并发

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
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。
在一段时间内，有这么多车要通过，这是并发。

并行不过是使用水平扩展方式解决并发的一种手段而已。

进程和线程

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）。一个程序的执行实例就是一个进程。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。

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。

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。

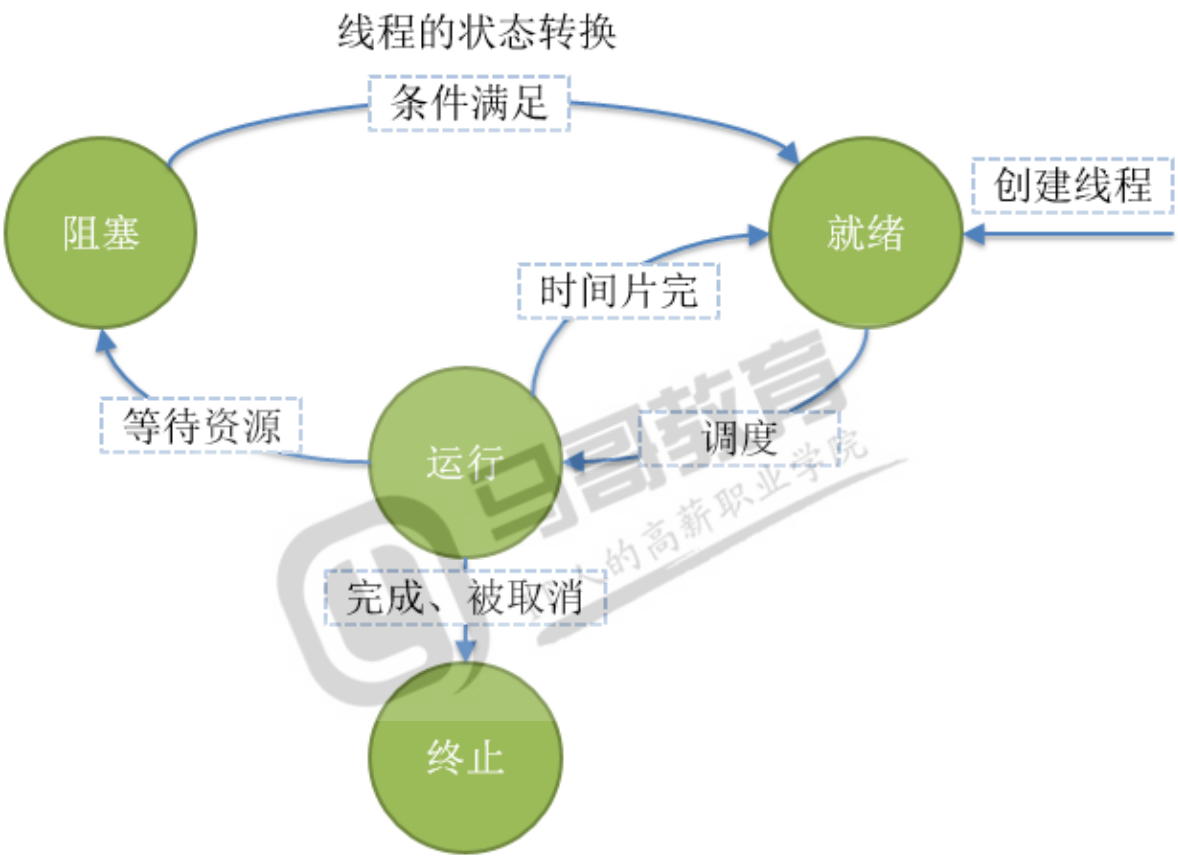
在许多系统中，创建一个线程比创建一个进程快10-100倍。

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

运行程序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。

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。

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，可以使用name为线程指定名称。
但是线程没有启动，需要调用start方法。

线程之所以执行函数，是因为线程中就是要执行代码的，而最简单的代码封装就是函数，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: # for i in range(10):
        time.sleep(0.5)
        print("I'm working")
    print('Fineshed')

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

print('=' * 30) # 注意看这行等号什么时候打印的?
```

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():
    for i in range(10):
        time.sleep(0.5)
        if i > 5:
            #break # 终止循环
            #return # 函数返回
            raise RuntimeError # 抛异常
        print('I am working')
    print('finished')

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

print('=' * 30)
```

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))

t1 = threading.Thread(target=add, name='add', args=(4, 5))
t1.start()
time.sleep(2)

t2 = threading.Thread(target=add, name='add', args=(6,), kwargs={'y':7})
t2.start()
time.sleep(2)

t3 = threading.Thread(target=add, name='add', kwargs={'x':8, 'y':9})
t3.start()
```

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 showtreadinfo():
    print('current thread = {}\nmain thread = {}\nactive count = {}'.format(
        threading.current_thread(), threading.main_thread(),
threading.active_count()
    ))

def worker():
    showtreadinfo()
    for i in range(5):
        time.sleep(1)
        print('i am working')
    print('finished')

t = threading.Thread(target=worker, name='worker') # 线程对象
showtreadinfo()
time.sleep(1)
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():
    for i in range(5):
```

```
        time.sleep(1)
        print('i am working')
    print('finished')

t = threading.Thread(target=worker, name='worker') # 线程对象
print(t.name, t.ident)
time.sleep(1)
t.start() # 启动

print('===end===')

while True:
    time.sleep(1)
    print('{} {} {}'.format(t.name, t.ident,
            'alive' if t.is_alive() else 'dead'))

    if not t.is_alive():
        print('{} restart'.format(t.name))
        t.start() # 线程重启？？
```

start和run方法

```
import threading
import time

def worker():
    for i in range(5):
        time.sleep(1)
        print('I am working')
    print('finished')

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

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

t = MyThread(target=worker, name='worker') # 线程对象
t.start() # 启动
t.start()
# t.run() # 或调用run方法
# t.run()
```

尝试start两次，或run两次都失败了，但是它们抛出的异常不一样。

但是单独运行start或者run都可以，是否可以不需要start方法了吗？在worker中打印线程名称、id。

```
import threading
import time

def worker():
    t = threading.current_thread()
    for i in range(5):
```

```
        time.sleep(1)
        print('I am working', t.name, t.ident)
    print('finished')

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

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

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

start方法才能启动操作系统线程，并运行run方法。run方法内部调用了目标函数。

多线程

顾名思义，多个线程，一个进程中如果有多个线程运行，就是多线程，实现一种并发。

```
import threading
import time
import sys

def worker(f=sys.stdout):
    t = threading.current_thread()
    for i in range(5):
        time.sleep(1)
        print('i am working', t.name, t.ident, file=f)
    print('finished', file=f)

t1 = threading.Thread(target=worker, name='worker1')
t2 = threading.Thread(target=worker, name='worker2', args=(sys.stderr,))
t1.start()
t2.start()
```

可以看到worker1和work2交替执行。

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

多线程执行一段代码，不会产生不确定的结果，那这段代码就是线程安全的。

多线程在运行过程中，由于共享同一进程中的数据，多线程并发使用同一个数据，那么数据就有可能被相互修改，从而导致某些时刻无法确定这个数据的值，最终随着多线程运行，运行结果不可预期，这就是线程不安全。

daemon线程

注：有人翻译成后台线程，也有人翻译成守护线程。

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
```

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。

```
class _MainThread(Thread):
    def __init__(self):
        Thread.__init__(self, name="MainThread", 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 Exits')
```

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，结果程序立即结束了，进程根本没有等daemon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

看一个例子，，看看主线程何时结束daemon线程

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
```

```
    print('{} working'.format(name))

# 主线程 是non-daemon线程
t1 = threading.Thread(target=worker, args=('t1', 5), daemon=True) # 调换5和10看看效果
t1.start()

t2 = threading.Thread(target=worker, args=('t2', 10), daemon=False)
t2.start()

print('Main Thread Exits')
```

上例说明，如果还有non-daemon线程在运行，进程不结束，进程也不会杀掉其它所有daemon线程。直到所有non-daemon线程全部运行结束（包括主线程），不管有没有daemon线程，程序退出。

总结

- 线程具有一个daemon属性，可以手动设置为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，程序退出，也就是除主线程之外剩下的只能都是daemon线程，主线程才能退出，否则主线程就只能等待

join方法

先看一个简单的例子，看看效果

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
    print('{} working'.format(name))

t1 = threading.Thread(target=worker, args=('t1', 3), daemon=True)
t1.start()
t1.join()# 设置join，取消join对比一下

print('Main Thread Exits')
```

使用了join方法后，当前线程阻塞了，daemon线程执行完了，主线程才退出了。

```
import time
import threading

def worker(name, timeout):
    time.sleep(timeout)
    print('{} working'.format(name))

t1 = threading.Thread(target=worker, args=('t1', 10), daemon=True)
t1.start()
t1.join(2)
```

```
print('~~~~~~~~~~~')
t1.join(2)
print('~~~~~~~~~~~')

print('Main Thread Exits')
```

`join(timeout=None)`

- join方法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，或阻塞超时
- 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

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，只用关心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。

threading.local类

```
import threading
import time
import logging

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

def worker():
    x = 0
    for i in range(100):
        time.sleep(0.0001)
        x += 1
    logging.info(x)

for i in range(10):
    threading.Thread(target=worker, name='t-{}'.format(i)).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，可以看出每一个线程的x是独立的，互不干扰的，为什么？

能否改造成使用全局变量完成？

```python
import threading
import time
import logging

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

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
    logging.info(global_data.x)

for i in range(10):
    threading.Thread(target=worker, name='t-{}'.format(i)).start()
```

上例虽然使用了全局对象，但是线程之间互相干扰，导致了不期望的结果。线程不安全。

能不能既使用全局对象，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。

```python
import threading
import time
import logging

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

# 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
    logging.info(global_data.x)

for i in range(10):
    threading.Thread(target=worker, name='t-{}'.format(i)).start()
```

结果显示和使用局部变量的效果一样。

再看threading.local的例子

```python
import threading
import time
```

```python
import logging

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

# 全局对象
X = 'abc'
global_data = threading.local()
global_data.x = 100
print(global_data, global_data.x)
print('~' * 30)
time.sleep(2)

def worker():
    logging.info(X)
    logging.info(global_data)
    logging.info(global_data.x)

worker() # 普通函数调用
print('=' * 30)
time.sleep(2)
threading.Thread(target=worker, name='worker').start() # 启动一个线程
```

从运行结果来看，另起一个线程打印 global_data.x 出错了。

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

但是，global_data打印没有出错，说明看到global_data，但是global_data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，存放所有线程相关的字典，定义如下：

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

每一线程实例的id为key，元组为value。

value中2部分为，线程对象引用，每个线程自己的字典。

> 本质
>
> 运行时，threading.local实例处在不同的线程中，就从大字典中找到当前线程相关键值对中的字典，覆盖threading.local实例的`__dict__`。
>
> 这样就可以在不同的线程中，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。