Git Lesson 2 -- Git Basics

Git Basics

well well well

实际上，很大可能，在相当长的一段时间里，你只会使用到本节课所学习到的`git commands`，因此这节课是最基础，同时也是最重要的一节课。

Getting a Git Repository

What is a Git repository?

`Repository`，有时简称`repo`，如其名所述，这是一个仓库/版本库，用于追踪并保存 Git 项目文件中的所有更改历史，然后把所有数据保存在`.git`,文件中。我们通过 Git 来进行版本控制，用户可以在此删除或复制已有的版本库或为现有的项目创建新的版本库。

上一节课说过，对于***DVCS***，在服务器端存储有一份项目代码仓库，这个叫做**远程仓库**，而每个开发者本机项目代码仓库，叫做**本地仓库**，将远程仓库整个拷贝到本机的操作，我们称之为**克隆**（`clone`），本地仓库可以向远程仓库**推送**（`push`）文件，也可以从远程仓库**拉取**（`pull`）。

Initializing a Repository in an Existing Directory

如果你有一个尚未进行版本控制的项目目录，想要用 Git 来控制它，那么首先需要进入该项目目录中。 对于 `Windows`，请在 `Git Bash` 中进入到项目目录，`Linux` 和 `Mac` 在 `CLI` 中进入即可。

随后执行指令 `git init`，该命令将你的项目文件夹转变成 `Git Repository`，并创建一个名为 `.git` 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干（我们并不会深入地学习 `Git` 的原理，因此你无需理解 `.git` 里的文件的作用）。 但是，在这个时候，我们仅仅是做了一个初始化的操作，你的项目里的文件还没有被跟踪（`tracked`）。

如果你的项目文件夹本身是一个非空目录且想要进行版本控制，那么你应该开始跟踪这些文件并且初始提交，你可以通过 `git add` 命令来指定所需的文件来进行追踪，然后执行 `git commit`：

```
git add *
git commit -m 'initial commit'
```

稍后我们再逐一解释这些指令的行为。 现在，你已经得到了一个存在被追踪文件与初始提交的 Git 仓库。

Cloning an Existing Repository

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，你想为某个开源项目贡献自己的一份力，这时就要用到 `git clone` 命令。Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 `git clone` 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。

克隆仓库的命令是 `git clone <url>` 。 比如，要克隆 eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee 的库 `eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee`，可以用下面的命令：

```
git clone https://github.com/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee.git
```

这会在当前目录下创建一个名为 “eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee” 的目录，并在这个目录下初始化一个 `.git` 文件夹， 从远程仓库拉取下所有数据放入 `.git` 文件夹，然后从中读取最新版本的文件的拷贝。 如果你进入到这个新建的 `eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee` 文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。

如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以通过额外的参数指定新的目录名：

```
git clone https://github.com/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee.git eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
```

Git 支持多种数据传输协议。 上面的例子使用的是 `https` 协议，不过你也可以使用 `SSH` 传输协议，比如 `git@github.com:eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee.git`。

Recording Changes to the Repository

首先，除了**版本库**外，你还需要掌握两个概念：

- **工作区/工作目录**

  简单来说，就是你能够看到的文件夹

- **暂存区（*Staging area*）**

  暂存区是位于Git仓库内部的一个中间区域，也就是版本库 `.git` 目录下的 `index` 文件。暂存区的含义是，在对项目文件进行修改后，这些修改并不会立即被提交到版本库中。相反，你需要将这些修改先添加到暂存区，然后才能将其作为一个整体提交到版本库中。其中包含了项目文件。当对项目文件进行修改时，这些修改只存在于工作目录中，并没有被Git跟踪。

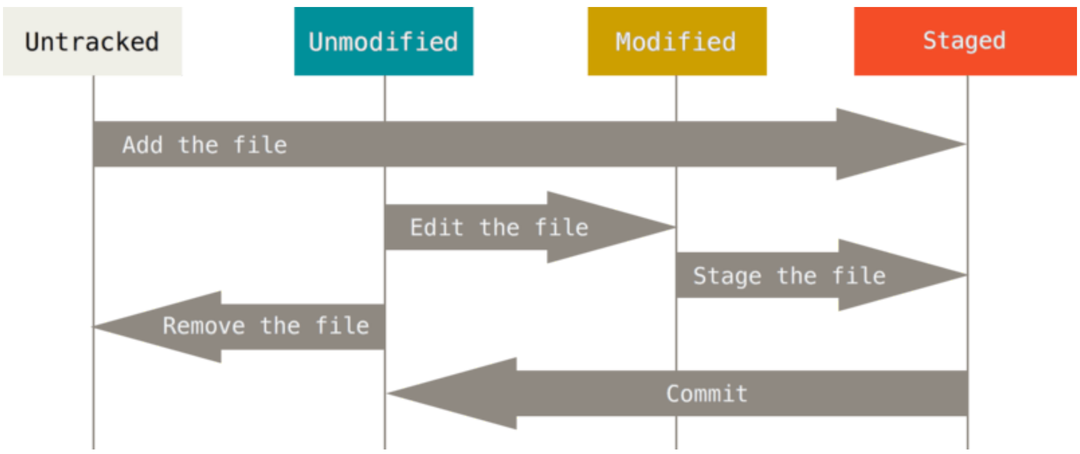
如果你暂时没有理解以上两个概念，没关系，先记住，马上就会了解了。

此时，你应该已经在本地计算机上有了一个真正的 Git 仓库，并有了它所有文件的签出或工作副本。通常情况下，每当项目达到你想要记录的状态时，你就会开始进行修改，并将这些修改的快照提交到仓库中。

请记住，工作目录中的每个文件都有两种状态：**已跟踪**或**未跟踪**。跟踪文件是上次快照中的文件，以及任何新缓存的文件；它们可以是未修改的、已修改的或已缓存的。简而言之，***跟踪文件就是 Git 知道的文件***。

未跟踪文件指的是**其他所有文件**--工作目录中的任何文件，这些文件*既不在上次快照中，也不在暂存区域中*。当你第一次克隆一个仓库时，所有文件都会被跟踪且未修改，因为 Git 刚刚签出了它们，你还没有编辑过任何东西。

当你编辑文件时，Git 会将它们视为修改过的文件，因为你在上次提交后已经修改了它们。在工作过程中，你会选择性地对这些修改过的文件进行阶段化，然后提交所有这些阶段化的修改，如此循环往复。



Checking the Status of Your Files

正如上图所示，文件的状态有`untracked`，`unmodified`，`modified`，`staged`，想要查看哪些文件处于什么状态，你可以使用`git status`指令。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ ls
modified  staged  unmodified  untracked

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working di
rectory)
        modified:   modified

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

Tracking New Files

Git并不会自动跟踪新文件，除非你明确地告诉它“我需要跟踪该文件”。 这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。现在假设我们新建了一个`README.md`，并想要跟踪它，那么可以使用命令 `git add` 开始跟踪，运行过后此时再运行 `git status` 命令，会看到 `README.md` 文件已被跟踪，并处于暂存状态。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ touch README.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ vim README.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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


EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git add README.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

只要在 `Changes to be committed` 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件在你运行 `git add` 时的版本将被留存在后续的历史记录中。 `git add` 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。例如`git add *`就是对当前目录下所有可执行该命令的文件执行`git add`。

Staging Modified Files

现在我们来修改一个已被跟踪的文件。 如果你修改了一个名为 `DONNOTMODIFY.md` 的已被跟踪的文件，然后运行 `git status` 命令

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ vim DONNOTMODIFY.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   DONNOTMODIFY.md
        modified:   modified

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

文件 `DONNOTMODIFY.md` 出现在 `Changes not staged for commit` 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 `git add` 命令。 这是个多功能命令：*可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。* ***将这个命令理解为“精确地将内容添加到下一次提交中”而不是“将一个文件添加到项目中”要更加合适。*** 现在让我们运行 `git add` 将 `DONNOTMODIFY.md` 放到暂存区，然后再看看 `git status` 的输出：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git add DONNOTMODIFY.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   DONNOTMODIFY.md
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

现在 `DONNOTMODIFY.md` 和 `README.md` 都已暂存，下次提交时就会一并记录到仓库。 假设此时，你想要在 `DONNOTMODIFY.md` 里再加条注释。 重新编辑存盘后，准备好提交。 不过且慢，再运行 `git status` 看看：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ vim DONNOTMODIFY.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   DONNOTMODIFY.md
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   DONNOTMODIFY.md
        modified:   modified

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

你会发现 `DONNOTMODIFY.md` 文件同时出现在暂存区和非暂存区。 这是怎么回事？实际上 Git 只不过暂存了你运行 `git add` 命令时的版本。 如果你现在提交， `DONNOTMODIFY.md` 的版本是你最后一次运行 `git add` 命令时的那个版本，而不是你运行 `git commit` （后面会讲这个指令）时，在工作目录中的当前版本。 所以，运行了 `git add` 之后又作了修订的文件，需要重新运行 `git add` 把最新版本重新暂存起来：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git add DONNOTMODIFY.md

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   DONNOTMODIFY.md
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

Short Status

`git status` 命令的输出十分详细，但其用语有些繁琐。 Git 有一个选项可以帮你缩短状态命令的输出，这样可以以简洁的方式查看更改。 如果你使用 `git status -s` 命令或 `git status --short` 命令，你将得到一种格式更为紧凑的输出。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status -s
M  DONNOTMODIFY.md
A  README.md
 M modified
A  staged
?? untracked
```

未跟踪文件前面有 `??` 标记，新添加到暂存区中的文件前面有 `A` 标记，修改过的文件前面有 `M` 标记。 输出中有两栏，左栏指明了暂存区的状态，右栏指明了工作区的状态（注意到有两个`M`，分别在左栏和右栏，左栏右栏`M`其实也可以同时出现在一个文件）。

Ignoring Files

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。 在这种情况下，我们可以创建一个名为 `.gitignore` 的文件，列出要忽略的文件的模式。 来看一个 `.gitignore` 的例子：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ 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。 第二行告诉 Git 忽略所有名字以波浪符（~）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。 要养成一开始就为你的新仓库设置好 `.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 `#` 开头的行都会被 Git 忽略。
- 可以使用标准的 glob 模式匹配，它会递归地应用在整个工作区中。
- 匹配模式可以以（`/`）开头防止递归。
- 匹配模式可以以（`/`）结尾指定目录。
- 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）。 使用两个星号（`**`）表示匹配任意中间目录，比如 `a/**/z` 可以匹配 `a/z` 、 `a/b/z` 或 `a/b/c/z` 等。

我们再看一个 `.gitignore` 文件的例子：

```
# 忽略所有的 .a 文件
*.a

# 但跟踪所有的 lib.a，即便你在前面忽略了 .a 文件
!lib.a

# 只忽略当前目录下的 TODO 文件，而不忽略 subdir/TODO
/TODO

# 忽略任何目录下名为 build 的文件夹
build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt
doc/*.txt

```

```
# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件
doc/**/*.pdf
```

> GitHub 有一个十分详细的针对数十种项目及语言的 `.gitignore` 文件列表， 你可以在 https://github.com/github/gitignore 找到它。

Viewing Your Staged and Unstaged Changes

如果你更想知道具体文件修改了什么地方，那么`git status`的输出内容就有些太过于简略了，那么你可以使用`git diff`指令。你通常可能会用它来回答这两个问题：当前做的哪些更新尚未暂存？ 有哪些更新已暂存并准备好下次提交？ 虽然`git status`已经通过在相应栏下列出文件名的方式回答了这个问题，但`git diff`能通过文件补丁的格式更加具体地显示哪些行发生了改变。

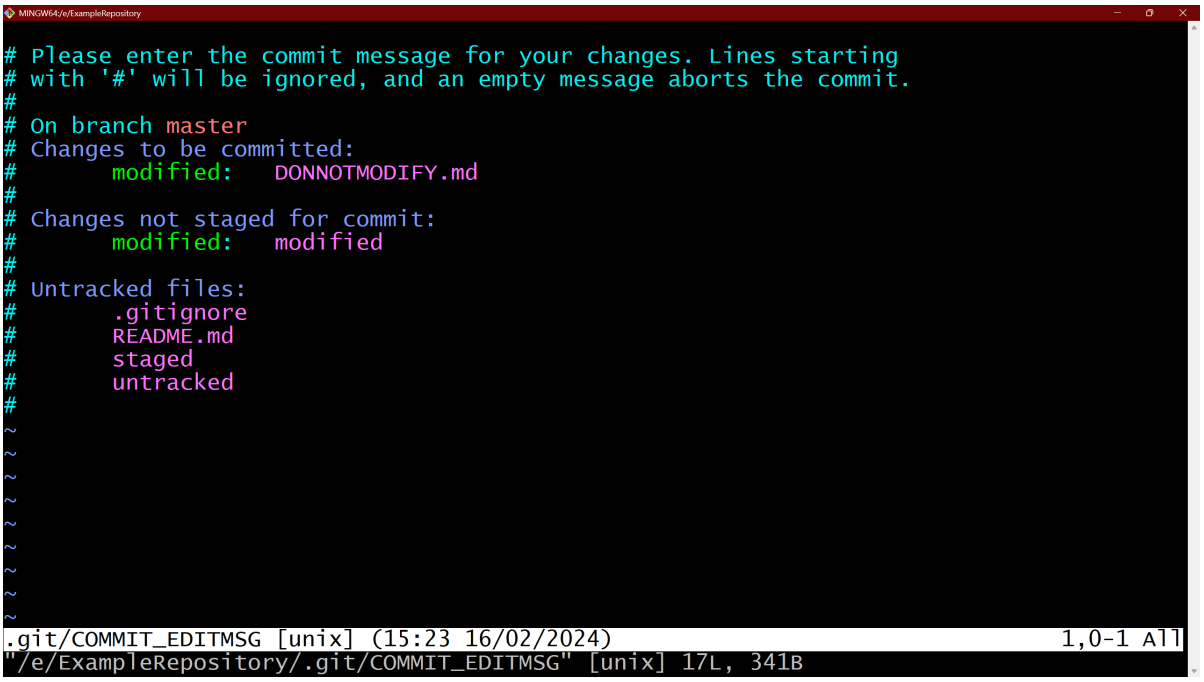
- 不加参数的`git diff`，比较的是工作目录中当前文件和暂存区域快照之间的差异。 也就是修改之后还没有暂存起来的变化内容。
- 若要查看已暂存的将要添加到下次提交里的内容，可以用`git diff --staged`命令。 这条命令将比对已暂存文件与最后一次提交的文件差异。
- 用`git diff --cached`可以查看已经暂存起来的变化。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   DONNOTMODIFY.md
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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


EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git diff
diff --git a/modified b/modified
index e69de29..a07e6a9 100644
--- a/modified
+++ b/modified
@@ -0,0 +1 @@
+Modified

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git diff --staged
diff --git a/DONNOTMODIFY.md b/DONNOTMODIFY.md
index d99a2aa..ca97c28 100644
--- a/DONNOTMODIFY.md
+++ b/DONNOTMODIFY.md
@@ -1,2 +1,3 @@
 # DO NOT MODIFY THIS FILE
-***Okay, but, no.***
+***No.***
+***DAMN!!! WHO MODIFIED THIS FILE!!!***
diff --git a/README.md b/README.md
new file mode 100644
index 0000000..194235a
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
+# Example Repository
+***The example repository for the Git lecture.***
diff --git a/staged b/staged
new file mode 100644
index 0000000..e69de29

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git diff --cached
diff --git a/DONNOTMODIFY.md b/DONNOTMODIFY.md
index d99a2aa..ca97c28 100644
--- a/DONNOTMODIFY.md
+++ b/DONNOTMODIFY.md
@@ -1,2 +1,3 @@
 # DO NOT MODIFY THIS FILE
-***Okay, but, no.***
+***No.***
+***DAMN!!! WHO MODIFIED THIS FILE!!!***
diff --git a/README.md b/README.md
new file mode 100644
index 0000000..194235a
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
```

```
+# Example Repository
+***The example repository for the Git lecture.***
diff --git a/staged b/staged
new file mode 100644
index 0000000..e69de29
```

Committing Your Changes

现在的暂存区已经准备就绪，可以提交了。 在此之前，请务必确认还有什么已修改或新建的文件还没有 `git add` 过， 否则提交的时候不会记录这些尚未暂存的变化。 这些已修改但未暂存的文件只会保留在本地磁盘。 所以，每次准备提交前，先用 `git status` 看下，你所需要的文件是不是都已暂存起来了， 然后再运行提交命令 `git commit`，这样会触发你配置时选择的文本编辑器来输入提交说明，如下图即 `vim`：



可以看到，默认的提交消息包含最后一次运行 `git status` 的输出，放在注释行里，另外开头还有一个空行，供你输入提交说明。 你完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。退出编辑器时，Git 会丢弃注释行，用你输入的提交说明生成一次提交。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   DONNOTMODIFY.md
        new file:   README.md
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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


EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git commit DONNOTMODIFY.md
[master 9150ae4] Demonstrate the commit operate.
 1 file changed, 2 insertions(+), 1 deletion(-)
```

你也可以在 `commit` 命令后添加 `-m` 选项，将提交信息与命令放在同一行，这样可能会更加便捷：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git commit README.md -m "Demonstrate the commit operation with -m option"
[master de0ea30] Demonstrate the commit operation with -m option
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

可以看到，提交后它会告诉你，当前是在哪个分支（`master`）提交的，本次提交的完整 SHA-1 校验和是什么，以及在本次提交中，有多少文件修订过，多少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。 任何还未暂存文件的仍然保持已修改状态，可以在下次提交时纳入版本管理。 每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

之前的两处演示都是提交部分文件，如果你直接键入`git commit`或者`git commit -m "message"`，那么将会提交所有已暂存文件。

Skipping the Staging Area

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。 Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤。

也就是说，使用该指令的话，再提交之前就不用再添加到暂存区了，因为 `-a` 选项使本次提交包含了所有修改过的文件。 这很方便，但是要小心，有时这个选项会将不需要的文件添加到提交中或者提交暂时不想要提交的修改。

Removing Files

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：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   removed
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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


EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ rm removed

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   removed
        new file:   staged

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified
        deleted:    removed

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

我们需要再运行 `git rm` 记录此次移除文件的操作：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git rm removed
rm 'removed'

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

下一次提交时，该文件就不再纳入版本管理了。 如果要删除之前修改过或已经放到暂存区的文件，则必须使用强制删除选项 `-f`（即 force）。 这是一种安全特性，用于防止误删尚未添加到快照的数据，这样的数据不能被 Git 恢复。

`git rm` 和 `git rm -f` 的主要区别在于它们处理已修改但未提交的文件的方式。

- `git rm`：如果文件自上次提交后已被修改，`git rm` 会拒绝删除该文件。这是因为 `git rm` 默认要求要删除的文件必须与分支的最新提交保持一致，并且在索引中不能有对其内容的更新。
- `git rm -f`：即使文件自上次提交后已被修改，`git rm -f` 也会强制删除该文件。这是因为 `-f` 参数可以覆盖对文件删除的安全检查。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ touch removed

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git add removed

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git commit removed -m "Commit to removed"
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

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

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ vim removed

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git rm removed
error: the following file has local modifications:
    removed
(use --cached to keep the file, or -f to force removal)

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git rm -f removed
rm 'removed'
```

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，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 `.gitignore` 文件，不小心把一个很大的日志文件或一堆 `.a` 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 `--cached` 选项：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git rm --cached removed
rm 'removed'

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ ls
DONNOTMODIFY.md  README.md  modified  removed  staged  unmodified  untracked
```

`git rm` 命令后面可以列出文件或者目录的名字，也可以使用 `glob` 模式。

Moving Files

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。 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，别问我就对了。

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：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    removed -> renamed
        modified:   staged

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified

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

其实，运行 `git mv` 就相当于运行了下面三条命令：

```
mv file_from file_to
git rm file_from
git add file_to
```

如此分开操作，Git 也会意识到这是一次重命名，所以不管何种方式结果都一样。 两者唯一的区别在于，`git mv` 是一条命令而非三条命令，直接使用 `git mv` 方便得多。 不过在使用其他工具重命名文件时，记得在提交前 `git rm` 删除旧文件名，再 `git add` 添加新文件名。

Viewing the Commit History

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。 完成这个任务最简单而又有效的工具是 `git log` 命令。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git log
commit fa2571aa6ca76c06e120fc68f0a656e7bbdbd7dd (HEAD -> master)
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:56:56 2024 +0800

    staged

commit a1f5ee36e19851325f30297db7fa8ce425e7cdb1
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:46:52 2024 +0800

    Commit to removed

commit de0ea30b62cd75e5477047f958b1d4337bb29a36
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:28:32 2024 +0800

    Demonstrate the commit operation with -m option

commit 9150ae452ae075106dbb8b44ed8d99a8dd4341a9
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:23:11 2024 +0800

    Demonstrate the commit operate.

commit 1053be257e2eb9689d35c968c390057183f2a328
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 14:08:41 2024 +0800

    Do not modify this filegit status!

commit 73eaf5ab26151f52863d2905859160d653941455
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Tue Feb 13 17:03:59 2024 +0800

    modified

commit 106f7337afd9c7210ad24297dba962ab64c14c03
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Tue Feb 13 17:03:45 2024 +0800

    modified
```

不传入任何参数的默认情况下，`git log` 会按时间先后顺序列出所有的提交，最近的更新排在最上面。 正如你所看到的，这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

`git log` 有许多选项可以帮助你搜寻你所要找的提交， 下面我们会介绍几个最常用的选项。

其中一个比较有用的选项是 `-p` 或 `--patch` ，它会显示每次提交所引入的差异（按 **补丁** 的格式输出）。 你也可以限制显示的日志条目数量，例如使用 `-2` 选项来只显示最近的两次提交：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git log -p -2
commit fa2571aa6ca76c06e120fc68f0a656e7bbdbd7dd (HEAD -> master)
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:56:56 2024 +0800

    staged

diff --git a/staged b/staged
new file mode 100644
index 0000000..e69de29

commit a1f5ee36e19851325f30297db7fa8ce425e7cdb1
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:46:52 2024 +0800

    Commit to removed

diff --git a/removed b/removed
new file mode 100644
index 0000000..e69de29
```

该选项除了显示基本信息之外，还附带了每次提交的变化。 当进行代码审查，或者快速浏览某个搭档的提交所带来的变化的时候，这个参数就非常有用了。 你也可以为 `git log` 附带一系列的总结性选项。 比如你想看到每次提交的简略统计信息，可以使用 `--stat` 选项：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git log --stat
commit fa2571aa6ca76c06e120fc68f0a656e7bbdbd7dd (HEAD -> master)
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:56:56 2024 +0800

    staged

 staged | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit a1f5ee36e19851325f30297db7fa8ce425e7cdb1
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:46:52 2024 +0800

    Commit to removed

 removed | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit de0ea30b62cd75e5477047f958b1d4337bb29a36
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:28:32 2024 +0800

    Demonstrate the commit operation with -m option

 README.md | 2 ++
 1 file changed, 2 insertions(+)

commit 9150ae452ae075106dbb8b44ed8d99a8dd4341a9
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:23:11 2024 +0800

    Demonstrate the commit operate.
```

正如你所看到的，`--stat` 选项在每次提交的下面列出所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。 在每次提交的最后还有一个总结。

另一个非常有用的选项是 `--pretty`。 这个选项可以使用不同于默认格式的方式展示提交历史。 这个选项有一些内建的子选项供你使用。 比如 `oneline` 会将每个提交放在一行显示，在浏览大量的提交时非常有用。 另外还有 `short`，`full` 和 `fuller` 选项，它们展示信息的格式基本一致，但是详尽程度不一：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git log --pretty=oneline
fa2571aa6ca76c06e120fc68f0a656e7bbdbd7dd (HEAD -> master) staged
a1f5ee36e19851325f30297db7fa8ce425e7cdb1 Commit to removed
de0ea30b62cd75e5477047f958b1d4337bb29a36 Demonstrate the commit operation with -m option
9150ae452ae075106dbb8b44ed8d99a8dd4341a9 Demonstrate the commit operate.
1053be257e2eb9689d35c968c390057183f2a328 Do not modify this filegit status!
73eaf5ab26151f52863d2905859160d653941455 modified
106f7337afd9c7210ad24297dba962ab64c14c03 modified
```

对于`pretty`更详尽的使用描述参见这里，我们不过多说明。

除了定制输出格式的选项之外，`git log` 还有许多非常实用的限制输出长度的选项，也就是只输出一部分的提交。 之前你已经看到过 `-2` 选项了，它只会显示最近的两条提交， 实际上，你可以使用类似 `-<n>` 的选项，其中的 `n` 可以是任何整数，表示仅显示最近的 `n` 条提交。 不过实践中这个选项不是很常用，因为 Git 默认会将所有的输出传送到分页程序中，所以你一次只会看到一页的内容。

更多的输出控制选项，参见此处，我们不过多说明。

Undoing Things

Undoing Commit

在任何一个阶段，你都有可能想要撤消某些操作。 有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 `--amend` 选项的提交命令来重新提交，这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令）， 那么快照会保持不变，而你所修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。 编辑后保存会覆盖原来的提交信息。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git commit forgotten_file
[master 8ee6df1] I forgett something.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 forgotten_file

EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git commit forgotten_file --amend
[master 1f926b4] I forget something.
 Date: Fri Feb 16 16:51:19 2024 +0800
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 forgotten_file
```

最终你只会有一个提交——第二次提交将代替第一次提交的结果。

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git log -2
commit 1f926b41c054aa05225218d6bd9c661d43c07285 (HEAD -> master)
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 16:51:19 2024 +0800

    I forget something.

commit fa2571aa6ca76c06e120fc68f0a656e7bbdbd7dd
Author: EvanWong <yufenghuang009@gmail.com>
Date:   Fri Feb 16 15:56:56 2024 +0800

    staged
```

Undoing Staged File

假设你已经修改了两个文件并且想要将它们作为两次独立的修改提交， 但是却意外地输入 `git add *` 暂存了它们两个。如何只取消暂存两个中的一个呢？ `git status` 命令提示了你：

```
EvanWong@LAPTOP-KPB1046C MINGW64 /e/ExampleRepository (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
```

在 “Changes to be committed” 文字正下方，提示使用 `git restore --staged <file>…` 来取消暂存。这条指令会从暂存区域删除文件，但保留实际修改内容。

Unmodifying a Modified File

如果你并不想保留文件的修改怎么办？ 你该如何方便地撤消修改——将它还原成上次提交时的样子（或者刚克隆完的样子，或者刚把它放入工作目录时的样子）？ 幸运的是，`git status` 也告诉了你应该如何做：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   modified
```

使用指令 `git restore <file>…` 来放弃修改。

最后，记住，在 Git 中任何 **已提交** 的东西几乎总是可以恢复的。 甚至那些被删除的分支中的提交或使用 `--amend` 选项覆盖的提交也可以恢复 。 然而，任何你未提交的东西丢失后很可能再也找不到了。