Git 魔法

Ben Lynn

2007 年 8 月

前言

Git 堪称版本控制瑞士军刀。这个可靠、多才多艺、用途多样的校订工具异常灵活，以致不易掌握，更别说精通了。

正如 Arthur C. Clarke 所说，足够先进的技术与魔法无二。这是学习 Git 的好办法：新手不妨忽略 Git 的内部机理，只当小把戏玩，借助 Git 其奇妙的能力，逗逗朋友，气气敌人。

为了不陷入细节，我们对特定功能提供大面上的讲解。在反复应用之后，慢慢地你会理解每个小技巧如何工作，以及如何组合这些技巧以满足你的需求。

- 简体中文：俊杰，萌和江薇。正体中文 由 + cconv -f UTF8-CN -t UTF8-TW + 转换。
- 法文：Alexandre Garel。也在 itaapy。
- 德文：Benjamin Bellee 和 Armin Stebich；也在 Armin 的网站。
- 葡萄牙文：Leonardo Siqueira Rodrigues [ODT 版]。
- 俄文：Tikhon Tarnavsky, Mikhail Dymskov, 和其他人。
- 西班牙：Rodrigo Toledo 和 Ariset Llerena Tapia。
- 越南文：Trần Ngọc Quân; 也在 他的网站.
- 单一文件：纯 HTML，无 CSS。
- PDF 文件：打印效果好.
- Debian 包，Ubuntu 包：本站快速本地拷贝。如果 下线了会方便些。
- 纸质书 [Amazon.com]：64 页, 15.24cm x 22.86cm, 黑白。没有电子设备的时候会方便些。

致谢！

那么多人对本文档的翻译让我受宠若惊。他们的付出拓宽了读者群，我非常感激。

Dustin Sallings、Alberto Bertogli、James Cameron、Douglas Livingstone、Michael Budde、Richard Albury、Tarmigan、Derek Mahar、Frode Aannevik、Keith Rarick、Andy Somerville、Ralf Recker、Øyvind A. Holm、Miklos Vajna、Sébastien Hinderer、Thomas Miedema、Joe Malin、和 Tyler Breisacher 对本文档正确性和优化做出了贡献。

François Marier 维护 Debian 包，该 Debian 包起初由 Daniel Baumann 创建。

感谢其他很多提供帮助和鼓励的人。名单太长了我无法一一写下。

如果我不小心把你的名字落下，请告诉我或者发一个补丁。

- http://repo.or.cz/ 为自由项目提供服务。第一个 Git 服务器。由最早的 Git 开发人员创建和维护。
- http://gitorious.org/ 是另一个 Git 服务站点，为开源项目提供服务。
- http://github.com/ 开源项目免费，私有项目收钱。

感谢以上站点为本文档提供伺服服务。

许可

入门

我将用类比的方式来介绍版本控制的概念。更严谨的解释参见 维基百科版本修订控制条目。

工作是玩

我从小就玩电脑游戏，直到今天；不过我只是在长大后才开始使用版本控制系统。我想我并不是个例，所以拿两者工作方式进行类比，可使一些概念更易解释，也易于理解。

编写代码，或编辑文档，和玩游戏差不多。在你做出了很多进展之后，你最好保存一下。要做到这点，点击你所信任的编辑器保存按钮就好了。

但这将覆盖老版本。就像那些学校里玩的老游戏，只有一个存档：你确实可以保存，但你不能回到更老的状态了。这真让人扫兴，因为那个状态可能恰好保存了这个游戏特别有意思一关，说不定哪天你想再玩一下呢。或者更糟糕的，你当前的保存是个必败局，这样你就不得不从头开始玩了。

版本控制

在编辑的时候，如果想保留旧版本，你可以将文件“另存为”一个不同的文件，或在保存之前将文件拷贝到别处。你可能会压缩这些文件以节省空间。这是一个初级的依赖手工进行的版本控制方式。游戏软件在这块早就做了很多提高，很多游戏都提供基于时间戳的多个存档槽。

让我们看看稍稍复杂的情况。比如你有很多放在一起的文件，比如项目源码，或网站文件。现在如你想保留旧版本，你不得不把整个目录存档。手工保存多个版本很不方便，而且很快会耗费巨大。

在一些电脑游戏里，一个存档真的包含在一个充满文件的目录里。这些游戏为玩家屏蔽了这些细节，并提供一个方便易用的界面来管理该目录的不同版本。

版本控制系统也没有不同。两者提供友好的用户界面，来管理目录里的东西。你可以频繁保存，也可以之后加载任一存档。不像大多数计算机游戏，版本控制系统通常精于节省存储空间。一般情况下，如果两个版本间只有少数文件的变更，每个文件的变更也不大，那就只存储差异的部分，而不是把全部拷贝的都保存下来，以节省存储空间。

分布控制

现在设想一个很难的游戏。太难打了，以至于世界各地很多骨灰级玩家决定组队，分享他们游戏存档以攻克它。Speedrun 就是现实中的例子：在同一个游戏里，玩家们分别攻克不同的等级，协同工作以创造惊人战绩。

你如何搭建一个系统，使得他们易于得到彼此的存档？并易于上载新的存档？

在过去，每个项目都使用中心式版本控制。某个服务器上放所有保存的游戏记录。其他人就不用再做备份了。每个玩家在他们机器上最多保留几个游戏记录。当一个玩家想更新至最新进度时候，他们需要把这个进度从主服务器下载下来，玩一会儿，保存并上载到主服务器以供其他人使用。

假如一个玩家由于某种原因，想得到一个较旧版本的游戏进度该怎么办？或许当前保存的游戏是一个注定的败局，因为某人在第三级忘记捡某个物品；他们希望能找到最近一个可以完成的游戏记录。或者他们想比较两个旧版本间的差异，来估算某个特定玩家干了多少活。

查看旧版本的理由有很多，但检查的办法都是一样的。他们必须去中心服务器索要那个旧版本的记录。需要的旧版本越多，和服务器的交互就越多。

Git 是新一代的版本控制系统中的一员，它的特点是分布式的，广义上也可以被看作是一种中心式系统。从主服务器下载时，玩家会得到所有保存的记录，而不仅是最新版。这看来，玩家们好像把中心服务器做了个镜像。

最初的克隆操作可能比较费时，特别当存档有很长历史的时候，但从长远看这是值得的。一个显而易见的好处是，当查看一个旧版本时，就不再需要和中心服务器通讯了。

一个误区

一个很常见的错误观念是，分布式系统不适合需要官方中心仓库的项目。这与事实并不相符。给谁照相也不会偷走他们的灵魂。类似地，克隆主仓库并不降低它的重要性。

一般来说，一个中心版本控制系统能做的任何事，一个良好设计的分布式系统都能做得更好。网络资源总要比本地资源耗费更昂贵。不过我们应该在稍后分析分布式方案的缺点，这样人们才不会按照习惯做出错误的比较。

一个小项目或许只需要分布式系统提供的一小部分功能，但是，在项目很小的时候，就理应使用规划并不好的系统？就好比说，在计算较小数目的时候应该使用罗马数字？

而且，你的项目的增长可能会超出你最初的预期。从一开始就使用 Git 好似带着一把瑞士军刀，尽管你很多时候只是用它来开开瓶盖。某天你迫切需要一把改锥，你就会庆幸你所有的不单单是一个启瓶器。

合并冲突

对于这个话题，电脑游戏的类比显得不够用。那让我们再来看看文档编辑的情况吧。

假设 Alice 在文档开头插入一行，并且 Bob 在文档末尾添加一行。他们都上传了他们的改动。大多数系统将自动给出一个合理的处理方式：接受且合并他们的改动，这样 Alice 和 Bob 两人的改动都会生效。

现在假设 Alice 和 Bob 对文件的同一行做了不同的改动。如果没有人工参与的话，这个冲突是无法解决的。第二个人在上载文件时，会收到 合并冲突的通知，要么用一个人的改动覆盖另一个的，要么完全修订这一行。

更复杂的情况也可能出现。版本控制系统自己处理相对简单的情况，把困难的情况留给人来处理。它们的行为通常是可配置的。

基本技巧

与其一头扎进 Git 命令的海洋中，不如来点基本的例子试试手。它们简单而且实用。实际上，在开始使用 Git 的头几个月，我所用的从来没超出本章介绍的内容。

保存状态

要不来点猛的？在做之前，先为当前目录所有文件做个快照，使用：

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

现在如果你的编辑乱了套，恢复之前的版本可以使用：

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

再次保存状态：

```
$ git commit -a -m "Another backup"
```

添加、删除、重命名

以上命令将只跟踪你第一次运行 **git add** 命令时就已经存在的文件。如果要添加新文件或子目录，你需要告诉 Git：

```
$ git add readme.txt Documentation
```

类似，如果你想让 Git 忘记某些文件：

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

这些文件如果还没被从系统中删除，Git 将会删除它们。

重命名文件同删除旧文件，并同时添加新文件一样。也有一个快捷方式 **git mv** ，和 **mv** 命令的用法一样。例如：

```
$ git mv bug.c feature.c
```

进阶撤销/重做

有时候你只想把某个时间点之后的所有改动都回滚掉，因为这些的改动是不正确的。那么使用这个命令：

```
$ git log
```

来显示最近提交列表，以及查看他们的 SHA1 哈希值:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

    Replace printf() with write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

哈希值的前几个字符足够确定一个提交；也可以拷贝粘贴完整的哈希值，输入：

```
$ git reset --hard 766f
```

来恢复到一个指定的提交状态，并从记录里永久抹掉所有比该记录新一些的提交。

另一些时候你想简单地回朔到某一个旧状态。这种情况，键入：

```
$ git checkout 82f5
```

这个操作将把你带回过去，同时也保留较新提交。然而，像科幻电影里时光旅行一样，如果你这时编辑并提交的话，你将身处另一个现实里，因为你的动作与开始时相比是不同的。

这另一个现实叫作“分支”(branch)，之后我们会对这点多讨论一些。至于现在，只要记住：

```
$ git checkout master
```

会把你带到当下来就可以了。另外，为避免 Git 的抱怨，应该在每次运行 checkout 之前提交 (commit) 或重置 (reset) 你的改动。

还以电脑游戏作为类比：

- **git reset --hard**: 加载一个旧记录并删除所有比之新的记录。
- **git checkout**: 加载一个旧记录，但如果你在这个记录上玩，游戏状态将偏离第一轮的较新状态。你现在打的所有游戏记录会在你刚进入的、代表另一个真实的分支里。我们稍后论述。

你可以选择只恢复特定文件和目录，这将通过将其加在命令之后来实现：

```
$ git checkout 82f5 some.file another.file
```

小心，这种形式的 **checkout** 会不声不响地覆盖当前文件。为阻止意外发生，在运行任何 checkout 命令之前做提交，尤其在初学 Git 的时候。通常，任何时候你觉得对运行某个命令不放心，无论 Git 命令还是不是 Git 命令，就先运行一下 **git commit -a** 。

不喜欢拷贝旧提交的哈希值？那就用：

```
$ git checkout :/"My first b"
```

来跳到以特定字符串开头的提交。你也可以回到倒数第五个保存状态：

```
$ git checkout master~5
```

撤销

在法庭上，事件可以从法庭记录里敲出来。同样，你可以检出特定提交以撤销。

```
$ git commit -a
$ git revert 1b6d
```

讲撤销给定哈希值的提交。本撤销被记录为一个新的提交，你可以通过运行 **git log** 来确认这一点。

变更日志生成

一些项目要求生成变更日志changelog. 若要生成一个变更日志，可以键入：

```
$ git log > ChangeLog
```

来实现。

下载文件

得到一个由 Git 管理的项目的拷贝，则键入：

```
$ git clone git://server/path/to/files
```

例如，得到我用来创建该站的所有文件：

```
$ git clone git://git.or.cz/gitmagic.git
```

我们很快会对 **clone** 命令谈的很多。

到最新

如果你已经使用 **git clone** 命令得到了一个项目的一份拷贝，你可以更新到最新版，通过：

```
$ git pull
```

快速发布

假设你写了一个脚本，想和他人分享。你可以只告诉他们从你的计算机下载，但如果此时你正在改进你的脚本，或加入了试验性质的改动，他们下载了你的脚本，他们可能因此陷入困境。当然，这就是发布周期存在的原因。开发人员可能频繁进行项目修改，但他们只在他们觉得代码可以见人的时候才择时发布。

用 Git 来完成这项，需要进入你的脚本所在目录：

```
$ git init
$ git add .
$ git commit -m "First release"
```

然后告诉你的用户去运行：

```
$ git clone your.computer:/path/to/script
```

来下载你的脚本。这要假定他们有 ssh 访问权限。如果没有，需要运行 **git daemon** 并告诉你的用户去运行：

```
$ git clone git://your.computer/path/to/script
```

从现在开始，每次你的脚本准备好发布时，就运行：

```
$ git commit -a -m "Next release"
```

而你的用户则可以进入包含你脚本的目录，并键入下列命令，来更新他们的版本：

```
$ git pull
```

你的用户永远也不会取到你不想让他们看到的脚本版本。显然这个技巧对所有代码库都适用，而不仅仅局限于脚本。

我们已经做了什么？

找出自从上次提交之后你已经做了什么改变：

```
$ git diff
```

或者自昨天的改变：

```
$ git diff "@{yesterday}"
```

或者一个特定版本与倒数第二个变更之间：

```
$ git diff 1b6d "master~2"
```

输出结果都是补丁格式，可以用 **git apply** 来把补丁打上。也可以试一下：

```
$ git whatchanged --since="2 weeks ago"
```

我也经常用qgit 浏览历史, 因为他的图形界面很养眼，或者 tig ，一个文本界面的东西，很慢的网络状况下也可以工作的很好。也可以安装 web 服务器，运行 **git instaweb** ，就可以用任意浏览器浏览了。

练习

比方 A，B，C，D 是四个连续的提交，其中 B 与 A 一样，除了一些文件删除了。我们想把这些删除的文件加回 D。我们如何做到这个呢？

至少有三个解决方案。假设我们在 D：

1. A 与 B 的差别是那些删除的文件。我们可以创建一个补丁代表这些差别，然后吧补丁打上：

   ```
   $ git diff B A | git apply
   ```

2. 既然这些文件存在 A，我们可以把它们拿出来：

   ```
   $ git checkout A foo.c bar.h
   ```

3. 我们可以把从 A 到 B 的变化视为可撤销的变更：

   ```
   $ git revert B
   ```

哪个选择最好？这取决于你的喜好。利用 Git 满足自己需求是容易，经常还有多个方法。

克隆代码库

在较老一代的版本控制系统里，checkout 是获取文件的标准操作。你将获得一组特定保存状态的文件。

在 Git 和其他分布式版本控制系统里，克隆是标准的操作。通过创建整个仓库的克隆来获得文件。或者说，你实际上把整个中心服务器做了个镜像。凡是主仓库上能做的事，你都能做。

计算机间同步

我可以忍受制作 tar 包或利用 rsync 来作备份和基本同步。但我有时在我笔记本上编辑，其他时间在台式机上，而且这俩之间也许并不交互。

在一个机器上初始化一个 Git 仓库并提交你的文件。然后转到另一台机器上：

```
$ git clone other.computer:/path/to/files
```

以创建这些文件和 Git 仓库的第二个拷贝。从现在开始，

```
$ git commit -a
$ git pull other.computer:/path/to/files HEAD
```

将把另一台机器上特定状态的文件“拉”到你正工作的机器上。如果你最近对同一个文件做了有冲突的修改，Git 将通知你，而你也应该在解决冲突之后再次提交。

典型源码控制

为你的文件初始化 Git 仓库：

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

在中心服务器，在某个目录初始化一个“裸仓库”：

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$ touch proj.git/git-daemon-export-ok
```

如需要的话，启动 Git 守护进程：

```
$ git daemon --detach  # 它也许已经在运行了
```

对一些 Git 伺服服务，按照其提示来初始化空 Git 仓库。一般是在服务器网页上填写一个表单。

把你的项目“推送”到中心服务器：$ git push central.server/path/to/proj.git HEAD

若要克隆源码，可以键入：

```
$ git clone central.server/path/to/proj.git
```

做了改动之后，开发保存变更到本地：

```
$ git commit -a
```

更新到最新版本：

```
$ git pull
```

所有冲突应被处理，然后提交：

```
$ git commit -a
```

把本地改动捡入到中心仓库：

```
$ git push
```

如果主服务器由于其他开发的活动，有了新的变更，这个推送将会失败，开发者应该把最新版本更新至本地机器，解决合并冲突，然后重试。

为使用上面的 pull 和 push 命令，开发者必须有 SSH 访问权限。不过，通过键入以下命令，任何人都可以看到源码：

```
$ git clone git://central.server/path/to/proj.git
```

本地 git 协议和 HTTP 类似：并无安全验证，因此任何人都能拿到项目。因此，默认情况 git 协议禁止推操作。

封闭源码

闭源项目须避免执行 touch 命令，并确保你从未创建 'git-daemon-export-ok'文件。资源库不再可以通过 git 协议获取；只有那些有 SSH 访问权限的人才能看到。如果你所有的资源库都是封闭的，那也没必要运行运行 git 守护了，因为所有沟通都走 SSH。

裸仓库

之所以叫裸仓库是因为其没有工作目录；它只包含正常情况下隐藏在 '.git'子目录下的文件。换句话说，它维护项目历史，而且从不保存任何给定版本的快照。

裸仓库扮演的角色和中心版本控制系统中中心服务器的角色类似：你项目的中心。开发从其中克隆项目，捡入新近改动。典型地裸仓库存在一个服务器上，该服务器除了分散数据外并不担负额外的任务。开发活动发生在克隆的代码库上，因此中心仓库没有工作目录也行。

很多 Git 命令在裸仓库上失败，除非指定仓库路径到环境变量 'GIT_DIR'，或者指定 '--bare'选项。

推还是拽

为什么我们介绍了 push 命令，而不是依赖熟悉的 pull 命令？首先，在裸仓库上 pull 会失败：除非你必须 "fetch"，一个之后我们要讨论的命令。但即使我们在中心服务器上保持一个正常的仓库，拽些东西进去仍然很繁琐。我们不得不登陆服务器先，给 pull 命令我们要拽自机器的网络地址。防火墙会阻碍，并且首先如果我们没有到服务器的 shell 访问怎么办呢？

然而，除了这个案例，我们反对推进仓库，因为当目标有工作目录时，困惑随之而来。

简短来说，学习 Git 的时候，只在目标是裸仓库的时候使用 push，否则用 pull 的方式。

项目分叉

项目走歪了吗？或者认为你可以做得更好？那么在服务器上：

```
$ git clone git://main.server/path/to/files
```

之后告诉每个相关的人你服务器上项目的分支。

在之后的时间，你可以合并来自原先项目的改变，使用命令：

```
$ git pull
```

终极备份

需要制作很多散布在各处，并禁止篡改的冗余存档吗？ 如果你的项目有很多开发者参与，那你并不需要再做什么了。每份代码克隆都是一个有效备份。它不仅包含当前代码库状态，也同时包含了项目的整个历史。多谢哈希加密算法，如果任何人的克隆库被损坏，只要其与其他克隆库交互，损坏的代码就会被显示出来。

如果你的项目并不是那么流行，那就找尽可能多的伺服来存放克隆库吧。

真正的偏执狂应该总是把 HEAD 最近 20 字节的 SHA1 哈希值写到安全的地方。应该保证安全，而不是把它藏起来。比如，把它发布到报纸上就不错，因为对攻击者而言，更改每份报纸是很难的。

轻快多任务

比如你想并行开发多个功能。那么提交你的项目并运行：

```
$ git clone . /some/new/directory
```

Git 使用硬链接和文件共享来尽可能安全地创建克隆，因此它一眨眼就完成了，因此你现在可以并行操作两个没有相互依赖的功能。例如，你可以编辑一个克隆，同时编译另一个。感谢 hardlinking, 本地克隆比简单备份省时省地。

现在你可以同时工作在两个彼此独立的特性上。比如，你可以在编译一个克隆的时候编辑另一个克隆。任何时候，你都可以从其它克隆提交并拖拽变更。

```
$ git pull /the/other/clone HEAD
```

游击版本控制

你正做一个使用其他版本控制系统的项目，而你非常思念 Git？那么在你的工作目录初始化一个 Git 仓库：

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

然后克隆它：

```
$ git clone . /some/new/directory
```

并在这个目录工作，按你所想在使用 Git。过一会，一旦你想和其他每个人同步，在这种情况下，转到原来的目录，用其他的版本控制工具同步，并键入：

```
$ git add .
$ git commit -m "Sync with everyone else"
```

现在转到新目录运行：

```
$ git commit -a -m "Description of my changes"
$ git pull
```

把你的变更提交给他人的过程依赖于其他版本控制系统。这个新目录包含你的改动的文件。需要运行其他版本控制系统的命令来上载这些变更到中心仓库。

Subversion, 或许是最好的中心式版本控制系统，为无数项目所用。**git svn** 命令为 Subversion 仓库自动化了上面的操作，并且也可以用作 导出 Git 项目到 Subversion 仓库 的替代。

Mercurial

Mercurial 是一个类似的的版本控制系统，几乎可以和 Git 一起无缝工作。使用 ‘hg-git‘插件，一个 Mercurial 用户可以无损地往 Git 仓库推送，从 Git 仓库拖拽。

使用 Git 获得 ‘hg-git‘插件：

```
$ git clone git://github.com/schacon/hg-git.git
```

或使用 Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

不好意思，我没注意 Git 有类似的插件。因此, 我主张使用 Git 而不是 Mercurial 作为主资源库，即使你偏爱 Mercurial。使用 Mercurial 项目，通常一个自愿者维护一个平行的 Git 项目以适应 Git 用户，然而感谢 ‘hg-git‘插件，一个 Git 项目自动地适应 Mercurial 用户。

尽管该插件可以把一个 Mercurial 仓库转成一个 Git 仓库，通过推送一个空仓库来实现，这个差事交给 ‘hg-fast-export.sh‘脚本来做还是更容易些。这个脚本来自：

```
$ git clone git://repo.or.cz/fast-export.git
```

如需转化，只需在一个空目录运行：

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

注意，需要先将该脚本加入你的 ‘$PATH‘路径中，再执行以上命令行。

Bazaar

我们简略提一下 Bazaar，它毕竟是紧跟 Git 和 Mercurial 之后最流行的自由分布式版本控制系统。

Bazaar 有后来者的优势，它相对年轻些；它的设计者可以从前人的错误中学习，并且避免先行者在历史上犯过的错误。另外，它的开发人员对可移植性以及和与其它版本控制系统的互操作性也考虑周全。

一个 'bzr-git'插件让 Bazaar 用户在一定程度下可以工作在 Git 仓库。'tailor'程序转换 Bazaar 仓库到 Git 仓库，并且可以递增的方式做，要知道 'bzr-fast-export'只是在一次性转换性情况下工作良好。

我偏爱 Git 的原因

我起先选择 Git 是因为我听说它能管理不可想象地不可管理的 Linux 内核源码。我从来没觉得有离开的必要。Git 已经工作的很好了，并且我也没有被其瑕疵所困扰。因为我主要使用 Linux，其他平台上的问题与我无关。

还有，我偏爱 C 程序和 bash 脚本，以及诸如 Python 的可执行可脚本：其代码依赖性较低，并且我也沉迷于快速的执行时间。

我考虑过 Git 如何才能提高，甚至自己写过类似的工具，但只作为学术研究练练手。即使我会完成这个项目，我也无论如何会继续使用 Git，因为使用一个古里古怪的系统所获甚微。

自然地，你的需求和期望可能不同，并且你可能感觉使用另一个系统会好些。即便如此，使用 Git 你都不至于错太远。

分支巫术

即时分支合并是 Git 最给力的杀手锏。

问题：外部因素要求必须切换场景。不如说，在发布版本中突然蹦出个严重缺陷；某个特性完成的截至日期就要来临；在项目关键部分可以提供帮助的一个开发正打算离职。所有情况逼迫你停下所有手头工作，全力扑到到这个完全不同的任务上。

打断思维的连续性会使你的生产力大大降低，并且切换上下文越麻烦，潜在的损失也越大。如果使用中心版本控制，我们就必须从中心服务器下载一个新的工作拷贝。分布式系统的情况就好多了，因为我们能够在本地克隆所需要的版本。

但是克隆仍然需要拷贝整个工作目录，还有直到给定点的整个历史记录。尽管 Git 使用文件共享和硬链接减少了花费，项目文件自身还是必须在新的工作目录里重建。

方案：Git 有一个更好的工具对付这种情况，比克隆快多了而且节省空间：**git branch** 。

使用这个魔咒，目录里的文件突然从一个版本变到另一个。除了只是在历史记录里上跳下窜外，这个转换还可以做更多。你的文件可以从上一个发布版变到实验版本到当前开发版本到你朋友的版本等等。

老板键

曾经玩过那样的游戏吗？按一个键（“老板键”），屏幕立即显示一个电子表格或别的？那么如果老板走进办公室，而你正在玩游戏，就可以快速将游戏藏起来。

在某个目录：

```
$ echo "I'm smarter than my boss" > myfile.txt
$ git init
$ git add .
$ git commit -m "Initial commit"
```

我们已经创建了一个 Git 仓库，该仓库保存了一个包含特定信息的文件。现在我们键入：

```
$ git checkout -b boss  # 之后似乎没啥变化
$ echo "My boss is smarter than me" > myfile.txt
$ git commit -a -m "Another commit"
```

看起来我们刚刚只是覆盖了原来的文件并提交了它。但这是个错觉。键入：

```
$ git checkout master  # 切到文件的原先版本
```

嘿真快！这个文件就恢复了。并且如果老板决定窥视这个目录，键入：

```
$ git checkout boss  # 切到适合老板看的版本
```

你可以在两个版本之间相切多少次就切多少次，而且每个版本都可以独立提交。

肮脏的工作

比如你正在开发某个特性，并且由于某种原因，你需要回退三个版本，临时加进几行打印语句来，来看看一些东西是如何工作的。那么：

```
$ git commit -a
$ git checkout HEAD~3
```

现在你可以到处加丑陋的临时代码。你甚至可以提交这些改动。当你做完的时候，

```
$ git checkout master
```

来返回到你原来的工作。看，所有未提交变更都结转了。

如果你后来想保存临时变更怎么办？简单：

```
$ git checkout -b dirty
```

只要在切换到主分支之前提交就可以了。无论你什么时候想回到脏的变更，只需键入：

```
$ git checkout dirty
```

我们在前面章节讨论加载旧状态的时候，曾经接触过这个命令。最终我们把故事说全：文件改变成请求的状态，但我们必须离开主分支。从现在开始的任何提交都会将你的文件提交到另一条不同的路，这个路可以之后命名。

换一个说法，在 checkout 一个旧状态之后，Git 自动把你放到一个新的，未命名的分支，这个分支可以使用 **git checkout -b** 来命名和保存。

快速修订

你正在做某件事的当间，被告知先停所有的事情，去修理一个新近发现的臭虫，这个臭虫在提交‘1b6d...‘：

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

那么一旦你修正了这个臭虫：

```
$ git commit -a -m "Bug fixed"
$ git checkout master
```

并可以继续你原来的任务。你甚至可以“合并”到最新修订：

```
$ git merge fixes
```

合并

一些版本控制系统，创建分支很容易，但把分支合并回来很难。使用 Git，合并简直是家常便饭，以至于甚至你可能对其发生没有察觉。

我们很久之前就遇到合并了。**pull** 命令取出提交并合并它们到你的当前分支。如果你没有本地变更，那这个合并就是一个“快进”，相当于中心式版本控制系统里的一个弱化的获取最新版本操作。但如有本地变更，Git 将自动合并，并报告任何冲突。

通常，一个提交只有一个“父提交”，也叫前一个提交。合并分支到一起产生一个至少有两个父的提交。这就引出了问题：HEAD~10 真正指哪个提交？一个提交可能有多个父，那我们跟哪个呢？

原来这个表示每次选择第一个父。这是可取的，因为在合并时候当前分支成了第一个父；多数情况下我们只关注我们在当前分支都改了什么，而不是从其他分支合并来的变更。

你可以用插入符号来特别指定父。比如，显示来自第二个父的日志：

```
$ git log HEAD^2
```

你可以忽略数字以指代第一个父。比如，显示与第一个父的差别：

```
$ git diff HEAD^
```

你可以结合其他类型使用这个记号。比如：

```
$ git checkout 1b6d^^2~10 -b ancient
```

开始一个新分支“ancient”，表示第一个父的第二个父的倒数第十次提交的状态。

不间断工作流

经常在硬件项目里，计划的第二步必须等第一步完成才能开始。待修的汽车傻等在车库里，直到特定的零件从工厂运来。一个原型在其可以构建之前，可能要苦等芯片成型。

软件项目可能也类似。新功能的第二部分不得不等待，直到第一部分发布并通过测试。一些项目要求你的代码需要审批才能接受，因此你可能需要等待第一部分得到批准，才能开始第二部分。

多亏了无痛分支合并，我们可以不必遵循这些规则，在第一部分正式准备好前开始第二部分的工作。假设你已经将第一部分提交并发去审批，比如说你现在在主分支。那么分岔：

```
$ git checkout -b part2
```

接下来，做第二部分，随时可以提交变更。只要是人就可能犯错误，经常你将回到第一部分在修修补补。如果你非常幸运，或者超级棒，你可能不必做这几行：

```
$ git checkout master  # 回到第一部分
$ 修复问题
$ git commit -a        # 提交变更
$ git checkout part2   # 回到第二部分
$ git merge master     # 合并这些改动
```

最终，第一部分获得批准：

```
$ git checkout master  # 回到第一部分
$ submit files         # 对世界发布
$ git merge part2      # 合并第二部分
$ git branch -d part2  # 删除分支“part2”
```

现在你再次处在主分支，第二部分的代码也在工作目录。

很容易扩展这个技巧，应用到任意数目的部分。它也很容易追溯分支：假如你很晚才意识到你本应在 7 次提交前就创建分支。那么键入：

```
$ git branch -m master part2  # 重命名“master”分支为“part2”。
$ git branch master HEAD~7    # 以七次前提交建一个新的“master”。
```

分支 master 只有第一部分内容，其他内容在分支 part2 。我们现在后一个分支；我们创建了 master 分支还没有切换过去，因为我们想继续在分支 part2 上工作。这是不寻常的。直到现在，我们已经在创建之后切换到分支，如：

```
$ git checkout HEAD~7 -b master  # 创建分支，并切换过去。
```

重组杂乱

或许你喜欢在同一个分支下完成工作的方方面面。你想为自己保留工作进度并希望其他人只能看到你仔细整理过后的提交。开启一对分支：

```
$ git branch sanitized    # 为干净提交创建分支
$ git checkout -b medley  # 创建并切换分支以进去工作
```

接下来，做任何事情：修臭虫，加特性，加临时代码，诸如此类，经常按这种方式提交。然后：

```
$ git checkout sanitized
$ git cherry-pick medley^^
```

应用分支“medley”的祖父提交到分支“sanitized”。通过合适的挑选（像选樱桃那样）你可以构建一个只包含成熟代码的分支，而且相关的提交也组织在一起。

管理分支

列出所有分支：

```
$ git branch
```

默认你从叫“master”的分支开始。一些人主张别碰“master”分支，而是创建你自己版本的新分支。

选项 **-d** 和 **-m** 允许你来删除和移动（重命名）分支。参见 **git help branch** 。

分支“master”是一个有用的惯例。其他人可能假定你的仓库有一个叫这个名字的分支，并且该分支包含你项目的官方版本。尽管你可以重命名或抹杀“master”分支，你最好还是尊重这个约定。

临时分支

很快你会发现你经常会因为一些相似的原因创建短期的分支：每个其它分支只是为了保存当前状态，那样你就可以直接跳到较老状态以修复高优先级的臭虫之类。

可以和电视的换台做类比，临时切到别的频道，来看看其它台那正放什么。但并不是简单地按几个按钮，你不得不创建，检出，合并，以及删除临时分支。幸运的是，Git 已经有了和电视机遥控器一样方便的快捷方式：

```
$ git stash
```

这个命令保存当前状态到一个临时的地方（一个隐藏的地方）并且恢复之前状态。你的工作目录看起来和你开始编辑之前一样，并且你可以修复臭虫，引入之前变更等。当你想回到隐藏状态的时候，键入：

```
$ git stash apply  # 你可能需要解决一些冲突
```

你可以有多个隐藏，并用不同的方式来操作他们。参见 **git help slash** 。也许你已经猜到，Git 维护在这个场景之后的分支以执行魔法技巧.

按你希望的方式工作

你可能犹疑于分支是否值得一试。毕竟，克隆也几乎一样快，并且你可以用 **cd** 来在彼此之间切换，而不是用 Git 深奥的命令。

考虑一下浏览器。为什么同时支持多标签和多窗口？因为允许两者同时接纳了多种风格的用户。一些用户喜欢只保持一个打开的窗口，然后用标签浏览多个网页。一些可能坚持另一个极端：任何地方都没有标签的多窗口。一些喜好处在两者之间。

分支类似于你工作目录的标签，克隆则类似于打开的浏览器新窗口。这些是本地操作很快，那为什么不试着找出最适合你的组合呢？Git 让你按你确实所希望的那样工作。

关于历史

Git 分布式本性使得历史可以轻易编辑。但你若篡改过去，需要小心：只重写你独自拥有的那部分。正如民族间会无休止的争论谁犯下了什么暴行一样，如果在另一个人的克隆里，历史版本与你的不同，当你们的树互操作时，你们会遇到一致性方面的问题。

一些开发人员强烈地感到历史应该永远不变，不好的部分也不变，所有都不变。另一些觉得代码树在向外发布之前，应该整得漂漂亮亮的。Git 同时支持两者的观点。像克隆，分支和合并一样，重写历史只是 Git 给你的另一强大功能，至于如何明智地使用它，那是你的事了。

我认错

刚提交，但你期望你输入的是一条不同的信息？那么键入：

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

来改变上一条信息。意识到你还忘记了加一个文件？运行 git add 来加，然后运行上面的命令。

希望在上次提交里包括多一点的改动？那么就做这些改动并运行：

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

更复杂情况

假设上面的问题再糟糕十倍，比如在漫长的时间里我们提交了一堆，但你不太喜欢他们的组织方式，而且一些提交信息需要重写。那么键入：

```
$ git rebase -i HEAD~10
```

则最后的 10 个提交会出现在你喜爱的 $EDITOR（即通过设置 EDITOR 环境变量决定使用哪个文本编辑器）。一个例子：

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

不像 git log，列表中的提交是旧的在前，新的在后。在上面的列表中，5c6eb73 是最老的提交，100834 是最新的提交。接下来可以在 $EDITOR 中：

- 通过删除行来移去提交。
- 通过对几行记录重新排序，来重组提交顺序。
- 替换 pick 命令：
  - edit 标记一个提交需要修订。

- `reword` 改变日志信息。
- `squash` 将一个提交与其前一个合并。
- `fixup` 将一个提交与其前一个合并，并丢弃日志信息。

比如，将第二个提交的 'pick'修改为 'squash': pick 5c6eb73 Added repo.or.cz link squash a311a64 Reordered analogies in "Work How You Want" pick 100834f Added push target to Makefile

保存退出，则 Git 会把 a311a64 合并进 5c6eb73。想象一下"挤压"动作，squash 的作用就是把一个提交"挤压"进前一个提交。

"挤压"的同时，Git 合并了两个提交的日志信息供编辑。比较一下 *fixup* 命令，其直接扔掉"挤压"后合并的日志信息。

如果你把一个提交标记为 *edit*，Git 会带你回到这个提交点，你可以修改当时的提交信息，甚至可以增加新的提交。修改完毕后执行:

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

直到遇到下一个 *edit* 标记点，Git 会回放所有遇到的提交。

也可以放弃修改:

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

这样尽早提交，经常提交: 你之后还可以用 rebase 来规整。

本地变更之后

你正在一个活跃的项目上工作。随着时间推移，你做了几个本地提交，然后你使用合并与官方版本同步。在你准备好提交到中心分支之前，这个循环会重复几次。

但现在你本地 Git 克隆掺杂了你的改动和官方改动。你更期望在变更列表里，你所有的变更能够连续列出。

这就是上面提到的 **git rebase** 所做的工作。在很多情况下你可以使用 **--onto** 标记以避免交互。

另外参见 **git help rebase** 以获取这个让人惊奇的命令更详细的例子。你可以拆分提交。你甚至可以重新组织一棵树的分支。

要小心的是，rebase 指令非常强悍，对于复杂的 rebase，最好首先使用 *git clone* 备份一下。

重写历史

偶尔，你需要做一些代码控制——好比从正式的照片中去除一些人一样——你需要从历史记录里面彻底的抹掉他们。例如，假设我们要发布一个项目，但由于一些原因，项目中的某个文件不能公开。或许我把我的信用卡号记录在了一个文本文件里，而我又意外的把它加入到了这个项目中。仅

仅删除这个文件是不够的，因为从别的提交记录中还是可以访问到这个文件。因此我们必须从所有的提交记录中彻底删除这个文件。

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

参见 **git help filter-branch** ，那里讨论了这个例子并给出一个更快的方法。一般地，**filter-branch** 允许你使用一个单一命令来大范围地更改历史。

此后，+.git/refs/original+ 目录描述操作之前的状态。检查命令 **filter-branch** 的确做了你想要做的，然后删除此目录，如果你想运行多次 filter-branch 命令。

最后，如果你想之后和修订过的版本交互的话，记得用你修订过的版本替换你的项目克隆。

制造历史

想把一个项目迁移到 Git 吗？如果这个项目是在用比较有名气的系统，那可以使用一些其他人已经写好的脚本，把整个项目历史记录导出来放到 Git 里。

否则，查一下 **git fast-import** ，这个命令会从一个特定格式的文本读入，从头来创建 Git 历史记录。通常可以用这个命令很快写一个脚本运行一次，一次迁移整个项目。

作为一个例子，粘贴以下所列到临时文件，比如/tmp/history：

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
```

```
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

之后从这个临时文件创建一个 Git 仓库，键入：

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

你可以从这个项目 checkout 出最新的版本，使用：

```
$ git checkout master .
```

命令 **git fast-export** 可将任意仓库转换成 **git fast-import** 格式，你可以研究其输出来写新的导出程序，或以人类可以理解的格式转移仓库。的确，这些命令可以通过只接受文本的渠道来发送仓库文件。

哪儿错了？

你刚刚发现程序里有一个功能出错了，而你十分确定几个月以前它运行的很正常。天啊！这个臭虫是从哪里冒出来的？要是那时候能按照开发的内容进行过测试该多好啊。

现在说这个已经太晚了。然而，即使你过去经常提交变更，Git 还是可以精确的找出问题所在：

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git 从历史记录中检出一个中间的状态。在这个状态上测试功能，如果还是有问题：

```
$ git bisect bad
```

如果可以工作了，则把”bad” 替换成”good”。Git 会再次帮你找到一个以确定的好版本和坏版本之间的状态，通过这种方式缩小范围。经过一系列的迭代，这种二分搜索会帮你找到导致这个错误的那次提交。一旦完成了问题定位的调查，你可以返回到原始状态，键入：

```
$ git bisect reset
```

不需要手工测试每一次改动，执行如下命令可以自动的完成上面的搜索：

```
$ git bisect run my_script
```

Git 使用指定命令（通常是一个一次性的脚本）的返回值来决定一次改动是否是正确的：命令退出时的代码 0 代表改动是正确的，125 代表要跳过对这次改动的检查，1 到 127 之间的其他数值代表改动是错误的。返回负数将会中断整个 bisect 的检查。

你还能做更多的事情：帮助文档解释了如何展示 bisects, 检查或重放 bisect 的日志, 并可以通过排除对已知正确改动的检查，得到更好的搜索速度。

谁让事情变糟了？

和其他许多版本控制系统一样，Git 也有一个”blame” 命令：

```
$ git blame bug.c
```

这个命令可以标注出一个指定的文件里每一行内容的最后修改者，和最后修改时间。但不像其他版本控制系统，Git 的这个操作是在线下完成的，它只需要从本地磁盘读取信息。

个人经验

在一个中心版本控制系统里，历史的更改是一个困难的操作，并且只有管理员才有权这么做。没有网络，克隆，分支和合并都没法做。像一些基本的操作如浏览历史，或提交变更也是如此。在一些系统里，用户使用网络连接仅仅是为了查看他们自己的变更，或打开文件进行编辑。

中心系统排斥离线工作，也需要更昂贵的网络设施，特别是当开发人员增多的时候。最重要的是，所有操作都一定程度变慢，一般在用户避免使用那些能不用则不用的高级命令时。在极端的情况下，即使是最基本的命令也会变慢。当用户必须运行缓慢的命令的时候，由于工作流被打断，生产力就会降低。

我有这方面的一手经验。Git 是我使用的第一个版本控制系统。我很快学会适应了它，并使用了它提供的许多功能。我简单地假设其他系统也是相似的：选择一个版本控制系统应该和选择一个编辑器或浏览器没啥两样。

在我之后被迫使用中心系统的时候，我被震惊到了。我那有些脆弱的网络没给 Git 带来大麻烦，但是当它需要像本地硬盘一样稳定工作的时候，它使开发变得困难重重。另外，我发现我自己有选择地避免使用特定的命令，以避免踏雷，这极大地影响了我，使我不能按照我喜欢的方式工作。

当我不得不运行一个速度缓慢的命令时，这种等待极大地破坏了我思绪的连续性。在等待服务器通讯完成的时候，我选择做其他的事情以度过这段时光，比如查看邮件或写其他的文档。当我返回我原先的工作场景的时候，这个命令早已结束，并且我还需要浪费时间试图记起我之前正在做什么。人类并不擅长场景间的切换。

还有一个有意思的大众悲剧效应：预料到网络拥挤，为了减少将来的等待时间，每个人将比以往消费更多的带宽在各种操作上。大家共同的努力结果加剧了拥挤，这等于是鼓励个人下次消费更多带宽以避免更长的等待时间。

多人 Git

我最初在一个私人项目上使用 Git，我是那个项目的唯一的开发者。在与 Git 分布式特性有关的命令中，我只用到了 **pull** 和 *clone*，以此即可在不同地方保持项目同步。

后来我想用 Git 发布我的代码，并且包括其他贡献者的变更。我不得不学习如何管理有来自世界各地的多个开发的项目，幸运的是，这是 Git 的长处，也可以说是其存在的理由。

我是谁？

每个提交都有一个作者姓名和电子信箱，这显示在 **git log** 里。Git 使用系统默认设定来填充这些信息。要设定这些信息，键入：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

去掉 global 选项设定则只对当前仓库生效。

Git 在 SSH, HTTP 上的使用

假设你有 ssh 访问权限，可以访问一个网页服务器，但上面并没有安装 Git。尽管比它的原生协议效率低，Git 也是可以通过 HTTP 来进行通信的。

在你的帐户下，下载，编译并安装 Git，并在你的网页目录里创建一个 Git 仓库：

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

对较老版本的 Git，只拷贝还不够，你还要运行：

```
$ chmod a+x hooks/post-update
```

现在你可以通过 SSH 从随便哪个克隆发布你的最新版本：

```
$ git push web.server:/path/to/proj.git master
```

而随便任何人都可以通过如下命令来取得你的项目：

```
$ git clone http://web.server/proj.git
```

Git 在随便什么上的使用

若想不依赖服务器，甚至无需网络连接来同步仓库？需要在紧急时期凑合一下？我们已经看过**git fast-export** 和 **git fast-import** 可以转换资源库到一个单一文件以及转回来。我们可以通过任何媒介，来来回回传送这些文件以同步 git 仓库。但一个更有效率的工具是 **git bundle** 。

发送者创建一个“文件包”：

```
$ git bundle create somefile HEAD
```

然后传输这个文件包，somefile ，给某个其他参与者：电子邮件，优盘，一个 **xxd** 打印品和一个 OCR 扫描仪，通过电话读字节，狼烟，等等。接收者通过键入如下命令从文件包获取提交：

```
$ git pull somefile
```

接收者甚至可以在一个空仓库做这个。不考虑大小，somefile 可以包含整个原先 git 仓库。

在较大的项目里，可以通过只打包其他仓库缺少的变更来消除存储浪费。例如，假设提交 “1b6d...” 是两个参与者共享的最近提交：

```
$ git bundle create somefile HEAD ^1b6d
```

如果提交频繁，人们可能很容易忘记刚发送了哪个提交。帮助页面建议使用标签解决这个问题。即，在你发了一个文件包后，键入：

```
$ git tag -f lastbundle HEAD
```

并创建较新文件包，使用：

```
$ git bundle create newbundle HEAD ^lastbundle
```

补丁：全球货币

补丁是变更的文本形式，易于计算机理解，人也类似。补丁可以通吃。你可以给开发者电邮一个补丁，不用管他们用的什么版本控制系统，只要对方可以读电子邮件，他们就能看到你的修改。类似的，在你这边，你只需要一个电子邮件帐号，而不必搭建一个在线的 Git 仓库。

回想一下第一章：

```
$ git diff 1b6d > my.patch
```

输出是一个补丁，可以粘贴到电子邮件里用以讨论。在一个 Git 仓库，键入：

```
$ git apply < my.patch
```

来打这个补丁。

在更正式些的设置里，当作者名字以及或许签名应该被记录下的时候，为过去某一刻生成补丁，键入：

```
$ git format-patch 1b6d
```

结果文件可以给 **git-send-email** 发送，或者手工发送。你也可以指定一个提交范围：

```
$ git format-patch 1b6d..HEAD^^
```

在接收一端，保存邮件到一个文件，然后键入：

```
$ git am < email.txt
```

这就打了补丁并创建了一个提交，其自动包含了诸如作者之类的信息。

使用浏览器的邮件客户端，在保存补丁为文件之前，你可能需要建一个按钮，看看邮件内容原来的原始形式。

对基于 mbox 的邮件客户端有些微不同，但如果你在使用的话，你可能是那种能轻易找出答案的那种人，不用读教程。

对不起，移走了

克隆一个仓库后，运行 **git push** 或 **git pull** 将自动推到或从原先的仓库 URL 拉出新内容。Git 如何做这个呢？秘密在于同克隆一起创建的配置选项里面。让我们看一下：

```
$ git config --list
```

选项 `remote.origin.url` 控制仓库的 URL 源；“origin” 是给源仓库的昵称。和 “master” 分支的惯例一样，我们可以改变或删除这个昵称，但通常没有理由这么做。

如果原先仓库移走，我们可以更新 URL，通过：

```
$ git config remote.origin.url git://new.url/proj.git
```

选项 `branch.master.merge` 指定 **git pull** 里的默认远端分支。在初始克隆的时候，它被设为原仓库的当前分支，因此即使原仓库之后挪到一个不同的分支，后来的 pull 也将忠实地跟着原来的分支。

这个选项只使用我们初次克隆的仓库，它的值记录在选项 `branch.master.remote` 。如果我们从其他仓库拉入，我们必须显示指定我们想要哪个分支：

```
$ git pull git://example.com/other.git master
```

以上也解释了为什么我们较早一些 push 和 pull 的例子没有参数。

远端分支

当你克隆了一个仓库，你也克隆了它的所有分支。你或许没有注意到这点，因为 Git 将它们隐藏起来了：你必须明确地要求。这使得远端仓库里的分支不至于干扰你的分支，也使 Git 对初学者稍稍容易些。

列出远端分支，使用：

```
$ git branch -r
```

你应该看到类似：

```
origin/HEAD
origin/master
origin/experimental
```

这显示了远端仓库的分支和 HEAD，可以用在常用的 Git 命令里。例如，假设你已经做了很多提交，并希望和最后取到的版本比较一下。你可以搜索适当的 SHA1 哈希值，但使用下面命令更容易些：

```
$ git diff origin/HEAD
```

或你可以看看 “experimental” 分支都有啥：

```
$ git log origin/experimental
```

多远端

假设另两个开发在同一个项目上工作，我们希望保持两个标签。我们可以同时跟踪多个仓库：

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

现在我们已经合并到第二个仓库的一个分支，并且我们已容易访问所有仓库的所有分支。

```
$ git diff origin/experimental^ other/some_branch~5
```

但如果为了不影响自己的工作，我们只想比较他们的变更怎么办呢？换句话说，我们想检查一下他们的分支，又不使他们的变更入侵我们的工作目录。这里我们并不要运行 pull 命令，而是运行：

```
$ git fetch        # Fetch from origin, the default.
$ git fetch other  # Fetch from the second programmer.
```

这只是获取历史。尽管工作目录维持不变，我们可以参考任何仓库的任何分支，使用一个 Git 命令，因为我们现在有一个本地拷贝。

回想一下，在幕后，一个 pull 是简单地一个 **fetch** 然后 **merge** 。通常，我们 **pull** 因为我们想在获取后合并最近提交；这个情况是一个值得注意的例外。

关于如何去除远端仓库，如何忽略特定分支等更多，参见 **git help remote** 。

我的喜好

对我手头的项目，我喜欢贡献者去准备仓库，这样我可以从其中拉。一些 Git 伺服让你点一个按钮，拥有自己的分叉项目。

在我获取一个树之后，我运行 Git 命令去浏览并检查这些变更，理想情况下这些变更组织良好，描述良好。我合并这些变更，也或许做些编辑。直到满意，我才把变更推入主资源库。

尽管我不经常收到贡献，我相信这个方法扩展性良好。参见 这篇来自 Linus Torvalds 的博客

呆在 Git 的世界里比补丁文件稍更方便，因为不用我将补丁转换到 Git 提交。更进一步，Git 处理诸如作者姓名和信箱地址的细节，还有时间和日期，以及要求作者描述他们的提交。

Git 大师技

到现在，你应该有能力查阅 **git help** 页，并理解几乎所有东西。然而，查明解决特定问题需要的确切命令可能是乏味的。或许我可以省你点功夫：以下是我过去曾经用到的一些技巧。

源码发布

就我的项目而言，Git 完全跟踪了我想打包并发布给用户的文件。如需创建一个源码包，我会运行：

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

提交变更

对特定项目而言，告诉 Git 你增加，删除和重命名了一些文件很麻烦。而键入如下命令会容易的多：

```
$ git add .
$ git add -u
```

Git 将查找当前目录的文件并计算出所有更改过的内容。除了第二个 add 命令，如果你也打算同时提交，则可以运行 'git commit -a'。关于如何指定应被忽略的文件，参见 **git help ignore** 。

你也可以用一行命令完成以上任务：

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

这里 **-z** 和 **-0** 选项可以消除包含特殊字符的文件名引起的不良副作用。注意这个命令也会添加本应被忽略的文件，这时你可能需要加上 `-x` 或 `-X` 选项。

我的提交太大了！

是不是忽略提交太久了？一直痴迷于写代码，直到现在才想起有源码控制工具这回事？或者提交一系列不相关的变更，因为你更喜欢这么做？

别担心，运行：

```
$ git add -p
```

对你做过的每次修改，Git 都将展示变动过的代码，并询问该变动是否应是下一次提交的一部分。回答“y”或者“n”。当然也有其他选项，比如延迟决定；键入“？”来学习更多。

一旦你满意，键入

```
$ git commit
```

来精确地提交你所选择的变更（阶段变更）。注意请勿加上 **-a** 选项，否则 Git 将提交所有修改。

如果你修改了许多地方的许多文件怎么办？一个一个地查看这些变更会令人沮丧，心态麻木。这种情况下，使用 **git add -i** ，它的界面不是很直观，但更灵活。敲几个键，你可以一次决定阶段或非阶段性提交几个文件，或查看并只选择特定文件的变更。作为另一种选择，你还可以运行 **git commit --interactive** ，这个命令会在你操作完后自动进行提交。

索引：Git 的中转区域

当目前为止，我们一直在忽略 Git 著名的“索引”概念，但现在我们必须面对它，以解释上面发生的事情。索引是一个临时中转区。Git 很少在你的项目和它的历史之间直接倒腾数据。通常，Git 先写数据到索引，然后拷贝索引中的数据到最终目的地。

例如，**commit -a** 实际上是一个两步过程。第一步把每个追踪文件当前状态的快照放到索引中。第二步永久记录索引中的快照。没有 **-a** 的提交只执行第二步，并且只在运行不知何故改变索引的命令才有意义，比如 **git add** 。

通常我们可以忽略索引并假装从历史中直接读并直接写。在这个情况下，我们希望更好地控制，因此我们操作索引。我们放我们变更的一些的快照到索引中，而不是所有的，然后永久地记录这个被小心操纵的快照。

别丢了你的 HEAD

HEAD 好似一个游标，通常指向最新提交，随最新提交向前移动。一些 Git 命令可让你来移动它。例如：

```
$ git reset HEAD~3
```

这将立即将 HEAD 向回移动三个提交。这样所有 Git 命令都表现得好似你没有做那最后三个提交，然而你的文件保持在现在的状态。具体应用参见帮助页。

但如何回到将来呢？过去的提交对将来一无所知。

如果你有原先 Head 的 SHA1 值，那么：

```
$ git reset 1b6d
```

但假设你从来没有记下呢？别担心，在这些命令里面，Git 会将原先的 Head 保存为一个叫做 ORIG_HEAD 的标记，你可以安全体面的返回那里：

```
$ git reset ORIG_HEAD
```

HEAD 捕猎

或许 ORIG_HEAD 还不够；或许你刚认识到你犯了个历史性的错误，你需要回到一个早已忘记分支上一个远古的提交。

默认的，Git 将会将一个提交保存至少两星期，即使你命令 Git 摧毁该提交所在的分支。难点是找到相应的哈希值。你可以查看在.git/objects 里所有的哈希值并尝试找到你期望的提交。但这里有一个更简单的办法。

Git 把算出的提交哈希值记录在 ".git/logs"。这个子目录引用包括所有分支上所有活动的历史，同时文件 HEAD 显示它曾经有过的所有哈希值。后者可用来发现分支上一些不小心丢掉提交的哈希值。

命令 reflog 为访问这些日志文件提供了友好的接口，可以试试

```
$ git reflog
```

而不是从 reflog 拷贝粘贴哈希值，试一下：

```
$ git checkout "@{10 minutes ago}"
```

或者捡出后五次访问过的提交，通过：

```
$ git checkout "@{5}"
```

更多内容参见 **git help rev-parse** 的 "Specifying Revisions" 部分。

你或许期望去为已删除的提交设置一个更长的保存周期。例如：

```
$ git config gc.pruneexpire "30 days"
```

意思是一个被删除的提交会在删除 30 天并运行 **git gc** 以后，被永久丢弃。

你或许还想关掉 **git gc** 的自动运行：

```
$ git config gc.auto 0
```

在这种情况下提交将只在你手工运行 **git gc** 的情况下才永久删除。

基于 Git 构建

依照真正的 UNIX 风格设计，Git 允许其易于用作其他程序的底层组件，比如图形界面，Web 界面，可选择的命令行界面，补丁管理工具，导入和转换工具等等。实际上，一些 Git 命令它们自己就是站在巨人肩膀上的脚本。通过一点修补，你可以定制 Git 适应你的偏好。

一个简单的技巧是，用 Git 内建 alias 命令来缩短你最常使用命令：

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # 显示当前别名
alias.co checkout
$ git co foo                              # 和"git checkout foo"一样
```

另一个技巧，在提示符或窗口标题上打印当前分支。调用：

```
$ git symbolic-ref HEAD
```

显示当前分支名。在实际应用中，你可能最想去掉 "refs/heads/" 并忽略错误：

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

子目录 `contrib` 是一个基于 Git 工具的宝库。它们中的一些命令不时的会被提升为官方命令。在 Debian 和 Ubuntu，这个目录位于 `/usr/share/doc/git-core/contrib` 。

一个受欢迎的居民是 `workdir/git-new-workdir` 。通过聪明的符号链接，这个脚本创建一个新的工作目录，其历史与原来的仓库共享：

```
$ git-new-workdir an/existing/repo new/directory
```

这个新的目录和其中的文件可被视为一个克隆，除了历史是共享的，两者的树会自动保持同步，而不必合并，推入或拉出。

大胆的特技

最近以来，Git 努力使用户因意外而销毁数据变得更困难。但如若你知道你在做什么，你可以突破为通用命令所设的保障措施。

Checkout：未提交的变更会导致捡出失败。销毁你的变更，并无论如何都 checkout 一个指定的提交，使用强制标记：

```
$ git checkout -f HEAD^
```

另外，如果你为捡出指定特别路径，那就没有安全检查了。提供的路径将被不加提示地覆盖。如你使用这种方式的检出，要小心。

Reset：如有未提交变更重置也会失败。强制其通过，运行：

```
$ git reset --hard 1b6d
```

Branch：引起变更丢失的分支删除会失败。强制删除，键入：

```
$ git branch -D dead_branch  # instead of -d
```

类似，通过移动试图覆盖分支，如果随之而来有数据丢失，那么覆盖也会失败。强制移动分支，键入：

```
$ git branch -M source target  # 而不是 -m
```

不像 checkout 和重置，这两个命令将延迟数据销毁。这个变更仍然存储在.git 的子目录里，并且可以通过恢复.git/logs 里的相应哈希值获取（参见上面上面 "HEAD 猎捕"）。默认情况下，这些数据会保存至少两星期。

Clean：一些 Git 命令拒绝执行，因为它们担心会重装未纳入管理的文件。如果你确信所有未纳入管理的文件都是消耗品，那就无情地删除它们，使用：

```
$ git clean -f -d
```

下次，那个讨厌的命令就会工作！

阻止坏提交

愚蠢的错误会污染我的代码库。最可怕的是由于忘记 **git add** 而引起的文件丢失。较小的错误则是行末追加空格而引发合并冲突：尽管危害少，我希望这些永远不要出现在公开记录里。

不过我购买了傻瓜保险，通过使用一个 __ 钩子 __ 来提醒我这些问题：

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # 对旧版本Git，先运行chmod +x
```

现在如果 Git 检测到无用的空格或未解决的合并冲突，它将放弃合并。

对本文档，我最终添加以下到 **pre-commit** 钩子的前面，来防止缺魂儿的事：

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

一部分其他的 Git 操作也支持钩子；参见 **git help hooks** 。我们早先激活了作为例子的 **post-update** 钩子，当讨论基于 HTTP 的 Git 的时候。无论 head 何时移动，这个钩子

都会运行。例子中的脚本 post-update 会在 Git 面对并不知晓的传输协议，诸如 HTTP 时，更新自己的资料库，以确保它有能力交换所需的文件。

揭开面纱

我们揭开 Git 神秘面纱，往里瞧瞧它是如何创造奇迹的。我会跳过细节，若要更深入的了解 Git 工作原理，可参见 用户手册。

大象无形

Git 怎么这么谦逊寡言呢？除了偶尔提交和合并外，你可以如常工作，就像不知道版本控制系统存在一样。那就是，直到你需要它，并且感到时间合适的时候以外，Git 都只是默默在后台看顾着你。

其他版本控制系统强迫你与繁文缛节和官僚主义不断斗争。文件的权限可能是只读的，除非你明确地告诉中心服务器哪些文件你打算编辑。即使最基本的命令，随着用户数目的增多，也会慢的像爬一样。中心服务器可能正跟踪什么人，什么时候 check out 了什么代码。当网络连接断了的时候，你就遭殃了。开发人员不断地与这些版本控制系统的种种限制作斗争。一旦网络或中心服务器瘫痪，工作就嘎然而止。

与之相反，Git 简单地在你工作目录下的`.git`目录保存你项目的历史。这是你自己的历史拷贝，因此你可以保持离线，直到你想和他人沟通为止。你拥有你的文件命运完全的控制权，因为 Git 可以轻易在任何时候从.git'重建一个曾经保存过的状态。

数据完整性

很多人把加密和保持信息机密关联起来，但一个同等重要的目标是保证信息安全。合理使用哈希加密功能可以防止无意或有意的数据损坏行为。

一个 SHA1 哈希值可被认为是一个唯一的 160 位 ID 数，用它可以唯一标识你一生中遇到的每个字节串。实际上不止如此：每个字节串可供任何人用好多辈子。

对一个文件而言，其整体内容的哈希值可以被看作这个文件的唯一标识 ID 数。

因为一个 SHA1 哈希值本身也是一个字节串，我们可以哈希包括其他哈希值的字节串。这个简单的观察出奇地有用：查看“哈希链”。我们之后会看 Git 如何利用这一点来高效地保证数据完整性。

简言之，Git 把数据保存在 '.git/objects'子目录，那里看不到正常文件名，相反你只看到 ID。通过用 ID 作为文件名，加上一些文件锁和时间戳技巧，Git 把任意一个原始的文件系统转化为一个高效而稳定的数据库。

智能

Git 是如何知道你重命名了一个文件，即使你从来没有明确提及这个事实？当然，你或许是运行了 **git mv** ，但这个命令和 **git add** 紧随 **git rm** 是完全一样的。

Git 启发式地找出相连版本之间的重命名和拷贝。实际上，它能检测文件之间代码块的移动或拷贝！尽管它不能覆盖所有的情况，但它已经做的很好了，并且这个功能也总在改进中。如果它在你那儿不工作的话，可以尝试打开开销更高的拷贝检测选项，并考虑升级。

索引

对每个加入库中管理的文件，Git 都会在一个名为 “index” 的文件里记录统计信息，诸如大小，创建时间和最后修改时间。为了确定文件是否被更改，Git 会将当前统计信息同那些在索引里的统计信息对比。如果一致，那 Git 就跳过该文件。

因为统计信息的调用比读文件内容快的很多，如果你仅仅编辑了少数几个文件，Git 几乎不需要什么时间就能更新他们的统计信息。

我们前面讲过索引是一个中转区。为什么一堆文件的统计数据是一个中转区？因为添加命令将文件放到 Git 的数据库并更新它们的统计信息，而无参数的提交命令将只基于统计信息和已经在数据库里的文件来创建一个全新的提交。

Git 的源起

这个 Linux 内核邮件列表帖子 描述了导致 Git 诞生的一系列事件。对 Git 史学家而言，整个讨论线是一个令人着迷的历史探究过程。

对象数据库

你数据的每个版本都保存在 “对象数据库” 里，其位于子目录.git/objects`内；其他位于.git/‘的较少数据：索引，分支名，标签，配置选项，日志，头提交的当前位置等。对象数据库朴素而优雅，是 Git 的力量之源。

‘.git/objects‘里的每个文件是一个对象。有 3 种对象跟我们有关：“blob” 对象，“tree” 对象，和 “commit” 对象。

Blob 对象

首先来一个小把戏。选择一个文件名，任意文件名。在一个空目录：

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

你将看到 .git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d 。

我如何在不知道文件名的情况下知道这个？这是因为以下内容的 SHA1 哈希值：

```
"blob" SP "6" NUL "sweet" LF
```

是 aa823728ea7d592acc69b36875a482cdf3fd5c8d，这里 SP 是一个空格，NUL 是一个 0 字节，LF 是一个换行符。你可以验证这一点，键入：

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git 基于“内容寻址”：文件并不按它们的文件名存储，而是按它们包含内容的哈希值，在一个叫“blob 对象”的文件里。我们可以把文件内容的哈希值看作一个唯一 ID，这样在某种意义上我们通过他们内容放置文件。开始的“blob 6”只是一个包含对象类型与其长度的头；它简化了内部存储。

这样我可以轻易预言你所看到的输出：文件名是无关的：只有里面的内容被用作构建 blob 对象。

你可能想知道对相同的文件会发生什么。试图填加一个你文件的拷贝，什么文件名都行。在 .git/objects 的内容保持不变，不管你加了多少。Git 都只存储一次数据。

顺便说一句，在 .git/objects 里的文件用 zlib 压缩，因此你不应该直接查看他们。可以通过zpipe -d 管道，或者键入：

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

这样可以漂亮地打印出给定的对象。注意，上面的 cat-file 命令中，aa 是目录名。

Tree 对象

但文件名在哪？它们必定在某个阶段保存在某个地方。Git 在提交时得到文件名：

```
$ git commit  # 输入一些信息。
$ find .git/objects -type f
```

你应看到 3 个对象。这次我不能告诉你这两个新文件是什么，因为它部分依赖你选择的文件名。我继续进行，假设你选了“rose”。如果你没有，你可以重写历史以让它看起来像似你做了：

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

现在你应看到文件 .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9，因为这是以下内容的 SHA1 哈希值：

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

通过键入以下命令来检查这个文件真的包含上面内容：

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

使用 zpipe，验证哈希值是容易的：

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

与查看文件相比，哈希值验证更轻巧一些，因为其输出不包含原始未压缩文件。

这里的输出是一个“tree”对象：一组包含文件类型，文件名和哈希值的数据。在我们的例子里，文件类型是 100644，这意味着“rose”是一个一般文件，并且哈希值指 blob 对象，包含“rose”的内容。其他可能文件类型有可执行，链接或者目录。在最后一个例子里，哈希值指向一个 tree 对象。

在一些过渡性的分支，你会有一些你不再需要的老的对象，尽管在宽限过期之后，它们会被自动清除，现在我们还是将其删除，以使我们比较容易跟上这个示范的例子。

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

在真实项目里你通常应该避免像这样的命令，因为你在破坏备份。如果你期望一个干净的仓库，通常最好做一个新的克隆。还有，直接操作 `.git` 时一定要小心：如果 Git 命令同时也在运行会怎样，或者突然停电？一般，引用应由 **git update-ref -d** 删除，尽管通常手工删除 `refs/original` 也是安全的。

Commit 对象

我们已经解释了三个对象中的两个。第三个是“commit”对象。其内容依赖于提交信息以及其创建的日期和时间。为满足这里我们所需的，我们不得不调整一下：

```
$ git commit --amend -m Shakespeare  # 改提交信息
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"'  # Rig timestamps and authors.
$ find .git/objects -type f
```

你现在应看到 `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` 是下列内容的 SHA1 哈希值：

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

和前面一样，你可以运行 zpipe 或者 cat-file 来自己看。

这是第一个提交，因此没有父提交，但之后的提交将总有至少一行，指定一个父提交。

没那么神

Git 的秘密似乎太简单。看起来似乎你可以整合几个 shell 脚本，加几行 C 代码来弄起来，也就几个小时的事：一个基本文件操作和 SHA1 哈希化的混杂，用锁文件装饰一下，文件同步保证健壮性。实际上，这准确描述了 Git 的最早期版本。尽管如此，除了巧妙地打包以节省空间，巧妙

地索引以省时间，我们现在知道 Git 如何灵巧地改造文件系统，使其成为一个完美的版本控制数据库。

例如，如果对象数据库里的任何一个文件由于硬盘错误损毁，那么其哈希值将不再匹配，这个错误会报告给我们。通过哈希化其他对象的哈希值，我们在所有层面维护数据完整性。Commit 对象是原子性的，也就是说，一个提交永远不会部分地记录变更：在我们已经存储所有关于 tree 对象，blob 对象和父 commit 对象之后，我们才可以计算提交的的哈希值并将其存储在数据库，对象数据库不受诸如停电之类的意外中断影响。

我们打败了即使是最狡猾的对手。假设有人试图悄悄修改一个项目里一个远古版本文件的内容，为使对象据库看起来健康，他们也必须修改相应 blob 对象的哈希值，既然它现在是一个不同的字节串。这意味着他们将不得不引用这个文件的 tree 对象的哈希值，并反过来改变所有与这个 tree 相关的 commit 对象的哈希值，还要加上这些提交所有后裔的哈希值。这暗示官方 head 的哈希值与这个坏仓库不同。通过跟踪不匹配哈希值线索，我们可以查明残缺文件，以及第一个被破坏的提交。

总之，只要 20 个字节代表最后一次的提交是安全的，我们就将不可能篡改一个 Git 仓库。

那么 Git 的著名功能怎样实现的呢？分支？合并？标签？这些都是单纯的细节。当前 head 保存在文件`.git /HEAD`，其中包含了一个 commit 对象的哈希值。该哈希值在运行提交以及其他命令时更新。分支几乎一样：它们是保存在 `.git/refs/heads` 的文件。标签也是：它们住在 `.git/refs/tags`，但它们由一套不同的命令更新。

附录 A: Git 的缺点

有一些 Git 的问题，我已经藏在毯子下面了。有些可以通过脚本或回调方法轻易地解决，有些需要重组或重定义项目，少数剩下的烦恼，还只能等待。或者更好地，你可以加入 Git 项目来帮忙。

SHA1 的弱点

随着时间的推移，密码学家发现越来越多的 SHA1 的弱点。人们发现，对资源雄厚的组织而言，找到哈希冲突是可能的。在几年内，或许甚至一个一般的 PC 也将有足够计算能力悄悄摧毁一个 Git 仓库。

希望在进一步研究出摧毁 SHA1 的方法之前，Git 能迁移到一个更好的哈希算法。

微软 Windows

Git 在微软 Windows 上可能有些繁琐：

- Cygwin，这是一个 Windows 下的类 Linux 的环境，其包含一个 一个 Git 在 Windows 下的移植.
- 基于 MSys 的 Git 是另一个，要求最小运行时支持，不过一些命令不能马上工作。

不相关的文件

如果你的项目非常大，包含很多不相关的文件，而且目录树在不断变更，Git 可能比其他系统更不可靠，因为独立的文件是不被跟踪的。Git 跟踪整个项目的变更，这通常才是有益的。

一个方案是将你的项目拆成小块，每个都由相关文件组成。如果你仍然希望在同一个资源库里保存所有内容的话，可以使用 **git submodule** 。

谁在编辑什么？

一些版本控制系统在编辑前会强迫你显示和用某个方法标记一个文件。尽管这种要求很烦人，尤其是需要和中心服务器通讯时。不过它还是有以下两个好处的：

1. 速度比较快，因为只有被标记的文件需要检查。
2. 通过查询在中心服务器谁把这个文件标记为编辑状态，可以知道谁正在这个文件上工作。

使用适当的脚本，你也可以使 Git 达到同样的效果。但这要求程序员协同工作，当他编辑一个文件的时候还要运行特定的脚本。

文件历史

因为 Git 记录的是项目范围内的变更，重造单一文件的变更历史比其他跟踪单一文件的版本控制系统要稍微麻烦些。

好在麻烦还不大，也是值得的，因为 Git 其他的操作难以置信地高效。例如，‘git checkout‘比‘cp -a‘都快，而且项目范围的 delta 压缩也比基于文件的 delta 集合的做法好多了。

初始克隆

当一个项目历史很长后，与在其他版本系统里的检出代码相比，创建一个克隆的开销会大的多。

长远来看，开始付出的代价还是值得付出的，因为大多将来的操作将由此变得很快，并可以离线完成。然而，在一些情况下，使用 ‘--depth‘创建一个浅层克隆比较划算些。这种克隆初始化的更快，但得到克隆的功能有所削减。

不稳定的项目

由变更的大小来决定写入的速度快慢是 Git 的特性。一般人做了小的改动就会提交新版本：这里修正一行错误，那里加一个新功能，或者修改注释等等。但如果你的文件在相邻版本之间存在极大的差异，那每次提交时，你的历史记录会随整个项目的大小增长。

任何版本控制系统对此都束手无策，但普通的 Git 用户将面对更多资源的消耗，因为一般来说，历史记录也会被克隆。

应该检查一下变更巨大的原因：或许文件格式需要改变一下。小修改应该仅仅导致几个文件的细小改动。

或许，数据库或备份/打包方案才是正选，而不是版本控制系统。例如，版本控制就不适宜用来管理网络摄像头周期性拍下的照片。

如果这些文件实在需要不断更改，他们实在需要版本控制，一个可能的办法是以中心的方式使用 Git。可以创建浅克隆，这样检出的较少，也没有项目的历史记录。当然，很多 Git 工具就不能用了，并且修复必须以补丁的形式提交。这也许还不错，因为似乎没人需要大幅度变化的不稳定文件历史。

另一个例子是基于固件的项目，因为要用到巨大的二进制文件形式。用户对固件文件的变化历史没有兴趣，更新的压缩比很低，因此固件修订将使仓库无谓的变大。

这种情况下，源码应该保存在一个 Git 仓库里，但二进制文件应该单独保存。为了简化问题，应该发布一个脚本，使用 Git 克隆源码，并对固件只做同步或 Git 浅层克隆。

全局计数器

一些中心版本控制系统都会维护一个正整数，当一个新提交被接受的时候这个整数就增长。Git 则是通过哈希值来记录所有变更，这在大多数情况下都工作的不错。

但一些人喜欢使用整数的方法。幸运的是，很容易就可以写个脚本，这样每次更新，中心 Git 仓库就增大这个整数，或使用 tag 的方式，把最新提交的哈希值与这个整数关联起来。

每个克隆都可以维护这么个计数器，但这或许没什么用，因为只有中心仓库以及它的计数器对每个人才有意义。

空子目录

空子目录不可加入管理。但可以通过创建一个空文件以绕过这个问题。

Git 的当前实现，而不是它的设计，是造成这个缺陷的原因。如果运气好，一旦 Git 得到更多关注，并其有更多用户要求这个功能，这个功能就会被实现。

初始提交

传统的计算机系统从 0 计数，而不是 1。不幸的是，关于提交，Git 并不遵从这一约定。很多命令在初始提交之前都不友好。另外，一些极少数的情况必须作特别地处理。例如重订一个使用不同初始提交的分支。

Git 将从定义零提交中受益：一旦一个仓库被创建起来，HEAD 将被设为包含 20 个零字节的字符串。这个特别的提交代表一棵空的树，没有父节点，早于所有 Git 仓库。

然后运行 git log，比如，通知用户至今还没有提交过变更，而不是报告致命错误并退出。这与其他工具类似。

每个初始提交都隐式地成为这个零提交的后代。

不幸的是还有更糟糕的情况。如果把几个具有不同初始提交的分支合并到一起，之后的重新修订不可避免的将需要人员的介入。

接口怪癖

对提交 A 和提交 B，表达式 “A..B” 和 “A...B” 的含义，取决于命令期望两个终点还是一个范围。参见 **git help diff** 和 **git help rev-parse** 。

附录 B: 本指南的翻译

我推荐如下步骤翻译本指南，这样我的脚本就可以快速生成 HTML 和 PDF 版本，并且所有翻译也可以共存于同一个仓库。

克隆源码，然后针对不同目标 语言的 IETF tag创建一个目录。参见 W3C 在国际化方面的文章。例如，英语是 “en”，日语是 “ja”，正体中文是 “zh-Hant”。然后在新建目录，翻译这些来自 “en” 目录的 `txt` 文件。

例如，要将本指南译为 克林贡语，你可以键入：

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" 是克林贡语的IETF语言码。
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # 翻译这个文件
```

对每个文件都一样。

打开 Makefile 文件，把语言码加入 ‘TRANSLATIONS‘变量，现在你可以时不时查看你的工作：

```
$ make tlh
$ firefox book-tlh/index.html
```

经常提交你的更新，这样我就知道他们什么时候可以完成。GitHub.com 提供一个便于 fork “gitmatic” 项目的界面，提交你的变更，然后告诉我去合并。

但请按照最适合你的方式做：例如，中文译者就使用 Google Docs。只要你的工作能使更多人看到我的工作，我就高兴。