1.事件机制

事件机制包含

- 事件触发三阶段
- 事件注册
- 事件代理

(1) 事件触发三阶段/事件流

捕获阶段、目标阶段、冒泡阶段

捕获阶段：事件从 window 节点流向目标阶段，途中流经各个 DOM 节点，直到到达目标节点

目标阶段：事件到达目标节点，事件将会在目标节点上被触发（window,document,html,body...)

冒泡阶段：事件在目标节点上触发后，不会终止，会一层层往上冒，最终回溯到 window 节点

每个 event 都有一个 `event.bubbles` 属性，可以知道它可否冒泡。浏览器中哪些事件会冒泡 https://segmentfault.com/q/1010000000687977/a-1020000000688757

(2) 事件注册

DOM 事件模型中，通过 addEventListener 注册事件

element.addEventListener(String type, Function listener, boolean useCaptrue);

参数说明如下：

- type：注册事件的类型名。事件类型与事件属性不同，事件类型名没有 on 前缀。

  例如，对于事件属性 onclick 来说，所对应的事件类型为 click。

- listener：监听函数，即事件处理函数。在指定类型的事件发生时将调用该函数。

  在调用这个函数时，默认传递给它的唯一参数是 event 对象。

- useCaptrue：是一个布尔值。如果为 true，则指定的事件处理函数将在事件传播的捕获阶段触发；如果为 false，则事件处理函数将在冒泡阶段触发。

- 销毁事件

  element.removeEventListener(String type, Function listener, boolean useCaptrue);

(3) 事件代理/事件委托

把事件注册到父元素上，叫做事件代理。事件委托就是利用事件冒泡机制，指定一个事件处理程序，来管理某一类型的所有事件。

如果每个子元素都需要绑定一次事件的话，就可以进行事件代理，可以节省内存，不需要给子节点注销事件

比如一个 ul 中有 100li，每个 li 都需要处理 click 事件，那我们可以遍历所有 li，给它们添加事件处理程序，但是这样做会有什么影响呢？我们知道添加到页面上的事件处理程序的数量将直接影响到页面的整体运行性能，因为这需要不停地与

dom 节点进行交互，访问 dom 的次数越多，引起浏览器重绘和重排的次数就越多，自然会延长页面的交互就绪时间，这也是为什么可以减少 dom 操作来优化页面的运行性能；而如果使用委托，我们可以将事件的操作统一放在 js 代码里，这样与 dom 的操作就可以减少到一次，大大减少与 dom 节点的交互次数提高性能。同时，将事件的操作进行统一管理也能节约内存，因为每个 js 函数都是一个对象，自然就会占用内存，给 dom 节点添加的事件处理程序越多，对象越多，占用的内存也就越多；而使用委托，我们就可以只在 dom 节点的父级添加事件处理程序，那么自然也就节省了很多内存，性能也更好。

2.跨域

九种 https://www.cnblogs.com/fundebug/p/10329202.html

跨域的产生来源于浏览器的同源策略，所谓同源，是指在地址的：

1.协议名：http,https

2.域名：http://baidu.com

3.端口名：http://baidu.com:9090

均一样的情况下，才可以访问相同的 cookie,localStorage 或者发送 ajax 请求

在不同源的情况下访问，就称为跨域。

2.1 为什么要有跨域限制

Ajax 的同源策略主要是为了防止 CSRF (跨站请求伪造) 攻击，如果没有 AJAX 同源策略，相当危险，我们发起的每一次 HTTP 请求都会带上请求地址对应的 cookie，那么可以做如下攻击：

- 用户登录了自己的银行页面 mybank.com，mybank.com 向用户的 cookie 中添加用户标识。
-
- 用户浏览了恶意页面 evil.com。执行了页面中的恶意 AJAX 请求代码。
-
- evil.com 向 http://mybank.com 发起 AJAX HTTP 请求，请求会默认把 http://mybank.com 对应 cookie 也...。
-
- 银行页面从发送的 cookie 中提取用户标识，验证用户无误，response 中返回请求数据。此时数据就泄露了。
-
- 而且由于 Ajax 在后台执行，用户无法感知这一过程
-

2.2 如何解决跨域

Json,CORS,WebSocket,postMessage

2.2.1jsonp(非官方)

https://www.cnblogs.com/liubingyjui/p/10804785.html

jsonp 全称是 JSON with Padding,是为了解决跨域请求资源而产生的解决方案,是一种依靠开发人员创造出的一种非官方跨域数据交互协议。
一个是描述信息的格式，一个是信息传递双方约定的方法。

jsonp 的产生:

1.AJAX 直接请求普通文件存在跨域无权限访问的问题,不管是静态页面也好.

2.不过我们在调用 js 文件的时候又不受跨域影响,比如引入 jquery 框架的,或者是调用相片的时候

3.凡是拥有 scr 这个属性的标签都可以跨域例如`<script><img><iframe>`,还有 link 标签的 href,他们没有被同源策略所限制。

4.如果想通过纯 web 端跨域访问数据只有一种可能,那就是把远程服务器上的数据装进 js 格式的文件里.

5.而 json 又是一个轻量级的数据格式,还被 js 原生支持

6.为了便于客户端使用数据，逐渐形成了一种非正式传输协议，人们把它称作 JSONP，该协议的一个要点就是允许用户传递一个 callback 参数给服务端,

```
$('#a').click(function(){
            var frame = document.createElement('script');
            frame.src = 'https://segmentfault.com/write?freshman=chan&callback=mount_chan';
            $('body').append(frame);
        });
        function mount_chan(res){
```

```
        console.log(res.message)
    }
```

后端返回的 script 里是执行 mount_chan 这个函数的代码, 其中需要的数据已经有了。参考 https://segmentfault.com/a/1190000022428321

缺点:

只能发送 get 请求

各种例子 https://www.cnblogs.com/liubingyjui/p/10804785.html

2.2.2 CORS

https://segmentfault.com/a/1190000023206598

CORS 是跨源资源分享(Cross-Origin Resource sharing) 的缩写，它是 W3c 标准，是跨域 AJAX 请求的根本解决办法。相比 JSONP 只能发 GET 请求，CORS 允许任何类型的请求。 CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览器不能低于 IE10。

整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS 通信与同源的 AJAX 通信没有差别，代码完全一样。浏览器一旦发现 AJAX 请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。

因此, 实现 CORS 通信的关键是服务器(服务器端可是判断, 让那些域可以请求)。只要服务器实现了 CORS 接口， 就可以跨源通信

3.1 CORS 标准

CORS 是一个 W3C 标准，全称是跨域资源共享（CORSs-origin resource sharing），它允许浏览器向跨源服务器，发出 XMLHttpRequest 请求。

其实，准确的来说，跨域机制是阻止了数据的跨域获取，不是阻止请求发送。

CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览器不能低于 IE10。

https://caniuse.com/#search=cors

整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS 通信与同源的 AJAX 通信没有差别，代码完全一样。浏览器一旦发现 AJAX 请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。

因此，实现 CORS 通信的关键是服务器。只要服务器实现了 CORS 接口，就可以跨域通信。

3.2 CORS 跨域判定的总体流程

如图所示，跨域的判定流程为：

- 网页上的 JS 代码，从浏览器上发送 XMLHttpRequest 请求到服务端
-
- 如果该请求为
- 简单请求
- ，浏览器会直接发送实际请求到服务端，浏览器会根据服务端的响应，判断该请求是否可以跨域：

- （1）如果不能跨域，浏览器会报错，阻止 JS 代码进一步执行；
- （2）如果能够跨域，则 JS 能正常处理响应，进行后续业务流程
-
- 如果该请求为
- 非简单请求
- ，浏览器会先发送一个预检请求(preflight)，方法为 OPTIONS，然后针对服务器的响应，做上述跟简单请求一样相同的判断：
- （1）如果不能跨域，则实际请求不会发送
- （2）如果能够跨域，则实际请求会进行发送，进行后续业务处理
- 值得说明的是，浏览器在跨域的情况下，请求都会发送出去，但是对于响应会判断是否满足跨域条件，如果不满足，则报错，阻止 JS 后续的执行流程，例如读取响应数据等。也就是说，跨域机制主要是阻止数据的跨域获取，不是阻止请求的发送。

2.2.3 WebSocket

它是一种通信协议，使用 ws:///(非加密),wss>///(加密)作为协议前缀，该协议不实行同源策略，只要服务器支持，就可以通过它进行通信。

```
var ws = new WebSocket('wss://echo.websocket.org');


        ws.onopen = function (evt) {

            console.log('Connection open ...');

            ws.send('Hello WebSockets!');

        };


        ws.onmessage = function (evt) {

            console.log('Received Message: ', evt.data);

            ws.close();

        };


        ws.onclose = function (evt) {

            console.log('Connection closed.');

        };
```

2.2.4 postMessage

```
window.addEventListener(“message”, function(){},false);

otherWindow.postMessage(message, targetOrigin);
```

-

利用 postMessage 不能和服务端交换数据，只能在两个窗口（iframe）之间交换数据

-
-

两个窗口能通信的前提是，一个窗口以 iframe 的形式存在于另一个窗口，或者一个窗口是从另一个窗口通过 window.open()或者超链接的形式打开的（同样可以用 window.opener 获取源窗口）

-
-

message: 将要发送到其他 window 的数据。

-
-

targetOrigin:通过窗口的 origin 属性来指定哪些窗口能接收到消息事件，其值可以是字符串"*"（表示无限制）或者一个 URI。在发送消息的时候，如果目标窗口的协议、主机地址或端口这三者的任意一项不匹配 targetOrigin 提供的值，那么消息就不会被发送；只有三者完全匹配，消息才会被发送。

-
-

transfer(可选)：是一串和 message 同时传递的 Transferable 对象．这些对象的所有权将被转移给消息的接收方，而发送一方将不再保有所有权。

-

3.事件循环/Event-loop

同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；

异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

异步任务：

1.普通事件，click,resize 等

2.资源加载：load,error 等

3.定时器：setInterval,setTimeout

事件循环的具体过程

https://blog.csdn.net/qazxbjp2010/article/details/104660409/

- 同步任务进入主线程，异步任务进入 Event Table 并注册函数
-
- 当异步任务完成时，Event Table 会将这个函数移入 Event Queue(任务队列)
-
- 主线程内的同步任务执行完毕，执行栈为空，会去 Event Queue 读取对应的函数，进入主线程执行
-
- 上述过程不断重复，就是常说的事件循环
-

所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

宏任务与微任务

https://www.jianshu.com/p/bfc3e319a96b

在 js 中，任务分宏任务（MacroTask）和微任务（MicroTask）

宏任务： script(整体代码), setTimeout, setInterval, setImmediate, I/O, UI rendering

微任务：Promise,await, Object.observe, MutationObserver

浏览器为了让 js 内部 task 与 DOM 任务能够有序执行，会在一个 task 结束后，在下一个 task 开始前，对页面进行渲染（task->渲染->task）

概念 5：宏任务和微任务

> ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `task` 宏任务是由宿主发起的，而微任务由 JavaScript 自身发起。

在 ES3 以及以前的版本中，JavaScript 本身没有发起异步请求的能力，也就没有微任务的存在。在 ES5 之后，JavaScript 引入了 `Promise`，这样，不需要浏览器，JavaScript 引擎自身也能够发起异步任务了。

所以，总结一下，两者区别为：

	宏任务（macrotask）	
谁发起的	宿主（Node、浏览器）	JS
具体事件	1. script (可以理解为外层同步代码) 2. setTimeout/setInterval 3. UI rendering/UI 事件 4. postMessage，MessageChannel 5. setImmediate，I/O（Node.js）	1. 弃
谁先运行	后运行	先
会触发新一轮 Tick 吗	会	不

宏任务与微任务执行顺序：

微任务的优先度高于宏任务

- 执行栈在执行完同步任务后，查看执行栈是否为空，如果执行栈为空，就会去检查微任务队列是否为空，如果为空的话，就执行宏任务，否则就一次性执行完所有微任务。
-
- 每次单个宏任务执行完毕后，检查微任务队列是否为空，如果不为空的话，会按照先入先出的规则全部执行完微任务后，设置微任务队列为 `null`，然后再执行宏任务，如此循环
-

4.渲染机制

参考输入 url 后发生了什么

https://blog.csdn.net/liujianfeng1214/article/details/86690284

关键渲染路径是指浏览器从最初接收请求来的 HTML、CSS、javascript 等资源，然后解析、构建树、渲染布局、绘制，最后呈现给客户能看到的界面这整个过程

5.Service Worker

https://zhuanlan.zhihu.com/p/115243059

一个服务器与浏览器之间的中间人角色（代理服务器），如果网站中注册了 service worker 那么它可以拦截当前网站所有的请求，进行判断（需要编写相应的判断程序），如果需要向服务器发起请求的就转给服务器，如果可以直接使用缓存的就直接返回缓存不再转给服务器。从而大大提高浏览体验

-

基于 web worker（一个独立于 JavaScript 主线程的独立线程，在里面执行需要消耗大量资源的操作不会堵塞主线程）

-
-

在 web worker 的基础上增加了离线缓存的能力

-
-

本质上充当 Web 应用程序（服务器）与浏览器之间的代理服务器（可以拦截全站的请求，并作出相应的动作->由开发者指定的动作）

-
-

创建有效的离线体验（将一些不常更新的内容缓存在浏览器，提高访问体验）

-
-

  由事件驱动的,具有生命周期

-
-

  可以访问 cache 和 indexDB

-
-

  支持推送

-
-

  并且可以让开发者自己控制管理缓存的内容以及版本

-

6.什么是 Web Worker?

当在 HTML 页面中执行脚本时，页面的状态是不可响应的，直到脚本已完成。

web worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 web worker 在后台运行。

https://blog.csdn.net/weixin_40122996/article/details/82533223