编程语言中的异步性

计算机在设计上是异步的。

异步意味着事情可以独立于主程序流而发生。

在当前的用户计算机中，每个程序都运行于特定的时间段，然后停止执行，以让另一个程序继续执行。 这件事运行得如此之快，以至于无法察觉。 我们以为计算机可以同时运行许多程序，但这是一种错觉（在多处理器计算机上除外）。

程序在内部会使用中断，一种被发送到处理器以获取系统关注的信号。

这里不会深入探讨这个问题，只要记住，程序是异步的且会暂停执行直到需要关注，这使得计算机可以同时执行其他操作。 当程序正在等待来自网络的响应时，则它无法在请求完成之前停止处理器。

通常，编程语言是同步的，有些会在语言或库中提供管理异步性的方法。 默认情况下，C、Java、C＃、PHP、Go、Ruby、Swift 和 Python 都是同步的。 其中一些语言通过使用线程（衍生新的进程）来处理异步操作。

JavaScript

JavaScript 默认情况下是同步的，并且是单线程的。 这意味着代码无法创建新的线程并且不能并行运行。

代码行是依次执行的，例如：

```
const a = 1 const b = 2 const c = a * b console.log(c) doSomething()
```

但是 JavaScript 诞生于浏览器内部，一开始的主要工作是响应用户的操作，例如 `onClick`、`onMouseOver`、`onChange`、`onSubmit` 等。 使用同步的编程模型该如何做到这一点?

答案就在于它的环境。 浏览器通过提供一组可以处理这种功能的 API 来提供了一种实现方式。

更近点，Node.js 引入了非阻塞的 I/O 环境，以将该概念扩展到文件访问、网络调用等。

回调

你不知道用户何时单击按钮。 因此，为点击事件定义了一个事件处理程序。 该事件处理程序会接受一个函数，该函数会在该事件被触发时被调用：

```
document.getElementById('button').addEventListener('click', () => { //被点击 })
```

这就是所谓的回调。

回调是一个简单的函数，会作为值被传给另一个函数，并且仅在事件发生时才被执行。 之所以这样做，是因为 JavaScript 具有顶级的函数，这些函数可以被分配给变量并传给其他函数（称为高阶函数）。

通常会将所有的客户端代码封装在 `window` 对象的 `load` 事件监听器中，其仅在页面准备就绪时才会运行回调函数：

```
window.addEventListener('load', () => { //window 已被加载。 //做需要做的。 })
```

回调无处不在，不仅在 DOM 事件中。

一个常见的示例是使用定时器：

```
setTimeout(() => { // 2 秒之后运行。 }, 2000)
```

XHR 请求也接受回调，在此示例中，会将一个函数分配给一个属性，该属性会在发生特定事件（在该示例中，是请求状态的改变）时被调用：

```
const xhr = new XMLHttpRequest() xhr.onreadystatechange = () => { if (xhr.readyState === 4) { xhr.status === 200 ? console.log(xhr.responseText) : console.error('出错') } } xhr.open('GET', 'http://nodejs.cn') xhr.send()
```

处理回调中的错误

如何处理回调的错误？ 一种非常常见的策略是使用 Node.js 所采用的方式：任何回调函数中的第一个参数为错误对象（即错误优先的回调）。

如果没有错误，则该对象为 null。 如果有错误，则它会包含对该错误的描述以及其他信息。

```
fs.readFile('/文件.json', (err, data) => { if (err !== null) { //处理错误 console.log(err) return } //没有错误，则处理数据。 console.log(data) })
```

回调的问题

回调适用于简单的场景！

但是，每个回调都可以添加嵌套的层级，并且当有很多回调时，代码就会很快变得非常复杂：

```
window.addEventListener('load', () => { document.getElementById('button').addEventListener('click', () => { setTimeout(() => { items.forEach(item => { //你的代码在这里。 }) }, 2000) }) })
```

这只是一个简单的 4 个层级的代码，但还有更多层级的嵌套，这很不好。

该如何解决？

回调的替代方法

从 ES6 开始，JavaScript 引入了一些特性，可以帮助处理异步代码而不涉及使用回调：Promise（ES6）和 Async/Await（ES2017）

了解 JavaScript Promise