Git 中文教程

介绍

Git --- The stupid content tracker, 傻瓜内容跟踪器。Linus 是这样给我们介绍 Git 的。

Git 是用于 Linux 内核开发的版本控制工具。与常用的版本控制工具 CVS, Subversion 等不同， 它采用了分布式版本库的方式，不必服务器端软件支持，使源代码的发布和交流极其方便。 Git 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。 Git 最为出色的是它的合并跟踪（merge tracing）能力。

实际上内核开发团队决定开始开发和使用 Git 来作为内核开发的版本控制系统的时候， 世界开源社群的反对声音不少，最大的理由是 Git 太艰涩难懂，从 Git 的内部工作机制来说，的确是这样。 但是随着开发的深入，Git 的正常使用都由一些友好的脚本命令来执行，使 Git 变得非常好用， 即使是用来管理我们自己的开发项目，Git 都是一个友好，有力的工具。 现在，越来越多的著名项目采用 Git 来管理项目开发，例如：wine, U-boot 等，详情看 http://www.kernel.org/git

作为开源自由原教旨主义项目，Git 没有对版本库的浏览和修改做任何的权限限制。 它只适用于 Linux / Unix 平台，没有 Windows 版本，目前也没有这样的开发计划。

本文将以 Git 官方文档 Tutorial， core-tutorial 和 Everyday GIT 作为蓝本翻译整理，但是暂时去掉了对 Git 内部工作机制的阐述， 力求简明扼要，并加入了作者使用 Git 的过程中的一些心得体会，注意事项，以及更多的例子。 建议你最好通过你所使用的 Unix / Linux 发行版的安装包来安装 Git, 你可以在线浏览本文 ，也可以通过下面的命令来得到本文最新的版本库，并且通过后面的学习用 Git 作为工具参加到本文的创作中来。

```
$ git-clone http://www.bitsun.com/git/gittutorcn.git
```

创建一个版本库：git-init-db

创建一个 Git 版本库是很容易的，只要用命令 `git-init-db` 就可以了。 现在我们来为本文的写作创建一个版本库：

```
$ mkdir gittutorcn
$ cd gittutorcn
$ git-init-db
```

git 将会作出以下的回应

```
defaulting to local storage area
```

这样，一个空的版本库就创建好了，并在当前目录中创建一个叫 .git 的子目录。 你可以用 `ls -a` 查看一下，并请注意其中的三项内容：

- 一个叫 HEAD 的文件，我们现在来查看一下它的内容：

  ```
  $ cat .git/HEAD
  ```

  现在 HEAD 的内容应该是这样：

  ```
  ref: refs/heads/master
  ```

  我们可以看到，HEAD 文件中的内容其实只是包含了一个索引信息， 并且，这个索引将总是指向你的项目中的当前开发分支。
- 一个叫 objects 的子目录，它包含了你的项目中的所有对象，我们不必直接地了解到这些对象内容， 我们应该关心是存放在这些对象中的项目的数据。

  Note 关于 git 对象的分类，以及 git 对象数据库的说明， 请参看 [Discussion]

- 一个叫 refs 的子目录，它用来保存指向对象的索引。

具体地说，子目录 `refs` 包含着两个子目录叫 `heads` 和 `tags`， 就像他们的名字所表达的意味一样：他们存放了不同的开发分支的头的索引, 或者是你用来标定版本的标签的索引。

请注意：`master` 是默认的分支，这也是为什么 `.git/HEAD` 创建的时候就指向 `master` 的原因，尽管目前它其实并不存在。 git 将假设你会在 `master` 上开始并展开你以后的工作，除非你自己创建你自己的分支。

另外，这只是一个约定俗成的习惯而已，实际上你可以将你的工作分支叫任何名字， 而不必在版本库中一定要有一个叫 `master` 的分支，尽管很多 git 工具都认为 `master` 分支是存在的。

现在已经创建好了一个 git 版本库，但是它是空的，还不能做任何事情，下一步就是怎么向版本库植入数据了。

植入内容跟踪信息：git-add

为了简明起见，我们创建两个文件作为练习：

```
$ echo "Hello world" > hello
$ echo "Silly example" > example
```

我们再用 `git-add` 命令将这两个文件加入到版本库文件索引当中：

```
$ git-add hello example
```

`git-add` 实际上是个脚本命令，它是对 git 内核命令 `git-update-index` 的调用。 因此上面的命令和下面的命令其实是等价的：

```
$ git-update-index --add hello example
```

如果你要将某个文件从 git 的目录跟踪系统中清除出去，同样可以用 `git-update-index` 命令。 例如：

```
$ git-update-index --force-remove foo.c
```

Note `git-add` 可以将某个目录下的所有内容全都纳入内容跟踪之下，例如：`git-add ./path/to/your/wanted`。 但是在这样做之前， 应该注意先将一些我们不希望跟踪的文件清理掉， 例如，gcc 编译出来的 *.o 文件，vim 的交换文件 .*.swp 之类。

应该建立一个清晰的概念就是，`git-add` 和 `git-update-index` 只是刷新了 git 的跟踪信息，hello 和 example 这两个文件中的内容并没有提交到 git 的内容跟踪范畴之内。

提交内容到版本库：git-commit

既然我们刷新了 Git 的跟踪信息，现在我们看看版本库的状态：

```
$ git-status
```

我们能看到 git 的状态提示：

```
#
# Initial commit
#
#
# Updated but not checked in:
#   (will commit)
```

```
#
#       new file: example
#       new file: hello
#
```

提示信息告诉我们版本库中加入了两个新的文件，并且 git 提示我们提交这些文件， 我们可以通过 `git-commit` 命令来提交：

```
$ git-commit -m "Initial commit of gittutor reposistory"
```

查看当前的工作：git-diff

`git-diff` 命令将比较当前的工作目录和版本库数据库中的差异。 现在我们编辑一些文件来体验一下 git 的跟踪功能。

```
$ echo "It's a new day for git" >> hello
```

我们再来比较一下，当前的工作目录和版本库中的数据的差别。

```
$ git-diff
```

差异将以典型的 patch 方式表示出来：

```
diff --git a/hello b/hello
index a5c1966..bd9212c 100644
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello, world
+It's a new day for git
```

此时，我们可以再次使用组合命令 `git-update-index` 和 `git-commit` 将我们的工作提交到版本库中。

```
$ git-update-index hello
$ git-commit -m "new day for git"
```

实际上，如果要提交的文件都是已经纳入 git 版本库的文件，那么不必为这些文件都应用 `git-update-index` 命令之后再进行提交，下面的命令更简捷并且和上面的命令是等价的。

```
$ git-commit -a -m "new day for git"
```

管理分支：git-branch

直至现在为止，我们的项目版本库一直都是只有一个分支 `master`。 在 git 版本库中创建分支的成本几乎为零，所以，不必吝啬多创建几个分支。 下面列举一些常见的分支策略，仅供大家参考：

- 创建一个属于自己的个人工作分支，以避免对主分支 `master` 造成太多的干扰， 也方便与他人交流协作。
- 当进行高风险的工作时，创建一个试验性的分支，扔掉一个烂摊子总比收拾一个烂摊子好得多。
- 合并别人的工作的时候，最好是创建一个临时的分支，关于如何用临时分支合并别人的工作的技巧， 将会在后面讲述。

创建分支

下面的命令将创建我自己的工作分支，名叫 robin，并且将以后的工作转移到这个分支上开展。

```
$ git-branch robin
$ git-checkout robin
```

删除分支

要删除版本库中的某个分支，使用 `git-branch -D` 命令就可以了，例如：

```
$ git-branch -D branch-name
```

查看项目的发展变化和比较差异

这一节介绍几个查看项目的版本库的发展变化以及比较差异的很有用的命令：

git-show-branch

git-diff

git-whatchanged

我们现在为 robin, master 两个分支都增加一些内容。

```
$ git-checkout robin
$ echo "Work, work, workd" >> hello
$ git-commit -m "Some workd" -i hello
```

```
$ git-checkout master
$ echo "Play, play, play" >> hello
$ echo "Lots of fun" >> example
$ git-commit -m "Some fun" -i hello example
```

git-show-branch 命令可以使我们看到版本库中每个分支的世系发展状态， 并且可以看到每次提交的内容是否已进入每个分支。

```
$ git-show-branch
```

这个命令让我们看到版本库的发展记录。

```
* [master] Some fun
 ! [robin] some work
--
*  [master] Some fun
 + [robin] some work
*+ [master^] a new day for git
```

譬如我们要查看世系标号为 master^ 和 robin 的版本的差异情况， 我们可以使用这样的命令：

```
$ git-diff master^ robin
```

我们可以看到这两个版本的差异：

```
diff --git a/hello b/hello
index 263414f..cc44c73 100644
--- a/hello
+++ b/hello
@@ -1,2 +1,3 @@
 Hello World
 It's a new day for git
+Work, work, work
```

Note 关于 GIT 版本世系编号的定义，请参看 git-rev-parse 。

我们现在再用 `git-whatchanged` 命令来看看 `master` 分支是怎么发展的。

```
$ git-checkout master
$ git-whatchanged
```

```
diff-tree 1d2fa05... (from 3ecebc0...)
Author: Vortune.Robin
Date:   Tue Mar 21 02:24:31 2006 +0800

    Some fun

:100644 100644 f24c74a... 7f8b141... M  example
:100644 100644 263414f... 06fa6a2... M  hello

diff-tree 3ecebc0... (from 895f09a...)
Author: Vortune.Robin
Date:   Tue Mar 21 02:17:23 2006 +0800

    a new day for git

:100644 100644 557db03... 263414f... M  hello
```

从上面的内容中我们可以看到，在 `robin` 分支中的日志为 "Some work" 的内容, 并没有在 `master` 分支中出现。

合并两个分支：git-merge

既然我们为项目创建了不同的分支， 那么我们就要经常地将自己或者是别人在一个分支上的工作合并到其他的分支上去。 现在我们看看怎么将 `robin` 分支上的工作合并到 `master` 分支中。 现在转移我们当前的工作分支到 `master`，并且将 `robin` 分支上的工作合并进来。

```
$ git-checkout master
$ git-merge "Merge work in robin" HEAD robin
```

合并两个分支，还有一个更简便的方式，下面的命令和上面的命令是等价的。

```
$ git-checkout master
$ git-pull . robin
```

但是，此时 git 会出现合并冲突提示：

```
Trying really trivial in-index merge...
fatal: Merge requires file-level merging
Nope.
Merging HEAD with d2659fcf690ec693c04c82b03202fc5530d50960
Merging:
1d2fa05b13b63e39f621d8ee911817df0662d9b7 Some fun
d2659fcf690ec693c04c82b03202fc5530d50960 some work
found 1 common ancestor(s):
3ecebc0cb4894a33208dfa7c7c6fc8b5f9da0eda a new day for git
Auto-merging hello
CONFLICT (content): Merge conflict in hello

Automatic merge failed; fix up by hand
```

git 的提示指出，在合并作用于文件 hello 的 'Some fun' 和 'some work' 这两个对象时有冲突， 具体通俗点说，就是在 `master`, `robin` 这两个分支中的 hello 文件的某些相同的行中的内容不一样。 我们需要手动解决这些冲突，现在先让我们看看现在的 hello 文件中的内容。

```
$ cat hello
```

此时的 hello 文件应是这样的，用过其他的版本控制系统的朋友应该很容易看出这个典型的冲突表示格式：

```
Hello World
It's a new day for git
<<<<<<< HEAD/hello
Play, play, play
=======
Work, work, work
>>>>>>> d2659fcf690ec693c04c82b03202fc5530d50960/hello
```

我们用编辑器将 hello 文件改为：

```
Hello World
It's a new day for git
Play, play, play
Work, work, work
```

现在可以将手动解决了冲突的文件提交了。

```
$ git-commit -i hello
```

以上是典型的两路合并（2-way merge）算法，绝大多数情况下已经够用。 但是还有更复杂的三路合并和多内容树合并的情况。详情可参看： git-read-tree， git-merge 等文档。

逆转与恢复：git-reset

项目跟踪工具的一个重要任务之一，就是使我们能够随时逆转（Undo）和恢复（Redo）某一阶段的工作。

git-reset 命令就是为这样的任务准备的。 它将当前的工作分支的 头 定位到以前提交的任何版本中，它有三个重置的算法选项。

命令形式：

git-reset [--mixed | --soft | --hard] [<commit-ish>]

命令的选项：

--mixed
: 仅是重置索引的位置，而不改变你的工作树中的任何东西（即，文件中的所有变化都会被保留， 也不标记他们为待提交状态），并且提示什么内容还没有被更新了。这个是默认的选项。

--soft
: 既不触动索引的位置，也不改变工作树中的任何内容，我们只是要求这些内容成为一份好的内容 （之后才成为真正的提交内容）。 这个选项使你可以将已经提交的东西重新逆转至“已更新但未提交（Updated but not Check in）”的状态。 就像已经执行过 `git-update-index` 命令，但是还没有执行 `git-commit` 命令一样。

--hard
: 将工作树中的内容和头索引都切换至指定的版本位置中，也就是说自 <commit-ish> 之后的所有的跟踪内容和工作树中的内容都会全部丢失。 因此，这个选项要慎用，除非你已经非常确定你的确不想再看到那些东西了。

一个重要技巧 - - 逆转提交与恢复

可能有人会问，--soft 选项既不重置头索引的位置，也不改变工作树中的内容， 那么它有什么用呢？现在我们介绍一个 --soft 选项的使用技巧。 下面我们用例子来说明：

```
$ git-checkout master
$ git-checkout -b softreset
$ git-show-branch
```

这里我们创建了一个 `master` 的拷贝分支 `softreset`， 现在我们可以看到两个分支是在同一起跑线上的。

```
! [master] Merge branch 'robin'
```

```
 ! [robin] some work
  * [softreset] Merge branch 'robin'
---
- - [master] Merge branch 'robin'
+ * [master^] Some fun
++* [robin] some work
```

我们为 文件增加一些内容并提交。

```
$ echo "Botch, botch, botch" >> hello
$ git-commit -a -m "some botch"
$ git-show-branch
```

我们可以看到此时 `softreset` 比 `master` 推进了一个版本 "some botch" 。

```
! [master] Merge branch 'robin'
 ! [robin] some work
  * [softreset] some botch
---
  * [softreset] some botch
- - [master] Merge branch 'robin'
+ * [master^] Some fun
++* [robin] some work
```

现在让我们来考虑这样的一种情况，假如我们现在对刚刚提交的内容不满意， 那么我们再编辑项目的内容，再提交的话，那么 "some botch" 的内容就会留在版本库中了。 我们当然不希望将有明显问题的内容留在版本库中，这个时候 `--soft` 选项就很有用了。 为了深入了解 `--soft` 的机制，我们看看现在 softreset 分支的头和 ORIG_HEAD 保存的索引。

```
$ cat .git/refs/heads/softreset .git/ORIG_HEAD
```

结果如下：

```
5e7cf906233e052bdca8c598cad2cb5478f9540a
7bbd1370e2c667d955b6f6652bf8274efdc1fbd3
```

现在用 `--soft` 选项逆转刚才提交的内容：

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

现在让我们再看看 .git/ORIG_HEAD 的中保存了什么？

```
$ cat .git/ORIG_HEAD
```

结果如下：

```
5e7cf906233e052bdca8c598cad2cb5478f9540a
```

看！现在的 .git/ORIG_HEAD 等于逆转前的 .git/refs/heads/softreset 。 也就是说，`git-reset --soft HEAD^` 命令逆转了刚才提交的版本进度， 但是它将那次提交的对象的索引拷贝到了 .git/ORIG_HEAD 中。

我们再编辑 hello 文件成为下面的内容：

```
Hello World
It's a new day for git
Play, play, play
Work, work, work
Nice, nice, nice
```

我们甚至可以比较一下现在的工作树中的内容和被取消了的那次提交的内容有什么差异：

```
$ git-diff ORIG_HEAD
```

结果如下：

```
diff --git a/hello b/hello
index f978676..dd02c32 100644
--- a/hello
+++ b/hello
@@ -2,4 +2,4 @@ Hello World
 It's a new day for git
 Play, play, play
 Work, work, work
-Botch, botch, botch
+Nice, nice, nice
```

接着，我们可以恢复刚才被取消了的那次提交了。

```
$ git-commit -a -c ORIG_HEAD
```

注意，这个命令会打开默认的文本编辑器以编辑原来提交的版本日志信息，我们改为 "nice work" 。 大家可以自行用 `git-show-branch` 命令来查看一下现在的分支状态。 并且我们还可以不断地重复上述的步骤，一直修改到你对这个版本进度满意为止。

git-reset 命令还有很多的用途和技巧，请参考 git-reset ，以及 Everyday GIT with 20 commands or So 。

提取版本库中的数据

这是个很有用的小技巧，如果你对你现在的工作目录下的东西已经不耐烦了， 随时可以取出你提交过的东西覆盖掉当前的文件，譬如：

```
$ git-checkout -f foo.c
```

标定版本

在 git 中，有两种类型的标签，“轻标签”和“署名标签”。

技术上说，一个“轻标签”和一个分支没有任何区别，只不过我们将它放在了 `.git/refs/tags/` 目录， 而不是 `heads` 目录。因此，打一个“轻标签”再简单不过了。

```
$ git-tag my-first-tag
```

“署名标签”是一个真正的 git 对象，它不但包含指向你想标记的状态的指针，还有一个标记名和信息， 可选的 PGP 签名。 你可以通过 `-a` 或者是 `-s` 选项来创建“署名标签”。

```
$ git-tag -s <tag-name>
```

合并外部工作

通常的情况下，合并其他的人的工作的情况会比合并自己的分支的情况要多， 这在 git 中是非常容易的事情，和你运行 `git-merge` 命令没有什么区别。 事实上，远程合并的无非就是“抓取（fetch）一个远程的版本库中的工作到一个临时的标签中”， 然后再使用 `git-merge` 命令。

可以通过下面的命令来抓取远程版本库:

```
$ git-fetch <remote-repository>
```

根据不同的远程版本库所使用的通讯协议的路径来替代上面的 `remoted-repository` 就可以了。

Rsync

```
rsync://remote.machine/patch/to/repo.git/
```

SSH

`remote.machine:/path/to/repo.git`

or

`ssh://remote.machine/patch/to/repo.git/`

这是可以上传和下载的双向传输协议，当然，你要有通过 `ssh` 协议登录远程机器的权限。 它可以找出两端的机器提交过的对象集之中相互缺少了那些对象，从而得到需要传输的最小对象集。 这是最高效地交换两个版本库之间的对象的方式（在 git 兼容的所有传输协议当中）。

下面是个取得 SSH 远程版本库的命令例子：

```
$ git-fetch robin@192.168.1.168:/path/to/gittutorcn.git  (1)
```

(1) 这里 `robin` 是登录的用户名，`192.168.1.168` 是保存着主版本库的机器的 `IP` 地址。

Local directory

`/path/to/repo.git/`

本地目录的情况和 SSH 情况是一样的。

git Native

`git://remote.machine/path/to/repo.git/`

git 自然协议是设计来用于匿名下载的，它的工作方式类似于 SSH 协议的交换方式。

HTTP(S)

`http://remote.machine/path/to/repo.git/`

到这里可能有些朋友已经想到， 实际上，我们可以通过 Rsync, SSH 之类的双向传输方式来建立类似 CVS，SVN 这样的中心版本库模式的开发组织形式。

通过电子邮件交换工作

读过上一节之后，有的朋友可能要问，如果版本库是通过单向的下载协议发布的，如 HTTP， 我们就无法将工作上传到公共的版本库中。 别人也不能访问我的机器来抓取我的工作，那怎么办呢？

不必担心，我们还有 email ！别忘了 git 本来就是为了管理 Linux 的内核开发而设计的。 所以，它非常适合像 Linux Kernel 这样的开发组织形式高度分散，严重依赖 email 来进行交流的项目。

下面模拟你参加到《Git 中文教程》的编写工作中来，看看我们可以怎么通过 email 进行工作交流。 你可以通过下面的命令下载这个项目的版本库。

```
$ git-clone http://www.bitsun.com/git/gittutorcn.git
```

之后，你会在当前目录下得到一个叫 `gittutorcn` 的目录， 这就是你的项目的工作目录了。默认地，它会有两个分支： `master` 和 `origin`，你可以直接在 `master` 下展开工作，也可以创建你自己的工作分支，但是千万不要修改 `origin` 分支，切记！ 因为它是公共版本库的镜像，如果你修改了它， 那么就不能生成正确的对公共版本库的 `patch` 文件了。

> **Note** 如果你的确修改过 `origin` 分支的内容，那么在生成 `patch` 文件之前，请用 `git-reset --hard` 命令将它逆转到最原始的，没经过任何修改的状态。

你可以直接在 `master` 下开展工作，也可以创建你自己的工作分支。 当你对项目做了一定的工作，并提交到库中。我们用 `git-show-branch` 命令先看下库的状态。

```
* [master] your buddy's contribution
 ! [origin] degining of git-format-patch example
--
*  [master] your buddy's contribution
*+ [origin] degining of git-format-patch example
```

上面就假设你已经提交了一个叫 "your buddy's contribution" 的工作。 现在我们来看看怎么通过 email 来交流工作了。

```
$ git-fetch origin    (1)
$ git-rebase origin    (2)
$ git-format-patch origin     (3)

(1)更新 origin 分支，防止 origin 分支不是最新的公共版本，产生错误的补丁文件；
(2)将你在 master 上提交的工作迁移到新的源版本库的状态的基础上；
(3)生成补丁文件；
```

上面的几个命令，会在当前目录下生成一个大概名为 `0001-your-buddy-s-contribution.txt` 补丁文件, 建议你用文本工具查看一下这个文件的具体形式，然后将这个文件以附件的形式发送到项目维护者的邮箱： vortune@gmail.com

当项目的维护者收到你的邮件后，只需要用 `git-am` 命令，就可以将你的工作合并到项目中来。

```
$ git-checkout -b buddy-incomming
$ git-am /path/to/0001-your-buddy-s-contribution.txt
```

用 Git 协同工作

假设 Alice 在一部机器上自己的个人目录中创建了一个项目 /home/alice/project, Bob 想在同一部机器自己的个人目录中为这个项目做点什么。

Bob 首先这样开始：

```
$ git-clone /home/alice/project myrepo
```

这样就创建了一个保存着 Alice 的版本库的镜像的新目录 "myrepo"。 这个镜像保存着原始项目的起点和它的发展历程。

接着 Bob 对项目做了些更改并提交了这些更改：

```
(编辑一些文件)

$ git-commit -a

(如果需要的话再重复这个步骤)
```

当他搞定之后，他告诉 Alice 将他的东西从 /home/bob/myrepo 中引入，她只需要这样：

```
$ cd /home/alice/project
$ git pull /home/bob/myrepo
```

这样就将 Bob 的版本库中的 "master" 分支的变化引入了。 Alice 也可以通过在 pull 命令的后面加入参数的方式来引入其他的分支。

在导入了 Bob 的工作之后，用 "git-whatchanged" 命令可以查看有什么信的提交对象。 如果这段时间里以来，Alice 也对项目做过自己的修改，当 Bob 的修改被合并进来的时候，那么她需要手动修复所有的合并冲突。

谨慎的 Alice 在导入 Bob 的工作之前，希望先检查一下。 那么她可以先将 Bob 的工作导入到一个新创建的临时分支中， 以方便研究 Bob 的工作：

```
$ git fetch /home/bob/myrepo master:bob-incoming
```

这个命令将 Bob 的 master 分支的导入到名为 bob-incoming 的分支中（ 不同于 git-pull 命令，git-fetch 命令只是取得 Bob 的开发工作的拷贝， 而不是合并经来）。 接着：

```
$ git whatchanged -p master..bob-incoming
```

这会列出 Bob 自取得 Alice 的 master 分支之后开始工作的所有变化。 检查过这些工作，并做过必须的调整之后， Alice 就可以将变化导入到她的 master 分支中：

```
$ git-checkout master
$git-pull . bob-incoming
```

最后的命令就是将 "bob-incoming" 分支的东西导入到 Alice 自己的版本库中的， 稍后，Bob 就可以通过下面的命令同步 Alice 的最新变化。

```
$ git-pull
```

注意不需为这个命令加入 Alice 的版本库的路径，因为当 Bob 克隆 Alice 的版本库的时候， git 已经将这个路径保存到 .git/remote/origin 文件中，它将会是所以的导入操作的默认路径。

Bob 可能已经注意到他并没有在他的版本库中创建过分支（但是分支已经存在了）：

```
$ git branch
* master
  origin
```

"origin" 分支，它是运行 "git-clone" 的时候自动创建的，他是 Alice 的 master 分支的原始镜像， Bob 应该永远不要向这个分支提交任何东西。

如果 Bob 以后决定在另外一部主机上开展工作，那么他仍然需要通过 SSH 协议从新克隆和导入（ Alice 的版本库）：

```
$ git-clone alice.org:/home/alice/project/ myrepo
```

我们可以使用 git 自然协议，或者是 rsync, http 等协议的任何一种，详情请参考 git-pull。

Git 同样可以建立类似 CVS 那样的开发模式，也就是所有开发者都向中心版本库提交工作的方式， 详情参考 git_push 和 git for CVS users 。

为版本库打包

在前面，我们已经看到在 `.git/objects/??/` 目录中保存着我们创建的每一个 git 对象。这样的方式对于自动和安全地创建对象很有效，但是对于网络传输则不方便。 git 对象一旦创建了，就不能被改变，但有一个方法可以优化对象的存储，就是将他们“打包到一起”。

```
$ git repack
```

上面的命令让你做到这点。如果你一直是做着我们的例子过来的，你现在大约会在

上面的命令让你做到这点，如果你一直是跟着我们的例子过来的，你现在大约会在 `.git/objects/??/` 目录下积累了17个对象。 `git-repack` 会告诉你有几个对象被打包了，并且将他们保存在 `.git/objects/pack` 目录当中。

> **Note** 你将会看到两个文件，`pack-*.pack` and `pack-*.idx` 在 `.git/objects/pack` 目录。他们的关系是很密切的， 如果你手动将他们拷贝到别的版本库中的话，你要决定将他们一起拷贝。 前者是保存着所有被打包的数据的文件，后者是随机访问的索引。

如果你是个偏执狂，就运行一下 `git-verity-pack` 命令来检查一下有缺陷的包吧， 不过，其实你无须太多担心，我们的程序非常出色 ;-).

一旦你已经对那些对象打包了，那么那些已经被打过包的原始的对象，就没有必要保留了。

```
$ git prune-packed
```

会帮你清楚他们。

如果你好奇的话，你可以在执行 `git-prune-repacked` 命令之前和之后， 都运行一下 `find .git/objects -type f`，这样你就能看到有多少没有打包的对象， 以及节省了多少磁盘空间。

> **Note** `git pull` `git-pull` 对于 HTTP 传输来说，一个打包过的版本库会将一定数量的 相关联的对象放进一个有关联性的打包中。如果你设想多次从 HTTP 公共版本库中导入数据， 你也许要频繁地 reapck & prune，要么就干脆从不这样做。

如果你此时再次运行 `git-repack`，它就会说 "Nothing to pack"。 要是你继续开发，并且积累了一定数量的变迁，再运行 `git-repack` 将会创建一个新的包， 它会包含你自上次对库打包以来创建的对象。我们建议你尽快在初始化提交之后打包一下你的版本库（除非你现在的项目是个涂鸦式的草稿项目），并且在项目经历过一段很活跃的时期时， 再运行 `git-repack` 一下。

当一个版本库通过 `git-push` 和 `git-pull` 命令来同步源版本库中打包过的对像的时候，通常保存到目标版本库中的是解包了的对象，除非你使用的是 rsync（远程同步协议）协议的传输方式。 正是这种容许你在两头的版本库中有不同的打包策略的方式，他意味着你也许在过一段时间之后， 需要在两头的版本库中都重新打包一下。

将工作捆绑到一起

通过 git 的分支功能，你可以非常容易地做到好像在同一时间进行许多“相关 – 或 – 无关”的工作一样。

我们已经通过前面的 "fun and work" 使用两个分支的例子，看到分支是怎么工作的。 这样的思想在多于两个的分支的时候也是一样的，比方说，你现在在 master 的头， 并有些新的代码在 master 中，另外还有两个互不相关的补丁分别在 "commit-fix" 和 "diff-fix" 两个分支中。

```
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Release candidate #1
---
 +  [diff-fix] Fix rename detection.
 +  [diff-fix~1] Better common substring algorithm.
+   [commit-fix] Fix commit message normalization.
  * [master] Release candidate #1
++* [diff-fix~2] Pretty-print messages.
```

两个补丁我们都测试好了，到这里，你想将他们俩合并起来， 于是你可以先合并 *diff-fix* ，然后再合并 *commit-fix*，像这样：

```
$ git merge 'Merge fix in diff-fix' master diff-fix
$ git merge 'Merge fix in commit-fix' master commit-fix
```

结果如下：

```
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Merge fix in commit-fix
---
  - [master] Merge fix in commit-fix
+ * [commit-fix] Fix commit message normalization.
  - [master~1] Merge fix in diff-fix
 +* [diff-fix] Fix rename detection.
 +* [diff-fix~1] Better common substring algorithm.
  * [master~2] Release candidate #1
++* [master~3] Pretty-print messages.
```

然而，当你确信你手头上的确是一堆互不相关的项目变化时，就没有任何理由将这堆东西一个个地合并（假如他们的先后顺序很重要，那么他们就不应该被定以为无关的变化）， 你可以一次性将那两个分支合并到当前的分支中，首先我们将我们刚刚做过的事情逆转一下， 我们需要通过将 master 分支重置到 *master~2* 位置的方法来将它逆转到合并那两个分支之前的状态。

```
$ git reset --hard master~2
```

你可以用 `git-show-branch` 来确认一下的确是回到了两次 `git-merge` 的状态了。 现在你可以用一行命令将那两个分支导入的方式来替代两次运行（也就是所谓的 `炮制章鱼 --making an Octopus`）`git-merge`：

```
$ git pull . commit-fix diff-fix
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Octopus merge of branches 'diff-fix' and 'commit-fix'
---
  - [master] Octopus merge of branches 'diff-fix' and 'commit-fix'
+ * [commit-fix] Fix commit message normalization.
 +* [diff-fix] Fix rename detection.
 +* [diff-fix~1] Better common substring algorithm.
  * [master~1] Release candidate #1
++* [master~2] Pretty-print messages.
```

注意那些不适合制作章鱼的场合，尽管你可以那样做。一只“章鱼”往往可以使项目的提交历史更具可读性， 前提是你在同一时间导入的两份以上的变更是互不关联的。 然而，如果你在合并任何分支的过程中出现合并冲突，并且需要手工解决的话， 那意味着这些分支当中有相互干涉的开发工作在进行，那么你就应该将这个两个冲突先合并， 并且记录下你是如何解决这个冲突，以及你首先处理他们的理由。（译者按：处理完冲突之后，你就可以放心制作“章鱼”了） 否则的话将会造成项目的发展历史很难跟踪。

管理版本库

版本库的管理员可以用下面的工具来建立和维护版本库。

- git-daemon(1) 容许匿名下载版本库。
- git-shell(1) 面向中心版本库模式 的用户的类似 受限的 *shell* 的命令。

update hook howto 一个很好的管理中心版本库的例子。

例子

在 */pub/scm* 上运行 *git* 守护进程

```
$ grep git /etc/inet.conf
git     stream  tcp     nowait  nobody \
  /usr/bin/git-daemon git-daemon --inetd --syslog --export-all /pub/scm
```

这个配置行应该在配置文件中用一行来写完。

仅给开发者 *push/pull* 的访问权限。

```
$ grep git /etc/passwd (1)
alice:x:1000:1000::/home/alice:/usr/bin/git-shell
bob:x:1001:1001::/home/bob:/usr/bin/git-shell
cindy:x:1002:1002::/home/cindy:/usr/bin/git-shell
david:x:1003:1003::/home/david:/usr/bin/git-shell
$ grep git /etc/shells (2)
/usr/bin/git-shell


(1) 将用户的登录 shell 设定为 /usr/bin/git-shell,
它除了运行 "git-push" 和 "git-pull" 不能做任何事。
这样用户就可以通过 ssh 来访问机器。
(2) 许多的发行版需要在 /etc/shells 配置文件中列明要用什么 shell 来作为登录 shell。
```

CVS - 模式的公共库。

```
$ grep git /etc/group (1)
git:x:9418:alice,bob,cindy,david
$ cd /home/devo.git
$ ls -l (2)
  lrwxrwxrwx   1 david git    17 Dec  4 22:40 HEAD -> refs/heads/master
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 branches
  -rw-rw-r--   1 david git    84 Dec  4 22:40 config
  -rw-rw-r--   1 david git    58 Dec  4 22:40 description
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 hooks
  -rw-rw-r--   1 david git 37504 Dec  4 22:40 index
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 info
  drwxrwsr-x   4 david git  4096 Dec  4 22:40 objects
  drwxrwsr-x   4 david git  4096 Nov  7 14:58 refs
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 remotes
$ ls -l hooks/update (3)
  -r-xr-xr-x   1 david git  3536 Dec  4 22:40 update
$ cat info/allowed-users (4)
refs/heads/master       alice\|cindy
refs/heads/doc-update   bob
refs/tags/v[0-9]*       david

(1) 将所有的开发人员都作为 git 组的成员。
(2) 并且给予他们公共版本库的写权限。
(3) 用一个在 Documentation/howto/ 中的 Carl 写的例子来实现版本库的分支控制策略。
(4) Alice 和 Cindy 可以提交入 master 分支，只有 Bob 能提交入 doc-update 分支，
David 则是发行经理只有他能创建并且 push 版本标签。
```

支持默协议传输的 *HTTP* 服务器。

```
dev$ git update-server-info (1)
dev$ ftp user@isp.example.com (2)
ftp> cp -r .git /home/user/myproject.git

(1) 保证 info/refs 和 object/info/packs 是最新的。
(2) 上传到你的 HTTP 服务器主机。
```

项目开发的模式推介

尽管 git 是一个正式项目发布系统，它却可以方便地将你的项目建立在松散的开发人员组织形式上。 Linux 内核的开发，就是按这样的模式进行的。在 Randy Dunlap 的著作中（"Merge to Mainline" 第17页） 就有很好的介绍（`http://tinyurl.com/a2jdg`）。

需要强调的是正真的**非常规**的开发组织形式， git 这种组织形式，意味着对于工作流程的约束，没有任何强迫性的原则。 你不必从唯一一个远程版本库中导入（工作目录）。

项目领导人（project lead）的工作推介

1. 在你自己的本地机器上准备好主版本库。你的所有工作都在这里完成。
2. 准备一个能让大家访问的公共版本库。

   如果其他人是通过默协议的方式（http）来导入版本库的，那么你有必要保持这个默协议的友好性。 `git-init-db` 之后，复制自标准模板库的 `$GIT_DIR/hooks/post-update` 将包含一个对 `git-update-server-info` 的调用，但是 `post-update` 默认是不能唤起它自身的。 通过 `chmod +x post-update` 命令使能它。这样让 `git-update-server-info` 保证那些必要的文件是最新的。
3. 将你的主版本库推入公共版本库。
4. `git-repack` 公共版本库。这将建立一个包含初始化提交对象集的打包作为项目的起始线， 可能的话，执行一下 `git-prune`，要是你的公共库是通过 pull 操作来从你打包过的版本库中导入的。
5. 在你的主版本库中开展工作，这些工作可能是你自己的最项目的编辑， 可能是你由 email 收到的一个补丁，也可能是你从这个项目的“子系统负责人” 的公共库中导入的工作等等。

   你可以在任何你喜欢的时候重新打包你的这个私人的版本库。
6. 将项目的进度推入公共库中，并给大家公布一下。
7. 尽管一段时间以后，"git-repack" 公共库。 并回到第5步继续工作。

项目的子系统负责人（subsystem maintainer）也有自己的公共库，工作流程大致如下：

1. 准被一个你自己的工作目录，它通过 `git-clone` 克隆自项目领导人的公共库。 原始的克隆地址（URL）将被保存在 `.git/remotes/origin` 中。
2. 准备一个可以给大家访问的公共库，就像项目领导人所做的那样。
3. 复制项目领导人的公共库中的打包文件到你的公共库中， 除非你的公共库和项目领导人的公共库是在同一部主机上。 以后你就可以通过 `objects/info/alternates` 文件的指向来 浏览它所指向的版本库了。
4. 将你的主版本库推入你的公共版本库，并运行 `git-repack`， 如果你的公共库是通过的公共库是通过 pull 来导入的数据的话， 再执行一下 `git-prune` 。
5. 在你的主版本库中开展工作。这些工作可能包括你自己的编辑，来自 email 的补丁， 从项目领导人，“下一级子项目负责人”的公共库哪里导入的工作等等。

你可以在任何时候重新打包你的私人版本库。

6. 将你的变更推入公共库中，并且请“项目领导人”和“下级子系统负责人”导入这些变更。
7. 每隔一段时间之后，`git-repack` 公共库。回到第 5 步继续工作。

“一般开发人员”无须自己的公共库，大致的工作方式是：

1. 准备你的工作库，它应该用 `git-clone` 克隆自“项目领导人”的公共库（如果你只是开发子项目，那么就克隆“子项目负责人”的）。 克隆的源地址（URL）会被保存到 `.git/remotes/origin` 中。
2. 在你的个人版本库中的 *master* 分支中开展工作。
3. 每隔一段时间，向上游的版本库运行一下 `git-fetch origin`。 这样只会做 `git-pull` 一半的操作，即只克隆不合并。 公共版本库的新的头就会被保存到 `.git/refs/heads/origins`。
4. 用 `git-cherry origin` 命令，看一下你有什么补丁被接纳了。 并用 `git-rebase origin` 命令将你以往的变更迁移到最新的上游版本库的状态中。 （关于 `git-rebase` 命令，请参考 git-rebase）
5. 用 `git-format-patch origin` 生成 email 形式的补丁并发给上游的维护者。 回到第二步接着工作。
