

Hprose for PHP 用户手册

书栈(BookStack.CN)

目 录

致谢

00 介绍

01 安装

02 如何在程序中引用 Hprose

03 Promise 异步编程

04 协程

05 Hprose 客户端

06 Hprose 服务器

07 Hprose 服务器事件

08 HTTP 服务器特殊设置

09 Socket 服务器特殊设置

10 推送服务

11 Hprose 过滤器

12 Hprose 中间件

附录A 2.0 新特征

附录B Hprose 的 pecl 扩展

致谢

当前文档《Hprose for PHP 用户手册》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建，生成于 2018-07-12。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/hprose-php>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

00 介绍

Hprose for PHP 用户手册

HPROSE 是 *High Performance Remote Object Service Engine* 的缩写，翻译成中文就是“高性能远程对象服务引擎”。

它是一个先进的轻量级的跨语言跨平台面向对象的高性能远程动态通讯中间件。它不仅简单易用，而且功能强大。你只需要稍许的时间去学习，就能用它轻松构建跨语言跨平台的分布式应用系统了。

Hprose 支持众多流行的编程语言，例如：

- AAuto Quicker
- ActionScript
- ASP
- C++
- Delphi/Free Pascal
- dotNET(C#, Visual Basic...)
- Golang
- Java
- JavaScript
- Node.js
- Objective-C
- Perl
- PHP
- Python
- Ruby

通过 Hprose，你就可以在这些语言之间方便高效的实现互通了。

本项目是 Hprose 的 PHP 版本实现。

如果你喜欢本项目，请点击右上角的 Star，这样就可以将本项目放入您的收藏。

如果你非常喜欢，请点击右上角的 Fork，这样就可以直接将本项目直接复制到您的名下。

如果您有问题需要反馈，请点击 github 上的 [issues](#) 提交您的问题。

如果您改进了代码，并且愿意将它合并到本项目中，你可以使用 github 的 [pull requests](#) 功能来提交您的修改。

接下来让我们开始 Hprose for PHP 之旅吧。

原文：<https://github.com/hprose/hprose-php/wiki>

01 安装

Hprose 可以直接下载源码使用，也可以使用 composer 来进行管理。

下载源码

直接使用：

```
1. git clone https://github.com/hprose/hprose-php
```

命令下载到本地。

也可以点击 <https://github.com/hprose/hprose-php/archive/master.zip> 下载最新内容的压缩包，然后解压。

通过 composer 来安装

在你的 composer 项目中的 composer.json 文件中，添加这部分：

```
1. {  
2.     "require": {  
3.         "hprose/hprose": ">=2.0.0"  
4.     }  
5. }
```

就可以了。

如果你需要使用 swoole 版本，可以添加：

```
1. {  
2.     "require": {  
3.         "hprose/hprose-swoole": "dev-master"  
4.     }  
5. }
```

如果你需要使用 Symfony 的服务器支持，可以添加：

```
1. {  
2.     "require": {  
3.         "hprose/hprose-symfony": "dev-master"  
4.     }  
5. }
```

如果你需要使用 Yii 的服务器支持，可以添加：

```
1. {  
2.     "require": {  
3.         "hprose/hprose-yii": "dev-master"  
4.     }  
5. }
```

如果你使用的服务器支持 PSR7 规范，可以添加：

```
1. {  
2.     "require": {  
3.         "hprose/hprose-psr7": "dev-master"  
4.     }  
5. }
```

如果你需要对其它 PHP 框架的支持，可以参照：

- <https://github.com/hprose/hprose-symfony>
 - <https://github.com/hprose/hprose-yii>
 - <https://github.com/hprose/hprose-psr7>
- 这三个项目来自自己实现，

原文： <https://github.com/hprose/hprose-php/wiki/01-%E5%AE%89%E8%A3%85>

02 如何在程序中引用 Hprose

composer 方式

如果你正在使用 composer 管理你的项目，那么你不需要做任何特别处理。只要在 composer.json 中的 `require` 段添加了对 `hprose/hprose` 的引用就可以了。如果你需要 swoole 支持，添加 `hprose/hprose-swoole` 就可以了。

然后在代码这样引用：

```
1. <?php
2. require_once "vendor/autoload.php";
3.
4. use Hprose\Swoole\Http\Server;
5.
6. function hello($name) {
7.     return "Hello $name!";
8. }
9.
10. $server = new Server("http://0.0.0.0:8000");
11. $server->add("hello");
12. $server->debug = true;
13. $server->crossDomain = true;
14. $server->start();
```

手动管理方式

如果你不打算使用 composer 来管理你的项目，那你可以直接把 hprose-php 里面的 src 目录复制到你的项目中，然后改成任何你喜欢的名字，比如改为 hprose。

如果你还需要使用 hprose-swoole 下的文件，而且也不想使用 composer 来管理项目。你只需要把 hprose-swoole 下的 src 中的文件，复制到 hprose-php 下的 src 下对应的目录中，就可以了。

然后像这样引用它：

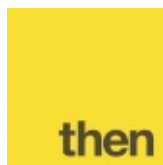
```
1. <?php
2. require_once 'hprose/Hprose.php';
3.
4. use Hprose\Swoole\Http\Server;
5.
6. function hello($name) {
7.     return "Hello $name!";
8. }
```

```
9.  
10. $server = new Server("http://0.0.0.0:8000");  
11. $server->add("hello");  
12. $server->debug = true;  
13. $server->crossDomain = true;  
14. $server->start();
```

但是在后面其他章节中，为了方便统一，我们一律采用 composer 方式来写示例代码，而不再采用上面这种手动管理方式。

原文: <https://github.com/hprose/hprose-php/wiki/02-%E5%A6%82%E4%BD%95%E5%9C%A8%E7%A8%8B%E5%BA%8F%E4%B8%AD%E5%BC%95%E7%94%A8-Hprose>

03 Promise 异步编程



概述

PHP 的主要编程模式是同步方式，如果要在 PHP 中进行异步编程，通常是采用回调的方式，因为这种方式简单直接，不需要第三方库的支持，但缺点是当回调层层嵌套使用时，会严重影响程序的可读性和可维护性，因此层层回调的异步编程让人望而生畏。

回调的问题在 JavaScript 中更加明显，因为异步编程模式是 JavaScript 的主要编程模式。为了解决这个问题，JavaScript 社区提出了一套 Promise 异步编程模型。[Promise/A+\(中文版\)](#) 是一个通用的、标准化的规范，它提供了一个可互操作的 `then` 方法的实现定义。Promise/A+ 规范的实现有很多，并不局限于 JavaScript 语言，它们的共同点就是都有一个标准的 `then` 方法，而其它的 API 则各不相同。

Hprose 2.0 为了更好的实现异步服务和异步调用，也为 PHP 提供了一套 Promise 异步模型实现。它基本上是参照 Promise/A+(中文版) 规范实现的。

Hprose 2.0 之前的版本提供了一组 `Future` / `Completer` 的 API，其中 `Future` 对象上也提供了 `then` 方法，但最初是参照 Dart 语言中的 `Future` / `Completer` 设计的。

而在 Hprose 2.0 版本中，我们对 `Future` 的实现做了比较大的改进，现在它既兼容 Dart 的 `Future` / `Completer` 使用方式，又兼容 [Promise/A+ 规范](#)，而且还增加了许多非常实用的方法。下面我们就来对这些方法做一个全面的介绍。

注意：下面的例子中，为了突出重点，代码中均省略了：

```
1. <?php
2. require_once "vendor/autoload.php";
```

请读者自行脑补。

创建 Future/Promise 对象

Hprose 中提供了多种方法来创建 Future/Promise 对象。为了方便讲解，在后面我们不再详细区分 Future 对象和 Promise 对象实例的差别，统一称为

`promise` 对象。

使用 Future 构造器

创建一个待定 (pending) 状态 promise 对象

```
1. use Hprose\Future;
2. $promise = new Future();
```

该 `promise` 对象的结果尚未确定，可以在将来通过 `resolve` 方法来设定其成功值，或通过 `reject` 方法来设定其失败原因。

创建一个成功 (fulfilled) 状态的 promise 对象

```
1. use Hprose\Future;
2. $promise = new Future(function() { return 'hprose'; });
3. $promise->then(function($value) {
4.     var_dump($value);
5. });
```

该 `promise` 对象中已经包含了成功值，可以使用 `then` 方法来得到它。

创建一个失败 (rejected) 状态的 promise 对象

```
1. use Hprose\Future;
2. $promise = new Future(function() { throw new Exception('hprose'); });
3. $promise->catchError(function($reason) {
4.     var_dump($reason);
5. });
```

该 `promise` 对象中已经包含了失败值，可以使用 `catchError` 方法来得到它。

上面的 `Future` 构造函数的参数可以是无参的函数、方法、闭包等，或者说只要是无参的 callable 对象就可以，不一定非要用闭包。

使用 Hprose\Future 名空间中的工厂方法

`Hprose\Future` 名空间内提供了 6 个工厂方法，它们分别是：

- `resolve`
- `value`
- `reject`

- error
- sync
- promise

其中 `resolve` 和 `value` 功能完全相同，`reject` 和 `error` 功能完全相同。

`resolve` 和 `reject` 这两个方法名则来自 ECMAScript 6 的 Promise 对象。

`value` 和 `error` 这两个方法名来自 Dart 语言的 `Future` 类。因为最初是按照 Dart 语言的 API 设计的，因此，这里保留了 `value` 和 `error` 这两个方法名。

`sync` 功能跟 `Future` 含参构造方法类似，但在返回值的处理上有所不同。

`promise` 方法跟 `Promise` 类的构造方法类似，但返回的是一个 `Future` 类型的对象，而 `Promise` 构造方法返回的是一个 `Promise` 类的对象，`Promise` 类是 `Future` 类的子类，但除了构造函数不同以外，其它都完全相同。

创建一个成功 (fulfilled) 状态的 promise 对象

```
1. use Hprose\Future;
2. $promise = Future\value('hprose'); // 换成 Future\resolve('hprose') 效果一样
3. $promise->then(function($value) {
4.     var_dump($value);
5. });
```

使用 `value` 或 `resolve` 来创建一个成功 (fulfilled) 状态的 `promise` 对象效果跟前面用 `Future` 构造器创建的效果一样，但是写起来更加简单，不再需要把结果放入一个函数中作为返回值返回了。

创建一个失败 (rejected) 状态的 promise 对象

```
1. use Hprose\Future;
2. $e = new Exception('hprose');
3. $promise = Future\error($e); // 换成 Future\reject($e) 效果一样
4. $promise->catchError(function($reason) {
5.     var_dump($reason);
6. });
```

使用 `error` 或 `reject` 来创建一个失败 (rejected) 状态的 `promise` 对象效果跟前面用 `Future` 构造器创建的效果也一样，但是写起来也更加简单，不再需要把失败原因放入一个函数中作为异常抛出了。

注意，这里的 `error` (或 `reject`) 函数的参数并不要求必须是异常类型的对

象，但最好是使用异常类型的对象。否则你的程序很难进行调试和统一处理。

通过 Future\sync 方法来创建 promise 对象

`Future` 上提供了一个：

```
1. Future\sync($computation);
```

方法可以让我们同步的创建一个 `promise` 对象。

实际上，Hprose for PHP 的 `Future` 构造方法也是同步的，这一点跟 JavaScript 版本的有所不同。`sync` 函数跟 `Futtrue` 构造方法区别在于结果上，通过 `Future` 构造方法的结果中如果包含生成器函数或者是生成器，则生成器函数和生成器将原样返回。而通过 `sync` 函数返回的生成器函数或生成器会作为协程执行之后，返回执行结果。

通过 Future\promise 方法来创建 promise 对象

该方法的参数跟 ECMAScript 6 的 `Promise` 构造器的参数相同，不同的是，使用该方法创建 `promise` 对象时，不需要使用 `new` 关键字。另外一点不同是，该方法创建的 `promise` 对象一定是 `Future` 的实例对象，而通过 `Promise` 构造器创建的 `promise` 对象是 `Promise` 实例对象。

```
1. use Hprose\Future;
2.
3. $p = Future\promise(function($resolve, $reject) {
4.     $a = 1;
5.     $b = 2;
6.     if ($a != $b) {
7.         $resolve('OK');
8.     }
9.     else {
10.         $reject(new Exception("$a == $b"));
11.     }
12. });
13. $p->then(function($value) {
14.     var_dump($value);
15. });
```

运行结果为：

```
1. string(2) "OK"
```

通过 Promise 构造方法来创建 promise 对象

```
1. use Hprose\Promise;
2. $p = new Promise(function($resolve, $reject) { ... });
```

该构造方法的参数跟 `Future\promise` 函数的参数一致，这里就不再单独举例了。

`Promise` 构造方法的参数可以省略，省略时，行为跟 `Future` 构造方法省略参数一致。这里也不再单独举例。

通过 Completer 来创建 promise 对象

```
1. use Hprose\Completer;
2.
3. $completer = new Completer();
4. $promise = $completer->future();
5. $promise->then(function($value) {
6.     var_dump($value);
7. });
8. var_dump($completer->isCompleted());
9. $completer->complete('hprose');
10. var_dump($completer->isCompleted());
```

运行结果为：

```
1. bool(false)
2. string(6) "hprose"
3. bool(true)
```

`Future/Completer` 这套 API 来自 Dart 语言，首先通过 `Completer` 构造器创建一个 `completer` 对象，然后通过 `completer` 对象上的 `future` 方法返回 `promise` 对象。通过 `completer` 的 `complete` 方法可以设置成功值。通过 `completeError` 方法可以设置失败原因。通过 `isCompleted` 方法，可以查看当前状态是否为已完成（在这里，成功（fulfilled）或失败（rejected）都算完成状态）。

在 Hprose 2.0 之前的版本中，这是唯一可用的方法。但在 Hprose 2.0 中，该方式已经被其他方式所代替。仅为兼容旧版本而保留。

Future 类上的方法

then 方法

`then` 方法是 `Promise` 的核心和精髓所在。它有两个参数：`$onfulfill`，`$onreject`。这两个参数皆为 `callable` 类型。当它们不是 `callable` 类型时，它们将会被忽略。当 `promise` 对象状态为待定（pending）时，这两个回调方法都不会执行，直到 `promise` 对象的状态变为成功（fulfilled）或失败（rejected）。当 `promise` 对象状态为成功（fulfilled）时，`$onfulfill` 函数会被回调，参数值为成功值。当 `promise` 对象状态为失败（rejected）时，`$onreject` 函数会被回调，参数值为失败原因。

`then` 方法的返回值是一个新的 `promise` 对象，它的值由 `$onfulfill` 或 `$onreject` 的返回值或抛出的异常来决定。如果 `$onfulfill` 或 `$onreject` 在执行过程中没有抛出异常，那么新的 `promise` 对象的状态为成功（fulfilled），其值为 `$onfulfill` 或 `$onreject` 的返回值。如果这两个回调中抛出了异常，那么新的 `promise` 对象的状态将被设置为失败（rejected），抛出的异常作为新的 `promise` 对象的失败原因。

同一个 `promise` 对象的 `then` 方法可以被多次调用，其值不会因为调用 `then` 方法而改变。当 `then` 方法被多次调用时，所有的 `$onfulfill`，`$onreject` 将按照原始的调用顺序被执行。

因为 `then` 方法的返回值还是一个 `promise` 对象，因此可以使用链式调用的方式实现异步编程串行化。

当 `promise` 的成功值被设置为另一个 `promise` 对象（为了区分，将其命名为 `promise2`）时，`then` 方法中的两个回调函数得到的参数是 `promise2` 对象的最终展开值，而不是 `promise2` 对象本身。当 `promise2` 的最终展开值为成功值时，`$onfulfill` 函数会被调用，当 `promise2` 的最终展开值为失败原因时，`$onreject` 函数会被调用。

当 `promise` 的失败原因被设置为另一个 `promise` 对象时，该对象会直接作为失败原因传给 `then` 方法的 `$onreject` 回调函数。因此最好不要这样做。

关于 `then` 方法的用法，这里不单独举例，您将在其它的例子中看到它的用法。

done 方法

跟 `then` 方法类似，但 `done` 方法没有返回值，不支持链式调用，因此在 `done` 方法的回调函数中，通常不会返回值。

如果在 `done` 方法的回调中发生异常，会直接抛出，并且无法被捕获。

因此，如果您不是在写单元测试，最好不要使用 `done` 方法。

fail 方法

该方法是 `done(null, $onreject)` 的简化方法。

如果您不是在写单元测试，最好不要使用 `fail` 方法。

catchError 方法

```
1. $promise->catchError($onreject);
```

该方法是 `then(null, $onreject)` 的简化写法。

```
1. $promise->catchError($onreject, $test);
```

该方法第一个参数 `$onreject` 跟上面的相同，第二个参数 `$test` 是一个测试函数（`callable` 类型）。当该测试函数返回值为 `true` 时，`$onreject` 才会执行。

```
1. use Hprose\Future;
2.
3. $p = Future\reject(new OutOfRangeException());
4.
5. $p->catchError(function($reason) { return 'this is a OverflowException'; },
6.               function($reason) { return $reason instanceof OverflowException;
7.               })
8.   ->catchError(function($reason) { return 'this is a OutOfRangeException'; },
9.               function($reason) { return $reason instanceof
    OutOfRangeException; })
10.  ->then(function($value) { var_dump($value); });
```

输出结果为：

```
1. string(29) "this is a OutOfRangeException"
```

resolve 方法

该方法可以将状态为待定（pending）的 `promise` 对象变为成功（fulfilled）状态。

该方法的参数值可以为任意类型。

reject 方法

该方法可以将状态为待定 (pending) 的 `promise` 对象变为失败 (rejected) 状态。

该方法的参数值可以为任意类型，但通常只使用异常类型。

inspect 方法

该方法返回当前 `promise` 对象的状态。

如果当前状态为待定 (pending)，返回值为：

```
1. array('state' => 'pending')
```

如果当前状态为成功 (fulfilled)，返回值为：

```
1. array('state' => 'fulfilled', 'value' => $promise->value)
```

如果当前状态为失败 (rejected)，返回值为：

```
1. array('state' => 'rejected', 'reason' => $promise->reason);
```

whenComplete 方法

有时候，你不但想要在成功 (fulfilled) 时执行某段代码，而且在失败 (rejected) 时也想执行这段代码，那你可以使用 `whenComplete` 方法。该方法的参数为一个无参回调函数。该方法执行后会返回一个新的 `promise` 对象，除非在回调函数中抛出异常，否则返回的 `promise` 对象的值跟原 `promise` 对象的值相同。

```
1. use Hprose\Future;
2.
3. $p1 = Future\resolve('resolve hprose');
4.
5. $p1->whenComplete(function() {
6.     var_dump('p1 complete');
7. })->then(function($value) {
8.     var_dump($value);
9. });
10.
11. $p2 = Future\reject(new Exception('reject thrift'));
12.
13. $p2->whenComplete(function() {
14.     var_dump('p2 complete');
15. })->catchError(function($reason) {
16.     var_dump($reason->getMessage());
17. });
```



```

18.
19. $p3 = Future\resolve('resolve protobuf');
20.
21. $p3->whenComplete(function() {
22.     var_dump('p3 complete');
23.     throw new Exception('reject protobuf');
24. })->catchError(function($reason) {
25.     var_dump($reason->getMessage());
26. });

```

运行结果如下：

```

1. string(11) "p1 complete"
2. string(14) "resolve hprose"
3. string(11) "p2 complete"
4. string(13) "reject thrift"
5. string(11) "p3 complete"
6. string(15) "reject protobuf"

```

complete 方法

该方法的回调函数 `oncomplete` 在不论成功还是失败的情况下都会执行，并且支持链式调用。相当于：`then(oncomplete, oncomplete)` 的简化写法。

always 方法

该方法的回调函数 `oncomplete` 在不论成功还是失败的情况下都会执行，但不支持链式调用。相当于：`done(oncomplete, oncomplete)` 的简化写法。

如果您不是在写单元测试，最好不要使用 `always` 方法。

fill 方法

将当前 `promise` 对象的值充填到参数所表示的 `promise` 对象中。

tap 方法

```

1. $promise->tap($onfulfilledSideEffect);

```

以下两种写法是等价的：

```

1. $promise->then(function($result) use ($onfulfilledSideEffect) {
2.     call_user_func($onfulfilledSideEffect, $result);
3.     return result;
4. });
5.
6. $promise->tap($onfulfilledSideEffect);

```

显然使用 `tap` 方法写起来更简单。

spread 方法

```

1. $promise->spread($onfulfilledArray);

```

以下两种写法是等价的：

```

1. $promise->then(function($array) use ($onfulfilledArray) {
2.     return call_user_func_array($onfulfilledArray, $array);
3. });
4.
5. $promise->spread($onfulfilledArray);

```

each 方法

```

1. $promise->each($callback);

```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以对该数组进行遍历。`$callback` 回调方法的格式如下：

```

1. function callback(mixed $value, mixed $key, array $array);

```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. function dumpArray($value, $key) {
4.     var_dump("a[$key] = $value");
5. }
6.
7. $a1 = Future\value(array(2, Future\value(5), 9));
8. $a2 = Future\value(array('name' => Future\value('Tom'), 'age' =>
    Future\value(18)));
9. $a1->each('dumpArray');
10. $a2->each('dumpArray');

```

输出结果为：

```

1. string(8) "a[0] = 2"
2. string(8) "a[1] = 5"
3. string(8) "a[2] = 9"
4. string(13) "a[name] = Tom"
5. string(11) "a[age] = 18"

```

every 方法

```
1. $promise->every($callback);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以遍历数组中的每一个元素并执行回调 `$callback`，当所有 `$callback` 的返回值都为 `true` 时，结果为 `true`，否则为 `false`。`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 10;
7. }
8.
9. $a1 = Future\value(array(12, Future\value(5), 8, Future\value(130), 44));
10. $a2 = Future\value(array(12, Future\value(54), 18, Future\value(130), 44));
11. $dump($a1->every('isBigEnough')); // false
12. $dump($a2->every('isBigEnough')); // true

```

运行结果如下：

```

1. bool(false)
2. bool(true)

```

some 方法

```
1. $promise->some($callback);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以遍历数组中的每一个元素并执行回调 `$callback`，当任意一个 `$callback` 的返回值为 `true` 时，结果为 `true`，否则为 `false`。`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```
1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 10;
7. }
8.
9. $a1 = Future\value(array(12, Future\value(5), 8, Future\value(130), 44));
10. $a2 = Future\value(array(1, Future\value(5), 8, Future\value(1), 4));
11. $dump($a1->some('isBigEnough')); // true
12. $dump($a2->some('isBigEnough')); // false
```

运行结果如下：

```
1. bool(true)
2. bool(false)
```

filter 方法

```
1. $promise->filter($callback, $preserveKeys = false);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的返回值为 `true` 的元素所组成的数组将作为 `filter` 返回结果的 `promise` 对象所包含的值。当参数 `$preserveKeys` 为 `true` 时，结果数组中的元素所对应的 `key` 保持原来的 `key`，否则将返回以 0 为起始下标的连续数字下标的数组。`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 8;
7. }
8.
9. $a = Future\value(array(
10.     'Tom' => 8,
11.     'Jerry' => Future\value(5),
12.     'Spike' => 10,
13.     'Tyke' => 3
14. ));
15.
16. $dump($a->filter('isBigEnough'));
17. $dump($a->filter('isBigEnough', true));

```

运行结果为：

```

1. array(2) {
2.     [0]=>
3.     int(8)
4.     [1]=>
5.     int(10)
6. }
7. array(2) {
8.     ["Tom"]=>
9.     int(8)
10.    ["Spike"]=>
11.    int(10)
12. }

```

map 方法

```
1. $promise->map($callback);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的返回值所组成的数组将作为 `map` 返回结果的 `promise` 对象所包含的值。`$callback` 回调方法的格式如下：

```
1. mixed callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```
1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $a = Future\value(array(1, Future\value(4), 9));
6. $dump($a->map('sqrt'));
```

运行结果为：

```
1. array(3) {
2.     [0]=>
3.     float(1)
4.     [1]=>
5.     float(2)
6.     [2]=>
7.     float(3)
8. }
```

reduce 方法

```
1. $promise->reduce($callback, $initial = NULL);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的第一个参数为 `$initial` 的值或者上一次调用的返回值。最后一次 `$callback` 的返回结果作为 `promise` 对象所包含的值。`$callback` 回调方法的格式如下：

```
1. mixed callback(mixed $carry, mixed $item);
```

关于该方法的更多描述可以参见 PHP 手册中的 `array_reduce` 方法。

```
1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $numbers = Future\value(array(Future\value(0), 1, Future\value(2), 3,
    Future\value(4)));
6.
```

```

7. function add($a, $b) {
8.     return $a + $b;
9. }
10.
11. $dump($numbers->reduce('add'));
12. $dump($numbers->reduce('add', 10));
13. $dump($numbers->reduce('add', Future\value(20)));

```

运行结果如下：

```

1. int(10)
2. int(20)
3. int(30)

```

search 方法

```
1. $promise->search($searchElement, $strict = false);
```

如果 `promise` 对象中包含的是一个数组，那么使用该方法可以在 `promise` 对象所包含的数组中查找 `$searchElement` 元素，返回值以 `promise` 对象形式返回，如果找到，返回的 `promise` 对象中将包含该元素对应的 `key`，否则为 `false`。当 `$strict` 为 `true` 时，使用 `===` 运算符进行相等测试。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $numbers = Future\value(array(Future\value(0), 1, Future\value(2), 3,
    Future\value(4)));
6.
7. $dump($numbers->search(2));
8. $dump($numbers->search(Future\value(3)));
9. $dump($numbers->search(true));
10. $dump($numbers->search(true, true));

```

运行结果如下：

```

1. int(2)
2. int(3)
3. int(1)
4. bool(false)

```

includes 方法

```
1. $promise->includes($searchElement, $strict = false);
```

该方法同 `search` 方法类似，只是在找到的情况下，仅仅返回包含 `true` 的 `promise` 对象。

魔术方法 `call` 和 `get`

`Future` 类上还定义了 `get` 和 `call` 这两个魔术方法，当 `promise` 对象中包含的值为 `object` 对象时，可以直接获取它的属性和调用它的方法。但是需要注意的是，返回的属性值是一个 `promise` 对象。调用方法的返回值也是一个 `promise` 对象，而且调用方法时，参数也可以是 `promise` 对象，即使原来的方法并不支持 `promise` 参数。因为在实际调用时，`__call` 会自动将 `promise` 的参数值转换为实际包含的值进行调用。

这在一定程度上可以使得异步代码在编写时看上去像是同步代码。

Future 类上的常量和属性

常量

```
1. const PENDING = 0;  
2. const FULFILLED = 1;  
3. const REJECTED = 2;
```

这三个常量表示 `promise` 对象的状态。`PENDING` 表示结果待定。`FULFILLED` 表示成功。`REJECTED` 表示失败。

Hprose\Future 名空间中的函数

isFuture 函数

```
1. bool isFuture(mixed $obj);
```

判断参数 `$obj` 是否为 `Future` 对象。

例如：


```
1. use Hprose\Future;  
2.  
3. var_dump(Future\isFuture(123));  
4. var_dump(Future\isFuture(Future\value(123)));
```

运行结果为：

```
1. bool(false)  
2. bool(true)
```

toFuture 函数

```
1. Future toFuture(mixed $obj);
```

如果 `$obj` 为 `Future` 对象则原样返回，否则返回 `Future\value($obj)`。

toPromise 函数

```
1. Future toPromise(mixed $obj);
```

该方法同 `toFuture` 函数类似。

如果 `$obj` 为生成器，则该方法将把生成器作为协程执行并以 `promise` 对象返回该协程的执行结果。

其他情况跟 `toFuture` 一致。

all 函数

```
1. Future all(mixed $array);
```

该方法的参数 `$array` 为数组或者值为数组的 `promise` 对象。该方法返回一个 `promise` 对象，该 `promise` 对象会在数组参数内的所有 `promise` 都被设置为成功（fulfilled）状态时，才被设置为成功（fulfilled）状态，其值为数组参数中所有 `promise` 对象的最终展开值组成的数组，其数组元素与原数组元素一一对应。

join 函数

```
1. Future join(mixed arg1[, mixed arg1[, ...]]);
```

该方法的功能同 `all` 方法类似，但它与 `all` 方法的参数不同，我们来举例看一下它们的差别：

```
1. use Hprose\Future;
2.
3. Future\all(array(1, Future\value(2), 3))->then(function($value) {
4.     var_dump($value);
5. });
6.
7. Future\join(1, Future\value(2), 3)->then(function($value) {
8.     var_dump($value);
9. });
```

运行结果为：

```
1. array(3) {
2.     [0]=>
3.     int(1)
4.     [1]=>
5.     int(2)
6.     [2]=>
7.     int(3)
8. }
9. array(3) {
10.    [0]=>
11.    int(1)
12.    [1]=>
13.    int(2)
14.    [2]=>
15.    int(3)
16. }
```

race 函数

```
1. Future race(mixed $array);
```

该方法返回一个 `promise` 对象，这个 `promise` 在数组参数中的任意一个 `promise` 被设置为成功 (fulfilled) 或失败 (rejected) 后，立刻以相同的成功值被设置为成功 (fulfilled) 或以相同的失败原因被设置为失败

(rejected)。

any 函数

```
1. Future any(mixed $array);
```

该方法是 `race` 函数的改进版。

对于 `race` 函数，如果输入的数组为空，返回的 `promise` 对象将永远保持为待定 (pending) 状态。

而对于 `any` 函数，如果输入的数组为空，返回的 `promise` 对象将被设置为失败状态，失败原因是一个 `RangeException` 对象。

对于 `race` 函数，数组参数中的任意一个 `promise` 被设置为成功 (fulfilled) 或失败 (rejected) 后，返回的 `promise` 对象就会被设定为成功 (fulfilled) 或失败 (rejected) 状态。

而对于 `any` 函数，只有当数组参数中的所有 `promise` 被设置为失败状态时，返回的 `promise` 对象才会被设定为失败状态。否则，返回的 `promise` 对象被设置为第一个被设置为成功 (fulfilled) 状态的成功值。

这两个函数通常跟计时器或者并发请求一起使用，用来获取最早完成的结果。

settle 函数

```
1. Future settle(mixed $array)
```

该方法返回一个 `promise` 对象，该 `promise` 对象会在数组参数内的所有 `promise` 都被设置为成功 (fulfilled) 状态或失败 (rejected) 状态时，才被设置为成功 (fulfilled) 状态，其值为数组参数中所有 `promise` 对象的 `inspect` 方法返回值，其数组元素与原数组元素一一对应。

例如：

```
1. use Hprose\Future;
2.
3. $p1 = Future\resolve(3);
4. $p2 = Future\reject(new Exception("x"));
5.
6. Future\settle(array(true, $p1, $p2))->then('print_r');
```

输出结果为：

```
1. Array
```

```

2. (
3.   [0] => Array
4.     (
5.       [state] => fulfilled
6.       [value] => 1
7.     )
8.
9.   [1] => Array
10.    (
11.      [state] => fulfilled
12.      [value] => 3
13.    )
14.
15.   [2] => Array
16.    (
17.      [state] => rejected
18.      [reason] => Exception Object
19.      (
20.        [message:protected] => x
21.        [string:Exception:private] =>
22.        [code:protected] => 0
23.      ...

```

注：上面输出中 `...` 表示省略掉的内容。

run 函数

```
1. Future run(callable $handler[, mixed $arg1[, mixed $arg2[, ...]]]);
```

`run` 方法的作用是执行 `$handler` 函数并返回一个包含执行结果的 `promise` 对象，`$handler` 的参数分别为 `$arg1`，`$arg2`，...。参数可以是普通值，也可以是 `promise` 对象，如果是 `promise` 对象，则等待其变为成功（`fulfilled`）状态时再将其成功值代入 `handler` 函数。如果变为失败（`rejected`）状态，`run` 返回的 `promise` 对象被设置为该失败原因。如果参数中，有多个 `promise` 对象变为失败（`rejected`）状态，则第一个变为失败状态的 `promise` 对象的失败原因被设置为 `run` 返回的 `promise` 对象的失败原因。当参数中的 `promise` 对象都变为成功（`fulfilled`）状态时，`$handler` 函数才会执行，如果在 `$handler` 执行的过程中，抛出了异常，则该异常作为 `run` 返回的 `promise` 对象的失败原因。如果没有异常，则 `$handler` 函数的返回值，作为 `run` 返回的 `promise` 对象的成功值。

例如：

```

1. use Hprose\Future;
2.
3. function add($a, $b) {
4.     return $a + $b;
5. }
6.
7. $p1 = Future\resolve(3);
8.
9. Future\run('add', 2, $p1)->then('var_dump');

```

输出结果为：

```

1. int(5)

```

wrap 函数

```

1. mixed wrap(mixed $handler);

```

`run` 函数虽然可以将 `promise` 参数带入普通函数执行并得到结果，但是不方便复用。`wrap` 函数可以很好的解决这个问题。

`wrap` 函数的参数可以是一个 `callable` 对象，也可以是一个普通对象。

如果参数是一个 `callable` 数据（比如函数，方法），则返回值是一个闭包对象，它是一个包装好的函数，该函数的执行方式跟使用 `Future\run` 的效果一样。

如果参数是一个普通对象，则返回值是一个 `\Hprose\Future\Wrapper` 对象。你可以像存取源对象一样存取它，但是它上面的方法的执行方式跟使用 `Future\run` 的效果一样。

如果参数是一个 `callable` 对象，则返回值是一个 `\Hprose\Future\CallableWrapper` 对象。它跟 `\Hprose\Future\Wrapper` 对象类似，只是该对象本身也可以被直接作为函数调用，而且执行方式跟使用 `Future\run` 的效果一样。

例如：

```

1. use Hprose\Future;
2.
3. class Test {
4.     function add($a, $b) {
5.         return $a + $b;
6.     }

```

```

7.     function sub($a, $b) {
8.         return $a - $b;
9.     }
10.    function mul($a, $b) {
11.        return $a * $b;
12.    }
13.    function div($a, $b) {
14.        return $a / $b;
15.    }
16. }
17.
18. $var_dump = Future\wrap('var_dump');
19.
20. $test = Future\wrap(new Test());
21.
22. $var_dump($test->add(1, Future\value(2)));
23. $var_dump($test->sub(Future\value(1), 2));
24. $var_dump($test->mul(Future\value(1), Future\value(2)));
25. $var_dump($test->div(1, 2));

```

该程序输出结果为：

```

1. int(3)
2. int(-1)
3. int(2)
4. float(0.5)

```

each 函数

```
1. void each(mixed $array, callable $callback);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数对该数组中的每个元素的展开值进行遍历。返回值是一个 `promise` 对象。如果参数数组中的 `promise` 对象为失败 (rejected) 状态，则该方法返回的 `promise` 对象被设置为失败 (rejected) 状态，且设为相同失败原因。如果在 `$callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败 (rejected) 状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. function callback(mixed $value, mixed $key, array $array);
```

后两个参数是可选的。

```
1. use Hprose\Future;
2.
3. function dumpArray($value, $key) {
4.     var_dump("a[$key] = $value");
5. }
6.
7. $a1 = array(2, Future\value(5), 9);
8. $a2 = array('name' => Future\value('Tom'), 'age' => Future\value(18));
9. Future\each($a1, 'dumpArray');
10. Future\each($a2, 'dumpArray');
```

输出结果为：

```
1. string(8) "a[0] = 2"
2. string(8) "a[1] = 5"
3. string(8) "a[2] = 9"
4. string(13) "a[name] = Tom"
5. string(11) "a[age] = 18"
```

every 函数

```
1. Future<bool> every(mixed $array, callable $callback);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以遍历数组中的每一个元素并执行回调 `$callback`，当所有 `$callback` 的返回值都为 `true` 时，结果为 `true`，否则为 `false`。该函数返回值是一个 `promise` 对象。如果参数数组中的 `promise` 对象为失败（rejected）状态，则该方法返回的 `promise` 对象被设置为失败（rejected）状态，且设为相同失败原因。如果在 `callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败（rejected）状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 10;
7. }
8.
9. $a1 = array(12, Future\value(5), 8, Future\value(130), 44);
10. $a2 = array(12, Future\value(54), 18, Future\value(130), 44);
11. $a3 = Future\value($a1);
12. $a4 = Future\value($a2);
13. $dump(Future\every($a1, 'isBigEnough')); // false
14. $dump(Future\every($a2, 'isBigEnough')); // true
15. $dump(Future\every($a3, 'isBigEnough')); // false
16. $dump(Future\every($a4, 'isBigEnough')); // true

```

运行结果如下：

```

1. bool(false)
2. bool(true)
3. bool(false)
4. bool(true)

```

some 函数

```
1. Future<bool> some(mixed $array, callable $callback);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以遍历数组中的每一个元素并执行回调 `$callback`，当任意一个 `$callback` 的返回值为 `true` 时，结果为 `true`，否则为 `false`。如果参数数组中的 `promise` 对象为失败（rejected）状态，则该方法返回的 `promise` 对象被设置为失败（rejected）状态，且设为相同失败原因。如果在 `callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败（rejected）状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。


```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 10;
7. }
8.
9. $a1 = array(12, Future\value(5), 8, Future\value(130), 44);
10. $a2 = array(1, Future\value(5), 8, Future\value(1), 4);
11. $a3 = Future\value($a1);
12. $a4 = Future\value($a2);
13. $dump(Future\some($a1, 'isBigEnough')); // true
14. $dump(Future\some($a2, 'isBigEnough')); // false
15. $dump(Future\some($a3, 'isBigEnough')); // true
16. $dump(Future\some($a4, 'isBigEnough')); // false

```

运行结果如下：

```

1. bool(true)
2. bool(false)
3. bool(true)
4. bool(false)

```

filter 函数

```
1. Future<array> filter(mixed $array, callable $callback);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的返回值为 `true` 的元素所组成的数组将作为 `filter` 返回结果的 `promise` 对象所包含的值。当参数 `$preserveKeys` 为 `true` 时，结果数组中的元素所对应的 `key` 保持原来的 `key`，否则将返回以 0 为起始下标的连续数字下标的数组。如果参数数组中的 `promise` 对象为失败 (rejected) 状态，则该方法返回的 `promise` 对象被设置为失败 (rejected) 状态，且设为相同失败原因。如果在 `callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败 (rejected) 状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. bool callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. function isBigEnough($value) {
6.     return $value >= 8;
7. }
8.
9. $a1 = array(12, Future\value(5), 8, Future\value(130), 44);
10. $a2 = Future\value($a1);
11. $dump(Future\filter($a1, 'isBigEnough'));
12. $dump(Future\filter($a2, 'isBigEnough'));
13.
14. $a3 = array('Tom' => 8, 'Jerry' => Future\value(5), 'Spike' => 10, 'Tyke' =>
15.     3);
16. $a4 = Future\value($a3);
17. $dump(Future\filter($a3, 'isBigEnough'));
18. $dump(Future\filter($a3, 'isBigEnough', true));
19. $dump(Future\filter($a4, 'isBigEnough', true));

```

运行结果为：

```

1. array(4) {
2.     [0]=>
3.     int(12)
4.     [1]=>
5.     int(8)
6.     [2]=>
7.     int(130)
8.     [3]=>
9.     int(44)
10. }
11. array(4) {
12.     [0]=>
13.     int(12)
14.     [1]=>
15.     int(8)
16.     [2]=>
17.     int(130)
18.     [3]=>
19.     int(44)
20. }
21. array(2) {
22.     [0]=>

```

```

23.     int(8)
24.     [1]=>
25.     int(10)
26. }
27. array(2) {
28.     ["Tom"]=>
29.     int(8)
30.     ["Spike"]=>
31.     int(10)
32. }
33. array(2) {
34.     ["Tom"]=>
35.     int(8)
36.     ["Spike"]=>
37.     int(10)
38. }

```

map 函数

```
1. Future<array> map(mixed $array, callable $callback);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的返回值所组成的数组将作为 `map` 返回结果的 `promise` 对象所包含的值。该函数返回值是一个 `promise` 对象。如果参数数组中的 `promise` 对象为失败（rejected）状态，则该方法返回的 `promise` 对象被设置为失败（rejected）状态，且设为相同失败原因。如果在 `callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败（rejected）状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. mixed callback(mixed $value[, mixed $key[, array $array]]);
```

后两个参数是可选的。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $a = array(1, Future\value(4), 9);
6. $dump(Future\map($a, 'sqrt'));

```

运行结果为：

```
1. array(3) {
2.     [0]=>
3.     float(1)
4.     [1]=>
5.     float(2)
6.     [2]=>
7.     float(3)
8. }
```

reduce 函数

```
1. Future<mixed> reduce(mixed $array, callable $callback[, $initial = NULL]);
```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以遍历数组中的每一个元素并执行回调 `$callback`，`$callback` 的第一个参数为 `$initial` 的值或者上一次调用的返回值。最后一次 `$callback` 的返回结果作为 `promise` 对象所包含的值。该函数返回值是一个 `promise` 对象。如果参数数组中的 `promise` 对象为失败（rejected）状态，则该方法返回的 `promise` 对象被设置为失败（rejected）状态，且设为相同失败原因。如果在 `callback` 回调中抛出了异常，则该方法返回的 `promise` 对象也被设置为失败（rejected）状态，失败原因被设置为抛出的异常值。

`$callback` 回调方法的格式如下：

```
1. mixed callback(mixed $carry, mixed $item);
```

关于该方法的更多描述可以参见 PHP 手册中的 `array_reduce` 方法。

```
1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $numbers = array(Future\value(0), 1, Future\value(2), 3, Future\value(4));
6.
7. function add($a, $b) {
8.     return $a + $b;
9. }
```

```

10.
11. $dump(Future\reduce($numbers, 'add'));
12. $dump(Future\reduce($numbers, 'add', 10));
13. $dump(Future\reduce($numbers, 'add', Future\value(20)));

```

运行结果如下：

```

1. int(10)
2. int(20)
3. int(30)

```

search 方法

```

1. Future<mixed> search(mixed $array, mixed $searchElement[, bool $strict = false]);

```

参数 `$array` 可以是一个包含数组的 `promise` 对象，也可以是一个包含有 `promise` 对象的数组。

该函数可以在 `promise` 对象所包含的数组中查找 `$searchElement` 元素，返回值以 `promise` 对象形式返回，如果找到，返回的 `promise` 对象中将包含该元素对应的 `key`，否则为 `false`。当 `$strict` 为 `true` 时，使用 `===` 运算符进行相等测试。该函数返回值是一个 `promise` 对象。如果参数数组中的 `promise` 对象为失败（rejected）状态，则该方法返回的 `promise` 对象被设置为失败（rejected）状态，且设为相同失败原因。

```

1. use Hprose\Future;
2.
3. $dump = Future\wrap('var_dump');
4.
5. $numbers = array(Future\value(0), 1, Future\value(2), 3, Future\value(4));
6.
7. $dump(Future\search($numbers, 2));
8. $dump(Future\search($numbers, Future\value(3)));
9. $dump(Future\search($numbers, true));
10. $dump(Future\search($numbers, true, true));

```

运行结果如下：

```

1. int(2)
2. int(3)

```

```
3. int(1)
4. bool(false)
```

includes 函数

```
1. Future<bool> includes(mixed $array, mixed $searchElement[, bool $strict = false]);
```

该方法同 `search` 方法类似，只是在找到的情况下，仅仅返回包含 `true` 的 `promise` 对象。

Hprose\Promise 类和名空间

`Hprose\Promise` 类是 `Hprose\Future` 的子类，除了构造方法跟 `Hprose\Future` 有所不同之外，其它方法都完全一致。

`Hprose\Promise` 名空间下的方法也都跟 `Hprose\Future` 名空间下的方法相同。只是有一个函数名不同，即 `Hprose\Promise\isPromise`，但它的功能跟 `Hprose\Future\isFuture` 是完全一样的。

因此你也可以完全使用 `Hprose\Promise` 来代替 `Hprose\Future`。

原文: <https://github.com/hprose/hprose-php/wiki/03-Promise-%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B>

04 协程

基本用法

PHP 5.5 引入了 `Generator`，通过封装之后，可以作为协程来进行使用。

Hprose 也提供了对 `Generator` 的一个封装，并且跟 `Promise` 相结合之后，可以实现异步代码同步化。

让我们来看一个例子：

```
1. use \Hprose\Future;
2. use \Hprose\Http\Client;
3.
4. Future\co(function() {
5.     $test = new Client("http://hprose.com/example/");
6.     var_dump((yield $test->hello("hprose")));
7.     $a = $test->sum(1, 2, 3);
8.     $b = $test->sum(4, 5, 6);
9.     $c = $test->sum(7, 8, 9);
10.    var_dump((yield $test->sum($a, $b, $c)));
11.    var_dump((yield $test->hello("world")));
12. });
```

`Future\co`（也可以用 `Promise\co`）就是一个协程封装函数。它的功能以协程的方式来执行生成器函数。该方法允许带入参数执行。

在上面的例子中，`$test` 是一个 Hprose 的异步 Http 客户端。Hprose 2.0 的默认客户端为异步客户端，而不是同步客户端，这一点与 1.x 不同。

所以 `$test->hello` 和 `$test->sum` 两个调用的返回值实际上是一个 `promise` 对象。而 `yield` 关键字在这里的作用就是，可以等待调用完成并返回 `promise` 所包含的值，如果 `promise` 的最后的状态为 `REJECTED`，那么 `yield` 将抛出一个异常，异常的值为 `promise` 对象中的 `reason` 属性值。

在上面的调用中，`$a`，`$b`，`$c` 三个变量都是 `promise` 对象，而 `$test->sum` 可以直接接受 `promise` 参数作为调用参数，当 `$a`，`$b`，`$c` 三个 `promise` 对象的状态都变为 `FULFILLED` 状态时，`$test->sum($a, $b, $c)` 才会真正的开始调用。而获取 `$a`，`$b`，`$c` 的三个调用是异步并发执行的。

上面程序的执行结果为：

```

1. string(12) "Hello hprose"
2. int(45)
3. string(11) "Hello world"

```

从结果我们可以看出，`co` 函数和 `yield` 的结合可以很方便的让异步程序编写同步化。这也是 Hprose 2.0 最有特色的改进之一。

虽然上面用 Hprose 远程调用来举例，但是 `co` 函数所实现的协程不是只对 Hprose 远程调用有效，而是对任何返回 `promise` 的对象都有效。所以，即使你不使用 Hprose 远程调用，也可以使用 `co` 函数和 Promise 来进行异步代码的同步化编写。

多协程并发

我们前面说过，如果在同一个协程内进行远程调用，如果不加 `yield` 关键字，多个远程调用就是并发执行的。加上 `yield` 关键字，就会变成顺序执行。

那么当开两个或多个协程时，结果是什么样子呢？我们来看一个例子：

```

1. use \Hprose\Future;
2. use \Hprose\Http\Client;
3.
4. $test = new Client("http://hprose.com/example/");
5.
6. Future\co(function() use ($test) {
7.     for ($i = 0; $i < 5; $i++) {
8.         var_dump((yield $test->hello("1-" . $i)));
9.     }
10. });
11.
12. Future\co(function() use ($test) {
13.     for ($i = 0; $i < 5; $i++) {
14.         var_dump((yield $test->hello("2-" . $i)));
15.     }
16. });

```

我们运行该程序之后，可以看到如下结果：

```

1. string(9) "Hello 2-0"
2. string(9) "Hello 1-0"
3. string(9) "Hello 1-1"
4. string(9) "Hello 2-1"
5. string(9) "Hello 2-2"
6. string(9) "Hello 1-2"

```



```

7. string(9) "Hello 1-3"
8. string(9) "Hello 2-3"
9. string(9) "Hello 2-4"
10. string(9) "Hello 1-4"

```

这个运行结果并不唯一，我们有可能看到不同顺序的输出，但是有一点可以保证，就是 `Hello-1-X` 中的 `X` 是按照顺序输出的，而 `Hello-2-Y` 中的 `Y` 也是按照顺序输出的。

也就是说，每个协程内的语句是按照顺序执行的，而两个协程确是并行执行的。

不过有一点要注意，上面的例子跟第一个例子有一点不同，那就是我们把 `$test` 客户端的创建拿到了协程外面。

如果像第一个例子那样放在协程中，我们就看不到这样的并发执行结果了。原因是，这里的每个客户端都有一个自己独立的事件循环，只有当一个事件循环执行完之后，才会执行另一个事件循环。

但如果换成 `Hprose` 的 `Swoole` 的客户端，则不管是放在里面还是外面，看到的都是并发执行的结果，原因是 `Swoole` 客户端的事件循环是统一的。而且 `Swoole` 的事件循环一旦开始，如果不手动执行 `swoole_event_exit`，事件循环不会结束，你也就不会看到程序退出。在使用 `Swoole` 客户端时，需要注意这一点。

协程的参数和返回值

`co` 函数允许传参给协程。

而且 `co` 函数本身的返回值也是一个 `promise` 对象。它的值在 `PHP 5` 和 `PHP 7` 中略有差别。

`PHP 5` 的生成器函数本身不允许使用 `return` 返回值。例如：

```

1. function test() {
2.     $x = (yield 1);
3.     return $x;
4. }

```

这样的代码是非法的（但你不用担心因为写了这样的代码而被抓去坐牢 😊 ">）。但是在 `PHP 7` 中，这种写法是被允许的，但这个返回值跟普通函数的返回值是不同的。`PHP 7` 中为 `Generator` 对象提供了一个 `getReturn` 方法来专门获取这个返回值。

`co` 函数的返回值在 `PHP 5` 中是最后一次执行的 `yield` 的返回值的

`promise` 包装。在 PHP 7 中，如果没有 `return` 语句，或者 `return` 语句没有返回值（或者返回值为 `NULL`），那么，`co` 函数的返回值跟 PHP 5 相同。如果在 PHP 7 中，使用了 `return` 语句并且有返回值，比如像上面那个 `test` 函数，那么返回值为 `return` 语句返回值的 `promise` 包装。

因此如果你的代码是按照 PHP 5 的方式编写的，那么执行的效果在 PHP 5 和 PHP 7 中是相同的，如果你是按照 PHP 7 的方式单独编写的，那么你也能够得到你希望得到的 PHP 7 的返回值。因此，`co` 函数既做到了对旧版本的兼容性，又做到了对新版本特殊功能的支持。

因为 `co` 函数的结果本身也是一个 `promise` 对象，因此，你也可以在另外一个协程中来 `yield` `co` 函数的执行结果。

下面这个例子演示了传参和 `co` 函数返回值的使用：

```

1. use \Hprose\Future;
2. use \Hprose\Http\Client;
3.
4. $test = new Client("http://hprose.com/example/");
5.
6. function hello($n, $test) {
7.     $result = array();
8.     for ($i = 0; $i < 5; $i++) {
9.         $result[] = $test->hello("$n-$i");
10.    }
11.    yield Future\all($result);
12. }
13.
14. Future\co(function() use ($test) {
15.     $result = (yield Future\co(function($test) {
16.         $result = array();
17.         for ($i = 0; $i < 3; $i++) {
18.             $result[] = Future\co('hello', $i, $test);
19.         }
20.         yield Future\all($result);
21.     }, $test));
22.     var_dump($result);
23. });
24. });

```

该程序执行结果为：

```

1. array(3) {
2.     [0]=>
3.     array(5) {
4.         [0]=>
5.         string(9) "Hello 0-0"
6.         [1]=>

```

```

7.     string(9) "Hello 0-1"
8.     [2]=>
9.     string(9) "Hello 0-2"
10.    [3]=>
11.    string(9) "Hello 0-3"
12.    [4]=>
13.    string(9) "Hello 0-4"
14.  }
15.  [1]=>
16.  array(5) {
17.    [0]=>
18.    string(9) "Hello 1-0"
19.    [1]=>
20.    string(9) "Hello 1-1"
21.    [2]=>
22.    string(9) "Hello 1-2"
23.    [3]=>
24.    string(9) "Hello 1-3"
25.    [4]=>
26.    string(9) "Hello 1-4"
27.  }
28.  [2]=>
29.  array(5) {
30.    [0]=>
31.    string(9) "Hello 2-0"
32.    [1]=>
33.    string(9) "Hello 2-1"
34.    [2]=>
35.    string(9) "Hello 2-2"
36.    [3]=>
37.    string(9) "Hello 2-3"
38.    [4]=>
39.    string(9) "Hello 2-4"
40.  }
41. }

```

在这个程序里，所有的调用都是并发执行的，最后一次 `yield` 汇集最终所有结果。`yield` 语句在这里同时扮演了 `return` 的角色。

wrap 包装函数和 yield 的区别

我们在 [Promise 异步编程](#) 一章中，介绍了功能强大的 `wrap` 函数。通过它包装的函数可以直接将 `promise` 对象像普通参数一样带入函数执行。但是要注意

意，`wrap` 包装之后的函数虽然看上去像是同步的，但是实际上是异步执行的。当你有多个 `wrap` 包装的函数顺序执行的时候，实际上并不保证执行顺序按照书写顺序来。而 `yield` 则是同步的，它一定会保证 `yield` 语句的执行顺序。

我们来看一个例子：

```
1. use \Hprose\Future;
2. use \Hprose\Http\Client;
3.
4. $test = new Client("http://hprose.com/example/");
5. Future\co(function() use ($test) {
6.     for ($i = 0; $i < 5; $i++) {
7.         var_dump((yield $test->hello("1-" . $i)));
8.     }
9.     $var_dump = Future\wrap('var_dump');
10.    for ($i = 0; $i < 5; $i++) {
11.        $var_dump($test->hello("2-" . $i));
12.    }
13.    for ($i = 0; $i < 5; $i++) {
14.        var_dump((yield $test->hello("3-" . $i)));
15.    }
16. });
```

运行该程序之后，执行结果为：

```
1. string(9) "Hello 1-0"
2. string(9) "Hello 1-1"
3. string(9) "Hello 1-2"
4. string(9) "Hello 1-3"
5. string(9) "Hello 1-4"
6. string(9) "Hello 2-0"
7. string(9) "Hello 2-2"
8. string(9) "Hello 3-0"
9. string(9) "Hello 2-1"
10. string(9) "Hello 2-3"
11. string(9) "Hello 2-4"
12. string(9) "Hello 3-1"
13. string(9) "Hello 3-2"
14. string(9) "Hello 3-3"
15. string(9) "Hello 3-4"
```

这个结果可能每次执行都不一样。

但是，`Hello 1-X` 始终都是按照顺序输出的，而且始终都是在 `Hello 2-Y` 和 `Hello 3-Z` 之前输出的。

`Hello 2-Y` 的输出则不是按照顺序输出的（虽然偶尔结果也是按照顺序输出，但这一点并不能保证），而且它甚至还会穿插在 `Hello 3-Z` 的输出结果中。

`Hello 3-Z` 本身也是按照顺序输出的，但是 `Hello 2-Y` 却可能穿插在它的输出中间，原因是 `Hello 2-Y` 先执行，并且是异步执行的，因此它并不等结果执行完，就开始执行后面的语句了，所以当它执行完时，可能已经执行过几条 `Hello 3-Z` 的 `yield` 语句了。

将协程包装成闭包函数

`wrap` 函数不仅仅可以将普通函数包装成支持 `promise` 参数的函数。

`wrap` 函数还支持将协程包装成闭包函数的功能，包装之后的函数，不仅可以将协程当做普通函数一样执行，而且还支持传递 `promise` 参数。例如：

```
1. use \Hprose\Future;
2. use \Hprose\Http\Client;
3.
4. $test = new Client("http://hprose.com/example/");
5.
6. $coroutine = Future\wrap(function($test) {
7.     var_dump(1);
8.     var_dump((yield $test->hello("hprose")));
9.     $a = $test->sum(1, 2, 3);
10.    $b = $test->sum(4, 5, 6);
11.    $c = $test->sum(7, 8, 9);
12.    var_dump((yield $test->sum($a, $b, $c)));
13.    var_dump((yield $test->hello("world")));
14. });
15.
16. $coroutine($test);
17. $coroutine(Future\value($test));
```

该程序执行结果为：

```
1. int(1)
2. int(1)
3. string(12) "Hello hprose"
4. string(12) "Hello hprose"
5. int(45)
6. int(45)
7. string(11) "Hello world"
8. string(11) "Hello world"
```

我们会发现通过 `wrap` 函数包装的协程，不再需要使用 `co` 函数来执行了。

另外，`wrap` 函数包装的对象上的生成器方法也会自动变为普通方法。例如：

```
1. use \Hprose\Future;
2.
3. class Test {
4.     function testco($x) {
5.         yield $x;
6.     }
7. }
8.
9. $test = Future\wrap(new Test());
10.
11. $test->testco(123)->then('var_dump');
12. $test->testco(Future\value('hello'))->then('var_dump');
```

该程序运行结果为：

```
1. int(123)
2. string(5) "hello"
```

协程与异常处理

在协程内，`yield` 不但可以将异步的 `promise` 结果转换成同步结果，而且可以将 `REJECTED` 状态的 `promise` 对象转换为抛出异常。例如：

```
1. use Hprose\Client;
2. use Hprose\Future;
3.
4. Future\co(function() {
5.     $client = Client::create('http://hprose.com/example/');
6.     try {
7.         (yield $client->ooxx());
8.     }
9.     catch (Exception $e) {
10.         echo $e->getMessage();
11.     }
12. });
```

该程序运行结果为：

```
1. Can't find this function ooxx().
```

在协程内抛出的异常如果没有用 `try` `catch` 语句捕获，那么第一个抛出的异常将会中断协程的执行，并将整个协程的返回值设置为 `REJECTED` 状态的 `promise` 对象，异常本身作为 `reason` 的值。例如：

```
1. use Hprose\Client;
2. use Hprose\Future;
3.
4. Future\co(function() {
5.     $client = Client::create('http://hprose.com/example/');
6.     (yield $client->oo());
7.     (yield $client->xx());
8. })->catchError(function($e) {
9.     echo $e->getMessage();
10. });
```

该程序运行结果为：

```
1. Can't find this function oo().
```

原文：<https://github.com/hprose/hprose-php/wiki/04-%E5%8D%8F%E7%A8%8B>

05 Hprose 客户端

概述

Hprose 2.0 for PHP 客户端相比 1.x 版本有了比较大的改动。

核心版本除了提供了客户端的基本实现的基类以外，还提供了 HTTP 客户端和 Socket 客户端。这两个客户端都可以创建为同步或异步客户端。这两个客户端既可以在命令行环境下使用，也可以在 php-fpm 或其他 PHP 环境下使用。















另外 swoole 版本 提供了纯异步的 HTTP 客户端，Socket 客户端和 WebSocket 客户端。Swoole 客户端只能在命令行环境下使用。

其中 HTTP 客户端支持跟 HTTP、HTTPS 绑定的 Hprose 服务器通讯。

Socket 客户端支持跟 TCP、Unix Socket 绑定的 Hprose 服务器通讯，并且支持全双工和半双工两种模式。

WebSocket 客户端支持跟 ws、wss 绑定的 Hprose 服务器通讯。

为了清晰对比，这里列出一个表格：

功能列表	hprose-php	hprose-swoole
同步调用	 "	 "
异步调用	 "	 "
HTTP 客户端	 "	 "
Socket 客户端	 "	 "
WebSocket 客户端	 "	 "
命令行环境	 "	 "
非命令行环境	 "	 "

尽管支持这么多不同的底层网络协议，但除了在对涉及到底层网络协议的参数设置上有所不同以外，其它的用法都完全相同。因此，我们在下面介绍 Hprose 客户端的功能时，若未涉及到底层网络协议的区别，就以 HTTP 客户端为例来进行说

明。

创建客户端

创建客户端有两种方式，一种是直接使用构造器，另一种是使用工厂方法 `create`。

使用构造器创建客户端

`Hprose\Client` 是一个抽象类，因此它不能作为构造器直接使用。如果你想创建一个具体的底层网络协议绑定的客户端，你可以将它作为父类，至于如何实现一个具体的底层网络协议绑定的客户端，这已经超出了本手册的内容范围，这里不做具体介绍，有兴趣的读者可以参考 `Hprose\Http\Client`、`Hprose\Socket\Client` 和 `Hprose\Swoole\WebSocket\Client` 等底层网络协议绑定客户端的实现源码。

`Hprose\Http\Client`、`Hprose\Socket\Client` 这两个类是可以直接使用的构造器。它们分别对应 HTTP 客户端、Socket 客户端。

创建方式如下：

```
1. $client = new \Hprose\Http\Client([$uriList = null[, $async = true]]);
```

`[]` 内的参数表示可选参数。

当两个参数都省略时，创建的客户端是未初始化的异步客户端，后面需要使用 `useService` 方法进行初始化，这是后话，暂且不表。

第 1 个参数 `$uriList` 是服务器地址，该服务器地址可以是单个的地址字符串，也可以是由多个地址字符串组成的数组。当该参数为多个地址字符串组成的数组时，客户端会从这些地址当中随机选择一个作为服务地址。因此需要保证这些地址发布的都是完全相同的服务。

第 2 个参数 `$async` 表示是否是异步客户端，在 Hprose 2.0 for PHP 中，默认创建的都是异步客户端，这是因为 Swoole 客户端只支持异步，为了可以方便的在普通客户端和 Swoole 客户端之间切换，所以默认设置为异步。异步客户端在进行远程调用时，返回值为一个 `promise` 对象。而同步客户端在进行远程调用时，返回值为实际返回值（或者抛出异常）。客户端创建之后，该类型不能被更改。

例如：

创建一个同步的 HTTP 客户端

```
1. $client = new \Hprose\Http\Client('http://hprose.com/example/', false);
```

创建一个同步的 TCP 客户端

```
1. $client = new \Hprose\Socket\Client('tcp://127.0.0.1:1314', false);
```

创建一个异步的 Unix Socket 客户端

```
1. $client = new \Hprose\Socket\Client('unix:/tmp/my.sock');
```

创建一个异步的 WebSocket 客户端

```
1. $client = new \Hprose\Swoole\WebSocket\Client('ws://127.0.0.1:8080/');
```

注意：如果要使用 swoole 客户端，需要在 composer.json 加入对 `hprose/hprose-swoole` 的引用。且 Swoole 客户端不支持第二个参数。

另外，如果创建的是 Swoole 的客户端，还有更简单的方式：

同样创建一个异步的 WebSocket 客户端

```
1. $client = new \Hprose\Swoole\Client('ws://127.0.0.1:8080/');
```

也就是说，只需要使用 `Hprose\Swoole\Client`，就可以创建所有 Swoole 支持的客户端了，Hprose 可以自动根据服务器地址的 scheme 来判断客户端类型。

通过工厂方法 create 创建客户端

```
1. $client = \Hprose\Client::create($uriList = null, [ $async = true]);
```

`create` 方法与构造器函数的参数一样，返回结果也一样。但是第一个参数 `$uriList` 不能被省略。

使用 `create` 方法更加方便，因此，除非在创建客户端的时候，不想指定服务地址，否则，应该优先考虑使用 `create` 方法来创建客户端。

`\Hprose\Client::create` 支持创建 Hprose 核心库上的客户端，`Hprose\Swoole\Client::create` 支持创建 swoole 的客户端。例如：

创建一个同步的 HTTP 客户端

```
1. $client = \Hprose\Client::create('http://hprose.com/example/', false);
```

创建一个同步的 TCP 客户端

```
1. $client = \Hprose\Client::create('tcp://127.0.0.1:1314', false);
```

创建一个异步的 **Unix Socket** 客户端

```
1. $client = \Hprose\Client::create('unix:/tmp/my.sock');
```

创建一个异步的 **WebSocket** 客户端

```
1. $client = \Hprose\Swoole\Client::create('ws://127.0.0.1:8080/');
```

注册自己的客户端实现类

如果你自己创建了一个客户端实现，你可以通过：

```
1. Client::registerClientFactory($scheme, $clientFactory);
```

或者

```
1. Client::tryRegisterClientFactory($scheme, $clientFactory);
```

这两个静态方法来注册自己的客户端实现。

注册之后，你就可以使用 `create` 方法来创建你的客户端对象了。

`registerClientFactory` 方法会覆盖原来已注册的相同 `$scheme` 的客户端类，`tryRegisterClientFactory` 方法不会覆盖。

uri 地址格式

HTTP 服务地址格式

HTTP 服务地址与普通的 URL 地址没有区别，支持 `http` 和 `https` 两种协议，这里不做介绍。

WebSocket 服务地址格式

除了协议从 `http` 改为 `ws`（或 `wss`）以外，其它部分与 `http` 地址表示方式完全相同，这里不再详述。

TCP 服务地址格式

```
1. <scheme>://<ip>:<port>
```

`<ip>` 是服务器的 IP 地址，也可以是域名。

`<port>` 是服务器的端口号，hprose 的 TCP 服务没有默认端口号，因此不可省略。

`<scheme>` 表示协议，它可以为以下取值：

- tcp
- ssl
- sslv2
- sslv3
- tls

`tcp` 表示 tcp 协议，地址可以是 ipv6 地址，也可以是 ipv4 地址。

`ssl`，`sslv2`，`sslv3` 和 `tls` 表示安全的 tcp 协议。如有必要，可设置客户端安全证书。

Unix Socket 服务地址格式

```
1. unix:<path>
```

其中 `<path>` 是绝对路径（以 `/` 开头）。例如：

```
1. unix:/tmp/my.sock
```

事件

onError 事件

该事件为 `callable` 类型，默认值为 `NULL`。

当客户端采用回调方式进行调用时，并且回调函数没有参数，如果发生异常，该事件会被调用。该事件的回调函数格式为：

```
1. function onError($name, $e);
```

`$name` 是字符串类型，`$e` 在 PHP 5 中为 `Exception` 类型或它的子类型对象，在 PHP 7 中是 `Throwable` 接口的实现类对象。

onFailswitch 事件

该属性为 `callable` 类型，默认值为 `NULL`。

当调用的 `failswitch` 属性设置为 `true` 时，如果在调用中出现网络错误，进行服务器切换时，该事件会被触发。该事件的回调函数格式为：

```
1. function onFailswitch($client);
```

`$client` 即客户端对象。

属性

uri 属性

只读属性。字符串类型，表示当前客户端所调用的服务地址。

filters 属性

只读属性。数组类型，表示当前客户端上添加的过滤器列表。

timeout 属性

整数类型，默认值为 `30000`，单位是毫秒（ms）。表示客户端在调用时的超时时间，如果调用超过该时间后仍然没有返回，则会以超时错误返回。

failswitch 属性

布尔类型。默认值为 `false`。表示当前客户端在因网络原因调用失败时是否自动切换服务地址。当客户端服务地址仅设置一个时，不管该属性值为何，都不会切换地址。

failround 属性

整数类型，只读属性。初始值为 `0`。当调用中发生服务地址切换时，如果服务列表中所有的服务地址都切换过一遍之后，该属性值会加 `1`。你可以根据该属性来决定是否更新服务列表。更新服务列表可以使用 `setUriList` 方法。

idempotent 属性

布尔类型，默认值为 `false`。表示调用是否为幂等性调用，幂等性调用表示不论

该调用被重复几次，对服务器的影响都是相同的。幂等性调用在因网络原因调用失败时，会自动重试。如果 `failswitch` 属性同时被设置为 `true`，并且客户端设置了多个服务地址，在重试时还会自动切换地址。

retry 属性

整数类型，默认值为 `10`。表示幂等性调用在因网络原因调用失败后的重试次数。只有 `idempotent` 属性为 `true` 时，该属性才有作用。

byref 属性

布尔类型，默认值为 `false`。表示调用是否为引用参数传递。当设置为引用参数传递时，服务器端会传回修改后的参数值（即使没有修改也会传回）。因此，当不需要该功能时，设置为 `false` 会比较节省流量。

simple 属性

布尔类型，默认值为 `false`。表示调用中所传输的数据是否为简单数据。

简单数据是指：`null`、数字（包括整数、浮点数）、`Boolean` 值、字符串、日期时间等基本类型的数据或者不包含引用的数组和对象。当该属性设置为 `true` 时，在进行序列化操作时，将忽略引用处理，加快序列化速度。但如果数据不是简单类型的情况下，将该属性设置为 `true`，可能会因为死循环导致堆栈溢出的错误。

简单的讲，用 `JSON` 可以表示的数据都是简单数据。但是对于比较复杂的 `JSON` 数据，设置 `simple` 为 `true` 可能不会加快速度，反而会减慢，比如对象数组。因为默认情况下，hprose 会对对象数组中的重复字符串的键值进行引用处理，这种引用处理可以对序列化起到优化作用。而关闭引用处理，也就关闭了这种优化。

因为不同调用的数据可能差别很大，因此，建议不要修改默认设置，而是针对某个调用进行单独设置。

方法

close 方法

关闭客户端。它会在析构方法中被自动调用，因此，你通常不需要手动调用它。

getUriList 方法

获取服务器列表。

setUriList 方法

设置服务器列表。

getTimeout 方法

获取 timeout 属性值。

setTimeout 方法

设置 timeout 属性值。

getRetry 方法

获取 retry 属性值。

setRetry 方法

设置 retry 属性值。

isIdempotent 方法

获取 idempotent 属性值。

setIdempotent 方法

设置 idempotent 属性值。

isFailswitch 方法

获取 failswitch 属性值。

setFailswitch 方法

设置 failswitch 属性值。

getFailround 方法

获取 failround 属性值。

isByref 方法

获取 byref 属性值。

setByref 方法

设置 byref 属性值。

isSimple 方法

获取 simple 属性值。

setSimple 方法

设置 simple 属性值。

getFilter 方法

获取添加的第一个过滤器对象。

setFilter 方法

设置一个过滤器对象，如果原来已经设置或添加了过滤器，该方法会先清除掉之前的设置，然后在设置参数所指定的过滤器。

addFilter 方法

添加一个过滤器。跟 `setFilter` 只能设置一个过滤器不同，通过 `addFilter` 方法，可以添加多个过滤器。

removeFilter 方法

删除指定的过滤器。如果没有找到返回 `false`，删除成功返回 `true`。

useService 方法

```
1. function useService([$uriList = array()[, $namespace = '']]);
```

该方法两个参数都是可选的。

该方法返回值为一个 `\Hprose\Proxy` 对象。该对象是远程服务调用代理对象，你可以在上面直接调用远程方法。

当设置了 `$uriList` 参数时，跟上面的功能相同，但是会替换当前的 `$uriList` 设置。`$uriList` 可以是单个地址，也可以是一个服务地址列表。

参数 `$namespace` 是远程服务的名称空间，它本质上是方法名的别名前缀。例如，服务器端发布的方法名是：`user_add`，`user_remove`，`user_update`，`user_get`。那么可以这样使用：

```
1. $userService = $client->useService($uri, 'user');
2. $userId = $userService->add(array('name' => 'Tom', 'age' => 18));
3. $userService->update($userId, array('age' => 14));
4. $user = $userService->get($userId);
5. $userService->remove($userId);
```

上面的例子中 `$userService` 是 `useService` 方法返回的一个远程服务代理对象。因为在调用 `useService` 方法时指定了 `user` 这个名称空间，所以在 `$userService` 对象上调用远程方法时，就不再加 `user_` 这个前缀了。代理对象在发起调用时，会自动帮你加上。

注意：`$userService` 和 `useService` 方法看上去很像，但他们拼写并不相同，意义也不相同，它们之间没有关系，没有关系，没有关系【重要的事情说三遍】。

invoke 方法

```
1. function invoke($name, array $args = array()[, $callback = null[,
    InvokeSettings $settings = null]]])
```

该方法是客户端最核心的方法，但是你可能永远也不会直接使用它，虽然你可能不用，但我还是要对它做一下介绍。

`$name` 是远程函数/方法名。

`$args` 是方法参数。如果没有参数就是空数组（当然你也可以不写该参数，因为默认值就是空数组），有一个参数就是有一个值的数组，有两个参数就是有两个值的数组。如果是异步客户端，`$args` 中可以包含 `promise` 对象的参数值。但

是同步客户端的同步调用不可以包含 `promise` 对象的参数。另外，请注意，`$args` 只支持位置参数，不支持命名参数。所以不要试图使用字符串键名来作为命名参数的名称，它是不管用的，不管用的，不管用的.....

`$callback` 是回调函数。这个是传统的异步调用方式。不管你创建的是同步客户端，还是异步客户端，使用回调函数的方式进行异步调用都是支持的。另外，如果指定了该参数，`$args` 参数中也可以包含 `promise` 对象的参数值，那怕是同步客户端。但是你真的愿意使用回调函数层层回调吗？真的愿意吗？不愿意吗？愿意吗？不管你愿意还是不愿意，我还是需要介绍一下回调函数的格式：

```
1. function callback();
2. function callback($result);
3. function callback($result, $args);
4. function callback($result, $args, $error);
```

回调函数支持这四种格式，在使用 `invoke` 方法时，`$callback` 只要是 `callable` 类型就可以，不管是函数，方法，还是闭包和可执行对象，都可以。但是如果是用方法名直接调用，`$callback` 必须是闭包类型。

下面介绍一下参数的意义

参数	解释
<code>\$result</code>	服务器端返回的结果，如果没有结果，该值为 <code>NULL</code> ，如果没有 <code>\$error</code> 参数，并且调用发生异常，该值为异常对象。
<code>\$args</code>	调用的参数，如果没有参数，该值为空数组。
<code>\$error</code>	如果调用发生异常，该值为异常对象，否则该值为 <code>NULL</code> 。

最后一个参数 `$settings` 是 Hprose 2.0 新加的。1.x 版本中在 `$callback` 后面还有一堆几个参数用户设置调用的特殊属性，但是 Hprose 2.0 为每个调用增加了许多特殊属性设置，所以再用位置参数的方式，用户恐怕就要晕了。所以，这里牺牲了兼容性，改成了使用一个 `$settings` 参数来存储所有的特殊属性设置。

`$settings` 参数是一个 `\Hprose\InvokeSettings` 类型的对象。`\Hprose\InvokeSettings` 是个什么东？让我们来看一下：

InvokeSettings 类

该类的构造参数是一个数组。你可以通过这个数组来设置调用的特殊属性，下面来看一下支持哪些设置：

- mode
- byref
- simple
- failswitch

- timeout
- idempotent
- retry
- oneway
- userdata

下面来分别介绍一下这些设置的意义：

mode

该设置表示结果返回的类型，它有4个取值，分别是：

- `Hprose\ResultMode::Normal`
- `Hprose\ResultMode::Serialized`
- `Hprose\ResultMode::Raw`
- `Hprose\ResultMode::RawWithEndTag`

`Hprose\ResultMode::Normal` 是默认值，表示返回正常的已被反序列化的结果。

`Hprose\ResultMode::Serialized` 表示返回的结果保持序列化的格式。

`Hprose\ResultMode::Raw` 表示返回原始数据。

`Hprose\ResultMode::RawWithEndTag` 表示返回带有结束标记的原始数据。

这样说明也许有些晦涩，让我们来看一个例子就清楚了：

```
1. use Hprose\Client;
2. use Hprose\InvokeSettings;
3. use Hprose\ResultMode;
4.
5. $client = Client::create('http://hprose.com/example/', false);
6.
7. var_dump($client->hello("World", new InvokeSettings(array('mode' =>
8.   ResultMode::Normal))));
9. var_dump($client->hello("World", new InvokeSettings(array('mode' =>
10.   ResultMode::Serialized))));
11. var_dump($client->hello("World", new InvokeSettings(array('mode' =>
12.   ResultMode::Raw))));
13. var_dump($client->hello("World", new InvokeSettings(array('mode' =>
14.   ResultMode::RawWithEndTag))));
```

该程序执行结果如下：

```
1. string(11) "Hello World"
2. string(16) "s11"Hello World""
3. string(17) "Rs11"Hello World""
4. string(18) "Rs11"Hello World"z"
```

眼尖的同学一定发现了，我们这里并没有使用 `invoke` 方法，而是直接使用远程方法名 `hello` 作为本地方法名来直接使用，参数值 `World`，也没有放在数组里，我们前面说过了，你可能永远都不会直接使用 `invoke` 方法，就是因为你可以直接这样调用，这比使用 `invoke` 方法简单直观的多。

`$callback` 参数在这里也省略了，因为我们这里是使用同步调用，所以不能有它。使用 `invoke` 方法也是通过省略它来进行同步调用的。而且，一旦习惯了 Hprose 2.0 的写法，即使是异步调用，你也不会再使用这个参数。所以，就让我们忽略它啊，后面我们也不再提它了。

你可能会这样的疑问，为啥不直接用数组来表示这些特殊设置，而要在数组外面再套一个 `new InvokeSettings` 呢？这样写起来很麻烦不是吗？确实是麻烦了一点，但是这也是不得已而为之，因为数组也是远程调用可以传递的参数类型，所以，如果用数组，调用可分不清这个数组是设置，还是传给服务器的参数，就会把它作为远程调用参数传给服务器了，这显然不是我们期望的。所以，只好包上一层 `InvokeSettings` 的包装，这样客户端就知道，这个设置不是要传给服务器的参数了。

不过这里可以透露一个小秘密，如果你是使用方法名进行远程调用（也就是说，你不是直接使用 `invoke` 方法），而且使用了 `$callback` 参数（我们这里不是为了提它，只是不得不提它），那么在 `$callback` 参数后面的设置确实可以使用一个普通数组，而不需要套一个 `new InvokeSettings` 的包装。不过为了避免你用顺了手，你最好还是不要这样玩。就当我没提好了。

byref

该设置表示调用是否为引用参数传递方式。例如：

```
1. <?php
2. require_once "../vendor/autoload.php";
3.
4. use Hprose\Client;
5. use Hprose\InvokeSettings;
6.
7. $client = Client::create('http://hprose.com/example/', false);
8.
9. $weeks = array(
10.     'Monday' => 'Mon',
11.     'Tuesday' => 'Tue',
12.     'Wednesday' => 'Wed',
13.     'Thursday' => 'Thu',
14.     'Friday' => 'Fri',
15.     'Saturday' => 'Sat',
16.     'Sunday' => 'Sun'
17. );
18.
19. $args = array($weeks);
20. $client->invoke('swapKeyAndValue', $args, new InvokeSettings(array('byref' =>
```

```

    true)));
21. var_dump($args[0]);
22.
23. $client->swapKeyAndValue($weeks, function($result, $args) {
24.     var_dump($args[0]);
25. }, array('byref' => true));

```

运行结果为：

```

1. array(7) {
2.     ["Mon"]=>
3.     string(6) "Monday"
4.     ["Tue"]=>
5.     string(7) "Tuesday"
6.     ["Wed"]=>
7.     string(9) "Wednesday"
8.     ["Thu"]=>
9.     string(8) "Thursday"
10.    ["Fri"]=>
11.    string(6) "Friday"
12.    ["Sat"]=>
13.    string(8) "Saturday"
14.    ["Sun"]=>
15.    string(6) "Sunday"
16. }
17. array(7) {
18.     ["Mon"]=>
19.     string(6) "Monday"
20.     ["Tue"]=>
21.     string(7) "Tuesday"
22.     ["Wed"]=>
23.     string(9) "Wednesday"
24.     ["Thu"]=>
25.     string(8) "Thursday"
26.     ["Fri"]=>
27.     string(6) "Friday"
28.     ["Sat"]=>
29.     string(8) "Saturday"
30.     ["Sun"]=>
31.     string(6) "Sunday"
32. }

```

注意：同步调用需要使用 `invoke` 方法才支持引用参数传递。异步调用要用回调方式才支持引用参数传递。

simple

该设置表示本次调用中所传输的参数是否为简单数据。前面在属性介绍中已经进行了说明，这里就不在重复。

failswitch

该设置表示当前调用在因网络原因失败时是否自动切换服务地址。

timeout

该设置表示本次调用的超时时间，如果调用超过该时间后仍然没有返回，则会以超时错误返回。

idempotent

该设置表示本次调用是否为幂等性调用，幂等性调用在因网络原因调用失败时，会自动重试。

retry

该设置表示幂等性调用在因网络原因调用失败后的重试次数。只有 `idempotent` 设置为 `true` 时，该设置才有作用。

oneway

该设置表示当前调用是否不等待返回值。当该设置为 `true` 时，请求发送之后，并不等待服务器返回结果，而是直接返回 `null`。

userdata

该属性是一个对象，它用于存放一些用户自定义的数据。这些数据可以通过 `$context` 对象在整个调用过程中进行传递。当你需要实现一些特殊功能的 Filter 或 Handler 时，可能会用到它。

上面这些设置除了可以作为 `settings` 参数的属性传入以外，还可以在远程方法上直接进行属性设置，这些设置会成为 `settings` 参数的默认值。例如上面引用参数传递的例子还可以写成这样：

```
1. use Hprose\Client;
2.
3. $client = Client::create('http://hprose.com/example/', false);
4.
```

```

5. $weeks = array(
6.     'Monday' => 'Mon',
7.     'Tuesday' => 'Tue',
8.     'Wednesday' => 'Wed',
9.     'Thursday' => 'Thu',
10.    'Friday' => 'Fri',
11.    'Saturday' => 'Sat',
12.    'Sunday' => 'Sun'
13. );
14.
15. $args = array($weeks);
16.
17. $client->swapKeyAndValue["byref"] = true;
18. $client->swapKeyAndValue($weeks, function($result, $args) {
19.     var_dump($args[0]);
20. });

```

但是要注意，这样设置的属性，只有用远程方法名直接调用才有效，使用 `invoke` 方法调用时无效。

直接用远程方法名调用

上面在介绍 `invoke` 方法时，我们已经在例子中看到过直接用远程方法名调用的用法了。这里再补充一点高级的。

当你的服务器发布了多个对象，并且每个对象上都有一组自己的方法，还有可能有重名的方法，因此，你可能会为每个对象添加一个名称空间（namespace），它在 Hprose 中是以别名前缀的方式实现的。

例如：服务器端发布了一个 `user` 对象，一个 `order` 对象。上面都有 `add`，`update`，`remove` 和 `get` 方法。发布 `user` 对象时，添加了一个 `user` 名空间，发布 `order` 对象时，添加了一个 `order` 名空间。

这样服务器端发布的方法名实际上是：`user_add`，`user_update`，`user_remove`，`user_get`，`order_add`，`order_update`，`order_remove` 和 `order_get`。

客户端可以直接用这些方法名进行调用，例如：

```

1. $userid = $client->user_add(array('name' => 'Tom', 'age' => 18));
2. $client->user_remove($userid);

```

还可以这样调用：

```

1. $userid = $client->user->add(array('name' => 'Tom', 'age' => 18));
2. $client->user->remove($userid);

```

这两种调用方式都是可以的，后一种方法看上去更清晰，而且还可以简化为：


```

1. $user = $client->user;
2. $userid = $user->add(array('name' => 'Tom', 'age' => 18));
3. $user->remove($userid);

```

链式调用

上面我们讲了很多同步调用，异步调用用的也是传统的回调方式。下面我们来讲讲 Hprose 2.0 新增的 `promise` 方式的异步调用。

对于 Hprose 异步客户端来说，使用 `invoke` 方法或者直接使用远程方法名调用，返回的结果是一个 `promise` 对象。因此，它可以进行链式调用，例如：

```

1. <?php
2. require_once "../vendor/autoload.php";
3.
4. use Hprose\Client;
5.
6. $client = Client::create('http://hprose.com/example/');
7.
8. $client->sum(1, 2)
9.     ->then(function($result) use ($client) {
10.         return $client->sum($result, 3);
11.     })
12.     ->then(function($result) use ($client) {
13.         return $client->sum($result, 4);
14.     })
15.     ->then(function($result) {
16.         var_dump($result);
17.     });

```

该程序的结果为：

```

1. 10

```

当你有很多回调的时候，这种方式要比层层回调的异步方式清晰的多。但是，这还不算最简单的。

更简单的顺序调用

前面我们讲过，当使用方法名调用时，远程调用的参数本身也可以是对象。

`promise`

因此，上面的链式调用还可以直接简化为：


```

1. use Hprose\Client;
2.
3. $client = Client::create('http://hprose.com/example/');
4.
5. $client->sum($client->sum($client->sum(1, 2), 3), 4)
6.     ->then(function($result) {
7.         var_dump($result);
8.     });

```

这比上面的链式调用更加直观。但是还可以更简单，例如：

```

1. use Hprose\Client;
2. use Hprose\Future;
3.
4. $client = Client::create('http://hprose.com/example/');
5.
6. $var_dump = Future\wrap('var_dump');
7. $sum = $client->sum;
8.
9. $var_dump($sum($sum($sum(1, 2), 3), 4));

```

远程方法可以直接作为一个闭包对象返回，之后可以单独调用。然后在加上用 `wrap` 函数包装的 `var_dump`。

现在代码看上去完全是同步的啦。

这种方式看上去是同步的，但是实际上却可以并行异步执行，尤其是当一个调用的参数依赖于其它几个调用的结果时候，例如：

```

1. use Hprose\Client;
2. use Hprose\Future;
3.
4. $client = Client::create('http://hprose.com/example/');
5.
6. $var_dump = Future\wrap('var_dump');
7. $sum = $client->sum;
8.
9. $r1 = $sum(1, 3, 5, 7, 9);
10. $r2 = $sum(2, 4, 6, 8, 10);
11. $r3 = $sum($r1, $r2);
12. $var_dump($r1, $r2, $r3);

```

这个程序的运行结果为：

```

1. int(25)
2. int(30)
3. int(55)

```

该程序虽然是异步执行，但是书写方式却是同步的，不需要写任何回调。

而且这里还有一个好处，`$r1` 和 `$r2` 两个调用的参数之间没有依赖关系，是两个相互独立的调用，因此它们将会并行执行，而 `$r3` 的调用依赖于 `$r1` 和 `$r2`，因此 `$r3` 会等 `$r1` 和 `$r2` 都执行结束后才会执行。也就是说，它不但保证了有依赖关系的调用会根据依赖关系顺序执行，而且对于没有依赖的调用还能保证并行执行。

这是回调方式和链式调用方式都很难做到的，即使可以做到，也会让代码变得晦涩难懂。

这也是 Hprose 2.0 最大的改进之一。

原文: <https://github.com/hprose/hprose-php/wiki/05-Hprose-%E5%AE%A2%E6%88%B7%E7%AB%AF>

06 Hprose 服务器

概述

Hprose 2.0 for PHP 支持多种底层网络协议绑定的服务器，比如：HTTP 服务器，Socket 服务器和 WebSocket 服务器。

其中 HTTP 服务器支持在 HTTP、HTTPS 协议上通讯。

Socket 服务器支持在 TCP、Unix Socket 协议上通讯，并且支持全双工和半双工两种模式。

WebSocket 服务器支持在 ws、wss 协议上通讯。

其中，Hprose 2.0 的核心库提供了 HTTP 服务器和 Socket 服务器，基于 Swoole 的版本提供了 HTTP、Socket、WebSocket 服务器。另外还有 Yii2、Symfony、PSR7 版本的服务器。

为了清晰对比，这里列出一个表格：

功能列表	hprose-php	hprose-swoole	hprose-yii	hprose-symfony	hprose-psr7
HTTP 服务器	✓	✓	✓	✓	✓
Socket 服务器	✓	✓	✗	✗	✗
WebSocket 服务器	✗	✓	✗	✗	✗
命令行环境	✓	✓	✗	✗	✓
非命令行环境	✓	✗	✓	✓	✓
推送服务	✓	✓	✗	✗	✗

尽管支持这么多不同的底层网络协议，但除了在对涉及到底层网络协议的参数设置上有所不同以外，其它的用法都完全相同。因此，我们在下面介绍 Hprose 服务器的功能时，若未涉及到底层网络协议的区别，就以 HTTP 服务器为例来进行说明。

Server 与 Service 的区别

Hprose 的服务器端的实现，分为 `Service` 和 `Server` 两部分。

其中 `Service` 部分是核心功能，包括接收请求，处理请求，服务调用，返回应答等整个服务的处理流程。

而 `Server` 则主要负责启动和关闭服务器，它包括设置服务地址和端口，设置服务器启动选项，启动服务器，接收来自客户端的连接然后传给 `Service` 进行处理。

之所以分开，是为了更方便的跟已有的库和框架结合，例如：swoole, yii、symfony 等版本都是直接继承 `Service` 来实现自己的服务，之后再创建自己的 `Server` 用于服务设置和启动。

分开的另外一个理由是，`Server` 部分的实现是很简单的，有时候开发者可能会希望把 Hprose 服务结合到自己的某个服务器中去，而不是作为一个单独的服务器来运行，在这种情况下，也是直接使用 `Service` 就可以了。

当开发者没有什么特殊需求，只是希望启动一个独立的 Hprose 服务器时，那使用 `Server` 就是一个最方便的选择了。

创建服务器

创建服务器有多种方式，我们先从最简单的方式说起。

创建 HTTP 服务器

```
1. use Hprose\Http\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server();
8. $server->addFunction('hello');
9. $server->start();
```

该服务是基于 php-fpm 或者任何支持运行 PHP 的 web 服务器（例如 Apache、IIS 等）。把它丢到配置好 PHP 的 web 服务器上，然后在浏览器端打开该文件，你应该会看到以下输出：

```
1. Fa2{u#s5"hello"}z
```

如果你用过 Hprose 1.x，你会发现 2.0 版本多了一个 `u#`，这表示有一个

名字叫 `#` 的方法，这是 Hprose 2.0 服务器端一个特殊的方法，用来产生一个唯一编号，客户端可以调用该方法得到一个唯一编号用来标识自己。Hprose 2.0 的客户端在使用推送功能时，如果没有手动指定 `id` 的情况下，会自动调用该方法来获取 `id`。

该方法的默认实现很简单，就是一个自增计数，所以一旦当服务器关闭之后重启，该方法会重新开始从 `0` 开始计数。这种方式可能不适用于某些场合，因此，你可能希望能够使用自己的实现来替换它，这是可以做到的。你只需要在发布方法时，将别名指定为 `'#'` 就可以覆盖默认实现了。

虽然上面对 `'#'` 这个方法介绍了这么多，然而该服务器并不支持推送服务。如果你真的需要推送服务，你需要创建一个独立的服务器。例如：

```
1. use Hprose\Swoole\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server("http://0.0.0.0:8086");
8. $server->addFunction('hello');
9. $server->start();
```

这是一个基于 Swoole 的 HTTP 的 Hprose 独立服务器。这代码跟上面的代码几乎一样。只是 `use` 的路径不同，`new Server` 时多了一个服务器地址。其它的都一模一样。

但是这段代码的运行方式跟上面那个服务器却完全不同。要运行该服务器，只需要在命令行中键入：

```
1. php HelloServer.php
```

上面假设跟这个文件保存为 `HelloServer.php`。你的服务就运行起来了，现在你的命令行会卡在那里不动了。但是你在浏览器中输入：`http://<IP>:8086`，其中 `<IP>` 是你运行这个程序的那台服务器的 IP 地址，如果跟你浏览器是同一台机器，IP 可以是 `127.0.0.1`，如果是不同的机器，你应该比我更清楚是多少，所以我就假设你输入正确了，然后你在浏览器端应该看到跟上面那个程序同样的输出（我就不再重复了）。

如果你的浏览器打不开，请检查你的地址是否输入正确，或者你是否开了防火墙把该服务给屏蔽了，或者你的网线是否插好了，诸如此类的问题我就不再一一列举并给出解决方案了。如果你解决不了，我也无能为力，所以请不要费劲地把此类问题提交到 [issues](#) 中了，我真的帮不了你。

创建 TCP 服务器

```

1. use Hprose\Socket\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6. $server = new Server("tcp://0.0.0.0:1314");
7. $server->addFunction('hello');
8. $server->start();

```

这是使用 Hprose 核心版本所提供的 Socket 服务器创建的 TCP 服务器。该服务的运行方式跟上面的 HTTP 独立服务器一样，在命令行中使用 php 命令执行就可以了。之后你就可以用客户端去调用它了，至于客户端如何使用，请参考 [05. Hprose 客户端](#) 一章。

该版本的 Socket 服务器没有使用 `pcntl` 之类的扩展，因此可以在 windows 中使用。该服务是通过单线程异步方式实现的（类似于 node.js），因此该服务支持高并发，也支持推送服务，但是对于每一个服务方法来说，最好执行时间不要过长，因为这会阻塞整个服务。如果你确实有比较耗时的服务要执行，可以考虑开起子进程，借助消息队列将结果返回异步化等方法来自行解决。

当然你也可以考虑使用 Swoole 版本的 Socket 服务器，例如：

```

1. use Hprose\Swoole\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6. $server = new Server("tcp://0.0.0.0:1314");
7. $server->addFunction('hello');
8. $server->start();

```

在基本方法的使用上，`Hprose\Swoole\Server` 和 `Hprose\Socket\Server` 是一样的，因此上面的代码中，只有 `use` 语句不同，其它的代码都相同。

对于上面的代码来说，`Hprose\Swoole\Server` 和 `Hprose\Socket\Server` 的性能是几乎一样的，看不出什么优势来。因为默认 Swoole 的 TCP 服务器是使用 Base 模式运行的，该方式跟 `Hprose\Socket\Server` 的方式是一致的。默认采用这种模式，是因为只有这种模式下才支持推送服务，而且该模式下服务编写简单，不需要考虑多进程数据通信问题，另外，新版本的 swoole 对 Base 模式也做了强化，提供了更多的设置和优化，因此 Hprose 默认采用这种模式。

swoole 的服务器还提供了一种进程模式，但是该模式下，多个进程因为不能共享内存，所以推送功能无法使用。如果你不需要推送服务，你可以在创建服务器时这样来指定进程模式：

```

1. $server = new Server("tcp://0.0.0.0:1314", SWOOLE_PROCESS);

```

关于 swoole 的这两种模式可以参见 [swoole 的文档](#)，文档里介绍了 3 种模

式，因为其中的线程模式，现在新版本的 swoole 已经不支持了，所以这里就不提了。

创建 UNIX Socket 服务器

```
1. use Hprose\Socket\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server("unix:/tmp/my.sock");
8. $server->addFunction('hello');
9. $server->start();
```

```
1. use Hprose\Swoole\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server("unix:/tmp/my.sock");
8. $server->addFunction('hello');
9. $server->start();
```

同样是这两种方式都可以，如何选择请随意。

这两个 Unix Socket 服务器，在启动时，稍微一些区别，对于

`Hprose\Socket\Server`，如果 `/tmp/my.sock` 文件已存在，服务器将不会启动，而是抛出异常。而对于 `Hprose\Swoole\Server` 则不管是否存在（哪怕有另一个服务正在运行），都会正常启动，不会报错。

创建 Web Socket 服务器

```
1. use Hprose\Swoole\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6. $server = new Server("ws://0.0.0.0:8088");
7. $server->addFunction('hello');
8. $server->start();
```

目前，Web Socket 服务器只有 Swoole 版本支持。创建方式除了地址改为 web socket 的地址格式以外，其它都一样。这里就不多做解释了。

其它方式

Hprose 还提供了可以跟 Yii、Symfony、PSR7 等框架结合使用的 HTTP 服务器，这些都是作为单独项目模块提供的，这里就不多做介绍了。

服务设置

debug 属性

该属性为 `boolean` 类型，默认值为 `false`。

用来设置服务器是否是工作在 `debug` 模式下，在该模式下，当服务器端发生异常时，将会将详细的错误堆栈信息返回给客户端，否则，只返回错误信息。

isDebugEnabled 方法

用于获取 `debug` 属性的当前值。

setDebugEnabled 方法

用于设置 `debug` 属性值。

simple 属性

该属性为 `boolean` 类型，默认值为 `false`。

简单数据是指：`null`、数字（包括整数、浮点数）、`Boolean` 值、字符串、日期时间等基本类型的数据或者不包含引用的数组和对象。当该属性设置为 `true` 时，在进行序列化操作时，将忽略引用处理，加快序列化速度。但如果数据不是简单类型的情况下，将该属性设置为 `true`，可能会因为死循环导致堆栈溢出的错误。

简单的讲，用 `JSON` 可以表示的数据都是简单数据。但是对于比较复杂的 `JSON` 数据，设置 `simple` 为 `true` 可能不会加快速度，反而会减慢，比如对象数组。因为默认情况下，hprose 会对对象数组中的重复字符串的键值进行引用处理，这种引用处理可以对序列化起到优化作用。而关闭引用处理，也就关闭了这种优化。

因为不同调用的数据可能差别很大，因此，建议不要修改默认设置，而是针对某个调用进行单独设置。

isSimple 方法

获取 `simple` 属性值。

setSimple 方法

设置 `simple` 属性值。

passContext 属性

该属性为 `boolean` 类型，默认值为 `false`。

该属性表示在调用中是否将 `$context` 自动作为最后一个参数传入调用方法。

你也可以针对某个服务函数/方法进行单独设置。

除非所有的服务方法的参数最后都定义了 `$context` 参数。否则，建议不要修改默认设置，而是针对某个服务函数/方法进行单独设置。

isPassContext 方法

获取 `passContext` 属性值。

setPassContext 方法

设置 `passContext` 属性值。

errorDelay 属性

该属性为整型值，默认值为 10000，单位是毫秒。

该属性表示在调用执行时，如果发生异常，将延时一段时间后再返回给客户端。

在关闭该功能的情况下，如果某个服务因为编写错误抛出异常，客户端又反复重试该调用，可能会导致服务器不能正常处理其它业务请求而造成的假死机现象。使用该功能，可以避免这种问题发生。

如果你不需要该功能，设置为 0 就可以关闭它。

getErrorDelay 方法

获取 `errorDelay` 的属性值。

setErrorDelay 方法

设置 `errorDelay` 的属性值。

errorTypes 属性

设置捕获错误的级别，默认值为 `error_reporting` 函数的返回值，即 PHP 的默认设置。

捕获到错误，警告、提示都会以异常的形式发送给客户端。

getErrorTypes 方法

获取 `errorTypes` 的属性值。

setErrorTypes 方法

设置 `errorTypes` 的属性值。

发布服务

hprose 为发布服务提供了多个方法，这些方法可以随意组合，通过这种组合，你所发布的服务将不会局限于某一个对象，或某一个类，而是可以将不同的函数和方法随意重新组合成一个服务。

addFunction 方法

```
1. public function addFunction($func, $alias = '', array $options = array());
```

该方法的功能是发布函数为远程调用。

`$func` 不仅仅可以是函数名，也可以是方法，例如：`array("SomeClass", "someMethod")`，`array($someObj, "someMethod")`，还可以是闭包（匿名函数），还可以是 callable 对象（就是实现了 `__invoke` 魔术方法的对象），还可以是生成器函数（就是 Hprose 支持的协程）。

`$alias` 是发布函数的别名，该别名是客户端调用时所使用的名字。如果本身是函数名，那么 `$alias` 参数可以省略，省略之后，发布的名称跟 `$func` 中的方法名部分相同。其它情况下，不能省略 `$alias` 参数。别名中，你可以使用 `_` 分隔符。当客户端调用时，根据不同的语言，可以自动

转换成 `.` 分隔的调用，或者 `->` 分隔的调用。在别名中不要使用 `.` 分隔符。

`$options` 是一个关联数组，它里面包含了一些对该服务函数的特殊设置，有以下设置项可以设置：

- mode
- simple
- oneway
- async
- passContext

mode

该设置表示该服务函数返回的结果类型，它有4个取值，分别是：

- `Hprose\ResultMode::Normal`
- `Hprose\ResultMode::Serialized`
- `Hprose\ResultMode::Raw`
- `Hprose\ResultMode::RawWithEndTag`

`Hprose\ResultMode::Normal` 是默认值，表示返回正常的已被反序列化的结果。

`Hprose\ResultMode::Serialized` 表示返回的结果保持序列化的格式。

`Hprose\ResultMode::Raw` 表示返回原始数据。

`Hprose\ResultMode::RawWithEndTag` 表示返回带有结束标记的原始数据。

这四种结果的形式在客户端的相关介绍中已有说明，这里不再重复。

不过要注意的是，这里的设置跟客户端的设置并没有直接关系，这里设置的是服务函数本身返回的数据格式，即使服务函数返回的是

`Hprose\ResultMode::RawWithEndTag` 格式的数据，客户端仍然可以以其它三种格式来接收数据。

该设置通常用于做高性能的缓存或代理服务器。我们在后面介绍

`addMissingFunction` 方法时再举例说明。

simple

该设置表示本服务函数所返回的结果是否为简单数据。默认值与全局设置一致。前面在属性介绍中已经进行了说明，这里就不再重复。

oneway

该设置表示本服务函数是否不需要等待返回值。当该设置为 `true` 时，调用会异

步开始，并且不等待结果，立即返回 `null` 给客户端。默认值为 `false`。

async

该设置表示本服务函数是否为异步函数，异步函数的最后一个参数是一个回调函数，用户需要在异步函数中调用该回调方法来传回返回值，例如：

```
1. use Hprose\Http\Server;
2.
3. function hello($name, $callback) {
4.     $callback("Hello $name!");
5. }
6.
7. $server = new Server();
8. $server->addFunction('hello', array("async" => true));
9. $server->start();
```

passContext

该设置与 `$server->passContext` 属性的功能相同。但在这里它是针对该服务函数的单独设置。例如：

```
1. use Hprose\Socket\Server;
2.
3. function hello($name, $context) {
4.     return "Hello $name! -- " . stream_socket_get_name($context->socket, true);
5. }
6.
7. $server = new Server("tcp://0.0.0.0:1314");
8. $server->addFunction('hello', array("passContext" => true));
9. $server->start();
```

这里使用的是 `Hprose\Socket\Server`，所以 `$context->socket` 可以得到服务器接收到的客户端连接对象。要注意不同的服务器，`$context` 中包含的内容是不同的，要根据具体使用的服务器来确定用法。所以如果要编写通用的服务，要避免使用这些跟具体服务器有关的上下文属性。

`$context->clients` 这个属性上面包含了关于推送的方法，这些方法是通用的，不过也仅在独立服务器上通用，在 `fpm`、`cgi` 等模式下运行的 HTTP 服务器上不支持。

当 `passContext` 和 `async` 同时设置为 `true` 的时候，服务函数的 `$context` 参数应该放在 `$callback` 参数之前，例如：

```
1. use Hprose\Socket\Server;
2.
3. function hello($name, $context, $callback) {
```

```

4.     $callback("Hello $name! -- " . stream_socket_get_name($context->socket,
5.     true));
6. }
7. $server = new Server("tcp://0.0.0.0:1314");
8. $server->addFunction('hello', array("async" => true, "passContext" => true));
9. $server->start();

```

addAsyncFunction 方法

```

1. public function addAsyncFunction($func, $alias = '', array $options =
    array());

```

该方法与 `addFunction` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addFunction` 发布异步方法的简写形式。

addMissingFunction 方法

```

1. public function addMissingFunction($func, array $options = array());

```

该方法用于发布一个用于处理客户端调用缺失服务的函数。缺失服务是指服务器端并没有明确发布的远程函数/方法。例如：

在服务器端没有发布 `hello` 函数时，在默认情况下，客户端调用该函数，服务器端会返回 `'Can't find this function hello()'` 这样一个错误。

但是如果服务器端通过本方法发布了一个用于处理客户端调用缺失服务的函数，则服务器端会返回这个 `$func` 函数的返回值。

该方法还可以用于做代理服务器，例如：

```

1. use Hprose\InvokeSettings;
2. use Hprose\Http\Client;
3. use Hprose\Socket\Server;
4. use Hprose\ResultMode;
5.
6. $client = new Client("http://www.hprose.com/example/", false);
7. $settings = new InvokeSettings(array("mode" => ResultMode::RawWithEndTag));
8.
9. $proxy = function($name, $args) use ($client, $settings) {
10.     return $client->invoke($name, $args, $settings);
11. };
12.
13. $server = new Server("tcp://0.0.0.0:1314");
14. $server->debug = true;
15. $server->addMissingFunction($proxy, array("mode" =>
    ResultMode::RawWithEndTag));
16. $server->start();

```

现在，客户端对这个服务器所发出的所有请求，都会通过 `$proxy` 函数转发到 `http://www.hprose.com/example/` 这个服务上，并把结果直接按照原始方式返回。

另外，这我们用的是同步客户端，如果用异步客户端，需要在调用 `$client->invoke` 之后，手动执行 `$client->loop()` 来开始调用，然后再把结果返回。

不过如果我们使用的是 Swoole 的 Http 客户端，就不需要手动调用 `loop` 方法了。所以用异步调用，首选 swoole 客户端。

如果用 swoole 异步客户端，`$client->invoke` 方法的返回值是一个 `promise` 对象，也就是说，服务函数/方法其实也可以直接返回 `promise` 对象，异步服务不一定非要用 `callback` 方式。

addAsyncMissingFunction 方法

```
1. public function addAsyncMissingFunction($func, array $options = array());
```

该方法与 `addMissingFunction` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addMissingFunction` 发布异步方法的简写形式。

addFunctions 方法

```
1. public function addFunctions(array $funcs, array $aliases = array(), array $options = array());
```

如果你想同时发布多个方法，可以使用该方法。

`$funcs` 是函数数组，数组元素必须为 `callable` 类型的对象。

`$aliases` 是别名数组，数组元素必须是字符串，并且需要与 `$funcs` 数组中每个相同键名的元素一一对应。

当 `$funcs` 中的函数全都是具名函数时，`$aliases` 可以省略。

`$options` 的选项值跟 `addFunction` 方法相同。

addAsyncFunctions 方法

```
1. public function addAsyncFunctions(array $funcs, array $aliases = array(), array $options = array());
```

该方法与 `addFunctions` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addFunctions` 发布异步方法的简写形式。

addMethod 方法

```
1. public function addMethod($method, $scope, $alias = '', array $options = array());
```

该方法跟 `addFunction` 类似，它的功能是添加方法。

`$method` 是方法名，字符串类型。

`$scope` 是 `$method` 所在的类或对象。如果 `$method` 是静态方法，那么 `$scope` 应为类名，如果 `$method` 是实例方法，那么 `$scope` 应为对象。不要搞混，切记，切记。

`$alias` 是发布方法的别名，忽略时，跟 `$method` 的相同。

`$options` 的选项值跟 `addFunction` 方法相同。

addAsyncMethod 方法

```
1. public function addAsyncMethod($method, $scope, $alias = '', array $options = array());
```

该方法与 `addMethod` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addMethod` 发布异步方法的简写形式。

addMissingMethod 方法

```
1. public function addMissingMethod($method, $scope, array $options = array());
```

该方法的功能与 `addMissingMethod` 类似。它们之前的区别跟 `addMethod` 和 `addFunction` 相同。这里就不详细介绍了。

addAsyncMissingMethod 方法

```
1. public function addAsyncMissingMethod($method, $scope, array $options = array());
```

该方法与 `addMissingMethod` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addMissingMethod` 发布异步方法的简写形式。

addMethods 方法


```
1. public function addMethods($methods, $scope, $aliases = array(), array $options = array());
```

该方法的功能与 `addFunctions` 类似。它们之前的区别跟 `addMethod` 和 `addFunction` 相同。这里就不详细介绍了。

addAsyncMethods 方法

```
1. public function addAsyncMethods($methods, $scope, $aliases = array(), array $options = array());
```

addInstanceMethods 方法

```
1. public function addInstanceMethods($object, $class = '', $aliasPrefix = '', array $options = array());
```

该方法用于发布 `$object` 对象上所在类上声明的所有 `public` 实例方法。

当指定了 `$class` 参数时，将只发布 `$class` 这一层上声明的所有 `public` 实例方法。

`$aliasPrefix` 是别名前缀，例如假设有一个 `$user` 对象，该对象上包含有 `add`，`del`，`update`，`query` 四个方法。那么当调用：

```
1. $server->addInstanceMethods($user, '', 'user');
```

的方式来发布 `$user` 对象上的这四个方法后，等同于这样的发布：

```
1. $server->addMethods(array('add', 'del', 'update', 'query'), $user,
2. array('user_add', 'user_del', 'user_update', 'user_query'));
```

即在每个发布的方法名之前都添加了一个 `user` 的前缀。注意这里前缀和方法名之间是使用 分隔的。

当省略 `$aliasPrefix` 参数时，发布的方法名前不会增加任何前缀。

最后的 `$options` 选项值跟 `addFunction` 方法相同。

addAsyncInstanceMethods 方法

```
1. public function addAsyncInstanceMethods($object, $class = '', $aliasPrefix = '', array $options = array());
```

该方法与 `addInstanceMethods` 功能相同，但是 `async` 选项被默认设置为

`true`。也就是说，它是 `addInstanceMethods` 发布异步方法的简写形式。

addClassMethods 方法

```
1. public function addClassMethods($class, $scope = '', $aliasPrefix = '',
    array $options = array())];];];
```

该方法用来发布 `$class` 上定义的所有静态方法。

`$scope` 是方法实际执行所在的类，通常 `$scope` 跟 `$class` 是等同的（默认值），不过你也可以设置 `$scope` 为 `$class` 的父类。

`$aliasPrefix` 作用跟 `addInstanceMethods` 的 `$aliasPrefix` 参数一样。

最后的 `$options` 选项值跟 `addFunction` 方法相同。

addAsyncClassMethods 方法

```
1. public function addAsyncClassMethods($class, $scope = '', $aliasPrefix = '',
    array $options = array())];];];
```

该方法与 `addClassMethods` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `addClassMethods` 发布异步方法的简写形式。

add 方法

上面如此之多的 `addXXX` 方法也许会把你搞晕，也许你不查阅本手册，都记不清该使用哪个方法来发布。

没关系，`add` 方法就是用来简化上面这些 `addXXX` 方法的。

`add` 方法不支持 `$options` 参数。其它参数你只要按照上面任何一个方法的参数来写，`add` 方法都可以自动根据参数的个数和类型判断该调用哪个方法进行发布，当你不需要设置 `$options` 参数时，它会大大简化你的工作量。

addAsync 方法

该方法与 `add` 功能相同，但是 `async` 选项被默认设置为 `true`。也就是说，它是 `add` 发布异步方法的简写形式。

remove 方法

该方法与 `add` 功能相反。使用该方法可以移除已经发布的函数，方法或者推送

主题。该方法的参数为发布的远程方法的别名。注意：该别名是大小写敏感的。

在RPC方法中获取当前 \$context 对象

在 addFunction 时给设置 passContext = true 参数可以在 RCP 回调的方法在最后一个参数传进去，但是这样做需要改动回调函数比较麻烦，那么可以在 RPC回调方法里直接使用到，如果是协程的，使用

```
$context = Hprose\Service::getCurrentContext()
```

获取

```
$context = Hprose\Service::getCurrentContextCo()
```

方法

获取。

原文：<https://github.com/hprose/hprose-php/wiki/06-Hprose-%E6%9C%8D%E5%8A%A1%E5%99%A8>

07 Hprose 服务器事件

Hprose 服务器端提供了几个事件，它们分别是：

- onBeforeInvoke
- onAfterInvoke
- onSendError
这三个事件所有的 Hprose 服务器都支持。
- onSendHeader
这个事件仅 HTTP 服务器支持。
- onAccept
- onClose
这两个事件 Socket 和 WebSocket 服务器支持。

这些事件是以属性方式提供的，只需要将事件函数赋值给这些属性即可。

例如：

```
1. $server->onBeforeInvoke = function($name, $args, $byref, \stdClass $context) {
2.     ...
3. }
4. $server->onAfterInvoke = function($name, $args, $byref, $result, \stdClass
   $context) {
5.     ...
6. }
7. $server->onSendError = function($error, \stdClass $context) {
8.     ...
9. }
```

对于 `onBeforeInvoke`，`onAfterInvoke` 和 `onSendError` 这三个事件属性的事件处理函数允许有返回值。

`onBeforeInvoke` 和 `onAfterInvoke` 可以返回一个 `promise` 对象，当该 `promise` 对象变为失败（`rejected`）状态时，将会返回失败原因作为返回给客户端的错误信息。

`onBeforeInvoke`，`onAfterInvoke` 和 `onSendError` 都可以直接返回一个 `Error` 实例对象来作为返回给客户端的错误信息。

`onBeforeInvoke`，`onAfterInvoke` 和 `onSendError` 还可以通过抛出异常来返回错误信息给客户端。

onBeforeInvoke 事件

该事件在调用执行前触发，该事件的处理函数形式为：

```
1. function($name, &$amp;args, $byref, \stdClass $context) { ... }
```

参数 `$name` 是服务函数/方法名。参数 `$args` 是调用的参数数组，可以声明为引用参数。参数 `$byref` 表示是否是引用参数传递。参数 `$context` 是该调用的上下文参数。

如果在该事件中抛出异常、返回错误对象、或者返回一个失败（`rejected`）状态的 `promise` 对象。则不再执行服务函数/方法。

onAfterInvoke 事件

该事件在调用执行后触发，该事件的处理函数形式为：

```
1. function($name, &$amp;args, $byref, &$amp;result, \stdClass $context) { ... }
```

参数 `$name` 是服务函数/方法名。参数 `$args` 是调用的参数数组，可以声明为引用参数。参数 `$byref` 表示是否是引用参数传递。参数 `$result` 是调用执行的结果，可以声明为引用参数。参数 `$context` 是该调用的上下文参数。

如果在该事件中抛出异常、返回错误对象、或者返回一个失败（`rejected`）状态的 `promise` 对象。则不再返回结果 `$result`，而是将错误信息返回给客户端。

onSendError 事件

该事件在服务端发生错误时触发，该事件的处理函数形式为：

```
1. function(&$error, \stdClass $context) { ... }
```

如果在该事件中抛出异常、返回错误对象。则该错误会替代原来的错误信息返回给客户端。

`$error` 参数可以声明为引用参数，在事件中可以对 `$error` 进行修改。

当服务器与客户端之间发生网络中断性的错误时，仍然会触发该事件，但是不会有错误信息发送给客户端。

onSendHeader 事件

该事件在服务器发送 HTTP 头时触发，该事件的处理函数形式为：

```
1. function(\stdClass $context) { ... }
```

如果在该事件中抛出异常，则不再执行后序操作，直接返回异常信息给客户端。

onAccept 事件

该事件在 Socket 或 WebSocket 服务器接受客户端连接时触发，该事件的处理函数形式为：

```
1. function(\stdClass $context) { ... }
```

如果在该事件中抛出异常，则会断开跟该客户端的连接。

onClose 事件

该事件在 Socket 或 WebSocket 服务器跟客户端之间的连接关闭时触发，该事件的处理函数形式为：

```
1. function(\stdClass $context) { ... }
```

该事件中抛出异常不会对服务器和客户端有任何影响。

onError 事件

该事件仅被 `Hprose\Socket\Server` 所支持，其它服务器不支持，该事件在服务器与客户端发生通讯错误，无法将错误发送给客户端时触发。该事件的处理函数形式为：

```
1. function($error, \stdClass $context) { ... }
```

该事件中抛出异常不会对服务器和客户端有任何影响。

原文：<https://github.com/hprose/hprose-php/wiki/07-Hprose-%E6%9C%8D%E5%8A%A1%E5%99%A8%E4%BA%8B%E4%BB%B6>

08 HTTP 服务器特殊设置

普通 HTTP 服务器

crossDomain 属性

该属性用于设置是否允许浏览器客户端跨域调用本服务。默认值为 `false`。当设置为 `true` 时，自动开启 CORS 跨域支持。当配合 `addAccessControlAllowOrigin` 和 `removeAccessControlAllowOrigin` 这两个方法时，还可以做细粒度的跨域设置。

isCrossDomainEnabled 方法

获取 `crossDomain` 的属性值。

setCrossDomainEnabled 方法

设置 `crossDomain` 的属性值。

p3p 属性

该属性用于设置是否允许 IE 浏览器跨域设置 Cookie。默认为 `false`。

isP3PEnabled 方法

获取 `p3p` 的属性值。

setP3PEnabled 方法

设置 `p3p` 的属性值。

get 属性

该属性用于设置是否接受 GET 请求。默认为 `true`。如果你不希望用户使用浏览器直接浏览器服务器函数列表，你可以将该属性设置为 `false`。

isGetEnabled 方法

获取 `get` 的属性值。

setEnabled 方法

设置 `get` 的属性值。

addAccessControlAllowOrigin 方法

添加允许跨域的地址。

removeAccessControlAllowOrigin 方法

删除允许跨域的地址。

Swoole 的 HTTP 服务器

Swoole 的 HTTP 服务器出了包含普通服务器提供的上面那些设置和方法以外，还有以下几个特殊属性和方法：

server 属性

只读属性，它是底层的 `swoole_http_server` 对象。你可以通过它来调用 swoole 服务器的功能。

settings 属性

用于设置 swoole 服务器运行时的各项参数。具体有哪些设置，可参见 swoole 的官方文档，不过需要注意，有一些关于协议解析的选项参数不要设置。

set 方法

用于设置 `settings` 的属性值。多次设置可以合并。在服务器启动之后，该方法就不能再调用了。

on 方法

用于设置 swoole 的服务事件。

addListener 方法

添加新的监听地址，添加的地址必须为相同的类型。

listen 方法

添加新的监听地址，并返回监听的服务端口对象，在该对象上进行设置后，可以实现不同类型服务的监听。

Swoole 的 WebSocket 服务器

Swoole 的 WebSocket 服务器跟 Swoole 的 HTTP 服务器所包含的属性和方法是一样的。而且 Swoole 的 WebSocket 服务器是双料服务器，同时支持 Web Socket 通讯和 HTTP 通讯，所以不论是 WebSocket 客户端还是 HTTP 客户端都可以访问该服务器。

server 属性

只读属性，它是底层的 `swoole_websocket_server` 对象。你可以通过它来调用 swoole 服务器的功能。

原文: <https://github.com/hprose/hprose-php/wiki/08-HTTP-%E6%9C%8D%E5%8A%A1%E5%99%A8%E7%89%B9%E6%AE%8A%E8%AE%BE%E7%BD%AE>

09 Socket 服务器特殊设置

普通 Socket 服务器

该服务器实现是基于 PHP 内置的 Stream 实现的，它支持以下属性和方法：

settings 属性

该属性的设置值为即 `stream_context_create` 的 `$options` 参数的值。其具体设置可以参见：[PHP 官方手册—上下文 \(Context\) 选项和参数](#)，当使用 TCP 服务器时，可以参见 [套接字上下文选项](#) 进行设置，当使用 SSL 服务器时，可以参见 [SSL 上下文选项](#) 进行设置。

set 方法

用于设置 `settings` 属性。多次设置可以被合并到 `settings` 属性中。

noDelay 属性

设置为 `true` 后，TCP 连接发送数据时，会关闭 Nagle 合并算法，立即发往客户端连接。默认值即为 `true`。

isNoDelay 方法

获取 `noDelay` 的属性值。

setNoDelay 方法

设置 `noDelay` 的属性值。

keepAlive 属性

开启 Socket 的 keepAlive 监测机制。默认为 `true`。

isKeepAlive 方法

获取 `keepAlive` 的属性值。

setKeepAlive 方法

设置 `keepAlive` 的属性值。

readBuffer 属性

该属性表示读取缓冲区大小，默认值为 8192。

getReadBuffer 方法

获取 `readBuffer` 的属性值。

setReadBuffer 方法

设置 `readBuffer` 的属性值。

writeBuffer 属性

该属性表示写入缓冲区大小，默认值为 8192。

getWriteBuffer 方法

获取 `writeBuffer` 的属性值。

setWriteBuffer 方法

设置 `writeBuffer` 的属性值。

defer 方法

```
1. function $server->defer(callable $callback);
```

延后执行一个 PHP 函数。底层会在 IO 事件循环完成后执行此函数。此方法的目的是为了能让一些 PHP 代码延后执行，程序优先处理 IO 事件。

after 方法

```
1. function $server->after($delay, callable $callback);
```

在指定的时间后执行函数 `$callback` 。

`after` 方法是一个一次性定时器，执行完成后就会销毁。

`$delay` 指定延时时间，单位为毫秒。

`$callback` 为时间到期后所执行的函数。 `$callback` 函数不接受任何参数。

tick 方法

```
1. function $server->tick($delay, callable $callback);
```

在指定的时间后周期性的执行函数 `$callback` 。

`tick` 方法是一个周期性定时器，每次执行完成后，会重新计时。

`$delay` 指定延时时间，单位为毫秒。

`$callback` 为时间到期后所执行的函数。 `$callback` 函数不接受任何参数。

clear 方法

```
1. function $server->clear($id);
```

用于清除 `after` 和 `tick` 计时器。参数 `$id` 为 `after` 和 `tick` 方法的返回值。

addListener 方法

用于添加新的监听地址。新增的监听地址必须为同一类型的。

Swoole 的 Socket 服务器

该服务器实现是基于 `swoole_server` 实现的，它支持以下属性和方法：

server 属性

只读属性，它是底层的 `swoole_server` 对象。你可以通过它来调用 swoole 服务器的功能。

settings 属性

用于设置 swoole 服务器运行时的各项参数。具体有哪些设置，可参见 swoole 的官方文档，不过需要注意，有一些关于协议解析的选项参数不要设置。

set 方法

用于设置 `settings` 的属性值。多次设置可以合并。在服务器启动之后，该方法就不能再调用了。

noDelay 属性

设置为 `true` 后，TCP 连接发送数据时，会关闭 Nagle 合并算法，立即发往客户端连接。默认值即为 `true`。

isNoDelay 方法

获取 `noDelay` 的属性值。

setNoDelay 方法

设置 `noDelay` 的属性值。

on 方法

用于设置 swoole 的服务事件。

addListener 方法

添加新的监听地址，添加的地址必须为相同的类型。

listen 方法

添加新的监听地址，并返回监听的服务端口对象，在该对象上进行设置后，可以实现不同类型服务的监听。

原文: <https://github.com/hprose/hprose-php/wiki/09-Socket-%E6%9C%8D%E5%8A%A1%E5%99%A8%E7%89%B9%E6%AE%8A%E8%AE%BE%E7%BD%AE>

10 推送服务

简介

Hprose 2.0 最大的亮点就是增加了推送功能的支持，而且这个功能的增加是在不修改现有通讯协议的方式下实现的，因此，这里的推送服务，即使不是 Hprose 2.0 的客户端或者服务器也可以使用。

当然，在旧版本的客户端调用推送服务，或者在旧版本的服务器上自己实现推送，需要多写一些代码。所以，如果你所使用的语言支持 Hprose 2.0，那么推荐直接使用 Hprose 2.0 的推送 API 来做推送，这样会极大的减少你的工作量。

Hprose 2.0 的客户端都支持推送功能，服务器端需要 Hprose 的独立服务器（即 fpm 方式的 HTTP 服务器不支持推送功能），例如：`\Hprose\Socket\Server` 和 `hprose-swoole` 版本的所有服务器都支持推送功能。

下面我们来分别介绍一下客户端和服务端增加的关于推送的 API。

客户端

客户端关于推送的方法只有两个，它们分别是：

subscribe 方法

```
1. function subscribe($name, $callback)
2. function subscribe($name, $id, $callback)
3. function subscribe($name, $callback, $timeout)
4. function subscribe($name, $id, $callback, $timeout)
5. function subscribe($name, $callback, $timeout, $failswitch)
6. function subscribe($name, $id, $callback, $timeout, $failswitch)
```

`subscribe` 方法的用处是订阅服务器端的推送服务。该方法有两种方式，一种是自动获取设置客户端 `$id`，另一种是手动设置客户端 `$id`。

参数 `$name` 是订阅的主题名，它实际上也是一个服务器端的方法，该方法与普通方法的区别是，它只有一个参数 `$id`，该参数表示客户端的唯一编号，该方法的返回值即推送信息，当返回值为 `NULL` 或者抛出异常时，客户端会忽略并再次调用该 `$name` 对应的方法。当该方法返回推送消息时，`$callback` 回调函数会执行，并同时再次调用该 `$name` 对应的方法。因此当没有推送消息时，该方法不应该立即返回值，而应该挂起等待，直到超时或者有推送消息时再返回结果。

当然，对于开发者来说，自己实现一个完善的推送方法还是有一定难度的。因此，Hprose 2.0 的服务器端已经提供了一整套的专门用于推送的 API，通过这些 API，可以方便的自动实现用于推送的服务方法。在后面介绍服务器端时，我们再介绍这部分内容。

参数 `$id` 是客户端的唯一编号，如果省略的话，客户端会使用自动编号机制，如果该自动编号未初始化，会自动调用一个名字为 `#` 的服务器端远程方法，之所以使用这个特殊的名字是为了防止跟用户发布的普通方法发生冲突。Hprose 2.0 服务器已经自动实现了该方法，但是用户也可以用自己的实现来替换它，它的默认实现是一个唯一字符串。当用户指定了 `$id` 参数时，客户端会将它作为该 `$name` 对应方法的参数值传给服务器端，但不会修改客户端自动获取的 `$id` 属性值。

参数 `$callback` 是用来处理推送消息的回调函数，该参数不能省略。

参数 `$timeout` 是等待推送消息的超时时间，单位是毫秒 (ms)，可以省略，默认值与 `timeout` 属性值相同。超时之后并不会产生异常，而是重新请求推送。因此，如果用户要在服务器端自己实现推送方法，应当注意处理好同一个客户端对同一个推送方法可能会进行重复调用的问题。如果使用 Hprose 2.0 提供的推送 API，则不需要关心这一点。

参数 `$failswitch` 表示当客户端与服务器端通讯中发生网络故障，是否自动切换服务器。默认值是 `false`，表示不切换。

对于同一个推送主题，`subscribe` 方法允许被多次调用，这样可以对同一个推送主题指定多个不同的回调方法。但通常没有必要也不推荐这样做。

unsubscribe 方法

```
1. function unsubscribe($name)
2. function unsubscribe($name, $callback)
3. function unsubscribe($name, $id)
4. function unsubscribe($name, $id, $callback)
```

该方法用于取消订阅推送主题。当调用该方法时，带有 `$callback` 参数，将只取消对该 `$callback` 方法的回调，除非这是该主题上最后一个 `$callback`，否则对该主题远程方法的调用并不会中断。当所有的 `$callback` 都被取消之后，或者当调用该方法时，没有指定 `$callback` 参数时，将会中断对该主题远程方法的循环调用。

如果 `$id` 参数未指定，那么当客户端自动获取的 `$id` 属性有值时，将只取消对该 `$id` 属性值对应的推送主题的订阅。当客户端自动获取的 `$id` 属性未初始化时，将会取消该主题上所有的订阅。

通常来说，当你调用 `subscribe` 方法时如果指定了 `$id` 参数，那么当调用 `unsubscribe` 方法时你也应该指定相同的 `$id` 参数。当你调用 `subscribe`

方法时没有指定 `$id` 参数，那么当调用 `unsubscribe` 方法时你也不需要指定 `$id` 参数。

isSubscribed 方法

```
1. function isSubscribed($name)
```

当名称为 `$name` 的主题已被订阅时，返回 `true`，否则返回 `false`。

subscribedList 方法

```
1. function subscribedList()
```

返回已被订阅的主题的列表，返回值是一个字符串数组。数组元素为已订阅的主题名称。

服务器端

服务器端提供了比较多的关于推送的 API，包括广播，多播和单播方式的推送，还有超时，心跳，推送事件等设置。

timeout 属性

该属性设置推送空闲超时的。该属性默认值为 120000，单位是毫秒（ms），即 2 分钟。

当服务器发布了推送主题后（后面会专门介绍推送），客户端会跟服务器端保持一个长连接，如果达到超时时间，仍然没有任何消息推送给客户端，则返回 `NULL`，此时，如果客户端仍然在线的话，则会立即再次发送获取推送主题的请求。服务器端通过这种方式可以获知客户端是否还在线。

getTimeout 方法

获取 `timeout` 属性值。

setTimeout 方法

设置 `timeout` 属性值。

heartbeat 属性

该属性用来设置推送的心跳检测间隔时间。该属性默认值为 3000，单位是毫秒（ms），即 3 秒钟。

当服务器端推送数据给客户端后，如果客户端在 `heartbeat` 时间内没有取走推送数据，则服务器端认为客户端以掉线。对于以掉线的客户端，服务器端会清除为该客户端分配的内存空间，并将该客户端从推送列表中移除。

`timeout` 和 `heartbeat` 属性在检测客户端是否离线时是相互配合的，当服务器端没有向客户端推送任何消息时，服务器端需要至少 `timeout` + `heartbeat` 的时间才能检测到客户端以离线。当服务器端有向客户端推送消息时，则在推送消息之后经过 `heartbeat` 时间可以检测到客户端以掉线。

`timeout` 和 `heartbeat` 设置的时间越短，检测到客户端离线的时间就越短。但是需要注意以下几个问题：

`timeout` 时间越短，服务器端和客户端之间的用于检测是否掉线的通讯就越频繁，所以不应该将 `timeout` 设置的过短，否则会严重增加服务器的负担。

因此，`timeout` 的设置一般不应少于 30 秒。对于负载比较高的服务器，保持默认值就是一个不错的选项。

对于推送频繁的服务器来说，`heartbeat` 时间越长，对于已经离线的客户端，在服务器端存储的离线消息就越多，这会严重的占用服务器端的内存，因此，不宜将 `heartbeat` 的时间设置的过长。

如果 `heartbeat` 的时间设置的过短，客户端可能会因为网络原因导致不能及时取走推送消息，这就会导致错误的离线判断，当错误离线判断发生后，会丢失一些推送消息。

因此，`heartbeat` 的选择则应根据客户端的网络情况来决定，如果客户端都是来自局域网，并且客户端数量较少，设置为 1 秒甚至更短的时间也是可以的。而对于比较慢速且不太稳定的移动网络，设置为 5 秒或者 10 秒可能是一个比较合适的取值。对于普通的互联网客户端来说，保持默认值就可以了。

getHeartbeat 方法

返回 `heartbeat` 属性值。

setHeartbeat 方法

设置 `heartbeat` 属性值。

onSubscribe 属性

该属性用来设置客户端订阅事件。

该事件方法格式为：

```
1. void onSubscribe(string $topic, string $id, \Hprose\Service service);
```

当编号为 `$id` 的客户端订阅主题 `$topic` 时，触发 `onSubscribe` 事件。

onUnsubscribe 属性

该属性用来设置客户端取消订阅事件。

该事件方法格式为：

```
1. void onUnsubscribe(string $topic, string $id, \Hprose\Service service);
```

当编号为 `$id` 的客户端退订主题 `$topic` 时，触发 `onUnsubscribe` 事件。

publish 方法

```
1. void publish(string $topic);
2. void publish(string $topic, array $options);
3. void publish(array $topics);
4. void publish(array $topics, array $options);
```

该方法用于发布一个或一组推送主题。这个推送的主题实际上是一个自动生成的远程服务方法。它的功能就是实现推送。

`$topic` 为主题名，`$topics` 为一组主题名。

`$options` 可以包含两个设置，分别是 `timeout` 和 `heartbeat`。

这里 `timeout` 和 `heartbeat` 参数在前面的属性介绍里已经说明过了，这里不再重复。

`publish` 方法仅仅是告诉客户端，现在有一个叫做 `$topic` 的推送主题可以订阅。

而要真正推送数据给客户端，则需要使用以下几个方法。

广播

```
1. void broadcast(string $topic, mixed $result);
2. void broadcast(string $topic, mixed $result, callable $callback);
```

```
3.
4. void push(string $topic, mixed $result);
```

这里有两个推送方法：`broadcast` 和 `push`。

这两个方法功能相同，但是 `broadcast` 方法支持回调，该回调方法有两个参数，这个参数都是数组类型，第一个数组中是所有推送成功的客户端 `$id`，第二个数组中是所有推送失败的客户端 `$id`。而 `push` 方法不支持回调。

一旦服务器启动，你可以在任何地方进行数据推送。比如在其它的服务方法中，在服务器事件中，甚至在服务器外的并行运行的函数中。例如：

时间推送服务器

```
1. use Hprose\Swoole\Server;
2.
3. $server = new Server("tcp://0.0.0.0:2016");
4. $server->publish('time');
5. $server->on('workerStart', function($serv) use ($server) {
6.     $serv->tick(1000, function() use ($server) {
7.         $server->push('time', microtime(true));
8.     });
9. });
10. $server->start();
```

时间显示客户端

```
1. use Hprose\Swoole\Client;
2.
3. $client = new Client("tcp://127.0.0.1:2016");
4. $count = 0;
5. $client->subscribe('time', function($date) use ($client, &$count) {
6.     if (++$count > 10) {
7.         $client->unsubscribe('time');
8.         swoole_event_exit();
9.     }
10.    else {
11.        var_dump($date);
12.    }
13. });
```

先运行服务器，后运行客户端，该然后客户端会每隔一秒钟输出一行，最后输出结果为：

```
1. float(1469610600.1522)
2. float(1469610601.1521)
3. float(1469610602.1523)
```

```

4. float(1469610603.1521)
5. float(1469610604.1522)
6. float(1469610605.1523)
7. float(1469610606.1521)
8. float(1469610607.1522)
9. float(1469610608.1521)
10. float(1469610609.1521)

```

有时候，你可能想在某个服务方法中推送数据给客户端，但是该服务方法可能在其它文件中定义。因此，你得不到 `$server` 对象。那这时还能进行推送吗？

答案是可以，没问题。我们前面说过，在服务方法中我们可以得到一个 `$context` 参数，这个 `$context` 参数中就包含有一个 `clients` 对象，这个对象上就包含了所有跟推送有关的方法，这些方法跟 `$server` 对象上的推送方法是完全一样的，例如：

```

1. $context->clients->broadcast($topic, $result[, $callback]);
2. $context->clients->push($topic, $result);

```

我们再来看一个例子：

服务器

```

1. use Hprose\Swoole\Server;
2.
3. function hello($name, $context) {
4.     $context->clients->push("news", "this is a pushed message: $name");
5.     $context->clients->broadcast("news", array('x' => 1, 'y' => 2));
6.     $fdinfo = $context->server->connection_info($context->socket);
7.     return "Hello $name! -- {$fdinfo['remote_ip']}";
8. }
9.
10. $server = new Server("tcp://0.0.0.0:1980");
11. $server->publish('news');
12. $server->addFunction('hello', array('passContext' => true));
13. $server->start();

```

客户端

```

1. use Hprose\Swoole\Client;
2. use Hprose\Future;
3.
4. Future\co(function() {
5.     $client = new Client("tcp://127.0.0.1:1980");
6.     $id = (yield $client->getId());
7.     $client->subscribe('news', $id, function($news) {
8.         var_dump($news);

```

```

9.     });
10.     var_dump((yield $client->hello('hprose')));
11. });

```

假设我们运行两个客户端，则第一个客户端显示：

```

1. string(32) "this is a pushed message: hprose"
2. string(26) "Hello hprose! -- 127.0.0.1"
3. array(2) {
4.     ["x"]=>
5.     int(1)
6.     ["y"]=>
7.     int(2)
8. }
9. string(32) "this is a pushed message: hprose"
10. array(2) {
11.     ["x"]=>
12.     int(1)
13.     ["y"]=>
14.     int(2)
15. }

```

第二个客户端显示：

```

1. string(32) "this is a pushed message: hprose"
2. string(26) "Hello hprose! -- 127.0.0.1"
3. array(2) {
4.     ["x"]=>
5.     int(1)
6.     ["y"]=>
7.     int(2)
8. }

```

这两个客户端显示结果之后并不会退出，如果有其它客户端再次运行时，这两个客户端还会继续显示推送信息。也就是说，对于已经执行了 `subscribe` 的客户端，在未执行对应的 `unsubscribe` 方法之前，该客户端会一直运行，接收推送数据，即使服务器已经关闭，客户端也不会退出。

多播

```

1. void multicast(string $topic, array $ids, mixed $result);
2. void multicast(string $topic, array $ids, mixed $result, callable $callback);
3.
4. void push(string $topic, array $ids, mixed $result);

```

跟广播类似，多播也有这样几种形式。跟广播相比，多播多了一个 `$ids` 参数，它是一个客户端 `$id` 的数组。也就是说，你可以向指定的一组客户端推送消息。在 `$context->clients` 上同样包含这些方法。

单播

```

1. void unicast(string $topic, string $id, mixed $result);
2. void unicast(string $topic, string $id, mixed $result, callable $callback);
3.
4. void push(string $topic, string $id, mixed $result);

```

单播是跟多播的形式也类似，只不过客户端 `$ids` 数组参数变成了一个客户端 `$id` 参数。

但是还有一点要注意，`unicast` 的回调方法跟 `broadcast` 和 `multicast` 不同，`unicast` 的回调方法只有一个参数，而且是一个布尔值，该值为 `true` 表示推送成功，为 `false` 表示推送失败。

在 `$context->clients` 上同样包含这些方法。

idlist 方法

```

1. array idlist(string $topic);

```

该方法用于获取当前在线的所有客户端的 `$id` 列表。在 `$context->clients` 上也包含该方法。

exist 方法

```

1. bool exist(string $topic, string $id);

```

该方法用于快速判断 `$id` 是否在当前在线的客户端列表中。在 `$context->clients` 上也包含该方法。

注意，客户端在线状态是针对主题的，同一个客户端可能针对一个主题处于在线状态，但是针对另一个主题却处于离线状态，这种情况是正常的。

原文: <https://github.com/hprose/hprose-php/wiki/10-%E6%8E%A8%E9%80%81%E6%9C%8D%E5%8A%A1>

11 Hprose 过滤器

简介

有时候，我们可能会希望在远程过程调用中对通讯的一些细节有更多的控制，比如对传输中的数据进行加密、压缩、签名、跟踪、协议转换等等，但是又希望这些工作能够跟服务函数/方法本身可以解耦。这个时候，Hprose 过滤器就是一个不错的选择。

Hprose 过滤器是一个接口，它有两个方法：

```
1. interface Filter {
2.     public function inputFilter($data, stdClass $context);
3.     public function outputFilter($data, stdClass $context);
4. }
```

其中 `inputFilter` 的作用是对输入数据进行处理，`outputFilter` 的作用是对输出数据进行处理。

`$data` 参数就是输入输出数据，它是 `string` 类型的。这两个方法的返回值也是 `string` 类型的数据，它表示已经处理过的数据，如果你不打算对数据进行修改，你可以直接将 `$data` 参数作为返回值返回。

`$context` 参数是调用的上下文对象，我们在服务器和客户端的介绍中已经多次提到过它。

执行顺序

不论是客户端，还是服务器，都可以添加多个过滤器。假设我们按照添加的顺序把它们叫做 `filter1`，`filter2`，... `filterN`。那么它们的执行顺序是这样的。

在客户端的执行顺序

```
1. +----- outputFilter -----+
2. | +-----+ +-----+ +-----+ |
3. | |filter1|----->|filter2|-----> ... ----->|filterN| |-----+
4. | +-----+ +-----+ +-----+ | v
5. +-----+ +-----+
6. | Hprose Server |
7. +----- inputFilter -----+ +-----+
8. | +-----+ +-----+ +-----+ |
9. | |filter1|<-----|filter2|<----- ... <-----|filterN| |<-----+
```

```

10.  | +-----+          +-----+          +-----+ |
11.  +-----+-----+-----+

```

在服务器端的执行顺序

```

1.          +-----+-----+ inputFilter -----+
2.          | +-----+          +-----+          +-----+ |
3.          +----->| |filterN|-----> ... ----->|filter2|----->|filter1| |
4.          |          | +-----+          +-----+          +-----+ |
5.  +-----+-----+ +-----+-----+-----+
6.  | Hprose Client |
7.  +-----+-----+ +-----+-----+ outputFilter -----+
8.          ^          | +-----+          +-----+          +-----+ |
9.          +-----<| |filterN|<----- ... <-----|filter2|<-----|filter1| |
10.         |          | +-----+          +-----+          +-----+ |
11.         +-----+-----+-----+

```

跟踪调试

有时候我们在调试过程中，可能会需要查看输入输出数据。用抓包工具抓取数据当然是一个办法，但是使用过滤器可以更方便更直接的显示出输入输出数据。

LogFilter.php

```

1. use Hprose\Filter;
2.
3. class LogFilter implements Filter {
4.     public function inputFilter($data, stdClass $context) {
5.         error_log($data);
6.         return $data;
7.     }
8.     public function outputFilter($data, stdClass $context) {
9.         error_log($data);
10.        return $data;
11.    }
12. }

```

Server.php

```

1. use Hprose\Socket\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.

```

```

7. $server = new Server('tcp://0.0.0.0:1143/');
8. $server->addFunction('hello')
9.     ->addFilter(new LogFilter())
10.    ->start();

```

Client.php

```

1. use Hprose\Client;
2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addFilter(new LogFilter());
5. var_dump($client->hello("world"));

```

上面的服务器和客户端代码我们省略了包含路径。请自行脑补，或者直接参见 [examples](#) 目录里面的例子。

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. Cs5"hello"a1{s5"world"}z
2. Rs12"Hello world!"z

```

客户端输出

```

1. Cs5"hello"a1{s5"world"}z
2. Rs12"Hello world!"z
3. string(12) "Hello world!"

```

上面输出操作我们用了 `error_log` 函数，并不是因为我们要输出的内容是错误信息，而是因为 Hprose 内部在服务器端过滤了 `echo`、`var_dump` 这些用 `ob_xxx` 操作可以过滤掉的输出信息，因为不过滤这些信息，一旦服务器有输出操作，就会造成客户端无法正常运行。所以，我们这里用 `error_log` 函数来进行输出。你也可以换成别的你喜欢的方式，只要不会被 `ob_xxx` 操作过滤掉就可以了。下面的例子我们同样使用这个函数来作为输出。

压缩传输

上面的例子，我们只使用了一个过滤器。在本例中，我们展示多个过滤器组合使用的效果。

CompressFilter.php

```

1. use Hprose\Filter;
2.
3. class CompressFilter implements Filter {
4.     public function inputFilter($data, stdClass $context) {
5.         return gzdecode($data);
6.     }
7.     public function outputFilter($data, stdClass $context) {
8.         return gzencode($data);
9.     }
10. }
```

SizeFilter.php

```

1. use Hprose\Filter;
2.
3. class SizeFilter implements Filter {
4.     private $message;
5.     public function __construct($message) {
6.         $this->message = $message;
7.     }
8.     public function inputFilter($data, stdClass $context) {
9.         error_log($this->message . ' input size: ' . strlen($data));
10.        return $data;
11.    }
12.    public function outputFilter($data, stdClass $context) {
13.        error_log($this->message . ' output size: ' . strlen($data));
14.        return $data;
15.    }
16. }
```

Server.php

```

1. use Hprose\Socket\Server;
2.
3. $server = new Server('tcp://0.0.0.0:1143/');
4. $server->addFilter(new SizeFilter('Non compressed'))
5.     ->addFilter(new CompressFilter())
6.     ->addFilter(new SizeFilter('Compressed'))
7.     ->addFunction(function($value) { return $value; }, 'echo')
8.     ->start();
```

Client.php

```

1. use Hprose\Client;
2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addFilter(new SizeFilter('Non compressed'))
5.     ->addFilter(new CompressFilter())
```

```

6.         ->addFilter(new SizeFilter('Compressed'));
7.
8. $value = range(0, 99999);
9. var_dump(count($client->echo($value)));

```

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. Compressed input size: 216266
2. Non compressed input size: 688893
3. Non compressed output size: 688881
4. Compressed output size: 216245

```

客户端输出

```

1. Non compressed output size: 688893
2. Compressed output size: 216266
3. Compressed input size: 216245
4. Non compressed input size: 688881
5. int(100000)

```

在这个例子中，压缩我们使用了 PHP 内置的 gzip 算法，运行前需要确认你开启了这个扩展（一般默认就是开着的）。

加密跟这个类似，这里就不再单独举加密的例子了。

运行时间统计

有时候，我们希望能够对调用执行时间做一个统计，对于客户端来说，也就是客户端调用发出前，到客户端收到调用结果的时间统计。对于服务器来说，就是收到客户端调用请求到要发出调用结果的这一段时间的统计。这个功能，通过过滤器也可以实现。

StatFilter.php

```

1. use Hprose\FILTER;
2.
3. class StatFilter implements Filter {
4.     private function stat(stdClass $context) {

```

```

5.         if (isset($context->userdata->starttime)) {
6.             $t = microtime(true) - $context->userdata->starttime;
7.             error_log("It takes $t s.");
8.         }
9.         else {
10.            $context->userdata->starttime = microtime(true);
11.        }
12.    }
13.    public function inputFilter($data, stdClass $context) {
14.        $this->stat($context);
15.        return $data;
16.    }
17.    public function outputFilter($data, stdClass $context) {
18.        $this->stat($context);
19.        return $data;
20.    }
21. }

```

Server.php

```

1. use Hprose\Socket\Server;
2.
3. $server = new Server('tcp://0.0.0.0:1143/');
4. $server->addFilter(new StatFilter());
5.     ->addFunction(function($value) { return $value; }, 'echo')
6.     ->start();

```

Client.php

```

1. use Hprose\Client;
2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addFilter(new StatFilter());
5.
6. $value = range(0, 99999);
7. var_dump(count($client->echo($value)));

```

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. It takes 0.028308868408203 s.

```

客户端输出

```

1. It takes 0.03558087348938 s.
2. int(100000)

```

最后让我们把这个这个运行时间统计的例子跟上面的压缩例子结合一下，可以看到更详细的时间统计。

Server.php

```

1. use Hprose\Socket\Server;
2.
3. $server = new Server('tcp://0.0.0.0:1143/');
4. $server->addFilter(new StatFilter())
5.     ->addFilter(new SizeFilter('Non compressed'))
6.     ->addFilter(new CompressFilter())
7.     ->addFilter(new SizeFilter('Compressed'))
8.     ->addFilter(new StatFilter());
9.     ->addFunction(function($value) { return $value; }, 'echo')
10.    ->start();

```

Client.php

```

1. use Hprose\Client;
2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addFilter(new StatFilter())
5.     ->addFilter(new SizeFilter('Non compressed'))
6.     ->addFilter(new CompressFilter())
7.     ->addFilter(new SizeFilter('Compressed'))
8.     ->addFilter(new StatFilter());
9.
10. $value = range(0, 99999);
11. var_dump(count($client->echo($value)));

```

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. Compressed input size: 216266
2. Non compressed input size: 688893
3. It takes 0.0014259815216064 s.
4. It takes 0.031302928924561 s.
5. Non compressed output size: 688881
6. Compressed output size: 216245
7. It takes 0.055199861526489 s.

```

客户端输出

```
1. Non compressed output size: 688893
2. Compressed output size: 216266
3. It takes 0.023594856262207 s.
4. It takes 0.082166910171509 s.
5. Compressed input size: 216245
6. Non compressed input size: 688881
7. It takes 0.083509922027588 s.
8. int(100000)
```

在这里，我们可以看到客户端和服务端分别输出了三段用时。

服务器端输出：

第一个 `0.0014259815216064 s` 是解压缩输入数据的时间。

第二个 `0.031302928924561 s` 是第一个阶段用时 + 反序列化 + 调用 + 序列化的总时间。

第三个 `0.055199861526489 s` 是前两个阶段用时 + 压缩输出数据的时间。

客户端输出：

第一个 `0.023594856262207 s` 是压缩输出数据的时间。

第二个 `0.082166910171509 s` 是第一个阶段用时 + 从客户端调用发出到服务器端返回数据的总时间。

第三个 `0.083509922027588 s` 是前两个阶段用时 + 解压缩输入数据的时间。

协议转换

Hprose 过滤器的功能不止于此，如果你对 Hprose 协议本身有所了解的话，你还可以直接在过滤器中对输入输出数据进行解析转换。

在 Hprose for PHP 中已经提供了现成的 JSONRPC、XMLRPC 的过滤器。使用它，你可以将 Hprose 服务器变身为 Hprose + JSONRPC + XMLRPC 三料服务器。也可以将 Hprose 客户端变身为 JSONRPC 客户端或 XMLRPC 客户端。

Hprose + JSONRPC + XMLRPC 三料服务器

```

1. use Hprose\Socket\Server;
2. use Hprose\Filter\JSONRPC;
3. use Hprose\Filter\XMLRPC;
4.
5. function hello($name) {
6.     return "Hello $name!";
7. }
8.
9. $server = new Server('tcp://0.0.0.0:1143/');
10. $server->addFunction('hello')
11.     ->addFilter(new JSONRPC\ServiceFilter())
12.     ->addFilter(new XMLRPC\ServiceFilter())
13.     ->addFilter(new LogFilter())
14.     ->start();

```

实现一个三料服务器就这么简单，只需要添加一个协议转换的 `ServiceFilter` 实例对象就可以了。而且这个服务器可以同时接收 Hprose 和 JSONRPC、XMLRPC 三种请求。

JSONRPC 客户端

```

1. use Hprose\Client;
2. use Hprose\Filter\JSONRPC;
3.
4. $client = Client::create('tcp://127.0.0.1:1143/', false);
5. $client->addFilter(new JSONRPC\ClientFilter())
6.     ->addFilter(new LogFilter());
7.
8. var_dump($client->hello("world"));

```

XMLRPC 客户端

```

1. use Hprose\Client;
2. use Hprose\Filter\XMLRPC;
3.
4. $client = Client::create('tcp://127.0.0.1:1143/', false);
5. $client->addFilter(new XMLRPC\ClientFilter())
6.     ->addFilter(new LogFilter());
7.
8. var_dump($client->hello("world"));

```

客户端也是同样的简单，只需要添加一个协议转换的 `ClientFilter` 实例对象，Hprose 客户端就马上变身为对应协议的客户端了。不过需要注意一点，跟服务器不同，添加了 `JSONRPC\ClientFilter` 的客户端，是一个纯 JSONRPC 客户端，这个客户端只能跟 JSONRPC 服务器通讯，不能再跟纯 Hprose 服务器通讯了，但是跟 Hprose + JSONRPC 的双料服务器通讯是没问题的。

上面的程序我们先执行服务器，然后分别执行两个客户端，结果为：

服务器输出

```

1.  {"jsonrpc":"2.0","method":"hello","params":["world"],"id":1}
2.  {"id":1,"jsonrpc":"2.0","result":"Hello world!"}
3.  <?xml version="1.0" encoding="iso-8859-1"?>
4.  <methodCall>
5.  <methodName>hello</methodName>
6.  <params>
7.  <param>
8.    <value>
9.      <string>world</string>
10.    </value>
11.  </param>
12. </params>
13. </methodCall>
14.
15. <?xml version="1.0" encoding="utf-8"?>
16. <params>
17. <param>
18.   <value>
19.     <string>Hello world!</string>
20.   </value>
21. </param>
22. </params>

```

JSONRPC 客户端输出

```

1.  {"jsonrpc":"2.0","method":"hello","params":["world"],"id":1}
2.  {"id":1,"jsonrpc":"2.0","result":"Hello world!"}
3.  string(12) "Hello world!"

```

XMLRPC 客户端输出

```

1.  <?xml version="1.0" encoding="iso-8859-1"?>
2.  <methodCall>
3.  <methodName>hello</methodName>
4.  <params>
5.  <param>
6.    <value>
7.      <string>world</string>
8.    </value>
9.  </param>
10. </params>
11. </methodCall>
12.
13. <?xml version="1.0" encoding="utf-8"?>
14. <params>
15. <param>
16.   <value>
17.     <string>Hello world!</string>
18.   </value>

```

```
19. </param>
20. </params>
21.
22. string(12) "Hello world!"
```

Hprose 过滤器的功能很强大，除了上面这些用法之外，你还可以结合服务器事件来实现更为复杂的功能。不过这里就不再继续举例说明了。

原文: <https://github.com/hprose/hprose-php/wiki/11-Hprose-%E8%BF%87%E6%BB%A4%E5%99%A8>

12 Hprose 中间件

简介

Hprose 过滤器的功能虽然比较强大，可以将 Hprose 的功能进行扩展。但是有些功能使用它仍然难以实现，比如缓存。

为此，Hprose 2.0 引入了更加强大的中间件功能。Hprose 中间件不仅可以对输入输出的数据进行操作，它还可以对调用本身的参数和结果进行操作，甚至你可以跳过中间的执行步骤，或者完全由你来接管中间数据的处理。

Hprose 中间件跟普通的 HTTP 服务器中间件有些类似，但又有所不同。

Hprose 中间件在客户端服务器端都支持。

Hprose 中间件分为两种：

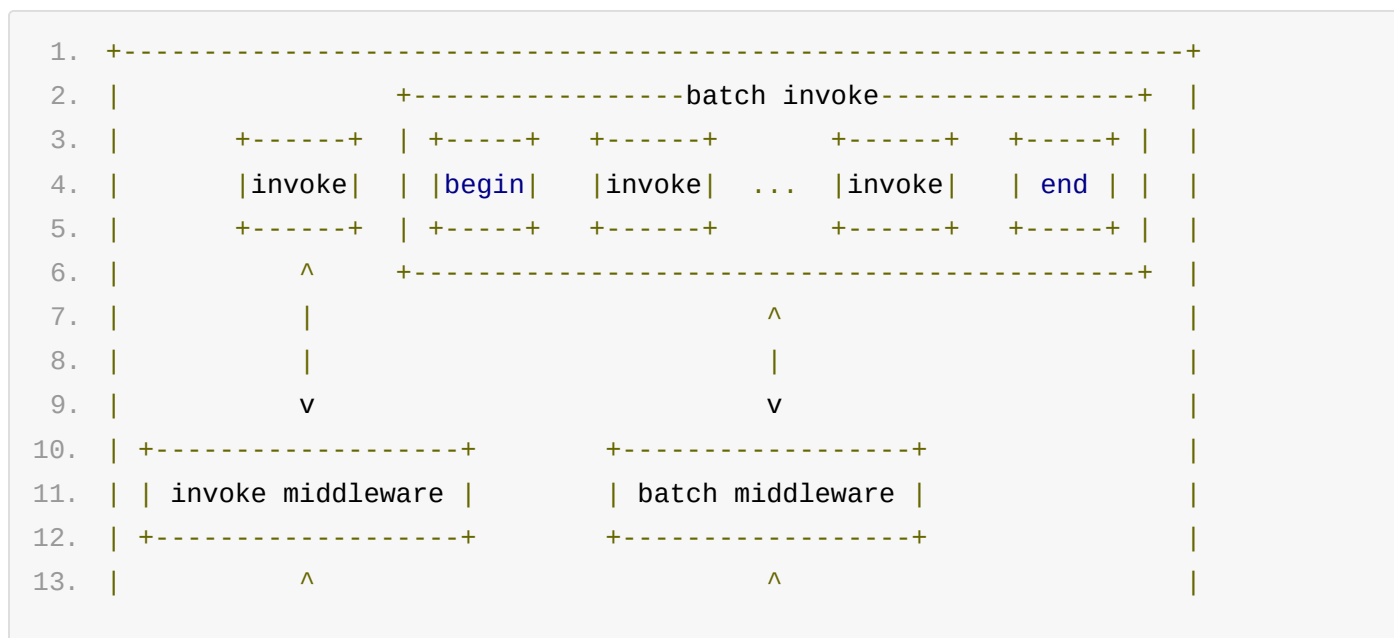
- 调用中间件
- 输入输出中间件

另外，输入输出中间件又可以细分为 `beforeFilter` 和 `afterFilter` 两种，但它们本质上没有什么区别，只是在执行顺序上有所区别。

执行顺序

Hprose 中间件的顺序执行是按照添加的前后顺序执行的，假设添加的中间件处理器分别为：`handler1`，`handler2` ... `handlerN`，那么执行顺序就是 `handler1`，`handler2` ... `handlerN`。

不同类型的 Hprose 中间件和 Hprose 其它过程的执行流程如下图所示：



- 114 -

中间件的返回值返回。

另外，对于服务器端来说，`$next` 的返回值 `$result` 总是 `promise` 对象。对于客户端来说，如果客户端是异步客户端，那么 `$next` 的返回值 `$result` 是 `promise` 对象，如果客户端是同步客户端，`$next` 的返回值 `$result` 是实际结果。

跟踪调试

我们来看一个例子：

LogHandler.php

```
1. use Hprose\Future;
2.
3. $logHandler = function($name, array &$args, stdClass $context, Closure $next) {
4.     error_log("before invoke:");
5.     error_log($name);
6.     error_log(var_export($args, true));
7.     $result = $next($name, $args, $context);
8.     error_log("after invoke:");
9.     if (Future\isFuture($result)) {
10.         $result->then(function($result) {
11.             error_log(var_export($result, true));
12.         });
13.     }
14.     else {
15.         error_log(var_export($result, true));
16.     }
17.     return $result;
18. };
```

Server.php

```
1. use Hprose\Socket\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server('tcp://0.0.0.0:1143/');
8. $server->addFunction('hello');
9. $server->debug = true;
10. $server->addInvokeHandler($logHandler);
11. $server->start();
```

Client.php

```
1. use Hprose\Client;
```

```

2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addInvokeHandler($logHandler);
5. var_dump($client->hello("world"));

```

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. before invoke:
2. hello
3. array (
4.     0 => 'world',
5. )
6. after invoke:
7. 'Hello world!'

```

客户端输出

```

1. before invoke:
2. hello
3. array (
4.     0 => 'world',
5. )
6. after invoke:
7. 'Hello world!'
8. string(12) "Hello world!"

```

这个例子中是使用的同步客户端，所以我们在 `$logHandler` 中判断了返回结果是同步结果还是异步结果，并根据结果的不同做了不同的处理。

如果我们的中间是单独为异步客户端或者服务器编写的，则不需要做这个判断。甚至我们还可以使用协程的方式来简化代码（需要 PHP 5.5+ 才可以）。

下面，我们来把上面的例子翻译成一个只针对异步客户端和服务器的使用协程方式编写的日志中间件：

coLogHandler.php

```

1. $coLogHandler = function($name, array &$args, stdClass $context, Closure $next)
2. {
    error_log("before invoke:");

```

```

3.     error_log($name);
4.     error_log(var_export($args, true));
5.     $result = (yield $next($name, $args, $context));
6.     error_log("after invoke:");
7.     error_log(var_export($result, true));
8. };

```

客户端和服务器的代码以及运行结果这里就省略了，如果你写的正确，运行结果跟上面的是一致的。

这个例子看上去要简单清爽的多，在这个例子中，我们使用 `yield` 关键字调用了 `$next` 方法，因为后面没有再次调用 `yield`，所以这个返回值也是该协程的返回值。

注意，不要用 `Future\wrap` 来包装这个协程，包装之后，不支持引用参数传递。

缓存调用

我们再来看一个实现缓存调用的例子，在这个例子中我们也使用了上面的日志中间件，用来观察我们的缓存是否真的有效。

CacheHandler.php

```

1. class CacheHandler {
2.     private $cache = array();
3.     function handle($name, array &$args, stdClass $context, Closure $next) {
4.         if (isset($context->userdata->cache)) {
5.             $key = hprose_serialize($args);
6.             if (isset($this->cache[$name])) {
7.                 if (isset($this->cache[$name][$key])) {
8.                     return $this->cache[$name][$key];
9.                 }
10.            }
11.            else {
12.                $this->cache[$name] = array();
13.            }
14.            $result = $next($name, $args, $context);
15.            $this->cache[$name][$key] = $result;
16.            return $result;
17.        }
18.        return $next($name, $args, $context);
19.    }
20. }

```

Client.php

```

1. use Hprose\Client;
2. use Hprose\InvokeSettings;
3.

```

```

4. $cacheSettings = new InvokeSettings(array("userdata" => array("cache" =>
    true)));
5. $client = Client::create('tcp://127.0.0.1:1143/', false);
6. $client->addInvokeHandler(array(new CacheHandler(), 'handle'));
7. $client->addInvokeHandler($logHandler);
8. var_dump($client->hello("cache world", $cacheSettings));
9. var_dump($client->hello("cache world", $cacheSettings));
10. var_dump($client->hello("no cache world"));
11. var_dump($client->hello("no cache world"));

```

我们的服务器仍然使用上面例子中的服务器。在确保服务器已启动的情况下，我们运行客户端，可以看到它们分别输出以下结果：

服务器输出

```

1. before invoke:
2. hello
3. array (
4.     0 => 'cache world',
5. )
6. after invoke:
7. 'Hello cache world!'
8. before invoke:
9. hello
10. array (
11.     0 => 'no cache world',
12. )
13. after invoke:
14. 'Hello no cache world!'
15. before invoke:
16. hello
17. array (
18.     0 => 'no cache world',
19. )
20. after invoke:
21. 'Hello no cache world!'

```

客户端输出

```

1. before invoke:
2. hello
3. array (
4.     0 => 'cache world',

```

```

5. )
6. after invoke:
7. 'Hello cache world!'
8. string(18) "Hello cache world!"
9. string(18) "Hello cache world!"
10. before invoke:
11. hello
12. array (
13.     0 => 'no cache world',
14. )
15. after invoke:
16. 'Hello no cache world!'
17. string(21) "Hello no cache world!"
18. before invoke:
19. hello
20. array (
21.     0 => 'no cache world',
22. )
23. after invoke:
24. 'Hello no cache world!'
25. string(21) "Hello no cache world!"

```

我们看到输出结果中 `'cache world'` 的日志只被打印了一次，而 `'no cache world'` 的日志被打印了两次。这说明 `'cache world'` 确实被缓存了。

在这个例子中，我们用到了 `userdata` 设置项和 `$context->userdata`，通过 `userdata` 配合 Hprose 中间件，我们就可以实现自定义选项功能了。

输入输出中间件

输入输出中间件可以完全代替 Hprose 过滤器。使用输入输出中间件还是使用 Hprose 过滤器完全看开发者喜好。

输入输出中间件的形式为：

```

1. function(string $request, stdClass $context, Closure $next) {
2.     ...
3.     $result = $next($request, $context);
4.     ...
5.     return $result;
6. }

```

`$request` 是原始请求数据，对于客户端来说它是输出数据，对于服务器端来说，它是输入数据。该数据的类型为 `string` 类型对象。

`$context` 是调用上下文对象。

`$next` 表示下一个中间件。通过调用 `$next` 将各个中间件串联起来。

`$next` 的返回值 `$response` 是返回的响应数据。对于客户端来说，它是输入数据。对于服务器端来说，它是输出数据。跟调用中间一样，服务器和异步客户端返回的 `$response` 是一个 `promise` 对象，而同步客户端返回的是一个 `string` 数据。

跟踪调试

下面我们来看一下 Hprose 过滤器中的跟踪调试的例子在这里如何实现。

logHandler2.php

```
1. use Hprose\Future;
2.
3. $logHandler2 = function($request, stdClass $context, Closure $next) {
4.     error_log($request);
5.     $response = $next($request, $context);
6.     Future\run('error_log', $response);
7.     return $response;
8. };
```

Server.php

```
1. use Hprose\Socket\Server;
2.
3. function hello($name) {
4.     return "Hello $name!";
5. }
6.
7. $server = new Server('tcp://0.0.0.0:1143/');
8. $server->addFunction('hello');
9. $server->debug = true;
10. $server->addBeforeFilterHandler($logHandler2);
11. $server->start();
```

Client.php

```
1. use Hprose\Client;
2.
3. $client = Client::create('tcp://127.0.0.1:1143/', false);
4. $client->addBeforeFilterHandler($logHandler2);
5. var_dump($client->hello("world"));
```

然后分别启动服务器和客户端，就会看到如下输出：

服务器输出

```
1. Cs5"hello"a1{s5"world"}z
2. Rs12"Hello world!"z
```

客户端输出

```
1. Cs5"hello"a1{s5"world"}z
2. Rs12"Hello world!"z
3. string(12) "Hello world!"
```

这个结果跟使用 Hprose 过滤器的例子的结果一模一样。

但是我们发现，这里使用 Hprose 中间件要写的代码比起 Hprose 过滤器来要多一些。主要原因是在 Hprose 中间件中，`$next` 的返回值为 `promise` 对象，需要异步处理，而 Hprose 过滤器只需要同步处理就可以了。在这个例子中，我们是直接使用 `Future\run` 来处理异步结果的。

另外，因为这个例子中，我们没有使用过滤器功能，因此使用 `addBeforeFilterHandler` 方法或者 `addAfterFilterHandler` 方法添加中间件处理器效果都是一样的。

但如果我们使用了过滤器的话，那么 `addBeforeFilterHandler` 添加的中间件处理器的 `$request` 数据是未经过过滤器处理的。过滤器的处理操作在 `$next` 的最后一环中执行。`$next` 返回的响应 `$response` 是经过过滤器处理的。

如果某个通过 `addBeforeFilterHandler` 添加的中间件处理器跳过了 `$next` 而直接返回了结果的话，则返回的 `$response` 也是未经过过滤器处理的。而且如果某个 `addBeforeFilterHandler` 添加的中间件处理器跳过了 `$next`，不但过滤器不会执行，而且在它之后使用 `addBeforeFilterHandler` 所添加的中间件处理器也不会执行，`addAfterFilterHandler` 方法所添加的所有中间件处理器也都不会执行。

而 `addAfterFilterHandler` 添加的处理器所收到的 `$request` 都是经过过滤器处理以后的，但它当中使用 `$next` 方法返回的 `$response` 是未经过过滤器处理的。

下面，我们在来看一个结合了压缩过滤器和输入输出缓存中间件的例子。

压缩、缓存、计时

CompressFilter.php

```

1. use Hprose\Filter;
2.
3. class CompressFilter implements Filter {
4.     public function inputFilter($data, stdClass $context) {
5.         return gzdecode($data);
6.     }
7.     public function outputFilter($data, stdClass $context) {
8.         return gzencode($data);
9.     }
10. }

```

上面的代码跟 Hprose 过滤器一章的压缩代码完全相同。

SizeHandler.php

```

1. class SizeHandler {
2.     private $message;
3.     public function __construct($message) {
4.         $this->message = $message;
5.     }
6.     public function asynchandle($request, stdClass $context, Closure $next) {
7.         error_log($this->message . ' request size: ' . strlen($request));
8.         $response = (yield $next($request, $context));
9.         error_log($this->message . ' response size: ' . strlen($response));
10.    }
11.    public function synchandle($request, stdClass $context, Closure $next) {
12.        error_log($this->message . ' request size: ' . strlen($request));
13.        $response = $next($request, $context);
14.        error_log($this->message . ' response size: ' . strlen($response));
15.        return $response;
16.    }
17. }

```

StatHandler.php

```

1. class StatHandler {
2.     private $message;
3.     public function __construct($message) {
4.         $this->message = $message;
5.     }
6.     public function asynchandle($name, array &$args, stdClass $context, Closure
    $next) {
7.         $start = microtime(true);
8.         yield $next($name, $args, $context);
9.         $end = microtime(true);
10.        error_log($this->message . ': It takes ' . ($end - $start) . ' s. ');
11.    }
12.    public function synchandle($name, array &$args, stdClass $context, Closure
    $next) {
13.        $start = microtime(true);

```

```

14.         $response = $next($name, $args, $context);
15.         $end = microtime(true);
16.         error_log($this->message . ': It takes ' . ($end - $start) . ' s. ');
17.         return $response;
18.     }
19. }

```

StatHandler2.php

```

1. class StatHandler2 {
2.     private $message;
3.     public function __construct($message) {
4.         $this->message = $message;
5.     }
6.     public function asynchandle($request, stdClass $context, Closure $next) {
7.         $start = microtime(true);
8.         yield $next($request, $context);
9.         $end = microtime(true);
10.        error_log($this->message . ': It takes ' . ($end - $start) . ' s. ');
11.    }
12.    public function synchandle($request, stdClass $context, Closure $next) {
13.        $start = microtime(true);
14.        $response = $next($request, $context);
15.        $end = microtime(true);
16.        error_log($this->message . ': It takes ' . ($end - $start) . ' s. ');
17.        return $response;
18.    }
19. }

```

这三个中间件，我们给它们分别编写了同步和异步处理程序。异步我们用的是协程方式，看上去跟同步一样简单。

CacheHandler2.php

```

1. class CacheHandler2 {
2.     private $cache = array();
3.     function handle($request, stdClass $context, Closure $next) {
4.         if (isset($context->userdata->cache)) {
5.             if (isset($this->cache[$request])) {
6.                 return $this->cache[$request];
7.             }
8.             $response = $next($request, $context);
9.             $this->cache[$request] = $response;
10.            return $response;
11.        }
12.        return $next($request, $context);
13.    }
14. }

```

缓存中间件因为不涉及到对结果的判断，所以同步和异步写法是一样的。

Server.php

```

1. use Hprose\Socket\Server;
2.
3. $server = new Server('tcp://0.0.0.0:1143/');
4. $server->addFunction(function($value) { return $value; }, 'echo')
5.     ->addBeforeFilterHandler(array(new StatHandler2("BeforeFilter"),
6.     'asynchandle'))
7.     ->addBeforeFilterHandler(array(new SizeHandler("compressedr"),
8.     'asynchandle'))
9.     ->addFilter(new CompressFilter())
10.    ->addAfterFilterHandler(array(new StatHandler2("AfterFilter"),
11.    'asynchandle'))
12.    ->addAfterFilterHandler(array(new SizeHandler("Non compressed"),
13.    'asynchandle'))
14.    ->addInvokeHandler(array(new StatHandler("Invoke"), 'asynchandle'))
15.    ->start();

```

在这里我们看到了发布方法，添加中间件，添加过滤器都支持链式调用。

Client.php

```

1. use Hprose\Client;
2. use Hprose\InvokeSettings;
3.
4. $cacheSettings = new InvokeSettings(array("userdata" => array("cache" =>
5.     true)));
6.
7. $client = Client::create('tcp://127.0.0.1:1143/', false);
8. $client->addBeforeFilterHandler(array(new CacheHandler2(), 'handle'))
9.     ->addBeforeFilterHandler(array(new StatHandler2('BeforeFilter'),
10.     'synchandle'))
11.     ->addBeforeFilterHandler(array(new SizeHandler('Non compressed'),
12.     'synchandle'))
13.     ->addFilter(new CompressFilter())
14.     ->addAfterFilterHandler(array(new StatHandler2('AfterFilter'),
15.     'synchandle'))
16.     ->addAfterFilterHandler(array(new SizeHandler('compressed'),
17.     'synchandle'))
18.     ->addInvokeHandler(array(new StatHandler("Invoke"), 'synchandle'));
19.
20. $value = range(0, 99999);
21. var_dump(count($client->echo($value, $cacheSettings)));
22. var_dump(count($client->echo($value, $cacheSettings)));

```

客户端添加过滤器和中间件也支持链式调用。

分别启动服务器和客户端，就会看到如下输出：

服务器输出

```

1. compressedr request size: 216266

```

```

2. Non compressed request size: 688893
3. Invoke: It takes 0.0062549114227295 s.
4. Non compressed response size: 688881
5. AfterFilter: It takes 0.039386034011841 s.
6. compressedr response size: 216245
7. BeforeFilter: It takes 0.066082954406738 s.

```

客户端输出

```

1. Non compressed request size: 688893
2. compressed request size: 216266
3. compressed response size: 216245
4. AfterFilter: It takes 0.068840026855469 s.
5. Non compressed response size: 688881
6. BeforeFilter: It takes 0.093698024749756 s.
7. Invoke: It takes 0.10785388946533 s.
8. int(100000)
9. Invoke: It takes 0.012754201889038 s.
10. int(100000)
11. >

```

我们可以看到两次的执行结果都出来了，但是中间件的输出内容只有一次。原因就是第二次执行时，缓存中间件将缓存的结果直接返回了。因此后面所有的步骤就都略过了。

通过这个例子，我们可以看出，将 Hprose 中间件和 Hprose 过滤器结合，可以实现非常强大的扩展功能。如果你有什么特殊的需求，直接使用 Hprose 无法实现的话，就考虑一下是否可以添加几个 Hprose 中间件和 Hprose 过滤器吧。

原文: <https://github.com/hprose/hprose-php/wiki/12-Hprose-%E4%B8%AD%E9%97%B4%E4%BB%B6>

附录A 2.0 新特征

Hprose 2.0 for PHP 新增了许多特征：

- 增加了数据推送的支持。
- oneway 调用支持。
- 增加了对幂等性 (idempotent) 调用自动重试的支持。
- 增加了异步调用支持。
- 客户端增加了负载均衡，故障切换的支持。
- 对客户端调用和服务端发布的 API 进行了优化，将多余的位置参数改为命名参数。
- 增加了新的中间件处理器支持，可以实现更强大的 AOP 编程。
- 新增 Promise 实现，不但完全实现了 [Promises/A+ 规范](#)，而且提供了许多功能强大，使用方便的 API。

原文：<https://github.com/hprose/hprose-php/wiki/%E9%99%84%E5%BD%95A-2.0-%E6%96%B0%E7%89%B9%E5%BE%81>

附录B Hprose 的 pecl 扩展

Hprose for PHP 有一个 pecl 扩

展：<https://github.com/hprose/hprose-pecl>

该扩展包含了 Hprose 序列化和反序列化部分的 C 语言实现，安装它之后可以有效提高 Hprose 的性能。但是它并不包括 Hprose 远程调用服务器和客户端的实现，因此，单纯安装它并不能代替 Hprose for PHP。

Windows 安装方式

可以直接从 pecl 官

网：<https://pecl.php.net/package/hprose/1.6.5/windows> 下载你需要的 Windows 版本的 dll，安装即可。

Linux 安装方式

可以直接通过：

```
1. pecl install hprose
```

方式安装，也可以下载源码之后，使用 phpize 安装。

Mac 安装方式

可以直接通过：

```
1. brew install phpXX-hprose
```

来安装。其中 XX 表示 PHP 的版本号。

也可以通过 pecl 命令安装，或者下载源码之后，通过 phpize 方式安装。

原文：<https://github.com/hprose/hprose-php/wiki/%E9%99%84%E5%BD%95B-Hprose-%E7%9A%84-pecl-%E6%89%A9%E5%B1%95>