Gitlab入门使用指南V0.1

一，Git基础操作

1.1 git用户属性设置

- 在开始我们需要声明下自己是谁，自己的邮箱是什么，这样方便我们后续的查询和管理

```
git config --global user.name "USERNAME"
git config --global user.email "USER@MAIL"
```

- 示例：

```
[root@localhost ~]# git config --global
user.name "mark"
[root@localhost ~]# git config --global
user.email "usertzc@163.com"
[root@localhost ~]# git config --list
user.name=mark
user.email=usertzc@163.com
```

1.2 颜色高亮

在后续使用中可能会用到颜色高亮，设置如下：

```
[root@localhost ~]# git config --global color.ui true
[root@localhost ~]# git config --list
user.name=mark
user.email=usertzc@163.com
color.ui=true
[root@localhost ~]#
```

二，Git仓库基本操作说明

2.1 初始化本地仓库

在初始化之前，我们仍然需要创建一个文件夹。

```
[root@localhost ~]# mkdir git0819 &&  cd git0819/
[root@localhost git0819]# git init
```

初始化空的 Git 版本库于 /root/git0819/.git/
当初始化成功后会出现几个文件，我们不建议你修改这几个文件内容
``
[root@localhost git0819]# ls -a
``
. .. .git

2.2 git文件提交

我们追加一个字符串到readme.txt中

```
[root@localhost git0819]# echo "1 helo git" readme.txt
[root@localhost git0819]# cat readme.txt
1 helo git
```

2.2.1 git add

追加完成后使用git add来进行提交到暂存区

`[root@localhost git0819]# git add readme.txt`

2.2.2 git status

请注意:

在每次使用git add提交到暂存区时，可以使用git status来查看，提交的分之，和第几次提交，提交的文件信息，如下所示：

```
[root@localhost git0819]# git status
# 位于分支 master
#
# 初始提交
#
# 要提交的变更：
#   （使用 "git rm --cached <file>..." 撤出暂存区）
#
#       新文件：     readme.txt
#
[root@localhost git0819]#
```

2.2.3 git commit

当git add到暂存区后我们使用git commit -m "备注"来提交

```
[root@localhost git0819]# git commit -m "the file commit"
[master（根提交） 5ea98f2] the file commit
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
[root@localhost git0819]#
```

当提交完成后，在使用git status查看工作目录则为干净的状态，由此可见，我们可以用git status来查看我们是否提交等信息

```
    [root@localhost git0819]# git status
    # 位于分支 master
    无文件要提交，干净的工作区
```

2.2.3.1 修改最后一次提交

有时候我们提交完了才发现漏掉了几个文件没有加，或者提交信息写错了。想要撤消刚才的提交操作，可以使用 `--amend` 选项重新提交：

```
$ git commit --amend
```

此命令将使用当前的暂存区域快照提交。如果刚才提交完没有作任何改动，直接运行此命令的话，相当于有机会重新编辑提交说明，但将要提交的文件快照和之前的一样。

启动文本编辑器后，会看到上次提交时的说明，编辑它确认没问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交。

如果刚才提交时忘了暂存某些修改，可以先补上暂存操作，然后再运行`--amend`提交：

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

上面的三条命令最终只是产生一个提交，第二个提交命令修正了第一个的提交内容。

2.2.3.2 取消已经暂存的文件

如何取消工作目录中已修改的文件。git status查看文件状态的时候就提示了该如何撤消

来看下面的例子，有两个修改过的文件，我们想要分开提交，但不小心用`git add .`全加到了暂存区域。该如何撤消暂存其中的一个文件呢？其实，`git status`的命令输出已经告诉了我们该怎么做：

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        modified:   benchmarks.rb
```

就在 “Changes to be committed” 下面，括号中有提示，可以使用`git reset HEAD ...`的方式取消暂存。好吧，我们来试试取消暂存 benchmarks.rb 文件：

```
$ git reset HEAD benchmarks.rb
Unstaged changes after reset:
M       benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt

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:   benchmarks.rb
```

现在 benchmarks.rb 文件又回到了之前已修改未暂存的状态。

2.3.3.3 取消对文件的修改

如果觉得刚才对 某个文件的修改完全没有必要，该如何取消修改，回到之前的状态（也就是修改之前的版本）呢？`git status` 同样提示了具体的撤消方法，接着上面的例子，现在未暂存区域看起来像这样：

```
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:   benchmarks.rb
```

如下：

```
$ git checkout -- benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
```

可以看到，该文件已经恢复到修改前的版本。你可能已经意识到了，这条命令有些危险，所有对文件的修改都没有了，因为我们刚刚把之前版本的文件复制过来重写了此文件。所以在用这条命令前，请务必确定真的不再需要保留刚才的修改。如果只是想回退版本，同时保留刚才的修改以便将来继续工作，可以用分支来处理，应该会更好些。

2.2.3.4 移除文件

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。

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）看到：

```
$ rm grit.gemspec
$ git status
On branch master
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:    grit.gemspec

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

然后再运行 `git rm` 记录此次移除文件的操作：

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    grit.gemspec
```

最后提交的时候，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），以防误删除文件后丢失修改的内容。

另外一种情况是，我们想把文件从 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，仅是从跟踪清单中删除。比如一些大型日志文件或者一堆 `.a` 编译文件，不小心纳入仓库后，要移除跟踪但不删除文件，以便稍后在 `.gitignore` 文件中补上，用 `--cached` 选项即可：

```
$ git rm --cached readme.txt
```

后面可以列出文件或者目录的名字，也可以使用 glob 模式。比方说：

```
$ git rm log/\*.log
```

注意到星号 `*` 之前的反斜杠 `\`，因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开（译注：实际上不加反斜杠也可以运行，只不过按照 shell 扩展的话，仅仅删除指定目录下的文件而不会递归匹配。上面的例子本来就指定了目录，所以效果等同，但下面的例子就会用递归方式匹配，所以必须加反斜杠。）。此命令删除所有 `log/` 目录下扩展名为 `.log` 的文件。类似的比如：

```
$ git rm \*~
```

会递归删除当前目录及其子目录中所有 `~` 结尾的文件。

2.2.3.5 移动文件

不像其他的 VCS 系统，Git 并不跟踪文件移动操作。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。不过 Git 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 Git 的 `mv` 命令时一定会困惑不已。要在 Git 中对文件改名，可以这么做：

```
$ git mv file_from file_to
```

它会恰如预期般正常工作。实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说明：

```
$ git mv README.txt README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README.txt -> README
```

其实，运行 `git mv` 就相当于运行了下面三条命令：

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式都一样。当然，直接用 `git mv` 轻便得多，不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

2.2.4 git暂存区和工作区

当我们在使用时，可以add多次到暂存区，而后一次commit到工作区，如果没有暂存区则无法到工作区

```
[root@localhost git0819]# echo 'is me mark' >> readme.txt
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# echo 'is me mark 1' >> readme.txt
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# git commit -m "is me mark"
[master 10325bb] is me mark
 1 file changed, 1 insertion(+)
[root@localhost git0819]#
```

2.3 查看历史提交信息git log

```
[root@localhost git0819]# git log
  commit 87ad99a080362adb10c07e80e0f75428d96120ac
  Author: mark <usertzc@163.com>
  Date:   Fri Aug 19 22:31:20 2016 +0800

      2th commit

  commit 5ea98f29e0d591ec2583eacf3af6e4df802ff356
  Author: mark <usertzc@163.com>
  Date:   Fri Aug 19 22:24:47 2016 +0800

      the file commit
  [root@localhost git0819]#
```

如上所示，当你输入git log后，你会看到commit的效验和，和提交人已经实际，和备注信息，提交的次数等

其他选项：

-p：展开显示每次提交的内容差异

`git log -p -2`

-2：-2仅显示最近的两次更新

2.4 文件比对git diff

当我们修改了文件后可以使用git diff来进行文件比对

示例如下：

```
[root@localhost git0819]# echo 2 helo github >> readme.txt
[root@localhost git0819]# git diff readme.txt
diff --git a/readme.txt b/readme.txt
index 74da0d9..eab080a 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1 +1,2 @@
 1 helo git
+2 helo github
[root@localhost git0819]#
```

其他选项：

当然，你也可以控制对比的条目：

`git log -U1 --word-diff`

-U1：如果你有三行，-U1选项则会显示1行

另外，git log还提供了许多摘要选项，比如 --stat，仅显示摘要行数统计：

`git log --stat`

`--pretty` 选项，另外还有`short`，`full` 和 `fuller`可以指定使用完全不同于默认格式的方式展示提交历史

当我们修改文件后，也就是文件发送改变，这时如果要让修改后的文件生效，仍然需要再次提交

```
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# git commit -m "readme.txt 2"

[master 3fc47d0] readme.txt 2
 1 file changed, 1 insertion(+)
[root@localhost git0819]# git log
commit 3fc47d0453dedf1923a5145562451d5157f75050
Author: mark <usertzc@163.com>
Date:   Fri Aug 19 22:35:45 2016 +0800

    readme.txt 2

commit 87ad99a080362adb10c07e80e0f75428d96120ac
Author: mark <usertzc@163.com>
Date:   Fri Aug 19 22:31:20 2016 +0800

    2th commit

commit 5ea98f29e0d591ec2583eacf3af6e4df802ff356
Author: mark <usertzc@163.com>
Date:   Fri Aug 19 22:24:47 2016 +0800

    the file commit
[root@localhost git0819]#
```

三，版本回退

3.1 回退到上一个版本

git reset --hard使用说明：这里不得不说的是在git中的另外一个命令`git reset --hard`,使用此命令我们能够回到任意一个版本
git reset --hard HEAD，其中HEAD表示当前版本，HEAD^表示上一个版本，`HEAD^^`表示上上一个版本，`HEAD^^^`表示上上一个版本

回退实例：

在开始前我将一个文件删掉，在创建同样的文件，输入不同的内容进行回退：

```
[root@localhost git0820]# cat readme.txt
date +%F-%T
[root@localhost git0820]# rm -rf readme.txt
[root@localhost git0820]# ls
dev.txt
```

分别提交三次不同的内容到readme.txt中，commit 名称分别是2 helo git 和3 helo git和4 helo git

```
[root@localhost git0819]# echo 2 helo git >> readme.txt
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# git commit -m "2 helo git"
[master a006a08] 2 helo git
 1 file changed, 1 insertion(+)
[root@localhost git0819]# echo 3 helo git >> readme.txt
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# git commit -m "3 helo git"
[master cca52f3] 3 helo git
 1 file changed, 1 insertion(+)
[root@localhost git0819]# echo 4 helo git >> readme.txt
[root@localhost git0819]# git add readme.txt
[root@localhost git0819]# git commit -m "4 helo git"
[master e9f932b] 4 helo git
 1 file changed, 1 insertion(+)
[root@localhost git0819]#
```

我们先回退到第一个版本

```
[root@localhost git0819]# git reset --hard HEAD^
HEAD 现在位于 cca52f3 3 helo git
```

回退上两个版本

```
[root@localhost git0819]# git reset --hard HEAD^^
HEAD 现在位于 5ea98f2 the file commit
[root@localhost git0819]# cat readme.txt
1 helo git
[root@localhost git0819]#
```

我提交三次，回退三次后就到了删除前的版本，回退正常。内容如下：

```
[root@localhost git0820]# cat readme.txt
date +%F-%T
```

请注意： 你会发现，当你回退到之前后，在此期间提交三次log是不会存在的，这相当于做了一次时空逆转，git log是看不到回退期间提交的日志

3.2 回退到指定版本

git reflog可以查看历史版本，我们可以使用git reset --herad 版本号即可回退到指定版本

示例如下：

```
[root@localhost git0819]# git reflog
5ea98f2 HEAD@{0}: reset: moving to HEAD^^
cca52f3 HEAD@{1}: reset: moving to HEAD^
e9f932b HEAD@{2}: commit: 4 helo git
cca52f3 HEAD@{3}: commit: 3 helo git
a006a08 HEAD@{4}: commit: 2 helo git
5ea98f2 HEAD@{5}: reset: moving to HEAD^^
7cf1ea2 HEAD@{6}: commit: git 4
87ad99a HEAD@{7}: reset: moving to HEAD^
3fc47d0 HEAD@{8}: commit: readme.txt 2
87ad99a HEAD@{9}: commit: 2th commit
5ea98f2 HEAD@{10}: commit (initial): the file commit
```

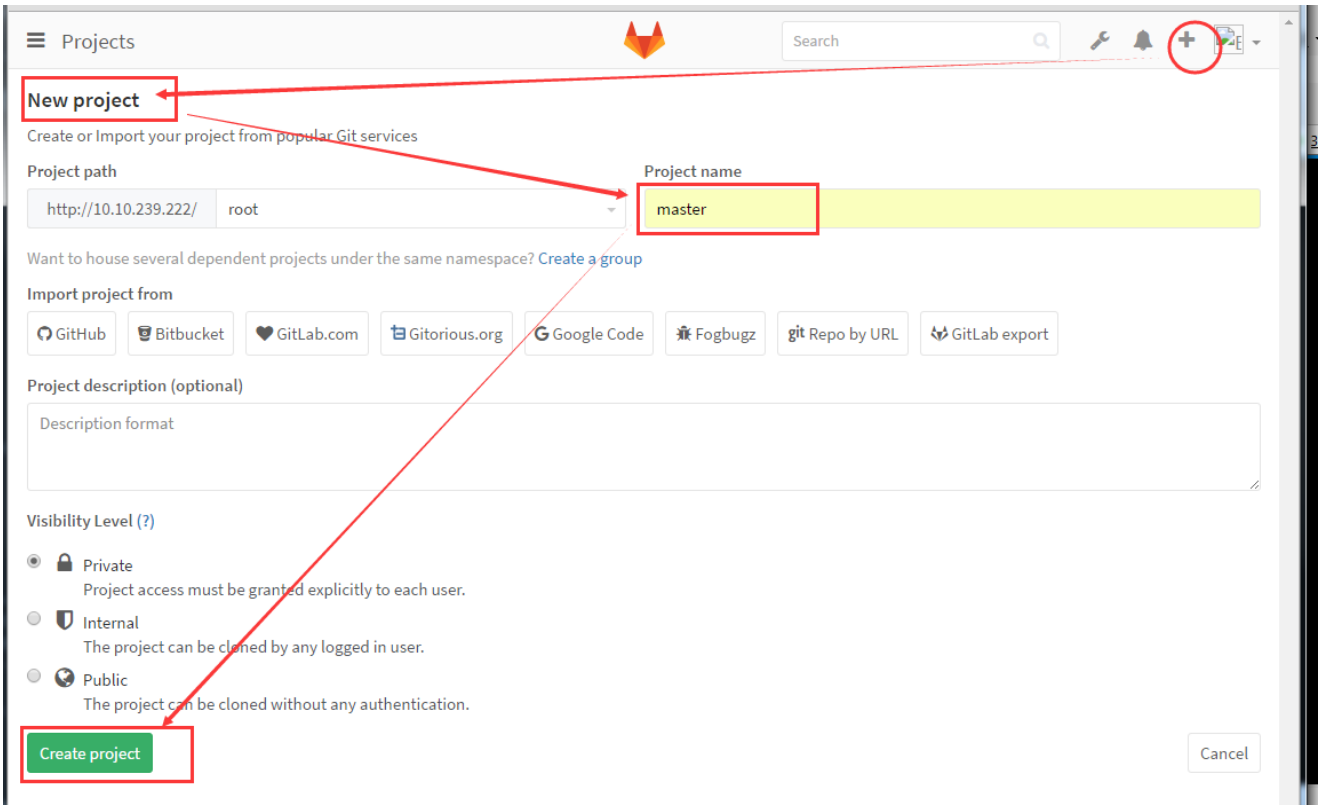
我们指定一个版本号进行回退：

```
[root@localhost git0819]# git reset --hard a006a08
HEAD 现在位于 a006a08 2 helo git
[root@localhost git0819]# cat readme.txt
1 helo git
2 helo git
[root@localhost git0819]#
```

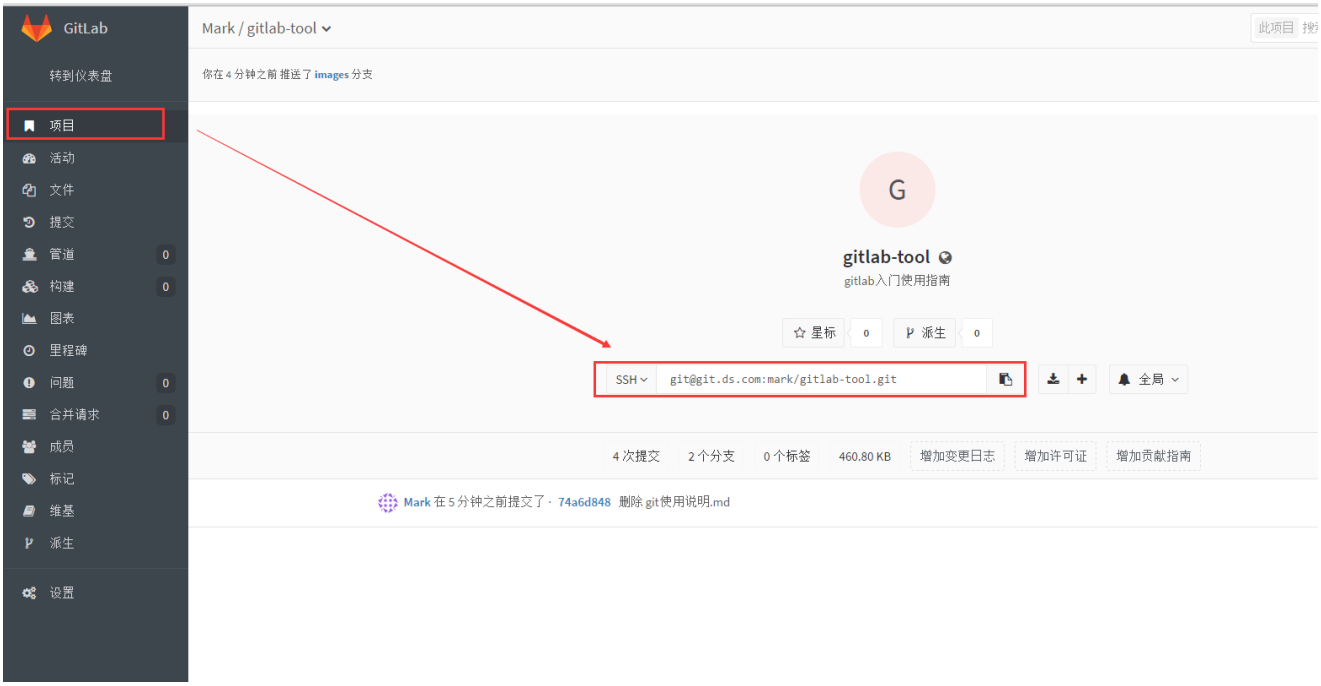
四，关联到gitlab

4.1 git lab New project

第一步：我们打开gitlab，成功登陆后 选择右上角的+号 New project



第二步：创建完成后会在调转的页面复制ssh中一行：如下所示



4.2 ssh key

在需要链接gitlab的主机上生成key，一路回车即可

```
[root@localhost 0822]# ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
/root/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
72:8a:c4:87:cb:d3:f6:35:5c:95:24:a6:7e:24:49:fc root@150
The key's randomart image is:
+--[ RSA 2048]----+
|         .. o .  |
|         ..+ o . |
|          +.. o  |
|   . .    . oE.  |
|    + o S . o    |
|   o = + . o     |
|    = +   +      |
|     o . . .     |
|        .        |
+-----------------+
[root@localhost 0822]#
```

将id_rsa.pub内容复制到剪切板

```
[root@localhost 0822]# cat /USERNAME/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC5kmoitUCzznv00m2GgK66ksHpTQHYcQR/dOhKgRI9ut8YAHJC6LjfddWad+2/az6k1E
6ecTbTvJNJnWQl3qv16E+MR6WSMvHWXyQT5t5w0mqBKDvITgXoZbmQW6QzVBYmy295T/IIbVxefv/rMmjHyj330DSZdad/Mdcw
vtfTZBvBAhBUXp28ifbEIMj00wYL7qspobniVDPjvcjkwYbhUqn0CjZdkXgcdSdyhEdcI1aLy3Ps/s3oyxXz/7jhrpI4s01Gx3
9caXX6e3Jn2oy8Ds8qcyhCTgFnt9QWEL6SOYX42FqHhOUxwSN3+/jOcj+c6zuZUanRch85QTplaiY1 root@150
[root@localhost 0822]#
```

现在回到gitlab web界面，在profile settings中选择ssh keys添加即可，如下图所示：

4.3 将本地文件提交到gitlab

创建目录，初始化，echo `date +%T-%F` > mark.txt文件中后提交

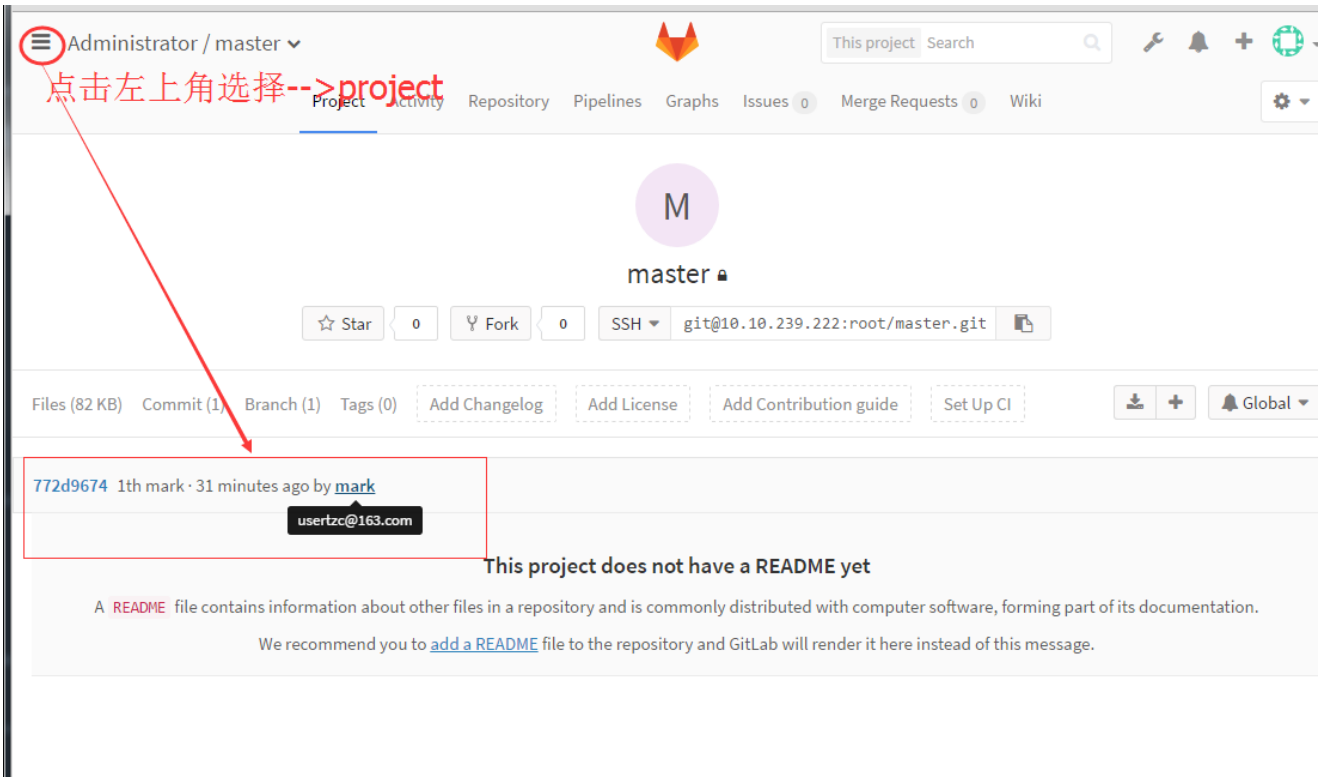
```
[root@localhost ~]# mkdir 0822
[root@localhost ~]# cd 0822/
[root@localhost 0822]# ls
[root@localhost 0822]# git config --global user.name "mark"
[root@localhost 0822]# git config --global user.email "usertzc@163.com"
[root@localhost 0822]# git init
初始化空的 Git 版本库于 /root/0822/.git/
[root@localhost 0822]# echo `date +%T-%F` > mark.txt
[root@localhost 0822]# git add mark.txt
[root@localhost 0822]# git commit -m "1th mark"
[master（根提交） 772d967] 1th mark
 1 file changed, 1 insertion(+)
 create mode 100644 mark.txt
```

git remote add origin git@10.10.239.222:root/master.git这段操作可以在4.1的第二步中看到

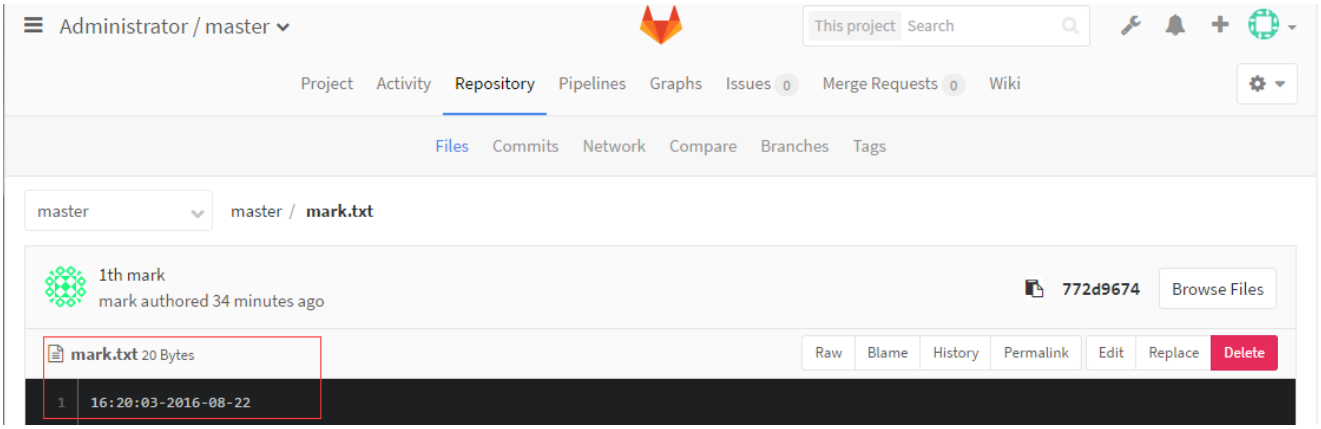
```
[root@localhost 0822]# git remote add origin git@10.10.239.222:root/master.git
[root@localhost 0822]# cat .git/config
[core]

        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = git@10.10.239.222:root/master.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[root@localhost 0822]# git add mark.txt
[root@localhost 0822]# git commit -m "1th mark"
# 位于分支 master
无文件要提交，干净的工作区
[root@localhost 0822]# git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@10.10.239.222:root/master.git
* [new branch]      master -> master
   分支 master 设置为跟踪来自 origin 的远程分支 master
```

当push完成后，我们打开gitlab,在左上角点击后，选择project，你会发现有最新的提交



也可看到内容



4.4 远程仓库的使用

4.4.1 查看当前的远程库

要查看当前配置有哪些远程仓库，可以用 `git remote` 命令，它会列出每个远程库的简短名字。在克隆完某个项目后，至少可以看到一个名为 origin 的远程库，Git 默认使用这个名字来标识你所克隆的原始仓库：

```
$ git clone git://github.com/schacon/ticgit.git
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

也可以加上 `-v` 选项（译注：此为 `--verbose` 的简写，取首字母），显示对应的克隆地址：

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

如果有多个远程仓库，此命令将全部列出。比如在我的 Grit 项目中，可以看到：

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

这样一来，我就可以非常轻松地从这些用户的仓库中，拉取他们的提交到本地。请注意，上面列出的地址只有 origin 用的是 SSH URL 链接，所以也只有这个仓库我能推送数据上去

4.4.2 添加远程仓库

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用，运行 `git remote add [shortname] [url]`：

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb  git://github.com/paulboone/ticgit.git
```

现在可以用字符串 `pb` 指代对应的仓库地址了。比如说，要抓取所有 Paul 有的，但本地仓库没有的信息，可以运行 `git fetch pb`：

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

现在，Paul 的主干分支（master）已经完全可以在本地访问了，对应的名字是 `pb/master`，你可以将它合并到自己的某个分支，或者切换到这个分支，看看有些什么更新。

4.4.3 从远程仓库抓取数据

正如之前所看到的，可以用下面的命令从远程仓库抓取数据到本地：

```
$ git fetch [remote-name]
```

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。

如果是克隆了一个仓库，此命令会自动将远程仓库归于 origin 名下。所以，`git fetch origin` 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 fetch 以来别人提交的更新）。有一点很重要，需要记住，fetch 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支，只有当你确实准备好了，才能手工合并。

如果设置了某个分支用于跟踪某个远端仓库的分支（可以使用 `git pull` 命令自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下 `git clone` 命令本质上就是自动创建了本地的 master 分支用于跟踪远程仓库中的 master 分支（假设远程仓库确实有 master 分支）。所以一般我们运行 `git pull`，目的都是要从原始克隆的远端仓库中抓取数据后，合并到工作目录中的当前分支。

4.4.4 推送到远程仓库

```
$ git push origin master
```

只有在所克隆的服务器上有写权限，或者同一时刻没有其他人在推数据，这条命令才会如期完成任务。如果在你推数据前，已经有其他人推送了若干更新，那你的推送操作就会被驳回。你必须先把他们的更新抓取到本地，合并到自己的项目中，然后才可以再次推送。

推送远程仓库在4.3中已经提到

4.4.5 查看远程仓库信息

我们可以通过命令 `git remote show [remote-name]` 查看某个远程仓库的详细信息，比如要看所克隆的 `origin` 仓库，可以运行：

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

除了对应的克隆地址外，它还给出了许多额外的信息。它友善地告诉你如果是在 master 分支，就可以用`git pull` 命令抓取数据合并到本地。另外还列出了所有处于跟踪状态中的远端分支。

上面的例子非常简单，而随着使用 Git 的深入，`git remote show` 给出的信息可能会像这样：

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

它告诉我们，运行 `git push` 时缺省推送的分支是什么（译注：最后两行）。它还显示了有哪些远端分支还没有同步到本地（译注：第六行的 `caching` 分支），哪些已同步到本地的远端分支在远端服务器上已被删除（译注：`Stale tracking branches` 下面的两个分支），以及运行 `git pull` 时将自动合并哪些分支（译注：前四行中列出的 `issues` 和 `master` 分支）。

4.4.6 远程仓库的删除和重命名

可以用 `git remote rename` 命令修改某个远程仓库在本地的简称，比如想把 `pb` 改成`paul`，可以这么运行：

```
$ git remote rename pb paul
$ git remote
origin
paul
```

注意，对远程仓库的重命名，也会使对应的分支名称发生变化，原来的 `pb/master` 分支现在成了`paul/master`。通常我们在web中进行修改

碰到远端仓库服务器迁移，或者原来的克隆镜像不再使用，又或者某个参与者不再贡献代码，那么需要移除对应的远端仓库，可以运行 `git remote rm` 命令：

```
$ git remote rm paul
$ git remote
origin
```

五，分之使用

5.1 git branch命令使用

创建分之使用git branch 即可

查看当前在什么分之使用：`git branch`

切换分之使用：`git checkout dev`

删除分之：`git branch -d dev`

查看各个分支最后一次提交的对象信息：`git branch -v`

查看哪些分支已被并入当前分支：`git branch --merged`

一般来说，列表中没有 `*` 的分支通常都可以用 `git branch -d` 来删掉。原因很简单，既然已经把它们所包含的工作整合到了其他分支，删掉也不会损失什么。如：

```
$ git branch --no-merged
  testing
```

另外可以用 `git branch --no-merged` 查看尚未合并的工作,它会显示还未合并进来的分支。由于这些分支中还包含着尚未合并进来的工作成果，所以简单地用 `git branch -d` 删除该分支会提示错误，因为那样做会丢失数据:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

不过，如果你确实想要删除该分支上的改动，可以用大写的删除选项 `-D` 强制执行，就像上面提示信息中给出的那样。

部分命令使用如下：

```
[root@localhost 0822]# git branch dev
[root@localhost 0822]# git branch
  dev
* master
  [root@localhost 0822]# git checkout dev
  切换到分支 'dev'
  [root@localhost 0822]# git branch
* dev
  master
  [root@localhost 0822]#
```

5.2 分之合并

分之中的文件在没有合并的情况下都是独立的。如：在已经创建的dev分之中创建一个文件，这个文件只能在dev分之中看到

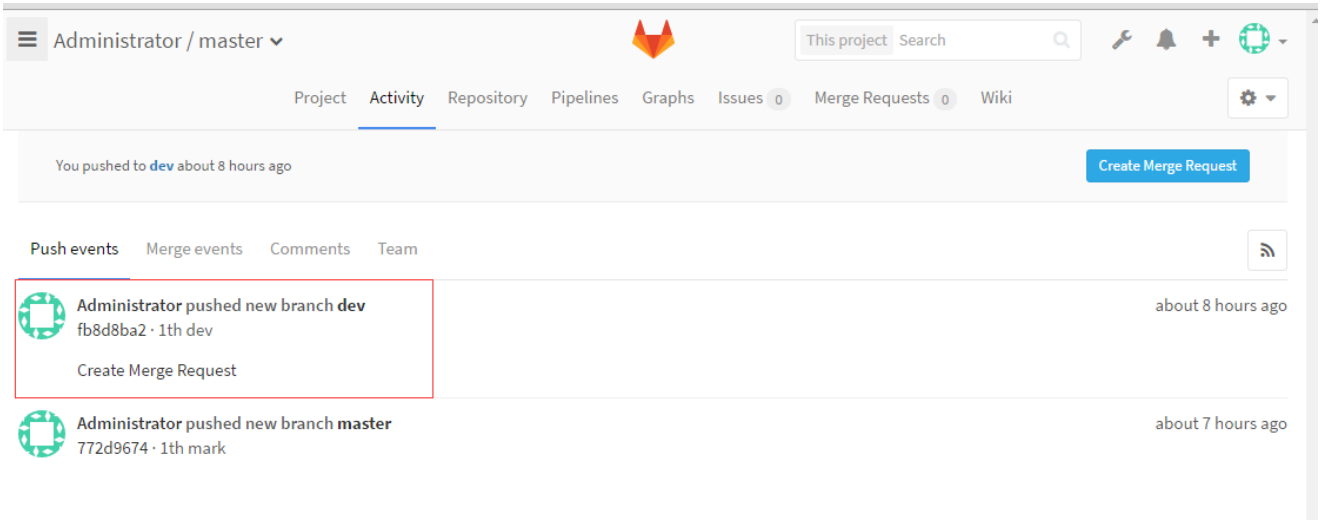
示例：当我在dev分之创建一个文dev.txt件在master分之中是看不到的，我现在将一个dev的分之合并到主干master中

5.2.1 演示分之上传

```
[root@localhost 0822]# echo "`date +%T-%F`dev" > dev.txt
[root@localhost 0822]# cat dev.txt
17:08:56-2016-08-22dev
[root@localhost 0822]# git add dev.txt
[root@localhost 0822]# git commit -m "1th dev"
[dev fb8d8ba] 1th dev
 1 file changed, 1 insertion(+)
 create mode 100644 dev.txt
[root@localhost 0822]# git push origin dev
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 282 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@10.10.239.222:root/master.git

* [new branch]      dev -> dev
[root@localhost 0822]#
```

提交完成后打开web界面查看，dev已经存在在master下



5.3 git merge 合并分之

我们需要切换到master分之后，在使用git merge 分之名称 来合并dev

```
[root@localhost 0822]# git checkout master
切换到分支 'master'
[root@localhost 0822]# ls
mark.txt
[root@localhost 0822]# git merge dev
更新 772d967..fb8d8ba
Fast-forward
 dev.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 dev.txt
[root@localhost 0822]# ll
总用量 8
-rw-r--r--. 1 root root 23 8月  22 17:15 dev.txt
-rw-r--r--. 1 root root 20 8月  22 16:20 mark.txt
[root@localhost 0822]#
```

请注意：

在合并分之的时候会出现一些问题，如下:
假设有两个分支，分别是master和dev，现在master分支下又一个文件叫做top.txt，dev分支下也有一个top.txt，并且两个分支下的top.txt都不相同是会出现合并不成功的。

六，轻量级标签管理

轻量级标签实际上就是一个保存着对应提交对象的校验和信息的文件

6.1 当前提交版本时间点打标签

在之前， 我们已经将分之合并到master中，现在master中有两个文件，分别是dev.txt mark.txt，现在，对当前打tag

git tag TAGNAME ： 打标签

git tag ： 查看当前分之标签

git show v1 查看当前标签

```
[root@localhost 0822]# git tag v1
[root@localhost 0822]# git tag
v1
[root@localhost 0822]# git show v1
commit fb8d8ba23030863245a9c8342b3b961937f93c44
Author: mark <usertzc@163.com>
Date:   Mon Aug 22 17:09:11 2016 +0800

    1th dev

diff --git a/dev.txt b/dev.txt
new file mode 100644
index 0000000..0477007
--- /dev/null
+++ b/dev.txt
@@ -0,0 +1 @@
+17:08:56-2016-08-22dev
[root@localhost 0822]#
```

6.2 推送标签

```
    [root@localhost 0822]# git push origin v1
    Total 0 (delta 0), reused 0 (delta 0)
    To git@10.10.239.222:root/master.git

    * [new tag]         v1 -> v1
```

推送标签，使用git push origin TAGNAME即可

6.3 拉取标签

git clone --bare :如果 `git clone` 出来的话，就是其中 `.git` 的目录；如果 `git clone --bare` 的话，新建的目录本身就是 Git 目录

我在另外一台机器上重新拉取标签

```
[root@localhost gitlab]#  mkdir gitlab && cd gitlab && git init
[root@localhost gitlab]# git clone git@10.10.239.222:root/master.git

正克隆到 'master'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
接收对象中: 100% (6/6), done.
```

当克隆本地后，你会发现只有一个文件，

```
    [root@localhost gitlab]# cd master/ && ls
    mark.txt
    [root@localhost master]# cat mark.txt
    16:20:03-2016-08-22
```

我们拉取到标签内的版本

```
[root@localhost master]# git checkout v1
Note: checking out 'v1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD 目前位于 fb8d8ba... 1th dev
[root@localhost master]# ls
dev.txt  mark.txt
[root@localhost master]#
```

6.4 后期添加标签

```
[root@localhost 0822]# echo 3 th up > 3th.txt
[root@localhost 0822]# git add 3th.txt
[root@localhost 0822]# ls
3th.txt  dev.txt  mark.txt
[root@localhost 0822]# git commit -m "3th up"
[master be7d412] 3th up
 1 file changed, 1 insertion(+)
 create mode 100644 3th.txt
[root@localhost 0822]# git pull origin master
来自 10.10.239.222:root/master
* branch            master     -> FETCH_HEAD
   Already up-to-date.
   [root@localhost 0822]# ls
   3th.txt  dev.txt  mark.txt
```

查看push次数和校验和：

```
[root@localhost 0822]#  git log --pretty=oneline
be7d4125b7f7563b3b55fc57a8abd1d0d897dd36 3th up
fb8d8ba23030863245a9c8342b3b961937f93c44 1th dev
772d9674472af6cf9797f95f9d40ffe6bcb63a62 1th mark
[root@localhost 0822]#
```

git tag -a v4 校验和即可

```
[root@localhost 0822]# git tag -a v4 be7d4125b7f7563b3b55fc57a8abd1d0d897dd36 -m "4th"
[root@localhost 0822]# git push origin v4
Counting objects: 1, done.
Writing objects: 100% (1/1), 148 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@10.10.239.222:root/master.git

* [new tag]         v4 -> v4
```

修改标签后在进行推送

git push origin 标签号

如果一次推送所有本地新增标签，可以使用git push origin --tags

七，忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。我们可以创建一个名为 `.gitignore` 的文件，列出要忽略的文件模式。来看一个实际的例子：

$ cat .gitignore
*.[oa]
*~

第一行告诉 Git 忽略所有以 `.o` 或 `.a` 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的，我们用不着跟踪它们的版本。第二行告诉 Git 忽略所有以波浪符（`~`）结尾的文件，许多文本编辑软件（比如 Emacs）都用这样的文件名保存副本。此外，你可能还需要忽略 `log`，`tmp` 或者 `pid` 目录，以及自动生成的文档等等。要养成一开始就设置好 `.gitignore` 文件的习惯，以免将来误提交这类无用的文件。

文件 `.gitignore` 的格式规范如下：

- 所有空行或者以注释符号 `#` 开头的行都会被 Git 忽略。
- 可以使用标准的 glob 模式匹配。
- 匹配模式最后跟反斜杠（`/`）说明要忽略的是目录。
- 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（`!`）取反。

所谓的 glob 模式是指 shell 所使用的简化了的正则表达式。星号（`*`）匹配零个或多个任意字符；`[abc]`匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；问号（`?`）只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 `[0-9]` 表示匹配所有 0 到 9 的数字）。

我们再看一个 `.gitignore` 文件的例子：

```
# 此为注释 – 将被 Git 忽略
# 忽略所有 .a 结尾的文件
*.a
# 但 lib.a 除外
!lib.a
# 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO
/TODO
# 忽略 build/ 目录下的所有文件
build/
# 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```