相对于 thread 包，threading 包提供了更多的功能。该包的用法基本分成两步：

* 第一步是构造一个 threading.Thread 实例对象，这时该对象对应的线程就处于“新建”状态；
* 第二步是操作该对象，如调用 start() 来将该线程转换到“就绪”状态。

**创建线程实例对象**

我们可以创建基于现有的 threading.Thread 类的实例对象，主要需要提供入口函数和对应的参数。入口函数仍复用前面的函数，代码如下：

**import time  
def** thread\_entry(id):

cnt = 0

**while** cnt < 10:

**print**('Thread:(%d) Time:%s' % (id, time.ctime()))

time.sleep(1)

cnt = cnt + 1

下面来创建该实例，代码如下：

>>> import threading

>>> thread1 = threading.Thread(target=thread\_entry, name="thread 1", args=(1,))

>>> thread1.isAlive() # 线程是否在运行

False

>>> thread1.name # 线程名称

'thread 1'

在创建线程实例时可以设定入口函数、入口函数的参数、线程名等。

而在 Python 3 中多了一个标志，即 daemon，表示该线程是否为后台线程。  
  
在 Python 3 中，默认创建的线程不是 daemon 进程。在所有非 daemon 线程结束时，进程结束。进程结束时，所有 daemon 线程会强制退出。这说明 daemon 线程会自动随进程一起结束，而非 daemon 线程则会阻止进程的结束，或者说进程会等待所有非 daemon 线程的结束。  
  
下面来演示 daemon 属性的用法。创建一个 daemon 线程，在其完成任务之前，主线程退出。这时可以看到该 daemon 线程会自动退出。代码如下：

import sys, time

import threading # 引入线程库

**def** thread\_entry(): # 线程入口函数

left\_round = 10 # 一共循环10轮

**print**(' Child Thread: Start Running')

**while** left\_round > 0:

**print**('Child Thread: Running, left round = %d' % left\_round)

time.sleep(0.5)

left\_round = left\_round – 1

**print**("Child Thread Quit") # 线程退出

**def** start\_threads(): # 启动线程

thread1 = threading.Thread(target=thread\_entry, daemon=True)

thread1.start() # 启动线程，使之处于“就绪”状态

time.sleep(0.8)

**print**("Active Thread Number = %d" % threading.active\_count())

time.sleep(1.8)

**print**("Main Thread Quit") # 主线程退出

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

start\_threads()

运行情况如下：

$ python3 demo2.py  
Thread: Start Running  
Child Thread: Running, left round = 10  
Child Thread: Running, left round = 9  
Active Thread Number = 2                 # 现在有2个线程了  
Child Thread: Running, left round = 8  
Child Thread: Running, left round = 7  
Child Thread: Running, left round = 6  
Child Thread: Running, left round = 5  
Main Thread Quit                         # 主线程结束，导致进程结束

可以看到主线程退出后进程就退出了。如果将 daemon 参数设为 False，则该新创建的线程就不是 daemon 线程了，这时即使主线程退出，进程也不会退出，直到所有的非 daemon 线程都退出。  
  
将上例中的 start\_threads() 函数修改如下，其他代码没有任何改变：

**def** start\_threads(): # 创建非daemon线程

thread1 = threading.Thread(target=thread\_entry, daemon=False)

thread1.start()

time.sleep(0.8)

**print**("Active Thread Number = %d" % threading.active\_count())

time.sleep(1.8)

**print**("Main Thread Quit")

再次运行该程序，结果如下：

$ python3 demo3.py  
Child Thread: Start Running  
Child Thread: Running, left round = 10  
Child Thread: Running, left round = 9  
Active Thread Number = 2  
Child Thread: Running, left round = 8  
Child Thread: Running, left round = 7  
Child Thread: Running, left round = 6  
Child Thread: Running, left round = 5  
Main Thread Quit                  # 主线程退出，但是进程没有退出  
Child Thread: Running, left round = 4  
Child Thread: Running, left round = 3  
Child Thread: Running, left round = 2  
Child Thread: Running, left round = 1  
Child Thread Quit                 # 所有的非daemon线程结束，进程结束

在 Python 2 ，创建线程时是不能指定 daemon 属性的，但是可以在创建后修改属性，方式如下：

Thread实例对象.set setDaemon(False)

在 Python 2 中，新创建的非主线程默认都是 daemon 线程，即会随着进程退出而强制退出。下面是 Python 2 中的一个例子，在该例子中，创建线程后将其设置为非 daemon 线程，这样进程便会一直等待该线程退出才退出，不会强制该线程退出。主要修改在 start\_threads() 部分，修改后的内容如下：

**def** start\_threads():

thread1 = threading.Thread(target=thread\_entry)

thread1.setDaemon(False) # 修改为非daemon线程

thread1.start()

time.sleep(0.8)

**print**("Active Thread Number = %d" % threading.active\_count())

time.sleep(1.8)

**print**("Main Thread Quit")

运行情况如下：

$ python demo4.py  
Child Thread: Start Running  
Child Thread: Running, left round = 10  
Child Thread: Running, left round = 9  
Active Thread Number = 2  
Child Thread: Running, left round = 8  
Child Thread: Running, left round = 7  
Child Thread: Running, left round = 6  
Child Thread: Running, left round = 5  
Main Thread Quit                        # 主线程退出，但是进程还没有退出  
Child Thread: Running, left round = 4   # 新建的线程仍然在执行  
Child Thread: Running, left round = 3  
Child Thread: Running, left round = 2  
Child Thread: Running, left round = 1  
Child Thread Quit                       # 新建的线程退出，进程退出

**派生自己的线程类**

上面是通过创建 threading.Thread 的实例对象来创建线程的。其实也可以创建自己的线程类，然后使用自己的线程类来创建实例对象以达到创建线程的目的。当然自己创建的线程类要派生自 threading.Thread。

import sys, time

import threading # 引入线程库

**class** CustomThread(threading.Thread): # 派生自己的线程类

**def** \_\_init\_\_(self, thread\_name):

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

self.thread\_name = thread\_name

**def** run(self): # 线程的主函数

left\_round = 10

**print**('Child Thread: Start Running')

**while** left\_round > 0:

**print**('Child Thread: Running, left round = %d' % left\_round)

time.sleep(0.5)

left\_round = left\_round - 1

**print**("Child Thread Quit")

**def** start\_threads(): # 启动线程

thread1 = CustomThread('thread 1') # 创建线程对象

thread1.setDaemon(False) # 设置为非daemon线程

thread1.start() # 启动线程

time.sleep(0.8)

**print**("Active Thread Number = %d" % threading.active\_count())

time.sleep(1.8)

**print**("Main Thread Quit") # 主线程退出

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

start\_threads()

运行结果如下：

$ python demo6.py  
Child Thread: Start Running  
Child Thread: Running, left round = 10  
Child Thread: Running, left round = 9  
Active Thread Number = 2                   # 2个线程  
Child Thread: Running, left round = 8  
Child Thread: Running, left round = 7  
Child Thread: Running, left round = 6  
Child Thread: Running, left round = 5  
Main Thread Quit                           # 主线程退出，但是进程还没有退出  
Child Thread: Running, left round = 4  
Child Thread: Running, left round = 3  
Child Thread: Running, left round = 2  
Child Thread: Running, left round = 1  
Child Thread Quit                          # 子线程退出，进程退出

这里需要注意的是：

* 在构造函数中一定要调用父类的构造函数，然后再进行其他的初始化操作。
* run() 就是线程的入口函数，在外部调用实例对象的 start() 接口时就会运行该函数。该函数不能有除 self 之外的其他参数，但是可以通过构造函数传入或者通过设置来传输。

**配置线程**

在启动线程之前，线程处于“新建”状态，这时可以配置线程的信息。  
  
1) 修改是否为 daemon 线程：这个仅在 Python 3 中才有效。我们可以操作线程对象的 daemon 属性，推荐的用法是使用线程对象的 setDaemon() 和 isDaemon() 接口函数。下面演示了这些用法。

>>> import threading # 引入线程库

>>> import time

>>> def thread\_entry(): # 定义入口函数

... time.sleep(30)

... # 入口函数定义结束

>>> thread1 = threading.Thread(target=thread\_entry) # 创建线程

>>> thread1.daemon # 查看该线程是否是daemon线程

False

>>> thread1.daemon = True # 修改daemon属性

>>> thread1.daemon # 查看该线程是否是daemon线程

True

>>> thread1.isDaemon() # 查看该线程是否是daemon线程

True

>>> thread1.setDaemon(False) # 修改daemon属性

>>> thread1.isDaemon() # 查看该线程是否是daemon线程

False

2) 修改线程的名称：每个线程都有一个名字，即使我们没有指定，系统也会自动为其分配一个名字。可以使用线程对象的 name 属性来修改或者获得线程的名字。和daemon属性一样，推荐使用线程对象的 setName() 和 getName() 接口函数来操作线程的名字。下面演示了这些用法。

>>> import threading # 引入threading库

>>> import time # 定义线程入口函数

>>> def thread\_entry():

... time.sleep(30)

... # 入口函数定义结束

>>> thread1 = threading.Thread(target=thread\_entry) # 创建线程

>>> thread1.name # 得到线程的名字

'Thread-1' # 这是系统分配的名字

>>> thread1.name = "new-name" # 修改线程的名字

>>> thread1.name # 得到线程的名字

'new-name'

>>> thread1.getName() # 得到线程的名字

'new-name'

>>> thread1.setName("abc") # 修改线程的名字

>>> thread1.getName() # 得到线程的名字

'abc'

**启动线程**

在配置好线程后，可以启动该线程。这时线程的状态由“新建”变为“就绪”。就绪状态的线程并没有在运行，只是说可以运行了。只有在操作系统可以为其分配相关的处理器资源时，才会将其调度到某个处理器上运行。  
  
threading.Thread 实例对象有一个接口函数 start()，通过调用该函数可以启动该线程。

>>> def thread\_entry(): # 定义入口函数

... time.sleep(30) # 等待30秒

... print("Child Thread Quit")

... # 入口函数结束

>>> thread1 = threading.Thread(target=thread\_entry) # 创建线程

>>> thread1

<Thread(Thread-5, initial)> # 线程状态为initial

>>> thread1.start() # 开始运行，进入“就绪”状态

>>> thread1

<Thread(Thread-5, started 123145558048768)>

start() 函数不能被调用多次，否则会抛出异常。下面的例子演示了多次调用 start() 的情形。

>>> thread1.start() # 再次启动线程

Traceback (most recent call last): # 抛出RuntimeError异常

File "<stdin>", line 1, in <module>

File "/usr/local/Cellar/python@

2/2.7.15\_1/Frameworks/Python.framework/Versions

/2.7/lib/python2.7/threading.py", line 730, in start

raise RuntimeError("threads can only be started once")

RuntimeError: threads can only be started once

**停止线程**

在 thread 包和 threading 包中都没有提供强制线程退出的接口函数，但可以通过给线程发送信号的方式来强制某个线程退出。

import threading # 引入线程库

import inspect, sys, time

import ctypes

**def** async\_raise\_exception(thread\_id, exctype): # 给线程发送信号

tid = ctypes.c\_long(thread\_id) # 得到线程的id

**if** **not** inspect.isclass(exctype):

exctype = type(exctype)

res = ctypes.pythonapi.PyThreadState\_SetAsyncExc(tid,

ctypes.py\_object(exctype))

**if** res == 0:

**raise** ValueError("invalid thread id")

**elif** res != 1:

ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, None)

**raise** SystemError("PyThreadState\_SetAsyncExc failed")

**def** thread\_entry(): # 线程入口函数

**print**('Child Thread: Start Running')

**while** True:

**try** :

time.sleep(0.2)

**except** SystemExit: # 接收到信号

**print**("Got an SystemExit Exception, Quit")

**break** # 退出线程

**print**("Child Thread Quit")

**def** start\_threads(): # 启动线程

thread1 = threading.Thread(target=thread\_entry)

thread1.setDaemon(False)

thread1.start()

time.sleep(2)

**print**("Active Thread Number = %d" % threading.active\_count())

async\_raise\_exception(thread1.ident, SystemExit)

time.sleep(1)

**print**("Active Thread Number = %d" % threading.active\_count())

**print**("Main Thread Quit")

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

start\_threads()

运行结果如下：

$ python forceQuit1.py  
Child Thread: Start Running  
Active Thread Number = 2  
Got an SystemExit Exception, Quit  
Child Thread Quit  
Active Thread Number = 1  
Main Thread Quit

**等待线程结束**

有时父线程需要等待所有的子线程完成任务，这时可以使用 join() 接口函数。该接口函数会一直等待，直到对象的线程退出。

import sys, time # 引入time库

import threading

**def** thread\_entry(): # 定义入口函数

left\_round = 10

**print**('Child Thread: Start Running')

**while** left\_round > 0:

**print**('Child Thread: Running, left round = %d' % left\_round)

time.sleep(0.2)

left\_round = left\_round - 1

**print**("Child Thread Quit")

**return** 88

**def** start\_threads(): # 启动线程

thread1 = threading.Thread(target=thread\_entry)

thread1.setDaemon(False)

thread1.start()

thread1.join() # 等待子线程结束

**print**("Main Thread Quit") # 主线程退出

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

start\_threads()

运行结果如下：

$ python joinDemo1.py  
Child Thread: Start Running  
Child Thread: Running, left round = 10  
Child Thread: Running, left round = 9  
Child Thread: Running, left round = 8  
Child Thread: Running, left round = 7  
Child Thread: Running, left round = 6  
Child Thread: Running, left round = 5  
Child Thread: Running, left round = 4  
Child Thread: Running, left round = 3  
Child Thread: Running, left round = 2  
Child Thread: Running, left round = 1  
Child Thread Quit  
Main Thread Quit

join() 函数还可以带一个时间参数，表示最多等待多少秒，如果等待超时，该函数返回。但不论是因为超时返回还是因为线程退出，join() 函数的返回值都是 None。为了分辨返回的原因，需要在 join() 之后调用 isAlive() 来判断等待的线程是否真的退出了。不能 join() 自己，否则会抛出异常。不能 join() 处于“新建”状态的线程，否则也会抛出异常。

**其他接口函数**

下面介绍 threading 包中其他常用的接口函数。  
  
1) 得到当前进程内运行状态线程的个数—— threading.active\_count()：这是一个类函数，不需要使用线程实例对象，也没有任何输入参数。

>>> def thread\_entry(): # 入口函数

... time.sleep(30)

... print("Child Thread Quit")

... # 入口函数结束

>>> thread1 = threading.Thread(target=thread\_entry) # 创建线程

>>> threading.active\_count() # 有效线程个数

1

>>> thread1.start() # 启动线程

>>> threading.active\_count() # 有效线程个数

2

2) 当前线程——threading.current\_thread()：这也是一个类函数，不需要使用线程实例对象，也不需要任何输入参数，返回的是一个线程实例对象。在不同的线程内调用会得到不同的实例对象。

>>> thread\_obj1 = threading.current\_thread() # 得到当前的线程

>>> type(thread\_obj1)

<class 'threading.\_MainThread'>

>>> isinstance(thread\_obj1, threading.Thread) # 是否为线程对象

True

>>> thread\_obj1 # 查看线程对象

<\_MainThread(MainThread, started 140735876817792)>

>>> thread\_obj1.name # 线程名

'MainThread'

>>> thread\_obj1.\_\_class\_\_ # 线程的类型

<class 'threading.\_MainThread'>

3) 设置线程栈大小——threading.stack\_size(size)：单位是字节。该函数对于所有后来新建的线程有效。目前阶段不能对某个线程设定栈的大小，只能设定系统参数。size 的值可以为 0 或者大于 32768（表示 32KB）的正整数，但最好不要设置为任意值，建议设置为 4KB 的整数倍，如 8192（8KB）；如果设为 0 表示取系统默认值。

该函数仅在 Windows 和支持 POSIX 的系统上有效。

4) 得到线程栈的大小——threading.stack\_size()：该函数返回的是一个整数，0 表示系统默认值，否则表示栈的字节数。

>>> threading.stack\_size() # 读取栈的大小

0 # 0表示默认值

>>> threading.stack\_size(1024\*1024\*2) # 设定栈的大小为2M

0

>>> threading.stack\_size() # 查看目前栈的大小

2097152

该函数仅在 Windows 和支持 POSIX 的系统上有效。

5) 得到线程 ID：该值可以从线程对象的 ident 属性得到。如果线程处于“新建”状态，那么该属性值为 0。该 ID 值是循环使用的，如果某个线程退出，那么其原来使用的 ID 可能会分配给新的线程使用。

>>> a = threading.current\_thread() # 得到当前线程的对象

>>> a # 查看线程对象

<\_MainThread(MainThread, started 140735876817792)>

>>> a.ident # 查看ID

140735876817792

6) 得到主线程——main\_thread()：该函数仅在 Python 3.4 以及后面的版本中可用，在 Python 2 和早期的 Python 3 中是不存在的，其返回的是主线程实例对象。

>>> import threading

>>> threading.main\_thread()

<\_MainThread(MainThread, started 140735876817792)>

7) 超时时间TIMEOUT\_MAX：在调用 Lock.acquire()、RLock.acquire() 时等待的默认时间，单位为秒。如果超过这个时间，调用返回错误码，不再继续等待。这个属性可读可写。

>>> threading.TIMEOUT\_MAX # 得到超时时间，单位为秒

9223372036.0

>>> threading.TIMEOUT\_MAX = 9223372038

>>> threading.TIMEOUT\_MAX

9223372038

8) 得到线程列表——enumerate()：该函数返回一个列表，成员是 threading.Thread 的实例对象。

>>> import threading # 引入线程库

>>> threading.enumerate() # 得到遍历线程的对象

[<\_MainThread(MainThread, started 140735876817792)>]