# Git使用简单介绍

## 背景故事

### 历史

Git开发是开始于2005年4月的，那是在许多linux内核开发人员放弃之前用来维护项目的BitKeeper(一个专有的源码控制管理系统)之后发生的。BitKeeper的版权持有者Larry McVoy(BitKeeper公司的CEO)，在声明Andrew Tridgell反编译了BitKeeper的协议后，撤回了该产品的免费使用。(一样的事情也推动了另一个版本控制系统的创建，[Mercurial](https://en.wikipedia.org/wiki/Mercurial))

Linus Torvalds想要一个可以像BitKeeper一样使用的分布式系统，但是他并没有找到可用的免费的满足他需求的系统。Torvalds引用了一个例子，源码控制管理系统需要30秒应用一个patch(补丁)和更新所有的关联数据，并且注意到了，当同步其他代码维护者时这样的行为可能需要250秒一次，这将无法扩展linux内核开发人员的需。他指出应用patch(补丁)应该花费不超过3秒，并且添加了额外的三点：

* 以并发版本系统作例子，说明不该做的事情；如果存在疑问，选择完全相反的决定
* 支持分布式，类似于BitKeeper的工作流程
* 包括更健壮的安全指导，不管无意或者恶意的

这些准则淘汰了当时存在的每一个版本控制系统，所以在2.6.12-rc2内核开发发布后，Torvalds马上着手自己开发。

开发工作开始于2005年4月3号，Torvalds在4月6号公布了项目，7号发布在了本机，第一个多版本合并，发生在18号。Torvalds在4月29号达到了他的性能目标，这个新生的Git达到了，补丁应用到linux内核树，每秒6.7个补丁的速率的指标。6月16号，Git管理了linux内核2.6.12版本的发布。

Torvalds在2005年7月26号将维护工作移交给了Junio Hamano(该项目的主要贡献者)。Hamano负责在12月21号发布1.0版本，并且仍然是该项目的维护者。

### 命名

“git”这个名称是Linus Torvalds在非常快的第一个版本中给定的。他用这名字描述该工具是”愚蠢的内容追踪”，这个名字怎么理解取决于你自己：

* 随机3个字母组合可拼读的且不是UNIX实际使用的命令。事实上它是”get”的错误拼写可能是也可能不是。
* 愚蠢的，卑鄙的，简单的，从你的俚语字典中挑选出来。
* “全局信息追踪”:你拥有一个好心情，并且事实上对你很有帮助。仿佛天使在唱歌，并且一道圣光照耀进了房间。
* “狗屎该死的白痴卡车”，当它出异常时。

## 基础概念

### 协同工作方式

#### 集中式版本控制系统

这类系统，诸如 CVS、Subversion 以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。



#### 分布式版本控制系统

Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。 这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。



### 数据保存

#### 存储文件变更列表

以文件变更列表的方式存储信息。 这类系统（CVS、Subversion、Perforce、Bazaar 等等）将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。



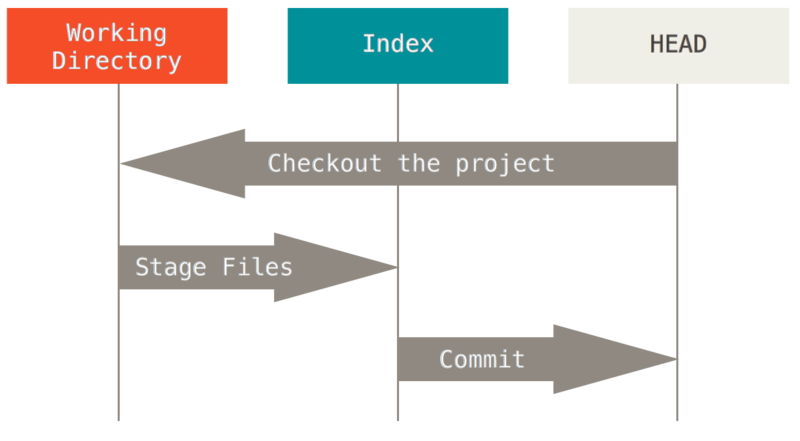
#### 存储文件版本快照

Git 不按照以上方式对待或保存数据。 反之，Git 更像是把数据看作是对小型文件系统的一组快照。 每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个 快照流



### 本地仓库组成

#### 三个区域



##### 工作目录(working dir)

持有实际文件，文件修改和变更直接影响的区域

##### 暂存区(Index)

我们也会将这个概念引用为 Git 的 “暂存区域”，也称为索引，是你预期的下一次提交。

##### HEAD区

它总是指向该分支上的最后一次提交，HEAD 是当前分支引用的指针。 这表示 HEAD 将是下一次提交的父结点。 通常，理解 HEAD 的最简方式，就是将它看做 你的上一次提交 的快照。

### 提交

#### 概念

每次提交的实质，Git保存一个提交对象(commit object).

该提交对象包含了以下信息

* 提交对象本身(包含了作者的姓名和邮箱、提交时输入的信息)
* 指向文件树(项目根目录)的指针，该文件树保存了文件快照(blob对象)指针
* 上一次提交对象(父对象)的指针(首次提交没有，合并提交则有多个)



多次提交后



### 分支

#### 概念

分支是包含一系列提交节点组成的链表上的一个指针，指针指向一个具体的提交节点。Git 的分支，其实本质上仅仅是指向提交对象的可变指针。



分支切换，则HEAD指向的分支也跟着变化

#### HEAD

这是一个特殊指针，名称为HEAD，Git用于标识当前处于哪个分支之上。



$ git checkout testing



#### 本地分支

##### 创建分支

git checkout -b [branch-name]

也可以拆开为两条命令

git branch [branch-name]

git checkout [branch-name]



##### 分支上的提交

本质上就是增加了一个commit节点，并将分支代表的指针向前移动。

下图展示了，在iss53的分支上进行了一次提交。



##### 分支的合并

###### 快进合并

在上文情况下，你需要基于master分支，进行问题修复，所以切换到了master，并新建了分支hotfix，然后在hotfix上进行了修复并提交(C4)。



接下来，使用git merge合并

先切换到master

git checkout master

合并hotfix分支

git merge hotfix

该情况下，Git就会进行快进(fast-forward)合并, 由于当前 master 分支所指向的提交是你当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动。 换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进（fast-forward）”。



###### 典型合并

当非直接上游的两个分支合并时，即为冲突分支。

此时会进行简单的三方合并。

git checkout master

git merge iss53



和之前将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。



冲突文件处理完毕，需要git add

最后进行git committ提交

#### 远程分支

使用git clone将远程仓库克隆下来后，git会产生两个分支指针：

其中一个为origin/master(默认名称) 是远程分支状态的引用，只有与远程仓库发生了网络通信后，才有可能发生变动。这个指针也是仅存在本地仓库。

其二为与该指针指向相同的master指针，当你在工作空间表更提交时，该指针则向前移动。



当你在本地的 master 分支做了一些工作，然而在同一时间，其他人推送提交到 git.ourcompany.com 并更新了它的 master 分支，那么你的提交历史将向不同的方向前进。



使用git fetch 更新你的远程仓库引用



## 快速开始

### 下载

Windos下载地址:

<https://gitforwindows.org/>

Mac下载地址：

<https://git-scm.com/download/mac>

Linux下载介绍(命令行下载安装)：

<https://book.git-scm.com/download/linux>

### 安装

Window:next大法好

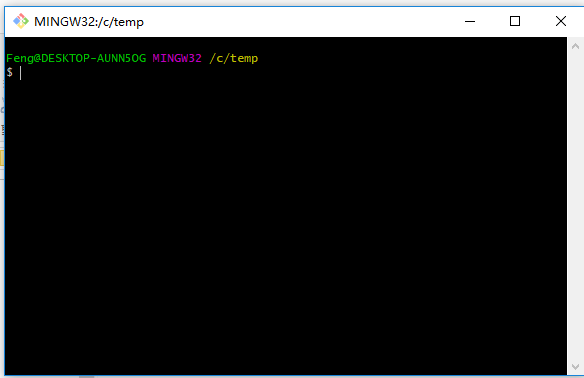
Mac:省略

Linux:yum install git

### Window特别说明

鼠标右键打开Git bash here选项

打开命令行窗口，即后续命令行参数皆是于此执行



### 基本用法

#### First Repository

##### 创建新仓库

1. 创建新文件夹
2. 进入文件夹
3. 执行命令

**git init**

##### 检出仓库

###### 本地仓库

创建本地仓库的一个克隆版本(在当前目录下创建，目标目录需要是一个git仓库)

**git clone /path/to/repository**

###### 远程仓库

拉取远程服务器的仓库克隆

**git clone username@host:/path/to/repository**

或者**git clone https://github.com/xx/xx.git**

#### 添加/提交

添加新增/修改的文件(到暂存区)，多功能命令

1. 开始追踪文件，
2. 把已追踪文件的变更提交到暂存区，
3. 合并时把冲突文件标记为已解决

**git add <filename>**

**git add \***(添加所有)

将(暂存区)数据提交到HEAD

**git commit –m** “提交内容描述”

#### 推送

推送本地HEAD仓库改动到远程仓库

**git push origin <master>**

master为远程分支名称

不是克隆现有仓库情况下，需先绑定远程分支

#### 绑定/解绑远程分支

将本地仓库绑定到远程仓库

**git remote add <shortname> <url>**

随后可以将本地变更推送至远程

解除本地仓库的绑定

**git remote remove origin**

#### 分支

##### 创建本地分支

**git checkout -b <branch-name>**

##### 切换分支

**git checkout <branch-name>**

##### 删除分支

**git branch -d <branch-name>**

##### 推送分支

**git push origin <branch-name>**

**未推送的分支仅仅自己本地可见**

#### 更新/合并

##### 更新本地到最新

**git pull**

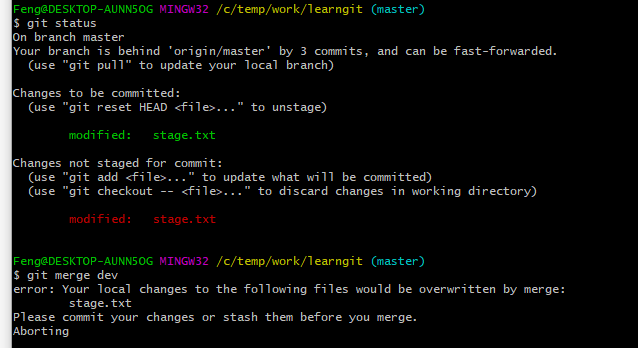
##### 合并其他分支到当前分支

**git merge < branch-name>**

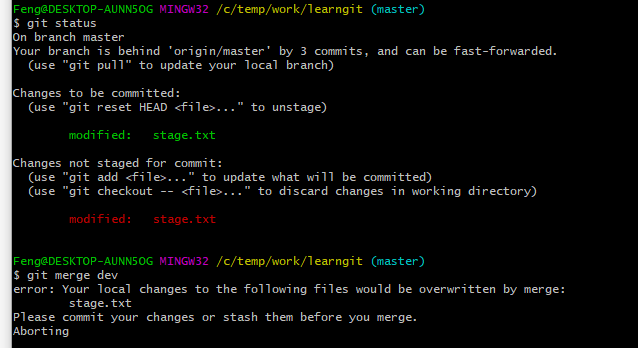
**以上两种操作，都可能导致自动合并产生冲突**

**需要手动处理冲突，并重新标记为合并 git add <filename>**

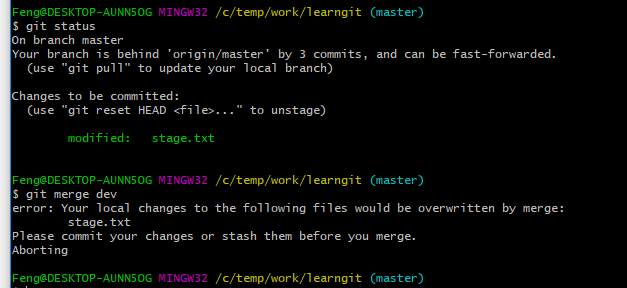
**合并前要求提交已经add的变更**



**只有工作目录有变更也不能合并。**



**只有暂存区有未提交内容也不能合并。**



##### 对比差异

**git diff <source\_branch> <target\_branch>**

#### 标签

##### 创建标签

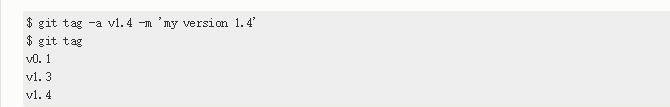
**git tag tag-1.0.0 1b2e1d63ff**

**其中tag-1.0.0为标签名，1b2e1d63ff 为commit id的前10位(不一定是10位，可以唯一标识一个commit即可)，可以通过下面的LOG相关指令获取。**

**Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）。**

**一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。**

**附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证。**



##### 展示具体的标签信息

**git show [tag-name]**



##### 共享标签

**git push origin v1.5**

默认情况下，git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上。

**git push origin –tags**

如果想要一次性推送很多标签，也可以使用带有 --tags 选项的 git push 命令。 这将会把所有不在远程仓库服务器上的标签全部传送到那里。

#### LOG

##### 查看本地仓库历史记录

**git log**

##### 指定参数查询提交记录

**git log --author=authorName**

**查询指定提交人的记录**

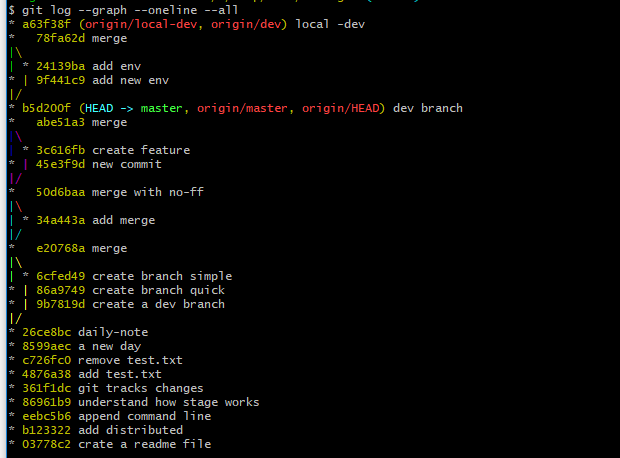
**git log --pretty=oneline**

**每条提交记录只占一行**

**git log --graph --oneline –all**

**查看提交树与各个分支指针的位置**

**q键退出**



**git log –help**

**查看其他可用参数**

#### 替换本地

##### 撤销提交到暂存区的文件

**git reset HEAD <filename>**

**该操作将撤销使用git add提交到暂存区的影响**

##### 撤销文件在工作目录中的变更

**git checkout -- <filename>­**

**即指定文件的变更被HEAD版本覆盖，放弃工作目录的修改**

**这是个危险的命令，拷贝了另一个文件来覆盖它。 除非你确实清楚不想要那个文件了，否则不要使用这个命令。**

##### 放弃本地所有改动与提交，使用远程服务最新版本

**git fetch origin**

**本地分支指向远程库**

**git reset --hard origin/master**

**使用远程分支覆盖本地库**

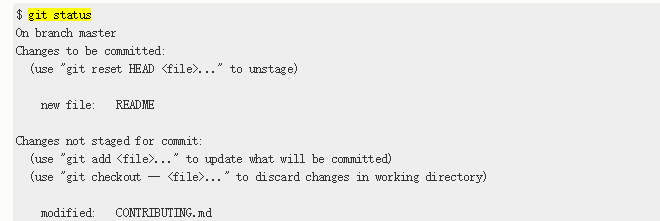
### 实用命令

#### git status

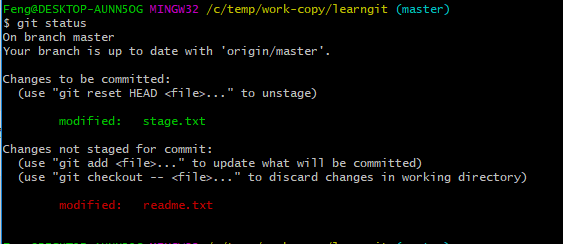
检查当前文件状态，并提示可以如何操作这些文件。



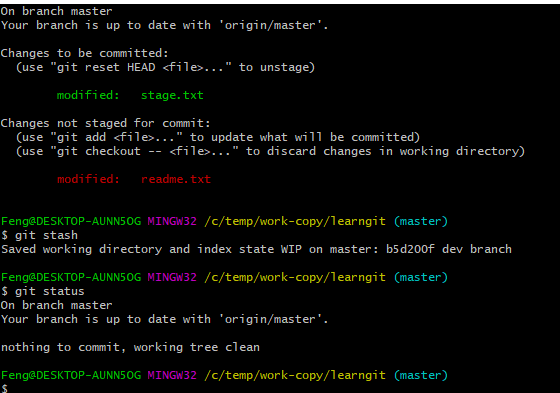




#### 储藏工作

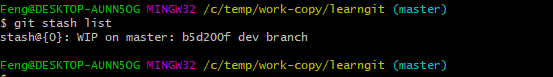


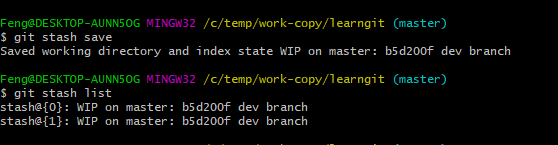
##### git stash 或 git stash save



未add的变更和已经add的变更都会被储藏起来

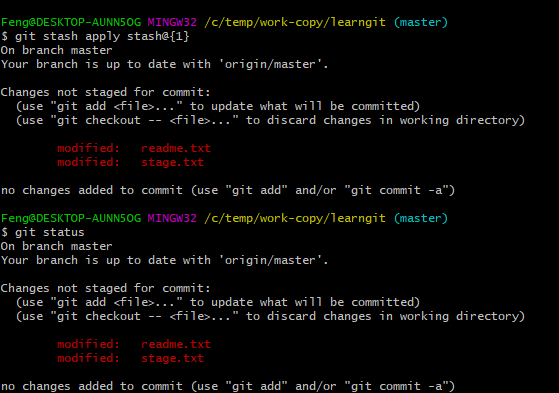
##### git stash list





##### git stash apply

可以指定名称(stash@{0})应用储藏的内容，不指定的话，默认使用最近一次储藏。



可以发现原本add的操作被撤销了。

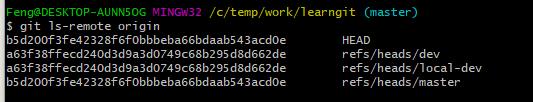
一些关于储藏的高级操作

<https://git-scm.com/book/zh/v2/Git-%E5%B7%A5%E5%85%B7-%E5%82%A8%E8%97%8F%E4%B8%8E%E6%B8%85%E7%90%86>

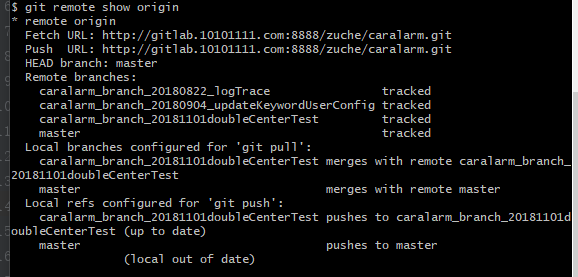
#### 远程分支

##### 查看命令

git ls-remote (remote)



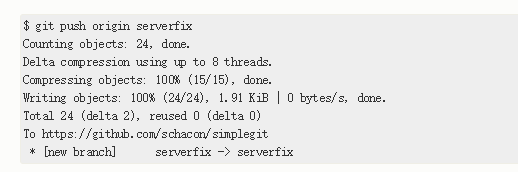
git remote show (remote)



##### 将分支推送至远程仓库

git push (remote) (branch)

git push (remote) (branch):(remote-branch)

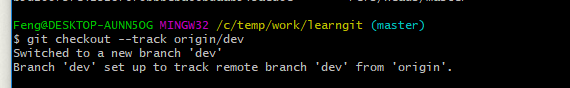


使用第二个命令可以指定远程分支的名称。

##### 创建跟踪分支

git checkout -b [branch] [remotename]/[branch]

git checkout --track origin/serverfix



当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。第一个命令指定了本地分支的名称，第二个命令使用相同的名称。

##### 指定跟踪的远程分支

git branch -u origin/serverfix

置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。

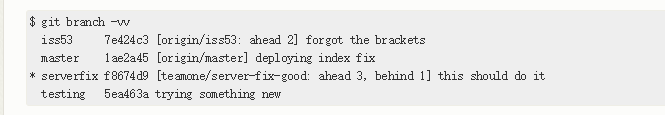
##### 删除当前分支的上游

git branch --unset-upstream

该命令可以删除当前分支绑定的上游

##### 查看本地分支跟踪情况

git branch –vv



##### 删除远程分支

git push origin --delete serverfix

#### Diff

##### git diff

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容

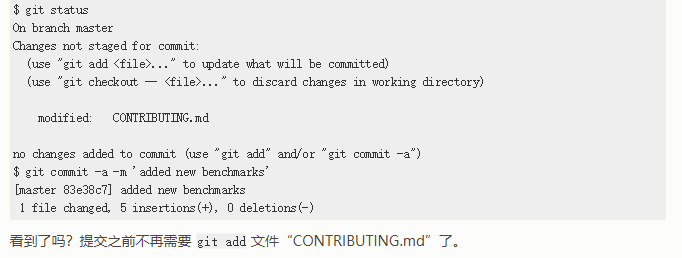
##### git diff –cached

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容查看已暂存的将要添加到下次提交里的内容，可以用 git diff --cached 命令。（Git 1.6.1 及更高版本还允许使用 git diff --staged，效果是相同的，但更好记些。）

#### 跳过使用暂存区

##### git commit –a –m “skip use stage”

Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤



#### 移除文件

##### rm <filename>

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分（也就是 未暂存清单）。

##### git rm <filename>

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。 可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

下一次提交时，该文件就不再纳入版本管理了。 如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f（译注：即 force 的首字母）。 这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

##### git rm –cached <filename>

把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 --cached 选项。

#### 文件改名

##### git mv file\_from file\_to

Git 并不显式跟踪文件移动操作。 如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。

其实，运行 git mv 就相当于运行了下面三条命令：

$ mv README.md README

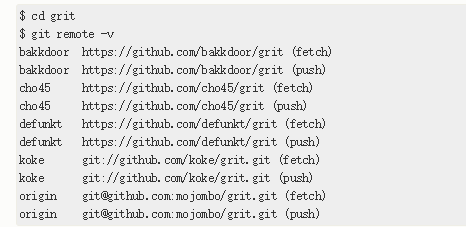
$ git rm README.md

$ git add README

#### 远程仓库

##### 查看远程仓库

###### git remote –v



###### git remote show [remote-name]

查看对应远程的具体信息



##### 抓取远程仓库

###### git fetch [remote-name]

这个命令会访问远程仓库，从中拉取所有你还没有的数据。 执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

该命令抓取克隆（或上一次抓取）后新推送的所有工作。 必须注意 git fetch 命令会将数据拉取到你的本地仓库 - 它并不会自动合并或修改你当前的工作。 当准备好时你必须手动将其合并入你的工作。

##### 拉取远程仓库

###### git pull

##### 移除与重命名

###### git remote rename [old\_name] [new\_name]

将远程仓库重命名，该命名仅仅针对的本地仓库

###### git remote rm [remote\_name]

与git remote remove 命令一致

移除对应的远程仓库

#### 分支管理

##### 查看分支

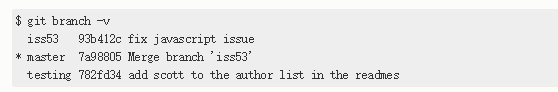
\*号为当前分支，HEAD指针指向的分支

git branch



git branch -v

查看分支和最后一次提交



##### 查看已合并分支

git branch --merged



##### 查看未合并分支

git branch --no-merged



未合并分支一般不能直接删除。

### 特别命令

#### 变基

##### 原理

首先找到这两个分支（即当前分支 experiment、变基操作的目标基底分支 master）的最近共同祖先 C2，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。



切换到experiment分支

git checkout experiment

将experiment分支的修改提交到master分支

git rebase master



变基完成后，需要合并(快进合并)

切换为master分支

git checkout master

合并(快进合并)

git merge experiment



##### 应用场景

不要对在你的仓库外有副本的分支执行变基。

只要你把变基命令当作是在推送前清理提交使之整洁的工具，并且只在从未推送至共用仓库的提交上执行变基命令，就不会有事。 假如在那些已经被推送至共用仓库的提交上执行变基命令，并因此丢弃了一些别人的开发所基于的提交，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

## 常用需求

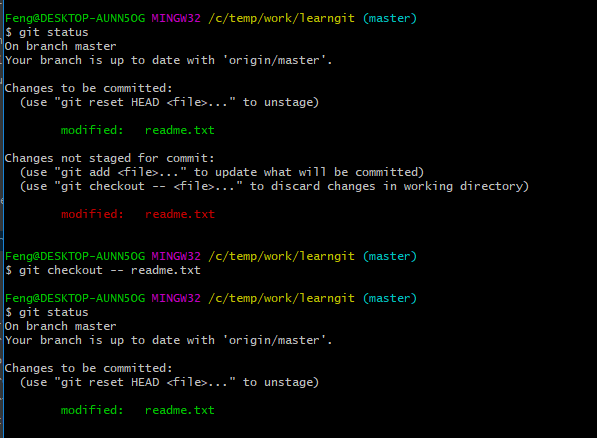
### 回滚修改

此处回滚修改指的是，回滚工作空间的修改。

#### 命令行

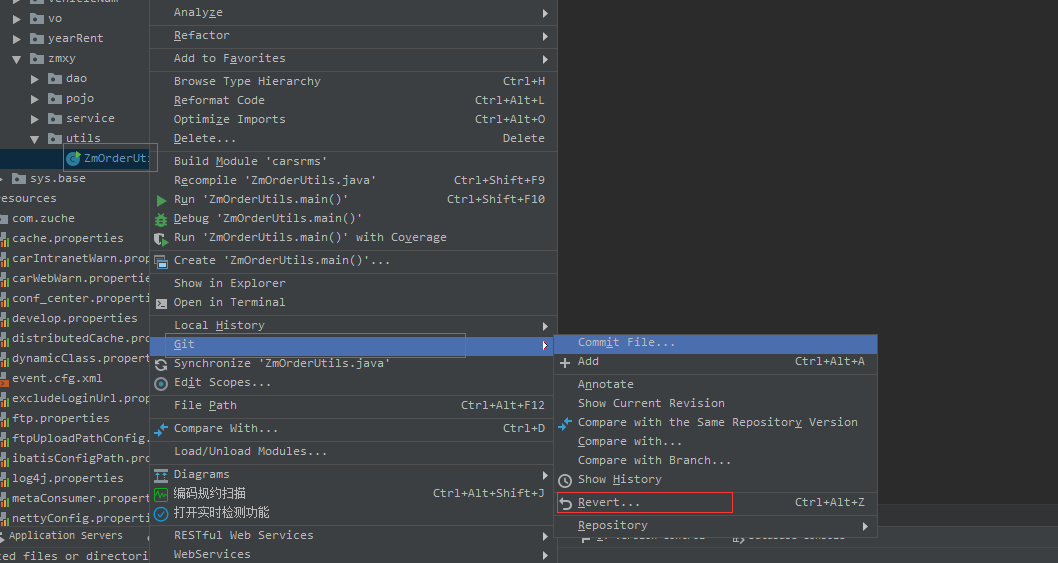
git checkout -- <file>

该命令会使用暂存区的文件的版本覆盖工作空间对应的文件。即已经add过但是未commit的变更可以保存，未add的修改将会被放弃。

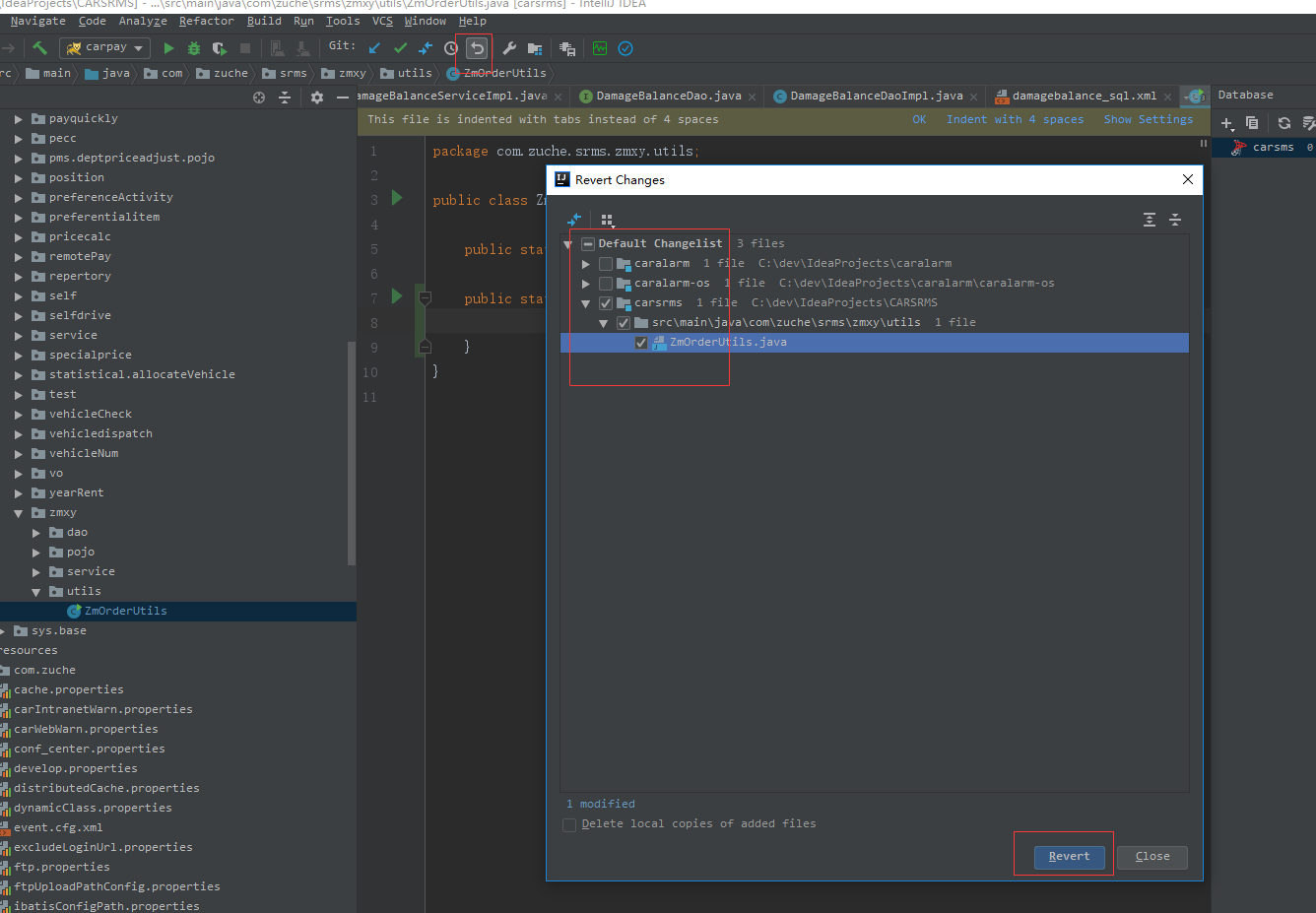


#### IDEA中操作

操作具体文件，选中文件，鼠标右键菜单。



直接工具栏操作



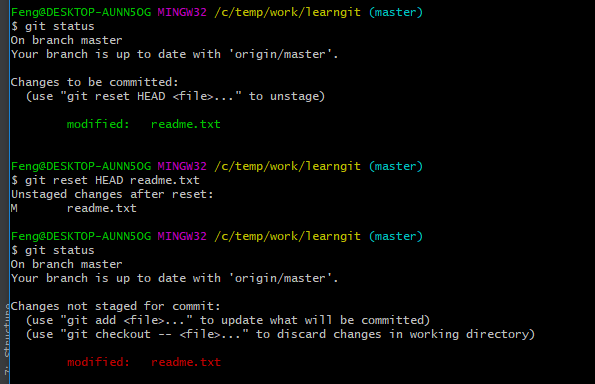
### 回滚add

该回滚指的是，撤销add操作，即提交到暂存区的变更

#### 命令行

git reset HEAD <file>

该命令会将暂存区的文件使用HEAD中的版本(最后一次提交的版本)替换，丢弃这期间的所有add变更，工作空间的修改仍存在，不会被修改。



#### 新文件撤销add(即不进行版本控制)

应用场景为：新增文件不想进行版本控制，不小心使用了git add导致添加到了版本控制。

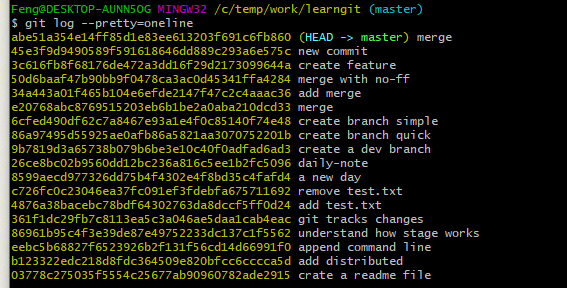
撤销命令

git rm –cached [filename]

### 回滚commit

#### 查看commit\_id

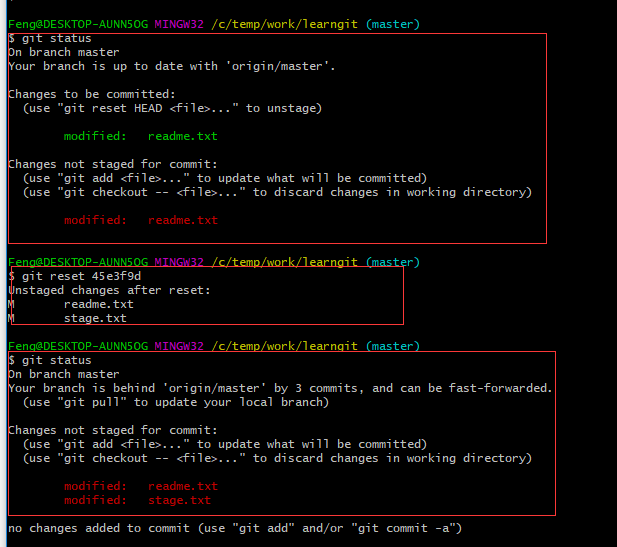
git log --pretty=oneline



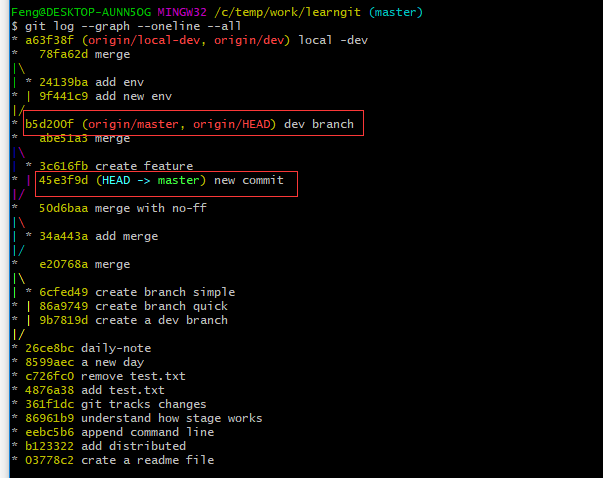
#### 保留变更回滚

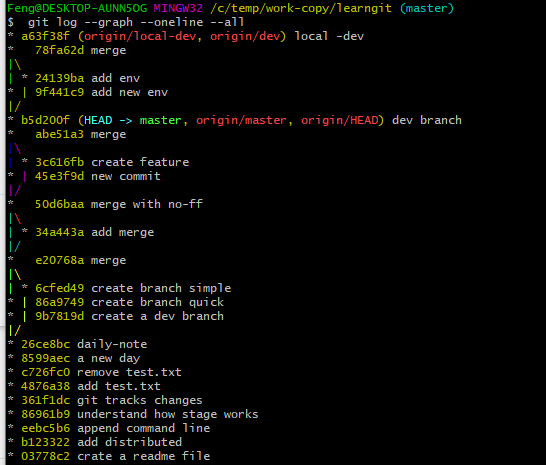
git reset [commit\_id]

该命令会将本地当前分支指针指向对应的commit\_id节点，清除暂存区的提交(add的内容)，其他变更(以commit\_id版本为对照)仍旧体现在工作区，即需要重新add然后commit。



指针变更，原本HEAD->master是与origin/master执行同一个提交节点的。

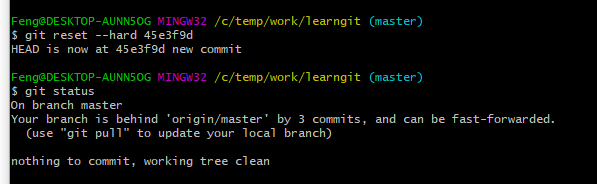




#### 清除变更回滚

git reset --hard [commit\_id]

慎用该命令，该命令与上述命令的差异为，本地工作空间的变更被清除，且无法找回(已提交的可以找回)，工作空间文件版本与当前分支指针(此时指向commit\_id)一致。



### 回滚push

#### 第一步，回滚commit

首先先根据上面的操作，回滚到需要的commit节点。

#### 第二步，将本地分支推送至远程

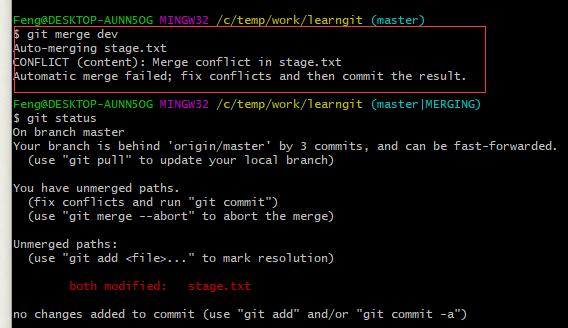
git push origin branch --force

### 回滚merge

#### 中断合并

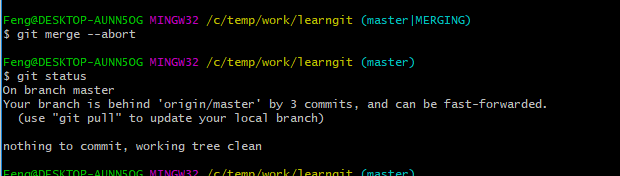
##### 适用场景

典型合并，出现了冲突文件，还未解决冲突(未add)，未进行提交(未commit)的情况下，放弃合并。

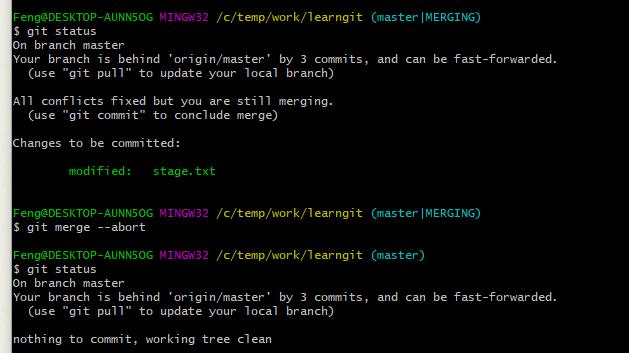


##### 解决方案

git merge –abort



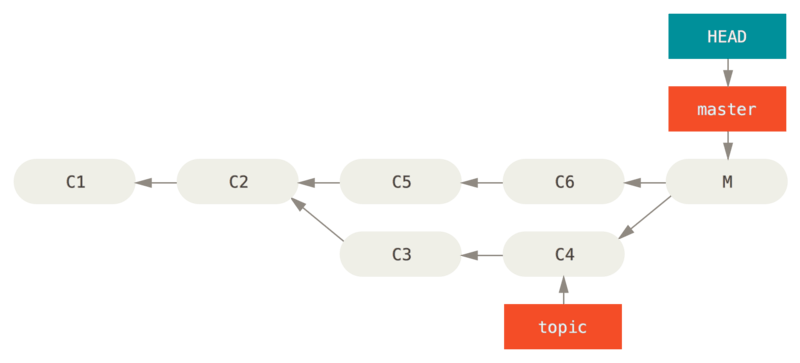
修改了冲突并且add之后，也可以直接使用该命令，放弃合并，撤销暂存区的修改。



#### 撤销合并

##### 使用场景

已经合并，但是合并完发现，这是一个错误的合并。



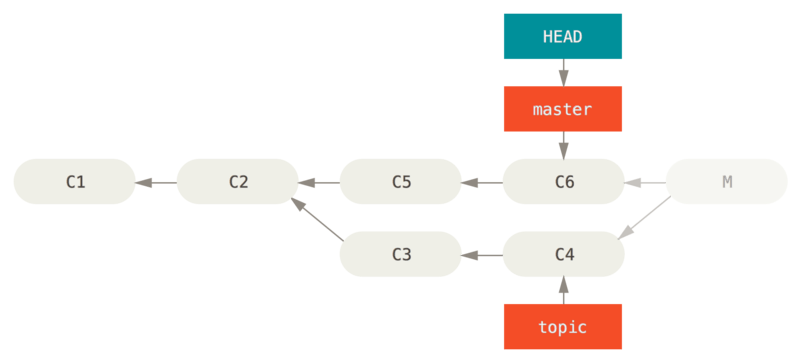
##### 解决方案

###### 合并提交只存在于本地仓库

git reset --hard HEAD~

或git reset --hard commit\_id

直接就可以撤销合并操作

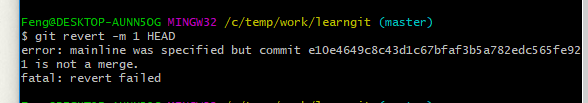


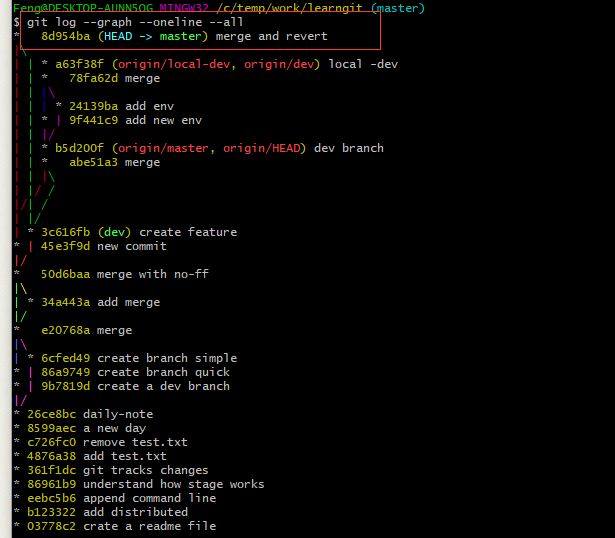
###### 合并提交已推送至远程仓库

创建新的提交和撤回合并的内容

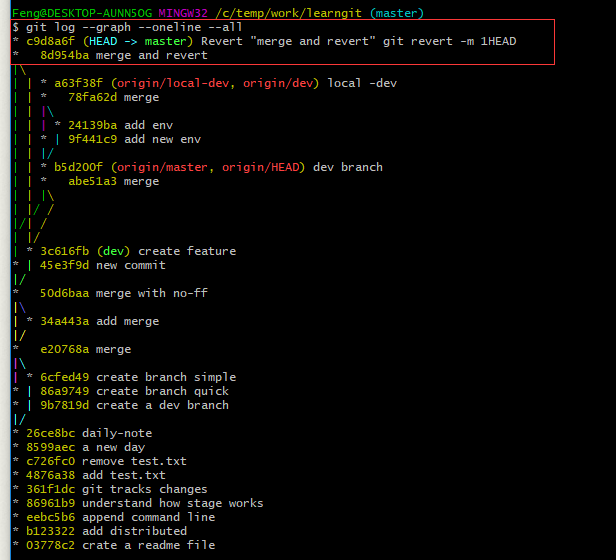
git revert -m 1 HEAD

该命令只能执行在合并的提交节点上(即多个父节点)，否则无法执行





使用该命令之后，Git 生成一个新提交的选项，提交将会撤消一个已存在提交的所有修改。



通过调整HEAD指针(使用git reset –hard)，可以查看到3c616fb 上的修改被撤销了，

c9d8a6f版本上只包含45e3f9d上的变更。

最终将新的提交推送至远程，不会对提交历史发生变更，从而导致其他人员的混乱。

恢复被删除的分支(被回收前)

git branch branchName <sha1>

### reset和revert命令的区别

#### reset

主要是指针操作

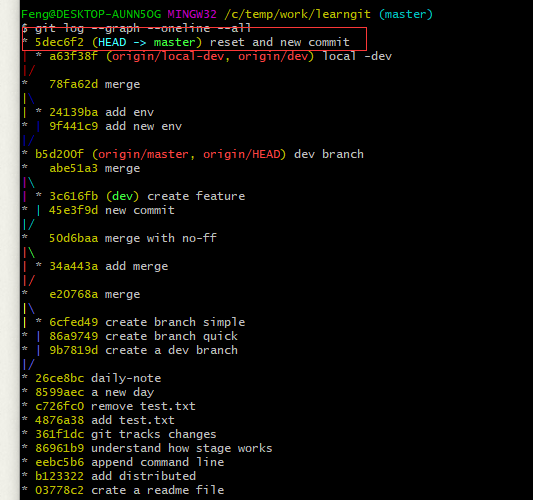
##### 操作文件

git reset HEAD <file>

该命令会将暂存区的文件使用HEAD中的版本(最后一次提交的版本)替换，丢弃这期间的所有add变更，工作空间的修改仍存在，不会被修改。

##### 操作commit-id

将HEAD指针切换到具体分支的指定节点(commit-id)，此时若进行了提交，则会产生新的分支。



#### Revert

以创建一个新的提交的形式，来撤销已提交变更内容。

##### 撤销合并

git revert -m 1 HEAD

要求目标commit\_id必须是一个合并节点(多个父节点)

生成新的提交节点，继承选中的父节点，撤销未选中节点的变更内容

##### 撤销提交

git revert HEAD

要求目要求目标commit\_id不是一个合并节点(多个父节点)

生成新的节点，继承选中节点之前的内容，之后的提交内容直接撤销。

## 实用贴士

### 设置全局邮件和名称

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址

**git config --global user.name "Your Name"**

**git config --global user.email "email@example.com"**

## 参考文档

<https://git-scm.com/book/zh/v2>