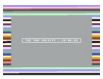
构建一个简单的 Promise 构造函数



马丁ISDN

2021 年 1 月 29 日 GPL3

JavaScript 承诺构造函数的因果关系。异步回调 vs Promise。

通过阅读本文，您应该更深入地了解内部 promise 和异步回调机制，并能够决定哪一个更适合您的问题。

下载源文件 - 679 B

介绍

> “一种低糖替代方法是在称为 *Continuation Passing Style* 的模式中使用普通回调。这种无承诺的风格可以帮助你避免我喜欢称之为代码糖尿病的情况，在这种情况下，重要的状态隐藏在语法糖之后，危险的角落案例潜伏着。” - 贝努瓦·埃西姆布雷

现在是时候看看 JavaScript Promises 了。它们是什么以及它们代表什么。他们承诺什么，他们提供什么。

我读过很多关于 Promises 的文章，并偶然发现了它的各种定义，其中大部分都提到了*proxy*这个词。我知道什么是代理服务器，但代理对象或代理值是什么？那，我只能想象。

让我们走一小段路，为自己构建一个 promise 构造函数。不是学术上认可的行业标准 A+/Promise，但对理解它们有很大帮助。首先，这个旅程从一个回调开始……

异步

JavaScript 是**单线程**、**非阻塞**和**异步的**。它有一个**调用堆栈**、一个**回调队列**和一个**事件循环**。现在让我们说实话，每个大胆的概念都需要一篇自己的文章。

诸如理解异步 JavaScript 之类的文本，或者如果您更喜欢视频，那么事件循环到底是什么？是必要的。

您可以将在计算机上执行的应用程序视为在操作系统中运行的进程。大多数应用程序在操作系统上作为单个进程运行，并且大多数应用程序只有一个执行线程。在 JavaScript 的情况下，该单线程执行将成为您的脚本。历史上，JS 程序在浏览器应用程序中执行。

为简单起见，我假设浏览器应用程序作为操作系统上的一个进程运行。浏览器应用程序如何管理其线程是特定于浏览器的，除了 Firefox 和 Chromium 之外，还有许多浏览器。

下一个简化将假设每个 HTML 文档有一个执行线程。该文档中引用或嵌入的所有脚本都组合在一个主体中，并且该统一脚本将在单个执行线程中运行。这是模型，不是事实。

人们通常认为异步函数需要时间。有时您自己的函数可以处理数据很长时间，而这本身并不会使其异步。我们可以编写一个函数，将运行一分钟的超大 3D 矩阵相乘，它仍然会同步执行。

而且，天哪，这需要很多时间是什么意思？2 毫秒对于一个函数来说是不是很多时间？是 103 毫秒吗？我见过对数据库的请求最多需要 60 秒...

很难争论哪些函数应该是异步的，哪些函数不应仅基于时间，因为时间是一个数量。即使是关于时间的最简单的问题也不能用“是”或“否”来回答。它需要多少钱？

另一方面，问题：单线程？和非阻塞？如果评估为`true`异步提升标志。每当您的执行脚本调用与同一进程的其他线程或您或其他人计算机上的其他进程通信的函数时，JavaScript 最好异步触发该函数。

对于这篇文章的目的，我会打电话给**异步函数**，其**通过异步回调函数传递数据**。您将该回调作为参数传递给被调用的异步函数。请注意异步函数和异步回调之间的区别。

JavaScript 会跟踪函数、它们的执行顺序以及它们在**调用堆栈**上的数据。您可以通过从脚本中调用函数来将函数放在调用堆栈中。被调用的函数总是放在调用函数顶部的堆栈中。

有两种方法可以进入调用堆栈。同步函数直接**在调用堆栈上启动**，由已经在**调用堆栈上**的某个函数**启动**。

当且仅当调用堆栈为空时，异步回调通过回调队列进入调用堆栈。

每个`async`函数都有一部分同步进入调用堆栈，向其他线程发出请求并在相对较短的时间内离开调用堆栈。当数据最终准备好时，作为异步回调的参数，该回调将在其最后一个位置进入**回调队列**。

事件循环的工作是在检测到它为空时将第一个回调从**事件队列移动**到**调用堆栈上。注意时间的差距。**异步回调不会在与请求数据的异步函数相同的时间范围内进入调用堆栈。调用堆栈在两者之间清空。

JavaScript 中**异步回调的**另一种说法是：**从回调队列一侧进入调用堆栈的函数。**

示例 1

JavaScript 复制代码

```
var fs = require("fs");

fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
    if (err) { console.trace(err); return }

    var file2 = data1.split(' ')[1] + ".txt";
    fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
        if (err) { console.trace(err); return }

        var file3 = data2.split(' ')[2] + ".txt";
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            if (err) { console.trace(err); return }

            console.log(data2);
            console.log(data3);
        });
    });
});
```

此示例工作所需的文件位于*files.tar.gz*存档中。Promise 构造函数的最终版本也是如此。

这个在 Node.js 中执行的简单示例演示了日常异步代码。我们发出异步请求。当数据准备好时，我们的回调被调用，将数据传递给它。然后我们从数据中提取第二个单词并使用它来生成我们的文件名。我们发出另一个文件读取等等.....直到第三次，我们终于读取了最后一个文件并打印了它，但我也打印了第二个文件。;)

代码清晰，忠实地代表了计算机中的内容。异步回调的反对者称之为回调地狱。这个特定的源代码缩进在处理回调地狱时出现，他们喜欢称之为厄运金字塔。

首先，如果我们取消嵌套回调函数，我们可以摆脱缩进。

示例 2

JavaScript 复制代码

```
var fs = require("fs");
```

```
function read1(err, data1) {
    if (err) { console.trace(err); return }

    var file2 = data1.split(' ')[1] + ".txt";
    fs.readFile(file2, {encoding: "ascii"}, read2);
}

function read2(err, data2) {
    if (err) { console.trace(err); return }

    var file3 = data2.split(' ')[2] + ".txt";
    fs.readFile(file3, {encoding: "ascii"}, read3);
}

function read3(err, data3) { console.log(data3); }

fs.readFile("file.txt", {encoding: "ascii"}, read1);
```

看起来更好看？现在它具有与使用 promise 的等效代码相同的缩进。虽然，示例 *1*和示例 *2*不是一回事。正确的源代码缩进表示其 2D 结构。提醒一下，在示例 *1* 中调用最后一个回调时，我可以打印`file3`，但我也可以打印`file2`。您不能`file2`在第二个示例中打印的函数中进行打印`file3`。

时间顺序对于理解异步回调非常重要。我想通过这两个例子来说明，每行同步代码需要 1 毫秒来执行，所有异步回调需要 100 毫秒才能到达**回调队列**。让我们假设**事件循环**在 zilch 毫秒内将事件队列中的第一个回调移动到执行堆栈中。

这些示例从`fs.readFile`想要从名为“ *file.txt* ”的文件中读取数据的函数开始。

在示例 *1* 中，此函数需要 1 毫秒才能完成并返回。从我们的角度来看，100 毫秒内什么都没有发生，尽管 Node.js 运行时使用了一个工作线程来完成我们与操作系统的工作并读取该文件。

之后这100毫秒突然浪费了我们定义为最后一个参数的回调`fs.readFile`函数被放置在**回调队列**在101ST毫秒。上面什么都没有，所以我们的回调是先进入先离开。该**事件循环**“看到”这一点，从队列中移动我们的异步回调到堆栈中。

注意从我们调用`fs.readFile`到异步回调进入**回调队列的**时间点经过了 101 毫秒的时间。还要注意自`fs.readFile`函数返回以来调用堆栈为空的 100 毫秒。这个时间间隔和异步回调无法进入**执行堆栈**的事实，而它被上面的函数使用并调用异步操作是你通常看不到回调显式返回值的原因。

继续执行我们的示例。异步回调输入执行在101ST毫秒和正在运行的，其完成的102码它的第一行第二毫秒。那将是`if`语句，检查错误。

接着，它会做一些数据处理，并提取下一个文件的名称被读入变量`file2`由103的端RD毫秒。一个新的呼叫`fs.readFile`在104个毫秒和工作转到另一个线程...

另一个100毫秒通过从时间`fs.readFile`已在返回104个毫秒，突然在204个毫秒，我们的回调进入回调队列，并立即去调用堆栈上...

多所有这些重复直到的`file2`和`file3`被印刷在308次和309次分别毫秒。

承诺

当 Promise 最终作为 JavaScript 的标准引入时，引起了很大的轰动。那是在正式的 ES6 标准的时候，也是大张旗鼓地宣布的。随着 ES6 添加到语言中，人们开始感到自豪和大胆。几乎达到了他们的 Java 同行的程度。不知道是`class`关键字的原因还是其他因素...

这里需要注意的是，除了使用`Promise`ES6 自带的对象之外，本文中的所有示例都将遵循 JavaScript 的 ES5 标准。

让我们用 Promise 重写一个相当于示例 *1* 的代码。

示例 3

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

new Promise(function(resolve, reject) {
    fs.readFile("./file.txt", {encoding: "ascii"}, function(err, data1) {
        if (!err) {
            resolve(data1);
        } else {
            reject(err);
        }
    });
}).then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new Promise(function(resolve, reject) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            if (!err) {
                resolve(data2);
            } else {
                reject(err);
            }
        });
    });
}, function(err) {
    console.trace(err);
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new Promise(function(resolve, reject) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            if (!err) {
                resolve(data3);
            } else {
                reject(err);
            }
        });
    });
}, function(err) {
    console.trace(err);
}).then(function(data3) {
    console.log(data3);
}, function(err) {
    console.trace(err);
});
```

我不知道我是否做错了什么，但这对我来说看起来更丑陋。当然，美是主观的。“消失”是示例 *1*的缩进，但打印`file3`和`file2`在同一位置的能力也是如此。除非，你`resolve`有一个这样的数组的第三个承诺：`resolve([data2, data3])`。

在示例 *1* 中，我可以轻松打印`data1`并`data3`在最后。要在示例 *3* 中做到这一点，我必须始终将数据推送到一个数组中，我必须在每个新承诺中传递解析。从该数组中提取信息还需要一两行...

该`Promise`构造函数返回具有的对象`then`方法。这个方法有两个参数。第一个是处理成功读取数据的函数。第二个也是函数，不过这个处理传入Promise构造函数的异步函数可能出现的错误。该`then`方法还返回一个新的承诺。在上面的例子中，我们使用返回的promise from `then`to chain *then* actions。

您会看到，在谈论承诺时，参考Dan Streetmentioner 博士的书来处理所有未来可能的代理解析的语法是很方便的。实际上，第一个参数`then`是一个函数，在调用成功时将被执行，`fs.readFile`第二个参数是一个可能出错的函数，当调用`fs.readFile`错误时。怀疑者说，即使其余的都是真的，但事实并非如此，这显然是不可能的。

撇开玩笑不谈，我希望示例尽可能简单，这样我们就可以专注于 promise 发生的事情。因此，我将排除处理异步 API 函数可能出现的错误（*willan on-errored*）的场景。以下是没有错误处理代码的所有三个示例。

示例 4

JavaScript 复制代码

```
var fs = require("fs");

fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
```

```
    var file2 = data1.split(' ')[1] + ".txt";
    fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {

        var file3 = data2.split(' ')[2] + ".txt";
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {

            console.log(data2);
            console.log(data3);
        });
    });
});
```

例 5

JavaScript 复制代码

```
var fs = require("fs");

function next1(err, data1) {
    var file2 = data1.split(' ')[1] + ".txt";
    fs.readFile(file2, {encoding: "ascii"}, next2);
}

function next2(err, data2) {
    var file3 = data2.split(' ')[2] + ".txt";
    fs.readFile(file3, {encoding: "ascii"}, next3);
}

function next3(err, data3) { console.log(data3); }

fs.readFile("file.txt", {encoding: "ascii"}, next1);
```

例 6

JavaScript 复制代码

```
var fs = require("fs");

new Promise(function(resolve) {
    fs.readFile("./file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
}).then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new Promise(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new Promise(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

我们的代码现在是 100% 乐观的。我们不需要`reject`参数。

让我们举一个例子，使用我们作为参数传递给相应`then`方法的已定义函数。这一次，明确地将方法中每个创建的 promise 分配`then`到它自己的变量中。

例 7

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function executor1(resolve) {
    function callback(err, datac) {
        resolve(datac);
    }
    fs.readFile("file.txt", {encoding: "ascii"}, callback);
}

function next1(data) {
    function executor2(resolve) {
        function callback(err, datac) {
            resolve(datac);
        }
        var file2 = data.split(' ')[1] + ".txt";
        fs.readFile(file2, {encoding: "ascii"}, callback);
    }
    return new Promise(executor2);
}

function next2(data) {
    function executor3(resolve) {
        function callback(err, datac) {
            resolve(datac);
        }
        var file3 = data.split(' ')[2] + ".txt";
        fs.readFile(file3, {encoding: "ascii"}, callback);
    }
    return new Promise(executor3);
}

function next3(data) { console.log(data) };

var promise1 = new Promise(executor1);
var promise2 = promise1.then(next1);
var promise3 = promise2.then(next2);
var promise4 = promise3.then(next3);
```

比较示例 *7*和示例 *5*，其中我已相应地更改了函数名称。啊，面向对象编程的暴行……与老派的过程执行相比，为什么人们将next函数塞进一个then方法中会感觉更好?

我之前在解释示例 *1*时提到过，请**注意**与示例 *7***的时间间隔**。利用1个毫秒前行执行同步码的比喻，无论多么复杂的那条线就是你的所有承诺：promise1，promise2，promise3并promise4在4毫秒创建。然后，在5个毫秒，调用堆栈被清空。在相当长的时间内没有什么可以执行的了。

executor1传递给Promise构造函数的函数在其中执行，并触发 Node.js 工作线程与操作系统对话并获取“ *file.txt* ”的文件内容。JavaScript 运行时将返回您在 中callback定义的函数中的数据executor1。从第promise11 行创建的时间到callback函数进入回调队列的时间，会经过我们约定的 100 毫秒。在执行的第二行，promise2从promise1.then我们声明promise1要next1在promise1解析时调用函数的方法返回。

现在callback里面executor1在101运行开始ST毫秒，并依次调用resolve其datac在102次毫秒，这是我们说，我们希望它来执行功能next中，唯一的可执行行next1是return new Promise(executor2)通过我们的模型，因此，该行打算在103执行的RD毫秒。

很明显，从 返回的承诺next1是不可能的promise2。如果你以为，因为promise2在发生2次执行毫秒，并从返回的承诺next1发生在103届执行毫秒的例子*7*。

如果不创建我们的 promise 构造函数，进一步的解释是不可能的。但是，为简单起见，让我们考虑一种仅调用一个异步函数的场景。

例 8

JavaScript 复制代码

```
var fs = require("fs");

fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {

    var file2 = data1.split(' ')[1] + ".txt";
    console.log(file2);
});
```

构造函数

"我无法创造的东西，我不明白。" ——理查德·费曼

例 9

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function P(executor) {
    var _then;

    var resolve = function(data) {
        _then(data);
    }

    this.then = function(next) {
        _then = next;
        return this;
    }

    executor(resolve);
}

var P1 = new P(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

P1.then(function(data1) {
    var file2 = data1.split(' ')[1] + ".txt";
    console.log(file2);
});

//or should I say

function executor1(resolve) {
    function callback(err, data1) {
        resolve(data1);
    }
    fs.readFile("file.txt", {encoding: "ascii"}, callback);
}

function next1(data1) {
    var file2 = data1.split(' ')[1] + ".txt";
    console.log(file2);
}

P11 = new P(executor1);
P11.then(next1);
```

最简单的promise构造函数，没有任何错误检查，不符合任何标准。但是，它有效！

让我们稍微修改一下，为面向对象编程的一些建设性想法腾出空间。那些不喜欢题外话的人可以安全地跳过文章的下一个标题，直接进入 promise 的构造。

为大众编程，而不是为班级编程

> “每一种使用 *class* 来表示类型的语言都是 *Simula* 的后代。*Kristen Nygaard* 和 *Ole-Johan Dahl* 是数学家，他们不考虑类型，但他们理解元素的集合和类，所以他们称之为他们的类型，类。基本上在 *C++* 和 *Simula* 中，类是用户定义的类型。” - 人工智能播客的 Bjarne Stroustrup

例 10

JavaScript 复制代码

```
var fs = require("fs");

function Q(executor) {
    this._then = function() { console.log("dummy") };
    this.foo = function(data) {
        this._then(data);
    }

    this.then = function(next) {
        this._then = next;
        return this;
    }

    executor(this.foo);
}

var Q1 = new Q(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

Q1.then(function(data1) {
    var file2 = data1.split(' ')[1] + ".txt";
    console.log(file2);
});
```

我们有祸了，这东西不管用！它说`this._then`不是一个函数。它肯定不是，除非您定义了一个`_then`在全局范围内命名的函数。

JavaScript 相当实用的特性使这更加明显。你看到被分配给的匿名函数了`this.foo`吗？当运行时执行构造函数时，`Q`它会解释此函数或即时编译它，无论如何。然后，它会将该函数的地址分配给确定新创建的对象`this`的内部`Q`构造，因为它的`foo` 属性。所谓的`foo`方法。

向下`executor`几行，它将执行我们传递给构造函数的函数，正确地将引用传递`this.foo`给`executor`，但该引用只是一个函数的地址。

我定义`executor`函数的方式是使用一个称为`resolve`函数的参数。我可能把这样的说法任何其他方式，例如`bar`，但它会始终代表着相同的功能对象`Q1`的`foo`方法。`foo`将始终是要调用的正确函数，但这次是在错误的**上下文中**。

JavaScript 中关于函数执行的最重要概念之一是上下文。**上下文是 的值**`this`。

您可以将其`this`视为无法直接分配给的特殊变量，但可以使用`Function.prototype`的方法对其进行处理，例如：`bind`、`apply`和`call`。

每当 JavaScript 看到像您试图（*mayan have on-try*）执行对象方法的语法时，它会立即将 的值切换`this`为该对象，然后跳转到该函数的执行。从该对象的方法返回时，将`this`弹出的旧值。

JavaScript 中的 OOP 到此为止。这不是让你在你的数据面前感到赤裸裸吗？很好，因为它应该是这样的。任何人告诉您必须在您和您的数据之间建立拜占庭式的类层次结构，并且您应该委托官僚作风……

如果你认为，“啊哈，我总是说 JS 是假的，但我的基于类的语言不是！”，想想这个。在您的 C++ 和 Java 中，当您调用对象的方法时，实际上是将该对象作为参数传递给函数。

我将使用 ANSI/ISO C++98 来说明我的观点。

例 11

C++ 缩小▲ 复制代码

```
#include <iostream>

class Cat {
public:
    Cat(int inital);
    int itsEnergy();
    void Eat(int e);
private:
    int energy;
};

int Cat::itsEnergy() {
    return this->energy;
}

void Cat::Eat(int e) {
    this->energy += e;
}

Cat::Cat(int inital) {
    this->energy = inital;
}

int main() {

    Cat Tom(5);
    std::cout << Tom.itsEnergy() << std::endl;
    Tom.Eat(11);
    std::cout << Tom.itsEnergy() << std::endl;

    return 0;
}
```

如果您认为您正在向对象发送`Eat`消息`Tom`..

您正在调用一个函数，其在 C 中的真实性质如下所示：`void cat_eat(Cat *this, int e) { this->energy += e; }`

要一窥真相，只需替换`this->energy`为`this.energy`in`Cat`的第一个“方法”`itsEnergy`。你会收到一个错误：

复制代码

```
[error: request for member ‘energy’ in ‘this’, which is of pointer type ‘Cat* const’
(maybe you meant to use ‘->’ ?)]
```

它是一个指向不那么常量的常量指针`Cat`。大小`Tom`等于唯一数据成员的`energy`大小。在我的机器上，它是一个 4 字节的对象，一个`integer`. 我没有将对象放在引号中，因为它确实是一个对象。仅由数据组成的对象。在程序员的术语中，**类是一种数据类型。**

我们在这里处理的是，在最基本的层面上，至少具有命名空间的能力。类的命名空间`Cat`，那的`Dog`，以及是的，`Mammal`和`StrayCat`。所有对象都是数据。

如果您使用虚函数来执行著名的运行时类型多态性，那么您将在对象中嵌入另一个隐藏的数据成员。指向`virtual`受尊重命名空间的函数表的指针。

如果你创建一个多态Cat对象，你virtual从Cat命名空间中嵌入了一个指向函数表的指针，所以当你将它Cat分配给一个Mammal类型指针时，它可以很容易地将映射调用解引用到类的完全命名的virtual函数Cat。

为了使示例 *10*成为一个有效的示例，我们只需要将函数绑定this.foo到Q构造函数中新创建的对象，如下所示：
executor(this.foo.bind(this))

例 12

JavaScript 复制代码

```
var fs = require("fs");

function Q(executor) {
    this._then = function() { console.log("dummy") };
    this.foo = function(data) {
        this._then(data);
    }

    this.then = function(next) {
        this._then = next;
        return this;
    }

    executor(this.foo.bind(this));
}

var Q1 = new Q(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

Q1.then(function(data1) {
    var file2 = data1.split(' ')[1] + ".txt";
    console.log(file2);
});
```

该绑定将创建另一个函数，在该函数中，JavaScript 将调用我们已知的引用为Q1.fooandresolve的函数，但在调用该函数之前，它会玩把 的值切换this为等于的技巧Q1。

这就像这样。

例 13

JavaScript 复制代码

```
var obj = { value: 4, see: function() { console.log(this.value) }}
var fSee = obj.see
obj.see()                   //4
fSee()                      //undefined
var bSee = fSee.bind(obj)
bSee()                      //4
bSee == obj.see             //false
bSee == fSee                //false
obj.see == fSee             //true
```

更多承诺

目的是构建一个 promise 构造函数，它将替换Promise给定基本示例的 ES6构造函数。因此，让我们继续并将所有异步调用添加到该fs.readFile函数。

例 14

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function P(executor) {
    var _then;

    var resolve = function(data) {
        _then(data);
    }

    this.then = function(next) {
        _then = next;
        return this;
    }

    executor(resolve);
}

var P1 = new P(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

P1.then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

它不打印`file3`，而是打印“ *file.txt* ”。`then`方法中创造的所有新“承诺”都烟消云散了。

因为我们一直在返回同一个对象，它的`_then`值被覆盖了 3 次，在最后一次调用`P1`的`then`方法时，它最终成为记录数据的函数。所有这些都发生在`async`回调开始并调用`resolve`函数之前，传递从第一个文件读取的数据。然后`then`调用该方法设置的最后一个函数，仅打印该数据。这个承诺构造函数只有短期记忆。

现在，我们将一步一步地把它推向更远的地方。我们这里问题的最佳数据类型是数组。

我们无法保存代表数据处理中后续步骤的函数，因为我们只有一个保存占位符。现在，使用数组，我们将拥有任意数量的占位符。

通过`then`方法将`resolve`函数按其所需的执行顺序下推到“promise”的数组中，但在通过移位调用函数时将它们从数组中删除。让它们按照塞入的顺序排列。更像是一个队列，而不是一个堆栈。

例 15

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function P(executor) {
    var _then = [];
```

```
    var resolve = function(data) {
        var f = _then.shift();
        f(data);
    }

    this.then = function(next) {
        return this;
    }

    executor(resolve);
}

var P1 = new P(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

P1.then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

这执行了一点然后它出错了，说f不是一个函数。及时将所有传递给thenmethod的函数塞进数组。然后，调用堆栈被清空。此后的一段时间，fs.readFile完成其工作（我们建议的 100 毫秒）并将第一个回调推送到回调队列中。

事件循环将回调从队列移动到调用堆栈。回调以data1参数“ *file.txt* ”的内容开始执行。resolve我们匿名回调中的函数执行时会从_then“promise”数组中取出它的第一个函数（第 0 个元素）。

该函数被分配给f并通过data1. 到现在为止还挺好。第一个填充函数data1用于提取必须读取的文件的下一个名称。它正在创建一个新的“承诺”，使其内部成为新的异步调用以读取file2并将该承诺返回到空气中……

当第二个匿名“承诺”最终完成其异步工作（大约又过了 100 毫秒后）并愉快地将其传递data2给其resolve函数时，该resolve函数正试图从其关联的数组中提取下一个要处理的过程data2，某些事情会中断。

我们只是将程序填充到我们首先创建的名为P1.

解决这个戈尔迪之结的一种方法是告诉所有新创建的“承诺”，这些“承诺”被归还为稀薄的空气，以解决其P1自身的resolve功能。

要么，要么P1告诉新创建的“promises”填充到其_then数组中的下一步是什么。这样，必须调用新创建并返回的“promises”then方法。这将改变代码的结构。我们使用“promise”的方式与我们使用PromiseES6构建的方式。

我们正在采取懦夫的方式告诉新创建的承诺解决这个问题，这是唯一一个知道必须逐步解决整个混乱局面的承诺。

一个小问题是，P构造函数resolve在调用 create 时将其函数嵌入到自己的闭包中P1。为了得到它，我们需要一个在内部定义P并分配给this它的函数，它将返回resolve我们创建的任何对象的地址，以便我们可以传递它。

与其编写这样的函数，不如让我们通过将其分配给构造函数内部来使其resolve成为P1对象的一部分this。我们将把 的引用resolve从闭包移动到对象中。该对象也是该闭包的一部分，但那是另一回事了。

例 16

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function P(executor) {
    var _then = [];

    this.resolve = function(data) {
        var f = _then.shift();
        f(data);
    }

    this.then = function(next) {
        _then.push(next);
        return this;
    }

    executor(this.resolve.bind(this));
}

var P1 = new P(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

P1.then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            P1.resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            P1.resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

这可以完成工作，但您知道这只是一个肮脏的黑客。它有几个问题。

首先，如果我没有将第一个“承诺”返回给变量`P1`，我就不能将它的`resolve`方法传递给那些新创建的“承诺”。

将这些“承诺”付诸于实践也让人感觉很糟糕。虽然，他们做他们的工作。当`async`调用它们内部的函数时，它正在调用正确的`resolve`方法。唯一知道`then`数组内的填充物。

更糟糕的是，你不应该只从一个`then`方法返回相同的承诺。这不是`Promise`ES6 示例中真正在做的事情。示例 *6* 中的每个`then`方法都返回一个新的 Promise，我们通过它的方法将另一个 Promise 链接到它上面，依此类推。该死的，我们的阵法来了……`then`

按书

承诺无处不在。到处。无处...

例 17

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");
```

```
function P(executor) {
    var _then;

    this.resolve = function(data) {
        this.resolvePromise = _then(data);
    }

    this.then = function(next) {
        _then = next;
        this.thenPromise = new P();
        return this.thenPromise;
    }

    if (typeof executor == "function") executor(this.resolve.bind(this));
}

var P1 = new P(function (resolve) {
    fs.readFile("file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
});

P1.then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
            resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

现在的情况更加疯狂。我们有来自两个不同地方的“承诺”。其中一个知道下一步要做什么，另一个负责数据。我们需要缩小差距，拉上拉链。

我们必须获取该`then`方法的参数并将其设置为将来返回的另一个“承诺”以解决数据。

返回的“承诺”`then`无关。我们没有执行人。唯一的工作就是窃取您将要处理（将不会被处理）您未来数据的程序。一个**IF**将做的工作。

让我们看看发生了什么，使用我们的 1ms 和 100ms 模型。首先，`P1`创建`async fs.readFile`将在 100 毫秒内完成的发出请求。一毫秒之后，`P1.then`使用一个函数参数调用方法。该函数将在准备好时处理数据并提取文件名进行第二次`async fs.readFile`调用，返回一个新的“承诺”。

该`then`方法会将处理数据的函数记录到`P1`的`_then`占位符中。

在接下来的毫秒`P1.then`内将产生一个新的“承诺”，这个是一个完整的假人，并返回它。我们将称之为“承诺”，`thenPromise`. 暂时按暂停。

只过去了 4-5 毫秒。我们有2个承诺。我们有一个异步回调在大约 95 毫秒内运行。该回调将`P1.resolve`使用数据调用。这将解决调用`_then`提取数据的函数，做出新的承诺，并返回它。

这个返回的承诺会在哪里流行?

里面`P1.resolve`。我们最好保存这个返回的承诺，因为这是第二次读取文件中的数据将出现的地方。我们将称之为`resolvePromise`.

所以，现在突然之间，`P1`我们持有了两个承诺。拥有未来数据的那个和应该告诉那个承诺如何处理它的数据的那个确切的那个。

*Example 17*读取第二个文件，然后在这个地方用`TypeError`“ `_then`is not a function”破解`this.resolvePromise = _then(data)`。我们错过了一项重要信息。我们必须将`_then`引用从复制`thenPromise`到 的`_then`引用中`resolvePromise`。

让我们先行一步，以节省墨水和纸张。要将其全部压缩，我们还必须将引用复制到`thenPromise.thenPromise`into 中`resolvePromise.thenPromise`。

当它`resolvePromise`自己解析时，它必须知道它自己`thenPromise`来复制它的`_then`引用。

在脚本的运行中，在第一个异步回调被放置在回调队列之前的相对较长的时间里，您只能针对您想要对数据执行的操作呈现单独的过程。没有可能的方法来接触这些数据。这些过程被塞进一个承诺链表中。除了捕获未来数据处理程序的“声明”（可以这么说）之外，所有这些承诺都无关紧要。

无论你链接了多少承诺，一旦你到达终点。调用堆栈将被清空，然后真正的动作开始。

新的承诺开始从另一面出现，就像在镜子中一样，对于每个`async`请求某些东西的函数。这些`async`函数的回调有数据（或错误），但他们不知道如何处理它。我们的工作是将这些承诺与数据纠缠在一起，将其与代码纠缠在一起，就像量子纠缠一样。

我要更改`_then`占位符。现在，它不仅仅是一个存在于闭包中的私有变量。为了`P`更轻松地在构造函数之外操纵它的值，并且由于我对封装并不那么疯狂，我将替换`var _then`为`this.thenFunction`. 这样，我们就不必编写访问器方法。

按磁带播放，让我们将所说的内容翻译成代码。

这篇文章的最终版本的承诺。

例 18

JavaScript 缩小▲ 复制代码

```
var fs = require("fs");

function P(executor) {

    this.resolve = function(data) {
        if (typeof this.thenFunction == "function") {
            this.resolvePromise = this.thenFunction(data);
            if (this.resolvePromise && this.resolvePromise.constructor == P) {
                this.resolvePromise.thenFunction = this.thenPromise.thenFunction;
                this.resolvePromise.thenPromise = this.thenPromise.thenPromise;
            }
        }
    }

    this.then = function(next) {
        this.thenFunction = next;
        this.thenPromise = new P();
        return this.thenPromise;
    }

    if (typeof executor == "function") executor(this.resolve.bind(this));
}

new P(function(resolve) {
    fs.readFile("./file.txt", {encoding: "ascii"}, function(err, data1) {
        resolve(data1);
    });
}).then(function(data1) {
    var file2 = "./" + data1.split(' ')[1] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file2, {encoding: "ascii"}, function(err, data2) {
            resolve(data2);
        });
    });
}).then(function(data2) {
    var file3 = "./" + data2.split(' ')[2] + ".txt";
    return new P(function(resolve) {
        fs.readFile(file3, {encoding: "ascii"}, function(err, data3) {
```

```
            resolve(data3);
        });
    });
}).then(function(data3) {
    console.log(data3);
});
```

我在这里学到的一个教训是，口头语言描述比代码冗长得多。

第二个想法

当你`resolve`之前打电话时会发生什么`then`?

例 19

JavaScript 复制代码

```
var fs = require("fs");

var rFile = new Promise(function(resolve, reject) {
    fs.readFile('./file1.txt', {encoding: 'ascii'}, function(err, data) {
        resolve(data);
    });
});

setTimeout(rFile.then, 3000, function(message) {console.log("message")});
```

在节点 6.2.2 上，3 秒后，我们收到错误：

复制代码

```
timers.js:333
      ontimeout = () => callback.call(timer, arguments[2]);
                                      ^
TypeError: #<Timeout> is not a promise
```

在节点 4.0.0 上，错误是：

复制代码

```
timers.js:89
        first._onTimeout();
              ^
TypeError: [object Object] is not a promise
```

节点 0.12.0：

复制代码

```
timers.js:223
      callback.apply(timer, args);
               ^
TypeError: [object Object] is not a promise
```

有趣的实现细节...

我真的没有看到使用 Promise 与异步回调的好处（尤其是在示例 *2*的形式中）。

一些高级语言专家说这是一个同步点。它肯定是将您之前编写的代码与您未来的输入数据同步 。这与一直以来的编程没有太大区别。

有人说，对于异步回调，您不知道是当前代码将首先完成还是异步请求的数据将首先弹出。好吧，在其他可能在并行线程中执行其程序的高级语言中可能就是这种情况。

在 JavaScript 的运行时模型中，这是不可能的。您当前的代码将首先完成，然后`async`回调可以进入执行堆栈。这是完全同步的。;)

网络上有很多文档美化了如何通过 Promise 重新获得控制权，如何知道事情何时完成……在 JavaScript 中，事情永远不会改变它们完成的方式。他们只会对你隐藏。

我更喜欢具有透明度的语言、库和框架。从高空到低尘。

如果使用 Promise 有好处，那么也必须有另一面。事情就是这样。

`Promise`在 ES6 中需要注意的一件事是“一旦 Promise 被实现或拒绝，相应的处理程序函数（`onFulfilled`或`onRejected`）将被异步调用（在当前线程循环中调度）。” 来自MDN 网络文档。到目前为止，以我的方式说话，完成意味着已解决，唯一提到的处理程序是`onFulfilled`，即传递给`then`构造函数方法的参数函数`P`。

只需使用`setTime`运行时提供的函数即可异步调用函数。大部分`setTime`在另一个线程中，所以无论你传递给它什么，它都会通过回调队列来，有效地使其“异步调用（在当前线程循环中调度）”。如果我没看错，我找不到原因。也许这与 ES2017 添加的语言有关，例如`async`和`await`。

该关键字`async`使函数异步执行，以便在其中标记为`await`进入回调队列的所有表达式/函数后进入回调队列。我是在准备一篇关于`async`&的文章之前做出这个假设的`await`。当我更好地理解事情时，我可以更正这篇文章是一件好事（*mayan on-uderstand re-correcten*）。

真正的 Promise 远不止我在上面的每 16 个示例中编写的内容。另一方面，我希望在阅读本文后，您不仅准备好使用 Promise，而且可以对它们及其怪癖进行推理。

我已经留下了错误案例和方便的`Promise`方法，例如`all`家庭作业。`promise`州旗：`pending`, `rejected`, `fulfilled`… 也用于作业。我发现`promise`s的状态只是它们功能的副产品，没有什么重要的。另一方面，`Promise`像这样`all`的方法很重要并且构造起来很有趣。

快乐编码！

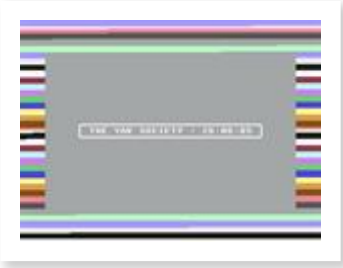
历史

- 2020 年 2 月6日：初始版本

执照

分享

关于作者



马丁ISDN

马其顿共和国

手表该会员

没有提供传记

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

更新?
Pete Lomax Member 10664505 11-Feb-20 13:57

回复：更新?
Martin ISDN 13-Feb-20 17:37

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。

永久链接
广告
隐私
Cookie
使用条款

布局：固定 | 体液