Git 用户手册（1.5.3 及后续版本适用）

- *翻译: **罗峥嵘 (Robin Steven)** < vortune@gmail.com >*
- *英文版本: http://www.kernel.org/pub/software/scm/git/docs/user-manual.html*

Contents

1. Maintaining topic branches for a Linux subsystem maintainer | Linux 子系统维护者如何维护主题分支

6. Chapter 5. Rewriting history and maintaining patch series 第五章. 改写历史与维护补丁串

1. Creating the perfect patch series 创建出色的补丁串
2. Keeping a patch series up to date using git-rebase 使用 git-rebase 保持补丁串的新颖
3. Rewriting a single commit 重写单个交付
4. Reordering or selecting from a patch series 在补丁串中选取与重新排序
5. Other tools 第三方工具
6. Problems with rewriting history 重写历史带来的问题
7. Why bisecting merge commits can be harder than bisecting linear history 为何定位合并交付中的问题要比在线性历史中困难

7. Chapter 6. Advanced branch management 第六章. 高级分支管理

1. Fetching individual branches
2. git fetch and fast-forwards 抓取与快进
3. Configuring remote branches 配置远程分支

8. Chapter 7. Git concepts 第七章. Git 概念

1. The Object Database 对象数据库

1. Commit Object 交付对象
2. Tree Object 树对象
3. Blob Object 片对象
4. Trust 信赖
5. Tag Object 标签对象
6. How git stores objects efficiently: pack files | git 如何高效地储存对象： 打包文件
7. Dangling objects 悬空对象
8. Recovering from repository corruption 从损坏中恢复

2. The index 索引

9. Chapter 8. Submodules 子模块

1.

1. Pitfalls with submodules 子模块陷阱

10. Chapter 9. Low-level git operations 第九章. 底层 git 操作

1. Object access and manipulation 对象访问与操作
2. The Workflow 运作流程

1. working directory -> index 工作树 -> 索引
2. index -> object database 索引 -> 对象数据库
3. object database -> index 对象数据库 -> 索引
4. index -> working directory 索引 -> 工作目录

Preface 前言

Git is a fast distributed revision control system.

Git 是一个快速的分布式版本控制系统

This manual is designed to be readable by someone with basic UNIX command-line skills, but no previous knowledge of git.

这个手册是面向那些具有基本的 Unix 命令行使用技能，但是没有 Git 知识的人设计的。

Chapter 1, Repositories and Branches and Chapter 2, Exploring git history explain how to fetch and study a project using git—read these chapters to learn how to build and test a particular version of a software project, search for regressions, and so on.

第一章 版本库与分支 和 第二章 考查 git 历史 将展示如何用 git 来获取和研究一个项目，通过阅读这些章节，我们学习如何建立和测试一个具体的软件项目的版本，学习“撤退”等等。

People needing to do actual development will also want to read Chapter 3, Developing with git and Chapter 4, Sharing development with others.

人们是需要开展真正的研发工作的，那么就学习 第三章， 用 git 进行开发 和 第四章，与他人共享研发成果。

Further chapters cover more specialized topics.

更多的一些章节会涉及到许多的专题话题。

Comprehensive reference documentation is available through the man pages, or git-help(1) command. For example, for the command "git clone <repo>", you can either use:

参考文档可以通过系统的手册页命令，或者是 git-help(1) 命令来查看。譬如，你想参考 "git clone <repo>"， 你可以用下面的两种方式：

```
$ man git-clone
```

or: 或者：

```
$ git help clone
```

With the latter, you can use the manual viewer of your choice; see git-help(1) for more information.

晚一点你就有机会用到这些手册查看器的；看 git-help(1) 会得到比较多的信息。

See also Appendix A, Git Quick Reference for a brief overview of git commands, without any explanation.

阅读 附录 A，那里是一个 git 命令的快速纵览，但是它不带任何的解说。

Finally, see Appendix B, Notes and todo list for this manual for ways that you can help make this manual more complete.

最后，看看 附录 B，这份手册的工作备忘和计划，通过它你可以帮助这份文档变得更完善。

Chapter 1. Repositories and Branches 第一章. 版本库与分支

How to get a git repository 如何获取一个版本库

It will be useful to have a git repository to experiment with as you read this manual.

有一个实验性的 git 版本库对我们阅读这份手册将非常有用。

The best way to get one is by using the git-clone(1) command to download a copy of an existing repository. If you don't already have a project in mind, here are some interesting examples:

获取一个已经存在的版本库，最佳的方法是用 git-clone 命令，如果你还没有什么心目中的项目的话，那么这里有些有趣的例子：

```
        # git itself (approx. 10MB download):
$ git clone git://git.kernel.org/pub/scm/git/git.git
        # the Linux kernel (approx. 150MB download):
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

The initial clone may be time-consuming for a large project, but you will only need to clone once.

对于一个大型项目来说，初始性的克隆是挺费时的，不过克隆只需要做一次。

The clone command creates a new directory named after the project ("git" or "linux-2.6" in the examples above). After you cd into this directory, you will see that it contains a copy of the project files, called the working tree, together with a special top-level directory named ".git", which contains all the information about the history of the project.

克隆命令会创建一个新的目录，并根据项目的名称来命名这个项目（譬如上说例子中的 “git” 和 “linux-2.6”）。当你进入这个目录的时候，你可以看到它已经包含了项目的所有文件，我们称之为工作树，在顶层目录中还连带了一个叫 ".git" 的特殊的目录，里面包含了项目的发展历史的所有信息。

How to check out a different version of a project 如何提取项目的不同版本

Git is best thought of as a tool for storing the history of a collection of files. It stores the history as a compressed collection of interrelated snapshots of the project's contents. In git each such version is called a commit.

最好将 git 当作是文件发展的历史纪录的收集工具，它压缩并保存了项目的发展的关联性快照。在 git 中，每个这些变更称作交付(commit)。

Those snapshots aren't necessarily all arranged in a single line from oldest to newest; instead, work may simultaneously proceed along parallel lines of development, called branches, which may merge and diverge.

这些快照并不需要按从旧到新单线索地发展；它可以是同步并行地发展的，称之为分支，它们是可以合并和分割的。

A single git repository can track development on multiple branches. It does this by keeping a list of heads which reference the latest commit on each branch; the git-branch(1) command shows you the list of branch heads:

一个 git 版本库可以跟踪多个分支的发展，它通过保存一个分支头列表的方式来做到这一点，每个分支头都是一个引用(reference)，它指向该分支最后的一个交付(commit)； git-branch(1) 命令可以向你展示每个分支头：

```
$ git branch
* master
```

A freshly cloned repository contains a single branch head, by default named "master", with the working directory initialized to the state of the project referred to by that branch head.

一个刚刚克隆的版本库只包含一个分支头，默认叫 “master” （主分支），并且工作目录已经被初始化为这个分支头所指向的项目状态。

Most projects also use tags. Tags, like heads, are references into the project's history, and can be listed using the git-tag(1) command:

大部分的项目还用到标签（tags）。标签（Tags）就好像头（heads），它指向项目的某个历史场面，它们可以通过 git-tag(1) 命令列举出来：

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

Tags are expected to always point at the same version of a project, while heads are expected to advance as development progresses.

Tags 被当做是项目统一的版本来对待，而 heads 则是项目前进的每一个步伐。

Create a new branch head pointing to one of these versions and check it out using git-checkout(1):

下面创建一个新的分支头，使其指向其中的某个版本，同时将它提取出来，可以用 git-checkout(1) 命令：

```
$ git checkout -b new v2.6.13
```

The working directory then reflects the contents that the project had when it was tagged v2.6.13, and git-branch(1) shows two branches, with an asterisk marking the currently checked-out branch:

工作目录将被镜像为项目中标记为 v2.6.13 的版本的内容，用 git-branch(1) 命令展示这个两个分支，前面带星号(*)的就是当前抽取的分支。

```
$ git branch
  master
* new
```

If you decide that you'd rather see version 2.6.17, you can modify the current branch to point at v2.6.17 instead, with

如果你打算看看 2.6.17 的版本，你可以迁移你当前的分支，让它指向 2.6.17, 使用一下命令：

```
$ git reset --hard v2.6.17
```

Note that if the current branch head was your only reference to a particular point in history, then resetting that branch may leave you with no way to find the history it used to point to; so use this command carefully.

注意，如果当前的分支头是你唯一的指向具体的历史场面的引用的话，那么复位 (resetting) 这个分支将令你无法找回这个分支以前的所有历史纪录，所以这个命令要慎用。

Understanding History: Commits 理解历史： 交付

Every change in the history of a project is represented by a commit. The git-show(1) command shows the most recent commit on the current branch:

项目的每一个历史变更体现为每一个交付 (commit)。git-show(1) 命令展示当前分支的最新交付：

```
$ git show
commit 17cf781661e6d38f737f15f53ab552f1e95960d7
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>
Date:   Tue Apr 19 14:11:06 2005 -0700

    Remove duplicate getenv(DB_ENVIRONMENT) call

    Noted by Tony Luck.

diff --git a/init-db.c b/init-db.c
index 65898fa..b002dc6 100644
--- a/init-db.c
+++ b/init-db.c
@@ -7,7 +7,7 @@

 int main(int argc, char **argv)
 {
-       char *sha1_dir = getenv(DB_ENVIRONMENT), *path;
+       char *sha1_dir, *path;
        int len, i;

        if (mkdir(".git", 0755) < 0) {
```

As you can see, a commit shows who made the latest change, what they did, and why.

正如你看到的那样，交付表明了谁做的最后的更改，改了什么，为什么改。

Every commit has a 40-hexdigit id, sometimes called the "object name" or the "SHA1 id", shown on the first line of the "git-show" output. You can usually refer to a commit by a shorter name, such as a tag or a branch name, but this longer name can also be useful.

Most importantly, it is a globally unique name for this commit: so if you tell somebody else the object name (for example in email), then you are guaranteed that name will refer to the same commit in their repository that it does in yours (assuming their repository has that commit at all). Since the object name is computed as a hash over the contents of the commit, you are guaranteed that the commit can never change without its name also changing.

每个交付都有一个 40 个 16 进制字符的标识号，称为 “对象名” 或者叫 “SHA1 id”，它显示在 git-show 命令的输出的第一行中。你通常可以用较简短的名字来指明一个交付，譬如标签和分支的名称等等，但是这个长长的名字是很有用的。最重要的是，对于某个交付来说它是全局唯一的名字： 譬如你告诉其他人某个对象名（通过 email 等方式），那么你需要确保这个名字不管是在你的版本库中，还是在他们的版本库中都是指向同一个交付的（假设他们的版本库也被提交了很多东西）。当对象名根据每个交付的内容通过哈希算法（hash）算出之后，你就可以确保每个交付中的内容的修改都不可能脱离它的名字。

In fact, in Chapter 7, Git concepts we shall see that everything stored in git history, including file data and directory contents, is stored in an object with a name that is a hash of its contents.

事实上，在 第七章，Git 的概念 中，我们可以看到保存在 git 历史中的所有东西，包括文件数据与目录内容都会被保存为对象，对象名就是他们的内容的哈希特征值。

Understanding history: commits, parents, and reachability 交付，父交付与可及性

Every commit (except the very first commit in a project) also has a parent commit which shows what happened before this commit. Following the chain of parents will eventually take you back to the beginning of the project.

每个交付（除非是项目的第一个交付）总是有他的父交付，这样就说明了到当前的交付为止到底发生过什么。追索父交付链，可以将我们带回项目的起始点。

However, the commits do not form a simple list; git allows lines of development to diverge and then reconverge, and the point where two lines of development reconverge is called a "merge". The commit representing a merge can therefore have more than one

parent, with each parent representing the most recent commit on one of the lines of development leading to that point.

无论如何，这些交付的组织形式都不会是简单的；git 容许开发路线可以分道扬镳，也可以殊途同归，两条开发线路的结合点我们叫“合并”（merge）。表示合并的交付就等于有一个以上的父交付了，每个父交付表示每个开发线路发展到这里的最贴近的交付。

The best way to see how this works is using the gitk(1) command; running gitk now on a git repository and looking for merge commits will help understand how the git organizes history.

最好的查看这个机制的方法是用 gitk(1) 命令；现在在版本库中运行 gitk 命令，并查看一下那些合并交付会对你理解 git 是如何组织历史的很有帮助。

In the following, we say that commit X is "reachable" from commit Y if commit X is an ancestor of commit Y. Equivalently, you could say that Y is a descendant of X, or that there is a chain of parents leading from commit Y to commit X.

接着，我们说 交付 X 对于 交付 Y 来说是“可及的”， 如果 交付 X 是 交付 Y 的祖先的话。同样地，你可以说 Y 是 X 的一个后裔， 或者说存在一个从 Y 追索到 X 的世系族谱。

Understanding history: History diagrams 历史沿革示图

We will sometimes represent git history using diagrams like the one below. Commits are shown as "o", and the links between them with lines drawn with - / and \. Time goes left to right:

某些时候，我们会用下面的示图来描述 git 的历史。所有的交付用 "o" 表示， 联系他们各个发展线路之间画上 / 和 \。时间的推移是由左至右。

```
         o--o--o <-- Branch A
        /
o--o--o <-- master
        \
         o--o--o <-- Branch B
```

If we need to talk about a particular commit, the character "o" may be replaced with another letter or number.

如果需要具体地谈论某个交付，那么就用其他的字母或者是数字来替代 "o" 。

Understanding history: What is a branch? 理解历史：什么是分支

When we need to be precise, we will use the word "branch" to mean a line of development, and "branch head" (or just "head") to mean a reference to the most recent commit on a branch. In the example above, the branch head named "A" is a pointer to one particular commit, but we refer to the line of three commits leading up to that point as all being part of "branch A".

为准确起见，我们用 “分支” 这个词来表达开发的线路， 并且用 “分支头”（或者是 “头”）来表达一个分支中最新的交付。在上面的例子中，那个叫 “A” 的分支头是一个指向一个具体的交付的指针。但是我们指出，该线路上发展到这个点的三个交付全都是 “分支 A” 的组成部分。

However, when no confusion will result, we often just use the term "branch" both for branches and for branch heads.

不过， 在不至于产生混淆的前提下，我们常常只是用 “分支” 这个术语来表示分支和分支头。

Manipulating branches 操作分支

Creating, deleting, and modifying branches is quick and easy; here's a summary of the commands:

创建，删除，和更改分支都是很快而容易的；这里是这个命令的摘要：

- *git branch*
  - list all branches
  - 列举所有的分支
- *git branch <branch>*

    - create a new branch named <branch>, referencing the same point in history as the current branch
    - 创建一个新的分支，并引用当前分支作为同一历史沿革
- *git branch <branch> <start-point>*
    - create a new branch named <branch>, referencing <start-point>, which may be specified any way you like, including using a branch name or a tag name
    - 创建一个名叫 <branch> 的新分支，引用 <start-point>，它是可以任意指定的，可以是现存的分支的名称或者是标签的名称
- *git branch -d <branch>*
    - delete the branch <branch>; if the branch you are deleting points to a commit which is not reachable from the current branch, this command will fail with a warning.
    - 删除一个叫 <branch> 的分支；如果你要删除的这个分支所指向的当前分支中一个不可及的交付的话，那么命令将返回失败并作出提示
- *git branch -D <branch>*
    - even if the branch points to a commit not reachable from the current branch, you may know that that commit is still reachable from some other branch or tag. In that case it is safe to use this command to force git to delete the branch.
    - 尽管需要删除一个当前分支不可及的交付，但是你知道那个交付仍然可有其他的分支或者是标签可及。在这种情况下，用这个命令强制删除一个分支是安全的。
- *git checkout <branch>*
    - make the current branch <branch>, updating the working directory to reflect the version referenced by <branch>
    - 提取分支，也即是引用 <branch> 版本状态更新工作目录的内容
- *git checkout -b <new> <start-point>*
    - create a new branch <new> referencing <start-point>, and check it out.

- 引用 <start-point> 创建一个叫 <new> 的分支，并且将它提取出来。

The special symbol "HEAD" can always be used to refer to the current branch. In fact, git uses a file named "HEAD" in the .git directory to remember which branch is current:

特殊的标号 "HEAD" 总是被用作引用，指向当前分支。事实上，git 是用 .git 目录中的名叫 "HEAD" 的文件来记住那个是当前分支。

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

Examining an old version without creating a new branch 不通过创建新分支来调查旧版本

The git-checkout command normally expects a branch head, but will also accept an arbitrary commit; for example, you can check out the commit referenced by a tag:

git-checkout 命令按常规是抽取分支头的，但是也可以接受任意的交付；例如，你可以引用一个标签来进行提取。

```
$ git checkout v2.6.17
Note: moving to "v2.6.17" which isn't a local branch
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
HEAD is now at 427abfa... Linux v2.6.17
```

The HEAD then refers to the SHA1 of the commit instead of to a branch, and git branch shows that you are no longer on a branch:

此时，HEAD 将指向交付的 SHA1 来代替分支名称， git branch 命令表明你现在的项目状态不从属于任何一个分支：

```
$ cat .git/HEAD
427abfa28afedffadfca9dd8b067eb6d36bac53f
$ git branch
* (no branch)
  master
```

In this case we say that the HEAD is "detached".

这种情况，我们说 HEAD 是 “游离的”。

This is an easy way to check out a particular version without having to make up a name for the new branch. You can still create a new branch (or tag) for this version later if you decide to.

这是一个方便的途径，既可以提取某个版本，又避免了创建与命名一个新的分支。如果你愿意，你当然还可以以这个版本为标本来创建一个新分支（或者是标签）。

Examining branches from a remote repository 调查远程版本库上的分支

The "master" branch that was created at the time you cloned is a copy of the HEAD in the repository that you cloned from. That repository may also have had other branches, though, and your local repository keeps branches which track each of those remote branches, which you can view using the "-r" option to git-branch(1):

"master" 分支是你克隆版本库的时候创建的，它是你的克隆的那个远程版本库上的 HEAD 的复件。那么远程版本库上可能还存在其他的分支，故而你的本地版本库中也保留了那些远程分支的踪迹。你可以用 git branch(1) 命令加上 "-r" 选项来查看。

```
$ git branch -r
  origin/HEAD
  origin/html
  origin/maint
  origin/man
```

```
  origin/master
  origin/next
  origin/pu
  origin/todo
```

You cannot check out these remote-tracking branches, but you can examine them on a branch of your own, just as you would a tag:

你不可能将这些远程分支提取出来，但是你可以用一个你自己的分支来调查他们，你当他们是一个标签好了。

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

Note that the name "origin" is just the name that git uses by default to refer to the repository that you cloned from.

注意一下，"origin" 是 git 用来指向你的版本库的克隆来源的默认名称。

Naming branches, tags, and other references 命名分支，标签，与其他引用

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with "refs"; the names we've been using so far are actually shorthand:

分支，远程踪迹分支，与标签所有这些都是交付的引用。它们都被命名为以 "refs" 起头的带斜杠的路径名；实际上我们一直以来都用他们的速记名。

- The branch "test" is short for "refs/heads/test".
- The tag "v2.6.18" is short for "refs/tags/v2.6.18".
- "origin/master" is short for "refs/remotes/origin/master".
- 分支 "branch" 作为 "refs/heads/test" 的简称。
- 标签 "v2.6.18" 作为 "refs/tags/v2.6.18" 的简称
- "origin/master" 作为 "refs/remotes/origin/master" 的简称

The full name is occasionally useful if, for example, there ever exists a tag and a branch with the same name.

全名偶尔也会很有用，例如存在一个同名的分支和标签的时候。

(Newly created refs are actually stored in the .git/refs directory, under the path given by their name. However, for efficiency reasons they may also be packed together in a single file; see git-pack-refs(1)).

（新创建的引用实际是保存在 .git/refs 目录中，在他们的名字表明的路径下面。不过，由于效率的原因，它们几乎总是被打包到一个单独的文件中了；参考 git-pack-refs(1)）

As another useful shortcut, the "HEAD" of a repository can be referred to just using the name of that repository. So, for example, "origin" is usually a shortcut for the HEAD branch in the repository "origin".

还有一个很有用的捷径，版本库中的 "HEAD" 可以被引用为一个版本库的名称。作为例子，"origin" 常常作为 "origin" 版本库中的 HEAD 分支的快捷引用。

For the complete list of paths which git checks for references, and the order it uses to decide which to choose when there are multiple references with the same shorthand name, see the "SPECIFYING REVISIONS" section of git-rev-parse(1).

完整的路径令 git 可以查验引用，并在速记名相同的情况下决定准确的引用。参考 git rev-parse(1) 中 "SPECIFYING REVISIONS" 一段。

Updating a repository with git-fetch 用 git fetch 更新版本库

Eventually the developer cloned from will do additional work in her repository, creating new commits and advancing the branches to point at the new commits.

你向她克隆版本库的那个开发者，总归是要将新的工作加入到版本库中的，如创建了新的交付，将分支推进到新的交付点上。

The command "git fetch", with no arguments, will update all of the remote-tracking branches to the latest version found in her repository. It will not touch any of your own branches—not even the "master" branch that was created for you on clone.

命令 "git fetch"，不用带任何参数，本地版本库中所有远程关联分支都会被更新到远程版本库中对应的分支的最新版本。它不会触动你的专属分支的任何东西，甚至是克隆的时候为你创建的 "master" 分支。

- **译注：** *你可以直接修改 .git/config 文件将 master 分支的远程跟踪属性去掉。*

Fetching branches from other repositories 获取其他版本库的分支

You can also track branches from repositories other than the one you cloned from, using git-remote(1):

你同样可以跟踪一个并不是你的克隆源头的版本库上的分支。使用 git remote(1):

```
$ git remote add linux-nfs git://linux-nfs.org/pub/nfs-2.6.git
$ git fetch linux-nfs
* refs/remotes/linux-nfs/master: storing branch 'master' ...
  commit: bf81b46
```

New remote-tracking branches will be stored under the shorthand name that you gave "git-remote add", in this case linux-nfs:

新的远程跟踪分支将以速记名的形式保存，这个名称是由命令 "git remote add" 给定的，在这里是 linux-nfs:

```
$ git branch -r
linux-nfs/master
origin/master
```

If you run "git fetch <remote>" later, the tracking branches for the named <remote> will be updated.

如果在晚一点的时候再你运行 "git fetch <remote>"，那么这个跟踪分支就会被更新。

If you examine the file .git/config, you will see that git has added a new stanza:

如果你检查一下 .git/config 文件，你发现 git 增加了一个新的配置段。

```
$ cat .git/config
...
[remote "linux-nfs"]
        url = git://linux-nfs.org/pub/nfs-2.6.git
        fetch = +refs/heads/*:refs/remotes/linux-nfs/*
...
```

This is what causes git to track the remote's branches; you may modify or delete these configuration options by editing .git/config with a text editor. (See the "CONFIGURATION FILE" section of git-config(1) for details.)

这里的配置就是 git 进行远端分支跟踪的机制，你可以用文本编辑器修改或者是删除 .git/config 文件中的配置。（详情请参考 git config(1) 中的 "CONFIGURATION FILE"）

Chapter 2. Exploring git history 第二章. 检索 git 历史

Git is best thought of as a tool for storing the history of a collection of files. It does this by storing compressed snapshots of the contents of a file hierarchy, together with "commits" which show the relationships between these snapshots.

最好将 Git 当做是纪录文件历史变更集的工具。它压缩并逐层地保存文件的快照，收集这些 "交付"，就展示了文件快照之间的关系。

Git provides extremely flexible and fast tools for exploring the history of a project.

Git 在项目的历史搜索上提供了很强的检索弹性和快速工具。

We start with one specialized tool that is useful for finding the commit that introduced a bug into a project.

我们用一个特别点的工具作为开始，它对于如何发现一个将 bug 引入到项目中的交付很有用。

How to use bisect to find a regression 如何用平分来定位撤退

Suppose version 2.6.18 of your project worked, but the version at "master" crashes. Sometimes the best way to find the cause of such a regression is to perform a brute-force search through the project's history to find the particular commit that caused the problem. The git-bisect(1) command can help you do this:

假设项目中的 2.6.18 版本工作良好，但是 "master" 分支不稳定。某些时候，通过对项目历史进行暴力搜索，确定是那个交付造成的问题，这样我们就可以知道我们应该撤退到什么地方。git-bisect(1) 命令可以帮你做到这点：

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

If you run "git branch" at this point, you'll see that git has temporarily moved you in "(no branch)". HEAD is now detached from any branch and points directly to a commit (with commit id 65934…) that is reachable from "master" but not from v2.6.18. Compile and test it, and see whether it crashes. Assume it does crash. Then:

如果这个时候你运行 "git branch"，你会发现 git 已经将你移到 “无分支的状态”。HEAD 目前是处于游离的状态的，他不从属于任何的分支，并直接指向一个交付

（id 为 65934...），这个交付对于 "master" 分支来说是可及的，但对于 v2.6.18 来说就不是。现在可以编译和测试一下，看它是否会崩溃，如果是，那么：

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

checks out an older version. Continue like this, telling git at each stage whether the version it gives you is good or bad, and notice that the number of revisions left to test is cut approximately in half each time.

现在 git 提取出了更旧的版本，继续类似的步骤（编译和测试），将 git 每次给你的版本的测试结果用 *good* 或者是 *bad* 告诉 git，并且注意每次提取出来用于测试的版本都大致是截取一半的变更跨度。

- **译注：** git 每次都抽取大致从 good 至 bad 之间这个项目发展区间一半的那个版本进行测试，如果测试的结果是 good，那么就以当前被提取出来的这个版本到 bad 版本为区间再次进行平分提取，如此不断循环，很快就定位到项目的漏洞。

After about 13 tests (in this case), it will output the commit id of the guilty commit. You can then examine the commit with git-show(1), find out who wrote it, and mail them your bug report with the commit id. Finally, run

在这个例子中，经过 13 个测试之后，它终于定位到了那个出现问题的交付的 id 。你可以通过 git show(1) 命令来检查是谁写的这个东西，并将缺陷报告通过邮件告诉他们，记得写上那个交付的 id 。最后运行：

```
$ git bisect reset
```

to return you to the branch you were on before.

这样就返回到你原来所在的分支了。

Note that the version which git-bisect checks out for you at each point is just a suggestion, and you're free to try a different version if you think it would be a good idea. For example, occasionally you may land on a commit that broke something unrelated; run

提醒一下，git bisect 为你提取的版本只是一个建议性的东西，其实你可以尝试任何的版本，如果你认为这样是个好主意的话，偶尔可能会空降到某个造成问题的交付上去；运行：

```
$ git bisect visualize
```

which will run gitk and label the commit it chose with a marker that says "bisect". Choose a safe-looking commit nearby, note its commit id, and check it out with:

这个命令会运行 gitk 并将它选取的那个交付标贴上一个叫 "bisect" 的记号，就近这个标记选择一个看起来是安全的交付，纪录下它的 id，并将它提取出来：

```
$ git reset --hard fb47ddb2db...
```

then test, run "bisect good" or "bisect bad" as appropriate, and continue.

进行测试，并恰当地运行 "bisect good" 或 "bisect bad"，不断尝试。

Instead of "git bisect visualize" and then "git reset —hard fb47ddb2db…", you might just want to tell git that you want to skip the current commit:

要是不想去做 "git bisect visualize" 和 "git reset --hard fb47ddb2db" 。你只是想跳过 git 为你选择的那个交付的话：

```
$ git bisect skip
```

In this case, though, git may not eventually be able to tell the first bad one between some first skipped commits and a later bad commit.

如此一来，git 就无法得知在第一个跳过的交付到最后一个坏交付中那些交付是坏交付。

There are also ways to automate the bisecting process if you have a test script that can tell a good from a bad commit. See git-bisect(1) for more information about this and other "git bisect" features.

其实存在自动进行平分操作的方法，不过你需要一个脚本来告诉 git 好和坏的交付，参考 git bisect(1) 有更多的信息，这是 git bisect 的一个特性。

Naming commits 交付的称谓

We have seen several ways of naming commits already:

我们有一系列的方法来称呼交付：

- 40-hexdigit object name
- branch name: refers to the commit at the head of the given branch
- tag name: refers to the commit pointed to by the given tag (we've seen branches and tags are special cases of references).
- HEAD: refers to the head of the current branch
- 40 个十六进制字符的对象名
- 分支名： 指向给定的分支头的交付
- 标签名： 指向给定的标签的交付的引用（我们已经了解过分支和标签都是引用的特例）；
- 头: 指向当前分支的引用

There are many more; see the "" section of the git-rev-parse(1) man page for the complete list of ways to name revisions. Some examples:

还有更多的称谓； 参考 git rev-parse(1) 手册页中的 "SPECIFYING REVISIONS" 的章节，那里有有关“世系名称”的介绍，例如：

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

Recall that merge commits may have more than one parent; by default, ^ and ~ follow the first parent listed in the commit, but you can also choose:

回想一下，一个合并的交付可能有一个以上的父交付；默认地，^ 和 ~ 是他们的父辈列表中的一个父交付，当然你也可以显式地指出他们，譬如：

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

In addition to HEAD, there are several other special names for commits:

附带说明一下，对于 HEAD，对于它还有一系列特殊的称谓。

Merges (to be discussed later), as well as operations such as git-reset, which change the currently checked-out commit, generally set ORIG_HEAD to the value HEAD had before the current operation.

合并（将会在后面讨论到），有如 git reset 操作，均是改变当前提取的交付，他们通常都是将命名执行之前的 HEAD 的值保存到 ORIG_HEAD。

- **译注：** *这就等于是一服后悔药，如果你对 git reset，git merge 和 git pull 之类的命令的结果不满意，那么用下面的命令倒退回去就了：*

```
$ git reset --hard ORIG_HEAD
```

The git-fetch operation always stores the head of the last fetched branch in FETCH_HEAD. For example, if you run git fetch without specifying a local branch as the target of the operation

git fetch 操作总是将最后抓取的那个分支的头 (head) 保存到 FETCH_HEAD 中，如果你在运行 git fetch 的时候没有给定本地分支作为操作的目标的话。

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

the fetched commits will still be available from FETCH_HEAD.

抓取的交付将总是在 FETCH_HEAD 中。

When we discuss merges we'll also see the special name MERGE_HEAD, which refers to the other branch that we're merging in to the current branch.

当我们要讨论合并的时候，还将看到一个特殊称谓 MERGE_HEAD，它指向被我们刚刚合并进当前分支的另外一个分支。

The git-rev-parse(1) command is a low-level command that is occasionally useful for translating some name for a commit to the object name for that commit:

git rev-parse 命名是一个底层命令，我们偶尔要将某些称谓翻译成对象名的时候非常有用。

```
$ git rev-parse origin
e05db0fd4f31dde7005f075a84f96b360d05984b
```

Creating tags 创建标签

We can also create a tag to refer to a particular commit; after running

我们可以创建一个标签指向某个具体的交付，运行下面命令之后

```
$ git tag stable-1 1b2e1d63ff
```

You can use stable-1 to refer to the commit 1b2e1d63ff.

你可以看到 stable-1 指向交付 1b2e1d62ff。

This creates a "lightweight" tag. If you would also like to include a comment with the tag, and possibly sign it cryptographically, then you should create a tag object instead; see the git-tag(1) man page for details.

这是创建一个 “轻量” 标签。如果你希望创建一个带注释和加密签注的标签的话，那么你应该创建一个标签对象的实例，详情请参考 git tag(1) 的手册页。

Browsing revisions 浏览修订

The git-log(1) command can show lists of commits. On its own, it shows all commits reachable from the parent commit; but you can also make more specific requests:

git log(1) 命令列举交付列表。它将列举所有可及的父交付；并且你还可以做更多指定的查询：

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

And of course you can combine all of these; the following finds commits since v2.5 which touch the Makefile or any file under fs:

你还可以组合这些查询请求；下面的命令就是查询 v2.6 版本一来有关 Makefile 和 fs/ 目录下的修订的交付。

```
$ git log v2.5.. Makefile fs/
```

You can also ask git log to show patches:

你还可以让 git log 出示补丁：

```
$ git log -p
```

See the "—pretty" option in the git-log(1) man page for more display options.

参考 git log(1) 手册页中的 "--pretty" 选项，会有更多的显示选项。

Note that git log starts with the most recent commit and works backwards through the parents; however, since git history can contain multiple independent lines of development, the particular order that commits are listed in may be somewhat arbitrary.

需要注意的是， git log 从最新的交付开始向后回溯父交付；然而，git 的历史中包含了许多并行的相互独立的开发线路，具体的列举顺序可能需要我们作出一定的仲裁。

Generating diffs 生成差异

You can generate diffs between any two versions using git-diff(1):

你可以用 git diff(1) 命令生成两个版本之间的差异：

```
$ git diff master..test
```

That will produce the diff between the tips of the two branches. If you'd prefer to find the diff from their common ancestor to test, you can use three dots instead of two:

这将产生两个分支之间的差异提示。如果你期望发现他们两个的共同的祖先的差异，那你可以用三个点号代替两个点号。

```
$ git diff master...test
```

Sometimes what you want instead is a set of patches; for this you can use git-format-patch(1):

有时候，你希望取得一个实际的补丁集；你可以用 git format-patch(1)：

```
$ git format-patch master..test
```

will generate a file with a patch for each commit reachable from test but not from master.

这样会产生一个补丁文件，该文件是所有 test 可及的交付, 而不是 master。

Viewing old file versions 查看旧的文件版本

You can always view an old version of a file by just checking out the correct revision first. But sometimes it is more convenient to be able to view an old version of a single file without checking anything out; this command does that:

你当然可以通过提取文件的恰当的旧版本的方式来查看文件。不过有些时候查看单个文件可以有比提取出来更方便的办法，这个命令就是：

```
$ git show v2.5:fs/locks.c
```

Before the colon may be anything that names a commit, and after it may be any path to a file tracked by git.

冒号之前可以是任意一个交付的名称，冒号之后跟着已经纳入 git 跟踪的文件的路径。

Examples 实例

Counting the number of commits on a branch 统计一个分支中的交付数目

Suppose you want to know how many commits you've made on "mybranch" since it diverged from "origin":

假如你想知道你自己那个派生自 "origin" 的，叫 "mybranch" 的分支中有多少个交付：

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

Alternatively, you may often see this sort of thing done with the lower-level command git-rev-list(1), which just lists the SHA1's of all the given commits:

另外一个途径是你可以通过底层命令 git rev-list(1) 来列举这些东西，不过它列举的只是那些交付的 SHA1 的 id 好。

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

Check whether two branches point at the same history 检查两分支是否在同一历史时期

Suppose you want to check whether two branches point at the same point in history.

若然你想检查两个分支是否在同一历史时期。

```
$ git diff origin..master
```

will tell you whether the contents of the project are the same at the two branches; in theory, however, it's possible that the same project contents could have been arrived at by two different historical routes. You could compare the object names:

这样就得知一个项目的内容在两个分支中是否是相同的；理论上，项目可以经由不同的历史进程抵达相同的历史场面。你可以比较一下他们对应的对象名就知道了。

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

Or you could recall that the … operator selects all commits contained reachable from either one reference or the other but not both: so

或者你可以回顾一下，选取他们两者之中任何一个，而不是两者共同的可及的交付的操作： 如

```
$ git log origin...master
```

will return no commits when the two branches are equal.

当两个分支是相等的情况下，将不返回任何的交付。

Find first tagged version including a given fix 找到包含给定修正的第一个标签版本

Suppose you know that the commit e05db0fd fixed a certain problem. You'd like to find the earliest tagged release that contains that fix.

假如你知道交付 e05db0fd 已经修正了一个问题，你一定想知道包含这个交付的最早的标签发行版本是什么。

Of course, there may be more than one answer—if the history branched after commit e05db0fd, then there could be multiple "earliest" tagged releases.

当然，可能会存在不止一个的答案，如果历史在交付 e05db0fd 之后发生了分叉的话，那么就会存在多个“最早”的标签版本。

You could just visually inspect the commits since e05db0fd:

你可能需要一个可视化的方式来检查交付 e05db0fd 之后的情况：

```
$ gitk e05db0fd..
```

Or you can use git-name-rev(1), which will give the commit a name based on any tag it finds pointing to one of the commit's descendants:

又或者你可以用命令 git name-rev(1), 它可以给出你指定的那个交付的后裔中已经打过标签的那个交付。

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

The git-describe(1) command does the opposite, naming the revision using a tag on which the given commit is based:

git describe(1) 命令则是逆操作，找到以给定的交付为根源的那个标签的名字。

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

but that may sometimes help you guess which tags might come after the given commit.

它对你要推断在给定的交付之后有那个标签会出现很有帮助。

If you just want to verify whether a given tagged version contains a given commit, you could use git-merge-base(1):

要是你只是想核实一下某个标签是否包含某个交付的话，你可以用 git merge-base(1):

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

The merge-base command finds a common ancestor of the given commits, and always returns one or the other in the case where one is a descendant of the other; so the above output shows that e05db0fd actually is an ancestor of v1.5.0-rc1.

merge-base 命令查找给定的交付的共同的祖先，它总是返回一个结果，或者是返回他们两个之中的其中一个，而这一个是另外一个的祖先；从输出的结果来看，e05db0fd 的确是 v1.5.0-rc1 的祖先。

Alternatively, note that

注意另外一个方式

```
$ git log v1.5.0-rc1..e05db0fd
```

will produce empty output if and only if v1.5.0-rc1 includes e05db0fd, because it outputs only commits that are not reachable from v1.5.0-rc1.

当且仅当 e05db0fd 是 v1.5.0-rc1 的源头的时候，这个命令输出为空， 这个命令只会输出 v1.5.0-rc1 不可及的交付。

As yet another alternative, the git-show-branch(1) command lists the commits reachable from its arguments with a display on the left-hand side that indicates which arguments that commit is reachable from. So, you can run something like

命令 git-show-branch(1) 根据参数之间的可及性关系向左列举各个参数， 你可以试试

```
$ git show-branch e05db0fd v1.5.0-rc0 v1.5.0-rc1 v1.5.0-rc2
! [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
 ! [v1.5.0-rc0] GIT v1.5.0 preview
  ! [v1.5.0-rc1] GIT v1.5.0-rc1
   ! [v1.5.0-rc2] GIT v1.5.0-rc2
...
```

then search for a line that looks like

查找一下类似下面的一行

```
+ ++ [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
```

Which shows that e05db0fd is reachable from itself, from v1.5.0-rc1, and from v1.5.0-rc2, but not from v1.5.0-rc0.

这个显示表明 e05db0fd 对于 v1.5.0-rc1, v1.5.0-rc2 以及它自身来说是可及的，而对于 v1.5.0-rc0 来说则不是。

- **译注：** 注意该行前面的加号 "+" 与所对应的 v1.5.0-rc0， v1.5.0-rc1, v1.5.0-rc2 的列对齐关系。

Showing commits unique to a given branch 显示仅属于某个分支的交付

Suppose you would like to see all the commits reachable from the branch head named "master" but not from any other head in your repository.

假如你想知道在你的版本库中，那些交付仅仅是可及分支 "master"， 而对其他的分支没有可及性。

We can list all the heads in this repository with git-show-ref(1):

我们可以用 git-show-ref 命令先列出版本库中所有的分支头。

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

We can get just the branch-head names, and remove "master", with the help of the standard utilities cut and grep:

我们可以用标准的辅助工具 cut 和 grep， 从分支列表中去掉 "master" 。

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

And then we can ask to see all the commits reachable from master but not from these other heads:

接着我们查看那些交付对 "master" 是可及的， 而对其他的分支没有可及性。

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

Obviously, endless variations are possible; for example, to see all commits reachable from some head but not from any tag in the repository:

显然，上述命令是可以带无限的参数的； 譬如， 查看对于某些分支头可及，但是对于任何的标签都不可及的所有交付的命令如下：

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

(See git-rev-parse(1) for explanations of commit-selecting syntax such as —not.)

（参看 git-rev-parse(1)， 那里有关于交付选取的语法的解释， 例如 --not）

Creating a changelog and tarball for a software release 为软件的发行制作变更日志和压缩包

The git-archive(1) command can create a tar or zip archive from any version of a project; for example:

git archive(1) 命令可以为项目的任何版本创建压缩包； 例如：

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

will use HEAD to produce a tar archive in which each filename is preceded by "project/".

这样将以 HEAD 为版本创建压缩包，并且每个文件名前面都将加上前缀。

If you're releasing a new version of a software project, you may want to simultaneously make a changelog to include in the release announcement.

如果你要发行一个新的软件版本， 你可能希望同时在发行声明中包含软件的变更日志。

Linus Torvalds, for example, makes new kernel releases by tagging them, then running:

譬如， Linus Torvalds 制作打过版本标签的新的内核版本时，就运行：

```
$ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7
```

where release-script is a shell script that looks like:

这里的 release-scritp 是一个 shell 脚本， 它大概会象下面的样子：

```
stable="$1"
last="$2"
new="$3"
```

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

and then he just cut-and-pastes the output commands after verifying that they look OK.

并且他仅仅是在命令运行过之后， 验证一下输出的东西，再剪切－粘贴一下。

Finding commits referencing a file with given content 寻找一个指向包含给定内容的文件的交付

Somebody hands you a copy of a file, and asks which commits modified a file such that it contained the given content either before or after the commit. You can find out with this:

某人给了你一个文件的拷贝， 并请求得到修改过这个文件的那些个交付，不过，他甚至不知道这个文件现在所包含的内容是否已经被提交过。 你可以这样地查找：

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

Figuring out why this works is left as an exercise to the (advanced) student. The git-log(1), git-diff-tree(1), and git-hash-object(1) man pages may prove helpful.

思考一下为什么这个命令可以做到这样的结果，并将这个问题留给（高级的）学生作为练习。 git-log(1)， git-diff-tree(1), 与 git-hash-object(1) 的手册页可能会提供有用的帮助。

Chapter 3. Developing with git 第三章. 用 git 进行研发

Telling git your name 告诉 git 你的名字

Before creating any commits, you should introduce yourself to git. The easiest way to do so is to make sure the following lines appear in a file named .gitconfig in your home directory:

在创建任何交付之前，你应该先将你自己介绍给 git。最容易的方法就是在你的家目录中创建一个叫 .gitconfig 的文件，并写入如下的配置项。

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

(See the "CONFIGURATION FILE" section of git-config(1) for details on the configuration file.)

（在 git-config(1) 中参考 "CONFIGURATION FILE" 那一节得到有关配置文件的帮助）

Creating a new repository 创建新的版本库

Creating a new repository from scratch is very easy:

从零开始创建新的版本库是很容易的：

```
$ mkdir project
$ cd project
$ git init
```

If you have some initial content (say, a tarball):

如果你已经有一个原始的内容的话 （意思是你已经有一个压缩包了）：

```
$ tar xzvf project.tar.gz
$ cd project
```

```
$ git init
$ git add . # include everything below ./ in the first commit:
$ git commit
```

How to make a commit 如何制作一个交付

Creating a new commit takes three steps:

1. Making some changes to the working directory using your favorite editor.
2. Telling git about your changes.
3. Creating the commit using the content you told git about in step 2.

创建一个新的交付有三个步骤：

1. 在你的工作目录中用你喜欢的编辑器做了某些变更。
2. 告诉 git 你所做的变更。
3. 将第二步中你告诉 git 的变更内容创建一个交付。

In practice, you can interleave and repeat steps 1 and 2 as many times as you want: in order to keep track of what you want committed at step 3, git maintains a snapshot of the tree's contents in a special staging area called "the index."

事实上，第一步 和 第二步 在任何时候都可以交替和重复地进行： 为了保持着对你打算在第三步提交的内容进行跟踪， git 在一个特殊的阶段性的区域内保留了你的工作目录树的内容快照，这个区域叫 "索引"。

At the beginning, the content of the index will be identical to that of the HEAD. The command "git diff —cached", which shows the difference between the HEAD and the index, should therefore produce no output at that point.

在开始的时候， 索引中的内容跟 HEAD 中的内容是一致的。对于命令 `git diff --cached`，它是显示 HEAD 与 索引之间的差异的，在这个时候，它应该没有任何的输出。

Modifying the index is easy:

更改索引是很容易的：

To update the index with the new contents of a modified file, use

用一个编辑过的文件的内容去刷新索引，用：

```
$ git add path/to/file
```

To add the contents of a new file to the index, use

用一个新的文件的内容去加入到索引中，也是用：

```
$ git add path/to/file
```

To remove a file from the index and from the working tree,

从索引和工作目录中删除一个文件，

```
$ git rm path/to/file
```

After each step you can verify that

上述的每个操作之后，你都可以验证一下：

```
$ git diff --cached
```

always shows the difference between the HEAD and the index file—this is what you'd commit if you created the commit now—and that

无论如何都应该显示一下 HEAD 和索引文件之间的差异 -- 这其实就是你即将要提交的交付中的东西

```
$ git diff
```

shows the difference between the working tree and the index file.

上面的命令显示工作树与索引之间的差异。

Note that "git-add" always adds just the current contents of a file to the index; further changes to the same file will be ignored unless you run git-add on the file again.

注意， "git-add" 仅仅是将当前文件的内容加入索引； 在此之后的同一个文件的变更将会被忽略，除非你再次对该文件应用 git add 命令。

When you're ready, just run

当你都准备好了之后，只需运行

```
$ git commit
```

and git will prompt you for a commit message and then create the new commit. Check to make sure it looks like what you expected with

git 将在创建新的交付时向你提示交付信息。要检查你要期望的东西的话，运行

```
$ git show
```

As a special shortcut,

所为一个特殊的快捷方式，

```
$ git commit -a
```

will update the index with any files that you've modified or removed and create a commit, all in one step.

它将用你所修改和删除的内容去刷新索引，并创建一个新的交付，一步完成。

A number of commands are useful for keeping track of what you're about to commit:

一些命令对跟踪你提交的东西非常有用：

```
$ git diff --cached # difference between HEAD and the index; what
```

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

You can also use git-gui(1) to create commits, view changes in the index and the working tree files, and individually select diff hunks for inclusion in the index (by right-clicking on the diff hunk and choosing "Stage Hunk For Commit").

你还可以用 git-gui(1) 来创建交付，你将看到在索引和工作树文件中的变更，并可以个别地选取在索引中的变更块进行提交（在变更块上右击鼠标并选择“Stage Hunk For Commit”）。

Creating good commit messages 写好交付信息

Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. Tools that turn commits into email, for example, use the first line on the Subject line and the rest of the commit in the body.

尽管不是必须的，但是好的交付信息应该是这样，用一个短句（少于 50 个字符）开头，接着跟一个空行，接下来就可以写更多的说明了。相关的工具将交付转化成电子邮件的时候，以第一行作为邮件的标题，交付的余下部分将作为邮件的正文。

- **译注：** 工作树中的 ".git/COMMIT_EDITMSG" 文件是默认的交付信息文件，如果你的 `git commit` 命令没有带 -m 参数，那么 git 将会为你打开系统默认编辑器编辑这个文件作为交付信息。

Ignoring files 忽略文件

A project will often generate files that you do not want to track with git. This typically includes files generated by a build process or temporary backup files made by your editor.

Of course, not tracking files with git is just a matter of not calling "git-add" on them. But it quickly becomes annoying to have these untracked files lying around; e.g. they make "git add ." practically useless, and they keep showing up in the output of "git status".

一个项目总是经常产生一些你不想让 git 来跟踪的文件。典型的就是编译过程中的中间文件，或者是编辑器的临时备份文件等等。当然了，不跟踪某个文件只不过是不用 "git add" 命令将它加入 git 而已。不过你很快会不胜其扰；譬如你不小心用 "git add ." 命令将它加入了 git，并且这些文件总是在 "git status" 命令的输出中出现。

You can tell git to ignore certain files by creating a file called .gitignore in the top level of your working directory, with contents such as:

你可以在工作树的顶层目录中创建一个叫 .gitignore 的文件来告诉 git 那些文件是要忽略的，文件的内容大致如下：

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

See gitignore(5) for a detailed explanation of the syntax. You can also place .gitignore files in other directories in your working tree, and they will apply to those directories and their subdirectories. The .gitignore files can be added to your repository like any other files (just run git add .gitignore and git commit, as usual), which is convenient when the exclude patterns (such as patterns matching build output files) would also make sense for other users who clone your repository.

参考 gitignore(5) 得到更多的语法的解析。你还可以将 .gitignore 文件放到工作树的其他目录下，这样它将影响它所处的目录以及下级目录。.gitignore 文件当然也可以像一般的文件那样纳入为 git 的跟踪中（只需要运行 git add .gitignore 和 git commit，就像处理一般的文件那样将行了）。如果存在 "排除匹配" 的情况（譬如

排除跟踪那些编译中间文件），为了对其他克隆你的版本库的用户方便起见，你应该给他们一点提示。

- **译注：**关于 *排除匹配* 可以参考 gitignore，排除匹配的表述文件是与 git 一起安装的，例如可能在 /usr/share/git-core/templates/info/exclude。

If you wish the exclude patterns to affect only certain repositories (instead of every repository for a given project), you may instead put them in a file in your repository named .git/info/exclude, or in any file specified by the core.excludesfile configuration variable. Some git commands can also take exclude patterns directly on the command line. See gitignore(5) for the details.

如果你希望排除匹配只是影响某些指定的版本库（而不是一个项目的每个版本库），你可以用在你的版本库中 .git/info/exclude 文件来取代全局的排除匹配，或者是针对任何一个文件在 git 的配置文件中用 core.excludesfile 配置参数来指定。某些 git 命令是可以做命令行参数中直接接受排除匹配的。 详情参考 gitignore(5)。

How to merge 如何合并

You can rejoin two diverging branches of development using git-merge(1):

你可以将两个研发的分支用 git-merge(1) 合并在一起：

```
$ git merge branchname
```

merges the development in the branch "branchname" into the current branch. If there are conflicts—for example, if the same file is modified in two different ways in the remote branch and the local branch—then you are warned; the output may look something like this:

这是合并另外一个分支 "branchname" 到当前分支。如果同一个文件无论是在远程分支，还是在本地分支上的发展进程中被修改过，那么将出现冲突，此时你会得到一个提示，提示的输出大概是下面的样子：

```
$ git merge next
 100% (4/4) done
```

```
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict markers are left in the problematic files, and after you resolve the conflicts manually, you can update the index with the contents and run git commit, as you normally would when creating a new file.

冲突标志会被保留在文件中，并且当你手动解决冲突之后，你可以用新的内容刷新索引，并运行 git commit，这类似你创建一个新的文件之后所做的那样。

If you examine the resulting commit using gitk, you will see that it has two parents, one pointing to the top of the current branch, and one to the top of the other branch.

如果你用 gitk 来解决冲突，你将会看到解决冲突的这个交付有两个父交付，一个指向当前分支的顶部，另一个指向两个另外一个分支的顶部。

Resolving a merge 解决合并冲突

When a merge isn't resolved automatically, git leaves the index and the working tree in a special state that gives you all the information you need to help resolve the merge.

当合并冲突不能自动解决时，git 将会让索引和工作树都保持在你可以得到所有信息的特殊状态，此时你需要协助 git 解决合并冲突。

Files with conflicts are marked specially in the index, so until you resolve the problem and update the index, git-commit(1) will fail:

冲突文件会在索引中特别被标记起来，直到你解决这些冲突并刷新索引为止，否则 git commit(1) 将出错：

```
$ git commit
file.txt: needs merge
```

Also, git-status(1) will list those files as "unmerged", and the files with conflicts will have conflict markers added, like this:

并且，git-status(1) 将列举出 "unmerged" 的那些文件，文件中也会被加入冲突的标志，它看起来大致是这样：

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

All you need to do is edit the files to resolve the conflicts, and then

无论如何你都需要编辑这些文件，解决冲突，之后

```
$ git add file.txt
$ git commit
```

Note that the commit message will already be filled in for you with some information about the merge. Normally you can just use this default message unchanged, but you may add additional commentary of your own if desired.

注意此时的交付信息中，总是会被填充一些有关合并的信息。正常情况下，你可以用它作为默认的信息而不要修改它，不过你当然可以加上你自己希望的注释。

The above is all you need to know to resolve a simple merge. But git also provides more information to help resolve conflicts:

以上你是必须知道的有关简单合并的知识，不过 git 会提供更多的信息帮助你解决冲突：

Getting conflict-resolution help during a merge 在合并过程中取得冲突解决帮助

All of the changes that git was able to merge automatically are already added to the index file, so git-diff(1) shows only the conflicts. It uses an unusual syntax:

所有 git 可以自动合并的东西都会被加入到索引中，而且 git-diff(1) 只会显示冲突，它使用一种特殊的语法来表示：

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

Recall that the commit which will be committed after we resolve this conflict will have two parents instead of the usual one: one parent will be HEAD, the tip of the current branch; the other will be the tip of the other branch, which is stored temporarily in MERGE_HEAD.

回想一下，你解决了冲突之后提交的交付将有两个父交付：一个是 HEAD，当前分支的顶端；另一个是另外一个分支的顶端，它临时性地保存为 MERGE_HEAD 。

During the merge, the index holds three versions of each file. Each of these three "file stages" represents a different version of the file:

合并的过程中，索引为每个冲突的文件保持三个版本，这三个 "文件阶段（file stages）" 表示三个文件的不同版本：

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

When you ask git-diff(1) to show the conflicts, it runs a three-way diff between the conflicted merge results in the work tree with stages 2 and 3 to show only hunks whose contents come from both sides, mixed (in other words, when a hunk's merge results come only from stage 2, that part is not conflicting and is not shown. Same for stage 3).

当你用 git diff(1) 来显示冲突的时候，它将根据你当前工作目录中的版本与文件阶段 2 和 3 进行三路差异运算之后，向你显示一个差异内容块，此内容混合了来自两个方面的差异（换一个讲法，就是工作树中的内容与第二个文件阶段中的内容合并，再去合并第三个文件阶段中的内容，但是冲突块仅仅是显示了这第二次合并中有冲突的地方，而没有冲突的地方则不显示）。

The diff above shows the differences between the working-tree version of file.txt and the stage 2 and stage 3 versions. So instead of preceding each line by a single "+" or "-", it now uses two columns: the first column is used for differences between the first parent and the working directory copy, and the second for differences between the second parent and the working directory copy. (See the "COMBINED DIFF FORMAT" section of git-diff-files(1) for a details of the format.)

上述的输出显示了 file.txt 文件在工作树中的版本对阶段 2（stage 2）与阶段 3（stage 3）中的差异，并且在每行中加上 "+" 或者 "-" 的前缀， 分两列排列： 第一列用于标记第一个父交付对工作树的差异，第二列用于表示第一个父交付对工作树的差异。（参考 git-diff-files(1) 中的 "COMBINED DIFF FORMAT" 段，取得详细信息）

After resolving the conflict in the obvious way (but before updating the index), the diff will look like:

解决了冲突之后（但未刷新索引之前）， 差异输出类似这样：

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

This shows that our resolved version deleted "Hello world" from the first parent, deleted "Goodbye" from the second parent, and added "Goodbye world", which was previously absent from both.

这个输出显示我们的解决方案是删除来自第一个父交付的 "Hello world"，删除来自第二个父交付中的 "Goodbye"，加入了 "Goodbye world"，这是在两个父交付中都从来没有出现过的内容。

Some special diff options allow diffing the working directory against any of these stages:

下面是一个特殊的差异选项，比较工作树对其他的文件阶段的差异：

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

The git-log(1) and gitk(1) commands also provide special help for merges:

git-log(1) 与 gitk(1) 命令会有关于合并的专题帮助：

```
$ git log --merge
$ gitk --merge
```

These will display all commits which exist only on HEAD or on MERGE_HEAD, and which touch an unmerged file.

如此将显示所有仅存于 HEAD 或者是 MERGE_HEAD 中的所有涉及未合并的文件的交付。

You may also use git-mergetool(1), which lets you merge the unmerged files using external tools such as Emacs or kdiff3.

你还可以用 git-mergetool(1) 命令，它容许你使用第三方合并工具进行合并操作，譬如 Emacs 或者是 kdiff3。

Each time you resolve the conflicts in a file and update the index:

每当你解决了冲突之后就应刷新索引：

```
$ git add file.txt
```

the different stages of that file will be "collapsed", after which git-diff will (by default) no longer show diffs for that file.

之后，其他的差异阶段文件将被丢弃， git-diff 命令将显示没有任何差异。

Undoing a merge 撤销合并

If you get stuck and decide to just give up and throw the whole mess away, you can always return to the pre-merge state with

如果你在合并过程中遇到麻烦，并且打算放弃和离开混乱处境，你总是可以回到合并之前的状态

```
$ git reset --hard HEAD
```

Or, if you've already committed the merge that you want to throw away,

又或者，你已经将合并提交了，但是你想扔掉它，

```
$ git reset --hard ORIG_HEAD
```

However, this last command can be dangerous in some cases—never throw away a commit you have already committed if that commit may itself have been merged into another branch, as doing so may confuse further merges.

可是，最后一个命令在某些情况下是危险的，它是扔掉一个已经提交了的合并，如果这个合并交付本身是曾经合并过其他的分支的话，这样做将会给今后的合并带来混乱。

Fast-forward merges 快进合并

There is one special case not mentioned above, which is treated differently. Normally, a merge results in a merge commit, with two parents, one pointing at each of the two lines of development that were merged.

有一个特殊的情况上面没有提及到的，这被看作是一个特例。正常情况下，合并的结果是一个合并交付，带两个父交付，分别指向被合并的两个研发分支。

However, if the current branch is a descendant of the other—so every commit present in the one is already contained in the other—then git just performs a "fast forward"; the head of the current branch is moved forward to point at the head of the merged-in branch, without any new commits being created.

不过，要是当前分支的东西都是另外一个分支中的东西的后裔的时候，git 只会进行 “快进” 操作；也就是说，它只会将当前的分支头快进到合并过后的分支头，而不会创建一个新的交付。

Fixing mistakes 修复失误

If you've messed up the working tree, but haven't yet committed your mistake, you can return the entire working tree to the last committed state with

要是工作树已经被你搞得一团糟了，但是你还没有提交过错误的东西的话，你可以整体地将工作树撤回到最后一个交付的状态

```
$ git reset --hard HEAD
```

If you make a commit that you later wish you hadn't, there are two fundamentally different ways to fix the problem:

要是你做了一个交付，但是你后来又想放弃它，那么基本上有两个途径让你修复问题：

- 1. You can create a new commit that undoes whatever was done by the old commit. This is the correct thing if your mistake has already been made public.
- 2. You can go back and modify the old commit. You should never do this if you have already made the history public; git does not normally expect the "history" of a project to change, and cannot correctly perform repeated merges from a branch that has had its history changed.
- 1. 你可以创建一个新的交付来撤销你旧的交付的东西。这个才是正确的方法，如果你的错误已经发布出去了。
- 2. 你可以回到那个老的交付中并修改它。可是当你的版本库历史已经发布出去之后，你绝对不应该这样做；通常地 git 不会设想项目的 “历史” 是会出现改变的。同时也不能够正确地对那些历史发生过改变的分支合并进行重新整合。

Fixing a mistake with a new commit 修复一个新的交付中的失误

Creating a new commit that reverts an earlier change is very easy; just pass the git-revert(1) command a reference to the bad commit; for example, to revert the most recent commit:

创建一个新的交付去逆转一个最近的交付的变更是很容易的；通过 git-revert(1) 命令指向一个坏交付就行；例如，逆转一个最新的交付：

```
$ git revert HEAD
```

This will create a new commit which undoes the change in HEAD. You will be given a chance to edit the commit message for the new commit.

这将创建一个新交付去撤销在 HEAD 中的变更。这样就给了你一个机会编辑新交付中的注释。

You can also revert an earlier change, for example, the next-to-last:

你可以一个接一个地逆转更早的变化，例如：

```
$ git revert HEAD^
```

In this case git will attempt to undo the old change while leaving intact any changes made since then. If more recent changes overlap with the changes to be reverted, then you will be asked to fix conflicts manually, just as in the case of resolving a merge.

在这种情况下，每当 git 离开过一个原封的变更时，它将试图将撤销的变更应用到其后的项目历史中。如果有多个变更重叠，你可能会被要求解决冲突，正如你要解决合并冲突那样。

Fixing a mistake by rewriting history 通过重写历史来修复失误

If the problematic commit is the most recent commit, and you have not yet made that commit public, then you may just destroy it using git-reset.

如果一个有问题的交付是最新的交付，并且你还没有将它发布出去，那么你可以用 git-reset 来销毁它。

Alternatively, you can edit the working directory and update the index to fix your mistake, just as if you were going to create a new commit, then run

另一个途径是编辑工作目录并刷新索引来修复你的失误，就像你创建一个新的交付那样，运行

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

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first.

它将用你更正后的新交付去覆盖旧的交付，首先你有机会改变旧的交付信息。

Again, you should never do this to a commit that may already have been merged into another branch; use git-revert(1) instead in that case.

重申一次，你绝对不应该在这个交付已经并合并经其他的分支的情况下这样做；此时你应该用 git-revert(1) 来处理失误。

It is also possible to replace commits further back in the history, but this is an advanced topic to be left for another chapter.

它还可以覆盖历史更久远的交付，不过这个是一个高级话题，留待其他的章节讨论。

Checking out an old version of a file 提取一个文件的旧版本

In the process of undoing a previous bad change, you may find it useful to check out an older version of a particular file using git-checkout(1). We've used git-checkout before to switch branches, but it has quite different behavior if it is given a path name: the command

在处理撤销以前坏变更的过程中，你可能已经认识到 git-checkout(1) 用于提取一个具体的文件的旧版本的有用性。以前我们用 git-checkout 来在旧的版本之间切换，不过如果你给出路径的名称，那么这个命令的表现会完全不同。

```
$ git checkout HEAD^ path/to/file
```

replaces path/to/file by the contents it had in the commit HEAD^, and also updates the index to match. It does not change branches.

它将以交付 HEAD^ 中的内容来覆盖 path/to/file 文件，并且同时恰当地刷新索引。它不会改变分支。

If you just want to look at an old version of the file, without modifying the working directory, you can do that with git-show(1):

假如你只是想看看一个文件的旧版本是什么样子，而不想触动工作树的话，你可以用 git-show(1) 命令这样做：

```
$ git show HEAD^:path/to/file
```

which will display the given version of the file.

它会显示给定的版本的文件内容。

Temporarily setting aside work in progress 临时放下手头上的工作

While you are in the middle of working on something complicated, you find an unrelated but obvious and trivial bug. You would like to fix it before continuing. You can use git-stash(1) to save the current state of your work, and after fixing the bug (or, optionally after doing so on a different branch and then coming back), unstash the work-in-progress changes.

当你正处于一个复杂的工作过程当中，但忽然发现一个与当前工作无关的，而又很明显的漏洞的时候。你也许希望先修复这个漏洞，再继续你手头上的工作。你可以用 git-stash(1) 将你当前工作树的状态先保存起来，待修复了那个漏洞之后（或者，在另外的分支上完成这件事之后再回来），再取回你刚才手头上的工作接着做。

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

This command will save your changes away to the stash, and reset your working tree and the index to match the tip of your current branch. Then you can make your fix as usual.

这个命令将你当前工作树中的变更保存到临时的空间（stash）中，并以你当前的分支状态重置工作树和索引。那你就可以进行漏洞的修复工作了。

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

After that, you can go back to what you were working on with git stash apply:

之后，你就可以用 git stash apply 命令回到原来的工作状态：

```
$ git stash apply
```

Ensuring good performance 确保好的性能

On large repositories, git depends on compression to keep the history information from taking up too much space on disk or in memory.

在大型项目中， git 依靠对历史信息进行压缩来利用磁盘空间和内存空间。

This compression is not performed automatically. Therefore you should occasionally run git-gc(1):

不过压缩不是自动的，你应该时不时运行一下 git-gc(1):

```
$ git gc
```

to recompress the archive. This can be very time-consuming, so you may prefer to run git-gc when you are not doing other work.

它会重新压缩项目。这可能是很费时的工作，你可以在完成其他的事情之后才做这件事。

Ensuring reliability 确保伸缩性

Checking the repository for corruption 检查版本的损坏

The git-fsck(1) command runs a number of self-consistency checks on the repository, and reports on any problems. This may take some time. The most common warning by far is about "dangling" objects:

git-fsck(1) 命令在版本库中执行一系列的一致性性检查，并报告错误。这可能需要一些时间，最常见的警告就是关于“悬空”对象：

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

```
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
...
```

Dangling objects are not a problem. At worst they may take up a little extra disk space. They can sometimes provide a last-resort method for recovering lost work—see the section called “Dangling objects” for details.

悬空对象不是一个问题。它们只是占多了一点的磁盘空间。某些时候，它们是恢复丢失的工作的救命稻草。详情请参考 "悬空对象" 一节。

Recovering lost changes 恢复丢失的变更

Reflogs 引用日志

Say you modify a branch with git-reset(1) —hard, and then realize that the branch was the only reference you had to that point in history.

要是你用 git-reset(1) --hard 来恢复了你的分支，接着你才醒悟到那个分支是你在版本库中所处的历史时刻的唯一引用。

Fortunately, git also keeps a log, called a "reflog", of all the previous values of each branch. So in this case you can still find the old history using, for example,

幸运的是，git 总是保存一个日志，叫“引用日志”，它保存了每个分支以前的值。在这种情况下，你仍然可以将旧历史找回来，例如：

```
$ git log master@{1}
```

This lists the commits reachable from the previous version of the "master" branch head. This syntax can be used with any git command that accepts a commit, not just with git log. Some other examples:

他将列举所有对于 "master" 分支头具有可及性的交付。这种形式的语法可以用于任何的 git 命令而不仅仅是 git log。举些例子：

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

A separate reflog is kept for the HEAD, so

有关 HEAD 的引用日志，则是

```
$ git show HEAD@{"1 week ago"}
```

will show what HEAD pointed to one week ago, not what the current branch pointed to one week ago. This allows you to see the history of what you've checked out.

它将显示 HEAD 一周之前指向什么，而不是当前分支指向什么。这就容许你看到如果你要提取历史的话将得到什么。

The reflogs are kept by default for 30 days, after which they may be pruned. See git-reflog(1) and git-gc(1) to learn how to control this pruning, and see the "SPECIFYING REVISIONS" section of git-rev-parse(1) for details.

应用日志默认会被保存 30 天，之后它们就会被剪裁掉。参考 git-reflog(1) 与 git-gc(1) 学习如何控制剪裁，详情查看 git-rev-parse(1) 中的 "SPECIFYING REVISION" 一节。

Note that the reflog history is very different from normal git history. While normal history is shared by every repository that works on the same project, the reflog history is not shared: it tells you only about how the branches in your local repository have changed over time.

注意，引用日志完全不同与正式的 git 历史记录。正式的历史记录在同一个项目中的任何一个版本库中都被共享，而引用日志是不共享的：它仅仅告诉你在某段时间内你的本地版本库中的分支如何变化。

Examining dangling objects 检验悬空对象

In some situations the reflog may not be able to save you. For example, suppose you delete a branch, then realize you need the history it contained. The reflog is also deleted; however, if you have not yet pruned the repository, then you may still be able to find the lost commits in the dangling objects that git-fsck reports. See the section called “Dangling objects” for the details.

某些状态下引用日志（reflogs）并不能打救你。譬如，你删除了一个分支，接着你醒悟到你还需要那个分支上的历史记录。此时引用日志也被删除了；不过如果你还没有剪裁版本库的话，你仍然可以在悬空对象中找回那些丢失的交付，git-fsck 会给你报告悬空对象的列表。详情参考 “悬空对象” 一节。

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

You can examine one of those dangling commits with, for example,

你可以查验那些悬空交付，例如，

```
$ gitk 7281251ddd --not --all
```

which does what it sounds like: it says that you want to see the commit history that is described by the dangling commit(s), but not the history that is described by all your existing branches and tags. Thus you get exactly the history reachable from that commit that is lost. (And notice that it might not be just one commit: we only report the "tip of the line" as being dangling, but there might be a whole deep and complex commit history that was dropped.)

这命令类似这样表达：你希望查看那些注明为悬空对象的交付的历史，而不是你现在拥有的分支或者是标签的历史。然而你清楚地知道那些交付的历史可及性已经丢失了。（并且需要注意，它不一定仅仅是一个交付：我们只是报告悬空对象链的顶端交付，可能有整个有连带关系的很深和很复杂的交付历史被抛弃了。）

If you decide you want the history back, you can always create a new reference pointing to it, for example, a new branch:

如果你决定恢复那部分历史，你可以创建一个新的引用指向它，例如创建一个新的分支：

```
$ git branch recovered-branch 7281251ddd
```

Other types of dangling objects (blobs and trees) are also possible, and dangling objects can arise in other situations.

其他类型的悬空对象（单元块和数）也是可能存在的，悬空对象有可能在其他的情况下出现的。

Chapter 4. Sharing development with others 第四章. 与他人协同研发

Getting updates with git-pull 用 git-pull 取得更新

After you clone a repository and make a few changes of your own, you may wish to check the original repository for updates and merge them into your own work.

你克隆了一个版本库，并且在你自己的版本库中做了一定的工作之后，你可能希望检查源头版本库的更新情况，并将那些更新合并到你的工作中来。

We have already seen how to keep remote tracking branches up to date with git-fetch(1), and how to merge two branches. So you can merge in changes from the original repository's master branch with:

我们已经看过如何用 git-fetch(1) 命令来保持对远程分支的更新跟踪，如何合并两个分支。那么你可以这样合并源头版本库上的主分支（master）中的变化：

```
$ git fetch
$ git merge origin/master
```

However, the git-pull(1) command provides a way to do this in one step:

不过，git-pull(1)命令提供了一个一步到位的方法：

```
$ git pull origin master
```

In fact, if you have "master" checked out, then by default "git pull" merges from the HEAD branch of the origin repository. So often you can accomplish the above with just a simple

事实上，如果你已经切换到 "master" 分支的话，"git pull" 命令默认就是合并源头版本库中的 HEAD。所以你经常可以通过下面简单的命令来完成上述的操作

```
$ git pull
```

More generally, a branch that is created from a remote branch will pull by default from that branch. See the descriptions of the branch.<name>.remote and branch.<name>.merge options in git-config(1), and the discussion of the —track option in git-checkout(1), to learn how to control these defaults.

更广泛地，一个由远程版本库上创建的分支将作为本地对应分支的默认拉入分支。参考 git-config(1) 中有关 branch.<name>.remote 和 branch.<name>.merge 中的有关说明，以及 git-checkout(1) 中有关 --track 的讨论，学习如何控制那些默认操作。

In addition to saving you keystrokes, "git pull" also helps you by producing a default commit message documenting the branch and repository that you pulled from.

为了节省你打字的功夫，"git pull" 已经帮你自动产生了交付信息来说明你拉入的版本库和分支了。

(But note that no such commit will be created in the case of a fast forward; instead, your branch will just be updated to point to the latest commit from the upstream branch.)

（不过当拉入操作是快进合并的时候，则不会新的交付产生；你的分支只是被推进到分支发展的进程的最新的那个交付。）

The git-pull command can also be given "." as the "remote" repository, in which case it just merges in a branch from the current repository; so the commands

git-pull 命令可以用一个 "." 作为 "远程的" 版本库，在这种情况下则是合并当前版本库的另外一个分支；所以如下命令

```
$ git pull . branch
$ git merge branch
```

are roughly equivalent. The former is actually very commonly used.

是大致等价的。前面一个事实上用的更多。

Submitting patches to a project 向项目提交补丁

If you just have a few changes, the simplest way to submit them may just be to send them as patches in email:

如果你做了一定的变更，提交他们的最简单办法就是将它们通过电子邮件发出去：

First, use git-format-patch(1); for example:

首先，使用 git-format-patch(1); 例子：

```
$ git format-patch origin
```

will produce a numbered series of files in the current directory, one for each patch in the current branch but not in origin/HEAD.

这将会在当前目录下产生一系列文件，每个补丁都是针对当前分支，而不是 origin/HEAD。

You can then import these into your mail client and send them by hand. However, if you have a lot to send at once, you may prefer to use the git-send-email(1) script to automate the process. Consult the mailing list for your project first to determine how they prefer such patches be handled.

你可以将这些文件导入你的邮件客户端工具并手工发送出去。不过，如果你一次性要发送很多东西的话，你可以使用 git-send-email(1) 脚本来自动处理。首先参考一下你的项目的邮件列表，了解一下他们是如何处理补丁的。

Importing patches to a project 给项目导入补丁

Git also provides a tool called git-am(1) (am stands for "apply mailbox"), for importing such an emailed series of patches. Just save all of the patch-containing messages, in order, into a single mailbox file, say "patches.mbox", then run

Git 还提供了专门的工具 git-am(1)（am 是 "apply mailbox" 的缩写），它用于导入邮件中的补丁串。补丁邮件中同时也包含了补丁信息，邮件包的名字叫 "patches.mbox"，你可以运行

```
$ git am -3 patches.mbox
```

Git will apply each patch in order; if any conflicts are found, it will stop, and you can fix the conflicts as described in "Resolving a merge". (The "-3" option tells git to perform a merge; if you would prefer it just to abort and leave your tree and index untouched, you may omit that option.)

Git 按顺序将补丁应用到项目中；要是出现冲突，它将停下来，你可以修正冲突，就像我们在 "解决合并冲突" 中所阐述的那样。（选项 "-3" 告诉执行合并操作；如果你想它不触动你的工作树和索引，你可以忽略这个选项。）

Once the index is updated with the results of the conflict resolution, instead of creating a new commit, just run

一旦索引的被冲突的结果更新，那就需要创建一个新的交付了，只需运行：

```
$ git am --resolved
```

and git will create the commit for you and continue applying the remaining patches from the mailbox.

这样 git 将创建一个新的交付，并继续导入邮件包中的余下补丁。0

The final result will be a series of commits, one for each patch in the original mailbox, with authorship and commit log message each taken from the message containing each patch.

最终的结果是我们得到一系列的交付，每个在邮件包中的补丁都附带了原著的信息，交付信息。

Public git repositories 发布 git 版本库

Another way to submit changes to a project is to tell the maintainer of that project to pull the changes from your repository using git-pull(1). In the section "Getting updates with git-pull" we described this as a way to get updates from the "main" repository, but it works just as well in the other direction.

另外一个提交变更到项目的方法是告诉项目的维护者从你的版本库中用 git-pull(1) 命令将你的变更拉入。在 "用 git-pull 取得更新" 一节中，我们已经阐述过怎么用这个命令来从 "主" 版本库中更新你自己的版本库，现在只是这个命令的反向应用而已。

If you and the maintainer both have accounts on the same machine, then you can just pull changes from each other's repositories directly; commands that accept repository URLs as arguments will also accept a local directory name:

如果你和项目的维护者都在同一部机器上有用户帐号，那么你就可以从其他的版本库中直接拉入变更；git 命令同样可以接受本地目录名作为连接名称。

```
$ git clone /path/to/repository
```

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

or an ssh URL:

或者是 ssh 的连接名：

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

For projects with few developers, or for synchronizing a few private repositories, this may be all you need.

对于开发人员不多的项目，或者是对于涉及同步隐私的版本库来说，这些已经足够满足你的需要了。

However, the more common way to do this is to maintain a separate public repository (usually on a different host) for others to pull changes from. This is usually more convenient, and allows you to cleanly separate private work in progress from publicly visible work.

不过，更常见的方式是从几个分立的公共版本库中（通常是在不同的主机上）向用户提供拉入变更的策略。这通常会更加方便，并且容许你清晰地将隐私工作从公共版本库中剥离出来。

You will continue to do your day-to-day work in your personal repository, but periodically "push" changes from your personal repository into your public repository, allowing other developers to pull from that repository. So the flow of changes, in a situation where there is one other developer with a public repository, looks like this:

你将在你的个人版本库中一天接一天地开展你的工作，并阶段性地 "push" 将变更从你的个人版本库推入到你的公共版本库，让其他的开发者从可以从你的公共版本库中拉入你的工作。变更的流程，就像你面向另外一个开发者，他也有自己的公共版本库那样，看起来如下：

```
                    you push
your personal repo ------------------> your public repo
      ^                                     |
```

```
      |                                        |
      | you pull                               | they pull
      |                                        |
      |                                        |
      |                they push               V
their public repo <-------------------- their repo
```

We explain how to do this in the following sections.

我们将在下一节中介绍如何运作这个模式。

Setting up a public repository 建立一个公共版本库

Assume your personal repository is in the directory ~/proj. We first create a new clone of the repository and tell git-daemon that it is meant to be public:

假设你的个人版本库在目录 ~/proj。我们首先克隆一个新的版本库并告诉 git 守护进程（git-daemon）这个是一个公共版本库：

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

The resulting directory proj.git contains a "bare" git repository—it is just the contents of the ".git" directory, without any files checked out around it.

运行的结果是创建了一个 proj.git 的裸目录 -- 他仅仅包含了 ".git" 目录，而不提取出版本库中包含的文件。

Next, copy proj.git to the server where you plan to host the public repository. You can use scp, rsync, or whatever is most convenient.

接着，拷贝 proj.git 到你计划用来部署公共版本库的主机上去。你可以用 scp, rsync，反正就是你用得最方便的工具。

Exporting a git repository via the git protocol 通过 git 协议公开版本库

This is the preferred method.

这个是推荐使用的方式。

If someone else administers the server, they should tell you what directory to put the repository in, and what git:// URL it will appear at. You can then skip to the section "Pushing changes to a public repository", below.

如果服务器维护者中的某人，他告诉了你在服务器中版本库应该放在什么目录和 git:// 的连接路径。你可以跳到下面 "将变更推入公共版本库" 一节。

Otherwise, all you need to do is start git-daemon(1); it will listen on port 9418. By default, it will allow access to any directory that looks like a git directory and contains the magic file git-daemon-export-ok. Passing some directory paths as git-daemon arguments will further restrict the exports to those paths.

否则，你就需要启动 git-daemon(1)；它将监听 9418 端口。默认地，它容许访问所有个的包含魔术文件 git-daemon-export-ok 的 git 目录。向 git-deamon 传递目录参数可以有效限制输出目录。

You can also run git-daemon as an inetd service; see the git-daemon(1) man page for details. (See especially the examples section.)

你可以将 git-deamon 作为系统服务；详情参考 git-daemon(1) 手册页（尤其是 example 一节）。

Exporting a git repository via http 通过 http 协议公开版本库

The git protocol gives better performance and reliability, but on a host with a web server set up, http exports may be simpler to set up.

使用 git 协议会得到更好的性能和伸缩性，不过对于已经配置好网页服务器的主机来说，用 http 协议公开版本库更简单。

All you need to do is place the newly created bare git repository in a directory that is exported by the web server, and make some adjustments to give web clients some extra information they need:

你需要做的就是在 Web 服务器的目录中放置新的裸版本库，并根据 Web 服务的扩展信息为 Web 客户端作一些调整。

```
$ mv proj.git /home/you/public_html/proj.git
$ cd proj.git
$ git --bare update-server-info
$ mv hooks/post-update.sample hooks/post-update
```

(For an explanation of the last two lines, see git-update-server-info(1) and githooks(5).)

（关于上述命令的最后两行的解释，参考 git-update-server-info(1) 与 githooks(5)。）

Advertise the URL of proj.git. Anybody else should then be able to clone or pull from that URL, for example with a command line like:

公布 proj.git 的 URL，任何人都应该可以通过 URL 克隆和拉入版本库的东西了，命令大致如下：

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

(See also setup-git-server-over-http for a slightly more sophisticated setup using WebDAV which also allows pushing over http.)

（还可以参考 setup-git-server-over-http，通过设置 WebDAV 也可以通过 http 来容许推入操作。）

Pushing changes to a public repository 将变更推入到公共版本库

Note that the two techniques outlined above (exporting via http or git) allow other maintainers to fetch your latest changes, but they do not allow write access, which you will need to update the public repository with the latest changes created in your private repository.

注意，上述的两个技术框架（通过 http 或者 git）容许其他的维护者抓取你的最新更新，但是他们是无法进行写入操作的，因此你需要用你个人版本库中的最新变更去更新公共版本库。

The simplest way to do this is using git-push(1) and ssh; to update the remote branch named "master" with the latest state of your branch named "master", run

最简单的方式是使用 git-push(1) 命令和 ssh， 将你个人版本库中的 "master" 分支的最新状态更新到远程版本库上 "master" 的分支，运行

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

or just

或者只需要

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

As with git-fetch, git-push will complain if this does not result in a fast forward; see the following section for details on handling this case.

正如 git-fetch 命令一样，git-push 命令的运行结果如果不是 快进 操作的话，将会遇到一些状况；下面的章节将详细介绍处理这些状况的方法。

Note that the target of a "push" is normally a bare repository. You can also push to a repository that has a checked-out working tree, but the working tree will not be updated by the push. This may lead to unexpected results if the branch you push to is the currently checked-out branch!

注意，“推入”的目标是一个普通的裸版本库。你当然可以将东西推入一个已经提取了工作树的版本库中，不过那个版本库的工作树是不会被推入操作刷新的。**这可**

能做成不可预测的后果！如果你推入的分支，正好就是那个版本库当前的提取分支的话！

As with git-fetch, you may also set up configuration options to save typing; so, for example, after

与 git-fetch 命令一样，你可以通过配置项来节省打字量；作为例子，在下面命令之后

```
$ cat >>.git/config <<EOF
[remote "public-repo"]
        url = ssh://yourserver.com/~you/proj.git
EOF
```

you should be able to perform the above push with just

你应该可以像下面的命令那样执行上面的命令了

```
$ git push public-repo master
```

See the explanations of the remote.<name>.url, branch.<name>.remote, and remote.<name>.push options in git-config(1) for details.

详细说明情参考 git-config(1) 中有关 remote.<name>.url, branch.<name>.remote, 与 remote.<name>.push。

What to do when a push fails 推入失败之后该怎么处理

If a push would not result in a fast forward of the remote branch, then it will fail with an error like:

如果推入操作对于远程分支来说不是快进的话，那么它将失败，大致有如下的输出：

```
error: remote 'refs/heads/master' is not an ancestor of
 local  'refs/heads/master'.
 Maybe you are not up-to-date and need to pull first?
```

```
error: failed to push to 'ssh://yourserver.com/~you/proj.git'
```

This can happen, for example, if you:

- use git-reset —hard to remove already-published commits, or
- use git-commit —amend to replace already-published commits (as in the section called “Fixing a mistake by rewriting history”), or
- use git-rebase to rebase any already-published commits (as in the section called “Keeping a patch series up to date using git-rebase”).

导致这种情况的可能原因是你：

- 使用 git-reset --hard 删除了某些已经传播出去的交付，或者
- 使用 git-commit --amend 覆盖了某些已经传播出去的交付（在 “通过重写历史修复失误” 一节中提到过），或者
- 使用 git-rebase 对已经传播出去的交付进行了重新定位（在 “使用 git-rebase 保持补丁串的新颖” 中说明）。

You may force git-push to perform the update anyway by preceding the branch name with a plus sign:

你可以用 git-push 命令在分支名前面加一个加号进行强制推入：

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

Normally whenever a branch head in a public repository is modified, it is modified to point to a descendant of the commit that it pointed to before. By forcing a push in this situation, you break that convention. (See the section called “Problems with rewriting history”.)

正常情况下，推入操作就是修改公共版本库的分支头，将它更新到它原来指向的那个交付的后裔交付。在这种状况下强行推入等如打破了这个规则。（参考 “重写历史带来的问题” 一节。）

Nevertheless, this is a common practice for people that need a simple way to publish a work-in-progress patch series, and it is an acceptable compromise as long as you warn other developers that this is how you intend to manage the branch.

对于那些需要一个简单的方式来传播工作进程的用户来说，这是一个现实的办法，也是一个可以接受的折衷方案，不过你要尽可能地提示其他开发者，你是打算这样管理你的分支的。

It's also possible for a push to fail in this way when other people have the right to push to the same repository. In that case, the correct solution is to retry the push after first updating your work: either by a pull, or by a fetch followed by a rebase; see the next section and gitcvs-migration(7) for more.

推入失败的可能性对于那些有权限写入同一个版本库的人来说同样存在。在这中情况下，正确的解决方案是更新你的工作之后再尝试推入：先做拉入（pull），或者是抓取（fetch）之后再进行重定位（rebase）；详情参考下一节和 gitcvs-migration(1)。

Setting up a shared repository 建立共享版本库

Another way to collaborate is by using a model similar to that commonly used in CVS, where several developers with special rights all push to and pull from a single shared repository. See gitcvs-migration(7) for instructions on how to set this up.

另外一个协同开发的策略就是使用类似 CVS 那样的模式，就是各地的开发者都对单个公共的版本库有特定的写入权限。参考 gitcvs-migration(7) 了解如何建立这种方式。

However, while there is nothing wrong with git's support for shared repositories, this mode of operation is not generally recommended, simply because the mode of collaboration that git supports—by exchanging patches and pulling from public repositories—has so many advantages over the central shared repository:

无论如何，即使 git 对支持共享版本库这种开发模式其实是没有问题的，但是不建议采用这种模式，最简单的理由就是 git 所支持的由公共版本库进行补丁交换和拉入的模式，要比中央共享版本库的模式要先进得多。

- Git's ability to quickly import and merge patches allows a single maintainer to process incoming changes even at very high rates. And when that becomes too much, git-pull provides an easy way for that

maintainer to delegate this job to other maintainers while still allowing optional review of incoming changes.

- Since every developer's repository has the same complete copy of the project history, no repository is special, and it is trivial for another developer to take over maintenance of a project, either by mutual agreement, or because a maintainer becomes unresponsive or difficult to work with.
- The lack of a central group of "committers" means there is less need for formal decisions about who is "in" and who is "out".
- Git 可以快速导入和合并补丁，容许单个的维护者处理高频率到达的变更。当情况实在是过分的时候，git-pull 容许该维护者将维护工作委托给其他的开发者的同时预览这些到来的变更。
- 每个开发者的版本库始终拥有项目的完整的历史记录，没有任何一个版本库是特殊的，这为其他的开发者成为项目维护者提供了便利，不管大家一致同意的，还是由于原来的项目维护者已经无法承担维护的责任或者是对承担这个工作存在困难。
- 中央版本库模式的局限性在于，至少你要决定“提交者”谁应该参与谁应该出局。

Allowing web browsing of a repository 容许 Web 浏览版本库

The gitweb cgi script provides users an easy way to browse your project's files and history without having to install git; see the file gitweb/INSTALL in the git source tree for instructions on setting it up.

gitweb cgi 脚本为你浏览项目的文件和历史提供了一个便利的方法，而无须安装 git；参考 git 代码树中的 gitweb/INSTALL 学习如何安装。

Examples 例子

Maintaining topic branches for a Linux subsystem maintainer | Linux 子系统维护者如何维护主题分支

This describes how Tony Luck uses git in his role as maintainer of the IA64 architecture for the Linux kernel.

这里介绍 Tony Luck 作为 Linux 内核 IA64 架构的维护者如何工作。

He uses two public branches:

- A "test" tree into which patches are initially placed so that they can get some exposure when integrated with other ongoing development. This tree is available to Andrew for pulling into -mm whenever he wants.
- A "release" tree into which tested patches are moved for final sanity checking, and as a vehicle to send them upstream to Linus (by sending him a "please pull" request.)

他使用两个公共分支：

- “测试”树里面放置的是那些整合过的正在进行的开发工作的补丁，这棵树可以让 Andrew 随时执行拉入操作。
- “发布”树里面放置的是那些已经测试过，要移交做最后的健全性检验的补丁，它作为向上游的 Linus 发送这些补丁的载体（向他发出一个“拉入”的请求就行了。）

He also uses a set of temporary branches ("topic branches"), each containing a logical grouping of patches.

他还有一组临时性的分支（“主题分支”），每个中都包含了一个逻辑分组补丁集。

To set this up, first create your work tree by cloning Linus's public tree:

首先通过克隆 Linus 的公共树来创建你自己的工作树。

```
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
work
$ cd work
```

Linus's tree will be stored in the remote branch named origin/master, and can be updated using git-fetch(1); you can track other public trees using git-remote(1) to set up a "remote" and git-fetch(1) to keep them up-to-date; see Chapter 1, Repositories and Branches.

Linus 的树将被保存为一个远程分支名叫 origin/master，并且可以用 git-fetch(1) 命令进行更新；你可以用 git-remote(1) 命令来跟踪另外的远程分支，并用 git-fetch(1) 保持他们的更新；参考 第一章，版本库与分支。

Now create the branches in which you are going to work; these start out at the current tip of origin/master branch, and should be set up (using the —track option to git-branch(1)) to merge changes in from Linus by default.

现在创建你用来开展工作的分支；它将以 origin/master 分支的顶端作为起始点，并且设置它默认用来合并来自 Linus 的变更（在 git-branch(1) 命令中使用 --track 选项）。

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

These can be easily kept up to date using git-pull(1).

通过 git-pull(1) 很容易就保持他们的更新。

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

Important note! If you have any local changes in these branches, then this merge will create a commit object in the history (with no local changes git will simply do a "Fast forward" merge). Many people dislike the "noise" that this creates in the Linux history, so you should avoid doing this capriciously in the "release" branch, as these noisy commits will become part of the permanent history when you ask Linus to pull from the release branch.

重要提示！如果你有任何的变更在本地的分支中，那么他们的合并都将是在历史中创建新的交付（对于没有本地变更的合并，git 则是简单地进行快进合并）。许多

人并不喜欢在 Linux 的发展历史中“噪音”，而这些噪音交付当你要求 Linus 从发行分支中拉入的话，将永远成为 Linux 历史的一部分。

A few configuration variables (see git-config(1)) can make it easy to push both branches to your public tree. (See the section called “Setting up a public repository”.)

通过配置变量（参考 git-config(1)）可以很容易地将这个两个分支推出到你的公共树中。（参考 “建立公共版本库” 一节。）

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

Then you can push both the test and release trees using git-push(1):

接着你可以用 git-push(1) 退出这两个分支了：

```
$ git push mytree
```

or push just one of the test and release branches using:

或者只是推出这两个分支中的其中一个:

```
$ git push mytree test
```

or

或

```
$ git push mytree release
```

Now to apply some patches from the community. Think of a short snappy name for a branch to hold this patch (or related group of patches), and create a new branch from the current tip of Linus's branch:

现在从社群中导入一些补丁。给这个装载这些补丁（或者是相关的补丁组）的分支起一个短而酷的名字吧，并且从 Linus 的分支顶端作为新分支的起点。

```
$ git checkout -b speed-up-spinlocks origin
```

Now you apply the patch(es), run some tests, and commit the change(s). If the patch is a multi-part series, then you should apply each as a separate commit to this branch.

现在你导入这些补丁，测试，并提交这些变更。如果这些补丁是分成多个补丁串的话，你应该将它们作为分立的交付导入这个分支中。

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

When you are happy with the state of this change, you can pull it into the "test" branch in preparation to make it public:

当你觉得这些变更的状态还不错，你可以将他们拉入到 "test" 分支中使他们发布出去了。

```
$ git checkout test && git pull . speed-up-spinlocks
```

It is unlikely that you would have any conflicts here … but you might if you spent a while on this step and had also pulled new versions from upstream.

在这里多半不会产生冲突...不过你要是在这一刻已经拉入过上游版本的话，还是可能会出现冲突的。

Some time later when enough time has passed and testing done, you can pull the same branch into the "release" tree ready to go upstream. This is where you see the value of keeping each patch (or patch series) in its own branch. It means that the patches can be moved into the "release" tree in any order.

当进行过充分的测试之后，你可以将这个分支的东西拉入到 "release" 分支中，并准备好向上游版本传递了。到现在你应该体会到将每个补丁（或者是补丁串）保留在他们自己专有的分支中的好处了。它意味着补丁串可以以不同的顺序加入到 "release" 树中。

```
$ git checkout release && git pull . speed-up-spinlocks
```

After a while, you will have a number of branches, and despite the well chosen names you picked for each of them, you may forget what they are for, or what status they are in. To get a reminder of what changes are in a specific branch, use:

经过一段时间之后，你将有一大堆的分支，即使你在建立它们的时候已经给他们取了个不错的名字，你也可能会忘记了他们是干什么用的了，或者是不清楚它们现在处在什么状态了。要找回对这些独特的分支的记忆，请用：

```
$ git log linux..branchname | git shortlog
```

To see whether it has already been merged into the test or release branches, use:

要查看他们是否已经合并到 test 或者是 release 分支的情况，使用：

```
$ git log test..branchname
```

or

或者

```
$ git log release..branchname
```

(If this branch has not yet been merged, you will see some log entries. If it has been merged, then there will be no output.)

（如果某个分支还没有被合并，你将看到几个日志的条目。如果它已经被合并过，就不会有任何的输出。）

Once a patch completes the great cycle (moving from test to release, then pulled by Linus, and finally coming back into your local "origin/master" branch), the branch for this change is no longer needed. You detect this when the output from:

一旦补丁完成了创建的周期（从 test 转移到了 release，并且被 Linus 拉入，最终成为你的本地分支 "origin/master" 的一部分），这个分支就没有再存在的必要了。你可以通过以下的命令的输出来判断：

```
$ git log origin..branchname
```

is empty. At this point the branch can be deleted:

输出为空，此时那个分支就可以删除了：

```
$ git branch -d branchname
```

Some changes are so trivial that it is not necessary to create a separate branch and then merge into each of the test and release branches. For these changes, just apply directly to the "release" branch, and then merge that into the "test" branch.

某些变更是没有必要为他们创建独立的分支的，将他们直接合并到 test 和 release 中去就可以。对于这些变更，可以直接将他们导入 release 分支，并合并到 test 分支中。

To create diffstat and shortlog summaries of changes to include in a "please pull" request to Linus you can use:

请求 Linus 进行拉入的时候，最好创建一个变更的短摘要：

```
$ git diff --stat origin..release
```

and

并且

```
$ git log -p origin..release | git shortlog
```

Here are some of the scripts that simplify all this even further.

这里有些脚本可以使今后的工作更加简便。

```
==== update script ====
# Update a branch in my GIT tree.  If the branch to be updated
# is origin, then pull from kernel.org.  Otherwise merge
# origin/master branch into test|release branch

case "$1" in
test|release)
        git checkout $1 && git pull . origin
        ;;
origin)
        before=$(git rev-parse refs/remotes/origin/master)
        git fetch origin
        after=$(git rev-parse refs/remotes/origin/master)
        if [ $before != $after ]
        then
                git log $before..$after | git shortlog
        fi
        ;;
*)
        echo "Usage: $0 origin|test|release" 1>&2
        exit 1
        ;;
esac
==== merge script ====
# Merge a branch into either the test or release branch

pname=$0

usage()
{
        echo "Usage: $pname branch test|release" 1>&2
        exit 1
```

```
}

git show-ref -q --verify -- refs/heads/"$1" || {
        echo "Can't see branch <$1>" 1>&2
        usage
}

case "$2" in
test|release)
        if [ $(git log $2..$1 | wc -c) -eq 0 ]
        then
                echo $1 already merged into $2 1>&2
                exit 1
        fi
        git checkout $2 && git pull . $1
        ;;
*)
        usage
        ;;
esac
==== status script ====
# report on status of my ia64 GIT tree

gb=$(tput setab 2)
rb=$(tput setab 1)
restore=$(tput setab 9)

if [ `git rev-list test..release | wc -c` -gt 0 ]
then
        echo $rb Warning: commits in release that are not in test
$restore
        git log test..release
fi

for branch in `git show-ref --heads | sed 's|^.*/||'`
```

```
do
        if [ $branch = test -o $branch = release ]
        then
                continue
        fi

        echo -n $gb ======= $branch ====== $restore " "
        status=
        for ref in test release origin/master
        do
                if [ `git rev-list $ref..$branch | wc -c` -gt 0 ]
                then
                        status=$status${ref:0:1}
                fi
        done
        case $status in
        trl)
                echo $rb Need to pull into test $restore
                ;;
        rl)
                echo "In test"
                ;;
        l)
                echo "Waiting for linus"
                ;;
        "")
                echo $rb All done $restore
                ;;
        *)
                echo $rb "<$status>" $restore
                ;;
        esac
        git log origin/master..$branch | git shortlog
done
```

Chapter 5. Rewriting history and maintaining patch series 第五章. 改写历史与维护补丁串

Normally commits are only added to a project, never taken away or replaced. Git is designed with this assumption, and violating it will cause git's merge machinery (for example) to do the wrong thing.

通常来说，交付一旦被加入到项目中，就永远不能被抛弃和覆盖。Git 就是以这样的假设来设计的，打破这个原则（只是假设）会导致 git 的合并机制发生错误。

However, there is a situation in which it can be useful to violate this assumption.

可是在某些情形下，打破这个假设可能是有用的。

Creating the perfect patch series 创建出色的补丁串

Suppose you are a contributor to a large project, and you want to add a complicated feature, and to present it to the other developers in a way that makes it easy for them to read your changes, verify that they are correct, and understand why you made each change.

假设你是一个大项目的贡献者，并希望给这个项目加入一个复杂的功能，将它献给其他的开发者。你应该提供一个便利来让他们查看你的变更，检验这些变更的正确性，并让其他人了解你的每个变更。

If you present all of your changes as a single patch (or commit), they may find that it is too much to digest all at once.

如果你一次过将你的变更放在一个补丁（或者是交付中），他们会觉得实在是太大了。

If you present them with the entire history of your work, complete with mistakes, corrections, and dead ends, they may be overwhelmed.

如果你献出你工作的整个历史过程，包括你的错误，修复，和终极版本，他们也许就被这些历史记录淹没了。

So the ideal is usually to produce a series of patches such that:

1. Each patch can be applied in order.
2. Each patch includes a single logical change, together with a message explaining the change.
3. No patch introduces a regression: after applying any initial part of the series, the resulting project still compiles and works, and has no bugs that it didn't have before.
4. The complete series produces the same end result as your own (probably much messier!) development process did.

所以，制作补丁串的要领如下：

1. 每个补丁应用起来都是规整的。
2. 每个补丁应该是一个独立的逻辑过程，并带有说明这个变迁的信息。
3. 补丁都不应该造成后退：项目应用了补丁串的任何初始部分之后，项目仍然是可以完整地工作的，并且不包含你原来工作中的漏洞。
4. 整个补丁串应用完之后，项目所得到的结果应该和你本来的工作（可能是非常散乱的）一样。

We will introduce some tools that can help you do this, explain how to use them, and then explain some of the problems that can arise because you are rewriting history.

我们提供了一些工具来帮助你实现这些目标，也说明了如何使用它们，以及他们所带来的问题，因为你可能需要改写历史了。

Keeping a patch series up to date using git-rebase 使用 git-rebase 保持补丁串的新颖

Suppose that you create a branch "mywork" on a remote-tracking branch "origin", and create some commits on top of it:

假定你创建了一个针对 "origin" 的远程跟踪分支叫 "mywork"，并且在这个分支上做了一定的工作。

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

You have performed no merges into mywork, so it is just a simple linear sequence of patches on top of "origin":

你还没有在 mywork 中做过任何的合并，所以目前它只是一个在 origin 的顶端上面一个很简单的队列。

```
o--o--o <-- origin
       \
        o--o--o <-- mywork
```

Some more interesting work has been done in the upstream project, and "origin" has advanced:

某些有趣的工作已经在项目的上游版本库中完成了，实际上 "origin" 已经向前迈进了：

```
o--o--O--o--o--o <-- origin
       \
        a--b--c <-- mywork
```

At this point, you could use "pull" to merge your changes back in; the result would create a new merge commit, like this:

此时，你可以用 "拉入" 来合并这些后台的变更，这会导致创建一个合并的交付，类似：

```
o--o--O--o--o--o <-- origin
       \        \
        a--b--c--m <-- mywork
```

However, if you prefer to keep the history in mywork a simple series of commits without any merges, you may instead choose to use git-rebase(1):

不过，如果你希望保持 mywork 中的简单交付队列而不进行合并的话，git-rebase(1) 就是替代方案：

```
$ git checkout mywork
$ git rebase origin
```

This will remove each of your commits from mywork, temporarily saving them as patches (in a directory named ".git/rebase-apply"), update mywork to point at the latest version of origin, then apply each of the saved patches to the new mywork. The result will look like:

这将从 mywork 中删除你的每个工作，临时保存为补丁集（在目录 ".git/rebase-apply"中），将 mywork 指向 origin 的最新版本，再将刚才保存过的补丁应用到 mywork 中。结果大致像下面这个样子：

```
o--o--O--o--o--o <-- origin
               \
                a'--b'--c' <-- mywork
```

In the process, it may discover conflicts. In that case it will stop and allow you to fix the conflicts; after fixing conflicts, use "git-add" to update the index with those contents, and then, instead of running git-commit, just run

在此过程中，它也许会发现冲突。并停下来让你解决冲突；之后用 "git-add" 刷新索引，但是不要运行 git-commit，而应该运行

```
$ git rebase --continue
```

and git will continue applying the rest of the patches.

这样 git 会继续应用补丁集中余下的补丁。

At any point you may use the —abort option to abort this process and return mywork to the state it had before you started the rebase:

你任何时候都可以用 --abort 选项终止重新定位的过程，而让 mywork 回到你开始执行 rebase 之前的状态：

```
$ git rebase --abort
```

Rewriting a single commit 重写单个交付

We saw in the section called “Fixing a mistake by rewriting history” that you can replace the most recent commit using

我们已经在 “通过重写历史修复失误” 一节中学习过你是怎么样通过下面的命令来覆盖当前的交付的

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

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first.

它会用一个新的交付去覆盖旧的交付的方式来整合你的变更，首先它会给出一个容许你修改旧的交付信息的机会。

You can also use a combination of this and git-rebase(1) to replace a commit further back in your history and recreate the intervening changes on top of it. First, tag the problematic commit with

你同样可以将这个方法结合 git-rebase(1) 来覆盖在你的历史进程中更久远的交付，并在它的上面再生成一个调解性的变更。首先我们将有问题的交付打个标签

```
$ git tag bad mywork~5
```

(Either gitk or git-log may be useful for finding the commit.)

（输入 gitk 或者是 git-log 对查找这个交付很有帮助）

Then check out that commit, edit it, and rebase the rest of the series on top of it (note that we could check out the commit on a temporary branch, but instead we're using a detached head):

提取这个交付，

```
$ git checkout bad
$ # make changes here and update the index
$ git commit --amend
$ git rebase --onto HEAD bad mywork
```

When you're done, you'll be left with mywork checked out, with the top patches on mywork reapplied on top of your modified commit. You can then clean up with

当你完成了之后，你可以离开，并将 mywork 提取出来，此时你所处的位置已经是在那个修改过的交付上再打了所有补丁的历史进程的顶端。那样你就可以进行清理工作了

```
$ git tag -d bad
```

Note that the immutable nature of git history means that you haven't really "modified" existing commits; instead, you have replaced the old commits with new commits having new object names.

注意 git 的历史不可变更的属性意味着你不可以修改现存的交付；你要用一个具有新的对象名称的交付去覆盖旧的交付。

Reordering or selecting from a patch series 在补丁串中选取与重新排序

Given one existing commit, the git-cherry-pick(1) command allows you to apply the change introduced by that commit and create a new commit that records it. So, for example, if "mywork" points to a series of patches on top of "origin", you might do something like:

给定一个现存的交付，git-cherry-pick(1) 命令可以让你创建一个新交付对那个现存交付所包含的变更进行重新应用和排序。举例，如果 "mywork" 是指向以 "origin" 为起点的补丁串的指针，你可以做类似以下的工作：

```
$ git checkout -b mywork-new origin
$ gitk origin..mywork &
```

and browse through the list of patches in the mywork branch using gitk, applying them (possibly in a different order) to mywork-new using cherry-pick, and possibly modifying them as you go using commit —amend. The git-gui(1) command may also help as it allows you to individually select diff hunks for inclusion in the index (by right-clicking on the diff hunk and choosing "Stage Hunk for Commit").

你可以用 gitk 来浏览在 mywork 中的补丁串列表，并用 cherry-pick 将他们重新应用到新的分支 mywork-new 上去（可能使用不同的顺序），并且可以用 commit --amend 对它们进行重新修订。git-gui(1) 命令可以帮助你独立地选取不同的差异块，包括在索引中的（鼠标右击该差异块并点 "Stage Hunk for Commit"）。

Another technique is to use git-format-patch to create a series of patches, then reset the state to before the patches:

另外一个技巧就是用 git-formate-patch 创建一个补丁串，之后回到这些补丁形成之前的状态：

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

Then modify, reorder, or eliminate patches as preferred before applying them again with git-am(1).

接着编辑，重排顺序，或者剔除补丁，如前面介绍过那样用 git-am 将他们重新应用一次。

Other tools 第三方工具

There are numerous other tools, such as StGIT, which exist for the purpose of maintaining a patch series. These are outside of the scope of this manual.

有许多的第三方工具，譬如 StGIT，专门用于补丁串的维护的。不过这已经超出了本文的覆盖范围。

Problems with rewriting history 重写历史带来的问题

The primary problem with rewriting the history of a branch has to do with merging. Suppose somebody fetches your branch and merges it into their branch, with a result something like this:

重写历史带来的主要问题是对合并操作的影响。假设某人抓取了你的分支并合并到他们的分支上，结果大致如下：

```
o--o--O--o--o--o <-- origin
       \        \
        t--t--t--m <-- their branch:
```

Then suppose you modify the last three commits:

那么设想一下你修改了最后的三个交付：

```
         o--o--o <-- new head of origin
        /
o--o--O--o--o--o <-- old head of origin
```

If we examined all this history together in one repository, it will look like:

加入我们检验一下在同一版本库中的历史，他看起来应该是这样：

```
         o--o--o <-- new head of origin
        /
o--o--O--o--o--o <-- old head of origin
       \         \
        t--t--t--m <-- their branch:
```

Git has no way of knowing that the new head is an updated version of the old head; it treats this situation exactly the same as it would if two developers had independently done the work on the old and new heads in parallel. At this point, if someone attempts to merge the new head in to their branch, git will attempt to merge together the two (old and new) lines of development, instead of trying to replace the old by the new. The results are likely to be unexpected.

Git 无从得知新的分支头是老的分支头的新版本；他会将它当成是两个独立的开发者分支在新分支和老分支上并行工作的结果。要是某人试图合并新分支到他们的分支中， git 将试图将这两个开发线路都合并进去（新的和老的），而不是用新分支去覆盖老的分支。这样一来就会出现可不预测的结果了。

You may still choose to publish branches whose history is rewritten, and it may be useful for others to be able to fetch those branches in order to examine or test them, but they should not attempt to pull such branches into their own work.

你仍然可以选择将重写过的历史作为你发布的内容，这对于其他人要抓取它进行检验和测试还是有用的，不过他们就不应该将这些东西拉入到他们的分支中了。

For true distributed development that supports proper merging, published branches should never be rewritten.

为了在开发工作中确保合并的正确性，用于公共传播的分支觉不应该进行历史的重写。

Why bisecting merge commits can be harder than bisecting linear history 为何定位合并交付中的问题要比在线性历史中困难

The git-bisect(1) command correctly handles history that includes merge commits. However, when the commit that it finds is a merge commit, the user may need to work harder than usual to figure out why that commit introduced a problem.

git-bisect(1) 命令可以正确地处理包含合并交付的历史进程。不过，当一个合并交付在切分过程中出现的时候，用户要推理出合并交付为何带来了问题要比正常的情况下困难。

Imagine this history:

想象这样一个历史场面：

```
      ---Z---o---X---...---o---A---C---D
          \                       /
           o---o---Y---...---o---B
```

Suppose that on the upper line of development, the meaning of one of the functions that exists at Z is changed at commit X. The commits from Z leading to A change both the function's implementation and all calling sites that exist at Z, as well as new calling sites they add, to be consistent. There is no bug at A.

假设上面的那条开发线路，某个函数在 Z 位置出现并在 X 处被修改过。交付从 Z 发展到 A 的变更中其实是函数的不同的实现，并且函数的调用已经在 Z 位置就开始出现了，以及新的函数调用还会被加入，并且调用会持续地被加入。在 A 处函数已经没有漏洞了。

Suppose that in the meantime on the lower line of development somebody adds a new calling site for that function at commit Y. The commits from Z leading to B all assume the old semantics of that function and the callers and the callee are consistent with each other. There is no bug at B, either.

再来设想下面的那条开发路线，某人在 Y 处加入了对那个函数新的调用。从 Z 到 B 的所有交付中，无论是函数的调用者还是被调用者，都会假设函数是旧的语义形式，并且它们每个都会持续使用这样的形式。项目发展到 B 处就没有漏洞了。

Suppose further that the two development lines merge cleanly at C, so no conflict resolution is required.

假设两条开发线路发展到 C 处进行了干净的合并，也就是说没有冲突需要处理。

Nevertheless, the code at C is broken, because the callers added on the lower line of development have not been converted to the new semantics introduced on the upper line of development. So if all you know is that D is bad, that Z is good, and that git-bisect(1) identifies C as the culprit, how will you figure out that the problem is due to this change in semantics?

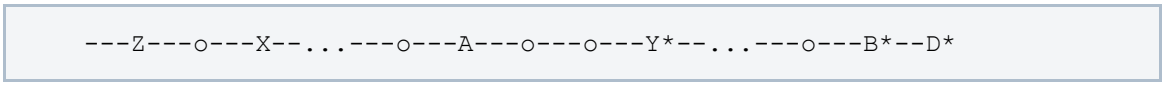
不过代码在 C 出就崩溃了，原因是下面的开发线路中的函数调用并没有针对在上面的研发线路中的函数的新的语义作出相应的调整。那么假设你在 D 处知道程序出事了，而且 Z 处是稳定的。 git-bisect(1) 将 C 位置判定为肇事者，那么你依据什么来推断问题其实是出在函数语义变迁上呢？

When the result of a git-bisect is a non-merge commit, you should normally be able to discover the problem by examining just that commit. Developers can make this easy by breaking their changes into small self-contained commits. That won't help in the case above, however, because the problem isn't obvious from examination of any single commit; instead, a global view of the development is required. To make matters worse, the change in semantics in the problematic function may be just one small part of the changes in the upper line of development.

当 git-bisect 是在做一个无合并的交付切分的时候，通常你应该可以通过检验那个交付来发现问题。开发者可以通过将他们的变更细分成小的自包含交付的方式来轻松地位问题。但是这样的方法在上面的情况下就无能为力了，因为问题在单个交付中表现得并不明显；故而，一个全局的研发视野对工程师来说是非常必要的。还有更糟糕的事情，要是在上面的开发线路中问题函数的语义变更只是变更集中很小的一部分的话。

On the other hand, if instead of merging at C you had rebased the history between Z to B on top of A, you would have gotten this linear history:

另一个方面，要是你不在 C 处进行合并操作，而是将你从 Z 到 B 的发展历史重新定位到 A 点这个历史顶端的话，你很容易就可以在一个线性历史中发现问题。



Bisecting between Z and D* would hit a single culprit commit Y*, and understanding why Y* was broken would probably be easier.

切分 Z 到 D* 的历史，马上就可以捕捉到 Y* 是肇事者，并且很容易就理解了 Y* 为什么会崩溃。

Partly for this reason, many experienced git users, even when working on an otherwise merge-heavy project, keep the history linear by rebasing against the latest upstream version before publishing.

基于这样的理由，许多经验丰富的 git 用户，即使他们工作在一个需要大规模合并的项目中时，在他们向上游版本发布工作的之前，总是用重定位来保持历史的线性发展。

Chapter 6. Advanced branch management 第六章. 高级分支管理

Fetching individual branches

Instead of using git-remote(1), you can also choose just to update one branch at a time, and to store it locally under an arbitrary name:

作为 git-remote 命令的替代品，你听样可以选择一次过更新一个分支，并将它保存为一个任意名称的本地分支：

```
$ git fetch origin todo:my-todo-work
```

The first argument, "origin", just tells git to fetch from the repository you originally cloned from. The second argument tells git to fetch the branch named "todo" from the remote repository, and to store it locally under the name refs/heads/my-todo-work.

第一个参数 "origin"，只是告诉 git 从你的版本库的克隆源头上抓取东西。第二个参数告诉 git 抓取远程分版本库中的 "todo" 分支，并保存为 refs/heads/my-todo-work 本地分支。

You can also fetch branches from other repositories; so

你同样可以抓取其他的远程分支；如：

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

will create a new branch named "example-master" and store in it the branch named "master" from the repository at the given URL. If you already have a branch named example-master, it will attempt to fast-forward to the commit given by example.com's master branch. In more detail:

这将创建一个新的分支名叫 "example-master" 并将指定的 URL 远程版本库中的 "master" 分支保存在这个分支中。如果你已经有一个 example-master 的分支，它将尝试快针对 example.com 上的 master 分支进行快进合并。下面会谈论更多的细节：

git fetch and fast-forwards 抓取与快进

In the previous example, when updating an existing branch, "git-fetch" checks to make sure that the most recent commit on the remote branch is a descendant of the most recent commit on your copy of the branch before updating your copy of the branch to point at the new commit. Git calls this process a fast forward.

在前面的范例中，当要更新的分支已经存在时， "git-fetch" 将检验并确认远程分支上最新的交付是你打算更新的这个分支的最新交付的后裔，并将这个本地分支的头指向拷贝下来的最新交付。Git 叫这个处理过程叫快进。

```
A fast forward looks something like this:

 o--o--o--o <-- old head of the branch
           \
            o--o--o <-- new head of the branch
```

In some cases it is possible that the new head will not actually be a descendant of the old head. For example, the developer may have realized she made a serious mistake, and decided to backtrack, resulting in a situation like:

在某些情况下，远程分支的最新的头未必就是本地分支头的后裔，譬如，开发者意识到她范了某些错误，并将历史倒退了，结果会类似下面的情形：

```
o--o--o--o--a--b <-- old head of the branch
          \
           o--o--o <-- new head of the branch
```

In this case, "git-fetch" will fail, and print out a warning.

这样的话，"git-fetch" 将失败，并打印警告信息。

In that case, you can still force git to update to the new head, as described in the following section. However, note that in the situation above this may mean losing the commits labeled "a" and "b", unless you've already created a reference of your own pointing to them.

在这种情况下，你仍然可以强制 git 更新到新的头，我们将在接着的章节说明。不过需要注意在上面的情况中，你将丢失交付 "a" 和 "b"，除非你已经创建了自己的引用指向他们。

Configuring remote branches 配置远程分支

We saw above that "origin" is just a shortcut to refer to the repository that you originally cloned from. This information is stored in git configuration variables, which you can see using git-config(1):

我们做前面已经知道 "origin" 是指向你的版本库的克隆源头的引用。这些配置信息保存做 git 的配置变量中，我们可以用 git-config(1) 来查看。

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

```
branch.master.merge=refs/heads/master
```

If there are other repositories that you also use frequently, you can create similar configuration options to save typing; for example, after

如果你有其他的远程版本库是需要频繁访问的，你可以配置一个简单的配置项来节省输入。譬如，配置了下面变量之后

```
$ git config remote.example.url git://example.com/proj.git
```

then the following two commands will do the same thing:

那么下面的两个命令是等价的。

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

Even better, if you add one more option:

更好的话，如果你加入更多的配置项：

```
$ git config remote.example.fetch master:refs/remotes/example/master
```

then the following commands will all do the same thing:

那么下面三个命令所做的事情是一样的：

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

You can also add a "+" to force the update each time:

你还可以加入一个 "+" 号，让每次都进行强制更新：

```
$ git config remote.example.fetch +master:ref/remotes/example/master
```

Don't do this unless you're sure you won't mind "git fetch" possibly throwing away commits on example/master.

不过，除非你确定你自己并不在乎 "git fetch" 可能会在 example/master 中抛弃任何交付，否则不要这样做。

Also note that all of the above configuration can be performed by directly editing the file .git/config instead of using git-config(1).

所有上面的配置项你都可以直接编辑 .git/config 文件来实现，而不用 git-config(1)命令。

See git-config(1) for more details on the configuration options mentioned above.

详情请参考 git-config(1) 中的有关上述配置项的介绍。

Chapter 7. Git concepts 第七章. Git 概念

Git is built on a small number of simple but powerful ideas. While it is possible to get things done without understanding them, you will find git much more intuitive if you do.

Git 是以精简强大这样的指导思想来开发的。即使你还没有完全理解它，但是你仍然可以发现 git 是很直观的。

We start with the most important, the object database and the index.

让我们先从最重要的东西开始吧，对象数据库和索引。

The Object Database 对象数据库

We already saw in the section called “Understanding History: Commits” that all commits are stored under a 40-digit "object name". In fact, all the information needed to represent

the history of a project is stored in objects with such names. In each case the name is calculated by taking the SHA1 hash of the contents of the object. The SHA1 hash is a cryptographic hash function. What that means to us is that it is impossible to find two different objects with the same name. This has a number of advantages; among others:

我们已经做“理解历史：交付”一节中看到，所有的交付都以一个 40 个十六进制字符组成的名字作为对象名来保存。事实上，所有需要展现项目历史的对象都是以这样的名字来保存的。这个名字是根据对象中的内容用 SHA1 哈希算法计算出来的。SHA1 是一个加密哈希函数。这意味着我们不可能用同一名字找到两个不同的对象，这有很多的好处；其中：

- Git can quickly determine whether two objects are identical or not, just by comparing names.
- Since object names are computed the same way in every repository, the same content stored in two repositories will always be stored under the same name.
- Git can detect errors when it reads an object, by checking that the object's name is still the SHA1 hash of its contents.

- Git 可以仅仅通过比较他们的名字就能快速地测定两个对象是否相同。
- 只要计算对象名的算法在每个版本库中都是一样的，那么在不同的版本库中，相同的内容他们的名字是一样的。
- Git 在读取一个对象的时候，可以通过检查对象名是否符合对象内容的 SHA1 哈希特征值的方式来测定错误。

(See the section called “Object storage format” for the details of the object formatting and SHA1 calculation.)

（参考“对象的存储格式”一节，有对象的格式和 SHA1 算法的详细介绍）

There are four different types of objects: "blob", "tree", "commit", and "tag".

有四种不同的对象形式：“片”，“树”，“交付”，和“标签”。

- A "blob" object is used to store file data.

- A "tree" object ties one or more "blob" objects into a directory structure. In addition, a tree object can refer to other tree objects, thus creating a directory hierarchy.
- A "commit" object ties such directory hierarchies together into a directed acyclic graph of revisions—each commit contains the object name of exactly one tree designating the directory hierarchy at the time of the commit. In addition, a commit refers to "parent" commit objects that describe the history of how we arrived at that directory hierarchy.
- A "tag" object symbolically identifies and can be used to sign other objects. It contains the object name and type of another object, a symbolic name (of course!) and, optionally, a signature.
- “片” 对象是保存文件的数据的。
- “树” 对象将 “片” 对象以目录结构的形式绑定起来，还有，树对象可以引用到其他的树对象，从而形成了目录层次关系。
- “交付” 对象将每次所提交的修订中所包含的对象名，以一个定向扩展的视图为目录结构形式绑定起来，这个树形结构准确地构建出每次提交的目录层次。另外，一个交付还指向他的父交付，这就是我们所看到的历史发展的层次关系。
- “标签” 对象用于标记其他的对象，它包含该对象的名称，类型，以及这个标记自己的名称，可选的还有签名。

The object types in some more detail:

对象类型详细介绍：

Commit Object 交付对象

The "commit" object links a physical state of a tree with a description of how we got there and why. Use the —pretty=raw option to git-show(1) or git-log(1) to examine your favorite commit:

“交付” 对象连接某个树对象的物理状态，并附带有如何与为什么到达该处的说明。在 git-show(1) 中使用 --pretty=raw 选项，或 git-log(1) 检查你需要的交付：

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
```

```
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

    Fix misspelling of 'suppress' in docs

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

As you can see, a commit is defined by:

你可以看到交付对象的定义：

- a tree: The SHA1 name of a tree object (as defined below), representing the contents of a directory at a certain point in time.
- parent(s): The SHA1 name of some number of commits which represent the immediately previous step(s) in the history of the project. The example above has one parent; merge commits may have more than one. A commit with no parents is called a "root" commit, and represents the initial revision of a project. Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea).
- an author: The name of the person responsible for this change, together with its date.
- a committer: The name of the person who actually created the commit, with the date it was done. This may be different from the author, for example, if the author was someone who wrote a patch and emailed it to the person who used it to create the commit.
- a comment describing this commit.
- 一个树对象： 一个树对象的 SHA1 名称 （在交付对象名的下面），他表达的是一个确定的时间点上的目录内容。
- 父交付： 一个或多个交付的 SHA1 名称，表达项目历史的上一步。上面的例子是一个父交付；合并交付对象就可能不止一个父交付。没有父交付的交付对象称为 “根”交付。表示项目的初始版本，每个

项目必须至少有一个根交付。可能某个项目有不止一个根，不过那不是通常的情形（可能你需要一个好的想象力来理解这种情况）。

- 作者： 表明是谁在什么时间创建的这个交付，不过可能存在不止一个作者的情况，譬如，某甲写了一个补丁用电子邮件发给了某乙，某乙用它来创建了这个交付。
- 交付的注释。

Note that a commit does not itself contain any information about what actually changed; all changes are calculated by comparing the contents of the tree referred to by this commit with the trees associated with its parents. In particular, git does not attempt to record file renames explicitly, though it can identify cases where the existence of the same file data at changing paths suggests a rename. (See, for example, the -M option to git-diff(1)).

注意，一个交付它本身不包含任何实际上的变更信息。所有的变更依靠计算这个交付本身所引用的那个树对象与这个交付的父交付所引用的树对象的内容比较得出。事实上，git 不会视图真正地纪录文件的重命名，尽管它可以将现存的文件数据的路径变更看成是文件的重命名。（参考例子， 在 git-diff(1) 中使用 -M 选项）。

A commit is usually created by git-commit(1), which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

一个交付正常是由 git-commit(1) 创建，通常情况下以当前的 HEAD 作为它的父交付，他所引用的树对象则来自当前索引中所保存的内容。

Tree Object 树对象

The ever-versatile git-show(1) command can also be used to examine tree objects, but git-ls-tree(1) will give you more details:

git-show(1) 命令是个多面手，它同样可以用来检验树对象，不过 git-ls-tree(1) 会给出更多的细节：

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap
```

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README
...
```

As you can see, a tree object contains a list of entries, each with a mode, object type, SHA1 name, and name, sorted by name. It represents the contents of a single directory tree.

正如你所看到的一样，一个树对象包含一个条目的列表，每个条目带有模式，对象类型，SHA1 名称，自然名称，条目按名称顺序排列。它描述了一个单层的树形目录结构。

The object type may be a blob, representing the contents of a file, or another tree, representing the contents of a subdirectory. Since trees and blobs, like all other objects, are named by the SHA1 hash of their contents, two trees have the same SHA1 name if and only if their contents (including, recursively, the contents of all subdirectories) are identical. This allows git to quickly determine the differences between two related tree objects, since it can ignore any entries with identical object names.

对象的类型可能是一个片对象，表示一个文件的内容，或者是一个树对象，表示一个子目录。即使是树对象与片对象，也和其他的对象一样，也是用 它们的内容的 SHA1 哈希特征值来命名，当且仅当它们内容是一样（包括所有的子目录的递归内容）的时候两个树对象的名字是一样的。这就可以令 git 快速地测定两个关联的树对象之间的差异，那么他就可以忽略有相同对象名的条目。

(Note: in the presence of submodules, trees may also have commits as entries. See Chapter 8, Submodules for documentation.)

（注意： 当出现子模块时，树对象可能还包含交付对象的条目。参考 第八章，文档子模块）

Note that the files all have mode 644 or 755: git actually only pays attention to the executable bit.

注意当文件有 644 或 755 模式时： git 才会真正地将该文件作为可执行的二进制文件来对待。

Blob Object 片对象

You can use git-show(1) to examine the contents of a blob; take, for example, the blob in the entry for "COPYING" from the tree above:

你可以用 git-show(1) 来检验片对象的内容；譬如，上面的树对象中的 “COPYING” 条目。

```
$ git show 6ff87c4664

 Note that the only valid version of the GPL as far as this project
 is concerned is _this_ particular version of the license (ie v2, not
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.
...
```

A "blob" object is nothing but a binary blob of data. It doesn't refer to anything else or have attributes of any kind.

一个 “片”对象除了一个二进制的数据片之外没有任何东西。它不引用任何东西，也没有任何类型属性。

Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with.

整个片对象都是定义自己的数据，如果两个文件在一个树对象中（或者是在版本库中有多重的不同版本）有相同的内容，他们将共享同一个片对象。片对象总是与它所处的目录树无关的，重命名文件并不能改变文件与片对象的关联性。

Note that any tree or blob object can be examined using git-show(1) with the <revision>:<path> syntax. This can sometimes be useful for browsing the contents of a tree that is not currently checked out.

注意任何树和片都可以用 git-show(1) 命令带上 <revision>:<path> 语法进行检验。这个方法有时对于浏览并不是当前提取的树对象的内容非常有用。

Trust 信赖

If you receive the SHA1 name of a blob from one source, and its contents from another (possibly untrusted) source, you can still trust that those contents are correct as long as the SHA1 name agrees. This is because the SHA1 is designed so that it is infeasible to find different contents that produce the same hash.

如果你从一个来源收到一个片对象的名称，并且从另一个来源收到它的内容（可能是不被信赖的），你仍然可以信赖它的内容，只要它能通过 SHA1 名称的许可。这是因为 SHA1 算法就设计成对于不同的内容不可能产生相同的哈希特征值。

Similarly, you need only trust the SHA1 name of a top-level tree object to trust the contents of the entire directory that it refers to, and if you receive the SHA1 name of a commit from a trusted source, then you can easily verify the entire history of commits reachable through parents of that commit, and all of those contents of the trees referred to by those commits.

类似地，你应该以只信赖顶层树对象的 SHA1 名称的方式来信赖它所指向的整个目录的内容，并且如果你收到从一个可信的来源得到的交付对象的 SHA1 名称，那么你就可以很容易通过该交付对象的父交付的所有可及交付来验证整个历史进程，以及这些交付所指向的树对象的内容。

So to introduce some real trust in the system, the only thing you need to do is to digitally sign just one special note, which includes the name of a top-level commit. Your digital signature shows others that you trust that commit, and the immutability of the history of commits tells others that they can trust the whole history.

正式依据系统的信赖机制，你唯一需要做的事情就是以数字形式做一个特殊的签注，它包含在顶层交付中。你的数字签注告诉别人可以信赖那个交付，已经交付历史的不可变更性表明了整个历史是可以信赖的。

In other words, you can easily validate a whole archive by just sending out a single email that tells the people the name (SHA1 hash) of the top commit, and digitally sign that email using something like GPG/PGP.

换言之，你可以容易地确认整个档案是有效的，只需要发出一个顶层交付的名称（SHA1 哈希特征值）给其他的人，并且以 GPG/PGP 等加密邮件。

- **译注：** GPG/PGP

To assist in this, git also provides the tag object…

作为这个机制的辅佐，git 还提供了标签对象...

Tag Object 标签对象

A tag object contains an object, object type, tag name, the name of the person ("tagger") who created the tag, and a message, which may contain a signature, as can be seen using git-cat-file(1):

一个标签对象包含一个对象，对象类型，标签名称，标签作者（“标记者”）的名称，以及一个注释，可能还包含一个密匙，这个密匙可以用 git-cat-file(1) 来查看。

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

See the git-tag(1) command to learn how to create and verify tag objects. (Note that git-tag(1) can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with "refs/tags/").

参考 git-tag(1) 命令学习如何创建一个核实标签对象。（注意，git-tag(1) 同样可以用来创建一个 “轻量标签”，其实它并不是一个真实的标签对象，只是在 "refs/tags/" 中的简单引用）。

How git stores objects efficiently: pack files | git 如何高效地储存对象： 打包文件

Newly created objects are initially created in a file named after the object's SHA1 hash (stored in .git/objects).

创建一个新的对象就是以一个哈希特征值作为文件名创建一个新的文件（保存在 .git/objects 中）。

Unfortunately this system becomes inefficient once a project has a lot of objects. Try this on an old project:

不幸的是，一旦系统有很多对象的时候，它就会变得很慢。你试试在一个旧项目中运行：

```
$ git count-objects
6930 objects, 47620 kilobytes
```

The first number is the number of objects which are kept in individual files. The second is the amount of space taken up by those "loose" objects.

第一个数字是有多少个独立的文件被保存了。第二个数字是那些 “松散” 对象所占的磁盘空间。

You can save space and make git faster by moving these loose objects in to a "pack file", which stores a group of objects in an efficient compressed format; the details of how pack files are formatted can be found in technical/pack-format.txt.

你可以通过将那些松散对象打包到 “包文件” 的方式来节省磁盘空间和提高 git 的速度，这种发放是以高效压缩的方式将对象分组储存；关于包文件的格式可以在 techincal/pack-formate.txt 文件中找到。

To put the loose objects into a pack, just run git repack:

要将松散对象打包，只需要运行 git repack:

```
$ git repack
Generating pack...
Done counting 6020 objects.
Deltifying 6020 objects.
 100% (6020/6020) done
Writing 6020 objects.
 100% (6020/6020) done
Total 6020, written 6020 (delta 4070), reused 0 (delta 0)
Pack pack-3e54ad29d5b2e05838c75df582c65257b8d08e1c created.
```

You can then run

接着可以运行

```
$ git prune
```

to remove any of the "loose" objects that are now contained in the pack. This will also remove any unreferenced objects (which may be created when, for example, you use "git-reset" to remove a commit). You can verify that the loose objects are gone by looking at the .git/objects directory or by running

就将已经打包的松散对象删除。这将会删除掉那些未被引用的对象（譬如，它们可能是由于你运行 "git-reset" 时遗弃了某些交付而产生的）。你可以通过查看 .git/objects 或者运行下面的命令来查看松散对象是否已经被清除了。

```
$ git count-objects
0 objects, 0 kilobytes
```

Although the object files are gone, any commands that refer to those objects will work exactly as they did before.

尽管对象文件已经被清除了，任何对对象操作的命令都将同样准确，就像他们仍然存在一样。

The git-gc(1) command performs packing, pruning, and more for you, so is normally the only high-level command you need.

git-gc(1) 命令一次完成你上面的打包，剪除工作。它只是提供方便的高层命令。

Dangling objects 悬空对象

The git-fsck(1) command will sometimes complain about dangling objects. They are not a problem.

有时 git-fsck(1) 命令会因为悬空对象带来点烦恼。不过这个不是一个问题。

The most common cause of dangling objects is that you've rebased a branch, or you have pulled from somebody else who rebased a branch—see Chapter 5, Rewriting history and maintaining patch series. In that case, the old head of the original branch still exists, as does everything it pointed to. The branch pointer itself just doesn't, since you replaced it with another one.

最常见的造成悬空对象的原因是你重定位了一个分支，或者是拉入了某个人的曾经重定位的分支（参考第五章），并在那时重写过历史和维护过补丁串。在这种情况下，老的源头分支的头仍然存在，就如它所指向的所有东西仍然存在一样。但是分支指向自身的指针却没有了，直至你用另外一个指针覆盖它。

There are also other situations that cause dangling objects. For example, a "dangling blob" may arise because you did a "git-add" of a file, but then, before you actually committed it and made it part of the bigger picture, you changed something else in that file and committed that updated thing—the old state that you added originally ends up not being pointed to by any commit or tree, so it's now a dangling blob object.

还存在其他造成悬空对象的情况。譬如，悬空对象可能会这样出现，你对一个文件做 "git-add"。不过，你之前已经提交过它并使它成为一个较大的交付的一个部分，你又变更过那个较大的交付中的某些这个文件之外的其他东西，并提交了那些更新了的内容 - 你在最后加入的状态就已经没有任何交付和树对象指向它了，所以它现在成了悬空的片对象。

Similarly, when the "recursive" merge strategy runs, and finds that there are criss-cross merges and thus more than one merge base (which is fairly unusual, but it does happen), it will generate one temporary midway tree (or possibly even more, if you had lots of

criss-crossing merges and more than two merge bases) as a temporary internal merge base, and again, those are real objects, but the end result will not end up pointing to them, so they end up "dangling" in your repository.

类似地，要是在 “递归” 合并运行时，发现存在十字合并多于一个合并的基点（这样的情况相当不寻常，不过的确存在），合并操作会产生一个临时的中间阶段的树对象（甚至可能会产生更多的临时树，如果存在更多的十字合并和多于两个的合并基点）作为临时的内部合并基点，再去合并其他的东西，这个临时的树对象是一个真实的对象，只不过操作完成了之后就在没有指针指向它了，最终他们成了悬空在你的版本库中。

Generally, dangling objects aren't anything to worry about. They can even be very useful: if you screw something up, the dangling objects can be how you recover your old tree (say, you did a rebase, and realized that you really didn't want to—you can look at what dangling objects you have, and decide to reset your head to some old dangling state).

总的来说，不必担心悬空对象的存在。他们甚至是有用的：如果你搞砸了某些东西的时候，悬空对象可以让你知道如何恢复旧的树对象（比方说，你做重定位，然后察觉到还不应该这样做，你可以查看以下你有什么悬空对象，然后决定你将分支头至位到什么悬空对象的状态）。

For commits, you can just use:

对于交付对象，你只需要：

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

This asks for all the history reachable from the given commit but not from any branch, tag, or other reference. If you decide it's something you want, you can always create a new reference to it, e.g.,

这将要求列举所有对于指定的交付来说有历史可及性的对象，而不是从分支，标签或者是其他的引用中查询。如果你决定从悬空对象中取回什么东西，你可以创建一个新的引用来指向它，譬如：

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

For blobs and trees, you can't do the same, but you can still examine them. You can just do

对于片和树，你就不能这样做了，不过你仍然可以检测它们，你只要：

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

to show what the contents of the blob were (or, for a tree, basically what the "ls" for that directory was), and that may give you some idea of what the operation was that left that dangling object.

这样可以看到片对象中的内容（或者，对于树对象，就像对一个目录运行 'ls' 命令一样），通过这样也许会提示到你在悬空对象中留下了什么东西。

Usually, dangling blobs and trees aren't very interesting. They're almost always the result of either being a half-way mergebase (the blob will often even have the conflict markers from a merge in it, if you have had conflicting merges that you fixed up by hand), or simply because you interrupted a "git-fetch" with ^C or something like that, leaving some of the new objects in the object database, but just dangling and useless.

通常地，我们对悬空的片和树都不太感兴趣。他们几乎总是多路合并的中间过程留下的临时合并基点（片甚至会经常有冲突标记，如果你进行的是一个有冲突的合并的话，并且你要手动处理它），或者是你在运行 "git-fetch" 命令的时候按了 Ctrl+C 之类的操作，你就会留下某些东西在对象数据库中，这也是悬空对象并且是毫无意义的悬空对象。

Anyway, once you are sure that you're not interested in any dangling state, you can just prune all unreachable objects:

好了，一旦你确定你对悬空对象的情况已经不感兴趣，你就可以剪除所有这些不可及的对象了：

```
$ git prune
```

and they'll be gone. But you should only run "git prune" on a quiescent repository—it's kind of like doing a filesystem fsck recovery: you don't want to do that while the filesystem is mounted.

于是悬空对象就被清除了，不过你之应该在一个没有运行其他操作的版本库中运行 "git prune"，这个就类似用 fsck 来进行文件系统的修复一样：你不应该在一个已经挂载了的文件系统中运行 fsck 。

(The same is true of "git-fsck" itself, btw, but since git-fsck never actually **changes** the repository, it just reports on what it found, git-fsck itself is never "dangerous" to run. Running it while somebody is actually changing the repository can cause confusing and scary messages, but it won't actually do anything bad. In contrast, running "git prune" while somebody is actively changing the repository is a **BAD** idea).

（"git-fsck"命令本身是可以信赖的，事实上 git-fsck 并不会**改变**版本库的任何东西，它仅仅会报告它检查到了什么，运行 git-fsck 命令本身绝对不“危险”。当有人正在更改版本库的时候，运行这个命令就会得到混乱和吓人的信息，不过这其实仍然不会搞坏任何事情。相对地，当有人正在改变版本库的时候，你同时运行 "git prune" 那就是**坏**主意了）。

Recovering from repository corruption 从损坏中恢复

By design, git treats data trusted to it with caution. However, even in the absence of bugs in git itself, it is still possible that hardware or operating system errors could corrupt data.

就设计而言，git 对待数据是很谨慎小心的。不过，git 自身的漏洞，硬件和操作系统的错误等都可能造成数据的损坏。

The first defense against such problems is backups. You can back up a git directory using clone, or just using cp, tar, or any other backup mechanism.

首要的预防措施就是备份，你可以通过克隆来备份 git 的目录，或者是用 cp, tar, 等任何可能的备份方法。

As a last resort, you can search for the corrupted objects and attempt to replace them by hand. Back up your repository before attempting this in case you corrupt things even more in the process.

作为最后的办法，你可以查找损坏的对象并尝试用手工的方式来修复他们。在进行这些尝试之前，你要备份你的版本库，因为你可能在修复的过程中造成更多的损坏。

We'll assume that the problem is a single missing or corrupted blob, which is sometimes a solvable problem. (Recovering missing trees and especially commits is much harder).

我们假设只是丢失和损坏了一个的片对象，某些时候这种情况是可能修复的。（修复丢失的树和特殊的交付就要困难的多了）。

Before starting, verify that there is corruption, and figure out where it is with git-fsck(1); this may be time-consuming.

开始修复之前，检查是否有损坏存在，并确定损坏发生在什么地方；这可能是挺费时的。

Assume the output looks like this:

假设输出的信息大致如下：

```
$ git fsck --full
broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
               to    blob 4b9458b3786228369c63936db65827de3cc06200
missing blob 4b9458b3786228369c63936db65827de3cc06200
```

(Typically there will be some "dangling object" messages too, but they aren't interesting.)

（典型的情况是还存在一些“悬空对象”的信息，不过这不是我们感兴趣的东西。）

Now you know that blob 4b9458b3 is missing, and that the tree 2d9263c6 points to it. If you could find just one copy of that missing blob object, possibly in some other repository, you could move it into .git/objects/4b/9458b3… and be done. Suppose you can't. You can still examine the tree that pointed to it with git-ls-tree(1), which might output something like:

现在你知道片 4b9458b3 丢失了，并且树 2d9263c6 指向它。要是你能够在你的版本库中发现这个片对象的副本的话，这在某些版本库中是可能的，你只要将这个副本移动到 .git/objects/4b/9458b3... 中，这样就算是修复它了。假如你找不到，你仍然可以用 git-ls-tree 命令以检查指向它的树的方式来得到类似下面的信息输出：

```
$ git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
```

```
100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore
100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING
...
100644 blob 4b9458b3786228369c63936db65827de3cc06200    myfile
...
```

So now you know that the missing blob was the data for a file named "myfile". And chances are you can also identify the directory—let's say it's in "somedirectory". If you're lucky the missing copy might be the same as the copy you have checked out in your working tree at "somedirectory/myfile"; you can test whether that's right with git-hash-object(1):

这样你就知道了丢失的片是文件 "myfile" 的数据。你现在有机会确认一下那个目录——应该说是某些目录。如果你幸运的话，丢失的东西就在你提出出来的工作树的 "somedirectory/myfile" 的位置上；你可以通过 git-hash-object(1) 命令来确定一下文件的内容是否正确。

```
$ git hash-object -w somedirectory/myfile
```

which will create and store a blob object with the contents of somedirectory/myfile, and output the sha1 of that object. if you're extremely lucky it might be 4b9458b3786228369c63936db65827de3cc06200, in which case you've guessed right, and the corruption is fixed!

这样会创建一个新的对象，这个片对象以 somedirectory/myfile 的数据为内容，并且会输出这个对象的 sha1。要是你超级幸运的话，输出将会是 4b9458b3786228369c63936db65827de3cc06200，这中情况就等于你搞定了，损坏已经被修复。

Otherwise, you need more information. How do you tell which version of the file has been lost?

不然的话，你还需要更多的信息来告诉你是那个版本的文件丢失了？

The easiest way to do this is with:

通过下面的命令就很容易做到：

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

Because you're asking for raw output, you'll now get something like

因为你是要求获得文件的原始数据，现在你看到的东西大概像下面那样：

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


commit xyz
Author:
Date:

...
:100644 100644 oldsha... 4b9458b... M somedirectory/myfile
```

This tells you that the immediately preceding version of the file was "newsha", and that the immediately following version was "oldsha". You also know the commit messages that went with the change from oldsha to 4b9458b and with the change from 4b9458b to newsha.

这会告诉你当前处理的文件版本是 "newsha"，并且紧接着的版本叫 "oldsha"。你还可以看到从 oldsha 到 469458b 以及从 4b9458 交付的信息。

If you've been committing small enough changes, you may now have a good shot at reconstructing the contents of the in-between state 4b9458b.

如果你的每个交付都足够短的话，你现在最好的办法是重新将文件的内容弥补至 4b9458b 的状态。

If you can do that, you can now recreate the missing object with

如果你可以将内容补充过来，你要重新生成这个丢失的对象。

```
$ git hash-object -w <recreated-file>
```

and your repository is good again!

这样你的版本库又回归到良好状态了！

(Btw, you could have ignored the fsck, and started with doing a

（顺便说一下，你完全可以跳过 fsck，并且直接用下面的命令开始

```
$ git log --raw --all
```

and just looked for the sha of the missing object (4b9458b..) in that whole thing. It's up to you - git does have a lot of information, it is just missing one particular blob version.

查找丢失的对象的 sha 哈希特征值。这完全取决于你 — 要是你不在乎仅仅是为了一个丢失的片版本，而检查 git 生成的一大堆的信息的话。

The index 索引

The index is a binary file (generally kept in .git/index) containing a sorted list of path names, each with permissions and the SHA1 of a blob object; git-ls-files(1) can show you the contents of the index:

索引是一个二进制文件（总是保存在为 .git/index）保存着一个有序的文件路径名称列表，以及每个文件的权限权限和每个片对象的 SHA1 哈希特征值；git-ls-files 可以向你展示索引中的内容：

```
$ git ls-files --stage
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0       .gitignore
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0       .mailmap
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0       COPYING
```

```
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0
Documentation/.gitignore
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0
Documentation/Makefile
...
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0       xdiff/xtypes.h
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0       xdiff/xutils.c
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0       xdiff/xutils.h
```

Note that in older documentation you may see the index called the "current directory cache" or just the "cache". It has three important properties:

注意，在旧的的 git 文档中，你可能会看到索引被叫做 “当前目录缓存” 或者只是叫 “cache”。他有三个主要的特性：

- The index contains all the information necessary to generate a single (uniquely determined) tree object.
- 索引保存着所有生成（唯一确定的）树对象的所有必要信息。
- For example, running git-commit(1) generates this tree object from the index, stores it in the object database, and uses it as the tree object associated with the new commit.
- 例如，运行 git-commit(1) 从索引中产生一个树对象，将它保存到对象数据库中，并且用它来作为树对象和新的交付的关联纽带。
- The index enables fast comparisons between the tree object it defines and the working tree.
- 索引可以加快树对象和工作目录树之间的差异比较。
- It does this by storing some additional data for each entry (such as the last modified time). This data is not displayed above, and is not stored in the created tree object, but it can be used to determine quickly which files in the working directory differ from what was stored in the index, and thus save git from having to read all of the data from such files to look for changes.

- 它还保存每个条目的其他的一些附带信息（譬如最后的修改日期）。这些数据没有在上面的输出中显示，也不会被保存到创建的树对象中，不过它有利于快速比较工作树和索引中保存的数据的差异，避免了 git 要查找变更的时候去读取每个文件的内容。
- It can efficiently represent information about merge conflicts between different tree objects, allowing each pathname to be associated with sufficient information about the trees involved that you can create a three-way merge between them.
- 他可以高效地描述不同的树对象之间的合并冲突，容许树对象所涉及的每个文件名都可以关联到充分的信息，令你可以以三路合并的方式来合并它们。
- We saw in the section called “Getting conflict-resolution help during a merge” that during a merge the index can store multiple versions of a single file (called "stages"). The third column in the git-ls-files(1) output above is the stage number, and will take on values other than 0 for files with merge conflicts.
- 我们看过了 “在合并中取得冲突解决帮助” 一节，我们看到了在合并过程中，索引保存了一个单一文件的多个不同版本（称为 “阶段”）。git-ls-files(1) 输出中的第三列就是阶段的号码，并且当合并出现冲突时会有多于 0 的其他编号。

The index is thus a sort of temporary staging area, which is filled with a tree which you are in the process of working on.

索引是一个临时的保存场所，它由一个你正在上面工作的目录树来填充。

If you blow the index away entirely, you generally haven't lost any information as long as you have the name of the tree that it described.

如果你扔掉整个的索引，你总是不会丢失任何信息，你的树对象中已经描述了这些文件的名称。

Chapter 8. Submodules 子模块

Large projects are often composed of smaller, self-contained modules. For example, an embedded Linux distribution's source tree would include every piece of software in the distribution with some local modifications; a movie player might need to build against a specific, known-working version of a decompression library; several independent programs might all share the same build scripts.

大型项目经常是由更小的，自包含的模块组成。譬如，一个嵌入式 Linux 发行版的源码树会包含许多的软件的补丁集，包含了该发行版对相应的软件的修改；也许一个视频播放器需要针对某些特性进行重新编译，已知的解压库的稳定版本；一系列的依赖程序可能在编译脚本中检查。

With centralized revision control systems this is often accomplished by including every module in one single repository. Developers can check out all modules or only the modules they need to work with. They can even modify files across several modules in a single commit while moving things around or updating APIs and translations.

对于中心型的修订控制系统，经常是将所有的模块都纺织在单个的版本库中。开发者可以提取所有的模块或者是单个的模块。当事情的变化涉及到 APIs 和翻译的时候，他们甚至可以在一个交付中包含跨越多个模块的文件修改。

Git does not allow partial checkouts, so duplicating this approach in Git would force developers to keep a local copy of modules they are not interested in touching. Commits in an enormous checkout would be slower than you'd expect as Git would have to scan every directory for changes. If modules have a lot of local history, clones would take forever.

Git 是不允许部分地提取的，这种副本式的实现方式，强制开发者在本地保持即使是他不感兴趣的模块的拷贝。巨大的提出操作可能比你预期的还要慢，当 Git 需要扫描每个目录的变动的时候。要是模块有一个相当长的历史发展过程的话，克隆操作会将这些历史始终保持。

On the plus side, distributed revision control systems can much better integrate with external sources. In a centralized model, a single arbitrary snapshot of the external project is exported from its own revision control and then imported into the local revision control on a vendor branch. All the history is hidden. With distributed revision control you can clone the entire external history and much more easily follow development and re-merge local changes.

从扩展的观点来看，分布式修订控制系统可以更好地整合外部资源。在中心修订控制的模式下，一个扩展的项目任意快照是从它自己的修订控制系统中导出，并且导入到本地修订控制系统的发行者分支中。所有的发展历史都是隐藏起来的。对于分布式修订控制系统，你可以克隆完整的历史，并很容易地跟随开发和对本地变动进行重新合并。

Git's submodule support allows a repository to contain, as a subdirectory, a checkout of an external project. Submodules maintain their own identity; the submodule support just stores the submodule repository location and commit ID, so other developers who clone the containing project ("superproject") can easily clone all the submodules at the same revision. Partial checkouts of the superproject are possible: you can tell Git to clone none, some or all of the submodules.

Git 的子模块支持容许版本库将扩展项目作为一个子目录那样包含，和提取。子模块维护它自己的验证机制；子模块只是支持保存子模块版本库地点和交付 ID，那么那些克隆上一级项目的开发者可以容易地克隆所有的子模块：你可以不必告诉 git 克隆某些或者是全部子模块。

The git-submodule(1) command is available since Git 1.5.3. Users with Git 1.5.2 can look up the submodule commits in the repository and manually check them out; earlier versions won't recognize the submodules at all.

git-submodule(1)命令从 1.5.3 版本开始支持。对于 Git 1.5.2 的用户可以手动查找子模块中的交付和提取它们。更早版本的 git 将无法识别子模块了。

To see how submodule support works, create (for example) four example repositories that can be used later as a submodule:

既然知道了子模块已经至此了，创建四个样板版本库（作为例子）用作子模块。

```
$ mkdir ~/git
$ cd ~/git
$ for i in a b c d
do
        mkdir $i
        cd $i
        git init
```

```
        echo "module $i" > $i.txt
        git add $i.txt
        git commit -m "Initial commit, submodule $i"
        cd ..
done
```

Now create the superproject and add all the submodules:

现在创建一个上级项目并加入所有的子模块：

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

- **Note 注意**
- Do not use local URLs here if you plan to publish your superproject!
- 如果你打算发布这个上级版本库就不要用本地路径作为 URLs 了！

See what files git-submodule created:

该查看一下 git-submodule 都创建了什么了：

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

The git-submodule add <repo> <path> command does a couple of things:

git-submodule add <repo> <path> 命令做了如下的事情：

- It clones the submodule from <repo> to the given <path> under the current directory and by default checks out the master branch.

- It adds the submodule's clone path to the gitmodules(5) file and adds this file to the index, ready to be committed.
- It adds the submodule's current commit ID to the index, ready to be committed.
- 它从 <repo> 子模块克隆到 <path> 指定的目录，并默认地提取 master 分支。
- 它将子模块的克隆路径加入到 .gitmodules(5) 文件中，并将该文件加入索引，也已经提交了该文件。
- 它将子模块的当前交付 ID 加入到索引中，并被提交。

Commit the superproject:

提交上级项目：

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

Now clone the superproject:

现在克隆上级项目：

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

The submodule directories are there, but they're empty:

子模块的目录已经存在了，不过他们是空的：

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

- **Note 注意**
- The commit object names shown above would be different for you, but they should match the HEAD commit object names of your repositories. You can check it by running git ls-remote ../a.
- 上面显示的对象名对于你来说应该是不同的，不过他们应该匹配你的版本库中的 HEAD 交付的对象名。你可以通过 git ls-remote ../a 命令将他们展示出来。

Pulling down the submodules is a two-step process. First run git submodule init to add the submodule repository URLs to .git/config:

拉入子模块的操作由两步组成，首先运行 git submodule init 将子模块的版本库 URLs 加入到 .git/config：

```
$ git submodule init
```

Now use git-submodule update to clone the repositories and check out the commits specified in the superproject:

接着运行 git submodule update 克隆并提取交付到特定的上级项目：

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

One major difference between git-submodule update and git-submodule add is that git-submodule update checks out a specific commit, rather than the tip of a branch. It's like checking out a tag: the head is detached, so you're not working on a branch.

git submodule update 和 git submodule add 的主要区别是 git submodule update 提取特定的交付，而不是分支的顶端。它类似提取一个标签：头被剥离，如此你并不是工作在分支上。

```
$ git branch
* (no branch)
```

```
  master
```

If you want to make a change within a submodule and you have a detached head, then you should create or checkout a branch, make your changes, publish the change within the submodule, and then update the superproject to reference the new commit:

如果你在子模块中进行了变动并且你处在剥离的头上，那么你应该创建或者是提取一个分支，作出你的更改，将这些更改发布到子模块中，并更新上级项目到一个新的交付引用。

```
$ git checkout master
```

or

或者

```
$ git checkout -b fix-up
```

then

于是

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

```
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

You have to run git submodule update after git pull if you want to update submodules, too.

你必须在运行 git pull 之后运行 git submodule update，如果你也想更新子模块的话。

Pitfalls with submodules 子模块陷阱

Always publish the submodule change before publishing the change to the superproject that references it. If you forget to publish the submodule change, others won't be able to clone the repository:

总是需要先将子模块的变更发布到指向它的上级项目，再将上级项目发布。如果你忘记了发布子模块的变动，其他人将无法克隆版本库：

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

You also should not rewind branches in a submodule beyond commits that were ever recorded in any superproject.

你同样不应该逆转在子模块中的分支，更多的交付其实是保存在上级项目中的。

It's not safe to run git submodule update if you've made and committed changes within a submodule without checking out a branch first. They will be silently overwritten:

如果你并不是在首先提取一个分支的情况下运行 git submodule update 是不安全的。他们会被无声地覆盖：

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

- **Note 注意**
- The changes are still visible in the submodule's reflog.
- 变更仍然在子模块的引用日志中可见。

This is not the case if you did not commit your changes.

这不算是什么问题，如果你没有提交你的变更的话。

Chapter 9. Low-level git operations 第九章. 底层 git 操作

Many of the higher-level commands were originally implemented as shell scripts using a smaller core of low-level git commands. These can still be useful when doing unusual things with git, or just as a way to understand its inner workings.

很多的高层的命令通过运行更小的底层 git 命令的 shell 脚本的方式实现的。底层命令在日常的 git 事务中也是有用的，或者仅是作为理解 git 的内部运作也是很有帮助的。

Object access and manipulation 对象访问与操作

The git-cat-file(1) command can show the contents of any object, though the higher-level git-show(1) is usually more useful.

git cat-file(1) 命令可以展示任何对象的内容，尽管高层命令 git show(1)通常更有力。

The git-commit-tree(1) command allows constructing commits with arbitrary parents and trees.

git commit-tree(1)命令容许构建任意的交付为父交付的交付以及树对象。

A tree can be created with git-write-tree(1) and its data can be accessed by git-ls-tree(1). Two trees can be compared with git-diff-tree(1).

一个树对象可以由 git write-tree(10) 创建并且它的数据可以用 git ls-tree(1)命令访问。两个树可以用 git diff-tree(1)来进行差异比较。

A tag is created with git-mktag(1), and the signature can be verified by git-verify-tag(1), though it is normally simpler to use git-tag(1) for both.

git mktage(1)命令用于创建标签，并且用 git verifgy-tage(1)来验证其加密。不过通常用 git tag(1) 命令来同时完成这两个步骤。

The Workflow 运作流程

High-level operations such as git-commit(1), git-checkout(1) and git-reset(1) work by moving data between the working tree, the index, and the object database. Git provides low-level operations which perform each of these steps individually.

高层操作，例如 git commit(1), git checkout(1) 以及 git reset(1) 用于在工作树，索引，与对象数据库中进行数据转移，实际上 git 提供的是更细分的底层操作。

Generally, all "git" operations work on the index file. Some operations work purely on the index file (showing the current state of the index), but most operations move data between the index file and either the database or the working directory. Thus there are four main combinations:

总地来说，所有 "git" 对索引文件的操作，都是纯粹操作索引文件（展示索引的当前状态），不过大部分的命令都是在索引文件，数据库或者是工作树之间迁移数据。存在四个主要的组合：

working directory -> index 工作树 -> 索引

The git-update-index(1) command updates the index with information from the working directory. You generally update the index information by just specifying the filename you want to update, like so:

git update-index(1) 命令用来自工作树的信息刷新索引。你通常以指定文件名的方式来刷新索引信息，例如：

```
$ git update-index filename
```

but to avoid common mistakes with filename globbing etc, the command will not normally add totally new entries or remove old entries, i.e. it will normally just update existing cache entries.

不过为了避免出现常见的文件名替换这类的错误，命令正常情况下总是不会加入一个新的条目或者是删除旧的条目。它仅仅是刷新现存的缓存条目。

To tell git that yes, you really do realize that certain files no longer exist, or that new files should be added, you should use the —remove and —add flags respectively.

正确告诉 git 增加或者删除文件的方法是，当你的确认为某些现存的文件已经没有再存在的必要，或者是需要加入新的文件的话，你应该分别地使用 --remove 和 --add 参数。

NOTE! A —remove flag does not mean that subsequent filenames will necessarily be removed: if the files still exist in your directory structure, the index will be updated with their new status, not removed. The only thing —remove means is that update-index will

be considering a removed file to be a valid thing, and if the file really does not exist any more, it will update the index accordingly.

注意！--remove 参数并不意味着这个参数的作用域中的文件名必须被删除：如果该文件仍然存在于你的目录结构中，索引只是更新了它们的状态，而不是删除它们。--remove 参数的唯一意义是表示删除那个文件是合法的。并且当那个文件的确已经不再存在的时候，索引会相应地更新自己的状态。

As a special case, you can also do git update-index —refresh, which will refresh the "stat" information of each index to match the current stat information. It will not update the object status itself, and it will only update the fields that are used to quickly test whether an object still matches its old backing store object.

作为特例，你可以运行一下 git update-index --refresh，这将会更新每个索引所匹配的当前状态信息。它不会更新对象的状态，并且它只会刷新那些用于快速检查一个对象是否仍然匹配它旧的后台储存的对象的那些索引域。

The previously introduced git-add(1) is just a wrapper for git-update-index(1).

前面介绍过的 git-add(1) 只是 git-update-index(1) 命令的一个封装。

index -> object database 索引 -> 对象数据库

You write your current index file to a "tree" object with the program

你要将当前索引的文件写入一个 "树" 对象用这个命令

```
$ git write-tree
```

that doesn't come with any options—it will just write out the current index into the set of tree objects that describe that state, and it will return the name of the resulting top-level tree. You can use that tree to re-generate the index at any time by going in the other direction:

它不需要任何参数选项，它不过是将当前的索引状态写进一个描述这个状态的树对象的集合，并且他返回该树对象集的顶层树。你可以在任何时候在其他的目录下面用它来重现索引：

object database -> index 对象数据库 -> 索引

You read a "tree" file from the object database, and use that to populate (and overwrite—don't do this if your index contains any unsaved state that you might want to restore later!) your current index. Normal operation is just

你可以从对象数据库中读取一个 "树" 文件，并用它来建立你当前的索引（如果你当前的索引中包含你希望保存但又未保存的内容的时候，就别这样做了）。标准的做法就是

```
$ git read-tree <sha1 of tree>
```

and your index file will now be equivalent to the tree that you saved earlier. However, that is only your index file: your working directory contents have not been modified.

这样你的索引文件就和你早前所保存的树等价起来了。不过那只是你的索引文件：你的工作树的内容仍然不被改变。

index -> working directory 索引 -> 工作目录

You update your working directory from the index by "checking out" files. This is not a very common operation, since normally you'd just keep your files updated, and rather than write to your working directory, you'd tell the index files about the changes in your working directory (i.e. git-update-index).

从索引中刷新你的工作目录是通过 “提取出” 文件的方法。这是非常不常见的操作，正常来说，你当前所保持的就是文件的最新状态的情况多过你把你曾经告诉过索引的文件更动（譬如，git-update-index），再从索引中写回到你的工作目录。

However, if you decide to jump to a new version, or check out somebody else's version, or just restore a previous tree, you'd populate your index file with read-tree, and then you need to check out the result with

无论如何，如果你打算跳过一个新的版本，或者是提取某人的版本，或者是恢复之前的某个树，你就需要用 read-tree 来生成索引文件，并且提取出结果

```
$ git checkout-index filename
```

or, if you want to check out all of the index, use -a.

或者，如果你要提取所有的索引的话，用 -a 选项。

NOTE! git-checkout-index normally refuses to overwrite old files, so if you have an old version of the tree already checked out, you will need to use the "-f" flag (before the "-a" flag or the filename) to force the checkout.

注意！ git-checkout 默认地拒绝覆盖旧的文件，如果你的工作目录中有一个就的树的版本，你需要用 -f 参数（在参数 -a 或者是要提取的文件名前面）强制提取。

Finally, there are a few odds and ends which are not purely moving from one representation to the other:

最后，还存在一些并非单纯是将东西从一个迁移到另外一个的情况：

Tying it all together 全盘了解

To commit a tree you have instantiated with "git write-tree", you'd create a "commit" object that refers to that tree and the history behind it—most notably the "parent" commits that preceded it in history.

你用 "git write-tree" 提交一个树对象的实例的同时，你应该也创建了一个 “交付” 对象，它包含那个树对象的引用，以及这个交付对象本身在历史中最接近的父交付对象的引用，以此来保持历史的延续。

Normally a "commit" has one parent: the previous state of the tree before a certain change was made. However, sometimes it can have two or more parent commits, in which case we call it a "merge", due to the fact that such a commit brings together ("merges") two or more previous states represented by other commits.

正常来说，一个 “交付” 会有一个父辈：在本次变动产生之前的树的状态。不过，某些时候会有两个父辈，这种情况称之为 “合并”，就原理来说，那样的交付对象整合（合并）了两个或者是两个以上的描述之前的历史状态的交付对象。

In other words, while a "tree" represents a particular directory state of a working directory, a "commit" represents that state in "time", and explains how we got there.

换言之，如果一个 “树” 是描述工作目录的具体状态的话，一个 “交付” 描述的就是 “时间” 状态，它说明了我们是怎么到达这里的。

You create a commit object by giving it the tree that describes the state at the time of the commit, and a list of parents:

你要创建一个交付对象需要指明它所引用的树，这个树描述了这个交付所处的时间的目录状态，并且列明一个父辈列表：

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

and then giving the reason for the commit on stdin (either through redirection from a pipe or file, or by just typing it at the tty).

并且我们可以用 stdin 标准输入的方式向交付命令提供参数依据（可以通过管道或者是文件，或者是直接在命令行终端输入）。

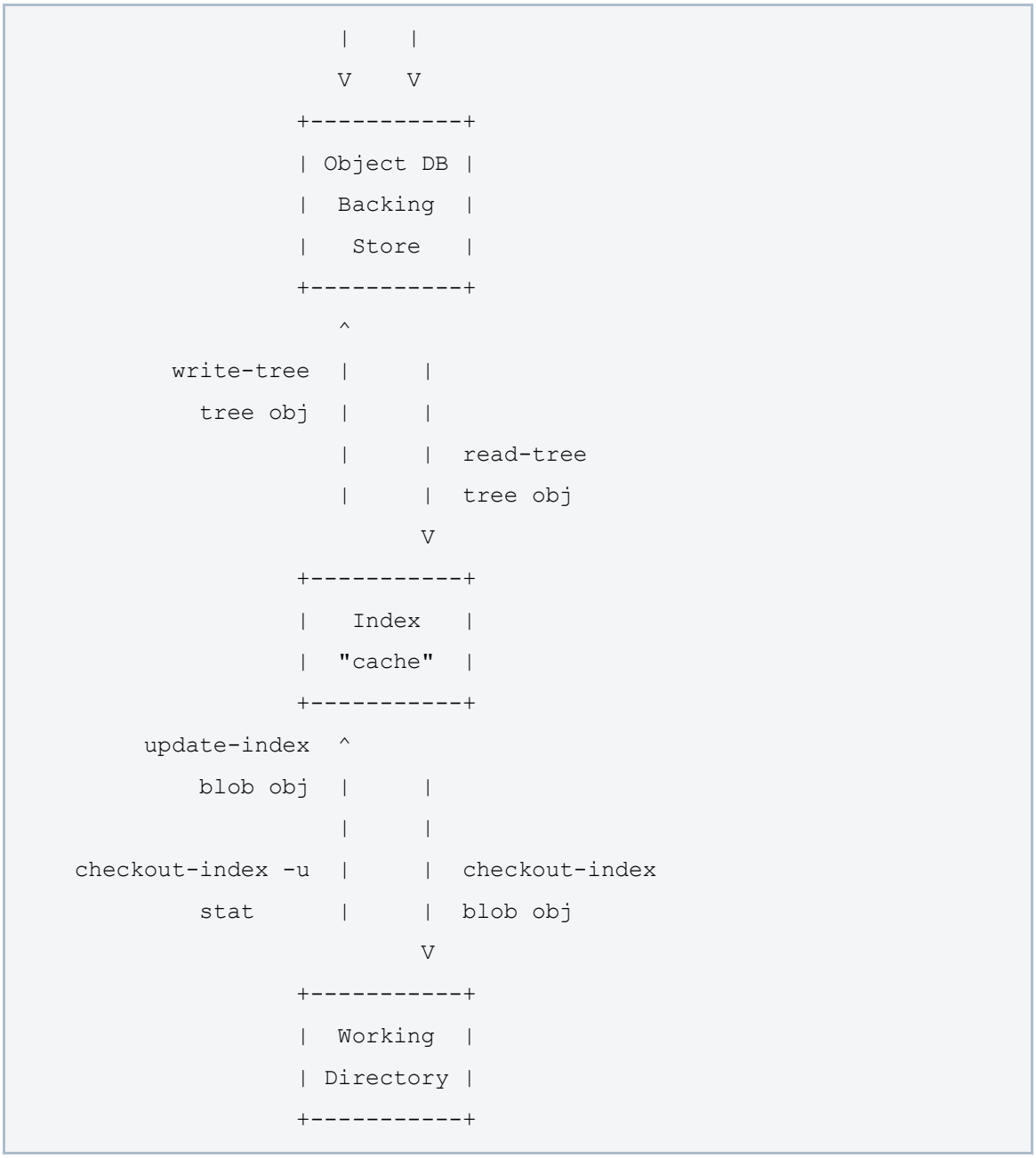
git-commit-tree will return the name of the object that represents that commit, and you should save it away for later use. Normally, you'd commit a new HEAD state, and while git doesn't care where you save the note about that state, in practice we tend to just write the result to the file pointed at by .git/HEAD, so that we can always see what the last committed state was.

git-commit-tree 将返回一个描述该交付的对象名，并且你应该记下它以备以后用到（**译注：**底层命令 git commit-tree 并不会像 git commit 那样将这个返回值写入 .git/HEAD）。正常地，你提交的是一个新的 HEAD 状态，其实 git 并不在乎你在哪里保存这个状态的记录，我们之所以关心将这个返回的结果写入到 .git/HEAD，是因为我们总是可以通过它来查询项目的最后状态。

Here is an ASCII art by Jon Loeliger that illustrates how various pieces fit together.

这里有一个 Jon Loeliger 用字符画的图，将一系列的操作都准确地描述了。

```
                       commit-tree
                        commit obj
                          +----+
                          |    |
```



Examining the data 检验数据

You can examine the data represented in the object database and the index with various helper tools. For every object, you can use git-cat-file(1) to examine details about the object:

你可以用一系列的辅助工具来检验对象数据库和索引。对于所有的对象，你可以用 git-cat-file(1) 检验对象的细节：

```
$ git cat-file -t <objectname>
```

shows the type of the object, and once you have the type (which is usually implicit in where you find the object), you can use

显示对象的类型，你可以用（对象类型通常是隐性的）：

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

to show its contents. NOTE! Trees have binary content, and as a result there is a special helper for showing that content, called git-ls-tree, which turns the binary content into a more easily readable form.

对于显示内容，注意！树对象有二进制的内容的，有专门的辅助工具来显示这些内容，称做 git-ls-tree，它将二进制内容转换为更加可读的形式。

It's especially instructive to look at "commit" objects, since those tend to be small and fairly self-explanatory. In particular, if you follow the convention of having the top commit name in .git/HEAD, you can do

“交付” 对象的结构独特，首先它很少，而且很明了。事实上，如果你遵守将顶端交付对象名写入 .git/HEAD 中的约定，你就可以

```
$ git cat-file commit HEAD
```

to see what the top commit was.

这样就可以看到顶端交付是什么样子的了。

Merging multiple trees 合并多个树

Git helps you do a three-way merge, which you can expand to n-way by repeating the merge procedure arbitrary times until you finally "commit" the state. The normal

situation is that you'd only do one three-way merge (two parents), and commit it, but if you like to, you can do multiple parents in one go.

Git 帮助你完成三路合并，其实你可以展开 N-路 合并进行任意的合并，最终得到你所需要提交的状态。正常的状态下，你只需要进行一个 三-路 合并（两个父辈）并提交结果，不过你愿意的话，你可以一次过进行多父辈的合并。

To do a three-way merge, you need the two sets of "commit" objects that you want to merge, use those to find the closest common parent (a third "commit" object), and then use those commit objects to find the state of the directory ("tree" object) at these points.

三路合并，你需要形成两个你希望合并的交付的对象集，并用他们来查找最接近的共同父交付（第三个“交付”对象），并用这些交付去查找它们所处的目录状态（“树”对象）。

To get the "base" for the merge, you first look up the common parent of two commits with

取得合并“基点”，你首先需要查找需要合并的两个交付的共同的父辈

```
$ git merge-base <commit1> <commit2>
```

which will return you the commit they are both based on. You should now look up the "tree" objects of those commits, which you can easily do with (for example)

命令返回他们的共同基点。现在你应该他们来查找他们所引用的“树”对象了，这个很容易（例如）

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

since the tree object information is always the first line in a commit object.

一直以来，树对象的信息总是在交付对象的第一行。

Once you know the three trees you are going to merge (the one "original" tree, aka the common tree, and the two "result" trees, aka the branches you want to merge), you do a "merge" read into the index. This will complain if it has to throw away your old index

contents, so you should make sure that you've committed those—in fact you would normally always do a merge against your last commit (which should thus match what you have in your current index anyway).

一旦你知道了三个要合并的树（一个“起源”树，表示共同的起点树，以及两个“结果”树，表示你需要合并的两个分支），将他们读入索引。这会带来一些麻烦，因为这样会抛弃掉你的旧的索引内容，所以你应该确保你旧的索引状态已经被提交过一事实上，正常情况下，你会总是对最后的交付进行合并（因而，这个交付应该总是匹配你当前的索引中的内容的）。

To do the merge, do

开始合并了，执行

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

which will do all trivial merge operations for you directly in the index file, and you can just write the result out with git write-tree.

它为你直接在索引文件中执行所有无干涉合并，并且你仅需要将合并的结果用 git write-tree 命令产生一个树对象。

Merging multiple trees, continued 合并多个树，续完

Sadly, many merges aren't trivial. If there are files that have been added, moved or removed, or if both branches have modified the same file, you will be left with an index tree that contains "merge entries" in it. Such an index tree can NOT be written out to a tree object, and you will have to resolve any such merge clashes using other tools before you can write out the result.

不幸的是许多的合并都不是互不干涉的。如果存在文件被加入，移动或者是删除，或者是两个分之都修改过同一个文件的话，你会在索引中留下包含 “合并条目” 的树。这是一个不能被写成树对象的索引树，在你将结果写为树对象之前，你必要用其他的工具解决合并冲突。

You can examine such index state with git ls-files —unmerged command. An example:

你可以用 git ls-files --unmerged 命令来检验索引状态，例如：

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

Each line of the git ls-files —unmerged output begins with the blob mode bits, blob SHA1, stage number, and the filename. The stage number is git's way to say which tree it came from: stage 1 corresponds to $orig tree, stage 2 HEAD tree, and stage3 $target tree.

每一行的 git ls-files --unmerged 输出的开始是片对象的模式，接着就是片的 SHA1，阶段编号，以及文件名。阶段编号是 git 用来说明树来源的：stage1 表示 源头树，stage2 表示 HEAD 树，以及 stage3 表示目标树。

Earlier we said that trivial merges are done inside git-read-tree -m. For example, if the file did not change from $orig to HEAD nor $target, or if the file changed from $orig to HEAD and $orig to $target the same way, obviously the final outcome is what is in HEAD. What the above example shows is that file hello.c was changed from $orig to HEAD and $orig to $target in a different way. You could resolve this by running your favorite 3-way merge program, e.g. diff3, merge, or git's own merge-file, on the blob objects from these three stages yourself, like this:

之前我们讲过，无干涉的合并会在 git-read-tree -m 命令内部完成。例如，如果文件从源头到 HEAD 或者是目标这两路发展过程中，其中一路没有修改过文件，也或者是文件从源头到 HEAD 以及从源头到目标的发展过程中，修改都来自同一路，显然，最终的结果都是 HEAD 中的东西。上面的例子中，显示文件 hello.c 从源头到 HEAD 以及从源头到目标的变化来自不同的路径。你可以运行你喜欢的 3-路合并工具来解决冲突，例如 diff3，merge，或者是 git 自带的 merge-file，用这些工具来处理这三个阶段文件，好像这样：

- **译注：** 我就喜欢用 vi 直接打开并修改 hello.c，并保存提交就可以了。

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

```
$ git cat-file blob cc44c73... >hello.c~3
$ git merge-file hello.c~2 hello.c~1 hello.c~3
```

This would leave the merge result in hello.c~2 file, along with conflict markers if there are conflicts. After verifying the merge result makes sense, you can tell git what the final merge result for this file is by:

这样合并的结果会出现在 hello.c~2 文件中，如果有合并冲突的话，文件中会有冲突标记。处理好冲突之后，告诉 git 最终的合并结果：

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

When a path is in the "unmerged" state, running git-update-index for that path tells git to mark the path resolved.

如果出现路径的“未合并”状态，对路径运行 git-update-index 告诉 git 路径合并的问题解决了。

The above is the description of a git merge at the lowest level, to help you understand what conceptually happens under the hood. In practice, nobody, not even git itself, runs git-cat-file three times for this. There is a git-merge-index program that extracts the stages to temporary files and calls a "merge" script on it:

上面的陈述是对 git 合并的底层介绍，帮助你从概念上理解合并的表面之下发生了什么，事实上，没有人，甚至是 git 自身都不会运行 git-cat-files 三次来处理这样的情况。git-merge-index 程序提取各个阶段到临时文件，称为“合并”脚本。

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

and that is what higher level git-merge -s resolve is implemented with.

并有一个高层命令 git-merge -s resolve 来实现。

Chapter 10. Hacking git 第十章. git 的开发

This chapter covers internal details of the git implementation which probably only git developers need to understand.

本章涵盖 git 的实现的内部细节，它只适合 git 的开发者学习理解。

Object storage format 对象的存储格式

All objects have a statically determined "type" which identifies the format of the object (i.e. how it is used, and how it can refer to other objects). There are currently four different object types: "blob", "tree", "commit", and "tag".

所有的对象都有一个固定的“类型”，它用于识别对象的格式（譬如，他如何使用，以及它如何引用到其他的类型）。现在有四种不同的对象类型：“片”，“树”，“交付”，以及 “标签”。

Regardless of object type, all objects share the following characteristics: they are all deflated with zlib, and have a header that not only specifies their type, but also provides size information about the data in the object. It's worth noting that the SHA1 hash that is used to name the object is the hash of the original data plus this header, so `sha1sum` *file* does not match the object name for *file*. (Historical note: in the dawn of the age of git the hash was the sha1 of the compressed object.)

与对象的类型无关，所有对象都具备以下的这些性质：他们都使用 zlib 库的 deflated 压缩算法，并且都带一个头，它不但指明了对象的类型，同时还提供了对象中的数据大小信息。这对于计算在原始数据中加入了对象头的对象的 SHA1 哈希特征值来说是很有价值的，这个哈希特征值用来命名该对象。所以，sha1sum *file* 这样的命令是不能匹配到一个 *file* 文件的对象名的。（历史提示：在 git 项目的初期是使用压缩之后的对象的哈希特征值的。）

As a result, the general consistency of an object can always be tested independently of the contents or the type of the object: all objects can be validated by verifying that (a) their

hashes match the content of the file and (b) the object successfully inflates to a stream of bytes that forms a sequence of <ascii type without space> + <space> + <ascii decimal size> + <byte\0> + <binary object data>.

这样处理就可以得到总体的一致性的效果，即对象的测试与对象的内容或者是对象的类型无关：所有的对象这样来验证他们的有效性：（a）他们的哈希特征值一定匹配他们的文件的内容，以及（b）对象可以有效地以一个字节串的队列形式展开<无空格的 ascii 编码类型标记> + <空格> + <ascii 编码的十进制大小标记> + <字节\0> + <二进制对象数据>。

The structured objects can further have their structure and connectivity to other objects verified. This is generally done with the git-fsck program, which generates a full dependency graph of all objects, and verifies their internal consistency (in addition to just verifying their superficial consistency through the hash).

构建好的对象会拥有自己的结构，以及对已经验证过的其他对象的连接。这可以用 git-fsck 程序来检验，它检验所有对象的关联视图，以及验证他们的内部一致性（验证他们的表层一致性只需要通过哈希算法）。

A birds-eye view of Git's source code 鸟瞰 git 源代码

It is not always easy for new developers to find their way through Git's source code. This section gives you a little guidance to show where to start.

对于一个新手来说，找到他们自己的理解 git 源代码的途径不是一件容易的事情。这一节给你一个从那里开始的小小指导。

A good place to start is with the contents of the initial commit, with:

初始的交付中的内容是一个好的起点：

```
$ git checkout e83c5163
```

The initial revision lays the foundation for almost everything git has today, but is small enough to read in one sitting.

初始的修订版本已经为今天的 git 的打下了基础，而且它对于阅读来说足够地小。

Note that terminology has changed since that revision. For example, the README in that revision uses the word "changeset" to describe what we now call a commit.

注意版本中的术语的变迁，譬如，在 README 中，我们现在叫的“交付”（commit）那时叫“变更集”（changeset）。

Also, we do not call it "cache" any more, but rather "index"; however, the file is still called cache.h. Remark: Not much reason to change it now, especially since there is no good single name for it anyway, because it is basically the header file which is included by all of Git's C sources.

还有，我们现在已经不再将“索引”（index）叫做“缓存”（cache）了，不过文件明还是叫 cache.h 。备注：现在还没有足够的理由改掉这个名字，尤其是在没有一个好的名字来替代它，因为它是基础性的头文件，被包含在所有 Git 的 C 源代码中。

If you grasp the ideas in that initial commit, you should check out a more recent version and skim cache.h, object.h and commit.h.

当你已经领悟了初始交付中的内容，你应该提取更新的版本来阅读，并且跳过 cache.h, object.h 和 commit.h 。

In the early days, Git (in the tradition of UNIX) was a bunch of programs which were extremely simple, and which you used in scripts, piping the output of one into another. This turned out to be good for initial development, since it was easier to test new things. However, recently many of these parts have become builtins, and some of the core has been "libified", i.e. put into libgit.a for performance, portability reasons, and to avoid code duplication.

在早期，Git（作为传统的 Unix 程序）是一个极其简单的程序集，你需要用脚本将一个工具的输出通过管道传输给另外一个工具。作为启动一个项目的初期开发来说是个好的方式，它便于对新的事物进行测试。不过，目前许多部分已经成为内建方式，并且某些核心部分已经“库化”了，譬如基于性能，移植性的理由，以及避免代码的重复，这些核心部分放到了 libgit.a 库中。

By now, you know what the index is (and find the corresponding data structures in cache.h), and that there are just a couple of object types (blobs, trees, commits and tags)

which inherit their common structure from struct object, which is their first member (and thus, you can cast e.g. (struct object *)commit to achieve the same as &commit->object, i.e. get at the object name and flags).

现在，你知道什么是索引了（并且你可以在 cache.h 中找到它相对的数据结构），还有一系列的对象类型（片，树，交付，标签），他们都继承自通用的对象结构，这个通用结构都是各个数据类型的结构的第一个成员变量（因此，你可以进行类似这样的指针变换 (struct object *)commit，等同由于 &commit->object, 也就可以得到对象的名称和标志）。

Now is a good point to take a break to let this information sink in.

现在是深入理解这些信息的突破点了。

Next step: get familiar with the object naming. Read the section called “Naming commits”. There are quite a few ways to name an object (and not only revisions!). All of these are handled in sha1_name.c. Just have a quick look at the function get_sha1(). A lot of the special handling is done by functions like get_sha1_basic() or the likes.

下一步：熟悉对象名称。参看“命名交付”那一节。有若干的途径来命名一个对象（并非只是一个修订版对象）。所有这些处理在 sha1_name.c 文件中实现。首先快速阅读一下函数 get_sha1()。很多特殊的处理由类似 get_sha1_basice() 之类的函数来处理。

This is just to get you into the groove for the most libified part of Git: the revision walker.

这个只是带你进入大量的 Git 库化部分的切入点：修订版步行者。

Basically, the initial version of git-log was a shell script:

基本上，初期的 git-log 命令是一个 shell 脚本：

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

What does this mean?

这是什么意思呢？

git-rev-list is the original version of the revision walker, which always printed a list of revisions to stdout. It is still functional, and needs to, since most new Git programs start out as scripts using git-rev-list.

git-rev-list 是修订版步行者的起源版本，它总是将修订版的列表打印到标准输出。它仍然是有用和有需要的，直至最新的 Git 程序中仍然是一个以 git-rev-list 为开始的脚本。

git-rev-parse is not as important any more; it was only used to filter out options that were relevant for the different plumbing commands that were called by the script.

git-rev-parse 就并不是那么重要了；它只是用作 Git 脚本调用不同的关联命令的选项过滤器。

Most of what git-rev-list did is contained in revision.c and revision.h. It wraps the options in a struct named rev_info, which controls how and what revisions are walked, and more.

大部分 git-rev-list 所做的事情是在 revision.c 和 revision.h 中实现的。那里封装了结构 rev_info，用于控制如何进行版本步行和处理等等。

The original job of git-rev-parse is now taken by the function setup_revisions(), which parses the revisions and the common command line options for the revision walker. This information is stored in the struct rev_info for later consumption. You can do your own command line option parsing after calling setup_revisions(). After that, you have to call prepare_revision_walk() for initialization, and then you can get the commits one by one with the function get_revision().

git-rev-parse 的本份的任务是调用函数 setup_revision()，它解析修订版并为修订版步行解析命令行选项。解析出来的信息保存在结构 rev_info 中，以备将来调用。调用函数 setup_revision() 对命令行选项进行过解析之后，你将来将可以使用你自己的命令行选项了。之后，你必须调用 prepare_revision_walk() 进行初始化，之后，你就可以通过 get_revision() 一个接一个地取得各个交付（commit）。

If you are interested in more details of the revision walking process, just have a look at the first implementation of cmd_log(); call git show v1.3.0~155^2~4 and scroll down to that function (note that you no longer need to call setup_pager() directly).

如果你对修订版步行的更多细节感兴趣，只要看看 cmd_log() 的第一个实现；在版本库中调用 git show v1.3.0~155^2~4 并下滚到该函数（注意你已经不必再直至调用 setup_pager() 函数）。

Nowadays, git-log is a builtin, which means that it is contained in the command git. The source side of a builtin is

现在，git-log 已经是一个内建命令，这意味着它包含做 git 命令中。从源代码的角度说，内建就是

- a function called cmd_<bla>, typically defined in builtin-<bla>.c, and declared in builtin.h,
- an entry in the commands[] array in git.c, and
- an entry in BUILTIN_OBJECTS in the Makefile.
- 函数名为 cmd_<bla>，典型地它在 builtin-<bla>.c 中实现，并在 builtin.h 中声明，
- 它将成为在 git.c 中的 commands[] 数组的一个条目，并且
- 成为 Makefile 文件中的 BUILTIN_OBJECTS 的一个条目。

Sometimes, more than one builtin is contained in one source file. For example, cmd_whatchanged() and cmd_log() both reside in builtin-log.c, since they share quite a bit of code. In that case, the commands which are not named like the .c file in which they live have to be listed in BUILT_INS in the Makefile.

某些时候内建命令被包装在一个源文件中。例如，com_whatchanged() 和 com_log() 两个都位于 builtin-log.c 中，如此他们就可以共享二进制代码。这样命令就不能像它是位于 .c 文件中那样来命名了，它必须在 Makefile 文件中的 BUILTIN_INS 中列明。

git-log looks more complicated in C than it does in the original script, but that allows for a much greater flexibility and performance.

git-log 在 C 中的实现，看起来要比原来用 shell 脚本实现的时候复杂多了，不过这样也得到了更好的性能和更大的伸缩性。

Here again it is a good point to take a pause.

这里是做个小的暂定的好地方。

Lesson three is: study the code. Really, it is the best way to learn about the organization of Git (after you know the basic concepts).

第三课是： 研究源代码。的确，这是学习 Git 的项目组织的最佳途径（当你知道了基本的概念之后）。

So, think about something which you are interested in, say, "how can I access a blob just knowing the object name of it?". The first step is to find a Git command with which you can do it. In this example, it is either git-show or git-cat-file.

于是，想象一下某些你感兴趣的事情，比方说，“我是如何通过对象名来访问一个片对象的呢？”。首先查找有什么 Git 命令可以做到这个。在这个例子中，git-show 或者是 git-cat-file 任何一个都可以。

For the sake of clarity, let's stay with git-cat-file, because it

为更为清晰的缘故，让我们先谈谈 git-cat-file, 因为它

- is plumbing, and
- was around even in the initial commit (it literally went only through some 20 revisions as cat-file.c, was renamed to builtin-cat-file.c when made a builtin, and then saw less than 10 versions).
- 它是一截水管（**译注：**极客们对 git 内核命令的戏称。），并且
- 它涉及到交付对象的初始化（cat-file.c 文件大约经历了 20 多个修订之后，当它成为内建命令之后，被重命名为 builtin-cat-file.c，并且我们至少可以看到又经历了 10 个版本的修订）。

So, look into builtin-cat-file.c, search for cmd_cat_file() and look what it does.

看看 builtin-cat-file.c 的内部，找到 cmd_cat_file() 函数并且看看它做了什么。

```
        git_config(git_default_config);
        if (argc != 3)
                usage("git-cat-file [-t|-s|-e|-p|<type>] <sha1>");
        if (get_sha1(argv[2], sha1))
                die("Not a valid object name %s", argv[2]);
```

Let's skip over the obvious details; the only really interesting part here is the call to get_sha1(). It tries to interpret argv[2] as an object name, and if it refers to an object which is present in the current repository, it writes the resulting SHA-1 into the variable sha1.

让我们跳过一些明显的细节，只关注它调用 get_sha1() 函数的地方。它尝试解析 argv[2] 作为对象名，当它可以发现它的确指向当前版本库中的一个对象时，它将 SHA-1 结果写入变量 sha1。

Two things are interesting here:

有两件事情值得注意：

- get_sha1() returns 0 on success. This might surprise some new Git hackers, but there is a long tradition in UNIX to return different negative numbers in case of different errors—and 0 on success.
- the variable sha1 in the function signature of get_sha1() is unsigned char *, but is actually expected to be a pointer to unsigned char[20]. This variable will contain the 160-bit SHA-1 of the given commit. Note that whenever a SHA-1 is passed as unsigned char *, it is the binary representation, as opposed to the ASCII representation in hex characters, which is passed as char *.

- get_sha1() 成功时返回 0。这对于一些新的 Git 黑客来说可能会感到奇怪，不过这个是 Unix 系统的长期传统，不同的错误返回不同的负数值，而成功则返回 0 。
- 函数 get_sha1() 的参数变量 sha1 是无符号字符指针 (unsigned char *)，不过实际上被当成是指向一个无符号字符数组 (unsigned char[20]) 的指针看待。这个变量包含一个 160 位二进制的 SHA-1 给指定的交付使用。注意，任何时候它作为二进制 SHA-1 的表述时，都应作为无符号字符指针 (unsigned char *) 来传递，当作为十六进制的 ASCII 编码表述时，它就要作为字符指针传递 (char *)。

You will see both of these things throughout the code.

上述的两件事情在源代码中处处可见。

Now, for the meat:

现在轮到分析我们的具体问题了：

```
        case 0:
                buf = read_object_with_reference(sha1, argv[1], &size,
NULL);
```

This is how you read a blob (actually, not only a blob, but any type of object). To know how the function read_object_with_reference() actually works, find the source code for it (something like git grep read_object_with | grep ":[a-z]" in the git repository), and read the source.

这里就是如何读取一个片（准确地说，不但是片，而是所有的对象类型）。要知道函数 read_object_with_referencee() 是如何工作的，那就查找一下这个函数（例如在 git 版本库中执行命令 git grep read_object_with | grep ":[a-z]"），并阅读分析。

- **译注：** 浏览和分析代码，还是 vim + ctags + cscope 这样的工具组合方便得多，不过这已经超出了本文的范围了。

To find out how the result can be used, just read on in cmd_cat_file():

查找返回的结果如何使用，就看看函数 cmd_cat_file()：

```
        write_or_die(1, buf, size);
```

Sometimes, you do not know where to look for a feature. In many such cases, it helps to search through the output of git log, and then git-show the corresponding commit.

有些时候，你并不知道在哪里找到一个功能特性。大多数情况下， git log 的输出可以帮助你发现你需要的东西，接着 git-show 也能找到响应的交付。

Example: If you know that there was some test case for git-bundle, but do not remember where it was (yes, you could git grep bundle t/, but that does not illustrate the point!):

据个例子： 如果你知道某些针对 git-bundle 测试工作，不过不记得它是在什么地方了（对，你可以用 git grep bundle t/ ，不过那样似乎不得要领！）：

```
$ git log --no-merges t/
```

In the pager (less), just search for "bundle", go a few lines back, and see that it is in commit 18449ab0… Now just copy this object name, and paste it into the command line

在输出的页面中（输出被管道定向到 less 工具），查找 "bundle"，回滚几行，就能看到关联交付对象名 18449ab0...，并将它复制到命令行

```
$ git show 18449ab0
```

Voila.

瞧。

Another example: Find out what to do in order to make some script a builtin:

另外一个例子： 查找从脚本方式到内建命令的方式的开发过程中做过什么：

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

You see, Git is actually the best tool to find out about the source of Git itself!

你可以看到，Git 自身事实上就是非常好的源代码搜索工具。

Chapter 11. GIT Glossary 第十一章. GIT 字典

alternate object database 轮替对象数据库

- Via the alternates mechanism, a repository can inherit part of its object database from another object database, which is called "alternate".
- 通过轮替机制，一个版本库可以将自己的某些部分作为另外的版本库的传承，所以叫“轮替”。

bare repository 裸库

- A bare repository is normally an appropriately named directory with a .git suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the git administrative and control files that would normally be present in the hidden .git sub-directory are directly present in the repository.git directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.
- 一个裸库就是正常版本库的那个叫 .git 的目录，并且不会提取出任何在版本库控制下的文件。所有 git 管理和控制的文件正常情况下是在隐藏的 .git 子目录中呈现的，对于裸库来说，这些被直接呈现在 repository.git 目录中（**译注：** 譬如叫 u-boot.git，linux-2.6.git 等），并且不会呈现和提取出任何的项目文件。通常来说，项目发布者会以裸库的形式发布项目。

blob object 片对象

- Untyped object, e.g. the contents of a file.
- 无类型的对象，只是一个文件的内容。

branch 分支

- A "branch" is an active line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.
- 一个“分支”就是一条研发的线路。在分支上的最新交付对象就是作为分支的顶端。分支的顶端也是分支头的引用，它将随项目的研发不断推进。一个单独的版本库可以跟中任意多的分支数目，不过你的工作树就只能跟它们其中一个关联对应了（即 “当前” 或者是 “提取” 的分支），HEAD 总是指向这个当前分支。

cache 缓存

- Obsolete for: index.
- 过时的概念了，参考： 索引

chain 链

- A list of objects, where each object in the list contains a reference to its successor (for example, the successor of a commit could be one of its parents).
- 一个对象链表，其中的每个对象都包含一个指向其被承继者的引用（例如，一个交付对象的被承继者，就是它的其中一个父交付）。

changeset 变更集

- BitKeeper/cvsps speak for "commit". Since git does not store changes, but states, it really does not make sense to use the term "changesets" with git.
- 在 BitKeeper/cvsps 中叫 “交付”。然而 git 是不保存项目的变迁的，它保存的是项目的状态，对于 git 来说，就不必为 “变更集” 这个概念费神了。

checkout 提取

- The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.
- 以对象数据库中的树对象和片对象来更新工作目录中的所有东西，如果工作树被指向了一个新的分支的话，同时刷新索引和 HEAD。

cherry-picking 樱桃摘取

- In SCM jargon, "cherry pick" means to choose a subset of changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In GIT, this is performed by the "git cherry-pick" command to extract the change introduced by an existing commit and to record it based on the tip of the current branch as a new commit.

- 在源代码管理的行话中，“樱桃摘取”意味着在一个变更（交付就是一个典型）序列中选择一个子集并将它保存为一个不同的代码起点的变更集。在 git 中，执行这个操作的是 git cherry-pick 命令。它展开从交付对象中导入的变更，并纪录下来，放置在当前的分支的顶端，成为一个新的交付。

clean 干净的

- A working tree is clean, if it corresponds to the revision referenced by the current head. Also see "dirty".
- 如果工作树完全对应于当前的分支头所引用的状态，我们称它是干净的。对应地，参看 “污浊”。

commit 交付

- As a noun: A single point in the git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.
- As a verb: The action of storing a new snapshot of the project's state in the git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.
- 作为名词： 是指 git 中的一个历史瞬间；整个项目的发展历史表现为一个相互关联的交付集。在 git 中，“交付”这个词所表达的意思，在其他的版本控制系统上用“修订”或者是“版本”。它还经常被用作交付对象的缩略写法。
- 作为动词： 是指在 git 历史中保存项目状态快照的动作，它描述索引的当前状态并将 HEAD 推进至当前新的交付。

commit object 交付对象

- An object which contains the information about a particular revision, such as parents, committer, author, date and the tree object which corresponds to the top directory of the stored revision.

- 一个对象，它包含了具体的修订信息，譬如它的父辈，提交人，作者，日期和树对象，该树对象保存了对应的项目目录的修订。

core git 核心 git

- Fundamental data structures and utilities of git. Exposes only limited source code management tools.
- Git 的基本数据结构和工具。只表现为最基本的源代码管理工具。

DAG

- Directed acyclic graph. The commit objects form a directed acyclic graph, because they have parents (directed), and the graph of commit objects is acyclic (there is no chain which begins and ends with the same object).
- 单向非环路视图。交付对象是按一个单向非环路是图的形式来组织的，因为他们有父辈（方向），以及非环路视图（不存在开始和结尾都是同一个对象的情况）。

dangling object

- An unreachable object which is not reachable even from other unreachable objects; a dangling object has no references to it from any reference or object in the repository.

detached HEAD 游离 HEAD

- Normally the HEAD stores the name of a branch. However, git also allows you to check out an arbitrary commit that isn't necessarily the tip of any particular branch. In this case HEAD is said to be "detached".
- 正常情况下，HEAD 保存的是一个分支，不过 git 还容许你提取任意的交付，它不必是一个具体的分支的顶端。这种情况下，我们成 HEAD 是“游离的”。

dircache 目录缓存

- You are waaaaay behind. See index.

- 你有点晕了吧，看“索引”吧。

directory 目录

- The list you get with "ls" :)
- 那当然就是你运行 "ls" 命令看到的东西了 :)

dirty 污浊的

- A working tree is said to be "dirty" if it contains modifications which have not been committed to the current branch.
- 当一个工作树的内容已经被修改过，但是仍然没有被提交到当前分支，则成为“污浊的”。

ent

- Favorite synonym to "tree-ish" by some total geeks. See http://en.wikipedia.org/wiki/Ent_(Middle-earth) for an in-depth explanation. Avoid this term, not to confuse people.
- 对于某部分极客来说，它是 "tree-ish" 的同义词。Ent 这个词是怎么来的，看维基百科吧 http://en.wikipedia.org/wiki/Ent_(Middle-earth) ，这里就不解释了，免得大家犯糊涂。

evil merge 恶劣合并

- An evil merge is a merge that introduces changes that do not appear in any parent.
- 恶劣合并就是合并了一个由没有父辈的东西所导入的变更。

fast forward 快进

- A fast-forward is a special type of merge where you have a revision and you are "merging" another branch's changes that happen to be a descendant of what you have. In such these cases, you do not make a new merge commit but instead just update to his revision. This will happen frequently on a tracking branch of a remote repository.

- 快进是一个特殊类型的合并，你要合并的来自其他分支的东西是你的合并的目标分支的后裔。这种情况下，你不用生成一个新的合并交付，取而代之的是将你的目标分支更新到他的修订上。这在一个跟踪远程版本库的分支上是经常发生的。

fetch 抓取

- Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too. See also git-fetch(1).
- 抓取一个分支，意味着取得一个远程版本库的头的引用，并找出本地版本库的对象数据库中有什么对象缺失了，并将缺失的对象抓取过来。参考 git-fetch(1)。

file system 文件系统

- Linus Torvalds originally designed git to be a user space file system, i.e. the infrastructure to hold files and directories. That ensured the efficiency and speed of git.
- Linus Torvalds 开始是就将 git 设计成一个用户空间的文件系统，也就是基础架构包含了文件和目录。这保证了 git 的效率和速度。

git archive | git 档案

- Synonym for repository (for arch people).
- 版本库的同义词（对于 arch 用户来说）。 ***译注：*** *Gnu Arch 是自由软件基金会开发的另外一个修订控制系统*

grafts 嫁接

- Grafts enables two otherwise different lines of development to be joined together by recording fake ancestry information for commits. This way you can make git pretend the set of parents a commit has is different from what was recorded when the commit was created. Configured via the .git/info/grafts file.

- 嫁接可以使两个不同的开发路线，通过伪造交付的祖先信息的方式接合到一起。通过这个方法你可以令 git 在一个交付被创建的时候，伪造一份与其真正来源不同的父辈集。它在 .git/info/grafts 文件中配置。

hash 哈希

- In git's context, synonym to object name.
- 在 git 的背景下，就是对象名的同义词。

head 头

- A named reference to the commit at the tip of a branch. Heads are stored in $GIT_DIR/refs/heads/, except when using packed refs. (See git-pack-refs(1).)
- 一个分支顶端交付的引用称谓。头都保存在 $GIT_DIR/refs/heads 中，当使用打包过的引用时除外（参看 git-pack-refs(1).）。

HEAD

- The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it may reference an arbitrary commit.
- 当前分支。更详细点： 你的工作树正常情况下是产生自 HEAD 所引用的树对象所描述的状态。HEAD 指向你的版本库中的其中一个头，除非你正在使用一个游离的 HEAD，这是一种指向一个任意的交付的情况。

head ref 头引用

- A synonym for head.
- 头的同义词。

hook 钩子

- During the normal execution of several git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the $GIT_DIR/hooks/ directory, and are enabled by simply removing the .sample suffix from the filename. In earlier versions of git you had to make them executable.
- 在一系列的 git 命令的正常执行过程中，会产生一个对外部可选的脚本调用，容许开发加入附加的功能和检测。典型地，钩子容许命令进行前期验证和隐性地中止命令的执行，以及在命令执行完成之后进行后期提示。钩子脚本可以在 $GIT_DIR/hooks/ 目录下找到，并可以简单地通过将文件名的 .sample 后续名去掉来使能它。在早期的 git 版本中，你还必要给他们加上可执行属性。

index 索引

- A collection of files with stat information, whose contents are stored as objects. The index is a stored version of your working tree. Truth be told, it can also contain a second, and even a third version of a working tree, which are used when merging.
- 一个文件状态信息的集合，它的内容会被保存为对象。索引保存了你的工作目录的版本。告诉你一个真相，它是可以包含第二个，甚至是第三个你的工作树的版本的，那是当你进行合并操作的时候。

index entry 索引条目

- The information regarding a particular file, stored in the index. An index entry can be unmerged, if a merge was started, but not yet finished (i.e. if the index contains multiple versions of that file).
- 针对每个具体的文件的信息，它保存在索引中。一个索引条目可以是未合并的，如果合并操作已经开始，但还没有完成时（例如，索引包含文件的多个版本）。

master 主分支

- The default development branch. Whenever you create a git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.
- 默认的开发分支，每当你创建一个 git 版本库的时候，名为 "master" 的分支会同时被创建，并成为活动的分支。大多数情况下，版本库会包含本地的开发分支，但是那纯粹是一种约定，并不是必须这样做的。

merge 合并

- As a verb: To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.
- As a noun: unless it is a fast forward, a successful merge results in the creation of a new commit representing the result of the merge, and having as parents the tips of the merged branches. This commit is referred to as a "merge commit", or sometimes just a "merge".
- 作为动词：就是将其他的分支内容（可能是来自远程版本库的）带入到当期分支。如果将要合并进来的分支来自远程版本库，完成合并的第一步应该是抓取远程分支，然后将抓取的结果合并进当前分支。组合这两个操作的命令叫拉入（pull）。合并的过程是自动的，当两个分支的变更是互补干涉的情况下，并且所有的变更都会被整合到一起。在存在变更冲突的情况下，要完成合并，就需要手工干预合并的过程了。
- 作为名词： 除非合并是一个快进，一个成功的合并将创建一个新的交付对象，该对象描述了合并的结果，并有一个合并分支的顶端交付作为父辈的集合。这个交付归类为“合并交付”，或者有时干脆就叫“合并”。

object 对象

- The unit of storage in git. It is uniquely identified by the SHA1 of its contents. Consequently, an object can not be changed.
- git 中的存储单元。它以 SHA1 （***译注：****安全哈希算法*）唯一地识别它的内容，因此，对象是不可更改的。

object database 对象数据库

- Stores a set of "objects", and an individual object is identified by its object name. The objects usually live in $GIT_DIR/objects/.
- 保存对象们的场所，不同的对象由它们自身的对象名识别。对象通常是保存在 $GIT_DIR/objects/ 目录中。

object identifier 对象标识

- Synonym for object name.
- 对象名的同义词。

object name 对象名

- The unique identifier of an object. The hash of the object's contents using the Secure Hash Algorithm 1 and usually represented by the 40 character hexadecimal encoding of the hash of the object.
- 对象的唯一标识。它是以对象的内容由安全哈希算法 1 计算出来的哈希特征值，通常表达为一串 40 位的十六进制编码的字符串。

object type 对象类型

- One of the identifiers "commit", "tree", "tag" or "blob" describing the type of an object.
- 一个标记，表明对象是属于 “交付”，“树”，“标签”或者是“片”中的那个类型。

octopus 章鱼

- To merge more than two branches. Also denotes an intelligent predator.

- 合并多于两个分支。也表示一个只聪明的食肉动物，呵呵。

origin 源头

- The default upstream repository. Most projects have at least one upstream project which they track. By default origin is used for that purpose. New upstream updates will be fetched into remote tracking branches named origin/name-of-upstream-branch, which you can see using "git branch -r".
- 默认的上游版本库。大部分的项目都有至少一个上游版本库需要跟踪。默认源头就是为这个目的来使用的。新的上游版本库跟踪分支将被抓取进 origin/name-of-upstream-branch 中，它们可以用 "git branch -r" 命令列举出来。

pack 打包

- A set of objects which have been compressed into one file (to save space or to transmit them efficiently).
- 对象集被压缩到一个文件当中（有利于节省磁盘空间和传输效率）。

pack index

- The list of identifiers, and other information, of the objects in a pack, to assist in efficiently accessing the contents of a pack.
- 已经被打包的对象的标识，以及其他信息的链表，辅助高效地在打包文件中访问对象的内容。

parent 父辈

- A commit object contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.
- 一个交付包含一个（可能是空）的链表，里面纪录了该对象在开发过程当中的逻辑前辈，例如，它的父交付。

pickaxe 手镐

- The term pickaxe refers to an option to the diffcore routines that help select changes that add or delete a given text string. With the —pickaxe-all option, it can be used to view the full changeset that introduced or removed, say, a particular line of text. See git-diff(1).
- 手镐递交一个选项给差异比较规程，帮助决定一个指定的文本串是加入或者是删除。结合 --packaxe-all 选项，它可以用于参看全部的变更是引入还是删除，对每个具体的文本行，参看 git-diff(1)。

plumbing 水管

- Cute name for core git.
- git 内核的戏称。

porcelain 瓷器

- Cute name for programs and program suites depending on core git, presenting a high level access to core git. Porcelains expose more of a SCM interface than the plumbing.
- 依赖于 git 内核的程序以及程序包的戏称，表现为规 git 内核的封装和调用。瓷器相对于水管来说，表现出更标准的 SCM 的界面。

pull 拉入

- Pulling a branch means to fetch it and merge it. See also git-pull(1).
- 拉入一个分支，意味着抓取并合并它。参看 git-pull(1)。

push 推出

- Pushing a branch means to get the branch's head ref from a remote repository, find out if it is a direct ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.
- 推出一个分支，意味着获取一个远程版本库的分之头，并查看它是否是本地分之头的祖先，如果是，则将本地分之头的所有可及的，

而远程分支上缺少的对象传输至远程对象数据库，并更新远程分之头。如果远程分之头不是本地分之头的前辈，推入将失败。

reachable 可及性

- All of the ancestors of a given commit are said to be "reachable" from that commit. More generally, one object is reachable from another if we can reach the one from the other by a chain that follows tags to whatever they tag, commits to their parents or trees, and trees to the trees or blobs that they contain.
- 一个给定的交付的所有先辈对于这个交付来说是可及的。广义地，如果一个对象对于另外一个对象来说是可及的，那么我们可以从在一个链条中从一个追索到另外一个，例如，从一个标签，我们可以追索到标签说标记的所有东西，从一个交付可以追索到它的父辈和树，从一个树可以追索到它说包含的其他树或者片。

rebase 重定基点

- To reapply a series of changes from a branch to a different base, and reset the head of that branch to the result.
- 将一系列的变动从一个分之应用到另外一个不同的基点，并将分之头重新放置到应用之后的结果上。

ref 引用

- A 40-byte hex representation of a SHA1 or a name that denotes a particular object. These may be stored in $GIT_DIR/refs/.
- 一个 40 个十六进制字符的哈希特征值，或者是表明具体对象的名称。他一般保存在 $GIT_DIR/refs/ 目录中。

reflog 引用日志

- A reflog shows the local "history" of a ref. In other words, it can tell you what the 3rd last revision in this repository was, and what was the current state in this repository, yesterday 9:14pm. See git-reflog(1) for details.

- 一个引用日志展示了一个引用的本地历史。换言之，它可以告诉你最后的第三次修订是什么，以及昨天 9：14pm 时的版本库的状态等诸如此类的信息。详情参看 git-reflog(1)。

refspec 引用规则

- A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref. They are combined with a colon in the format <src>:<dst>, preceded by an optional plus sign, +. For example: git fetch $URL refs/heads/master:refs/heads/origin means "grab the master branch head from the $URL and store it as my origin branch head". And git push $URL refs/heads/master:refs/heads/to-upstream means "publish my master branch head as to-upstream branch at $URL". See also git-push(1).
- “ 引用规则” 用于在抓取和推出操作中映射远程引用和本地引用的。它们以冒号作为组合格式 <源>:<目标>，还可以在前面加上一个 + 号作为选项。例如：git fetch $URL refs/heads/master:refs/heads/origin 意味着“从远程版本库 $URL 中获取 master 分之头并将其保存为我的本地 origin 的分支头”。以及 git push $URL refs/heads/master:refs/heads/to-upstream 意味着“推出我的 master 分支头作为 $URL 远程版本库中的 to-upstream 分支”。参看 git-push(1)。

repository 版本库

- A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.
- 一个引用自带对象数据库的引用集合，所有对象都有它们可及的引用，可能还伴随带有来自“瓷器”的元数据。一个版本库可以通过替换机制与另外一个版本库共享它的对象数据库。

resolve 解决

- The action of fixing up manually what a failed automatic merge left behind.

- 手工修复自动合并失败所遗留的问题。

revision 修订

- A particular state of files and directories which was stored in the object database. It is referenced by a commit object.
- 文件和目录的具体状态，它被保存在对象数据库中。该对象被一个交付对象引用。

rewind 逆转

- To throw away part of the development, i.e. to assign the head to an earlier revision.
- 抛弃部分的开发工作，例如，将头设定到较早时的修订。

SCM

- Source code management (tool).
- 源代码管理（工具）。

SHA1

- Synonym for object name.
- 对象名的同义词。

shallow repository 浅库

- A shallow repository has an incomplete history some of whose commits have parents cauterized away (in other words, git is told to pretend that these commits do not have the parents, even though they are recorded in the commit object). This is sometimes useful when you are interested only in the recent history of a project even though the real history recorded in the upstream is much larger. A shallow repository is created by giving the —depth option to git-clone(1), and its history can be later deepened with git-fetch(1).

- 一个浅库是指一个版本库拥有一个不完整的项目历史，某些交付对象的先辈被剔除了（换言之，git 被告知那些交付是没有父辈的，即使它们的父辈其实记录在交付对象中）。这个有时候很有用，当你只是对项目的最近的历史感兴趣，而上游版本库的历史记录又非常庞大的时候。创建一个浅库只需在 git-clone(1) 命令中带上 --depth 选项，并且它的历史深度会被 git-fetch(1) 命令缩短。

symref 代号引用

- Symbolic reference: instead of containing the SHA1 id itself, it is of the format ref: refs/some/thing and when referenced, it recursively dereferences to this reference. HEAD is a prime example of a symref. Symbolic references are manipulated with the git-symbolic-ref(1) command.
- 代号引用：代替对象自己的 SHA1 识别号，当需要引用某个对象时，它的格式: refs/some/thing，git 会递归寻址该引用。HEAD 就是一个代号引号的好例子。代号引用由 git-synboblic-ref(1) 命令来维护。

tag 标签

- A ref pointing to a tag or commit object. In contrast to a head, a tag is not changed by a commit. Tags (not tag objects) are stored in $GIT_DIR/refs/tags/. A git tag has nothing to do with a Lisp tag (which would be called an object type in git's context). A tag is most typically used to mark a particular point in the commit ancestry chain.
- 一个指向一个标记或者是交付对象的引用。相对于头，一个标签是不会被一个交付对象改变的。标签（不是标签对象）保存在 $GIT_DIR/refs/tags 中。一个标签大多数情况下被用作标记交付对象的祖先链中的某个具体点。

tag object 标签对象

- An object containing a ref pointing to another object, which can contain a message just like a commit object. It can also contain a (PGP) signature, in which case it is called a "signed tag object".

- 一个包含指向其他对象的引用的对象，它包含一个信息，就如交付对象那样。它还可能包含一个(PGP)加密签注，在这中情况下，称它为“已签注标签对象”。

topic branch 主题分支

- A regular git branch that is used by a developer to identify a conceptual line of development. Since branches are very easy and inexpensive, it is often desirable to have several small branches that each contain very well defined concepts or small incremental yet related changes.
- 一个常规的 git 分支，开发者用来分辨某个概念性的开发线路。创建分支是非常容易和开销很低的，时常用一系列的小分支来装载各个概念的实现，或者是一组关联的变更。

tracking branch 跟随分支

- A regular git branch that is used to follow changes from another repository. A tracking branch should not contain direct modifications or have local commits made to it. A tracking branch can usually be identified as the right-hand-side ref in a Pull: refspec.
- 一个常规的分支，用于跟踪其他版本库的变更。一个跟随分支不应该被直接修改或者是向它提交本地的交付对象。一个跟随在拉入 (pull) 操作时，通常是在引用规则 (refspec) 的冒号右侧。

tree 树

- Either a working tree, or a tree object together with the dependent blob and tree objects (i.e. a stored representation of a working tree).
- 要么是指你的工作目录树，要么是指一个树对象它汇集了有关的片对象和树对象（一个对工作树的状态描述）。

tree object 树对象

- An object containing a list of file names and modes along with refs to the associated blob and/or tree objects. A tree is equivalent to a directory.

- 一个对象，它包含一个文件名称和属性的列表，以及它所关联的片对象和树对象的引用。一个树就如同一个目录。

tree-ish

- A ref pointing to either a commit object, a tree object, or a tag object pointing to a tag or commit or tree object.
- 一个引用，要么是指向一个交付对象，树对象，要么是一个标签对象，它指向一个标签或者是交付对象，或者是树对象。

Appendix A. Git Quick Reference 附录 A. Git 快速参考

This is a quick summary of the major commands; the previous chapters explain how these work in more detail.

这是一个主要命令的快捷摘要；前面的章节已经详细阐述了如何使用它们。

Creating a new repository 创建一个新的版本库

From a tarball:

从一个压缩包中创建：

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

From a remote repository:

从远程版本库创建：

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

Managing branches 管理分支

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

Instead of basing a new branch on current HEAD (the default), use:

创建一个不以当前的 HEAD 为起点的分支，用：

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

Create and switch to a new branch at the same time:

创建并同时切换至新的分支：

```
$ git checkout -b new v2.6.15
```

Update and examine branches from the repository you cloned from:

更新和检验从远程版本库中克隆过来的分支：

```
$ git fetch             # update
$ git branch -r         # list
  origin/master
  origin/next
```

```
  ...
$ git checkout -b masterwork origin/master
```

Fetch a branch from a different repository, and give it a new name in your repository:

从不同的版本库中抓取分支，并给予一个在你的版本库中新的分支名称：

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

Keep a list of repositories you work with regularly:

给你要定期地协同工作的版本库制作一个列表：

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

Exploring history 勘查历史

```
$ gitk                      # visualize and browse history
$ git log                   # list all commits
$ git log src/              # ...modifying src/
$ git log v2.6.15..v2.6.16  # ...in v2.6.16, not in v2.6.15
```

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

Search for regressions:

查找撤退点：

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

Making changes 制作变更

Make sure git knows who to blame:

告诉 git 责任该由谁负：

```
$ cat >>~/.gitconfig <<\EOF
[user]
        name = Your Name Comes Here
```

```
        email = you@yourdomain.example.com
EOF
```

Select file contents to include in the next commit, then make the commit:

选择这下次提交的时候要包含那些文件，接着制作交付：

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

Or, prepare and create the commit in one step:

或者是准备提交和创建交付一步完成：

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

Merging 合并

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

Sharing your changes 共享你的变更

Importing or exporting patches:

引入或者导出补丁：

```
$ git format-patch origin..HEAD # format a patch for each commit
```

```
                                    # in HEAD but not in origin
$ git am mbox # import patches from the mailbox "mbox"
```

Fetch a branch in a different git repository, then merge into the current branch:

抓取一个不同的 git 版本库的分支，并合并进当前分支：

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

Store the fetched branch into a local branch before merging into the current branch:

在合并至当前分支之前，将远程分支的变更保存为本地的分支：

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

After creating commits on a local branch, update the remote branch with your commits:

创建了本地分支的交付之后，用这些交付更新远程分支。

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

When remote and local branch are both named "test":

当本地和远程分支都是叫 "test" 时：

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

Shortcut version for a frequently used remote repository:

对于经常通讯的远程版本库，有快捷命令的版本：

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

Repository maintenance 版本库的维护

Check for corruption:

检查损坏：

```
$ git fsck
```

Recompress, remove unused cruft:

重新打包，删除无用的杂物：

```
$ git gc
```

Appendix B. Notes and todo list for this manual 附录 B. 备忘与本手册的工作计划

This is a work in progress.

本手册的编写工作，仍然在进行中。

The basic requirements:

基本需求：

- It must be readable in order, from beginning to end, by someone intelligent with a basic grasp of the UNIX command line, but without any special knowledge of git. If necessary, any other prerequisites should be specifically mentioned as they arise.
- Whenever possible, section headings should clearly describe the task they explain how to do, in language that requires no more knowledge than necessary: for example, "importing patches into a project" rather than "the git-am command"

- 它必须是从头到尾按顺序阅读的话是易于理解的，读者的对象是那些有基本的 UNIX 命令行知识，但是对 git 没有什么特别认识的人。若有必要的话，其他的背景知识就提示他们注意。
- 尽可能地在每个章节的开头就阐明他们准备说明什么东西，千言万语都不及动手操作实在：例如，"importing patches into a project" 就不如 "the git-am command"。

Think about how to create a clear chapter dependency graph that will allow people to get to important topics without necessarily reading everything in between.

考虑一下如何创建一个形象地说明问题的章节，使得读者明白它的主题而不必在其他资料和该章节中来回翻阅。

Scan Documentation/ for other stuff left out; in particular:

浏览 git 源代码中的 Documentation/ 目录，看看本文还有什么遗漏的；具体地：

- howto's
- some of technical/?
- hooks
- list of commands in git(1)
- 如何操作
- 某些技巧
- 钩子
- git 命令的列表

Scan email archives for other stuff left out

参看邮件列表看看本文有什么遗留。

Scan man pages to see if any assume more background than this manual provides.

查看帮助页，看看本文指出的背景知识当中还有什么需要补充。

Simplify beginning by suggesting disconnected head instead of temporary branch creation?

对于离线头替代临时分支的创建的表述，你有什么简明地阐述的建议？

Add more good examples. Entire sections of just cookbook examples might be a good idea; maybe make an "advanced examples" section a standard end-of-chapter section?

加入更多的好例子。整个章节都带菜谱那样的例子来说明问题应该是个好主意；在每一章的结尾都有一个“高级范例”的节作为每一章的收尾你认为如何？

Include cross-references to the glossary, where appropriate.

包含一个参考资料的词汇表，以及资料的出处。

Document shallow clones? See draft 1.5.0 release notes for some documentation.

更详细的浅克隆文档，可以参考 1.5.0 版本的草案发布注解，那里有一些文档。

Add a section on working with other version control systems, including CVS, Subversion, and just imports of series of release tarballs.

加入与其他修订控制系统的协调工作的章节，包括 CVS, Subversion，以及导入发行压缩包串。

More details on gitweb?

需要详细介绍 gitweb 吗？

Write a chapter on using plumbing and writing scripts.

写一章来介绍如何使用“水管”和如何写脚本吧。

Alternates, clone -reference, etc.

还有对 clone -reference 的介绍，等等。

More on recovery from repository corruption. See:
http://marc.theaimsgroup.com/?l=git&m=117263864820799&w=2
http://marc.theaimsgroup.com/?l=git&m=117147855503798&w=2

更多的版本库损坏修复，参看：
http://marc.theaimsgroup.com/?l=git&m=117263864820799&w=2
http://marc.theaimsgroup.com/?l=git&m=117147855503798&w=2

GitUserManualChinese (last edited 2009-03-30 22:03:21 by RobinSteven)