Hprose for HTML5 用户手册

书栈(BookStack.CN)

目　录

致谢

当前文档 《Hprose for HTML5 用户手册》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-09。

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-html5

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。

Hprose for HTML5 用户手册

Hprose for HTML5 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 HTML5 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 HTML5 之旅吧。

> 原文: *https://github.com/hprose/hprose-html5/wiki*

2.0 新特征

Hprose 2.0 for HTML5 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(伪)同步调用支持。
- 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。
- 增强了批处理功能。
- 新的 Future 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。另外，还提供了对 ECMAScript 6 中 Promise 对象的模拟实现。

> 原文: *https://github.com/hprose/hprose-html5/wiki/2.0-%E6%96%B0%E7%89%B9%E5%BE%81*

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。

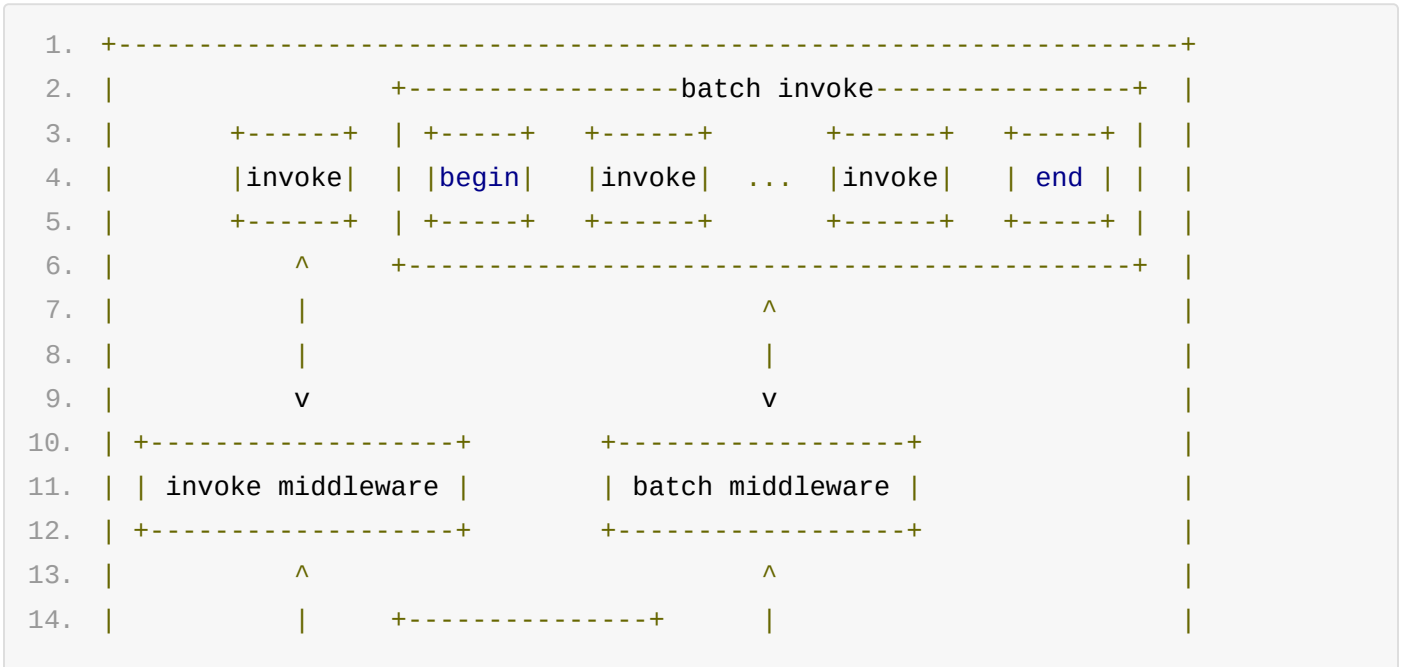
Hprose 客户端中间件分为三种：

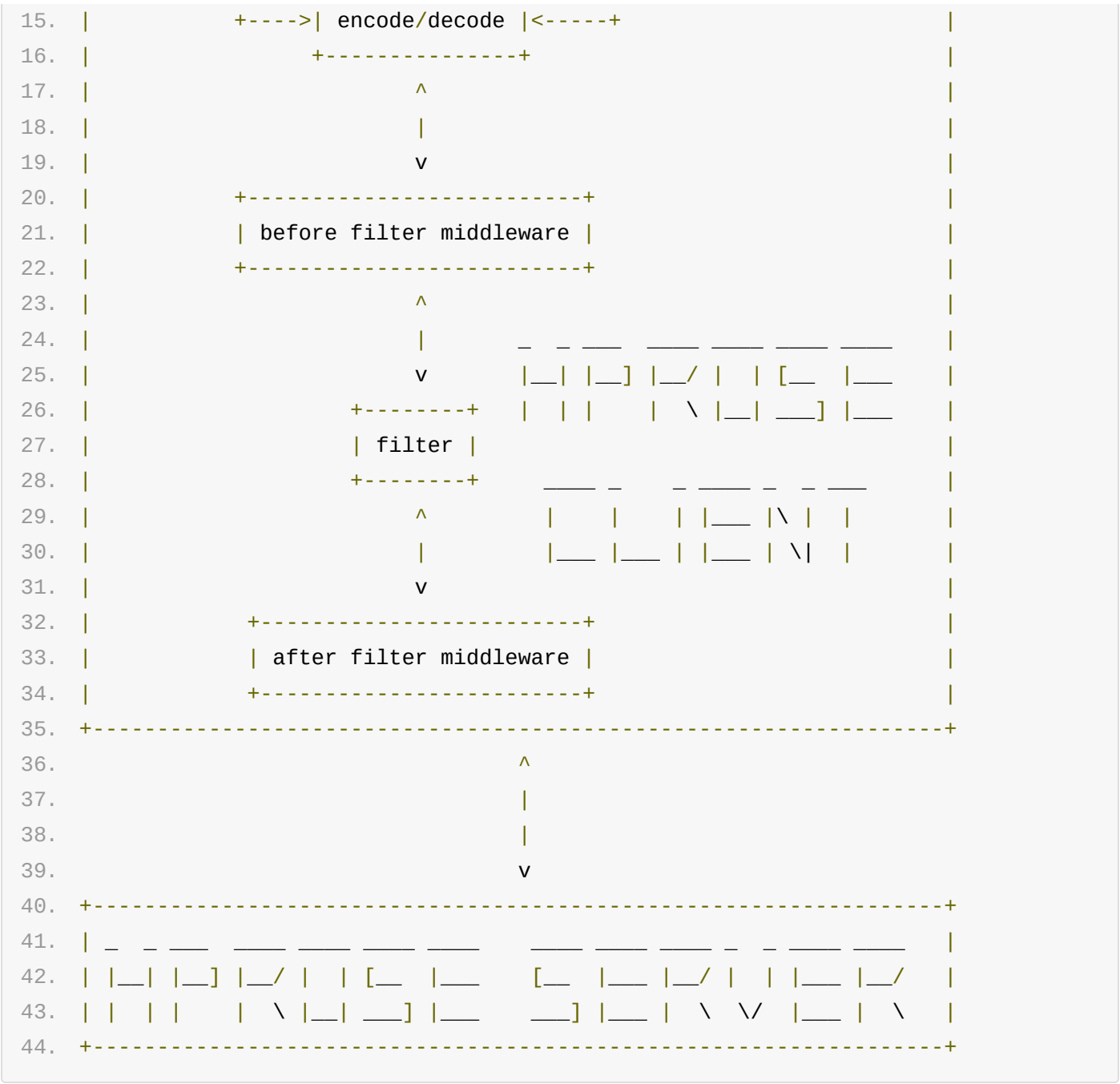
- 调用中间件
- 批处理调用中间件
- 输入输出中间件
  另外，输入输出中间件又可以细分为 `beforeFilter` 和 `afterFilter` 两种，但它们本质上没有什么区别，只是在执行顺序上有所区别。

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：





调用中间件

调用中间件的形式为：

```
function(name, args, context, next) {
    ...
    var result = next(name, args, context);
    ...
    return result;
}
```

`name` 是调用的远程函数/方法名。

`args` 是调用参数。

`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。

在调用 `next` 之前的操作在调用发生前执行，在调用 `next` 之后的操作在调用发生后执行，如果你不想修改返回结果，你应该将 `next` 的返回值作为该中间件的返回值返回。

跟踪调试

我们来看一个例子：

client.js

```
function loghandler(name, args, context, next) {
    console.log("before invoke:", name, args);
    var result = next(name, args, context);
    result.then(function(result) {
        console.log("after invoke:", name, args, result);
    });
    return result;
}
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.use(loghandler);
client.hello("world", function(result) {
    console.log(result);
});
```

客户端输出

```
before invoke: hello ["world"]
after invoke: hello ["world"] Hello world
Hello world
```

通过上面的输出，我们会发现结果 result 是个 promise 对象。但参数值 args 在 loghandler 里并不包含 Promise 的值，原因是 Hprose 内部已经对参数值处理过了。这样对于中间件编写就方便了很多，只需要处理异步结果就可以了。

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。

client.js

```
function loghandler(name, args, context, next) {
    console.log("before invoke:", name, args);
    var result = next(name, args, context);
    result.then(function(result) {
        console.log("after invoke:", name, args, result);
    });
    return result;
}
var cache = {};
function cachehandler(name, args, context, next) {
    if (context.userdata.cache) {
        var key = JSON.stringify(args);
        if (name in cache) {
            if (key in cache[name]) {
                return cache[name][key];
            }
        }
        else {
            cache[name] = {};
        }
        var result = next(name, args, context);
        cache[name][key] = result;
        return result;
    }
    return next(name, args, context);
}
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.use(cachehandler)
      .use(loghandler);
client.hello("cache world", function(result) {
    console.log(result);
}, { userdata: { cache: true } });
client.hello("cache world", function(result) {
    console.log(result);
}, { userdata: { cache: true } });
client.hello("no cache world", function(result) {
    console.log(result);
});
client.hello("no cache world", function(result) {
    console.log(result);
});
```

客户端输出

```
before invoke: hello ["cache world"]
before invoke: hello ["no cache world"]
before invoke: hello ["no cache world"]
```

```
after invoke: hello ["no cache world"] Hello no cache world
Hello no cache world
after invoke: hello ["no cache world"] Hello no cache world
Hello no cache world
after invoke: hello ["cache world"] Hello cache world
Hello cache world
Hello 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。

另外，我们在这个例子中可以看到，`use` 方法可以链式调用。

批处理调用中间件

上面的调用中间件对于批处理调用是不起作用的，因为批处理调用是单独处理的。

批处理调用中间件的形式为：

```
function(batches, context, next) {
    ...
    var result = next(batches, context);
    ...
    return result;
}
```

`batches` 是个数组。它的每个元素都是一个对象，该对象表示一个单独的调用，它包含有以下属性：

- name 是调用的远程函数/方法名。
- args 是调用的参数。
- context 是调用的上下文对象。
- resolve 用于返回成功结果的回调函数。
- reject 用于返回失败结果（异常）的回调函数。

`context` 是批处理调用的上下文对象。

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

在调用 `next` 之前的操作在批处理调用发生前执行，在调用 `next` 之后的操作在批处理调用发生后执行，如果你不想修改返回结果，你应该将 `next` 的返回值作为该中间件的返回值返回。

批处理跟踪调试

client.js

```
function batchloghandler(batches, context, next) {
    console.log("before invoke:", batches);
    var result = next(batches, context);
    result.then(function(result) {
        console.log("after invoke:", batches, result);
    });
    return result;
}
var log = hprose.Future.wrap(console.log, console);
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.batch.use(batchloghandler);
client.batch.begin();
var r1 = client.hello("world 1");
var r2 = client.hello("world 2");
var r3 = client.hello("world 3");
client.batch.end();
log(r1, r2, r3);
```

服务器端我们不修改，还用上面那个例子中的服务器。启动客户端之后，我们会看到客户端输出如下内容：

```
before invoke: [Object, Object, Object]
after invoke: [Object, Object, Object] [Object, Object, Object]
Hello world 1 Hello world 2 Hello world 3
```

这段输出这里就不需要解释了。

这里有一点要注意，那就是批处理调用中间件使用： `client.batch.use` 方法来添加，该方法也支持链式调用，链式调用方式为：

```
client.batch.use(handler1)
            .use(handler2)
            .use(handler3);
```

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：

```
function(request, context, next) {
    ...
    var response = next(request, context);
    ...
    return response;
}
```

`request` 是原始请求数据，对于客户端来说它是输出数据。该数据的类型为 `Uint8Array` 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。这个 `response` 必须为 `promise` 对象，该 `promise` 对象的成功值必须为 `Uint8Array` 类型的对象，如果是失败值，则失败原因可以是任意类型，但最好是 `Error` 类型（或子类型）的对象。

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。

client.js

```
function loghandler(request, context, next) {
    console.log(request);
    var response = next(request, context);
    response.then(function(data) {
        console.log(data);
    });
    return response;
}
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.beforeFilter.use(loghandler);
client.hello("world", function(result) {
    console.log(result);
});
```

然后分别启动服务器和客户端，就会看到如下输出：

客户端输出

```
Cs5"hello"a1{s5"world"}z
```

```
2. Rs12"Hello world!"z
3. 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，返回的响应对象是 `promise` 对象，需要单独处理。而 Hprose 过滤器则不需要。

另外，因为这个例子中，我们没有使用过滤器功能，因此使用 `beforeFilter.use` 方法或者 `afterFilter.use` 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 `beforeFilter.use` 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 `beforeFilter.use` 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 `beforeFilter.use` 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 `beforeFilter.use` 所添加的中间件处理器也不会执行，`afterFilter.use` 方法所添加的所有中间件处理器也都不会执行。

而 `afterFilter.use` 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。

关于 Hprose 中间件的更多用法，还可以参见 Hprose for Node.js 用户手册的相关内容。它们的实现是完全一样的。

> 原文: *https://github.com/hprose/hprose-html5/wiki/Hprose-%E4%B8%AD%E9%97%B4%E4%BB%B6*

Hprose 客户端

概述

Hprose 2.0 for HTML5 支持多种底层网络协议绑定的客户端，比如：HTTP 客户端，TCP 客户端和 WebSocket 客户端。

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

TCP 客户端支持跟 TCP 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。

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函数，另一种是使用工厂方法 `create` 。

使用构造器函数创建客户端

`hprose.Client` 是一个抽象类，因此它不能作为构造器函数直接使用。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.HttpClient` 、 `hprose.TcpClient` 和 `hprose.WebSocketClient` 这几个底层网络协议绑定客户端的实现源码。

`hprose.HttpClient` 、 `hprose.TcpClient` 和 `hprose.WebSocketClient` 这三个函数是可以直接使用的构造器函数。它们分别对应 http 客户端、tcp 客户端和 WebSocket 客户端。

```
1. new hprose.HttpClient([uri[, functions[, settings]]]);
2. new hprose.TcpClient([uri[, functions[, settings]]]);
3. new hprose.WebSocketClient([uri[, functions[, settings]]]);
```

这两个构造器的参数格式是相同的。开头的关键字 new 也可以省略，但最好不要省略。

构造器中包含了 3 个参数，这 3 个参数都可以省略。

当 3 个参数都省略时，创建的客户端是未初始化的，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 `uri` 是服务器地址，该服务器地址可以是单个的 `uri` 字符串，也可以是由多个 `uri` 字符串组成的数组。当该参数为多个 `uri` 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 `functions` 是远程函数名集合。它可以是单个函数名的字符串表示，也可以是多个函数名的字符串数组，还可以是一个对象。

第 3 个参数 `settings` 用于初始化客户端的设置。它可以初始化客户端的以下设置：

- failswitch
- timeout
- retry
- idempotent
- keepAlive
- byref
- simple
- useHarmonyMap
- filter
  这些设置都有对应的客户端属性，这里暂不解释，在后面介绍属性时，再分别介绍。

例如：

```
1.  var client = new hprose.HttpClient(uri, 'hello', { timeout: 20000 });
```

创建的 `client` 对象上，就有一个叫 `hello` 的远程方法，并且客户端的超时被初始化为 20000ms。

这个 `hello` 方法可以直接这样调用：

```
1.  var result = client.hello('world');
```

再举一例：

```
1.  var client = new hprose.HttpClient(uri, ['hello', 'sum']);
```

这样创建的 `client` 对象上，就有两个远程方法，他们分别是 `hello` 和 `sum`。

```
var client = new hprose.HttpClient(uri, {
    user: ['add', 'update', 'del', 'get']
});
```

这样创建的 `client` 对象上，就有一个叫 `user` 的对象，在这个 `user` 对象上有 4 个方法，他们分别是 `add`，`update`，`del`，`get`。

可以这样调用：

```
var result = client.user.get(id);
```

注意：这里的 `user.add`、`user.update`、`user.del` 和 `user.get` 分别对应服务器端发布的别名为：`user_add`，`user_update`，`user_del` 和 `user_get` 方法。

服务器端的别名可以通过 `_` 分隔成好几段，每一段都可以转换为 `.` 调用的方式。

另外，对象和数组方式还可以组合使用，例如下面这个复杂一点的例子：

```
var functions = {
    user: ['add', 'update', 'del', 'get'],
    order: [ 'add', 'update', 'del', 'get', {
        today: [ 'add', 'udate', 'del', 'get' ],
        yesterday: [ 'add', 'udate', 'del', 'get' ],
        tomorrow: [ 'add', 'udate', 'del', 'get' ]
    }]
};

var client = new hprose.HttpClient(uri, functions);
```

在上面的例子中：`client.order.add` 方法对应服务器端的 `order_add` 方法，而 `client.order.today.add` 方法则对应服务器端的 `order_today_add` 方法。

当然，如果方法有很多，像这样一个一个都列出来，或许有些麻烦。所以这个函数列表可以省略。

如果省略的话，想要直接使用方法名调用需要在 `client` 对象的 `ready` 方法回调中才能使用，例如：

```
var client = new hprose.HttpClient(uri);
client.ready(function(proxy) {
    proxy.hello('world', function(result) {
        console.log(result);
    });
});
```

因为当省略函数名列表时，客户端会向服务器端请求这个函数名列表，当获取到之

后才会将这些函数绑定到客户端对象上。而获取的过程是异步的，因此需要使用 `ready` 方法。

在省略函数名列表的情况下，对于 `user_add` 这样的方法，在旧版本中是不能使用 `user.add` 的方式调用的。

但是从 hprose-html5 v2.0.35 版本之后，也可以使用 `user.add` 这种方式了。

而且从 v2.0.36 版本之后，hprose-html5 已通过浏览器内置的 `Proxy` 实现了动态代理，无需再通过网络获取远程方法列表，但是请注意，**IE** 系列浏览器不支持此功能。

通过工厂方法 create 创建客户端

```
1. hprose.Client.create(uri[, functions[, settings]]);
2. hprose.HttpClient.create(uri[, functions[, settings]]);
3. hprose.TcpClient.create(uri[, functions[, settings]]);
4. hprose.WebSocketClient.create(uri[, functions[, settings]]);
```

与构造器函数不同，工厂方法 `create` 可以在 `hprose.Client` 上被调用，它会根据 `uri` 的协议来决定创建什么类型的客户端。

`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 `uri` 不能被省略。

`create` 方法与构造器函数还有一点不同，`create` 会检查 `uri` 的有效性（是指格式是否有效，而不是指服务器是否可以连通），而构造器函数不会检查。

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。

使用 `hprose.Client.create` 方法还有个好处，当你变更底层通讯协议不需要修改代码，只需要修改 `uri` 地址就可以了，而 `uri` 地址可以通过各种方式动态加载，因此更加灵活。

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. <protocol>://<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。

`<protocol>` 表示协议，它可以为以下取值：

- tcp
- tcp4
- tcp6
- tls
- tcps
- tcp4s
- tcp6s

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

`tcp4` 表示地址为 ipv4 的 tcp 协议。

`tcp6` 表示地址为 ipv6 的 tcp 协议。

`tls` 和 `tcps` 意义相同，表示安全的 tcp 协议，地址可以是 ipv6 地址，也可以是 ipv4 地址。如有必要，可设置客户端安全证书。

`tcp4s` 表示地址为 ipv4 的安全的 tcp 协议。

`tcp6s` 表示地址为 ipv6 的安全的 tcp 协议。

客户端属性

uri 属性

只读属性。该属性表示客户端当前所使用的地址。如果客户端在创建时设置了多个服务地址，该属性的值仅为这多个地址中当前正在使用中的那个地址。

uriList 属性

读写属性。该属性表示客户端可以使用的服务器地址列表。

id 属性

只读属性。该属性表示当前客户端在进行推送订阅时的唯一编号。在没有进行推送订阅或者使用自己指定 id 方式进行推送订阅时，该属性的值为 null。你也可以调用 `client'#';` 方法来手动从服务器端获取该 `id` 的值。

failswitch 属性

该属性为 `Boolean` 类型。默认值为 `false` 。

该属性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。

你也可以针对某个调用进行单独设置。

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。更新服务列表可以通过设置 `uriList` 属性来完成。

timeout 属性

该属性为整数类型，默认值为 `30000` ，单位是毫秒（ms）。

该属性表示当前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。

你也可以针对某个调用进行单独设置。

idempotent 属性

该属性为 `Boolean` 类型，默认值为 `false` 。

该属性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。

你也可以针对某个调用进行单独设置。

retry 属性

该属性为整数类型，默认值为 `10`。

该属性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。

你也可以针对某个调用进行单独设置。

byref 属性

该属性为 `Boolean` 类型，默认值为 `false`。

该属性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。

你也可以针对某个调用进行单独设置。

simple 属性

该属性为 `Boolean` 类型，默认值为 `false`。

该属性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、长整数、浮点数）、Boolean 值、字符串、二进制数据、日期时间等基本类型的数据或者不包含引用的数组、Map 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。

你也可以针对某个调用进行单独设置。

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。

useHarmonyMap 属性

该属性为 `Boolean` 类型，默认值为 `false`。

该属性表示调用所返回的数据中，如果包含有 Map 类型的数据，是否反序列化为 ECMAScript 6 中的 `Map` 类型对象。当该属性设置为 `false` 时（即默认

值），Map 类型的数据将会被反序列化为 `Object` 实例对象的数据。

除非 Map 中的键不是字符串类型，否则没必要将该属性设置为 `true`。

你也可以针对某个调用进行单独设置。

keepAlive 属性

该属性为 `Boolean` 类型，默认值为 `true`。

该属性表示客户端和服务器端之间是否保持长连接。该属性只对 WebSocket 客户端有效。

filter 属性

该属性可以为对象类型或对象数组类型。默认值为 null。

该属性的作用是可以设置一个或多个 `Filter` 对象。关于 `Filter` 对象，我们将作为单独的章节进行介绍，这里暂且略过。

客户端事件属性

onerror 事件

该事件属性为函数类型，默认值为空函数（即无任何操作的函数）。

当客户端调用发生错误时，如果没有为调用设置异常处理回调函数，则该属性事件将被回调。回调函数有两个参数，第一个参数是方法名（字符串类型），第二个参数是调用中发生的错误（通常为 `Error` 对象）。

onfailswitch 事件

该事件属性为函数类型，默认值为空函数（即无任何操作的函数）。

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。该事件的回调函数只有一个参数，即客户端对象本身。

客户端方法

addFilter 方法

```
client.addFilter(filter);
```

该方法同设置 `filter` 属性类似。该方法用于添加一个 `filter` 对象到 Filter 链的末尾。

removeFilter 方法

```
client.removeFilter(filter);
```

该方法同设置 `filter` 属性类似。该方法用于从 Filter 链中删除指定的 `filter` 对象。

useService 方法

```
client.useService();
client.useService(uri);
client.useService(functions);
client.useService(uri, functions);
```

该方法的用处是对于未初始化的客户端对象，进行后期初始化设置，或者用于变更服务器地址。

当未设置任何参数调用时，该客户端返回一个 `promise` 对象，该 `promise` 对象的成功值为远程服务代理对象。

当仅设置了 `uri` 参数时，跟上面的功能相同，但是会替换当前的 `uri` 设置。注意，这里的 `uri` 地址只能是单个的服务地址，而不能是服务地址数组列表。

`functions` 参数是一个服务方法列表，与创建客户端时的 `functions` 参数相同，但不能是单个的字符串方法名。当设置了该列表参数后，会直接返回一个远程服务代理对象。

例如：

```
var client = new hprose.HttpClient('http://www.hprose.com/example/',
['hello']);
client.hello("World", function(result) { console.log(result); });
```

跟下面的代码的效果完全相同。

```
var client = new hprose.HttpClient();
var proxy = client.useService('http://www.hprose.com/example/', ['hello']);
proxy.hello("World", function(result) { console.log(result); });
```

注意，这里的 `proxy` 对象跟 `client` 实际上是同一个对象。如果希望有所区别，可以在 `useService` 方法的最后加上一个 `true` 的参数。该参数表示创建一个不同于 `client` 的新的 `proxy` 对象。

ready 方法

前面在介绍创建客户端的时候已经介绍过了。它的作用是，当远程方法被动态绑定到客户端上后，触发 ready 中的回调函数，执行对远程方法的调用。

invoke 方法

使用远程方法名调用

```
1. client[name]([arg1, arg2, ... argn[, onsuccess[, onerror[, settings]]]]);
```

`name` 是要调用的远程函数/方法名。

`arg1` ... `argn` 是这个远程函数/方法的参数。如果这个方法没有参数，那就不需要写任何参数。有几个就写几个。参数不能是 `function` 类型，这一点不难理解，因为一个函数是不能作为参数传给远程服务器执行的。

`onsuccess` 是远程函数/方法调用成功时的回调函数。它是 `function` 类型。正是因为前面的参数不可能是 `function` 类型，因此这里只要遇到第一个是 `function` 类型的参数，那么就可以认为它是回调函数。

`onerror` 是远程函数/方法调用失败时的回调函数。它也是 `function` 类型。它跟在 `onsuccess` 之后，因此，如果你想传入一个处理失败情况的回调函数，那么 `onsuccess` 这个回调函数是不能省略的。

`settings` 是对该远程函数/方法的单独设置，这里面包括前面介绍属性时提到的那些可以单独设置的属性，还有几个是属性中不具有而特别针对调用时的设置。因为该参数是一个对象，因此，它之前的 `onsuccess` 参数也不能省略（但是 `onerror` 可以省略），否则它无法被识别为是远程方法的参数还是远程方法的设置。

使用 invoke 调用

```
1. client.invoke(name[, args[, onsuccess[, onerror[, settings]]]]);
```

该方法是客户端的最核心方法，它的功能就是进行远程调用，并返回一个表示结果的 `promise` 对象。

该方法与直接使用远程方法名调用功能类似。但有以下几点区别：

- 直接使用远程方法名调用时，参数 arg1…argn 中的参数可以是 promise 对象。直接使用 invoke 方法时，args 是参数数组，里面的元素不可以包含 promise 对象。
- 当同时使用远程方法名和 invoke 方法进行远程调用时，不管哪个写在前面，都是 invoke 方法先执行，原因是使用远程方法名调用时，会先对对参数中的 promise 对象进行取值操作，而 invoke 方法调用不会有这个过程。
- invoke 方法的 args 必须是数组类型，但可以省略，省略的话，被认为该调用没有参数。

- invoke 方法的 onsuccess 在省略的情况下，仍然可以带入 settings 参数。但直接使用远程方法名调用时，如果 onsuccess 省略，则不能带入 settings 参数。
  `settings` 参数可以包括以下设置:

- mode

- byref
- simple
- failswitch
- timeout
- idempotent
- retry
- oneway
- sync
- onsuccess
- onerror
- useHarmonyMap
- 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.Normal （或 hprose.ResultMode.Normal）
- hprose.Serialized （或 hprose.ResultMode.Serialized）
- hprose.Raw （或 hprose.ResultMode.Raw）
- hprose.RawWithEndTag （或 hprose.ResultMode.RawWithEndTag）
  `hprose.Normal` 是默认值，表示返回正常的已被反序列化的结果。

`hprose.Serialized` 表示返回的结果保持序列化的格式。

`hprose.Raw` 表示返回原始数据。

`hprose.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：

```
var BytesIO = hprose.BytesIO;
var client = hprose.Client.create('http://www.hprose.com/example/', ['hello']);
function onsuccess(result) {
    console.log(result.constructor, BytesIO.toString(result));
}
client.hello("World", onsuccess, { mode: hprose.Normal, sync: true });
client.hello("World", onsuccess, { mode: hprose.Serialized, sync: true });
client.hello("World", onsuccess, { mode: hprose.Raw, sync: true });
client.hello("World", onsuccess, { mode: hprose.RawWithEndTag, sync: true });
```

为了保证执行顺序，这里还加了一个 `sync` 设置，对于该设置在后面再做详细解释。

该程序执行结果如下：

```
[Function: String] 'Hello World'
[Function: Uint8Array] 's11"Hello World"'
[Function: Uint8Array] 'Rs11"Hello World"'
[Function: Uint8Array] 'Rs11"Hello World"z'
```

由于历史原因，为了兼容旧版本的 hprose 的写法，该设置也可以不写在 `settings` 对象中，例如上面程序还可以这样写：

```
var BytesIO = hprose.BytesIO;
var client = hprose.Client.create('http://www.hprose.com/example/', ['hello']);
function onsuccess(result) {
    console.log(result.constructor, BytesIO.toString(result));
}
client.hello("World", onsuccess, hprose.Normal, { sync: true });
client.hello("World", onsuccess, hprose.Serialized, { sync: true });
client.hello("World", onsuccess, hprose.Raw, { sync: true });
client.hello("World", onsuccess, hprose.RawWithEndTag, { sync: true });
```

但在新版本中，不再推荐这种写法。

byref

该设置表示调用是否为引用参数传递方式。例如：

```
var client = hprose.Client.create('http://www.hprose.com/example/',
                                  ['swapKeyAndValue']);
```

```
var weeks = {
    'Monday': 'Mon',
    'Tuesday': 'Tue',
    'Wednesday': 'Wed',
    'Thursday': 'Thu',
    'Friday': 'Fri',
    'Saturday': 'Sat',
    'Sunday': 'Sun',
};
function onsuccess(result, args) {
    console.log(weeks.constructor, weeks);
    console.log(result.constructor, result);
    console.log(args.constructor, args);
}
client.swapKeyAndValue(weeks, onsuccess, { byref: true });
```

该程序执行结果为：

```
[Function: Object] { Monday: 'Mon',
  Tuesday: 'Tue',
  Wednesday: 'Wed',
  Thursday: 'Thu',
  Friday: 'Fri',
  Saturday: 'Sat',
  Sunday: 'Sun' }
[Function: Object] { Mon: 'Monday',
  Tue: 'Tuesday',
  Wed: 'Wednesday',
  Thu: 'Thursday',
  Fri: 'Friday',
  Sat: 'Saturday',
  Sun: 'Sunday' }
[Function: Array] [ { Mon: 'Monday',
    Tue: 'Tuesday',
    Wed: 'Wednesday',
    Thu: 'Thursday',
    Fri: 'Friday',
    Sat: 'Saturday',
    Sun: 'Sunday' } ]
```

我们可以看到在回调方法中的 `args` 参数被改变了，但是原来的参数对象 `weeks` 并没有被改变。也就是说，这里的引用参数传递只体现在回调函数返回的参数上，对原始的参数并不会修改。

同样，由于历史原因，为了兼容旧版本的 hprose 的写法，该设置也可以不写在

`settings` 对象中，而直接将 `true` 跟在 `onsuccess` 之后也是可以的。

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，回调函数将立即被调用，结果被设置为 `undefined`。

sync

该设置表示当前调用是否为“同步”调用。这里的“同步”调用是伪同步。它仅表示在该调用之后，同一个客户端所发起的其它远程调用一定是在本次调用执行完之后才会被调用。它可以保证几个连续的调用将按书写顺序执行。但每个调用本身还是异步的。

onsuccess

该设置表示调用成功时的回调函数，跟 `invoke` 方法的 `onsuccess` 参数是一个意思。它通常不会在 `settings` 参数中设置，因为在使用方法名调用时，没有

`onsuccess` 参数的情况下，无法传递 `settings` 参数。但是如果在 `settings` 参数中也设置了该属性，那么它将会覆盖 `invoke` 方法的 `onsuccess` 参数的设置。

onerror

该设置表示调用失败时的回调函数，跟 `invoke` 方法的 `onerror` 参数是一个意思。它通常也不会在 `settings` 参数中设置。但是如果在 `settings` 参数中也设置了该属性，那么它将会覆盖 `invoke` 方法的 `onerror` 参数的设置。

useHarmonyMap

该设置跟前面介绍的 `useHarmonyMap` 属性功能相同，但只针对当前调用有效。

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。例如上面那个引用参数传递的例子还可以写成这样：

```
var client = hprose.Client.create('http://www.hprose.com/example/',
                                  ['swapKeyAndValue']);
var weeks = {
    'Monday': 'Mon',
    'Tuesday': 'Tue',
    'Wednesday': 'Wed',
    'Thursday': 'Thu',
    'Friday': 'Fri',
    'Saturday': 'Sat',
    'Sunday': 'Sun',
};

client.swapKeyAndValue.onsuccess = function(result, args) {
    console.log(weeks.constructor, weeks);
    console.log(result.constructor, result);
    console.log(args.constructor, args);
};

client.swapKeyAndValue.byref = true;

client.swapKeyAndValue(weeks);
```

运行结果是一样的。这里就不在重复了。

链式调用

因为 invoke 方法的返回值是一个 `promise` 对象，因此它可以进行链式调用，例如：

```
var client = hprose.Client.create('http://www.hprose.com/example/', ['sum']);
client.sum(1, 2)
      .then(function(result) {
          return client.sum(result, 3);
      })
      .then(function(result) {
          return client.sum(result, 4);
      })
      .then(function(result) {
          console.log(result);
      });
```

该程序的结果为：

```
10
```

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 `promise` 对象。

因此，上面的链式调用还可以直接简化为：

```
var client = hprose.Client.create('http://www.hprose.com/example/', ['sum']);
client.sum(client.sum(client.sum(1, 2), 3), 4).then(function(result) {
    console.log(result);
});
```

这比上面的链式调用更加直观。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：

```
var hprose = require('hprose'),
    wrap = hprose.Future.wrap,
    client = hprose.Client.create('http://www.hprose.com/example/', ['sum']),
    log = wrap(console.log, console),
    r1 = client.sum(1, 3, 5, 7, 9),
    r2 = client.sum(2, 4, 6, 8, 10),
    r3 = client.sum(r1, r2);
log(r1, r2, r3);
```

这个程序的运行结果为：

```
25 30 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。

批处理调用

```
client.batch.begin();
...
client.batch.end([settings]);
```

通过这两个方法，可以实现批处理调用。例如：

```
var BytesIO = hprose.BytesIO;
var client = hprose.Client.create('http://www.hprose.com/example/', ['hello']);
function onsuccess(result) {
    console.log(result.constructor, BytesIO.toString(result));
}
client.batch.begin();
client.hello("World", onsuccess, { mode: hprose.Normal });
client.hello("World", onsuccess, { mode: hprose.Serialized });
client.hello("World", onsuccess, { mode: hprose.Raw });
client.hello("World", onsuccess, { mode: hprose.RawWithEndTag });
client.batch.end({ idempotent: true });
```

运行结果为：

```
[Function: String] 'Hello World'
[Function: Uint8Array] 's11"Hello World"'
[Function: Uint8Array] 'Rs11"Hello World"'
```

```
4. [Function: Uint8Array] 'Rs11"Hello World"z'
```

需要注意的是，虽然这里结果的显示顺序跟调用的顺序完全相同，但是这并不代表在服务器端这些方法是顺序执行的。

批处理调用的功能是可以在一个请求中同时发送多个调用，这多个调用在服务器端并发执行，最后汇总结果一次性返回。

因此，批处理调用的方法之间不能有前后依赖的顺序。

因此前面这个例子：

```
1. var hprose = require('hprose'),
2.     wrap = hprose.Future.wrap,
3.     client = hprose.Client.create('http://www.hprose.com/example/', ['sum']),
4.     log = wrap(console.log, console),
5.     r1 = client.sum(1, 3, 5, 7, 9),
6.     r2 = client.sum(2, 4, 6, 8, 10),
7.     r3 = client.sum(r1, r2);
8. log(r1, r2, r3);
```

`r3` 和 `r1`、`r2` 之间有先后依赖关系，因此，这三个调用是不能同时放在批处理调用中的。但是 `r1` 和 `r2` 之间并没有依赖关系，所以，它们两个可以放在批处理调用中。

另外需要注意的一点是，在批处理调用的设置中，以下选项仅在每个单独的调用中设置有效：

- mode
- byref
- simple
- onsuccess
- onerror

- useHarmonyMap
  以下选项仅在 `endBatch` 方法中设置有效：

- failswitch

- timeout
- idempotent
- retry
- oneway
- sync
  另外，还有一个 `userdata` 选项比较特殊，它在两个里面都可以设置，但两处设置所在的上下文是不同的。

> 
>
> - - - 在 *hprose 1.4* 到 *1.6* 的 *HTML5* 版本中也实验性的加入了批处理调用支持，旧版本中批处理的 *API* 是 `beginBatch` 和 `endBatch` 这两个方法，这两个方法都没有参数。这个功能在这些旧版本中有很多限制，因此不推荐在旧版本中使用该功能。在 *hprose 2.0* 中 `beginBatch` 和 `endBatch` 这两个方法已废止。请使用新的 `batch.begin` 和 `batch.end` 方法来代替。- - -

> 原文: *https://github.com/hprose/hprose-html5/wiki/Hprose-%E5%AE%A2%E6%88%B7%E7%AB%AF*

Hprose 序列化

概述

Hprose 提供了一套自己的序列化格式用来实现高效的跨语言跨平台的数据存储和交换。该序列化格式，在 hprose for HTML5 中被实现为以下几个对象：

- hprose.Tags
- hprose.ClassManager
- hprose.Writer
- hprose.Reader
- hprose.Formatter
  其中 `hprose.Tags` 对象中包含了所有的 Hprose 序列化和 RPC 标记定义。Hprose 的使用者通常不需要关心该对象，因此这里不对该对象做详细介绍。

`hprose.ClassManager` 用于管理自定义类型与其它语言之间的映射关系。

`hprose.Writer` 用于进行细粒度的 Hprose 序列化操作。

`hprose.Reader` 用于进行细粒度的 Hprose 反序列化操作。

`hprose.Formatter` 用于进行粗粒度的 Hprose 序列化和反序列化操作。

另外，`hprose` 对象上也提供了三个帮助方法，用于注册自定义类型（`register`），序列化（`serialize`）和反序列化（`unserialize`）数据。下面我们将对这几个对象和方法进行详细的介绍。

hprose.ClassManager

register 方法

```
1.  hprose.ClassManager.register(class, alias);
```

在 javascript 中，是没有类的概念的，这里的类是指对象的构造器函数。

当 hprose 序列化对象时，需要知道对象的类名，但有时候，对象的构造器函数可能是个匿名函数，并没有函数名。或者我们在不同的语言中定义的类名可能不同，但结构相同或相近，我们希望这些定义不同的类的对象可以相互传递。那么就需要使用该方法来进行注册，注册成统一的别名之后，就可以相互传递了。

其中 `class` 表示要注册的类（对象的构造器函数），`alias` 表示注册的别

名。例如：

```
var User = function() {
    this.name = 'Tom';
    this.age = 18;
}

hprose.ClassManager.register(User, 'User');
```

在有些语言中，类名是有名称空间（NameSpace）的，例如在 java 中，可能有这样一个类：my.package.User，我们希望它能跟这里的 User 进行交互，那我们应该这样做：

```
hprose.ClassManager.register(User, 'my_package_User');
```

注意上面的别名中，不是使用 `.` 做分隔符的，而是使用 `_`，*hprose* 之所以这样做是因为有些语言不支持名称空间（*NameSpace*），还有些语言的名称空间（*NameSpace*）和类名之间不是使用 `.` 分隔符的，因此这里统一成 `_`，这样既可以支持没有名称空间（NameSpace）的语言，也可以支持具有名称空间（NameSpace）的语言。

因为该方法比较常用，所以 hprose 还提供了一个简单的写法：

```
hprose.register(User, 'my_package_User');
```

getClassAlias 方法

```
hprose.ClassManager.getClassAlias(class);
```

通过类来查找别名，例如：

```
hprose.register(User, 'my_package_User');
console.log(hprose.ClassManager.getClassAlias(User));
```

输出结果为：

```
my_package_User
```

getClass 方法

```
1.  hprose.ClassManager.getClass(alias);
```

通过别名来查找注册的类。例如：

```
1.  var User = hprose.ClassManager.getClass('my_package_User');
```

hprose.Writer

构造器方法

```
1.  var writer = new Writer(stream[, simple]);
```

第一个参数 `stream` 是一个 `BytesIO` 的实例对象，序列化数据将会写入到该对象中。

第二个参数 `simple` 如果为 `true`，则不使用引用方式序列化，通常在序列化的数据中不包含引用类型数据时，设置为 `true` 可以加快速度。当包含引用类型数据时，需要设置为 `false`（即默认值），尤其是当引用数据中包括递归数据时，如果不使用引用方式，会陷入死循环导致堆栈溢出的错误。

stream 属性

只读属性，返回当前用于写入序列化数据的 `BytesIO` 实例对象。该属性的值即上面构造器中的第一个参数。

serialize 方法

```
1.  writer.serialize(value);
```

序列化数据 `value`。其中 `value` 为任意 hprose 支持的类型。

writeInteger 方法

```
1.  writer.writerInteger(value);
```

序列化一个整数 `value`。其中 `value` 为一个 32 位有符号整型数。超过这个范围的整数按照长整型数格式序列化。

writeDouble 方法

```
writer.writeDouble(value);
```

序列化一个浮点数 `value` 。

writeBoolean 方法

```
writer.writeBoolean(value);
```

序列化一个布尔值 `value` 。

writeUTCDate 方法

```
writer.writeUTCDate(value);
```

序列化一个 UTC 日期时间值 `value` 。其中 `value` 为 `Date` 类型的实例对象。

```
writer.writeUTCDateWithRef(value);
```

序列化一个 UTC 日期时间值 `value` ，如果该值之前被序列化过，则作为引用序列化。

writeDate 方法

```
writer.writeDate(value);
```

序列化一个本地日期时间值 `value` 。其中 `value` 为 `Date` 类型的实例对象。

```
writer.writeDateWithRef(value);
```

序列化一个本地日期时间值 `value` ，如果该值之前被序列化过，则作为引用序列化。

writeTime 方法

```
writer.writeTime(value);
```

序列化一个本地时间值 `value`。其中 `value` 为 `Date` 类型的实例对象。

```
writer.writeTimeWithRef(value);
```

序列化一个本地时间值 `value`，如果该值之前被序列化过，则作为引用序列化。

writeBytes 方法

```
writer.writeBytes(value);
```

序列化一个二进制数据值 `value`。其中 `value` 为 `ArrayBuffer`、`Uint8Array`、`BytesIO` 类型的实例对象或是元素是纯整数且数字范围是 0-255 的普通数组。

```
writer.writeBytesWithRef(value);
```

序列化一个二进制数据值 `value`，如果该值之前被序列化过，则作为引用序列化。

writeString 方法

```
writer.writeString(value);
```

序列化一个字符串值 `value`。

```
writer.writeStringWithRef(value);
```

序列化一个字符串值 `value`，如果该值之前被序列化过，则作为引用序列化。

writeList 方法

```
writer.writeList(value);
```

序列化一个 List 值 `value`。其中 `value` 是一个普通数组或者 TypedArray 类型的实例对象。

> 注意：对 **Uint8Array** 类型的实例对象使用 **writeBytes** 会更加高效。

```
writer.writeListWithRef(value);
```

序列化一个 List 值 `value`，如果该值之前被序列化过，则作为引用序列化。

writeMap 方法

```
writer.writeMap(value);
```

序列化一个 Map 值 `value`。其中 `value` 是一个普通的 javascript 对象（比如直接使用 JSON 形式创建的对象）或 ECMAScript 6 的 Map 对象。

```
writer.writeMapWithRef(value);
```

序列化一个 Map 值 `value`，如果该值之前被序列化过，则作为引用序列化。

writeObject 方法

```
writer.writeObject(value);
```

序列化一个对象值 `value`。其中 `value` 是一个对象。该对象的构造器已经通过 `hprose.ClassManager.register` 方法注册，或者该对象的构造器是一个非匿名函数。

```
writer.writeObjectWithRef(value);
```

序列化一个对象值 `value`，如果该值之前被序列化过，则作为引用序列化。

reset 方法

```
writer.reset();
```

将序列化的引用计数器重置。

hprose.Reader

构造器方法

```
var reader = new Reader(stream[, simple[, useHarmonyMap]]);
```

第一个参数 `stream` 是一个 `BytesIO` 的实例对象，反序列化数据将会从该对

象中读取。

第二个参数 `simple` 如果为 `true`，则不使用引用方式反序列化，通常在反序列化的数据中不包含引用类型数据时，设置为 `true` 可以加快速度。当包含引用类型数据时，需要设置为 `false`（即默认值），否则会抛出异常。

第三个参数 `useHarmonyMap` 如果为 `true`，则反序列化 Map 时，会以 ECMAScript 6 中的 Map 类型的实例对象返回反序列化值。否则返回 Object 对象。当反序列化的 Map 的键（key）为非字符串以外的其它类型（或多种混合类型）时，将该参数设置为 `true` 可以得到更准确的反序列化结果。默认值为 `false`。

stream 属性

只读属性，返回当前用于读取反序列化数据的 `BytesIO` 实例对象。该属性的值即上面构造器中的第一个参数。

useHarmonyMap 属性

功能同上面的第三个参数一样，可以在对象创建之后进行设置。

checkTag 方法

```
1.  reader.checkTag(expectTag[, tag]);
```

如果第二个参数不存在，则自动读取当前流中的一个字节作为 `tag` 值。如果 `expectTag` 和 `tag` 不一致，则抛出异常。

该方法没有返回值。

checkTags 方法

```
1.  reader.checkTags(expectTags[, tag]);
```

如果第二个参数不存在，则自动读取当前流中的一个字节作为 `tag` 值。如果 `expectTags` 中不包含 `tag`，则抛出异常。

`expectTags` 是一个数组，该数组中包含一个或多个 hprose.Tags 中的枚举值，当然也可以是其它 0-255 的任意整数，但通常不会用其它取值。

如果该方法执行成功，返回 `tag` 值。

unserialize 方法

```
1.  reader.unserialize();
```

从当前数据流中读取数据并返回反序列化结果。如果当前数据流中包含有多个序列化数据，则一次只返回一个结果。

如果反序列化过程中发生错误，则会抛出异常。

readInteger 方法

```
1.  reader.readInteger();
```

从当前数据流中反序列化一个整数结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

readLong 方法

```
1.  reader.readLong();
```

从当前数据流中反序列化一个长整数结果并返回，如果该结果超出 JavaScript 可表示的整数范围，则以字符串形式返回结果。

如果反序列化过程中发生错误，则会抛出异常。

readDouble 方法

```
1.  reader.readDouble();
```

从当前数据流中反序列化一个浮点数结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

readBoolean 方法

```
1.  reader.readBoolean();
```

从当前数据流中反序列化一个布尔值结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

readDate 方法

```
1.  reader.readDate();
```

从当前数据流中反序列化一个 Date 结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readDateWithoutTag();
```

从当前数据流中反序列化一个 Date 结果并返回，该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagDate。

如果反序列化过程中发生错误，则会抛出异常。

readTime 方法

```
1.  reader.readTime();
```

从当前数据流中反序列化一个只包含时间的 Date 结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readTimeWithoutTag();
```

从当前数据流中反序列化一个只包含时间的 Date 结果并返回，该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagTime。

如果反序列化过程中发生错误，则会抛出异常。

readBytes 方法

```
1.  reader.readBytes();
```

从当前数据流中反序列化一个二进制数据结果并返回。返回结果为 `Uint8Array` 类型的实例对象。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readBytesWithoutTag();
```

从当前数据流中反序列化一个二进制数据结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagBytes。

如果反序列化过程中发生错误，则会抛出异常。

readString 方法

```
1.  reader.readString();
```

从当前数据流中反序列化一个字符串结果并返回。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readStringWithoutTag();
```

从当前数据流中反序列化一个字符串结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagString。

如果反序列化过程中发生错误，则会抛出异常。

readGuid 方法

```
1.  reader.readGuid();
```

从当前数据流中反序列化一个 GUID 结果并返回，结果为字符串类型。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readGuidWithoutTag();
```

从当前数据流中反序列化一个 GUID 结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagGuid。

如果反序列化过程中发生错误，则会抛出异常。

readList 方法

```
1.  reader.readList();
```

从当前数据流中反序列化一个 List 结果并返回，结果为数组类型。

如果反序列化过程中发生错误，则会抛出异常。

```
1.  reader.readListWithoutTag();
```

从当前数据流中反序列化一个 List 结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagList。

如果反序列化过程中发生错误，则会抛出异常。

readMap 方法

```
1. reader.readMap();
```

从当前数据流中反序列化一个 Map 结果并返回，结果为 Object 类型的实例对象或 ECMAScript 6 的 Map 类型的实例对象。

如果反序列化过程中发生错误，则会抛出异常。

```
1. reader.readMapWithoutTag();
```

从当前数据流中反序列化一个 Map 结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagMap。

如果反序列化过程中发生错误，则会抛出异常。

readObject 方法

```
1. reader.readObject();
```

从当前数据流中反序列化一个对象结果并返回。该对象的构造器会通过以下方式查找或生成：

如果已经通过 `hprose.ClassManager.register` 方法注册，并可以查找到，直接返回。

如果在全局通过遍历搜索可以查找到同名构造器，则自动注册，并返回。

通过将读取到的类名中的 `_` 分别替换成 `.` 来全局遍历查找同名构造器，如果找到，则自动注册，并返回。

如果通过以上方式都查找不到，则自动生成一个构造器，然后自动注册，并返回。

自动生成的构造器上会添加一个 getClassName 方法，该方法会返回反序列化时读取到的类名。

如果反序列化过程中发生错误，则会抛出异常。

```
1. reader.readObjectWithoutTag();
```

从当前数据流中反序列化一个对象结果并返回。该方法假设序列化标记已被读取，并且其值为 hprose.Tags.TagObject。

如果反序列化过程中发生错误，则会抛出异常。

reset 方法

```
1. reader.reset();
```

将反序列化的引用计数器重置。

hprose.Formatter

serialize 方法

```
1. hprose.Formatter.serialize(value[, simple]);
```

该方法会将 `value` 进行序列化并返回序列化后的数据。如果 `simple` 为 `true`，则不使用引用方式序列化，通常在序列化的数据 `value` 中不包含引用类型数据时，设置为 `true` 可以加快速度。当包含引用类型数据时，需要设置为 `false`（即默认值），尤其是当引用数据中包括递归数据时，如果不使用引用方式，会陷入死循环导致堆栈溢出的错误。

返回结果为 Uint8Array 类型的对象实例。

如果序列化过程中发生错误，则会抛出异常。

hprose 还提供了一个简化写法：

```
1. hprose.serialize(value[, simple]);
```

注意：该简化写法跟完整写法返回值类型不同，简化写法的返回值为 BytesIO 类型的对象实例。

unserialize 方法

```
1. hprose.Formatter.unserialize(stream[, simple[, useHarmonyMap]]);
```

从数据流 `stream` 中读取数据并返回反序列化结果。如果当前数据流中包含有多个序列化数据，则一次只返回一个结果。

`stream` 为反序列化的数据来源，它可以是一个 `BytesIO` 对象，也可以是一个

`ArrayBuffer` 或 `Uint8Array` 对象。

第二个参数 `simple` 如果为 `true`，则不使用引用方式反序列化，通常在反序列化的数据中不包含引用类型数据时，设置为 `true` 可以加快速度。当包含引用类型数据时，需要设置为 `false`（即默认值），否则会抛出异常。

第三个参数 `useHarmonyMap` 如果为 `true`，则反序列化 Map 时，会以 ECMAScript 6 中的 Map 类型的实例对象返回反序列化值。否则返回 Object 对象。当反序列化的 Map 的键（key）为非字符串以外的其它类型（或多种混合类型）时，将该参数设置为 `true` 可以得到更准确的反序列化结果。默认值为 `false`。

如果反序列化过程中发生错误，则会抛出异常。

hprose 还提供了一个简化写法：

```
1. hprose.unserialize(stream[, simple[, useHarmonyMap]]);
```

该简化写法跟完整写法在功能上没有任何区别。

> 原文: *https://github.com/hprose/hprose-html5/wiki/Hprose-%E5%BA%8F%E5%88%97%E5%8C%96*

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：

```
inputFilter(data, context)
outputFilter(data, context)
```

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。

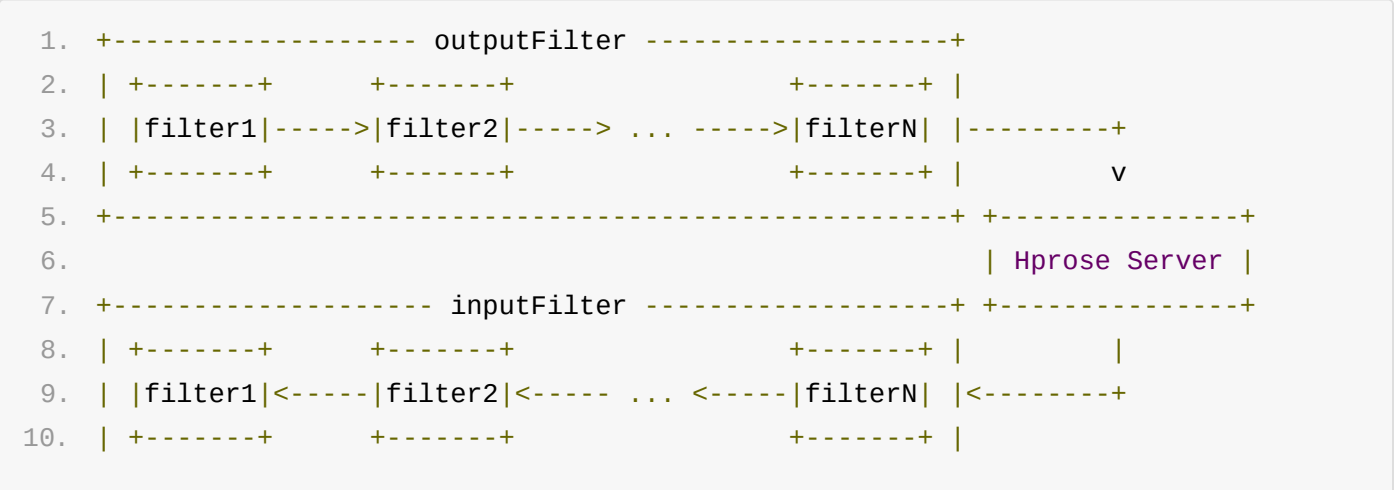
`data` 参数就是输入输出数据，它是 `Uint8Array` 类型的。这两个方法的返回值也是 `Uint8Array` 类型的数据，它表示已经处理过的数据，如果你不打算对数据进行修改，你可以直接将 `data` 参数作为返回值返回。

`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。

客户端的执行顺序



```
+-----------------------------------------------------+
```

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。

client.js

```
function log(data) {
    console.log(hprose.BytesIO.toString(data));
    return data;
}
var logfilter = {
    inputFilter: log,
    outputFilter: log
};
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.addFilter(logfilter);
client.hello("world", function(result) {
    console.log(result);
});
```

客户端输出

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

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。

client.js

```
function stat(data, context) {
    if ('starttime' in context.userdata) {
        var t = Date.now() - context.userdata.starttime;
        console.log('It takes ' + t + ' ms.');
    }
```

```
    else {
        context.userdata.starttime = Date.now();
    }
    return data;
}
var statfilter = {
    inputFilter: stat,
    outputFilter: stat
};
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.addFilter(statfilter);
client.hello('world', function(result) {
    console.log(result);
});
```

客户端输出

```
It takes 34 ms.
Hello world!
```

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 HTML5 中已经提供了一个现成的 JSONRPC 的过滤器。使用它，你可以将 Hprose 客户端变身为 JSONRPC 客户端。

JSONRPC 客户端

```
var client = hprose.Client.create("http://www.hprose.com/example/", ['hello']);
client.filter = new hprose.JSONRPCClientFilter();
client.hello("world", function(result) {
    console.log(result);
});
```

客户端只需要设置 `filter` 属性（或者用 addFilter 方法）为一个 `JSONRPCClientFilter` 实例对象，Hprose 客户端就马上变身为 JSONRPC 客户端了。不过需要注意一点，添加了 `JSONRPCClientFilter` 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。

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 Hprose

中间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-html5/wiki/Hprose-%E8%BF%87%E6%BB%A4%E5%99%A8*

Promise 异步编程

概述

JavaScript 层层回调的异步编程让人望而生畏。而 Promise 的诞生就是为了解决这个问题，它提供了一种 Future 模式，大大简化了异步编程的复杂性。而 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，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。

ECMAScript 6 提供了一套 `Promise` 的标准实现，目前大部分浏览器和较新版本的 Node.js 也都已经支持，但老旧的浏览器和旧版本的 Node.js 上是没有内置 `Promise` 实现的。另外，ECMAScript 6 这套 `Promise` 实现，所提供的 API 也是极其有限，只能满足基本需求。

为了能够有一套统一的 `Promise` 实现，并且不依赖第三方库，hprose 自己实现了一套完全兼容 Promise/A+ 规范的 API。

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，但并不完全兼容 Promise/A+ 规范。它最初是参照 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，又完全兼容 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。

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

```
var Future =  hprose.Future;
var 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

```
var Future =  hprose.Future;
var promise = new Future(function() { return 'hprose'; });
promise.then(function(value) {
    console.log(value);
});
```

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

```
var Future =  hprose.Future;
var promise = new Future(function() { throw 'hprose'; });
promise.catch(function(reason) {
    console.log(reason);
});
```

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

使用 Future 上的工厂方法

`Future` 上提供了 6 个工厂方法，它们分别是：

- value
- resolve
- error
- reject
- sync
- delayed

  其中 `value` 和 `resolve` 功能完全相同，`error` 和 `reject` 功能完全相同。`value` 和 `error` 这两个方法名来自 Dart 语言的 `Future` 类。而 `resolve` 和 `reject` 这两个方法名则来自 ECMAScript 6 的 Promise 对象。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。

创建一个成功（fulfilled）状态的 promise 对象

```
var Future =  hprose.Future;
var promise = Future.value('hprose'); // 换成 Future.resolve('hprose') 效果一样
promise.then(function(value) {
    console.log(value);
});
```

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

```
var Future =  hprose.Future;
var promise = Future.error('hprose'); // 换成 Future.reject('hprose') 效果一样
promise.catch(function(reason) {
    console.log(reason);
});
```

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。

同步创建一个 promise 对象

`Future` 上提供了一个：

```
Future.sync(computation)
```

方法可以让我们同步的创建一个 `promise` 对象。

这里“同步”的意思是指 `computation` 的执行是同步执行的。而通过 `Future` 构造器创建 `promise` 对象时，`computation` 是异步执行的。为了可以更好地理解这一点，我们来看一个具体的例子：

```
var Future =  hprose.Future;

function async() {
    console.log('before Future constructor');
    var promise = new Future(function() {
        console.log('running Future constructor');
        return 'promise from Future constructor';
    });
    promise.then(function(value) {
        console.log(value);
    });
    console.log('after Future constructor');
}
```

```
14.
15. function sync() {
16.     console.log('before Future.sync');
17.     var promise = Future.sync(function() {
18.         console.log('running Future.sync');
19.         return 'promise from Future.sync';
20.     });
21.     promise.then(function(value) {
22.         console.log(value);
23.     });
24.     console.log('after Future.sync');
25. }
26.
27. async();
28. sync();
```

这个程序的执行结果是：

```
1. before Future constructor
2. after Future constructor
3. before Future.sync
4. running Future.sync
5. after Future.sync
6. running Future constructor
7. promise from Future.sync
8. promise from Future constructor
```

从这里我们可以看出，`Future.sync` 方法中的 `computation` 确实是同步执行的，而 `Future` 构造器中的 `computation` 也确实是异步执行的。但是对于 `then` 中回调的执行，却都是异步的。

仔细观察结果，你也许会发现另一个有趣的区别，这里就不再细说了，留给读者自己研究。

另外，前面所说的 `Future.value`、`Future.resolve`、`Future.error`、`Future.reject` 这四个静态方法在创建 `promise` 对象时，跟使用 `Future` 构造器也存在同样的差别，但通常你可能不会注意到。

创建一个延迟 promise 对象

虽然通过 `Future` 构造器来创建一个 `promise` 对象跟使用 `Future.sync` 方法来比是异步的，但只是在执行顺序上能看出差别来，但是它并不会让你感到有明显的延时。如果你需要创建一个 `promise` 对象并且延迟一段时间后再执行

`computation` 函数，那么你可以使用

```
Future.delayed(duration, value)
```

这个方法。

`delayed` 方法的第一个参数 `duration` 是一个毫秒值，第二个参数 `value` 既可以是一个 `computation` 函数，也可以是一个其它类型的值（包括 `promise` 对象）。当 `value` 不是函数时，相当于传入了一个：

```
function() { return value; };
```

这样的 `computation` 函数。

这个 `computation` 函数会在延迟 `duration` 毫秒后执行，并将结果或失败原因充填入 `promise` 对象。

我们来看下面这个例子：

```
var Future =  hprose.Future;

function normal() {
    console.log(Date.now() + ': before Future constructor');
    var promise = new Future(function() {
        console.log(Date.now() + ': running Future constructor');
        return "promise from Future constructor";
    });
    promise.then(function(value) {
        console.log(Date.now() + ': ' + value);
    });
    console.log(Date.now() + ': after Future constructor');
}

function delayed() {
    console.log(Date.now() + ': before Future.delayed');
    var promise = Future.delayed(300, function() {
        console.log(Date.now() + ': running Future.delayed');
        return "promise from Future.delayed";
    });
    promise.then(function(value) {
        console.log(Date.now() + ': ' + value);
    });
    console.log(Date.now() + ': after Future.delayed');
}

normal();
delayed();
```

该程序的执行结果是：

```
1437889453869: before Future constructor
1437889453871: after Future constructor
1437889453872: before Future.delayed
1437889453872: after Future.delayed
1437889453873: running Future constructor
1437889453873: promise from Future constructor
1437889454173: running Future.delayed
1437889454173: promise from Future.delayed
```

这个结果一目了然，就不需要多做解释了。

通过 Completer 来创建 promise 对象

```
var Completer =  hprose.Completer;

var completer = new Completer();
var promise = completer.future;
promise.then(function(value) {
    console.log(value);
});
console.log('isComplete: ' + completer.isCompleted);
completer.complete('hprose')
console.log('isComplete: ' + completer.isCompleted);
```

运行结果：

```
isComplete: false
isComplete: true
hprose
```

`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，然后这个 `completer` 对象上的 `future` 属性就是一个 `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` 属性，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。

通过 ECMAScript 6 方式来创建 promise 对象

hprose 提供了 ECMAScript 6 的 `Promise` 对象的兼容实现。

具体使用方式可以直接参见该文档：MDN: Promise

这里就不再重复了。

hprose-html5.js 里面已经自动创建了全局的 `Promise` 实现，如果检测已有内置实现或者其它第三方实现的话，将不会替换成 hprose 版本的。

使用该方式创建的 `promise` 对象，你应该只使用 ECMAScript 6 文档中记载的 API，而不能使用本文档中的大部分 API。因此，在使用 hprose 时，并不推荐使用此方式创建 `promise` 对象。

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` 对象可能会是内置或第三方的 `Promise` 实例对象。

因此，推荐使用该方法来代替 ECMAScript 6 的 `Promise` 构造器方式。

通过 hprose 上的方法来创建 promise 对象

Promises/A+ Compliance Test Suite 上提供了一套用于测试是否符合 Promises/A+ 规范的最小适配器接口。hprose 对象上已经实现了这套接口，即：

- hprose.resolved(value)
- hprose.rejected(reason)
- hprose.deferred()
  - promise
  - resolve(value)
  - reject(reason)

  当然，这套接口不仅仅可以用来测试。你也可以用于实际用途。

其中，`hprose.resolved` 方法跟 `hprose.Future.value`、`hprose.Future.resolve` 功能相同。`hprose.rejected` 方法跟 `hprose.Future.error`、`hprose.Future.reject` 功能相同。

而 `hprose.deferred()` 跟 `new Completer()` 的作用类似。其中 `promise` 属性跟 `completer` 的 `future` 属性作用相同。`resolve` 方法跟 `completer` 的

`complete` 方法作用相同。`reject` 方法跟 `completer` 的 `completeError` 方法作用相同。

这里也不再重复举例。

Future.prototype 上的基本方法

then 方法

`then` 方法是 `Promise` 的核心和精髓所在。它有两个参数：`onFulfilled`，`onRejected`。这两个参数皆为 `function` 类型。当它们不是 `function` 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，`onFulfilled` 函数会被回调，参数值为成功值。当 `promise` 对象状态为失败（rejected）时，`onRejected` 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 `onFulfilled` 或 `onRejected` 的返回值或抛出的异常来决定。如果 `onFulfilled` 或 `onRejected` 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 `onFulfilled` 或 `onRejected` 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。

`then` 方法的 `onFulfilled`，`onRejected` 这两个回调函数是异步执行的，即使当前的 `promise` 对象的状态为已完成（fulfilled 或 rejected）。

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 `onFulfilled`，`onRejected` 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，`onFulfilled` 函数会被调用，当 `promise2` 的最终展开值为失败原因时，`onRejected` 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 `onRejected` 回调函数。

`then` 方法是 Promise/A+ 规范的完整实现。

具体使用方法可参见：MDN: Promise.prototype.then()

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。

catch 方法

该方法是 `then(null, onRejected)` 的简化写法。

具体使用方法可参见：MDN: Promise.prototype.catch()

fail 方法

该方法是 `done(null, onRejected)` 的简化方法。

catchError 方法

该方法是 `catch` 的增强版，它具有两个参数，第一个参数 `onRejected` 跟 `catch` 方法相同，第二个参数是一个测试函数。当该测试函数省略时，它的效果跟 `catch` 方法相同。例如：

```
var Future = hprose.Future;

var p = Future.reject(new TypeError('typeError'));

p
 .catchError(function(reason) { return 'this is a syntax error'; },
             function(reason) { return reason instanceof SyntaxError; })
 .catchError(function(reason) { return 'this is a type error'; },
             function(reason) { return reason instanceof TypeError; })
 .then(function(value) { console.log(value);  });
```

输出结果为：

```
this is a type error
```

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。

该方法已绑定到它所在的 `promise` 对象，因此可以安全的作为回调函数进行传递。

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。

该方法已绑定到它所在的 `promise` 对象，因此可以安全的作为回调函数进行传递。

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. { state: 'pending' }
```

如果当前状态为成功（fulfilled），返回值为：

```
1. { state: 'fulfilled', value: value };
```

如果当前状态为失败（rejected），返回值为：

```
1. { state: 'rejected', reason: 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. var Future = hprose.Future;
2.
3. var p1 = Future.resolve('resolve hprose');
```

```
4.
5.  p1.whenComplete(function() { console.log('p1 complete'); })
6.    .then(function(value) { console.log(value);  });
7.
8.  var p2 = Future.reject('reject thrift');
9.
10. p2.whenComplete(function() { console.log('p2 complete'); })
11.   .catch(function(reason) { console.log(reason);  });
12.
13. var p3 = Future.resolve('resolve protobuf');
14.
15. p3.whenComplete(function() { console.log('p3 complete');
16.                              throw 'reject protobuf'; })
17.   .catch(function(reason) { console.log(reason);  });
```

运行结果如下：

```
1.  p1 complete
2.  p2 complete
3.  p3 complete
4.  resolve hprose
5.  reject thrift
6.  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。

该方法的最新版本支持不带参数调用，当不带参数调用时，返回一个新的 promise 对象，该对象会将源 promise 对象的失败（rejected）值转换为成功（fulfilled）值，这样在后面可以直接使用 `then` 的第一个回调参数统一处理。它的主要作用是当配合协程一起使用时，可以避免使用 `try` `catch` 来捕获异常。

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。

fill 方法

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

Future 上的辅助方法

isFuture 方法

```
1. Future.isFuture(obj)
```

用来判断是否是 `Future` 的实例对象。

isPromise 方法

```
1. Future.isPromise(obj)
```

用来判断是否是 `Future` 或 ECMAScript 6 的 `Promise` 实例对象。

注意，该方法对其它符合 Promise/A+ 规范实现的 `thenable` 对象进行判断的返回值是 `false`。

如果你需要一个 `promise` 对象，保险的做法是用 `Future.value()` 方法包装一下。

toPromise 方法

```
1. Future.toPromise(obj)
```

如果 `obj` 是一个 `Promise` 对象，那么直接返回 `obj`，否则返回 `Future.value(obj)`。

all 方法

```
1. Future.all(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。

具体使用方法可参见：MDN: Promise.all()

`Future.all` 方法与 `Promise.all` 方法在参数上有一点区别，`Future.all` 方法的数组参数本身也可以是一个值为数组的 `promise` 对象。

race 方法

```
1. Future.race(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）。

具体使用方法可参见：MDN: Promise.race()

`Future.race` 方法与 `Promise.race` 方法在参数上有一点区别，`Future.race` 方法的数组参数本身也可以是一个值为数组的 `promise` 对象。

join 方法

```
1. Future.join([arg1[, arg2[, arg3...]]]);
```

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. var Future = hprose.Future;
2.
3. var promise = Future.resolve(3);
4.
5. Future.all([true, promise])
6.       .then(function(values) {
7.           console.log(values);
8.        });
9.
10. Future.join(true, promise)
11.       .then(function(values) {
12.           console.log(values);
13.        });
```

输出结果如下：

```
1. [ true, 3 ]
2. [ true, 3 ]
```

any 方法

```
1.  Future.any(array)
```

该方法是 `race` 方法的改进版。

对于 `race` 方法，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` 方法，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 `RangeError` 对象。

对于 `race` 方法，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` 方法，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。

settle 方法

```
1. Future.settle(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（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 `inspect` 方法返回值，其数组元素与原数组元素一一对应。

例如：

```
1. var Future = hprose.Future;
2.
3. var p1 = Future.resolve(3);
4. var p2 = Future.reject("x");
5.
6. Future.settle([true, p1, p2])
7.       .then(function(values) {
8.           console.log(values);
9.        });
```

输出结果为：

```
1. [ { state: 'fulfilled', value: true },
2.   { state: 'fulfilled', value: 3 },
3.   { state: 'rejected', reason: 'x' } ]
```

attempt 方法

```
1. Future.attempt(handler[, arg1[, arg2[, arg3...]]]);
```

`attempt` 方法的作用是异步执行 `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`，`arg3` ….。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，`attempt` 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 `attempt` 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 `attempt` 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 `attempt` 返回的 `promise` 对象的成功值。

```
var Future = hprose.Future;

function add(a, b) {
    return a + b;
}

var p1 = Future.resolve(3);

Future.attempt(add, 2, p1)
      .then(function(value) {
          console.log(value);
      });
```

输出结果为：

```
5
```

run 方法

```
Future.run(handler[, thisArg[, arg1[, arg2[, arg3...]]]]);
```

`run` 方法跟上面的 `attempt` 方法功能类似，唯一的区别是 `run` 方法的第二个参数为 `thisArg`，他表示 `handler` 的执行上下文。当 `thisArg` 的值为 `undefine` 时，行为跟 `attempt` 方法完全一致。

例如：

```
var Future = hprose.Future;

function add(a, b) {
    return a + b;
}

var p1 = Future.resolve(3);

Future.run(console.log, console, Future.attempt(add, 2, p1));
```

输出结果为：

```
5
```

wrap 方法

```
Future.wrap(handler[, thisArg]);
```

`wrap` 方法返回一个包装好的函数，该函数的执行方式跟使用 `Future.run` 的效果一样。例如：

```
var Future = hprose.Future;

var add = Future.wrap(function(a, b) {
    return a + b;
});

var log = Future.wrap(console.log, console);

var p1 = Future.resolve(3);

log(add(2, p1));
```

输出结果为：

```
5
```

forEach 方法

```
Future.forEach(array, callback[, thisArg])
```

该方法你可以认为是 `Array.forEach` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

```
var Future = hprose.Future;

```

```
function logArrayElements(element, index, array) {
  console.log('a[' + index + '] = ' + element);
}

// Note elision, there is no member at 2 so it isn't visited
Future.forEach([2, Future.value(5), , 9], logArrayElements);
```

输出结果为：

```
a[0] = 2
a[1] = 5
a[3] = 9
```

every 方法

```
Future.every(array, callback[, thisArg])
```

该方法你可以认为是 `Array.every` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isBigEnough(element, index, array) {
  return element >= 10;
}

var a1 = [12, Future.value(5), 8, Future.value(130), 44];
var a2 = [12, Future.value(54), 18, Future.value(130), 44];
var a3 = Future.value(a1);
var a4 = Future.value(a2);
log(Future.every(a1, isBigEnough));   // false
log(Future.every(a2, isBigEnough));   // true
log(Future.every(a3, isBigEnough));   // false
log(Future.every(a4, isBigEnough));   // true
```

输出结果为：

```
false
true
false
true
```

some 方法

```
Future.some(array, callback[, thisArg])
```

该方法你可以认为是 `Array.some` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isBiggerThan10(element, index, array) {
  return element > 10;
}

var a1 = [2, Future.value(5), 8, Future.value(1), 4];
var a2 = [12, Future.value(5), 8, Future.value(1), 4];
var a3 = Future.value(a1);
var a4 = Future.value(a2);
log(Future.some(a1, isBiggerThan10));   // false
log(Future.some(a2, isBiggerThan10));   // true
log(Future.some(a3, isBiggerThan10));   // false
log(Future.some(a4, isBiggerThan10));   // true
```

输出结果为：

```
false
true
false
true
```

filter 方法

```
Future.filter(array, callback[, thisArg])
```

该方法你可以认为是 `Array.filter` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isBigEnough(value) {
  return value >= 10;
}

var a1 = [12, Future.value(5), 8, Future.value(130), 44];
var a2 = Future.value(a1);
log(Future.filter(a1, isBigEnough));
log(Future.filter(a2, isBigEnough));
```

输出结果为：

```
[ 12, 130, 44 ]
[ 12, 130, 44 ]
```

map 方法

```
Future.map(array, callback[, thisArg])
```

该方法你可以认为是 `Array.map` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var numbers = [1, Future.value(4), Future.value(9)];
log(Future.map(numbers, Math.sqrt));
log(Future.map(numbers, function(num) {
  return num * 2;
}));
```

输出结果为：

```
[ 1, 2, 3 ]
[ 2, 8, 18 ]
```

reduce 方法

```
Future.reduce(array, callback[, initialValue])
```

该方法你可以认为是 `Array.reduce` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

`initialValue` 的值也可以是一个 `promise` 对象。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var numbers = [Future.value(0), 1, Future.value(2), 3, Future.value(4)];

function callback(previousValue, currentValue, index, array) {
  return previousValue + currentValue;
}

log(Future.reduce(numbers, callback));
```

```
log(Future.reduce(numbers, callback, 10));
log(Future.reduce(numbers, callback, Future.value(20)));
```

输出结果为：

```
10
20
30
```

reduceRight 方法

```
Future.reduceRight(array, callback[, initialValue])
```

该方法你可以认为是 `Array.reduceRight` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

`initialValue` 的值也可以是一个 `promise` 对象。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function concat(a, b) {
    return a.concat(b);
}

var array = [[0, 1], Future.value([2, 3]), Future.value([4, 5])];

log(Future.reduceRight(array, concat, []));
log(Future.reduceRight(array, concat, Future.value([6, 7])));
```

输出结果为：

```
[ 4, 5, 2, 3, 0, 1 ]
[ 6, 7, 4, 5, 2, 3, 0, 1 ]
```

indexOf 方法

```
Future.indexOf(array, searchElement[, fromIndex])
```

该方法你可以认为是 `Array.indexOf` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

`searchElement` 的值也可以是一个 `promise` 对象。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var array = [1, Future.value(2), Future.value(3)];

log(Future.indexOf(array, 2));
log(Future.indexOf(array, Future.value(3), 1));
log(Future.indexOf(array, 1, 1));
```

输出结果为：

```
1
2
-1
```

lastIndexOf 方法

```
Future.lastIndexOf(array, searchElement[, fromIndex])
```

该方法你可以认为是 `Array.lastIndexOf` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

`searchElement` 的值也可以是一个 `promise` 对象。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var array = [1, Future.value(2), Future.value(3)];

log(Future.lastIndexOf(array, 2));
log(Future.lastIndexOf(array, Future.value(3), 1));
log(Future.lastIndexOf(array, 1, 1));
```

输出结果为：

```
1
-1
0
```

includes 方法

```
Future.includes(array, searchElement[, fromIndex])
```

该方法你可以认为是 `Array.includes` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。

`searchElement` 的值也可以是一个 `promise` 对象。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var array = [1, Future.value(2), Future.value(3)];

log(Future.includes(array, 2));
log(Future.includes(array, Future.value(3), 1));
log(Future.includes(array, 1, 1));
```

输出结果为：

```
true
```

```
true
false
```

find 方法

```
Future.find(array, predicate[, thisArg])
```

该方法你可以认为是 `Array.find` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。`predicate` 是一个回调方法。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isPrime(element, index, array) {
  var start = 2;
  while (start <= Math.sqrt(element)) {
    if (element % start++ < 1) {
      return false;
    }
  }
  return element > 1;
}

var array1 = [4, Future.value(6), 8, Future.value(12)];
var array2 = [4, Future.value(5), 7, Future.value(12)];

log(Future.find(array1, isPrime));
log(Future.find(array2, isPrime));
```

输出结果为：

```
undefined
5
```

findIndex 方法

```
Future.findIndex(array, predicate[, thisArg])
```

该方法你可以认为是 `Array.findIndex` 的 `promise` 版本，其中参数 `array` 可以是一个包含了 `promise` 元素的数组，也可以是一个包含了数组的 `promise` 对象。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。 `predicate` 是一个回调方法。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isPrime(element, index, array) {
  var start = 2;
  while (start <= Math.sqrt(element)) {
    if (element % start++ < 1) {
      return false;
    }
  }
  return element > 1;
}

var array1 = [4, Future.value(6), 8, Future.value(12)];
var array2 = [4, Future.value(5), 7, Future.value(12)];

log(Future.findIndex(array1, isPrime));
log(Future.findIndex(array2, isPrime));
```

输出结果为：

```
-1
1
```

Future.prototype 上的辅助方法

timeout 方法

```
Future.prototype.timeout(duration[, reason])
```

创建一个新的 `promise` 对象，当超过设定的时间 `duration`（单位毫秒），源 `promise` 对象如果还未被设置为成功（fulfilled）或失败（rejected），则

新的 `promise` 对象被设置为一个 `TimeoutError` 或者自定义的 `reason`。否则，其值跟源 `promise` 相同。

```
var Future = hprose.Future;

var add = Future.wrap(function(a, b) {
    return a + b;
});

var log = Future.wrap(console.log, console);

var p1 = Future.delayed(200, 3).timeout(300);
var p2 = Future.delayed(500, 3).timeout(300);

log(add(p1, 2)).catch(log);
log(add(p2, 5)).catch(log);
```

输出结果为：

```
5
{ [TimeoutError: timeout] message: 'timeout', name: 'TimeoutError' }
```

delay 方法

```
Future.prototype.delay(duration)
```

创建一个新的 `promise` 对象，当经过 `duration` 毫秒后，该 `promise` 对象的值将被设置为跟源 `promise` 对象相同的成功值。如果源 `promise` 对象被设置为失败（rejected）状态，新的 `promise` 对象将立即被设为相同的失败原因，而无等待。

```
var Future = hprose.Future;

var add = Future.wrap(function(a, b) {
    return a + b;
});

var log = Future.wrap(console.log, console);

var p = Future.value(3);
var p1 = p.delay(200).timeout(300);
var p2 = p.delay(500).timeout(300);

log(add(p1, 2)).catch(log);
```

```
14. log(add(p2, 5)).catch(log);
```

输出结果为：

```
1. 5
2. { [TimeoutError: timeout] message: 'timeout', name: 'TimeoutError' }
```

tap 方法

```
1. Future.prototype.tap(onfulfilledSideEffect[, thisArg])
```

以下两种写法是等价的：

```
1. promise.then(function(result) {
2.     onfulfilledSideEffect.call(thisArg, result);
3.     return result;
4. });
5.
6. promise.tap(onfulfilledSideEffect, thisArg);
```

spread 方法

```
1. Future.prototype.spread(onfulfilledArray[, thisArg])
```

以下两种写法是等价的：

```
1. promise.then(function(array) {
2.     return onfulfilledArray.apply(thisArg, array);
3. });
4.
5. promise.spread(onfulfilledArray, thisArg);
```

get 方法

```
1. Future.prototype.get(key)
```

以下两种写法是等价的：

```
1. promise.then(function(result) {
```

```
    return result[key];
});

promise.get(key);
```

set 方法

```
Future.prototype.set(key, value)
```

以下两种写法是等价的：

```
promise.then(function(result) {
    result[key] = value;
    return result;
});

promise.set(key, value);
```

下面我们来看一个例子：

```
var Future = hprose.Future;

function User() {
    this.name = "Tom";
    this.age = 18;
}

var log = Future.wrap(console.log, console);

var p = Future.value(new User());

p.get('name').then(function(result) { console.log(result); });
log(p.get('age'));
p.set('password', 'hprose')
 .tap(log)
 .tap(function(result) { console.log(result); })
 .set('password', 'I love hprose!')
 .get('password')
 .then(log);
```

输出结果：

```
Tom
{ name: 'Tom', age: 18, password: 'hprose' }
18
{ name: 'Tom', age: 18, password: 'I love hprose!' }
```

```
5. I love hprose!
```

注意上面的结果中，被 `Future.wrap` 包装过的 `log` 函数在执行时，并不是按照代码书写顺序执行的，也不是按照链式调用顺序执行的，它比在代码中书写的位置执行的要晚，因为被包装过的函数本身也是异步执行的，因此在输出结果上会比较反直觉。

比如上面输出结果中的这一句：

```
1. { name: 'Tom', age: 18, password: 'I love hprose!' }
```

实际上是这一句代码：

```
1.   .tap(log)
```

输出的。

因此，在链式调用中使用 `Future.wrap` 包装过的函数，一定要注意这点。后面介绍的 `bind` 方法也同样需要注意这个问题。

apply 方法

```
1. Future.prototype.apply(method[, args])
```

异步执行当前 `promise` 对象上的方法名为 `method` 的方法，参数为 `args` 数组中的元素，返回结果为包含了执行结果的 `promise` 对象。其中参数数组中的元素也可以为 `promise` 对象，在执行时，会带入这些 `promise` 对象所包含的成功值，如果这些 `promise` 对象中包含有失败状态的，则 `method` 方法不会执行，返回的 `promise` 对象中包含的失败原因为第一个变为失败（rejected）状态的参数的失败原因。

例如：

```
1. var Future = hprose.Future;
2.
3. var log = Future.wrap(console.log, console);
4.
5. var p = Future.value([]);
6.
7. p.apply('push', ['banana', Future.value('apple'), 'peach'])
8.  .then(function(length) {
```

```
    console.log(length);
    log(p);
 });

p.apply('push', ['banana', Future.error('apple'), 'peach'])
 .catch(function(reason) {
    console.log(reason);
 });
```

输出结果为：

```
apple
3
[ 'banana', 'apple', 'peach' ]
```

因为第二个 `apply` 的参数中包含有失败（rejected）状态的 `promise` 参数，因此第二个 `apply` 中的 `push` 方法并没有执行而提前返回，所以失败原因 `apple` 被先显示出来。

call 方法

```
Future.prototype.call(method[, arg1[, arg2[, arg3...]]])
```

与上面的 `apply` 方法类似，唯一不同的是参数。看下面的例子：

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var p = Future.value([]);

p.call('push', 'banana', Future.value('apple'), 'peach')
 .then(function(length) {
    console.log(length);
    log(p);
 });

p.call('push', 'banana', Future.error('apple'), 'peach')
 .catch(function(reason) {
    console.log(reason);
 });
```

执行结果与上面的 `apply` 的例子执行结果相同。

bind 方法

```
Future.prototype.bind(method[, arg1[, arg2[, arg3...]]])
Future.prototype.bind(methods[, arg1[, arg2[, arg3...]]])
```

`bind` 方法可以将当前 `promise` 对象所包含的成功值上的一个或多个方法及其参数绑定到当前的 `promise` 对象上。其中参数可以是 `promise` 对象，且绑定之后的方法在调用时也支持 `promise` 对象参数，返回值也为 `promise` 对象。例如：

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var p = Future.value([]);
p.bind('push');

p.push('banana', Future.value('apple'), 'peach')
 .then(function(length) {
    console.log(length);
    log(p);
 });

p.push('banana', Future.error('apple'), 'peach')
 .catch(function(reason) {
    console.log(reason);
 });

var p2 = Future.value(new Date());
p2.bind(Object.getOwnPropertyNames(Date.prototype));

log(p2.toString());
p2.setFullYear(1980).then(function() {
    log(p2.toString());
});
```

运行结果如下：

```
apple
3
Tue Jul 28 2015 20:51:47 GMT+0800 (CST)
[ 'banana', 'apple', 'peach' ]
Mon Jul 28 1980 20:51:47 GMT+0800 (CST)
```

forEach 方法

```
Future.prototype.forEach(callback[, thisArg])
```

该方法是 `Array.prototype.forEach` 的 `promise` 版本。它与 `Array.prototype.forEach` 的区别跟 `Future.forEach` 和 `Array.forEach` 的区别相同。

```
var Future = hprose.Future;

function logArrayElements(element, index, array) {
  console.log('a[' + index + '] = ' + element);
}

var p = Future.value([2, Future.value(5), , 9]);
// Note elision, there is no member at 2 so it isn't visited
p.forEach(logArrayElements);
```

输出结果为：

```
a[0] = 2
a[1] = 5
a[3] = 9
```

every 方法

```
Future.prototype.every(callback[, thisArg])
```

该方法是 `Array.prototype.every` 的 `promise` 版本。它与 `Array.prototype.every` 的区别跟 `Future.every` 和 `Array.every` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isBigEnough(element, index, array) {
  return element >= 10;
}

var a1 = Future.value([12, Future.value(5), 8, Future.value(130), 44]);
var a2 = Future.value([12, Future.value(54), 18, Future.value(130), 44]);
```

```
var a3 = Future.value(a1);
var a4 = Future.value(a2);
log(a1.every(isBigEnough));   // false
log(a2.every(isBigEnough));   // true
log(a3.every(isBigEnough));   // false
log(a4.every(isBigEnough));   // true
```

输出结果为：

```
false
true
false
true
```

some 方法

```
Future.prototype.some(callback[, thisArg])
```

该方法是 `Array.prototype.some` 的 `promise` 版本。它与 `Array.prototype.some` 的区别跟 `Future.some` 和 `Array.some` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isBiggerThan10(element, index, array) {
  return element > 10;
}

var a1 = Future.value([2, Future.value(5), 8, Future.value(1), 4]);
var a2 = Future.value([12, Future.value(5), 8, Future.value(1), 4]);
var a3 = Future.value(a1);
var a4 = Future.value(a2);
log(a1.some(isBiggerThan10));   // false
log(a2.some(isBiggerThan10));   // true
log(a3.some(isBiggerThan10));   // false
log(a4.some(isBiggerThan10));   // true
```

输出结果为：

```
false
true
```

```
3. false
4. true
```

filter 方法

```
1. Future.prototype.filter(callback[, thisArg])
```

该方法是 `Array.prototype.filter` 的 `promise` 版本。它与 `Array.prototype.filter` 的区别跟 `Future.filter` 和 `Array.filter` 的区别相同。

```
1. var Future = hprose.Future;
2.
3. var log = Future.wrap(console.log, console);
4.
5. function isBigEnough(value) {
6.   return value >= 10;
7. }
8.
9. var p = Future.value([12, Future.value(5), 8, Future.value(130), 44]);
10. log(p.filter(isBigEnough));
```

输出结果为：

```
1. [ 12, 130, 44 ]
```

map 方法

```
1. Future.prototype.map(callback[, thisArg])
```

该方法是 `Array.prototype.map` 的 `promise` 版本。它与 `Array.prototype.map` 的区别跟 `Future.map` 和 `Array.map` 的区别相同。

```
1. var Future = hprose.Future;
2.
3. var log = Future.wrap(console.log, console);
4.
5. var p = Future.value([1, Future.value(4), Future.value(9)]);
6. log(p.map(Math.sqrt));
7. log(p.map(function(num) {
```

```
  return num * 2;
}));
```

输出结果为：

```
[ 1, 2, 3 ]
[ 2, 8, 18 ]
```

reduce 方法

```
Future.prototype.reduce(callback[, initialValue])
```

该方法是 `Array.prototype.reduce` 的 `promise` 版本。它与 `Array.prototype.reduce` 的区别跟 `Future.reduce` 和 `Array.reduce` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var p = Future.value([Future.value(0), 1, Future.value(2), 3,
Future.value(4)]);

function callback(previousValue, currentValue, index, array) {
  return previousValue + currentValue;
}

log(p.reduce(callback));
log(p.reduce(callback, 10));
log(p.reduce(callback, Future.value(20)));
```

输出结果为：

```
10
20
30
```

reduceRight 方法

```
Future.prototype.reduceRight(callback[, initialValue])
```

该方法是 `Array.prototype.reduceRight` 的 `promise` 版本。它与 `Array.prototype.reduceRight` 的区别跟 `Future.reduceRight` 和 `Array.reduceRight` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function concat(a, b) {
    return a.concat(b);
}

var p = Future.value([[0, 1], Future.value([2, 3]), Future.value([4, 5])]);

log(p.reduceRight(concat, []));
log(p.reduceRight(concat, Future.value([6, 7])));
```

输出结果为：

```
[ 4, 5, 2, 3, 0, 1 ]
[ 6, 7, 4, 5, 2, 3, 0, 1 ]
```

indexOf 方法

```
Future.prototype.indexOf(searchElement[, fromIndex])
```

该方法是 `Array.prototype.indexOf` 的 `promise` 版本。它与 `Array.prototype.indexOf` 的区别跟 `Future.indexOf` 和 `Array.indexOf` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var array = Future.value([1, Future.value(2), Future.value(3)]);

log(array.indexOf(2));
log(array.indexOf(Future.value(3), 1));
log(array.indexOf(1, 1));
```

输出结果为：

```
1
2
-1
```

lastIndexOf 方法

```
Future.prototype.lastIndexOf(searchElement[, fromIndex])
```

该方法是 `Array.prototype.lastIndexOf` 的 `promise` 版本。它与 `Array.prototype.lastIndexOf` 的区别跟 `Future.lastIndexOf` 和 `Array.lastIndexOf` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

var array = Future.value([1, Future.value(2), Future.value(3)]);

log(array.lastIndexOf(2));
log(array.lastIndexOf(Future.value(3), 1));
log(array.lastIndexOf(1, 1));
```

输出结果为：

```
1
-1
0
```

includes 方法

```
Future.prototype.includes(searchElement[, fromIndex])
```

该方法是 `Array.prototype.includes` 的 `promise` 版本。它与 `Array.prototype.includes` 的区别跟 `Future.includes` 和 `Array.includes` 的区别相同。

```
var Future = hprose.Future;
```

```
2.
3.  var log = Future.wrap(console.log, console);
4.
5.  var array = Future.value([1, Future.value(2), Future.value(3)]);
6.
7.  log(array.includes(2));
8.  log(array.includes(Future.value(3), 1));
9.  log(array.includes(1, 1));
```

输出结果为:

```
1.  true
2.  true
3.  false
```

find 方法

```
1.  Future.prototype.find(predicate[, thisArg])
```

该方法是 `Array.prototype.find` 的 `promise` 版本。它与 `Array.prototype.find` 的区别跟 `Future.find` 和 `Array.find` 的区别相同。

```
1.  var Future = hprose.Future;
2.
3.  var log = Future.wrap(console.log, console);
4.
5.  function isPrime(element, index, array) {
6.    var start = 2;
7.    while (start <= Math.sqrt(element)) {
8.      if (element % start++ < 1) {
9.        return false;
10.     }
11.   }
12.   return element > 1;
13. }
14.
15. var array1 = Future.value([4, Future.value(6), 8, Future.value(12)]);
16. var array2 = Future.value([4, Future.value(5), 7, Future.value(12)]);
17.
18. log(array1.find(isPrime));
19. log(array2.find(isPrime));
```

输出结果为:

```
undefined
5
```

findIndex 方法

```
Future.prototype.findIndex(predicate[, thisArg])
```

该方法是 `Array.prototype.findIndex` 的 `promise` 版本。它与 `Array.prototype.findIndex` 的区别跟 `Future.findIndex` 和 `Array.findIndex` 的区别相同。

```
var Future = hprose.Future;

var log = Future.wrap(console.log, console);

function isPrime(element, index, array) {
  var start = 2;
  while (start <= Math.sqrt(element)) {
    if (element % start++ < 1) {
      return false;
    }
  }
  return element > 1;
}

var array1 = Future.value([4, Future.value(6), 8, Future.value(12)]);
var array2 = Future.value([4, Future.value(5), 7, Future.value(12)]);

log(array1.findIndex(isPrime));
log(array2.findIndex(isPrime));
```

输出结果为：

```
-1
1
```

> 原文: *https://github.com/hprose/hprose-html5/wiki/Promise-%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B*

协程

基本用法

ES6 中引入了 `Generator`，`Generator` 通过封装之后，可以作为协程来进行使用。

其中对 `Generator` 封装最为著名的当属 `tj/co`，但是 `tj/co` 跟 ES2016 的 async/await 相比的话，还存在一些比较严重的缺陷。

hprose 中也引入了对 `Generator` 封装的协程支持，而且比 `tj/co` 更加完善，后面我们会详细介绍它们之间的一些差别。

下面的例子都是在 nodejs 的运行环境中执行的，但是它们对其它执行环境同样适用（除了脚本的引入方式）。

下面先让我们来看一个例子：

```
1. var hprose = require('hprose');
2.
3. hprose.co(function*() {
4.     var client = hprose.Client.create('http://hprose.com/example/');
5.     yield client.useService();
6.     console.log(yield client.hello("World"));
7. });
```

`hprose.co`（也可以是 `hprose.Future.co`）就是一个协程封装函数。它的功能是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，`client` 是一个 Hprose 的 HTTP 客户端。Hprose 的 JavaScript 版本（包括 NodeJS, HTML5, JavaScript 和微信小程序专用版）的客户端为异步客户端，所以它上面的调用都是异步调用。

因为 JavaScript 的 `Proxy`（中文版）具有浏览器兼容性问题，所以在客户端代理对象的生成上，使用的是动态获取服务列表的方式来实现的。

上面的 `yield client.useService()` 语句就是用于返回这个客户端代理对象，使用默认参数调用时，会将 `client` 对象本身设置为代理对象，因此这里我们没有使用返回值，如果要在其它地方使用的话，最好是保存它的返回值。

`client.useService()` 的返回值是一个 `Promise` 对象，如果不使用协程，那么在使用时，需要使用 `then` 方法，然后在其回调中才能使用，而使用协程，直接使用 `yield` 就可以获得实际的代理对象了。而且因为在这里 `client` 本身就是代理对象，因此当 `yield` 返回代理对象之后，`client` 对象就已经被初始化好了，因此后面就可以直接调用 `client` 上的 `hello` 方法了。

`client` 的 `hello` 方法的返回值也是个 `Promise` 对象，使用 `yield` 之后，它的返回值就变成了实际值，也就可以直接用 `console.log` 进行打印了。

通过上面的例子，我们可以看出，使用协程方式，Hprose 调用就被完全同步化了。这可以大大简化异步程序的编写。

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。

协程兼容性问题

因为 `Generator` 是在 ES6 中引入了，所以比较老版本的 NodeJS 是不支持的，而浏览器的支持就更少了，目前只有 Chrome 和 Firefox 支持，而 IE、Opera、Safari 都不支持，HyBird App 也不支持。

那是否意味着这个功能很鸡肋呢？并不是，因为现在有许多工具可以将 ES6 代码转换为 ES5 代码，比如 Babel。其中就包括对 `Generator` 的支持。所以，即使你使用了协程，仍然可以通过这些转换器转换为在各种浏览器中都可以运行的程序。

为了方便用户使用，在 hprose 中还直接集成了 regenerator-runtime.js，不需要额外引入这个文件了。

微信小程序因为缺少全局对象，仍然需要使用：

```
1. const regeneratorRuntime = require("regenerator-runtime.js");
```

的方式来单独引入该文件，不过该文件也已经放在微信小程序专用版中了，免去了用户单独寻找该文件的麻烦。

与 tj/co 库的区别

`tj/co` 有以下几个方面的问题：

首先，`tj/co` 库中的 `yield` 只支持 thunk 函数，生成器函数，promise 对象，以及数组和对象，但是不支持普通的基本类型的数据，比如 `null`，数字，字符串等都不支持。这对于 `yield` 一个类型不确定的变量来说，是很不方便的。而且这跟 `await` 也是不兼容的。

其次，在 `yield` 数组和对象时，`tj/co` 库会自动对数组中的元素和对象中的字段递归的遍历，将其中的所有的 `Promise` 元素和字段替换为实际值，这对于简单的数据来说，会方便一些。但是对于带有循环引用的数组和对象来说，会导致无法获取到结果，这是一个致命的问题。即使对于不带有循环引用结构的数组和对

象来说，如果该数组和对象比较复杂，这也会消耗大量的时间。而且这跟 `await` 也是不兼容的。

再次，对于 thunk 函数，`tj/co` 库会认为回调函数第一个参数必须是表示错误，从第二个参数开始才表示返回值。而这对于回调函数只有一个返回值参数的函数，或者回调函数的第一个参数不表示错误的函数来说，`tj/co` 库就无法使用了。

而 `hprose.co` 对 `yield` 的支持则跟 `await` 完全兼容，支持对所有类型的数据进行 `yield`。

当 `hprose.co` 对 chunk 函数进行 `yield` 时，如果回调函数第一个参数是 `Error` 类型的对象才会被当做错误处理。如果回调函数只有一个参数且不是 `Error` 类型的对象，则作为返回值对待。如果回调函数有两个以上的参数，如果第一个参数为 `null` 或 `undefined`，则第一个参数被当做无错误被忽略，否则，全部回调参数都被当做返回值对待。如果被当做返回值的回调参数有多个，则这多个参数被当做数组结果对待，如果只有一个，则该参数被直接当做返回值对待。

下面我们来举例说明一下：

yield 基本类型

首先我们来看一下 `tj/co` 库的例子：

```
var co = require('co');

co(function*() {
    try {
        console.log(yield Promise.resolve("promise"));
        console.log(yield function *() { return "generator" });
        console.log(yield new Date());
        console.log(yield 123);
        console.log(yield 3.14);
        console.log(yield "hello");
        console.log(yield true);
    }
    catch (e) {
        console.error(e);
    }
});
```

该程序运行结果为：

```
promise
generator
```

```
3. TypeError: You may only yield a function, promise, generator, array, or object,
   but the following object was passed: "Sat Nov 19 2016 14:51:09 GMT+0800 (CST)"
4.     at next (/usr/local/lib/node_modules/co/index.js:101:25)
5.     at onFulfilled (/usr/local/lib/node_modules/co/index.js:69:7)
6.     at process._tickCallback (internal/process/next_tick.js:103:7)
7.     at Module.runMain (module.js:577:11)
8.     at run (bootstrap_node.js:352:7)
9.     at startup (bootstrap_node.js:144:9)
10.    at bootstrap_node.js:467:3
```

其实除了前两个，后面的几个基本类型的数据都不能被 `yield` 。如果我们把上面代码的第一句改为：

```
1. var co = require('hprose').co;
```

后面的代码都不需要修改，我们来看看运行结果：

```
1. promise
2. generator
3. 2016-11-19T06:54:30.081Z
4. 123
5. 3.14
6. hello
7. true
```

也就是说， `hprose.co` 支持对所有类型进行 `yield` 操作。下面我们再来看看 async/await 是什么效果：

```
1. (async function() {
2.     try {
3.         console.log(await Promise.resolve("promise"));
4.         console.log(await function *() { return "generator" });
5.         console.log(await new Date());
6.         console.log(await 123);
7.         console.log(await 3.14);
8.         console.log(await "hello");
9.         console.log(await true);
10.    }
11.    catch (e) {
12.        console.error(e);
13.    }
14. })();
```

上面的代码基本上就是把 `co(function*…)` 替换成了 `async function…`，把 `yield` 替换成了 `await`。

我们来运行上面的程序，注意，对于当前版本的 node 运行时需要加上 `--harmony_async_await` 参数，运行结果如下：

```
promise
[Function]
2016-11-19T08:16:25.316Z
123
3.14
hello
true
```

我们可以看出，`await` 和 `hprose.co` 除了对生成器的处理不同以外，其它的都相同。对于生成器函数，`await` 是按原样返回的，而 `hprose.co` 则是按照 `tj/co` 的方式处理。也就是说 `hprose.co` 综合了 `await` 和 `tj/co` 的全部优点。使用 `hprose.co` 比使用 `await` 或 `tj/co` 都方便。

yield 数组或对象

我们来看第二个让 `tj/co` 崩溃的例子：

```
var co = require('co');

co(function*() {
    try {
        var a = [];
        for (i = 0; i < 1000000; i++) {
            a[i] = i;
        }
        var start = Date.now();;
        yield a;
        var end = Date.now();;
        console.log(end - start);
    }
    catch (e) {
        console.error(e);
    }
});

co(function*() {
    try {
        var a = [];
        a[0] = a;
```

```
        console.log(yield a);
    }
    catch (e) {
        console.error(e);
    }
});

co(function*() {
    try {
        var o = {};
        o.self = o;
        console.log(yield o);
    }
    catch (e) {
        console.error(e);
    }
});
```

运行该程序，我们会看到程序会卡一会儿，然后出现下面的结果：

```
2530
(node:70754) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 1): RangeError: Maximum call stack size exceeded
(node:70754) DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.
(node:70754) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 2): RangeError: Maximum call stack size exceeded
```

上面的 `2530` 是第一个 `co` 程序段输出的结果，也就是说这个 `yield` 要等待 2.5 秒才能返回结果。而后面两个 `co` 程序段则直接调用栈溢出了。如果在实际应用中，出现了这样的数据，使用 `tj/co` 你的程序就会变得很慢，或者直接崩溃了。

下面看看 `hprose.co` 的效果，同样只替换第一句话为：

```
var co = require('hprose').co;
```

后面的代码都不需要修改，我们来看看运行结果：

```
7
[ [Circular] ]
{ self: [Circular] }
```

第一个 `co` 程序段用时很短，只需要 `7` ms。注意，这还是包含了后面两个程序段的时间，因为这三个协程是并发的，如果去掉后面两个程序段，你看的输出可能是 `1` ms 或者 `0` ms。而后面两个程序段也完美的返回了带有循环引用的数据。这才是我们期望的结果。

我们再来看看 `async/await` 下是什么效果，程序代码如下：

```
(async function() {
    try {
        var a = [];
        for (i = 0; i < 1000000; i++) {
            a[i] = i;
        }
        var start = Date.now();
        await a;
        var end = Date.now();
        console.log(end - start);
    }
    catch (e) {
        console.error(e);
    }
})();

(async function() {
    try {
        var a = [];
        a[0] = a;
        console.log(await a);
    }
    catch (e) {
        console.error(e);
    }
})();

(async function() {
    try {
        var o = {};
        o.self = o;
        console.log(await o);
    }
    catch (e) {
        console.error(e);
    }
})();
```

运行结果如下：

```
14
```

```
2. [ [Circular] ]
3. { self: [Circular] }
```

我们发现 `async/await` 的输出结果跟 `hprose.co` 是一致的，但是在性能上，`hprose.co` 则比 `async/await` 还要快 1 倍。因此，第二个回合，`hprose.co` 仍然是完胜 `tj/co` 和 `async/await`。

yield thunk 函数

我们再来看看 `tj/co` 和 `tj/thunkify` 是多么的让人抓狂，以及 `hprose.co` 和 `hprose.thunkify` 是如何优雅的解决 `tj/co` 和 `tj/thunkify` 带来的这些让人抓狂的问题的。

首先我们来看第一个问题：

`tj/thunkify` 返回的 thunk 函数的执行结果是一次性的，不能像 `promise` 结果那样被使用多次，我们来看看下面这个例子：

```
1. var co = require("co");
2. var thunkify = require("thunkify");
3.
4. var sum = thunkify(function(a, b, callback) {
5.     callback(null, a + b);
6. });
7.
8. co(function*() {
9.     var result = sum(1, 2);
10.    console.log(yield result);
11.    console.log(yield sum(2, 3));
12.    console.log(yield result);
13. });
```

这个例子很简单，输出结果你猜是啥？

```
1. 3
2. 5
3. 3
```

是上面的结果吗？恭喜你，答错了！不过，这不是你的错，而是 `tj/thunkify` 的错，它的结果是：

```
3
5
```

什么？最后的 `console.log(yield result)` 输出结果哪儿去了？不好意思，`tj/thunkify` 解释说是为了防止 `callback` 被重复执行，所以就只能这么玩了。可是真的是这样吗？

我们来看看使用 `hprose.co` 和 `hprose.thunkify` 的执行结果吧，把开头两行换成下面三行：

```
var hprose = require("hprose");
var co = hprose.co;
var thunkify = hprose.thunkify;
```

其它代码都不用改，运行它，你会发现预期的结果出来了，就是：

```
3
5
3
```

可能你还不服气，你会说，`tj/thunkify` 这样做是为了防止类似被 `thunkify` 的函数中，回调被多次调用时，`yield` 的结果不正确，比如：

```
var sum = thunkify(function(a, b, callback) {
    callback(null, a + b);
    callback(null, a + b + a);
});

co(function*() {
    var result = sum(1, 2);
    console.log(yield result);
    console.log(yield sum(2, 3));
    console.log(yield result);
});
```

如果 `tj/thunkify` 不这样做，结果可能就会变成：

```
3
4
5
```

可是真的是这样吗？你会发现，即使改成上面的样子，`hprose.thunkify` 配合 `hprose.co` 返回的结果仍然是：

```
3
5
3
```

跟预期的一样，回调函数并没有重复执行，错误的结果并没有出现。而且当需要重复 `yield` 结果函数时，还能够正确得到结果。

最后我们再来看一下，`tj/thunkify` 这样做真的解决了问题了吗？我们把代码改成下面这样：

```
var sum = thunkify(function(a, b, callback) {
    console.log("call sum(" + Array.prototype.join.call(arguments) + ")");
    callback(null, a + b);
    callback(null, a + b + a);
});

co(function*() {
    var result = sum(1, 2);
    console.log(yield result);
    console.log(yield sum(2, 3));
    console.log(yield result);
});
```

然后替换不同的 `co` 和 `thunkify`，然后执行，我们会发现，`tj` 版本的输出如下：

```
call sum(1,2,function (){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      })
3
call sum(2,3,function (){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      })
5
```

```
call sum(1,2,function (){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      },function (){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      })
```

而 `hprose` 版本的输出结果如下：

```
call sum(1,2,function () {
                thisArg = this;
                results.resolve(arguments);
            })
3
call sum(2,3,function () {
                thisArg = this;
                results.resolve(arguments);
            })
5
3
```

从这里，我们可以看出，`tj` 版本的程序在执行第二次 `yield result` 时，简直错的离谱，它不但没有让我们得到预期的结果，反而还重复执行了 `thunkify` 后的函数，而且带入的参数也完全不对了，所以，这是一个完全错误的实现。

而从 `hprose` 版本的输出来看，`hprose` 不但完美的避免了回调被重复执行，而且保证了被 `thunkify` 后的函数执行的结果被多次 `yield` 时，也不会被重复执行，而且还能够得到预期的结果，可以实现跟返回 promise 对象一样的效果。

`tj` 因为没有解决他所实现的 `thunkify` 函数带来的这些问题，所以在后期推荐大家放弃 `thunkify`，转而投奔到返回 `promise` 对象的怀抱中，而实际上，这个问题并非是不能解决的。

`hprose` 在对 `thunkify` 函数的处理上，再次完胜 `tj`。而这个回合中，`async/await` 就不用提了，因为 `async/await` 完全不支持对 thunk 函数进行 `await`。

这还不是 `hprose.co` 和 `hprose.thunkify` 的全部呢，再继续看下面这个例子：

```
var sum = thunkify(function(a, b, callback) {
    callback(a + b);
});

co(function*() {
    var result = sum(1, 2);
    console.log(yield result);
    console.log(yield sum(2, 3));
    console.log(yield result);
});
```

这里开头对 `hprose` 和 `tj` 版本的不同 `co` 和 `thunkify` 实现的引用就省略了，请大家自行脑补。

上面这段程序，如果使用 `tj` 版本的 `co` 和 `thunkify` 实现，运行结果是这样的：

```
(node:75927) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 2): 3
(node:75927) DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.
```

而如果使用 `hprose` 版本的 `co` 和 `thunkify` 实现，运行结果是这样的：

```
3
5
3
```

`hprose` 版本的运行结果再次符合预期，而 `tj` 版本的运行结果再次让人失望之极。

进过上面三个回合的较量，我们发现 hprose 的协程完胜 `tj` 和 `async/await`，而且 `tj` 的实现是惨败，`async/await` 虽然比 `tj` 稍微好那么一点，但是跟 `hprose` 所实现协程比起来，也是望尘莫及。

所以，用 `tj/co` 和 `async/await` 感觉很不爽的同学，可以试试 `hprose.co` 了，绝对让你爽歪歪。

多协程并发

协程内的并发

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。对于其它的异步函数也是如此。例如：

```
var hprose = require('hprose');

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    yield client.useService();
    console.log(yield client.hello("Hprose"));
    var a = client.sum(1, 2, 3);
    var b = client.sum(4, 5, 6);
    var c = client.sum(7, 8, 9);
    console.log(yield client.sum(a, b, c));
    console.log(yield client.hello("World"));
});
```

在上面的例子中，client 是一个 Hprose 的异步 Http 客户端。

所以 client.hello 和 client.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 `client.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，`client.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：

```
Hello Hprose
45
Hello World
```

从结果中，我们可以看出，三次调用的结果是顺序输出的，因为这三个输出都是用 `yield` 来同步获取结果的。

协程间的并发

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：

```
var hprose = require('hprose');

var client = hprose.Client.create('http://hprose.com/example/');
var proxy = client.useService();

hprose.co(function*() {
    var client = yield proxy;
    for (var i = 0; i < 5; i++) {
        console.log((yield client.hello("1-" + i)));
    }
});

hprose.co(function*() {
    var client = yield proxy;
    for (var i = 0; i < 5; i++) {
        console.log((yield client.hello("2-" + i)));
    }
});
```

我们运行该程序之后，可以看到如下结果：

```
Hello 1-0
Hello 2-0
Hello 2-1
Hello 1-1
Hello 1-2
Hello 2-2
Hello 1-3
Hello 2-3
Hello 1-4
Hello 2-4
```

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 client 客户端的创建拿到了协程外面。

但是对于 `client.useService` 返回的 `proxy`，我们在两个协程中都对它进行了一次 `yield`，原因是我们如果不这样做，就不能保证后面的 client 已同步的获取到了服务列表。

协程的参数和返回值

`co` 函数允许传参给协程。

`co` 函数本身的返回值也是一个 `promise` 对象。

下面这个例子演示了传参和 co 函数返回值的使用：

```
var hprose = require('hprose');

function *hello(n, client) {
    var result = [];
    for (var i = 0; i < 5; i++) {
        result[i] = client.hello(n + "-" + i);
    }
    return Promise.all(result);
}

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    yield client.useService();
    var result = yield hprose.co(function *(client) {
        var result = [];
        for (var i = 0; i < 3; i++) {
            result[i] = hprose.co(hello, i, client);
        }
        return Promise.all(result);
    }, client);
    console.log(result);
});
```

该程序执行结果为：

```
[ [ 'Hello 0-0', 'Hello 0-1', 'Hello 0-2', 'Hello 0-3', 'Hello 0-4' ],
  [ 'Hello 1-0', 'Hello 1-1', 'Hello 1-2', 'Hello 1-3', 'Hello 1-4' ],
  [ 'Hello 2-0', 'Hello 2-1', 'Hello 2-2', 'Hello 2-3', 'Hello 2-4' ] ]
```

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。

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，`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：

```
var hprose = require('hprose');

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    yield client.useService();
    for (var i = 0; i < 5; i++) {
        console.log(yield client.hello("1-" + i));
    }
    var console_log = hprose.wrap(console.log, console);
    for (var i = 0; i < 5; i++) {
        console_log(client.hello("2-" + i));
    }
    for (var i = 0; i < 5; i++) {
        console.log(yield client.hello("3-" + i));
    }
});
```

运行该程序之后，执行结果为：

```
Hello 1-0
Hello 1-1
Hello 1-2
Hello 1-3
Hello 1-4
Hello 2-0
Hello 2-1
Hello 2-4
Hello 3-0
Hello 2-2
Hello 2-3
Hello 3-1
Hello 3-2
Hello 3-3
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（生成器）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：

```
var hprose = require('hprose');


var coroutine = hprose.wrap(function*(client) {
    console.log(1);
    console.log((yield client.hello("hprose")));
    var a = client.sum(1, 2, 3);
    var b = client.sum(4, 5, 6);
    var c = client.sum(7, 8, 9);
    console.log((yield client.sum(a, b, c)));
    console.log((yield client.hello("world")));
});

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    yield client.useService();
    coroutine(client);
    coroutine(Promise.resolve(client));
});
```

该程序执行结果为：

```
1
1
Hello hprose
Hello hprose
45
45
Hello world
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。

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：

```
var hprose = require('hprose');

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    try {
        console.log(yield client.invoke('ooxx'));
    }
    catch (e) {
        console.log(e.message);
    }
});
```

该程序运行结果为：

```
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：

```
var hprose = require('hprose');

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    console.log(yield client.invoke('oo'));
    console.log(yield client.invoke('xx'));
}).catch(function(e) {
    console.log(e.message);
});
```

该程序运行结果为：

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

有时候，我们并不想对异常使用 `try` `catch` 处理，而是希望异常也能跟正常返回值一样被返回，那么我们可以使用 `complete` 方法。例如：

```
var hprose = require('hprose');

hprose.co(function*() {
    var client = hprose.Client.create('http://hprose.com/example/');
    console.log(yield client.invoke('oo').complete());
    console.log(yield client.invoke('xx').complete());
});
```

该程序运行结果如下：

```
Error: Can't find this function oo().
    at /usr/local/lib/node_modules/hprose/lib/client/Client.js:437:37
    at /usr/local/lib/node_modules/hprose/lib/common/Future.js:531:25
    at _combinedTickCallback (internal/process/next_tick.js:67:7)
    at process._tickCallback (internal/process/next_tick.js:98:9)
Error: Can't find this function xx().
    at /usr/local/lib/node_modules/hprose/lib/client/Client.js:437:37
    at /usr/local/lib/node_modules/hprose/lib/common/Future.js:531:25
    at _combinedTickCallback (internal/process/next_tick.js:67:7)
    at process._tickCallback (internal/process/next_tick.js:98:9)
```

因为 `complete` 是 hprose 实现的 `Future` 类型对象上的一个方法，因此，如果你使用的是其它的 `Promise` 库，你可以自行实现该方法，或者先通过 `hprose.Future.resolve` 方法将其它方式实现的 promise 对象转换为 hprose 的 `Future` 对象再调用该方法。

promisify 方法

`promisify` 的作用是将一个使用回调的方法转换为一个返回 promise 对象的方法。它所支持的被包装的方法跟 `thunkify` 所支持的是一样的，但他们返回的是不同的包装函数。例如：

```
var hprose = require('hprose');

function sum(a, b, callback) {
    callback(a + b);
}

var sum1 = hprose.promisify(sum);
```

```
var sum2 = hprose.thunkify(sum);

sum1(1, 2).then(function(result) {
    console.log(result);
});

sum2(2, 3)(function(result) {
    console.log(result);
});

hprose.co(function*() {
    console.log(yield sum1(3, 4));
    console.log(yield sum2(4, 5));
});

(async function() {
    console.log(await sum1(5, 6));
    console.log(await sum2(6, 7));
})();
```

执行结果为：

```
3
5
7
9
11
[Function]
```

不管是通过 `thunkify` 包装的函数，还是通过 `promisify` 包装的函数，都支持在 `hprose.co` 协程中进行 `yield` 调用。但是只有 `promisify` 包装的函数支持在 `async/await` 中使用 `await` 调用，而 `thunkify` 包装的则不支持。因此，在 `promisify` 和 `thunkify` 之间，首选 `promisify`。

toPromise 方法

这是 `Future` 对象上的一个方法，在 Promise 异步编程 一章中我们介绍过，但是它还有一个功能，我们在那一章没有提到过。因为它是跟协程相关的。

在前面的介绍中，我们知道迭代器函数作为协程执行有两种方式，一种是通过 `hprose.co` 来直接执行，另一种是通过 `hprose.wrap` 来包装成普通函数执行。但是这里还有一个问题没有解决。当然这个问题比较特殊，那就是当我们要执行的函数不知道是一个普通函数还是一个迭代器函数，而且函数的返回值还是一个函数的时候，我们就不能使用 `hprose.co` 来直接执行，或者使用 `hprose.wrap` 包

装一下再执行了。因为前者会导致返回的函数结果被 `thunkify`，后者可能会导致重复包装，降低函数执行效率，还会带来一个副作用，就是函数想要接收的参数本来就是原始的 `Promise` 参数的话，在这种情况下，会变成普通参数被带入，这虽然方便，但有时候却不是我们期望的方式。

我来举一个例子：

比如有这样两个函数：

```
function normal(p) {
    console.log(p);
    return normal;
}

function* coroutine(p) {
    console.log(yield p);
    return coroutine;
}
```

现在我们希望把它们作为参数带入另一个函数中执行，假设函数叫 `run`：

```
function* run(fn) {
   var p = Promise.resolve(123);
   ...
}

hprose.co(function*() {
    yield run(normal);
    yield run(coroutine);
});
```

这里我们先不定义 `run` 的函数体，但是我们希望上面的执行结果跟下面这段代码一样：

```
hprose.co(function*() {
    var p = Promise.resolve(123);
    console.log(normal(p));
    console.log(yield coroutine(p));
})
```

显然这样写是不行的：

```
function* run(fn) {
   var p = Promise.resolve(123);
   console.log(fn(p));
}
```

这样写的执行结果是：

```
Promise { 123 }
[Function: normal]
{}
```

我们得不到 `coroutine` 执行的结果。

而如果这样写：

```
function* run(fn) {
   var p = Promise.resolve(123);
   console.log(yield fn(p));
}
```

执行结果为：

```
Promise { 123 }
[Function]
```

也不对。

如果这样写：

```
function* run(fn) {
   var p = Promise.resolve(123);
   console.log(yield hprose.co(fn(p)));
}
```

执行结果为：

```
Promise { 123 }
undefined
[Function: normal]
123
[Function: coroutine]
```

还是不对。

改成：

```
function* run(fn) {
    var p = Promise.resolve(123);
    console.log(yield hprose.wrap(fn)(p));
}
```

结果为：

```
123
[Function: normal]
123
[Function: coroutine]
```

仍然不对。

好，最后我们来看大招：

```
function* run(fn) {
    var p = Promise.resolve(123);
    console.log(yield hprose.Future.toPromise(fn(p)));
}
```

现在结果对了：

```
Promise { 123 }
[Function: normal]
123
[Function: coroutine]
```

这个结果跟我们期望的结果是一样的。这就是 `Future.toPromise` 的另一个用处。

> 原文: *https://github.com/hprose/hprose-html5/wiki/%E5%8D%8F%E7%A8%8B*

客户端的特殊设置

Hprose 的 HTTP 客户端

setHeader 方法

```
client.setHeader(name, value);
```

这个方法专门用于设置 HTTP 的头信息。

certificate 属性

该属性用于设置 HTTPS 客户端证书。但仅支持 APICloud 客户端。

onprogress 事件

对应 `XMLHttpRequest.upload` 的 `onprogress` 事件。

onRequestProgress 事件

对应 `XMLHttpRequest.upload` 的 `onprogress` 事件。

onResponseProgress 事件

对应 `XMLHttpRequest` 的 `onprogress` 事件。

Hprose 的 TCP 客户端

hprose for HTML5 的 TCP 客户端是基于 chrome.sockets.tcp 实现的。因此它只可以在 chrome apps 或基于 ionic/cordova 的 android、iOS 应用中使用。

在 ionic/cordova 中使用时，需要安装 cordova-plugin-chrome-apps-sockets-tcp 插件。

noDelay 属性

用于设置在 TCP 上是否禁用 Nagle 算法。默认为 `true`。

fullDuplex 属性

用于设置是否启用全双工通讯方式。在全双工通讯方式下，客户端发出调用请求后，不需要等待返回响应结果，就可以在同一个连接上发送下一个调用请求，而且在发送请求的过程中，可以同时接收服务器端返回的响应数据。也就是说，在该模式下，客户端和服务器之间的连接的利用率将会更高。尤其是在服务器返回响应较慢，或者服务器和客户端之间有推送请求时，客户端和服务器端只需要一个连接就足够了。而在半双工通讯模式下，则需要多个连接才可以。

不过该属性默认值为 `false`，原因是 hprose 旧版本的 TCP 服务器仅支持半双工通讯方式。你只有确认服务器端支持全双工通讯方式的情况下，才可以开启该选项，否则，在通讯中会得到异常信息。

maxPoolSize 属性

用于设置连接池中的最大连接数。默认值为 10。当开启全双工通讯方式后，可以将该值设置为 1。

poolTimeout 属性

用于设置连接池中的连接的空闲超时时间。默认值为 30000，单位是毫秒（ms）。

当连接池中的连接超过这个时间还没有被使用的情况下，将会自动关闭。

> 原文: *https://github.com/hprose/hprose-html5/wiki/%E5%AE%A2%E6%88%B7%E7%AB%AF%E7%9A%84%E7%89%B9%E6%AE%8A%E8%AE%BE%E7%BD%AE*

推送服务

subscribe 方法

```
client.subscribe(topic, callback[, timeout[, failswitch]]);
client.subscribe(topic, 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 `topic` 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 `topic` 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 `topic` 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，客户端会自动通过 `client` 的 `id` 属性来获取，如果该属性未初始化，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整数自增计数器。当用户指定了 `id` 参数时，客户端会将它作为该 `topic` 方法的参数值传给服务器端，但不会修改客户端的 `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，而是重新发起对 `topic` 方法的调用。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

```
client.unsubscribe(topic)
client.unsubscribe(topic, callback)
client.unsubscribe(topic, id)
client.unsubscribe(topic, 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` 参数若未指定，那么当客户端 `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 `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

```
client.isSubscribed(topic);
```

当 topic 已被订阅时，返回 `true`，否则返回 `false`。

subscribedList 方法

```
client.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。

> 原文: *https://github.com/hprose/hprose-html5/wiki/%E6%8E%A8%E9%80%81%E6%9C%8D%E5%8A%A1*

输入输出——BytesIO

概述

JavaScript 中提供了 `ArrayBuffer` 和 `Uint8Array` 等 `TypedArray` 可以对二进制数据进行操作。但这些对象没有自动伸缩性，不方便流式操作，相互转换也比较麻烦。

为方便操作二进制数据，hprose 提供了一个 `BytesIO` 对象。`BytesIO` 是一个可自动伸缩的，可流式读写数据的工具类。

该工具类还可以对 `String`、`ArrayBuffer`、`Uint8Array` 进行快速的相互转换。

创建 BytesIO 对象

```
1. new BytesIO();
2. new BytesIO(data);
3. new BytesIO(capacity);
4. new BytesIO(data[, byteOffset[, length]]);
```

不带参数的构造器创建一个空的 `BytesIO` 对象。空的 `BytesIO` 对象可以进行写操作。

带 1 个参数的构造器，其中 `data` 参数可以是 `String`、`BytesIO`、`Uint8Array`、`ArrayBuffer`、其它 `TypedArray` 类型的对象或是元素是纯整数且数字范围是 0-255 的普通数组。

如果参数为 1 个整数值 `capacity`，则表示预分配 `capacity` 个字节的空间，当写入比较大量的数据时，预分配合适的空间将会有效的提高运行效率。

带 2-3 个参数的构造器，其中 `data` 参数为 `ArrayBuffer` 类型的对象，`byteOffset` 为起始偏移量，`length` 为数据长度。

方法

toString 方法

```
1. BytesIO.toString(data);
```

该方法的功能是将 `data` 转换为字符串返回，`data` 本身不会被修改。

其中，`data` 参数可以为：`String`、`BytesIO`、`ArrayBuffer`、`Uint8Array` 类型的对象或保存有 charcode 的数字数组。

如果是 `String` 类型的对象，则直接返回该对象。

如果是 `BytesIO`、`ArrayBuffer`、`Uint8Array` 类型的对象，则按照 UTF-8 编码对其中的二进制数据进行解析并返回解析后的 `String` 对象。

如果这些二进制数据不能按照 `UTF-8` 编码解析成字符串，会抛出异常。

如果是保存有 charcode 的数字数组，则按照 UTF-16 编码进行解析，并返回解析后的 `String` 对象。

BytesIO.prototype 上的属性

下面我们用 `bytesIO` 这个变量名来指代 `BytesIO` 的实例对象。

length 属性

```
1. bytesIO.length;
```

只读属性，返回当前 `bytesIO` 对象的长度。

capacity 属性

```
1. bytesIO.capacity;
```

只读属性，返回当前 `bytesIO` 对象的容量，当写入数据超过该容量时，`bytesIO` 对象会自动扩容，`capacity` 的数值也会相应改变。

position 属性

```
1. bytesIO.position;
```

只读属性，该属性表示在对 `bytesIO` 对象进行读取操作时的当前位置。当 `position` 为 0 时，表示位于数据的开头，当 `position` 与 `length` 相同时，表示已没有可读取的数据。

bytes 属性

```
1. bytesIO.bytes;
```

只读属性，该属性返回 `bytesIO` 对象内部二进制数据的一个 `Uint8Array` 的共享视图。因为没有数据复制操作，所以返回该属性非常快速。注意对该属性所返回的数据进行写操作是不安全的，它会连同 `bytesIO` 对象内部的数据一起修改。

BytesIO.prototype 上的方法

下面我们同样用 `bytesIO` 这个变量名来指代 `BytesIO` 的实例对象。

mark 方法

```
1. bytesIO.mark();
```

保存当前的读写位置。如果对该方法进行多次调用，则只有最后一次保存的读写位置可以被恢复。

无返回值。

reset 方法

```
1. bytesIO.reset();
```

恢复由 `mark` 方法保存的读写位置。

无返回值。

clear 方法

```
1. bytesIO.reset();
```

清空 `bytesIO` 对象，并将读写位置归零。

无返回值。

writeByte 方法

```
1. bytesIO.writeByte(byte);
```

往 `bytesIO` 对象的末尾写入一个字节。

byte 的范围是 0-255 的整数。

执行成功后，`bytesIO` 对象的 `length` 属性将会相应加 1。如果容量不足，将会自动扩容。

无返回值。

writeInt32BE 方法

```
1. bytesIO.writeInt32BE(int32);
```

往 `bytesIO` 对象的末尾按照 BigEndian 方式写入一个32位有符号整数（4个字节）。

`int32` 的范围是从 -2147483648 到 2147483647，超出范围会抛出 `TypeError('value is out of bounds')` 的异常。

执行成功后，`bytesIO` 对象的 `length` 属性将会相应加 4。如果容量不足，将会自动扩容。

无返回值。

writeUInt32BE 方法

```
1. bytesIO.writeUInt32BE(uint32);
```

往 `bytesIO` 对象的末尾按照 BigEndian 方式写入一个32位无符号整数（4个字节）。

`uint32` 的范围是从 0 到 4294967295，超出范围会抛出 `TypeError('value is out of bounds')` 的异常。

执行成功后，`bytesIO` 对象的 `length` 属性将会相应加 4。如果容量不足，将会自动扩容。

无返回值。

writeInt32LE 方法

```
1. bytesIO.writeInt32LE(int32);
```

往 `bytesIO` 对象的末尾按照 LittleEndian 方式写入一个32位有符号整数（4个字节）。

`int32` 的范围是从 -2147483648 到 2147483647，超出范围会抛出 `TypeError('value is out of bounds')` 的异常。

执行成功后，`bytesIO` 对象的 `length` 属性将会相应加 4。如果容量不足，将会自动扩容。

无返回值。

writeUInt32LE 方法

```
1. bytesIO.writeUInt32LE(uint32);
```

往 `bytesIO` 对象的末尾按照 LittleEndian 方式写入一个32位无符号整数（4个字节）。

`uint32` 的范围是从 0 到 4294967295，超出范围会抛出 `TypeError('value is out of bounds')` 的异常。

执行成功后，`bytesIO` 对象的 `length` 属性将会相应加 4。如果容量不足，将会自动扩容。

无返回值。

write 方法

```
1. bytesIO.write(data);
```

往 `bytesIO` 对象的末尾写入二进制数据 `data`。

其中 `data` 可以是 `ArrayBuffer`、`Uint8Array`、`BytesIO` 类型的对象，或元素是纯整数且数字范围是 0-255 的普通数组。

执行成功后，`bytesIO` 对象的 `length` 属性将会增加相应的长度。如果容量不足，将会自动扩容。

无返回值。

writeAsciiString 方法

```
1. bytesIO.writeAsciiString(str);
```

往 `bytesIO` 对象的末尾写入 ASCII 编码的字符串数据 `str`。

其中 `str` 中的每个字符的 charcode 编码范围为 0-255。

执行成功后，`bytesIO` 对象的 `length` 属性将会增加相应的长度。如果容量不足，将会自动扩容。

无返回值。

writeString 方法

```
1. bytesIO.writeString(str);
```

往 `bytesIO` 对象的末尾按照 UTF-8 编码的方式写入字符串数据 `str` 。

执行成功后，`bytesIO` 对象的 `length` 属性将会增加相应的长度。如果容量不足，将会自动扩容。

无返回值。

readByte 方法

```
1. bytesIO.readByte();
```

从 `bytesIO` 对象的当前位置读取一个字节，并返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 1。

如果当前位置已在结尾，则返回 -1。

readInt32BE 方法

```
1. bytesIO.readInt32BE();
```

从 `bytesIO` 对象的当前位置读取 4 个字节，按照 BigEndian 编码方式转换为一个 32 位有符号整型数，并返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 4。

如果当前位置到结尾不足 4 个字节，则抛出 `Error('EOF')` 异常。

readUInt32BE 方法

```
1. bytesIO.readUInt32BE();
```

从 `bytesIO` 对象的当前位置读取 4 个字节，按照 BigEndian 编码方式转换

为一个 32 位无符号整型数，并返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 4。

如果当前位置到结尾不足 4 个字节，则抛出 `Error('EOF')` 异常。

readInt32LE 方法

```
1. bytesIO.readInt32LE();
```

从 `bytesIO` 对象的当前位置读取 4 个字节，按照 LittleEndian 编码方式转换为一个 32 位有符号整型数，并返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 4。

如果当前位置到结尾不足 4 个字节，则抛出 `Error('EOF')` 异常。

readUInt32LE 方法

```
1. bytesIO.readUInt32LE();
```

从 `bytesIO` 对象的当前位置读取 4 个字节，按照 LittleEndian 编码方式转换为一个 32 位无符号整型数，并返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 4。

如果当前位置到结尾不足 4 个字节，则抛出 `Error('EOF')` 异常。

read 方法

```
1. bytesIO.read(n);
```

从 `bytesIO` 对象的当前位置读取 `n` 个字节，并返回。返回结果是一个 Uint8Array 类型的共享视图。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 `n`。

如果当前位置到结尾不足 `n` 个字节，则读取全部剩余字节并返回，`bytesIO` 对象的 `position` 属性将被设置为与 `length` 相同。

因为返回结果是 `bytesIO` 对象内部数据的一部分的共享视图，因此该操作速度很快。但是对读取结果进行写操作是不安全的，它会影响 `bytesIO` 的内部数据。因此，如果需要对返回结果进行写操作，最好是使用 `new Uint8Array(data)` 来获取一个数据副本，在该副本上进行写操作，其中 `data` 表示返回数据。

skip 方法

```
1. bytesIO.skip(n);
```

从 `bytesIO` 对象的当前位置略过 `n` 个字节，并返回实际略过的字节数。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应加 n。

如果当前位置到结尾不足 `n` 个字节，则略过全部剩余字节并返回略过的剩余字节数，`bytesIO` 对象的 `position` 属性将被设置为与 `length` 相同。

readBytes 方法

```
1. bytesIO.readBytes(tag);
```

从 `bytesIO` 对象的当前位置开始读取，直到遇到与 `tag` 相同的字节为止，并以 `Uint8Array` 类型的对象返回读取到的二进制数据。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应增加读取到的字节数。

读取到的二进制数据中包含有最后的 `tag` 字节。

如果读取到结尾仍然没有遇到与 `tag` 相同的字节，则返回所有剩余的字节，`bytesIO` 对象的 `position` 属性将被设置为与 `length` 相同。

返回结果是 `bytesIO` 对象内部数据的一部分的共享视图，因此该操作速度很快。但是对读取结果进行写操作是不安全的，它会影响 `bytesIO` 的内部数据。因此，如果需要对返回结果进行写操作，最好是使用 `new Uint8Array(data)` 来获取一个数据副本，在该副本上进行写操作，其中 `data` 表示返回数据。

readUntil 方法

```
1. bytesIO.readUntil(tag);
```

从 `bytesIO` 对象的当前位置开始读取，直到遇到与 `tag` 相同的字节为止，并将读取到的二进制数据按照 UTF-8 编码解析为字符串返回。

执行成功后，`bytesIO` 对象的 `position` 属性将会相应增加读取到的字节数。

读取到的数据中不包含有最后的 `tag` 字节。但在位置计算上会算上 `tag` 字节。

如果读取到结尾仍然没有遇到与 `tag` 相同的字节，则所有剩余的数据按照 UTF-8 编码解析为字符串返回。`bytesIO` 对象的 `position` 属性将被设置为

与 `length` 相同。

readAsciiString 方法

```
1. bytesIO.readAsciiString(n);
```

读取 `n` 个字节，并按照 ASCII 编码方式解析为字符串返回。

执行成功后， `bytesIO` 对象的 `position` 属性将会相应增加读取到的字节数。

如果当前位置到结尾不足 `n` 个字节，则读取全部剩余字节并按照 ASCII 编码方式解析为字符串返回， `bytesIO` 对象的 `position` 属性将被设置为与 `length` 相同。

readString 方法

```
1. bytesIO.readString(n);
```

读取 `n` 个字符，并按照 UTF-8 编码方式解析为字符串返回。这里的 `n` 指的是最后读取到的字符串的 `length` 长度，而不是读取的字节数。

执行成功后， `bytesIO` 对象的 `position` 属性将会相应增加读取到的字节数。

如果当前位置到结尾不足 `n` 个字符，则读取全部剩余字节并按照 UTF-8 编码方式解析为字符串返回， `bytesIO` 对象的 `position` 属性将被设置为与 `length` 相同。

如果读取到的二进制数据不能按照 `UTF-8` 编码解析成字符串，会抛出异常。

readStringAsBytes 方法

```
1. bytesIO.readStringAsBytes(n);
```

该功能与上面的 `readString` 方法类似，但是返回的结果是 UTF-8 编码的 `Uint8Array` 实例对象，同 `readString` 方法相比，因为减少了解码过程，所以该方法速度要明显快于 `readString` 方法。

另外要注意，本方法的返回结果是 `bytesIO` 对象内部数据的一部分的共享视图，因此对读取结果进行写操作是不安全的，它会影响 `bytesIO` 的内部数据。如果需要对返回结果进行写操作，最好是使用 `new Uint8Array(data)` 来获取一个数据副本，在该副本上进行写操作，其中 `data` 表示返回数据。

takeBytes 方法

```
1. bytesIO.takeBytes();
```

该方法的返回值跟 `bytes` 属性返回值相同。区别是该方法在返回值后，`bytesIO` 中的数据会被清空。因此返回的数据可以安全进行写操作。

toBytes 方法

```
1. bytesIO.toBytes();
```

该方法的返回值是 `bytes` 属性返回值的副本。因此返回的数据可以安全进行写操作。但因为要创建副本，因此，速度上要比使用 `bytes` 属性慢一些。

toString 方法

```
1. bytesIO.toString();
```

该方法将对 `bytesIO` 对象中的所有二进制数据按照 UTF-8 编码进行解析并返回字符串。

如果 `bytesIO` 对象中的二进制数据不能按照 `UTF-8` 编码解析成字符串，会抛出异常。

clone 方法

```
1. bytesIO.clone();
```

返回 `bytesIO` 对象的副本。

trunc 方法

```
1. bytesIO.trunc();
```

对 `bytesIO` 对象中已经读取的数据部分进行截断处理，只保留未读取的数据，读取偏移量将归零，数据长度将变为剩余数据的长度。如果之前执行过 `mark` 方法，`mark` 方法保存的位置信息也会被归零。

> 原文: *https://github.com/hprose/hprose-html5/wiki/%E8%BE%93%E5%85%A5%E8%BE%93%E5%87%BA%E2%80%94%E2%80%94BytesIO*