

Git原理与命令大全

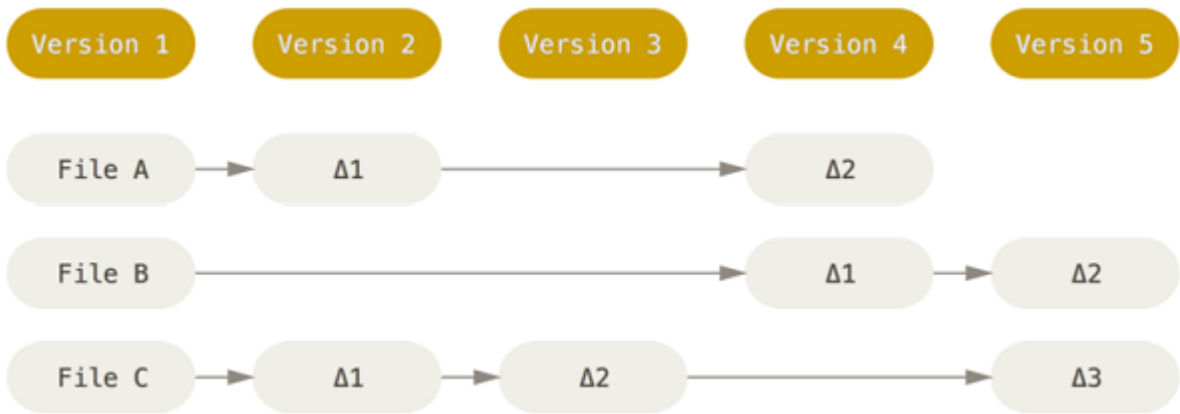
Git (wiki: [en](#) [chs](#)) 是一个免费开源的分布式版本控制系统，由linux内核作者linus Torvalds开发，大型开源项目[linux kernel](#)、[Android](#)、[chromium](#)、[mono](#)、[dotnet](#)、[UE4](#)等都使用Git管理项目

著名github网站使用Git托管所有项目代码，Git的代码也托管在github上，链接为：<https://github.com/git>

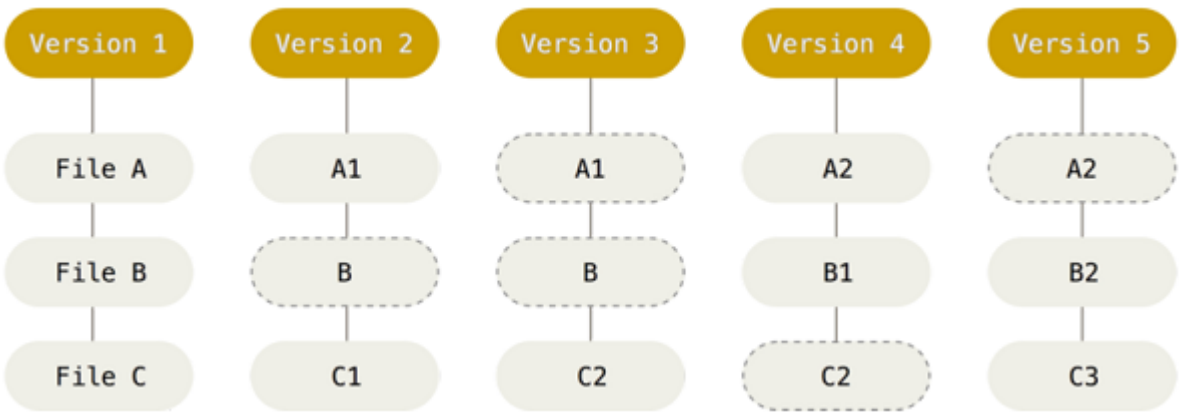
与集中式版本控制系统（开源软件：SVN；免费软件：CVS；商业软件：微软的VSS、IBM的Rational ClearCase）相比

Git优点：

- ① 本地是版本库的完整镜像，因此支持离线工作
- ② 绝大多数操作都只需要访问本地文件和资源，而且与每个提交都是所有文件的完整副本，因此速度非常快



注：SVN等集中式版本控制系统存储每个文件与初始化版本的差异



注：Git每个提交都是所有文件的完整副本，使得Git在回溯到某个提交时，不会对所有文件执行差异计算还原，因此速度会非常快

③ 强大快捷的分支功能，非常适合非线性开发过程

Git缺点：

① 只能全量整体，而不能以子目录和分支为单位进行更新、提交等操作

② 子目录和分支不能单独进行权限控制

③ 由于每个提交都是所有文件的完整副本，因此更占磁盘空间

这使得源代码、配置文件等更适合用Git来管理，而资源等较大的二进制文件则容易导致版本库体积膨胀

在项目实践中，对于资源等较大的二进制文件可以采用Git-LFS来管理，UE4则是使用自己开发的GitDependencies来管理

基本概念

origin：默认远程版本库名

master：默认分支名

origin/master：远程默认分支名

HEAD：当前分支顶端Commit的别名，即当前分支最近的一个提交的SHA-1哈希值

ORIG_HEAD：上次HEAD指针的位置。注：当执行git reset/git pull/git merge命令时，git会把老的HEAD拷贝到文件.git/ORIG_HEAD中，在后续命令中可以使用ORIG_HEAD引用这个提交

commit（提交）：每个commit都是全部文件的完整快照，并用一个 commitID（基于文件的内容或目录结构计算出来的40位十六进制的SHA-1哈希值）来唯一标志。从某个角度上来说，Git维护的就是一个commitID有向无环图

detached HEAD：HEAD没有指向任何分支的状态。一般有以下几种情况会出现这种情况：

① 使用checkout命令跳到某个没有分支指着的commit时

② rebase处理冲突时所处的状态

③ 切换到某个远程分支cache上时

在Git中，在执行命令时，一定要清楚：你在哪？对谁执行这个命令？

本文使用git版本为：git version 2.13.0.windows

运行命令行建议使用：git bash（可通过右键菜单 Git Bash here来启动），主要有3个原因：

- ① 在windows的cmd下执行git log等需要显示多页内容的命令时，会导致cmd卡死（有时按Q键也没法退出）
- ② git bash中可以使用MinGW中自带的linux环境下常用的命令工具
- ③ git bash着色做得更好，利于阅读

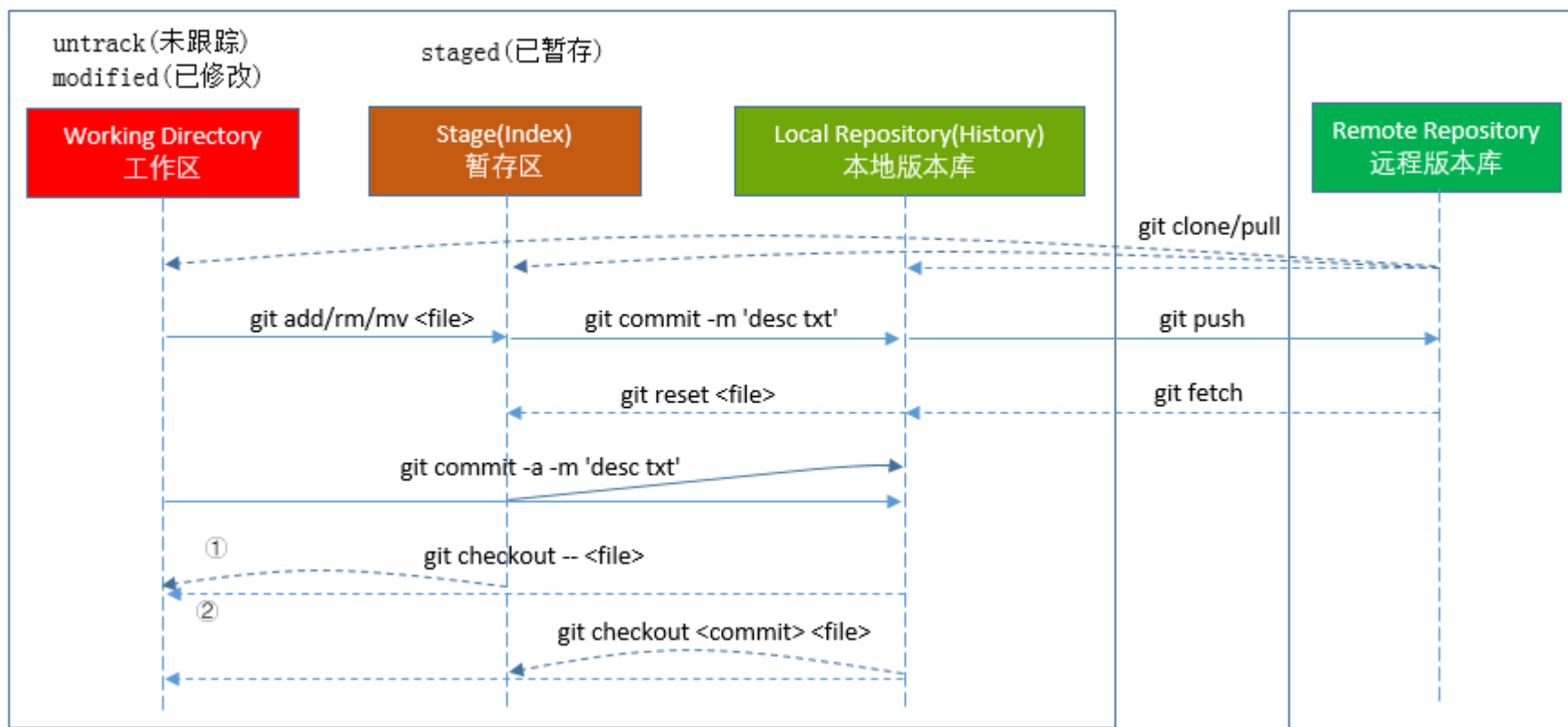
图解常见操作

Working Directory：即工作区。操作系统层面的目录树结构，也可以理解为一个tree目录对象

Stage(Index)：即暂存区，为等待Commit的文件列表。是以扁平的文件清单实现的，不过从理解层面上也可以理解为tree目录对象

Local Repository(History)：本地版本库。有向无环图，其每一个节点都是一个tree目录对象

Remote Repository：远程版本库。有向无环图，其每一个节点都是一个tree目录对象



注：图中的git checkout -- <file>①②步骤的含义是当在暂存区中有修改时，优先使用暂存区中的修改覆盖工作区

svn命令对比一览

svn	git	说明
svn checkout	git clone	检出项目
svn update	git fetch git pull	更新
svn commit	git commit git push	提交
svn add	git add	添加
svn mv	git mv	移动
svn rm	git rm	删除
svn status	git status	查看状态
svn log	git log	查看log
svn diff	git diff	查看差异
svn revert	git checkout git reset git revert	撤销、丢弃修改
svn copy	git checkout -b/-B git branch	创建分支

svn switch	git checkout	切换分支
svn copy	git tag	创建tag
svn merge	git merge git rebase	分支合并

文件存储机制

Git存储使用的是一个内容寻址的文件系统，其核心部分是一个简单的键值对（key-value）数据库，当向数据库中插入任意类型的内容，它会返回一个40位十六进制的SHA-1哈希值用作索引。

在版本库中，Git维护的数据结构有：以下4种对象及索引，并通过保存commitID有向无环图的log日志来维护与管理项目的修订版本和历史信息。

blob -- 1个blob保存1个文件的1个版本的数据

tree -- 表示1个目录，记录着目录里所有文件blob哈希值、文件名子目录名及其他元数据。通过递归引用其他目录树，从而建立一个包含文件和子目录的完整层次结构

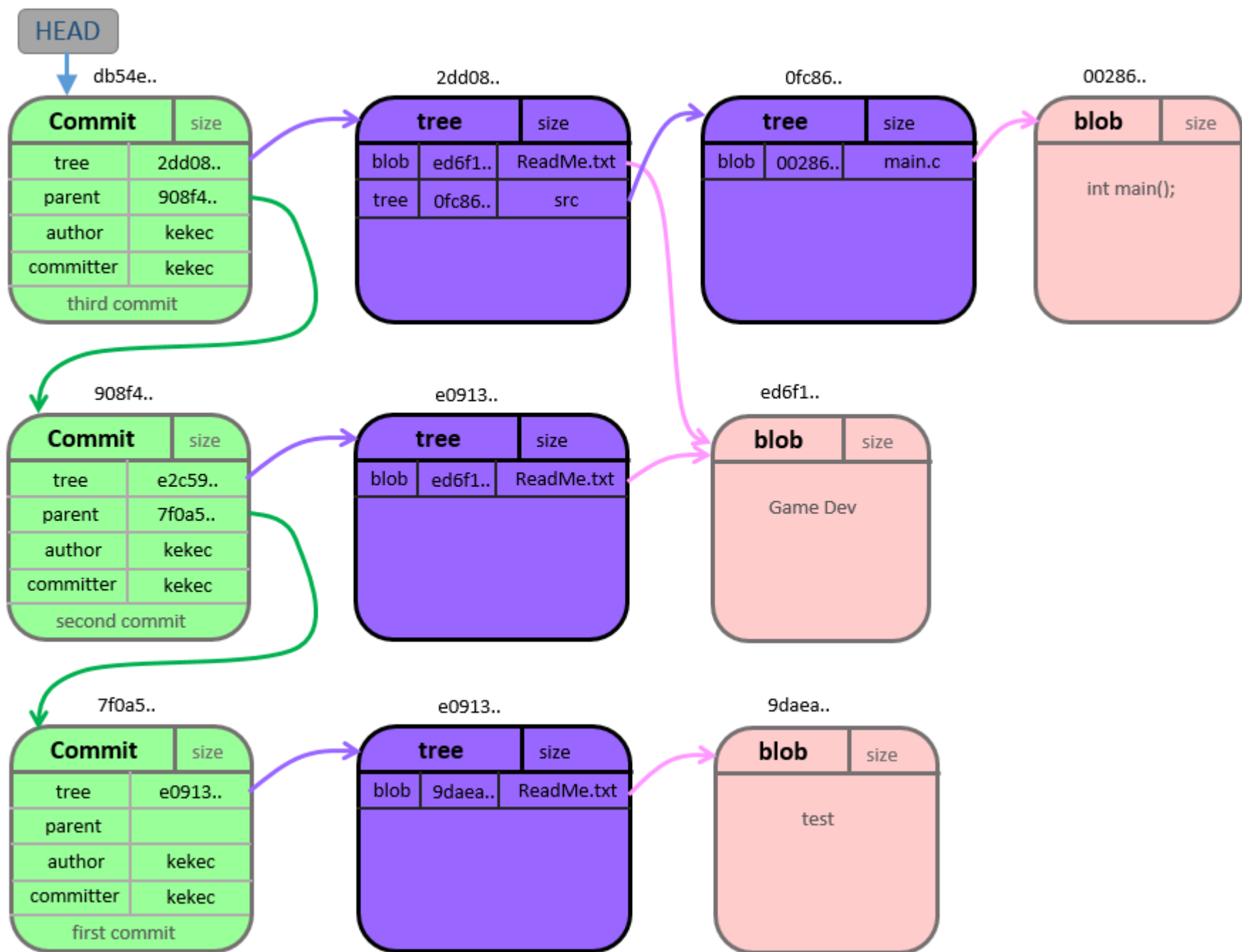
commit -- 1个提交对象保存版本库中每一次变化的元数据，每个提交对象指向一个版本的git目录树对象

tag -- 分为轻量标签和附注标签。轻量标签实际上是一个特定提交的引用，附注标签是存储在git中的一个完整可被校验的对象（保存在.git/refs/tags中），还包含打标签者的名字、e-mail、日志、注释等信息

git使用zlib将头部信息（对象类型：blob或tree或commit + 1个空格 + 数据内容长度 + 1个空字节）和对象数据拼接一起的内容进行压缩存储成一个文件

压缩的文件被十六进制的SHA-1哈希值命名，该文件可以用pigz.exe -dz < 文件路径来解压查看。注：windows版的[pigz.exe](#)可以从这儿 [下载](#)

40位十六进制的SHA-1哈希值 = sha1("blob/tree/commit " + filesize + "\0" + data) 如：sha1("blob 7\0foobar\n") = "323fae03f4606ea9991df8befbb2fca795e648fa" 注：\n的二进制为0a



底层命令 -- 剖析Git对象

find .git/objects -type f // 用find命令查看.git/objects目录（递归子目录）中的所有文件

git rev-list --objects --all // 查看所有git对象的SHA-1哈希值与文件名的对应关系

git rev-list --objects --all | grep 83c4fbc43a6f187d4e8a247a1c9aced872b2315d // 查看SHA-1哈希值为83c4fbc43a6f187d4e8a247a1c9aced872b2315d的文件名

echo "Hello World!" | git hash-object --stdin // 计算内容为Hello World!文件的SHA-1哈希值

echo "Hello World!" | git hash-object -w --stdin // 计算内容为Hello World!文件的SHA-1哈希值并写入到当前git本地版本库中

git hash-object README.txt // 查看README.txt的SHA-1哈希值

git hash-object -w README.txt // 查看README.txt的SHA-1哈希值并写入到当前git本地版本库中

git cat-file -p master^{tree} // 查看master分支HEAD指针git目录（tree对象）下的各子目录（tree对象）和文件（blob对象）的SHA-1哈希值

```
100644 blob 7abd3a56703ad4a7120571967f5d06607b5e5502 README.txt
040000 tree 9f448c40e684dc38109574007c661277c815fb7e ss
```

注：040000：表示目录 100644：表示一般文件 100755：表示可执行文件 120000：表示符号链接

git cat-file -p 7abd3a56703ad4a7120571967f5d06607b5e5502 // 查看SHA-1哈希值为7abd3a56703ad4a7120571967f5d06607b5e5502文件的内容

git show 7abd3a56703ad4a7120571967f5d06607b5e5502 // 查看SHA-1哈希值为7abd3a56703ad4a7120571967f5d06607b5e5502文件的内容

git cat-file -t f3961f5 // 查看f3961f5提交对象的类型：显示为commit

git cat-file -p f3961f5 // 查看f3961f5提交对象的信息：包含git目录（tree对象）、上次提交对象的SHA-1哈希值及提交时Author、Date和注释信息

```
tree ead34240822030a3f71df4fc351057d80d7d83f8
parent 33d5bbc5d61b024aab5078e40548c4e3da808e0e
author kekec <kekec@qq.com> 1537258258 +0800
committer kekec <kekec@qq.com> 1537258258 +0800
```

```
123 desc txt
```

git cat-file -p tag1.0 // 查看轻量标签或附注标签tag1.0信息

git cat-file tag tag1.0 // 查看附注标签tag1.0信息

git ls-tree ead34240822030a3f71df4fc351057d80d7d83f8 // 查看tree目录对象ead34240822030a3f71df4fc351057d80d7d83f8中包含的blob文件对象和tree目录对象

git ls-tree HEAD // 查看HEAD所指向tree目录对象中包含的blob文件对象和tree目录对象

git verify-pack -v .git/objects/pack/pack-a9282552b62cbe3f255fbb20374695a17c1ba2a2.idx // 查看pack-a9282552b62cbe3f255fbb20374695a17c1ba2a2.pack压缩包中的内容

git update-index n.txt // 将修改状态的n.txt文件添加到暂存区

git update-index --add n.txt // 将未追踪状态或修改状态的n.txt文件添加到暂存区

git update-index --add --cacheinfo 100644 5d11580eed65ffd34b6786274a60460b3582aa7d n.txt // 使用类型为100644、SHA-1哈希值为5d11580eed65ffd34b6786274a60460b3582aa7d的信息将追踪状态或修改状态的n.txt添加到暂存区

git write-tree // 将整个暂存区内容生成一个tree对象，并输出其SHA-1哈希值

echo "add n.txt" | git commit-tree 31b7ca405196ca9e8fb4d5404b315bef9f2c841f -p HEAD // 用git write-tree得到的31b7ca405196ca9e8fb4d5404b315bef9f2c841f树对象创建一个注释为add n.txt的提交对象，并将提交对象的父亲设置为当前HEAD

git update-ref refs/heads/master 372aa8e425b57ca30e2974b8e7737133caaa0b7f // 若当前分支为master，更新HEAD指向上面git commit-tree命令得到的372aa8e425b57ca30e2974b8e7737133caaa0b7f提交对象，此时用git log就可以看到这条commit记录

git write-tree --prefix=ss // 将暂存区中ss目录下的内容 生成一个tree对象，并输出其SHA-1哈希值

git update-ref -d refs/remotes/origin/v1.0 // 删除v1.0远程分支cache

git update-index --chmod=+x engine_mac.sh // 为engine_mac.sh增加可执行权限（linux、unix、mac os x系统上需要）

命令大全

查看命令帮助

git config --help // 查看git config命令详细用法

git help config // 功能同上

配置

git config --global user.name "kekec" // 配置提交用户名

git config --global user.email "kekec@qq.com" // 配置e-mail信息

git config --global core.editor vim // 配置默认文本编辑器，当Git需要你输入信息时会调用它

git config --global alias.st status // 为status配置别名st，这样git status就可以写成git st

git config --list // 查看当前仓库的所有配置信息（包括分支相关的信息）

git config user.name // 查看当前仓库的用户名信息

git config -e --global // 编辑全局配置文件（用户名和e-mail信息就记录在其中）所在目录：c:/users/<用户名>/ .gitconfig

git config -e // 编辑当前仓库的配置文件 所在目录：.git\config

创建版本库

git init // 在当前目录创建一个空的git代码库

git init MyGame // 在当前目录创建一个名为MyGame的文件夹，然后在其中创建一个空的git代码库

.git目录结构如下：

名字 (N)	大小 (Z)	修改 (M)
hooks	15,321	2018/9/18 10:20:49
■ applypatch-msg.sample	478	2018/9/18 10:20:49
■ commit-msg.sample	896	2018/9/18 10:20:49
■ post-update.sample	189	2018/9/18 10:20:49
■ pre-applypatch.sample	424	2018/9/18 10:20:49
■ pre-commit.sample	1,642	2018/9/18 10:20:49
■ prepare-commit-msg.sample	1,239	2018/9/18 10:20:49
■ pre-push.sample	1,348	2018/9/18 10:20:49
■ pre-rebase.sample	4,951	2018/9/18 10:20:49
■ pre-receive.sample	544	2018/9/18 10:20:49
■ update.sample	3,610	2018/9/18 10:20:49
info	240	2018/9/18 10:20:49
■ exclude	240	2018/9/18 10:20:49
logs	7,465	2018/9/18 10:20:52
refs	2,401	2018/9/19 10:54:30
heads	1,747	2018/9/18 10:26:23
■ master	1,430	2018/9/18 16:10:58
■ x3.2	317	2018/9/18 11:10:35
remotes	330	2018/9/18 10:20:52
origin	330	2018/9/18 10:39:14
■ HEAD	177	2018/9/18 10:20:52
■ master	153	2018/9/18 10:39:14
■ stash	324	2018/9/19 10:55:55
■ HEAD	5,064	2018/9/19 10:55:58
objects	11,086	2018/9/18 20:15:22
00	55	2018/9/18 11:16:42
■ 0d2a09e0e7ddf6d11430995d111231a7436d4d (r)	55	2018/9/18 11:16:42
01	84	2018/9/18 15:02:43
■ 23a06b4579334cc560c458a48aa8ff06b64517 (r)	84	2018/9/18 15:02:43
info		

└─ packs	54	2018/10/9 10:58:08
└─ pack	51,749	2018/10/9 10:58:08
└─ pack-b340eea7566df839294b71ec91a327ca2ece0b94.idx (r)	3,368	2018/10/9 10:58:08
└─ pack-b340eea7566df839294b71ec91a327ca2ece0b94.pack (r)	48,381	2018/10/9 10:58:08
refs	196	2018/9/18 15:14:13
└─ heads	82	2018/9/18 16:10:58
└─ master	41	2018/9/18 16:10:58
└─ x3.2	41	2018/9/18 11:10:35
└─ remotes	73	2018/9/18 10:20:52
└─ origin	73	2018/9/18 10:39:14
└─ HEAD	32	2018/9/18 10:20:52
└─ master	41	2018/9/18 10:39:14
└─ tags	41	2018/9/18 20:15:22
└─ tag3.0	41	2018/9/18 20:15:22
COMMIT_EDITMSG	8	2018/9/18 16:10:58
config	354	2018/9/18 11:17:49
description	73	2018/9/18 10:20:49
FETCH_HEAD	193	2018/9/18 10:39:14
HEAD	23	2018/9/18 11:17:49
index	252	2018/9/18 16:10:58
ORIG_HEAD	41	2018/9/18 15:55:02
packed-refs	289	2018/9/18 10:20:52

hooks：不同操作时执行的hook脚本

info/exclude：与.gitignore文件（该文件需放在.git文件夹的同级目录中，windows下可通过命令行type nul > .gitignore来创建）一样，用作文件过滤。不同的是：该文件不会提交到版本库，因此过滤只对本地生效，不影响其他人



```
# 忽略所有 .so 结尾的文件
*.so
# 但 game.so 除外
!game.so
# 仅仅忽略项目根目录下的 README.md 文件，不包括 subdir/README.md
/README.md
# 忽略 .svn/ 目录下的所有文件
.svn/
```

```
# 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
doc/*.txt
# 忽略 doc/ 目录下所有扩展名为 txt 的文件
doc/**/*.txt
```



logs/refs/heads：各个本地分支的版本log记录

logs/refs/remotes：各个远程分支cache的log记录

logs/refs/stash：储藏区数据

logs/HEAD：git操作记录

objects：2级文件索引（把SHA-1哈希值拆成了：2位+38位），存储commit数据、blob文件数据和tree目录数据

objects/pack：pack文件为存储commit、tree目录及blob文件的压缩数据；idx文件为pack文件中各数据对象的索引

objects/info/packs：该文件记录所有git库的pack文件列表

refs/heads：各个本地分支HEAD

refs/remotes：各个远程分支cache的HEAD

refs/tags：各个附注标签的信息

COMMIT_EDITMSG：上一次提交的注释

config：版本库相关的配置信息

description：仓库描述信息，供gitweb程序使用

index：暂存区相关的信息

HEAD：指向当前分支的最近提交（如：ref: refs/heads/master）

ORIG_HEAD：执行git merge/git pull/git reset操作时，会把调整为新值之前的先前版本的HEAD记录到OERG_HEAD中，用于恢复或回滚之前的状态

FETCH_HEAD：git fech将所有抓取分支的HEAD记录到.git/FETCH_HEAD中

MERGEHEAD：正在合并进HEAD的commit id

packed-refs：远程版本库cache和远程标签cache

日志与文件状态

git reflog // 查看操作记录 注：每条操作记录使用HEAD@{n}来标识

git show HEAD@{5} // 查看索引为5的操作记录的详细信息

git status // 查看当前所处的分支暂存区和工作区的文件（会显示当前所处分支）

注1：处于暂存区的文件状态：staged(已暂存)；处于工作区的文件状态：untrack(未跟踪)、modified(已修改)

注2：工作区中的空目录不会被git追踪

git status -s --ignored // 以简洁模式查看暂存区和工作区的文件（全部显示，不执行文件过滤）

git status -uno // 查看暂存区和工作区的非untrack(未跟踪)状态文件

git status -uall // 查看暂存区和工作区的状态文件（递归子目录显示出里面的文件）

git log // 查看本地版本库提交记录（会显示当前所处分支，HEAD指针指向哪个分支的哪条提交）

git log --stat // 查看本地版本库提交记录（会显示当前所处分支，HEAD指针指向哪个分支的哪条提交和每次提交的文件变更简略统计信息）

git log -- README.md // 查看README.md文件的本地版本库提交记录

git log --graph -- README.md // 以图形化方式查看README.md文件的本地版本库提交记录

git log -p README.md // 查看README.md文件的本地版本库提交记录（显示出每次的修改内容）

git log --grep "test" // 显示注释中含有test字符串的提交

git log --author=kekec // 查看本地版本库中作者为kekec的提交记录

git log -S "SplitPath(FString& str)" // 查看SplitPath(FString& str)内容是什么时候添加到项目中那些文件中去的

git log --since=2.weeks // 查看最近2周的提交记录

git log --since="2 weeks 3 days 2 hours 30 minutes 59 seconds ago" // 查看2周3天2小时30分59秒以前的提交记录

git log --after="2018-10-7" --before="2018-10-12" // 查看2018.10.7~2018.10.12之间的提交记录

git log --since="2018-10-7" --until="2018-10-12" // 功能同上：git log --after="2018-10-7" --before="2018-10-12"

注：--since、--until 标记和 --after、--before 标记分别是等价的

git whatchanged README.md // 查看README.md文件的本地版本库提交记录（包括文件改名）

git log --follow README.md // 功能同上：git whatchanged README.md

git log -3 // 查看最近3条本地版本库提交记录

git log -3 --pretty --oneline // 查看最近3条本地版本库提交记录（简洁模式，一行显示一个提交）

git log --graph --oneline // 以图形化简洁模式查看当前分支的本地版本库提交记录

git log release --graph --oneline // 以图形化简洁模式查看release分支的本地版本库提交记录

git log --graph --oneline --no-merges // 以图形化简洁模式查看当前分支的本地版本库提交记录（过滤merge过来的提交）

git log --graph --oneline --merges // 以图形化简洁模式查看当前分支的本地版本库提交记录（只显示有2个及以上父亲节点的提交）

git log --graph --oneline --name-only // 以图形化简洁模式查看当前分支的本地版本库提交记录（并显示每次提交的文件名称清单）

git log --graph --oneline --name-status // 以图形化简洁模式查看当前分支的本地版本库提交记录（并显示每次提交的文件状态、名称清单）

git log --graph --oneline --stat // 以图形化简洁模式查看当前分支的本地版本库提交记录（并显示每次提交的文件变化统计、各文件名及增删记录）

git log --graph --oneline --shortstat // 以图形化简洁模式查看当前分支的本地版本库提交记录（并显示每次提交的文件变化统计及增删记录）

git log --graph --oneline --decorate --all // 以图形化简洁模式查看所有分支的本地版本库提交记录树

git log --graph --pretty=format:"%H - %an, %ad : %s" // 自定义格式图形化查看所有分支的本地版本库提交记录树

%H 提交对象（commit）的完整哈希字符串

%h 提交对象的简短哈希字符串

%T 树对象（tree）的完整哈希字符串

%t 树对象的简短哈希字符串

%P 父对象（parent）的完整哈希字符串

%p 父对象的简短哈希字符串

%an 作者（author）的名字

%ae 作者的电子邮件地址

%ad 作者修订日期（可以用 --date= 选项定制格式）

%ar 作者修订日期，按多久以前的方式显示

%cn 提交者(committer)的名字

%ce 提交者的电子邮件地址

%cd 提交日期

%cr 提交日期，按多久以前的方式显示

%s 提交说明

git log master..v5.0 // 查看v5.0分支还未合并到master分支上的提交记录列表

git log v5.0..master // 查看master分支还未合并到v5.0分支上的提交记录列表

git log master...v5.0 // git log master..v5.0 + git log v5.0..master

git shortlog -sn // 统计各个提交者的次数

git blame README.md // 显示README.md最近一次的修改信息

git show 3a6c702376168aa15a2f3d7bc98000d07a70d023 README.md // 查看README.md文件的3a6c702376168aa15a2f3d7bc98000d07a70d023提交的修改内容

git show HEAD // 查看最近一次提交的修改内容

git show --name-only HEAD // 查看最近一次提交的文件列表（不显示具体的修改内容）

标签（查看/新建/切换/删除）

git tag // 列出所有的标签

git tag -l 'tag1*' // 列出所有tag1开头的标签

git tag tag1.0 // 创建名为tag1.0的轻量标签

git tag -a tag1.0 -m "tag1.0 desc" // 添加tag1.0 desc注释并创建名为tag1.0的附注标签

git tag tag2.0 abffefc5d82078cbaea7fcbb5106ab0c21cbeba9 // 在abffefc5d82078cbaea7fcbb5106ab0c21cbeba9提交处创建名为tag2.0的轻量标签

git tag -a tag2.0 -m "tag2.0 desc" abffefc // 在abffefc提交处创建名为tag2.0的附注标签

git tag -d tag2.0 // 删除名为tag2.0的标签

git show tag1.0 // 查看名为tag1.0相关的信息

git ls-remote --tags // 查看所有远端的标签

分支（查看/新建/切换/删除）

git branch // 列出所有本地分支

git branch -r // 列出所有远程分支cache

git branch -a // 列出所有本地分支和远程分支cache

git branch -av // 列出所有本地分支和远程分支cache（含简单说明）

git branch -vv // 查看所有本地分支和远程分支cache之间的追踪关系

git branch v1.0 // 在当前分支的HEAD指针下创建名为v1.0的分支（创建完不会切到v1.0分支上）

git branch --track v1.0 origin/v1.0 // 若v1.0分支不存在则先新建，然后将其与远程分支origin/v1.0建立追踪关系 ① 远程分支origin/v1.0要存在，否则命令执行失败 ② 执行完不会切到v1.0分支上

git branch v2.0 372aa8e425b57ca30e2974b8e7737133caaa0b7f // 在372aa8e425b57ca30e2974b8e7737133caaa0b7f提交处创建名为v2.0的分支（创建完不会切到v2.0分支上）

git branch -m v1.0 x1.0 // 将分支v1.0重命名为x1.0

git checkout v1.0 // 切换到v1.0分支上（v1.0分支不存在则命令执行失败）

git checkout -b v1.0 // 创建并切换到v1.0分支上（v1.0分支存在则命令执行失败）

git checkout -B v1.0 // 不存在则创建，并切换到v1.0分支上

git checkout -b v1.0 5a95f2d // 在5a95f2d提交处创建并切换到v1.0的分支上

git checkout -b v1.0 tag1.0 // 在标签tag1.0处创建并切换到v1.0的分支上

git checkout -t origin/v1.0 // 创建并切换到origin/v1.0远程分支cache的名为v1.0本地分支上，并建立两者追踪关系（本地分支v1.0存在则命令执行失败）

git checkout -b x1.0 -t origin/v1.0 // 创建并切换到origin/v1.0远程分支cache的名为x1.0本地分支上，并建立两者追踪关系（本地分支x1.0存在则命令执行失败）

注1：切换分支前，必须处理工作区（未追踪的文件不用处理）和暂存区的修改才能切换成功

注2：切换成功后，工作区会被设置成分支的内容

注3：不允许在远程分支cache上提交，需要创建对应关联的本地分支，然后在本地分支上进行提交

git checkout -f v1.0 // 强制切换到v1.0分支上，丢弃暂存区和工作区中的所有文件的修改（工作区中未追踪的文件不受影响）

git checkout -f -B v1.0 origin/v1.0 // 不存在则创建，强制切换到v1.0分支上，丢弃暂存区和工作区中的所有文件的修改，并将HEAD指向origin/v1.0处（工作区中未追踪的文件不受影响）

git checkout - // 切换到上一次分支

git branch -d v2.0 // 删除名为v2.0的分支（必须先切到其他分支上才能执行删除操作）

git branch -D v2.0 // 强制删除名为v2.0的分支（必须先切到其他分支上才能执行删除操作）

git branch -dr origin/v2.0 // 删除远程分支origin/v2.0 cache

文件（增加/删除/提交/撤销）

git add README.md // 将当前目录下的README.md文件加入到暂存区

git add . // 将当前目录下（递归子目录）所有文件加入到暂存区

git add -u . // 将当前目录下（递归子目录）所有追踪状态的文件加入到暂存区

git add Doc/*.txt // 将当前目录的Doc文件夹下（递归子目录）所有txt后缀的文件加入到暂存区

git rm README.md // 删除工作区文件，并且将这次删除放入暂存区（若README.md在工作区或暂存区中有修改，命令会执行失败）

git rm -f README.md // 强制删除工作区文件，并且将这次删除放入暂存区（即使README.md在工作区或暂存区中有修改，也会执行删除操作）

git rm --cached README.md // 不删除工作区对应的文件，只将README.md删除放入暂存区以供提交

git mv README.md test.md // 将README.md改名为test.md，并且将这个改名放入暂存区

git commit -m "desc" // 添加desc注释并将暂存区中的所有修改提交到本地仓库

git commit README.md -m "desc" // 添加desc注释并将暂存区中的README.md的修改提交到本地仓库

git commit --amend -m "desc" // 添加desc注释使用当前提交覆盖上一次的提交（若上一次提交包含1.txt和2.txt的修改，当前提交只包含1.txt的修改；执行命令后，本地版本库中为本次的1.txt和上一次2.txt）。若没有提交内容，则用来改写上一次提交的日志信息

git commit -m "desc" --amend README.txt // 添加desc注释使用README.txt的当前提交覆盖上一次的提交

git commit -a -m "desc" // 添加desc注释并将工作区和暂存区中的所有修改提交到本地仓库

git commit -am "desc" // 功能同上

git commit -c b5cad94d229e72bd7aff5fe2c6fo22b29c30e7a8 // 拿372aa8e425b57ca30e2974b8e7737133caaa0b7f提交的信息（作者、提交者、注释、时间戳等）来提交当前修改

git reset -- README.md // 丢弃暂存区中的README.md文件的修改

git reset README.md // 功能如上 丢弃暂存区中的README.md文件的修改

git reset b5cad94 README.md // 使用本地版本库b5cad94提交处的README.md版本覆盖暂存区中的README.md

git reset // 丢弃暂存区中的所有文件的修改（工作区不受影响）

git reset --mixed // --mixed为缺省参数，命令与上面git reset一样

git reset --hard // 丢弃暂存区和工作区中的所有文件的修改（工作区中未追踪的文件不受影响）

git reset --soft b5cad94d229e72bd7aff5fe2c6fo22b29c30e7a8 // 仅将当前分支的HEAD指向372aa8e425b57ca30e2974b8e7737133caaa0b7f提交（暂存区和工作区中的所有文件的修改都不丢弃）

git reset --soft HEAD~ // 仅将当前分支的HEAD指向上一次提交（暂存区和工作区中的所有文件的修改都不丢弃）

git reset --soft HEAD~2 // 仅将当前分支的HEAD指向上两次提交（暂存区和工作区中的所有文件的修改都不丢弃）

git reset --merge <commit> // 在被污染的工作区中回滚merge或者pull

```
$ git pull (1)
Auto-merging nitfol
Merge made by recursive.
```



```
nitfol          |    20 +++++----  
...  
$ git reset --merge ORIG_HEAD      (2)
```


(1) 即便你已经在本地更改了一些你的工作区，你也可安全的git pull，前提是你知道将要pull的内容不会覆盖你的工作区中的内容。

(2) git pull完后，你发现这次pull下来的修改不满意，想要回滚到pull之前的状态，我们可以执行git reset --hard ORIG_HEAD，但是这个命令有个副作用就是清空你的工作区，即丢弃你的本地未add的那些改变。


为了避免丢弃工作区中的内容，可以使用git reset --merge ORIG_HEAD，注意其中的--hard 换成了 --merge，这样就可以避免在回滚时清除工作区。

git reset --keep <commit> // 保留工作区并丢弃一些之前的提交

假设你正在编辑一些文件，并且已经提交，接着继续工作，但是现在你发现当前在工作区中的内容应该属于另一个分支，与之前的提交没有什么关系。此时，可以开启一个新的分支，并且保留着工作区中的内容。



```
$ git tag start  
$ git checkout -b branch1  
$ edit  
$ git commit ...                (1)  
$ edit  
$ git checkout -b branch2        (2)  
$ git reset --keep start         (3)
```



(1) 这次是把branch1中的改变提交了。

(2) 此时发现，之前的提交不属于这个分支，此时新建了branch2分支，并切换到了branch2上。

(3) 此时可以用reset --keep把在start之后的提交清除掉，但是保持工作区不变。

git checkout -- README.md // -- 符号非常重，否则就变成了切换到README.md分支了

// 当README.md在暂存区中有修改时，使用暂存区中的修改覆盖工作区中的README.md

// 当README.md不在暂存区中时，使用本地版本库中的HEAD指针处的修改覆盖工作区中的README.md

git checkout -- . // 使用暂存区和本地版本库来恢复当前目录（递归子目录）下的所有文件 注：若暂存区中有修改，优先使用暂存区

git checkout HEAD README.md // 使用本地版本库中的HEAD处提交覆盖暂存区和工作区中的README.md

git checkout 9a387f22ff949fa16336508adc2284384bd6a890 README.md // 使用本地版本库中的9a387f22ff949fa16336508adc2284384bd6a890修改覆盖暂存区和工作区中的README.md

git checkout -b v2.0 tag2.0 // 在名为tag2.0的提交处创建并切换到v2.0分支上（v2.0分支存在则命令执行失败）

git revert --no-edit 3a6c702376168aa15a2f3d7bc98000d07a70d023 // 回滚3a6c702376168aa15a2f3d7bc98000d07a70d023提交，然后提交到本地仓库

git revert HEAD~ // 回滚HEAD的上一次提交，然后会弹出vim环境编辑注释（输入:q直接使用默认注释内容、输入:q!放弃修改使用默认注释内容、输入:x或:wq保存当前修改的注释内容），然后提交到本地仓库

git revert -n HEAD~3 // 回滚掉HEAD~3处的提交，不自动提交到本地仓库

git revert -n HEAD~2..HEAD // 回滚掉(HEAD~2, HEAD]之间的2次提交，不自动提交到本地仓库

注：git reset是把HEAD向后移动来删除提交，而git revert是用一次新的提交来回滚之前的提交（HEAD会继续前进）

查看差异

git diff README.md // 查看当前目录下的README.md在工作区和暂存区之间的差异

git diff --cached README.md // 查看当前目录下的README.md在暂存区和本地仓库最后一次提交之间的差异

git diff --cached 372aa8e425b57ca30e2974b8e7737133caaa0b7f README.md // 查看当前目录下的README.md在暂存区和本地仓库的372aa8e425b57ca30e2974b8e7737133caaa0b7f提交之间的差异

git diff HEAD README.md // 查看当前目录下的README.md在工作区和本地仓库HEAD指针处提交之间的差异

git diff 372aa8e425b57ca30e2974b8e7737133caaa0b7f README.md // 查看当前目录下的README.md在工作区和本地仓库的372aa8e425b57ca30e2974b8e7737133caaa0b7f提交之间的差异

git diff 372aa8e425b57ca30e2974b8e7737133caaa0b7f HEAD README.md // 查看当前目录下的README.md在本地仓库的372aa8e425b57ca30e2974b8e7737133caaa0b7f提交和最后一次提交之间的差异

git diff 372aa8e425b57ca30e2974b8e7737133caaa0b7f HEAD // 查看本地仓库的372aa8e425b57ca30e2974b8e7737133caaa0b7f提交和最后一次提交之间的差异

git diff 372aa8e b5cad94 README.md // 查看当前目录下的README.md在本地仓库的372aa8e提交和b5cad94提交之间的差异

注：可以将git diff换成git difftool来使用外部diff工具（可以在c:/users/<用户名>/.gitconfig文件配置beyond compare作为默认的difftool和mergetool）来查看差异

```
[diff]
  tool = bc3
[difftool]
  prompt = false
```

```
[difftool "bc3"]  
cmd = "\"e:/program files (x86)/beyond compare 3/bcomp.exe\" \"$LOCAL\" \"$REMOTE\""
```

分支合并

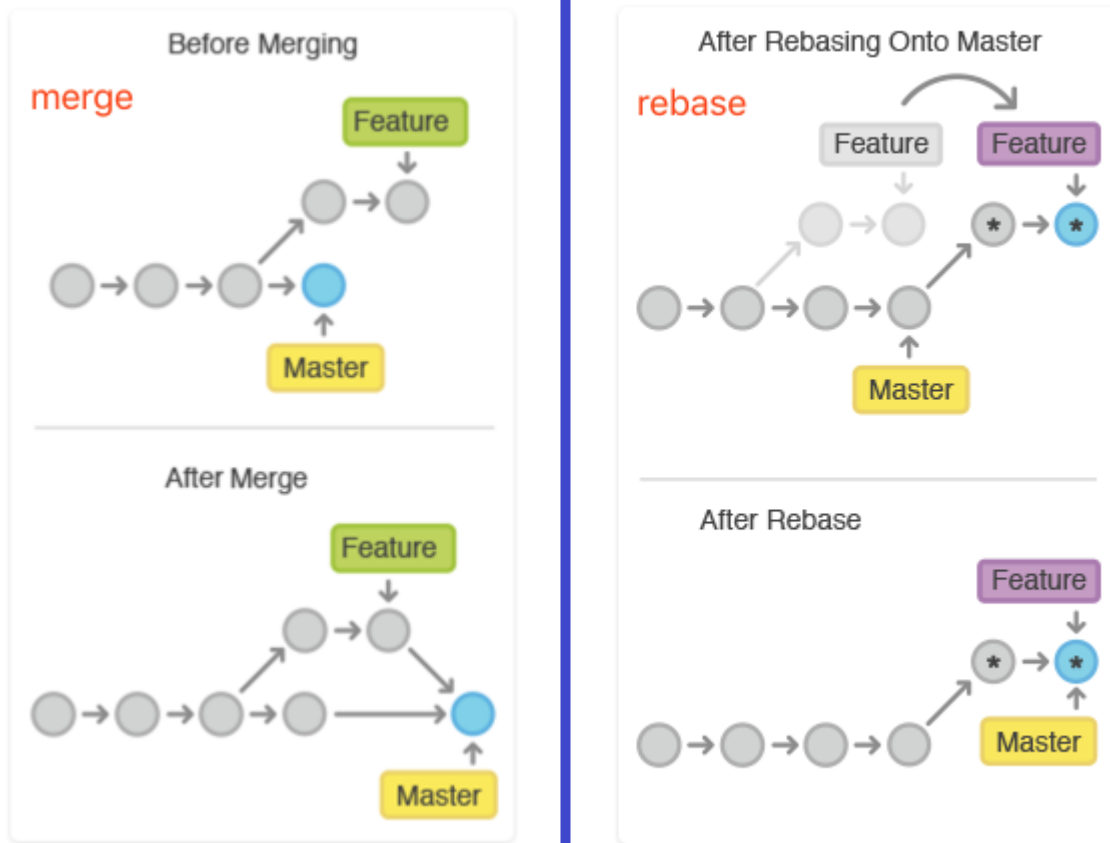
git merge-base Master Feature // 查看Master和Feature分支的最优共同commit父节点

git merge Feature // 将Feature分支merge合并到当前分支Master（无冲突时会直接提交）

git merge -m "merge test" Feature // 将Feature分支merge合并到当前分支Master（无冲突时使用merge test注释直接提交）

git merge --no-commit Feature // 将Feature分支merge合并到当前分支Master（不自动提交）

git rebase Feature // 将Feature分支rebase合并到当前分支Master



注1：git rebase会先找出共同的祖先节点，从祖先节点把Feature分支的提交记录全都剪切下来，然后合到Master分支（合并前后commitID会不一样）

注2：相对来说，git merge处理冲突更直接，但会增加一些冗余的提交记录；而git rebase能够保证清晰线性的提交记录，但这也将合并的操作没有被记录下来

注3：最好是用git rebase合并远程分支到本地，git merge合并Feature分支到Master分支

注4：在合并Feature分支到Master分支前，务必先执行git pull -r origin Feature来进行远程分支与本地分支的rebase合并

注5：处于冲突状态（ conflict ）的文件为UU（ 可通过git status -s --ignored来查找 ），手动处理完冲突后，然后使用git add该文件，最后继续执行git merge/rebase --continue来完成合并的提交工作

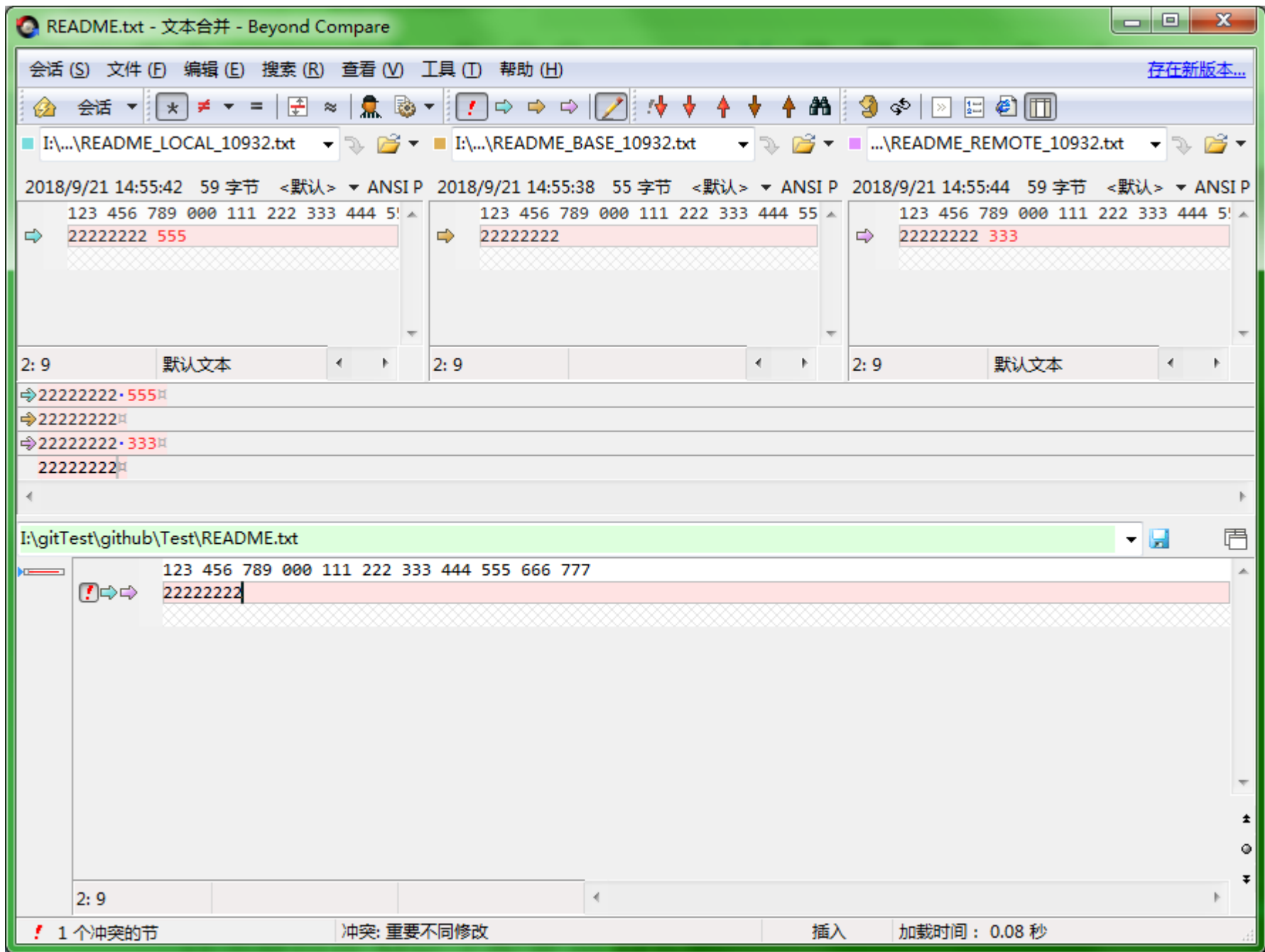
注6：README.md文件冲突内容如下

```
<<<<<< HEAD
123 456 789 000 111 222 333 444 555 ss // 当前分支的内容
=====
123 456 789 000 ss tt // Feature分支的内容
>>>>>> Feature
```

注7：可以使用git mergetool来使用外部merge工具（ 可以在c:/users/<用户名>/.gitconfig文件配置beyond compare作为默认的mergetool ）来处理冲突。

修改完当前文件后，可再次调用git mergetool来处理下一个冲突，直至全部处理完毕，然后使用git add该文件，最后继续执行git merge/rebase --continue来完成合并的提交工作

```
[merge]
  tool = bc3
[mergetool]
  prompt = false
[mergetool "bc3"]
  cmd = "\"e:/program files (x86)/beyond compare 3/bcomp.exe\" \"$LOCAL\" \"$REMOTE\" \"$BASE\" \"$MERGED\""
```



git rebase /i Feature // 将Feature分支采用手动交互方式rebase合并到当前分支Master

pick 07c5abd Introduce OpenPGP and teach basic usage

pick de9b1eb Fix PostChecker::Post#urls

pick 3e7ee36 Hey kids, stop all the highlighting
pick fa20af3 git interactive rebase, squash, amend

```
# Rebase 8db7e8b..fa20af3 onto 8db7e8b
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

git merge/rebase --abort // 撤销当前merge或rebase操作

git merge/rebase --skip // 强制使用Feature分支的内容

git merge/rebase --continue // 手动处理完冲突后使用git add该文件，最后继续执行git merge/rebase --continue来完成合并的提交工作

git merge origin/master // fetch完之后，可以将远程分支cache master分支merge合并到当前分支上

git rebase origin/master // fetch完之后，可以将远程分支cache master分支rebase合并到当前分支上

git rebase --onto master 76cada~ // 将当前分支从[76cada, HEAD]区间段的提交ebase合并到master上

git cherry-pick 9a341e // 将9a341e提交合入当前分支。若不冲突，则直接使用9a341e的提交信息进行commit，否则要先进行冲突处理，然后继续执行git cherry-pick --continue来完成合并的提交工作

git cherry-pick 371c2...971209 // 将[371c2, 971209]提交合入当前分支（每个提交都会在当前分支上创建一个commit）

git cherry-pick 371c2~...971209 // 将[371c2, 971209]提交合入当前分支（每个提交都会在当前分支上创建一个commit）

git cherry-pick -n 9a341e d2f99e // 将9a341e和d2f99e提交合入当前分支（不提交），后续需要手动commit

git cherry-pick --abort // 撤销当前cherry-pick操作

git cherry-pick --quit // 清理当前操作状态，不撤销修改强制退出cherry-pick操作过程

git cherry-pick --continue // 手动处理完冲突后，最后继续执行git cherry-pick --continue来完成合并的提交工作

查看远程版本库

`git remote -v` // 显示远程仓库的URL 注：由于git是分布式的，所有远程仓库可能有很多个

origin https://github.com/kekec/Test.git (fetch)

origin https://github.com/kekec/Test.git (push)

`git remote-ls` // 查看远程仓库URL和分支信息

From https://github.com/kekec/Test.git

fae0fc82d711425daa897a63137d7e1af09512ba HEAD

fae0fc82d711425daa897a63137d7e1af09512ba refs/heads/master

`git remote` // 查看远程仓库名称 一般为origin

`git remote rename origin test` // 将远程仓库名称从origin修改为test

`git remote show origin` // 显示远程仓库的信息

* remote origin

Fetch URL: https://github.com/kekec/Test.git

Push URL: https://github.com/kekec/Test.git

HEAD branch: master

Remote branches:

master tracked

v3.1 tracked

Local branch configured for 'git pull':

master merges with remote master

Local refs configured for 'git push':

master pushes to master (fast-forwardable)

v3.1 pushes to v3.1 (up to date)

`git remote rm origin` // 删除.git/config文件中添加remote origin相关的信息

`git remote add origin https://github.com/kekec/Test.git` // 在.git/config文件中添加remote origin指向的远程仓库URL (若已存在，则命令执行失败)

```
[remote "origin"]
  url = https://github.com/kekec/Test.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

`git remote set-url origin https://github.com/kekec/Test.git` // 修改.git/config文件中添加remote origin指向的远程仓库URL

`git remote prune origin` // 对于远程仓库不存在的分支，清除对应的远程分支cache

远程操作

`git clone https://github.com/kekec/Test.git` // 将https://github.com/kekec/Test.git上的当前分支克隆到本地（会创建一个名为Test目录，远程仓库名称使用默认名origin）

`git clone https://github.com/kekec/Test.git MyProject` // 将https://github.com/kekec/Test.git上的当前分支克隆到本地（会创建一个名为MyProject目录，远程仓库名称使用默认名origin）

`git clone -b v1.0 https://github.com/kekec/Test.git` // 将https://github.com/kekec/Test.git上的v1.0分支克隆到本地（会创建一个名为Test目录，远程仓库名称使用默认名origin）

`git clone -b v1.0 https://github.com/kekec/Test.git d:\MyGame` // 将https://github.com/kekec/Test.git上的v1.0分支克隆到d:\MyGame目录（会在d:\MyGame中创建一个名为Test目录，远程仓库名称使用默认名origin）

`git clone -o TestPrj https://github.com/kekec/Test.git` // 将https://github.com/kekec/Test.git上的当前分支克隆到本地（会创建一个名为Test目录，并将远程仓库名称设置为TestPrj）

`git fetch origin master` // 从远程仓库拉取master分支状态的变化信息（工作区文件不会更新）

`git fetch` // 从远程仓库拉取所有分支和tag状态的变化信息（工作区文件不会更新）

`git fetch -p` // 从远程仓库拉取所有分支和tag状态的变化信息，并清除已被删除的远程分支和tag在本地的缓存（工作区文件不会更新）

`git fetch origin --tags` // 从远程仓库拉取所有tag到本地（工作区文件不会更新）

`git pull <远程仓库名> <远程分支名>:<本地分支名>`

`git pull origin master` // 先执行fetch，然后将远程origin/master分支merge合并到当前分支（最后会更新origin/master, origin/HEAD指针到最新提交）

`git pull https://github.com/kekec/Test.git master` // 先执行fetch，将远程origin/master分支merge合并到当前分支（最后不会更新origin/master, origin/HEAD指针到最新提交）

`git pull origin v1.0:master` // 先执行fetch，然后将远程origin/v1.0分支merge合并到本地master分支

`git pull origin` // 先执行fetch，然后将对应的远程分支merge合并到当前分支（当前分支需要预存远程分支的追踪关系）

`git pull` // 先执行fetch，然后将对应的远程分支merge合并到当前分支（当前分支需要预存远程分支的追踪关系，而且当前分支只有一个远程仓库）

`git pull -p` // 先执行fetch，然后将对应的远程分支merge合并到当前分支，并清除已被删除的远程分支和tag在本地的缓存

`git pull -r origin master` // 先执行fetch，然后将远程origin/master分支rebase合并到master分支

`git push <远程仓库名> <本地分支名>:<远程分支名>`

`git push -u origin master` // 将本地仓库的修改push到origin所指向的远程仓库URL的master分支上，并在.git/config文件中记录当前分支与远程分支master的对应关系

`git push origin` // 将当前分支更新推送给对应的远端分支

`git push` // 将当前分支更新推送给对应的远端分支（当前分支只有一个远程仓库，可以省略仓库名origin）

git push origin -f // 使用当前分支更新强行覆盖对应的远端分支（合入远端分支有冲突时，也使用当前分支更新）

git push origin v1.0 // 将本地分支v1.0更新推送给对应的远端分支remotes/origin/v1.0

git push origin --all // 将本地所有分支更新推送给各自对应的远端分支

git push origin tag1.0 // 将本地标签tag1.0更新到远端标签tag1.0

git push origin --tags // 将本地所有标签更新到对应的远端标签

git push origin :v1.0 // 删除远端分支v1.0

git push origin :refs/tags/tag1.0 // 删除远程标签tag1.0

git push origin -d v1.0 // 删除远端分支v1.0 功能同上

储藏区

git stash // 将工作区中所有文件的修改备份压栈到储藏区，然后丢弃工作区与暂存区的所有文件的修改

git stash pop // 使用储藏区的栈顶处备份（stash@{0}）来恢复当前分支的工作区，并将栈顶备份移除

git stash apply stash@{1} // 使用储藏区的栈顶下面一个备份（stash@{1}）来恢复当前分支的工作区，但不移除储藏区中任何备份

git stash list // 查看储藏区栈列表

git stash show -p stash@{0} // 查看储藏区的栈顶处备份中各个文件的内容

git stash drop // 直接移除储藏区的栈顶处备份（不用于恢复当前分支的工作区）

git stash clear // 清除储藏区栈列表

工作区

git clean -nd // 探测工作区中有哪些未追踪状态的文件和目录

git clean -fd // 删除工作区中未追踪状态的文件和目录

暂存区

git ls-files // 查询暂存区中的文件列表（递归子目录）

git ls-files -s // 查看暂存区中所有文件的blob数据块信息

git ls-files -s -- README.md // 查看暂存区中的README.md文件的blob数据块信息

其他命令

`git fsck --full` // 列出所有未引用的blob、tree、commit对象

`git archive --format zip --output d:/file.zip master` // 将当前master分支所有文件使用zip压缩方式打包到d:/file.zip

Git瘦身

`git count-objects -v` // 查看git对象的统计信息

`find .git/objects -type f -print0 | xargs -o du -hk | sort -nr | head -5` // 查找git库中最大的5个文件（du -hk中的k代表单位为KB）

`find .git/objects -type f -size +1M -print0 | xargs -o du -hm | sort -nr | head -5` // 查找git库中size超过1M的最大的5个文件（du -hm中的k代表单位为MB）

`git verify-pack -v .git/objects/pack/pack-b340eea7566df839294b71ec91a327ca2ece0b94.idx | sort -k 3 -nr | head -5` // 对压缩存储的git库查找最大的5个文件

`git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch FramePro.cpp' --prune-empty --tag-name-filter cat -- --all` // 从git库的历史记录中彻底清理FramePro.cpp

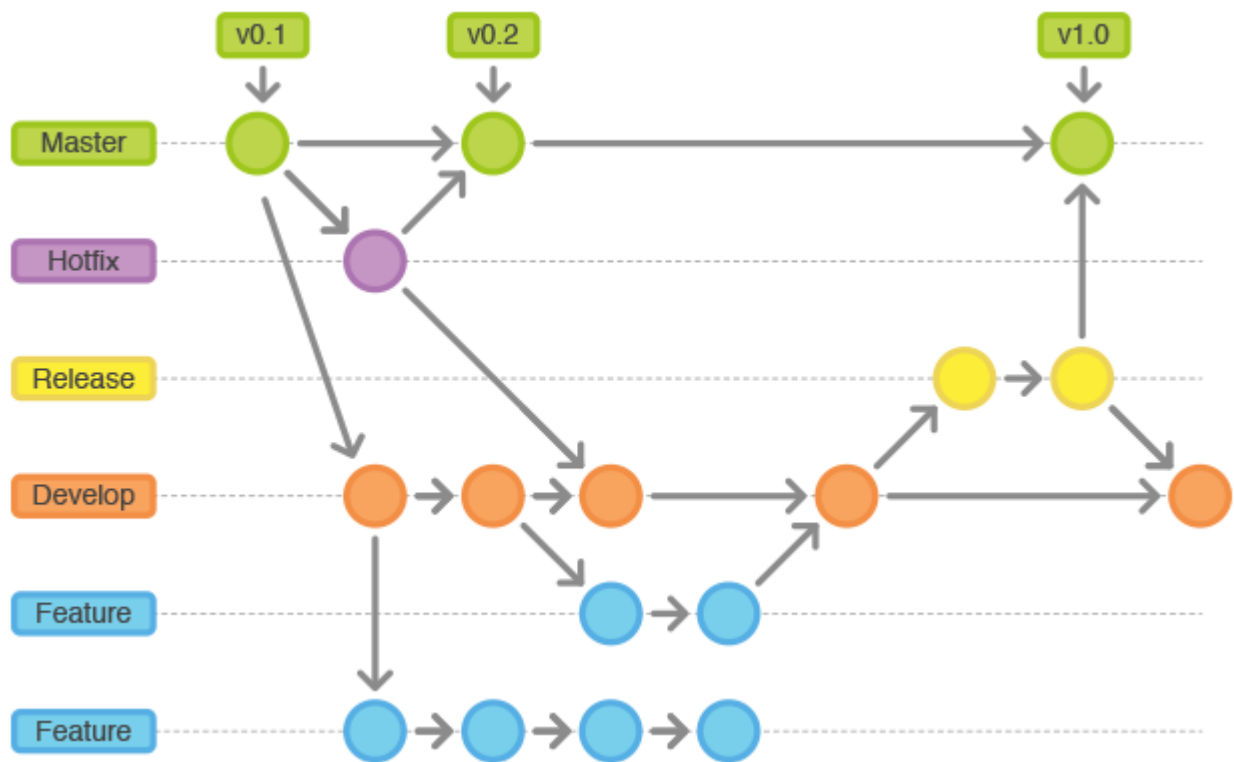
`git for-each-ref --format='delete %(refname)' refs/original | git update-ref --stdin` // 清理所有废弃的ref引用

`git gc --prune=now` // ①将所有的对象压缩存储到pack二进制文件中，以节省空间和提高效率 ②移除与任何提交都不相关的陈旧对象

`git reflog expire --expire=now --all` // 清除所有操作记录日志

除了使用git原生命令外，可以使用专门的工具BFG（java实现）来对Git库瘦身

经典Gitflow



(1) master分支存储了正式发布的历史（master分支上的所有提交都会分配一个版本号）

(2) develop分支作为功能的集成分支

(3) 每个新功能位于一个自己的Feature分支，该分支使用develop分支作为父分支。当新功能完成时，合并回develop分支。新功能提交应该从不直接与master分支交互

(4) 一旦develop分支上有了做一次发布（或者说快到了既定的发布日）的足够功能，就从develop分支上fork一个release分支。

新建的分支用于开始发布循环，所以从这个时间点开始之后新的功能不能再加到这个分支上。这个分支只应该做Bug修复、文档生成和其它面向发布任务。

对外发布的工作完成后，发布分支会合并到master分支并分配一个版本号打好Tag。另外，这些从新建发布分支以来的做的修改要合并回develop分支。

(5) hotfix分支用于生成快速给产品发布版本（production releases）打补丁，修复完成，修改应该马上合并回master分支（打好Tag）和develop分支（当前的发布分支）。

参考

[Pro Git第二版pdf](#)

[Git原理入门](#)

[常用 Git 命令清单](#)

[Git远程操作详解](#)

[Git 工作流程](#)

[Git 使用规范流程](#)

[Git分支管理策略](#)

[GIT基本概念和用法总结](#)

[Git工作流指南：Gitflow工作流](#)