

УПРАВЛЕНИЕ ПОТОКАМИ

ЦЕЛЬ

Объяснить различия между блокировками Lock и RLock в Python и помочь определить, когда следует использовать каждую из них в многопоточных приложениях

В Python есть несколько способов запустить потоки.
Один из них – использовать модуль **threading**

Пример

```
...
from threading import Thread
import time

def parser(site_name):
    time.sleep(2)
    print(site_name)
    return site_name

if __name__ == '__main__':
    start_time = time.time()
    t1 = Thread(target=parser, args=("Site 1",))
    t2 = Thread(target=parser, args=("Site 2",))

    t1.start()
    t2.start()

    rs1 = t1.join()
    rs2 = t2.join()

    print(rs1, rs2)

    print("asdasd")

    print("--- %s seconds ---" % (time.time() - start_time))
```

В примере мы создаем функцию **parser**, которая будет выполняться в отдельном потоке. Затем мы создаем объект **Thread** из модуля **threading** и передаем ему нашу функцию в качестве целевого (**target**) аргумента. Далее, мы вызываем метод **start()** для запуска потока

В модуле **threading** в Python есть метод **join**, который позволяет главному потоку кода ожидать завершения выполнения других потоков

Когда мы создаем и запускаем поток с помощью класса **Thread**, как в предыдущем примере, мы можем вызвать метод **join** для этого потока

Пример

```
...
from threading import Thread
import time

class CustomThread(Thread):
    def __init__(self, site):
        Thread.__init__(self)
        self.site = site

    # function executed in a new thread
    def run(self):
        time.sleep(1)
        self.site = "End Process On Site " + self.site

if __name__ == '__main__':
    start_time = time.time()
    t1 = CustomThread("Site 1")
    t2 = CustomThread("Site 2")

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print(t1.site, t2.site)

    print("asdasd")

    print("--- %s seconds ---" % (time.time() - start_time))
```

Метод **join** блокирует выполнение главного потока до тех пор, пока указанный поток не завершит свою работу. Это означает, что строка кода "Программа завершила выполнение" не будет выполнена до того, как поток завершит свою работу

Метод **join** может принимать необязательный аргумент **timeout**, который указывает максимальное время ожидания завершения потока

Если время ожидания истекает, главный поток продолжит выполнение без ожидания завершения потока

Этот метод очень полезен, когда нужно выполнить определенные действия после завершения фоновых задач в потоках

Для обмена информацией между потоками можно использовать класс `Queue` из модуля `queue`

```
...
from threading import Thread
from queue import Queue
import time

class CustomThreadProducer(Thread):
    def __init__(self, site, queue):
        Thread.__init__(self)
        self.site = site
        self.queue = queue

    # function executed in a new thread
    def run(self):
        time.sleep(1)
        self.queue.put("End Process On Site " + self.site)

class CustomThreadConsumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    # function executed in a new thread
    def run(self):
        while True:
            print(self.queue.get())

if __name__ == '__main__':
    start_time = time.time()
    queue = Queue()
    t1 = CustomThreadProducer("Site 1", queue).start()
    t2 = CustomThreadProducer("Site 2", queue).start()

    q = CustomThreadConsumer(queue)
    q.start()
    print("asdasd")

    print("asdasd")

    print("---- %s seconds ----" % (time.time() - start_time))
```

В примере, мы:



Создали два потока, которые пишут информацию в очередь



Запустили слушателя этой очереди

В Python, модуль `threading` предоставляет класс `Lock`, который представляет примитив синхронизации в виде блокировки. **Lock** (блокировка) позволяет управлять доступом к общим ресурсам из нескольких потоков

Lock

работает по принципу "захватить" и "освободить"

Захватить

Когда поток хочет получить доступ к общему ресурсу, он "захватывает" блокировку

Если блокировка уже захвачена другим потоком, текущий поток будет ожидать, пока блокировка не будет освобождена

Освободить

После того, как текущий поток закончит работу с общим ресурсом, он "освобождает" блокировку, чтобы другие потоки могли ее захватить

Пример использования Lock

```
...
import threading

# Общий ресурс
counter = 0

# Создаем блокировку
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        # Захватываем блокировку
        lock.acquire()
        counter += 1
        # Освобождаем блокировку
        lock.release()
```

```
# Создаем и запускаем два потока
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)
thread1.start()
thread2.start()

# Ожидаем завершения обоих потоков
thread1.join()
thread2.join()

# Выводим значение счетчика
print("Значение счетчика:", counter) # Ожидаемый результат:
```

В этом примере у нас есть общий ресурс `counter`, который нужно инкрементировать в двух потоках. Мы создаем объект **Lock** с помощью `threading.Lock()`. В функции `increment` мы захватываем блокировку перед изменением `counter` и освобождаем ее после завершения операции. Таким образом, только один поток имеет доступ к ресурсу в данный момент времени, а другие потоки ждут, пока блокировка не освободится

Использование блокировок (**Lock**) помогает избежать проблемы с одновременным доступом к общим ресурсам из разных потоков. Это позволяет гарантировать, что только один поток работает с данными в конкретный момент времени, обеспечивая безопасность и корректность взаимодействия потоков

RLock рекурсивная блокировка

это расширение блокировки (Lock) в модуле `threading` в Python. В отличие от обычной блокировки, RLock позволяет потоку захватить блокировку несколько раз без блокировки самого себя. Это означает, что тот же самый поток может получить блокировку несколько раз, прежде чем должен будет ее освободить

Основным отличием RLock от Lock является его поддержка рекурсии

Когда поток захватывает RLock, он увеличивает счетчик блокировки. Каждый раз, когда поток снова захватывает RLock, этот счетчик увеличивается. Поток должен освободить RLock столько же раз, сколько он его захватил, чтобы другие потоки могли получить доступ к ресурсу

Пример использования RLock

```
...
import threading

# Общий ресурс
counter = 0

# Создаем рекурсивную блокировку
lock = threading.RLock()

def increment():
    global counter
    with lock:
        counter += 1
        # Вложенное захватывание RLock
        with lock:
            counter += 1

    # Создаем и запускаем два потока
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)
thread1.start()
thread2.start()

# Ожидаем завершения обоих потоков
thread1.join()
thread2.join()

# Выводим значение счетчика
print("Значение счетчика:", counter) # Ожидаемый результат: 4
```

В этом примере у нас есть общий ресурс `counter`, который мы инкрементируем с использованием рекурсивной блокировки `RLock`. В функции `increment` мы используем контекстный менеджер `with lock` для захвата и освобождения блокировки. Заметьте, что мы вызываем `with lock` дважды внутри `increment`, что позволяет одному потоку удерживать блокировку вложенным образом.

Использование `RLock` полезно, когда вам нужно предотвратить взаимоблокировку (**deadlock**), возникающую из-за повторного захвата блокировки тем же самым потоком. Рекурсивная блокировка `RLock` позволяет потокам безопасно взаимодействовать с общими ресурсами, даже если они вызывают вложенные блокировки.

Lock и RLock

это две разные реализации блокировок в Python для синхронизации доступа к общим ресурсам в многопоточных программных средах

Lock блокировка

это простая блокировка, доступная в модуле `threading` в Python. Когда поток захватывает Lock, он получает исключительное право доступа к общему ресурсу. Другие потоки, пытающиеся захватить эту блокировку, будут блокироваться до тех пор, пока она не будет освобождена. Это обеспечивает многопоточную безопасность, но не допускает повторное захватывание блокировки тем же самым потоком. Если поток попытается повторно захватить Lock, он заблокируется, что может привести к взаимоблокировке (deadlock)

RLock рекурсивная блокировка

это расширение Lock, доступное также в модуле `threading`. Основное отличие RLock от Lock заключается в его поддержке рекурсии. Когда поток захватывает RLock, он увеличивает счетчик блокировки. Поток может многократно захватывать RLock без блокировки самого себя. При каждом повторном захвате, счетчик увеличивается. Чтобы освободить RLock, поток должен его освободить столько же раз, сколько он его захватил. Это позволяет использовать рекурсию в коде, где один поток может многократно получать доступ к общим ресурсам

Использование **Lock** рекомендуется, когда нет необходимости в повторном захвате блокировки и есть потенциальный риск взаимоблокировки, а использование **RLock** имеет смысл, когда вам нужна поддержка рекурсии и потокам требуется возможность многократного захвата блокировки

Пример использования Lock и RLock

```
...
import threading

# Общий ресурс
counter_lock = threading.Lock()
counter = 0

def increment():
    global counter
    with counter_lock:
        counter += 1

def decrement():
    global counter
    with counter_lock:
        counter -= 1

def recursive_example(lock):
    with lock:
        print("Захват блокировки")
        recursive_example(lock) # Рекурсивный захват блокировки

# Создаем и запускаем два потока
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=decrement)
thread1.start()
thread2.start()

# Ожидаем завершения обоих потоков
thread1.join()
thread2.join()

print("Значение счетчика:", counter) # Ожидаемый результат: 0

# Создаем рекурсивную блокировку
recursive_lock = threading.RLock()

# Создаем и запускаем поток с рекурсивным захватом блокировки
recursive_thread = threading.Thread(target=recursive_example,
                                     args=(recursive_lock,))
recursive_thread.start()

# Ожидаем завершения потока
recursive_thread.join()
```

В этом примере у нас есть общий ресурс **counter**, к которому обращаются два потока с использованием обычной блокировки **Lock**. Мы используем контекстный менеджер **with counter_lock** для захвата и освобождения блокировки

Во второй части примера у нас есть функция **recursive_example**, которая демонстрирует рекурсивное использование **RLock**. Мы создаем рекурсивную блокировку **RLock**, и внутри функции вызываем **recursive_example**, рекурсивно с захватом блокировки. Это демонстрирует возможность многократного захвата блокировки с использованием **RLock**

ИТОГИ

- ✓ Блокировка **Lock** подходит для обеспечения безопасного доступа к общим ресурсам, когда не требуется рекурсивное захватывание блокировки одним потоком
- ✓ Блокировка **RLock** предоставляет возможность рекурсивного захвата блокировки одним потоком и может быть полезна, если требуется поддержка такой функциональности
- ✓ Выбор между **Lock** и **RLock** зависит от конкретных требований приложения к работе с общими ресурсами в многопоточной среде

