版本回退

现在，你已经学会了修改文件，然后把修改提交到 Git 版本库，现在，再练习一次，修改 readme.txt 文件如下：

```
Git is a distributed version control system.

Git is free software distributed under the GPL.
```

然后尝试提交：

```
$ git add readme.txt

$ git commit -m "append GPL"

[master 3628164] append GPL

1 file changed, 1 insertion(+), 1 deletion(-)
```

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩 RPG 游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打 Boss 之前，你会手动存盘，以便万一打 Boss 失败了，可以从最近的地方重新开始。Git 也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在 Git 中被称为 commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个 commit 恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在，我们回顾一下 readme.txt 文件一共有几个版本被提交到 Git 仓库里了：

版本 1：wrote a readme file

```
Git is a version control system.

Git is free software.
```

版本 2：add distributed

```
Git is a distributed version control system.

Git is free software.
```

版本 3：append GPL

```
Git is a distributed version control system.

Git is free software distributed under the GPL.
```

当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在 Git 中，我们用 git log 命令查看：

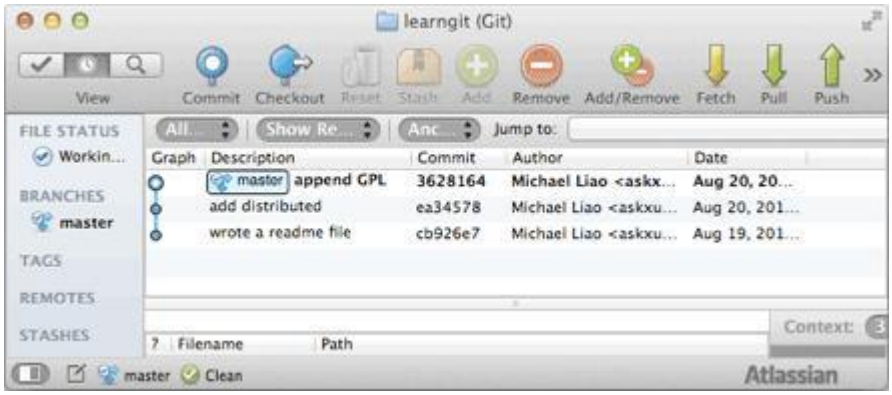
```
$ git log

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 15:11:49 2013 +0800

    append GPL



commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 14:53:12 2013 +0800

add distributed



commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Mon Aug 19 17:51:55 2013 +0800

   wrote a readme file
```

git log 命令显示从最近到最远的提交日志，我们可以看到 3 次提交，最近的一次是“append GPL”，上一次是“add distributed”，最早的一次是“wrote a readme file”。 如果嫌输出信息太多，看得眼花缭乱的，可以试试加上 --pretty=oneline 参数：

```
$ git log --pretty=oneline

3628164fb26d48395383f8f31179f24e0882e1e0 append GPL

ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed

cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file
```

需要友情提示的是，你看到的一大串类似“ 3628164...882e1e0”的是 commit id（版本号），和 SVN 不一样，Git 的 commit id 不是 1，2，3……递增的数字，而是一个 SHA1 计算出来的一个非常大的数字，用十六进制表示，而且你看到的 commit id 和我的肯定不一样，以你自己的为准。为什么 commit id 需要用这么一大串数字表示呢？因为 Git 是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用 1，2，3……作为版本号，那肯定就冲突了。

每提交一个新版本，实际上 Git 就会把它们自动串成一条时间线。如果使用可视化工具查看 Git 历史，就可以更清楚地看到提交历史的时间线：



好了，现在我们启动时光穿梭机，准备把 readme.txt 回退到上一个版本，也就是“add distributed”的那个版本，怎么做呢？

首先，Git 必须知道当前版本是哪个版本，在 Git 中，用 HEAD 表示当前版本，也就是最新的提交“ 3628164...882e1e0”（注意我的提交 ID 和你的肯定不一样），上一个版本就是 HEAD^，上上一个版本就是 HEAD^^，当然往上 100 个版本写 100 个^比较容易数不过来，所以写成 HEAD~100。

现在，我们要把当前版本“append GPL”回退到上一个版本“add distributed”，就可以使用 git reset 命令：

```
$ git reset --hard HEAD^

HEAD is now at ea34578 add distributed
```

--hard 参数有啥意义？这个后面再讲，现在你先放心使用。

看看 readme.txt 的内容是不是版本“add distributed”：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software.
```

果然。

还可以继续回退到上一个版本“wrote a readme file”，不过且慢，然我们用 *git log* 再看看现在版本库的状态：

```
$ git log

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 14:53:12 2013 +0800


    add distributed
```

```
commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Mon Aug 19 17:51:55 2013 +0800



    wrote a readme file
```

最新的那个版本“append GPL”已经看不到了！好比你从 21 世纪坐时光穿梭机来到了 19 世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个“append GPL”的 commit id 是“3628164...”，于是就可以指定回到未来的某个版本：

```
$ git reset --hard 3628164

HEAD is now at 3628164 append GPL
```

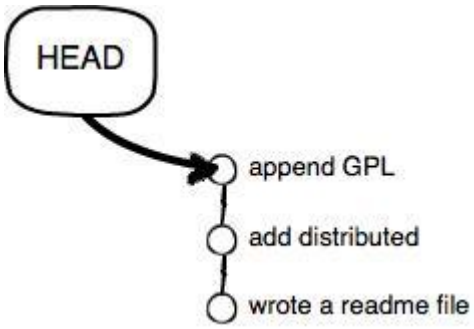
版本号没必要写全，前几位就可以了，Git 会自动去找。当然也不能只写前一两位，因为 Git 可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看 readme.txt 的内容：

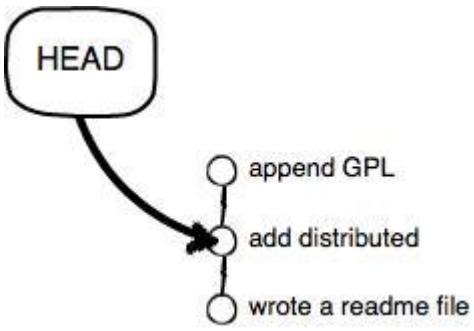
```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.
```

果然，我胡汉山又回来了。

Git 的版本回退速度非常快，因为 Git 在内部有个指向当前版本的 HEAD 指针，当你回退版本的时候，Git 仅仅是把 HEAD 从指向“append GPL”：



改为指向“add distributed”：



然后顺便把工作区的文件更新了。所以你让 HEAD 指向哪个版本号，你就把当前版本定位在哪。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的 commit id 怎么办？

在 Git 中，总是有后悔药可以吃的。当你用*$ git reset --hard HEAD^*回退到“add distributed”版本时，再想恢复到“append GPL”，就必须找到“append GPL”的 commit id。Git 提供了一个命令 *git reflog* 用来记录你的每一次命令：

```
$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file
```

终于舒了口气，第二行显示“append GPL”的 commit id 是 3628164，现在，你又可以乘坐时光机回到未来了。

小结

现在总结一下：

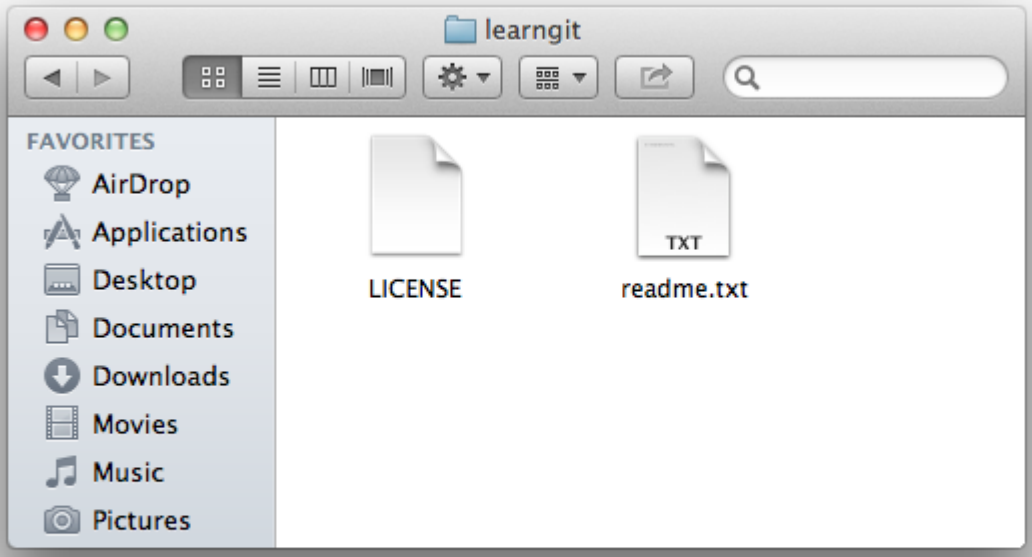
- HEAD 指向的版本就是当前版本，因此，Git 允许我们在版本的历史之间穿梭，使用命令 *git reset --hard commit_id*。
- 穿梭前，用 *git log* 可以查看提交历史，以便确定要回退到哪个版本。
- 要重返未来，用 *git reflog* 查看命令历史，以便确定要回到未来的哪个版本。

工作区和暂存区

Git 和其他版本控制系统如 SVN 的一个不同之处就是有暂存区的概念。

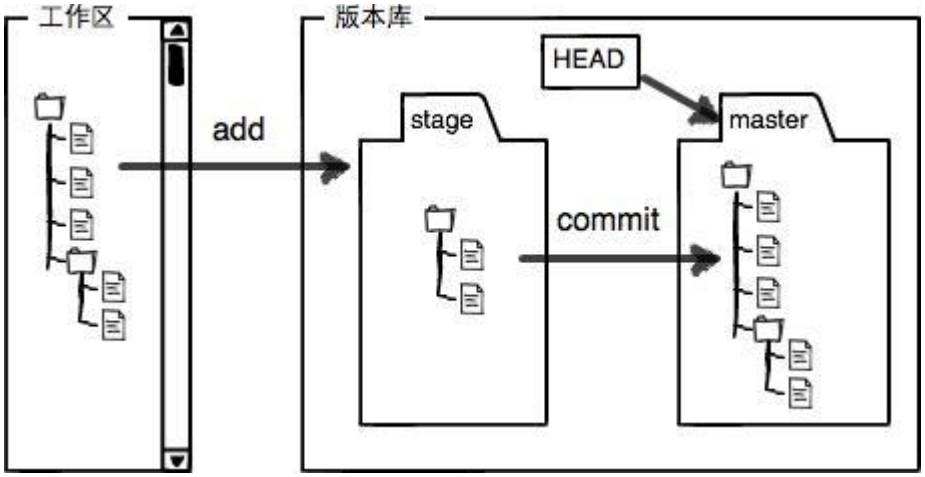
先来看名词解释。

工作区（Working Directory）：就是你在电脑里能看到的目录，比如我的 learngit 文件夹就是一个工作区：



版本库（Repository）：工作区有一个隐藏目录“.git”，这个不算工作区，而是 Git 的版本库。

Git 的版本库里存了很多东西，其中最重要的就是称为 stage（或者叫 index）的暂存区，还有 Git 为我们自动创建的第一个分支 master，以及指向 master 的一个指针叫 HEAD。



分支和 HEAD 的概念我们以后再讲。

前面讲了我们把文件往 Git 版本库里添加的时候，是分两步执行的：

第一步是用“git add”把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用“git commit”提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建 Git 版本库时，Git 自动为我们创建了唯一一个 master 分支，所以，现在，commit 就是往 master 分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

俗话说，实践出真知。现在，我们再练习一遍，先对 readme.txt 做个修改，比如加上一行内容：

```
Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.
```

然后，在工作区新增一个 LICENSE 文本文件（内容随便写）。

先用 git status 查看一下状态：

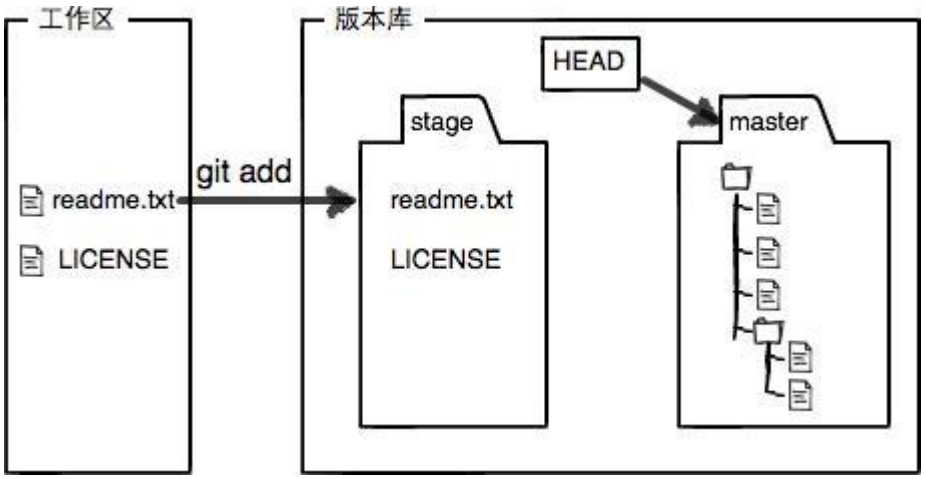
```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#       modified:   readme.txt

#

# Untracked files:

#   (use "git add <file>..." to include in what will be committed)

#

#       LICENSE

no changes added to commit (use "git add" and/or "git commit -a")
```

Git 非常清楚地告诉我们，*readme.txt* 被修改了，而 *LICENSE* 还从来没有被添加过，所以它的状态是 Untracked。

现在，使用两次命令 git add，把 readme.txt 和 LICENSE 都添加后，用 git status 再查看一下：

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       new file:   LICENSE

#       modified:   readme.txt

#
```

现在，暂存区的状态就变成这样了：



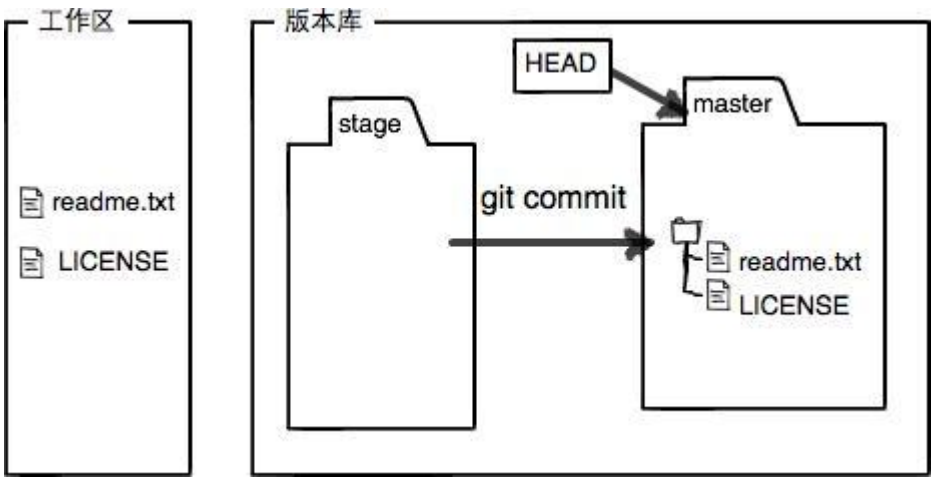
所以，git add 命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行 git commit 就可以一次性把暂存区的所有修改提交到分支。

```
$ git commit -m "understand how stage works"

[master 27c9860] understand how stage works

 2 files changed, 675 insertions(+)

 create mode 100644 LICENSE
```

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

```
$ git status

# On branch master

nothing to commit (working directory clean)
```

现在版本库变成了这样，暂存区就没有任何内容了：



小结

暂存区是 Git 非常重要的概念，弄明白了暂存区，就弄明白了 Git 的很多操作到底干了什么。

没弄明白暂存区是怎么回事的童鞋，请向上滚动页面，再看一次。

管理修改

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，为什么 Git 比其他版本控制系统设计得优秀，因为 Git 跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

为什么说 Git 管理的是修改，而不是文件呢？我们还是做实验。第一步，对 readme.txt 做一个修改，比如加一行内容：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.
```

```
Git tracks changes.
```

然后，添加：

```
$ git add readme.txt

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   readme.txt

#
```

然后，再修改 readme.txt：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.
```

提交：

```
$ git commit -m "git tracks changes"

[master d4f25b6] git tracks changes

 1 file changed, 1 insertion(+)
```

提交后，再看看状态：

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#       modified:   readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")
```

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

第一次修改 -> git add -> 第二次修改 -> git commit

你看，我们前面讲了，Git 管理的是修改，当你用“git add”命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，“git commit”只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用“git diff HEAD -- readme.txt”命令可以查看工作区和版本库里面最新版本的区别：

```
$ git diff HEAD -- readme.txt

diff --git a/readme.txt b/readme.txt

index 76d770f..a9c5755 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,4 +1,4 @@

 Git is a distributed version control system.

 Git is free software distributed under the GPL.

 Git has a mutable index called stage.

-Git tracks changes.

+Git tracks changes of files.
```

可见，第二次修改确实没有被提交。

那怎么提交第二次修改呢？你可以继续 add 再 commit，也可以别着急提交第一次修改，先 add 第二次修改，再 commit，就相当于把两次修改合并后一块提交了：

第一次修改 -> add -> 第二次修改 -> add -> commit

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了 Git 是如何跟踪修改的，每次修改，如果不 add 到暂存区，那就不会加入到 commit 中。

撤销修改

自然，你是不会犯错的。不过现在是凌晨两点，你正在赶一份工作报告，你在 readme.txt 中添加了一行：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.
```

在你准备提交前，一杯咖啡起了作用，你猛然发现了“stupid boss”可能会让你丢掉这个月的奖金！

既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用 *git status* 查看一下：

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#       modified:   readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")
```

你可以发现，Git 会告诉你，*git checkout -- file* 可以丢弃工作区的修改：

```
$ git checkout -- readme.txt
```

命令 *git checkout -- readme.txt* 意思就是，把 readme.txt 文件在工作区的修改全部撤销，这里有两种情况：

一种是 readme.txt 自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是 readme.txt 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次 git commit 或 git add 时的状态。

现在，看看 readme.txt 的文件内容：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.
```

文件内容果然复原了。

git checkout -- file 命令中的“--”很重要，没有“--”，就变成了“创建一个新分支”的命令，我们在后面的分支管理中会再次遇到 *git checkout* 命令。

现在假定是凌晨 3 点，你不但写了一些胡话，还 *git add* 到暂存区了：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.



$ git add readme.txt
```

庆幸的是，在 commit 之前，你发现了这个问题。用 *git status* 查看一下，修改只是添加到了暂存区，还没有提交：

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   readme.txt

#
```

Git 同样告诉我们，用命令 *git reset HEAD file* 可以把暂存区的修改撤销掉（unstage），重新放回工作区：

```
$ git reset HEAD readme.txt
```

```
Unstaged changes after reset:

M       readme.txt
```

git reset 命令既可以回退版本，也可以把工作区的某些文件替换为版本库中的文件。当我们用 HEAD 时，表示最新的版本。

再用 *git status* 查看一下，现在暂存区是干净的，工作区有修改：

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#       modified:   readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")
```

还记得如何丢弃工作区的修改吗？

```
$ git checkout -- readme.txt



$ git status

# On branch master

nothing to commit (working directory clean)
```

整个世界终于清静了！

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？还记得版本回退一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得 Git 是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把“stupid boss”提交推送到远程版本库，你就真的惨了……

小结

又到了小结时间。

场景 1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令 *git checkout -- file*。

场景 2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令 *git reset HEAD file*，就回到了场景 1，第二步按场景 1 操作。

场景 3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。