Future & Tasks

This lesson discusses futures and tasks that can get scheduled on the event loop.

Futures & Tasks

Future

`Future` represents a computation that is either in progress or will get scheduled in the future. It is a special low-level awaitable object that represents an eventual result of an asynchronous operation. Don't confuse **`threading.Future`** and **`asyncio.Future`**. The former is part of the **`threading`** module and doesn't have an **`__iter__()`** method defined on it. **`asyncio.Future`** is an awaitable and can be used with the **`yield from`** statement. In general you shouldn't need to deal with futures directly. They are usually exposed by libraries or asyncio APIs. For instructional purposes we'll show an example that creates a future that is awaited by a coroutine. Study the snippet below:

```
import asyncio
from asyncio import Future


async def bar(future):
    print("bar will sleep for 3 seconds")
    await asyncio.sleep(3)
    print("bar resolving the future")
    future.done()
    future.set_result("future is resolved")


async def foo(future):
    print("foo will await the future")
    await future
    print("foo finds the future resolved")


async def main():
    future = Future()
    results = await asyncio.gather(foo(future), bar(future))


if __name__ == "__main__":
    asyncio.run(main())
    print("main exiting")
```

Both the coroutines are passed a future. The `foo()` coroutine awaits for the future to get resolved, while the `bar()` coroutine resolves the future after three seconds.

```
import asyncio
from asyncio import Future


async def bar(future):
    print("bar will sleep for 3 seconds")
    await asyncio.sleep(3)
    print("bar resolving the future")
    future.done()
    future.set_result("future is resolved")


async def foo(future):
    print("foo will await the future")
    await future
    print("foo finds the future resolved")


async def main():
    future = Future()
    results = await asyncio.gather(foo(future), bar(future))


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    print("main exiting")

    # syntax for Python 3.7
    # asyncio.run(main())
    # print("main exiting")
```

Tasks

Tasks are like futures in fact **`Task`** is a subclass of **`Future`**. Tasks can be created using the following methods:

- **`asyncio.create_task()`** introduced in Python 3.7 and preferred way of creating tasks. The method accepts coroutines and wraps

them as tasks.

- **`loop.create_task()`** only accepts coroutines.
- **`asyncio.ensure_future()`** accepts Futures, coroutines and any awaitable objects.

Tasks wrap coroutines and run them in event loops. If a coroutine awaits on a **`Future`**, the **`Task`** suspends the execution of the coroutine and waits for the **`Future`** to complete. When the **`Future`** is done, the execution of the wrapped coroutine resumes. Event loops use cooperative scheduling, the event loop runs one **`Task`** at a time. While a **`Task`** awaits for the completion of a **`Future`**, the event loop runs other tasks, callbacks, or performs IO operations. Tasks can also be cancelled.

We rewrite the future example using tasks as follows. The code is compatible with Python 3.7+, however, the code that appears in the code widget works with Python 3.5 since the widget runs version 3.5.

```
import asyncio
from asyncio import Future


async def bar(future):
    print("bar will sleep for 3 seconds")
    await asyncio.sleep(3)
    print("bar resolving the future")
    future.done()
    future.set_result("future is resolved")


async def foo(future):
    print("foo will await the future")
    await future
    print("foo finds the future resolved")


async def main():
    future = Future()

    loop = asyncio.get_event_loop()
    t1 = loop.create_task(bar(future))
    t2 = loop.create_task(foo(future))

    await t2, t1


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    print("main exiting")
```

```
import asyncio
from asyncio import Future


async def bar(future):
    print("bar will sleep for 3 seconds")
    await asyncio.sleep(3)
    print("bar resolving the future")
    future.done()
    future.set_result("future is resolved")


async def foo(future):
    print("foo will await the future")
    await future
    print("foo finds the future resolved")


async def main():
    future = Future()

    loop = asyncio.get_event_loop()
    t1 = loop.create_task(bar(future))
    t2 = loop.create_task(foo(future))

    await t2, t1


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    print("main exiting")
```

Back

Chaining Coroutines

Next

Web Crawler Example

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/future-tasks__asyncio__python-concurrency-for-senior-engineering-interviews)