1. Git

笔记转录于B站up：**GeekHour**，发表的《一小时Git教程》。笔记中若有讲解不清楚的地方，可访问原视频观看学习。URL：https://www.bilibili.com/video/BV1HM411377j?p=1&vd_source=c69af8172c01f57ba6034bbf27b53d5f。

1.1 版本控制系统

版本控制系统，顾名思义，是一个文件的版本的控制软件。

文件的版本会随着程序的更新、修改等不断的迭代。好比在游戏每隔一段时间就会更新一个新版本；服饰隔半年或一年就会出一个同代的新款。

但是随着其存在时间的长久，版本多了就不便于管理。设计时，乙方往往要给甲方多套方案，而甲方总喜欢在一句句“不行”之后加上一句“还是用最开始的吧...”这时候版本控制就发挥了作用，设计时可以直接找到最开始的版本并将其发给甲方。

版本控制系统就像一个大的仓库，里面存放着一款款产品（可以是文本，程序代码等）的迭代版本。用户可以在里面下载指定版本的产品或上传修改后产品。

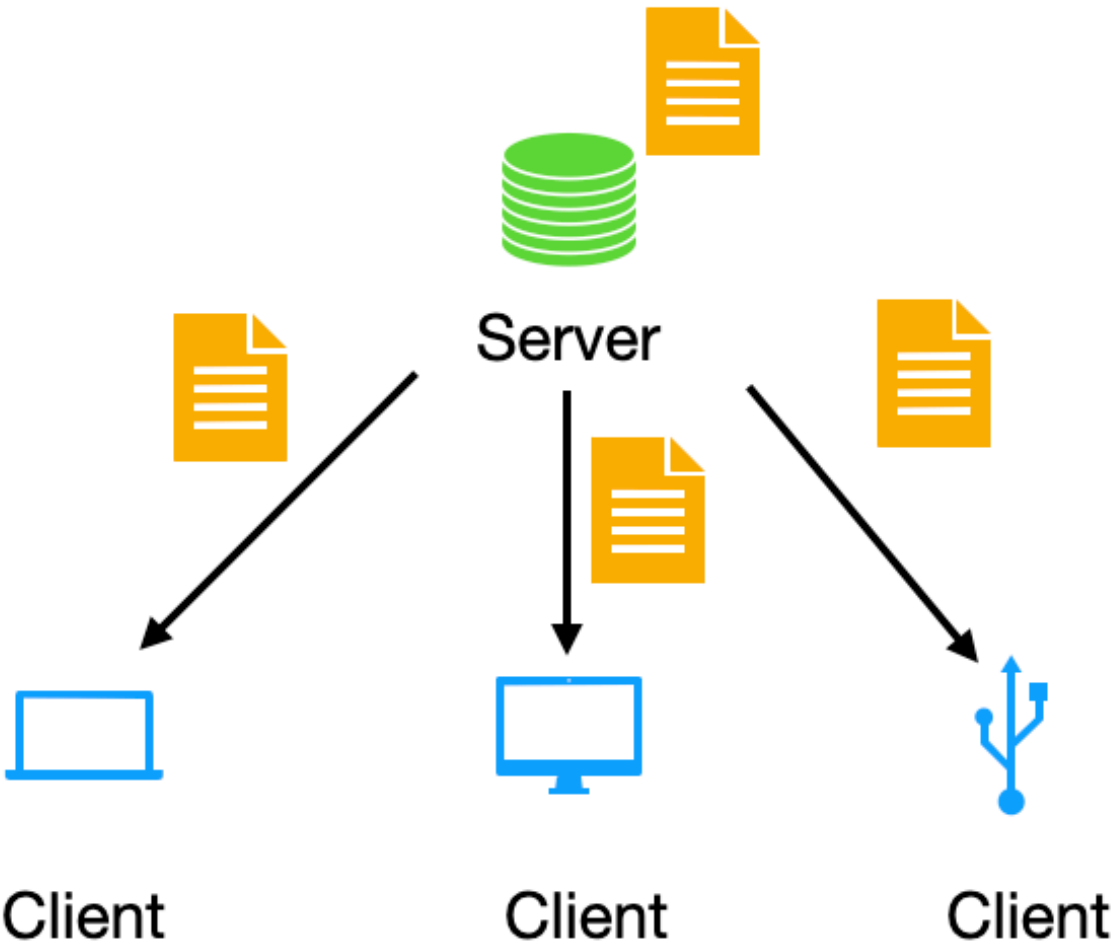
这里有利于团队的集体开发。

1.2 版本控制系统的分类

1. 集中式

   集中式为所有电脑共用一个中央 Server，每台电脑可以在这个Server里下载或上传产品新的版本。

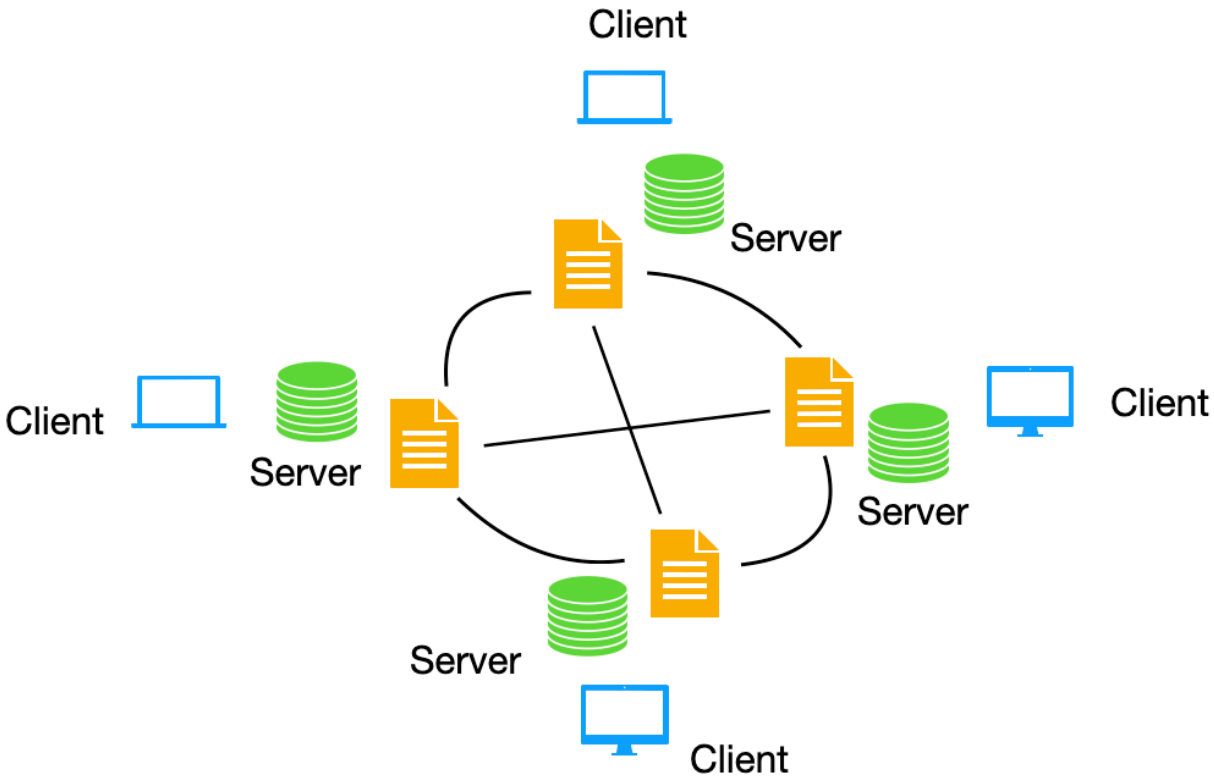
   但他有个明显的问题，Server一旦出现问题，整个工作部门的工作进程就会受到影响。



2. 分布式

分布式版本控制系统，每个电脑都有一个完整的版本库Server，文件可以同步到不同的电脑Server上，来实现协同的版本控制效果。这样也不用考虑中央Server暴毙的影响。

Git 就是一种分布式版本控制系统，因为其开源、速度快、支持离线管理等特性，可以说已经成为目前最流行的版本控制系统了。像是GitHub、GitLab、Vue的等网站上的托管开源项目都是 Git 来管理的。



1.3 Git 的安装

找到 git 的官网，下载对应操作系统的git。这里下载的都是命令行版本的，要是用不习惯也可以下载图形操作界面版本的 git 或着插件版本。

本人电脑Mac mini，MacOS，M2芯片，命令行版本

官网url: https://git-scm.com。官网还有详细的操作指南，本文档只做快速了解。

macOS

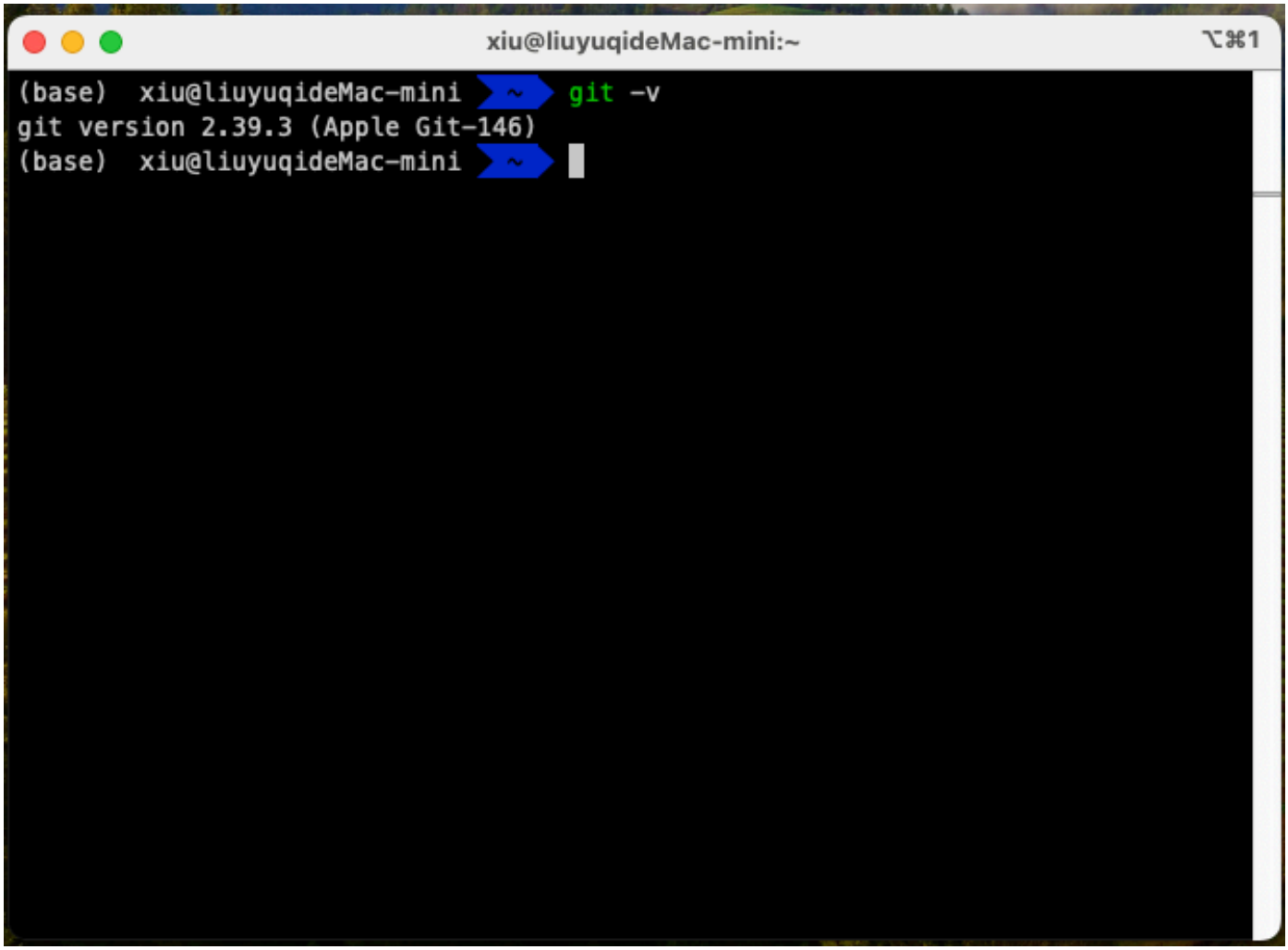
```
brew install git
```

Linux

```
# ubuntu/debian
apt-get install git

# fedora
yum install git (up to fedora21)
dnf install git (fedora22 and later)
```

检查是否安装完成

```
git -v # 查看git版本
> git version 2.39.3 (Apple Git-146) # 安装成功
```



1.4 Git的配置

安装成功后需要配置一下 Git ， 这样才能识别出来是谁提交了内容。类似于配置用户信息。

```
# 配置用户名
git config --global user.name "your_name"

# Local(省略)：本地配置，只对本地仓库有效
# --global：全局配置，所有仓库有效
# --system：系统配置，对所有用户生效

# 用户邮箱
git config --global user.email "your_email_path"

# 保存用户名个密码
git config --global credential.helper store

# 查看git的配置信息，按q退出
git config --global --list
```

2. Git 仓库

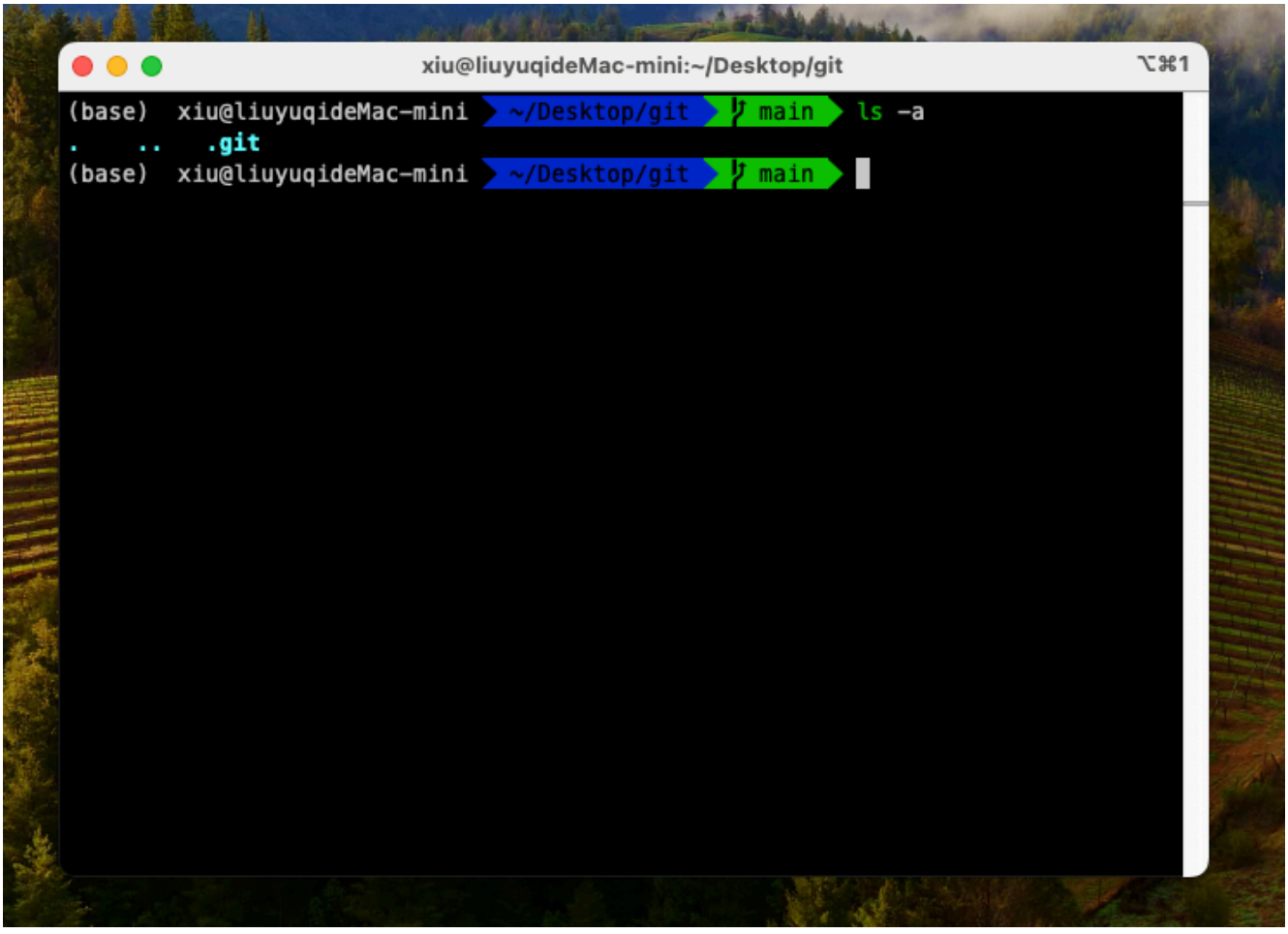
仓库，repository，简称 Repo。用来存放文件，在仓库里的文件会被 git 管理起来，用户对文件的操作会被记录追踪，以便用户可以查询或还原到某一历史版本。

2.1 创建仓库

1. **git init**

   本地创建仓库，把本地某一文件变为仓库这个角色

```
# 创建一个目录
mkdir "your_dir_name"
cd "your_dir_name"
git init

# 创建成功后会出现一个（main）的指示
# 该目录成为仓库后会自动创建一个 .git 文件，这个文件比较重要，所以会隐藏起来
# 查看 .git 文件
ls -a
> . .. .git

# 若想指定一个文件作为仓库，可在后面加上仓库名称，会在当前位置创建一个新的目录
git init "repo_name"
cd "repo_name"
ls -a
> . .. .git
```



2. **git clone**

   克隆仓库，从远程服务器上克隆一个仓库到本地。这应该是用的比较多的，克隆github上的项目经藏会用到git

```
# 从GitHub上拉取视频作者的项目
git clone https://github.com/geekhall-laoyang/remote-repo.git
```

3. 工作区域和文件状态

3.1 工作空间

git的工作区分为三个内容：工作区、暂存区、本地仓库。

- 工作区

  .git 所在的目录，是我们实际操作的地方

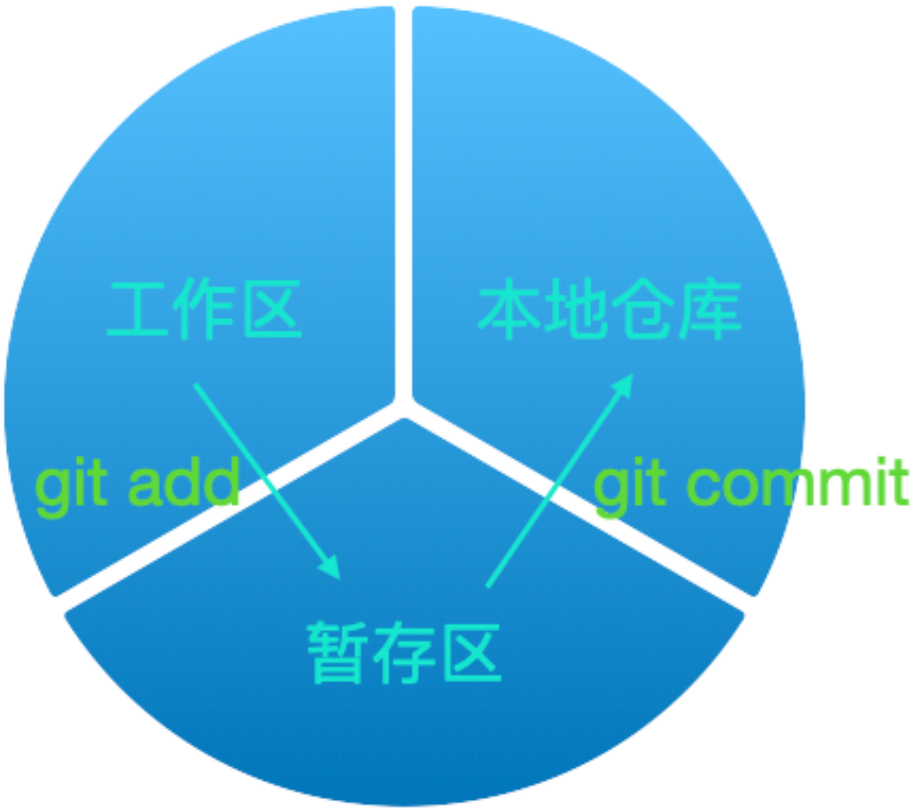
- 暂存区

  .git/index，临时存放数据的地方

- 本地仓库

  .git/objects，是 git 存储代码和版本信息的主要位置

三者关系：现在 “工作区” 完成任务，提交至 “暂存区”，再上传到 “本地仓库”。



3.2 文件状态

- 未跟踪 Untrack

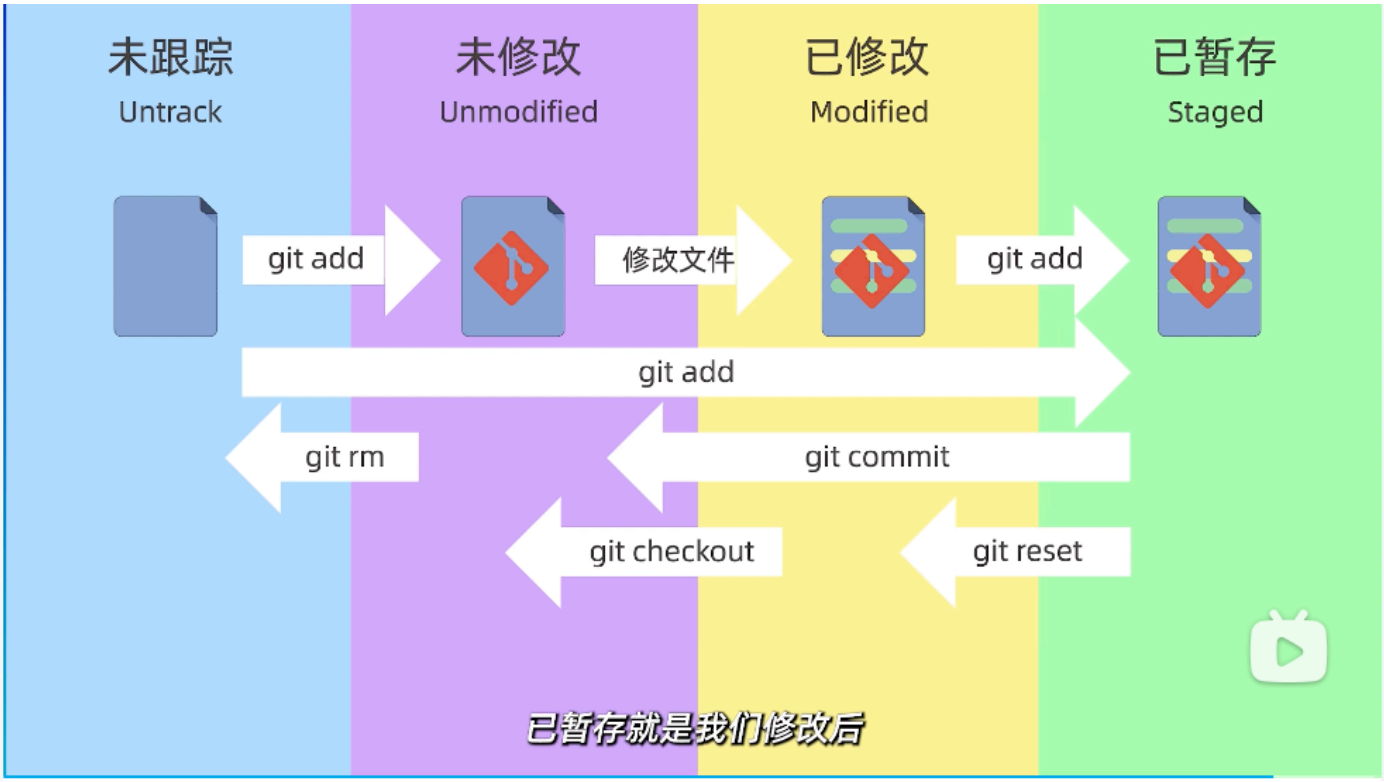
  新创建，但还未被 git 管理起来的文件
- 未修改 Unmodified

  已经被 git 管理起来的文件，但是文件的内容只有被修改
- 已修改 Modified

  文件内容已经被修改，还未添加到暂存区的文件
- 已暂存 Staged

  已添加到暂存区的文件

这里放一张视频截图表示四者关系



4. Git 指令操作

4.1 git init 创建仓库

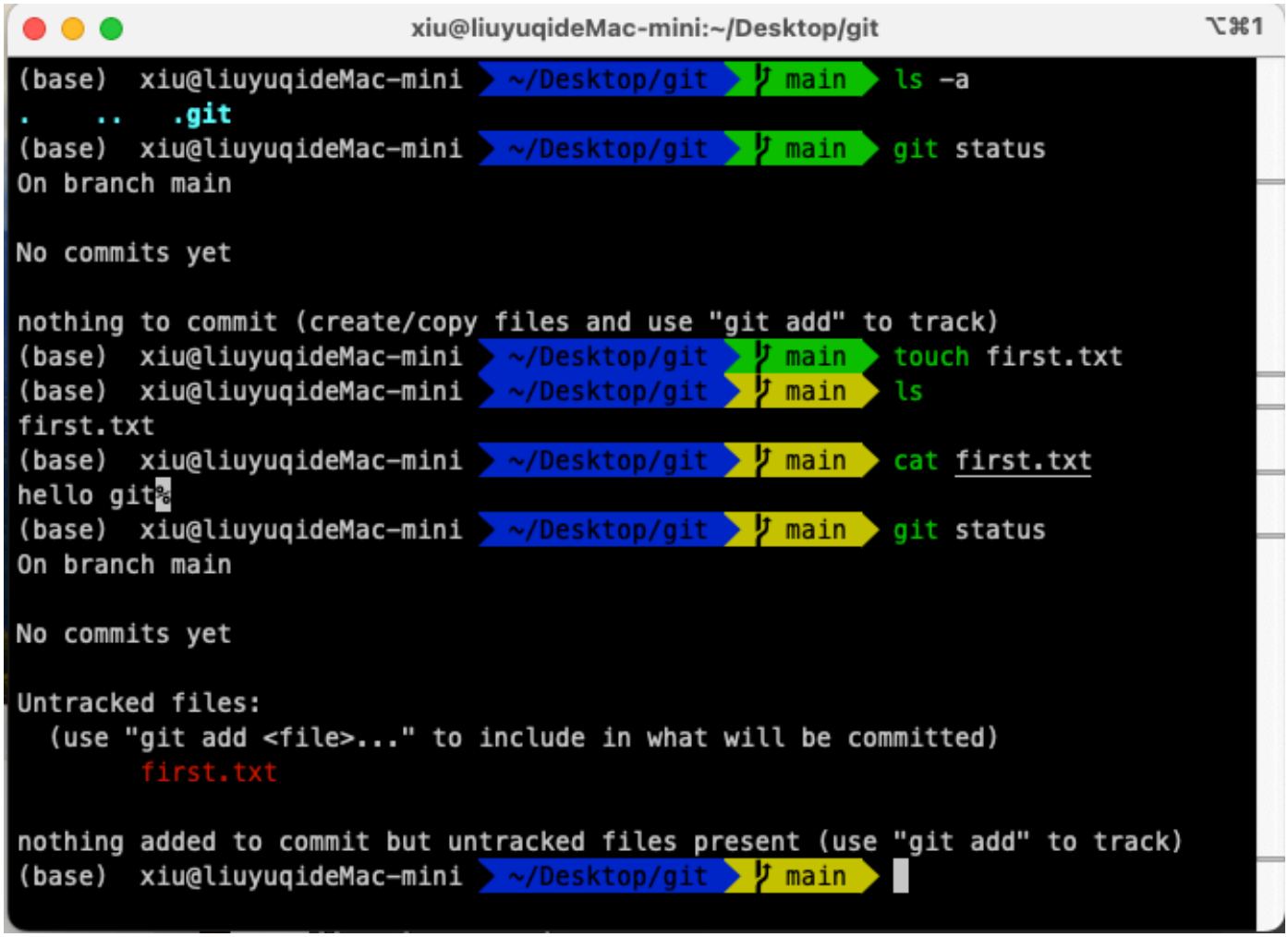
4.2 git status 查看仓库状态

可以查看当前仓库的状态，如处于那个分枝，文件提交状态等。

```
git status
> On branch main # 当前处在main分支
>
> No commits yet
>
> nothing to commit (create/copy files and use "git add" to track)
```

当我们创建文件之后，就会有变化了。可以看到和最开始的 status 不一样，这次多了一个返回，提示有一个未跟踪的文件，还提示我们使用 `git add <file>...` 指令将文件添加到暂存区

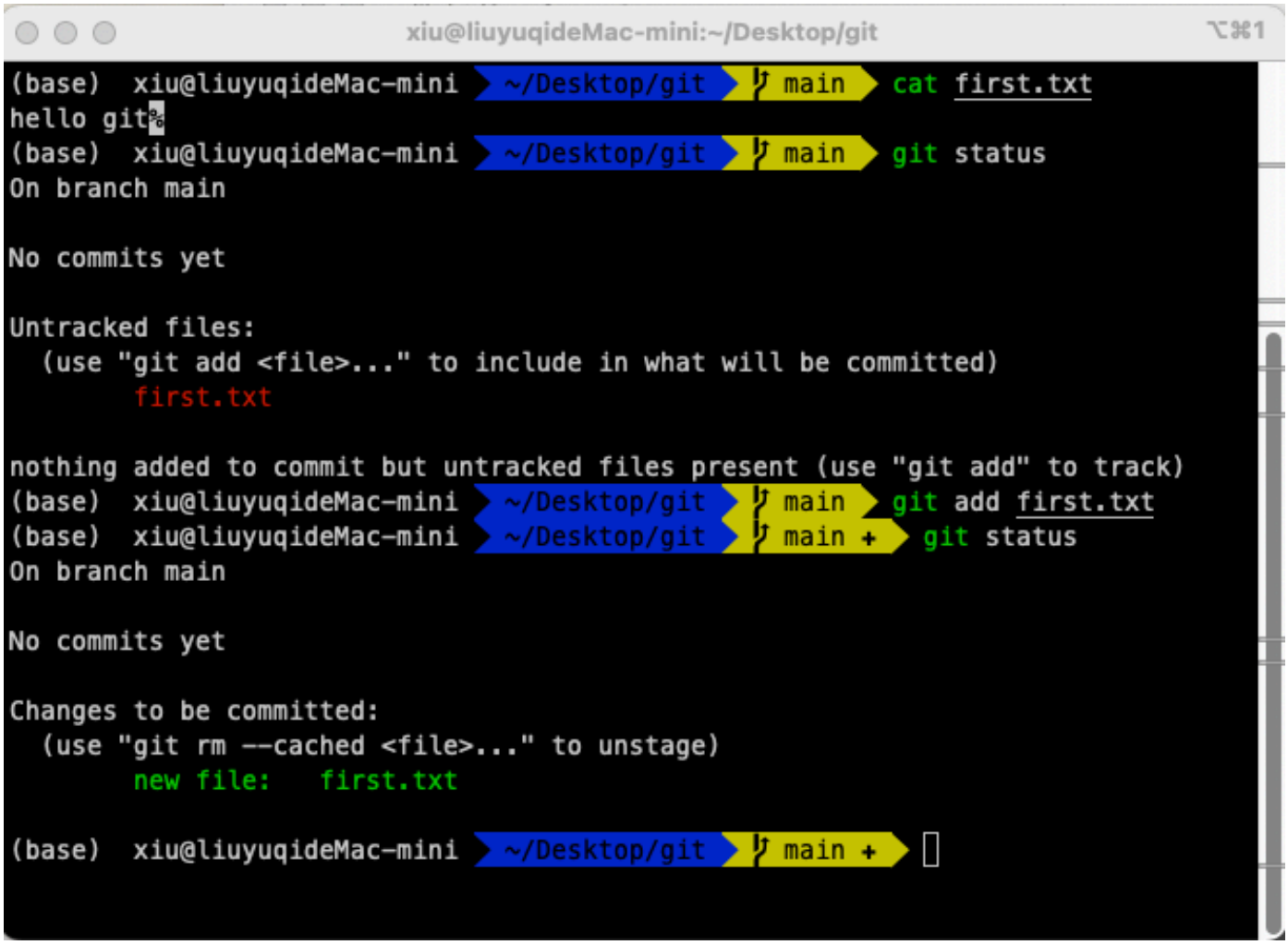
```
touch first.text  # 创建一个空白的.txt文件

ls                # 查看当前目录下的文件
> first.txt

cat first.txt     # 查看文件内容
> hello git%      # 我是已经对文本编辑过了，所以才会有内容

git status        # 查看仓库状态
> On branch main
>
> No commits yet
>
> Untracked files:
>   (use "git add <file>..." to include in what will be committed)
>         first.txt
>
> nothing added to commit but untracked files present (use "git add" to track)
```



4.3 git add 添加到暂存区

添加文件到暂存区。文件变成了绿色，表示已经添加到暂存区，等待上传到本地仓库。提示中有个指令`git rm --cached <file>...`表示将文件移除暂存区，放回工作区。

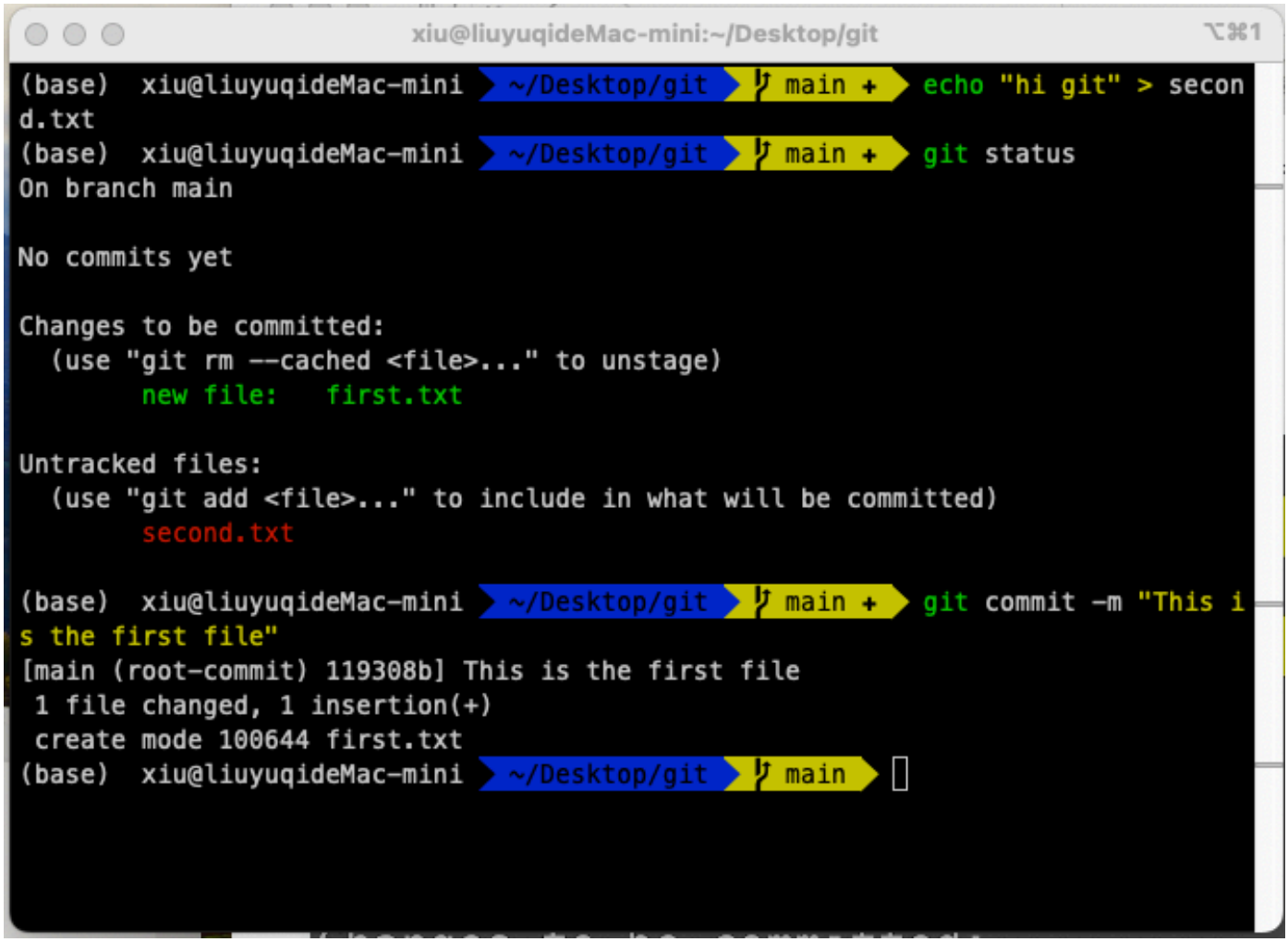
```
git add first.txt
```



4.4 git commit 提交

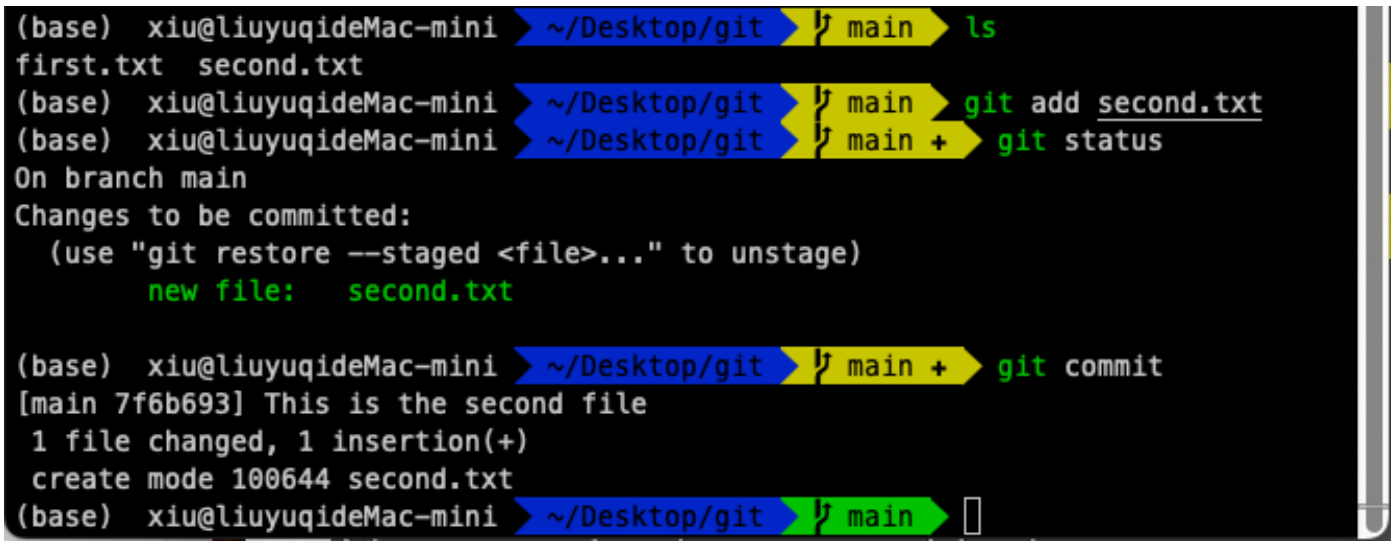
文件添加到暂存区后下一步需要上传到本地仓库，使用命令`git commit -m <file_message>...`。只有上传到本地仓库，文件才会被 git 控制起来。`git commit -m <file_message>...`。**只能将暂存区的文件上传**，工作区的文件不受影响。

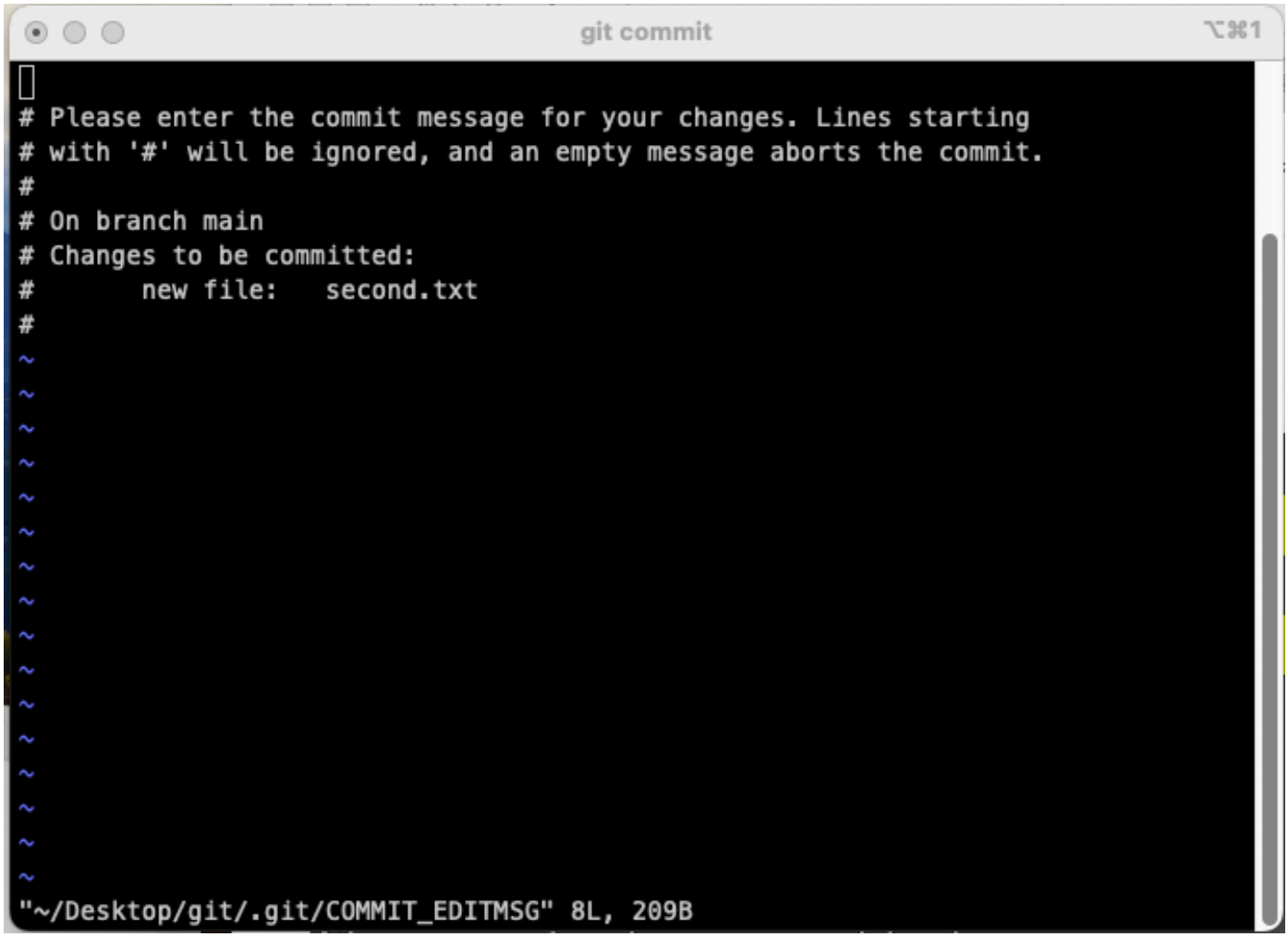
```
# 创建第二个.txt文件，内容为“hi git”
echo "hi git" > second.txt

# 查看仓库状态
git status

# first.txt提交到本地仓库。-m不加的话会出现一个交互界面，下面展示
git commit -m "This is the first file"
```



第二个文件上传，上传时不添加 `-m`，会出现一个交互式窗口。和上面其实是一样的，也要添加上传信息。

```
git add second.txt
git status
git commit
```

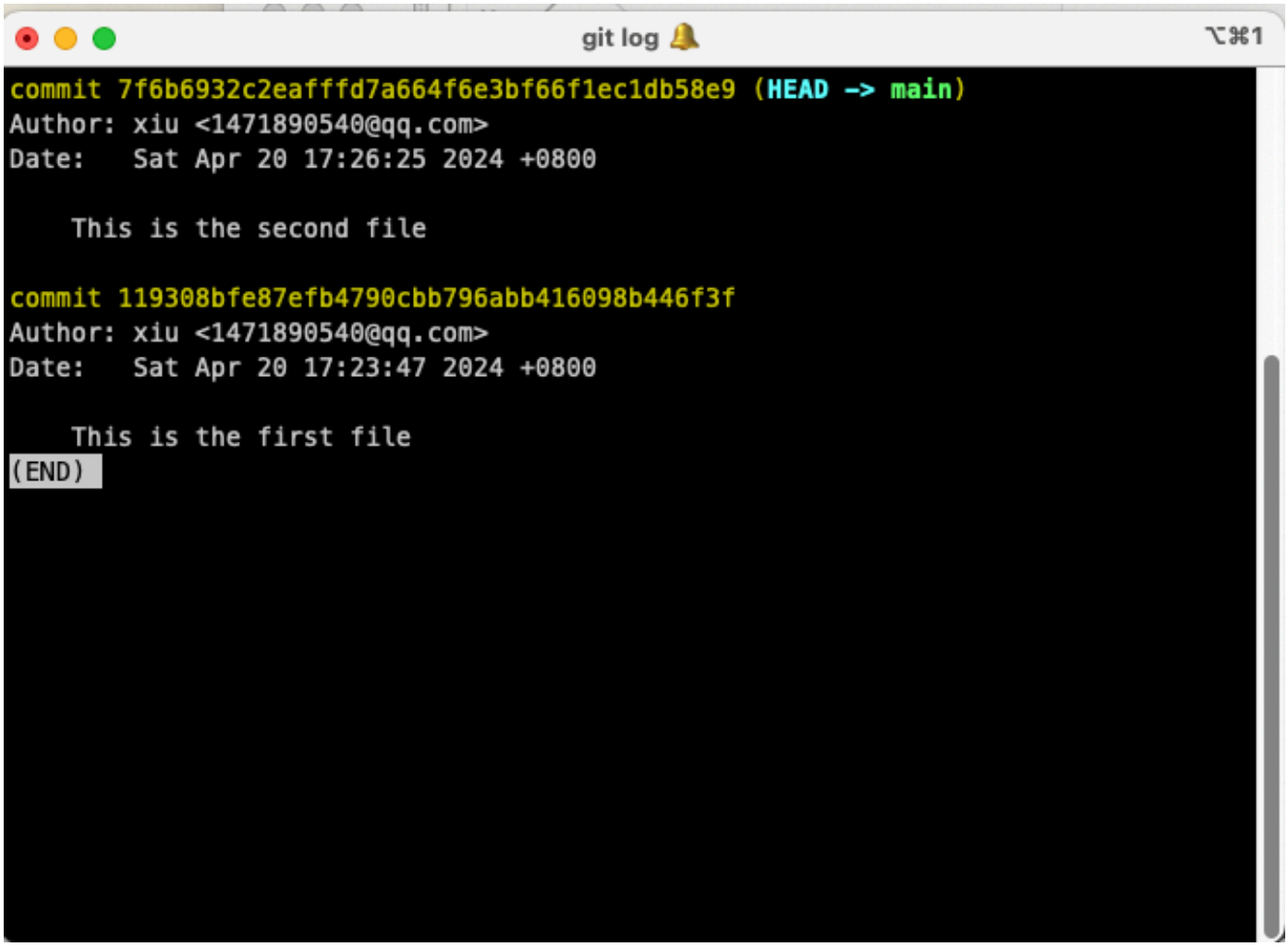


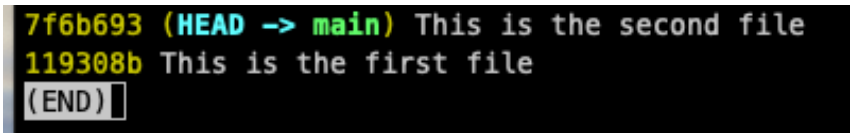


4.5 git log 日志

还可以使用 `git log` 命令来查看日志。log 会展示每次提交产生的id，提交人员，提交时间等信息。

```
git log

git log --oneline # 简单版日志
```





4.6 git reset 回溯

在开发的时候，我们有时候需要回退到某一版本，这时候就可以使用`git reset`这个命令。有三个方法：

- `git reset --soft`

  表示回退到某一版本，并保留工作区和暂存区的所有修改内容

- `git reset --hard`

  回退到某一版本，并且丢弃工作区和暂存区的所有修改内容

- `git reset --mixed`

  回退到某一个版本，只保留工作区的修改内容，丢弃暂存区的修改内容。这也是`git reset`的默认方式。

方式	工作区	暂存区
--soft	✅	✅
--hard	❌	❌
--mixed	✅	❌

他这里的保留于丢弃用法理解如下。

```
# 创建三个新的文件
echo "one" > one.txt
echo "two" > two.txt
echo "three" > three.txt

# 上传到暂存区
# 这里分别提交以代替不同版本
git add one.txt
git commit -m "one"

git add two.txt
git commit -m "two"

git add three.txt
git commit -m "three"

# 查看log 可以看到之前的两个上传和这三次的上传，一这三次的上传为例，one two three
git log --oneline
> fad47e0 (HEAD -> main) three
> d9937fa two
> ebbe386 one
> 7f6b693 This is the second file
> 119308b This is the first file
> (END)

# 分别复制了三个文件以展示效果

#####################################
# --soft
git reset --soft d9937fa  #回退到two
git log --oneline
> d9937fa (HEAD -> main) two
> ebbe386 one
> 7f6b693 This is the second file
> 119308b This is the first file
> (END)
# 可以看到 回退到two版本了，three已经不在仓库里了

# 查看工作区和暂存区
# 在工作区 three.txt还在，且内容没被修改
ls
> first.txt  second.txt two.txt
one.txt    three.txt
cat three.txt
> three

# 暂存区 three.txt也是还在的
git ls-files
>
> first.txt
```

```
> one.txt
> second.txt
> three.txt
> two.txt

# 查看仓库状态 three是新文件。因为返回到two版本时three还没上传，当时的three就是新文件。所以回退的时候就保留了原样。
git status
> new file:   three.txt


#################################
#  --hard  d9937fa也可以换成HEAD^,表示上一代
git reset --hard d9937fa

# 工作区，three.txt已经不见了
ls
> first.txt  second.txt
one.txt    two.txt

# 暂存区也是一样
git ls-files
> first.txt
> one.txt
> second.txt
> two.txt

# 仓库状态，没有显示有新文件
git status


###################################
#   --mixed
git reset --mixed d9937fa

# 工作区
ls
> first.txt  second.txt two.txt
one.txt    three.txt

# 暂存区
git ls-files
> first.txt
> one.txt
> second.txt
> two.txt

# 仓库状态 three是未修改文件，需要加到暂存区
git status
> Untracked files:
>   (use "git add <file>..." to include in what will be committed)
>         three.txt
```

这三种用法都是对两个版本间进行操作，所及谨慎使用 `--hard` 他会直接删除两版本间的所有文件。如果误操作了也可以使用回溯来挽救。

```
# 找到误操作的ID
git reflog

# 回溯
git reset --hard <'误操作ID'>
```

4.7 git diff 差异

用于找到存在的差异，可以是各个区间的差异，也可以是两个不同版本之间的差异，甚至是可两个分支间的差异。默认是比较工作区和暂存区之间的差异。

```
cat three.txt
> three

vim three.txt # 改成了333

cat three.txt
> 333

git diff
> diff --git a/three.txt b/three.txt
> index 2bdf67a..55bd0ac 100644
> --- a/three.txt
> +++ b/three.txt
> @@ -1 +1 @@
> -three  # 删除的内容
> +333    # 新增内容
> (END)

# 新内容添加到暂存区，再次diff就没有差异了
git add three.txt
```

还可以比较工作区和仓库之间的差异。继续使用上面的文件，three还没有上传到仓库

```
git diff HEAD
> index 2bdf67a..55bd0ac 100644
> --- a/three.txt
> +++ b/three.txt
> @@ -1 +1 @@
> -three  # 删除的内容
> +333    # 新增内容
> (END)
```

暂存区与仓库

```
git diff --cached
> index 2bdf67a..55bd0ac 100644
> --- a/three.txt
> +++ b/three.txt
> @@ -1 +1 @@
> -three  # 删除的内容
> +333    # 新增内容
> (END)
```

不同版本之间

```
git diff <id1> <id2>
```

与上一个版本比较

```
git diff HEAD~ HEAD
# HEAD^也是一样的
# HEAD~n 表示上n个版本
```

4.8 git rm 删除

对于已经提交到仓库的文件，可以使用 `git rm` 来将文件从仓库删除，一同会删除工作区和暂存区的文件。

```
# 删除工作区文件 Linux方法
rm <file_name>
git add . # 更新暂存区

# git方法
git rm <file_name>

# 删除暂存区的文件（一同删除仓库内的文件），工作区保留
git rm --cached <file_name>
```

4.9 gitignore 忽略文件

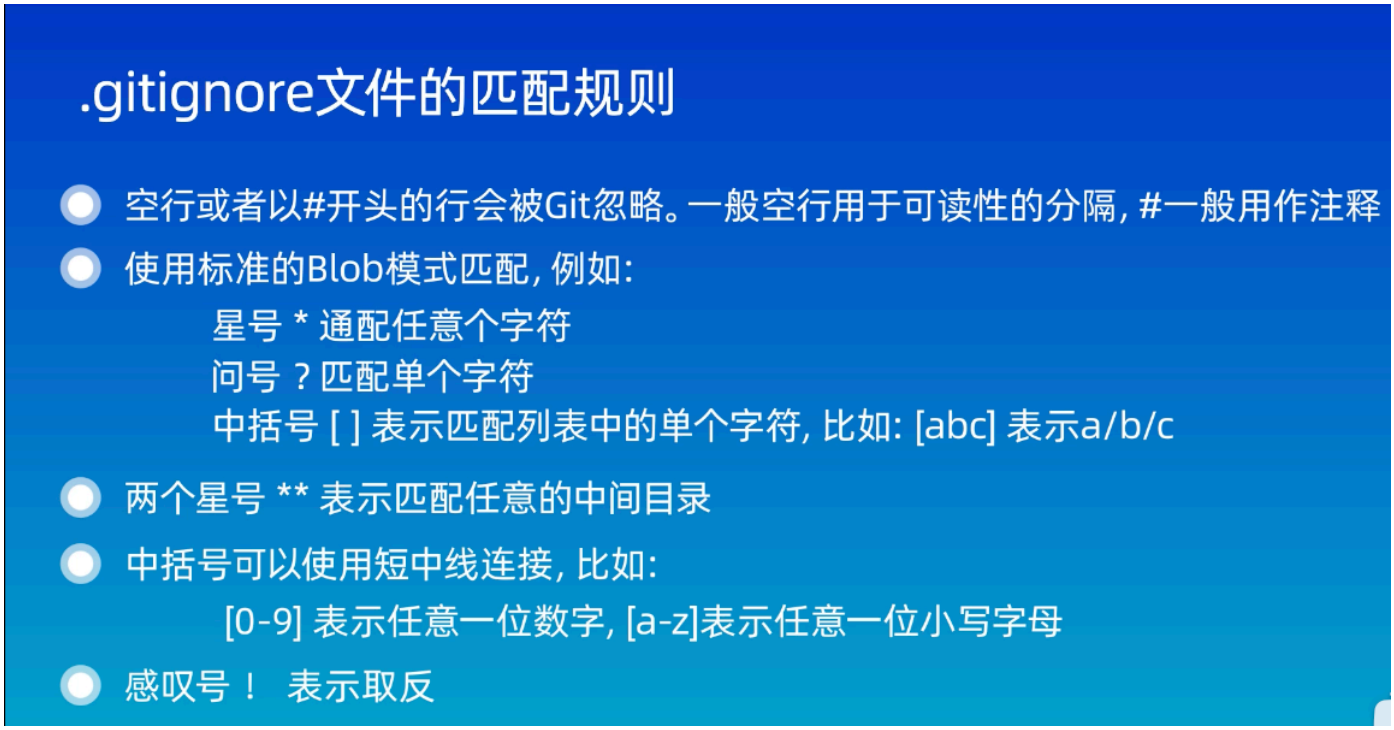
在上传仓库的时候，有些文件是我们不需要上传的，为了做仓库内文件更干净，体积更小，对于一些不需要上传的文件我们就会把它放到 `.gitignore` 文件中，这样在上传的时候就不会一同上传到仓库中了。

一些不需要上传的文件：

- 系统或者软件自动生成的文件
- 编译产生的中间文件
- 运行时自动生成的日志文件、缓存文件、临时文件
- **涉及身份、密码、口令、密钥等敏感文件**

在 `.gitignore` 文件中列出需要忽略的文件类型，前提是该文件还未被加入到仓库中

```
# 创建 .gitignore 文件
touch .gitignore

# 编写 .gitignore文件内容
vim .gitignore

# 会进入文本编译器，如忽略所有.log文件
*.log      # *表示所有，*log表示所有以.log结尾的文件，也就是log日志文件
```

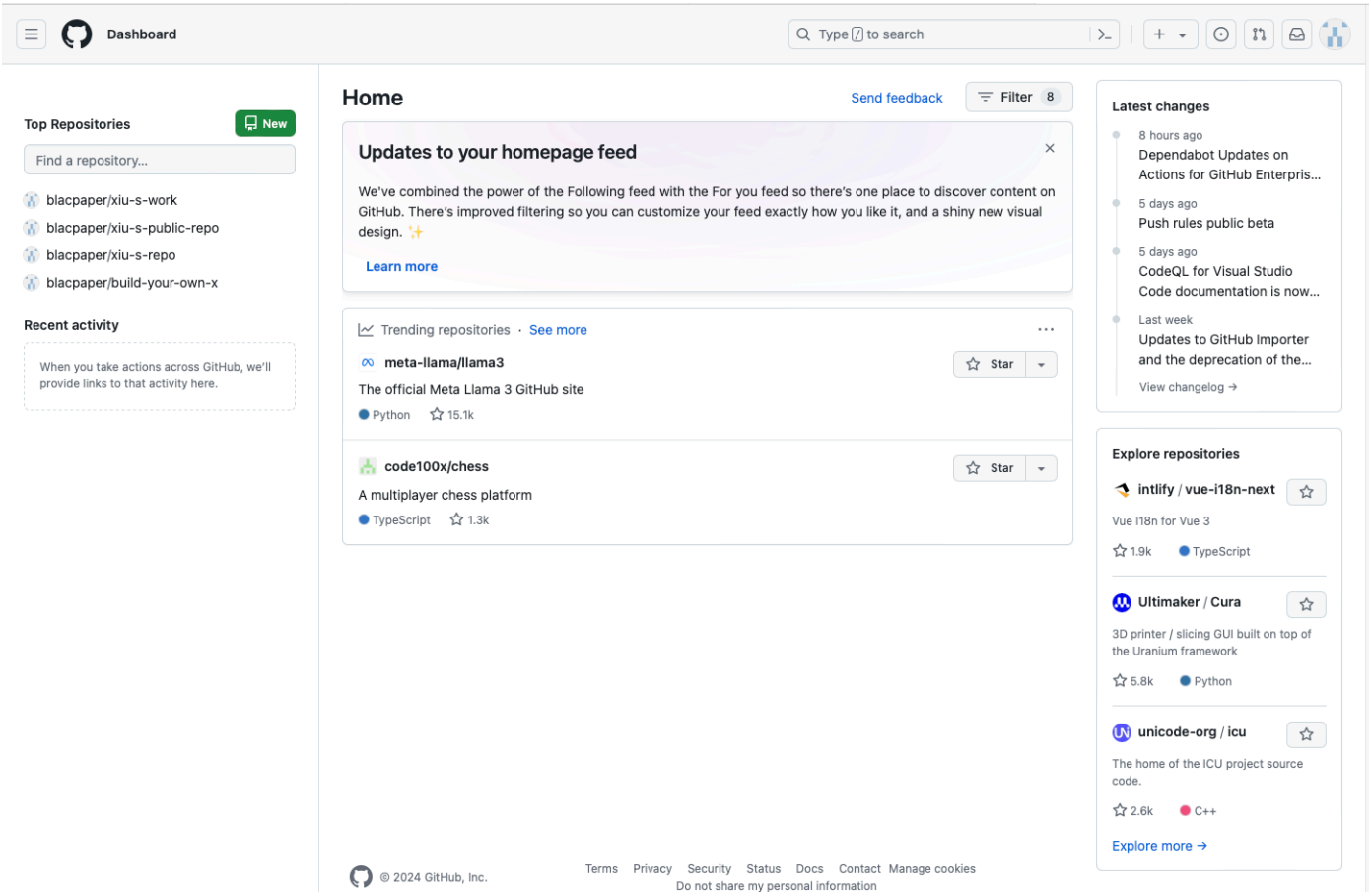


5. GitHub

GitHub是一个开源的代码托管平台，URL：https://github.com。我们可以在这里创建账号，管理自己的仓库。账号注册和其他的平台相同。如果打不开，挂梯子翻墙，这个不会被查水表。

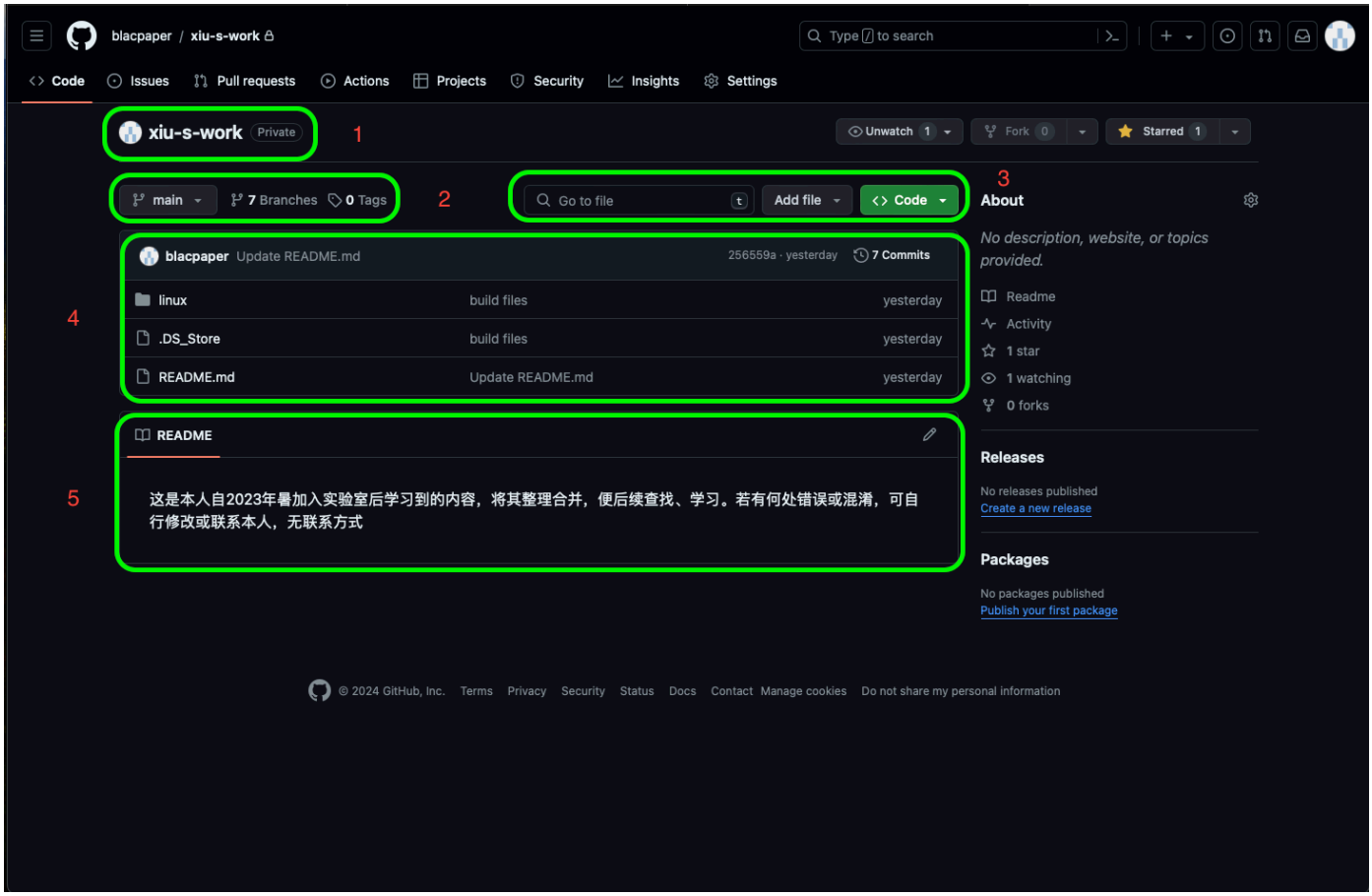
5.1 GitHub使用

在GitHub平台我们可以搜索到不错的项目、开源软件。也可以创建自己的仓库存放自己的工程代码。除了国内连接不稳定、是一个开源的平台，其他的都还是很值得去学习了解的。



下面是GitHub的仓库，和上面说到的仓库一样，这里会存放我们`commit`的文件，但不同的是，我们`commit`之后还需要上传`push`到GitHub服务器上。

1. 仓库名
2. 项目线程，main主线，7条分支（含主线），0标签
3. 项目相关操作，查找文件，添加文件，下载文件
4. 项目文件
5. README，项目的概述。里面写什么都可以。



若对GitHub感兴趣，可阅读`GitHub`文件下的内容

5.2 上传下载GitHub文件

仓库绑定

对于自己的项目需要GitHub托管的，可以将本地仓库与GitHub仓库绑定。在GitHub上创建一个仓库后会有提示如何绑定仓库。如果需要这一部分，建议观看原视频，里面清楚的讲解了这方面的内容，还处理了ssh密钥问题。

上传文件至GitHub

`git push origin <branch_name>`

从本地上传修改后的文件到GitHub服务器端，可使用上面的指令，其中`<branch_name>`换成分支名称。（分支的内容在下面，现在还没讲到）

```
# eg:将main分支下的文件上传
git push origin main  # 这里应该是main:main，由于github仓库和本地仓库的分支一样，写一个就行
```

下载文件从GitHub

拉取：`git pull <github_repo_name> <github_branch_name>:<branch:name>`

```
# eg:拉取绑定仓库的main分支。并于本地仓库合并
git pull origin main
```

克隆：`git clone <repo_path>`

```
git clone https://........
```

下载： Download

如果需要整个项目的副本，克隆是最佳选择。

如果你只是想要快速下载单个文件或整个项目作为压缩包，GitHub的网页界面提供了便捷的选项，可以将整个仓库作为ZIP文件下载；对于只需要下载单个文件的情景，可以在GitHub仓库的文件浏览界面找到该文件，然后点击文件右上角的 "Download" 或 "Raw" 按钮来下载文件的原始版本。

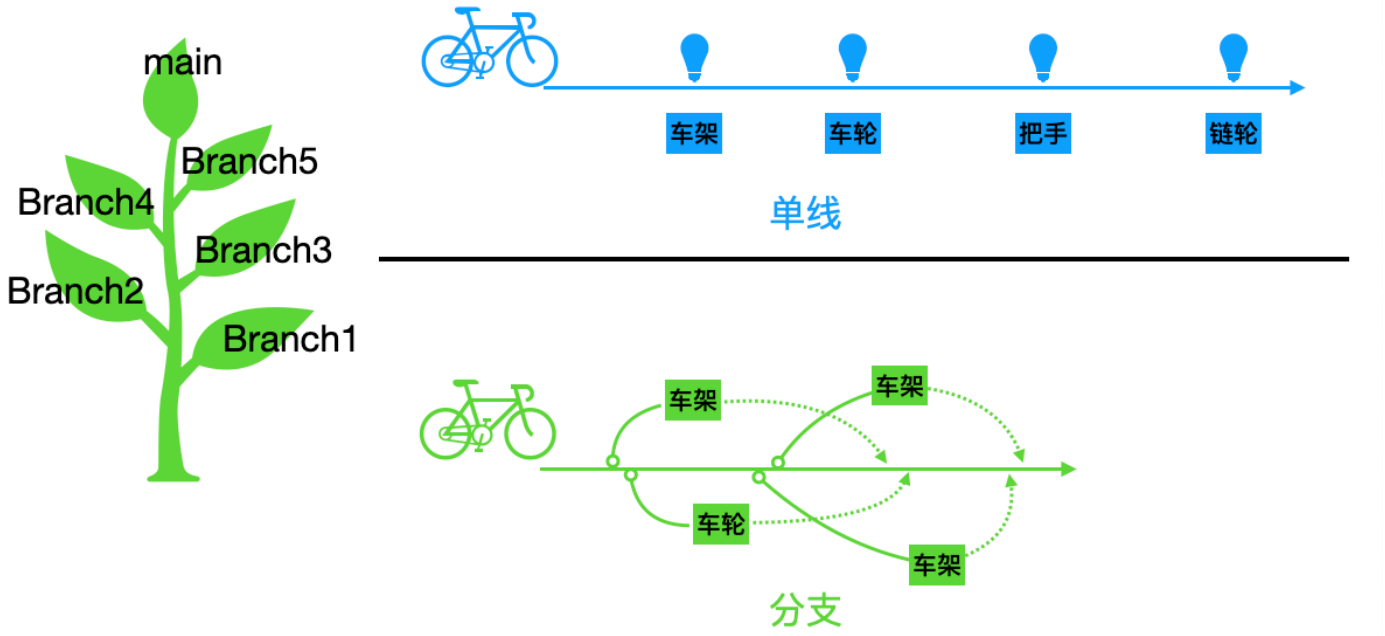
6. 分支 Branch

分支在项目里的存在我认为是很巧妙的，项目不再是枯燥的单线程，而是更加的随心所欲。

下面这张图可以大致看出单线和分支的对比关系。同样对于制造自行车这个项目，单线和分支的计划工作量是大差不差的，但实际开发中我更偏向于分支开发，下面对比偏主观。

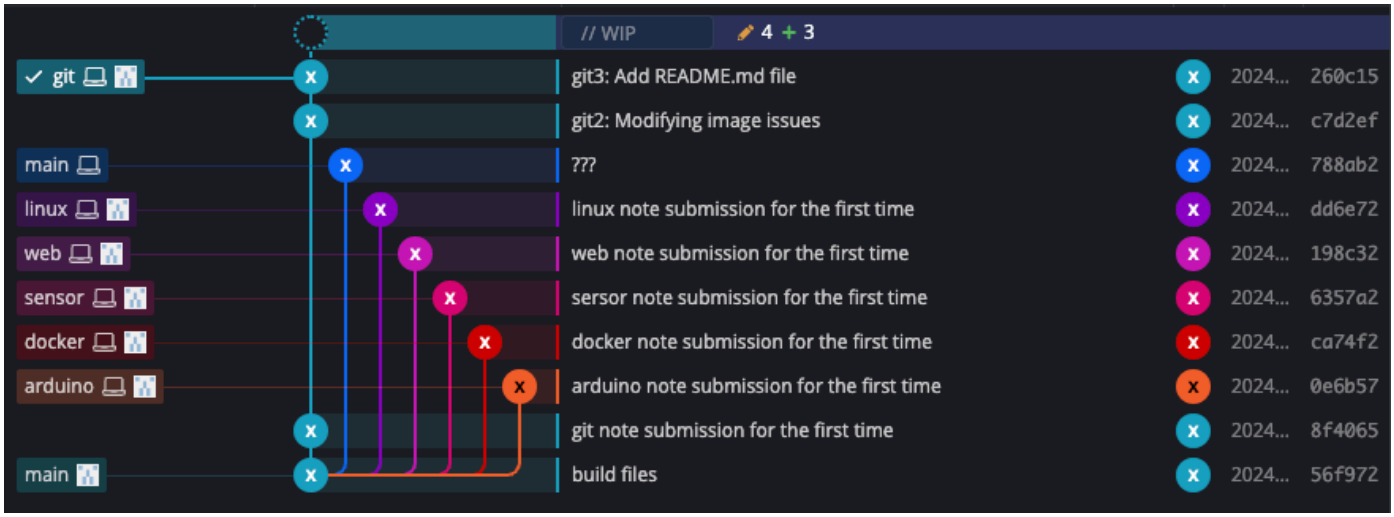
单线需要一步一步完成，有头才有尾，每一步的目的明确。但项目前后匹配不理想时，就需要“重来”。尽管 git 能够回溯（4.6），提供了很大的容错，但是一遍遍的修改，主线main就会比较乱。

分支操作允许我们同步开展多项任务，每个分支之间不互相干扰，分支任务完成后再与主线融合绑定。这样在没完成所有项目时对每个分支都可以修改且不干扰其他分支。（个人感觉，比单线开发更舒服，我不太喜欢单线...）



但实际中需要两个都掌握，分支毕竟还是在单线的基础上的出现的一种新的开发方式，实际进入每个分支中，其实还是单线。且分支越多，在开发团队人员过少的情况下，反而容易手忙脚乱。单人或少人小项目开发，建议使用单线；多人大项目开发，分支值得优先考虑。

下面是这个项目的初步计划，现在已经有点凌乱了，感觉哪里都是活。



6.1 git branch <branch_name> 创建分支

通过这个指令，我们就可以创建一个分支，以后我们就可以在这个分支里进行一些工作，和之前讲的操作是一样的。在合并到main线前，我们在分支中的操作在main线中是没有显示的，当我们在分支中`git add`或`git commit`后，main线上是查不到对应的文件的。

```
# 创建一个分支，名branch1
git branch branch1

# 检查当前所有分支，会返回目前创建的所有分支名
git branch
> branch1
> main
```

6.2 checkout & switch 切换分支

checkout

`checkout`是一个多用途的命令，用于在不同分支之间切换，也可以用于检出代码库中的特定文件或提交。

```
# 切换分支
git checkout <branch_name>

# 检出文件
git checkout <branch_name> -- file_path   # 从 source_branch 分支检出 file_path 文件到当前分支

# 切换到特定的提交
git checkout <commit_hash>  # 也可以用这个方法恢复到之前的某个状态，但<commit_hash>和
<branch_name>不能一样，否则会冲突

# 创建并切换到新分支
git checkout -b <new_branch_name>
```

switch

`switch` 是一个较新的命令，在2.23版本中引用，专门用于在分支之间切换。它不能用于检出文件或提交，其设计目的是简化分支切换的操作。

```
# 切换分支
git switch <branch_name>

# 创建并切换到新分支
git switch -c <new_branch_name>
```

6.3 merge & rebase 分支合并

上面说到了我们为了工作方便，会把相关的一部分分出来单独工作，当我们完成这部分工作时，就需要将分支合并到main线。`merge` 和 `rebase` 是git中的两种分支合并方式。

merge

`merge` 命令将两个分支的历史**合并**在一起。当你执行 `git merge <branch_name>` 时，Git会**在当前分支上**创建一个新的“合并提交”，它将两个分支的更改合并在一起。这种方式保留了项目的历史脉络，可以清晰地看到分支的合并点。

这种操作保留了完整的历史记录和分支点，但可能会产生多余的合并提交，使历史记录变得复杂。

```
# 切换到合并的基线上，这条分支线是不会动的，以main线为例
git switch main

# 将分支branch1合并到main上
git merga branch1
```

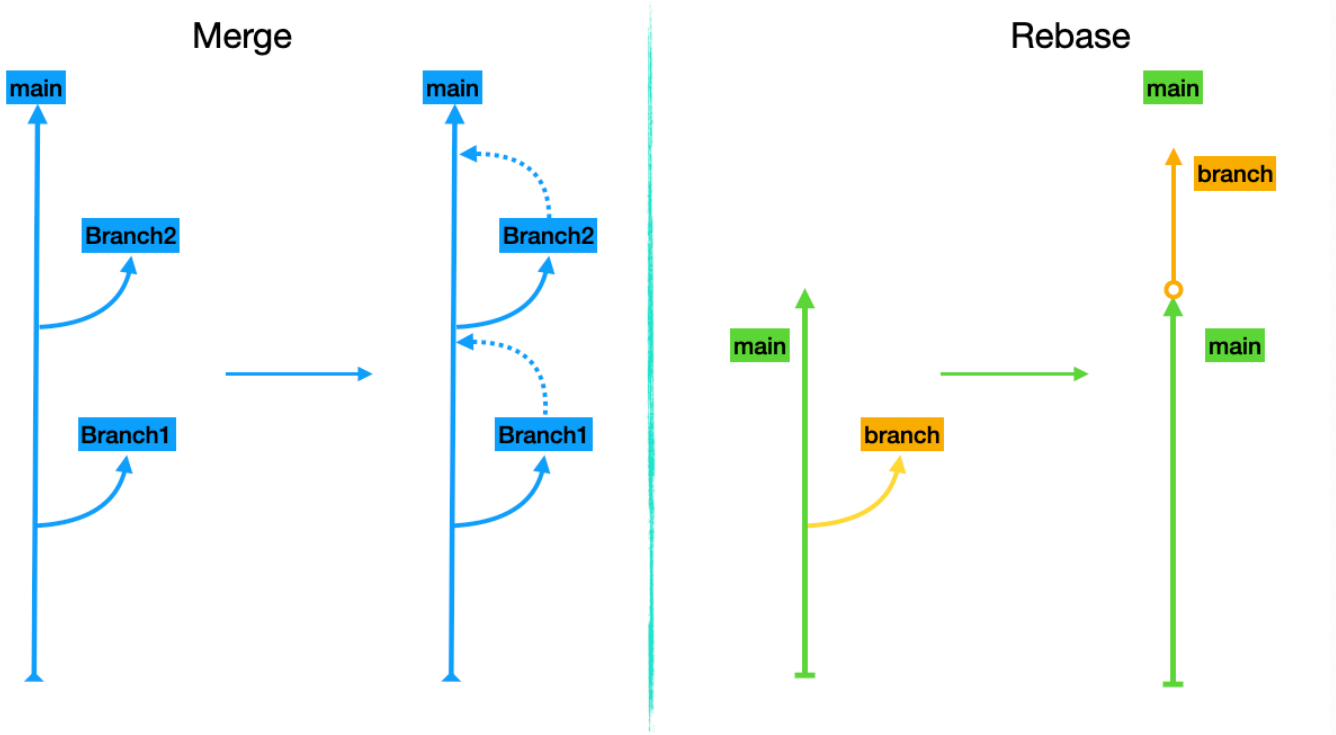
rebase（变基）

`rebase` 命令将一系列提交从一个分支上**“移植”**到另一个分支上。执行 `git rebase <branch_name>` 时，Git会将当前分支上的提交暂存起来，然后将当前分支的指针移动到 `<branch_name>` 分支的顶端，最后将暂存的提交依次应用在新的基分支上。

这种方式可以创建一个更干净、线性的提交历史，且避免了不必要的合并提交。但最致命的是会重新编写项目历史，可能会使其他人感到困惑，分支变单线。

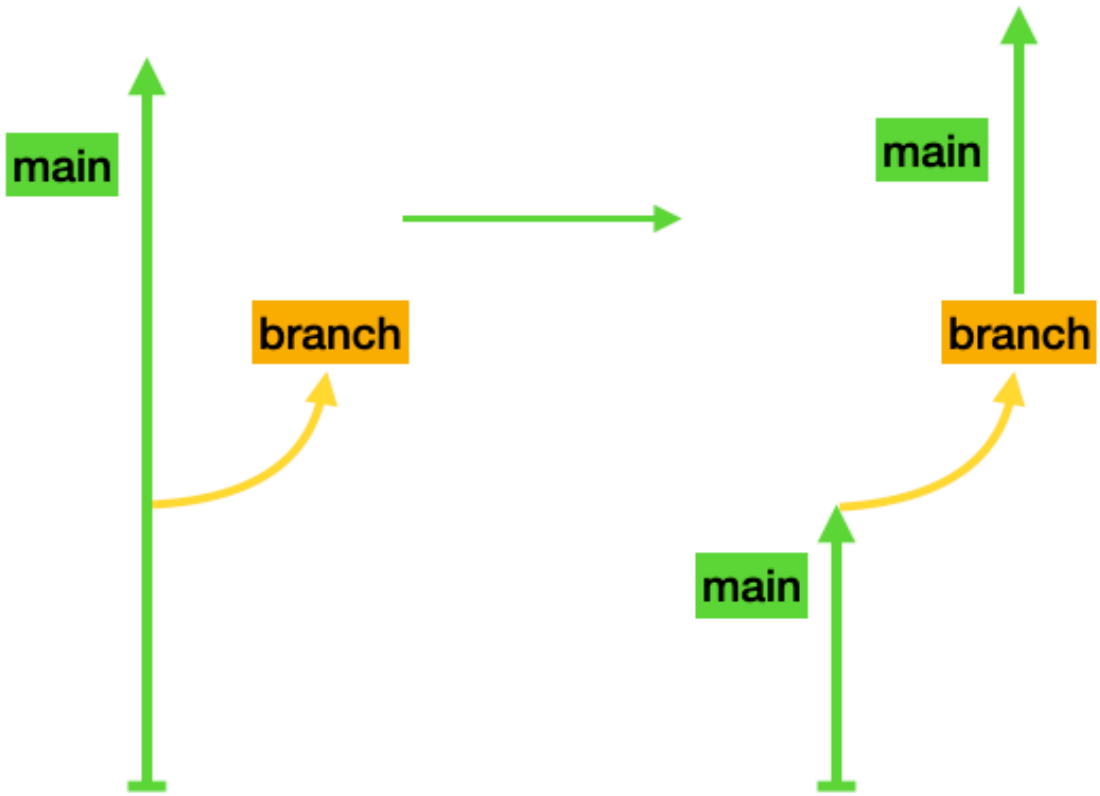
```
# 对应第一个rebase图
# 切换到合并线
git switch branch1

# 将分支branch1变基到基线main上
git rebase main
```

说起来可能会有些抽象，下面有张图便于理解。使用`merge`合并，我们能够看到分支合并的完整历史，一旦合并的过于频繁，就会变得很复杂；使用`rebase`合并，分支也会合并到当前分支上，整体就是一条线了，更加简洁，直观。对于开发者来说，短期还能记得，对于其他成员或长时间再次修改时，就会感觉困惑。



对于`rebase`，还可以出现下面的情况。这里引用视频原话，因为使用`rebase`时，会先查找基线和合并线的共同“祖先“，即**分支前最新的一次提交**，在将这个提交点后的合并线合并到基线上。**这里建议观看原视频**

```
# 对应第二个rebase图
# 切换到合并线
git switch main

# 将分支main变基到基线branch1上
git rebase branch1
```



我们也可以使用指令查看分支图。

```
git log --graph --oneline --decorate --all
```

6.4 git branch -d 删除分支

分支合并之后并不会消失，还是会继续存在的，要是想删除一条分支，可以使用`git branch -d <branch_name>`，来删除该分支。

如果这条分支还没有被合并，就使用`git branch -D <branch_name>`来强制删除这条分支。用法一样。

```
# 删除分支branch1
git branch -d branch1
```

6.5 解决分支合并冲突

多分支工作时，如果main的同一个文件的同一位置被两个分支修改，当这两分支提交时，会出现冲突，git无法确定哪个是需要被保存的。这种冲突情况是需要手动解决的。

可以使用`git status`或者`git diff`来检查冲突的地方。然后就需要手动修改文件内容。修改完后再`add`，`commoit`就可以了，由于之前是合并时报错，这里再次提交他就自动合并了。

也可以使用`git merge --abort`终止合并