多进程编程

一、 multiprocessing 模块

1. `multiprocessing` 模块提供了多进程编程的能力
   - 它的API非常类似于 `threading` 模块，但是也提供了一些`threading` 模块不具有的能力
   - 相比于线程，它通过派生子进程来绕过`GIL` 的限制，从而可以充分利用多核处理器的能力
   - 它提供了进程间的同步

1.1 派生模式

1. `multiprocessing` 模块支持三种派生子进程的模式：
   - `spawn` 模式：父进程重新开启一个全新的 python 解释器进程，然后子进程的代码运行在这个新的解释器进程中
     - 子进程只继承了必要的资源来完成任务。父进程打开的文件描述符和一些句柄不会被子进程继承。
     - 该模式在三种模式中速度最慢
     - 可以用于 `Unix/Windows`，是`Windows` 上的默认方式
   - `fork` 模式：父进程使用`os.fork()` 来派生一个 python 解释器进程。子进程的代码运行在这个新的解释器进程中
     - 子进程几乎与父进程完全相同；子进程继承了父进程的所有资源
     - 注意：`fork` 一个多线程的进程是有问题的
     - 只可用于`Unix`，是`Unix` 上的默认方式
   - `forkserver` 模式：在这种模式下，父进程派生了一个 `server` 进程。当任何时候，父进程需要创建子进程时，由父进程通知 `server` 进程来派生一个子进程
     - `server` 进程是个单线程的进程，因此`server` 进程本身可以安全的使用 `os.fork()`
     - 不必要的资源不会被继承（如：父进程打开的文件描述符和一些句柄）
     - 只可用于部分`Unix`（支持在管道上传递文件描述符的 `Unix`）
2. 在`Unix` 上，使用 `spawn,forkserver` 模式派生子进程时，父进程会额外派生一个 `semaphore tracker` 进程。
   - 这个 `tracker` 进程用于跟踪程序中所有进程创建的信号量

     > 不属于本程序的进程，则`tracker` 无法管理

   - 理论上每个信号量都与某个进程关联。但是如果进程被信号意外终止，则它会导致该信号量失联，成为泄露的信号量。
     - 泄露的信号量是个问题。因为操作系统能够分配的信号量的数量时固定的。

       当泄露的信号量增多（这些信号量并不会操作系统被自动收回），则系统可分配的信号量的数量减少。

       当系统无法分配信号量时，则申请创建信号量会失败。此时只能重启操作系统来解决。
   - 当程序中有任何进程终止时，`tracker` 进程会查看该终止进程关联的信号量。若这些信号量并没有被解绑，则`tracker` 进程会解绑。
3. 设置派生模式：

- 在 `main` 模块的 `if __name__=='__main__'` 子句中，调用 `set_start_method()` 函数
  - `set_start_method()` 函数在程序中最多只能调用一次。这意味着所有的子程序都是同一种派生模式

```
if __name__ == '__main__':
    multiprocessing.set_start_method('spawn')
```

- 通过 `get_context('fork_style')` 获取一个 `fork_style` 模式的 `context` 对象，然后将该 `context` 当做 `multiprocessing` 模块使用
  - 该 `context` 对象具有 `multiprocessing` 模块相同的 `API`
  - 通过它创建子进程的模式就是 `fork_style` 模式
  - 通过在一个程序中调用多次 `get_context()` ，则可以使用多种派生模式来创建子进程
  - 注意：与某个模式 `context` 创建的对象（如 `Lock,Queue` ）与另一种模式 `context` 创建的进程是不兼容的

```
if __name__ == '__main__':
    ctx = multiprocessing.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,)) #spawn 模式
    ...
    ctx2 = multiprocessing.get_context('forkserver')
    q2 = ctx.Queue()
    p2 = ctx.Process(target=foo, args=(q,)) # forkserver 模式
```

4. `multiprocessing.get_all_start_methods()` ：获取可用的派生模式列表
   - 返回的列表的首个元素就是默认的派生模式
5. `multiprocessing.set_start_method(method)` ：设置派生模式
6. `multiprocessing.get_start_method(allow_none=False)` ：返回当前设置的派生模式名
   - 如果当前的派生模式未设定
     - 如果 `allow_none=False` ：则返回默认的派生模式名
     - 如果 `allow_none=True` ：则返回 `None`
   - 如果当前的派生模式已设定：则返回已设定的派生模式名
7. `multiprocessing.get_context(method=None)` ：获取一个 `context` 对象：
   - 该对象具有和 `multiprocessing` 模块相同的属性
   - 对于 `method` 参数：
     - 如果为 `None` ，则返回默认的 `context`
     - 如果为一个派生模式名，则返回指定类型的 `context`
     - 否则抛出 `ValueError` 异常

1.2 其它

1. `multiprocessing.active_children()` ：返回当前进程的存活的子进程列表。

- 调用此函数的副作用是：当前进程的所有已经结束的子进程会被 `join`

2. `multiprocessing.cpu_count()`：返回系统的CPU数量。
   - 该数量并不等于当前进程可以使用的CPU数量。可用的CPU数量为 `len(os.sched_getaffinity(0))`
3. `multiprocessing.current_process()`：返回当前进程对应的 `Process` 对象
4. `multiprocessing.freeze_support()`：对 `windows executable` 的 `frozen` 程序进行 `multiprocessing` 支持
   - 它必须在 `if __name__ == '__main__':` 这一行之后紧接着调用
5. `multiprocessing.set_executable(path)`：设置启动子进程的 Python 解释器的路径。
   - 默认情况下，使用 `sys.executable`
6. 父进程的全局变量在子进程中也可用。但是注意：
   - 在子进程的 `.start()` 方法调用时，同一个全局变量在父子进程中的值可能改变了。
7. 当使用 `spawn` 或 `forkserver` 模式时，需要使用 `if __name__=='__main__'` 来防止出现进程的循环派生：

```
from multiprocessing import Process
def foo():
    print('hello')
### 这里应该加上： if __name__=='__main__'
p = Process(target=foo)
p.start()#由于子进程会重新import 该模块，这会导致进程的循环派生
```

二、 Process

1. `Process` 类的 API非常类似于 `threading.Thread`
2. API：

```
class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={},
*, daemon=None)
```

   - `Process` 代表一个独立的进程
   - 它的参数应该使用以关键词参数调用。
   - 每个参数应该支持 `pickle`
3. 如果一个子类派生自 `Process`，则：
   - 子类必须在自己的初始化方法中首先调用父类的初始化方法
   - 子类可以重写 `.run()` 方法来实现自己的子进程处理逻辑
4. 参数：
   - `group` ： 为 `None`。它是为了保持与 `threading.Thread` 接口一致
   - `target`：可调用对象。它是子进程的执行逻辑。
     - 它会被 `.run()` 方法自动调用
     - 默认为 `None`，表示子进程什么都不做
   - `name`：为子进程的名字

- `args`：一个元组，为 `target` 提供位置参数
- `kwargs`：一个字典，为 `target` 提供关键字参数
- `daemon`：是一个关键字参数。
  - 如果为 `True`，表示子进程是个 `daemon` 进程
  - 如果为 `False`，表示子进程不是个 `daemon` 进程
  - 如果为 `None`，则表示子进程是否为 `daemon` 与父进程一致

2.1 方法

1. `.run()`： 该方法代表子进程的执行逻辑。
   - 该方法的默认行为是：调用初始化方法中的 `target`
   - 你可以在 `Process` 的子类中重写该方法
2. `.start()`：启动子进程。
   - 该方法最多被调用一次
   - 它会在新的进程空间中自动调用 `.run()` 方法
3. `.join([timeout])`：等待子进程结束。
   - 如果 `timeout=None`，则父进程会阻塞，直到子进程结束
   - 如果 `timeout` 为一个正数，则父进程阻塞的时间最多为 `timeout` 秒。
   - 如果想判断子进程是结束还是等待超时，则可以通过子进程的 `exitcode` 属性来判断
   - 一个进程可以被 `join` 多次
   - 一个进程不可以 `join` 它自己，这会引起死锁
   - 如果在进程 `start` 之间就 `join` 它，则抛出异常
4. `.is_alive()`：返回进程是否存活。
   - 进程在 `start` 之后，在结束之前，都是存活状态。
5. `.terminate()`：终止子进程。
   - 在 `UNIX` 上，这是通过发送 `SIGTERM` 信号来实现；在 `Windows` 上，这是通过调用 `TerminateProcess()` 来实现的
   - 该方法的调用导致 `exit` 处理程序和 `finally` 语句不被执行（此时可能会导致进程使用的管道、队列、信号量处于异常状态）
   - 该方法的调用会使得子进程的子孙进程成为孤儿进程

2.2 属性

1. `name`：进程的名字
   - 多个进程可能使用同一个名字
   - 最初的进程名字由初始化函数的 `name` 参数而来。如果未给出该参数，则由程序自动编号
2. `daemon`：进程的 `daemon` 标记
   - 如果要设置该属性，则必须在进程的 `.start()` 方法调用之前设置
   - 最初的进程名字由初始化函数的 `daemon` 参数而来。如果未给出该参数，则继承自父进程
   - 当一个进程退出时，它会试图终止它的所有 `daemonic` 子进程（这就是它与非 `daemonic` 子进程的区别）
     - 这些被终止的 `daemonic` 并不会经过 `join`
   - 一个 `daemonic` 进程禁止创建子进程（否则可能导致产生大量的孤儿进程）

- 它不同于`UNIX`中的守护进程。这里的`daemonic`进程事实上就是普通的进程

3. `pid`：子进程的ID。
   - 在进程被派生出来之前，该属性为`None`
4. `exitcode`：子进程的退出码
   - 如果子进程尚未退出，则它是`None`
   - 如果子进程被信号 N 终止，则退出码为 $-N$
5. `authkey`：进程的验证`key`（一个字符串）
   - 主进程的`authkey`是一个随机数，由`os.urandom()`给出
   - 子进程的`authkey`继承自父进程，但是也可以修改
6. `sentinel`：它是一个系统提供的数字句柄，作用是一个哨兵。当子进程结束时，它就可用
   - 通常它用于判断子进程是否结束。在这个意义上，推荐使用`.join()`方法
7. `.start()/.join()/.is_alive()/.terminate()/.exitcode`必须由父进程调用，一定不能由子进程调用。

2.3 异常

1. `multiprocessing.ProcessError`：所有`multiprocessing`异常的基类
2. `multiprocessing.BufferTooShort`：当调用`Connection.recv_byte_into()`时，如果缓冲区太小，则抛出该异常
   - 假设`e`为该类异常的一个实例，则`e.args[0]`将给出读到的内容
3. `multiprocessing.AuthenticationError`：当验证异常时，抛出该异常
4. `multiprocessing.TimeoutError`：当超时时，抛出该异常

三、进程间通信

1. `multiprocessing`支持两种类型的进程间通信：`Queue`，`Pipe`
   - `Pipe`只能让两个进程之间相互通信
   - `Queue`能够让多个进程之间相互通信

3.1 Pipe

1. API：

```
multiprocessing.Pipe([duplex])
```

它是一个函数，调用时返回一对`Connection`对象：`(conn1,conn2)`，代表一个管道的两个端口。

- 如果`duplex=True`（默认值），则管道是双向的。如果`duplex=False`，则管道是单向的：
  - `conn1`只能接收消息
  - `conn2`只能发送消息

2. 每个`Connection`对象都有一个`.send()`方法和一个`.recv()`方法
3. 如果同一时间有多个进程（或进程）同时读同一个管道的某个端口，或者同时写同一个管道的某个端口，则会导致管道的数据被破坏

3.1.1 Connection

1. `Connection` 对象可以发送、接收字符串、支持 `pickle` 的对象
    - 你可以将它理解为面向 `socket` 的消息传递机制
    - 通常使用 `Pipe()` 来创建
2. `API`：

```
class multiprocessing.Connection
```

3. 方法：
    - `.send(obj)`：发送一个对象。
        - `obj` 必须支持 `pickle`，且 `pickle` 之后的大小不能太大（通常不超过大约 32MB）。如果太大，则抛出 `ValueError` 异常
    - `.recv()`：接收一个对象。
        - 接收的对象由 `.send()` 发送过来的
        - 如果没有任何数据，则该方法会阻塞当前进程
        - 如果没有任何数据，且接收到发送端关闭的标记，则抛出 `EOFError`
    - `.close()`：关闭 `connection`
        - 当 `connection` 由垃圾收集机制收集时，该方法被自动调用
    - `.poll([timeout])`：返回 `connection` 是否有数据可读
        - 若未设定 `timeout`，则立即返回结果
        - 若设定了 `timeout` 为一个正数，则阻塞直到超时 `timeout` 秒。
        - 如果设定了 `timeout=None`，则阻塞并且永不超时。
    - `.send_bytes(buffer[,offset[,size]])`：发送字节序列作为一个消息
        - `buffer` 指定字节序列
        - `offset`：指定从 `buffer` 的多少偏移量开始读取
        - `size`：指定读取多少字节
        - 如果读取的数量太大，则抛出 `ValueError` 异常
    - `.recv_bytes([maxlength])`：接收字节序列为一个字符串
        - 如果没有任何数据，则该方法会阻塞当前进程
        - 如果没有任何数据，且接收到发送端关闭的标记，则抛出 `EOFError`
        - 如果接收的信息字节数超过了 `maxlength` ，则抛出 `OSError`，并且该 `connection` 再也不可读了
    - `.recv_bytes_into(buffer,[offset])`：接收字节序列到 `buffer` 中，返回读取到的消息字节数
        - 如果没有任何数据，则该方法会阻塞当前进程
        - 如果没有任何数据，且接收到发送端关闭的标记，则抛出 `EOFError`
        - `buffer` 必须是可写的 `bytes-like` 对象。
        - 若 `offset` 给定，则消息从 `buffer` 的指定偏移处开始写
            - `offset` 必须非负，而且小于 `buffer` 的长度
        - 如果 `buffer` 太短，则抛出 `BufferTooShort` 异常，同时 `e.args[0]` 中存储了完整的消息
4. 由于 `Connection.recv()` 会自动对它接收到的消息进行 `unpickle`，因此可能会存在安全隐患
    - 除非你相信消息的来源，否则不要轻易的使用 `.recv()` （可以考虑使用 `.recv_bytes()` ）
5. 如果一个进程在试图读/写一个管道的时候被意外杀死，则管道的数据就被破坏掉了。

- 此时的消息边界不再可靠。

3.2 Queue

1. `multiprocessing.Queue`：其用法类似于 `queue.Queue`
   - 它是线程安全的，也是进程安全的
2. `Queue,SimpleQueue,JoinableQueue` 是基于 `queue.Queue` 的一个先进先出的生产者-消费者模型
3. 当一个对象入队列时，首先将它 `pickle`，然后一个后台线程会将 `pickle` 的数据 `flush` 到底层的管道中
4. 如果多个进程向同一个队列写数据，则取的数据的顺序可能与写的顺序不同
   - 如果一个进程向一个队列写数据，则取的数据的顺序与写的顺序相同

3.2.1 Queue

1. API：

```
class multiprocessing.Queue([maxsize])
```

   `Queue` 是通过一些 `lock/semaphore` 和管道来实现的进程间共享队列。除了 `task_done(),join()` 之外，它几乎实现了 `queue.Queue` 的所有方法。

2. 方法：
   - `.qsize()`：返回队列的大概大小。因为多进程/多线程的语义，这个数字是不可靠的
   - `.empty()`：返回队列是否为空。因为多进程/多线程的语义，这个结果是不可靠的
   - `.full()`：返回队列是否为满。因为多进程/多线程的语义，这个结果是不可靠的
   - `.put(obj[,block[,timeout]])`：将对象 `obj` 入队列。
     - 如果 `block=False`：如果队列有空槽，则将 `obj` 入队，并立即返回。否则队列为满，则抛出 `queue.Full` 异常
     - 如果 `block=True`：
       - 如果 `timeout=None`：如果队列有空槽，则将 `obj` 入队，并立即返回。否则队列为满，则阻塞进程，直到队列有空槽
       - 如果 `time` 为正数：如果队列有空槽，则将 `obj` 入队，并立即返回。否则队列为满，则等待不超过 `time` 秒。如果超时，则抛出 `queue.Full` 异常
   - `.put_nowait(obj)`：等价于 `.put(obj,False)`
   - `.get([block[,timeout]])`：出队列获取数据。
     - 如果 `block=False`：如果队列有数据，则出队列并返回弹出的数据。否则队列为空，则抛出 `queue.Empty` 异常
     - 如果 `block=True`：
       - 如果 `timeout=None`：如果队列有数据，则出队列并返回弹出的数据。否则队列为空，则阻塞进程，直到队列有数据
       - 如果 `time` 为正数：如果队列有数据，则出队列并返回弹出的数据。否则队列为空，则等待不超过 `time` 秒。如果超时，则抛出 `queue.Empty` 异常
   - `.get_nowait()`：等价于 `.get(False)`
   - `.close()`：关闭队列。该方法由垃圾收集过程自动调用。

- `.join_thread()`：`join` 底层的数据`flush` 线程。它仅仅能在 `.close()` 方法之后调用。该方法通常由垃圾收集过程自动调用。
- `.cancel_join_thread()`：阻止 `.join_thread()` 处于阻塞状态，从而不必`join` 底层的数据`flush` 线程而退出。该方法可能导致数据丢失，建议尽量少用。

3.2.2 SimpleQueue

1. API:

```
class multiprocessing.SimpleQueue
```

它是一个简化版的队列，非常接近于一个加锁的管道。

2. 方法：
   - `.empty()`：返回队列是否为空。
   - `.get()`：出队列并返回弹出的数据。
   - `.put(item)`：将数据入队列

3.2.3 JoinableQueue

1. API：

```
class multiprocessing.JoinableQueue([maxsize])
```

`JoinableQueue` 是 `Queue` 的子类，它额外实现了 `.task_done()/.join()` 方法

2. 对于 `JoinableQueue`，对于队列里每一个对象，当你把它从队列中取出时，必须调用 `JoinableQueue.task_done()`
   - 队列维护一个 `unfinished` 信号量；每当有对象入队列时，该信号量加1；当调用一次 `JoinableQueue.task_done()` 时，该信号量减1。
   - 如果出队列时，未调用 `JoinableQueue.task_done()` ，则可能信号量溢出（此时会抛出异常）
3. 方法：
   - `.task_done()`：对于每一个`.get()` 获取的对象，必须紧接着调用该方法。该方法意味着数据的一个完成的`入队列- 出队列` 流程结束
     - 该方法由数据的消费者调用
     - 如果调用的次数超过了数据入队列的次数，则抛出 `ValueError` 异常
   - `.join()` ：阻塞进程，直到队列中所有的数据都得到处理
     - 所谓的得到处理，指的是：`.get()` 获取数据之后，紧接着调用`.task_done()`
     - 当生产者写入队列时，未处理数据的数量增加；当消费者调用 `.task_done()` 时，未处理数据的数量降低。当未处理的数据数量为 0 时，解除阻塞

四、进程间同步

1. `multiprocessing` 使用的进程间同步原语几乎与`threading` 模块相同
2. 相比较于多线程环境，多进程环境中比较少的使用到同步原语。
   - 你可以仅仅通过 `Manager` 对象来创建同步原语。

3. `Barrier`：用法类似于`threading.Barrier`

```
class multiprocessing.Barrier(parties[, action[, timeout]])
```

4. `BoundedSemaphore`：用法类似于`threading.BoundedSemaphore`

```
class multiprocessing.BoundedSemaphore([value])
```

5. `Condition`：用法类似于`threading.Condition`

```
class multiprocessing.Condition([lock])
```

6. `Event`：用法类似于`threading.Event`：

```
class multiprocessing.Event
```

7. `Lock`：用法类似于`threading.Lock`：

```
class multiprocessing.Lock
```

   与`threading.Lock`相比，`multiprocessing.Lock`有以下几点不同：

   - `.acquire(block=True, timeout=None)`：
     - `threading.Lock`中，第一个参数为`blocking`，而这里为`block`
     - `threading.Lock`中，`timeout`的意义为：-1 表示永不超时。而这里`timeout`为`None`表示永不超时，为负数等价于`timeout=0`
   - `.release()`：
     - `threading.Lock`中，如果对一个未加锁的`Lock`调用`.release()`，则抛出`RuntimeError`。而这里会抛出`ValueError`

8. `RLock`：用法类似于`threading.RLock`：

```
class multiprocessing.RLock
```

   与`threading.RLock`相比，`multiprocessing.RLock`有以下几点不同：

   - `.acquire(block=True, timeout=None)`：
     - `threading.Lock`中，第一个参数为`blocking`，而这里为`block`
     - `threading.Lock`中，`timeout`的意义为：-1 表示永不超时。而这里`timeout`为`None`表示永不超时，为负数等价于`timeout=0`
   - `.release()`：
     - `threading.Lock`中，如果对一个未加锁的`Lock`调用`.release()`，则抛出`RuntimeError`。而这里会抛出`AssertionError`

9. `Semaphore`：用法类似于`threading.Semaphore`：

```
class multiprocessing.Semaphore
```

五、进程间数据共享

1. 通常尽量不要在进程间共享数据，因为你需要维护数据的一致性。但是 `multiprocessing` 也提供了进程间共享数据的接口。
    - `multiprocessing` 在共享内存中创建了共享对象，这些对象被子进程继承和访问。

5.1 Value

1. `API`：

```
multiprocessing.Value(typecode_or_type, *args, lock=True)
```

该函数返回一个`ctypes` 对象，该`ctypes` 对象封装了一个从共享内存区分配的共享数据。

> `ctypes` 模块封装了一些兼容`C` 的数据类型。

你可以通过`ctypes` 对象的 `.value` 属性来访问共享数据

```
from multiprocessing import Process, Value
if __name__=='__main__':
    num = Value('d', 0.0)
    p = Process(target=f, args=(num,))
```

2. 参数：
    - `typecode_or_type`：决定了共享数据的数据类型。
        - 可以是 `ctype` 的类型。如 `ctypes.c_bool/ctypes.c_char/...`
        - 可以是单个字符。如`'d'`
    - `*args`：传给了`ctypes` 对象的初始化函数
    - `lock`：如果为`True`，则自动创建一个锁来控制共享数据的访问，此时该共享数据的访问是进程安全的。如果为`False`，则不会自动创建锁，此时对该共享数据的访问不是进程安全的。
3. 共享数据的`+=` 操作符并不是原子的，因此你需要使用：

```
with xxx.get_lock():
    xxx.value+=1
```

5.2 Array

1. API：

```
multiprocessing.Array(typecode_or_type, size_or_initializer, *, lock=True)
```

该函数返回一个`ctypes array` ，该`ctypes array`封装了一个从共享内存区分配的共享`array`。

```
from multiprocessing import Process, Array
if __name__=='__main__':
    arr = Array('i', range(10))
    p = Process(target=f, args=(arr,))
```

2. 参数：
   - `typecode_or_type`：决定了共享`array`的元素的数据类型。
     - 可以是`ctype`的类型
     - 可以是单个字符
   - `size_or_initializer`：如果是个整数，则它给出了共享`array`的长度，此时数组被初始化为0；如果是个序列，则它用于给出数组的长度和内容。
   - `lock`：如果为`True`，则自动创建一个锁来控制共享数据的访问，此时该共享数据的访问是进程安全的。如果为`False`，则不会自动创建锁，此时对该共享数据的访问不是进程安全的。

5.3 sharedctypes

1. 如果你需要在共享内存中存储任意类型的对象，则使用`multiprocessing.sharedctypes`模块
   - 它让子进程继承了来自父进程的共享存储
2. `RawArray`：

   ```
   multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)
   ```

   该函数返回一个`ctypes array`，该`ctypes array`封装了一个从共享内存区分配的共享`array`。
   - 它类似于`multiprocessing.Array`，但是没有使用任何锁机制
3. `RawValue`：

   ```
   multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)
   ```

   该函数返回一个`ctypes`对象，该`ctypes`对象封装了一个从共享内存区分配的共享数据。
   - 它类似于`multiprocessing.Value`，但是没有使用任何锁机制
4. `Array`：

   ```
   multiprocessing.sharedctypes.Array(typecode_or_type, size_or_initializer, *, lock=True)
   ```

   - 它就是`multiprocessing.Array`的底层实现
5. `Value`：

   ```
   multiprocessing.sharedctypes.Value(typecode_or_type, *args, lock=True)
   ```

   - 它就是`multiprocessing.Value`的底层实现
6. `multiprocessing.sharedctypes.copy(obj)`：在共享存储中创建一份`obj`对象的拷贝。其中`obj`是一个`ctypes`对象

7. `multiprocessing.sharedctypes.synchronized(obj[,lock])`：创建一个进程安全的`ctypes`对象，它就是`obj`的拷贝。
    - 如果`lock=None`，则默认使用`multiprocessing.RLock`对象
    - 返回的对象有两个方法：
        - `.get_obj()`：返回被封装的对象
        - `.get_lock()`：返回同步用的锁
    - 返回的对象支持上下文管理器协议

5.4 Manager

1. `Manager`对象会创建一个`server`进程，用于持有共享数据。
    - 所有进程通过`Manager`提供的代理访问这些共享数据
    - 在内部是通过`socket`进行通信，从而使得无关进程（非父子进程）之间也能通信
    - 它的灵活性较高，可以共享多种`python`数据类型，但是访问速度比`multiprocessing.Value,multiprocessing.Array`要慢。
2. API：

```
multiprocessing.Manager()
```

   创建并返回一个`SysncManager`对象
    - 该对象可以创建一个`server`进程
    - 该对象提供一些方法，这些方法会创建共享数据并返回共享数据的代理对象。你可以通过代理对象来访问底层的共享数据。
3. `server`进程在两种情况下会退出：
    - `Manager`对象被垃圾收集过程处理
    - 派生`server`的父进程退出

5.4.1 BaseManager

1. API：

```
class multiprocessing.managers.BaseManager([address[, authkey]])
```

   参数：
    - `address`：一个监听地址。`server`进程将在这个地址上监听新的连接。如果为`None`，则将任意选择一个
    - `authkey`：一个验证`key`。`server`进程将用它来验证连接是否有效。如果为`None`，则使用`current_process().authkey`
2. 一旦创建了`Manager`对象，你应该调用`.start()`方法或者`.get_server().serve_forever()`来确保`Manager`对象启动了一个`server`进程
3. 方法：
    - `.start([initializer[, initargs]])`：启动`server`进程。
        - 如果`initializer`非`None`，则`server`子进程在开始时会调用`initializer(initargs)`

- `.get_server()`：获取代表`server`进程（由该`Manager`管理） 的`Server`对象。
  - `Server`对象的`.serve_forever()`会启动`server`子进程

    > 这是另一种启动`serer`进程的方式

  - `Server`对象的`address`属性给出了`server`子进程监听的地址
- `.connect()`：将一个`manager`对象连接到`server`进程。这是客户端进程访问`server`进程所需要执行的动作。

```
from multiprocessing.managers import BaseManager
m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
m.connect() #客户端进程访问 server 进程
```

- `.shutdown()`：关闭`server`进程
  - 当且仅当通过`.start()`方法启动`server`进程时，`.shutdown()`方法可用
  - 该方法可以反复调用
- `.register(typeid[,callable[,proxytype[,exposed[,method_to_typeid[,create_method]]]]])`：它是一个类方法，作用是：注册一个共享数据类型，或注册一个可调用对象
  - `typeid`：代表共享数据的类型标识符，是个字符串。
  - `callable`：用于创建指定类型（由`typeid`指定）的共享数据的可调用对象。
    - 如果`create_method`为`False`，则该参数可以为`None`
    - 如果`Manager`对象随后使用`.connect()`方法链接到`server`进程，则该参数可以为`None`。因为此时说明是客户端进程在访问`server`，而客户端进程不需要创建共享数据。
  - `proxytype`：是一个`BaseProxy`的子类。它用于为共享数据创建代理。
    - 该代理用于管理共享数据的访问
    - 如果为`None`，则自动给出
  - `exposed`：给出了一些方法名列表，指定了代理可以使用哪些方法来访问共享数据。若未指定，则使用默认值（所有的、不是以`_`开始的方法名）
  - `method_to_typeid`：一个映射，给出了每一个`exposed`方法返回的数据类型。
  - `create_method`：一个布尔值。决定了是否应该使用`typeid`创建一个方法，`server`进程可以通过该方法来创建一个共享数据并返回一个代理。

4. 属性：
   - `address`：一个只读的属性，给出了`server`进程监听的端口地址。
5. `Manager`对象支持上下文管理器协议
   - 其`.__enter__()`方法会通过`.start()`方法启动`server`服务器（当它未启动时）
   - 其`.__exit__()`方法会通过`.shutdown()`方法关闭`server`服务器

5.4.2 SyncManager

1. `SyncManager`是`BaseManager`的子类，它提供了一些方法来创建共享数据及其代理
   - `.Barrier(parties[, action[, timeout]])`：创建了一个共享的`threading.Barrier`对象，然后返回它的代理

- `.BoundedSemaphore([value])`：创建了一个共享的`threading.BoundedSemaphore`对象，然后返回它的代理
- `.Condition([lock])`：创建了一个共享的`threading.Condition`对象，然后返回它的代理
  - 如果给定的`lock`参数，则它必须是`threading.Lock/threading.RLock`对象的代理
- `.Event()`：创建了一个共享的`threading.Event`对象，然后返回它的代理
- `.Lock()`：创建了一个共享的`threading.Lock`对象，然后返回它的代理
- `.Namespace()`：创建了一个共享的`Namespace`对象，然后返回它的代理
- `.Queue([maxsize])`：创建了一个共享的`queue.Queue`对象，然后返回它的代理
- `.RLock()`：创建了一个共享的`threading.RLock`对象，然后返回它的代理
- `.Semaphore([value])`：创建了一个共享的`threading.Semaphore`对象，然后返回它的代理
- `.Array([typecode, sequence])`：创建了一个共享的`Array`对象，然后返回它的代理
- `.Value([typecode, value])`：创建了一个共享的`Value`对象，然后返回它的代理
- `.dict()`：创建了一个共享的字典对象，然后返回它的代理
- `.dict(mapping)`：创建了一个共享的字典对象，然后返回它的代理
- `.dict(sequence)`：创建了一个共享的字典对象，然后返回它的代理
- `.list()`：创建了一个共享的列表对象，然后返回它的代理
- `.list(sequence)`：创建了一个共享的列表对象，然后返回它的代理

2. 示例：

```
from multiprocessing import Process, Manager
if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict() #d 是一个代理，对共享的字典的访问通过它进行
        l = manager.list(range(10)) #l 是个代理，对列表的访问通过它进行
        p = Process(target=f, args=(d, l))
```

3. 从`python 3.6`开始，共享对象可以嵌套：如共享的列表对象中，可以包含共享的字典对象。（它们都由`manager`对象管理）

4. `Namespace`对象：

```
class multiprocessing.managers.Namespace
```

它不包含任何公开的方法，仅仅包含一些可以读写的属性

- 带`_`前缀的属性是它的代理的属性，而不是它自己的属性

```
manager = multiprocessing.Manager()
Global = manager.Namespace()
Global.x = 10        #Namespace 的属性
Global.y = 'hello'   #Namespace 的属性
Global._z = 12.3     #代理的属性
```

5.4.3 自定义 Manager

1. 自定义的 `Manager` 只需要子类化`BaseManager` ，并且利用 `register()` 类方法来注册你所需的共享数据类型即可

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
class MyManager(BaseManager):
    pass
MyManager.register('Maths', MathsClass)
if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
```

5.4.4 远程 Manager

1. 有些时候， `server` 进程运行在一台机器上，而需要访问共享数据的子进程运行在另一台机器上
2. 示例：
    - 运行 `server` 进程的机器：

```
from multiprocessing import Process, Queue
from multiprocessing.managers import BaseManager
class Worker(Process): #自定义的 Process
    def __init__(self, q):
        self.q = q
        super(Worker, self).__init__()
    def run(self):
        self.q.put('local hello')
class QueueManager(BaseManager):  #自定义的 Manager
    pass
QueueManager.register('get_queue', callable=lambda: queue) #server 会使用共享的
queue
# 注册的名字 get_queue 是一个可调用对象，而不是一个类型

if __name__=='__main__':
    queue = Queue()  #父进程的 queue
    w = Worker(queue) #本地的子进程访问共享的queue
    w.start()
    ....

    m = QueueManager(address=('', 50000), authkey=b'abracadabra')
    s = m.get_server()
    s.serve_forever()#这里启动 server 进程，server 进程拥有自己的 queue (fork 而来)
```

- 访问共享内存的机器：

```
from multiprocessing.managers import BaseManager
class QueueManager(BaseManager): pass
QueueManager.register('get_queue')
m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
#address 为server 进程监听的地址
m.connect()#客户进程访问 server
queue = m.get_queue()
queue.put('hello')
queue.get()
```

5.5 Proxy

1. `Proxy` 对象引用了存活在不同进程中的共享数据对象。
    - 该共享数据对象称作代理对象的 `referent`
    - 多个代理对象可能拥有同一个 `referent`
2. 代理对象有 `referent` 同样的接口

- 对代理对象调用 `str()`，将返回的是 `referent` 的信息
- 对代理对象调用 `repr()`，将返回的是代理对象的信息

3. 代理对象是支持 `pickle` 的，因此可以在进程之间传递代理对象。因此一个 `referent` 可以包含某个代理对象。

4. 如果 `referent` 包含了 `list/dict` 等可变对象，则对 `list/dict` 的修改并不会同步到共享内存中。你必须触发代理对象的 `.__setitem__` 方法来通知 `manager` 来同步这个修改

```
lproxy = manager.list()
lproxy.append({}) # referent 包含字典
d = lproxy[0]
## 对字典进行修改
d['a'] = 1
d['b'] = 2
# 现在代理对象根本不知道发生了这个修改
lproxy[0] = d#同步
```

5. 代理对象不支持基于值的比较：

```
manager.list([1,2,3]) == [1,2,3] #返回 False
```

5.5.1 BaseProxy

1. API：

```
class multiprocessing.managers.BaseProxy
```

所有代理对象的基类。

2. 方法：
   - `._callmethod(methodname[,args[,kwds]])`：调用 `referent` 对象的 `methodname` 方法，并返回其结果。即在 `manager` 进程中调用：`getattr(referent,methodname)(*args,**kwds)`
     - 返回值会进行一次 `value copy`
     - 如果 `methodname` 调用时抛出异常，则在 `._callmethod` 方法中重新抛出
     - 如果 `methodname` 并没有被 `exposed` ，则抛出异常
   - `._getvalue()`：返回 `referent` 的一份拷贝。若 `referent` 不支持 `pickle`，则抛出异常
   - `.__repr__()`：返回代理对象的一份 `reprensentaion`
   - `.__str__()`：返回 `referent` 对象的一份 `reprensentaion`

六、进程池

1. 可以通过 `Pool` 类来创建一个进程池来完成一组任务。
   - 一个进程池对象管理了一组工作进程。

2. API：

```
class multiprocessing.pool.Pool([processes[, initializer[, initargs[,
maxtasksperchild[, context]]]]])
```

参数：

- `processes`：指定工作进程的数量。如果为`None`，则使用`os.cpu_count()`的返回值。
- `initializer`：如果不是`None`，则每个工作进程开启时，执行`initializer(*initargs)`
- `maxtasksperchild`：一个工作进程在退出之前可以完成的任务数。
  - 一旦超过了这个数，则该工作进程退出，然后派生一个新的工作进程。（某些场景下，如mod_wsgi 中会使用这个功能来周期性的释放资源）
  - 如果为`None`，则只要进程池存在工作进程就不退出
- `context`：指定了工作进程的上下文。

3. 方法：(进程池的方法只能在创建进程池的那个进程被调用)
   - `.apply(func[, args[, kwds]])`：在进程池的某个工作进程中执行`func(*args,**kwds)`，返回其调用结果
     - 选择哪个工作进程是不确定的
     - 当前进程会被阻塞，直到`func`返回
     - 通常会建议使用并行的`.apply_async()`方法，因为它不会阻塞当前进程
   - `.apply_async(func[, args[, kwds[, callback[, error_callback]]]])`：`.apply`方法的异步版本
     - `func`为待调用的任务，`args,kwds`为任务的位置参数和关键字参数
     - `callback`：如果指定了该参数，则它必须是一个可调用对象，并且接收一个参数。
       - 当`func`成功结束时，`callback`被调用，参数就是`func`的返回值。
     - `error_callback`：如果指定了该参数，则它必须是一个可调用对象，并且接收一个参数。
       - 当`func`抛出异常，则`error_callback`被调用，参数就是`func`抛出的异常对象
   - `.map(func,iterable[,chunksize])`：内建的`map()`函数的并行版本。（对每个元素调用`func`并返回一个列表）
     - 该方法会阻塞当前进程，直到结果返回。
     - 该方法将`iterable`划分成一些数据块，然后作为一个任务分配到工作进程中。数据块的大小由`chunksize`指定（默认为1）
   - `.map_async(func,iterable[, chunksize[,callback[, error_callback]]])`：`.map`方法的异步版本
     - `callback`：如果指定了该参数，则它必须是一个可调用对象，并且接收一个参数。
       - 当结果可用时，`callback`被调用，参数就是返回的结果。
     - `error_callback`：如果指定了该参数，则它必须是一个可调用对象，并且接收一个参数。
       - 当`func`抛出异常，则`error_callback`被调用，参数就是`func`抛出的异常对象
     - 这两个`callback`必须要很快完成，否则他会阻塞当前进程
   - `.imap(func,iterable[,chunksize])`：`.map`方法的一个懒加载的版本（类似于生成器，结果不是一次性获取），它返回一个迭代器对象（而不是一个列表）
     - 对该迭代器对象调用`.next(timeout)`方法时，可以指定一个`timeout`超时参数。当发生超时，抛出`multiprocessing.TimeoutError`

- `.imap_unordered(func,iterable[,chunksize])`：`.imap` 方法的无序版本。它返回一个迭代器对象，对该迭代器的迭代结果可能是任意顺序（与输入数据的顺序不匹配）。
- `.starmap(func,iterable[,chunksize])`：类似 `.map` 方法，唯一区别在于：`iterable` 中的每个元素作为参数传递给 `func` 时，都被解包：

```
iterable=[(1,2),(3,4)]
myPool.starmap(func,iterable)# 参数为：func(1,2)
```

- `.starmap_async(func,iterable[, chunksize[,callback[, error_callback]]])`：`starmap` 的异步版本
- `.close()`：关闭进程池
  - 调用该方法之后，不再允许向进程池提交新的任务。一旦所有现有的工作进程的任务完成，则会关闭进程池。
- `.terminate()`：终止进程池
  - 调用该方法之后，不再允许向进程池提交新的任务。同时现有的工作进程立即终止（不管其是否还有任务）。
  - 当进程池被垃圾收集过程处理时，该方法被立即调用
- `.join()`：等待工作进程终止。必须在调用它之前调用 `.close()` 或者 `.terminate()`

4. 进程池对象支持上下文管理器协议：
   - `.__enter__()` 方法返回该进程池对象
   - `.__exit__()` 方法会调用 `.terminate()` 方法
5. `AysncResult` 对象代表异步执行的结果：

```
class multiprocessing.pool.AsyncResult
```

   - 该对象由进程池的异步方法返回
   - 该对象的方法：
     - `.get([timeout])`：获取结果。如果设定了超时时间并且发生超时，则抛出 `multiprocessing.TimeoutError` 异常
     - `.wait([timeout])`：等待结果可用。如果设定了超时时间并且发生超时，则抛出 `multiprocessing.TimeoutError` 异常
     - `.ready()`：返回一个布尔值，指示异步调用是否完成
     - `.successful()`：返回一个布尔值，指示异步调用是否成功（没有抛出异常）
       - 如果异步调用还未完成，则抛出 `AssertionError` 异常

七、 connection

1. `Connection` 对象使用管道来进行进程间通信，但是`multiprocessing.connection` 模块提供了更多的进程间通信功能。
2. `multiprocessing.connection.deliver_challenge(connection, authkey)`：向对端发送一个随机生成的消息，然后等待响应
   - 如果响应OK，则向对端发送一个`welcome message`。否则抛出一个 `AuthenticationError` 异常

3. `multiprocessing.connection.answer_challenge(connection,authkey)`：接收一个消息，通过`authkey`作为`key`来计算消息的摘要，并向对端发送摘要。
    - 如果未收到`welcome message`，则抛出一个`AuthenticationError`异常
4. `Client`：代表一个客户端对象：

```
multiprocessing.connection.Client(address[, family[, authenticate[, authkey]]])
```

试图向`listener`对象（由`address`地址指定的）发送一个连接，并返回一个`Connection`对象代表这个连接。

- `address`：决定了`listener`的地址
- `family`：决定了连接的类型。事实上当连接类型可以从`address`的格式中推断时，该参数可以省略
- `authenticate`：确定是否需要验证。
    - 如果为`True`，则`authkey`是一个字符串，给出了计算摘要需要的`key`。如果未给出`authkey`，则抛出一个`AuthenticationError`异常

5. `Listener`：代表一个监听器对象：

```
class multiprocessing.connection.Listener([address[, family[, backlog[, authenticate[,
authkey]]]]])
```

它封装了一个绑定的套接字或者一个命名管道。

参数：

- `address`：监听器监听的地址。
- `family`：给定了套接字的类型。可以为：
    - `'AF_INET'`：为`TCP`套接字
    - `'AF_UNIX'`： 为`Unix domain socket`
    - `'AF_PIPE'`：为`windows`命名管道
    - 如果为`None`，则`socket`类型从`address`的格式中推断
- `backlog`：如果监听器使用一个`socket`，则该参数会传给`socket`的`listen()`方法。该参数的默认值为1
- `authenticate`：确定是否需要验证，如果验证失败则抛出一个`AuthenticationError`异常。如果为`False`则不需要验证；如果为`True`：
    - 若`authkey`是一个字符串，给出了计算摘要需要的`key`。
    - 若`authkey=None`，则使用`current_process().authkey`。

方法：

- `.accept()`：从`bound socket`或者`named pipe`上接收到一条连接，并且返回一个`Connection`对象。如果验证失败，则抛出一个`AuthenticationError`异常
- `.close()`：关闭监听器的`bound socket`或者`named pipe`
    - 当监听器被垃圾收集过程处理时，该方法被自动调用

属性：

- `address`：监听器监听的地址
- `.last_accepted`：最近一次接收到的连接来自的地址。如果尚未收到连接，则为`None`

6. `Listener` 对象支持上下文管理协议
    - `.__enter__()` 方法返回这个监听器对象
    - `.__exit__()` 方法调用 `.close()` 方法
7. `multiprocessing.connection.wait(object_list, timeout=None)`：等待，直到 `object_list` 列表中任何一个对象可用，或者超时。返回 `object_list` 中那些可用的对象。
    - 如果 `timeout=None`，则表示永不超时。如果为负值，则等价于 `timeout=0`
    - 能够出现在 `object_list` 中的对象类型可以为：
        - 可读的 `Connection` 对象（可读表示连接中有数据，或者对端关闭）
        - 已连接且可读的 `socket.socket` 对象（可读表示连接中有数据，或者对端关闭）
        - `Process` 对象的 `sentinel` 属性
8. 示例：
    - 监听器

```
from multiprocessing.connection import Listener
from array import array
address = ('localhost', 6000)     # family is deduced to be 'AF_INET'
with Listener(address, authkey=b'secret password') as listener: #监听器
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)
        conn.send([2.25, None, 'junk', float])
        conn.send_bytes(b'hello')
        conn.send_bytes(array('i', [42, 1729]))
```

    - 客户端

```
from multiprocessing.connection import Client
from array import array
address = ('localhost', 6000)
with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]
    print(conn.recv_bytes())            # => 'hello'
    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)
```

八、日志

1. 由于 `logging` 模块并没有使用任何进程同步机制，因此直接在多进程环境中使用它可能会引起多个进程的日志消息相互混杂的问题。
2. `multiprocessing.get_logger()`：返回一个用于多进程环境的日志对象。
3. `multiprocessing.log_to_stderr()`：它在 `get_logger()` 的基础上，添加了一个 `handler`（它向标准错误输出信息，格式为：`'[%(levelname)s/%(processName)s] %(message)s'`）

九、编程指导

1. 尽量避免进程之间共享状态。
    - 尽量避免在进程之间移动大量的数据
    - 如果必须通信，则坚持使用`queue`或者管道。而不要使用低级的同步原语。
2. 确保代理对象的方法的参数支持`pickle`
3. 确保代理对象的使用在单线程中使用。如果在多线程中使用，则必须使用锁机制。
4. 对僵尸进程执行`join`
    - 实时上，当调用`.active_children()`时，对所有已完成的子进程自动的调用了`join`。而调用`Process.is_alive`时，对该进程（若已完成）也自动调用了`join`
    - 但是还是推荐显式的调用`join`
5. 尽量用继承，而不是`pickle/unpickle`来传递数据。
    - 尽量安排子进程通过派生而从父进程取得共享的数据，而不是通过`queue`或者管道来传递数据。
6. 避免`terminate`一个进程。
    - 如果对一个进程调用`Process.terminate()`方法，则该进程会立即终止，从而导致一些锁、管道、信号量、队列等共享对象出现问题。
    - 因此当且仅当一个进程没有使用任何共享对象时，才可以对他调用`terminate`
7. 仔细安排那些使用`queue`的进程的`join`：
    - 当一个进程向`Queue`写入数据时，确保写入的数据在进程被`join`之前被读取。否则容易死锁：

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()  #queue 有数据未被读取，死锁
    obj = queue.get()
```

        - 正确的做法是交换最后两行：先获取数据，再`join`
    - 非`non-daemonic`进程可能被自动`join`
8. 显式向子进程传递资源。有两个原因：
    - 更好的在`Unix/Windows`上统一接口
    - 防止父进程的资源被垃圾收集处理收回。

```python
from multiprocessing import Process, Lock

def f():
    # 子进程中，隐式传递了 "lock"
def f2(l):
    # 子进程中，显式式传递了 "lock"
if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
        Process(target=f, args=(lock,)).start()
```