1. java创建线程对象有两种方式：

创建Thread子类的实例

以Runnable或Callable对象为target，创建Thread对象。

(Runnable或Callable对象的核心就是作为线程执行体的函数)

1. python创建线程对象同样有两种方式：

创建Thread子类的实例

以指定函数为target，创建Thread对象

# 线程创建

import threading

def action(max):

for i in range(max):

print(threading.current\_thread().getName() + " " + str(i))

for i in range(100):

print(threading.current\_thread().getName() + " " + str(i))

if i == 20:

t1 = threading.Thread(target=action, args=(100,))

t1.start()

t2 = threading.Thread(target=action, args=(100,))

t2.start()

print("main process done!")

# run和start

run线程执行体

start启动线程

//可以通过构造函数传递线程参数

import threading

class mythread(threading.Thread):

def \_\_init\_\_(self):

threading.Thread.\_\_init\_\_(self)

self.i = 0

def run(self):

while self.i < 100:

print(threading.current\_thread().getName() + " " + str(self.i))

self.i += 1

for i in range(100):

print(threading.current\_thread().getName() + " " + str(i))

if i == 20:

my1 = mythread()

my1.start()

my2 = mythread()

my2.start()

print('main done!')

import threading

def action(max):

for i in range(max):

print(threading.current\_thread().getName() + " " + str(i))

for i in range(100):

print(threading.current\_thread().getName() + " " + str(i))

if i == 20:

#不启动线程，直接执行线程方法

#t1 = threading.Thread(target=action, args=(100,))

#t1.start()

threading.Thread(target=action, args=(100,)).run()

#t2 = threading.Thread(target=action, args=(100,))

#t2.start()

threading.Thread(target=action, args=(100,)).run()

print("main process done!")

# 线程方法

tt = threading.Thread(target=action, args=(100,))

tt.is\_alive()

tt.join #主线程必须等待该线程执行结束后，才会向下执行

#后台线程

tt.daemon = True

#睡眠

import time

time.sleep(1)

time.sleep(0.001)

# 同步和互斥

acquire(blocking=True, timeout=-1)

release()

threading.Lock #基本锁，只能锁定一次，其余锁请求要等释放锁

threading.RLock #可重入锁, 比较常用的是RLock

--------------------------------------------------

import threading

import time

class Account:

def \_\_init\_\_(self, no, balance):

self.no = no

self.balance = balance

self.lock = threading.RLock()

def draw(account, draw\_amount):

account.lock.acquire()

try:

if account.balance >= draw\_amount:

print(threading.current\_thread().name + "pay ok:" + str(draw\_amount))

time.sleep(0.001)

account.balance -= draw\_amount

print("\n current amount: " + str(account.balance))

else:

print(threading.current\_thread().name + "pay failed:" + "lack of money")

finally:

account.lock.release()

acct = Account("1234567", 1000)

threading.Thread(name='A', target=draw, args=(acct, 800)).start()

threading.Thread(name='B', target=draw, args=(acct, 800)).start()

|  |  |
| --- | --- |
| 无锁保护  root@2004a:/home/test# python3 test.py  Apay ok:800  Bpay ok:800  current amount: 200  current amount: -600 | 有锁保护  root@2004a:/home/test# python3 test.py  Apay ok:800  current amount: 200  Bpay failed:lack of money |

# 线程通信Condition

acquire(timeout)/release()

wait(timeout) // wait会先解锁，再睡眠，被唤醒之后，再加锁

notify()

notify\_all()

/\* draw是花钱，deposit是存钱，flag=true表示账户有钱，可以消费，flag=false表示账户为空，可以存钱\*/

root@2004a:/home/test# cat test.py

import threading

import time

class Account:

def \_\_init\_\_(self, no, balance):

self.no = no

self.balance = balance

self.cond = threading.Condition()

self.flag = False

def getBalance(self):

return self.balance

def draw(self, draw\_amount):

self.cond.acquire()

try:

if not self.flag:

self.cond.wait()

else:

print(threading.current\_thread().name + "取钱:" + str(draw\_amount))

self.balance -= draw\_amount

print("账户金额为: " + str(self.balance)+"\n")

self.flag = False

self.cond.notify\_all()

finally:

self.cond.release()

def deposit(self, amount):

self.cond.acquire()

try:

if self.flag:

self.cond.wait()

else:

print(threading.current\_thread().name + "存款:" + str(amount))

self.balance += amount

print("账户金额为: " + str(self.balance)+"\n")

self.flag = True

self.cond.notify\_all()

finally:

self.cond.release()

def draw\_many(account, amount, max):

for i in range(max):

account.draw(amount)

def deposit\_many(account, amount, max):

for i in range(max):

account.deposit(amount)

acct = Account("1234567", 0)

threading.Thread(name="取款者", target=draw\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者A", target=deposit\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者B", target=deposit\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者C", target=deposit\_many, args = (acct, 800, 10)).start()

root@2004a:/home/test#

root@2004a:/home/test# python3 test.py

存款者A存款:800

账户金额为: 800

取款者取钱:800

账户金额为: 0

存款者A存款:800

账户金额为: 800

取款者取钱:800

账户金额为: 0

存款者A存款:800

账户金额为: 800

取款者取钱:800

账户金额为: 0

存款者A存款:800

账户金额为: 800

取款者取钱:800

账户金额为: 0

存款者A存款:800

账户金额为: 800

取款者取钱:800

账户金额为: 0

存款者B存款:800

账户金额为: 800

# 线程通信Queue

print(queue.\_\_doc\_\_)

help(queue)

## example1

import queue

qq = queue.Queue(2)

qq.put('111')

qq.put('222')

print("======test put\_nowait======")

try:

qq.put\_nowait('333')

except Exception as e:

print("error")

finally:

print("skip error")

print(qq.get())

print(qq.get())

#print("======test put\_nowait======")

#print(qq.get\_nowait())

try:

qq.get(timeout=1)

except Exception as e:

print("error")

finally:

print("skip error")

## example2

root@2004a:/home/test# cat test.py

import threading

import time

import queue

def product(qq):

arr=("python3", "Kotlin", "swift")

for i in range(3):

print(threading.current\_thread().name + "生产者准备生产元组元素!")

time.sleep(0.2)

qq.put(arr[i%3])

print(threading.current\_thread().name + "element generated!\n")

def consume(qq):

while True:

print(threading.current\_thread().name + "消费者准备消费元组元素!")

time.sleep(0.2)

t = qq.get()

print(threading.current\_thread().name + "element[%s] consumed!\n" % t)

qq = queue.Queue(maxsize=1)

threading.Thread(target=product, args=(qq,)).start()

threading.Thread(target=product, args=(qq,)).start()

threading.Thread(target=consume, args=(qq,)).start()

root@2004a:/home/test# python3 test.py

Thread-1生产者准备生产元组元素!

Thread-2生产者准备生产元组元素!

Thread-3消费者准备消费元组元素!

Thread-1element generated!

Thread-1生产者准备生产元组元素!

Thread-3element[python3] consumed!

Thread-3消费者准备消费元组元素!

Thread-2element generated!

Thread-2生产者准备生产元组元素!

Thread-3element[python3] consumed!

Thread-3消费者准备消费元组元素!

Thread-2element generated!

Thread-2生产者准备生产元组元素!

Thread-3element[Kotlin] consumed!

Thread-3消费者准备消费元组元素!

Thread-2element generated!

Thread-3element[swift] consumed!

Thread-3消费者准备消费元组元素!

Thread-1element generated!

Thread-1生产者准备生产元组元素!

Thread-3element[Kotlin] consumed!

Thread-3消费者准备消费元组元素!

Thread-1element generated!

Thread-3element[swift] consumed!

Thread-3消费者准备消费元组元素!

# 线程通信Event

#发次事件，多人接收

## example1

root@2004a:/home/test# cat test.py

import threading

import time

event=threading.Event()

def cal(name):

print('%s startup' % threading.currentThread().getName())

print('%s ready to calc' % name)

event.wait()

print('%s finish calc\n' % name)

threading.Thread(target=cal, args=('jia',)).start()

threading.Thread(target=cal, args=('yi',)).start()

time.sleep(2)

print('-------------------------------------')

print('main process send event')

event.set()

root@2004a:/home/test# python3 test.py

Thread-1 startup

jia ready to calc

Thread-2 startup

yi ready to calc

-------------------------------------

main process send event

jia finish calc

yi finish calc

root@2004a:/home/test#

## example2

root@2004a:/home/test# cat test.py

import threading

import time

class Account:

def \_\_init\_\_(self, no, balance):

self.no = no

self.balance = balance

self.lock = threading.Lock()

self.event = threading.Event()

def getBalance(self):

return self.balance

def draw(self, draw\_amount):

self.lock.acquire()

if self.event.is\_set():

print(threading.current\_thread().name + "取钱:" + str(draw\_amount))

self.balance -= draw\_amount

print("账户金额为: " + str(self.balance)+"\n")

self.event.clear()

self.lock.release()

self.event.wait()

else:

self.lock.release()

self.event.wait()

def deposit(self, amount):

self.lock.acquire()

if not self.event.is\_set():

print(threading.current\_thread().name + "存款:" + str(amount))

self.balance += amount

print("账户金额为: " + str(self.balance)+"\n")

self.event.set()

self.lock.release()

self.event.wait()

else:

self.lock.release()

self.event.wait()

def draw\_many(account, amount, max):

for i in range(max):

account.draw(amount)

def deposit\_many(account, amount, max):

for i in range(max):

account.deposit(amount)

acct = Account("1234567", 0)

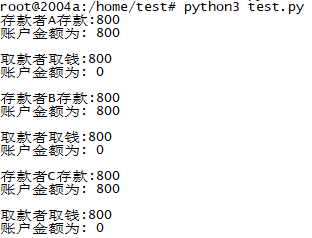
threading.Thread(name="取款者", target=draw\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者A", target=deposit\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者B", target=deposit\_many, args = (acct, 800, 10)).start()

threading.Thread(name="存款者C", target=deposit\_many, args = (acct, 800, 10)).start()

root@2004a:/home/test#



# 使用线程池

concurrent.futures模块有个类是Executor

Executor有两个子类：

Thread PoolExecutor用于创建线程池

ProcessPoolExecutor用于创建进程池

Executor有如下方法：

submit(fn, \*args, \*\*kwargs)

/\*\*\*\*map函数类似于全局函数map(func, \*iterables),只是上述函数会启动多个线程，以异步方式对iterables执行map处理\*\*\*\*/

map(func, \*iterables, timeout=None, chunksize=1)

shutdown(wait=True)

## example1

from concurrent.futures import ThreadPoolExecutor

import threading

import time

def action(max):

sum=0

for i in range(max):

#print(threading.current\_thread().name + ' ' + str(i))

sum += i

return sum

pool = ThreadPoolExecutor(max\_workers=2)

future1 = pool.submit(action, 50)

future2 = pool.submit(action, 100)

print('----test future.done----')

print(future1.done())

time.sleep(3)

print(future2.done())

print('-----test future.result------')

print(future1.result())

print(future2.result())

print('-----destroy pool--------')

pool.shutdown()

root@2004a:/home/test# python3 test.py

----test future.done----

True

True

-----test future.result------

1225

4950

-----destroy pool--------

root@2004a:/home/test#

## example2 – add\_done\_callback

from concurrent.futures import ThreadPoolExecutor

import threading

import time

def action(max):

sum=0

for i in range(max):

#print(threading.current\_thread().name + ' ' + str(i))

sum += i

return sum

def get\_result(future):

print(future.result())

#pool = ThreadPoolExecutor(max\_workers=2)

with ThreadPoolExecutor(max\_workers=2) as pool:

future1 = pool.submit(action, 50)

future2 = pool.submit(action, 100)

future1.add\_done\_callback(get\_result)

future2.add\_done\_callback(get\_result)

#print(future1.result())

#print(future2.result())

#pool.shutdown()

print('--------------')

root@2004a:/home/test#

root@2004a:/home/test# python3 test.py

1225

--------------

4950

root@2004a:/home/test#

## example3 - map

from concurrent.futures import ThreadPoolExecutor

import threading

import time

def action(max):

sum=0

for i in range(max):

#print(threading.current\_thread().name + ' ' + str(i))

time.sleep(0.01)

sum += i

return sum

def get\_result(future):

print(future.result())

#pool = ThreadPoolExecutor(max\_workers=4)

with ThreadPoolExecutor(max\_workers=4) as pool:

res = pool.map(action, (50, 100, 150))

print('---------------------')

for r in res:

print(r)

root@2004a:/home/test# python3 test.py

---------------------

1225

4950

11175

# 线程局部变量

通常建议：

如果多个线程之间需要共享资源，以实现线程通信，则使用同步机制

如果仅仅需要隔离多个线程之间的共享冲突，则可以使用线程局部变量

import threading

from concurrent.futures import ThreadPoolExecutor

mydata = threading.local()

def action(max):

for i in range(max):

try:

mydata.x += i

except:

mydata.x = i

print("name=%s mydata.x=%d" % (threading.current\_thread().name, mydata.x))

with ThreadPoolExecutor(max\_workers=2) as pool:

pool.submit(action, 10)

pool.submit(action, 10)

# timer

from threading import Timer

def hello():

print("hello, world")

t = Timer(5.0, hello)

t.start()

root@2004a:/home/test# cat test.py

from threading import Timer

import time

count=0

def print\_time():

print("current time: %s" % time.ctime())

global t,count

count += 1

if count < 10:

t = Timer(1, print\_time)

t.start()

t = Timer(1, print\_time)

t.start()

root@2004a:/home/test# python3 test.py

current time: Wed Feb 23 14:51:18 2022

current time: Wed Feb 23 14:51:19 2022

current time: Wed Feb 23 14:51:20 2022

current time: Wed Feb 23 14:51:21 2022

current time: Wed Feb 23 14:51:22 2022

current time: Wed Feb 23 14:51:23 2022

current time: Wed Feb 23 14:51:24 2022

current time: Wed Feb 23 14:51:25 2022

current time: Wed Feb 23 14:51:26 2022

current time: Wed Feb 23 14:51:27 2022

root@2004a:/home/test#

# sched

import sched,time

import threading

import os

s = sched.scheduler()

def print\_time(name="default"):

print("%s 的时间: %s" % (name, time.ctime()))

print("pid=%d\n" % os.getpid())

print('main process:', time.ctime())

s.enter(6, 1, print\_time)

#指定3秒后执行，优先级2

s.enter(3, 2, print\_time, argument=('位置参数',))

#指定3秒后执行，优先级1

s.enter(3, 1, print\_time, kwargs={'name':'关键字参数'})

s.run()

print('主线程', time.ctime)

# fork和multiprocessing.Process

fork创建进程不适合windows

multiprocessing.Process创建进程能跨平台运行

## fork

root@2004a:/home/test# cat test.py

import os

print("parent pid: %d" % os.getpid())

pid = os.fork()

if pid == 0:

print("I am fater: %d" % os.getpid())

else:

print("I am son: %d" % os.getpid())

print("process %d fininshed!" % os.getpid())

root@2004a:/home/test# python3 test.py

parent pid: 11688

I am son: 11688

process 11688 fininshed!

I am fater: 11689

process 11689 fininshed!

## multiprocessing

root@2004a:/home/test# cat test.py

import multiprocessing

import os

def action(max):

for i in range(max):

print("parent=%s son=%s: %d" % (os.getpid(), os.getppid(), i))

if \_\_name\_\_ == '\_\_main\_\_':

for i in range(10):

print("mainprocess=%d: %d" % (os.getpid(), i))

if i == 2:

mp1 = multiprocessing.Process(target=action, args=(10,))

mp1.start()

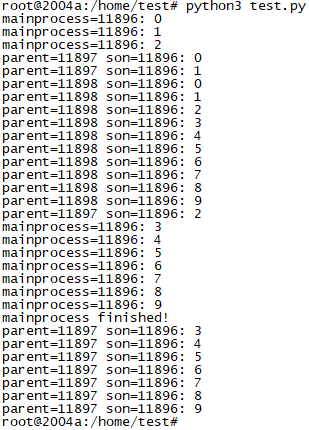
mp2 = multiprocessing.Process(target=action, args=(10,))

mp2.start()

mp2.join()

print('mainprocess finished!')

root@2004a:/home/test#



## 继承multiprocessing

#效果和上页的例子完全一样

import multiprocessing

import os

class MyProcess(multiprocessing.Process):

def \_\_init\_\_(self, max):

self.max =max

super().\_\_init\_\_()

def run(self):

for i in range(self.max):

print("parent=%s son=%s: %d" % (os.getpid(), os.getppid(), i))

if \_\_name\_\_ == '\_\_main\_\_':

for i in range(10):

print("mainprocess=%d: %d" % (os.getpid(), i))

if i == 2:

#mp1 = multiprocessing.Process(target=action, args=(10,))

mp1 = MyProcess(10)

mp1.start()

#mp2 = multiprocessing.Process(target=action, args=(10,))

mp2 = MyProcess(10)

mp2.start()

mp2.join()

print('mainprocess finished!')

## Context和启动进程的方式

根据平台的支持，python支持三种启动进程的方式：

spawn：父进程启动一个python解释器，子进程只能继承处理run()方法所必须的资源，那些文件描述符和handle都不会被继承。只在windows平台上有效。

fork：父进程启动一个python解释器，子进程会继承父进程的所有资源，因此子进程基本等效于父进程。只在unix平台上有效。

forkserver：启动服务器进程。只在unix平台上有效。

### set\_start\_method用于设置启动进程的方式

import multiprocessing

import os

def foo(q):

print("new process: %s" % os.getpid())

q.put('python3.8')

if \_\_name\_\_ == '\_\_main\_\_':

#设置使用fork方式启动进程

multiprocessing.set\_start\_method('fork')

q = multiprocessing.Queue()

mp = multiprocessing.Process(target=foo, args=(q,))

mp.start()

print(q.get())

mp.join()

### get\_context

#用Context对象来代替multiprocessing进程

import multiprocessing

import os

def foo(q):

print("new process: %d" % os.getpid())

q.put('python3.8')

if \_\_name\_\_ == '\_\_main\_\_':

#multiprocessing.set\_start\_method('fork')

#q = multiprocessing.Queue()

#设置使用fork方式启动进程，并获取Context对象

ctx = multiprocessing.get\_context('fork')

q = ctx.Queue()

#mp = multiprocessing.Process(target=foo, args=(q,))

mp = ctx.Process(target=foo, args=(q,))

mp.start()

print(q.get())

mp.join()

# 进程池

进程池具有如下方法：

apply：将函数交给进程池处理

apply\_async: apply方法的异步版本

map: 对itarable的每一个元素执行func函数

map\_async

imap #map的延迟版本

#功能类似于imap方法，但不能保证所生成的结果与原iterable中的元素顺序一致

imap\_unordered(func, iterable[, chunksize])

startmap

terminate

join

## apply\_async

import multiprocessing

import time

import os

def action(name='default'):

print('pid:%s is executing, and param=%s' % (os.getpid(), name))

time.sleep(3)

if \_\_name\_\_ == '\_\_main\_\_':

pool = multiprocessing.Pool(processes=4)

pool.apply\_async(action)

pool.apply\_async(action, args=('位置参数',))

pool.apply\_async(action, kwds={'name':'关键字参数'})

pool.close()

pool.join()

root@2004a:/home/test# python3 tets.py

pid:12144 is executing, and param=default

pid:12145 is executing, and param=位置参数

pid:12146 is executing, and param=关键字参数

root@2004a:/home/test#

## map

import multiprocessing

import time

import os

def action(max):

sum = 0

for i in range(max):

print('process:%s is executing:%d' % (os.getpid(), i))

sum += i

return sum

if \_\_name\_\_ == '\_\_main\_\_':

with multiprocessing.Pool(processes=4) as pool:

res = pool.map(action, (5, 10, 15))

print('-------------')

for i in res:

print(i)

# 进程通信

## Queue

import multiprocessing

def f(q):

print("%s begin to produce data..." % (multiprocessing.current\_process().pid))

q.put('python3')

if \_\_name\_\_ == '\_\_main\_\_':

q = multiprocessing.Queue()

p = multiprocessing.Process(target=f, args=(q,))

p.start()

print("%s begin to consume data..." % (multiprocessing.current\_process().pid))

print(q.get())

p.join()

## Pipe

root@2004a:/home/test# cat test.py

import multiprocessing

def f(conn):

print("process %s is sending data..." % multiprocessing.current\_process().pid)

conn.send('swift')

if \_\_name\_\_ == '\_\_main\_\_':

#conn1: father; conn2:child

conn1,conn2 = multiprocessing.Pipe()

p = multiprocessing.Process(target=f, args=(conn2,))

p.start()

print("process %s is receiving data..." % multiprocessing.current\_process().pid)

print(conn1.recv())

p.join()

root@2004a:/home/test# python3 test.py

process 12270 is receiving data...

process 12271 is sending data...

swift

root@2004a:/home/test#