BFC 作用：

隔绝容器内部元素和容器外部元素的相互作用。

创建 BFC 的方法

1. 浮动
2. 绝对定位，position 为 absolute 或者 fixed
3. 行内块 inline-block
4. 表格单元格 display: table-cell;
5. overflow 不为 visible
6. 弹性盒 display: flex;

实际用法：

1. 清除浮动
2. 防止与浮动元素重叠
3. 利用 BFC 包含一个元素，避免边距重叠，margin collapse

DOM事件相关问题

DOM 事件级别：

DOM 0 ：ele.onclick = function(){}

DOM 2 ：ele.addEventListener('click',function(){})

DOM 3 ：ele.addEventListener('keyup',function(){}) // 支持了更多的事件

DOM 事件模型： 捕获与冒泡

DOM 事件流 ： 捕获阶段 ---> 处于目标阶段 ----> 冒泡阶段

描述DOM事件捕获的具体流程：

window ---> document ---> html ---> body ---> ....(目标父级元素) --->目标元素

Event对象常见应用：

```
event.preventDefault() // 阻止默认事件
// 阻止后续节点的事件冒泡，不阻止当前元素的后续事件冒泡
// 阻止冒泡，比如父元素和子元素都绑定了点击事件，如果不想点击子元素时触发父级元素的点击事件，需要阻止冒泡
event.stopPropagation()
// 执行完当前事件操作之后，阻止当前元素及后续节点的事件冒泡
// 如果给一个元素使用 addEventListener 同时注册了两个点击事件 A 和 B，在触发了A之后，不想出发B，那么在A中就应该使用此方法
event.stopImmediatePropagation(); // 立即阻止冒泡
// 使用父级元素代理事件的时候，currentTarget 指向父级元素，即事件绑定元素
event.currentTarget
// 使用父级元素代理事件的时候，currentTarget 指向触发事件的元素
event.target
```

自定义事件：

new Event() 和 new CustomEvent(),区别在于 CustomEvent 可以携带一个自定义数据

```
// 自定义事件
const selfEvent = new Event("self");
const customEvent = new CustomEvent('custom',{age:18});

document.getElementById("selfButton").addEventListener("click", () => {
  document.dispatchEvent(selfEvent);
});
document.addEventListener("self", (e) => {
  console.log("slef事件触发了");
});
```

变量类型和计算

1. **JS中使用typeof 都能得到哪些类型?**

   值类型：string、number、boolean、undefined

   引用类型：object、function

   ```
   typeof "abc"; // string
   typeof 1; // number
   typeof true; // boolean
   typeof undefined; // undefined

   typeof {}; // object
   typeof []; // object
   typeof null; // object
   let fn = function() {};
   typeof fn; // function
   ```

2. **何时使用 == ,何时使用 ===**

   使用 == 时会进行类型转换，而 === 不会。

   只在 if 语句中，判断是否 == null 或者 == undefined 才使用，jQuery 源码中推荐这种简写。

   可以通过 !! 来快速判断在 if 语句中，变量转换后的值。

   ```
   if(a == null){
   }
   -------------------------
   let a = 0;
   !!a // false
   ```

3. **JS 中有哪些内置函数?**

   Boolean 、String 、Number 、Object、Function

   容易搞忘的：**Array、Error、RegExp、Date**

4. **JS变量按照存储方式区分为哪些类型，并描述其特点**

   分为 值类型 和 引用类型

   值类型每个变量存储各自的值，不会相互影响

   引用类型是将值存放到同一个内存中，变量通过指针去获取，不同的变量之间的值会相互影响

5. **如何理解JSON**

在JS 中就是一个 JS 对象，同时也是一种数据格式

面向对象

类的声明方式与实例化：

```
function Maker(name) {
  this.name = name;
}
let person = new Maker("yu");

// es6 类声明
class Maker2 {
  constructor(name) {
    this.name = name;
  }
}
let person2 = new Maker2("yi");
```

如何实现继承，继承有几种方式?

第一种方式：使用构造函数

缺点：无法继承Parent的原型对象(prototype)上的方法和属性

```
function Parent (name='person'){
    this.name = name;
}
function Children(type){
    Parent.call(this)
    this.type = type;
}
let coder = new Children('coder')
---------------------------------------
function Parent(name = "person") {
  this.name = name;
}
// child 无法继承 parent 原型对象的方法和属性
Parent.prototype.say = function() {
  console.log("say");
};
function Children(type) {
  Parent.call(this);
  this.type = type;
}
let coder = new Children("coder");
coder.say(); // 报错  Uncaught TypeError: coder.say is not a function
```

第二种方式：继承方式

```
function Parent(name = "person") {
  this.name = name;
}
function Children(type) {
  this.type = type;
}
Children.prototype = new Parent();
let coder = new Children("coder");
```

缺点：所有children生成的实例的原型对象都是同一个引用，所以如果其中一个实例修改了原型对象中的属性或者方法，其他的实例所继承的属性和方法都会改变，非常危险！！！

```
function Parent(name = "person") {
  this.name = name;
  this.play = [1, 2, 3];
}
function Children(type) {
  this.type = type;
}
Children.prototype = new Parent();
let coder1 = new Children("coder1");
let coder2 = new Children("coder2");
coder1.play.push(4);
console.log(coder1.play); // [1, 2, 3, 4]
console.log(coder2.play); // [1, 2, 3, 4]
```

第三种方式：组合继承法，同时使用构造函数和继承，是最常用的方法

缺点：Parent 函数运行了两次，存在优化空间

```
function Parent(name = "person") {
  this.name = name;
  this.play = [1, 2, 3];
}
function Children(type) {
  // 在这儿执行一次构造函数
  Parent.call(this);
  this.type = type;
}
// 在这儿再执行一次构造函数
Children.prototype = new Parent();
let coder1 = new Children("coder1");
let coder2 = new Children("coder2");
coder1.play.push(4);
console.log(coder1.play); // [1, 2, 3, 4]
console.log(coder2.play); // [1, 2, 3]
```

组合继承方法的优化方案1：

不执行new Parent，直接将 Children 的原型引用改为 Parent 的原型

```
function Parent(name = "person") {
  this.name = name;
  this.play = [1, 2, 3];
}
function Children(type) {
```

```
  Parent.call(this);
  this.type = type;
}
// 不执行new Parent，直接将 Children 的原型引用改为 Parent 的原型
Children.prototype = Parent.prototype;
let coder1 = new Children("coder1");
let coder2 = new Children("coder2");
coder1.play.push(4);
console.log(coder1.play); // [1, 2, 3, 4]
console.log(coder2.play); // [1, 2, 3]
```

缺点：Children 生成的实例的原型对象(__ proto __)的constructor属性，指向的是Parent ，而不是Children

```
console.log(coder1.__proto__.constructor);
// ƒ Parent(name = "person") {
//   this.name = name;
//   this.play = [1, 2, 3];
// }
```

组合继承方法优化方案2：

使用 Object.create 方法创建一个中间对象，并且将这个中间对象的 constructor 属性，手动指向 Children，完美收官。

```
function Parent(name = "person") {
  this.name = name;
  this.play = [1, 2, 3];
}
function Children(type) {
  Parent.call(this);
  this.type = type;
}
// 使用 Object.create 创建一个中间对象，虽然有了一个中间对象，但是constructor依然指向 Parent
Children.prototype = Object.create(Parent.prototype);
// 手动将constructor指向Children，这样子就完美
Children.prototype.constructor = Children;
let coder1 = new Children("coder1");
let coder2 = new Children("coder2");
coder1.play.push(4);
console.log(coder1.play); // [1, 2, 3, 4]
console.log(coder2.play); // [1, 2, 3]

console.log(coder1.__proto__.constructor);
// ƒ Children(type) {
//     Parent.call(this);
//     this.type = type;
//   }
```

###

第四种方式：使用 ES6 的 class 继承：

```
class Parent {
  constructor(name) {
    this.name = name;
```

```
  }
  printName() {
    console.log(this.name);
  }
}

class Children extends Parent {
  constructor(name, age) {
    super(name);
    this.age = age;
  }
  printAge() {
    console.log(this.age);
  }
}

let coder = new Children("yu", 26);
coder.printName();
coder.printAge();
```

创建格式化时间戳为 2019-09-01 格式的时间函数

```
function formatDate(date) {
  if (!date) {
    date = new Date();
  }
  const year = date.getFullYear();
  let month = date.getMonth() + 1; // getMonth 返回 0-11
  if (month < 10) {
    month = "0" + month;
  }
  let day = date.getDate();
  if (day < 10) {
    day = "0" + day;
  }
  return `${year}-${month}-${day}`;
}

let now = new Date();
let formatedDate = formatDate(now); // 2019-09-01
```

获取随机数，要求长度一致的字符串格式

```
function getRandomNum() {
  let random = Math.random();
  let tempRandom = random + "0000000000"; // 后面添加10个0
  return tempRandom.slice(0, 10);
}
let randomNum = getRandomNum();
```

创建一个可以同时处理对象和数组的 forEach 函数

```
function ownForEach(obj, fn) {
```

```
  if (obj instanceof Array) {
    obj.forEach((item, index) => {
      fn(index, item);
    });
  } else {
    for (let key in obj) {
      if (obj.hasOwnProperty(key)) {
        fn(key, obj[key]);
      }
    }
  }
}
ownForEach([1, 2, 3], (index, item) => {
  console.log(index, item);
});   // 0 1     1 2    2 3
ownForEach({ x: 1, y: 2, z: 3 }, (index, item) => {
  console.log(index, item);
});   // X:1   y:2   z:3
```

数组

map，filter，reduce的作用与区别

map 的作用是对数组每一项进行操作后返回新数组。

filter 的作用是过滤数组中不需要的部分

reduce 的作用是将数组中的项进行操作后返回一个值

```
const arr = [1, 2, 3];
const arrMap = arr.map(item => item + 1);
const arrFilter = arr.filter(item => item < 3);
const arrReduce = arr.reduce((acc, current) => acc + current, 0);
console.log(arrMap, arrFilter, arrReduce);
// [2,3,4]  [1,2]  6
```

单线程与异步

单线程

1.js 单线程概念：

js 在同一时间只能做一件事

2.原因

避免 DOM 渲染冲突，因为 js 可以修改 DOM，如果同时执行两段 js，就可能导致 DOM 渲染出错。

3.单线程解决方案

异步

异步

1.异步存在的问题

1. 代码执行顺序和书写顺序不一致，容易导致错误
2. callback 中不容易实现模块化，容易陷入回调地狱，即一层回调嵌一层回调

2.任务队列

包含同步队列(主进程、运行栈)和异步队列

3.异步实现机制

事件轮询

4.事件轮询 event loop

1. 浏览器在主进程中从上到下执行语句，同步语句直接执行。
2. 异步任务，放到异步队列中去。setTimeout 0 函数，直接放入异步队列。如果有定时时间，到了定时时间再放入异步队列。如果是 ajax 请求，在收到响应后再放入异步队列。
3. 当主进程中的任务执行完了之后，浏览器将去异步队列中查询任务。

   如果有任务就放到主进程中开始执行。

   如果没有，就监听异步队列，当异步队列中有新的任务时，直接放到主进程中执行。

5.异步类任务

1. setTimeout setInterval
2. DOM 事件，比如 addEventListener
3. ES6中的Promise

6. 异步解决方案

1. jQuery Deferred
2. Promise
3. async / await
4. generator

原型与原型链相关

创建对象的几种方法：

```
// 第一种，对象字面量
let o1 = { name: "yu" };
let o2 = new Object({ name: "yi" });
// 第二种，使用构造函数
let M = function(name) {
  this.name = name;
};
let o3 = new M("yu yi");
// 第三种，使用Object.create
let p = { name: "yu yi" };
let o4 = Object.create(p);
```

原型、构造函数、实例、原型链：

instanceof 原理：

instanceof 用来判断实例是否由属于某种类型，注意，也可以判断是否是其父类型或者祖先类型的实例。所以使用instanceof 不是很准确。如果要判断实例是否由某个构造函数直接生成，应该使用下面的代码。

实际上 instanceof 判断的是,实例的 __ proto__ 指向的原型对象 A 和 构造函数的 prototype 指向的原型对象 B ,是否在同一个原型链上。

```
let M = function(name) {
  this.name = name;
};
let o3 = new M("yu yi");
o3.__proto__.constructor === M // 准确判断
```

如何判断一个变量是否为数组：

```
const a = [];
a instanceof Array // true
Object.prototype.toString.call(a) // "[object Array]"
```

new 运算符的工作原理：

```
let obj = new M('YUI');
```

1. 创建一个空对象A，继承自构造函数的原型对象
2. 执行构造函数，并把构造函数中的 this 指向这个空对象A
3. 判断构造函数执行完后会不会返回一个新对象B，如果有，就抛出这个B，如果没有就返回之前创造的空对象A

```
let new2 = function(func) {
  let obj = Object.create(func.prototype); // 1
  let k = func.call(obj); // 2
  if (typeof k === "object") {  // 3
    return k;
  } else {
    return obj;
  }
};
```

原型链与闭包高级应用

zepto 中 $ 方法的简单实现：

```
(function(window) {
  // 先定义一个 zepto对象，不会被外界干扰
  // $ 运算符的生效过程    $ --> zepto.init --> zepto.z --> 构造函数Z ---> Z的
prototype,同时也是 $的fn
  // 在 $.fn上去处理相应的函数调用
  //  使用了闭包和原型链
  let zepto = {};
  function Z(dom, selector) {
    let i,
      len = dom ? dom.length : 0;
    for (i = 0; i < len; i++) {
      this[i] = dom[i];
```

```
    }
    this.length = len;
    this.selector = selector;
  }
  zepto.Z = function(dom, selector) {
    return new Z(dom, selector);
  };
  zepto.init = function(selector) {
    let slice = Array.prototype.slice;
    let dom = slice.call(document.querySelectorAll(selector));
    return zepto.Z(dom, selector);
  };
  let $ = function(selector) {
    return zepto.init(selector);
  };
  $.fn = {
    css(key,value) {
      alert("zepto css");
    },
    html(value) {
      alert("zepto html");
    }
  };
  Z.prototype = $.fn;
  window.$ = $;
})(window);
```

jQuery 中 $ 方法的简单实现：

```
(function(window) {
  // $ --> new init()
  // init 的 prototype 指向 $.fn 对象，在这个对象下挂载相应方法
  let jQuery = function(selector) {
    return new jQuery.fn.init(selector);
  };
  jQuery.fn = {
    css(key, val) {
      alert("jQuery css");
    },
    html(val) {
      alert("jQuery html");
    }
  };
  let init = (jQuery.fn.init = function(selector) {
    let slice = Array.prototype.slice;
    let dom = slice.call(document.querySelectorAll(selector));
    let i,
      len = dom ? dom.length : 0;
    for (i = 0; i < len; i++) {
      this[i] = dom[i];
    }
    this.length = len;
    this.selector = selector || "";
  });
  init.prototype = jQuery.fn;
  window.$ = jQuery;
```

```
})(window);
```

jQuery 的插件机制

通过将方法挂载到 $.fn 下，实现了仅暴露 $ 变量就可以进行插件扩展。

优点：1. 只暴露$，不暴露jQuery构造方法，使得外部无法修改jQuery内部的变量

2. 易于通过 $.fn 进行插件扩展。

作用域和闭包

说一下对变量提升的理解：

使用 var 定义的变量和函数声明，都会被提升到代码的最顶端。函数声明会将整个函数提升到顶端，而变量提升只提前声明变量。

使用 let 和 const 声明的变量不会。

```
console.log(a); // undefined
var a = 100;

fn(); // 12
function fn() {
  console.log(12);
}
----------------------------------
var a;
console.log(a); // undefined
a = 100;

function fn() {
  console.log(12);
}
fn(); // 12
----------------------------------
console.log(a); // Uncaught ReferenceError: a is not defined
let a = 100;

console.log(a); // Uncaught ReferenceError: a is not defined
const a = 100;
```

说明 this 几种不同的使用场景：

1. 构造函数中使用：this 指向 new 运算符创建的空对象

```
function Foo(name) {
  // this 指向 new 运算符创建的空对象
  this.name = name;
}
let user = new Foo("yu");
```

2. 普通函数中使用，this 指向 window

```
var a = 10;
function print() {
  // this 指向 window
  console.log(this.a); // 10
}
print();
```

3. 对象中使用，this 指向该对象

```
let obj = {
  name: "yu",
  getName() {
    console.log(this.name);
  }
};
obj.getName();
```

4. 使用 call、apply、bind时，this 指向传入的对象

```
let obj = {
  name: "yu",
  age: 26,
  getNameAndAge(age) {
    console.log(this);
    console.log(this.name);
    console.log(age);
  }
};
obj.getNameAndAge.call({ name: "gou" }, 25);
// { name: "gou" }  gou   25
// apply 传参数要用数组的形式
obj.getNameAndAge.apply({ name: "mi" }, [23]);
// { name: "mi" }  mi  23

let print = function() {
  console.log(this.name);
}.bind({ name: 'yui' });
print() // yui
```

总结：this 的指向是在运行时确定的，而不是定义时候

作用域相关：

自由变量：当前作用域中未定义的变量

作用域链：当要获取一个变量的值的时候，先判断是否能在当前作用域获取，如果不能再往上级作用域寻找，直到window

```
// 创建10个a标签，点击的时候弹出相应的序号：
for (var i = 0; i < 10; i++) {
  (function(i) {
    let a = document.createElement("a");
    a.innerHTML = i + "<br/>";
    a.addEventListener("click", e => {
      e.preventDefault();
      console.log(i);
    });
    document.body.appendChild(a);
  })(i);
}
```

闭包：闭包主要是为了防止污染变量

1. 函数作为返回值

```
function printFn() {
  var a = 100;
  return function() {
    console.log(a);
  };
}

var a = 20;
var print = printFn();
print(); // 100
```

2. 函数作为参数传递

```
function F1() {
  var a = 10;
  return function() {
    console.log(a);
  };
}
var a = 30;
var f1 = F1();
function f2(fn) {
  var a = 20;
  fn();
}
f2(f1); // 10
```

```
// 实际开发中闭包的用法
function isFirstLoad(id) {
  let _list = [];
  return function() {
    if (_list.includes(id)) {
      return false;
    } else {
      _list.push(id);
      return true;
    }
  };
```

```
}
let firstLoad = isFirstLoad();
firstLoad(10); // true
firstLoad(10); // false
```

总结：函数和变量的作用域是在定义的时候就确定了的，和运行环境无关。

ES6

模块化

语法：import export

环境：使用 babel 编译ES6语法，模块化可以用 webpack 或者 rollup

常用功能：

1. let　const

   const 定义的变量无法修改

   ```
   let name = "yu";
   const age = 26;
   age = 24; // 报错，  Uncaught TypeError: Assignment to constant variable.
   ```

2. 块级作用域

   ```
   if (true) {
     var a = 0;
   }
   console.log(a); // 0

   if (true) {
     let b = 1;
     const c = 2;
   }
   console.log(b); // 报错，Uncaught ReferenceError: b is not defined
   console.log(c); // 报错，Uncaught ReferenceError: c is not defined
   ```

3. class 类

   ```
   class Parent {
     constructor(name) {
       this.name = name;
     }
     printName() {
       console.log(this.name);
     }
   }
   ```

4. 字符串模板

```
const name ='yu';
let user = `my name is ${name}`; // "my name is yu"
```

5. async await

```
async function getSource(id) {
  const res = await ce.get(id);
  console.log(res);
}
```

6. 解构赋值

```
let obj = {
  a: 1,
  b: 2
};

let { a, b } = obj;
console.log(a); // 1
console.log(b); //2
```

7. 箭头函数

箭头函数在定义时，this就确定了，不是运行时指定的。

普通函数中，this是运行时确定的，很容易出现奇怪的问题

```
function arrow() {
  let realObj = this; // {name: yu }
  let arr = [1, 2, 3];
  arr.map(function(item) {
    console.log(this); // window
  });
  arr.map(item => console.log(this)); // {name: yu }
}
arrow.call({ name: "yu" });
```

8. 函数默认参数

```
function getName(name = "yu") {
  console.log(name);
}
getName();  // yu
getName("gou"); // gou
```

9. Promise

```
function loadImg(src) {
  let promise = new Promise((resolve, reject) => {
    let img = document.createElement("img");
    img.onload = function() {
      resolve(img);
    };
```

```
    img.onerror = function() {
      reject(new Error("资源加载错误"));
    };
    img.src = src;
  });
  return promise;
}
let src = "https://qna.smzdm.com/201907/30/5d3fbd6ce29ad5187.jpg_a200.jpg";
loadImg(src)
  .then(img => {
    console.log("width", img.width);
    // 这儿返回了 img，后面的 then 才能接受到 img
    return img;
  })
  .then(img => console.log("height", img.height))
  .catch(err => console.log(err));
```

HTTP相关

HTTP 协议的特点：简单快速、灵活、无状态、无连接

简单快速：客户端请求服务的时候只需要发送方法和地址，而且 HTTP 比较简单，所以通信速度快。

灵活：HTTP 协议允许传输任意类型的数据，用 Content-Type 进行标记。

无状态：HTTP 协议是无状态协议，所以如果需要后续处理前面的信息，必须重传。另一方面，不需要先前信息，所以应答很快。

无连接：每次连接只处理一个请求，在服务器在处理完请求，并收到客户端的应答后就断开连接。可以节约传输时间。

HTTP 报文组成部分：请求报文、响应报文

请求报文：请求行、请求头、空行、请求体

请求行：请求方法、地址、HTTP 协议版本

请求头： key -value 值

响应报文：状态行、响应头、空行、响应体

状态行：状态码

响应头：key-value值

HTTP方法：GET 获取资源

POST 传输资源

PUT 更新资源

DELETE 删除资源

HEAD 获取报文首部

POST 和 GET 方法的区别：

1. GET 请求能够被浏览器主动缓存，POST 不会，除非手动设置
2. GET 请求的参数放在 url 中，不安全，POST 放在请求体中
3. GET 请求在 url 中传递参数是有长度限制的，POST 没有

4. GET 方法在浏览器回退的时候是无害的，POST 会再次提交请求
5. GET 产生的 url 地址能够被浏览器收藏，POST 不能

HTTP 状态码：

1. 1xx：请求已接受，继续处理
2. 2xx：成功 200 成功 / 206 客户端发送了一个带有range头的GET请求，服务器完成了他
3. 3xx：重定向 301 永久重定向 / 302 临时重定向
4. 4xx：客户端错误 403 禁止访问 / 404 资源不存在
5. 5xx：服务器端错误

持久化连接：

默认情况下，HTTP 协议使用"请求-应答"模式，每个请求/应答客户端和服务器端都要重新建立连接。但是使用 keep-alive模式可以使客户端和服务端持久化的连接。当对服务器有后续请求的时候，可以避免重新建立连接。

HTTP 1.1 支持。

管线化：需要通过持久化连接。持久化有一个缺点是请求和响应是顺序执行的，只有在请求1的响应接受到以后才能发送请求2.而管线化则不需要等到响应返回就可以继续发送请求。

1. 只有 GET 和 HEAD 可以管线化，POST 有限制。
2. 初次连接不应该启动管线化，因为服务器端可能不支持。
3. 启动管线化之后，性能也不一定会有大幅度提升，chrome 和 firefox 默认是关闭了管线化的。

错误监控

前端错误分类： 1. 即时运行错误(js报错) 2. 资源加载错误

错误捕获方式：

1. 即时运行错误类：try...catch 、 window.onerror
2. 资源加载错误：
    1. 监听相应dom的onerror事件，比如img的onerror
    2. 使用performance.getEntries()获取资源列表，配合 document.getElementsByTagName('xx'),间接获取资源加载错误
    3. 通过捕获window.onerror，资源加载错误不会冒泡到window，所以只能在捕获阶段去处理，需要 window.addEventListener('error',()={console.log('出错了')},**true**)；最后一个参数配置为true

跨域js报错： 默认只能拿到 script error，如果要拿到详细信息，需要以下两步

1. 客户端script标签添加 crossorigin 属性
2. 服务器端响应js文件时添加 Access-Control-Allow-Origin ： * 或者是js运行的域名

错误上报：

1. 使用Ajax 请求
2. 使用Image 对象，一般都采用这种方法

```
(new Image()).src = 'http://xxx.com?info=xx' // 错误上报地址
```

什么是同源策略及限制：

如果两个页面拥有相同的协议、域名、端口，那么这两个页面就是同一个源，只要这三者有一个不同，那么都是不同源的。

同源策略是浏览器的一个安全机制，不同源的脚本在没有明确授权的情况下，不能读写对方的资源。

限制： Ajax 请求不能发送

DOM 无法获取

cookie、localstorage、indexdDB无法读取。

前后端如何通信：

1. Ajax ---- 不能跨域
2. websocket
3. CORS 跨域资源共享

如何创建Ajax：

```
// 创建对象
let xhr = new XMLHttpRequest();
// 准备发送请求
xhr.open("get", "http：//m.baidu.com");
// 发送请求
xhr.send({ name: "yu" });
// 监听回调
xhr.onreadystatechange = () => {
  // 为 4 代表服务器返回数据可用
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log("xhr 请求成功");
    }
  }
};
```

跨域通信的几种方式： JSONP、websocket、Hash、postMessage、CORS

1. JSONP

   script标签是不受跨域限制的，所以可以动态的给html页面添加一个script标签，来获取数据。

   1. 首先创建一个script 标签，并且设置src的值，包含服务器的地址、callback函数名称、需要向服务器端发送的数据等。
   2. 在window中注册一个与发送的callback名称相同的函数。
   3. 将创建好的标签添加到html中，此时将会向服务器发送请求。
   4. 服务器收到请求后，会远程调用注册在window中的callback函数，并将响应数据作为参数传递。
   5. 在callback函数中处理接受到的数据，收到并处理完数据后，需要将添加的script标签和注册在window中的函数删除，避免继续占用内存。
2. websocket

```
let ws = new WebSocket("ws://m.biadu.com");
ws.onopen = () => {
  ws.send({ name: "yu" });
};
ws.onmessage = e => {
  const data = e.data;
  console.log("ws 接受数据", data);
  ws.close();
};
ws.onclose = () => {
  console.log("ws 关闭了");
};
```

3. Hash　浏览器通过更改hash值是不会刷新的，所以可以通过更改hash值来进行通信。

   A页面通过iframe标签引入了B页面，可以通过hash，将数据从a页面传递到b页面

4. postMessage 也是通过iframe实现

```
// a 页面通过iframe引入B页面，先获取B页面
let receiver = document.getElementById('receiver').contentWindow;
// 发送数据
receiver.postMessage(data, B页面地址);

// B页面中，监听message事件
window.addEventListener('message', (e) => {
    const data = e.data;
    console.log(data)
})
```

5. CORS 通过服务器端进行配置

渲染类

什么是 DOCTYPE 及作用：

DOCTYPE 就是网页的文档类型声明，用来告诉浏览器应该用什么渲染方式来呈现网页。

H5 的 DOCTYPE :

浏览器渲染过程：

1. 解析HTML，得到 DOM tree
2. 解析CSS ，得到 CSSOM tree
3. 将两者结合，得到 render tree
4. 进行布局 layout，根据render tree 计算每一个元素的大小和位置
5. 绘制 painting，将元素呈现到显示器上

1、2、3 非常快，4、5 比较慢

重排 reflow 发生在4，重绘 repaint 发生在5.

重排只的是重新计算节点的大小和位置，所以一定会导致重绘。重绘不一定会导致重排，比如改背景颜色等。

重排一定会导致重绘，但是重绘不一定会导致重排

会导致重排的操作：对dom 进行增删改、修改css，比如修改元素宽高、修改浏览器窗口大小

会导致重绘的操作：修改dom 、 修改css

优化重排：比如在往ul 中添加好几条li时，先使用 document.fragment来包裹这些li，然后一次性添加到ul中。

页面性能

提升页面性能的方法：

1. 资源压缩与合并，减少HTTP请求。比如使用tinyPNG压缩图片，使用webpack打包文件、服务器端开启Gzip等。
2. 使用浏览器缓存，这是提高性能最重要的方式。分为强缓存和协商缓存两种。
    1. 强缓存是通过 Expires 和 Cache-Control 字段控制的，浏览器不需要请求，通过判断资源的过期时间，如果资源没有过期就直接从缓存中取。如果这两个字段有冲突，以 Cache-Control为准。
    2. 协商缓存是通过 Last-Modified 和 if-Modified-Since 字段控制，浏览器需要向服务器端发送请求，如果服务器端返回说本地资源没有过期，就使用缓存。
3. 非核心资源延迟加载。
    1. 通过js动态添加标签
    2. 给script标签添加defer属性
    3. 给script标签添加async属性

       常规的script标签会让浏览器下载并执行后再执行后面的html解析

       defer属性的script标签会在html解析完成后，顺序执行。

       async属性的script标签会在js文件下载完成后立即执行，和顺序无关，此时html不一定解析完了
4. 使用CDN
5. DNS 预解析

```
<-- 预解析页面中的a标签链接，特别是https -->
<meta http-equiv="x-dns-prefetch-control" content="on">
<-- 预解析DNS -->
<link rel="dns-prefetch" href="//m.baidu.com">
```

MVVM

1.如何理解 MVVM

mvvm是mvc的改进版，由model 数据模型、view 视图、viewModel 组成，vm是m 和 v 的桥梁，m通过数据绑定将数据呈现到view 上，view 通过 dom listenner 将用户操作反馈到model上，再通过数据绑定展现到view层，中间通过 viewModel实现。