**错误：**

error: insufficient permission for adding an object to repository database ./objects

解决：sudo git push

I find neither http.proxy nor GIT\_PROXY\_COMMAND work for my authenticated http proxy. The proxy is not triggered in either way. But I find a way to work around this.

Install corkscrew, or other alternatives you want.

Create a authfile. The format for authfile is: user\_name:password, and user\_name, password is your username and password to access your proxy. To create such a file, simply run command like this: echo "username:password" > ~/.ssh/authfile.

Edit ~/.ssh/config, and make sure its permission is 644: chmod 644 ~/.ssh/config

Take github.com as an example, add the following lines to ~/.ssh/config:

Host github.com

        HostName github.com

        ProxyCommand /usr/local/bin/corkscrew <your.proxy> <proxy port> %h %p <path/to/authfile>

        User git

**版本控制系统：集中式 vs 分布式：**

集中式必须联网才能工作。

分布式每台电脑都是一个完整的版本库，只是需要不定期和其他人的库交换一下数据。

**核心概念：**

Git的核心－－Git管理的是修改而不是文件

Git的核心－－一个Git的Repo是由许多的Commit所组成的有向图；Branch，Tag，HEAD仅仅是指向其中某个Commit的一个快捷指针。其中HEAD又有些特别，HEAD作为当前暂存区和工作区的基础版本；HEAD不仅可以指向一个实际的Commit，也可以通过指向Branch而间接的指向一个Commit。

Git的核心－－每一个Commit会用一个唯一的ID来表示，另外由于Branch，Tag，Head也都是指向一个具体的Commit的，因此可以在任何需要Commit作为参数的地方使用Branch，Tag和HEAD。

Git的核心－－Merge和Rebase的本质是将别人和代码（Commit）和自己的代码（Commit）进行合并，Merge和Rebase的不同在于产生的提交历史有所不同。

版本库之间的交互通过推送/push和拉回/pull操作来进行，pull和push操作以分支为中心。通过为每个分支设置上游（remote/branch）来设置该分支默认pull或者push的远程分支。

引用refs/heads/master就好像是一个游标，在有新的提交发生的时候指向了新的提交。可是如果只可上、不可下，就不能称为“游标”。Git提供了git reset命令，可以将“游标”指向任意一个存在的提交ID。

**次要概念：**

裸版本库－－没有工作区 －－bare

源版本库（上游版本库）－－origin

对等工作区

快进式推送与非快进式推送

强制非快进式推送与合并后推送

智能协议与哑协议

**工作区和暂存区：**

工作区：

.git所在的目录

暂存区（stage或index）：

在.git目录中，暂存区是一个介于工作区和版本库的中间状态，当执行提交时实际上是将暂存区的内容提交到版本库中。

版本库：

.git目录

HEAD－－项目当前分支

config－－项目配置信息

description－－项目描述信息

hooks/－－系统默认钩子脚本目录

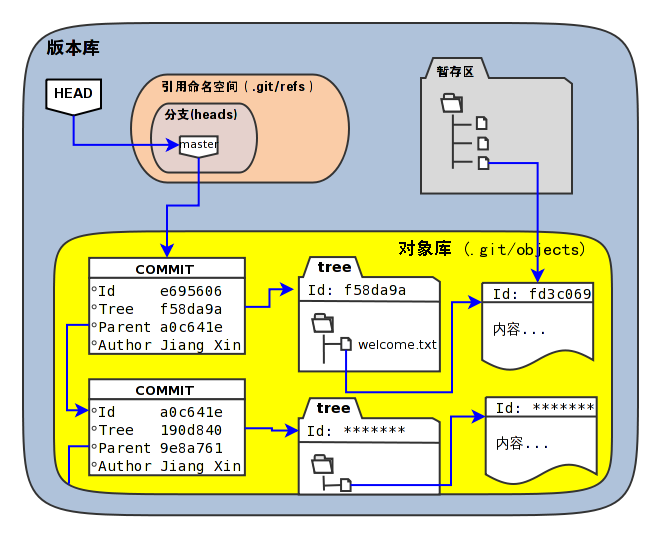
index－－索引文件

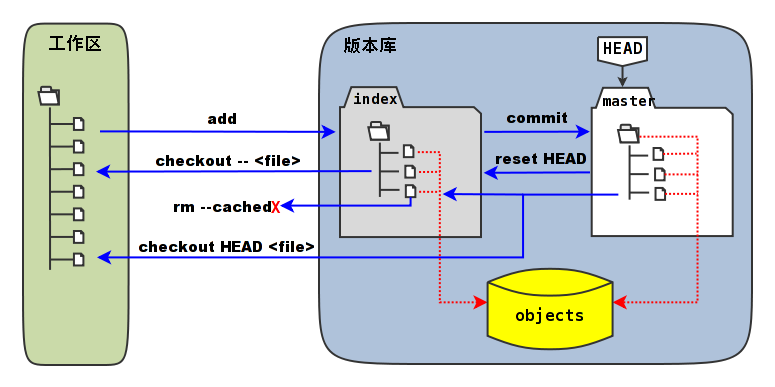
logs/－－各个refs的历史信息

objects/－－Git本地倉庫的所有對象

refs/－－标识你项目里的每个分支指向的Commit

说明：Git中的工作区、暂存区、和版本库最新版本（当前分支）分别是三个不同的目录树。





图中左侧为工作区，右侧为版本库。在版本库中标记为index的区域是暂存区（stage，亦称index），标记为master的是master分支所代表的目录树。

图中可以看出此时HEAD实际是指向master分支的一个“游标”。所以图示的命令中出现HEAD的地方可以用master来替换。

图中的objects标识的区域为Git的对象库，实际位于.git/objects目录下，会在后面的章节重点介绍。

当对工作区修改（或新增）的文件执行git add命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。

当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master分支会做相应的更新。即master最新指向的目录树就是提交时原暂存区的目录树。

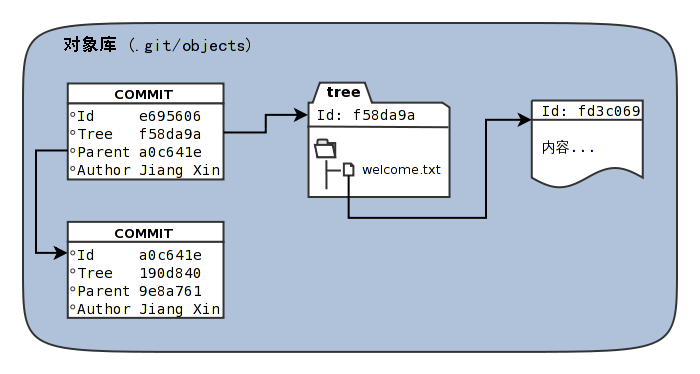
当执行git reset HEAD命令时，暂存区的目录树会被重写，被master分支指向的目录树所替换，但是工作区不受影响。

当执行git rm –cached <file>命令时，会直接从暂存区删除文件，工作区则不做出改变。

当执行git checkout .或者git checkout – <file>命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

当执行git checkout HEAD .或者git checkout HEAD <file>命令时，会用HEAD指向的master分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

Git对象模型：



对象：cat-file

名称：40个字符的字符串，以对象内容的SHA1哈希命名

类型：

blob：文件

tree：

trees-refs

blobs-refs

commit：

tree-ref

parent-ref

author

committer

message

tag：tag是一个"固化的分支'，一旦创建，其代表的内容永不可变。

object-ref

type

tagger

message

内容：取决于对象的类型

大小：内容的大小

检查：

#类型

git cat-file -t <obj>

#大小

git cat-file -s <obj>

#内容

git cat-file -p <obj>

**Git符号：man gitrevisions**

Commit表示法：

Commit ID => 使用部分SHA1哈希值（4位以上）

HEAD  => HEAD所对应的提交

HEAD:{1} => HEAD本身的上一个版本指向的提交

<branch> => <branch>所对应的提交

^ => 符号^指代父提交，如：HEAD^, HEAD^^

^[nun] => 对于一个提交有多个父提交，可以在符号^后面用数字表示第几个父提交

~[num] => 符号~也可以指代祖先提交

Tree表示法：

<commit>^{tree} => 指代提交所对应的树对象

<commit>: => 指代提交所对应的树对象

File表示法：

File in commit：

<commit>^{tree}:path/to/file

<commit>:path/to/file

File in stage：

:path/to/fie

Branch表示法：

<branch>

heads/<branch>

refs/heads/<branch>

<remote>/<branch>

remotes/<remote>/<branch>

refs/remotes/<remote>/<branch>

Tag表示法：

<tag>

tags/<tag>

refs/tags/<tag>

<remote>/<tag>

说明：

里程碑对象不一定是提交，有可能是一个Tag对象。里程碑A指向了一个Tag对象而非提交的时候，用下面的三个表示法都可以指向里程碑对应的提交。A^{} A^0 A^{commit}。

Range of Commit表示法：

一个提交ID实际上就可以代表一个版本列表。含义是：该版本开始的所有历史提交。

两个或多个版本，相当于每个版本单独使用时指代的列表的并集。

在一个版本前面加上符号（^）含义是取反，即排除这个版本及其历史版本。参数的顺序不重要。

使用两个点连接两个版本，如G..D，就相当于^G D。参数的顺序很重要。

三点表示法的含义是两个版本共同能够访问到的除外。参数的顺序不重要。

某提交的历史提交，自身除外，用语法r1^@表示。

提交本身不包括其历史提交，用语法r1^!表示。

<left-commit> <right-commit> 从left-commit到right-commit不包括left-commit但是包括right-commit。

**配置文件：**

Git配置文件为INI文件，git config命令用来读取或设置其中的属性的值。

设值：git config <option> <section>.<key> <value>

取值：git config <option> <setction>.<key>

删除：git config <option> --unset <section>.<key>

三个级别的配置：

版本库级别的配置：<repo>/.git/config

git config -e

全局配置文件：~/.gitconfig

git config -e --global

系统级配置文件：/etc/gitconfig

git config -e --system

常用Git配置命令：

git  config -l

配置命令别名：

git config --global alias.ci 'commit'

git config --global alias.st 'status'

git config --global alias.co 'checkout'

git config --global alias.gpr 'pull --rebase'

git config --global alias.smui 'submodule update --init --recursive'

git config --global alias.glog "log --graph"

长路径支持：

git config --global core.longpaths true

文件名大小写：

git config --global core.ignorecase false

配置用户信息：

git config --global user.name "<name>"

git config --global user.email "<email>"

 配置使用keychain保存HTTP密码：

设置：

git config --system credential.helper osxkeychain

取消：

git config --system --unset credential.helper

注意：

设置后使用http协议的Git仓库，会在第一次请求时，要求输入密码，密码会被保存在keychain中，再次请求将不用输入密码。

注意密码将按Git仓库的站点地址保存在keychain中，比如仓库为http://code.itc.cmbchina.cn/Bank\_to\_Bank\_Team/lr15-02\_emall\_api\_phillips.git，则密码被保存在code.itc.cmbchina.cn的keychain条目下，Where地址为http://code.itc.cmbchina.cn。

打开RefLog：

core.logallrefupdates—>true

关闭非快进式推送：

receive.denyNonFastForwards—>true

关闭分支删除：

receive.denyDeletes—>true

**版本库－repository：**

Create new：

git init [directory]

git init --bare [directory.git]

Clone:

git clone <repository> <directory>

git clone —bare <repository> <directory.git>

git clone —mirror <repository> <directory.git>

Add file:

git add <path>

git add --all/-A

git add -u

git add -p

Remove file:

git rm <path>

git rm -f <path>

git rm —cached <path>

Rename or Move file:

git mv <source> <dist>

Commit:

说明：

在Git提交时，如果在命令行不提供提交说明（没有使用-m参数），Git会自动打开一个编辑器，要求您在其中输入提交说明，输入完毕保存退出。

正常提交：

git commit -m “xxx”

修补提交：

git commit —amend -m “xxx”

Commit-Tree:

使用git commit-tree命令可以直接从目录树创建提交，命令git commit-tree的输出是一个提交的SHA1哈希值。查看这个提交会发现这个提交没有历史提交，可以称之为孤儿提交。

git commit-tree -m "<message>" <tree>

Revert:

git revert <commit>

Discard file in stage:

git reset HEAD <path>

git reset HEAD .

Discard file in working directory：

git checkout -- <path>

git checkout .

Discard untracked file in working directory:

git clean -f <path>

git clean -fd <path>

Stash：

git stash

git stash list

git stash save -m “xxx"

git stash pop

Reset: Will change the cmmit history.

用法一：不重置引用，仅用指定<commit>（默认为 HEAD）下的文件<paths>替换掉暂存区中的文件。

git reset [-q] [<commit>] [—] <paths>

用法二：会用指定的<commit>（默认为 HEAD）重置引用（HEAD指向的游标如refs/heads/master），并根据不同的选项（缺省即为--mixed）可以对暂存区或工作区进行重置。

git reset [--soft | mixed | hard | marge | keep] [-q] [commit]

—hard:

1.替换引用的指向

2.替换暂存区

3.替换工作区

—soft:

1.替换引用的指向

—mixed:

1.替换引用的指向

2.替换暂存区

git reset | get reset HEAD

仅用HEAD指向的目录树重置暂存区。

git reset —filename | get reset HEAD —filename

仅将文件filename的改动撤出暂存区。

git reset HEAD^ | git reset –mixed HEAD^

工作区不变，暂存区回退到上一个提交之前，引用也回退一次。

git reset –soft HEAD^

工作区和暂存区不改变，但是引用向前回退一次。

git reset –hard HEAD^

彻底撤销最近的提交。引用回退到前一次，而且工作区和暂存区都会回退到上一次提交的状态。自上一次以来的提交全部丢失。

Checkout：HEAD的重置即Checkout

HEAD可以理解为“头指针”，是当前工作区的“基础版本”，当提交时HEAD指向的提交将作为新提交的父提交。

“分离头指针”状态指的就是HEAD头指针指向了一个具体的提交ID，而不是一个引用（分支）。

用法一： git checkout [-q] [<commit>] [--] <paths>...

用法二： git checkout [<branch>]

用法三： git checkout [-m] [[-b|--orphan] <new\_branch>] [<start\_point>]

第一种用法（包含了路径<paths>的用法）不会改变HEAD头指针，主要是用于指定版本的文件覆盖工作区中对应的文件。如果省略<commit>，会拿暂存区的文件覆盖工作区的文件，否则用指定提交中的文件覆盖暂存区和工作区中对应的文件。

第二种用法（不使用路径<paths>的用法）则会改变HEAD头指针。之所以后面的参数写作<branch>，是因为只有HEAD切换到一个分支才可以对提交进行跟踪，否则仍然会进入“分离头指针”的状态。在“分离头指针”状态下的提交不能被引用关联到而可能会丢失。所以用法二最主要的作用就是切换到分支。如果省略<branch>则相当于对工作区进行状态检查。

第三种用法主要是创建和切换到新的分支（<new\_branch>），新的分支从<start\_point>指定的提交开始创建。新分支和我们熟悉的master分支没有什么实质的不同，都是在refs/heads命名空间下的引用。

Diff:

Commit与Commit

git diff <left-commit> <right-commit>

工作区和暂存区：

git diff

git diff <file>

工作区和HEAD或Commit：

git diff HEAD｜<right-commit>

git diff HEAD｜<right-commit> <file>

暂存区和HEAD或Commit

git diff --cached [<right-commit>]

git diff --cached [<right-commit>] <file>

Blame:

git blame <file>

Patch：

single:

git format-patch <commit>^!

range:

git format-patch [-s] <commit-range>

apply:

cat \*.patch|git am

git apply <patchfile>

RefLog: Reference Log(.git/logs/\*\*): 找提交

#Log of Head:

git reflog

#Log of Branch:

git reflog <branch>

Fsck：

检查悬空对象即Dangling Objects（Dangling Objects分为Dangling Commit和Dangling Blob；Dangling Commit由Dangling Blob组成；.git/lost-found）

git fsck

git fsck --no-reglogs

Prune && GC:

对Git库执行清理操作。

Log:

git log [<options>] [<revision range>] [[--] <path>…]

options:

-<num>

—graph

—stat

—decorate

--full-diff

--pretty=raw

path:

<remote>/<branch>

Show：Show various types of objects

Cat-File：check repository objects

git cat-file commit <branch>

git cat-file tree <branch>^{tree}

git cat-file blob <branch>:<file>

Rev-Parse:

git rev-parse <branch>

git rev-parse <branch>:<file>

git rev-parse <branch>^{tree}

Cherry-pick:

git cherry-pick <commit>

从众多的提交中挑选出一个提交应用在当前的工作分支中，操作过程相当于将选择的提交导出为补丁文件，然后在当前HEAD上重放。

Ls-Files:

git ls-files -s

git ls-files --with-tree=<branch/commit>

Show refs:

git show-ref

Pack refs:

git pack-refs —all

Archive:

git archive -o

Status:

git status

git status -s

git reflog

**子模块：**

**数据：**

**.gitmodules：保存submodule、repo\_url和folder**

**folder：虚拟文件， 用来保存Commit ID**

**场景：**

场景一：Status:

git submodule status

场景二：Add a submodule：

git submodule add <repo\_url> <path-of-submodule>

git commit -m "add submodule"

场景三：Clone repo which have submodules:

方式一：

git clone <repo\_url>

#只是修改.git/config文件

git submodule init

#真正克隆子模块并Checkout到相应的Commit

git submodule update --recursive

方式二：

git clone <repo\_url>

git submodule update --init --recursive

方式三：

git clone <repo\_url> --recursive

场景四：修改子模块，注意子模块的分支，必要时checkout branch。

Step 1: 修改子模块/提交子模块/Push子模块

Step 2: 提交主模块/Push主模块

场景四：更新子模块，注意子模块的分支，必要时checkout branch 或pull时指定branch。

方式一：

cd <submodule>/git pull 或 cd <submodule>/git pull origin master

方式二：

git submodule foreach git pull  或 git submodule foreach --recursive git pull origin master

**分支管理：**

查看分支：

git branch/git branch -v | -a | -r

创建分支：

git branch <branchname>

git branch <branchname> <start-point>

删除分支：

git branch -d <branchname>

git branch -D <branchname>

重命名分支：

git branch -m <oldbranch> <newbranch>

git branch -M <oldbranch> <newbranch>

切换分支：

git checkout <branchname>

创建+切换分支：

git checkout －b <branchname>

git checkout －b <branchname> <start-point>

合并某分支到当前分支：

git merge < branchname >

设置跟踪：

#Create new local-branch for remote-branch

git branch —track <local-branch> <remote-branch>

#Set upstream for existed local-branch or create new local-branch

git branch —set-upstream <local-branch> <remote-branch>

#Remove track for local-branch

git branch —no-track <local-branch>

#Set upstream for existed local-branch

git branch (--set-upstream-to=<upstream> | -u <upstream>) [<local-branch>]

#Unset upstream for existed local-branch

git branch --unset-upstream [<local-branch>]

分支模式：

为什么需要分支：

成功的软件项目大多要经过多个开发周期，发布多个软件版本。每个已经发布的版本都可能发现bug，这就需要对历史版本进行更改。

有前瞻性的项目管理，新版本的开发往往是和当前版本同步进行的。如果两个版本的开发都混杂在master分支中，肯定会是一场灾难。

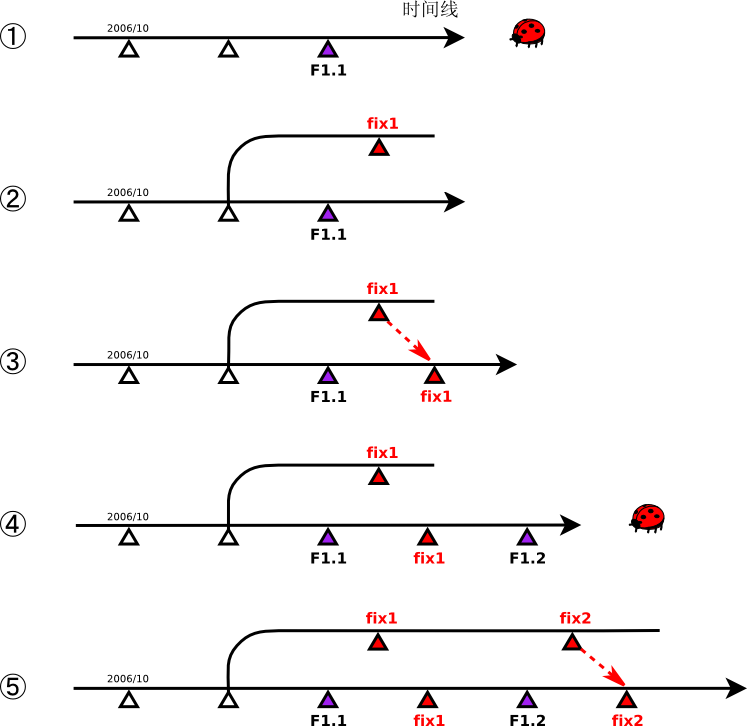
如果产品要针对不同的客户定制，肯定是希望客户越多越好。如果所有的客户定制都混杂在一个分支中，必定会带来混乱。如果使用多个分支管理不同的定制，但如果管理不善，分支之间定制功能的迁移就会成为头痛的问题。

即便是所有成员都在为同一个项目的同一个版本进行工作，每个人领受任务却不尽相同，有的任务开发周期会很长，有的任务需要对软件架构进行较大的修改，如果所有人都工作在同一分支中，就会因为过多过频的冲突导致效率低下。

敏捷开发（不管是极限编程XP还是Scrum或其他）是最有效的项目管理模式，其最有效的一个实践就是快速迭代、每晚编译。如果不能将项目的各个功能模块的开发通过分支进行隔离，在软件集成上就会遭遇困难。

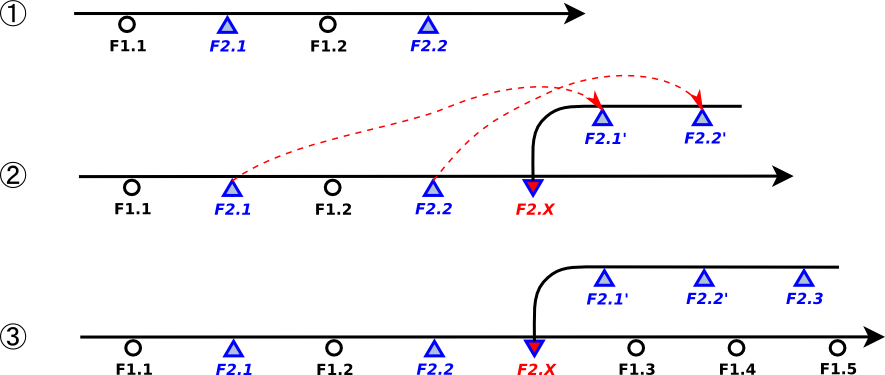
发布分支－Release Branch（或者Bugfix分支）- 发布后的Bugfix与主线开发互不影响

使用版本控制系统的分支功能，可以避免对已发布的软件版本进行bug修正时引入新功能的代码，或者因误删其他bug修正代码导致已修复问题重现。在这种情况下创建的分支有一个专有的名称：bugfix分支或发布分支（Release Branch），因为在软件新版本发布后经常使用此技术进行软件维护，发布升级版本。



特性分支－Feature Branch （或者主题分支Topic Branch）- 多Feature开发互不影响

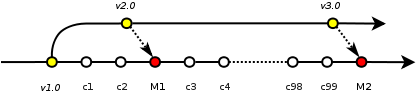
采用分支将某个功能或模块的开发与开发主线独立出来，这种用途的分支被称为特性分支（Feature Branch）或主题分支（Topic Branch）。那么在什么情况下使用特性分支呢？试验性、探索性的功能开发应该为其建立特性分支。功能复杂、开发周期长（有可能在本次发布中取消）的模块应该为其建立特性分支。会更改软件体系架构，破坏软件集成，或者容易导致冲突、影响他人开发进度的模块，应该为其建立特性分支。



卖主分支－Vendor Branch - 上游代码功能开发和本地定制功能开发互不影响

有的项目要引用到第三方的代码模块并且需要对其进行定制，有的项目甚至整个就是基于某个开源项目进行的定制。如何有效地管理本地定制和第三方（上游）代码的变更就成为了一个难题。卖主分支（Vendor Branch）可以部分解决这个难题。如果定制较少，使用卖主分支可以工作得很好，但是如果定制的内容非常多，在合并的时候就会遇到非常多的冲突。定制的代码越多，混杂的越厉害，冲突解决就越困难。

所谓卖主分支，就是在版本库中创建一个专门和上游代码进行同步的分支，一旦有上游代码发布就检入到卖主分支中。



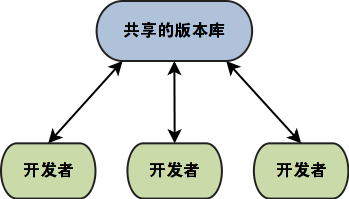
**协作模型：**

**说明：**

**团队协作和个人之间有何不同？关键就在于团队成员之间存在着数据交换，与远程版本库进行数据交换，是Git协同的主要手段：数据交换需要协议；数据交换可能会因为冲突造成中断，如何解决冲突；里程碑为数据建立标识，是数据交换的参照点；分支会为数据交换开辟不同的通道，从而减少冲突和混乱的发生。**

**经典Git协同模型：**

集中式协作模型：



传统集中式协作模型（主要应用于公司范围内和商业软件开发中），可以像集中式版本控制系统那样使用Git，在一个大家都可以访问到的服务器上架设Git服务器，每个人从该服务器克隆代码，然后将本地提交推送到服务器上。

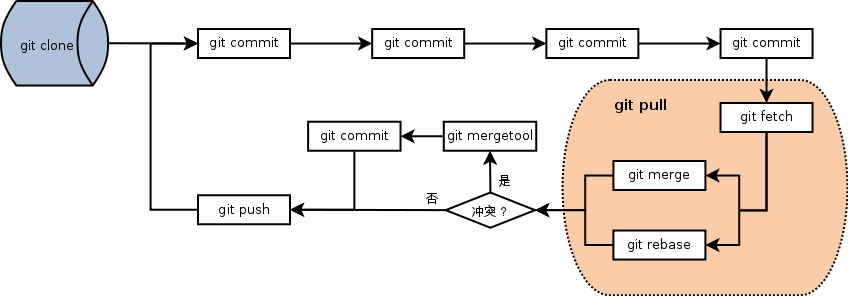
使用集中式协作模型时，对版本库授权和管理上的要求如下：1）能够设置谁能够访问版本库，谁不能访问版本库；2）只允许用户访问版本库中的某些路径，其他路径不能访问，可以限制哪些分支不允许写，哪些路径不允许写；3）特定目录只允许特定用户执行写操作；4）服务器可以通过钩子实现特殊功能，如对提交说明（commit log）的检查，数据镜像等；5）可以设置谁可以创建新的分支；6）可以设置谁可以创建新的版本库；7）可以设置谁可以强制更新

Git在授权读访问上有如下的限制：1）Git不支持对版本库读取的精确授权，只能是非零即壹的授权。即或者能够读取一个版本库的全部，或者什么也读不到。2）因为Git的提交是一个整体，提交中包含了完整目录树（tree）的哈希，因此完整性不容破坏。3）Git是分布式版本控制系统，如果允许不完整的克隆，那么本地就是截然不同的版本库，在向服务器推送的时候，会被拒绝或者产生新的分支。

对于集中式的工作模型的核心就是架设集中式的Git服务器，而且尽量能够满足前面提到的对授权和版本库管理上的需求。使用Gitolite架设Git服务器，可以实现集中式协同模型对版本库授权和管理上的要求。

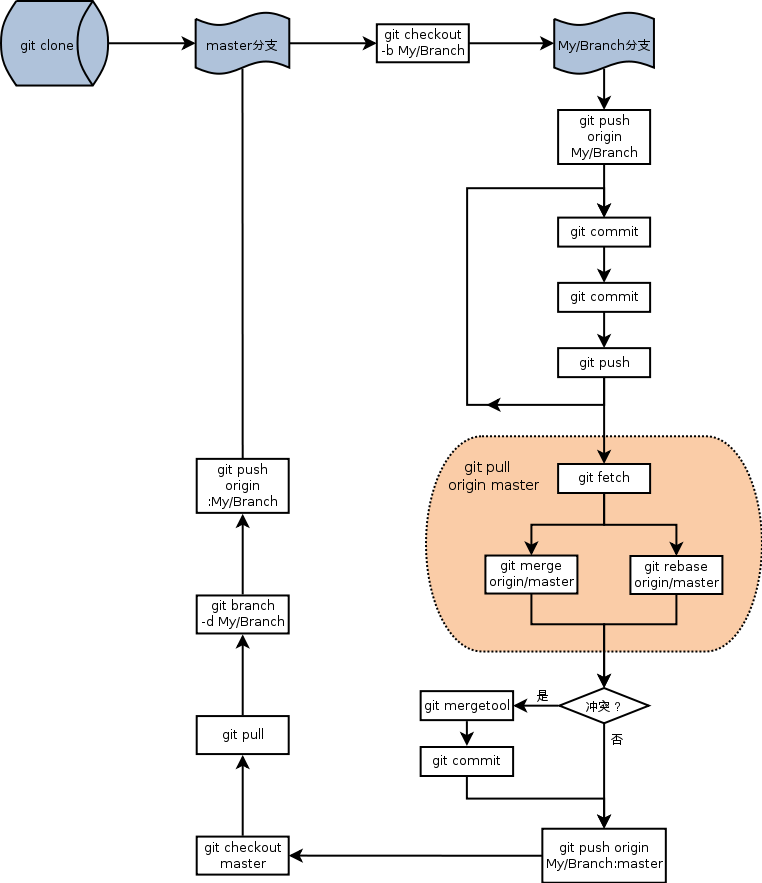
集中式协同模型的工作流：

工作流一：基于Remote分支的协同工作流



工作流二：基于独享特性分支的协同工作流

在本地创建本地分支（功能分支），并且同时在服务器端（共享版本库）也创建自己独享的功能分支。本地提交推送到共享版本库的自己独享的分支上。当开发完成之后，将功能分支合并到主线上，推送到共享版本库，完成开发。当然如果该特性分支不再需要时需要作些清理工作。



社交网络式协作模型：

贡献者开放只读版本库以贡献代码（get request-pull）

贡献者以补丁方式贡献代码（git format-patch git send-email git am）

Topgit(Topic Git)协作模式：

**Tag：**

**两种类型：**

轻量级Tag -- Git底层不会创建一个真正意义的Tag对象，使用git cat-file -t tagname会返回commit。

带注释的Tag -- 使用-a，-s或-u 创建，Git底层会创建一个真正意义的Tag对象

**Tag管理：**

List Tags：

git tag

Add Tag：

Delete Tag：

git tag -d <tagname>…

共享与同步：

Tag共享，必须显式Push。

Tag同步，使用Fetch或者Pull（只能同步远程分支上新的Tag）。

远程Tag删除，git push <remote\_url> :<tagname>

**变基操作：**

基本变基操作：遇到冲突，变基操作中断。

git rebase --onto <newbase> <since> <till>

git rebase --onto <newbase> <since> | git rebase --onto <newbase> <since> HEAD

git rebase <newbase> <till> | git rebase <newbase>  <newbase as since> <till>

git rebase <newbase> | git rebase <newbase>  <newbase as since> HEAD

变基操作中断时后续命令：

变基冲突致暂停，先解决冲突（添加到暂存区，不提交），然后继续变基操作。

git rebase —continue

变基冲突致暂停，跳过当前提交：

git rebase —skip

变基冲突致暂停，终止变基操作，切换回变基前分支：

git rebase —abort

交互式变基操作：

执行交互式变基操作，会将<since>..<till>的提交悉数罗列在一个文件中，然后自动打开一个编辑器来编辑这个文件。可以通过修改文件的内容（删除提交，修改提交的动作关键字）实现删除提交，压缩多个提交为一个提交，更改提交的顺序，更改历史提交的提交说明。

git rebase -i --onto <newbase> <since> <till>

git rebase -i --onto <newbase> <since> | git rebase -i --onto <newbase> <since> HEAD

git rebase -i <newbase> <till> | git rebase -i  <newbase>  <newbase as since> <till>

git rebase -i <newbase> | git rebase -i <newbase>  <newbase as since> HEAD

**合并操作：**

合并操作可以由拉回操作（git pull）隐式的执行，将其他版本库的提交和本地版本库的提交进行合并。还可以针对本版本库中的其他分支进行显示的合并操作，将其他分支的提交和当前分支的提交进行合并。

用法：

git merge [选项...] <commit>...

合并操作的大多数情况，只须提供一个<commit>（提交ID或对应的引用：分支、里程碑等）作为参数。合并操作将<commit>对应的目录树和当前工作分支的目录树的内容进行合并，合并后的提交以当前分支的提交作为第一个父提交，以<commit>为第二个父提交。合并操作还支持将多个<commit>代表的分支和当前分支进行合并。

记录合并的特殊文件：

文件.git/MERGE\_HEAD记录所合并的提交ID。

文件.git/MERGE\_MSG记录合并失败的信息。

文件.git/MERGE\_MODE标识合并状态。

合并可能出现的结果：

成功的自动合并

合并产生逻辑冲突

真正的冲突

树冲突

合并冲突后的后续操作：

冲突解决的实质就是通过编辑操作，将冲突标识符所标识的冲突内容替换为合适的内容，并去掉冲突标识符。编辑完毕后执行git add命令将文件添加到暂存区（标号0），然后再提交就完成了冲突解决。

当工作区处于合并冲突状态时，无法再执行提交操作。此时有两个选择：放弃合并操作，或者对合并冲突进行冲突解决操作。放弃合并操作非常简单，只须执行git reset将暂存区重置即可。有两个方法进行冲突解决，一个是对少量冲突非常适合的手工编辑操作，另外一个是使用图形化冲突解决工具。

放弃合并操作

git merge --abort

解决合并冲突

冲突文件中的冲突标识符：

<<<<<<<=======>>>>>>>

Merge 合并所有提交：

git merge --squash <branch>

**远程仓库：**

远程仓库协议支持：

ssh：智能协议

ssh://[user@]example.com[:port]/path/to/repo.git/

[user@]example.com:path/to/repo.git/   #SCP格式表示法

git：智能协议

git://example.com[:port]/path/to/repo.git/

http[s]：兼有智能协议和哑协议

http[s]://example.com[:port]/path/to/repo.git/

可以通过在文件~/.netrc中写入明文口令实现使用 HTTPS 协议时也能自动完成认证。

ftp[s]：哑协议

ftp[s]://example.com[:port]/path/to/repo.git/

rsync：哑协议

rsync://example.com/path/to/repo.git/

local：智能协议

file:///path/to/repo.git/

/path/to/repo.git

查看远程仓库中的Refs:

git ls-remote <remote-repo> [options]

options:

-tags

-heads

从远程仓库克隆：

git clone <url>

查看远程仓库：

git remote/git remote -v

添加远程仓库：

git remote add <name> <url>

更换远程仓库地址：

git remote set-url <name> <new\_url>

查看远程分支：

git branch －r

Branch track in config:

[branch “<local\_branch>”]

remote = <remote>

merge = refs/heads/<remote\_branch>

签出远程分支(Setup track):

git checkout <branch>

git checkout -b <local\_branch> <remote>/<remote\_branch>

Fetch:

git fetch <remote-repo> <branch>

git fetch --all

Push:

Git通过对推送操作是否是“快进式”操作进行检查，从而保证用户的提交不会相互覆盖。一般情况下，推送只允许“快进式”推送。所谓快进式推送，就是要推送的本地版本库的提交是建立在远程版本库相应分支的现有提交基础上的，即远程版本库相应分支的最新提交是本地版本库最新提交的祖先提交。在推送命令的后面使用-f参数可以进行强制推送，即使是非快进式的推送也会成功执行。

理性的工作协同要避免非快进式推送。一旦向服务器推送后，如果发现错误，不要使用会更改历史的操作（变基、修补提交），而是采用不会改变历史提交的反转提交等操作。

如果在向服务器推送过程中遇到了非快进式推送的警告，应该进行如下操作才更为理性：执行git pull获取服务器端最新的提交并和本地提交进行合并，合并成功后再向服务器提交。

Git提供了至少两种方式对非快进式推送进行限制。一个是通过版本库的配置，另一个是通过版本库的钩子脚本。

将版本库的参数receive.denyNonFastForwards设置为true可以禁止任何用户进行非快进式推送。

git --git-dir=/path/to/repo/shared.git config receive.denyNonFastForwards true

git push [<remote-repo> [<refspec>]]

#Push current branch base on the Track（up-stream）

git push

#Push a branch from local to remote with same name

git push <remote-repo> <branch>

#Push a local\_branch to remote\_branch of remote\_repo

git push <remote-repo> <local\_branch>:<remote\_branch>

#Remove:

git push  --delete <remote-repo> <remote\_branch>

Pull:

git push [<remote-repo> [<refspec>]]

#Pull current branch base on the Track

git pull [—rebase]

#Pull current branch base on the Remote

git pull [—rebase] <remote-repo> <branch>

**建立Git Server：**

Create Server Repo:

#create new

mkdir REPO.git / git init —bare

#create from existed repo

git clone --bare REPO\_PATH REPO.git

使用ssh协议：ssh://

#add public key

~/.ssh/authorized\_keys

#how to check out

git clone ssh://USER@host\_address/PATH/REPO.git

使用git协议：git://

# how to host

touch REPO.git/git-daemon-export-ok

mv REPO.git PUBLIC\_PATH/

git daemon --base-path=PUBLIC\_PATH

git daemon --reuseaddr --base-path=. --export-all --verbose --enable=receive-pack

# how to checkout

git clone git://host\_address/REPO.git

**GIt with SSH：**

Git - Swich ssh-key：

方法1-全局：

1. 修改～/.ssh/config（默认配置文件）

IdentityFile ~/.ssh/id\_new\_rsa

IdentityFile ~/.ssh/id\_rsa

2. git 操作

方法2-Github：

1. 修改～/.ssh/config（默认配置文件）

Host github.com

IdentityFile ~/.ssh/id\_new\_rsa

2. git 操作

Git - Use ssh-proxy：

With ssh-config -- for any:

1. 修改 修改～/.ssh/config（默认配置文件）

UseRoaming no

# prod-kwnc dotcms

Host 192.168.32.48

IdentitiesOnly yes

User wguser

IdentityFile ~/.ssh/wg/kwnc-internal-wguser.key

ProxyCommand ssh -W %h:%p -l wguser -o IdentitiesOnly=yes -i ~/.ssh/wg/kwnc-wguser.key 202.86.189.242

# Other Hosts......

2. Clone repo from ssh:

git clone <ssh://wguser@192.168.32.48/opt/apps/kwnc-dotcms-backup.git>q/

Without ssh-config -- only for kwnc:

Create file ssh-proxy:

ssh -o "ProxyCommand ssh -W %h:%p -o IdentitiesOnly=yes -i ~/.ssh/wg/kwnc-wguser.key wguser@202.86.189.242" "$@"

chmod u+x /PATH/TO/ssh-proxy

Clone repo from ssh:

export GIT\_SSH="/PATH/TO/ssh-proxy"

git clone <ssh://wguser@192.168.32.48/opt/apps/kwnc-dotcms-backup.git>

思考：

为什么工作区下有一个.git目录？

Git Config命令参数的区别？

是谁完成的提交？

随意设置提交者姓名，是否太不安全？

命令别名是干什么的？

问题：

SHA1哈希值到底是什么，如何生成的？

为什么不用顺序的数字来表示提交？

git hooks分为客户端hook和服务端hook。客户端hook又分为pre-commit、prepare-commit-msg、commit-msg、post-commit等，主要用于控制客户端git的提交工作流。用户可以在项目根目录的.git目录下面配置使用，也可以配置全局git template用于个人pc上的所有git项目使用。服务端hook又分为pre-receive、post-receive、update，主要在服务端接受提交对象时进行调用。