并发

基本概念

并发和并行区别

并行，parallel
同时做某些事，可以互不干扰的同一个时刻做几件事

并发，concurrency
也是同时做某些事，但是强调，一个时段内有事情要处理。

举例
乡村公路一条车道，半幅路面出现了坑，交警指挥交通。
众多车辆在这一时段要通过路面的事件，这就是并发。
交警指挥，车辆排队通过另外半幅路面，一个方向放行3分钟，停止该方向通行，换另一个方向放行。

高速公路的车道，双向4车道，所有车辆（数据）可以互不干扰的在自己的车道上奔跑（传输）。
在同一个时刻，每条车道上可能同时有车辆在跑，是同时发生的概念，这是并行。

并发的解决

“食堂打饭模型”

中午12点，开饭啦，大家都涌向食堂，这就是并发。如果人很多，就是高并发。

1、队列、缓冲区

假设只有一个窗口，陆续涌入食堂的人，排队打菜是比较好的方式。
所以，排队（队列）是一种天然解决并发的办法。
排队就是把人排成 **队列**，先进先出，解决了资源使用的问题。
排成的队列，其实就是一个缓冲地带，就是 **缓冲区**。

假设女生优先，那么这个窗口就得是两队，只要有女生来就可以先打饭，男生队列等着，女生队伍就是一个 **优先队列**。

例如queue模块的类Queue、LifoQueue、PriorityQueue。

2、争抢

只开一个窗口，有可能没有秩序，也就是谁挤进去就给谁打饭。
挤到窗口的人占据窗口，直到打到饭菜离开。
其他人继续争抢，会有一个人占据着窗口，可以视为锁定窗口，窗口就不能为其他人提供服务了。这是一种 **锁机制**。
谁抢到资源就上锁，排他性的锁，其他人只能等候。

争抢也是一种高并发解决方案，但是，这样不好，因为有可能有人很长时间抢不到

3、预处理

如果排长队的原因，是由于每个人打菜等候时间长，因为要吃的菜没有，需要现做，没打着饭不走开，锁定着窗口。
食堂可以提前统计大多数人最爱吃的菜品，将最爱吃的80%的热门菜，提前做好，保证供应，20%的冷门菜，现做。
这样大多数人，就算锁定窗口，也很快就是释放窗口了。

一种提前加载用户需要的数据的思路，**预处理** 思想，缓存常用。

4、并行

成百上千人同时来吃饭，一个队伍搞不定的，多开打饭窗口形成多个队列，如同开多个车道一样，并行打菜。

开窗口就得扩大食堂，得多雇人在每一个窗口提供服务，造成 成本上升。

日常可以通过购买更多服务器，或多开进程、线程实现并行处理，来解决并发问题。
注意这些都是 **水平扩展** 思想。

注：
如果线程在单CPU上处理，就不是并行了。
但是多数服务器都是多CPU的，服务的部署往往是多机的、分布式的，这都是并行处理。

5、提速

提高单个窗口的打饭速度，也是解决并发的方式。

打饭人员提高工作技能，或为单个窗口配备更多的服务人员，都是提速的办法。

提高单个CPU性能，或单个服务器安装更多的CPU。

这是一种 **垂直扩展** 思想。

6、消息中间件

上地、西二旗地铁 **站外 的九曲回肠的走廊**，缓冲人流，进去之后再多口安检进站。
常见的消息中间件有RabbitMQ、ActiveMQ（Apache）、RocketMQ（阿里Apache）、kafka（Apache）等。

当然还有其他手段解决并发问题，但是已经列举除了最常用的解决方案，一般来说不同的并发场景用不同的策略，而策略可能是多种方式的优化组合。

例如多开食堂（多地），也可以把食堂建设到宿舍生活区（就近），所以说，技术来源于生活。

进程和线程

在实现了线程的操作系统中，线程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。一个程序的执行实例就是一个进程。

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

进程和程序的关系
程序是源代码编译后的文件，而这些文件存放在磁盘上。当程序被操作系统加载到内存中，就是进程，进程中存放着指令和数据（资源），它也是线程的容器。

Linux进程有父进程、子进程，Windows的进程是平等关系。

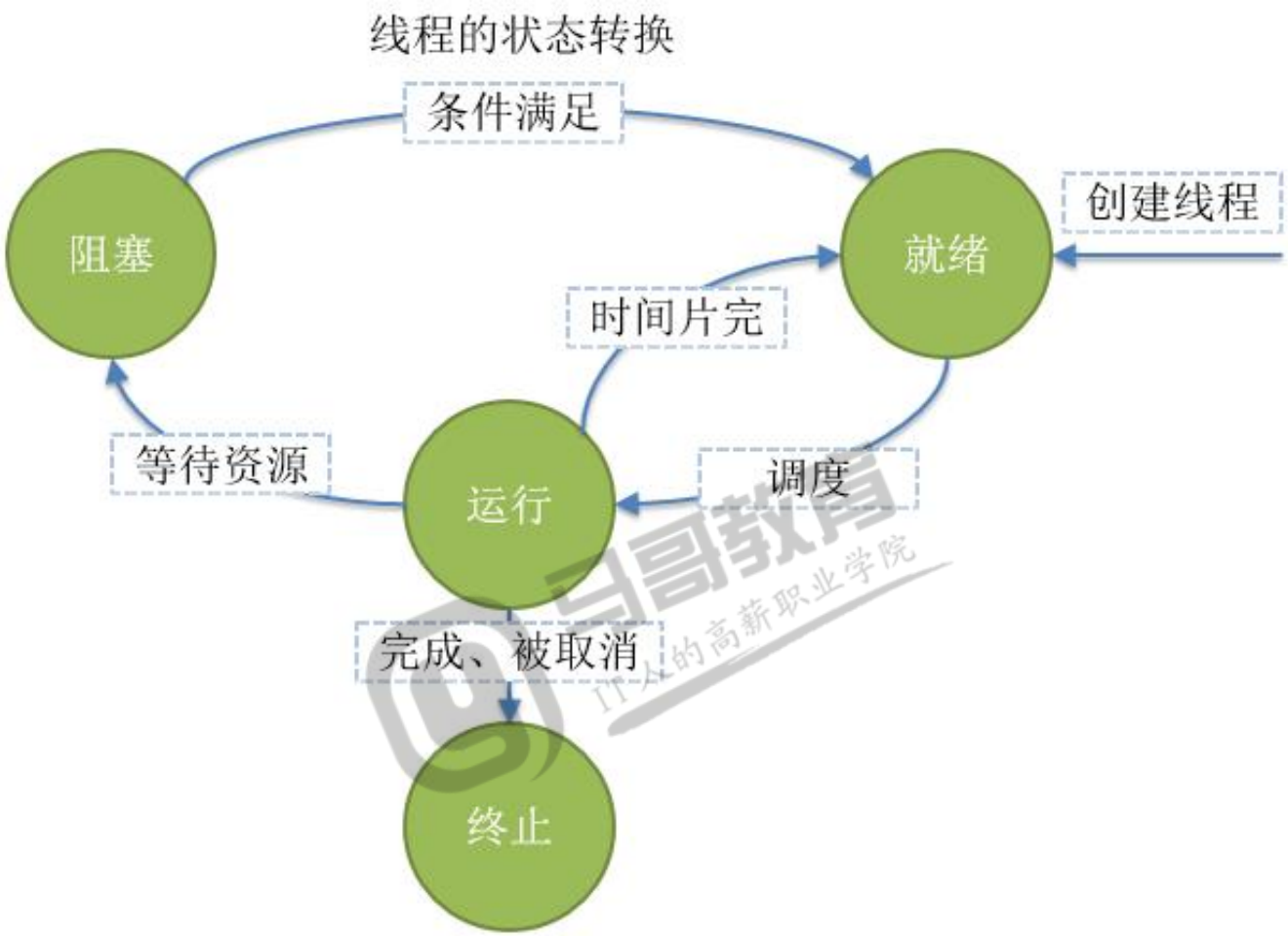
线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。
一个标准的线程由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。
在许多系统中，创建一个线程比创建一个进程快10-100倍。

进程、线程的理解
现代操作系统提出进程的概念，每一个进程都认为自己独占所有的计算机硬件资源。
进程就是独立的王国，进程间不可以随便的共享数据。
线程就是省份，同一个进程内的线程可以共享进程的资源，每一个线程拥有自己独立的堆栈。

线程的状态

状态	含义
就绪(Ready)	线程能够运行，但在等待被调度。可能线程刚刚创建启动，或刚刚从阻塞中恢复，或者被其他线程抢占

运行 (Running)	线程正在运行
阻塞 (Blocked)	线程等待外部事件发生而无法运行，如I/O操作
终止 (Terminated)	线程完成，或退出，或被取消



Python中的进程和线程

进程会启动一个解释器进程，线程共享一个解释器进程。

Python的线程开发

Python的线程开发使用标准库threading

Thread类

```
# 签名
def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None)
```

参数名	含义
target	线程调用的对象，就是目标函数
name	为线程起个名字
args	为目标函数传递实参，元组
kwargs	为目标函数关键字传参，字典

线程启动

```
import threading

# 最简单的线程程序
def worker():
    print("I'm working")
    print('Fineshed')

t = threading.Thread(target=worker, name='worker') # 线程对象
t.start() #  启动
```


通过threading.Thread创建一个线程对象，target是目标函数，name可以指定名称。
但是线程没有启动，需要调用start方法。
线程之所以执行函数，是因为线程中就是执行代码的，而最简单的封装就是函数，所以还是函数调用。
函数执行完，线程也就退出了。
那么，如果不让线程退出，或者让线程一直工作怎么办呢？

```
import threading
import time

def worker():
    while True:
        time.sleep(1)
        print("I'm working")
    print('Fineshed')
```

```
t = threading.Thread(target=worker, name='worker') # 线程对象
t.start() #  启动
```

线程退出

Python没有提供线程退出的方法，线程在下面情况时退出
1、线程函数内语句执行完毕
2、线程函数中抛出未处理的异常

```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            # raise RuntimeError(count)
            # return
            break
        time.sleep(1)
        print("I'm working")
        count += 1


t = threading.Thread(target=worker, name='worker') # 线程对象
t.start() #  启动

print('==End==')
```

Python的线程没有优先级、没有线程组的概念，也不能被销毁、停止、挂起，那也就没有恢复、中断了。

线程的传参

```
import threading
import time

def add(x, y):
    print('{} + {} = {}'.format(x, y, x + y, threading.current_thread().ident))
```

```
thread1 = threading.Thread(target=add, name='add', args=(4, 5))  # 线程对象
thread1.start()  # 启动
time.sleep(2)

thread2 = threading.Thread(target=add, name='add', args=(5,), kwargs={'y': 4})  # 线程对象
thread2.start()  # 启动
time.sleep(2)

thread3 = threading.Thread(target=add, name='add', kwargs={'x': 4, 'y': 5})  # 线程对象
thread3.start() # 启动
```

线程传参和函数传参没什么区别，本质上就是函数传参。

threading的属性和方法

名称	含义
current_thread()	返回当前线程对象
main_thread()	返回主线程对象
active_count()	当前处于alive状态的线程个数
enumerate()	返回所有活着的线程的列表，不包括已经终止的线程和未开始的线程
get_ident()	返回当前线程的ID，非0整数

active_count、enumerate方法返回的值还包括主线程。

```
import threading
import time

def showthreadinfo():
    print("currentthread = {}".format(threading.current_thread()))
    print("main thread = {}".format(threading.main_thread()))
    print("acive count = {}".format(threading.active_count()))

def worker():
    count = 0
    showthreadinfo()
```

```
    while True:
        if (count > 5):
            break
        time.sleep(1)
        count += 1
        print("I'm working")


t = threading.Thread(target=worker, name='worker') # 线程对象
showthreadinfo()
t.start() #  启动

print('==End==')
```

Thread实例的属性和方法

名称	含义
name	只是一个名字，只是个标识，名称可以重名。getName()、setName()获取、设置这个名词
ident	线程ID，它是非0整数。线程启动后才会有ID，否则为None。线程退出，此ID依旧可以访问。此ID可以重复使用
is_alive()	返回线程是否活着

注意：线程的name这是一个名称，可以重复；ID必须唯一，但可以在线程退出后再利用。

```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            break
        time.sleep(1)
        count += 1
        print(threading.current_thread().name)

t = threading.Thread(name='worker', target=worker)
```

```
print(t.ident)
t.start()

while True:
    time.sleep(1)
    if t.is_alive():
        print('{} {} alive'.format(t.name, t.ident))
    else:
        print('{} {} dead'.format(t.name, t.ident))
        t.start() # 可以吗?
```

名称	含义
start()	启动线程。每一个线程必须且只能执行该方法一次
run()	运行线程函数

为了演示，派生一个Thread的子类

start方法


```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            break
        time.sleep(1)
        count += 1
        print('worker running')

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~~~~~~~~~~~~')
        super().start()

    def run(self):
        print('run~~~~~~~~~~~~~~~')
        super().run()
```

```
t = MyThread(name='worker', target=worker)
t.start()
# 运行结果
start~~~~~~~~~~~~~~~
run~~~~~~~~~~~~~~~
worker running
```

run方法

```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            break
        time.sleep(1)
        count += 1
        print('worker running')

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~~~~~~~~~~~~')
        super().start()

    def run(self):
        print('run~~~~~~~~~~~~~~~')
        super().run()


t = MyThread(name='worker', target=worker)
# t.start()
t.run()

# 运行结果
run~~~~~~~~~~~~~~~
worker running
```

start()方法会调用run()方法，而run()方法可以运行函数。
这两个方法看似功能重复了，这么看来留一个方法就可以了。是这样吗？

start和run的区别

在线程函数中，增加打印线程的名字的语句，看看能看到什么信息。

```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            break
        time.sleep(1)
        count += 1
        print('worker running')
        print(threading.current_thread().name)

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~~~~~~~~~~~')
        super().start()

    def run(self):
        print('run~~~~~~~~~~~~~~~')
        super().run()


t = MyThread(name='worker', target=worker)
t.start()
# t.run() # 分别执行start或者run方法
```

使用start方法启动线程，启动了一个新的线程，名字叫做worker运行。但是使用run方法的，并没有启动新的线程，就是在主线程中调用了一个普通的函数而已。
因此，启动线程请使用start方法，才能启动多个线程。

多线程

顾名思义，多个线程，一个进程中如果有多个线程，就是多线程，实现一种并发。

```
import threading
import time

def worker():
    count = 0
    while True:
        if (count > 5):
            break
        time.sleep(0.5)
        count += 1
        print('worker running')
        print(threading.current_thread().name, threading.current_thread().ident)

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~~~~~~~~~~~')
        super().start()

    def run(self):
        print('run~~~~~~~~~~~~~~')
        super().run() # 看看父类再做什么


t1 = MyThread(name='worker1', target=worker)
t2 = MyThread(name='worker2', target=worker)

t1.start()
t2.start()
```

可以看到worker1和work2交替执行
改成run方法试试看

```
import threading
import time

def worker():
    count = 0
    while True:
```

```
        if (count > 5):
            break
        time.sleep(0.5)
        count += 1
        print('worker running')
        print(threading.current_thread().name, threading.current_thread().ident)

class MyThread(threading.Thread):
    def start(self):
        print('start~~~~~~~~~~~~~~~')
        super().start()

    def run(self):
        print('run~~~~~~~~~~~~~~~')
        super().run()


t1 = MyThread(name='worker1', target=worker)
t2 = MyThread(name='worker2', target=worker)

# t1.start()
# t2.start()
t1.run()
t2.run()
```


没有开新的线程，这就是普通函数调用，所以执行完t1.run()，然后执行t2.run()，这里就不是多线程。

当使用start方法启动线程后，进程内有多个活动的线程并行的工作，就是多线程。

一个进程中至少有一个线程，并作为程序的入口，这个线程就是**主线程**。一个进程至少有一个主线程。
其他线程称为工作线程。

线程安全

IPython中演示，python命令行、pycharm都不能演示出效果

```
import threading
```

```
def worker():
    for x in range(100):
        print("{} is running.".format(threading.current_thread().name))

for x in range(1, 5):
    name = "worker{}".format(x)
    t = threading.Thread(name=name, target=worker)
    t.start()
```

```
worker2 is running.worker4 is running.worker3 is running.
worker1 is running.


worker2 is running.worker4 is running.worker3 is running.
worker1 is running.


worker2 is running.worker4 is running.worker3 is running.


worker1 is running.worker2 is running.
worker3 is running.worker4 is running.
```

看代码，应该是一行行打印，但是很多字符串打在了一起，为什么？
说明，print函数被打断了，被线程切换打断了。print函数分两步，第一步打印字符串，第二步换行，就在这之间，发生了线程的切换。
这说明print函数是**线程不安全**的。

> 线程安全
> 线程执行一段代码，不会产生不确定的结果，那这段代码就是线程安全的

上例中，本以为print应该是打印文本之后紧跟着一个换行的，但是有时候确实好几个文本在一起，后面跟上换行，而且发生这种情况的时机不确定，所以，print函数不是线程安全函数。
如果是这样，多线程编程的时候，print输出日志，不能保证一个输出一定后面立即换行了，怎么办？

1、不让print打印换行

```
import threading

def worker():
```

```
    for x in range(100):
        print("{} is running.\n".format(threading.current_thread().name), end='')

for x in range(1, 5):
    name = "worker{}".format(x)
    t = threading.Thread(name=name, target=worker)
    t.start()
```

字符串是不可变的类型，它可以作为一个整体不可分割输出。end=''就不在让print输出换行了。

2、使用logging

标准库里面的logging模块，日志处理模块，线程安全的，生成环境代码都使用logging

```
import threading
import logging

def worker():
    for x in range(100):
        #print("{} is running.\n".format(threading.current_thread().name), end='')
        logging.warning("{} is running.".format(threading.current_thread().name))

for x in range(1, 5):
    name = "worker{}".format(x)
    t = threading.Thread(name=name, target=worker)
    t.start()
```

daemon线程和non-daemon线程

注意：这里的daemon不是Linux中的守护进程

进程靠线程执行代码，至少有一个主线程，其它线程是工作线程。
主线程是第一个启动的线程。
父线程：如果线程A中启动了一个线程B，A就是B的父线程。
子线程：B就是A的子线程。

Python中，构造线程的时候，可以设置daemon属性，这个属性必须在start方法前设置好。

```
# 源码Thread的__init__方法中
if daemon is not None:
    self._daemonic = daemon # 用户设定bool值
```

```
else:
    self._daemonic = current_thread().daemon
self._ident = None
```

线程daemon属性，如果设定就是用户的设置，否则就取当前线程的daemon值。
主线程是non-daemon线程，即daemon = False。

```
import time
import threading


def foo():
    time.sleep(5)
    for i in range(20):
        print(i)

# 主线程是non-daemon线程
t = threading.Thread(target=foo, daemon=False)
t.start()

print('Main Thread Exiting')
```

发现线程t依然执行，主线程已经执行完，但是一直等着线程t。
修改为 `t = threading.Thread(target=foo, daemon=True)` 试一试
程序立即结束了，根本没有等线程t。

名称	含义
daemon属性	表示线程是否是daemon线程，这个值必须在start()之前设置，否则引发RuntimeError异常
isDaemon()	是否是daemon线程
setDaemon	设置为daemon线程，必须在start方法之前设置

总结
线程具有一个daemon属性，可以显示设置为True或False，也可以不设置，则取默认值None。
如果不设置daemon，就取当前线程的daemon来设置它。
主线程是non-daemon线程，即daemon = False。
从主线程创建的所有线程的不设置daemon属性，则默认都是daemon = False，也就是non-daemon线程。
Python程序在没有活着的non-daemon线程运行时退出，也就是剩下的只能是daemon线程，主线程

才能退出，否则主线程就只能等待。

思考下面的程序的输出是什么？

```
import time
import threading

def bar():
    time.sleep(10)
    print('bar')

def foo():
    for i in range(20):
        print(i)
    t = threading.Thread(target=bar, daemon=False)
    t.start()

# 主线程是non-daemon线程
t = threading.Thread(target=foo, daemon=True)
t.start()

print('Main Thread Exiting')
```

上例中，会不会输出bar这个字符串，如果没有，如何修改才能打印出来？

```

time.sleep(2)
print('Main Thread Exiting')
```

再看一个例子，看看主线程合适结束daemon线程

```
import time
import threading

def foo(n):
    for i in range(n):
        print(i)
        time.sleep(1)

t1 = threading.Thread(target=foo, args=(10,), daemon=True) # 调换10和20看看效果
t1.start()
```

```
t2 = threading.Thread(target=foo, args=(20,),  daemon=False) #
t2.start()

time.sleep(2)
print('Main Thread Exiting')
```

上例说明，如果有non-daemon线程的时候，主线程退出时，也不会杀掉所有daemon线程，直到所有non-daemon线程全部结束，如果还有daemon线程，主线程需要退出，会结束所有daemon线程，退出。

join方法

先看一个简单的例子，看看效果

```
import time
import threading


def foo(n):
    for i in range(n):
        print(i)
        time.sleep(1)

t1 = threading.Thread(target=foo, args=(10,), daemon=True)
t1.start()
t1.join() # 设置join，取消join对比一下

print('Main Thread Exiting')
```


使用了join方法后，daemon线程执行完了，主线程才退出了。

join(timeout=None)，是线程的标准方法之一。
一个线程中调用另一个线程的join方法，调用者将被阻塞，直到被调用线程终止。
一个线程可以被join多次。
timeout参数指定调用者等待多久，没有设置超时，就一直等到被调用线程结束。
调用谁的join方法，就是join谁，就要等谁。

daemon线程应用场景

简单来说就是，本来并没有 daemon thread，为了简化程序员的工作，让他们不用去记录和管理那些后台线程，创造了一个 daemon thread 的概念。这个概念唯一的作用就是，当你把一个线程设置为 daemon，它会随主线程的退出而退出。
主要应用场景有：
1、后台任务。如发送心跳包、监控，这种场景最多。
2、主线程工作才有用的线程。如主线程中维护这公共的资源，主线程已经清理了，准备退出，而工作线程使用这些资源工作也没有意义了，一起退出最合适。
3、随时可以被终止的线程

如果主线程退出，想所有其它工作线程一起退出，就使用daemon=True来创建工作线程。
比如，开启一个线程定时判断WEB服务是否正常工作，主线程退出，工作线程也没有必须存在了，应该随着主线程退出一起退出。这种daemon线程一旦创建，就可以忘记它了，只用关系主线程什么时候退出就行了。
daemon线程，简化了程序员手动关闭线程的工作。

如果在non-daemon线程A中，对另一个daemon线程B使用了join方法，这个线程B设置成daemon就没有什么意义了，因为non-daemon线程A总是要等待B。
如果在一个daemon线程C中，对另一个daemon线程D使用了join方法，只能说明C要等待D，主线程退出，C和D不管是否结束，也不管它们谁等谁，都要被杀掉。

举例

```
import time
import threading

def bar():
    while True:
        time.sleep(1)
        print('bar')

def foo():
    print("t1's daemon = {}".format(threading.current_thread().isDaemon()))
    t2 = threading.Thread(target=bar)
    t2.start()
    print("t2's daemon = {}".format(t2.isDaemon()))

t1 = threading.Thread(target=foo, daemon=True)
t1.start()

time.sleep(3)
print('Main Thread Exiting')
```

上例，只要主线程要退出，2个工作线程都结束。
可以使用join，让线程结束不了，怎么做？

```
import time
import threading

def bar():
    while True:
        time.sleep(1)
        print('bar')

def foo():
    print("t1's daemon = {}".format(threading.current_thread().isDaemon()))
    t2 = threading.Thread(target=bar)
    t2.start()
    print("t2's daemon = {}".format(t2.isDaemon()))
    t2.join()

t1 = threading.Thread(target=foo, daemon=True)
t1.start()

t1.join()
time.sleep(3)
print('Main Thread Exiting')
```


threading.local类

```
import threading
import time

# 局部变量实现
def worker():
    x = 0
    for i in range(100):
        time.sleep(0.0001)
        x += 1
    print(threading.current_thread(), x)
```

```
for i in range(10):
    threading.Thread(target=worker).start()
```

上例使用多线程，每个线程完成不同的计算任务。x是局部变量。
能否改造成使用全局变量完成。

```
import threading
import time

class A:
    def __init__(self):
        self.x = 0

# 全局对象
global_data = A()

def worker():
    global_data.x = 0
    for i in range(100):
        time.sleep(0.0001)
        global_data.x += 1
    print(threading.current_thread(), global_data.x)

for i in range(10):
    threading.Thread(target=worker).start()
```

上例虽然使用了全局对象，但是线程之间互相干扰，导致了错误的结果。
能不能使用全局对象，还能保持每个线程使用不同的数据呢？

python提供 `threading.local` 类，将这个类实例化得到一个全局对象，但是不同的线程使用这个对象存储的数据其他线程看不见。

```
import threading
import time

# 全局对象
global_data = threading.local()

def worker():
    global_data.x = 0
    for i in range(100):
```

```
        time.sleep(0.0001)
        global_data.x += 1
    print(threading.current_thread(), global_data.x)

for i in range(10):
    threading.Thread(target=worker).start()
```

结果显示和使用局部变量的效果一样。

再看threading.local的例子

```
import threading

X = 'abc'
ctx = threading.local() # 注意这个对象所处的线程
ctx.x = 123

print(ctx, type(ctx), ctx.x)

def worker():
    print(X)
    print(ctx)
    print(ctx.x)
    print('working')

worker() # 普通函数调用
print()
threading.Thread(target=worker).start() # 另起一个线程
```

从运行结果来看，另起一个线程打印ctx.x出错了。

`AttributeError: '_thread._local' object has no attribute 'x'`

但是，ctx打印没有出错，说明看到ctx，但是ctx中的x看不到，这个x不能跨线程。

threading.local类构建了一个大字典，其元素是每一线程实例的地址为key和线程对象引用线程单独的字典的映射，如下：

`{ id(Thread) -> (ref(Thread), thread-local dict) }`

通过threading.local实例就可在不同的线程中，安全地使用线程独有的数据，做到了线程间数据隔离，如同本地变量一样安全。

定时器 Timer/延迟执行

threading.Timer继承自Thread，这个类用来定义多久执行一个函数。
class threading.Timer(interval, function, args=None, kwargs=None)
start方法执行之后，Timer对象会处于等待状态，等待了interval之后，开始执行function函数的。
如果在执行函数之前的等待阶段，使用了cancel方法，就会跳过执行函数结束。

```
import threading
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)d %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

def worker():
    logging.info('in worker')
    time.sleep(2)

t = threading.Timer(5, worker)
t.setName('w1')
t.start() # 启动线程
print(threading.enumerate())
t.cancel() # 取消，可以注释这一句看看如何定时执行
time.sleep(1)
print(threading.enumerate())
```

如果线程中worker函数已经开始执行，cancel就没有任何效果了。

总结
Timer是线程Thread的子类，就是线程类，具有线程的能力和特征。
它的实例是能够延时执行目标函数的线程，在真正执行目标函数之前，都可以cancel它

提前cancel

```
import threading
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)d %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

def worker():
    logging.info('in worker')
```

```
    time.sleep(2)

t = threading.Timer(5, worker)
t.setName('w1')
t.cancel() # 提前取消
t.start() # 启动线程
print(threading.enumerate())
time.sleep(8)
print(threading.enumerate())
```