12.1 启动与停止线程¶

问题¶

你要为需要并发执行的代码创建/销毁线程

解决方案¶

`threading` 库可以在单独的线程中执行任何的在 Python 中可以调用的对象。你可以创建一个 `Thread` 对象并将你要执行的对象以 target 参数的形式提供给该对象。 下面是一个简单的例子：

```
# Code to execute in an independent thread
import time
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create and launch a thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

当你创建好一个线程对象后，该对象并不会立即执行，除非你调用它的 `start()` 方法（当你调用 `start()` 方法时，它会调用你传递进来的函数，并把你传递进来的参数传递给该函数）。Python中的线程会在一个单独的系统级线程中执行（比如说一个 POSIX 线程或者一个 Windows 线程），这些线程将由操作系统来全权管理。线程一旦启动，将独立执行直到目标函数返回。你可以查询一个线程对象的状态，看它是否还在执行：

```
if t.is_alive():
    print('Still running')
else:
    print('Completed')
```

你也可以将一个线程加入到当前线程，并等待它终止：

```
t.join()
```

Python解释器直到所有线程都终止前仍保持运行。对于需要长时间运行的线程或者需要一直运行的后台任务，你应当考虑使用后台线程。 例如：

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

后台线程无法等待，不过，这些线程会在主线程终止时自动销毁。 除了如上所示的两个操作，并没有太多可以对线程做的事情。你无法结束一个线程，无法给它发送信号，无法调整它的调度，也无法执行其他高级操作。如果需要这些特性，你需要自己添加。比如说，如果你需要终止线程，那么这个线程必须通过编程在某个特定点轮询来退出。你可以像下边这样把线程放入一个类中：

```
class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
```

```
t.start()
c.terminate() # Signal termination
t.join()      # Wait for actual termination (if needed)
```

如果线程执行一些像I/O这样的阻塞操作，那么通过轮询来终止线程将使得线程之间的协调变得非常棘手。比如，如果一个线程一直阻塞在一个I/O操作上，它就永远无法返回，也就无法检查自己是否已经被结束了。要正确处理这些问题，你需要利用超时循环来小心操作线程。 例子如下：

```
class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)        # Set timeout period
        while self._running:
            # Perform a blocking I/O operation w/ timeout
            try:
                data = sock.recv(8192)
                break
            except socket.timeout:
                continue
            # Continued processing
            ...
        # Terminated
        return
```

讨论¶

由于全局解释锁（GIL）的原因，Python 的线程被限制到同一时刻只允许一个线程执行这样一个执行模型。所以，Python 的线程更适用于处理I/O和其他需要并发执行的阻塞操作（比如等待I/O、等待从数据库获取数据等等），而不是需要多处理器并行的计算密集型任务。

有时你会看到下边这种通过继承 `Thread` 类来实现的线程：

```
from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = n
    def run(self):
        while self.n > 0:

            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

c = CountdownThread(5)
c.start()
```

尽管这样也可以工作，但这使得你的代码依赖于 `threading` 库，所以你的这些代码只能在线程上下文中使用。上文所写的那些代码、函数都是与 `threading` 库无关的，这样就使得这些代码可以被用在其他的上下文中，可能与线程有关，也可能与线程无关。比如，你可以通过 `multiprocessing` 模块在一个单独的进程中执行你的代码：

```
import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
```

再次重申，这段代码仅适用于 CountdownTask 类是以独立于实际的并发手段（多线程、多进程等等）实现的情况。

12.10 定义一个Actor任务¶

问题¶

你想定义跟actor模式中类似“actors”角色的任务

解决方案¶

actor模式是一种最古老的也是最简单的并行和分布式计算解决方案。 事实上，它天生的简单性是它如此受欢迎的重要原因之一。 简单来讲，一个actor就是一个并发执行的任务，只是简单的执行发送给它的消息任务。 响应这些消息时，它可能还会给其他actor发送更进一步的消息。 actor之间的通信是单向和异步的。因此，消息发送者不知道消息是什么时候被发送， 也不会接收到一个消息已被处理的回应或通知。

结合使用一个线程和一个队列可以很容易的定义actor，例如：

```
from queue import Queue
from threading import Thread, Event

# Sentinel used for shutdown
class ActorExit(Exception):
    pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        '''
        Send a message to the actor
        '''
        self._mailbox.put(msg)

    def recv(self):
        '''
        Receive an incoming message
        '''
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        '''
        Close the actor, thus shutting it down
        '''
        self.send(ActorExit)

    def start(self):
        '''
        Start concurrent execution
        '''
        self._terminated = Event()
        t = Thread(target=self._bootstrap)

        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
```

```
        self._terminated.wait()

    def run(self):
        '''
        Run method to be implemented by the user
        '''
        while True:
            msg = self.recv()

# Sample ActorTask
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print('Got:', msg)

# Sample use
p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()
```

这个例子中，你使用actor实例的 `send()` 方法发送消息给它们。 其机制是，这个方法会将消息放入一个队里中， 然后将其转交给处理被接受消息的一个内部线程。 `close()` 方法通过在队列中放入一个特殊的哨兵值（ActorExit）来关闭这个actor。 用户可以通过继承Actor并定义实现自己处理逻辑run()方法来定义新的actor。 `ActorExit` 异常的使用就是用户自定义代码可以在需要的时候来捕获终止请求 （异常被get()方法抛出并传播出去）。

如果你放宽对于同步和异步消息发送的要求， 类actor对象还可以通过生成器来简化定义。例如：

```
def print_actor():
    while True:

        try:
            msg = yield      # Get a message
            print('Got:', msg)
        except GeneratorExit:
            print('Actor terminating')

# Sample use
p = print_actor()
next(p)     # Advance to the yield (ready to receive)
p.send('Hello')
p.send('World')
p.close()
```

讨论¶

actor模式的魅力就在于它的简单性。 实际上，这里仅仅只有一个核心操作 `send()` . 甚至，对于在基于actor系统中的“消息”的泛化概念可以已多种方式被扩展。 例如，你可以以元组形式传递标签消息，让actor执行不同的操作，如下：

```
class TaggedActor(Actor):
    def run(self):
        while True:
             tag, *payload = self.recv()
             getattr(self,'do_'+tag)(*payload)

    # Methods correponding to different message tags
    def do_A(self, x):
        print('Running A', x)

    def do_B(self, x, y):
        print('Running B', x, y)

# Example
a = TaggedActor()
a.start()
```

```
a.send(('A', 1))      # Invokes do_A(1)
a.send(('B', 2, 3))   # Invokes do_B(2,3)
a.close()
a.join()
```

作为另外一个例子，下面的actor允许在一个工作者中运行任意的函数， 并且通过一个特殊的Result对象返回结果：

```
from threading import Event
class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value

        self._evt.set()

    def result(self):
        self._evt.wait()
        return self._result

class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Example use
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
worker.close()
worker.join()
print(r.result())
```

最后，“发送”一个任务消息的概念可以被扩展到多进程甚至是大型分布式系统中去。 例如，一个类actor对象的 `send()` 方法可以被编程让它能在一个套接字连接上传输数据 或通过某些消息中间件（比如AMQP、ZMQ等）来发送。

12.11 实现消息发布/订阅模型¶

问题¶

你有一个基于线程通信的程序，想让它们实现发布/订阅模式的消息通信。

解决方案¶

要实现发布/订阅的消息通信模式， 你通常要引入一个单独的“交换机”或“网关”对象作为所有消息的中介。 也就是说，不直接将消息从一个任务发送到另一个，而是将其发送给交换机， 然后由交换机将它发送给一个或多个被关联任务。下面是一个非常简单的交换机实现例子：

```
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]
```

一个交换机就是一个普通对象，负责维护一个活跃的订阅者集合，并为绑定、解绑和发送消息提供相应的方法。 每个交换机通过一个名称定位，`get_exchange()` 通过给定一个名称返回相应的 `Exchange` 实例。

下面是一个简单例子，演示了如何使用一个交换机：

```
# Example of a task.  Any object with a send() method

class Task:
    ...
    def send(self, msg):
        ...

task_a = Task()
task_b = Task()

# Example of getting an exchange
exc = get_exchange('name')

# Examples of subscribing tasks to it
exc.attach(task_a)
exc.attach(task_b)

# Example of sending messages
exc.send('msg1')
exc.send('msg2')

# Example of unsubscribing
exc.detach(task_a)
exc.detach(task_b)
```

尽管对于这个问题有很多的变种，不过万变不离其宗。 消息会被发送给一个交换机，然后交换机会将它们发送给被绑定的订阅者。

讨论¶

通过队列发送消息的任务或线程的模式很容易被实现并且也非常普遍。 不过，使用发布/订阅模式的好处更加明显。

首先，使用一个交换机可以简化大部分涉及到线程通信的工作。 无需去写通过多进程模块来操作多个线程，你只需要使用这个交换机来连接它们。 某种程度上，这个就跟日志模块的工作原理类似。 实际上，它可以轻松的解耦程序中多个任务。

其次，交换机广播消息给多个订阅者的能力带来了一个全新的通信模式。 例如，你可以使用多任务系统、广播或扇出。 你还可以通过以普通订阅者身份绑定来构建调试和诊断工具。 例如，下面是一个简单的诊断类，可以显示被发送的消息：

```
class DisplayMessages:
    def __init__(self):
        self.count = 0
    def send(self, msg):
        self.count += 1
        print('msg[{}]: {!r}'.format(self.count, msg))

exc = get_exchange('name')
d = DisplayMessages()
exc.attach(d)
```

最后，该实现的一个重要特点是它能兼容多个“task-like”对象。 例如，消息接受者可以是actor（12.10小节介绍）、协程、网络连接或任何实现了正确的 `send()` 方法的东西。

关于交换机的一个可能问题是对于订阅者的正确绑定和解绑。 为了正确的管理资源，每一个绑定的订阅者必须最终要解绑。 在代码中通常会是像下面这样的模式：

```
exc = get_exchange('name')
exc.attach(some_task)
try:
    ...
finally:
    exc.detach(some_task)
```

某种意义上，这个和使用文件、锁和类似对象很像。 通常很容易会忘记最后的 `detach()` 步骤。 为了简化这个，你可以考虑使用上下文管理器协议。 例如，在交换机对象上增加一个 `subscribe()` 方法，如下：

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
        for task in tasks:
            self.attach(task)
        try:
            yield
        finally:
            for task in tasks:
                self.detach(task)

    def send(self, msg):
```

```
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]

# Example of using the subscribe() method
exc = get_exchange('name')
with exc.subscribe(task_a, task_b):
     ...
     exc.send('msg1')
     exc.send('msg2')
     ...

# task_a and task_b detached here
```

最后还应该注意的是关于交换机的思想有很多种的扩展实现。 例如，交换机可以实现一整个消息通道集合或提供交换机名称的模式匹配规则。 交换机还可以被扩展到分布式计算程序中（比如，将消息路由到不同机器上面的任务中去）。

12.12 使用生成器代替线程¶

问题¶

你想使用生成器（协程）替代系统线程来实现并发。这个有时又被称为用户级线程或绿色线程。

解决方案¶

要使用生成器实现自己的并发，你首先要对生成器函数和 `yield` 语句有深刻理解。 `yield` 语句会让一个生成器挂起它的执行，这样就可以编写一个调度器， 将生成器当做某种“任务”并使用任务协作切换来替换它们的执行。 要演示这种思想，考虑下面两个使用简单的 `yield` 语句的生成器函数：

```
# Two simple generator functions
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
    print('Blastoff!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1
```

这些函数在内部使用yield语句，下面是一个实现了简单任务调度器的代码：

```
from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        '''
        Admit a newly started task to the scheduler
        '''
        self._task_queue.append(task)

    def run(self):
        '''
        Run until there are no more tasks
        '''
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                # Run until the next yield statement
                next(task)
                self._task_queue.append(task)
            except StopIteration:
                # Generator is no longer executing
                pass

# Example use
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

`TaskScheduler` 类在一个循环中运行生成器集合——每个都运行到碰到yield语句为止。 运行这个例子，输出如下：

```
T-minus 10
T-minus 5
```

```
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...
```

到此为止，我们实际上已经实现了一个“操作系统”的最小核心部分。 生成器函数就是任务，而yield语句是任务挂起的信号。 调度器循环检查任务列表直到没有任务要执行为止。

实际上，你可能想要使用生成器来实现简单的并发。 那么，在实现actor或网络服务器的时候你可以使用生成器来替代线程的使用。

下面的代码演示了使用生成器来实现一个不依赖线程的actor：

```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = {}          # Mapping of names to actors
        self._msg_queue = deque()   # Message queue

    def new_actor(self, name, actor):
        '''
        Admit a newly started actor to the scheduler and give it a name
        '''
        self._msg_queue.append((actor,None))
        self._actors[name] = actor

    def send(self, name, msg):
        '''
        Send a message to a named actor
        '''
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor,msg))

    def run(self):
        '''
        Run as long as there are pending messages.
        '''
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                 actor.send(msg)
            except StopIteration:
                 pass

# Example use
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            # Receive the current count
            n = yield
            if n == 0:
                break
            # Send to the printer task
            sched.send('printer', n)
            # Send the next count to the counter task (recursive)
            sched.send('counter', n-1)
```

```
    sched = ActorScheduler()
    # Create the initial actors
    sched.new_actor('printer', printer())
    sched.new_actor('counter', counter(sched))

    # Send an initial message to the counter to initiate
    sched.send('counter', 10000)
    sched.run()
```

完全弄懂这段代码需要更深入的学习，但是关键点在于收集消息的队列。 本质上，调度器在有需要发送的消息时会一直运行着。 计数生成器会给自己发送消息并在一个递归循环中结束。

下面是一个更加高级的例子，演示了使用生成器来实现一个并发网络应用程序：

```
from collections import deque
from select import select

# This class represents a generic yield event in the scheduler
class YieldEvent:
    def handle_yield(self, sched, task):
        pass

    def handle_resume(self, sched, task):
        pass

# Task Scheduler
class Scheduler:
    def __init__(self):
        self._numtasks = 0       # Total num of tasks
        self._ready = deque()    # Tasks ready to run
        self._read_waiting = {}  # Tasks waiting to read
        self._write_waiting = {} # Tasks waiting to write

    # Poll for I/O events and restart waiting tasks
    def _iopoll(self):
        rset,wset,eset = select(self._read_waiting,
                                self._write_waiting,[])
        for r in rset:
            evt, task = self._read_waiting.pop(r)
            evt.handle_resume(self, task)
        for w in wset:
            evt, task = self._write_waiting.pop(w)
            evt.handle_resume(self, task)

    def new(self,task):
        '''
        Add a newly started task to the scheduler
        '''
        self._ready.append((task, None))
        self._numtasks += 1

    def add_ready(self, task, msg=None):
        '''
        Append an already started task to the ready queue.
        msg is what to send into the task when it resumes.
        '''
        self._ready.append((task, msg))

    # Add a task to the reading set
    def _read_wait(self, fileno, evt, task):
        self._read_waiting[fileno] = (evt, task)

    # Add a task to the write set
    def _write_wait(self, fileno, evt, task):
        self._write_waiting[fileno] = (evt, task)

    def run(self):
        '''
        Run the task scheduler until there are no tasks
        '''
```

```
        while self._numtasks:
             if not self._ready:
                  self._iopoll()
             task, msg = self._ready.popleft()
             try:
                 # Run the coroutine to the next yield
                 r = task.send(msg)
                 if isinstance(r, YieldEvent):
                     r.handle_yield(self, task)
                 else:
                     raise RuntimeError('unrecognized yield event')
             except StopIteration:
                 self._numtasks -= 1

# Example implementation of coroutine-based socket I/O
class ReadSocket(YieldEvent):
    def __init__(self, sock, nbytes):
        self.sock = sock
        self.nbytes = nbytes
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        data = self.sock.recv(self.nbytes)
        sched.add_ready(task, data)

class WriteSocket(YieldEvent):
    def __init__(self, sock, data):
        self.sock = sock
        self.data = data

    def handle_yield(self, sched, task):
        sched._write_wait(self.sock.fileno(), self, task)

    def handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket(YieldEvent):
    def __init__(self, sock):
        self.sock = sock

    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)

    def handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Wrapper around a socket object for use with yield
class Socket(object):
    def __init__(self, sock):
        self._sock = sock

    def recv(self, maxbytes):
        return ReadSocket(self._sock, maxbytes)

    def send(self, data):
        return WriteSocket(self._sock, data)

    def accept(self):
        return AcceptSocket(self._sock)

    def __getattr__(self, name):
        return getattr(self._sock, name)

if __name__ == '__main__':
    from socket import socket, AF_INET, SOCK_STREAM
    import time

    # Example of a function involving generators.  This should
    # be called using line = yield from readline(sock)
```

```
    def readline(sock):
        chars = []
        while True:
            c = yield sock.recv(1)
            if not c:
                break
            chars.append(c)
            if c == b'\n':
                break
        return b''.join(chars)

    # Echo server using generators
    class EchoServer:
        def __init__(self,addr,sched):
            self.sched = sched
            sched.new(self.server_loop(addr))

        def server_loop(self,addr):
            s = Socket(socket(AF_INET,SOCK_STREAM))

            s.bind(addr)
            s.listen(5)
            while True:
                c,a = yield s.accept()
                print('Got connection from ', a)
                self.sched.new(self.client_handler(Socket(c)))

        def client_handler(self,client):
            while True:
                line = yield from readline(client)
                if not line:
                    break
                line = b'GOT:' + line
                while line:
                    nsent = yield client.send(line)
                    line = line[nsent:]
            client.close()
            print('Client closed')

    sched = Scheduler()
    EchoServer(('',16000),sched)
    sched.run()
```

这段代码有点复杂。不过，它实现了一个小型的操作系统。 有一个就绪的任务队列，并且还有因I/O休眠的任务等待区域。 还有很多调度器负责在就绪队列和I/O等待区域之间移动任务。

讨论¶

在构建基于生成器的并发框架时，通常会使用更常见的yield形式：

```
def some_generator():
    ...
    result = yield data
    ...
```

使用这种形式的yield语句的函数通常被称为“协程”。 通过调度器，yield语句在一个循环中被处理，如下：

```
f = some_generator()

# Initial result. Is None to start since nothing has been computed
result = None
while True:
    try:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration:
        break
```

这里的逻辑稍微有点复杂。不过，被传给 `send()` 的值定义了在yield语句醒来时的返回值。 因此，如果一个yield准备在

对之前yield数据的回应中返回结果时，会在下一次 `send()` 操作返回。 如果一个生成器函数刚开始运行，发送一个None值会让它排在第一个yield语句前面。

除了发送值外，还可以在一个生成器上面执行一个 `close()` 方法。 它会导致在执行yield语句时抛出一个 `GeneratorExit` 异常，从而终止执行。 如果进一步设计，一个生成器可以捕获这个异常并执行清理操作。 同样还可以使用生成器的 `throw()` 方法在yield语句执行时生成一个任意的执行指令。 一个任务调度器可利用它来在运行的生成器中处理错误。

最后一个例子中使用的 `yield from` 语句被用来实现协程，可以被其它生成器作为子程序或过程来调用。 本质上就是将控制权透明的传输给新的函数。 不像普通的生成器，一个使用 `yield from` 被调用的函数可以返回一个作为 `yield from` 语句结果的值。 关于 `yield from` 的更多信息可以在 PEP 380 中找到。

最后，如果使用生成器编程，要提醒你的是它还是有很多缺点的。 特别是，你得不到任何线程可以提供的好处。例如，如果你执行CPU依赖或I/O阻塞程序， 它会将整个任务挂起直到操作完成。为了解决这个问题， 你只能选择将操作委派给另外一个可以独立运行的线程或进程。 另外一个限制是大部分Python库并不能很好的兼容基于生成器的线程。 如果你选择这个方案，你会发现你需要自己改写很多标准库函数。 作为本节提到的协程和相关技术的一个基础背景，可以查看 PEP 342 和 “协程和并发的一门有趣课程”

PEP 3156 同样有一个关于使用协程的异步I/O模型。 特别的，你不可能自己去实现一个底层的协程调度器。 不过，关于协程的思想是很多流行库的基础， 包括 gevent, greenlet, Stackless Python 以及其他类似工程。

12.13 多个线程队列轮询¶

问题¶

你有一个线程队列集合，想为到来的元素轮询它们， 就跟你为一个客户端请求去轮询一个网络连接集合的方式一样。

解决方案¶

对于轮询问题的一个常见解决方案中有个很少有人知道的技巧，包含了一个隐藏的回路网络连接。 本质上讲其思想就是：对于每个你想要轮询的队列，你创建一对连接的套接字。 然后你在其中一个套接字上面编写代码来标识存在的数据， 另外一个套接字被传给 `select()` 或类似的一个轮询数据到达的函数。下面的例子演示了这个思想：

```
import queue
import socket
import os

class PollableQueue(queue.Queue):
    def __init__(self):
        super().__init__()
        # Create a pair of connected sockets
        if os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair()
        else:
            # Compatibility on non-POSIX systems
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self._putsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()

    def fileno(self):
        return self._getsocket.fileno()

    def put(self, item):
        super().put(item)
        self._putsocket.send(b'x')

    def get(self):
        self._getsocket.recv(1)
        return super().get()
```

在这个代码中，一个新的 `Queue` 实例类型被定义，底层是一个被连接套接字对。 在Unix机器上的 `socketpair()` 函数能轻松的创建这样的套接字。 在Windows上面，你必须使用类似代码来模拟它。 然后定义普通的 `get()` 和 `put()` 方法在这些套接字上面来执行I/O操作。 `put()` 方法再将数据放入队列后会写一个单字节到某个套接字中去。 而 `get()` 方法在从队列中移除一个元素时会从另外一个套接字中读取到这个单字节数据。

`fileno()` 方法使用一个函数比如 `select()` 来让这个队列可以被轮询。 它仅仅只是暴露了底层被 `get()` 函数使用到的socket的文件描述符而已。

下面是一个例子，定义了一个为到来的元素监控多个队列的消费者：

```
import select
import threading

def consumer(queues):
    '''
    Consumer that reads data on multiple queues simultaneously
    '''
    while True:
        can_read, _, _ = select.select(queues,[],[])
        for r in can_read:
            item = r.get()
```

```
            print('Got:', item)

q1 = PollableQueue()
q2 = PollableQueue()
q3 = PollableQueue()
t = threading.Thread(target=consumer, args=([q1,q2,q3],))
t.daemon = True
t.start()

# Feed data to the queues
q1.put(1)
q2.put(10)
q3.put('hello')
q2.put(15)
...
```

如果你试着运行它，你会发现这个消费者会接受到所有的被放入的元素，不管元素被放进了哪个队列中。

讨论¶

对于轮询非类文件对象，比如队列通常都是比较棘手的问题。 例如，如果你不使用上面的套接字技术， 你唯一的选择就是编写代码来循环遍历这些队列并使用一个定时器。像下面这样：

```
import time
def consumer(queues):
    while True:
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)

        # Sleep briefly to avoid 100% CPU
        time.sleep(0.01)
```

这样做其实不合理，还会引入其他的性能问题。 例如，如果新的数据被加入到一个队列中，至少要花10毫秒才能被发现。 如果你之前的轮询还要去轮询其他对象，比如网络套接字那还会有更多问题。 例如，如果你想同时轮询套接字和队列，你可能要像下面这样使用：

```
import select

def event_loop(sockets, queues):
    while True:
        # polling with a timeout
        can_read, _, _ = select.select(sockets, [], [], 0.01)
        for r in can_read:
            handle_read(r)
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)
```

这个方案通过将队列和套接字等同对待来解决了大部分的问题。 一个单独的 `select()` 调用可被同时用来轮询。 使用超时或其他基于时间的机制来执行周期性检查并没有必要。 甚至，如果数据被加入到一个队列，消费者几乎可以实时的被通知。 尽管会有一点点底层的I/O损耗，使用它通常会获得更好的响应时间并简化编程。

12.14 在Unix系统上面启动守护进程¶

问题¶

你想编写一个作为一个在Unix或类Unix系统上面运行的守护进程运行的程序。

解决方案¶

创建一个正确的守护进程需要一个精确的系统调用序列以及对于细节的控制。 下面的代码展示了怎样定义一个守护进程，可以启动后很容易的停止它。

```
#!/usr/bin/env python3
# daemon.py

import os
import sys

import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
                          stdout='/dev/null',
                          stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # First fork (detaches from parent)
    try:
        if os.fork() > 0:
            raise SystemExit(0)   # Parent exit
    except OSError as e:
        raise RuntimeError('fork #1 failed.')

    os.chdir('/')
    os.umask(0)
    os.setsid()
    # Second fork (relinquish session leadership)
    try:
        if os.fork() > 0:
            raise SystemExit(0)
    except OSError as e:
        raise RuntimeError('fork #2 failed.')

    # Flush I/O buffers
    sys.stdout.flush()
    sys.stderr.flush()

    # Replace file descriptors for stdin, stdout, and stderr
    with open(stdin, 'rb', 0) as f:
        os.dup2(f.fileno(), sys.stdin.fileno())
    with open(stdout, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stdout.fileno())
    with open(stderr, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stderr.fileno())

    # Write the PID file
    with open(pidfile,'w') as f:
        print(os.getpid(),file=f)

    # Arrange to have the PID file removed on exit/signal
    atexit.register(lambda: os.remove(pidfile))

    # Signal handler for termination (required)
    def sigterm_handler(signo, frame):
        raise SystemExit(1)
```

```
    signal.signal(signal.SIGTERM, sigterm_handler)

def main():
    import time
    sys.stdout.write('Daemon started with pid {}\n'.format(os.getpid()))
    while True:
        sys.stdout.write('Daemon Alive! {}\n'.format(time.ctime()))
        time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit(1)

    if sys.argv[1] == 'start':
        try:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
                      stderr='/tmp/dameon.log')
        except RuntimeError as e:
            print(e, file=sys.stderr)
            raise SystemExit(1)

        main()

    elif sys.argv[1] == 'stop':
        if os.path.exists(PIDFILE):
            with open(PIDFILE) as f:
                os.kill(int(f.read()), signal.SIGTERM)
        else:
            print('Not running', file=sys.stderr)
            raise SystemExit(1)

    else:
        print('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr)
        raise SystemExit(1)
```

要启动这个守护进程，用户需要使用如下的命令：

```
bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...
```

守护进程可以完全在后台运行，因此这个命令会立即返回。 不过，你可以像上面那样查看与它相关的pid文件和日志。 要停止这个守护进程，使用：

```
bash % daemon.py stop
bash %
```

讨论¶

本节定义了一个函数 `daemonize()` ，在程序启动时被调用使得程序以一个守护进程来运行。 `daemonize()` 函数只接受关键字参数，这样的话可选参数在被使用时就更清晰了。 它会强制用户像下面这样使用它：

```
daemonize('daemon.pid',
          stdin='/dev/null',
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

而不是像下面这样含糊不清的调用：

```
# Illegal. Must use keyword arguments
daemonize('daemon.pid',
          '/dev/null', '/tmp/daemon.log','/tmp/daemon.log')
```

创建一个守护进程的步骤看上去不是很易懂，但是大体思想是这样的， 首先，一个守护进程必须要从父进程中脱离。这是由 `os.fork()` 操作来完成的，子进程创建之后，父进程立即被终止。

在子进程变成孤儿后，调用 `os.setsid()` 创建了一个全新的进程会话，并设置子进程为首领。 它会设置这个子进程为新的进程组的首领，并确保不会再有控制终端。 如果这些听上去太魔幻，因为它需要将守护进程同终端分离开并确保信号机制对它不起作用。 调用 `os.chdir()` 和 `os.umask(0)` 改变了当前工作目录并重置文件权限掩码。 修改目录通常是个好主意，因为这样可以使得它不再工作在被启动时的目录。

另外一个调用 `os.fork()` 在这里更加神秘点。 这一步使得守护进程失去了获取新的控制终端的能力并且让它更加独立（本质上，该daemon放弃了它的会话首领地位，因此再也没有权限去打开控制终端了）。 尽管你可以忽略这一步，但是最好不要这么做。

一旦守护进程被正确的分离，它会重新初始化标准I/O流指向用户指定的文件。 这一部分有点难懂。跟标准I/O流相关的文件对象的引用在解释器中多个地方被找到 （sys.stdout, sys.__stdout__等）。 仅仅简单的关闭 `sys.stdout` 并重新指定它是行不通的， 因为没办法知道它是否全部都是用的是 `sys.stdout` 。 这里，我们打开了一个单独的文件对象，并调用 `os.dup2()` ， 用它来代替被 `sys.stdout` 使用的文件描述符。 这样，`sys.stdout` 使用的原始文件会被关闭并由新的来替换。 还要强调的是任何用于文件编码或文本处理的标准I/O流还会保留原状。

守护进程的一个通常实践是在一个文件中写入进程ID，可以被其他程序后面使用到。 `daemonize()` 函数的最后部分写了这个文件，但是在程序终止时删除了它。 `atexit.register()` 函数注册了一个函数在Python解释器终止时执行。 一个对于SIGTERM的信号处理器的定义同样需要被优雅的关闭。 信号处理器简单的抛出了 `SystemExit()` 异常。 或许这一步看上去没必要，但是没有它， 终止信号会使得不执行 `atexit.register()` 注册的清理操作的时候就杀掉了解释器。 一个杀掉进程的例子代码可以在程序最后的 `stop` 命令的操作中看到。

更多关于编写守护进程的信息可以查看《UNIX 环境高级编程》, 第二版 by W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005)。 尽管它是关注与C语言编程，但是所有的内容都适用于Python， 因为所有需要的POSIX函数都可以在标准库中找到。

12.2 判断线程是否已经启动¶

问题¶

你已经启动了一个线程，但是你想知道它是不是真的已经开始运行了。

解决方案¶

线程的一个关键特性是每个线程都是独立运行且状态不可预测。如果程序中的其他线程需要通过判断某个线程的状态来确定自己下一步的操作，这时线程同步问题就会变得非常棘手。为了解决这些问题，我们需要使用 `threading` 库中的 `Event` 对象。 `Event` 对象包含一个可由线程设置的信号标志，它允许线程等待某些事件的发生。在初始情况下，event 对象中的信号标志被设置为假。如果有线程等待一个 event 对象，而这个 event 对象的标志为假，那么这个线程将会被一直阻塞直至该标志为真。一个线程如果将一个 event 对象的信号标志设置为真，它将唤醒所有等待这个 event 对象的线程。如果一个线程等待一个已经被设置为真的 event 对象，那么它将忽略这个事件，继续执行。 下边的代码展示了如何使用 `Event` 来协调线程的启动：

```
from threading import Thread, Event
import time

# Code to execute in an independent thread
def countdown(n, started_evt):
    print('countdown starting')
    started_evt.set()
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create the event object that will be used to signal startup
started_evt = Event()

# Launch the thread and pass the startup event
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Wait for the thread to start
started_evt.wait()
print('countdown is running')
```

当你执行这段代码，“countdown is running” 总是显示在 “countdown starting” 之后显示。这是由于使用 event 来协调线程，使得主线程要等到 `countdown()` 函数输出启动信息后，才能继续执行。

讨论¶

event 对象最好单次使用，就是说，你创建一个 event 对象，让某个线程等待这个对象，一旦这个对象被设置为真，你就应该丢弃它。尽管可以通过 `clear()` 方法来重置 event 对象，但是很难确保安全地清理 event 对象并对它重新赋值。很可能会发生错过事件、死锁或者其他问题（特别是，你无法保证重置 event 对象的代码会在线程再次等待这个 event 对象之前执行）。如果一个线程需要不停地重复使用 event 对象，你最好使用 `Condition` 对象来代替。下面的代码使用 `Condition` 对象实现了一个周期定时器，每当定时器超时的时候，其他线程都可以监测到：

```
import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
```

```
        t.daemon = True

        t.start()

    def run(self):
        '''
        Run the timer and notify waiting threads after each interval
        '''
        while True:
            time.sleep(self._interval)
            with self._cv:
                 self._flag ^= 1
                 self._cv.notify_all()

    def wait_for_tick(self):
        '''
        Wait for the next tick of the timer
        '''
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()

# Two threads that synchronize on the timer
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1

def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()
```

event对象的一个重要特点是当它被设置为真时会唤醒所有等待它的线程。如果你只想唤醒单个线程，最好是使用信号量或者 `Condition` 对象来替代。考虑一下这段使用信号量实现的代码：

```
# Worker thread
def worker(n, sema):
    # Wait to be signaled
    sema.acquire()

    # Do some work
    print('Working', n)

# Create some threads
sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.start()
```

运行上边的代码将会启动一个线程池，但是并没有什么事情发生。这是因为所有的线程都在等待获取信号量。每次信号量被释放，只有一个线程会被唤醒并执行，示例如下：

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

编写涉及到大量的线程间同步问题的代码会让你痛不欲生。比较合适的方式是使用队列来进行线程间通信或者每个把线程当作一个Actor，利用Actor模型来控制并发。下一节将会介绍到队列，而Actor模型将在12.10节介绍。

12.3 线程间通信¶

问题¶

你的程序中有多个线程，你需要在这些线程之间安全地交换信息或数据

解决方案¶

从一个线程向另一个线程发送数据最安全的方式可能就是使用 queue 库中的队列了。创建一个被多个线程共享的 Queue 对象，这些线程通过使用 put() 和 get() 操作来向队列中添加或者删除元素。 例如：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
# Get some data
        data = in_q.get()
        # Process the data
        ...

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Queue 对象已经包含了必要的锁，所以你可以通过它在多个线程间多安全地共享数据。 当使用队列时，协调生产者和消费者的关闭问题可能会有一些麻烦。一个通用的解决方法是在队列中放置一个特殊的值，当消费者读到这个值的时候，终止执行。例如：

```
from queue import Queue
from threading import Thread

# Object that signals shutdown
_sentinel = object()

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

    # Put the sentinel on the queue to indicate completion
    out_q.put(_sentinel)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Check for termination
        if data is _sentinel:
            in_q.put(_sentinel)
            break
```

```
        # Process the data
        ...
```

本例中有一个特殊的地方：消费者在读到这个特殊值之后立即又把它放回到队列中，将之传递下去。这样，所有监听这个队列的消费者线程就可以全部关闭了。 尽管队列是最常见的线程间通信机制，但是仍然可以自己通过创建自己的数据结构并添加所需的锁和同步机制来实现线程间通信。最常见的方法是使用 `Condition` 变量来包装你的数据结构。下边这个例子演示了如何创建一个线程安全的优先级队列，如同1.5节中介绍的那样。

```
import heapq
import threading

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

使用队列来进行线程间通信是一个单向、不确定的过程。通常情况下，你没有办法知道接收数据的线程是什么时候接收到的数据并开始工作的。不过队列对象提供一些基本完成的特性，比如下边这个例子中的 `task_done()` 和 `join()`：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Process the data
        ...
        # Indicate completion
        in_q.task_done()

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

# Wait for all produced items to be consumed
q.join()
```

如果一个线程需要在一个“消费者”线程处理完特定的数据项时立即得到通知，你可以把要发送的数据和一个 `Event` 放到一起使用，这样“生产者”就可以通过这个Event对象来监测处理的过程了。示例如下：

```
from queue import Queue
from threading import Thread, Event
```

```
# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        # Make an (data, event) pair and hand it to the consumer
        evt = Event()
        out_q.put((data, evt))
        ...
        # Wait for the consumer to process the item
        evt.wait()

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data, evt = in_q.get()
        # Process the data
        ...
        # Indicate completion
        evt.set()
```

讨论¶

基于简单队列编写多线程程序在多数情况下是一个比较明智的选择。从线程安全队列的底层实现来看，你无需在你的代码中使用锁和其他底层的同步机制，这些只会把你的程序弄得乱七八糟。此外，使用队列这种基于消息的通信机制可以被扩展到更大的应用范畴，比如，你可以把你的程序放入多个进程甚至是分布式系统而无需改变底层的队列结构。 使用线程队列有一个要注意的问题是，向队列中添加数据项时并不会复制此数据项，线程间通信实际上是在线程间传递对象引用。如果你担心对象的共享状态，那你最好只传递不可修改的数据结构（如：整型、字符串或者元组）或者一个对象的深拷贝。例如：

```
from queue import Queue
from threading import Thread
import copy

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(copy.deepcopy(data))

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...
```

`Queue` 对象提供一些在当前上下文很有用的附加特性。比如在创建 Queue 对象时提供可选的 `size` 参数来限制可以添加到队列中的元素数量。对于“生产者”与“消费者”速度有差异的情况，为队列中的元素数量添加上限是有意义的。比如，一个“生产者”产生项目的速度比“消费者”“消费”的速度快，那么使用固定大小的队列就可以在队列已满的时候阻塞队列，以免未预期的连锁效应扩散整个程序造成死锁或者程序运行失常。在通信的线程之间进行“流量控制”是一个看起来容易实现起来困难的问题。如果你发现自己曾经试图通过摆弄队列大小来解决一个问题，这也许就标志着你的程序可能存在脆弱设计或者固有的可伸缩问题。 `get()` 和 `put()` 方法都支持非阻塞方式和设定超时，例如：

```
import queue
q = queue.Queue()

try:
    data = q.get(block=False)
except queue.Empty:
    ...

try:
    q.put(item, block=False)
```

```
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...
```

这些操作都可以用来避免当执行某些特定队列操作时发生无限阻塞的情况，比如，一个非阻塞的 `put()` 方法和一个固定大小的队列一起使用，这样当队列已满时就可以执行不同的代码。比如输出一条日志信息并丢弃。

```
def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)
```

如果你试图让消费者线程在执行像 `q.get()` 这样的操作时，超时自动终止以便检查终止标志，你应该使用 `q.get()` 的可选参数 `timeout`，如下：

```
_running = True

def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Process item
            ...
        except queue.Empty:
            pass
```

最后，有 `q.qsize()`， `q.full()`， `q.empty()` 等实用方法可以获取一个队列的当前大小和状态。但要注意，这些方法都不是线程安全的。可能你对一个队列使用 `empty()` 判断出这个队列为空，但同时另外一个线程可能已经向这个队列中插入一个数据项。所以，你最好不要在你的代码中使用这些方法。

12.4 给关键部分加锁¶

问题¶

你需要对多线程程序中的临界区加锁以避免竞争条件。

解决方案¶

要在多线程程序中安全使用可变对象，你需要使用 threading 库中的 `Lock` 对象，就像下边这个例子这样：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with self._value_lock:
             self._value += delta

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        with self._value_lock:
             self._value -= delta
```

`Lock` 对象和 `with` 语句块一起使用可以保证互斥执行，就是每次只有一个线程可以执行 with 语句包含的代码块。with 语句会在这个代码块执行前自动获取锁，在执行结束后自动释放锁。

讨论¶

线程调度本质上是不确定的，因此，在多线程程序中错误地使用锁机制可能会导致随机数据损坏或者其他的异常行为，我们称之为竞争条件。为了避免竞争条件，最好只在临界区（对临界资源进行操作的那部分代码）使用锁。 在一些“老的”Python 代码中，显式获取和释放锁是很常见的。下边是一个上一个例子的变种：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
```

```
        '''
        self._value_lock.acquire()
        self._value -= delta
        self._value_lock.release()
```

相比于这种显式调用的方法，with 语句更加优雅，也更不容易出错，特别是程序员可能会忘记调用 release() 方法或者程序在获得锁之后产生异常这两种情况（使用 with 语句可以保证在这两种情况下仍能正确释放锁）。 为了避免出现死锁的情况，使用锁机制的程序应该设定为每个线程一次只允许获取一个锁。如果不能这样做的话，你就需要更高级的死锁避免机制，我们将在12.5节介绍。 在 `threading` 库中还提供了其他的同步原语，比如 `RLock` 和 `Semaphore` 对象。但是根据以往经验，这些原语是用于一些特殊的情况，如果你只是需要简单地对可变对象进行锁定，那就不应该使用它们。一个 `RLock` （可重入锁）可以被同一个线程多次获取，主要用来实现基于监测对象模式的锁定和同步。在使用这种锁的情况下，当锁被持有时，只有一个线程可以使用完整的函数或者类中的方法。比如，你可以实现一个这样的 SharedCounter 类：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    _lock = threading.RLock()
    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with SharedCounter._lock:
            self._value += delta

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        with SharedCounter._lock:
             self.incr(-delta)
```

在上边这个例子中，没有对每一个实例中的可变对象加锁，取而代之的是一个被所有实例共享的类级锁。这个锁用来同步类方法，具体来说就是，这个锁可以保证一次只有一个线程可以调用这个类方法。不过，与一个标准的锁不同的是，已经持有这个锁的方法在调用同样使用这个锁的方法时，无需再次获取锁。比如 decr 方法。 这种实现方式的一个特点是，无论这个类有多少个实例都只用一个锁。因此在需要大量使用计数器的情况下内存效率更高。不过这样做也有缺点，就是在程序中使用大量线程并频繁更新计数器时会有争用锁的问题。 信号量对象是一个建立在共享计数器基础上的同步原语。如果计数器不为0，with 语句将计数器减1，线程被允许执行。with 语句执行结束后，计数器加 1 。如果计数器为0，线程将被阻塞，直到其他线程结束将计数器加1。尽管你可以在程序中像标准锁一样使用信号量来做线程同步，但是这种方式并不被推荐，因为使用信号量为程序增加的复杂性会影响程序性能。相对于简单地作为锁使用，信号量更适用于那些需要在线程之间引入信号或者限制的程序。比如，你需要限制一段代码的并发访问量，你就可以像下面这样使用信号量完成：

```
from threading import Semaphore
import urllib.request

# At most, five threads allowed to run at once
_fetch_url_sema = Semaphore(5)

def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)
```

如果你对线程同步原语的底层理论和实现感兴趣，可以参考操作系统相关书籍，绝大多数都有提及。

12.5 防止死锁的加锁机制¶

问题¶

你正在写一个多线程程序，其中线程需要一次获取多个锁，此时如何避免死锁问题。

解决方案¶

在多线程程序中，死锁问题很大一部分是由于线程同时获取多个锁造成的。举个例子：一个线程获取了第一个锁，然后在获取第二个锁的 时候发生阻塞，那么这个线程就可能阻塞其他线程的执行，从而导致整个程序假死。 解决死锁问题的一种方案是为程序中的每一个锁分配一个唯一的id，然后只允许按照升序规则来使用多个锁，这个规则使用上下文管理器 是非常容易实现的，示例如下：

```
import threading
from contextlib import contextmanager

# Thread-local state to stored information on locks already acquired
_local = threading.local()

@contextmanager
def acquire(*locks):
    # Sort locks by object identifier
    locks = sorted(locks, key=lambda x: id(x))

    # Make sure lock order of previously acquired locks is not violated
    acquired = getattr(_local,'acquired',[])
    if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
        raise RuntimeError('Lock Order Violation')

    # Acquire all of the locks
    acquired.extend(locks)
    _local.acquired = acquired

    try:
        for lock in locks:
            lock.acquire()
        yield
    finally:
        # Release locks in reverse order of acquisition
        for lock in reversed(locks):
            lock.release()
        del acquired[-len(locks):]
```

如何使用这个上下文管理器呢？你可以按照正常途径创建一个锁对象，但不论是单个锁还是多个锁中都使用 `acquire()` 函数来申请锁， 示例如下：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
```

```
t2.daemon = True
t2.start()
```

如果你执行这段代码，你会发现它即使在不同的函数中以不同的顺序获取锁也没有发生死锁。 其关键在于，在第一段代码中，我们对这些锁进行了排序。通过排序，使得不管用户以什么样的顺序来请求锁，这些锁都会按照固定的顺序被获取。 如果有多个 `acquire()` 操作被嵌套调用，可以通过线程本地存储（TLS）来检测潜在的死锁问题。 假设你的代码是这样写的：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():

    while True:
        with acquire(x_lock):
            with acquire(y_lock):
                print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果你运行这个版本的代码，必定会有一个线程发生崩溃，异常信息可能像这样：

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_inner
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
  File "deadlock.py", line 15, in acquire
    raise RuntimeError("Lock Order Violation")
RuntimeError: Lock Order Violation
>>>
```

发生崩溃的原因在于，每个线程都记录着自己已经获取到的锁。 `acquire()` 函数会检查之前已经获取的锁列表， 由于锁是按照升序排列获取的，所以函数会认为之前已获取的锁的id必定小于新申请到的锁，这时就会触发异常。

讨论¶

死锁是每一个多线程程序都会面临的一个问题（就像它是每一本操作系统课本的共同话题一样）。根据经验来讲，尽可能保证每一个 线程只能同时保持一个锁，这样程序就不会被死锁问题所困扰。一旦有线程同时申请多个锁，一切就不可预料了。

死锁的检测与恢复是一个几乎没有优雅的解决方案的扩展话题。一个比较常用的死锁检测与恢复的方案是引入看门狗计数器。当线程正常 运行的时候会每隔一段时间重置计数器，在没有发生死锁的情况下，一切都正常进行。一旦发生死锁，由于无法重置计数器导致定时器 超时，这时程序会通过重启自身恢复到正常状态。

避免死锁是另外一种解决死锁问题的方式，在进程获取锁的时候会严格按照对象id升序排列获取，经过数学证明，这样保证程序不会进入 死锁状态。证明就留给读者作为练习了。避免死锁的主要思想是，单纯地按照对象id递增的顺序加锁不会产生循环依赖，而循环依赖是 死锁的一个必要条件，从而避免程序进入死锁状态。

下面以一个关于线程死锁的经典问题：“哲学家就餐问题”，作为本节最后一个例子。题目是这样的：五位哲学家围坐在一张桌子前，每个人 面前有一碗饭和一只筷子。在这里每个哲学家可以看做是一个独立的线程，而每只筷子可以看做是一个锁。每个哲学家可以处在静坐、 思考、吃饭三种状态中的一个。需要注意的是，每个哲学家吃饭是需要两只筷子的，这样问题就来了：如果每个哲学家都拿起自己左边的筷子， 那么他们五个都只能拿着一只筷子坐在那儿，直到饿死。此时他们就进入了死锁状态。 下面是一个简单的使用死锁避免机制解决“哲学家就餐问题”的实现：

```
import threading

# The philosopher thread
def philosopher(left, right):
    while True:
        with acquire(left,right):
             print(threading.currentThread(), 'eating')

# The chopsticks (represented by locks)
NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

# Create all of the philosophers
for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                         args=(chopsticks[n],chopsticks[(n+1) % NSTICKS]))
    t.start()
```

最后，要特别注意到，为了避免死锁，所有的加锁操作必须使用 `acquire()` 函数。如果代码中的某部分绕过acquire 函数直接申请锁，那么整个死锁避免机制就不起作用了。

12.6 保存线程的状态信息¶

问题¶

你需要保存正在运行线程的状态，这个状态对于其他的线程是不可见的。

解决方案¶

有时在多线程编程中，你需要只保存当前运行线程的状态。 要这么做，可使用 `thread.local()` 创建一个本地线程存储对象。 对这个对象的属性的保存和读取操作都只会对执行线程可见，而其他线程并不可见。

作为使用本地存储的一个有趣的实际例子， 考虑在8.3小节定义过的 `LazyConnection` 上下文管理器类。 下面我们对它进行一些小的修改使得它可以适用于多线程：

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        if hasattr(self.local, 'sock'):
            raise RuntimeError('Already connected')
        self.local.sock = socket(self.family, self.type)
        self.local.sock.connect(self.address)
        return self.local.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.sock.close()
        del self.local.sock
```

代码中，自己观察对于 `self.local` 属性的使用。 它被初始化为一个 `threading.local()` 实例。 其他方法操作被存储为 `self.local.sock` 的套接字对象。 有了这些就可以在多线程中安全的使用 `LazyConnection` 实例了。例如：

```
from functools import partial
def test(conn):
    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')

        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got {} bytes'.format(len(resp)))

if __name__ == '__main__':
    conn = LazyConnection(('www.python.org', 80))

    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

它之所以行得通的原因是每个线程会创建一个自己专属的套接字连接（存储为self.local.sock）。 因此，当不同的线程执行套接字操作时，由于操作的是不同的套接字，因此它们不会相互影响。

讨论¶

在大部分程序中创建和操作线程特定状态并不会有什么问题。 不过，当出了问题的时候，通常是因为某个对象被多个线程使用到，用来操作一些专用的系统资源， 比如一个套接字或文件。你不能让所有线程共享一个单独对象， 因为多个线程同时读和写的时候会产生混乱。 本地线程存储通过让这些资源只能在被使用的线程中可见来解决这个问题。

本节中，使用 `thread.local()` 可以让 `LazyConnection` 类支持一个线程一个连接， 而不是对于所有的进程都只有一个连接。

其原理是，每个 `threading.local()` 实例为每个线程维护着一个单独的实例字典。 所有普通实例操作比如获取、修改和删除值仅仅操作这个字典。 每个线程使用一个独立的字典就可以保证数据的隔离了。

12.7 创建一个线程池¶

问题¶

你创建一个工作者线程池，用来响应客户端请求或执行其他的工作。

解决方案¶

`concurrent.futures` 函数库有一个 `ThreadPoolExecutor` 类可以被用来完成这个任务。 下面是一个简单的TCP服务器，使用了一个线程池来响应客户端：

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr):
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

如果你想手动创建你自己的线程池， 通常可以使用一个Queue来轻松实现。下面是一个稍微不同但是手动实现的例子：

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    '''
    Handle a client connection
    '''
    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')

    sock.close()

def echo_server(addr, nworkers):
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
```

```
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

使用 `ThreadPoolExecutor` 相对于手动实现的一个好处在于它使得 任务提交者更方便的从被调用函数中获取返回值。例如，你可能会像下面这样写：

```
from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)
# Submit work to the pool
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Get the results back
x = a.result()
y = b.result()
```

例子中返回的handle对象会帮你处理所有的阻塞与协作，然后从工作线程中返回数据给你。 特别的，`a.result()` 操作会阻塞进程直到对应的函数执行完成并返回一个结果。

讨论¶

通常来讲，你应该避免编写线程数量可以无限制增长的程序。例如，看看下面这个服务器：

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        t = Thread(target=echo_client, args=(client_sock, client_addr))
        t.daemon = True
        t.start()

echo_server(('',15000))
```

尽管这个也可以工作， 但是它不能抵御有人试图通过创建大量线程让你服务器资源枯竭而崩溃的攻击行为。 通过使用预先初始化的线程池，你可以设置同时运行线程的上限数量。

你可能会关心创建大量线程会有什么后果。 现代操作系统可以很轻松的创建几千个线程的线程池。 甚至，同时几千个线程等待工作并不会对其他代码产生性能影响。 当然了，如果所有线程同时被唤醒并立即在CPU上执行，那就不同了——特别是有了全局解释器锁GIL。 通常，你应该只在I/O处理相关代码中使用线程池。

创建大的线程池的一个可能需要关注的问题是内存的使用。 例如，如果你在OS X系统上面创建2000个线程，系统显示Python进程使用了超过9GB的虚拟内存。 不过，这个计算通常是有误差的。当创建一个线程时，操作系统会预留一个虚拟内存区域来 放置线程的执行栈（通常是8MB大小）。但是这个内存只有一小片段被实际映射到真实内存中。 因此，Python进程使用到的真实内存其实很小 （比如，对于2000个线程来讲，只使用到了70MB的真实内存，而不是9GB）。 如果你担心虚拟内存大小，可以使用 `threading.stack_size()` 函数来降低它。例如：

```
import threading
threading.stack_size(65536)
```

如果你加上这条语句并再次运行前面的创建2000个线程试验， 你会发现Python进程只使用到了大概210MB的虚拟内存，而真实内存使用量没有变。 注意线程栈大小必须至少为32768字节，通常是系统内存页大小（4096、8192等）的整数倍。

12.8 简单的并行编程¶

问题¶

你有个程序要执行CPU密集型工作，你想让他利用多核CPU的优势来运行的快一点。

解决方案¶

`concurrent.futures` 库提供了一个 `ProcessPoolExecutor` 类， 可被用来在一个单独的Python解释器中执行计算密集型函数。 不过，要使用它，你首先要有一些计算密集型的任务。 我们通过一个简单而实际的例子来演示它。假定你有个Apache web服务器日志目录的gzip压缩包：

```
logs/
   20120701.log.gz
   20120702.log.gz
   20120703.log.gz
   20120704.log.gz
   20120705.log.gz
   20120706.log.gz
   ...
```

进一步假设每个日志文件内容类似下面这样：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

下面是一个脚本，在这些日志文件中查找出所有访问过robots.txt文件的主机：

```
# findrobots.py

import gzip
import io
import glob

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

前面的程序使用了通常的map-reduce风格来编写。 函数 `find_robots()` 在一个文件名集合上做map操作，并将结果汇总

为一个单独的结果， 也就是 `find_all_robots()` 函数中的 `all_robots` 集合。 现在，假设你想要修改这个程序让它使用多核CPU。 很简单——只需要将map()操作替换为一个 `concurrent.futures` 库中生成的类似操作即可。 下面是一个简单修改版本：

```
# findrobots.py

import gzip
import io
import glob
from concurrent import futures

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file

    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    with futures.ProcessPoolExecutor() as pool:
        for robots in pool.map(find_robots, files):
            all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

通过这个修改后，运行这个脚本产生同样的结果，但是在四核机器上面比之前快了3.5倍。 实际的性能优化效果根据你的机器CPU数量的不同而不同。

讨论¶

`ProcessPoolExecutor` 的典型用法如下：

```
from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as pool:
    ...
    do work in parallel using pool
    ...
```

其原理是，一个 `ProcessPoolExecutor` 创建N个独立的Python解释器， N是系统上面可用CPU的个数。你可以通过提供可选参数给 `ProcessPoolExecutor(N)` 来修改 处理器数量。这个处理池会一直运行到with块中最后一个语句执行完成，然后处理池被关闭。不过，程序会一直等待直到所有提交的工作被处理完成。

被提交到池中的工作必须被定义为一个函数。有两种方法去提交。 如果你想让一个列表推导或一个 `map()` 操作并行执行的话，可使用 `pool.map()`：

```
# A function that performs a lot of work
def work(x):
    ...
    return result

# Nonparallel code
```

```
results = map(work, data)

# Parallel implementation
with ProcessPoolExecutor() as pool:
    results = pool.map(work, data)
```

另外，你可以使用 `pool.submit()` 来手动的提交单个任务：

```
# Some function
def work(x):
    ...
    return result

with ProcessPoolExecutor() as pool:
    ...
    # Example of submitting work to the pool
    future_result = pool.submit(work, arg)

    # Obtaining the result (blocks until done)
    r = future_result.result()
    ...
```

如果你手动提交一个任务，结果是一个 `Future` 实例。 要获取最终结果，你需要调用它的 `result()` 方法。 它会阻塞进程直到结果被返回来。

如果不想阻塞，你还可以使用一个回调函数，例如：

```
def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
     future_result = pool.submit(work, arg)
     future_result.add_done_callback(when_done)
```

回调函数接受一个 `Future` 实例，被用来获取最终的结果（比如通过调用它的result()方法）。 尽管处理池很容易使用，在设计大程序的时候还是有很多需要注意的地方，如下几点：

- 这种并行处理技术只适用于那些可以被分解为互相独立部分的问题。
- 被提交的任务必须是简单函数形式。对于方法、闭包和其他类型的并行执行还不支持。
- 函数参数和返回值必须兼容pickle，因为要使用到进程间的通信，所有解释器之间的交换数据必须被序列化
- 被提交的任务函数不应保留状态或有副作用。除了打印日志之类简单的事情，

一旦启动你不能控制子进程的任何行为，因此最好保持简单和纯洁——函数不要去修改环境。

- 在Unix上进程池通过调用 `fork()` 系统调用被创建，

它会克隆Python解释器，包括fork时的所有程序状态。 而在Windows上，克隆解释器时不会克隆状态。 实际的fork操作会在第一次调用 `pool.map()` 或 `pool.submit()` 后发生。

- 当你混合使用进程池和多线程的时候要特别小心。

你应该在创建任何线程之前先创建并激活进程池（比如在程序启动的main线程中创建进程池）。

12.9 Python的全局锁问题¶

问题¶

你已经听说过全局解释器锁GIL，担心它会影响到多线程程序的执行性能。

解决方案¶

尽管Python完全支持多线程编程， 但是解释器的C语言实现部分在完全并行执行时并不是线程安全的。 实际上，解释器被一个全局解释器锁保护着，它确保任何时候都只有一个Python线程执行。 GIL最大的问题就是Python的多线程程序并不能利用多核CPU的优势 （比如一个使用了多个线程的计算密集型程序只会在一个单CPU上面运行）。

在讨论普通的GIL之前，有一点要强调的是GIL只会影响到那些严重依赖CPU的程序（比如计算型的）。 如果你的程序大部分只会涉及到I/O，比如网络交互，那么使用多线程就很合适， 因为它们大部分时间都在等待。实际上，你完全可以放心的创建几千个Python线程， 现代操作系统运行这么多线程没有任何压力，没啥可担心的。

而对于依赖CPU的程序，你需要弄清楚执行的计算的特点。 例如，优化底层算法要比使用多线程运行快得多。 类似的，由于Python是解释执行的，如果你将那些性能瓶颈代码移到一个C语言扩展模块中， 速度也会提升的很快。如果你要操作数组，那么使用NumPy这样的扩展会非常的高效。 最后，你还可以考虑下其他可选实现方案，比如PyPy，它通过一个JIT编译器来优化执行效率 （不过在写这本书的时候它还不能支持Python 3）。

还有一点要注意的是，线程不是专门用来优化性能的。 一个CPU依赖型程序可能会使用线程来管理一个图形用户界面、一个网络连接或其他服务。 这时候，GIL会产生一些问题，因为如果一个线程长期持有GIL的话会导致其他非CPU型线程一直等待。 事实上，一个写的不好的C语言扩展会导致这个问题更加严重， 尽管代码的计算部分会比之前运行的更快些。

说了这么多，现在想说的是我们有两种策略来解决GIL的缺点。 首先，如果你完全工作于Python环境中，你可以使用 `multiprocessing` 模块来创建一个进程池， 并像协同处理器一样的使用它。例如，假如你有如下的线程代码：

```
# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = some_work(args)
    ...
```

修改代码，使用进程池：

```
# Processing pool (see below for initiazation)
pool = None

# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Initiaze the pool
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
```

这个通过使用一个技巧利用进程池解决了GIL的问题。 当一个线程想要执行CPU密集型工作时，会将任务发给进程池。 然后进程池会在另外一个进程中启动一个单独的Python解释器来工作。 当线程等待结果的时候会释放GIL。 并且，由于计算任务在单独解释器中执行，那么就不会受限于GIL了。 在一个多核系统上面，你会发现这个技术可以让你很好的利用多CPU的优势。

另外一个解决GIL的策略是使用C扩展编程技术。 主要思想是将计算密集型任务转移给C，跟Python独立，在工作的时候在C代码中释放GIL。 这可以通过在C代码中插入下面这样的特殊宏来完成：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code
   ...
   Py_END_ALLOW_THREADS
   ...
}
```

如果你使用其他工具访问C语言，比如对于Cython的ctypes库，你不需要做任何事。 例如，ctypes在调用C时会自动释放GIL。

讨论¶

许多程序员在面对线程性能问题的时候，马上就会怪罪GIL，什么都是它的问题。 其实这样子太不厚道也太天真了点。 作为一个真实的例子，在多线程的网络编程中神秘的 `stalls` 可能是因为其他原因比如一个DNS查找延时，而跟GIL毫无关系。 最后你真的需要先去搞懂你的代码是否真的被GIL影响到。 同时还要明白GIL大部分都应该只关注CPU的处理而不是I/O.

如果你准备使用一个处理器池，注意的是这样做涉及到数据序列化和在不同Python解释器通信。 被执行的操作需要放在一个通过def语句定义的Python函数中，不能是lambda、闭包可调用实例等， 并且函数参数和返回值必须要兼容pickle。 同样，要执行的任务量必须足够大以弥补额外的通信开销。

另外一个难点是当混合使用线程和进程池的时候会让你很头疼。 如果你要同时使用两者，最好在程序启动时，创建任何线程之前先创建一个单例的进程池。 然后线程使用同样的进程池来进行它们的计算密集型工作。

C扩展最重要的特征是它们和Python解释器是保持独立的。 也就是说，如果你准备将Python中的任务分配到C中去执行， 你需要确保C代码的操作跟Python保持独立， 这就意味着不要使用Python数据结构以及不要调用Python的C API。 另外一个就是你要确保C扩展所做的工作是足够的，值得你这样做。 也就是说C扩展担负起了大量的计算任务，而不是少数几个计算。

这些解决GIL的方案并不能适用于所有问题。 例如，某些类型的应用程序如果被分解为多个进程处理的话并不能很好的工作， 也不能将它的部分代码改成C语言执行。 对于这些应用程序，你就要自己需求解决方案了（比如多进程访问共享内存区，多解析器运行于同一个进程等）。 或者，你还可以考虑下其他的解释器实现，比如PyPy。

了解更多关于在C扩展中释放GIL，请参考15.7和15.10小节。

第十二章：并发编程¶

对于并发编程, Python有多种长期支持的方法, 包括多线程, 调用子进程, 以及各种各样的关于生成器函数的技巧. 这一章将会给出并发编程各种方面的技巧, 包括通用的多线程技术以及并行计算的实现方法.

像经验丰富的程序员所知道的那样, 大家担心并发的程序有潜在的危险. 因此, 本章的主要目标之一是给出更加可信赖和易调试的代码.

Contents: