

# 类设计模式、并发编程 C04



北京大学计算机学院  
胡俊峰 2022-03-07

# 关于助教排班及作业布置


- 常规课程辅导助教排班表，负责课程作业布置及网上答疑。
  - 课程后期应该会有些调整，回头再另行通知
- 周一作业的发布时间在下午7点，周四作业发布时间在下午4点。提交截止时间由助教群内通知。

日期	周数	星期	负责助教
2.21	1	一	姜和丰
2.28	2	一	姜腾
3.3	2	四	郭明非
3.7	3	一	李隽仁
3.14	4	一	姜和丰
3.17	4	四	郭明非
3.21	5	一	姜腾
3.28	6	一	李隽仁
3.31	6	四	郭明非
4.2	7	六	姜和丰
4.11	8	一	姜腾
4.14	8	四	郭明非
4.18	9	一	李隽仁
4.25	10	一	姜和丰
4.28	10	四	郭明非
5.2	11	一	放假
5.9	12	一	姜腾
5.12	12	四	郭明非
5.16	13	一	李隽仁
5.23	14	一	姜和丰
5.26	14	四	郭明非
5.3	15	一	姜腾
6.6	16	一	李隽仁
6.9	16	四	郭明非




# 上次课重要内容回顾

- 基于self指针的对象生成方案。
- 可执行对象-可迭代对象-类装饰器的实现



# 本次课内容提要

- Python类体系与架构编程
  - 模块 (Module)
  - 进程、线程
- 

# Python类体系与架构编程

- 静态方法、类方法、实例方法
- 单例模式
- 工厂模式
- 发布-订阅模式

# Python设计模式

- 设计模式是面型对象方法里的一种解决方案的抽象
- 目的是把一些常见的应用抽象为一种类设计模式，在具体实现中只要套用或稍作修改，就能完成逻辑清晰的类实现方案
- 通过对设计模式的学习也可以达到对类体系的深入理解

# 单例模式 — 所有生成实例都指向同一个对象

```
class Singleton(object):  
  
    attr = None                # 类属性  
  
    def __init__(self):  
        print("Do something.") ← 调用 __new__()方法创建实例对象  
  
    def __new__(cls, *args, **kwargs):                # 重载__new__()方法  
        if not cls.attr:  
            cls.attr = super(Singleton, cls).__new__(cls)  
  
        return cls.attr ←  
  
obj1 = Singleton()  
obj2 = Singleton()  
print(obj1, obj2)
```

Do something.

Do something.

<\_\_main\_\_.Singleton object at 0x00000215E55BC308> <\_\_main\_\_.Singleton object at 0x00000215E55BC308>

# 单例模式 — 所有生成实例都指向同一个对象

```
class Singleton(object):
    def __init__(self):
        print("Do something new.")

    def __new__(cls, *args, **kwargs):
        if not hasattr(Singleton, "_instance"):
            Singleton._instance = object.__new__(cls)

        return Singleton._instance
```

在cls空间动态生成一个内部属性  
(另一种实现单例模式的方案)

```
obj1 = Singleton()
obj2 = Singleton()
print(obj1, obj2)
```

Do something new.

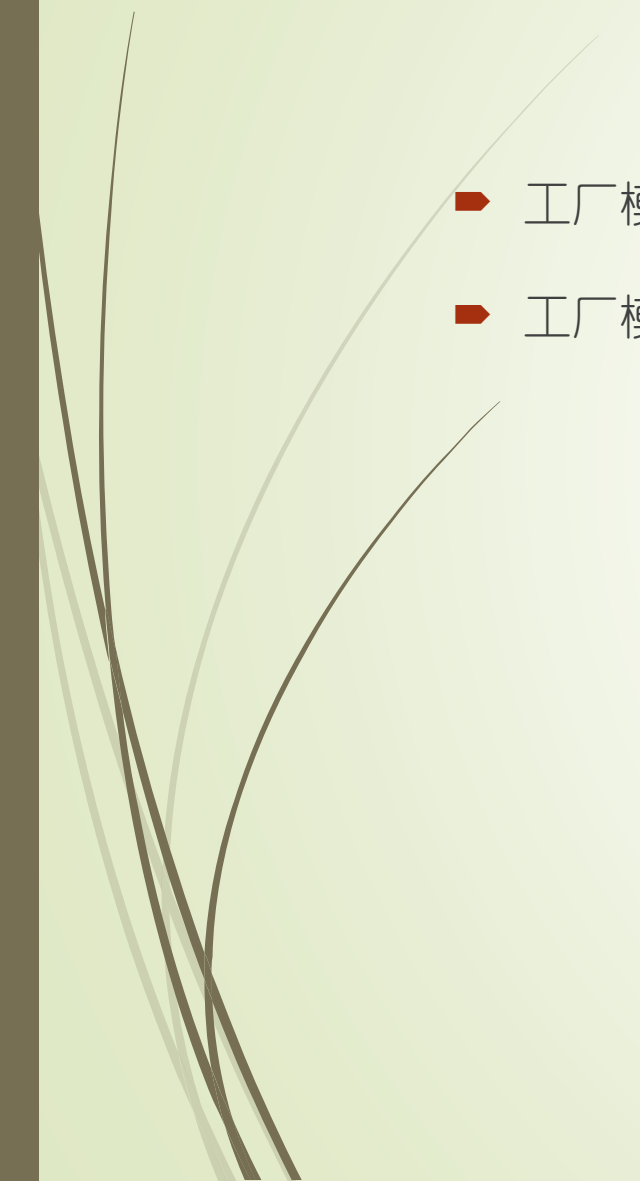
Do something new.

<\_\_main\_\_.Singleton object at 0x00000215E55B2A88> <\_\_main\_\_.Singleton object at 0x00000215E55B2A88>





# 工厂模式简介：用来实现参数化定制类

- 工厂模式的本质上是用类定制类，然后到具体实例
  - 工厂模式的基础是共性抽象，是把相关类的共性和个性化定制相融合的解决方案
- 

# 工厂类的示例：

```
class StandardFactory(object):
```

```
    @staticmethod                # 静态方法
```

```
    def get_factory(factory):    # 实际可以传入更多的参数
```

```
        ''' 根据参数找到对实际操作的工厂'''
```

```
        if factory == 'cat':
```

```
            return CatFactory() # 这里如果要带参数，就会用到类属性，类方法
```

```
        elif factory == 'dog':
```

```
            return DogFactory()
```

```
        raise TypeError('Unknown Factory.')
```

```
class DogFactory(object):
```

```
    def get_pet(self): # 这里还可以带参数，甚至组合其他类，来定义不同类的dog
```

```
        return Dog(); # 返回一个dog类的实例
```

```
class CatFactory(object):
```

```
    def get_pet(self):
```

```
        return Cat();
```

# 工厂类的示例（抽象类）：

```
class Pet(abc.ABC):          # 抽象类可以通过MyIterable方法来查询所有的派生子类
    @abc.abstractmethod     # 强制子类必须实现此方法
    def eat(self):
        pass

    def jump(self):          # 不能创建实例，但可以被继承
        print("jump...")

# Dog类的具体实现
class Dog(Pet):
    def eat(self):          # 必须实现抽象类Pet中规定的方法
        return 'Dog eat...'

class Cat(Pet):
    def eat(self):
        return 'Cat eat...'
```

## 工厂类的示例（实际使用）：

```
if __name__ == "__main__":    # 如果被包含则 __name__ 会等于模块名，下面代码不会执行
    factory = StandardFactory.get_factory('cat')    # 配置抽象工厂参数，生成一个猫工厂
    cat = factory.get_pet()    # 生成一个猫实例
    print (cat.eat())    # cat eat

    factory = StandardFactory.get_factory('dog')
    dog = factory.get_pet()    ##这里工厂的操作与上面的生成cat是完全一样的，但结果不同
    print (dog.eat())    # dog eat
    dog.jump()    # 继承自抽象类的jump
    cat.jump()
    #Pet().jump() TypeError: Can't instantiate abstract class Pet with abstract
```

Cat eat...

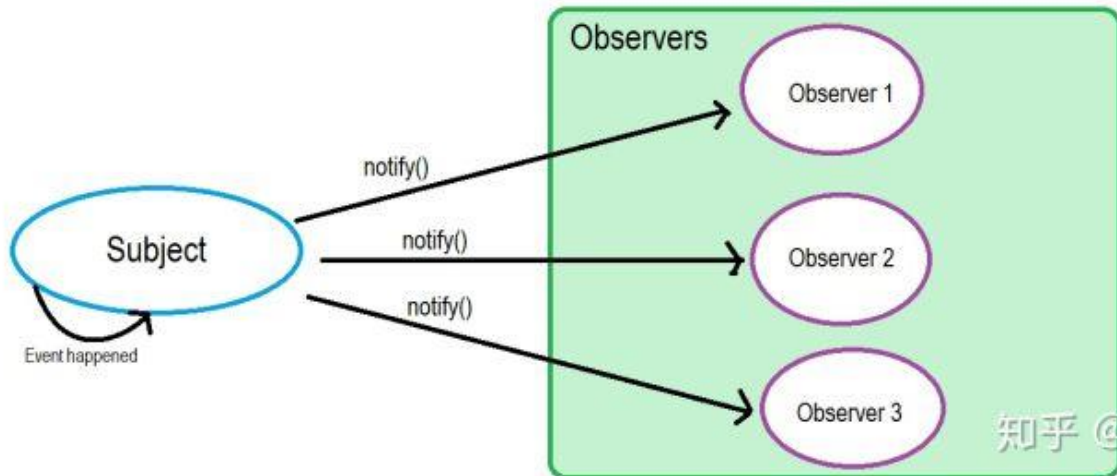
Dog eat...

jump...

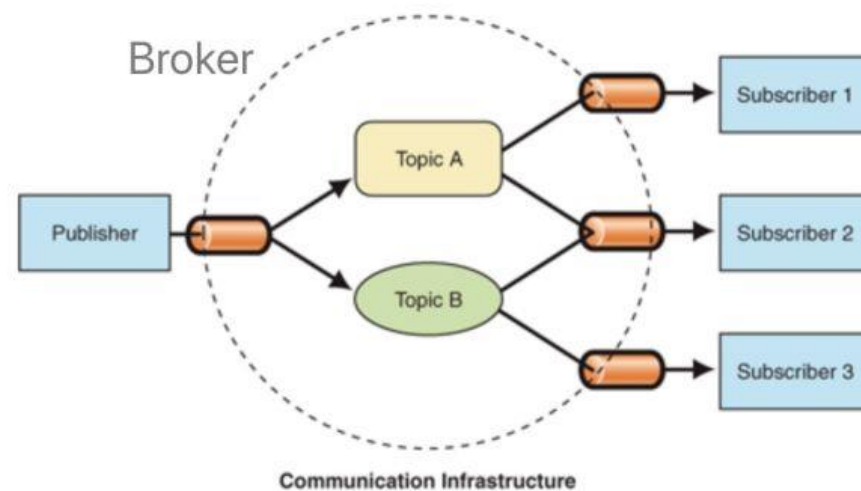
jump...

# 观察者模式（发布-订阅模式？）

- 多使用一种“注册—通知—撤销注册”的形式
- Observer) 将自己注册到被观察对象 (Subject) 中，被观察对象将观察者存放在一个容器 (Container) 里
- 被观察对象发生了某种变化，从容器中得到所有注册过的观察者，将变化通知观察者



知乎 @柳树



Pub-Sub Pattern (image credit: MSDN blog)

知乎 @柳树

```
class Subject:
    """ 某主题 """
    def __init__(self):
        self.observers = []

    def add_observers(self, observer):
        self.observers.append(observer) # 这里利用了list的append方法
        return self

    def remove_observer(self, observer):
        self.observers.remove(observer)
        return self

    def notify(self, msg):
        for observer in self.observers:
            observer.update(msg)

xiaoming = Observer("xiaoming")
lihua = Observer("lihua")

rain = Subject() # 生成主题。可以有主题词?

# 添加订阅
rain.add_observers(xiaoming)
rain.add_observers(lihua)

rain.notify("下雨了!")

# 取消订阅
rain.remove_observer(lihua) # 可以主动订阅? 条件约束订阅?

xiaoming收到信息: 下雨了!
lihua收到信息: 下雨了!
```

# 命令模式：

- 命令模式是一种行为设计模式，他用于封装触发事件（完成任何一个操作）所包含的所有信息。初衷是用于适配复合交互指令的需要。
  - 优点
    - 把调用操作的类与执行该操作分离（解耦合，多了一个任务管理前台）
    - 结合队列可以更加灵活的构造新命令
    - 添加新命令不用改现有代码框架
    - 可以实现用命令模式定义层级回滚系统
  - 缺点
    - 体系结构复杂度高
    - 每个单独的命令都是一个类，增加了实现和维护的类的数量

## 模块 (Module) : 文件形式保存的功能模组

```
%%writefile fib.py

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Writing fib.py



# 模块有自己的名字空间

```
import fib as fb
```

```
fb.fib(4)
```

```
fb.fib2(3)
```

```
1 1 2 3
```

```
[1, 1, 2]
```

```
from fib import fib
```

```
fib(5)
```

```
1 1 2 3
```

# \_\_name\_\_属性

- 模块（.py文件）在创建之初会自动加载一些内建变量，\_\_name\_\_就是其中之一
- **if \_\_name\_\_ == '\_\_main\_\_':**  
保护模块私有的执行代码及定义被包含到其他模块中

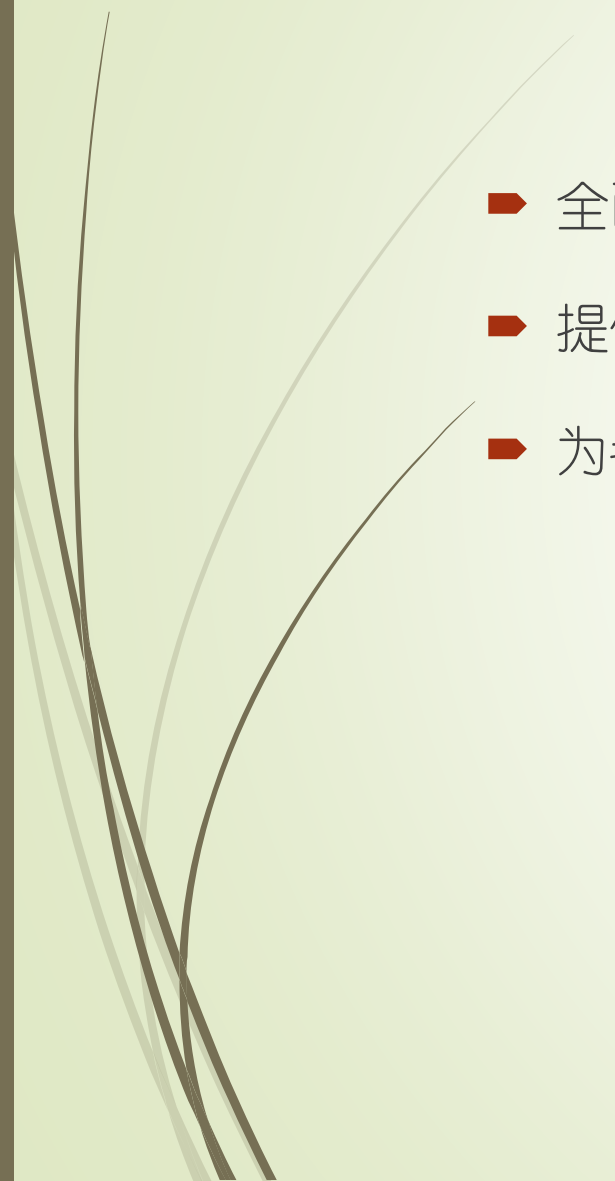


# Packages (包)

- 特定功能的模組文件集合构成的目录
- 可以通过\_\_init\_\_.py文件设定目录中文件的子目录结构及被加载的方案



# 总结一下Python的类实现方案


- 全面实现了面向对象的解决方案
  - 提供了更高的灵活性
  - 为参数化软件架构编程提供了好的支持
- 

# Python类编程架构与函数编程架构总结

- ‘类’定义了一个私有的数据和计算子环境。类又是可以通过继承、组合和派生（函数或工厂类）被生成出来的。类也可以是函数。
- 函数装饰器对应于类体系中的抽象类。
- 设计模式只是一些常见解决方案的经验模式，课程上主要用于对类编程进行深入学习，实现一些较复杂的类体系架构下的编程实现

# Python中的并发处理

- 多进程
- 进程通讯机制
- 多线程
- 线程锁与同步



并发：

- 1、程序要同时处理多个任务
- 2、经常需要等待资源
- 3、黑板系统逻辑

- 游戏：同时显示场景、播放声音、相应用户输入
- 网络服务器：同时与多个客户端建立连接、处理请求
- 多个功能化例程协作完成任务
- 方法：并行/并发编程
  - 并行：如多核CPU不同核上跑的两个进程。两个计算流在时间上重叠。
  - 并发：如单核CPU上跑的两个进程。两个计算流在时间上交替执行。  
给我们带来宏观上两个进程同时运行的假象。

# 计算机如何执行程序？

- 翻译成机器指令

- 把寄存器X的值  $\times 2$  放到寄存器Y里
- 把寄存器X的值存入内存地址Z处
- 跳转到.....
- .....

- 程序计数器

- 控制读到哪一句

- 通用目的寄存器、浮点寄存器、条件码寄存器

- 程序运行过程中的草稿纸
- 存某些变量的值、中间计算结果、栈指针、子程序返回值.....



# 计算机如何执行程序

## 虚拟内存划分

- 静态存取区（全局变量）
- 堆区（动态内存分配区，new创建）
- 栈区（子程序的局部变量）
- 只读代码段（存放机器代码）
- .....

进程用到的内存地址只是一个虚拟的地址

内核负责把这个虚拟的地址映射到物理内存

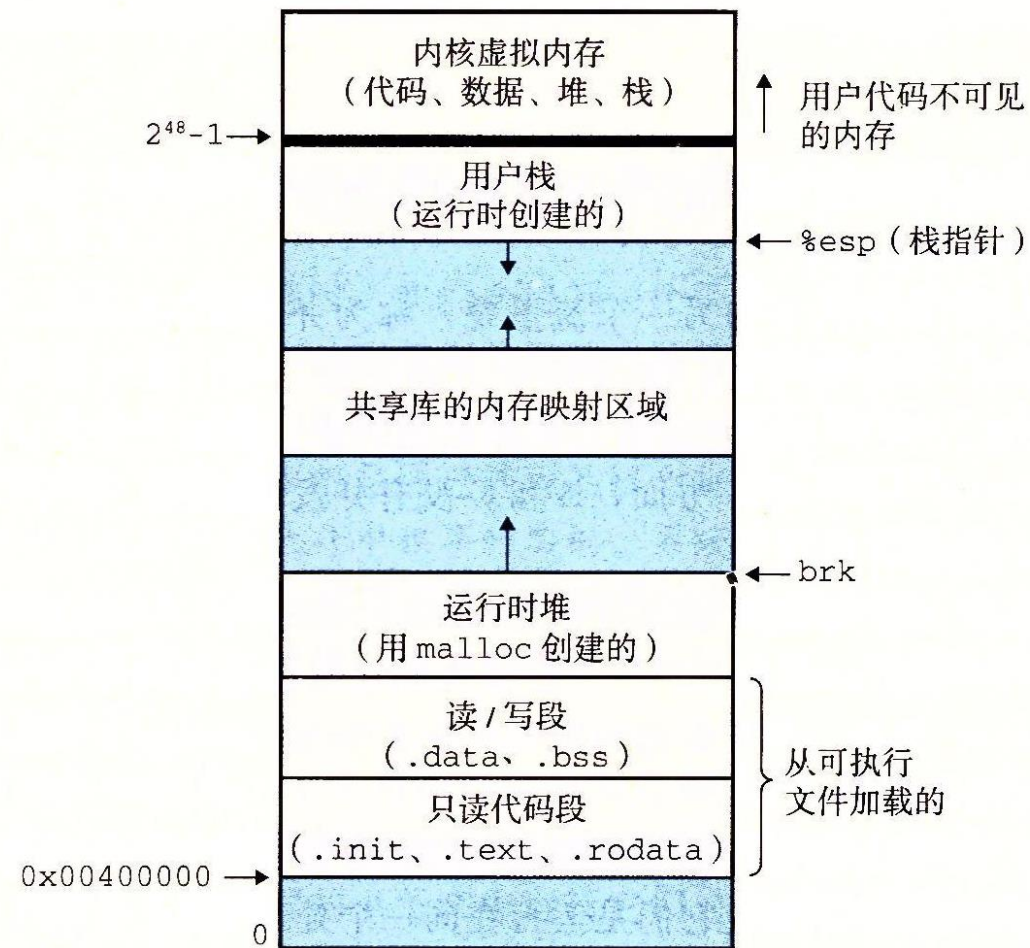


图 8-13 进程地址空间

# 计算机如何执行程序？

- 内核在负责资源的调度
- 每个进程有独立的逻辑控制流、私有的虚拟地址空间
- 维护一个进程需要
  - 程序计数器、通用目的寄存器、浮点寄存器、状态寄存器
  - 用户栈、内核栈、内核数据结构（用来映射虚拟地址的页表、当前打开文件信息的文件表）——进程的上下文

具体执行中是通过进程头与**CPU**的寄存器组配合保留-回复进程的私有运行环境

# 内核：调度和分配资源

- 内核负责进程的挂起和唤醒，
  - 在进程执行期间进行上下文的切换
- 进程（process）：资源分配的最小单元。其资源被内核保护起来。

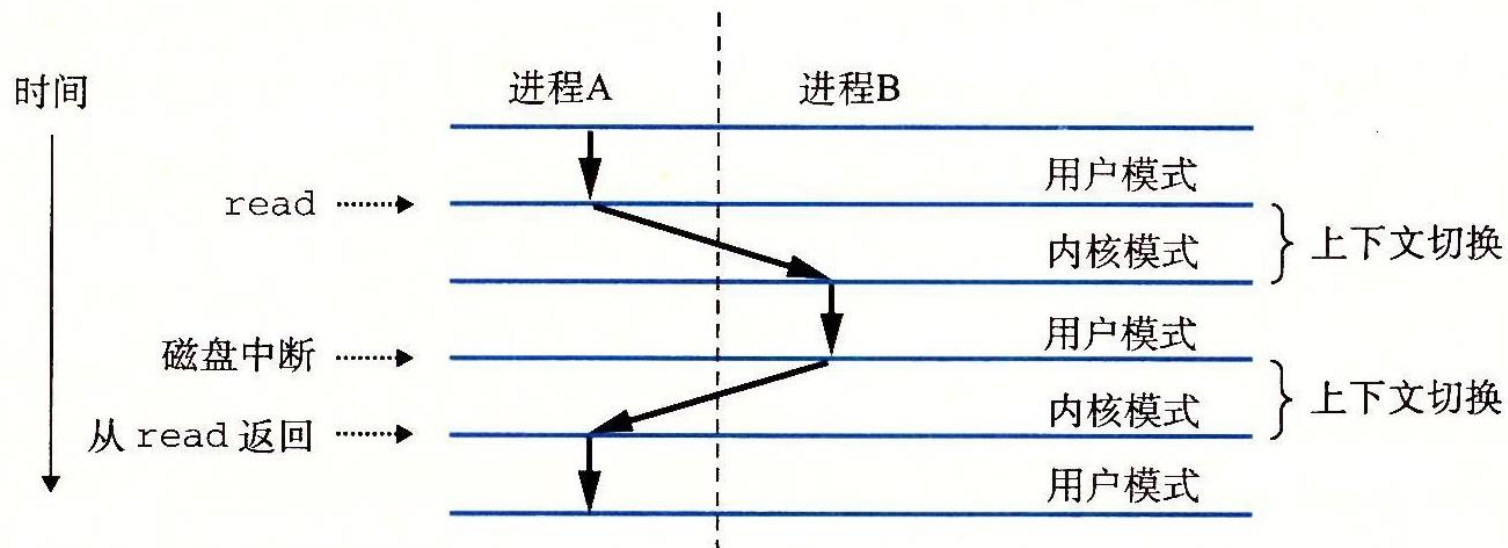


图 8-14 进程上下文切换的剖析

# 线程

- 由于进程之间不共享内存，进程的切换效率不太高
- 线程（thread）：调度执行的最小单元
  - 每个线程运行在单一进程中
  - 一个进程中可以有多个线程
- 线程上下文
  - 程序计数器、通用目的寄存器、浮点寄存器、条件码
  - 线程ID，栈，栈指针
- 线程间可以共享：进程的公有数据、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。因此可以方便的进行通讯。

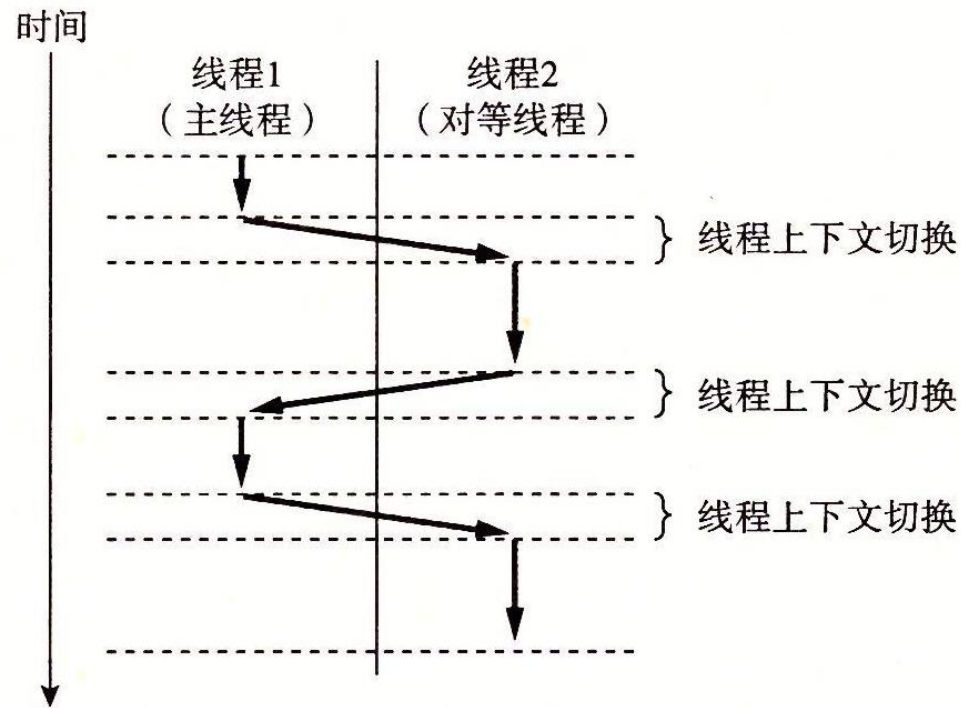


图 12-12 并发线程执行

# 线程锁机制

- 多个逻辑控制流同时读写共享的资源时，需要加入锁机制
- 例：100个线程，每个都对一个全局变量cnt操作cnt+=1
- 执行完后结果 $\leq 100$

## 加锁

- 需要读写某个共享变量时，读写前上锁，读写后释放锁
- 如果要读写的共享变量上了锁，等待锁释放后再上锁读写。



# 多进程、多线程编程

## ➤ 多进程

程序之间不共享内存，使用的资源独立

➤ 优点：一个进程挂了不影响别的进程

➤ 缺点：切换上下文效率低，通信和信息共享不太方便

## ➤ 多线程

➤ 优点：上下文切换效率比多进程高，线程之间信息共享和通信方便

➤ 缺点：一个线程挂了会使整个进程挂掉。操作全局变量需要锁机制不太方便。

不同的进程、线程，之间总是并发/并行的，

➤ 如果运行在多核CPU不同的核上，是并行的。

# 多进程模式：Process对象

p.start ()	启动进程,并调用子进程中的p.run ()
p.run ()	进程启动时运行的方法，正是他去调用target指定的函数
p.terminate ()	强制终止进程p，不会进行任何清理操作,如果p创建了子进程,该子进程就成了僵尸进程，使用该方法需要特别小心这种情况,如果p还保存了一个锁那么也将不会被释放，进而导致死锁
p.is_alive ()	如果p仍然运行，返回True
p.join ([timeout])	主线程等待p终止(强调：是主线程处于等待的状态，而p是处于运行状态)，timeout是可选的超时时间，需要强调的是，p.join只能join start开启的进程，而不能join run开启的进程

- Tips：无论是进程还是线程，都遵循：守护xx会等待主xx运行完毕后被销毁。
  - 对主进程来说，运行完毕指的是主进程代码运行完毕
  - 对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕

# Python 多进程编程

```
from multiprocessing import Process
import os
import time
```

```
def hello(i):
    print('son process id {} - for task {}'.format(os.getpid(), i))
    time.sleep(2)
```

```
if __name__ == '__main__':
    print('current father process {}'.format(os.getpid()))
    start = time.time()
    p1 = Process(target=hello, args=(1,))
    p2 = Process(target=hello, args=(2,))
    p1.start() # start son process
    p2.start()
    p1.join() # wait until it finishes
    p2.join()
    end = time.time()
    print("Totally take {} seconds".format((end - start)))
```

```
current father process 9976
son process id 8508 - for task 1
son process id 5836 - for task 2
Totally take 2.420004367828369 seconds
```

← 包装一个进程



# 多进程模式

## ➤ 进程池

- 在实际处理问题的过程中, 有时会有成千上万的任务需要被执行, 我们不可能创建那么多进程去完成任务. 首先创建进程需要时间, 销毁进程同样需要时间. 即便是真的创建好了这么多进程, 操作系统也不允许他们同时执行的, 这样反而影响了程序的效率.
- 进程池 -- 定义一个池子, 在里面放上固定数量的进程, 有任务要处理的时候就会拿一个池中的进程来处理任务, 等到处理完毕, 进程并不关闭而是放回进程池中继续等待任务. 如果需要有很多任务需要执行, 池中的进程数不够, 任务会就要等待进程执行完任务回到进程池, 拿到空闲的进程才能继续执行. 池中的进程数量是固定的, 那么同一时间最多有固定数量的进程在运行. 这样不会增加操作系统的调度难度, 还节省了开闭进程的时间, 也一定程度上能够实现并发效果.

## ➤ multiprocessing.Pool模块

numprocess	要创建的进程数, 如果省略, 将默认使用os.cpu_count () 的值
initializer	是每个工作进程启动时要执行的可调用对象, 默认为None
initargs	传给initializer的参数组

# 多进程模式

## ➤ multiprocessing.Pool

p.apply(func [ ,args [ ,kwargs] ] )

在一个池工作进程中执行func(\*args,\*\*kwargs), 然后返回结果 (同步调用)

p.apply\_async(func [ ,args [ ,kwargs] ] )

在一个池工作进程中执行func(\*args,\*\*kwargs), 然后返回结果 (异步调用)

p.close()

关闭进程池, 防止进一步操作. 如果所有操作持续挂起, 他们将在工作进程终止前完成

p.join()

等待所有工作进程退出. 此方法只能在close () 或 terminate () 之后调用

# Python 多进程编程

- 当进程很多，可以使用进程池管理

```
from multiprocessing import Process, Pool
import os
import time
```

```
def hello(i):
    print('son process id {} - for task {}'.format(os.getpid(), i))
    time.sleep(1)
```

```
if __name__ == '__main__':
    print('current father process {}'.format(os.getpid()))
    start = time.time()
    p = Pool(4) # 4 kernel CPU.
    for i in range(5):
        p.apply_async(hello, args=(i,))
    p.close() # no longer receive new process
    p.join() # wait until all processes in the pool finishes
    end = time.time()
    print("Totally take {} seconds".format((end - start)))
```

```
current father process 6316
son process id 7632 - for task 0
son process id 9044 - for task 1
son process id 8376 - for task 2
son process id 7820 - for task 3
son process id 7632 - for task 4
Totally take 2.717404842376709 seconds
```

close方法意味着不能再添加新的Process了。对Pool对象调用join ()方法，会暂停主进程，等待所有的子进程执行完

# 多进程模式

## multiprocessing.Pool

```
(base) C:\Users>python tester.py
Parent process 8440.
Waiting for all subprocesses done...
Run task 0 (14028)...
Run task 1 (1640)...
Run task 2 (6380)...
Run task 3 (14212)...
Task 0 runs 0.35 seconds.
Run task 4 (14028)...
Task 3 runs 1.41 seconds.
Task 2 runs 1.71 seconds.
Task 4 runs 1.53 seconds.
Task 1 runs 2.39 seconds.
All subprocesses done.
```

- `apply_async`是异步的，就是说子进程执行的同时，主进程继续向下执行。所以“Waiting for all subprocesses done...”先打印出来，“All subprocesses done.”最后打印。另，task 0, 1, 2, 3是立刻执行的，而task 4要等待前面某个task完成后才执行，这是因进程池中的进程数为4： `p=Pool(4)`

In [\*]:

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__ == '__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

```
Parent process 24148.
Waiting for all subprocesses done...
```

# 进程间通讯

- 使用多个进程时，通常使用消息传递来进行进程之间的通信，并避免必须使用任何同步原语（如锁）。对于传递消息，可以使用**Pipe()**（用于两个进程之间的连接）或队列**Queue**（允许多个生产者和消费者）。
- multiprocessing使用**queue.Empty**和 **queue.Full**异常
- **Queue** 有两个方法，**get** 和 **put** （可以设定阻塞或非阻塞）

# 多进程模式下的通讯

## ➤ Queue

➤ Queue 是一个近似 queue.Queue 的克隆

q.put (item)	将item放入队列中, 如果当前队列已满, 就会阻塞, 直到有数据从管道中取出
q.put_nowait (item)	将item放入队列中, 如果当前队列已满, 不会阻塞, 但是会报错
q.get ()	返回放入队列中的一项数据, 取出的数据将是先放进去的数据, 若当前队列为空, 就会阻塞, 直到放入数据进来
q.get_nowait ()	返回放入队列中的一项数据, 同样是取先放进队列中的数据, 若当前队列为空, 不会阻塞, 但是会报错



# 多进程模式下的主从通讯

## ➤ Pipe

- Pipe() 返回一个由管道连接的连接对象，默认情况下是双工（双向）

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()
```

```
PS C:\Data\pyrhonC4tmp> python .\test6.py
[42, None, 'hello']
```

Test6.py

# 常见应用模式：生产者-消费者

- 生产者产生数据
- 消费者读取数据
- 数据常常保存在一个消息队列中



# Python 多进程编程

- Python中提供了队列进行数据共享
  - Multiprocessing.Queue

```
Produce 0
Consume 0
Produce 1
Consume 1
Produce 2
Consume 2
Produce 3
Consume 3
Produce 4
Consume 4
Take 5.543811798095703 s.
```

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def producer(q):
    for value in range(5):
        print('Produce %d' % value)
        q.put(value)
        time.sleep(1)

# 读数据进程执行的代码:
def consumer(q):
    while True:
        value = q.get(True)
        print('Consume %d' % value)
        time.sleep(1)

if __name__ == '__main__':
    t0 = time.time()
    # 父进程创建Queue, 并传给各个子进程
    q = Queue()
    pw = Process(target=producer, args=(q,))
    pr = Process(target=consumer, args=(q,))
    # 启动子进程pw, 写入
    pw.start()
    # 启动子进程pr, 读取
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环, 无法等待其结束, 只能强行终止
    pr.terminate()
    print("Take %s s." % (time.time() - t0))
```

# Python 多线程编程

- 和多进程非常类似

```
import threading
import time

def hello(i):
    print('thread id: {} for task {}'.format(threading.current_thread().name, i))
    time.sleep(2)


if __name__ == '__main__':
    start = time.time()

    t1 = threading.Thread(target=hello, args=(1,))
    t2 = threading.Thread(target=hello, args=(2,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    end = time.time()
    print("Take {} s".format((end - start)))
```

```
thread id: Thread-1 for task 1
thread id: Thread-2 for task 2
Take 2.0140459537506104 s
```

# Python 多线程编程

- 在CPython中由于全局解释器锁（GIL）的存在
  - 全局解释器锁：一个进程任一时刻仅有一个线程在执行
  - 多核CPU并不能为它显著提高效率
- 可以考虑选择没有GIL的Python解释器（如JPython）



# 协程

- 比线程更轻量
- 把一个任务拆成“好几截”
- 一个线程可以拥有多个协程
- 其调度由程序（而非系统内核）控制

# Python迭代器

```
def fib(n):  
    index = 0  
    a = 0  
    b = 1  
  
    while index < n:  
        receive = yield b  
        print('`fib` receive %d' % receive)  
        a, b = b, a+b  
        index += 1
```

```
fib = fib(20)  
print('`fib` yield %d ' % fib.send(None)) # 效果等同于print(next(fib))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))
```

executed in 36ms, finished 10:32:16 2020-11-25

```
`fib` yield 1  
`fib` receive 1  
`fib` yield 1  
`fib` receive 1  
`fib` yield 2  
`fib` receive 1  
`fib` yield 3  
`fib` receive 1  
`fib` yield 5
```

# Python用迭代器实现协程

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

c = consumer()
produce(c)
```

被动端-lazy

主动端

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```



# 作业7点发布

