# 面向对象编程和Git初步

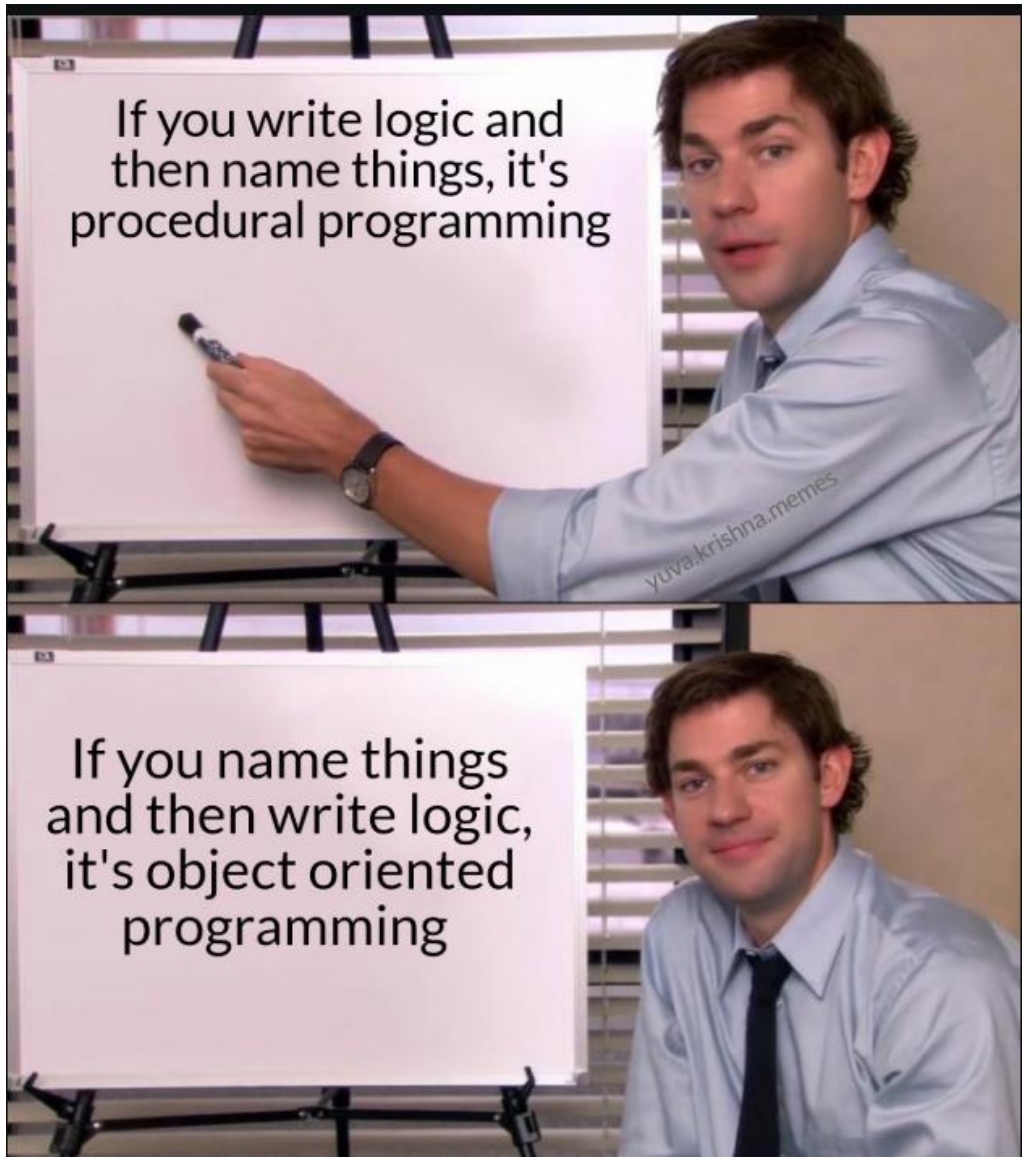
2024年12月12日

- 还在为头脑简单而只会prompt GPT而发愁吗？还在为码力薄弱，只会人云亦云、鹦鹉学舌、复制CSDN而遭人鄙夷吗？带你领略OOP思想，从另一维度解读python语言特性，成为见解独特的git clone玩家。

# What is Object-Oriented Programming (OOP)? (too-long-not-read ver.)

- 一种现代编程范式，强调易于扩展、理解、维护
- 具备封装、继承、多态三大特性
- 与接近硬件执行逻辑的过程式编程(imperative programming）以及采用现代数学思想的函数式编程(functional programming）相比，OOP更接近通常人们的思考方式。

# What is Object-Oriented Programming (OOP)? (meme ver.)



# What is Object-Oriented Programming (OOP)? (python lang. syntactic ver.)

```python
class Student:

    def __init__(self):
        pass

Tomori = Student()

print(Tomori) # <__main__.Student object at 0x000001977797E9040>
```

Tomori是一个被实例化的Student对象

OOP强调对象之间的交互和关系：任何operation都是对object的operation，一切data都是object的data

# 为什么OOP是好的（例子）

## 为什么要演奏春日影

なんで春日影やったの？！！

为什么要演奏春日影？！！

以下内容含有剧透成分，可能影响观赏作品兴趣，请酌情阅读

なんで春日影やったの（为什么要演奏春日影） [1/1] 显示视频

《春日影》作为CRYCHIC的第一首也是最后一首原创歌曲，对乐队、成员们以及分别负责作词和作曲的灯和祥子而言有着不可言喻的重要性，对于本作的剧情推动也起着至关重要的作用。

Q：假如你要把“为什么要演奏春日影”的情节写成一段cutscene（游戏里的剧情过场），你要怎么编写它?

# 主函数：why_play_haruhikage()

why_play_haruhikage_template.py > ...

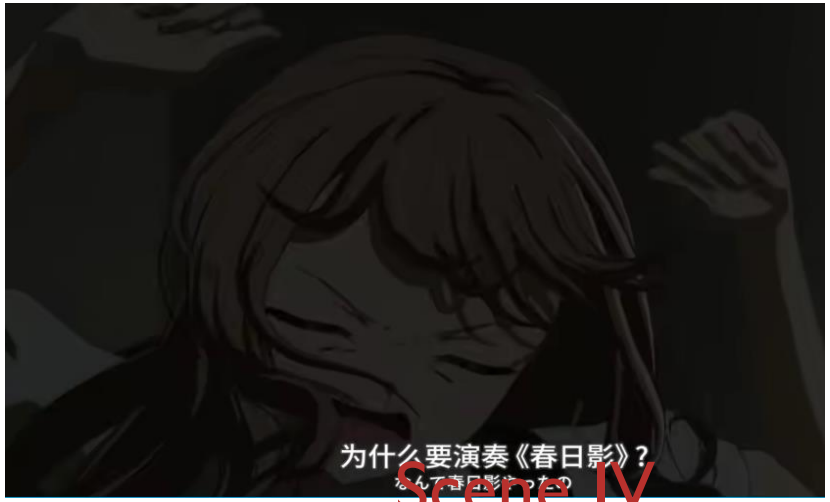
```python
def why_play_haruhikage():
    raise NotImplementedError

if __name__ == '__main__':
    why_play_haruhikage()    # NotImplementedError
```

# MyGO!!!!! #7 就算今天的live结束 14:42~21:58


Scene I 14:48

Scene II 15:44


Scene III 17:44




Scene IV 21:56

## How to organize the code?
## 情况一：Procedural Programming的恶果

- Scene 1: 念白
- Scene 2: 吉他前奏
- Scene 3: run
- Scene 4: yell
- 需要给每一个scene编写一个函数

# 为了和过程一一对应，我们不得不在函数内部根据传参进行分支判断

imperative_haruhikage.py > ...

```python
def monologue(actor: str, content: str):
    if actor == "Tomori" and content == "あの時...":
        print("This is Scene 1.")
        pass
    else:
        raise NotImplementedError

def play(actor: str, instrument: str, song: str):
    if actor == "Lena" and instrument == "guitar" and song == "Haruhikage":
        print("This is Scene 2.")
        pass
    else:
        raise NotImplementedError

def why_play_haruhikage():
    monologue("Tomori", "あの時...")
    play("Lena", "guitar", "Haruhikage")

if __name__ == '__main__':
    why_play_haruhikage()   # NotImplementedError
```

# 过程式编程的问题一：难以扩展，代码复用性差

- 以play()函数为例，每当cutscene中新出现一名乐手，都不得不增加一个判断分支

```
12      elif actor == "Soyo" and instrument == "Bass":
13          raise NotImplementedError
```

# 过程式编程的问题二：难以阅读和维护

- 当一个script里堆积了很多过程式函数，每个过程式函数堆积了很多分支。。。

```
def play(actor: str, instrument: str, song: str):
    if actor == "Lena" and instrument == "Guitar" and song == "Haruhikage":…
    elif actor == "Soyo" and instrument == "Bass":…
    elif actor == "Lena" and instrument == "Piano":…
    elif  actor == "Lena" and instrument == "Guitar" and song == "Mayoiuta":…
    elif actor == "Tomori" and instrument == "Angle iron":…
    elif actor == "Taki" and instrument == "Drum":…
```

# Why OOP thinking is your need?

- 在大型项目中，会有许多procedures（变量声明、执行运算操作、传递值...）。并且，随着项目扩展，这些procedures可能会指数级变多。
- 然而，objects，或者说整个项目的“演员”， 通常是有限的。比如，MyGO!!!!!总共有五个乐队成员。
- 因此，我们在编写程序的时候，首先考虑的是这些对象的属性和功能，而procedures无非是这些属性和功能的组合。

# How to organize the code?
# 情况一： Procedural Programming的恶果

- Scene 1:Tomori（念白）
- Scene 2: Lena（吉他前奏）
- Scene 3: Sakiko（run）
- Scene 4: Soyo（yell）

```
Tomori.monologue()
Lena.play()
Sakiko.run()
Soyo.yell()
```

- 每一个过程都是以实例化的对象为中心的

# Python OOP的核心：class（类）
# 理解类的属性、方法（以及之前提到的，实例化）

oop_haruhikage.py > ...

```python
class Student:

    def __init__(self, age: int, name: str, school_name: str, vocal: bool, instrument: str):
        self.age = age
        self.name = name
        self.school_name = school_name
        self.vocal = vocal
        self.instrument = instrument
    def self_introduction(self):
        print(f"大家好，我的名字是 {self.name}，今年 {self.age} 岁，我在 {self.school_name} 学习。")
    def sing(self):
        if self.vocal == True:
            print(f"{self.name} 担任主唱。")
        else:
            pass
    def monologue(self):
        raise NotImplementedError
    def talk(self):···
    def play(self):
        print(f"{self.name} 正在用 {self.instrument} 演奏音乐。")
```

# Python OOP特性一：继承

- 比如，定义mygo_member是Student的子类，从Student继承self.school_name属性，可以在自身方法中直接调用

```
30
31  class mygo_member(Student):
32      def is_from_羽丘女子学園(self):
33          print(self.school_name=="羽丘女子学園")
34
35  Tomori = mygo_member(age = 16, name = "Takamatsu Tomori", school_name = "羽丘女子学園")
36  Tomori.is_from_羽丘女子学園()
37  # True
```

# Python OOP特性二：多态

- 对于不同的子类实例，我们可以调用同一个函数(let_play()），因为它们有共同的方法(play())，这是容易直观理解的。

```
42  Tomori = mygo_member(age = 16, name = "Takamatsu Tomori", school_name = "羽丘女子学園", instrument = "Angle iron")
43  Mutsumi = pre_crychic_member(age = 16, name = "Wakaba Mutsumi", school_name = "月之森女子學園", instrument = "Guitar")
44
45  def let_play(student):
46      student.play()
47
48  let_play(Tomori)
49  # Takamatsu Tomori 正在用 Angle iron 演奏音乐。
50  let_play(Mutsumi)
51  # Wakaba Mutsumi 正在用 Guitar 演奏音乐。
```

# Python OOP特性三：组合

- 我们可以将不同的类组合成一个大类，比如Sakiko作为人类有作为student和musician的两种身份，可以将将这两个class组合成一个大class。如果更改其分别的属性和方法，不影响组合的逻辑。

```python
class Human:
    def __init__(self, age, name, school_name, vocal, instrument, style):
        self.as_student = Student(age, name, school_name, vocal, instrument)
        self.as_musician = Musician(style)

sakiko = Human(age=16, name="Togawa Sakiko", school_name="羽丘女子学園", vocal=False, instrument="Piano", style="Symphonic Metal")
print(sakiko.as_student.name)
# Togawa Sakiko
print(sakiko.as_musician.style)
# Symphonic Metal
```

# 多重继承

```
45  class pre_crychic_member(Student):
46      def is_from_羽丘女子学園(self):
47          print(self.school_name=="羽丘女子学園")
48      def is_crychic_dismissed(self):
49          return False
50  class current_mygo_member(pre_crychic_member):
51      def is_from_羽丘女子学園(self):
52          print(self.school_name=="羽丘女子学園")
53      def is_crychic_dismissed(self):
54          return True
55
56  Soyo = current_mygo_member(age=16, name="Nagasaki Soyo", school_name="月之森女子學園", instrument="Bass")
57  print(Soyo.is_crychic_dismissed())
58  # True
```

# 方法解析顺序（MRO)

- 使用类的__mro__内置函数来查看方法解析顺序，在本例子中，current_mygo_member类首先查看自身的函数，如果找不到再从pre_crychic_member中查询，再找不到，到Student类中查询。

```
print(current_mygo_member.__mro__)
# (<class '__main__.current_mygo_member'>, <class '__main__.pre_crychic_member'>, <class '__main__.Student'>)
```

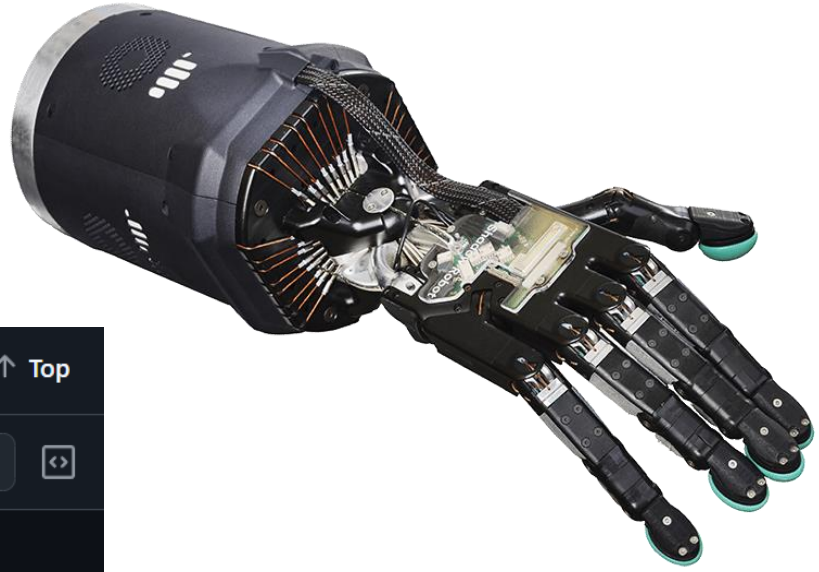
# super()内置函数：确保调用父类（如果项目需要）

```
45  class pre_crychic_member(Student):
46      def is_from_羽丘女子学園(self):
47          print(self.school_name=="羽丘女子学園")
48      def is_crychic_dismissed(self):
49          return False
50  class current_mygo_member(pre_crychic_member):
51      def is_from_羽丘女子学園(self):
52          print(self.school_name=="羽丘女子学園")
53      def is_crychic_dismissed(self):
54          super().is_crychic_dismissed
55          return True
56
57  Soyo = current_mygo_member(age=16, name="Nagasaki Soyo", school_name="月之森女子學園", instrument="Bass")
58  print(Soyo.is_crychic_dismissed())
59  # True
60  # False
```

# 最简单的游戏战斗系统也离不开OOP实现

- hero.HP
- hero.Attack
- hero.yellow_key

# 具体科研中的OOP（机器手模型）



BimanGrasp-Dataset / hand_model.py ↑ Top

Code Blame Executable File · 235 lines (201 loc) · 11.8 KB Raw

```
13
14  class HandModel:
15      def __init__(self,
16                   mjcf_path,
17                   mesh_path,
18                   contact_points_path,
19                   penetration_points_path,
20                   n_surface_points=0,
21                   device='cpu',
22                   handedness=None):
23
24          self.device = device
25          self.handedness = handedness
26
27          # load articulation
28
29          self.chain = pk.build_chain_from_mjcf(open(mjcf_path).read()).to(dtype=torch.float, device=device)
30          self.n_dofs = len(self.chain.get_joint_parameter_names())
```

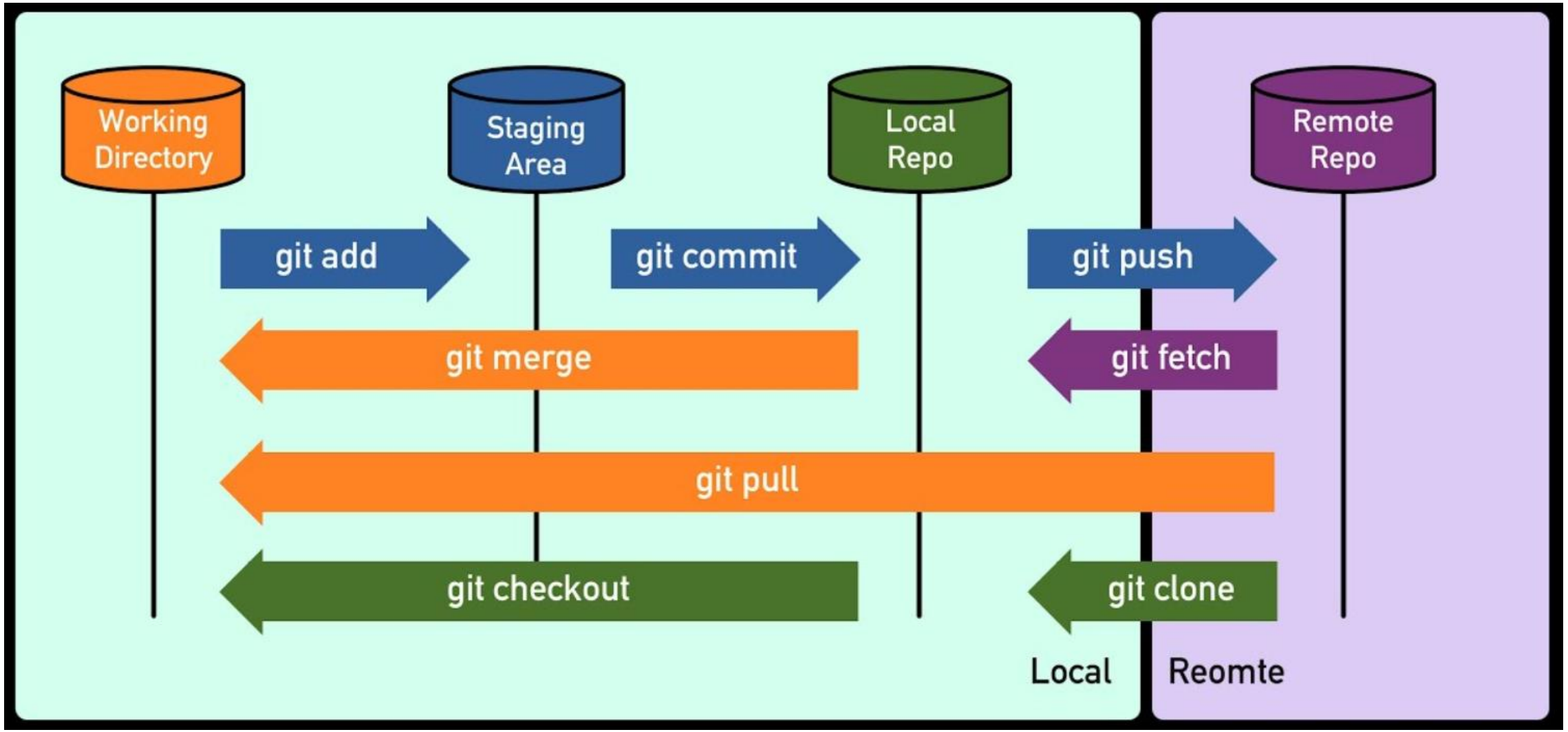
# Git初步

Git：一个（~~代码~~）版本管理工具

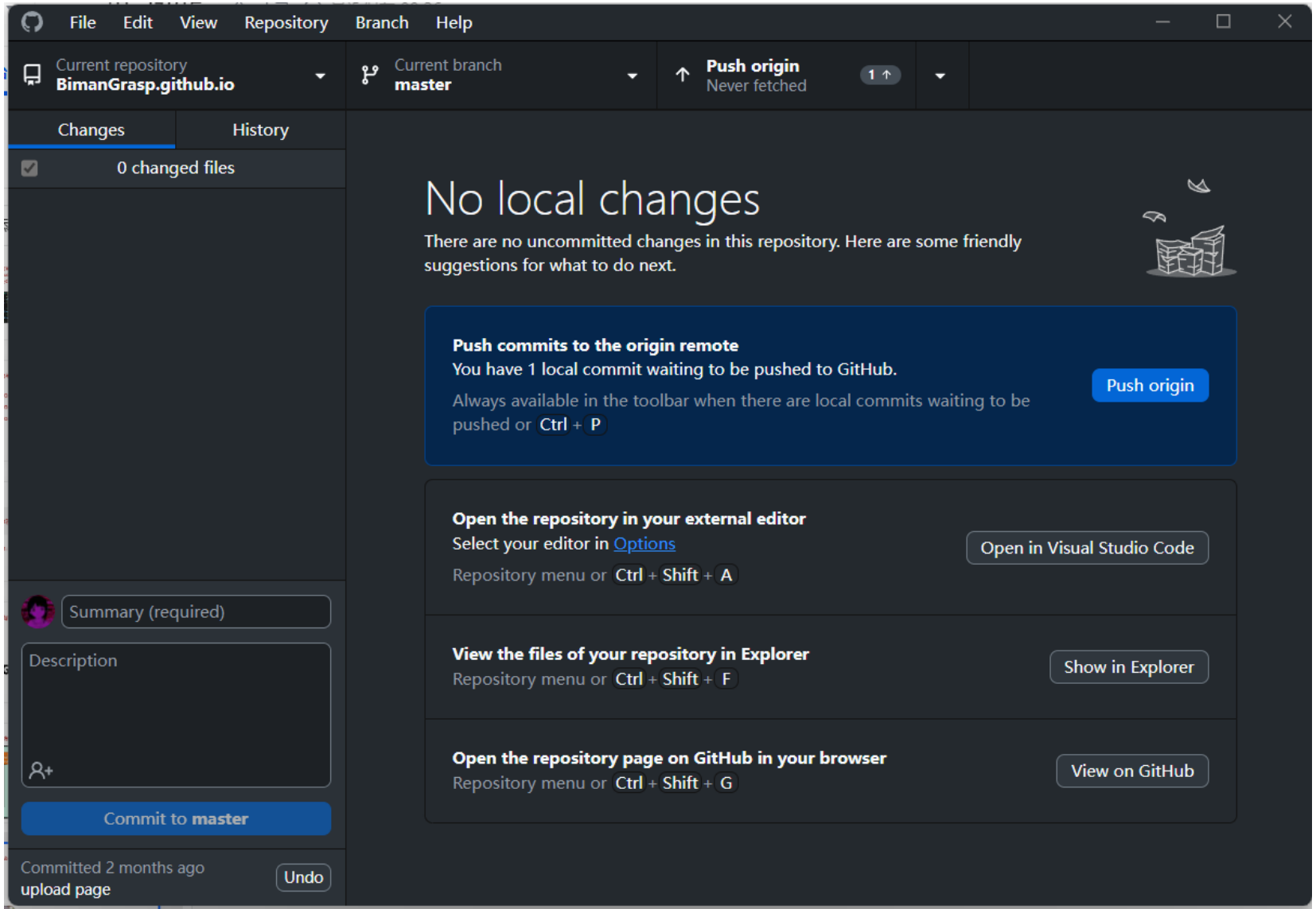
# 使用git的工具



# · 分布式（本地和远程服务器的关系）



# github desktop （我最爱用，有手就行）



# 基本的git命令

pull: 把远程仓库的更新拉到本地工作空间
push：把本地仓库的更新推到远程仓库

git add . & git commit：将全部工作空间更新到本地仓库

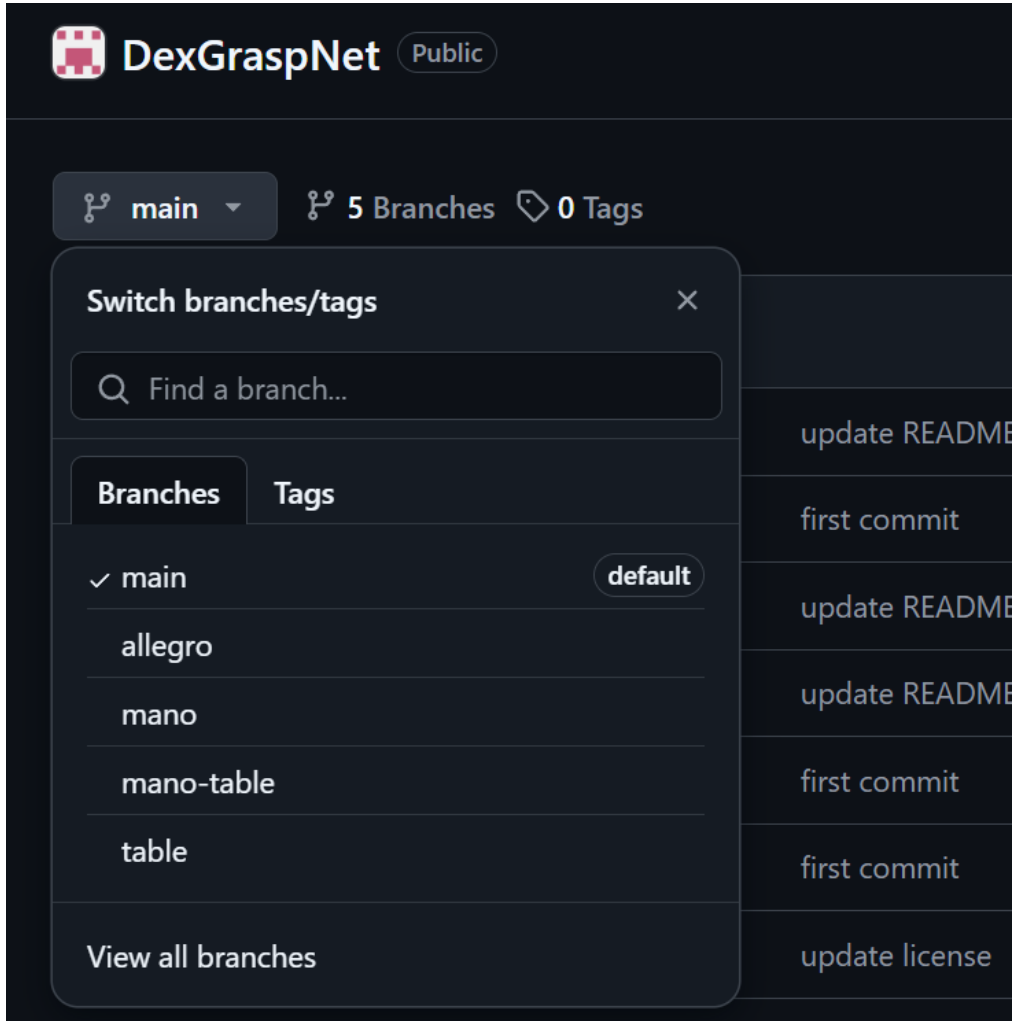
# 分支和checkout

- git checkout <branch-name>

切换到所需分支

- git checkout -b <new-branch-name>

创建新分支，并切换到它



# 版本控制（多人协作）

- git diff

比较版本差异

git diff <file-path>

git diff <branch1>..<branch2>

# PR和外部协作


awesome-humanoid-manipulation
Public
Unpin
Unwatch 1
Fork 1
Starred 21

# 其他常用工具

- cursor/ copilot