



TAIJI LABORATORY
FOR GRAVITATIONAL WAVE UNIVERSE



ICTP-AP
International Centre
for Theoretical Physics Asia-Pacific
国际理论物理中心-亚太地区



中国科学院大学
University of Chinese Academy of Sciences

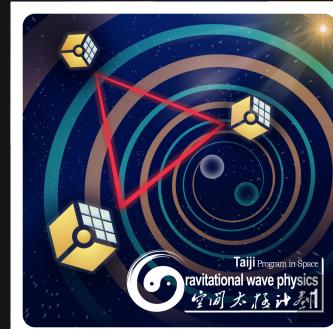
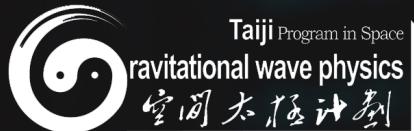
引力波数据探索：编程与分析实战训练营

第 1 部分 编程开发环境与工作流 Git 分布式版本控制系统

主讲老师：王赫

ICTP-AP, UCAS

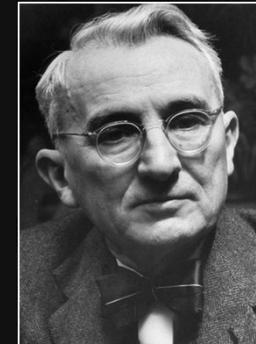
2023/11/19



关于上一讲的学员反馈



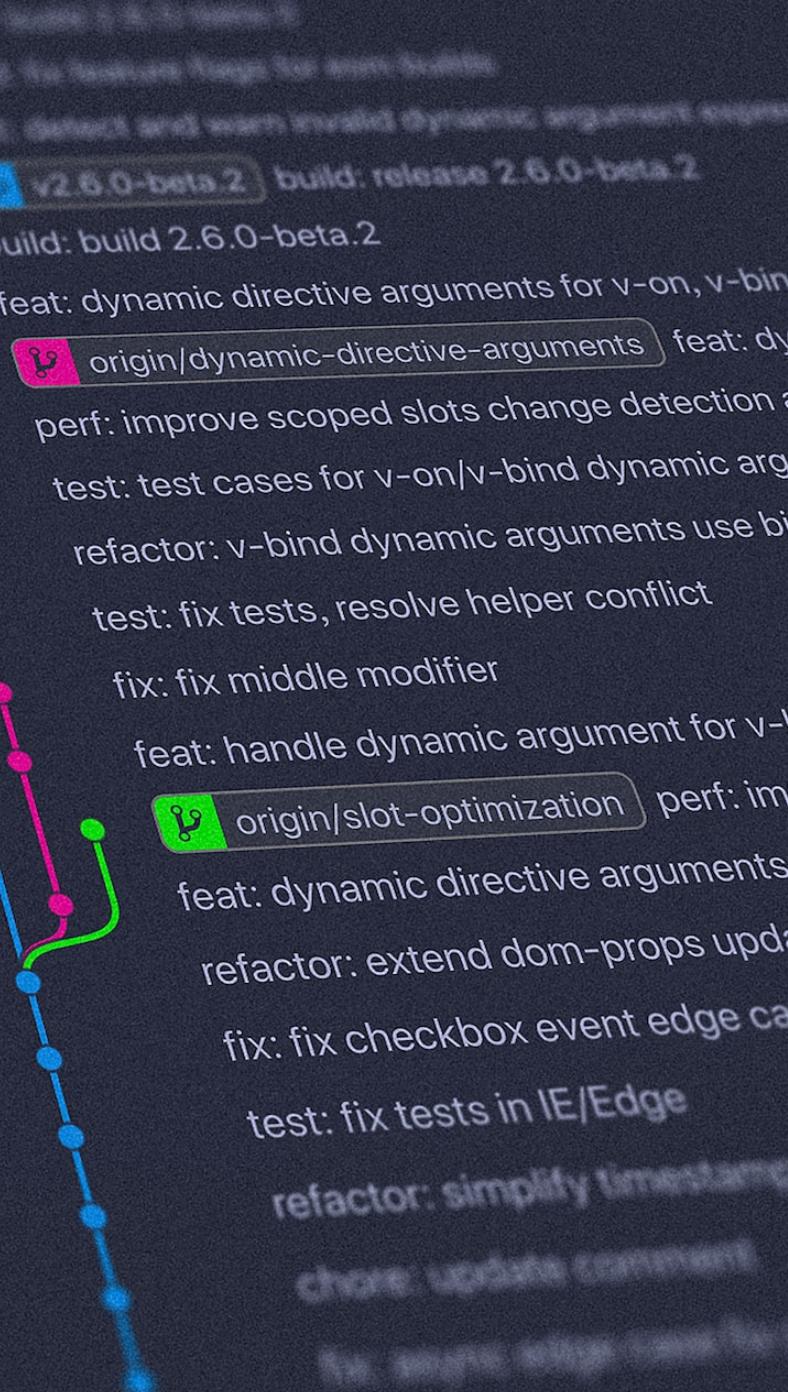
- 授课主要面向的是国科大以引力波数据处理为主要研究方向的研究生和高年级本科生。
- 不管“包学包会”，但会尽可能弥补和完善听课体验和授课内容。



You cannot teach a man anything,
you can only help him find it within
himself.

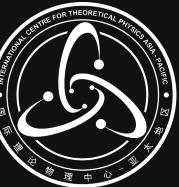
— Dale Carnegie —

AZ QUOTES



Git 分布式版本控制系统

- Git 安装/创建版本库
- 工作区、暂存区、版本库
- 远程仓库
- 分支管理
- Git 可视化管理工具



Git 分布式版本控制系统



现代的版本控制系统可以帮助您轻松地（甚至自动地）回答以下问题：

- 当前的代码模块是谁编写的？
- 这个文件的这一行是什么时候被编辑的？是谁作出的修改？修改原因是什么呢？
- 最近的1000个版本中，何时/为什么导致了单元测试失败？

廖雪峰的Git简介

Linux就该这么学：使用Git分布式版本控制系统

版本	用户	说明	日期
1	Ronny	创建Git章节文档	10/12 13:48
2	Dave	新增Git命令介绍	10/15 12:19
3	Aaron	新增Github使用方法	10/20 8:32
4	Kim	改正文章中的错别字	10/30 15:17

- **Git**

Git不仅是一款开源的**分布式版本控制系统**，而且有其独特的功能特性，

- 例如大多数的分布式版本控制系统只会记录每次文件的变化，说白了就是只会关心文件的内容变化差异，而Git则是关注于文件数据整体的变化，直接会将文件提交时的数据保存成快照，而非仅记录差异内容，并且使用SHA-1加密算法保证数据的完整性。

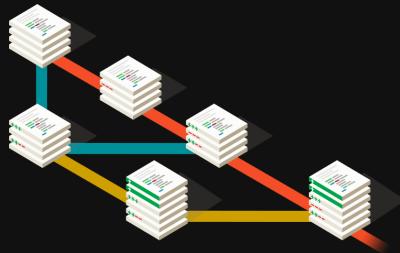
```

File View Diff View
200
201
202
203
204
205
206
207
208
209 %
210 % more detail on deep learning
211 %
212 A deep learning algorithm is composed of arrays of processing units, called
213 neurons, which can be anywhere from one to several layers deep. A neuron acts
214 as a filter, whereby it is passed a vector of inputs, performs a transformation
215 on them and then outputs a single scalar value. Deep learning algorithms
216 typically consist of an input layer, followed by one to several hidden layers
217 and then one to $NS$->chris{what is $NS$? Don't introduce something without
218 defining it or that we won't use again.} fully-connected neurons. This value
219 can then either be used to solve classification, or regression-like problems. In
220 the case of classification, each output neuron corresponds to the probability
221 that a particular input sample is of a certain class.
222
223
224
225
226 %
227 In this letter we investigate the simplest case of establishing whether a
228 signal is present in the data or if the data contains only detector noise. We
229 propose a deep learning procedure requiring only the raw data time
230 series as input with minimum signal pre-processing. We show how this approach
231 can be pre-trained using simulated data-sets and applied in low-latency
232 to achieve the same sensitivity as established matched filtering techniques.
233
234 %
235 % the structure of the paper
236 %
237 In the following sections we will discuss our choice of network architecture
238 and tuning of its hyperparameters, compare the results of our network with the
239 widely used matched-filtering gravitational-wave signal classification
technique, and comment on future improvements and applications of this work.

```

可以方便对 latex 内容的修改跟踪

Git 分布式版本控制系统



• Git 安装

- 官方网址: <https://git-scm.com/>
- 安装完成后, 第一个要配置的是你个人的用户名和电子邮件地址, 这两条配置很重要, 每次 Git 提交时都会引用这两条信息, 记录是谁提交了文件, 并且会随更新内容一起被永久纳入历史记录:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
$ git config --global core.editor vim
$ git config --list # 查看刚刚配置的工作环境
```



Generally, the best way to learn git is probably to first only do very basic things and not even look at some of the things you can do until you are familiar and confident about the basics.

— Linus Torvalds —

AZ QUOTES

• 创建版本库

- 版本库又名仓库, 英文名 repository (repo)
- 创建后的[工作区](#)的所有文件都可以被Git管理起来, 每个文件的修改、删除, Git都能跟踪, 以便任何时刻都可以追踪历史, 或者在将来某个时刻可以“还原”。

```
$ git init
```

- 工作区中多出一个 [git](#) 的目录, 这个目录是Git来跟踪管理版本库的, 没事千万不要手动修改这个目录里面的文件, 不然改乱了, 就把 Git仓库给破坏了。

```
~/Documents/git_test|=> git init
```

提示: 使用 'master' 作为初始分支的名称。这个默认分支名称可能会更改。要在新仓库中配置使用初始分支名, 并消除这条警告, 请执行:

提示:

提示: `git config --global init.defaultBranch <名称>`

提示:

提示: 除了 'master' 之外, 通常选定的名字有 'main'、'trunk' 和 'development'。

提示: 可以通过以下命令重命名刚创建的分支:

提示:

提示: `git branch -m <name>`

已初始化空的 Git 仓库于 /Users/herb/Documents/git_test/.git/

```
~/Documents/git_test|master = []
```

• 把文件添加到版本库

- 所有的版本控制系统，包括Git，其实只能跟踪纯文本文件的改动，比如TXT文件，网页，所有的程序代码等等。
- 图片、视频，以及Microsoft的Word格式等这些二进制文件，没法跟踪文件内部的变化。
- 第0步，有事没事都可以用 `git status` 查看在当前版本库的状态

```
$ git status
```

- 第一步，用命令 `git add` 告诉Git，把文件添加到版本库里：

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

- 第二步，用命令 `git commit` 告诉Git，把所有文件提交：

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

- 尝试修改 `readme.txt` 文件后，看一下文件是如何被记录下修改的？

```
$ git diff readme.txt
```

- 表示第 1 个文件
- + 表示第 2 个文件
- @@ -1,2 +1,2 @@ 表示比较的区块：
 - 第 1 个文件的第 1 行起的连续 2 行
 - 第 2 个文件的第 1 行起的连续 2 行

```
$ git add readme.txt  
$ git commit -m "fix bug: association, add a"
```

- 在你的工作区内新建一个 `readme.txt`，内容如下：

```
ICTP-AP operates in affiliation to UCAS.  
Taiji Lab is systematic project, not simple scientific laboratory.
```

- 修改 `readme.txt` 的内容如下：

```
ICTP-AP operates in association with UCAS.  
Taiji Lab is a systematic project, not a simple scientific  
laboratory.
```

```
diff --git a/readme.txt b/readme.txt  
index f4c6252..050e9ad 100644  
--- a/readme.txt  
+++ b/readme.txt  
@@ -1,2 +1,2 @@  
-ICTP-AP operates in affiliation to UCAS.  
-Taiji Lab is systematic project, not simple scientific laboratory.  
+ICTP-AP operates in association with UCAS.  
+Taiji Lab is a systematic project, not a simple scientific laboratory.  
(END)
```

- `.gitignore` 文件可以控制工作区内不跟踪哪些文件

• 版本回退

- 再一次修改 `readme.txt` 后，把修改提交一次到版本库

```
$ git add readme.txt  
$ git commit -m "append collaboration"
```

- 在Git中，用 `git log` 查看历史记录（版本号）

```
$ git log --pretty=oneline
```

- `HEAD`表示当前版本，上一个版本就是`HEAD^`上上一个就是`HEAD^^`

- `git reset` 回退到某版本：

```
$ git reset --hard HEAD^          # 回退到上一个版本  
$ git reset --hard <commit id>    # 跳转到某一个版本
```

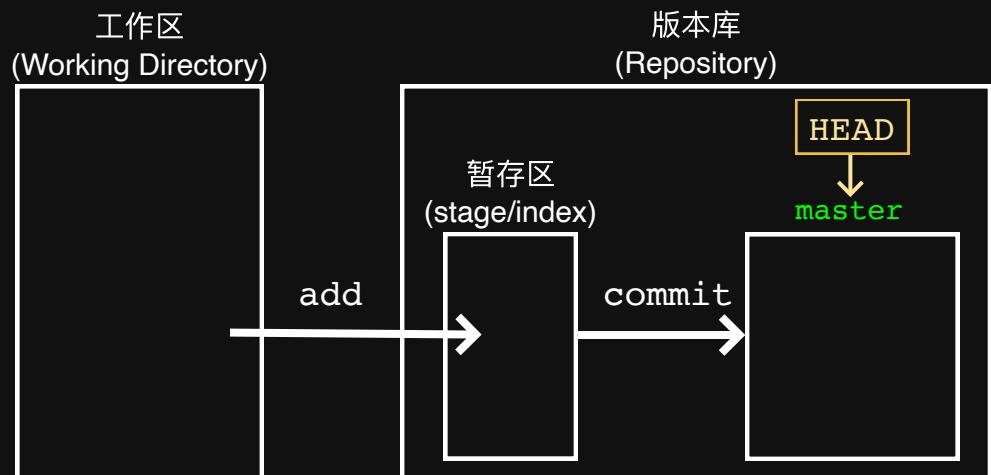
- 找不到历史的 `<commit id>` 怎么办？

```
$ git reflog      # 记录了工作区里的每一次git命令
```

- 修改 `readme.txt` 的内容如下：

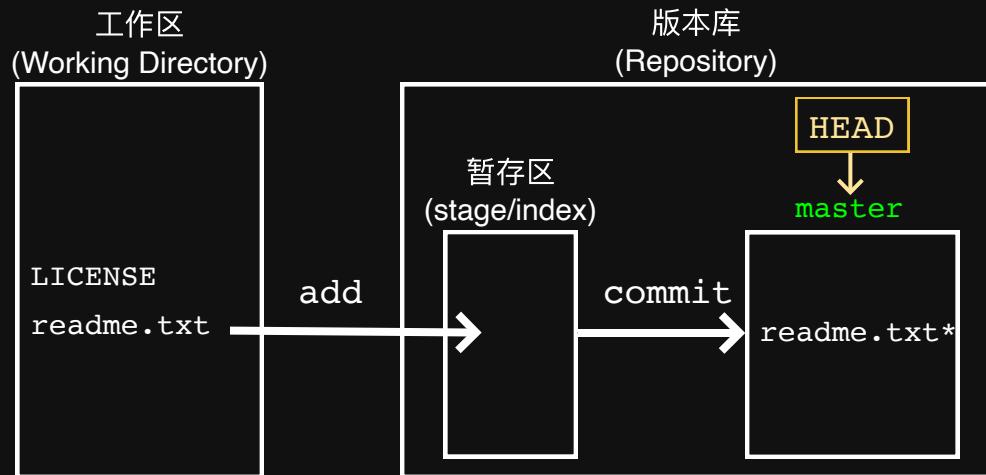
ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

```
$ cat readme.txt          # 看看内容是哪个版本  
$ git log                  # 看看Git在哪个历史版本  
$ ls -lht                  # 实际上对文件是修改的
```



• 工作区和暂存区

- 第一步: `git add` 实际上就是把文件修改添加到暂存区;
- 第二步: `git commit` 实际上就是把暂存区的所有内容提交到当前分支。



- 先对 `readme.txt` 做个修改, 比如加上一行内容:

```
ICTP-AP operates in affiliation to UCAS, fostering collaboration.  
Taiji Lab is a systematic project, not a simple scientific  
laboratory.  
Git has a mutable index called stage.
```

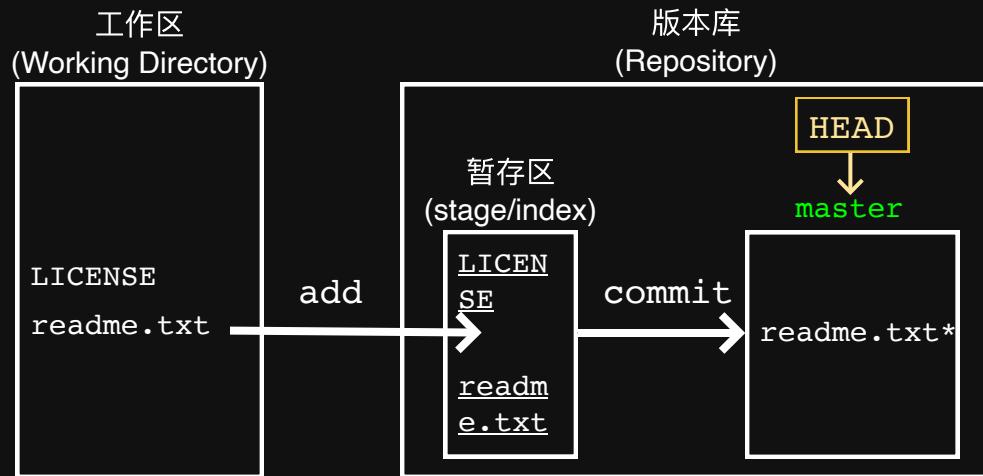
- 然后, 在工作区新增一个 `LICENSE` 文本文件 (内容随便写)。

```
$ git status
```

```
~/Documents/git_test2|master> ➜ git status  
位于分支 master  
尚未暂存以备提交的变更：  
  (使用 "git add <文件>..." 更新要提交的内容)  
  (使用 "git restore <文件>..." 丢弃工作区的改动)  
    修改：      readme.txt  
  
未跟踪的文件：  
  (使用 "git add <文件>..." 以包含要提交的内容)  
    LICENSE  
  
修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
```

• 工作区和暂存区

- 第一步: `git add` 实际上就是把文件修改添加到暂存区;
- 第二步: `git commit` 实际上就是把暂存区的所有内容提交到当前分支。



- 先对 `readme.txt` 做个修改, 比如加上一行内容:

```
ICTP-AP operates in affiliation to UCAS, fostering collaboration.  
Taiji Lab is a systematic project, not a simple scientific  
laboratory.  
Git has a mutable index called stage.
```

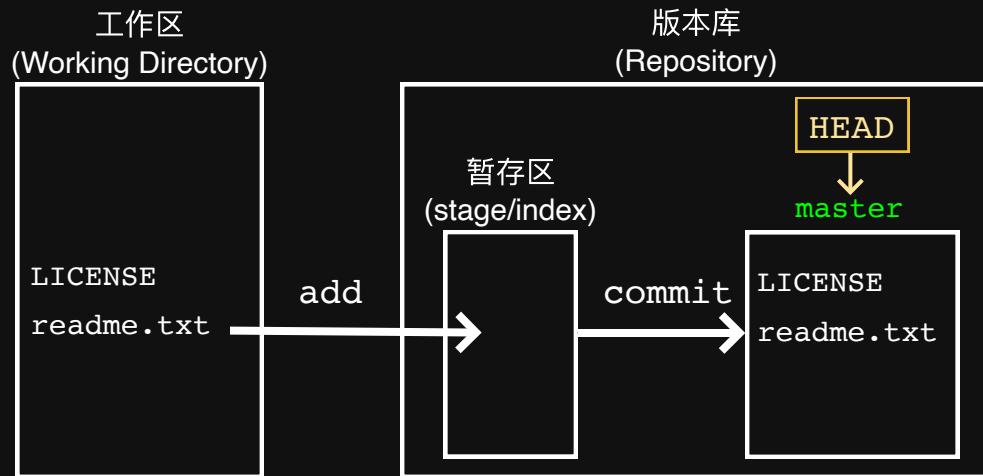
- 然后, 在工作区新增一个 `LICENSE` 文本文件 (内容随便写)。

```
$ git status  
$ git add readme.txt LICENSE  
$ git status
```

```
~/Documents/git_test2|master> git status  
位于分支 master  
要提交的变更:  
(使用 "git restore --staged <文件>..." 以取消暂存)  
新文件: LICENSE  
修改: readme.txt
```

• 工作区和暂存区

- 第一步: `git add` 实际上就是把文件修改添加到暂存区;
- 第二步: `git commit` 实际上就是把暂存区的所有内容提交到当前分支。



- 先对 `readme.txt` 做个修改, 比如加上一行内容:

```
ICTP-AP operates in affiliation to UCAS, fostering collaboration.  
Taiji Lab is a systematic project, not a simple scientific  
laboratory.  
Git has a mutable index called stage.
```

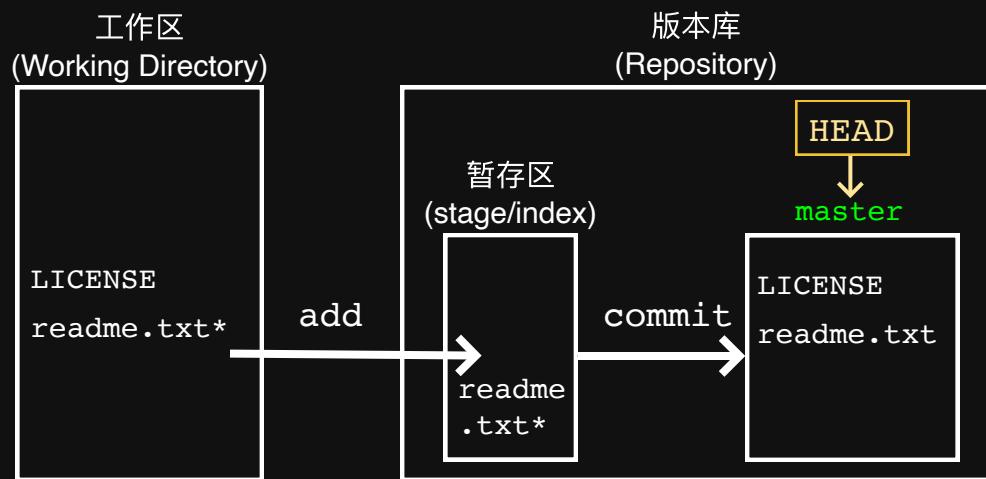
- 然后, 在工作区新增一个 LICENSE 文本文件 (内容随便写)。

```
$ git status  
$ git add readme.txt LICENSE  
$ git status  
$ git commit -m "understand how stage works"  
$ git status
```

```
~/Documents/git_test2|master => git status  
位于分支 master  
无文件要提交, 干净的工作区
```

• 管理修改

- Git 跟踪并管理的是修改，而非文件。



- 再对 `readme.txt` 做个修改，比如加上一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

Git has a mutable index called stage.

Git tracks changes.

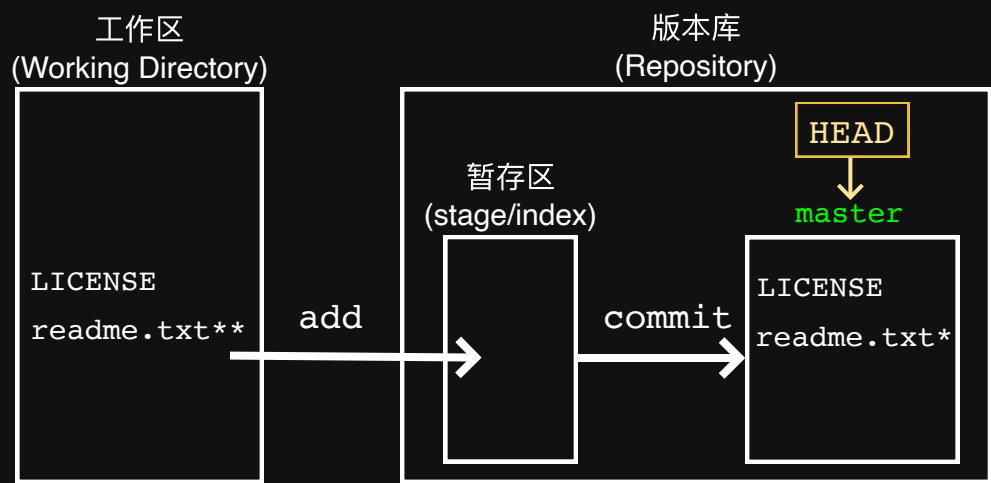
- 查看把修改加入到暂存区后的状态

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

```
~/Documents/git_test2|master> ➔ git status  
位于分支 master  
要提交的变更：  
(使用 "git restore --staged <文件>..." 以取消暂存)  
    修改：      readme.txt
```

• 管理修改

- Git 跟踪并管理的是修改，而非文件。



- 然后再对 `readme.txt` 做个修改，比如修改最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

Git has a mutable index called stage.

Git tracks changes of files.

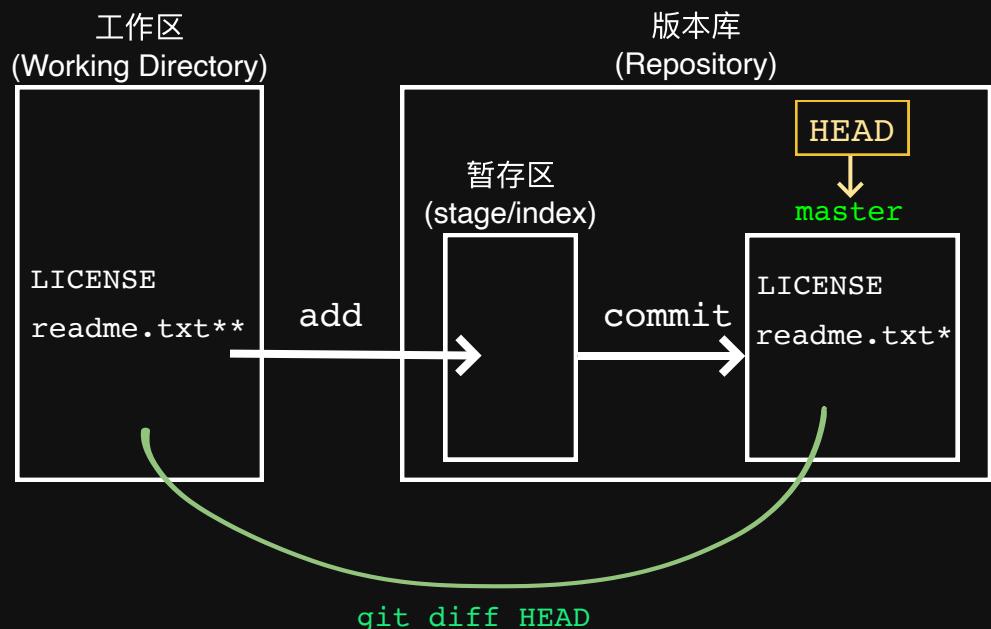
- 查看状态，并直接提交

```
$ git status  
$ git commit -m "Add: Git tracks changes"  
$ git status
```

```
~/Documents/git_test2|master> git status  
位于分支 master  
要提交的变更：  
  (使用 "git restore --staged <文件>..." 以取消暂存)  
    修改： readme.txt  
  
尚未暂存以备提交的变更：  
  (使用 "git add <文件>..." 更新要提交的内容)  
  (使用 "git restore <文件>..." 丢弃工作区的改动)  
    修改： readme.txt  
  
~/Documents/git_test2|master> git commit -m "Add: Git tracks changes"  
[master a53407b] Add: Git tracks changes  
  1 file changed, 1 insertion(+)  
~/Documents/git_test2|master> git status  
位于分支 master  
尚未暂存以备提交的变更：  
  (使用 "git add <文件>..." 更新要提交的内容)  
  (使用 "git restore <文件>..." 丢弃工作区的改动)  
    修改： readme.txt  
  
修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
```

• 管理修改

- Git 跟踪并管理的是修改，而非文件。



- 然后再对 `readme.txt` 做个修改，比如修改最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific laboratory.

Git has a mutable index called stage.

Git tracks changes of files.

- 查看状态，并直接提交

```
$ git status  
$ git commit -m "Add: Git tracks changes"  
$ git status
```

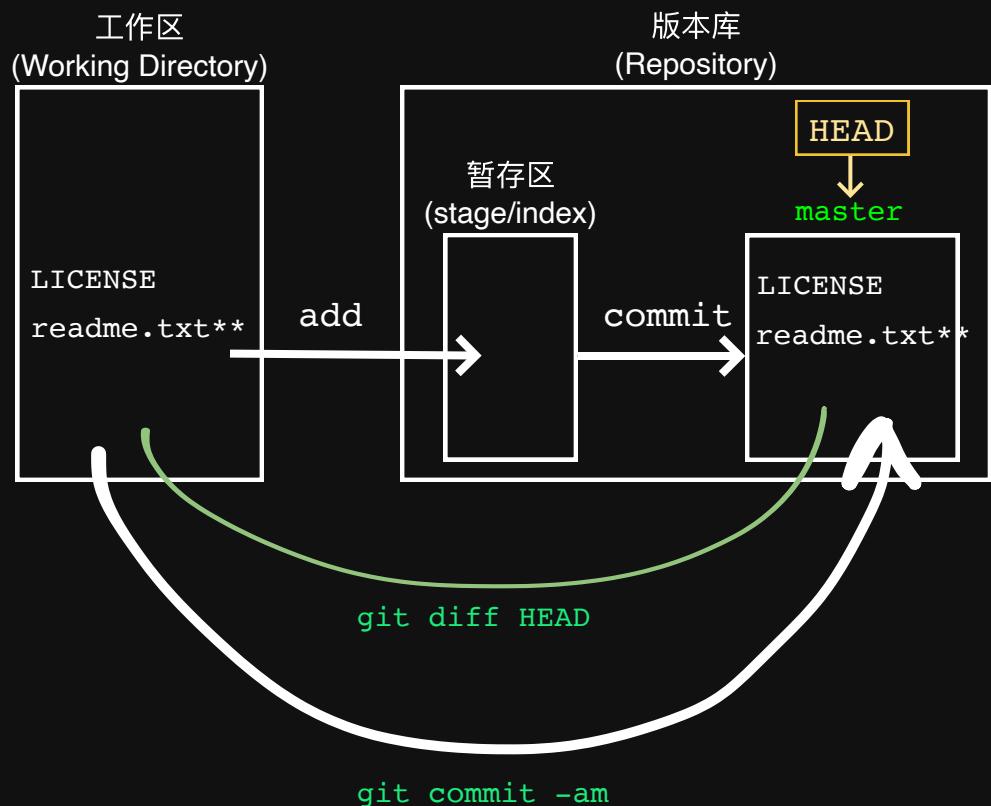
- 用 `git diff HEAD -- readme.txt` 命令可以查看工作区和版本库里面最新版本的区别：

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

```
diff --git a/readme.txt b/readme.txt  
index cede272..dc54cab 100644  
--- a/readme.txt  
+++ b/readme.txt  
@@ -1,4 +1,4 @@  
 ICTP-AP operates in association with UCAS, fostering collaboration.  
 Taiji Lab is a systematic project, not a simple scientific laboratory.  
 Git has a mutable index called stage.  
-Git tracks changes.  
+Git tracks changes of files.  
(END)
```

• 管理修改

- Git 跟踪并管理的是修改，而非文件。



- 然后再对 `readme.txt` 做个修改，比如修改最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific laboratory.

Git has a mutable index called stage.

Git tracks changes of files.

- 查看状态，并直接提交

```
$ git status  
$ git commit -m "Add: Git tracks changes"  
$ git status
```

- 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：

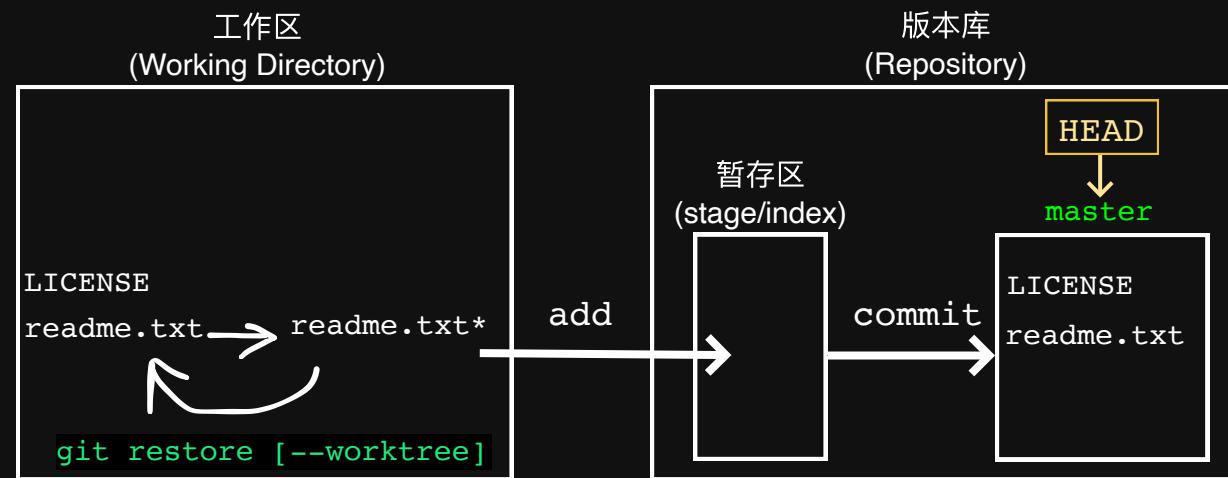
```
$ git diff HEAD -- readme.txt
```

```
diff --git a/readme.txt b/readme.txt  
index cede272..dc54cab 100644  
--- a/readme.txt  
+++ b/readme.txt  
@@ -1,4 +1,4 @@  
 ICTP-AP operates in association with UCAS, fostering collaboration.  
 Taiji Lab is a systematic project, not a simple scientific laboratory.  
 Git has a mutable index called stage.  
-Git tracks changes.  
+Git tracks changes of files.  
(END)
```

```
$ git commit -am "Append: of files"
```

• 撤销修改

- Git 跟踪并管理的是修改，而非文件。



- 再对 `readme.txt` 做个错误修改，比如添加最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

Git has a mutable index called `stage`.

Git tracks changes of files.

My stupid boss ...

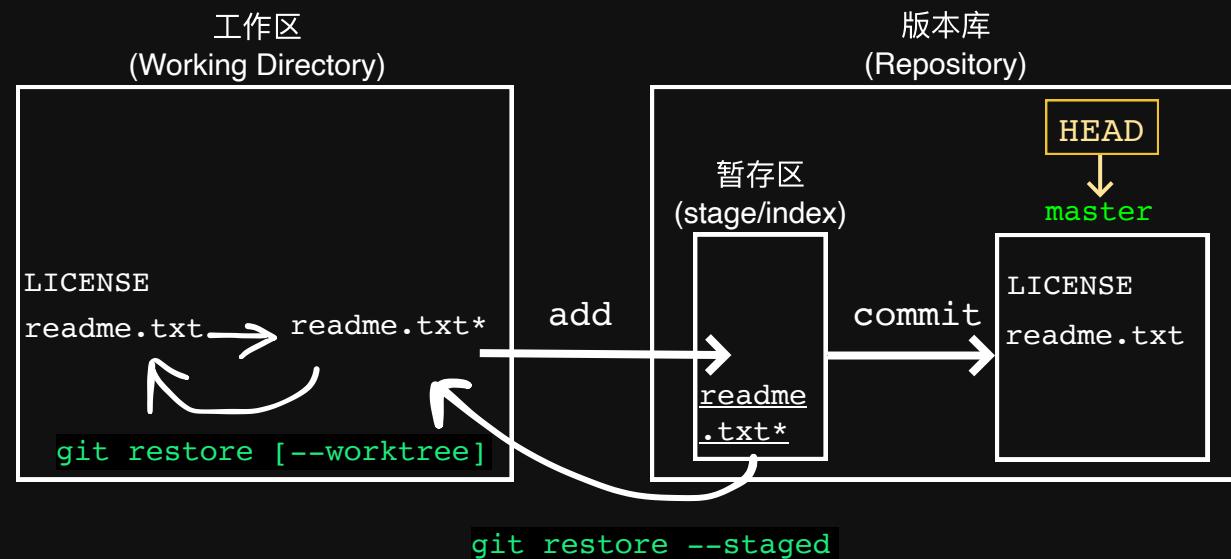
- 查看状态，并丢弃工作区的改动

```
$ git status  
$ git restore readme.txt
```

```
~/Documents/git_test2|master> git status  
位于分支 master  
尚未暂存以备提交的变更：  
  (使用 "git add <文件>..." 更新要提交的内容)  
  (使用 "git restore <文件>..." 丢弃工作区的改动)  
    修改：      readme.txt  
  
修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
```

• 撤销修改

- Git 跟踪并管理的是修改，而非文件。



- 再对 `readme.txt` 做个错误修改，比如添加最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

Git has a mutable index called `stage`.

Git tracks changes of files.

My stupid boss ...

- 存到暂存区后，查看状态，再取消暂存，并工作区里丢弃该修改。

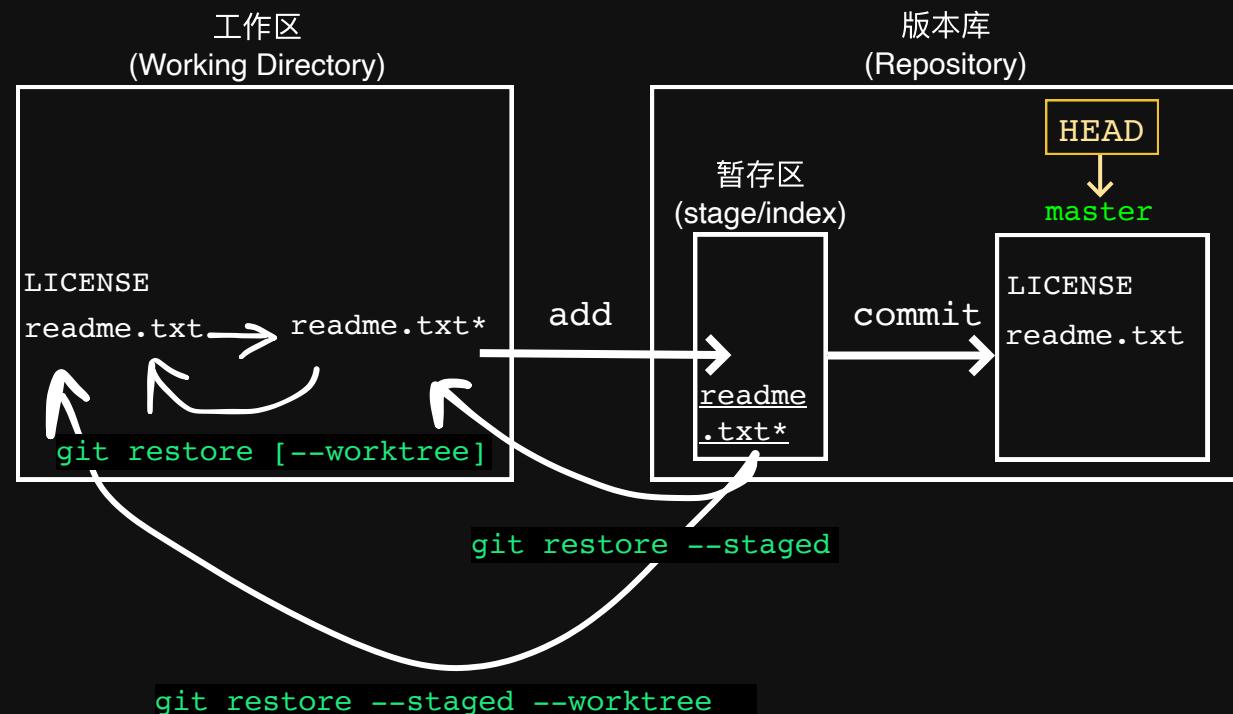
```
$ git add readme.txt  
$ git status
```

```
~/Documents/git_test2|master> => git status  
位于分支 master  
要提交的变更：  
(使用 "git restore --staged <文件>..." 以取消暂存)  
修改：      readme.txt
```

```
$ git restore --staged readme.txt  
$ git status  
$ git restore readme.txt
```

• 撤销修改

- Git 跟踪并管理的是修改，而非文件。



- 再对 `readme.txt` 做个错误修改，比如添加最后一行内容：

ICTP-AP operates **in** affiliation to UCAS, fostering collaboration.
Taiji Lab is a systematic project, not a simple scientific
laboratory.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss ...

- 存到暂存区后，查看状态，再取消暂存，并工作区里丢弃该修改。

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

```
~/Documents/git_test2|master> git status  
位于分支 master  
要提交的变更：  
(使用 "git restore --staged <文件>..." 以取消暂存)  
修改：      readme.txt
```

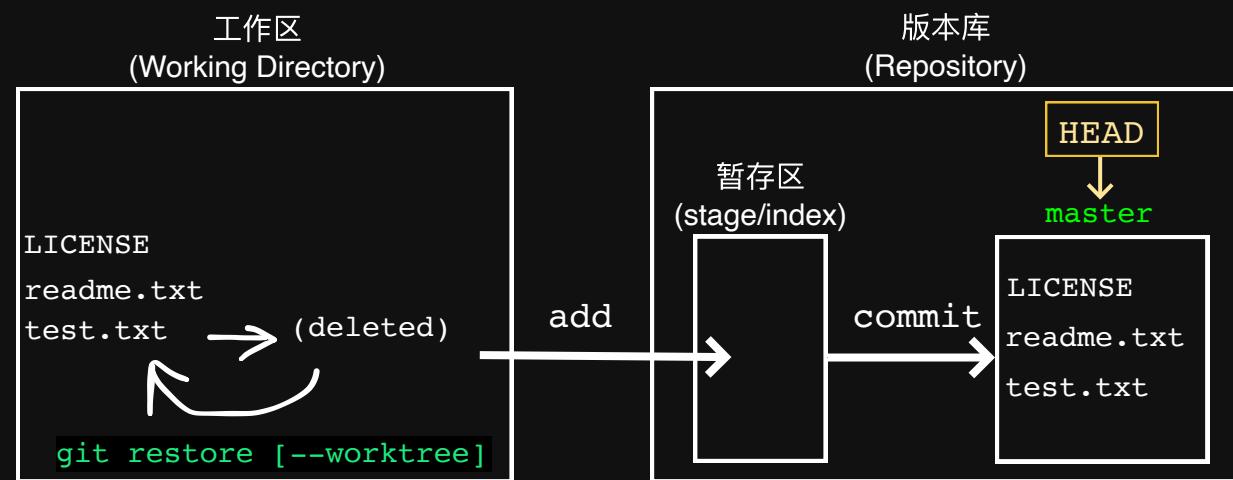
```
$ git restore --staged readme.txt  
$ git status  
$ git restore readme.txt
```

- 当然也有一步到位的办法，存到暂存区后，直接在暂存区和工作区里丢弃该修改。

```
$ git restore --staged --worktree readme.txt
```

• 删除文件

- 在Git中，删除也是一个修改操作。



- 先添加一个新文件 `test.txt` 到Git并且提交：

```
$ git add test.txt
$ git commit -m "add test.txt"
$ rm test.txt
$ git status

~/Documents/git_test2|master> git status
位于分支 master
尚未暂存以备提交的变更：
  (使用 "git add/rm <文件>..." 更新要提交的内容)
  (使用 "git restore <文件>..." 丢弃工作区的改动)
    删除:      test.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
```

- 情况一：确实要从版本库中删除 `test.txt` 文件，直接 `add` 和 `commit` 即可

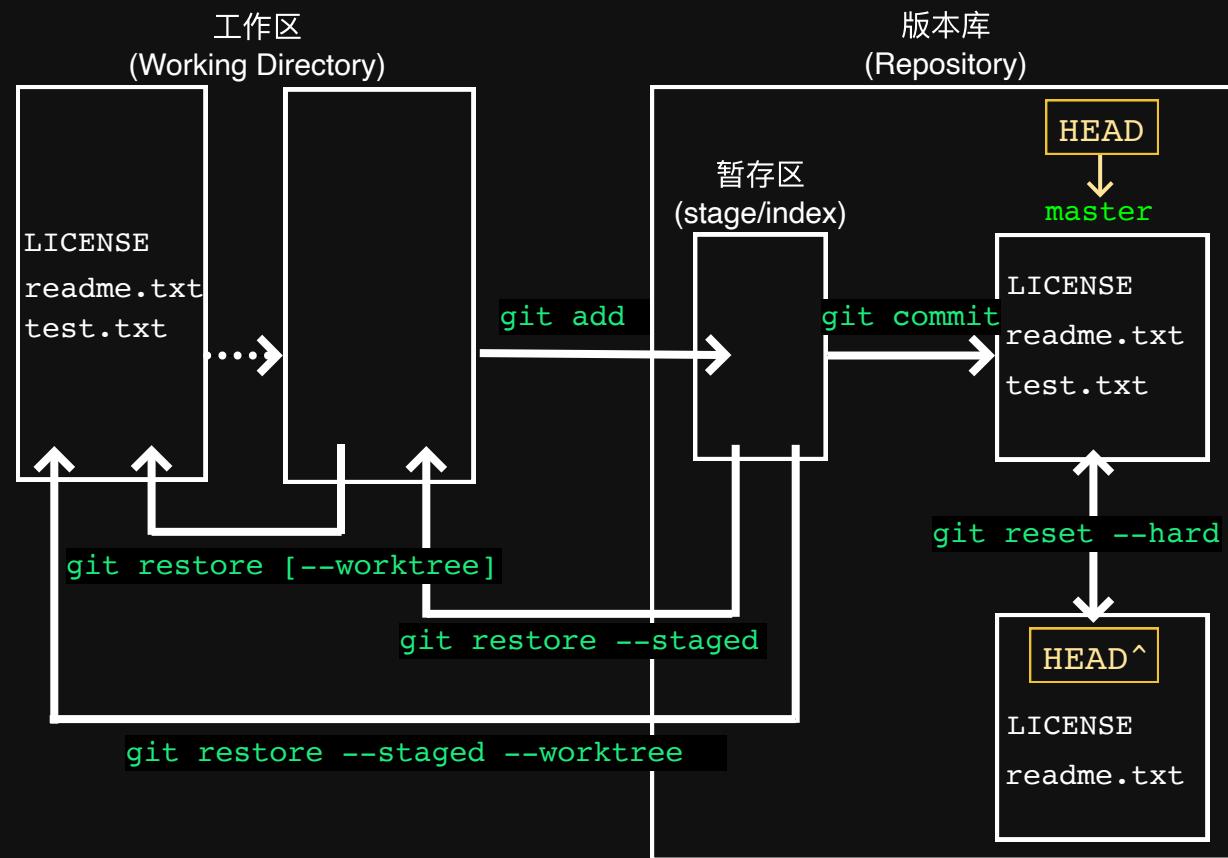
```
$ git add .
$ git commit -m "delete test.txt"
```

- 情况二：把误删的文件恢复到最新版本

```
$ git restore .
```

• 小结

- 一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。

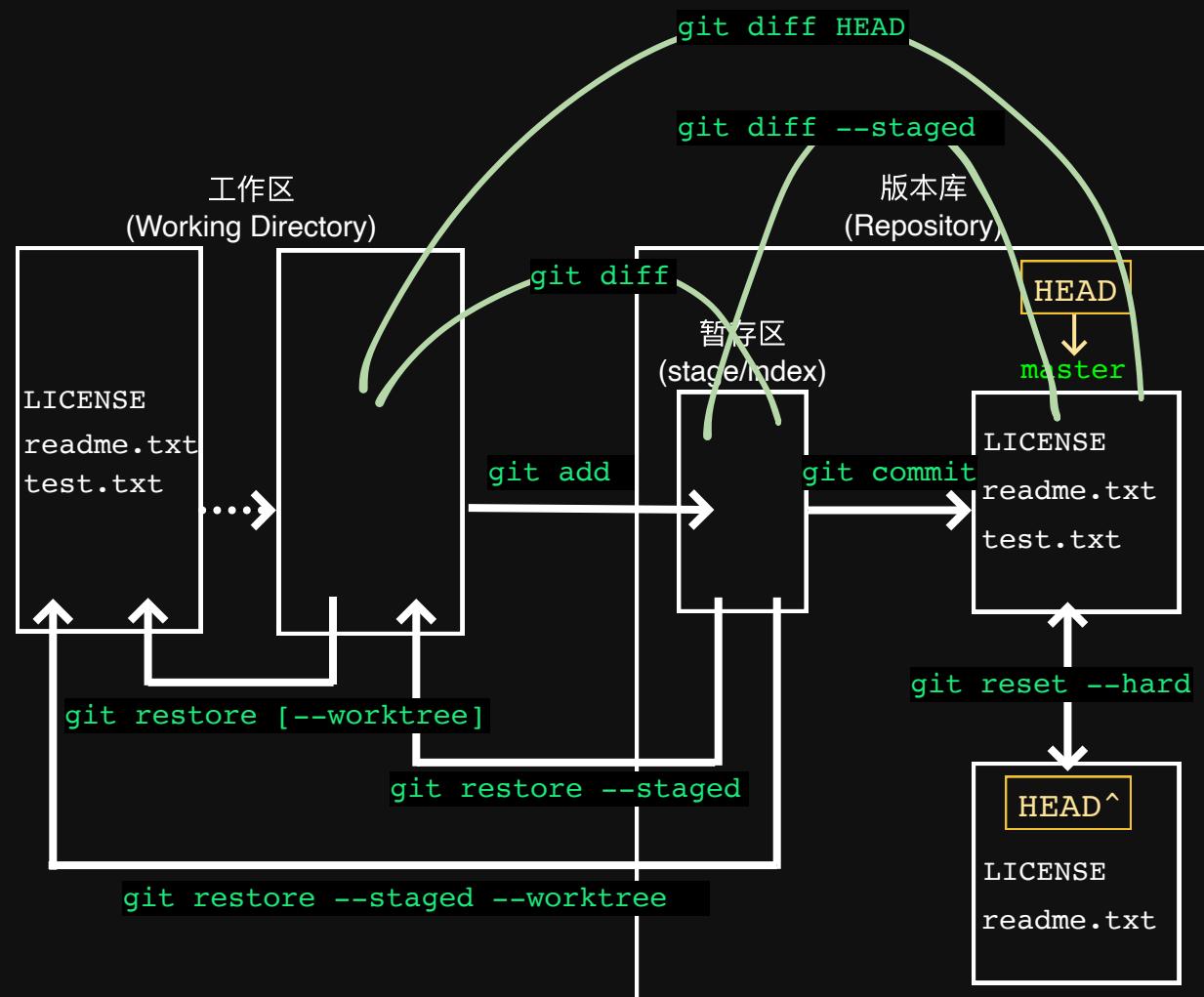


• Git 暂存区的作用

- 对提交(commit)提供防错机制、补救机制以及灵活机制。
- 原子性提交：尽量保证提交的内容是完整且正确的，注重的是结果，操作过程可以是灵活的，应交由用户把控。

• 小结

- 一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。



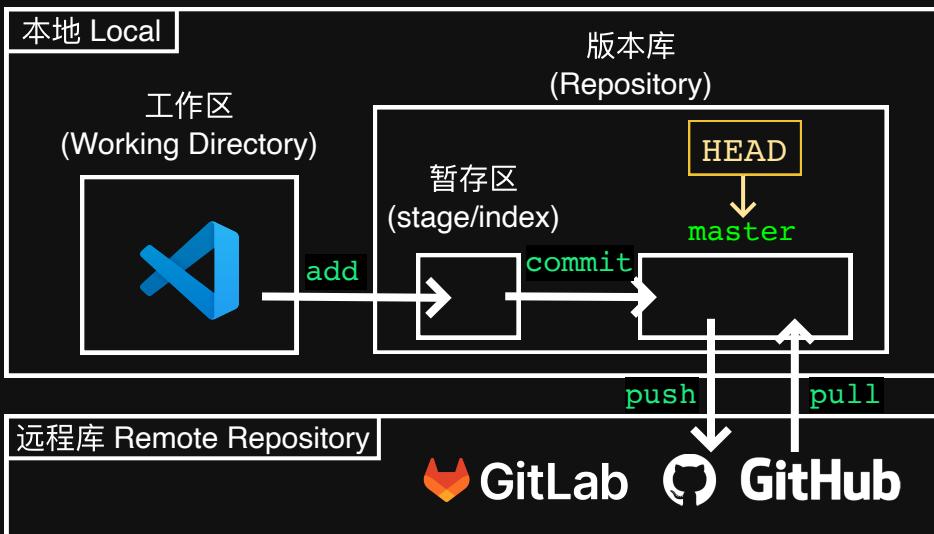
• Git 暂存区的作用

- 对提交(commit)提供 **防错机制**、**补救机制** 以及 **灵活机制**。
- 原子性提交：尽量保证提交的内容是**完整**且**正确的**，注重的是**结果**，操作过程可以是**灵活**的，应交由用户把控。
- 防错机制：
 - 通过预览功能检查完整性
\$ git status -s
 - 通过预览功能检查正确性
\$ git diff
 - 让用户养成二次检查的习惯
- 补救机制：
 - 用户提交前补救错误
 - 在本地已经提交，未PUSH远程仓库时，建议修补历史提交
\$ git commit --amend = git reset --soft head~1 + git commit
- 灵活机制：
 - 通过暂存区，可以额外支持分阶段标记待提交内容

Git为什么要设计暂存区？ - 知乎

• 远程仓库

- GitHub : 提供Git仓库托管服务
 - <https://github.com/>
- 请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以需要设置一下。可以参考 [官方文档](#)，或 [X](#) 和 [X](#)。
- 确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。



git	GitHub
1. It is a software	1. It is a service
2. It is installed locally on the system	2. It is hosted on Web
3. It is a command line tool	3. It provides a graphical interface
4. It is a tool to manage different versions of edits, made to files in a git repository	4. It is a space to upload a copy of the Git repository
5. It provides functionalities like Version Control System Source Code Management	5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features

• 添加远程库

- 你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。
- 在 Github 网站上新建一个远程仓库，根据GitHub的提示：

```
$ git remote add origin git@github.com:<你的用户名>/<仓库名称>.git  
$ git remote -v # 查看该本地库关联的远程库的信息  
$ git remote rm origin # 解除本地和远程的绑定关系
```

- 添加后，远程库的名字就是 origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

```
# 第一次推送master分支的所有内容;  
$ git push -u origin master  
# 把本地master分支的最新修改推送至GitHub  
$ git push origin master
```

- 出现如下报错，一般就是GitHub还不知道你的SSH公钥的缘故：

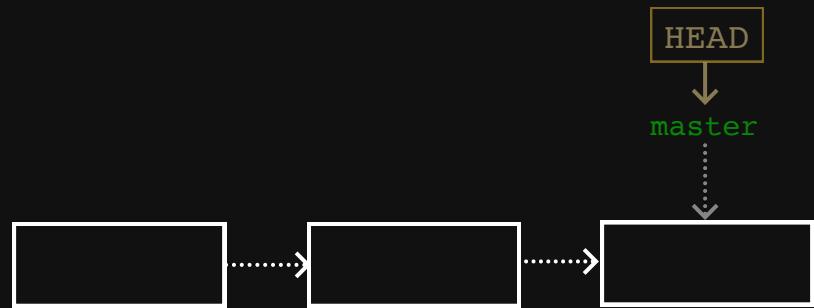
```
~/Documents/git_test2|master => git push -u origin master  
The authenticity of host 'github.com (20.205.243.166)' can't be established.  
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbPZisF/zLDA0zPMSSvHdkr4UvC0qU.  
This key is not known by any other names  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.  
git@github.com: Permission denied (publickey).  
致命错误：无法读取远程仓库。
```

请确认您有正确的访问权限并且仓库存在。

• 从远程库克隆（略）

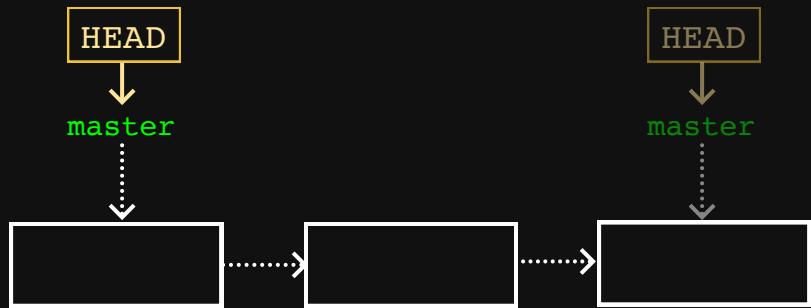
• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。



• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。

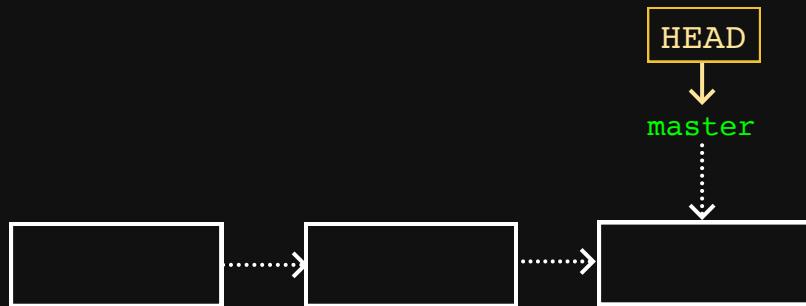


• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。

- 创建新的分支 **dev**，即新建了一个指针叫 dev，指向 master 相同的提交，再把HEAD指向dev，就表示当前分支在dev上。

```
$ git switch -c dev      # 创建并切换分支到 dev
```



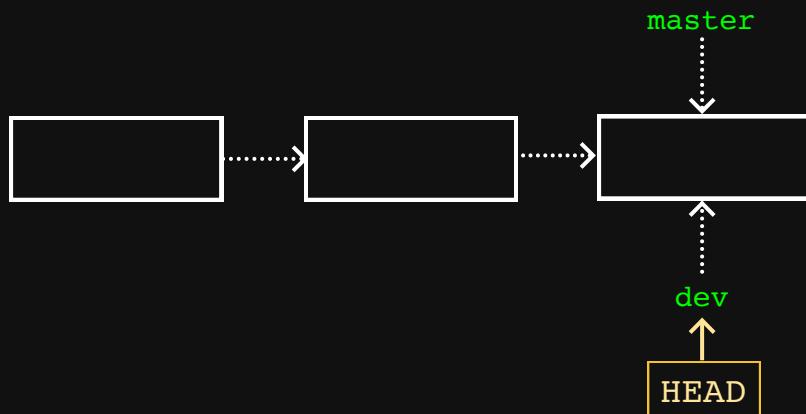
• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。

- 创建新的分支 **dev**，即新建了一个指针叫 dev，指向 master 相同的提交，再把HEAD指向dev，就表示当前分支在dev上。

```
$ git switch -c dev      # 创建并切换分支到 dev  
$ git branch              # 查看当前分支
```

- **git branch** 命令会列出所有分支，当前分支前面会标一个*号。

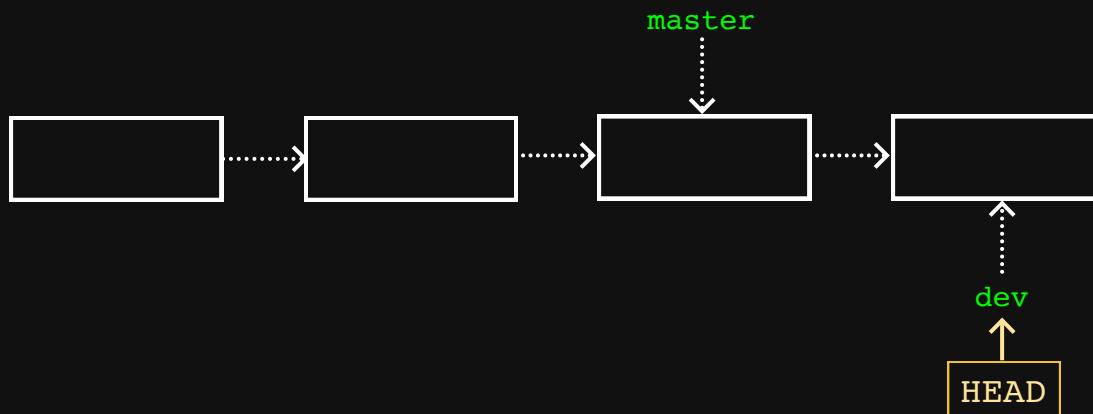


• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。

- 我们在 dev 分支上正常提交，对readme.txt做个修改，加上一行：

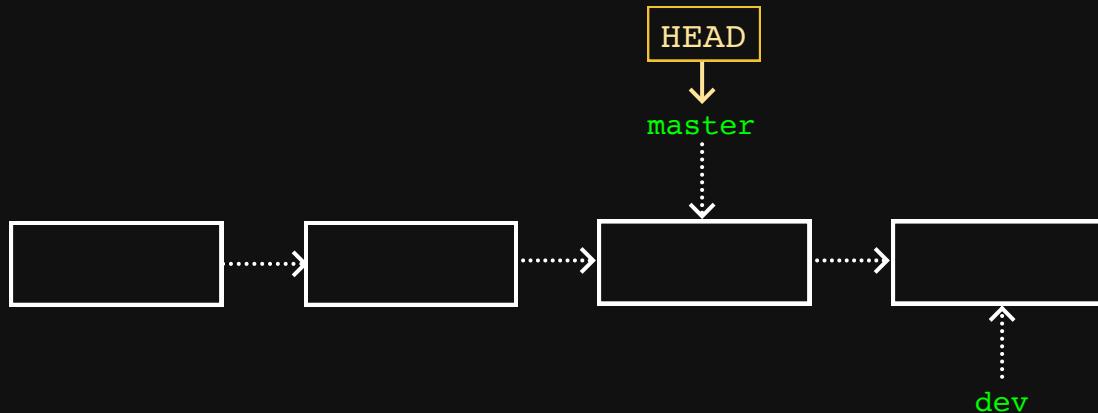
```
$ echo "Creating a new branch is quick." >> readme.txt
$ git commit -am "branch test"
$ git log
$ cat readme.txt
```



• 创建与合并分支

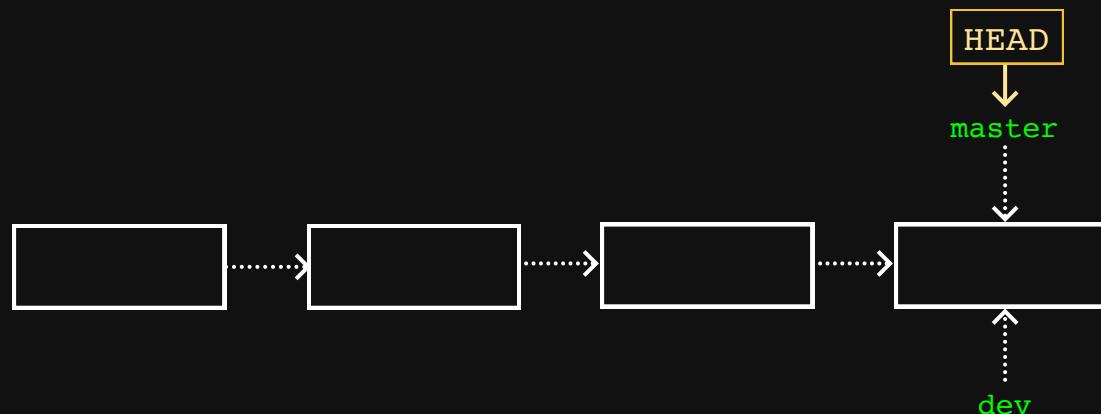
- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。
- 现在，dev 分支的工作完成，我们就可以切换回 master 分支：

```
$ git switch master  
$ git log  
$ cat readme.txt
```



• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。
- 我们把 dev 分支的工作成果**合并到当前的 master 分支上**：



```
$ git merge dev  
$ git log  
$ cat readme.txt
```

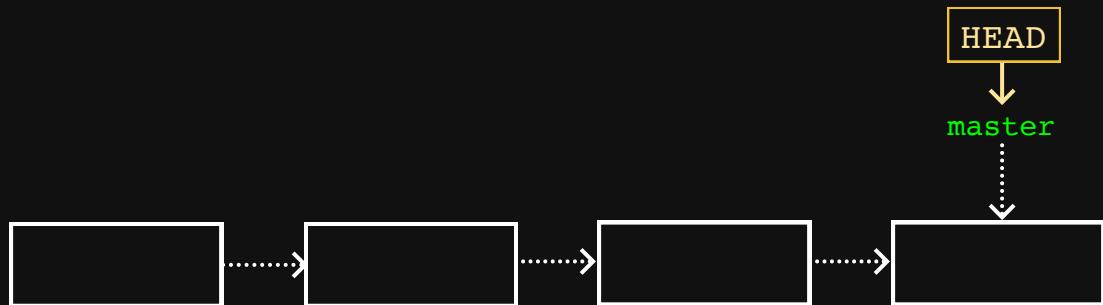
```
~/Documents/git_test2|master ➔ git merge dev  
更新 3d2a769..4185701  
Fast-forward  
 readme.txt | 1 +  
 1 file changed, 1 insertion(+)
```

- 注意到上面的**Fast-forward**信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

• 创建与合并分支

- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。
- 截止到目前，只有一条时间线，在Git里，这个分支叫**主分支**，即**master** 分支。
- HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，**HEAD** 指向的就是**当前分支**。
- 合并完成后，就可以放心地删除dev分支了：

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



• 解决冲突

- 人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

- 准备新的 feature1 分支，继续我们的新分支开发：

```
$ git switch -c feature1
$ echo "Creating a new branch is quick AND simple." >> readme.txt
$ cat readme.txt
$ git commit -am "AND simple"
```

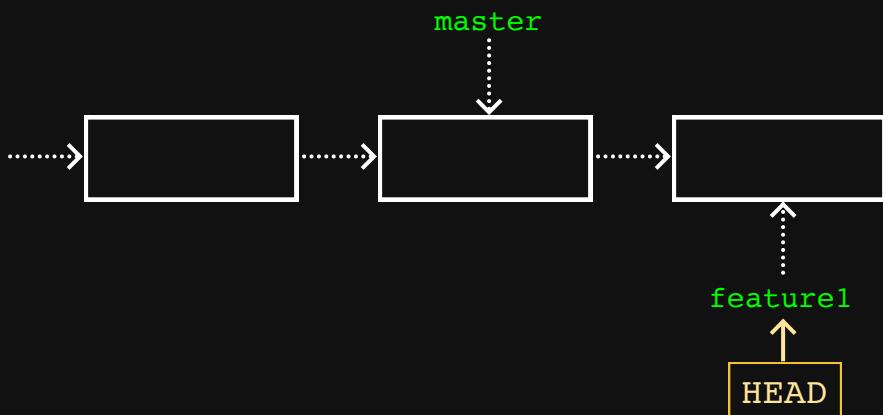


• 解决冲突

- 人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

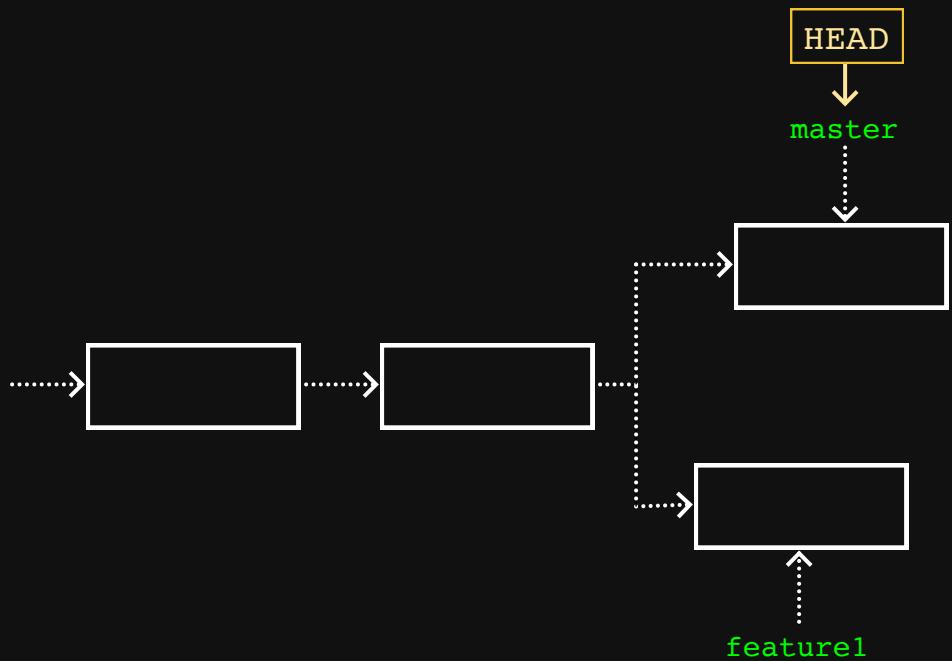
- 准备新的 feature1 分支，继续我们的新分支开发：

```
$ git switch -c feature1
$ echo "Creating a new branch is quick AND simple." >> readme.txt
$ cat readme.txt
$ git commit -am "AND simple"
```



• 解决冲突

- 人生不如意之事十之八九，合并分支往往也不是一帆风顺的。



- 然后我们切换到master分支：

```
$ git switch master
```

```
~/Documents/git_test2|feature1 => git switch master
切换到分支 'master'
您的分支领先 'origin/master' 共 1 个提交。
(使用 "git push" 来发布您的本地提交)
```

- 可以看到，Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。
- 在master分支上把readme.txt文件的最后一行按如下修改，并再一次提交。

```
Creating a new branch is quick & simple.
```

```
$ git commit -am "& simple"
```

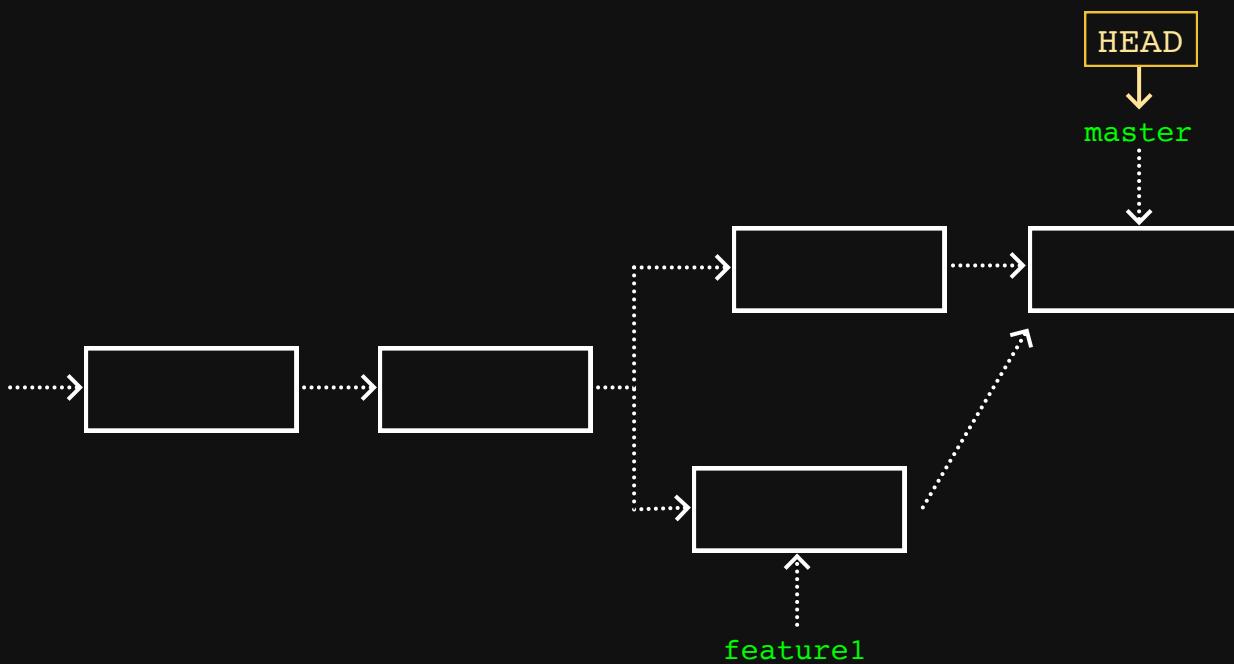
- 这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突：

```
$ git merge feature1
```

```
~/Documents/git_test2|master => git merge feature1
自动合并 readme.txt
冲突 (内容)：合并冲突于 readme.txt
自动合并失败，修正冲突然后提交修正的结果。
```

• 解决冲突

- 人生不如意之事十之八九，合并分支往往也不是一帆风顺的。



```
* cc0053c (HEAD -> master) conflict fixed
| \
| * 0e7c294 And simple
| * 9b11fd0 & simple
| /
| * 4185701 branch test
| * 3d2a769 (origin/master) add test.txt
| * e58f963 Append: of files
| * a53407b Add: Git tracks changes
| * cf597eb understand how stage works
| * fe0e670 append collaboration
| * c19452d fix bug: association, add a
| * 15453fd wrote a readme file
```

- 果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

```
$ git status
~/Documents/git_test2|master> git status
位于分支 master
您的分支领先 'origin/master' 共 2 个提交。
(使用 "git push" 来发布您的本地提交)

您有尚未合并的路径。
(解决冲突并运行 "git commit")
(使用 "git merge --abort" 终止合并)

未合并的路径：
(使用 "git add <文件>..." 标记解决方案)
双方修改: readme.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
```

```
$ vim readme.txt
```

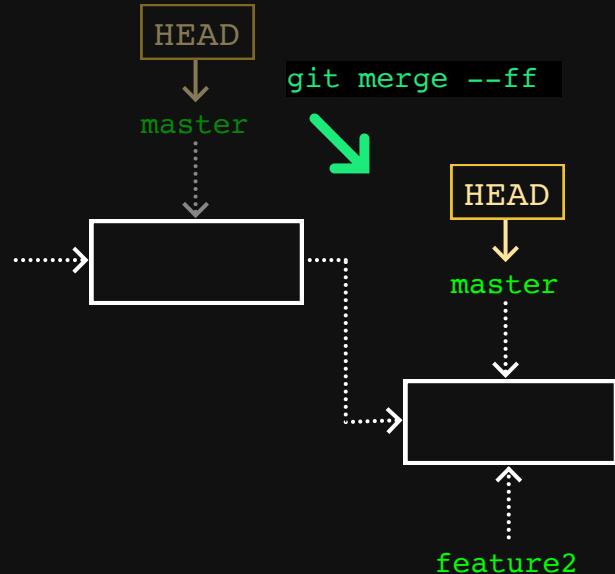
- Git用<<<<<，=====，>>>>>标记出不同分支的内容，我们把最后一行修改如下后保存，并再一次重新提交。

```
Creating a new branch is quick and simple.
```

```
$ git commit -am "conflict fixed"
$ git status
$ git log
$ git log --graph --pretty=oneline --abbrev-commit
$ git branch -d feature1
```

• 分支管理策略

- 通常，合并分支时，如果可能，Git会用Fast forward模式 (`--ff`)，但这种模式下，删除分支后，会丢掉分支信息。

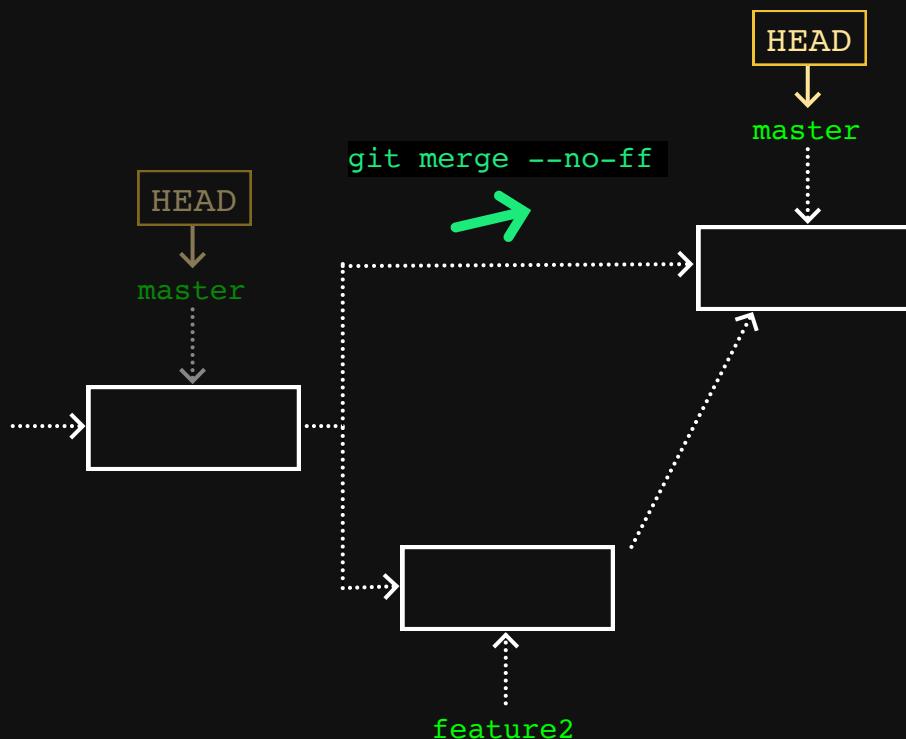


```
$ git switch -c feature2
$ echo "Try --ff and --no-ff here." >> readme.txt
$ git commit -am "try --ff"
$ git log
$ git switch master
$ git merge feature2 --ff
$ cat readme.txt
$ git log --graph --pretty=oneline --abbrev-commit
```

```
* fbd56d8 (HEAD -> master, feature2) try --ff
* cc0053c conflict fixed
| \
| * 9e0299d Add simple
| * 9e0299d & complex
|
| * 420079f branch test
| * 3a2a79f (tag:v0.0.1) add test.txt
| * 0507963 Append of file
| * 0507963 Add .gitignore changes
| * 0507963 understand how stage works
| * 0507963 append collaboration
| * 0507963 Fix bug: association with a
| * 1945376 write a readme file
```

• 分支管理策略

- 如果要强制禁用Fast forward模式 (`--no-ff`), Git就会在merge时生成一个新的commit, 这样, 从分支历史上就可以看出分支信息。



```
# 回退到 merge 之前的状态  
$ git reset HEAD~  
$ git status      # 暂存区为空  
$ git restore readme.txt  
  
# 再稍微修改下 feature2 中上一次commit的message内容  
$ git commit --amend  
$ git log
```

```
$ git switch master  
$ git merge feature2 --ff  
$ cat readme.txt  
$ git log --graph --pretty=oneline --abbrev-commit  
$ git branch -d feature2
```

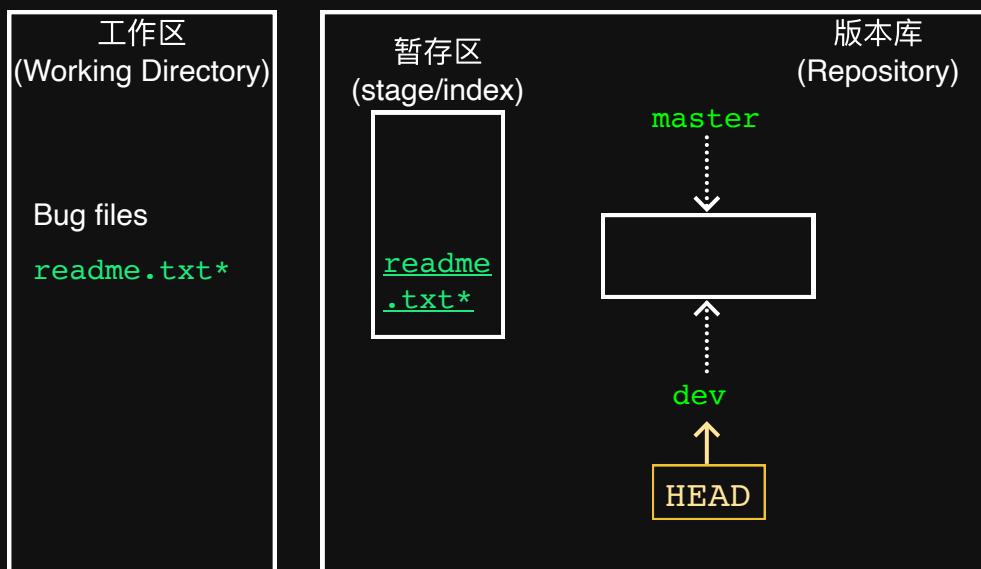
```
* c3da7e1 (HEAD -> master) Merge branch 'feature2' (--no-ff used)  
|\ \ * 242de06 (feature2) try --no-ff  
|\ /  
* cc0053c conflict fixed  
|\ \ * 4000000 (branch-test)  
|\ \ * 3000000 (branch-test) add README  
|\ \ * 2000000 Append .gitignore  
|\ \ * 1000000 Add .gitignore changes  
|\ \ * 0000000 understand how stage works  
|\ \ * f000000 append to .gitignore  
|\ \ * e000000 Fix bug in execution, add a  
|\ \ * d000000 WRITE a README file
```

• Bug分支

Git Stash: 临时保存和切换工作状态的利器

- 在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支。这个时候很有可能需要 `git stash`。
- Git Stash可以帮助我们在切换分支或保存未完成的工作时，临时保存当前的修改（工作区和暂存区），以便稍后重新应用。然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支 `issue-101` 来修复它，但是，等等，当前正在 `dev` 上进行的工作还没有提交：



```
# 构建一个未提交的 dev 分支
$ git switch -c dev
$ echo "Try stash on stage" >> readme.txt
$ git add readme.txt
$ echo "Try stash on worktree" >> readme.txt
$ git status

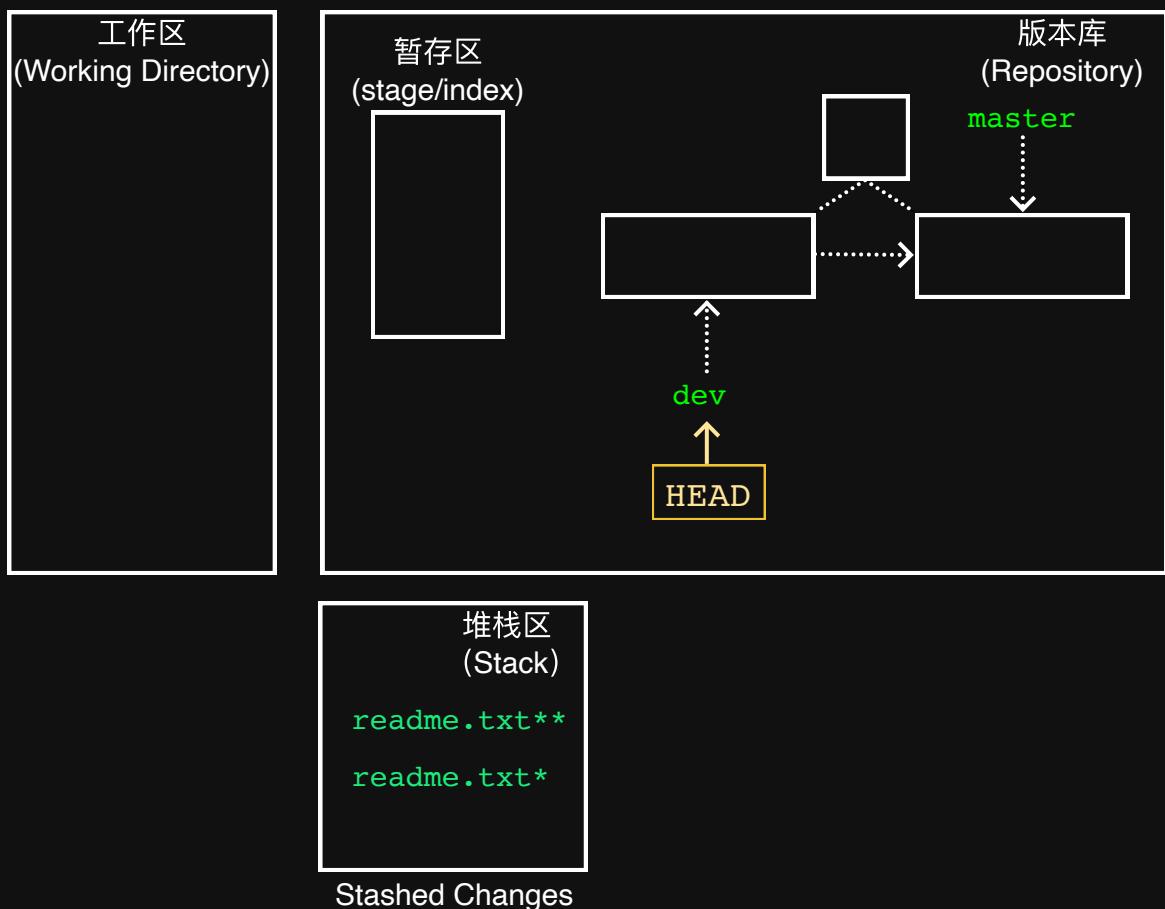
# 在不同分支上观察工作区和暂存区
$ git switch master
$ git status
```

• Bug分支

Git Stash: 临时保存和切换工作状态的利器

- 在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支。这个时候很有可能需要 `git stash`。
- Git Stash可以帮助我们在切换分支或保存未完成的工作时，临时保存当前的修改（工作区和暂存区），以便稍后重新应用。然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支 `issue-101` 来修复它，但是，等等，当前正在 `dev` 上进行的工作还没有提交：



```
$ git switch dev  
$ git stash      # 保存当前工作区和暂存区的修改，但不保存已提交的修改  
~/Documents/git_test2|dev> → git stash  
保存工作目录和索引状态 WIP on dev: c3da7e1 Merge branch 'feature2' (--no-ff used)  
$ git status  
$ git stash list      # 列出所有临时的存储记录
```

```
# 开始修复bug并合并到master主分支上  
$ git switch -c issue-101  
$ echo "Fix a bug" >> readme.txt  
$ git commit -am 'A bug is fixed'  
$ git switch master  
$ git merge issue-101 --no-ff -m "merged bug fix 101"  
$ git branch -d issue-101  
$ git log
```

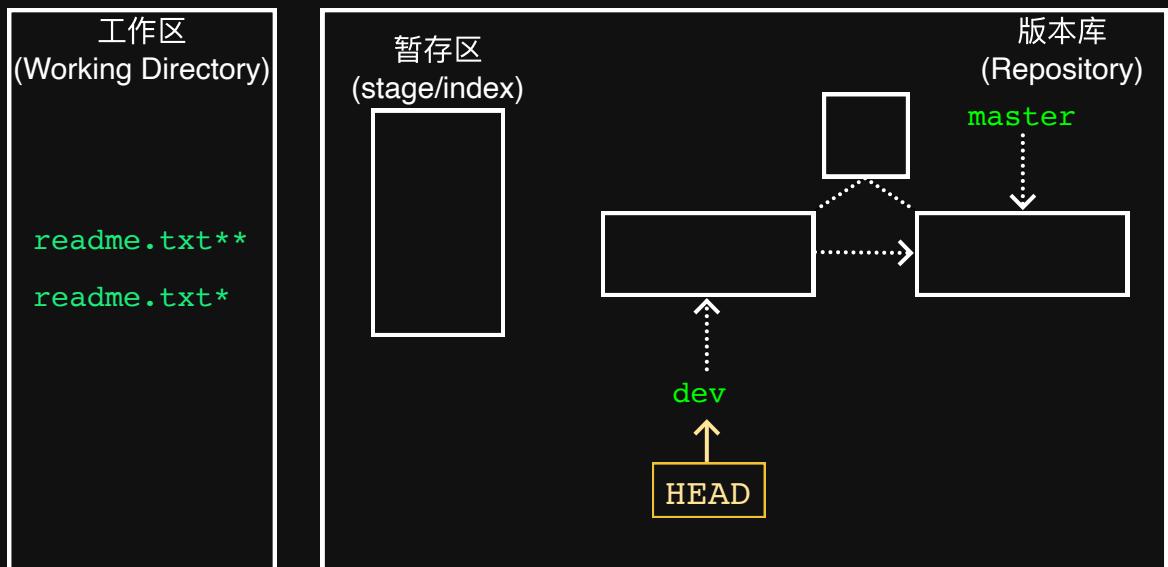
```
$ git switch dev  
$ git status
```

• Bug分支

Git Stash: 临时保存和切换工作状态的利器

- 在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支。这个时候很有可能需要 `git stash`。
- Git Stash可以帮助我们在切换分支或保存未完成的工作时，临时保存当前的修改（工作区和暂存区），以便稍后重新应用。然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支 `issue-101` 来修复它，但是，等等，当前正在 `dev` 上进行的工作还没有提交：



```
$ git stash pop # 恢复最近一个的同时把stash内容也删了  
$ git status  
$ cat readme.txt  
$ git stash list
```

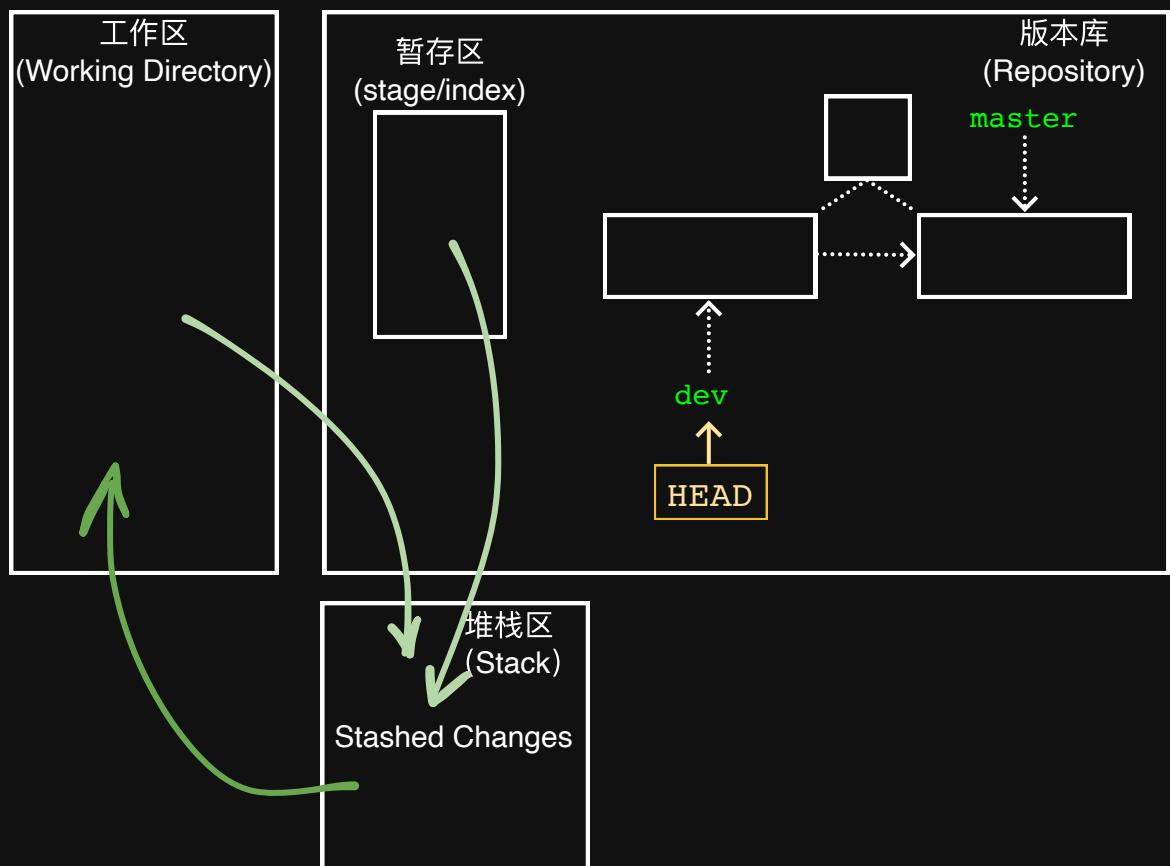
- 除了 `stash pop` 方式恢复之外，另一种方法是 `stash apply` 来恢复：

```
$ git stash save "try stash apply"  
$ git stash list  
$ git status  
$ cat readme.txt  
  
$ git stash apply # 恢复后，stash内容并不删除  
# git stash apply stash@{0} # 可以指定某一次内容来恢复  
$ git status  
$ cat readme.txt  
$ git stash list  
  
$ git stash drop stash@{0} # 删除存储的内容
```

• Bug分支

Git Stash: 临时保存和切换工作状态的利器

- 在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支。这个时候很有可能需要 `git stash`。
- Git Stash可以帮助我们在切换分支或保存未完成的工作时，临时保存当前的修改（工作区和暂存区），以便稍后重新应用。然后将临时分支删除。



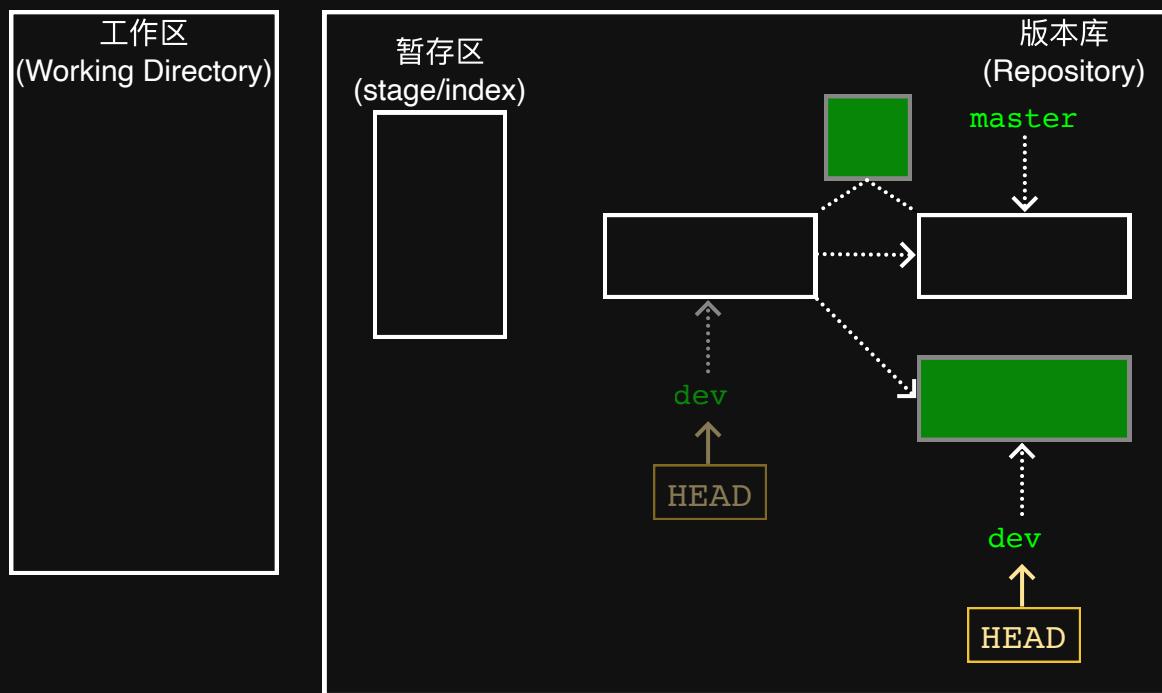
Git Stash的用途非常广泛，特别适用于以下几种常见场景：

- 分支切换**: 当你正在进行某个分支上的开发工作，但需要紧急修复其他分支的bug或进行其他任务时，你可以使用Git Stash来保存当前分支上的修改。这样，你可以切换到其他分支，完成紧急任务后再回到原来的分支，并通过恢复Stash来重新应用你之前保存的修改。
- 临时保存工作状态**: 有时候你可能需要中断当前的工作，但又不想提交未完成的修改。使用Git Stash可以将你的修改临时保存起来，以便稍后继续工作。这在你需要暂时切换到其他任务、参与会议或处理其他紧急问题时非常有用。
- 解决代码冲突**: 当你在合并分支或拉取远程更新时遇到代码冲突，你可以使用Git Stash来保存当前的修改，并将工作区恢复到干净的状态。然后你可以解决冲突，再重新应用你之前保存的修改。

• Bug分支

Git Stash: 临时保存和切换工作状态的利器

- 在master分支上修复了bug后，我们要想一想，dev分支是早期从master分支分出来的，所以，这个bug其实在当前dev分支上也存在。
- 那怎么在dev分支上修复同样的bug？重复操作一次，提交不就行了？
- 有木有更简单的方法？有！
- 为了方便操作，Git专门提供了一个 **cherry-pick** 命令，让我们能从一个特定的提交合并到当前分支：

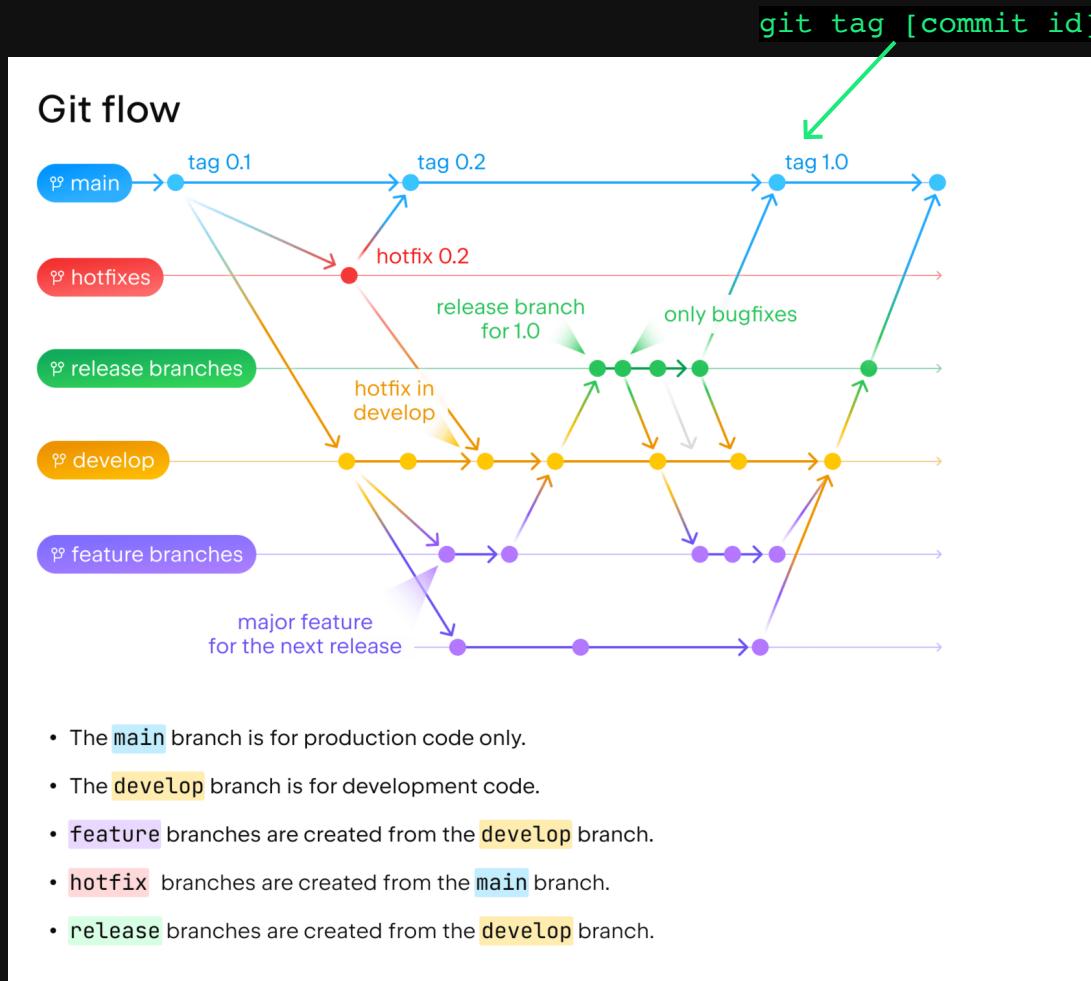


```
$ git switch master
~/Documents/git_test2|dev$ ➔ git switch master
错误：您对下列文件的本地修改将被检出操作覆盖：
readme.txt
请在切换分支前提交或贮藏您的修改。
正在终止

$ git commit -am 'Try stash'
# 记下你想要合并的某一次 commit id
$ git log --graph --pretty=oneline --abbrev-commit
$ git switch dev
$ git cherry-pick <commit id>
$ git status
$ vim readme.txt          # 解决冲突问题
$ git commit -am 'Try cherry-pick'
$ git log --graph --pretty=oneline --abbrev-commit
$ cat readme.txt
```

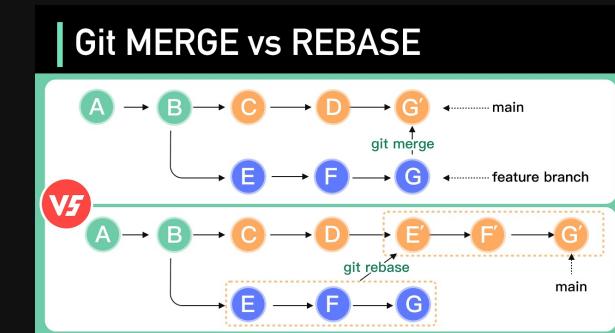
• Feature分支

- 软件开发中，总有无穷无尽的新的功能要不断添加进来。
- 添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，**每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。**



• 多人协作

- 本地新建的分支如果不推送到远程，对其他人就是不可见的。
- 多人协作的工作模式通常是这样：
 - 首先，可以试图用 `git push origin <branch-name>` 推送自己的修改；
 - 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；
 - 如果合并有冲突，则解决冲突，并在本地提交；
 - 无冲突或解决冲突后，再用 `git push origin <branch-name>` 推送就能成功！
- 如果 `git pull` 提示 `no tracking information`，则说明本地分支和远程分支的链接关系没有创建，用命令：
`git branch --set-upstream-to <branch-name> origin/<branch-name>`

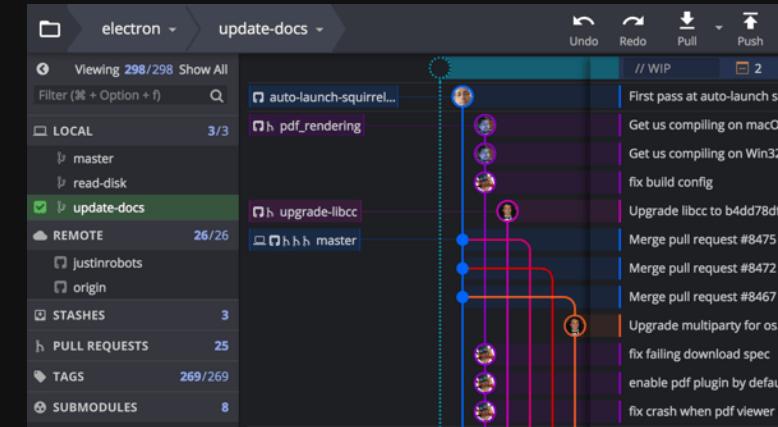
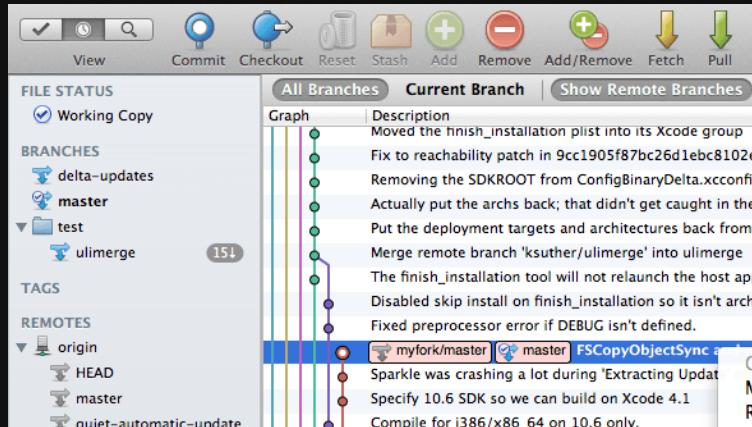
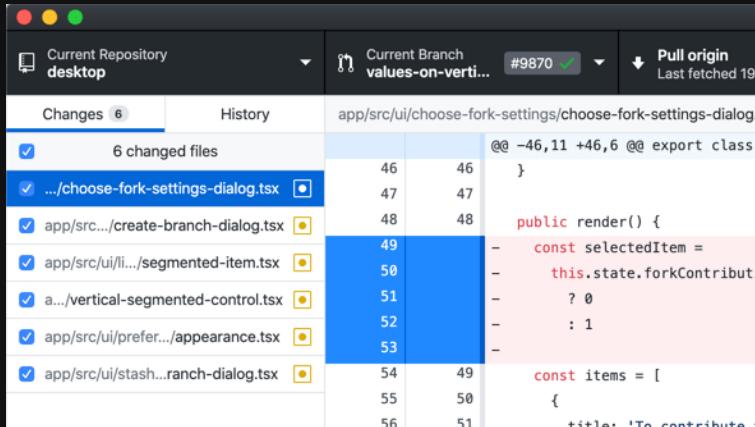


Git MERGE vs REBASE:
Everything You Need to Know

- `git rebase` 操作可以把本地未 `push` 的分叉提交历史整理成直线；
- `git rebase` 的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比（在不同分支间）。

- Git 可视化管理工具

- GUI Clients 官方整理: <https://git-scm.com/downloads/guis>



- GitHub Desktop

Platforms: Mac, Windows
Price: Free
License: MIT

- SourceTree

Platforms: Mac, Windows
Price: Free
License: Proprietary

- GitKraken

Platforms: Linux, Mac, Windows
Price: Free / \$59+/user annually
License: Proprietary

Repo of the course: <https://github.com/iphysresearch/GWData-Bootcamp>

Homework

1. 自己创建一个 GitHub 账户, fork 本课程的远程仓库后再 clone 到本地。

```
$ git clone git@github.com:<你的github账户名>/GWData-Bootcamp.git
```

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks](#).

Required fields are marked with an asterisk (*).

Owner * Repository name * GWData-Bootcamp is available.

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

不要勾选, 可以把homework分支也一并fork

Copy the main branch only
Contribute back to iphysresearch/GWData-Bootcamp by adding your own branch. [Learn more](#).

You are creating a fork in the BNUGW organization.

Create fork

2. 未来的编程作业将仅在 `homework` 分支上完成。你需要在仓库中新建目录 `/GWData-Bootcamp/2023/homework/<你的名字>/`; 并最终把该分支的修改 push 到本课程的远程仓库中, 即 `$ git push origin homework`; 并在 GitHub 该分支下发起 Pull Request (PR) 至本课程仓库。

3. 不要修改其他学员的作业目录和作业内容!

