

Lecture Tips



JavaScript中的异步编程

* 回调函数

- * 使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

* Promise

- * 使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

* generator

- * 可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

* async

- * async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

回调函数

- ❄ 回调函数只是一种存放待办事物的方法。
- ❄ 事件并不是从头到尾按顺序执行的，而是按时间完成的顺序执行。

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto);

function handlePhoto(error, photo) {
  if (error) console.error('Download error!', error);
  else console.log('Download finished', photo);
}

console.log('Download started');
```

使用回调的JavaScript异步编程

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }).bind(this))
        }
      })
    })
  }
})
```

回调地狱

- ✿ 大量的闭包和就地定义的回调函数使代码变得不可读并难以控制的情况被称为回调地狱

```
asyncFunc1(opt, (...args1) => {
  asyncFunc2(opt, (...args2) => {
    asyncFunc3(opt, (...args3) => {
      asyncFunc4(opt, (...args4) => {
        // some operation
      });
    });
  });
});
```

怎样才能避免回调地狱？

遵循下面三个原则

- * 保持代码简洁易懂
- * 模块化
- * 处理每一处错误



保持代码简洁易懂

- ❄ 这里有一些比较混乱的 JavaScript 代码，运行在浏览器中，使用 browser-request 实现 AJAX 请求。

```
var form = document.querySelector('form');
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value;
  request(
    {
      uri: 'http://example.com/upload',
      body: name,
      method: 'POST',
    },
    function (err, response, body) {
      var statusMessage = document.querySelector('.status');
      if (err) return (statusMessage.value = err);
      statusMessage.value = body;
    }
  );
};
```

```
var form = document.querySelector('form');
form.onsubmit = function formSubmit(submitEvent) {
  var name = document.querySelector('input').value;
  request(
    {
      uri: 'http://example.com/upload',
      body: name,
      method: 'POST',
    },
    function postResponse(err, response, body) {
      var statusMessage = document.querySelector('.status');
      if (err) return (statusMessage.value = err);
      statusMessage.value = body;
    }
  );
};
```

获益

- ❄ 函数名可以用来描述函数的作用，使代码更容易阅读
- ❄ 出现错误时，可以获得确切的堆栈跟踪信息，知道错误来自哪个函数而不是「匿名」函数
- ❄ 可以通过更改函数名的方式来变更函数

```
document.querySelector('form').onsubmit = formSubmit;

function formSubmit(submitEvent) {
  var name = document.querySelector('input').value;
  request(
    {
      uri: 'http://example.com/upload',
      body: name,
      method: 'POST',
    },
    postResponse
  );
}

function postResponse(err, response, body) {
  var statusMessage = document.querySelector('.status');
  if (err) return (statusMessage.value = err);
  statusMessage.value = body;
}
```

模块化

- ❄ 任何人都可以（应该）创建模块（类似库）！！！
- ❄ Isaac Schlueter：“编写只做一件事的小模块，然后把它们组装成做更多事情的大模块。如果你能做到这一点，那么你就不会陷入回调地狱。”

✿ 把上面例子中的代码分成几个文件，然后将每个文件转换成模块。

```
formuploader.js
module.exports.submit = formSubmit;

function formSubmit(submitEvent) {
  var name = document.querySelector('input').value;
  request(
    {
      uri: 'http://example.com/upload',
      body: name,
      method: 'POST',
    },
    postResponse
  );
}

function postResponse(err, response, body) {
  var statusMessage = document.querySelector('.status');
  if (err) return (statusMessage.value = err);
  statusMessage.value = body;
}
```

获益

- ❄ 新加入的开发者更容易理解代码——他们不用担心要读完整个 formuploader 函数
- ❄ 函数可以用在其他的地方而不用再写一遍，而且这个函数可以被简单地分享在 Github 或者 NPM 上。



```
var formUploader = require('formuploader');
document.querySelector('form').onsubmit = formUploader.submit;
```

处理每一处错误

❄ 写程序的过程中会遇到各种类型的错误：

- * 由程序员导致的语法错误（通常发生在第一次运行程序的时候），
- * 由程序员导致的运行时错误（代码已经运行了，但是有一个错误导致了程序产生了混乱），由无效的文件权限、硬盘故障、没有网络链接等导致的平台错误。

❄ 涉及后一种错误。

- ❄ 对于回调，最流行的是处理错误的方式是 Node.js 风格，它总是把回调函数的第一个参数作为错误信息返回
- ❄ 把第一个参数作为 error 是一个简单的习惯，可以帮助记住要处理错误。如果错误处理是第二个参数，那么也许会写出类似 function handleFile (file) {} 这样的代码，很容易忽略了错误的处理。
- ❄ 代码检查工具也可以帮助记住去处理回调错误。最简单的一个是 standard。要做的只是在代码所在的文件夹里运行 \$ standard 命令，然后它就会显示代码中每一个未处理错误的回调。

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

fs.readFile('/Does/not/exist', handleFile);

function handleFile(error, file) {
  if (error) return console.error('Uhoh, there was an error', error);
  // otherwise, continue on and use `file` in your code
}
```

小结

- ❄ 不要嵌套函数。给函数命名然后把它们放在程序的最外层。
- ❄ 合理地利用函数提升，把函数放在不显眼的位置。
- ❄ 在每个回调中处理每个错误，可以使用 standard 之类的代码检查工具。
- ❄ 编写可复用的函数，并把它们放入一个模块，这样可以减少理解代码所需的认知负荷。把代码分成小的部分可以帮助处理错误，编写测试，强制你去构建一个稳定和有文档的公共 API，还有利于代码的重构。



创建模块的经验

- ❄ 从把重复使用的代码移动到一个函数里做起
- ❄ 当你写的函数足够大时， 把它们移动到另一个文件中， 然后使用 `module.exports` 语句把接口暴露出来。然后可以使用 `require` 来使用它
- ❄ 如果你写的代码被用于多个项目， 那么应该给它编写 `README`、测试以及 `package.json`， 然后把它发布到 `Github` 和 `NPM`。这种方法有很多好处。
- ❄ 一个好的模块小巧而且专注于解决一个问题
- ❄ 对于模块中的单个文件， `JavaScript` 代码不应该超过 150 行左右
- ❄ 一个模块不应该有超过一层的嵌套文件夹， 里面都是 `JavaScript` 文件， 如果有， 那么这个模块可以做了太多的事。
- ❄ 可以让有经验的程序员给你展示一下好的模块， 知道你对好的模块有印象。如果如果你需要几分钟才能理解程序干了什么， 那么这可能不是一个好的模块。

Promise



*

```
let myPromise = new Promise(function(myResolve, myReject) {  
    // "Producing Code" (可能需要一些时间)  
  
    myResolve(); // 成功时  
    myReject(); // 出错时  
});  
  
// "Consuming Code" (必须等待一个兑现的承诺)  
myPromise.then(  
    function(value) { /* 成功时的代码 */ },  
    function(error) { /* 出错时的代码 */ }  
);
```

❄ 链式调用

```
new Promise(resolve => {
  resolve('Hello')
}).then(result => {
  return ` ${result} world!`;
}).then(result => {
  console.log(result);
});
```

generator

- ❄ ES6 新引入了 Generator 函数。在function关键字后添加*即可将函数变为generator。
- ❄ 可以通过 yield 关键字，把函数的执行流挂起，为改变执行流程提供了可能，从而为异步编程提供解决方案。

```
const gen = function* () {  
  yield 1;  
  yield 2;  
  return 3;  
}
```

❄️ 执行generator将会返回一个遍历器对象，用于遍历generator内部的状态。



```
let g = gen();
g.next(); // { value: 1, done: false }
g.next(); // { value: 2, done: false }
g.next(); // { value: 3, done: true }
g.next(); // { value: undefined, done: true }
```

async/await

- ❄ ECMAScript 2017 引入了 JavaScript 关键字 `async` 和 `await`。`async` 和`await`关键字让我们可以用一种更简洁的方式写出基于Promise的异步行为，而无需刻意地链式调用`promise`。在`async`函数中可以使用`await`语句。`await`后一般是一个Promise对象。

```
const generateAPromise = async () => {
  const result1 = await Promise.resolve('Hello');
  const result2 = await Promise.resolve(` ${result1} world!`);
  console.log(result2); // Hello world!
};

generateAPromise().catch(onerror);
```

跨域

❄ 同源策略SOP

<https://www.a.com:8080/scripts/jtest.js>

协议

域名

端口

❄ 当协议、子域名、主域名、端口号中任意一个不相同时，都算作不同域。

同源策略

❄ 同源策略限制内容有：

- * Cookie、LocalStorage、IndexedDB 等存储性内容
- * DOM 节点
- * AJAX 请求不能发送

❄ 但是有三个标签是允许跨域加载资源：

- *
- * <link href=XXX>
- * <script src=XXX>

处理跨域方法——JSONP

✿ 利用 <script> 元素

```
<script>
var script = document.createElement('script');
script.type = 'text/javascript';

// 传参并指定回调执行函数为onBack
script.src = 'http://www.domain2.com:8080/login?user=admin&callback=onBack';
document.head.appendChild(script);

// 回调执行函数
function onBack(res) {
  alert(JSON.stringify(res));
}
</script>
```

后端node.js代码示例

```
var querystring = require('querystring');
var http = require('http');
var server = http.createServer();

server.on('request', function(req, res) {
  var params = querystring.parse(req.url.split('?')[1]);
  var fn = params.callback;

  // jsonp返回设置
  res.writeHead(200, { 'Content-Type': 'text/javascript' });
  res.write(fn + '(' + JSON.stringify(params) + ')');

  res.end();
});

server.listen('8080');
console.log('Server is running at port 8080...');
```

document.domain + iframe跨域

- ❄ 此方案仅限主域相同，子域不同的跨域应用场景。
- ❄ 实现原理：两个页面都通过js强制设置document.domain为基础主域，就实现了同域。

父窗口：(<http://www.domain.com/a.html>)

```
<iframe id="iframe" src="http://child.domain.com/b.html"></iframe>
<script>
  document.domain = 'domain.com';
  var user = 'admin';
</script>
```

子窗口：(<http://child.domain.com/b.html>)

```
<script>
  document.domain = 'domain.com';
  // 获取父窗口中变量
  alert('get js data from parent ---> ' + window.parent.user);
</script>
```

location.hash + iframe跨域

- ❄ 实现原理： a欲与b跨域相互通信，通过中间页c来实现。三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。
- ❄ 具体实现： A域： a.html -> B域： b.html -> A域： c.html, a与b不同域只能通过hash值单向通信，b与c也不同域也只能单向通信，但c与a同域，所以c可通过parent.parent访问a页面所有对象。



a.html: (<http://www.domain1.com/a.html>)

```
<iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;"></iframe>
<script>
    var iframe = document.getElementById('iframe');

    // 向b.html传hash值
    setTimeout(function() {
        iframe.src = iframe.src + '#user=admin';
    }, 1000);

    // 开放给同域c.html的回调方法
    function onCallback(res) {
        alert('data from c.html ---> ' + res);
    }
</script>
```

```
b.html: (http://www.domain2.com/b.html)
<iframe id="iframe" src="http://www.domain1.com/c.html" style="display:none;"></iframe>
<script>
    var iframe = document.getElementById('iframe');

    // 监听a.html传来的hash值，再传给c.html
    window.onhashchange = function () {
        iframe.src = iframe.src + location.hash;
    };
</script>
```

```
c.html: (http://www.domain1.com/c.html)
<script>
    // 监听b.html传来的hash值
    window.onhashchange = function () {
        // 再通过操作同域a.html的js回调，将结果传回
        window.parent.parent.onCallback('hello: ' + location.hash.replace('#user=', ''));
    };
</script>
```

window.name + iframe跨域

- ❄️ window.name属性的独特之处：name值在不同的页面（甚至不同域名）加载后依旧存在，并且可以支持非常长的 name 值（2MB）。

CORS

- * 跨源资源共享标准新增了一组 HTTP 首部字段，允许服务器声明哪些源站通过浏览器有权限访问哪些资源。另外，规范要求，对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 GET 以外的 HTTP 请求，或者搭配某些 MIME 类型的 POST 请求），浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflight request），从而获知服务端是否允许该跨源请求。服务器确认允许之后，才发起实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 Cookies 和 HTTP 认证相关数据）。
 - * 普通跨域请求：只服务端设置Access-Control-Allow-Origin即可，前端无须设置，若要带cookie请求：前后端都需要设置。
 - * 需注意的是：由于同源策略的限制，所读取的cookie为跨域请求接口所在域的cookie，而非当前页。
- * 目前，所有浏览器都支持该功能(IE8+：IE8/9需要使用XDomainRequest对象来支持CORS))，CORS已经成为主流的跨域解决方案。

```
header("Access-Control-Allow-Origin:*");
header("Access-Control-Allow-Methods:POST,GET");
```

例子

- ✿ 比如说，假如站点 `http://foo.example` 的网页应用想要访问 `http://bar.other` 的资源。`http://foo.example` 的网页中可能包含类似于下面的 JavaScript 代码：

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/public-data/';

function callOtherDomain() {
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

Client Server

GET /doc HTTP/1.1
Origin: foo.example

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *

WebSocket

❄️ WebSocket protocol是HTML5一种新的协议。它实现了浏览器与服务器全双工通信，同时允许跨域通讯，是server push技术的一种很好的实现。

```
<div>user input: <input type="text"></div>
<script src="./socket.io.js"></script>
<script>
var socket = io('http://www.domain2.com:8080');
// 连接成功处理
socket.on('connect', function() {
    // 监听服务端消息
    socket.on('message', function(msg) {
        console.log('data from server: ---> ' + msg);
    });
    // 监听服务端关闭
    socket.on('disconnect', function() {
        console.log('Server socket has closed.');
    });
});

document.getElementsByTagName('input')[0].onblur = function() {
    socket.send(this.value);
};
</script>
```

WebSocket-后台

```
var http = require('http');
var socket = require('socket.io');

// 启http服务
var server = http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-type': 'text/html'
  });
  res.end();
});

server.listen('8080');
console.log('Server is running at port 8080...');

// 监听socket连接
socket.listen(server).on('connection', function(client) {
  // 接收信息
  client.on('message', function(msg) {
    client.send('hello: ' + msg);
    console.log('data from client: ---> ' + msg);
  });

  // 断开处理
  client.on('disconnect', function() {
    console.log('Client socket has closed.');
  });
});
```

postMessage

❄ postMessage是HTML5 XMLHttpRequest Level 2中的API，且是为数不多可以跨域操作的window属性之一，可用于解决以下方面的问题：

- * 页面和其打开的新窗口的数据传递
- * 多窗口之间消息传递
- * 页面与嵌套的iframe消息传递
- * 上面三个场景的跨域数据传递

❄ 用法：postMessage(data,origin)方法接受两个参数

- * data：html5规范支持任意基本类型或可复制的对象，但部分浏览器只支持字符串，所以传参时最好用JSON.stringify()序列化。
- * origin：协议+主机+端口号，也可以设置为“*”，表示可以传递给任意窗口，如果要指定和当前窗口同源的话设置为“/”。

Thanks!!!

