Executing Tasks Concurrently

Tasks are one of the primary ways to interact with the event loop. Tasks wrap coroutines and track when they are complete. Tasks are subclasses of `Future`, so other coroutines can wait for them and each has a result that can be retrieved after the task completes.

Starting a Task

To start a task, use `create_task()` to create a Task instance. The resulting task will run as part of the concurrent operations managed by the event loop as long as the loop is running and the coroutine does not return.

```python
# asyncio_create_task.py

import asyncio


async def task_func():
    print('in task_func')
    return 'the result'


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    print('waiting for {!r}'.format(task))
    return_value = await task
    print('task completed {!r}'.format(task))
    print('return value: {!r}'.format(return_value))


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

This example waits for the task to return a result before the `main()` function exits.

```
$ python3 asyncio_create_task.py

creating task
waiting for <Task pending coro=<task_func() running at
asyncio_create_task.py:12>>
in task_func
task completed <Task finished coro=<task_func() done, defined at
asyncio_create_task.py:12> result='the result'>
return value: 'the result'
```

Canceling a Task

By retaining the Task object returned from `create_task()`, it is possible to cancel the operation of the task before it completes.

```python
# asyncio_cancel_task.py

import asyncio


async def task_func():
    print('in task_func')
    return 'the result'


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())

    print('canceling task')
    task.cancel()

    print('canceled task {!r}'.format(task))
    try:
        await task
    except asyncio.CancelledError:
        print('caught error from canceled task')
    else:
        print('task result: {!r}'.format(task.result()))


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

This example creates and then cancels a task before starting the event loop. The result is a `CancelledError` exception from `run_until_complete()`.

```
$ python3 asyncio_cancel_task.py

creating task
canceling task
canceled task <Task cancelling coro=<task_func() running at
asyncio_cancel_task.py:12>>
caught error from canceled task
```

If a task is canceled while it is waiting for another concurrent operation, the task is notified of its cancellation by having a `CancelledError` exception raised at the point where it is waiting.

```
# asyncio_cancel_task2.py

import asyncio


async def task_func():
    print('in task_func, sleeping')
    try:
        await asyncio.sleep(1)
    except asyncio.CancelledError:
        print('task_func was canceled')
        raise
    return 'the result'


def task_canceller(t):
    print('in task_canceller')
    t.cancel()
    print('canceled the task')


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    loop.call_soon(task_canceller, task)
    try:
        await task
    except asyncio.CancelledError:
        print('main() also sees task as canceled')


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

Catching the exception provides an opportunity to clean up work already done, if necessary.

```
$ python3 asyncio_cancel_task2.py

creating task
in task_func, sleeping
in task_canceller
canceled the task
task_func was canceled
main() also sees task as canceled
```

Creating Tasks from Coroutines

The ensure_future() function returns a Task tied to the execution of a coroutine. That Task instance can then be passed to other code, which can wait for it without knowing how the original coroutine was constructed or called.

```
# asyncio_ensure_future.py

import asyncio


async def wrapped():
    print('wrapped')
    return 'result'


async def inner(task):
    print('inner: starting')
    print('inner: waiting for {!r}'.format(task))
    result = await task
    print('inner: task returned {!r}'.format(result))


async def starter():
    print('starter: creating task')
    task = asyncio.ensure_future(wrapped())
    print('starter: waiting for inner')
    await inner(task)
    print('starter: inner returned')


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    result = event_loop.run_until_complete(starter())
finally:
    event_loop.close()
```

Note that the coroutine given to ensure_future() is not started until something uses await to allow it to be executed.

```
$ python3 asyncio_ensure_future.py

entering event loop
starter: creating task
starter: waiting for inner
inner: starting
inner: waiting for <Task pending coro=<wrapped() running at
asyncio_ensure_future.py:12>>
wrapped
inner: task returned 'result'
starter: inner returned
```

This page was last updated 2016-12-18.

Navigation

Producing Results Asynchronously
Composing Coroutines with Control Structures