Common Mistakes Using Python3 asyncio

2017-07-31

Preface

Python3 `asyncio` is a powerful asynchronous library. However, the complexity results in a very steep learning curve. Compared to C# `async` / `await`, the interfaces of Python3 `asyncio` is verbose and difficult to use. And the document is somewhat difficult to understand. (Even Guido admited the document is not clear!) Here I summerize some of the common mistakes when using `asyncio`.

RuntimeWarning: coroutine `foo` was never awaited

This runtime warning can happen in many scenarios, but the cause are same: A coroutine object is created by the invocation of an `async` function, but is never inserted into an `EventLoop`.

Consider following `async` function `foo()`:

```
async def foo():
    # a long async operation
    # no value is returned
```

If you want to call `foo()` as an asynchronous task, and doesn't care about the result:

```
prepare_for_foo()
foo()                                # RuntimeWarning: coroutine foo was never a
remaining_work_not_depends_on_foo()
```

This is because invoking `foo()` doesn't actually runs the function `foo()`, but created a "coroutine object" instead. This "coroutine object" will be executed when current `EventLoop` gets a chance: `awaited` / `yield from` is called or all previous tasks are finished.

To execute an asynchronous task without `await`, use `loop.create_task()` with `loop.run_until_complete()`:

```
prepare_for_foo()
task = loop.create_task(foo())
```

```
remaining_work_not_depends_on_foo()
loop.run_until_complete(task)
```

If the coroutine object is created and inserted into an `EventLoop`, but was never finished, the next warning will appear.

Task was destroyed but it is pending!

The cause of this problem is that the `EventLoop` is closed right after canceling pending tasks. Because the `Task.cancel()` "arranges for a `CancelledError` to be thrown into the wrapped coroutine on the next cycle through the event loop", and "*the coroutine then has a chance to clean up or even deny the request using try/except/finally.*"

To correctly cancel all tasks and close `EventLoop`, the `EventLoop` should be given the last chance to run all the canceled, but unfinished tasks.

For example, this is the code to cancel all the tasks:

```
def cancel_tasks():
    tasks = Task.all_tasks()              # get all task in current loop
    for t in tasks:
        t.cancel()

cancel_tasks()
loop.stop()
```

Below code correctly handle task canceling and clean up. It starts the `EventLoop` by calling `loop.run_forever()`, and cleans up tasks after receiving `loop.stop()`:

```
try:
    loop.run_forever()                # run_forever() returns after calling loo
    tasks = Task.all_tasks()
    for t in [t for t in tasks if not (t.done() or t.cancelled())]:
        loop.run_until_complete(t)    # give canceled tasks the last chance to
finally:
    loop.close()
```

`Task` / `Future` is awaited in a different `EventLoop` than it is created

This error is especially surprising to people who are familiar with C# `async` / `await`. It is because most of `asyncio` is not thread-safe, nor is `asyncio.Future` or `asyncio.Task`.

Also don't confuse `asyncio.Future` with `concurrent.futures.Future` because they are not compatible (at least until Python 3.6): the latter is thread-safe while the former is not.

In order to await an `asyncio.Future` in a different thread, `asyncio.Future` can be wrapped in a `concurrent.Future`:

```
def wrap_future(asyncio_future):
    def done_callback(af, cf):
        try:
            cf.set_result(af.result())
        except Exception as e:
            cf.set_exception(e)

    concur_future = concurrent.Future()
    asyncio_future.add_done_callback(lambda f: done_callback(f, cf=concur_futur
    return concur_future
```

`asyncio.Task` is a subclass of `asyncio.Future`, so above code will also work.

Home Top