第 9 篇：附录

附录A Git 命令索引

每一个 Git 子命令都和特定目录下的一个名为 *git-<cmd>* 的文件相对应，也就是在这个特定目录下存在的名为 *git-<cmd>* 的可执行文件[1]可以用命令 *git <cmd>* 来执行。可以用下面的命令查看这个特定目录的位置：

```
$ git --exec-path
/usr/lib/git-core/
```

在这个目录下有 150 多个可执行文件，也就是说 Git 有非常多的子命令。在如此众多的子命令中，常用的实际上只有不到三分之一，其余的命令或者作为底层命令供其他命令及脚本调用，或者用于某些生僻场合，或者已经过时但出于兼容性的考虑而仍然保留。下面的表格分门别类地对所有的 Git 命令进行概要性的介绍，凡是在本书出现过的命令将标出其所在的章节号和页码。

A.1 常用的 Git 命令

命令	相关章节	页数	简要说明
git add	4.1; 10.2.3; 10.4	59; 117; 118	添加至暂存区
git add--interactive	10.6	122	交互式添加
git apply	38.1	534	应用补丁
git am	20.2	298	应用邮件格式补丁
git annotate	-	-	同义词，等同于 git blame

[1] 其中有几个脚本不能单独运行，而是被其他脚本包含，用于提供相应的函数库。如 *git-sh-setup*

git archive	10.9	129	文件归档打包
git bisect	11.4.6	152	二分查找
git blame	11.4.5	151	文件逐行追溯
git branch	18.2	258	分支管理
git cat-file	6.1	83	版本库对象研究工具
git checkout	8.1; 18.4.2	99; 262	检出到工作区、切换或创建分支
git cherry-pick	12.3.1	162	提交拣选
git citool	11.1	131	图形化提交，相当于 git gui 命令
git clean	5.3	78	清除工作区未跟踪文件
git clone	13.1; 41.3.2	180; 567	克隆版本库
git commit	4.1; 4.4; 5.4	60; 65; 81	提交
git config	4.3	63	查询和修改配置
git describe	17.1	235	通过里程碑直观地显示提交 ID
git diff	5.3	80	差异比较
git difftool	-	-	调用图形化差异比较工具
git fetch	19.1	286	获取远程版本库的提交

git format-patch	20.1	296	创建邮件格式的补丁文件。参见 git am 命令
git grep	4.2	61	文件内容搜索定位工具
git gui	11.1	131	基于 Tcl/Tk 的图形化工具，侧重提交等操作
git help	3.1.2	23	帮助
git init	4.1; 13.4	59; 185	版本库初始化
git init-db	-	-	同义词，等同于 git init
git log	11.4.3	147	显示提交日志
git merge	16.1	211	分支合并
git mergetool	16.4.2	222	图形化冲突解决
git mv	10.4	119	重命名
git pull	13.1	180	拉回远程版本库的提交
git push	13.1	180	推送至远程版本库
git rebase	12.3.2	167	分支变基
git rebase--interactive	12.3.3	171	交互式分支变基
git reflog	7.2	95	分支等引用变更记录管理
git remote	19.3	290	远程版本库管理

git repo-config	-	-	同义词，等同于 git config
git reset	7.1	94	重置改变分支“游标”指向
git rev-parse	4.2; 11.4.1	62; 141	将各种引用表示法转换为哈希值等
git revert	12.5	177	反转提交
git rm	10.2.2	116	删除文件
git show	11.4.3	149	显示各种类型的对象
git stage	-	-	同义词，等同于 git add
git stash	9.2	108	保存和恢复进度
git status	5.1	71	显示工作区文件状态
git tag	17.1	234	里程碑管理

A.2 对象库操作相关命令

命令	相关章节	页数	简要说明
git commit-tree	12.4	175	从树对象创建提交
git hash-object	12.4	176	从标准输入或文件计算哈希值或创建对象
git ls-files	5.3; 41.3.1	79; 564	显示工作区和暂存区文件
git ls-tree	5.3	79	显示树对象包含的文件

git mktag	-	-	读取标准输入创建一个里程碑对象
git mktree	-	-	读取标准输入创建一个树对象
git read-tree	24.2	349	读取树对象到暂存区
git update-index	41.3.1	565	工作区内容注册到暂存区及暂存区管理
git unpack-file	-	-	创建临时文件包含指定 blob 的内容
git write-tree	5.3	79	从暂存区创建一个树对象

A.3 引用操作相关命令

命令	相关章节	页数	简要说明
git check-ref-format	17.7	248	检查引用名称是否符合规范
git for-each-ref	-	-	引用迭代器，用于 shell 编程
git ls-remote	13.4	186	显示远程版本库的引用
git name-rev	17.1	236	将提交 ID 显示为友好名称
git peek-remote	-	-	过时命令，请使用 git ls-remote
git rev-list	11.4.2	144	显示版本范围
git show-branch	-	-	显示分支列表及拓扑关系
git show-ref	14.1	187	显示本地引用

git symbolic-ref	-	-	显示或设置符号引用
git update-ref	-	-	更新引用的指向
git verify-tag	-	-	校验 GPG 签名的 Tag

A.4 版本库管理相关命令

命令	相关章节	页数	简要说明
git count-objects	-	-	显示松散对象的数量和磁盘占用
git filter-branch	35.4	511	版本库重构
git fsck	14.2	190	对象库完整性检查
git fsck-objects	-	-	同义词，等同于 git fsck
git gc	14.4	193	版本库存储优化
git index-pack	-	-	从打包文件创建对应的索引文件
git lost-found	-	-	过时，请使用 git fsck --lost-found 命令
git pack-objects	-	-	从标准输入读入对象 ID，打包到文件
git pack-redundant	-	-	查找多余的 pack 文件
git pack-refs	14.1	188	将引用打包到 .git/packed-refs 文件中
git prune	14.2	190	从对象库删除过期对象

git prune-packed	-	-	将已经打包的松散对象删除
git relink	-	-	为本地版本库中相同的对象建立硬连接
git repack	14.4	193	将版本库未打包的松散对象打包
git show-index	14.1	188	读取包的索引文件，显示打包文件中的内容
git unpack-objects	-	-	从打包文件释放文件
git verify-pack	-	-	校验对象库打包文件

A.5 数据传输相关命令

命令	相关章节	页数	简要说明
git fetch-pack	15.1	201	执行 git fetch 或 git pull 命令时在本地执行此命令，用于从其他版本库获取缺失的对象
git receive-pack	15.1	201	执行 git push 命令时在远程执行的命令，用于接受推送的数据
git send-pack	15.1	201	执行 git push 命令时在本地执行的命令，用于向其他版本库推送数据
git upload-archive	-	-	执行 git archive --remote 命令基于远程版本库创建归档时，远程版本库执行此命令传送归档
git upload-pack	15.1	201	执行 git fetch 或 git pull 命令时在远程执

			行此命令，将对象打包、上传

A.6 邮件相关命令

命令	相关章节	页数	简要说明
git imap-send	-	-	将补丁通过 IMAP 发送
git mailinfo	-	-	从邮件导出提交说明和补丁
git mailsplit	-	-	将 mbox 或 Maildir 格式邮箱中邮件逐一提取为文件
git request-pull	21.2.1	313	创建包含提交间差异和执行 PULL 操作地址的信息
git send-email	20.1	297	发送邮件

A.7 协议相关命令

命令	相关章节	页数	简要说明
git daemon	28.2	406	实现 Git 协议
git http-backend	27.2	400	实现 HTTP 协议的 CGI 程序，支持智能 HTTP 协议
git instaweb	27.3.4	405	即时启动浏览器通过 gitweb 浏览当前版本库
git shell	-	-	受限制的 shell，提供仅执行 Git 命令的 SSH 访

			问
git update-server-info	15.1	201	更新哑协议需要的辅助文件
git http-fetch	-	-	通过 HTTP 协议获取版本库
git http-push	-	-	通过 HTTP/DAV 协议推送
git remote-ext	-	-	由 Git 命令调用，通过外部命令提供扩展协议支持
git remote-fd	-	-	由 Git 命令调用，使用文件描述符作为协议接口
git remote-ftp	-	-	由 Git 命令调用，提供对 FTP 协议的支持
git remote-ftps	-	-	由 Git 命令调用，提供对 FTPS 协议的支持
git remote-http	-	-	由 Git 命令调用，提供对 HTTP 协议的支持
git remote-https	-	-	由 Git 命令调用，提供对 HTTPS 协议的支持
git remote-testgit	-	-	协议扩展示例脚本

A.8 版本库转换和交互相关命令

命令	相关章节	页数	简要说明
git archimport	-	-	导入 Arch 版本库到 Git
git bundle	-	-	提交打包和解包，以便在不同版本库间传递

git cvsexportcommit	-	-	将 Git 的一个提交作为一个 CVS 检出
git cvsimport	-	-	导入 CVS 版本库到 Git。或者使用 cvs2git
git cvsserver	-	-	Git 的 CVS 协议模拟器，可供 CVS 命令访问 Git 版本库
git fast-export	-	-	将提交导出为 git-fast-import 格式
git fast-import	35.3	506	其他版本库迁移至 Git 的通用工具
git svn	26.1	380	Git 作为前端操作 Subversion

A.9 合并相关的辅助命令

命令	相关章节	页数	简要说明
git merge-base	11.4.2	146	供其他脚本调用，找到两个或多个提交最近的共同祖先
git merge-file	-	-	针对文件的两个不同版本执行三向文件合并
git merge-index	-	-	对 index 中的冲突文件调用指定的冲突解决工具
git merge-octopus	-	-	合并两个以上的分支。参见 git merge 的 octopus 合并策略
git merge-one-file	-	-	由 git merge-index 调用的标准辅助程序
git merge-ours	-	-	合并使用本地版本，抛弃他人版本。参见 git

			merge 的 ours 合并策略
git merge-recursive	-	-	针对两个分支的三向合并。参见 git merge 的 recursive 合并策略
git merge-resolve	-	-	针对两个分支的三向合并。参见 git merge 的 resolve 合并策略
git merge-subtree	-	-	子树合并。参见 git merge 的 subtree 合并策略
git merge-tree	-	-	显式三向合并结果，不改变暂存区
git fmt-merge-msg	-	-	供执行合并操作的脚本调用，用于创建一个合并提交说明
git rerere	-	-	重用所记录的冲突解决方案

A.10 杂项

命令	相关章节	页数	简要说明
git bisect--helper	-	-	由 git bisect 命令调用，确认二分查找进度
git check-attr	41.1.2	551	显示某个文件是否设置了某个属性
git checkout-index	-	-	从暂存区拷贝文件至工作区
git cherry	18.4.4	265	查找没有合并到上游的提交
git diff-files	-	-	比较暂存区和工作区，相当于 git diff --raw

git diff-index	-	-	比较暂存区和版本库，相当于 git diff --cached --raw
git diff-tree	-	-	比较两个树对象，相当于 git diff --raw A B
git difftool--helper	-	-	由 git difftool 命令调用，默认要使用的差异比较工具
git get-tar-commit-id	10.9	129	从 git archive 创建的 tar 包中提取提交 ID
git gui--askpass	-	-	命令 git gui 的获取用户口令输入界面
git notes	41.5.2	570	提交评论管理
git patch-id	-	-	补丁过滤行号和空白字符后生成补丁唯一 ID
git quiltimport	20.3.2	305	将 Quilt 补丁列表应用到当前分支
git replace	41.4.2	569	提交替换
git shortlog	-	-	对 git log 的汇总输出，适合于产品发布说明
git stripspace	-	-	删除空行，供其他脚本调用
git submodule	23.1	337	子模组管理
git tar-tree	-	-	过时命令，请使用 git archive
git var	-	-	显示 Git 环境变量
git web--browse	-	-	启动浏览器以查看目录或文件

git whatchanged	-	-	显示提交历史及每次提交的改动
git-mergetool--lib	-	-	包含于其他脚本中，提供合并/差异比较工具的选择和执行
git-parse-remote	-	-	包含于其他脚本中，提供操作远程版本库的函数
git-sh-setup	-	-	包含于其他脚本中，提供 shell 编程的函数库

附录B Git 与 CVS 面对面

B.1 面对面访谈录

Git：我的提交是原子提交。每次提交都对应于一个目录树（树对象）。因为我的提交 ID 是对目录树及相关的提交信息建立的一个 SHA1 哈希值，所以可以保证数据的完整性。

CVS：我承认这是我的软肋，一次错误或冲突的提交会导致部分数据被提交，而部分数据没有提交，版本库完整性被破坏，所以人们才设计出来 Subversion（SVN） 来取代我。

Git：我的分支和里程碑管理非常快捷。因为我的分支和里程碑就是一个记录提交 ID 的 40 字节的文件，你的呢？

CVS：你怎么又提到别人的痛处了！我的分支和里程碑创建速度还是很快的，...嗯... ， 如果在版本库中只有几个文件的话。当然如果版本库中的文件很多，创建分支和里程碑就需要花费很长的时间。有些人对此忍无可忍，于是设计出 SVN 来取代我。

Git：其实我不用里程碑都没有关系，因为每个提交 ID 就对应于唯一的一个提交状态。

CVS：这也是我做不到的。我没有全局版本号的概念，每个文件都通过单独的版本号记录其变更历史，所以人们在使用我的时候必须经常用里程碑（tag）对我的状态进行标识。还需要提醒一句的是，如果版本库中的文件太多，创建里程碑是很耗时的，因为要逐一打开每个版本库中的文件，并在其中记录里程碑和文件版本的关联。

Git：我的工作区很干净。只在工作区的根目录下有一个 `.git` 目录，此外再无其他辅助目录或文件。

CVS：我要在工作区的每一个目录下都放置一个 `CVS` 目录，这个目录下有个 `Entries` 文件很重要，记录了对应工作区文件的检出版本及时间戳等信息。这样做的好处是可以将工作区移动到任何其他磁盘和目录，而毫不影响使用，甚至我可以将工作区的一个子目录拿出来，作为独立的工作区。

Git：我也可以将工作区移动到其他磁盘，但是要保证工作区下的 `.git` 目录和工作区一同移动。不可以只移动工作区下的一个目录到其他磁盘或目录，那样的话移出的目录就不能工作了。

Git：我的网络传输效率很高。在和其他版本库交互时，对方会告诉我他有什么，我也知道我有什么，因为只传输缺失对象的打包文件，所以效率很高而且能够显示传输进度。

CVS：这一点我不行。因为我本地没有文件做对照，所以我在传输的时候不可能做到增量传输。

Git：我甚至可以不需要网络，因为我在本地拥有完整的版本库，几乎所有的操作都在本地完成。

CVS：我的操作处处需要网络，如果版本库在网络中的其他服务器上，而且网速又比较慢，那么查看日志或历史版本都需要等很长时间。

CVS：你怎么没有更新（*update*）命令？还有你为什么老是要执行检出命令（*checkout*）？对我而言，检出命令只在工作区创建时一次性完成。

Git：你的检出命令（*checkout*）是从远程版本库服务器获取数据来完成本地工作区的创建，版本库仍然位于远程服务器上。你的更新（*update*）命令执行的很慢，对么？之所以你需要执行更新命令是因为你的版本库在远程啊。别忘了我的版本库是在本地，本地的版本库会随着我在本地工作区中的操作（如提交）而更新。我的检出（*checkout*）操作是将本地版本库的数据检出到本地工作区，用于恢复本地丢失或错误改动的文件，也用于切换不同的分支。我也有一个和你的更新（*update*）操作类似的、比较耗时的网络操作命令： *git fetch* 或 *git pull*，这两个操作是从别人的版本库获取他人的改动。一般使用我（Git）做团队协作的时候，会部署一个集中共享的版本库，我就用这两个命令（*git fetch* 或 *git pull*）从共享的版本库执行拉回操作。也许你（CVS）会觉得 *git fetch* 或 *git pul*l 和你的 *cvs update* 命令更像吧。至于你的检出命令（*cvs checkout*），实际上和我克隆命令（*git clone*）很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

CVS：为什么你的检入（*commit*）命令执行得那么快？

Git：是的，我的检入命令飞一般就执行完了，这也是因为版本库就在本地。也许你（CVS）会觉得我的推送命令（*git push*）和你的检入命令（*cvs commit*）更相像，其实这是一个误会。如果我不做本地提交，是没有东西可以推送（*git push*）的。你（CVS）每一次提交都要和版本库进行网络通讯，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次 *git push*，将我本地版本库中的新提交推送给远程版本库。

CVS：我每个文件都一个独立的版本号，你有么？

Git：每个文件一个版本号？这有什么值得夸耀的？我听说你最早是用脚本对 RCS 系统进行封装实现的，所以你的每个文件都有一个独立的版本控制，这让你变得很零碎。我听说某些商业版本控制系统也是这样，真糟糕。我的每次提交都有一个全球唯一的版本号，不但在本地版本库中是唯一的，和其他人的版本库也不会有冲突。

CVS：我能一次检出一个目录，你好像不能吧？

Git：所以我有子模组，以及 repo 等第三方工具，可以帮助我把一个大的版本库拆成多个版本库组合来使用啊。而且我还有稀疏检出的功能，只不过很少有人用到罢了。

CVS：我能添加空目录，你好像不能吧！

Git：是的，我现在还不能记录空目录。但是用户可以在空目录下创建一个隐含文件，并将该隐含文件添加到版本库中，这就实现了空目录的添加功能。你，CVS，目录管理是你的软肋，你很难实现目录的重命名，而目录重命名对我来说却是小菜一碟。

B.2 CVS 和 Git 命令对照

比较项目	CVS 命令	GIT 命令
URL	:pserver:user@host:/path/to/cvsroot	git://host/path/to/repos.git
	/path/to/cvsroot	ssh://user@host/path/to/repos.git
		user@host:path/to/repos.git
		file:///path/to/repos.git
		/path/to/repos.git
版本库初始化	cvs -d <path> init	git init [--bare] <path>
导入数据	cvs -d <url> import -m ...	git clone; git add .; git commit
版本库检出	cvs -d <url> checkout [-d <path>] <module>	git clone <url> <path>
版本库分支检出	cvs -d <url> checkout -r <branch> <module>	git clone -b <branch> <url>

工作区更新	cvs update	git pull
更新至历史版本	cvs update -r <rev>	git checkout <commit>
更新到指定日期	cvs update -D <date>	git checkout HEAD@'{<date>}'
更新至最新提交	cvs update -A	git checkout master
切换至里程碑	cvs update -r <tag>	git checkout <tag>
切换至分支	cvs update -r <branch>	git checkout <branch>
还原文件 / 强制覆盖	cvs up -C <path>	git checkout -- <path>
添加文件	cvs add <TextFile>	git add <TextFile>
添加文件（二进制）	cvs add -kb <BinaryFile>	git add <BinaryFile>
删除文件	cvs remove -f <path>	git rm <path>
移动文件	mv <old> <new>; cvs rm <old>; cvs add <new>	git mv <old> <new>
反删除文件	cvs add <path>	git add <path>
工作区差异比较	cvs diff -u	git diff
		git diff --cached

		git diff HEAD
版本间差异比较	cvs diff -u -r <rev1> -r <rev2> <path>	git diff <commit1> <commit2> -- <path>
查看工作区状态	cvs -n up	git status
提交	cvs commit -m "<msg>"	git commit -a -m "<msg>" ; git push
显示提交日志	cvs log <path> | less	git log
逐行追溯	cvs annotate	git blame
显示里程碑 / 分支	cvs status -v	git tag
		git branch
		git show-ref
创建里程碑	cvs tag [-r <rev>] <tagname> .	git tag [-m "<msg>"] <tagname> [<commit>]
删除里程碑	cvs rtag -d <tagname>	git tag -d <tagname>
创建分支	cvs rtag -b -r <rev> -b <branch> <module>	git branch <branch> <commit>
		git checkout -b <branch> <commit>
删除分支	cvs rtag -d <branch>	git branch -d <branch>
导出项目文件	cvs -d <url> export -r <tag> <module>	git archive -o <output.tar> <tag> <path>
		git archive -o <output.tar> --remote=<url>

		<tag> <path>
分支合并	cvs update [-j <start>] -j <end>; cvs commit	git merge <branch>
显示文件列表	cvs ls	git ls-files
	cvs -d <url> rls -r <rev>	git ls-tree <commit>
更改提交说明	cvs admin -m <rev>:<msg> <path>	git commit --amend
撤消提交	cvs admin -o <range> <path>	git reset [--soft | --hard] HEAD^
杂项	.cvsignore 文件	.gitignore 文件
	参数 -kb 设置二进制模式	-text 属性
	参数 -kv 开启关键字扩展	export-subst 属性

附录C Git 与 SVN 面对面

C.1 面对面访谈录

Git：我的提交历史本身就是一幅美丽的图画 —— DAG（Directed Acylic Graph，有向无环图），可以看到各个分支之间的合并关系。而你 SVN，你的提交历史怎么是一条直线呢？要是在重症监护室看到你，还以为你挂掉了呢？

SVN：我觉得挺好，至少我每次提交会有一个全局的版本号，而且我的版本号是递增的。你的版本号不是递增的吧！

Git：你说的对，我的版本号不是一个简单递增的数字，而是一个长达 40 位的十六进制数字（哈希值），但是可以使用短格式，只要不冲突。虽然我的提交编号看起来似乎是无序的，但实际上我的每一个提交都记录了父提交甚至是双亲或多亲提交，因此可以很容易地从任意一个提交开始建立一条指向历史提交的跟踪链。

SVN：是啊，我的一个提交和前一个提交有时根本没有关系，例如一个提交是发生在主线 `/trunk` 中的，下一个提交可能就发生在 `/branches/1.3.x` 分支中。你要知道，要想画出一个像你那样的分支图，我要做多少工作么？我不容易呀。

Git：我一直很奇怪，你的分支和里程碑怎么看起来和目录一样？我的分支和里程碑名字虽然看起来像是目录，但实际上和工作区的目录完全没有关系，只是对提交 ID 的一个记号而已。

SVN：我一开始觉得我用轻量级拷贝的方式实现分支和里程碑会很酷，也很快。但是我发现很多人在使用我的时候，直接在版本库的根目录下创建文件而不是把文件创建在 `/trunk` 目录下，这就导致这些人无法再创建分支和里程碑了，因为无法将根目录拷贝到子目录呀！

Git：那么你是如何对分支合并进行跟踪的呢？因为我有 DAG 的提交关系图，很容易就可以看出分支之间的合并历史，但是你是怎么做到的呢？

SVN：我用了一点小技巧，通过属性（`svn:mergeinfo`）记录了合并的分支名和版本范围，这样再合并的时候，我会根据相关属性确定是否要合并。但是如果经常在子目录下合并，有太多的 `svn:mergeinfo` 属性等待我检查，我会很困扰。还有我的这个功能是在 1.5 以后的版本才提供的，因此老版本会破坏这个机制。

SVN：对了，我的属性能干很多事哦，我甚至可以把我的照片作为属性附加在文

件上。

Git：这点我承认，你的属性非常强大。其实我也支持属性，只不过实现方式不同罢了。而且我可以通过评注的方式为任意对象（提交、文件、里程碑等）添加评注，也可以实现把照片作为评注附加在文件上，可是这个功能有什么实际用处么？

SVN：我有轻量级拷贝，而我的分支和里程碑就是通过拷贝实现的，很强大哦。

Git：我根本就不需要轻量级拷贝，因为我对文件的保存和文件的路径无关，我只关心内容。所以相同内容的文件无论它们的文件名相差有多大，在我这里只保存一份。而你 SVN，如果用户忘了用轻量级拷贝，版本库是不是负担很重啊。

SVN：听说你不能针对目录授权，这可是我的强项，所以公司无论大小都在用我作为版本控制系统。

Git：不要说你的授权了，简直是一团糟。虽然这本书的作者为你写了一个图形化的授权管理工具[1]，对你的授权会有所改善，但是你糟糕的分支和里程碑的实现，会导致授权在新的分支和里程碑中又要逐一进行设置，工作量其大无比。虽然泛路径授权是一个解决方案，但是官方并没有提供啊。

Git：说说我的授权吧。如果你认真地读过本书服务器架设的相关章节，你会为我能够提供按照分支，以及按照路径进行写操作授权的功能而击掌叫好的。当然我的读操作授权还不能做到很精细，但是可以将版本库拆分成若干个小的版本库啊，再参照本书介绍的各种多版本库协同模式，也会找到一个适合的解决方案的啊。

Git：我的工作区很干净。只在工作区的根目录下有一个 `.git` 目录，此外再无其他。

SVN：我要在工作区的每一个目录下都放置一个 `.svn` 目录，这个目录在 Linux 下可是隐藏的哦。这个目录下不但有跟踪工作区文件状态的跟踪文件，而且还有每一个文件的原始拷贝呢。这样有的操作就可以脱离网络执行了，例如：差异比较、工作区文件的回滚。

Git：嗯，你要是像我一样能再多保存一点内容（整个版本库）就更好了。像你这样在每个工作区子目录下都有一个 `.svn` 目录，而且每个 `.svn` 目录下都有文件的原始拷贝，在进行内容搜索的时候会搜索出两份吧，太干扰了。而且你这么做和 CVS 一样有安全风险，造成本地文件名的信息泄漏，千万不要在 Web 服务器上用 SVN 检出哦。

Git：我的操作可以不需要网络。因为我在本地拥有完整的版本库，几乎所有操作都是在本地

[1] http://www.ossxp.com/doc/pysvnmanager/user-guide/user-guide.html

完成的。

SVN：正如前面说到的，我有部分命令可以不需要网络，但是其他绝大多数命令还是要依赖网络的。

SVN：你怎么没有更新（*update*）命令啊？还有你为什么老是要执行检出命令（*checkout*）？对我而言，检出命令只在工作区创建时一次性完成。

Git：你的这个问题怎么和 CVS 问的一样。你的更新（*update*）命令执行的很慢，对么？首先你要用检出命令（*checkout*）建立工作区，然后你要经常执行更新（*update*）命令进行更新，否则很容易造成你的更改和他人的更改发生冲突。

Git：之所以你需要更新是因为你的版本库在远程啊。别忘了我的版本库是在本地，本地的版本库会随着我在本地工作区中的操作（如提交）而更新。我的检出（*checkout*）操作一般用于用户切换分支，或者从本地版本库检出丢失的文件或覆盖本地错误改动的文件。如果我没记错的话，你切换分支用的是 *svn switch* 命令对么？

Git：实际上我也有一个比较耗时的网络操作命令： *git fetch* 或 *git pull* ，这两个操作是从远程版本库获取他人的改动。一般使用我（Git）做团队协作的时候，会部署一个集中共享的版本库，我就从这个共享的版本库执行拉回操作。也许你（SVN）会觉得 *git fetch* 或 *git pull* 和你的 *svn update* 命令更像吧。至于你的检出命令（*svn checkout*），实际上和我克隆命令（*git clone*）很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

SVN：为什么你的检入（*commit*）命令执行得那么快？

Git：是的，我的检入命令飞一般就执行完了，这也是因为我的版本库就在本地。也许你（SVN）会觉得我的推送命令（*git push*）和你的检入命令（*svn commit*）更相像，其实这是一个误会。如果我不做本地提交，是不能通过推送命令（*git push*）将我的本地提交共享给（推送给）其他版本库的。你（SVN）每一次的提交都要和版本库进行网络通讯，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次 *git push*，将我本地版本库中的新提交推送给远程版本库。

SVN：我能一次检出一个目录，你好像不能吧？

Git：所以我有子模组，以及 repo 等第三方工具，可以帮助我把一个大的版本库拆成多个版本库组合来使用啊。而且我还有稀疏检出的功能，只不过很少有人用到罢了。

SVN：我能添加空目录，你好像不能吧！

Git：是的，我现在还不能记录空目录，但是用户往往在空目录下创建一个隐含文件，并将该

隐含文件添加到版本库中，这就实现了空目录添加的功能。

C.2 SVN 和 Git 命令对照

比较项目	SVN 命令	GIT 命令
URL	svn://host/path/to/repos	git://host/path/to/repos.git
	https://host/path/to/repos	ssh://user@host/path/to/repos.git
	file:///path/to/repos	user@host:path/to/repos.git
		file:///path/to/repos.git
		/path/to/repos.git
版本库初始化	svnadmin create <path>	git init [--bare] <path>
导入数据	svn import <path> <url> -m ...	git clone; git add .; git commit
版本库检出	svn checkout <url/of/trunk> <path>	git clone <url> <path>
版本库分支检出	svn checkout <url/of/branches/name> <path>	git clone -b <branch> <url> <path>
工作区更新	svn update	git pull
更新至历史版本	svn update -r <rev>	git checkout <commit>
更新到指定日期	svn update -r {<date>}	git checkout HEAD@'{<date>}'
更新至最新提交	svn update -r HEAD	git checkout master

切换至里程碑	svn switch <url/of/tags/name>	git checkout <tag>
切换至分支	svn switch <url/of/branches/name>	git checkout <branch>
还原文件 / 强制覆盖	svn revert <path>	git checkout -- <path>
添加文件	svn add <path>	git add <path>
删除文件	svn rm <path>	git rm <path>
移动文件	svn mv <old> <new>	git mv <old> <new>
清除未跟踪文件	svn status | sed -e "s/^?//" | xargs rm	git clean
清除工作锁定	svn clean	-
获取文件历史版本	svn cat -r<rev> <url/of/file>@<rev> > <output>	git show <commit>:<path> > <output>
反删除文件	svn cp -r<rev> <url/of/file>@<rev> <path>	git add <path>
工作区差异比较	svn diff	git diff
		git diff --cached
		git diff HEAD
版本间差异比较	svn diff -r <rev1>:<rev2> <path>	git diff <commit1> <commit2> -- <path>

查看工作区状态	svn status	git status -s
提交	svn commit -m "<msg>"	git commit -a -m "<msg>" ; git push
显示提交日志	svn log | less	git log
逐行追溯	svn blame	git blame
显示里程碑 / 分支	svn ls <url/of/tags/>	git tag
	svn ls <url/of/branches/>	git branch
		git show-ref
创建里程碑	svn cp <url/of/trunk/> <url/of/tags/name>	git tag [-m "<msg>"] <tagname> [<commit>]
删除里程碑	svn rm <url/of/tags/name>	git tag -d <tagname>
创建分支	svn cp <url/of/trunk/> <url/of/branches/name>	git branch <branch> <commit>
		git checkout -b <branch> <commit>
删除分支	svn rm <url/of/branches/name>	git branch -d <branch>
导出项目文件	svn export -r <rev> <path> <output/path>	git archive -o <output.tar> <commit>
	svn export -r <rev> <url> <output/path>	git archive -o <output.tar> --remote=<url> <commit>
反转提交	svn merge -c -<rev>	git revert <commit>

提交拣选	svn merge -c <rev>	git cherry-pick <commit>
分支合并	svn merge <url/of/branch>	git merge <branch>
冲突解决	svn resolve --accept=<ARG> <path>	git mergetool
	svn resolved <path>	git add <path>
显示文件列表	svn ls	git ls-files
	svn ls <url> -r <rev>	git ls-tree <commit>
更改提交说明	svn ps --revprop -r<rev> svn:log "<msg>"	git commit --amend
撤消提交	svnadmin dump、svnadmin load 及 svndumpfilter	git reset [--soft | --hard] HEAD^
属性	svn:ignore	.gitignore 文件
	svn:mime-type	text 属性
	svn:eol-style	eol 属性
	svn:externals	git submodule 命令
	svn:keywords	export-subst 属性

附录D Git 与 Hg 面对面

D.1 面对面访谈录

Git：你好 Hg，我发现我们真的很像。

Hg：是啊，人们把我们都归类为分布式版本控制工具，所以我们之间的相似度，要比和 CVS、SVN 的相似度高得多了。

Hg：我是用 Python 和少部分的 C 语言实现的，你呢？

Git：我的核心当然是使用 C 语言了，因为 Linus Torvalds 最爱用 C 语言了。我的很多命令还使用了 Shell 脚本和 Perl 语言开发，Python 用的很少。

Hg：大量使用 C 语言，是你的性能比我高的原因么？

Git：当然不是了，你不也在核心模块使用 C 语言了么？问题的关键在于我的对象库设计得非常优秀。你不要忘了我是谁发明的，那可是大名鼎鼎的 Linux 之父 Linus Torvalds 啊，他对 Linux 文件系统可是再熟悉不过的了，所以他能够以文件系统开发者的视角来实现我的核心。

Git：还有我的网络传输过程非常直观，可以显示实时的进度，好像我从你那里没有看到。之所以我能够有这样的实现，是因为我使用了“智能协议”。在网络传输的两端都启用了相应的辅助程序，实现差异传输及传输进度的计算和显示。

Hg：实际上我也支持进度显示[1]，不过是通过 Progress 插件[2]实现的，需要通过修改配置文件启用该插件。

Hg：我有一个特点是 SVN 用户非常喜欢的，就是我的顺序数字版本号。

Git：你的顺序数字版本号只在本地版本库中有效。也就是说，你不能像 SVN 那样将顺序数字版本号作为项目本身的版本号，因为换成另外一个版本库的克隆，那个数字版本号就会不一样了。

Hg：我觉得你的暂存区（stage）的概念太古怪了。我提交的时候，改动的文件会直接提交而不需要进行什么注册到暂存区的操作。

Git：让读者来做评判吧。如果读者读过本书的第 2 篇，一定会说 Git 的暂存区帅呆了。

Hg：我只允许用户对最近的一次提交进行回滚撤销，而你（Git）怎么能允许用户撤销任意多次历史提交呢？那样安全么？

[1] 感谢来自台湾的 Willie Wu 对我博客的评论。

[2] http://mercurial.selenic.com/wiki/ProgressExtension

Git：这就是我的对象库和引用设计的强大之处，我可以使用 *git reset* 命令将工作分支进行任意的重置，丢弃任意多的历史。至于安全性，我的重置命令有一个保险： *reflog* ，我随时可以参照 *reflog* 的记录来弥补错误的重置。

Hg：我们的 *revert* 命令好像不同？

Git：你 Hg 的 *hg revert* 命令和 SVN 的 *svn revert* 命令相似，是取消本地修改，用原始拷贝覆盖。你的这个操作在我这里是用 *git checkout* 命令来实现的。我也有一个 *git revert* 命令，但是这个命令是针对某个历史提交进行的反向操作，以取消该历史提交的改动的。

Hg：我执行日志查看能够看到文本显示的分支图，你呢？

Git：我需要在日志显示时添加参数，即使用命令 *git log --graph* 。我支持通过建立别名来实现简洁的调用，例如建立一个名为 *glog* 的别名。

Git：我听说你 Hg 不支持分支？

Hg：你说的是昨天的我，现在有了 Bookmarks 插件[1]，我也拥有和你类似的分支实现。不过传统来讲我还是以克隆来实现分支的。

Git：实际上我的每一个克隆的版本库也相当于独立的分支，但是因为我有强大的分支功能，因此很多用户还没有意识到。使用 Topgit 的用户就应该使用版本库克隆作为 Topgit 本身的分支管理。

Git：还有，因为我对分支的完整支持，使得我可以和 SVN 很好地协同工作。我可以将整个 SVN 转换为本地的 Git 库，但是你 Hg，显然只能每次转换一个分支。

Hg：是的，我要向你多学习。

D.2 Hg 和 Git 命令对照

比较项目	HG 命令	GIT 命令
URL	http://host/path/to/repos	git://host/path/to/repos.git
	ssh://user@host/path/to/repos	ssh://user@host/path/to/repos.git

[1] http://mercurial.selenic.com/wiki/BookmarksExtension

	file:///path/to/repos	user@host:path/to/repos.git
	/path/to/repos	file:///path/to/repos.git
		/path/to/repos.git
配置		
	[ui] username = Firstname Lastname <mail@addr>	[user] name = Firstname Lastname email = mail@addr
版本库初始化	hg init <path>	git init [--bare] <path>
版本库克隆	hg clone <url> <path>	git clone <url> <path>
获取版本库更新	hg pull --update	git pull
更新至历史版本	hg update -r <rev>	git checkout <commit>
更新到指定日期	hg update -d <date>	git checkout HEAD@'{<date>}'
更新至最新提交	hg update	git checkout master
切换至里程碑	hg update -r <tag>	git checkout <tag>
切换至分支	hg update -r <branch>	git checkout <branch>
还原文件 / 强制覆盖	hg update -C <path>	git checkout -- <path>
添加文件	hg add <path>	git add <path>

删除文件	hg rm <path>	git rm <path>
添加及删除文件	hg addremove	git add -A
移动文件	hg mv <old> <new>	git mv <old> <new>
撤消添加、删除等操作	hg revert <path>	git reset -- <path>
清除未跟踪文件	hg clean	git clean -fd
获取文件历史版本	hg cat -r<rev> <path> > <output>	git show <commit>:<path> > <output>
反删除文件	hg add <path>	git add <path>
工作区差异比较	hg diff	git diff
		git diff --cached
		git diff HEAD
版本间差异比较	hg diff -r <rev1> -r <rev2> <path>	git diff <commit1> <commit2> -- <path>
查看工作区状态	hg status	git status -s
提交	hg commit -m "<msg>"	git commit -a -m "<msg>"
推送提交	hg push	git push
显示提交日志	hg log | less	git log

	hg glog | less	git log --graph
逐行追溯	hg annotate	git annotate, git blame
显示里程碑/分支	hg tags	git tag
	hg branches hg bookmarks	git branch
	hg heads	git show-ref
创建里程碑	hg tag [-m "<msg>"] [-r <rev>] <tagname>	git tag [-m "<msg>"] <tagname> [<commit>]
删除里程碑	hg tag --remove <tagname>	git tag -d <tagname>
创建分支	hg branch <branch> hg bookmark <branch>	git branch <branch> <commit>
		git checkout -b <branch> <commit>
删除分支	hg commit --close-branch hg bookmark -d <branch>	git branch -d <branch>
导出项目文件	hg archive -r <rev> <output.tar.gz>	git archive -o <output.tar> <commit>
		git archive -o <output.tar> --remote=<url> <commit>
反转提交	hg backout <rev>	git revert <commit>
提交拣选	-	git cherry-pick <commit>
分支合并	hg merge <rev>	git merge <commit>

变基	hg rebase	git rebase
冲突解决	hg resolve --tool=<tool>	git mergetool
	hg resolve -m <path>	git add <path>
更改提交说明	Hg + MQ	git commit --amend
撤消最后一次提交	hg rollback	git reset [--soft | --hard] HEAD^
撤消多次提交	Hg + MQ	git reset [--soft | --hard] HEAD~<n>
撤消历史提交	Hg + MQ	git rebase -i <commit>^
启动 Web 浏览	hg serve	git instaweb
二分查找	hg bisect	git bisect
内容搜索	hg grep	git grep
提交导出补丁文件	hg export	git format-patch
工作区根目录	hg root	git rev-parse --show-toplevel
杂项	.hgignore 文件	.gitignore 文件
	pager 扩展	内置分页器
	color 扩展	color.* 配置变量

	mq 扩展	StGit, Topgit
	graphlog 扩展	git log --graph
	hgk 扩展	gitk