Synchronization Primitives

Although asyncio applications usually run as a single-threaded process, they are still built as concurrent applications. Each coroutine or task may execute in an unpredictable order, based on delays and interrupts from I/O and other external events. To support safe concurrency, asyncio includes implementations of some of the same low-level primitives found in the threading and multiprocessing modules.

Locks

A Lock can be used to guard access to a shared resource. Only the holder of the lock can use the resource. Multiple attempts to acquire the lock will block so that there is only one holder at a time.

```
# asyncio_lock.py

import asyncio
import functools


def unlock(lock):
    print('callback releasing lock')
    lock.release()


async def coro1(lock):
    print('coro1 waiting for the lock')
    async with lock:
        print('coro1 acquired lock')
    print('coro1 released lock')


async def coro2(lock):
    print('coro2 waiting for the lock')
    await lock.acquire()
    try:
        print('coro2 acquired lock')
    finally:
        print('coro2 released lock')
        lock.release()


async def main(loop):
    # Create and acquire a shared lock.
    lock = asyncio.Lock()
    print('acquiring the lock before starting coroutines')
    await lock.acquire()
    print('lock acquired: {}'.format(lock.locked()))

    # Schedule a callback to unlock the lock.
    loop.call_later(0.1, functools.partial(unlock, lock))

    # Run the coroutines that want to use the lock.
    print('waiting for coroutines')
    await asyncio.wait([coro1(lock), coro2(lock)]),


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

A lock's acquire() method can be invoked directly, using await, and calling the release() method when done as in coro2()

in this example. They also can be used as asynchronous context managers with the `with await` keywords, as in `coro1()`.

```
$ python3 asyncio_lock.py

acquiring the lock before starting coroutines
lock acquired: True
waiting for coroutines
coro2 waiting for the lock
coro1 waiting for the lock
callback releasing lock
coro2 acquired lock
coro2 released lock
coro1 acquired lock
coro1 released lock
```

Events

An `asyncio.Event` is based on `threading.Event`, and is used to allow multiple consumers to wait for something to happen without looking for a specific value to be associated with the notification.

```python
# asyncio_event.py

import asyncio
import functools


def set_event(event):
    print('setting event in callback')
    event.set()


async def coro1(event):
    print('coro1 waiting for event')
    await event.wait()
    print('coro1 triggered')


async def coro2(event):
    print('coro2 waiting for event')
    await event.wait()
    print('coro2 triggered')


async def main(loop):
    # Create a shared event
    event = asyncio.Event()
    print('event start state: {}'.format(event.is_set()))

    loop.call_later(
        0.1, functools.partial(set_event, event)
    )

    await asyncio.wait([coro1(event), coro2(event)])
    print('event end state: {}'.format(event.is_set()))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

As with the Lock, both `coro1()` and `coro2()` wait for the event to be set. The difference is that both can start as soon as the event state changes, and they do not need to acquire a unique hold on the event object.

```
$ python3 asyncio_event.py

event start state: False
coro2 waiting for event
```

```
coro1 waiting for event
setting event in callback
coro2 triggered
coro1 triggered
event end state: True
```

Conditions

A Condition works similarly to an Event except that rather than notifying all waiting coroutines the number of waiters awakened is controlled with an argument to `notify()`.

```python
# asyncio_condition.py

import asyncio


async def consumer(condition, n):
    async with condition:
        print('consumer {} is waiting'.format(n))
        await condition.wait()
        print('consumer {} triggered'.format(n))
    print('ending consumer {}'.format(n))


async def manipulate_condition(condition):
    print('starting manipulate_condition')

    # pause to let consumers start
    await asyncio.sleep(0.1)

    for i in range(1, 3):
        async with condition:
            print('notifying {} consumers'.format(i))
            condition.notify(n=i)
        await asyncio.sleep(0.1)

    async with condition:
        print('notifying remaining consumers')
        condition.notify_all()

    print('ending manipulate_condition')


async def main(loop):
    # Create a condition
    condition = asyncio.Condition()

    # Set up tasks watching the condition
    consumers = [
        consumer(condition, i)
        for i in range(5)
    ]

    # Schedule a task to manipulate the condition variable
    loop.create_task(manipulate_condition(condition))

    # Wait for the consumers to be done
    await asyncio.wait(consumers)


event_loop = asyncio.get_event_loop()
try:
    result = event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

This example starts five consumers of the `Condition`. Each uses the `wait()` method to wait for a notification that they can proceed. `manipulate_condition()` notifies one consumer, then two consumers, then all of the remaining consumers.

```
$ python3 asyncio_condition.py
```

```
$ python3 asyncio_condition.py

starting manipulate_condition
consumer 3 is waiting
consumer 0 is waiting
consumer 4 is waiting
consumer 1 is waiting
consumer 2 is waiting
notifying 1 consumers
consumer 3 triggered
ending consumer 3
notifying 2 consumers
consumer 0 triggered
ending consumer 0
consumer 4 triggered
ending consumer 4
notifying remaining consumers
ending manipulate_condition
consumer 1 triggered
ending consumer 1
consumer 2 triggered
ending consumer 2
```

Queues

An `asyncio.Queue` provides a first-in, first-out data structure for coroutines like a `queue.Queue` does for threads or a `multiprocessing.Queue` does for processes.

```python
# asyncio_queue.py

import asyncio


async def consumer(n, q):
    print('consumer {}: starting'.format(n))
    while True:
        print('consumer {}: waiting for item'.format(n))
        item = await q.get()
        print('consumer {}: has item {}'.format(n, item))
        if item is None:
            # None is the signal to stop.
            q.task_done()
            break
        else:
            await asyncio.sleep(0.01 * item)
            q.task_done()
    print('consumer {}: ending'.format(n))


async def producer(q, num_workers):
    print('producer: starting')
    # Add some numbers to the queue to simulate jobs
    for i in range(num_workers * 3):
        await q.put(i)
        print('producer: added task {} to the queue'.format(i))
    # Add None entries in the queue
    # to signal the consumers to exit
    print('producer: adding stop signals to the queue')
    for i in range(num_workers):
        await q.put(None)
    print('producer: waiting for queue to empty')
    await q.join()
    print('producer: ending')


async def main(loop, num_consumers):
    # Create the queue with a fixed size so the producer
    # will block until the consumers pull some items out.
    q = asyncio.Queue(maxsize=num_consumers)

    # Scheduled the consumer tasks.
```

```
    # Scheduled the consumer tasks.
    consumers = [
        loop.create_task(consumer(i, q))
        for i in range(num_consumers)
    ]

    # Schedule the producer task.
    prod = loop.create_task(producer(q, num_consumers))

    # Wait for all of the coroutines to finish.
    await asyncio.wait(consumers + [prod])


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop, 2))
finally:
    event_loop.close()
```

Adding items with `put()` or removing items with `get()` are both asynchronous operations, since the queue size might be fixed (blocking an addition) or the queue might be empty (blocking a call to fetch an item).

```
$ python3 asyncio_queue.py

consumer 0: starting
consumer 0: waiting for item
consumer 1: starting
consumer 1: waiting for item
producer: starting
producer: added task 0 to the queue
producer: added task 1 to the queue
consumer 0: has item 0
consumer 1: has item 1
producer: added task 2 to the queue
producer: added task 3 to the queue
consumer 0: waiting for item
consumer 0: has item 2
producer: added task 4 to the queue
consumer 1: waiting for item
consumer 1: has item 3
producer: added task 5 to the queue
producer: adding stop signals to the queue
consumer 0: waiting for item
consumer 0: has item 4
consumer 1: waiting for item
consumer 1: has item 5
producer: waiting for queue to empty
consumer 0: waiting for item
consumer 0: has item None
consumer 0: ending
consumer 1: waiting for item
consumer 1: has item None
consumer 1: ending
producer: ending
```

Composing Coroutines with Control Structures   Asynchronous I/O with Protocol Class Abstractions

Quick Links

Locks
Events
Conditions
Queues

This page was last updated 2018-12-09.

Navigation

Composing Coroutines with Control Structures
Asynchronous I/O with Protocol Class Abstractions

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?