS 访问权限已经切换成只读状态。并且， 项目管理的介绍中已将"CVS"三个字符划掉，而且很萌地写道，"Ding dong, the witch is dead." 挂了"。

不仅如此，笔者最近也收到了全球最大开源代码托管平台——SourceForge 的升级通知。其中， CVS 被系统默认自动升级到了 Git。而对于另一个较为复杂的 CVS 项目Toolbox for Java/JTOpe 级，估计是等待笔者做升级前的最后准备工作。笔者希望通过分享自己的 Git 学习体验与实践 也是本文之意义所在。

为什么选择 Git

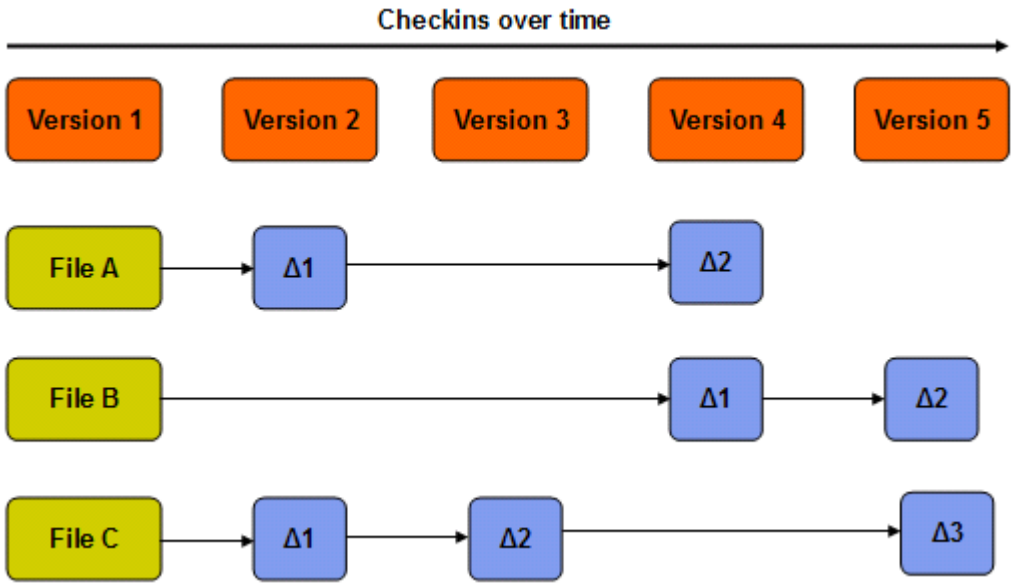
实际上，相对于 CVS、SVN 等主流版本控制软件，Git 的学习成本甚至会更高。比如，对于 Su 么是文件、工作目录、资源库、版本、分支和标签等概念，差不多就够用了。而对于 Git 用户，

括文件、快照、工作树、索引、本地资源库、远程资源库、远程、提交、分支和 Stash 等。那
趋之若鹜呢？相比于 CVS 与 SVN，Git 的优势到底体现在哪里？

关于 Git 的各种优势，互联网以及各种 Git 书籍都给出了自己的答案。笔者认为，存储快照与分
点，理由如下：

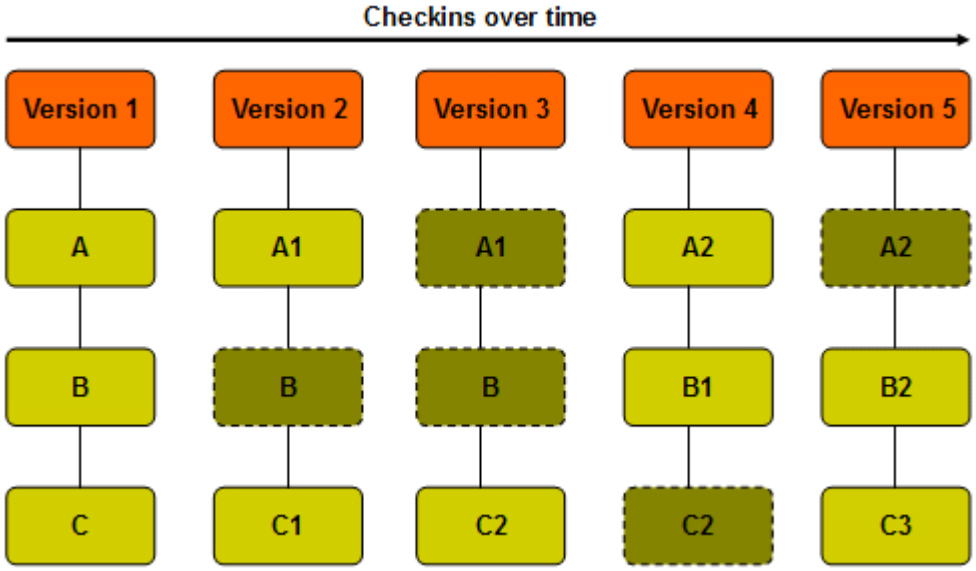
第一，Git 底层自行维护的存储文件系统是一大亮点。CVS、SVN 底层采用的为增量式文件系统
的特点是：当文件变动发生提交时，该文件系统存储的是文件的差异信息。

图 1. CVS、SVN 记录文件内容差异



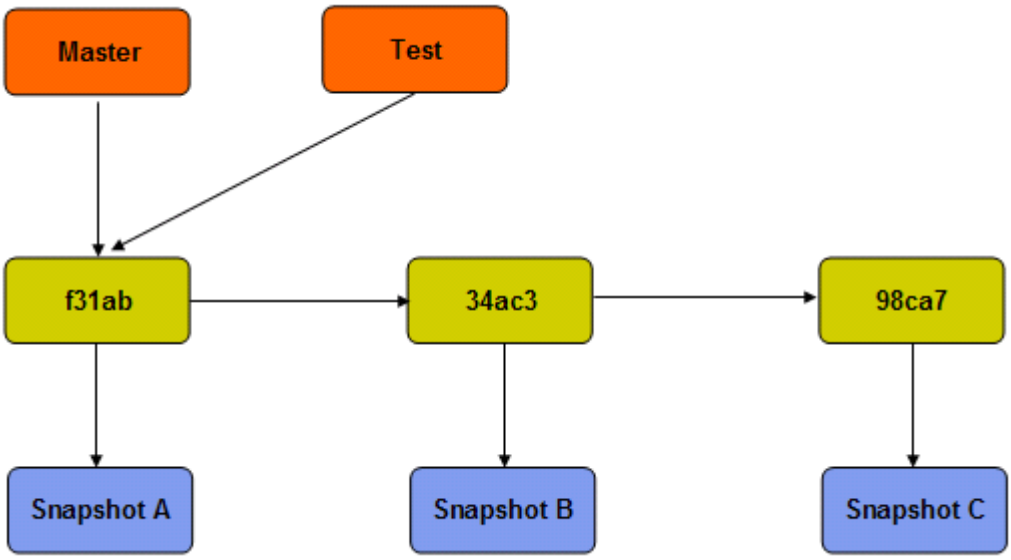
同样是文件变更提交，Git 底层文件系统存储的则为文件快照，即整个文件内容，并保存指向快
性能因素，如果文件内容没有发生任何变化，该文件系统则不会重复保存文件，只是简单地保

图 2. Git 记录整个文件快照



Git 之所以选择这样的底层存储数据结构，主要是为了提高 Git 分支的使用效率。实际上，Git 分
可变指针，而每一个索引对象又指向文件快照，如图 3 所示。

图 3. Git 分支对应的数据结构

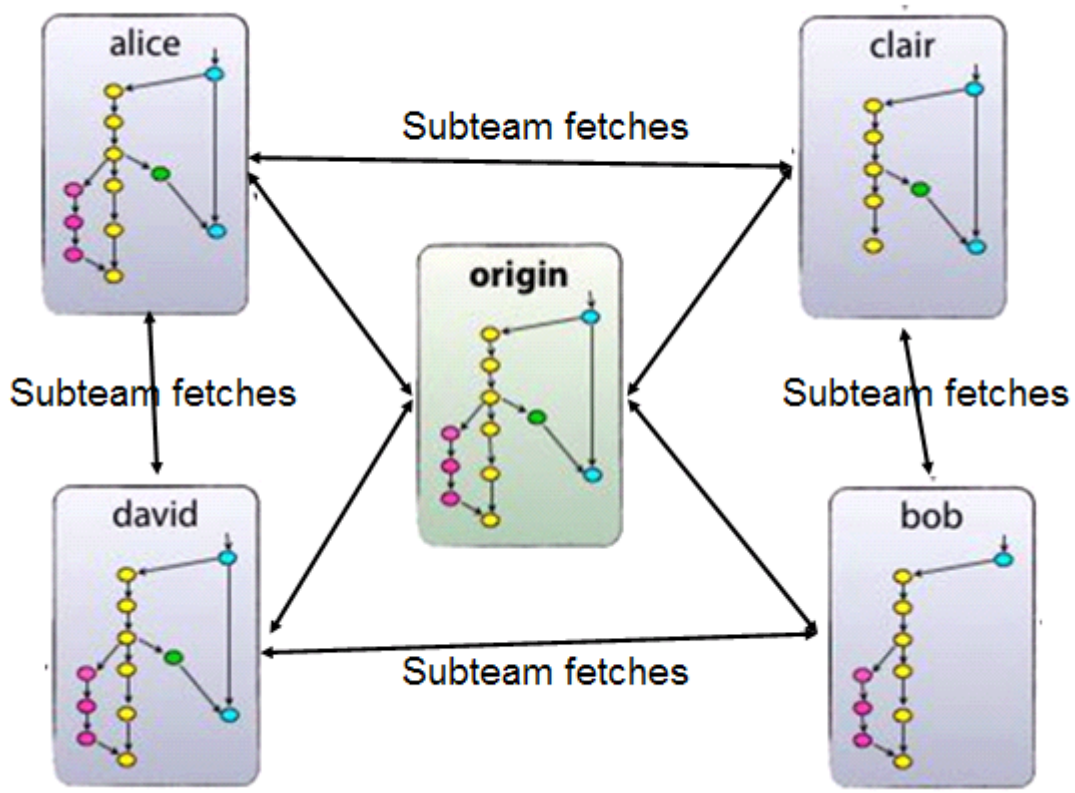


这样一来，创建分支可以瞬间完成，几乎不需要花费太多代价。换句话说，Git 分支是廉价的、
SVN 项目，分支通常意味着源代码的完整拷贝，其代价是昂贵的、重量级的。而对于大型项目
的，这与 Git 鼓励频繁创建与合并分支的理念相吻合。

第二，Git 版本控制系统的设计思想是"去中心化"。传统的 CVS 、SVN 等工具采用的是 C/S 架
服务器端。而一旦由于服务器系统宕机、网络不通等各种原因造成中心仓库不可用，整个 CVS
瘓了。即便考虑到高可用性，通过迁移另一个中心仓库继续代码提交操作，相应的运营维护成

为了摆脱对中心仓库的依赖，Git 的初始设计目标之一就是分布式控制管理。我们给出一个样例
个项目组，开发者主要由 Alice、Bob、Clair、David 四名成员组成。其中，除了中心仓库 origi
每一名成员各自负责一个本地仓库。从分布式的观点来看，David 可看成是 Alice 的远程仓库，
计理念有助于减少对中心仓库的依赖，从而有效降低中心仓库的负载，改善代码提交的灵活性

图 4. Git 分布式工作示意图



Git 分布式设计思想所带来的另外一大好处是支持离线工作。离线工作的好处不言而喻，对于 C C/S 工具而言，没有了网络或者 VPN ，就意味着失去了左膀右臂，代码检入与检出操作就无法 在没有 WIFI 的飞机或者火车上，照样可以频繁地提交代码，只不过先提交到本地仓库，等到 的镜像仓库。

有关 Git 更多详细信息，请参考 Git 官方网站：http://git-scm.com/。

工欲善其事，必先利其器。在理解 Git 灵活的快照存储与分布式设计理念之后，我们介绍 Git 针 要指出的是，这里仅仅粗线条地介绍 Git 的安装方法，至于 Git 安装前提条件、安装过程出现的 述，均不在本文的讨论范围。

如何安装 Git

总结起来，Git 安装方式通常分为两种：一种是选择 Git 源码编译安装；另一种使用针对特定平 Linux、Mac、Windows 等，其安装说明如下。

1. 源码编译安装

从 Git 源码安装至少可以保证版本是最新的。在安装 Git 之前，需要安装其依赖的软件包，包括 libiconv 等。根据不同类型的 Linux，读者可以选择不同的软件包安装工具，这里以 yum 为例，

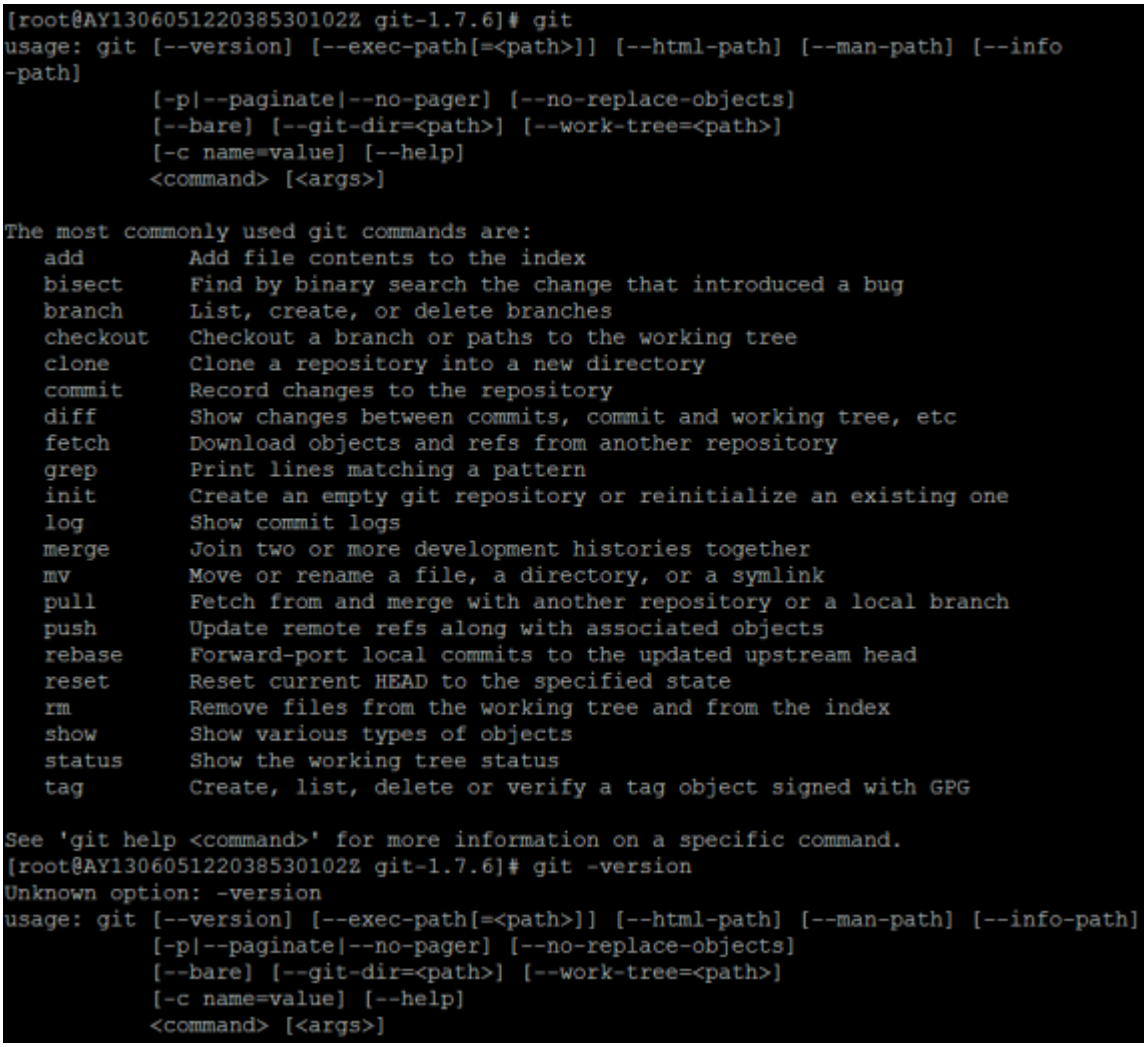
```
$ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel
```

接下来，读者可以从 Git 官方站点 http://git-scm.com/download 下载最新 Git 源代码（由于时 Git 为最新版本），执行以下命令编译安装。

```
$ tar -zxf git-1.7.6.tar.gz
$ cd git-1.7.6
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

最后，敲入 git 命令，检验安装是否成功，如图 5 所示。可以看到，我们已经成功安装 Git 了。

图 5. 通过源码安装 Git



2. 在 Linux 上安装

要在 Linux 上安装预编译好的 Git 二进制安装包，可选择系统支持的软件包管理器。对于红帽 L

```
$ yum install git-core
```

而对于 Ubuntu 这类 Debian 体系的 Linux 系统，使用 apt-get 命令安装：

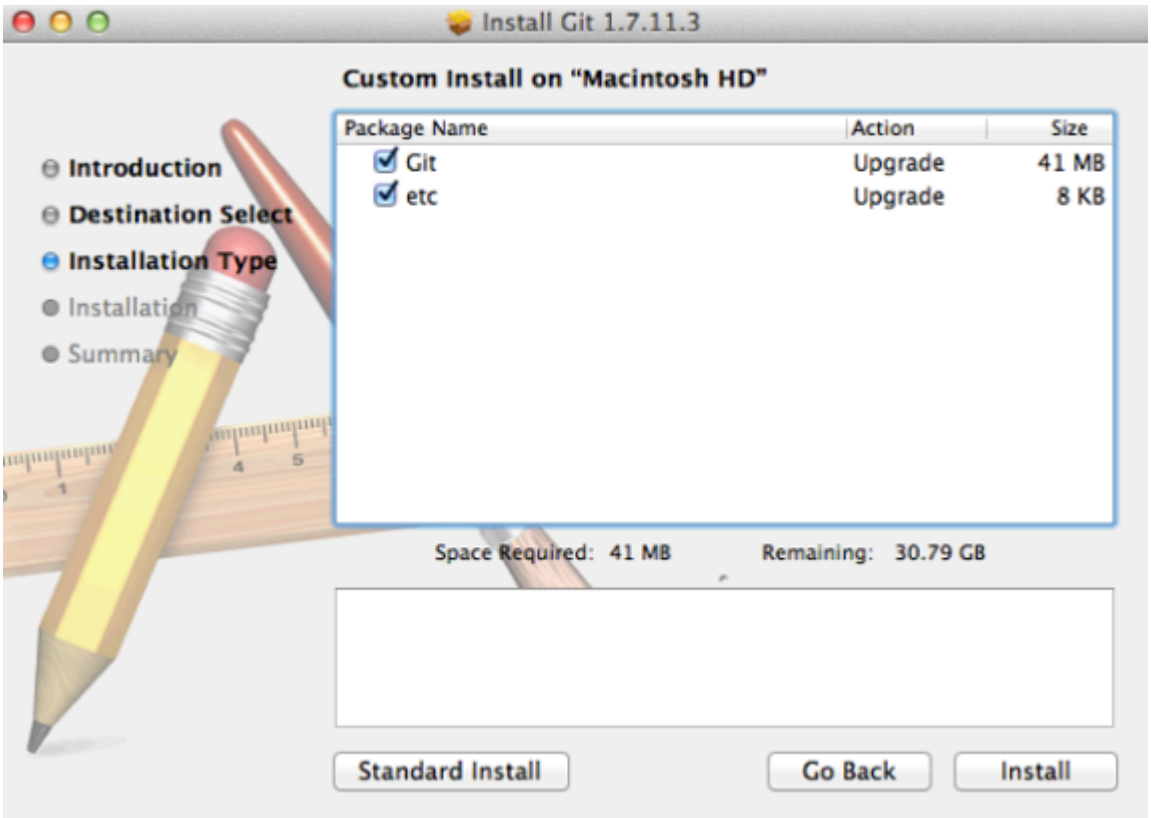
```
$ apt-get install git-core
```

由于此种安装方式非常简单，这里不做贴图展示了。

3. 在 Mac 上安装

Mac 系统支持 Git 安装的方式分为两种：编译安装与图形安装。其命令行式的编译安装与 Linux
之下，Mac 图形安装 Git 更加简单，其安装截图如图 6 所示。读者可去 http://code.google.com
Mac 系统的 Git 版本。

图 6. 从 Mac 上安装 Git

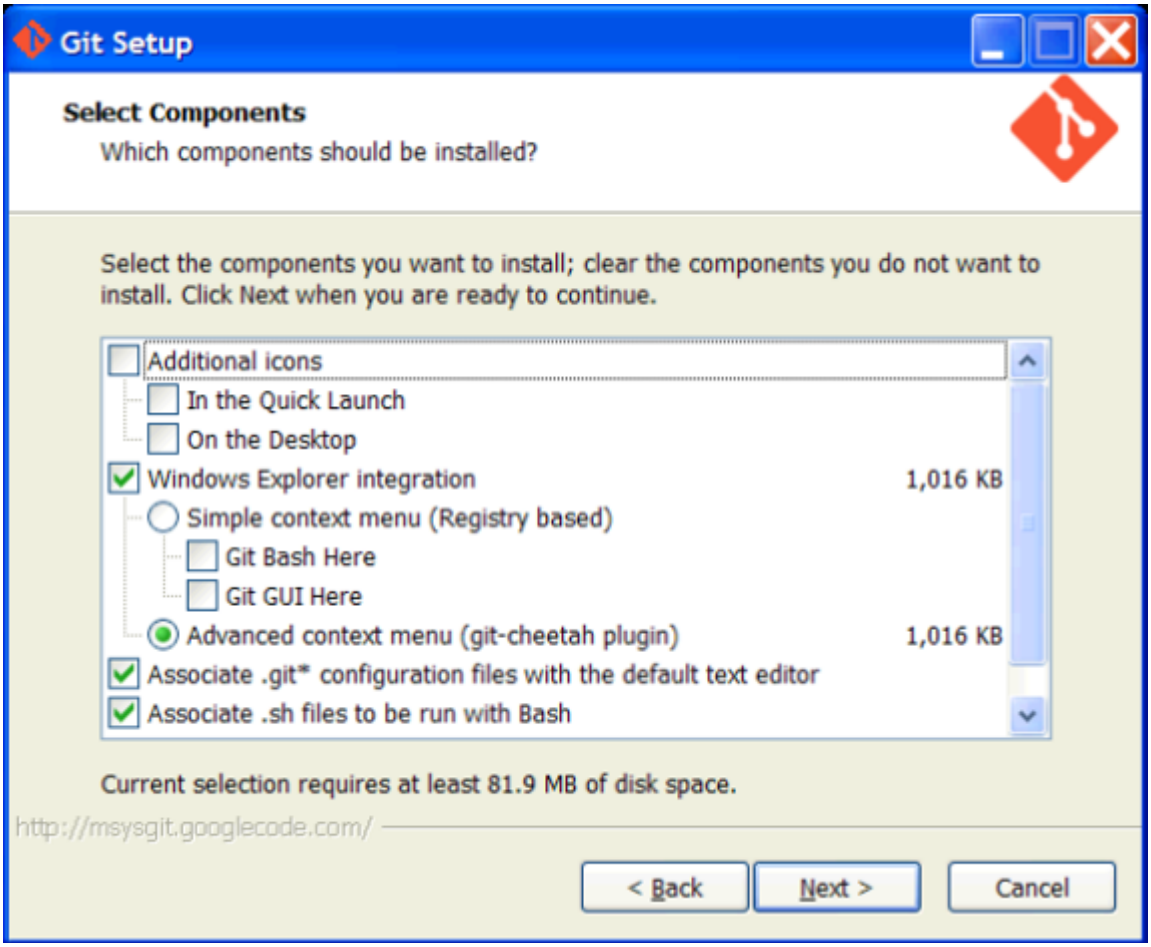


4. 在 Windows 上安装

与之前所述的 Mac 安装 Git 一样，在 Windows 上安装 Git 也同样轻松。根据用户的使用习惯，

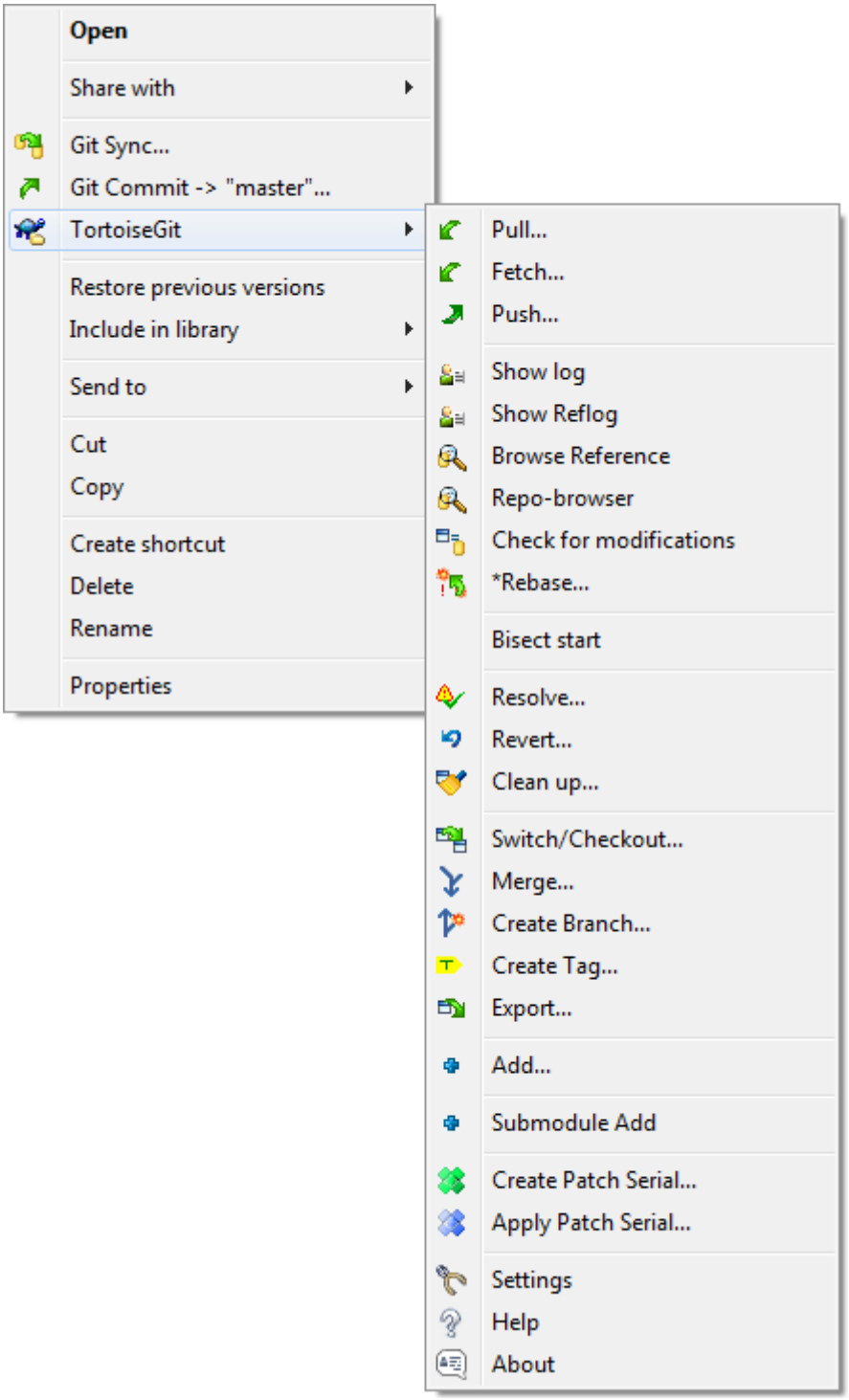
习惯命令行的用户，可选择 msysGit 安装包，安装截图如 7 所示。msysGit 的官方下载地址为 http://code.google.com/p/msysgit。

图 7. 从 Windows 上安装命令行 Git 工具——msysGit



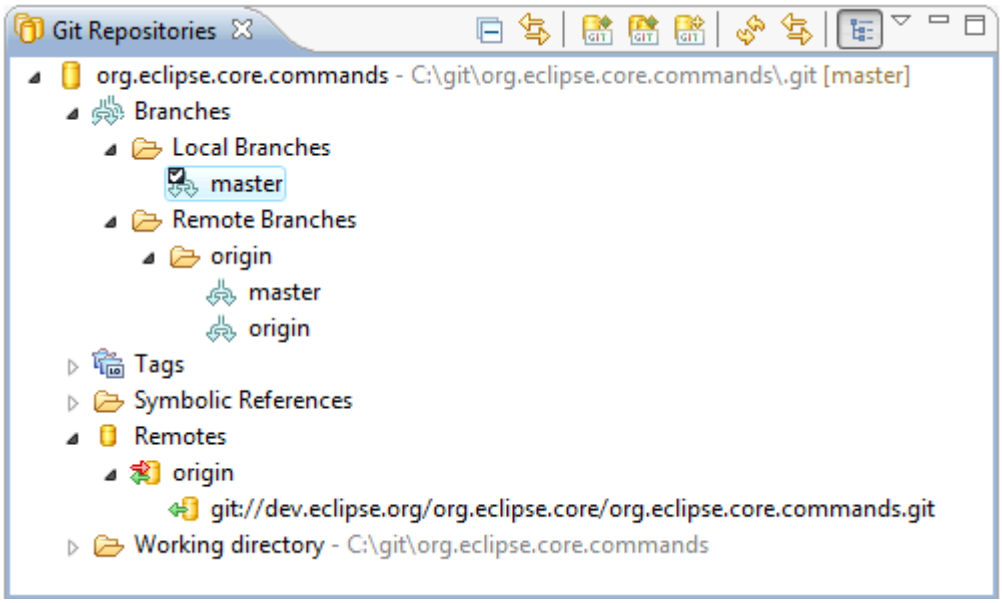
对于习惯 Tortoise 风格的用户，可以选择 TortoiseGit 安装包，安装后的右键截图如图 8 所示。http://code.google.com/p/tortoisegit/。

图 8. 从 Windows 上安装“右键”Git 工具——TortoiseGit



而对于习惯 Eclipse 风格的用户，可以选择 Eclipse 插件——EGit 方式安装，其 Git Repositories
载地址为：http://download.eclipse.org/egit/updates。

图 9. 从 Windows 上安装 Eclipse 的 Git 插件——EGit



无论是哪一种安装方式，如果是第一次使用 Git，均需要配置用户信息，包括用户名与 Email（
交时都可以自动引用这两条信息，说明是谁更新与提交了代码。

```
$ git config --global user.name "Pi Guang Ming"
$ git config --global user.email piguangming@gmail.com
```

到此为止，我们已经介绍了 Git 的分布式模型、快照模型、针对不同操作系统平台的 Git 安装之
即 Git 的使用。

如何使用 Git

前面提到，这一部分是本文的重点。我们将主要精力集中在 Git 底层的工作原理、以及实际工程
补丁、CVS 与 SVN 针对 Git 的迁移等，关于 Git 的各种基础命令语法、解释说明，以及本文没有
关使用指南。

创建 Git 项目仓库

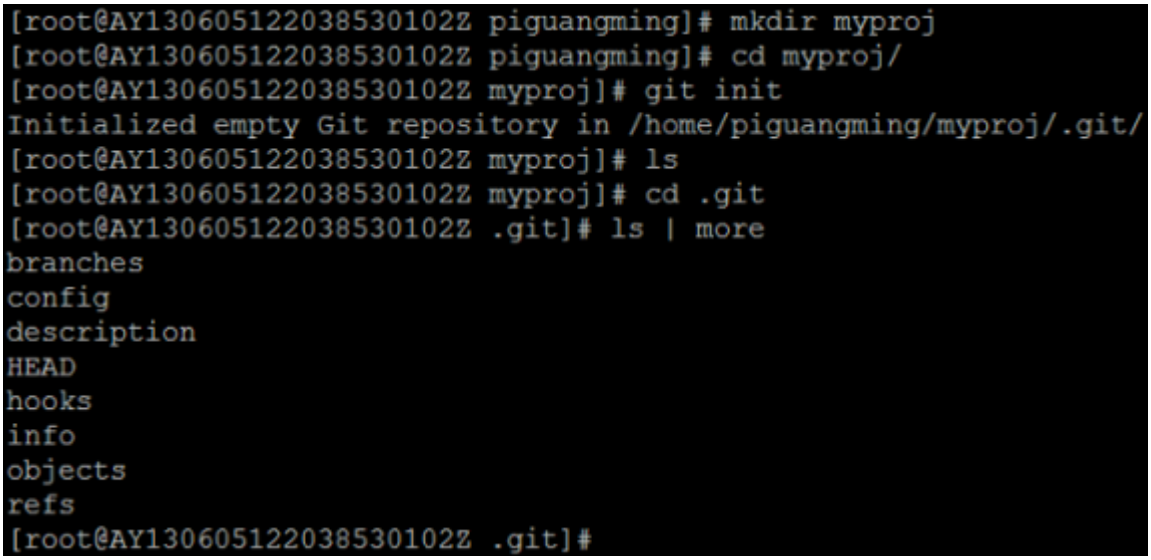
在正式使用 Git 之前，我们至少需要创建一个 Git 代码仓库（简称 Git 仓库）。通常而言，取得
种是在现存的目录下，通过导入所有文件来创建新的 Git 仓库；第二种是从远程 Git 镜像仓库直

针对第一类 Git 仓库，我们可以使用 git init 命令创建一个崭新的 Git 项目仓库，如下：

```
$ git init
```

初始化 Git 后，在当前目录下会出现一个名为 .git 的隐藏目录，如图 10 所示。

图 10. .git 目录



之所以特意强调 .git 目录，是因为它十分重要。对于一个 Git 仓库来说，其 .git 目录保存了整个 .git 目录中各种文件的简要解释说明，如表 1 所示。如果需要了解详细信息，请参见 Git 官方网

表 1 .git 目录简要说明

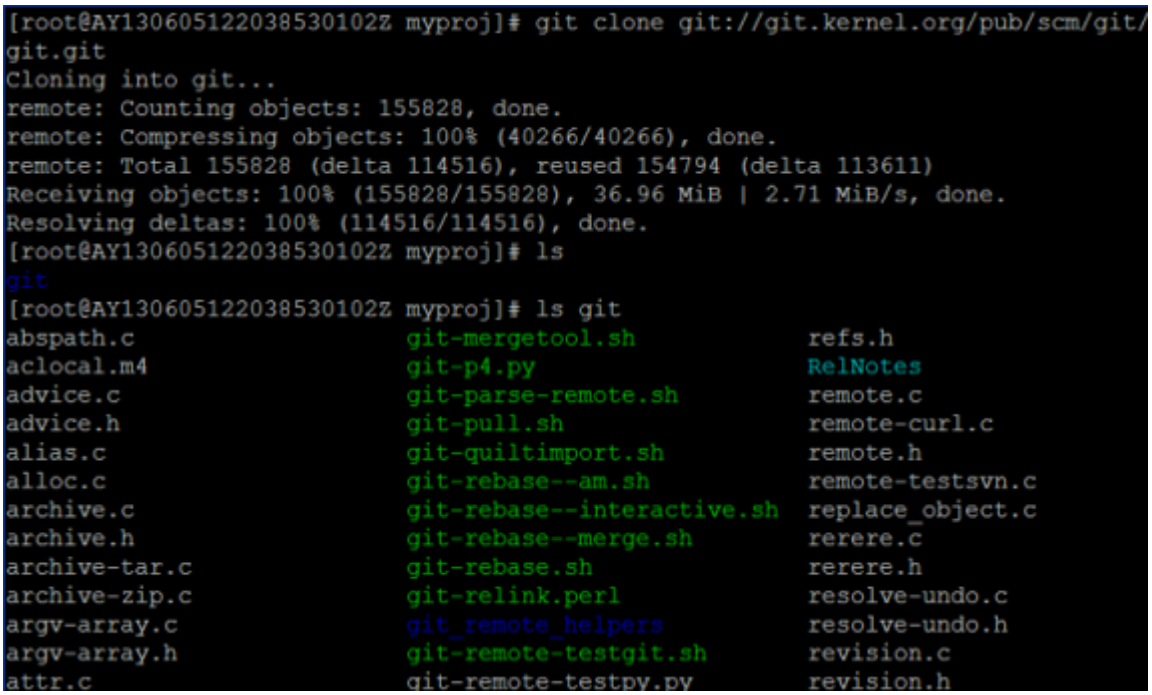
子目录名	简要描述
branches	Git 项目分支信息，新版 Git 已经不再使用该目录。
config	Git 项目配置信息
description	Git 项目描述信息
HEAD	指向 Git 项目当前分支的头指针
hooks	默认的"hooks"脚本，被特定事件发生前后触发。
info	里面含一个 exclude 文件，指 Git 项目要忽略的文件。
objects	Git 的数据对象，包括：commits, trees, blobs, tags。
refs	指向所有 Git 项目分支的指针

针对第二类 Git 仓库，我们不需要 git init 初始化仓库，取而代之的是，使用 git clone 直接将远们以下载 Git 软件本身的源代码为例，其 git clone 命令如下：

```
1 git clone git://git.kernel.org/pub/scm/git/git.git
```

通过 ls git 命令，我们可以查看 Git 仓库中的内容，如图 11 所示。需要说明的是，针对远程仓
录下的数据，然后根据元数据恢复成原来的整个项目结构，也即是图 11 所示的内容。

图 11. 克隆 Git 源代码



此外，除了 git:// 协议，针对不同的使用场景，git clone 还支持 ssh://、http(s):// 等各种不同协

Git 对象模型

应该说，Git 对象模型是整个 Git 设计思想中最核心的部分。理解 Git 对象模型是理解整个 Git 的
包含三部分：类型，大小和内容。其中，对象的类型又分为 commits, trees, blobs, tags，其简

- blob 对象：一块二进制数据，用来存储文件数据，通常是一个文件。
- tree 对象：指向 blob 对象或是其它 tree 对象的指针，一般用来表示内容之间的目录层次关
- commit 对象：一个 commit 对象只指向一个 tree 对象，用来标记项目某一个特定时间点的
  提交者等。
- tag 对象：与 CVS、SVN 标签的概念类似。

接下来，我们结合一个示例来解释不同 Git 对象之间的关系。图 12 展示的是一个样例 Ruby 项
单，仅作示意。

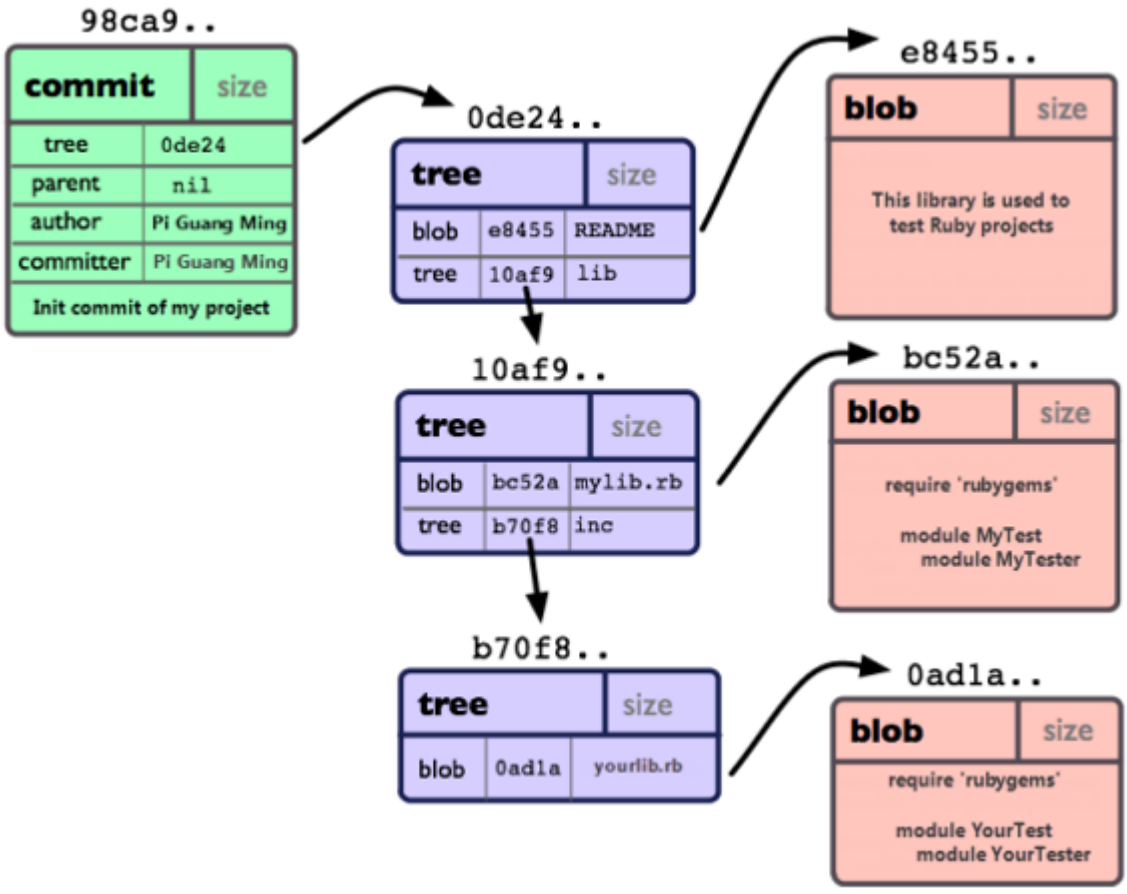
图 12. Ruby 项目的目录层次结构

```
[root@AY1306051220385301022 myproj]# tree
.
|-- README
`-- lib
    |-- inc
    |   `-- yourlib.rb
    `-- mylib.rb

2 directories, 3 files
```

如果我们把该项目提交到 Git 仓库中，那么它的 Git 对象关系就如图 13 所示。其中，3 个 blob mylib.rb、yourlib.rb 三个文件的内容快照。而 3 个 tree 对象指针则完整描述了项目的整个目录 blob 对象的对应关系，各个文件对应 blob 对象索引等信息。而每一次提交都会生成一个 comm 根节点，不仅如此，commit 对象还包含作者、提交人等详细信息。

图 13. Ruby 项目的 Git 对象关系图



不难看出，众多 tree 对象与 blob 一起，作为内容节点（目录或文件），构成了一个有向无环图 对象关联的根节点，就可以遍历出整个项目在本次提交时的所有内容。而前面提到，Git 分支本 两个 Git 分支的合并，实质上是等价于两个有向无环图的合并，而有向无环图可以让 Git 更加高 此，Git 对象模型设计赋予了开发人员最大的灵活性来任意创建分支，并在自己的分支上进行开

尽管以上几种对象的类型不同，每一种对象都拥有同一长度的唯一标识，以 40 位字符串表示。 写，其中，commit 对象完整的标识如下：

```
98ca9e0acb0be0321191a59e1d34ba5c867fa3
```

为保证对象标识的唯一性，Git 采用了 SHA1 哈希算法。这样做，起码有三大好处：

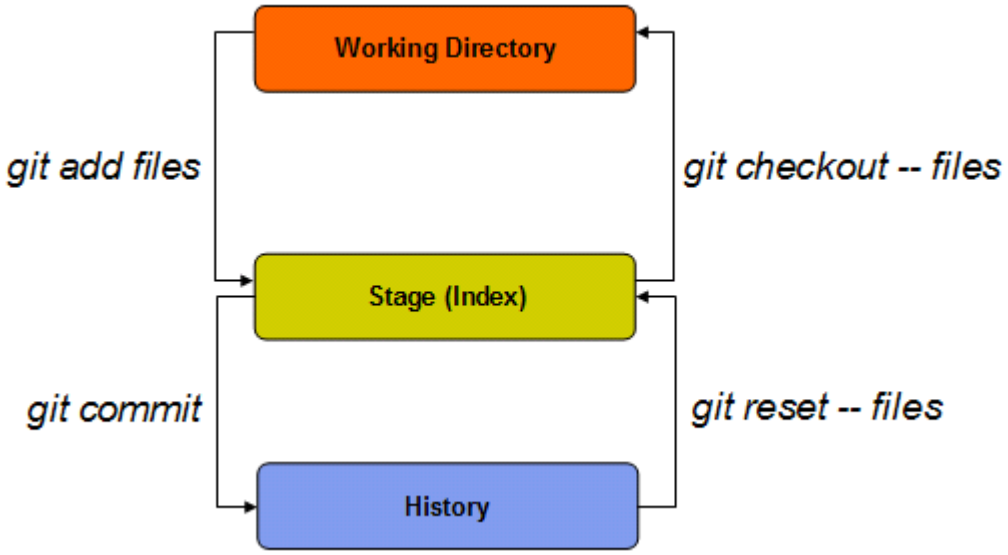
- Git 只要比较对象名，就可以很快的判断两个对象是否相同。
- 由于每个仓库中"对象名"的计算方法都完全一样，因此，如果同样的内容存在两个不同的仓
  名"下。
- Git 还可以通过检查对象内容的 SHA1 哈希值与"对象名"是否相同，来判断对象内容是否正

总结一下 Git 对象模型，blob 对象即项目中的所有实体文件，包括源代码、图片资源、xml 配置
是，blob 对象记录的仅仅是文件内容，而关于文件所在目录、名字大小等信息，则统统记录在
提交文件，都会产生一个 commit 对象，并更新改动文件所关联的 tree 对象。

Git 三种状态

在理解 Git 对象模型之后，我们的焦点转向 Git 文件的检入与检出。Git 仓库模型大致分为三个
（Working Directory），暂存区域（Stage 或 Index），以及本地仓库（History），相应的检

图 14. Git 三种状态之间的转换（1）



相关命令的简要说明如下：

- git add files：把当前工作文件拷贝到暂存区域。
- git commit：在暂存区域生成文件快照并提交到本地仓库。
- git reset -- files：用来撤销最后一次 git add files，也可以用 git reset 撤销所有暂存区域文
- git checkout -- files：把文件从暂存区域覆盖到工作目录，用来丢弃本地修改。

作为示例，图 15 演示了如何通过 git add 与 git checkout 分别在工作目录与暂存区之间来回复
与 git reset 命令。首先，我们在工作目录创建一个内容为"hello git"的 test 文件，通过 git add

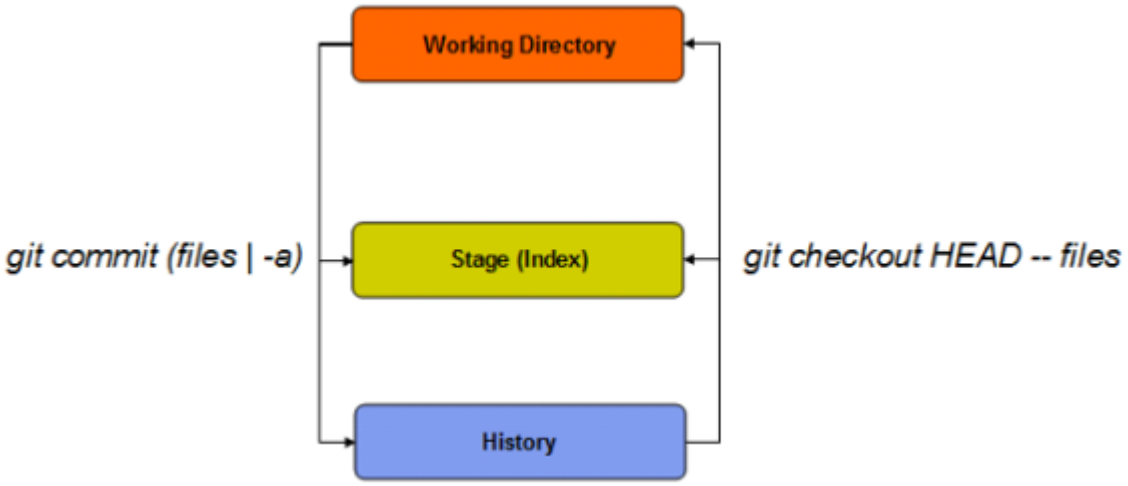
然后，在工作目录修改 test 文件，添加一行"hello git branch"。此时，暂存区的内容依然为"he
git checkout 将暂存区的 test 文件覆盖工作目录，即放弃了本地修改，最终文件内容为"hello g

图 15. git checkout -- files 示例

```
[root@AY1306051220385301022 piguangming]# cd myproj/
[root@AY1306051220385301022 myproj]# git init
Initialized empty Git repository in /home/piguangming/myproj/.git/
[root@AY1306051220385301022 myproj]# echo "hello git" >> test
[root@AY1306051220385301022 myproj]# git add test
[root@AY1306051220385301022 myproj]# echo "hello git branch" >> test
[root@AY1306051220385301022 myproj]# git checkout -- test
[root@AY1306051220385301022 myproj]# cat test
hello git
[root@AY1306051220385301022 myproj]#
```

实际上，工作目录与仓库之间的复制也可以一步到位，如图 16 所示。

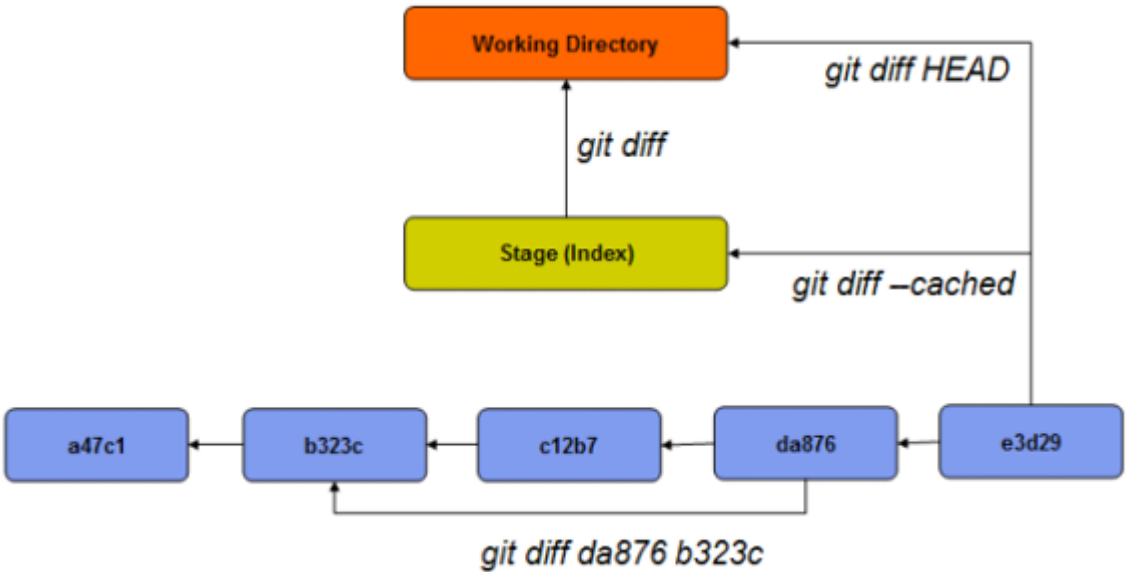
图 16. Git 三种状态之间的转换（2）



其中，git commit -a 等价于 git add 与 git commit，即先把文件从工作目录复制到暂存区，然后
checkout HEAD -- files 的过程刚好相反，即回滚到最后一次提交。

为了查看工作目录，暂存区域，以及本地仓库的文件有哪些不同，可以使用 git diff 命令，如图

图 17. Git 三种状态之间的比较



git diff 命令相关的简要说明如下：

- git diff：查看尚未暂存的文件更新了哪些部分。
- git diff --cached：查看已暂存文件和上次提交时的快照之间的差异。
- git diff HEAD：查看未暂存文件与最新提交文件快照的区别。
- git diff <index1> <index2>：查看不同快照之间的区别。

作为示例，图 18 演示了 git diff 的用法。可以看到，通过 git diff 比较，知道工作目录比暂存区
git diff HEAD 比较，知道工作目录又比仓库最新提交文件多了两行，分别是"hello git branch"与
区比仓库多了一行"hello git branch"，而这恰好与 git diff –cached 的结论相吻合。

图 18. Git 三种状态之间的比较——示例

```
[root@AY1306051220385301022 piguangming]# cd myproj/
[root@AY1306051220385301022 myproj]# ls
[root@AY1306051220385301022 myproj]# git init
Initialized empty Git repository in /home/piguangming/myproj/.git/
[root@AY1306051220385301022 myproj]# echo "hello git" >> test
[root@AY1306051220385301022 myproj]# git add test
[root@AY1306051220385301022 myproj]# git commit test -m 'init'
[master (root-commit) cba45d6] init
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 test
[root@AY1306051220385301022 myproj]# echo "hello git branch" >> test
[root@AY1306051220385301022 myproj]# git add test
[root@AY1306051220385301022 myproj]# echo "hello git tag" >> test
[root@AY1306051220385301022 myproj]# git diff
diff --git a/test b/test
index 405d568..7a78dc4 100644
--- a/test
+++ b/test
@@ -1,2 +1,3 @@
 hello git
 hello git branch
+hello git tag
[root@AY1306051220385301022 myproj]# git diff HEAD
diff --git a/test b/test
index 8d0e412..7a78dc4 100644
--- a/test
+++ b/test
@@ -1 +1,3 @@
 hello git
+hello git branch
+hello git tag
[root@AY1306051220385301022 myproj]# git diff --cached
diff --git a/test b/test
index 8d0e412..405d568 100644
--- a/test
+++ b/test
@@ -1 +1,2 @@
 hello git
+hello git branch
[root@AY1306051220385301022 myproj]#
```

以上是关于 Git 检入与检出操作的基础用法，关于更详细命令以及语法说明，可参见相关 Git 学

接下来，我们介绍 Git 更加高级的功能与特性。

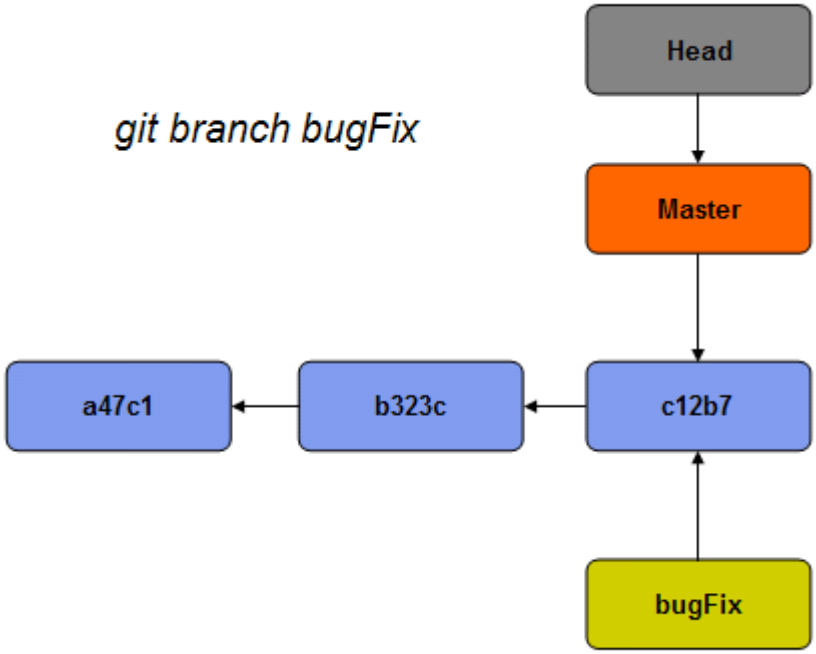
Git 分支模型

前面提到，Git 中的分支本质上是一个指向 commit 对象的可变指针。Git 会维护一个默认分支-master 指针都会自动向前移动。而如果要创建一个新的分支，可以使用 git branch 命令：

```
1 $ git branch bugFix
```

这会在当前 commit 对象上新建一个分支指针，如图 19 所示。

图 19. 新建分支 bugFix



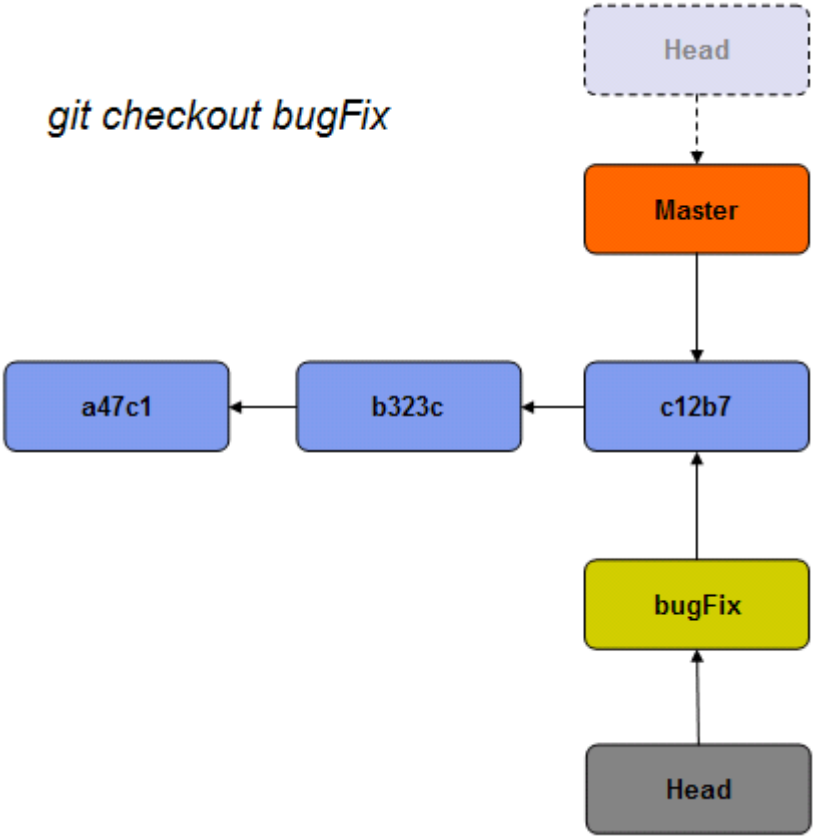
那么，Git 是如何知道当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HE
前工作分支的指针。我们可以将 HEAD 想象为当前分支的别名。在这一点上，它和 CVS、SVN

运行 git branch 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在
分支里工作。要切换到其他分支，可以执行 git checkout 命令。

```
1 $ git checkout bugFix
```

这样 HEAD 就指向了 bugFix 分支，见图 20 所示。

图 20. 切换到 bugFix 分支



实际上，我们可以将分支的创建与切换两步合二为一。要新建并切换到该分支，运行 git check

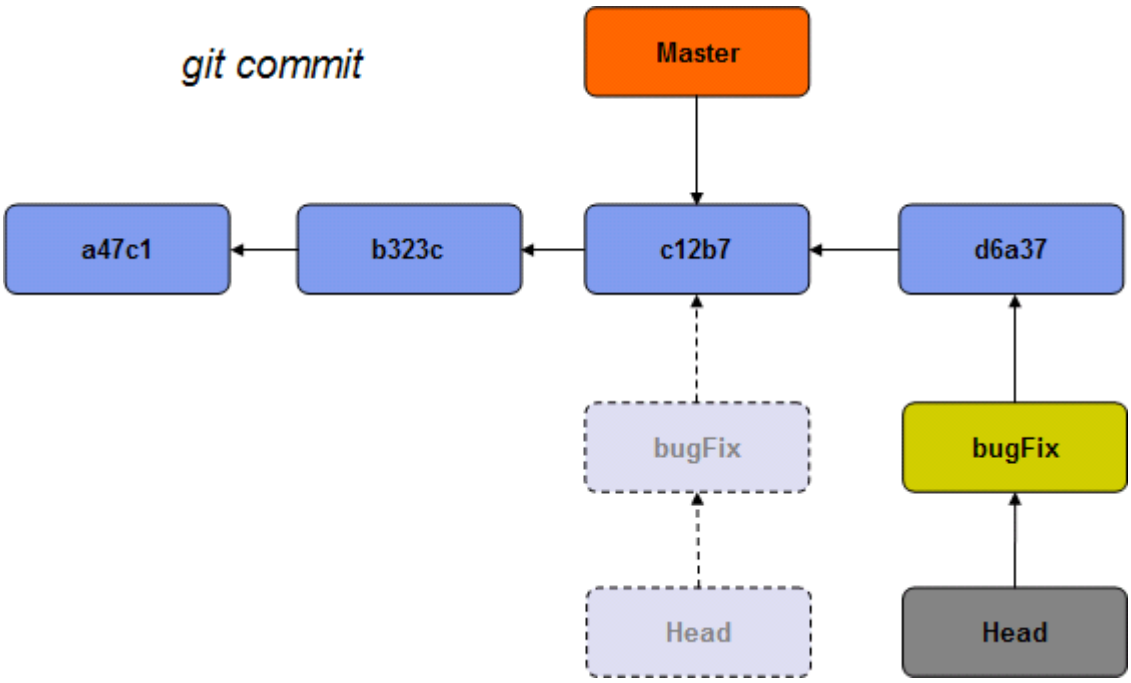
```
$ git checkout -b bugFix
```

接下来，再提交一次：

```
$ vi test.rb
$ git commit -a -m 'update copyright'
```

图 21 展示了提交后的结果。非常有趣，现在 bugFix 分支向前移动了一格，而 master 分支仍然
commit 对象。

图 21. 在 bugFix 分支提交文件

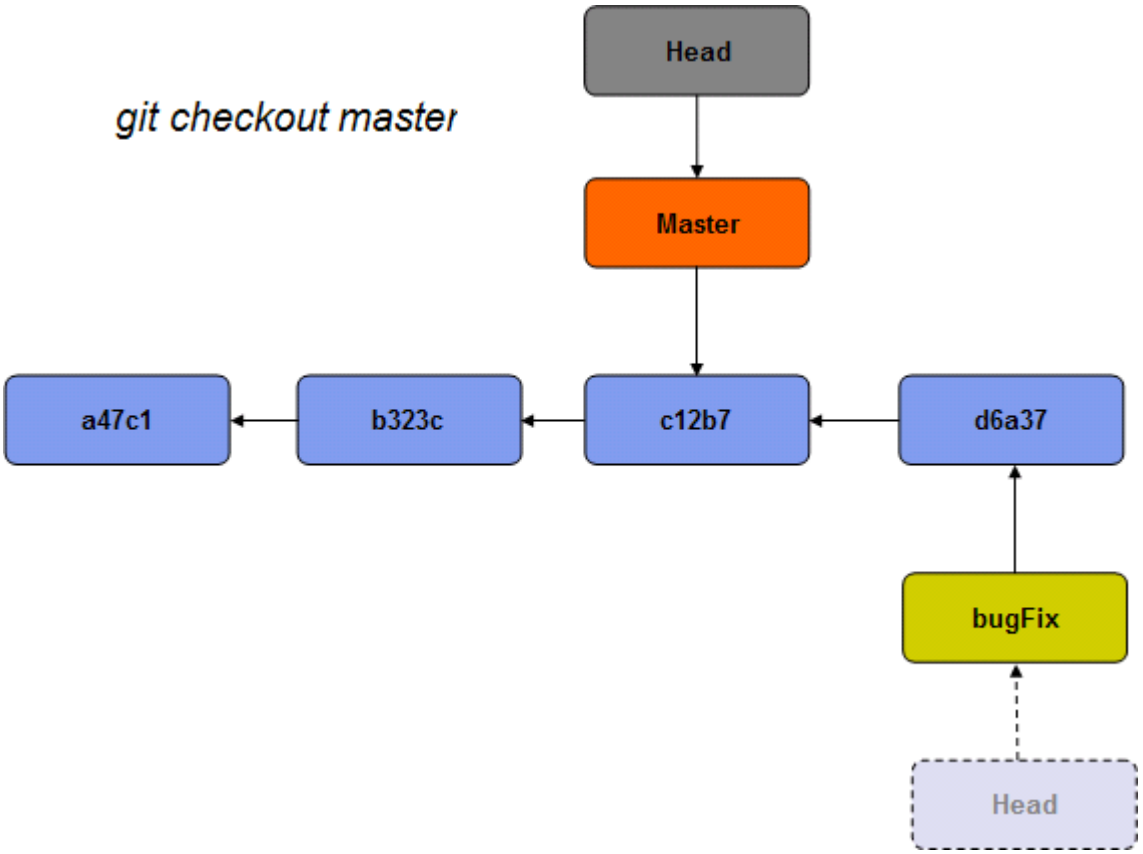


我们再切换到 master 分支：

```
$ git checkout master
```

其结构如图 22 所示。这条命令做了两件事。第一，它把 HEAD 指针移回到 master 分支；第二
master 分支所指向的快照内容。也就是说，从现在开始，基于该文件的一系列提交都将始于一
于，可以将 bugFix 分支里作出的修改暂时取消，隔离 bugFix 分支对 master 分支的影响。在实
求，即采用 developer 分支开发主要版本，bugFix 分支负责修复 bug，彼此互相隔离，最后合

图 22. 切换成 master 分支

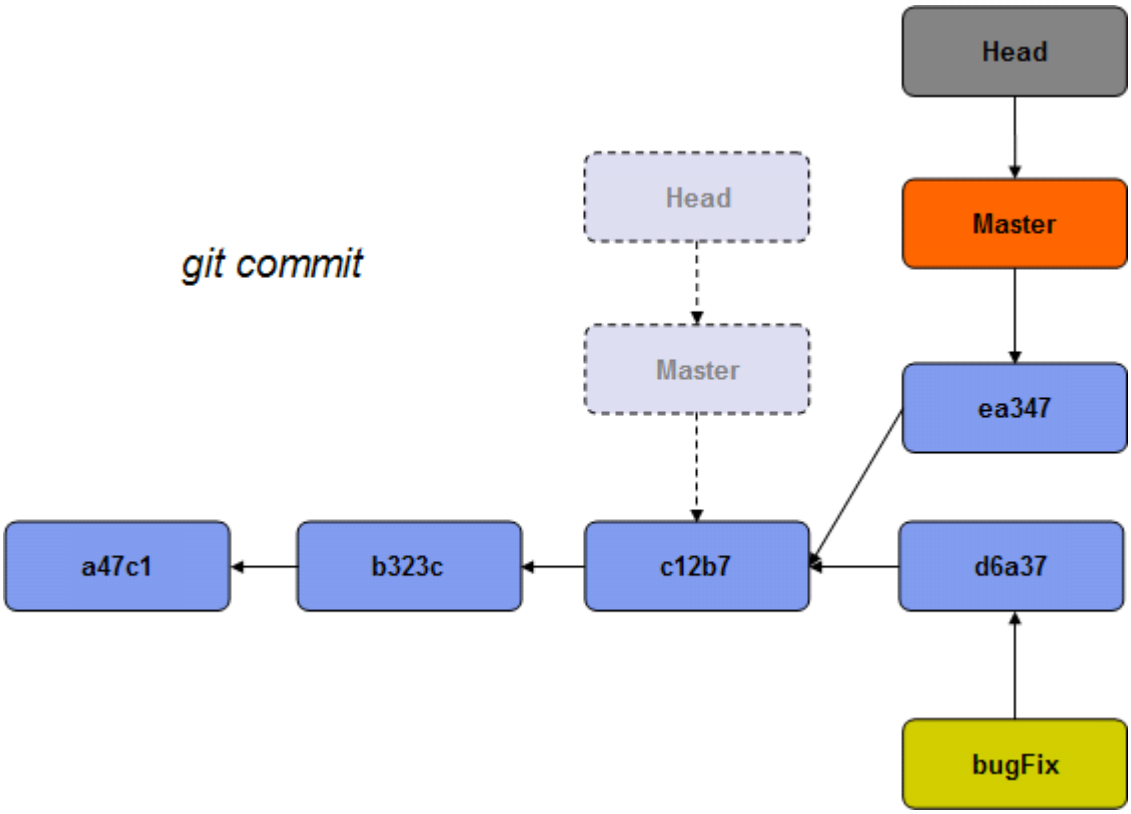


我们作些修改后再次提交：

```
$ vi test.rb
$ git commit -a -m 'made other changes'
```

现在我们的项目提交历史产生了分叉，如图 23 所示，原因是刚才我们创建了一个分支，进行了
进行了另一些工作。我们可以在不同分支里反复切换，并在时机成熟时将它们合并到一起。

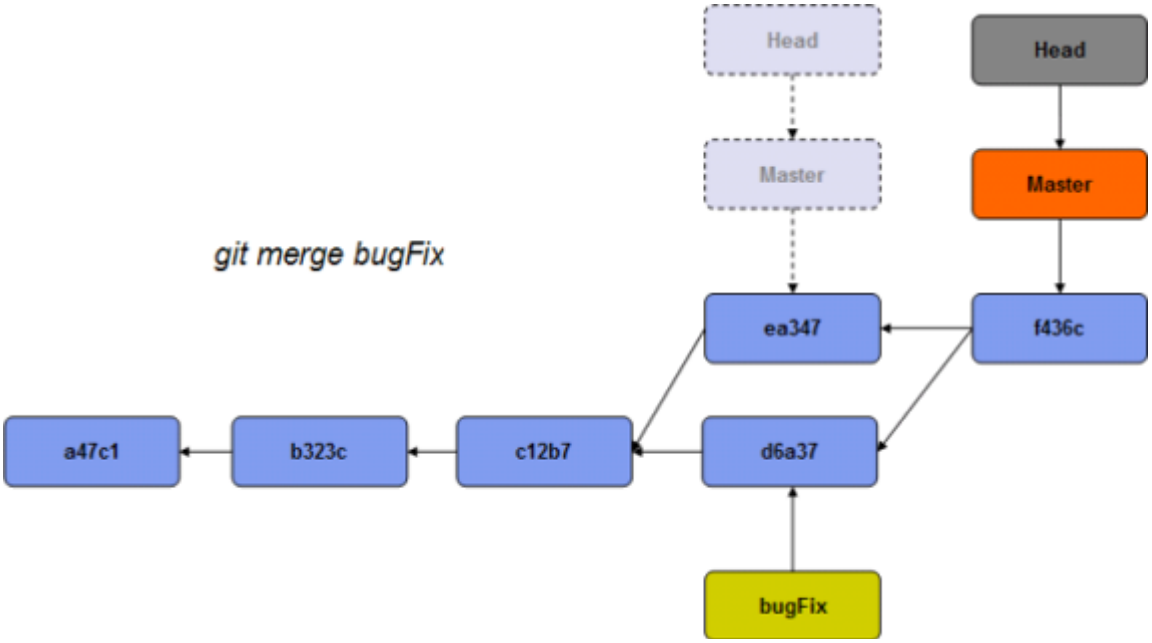
图 23. 在 master 分支提交文件



git merge 命令把不同分支合并起来。合并前，HEAD 必须指向当前最新的提交。按使用场景不
况：

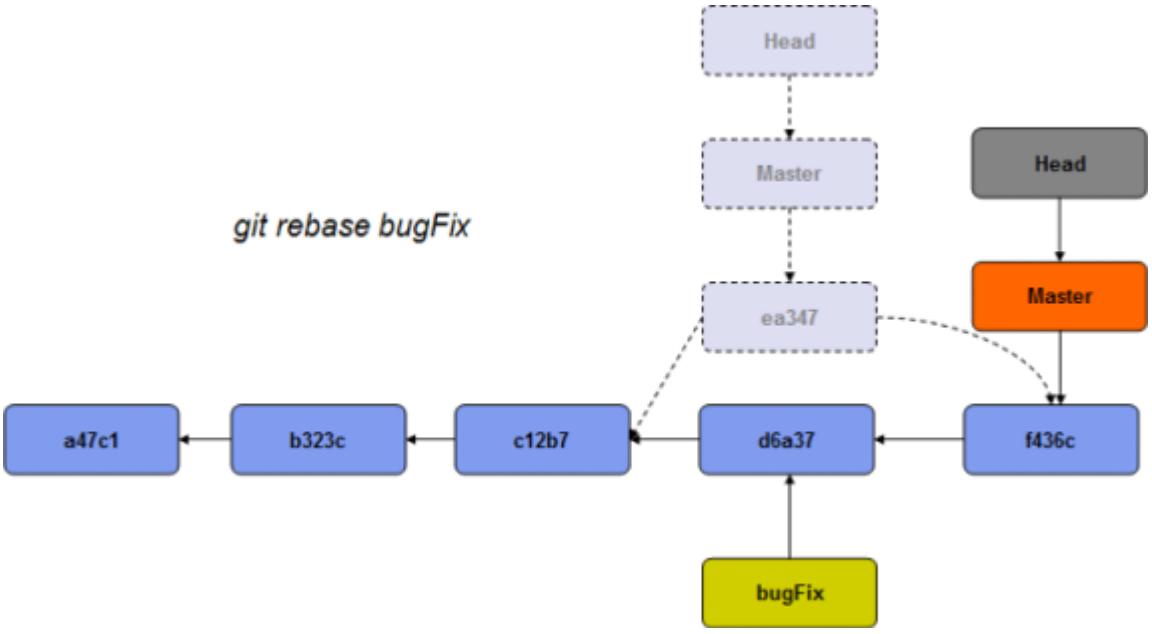
- 如果另一个分支是当前提交的祖父节点，那么 git merge 命令将什么也不做。
- 反过来，如果当前提交是另一个分支的祖父节点，就导致 fast-forward 合并。指向只是简单
- 否则就是一次真正的合并。默认把当前提交 (ed489 如下所示) 和另一个提交 (33104) 以及
一次三方合并。结果是先保存当前目录和索引，然后和父节点 33104 一起做一次新提交，

图 24. 合并分支



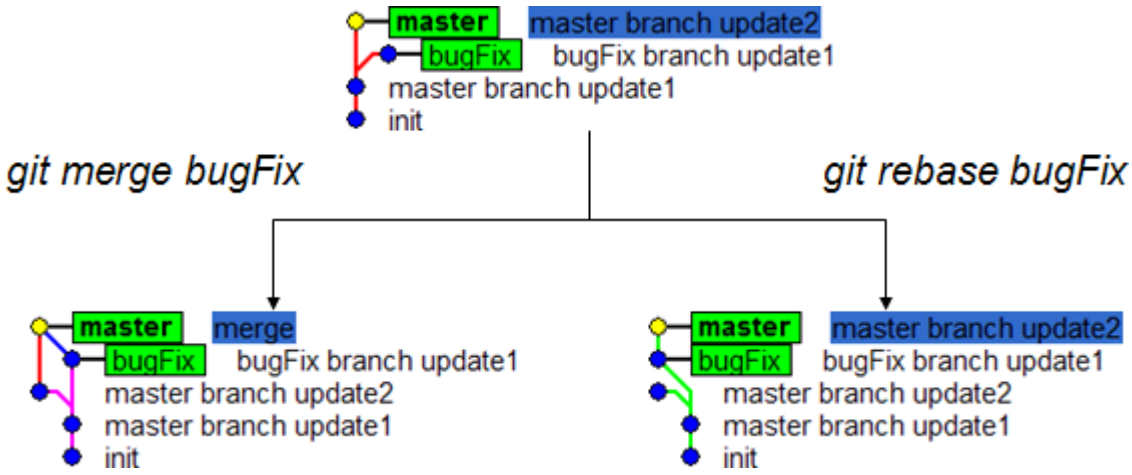
可以看到，git merge 命令把两个父分支合并进行一次提交，但提交历史不是线性的。相比之下分支上重演另一个分支的历史，从而保证提交历史是线性的，如图 25 所示。

图 25. 衍合分支



作为示例，我们演示关于 git merge 与 git rebase 的区别，见图 26 所示。

图 26. 合并分支 vs 衍合分支



有时候合并操作并不会如此顺利，如果在不同的分支中都修改了同一个文件的同一部分，会造成
两者合到一起。此时，Git 仅作合并，但不提交，它会停下来等人为地解决冲突，如下：

```
$ cat test.rb
init
master update1
master update2
bugFix update1
<<<<<<< HEAD
master updated3
=======
bugFix update2
>>>>>>> bugFix
```

要查看哪些文件在合并时发生冲突，可以使用 git status：

```
$ git status
# On branch master
# Unmerged paths:
# (use "git add/rm <file>..." as appropriate to mark resolution)
#
# both modified: test.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

待修补发布以后，bugFix 分支已经完成了历史使命，我们可以使用 git branch 的 -d 选项执行删

```
$ git branch -d bugFix
```

以上，我们介绍了 Git 分支的创建，切换，合并（线性与非线性），冲突，以及删除。

Git 标签

与 CVS、SVN 等其它版本控制系统一样，Git 也支持打 Git 标签 。在程序开发到一个阶段后，我
本，如 0.1.2，v0.1.2 等。

Git 使用的标签有两种类型：轻量级的（lightweight）和含附注的（annotated）。轻量级标签
引用；而含附注标签实际上是存储在仓库中的一个独立 Git 对象。相比之下，含附注标签包含
签名字，Email，标签日期，以及标签说明等。含附注标签本身也允许使用 GNU Privacy Guard
荐使用含附注的标签，以便保留相关信息。

要打上标签，可执行以下 Git 命令：

```
$ git tag -a v0.1.2 -m "Release version 0.1.2"
```

相应地，要查看标签，执行下列 Git 命令：

```
$ git tag –l
```

当然，也可采用 git show 命令查看标签版本与提交对象等详细信息。

```
$ git show v0.1.2
```

删除标签的 Git 命令如下：

```
git tag -d v0.1.2
```

如果我们有自己的私钥，还可以用 GPG 来签署标签，只需要把之前的 -a 改为 -s，如下

```
$ git tag -s v0.1.2 -m "My signed 0.1.2 tag"
```

要验证已经签署的标签，可以先取到对应的公钥，然后使用 git tag –v 命令验证，如下：

```
$ git tag -v v0.1.2
```

需要注意的，默认情况下，git push 并不会把标签传送到远端仓库上。我们只能通过显式命令

```
$ git push origin v0.1.2
```

如果希望一次性推送所有本地新增的标签，可以使用 --tags 选项：

```
$ git push origin --tags
```

如此一来，其他人克隆共享仓库或拉取数据同步后，也会看到这些标签。

Git 补丁

UNIX 世界中，补丁（Patch）的概念非常重要，几乎所有大型 UNIX 项目的普通贡献者，都是
内核项目而言，普通开发者先从 Git 项目仓库克隆下代码，然后写入代码，做一个补丁，最后
就可以了。

Git 提供了两种简单的补丁生成方案。一是使用 git diff 生成的标准补丁，二是使用 git format-
我们重点介绍第二种方式，关于第一种 git diff 方式，比较简单，这里不做介绍。

假设我们有一个项目 myproj，其工作目录里最初有一个文件 test，内容是"hello git"，默认提
一个新分支 bugFix 用于代码修改，如图 27 所示：

图 27. 创建分支

```
[root@AY1306051220385301022 piguangming]# mkdir myproj
[root@AY1306051220385301022 piguangming]# cd myproj/
[root@AY1306051220385301022 myproj]# git init
Initialized empty Git repository in /home/piguangming/myproj/.git/
[root@AY1306051220385301022 myproj]# echo 'hello git' >> test
[root@AY1306051220385301022 myproj]# git add .
[root@AY1306051220385301022 myproj]# git commit -a -m 'init'
[master (root-commit) 2cd78b9] init
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 test
[root@AY1306051220385301022 myproj]# git branch bugFix
[root@AY1306051220385301022 myproj]# git checkout bugFix
Switched to branch 'bugFix'
[root@AY1306051220385301022 myproj]#
```

接下来，我们在 test 文件里面追加一行"fix"，并使用 git format-patch 生成一个 patch，如图 2
的 -M 选项表示这个 patch 要和哪个分支比对。

图 28. 生成补丁

```
[root@AY1306051220385301022 myproj]# echo 'fix' >> test
[root@AY1306051220385301022 myproj]# git commit -a -m 'fix'
[bugFix 529db12] fix
 1 files changed, 1 insertions(+), 0 deletions(-)
[root@AY1306051220385301022 myproj]# git format-patch -M master
0001-fix.patch
[root@AY1306051220385301022 myproj]# cat 0001-fix.patch
From 529db1216b95d61c142a8db0679863761e45658a Mon Sep 17 00:00:00 2001
From: Pi Guang Ming <piguangming@gmail.com>
Date: Sat, 8 Jun 2013 13:00:58 +0800
Subject: [PATCH] fix

---
 test |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

diff --git a/test b/test
index 8d0e412..91b84d9 100644
--- a/test
+++ b/test
@@ -1 +1,2 @@
 hello git
+fix
--
1.7.6

[root@AY1306051220385301022 myproj]#
```

可以看到，补丁文件 0001-fix.patch 包含各种信息，不仅有 diff 的信息，还有提交者，时间等
mail 的文件，可以直接发送。

接下来，可以使用 git am 来应用补丁，如图 29 所示。可以看到，相比于原来的 test 文件，打

图 29. 应用补丁

```
[root@AY1306051220385301022 myproj]# git checkout master
Switched to branch 'master'
[root@AY1306051220385301022 myproj]# git branch PATCH
[root@AY1306051220385301022 myproj]# git checkout PATCH
Switched to branch 'PATCH'
[root@AY1306051220385301022 myproj]# git am 0001-fix.patch
Applying: fix
[root@AY1306051220385301022 myproj]# git add .
[root@AY1306051220385301022 myproj]# git commit -m 'patch applied'
[PATCH 6cc6d7d] patch applied
 1 files changed, 19 insertions(+), 0 deletions(-)
 create mode 100644 0001-fix.patch
[root@AY1306051220385301022 myproj]# cat test
hello git
fix
[root@AY1306051220385301022 myproj]#
```

关于以上两种生成补丁方式的比较，很明显，相比于 git diff 生成的通用补丁，git format-patcl
弱。不过，Git 专用补丁中含有补丁开发者的名字，在应用补丁时，这个名字会被记录进版本库
往往建议大家使用 format-patch 生成补丁。

Git 远程仓库操作

前面提到，Git 是分布式版本控制系统。对于一个分布式节点来说，其它节点的 Git 仓库都可以
当前配置有哪些远程仓库，可以使用以下命令：

```
$ git remote
```

在克隆完某个项目后，至少可以看到一个名为 origin 的远程库，Git 默认使用这个名字来标识你

项目进行到一个阶段，要同别人分享目前的成果，可以使用 git push 命令将本地仓库中的数据

```
$ git push origin master
```

而要将远程仓库抓取数据到本地，可以使用 git fetch 命令，从而获取所有本地仓库中还没有的

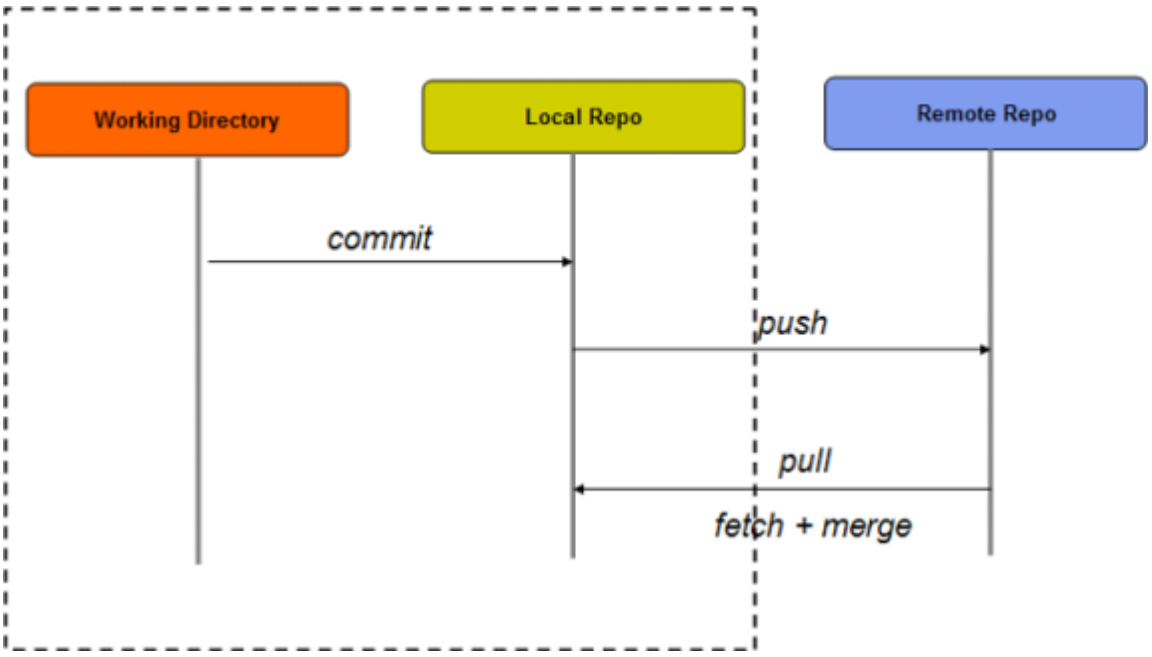
```
$ git fetch [remote-name]
```

如果设置了某个分支用于跟踪某个远端仓库的分支，可以使用 git pull 命令自动抓取数据下来，
仓库中当前分支。从这个角度，git pull 等价于 git fetch + git merge 的功能。

```
$ git pull [remote-name]
```

关于以上几种 Git 远程仓库的相关操作，其关系见图 30 所示。要了解 Git 远程仓库的更多命令
Git 操作指南。

图 30. Git 远程仓库的操作



CVS 迁移到 Git

对于想要从 CVS 迁移到 Git 的用户，可以使用 git cvsimport 工具解决迁移问题，前提是安装相

关于 git-cvs 工具，可以使用 yum 或者 apt-get 命令安装。以 yum 为例，其安装命令如下：

```
$ yum install git-cvs
```

如果是源码编译安装 Git，则需要安装 cvsps，下载地址：http://www.cobite.com/cvsps/

```
$ tar -zxvf cvsps-2.1.tar.gz
$ cd cvsps-2.1
$ make && make install
```

作为示例，我们新建一个目录 jt400.cvs，并将文章开头提到的 SourceForge 托管的 CVS 项目

入到 Git 中来，操作过程如下：

```
$ mkdir jt400.cvs
$ cd jt400.cvs
$ export CVSROOT=:pserver:piguangming@jt400.cvs.sourceforge.net:/cvsroot/jt400
$ cvs login
$ git cvsimport -C src src

其中，-C src 是要往 git 仓库里创建的项目名称，最后那个 src 是 cvs 中要导入的模块。
```

SVN 迁移到 Git

同样，Git 也提供了 git svn 相关工具，提供 SVN 项目到 Git 的迁移，前提是安装相关工具 subv

```
$ yum install install subversion-perl

作为示例，我们新建一个目录 photon-android.svn，并将 googlecode 托管的 SVN 项目 photon

$ mkdir photon-android.svn
$ cd photon-android.svn
$ git svn clone <a href="http://photon-android.googlecode.com/svn/">http://pho
```

总结

本文系统性地介绍了分布式版本控制工具——Git，包括为什么使用 Git，Git 的安装，Git 的工作

SVN 向 Git 迁移等。有关 Git 更全面的使用方法，请参见文档：https://github.com/progit/prog

---

相关主题

- 访问 progit。
- 访问 git-scm.com 及其下载。
- 访问 Git for OS X。
- 访问 msysgit。
- 访问 tortoisegit。
- 访问 eclipse.org/egit/updates。
- 访问 CVSps。
- 访问 photon-android。
- 访问 Toolbox for Java/JTOpen。
- 访问 developerWorks Open source 专区获得丰富的 how-to 信息、工具和项目更新，帮助您
  它们与 IBM 产品结合使用。

评论

添加或订阅评论，请先登录或注册。

有新评论时提醒我