问题

异步编程和多进程多线程的目的分别是什么？有什么异同点？
多进程和多线程一定要基于函数吗？可不可以是其他形式？
进程和线程的区别是什么？
异步函数的基本结构和组件是什么？
多进程的基本结构和组件是什么？
多线程的基本结构和组件是什么？
设置的启动方式 spawn 等有什么区别？

Python 里的 async 异步函数里一定要有 await 关键字吗？如果没有，使用什么来表示任务的切换？
协程在本任务等待时执行另一个任务时，是在本任务等待结束后就跳转回本任务，还是等待另一个任务执行结束后再跳转回本任务？
协程的处理器是指线程还是进程？

数据传递过程中建立的数据结构相对于各个进程是什么关系？在代码结构上又是什么样的关系？
数据结构 queue 和普通的 queue 有什么异同点？

线程锁的设置时机是什么时候？创建线程锁安全对象又是什么意思？
Manager 管理对象是什么？具体是如何工作的？

进程池和线程池是什么，有哪些函数映射方式？
　　线程池进程池好普通的多进程多线程单独创建然后，start（），join()启动，等待，有什么异同点

创建进程池/线程池时，应该使用独立的函数还是类中的方法？进程池中的函数参数又该是独立的函数还是类中的方法？

　　进程就需要明确进程的创建，终止（terminate 和 kill 的区别），启动事件（什么时候创建，什么时候启动，启动之后对别的代码和事件本身各有什么影响）和通信队列（如何创建，如何实现通信，如何获取，如何保证数据的安全）。

异步编程

异步编程基本概念和实例

异步编程是现代编程中一种重要的技巧，指的是“同时”执行多个任务，与之对应的是同步编程，指的是按照顺序依次执行多个任务。异步编程需要注意区分多线程和多进程，异步编程本质还是只有一个处理器，只是这个处理器会交替执行不同的任务，比如遇到 IO 等待时间，这个处理器会利用空闲的时间去执行其它任务（具体如何交替则是内部算法实现的）。结束之后再执行原来的语句（这里是执行完这个语句再返回去执行还是直接返回去执行）。

异步编程又称协程，它不是计算机提供的，而是由程序员人为创造的。也称为维线程，是一种用户态内的上下文切换技术，是利用 1 个线程实现代码块切换执行的。

举例来说，一般代码块是按照顺序执行的，比如下面两个函数：

```
func1():
    print(1)
func2():
    print(2)
func1()
func2()
```

此时两个函数是按照顺序执行的，执行完 func1 之后再执行 func2，而协程则是可以允许代码先执行 func1 的部分，然后跳转到 func2 执行部分内容，再跳回到 func1，再跳回到 func2，不断切换，直到完成全部代码（所以关键就是如何判断切换时机）。

那么接下来就应该学习一下在编程上具体是如何实现的了。

结合上面的例子，异步编程应该包括以下几个核心部分，一是需要执行的任务，任务中有需要“等待”的语句；二是一个循环池，这个池子会控制哪个任务该执行，哪个任务在等待。

Python 中的 asyncio 模块提供了一种方便的方式来实现异步编程。

异步函数：模块将待执行的任务，前面用关键字 async 表示，在需要等待的语句前面再加上 await 关键字，执行到这个语句时，就会挂起协程。

协程对象（Coroutine）：未激活的异步操作代码块

任务对象（Task）：已加入事件循环调度的协程

事件循环：异步任务调度器，可以理解一个为一个死循环（类似大池子），去不断地轮询检查任务列表中的任务状态，任务有三种状态，就绪态（当前可执行），堵塞态（等待中，不可执行），完成态（任务已完成）；

事件循环是异步编程的核心，负责调度和执行协程。而协程对象是需要被事件循环处理的任务单元。任务对象是对协程对象的封装，事件循环必须在协程运行前存在，协程对象需要被事件循环调用才能执行。

创建顺序上，通常先定义异步函数，将异步函数先转化为协程对象继而封装为 task 对象，然后投入到事件循环中。

下面是一个精简的异步编程实例：

```
import asyncio
async def func1():
    print(1)
    await asyncio.sleep(2) #  模拟网络 IO
```

```
    print(2)
async def func2():
    print(3)
    await asyncio.sleep(2) #  模拟网络 IO
    print(4)
async def main(): #  创建主协程包装任务
"""主协程：负责创建和管理所有任务"""
#尽可能在主协程函数中创建任务，不能在同级代码下创建
    task1=asyncio.create_task(func1())
    task2=asyncio.create_task(func2())
    await task1
    await    task2
#  使用现代标准方式启动
if __name__ == "__main__":
        asyncio.run(main()) # Python 3.7+推荐方式
```

综上，现代方式创建一个基本的异步编程实例需要下面几步：

（1）aysnc 关键字创建异步函数，这是程序运行的主体；直接调用协程函数不会执行，而是返回协程对象，必须通过 await 或者事件循环来运行。

（2）在异步函数里需要 io 等待或其它单独耗时等待而不需要处理器执行的语句前添加 await 关键字作为切换开关；

（3）创建主协程函数，在其中创建协程对象并将其投入到任务列表 tasks 里，然后任务列表在事件循环里运行；Task **运行（非创建）**之前要先创建事件循环。

（4）使用现代标准方式启动事件循环；

异步编程问题探究

协程对象创建方法

协程对象的创建通过定义 async def 异步函数，并调用该函数来生成协程对象。比如

```
import asyncio
async def my_coroutine():
    await asyncio.sleep(1)
coro = my_coroutine() #   # 注意调用异步函数不会和传统的函数一样执行，而是返回协程对象
```

这时 coro 就是一个协程对象，尚未执行。

任务对象创建方法 task

协程对象需要被包装成任务（Task）以便并发执行，

asyncio.create_task()：create_task 必须在事件循环已经运行的情况下调用，或者在协程内部调用，避免在全局作用域创建 Task，推荐定义一个主协程函数

在里面使用 asyncio.create_task()。如果当前没有运行的事件循环，直接调用 create_task 会抛出错误，因为此时事件循环尚未启动，无法创建任务。

```
async def async_main():
    coro = my_coroutine() # 协程对象（Coroutine）
    task = asyncio.create_task(coro) # 转换为任务对象（Task）
    print(type(task)) # 输出：<class '_asyncio.Task'>
    await task
asyncio.run(main())
```

此外还有 1 种兼容旧版本的方法，简单介绍一下。不推荐使用循环的方法显式实例化任务对象。

```
asyncio.ensure_future()（兼容方式）
async def legacy_approach():
    coro = my_coroutine()
future = asyncio.ensure_future(coro) # 可能返回 Task 或 Future
print(type(future)) # 输出：<class '_asyncio.Task'>
await future
```

事件循环创建方法

Python 中事件循环的创建通常通过 asyncio 模块，特别是从 Python 3.7 开始推荐使用 asyncio.run()，它会自动创建和管理事件循环。而之前可能需要显式获取循环，比如使用 asyncio.get_event_loop()。另外，高级用户可能需要自定义事件循环，比如使用 uvloop，但标准库中通常用默认的即可。

旧版代码

```
loop = asyncio.get_event_loop()，
loop.run_until_complete(aync_main())
```

新版推荐 asyncio.run(aync_main())。

在 asyncio 中，所有未显式指定事件循环的异步操作（包括库函数和自定义函数），都会使用当前线程的 “默认事件循环”

await 关键字

await 关键字主要用于等待一个可等待对象（Awaitable）完成任务，这包括协程（coroutine）、任务（Task）和未来对象（Future）等。 await 只能用在异步函数内部。异步函数内部使用的 await asyncio.sleep(1) 等待的就是协程对象执行完毕。

await+协程对象:立即开始执行该协程并挂起当前协程，让出控制权给事件循环，直到被等待的协程完成。协程对象在被 await 之前，并没有被调度到事件循环中。直接 await 一个协程对象，那么它是顺序执行的，不会并发。

await+任务对象：等待的是任务对象完成。而任务对象在创建时就已经被调度到事件循环中，它可能已经在运行中，或者即将被运行。Await 只是在等待结果。

下面通过一个例子来说明： 假设有两个异步函数 foo 和 bar，我们想要并发执行它们。

错误的方式（顺序执行）：

```
await foo()

await bar()
```

正确的方式（并发执行）：

```
task1 = asyncio.create_task(foo())

task2 = asyncio.create_task(bar())

await task1

await task2
```


在并发执行的情况下，foo 和 bar 会同时运行，总时间约为运行时间较长的那个。而顺序执行的总时间为两者之和。

在 asyncio 中，await 用于等待一个任务（task），而等待一个任务列表通常使用 `asyncio.wait` 或 `asyncio.gather`，它们可以同时等待多个任务，并可以选择等待策略（例如，等待所有任务完成，或等待第一个任务完成）。

await task：会暂停当前协程，直到该任务完成。然后返回任务的结果（如果任务有异常，则会抛出异常）。

asyncio.wait： 返回一个元组 (done, pending)，其中 done 是已完成的任务集合，pending 是未完成的任务集合，需要手动从每个任务中获取结果。可以指定等待条件（例如，等待所有任务完成或第一个任务完成）。

asyncio.gather:会等待所有任务完成,并返回一个结果列表(按输入顺序)。如果任何一个任务异常，它会立即抛出异常，但你可以设置 return_exceptions=True 来将异常作为结果返回。

多进程

多进程多线程的核心是什么，如何把不同的任务分配到不同的进程和线程上，以及如何操作资源并获得收集正确的结果。

基本概念

进程是**程序的一次完整执行过程**，是操作系统进行资源分配（如内存、CPU 时间片、文件句柄）的基本单位。可以理解为：一个运行中的程序（如浏览器、微信、Python 脚本）就是一个或多个进程。

多进程是操作系统中管理程序执行的一种机制，指的是系统中同时运行多个独立的进程（进程是程序的一次执行过程，包含独立的内存空间、资源分配和执行状态）。这些进程可以并发（分时）或并行（真正的同时）执行，通过操作系统的调度机制共享硬件资源，从而提高系统效率和资源利用率。

CPU 核心是硬件层面负责执行指令的计算单元，多进程的运行依赖 CPU 核心的调度，二者关系体现在“并发”与“并行”的实现上：

单 CPU 核心场景：单核心只能物理上同时执行一个进程的指令。多进程的“同时运行”是通过操作系统的“时间分片”调度实现的 —— 快速切换不同进程的执行权（比如每毫秒切换一次），从宏观上看多个进程在同时运行（这称为“并发”）。

多 CPU 核心场景：多个核心可以物理上同时执行不同的进程（这称为“并行”）。例如 4 核 CPU 理论上可同时运行 4 个进程，操作系统会将进程分配到不同核心上，减少切换开销，大幅提升多进程的执行效率。

基本用法

了解了多进程的基本概念之后，接下来要学习它的用法，首先介绍一下创建多进程最基础的方法，通常用 multiprocessing 模块。

多进程核心步骤如下：
1.创建执行函数；def calculate(num)
2.创建进程对象，2 个关键参数，target 赋值该进程要运行的函数名，不带括号，args 赋值运行函数的参数
p1 = multiprocessing.Process(target=calculate_square, args=(numbers[:2], result) # 切片分配任务)
3.启动进程；p1.start()
4.等待进程结束 p1.join()

多进程的创建和调度完全由操作系统自动管理：

进程创建时，系统自动分配资源；

单核心CPU多进程运行时，系统通过时间分片自动切换进程，实现“并发”效果。

用户或程序只需发起创建进程的请求（如运行多个程序），无需关心底层的资源分配和调度细节

代码如下：
案例 1：CPU 密集型任务（计算平方和）

```
import multiprocessing
#创建待执行函数
def calculate_square(nums, result_queue):
"""计算平方并存入队列"""
    for n in nums:
        result_queue.put(n**2) # int 类型数据入队
if __name__ == '__main__':
    numbers = [1, 2, 3, 4] # list[int]待处理数据
    result = multiprocessing.Queue() # 进程安全队列
# 创建两个进程
    p1 = multiprocessing.Process(target=calculate_square,
            args=(numbers[:2], result) # 切片分配任务)
    p2 = multiprocessing.Process(target=calculate_square,
            args=(numbers[2:], result)
    p1.start() # 启动进程 1
    p2.start() # 启动进程 2
    p1.join() # 等待进程 1 结束
    p2.join() # 等待进程 2 结束
# 从队列提取结果
    while not result.empty():
        print(result.get()) # 输出结果顺序可能不固定
"""
输出示例（顺序可能变化）：
1 4 9 16
"""
```

多进程工作流：

```
主进程
├─ 创建子进程 1 → 独立内存空间 → 执行任务 → IPC 通信 → 结果返回
├─ 创建子进程 2 → 独立内存空间 → 执行任务 → IPC 通信 → 结果返回
└─ 等待所有子进程终止 → 汇总结果
```

进程创建位置 `if __name__ == '__main__'`下

在 Python 中，if __name__ == ’__main__’ 是一个特殊的代码块，其作用与多进程（尤其是 spawn 启动方式）的正常工作密切相关。如果将进程启动代码放在这个代码块之外，可能会导致意想不到的错误，尤其是在跨平台场景中。

一、if __name__ == '__main__' 的作用

__name__ 是 Python 的内置变量，用于标识当前模块的运行状态：

- 当模块**直接被运行**时（如 python script.py），__name__ 的值会被设为 '__main__'，此时 if 条件成立，代码块内的内容会执行。
- 当模块**被导入**时（如 import script），__name__ 的值会被设为模块名（如 'script'），此时 if 条件不成立，代码块内的内容不会执行。

因此，这个代码块的核心作用是：**区分模块的 “直接运行” 和 “被导入” 两种状态**，确保某些代码（如程序入口、测试代码、进程启动逻辑）仅在模块直接运行时执行，而在被导入时不执行。

二、不将进程启动代码放入该代码块的问题

这个问题在使用 spawn 方式创建进程时尤为突出（Windows 系统默认用 spawn，Unix 系统也可手动指定），具体表现为：**无限递归创建进程，最终导致程序崩溃**。

原因如下：

1. spawn 方式创建子进程时，会启动一个全新的 Python 解释器，并**重新导入父进程的整个模块**（而非复制内存，这与 fork 不同）。
2. 如果进程启动代码（如 Process(target=xxx).start()）不在 if __name__ == '__main__' 内，那么子进程在导入模块时，会再次执行这段启动代码，从而创建新的子进程。
3. 新的子进程导入模块时又会重复这个过程，形成无限递归，最终耗尽系统资源导致崩溃。报错信息可能包含 RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase。

启动方法

在多进程编程中，fork、spawn 等是不同的**进程创建方式（启动方法）**，决定了新进程如何从父进程初始化、继承资源以及与父进程的关系。

启动模式主要用于子进程继承父进程的继承方式的，而不同的系统（Linux，Windows 等）支持的模式是不同的，因此需要显示声明。主要有 fork，spawn，forkserver 三种模式。下面是它们的对比：

启动模式	工作原理	支持平台	特点	适用
fork	直接复制父进程内存空间（使用 os.fork()）-完全继承	Unix 系统	快速，但可能继承不安全状态；多线程环境下危险	性能要求高时
spawn	启动新 Python 解释器，重新导入主模块——不继承	所有平台	安全，较慢；Windows/macOS 默认方式	**跨平台，多线程+多进程**
forkserver	预先启动服务器进程，后续进程从该服务器 fork-部分继承	Unix 系统	平衡安全性和性能；避免重复初始化资源	需要创建大量进程

模式的声明语句必须在程序最开头设置，在任何进程创建之前，如下：

```
if __name__ == '__main__':
    # 只能设置一次，且必须在创建任何进程对象前
    mp.set_start_method('spawn')  # 或 'fork'/'forkserver'
    # 后续进程创建
    p = mp.Process(target=worker)
    p.start()
```

多线程不需要设置启动方式，原因如下：

线程共享相同内存空间，创建方式统一

所有平台使用相同线程创建机制（通过操作系统 API）

Python 的 threading 模块不提供启动方式配置

```
import multiprocessing as mp
import osimport time
def worker():
    print(f"[Child] PID: {os.getpid()}, Data: {shared_list}")
if __name__ == '__main__':
```

```
    # 尝试不同启动方式观察行为差异
    # mp.set_start_method('fork')       # Unix only
    # mp.set_start_method('spawn')      # Default on Win/Mac
    # mp.set_start_method('forkserver')  # Unix only
    shared_list = [1, 2, 3]  # 父进程数据
    p = mp.Process(target=worker)
    p.start()
    p.join()
    print(f"[Main] PID: {os.getpid()}")
```

不同启动方式下的行为差异：

fork 方式 (Unix)：

[Child] PID: 1234, Data: [1, 2, 3] # 继承父进程状态

[Main] PID: 1233

spawn 方式 (跨平台)：

[Child] PID: 1234, Data: [] # 新进程，不继承状态

[Main] PID: 1233

forkserver 方式 (Unix)：

[Forkserver] Started PID: 1235

[Child] PID: 1236, Data: [] # 从服务器进程 fork

[Main] PID: 1233

启动事件

启动事件的位置，包括 set，clear 等，可以放到执行函数里吗？

创建好多个进程后，有些情况下需要这些进程函数同时工作，可以设置一个发令枪一样的东西，这就是多进程的启动事件。

启动事件是一个同步对象，用于在不同进程间传递状态信息，内部维护了一个布尔标志属性，默认为 false。

创建方式：1.event = multiprocessing.Event()

2.multiprocessing.Manager().Event()

事件对象通常在父进程（一般在主函数）中创建，然后作为参数传递给子进程函数。这样，父子进程就可以通过同一个事件对象进行同步。启动事件（Event）的 set、wait，clear 等方法可以放在主函数或者执行函数（即子进程的目标函数）内部。

启动事件是一个对象，相当于是一个秒表，这个对象可以在父进程中创建，并作为参数传递给多个子进程（实际是进程函数），这样所有的父子进程都有了这个秒表，并根据秒表的状态来决定是否执行进程函数。而这个秒表提供了几个按钮，一个按钮为 set 方法，所有进程都可以按下这个按钮（执行 set 方法），按下这个按钮后，所有拥有这个秒表的进程的状态标志都变为可执行，因此在满足特定条件后可以按下按钮；另一个按钮为 wait，按下这个按钮的进程会被堵塞（等待状态标志变为 true，若已为 true，则直接执行）因此在需要等待某个条件满足的代码前面按下这个按钮，一个进程可以有多个秒表？

核心方法有：

启动事件 set（）：通常在某个进程（可能是父进程，也可能是子进程）中，放在条件满足后需要通知其他进程的代码位置。当某个条件满足时，调用 event.set()将事件标志设为 true，并通知所有使用了 wait 方法等待此事件的进程。

等待事件 wait(timeout=None)：放在需要等待事件发生的代码位置之前。调用 event.wait()后，该进程会被阻塞，直到事件被设置（即标志变为`True`）。如果调用时标志已经是 True，则立即返回。也可以设置超时时间。

set() 和 wait() 不一定必须结合使用：可以单独使用 `set()` 来标记状态,可以单独使用 `wait()` 来等待可能已设置的事件,但它们通常是为了实现进程间同步而配合使用的。

重置事件 event.clear（）：将事件标志重置为 false,使其回到未设置状态，以便重复使用。

简单代码实例

```
import multiprocessing as mp
import time
def worker(event, id):
```

```
    print(f"Worker-{id} waiting to start...")
    event.wait()  # 阻塞直到事件被设置
    print(f"Worker-{id} started at {time.time():.2f}")
if __name__ == '__main__':
    # 创建事件对象 (一般在主进程)
    start_event = mp.Event()
    # 创建并启动工作进程
    processes = []
    for i in range(3):
        p = mp.Process(target=worker, args=(start_event, i))
        p.start()
        processes.append(p)
    # 模拟初始化时间
    print("Main process initializing resources...")
    time.sleep(2)
    # 设置事件唤醒所有工作进程 (关键位置!)
    print("Main process setting event at", time.time())
    start_event.set()  # 所有等待的 worker 同时唤醒

    # 等待子进程结束
    for p in processes:
        p.join()
    print("All workers completed")
```

数据传递-IPC 机制（进程间通信）

数据传递过程中建立的数据结构相对于各个进程是什么关系？在代码结构上又是什么样的关系？在主函数下实现，并作为进程函数的参数传入参数

进程间的数据传递核心是借助管道 queue 等中介结构（这个数据结构是独立的进程吗？）作为不同进程的待执行函数的参数，这样每个进程都可以访问这个数据结构，以上面的简单实现来看，

必须使用进程间共享数据结构如共享队列，不能使用普通的数据结构；共享数据结构的创建位置在主函数下，并作为进程函数的参数传入；创建方式又有多种，如上面的代码，

第一种是多进程模块的队列对象（与创建进程对象同级），还可以设置最多传入的数据数量；

```
shared_queue = multiprocessing.Queue(maxsize=3) #  进程安全队列
```

第二种是代理对象 Manager，

```
manager=Manager()
shared_queue=manager.Queue()   #法 1 创建共享队列
```

进程间的数据传递核心是借助管道 queue 等中介结构作为不同进程的待执行函数的参数，这样每个进程都可以访问这个数据结构，以上面的简单实现来看，

下面是几种不同的数据结构作为信息传递的媒介的特点。

方式	实现方法	特点	适用场景
队列(Queue)	multiprocessing.Queue()	- 进程安全- 自动序列化数据- 支持多生产者/消费者	结构化数据传输
管道(Pipe)	multiprocessing.Pipe()	- 双向通信- 需要手动管理连接端- 高性能但易出错	低延迟简单通信
共享内存(Value/Array)	multiprocessing.Value('i', 0)multiprocessing.Array('i', [0]*10)	- 直接内存操作- 需配合锁使用- 只支持基础数据类型	高频小数据量操作
Manager 代理对象	multiprocessing.Manager().dict()	- 支持复杂数据结构- 网络透明访问- 性能较低	跨机器分布式场景

底层实现示意图

class Queue:

def __init__(self):

self._pipe = Pipe() # 底层使用双向管道

self._rlock = Lock() # 读锁

self._wlock = Lock() # 写锁

self._sem = Semaphore(0) # 信号量计数器

Manager

Manager`是一个用于创建共享数据和共享状态的对象，它允许不同的进程之间安全地共享数据。`Manager`对象控制一个服务器进程，该进程持有共享对象，并允许其他进程通过代理来操作这些共享对象。下面主要是和 multiprocessing 自带的共享数据结构，启动事件做比较。

multiprocessing.Queue 是 “类”，用 Queue() 直接创建实例；manager.Queue() 是 “方法调用”，通过管理器对象的方法间接创建队列实例，二者适用的进程通信场景不同。

共享数据结构 queue

对比维度	multiprocessing.Queue	manager.Queue()
实现方式	基于管道（Pipe）和锁 / 信号量实现，是独立队列	由 Manager 服务进程持有真正队列，其他进程通过代理访问
进程间通信范围	适用于父进程与子进程、同父进程的子进程之间通信	支持同一机器多进程通信，可配置为跨网络计算机进程间通信
支持的数据类型	仅队列本身	除队列外，还支持共享列表、字典等多种数据结构
性能和开销	直接通过管道通信，速度快，开销小	需通过代理通信，存在序列化和网络通信开销，速度较慢
适用场景	仅需在多个进程间共享队列时使用	需要共享复杂数据结构或跨网络共享时使用
生命周期	与创建它的进程关联，创建进程结束后可能不可用	由 Manager 进程管理，只要 Manager 未结束则可用

实现方式：multiprocessing.Queue 是一个独立的队列，它是使用管道和锁/信号量实现的，可以在进程间直接使用。

manager.Queue()是通过一个 Manager 对象创建的，Manager 负责管理一个服务进程，该进程持有真正的队列对象，其他进程通过代理来访问这个队列。

进程间通信：multiprocessing.Queue 通常用于父进程和子进程之间或者同父进程的子进程之间的通信。

manager.Queue()可以在网络上的不同计算机的进程间共享（如果 Manager 被配置为监听网络），但通常也是在同一台机器的多个进程间使用。Manager 提供了更多共享数据类型（如列表、字典等）的支持。

性能和开销：multiprocessing.Queue 由于是直接使用管道和锁，所以速度相对较快。manager.Queue()` 因为通过代理进行通信，会有更多的序列化和通信开销，所以速度较慢。

使用场景： - 如果只需要在多个进程间共享一个队列，使用 `multiprocessing.Queue`即可。- 如果需要共享多个复杂的数据结构(如列表、字典等）或者需要跨网络共享，则使用 Manager。

生命周期： multiprocessing.Queue 的生命周期与其所在的进程相关，如果创建它的进程结束了，那么队列可能无法被其他进程使用。 manager.Queue()的生命周期由 Manager 进程管理，只要 Manager 进程没有结束，队列就可以被其他进程使用。

除了共享队列，multiprocessing 模块还提供了以下共享数据结构：

- multiprocessing.Array：共享内存的数组，可以存储指定类型的元素。

- multiprocessing.Value：共享内存的变量，可以存储一个指定类型的值。
- 通过 Manager 可以创建多种共享数据结构，包括：

  - Manager().list()：共享列表

  - Manager().dict()：共享字典

  - Manager().Namespace()：共享命名空间

  - Manager().SimpleQueue()

进程的终止

进程的 terminate()和 kill()方法都是通过向进程发送信号强制结束进程的方法，都会导致进程的非正常结束，因此可能会造成资源未释放等问题（比如

打开的文件未关闭，锁未释放等）。

区别：

1. p.terminate()：该方法会发送 SIGTERM 信号给进程。SIGTERM 信号是一个比较温和的终止信号，进程可以捕获这个信号并进行一些清理工作然后退出，当然进程也可以忽略这个信号。

2. p.kill()：该方法会发送 SIGKILL 信号给进程。SIGKILL 信号会立即终止进程，并且进程无法捕获或忽略这个信号。因此，这是一个强制终止的方式，不会给进程任何清理的机会。

使用场景：

- 如果你希望给进程一个机会去清理并正常退出，可以使用 terminate()。
- 如果进程无视 SIGTERM 信号，或者你需要立即结束进程，那么可以使用 kill()。

注意：

- 使用 terminate()或 kill()后，应该使用 join()或设置超时来等待进程实际结束，并检查进程是否已经终止。因为信号发送后，进程的终止可能需要一点时间。
- 强制终止进程可能会导致资源泄漏，因此应尽量避免，除非进程无法正常结束。

进程池

进程池（Process Pool）是多进程编程中用于管理和复用多个子进程的机制，它通过预先创建一定数量的进程，避免频繁创建和销毁进程的开销，从而提高多任务处理效率。

1.使用 pool 创建进程池，参数 processes 指定启动进程数；#基本一样
2.使用不同的映射方式将函数映射到不同的进程上；类似于单独创建时将函数绑定到进程上；#主要差异点
3.收集不同进程处理后的结果；

进程池的核心思想是：

> **预先创建**：程序启动时创建固定数量（或动态调整）的子进程，组成一个“池”。
>
> **任务复用**：当有新任务时，从池中分配一个空闲进程处理任务，任务完成后进程不销毁，继续等待新任务。
>
> **自动管理**：进程池负责进程的创建、调度、复用和销毁，无需手动管理每个进程的生命周期。

适用于**任务数量多、单个任务执行时间短**的场景（如批量数据处理、网络请求并发），避免频繁创建进程的性能损耗。

特性	线程池/进程池	手动创建线程/进程
创建方式	预创建一组工作单元	按需创建单个工作单元

资源管理	自动复用工作单元	每次任务都新建工作单元
任务分配	自动队列管理	需手动分配任务
并发控制	内置最大并发限制	需手动控制并发量
错误处理	集中处理异常	需单独处理每个异常
结果收集	统一结果接口	需自行实现收集机制
资源开销	固定开销，适合短任务	创建开销大，适合长任务
适用场景	I/O 密集型、短任务	CPU 密集型、长任务、特殊需求
代码复杂度	低（高级抽象）	高（需管理底层细节）

下面将着重介绍一下不同的映射方式，并通过实例说明。在 Python 的 multiprocessing.Pool 中，常用的映射方法包括：map, map_async, imap, imap_unordered, starmap, starmap_async。 注意：在 Windows 系统中，由于没有 fork，我们需要将代码放在 if __name__ == '__main__':中。

map 和 map_async(func, iterable[, chunksize]):

参数为执行函数和可迭代对象（如列表），这里的可迭代对象会被分割成若干块（每个块包含 chunksize 个元素）作为参数传递给 func 函数（因此注意这里 func 函数只能接受一个参数）。函数执行完毕后会返回结果列表，顺序和输入顺序一致。

进程池执行 map(func, iterable) 时，本质是将 iterable 中的每个元素作为独立任务分发给子进程。但如果 iterable 元素数量极多（如 10 万个）且单个任务耗时极短（如毫秒级），**频繁的 “单个任务分配 - 结果返回” 会产生大量进程间通信开销**（进程间数据传递需要序列化 / 反序列化）。

chunksize 的作用就是将 iterable 拆分为多个 “任务块”（每个块包含 chunksize 个元素），每个子进程一次性获取一个 “块” 并批量执行，执行完后再获取下一个块 —— 通过减少通信次数降低开销。

```
import multiprocessing
import time
def square(x):    #后面演示代码均使用此执行函数
    time.sleep(0.2 - x*0.01) # 数字越大执行越快
    print(f"处理: {x}") # 显示处理顺序
    return x * x
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool: # map 阻塞直到所有任务完成
        results = pool.map(square, range(10))
        print("主进程不可以继续执行其他任务...")
```

```
        print("map results:", results)
```
输出: map results: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

映射的阻塞版本，阻塞就是在执行完这一行代码之前（各进程完成返回结果之前），不会执行主代码的下面部分，即下面例子的 print()。适用于需要按输入顺序获取结果的简单任务

那如果待执行函数有多个参数怎么办呢？有 2 种方法，一是使用下面的 starmap 映射方法，二是使用 functools 的 partial 方法先固定为一个参数映射为新的函数，然后将新函数作为参数传递到 map 映射方法里。

map_async(func,iterable[,chunksize])

参数为待执行函数和可迭代对象，相比于上面的阻塞式调用，该映射方式为非阻塞异步调用，适合需要继续执行其他操作而不等待结果完成的场景，后台任务处理，需要获取结果时再等待。每个进程完成后都会立即返回 AsyncResult 对象，可以通过调用 AsyncResult.get()获取结果（这时会阻塞直到全部结果就绪），结果也是有序列表。

```
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool: #立即返回，不阻塞主进程
        async_result = pool.map_async(square, range(10))
        print("主进程可以继续执行其他任务...")
        time.sleep(0.2) # 模拟其他工作
        # 需要结果时调用 get()，会阻塞直到结果就绪
        results = async_result.get()
```

imap 和 imap_unordered

imap(func, iterable[, chunksize])

参数为待执行函数和可迭代对象，返回一个迭代器，逐个获取结果（按输入顺序）。与 map 的区别是：map 返回整个列表，而 imap 返回一个惰性迭代器，适用于处理大型数据集，可以边计算边获取结果。 chunksize 参数同上，但默认值为 1。

```
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool: # 返回按输入顺序排序的迭代器
    results_iter = pool.imap(square, range(10))
    print("开始获取结果:")
    for i, result in enumerate(results_iter, 1):
        print(f"获取结果 #{i}: {result}")
```

输出: 开始获取结果:
处理: 0 处理: 1 处理: 2 处理: 3
获取结果 #1: 0

处理: 4
获取结果 #2: 1

imap_unordered(func, iterable[, chunksize])

与 imap 类似，但返回的结果顺序不保证与输入顺序一致（按照完成顺序返回）。 当不关心顺序而希望尽快处理结果时使用。

```
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool: # 按完成顺序返回结果
    results_iter = pool.imap_unordered(square, range(10))
    print("开始获取结果 (按完成顺序):")
    for i, result in enumerate(results_iter, 1):
        value, squared = result
        print(f"获取结果 #{i}: {value}² = {squared}")
```

输出: 开始获取结果 (按完成顺序):
处理: 9
获取结果 #1: 9² = 81
处理: 8 获取结果 #2: 8² = 64

starmap 和 starmap_async

starmap(func, iterable[, chunksize])

与 map 类似，与 map 类似，但可迭代对象中的每个元素（有多个参数）。 适用于函数有多个参数的情况。适用于函数有多个参数的情况。

```
def power(base, exponent):
    return base ** exponent
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool:
    # 参数是元组列表，每个元组包含多个参数
    arguments = [(2, 3), (3, 2), (4, 2), (5, 3)]
    results = pool.starmap(power, arguments)
    print("starmap results:", results)
```

输出:
starmap results: [8, 9, 16, 125]

starmap_async(func, iterable[, chunksize])

starmap 的非阻塞版本，返回 AsyncResult 对象。处理多参数函数的异步调用。

```
if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool:
    arguments = [(2, 3), (3, 2), (4, 2), (5, 3)]
    async_result = pool.starmap_async(power, arguments)
    print("主进程执行其他任务...")
    # 等待结果就绪
    results = async_result.get()
```

```
print("starmap_async results:", results)
```

输出:

```
主进程执行其他任务...
starmap_async results: [8, 9, 16, 125]
```

方法对比总结

方法	阻塞	顺序	参数	返回类型	适用场景
map	是	输入顺序	单参数	列表	简单任务，需要顺序结果
map_async	否	输入顺序	单参数	AsyncResult	后台处理，不阻塞主进程
imap	部分	输入顺序	单参数	迭代器	大型数据集，流式处理
imap_unordered	部分	完成顺序	单参数	迭代器	尽快获取结果，不关心顺序
starmap	是	输入顺序	多参数	列表	多参数函数处理
starmap_async	否	输入顺序	多参数	AsyncResult	多参数函数的异步处理

使用建议

任务简单且需要顺序结果 → map
需要后台处理不阻塞主进程 → map_async
处理大型数据集需要流式获取结果 → imap
任务执行时间差异大，需要尽快获取结果 → imap_unordered
函数需要多个参数 → starmap 或 starmap_async
需要控制任务分块大小 → 使用 chunksize 参数优化性能

子进程 subprocess 模块

subprocess 是完全独立的新进程，执行外部程序和命令，和 multiprocessing 本质解决不同问题，但可在多进程编程中协同工作：

模块	核心目标	典型场景
multiprocessing	并行执行 Python 函数/方法	拆分计算密集型任务，利用多核 CPU 加速 Python 代码
subprocess	调用 外部程序/系统命令	执行非 Python 程序（如 Shell 命令、C++编译后的可执行文件等）

关键特性对比：

特性	subprocess	multiprocessing.Process
执行内容	外部程序/命令	Python 函数
进程类型		Python 解释器的子进程
数据交互	通过管道(PIPE)/文件	队列(Queue)/共享内存

错误处理 需捕获 CalledProcessError 通过异常传递到父进程
资源消耗 高（启动全新进程）　　中（共享 Python 解释器环境）

纯 Python 计算任务 → 仅用 multiprocessing
需要调用外部程序 → 在 multiprocessing 进程中嵌套 subprocess
需精细控制子进程 IO → 优先选择 subprocess.Popen
跨语言混合编程 → subprocess + 序列化协议（如 JSON）

subprocess 模块创建的子进程用于调用**外部程序和系统命令**，比如 webui 命令，用以告诉其它进程 UI 界面已启动；

1. **调用外部程序/命令**

需要运行系统命令（如 ls, grep）或其他语言编写的程序,并获取其输出，可以使用 *subprocess.run()*函数。例如，要运行 *ls -l* 命令并捕获其输出，可以这样做：

```
import subprocess
# 在 Python 进程中调用系统命令
result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
print(result.stdout)
```

*subprocess.run()*是 Python 3.5 中引入的一个函数，它执行指定的命令，并等待命令执行完成后返回一个包含执行结果的 CompletedProcess 类的实例。

如果需要更复杂的子进程管理，可以直接使用 *subprocess.Popen()*类。这个类提供了更多的控制选项，例如可以分别管理子进程的标准输入、输出和错误流。

2. **混合 Python 与外部进程**

主进程用 multiprocessing 管理并行，子任务中需要调用外部程序

```
from multiprocessing import Process
import subprocess
def run_external_tool(file_path):
# 在子进程中调用外部程序
    subprocess.run(["ffmpeg", "-i", file_path, "output.mp4"], check=True)
if __name__ == "__main__":
    files = ["video1.avi", "video2.mov"]
    processes = []
    for f in files:
    p = Process(target=run_external_tool, args=(f,))
    processes.append(p)
    p.start()
    for p in processes:
    p.join()
```

多线程

基本概念

多线程是操作系统中实现并发执行的一种机制，指在**同一个进程内创建多个线程**，这些线程共享进程的资源（如内存空间、文件描述符等），但拥有独立的执行路径和栈空间。

1. **资源共享**：同一进程内的所有线程共享该进程的内存空间、全局变量、打开的文件等资源，无需额外的通信机制即可访问共享数据（但需注意同步问题）。
2. **轻量级**：线程的创建、销毁和切换开销远小于进程（无需复制整个进程资源），适合频繁创建和销毁的场景。
3. **并发执行**：多个线程可在宏观上“同时”推进（单核心下通过时间分片实现，多核心下可真正并行）。
4. **统一生命周期**：线程依附于进程存在，进程终止时，其所有线程也会被强制终止。

特性	多进程 (Multiprocessing)	多线程 (Multithreading)
内存空间	每个进程独立内存空间	所有线程共享同一进程内存空间
资源开销	高（需复制完整上下文）	低（共享资源）
数据共享	需通过 IPC 机制（队列/管道）	可直接共享变量（需线程锁）
GIL 影响	完全规避 GIL 限制	受 GIL 限制（CPU 密集型任务并行性差）
适用场景	CPU 密集型任务（如数学计算）	I/O 密集型任务（如网络请求/文件操作）
错误隔离	进程崩溃不影响主进程	线程崩溃影响整个程序
创建方式	multiprocessing.Process	threading.Thread

基本用法

多线程核心步骤如下：

1.创建执行函数；def calculate(num)
2.创建线程，2 个关键参数，target 赋值该线程要运行的函数名，不带括号，args 赋值运行函数的参数 t1 = threading.Thread(
target=calculate_square_thread, args=(numbers[:2],))
3.启动线程；t1.start()
4.等待线程结束 t1.join()

可以看到，多线程的核心步骤与多进程几乎一致，差别主要在数据共享和锁机制上，后面再介绍。

```
import threading
results = [] #  共享列表（需同步控制）
lock = threading.Lock()
#创建待执行函数
def calculate_square_thread(nums):
    global results
    for n in nums:
        with lock: #  获取线程锁
            results.append(n**2) #  直接修改共享变量
numbers = [1, 2, 3, 4]
#创建线程
t1 = threading.Thread(target=calculate_square_thread, args=(numbers[:2],))
t2 = threading.Thread(target=calculate_square_thread, args=(numbers[2:],))
#启动线程
t1.start(); t2.start()
#等待线程结束
t1.join(); t2.join()
print(results) #  输出结果顺序可能混乱
"""
可能输出：
[1, 4, 9, 16]  或  [1, 9, 4, 16]
"""
多线程工作流：
主线程
├─ 创建线程 1 → 共享内存 → 执行任务 → 直接修改共享数据
├─ 创建线程 2 → 共享内存 → 执行任务 → 直接修改共享数据
└─ 等待所有线程结束 → 处理最终数据
```


数据传递方式-共享内存

多线程的数据传递方式使用普通的数据结构即可，但是需要注意的是，在不同线程操作处理共享数据结构前，需要先添加锁。如上面的简单实例：

创建普通的共享数据结构-列表

```
results = [] #  共享列表
```

（与线程函数同级，即线程函数可以不传参直接操作此数据结构）

在执行函数里操作数据结构前需要先获取线程锁：

```
with lock: #  获取线程锁
```

results.append(n**2) # 直接修改共享变量

方式	实现方法	特点
全局变量	直接定义模块级变量	- 零拷贝访问- 需要同步机制- 易出现竞态条件
线程安全容器	queue.Queue()	- 自带锁机制- 支持生产者消费者模式- 适用于异步任务调度
Lock/RLock	threading.Lock()	- 基础同步原语- 防止数据竞争- 可能引发死锁
Condition	threading.Condition()	- 高级通知机制- 可实现等待/通知模式

线程锁

前面讲了多线程的数据传递需要使用锁，接下来详细介绍一下线程锁的功能和用法。线程锁，也叫互斥锁，能够保证同一时间只有一个线程访问共享资源（如数据结构-共享列表），以防止出现多个线程同时访问导致数据紊乱的现象，解决多线程中的资源竞争问题。

具体来说：

1. 当一个线程想要访问或修改共享对象时，它必须先获得锁（通过调用`acquire()`方法）。

2. 如果锁已经被其他线程获得，那么当前线程将被阻塞，直到锁被释放（如何判断锁是否已被获取？）。

3. 一旦线程获得了锁，它就可以安全地操作共享对象，因为此时其他线程无法同时获得锁，因此不会同时操作该对象。

4. 操作完成后，线程必须释放锁（通过调用`release()`方法），以便其他线程可以获取锁并操作共享对象。

注意：锁本身并不改变共享对象的内容，它只是控制线程访问共享对象的顺序，确保操作的原子性。

线程锁代码实例

比如，一个计数器，多个线程同时增加它的值，不加锁的话结果可能会不正确。加了锁之后，结果就会正确。

```
import threading
# 共享资源
counter = 0
# 创建锁对象
lock = threading.Lock()
def increment_without_lock():
"""不加锁的计数器增加"""
    global counter#全局共享资源
    for _ in range(1000):
        temp = counter
        temp += 1
        counter = temp
def increment_with_lock():
"""加锁的计数器增加"""
    global counter#全局共享资源
    for _ in range(1000):
        #在对共享资源操作前获取锁，最后释放锁
        Lock.acquire():
        try:
            temp = counter
            temp += 1
            counter = temp
        finally:
            lock.release()
def run_demo(func):
"""测试函数"""
```

```
    global counter
    counter = 0 # 重置计数器
# 创建 5 个线程
    threads = [threading.Thread(target=func) for _ in range(5)]
    for t in threads:
    t.start()
    for t in threads:
        t.join()
        print(f"最终计数器值: {counter}")
print("不加锁情况：")
run_demo(increment_without_lock) # 输出通常小于 5000
print("\n 加锁情况：")
run_demo(increment_with_lock) # 正确输出 5000
```

（1）锁的本质：锁像是使用 acquire 方法创建一个密闭区（临界区），将对共享资源的操作（如上面的 temp+=1 三行代码）放到密闭区里，其它线程再想操作共享资源的时候因为已经上锁，进不到密闭区，只能等待，直到第一个线程释放锁（release 方法）之后，才能进行访问。保护的范围就是 acquire 和 release 之间的密闭区（或者 with 语句内）。

（2）内部原理：锁内部维护了一个状态（锁定/未锁定）和一个等待队列。- 当一个线程尝试获取一个已经锁定的锁时，该线程会被阻塞（进入等待状态），并被放入该锁的等待队列中。- 当持有锁的线程释放锁时，系统会从等待队列中唤醒一个线程（具体唤醒哪个取决于调度策略，通常是先进先出）。

（3）锁的作用（临界区）范围：锁只保护在同一个线程中，从获取锁之后到释放锁之前的代码块（临界区）。如果代码块内有 yield 将一个值返回给调用者（例如在生成器函数中），当前线程在 yield 处暂停，此时，锁仍然被当前线程持有（因为还没有执行到释放锁的代码）。其他线程如果尝试获取同一个锁，会被阻塞。 直到当生成器再次被唤醒（通过 next()或 send()）时，该线程会从 yield 之后继续执行，直到遇到下一个 yield 或函数结束，然后才会释放锁（如果使用了 with 语句，则会在离开 with 块时释放；或者手动释放）。

但是，被 yield 的对象本身的操作（即调用者在生成器外部对这个对象的操作）是否受锁保护呢？答案是否定的，因为： - 锁保护的是临界区代码（即生成器函数中锁之间的代码），并不保护被 yield 出去的对象在生成器外部的操作。- 调用者在获取到 yield 的对象后，可以在任何线程中操作这个对象，而该操作并不在锁的保护范围内。 因此，如果这个对象是共享的，并且需要线程安全地操作，那么调用者必须自己确保对其操作的线程安全性（例如，在操作该对象时获取相同的锁，或者使用其他同步机制）。

```
with obj.acquire() as cache_obj:
    # 在这个块内，对 cache_obj 的操作是受锁保护
    cache_obj.do_something()
# 这里，锁已经释放，对 cache_obj 的操作不再受锁保护
cache_obj.do_something_else() # 危险！如果多个线程同时操作，可能会发生竞态条件
```

（4）锁的获取与释放方式：

显式锁（threading.RLock）：手动获取与释放

```
# 手动获取锁
lock.acquire()          # 获取锁，如果已被占用则阻塞等待
try:
    # 临界区代码
    print("在锁保护中执行操作")
finally:
    lock.release()      # 手动释放锁
```

隐式锁（with acquire）：with 语句自动管理（推荐）

```
# 使用上下文管理器自动管理锁
with lock:              # 自动获取锁
    # 临界区代码
    print("在 with 块中自动管理锁")# 自动释放锁
```

（5）如何区分多个锁：一个代码文件中可以同时有多个锁，一个锁可以放到不同的共享资源前，会同时保护所有使用该锁的资源。代码中依据锁对象的身份（在内存中的地址）区分锁，而不是锁的名字或类型。但在代码中，可以给不同的锁

起不同的名字进行区分。锁还可以交替获取释放，获取 A，获取 B，释放 A，释放 B，但需要注意避免死锁。如下：

```
import threading
# 创建两个不同的锁对象
lock1 = threading.Lock()
lock2 = threading.Lock()
print(id(lock1)) # 输出锁 1 的内存地址（例如：140735812859664）
print(id(lock2)) # 输出锁 2 的内存地址（不同于锁 1）
def task1():
    with lock1:
    # 使用锁 1
    print("Task1 获得了 lock1")
def task2():
    with lock1:
    # 使用同一个锁 1（同一个对象）
    print("Task2 获得了 lock1")
def task3():
    with lock2:
    # 使用不同的锁 2
    print("Task3 获得了 lock2")
    # 创建线程
t1 = threading.Thread(target=task1)
t2 = threading.Thread(target=task2)
t3 = threading.Thread(target=task3)
t1.start()t2.start()t3.start()
```

- 线程 t1 和 t2 使用的是同一个锁对象`lock1`，因此它们之间会互斥（一个线程获得锁，另一个线程必须等待）。

- 线程 t3 使用的是`lock2`，与`lock1`无关，所以 t3 的执行不会受 t1 或 t2 的影响。

（5）注意事项：

锁的范围应尽量小，避免降低并发性能；

避免死锁（多个锁的获取顺序要一致）；

不是所有共享资源都需要锁（如只读访问）

锁的种类——原子锁和对象锁

根据细化功能的不同，Python 中的 threading 模块提供了两种锁：Lock（互斥锁）和 RLock（可重入锁）。

1. Lock（互斥锁）：

- 基本功能：确保每次只有一个线程可以持有锁。如果另一个线程试图获取一个已经被锁住的锁，它将会被阻塞，直到锁被释放。

- 缺点：同一个线程在已经持有锁的情况下再次尝试获取同一个锁，会导致死锁。

2. RLock（可重入锁）：

- 基本功能：允许同一个线程多次获取同一个锁。每次获取锁后，必须释放相同的次数才能真正释放锁。

- 优点：同一个线程可以多次获取锁而不会造成死锁，适用于递归函数或者多个方法需要同一个锁的情况。

RLock 内部维护计数器：

首次获取：计数器=1，锁定资源

再次获取：计数器+1（不会阻塞）

退出 with 块时计数器-1，归零时释放锁

应用实例：线程安全计数器

```
import threadingimport time

class ThreadSafeCounter:

    def __init__(self):

        self.value = 0   # 计数器值
```

```
        # 创建 RLock 原子锁（允许同一线程重入）
        self.atomic = threading.RLock()
    def increment(self):
        # 获取 RLock 锁（同一线程可多次调用）
        with self.atomic:  # 第一次获取锁
            temp = self.value  # 读取临时值
            time.sleep(0.001)  # 模拟处理延迟
            # 嵌套获取同一锁（RLock 允许）
            with self.atomic:  # 第二次获取锁（计数器+1）
                self.value = temp + 1  # 安全更新值
# 创建计数器实例
counter = ThreadSafeCounter()
# 定义线程任务函数
def worker(counter_obj: ThreadSafeCounter, n: int):
    for _ in range(n):
        counter_obj.increment()
# 创建两个线程
threads = [
    threading.Thread(target=worker, args=(counter, 100)),
        threading.Thread(target=worker, args=(counter, 100))  ]
# 启动线程
for t in threads:
    t.start()
# 等待线程结束
for t in threads:
    t.join()
# 输出结果
print(f"Final counter value: {counter.value}")
```

预期输出：200

1.即使存在 sleep(0.001)制造竞争条件，锁仍保证：线程 A 完成 **读→计算→写** 全过程后，线程 B 才能操作

2.若替换为普通 Lock 会怎样？

将初始化改为：self.atomic = threading.Lock() # 普通互斥锁

结果：程序死锁！
原因：同一个线程在持有锁时再次尝试获取（嵌套 with self.atomic），普通 Lock 会阻塞自身。

关键概念总结

特性	Lock（对象锁）	RLock（原子锁）
重入性	✗ 同一线程重复获取会死锁	✓ 允许同一线程多次获取
内部机制	二元状态（锁定/未锁定）	计数器 + 持有线程 ID
适用场景	简单互斥操作	嵌套/递归调用
性能	略高	略低（需维护计数器）

经验法则：需要锁嵌套时用 RLock，简单互斥用 Lock。

线程池

接下来分析 Python 中的多线程（threading`块）的各种映射（调度）方式。由于 Python 的全局解释器锁（GIL）的存在，多线程在 CPU 密集型任务中并不能利用多核优势，但在 I/O 密集型任务中非常有效。 在`threading`模块中，并没有像`multiprocessing.Pool`那样直接的映射方法，但可以使用 ThreadPoolExecutor（来自`concurrent.futures`模块）来实现类似的功能。而且它的接口与`multiprocessing.Pool`非常相似，便于比较。

map(func, *iterables, timeout=None, chunksize=1)

参数同样为待执行函数和可迭代对象，为阻塞式调用，按顺序返回结果，将函数应用到可迭代对象的每个元素上 适用于需要按输入顺序获取结果的简单任务 ```

from concurrent.futures import ThreadPoolExecutor

```
import time
def square(x):
    time.sleep(0.1)
# 模拟 I/O 操作
    return x * x
if __name__ == '__main__':
    with ThreadPoolExecutor(max_workers=4) as executor:
        # map 阻塞直到所有任务完成
        results = list(executor.map(square, range(10)))
        print("map results:", results)
```
``` **输出**: ``` map results: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] ```

submit(fn, *args, **kwargs)和 as_completed(fs, timeout=None)

非阻塞提交任务，然后按完成顺序获取结果，适合任务执行时间差异大，需要尽快获取已完成任务结果的场景

```
    def square(x):
    sleep_time = random.uniform(0.05, 0.2)
    time.sleep(sleep_time)
    return (x, x * x, sleep_time)
if __name__ == '__main__':
    with ThreadPoolExecutor(max_workers=4) as executor:
    # 提交任务
        futures = [executor.submit(square, i) for i in range(10)]
        print("按完成顺序获取结果:")
    #     as_completed 返回一个迭代器，在任务完成时产生 future 对象
        for future in as_completed(futures):
            x, result, sleep_time = future.result()
            print(f"任务 {x} = {result} (耗时 {sleep_time:.3f}s)")
```
输出:
按完成顺序获取结果:
任务 1: 1^2 = 1 (耗时 0.052s) 任务 0: 0^2 = 0 (耗时 0.058s) 任务 3: 3^2 = 9 (耗时 0.067s) 任务 2: 2^2 = 4 (耗时 0.119s) ...

wait(fs, timeout=None, return_when=ALL_COMPLETED)

等待一组 future 完成，可以设置等待条件（如全部完成、第一个完成等），适用于批量任务处理，需要控制等待条件

```
ffrom concurrent.futures import ThreadPoolExecutor, wait, FIRST_COMPLETED
import time
def square(x):
    time.sleep(0.1 * (10 - x))# 数字越大执行越快
    return x * x
if __name__ == '__main__':
```
```

```
    with ThreadPoolExecutor(max_workers=4) as executor: # 提交 10 个任务
futures = [executor.submit(square, i) for i in range(10)]
    # 等待至少一个任务完成
        done, not_done = wait(futures, return_when=FIRST_COMPLETED)
        print(f"\n 一个任务完成: 已完成 {len(done)} 个, 未完成
{len(not_done)} 个")
        for f in done:
            print(f"结果: {f.result()}")
            # 再等待全部完成
            done, not_done = wait(futures)
        print(f"\n 所有任务完成: 已完成 {len(done)} 个")
# 按输入顺序打印结果
        results = [f.result() for f in futures]
        print("所有结果:", results) ``` **输出**: ```
一个任务完成: 已完成 1 个, 未完成 9 个 结果: 81 所有任务完成: 已完成 10
个 所有结果: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] ```
```

方法对比总结

方法	特点	顺序	复杂度	适用场景
ThreadPoolExecutor.map	阻塞，顺序返回	输入顺序	低	简单任务，顺序结果
submit + as_completed	非阻塞，按完成顺序	完成顺序	中	尽快处理完成的任务
map + 多参数	处理多参数	输入顺序	中	多参数函数处理

注意 pool.map 只能映射 function 一个参数，所以需要先用 functools.partial 固定一个参数得到一个新的函数，再使用 map 去映射这个新的函数。所以 partial 是如何固定一个参数的，map 又是如何完成映射的。

协程&多进程&多线程

以下是协程与多进程/多线程的详细对比分析，包含代码示例和原理说明：
核心概念对比

特性	协程 (Coroutine)	多线程 (Threading)	多进程 (Multiprocessing)

并发层级	单线程内并发	单进程内并发	跨进程并发
调度方式	用户态主动切换	操作系统内核调度	操作系统内核调度
内存占用	极小（KB 级）	中等（MB 级）	大（独立内存空间，GB 级）
上下文切换成本	几乎为零	较高（需要内核介入）	最高（内存隔离）
数据共享	直接共享内存	需要线程锁机制	必须通过 IPC（管道/队列等）
适用场景	I/O 密集型任务	I/O 密集型任务（受 GIL 限制）	CPU 密集型任务
Python 实现模块	asyncio	threading	multiprocessing