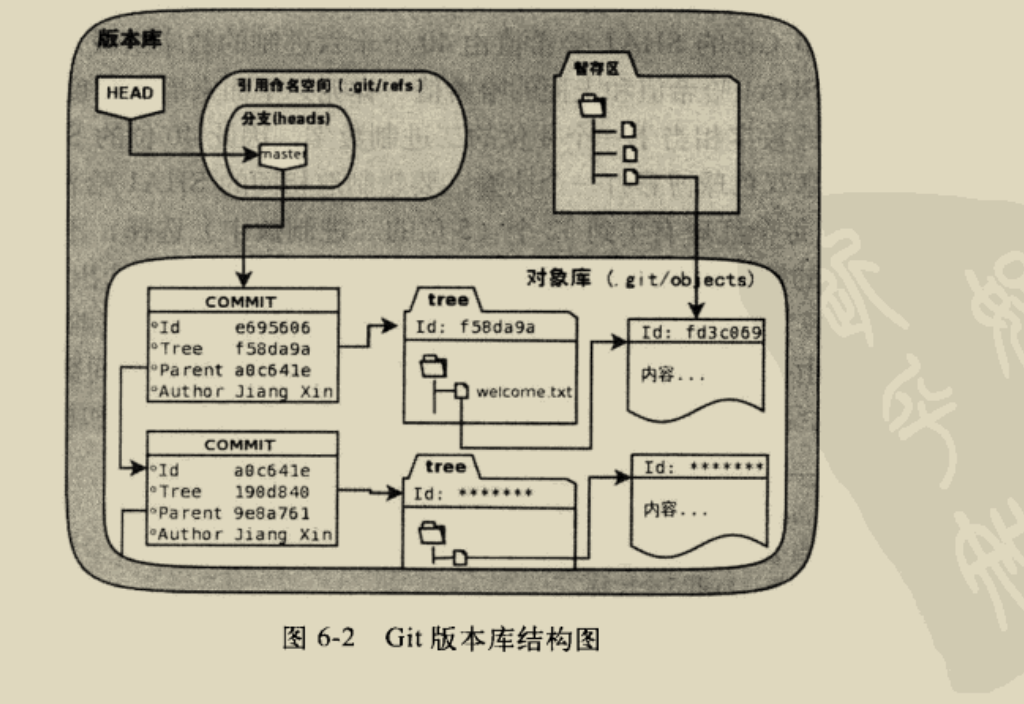
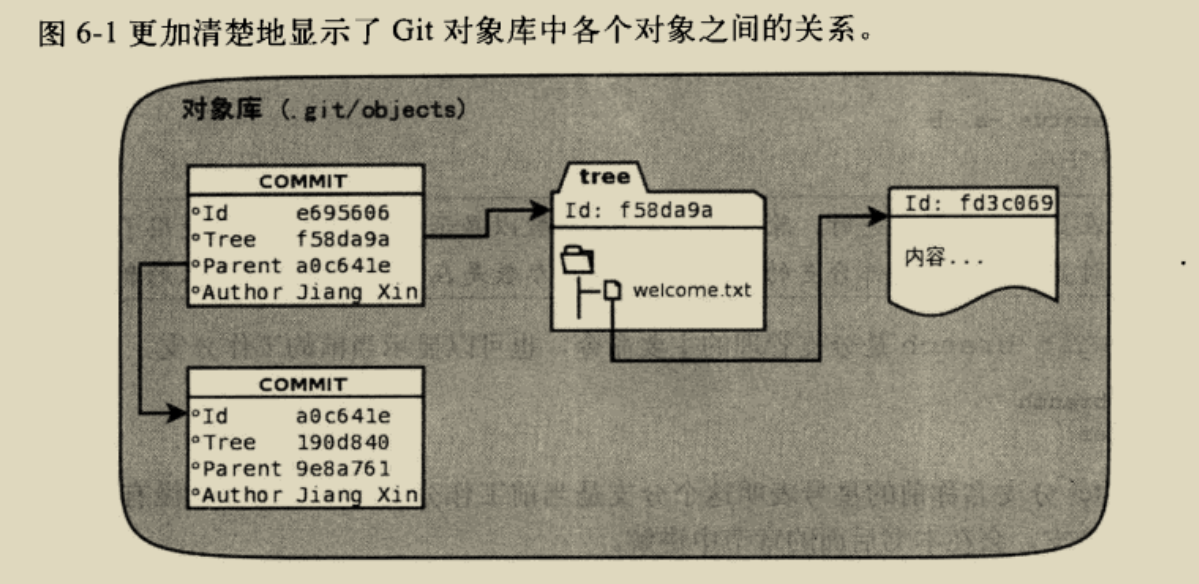
# Git对象

使用时间戳、文件长度等信息进行比较要比通过文件比较要快的多，故Git让工作区状态扫描更快速地执行。

## 一、git对象库



目录.git/refs是保存引用的命名空间，其中.git/refs/heads目录下引用又称为分支。Master指向的是一个提交ID（最新提交）。git rev-parse master 显示应用对应的提交ID。

Git库中的Objects目录下存放了git对象，这些对象ID的前两位作为目录名，后38位作为文件名。

Git库中的logs目录下存放的是日志文件，可以很好的跟踪相关操作的日志。

## 二、访问Git库中的对象

1.HEAD代表版本中最近的一次提交。

2.符号**^**可以用于指代父提交。

2.1 HEAD**^**代表版本库的上一个提交，即最近一次提交的父提交。

2.2 HEAD**^**代表HEAD**^**的父提交。

3.**~<**n>也可以用于指代祖先提交。如:HEAD**^^**即相当于HEAD**~**2

## 三、相关命令

**git status -s**  以精简的方式显示文件状态。

**git status -s –b** 以精简的方式显示文件状态，并显示当前的分支。

**git cat-file -t** <SHAI哈希值> 查看ID的类型，通常有**commit，tree，parent，blob，tag。**

**git cat-file -p** <SHAI哈希值> 查看内容。

**git rev-parse master** 显示应用对应的提交ID。

**printf Git | sha1sum** 生成Git的SHA1哈希值。

## 四、生成提交的哈希值

1、查看HEAD对应的提交的内容

**git cat-file commit HEAD**

2、获取提交中总共包含的字符串

**git cat-file commit HEAD | wc –c**

-->234

3、在提交信息的前面加上内容**commit 234<null>**(<null>为空字符串)，然后执行哈希算法

**（printf “commit 234\000”;git cat-file commit HEAD）| sha1sum**

4、利用**git rev-parse**进行对比

**git rev-parse HEAD**

# Git重置

## 一、重置的等级

重置的三个等级（**--soft、--mixed、--hard**）:

1、使用**--soft**就仅仅将头指针恢复，已经**add**的缓存以及工作空间的所有东西都不变。

2、如果使用**--mixed**，就将头恢复掉，已经**add**的缓存也会丢失掉，工作空间的代码什么的是不变的。

3、如果使用**--hard**，那么一切就全都恢复了，头变，**aad**的缓存消失，代码什么的也恢复到以前状态。

## 二、版本重置原理

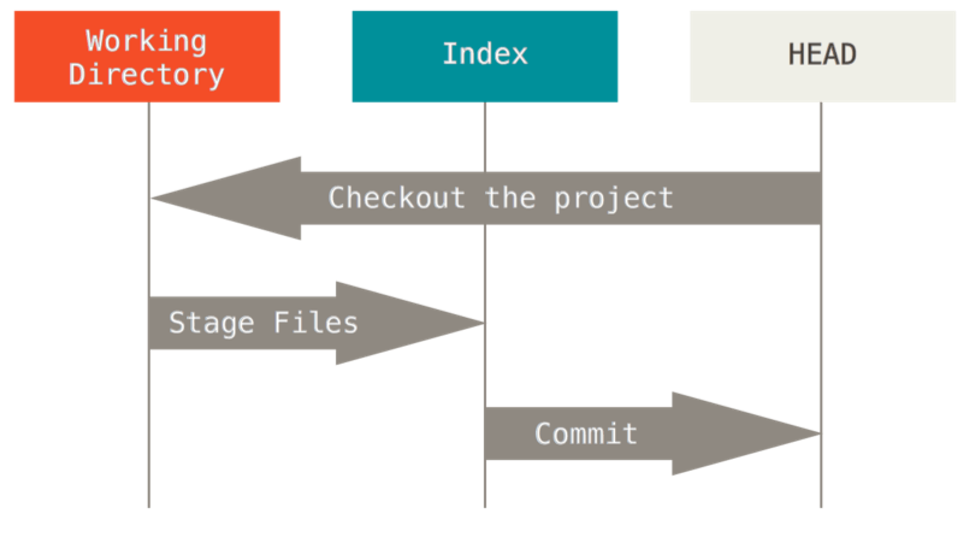
1、重要概念

三棵树（此处的树指文件的集合，不是特定的数据结构）。

|  |  |  |
| --- | --- | --- |
| 树 | 说明 | 用途 |
| HEAD | HEAD 是当前分支引用的指针，总指向该分支上最新的提交 | 上一次提交的快照，下一次提交的父节点 |
| Index | Git的暂存区域 | 预期的下一次提交的快照 |
| Working Directory | 实际的工作目录和文件 | 沙盒 |

2、工作流程

Git 主要的目的是通过操纵这三棵树来以更加连续的状态记录项目的快照。



reset 命令会以特定的顺序重写这三棵树，在你指定以下选项时停止：

(1)、移动HEAD分支指针的指向（若指定了--soft,则到此为止，Index和Working Directory树没有任何变化）。

(2)、使索引看起来像HEAD（若未指定--hard,则到此停止，Index和HEAD都发生了变化，而且看起来两者相同，Working Directory树没有任何变化）。

(3)、使工作目录像索引（指定了--hard,HEAD，Index和Working Directory都发生了变化，工作目录被全部重置，如没有提交版本，则无法恢复）。

3、通过路径来重置

若指定了一个路径，reset将会跳过第1步，并且将它的作用范围限定为指定的文件或文件集合。

(1)、移动HEAD分支的指向（已跳过）。

(2)、让索引看起来像 HEAD（到此处停止)。

**git reset file.txt (git reset --mixed HEAD file.txt)**

有两个效果：

(1)、将file.txt从HAED复制到索引中。

(2)、取消暂存文件。

4、**reset**和**checkout**的区别

4.1、不带路径：

**git checkout [branch]**与**git reset [--hard|--soft|--mixed][branch]**

相同点：

都会更新三棵树

不同点：

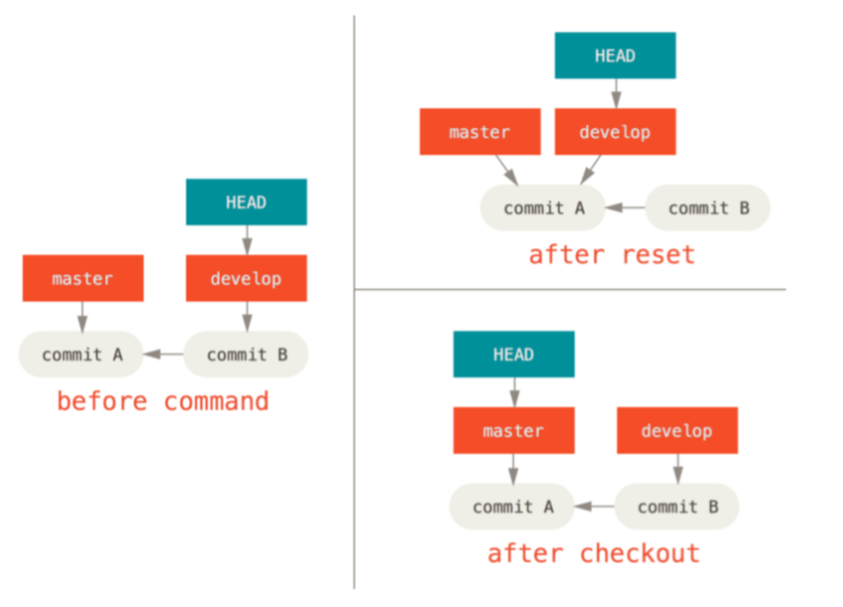
**checkout**对工作目录是安全的，它会通过检查来确保不会将已更改的文件弄丢。

**reset --hard** 则会不做检查就全面地替换所有东西。

**checkout**只会移动**HEAD**自身来指向另一个分支。

**reset** 不仅会移动**HEAD**分支指针的指向，而且会移动**HEAD**自身指针的指向。

重置一般用于重置暂存区，而检出命令主要用于覆盖工作区。



4.2、带路径

**git checkout –- file** 重置文件，不移动**HEAD**，工作区会重置。

**git reset [branch] <path>** 不会重置引用，更不会改变工作区,而是用指定的提交状态下的文件**<path>**替换暂存区的文件。

总结：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **HEAD** | **Index** | **Workdir** | **Safe** |
| **Commit Level** | | | | |
| **reset --soft [commit]** | **REF** | **NO** | **NO** | **YES** |
| **reset [commit]** | **REF** | **YES** | **NO** | **YES** |
| **reset --hard [commit]** | **REF** | **YES** | **YES** | **NO** |
| **checkout [commit]** | **HEAD** | **YES** | **YES** | **YES** |
| **File Level** | | | | |
| **reset (commit) [file]** | **NO** | **YES** | **NO** | **YES** |
| **checkout (commit) [file]** | **NO** | **YES** | **YES** | **NO** |

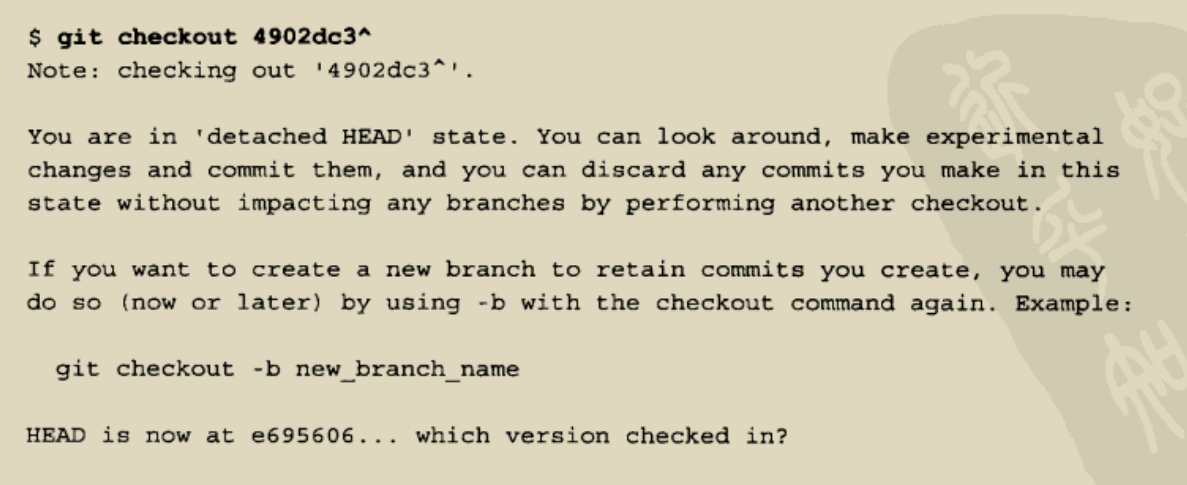
**REF-HEAD**指向的分支引用

**HEAD-HEAD**本身的指针

# Git检出

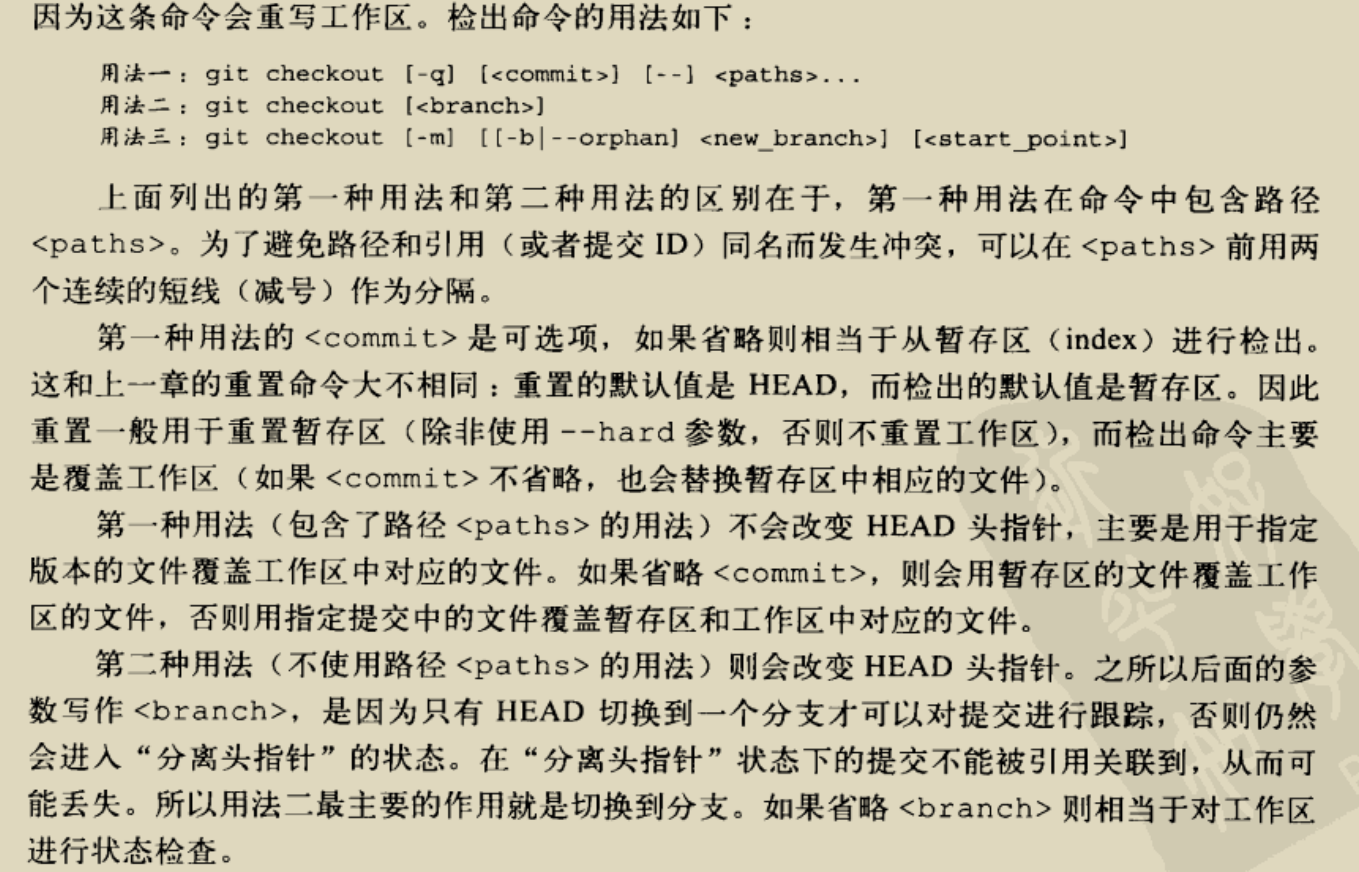
重置命令体现为分支“游标”的表更，HEAD本身一直指向的是**refs/heads/master**，并没有在重置时改变。

检出命令，实质就是修改HEAD本身的指向，该命令不会影响分支“游标”。

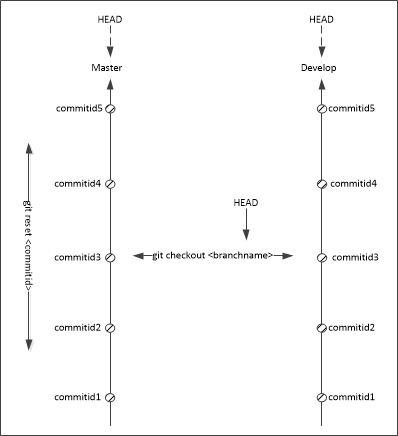


头指针分离：当前不处于任何分支，即HEAD头指针指向了一个具体的提交ID，而不是一个引用（分支）。

挽救分离头指针：**git merge <commitId>**



重置和检出的区别：



# 恢复现场

用**git stash** 保存进度，实际上相当于将进度保存在引用**refs/stash**所指向的提交中。多次的进度保存，实际上相当于引用**refs/stash**一次又一次的变化，而**refs/stash**引用的变化由**reflog**(即**.git/logs/refs/stash**)所记录下来的。

**git reflog show refs/stash** 与**git stash**的结果一样，实际上会将进度保存在引用**refs/stash**所指向的提交中。

## 一、应用场景

1.修复**bug**，通过创建新的**bug**分支进行修复，然后合并，最后删除。

2.当工作没有完成，要修复**bug**,先保护现场**git stash**,修复以后，再**git stash pop**,回到工作现场。

## 二、常用命令

1.**git stash list**：查看stash的内容

2.**git stash apply**：恢复工作区，stash内容不删除

3.**git stash apply stash@{n}**：恢复**stash@{n}**到工作区，stash内容不删除

4.**git stash drop**：删除stash内容

5.**git stash pop**：恢复现场，并删除stash内容

6.**git stash pop** = **git stash apply + git stash drop**

7.**git stash --keep-index**：告诉git不要储存任何通过**git add**命令已暂存的东西；

特别适用的场景：当你做了几个改动并只想提交其中的一部分，过一会儿再回来处理剩余改动时。（多个文件的修改，存在不同的功能点，这样可以分开提交）

8.**git stash -u (--include-untracked)**：告诉git储存任何创建的未跟踪的文件， 默认情况下，**git stash** 只会储藏已经在索引中的文件。

适用的场景：存储新建的文件和修改的文件

9.**git clean**：去除冗余文件或者清理工作目录

10.**git clean -fd** ：移除工作目录中所有未追踪的文件以及空的子目录。-f 强制 -d 删除

11.**git clean -nd**：提示可以移除那些文件

12.**git stash show -p stash@{n}** 查看stash里面的具体内容

# 改变历史

1.悔棋

**git commit –amend –m ‘message’**

2.多补悔棋-合并多个版本为一个提交

**git reset –soft HEAD^^** 将最近的两个提价压缩成一个提交。

3.回到未来

3.1 **git cherry-pick**的用法

**git cherry-pick <commitId>** 分拣某个版本，并自动合并

**git cherry-pick –n <commitId>** 分拣某个版本，不并自动合并

**git cherry-pick <commitId>..<commitId>** 批量分拣操作（左闭右开）

**git cherry-pick –-continue | –-abort | –-quit**

3.2 git rebase的用法

该命令主要用在从上游分支获取最新commit信息，并有机的将当前分支和上游分支进行合并。对所有涉及的commit("pick")执行默认操作，它只是将历史记录回滚到最后一个公共父节点，并重新生成两个分支的commit。

命令格式：**git rebase –onto <newbase> <since><till>**

变基操作的过程：

（1）首先会执行**git checkout** 切换到**<till>**

如果<till>指向的不是一个分支，则变基操作时在detached HEAD(分离头指针)状态进行的;编辑结束后，对master分支执行重置以实现变基结果在分支中生效。

（2）将**<since>…<till>**所标识的提交范围写到一个临时文件中。**<since>…<till>**范围是左开右闭。

（3）将当前分支强制重置（**git reset --hard**）到**<newbase>**。相当于执行：

**git reset --hard <newbase>**

（4）从保存在临时文件中的提交列表中，将提交逐一按顺序重新提交到重置之后的分支上。

（5）如果遇到提交已经在分支中包含，则跳过该提交。

（6）如果在提交过程中遇到冲突，则变基过程暂停。用户解决冲突后，执行**git rebase –continue** 继续变基操作。或者执行**git rebase –skip**跳过此提交。或者执行**git rebase –abort**就此终止变基操作切换到变基前的分支上。

对单个分支进行操作：**git rebase -i**

(1)、将多个提交合成一个提交

(2)、重新编辑提交的信息

(3)、对已经提交的commit进行排序

(4)、跳过某个提交

(5)、将一个提交分割成多个提交

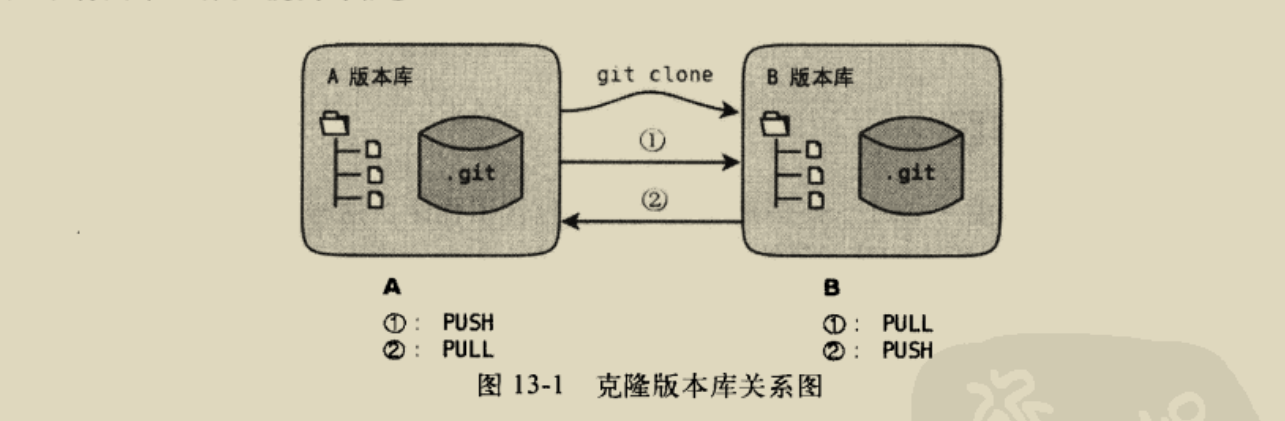
多个分支操作：合并和更新代码

对多个分支进行rebase操作，rebase命令为一个分支的更改生成补丁，然后把这个补丁应用到另外一个分支上，那么最后的源代码和merge是一样的，使用这种方式进行分支的合并更为合理。

4.反转提交用错误的历史提交的反向提交，来修正错误的历史提交。

**git revert**

# Git克隆



上图中一个项目使用了两个版本进行维护，两个版本之间通过PULL或PUSH实现同步：

①版本库A通过克隆操作创建克隆版本库B。

②版本库A可以通过PUSH（推送操作），将新提交传递给版本库B。

③版本库A可以通过PULL（拉回操作），将版本库B中的新提交拉回到自身（A）。

④版本库B可以通过PULL（拉回操作），将版本库A中的新提交拉回到自身（B）。

⑤版本库B可以通过PUSH（推送操作），将新提交传递给版本库A。

三种用法：

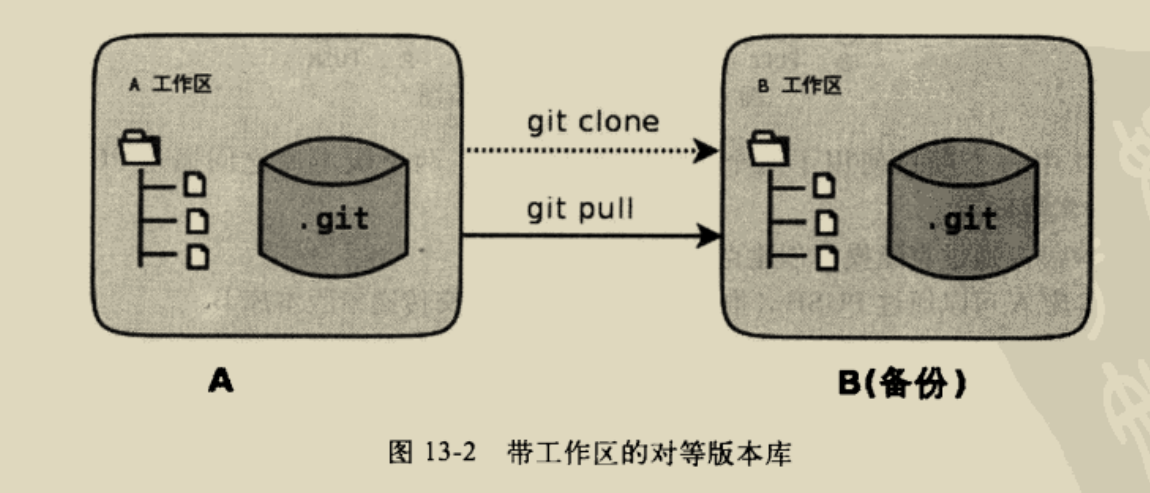
用法1：**git clone <repository> <directory>**克隆一个对等目录，既包含工作区，也包含版本库

用法2：**git clone --bare <repository> <directory.git>**

用法3：**git clone --mirror <repository> <directory.git>**

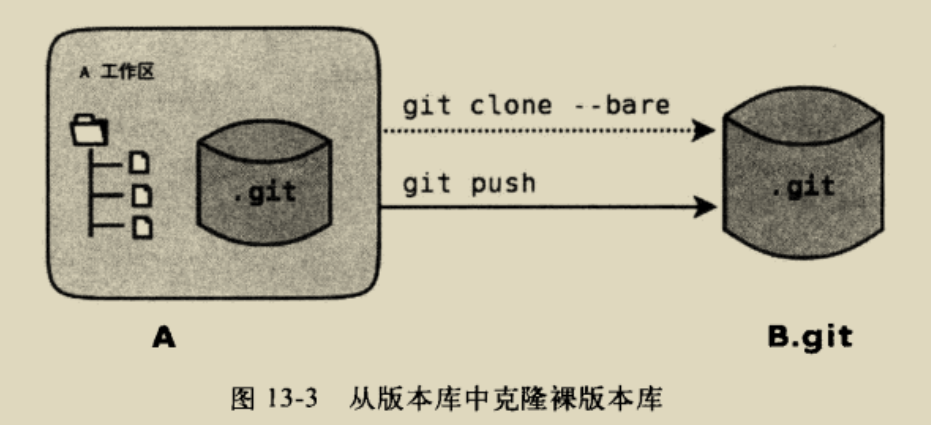
用法2和用法3创建的克隆版本库都不包含工作区，直接就是版本库，这样的版本库称为裸版本库。

1.对等工作区



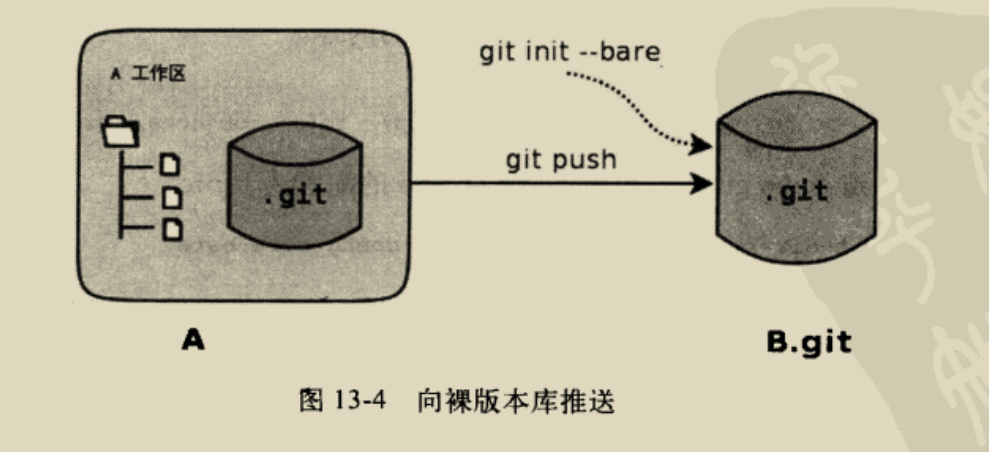
A作为提交库，B作为备份库，版本库之间的同步只有一种可行的方式，就是备份库B执行git pull命令从源库A中拉回新的提交实现版本库的同步。原因是克隆出来的版本对源版本进行了注册。

2.克隆裸版本库



3.创建生成裸版本库

**git init –bare**



|  |  |  |
| --- | --- | --- |
| 同步方式 | 对等工作区 | 裸仓库 |
| A<->B | ③④ | ② |

# git 库管理

Git对于以SHA1哈希值作为目录名和文件名保存的对象有一个术语，称为松散对象。松散对象打包以后会提高访问效率，而且不同的对象可以通过增量存储节省磁盘空间。

1、相关命令：

（1）、**git fsck** 查看版本库中包含的没有任何引用的关联的松散对象。标识为dangling的对象就是没有被任何引用直接或间接关联到的对象。

（2）、**git prune** 清除未被关联到的松散对象。该命令会清除暂存区操作时引入的临时对象，但是如果用重置命令抛弃的提交和文件就不会轻易地被清除。

（3）、**git fsck --no-reflogs**

（4）、**git reflog expire –expire=now --all** 强制让<date>之前的记录全部过期

实际情况中，很少使用git prune 命令来清理版本库。

（5）、git gc

不带参数虽然不会清除尚未过期（未到两周）的大文件，但是会将未被关联的对象从打包文件中移除，称为松散文件。故.git版本库的空间占用又反弹了。

（6）、**git gc --prune=now**

立即完成对未关联的对象的清除。

# 分支

分支的类型：

发布分支、特性分支（主题分支）、卖主分支

使用特性分支的场景：

（1）试验性、探索性的功能开发应该为其建立特性分支。

（2）功能复杂、开发周期长（有可能在本次发版中取消）的模块应该为其建立特性分支。

（3）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。

常用命令：

**git branch**

**git branch <branchname>**

**git branch <branchname> <start-point>**

**git branch –d <branchname>**

**git branch –D <branchname>**

**git checkout –b <newbranch> [<start-point>]**

**git cherry** 查看哪些提交领先（未被推送到上游跟踪分支中）

**git cherry –v <branchname>** 查看本地分支哪些提交领先远程分支branchname

**git branch –vv** 查看本分支和远程分支的跟踪关系

## 一、远程分支

**git show-ref** 查看本地分支的引用

**git branch –r** 查看远程分支

从远程版本库执行获取操作时，不是把远程版本库的分支原封不动复制到本地的分支目录中，而是复制到另外的命名空间。如在克隆一个版本库时，会将远程分支都复制到目录 .git/remotes/origin/下。这样从不同的远程版本库执行获取操作时，因为通过命名空间的相互隔离，故避免了在本地的相互覆盖。

远程分支不是真正意义上的分支，而是类似于里程碑一样的引用。如果针对远程分支执行检出操作时，会出现头指针分离的警告。

## 二、分支的追踪

1、基于远程分支创建的本地分支，会建立本地分支和远程分支的追踪。

本地分支就具有下列特征：

(1).检查工作区时，会显示本地分支和被跟踪远程分支提交之间的关系。

(2).当执行git pull命令时，会和被跟踪的远程分支进行和并（或者变基）。

(3).当执行git push命令时，会推送到远程版本库的同名分支中。

2、基于一个本地分支创建的另一个本地分支，不会建立分支间的跟踪。

## 三、远程版本库

**git remote –v**

**git remote**

push命令形式：**git push <host> <localbranchname> <remotebranchname>**

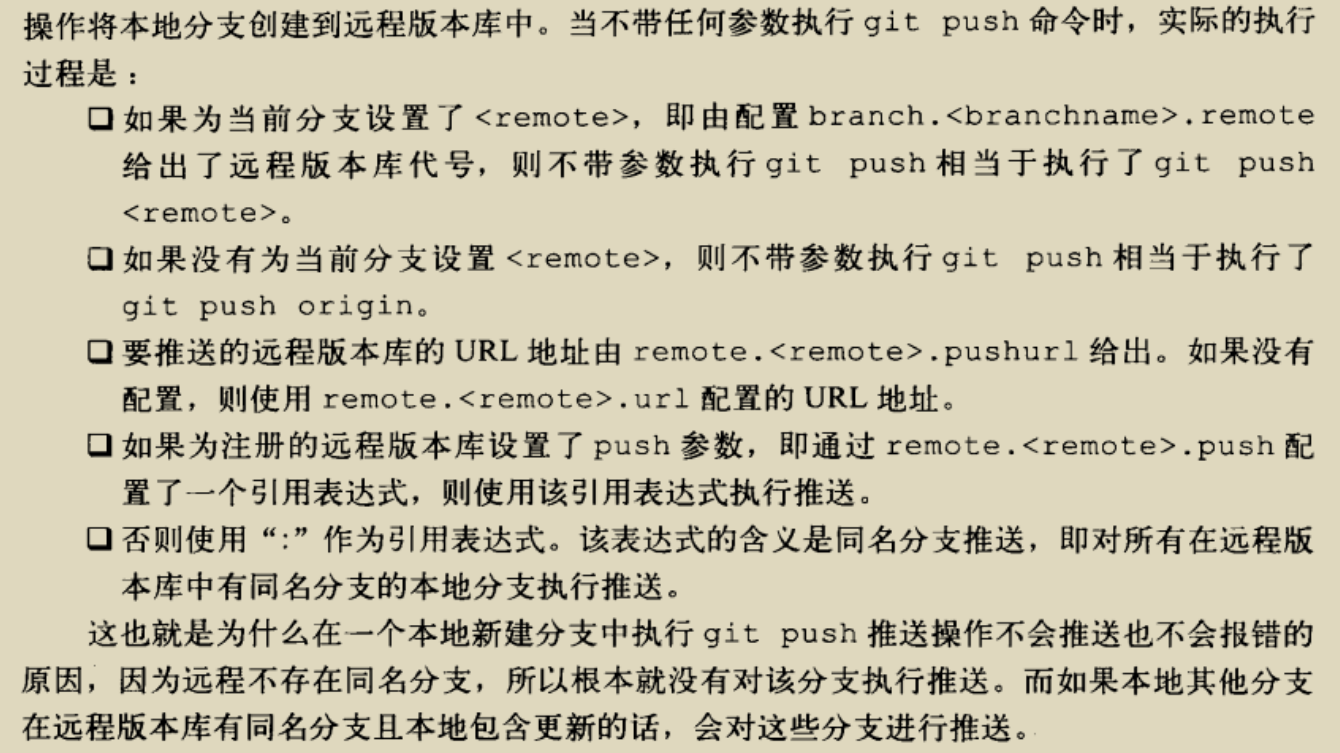
(1)、**git push origin <branchname>**

如果远程分支被省略，如上则表示将本地分支推送到与之存在追踪关系的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

(2)、**git push origin <branchname1>:< branchname2>**

(3)、**git push origin :< branchname2>**

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支，等同于**git push origin --delete master**。



# Git合并

## 一、常用命令

**1、git merge [param选填] <branchname 必填>**

**--no-commit：**合并，但是不产生commit,只是添加到暂存区。

**--no-ff：**非快速合并，合并是要产生一个新的提交版本信息。

**--ff：**快速合并模式，不会产生一个新的提交版本信息。

**--ff-only：**只允许快速合并模式。

**-e：**可以修改合并提交的commit信息。

**--squash：**压缩成一个提交信息。

融合的两种模式：

快速（fast-forward）：融合不会产生新的提交版本信息。

真正融合：会产生一个新的提交版本信息。

**2、git merge –-abort | --continue | --quit**

## 二、合并的策略

Git 支持很多合并策略，默认会选择最合适的合并策略。可以通过传递参数使用指定的合并策略，命令如下：

**git merge [-s <strategy>] [-X < strategy -option>] <commitid>**

**-s**用于设置合并策略，**-x**用于为所选的合并策略提供附件参数。

合并策略如下：

（1）resolve

该策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略被认为是最安全、最快的合并策略。

（2）recursive-递归

该策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。该合并策略是合并两个头指针的默认合并策略。

当合并的头指针拥有一个以上的祖先的时候，会针对多个公共祖先创建一个合并树，并以此作为三向合并的参照。这个合并策略被认为是实现冲突的最小化，而且可以发现和处理由于重命名导致的合并冲突。

该策略可以使用以下选项：

A、ours

遇到冲突时，选择当前分支的版本，忽略他人的版本。

B、their

与ours相反，遇到冲突时，选择他人的版本,忽略当前分支的版本。

C、subtree[=tree]-子树合并策略

（3）octopus

可以合并两个以上的头指针，但是拒绝执行需要手动解决的复杂合并。主要用途是将多个主题分支合并到一起。这个合并策略是对三个及三个以上的头指针进行合并时的默认合并策略。

（4）ours

可以合并任意数量的头指针，但是合并的结果总是使用当前分支的内容，丢弃其他分支的内容。

（5）subtree

## 三、冲突解决

1、合并一：自动合并

（1）修改不同的文件

（2）修改相同的文件的不同区域

（3）同时更改文件名和文件内容

2、合并二：逻辑冲突

3、合并三：冲突解决

（1）手工编辑解决冲突

（2）图形工具完成冲突解决

4、合并四：树冲突

# 常用的Git命令

|  |  |
| --- | --- |
| 命令 | 简要说明 |
| **git add** | 添加至暂存区 |
| **git add --interactive** | 交互式添加 |
| **git annotate** | 同义词，git blame |
| **git branch** | 分支管理 |
| **git cat-file** | 版本库对象研究工具 |
| **git checkout** | 检出到工作区、切换或创建分支 |
| **git cherry-pick** | 提交拣选 |
| **git clean** | 清除工作区未跟踪的文件 |
| **git clone** | 克隆版本库 |
| **git commit** | 提交 |
| **git config** | 查询和修改配置 |
| **git difftool** | 调用图形差异化比较工具 |
| **git fetch** | 获取远程版本库的提交 |
| **git grep** | 文件内容搜索定位工具 |
| **git help** | 帮助 |
| **git init** | 版本库初始化 |
| **git log** | 查看提交日志 |
| **git merge** | 分支合并 |
| **git mergetool** | 图形化冲突解决 |
| **git push** | 推送至远程版本库 |
| **git pull** | 拉回远程版本库的提交 |
| **git rebase** | 分支变基 |
| **git rebase--interactive** | 交互式分支变基 |
| **git reflog** | 分支等引用变更记录管理 |
| **git remote** | 远程版本库管理 |
| **git reset** | 重新改变分支“游标”指向 |
| **git rev-parse** | 将各种引用表示法转换为哈希值等 |
| **git revert** | 反转提交 |
| **git show** | 显示各种类型的对象 |
| **git stash** | 保存和恢复对象 |
| **git status** | 显示工作区文件状态 |
| **git tags** | 标签管理 |

对象库操作相关命令

|  |  |
| --- | --- |
| **git ls-files** | 显示工作区和暂存区的文件 |
| **git ls-tree** | 显示树对象包含的文件 |

引用操作相关命令

|  |  |
| --- | --- |
| **git ls-remote** | 显示远程版本库的引用 |
| **git show-branch** | 显示分支列表及拓扑关系 |
| **git show-ref** | 在本地存储库中列出引用 |

版本管理相关命令

|  |  |
| --- | --- |
| **git count-objects** | 显示松散对象的数量和磁盘占用 |
| **git fsck** | 对象库完整性检查 |
| **git gc** | 版本库存储优化 |
| **git prune** | 从对象库中删除过期对象 |

其他

|  |  |
| --- | --- |
| **git cherry** | 查找没有和并到上游的提交 |