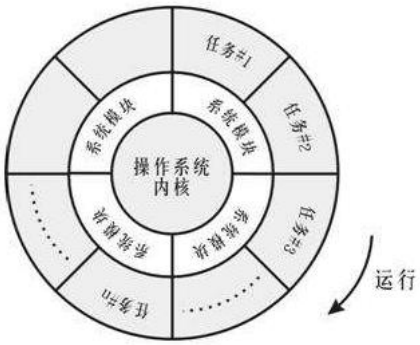
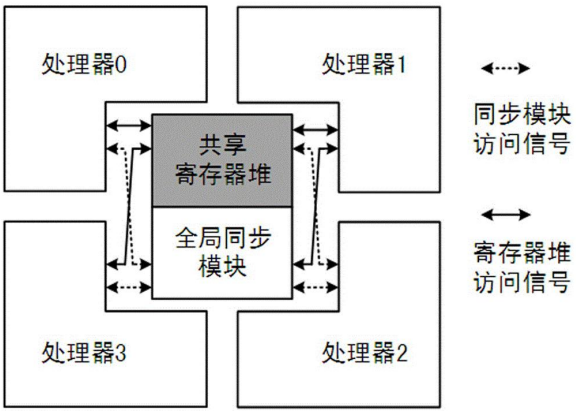
并发编程

1 为什么要有进程、线程

现代操作系统比如 Mac OS X，Linux，Windows 等，都是支持“多任务”的操作系统什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用逛淘宝，一边在听音乐，一边在用微信聊天，这就是多任务，至少同时有 3 个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。



现在，多核 CPU 已经非常普及了，但是，即使过去的单核 CPU，也可以执行多任务。由于 CPU 执行代码都是顺序执行的，那么，单核 CPU 是怎么执行多任务的呢？



答案就是操作系统轮流让各个任务交替执行，任务 1 执行 0.01 秒，切换到任务 2，任务 2 执行 0.01 秒，再切换到任务 3，执行 0.01 秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于 CPU 的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核 CPU 上实现，但是，由于任务数量远远多于 CPU 的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

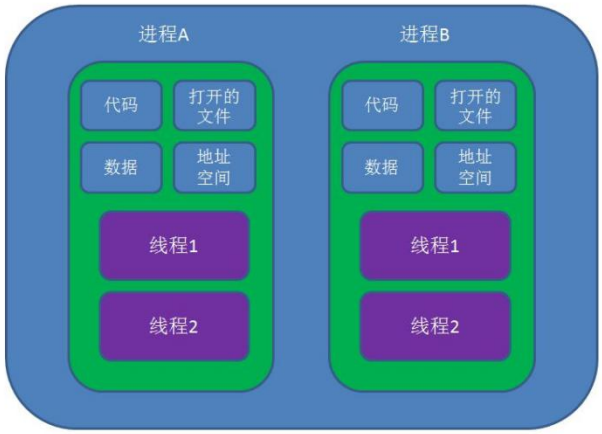
对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个 Word 就启动了一个 Word 进程。

有些进程还不止同时干一件事，比如微信，它可以同时进行打字聊天，视频聊天，朋友圈等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像微信这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核 CPU 才可能实现。

进程（Process）是一个具有一定独立功能的程序关于某个数据集合的一次运行活动

线程（Thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。



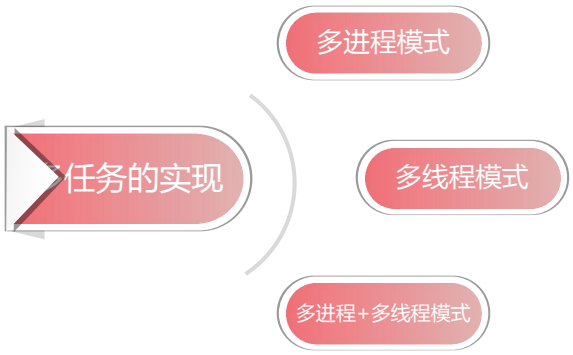
我们前面编写的所有的 Python 程序，都是执行单任务的进程，也就是只有一个线程。那说到这有的同学可能想，我可不可以让 Python 同时做多个任务，这样我就可以提高程序有效率了，如果能又如何开启呢？

并发编程解决方案：

1. 启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务
2. 启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务
3. 启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有 3 种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。



2 线程创建与使用

Python 的标准库提供了两个模块：_thread 和 threading，_thread 是低级模块，threading 是高级模块，对_thread 进行了封装。绝大多数情况下，我们只需要使用 threading 这个高级模块。

线程的创建可以通过分为两种方式：

1. 方法包装
2. 类包装

线程的执行统一通过 start()方法

注意：

- 主线程不会等待子线程运行结束，如果需要等待可使用 join()方法
- 不要启动线程后立即 join()，很容易造成串行运行，导致并发失效

```
    # 方法包装-启动多线程
from threading import Thread
from time import sleep, time

def func1(name):
    print("Threading:{} start".format(name))
    sleep(3)
    print("Threading:{} end".format(name))

if __name__ == '__main__':
    # 开始时间
    start = time()
    # 创建线程列表
    t_list = []
    # 循环创建线程
    for i in range(10):
        t = Thread(target=func1, args=('t{}'.format(i),))
        t.start()
        t_list.append(t)
    # 等待线程结束
    for t in t_list:
        t.join()
    # 计算使用时间
    end = time() - start
    print(end)
```

```
    # 类包装-启动多线程
from threading import Thread
from time import sleep, time


class MyThread(Thread):
    def __init__(self,name):
        Thread.__init__(self)
        self.name =name
    def run(self):
        print("Threading:{} start".format(self.name))
        sleep(3)
        print("Threading:{} end".format(self.name))


if __name__ == '__main__':
    # 开始时间
    start = time()
    # 创建线程列表
    t_list = []
    # 循环创建线程
    for i in range(10):
        t = MyThread('t{}'.format(i))
        t.start()
        t_list.append(t)
    # 等待线程结束
    for t in t_list:
        t.join()
    # 计算使用时间
    end = time() - start
    print(end)
```

3 守护线程与子线程

线程在分法有：

- 主线程：程序的本身
- 子线程：在程序另开起的线程

注意：

在行为上还有一种叫守护线程，主要的特征是它的生命周期。主线程死亡，它也就随之死亡

```
    # 类包装-启动多线程
from threading import Thread,current_thread
from time import sleep, time
class MyThread(Thread):
    def __init__(self,name):
        Thread.__init__(self)
        self.name =name
    def run(self):
        print("Threading:{} start".format(self.name),current_thread())
        sleep(3)
        print("Threading:{} end".format(self.name))

if __name__ == '__main__':
    # 开始时间
    start = time()
    # 创建线程列表
    t_list = []
    # 循环创建线程
    for i in range(10):
        t = MyThread('t{}'.format(i))
        t.setDaemon(True)
        t.start()
        t_list.append(t)
    # 等待线程结束
    # for t in t_list:xx
    #     t.join()
    # 计算使用时间
    end = time() - start
    print(end,current_thread())
```

4 线程锁（互斥锁）

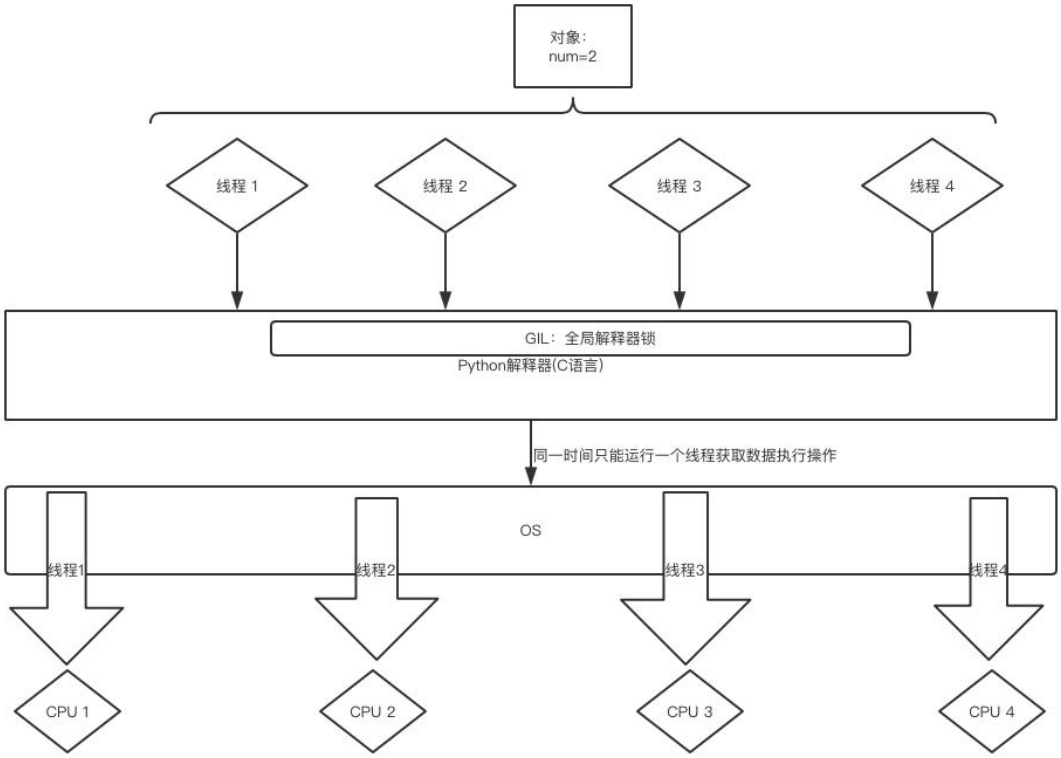
在 python 中，无论你有多少核，（在 Cpython 中）永远都是假象。无论你是 4 核，8 核，还是 16 核.....不好意思，同一时间执行的线程只有一个线程，它就是这个样子的。这个是 python 的一个开发时候，设计的一个缺陷，所以说 python 中的线程是假线程。

Python GIL(Global Interpreter Lock)

Python 代码的执行由 Python 虚拟机(也叫解释器主循环，CPython 版本)来控制，Python 在设计之初就考虑到要在解释器的主循环中，同时只有一个线程在执行，即在任意时刻，只有一个线程在解释器中运行。对 Python 虚拟机的访问由全局解释器锁（GIL）来控制，正是这个锁能保证同一时刻只有一个线程在运行

为什么要用这个 GIL 锁呢？

因为 Python 的线程是调用 C 语言的原生线程。因为 Python 是用 C 写的，启动的时候就是调用的 C 语言的接口。因为启动的 C 语言的远程线程，那它要调这个线程去执行任务就必须知道上下文，所以 Python 要去调 C 语言的接口的线程，必须要把这个上下文关系传给 Python，那就变成了一个我在加减的时候要让程序串行才能一次计算。就是先让线程 1，再让线程 2.....



注意：

GIL 并不是 Python 的特性，它是在实现 Python 解析器(CPython)时所引入的一个概念,同样一段代码可以通过 CPython，PyPy，Psyco 等不同的 Python 执行环境来执行,就没有 GIL 的问题。然而因为 CPython 是大部分环境下默认的 Python 执行环境。所以在很多人的概念里 CPython 就是 Python，也就想当然的把 GIL 归结为 Python 语言的缺陷

举例：

```
    from threading import Thread


def func1(name):
    print('Threading:{} start'.format(name))
    global num
    for i in range(50000000): # 有问题
    #for i in range(5000): # 无问题
        num += 1
    print('Threading:{} end num={}'.format(name, num))
```

```
if __name__ == '__main__':
    num =0
    # 创建线程列表
    t_list = []
    # 循环创建线程
    for i in range(5):
        t = Thread(target=func1, args=('t{}'.format(i),))
        t.start()
        t_list.append(t)
    # 等待线程结束
    for t in t_list:
        t.join()
```

Python 使用线程的时候，会定时释放 GIL 锁，这时会 sleep,所以才会出现上面的问题。面对这个问题，如果要解决此问题，我们可以使用 Lock 锁解决此问题

```
    from threading import Thread,Lock

def func1(name):
    print('Threading:{} start'.format(name))
    global num

    lock.acquire()
    for i in range(1000000): # 有问题
    #for i in range(5000): # 无问题
        num += 1
    lock.release()
    print('Threading:{} end num={}'.format(name, num))

if __name__ == '__main__':
    #创建锁
    lock = Lock()
    num =0
    # 创建线程列表
```

```
t_list = []
# 循环创建线程
for i in range(5):
    t = Thread(target=func1, args=('t{}'.format(i),))
    t.start()
    t_list.append(t)
# 等待线程结束
# for t in t_list:
#     t.join()
```

注意：

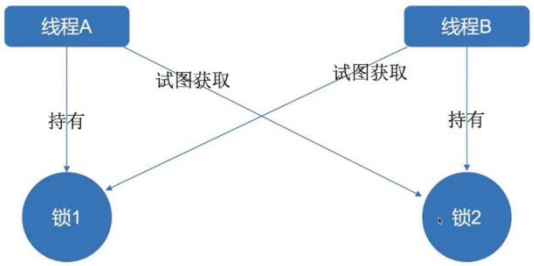
1. 加锁还可以使用 with 效果一样
2. 必须使用同一把锁
3. 如果使用锁，程序会变成串行，因此应该是在适当的地再加锁

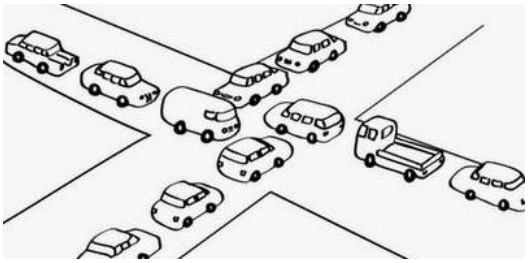
线程调度本质上是不确定的，因此，在多线程程序中错误地使用锁机制可能会导致随机数据损坏或者其他的异常行为，我们称之为竞争条件。为了避免竞争条件，最好只在临界区（对临界资源进行操作的那部分代码）使用锁

5 死锁

在多线程程序中，死锁问题很大一部分是由于线程同时获取多个锁造成的。举例：

1. 有两个人分别做“西兰花”和“红烧肉”，每个人都需要“锅”和“菜刀”才能炒菜。





```
    from threading import Thread, Lock
from time import sleep


def fun1():
    lock1.acquire()
    print('fun1 拿到菜刀')
    sleep(2)
    lock2.acquire()
    print('fun1 拿到锅')

    lock2.release()
    print('fun1 释放锅')
    lock1.release()
    print('fun1 释放菜刀')


def fun2():
    lock2.acquire()
    print('fun2 拿到锅')
    lock1.acquire()
    print('fun2 拿到菜刀')
    lock1.release()
    print('fun2 释放菜刀')
    lock2.release()
    print('fun2 释放锅')


if __name__ == '__main__':
```

```
lock1 = Lock()
lock2 = Lock()

t1 = Thread(target=fun1)
t2 = Thread(target=fun2)
t1.start()
t2.start()
```

2. 还一种就是使用锁的使用，多重使用了，如下：

```
        def fun01():
    lock3.acquire()
    print('关闭 1 大门')
    fun02()
    lock3.release()
    print('开启 1 大门')
def fun02():
    lock3.acquire()
    print('关闭 2 大门')
    lock3.release()
    print('开启 2 大门')
def fun03():
    fun01()
    fun02()
if __name__ == '__main__':
    lock3 = RLock()
```

6 信号量（Semaphore）

我们都知道在加锁的情况下，程序就变成了串行，也就是单线程，而有时，我们在不用考虑数据安全时，为了避免业务开启过多的线程时。我们就可以通过信号量（Semaphore）来设置指定个数的线程。举个简单例子：车站有 3 个安检口，那么同时只能有 3 个人安检，别人来了，只能等着别人安检完才可以过。

```
    from threading import Thread,BoundedSemaphore
from time import sleep
```

```
def an_jian(num):
    semapshore.acquire()
    print('第{}个人安检完成！'.format(num))
    sleep(2)
    semapshore.release()

if __name__ == '__main__':
    semapshore = BoundedSemaphore(3)
    for i in range(20):
        t = Thread(target=an_jian,args=(i,))
        t.start()
```

7 事件（Event）

Event()可以创建一个事件管理标志，该标志（event）默认为 False，event 对象主要有四种方法可以调用：

- event.wait(timeout=None)：调用该方法的线程会被阻塞，如果设置了 timeout 参数，超时后，线程会停止阻塞继续执行；
- event.set()：将 event 的标志设置为 True，调用 wait 方法的所有线程将被唤醒；
- event.clear()：将 event 的标志设置为 False，调用 wait 方法的所有线程将被阻塞；
- event.is_set()：判断 event 的标志是否为 True。

```
    from threading import Thread, Event
from time import sleep
from random import randint

state = 0
def door():
    global state
    while True:
        if even.is_set():
            print('门开着，可以通行~')
```

```
            sleep(1)
        else:
            print('门关了~请刷卡！')
            state = 0
            even.wait()
        if state > 3:
            print('超过 3 秒，门自动关门')
            even.clear()
        state +=1
        sleep(1)
def person():
    global state
    n = 0
    while True:
        n += 1
        if even.is_set():
            print('门开着：{}号进入'.format(n))
        else:
            even.set()
            state = 0
            print('门关着，{}号人刷卡进门'.format(n))
        sleep(randint(1, 10))

if __name__ == '__main__':
    even = Event()
    even.set()
    d = Thread(target=door)
    d.start()

    p = Thread(target=person)
    p.start()
```

8 线程通信-队列

之前的我们使用多个线程操作同一个变量时，我们都可以操作，但是我们在操作时，应该注意一个问题，那就是线程安全。那什么是线程安全呢？

线程安全是多线程编程时的计算机程序代码中的一个概念。在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况

8.1 使用的队列的好处

1. 安全
2. 解耦
3. 提高效率

```
    from threading import Thread, Lock

def change_num():
    for i in range(1000000):
        a[0]+=1
if __name__ == '__main__':
    a = [0]
    thread_list = []
    for i in range(10):
        t = Thread(target=change_num)
        t.start()
        thread_list.append(t)

    for t in thread_list:
        t.join()
    print(a[0])
```

从一个线程向另一个线程发送数据最安全的方式可能就是使用 queue 库中的队列了。创建一个被多个线程共享的 Queue 对象，这些线程通过使用 put() 和 get() 操作来向队列中添

加或者删除元素。Queue 对象已经包含了必要的锁，所以你可以通过它在多个线程间多安全地共享数据。

Python 包括 FIFO（先入先出)队列 Queue，LIFO（后入先出）队列 LifoQueue，和优先级队列 PriorityQueue。Queue 中包含以下方法：

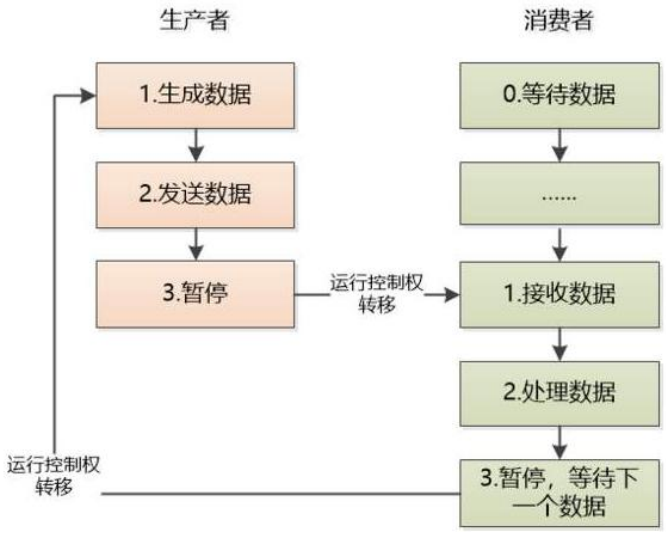
- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回 True,反之 False
- Queue.full() 如果队列满了，返回 True,反之 False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout 等待时间
- Queue.get_nowait() 相当 Queue.get(False)
- Queue.put(item) 写入队列，timeout 等待时间
- Queue.put_nowait(item) 相当 Queue.put(item, False)
- Queue.task*done() 在完成一项工作之后，Queue.task*done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

```
    from threading import Thread
from queue import Queue

def change_num():
    for i in range(1000000):
        n = q.get()
        print(n)
        q.put(n + 1)
if __name__ == '__main__':
    q = Queue()
    q.put(1)
    thread_list = []
    for i in range(10):
        t = Thread(target=change_num)
        t.start()
        thread_list.append(t)
    for t in thread_list:
        t.join()
```

```
print(q.get())
```

8.2 生产者与消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。



生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

```
    from queue import Queue
from threading import Thread
from time import sleep
from random import randint

def producer():
    num = 1
    while True:
        if queue.qsize() < 5:
            print(f'生产:{num}号加菲猫')
            queue.put(f'加菲猫{num}号')
            num += 1
```

```
        else:
            print('加菲猫满仓了,等待来人胜任领！')
        sleep(1)

def consumer():
    while True:
        print('获取：{}'.format(queue.get()))
        sleep(randint(1, 3))

if __name__ == '__main__':
    queue = Queue()
    t = Thread(target=producer)
    t.start()
    c = Thread(target=consumer)
    c.start()
    c2 = Thread(target=consumer)
    c2.start()
```

9 进程的创建与使用

Python 的标准库提供了个模块：multiprocessing

线程的创建可以通过分为两种方式

1. 方法包装

2. 类包装

使用 start()启动进程

```
# 方法包装-多进程实现
from multiprocessing import Process
improt os
from threading import Thread
from time import sleep, time
def func1(name):
print("当前进程 ID：",os.getpid())
```

```
print("父进程 ID：",os.getppid())
print("Process:{} start".format(name))
sleep(3)
print("Process:{} end".format(name))
if name == 'main':
# 开始时间
start = time()

# 创建进程列表
t_list = []
print("当前进程 ID：",os.getpid())
# 循环创建进程
for i in range(10):
t = Process(target=func1, args=('p{}'.format(i),))
t.start()
t_list.append(t)
# 等待进程结束
for t in t_list:
t.join()
  计算使用时间
end = time() - start
print(end)
```

```
# 类包装-进程实现
from multiprocessing import Process
from time import sleep, time

class MyProcess(Process):
```

```
    def __init__(self,name):
        Process.__init__(self)
        self.name =name
    def run(self):
        print("Process:{} start".format(self.name))
        sleep(3)
        print("Process:{} end".format(self.name))


if __name__ == '__main__':
    # 开始时间
    start = time()
    # 创建进程列表


t_list = []
    # 循环创建进程
    for i in range(10):
        t = MyProcess('t{}'.format(i))
        t.start()
        t_list.append(t)
    # 等待进程结束
    for t in t_list:
        t.join()
    # 计算使用时间
    end = time() - start
    print(end)
```

使用进程优点：

可以使用计算机多核，进行任务的并发执行，提高执行效率

运行不受其他进程影响，创建方便

空间独立，数据安全

使用进程缺点：

进程的创建和删除消耗的系统资源较多

10 进程的通信

Python 提供了多种实现进程间通信的机制，主要有以下 2 种：

1. Python multiprocessing 模块下的 Queue 类，提供了多个进程之间实现通信的诸多方法

2. Pipe，又被称为“管道”，常用于实现 2 个进程之间的通信，这 2 个进程分别位于管道的两端

10.1 Queue 实现进程间通信

前面讲解了使用 Queue 模块中的 Queue 类实现线程间通信，但要实现进程间通信，需要使用 multiprocessing 模块中的 Queue 类。

简单的理解 Queue 实现进程间通信的方式，就是使用了操作系统给开辟的一个队列空间，各个进程可以把数据放到该队列中，当然也可以从队列中把自己需要的信息取走。

```
from multiprocessing import Process,Queue
class MyProcess(Process):
    def __init__(self,name,mq):
        Process.__init__(self)
        self.name = name
        self.mq = mq
    def run(self):
        print("Process:{} start".format(self.name))
        print('--------------',self.mq.get(),'-------------')
        self.mq.put(self.name)
        print("Process:{} end".format(self.name))
if __name__ == '__main__':
    # 创建进程列表
```

```
    t_list = []
    mq = Queue()
    mq.put('1')
    mq.put('2')
    mq.put('3')
    # 循环创建进程
    for i in range(3):
        t = MyProcess('p{}'.format(i),mq)
        t.start()
        t_list.append(t)
    # 等待进程结束
    for t in t_list:
        t.join()
    print(mq.get())
    print(mq.get())
    print(mq.get())
```

10.2 Pipe 实现进程间通信

Pipe 直译过来的意思是“管”或“管道”，该种实现多进程编程的方式，和实际生活中的管（管道）是非常类似的。通常情况下，管道有 2 个口，而 Pipe 也常用来实现 2 个进程之间的通信，这 2 个进程分别位于管道的两端，一端用来发送数据，另一端用来接收数据

- send(obj)

发送一个 obj 给管道的另一端，另一端使用 recv() 方法接收。需要说明的是，该 obj 必须是可序列化的，如果该对象序列化之后超过 32MB，则很可能会引发 ValueError 异常

- recv()

接收另一端通过 send() 方法发送过来的数据

- close()

关闭连接

- poll([timeout])

返回连接中是否还有数据可以读取

- end_bytes(buffer[, offset[, size]])

发送字节数据。如果没有指定 offset、size 参数，则默认发送 buffer 字节串的全部数据；如果指定了 offset 和 size 参数，则只发送 buffer 字节串中从 offset 开始、长度为 size 的字节数据。通过该方法发送的数据，应该使用 recv_bytes() 或 recv_bytes_into 方法接收

- recv_bytes([maxlength])

接收通过 send_bytes() 方法发送的数据，maxlength 指定最多接收的字节数。该方法返回接收到的字节数据

- recv_bytes_into(buffer[, offset])

功能与 recv_bytes() 方法类似，只是该方法将接收到的数据放在 buffer 中

```
import multiprocessing


def func1():
    print(multiprocessing.current_process().pid,"进程发送数据：","Hello!")
    conn1.send("Hello!")
    print("from parent",conn1.recv())


if __name__ == '__main__':
    #创建管道
    conn1,conn2 = multiprocessing.Pipe()
    # 创建子进程
    process = multiprocessing.Process(target=func1)
    # 启动子进程
    process.start()
    conn2.send('你好！！')
    process.join()
    print(multiprocessing.current_process().pid,"接收数据：")
    print(conn2.recv())
```

10.3 Manager 管理器

管理器提供了一种创建共享数据的方法，从而可以在不同进程中共享

```
import multiprocessing
from multiprocessing import Manager
def func1():
    m_dict['sxt'] = 'true'
    m_list.append(123)
if __name__ == '__main__':
    with Manager() as mgr:
        m_list = mgr.list()
        m_dict = mgr.dict()
        # 创建子进程
        process = multiprocessing.Process(target=func1)
        # 启动子进程
        process.start()
        print(multiprocessing.cpu_count())
        process.join()
        print(m_dict)
        print(m_list)
        process.join()
```

11 进程池（Pool）

Python 提供了更好的管理多个进程的方式，就是使用进程池。

进程池可以提供指定数量的进程给用户使用，即当有新的请求提交到进程池中时，如果池未满，则会创建一个新的进程用来执行该请求；反之，如果池中的进程数已经达到规定最大值，那么该请求就会等待，只要池中有进程空闲下来，该请求就能得到执行。

使用进程池的优点

1. 提高效率，节省开辟进程和开辟内存空间的时间及销毁进程的时间
2. 节省内存空间

类/方法	功能	参数
Pool(processes)	创建进程池对象	processes 表示进程池中有多少进程
pool.apply_async(func,args,kwds)	异步执行 ；将事件放入到进程池队列	func 事件函数 args 以元组形式给 func 传参 kwds 以字典形式给 func 传参 返回值：返回一个代表进程池事件的对象，通过返回值的 get 方法可以得到事件函数的返回值
pool.apply(func,args,kwds)	同步执行；将事件放入到进程池队列	func 事件函数 args 以元组形式给 func 传参 kwds 以字典形式给 func 传参
pool.close()	关闭进程池	
pool.join()	回收进程池	
pool.map(func,iter)	类似于 python 的 map 函数，将要做的事件放入进程池	func 要执行的函数 iter 迭代对象

```
from multiprocessing import Pool
import os
from time import sleep
def func1(name):
    print(f"当前进程的 ID:{os.getpid()},{name}")
    sleep(2)
    return name

def func2(args):
```

```
    print(args)

if __name__ == "__main__":
    pool = Pool(5)

    pool.apply_async(func = func1,args=('sxt1',),callback=func2)
    pool.apply_async(func = func1,args=('sxt2',),callback=func2)
    pool.apply_async(func = func1,args=('sxt3',),callback=func2)
    pool.apply_async(func = func1,args=('sxt4',))
    pool.apply_async(func = func1,args=('sxt5',))
    pool.apply_async(func = func1,args=('sxt6',))
    pool.apply_async(func = func1,args=('sxt7',))
    pool.apply_async(func = func1,args=('sxt8',))

    pool.close()
    pool.join()
```

除此之外，我们可以使用 with 语句来管理进程池，这意味着我们无需手动调用 close() 方法关闭进程池。

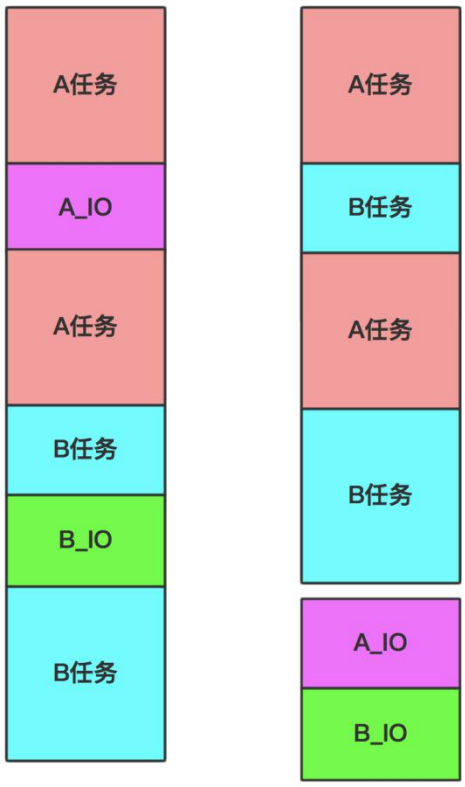
```
from multiprocessing import Pool
import os
from time import sleep
def func1(name):
    print(f"当前进程的 ID:{os.getpid()},{name}")
    sleep(2)
    return name
if __name__ == "__main__":
  with Pool(5) as pool:
      args = pool.map(func1,('sxt1,','sxt2,','sxt3,','sxt4,','sxt5,','sxt6,','sxt7,','sxt8,'))
```

```
for a in args:
    print(a)
```

12 协程

协程（coroutine），又称为微线程，纤程。(协程是一种用户态的轻量级线程)

作用：在执行 A 函数的时候，可以随时中断，去执行 B 函数，然后中断继续执行 A 函数（可以自动切换），单着一过程并不是函数调用（没有调用语句），过程很像多线程，然而协程只有一个线程在执行



12.1 使用协程的目的

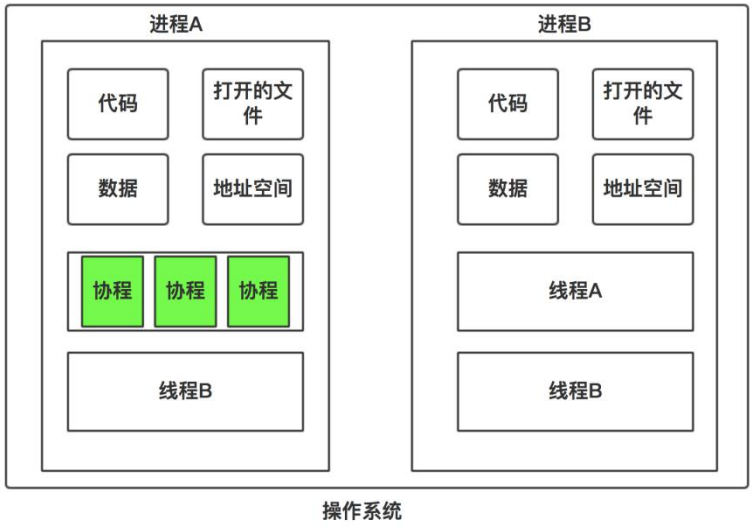
我们都知道 Cpython 解释器特别厉害，搞了个 GIL 锁吗！！！那好，我就自己搞成一个线程让你去执行，省去你切换线程的时间，我自己切换比你切换要快很多，避免了很多的开销。

对于单线程下，我们不可避免程序中出现 io 操作，但如果我们能在自己的程序中（即用户程序级别，而非操作系统级别）控制单线程下的多个任务能在一个任务遇到 io 阻塞时就将寄存器上下文和栈保存到其他地方，切换到另外一个任务去计算。在任务切回来的时候，恢复先前保存的寄存器上下文和栈，这样就保证了该线程能够最大限度地处于就绪态，即随时都可以被 cpu 执行的状态，相当于我们在用户程序级别将自己的 io 操作最大限度地隐藏起来，从而可以迷惑操作系统，让其看到：该线程好像是一直在计算，io 比较少，从而更多的将 cpu 的执行权限分配给我们的线程（注意：线程是 CPU 控制的，而协程是程序自身控制的）

12.2 协作的标准

必须在只有一个单线程里实现并发

- 修改共享数据不需加锁
- 用户程序里自己保存多个控制流的上下文栈
- 一个协程遇到 IO 操作自动切换到其它协程



12.3 使用协程的优点

由于自身带有上下文和栈，无需线程上下文切换的开销,属于程序级别的切换，操作系统完全感知不到，因而更加轻量级；

无需原子操作的锁定及同步的开销；

方便切换控制流，简化编程模型

单线程内就可以实现并发的效果，最大限度地利用 cpu，且可扩展性高，成本低（注：一个 CPU 支持上万的协程都不是问题。所以很适合用于高并发处理）

12.4 使用协程的缺点

无法利用多核资源：协程的本质是个单线程,它不能同时将 单个 CPU 的多个核用上,协程需要和进程配合才能运行在多 CPU 上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

进行阻塞（Blocking）操作（如 IO 时）会阻塞掉整个程序

12.5 旧知识实现任务切换

通过以前的学习的知识实现任务切换（yield 本身就是一种在单线程下可以保存任务运行状态的方法）

yiled 可以保存状态，yield 的状态保存与操作系统的保存线程状态很像，但是 yield 是代码级别控制的，更轻量级

send 可以把一个函数的结果传给另外一个函数，以此实现单线程内程序之间的切换

```
import time
def func1():
    for i in range(11):
        yield
        print(f'北京：第{i}次打印啦')
        time.sleep(1)
def func2():
    g = func1()
    next(g)
    for k in range(10):
        print(f'上海：第{k}次打印了' )
        time.sleep(1)
        next(g)
#不写 yield，下面两个任务是执行完 func1 里面所有的程序才会执行 func2 里面的程序，有了 yield，我们实现了两个任务的切换+保存状态
# func1()
func2()
```

12.6 单纯的切换不能提高运行效率

基于 yield 并发执行，多任务之间来回切换，这就是个简单的协程的体现，但是他能够节省 I/O 时间吗？不能

```
import time
def consumer(): # 并没参数：x。串有参数：x
    '''任务 1:接收数据,处理数据'''
    while True: # 并
        x=yield # 并
        time.sleep(1) #发现什么？只是进行了切换，但是并没有节省 I/O 时间
        print('处理了数据：',x)
def producer():
    '''任务 2:生产数据'''
    g=consumer() # 并
    next(g)  #并:找到了 consumer 函数的 yield 位置
    for i in range(3):
        g.send(i)  #并:给 yield 传值，然后再循环给下一个 yield 传值，并且多了切换的程序，比直接串行执行还多了一些步骤，导致执行效率反而更低了。
        print('发送了数据：',i)
        # consumer(i) # 串
if __name__ == "__main__":
    start=time.time()
    #基于 yield 保存状态,实现两个任务直接来回切换,即并发的效果
    #PS:如果每个任务中都加上打印,那么明显地看到两个任务的打印是你一次我一次,即并发执行的.
    producer() #我在当前线程中只执行了这个函数，但是通过这个函数里面的 send 切换了另外一个任务
    stop=time.time()
    print(stop-start)
```

12.7 Greenlet 的使用

如果我们在单个线程内有 20 个任务，要想实现在多个任务之间切换，使用 yield 生成器的方式过于麻烦（需要先得到初始化一次的生成器，然后再调用 send。。。非常麻烦），而使用 greenlet 模块可以非常简单地实现这 20 个任务直接的切换。

```
# 真正的协程模块就是使用 greenlet 完成的切换
from greenlet import greenlet


def attack(name):
    print(f'{name} ：我要买包！') # 2
    g2.switch('周瑜') # 3
    print(f'{name} ：我要去学编程！') # 6
    g2.switch() # 7


def player(name):
    print(f'{name} ：买买买！！') # 4
    g1.switch() # 5
    print(f'{name} ：一定去尚学堂学！！！！') # 8


g1 = greenlet(attack)
g2 = greenlet(player)


g1.switch('大乔') # 可以在第一次 switch 时传入参数，以后都不需要 #1
```

12.8 单纯的切换不能提高运行效率

无 IO 代码

```
#顺序执行
import time
def f1():
    res=1
    for i in range(10000000):
        res+=i
def f2():
    res=1
    for i in range(10000000):
        res+=i


start=time.time()
f1()
f2()
stop=time.time()
print('run time is %s' %(stop-start)) #1.1117279529571533
```

无 IO，有任务切换

```
#切换
from greenlet import greenlet
import time
def f1():
```

```
    res=1
    for i in range(10000000):
        res+=i
        g2.switch()

def f2():
    res=1
    for i in range(10000000):
        res+=i
        g1.switch()

start=time.time()
g1=greenlet(f1)
g2=greenlet(f2)
g1.switch()
stop=time.time()
print('run time is %s' %(stop-start)) # 4.465435981750488
```

12.9 Gevent 模块

Gevent 是一个第三方库，可以轻松通过 gevent 实现并发同步或异步编程，在 gevent 中用到的主要模式是 Greenlet，它是以 C 扩展模块形式接入 Python 的轻量级协程。Greenlet 全部运行在主程序操作系统进程的内部，但他们被协作式地调度。

安装：

```
pip install gevent
```

代码：

```
import gevent

def gf(name):

    print(f'{name} ：我要买包！') # 2

    gevent.sleep(2) # 3

    print(f'{name} ：我要去学编程！' ) # 6

def bf(name):

    print(f'{name} ：买买买！！ ') # 4

    gevent.sleep(2)

    print(f'{name} ：一定去尚学堂学！！！！') # 8


g1 = gevent.spawn(gf, '小乔')

g2 = gevent.spawn(bf, name='周瑜')

gevent.joinall([g1,g2])
```

注意：上例 gevent.sleep(2)模拟的是 gevent 可以识别的 io 阻塞;

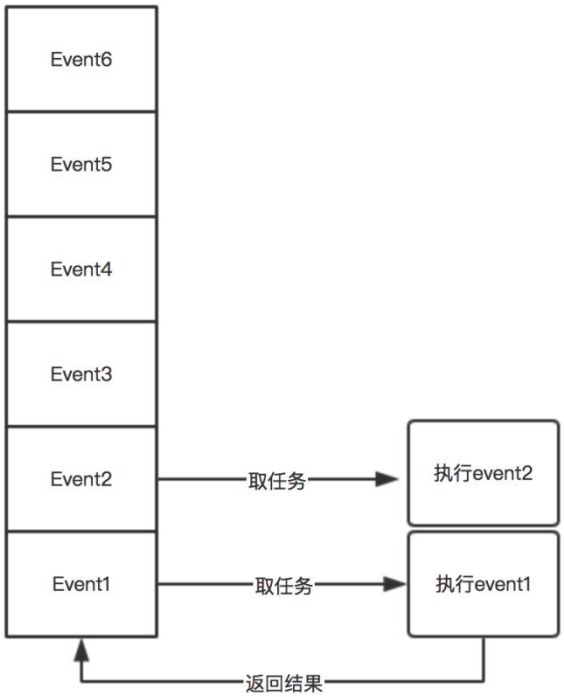
而 time.sleep(2)或其他的阻塞,gevent 是不能直接识别的需要用下面一行代码,打补丁,就可以识别了

```
from gevent import monkey;
monkey.patch_all() #必须放到被打补丁者的前面，如 time，socket 模块之前
```

12.10 async io 异步 IO

asyncio 是 python3.4 之后的协程模块，是 python 实现并发重要的包，这个包使用事件循环驱动实现并发。

事件循环是一种处理多并发量的有效方式，在维基百科中它被描述为「一种等待程序分配事件或消息的编程架构」，我们可以定义事件循环来简化使用轮询方法来监控事件，通俗的说法就是「当 A 发生时，执行 B」。事件循环利用 poller 对象，使得程序员不用控制任务的添加、删除和事件的控制。事件循环使用回调方法来知道事件的发生。



那么如何使用 asyncio 呢？我先下演示下代码：

```
import asyncio

@asyncio.coroutine # python3.5 之前

def func1():

    for i in range(5):

        print('Python 大法好！！')
```

```
        yield from asyncio.sleep(1)


async def func2(): # python3.5 之后
    for i in range(5):
        print('使用协程能提高薪资！！')
        await asyncio.sleep(2)


#创建协程对象
g1 = func1()
g2 = func2()
#获取事件循环
loop = asyncio.get_event_loop()
# 监听事件循环
loop.run_until_complete(asyncio.gather(g1,g2))
# 关闭事件
loop.close()
```

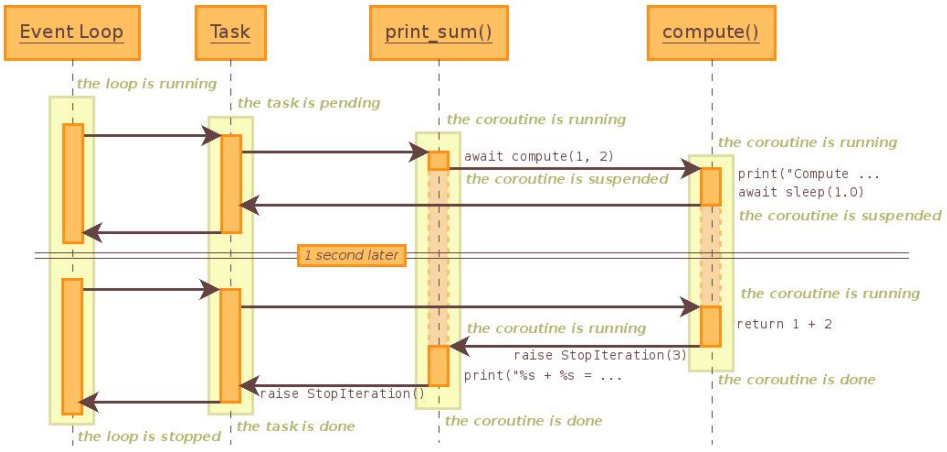
- @asyncio.coroutine 协程装饰器装饰
- asyncio.sleep() 可以避免事件循环阻塞
- get_event_loop() 获取事件循环
- Loop.run_until_complete() 监听事件循环
- gather() 封装任务
- await 等于 yield from 就是在等待 task 结果

```
import asyncio


async def compute(x, y):
    print(f"compute: {x}+{y} ...")
    await asyncio.sleep(1)
    return x + y


async def print_sum(x, y):
    result = await compute(x, y)
    print(f"{x}+{y}={result}")


loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

通过回调函数获取结果

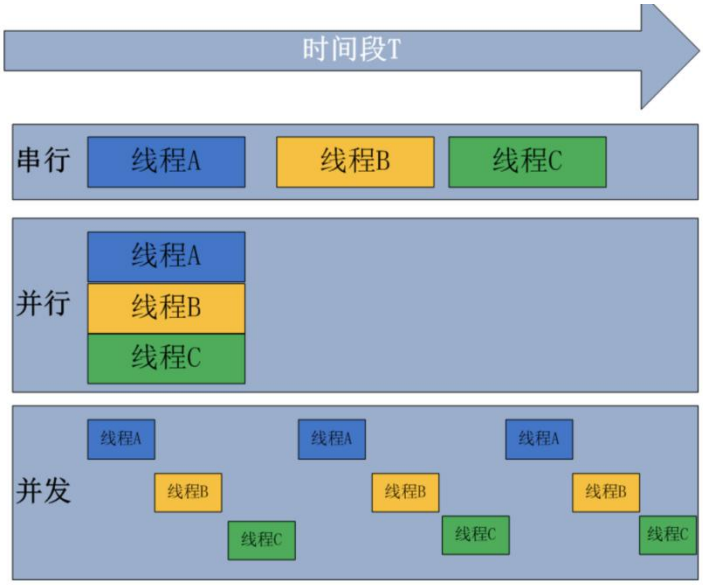
```
import asyncio
import functools
async def compute(x, y):
    print(f"compute: {x}+{y} ...")
    await asyncio.sleep(1)
    return x + y
async def print_sum(x, y):
```

```
    task = asyncio.create_task(compute(x,y))
    task.add_done_callback(functools.partial(end,x,y))
    await asyncio.sleep(0.1)
    for i in range(100000):
        if i% 500 ==0:
            print(i)
            await asyncio.sleep(0.1)
def end(x,y,t):
    print(f"{x}+{y}={t.result()}")


loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```



13 总结

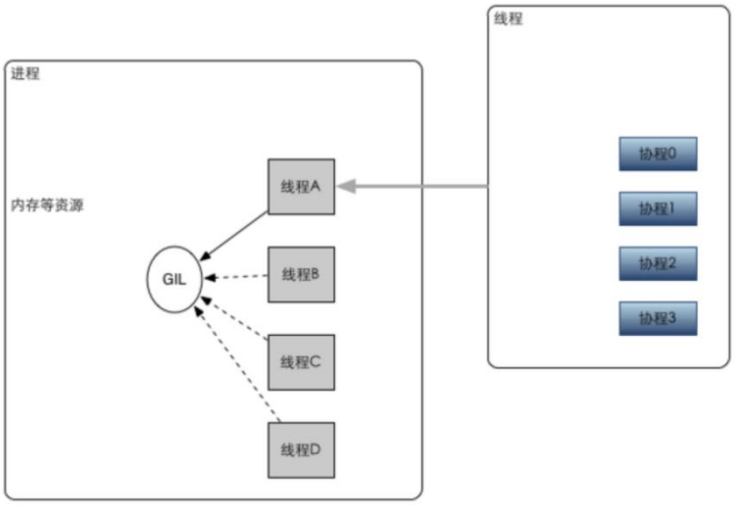
13.1 串行、并行与并发的区别



- 并行：指的是任务数小于等于 cpu 核数，即任务真的是一起执行的
- 并发：指的是任务数多余 cpu 核数，通过操作系统的各种任务调度算法，实现用多个任务“一起”执行（实际上总有一些任务不在执行，因为切换任务的速度相当快，看上去一起执行而已）

13.2 进程与线程的区别

前面讲了进程、线程与协程，但可能你还觉得迷糊，感觉他们很类似



1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；
4. 调度和切换：线程上下文切换比进程上下文切换要快得多。

有一个老板想要开个工厂进行生产某件商品（例如：手机）

他需要花一些财力物力制作一条生产线，这个生产线上有很多的器件以及材料这些所有的为了能够生产手机而准备的资源称之为：进程

只有生产线是不能够进行生产的，所以老板的找个工人来进行生产，这个工人能够利用这些材料最终一步步的将手机做出来，这个来做事情的工人称之为：线程

这个老板为了**提高生产率**，想到 3 种办法：

1. 在这条生产线上多招些工人，一起来做手机，这样效率是成倍增长，即单进程 多线程方式

2. 老板发现这条生产线上的工人不是越多越好，因为一条生产线的资源以及材料毕竟有限，所以老板又花了些财力物力购置了另外一条生产线，然后再招些工人这样效率又再一步提高了，即多进程 多线程方式

3. 老板发现，现在已经有了很多条生产线，并且每条生产线上已经有很多工人了（即程序是多进程的，每个进程中又有多个线程），为了再次提高效率，老板想了个损招，规定：如果某个员工在上班时临时没事或者再等待某些条件（比如等待另一个工人生产完谋道工序 之后他才能再次工作），那么这个员工就利用这个时间去做其它的事情，那么也就是说：如果一个线程等待某些条件，可以充分利用这个时间去做其它事情，其实这就是：协程方式

13.3 进程、线程和协程的特点

进程：拥有自己独立的堆和栈，既不共享堆，也不共享栈，进程由操作系统调度；进程切换需要的资源很最大，效率很低

线程：拥有自己独立的栈和共享的堆，共享堆，不共享栈，标准线程由操作系统调度；线程切换需要的资源一般，效率一般（当然了在不考虑 GIL 的情况下）

协程：拥有自己独立的栈和共享的堆，共享堆，不共享栈，协程由程序员在协程的代码里显示调度；协程切换任务资源很小，效率高

多进程、多线程根据 cpu 核数不一样可能是并行的，但是协程是在一个线程中 所以是并发

选择技术考虑的因素：切换的效率、数据共享的问题、 数据安全、是否需要并发