如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

创建版本库

```
$ git init
```

添加文件--将文件添加到缓存区（stage）

```
$ git add readme.txt
```

提交文件--将缓存区的修改提交到版本库

```
$ git commit -m "wrote a readme file"
```

简单解释一下`git commit`命令，`-m`后面输入的是本次提交的说明

查看当前版本库的状态

```
$ git status
```

查看某文件的修改

```
$ git diff  readme.txt

git diff    #是工作区(work dict)和暂存区(stage)的比较

git diff  --cached   #是暂存区(stage)和分支(master)的比较
```

stage或cache虽说是暂存区，缓冲区，但commit并不是像想像那样把这个区清空，估计只是打个同步的标志，内容还在，就能理解了。
add 是 把工作区的更新到暂存区，commit是把缓冲区更新到仓库。所以经过add, commit,修改再add,再修改，就会出现工作区、缓冲区和仓库三者都不同。
可以下面的比较了： git diff 是工作区和 中间区比较，git diff --cached是中间区和仓库比较。

查看记录（回退版本会消失）

```
$ git log

$ git log --pretty=oneline
```

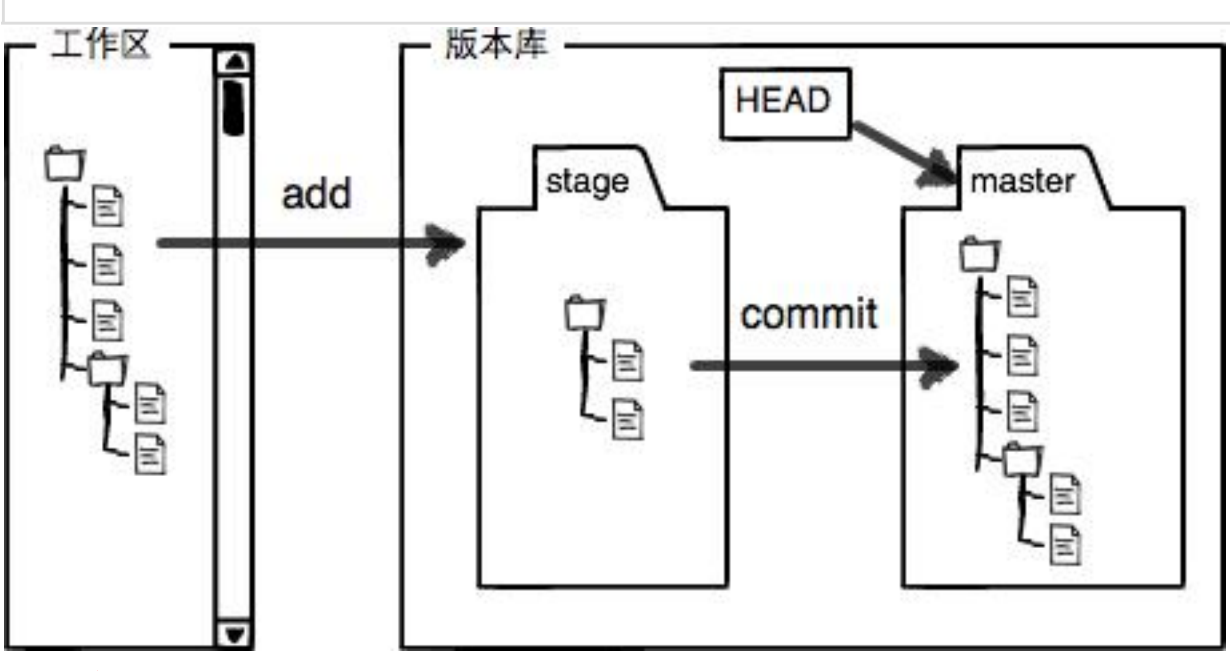
查看所有记录（包括回退记录）

```
$ git reflog
```

版本回退

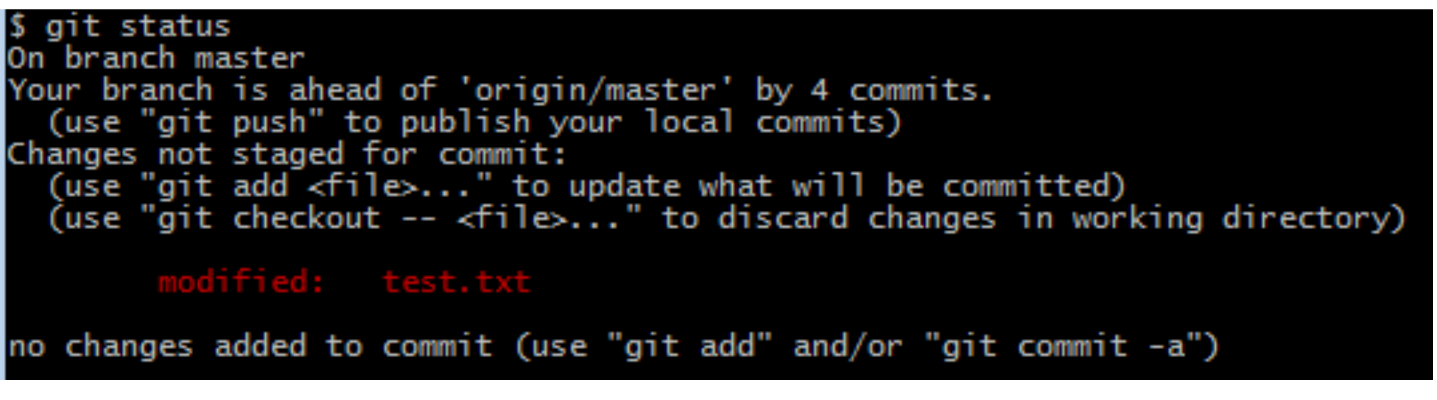
首先，Git必须知道当前版本是哪个版本，在Git中，用`HEAD`表示当前版本，也就是最新的提交`3628164...882e1e0`（注意我的提交ID和你的肯定不一样），上一个版本就是`HEAD^`，上上一个版本就是`HEAD^^`，当然往上100个版本写100个`^`比较容易数不过来，所以写成`HEAD~100`。

```
$ git reset --hard HEAD^

$ git reset --hard 3628164
```



撤销修改

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。
注意： git checkout -- file 命令中的 -- 很重要，没有 --，就变成了“切换到另一个分支”的命令
命令git reset HEAD file可以把暂存区的修改撤销掉（unstage）
注：
使用 git reset HEAD file 是可以把暂存区的修改撤销掉的，但不会覆盖工作区，比如：
添加一个文件，add->commit->modify->add->modify->reset
此时git status的提示是：暂存区内的修改撤销（no changes added to commit），工作区的内容依然是 最后一次modify之后的数据。

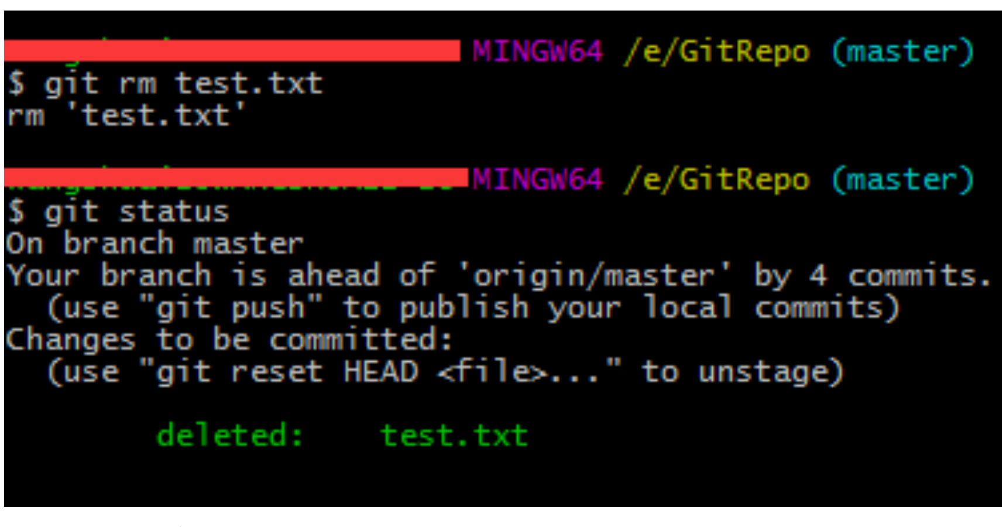


```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

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

此时如果使用参数 --hard （就是版本回退）则可以把暂存区的修改撤销掉（unstage），并重新放回工作区

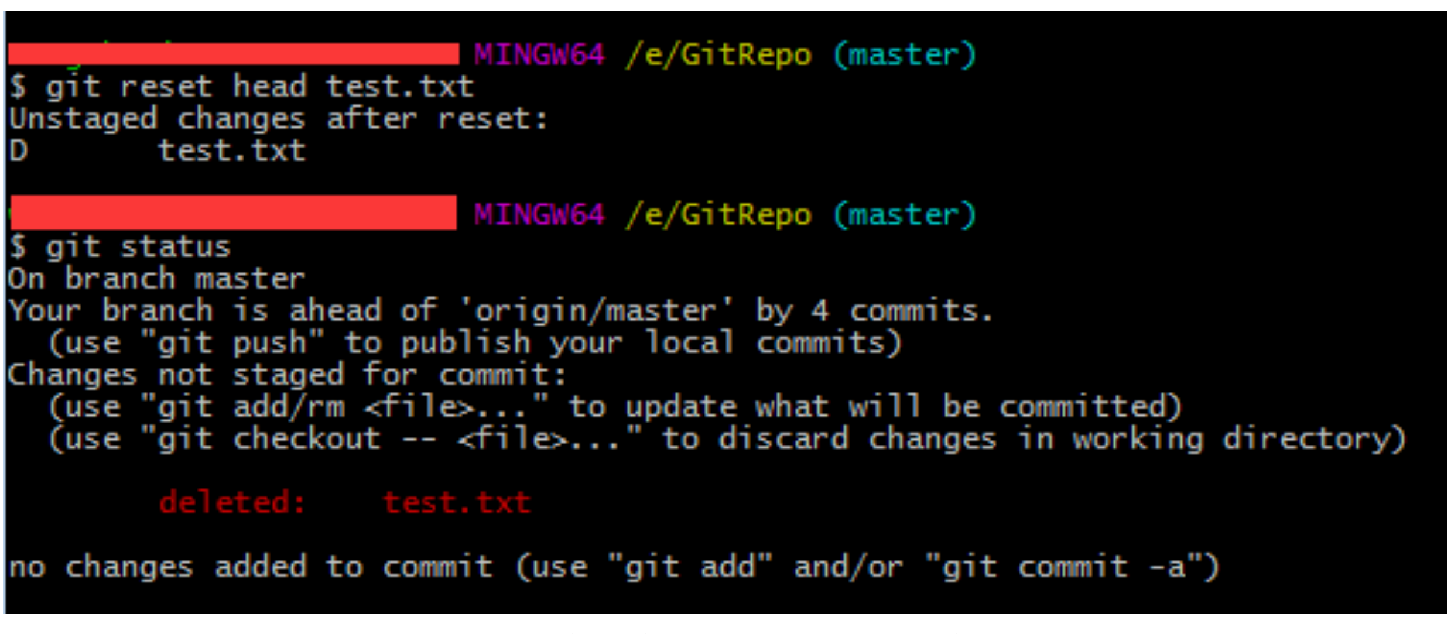
删除文件

git rm file命令会直接删除文件并提交到stage
相当于操作系统的rm命令之后，再 git add file



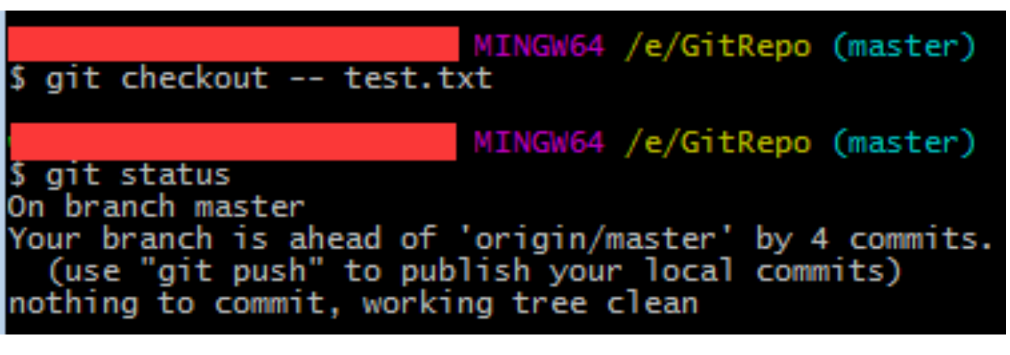
```
[redacted] MINGW64 /e/GitRepo (master)
$ git rm test.txt
rm 'test.txt'

[redacted] MINGW64 /e/GitRepo (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    test.txt
```

如果需要恢复，则需要 git reset head file 相当于清空stage（可以理解为将git add file命令回退，此时工作区是删除状态）



```
[redacted] MINGW64 /e/GitRepo (master)
$ git reset head test.txt
Unstaged changes after reset:
D       test.txt

[redacted] MINGW64 /e/GitRepo (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    test.txt

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

然后需要 git checkout file 从版本库恢复文件（覆盖本地文件）



```
[redacted] MINGW64 /e/GitRepo (master)
$ git checkout -- test.txt

[redacted] MINGW64 /e/GitRepo (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

创建ssh key

```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

本地关联/取消远程库

```
$ git remote add origin  git@github.com:wang-shuai/hello-world.git
```

```
$ git remote remove origin
```

克隆远程库到本地

```
$ git clone git@github.com:wang-shuai/hello-world.git
```

推送本地本版库（当前分支master）内容到远程库（origin） （add->commit 之后的内容）

```
$ git push -u origin master
```

创建分支切换分支

```
$ git checkout -b dev
```

相当于下边两句命令：

创建分支：

```
$ git branch dev
```

检出分支（切换分支）

```
$ git checkout dev
```

查看分支

```
$ git branch
```

`git branch`命令会列出所有分支，当前分支前面会标一个`*`号。
合并分支 （后边dev是要被合并到当前分支的）

```
$ git merge dev
```

`git merge`命令用于合并指定分支到当前分支。

删除分支

```
$ git branch -d dev
```

若分支未合并，则提示
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
大写 -D 强制删除未合并的分支

查看修改记录（图形格式）

```
$ git log --graph --pretty=oneline --abbrev-commit
```

abbrev-commit 缩略记录号
pretty=oneline 一行展示记录
graph 图形格式（分支路线）

fast forward模式

```
$ git merge --no-ff  -m "merge with no-ff" dev
```

合并分支时，加上`--no-ff`参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而`fast forward`合并就看不出来曾经做过合并。

存储工作现场

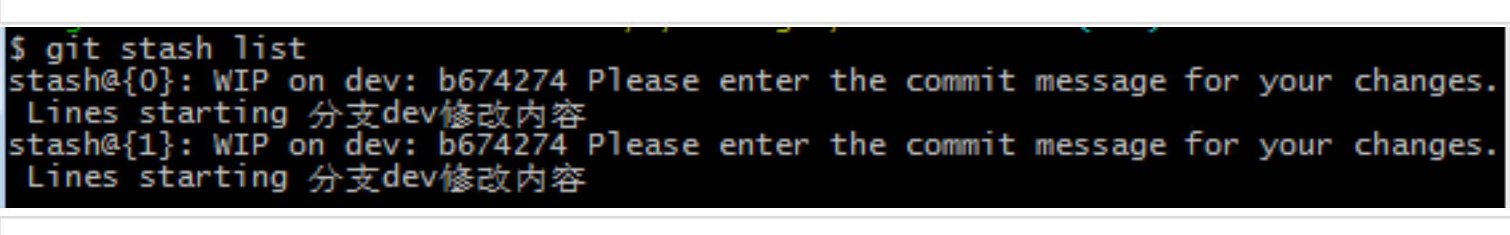
```
$ git stash
```

当在分支工作时，尚未提交（很可能是开发一半不能提交），又急需要去修复其他分支的bug时，可使用该命令，存储工作现场。
如果在分支dev1上工作，没有add和commit，直接切换到master上，会发现改动的内容直接体现在master的文件上，所以需要stash一下工作分支，然后新建bug分支开始新的工作。采用stash多个分支可以切换分支工作而不影响其他分支工作。

恢复工作现场

当我们修改为其他分支的bug后，checkout回到之前stash的分支，通过命令 `git stash list` 查看现场列表（我分别在dev分支修改了两次并进行了两次stash）

```
$ git stash list
```



```
$ git stash list
stash@{0}: WIP on dev: b674274 Please enter the commit message for your changes.
 Lines starting 分支dev修改内容
stash@{1}: WIP on dev: b674274 Please enter the commit message for your changes.
 Lines starting 分支dev修改内容
```

```
$ git stash pop
```

该命令会从stash列表里恢复一个最近现场A（按先进后出的顺序，从pop命令也可以看出来），并且同时会销毁列表里的现场A。

```
$ git stash apply stash@{0}
```

该命令也会从stash列表里恢复一个特定现场，但并不会销毁列表里的现场

大体使用过程：

概览：**master合并merge解决好的bug后，不要先把dev解印，先合并master，获取里面的bug方案后，在解印。解印时会有提示冲突，需手动改一次文件。**
1：在 dev 下正常开发中，说有1个bug要解决，首先我需要把dev分支封存stash
2：在master下新建一个issue-101分支，解决bug，成功后
3：在master下合并issue-101
4：在 dev 下合并master， 这样才同步了里面的bug解决方案
5：解开dev封印stash pop，系统自动合并 & 提示有冲突，因为封存前dev写了东西，此时去文件里手动改冲突
6：继续开发dev，最后add，commit
7：在master下合并最后完成的dev

查看远程库信息

```
$ git remote
```

用`git remote -v`显示更详细的信息

github官方库-->fork后成为自己的远程库<--->本地库
修复完的内容只能push到自己fork的远程库，若想共享代码到官方库，需要提交pull request

别名

```
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.br branch
```

git commit --amend 可以修改提交的日志文案