一、异步与同步

理解异步与同步

1、在生活中，按照字面量来理解，异步指的是，比如我先吃完苹果再看电视，这是我们生活中理解的异步。同步就是我边吃苹果边看电视。

2、然而，对我们的电脑程序来说，同步与异步的概念恰恰跟我们在生活中的理解完全相反。同步指的是我先吃完苹果，然后再看电视（**执行完一个事件再去执行另一个事件，若上一个事件没执行完，下一个事件就无法执行**）；异步指的是我边吃苹果边看电视（**多个事件可以同时执行，不会发生阻塞**）。理解js中的异步和同步，如果对这两个概念混淆了，你只要想到跟我们生活中异步与同步的理解相反就行了。

谈谈JavaScript的单线程机制（event loop？）

1、**单线程**是JavaScript中一个比较重要的特性。JS的主要运行环境就是我们的客户端（浏览器），因此，单线程的机制也就比较恰当了。比如，假设JS是多线程的，一个线程添加了一个DOM节点，另一个线程删除了这个节点，这时候浏览器就不知道要以哪个线程为准，这样就太混乱了。

2、单线程那就表示事件只能一 个一个轮着执行，前面没执行完后面就无法执行（只能干等着，这样也太低效了）。比如，我用ajax向服务端请求一个数据（假设数据量很大，花费时间很久），这时候就会卡在那里，因为后面的只能等前一个事件把数据成功请求回来再执行接下的代码，这样对用户的体验就太不友好了。这时候 **任务队列** 就出场了。JS中将所有任务分为**同步任务**和 **异步任务**。

3、目前个人的理解是，只有所有的同步任务执行完才会去执行异步任务。常见的异步事件有setTimeout函数，http请求，数据库查询。

> （1）所有任务都在主线程上执行，形成一个执行栈（execution context stack）。
> （2）主线程之外，还存在一个"任务队列"（task queue）。系统把异步任务放到"任务队 列"之中，然后继续执行后续的任务。
> （3）一旦"执行栈"中的所有任务**执行完毕**，系统就会读取"任务队列"。如果这个时候，异步任务已经结束了等待状态，就会从"任务队列"进入执行栈，恢复执行。
> （4）主线程不断重复上面的第三步。
> 详情查看此文章

```
console.log('这里是调试1');
setTimeout(() => {
   console.log('这里是调试2');
},0)
console.log('这里是调试3');
console.log('这里是调试4');
console.log('这里是调试5');
// 输出
这里是调试1
这里是调试3
这里是调试4
这里是调试5
这里是调试2 // setTimeout中的最后面才输出来
```

选择异步还是同步？

1、在JS中默认的方式是同步的，个别是异步的（http请求，setTimeout）。但是现在这里有个场景，有一个前端页面，客户端向服务端发起一个http请求，现在的情况是，只有请求成功返回数据（允许渲染），才向用户展示页面的内容。这里我们如果采用传统的方式，因为http请求属于异步任务，只有主线程的所有任务执行完才能拿到http请求返回的数据。这样就跟我们场景的需求相悖了。为了符合我们场景的需求，我们必须将异步任务（http请求）转为同步任务（拿到http请求的数据接着执行主线程的其他事件）

2、常见的处理异步（将异步变为同步）的方式按照出现时间排序有：回调函数，Promise，async与await。

二、闭包与回调函数

> 1、无论通过何种手段将**内部函数**传递到这个函数所在的词法作用域**以外**，**他都会持有对原始定义作用域的引用**，无论在何处执行这个函数都会使用闭包。
> 2、无论函数在哪里被调用，也无论它如何被调用，它的词法作用域都只由函数被声明时所处的位置决定。
> 3、闭包使得函数可以继续访问定义时的词法作用域。
> 4、只要使用了回调函数，实际上就是在使用闭包。
> ---《你不知道的JavaScript》

1、block（块级）作用域被执行完后，里面的数据就被回收了。

2、作用： 将某个块级作用域里面的变量可以被外部使用。

我们来看一个闭包的案例：

```
function foo() {
    let a = 2;
    function bar() {
        console.log(a);
        a++;
    }
    return bar;
}
let baz = foo();
baz(); // 2
baz(); // 3
```

函数foo()中返回了一个名为bar的函数，接着 `let baz = foo();` 就将bar（...）这个函数赋值给了baz，所以在一定程度上baz相当于bar（...），接着我们就执行baz（在bar的词法作用域以为执行了）（词法作用域： 函数在哪里定义，他的词法作用域就在哪里）；这也就印证了我们前面说的，**将一个内部函数传递到这个函数所在的词法作用域以外执行**，所以就产生了闭包，因为我们在外面获取到了foo函数内部的变量a，同时这也是闭包产生的效果。

3、总结：

- 什么时候产生闭包： 将一个函数传递到这个函数所在的词法作用域以外执行；
- 闭包的作用： 引用某个块级作用域里面变量（持续引用）

4、只要使用了回调函数，实际上就是在使用闭包。下面我们来看一个案例。

```
function wait(message) {
    setTimeout(function timer() {
      console.log(message);
    }, 1000);
}
wait('Hello, World');
```

将一个内部函数（名为timer）传递给setTimeout（...）。timer具有涵盖wait（...）作用域的闭包，因此还保有对变量message的引用。wait（...）执行1000毫秒后，它的内部作用域并不会消失，timer函数依然保有wait（...）作用域的闭包。因为它可以引用到message这个变量。

5、下面我们来看一个稍微复杂点的案例（利用回调函数进行同步传值）：

```
// 模拟获取ajax请求的数据(异步)
function getAjaxData(cb) {
    // 用setTimeout实现异步请求
    setTimeout(function() {
      // 假设data是我们请求得到的数据 我们需要将数据发送给别人
      const data = "请求得到的数据";
      cb(data);
    }, 1000)
}
// 获取ajax请求的响应数据并对数据进行处理
getAjaxData(function handleData(tempData) {
    tempData = tempData + '666';
    console.log(tempData); // 请求得到的数据666
});
```

将handleData(...)作为参数传进getAjaxData(...)中，因此**cb(data)**中的**data**就作为参数传进了handleData（...）中，这样也就达到了传值的作用了。
但是利用回调函数进行传值有个不好的地方，就是如果逻辑复杂点的话，就需要有很多层回调，就形成了回调地狱。。。

6、备注： 关于js中内存回收的问题