Git 魔法

Ben Lynn

Git 魔法

前言

1. 致谢！

2. 许可

1. 入门

1.1. 游戏这件事

1.2. 版本控制

1.3. 分布控制

1.3.1. 一个误区

1.4. 冲突合并

2. 基本技巧

2.1. 保存状态

2.1.1. 添加、删除、重命名

2.2. 进阶撤销、重做

2.2.1 还原

2.3. 下载文件

2.4. 最前沿

2.5. 即时发布

2.6. 我们已经做了什么？

3. 克隆进阶

3.1. 计算机间的同步

3.2. 典型源码控制

3.3. 项目分叉

火龙果•整理
uml.org.cn

3.4. 终极备份

3.5. 轻快多任务

3.6. 游击版本控制

4. 分支向导

4.1. 老板键

4.2. 脏活

4.3. 快速修订

4.4. 不间断工作流

4.6. 管理分支

4.7. 临时分支

4.8. 按你希望的方式工作

5. 关于历史

5.1.改正标准

5.2.在这之后

5.3. 本地变更最后部分

5.4. 重写历史

5.5. 制造历史

5.6. 哪儿错了？

5.7. 谁让事情变糟了？

5.8. 个人经验

6. Git 大师

6.1. 源码发布

6.2. 历史记录生成

6.3. 通过 SSH，HTTP 使用 Git

6.4. 提交变更

6.5. 我的提交太大了！

6.6. 别丢了你的 HEAD

6.7. 猎捕 HEAD

6.8. 在 Git 上编译

6.9. 大胆的绝技

7. 解密

7.1. 大象无形

7.2. 数据完整性

7.3. 智能

7.4. 索引

7.5. 裸资源库

7.6. Git 起源

附录 A. Git 的缺点

8.1. 微软 Windows

8.2. 无关的文件

8.3. 谁在编辑什么？

8.4. 文件历史

8.5. 初始克隆

8.6. 不稳定的项目

火龙果•整理
uml.org.cn

8.7. 全局计数器

8.8. 空子目录

8.9. 初始提交

8.10 怪异的接口

附录 B. 本指南的翻译

前言

Git(http://git.or.cz/)是一个版本控制的瑞士军刀，一个可靠的多功能多用途版本控制工具。其非凡的灵活性使得学着用都没那么简单，更别说精通了它。

正如 Arthur C. Clarke 所观察到的，所有称得上足够先进的技术都和魔法难以区分。这是一个走近 Git 的不错途径：新手们可以忽略它内部机理，只把 Git 当个小把戏看，用它奇妙的功能赚取朋友的惊奇，敌人的暴怒。

为了不陷入细节，我们针对特定效应提供大面上的指导。在重复使用之后，慢慢地你会理解每个小技巧是如何工作的，并且还会领会如何合理剪裁组合这些技巧以满足自己的需求。

翻译

中文版（简体）：由 JunJie，Meng 和 JiangWei 提供。

西班牙文：由 Rodrigo Toledo 提供。

葡萄牙文：由 Leonardo Siqueira Rodrigues 提供 [ODT 版本]。

其他版本

单一网页：单一 HTML，无 CSS。

PDF 文件：适合打印。

Debian 包：得到英文网站一份本地拷贝。Ubuntu 包（Jaunty Jackalope） 检查网站是否下线维护。

1. 致谢！

感谢 Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan，Derek Mahar，Frode Aannevik，和 Keith Rarick 提出的建议和改进。

Daniel Baumann 维护 Debian 包。

火龙果•整理
uml.org.cn

JunJie, Meng, JiangWei, Rodrigo Toledo，和 Leonardo Rodrigues 致力于本文档翻译。

感谢其他拥护者和赞美者。如果这是本纸质书，我会在封面上引用你们慷慨的赞美以提高销量。

如果我不小心忘记了感谢谁，请告诉我或者发一个补丁（Patch）给我。

免费 Git 伺服

http://repo.or.cz/ 为自由项目提供服务。第一个 Git 服务器。由最早的一个 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。

2. 许可

本指南在 GNU 通用公共许可协议版本 3 之下发布。当然，源码保存在一个 Git 资源库里，并且可以通过以下命令获得：

$ git clone git://repo.or.cz/gitmagic.git # 创建“gitmatic”目录。

或从以下镜像得到：

$ git clone git://github.com/blynn/gitmagic.git

$ git clone git://gitorious.org/gitmagic/mainline.git

---

1. 入门

我将用类比的方式来介绍版本控制。更严谨些的解释可以参见维基百科版本修订控制条目。

1.1. 游戏这件事

我几乎一生下来就开始玩电脑游戏，直到现在。与之相反，我仅在成人之后才开始使用版本控制系统。我想我的情况并非个案，对比这两者也许可以使得一些概念更容易解释，也更容易理解。

把编辑代码或文档，或无论什么，当作玩游戏。一旦你取得了很多进展，你会想到保存。你会点击值得信赖的编辑器的保存按钮来进行保存。

但这将覆盖旧版本。就像那些在学校里玩的老游戏那样，只有一个存档：你确实可以保存，但一旦保存就不能回到原来的状态。这真让人扫兴，因为可能你之前的存档恰好在这个游戏特别有意思一关，也许某天你想回顾一下。或者更糟糕的，你当前的存档是个必然的败局，这样你不得不从头开始玩。

1.2. 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。或许还会把这些文件进行压缩,来节省空间。这是一个原始的费功夫的版本控制形式。在很久以前，游戏软件就提高了这块，它们里很多都提供了多个自动保存记录功能，基于时间戳的。

火龙果•整理
uml.org.cn

让我们看一些更复杂的情况。比如你有很多在一起的文件，比如项目的源代码，或网站的文件。现在如果你想保留旧版本你不得存档整个目录。手工保存很多个版本不方便，并且很快会变得耗费巨大。

对一些电脑游戏而言，一个保存的游戏记录真的包含在一个充满文件的目录里。这些游戏对玩家屏蔽了具体细节，并展现一个方便的界面来管理目录里的不同版本。

版本控制系统也没有两样。两者都有友好的界面来管理目录里的东西。你可以频繁保存目录的状态，也可以之后加载任一已有保存过的状态。不像大多计算机游戏，版本控制系统通常精于节省存储空间。通常情况下两个版本间只有少数文件的改变，每个文件的改变也不大。只存储差异的部分节省存储空间，而不是把所有的都保存下来。

1.3. 分布控制

现在设想有一个非常难的电脑游戏。太难打了，以至于全世界很多高级玩家决定组队，分享他们保存的游戏记录以攻克它。Speedrun 们就是真实生活中的例子：在同一个游戏里,玩家们专注攻克不同的等级,协同工作创造了惊人的战绩。

如何搭建一个系统，使得他们可以很容易得到其他人的保存记录？并且很容易上载新的记录？

在过去，每个项目都使用中心式版本控制。一个服务器存放所有保存的游戏记录。其他人都不用保存所有的。每个玩家在他们机器上最多保留几个游戏记录。当一个玩家想更新进度时候，他们需要把最新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怎么样？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。他们必须去问中心服务器要那个旧版本的记录。需要的旧版本越多，需要和服务器的交互就越多。

新一代的版本控制系统，Git 就是其中之一，是分布式系统，可以被认作广义上的中心式系统。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。这看起来他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，特别当有漫长历史的时候，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，不再需要和中心服务器通讯了。

1.3.1. 一个误区

一个很常见的错误观念是，分布式系统不太适合需要正式中心存储的项目。这与事实并不相符。给某人照相并不会导致他们的灵魂被偷走。类似地，对主存储进行克隆并不降低它的重要性。

火龙果•整理
uml.org.cn

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会做得更好。网络资源总要比本地资源耗费更多资源。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，在你的项目很小的时候，说你应该用规划不好的系统，好比说，在计算较小数目的时候应该使用罗马数字吗？

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。到你迫切需要一把改锥的那一天，你就会庆幸你有的不单单是一个启瓶器。

1.4. 冲突合并

对于这个话题，电脑游戏的类比显得太单薄了。那么让我们再来看看文档编辑的情况吧。

假设 Alice 在文件开头插入一行，并且 Bob 在文件末尾添加一行。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，并且他们必须选择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。通常它们的行为是可配置的。

---

2. 基本技巧

与其一头扎进 Git 命令的海洋中，不如来点一些基本的例子试试手。例子尽管很简单但每个都很有用。实际上，我开始使用 Git 的头几个月，从来没觉得本章介绍的命令不够用。

2.1. 保存状态

要不要来点猛的？在做之前，先为当前目录所有文件做个快照，使用下列命令：

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

现在如果你的文件编辑乱了，运行：

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

可以回到你原来编辑的地方。再次保存状态：

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

火龙果•整理
uml.org.cn

2.1.1. 添加、删除、重命名

以上的命令将只对在你第一次运行 git add 命令时就已经存在的文件有用。如果要添加新文件或子目录，你需要告诉 Git：

```
$ git add NEWFILES...
```

类似地，如果你想让 Git 忘记某些文件（或许是由于你已经删除了他们）：

```
$ git rm OLDFILES...
```

重命名一个文件和先删除旧文件，再添加新文件的一样。也有一个快捷方式 git mv，这和 mv 命令的语法一样。例如：

```
$ git mv OLDFILE NEWFILE
```

2.2. 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。那么可以使用：

```
$ git log
```

来显示最近提交列表，以及他们的 SHA1 哈希值。下面，键入：

```
$ git reset --hard SHA1_HASH
```

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更新的提交。

另一些时候你想简单地跳到一个旧状态。这种情况，键入：

```
$ git checkout SHA1_HASH
```

这个操作将把你带回过去，同时保存较新提交。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完全不同的。

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。另外，为避免 Git 的抱怨，应该在每次运行 checkout 之前执行 commit 或 reset。

再次用计算机游戏做类比：

火龙果•整理
uml.org.cn

git reset -- hard：加载一个旧的游戏存档，并删除所有比它新的存档。

git checkout：加载旧游戏存档，但如果你从这个存档开始玩，游戏状态将有别于你之前保存的更新的记录。此后任何记录都保存到另一个分支中，代表你进入的另一个现实。后面会详细讲。

你可以选择只恢复特定文件或子目录，把这些加到该命令后面就可以了。

不喜欢剪切粘贴哈希值？那么用：

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

其他的时间格式也是可以的。比如，你可以要求倒数第五次保存状态：

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

2.2.1 还原

在法庭上，事件可以从法庭记录里追踪出来。同样，你可以选择还原特定的提交。

```
$ git commit -a
$ git revert SHA1_HASH
```

将还原特定哈希值的提交。运行 git log 显示该还原记录为一个新的提交。

2.3. 下载文件

得到一个由 Git 管理的项目的拷贝，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 命令谈的更多。

2.4. 最前沿

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：

```
$ git pull
```

火龙果•整理
uml.org.cn

2.5. 即时发布

假设你已经写了一个脚本，你想和他人分享。你可以只告诉他们从你的计算机下载，但如果在你正在改进你的脚本，或加入试验性质的改动时，他们下载了你的脚本，他们可能由此陷入困境。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。这要假设他们有 ssh 访问权限。如果他们没有，你需要运行 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"
```

并且你的用户可以通过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的东西都是可以，不仅是对脚本。

2.6. 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：

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

或者一个特定的版本和版本 2 之间：

```
$ git diff SHA1_HASH "@{2}"
```

除了直接运行 log 和 diff，有时我也用 qgit 浏览历史（http://sourceforge.net/projects/qgit），因为它的图形界面很养眼，或者用

火龙果•整理
uml.org.cn

tig（http://jonas.nitro.dk/tig），一个文本界面东西，在网速不快的情况下也工作的很好。另外，安装一个 Web 服务器，运行 git instaweb，那就可以用任意的浏览器浏览了。

---

3. 克隆进阶

在旧一代的版本控制系统里，checkout 是获取文件的标准操作。你将获得一组期望状态下的文件。

在 Git 和其他分布式版本控制系统里，克隆是标准的操作。通过创建整个资源库的克隆来获得文件。或者说，你实际上把整个服务器做了个镜像。主资源库上能做的是，你都能做。

3.1. 计算机间的同步

这也是我第一次使用 Git 的原因。我可以忍受制作 tar 包或利用 rsync 来作备份或者作简单的同步。但我时而在我笔记本上编辑，时而在台式机上，而且这两台电脑之间也许不能交互。

在一个机器上初始化一个 Git 资源库并提交你的文件。然后转到另一台机器上：

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

创建这些文件和 Git 资源库的第二个拷贝。从现在开始：

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

将从另一台机器拷贝文件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。

3.2. 典型源码控制

为你的文件初始化 Git 资源库：

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

在中心服务器，初始化一个空的 Git 资源库， 如果需要的话，启动 Git 守护进程：

```
$ GIT_DIR=proj.git git init
$ git daemon --detach # it might already be running
```

一些公用主机，比如 repo.or.cz（http://repo.or.cz），通过不同于以上的方法来搭建最初空 Git 资源库，比如在网页上填一个表单。

把你的项目推到中心服务器：

```
$ git push git://central.server/path/to/proj.git HEAD
```

火龙果•整理
uml.org.cn

这样我们准备好了。一个开发人员可以这样 check out 源码：

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

在做了改动之后，check in 源码到主服务器：

```
$ git commit -a
$ git push
```

如果主服务器已经更新了，最新版在 push 之前需要 check out。同步到最新版：

```
$ git commit -a
$ git pull
```

3.3. 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
```

3.4. 终极备份

会有很多禁止篡改的不同位置上的冗余存档吗? 如果你的项目有很多开发人员，那干脆别想了。你的每份代码克隆是一个有效备份。不仅当前状态，还包括你项目整个历史。感谢哈希加密算法，如果任何人的克隆被损坏，只要他们与其他的交互，这个克隆就会打上记号。

如果你的项目并不是那么流行，那就找尽可能多的主服务器来放克隆。

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。

3.5. 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。

火龙果•整理
uml.org.cn

在任何时间，你都可以从另一个克隆提交或拿下变更。

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

3.6. 游击版本控制

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

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

然后以很快的速度克隆它：

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

现在转到新目录并在这个目录工作，在你心里的内容上使用 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"
```

然后转到新目录并运行：

```
$ 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资源库。

命令 git svn 为 Subversion 资源库自动化了以上步骤，并且也可以用作 Git 项目导出到一个 Subversion 资源库（参见 http://google-opensource.blogspot.com/2008/05/export-git-project- to-google-code.html）。

---

4. 分支向导

瞬间就能完成的分支和合并是 Git 最致命的杀手锏。

问题： 外部因素不可避免地使场景切换成为必须。在发布版本中没有任何前兆的突然出现了一个严重臭虫，并且必须不惜一切代价尽快修复。某个特性的截至日期就要来临。写这个功能的家伙正打算离职，因此你应该停下所有你正在做的，抓住他帮你理解这个功能。

中断连贯的思维会降低你的生产力，并且切换上下文也比较缓慢，不方便，中断的越长损失越大。使用中心版本控制我们必须从中心服务器下载一个崭新的工作拷贝。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。

火龙果•整理
uml.org.cn

但是克隆仍然需要拷贝整个工作目录，以及一直到指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，项目文件自身必须在新的工作目录里重新创建。

方案：针对这些情况，Git 有一个更好的工具，比克隆更快速而且节省空间：git branch

在这个神奇的世界里，你目录里的文件突然从一个版本变到另一个版本。除了只是在历史记录里前前后后的转变之外，这个转换还可以做更多。你的文件可以从上一个版本变到实验版本到当前开发版本到你朋友的版本等等。

4.1. 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，就可以用这个功能来遮掩一下。

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 资源库，该资源库记录一个包含特定信息的文件。现在我们键入：

```
$ git checkout -b boss # 之后似乎没啥改变

$ 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 # 切到原来版本的文件
```

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 # 切到适合老板眼睛的版本
```

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。

4.2. 脏活

比如你正在开发某个特性，并且由于某种原因，你需要回到一个旧版本，临时加进几行打印语句来看看一些东西是如何工作的。那么：

```
$ git commit -a
$ git checkout SHA1_HASH
```

火龙果•整理
uml.org.cn

现在你可以满世界加丑陋的临时代码。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。观察任何未提交变更都转结了。

如果你想之后保存临时变更怎么办？简单：

$ git checkout -b dirty

并且在切换到主分支之前提交。无论你什么时候想返回脏的变更，只需键入：

$ git checkout dirty

当我们讨论加载旧状态的时候，我们在较早章节曾接触过这个命令。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。

4.3. 快速修订

你正在做某件事的当间，被告知先停下来所有的事情去修理一个新近发现的臭虫：

$ git commit -a

$ git checkout -b fixes SHA1_HASH

那么一旦你修正了这个臭虫：

$ git commit -a -m "Bug fixed"
$ git push # 到中心资源库
$ git checkout master

并可以继续你原来的任务。

4.4. 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。待修的汽车呆在车库里，毫无办法，直到特定的零件从工厂运来。一个原型的构建在可以继续之前可能需要等待成型的芯片。

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

在 Git 里，多亏了轻松的分支和合并操作，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

火龙果•整理
uml.org.cn

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，你将经常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

$ edit files # 修修补补

$ git checkout part2 # 返回第二部分

$ 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

$ some_command # 或许你需要运行一些东西

在正式准备好的当前目录

$ git merge part2 # 合并第二部分

$ git branch -d 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应该在几次提交前创建分支。那么键入：

$ git branch -m master part2 # 重命名 "master" 分支到 "part2".

$ git checkout SHA1 -b master # 第一部分的提交

主分支只包括第一部分，第二部分包括其他的。

4.5. 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 checkout -b sanitized

$ git checkout -b medley

接下来，可以做任何事情：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 SHA1_HASH

在“sanitized”分支应用给定提交。在分支上选择一个最合适的点，你可以建设一个只包含永久代码，有关提交组合在一起的分支。

4.6. 管理分支

键入：

$ git branch

火龙果•整理
uml.org.cn

来列出所有分支。总有一个叫做“master”，并且你默认从这开始。一些人主张别碰“master”分支，而是创建你自己版本的新分支。

使用-d -m 选项来删除、移动（重命名）分支。参见 git help branch。

4.7. 临时分支

很快你会发现你经常会因为一些相同的原因创建短期的分支。或许有些时候你创建分支只是为了保存当前状态，同时你方便在上一个保存状态执行一些操作，诸如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，来看看那正演什么，但并不是简单地按几个按钮，你必须创建，检出,删除并提交临时分支。幸运的是，Git 已经有了和电视机遥控器一样方便的功能：

$ 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。

4.8. 按你希望的方式工作

诸如 Mozzila Firefox（http://www.mozilla.com/)允许你打开多个标签和多个窗口。切换标签可以在同一个窗口展示不同内容。Git 的分支就像是你工作目录的标签。继续这个类比，Git 克隆好似打开一个新的窗口。这两种操作的易用性提高了用户体验。

在更高层，几个 Linux 窗口管理器允许你有多个桌面：你可以立即切到一个显示器

另一个相似的例子是 screen（http://www.gnu.org/software/screen/）工具。这个好东西允许你在同一个终端上创建，销毁终端会话,还可以在多个终端会话之间切换。不是打开一个新的终端(类似于克隆），如果你使用 screen（类似于分支）,你可以使用同一个终端。实际上，你还可以利用 screen 做更多的事，不过这是另一个话题的内容了。

在 Git 里，克隆、分支和合并是本地操作，很快。促使你能使用最适合你的组合。Git 让你能按你确切想要的工作。

---

5. 关于历史

Git 一系列的分布式特性使得历史可以很容易的被修改掉。但如果你篡改了过去，需要小心：只重写你独自拥有的那部分。如同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，这时如果合并的话,你会遇到一致性方面的问题。

当然，如果你掌管所有分支的话，那就没啥问题了，既然你可以覆盖他们的。

一些开发人员强烈地感觉历史应该永远不变，无论不好的部分或这所有的部分。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 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，至于如何使用它，那是你的事了。

火龙果•整理
uml.org.cn

5.1.改正标准

已经提交了，但你期望你已经输入了另外的信息？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，然后与性上面的命令。

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
```

5.2.在这之后

假设前面的问题还要糟糕十倍。在很长一段时间里我们做了一系列的提交。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
```

并且后 10 个提交会出现在你喜爱的$EDITOR。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
```

之后：

通过删除行来移去提交。

通过为行重新排序来为提交重新排序。

用“edit”来替换“pick”来标志一个提交可修改。

用“squash”来替换“pick”来将一个提交和前一个合并。

如果你把一个提交标记为可编辑，那么运行

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

或者，运行：

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

这样尽早提交，经常提交都没关系：你可以之后使用 rebase 来梳理清楚。

5.3. 本地变更最后部分

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。

火龙果•整理
uml.org.cn

但现在你本地 Git 克隆的变更历史与官方变更乱成了一锅粥。你会更期望在变更列表里，你所有的变更能够连续的

这就是上面提到的 git rebase 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。

5.4. 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，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，那讨论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。

事后，你必须使用你更改过的版本替换你项目的克隆，如果你希望之后和他们交互的话。

5.5. 制造历史

想把一个项目迁移到 Git 吗？如果这个项目是用一些比较有名的系统管理着，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到 Git 里。

否则，看一下 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。通常可以很快地组织一个包含这个命令的脚本，运行一次就可以一次迁移整个项目。

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
```

火龙果•整理
uml.org.cn

```
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 资源库，键入：

```
$ mkdir project; cd project; git init
$ git fast-import < /tmp/history
```

你可以从这个项目 checkout 出最新的版本，使用：

```
$ git checkout master .
```

5.6. 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。然后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 SHA1_OF_BAD_VERSION
$ git bisect good SHA1_OF_GOOD_VERSION
```

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，如果还是错误的：

$ 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，这种二进制查询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 COMMANDGit 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。

火龙果•整理
uml.org.cn

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。

5.7. 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 FILE 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。

5.8. 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，通常当用户试图避免使用那些能不用则不用的高级命令时，所有操作都变得很慢。在极端的情况下，即使是最基本的命令也会变慢。当用户必须运行慢的命令的时候，由于工作流被打断，生产力降低。

我有这些现象的一手经验。Git 是我使用的第一个版本控制系统。我很快变得适应了它，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，我被震惊了。我那有些脆弱的网络连接没给 Git 带来大麻烦，但是当它需要像本地硬盘一样稳定的时候，它使开发困难重重。另外，我发现我自己有选择地避免特定的命令，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慢的命令的时候，这种等待极大地破坏了我思绪连续性。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。共同的努力加剧了拥挤，这等于是鼓励个人下次消费更多带宽以避免更长时间的等待。

---

6. Git 大师

这个有点自夸的标题下是我压箱底的 Git 不在编的技巧。

6.1. 源码发布

就我的项目而言，Git 确切跟踪了我想打包并发布给用户的文件。创建一个源码包，我运行：

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

6.2. 历史记录生成

保留一份历史记录是一个好实践（http://en.wikipedia.org/wiki/Changelog），这甚至是一些项目所要求的。如果你变更提交的很频繁，这是应该的，可以生成一份历史记录，通过键入：

$ git log > Changelog

火龙果•整理
uml.org.cn

6.3. 通过 SSH，HTTP 使用 Git

假设你有一台 web 服务器的 ssh 访问权限，但上面并没有安装 Git。没有比服务器上自带协议更省事的方法的，Git 可以通过 HTTP 来进行通信。

那么在你的帐户下下载，编译并安装 Git。在你的 Web 目录里创建一个 Git 资源库：

```
$ GIT_DIR=proj.git git init
```

在目录“proj.git”目录，运行

```
$ git --bare update-server-info
$ chmod a+x hooks/post-update
```

从你的计算机通过 SSH 推送：

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

别人可以通过如下命令，得到你的项目：

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

6.4. 提交变更

对特定项目，当你加入、删除、重命名一些文件时，告诉 Git 这个操作是有困难的。你可以键入：

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

Git 会查看当前目录的这些文件，并独立计算出所有细节。如果你还试图同时提交的话，可以不使用第二个添加命令，而是运行 git commit -a。

也可以用单独的一行完成以上的任务：

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

这里-z 和-0 选项可以避免文件名中包含特殊字符带来的错误。注意该命令也添加应该被忽略的文件。这时你或许希望使用-x 或-X 选项。

6.5. 我的提交太大了！

是不是太长时间忘记了提交？痴迷地编码，以至到现在才想起有源码控制工具这回事？提交一系列不相关的变更，因为那是你的风格？

别着急，运行：

```
$ git add -p
```

火龙果•整理
uml.org.cn

为你做的每次修改，Git 将为你展示变动的代码，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”。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）。确信你没加上“-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？一个一个地查看变更令人沮丧，心态麻木。这种情况下，使用 git add -i，界面不是很直观，但更灵活。敲几个键，你可以一次决定阶段提交几个文件，或查看并检查特定文件的变更。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。

6.6. 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。Git 提供了命令来移动它。 例如：

$ git reset HEAD~3

将立即向回移动 HEAD 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 git reset 手册页。

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 SHA1

但假设你从来没有记下呢？别着急，像这些命令，Git 保存原先的 Head 为一个叫 ORGI_HEAD 的标记，你可以安全体面的返回：

$ git reset ORIG_HEAD

6.7. 猎捕 HEAD

或许 ORG_HEAD 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的分支上的一个远古的提交。

默认，Git 保存一个提交至少两星期，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。但有更容易的办法。

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。后者可以被用来分支上一些不小心丢掉的提交的哈希值。

命令 reflog 为访问这些日志提供文件友好的接口，试试

$ git reflog

火龙果•整理
uml.org.cn

更多信息参见其文档页。

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 天后，且运行 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。

6.8. 在 Git 上编译

依照真正的 UNIX 风格，Git 的设计允许其可以被很容易用作其他程序的底层组件。有图形界面，Web 界面，可选择的命令行界面，并且或许很快你将有一个或两个你自己的脚本，调用 Git。

一个简单的技巧是，用 Git 内建 alias 命令来缩短你最常用命令的长度：

```
$ 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
```

另一个是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-
```

更多例子参见 Git 主页（http://git.or.cz/）。

6.9. 大胆的绝技

最近版本的 Git 使得用户意外销毁数据变得更困难了。这或许是升级的最令人瞩目的原因了。虽然如此，有时你的确想销毁数据。我们展示如何覆盖为通用命令所设的防卫设施。只在你知道你在做什么的时候使用他们。

Checkout:如果你有未提交的变更，平常的 checkout 会失败。销毁你的变更，并无论如何都 checkout 一个指定的提交，使用强制选项：

```
$ git checkout -f COMMIT
```

火龙果•整理
uml.org.cn

另外，如果你为 checkout 指定的路径，那就没有安全检查了。提供的路径将被不加提示地覆盖。如果你使用这种方式的 checkout，一定要小心。

重置：在有未提交变更的情况下，重置也会失败。强制其通过，运行：

$ git reset --hard [COMMIT]

分支：如果删除一个分支会导致变更丢失的，删除操作会失败。强制删除，键入：

$ git branch -D BRANCH # 注意不是 -d

类似地，试图通过移动覆盖一个分支也会失败，如果数据确信会丢失的话。强制移动分支，键入：

$ git branch -M [SOURCE] TARGET # 注意不是 -m

不像 checkout 和重置，这两个命令延迟数据销毁的时间。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到（参见上面“猎捕 HEAD”一节）。默认情况下，这些数据会保存至少两星期。

---

7. 解密

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。更深入的描述参见用户手册（http://www.kernel.org/pub/software/scm/git/docs/user-manual.html）。

7.1. 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。那是，直到你需要它，而且那时候你也很高兴 Git 一直默默注视着你整个的工作。

其他版本控制系统可不许你忘记他们。在你显示地告诉服务器你打算编辑哪些文件前，文件权限也许是只读的。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。

秘诀就在你工作目录里的.git 目录。Git 在这里记录你项目的历史。最前面的“.”使这个文件在 ls 的输出中不显示。除了你的提交或同步变更以外，所有版本控制操作都在这个目录里操作。

你对你文件的命运有完全的掌控，因为 Git 根本不关心你对他们做了什么。Git 可以很容易地从.git 里重新创建出一个保存状态。

7.2. 数据完整性

很多人把加密和保持信息机密性关联起来，但一个同等重要的目标是保证信息的安全性。合理使用哈希加密功能可以防止意外地或恶意地数据损坏

一个 SHA1 哈希值是一个 160 位的 ID 数，用它可以唯一标识你一生中遇到的每个二进制字符串。 实际上不止如此：任何人几辈子用到的所有字符串都可以用它来唯一标识。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。

火龙果•整理
uml.org.cn

一个重要的发现是，一个 SHA1 哈希值本身也是一个二进制字符串，因此我们可以为包含其他哈希值的字符串生成哈希值。

大略地讲，所有 Git 处理的文件通过他们的唯一 ID 来引用，而不是通过他们的文件名。所有数据保存在“.git/objects”子目录里，那里你找不到任何标准的文件名。文件内容都是我们叫做‘blobs’的二进制字符串，这些二进制字符串跟他们的文件名也是剥离的。

而文件名会被记录在别的地方。他们被存储在‘树’对象里，这里记录着文件名和他们内容的 ID 值对应关系列表。既然树本身是一个二进制字符串，它也有一个唯一 ID，这也是它是如何被存储在“.git/objects”子目录里的办法。树可以出现在其他树的列表里，因此一个目录树和其所有文件可以被表达为树和 blob。

最终，一个‘提交’包含一条消息：一些树 ID 和他们之间如何关联的信息。一个提交也是一个二进制字符串，因为它也有一个唯一 ID。

你可以自己看看：在.git/objects 目录里随便找一个哈希值，然后键入：

$ git cat-file -p SHA1_HASH

现假设某人试图重写历史记录，并准备更改一个古老版本里一个文件的内容。那么这个文件的 ID 将被更改，因为这个文件变成了另一个二进制字符串。这会影响每一个引用这个文件的的树对象，并导致所有与该树相关的已提交对象都需要被更改。因为有多个文件中包含着错误的 ID，当人们完成提交操作时，就会发现这个遭到破坏的资源库。

我已经忽略了诸如文件权限和签名之类的细节。但简单地讲，只要 20 字节的长度就能代表最后的提交是安全的，破坏 Git 资源库是不可能的。

7.3. 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 紧接 git rm 是完全一致的。

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！虽然它不能包揽所有的情况，但它已经做的很好了，并且这个功能也总在改进改进。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。

7.4. 索引

为每个跟踪的文件，Git 在一个名为 index 的文件里记录其诸如大小，创建时间和最后修改时间的信息。Git 比较其当前与在 index 里的统计资料，以确定文件是否更改。如果一致，那 Git 就避免重新读取该文件。

因为调用统计资料比读文件内容的开销小很多，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状态。

7.5. 裸资源库

你或许想知道那些在线 Git 资源库用的是什么格式。他们除了具有类似 proj.git 的名字以外，和你的.git 目录一样，是无格式的。另一个不同点是他们没有关联任何工作目录。

大多数 Git 命令默认 Git 的索引是存放在.git 目录，但在这些裸资源库上并不是这样，所以会导致命令执行失败。为了解决这个问题，可以设置环境变量 GIT_DIR 来指定裸资源库所在路径，或者在裸资源库的目录里运行 Git 时加上--bare 选项。

7.6. Git 起源

Linux 内核邮件列表上的一则帖子（http://lkml.org/lkml/2005/4/6/121）描述了形成 Git 的一系列事件。对 Git 史学家而言，整个的讨论是一个令人着

火龙果•整理
uml.org.cn

迷的历史探索过程。

附录 A. Git 的缺点

我避开了对一些 Git 现存问题的讨论。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，还有少数无解问题，只能暂时等待。更好的办法是，投入进来，一起解决这些问题。

我对版本控制系统已经有了一些想法，并基于 Git 写了一个实验性系统（http://www-cs-students.stanford.edu/~blynn/gg/），并解决了一部分问题。

8.1. 微软 Windows

Git 在微软 Windows 上可能有些繁琐：

Cygwin（http://cygwin.com），一个 Windows 下的类 Linux 的环境，里面有一个 Windows 移植 Git（http://cygwin.com/package/git/）。

Git on MSys（http://code.google.com/p/msysgit/）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。

8.2. 无关的文件

如果你的项目非常大，包含很多无关的文件，而且正在不断改变，Git 可能比其他系统更不管用，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。

8.3. 谁在编辑什么？

一些版本控制系统在编辑前强迫你显示地用某个方法标记一个文件。尽管这种要求很繁琐，特别是这个动作需要和中心服务器通讯时。它还是有以下两个好处的：

比较速度快，因为只有被标记的文件需要检查。

通过查询在中心服务器上谁把某个文件标记为编辑状态，人员可以获知还有谁正在编辑这个文件。

使用适当的脚本，你也可以使用 Git 达到同样的效果。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。

8.4. 文件历史

因为 Git 记录的是项目范围的变更，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 delta 集合的做法好多了。

8.5. 初始克隆

当一个项目有相当长度的历史后，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。

从长时间使用看，初始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 创建一个浅克隆比较划算些。这种克隆初始化的更快，但生成的克隆仅具有较少的功能。

8.6. 不稳定的项目

火龙果•整理
uml.org.cn

Git 是变更的大小决定写入的速度快慢。一般人做了小的改动就会提交新版本。这里一行臭虫修改，那里一个新功能，修改掉的注释等等。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，你的历史记录会以整个项目的大小增长。

任何版本控制系统对此都束手无策，但标准的 Git 用户将遭受更多，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。

又或许，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持续更改，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，使用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，因此固件修订将使资源库无谓的变大。

这种情况，源码应该保存在一个 Git 资源库里，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，对固件只做同步或 Git 浅克隆。

8.7. 全局计数器

一些中心版本控制系统维护一个正整数，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 资源库就增大这个整数，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，因为只有中心资源库以及它的计数器对每个人才有意义。

8.8. 空子目录

空子目录不可追踪。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，更多用户要求这个功能，这个功能就会被实现。

8.9. 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：一旦一个资源库被创建起来，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 资源库。

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。例如，重新修订一个不相关的分支将使得整个分支被加入到零提交里。这样，除了初始提交命令之

火龙果•整理
uml.org.cn

外的所有命令都会被应用，这会导致一个合并冲突。一个绕过的办法是在初始提交上紧跟着 git commit -c，再使用 git checkout，之后继续重新修订剩下的。

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需要人员的介入。

8.10 怪异的接口

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点还是一个区域。参见 git help diff 和 git help revparse。

有些错误信息很莫测。比如，如果 Git 不能创建锁文件 packed-refs.lock，那么你将看到：

$ git branch -D foo

error: cannot delete 'refs/heads/foo' from packed refs

error: Error deleting branch 'foo"

如果文件系统不许前一个命令删除锁文件，这就可能发生。解决这个问题的办法是，先确信没有 Git 命令在运行，然后删除锁文件重试。

Git 的 push 命令忽略目标资源库的工作目录状态。为避免出现问题，只在空的资源库运行 push 命令，或建立一个专用的只推的分支。

附录 B. 本指南的翻译

克隆源码，之后创建相应于 language’s IETF tag 目录：参见 W3C article on internationalization。例如，英语是“en”，日文是“ja”，正体中文是“zh-Hant”。在新目录存放译自“en”目录下的文本文件。

比如，翻译本指南到克林贡文，你可以键入：

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

$ cd gitmagic

$ mkdir tlh # “tlh”是克林贡文的语言代码。

$ cd tlh

$ cp ../en/intro.txt .

$ edit intro.txt # 翻译这个文件

并翻译其他的文本文件。你可以递增地查看你的工作：

$ make LANG=tlh

$ firefox book.html

经常提交你的改动，然后当它们可以的时候让我知道。GitHub.com 有一个这样运行的接口：将项目“gitmagic”分叉，推入你的改动，然后要求我来合并。

我喜欢这种翻译办法，这样我的脚本就可以生成 HTML 和 PDF 版本。并且，它也有利于在官方资源库保存所有翻译的版本。不过请选择最适合你的来做：比如，中文译者使用了 Google Docs。只要你的工作能使更多人看到我的工作，我就很高兴。

火龙果•整理
uml.org.cn

火龙果•整理
uml.org.cn