```JavaScript
function f() {
    return Promise.resolve('TEST');
}

// asyncF is equivalent to f!
async function asyncF() {
    return 'TEST';
}
```

21.3.2. await

正常情况下，`await` 命令后面是一个 `Promise` 对象，返回该对象的结果。如果不是 `Promise` 对象，就直接返回对应的值

```JavaScript
async function f(){
    // 等同于
    // return 123
    return await 123
}
f().then(v => console.log(v)) // 123
```

不管 `await` 后面跟着的是什么， `await` 都会阻塞后面的代码

```JavaScript
async function fn1 (){
    console.log(1)
    await fn2()
    console.log(2) // 阻塞
}

async function fn2 (){
    console.log('fn2')
}

fn1()
console.log(3)
```

上面的例子中， `await` 会阻塞下面的代码（即加入微任务队列），先执行 `async` 外面的同步代码，同步代码执行完，再回到 `async` 函数中，再执行之前阻塞的代码

所以上述输出结果为：`1`，`fn2`，`3`，`2`

21.4. 流程分析

通过对上面的了解，我们对`JavaScript`对各种场景的执行顺序有了大致的了解

这里直接上代码：

JavaScript | 复制代码

```
async function async1() {
    console.log('async1 start')
    await async2()
    console.log('async1 end')
}
async function async2() {
    console.log('async2')
}
console.log('script start')
setTimeout(function () {
    console.log('settimeout')
})
async1()
new Promise(function (resolve) {
    console.log('promise1')
    resolve()
}).then(function () {
    console.log('promise2')
})
console.log('script end')
```

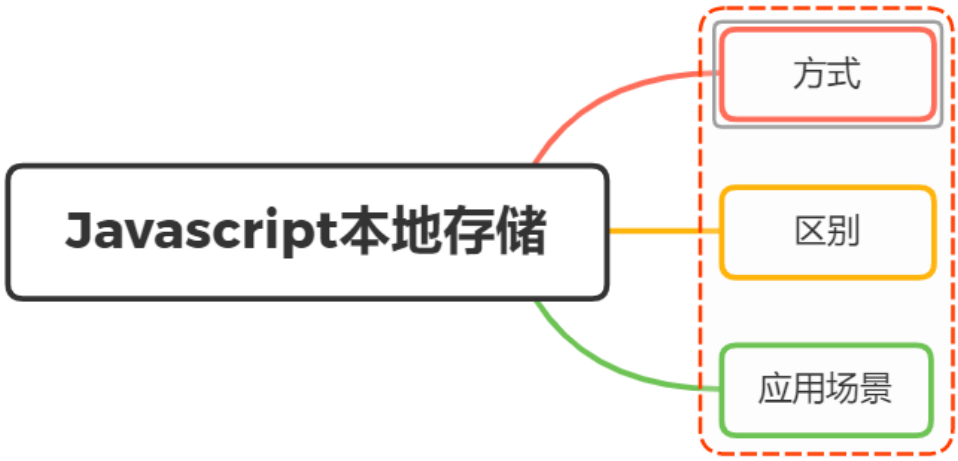
分析过程：

1. 执行整段代码，遇到`console.log('script start')`直接打印结果，输出`script start`
2. 遇到定时器了，它是宏任务，先放着不执行
3. 遇到`async1()`，执行`async1`函数，先打印`async1 start`，下面遇到`await`怎么办？先执行`async2`，打印`async2`，然后阻塞下面代码（即加入微任务列表），跳出去执行同步代码
4. 跳到`new Promise`这里，直接执行，打印`promise1`，下面遇到`.then()`，它是微任务，放到微任务列表等待执行
5. 最后一行直接打印`script end`，现在同步代码执行完了，开始执行微任务，即`await`下面的代码，打印`async1 end`

6. 继续执行下一个微任务，即执行 `then` 的回调，打印 `promise2`
7. 上一个宏任务所有事都做完了，开始下一个宏任务，就是定时器，打印 `settimeout`

所以最后的结果是：`script start` 、`async1 start` 、`async2` 、`promise1` 、`script end` 、`async1 end` 、`promise2` 、`settimeout`

22. Javascript本地存储的方式有哪些？区别及应用场景?



22.1. 方式

`javaScript` 本地缓存的方法我们主要讲述以下四种：

- cookie
- sessionStorage
- localStorage
- indexedDB

22.1.1. cookie

`Cookie`，类型为「小型文本文件」，指某些网站为了辨别用户身份而储存在用户本地终端上的数据。是为了解决 `HTTP` 无状态导致的问题

作为一段一般不超过 4KB 的小型文本数据，它由一个名称（Name）、一个值（Value）和其它几个用于控制 `cookie` 有效期、安全性、使用范围的可选属性组成

但是 `cookie` 在每次请求中都会被发送，如果不使用 `HTTPS` 并对其加密，其保存的信息很容易被窃取，导致安全风险。举个例子，在一些使用 `cookie` 保持登录态的网站上，如果 `cookie` 被窃取，他人很容易利用你的 `cookie` 来假扮成你登录网站

关于 `cookie` 常用的属性如下：

- Expires 用于设置 Cookie 的过期时间

```
JavaScript | 复制代码
1  Expires=Wed, 21 Oct 2015 07:28:00 GMT
```

- Max-Age 用于设置在 Cookie 失效之前需要经过的秒数（优先级比 `Expires` 高）

```
JavaScript | 复制代码
1  Max-Age=604800
```

- `Domain` 指定了 `Cookie` 可以送达的主机名
- `Path` 指定了一个 `URL` 路径，这个路径必须出现在要请求的资源的路径中才可以发送 `Cookie` 首部

```
JavaScript | 复制代码
1  Path=/docs   # /docs/Web/ 下的资源会带 Cookie 首部
```

- 标记为 `Secure` 的 `Cookie` 只应通过被 `HTTPS` 协议加密过的请求发送给服务端

通过上述，我们可以看到 `cookie` 又开始的作用并不是为了缓存而设计出来，只是借用了 `cookie` 的特性实现缓存

关于 `cookie` 的使用如下：

```
JavaScript | 复制代码
1  document.cookie = '名字=值';
```

关于 `cookie` 的修改，首先要确定 `domain` 和 `path` 属性都是相同的才可以，其中有一个不同得时候都会创建出一个新的 `cookie`

```
JavaScript | 复制代码
1  Set-Cookie:name=aa; domain=aa.net; path=/  # 服务端设置
2  document.cookie =name=bb; domain=aa.net; path=/  # 客户端设置
```

最后 `cookie` 的删除，最常用的方法就是给 `cookie` 设置一个过期的事件，这样 `cookie` 过期后会被浏览器删除

22.1.2. localStorage

`HTML5` 新方法，IE8及以上浏览器都兼容

22.1.3. 特点

- 生命周期：持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的
- 存储的信息在同一域中是共享的
- 当本页操作（新增、修改、删除）了 `localStorage` 的时候，本页面不会触发 `storage` 事件,但是别的页面会触发 `storage` 事件。
- 大小：5M（跟浏览器厂商有关系）
- `localStorage` 本质上是对字符串的读取，如果存储内容多的话会消耗内存空间，会导致页面变卡
- 受同源策略的限制

下面再看看关于 `localStorage` 的使用

设置

```javascript
localStorage.setItem('username','cfangxu');
```

获取

```javascript
localStorage.getItem('username')
```

获取键名

```javascript
localStorage.key(0) //获取第一个键名
```

删除

```JavaScript
localStorage.removeItem('username')
```

一次性清除所有存储

```JavaScript
localStorage.clear()
```

`localStorage` 也不是完美的，它有两个缺点：

- 无法像 `Cookie` 一样设置过期时间
- 只能存入字符串，无法直接存对象

```JavaScript
localStorage.setItem('key', {name: 'value'});
console.log(localStorage.getItem('key')); // '[object, Object]'
```

22.1.4. sessionStorage

`sessionStorage` 和 `localStorage` 使用方法基本一致，唯一不同的是生命周期，一旦页面（会话）关闭，`sessionStorage` 将会删除数据

22.1.5. 扩展的前端存储方式

`indexedDB` 是一种低级API，用于客户端存储大量结构化数据(包括, 文件/ blobs)。该API使用索引来实现对该数据的高性能搜索

虽然 `Web Storage` 对于存储较少量的数据很有用，但对于存储更大量的结构化数据来说，这种方法不太有用。`IndexedDB` 提供了一个解决方案

22.1.5.1. 优点：

- 储存量理论上没有上限
- 所有操作都是异步的，相比 `LocalStorage` 同步操作性能更高，尤其是数据量较大时
- 原生支持储存 `JS` 的对象
- 是个正经的数据库，意味着数据库能干的事它都能干

22.1.5.2. 缺点：

- 操作非常繁琐
- 本身有一定门槛

关于 `indexedDB` 的使用基本使用步骤如下：

- 打开数据库并且开始一个事务
- 创建一个 `object store`
- 构建一个请求来执行一些数据库操作，像增加或提取数据等。
- 通过监听正确类型的 `DOM` 事件以等待操作完成。
- 在操作结果上进行一些操作（可以在 `request` 对象中找到）

关于使用 `indexdb` 的使用会比较繁琐，大家可以通过使用 `Godb.js` 库进行缓存，最大化的降低操作难度

22.2. 区别

关于 `cookie` 、 `sessionStorage` 、 `localStorage` 三者的区别主要如下：

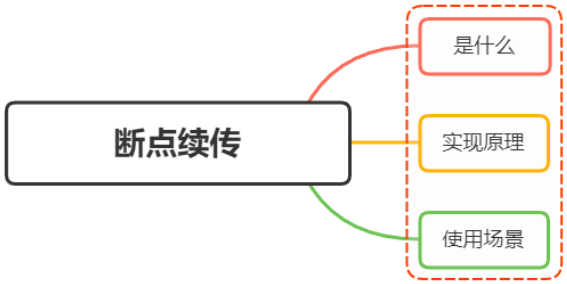
- 存储大小： `cookie` 数据大小不能超过 `4k` ， `sessionStorage` 和 `localStorage` 虽然也有存储大小的限制，但比 `cookie` 大得多，可以达到5M或更大
- 有效时间： `localStorage` 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据； `sessionStorage` 数据在当前浏览器窗口关闭后自动删除； `cookie` 设置的 `cookie` 过期时间之前一直有效，即使窗口或浏览器关闭
- 数据与服务器之间的交互方式， `cookie` 的数据会自动的传递到服务器，服务器端也可以写 `cookie` 到客户端； `sessionStorage` 和 `localStorage` 不会自动把数据发给服务器，仅在本地保存

22.3. 应用场景

在了解了上述的前端的缓存方式后，我们可以看看针对不对场景的使用选择：

- 标记用户与跟踪用户行为的情况，推荐使用 `cookie`
- 适合长期保存在本地的数据（令牌），推荐使用 `localStorage`
- 敏感账号一次性登录，推荐使用 `sessionStorage`
- 存储大量数据的情况、在线文档（富文本编辑器）保存编辑历史的情况，推荐使用 `indexedDB`

23. 大文件上传如何做断点续传?



23.1. 是什么

不管怎样简单的需求，在量级达到一定层次时，都会变得异常复杂

文件上传简单，文件变大就复杂

上传大文件时，以下几个变量会影响我们的用户体验

- 服务器处理数据的能力
- 请求超时
- 网络波动

上传时间会变长，高频次文件上传失败，失败后又需要重新上传等等

为了解决上述问题，我们需要对大文件上传单独处理

这里涉及到分片上传及断点续传两个概念

23.1.1. 分片上传

分片上传，就是将所要上传的文件，按照一定的大小，将整个文件分隔成多个数据块（Part）来进行分片上传

如下图

上传完之后再由服务端对所有上传的文件进行汇总整合成原始的文件

大致流程如下：

1. 将需要上传的文件按照一定的分割规则，分割成相同大小的数据块；
2. 初始化一个分片上传任务，返回本次分片上传唯一标识；
3. 按照一定的策略（串行或并行）发送各个分片数据块；
4. 发送完成后，服务端根据判断数据上传是否完整，如果完整，则进行数据块合成得到原始文件

23.1.2. 断点续传

断点续传指的是在下载或上传时，将下载或上传任务人为的划分为几个部分

每一个部分采用一个线程进行上传或下载，如果碰到网络故障，可以从已经上传或下载的部分开始继续上传下载未完成的部分，而没有必要从头开始上传下载。用户可以节省时间，提高速度

一般实现方式有两种：

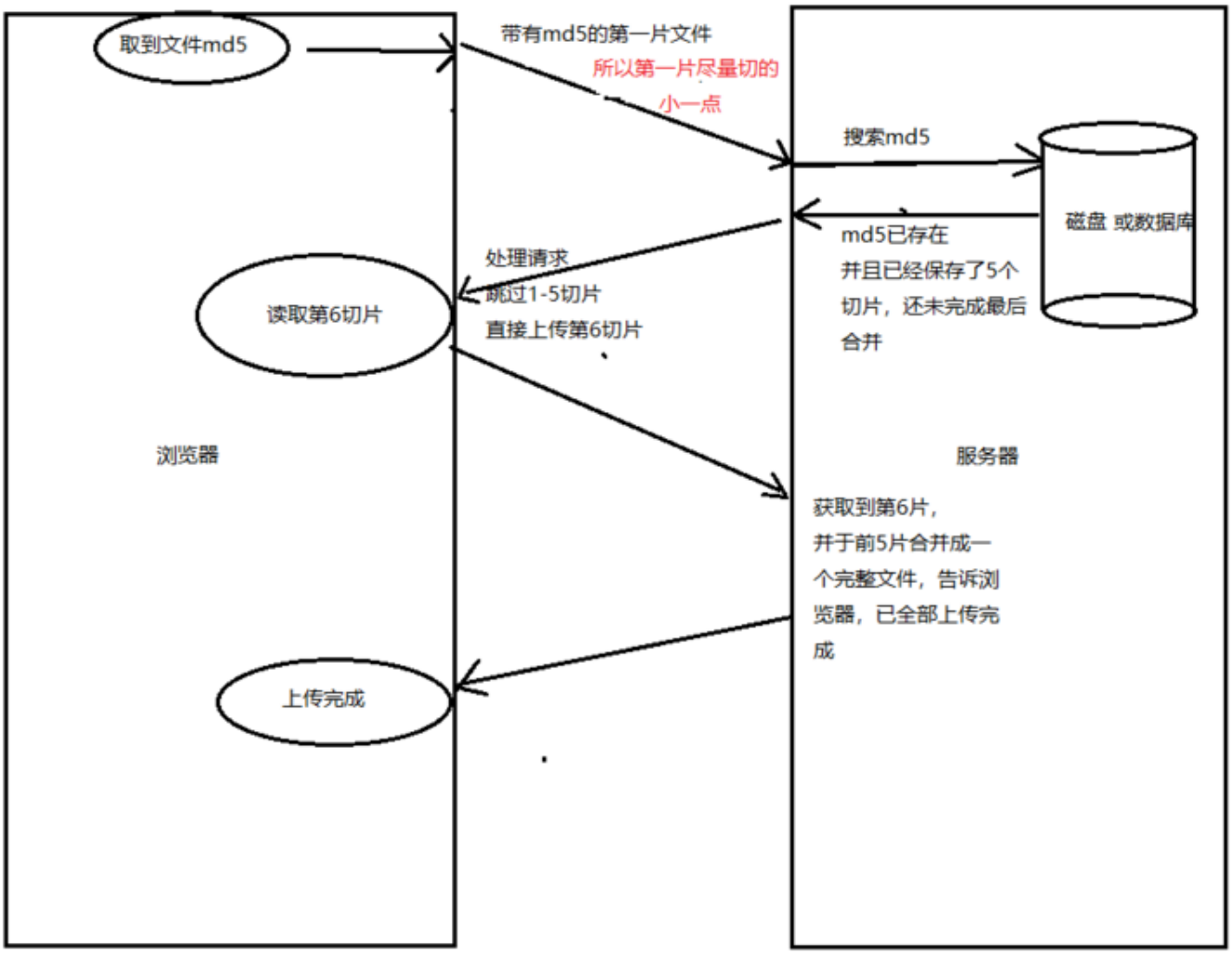
- 服务器端返回，告知从哪开始
- 浏览器端自行处理

上传过程中将文件在服务器写为临时文件，等全部写完了（文件上传完），将此临时文件重命名为正式文件即可

如果中途上传中断过，下次上传的时候根据当前临时文件大小，作为在客户端读取文件的偏移量，从此位置继续读取文件数据块，上传到服务器从此偏移量继续写入文件即可

23.2. 实现思路

整体思路比较简单，拿到文件，保存文件唯一性标识，切割文件，分段上传，每次上传一段，根据唯一性标识判断文件上传进度，直到文件的全部片段上传完毕



下面的内容都是伪代码

读取文件内容：

JavaScript 复制代码

```
const input = document.querySelector('input');
input.addEventListener('change', function() {
    var file = this.files[0];
});
```

可以使用 `md5` 实现文件的唯一性

JavaScript 复制代码

```
const md5code = md5(file);
```

然后开始对文件进行分割

```javascript
var reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.addEventListener("load", function(e) {
    //每10M切割一段,这里只做一个切割演示，实际切割需要循环切割，
    var slice = e.target.result.slice(0, 10*1024*1024);
});
```

h5上传一个（一片）

```javascript
const formdata = new FormData();
formdata.append('0', slice);
//这里是有一个坑的，部分设备无法获取文件名称，和文件类型，这个在最后给出解决方案
formdata.append('filename', file.filename);
var xhr = new XMLHttpRequest();
xhr.addEventListener('load', function() {
    //xhr.responseText
});
xhr.open('POST', '');
xhr.send(formdata);
xhr.addEventListener('progress', updateProgress);
xhr.upload.addEventListener('progress', updateProgress);

function updateProgress(event) {
    if (event.lengthComputable) {
        //进度条
    }
}
```

这里给出常见的图片和视频的文件类型判断

JavaScript | 复制代码

```javascript
function checkFileType(type, file, back) {
/**
* type png jpg mp4 ...
* file input.change=> this.files[0]
* back callback(boolean)
*/
    var args = arguments;
    if (args.length != 3) {
        back(0);
    }
    var type = args[0]; // type = '(png|jpg)' , 'png'
    var file = args[1];
    var back = typeof args[2] == 'function' ? args[2] : function() {};
    if (file.type == '') {
        // 如果系统无法获取文件类型，则读取二进制流，对二进制进行解析文件类型
        var imgType = [
            'ff d8 ff', //jpg
            '89 50 4e', //png

            '0 0 0 14 66 74 79 70 69 73 6F 6D', //mp4
            '0 0 0 18 66 74 79 70 33 67 70 35', //mp4
            '0 0 0 0 66 74 79 70 33 67 70 35', //mp4
            '0 0 0 0 66 74 79 70 4D 53 4E 56', //mp4
            '0 0 0 0 66 74 79 70 69 73 6F 6D', //mp4

            '0 0 0 18 66 74 79 70 6D 70 34 32', //m4v
            '0 0 0 0 66 74 79 70 6D 70 34 32', //m4v

            '0 0 0 14 66 74 79 70 71 74 20 20', //mov
            '0 0 0 0 66 74 79 70 71 74 20 20', //mov
            '0 0 0 0 6D 6F 6F 76', //mov

            '4F 67 67 53 0 02', //ogg
            '1A 45 DF A3', //ogg

            '52 49 46 46 x x x x 41 56 49 20', //avi (RIFF fileSize fileType LIST)(52 49 46 46,DC 6C 57 09,41 56 49 20,4C 49 53 54)
        ];
        var typeName = [
            'jpg',
            'png',
            'mp4',
            'mp4',
            'mp4',
            'mp4',
```

```
            'mp4',
            'm4v',
            'm4v',
            'mov',
            'mov',
            'mov',
            'ogg',
            'ogg',
            'avi',
        ];
        var sliceSize = /png|jpg|jpeg/.test(type) ? 3 : 12;
        var reader = new FileReader();
        reader.readAsArrayBuffer(file);
        reader.addEventListener("load", function(e) {
            var slice = e.target.result.slice(0, sliceSize);
            reader = null;
            if (slice && slice.byteLength == sliceSize) {
                var view = new Uint8Array(slice);
                var arr = [];
                view.forEach(function(v) {
                    arr.push(v.toString(16));
                });
                view = null;
                var idx = arr.join(' ').indexOf(imgType);
                if (idx > -1) {
                    back(typeName[idx]);
                } else {
                    arr = arr.map(function(v) {
                        if (i > 3 && i < 8) {
                            return 'x';
                        }
                        return v;
                    });
                    var idx = arr.join(' ').indexOf(imgType);
                    if (idx > -1) {
                        back(typeName[idx]);
                    } else {
                        back(false);
                    }

                }
            } else {
                back(false);
            }

        });
    } else {
        var type = file.name.match(/\.(\w+)$/)[1];
```

```
            back(type);
        }
    }
```

调用方法如下

```JavaScript
checkFileType('(mov|mp4|avi)',file,function(fileType){
    // fileType = mp4,
    // 如果file的类型不在枚举之列，则返回false
});
```

上面上传文件的一步，可以改成：

```JavaScript
formdata.append('filename', md5code+'.'+fileType);
```

有了切割上传后，也就有了文件唯一标识信息，断点续传变成了后台的一个小小的逻辑判断

后端主要做的内容为：根据前端传给后台的 `md5` 值，到服务器磁盘查找是否有之前未完成的文件合并信息（也就是未完成的半成品文件切片），取到之后根据上传切片的数量，返回数据告诉前端开始从第几节上传

如果想要暂停切片的上传，可以使用 `XMLHttpRequest` 的 `abort` 方法

23.3. 使用场景

- 大文件加速上传：当文件大小超过预期大小时，使用分片上传可实现并行上传多个 Part， 以加快上传速度
- 网络环境较差：建议使用分片上传。当出现上传失败的时候，仅需重传失败的Part
- 流式上传：可以在需要上传的文件大小还不确定的情况下开始上传。这种场景在视频监控等行业应用中比较常见

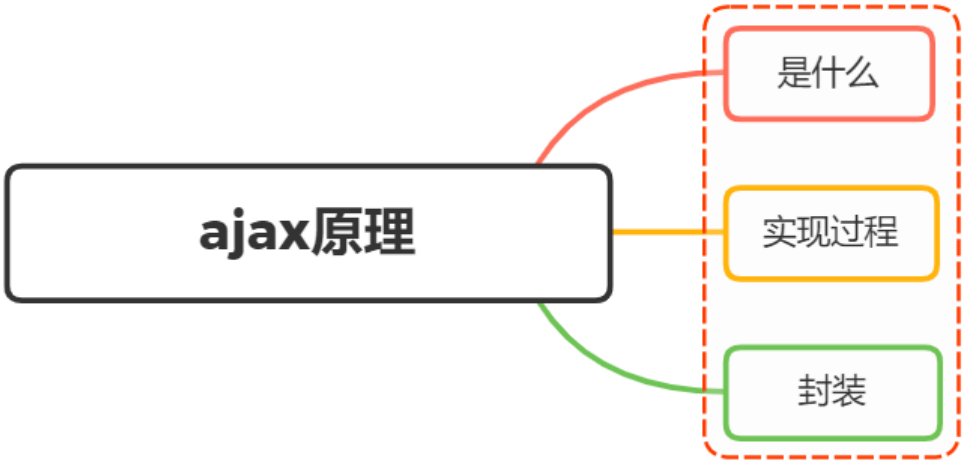
23.4. 小结

当前的伪代码，只是提供一个简单的思路，想要把事情做到极致，我们还需要考虑到更多场景，比如

- 切片上传失败怎么办
- 上传过程中刷新页面怎么办

- 如何进行并行上传
- 切片什么时候按数量切，什么时候按大小切
- 如何结合 Web Worker 处理大文件上传
- 如何实现秒传

24. ajax原理是什么？如何实现?



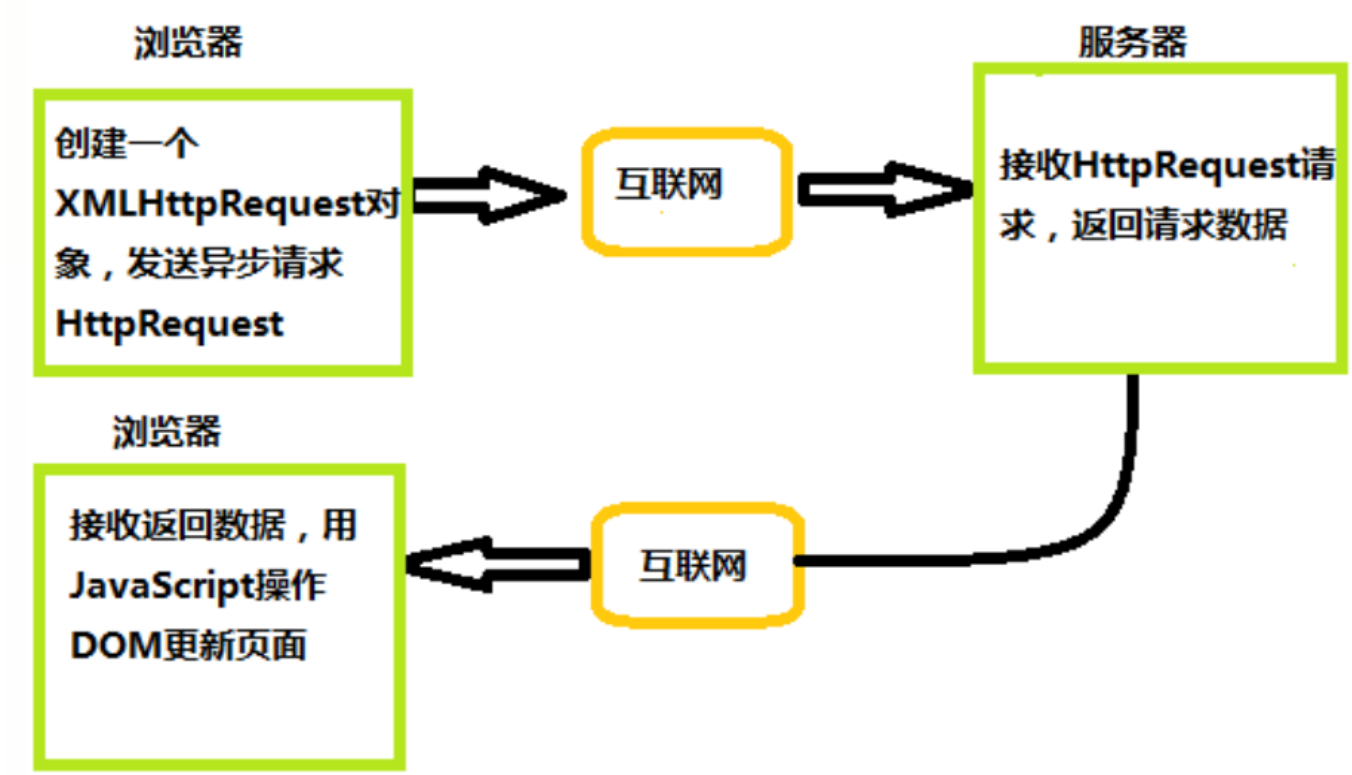
24.1. 是什么

`AJAX` 全称(Async Javascript and XML)

即异步的 `JavaScript` 和 `XML`，是一种创建交互式网页应用的网页开发技术，可以在不重新加载整个网页的情况下，与服务器交换数据，并且更新部分网页

`Ajax` 的原理简单来说通过 `XmlHttpRequest` 对象来向服务器发异步请求，从服务器获得数据，然后用 `JavaScript` 来操作 `DOM` 而更新页面

流程图如下：



下面举个例子：

领导想找小李汇报一下工作，就委托秘书去叫小李，自己就接着做其他事情，直到秘书告诉他小李已经到了，最后小李跟领导汇报工作

`Ajax` 请求数据流程与“领导想找小李汇报一下工作”类似，上述秘书就相当于 `XMLHttpRequest` 对象，领导相当于浏览器，响应数据相当于小李

浏览器可以发送 `HTTP` 请求后，接着做其他事情，等收到 `XHR` 返回来的数据再进行操作

24.2. 实现过程

实现 `Ajax` 异步交互需要服务器逻辑进行配合，需要完成以下步骤：

- 创建 `Ajax` 的核心对象 `XMLHttpRequest` 对象
- 通过 `XMLHttpRequest` 对象的 `open()` 方法与服务端建立连接
- 构建请求所需的数据内容，并通过 `XMLHttpRequest` 对象的 `send()` 方法发送给服务器端
- 通过 `XMLHttpRequest` 对象提供的 `onreadystatechange` 事件监听服务器端你的通信状态
- 接受并处理服务端向客户端响应的数据结果
- 将处理结果更新到 `HTML` 页面中

24.2.1. 创建XMLHttpRequest对象

通过 `XMLHttpRequest()` 构造函数用于初始化一个 `XMLHttpRequest` 实例对象

```JavaScript
const xhr = new XMLHttpRequest();
```

24.2.2. 与服务器建立连接

通过 `XMLHttpRequest` 对象的 `open()` 方法与服务器建立连接

```JavaScript
xhr.open(method, url, [async][, user][, password])
```

参数说明：

- `method`：表示当前的请求方式，常见的有 `GET`、`POST`
- `url`：服务端地址
- `async`：布尔值，表示是否异步执行操作，默认为 `true`
- `user`：可选的用户名用于认证用途；默认为`null
- `password`：可选的密码用于认证用途，默认为`null

24.2.3. 给服务端发送数据

通过 `XMLHttpRequest` 对象的 `send()` 方法，将客户端页面的数据发送给服务端

```JavaScript
xhr.send([body])
```

`body`：在 `XHR` 请求中要发送的数据体，如果不传递数据则为 `null`

如果使用 `GET` 请求发送数据的时候，需要注意如下：

- 将请求数据添加到 `open()` 方法中的 `url` 地址中
- 发送请求数据中的 `send()` 方法中参数设置为 `null`

24.2.4. 绑定onreadystatechange事件

`onreadystatechange` 事件用于监听服务器端的通信状态，主要监听的属性为 `XMLHttpRequest.readyState` ，

关于 `XMLHttpRequest.readyState` 属性有五个状态，如下图显示

值	状态	描述
0	UNSENT(未打开)	open()方法还未被调用
1	OPENED(未发送)	send()方法还未被调用
2	HEADERS_RECEIVED(以获取响应头)	send()方法已经被调用，响应头和响应状态已经返回
3	LOADING(正在下载响应体)	响应体下载中；responseText中已经获取部分数据
4	DONE(请求完成)	整个请求过程已完毕

只要 `readyState` 属性值一变化，就会触发一次 `readystatechange` 事件

`XMLHttpRequest.responseText` 属性用于接收服务器端的响应结果

举个例子：

JavaScript | 复制代码

```
const request = new XMLHttpRequest()
request.onreadystatechange = function(e){
    if(request.readyState === 4){ // 整个请求过程完毕
        if(request.status >= 200 && request.status <= 300){
            console.log(request.responseText) // 服务端返回的结果
        }else if(request.status >=400){
            console.log("错误信息：" + request.status)
        }
    }
}
request.open('POST','http://xxxx')
request.send()
```

24.3. 封装

通过上面对 `XMLHttpRequest` 对象的了解，下面来封装一个简单的 `ajax` 请求

JavaScript | 复制代码

```javascript
//封装一个ajax请求
function ajax(options) {
    //创建XMLHttpRequest对象
    const xhr = new XMLHttpRequest()


    //初始化参数的内容
    options = options || {}
    options.type = (options.type || 'GET').toUpperCase()
    options.dataType = options.dataType || 'json'
    const params = options.data

    //发送请求
    if (options.type === 'GET') {
        xhr.open('GET', options.url + '?' + params, true)
        xhr.send(null)
    } else if (options.type === 'POST') {
        xhr.open('POST', options.url, true)
        xhr.send(params)

    //接收请求
    xhr.onreadystatechange = function () {
        if (xhr.readyState === 4) {
            let status = xhr.status
            if (status >= 200 && status < 300) {
                options.success && options.success(xhr.responseText, xhr.responseXML)
            } else {
                options.fail && options.fail(status)
            }
        }
    }
}
```

使用方式如下