链滴

秒杀 tj/co 的 hprose 协程库

作者：andot

原文链接：https://ld246.com/article/1479867699310

来源网站：链滴

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，但是比 tj/co 更加完善，下面我们就来详细介绍下它们之间的差别。

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。

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

```
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：

```
promise
generator
2016-11-19T06:54:30.081Z
123
3.14
hello
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：

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

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，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，wait 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 futu
e, promise rejections that are not handled will terminate the Node.js process with a non-zero
xit 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
[ [Circular] ]
{ 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，我们看看下面这个例子：

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

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

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

这个例子很简单，输出结果你猜是啥？

```
3
5
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 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，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，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 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。

hprose 有 4 个 JavaScript 版本，它们都支持上面的协程库，它们的地址分别是：

- hprose for Node.js(oschina镜像)
- hprose for HTML5(oschina镜像)
- hprose for JavaScript(oschina镜像)
- hprose for 微信小程序(oschina镜像)

另外，如果你不需要使用 hprose 序列化和远程调用的话，下面还有一个专门的从 hprose 中精简出的 Promise A+ 实现和协程库：Future.js(oschina镜像)

当然该协程库的功能不止于此，更多介绍请参见：

- Promise 异步编程
- 协程