《源码探秘 CPython》90. 剖析 Python 的协程

原创 古明地觉 古明地觉的编程教室 2022-05-15 09:00 发表于北京

楔子

了解完协程的诞生背景以及基本概念之后，我们来探究一下 Python 的协程是如何实现的。首先 Python 的协程和生成器之间有着非常紧密的联系，因此在看协程之前，强烈推荐你先看本系列的第 65 和 第 66 篇文章。

另外，关于协程的基础知识，以及 asyncio 模块的一些基本用法这里就不再赘述了，我们重点看协程的实现原理。

协程的底层结构

使用 async def 定义出来的函数叫协程函数，当然协程函数本质上也是一个函数。记得我们说过，函数的 __code__ 里面有一个 co_flags 字段，它可以用来判断函数的种类。

```
def func():
    pass

async def coro_func():
    pass

print(func.__class__)  # <class 'function'>
print(coro_func.__class__)  # <class 'function'>

# 两者都是 <class 'function'> 类型
print(func.__code__.co_flags & 0x80)  # 0
print(coro_func.__code__.co_flags & 0x80)  # 128

# 但如果是协程函数，那么 co_flags & 0x80 为真
```

普通函数调用之后，会将内部的字节码全部执行完毕；协程函数调用之后，不会执行内部的字节码，而是返回一个协程对象，简称协程，协程需要扔到事件循环里面执行。

那么下面看看协程的底层结构，开头说协程和生成器有着很紧密的联系，看完之后你就知道这个联系有多紧密了。协程对应的结构体定义在 genobject.h 中，咦，这不是生成器相关的头文件吗？

```
#define _PyGenObject_HEAD(prefix)
    PyObject_HEAD
    /* Note: gi_frame can be NULL if the generator
    struct _frame *prefix##_frame;
    /* True if generator is being executed. */
    char prefix##_running;
    /* The code object backing the generator */
    PyObject *prefix##_code;
    /* List of weak reference. */
    PyObject *prefix##_weakreflist;
    /* Name of the generator. */
```

```
    PyObject *prefix##_name;
    /* Qualified name of the generator. */
    PyObject *prefix##_qualname;
    _PyErr_StackItem prefix##_exc_state;

typedef struct {
    /* The gi_ prefix is intended to remind of ge
    _PyGenObject_HEAD(gi)
} PyGenObject;
```


红色框框内的部分是生成器的底层结构，不要眨眼，再来看看协程的底层结构。

```
typedef struct {
    _PyGenObject_HEAD(cr)
    PyObject *cr_origin;
} PyCoroObject;
```


所以我们看到所谓协程，本质上还是基于生成器实现的，只是字段名不一样。比如生成器有 gi_frame, gi_running, gi_code 等字段，协程则是 cr_frame, cr_running, cr_code。

```
async def coro_func():
    pass

coro = coro_func()
coro.
    close(self)                    Coroutine
    send(self, value)              Coroutine
    __doc__                        Coroutine
    __class__                         object
    __module__                     Coroutine
    cr_await                       Coroutine
    cr_code                        Coroutine
    cr_frame                       Coroutine
    cr_running                     Coroutine
    throw(self, typ, val, tb)      Coroutine
    __sizeof__(self)                  object
Press Enter to insert, Tab to replace  Next Tip
```


而且协程也有 send、close、throw 等方法，我们来对比一下生成器和协程。

```python
def gen_func():
    yield
    return "gen result"

async def coro_func():
    return f"coroutine result"


# 先来看看生成器
gen = gen_func()
gen.__next__()
try:
    gen.__next__()
except StopIteration as e:
    print(e.value)  # gen result
# 当生成器 return 时，会抛出一个 StopIteration
# 并将返回值设置在里面，因此 return 只是一个语法糖
# 对于协程亦是如此
```

```
coro = coro_func()
try:
    coro.__await__().__next__()
except StopIteration as e:
    print(e.value)  # coroutine result
```

生成器就不说了，直接看协程。首先基于 async def 定义的协程函数，内部不可以出现 yield from，否则会出现语法错误，这是一个编译阶段就能检测出来的错误。但是出现 yield 是可以的，只不过此时就不叫协程函数了，而是叫异步生成器函数。

而对于协程而言，只要运行一次 __next__，那么就会将内部的逻辑全部执行完。但需要注意，协程本身是没有 __next__ 方法的，它需要先调用 __await__。那么问题来了，这个 __await__ 又是什么呢？下面来解释一下。

我们知道，如果想在一个协程内部驱动另一个协程执行，那么可以使用 await 关键字。

```
async def coro_func():
    return f"coroutine result"

async def main():
    result = await coro_func()
    return f"来自 coro_func 的返回值: {result}"

try:
    main().__await__().__next__()
except StopIteration as e:
    print(e)
# 来自 coro_func 的返回值: coroutine result
```

await 后面可以跟协程、asyncio.Task 对象、asyncio.Future 对象，而 await obj 本质上会调用 obj 的 __await__ 方法，然后通过 __next__ 驱动执行。

为了更好地理解 await，我们对比一下 yield from，之前说过 yield from 可以用来实现委托生成器。

```
def gen_func():
    yield 123
    return "result"

def middle():
    res = yield (yield from gen_func())
    print(res)

# 我们说委托生成器有一个重要的作用
# 它会在子生成器和调用方之间建立一个双向通道
g = middle()
# 此时是调用方和子生成器之间直接通信
print(g.__next__())  # 123
# 显然此时要抛异常了，那么委托生成器要负责兜底
# 会拿到子生成器的返回值，在自身内部寻找下一个 yield
print(g.__next__())  # result
```

而对于 await 也是同样的道理，它的作用和 yield from 类似。

```
async def coro_func():
```

```
    return f"coroutine result"

async def main():
    result = await coro_func()
    return f"来自 coro_func 的返回值: {result}"

# main() 里面 await 一个子协程
# 此时 await 同样会建立一个双向通道
# 子协程会和调用方直接通信
# 另外这里的调用方可以是我们手动调用，也可以是事件循环
try:
    # 子协程要抛异常了，那么 main() 要兜底
    # await 会捕获异常，然后将返回值取出来
    main().__await__().__next__()
except StopIteration as e:
    print(e.value)
# 来自 coro_func 的返回值: coroutine result
```

我们在协程里面，可以 await 另一个协程，然后驱动它执行。等到执行完毕之后，再拿到它的返回值。或者我们还有一种写法：

```
async def coro_func():
    return f"coroutine result"

async def main():
    try:
        coro_func().__await__().__next__()
    except StopIteration as e:
        return f"来自 coro_func 的返回值: {e.value}"

try:
    main().__await__().__next__()
except StopIteration as e:
    print(e.value)
# 来自 coro_func 的返回值: coroutine result
```

此时两者的做法是等价的，但很明显使用 await 要方便的多。

另外，我们上面运行协程的时候并没有依赖事件循环，原因是协程本质上就是个生成器，即使不依赖事件循环也可以运行。当然啦，在工作中肯定还是要交给事件循环进行调度的。

接下来，我们通过字节码来进一步观察这背后的细节。

```
s = """
async def coro_func():
    return f"coroutine result"

async def main():
    result = await coro_func()
    return f"来自 coro_func 的返回值: {result}"
"""
if __name__ == '__main__':
    import dis
    dis.dis(compile(s, "<file>", "exec"))
```

先来看看模块对应的字节码：

```
# 构造两个协程函数，显然这些字节码已经不需要说了
 0 LOAD_CONST               0 (<code object coro_func at 0x00...>)
 2 LOAD_CONST               1 ('coro_func')
 4 MAKE_FUNCTION            0
 6 STORE_NAME               0 (coro_func)

 8 LOAD_CONST               2 (<code object main at 0x000...>)
10 LOAD_CONST               3 ('main')
```

```
12 MAKE_FUNCTION            0
14 STORE_NAME               1 (main)
16 LOAD_CONST               4 (None)
18 RETURN_VALUE
```


模块对应的字节码没什么好说的，我们重点看一下函数的字节码。

```
Disassembly of <code object coro_func at 0x0000...>:
 0 LOAD_CONST               1 ('coroutine result')
 2 RETURN_VALUE

Disassembly of <code object main at 0x000...>:
 # 加载协程函数
 0 LOAD_GLOBAL              0 (coro_func)
 # 调用
 2 CALL_FUNCTION            0
 # 全新的指令
 4 GET_AWAITABLE
 # 加载一个 None 进来
 6 LOAD_CONST               0 (None)
 # 注意这里出现了 YIELD_FROM
 8 YIELD_FROM
 # 将 coro_func 的返回值交给 result 保存
10 STORE_FAST               0 (result)

 # 加载字符串常量
12 LOAD_CONST               1 ('来自 coro_func 的返回值: ')
 # 加载局部变量
14 LOAD_FAST                0 (result)
 # f-string, 构建字符串
16 FORMAT_VALUE             0
18 BUILD_STRING             2
20 RETURN_VALUE
```


以上是两个函数的字节码，coro_func 也没什么可说的，重点是 main。

里面出现了一个指令 GET_AWAITABLE，它所做的事情就是调用协程对象的 __await__ 方法。

```
case TARGET(GET_AWAITABLE): {
    PREDICTED(GET_AWAITABLE);
    //上一步的 CALL_FUNCTION 指令会构建一个协程
    //并且将协程压入到栈顶, 也就是这里的iterable
    PyObject *iterable = TOP();
    //调用协程的 __await__
    PyObject *iter = _PyCoro_GetAwaitableIter(iterable);

    //...
    //将 iter 设置为新的栈顶元素
    SET_TOP(iter); /* Even if it's NULL */

    if (iter == NULL) {
        goto error;
    }

    PREDICT(LOAD_CONST);
    DISPATCH();
}
```

所以重点是 _PyCoro_GetAwaitableIter 这个函数，我们看看协程的 __await__ 究竟返回了个啥？

```
PyObject *
_PyCoro_GetAwaitableIter(PyObject *o)
{
    unaryfunc getter = NULL;
```

```
    PyTypeObject *ot;
    //如果是一个使用 async def 定义的协程
    //或者是一个被 @asyncio.coroutine 装饰的协程(不推荐)
    //那么直接返回
    if (PyCoro_CheckExact(o) || gen_is_coroutine(o)) {
        /* 'o' is a coroutine. */
        Py_INCREF(o);
        return o;
    }

    //如果不是一个协程，那么它必须实现 __await__
    //对应 tp_as_async 的 am_await 成员
    ot = Py_TYPE(o);
    if (ot->tp_as_async != NULL) {
        getter = ot->tp_as_async->am_await;
    }
    //如果实现了 __await__
    if (getter != NULL) {
        //那么进行调用
        PyObject *res = (*getter)(o);
        if (res != NULL) {
            //如果一个对象不是协程，但又实现了 __await__
            //那么 __await__ 必须返回迭代器
            if (PyCoro_CheckExact(res) || gen_is_coroutine(res)) {
                /* __await__ must return an *iterator*, not
                   a coroutine or another awaitable (see PEP 492) */
                PyErr_SetString(PyExc_TypeError,
                                "__await__() returned a coroutine");
                Py_CLEAR(res);
            } else if (!PyIter_Check(res)) {
                PyErr_Format(PyExc_TypeError,
                             "__await__() returned non-iterator "
                             "of type '%.100s'",
                             Py_TYPE(res)->tp_name);
                Py_CLEAR(res);
            }
        }
        return res;
    }
    //否则就说明，该对象不可以被 await
    PyErr_Format(PyExc_TypeError,
                 "object %.100s can't be used in 'await' expression",
                 ot->tp_name);
    return NULL;
}
```

关于 __await__ 必须返回迭代器，我们举个栗子：

```
class A:

    def __await__(self):
        return coro_func().__await__()

async def coro_func():
    return f"coroutine result"

async def main():
    # A() 不是一个协程，那么它必须实现 __await__
    # 并且 __await__ 里面必须返回一个迭代器
    # 只有这样，才能调用 __next__ 方法
    result = await A()
    return f"来自 A().__await__ 的返回值: {result}"


```

```python
try:
    main().__await__().__next__()
except StopIteration as e:
    print(e.value)
# 来自 A().__await__ 的返回值: coroutine result
```

然后重点来了，后面又调用了 YIELD_FROM。因为 await obj 实际上分为两步，第一步是调用 obj 的 __await__ 返回一个迭代器，由指令 `GET_AWAITABLE` 完成；第二步是通过 __next__ 驱动执行，由指令 `YIELD_FROM` 完成。

也就是说，await obj.__await__().__next__() 再加一个 StopIteration异常的捕获逻辑，等价于 await obj，因为 YIELD_FROM 会捕获子协程 raise 的 StopIteration。

因此所谓的协程无非就是基于生成器进行的一个封装罢了，只是生成器既可以用作生成器本身，也可以用作协程，那么这就会出现混乱。于是 Python 在 3.5 的时候引入了 async 和 await 两个关键字，专门用于协程，但我们知道它们本质上还是基于生成器实现的即可。

通过事件循环运行协程

虽然我们可以直接驱动协程执行，但这不是一个好的方式，协程应该放到事件循环中，由事件循环驱动执行。

```python
import asyncio

async def coro_func(n):
    await asyncio.sleep(n)
    print(f"我睡了 {n} 秒")
    return f"coroutine result"

async def main():
    # 基于协程创建 asyncio.Task 对象
    task1 = asyncio.create_task(coro_func(3))
    task2 = asyncio.create_task(coro_func(1))

    await task1


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
我睡了 3 秒
"""
```

这里有一个比较神奇的地方，我们明明只 await task1，但是 task2 居然也被执行了，这是什么情况。原因就在于事件循环的运行单元是 asyncio.Task 对象，当我们调用 create_task 创建 Task 对象的时候，这个 Task 对象就已经被加入到事件循环中了。

所以此时事件循环里面有 task1 和 task2 两个任务，当我们 await task1 时，事件循环就会驱动 task1 执行。而一旦事件循环启动，那么不单单会执行 task1，而是会执行所有注册进来的任务。

所以先打印了 "我睡了 1 秒"，因为该协程 sleep 的时间更短，尽管它后执行，说明在 task1 的 sleep 时发生了协程切换。

```python
import asyncio

```

```
async def coro_func(n):
    await asyncio.sleep(n)
    print(f"我睡了 {n} 秒")
    return f"coroutine result"

async def main():
    task1 = asyncio.create_task(coro_func(3))
    task2 = asyncio.create_task(coro_func(1))
    task3 = coro_func(1)

    await task1


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
我睡了 3 秒
"""
```

还是之前的代码，但此时 task3 就没有执行，原因是它没有注册到事件循环里面。而且此时还抛出了警告，告诉我们没有 await。因为协程一旦创建了，就肯定要执行，而执行有两种方式，一种是封装成 Task 对象交给事件循环执行，另一种是直接 await。那么这两种方式有什么区别呢？

```
import asyncio
import time

async def coro_func(n):
    await asyncio.sleep(n)
    print(f"我睡了 {n} 秒")

async def main1():
    task1 = asyncio.create_task(coro_func(3))
    task2 = asyncio.create_task(coro_func(3))

    await task1
    await task2

async def main2():
    await coro_func(3)
    await coro_func(3)


loop = asyncio.get_event_loop()
start = time.perf_counter()
loop.run_until_complete(main1())
print(time.perf_counter() - start)
"""
我睡了 3 秒
我睡了 3 秒
3.0026131
"""

start = time.perf_counter()
loop.run_until_complete(main2())
print(time.perf_counter() - start)
"""
我睡了 3 秒
我睡了 3 秒
6.0027813000000005
"""
```

可以看到，执行 main1 花了 3 秒钟，但是执行 main2 花了 6 秒钟。

对于 main1 来讲，里面的协程在被包装成任务的时候，就已经注册到事件循环里面去了。驱动任何一个任务执行，均会使得事件循环里的所有任务都被执行。即使里面只有 await task1，task2 同样会被执行，只不过加上 await task2 会使得逻辑必须在 task2 完成之后才能往下走。

对于 main2 来讲，await 后面不是 Task 对象，而是一个协程，所以此时不会进入事件循环。而是我们之前说的，像驱动生成器一样，但此时无法实现切换，两个协程串行执行。

所以实际工作中，我们需要通过事件循环来驱动协程执行，并且好的习惯是不要直接 await 一个协程，而是要把协程封装成 Task 对象之后再 await。在 Python 里面一个协程就是调用一个原生可以挂起的函数，任务则是对协程的进一步封装，里面包含了协程在执行时的各种状态。

而正是基于 Task 对象，我们才能够实现协程间的切换，因为它维护着协程的执行状态。而除了 Task 对象，还有一个 Future 对象，负责保存返回值。当然，由于 Task 是 Future 的子类，所以 Task 对象包含了 Future 对象的所有功能。

Task 和 Future 都定义在 _asynciomodule.c 中。

```
#define FutureObj_HEAD(prefix)
    PyObject_HEAD
    PyObject *prefix##_loop;
    PyObject *prefix##_callback0;
    PyObject *prefix##_context0;
    PyObject *prefix##_callbacks;
    PyObject *prefix##_exception;
    PyObject *prefix##_result;
    PyObject *prefix##_source_tb;
    fut_state prefix##_state;
    int prefix##_log_tb;
    int prefix##_blocking;
    PyObject *dict;
    PyObject *prefix##_weakreflist;

typedef struct {
    FutureObj_HEAD(fut)
} FutureObj;

typedef struct {
    FutureObj_HEAD(task)
    PyObject *task_fut_waiter;
    PyObject *task_coro;
    PyObject *task_name;
    PyObject *task_context;
    int task_must_cancel;
    int task_log_destroy_pending;
} TaskObj;
```


asyncio 模块依赖于 _asyncio，这是一个内嵌在解释器里的模块，感兴趣的话可以去看一下。

小结

这里我们就介绍了协程的实现原理，它和生成器具有高度的相似性。当然了，我们不仅要理解协

程，还要知道如何使用 asyncio 这个协程库，关于 asyncio 这里就不多说了，可以参考官方文档。

至于 _asynciomodule.c 有兴趣的话可以读一读，因为里面的内容读起来难度还是蛮大的。当然，如果能从基本概念上理解 asyncio、并且会用，也已经足够了。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》91. 可执行文件的内存模型，变量的值是放在栈上还是放在堆上

下一篇 ›
《源码探秘 CPython》89. 为什么要有协程，协程是如何实现的?