链滴

在 PHP 中使用 Promise + co/yield 协程

作者：andot
原文链接：https://ld246.com/article/1481384838707
来源网站：链滴

摘要: 我们知道 JavaScript 自从有了 Generator 之后，就有了各种基于 Generator 封装的协程。其中 hprose 中封装的 Promise 和协程库实现了跟 ES2016 的 async/await 一样的功能，并且更加灵活。们还知道 PHP 自从 5.5 之后，也引入了 Generator，同样也有了各种基于它封装的 PHP 协程库，hpose 同样也为 PHP 提供的跟 JavaScript 版本类似的 Promise 和协程库。下面我们就来看一下它跟 soole 结合的效果。

为什么需要异步方式

一个函数执行之后，在它后面顺序编写的代码中，如果能够直接使用它的返回结果或者它修改之后的用参数，那么我们通常认为该函数是同步的。

而如果一个函数的执行结果或者其修改的引用参数，需要通过设置回调函数或者回调事件的方式来获，而在其后顺序编写的代码中无法直接获取的话，那么我们通常认为这样的函数是异步的。

PHP 提供的大部分函数都是同步的。通常我们会有一个误解，那就是容易把同步和阻塞当成同一个概，但实际上同步代码不一定都是阻塞的，只是同步代码对阻塞天然友好，当同步代码和阻塞结合时，码通常是简单易懂的。

阻塞带来的问题是当前线程（或进程）会陷入等待，一直等到阻塞结束，这样就会造成线程（或进程资源的浪费。所以，通常认为阻塞是不够高效的。

但是如果要编写非阻塞代码，使用同步方式会变得有些复杂，且不够灵活。同步方式的非阻塞代码通会使用 select 模式，例如 curl_multi_select, stream_select, socket_select 等就是 PHP 中提供的一典型的 select 模式的函数。

我们说它复杂且不够灵活是有理由的，例如使用上面的 select 模式编写同步的非阻塞代码时，我们需先构造一个并发任务的列表，之后手动构造循环来执行这些并发的任务，在循环开始之后，虽然这几任务可以并发，但是这个循环相对于其后的代码总体上仍然是阻塞的，我们要想拿到这些并发任务的果时，仍然需要等待。select 虽然可以同时等待多个任务中某一个或几个就位后，再执行后续操作，仍然有一部分时间是被等待消耗掉的。而且如果是纯同步非阻塞的情况下，我们也很难在循环开始后动态添加更多的任务到这个循环中去。

所以，如果我们希望程序能够更加高效，更加灵活，就需要引入异步方式。

传统的异步方式有什么问题

一提到异步模式，大家脑子中的第一印象可能就是**回调、回调、回调**。是的，这是最简单最直接也是前最常见的异步模式。只要在调用异步函数时设置一个或多个回调函数，函数就会在完成时自动调用调函数。或者为一个对象设置一堆事件，之后调用该对象上的某个异步方法，虽然这个异步方法本身能不再需要设置回调函数，但是设置的这堆事件实际上跟回调函数所起到的作用是一样的。

如果你的程序逻辑够简单，简单的一两层回调也许并不会让你觉得异步方式的编程有什么麻烦。但如你的程序逻辑一旦有些复杂，你可能就会被层层回调搞得疲惫不堪了。当然，实际上你的程序需要层回调的原因，也许并不是你的程序逻辑真的复杂，而是你没有办法将回调函数中的参数结果传出来，以，你就不得不将另一个回调函数传进去。

我们来举一个简单的例子，假设我们有 1 个同步函数：

```
function sum($a, $b) {
    return $a + $b;
}
```

然后我们按照下面的方式去调用它：

```
$a = sum(1, 2);
$b = sum($a, 3);
$c = sum($b, 4);
var_dump(array($a, $b, $c));
```

虽然上面的代码很不精简，但我们要表达的意图很明确，而且代码看起来很清楚。

那接下来我们把这个函数换成一个形式上的异步函数，例如：

```
function async_sum($a, $b, $callback) {
    $callback($a + $b);
}
```

当然，它的执行并不是异步的，这里我们先不关心它的实现是不是真异步的。

现在如果要做上面同样的操作，代码就要这样写了：

```
async_sum(1, 2, function($a) {
    async_sum($a, 3, function($b) use ($a) {
        async_sum($b, 4, function($c) use ($a, $b) {
            var_dump(array($a, $b, $c));
        });
    });
});
```

代码的执行结果是一样的。但异步的代码看起来显然更难读一些，虽然这已经是很简单的例子了。

好了，看到这里，有些读者可能会觉的我上面的这个例子很糟糕。因为明明有同步的函数可以使用，且代码清晰可读，为啥非要写个形似异步的函数，把本来同步可以做的很好的事情用异步方式复杂化？而且那个异步调用的方式，最后不还是想要实现同步化的结果吗？

如果你这么想的话，一点都没错。但我们这里想要解决的问题是，如果我们拿到的只有一个异步函数这个函数没有同步实现，我们也不知道这个异步函数的内部定义是怎样的，我们也没办法将这个异步数改为同步函数实现。那我们有没有办法将上面的程序改的更可读一些呢？

当然是可以的，所以，现在 Promise 要登场了。

为什么要引入 Promise

通常我们对 Promise 的一个误解就是，它要解决的是层层回调的问题，比如上面的问题看上去就是个典型的层层回调的问题。

然而实际上，Promise 要解决的并不是回调不回调的问题，如果你使用过 Promise 的话，你会发现用 Promise 你仍然少不了要使用回调。Promise 要解决的问题是，如何将回调方法的参数从回调方中传递出来，让它可以像同步函数的返回结果一样，在回调函数以外的控制范围内，可以传递和复用。

下面这几篇文章可能会对大家理解 Promise 有所帮助：

- 深入理解 Promise 五部曲：1. 异步问题
- 深入理解 Promise 五部曲：2. 控制权转换问题
- 深入理解 Promise 五部曲：3. 可靠性问题
- 深入理解 Promise 五部曲：4. 扩展问题
- 深入理解 Promise 五部曲：5. LEGO

我觉得这几篇文章讲的比较透彻，所以我就不重复文章中的内容了。

下面我们来看上面的例子用 Promise 如何解。

我们现在用最简单粗暴的方式来引入 Hprose 的库，直接复制源码而不是使用 composer。然后我们代码中直接使用：

```
<?php
require_once("Hprose.php");
use Hprose\Promise;
```

这种方式来引入 Hprose 的 Promise 库，当然你也可以写成：

```
<?php
require_once("Hprose.php");
use Hprose\Future;
```

Future 库跟 Promise 库基本上是一样的，你可以认为 Future 是 Promise 的具体实现，Promise 只 Future 实现的一个包装。这个区别你可以从源码中直接看出来，这里就不多做解释了。

接下来，我们要把前面的 async_sum 函数 Promise 化，Hprose 提供了这样一个函数：Promise\promisify（或者 Future\promisify），它的作用就是将一个使用回调方式的异步函数变成一个返回 Promise 对象的异步函数。这样说，也许有些不好理解，下面直接上代码：

```
<?php
require_once("Hprose.php");

use Hprose\Promise;

function async_sum($a, $b, $callback) {
    $callback($a + $b);
}

$sum = Promise\promisify('async_sum');

$a = $sum(1, 2);
$b = $a->then(function($a) use ($sum) {
    return $sum($a, 3);
});
$c = $b->then(function($b) use ($sum) {
   return $sum($b, 4);
});

Promise\all(array($a, $b, $c))->then(function($result) {
    var_dump($result);
});
```

好了，看到这里，如果你对 Promise 的理解还不够深入的话，你的第一反应可能是：这不是把程序得更复杂了吗？原来的程序是 3 个回调，现在仍然是 3 个回调，还多了包装，都玩出花来了，有意思？

确实，从上面的代码来看，代码并没有被简化，但是你会发现，现在回调函数中的参数已经通过 Promise 返回值的方式传递出来了，而且可以在原本的回调函数控制范围以外被传递和复用了。

但是你可能会说然并卵，程序不是仍然很复杂吗？那我们就来进一步简化一下：

```php
<?php
require_once("Hprose.php");

use Hprose\Promise;

function async_sum($a, $b, $callback) {
    $callback($a + $b);
}

$sum = Promise\wrap(Promise\promisify('async_sum'));
$var_dump = Promise\wrap('var_dump');

$a = $sum(1, 2);
$b = $sum($a, 3);
$c = $sum($b, 4);

$var_dump(Promise\all(array($a, $b, $c)));
```

现在，代码中再也看不到回调了。因为我们把函数包装成了可以接收 Promise 变量的函数。当然，实现细节略微有些复杂，如果你感兴趣，可以去看一下源码，这里就不做源码剖析了。如果感兴趣的者多得话，以后有时间再写源码剖析。

当然，如果你只是想把异步调用同步化，除了 Promise\wrap 外，你还可以通过 co/yield 协程来实。

Hprose 中的 co/yield 协程

还是上面的例子，如果你使用的是 PHP 5.5 或者更高版本，那么你可以这样来写代码了。

```php
<?php
require_once("Hprose.php");

use Hprose\Promise;

function async_sum($a, $b, $callback) {
    $callback($a + $b);
}

Promise\co(function() {
    $sum = Promise\promisify('async_sum');

    $a = (yield $sum(1, 2));
    $b = (yield $sum($a, 3));
    $c = (yield $sum($b, 4));

    var_dump(array($a, $b, $c));
});
```

这代码比使用 Promise\wrap 的又要简单了。这里，代码中的变量 $a, $b, $c 不再是 Promise 变量而是实实在在的整数变量。也就是说，yield 把一个 Promise 变量变成了一个普通变量。

现在 Promise\co 中的代码已经被实实在在的同步化了。

现在你可能有新的疑问了，异步不是为了高效吗？现在把原本的异步代码同步化了，那还会高效吗？

当然，对这个例子上来说，效率肯定是没有提高，反而是严重降低的。甚至在这个例子中，最原始的个形似异步的实现也不比同步实现更高效。因为在这个例子中，并没有涉及到并发和 IO 阻塞的情况。

下面我们就放到真实场景下来看看 Promise 和 co/yield 协程是怎么用的。

在 swoole 下使用 Promise 和 co/yield 协程

我们知道在 PHP 中，如果要让程序延时可以使用 sleep 函数（或者 usleep, time_nanosleep 函数来让程序阻塞一会儿，但是这个阻塞会让整个进程都阻塞，所以在阻塞期间，什么都不能干。

下面我们来看看使用 swoole_timer_after 实现的延时执行：

```
<?php
require_once("Hprose.php");

use Hprose\Future;

date_default_timezone_set('UTC');

function wait($time) {
    $wait = Future\promisify('swoole_timer_after');
    for ($i = 0; $i < 5; $i++) {
        yield $wait($time);
        var_dump("wait ". ($time / 1000) . "s, now is " . date("H:i:s"));
    }
}

Future\co(wait(2000));
Future\co(wait(1000));
```

该程序执行结果如下：

```
string(24) "wait 1s, now is 13:48:25"
string(24) "wait 2s, now is 13:48:26"
string(24) "wait 1s, now is 13:48:26"
string(24) "wait 1s, now is 13:48:27"
string(24) "wait 2s, now is 13:48:28"
string(24) "wait 1s, now is 13:48:28"
string(24) "wait 1s, now is 13:48:29"
string(24) "wait 2s, now is 13:48:30"
string(24) "wait 2s, now is 13:48:32"
string(24) "wait 2s, now is 13:48:34"
```

从结果中我们可以看出，wait(2000) 和 wait(1000) 各自都是顺序阻塞执行的，但是它们之间却是并执行的。

也就是说，协程之间并不会相互阻塞，虽然这几个并发的协程是在同一个进程内跑的。

最后我们再来看一个用 co/yield 协程实现的并发抓图程序：

```
<?php
require_once("Hprose.php");

use Hprose\Promise;
```

```php
function fetch($url) {
    $dns_lookup = Promise\promisify('swoole_async_dns_lookup');
    $writefile = Promise\promisify('swoole_async_writefile');
    $url = parse_url($url);
    list($host, $ip) = (yield $dns_lookup($url['host']));
    $cli = new swoole_http_client($ip, isset($url['port']) ? $url['port'] : 80);
    $cli->setHeaders([
        'Host' => $host,
        "User-Agent" => 'Chrome/49.0.2587.3',
    ]);
    $get = Promise\promisify([$cli, 'get']);
    yield $get($url['path']);
    list($filename) = (yield $writefile(basename($url['path']), $cli->body));
    echo "write $filename ok.\r\n";
    $cli->close();
}

$urls = array(
    'http://b.hiphotos.baidu.com/baike/c0%3Dbaike116%2C5%2C5%2C116%2C38/sign=5f451ba037b020818c437b303b099b6/472309f790529822434d08dcdeca7bcb0a46d4b6.jpg',
    'http://f.hiphotos.baidu.com/baike/c0%3Dbaike116%2C5%2C5%2C116%2C38/sign=1c3778b3cc79f3d9becec62dbc8a674/38dbb6fd5266d016dc2eaa5c902bd40735fa358a.jpg',
    'http://h.hiphotos.baidu.com/baike/c0%3Dbaike116%2C5%2C5%2C116%2C38/sign=edd059c502c11dfcadcb771024e09b5/d6ca7bcb0a46f21f3100c52cf1246b600c33ae9d.jpg',
    'http://a.hiphotos.baidu.com/baike/c0%3Dbaike92%2C5%2C5%2C92%2C30/sign=46937568094a4c21e2eef796f9d70b0/54fbb2fb43166d22df5181f5412309f79052d2a9.jpg',
    'http://a.hiphotos.baidu.com/baike/c0%3Dbaike92%2C5%2C5%2C92%2C30/sign=938850744a98226accc2375ebebd264/faf2b2119313b07eb2cc820c0bd7912397dd8c45.jpg',
);

foreach ($urls as $url) {
    Promise\co(fetch($url));
}
```

在这个程序中，fetch 函数内的代码是同步执行的，但是多个 fetch 之间却是并发执行的，从结果输就可以看出来，输出顺序是不一定的。但最后，你总能得到所有的美图。

总结：通过 swoole 跟 hprose 中的 Promise 和 co/yield 协程相结合，你可以方便的使用同步的方来调用 swoole 中的异步函数和方法，并可以实现协程间的并发。

因为篇幅所限，这里无法把 hprose 中 Promise 和 co/yield 协程的全部内容都介绍完，如果你想了更多，可以参考下面两篇内容：

- Promise 异步编程
- co/yield 协程