

Contents

Git Basic Usage.....	1
1. Git Configurations	1
2. Creating New Repos	1
2.1 git init	1
2.2 git clone.....	2
2.3 git status	2
3. Review a Repo's History	3
3.1 git log	3
3.2 git show	5
4. Add Commits To A Repo	6
4.1 git add.....	6
4.2 git commit	6
4.3 git diff.....	6
4.4 Having Git Ignore Files	7
5. Tagging, Branching, and Merging	9
5.1 git tag	10
5.2 git branch	10
5.3 git merge	12
5.4 Merge Conflicts	14
6. Undoing Changes.....	15
6.1 git commit --amend.....	16
6.2 git revert.....	16
6.3 git reset.....	17

Git Basic Usage

1. Git Configurations

基本配置命令：

```
git config --global user.name "xxxxx"
```

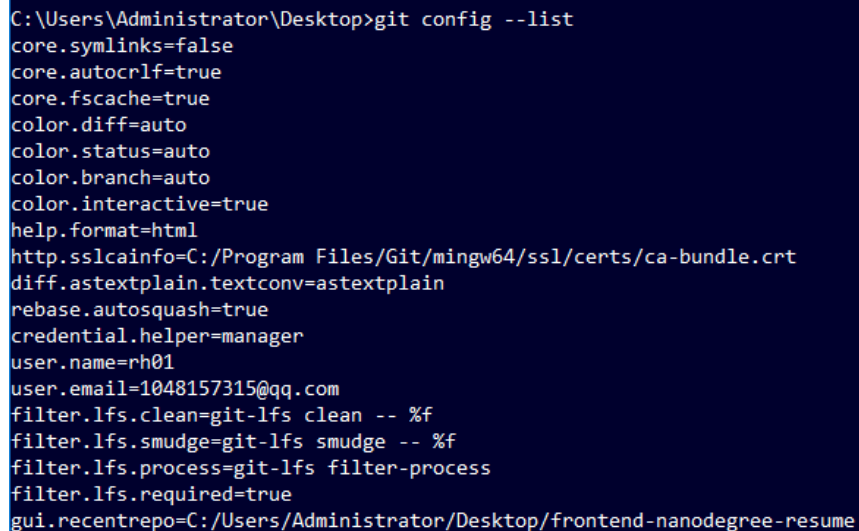
```
git config --global user.email xxx@xx.com
```

使用上述命令时，`user.name`和`user.email`均为 GitHub 注册的用户名和邮箱。

查看 Git 的配置内容：

```
git config --list
```

一般情况下打印的内容如下图所示：



```
C:\Users\Administrator\Desktop>git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
credential.helper=manager
user.name=rh01
user.email=1048157315@qq.com
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
gui.recentrepo=C:/Users/Administrator/Desktop/frontend-nanodegree-resume
```

图 1 Git 配置内容

2. Creating New Repos

主要学习的命令为：`git init`，`git clone`，`git status`

`git init`为在本地计算机创建一个新的仓库。

`git clone`为从远程或者本地 Git 服务器上 copy 一个 repo 到本地计算机。

`git status`为检查 repo 的状态信息，比如那些文件发生改变，那些需要 commit 等等。

2.1 git init

运行 `git init` 命令后，`Git`将跟踪当前文件夹下的所有内容，包括文件和目录。

所有这些文件信息都存储在名为`.git`的目录中（注意在开头`.`，这意味着它将是 Mac / Linux 上的隐藏目录）。这个`.git`目录是 `Git` 记录“repo”跟踪所有变化和提交的地方！

警告 - 不要直接编辑`.git`目录中的任何文件。这是版本库（repo）的核心。如果更改文件名和/或文件内容，git 可能会丢失在 repo 中保留的文件，并且可能会失去很多工作！可以查看这些文件，但不要编辑或删除它们。

以下是`.git`目录中每个文件夹的简要描述：

config file - 存储所有特定的项目的配置。

From the [Git Book](#):

Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. These values are specific to that single repository.

Git 在您当前使用的任何存储库的 Git 目录（`.git / config`）中的配置文件中查找配置值。这些值特定于该单个存储库。

例如，假设您在 Git 的全局配置（global）使用您的个人电子邮件地址。但是如果您希望将工作电子邮件用于特定项目而不是您的个人电子邮件，那么就应该修改该文件中的邮件设置。

description file - 此文件仅由 GitWeb 程序使用，可以忽略它。

hooks directory - 我们可以在该目录中放置客户端或服务器端脚本，这些脚本用来挂钩 Git 不同的生命周期事件。

info directory - 包含全局排除文件 objects 目录 - 此目录将存储我们所做的所有提交。

refs directory - 此目录包含提交的指针（基本上是“分支”和“标签”）。

2.2 git clone

目的：在本地构建一个副本

usage:

```
git clone https://github.com/xxx/xxxx.git
```

```
git clone git@github.com:xxx/xxx.git
```

2.3 git status

`git status` 会告诉我们 Git 正在考虑什么，以及 Git 看到的版本库的状态。当你是第一次使用它，你应该每一步都要使用 `git status` 命令！在执行完其他命令后，都要很习惯性地运行该命令。这将帮助您了解 Git 的工作原理，并帮助您对文件/版本库的状态做出正确的判断。

Git Status Explanation

比如下列一段输出：

```
On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

The output tells us two things:

1. `On branch master` – this tells us that Git is on the master branch. You've got a description of a branch on your terms sheet so this is the "master" branch (which is the default branch).
2. `Your branch is up-to-date with 'origin/master'.` – Because git clone was used to copy this repository from another computer, this is telling us if our project is in sync with the one we copied from. We won't be dealing with the project on the other computer, so this line can be ignored.
3. `nothing to commit, working directory clean` – this is saying that there are no pending changes.

Explanation Of Git Status In A New Repo

```
$ git status

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

3. Review a Repo's History

学习一下两个命令：

`git log`, `git show`

3.1 git log

`git log` 命令用于显示版本库的所有提交。

默认情况下，此命令显示：SHA、作者、日期和消息。如下图所示：

```
commit 94d305ee8f81ccaf866a76222f10357940beb5de
Author: 申恒恒(Shine) <1048157315@qq.com>
Date:   Mon May 8 14:41:30 2017 +0800

    Add files via upload

commit 589415654d071a7d5fe7ff903e18743f4dac45eb
Author: 申恒恒(Shine) <1048157315@qq.com>
Date:   Mon May 8 14:37:20 2017 +0800

    Delete db.sqlite3

commit dab3b169e12b5bda1b0c51e96877246048b5ab6
Author: rh01 <1048157315@qq.com>
Date:   Fri Mar 31 21:46:30 2017 +0800

    add publish file

commit edd7309147531b5289a37e38eb4c3ed261600947
Author: rh01 <1048157315@qq.com>
Date:   Fri Mar 31 20:27:43 2017 +0800

    first commit
```

图 2 git log 输出信息

the SHA - git log will display the complete SHA for every single commit. Each SHA is unique, so we don't really need to see the *entire* SHA. We could get by perfectly fine with knowing just the first 6-8 characters. Wouldn't it be great if we could save some space and show just the first 5 or so characters of the SHA?

the author - the git log output displays the commit author for *every single commit*! It could be different for other repositories that have multiple people collaborating together, but for this one, there's only one person making all of the commits, so the commit author will be identical for all of them. Do we need to see the author for each one? What if we wanted to hide that information?

the date - By default, git log will display the date for each commit. But do we really care about the commit's date? Knowing the date might be important occasionally, but typically knowing the date isn't vitally important and can be ignored in a lot of cases. Is there a way we could hide that to save space?

the commit message - this is one of the most important parts of a commit message...we usually always want to see this

学习一个新的命令：

```
$ git log --oneline
```

`git log --oneline` 这个命令：

1. 列出每行一个提交
2. 显示提交的 SHA 的前 7 个字符
3. 显示提交的消息

```
$ git log --stat
```

`git log --stat` 这个命令：

1. 显示已修改的文件
2. 显示已添加/删除的行数
3. 显示已添加/删除的行总数和已修改文件数的总数目的摘要行

```
$ git log -p
```

The git log command has a flag that can be used to display the actual changes made to a file. The flag is --patch which can be shortened to just -p:

```
$ git log -p
```

`git log -p` 命令用于显示对文件所做的实际更改。

1. 显示已修改的文件
2. 显示已添加/删除的行的位置
3. 显示所做的实际更改

3.2 git show

```
$ git show
```

`git show` 它只会显示最近的提交。通常，提供 SHA 作为参数：

```
$ git show fdf5493
```

git show 命令只显示一个提交。

git show 命令的输出与 git log -p 命令输出完全相同。所以默认情况下，git show 显示：

1. 提交
2. 作者日期
3. 提交消息
4. 补丁信息

4. Add Commits To A Repo

介绍三个命令：`git add`，`git commit`，`git diff`

4.1 git add

`git add` 命令用于将文件从工作目录添加到分段索引。

```
$ git add <file1> <file2> ... <fileN>
```

这个命令：

1. 采用空格分隔的文件名列表
2. 或者，`.` 可以使用代替文件列表来告诉 Git 添加当前目录（和所有嵌套文件）

4.2 git commit

一般情况下，只输入 `git commit` 指令，将会弹出没有设置编辑器路径等类似的错误信息。因此需要在第 1 节中配置时添加上默认的编辑器路径或者链接。

```
$ git config --global core.editor <your-editor's-config-went-here>
```

接下来，如果要提交时，就会弹出编辑器窗口，在里面添加提交信息，保存并关闭，即可提交上去。

```
git commit -m "xxxx"
```

这条命令绕过了编辑器。如果您正在撰写的提交消息很短，并且您不想等待代码编辑器打开再输入，可以使用 `-m` 标志直接在命令行中传递消息。

比如：

```
$ git commit -m "Initial commit"
```

输入完成后，可以通过 `git log` 命令查看提交信息。

4.3 git diff

`git diff` 命令的作用是比较两次提交之间的差距，比如改变了那些内容，添加或者删除那些文件。一般配合 `git log` 使用。

比如下面一个例子：

首先利用 `git log` 查看一下提交的历史记录。

```
C:\Users\Administrator\Desktop>Note>git log
commit 9aab8ce85d119c009299d97ceb343de88aab3b65
Author: rh01 <1048157315@qq.com>
Date:   Wed Jun 14 22:55:30 2017 +0800

    add README file.

commit 60113d29cf3f90e50430e7b83ce3b57deb01008e
Author: rh01 <1048157315@qq.com>
Date:   Wed Jun 14 22:40:34 2017 +0800

    some note and mind maps.
```

图 3 打印提交历史

然后使用 `git diff` 查看两次提交之间的不同。使用方法：`git diff SHA1 SHA2`，一般情况下给出 SHA 的前 7 位就可以了。

```
git diff 9aab8ce 60113d2
```

最后的输出结果为：

```
C:\Users\Administrator\Desktop>Note>git diff 9aab8 60113
diff --git a/README.md b/README.md
deleted file mode 100644
index 9d83993..0000000
--- a/README.md
+++ /dev/null

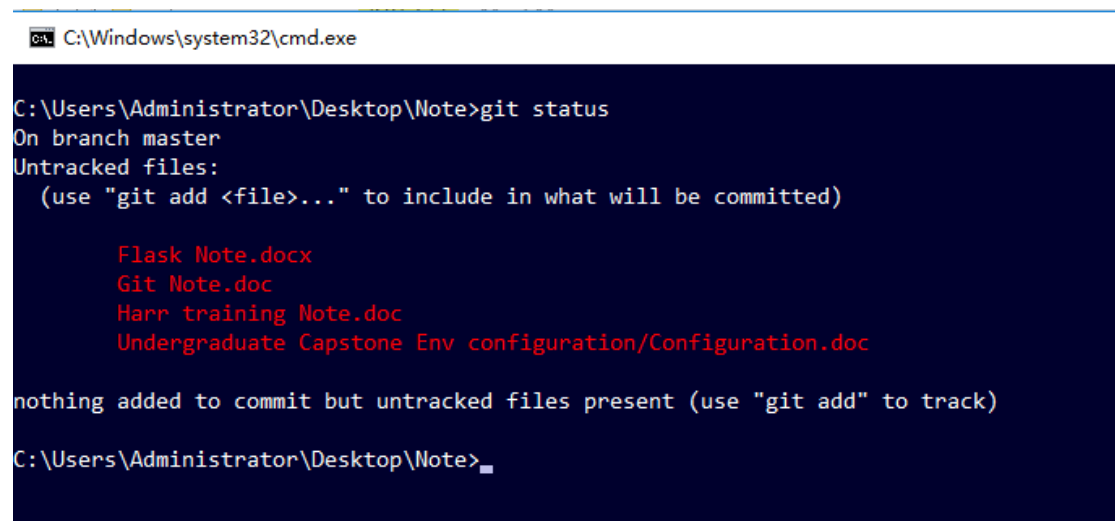
-## Project Description
-
-Notes and mind maps about learning methods and tutorials for [capstone](https://github.com/rh01/capstone) undergraduate.
-
-## Contents
-
-* Flask note,
-* harr training note.
-* build self driving RC car tutorials.
-* some beautiful mind maps.
-
-Has not yet finished updating.....
-
-## Credit
-
-Credit for [Shine](https://github.com/rh01)
```

图 4 打印出两次提交的差异

4.4 Having Git Ignore Files

上面提到了 `git add` 如何快速的添加文件，比如使用 `.` 来代替添加当前目录。但这样会存在一个问题，就是这样会把自己不想添加的文件也添加了进去。那么该如何避免添加或者如何自动化地将其忽略掉，就算添加上去，如何移除掉等等。这就是本节要学习的目标。

假设将 Word 文档添加到存储项目的目录中，但不希望将其添加到存储（repo）库。Git 会看到这个新文件，所以当运行 `git status` 命令，Word 文档会显示在文件列表中。



```
Ca: C:\Windows\system32\cmd.exe

C:\Users\Administrator\Desktop>Note>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Flask Note.docx
        Git Note.doc
        Harr training Note.doc
        Undergraduate Capstone Env configuration/Configuration.doc

nothing added to commit but untracked files present (use "git add" to track)
C:\Users\Administrator\Desktop>Note>
```

图 5 终端应用程序显示 `git status` 命令的输出。输出显示 Word 文档，位于 Git 的“未跟踪文件”部分。

`git add .` 会添加本地目录下所有文件，因此 Word 文档可能会提交到存储库。

如果要将文件保存在项目的目录结构中，但是要确保文件不会意外地提交到项目中，因此我们可以使用特殊命名文件 `.gitignore`（注意前面的点和文件名称，重要！）。将该文件添加到与 `.git` 目录所在目录相同的目录中（项目的顶级目录）。我们需要做的就是列出我们希望 Git 忽略（不跟踪）的文件名称，Git 将忽略这些文件。

在 `.gitignore` 文件中添加以下行：

```
*.doc
*.docx
```

接下来再运行 `git status` 命令。

```
C:\Windows\system32\cmd.exe

C:\Users\Administrator\Desktop\Note>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
C:\Users\Administrator\Desktop\Note>
```

图 6 终端显示 git status 的输出。 Word 文档不再列为未跟踪的文件。但是，列出了新的未跟踪文件“.gitignore”。

通配符可以匹配某种模式或者字符。常用到的通配符如下所示。

- blank lines can be used for spacing
- # - marks line as a comment
- matches 0 or more characters
- ? - matches 1 character
- [abc] - matches a, b, *or* c
- ** - matches nested directories - a/**/z matches
 - a/z
 - a/b/z
 - a/b/c/z

总结：

`.gitignore` 文件用于告诉 Git 不应该跟踪的文件。该文件应放在 `.git` 目录所在的目录中。

5. Tagging, Branching, and Merging

需要学习四个命令：`git tag`, `git branch`, `git checkout`, `git merge`。作用如表 1 所示。

表 1 功能

git tag	Add tag to specific commits.
git branch	Allow multiple lines of development.

git checkout	Switch between different branches and tags.
git merge	Combine changes on different banches.

5.1 git tag

用于与存储库标签交互的命令是 `git tag` 命令：

```
$ git tag -a v1.0
```

这将打开代码编辑器，等待提供给标签的消息。

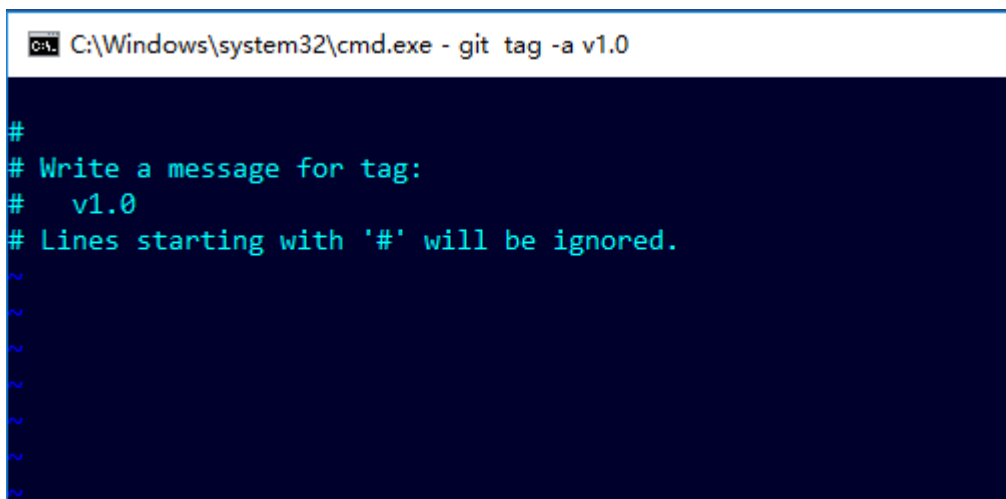


图 7 等待输入消息

保存并退出编辑器后，终端上不显示任何内容。输入 `git tag`，它将显示 repo 中的所有标签。

```
C:\Users\Administrator\Desktop>Note>git tag
v1.0
```

图 8 所有标签

`git log` 是一个非常强大的命令，它检查 repo 的所有提交。`--decorate` 标志显示隐藏在默认视图中的一些细节。

5.2 git branch

`git branch` 命令主要用来生成项目分支的。用于与 Git 的分支进行交互：

```
git branch
```

可以用于：

1. 列出存储库中的所有分支名称
2. 创建新的分支机构

3. 删除分支

如果仅仅是 `git branch` 命令，控制台将输出当前的仓库中有哪些分支和当前位于哪一个分支上。比如

```
C:\Users\Administrator\Desktop>Note>git branch
* master
```

图 10 `git branch` 的输出结果，结果表明当前只有 `master` 分支，并且当前位于 `master` 分支。

要创建一个分支，您只需使用 `git branch`，并且提供分支的名称。所以如果你想要创建一个叫做“`dev`”的分支，你可以运行这个命令：

```
git branch dev
```

记住，当提交完所有的程序之后，它会将增加到当前的分支上。所以即使你创建了新的分支 `dev`，也不会添加新的提交到 `dev` 上，因为我们没有切换到 `dev` 分支上。如果你现在增加了一个新的提交，那么该提交将被添加到主分支（`master`）上，而不是 `dev` 分支。如果要在分支之间切换，你需要使用 `Git` 的 `checkout` 命令。

```
git checkout dev
```

1. 从 `Git` 正在跟踪的工作目录中删除所有文件和目录

1. （`Git` 跟踪文件将被存储在存储库中，所以没有丢失）

2. 进入存储库并拉出对应分支所指向的所有文件和目录

因此，这将删除主分支中提交的所有文件。它将用 `dev` 分支中的提交的文件替换它们。

一个分支用于开发或对原有项目打补丁但不会影响原有的项目（因为更改是在分支上进行的）。一旦你在分支上进行更改，你可以将该分支合并到主分支中

现在分支的变更已经合并，您可能不再需要这个分支了。所以如果要删除这个分支，则可以使用 `-d` 标志。下面的命令包括 `-d` 标志，它提供给 `Git` 所要删除的分支名字（在这种情况下是“`dev`”分支）。

```
git branch -d dev
```

有一点需要注意的是，不能删除你当前所在的分支。因此，为了删除 `dev` 分支，你必须切换到主分支或创建新的分支并切换到新分支上进行删除操作。

Git Branch Recap

To recap, the `git branch` command is used to manage branches in Git:

```
# to list all branches
```

```
$ git branch
```

```
# to create a new "dev" branch
```

```
$ git branch dev
```

```
# to delete the "dev" branch
```

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

上面的命令主要说明：

1. 列出本地所有分支
2. 创建新的分支
3. 删除分支

5.3 git merge

请记住，主题分支（如 `dev`）的目的是允许你进行更改内容但不影响 `master` 主分支。对当前所在分支进行更改后，你可以删除该分支，或者决定要保留所在分支上的更改，并将其与另一个分支进行组合。

将分支结合在一起的操作称为合并（`merging`）。

Git 可以自动地把不同分支上的更改合并在一起。这种分支之间的合并操作使得 Git 变得非常强大。你可以对分支进行不同程度地修改，然后只需使用 Git 将这些更改组合在一起。

The Merge Command

The `git merge` command is used to combine Git branches:

```
$ git merge <name-of-branch-to-merge-in>
```

When a merge happens, Git will:

1. look at the branches that it's going to merge
2. look back along the branch's history to find a single commit that *both* branches have in their commit history

3. combine the lines of code that were changed on the separate branches together
4. makes a commit to record the merge

要在 dev 分支中合并，请确保在（master）主分支上运行：

```
$ git merge dev
```

因为这结合了两个不同的分支，将会做出一个承诺。当提交时，需要提供提交消息。由于这是一个合并提交，因此已经提供了一个默认消息。你可以根据需要更改消息，但常见的做法是使用默认的合并提交消息。所以当您的代码编辑器打开消息时，只需再次关闭它并接受该提交消息。

```
C:\Users\Administrator\Desktop\Note>git checkout dev
Switched to branch 'dev'

C:\Users\Administrator\Desktop\Note>git status
On branch dev
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.md

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

        Flask Note.doc
        Git Note.doc
        Harr training Note.doc
        Mind Maps/Undergraduate summary.xmind
        Mind Maps/Write README - ud777.xmind
        Undergraduate Capstone Env configuration/Configuration.doc
        ~ask Note.doc

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

C:\Users\Administrator\Desktop\Note>git commit -am "add git item in readme"
[dev fa92025] add git item in readme
1 file changed, 1 insertion(+)

C:\Users\Administrator\Desktop\Note>git checkout master
Switched to branch 'master'

C:\Users\Administrator\Desktop\Note>git merge dev
Merge made by the 'recursive' strategy.
 README.md | 1 +
1 file changed, 1 insertion(+)
```

图 11 常规的 git merge 操作

```

C:\Users\Administrator\Desktop>Note>git merge dev
CONFLICT (modify/delete): Git Note.pdf deleted in HEAD and modified in dev. Version dev of Git Note.pdf left in tree.
Automatic merge failed; fix conflicts and then commit the result.

C:\Users\Administrator\Desktop>Note>git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)

        deleted by us:   Git Note.pdf

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

        Mind Maps/Undergraduate summary.xmind
        Mind Maps/Write README - ud777.xmind
        ~WRL0001.tmp

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

C:\Users\Administrator\Desktop>Note>git commit -a
[master 94b9316] Merge branch 'dev'

C:\Users\Administrator\Desktop>Note>_

```

图 12 出现 conflicts，操作是 `git commit`，打开默认的代码编辑器，然后关闭即可，即选择默认的 commit message

Merge Recap

To recap, the git merge command is used to combine branches in Git:

```
$ git merge <other-branch>
```

There are two types of merges:

1. Fast-forward merge – the branch being merged in must be ahead of the checked out branch. The checked out branch's pointer will just be moved forward to point to the same commit as the other branch.
2. the regular type of merge
3. two divergent branches are combined
4. a merge commit is created

5.4 Merge Conflicts

大多数情况下，Git 可以将分支合并在一起，完全没有任何问题。但是，有一些合并无法自动完全执行。将合并失败的情况称为合并冲突。

如果确实发生合并冲突，Git 会尝试尽可能多地合并（就是尽量合并一些文件或目录），但是没有完成合并的则会留下特殊的标记（例如>>>和<<<），告诉

你哪些部分需要你手动修复。

那些情况会导致合并冲突

我们都知道，Git 会跟踪文件中的行。当完全相同的行在单独的分支中更改时，会发生合并冲突。例如，如果您使用 `dev` 的分支，并将 **README.md** 的标题更改为“About Me”，然后在另一个分支上，并将标题更改为“Information About Me”，Git 应选择哪个标题？你已经改变了两个分支中的 **README.md** 的标题，所以 Git 没有办法知道你实际想要保留哪一个。而且肯定不会随便选择其中一个进行合并！你可以强制合并冲突，以便可以解决冲突。

请记住，当 Git 不确定要从正在合并的分支中使用哪一行作为最终的合并对象时，会发生合并冲突。所以你需要在两个不同的分支上编辑同一行，然后尝试合并它们。

总结

Merge Conflict Recap

A merge conflict happens when the same line or lines have been changed on different branches that are being merged. Git will pause mid-merge telling you that there is a conflict and will tell you in what file or files the conflict occurred. To resolve the conflict in a file:

1. locate and remove all lines with merge conflict indicators
2. determine what to keep
3. save the file(s)
4. stage the file(s)
5. make a commit

Be careful that a file might have merge conflicts in multiple parts of the file, so make sure you check the entire file for merge conflict indicators - a quick search for `<<<` should help you locate all of them.

6. Undoing Changes

需要学习 3 个命令，`git commit --amend`，`git revert`，`git reset`，具体的作用如表 2 所示。

表 2 功能

git commit --amend	Alter the most-recent(Last) commit
git branch xxxx	Reverses given commit.
git reset	Erases commits

6.1 git commit --amend

假设你已经使用 `git commit` 命令进行了大量的提交。现在使用 `--amend` 标志，可以更改最近的一次提交。

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

如果您的工作目录很干净（意味着存储库中没有任何未提交的更改信息），那么运行 `git commit --amend` 将要求你提供一个新的提交消息。此时代码编辑器将打开并显示原始提交消息。你只需要增加内容或完全重写消息！然后保存并关闭编辑器以确定新的提交消息。

6.2 git revert

当你告诉 Git 要还原一个特定的提交时，Git 会改变上次提交的内容，并且完全相反操作。

举个例子，如果在提交 A 中添加了一个字符，如果像回溯上次提交 A，那么 Git 将会把提交 A 中的字符进行删除并进行一次新的提交。它也可以以另一种方式工作，如果一个字符/行被删除，然后还原该提交以添加已删除的内容！

```
$ git revert <SHA-of-commit-to-revert>
```

<SHA-of-commit-to-revert>表示要回溯提交的 ID 哈希值（7 位）。

Revert Recap

To recap, the git revert command is used to reverse a previously made commit:

```
$ git revert <SHA-of-commit-to-revert>
```

This command:

1. will undo the changes that were made by the provided commit

2. creates a new commit to record the change

6.3 git reset

`reset` 可能似乎巧合地接近 `revert`，但实际上是完全不同的。`revert->恢复` 会创建一个新的提交，它会恢复或取消先前的提交。另一方面，`reset->重置` 会擦除提交！

⚠ 重置十分危险 ⚠

你必须小心 Git 的重置功能。这是允许用户从存储库中删除提交的少数命令之一。如果提交不再在存储库中，则提交改变或添加删除的内容将不复存在。

但是，Git 在完全擦除任何东西之前，会跟踪所有内容约 30 天。要访问此内容，您需要使用 `git reflog` 命令。查看这些链接了解更多信息：

[git-reflog](#)

[Rewriting History](#)

[reflog, your safety net](#)

由于不经常使用，所以在这部分不做太多的陈述。[具体详见](#)