`git reset` 命令用于将当前 `HEAD` 复位到指定状态。一般用于撤消之前的一些操作(如：`git add`/`git commit` 等)。

简介

```
git reset [-q] [<tree-ish>] [--] <paths>…
git reset (--patch | -p) [<tree-ish>] [--] [<paths>…]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

描述

在第一和第二种形式中，将条目从`<tree-ish>`复制到索引。 在第三种形式中，将当前分支头(`HEAD`)设置为`<commit>`，可选择修改索引和工作树进行匹配。所有形式的`<tree-ish>/<commit>`默认为 `HEAD` 。

这里的 `HEAD` 关键字指的是当前分支最末梢最新的一个提交。也就是版本库中该分支上的最新版本。

示例

以下是一些示例 -

在 git 的一般使用中，如果发现错误的将不想暂存的文件被 `git add` 进入索引之后，想回退取消，则可以使用命令：`git reset HEAD <file>`，同时 `git add` 完毕之后，git 也会做相应的提示，比如：

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   test.py
```

`git reset [--hard|soft|mixed|merge|keep] [<commit>或 HEAD]`：将当前的分支重设(`reset`)到指定的`<commit>`或者 `HEAD`(默认，如果不显示指定`<commit>`，默认是 `HEAD`，即最新的一次提交)，并且根据`[mode]`有可能更新索引和工作目录。`mode` 的取值可以是 `hard`、`soft`、`mixed`、`merged`、`keep`。下面来详细说明每种模式的意义和效果。

A). `--hard`：重设(reset) 索引和工作目录，自从`<commit>`以来在工作目录中的任何改变都被丢弃，并把 HEAD 指向`<commit>`。

下面是具体一个例子，假设有三个 commit， 执行 `git status` 结果如下:

```
commit3: add test3.c
commit2: add test2.c
commit1: add test1.c
```

执行 `git reset --hard HEAD~1` 命令后，
显示：`HEAD is now at commit2`，运行 `git log`，如下所示 -

```
commit2: add test2.c
commit1: add test1.c
```

应用场景

下面列出一些 git reset 的典型的应用场景：

(A) 回滚添加操作

```
$ edit    file1.c file2.c           # (1)
$ git add file1.c file1.c           # (1.1) 添加两个文件到暂存
$ mailx                             #  (2)
$ git reset                           # (3)
$ git pull git://info.example.com/ nitfol    # (4)
```

(1). 编辑文件 `file1.c`, `file2.c`，做了些更改，并把更改添加到了暂存区。
(2). 查看邮件，发现某人要您执行 `git pull`，有一些改变需要合并下来。
(3). 然而，您已经把暂存区搞乱了，因为暂存区同 HEAD commit 不匹配了，但是即将 `git pull` 下来的东西不会影响已经修改的 `file1.c` 和 `file2.c`，因此可以 `revert` 这两个文件的改变。在 revert 后，那些改变应该依旧在工作目录中，因此执行 `git reset`。
(4). 然后，执行了 `git pull` 之后，自动合并，`file1.c` 和 `file2.c` 这些改变依然在工作目录中。

(B)回滚最近一次提交

```
$ git commit -a -m "这是提交的备注信息"
$ git reset --soft HEAD^      #(1)
$ edit code                         #(2) 编辑代码操作
$ git commit -a -c ORIG_HEAD  #(3)
```

(1) 当提交了之后，又发现代码没有提交完整，或者想重新编辑一下提交的信息，可执行 `git reset --soft HEAD^`，让工作目录还跟 `reset` 之前一样，不作任何改变。
`HEAD^`表示指向 `HEAD` 之前最近的一次提交。
(2) 对工作目录下的文件做修改，比如：修改文件中的代码等。
(3) 然后使用 `reset` 之前那次提交的注释、作者、日期等信息重新提交。注意，当执行 `git reset` 命令时，git 会把老的 HEAD 拷贝到文件`.git/ORIG_HEAD` 中，在命令中可以使用 ORIG_HEAD 引用这个提交。`git commit` 命令中 `-a` 参数的意思是告诉 git，自动把所有修改的和删除的文件都放进暂存区，未被 git 跟踪的新建的文件不受影响。`commit` 命令中`-c <commit>` 或者 `-C <commit>`意思是拿已经提交的对象中的信息(作者，提交者，注释，时间戳等)提交，那么这条 `git commit` 命令的意思就非常清晰了，把所有更改的文件加入暂存区，并使用上次的提交信息重新提交。

(C) 回滚最近几次提交，并把这几次提交放到指定分支中

回滚最近几次提交，并把这几次提交放到叫做 `topic/wip` 的分支上去。

```
$ git branch topic/wip     (1)
$ git reset --hard HEAD~3  (2)
$ git checkout topic/wip   (3)
```

(1) 假设已经提交了一些代码，但是此时发现这些提交还不够成熟，不能进入 `master` 分支，希望在新的 `branch` 上暂存这些改动。因此执行了 `git branch` 命令在当前的 HEAD 上建立了新的叫做 `topic/wip` 的分支。
(2) 然后回滚 `master` 分支上的最近三次提交。`HEAD~3` 指向当前 `HEAD-3` 个提交，`git reset --hard HEAD~3`，即删除最近的三个提交(删除 `HEAD`, `HEAD^`, `HEAD~2`)，将 HEAD 指向 `HEAD~3`。

(D) 永久删除最后几个提交

```
$ git commit ## 执行一些提交
$ git reset --hard HEAD~3   (1)
```

(1) 最后三个提交(即 `HEAD`, `HEAD^`和 `HEAD~2`)提交有问题，想永久删除这三个提交。

(E) 回滚 merge 和 pull 操作

```
$ git pull                         (1)
Auto-merging nitfol
CONFLICT (content): Merge conflict in nitfol
Automatic merge failed; fix conflicts and then commit the result.
$ git reset --hard                 (2)
$ git pull . topic/branch          (3)
Updating from 41223... to 13134...
Fast-forward
$ git reset --hard ORIG_HEAD       (4)
`
```

(1) 从 `origin` 拉取下来一些更新，但是产生了很多冲突，但您暂时没有这么多时间去解决这些冲突，因此决定稍候有空的时候再重新执行 `git pull` 操作。
(2) 由于 `git pull` 操作产生了冲突，因此所有拉取下来的改变尚未提交，仍然再暂存区中，这种情况下 `git reset --hard` 与 `git reset --hard HEAD` 意思相同，即都是清除索引和工作区中被搞乱的东西。
(3) 将 `topic/branch` 分支合并到当前的分支，这次没有产生冲突，并且合并后的更改自动提交。
(4) 但是此时又发现将 `topic/branch` 合并过来为时尚早，因此决定退滚合并，执行 `git reset --hard ORIG_HEAD` 回滚刚才的 `pull/merge` 操作。说明：前面讲过，执行 `git reset` 时，git 会把 `reset` 之前的 HEAD 放入`.git/ORIG_HEAD` 文件中，命令行中使用 ORIG_HEAD 引用这个提交。同样的，执行 `git pull` 和 `git merge` 操作时，git 都会把执行操作前的 HEAD 放入 `ORIG_HEAD` 中，以防回滚操作。

(F) 在污染的工作区中回滚合并或者拉取

```
$ 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工作区中的一些东西，可安全的执行 `git pull` 操作，前提是要知道将要 `git pull` 下面的内容不会覆盖工作区中的内容。
(2) `git pull` 完后，发现这次拉取下来的修改不满意，想要回滚到 `git pull` 之前的状态，从前面的介绍知道，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工作区，即丢弃本地未使用 `git 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。

(G) 中断的工作流程处理

在实际开发中经常出现这样的情形：你正在开发一个大的新功能(工作在分支：`feature` 中)，此时来了一个紧急的 bug 需要修复，但是目前在工作区中的内容还没有成型，还不足以提交，但是又必须切换的另外的分支去修改 bug。请看下面的例子 -

```
$ git checkout feature ;# you were working in "feature" branch and
$ work work work       ;# got interrupted
$ git commit -a -m "snapshot WIP"                 (1)
$ git checkout master
$ fix fix fix
$ git commit ;# commit with real log
$ git checkout feature
$ git reset --soft HEAD^ ;# go back to WIP state  (2)
$ git reset                                       (3)
```

(1) 这次属于临时提交，因此随便添加一个临时注释即可。
(2) 这次 `reset` 删除了 WIP commit，并且把工作区设置成提交 WIP 快照之前的状态。
(3) 此时，在索引中依然遗留着“*snapshot WIP*”提交时所做的未提交变化，`git reset` 将会清理索引成为尚未提交”*snapshot WIP*“时的状态便于接下来继续工作。

(H) 重置单独的一个文件

假设你已经添加了一个文件进入索引，但是而后又不打算把这个文件提交，此时可以使用 `git reset` 把这个文件从索引中去除。

```
$ git reset -- frotz.c                      (1)
$ git commit -m "Commit files in index"     (2)
$ git add frotz.c                           (3)
```

(1) 把文件 `frotz.c` 从索引中去除，
(2) 把索引中的文件提交
(3) 再次把 `frotz.c` 加入索引

(I) 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) 这次是把在 `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。