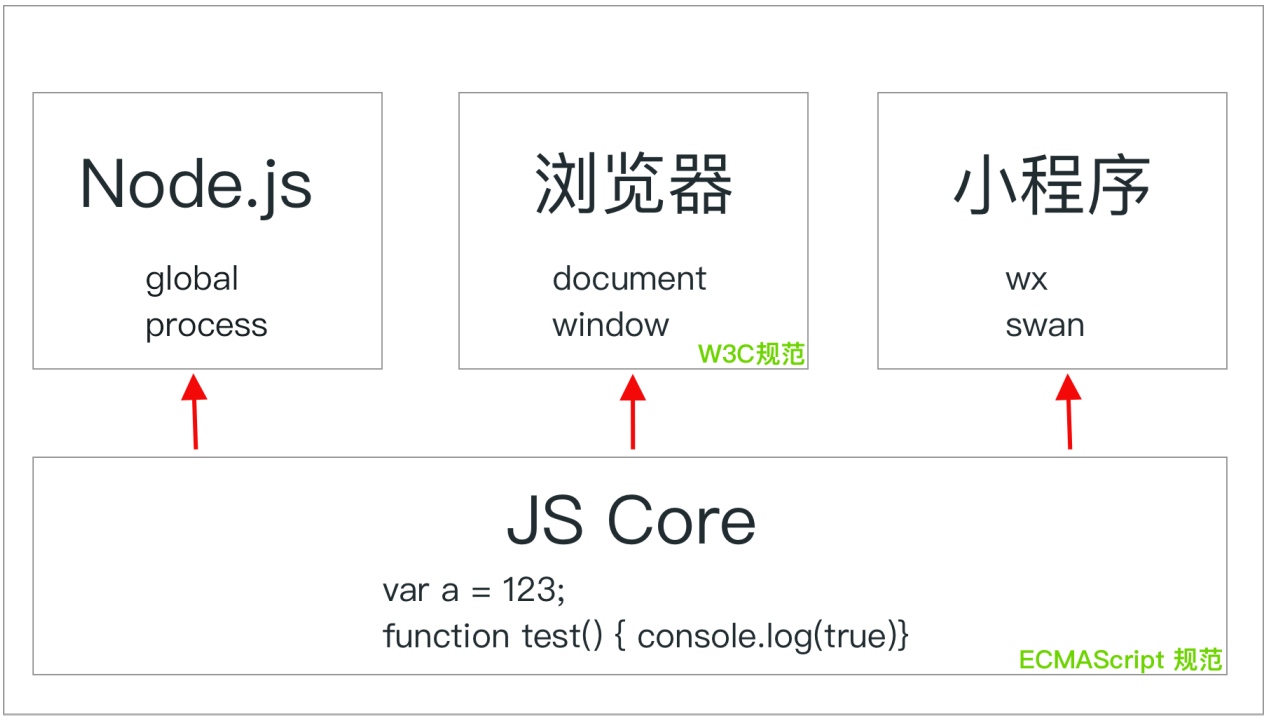
浏览器内置对象/事件/ajax

浏览器是一个 JS 的运行时环境，它基于 JS 解析器的同时，增加了许多环境相关的内容。用一张图表示各个运行环境和 JS 解析器的关系如下：



我们把常见的，能够用 JS 这门语言控制的内容称为一个 JS 的运行环境。常见的运行环境有 Node.js，浏览器，小程序，一些物联网设备等等。所有的运行环境都必须有一个 JS 的解释器，在解释器层面符合 ECMAScript 规范，定义了 JS 本身语言层面的东西比如关键字，语法等等。

在每个环境中，也会基于 JS 开发一些当前环境中的特性，例如 Node.js 中的 global 对象，process 对象；浏览器环境中的 window 对象，document 对象等等，这些属于运行环境在 JS 基础上的内容。

这也就解释了为什么在 node.js 和浏览器中都能使用数组，函数，但是只能在 node.js 使用 require 加载模块，而不能在浏览器端使用的原因，因为 require 是 node.js 特有的运行环境中的内容。

内置对象属性

本小节主要是针对浏览器中的一些常见内置对象，进行学习方法和归纳总结。

Window

window 是在浏览器中代表全局作用域，所有在全局作用域下声明的变量和内容最终都会变成 window 对象下的属性。比如：

```
var num = 123;
console.log(window.num); // 123
```

访问未声明的变量时，如果直接访问则会报错，而如果使用 window 进行访问，就像通过对象访问那样，会返回 undefined。

```
var name = oldName; // 报错
var name2 = window.oldName; // undefined
```

`setTimeout` 和 `setInterval`

setTimeout 和 setInterval 他们都可以接受两个参数，第一个参数是一个**回调函数**，第二个参数是**等待执行的时间**。在等待时间结束之后，就会将回调函数放到 event loop 中进行执行。他们都返回一个 id，传入 clearTimeout 和 clearInterval 能够清除这次的定时操作。

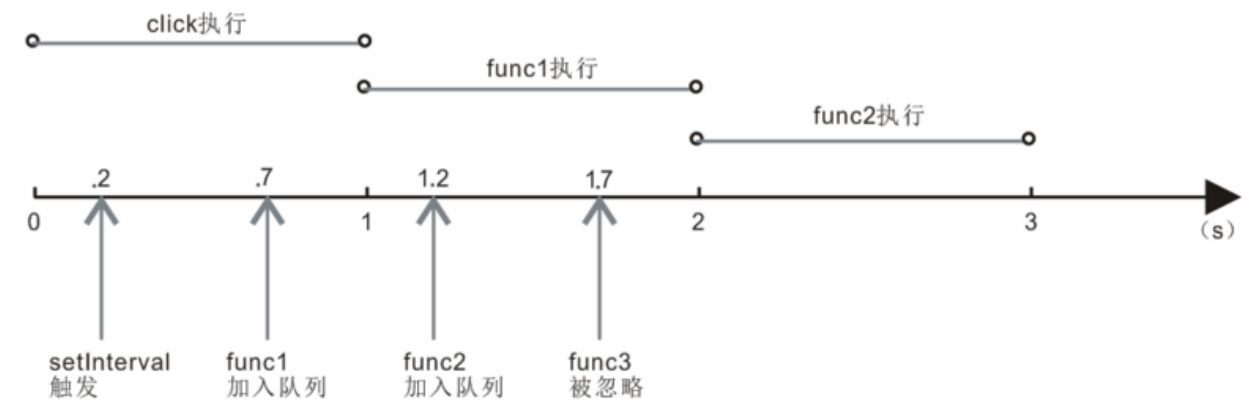
```
var id = setTimeout(function() {
  console.log('hello world');
}, 2000);
clearTimeout(id);
```

可视化工具网站：http://latentflip.com/loupe/

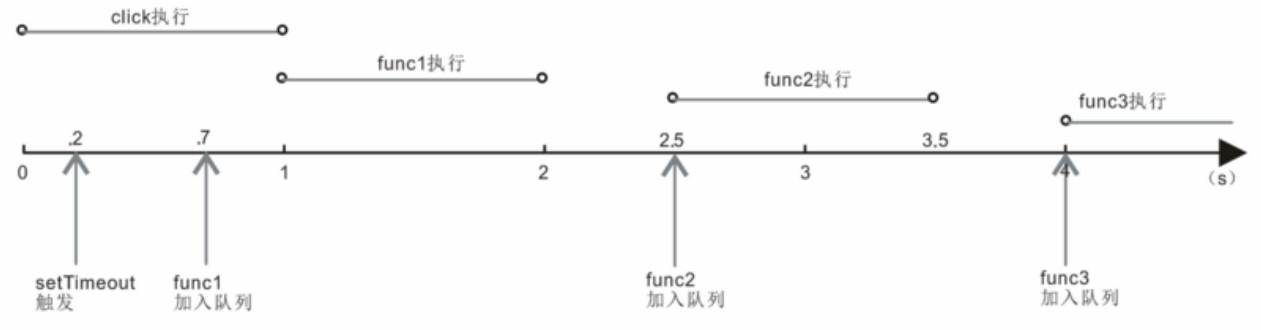
重点：如果此时队列中没有内容，则会立即执行此回调函数，如果此时队列中有内容的话，则会等待内容执行完成之后再执行此函数。（所以即使等待时间结束，也不是立刻执行这个回调函数的！）

因为 setInterval 执行时间的不可确定性，所以大部分时候，我们会使用 setTimeout 来模拟 setInterval。

假设我们点击事件之后会触发 `setInterval(func, 500)`，那么每隔 500ms 就会将 func 放入一次消息队列，如果此时主栈中有其他代码执行的话，就会等待其他代码执行之后再读取消息队列中的函数执行。但对于 setInterval，仅当没有该定时器的任何其他代码实例时，才将定时器代码添加到队列中，所以就会造成某个瞬间有次回调函数没有加进事件队列中去，造成丢帧。



使用 setTimeout 模拟之后的样子，每次执行完成之后再将下次的事件推入事件队列中：



alert，confirm，prompt 等交互相关 API

alert 会弹出一个警告框，而 confirm 和 prompt 则可以与用户交互，confirm 会弹出一个确认框，最终返回 true （用户点击确定）返回 false （用户点击取消）而 prompt 用户则可以输入一段文字，最终返回用户输出的结果。

这里使用了这类 API 之后，会导致页面 JS 停止执行，需要我们格外慎重。

Location

https://baidu.com:8010/api/getSearchResule?foo=bar#hash

```
┌──────────────────────────────────────────────────────────────────────────────────────────────┐
│                                             href                                             │
├──────────┬──┬─────────────────────┬────────────────────┬─────────────────────────────┬────────┤
│ protocol │  │        auth         │        host        │            path             │  hash  │
│          │  │                     ├──────────┬─────────┼──────────┬──────────────────┤        │
│          │  │                     │ hostname │ port    │ pathname │     search       │        │
│          │  │                     │          │         │          ├─┬────────────────┤        │
│          │  │                     │          │         │          │ │    query       │        │
"  https:   //    user   :   pass   @ sub.example.com : 8080   /p/a/t/h  ?  query=string   #hash "
│          │  │          │          │ hostname │ port    │          │                  │        │
│          │  │          │          ├──────────┴─────────┤          │                  │        │
│ protocol │  │ username │ password │        host        │          │                  │        │
├──────────┴──┼──────────┴──────────┼────────────────────┤          │                  │        │
│   origin    │                     │       origin       │ pathname │     search       │ hash   │
├─────────────┴─────────────────────┴────────────────────┴──────────┴──────────────────┴────────┤
│                                             href                                             │
└──────────────────────────────────────────────────────────────────────────────────────────────┘
```

属性

- hash：返回一个URL的锚部分。
- host：返回一个URL的主机名和端口
- hostname：返回URL的主机名
- href：当前 url
- pathname： 返回的URL路径名。
- port： 返回一个URL服务器使用的端口号
- protocol：返回一个URL协议
- search：返回一个URL的查询部分

方法

- reload：重新载入当前页面
- replace：用新的页面替换当前页面

Document

方法：选择器

选择器是考察浏览器相关知识点的重中之重，一般会结合实际场景进行考察。

`getElementById`，`getElementsByClassName`，`getElementsBytagName` 等早期规范定义的 API，还有新增的 `querySelector` `querySelectorAll` 等新规范增加的选择器

重点：`getElementsByTagName` 等返回多个 node 节点的函数返回值并不是数组，而是浏览器实现的一种数据结构。

方法：创建元素

`document.createElement` 能够创建一个 dom 元素，在新增多个元素时，可以先在内存中拼接出所有的 dom 元素后一次插入。

```
var fruits = ['Apple', 'Orange', 'Banana', 'Melon'];

var fragment = document.createDocumentFragment();

fruits.forEach(fruit => {
  const li = document.createElement('li');
  li.innerHTML = fruit;
  fragment.appendChild(li);
});

document.body.appendChild(fragment);
```

属性

title: `document.title` 可以设置或返回当前页面标题

domain: 展示当前网站的域名

url: 当前网站的链接

anchors: 返回所有的锚点，带 name 属性的 a 标签

forms: 返回所有的 form 标签集合

images: 返回所有的 img 标签集合

links: 返回所有带 href 属性的 a 标签

Element

Element 元素的 nodeType 均为 1 ，大多数标签都是一个 Element 实例

属性

tagName：返回当前元素的标签名

方法

- getAttribute：获取当前节点属性的结果
- setAttribute: 设置当前节点属性

Text 类型

Text 类型包含所有纯文本内容，他不支持子节点，同时他的 nodeType 为 3

History

History 对象包含用户（在浏览器窗口中）访问过的 URL。在 HTML 5 中，history 还与客户端路由息息相关。

属性

length： 返回历史列表中的网址数

方法

back：加载 history 列表中的前一个 URL

forward：加载 history 列表中的下一个 URL

go： 加载 history 列表中的某个具体页面

pushState：替换地址栏地址，并且加入 history 列表，但并不会刷新页面

replaceState：替换地址栏地址，替换当前页面在 history 列表中的记录，并不刷新页面

总结（面试常考点 & 易错点）

- 全局定义的变量均可以通过 window 来进行访问。使用 setInterval 需要注意，有可能代码并不是以相同间隔执行。使用 alert 等 API 需要注意，JS 代码可能会被阻塞。
- location 对象需要明确对于 URL 来说，每一个类型代表的具体值是什么。
- document 对象主要衔接 JS 和我们的 DOM 元素。需要注意这里很多选择的结果是 array-like 的类数组元素。以及使用 createFragment 代码片段等优化，来防止浏览器多次重排造成性能问题。
- Element 和 Text 是两个我们常见且易考易用的两个 DOM 对象。熟悉常见的方法和 debug 方式（console.dir）其次写代码时需要明确我们当前的方法究竟是 JS 层面的，还是环境层面的。
- history 因为和前端路由息息相关，我们需要熟悉新增的 pushState 和 replaceState 方法。

浏览器内置对象需要我们多看多练，以上是我们总结的一些常考点、易错点，对于基础的属性作用和用途，就需要同学们多在动手中打印熟悉确定了。

事件

事件是浏览器中的一个非常重要的内容，无论是面试还是工作中都是重点考察和使用的内容。

定义事件

我们可以通过多种方式对 DOM 元素定义一个事件：

```
<!-- 点击 p 标签弹出 alert -->
<p>点击后弹出 alert </p>
```

第一种方式，直接在 dom 元素中添加，不过这种方式一般不推荐，过分的将视图与逻辑部分的代码耦合。

```
<script>
  function showAlert() {
    alert('hello event');
  }
</scripts>
<p onclick="showAlert()">点击后弹出 alert </p>
```

第二种方式，纯 JS 解决，获取 dom 元素之后通过设置其 onclick 属性

```
document.getElementsByTagName('p')[0].onclick = function() {
  alert('hello world');
}

// 取消事件只需要设置 onclick 属性为 null 即可
document.getElementsByTagName('p')[0].onclick = null;
```

- 优点：纯 JS 实现，视图与逻辑解耦。
- 缺点：一个 dom 元素仅能设置一个 onclick 事件

第三种方式，纯 JS 解决，DOM2 级规范实现新的 API，`addEventListener` 和 `removeEventListener` 两个 API

```
var onClickFunc = function() {
  alert('hello world');
};

document.getElementsByTagName('p')[0].addEventListener('click', onClickFunc);

// 取消事件，使用 removeEventListener 即可
document.getElementsByTagName('p')[0].removeEventListener('click',
onClickFunc);
```

- 优点：
  - 纯 JS 实现，视图与逻辑解耦；
  - 通过 addEventListener 可以对 click 设置多个事件回调函数，他们会依次触发
- 缺点：
  - removeEventListener 删除的事件函数必须与设置时保持相同的函数引用，所以设置事件时尽量不使用匿名函数。

在 IE 中，为保证兼容性，我们需要通过 attachEvent 和 detachEvent 定义和删除事件，第一个参数接受事件名，第二个参数接受触发事件时的回调函数。

与 DOM2 规范定义的方法有区别的是，attachEvent 的事件名参数（第一个参数），需要加上 on 的前缀，例如：

```
var btn = document.getElementById('btn');
var onClickFunc = function() {
  alert('hello attachEvent');
}
btn.attachEvent('onclick', onClickFunc);
btn.detachEvent('onclick', onClickFunc);
```

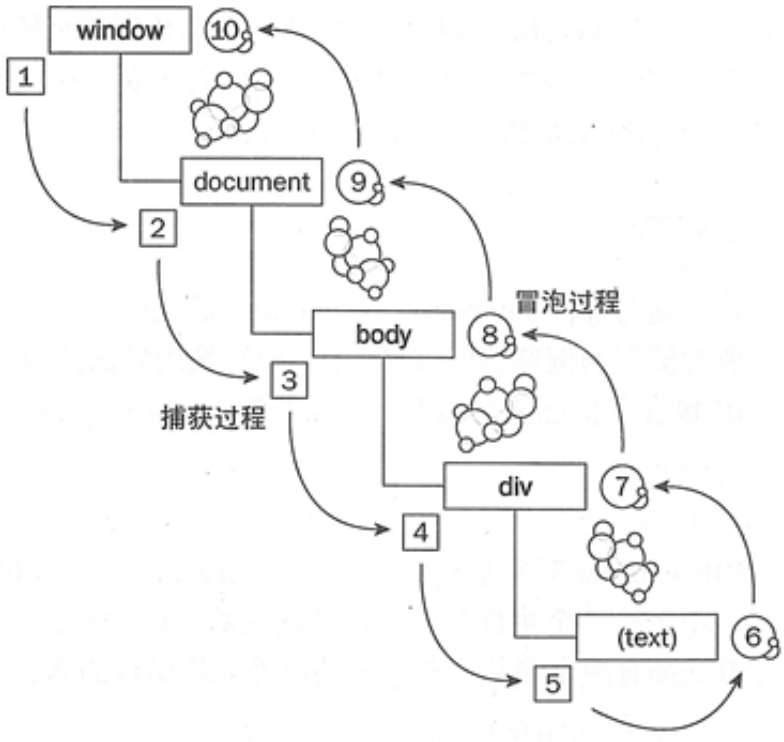
同时，如果多次对同一个元素设置相同事件，attachEvent 会按照相反的顺序来进行执行。

与 DOM2 事件规范相似的是，移除一个事件时还必须保证着事件的相同引用，否则无法清除事件。

事件捕获及冒泡

DOM 是一个嵌套性的树形树状结构，在浏览器中的表现就是叠加在一起的，所以在浏览器中点击一个区域，在 DOM 结构中会依次遍历多个 dom，自顶向下我们称为「事件捕获」，自下而上称为 「事件冒泡」。

DOM2 事件规范规定，一个标准的事件流分为三个阶段。首先是自上而下的「事件捕获」状态，然后是到达真正触发事件的元素，最后再从这个元素回到顶部的「事件冒泡」。



DOM2 级事件规范新增的事件定义函数 addEventListener，就可以通过第三个参数来指定究竟是在捕获阶段触发事件还是在冒泡阶段出发事件。第三个参数为 true 则在捕获阶段触发，第三个参数为 false 则在冒泡阶段触发。

IE 中的 attachEvent 不支持捕获或冒泡阶段的选择，仅支持在冒泡阶段触发。

事件对象

触发事件之后，浏览器会传入一个事件对象进入事件回调函数本身。

```
document.getElementsByTagName('p')[0].onclick = function(event) {
  console.log(event);
  alert('hello event');
};

document.getElementsByTagName('p')[0].addEventListener('click',
function(event) {
  console.log(event);
  alert('hello event');
});
```

event 对象下的属性

- bubbles：表明事件是否冒泡
- cancelable：表示是否可以取消事件的默认行为
- currentTarget：事件当前正在处理的元素
- defaultPrevented：为 true 则代表已经调用了 preventDefault 函数
- detail：事件细节
- eventPhase：事件所处阶段，1 代表捕获 2 代表在事件目标 3 代表冒泡
- type：事件类型（click 等）

event 对象下的方法

- preventDefault：取消事件的默认行为
- stopImmediatePropagation：取消事件的进一步捕获或冒泡，同时阻止事件处理程序调用
- stopPropagation：取消事件的进一步捕获或冒泡

IE 对象下的 event 有些许不同，如果通过 DOM0 规范定义的事件，是通过 window 来获取 event 内容，如果是 attachEvent 定义事件，同样也是通过传入回调函数中去。

```
var btn = document.getElementById('btn');

// DOM0 方式定义事件
btn.onclick = function() {
  var event = window.event;
};

btn.attachEvent('onclick', function(event) {
    console.log(event.type); // click
});
```

IE 下的 event 的属性方法

- cancelBubble：默认为 false，设置为 true 及取消了事件冒泡
- returnValue：默认为 true，设置 false 就会取消事件默认行为
- srcElement：事件的目标
- type：被触发的类型

事件委托

```
<ul id="ul">
    <p>1234</p>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
</ul>
```

```
document.getElementById('ul').onclick = function(event) {
  var target = event.target;
  if (target.nodeName.toLowerCase() === 'li') {
    alert(target.innerHTML);
  }
}
```

一个通用的事件模型

通用的事件模型主要是为了兼容多个 DOM 等级间设置事件的区别及 IE 和主流规范的不同，同时需要兼容 event 事件本身的内容。

```
var addEvent = function(element, eventType, handler) {
  if (element.addEventListener) {
    element.addEventListener(eventType, handler, false);
  } else if (element.attachEvent) {
    element.attachEvent('on' + eventType, handler);
  } else {
    element['on' + eventType] = handler;
  }
}

var removeEvent = function(element, eventType, handler) {
  if (element.removeEventListener) {
    element.removeEventListener(eventType, handler, false);
  } else if (element.detachEvent) {
    element.detachEvent('on' + eventType, handler);
  } else {
    element['on' + eventType] = null;
  }
}

var getEvent = function(event) {
  return event ? event : window.event;
}

var getTarget = function(event) {
```

```
  return event.target || event.srcElement;
}

var preventDefault = function(event) {
  if (event.preventDefault) {
    event.preventDefault();
  } else {
    event.returnValue = false;
  }
}

var stopPropagation = function(event) {
  if (event.stopPropation) {
    event.stopPropation();
  } else {
    event.cancelBubble = true;
  }
}
```

ajax

2005 年开始，ajax 作为一项新兴的交互技术开始影响 web 的发展。ajax 的核心是 `XMLHttpRequest` 对象。

```
var xhr = new XMLHttpRequest();

// xhr.open 接受三个参数，要发送的请求类型 get post、请求的 url、是否异步发送的布尔值
xhr.open('get', '/ajax.json', true);

// 调用 send 函数发送这个请求，参数为携带的参数
xhr.send(null);
```

比较常见的是发送 get 请求和 post 请求。

- 发送 get 请求时，我们一般把参数放置在 url 路径中，以 `?foo=bar&bar=foo` 这样的形式。
- 发送 post 请求时，数据放在 body 中，一般我们会以 form 表单的形式发送或者以 json 的形式发送数据。

get 请求发送数据：

```
var xhr = new XMLHttpRequest();

// xhr.open 接受三个参数，要发送的请求类型 get post、请求的 url、是否异步发送的布尔值
xhr.open('get', '/ajax?foo=bar&bar=foo', true);

// 调用 send 函数发送这个请求，参数为携带的参数
xhr.send(null);
```

post 请求发送 form 数据和 json 数据的示例：

```
var xhr = new XMLHttpRequest();

xhr.open('post', '/ajaxPost', true);
// 设置 request 的 content-type 为 application/x-www-form-urlencoded
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

var data = new FormData();
data.append('foo', 'bar');
data.append('bar', 'foo');

xhr.send(data);
```

```
var xhr = new XMLHttpRequest();

xhr.open('post', '/ajaxPost', true);
// 设置 request 的 content-type 为 application/json
xhr.setRequestHeader('Content-Type', 'application/json');
var data = JSON.stringify({
  foo: 'bar',
  bar: 'foo'
});
xhr.send(data);
```

上面这个例子中我们通过 setRequestHeader 向后端发送一些自定义 header，除了浏览器默认发送的 header 以外，也会带上我们自定义的 header 头部，后端同学收到这些内容就可以进行处理。

同样的，我们可以通过 onreadystatechange 事件，监听当前 xhr 实例的阶段，通过判断 xhr.readyState 的阶段，来判断当前请求的状态。

readyState 状态如下：

- 0：未调用 open 方法
- 1：已调用 open 方法但未调用 send 方法
- 2：已调用 send 方法但尚未收到返回
- 3：收到部分响应数据
- 4：收到所有响应数据

同时 xhr 实例上还有 `xhr.responseText` 代表响应主体返回的文本，`xhr.status` 代表响应的 HTTP 状态码，`xhr.statusText` HTTP 状态说明

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
  if (xhr.readystate !== 4) return;
  if ((xhr.status >= 200 && xhr.status < 300) || xhr.status === 304) {
    alert(xhr.responseText);
  } else {
```

```
    alert("错误：状态码 " + xhr.status + xhr.statusText);
  }
}

xhr.open('get', '/ajax?foo=bar&bar=foo', true);

xhr.send(null);
```

我们也可以通过 `xhr.getRequestHeader` 来获取服务端返回的 header

ES6 之后的 fetch API

在 ES6 之后，浏览器端新增了一个 fetch api， 他有以下的几个特点：

- fetch api 返回一个 promise 的结果
- 默认不带 cookie，需要使用配置 `credentials: "include"`
- 当网络故障时或请求被阻止时，才会标记为 reject。否则即使返回码是 500，也会 resolve 这个 promise

```
fetch('/ajax?foo=bar')
  .then(function() {
    console.log('请求发送成功');
  })
```

这就是一个简单的请求，发送之后就会进入 resolve 状态。与普通的 ajax 请求不同的是，在服务端返回内容时，我们还需要调用一些方法才能拿到真正返回的结果。

```
fetch('/ajax?foo=bar')
  .then(function(response) {
    response.text(); // 返回字符串
    response.json(); // 返回 json
    response.blob(); // 一般指返回文件对象
    response.arrayBuffer(); // 返回一个二进制文件
    response.formData(): // 返回表单格式内容
  });
```

常见的 json 请求，我们需要再调用一次 `response.json` 来让 fetch API 返回的结果序列化为 json

```
fetch('/ajaxPost')
  .then(function(response) {
    return response.json();
  })
  .then(function(result) {
    console.log(result);
  });
```

封装的通用 ajax 请求

```
function fetch(url, config = {}) {
    if (window.fetch) return window.fetch(url, config);
    return new Promise((resolve, reject) => {
        function createXHR() {
            if (typeof XMLHttpRequest !== undefined) {
                return new XMLHttpRequest();
            }
            // 兼容早期 IE
            if (typeof ActiveXObject !== undefined) {
                if (typeof arguments.callee.activeXString !== 'string') {
                    var versions = ['MSXML2.XMLHttp.6.0',
'MSXML2.XML2.XMLHttp.3.0', 'MSXML2.XMLHttp'];
                    for (var i = 0; i < versions.length; i++) {
                        try {
                            new ActiveXObject(versions[i]);
                            arguments.callee.activeXString = versions[i];
                            break;
                        } catch (e) {}
                    }
                }
                return new ActiveXObject(arguments.callee.activeXString);
            }

            throw new Error('不支持 xhr 相关内容');
        }

        var xhr = createXHR();

        xhr.onreadystatechange = function() {
            console.log(xhr);
            if (xhr.readyState !== 4) return;
            var options = {
                status: xhr.status,
                statusText: xhr.statusText
            };
            var body = 'response' in xhr ? xhr.response : xhr.responseText;
            var response = {
                status: options.status || 200,
                statusText: options.statusText || 'ok',
                ok: options.status >= 200 && options.status < 300,
                text() {
                    if (typeof body === 'string') {
                        return Promise.resolve(body);
                    }
                },
                json() {
                    return this.text().then(JSON.parse);
                }
            };
```

```
            resolve(response);
        }
        xhr.open(config.method || 'get', url, true);
        xhr.send();
    });
}
```