如何精确控制 asyncio 中并发运行的多个任务

原创 古明地觉 古明地觉的编程教室 2023-09-21 08:30 发表于北京

之前我们了解了如何创建多个任务来并发运行程序，方式是通过 asyncio.create_task 将协程包装成任务，如下所示：

```
import asyncio, time

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

    await task1
    await task2
    await task3

start = time.perf_counter()
asyncio.run(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
总耗时: 3.003109625
"""
```

但这种代码编写方式只适用于简单情况，如果在同时发出数百、数千甚至更多 Web 请求的情况下，这种编写方式将变得冗长且混乱。所以 asyncio 提供了许多便利的函数，支持我们一次性等待多个任务。

等待一组任务全部完成

一个被广泛用于等待一组任务的方式是使用 asyncio.gather，这个函数接收一系列的可等待对象，允许我们在一行代码中同时运行它们。如果传入的 awaitable 对象是协程，gather 函数会自动将其包装成任务，以确保它们可以同时运行。这意味着不必像之前那样，用 asyncio.create_task 单独包装，

但即便如此，还是建议手动包装一下。

asyncio.gather 同样返回一个 awaitable 对象，在 await 表达式中使用它时，它将暂停，直到传递给它的所有 awaitable 对象都完成为止。一旦所有任务都完成，asyncio.gather 将返回这些任务的结果所组成的列表。

```
import asyncio
import time
from aiohttp import ClientSession

async def fetch_status(session: ClientSession, url: str):
    async with session.get(url) as resp:
        return resp.status

async def main():
    async with ClientSession() as session:
        # 注意：requests 里面是 100 个协程
        # 传递给 asyncio.gather 之后会自动被包装成任务
        requests = [fetch_status(session, "http://www.baidu.com")
                    for _ in range(100)]
        # 并发运行 100 个任务，并等待这些任务全部完成
        # 相比写 for 循环再单独 await，这种方式就简便多了
        status_codes = await asyncio.gather(*requests)
        print(f"{len(status_codes)} 个任务已全部完成")

start = time.perf_counter()
asyncio.run(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
100 个任务已全部完成
总耗时: 0.552532458
"""
```

完成 100 个请求只需要 0.55 秒钟，由于网络问题，测试的结果可能不准确，但异步肯定比同步要快。

另外传给 gather 的每个 awaitable 对象可能不是按照确定性顺序完成的，例如将协程 a 和 b 按顺序传递给 gather，但 b 可能会在 a 之前完成。不过 gather 的一个很好的特性是，不管 awaitable 对象何时完成，都保证结果会按照传递它们的顺序返回。

```python
import asyncio
import time

async def main():
    # asyncio.sleep 还可以接收一个 result 参数，作为 await 表达式的值
    tasks = [asyncio.sleep(second, result=f"我睡了 {second} 秒")
             for second in (5, 3, 4)]
    print(await asyncio.gather(*tasks))

start = time.perf_counter()
asyncio.run(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
['我睡了 5 秒', '我睡了 3 秒', '我睡了 4 秒']
总耗时: 5.002968417
"""
```

然后 gather 还可以实现分组，什么意思呢?

```python
import asyncio
import time

async def main():
    gather1 = asyncio.gather(
        *[asyncio.sleep(second, result=f"我睡了 {second} 秒")
          for second in (5, 3, 4)]
    )
    gather2 = asyncio.gather(
        *[asyncio.sleep(second, result=f"我睡了 {second} 秒")
          for second in (3, 3, 3)]
    )
    results = await asyncio.gather(
        gather1, gather2, asyncio.sleep(6, "我睡了 6 秒")
    )
    print(results)


start = time.perf_counter()
asyncio.run(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
[['我睡了 5 秒', '我睡了 3 秒', '我睡了 4 秒'],
 ['我睡了 3 秒', '我睡了 3 秒', '我睡了 3 秒'],
 '我睡了 6 秒']
总耗时: 6.002826208
"""
```

asyncio.gather 里面可以通过继续接收 asyncio.gather 返回的对象，从而实现分组功能，还是比较强大的。

> 如果 gather 里面啥都不传的话，那么会返回一个空列表。

问题来了，在上面的例子中，我们假设所有请求都不会失败或抛出异常，这是理想情况。但如果请求失败了呢？我们来看一下，当 gather 里面的任务出现异常时会发生什么？

```python
import asyncio

async def normal_running():
    await asyncio.sleep(3)
    return "正常运行"

async def raise_error():
    raise ValueError("出错啦")

async def main():
    results = await asyncio.gather(normal_running(), raise_error())
    print(results)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
Traceback (most recent call last):
    ......
    raise ValueError("出错啦")
ValueError: 出错啦
"""
```

我们看到抛异常了，其实 gather 函数的原理就是等待一组任务运行完毕，当某个任务完成时，就调用它的 result 方法，拿到返回值。但我们之前介绍 Future 和 Task 的时候说过，如果出错了，调用 result 方法会将异常抛出来。

```python
import asyncio

async def normal_running():
    await asyncio.sleep(3)
    return "正常运行"

async def raise_error():
    raise ValueError("出错啦")

async def main():
    try:
        await asyncio.gather(normal_running(), raise_error())
    except Exception:
        print("执行时出现了异常")
    # 但是剩余的任务仍在执行，拿到当前的所有正在执行的任务
    all_tasks = asyncio.all_tasks()
    # task 相当于对协程做了一个封装，那么通过 get_coro 方法也可以拿到对应的协程
    print(f"当前剩余的任务:", [task.get_coro().__name__ for task in all_tasks])
    # 继续等待剩余的任务完成
    results = await asyncio.gather(
        *[task for task in all_tasks if task.get_coro().__name__ != "main"]
    )
    print(results)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
执行时出现了异常
当前剩余的任务: ['main', 'normal_running']
['正常运行']
"""
```

可以看到在 await asyncio.gather() 的时候，raise_error() 协程抛异常了，那么异常会向上传播，在 main() 里面 await 处产生 ValueError。我们捕获之后查看剩余未完成的任务，显然只剩下 normal_running() 和 main()，因为任务执行出现异常也代表它完成了。

需要注意的是，一个任务出现了异常，并不影响剩余未完成的任务，它们仍在后台运行。我们举个例子证明这一点：

```python
import asyncio, time

async def normal_running():
    await asyncio.sleep(5)
    return "正常运行"

async def raise_error():
    await asyncio.sleep(3)
    raise ValueError("出错啦")

async def main():
    try:
        await asyncio.gather(normal_running(), raise_error())
    except Exception:
        print("执行时出现了异常")
    # raise_error() 会在 3 秒后抛异常，然后向上抛，被这里捕获
    # 而 normal_running() 不会受到影响，它仍然在后台运行
    # 显然接下来它只需要再过 2 秒就能运行完毕
    time.sleep(2)  # 注意：此处会阻塞整个线程
    # asyncio.sleep 是不耗费 CPU 的，因此即使 time.sleep 将整个线程阻塞了，也不影响
    # 因为执行 time.sleep 时，normal_running() 里面的 await asyncio.sleep(5) 已经开始执行了
    results = await asyncio.gather(*[task for task in asyncio.all_tasks()
                                     if task.get_coro().__name__ != "main"])
    print(results)

loop = asyncio.get_event_loop()
start = time.perf_counter()
loop.run_until_complete(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
执行时出现了异常
['正常运行']
总耗时: 5.004949666
"""
```

这里耗时是 5 秒，说明一个任务抛异常不会影响其它任务，因为 time.sleep(2) 执行完毕之后，normal_running() 里面 asyncio.sleep(5) 也已经执行完毕了，说明异常捕获之后，剩余的任务没有受到影响。

并且这里我们使用了 time.sleep，在工作中千万不要这么做，因为它会阻塞整个线程，导致主线程无法再做其他事情了。而这里之所以用 time.sleep，主要是想说明一个任务出错，那么将异常捕获之后，其它任务不会受到影响。

那么问题来了，如果发生异常，我不希望它将异常向上抛该怎么办呢？可能有人觉得这还不简单，直接来一个异常捕获不就行了？这是一个解决办法，但 asyncio.gather 提供了一个参数，可以更优雅的实现这一点。

```
import asyncio

async def normal_running():
    await asyncio.sleep(3)
    return "正常运行"

async def raise_error():
    raise ValueError("出错啦")

async def main():
    results = await asyncio.gather(
        normal_running(), raise_error(),
        return_exceptions=True
    )
    print(results)


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
['正常运行', ValueError('出错啦')]
"""
```

之前在介绍任务的时候我们说了，不管正常执行结束还是出错，都代表任务已完成，会将结果和异常都收集起来，只不过其中肯定有一个为 None。然后根据不同的情况，选择是否将异常抛出来。所以在 asyncio 里面，异常只是一个普通的属性，会保存在任务对象里面。

对于 asyncio.gather 也是同理，它里面有一个 return_exceptions 参数，默认为 False，当任务出现异常时，会抛给 await 所在的位置。如果该参数设置为 True，那么出现异常时，会直接把异常本身返回（此时任务也算是结束了）。

> 在 asyncio 里面，异常变成了一个可控的属性。因为执行是以任务为单位的，当出现异常时，也会作为任务的一个普通的属性。我们可以选择将它抛出来，也可以选择隐式处理掉。

至于我们要判断哪些任务是正常执行，哪些任务是抛了异常，便可以通过返回值来判断。如果 `isinstance(res, Exception)` 为 True，那么证明任务出现了异常，否则正常执行。虽然这有点笨拙，但也能凑合用，因为 API 并不完美。

当然以上这些都不能算是缺点，gather 真正的缺点有两个：

- 如果我希望所有任务都执行成功，要是有一个任务失败，其它任务自动取消，该怎么实现呢？比如发送 Web 请求，如果一个请求失败，其他所有请求也会失败（要取消请求以释放资源）。显然要做到这一点不容易，因为协程被包装在后台的任务中；
- 其次，必须等待所有任务执行完成，才能处理结果，如果想要在结果完成后立即处理它们，这就存在问题。例如有一个请求需要 100 毫秒，而另一个请求需要 20 秒，那么在处理 100 毫秒完成的那个请求之前，我们将等待 20 秒。

而 asyncio 也提供了用于解决这两个问题的 API。

在任务完成时立即处理

如果想在某个结果生成之后就对其进行处理，这是一个问题；如果有一些可以快速完成的等待对象，和一些可能需要很长时间完成的等待对象，这也可能是一个问题。因为 gather 需要等待所有对象执行完毕，这就导致应用程序可能变得无法响应。

想象一个用户发出 100 个请求，其中两个很慢，但其余的都很快完成。如果一旦有请求完成，可以向用户输出一些信息，来提升用户的使用体验。

为处理这种情况，asyncio 公开了一个名为 as_completed 的 API 函数，这个函数接收一个可等待对象（awaitable）组成的列表，并返回一个生成器。通过遍历，等待它们中的每一个对象都完成，并且哪个先完成，哪个就先被迭代。这意味着将能在结果可用时立即就处理它们，但很明显此时就没有所谓的顺序了，因为无法保证哪些请求先完成。

```python
import asyncio
import time

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    # asyncio 提供的用于等待一组 awaitable 对象的 API 都很智能
    # 如果检测到你传递的是协程，那么会自动包装成任务
    # 不过还是建议手动包装一下
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in (3, 5, 2, 4, 6, 1)]
    for finished in asyncio.as_completed(tasks):
        print(await finished)

loop = asyncio.get_event_loop()
start = time.perf_counter()
loop.run_until_complete(main())
end = time.perf_counter()
print("总耗时:", end - start)
"""
我睡了 1 秒
我睡了 2 秒
我睡了 3 秒
我睡了 4 秒
我睡了 5 秒
我睡了 6 秒
总耗时: 6.000872417
"""
```

和 gather 不同，gather 是等待一组任务全部完成之后才返回，并且会自动将结果取出来，结果值的顺序和添加任务的顺序是一致的。对于 as_completed 而言，它会返回一个生成器，我们遍历它，哪个任务先完成则哪个就先被处理。

那么问题来了，如果出现异常了该怎么办？很简单，直接异常捕获即可。

然后我们再来思考一个问题，任何基于 Web 的请求都存在花费很长时间的风险，服务器可能处于过重的资源负载下，或者网络连接可能很差。

之前我们看到了通过 wait_for 函数可以为特定请求添加超时，但如果想为一组请求设置超时怎么办？as_completed 函数通过提供一个可选的 timeout 参数来处理这种情况，它允许以秒为单位指定超时时间。如果花费的时间超过设定的时间，那么迭代器中的每个可等待对象都会在等待时抛出 TimeoutException。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in (1, 5, 6)]
    for finished in asyncio.as_completed(tasks, timeout=3):
        try:
            print(await finished)
        except asyncio.TimeoutError:
            print("超时啦")

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

as_completed 非常适合用于尽快获得结果，但它也有缺点。

第一个缺点是没有任何方法可快速了解我们正在等待哪个协程或任务，因为运行顺序是完全不确定的。如果不关心顺序，这可能没问题，但如果需要以某种方式将结果与请求相关联，那么将面临挑战。

第二个缺点是超时，虽然会正确地抛出异常并继续运行程序，但创建的所有任务仍将在后台运行。如果想取消它们，很难确定哪些任务仍在运行，这是我们面临的另一个挑战。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in (1, 5, 6)]
    for finished in asyncio.as_completed(tasks, timeout=3):
        try:
            print(await finished)
        except asyncio.TimeoutError:
            print("超时啦")

    # tasks[1] 还需要 2 秒运行完毕，tasks[2] 还需要 3 秒运行完毕
    print(tasks[1].done(), tasks[2].done())

    await asyncio.sleep(2)
    # 此时只剩下 tasks[2]，还需要 1 秒运行完毕
    print(tasks[1].done(), tasks[2].done())

    await asyncio.sleep(1)
    # tasks[2] 也运行完毕
    print(tasks[1].done(), tasks[2].done())


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
超时啦
超时啦
False False
True False
True True
"""
```

根据输出结果可以发现，虽然因为抵达超时时间， await 会导致 TimeoutError，但未完成的任务不会

受到影响，它们仍然在后台执行。

但这对于我们来说，有时却不是一件好事，因为我们希望如果抵达超时时间，那么未完成的任务就别在执行了，这时候如何快速找到那些未完成的任务呢？为处理这种情况，asyncio 提供了另一个 API 函数：wait。

使用 wait 进行细粒度控制

gather 和 as_completed 的缺点之一是，当我们看到异常时，没有简单的方法可以取消已经在运行的任务。这在很多情况下可能没问题，但是想象一个场景：同时发送大批量 Web 请求（参数格式是相同的），如果某个请求的参数格式错误（说明所有请求的参数格式都错了），那么剩余的请求还有必要执行吗？显然是没有必要的，而且还会消耗更多资源。另外 as_completed 的另一个缺点是，由于迭代顺序是不确定的，因此很难准确跟踪已完成的任务。

于是 asyncio 提供了 wait 函数，注意它和 wait_for 的区别，wait_for 针对的是单个任务，而 wait 则针对一组任务（不限数量）。

> 注：wait 函数接收的是一组 awaitable 对象，但未来的版本改为仅接收任务对象。因此对于 gather、as_completed、wait 等函数，虽然它们会自动包装成任务，但我们更建议先手动包装成任务，然后再传过去。
>
> 并且 wait 和 as_completed 接收的都是任务列表，而 gather 则要求将列表打散，以多个位置参数的方式传递，因此这些 API 的参数格式不要搞混了。

然后是 wait 函数的返回值，它会返回两个集合：一个由已完成的任务（执行结束或出现异常）组成的集合，另一个由未完成的任务组成的集合。而 wait 函数的参数，它除了可以接收一个任务列表之外，还可以接收一个 timeout（超时时间）和一个 return_when（用于控制返回条件）。光说很容易乱，我们来实际演示一下。

等待所有任务完成

如果未指定 retun_when，则此选项使用默认值，并且它的行为与 asyncio.gather 最接近，但也存在一些差异。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds)) for seconds in (3, 2, 4)]
    # 和 gather 一样，默认会等待所有任务都完成
    done, pending = await asyncio.wait(tasks)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")

    for done_task in done:
        print(await done_task)


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
已完成的任务数: 3
未完成的任务数: 0
我睡了 2 秒
我睡了 4 秒
我睡了 3 秒
"""
```

await asynio.wait 时，会返回两个集合，分别保存已完成的任务和仍然运行的任务。并且由于返回的是集合，所以是无序的。默认情况下，asyncio.wait 会等到所有任务都完成后才返回，所以待处理集合的长度为 0。

然后还是要说一下异常，如果某个任务执行时出现异常了该怎么办呢?

```
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds)) for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
已完成的任务数: 5
未完成的任务数: 0
Task exception was never retrieved
future: <Task finished ... coro=<delay() done, defined at .../main.py:3>
         exception=ValueError('我出错了(second is 3)')>
    ......
    raise ValueError("我出错了(second is 3)")
ValueError: 我出错了(second is 3)
"""
```

对于 asyncio.gather 而言，如果某个任务出现异常，那么异常会向上抛给 await 所在的位置。如果不希望它抛，那么可以将 gather 里面的 return_exceptions 参数指定为 True，这样当出现异常时，会将异常返回。

而 asyncio.wait 也是如此，如果任务出现异常了，那么会直接视为已完成，异常同样不会向上抛。但是从程序开发的角度来讲，返回值可以不要，但异常不能不处理。

所以当任务执行出错时，虽然异常不会向上抛，但 asyncio 会将它打印出来，于是就有了：Task exception was never retrieved。意思就是该任务出现异常了，但你没有处理它。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds)) for seconds in range(1, 6)]
    # done 里面保存的都是已完成的任务
    done, pending = await asyncio.wait(tasks)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")

    # 所以我们直接遍历 done 即可
    for done_task in done:
        # 这里不能使用 await done_task，因为当任务完成时，它就等价于 done_task.result()
        # 而任务出现异常时，调用 result() 是会将异常抛出来的，所以我们需要先检测异常是否为空
        exc = done_task.exception()
        if exc:
            print(exc)
        else:
            print(done_task.result())


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
已完成的任务数: 5
未完成的任务数: 0
我睡了 5 秒
我睡了 2 秒
我出错了(second is 3)
我睡了 4 秒
我睡了 1 秒
"""
```

这里调用 result 和 exception 有一个前提，就是任务必须处于已完成状态，否则会抛异常：InvalidStateError: Result is not ready.。但对于我们当前是没有问题的，因为 done 里面的都是

已完成的任务。

这里能再次看到和 gather 的区别，gather 会帮你把返回值都取出来，放在一个列表中，并且顺序就是任务添加的顺序。而 wait 返回的是集合，集合里面是任务，我们需要手动拿到返回值。

某个完成出现异常时取消其它任务

从目前来讲，wait 的作用和 gather 没有太大的区别，都是等到任务全部结束再解除等待（出现异常也视作任务完成，并且其它任务不受影响）。那如果我希望当有任务出现异常时，立即取消其它任务该怎么做呢？显然这就依赖 wait 函数里面的 return_when，它有三个可选值：

- asyncio.ALL_COMPLETED：等待所有任务完成后返回；
- asyncio.FIRST_COMPLETED：有一个任务完成就返回；
- asyncio.FIRST_EXCEPTION：当有任务出现异常时返回；

显然为完成这个需求，我们应该将 return_when 指定为 FIRST_EXCEPTION。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds), name=f"睡了 {seconds} 秒的任务")
             for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks, return_when=asyncio.FIRST_EXCEPTION)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")

    print("都有哪些任务完成了? ")
    for t in done:
        print("    " + t.get_name())

    print("还有哪些任务没完成? ")
    for t in pending:
        print("    " + t.get_name())

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
已完成的任务数: 3
未完成的任务数: 2
都有哪些任务完成了?
    睡了 2 秒的任务
    睡了 3 秒的任务
    睡了 1 秒的任务
还有哪些任务没完成?
    睡了 4 秒的任务
    睡了 5 秒的任务
"""
```

当 delay(3) 失败时，显然 delay(1)、delay(2) 已完成，而 delay(4) 和 delay(5) 未完成。此时集合 done 里面的就是已完成的任务，pending 里面则是未完成的任务。

当 wait 返回时，未完成的任务仍在后台继续运行，如果我们希望将剩余未完成的任务取消掉，那么直接遍历 pending 集合即可。

```
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    print(f"我睡了 {seconds} 秒")

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks, return_when=asyncio.FIRST_EXCEPTION)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")
    # 此时未完成的任务仍然在后台运行，这时候我们可以将它们取消掉
    for t in pending:
        t.cancel()
    # 阻塞 3 秒
    await asyncio.sleep(3)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
我睡了 2 秒
已完成的任务数: 3
未完成的任务数: 2
"""
```

在 await asyncio.sleep(3) 的时候，剩余两个任务并没有输出，所以任务确实被取消了。注：出现异常的任务会被挂在已完成集合里面，如果没有任务在执行时出现异常，那么效果等价于 ALL_COMPLETED。

当任务完成时处理结果

ALL_COMPLETED 和 FIRST_EXCEPTION 都有一个缺点，在任务成功且不抛出异常的情况下，必须等待所有任务完成。对于之前的用例，这可能是可以接受的，但如果想要在某个协程成功完成后立即

处理结果，那么现在的情况将不能满足我们的需求。

虽然这个场景可使用 as_completed 实现，但 as_completed 的问题是没有简单的方法可以查看哪些任务还在运行，哪些任务已经完成。因为遍历的时候，我们无法得知哪个任务先完成，所以 as_completed 无法完成我们的需求。

好在 wait 函数的 return_when 参数可以接收 FIRST_COMPLETED 选项，表示只要有一个任务完成就立即返回，而返回的可以是执行出错的任务，也可以是成功运行的任务（任务失败也表示已完成）。然后，我们可以取消其他正在运行的任务，或者让某些任务继续运行，具体取决于用例。

```
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    print(f"我睡了 {seconds} 秒")

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
已完成的任务数: 1
未完成的任务数: 4
"""
```

当 return_when 参数为 FIRST_COMPLETED 时，那么只要有一个任务完成就会立即返回，然后我们处理完成的任务即可。至于剩余的任务，它们仍在后台运行，我们可以继续对其使用 wait 函数。

```python
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    if seconds == 3:
        raise ValueError("我出错了(second is 3)")
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in range(1, 6)]
    while True:
        done, pending = await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED)
        for t in done:
            exc = t.exception()
            print(exc) if exc else print(t.result())

        if pending:  # 还有未完成的任务，那么继续使用 wait
            tasks = pending
        else:
            break

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
我睡了 1 秒
我睡了 2 秒
我出错了(second is 3)
我睡了 4 秒
我睡了 5 秒
"""
```

整个行为和 as_completed 是一致的，但这种做法有一个好处，就是我们每一步都可以准确地知晓哪些任务已经完成，哪些任务仍然运行，并且也可以做到精确取消指定任务。

处理超时

除了允许对如何等待协程完成进行更细粒度的控制外，wait 还允许设置超时，以指定我们希望等待完成的时间。要启用此功能，可将 timeout 参数设置为所需的最大秒数，如果超过了这个超时时间，wait 将立即返回 done 和 pending 任务集。

不过与目前所看到的 wait_for 和 as_completed 相比，超时在 wait 中的行为方式存在一些差异。

1）协程不会被取消。

当使用 wait_for 时，如果任务超时，则引发 TimeouError，并且任务也会自动取消。但使用 wait 的情况并非如此，它的行为更接近我们在 as_completed 中看到的情况。如果想因为超时而取消协程，必须显式地遍历任务并取消，否则它们仍在后台运行。

2）不会引发超时错误。

如果发生超时，则 wait 返回所有已完成的任务，以及在发生超时的时候仍处于运行状态的所有任务。

```
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks, timeout=3.1)
    print(f"已完成的任务数: {len(done)}")
    print(f"未完成的任务数: {len(pending)}")

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
已完成的任务数: 3
未完成的任务数: 2
"""
```

wait 调用将在 3 秒后返回 done 和 pending 集合，在 done 集合中，会有三个已完成的任务。而耗时 4 秒和 5 秒的任务，由于仍在运行，因此它们将出现在 pending 集合中。我们可以继续等待它们完成并提取返回值，也可以将它们取消掉。

需要注意：和之前一样，pending 集合中的任务不会被取消，并且继续运行，尽管会超时。对于要终止待处理任务的情况，我们需要显式地遍历 pending 集合并在每个任务上调用 cancel。

为什么要先将协程包装成任务

我们说协程在传给 wait 的时候会自动包装成任务，那为什么我们还要手动包装呢？

```
import asyncio

async def delay(seconds):
    await asyncio.sleep(seconds)
    return f"我睡了 {seconds} 秒"

async def main():
    tasks = [asyncio.create_task(delay(seconds))
             for seconds in range(1, 6)]
    done, pending = await asyncio.wait(tasks, timeout=3.1)
    print(all(map(lambda t: t in tasks, done)))
    print(all(map(lambda t: t in tasks, pending)))

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
"""
True
True
"""
```

如果 wait 函数接收的就是任务，那么 wait 函数就不会再包装了，所以 done 和 pending 里面的任务和 tasks 里面的任务是相同的。基于这个条件，我们后续可以做一些比较之类的。

比如有很多 Web 请求任务，但如果当未完成的任务是 task1、task2、task3，那么就取消掉，于是可以这么做。

```
for t in pending:
    if t in (task1, task2, task3):
        t.cancel()
```

如果返回的 done 和 pending 里的任务，是在 wait 函数中自动创建的，那么我们就无法进行任何比较来查看 pending 集合中的特定任务。

小结

1）asyncio.gather 函数允许同时运行多个任务，并等待它们完成。一旦传递给它的所有任务全部完成，这个函数就会返回。由于 gather 会拿到里面每个任务的返回值，所以它要求每个任务都是成功的，如果有任务执行出错（没有返回值），那么获取返回值的时候就会将异常抛出来，然后向上传递给 await asyncio.gather。

为此，可以将 return_exceptions 设置为 True，这将返回成功完成的可等待对象的结果，以及产生的异常（异常会作为一个普通的属性返回，和返回值是等价的）。

2）可使用 as_completed 函数在可等待对象列表完成时处理它们的结果，它会返回一个可以循环遍历的生成器。一旦某个协程或任务完成，就能访问结果并处理它。

3）如果希望同时运行多个任务，并且还希望能了解哪些任务已经完成，哪些任务在运行，则可以使用 wait。这个函数还允许在返回结果时进行更多控制，返回时，我们会得到一组已经完成的任务和一组仍在运行的任务。

然后可以取消任何想要取消的任务，或执行其他任何需要执行的任务。并且 wait 里面的任务出现异常，也不会影响其它任务，异常会作为任务的一个属性，只是在我们没有处理的时候会给出警告。至于具体的处理方式，我们直接通过 exception 方法判断是否发生了异常即可，没有异常返回 result()，有异常返回 exception()。
