在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

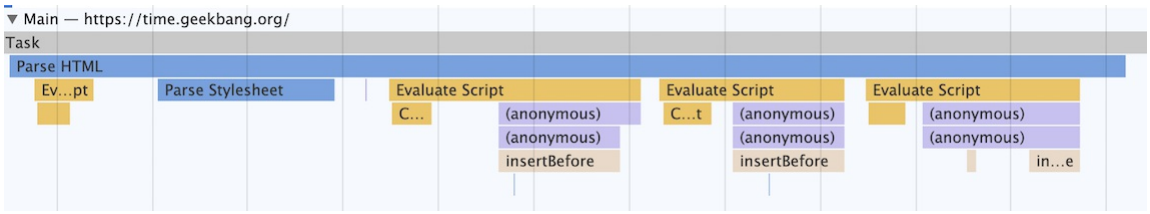
```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

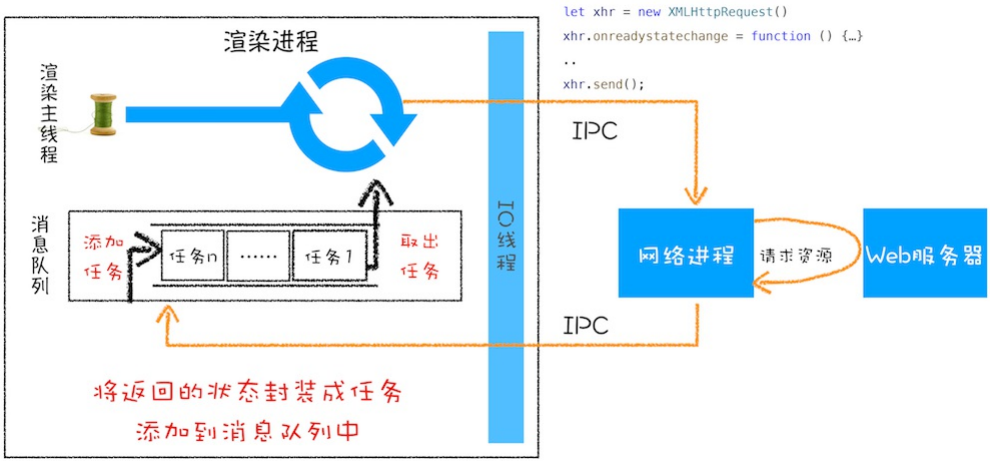
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }

    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：

HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

```
let callback = function(){
    console.log('i am do homework')
```

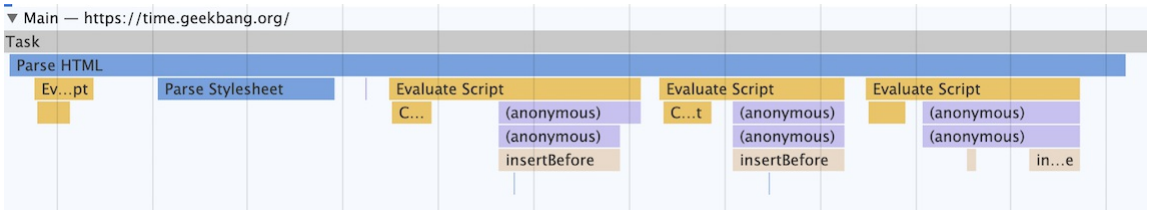
```
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

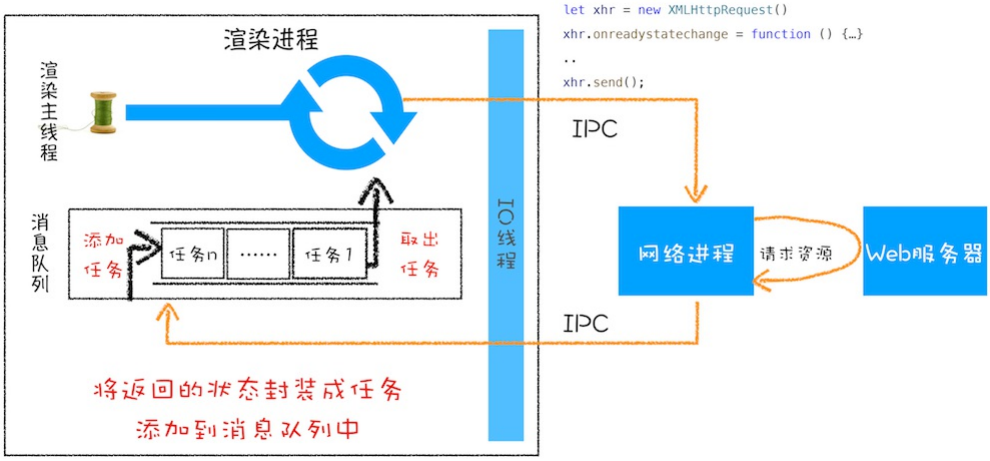
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }
```

```
    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
```

```
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

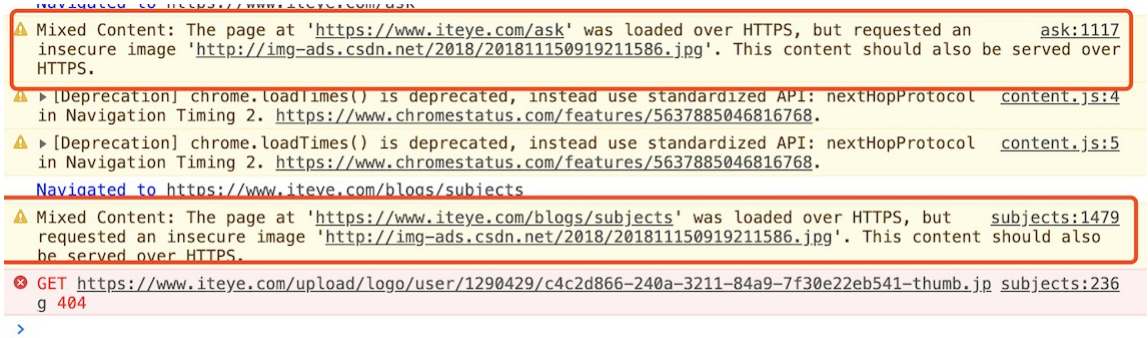
```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：



HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

```
        /**
         * 3:打开请求
         */
        xhr.open('Get', URL, true);//创建一个Get请求,采用异步

        /**
         * 4:配置参数
         */
        xhr.timeout = 3000 //设置xhr请求的超时时间
        xhr.responseType = "text" //设置响应返回的数据格式
        xhr.setRequestHeader("X_TEST","time.geekbang")

        /**
         * 5:发送请求
         */
        xhr.send();
    }
    GetWebData('http://img-ads.csdn.net/2018/201811150919211586.jpg')
    OPENED                                                                    VM252:16
▶Mixed Content: The page at 'https://www.iteye.com/groups' was loaded over HTTPS, but requested an insecure XMLHttpRequest endpoint 'http://img-ads.csdn.net/2018/201811150919211586.jpg'. This request has been blocked; the content must be served over HTTPS.
VM252:52
```

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

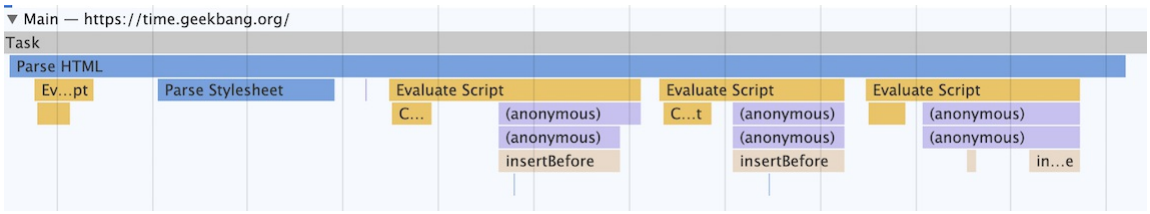
```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

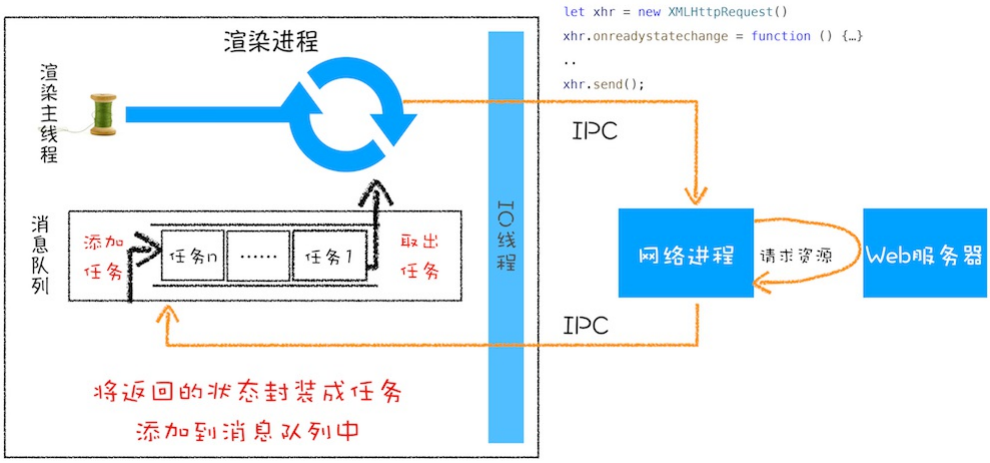
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }

    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：

HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

```
let callback = function(){
    console.log('i am do homework')
```

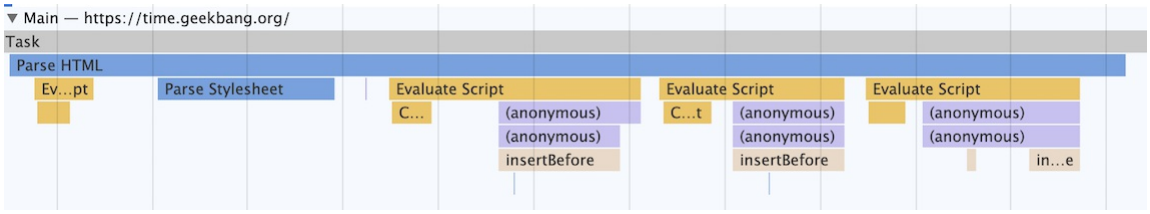
```
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

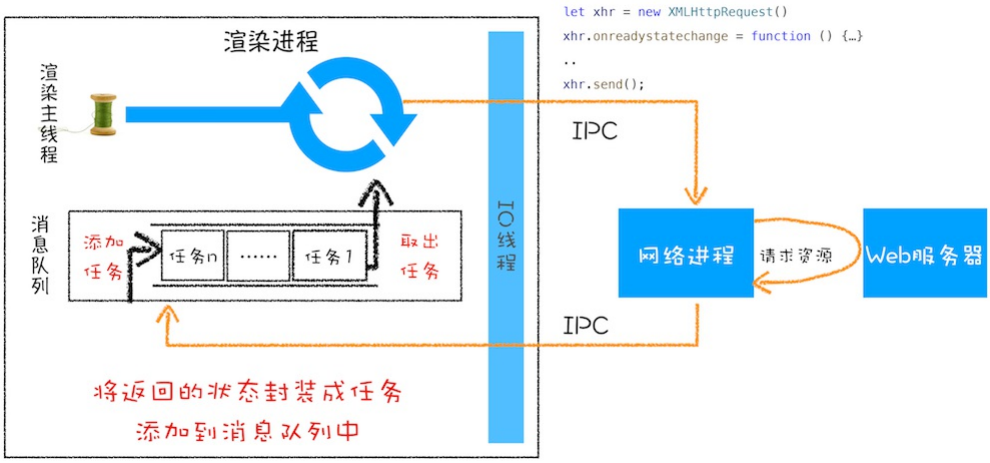
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }
```

```
    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
```

```
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

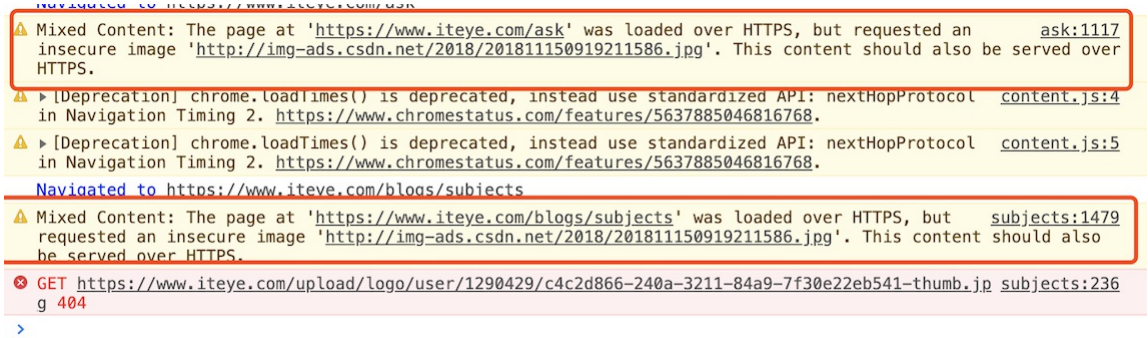
```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：



HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

```
        /**
         * 3:打开请求
         */
        xhr.open('Get', URL, true);//创建一个Get请求,采用异步

        /**
         * 4:配置参数
         */
        xhr.timeout = 3000 //设置xhr请求的超时时间
        xhr.responseType = "text" //设置响应返回的数据格式
        xhr.setRequestHeader("X_TEST","time.geekbang")

        /**
         * 5:发送请求
         */
        xhr.send();
    }
    GetWebData('http://img-ads.csdn.net/2018/201811150919211586.jpg')
    OPENED                                                                    VM252:16
▶Mixed Content: The page at 'https://www.iteye.com/groups' was loaded over HTTPS, but requested an insecure XMLHttpRequest endpoint 'http://img-ads.csdn.net/2018/201811150919211586.jpg'. This request has been blocked; the content must be served over HTTPS.
VM252:52
```

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

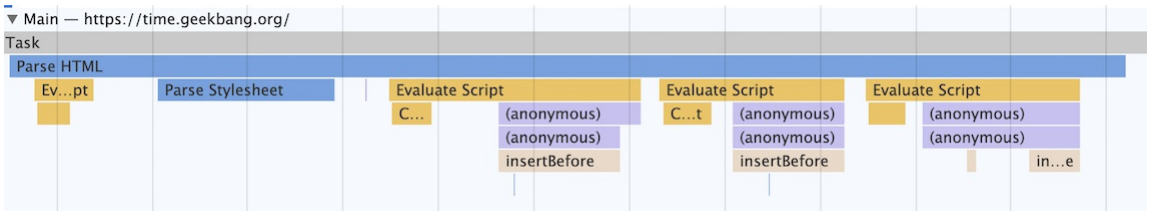
```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

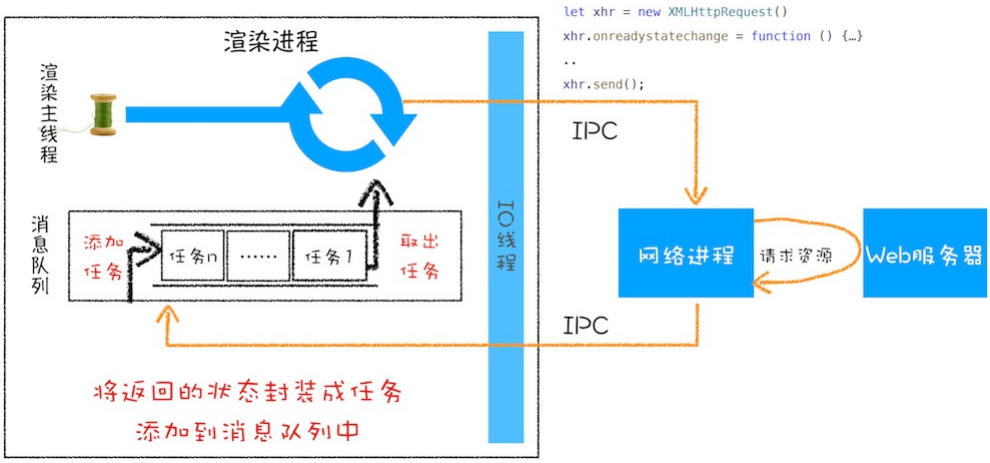
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }

    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：

HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

```
let callback = function(){
    console.log('i am do homework')
```

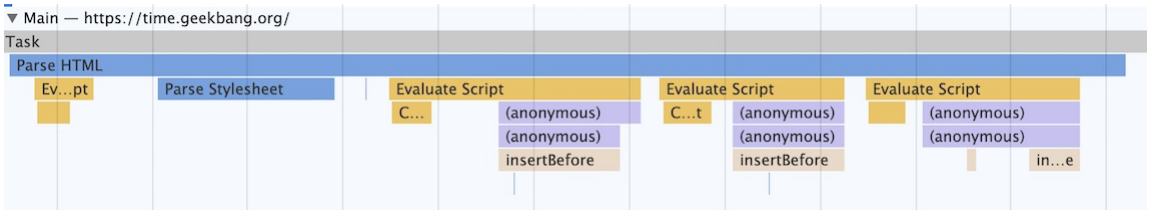
```
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

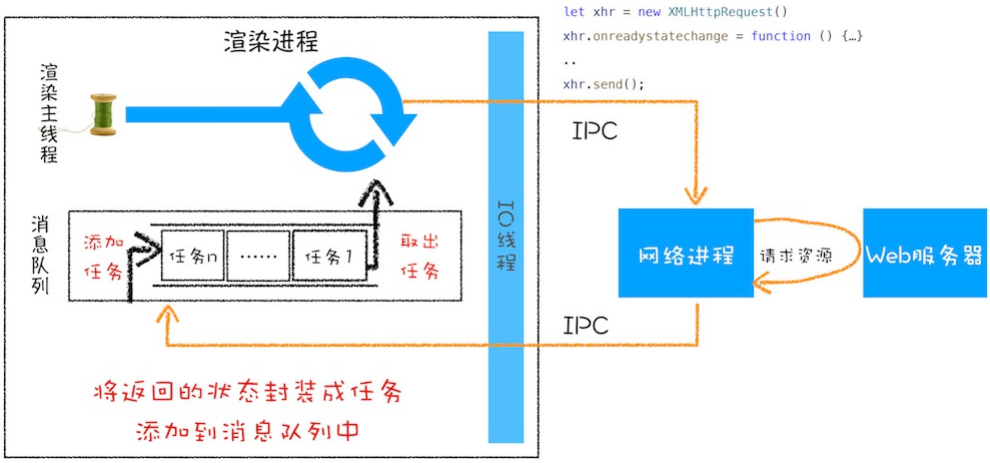
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }
```

```
    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
```

```
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

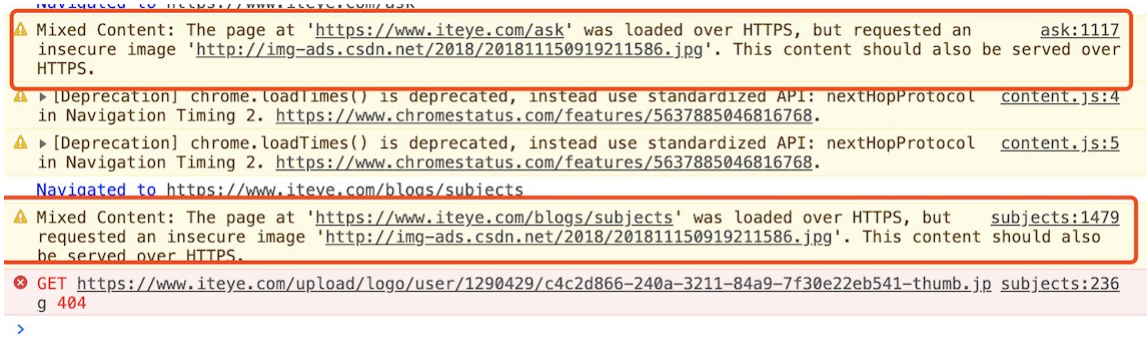
```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：



HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

```
        /**
         * 3:打开请求
         */
        xhr.open('Get', URL, true);//创建一个Get请求,采用异步

        /**
         * 4:配置参数
         */
        xhr.timeout = 3000 //设置xhr请求的超时时间
        xhr.responseType = "text" //设置响应返回的数据格式
        xhr.setRequestHeader("X_TEST","time.geekbang")

        /**
         * 5:发送请求
         */
        xhr.send();
    }
    GetWebData('http://img-ads.csdn.net/2018/201811150919211586.jpg')
    OPENED                                                                    VM252:16
▶Mixed Content: The page at 'https://www.iteye.com/groups' was loaded over HTTPS, but requested an insecure XMLHttpRequest endpoint 'http://img-ads.csdn.net/2018/201811150919211586.jpg'. This request has been blocked; the content must be served over HTTPS.
VM252:52
```

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

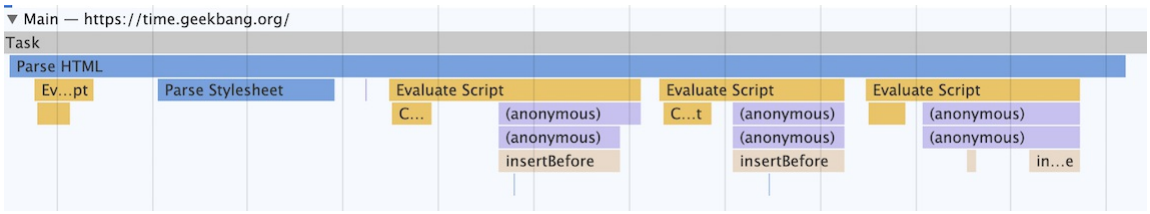
```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

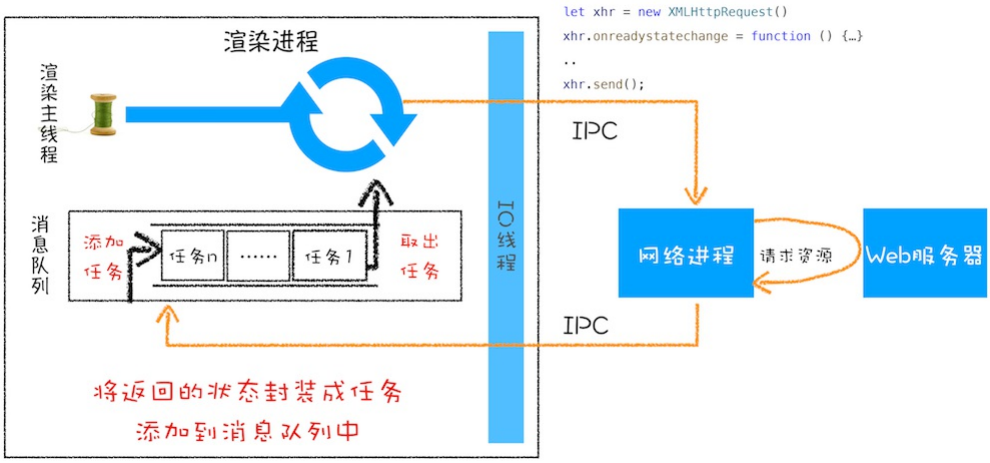
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }

    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：

HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上一篇文章中我们介绍了setTimeout是如何结合渲染进程的循环系统工作的，那本篇文章我们就继续介绍另外一种类型的WebAPI——XMLHttpRequest。

自从网页中引入了JavaScript，我们就可以操作DOM树中任意一个节点，例如隐藏/显示节点、改变颜色、获得或改变文本内容、为元素添加事件响应函数等等， 几乎可以“为所欲为”了。

不过在XMLHttpRequest出现之前，如果服务器数据有更新，依然需要重新刷新整个页面。而XMLHttpRequest提供了从Web服务器获取数据的能力，如果你想要更新某条数据，只需要通过XMLHttpRequest请求服务器提供的接口，就可以获取到服务器的数据，然后再操作DOM来更新页面内容，整个过程只需要更新网页的一部分就可以了，而不用像之前那样还得刷新整个页面，这样既有效率又不会打扰到用户。

关于XMLHttpRequest，本来我是想一带而过的，后来发现这个WebAPI用于教学非常好。首先前面讲了那么网络内容，现在可以通过它把HTTP协议实践一遍；其次，XMLHttpRequest是一个非常典型的WebAPI，通过它来讲解浏览器是如何实现WebAPI的很合适，这对于你理解其他WebAPI也有非常大的帮助，同时在这个过程中我们还可以把一些安全问题给串起来。

但在深入讲解XMLHttpRequest之前，我们得先介绍下**同步回调**和**异步回调**这两个概念，这会帮助你更加深刻地理解WebAPI是怎么工作的。

回调函数 VS 系统调用栈

那什么是回调函数呢（Callback Function）？

将一个函数作为参数传递给另外一个函数，那作为参数的这个函数就是**回调函数**。简化的代码如下所示：

```
let callback = function(){
    console.log('i am do homework')
}
function doWork(cb) {
    console.log('start do work')
    cb()
    console.log('end do work')
}
doWork(callback)
```

在上面示例代码中，我们将一个匿名函数赋值给变量callback，同时将callback作为参数传递给了doWork()函数，这时在函数doWork()中callback就是回调函数。

上面的回调方法有个特点，就是回调函数callback是在主函数doWork返回之前执行的，我们把这个回调过程称为**同步回调**。

既然有同步回调，那肯定也有异步回调。下面我们再来看看异步回调的例子：

```
let callback = function(){
    console.log('i am do homework')
```

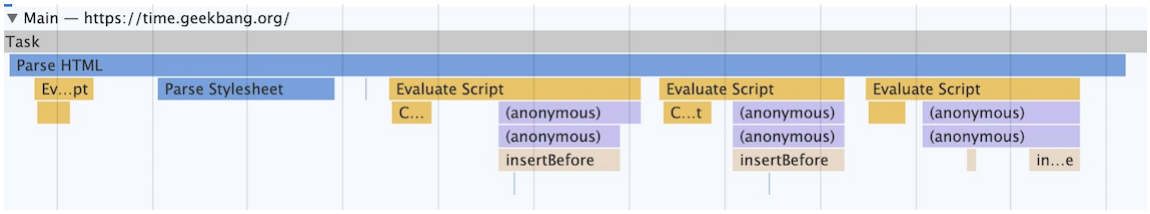
```
}
function doWork(cb) {
    console.log('start do work')
    setTimeout(cb,1000)
    console.log('end do work')
}
doWork(callback)
```

在这个例子中，我们使用了setTimeout函数让callback在doWork函数执行结束后，又延时了1秒再执行，这次callback并没有在主函数doWork内部被调用，我们把这种回调函数在主函数外部执行的过程称为**异步回调**。

现在你应该知道什么是同步回调和异步回调了，那下面我们再深入点，站在消息循环的视角来看看同步回调和异步回调的区别。理解了这些，可以让你从本质上理解什么是回调。

我们还是先来回顾下页面的事件循环系统，通过《15 | 消息队列和事件循环：页面是怎么“活”起来的？》的学习，你应该已经知道浏览器页面是通过事件循环机制来驱动的，每个渲染进程都有一个消息队列，页面主线程按照顺序来执行消息队列中的事件，如执行JavaScript事件、解析DOM事件、计算布局事件、用户输入事件等等，如果页面有新的事件产生，那新的事件将会追加到事件队列的尾部。所以可以说是**消息队列和主线程循环机制保证了页面有条不紊地运行**。

这里还需要补充一点，那就是当循环系统在执行一个任务的时候，都要为这个任务维护一个**系统调用栈**。这个**系统调用栈**类似于JavaScript的调用栈，只不过系统调用栈是Chromium的开发语言C++来维护的，其完整的调用栈信息你可以通过chrome://tracing/来抓取。当然，你也可以通过Performance来抓取它核心的调用信息，如下图所示：



消息循环系统调用栈记录

这幅图记录了一个Parse HTML的任务执行过程，其中黄色的条目表示执行JavaScript的过程，其他颜色的条目表示浏览器内部系统的执行过程。

通过该图你可以看出来，Parse HTML任务在执行过程中会遇到一系列的子过程，比如在解析页面的过程中遇到了JavaScript脚本，那么就暂停解析过程去执行该脚本，等执行完成之后，再恢复解析过程。然后又遇到了样式表，这时候又开始解析样式表……直到整个任务执行完成。

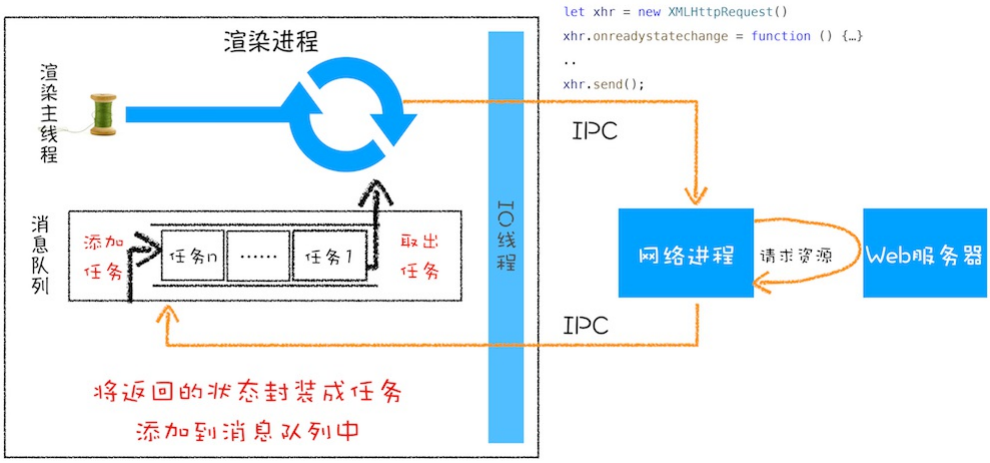
需要说明的是，整个Parse HTML是一个完整的任务，在执行过程中的脚本解析、样式表解析都是该任务的子过程，其下拉的长条就是执行过程中调用栈的信息。

每个任务在执行过程中都有自己的调用栈，那么同步回调就是在当前主函数的上下文中执行回调函数，这个没有太多可讲的。下面我们主要来看看异步回调过程，异步回调是指回调函数在主函数之外执行，一般有两种方式：

- 第一种是把异步函数做成一个任务，添加到信息队列尾部；
- 第二种是把异步函数添加到微任务队列中，这样就可以在当前任务的末尾处执行微任务了。

XMLHttpRequest运作机制

理解了什么是同步回调和异步回调，接下来我们就来分析XMLHttpRequest背后的实现机制，具体工作过程你可以参考下图：



XMLHttpRequest工作流程图

这是XMLHttpRequest的总执行流程图，下面我们就来分析从发起请求到接收数据的完整流程。

我们先从XMLHttpRequest的用法开始，首先看下面这样一段请求代码：

```
function GetWebData(URL){
    /**
     * 1:新建XMLHttpRequest请求对象
     */
    let xhr = new XMLHttpRequest()

    /**
     * 2:注册相关事件回调处理函数
     */
    xhr.onreadystatechange = function () {
        switch(xhr.readyState){
          case 0: //请求未初始化
            console.log("请求未初始化")
            break;
          case 1://OPENED
            console.log("OPENED")
            break;
          case 2://HEADERS_RECEIVED
            console.log("HEADERS_RECEIVED")
            break;
          case 3://LOADING
            console.log("LOADING")
            break;
          case 4://DONE
            if(this.status == 200||this.status == 304){
                console.log(this.responseText);
                }
            console.log("DONE")
            break;
        }
    }
```

```
    xhr.ontimeout = function(e) { console.log('ontimeout') }
    xhr.onerror = function(e) { console.log('onerror') }

    /**
     * 3:打开请求
     */
    xhr.open('Get', URL, true);//创建一个Get请求,采用异步


    /**
     * 4:配置参数
     */
    xhr.timeout = 3000 //设置xhr请求的超时时间
    xhr.responseType = "text" //设置响应返回的数据格式
    xhr.setRequestHeader("X_TEST","time.geekbang")

    /**
     * 5:发送请求
     */
    xhr.send();
}
```

上面是一段利用了XMLHttpRequest来请求数据的代码，再结合上面的流程图，我们可以分析下这段代码是怎么执行的。

第一步：创建XMLHttpRequest对象。

当执行到`let xhr = new XMLHttpRequest()`后，JavaScript会创建一个XMLHttpRequest对象**xhr**，用来执行实际的网络请求操作。

第二步：为xhr对象注册回调函数。

因为网络请求比较耗时，所以要注册回调函数，这样后台任务执行完成之后就会通过调用回调函数来告诉其执行结果。

XMLHttpRequest的回调函数主要有下面几种：

- ontimeout，用来监控超时请求，如果后台请求超时了，该函数会被调用；
- onerror，用来监控出错信息，如果后台请求出错了，该函数会被调用；
- onreadystatechange，用来监控后台请求过程中的状态，比如可以监控到HTTP头加载完成的消息、HTTP响应体消息以及数据加载完成的消息等。

第三步：配置基础的请求信息。

注册好回调事件之后，接下来就需要配置基础的请求信息了，首先要通过open接口配置一些基础的请求信息，包括请求的地址、请求方法（是get还是post）和请求方式（同步还是异步请求）。

然后通过xhr内部属性类配置一些其他可选的请求信息，你可以参考文中示例代码，我们通过`xhr.timeout = 3000`来配置超时时间，也就是说如果请求超过3000毫秒还没有响应，那么这次请求就被判断为失败了。

我们还可以通过`xhr.responseType = "text"`来配置服务器返回的格式，将服务器返回的数据自动转换为自己想要的格式，如果将responseType的值设置为json，那么系统会自动将服务器返回的数据转换为JavaScript对象格式。下面的图表是我列出的一些返回类型的描述：

类型	描述
""	将 responseType 设为空字符串，与设置为"text"相同， 是默认类型（UTF-16字符串）。
"text"	返回的是UTF-16字符串文本。
"json"	response 是一个 JavaScript 对象。
"document"	response 是一个 DOM对象。
"blob"	response 是一个包含二进制数据的 Blob 对象 。
"arraybuffer"	response 是一个包含二进制数据的 JavaScript ArrayBuffer 。

假如你还需要添加自己专用的请求头属性，可以通过xhr.setRequestHeader来添加。

第四步：发起请求。

一切准备就绪之后，就可以调用`xhr.send`来发起网络请求了。你可以对照上面那张请求流程图，可以看到：渲染进程会将请求发送给网络进程，然后网络进程负责资源的下载，等网络进程接收到数据之后，就会利用IPC来通知渲染进程；渲染进程接收到消息之后，会将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务的时候，就会根据相关的状态来调用对应的回调函数。

- 如果网络请求出错了，就会执行xhr.onerror；
- 如果超时了，就会执行xhr.ontimeout；
- 如果是正常的数据接收，就会执行onreadystatechange来反馈相应的状态。

这就是一个完整的XMLHttpRequest请求流程，如果你感兴趣，可以参考下Chromium对XMLHttpRequest的实现，点击这里查看代码。

XMLHttpRequest使用过程中的“坑”

上述过程看似简单，但由于浏览器很多安全策略的限制，所以会导致你在使用过程中踩到非常多的“坑”。

浏览器安全问题是前端工程师避不开的一道坎，通常在使用过程中遇到的“坑”，很大一部分都是由安全策略引起的，不管你喜不喜欢，它都在这里。本来很完美的一个方案，正是由于加了安全限制，导致使用起来非常麻烦。

而你要做的就是去正视这各种的安全问题。也就是说要想更加完美地使用XMLHttpRequest，你就要了解浏览器的安全策略。

下面我们就来看看在使用XMLHttpRequest的过程中所遇到的跨域问题和混合内容问题。

1. 跨域问题

比如在极客邦的官网使用XMLHttpRequest请求极客时间的页面内容，由于极客邦的官网是www.geekbang.org，极客时间的官网是time.geekbang.org，它们不是同一个源，所以就涉及到了跨域（在A站点中去访问不同源的B站点的内容）。默认情况下，跨域请求是不被允许的，你可以看下面的示例代码：

```
var xhr = new XMLHttpRequest()
var url = 'https://time.geekbang.org/'
function handler() {
    switch(xhr.readyState){
        case 0: //请求未初始化
```

```
        console.log("请求未初始化")
        break;
        case 1://OPENED
        console.log("OPENED")
        break;
        case 2://HEADERS_RECEIVED
        console.log("HEADERS_RECEIVED")
        break;
        case 3://LOADING
        console.log("LOADING")
        break;
        case 4://DONE
        if(this.status == 200||this.status == 304){
            console.log(this.responseText);
            }
        console.log("DONE")
        break;
    }
}

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

你可以在控制台测试下。首先通过浏览器打开www.geekbang.org，然后打开控制台，在控制台输入以上示例代码，再执行，会看到请求被Block了。控制台的提示信息如下：

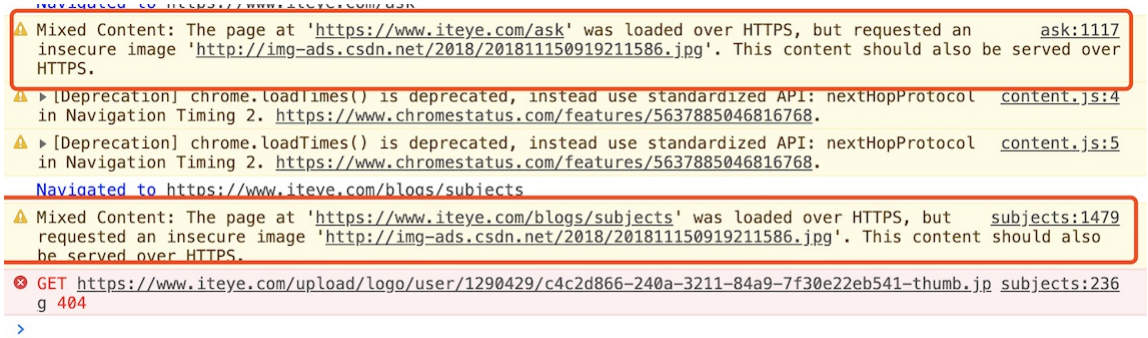
```
Access to XMLHttpRequest at 'https://time.geekbang.org/' from origin 'https://www.geekbang.org' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present
```

因为 www.geekbang.org 和 time.geekbang.com 不属于一个域，所以以上访问就属于跨域访问了，这次访问失败就是由于跨域问题导致的。

2. HTTPS混合内容的问题

了解完跨域问题后，我们再来看看HTTPS的混合内容。HTTPS混合内容是HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等，都属于混合内容。

通常，如果HTTPS请求页面中使用混合内容，浏览器会针对HTTPS混合内容显示警告，用来向用户表明此HTTPS页面包含不安全的资源。比如打开站点 https://www.iteye.com/groups ，可以通过控制台看到混合内容的警告，参考下图：



HTTPS混合内容警告

从上图可以看出，通过HTML文件加载的混合资源，虽然给出警告，但大部分类型还是能加载的。而使用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会阻止此类危险的请求。比如我通过浏览器打开地址 https://www.iteye.com/groups ，然后通过控制台，使用XMLHttpRequest来请求 http://img-ads.csdn.net/2018/201811150919211586.jpg ，这时候请求就会报错，出错信息如下图所示：

```
        /**
         * 3:打开请求
         */
        xhr.open('Get', URL, true);//创建一个Get请求,采用异步

        /**
         * 4:配置参数
         */
        xhr.timeout = 3000 //设置xhr请求的超时时间
        xhr.responseType = "text" //设置响应返回的数据格式
        xhr.setRequestHeader("X_TEST","time.geekbang")

        /**
         * 5:发送请求
         */
        xhr.send();
    }
    GetWebData('http://img-ads.csdn.net/2018/201811150919211586.jpg')
    OPENED                                                                    VM252:16
▶Mixed Content: The page at 'https://www.iteye.com/groups' was loaded over HTTPS, but requested an insecure XMLHttpRequest endpoint 'http://img-ads.csdn.net/2018/201811150919211586.jpg'. This request has been blocked; the content must be served over HTTPS.
VM252:52
```

使用XMLHttpRequest混合资源失效

总结

好了，今天我们就讲到这里，下面我来总结下今天的内容。

首先我们介绍了回调函数和系统调用栈；接下来我们站在循环系统的视角，分析了XMLHttpRequest是怎么工作的；最后又说明了由于一些安全因素的限制，在使用XMLHttpRequest的过程中会遇到跨域问题和混合内容的问题。

本篇文章跨度比较大，不是单纯地讲一个问题，而是将回调类型、循环系统、网络请求和安全问题“串联”起来了。

对比上一篇文章，setTimeout是直接将延迟任务添加到延迟队列中，而XMLHttpRequest发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用IPC的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中。如果你搞懂了setTimeout和XMLHttpRequest的工作机制后，再来理解其他WebAPI就会轻松很多了，因为大部分WebAPI的工作逻辑都是类似的。

思考时间

网络安全很重要，但是又很容易被忽视，因为项目需求很少涉及到基础的Web安全。如果忽视了这些基础安全策略，在开发过程中会处处遇到安全策略挖下的“大坑”，所以对于一名开发者来说，Web安全理论很重要，也必须要学好。

那么今天我留给你一道开放性的思考题：你认为作为一名开发工程师，要如何去高效地学习前端的Web安全理论呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。