Promise 方式实现 Node.js 应用的实践①

邓森泉[1,2], 杨海波[1]

[1](中国科学院 沈阳计算技术研究所, 沈阳 110168)
[2](中国科学院大学, 北京 100049)

摘 要: Node.js 是目前非常火热的技术之一, 它是运行在服务器端的 JavaScript 执行环境. Node.js 借助 JavaScript 的事件驱动机制加上 V8 高性能引擎, 使得编写高性能 Web 服务轻而易举. Node.js 在处理异步问题时一般采用的是 callback 回调的方式, 但 callback 回调的方式存在 Callback Hell 的问题, 无论是阅读还是调试都很不方便, 甚至无法获取代码的堆栈. 基于 Node.js 平台, 采用 Promise 方式, 编写了一套网络爬虫的应用, 在编写过程中详细的描述了如何使用 Promise 方式处理异步回调问题.

关键词: Node.js; Promise; Web 应用

Web Application Based on Node.js in the Way of Promise

DENG Sen-Quan[1,2], YANG Hai-Bo[1]

[1](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Node.js is one of the most popular technologies at present, and it is the JavaScript execution environment running on the server. With event-driven mechanism Node.js JavaScript plus high-performance V8 engine, it's easy to achieve high-performance Web services. When Node.js deals with the problem of asynchronous, it generally uses callback method, but there are Callback Hell problems in the way of callback. Whether reading or debugging is very inconvenient, it is even impossible to get the code stack. Based on Node.js platform, using Promise method, we realize a Web crawler application. We describe in detail how to use the Promise approach to deal with the problem of asynchronous callback during the application process.

Key words: Node.js; Promise; Web application

Node.js 是建立在 Chrome V8 引擎的 javaScript 运行时之上的平台, 用于构建快速、可扩展的 Web 应用程序. Node.js 采用单线程、事件驱动、非阻塞的 I/O 模型, 这些特性不仅带来了巨大的性能提升, 还减少了多线程程序设计的复杂性, 进而提高了开发效率, 使其轻量又高效. 传统的 Node.js 在处理异步问题时, 一般采用的是 callback 回调的方式. callback 回调存在一个很严重的金字塔问题——大量的回调函数慢慢向右侧屏幕延伸的一种状态.

Promise 是异步编程的一种解决方案, 比传统的解决方案——回调函数和事件, 更合理和强大. 它最早由 javascript 社区提出和实现, 目前最新的 JavaScript 语言标准 ES6 已将其写进了标准中, 统一了用法, 原生提供了 Promise 对象. 借助 Promise 对象, 可以将异步操作以同步操作的流程表达出来, 避免了层层嵌套的回调函数.

本文就是采用 Promise 方式在 Node.js 平台上搭建了一个网络爬虫的应用. 本文首先介绍了 Node.js 平台以及其相关的一些特点和概念, 然后在此基础上, 针对其传统的 callback 的回调方式的"回调地狱"等问题, 引入了 Promise 对象来处理这种异步回调的问题. 通过深入分析 Promise 对象的理论知识以及规范, 将其合理地运用到网络爬虫的应用中去. 最后通过爬取一个课程网站的视频课程信息, 充分展示了 Node.js 平

① 收稿时间:2016-07-26;收到修改稿时间:2016-08-25 [doi: 10.15888/j.cnki.csa.005700]

台的强大和方便, 以及 Promise 对象在处理异步回调问题上的优越性以及新思路.

1 Node.js平台介绍

Node.js 是一位叫 Ryan Dahl 的程序员发明的. 他的工作是用 C/C++写高性能 Web 服务. 对于高性能, 异步 IO、事件驱动是基本原则, 但是用 C/C++写就太痛苦了. 于是Ryan开始设想用高级语言开发Web服务. 他评估了很多种高级语言, 发现很多语言虽然同时提供了同步 IO 和异步 IO, 但是开发人员一旦用了同步 IO, 他们就再也懒得写异步 IO 了, 所以, 最终, Ryan 瞄向了 JavaScript. 因为 JavaScript 是单线程执行, 根本不能进行同步 IO 操作, 所以, JavaScript 的这一“缺陷”导致了它只能使用异步 IO.

选定了开发语言, 还要有运行时引擎. Ryan 曾考虑过自己写一个, 不过明智地放弃了, 因为 V8 就是开源的 JavaScript 引擎. 让 Google 投资去优化 V8, 我们只管拿过来用就好了.

于是在 2009 年, Ryan 正式推出了基于 JavaScript 语言和 V8 引擎的开源 Web 服务器项目, 命名为 Node.js. Node 第一次把 JavaScript 带入到后端服务器开发, 加上世界上已经有无数的 JavaScript 开发人员, 所以 Node.js 一下子就火了起来.

Node.js 架构如图 1 所示.

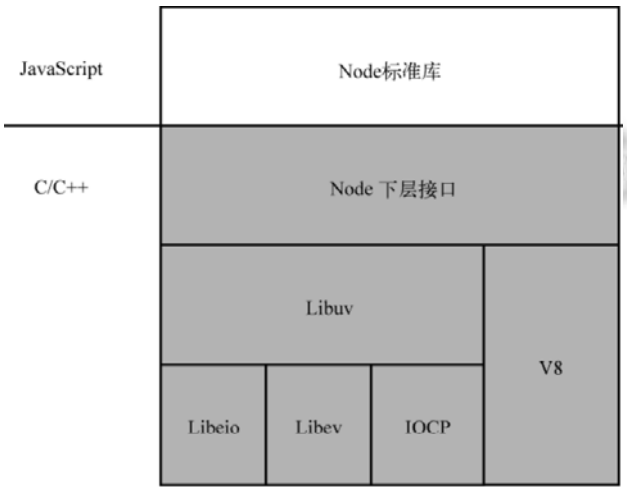

图 1 Node.js 架构

Node.js 主要特点是(1)时间驱动、异步编程; (2)单进程单线程.

1.1 事件驱动、异步编程

事件驱动并不是Node.js专属, 在某些传统语言的网络编程中, 我们会用到回调函数, 比如当 socket 资源达到某种状态时, 注册的回调函数就会执行. Node.js 的设计思想中以事件驱动为核心, 它提供的绝大多数 API 都是基于事件的、异步的风格. 以 Net 模块为例, 其中的 net.Socket 对象就有以下事件: connect、data、end、timeout、drain、error、close 等, 使用 Node.js 的开发人员需要根据自己的业务逻辑注册相应的回调函数. 这些回调函数都是异步执行的, 这意味着虽然在代码结构中, 这些函数看似是依次注册的, 但是它们并不依赖于自身出现的顺序, 而是等待相应的事件触发. 事件驱动、异步编程的设计重要的优势在于, 充分利用了系统资源, 执行代码无须阻塞等待某种操作完成, 有限的资源可以用于其他的任务. 此类设计非常适合于后端的网络服务编程, Node.js 的目标也在于此. 在服务器开发中, 并发的请求处理是个大问题, 阻塞式的函数会导致资源浪费和时间延迟. 通过事件注册、异步函数, 开发人员可以提高资源的利用率, 性能也会改善.

从Node.js提供的支持模块中, 我们可以看到包括文件操作在内的许多函数都是异步执行的, 这和传统语言存在区别, 而且为了方便服务器开发, Node.js 的网络模块特别多, 包括 HTTP、DNS、NET、UDP、HTTPS、TLS 等, 开发人员可以在此基础上快速构建 Web 服务器.

比如搭建一个简单的 http 服务器:

```
//引入http模块
var http = require('http');
//创建一个http服务器
http.createServer(function (req, res) {
    //返回响应头
    res.writeHead(200, {'Content-Type': 'text/plain'});
    //响应结束
    res.end('Hello World\n');
    //在80端口监听
}).listen(80, "127.0.0.1");
```

1.2 单进程单线程

1.2.1 高性能

Node.js 单线程模式避免了传统 php 那样频繁创建、切换线程的花销, 执行速度更快. 而且, 资源占用小, Node.js 在大负荷下对内存占用任然很低.

1.2.2 线程安全

单线程的 node.js 还保证了绝对的线程安全, 不用

担心统一变量同时被多个线程进行读写而造成程序崩溃. 线程安全的同时也解放了开发人员, 免去了多线程编程中忘记对变量加锁或者解锁造成的隐患.

2 Promise

Promise 主要解决 JavaScript 中异步的场景. Promise 是个对象, 同 JavaScript 中其它对象没什么区别, 但同时它也是一个规范, 针对异步操作约定了统一的接口, 表示一个一步操作最终的结果, 以同步的方式来写代码, 执行的操作是异步的, 但是又保证程序的执行顺序是同步的. 这原本是JavaScript社区的一个规范的构想, 现在已经被加入到了 ES6 的语言标准中, Firefox 和 Chrome 等浏览器已经对它进行了实现.

2.1 同步与异步

JS 引擎是单线程的. 这意味着在任何环境中, 只有一段 JS 代码会被执行. 每个函数是一个不可分割的片段或者代码块. 当 JS 引擎开始执行一个函数(比如回调函数)时, 它就会把这个函数执行完, 只有执行完这段代码才会继续执行后面的代码. 这就是 JS 中的同步. Promise 对象的 then()方法就是同步处理每个 Promise 对象.

异步是指在执行一段代码时, 这段代码依赖一些其他的操作或者数据, 这时就不用等待数据或者操作的返回, 直接执行下一段代码, 当有数据或操作返回时再去响应之前的代码, 从而提高代码执行的效率.

2.2 Promise 对象的状态

Promise 对象只有三种状态:

(1) Pending: 初始状态, 进行中.

(2) Resolved(或 Fulfilled): 成功的操作.

(3) Rejected: 失败的操作.

```
var promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);//状态从Pending变为Resolved
  } else {
    reject(error);//状态从Pending变为Rejected
  }
});
```

(1) Promise 对象的状态不受外界影响.

Promise 对象代表一个异步操作, 有三种状态: Pending(进行中)、Resolved(已完成, 又称 Fulfilled)和 Rejected(已失败). 只有异步操作的结果, 可以决定当前是哪一种状态, 任何其他操作都无法改变这个状态.

(2) Promise 对象一旦状态改变, 就不会再变, 任何时候都可以得到这个结果.

Promise 对象的状态改变, 只有两种可能: 从 Pending 变为 Resolved 和从 Pending 变为 Rejected. 只要这两种情况发生, 状态就凝固了, 不会再变了, 会一直保持这个结果. 就算改变已经发生了, 再对 Promise 对象添加回调函数, 也会立即得到这个结果.

2.3 Promise 的核心方法

Promise对象的核心部件是它的then方法, 它的作用是为 Promise 实例添加状态改变时的回调函数. then 方法接受两个回调函数作为参数. 第一个回调函数是 Promise对象的状态变为Resolved时调用, 第二个回调函数是Promise对象的状态变为Rejected时调用. 其中, 第二个函数是可选的, 不一定要提供. 这两个函数都接受 Promise 对象传出的值作为参数.

```
promise.then(function(value) {
  // Resolved状态时调用
}, function(error) {
  // Rejected状态时调用
});
```

Promise 对象另一个核心方法是它的 catch 方法, 用于指定发生错误时的回调函数, 是 then(null, rejection)的别名. catch 方法可以捕捉 promise 实例中的异常还能捕获在它之前太狠方法中发生的异常, 所以在实际的使用中, 多用 catch 方法来取代 then(null, rejection)处理异常.

```
promise.then(function(value) {
  // Resolved状态时调用
}).catch(function(error) {
  // 处理promise和前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

3 爬虫应用设计与实现

3.1 模块加载

新建一个 promise_crawler.js 文件, 首先把需要的相应的模块加载进来.

http模块: 主要用于搭建 HTTP 服务端和客户端, 使用 HTTP 服务器或客户端功能必须调用 http 模块;

bluebird模块: Promise类库(在最新的Node.js里已经引入了 Promise 模块, 可直接使用, 但考虑到兼容性问题, 本例中采用 bluebird 模块);

cheerio 模块: 类似于前端的 jQuery, 能够简单方便地操作装在后台的 html.

代码如下:

```
var http = require('http');
var Promise = require('bluebird');
var cheerio = require('cheerio');
```

3.2 组织数据结构

首先在 chrome 浏览器中打开需要爬取的网页, 同时打开控制台查看网页 html DOM 结构, 分析出所需信息, 组织好数据结构, 然后根据 DOM 结构去获取所需信息. 如图 2 所示.

图 2 网页及 DOM 结构

分析所需数据, 组织好数据结构:

```
// 课程数据
courseData={
    // 课程标题
    courseTitle: courseTitle,
    // 学习人数
    number: number,
    // 课程章节信息
    chaptersData: [{
        // 章节标题
        chapterTitle: '',
        // 章节视频
        videos: [
            // 视频标题
            title: '',
            // 视频id
            id: ''
        ]
    }]
}
```

3.3 Promise 主要流程

本例中完成的主要功能是, 同时爬取一个课程网站的多个页面, 获取相关信息, 然后将数据按照组织好的数据结构打印出来.

核心代码如下:

```
var baseurl = 'http://www.imooc.com/learn/';//课程网站url
var videoIds = [85,124,96,107,110];//课程id
var fetchCourseArray = [];//一个Promise的数组


// 根据url同时爬取多个网页, 将返回的Promise对象添加到数组中
videoIds.forEach(function(id){
    fetchCourseArray.push(getPageAsync(baseurl + id));
});

Promise
    // 调用all()将多个Promise实例包装成一个新Promise实例
    .all(fetchCourseArray)
    // 当all()返回的Promise实例为Resolved状态是调用then()
    .then(function(pages){
        var coursesData = [];

        pages.forEach(function(html){
            // 对每个网页进行数据过滤
            var courses = filterChapters(html);

            coursesData.push(courses);
        });
        // 将所有课程按照学习人数number排序
        coursesData.sort(function(a,b){
            return a.number < b.number;
        });
        // 打印出所有课程信息
        printCourseInfo(coursesData);
    })
    //调用catch方法捕捉Promise实例和then方法中的异常
    .catch(function(e){
        console.log('error: '+e);
    });
```

代码中所用到的 Promise.all 方法用于将多个 Promise 实例, 包装成一个新的 Promise 实例.

```
var p = Promise.all([p1, p2, p3]);
```

该方法接收一个 Promise 对象数组作为参数, p1、p2、p3 都是 Promise 对象的实例.

p 的状态由 p1、p2、p3 决定, 分成两种情况.

(1) 只有 p1、p2、p3 的状态都变成 Resolved, p 的状态才会变成 Resolved, 此时 p1、p2、p3 的返回值组成一个数组, 传递给 p 的回调函数.

(2) 只要 p1、p2、p3 之中有一个被 rejected, p 的状态就变成 Rejected, 此时第一个被 Rejected 的实例的返回值, 会传递给 p 的回调函数.

3.4 相关函数实现

3.4.1 爬取页面 getPageAsync(url)

通过 http 模块的 get 方法爬取页面数据, 最后返回一个 Promise 对象, 方便异步处理.

核心代码如下:

```
// 异步爬取网页
function getPageAsync(url){
    // 函数返回一个包含整个网页内容的Promise对象，用于后面的异步数据处理
    return new Promise(function(resolve,reject){
        console.log('正在爬取: ' + url);
        //异步处理http应答
        http
            .get(url,function(res){
                var html = '';
                // 当有应答数据块chunk返回时，不断的添加到变量html中去
                res.on('data',function(chunk){
                    html += chunk;
                });
                // 当数据块返回完毕时，异步调用Promise对象的resolve方法处理所得数据html
                res.on('end',function(){
                    resolve(html);
                });
            })
            // 在获取数据过程中出错时异步调用Promise对象的reject（）方法处理错误信息
            .on('error',function(e){
                console.log('获取课程数据出错');
                reject(e);
            });
    });
}
```

3.4.2 过滤数据 filterChapters(html)

过滤出每个页面所需的数据，然后按一定的数据结构组织起来.

核心代码如下:

```
function filterChapters(html){
    // 加载cheerio模块处理html页面
    var $ = cheerio.load(html);
    // 数据结构
    var courseData = {
        courseTitle: courseTitle,
        number: number,
        chaptersData: [
        ...
        ]
    };

    // cheerio模块的$变量获取数据结构用所需的信息
    var chapters = $('.chapter');
    var courseTitle = $($('.hd')[0]).find('h2').text();
    var number = parseInt($($('.static-item')[2])
                .find('strong').text(),10);
    ...

    // 返回课程信息
    return courseData;
}
```

3.4.3 打印数据 printCourseInfo(coursesData)

将爬取到的数据，按照组织好的数据结构打印出来.

核心代码如下:

```
// 将爬取到的课程信息按一定格式打印出来
function printCourseInfo(coursesData){
    coursesData.forEach(function(courseData){
        console.log(courseData.number + ' 人学过 '
            + courseData.courseTitle + '\n');
        console.log('### ' + courseData.courseTitle
            + ' ###' + '\n');

        courseData.chaptersData.forEach(function(item){
            var chapterTitle = item.chapterTitle;

            console.log(chapterTitle + '\n');

            item.videos.forEach(function(video){
                console.log('  【'+ video.id + '】  '
                    + video.title + '\n');
            });
        });
    });
}
```

3.4 实验结果

执行 promise_crawler.js 文件，即可看到输出的相关信息如图 3.

```
$ node promise_crawler.js
正在爬取: http://www.imooc.com/learn/85
正在爬取: http://www.imooc.com/learn/124
正在爬取: http://www.imooc.com/learn/96
正在爬取: http://www.imooc.com/learn/107
正在爬取: http://www.imooc.com/learn/110
385548 人学过 Java入门第一季

### Java入门第一季 ###

第1章 Java初体验
  【1430】  1-1 Java简介 (05:49)

  【1459】  1-2 Java开发环境搭建 (07:30)

  【1501】  1-3 使用记事本编写Java程序 (07:00)

  【1474】  1-4 练习题

  【1412】  1-5 使用Eclipse开发Java程序 (08:59)

  【1414】  1-6 MyEclipse的使用简介 (03:53)

  【1475】  1-7 练习题

  【1416】  1-8 程序的移植 (03:08)

  【1422】  1-9 经验技巧分享 (01:52)

  【1423】  1-10 练习题

第2章 变量和常量
  【1176】  2-1 Java中的关键字
```

图 3 输出的相关信息

实验中同爬取了 4 个页面，可以看到，实验结果是按照代码中设定好的数据结构爬取并打印出来的，符合实验预期. Promise 对象是基于异步的方式来处理程序的. 爬取每个页面时，不用等待页面的数据处理

完毕再去爬取下一个页面, 而是无阻塞不间断的去爬取每个页面, 当有异步的数据返回时调用 Promise 对象的 resolve()方法去处理, 出现错误异常时调用 reject()方法去解决. 当有多个 Promise 对象时, 调用 then(onFulfilled)方法, 同步处理每个 Promise 对象, 一旦处理哪个 Promise 对象出错时, 可以立即调用 catch 方法处理异常, 中止程序往下执行, 及时发现错误. 而且 onFulfilled()方法每次返回的是新的 Promise 对象, 这样保证了 then()可以一直链式调用下去, 提高了程序的效率和可靠性.

4 结语

Node.js 作为一门新兴的技术, 打通了前后端的界限. 由于采用事件驱动和无阻塞模型, 可以很方便的构建高效、可扩展的网络应用, 这是 Node.js 最大的一个优点, 同时也是最大的一个缺点, 由于事件驱动和无阻塞模型是建立在 callback 这种回调方式上的, 随着回调的增加, 代码嵌套的层次就会增加, 这样很容易陷入“回调地狱”, 这种代码难以编写, 难以理解而且难以维护.

Promise 对象是解决 Node.js 中异步回调的一种很有效的方式. 借助 Promise 对象, 可以将异步操作以同步操作的流程表达出来, 避免了层层嵌套的回调函数. 在保证异步回调的基础上又实现了多个 promise 对象之间的同步顺序, 使程序能快速高效的执行下去, 给我们的开发带来很大的便利.

参考文献

1 顾宁,刘家茂,柴晓路.Web Services 原理与研发实践.北京:机械工业出版社,2006.
2 朴灵.深入浅出 Node.js.北京:人民邮电出版社,2013.
3 赵昆.改变 Web 开发格局的新技术 node.js.程序员, 2011,(7):124–125.
4 Burnhamt. Javascript 异步编程:设计快速响应的网络应用.北京:人民邮电出版社.
5 Node.js 官方网站.http://www.nodejs.org.
6 Getify. Promise: The Inversion Problem(part 2), https://blog.getify.com/promise-part-2/. [2014-5-19].
7 Brett M. What is node? California: O’Reily Media, 2011.