1. 基本类型有哪几种?

基本数据类型：`undefined`，`null`，`string`，`number`，`bool`，`symbol(es6新增)`。

symbol的作用：

symbol 是ES6 新加的一种新的基本数据类型。 表示独一无二的。

- 作为属性名：

```
const mySymbol = symbol();
const obj = {
  [mySymbol]: "hahah"
}
```

- 作为常量（唯一的）

6SESymbol

3. 复杂(引用)数据有哪几种?

引用数据类型：`Function`，`Object`，`Array`，`Date`

4. 基本数据类型和复杂数据类型存储有什么区别?

基本数据类型存储在`栈内存`，存的是值。

引用数据类型的值是存在`堆内存`， 地址(就是只想这个堆的地址)存在`栈内存`。当我们把对象赋值给另外一个变量的时候，复制的是地址，指向同一块内存空间，当其中一个对象改变时，另一个对象也会变化。

栈内存主要用于存储各种***基本类型***的变量，包括Boolean、Number、String、Undefined、Null...以及***对象变量的指针***，这时候栈内存给人的感觉就像一个线性排列的空间，每个小单元大小基本相等，栈内存中的变量一般都是已知大小或者有范围上限的，算作一种简单存储.

而***堆内存***主要负责像对象Object这种变量类型的存储，堆内存存储的对象类型数据对于大小这方面，一般都是未知的，（所以这大概也是为什么null作为一个object类型的变量却存储在栈内存中的原因。

使用new关键字初始化的之后是不存储在栈内存中的。为什么呢？new大家都知道，根据构造函数生成新实例，这个时候生成的是**对象**，而不是基本类型。再看一个例子

```
var a = new String('123')
var b = String('123')
var c = '123'
console.log(a==b, a===b, b==c, b===c, a==c, a===c)
>>> true false true true true false
console.log(typeof a)
>>> 'object'
```

我们可以看到new一个String，出来的是对象，而直接字面量赋值和工厂模式出来的都是字符串。但是根据我们上面的分析大小相对固定可预期的即便是对象也可以存储在栈内存的，比如null，为啥这个不是呢？再继续看

```
var a = new String('123')
var b = new String('123')
console.log(a==b, a===b)
>>> false false
```

很明显，如果a，b是存储在栈内存中的话，两者应该是明显相等的，就像null === null是true一样，但结果两者并不相等，说明两者都是存储在堆内存中的，指针指向不一致。

5.null 是对象吗?

null 其实不是一个对象，尽管`typeof null`输出的是`object`,但是这其实是一个bug。在js最初的版本中使用的是32位系统，为了性能考虑地位存储变量的类型信息，`000`开头表示为对象类型，然而`null`为全0，故而`null`被判断为对象类型。编程语言最后的形式都是二进制，所以 JavaScript 中的对象在底层肯定也是以二进制表示的。如果底层有前三位都是零的二进制，就会被判定为对象。底层中 null 的二进制表示都是零。所以在对 null 的类型判定时，发现其二进制前三位都是零，因此判定为 object。

6. typeof 是否正确判断类型? instanceof呢？ instanceof 的实现原理是什么?

- typeof 是一个一元运算，放在一个运算数之前，运算数可以是任意类型。**它返回值是一个字符串，该字符串说明运算数的类型。**

```
// typeof 判断基本数据类型:都可以判断，除了null 意外
typeof(undefined) // "undefined"
typeof("shuliqi") // "string"
typeof(false) // "boolean"
typeof(12) // "number"
typeof(Symbol()) // "symbol"
typeof(null)  // "object"

// typeof 判断复杂数据类型：不能判断复杂数据类型
// 除了typeof 函数是 “function”之外，其他的都是返回“object”
typeof(Array())  // "object"
typeof(new Date()) //  "object"
typeof({}) // "object"
```

```
typeof(function() {}) // "function"
```

- **instanceof** 运算符用来测试一个对象在其原型链中是否存在一个构造函数的 `prototype` 属性。

  语法：`object instanceof constructor` 看这个object(对象)是否有构造函数的prototype属性

  参数：`object`（要检测的对象.）`constructor`（某个构造函数）

```
// instanceof 是用来检车一个对象在它的原型链上是有一个构造函数的prototype属性
// object instanceof constructor
// 参数：object(要检测的对象)， constructor（某个构造函数）

// 1.如果通过 字面量 的方式创建字符串，那么无法通过 instanceof 判断某个变量是否是字符串
let str2 = 'aaaa'
console.log(str2 instanceof String) // false
console.log(str2 instanceof Object) // false

// 2. 通过 new 方式，是可以使用 instanceof 判断 变量是否是字符串。
let str1 = new String('aaa')
console.log(str1 instanceof String) // true
console.log(str1 instanceof Object) // true
console.log(str1.__proto__ === String.prototype) // true
// 3. 复杂数据类型
const date = new Date();
console.log(date instanceof Date) // // true

const obj1 = new Object({ shu: 'asa' });
console.log(obj1 instanceof Object)  // true

const obj2 = { age: 12 };
console.log(obj2 instanceof Object) // true

const myFun1 = new Function();
console.log(myFun1 instanceof Function) //  true

const myFun2 = function() {};
console.log(myFun2 instanceof Function) //  true
```

- instanceof 的实现原理

```
// instanceof 的实现原理，A instanceof B
function  myInstanceof(A, B) {
  const O = B.prototype; // 构造函数B的prototype属性
  A = A.__proto__;       // 取 A 的隐形原型
  while(true) {
    if (A === 'null' ) {  //已经找到顶层
```

```
        return false;
      }
      if ( A === O) {  //当 O 严格等于 L 时，返回 true
        return true;
      }
      A = A.__proto__;
    }
}

const a = [1, 2];
console.log(myInstanceof(a, Array ));  // true
```

7. for of , for in 和 forEach,map 的区别。

- **for ... of**: 只要具有interator接口， 就可以使用for...of 遍历他的属性值。
  - 数组的遍历器接口只返回具有数字索引的属性(数组原生具备 iterator 接口, 默认部署了 Symbol.iterator 属性)
  - 对于普通的对象, for...of 结构不能直接使用，会报错，必须部署了 Iterator 接口后才能使用。
  - 可以中断循环
  - 可以使用for ...of 遍历的数据结构：数组，Map 和Set接口，类似数组的对象。Generator 对象, 字符串。

```
{
  let arr = ["shuliqi","name"];
  for (let item of arr) {
    console.log(item); // "shuliqi","name" 遍历的是属性值
  }
  arr.age = 12;
  for (let item of arr) {
    console.log(item); // "shuliqi","name"  输出的是属性值(age不是数字索引，不会被遍历)
  }
  for (let item of arr) {
    if (item === "name") {
      break;
    }
    console.log(item); // 最后只返回了“shuliqi”， 表示可以中断循环
  }
  let obj = { name: 'shu', age: 12};
  try{
    for (const item of obj) {
      console.log(item);
      // Uncaught TypeError: obj is not iterable, 抛错，因为普通对象没有 Iterator 接口
    }
  }catch(e) {
    console.log(e);
  }
```

```
}
```

- **for...in:**遍历对象自身和继承的可枚举的属性，遍历的是属性名，可以中断循环

```
{
  let arr = ['shu', "li", 12];
  for(let key in arr) {
    console.log(key); // 0,  1,  2 遍历的属性名
  }
  arr.foo = 1000;
  for(let key in arr) {
    console.log(key); // 0,  1,  2, foo 遍历的属性名
  }

  let arr = ['shu', "li", 12];
  for(let key in arr) {
    if (key === 2) {
      break;
    }
    console.log(key); // 0,  1,  2 遍历的属性名 , 表示可以中断循环
  }
  //  注意:(因为for in 可以遍历自身和继承的属性名，所以hasOwnProperty 是用来判断一个属性名是否是自身的属性)
  let object = { name:"shu", age: 12 };
  for (const key in object) {
    console.log(key); // name, age
  }
}
```

- **forEach**：只能遍历数组， 没有返回值（或认为返回值是undefined)）,不能中断

```
{
  let arr = ["shu", "li"];
  const result = arr.forEach((item, index) => {
    console.log(item, index); //  shu , 0 li, 1
    item = item + 1;
  });
  console.log(result, arr); // undefined ,  arr = ["shu", "li"];
  // 表明没有改变原数组除非我们手动操作改变数组，  没有返回值。

  arr.forEach((item, index) => {
    console.log(item, index);    //  shu , 0 li, 1, 表明不能中断循环
    if (index === 1) {
      return;
    }
  })
}
```

- **map:** 只能遍历数组， 返回修改后的数组， 不能中断

```
{
  const arr = ["shu", 'li'];
  const result = arr.map((item, index) => {
    return item + 'haha';
  });
  console.log(arr, result); // ["shu", "li"], ["shuhaha", "lihaha"]
  // 说明： 愿数组没有改变，返回改变之后的数组
}
```

- **Object.keys**：返回指定对象的的自身的可枚举属性的字符串数组

```
// Object.keys()
{
  const arr = ["shu", "li"];
  const result = Object.keys(arr);
  console.log(result); // ["0", "1"]

  const obj = {
    name: "shuliqi",
    age: 12,
  }
  const result2 = Object.keys(obj);
  console.log(result2); //  ["name", "age"]
}
```

8.如何判断一个变量是不是数组?

- 使用**Array.isArray()**, 如果返回true, 说明是数组。
- 使用 **...instanceof Array**, 如果返回true, 说明是数组。
- 使用**Object.prototype.toString.call([1,2])**，如果返回”[object Array]“, 说明是数组。
- 使用**constructor**如果 arr.cunstructor = Array，说明是数组。constructor属性返回构造该对象的数组函数的引用 注意：不是很准确， 因为有时候我们可以设置一个对象的constructor属性

```
const arr = ['shu', 'li'];
console.log(Array.isArray(arr)); // true
console.log(arr instanceof Array); // true
console.log(Object.prototype.toString.call(arr) === "[object Array]"); //
true
console.log(arr.constructor === Array) // true
// constructor  不是很准确， 因为有时候我们可以设置一个对象的constructor属性
const obj = {};
console.log(obj.constructor === Array); // false
obj.constructor = Array;
console.log(obj.constructor === Array); // true
```

9. 类数组和数组的区别是什么?

- 类数组具有length属性,其他属性(索引)是非负整数
- 不具备数据所有的方法

类数组：函数的参数(arguments), Dom对象列表，document.getquerySelectAll(), (getelementByTagName, document.querySelectorAll等)的都是类数组

类数组转数组的方法

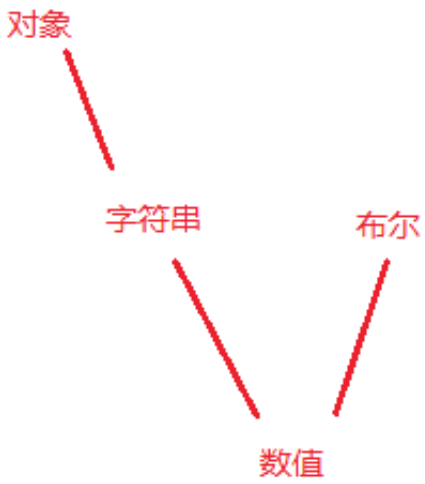
- Array.prototype.slice.call(likeArr)
- Array.from(likeArr)。Array.from可以将类似数据的对象和具有inerator接口的对象转换转换成真正的数数组。
- [...likeArr]。只要具有interator接口的对象，都能使用扩展操作符转换成真正的数组。

```
// 类似数组对象传承数组的方法
{
  function test() {
    const likeArr = arguments;
    const arr1 = Array.prototype.slice.call(likeArr);
    const arr2 = [...likeArr];
    const arr3 = Array.from(likeArr);
    console.log(likeArr instanceof Array); // false
    console.log(arr1 instanceof Array); // true
    console.log(arr2 instanceof Array); // true
    console.log(arr3 instanceof Array); // true
  }
  test()
}
```

10. == 和 === 有什么区别?

=== 是不需要类型转换的，只要类型相同和值相同 才返回true

== 如果两者的类型不同，则需要类型转换。转换的规则如下:




```
{
// == 隐形转换
  {
    // 1.对象和布尔值进行比较时，对象先转换为字符串，然后再转换为数字，布尔值直接转换为数字
    console.log([] == true);


    // 结果：false
    // 转换流程
    String([]); // ""
    Number(""); // 0
    Number(true); // 1
    console.log(0 == 1); // false
  }
  {
    // 2.对象和字符串进行比较时，对象转换为字符串，然后两者进行比较
    console.log([1,2,3] == "1,2,3")


    // 结果：true
    // 转换流程
    String([1,2,3]); // '1,2,3'
    console.log("1,2,3" == "1,2,3"); //  true
  }
  {
    // 3.对象和数字比较时，对象转化为字符串,然后转换为数字，再和数字进行比较
```

```
    console.log([1] == 1);


    // 结果为true
    // 转换规则：
    String([1]); // "1"
    Number("1"); // 1
    console.log(1 == 1); //  true
}
{
    // 4.字符串和数字比较时，字符串转换为数字
    console.log('1' == 1);



    // true
    //  转换流程：
    String([1]); // "1"
    Number("1"); // 1
}

{
    // 5.字符串和布尔值进行比较时，二者全部转换成数值再比较
    console.log('1' == true)



    // true
    Number("1"); // 1
    Number(true); //  1
}
{
    // 6.布尔值和数字进行比较时，布尔转换为数字
    console.log(true == 1);



    // true
}
{
    // 7.特殊的比较
    console.log(undefined == undefined); // true
    console.log(null == null); // true
    console.log(null == undefined); // true
    console.log(+0 === -0);// true
    console.log(Number(null)); // 0
    console.log(Number(undefined)); // NaN
```

```
    console.log(NaN === NaN); // false

  }
  {
    // 思考: [] == ![]
    console.log([] == ![]);
    String([]); // ""
    Number(""); // 0
    // ![] 引用数据类型专程布尔值都是true,  因此 ![] 是false
    // false 转成Number 是0
    // 因此 0 == 0; tru

  }
  {
    //  7.复杂数据类型的比较,  比较的是引用地址
    const obj1 = {
      name: 'shulii',
    }
    const obj2 = obj1;
    obj2.name = 'haha';
    console.log(obj1 == obj2, obj1 === obj2 ); // true, true
  }
}
```

Object.is. 是用来判断两个值是否相等。

Object.is 也不会进行类型的转换, 与 === 的区别是:

```
console.log(+0 === -0);// true
console.log(Object.is(+0, -0)); // false
console.log(NaN === NaN); // false
console.log(Object.is(NaN, NaN)); // true
```

11.ES6中的class和ES5的类有什么区别?

```
{
// ES5
  function test(name, age) {
    this.name = name;
    this.age = age;
  }
  test.prototype.getName = function() {
    return this.name
  }
  var myFun = new test("shuliqi", 12);
}
```

```
{
  // ES6
  class test {
    // 构造函数
    constructor(name, age) {
      this.name = name;
      this.age = age
    }
    // 添加方法，  注意ES6的calss内部只能允许添加方法，  不允许添加属性
    getName() {
      return this.name;
    }
  }
  const myFun = new test('shuloqo', 23);
}
```

- ES6 calss 内部定义的所有方法都是不可枚举的。

```
{
  // ES5
  function test(name, age) {
    this.name = name;
    this.age = age;
  }
  test1.prototype.getName = function() {
    return this.name
  }
  console.log(Object.keys(test.prototype));  // ["getName"]
}

{
  // ES6
  class test {
    // 构造函数
    constructor(name, age) {
      this.name = name;
      this.age = age
    }
    // 添加方法，  注意ES6的calss内部只能允许添加方法，  不允许添加属性
    getName() {
      return this.name;
    }
  }
  console.log(Object.keys(test.prototype)); // []
}
```

- ES6 calss 必须使用new 调用, 不使用会报错;

```
class test {
  // ...
}

// 报错
var point = test(2, 3);

// 正确
var point = new test(2, 3);
```

- ES6 calss 不存在变量提升

```
new test(); // 报错，  类不存在变量提升（hoist），这一点与 ES5 完全不同
class test {};


// 不能提升的原因
{
  let Foo = class {};
  class Bar extends Foo {
  }
}
//上面的代码不会报错，因为Bar继承Foo的时候，Foo已经有定义了。但是，如果存在class的提升，
// 上面代码就会报错，因为class会被提升到代码头部，而let命令是不提升的，所以导致Bar继承Foo的时候，Foo还没有定
```

- ES6 class 默认即是严格模式
- ES6 class 子类必须在`constructor`方法中调用`super`方法，这样才有this对象;ES5中类继承的关系是相反的，先有子类的this，然后用父类的方法应用在this上。

```
{
  class test {
    constructor (name,age) {
      this.name = name;
      this.age = age;
    }
  }
  class test1 extends test {
    constructor () {}
  }
  myfun = new test1("shuliqi", 1); //  ReferenceError
}

{
  class test {
    constructor (name,age) {
```

```
        this.name = name;
        this.age = age;
      }
    }
    class test1 extends test {
      constructor () {
        super(name, age)
      }
    }
    myfun = new test1("shuliqi", 1); //  不会报错
}
```

- 类的继承

```
{

    class People {
      constructor() {
        this.age = 38;
      }
    }
    class MyPeople extends People {
      constructor() {
        super();
      }
      static get() {
        this.age = 10;
        super.age = 20;
        console.log(super.age);
        console.log(this.age);

      }
    }
    MyPeople.get();
    MyPeople.age;
```

```
  // undefined 20
}



{
  class People {
    constructor() {
      this.age = 38;
    }
  }
  People.sex = 30;

  class MyPeople extends People {
    constructor() {
      super();
      this.age = 10;
      super.age = 20;
      console.log(super.age);
      console.log(this.age);

    }
  }
  const people = new MyPeople();







  // undefined 20
  // 原因：super.age 中的super是在普通方法中使用的，super指向分类的原型。父类的原型上没有age属性
}



{
  class A {
    p() {
      return 2;
    }
  }

  class B extends A {
    constructor() {
      super();
      console.log(super.p());
    }
```

```
  }

  let b = new B();



  // 2
}

{
  class A {
    constructor() {
      this.p = 2;
    }
  }

  class B extends A {
    get m() {
      return super.p;
    }
  }

  let b = new B();
  b.m




  // undefined
}

{
  class Point {
  }
  class ColorPoint extends Point {
  }
  const shu = new ColorPoint()
}


{
  class A {}
  A.prototype.x = 2;

  class B extends A {
    constructor() {
      super();
      console.log(super.x)
```

```
    }
  }

  let b = new B();



  // 2
}



{
  class A {
    constructor() {
      this.x = 1;
    }
    print() {
      console.log(this.x);
    }
  }

  class B extends A {
    constructor() {
      super();
      this.x = 2;
    }
    m() {
      super.print();
    }
  }

  let b = new B();
  b.m()



  // 2
}


{
  class A {
    constructor() {
      this.x = 1;
    }
  }
```

```
  class B extends A {
    constructor() {
      super();
      this.x = 2;
      super.x = 3;
      console.log(super.x);
      console.log(this.x);
    }
  }

  let b = new B();




   // undefined 3
}


{
  class Parent {
    static myMethod(msg) {
      console.log('static', msg);
    }

    myMethod(msg) {
      console.log('instance', msg);
    }
  }

  class Child extends Parent {
    static myMethod(msg) {
      super.myMethod(msg);
    }

    myMethod(msg) {
      super.myMethod(msg);
    }
  }

  Child.myMethod(1);

  var child = new Child();
  child.myMethod(2);
```

```
    // static 1
    // instance 2
}




{
  class A {
    constructor() {
      this.x = 1;
    }
    static print() {
      console.log(this.x);
    }
  }

  class B extends A {
    constructor() {
      super();
      this.x = 2;
    }
    static m() {
      super.print();
    }
  }

  B.x = 3;
  B.m()



  // 3
}



{
  class A {}

  class B extends A {
    constructor() {
      super();
      console.log(super);
    }
  }
```

```
  // 报错
}



{
  class A {}

  class B extends A {
    constructor() {
      super();
      console.log(super.valueOf() instanceof B);
    }
  }

  let b = new B();




  // true
}



{
  var obj = {
    toString() {
      return "MyObject: " + super.toString();
    }
  };

  obj.toString();




  // MyObject: [object Object]
}
```

12. 数组的哪些API会改变原数组?

- **join()**： 将数组的元素的转换成字符串并拼接，最后返回这个字符串。与之相反的是： split()： 将字符串转成数组

```
// join()
const arr = [ 6, 5, 9];
console.log(arr); //  [6, 5, 9] 原数组不会改变
console.log(arr.join());  // "6,5,9"
console.log(arr.join('-')) // "6-5-9"
//  与之相反的是： split()： 将字符串转成数组
const str = '6-5-9';
console.log(str.split('-')); // [ 6, 5, 9]
```

- **sort()**:将数组进行排序，默认是按照字母表排序 （会改变原数组）

```
// sort()
{
  const arr = [ 6, 5, 9];
  const result = arr.sort();
  console.log(arr); //  [5, 6, 9] // 会改变原数组
  console.log(result); // [5, 6, 9] // 进行排序了
}

{
  // 按照从小到大排序
  const arr = [ 6, 5, 9];
  const result = arr.sort((a, b) => a - b);
  console.log(result)
}

{
  // 按照从大到小排序
  const arr = [ 6, 5, 9];
  const result = arr.sort((a, b) => b - a); // [9, 6, 5]
  console.log(result)
}

{
  // 不区分字母大小排序
  const arr = [ "c", "B", "a", "f", "E"];
  const result = arr.sort((a, b) => a.toLowerCase() > b.toLowerCase() ? 1 :
-1);
  // toLowerCase(): 将字母转换成小写
  // toUpCase(): 将字母转换成大写
```

```
  console.log(result); //  ["a", "B", "c", "E", "f"]
}
```

- **reverse()**:将数组进行倒序 （会改变原数组）

```
// reverse()
const arr = [5, 4, 3, 2, 1];
const result = arr.reverse();
console.log(arr); // [1, 2, 3, 4, 5] 说明会改变原数组
console.log(result); // [1, 2, 3, 4, 5]
```

- **concat()**:链接两个数组， 形成一个新的数组，旧的数组不变（不会改变原数组）

```
// concat()
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const result = arr1.concat(arr2);
console.log(arr1); // [1, 2, 3] 说明不会改变旧数组
console.log(arr2); // [4, 5, 6] 说明不会改变旧数组
console.log(result); // [1, 2, 3, 4, 5, 6]
```

- **slice()**:返回数组的一个片段，接受两个参数， 第一个参数表示指定片段的起始位置(数组的下标)，第二个参数表示结束位置（但不包含元素)。如果是负数的话， 表示倒数。 （不会改变）

```
// slice()
const arr = [1, 2, 3, 4, 5];
const result = arr.slice(1, 2);
const result2 = arr.slice(1, -1)
console.log(arr); // [1, 2, 3, 4, 5]; //  不会改变原数组
console.log(result); // [2]

console.log(result2); // [2, 3, 4]
```

- **splice()**: 接受至少2个参数， 第一个：删除元素的起始位置(下标)，第二个参数：删除的个数，之后的数据就是插入的。最后返回被删除的元素形成的数组。 （会改变原数组）

```
{
  // splice()
  const arr = [1, 2, 3, 4, 5];
  const result = arr.splice(1,2, 3);
  console.log(arr); // [1, 3, 4, 5] // 表示会改变原数组
  console.log(result); // [2, 3]
}
```

- **shift()**： 删除数组的第一个元素，返回删除的元素。 （会改变原数组）

```
{
 // shift()
 const arr = [1, 2, 3, 4, 5];
 const result = arr.shift();
 console.log(arr); // [2, 3, 4, 5] // 会改变原数组
 console.log(result); // 1 // 返回被删除的元素
}
```

- **unshift()**：添加元素的到数组的第一位， 返回新数组的长度。（会改变原数组）

```
{
 // shift()
 const arr = [1, 2, 3, 4, 5];
 const result = arr.unshift(5);
 console.log(arr); // [5, 1, 2, 3, 4, 5] // 会改变元素组
 console.log(result); // 6 // 返回新数组的长度
}
```

- **pop()**:删除元素的最后一个元素。并返回删除的元素。（会改变原数组）

```
{
  // pop
  const arr = [1, 2, 3, 4, 5];
  const result = arr.pop();
  console.log(arr); // [1, 2, 3, 4]
  console.log(result); // 5
}
```

- **push()**：在数组的尾部添加一个元素。并返回新的数组的长度。（会改变原数组）

```
{
  // push
  const arr = [1, 2, 3, 4, 5];
  const result = arr.push(6);
  console.log(arr); // [1, 2, 3, 4, 5, 6]
  console.log(result); // 6
}
```

- **forEach()**：遍历数组的每个元素， 每个元素调用一个函数。函数第一个参数：该元素本身，第二个元素：元素的下标。第三个元素：数组本身。没有返回数。

```
{
  // forEach
  const arr = [1, 2, 3, 4, 5];
  const result = arr.forEach((item, index, shelf) => {
    console.log("item", item);
    console.log("index", index);
    console.log("index", shelf);
  })
  console.log(result); // undefined  表示没有返回值
}
```

- **map():**遍历数组的每个元素， 每个元素调用一个函数。函数第一个参数：该元素本身，第二个元素：元素的下标。第三个元素：数组本身。有返回值，不会改变原来数组。

```
{
 // map
 const arr = [1, 2, 3, 4, 5];
 const result = arr.map((item, index, shelf) => {
  return item + 1;
 })
 console.log(arr); // [1, 2, 3, 4, 5]; 表示原数组没有改变
 console.log(result); // [2, 3, 4, 5, 6]; 表示有返回值
}
```

- **filter()**：返回数组的一个子集。

```
{
  // filter
  const arr = [1, 2, 3, 4, 5];
  const result = arr.filter((item) => {
   return item  > 2;
  })
  console.log(arr); // [1, 2, 3, 4, 5]; 表示原数组没有改变
  console.log(result); // [3, 4, 5]; 表示有返回值
}
```

- 数组的方法

```
// join() ---> 不会改变
const arr = [1, 2, 3];
console.log(arr.join())
console.log(arr.join('-'))
console.log(arr); // 不会改变原数组
```

```
const str = "1,2,3";
console.log(str.split(','))


// sort() 排序，默认是按字母表排序----会改变原数组

const arr = [4, 1, 2, 3]
const result = arr.sort();
// console.log(arr, result);
// 从小到达排序
arr.sort((a, b) => a -b)
console.log(arr);
// 从大到小
arr.sort((a, b) => b - a)
console.log(arr);
// 不区分字母大小
const arr = ['f', 'G', 'B', 'a', 'C',];
arr.sort((a, b) => a.toLowerCase() > b.toLowerCase() ? 1 : -1)
console.log(arr);

// reverse() 倒叙 ---> 会改变数组
const arr = [1,2,3];
arr.reverse();
console.log(arr)

// [3,2,1]


// concat() ---> 链接数组-- > 不会改变原数组
const a1 = [1,2], a2 = [3,4,5];
const newArr = a1.concat(a2);
console.log(newArr, a1, a2);


// [1,2,3,4,5]   [1,2]  [3,4,5]



// slice(index1, index2)// 返回数组的一个片段，第一个参数，起始下标
// 第二个参数，终止下标（但不包含改元素）。不会改变原数组
const a = [1,2,3,4,5,6];
const arr = a.slice(1,2);
console.log(a, arr);


// [1,2,3,4,5,6] [2]
```

```
// splice(), 删除和添加元素，第一个参数，是删除元素的起始位置
// 第二个参数是删除的个数
// 之后的参数都是添加的元素。从删除的地方开始添加
// 返回被删除的元素的数组
// 会改变原数组
const a = [1,2,3,3,4];
const deletArr = a.splice(1,1,5,6,7);
console.log(deletArr);
console.log(a);


 // [2]  [1,5,6,7,3,3,4]




// push()  像数组末尾添加元素，返回新新的数组的长度--改变原数组
const a = [1,2];
const lenth = a.push(3);
console.log(lenth);
console.log(a);

// 3 [1,2,3]



// pop()；  数组的末尾删除一个元素，返回被删除的元素，  会改变原数组
const a = [1,2,3,4];
const del = a.pop();
console.log(a, del)
// [1,2,3] 4


// unshift() 向头部添加一个元素，返回数组的长度，会改变原数组
const a = [2,3,4];
const len = a.unshift(1);
console.log(a, len);

// [1,2,3,4] 4


// shift() 头部删除一个元素，  返回删除的元素，会改变原数组
const a = [1,2,3];
const de = a.shift();
```

```
console.log(de, a);
// 1 [2,3]


// forEach() 编辑数组，  没有返回值，不改变原数组
```

13.let、const 以及 var 的区别是什么?

- let 和 const 定义的变量不会出现变量提升，var定义的变量会出现变量提升。
- let 和 const 是js中的块级作用域。
- let 和 const 不能重复声明（会抛出错误）
- let 和const 没有定义就使用会出现暂时性死区，而 var 不会。
- const 声明的变量只是一个只读常量，如果声明的是一个对象， 那么就不能改变对象的引用地址。

14.在JS中什么是变量提升？什么是暂时性死区?

变量提升：变量提升是指在变量声明之前就可以使用，值为undefined。

暂时性死区：在代码块中，使用let/const声明的变量之前，这些变量是不可用的（会抛出错误），这种在语法上就是成为暂时性死区。

```
typeof x;  // ReferenceError(暂时性死区，抛错)
let x;
```

```
typeof y; // 值是undefined,不会报错
```

暂时性死区的本质：只要进入到当前作用域中，所要使用的变量就已经存在了， 但是不可获取，只有等到变量声明之后才能获取和使用。

15.如何正确的判断this? 箭头函数的this是什么?

https://shuliqi.github.io/shuliqi.github.io/2018/07/02/%E5%85%B3%E4%BA%8Ethis%E7%9A%84%E6%8C%87%E5%90%91%E9%97%AE%E9%A2%98/

16.词法作用域和this的区别

- 词法作用域是由你在写代码的时候将变量和块级作用域写在哪里来决定的
- this是在绑定的时候确定的。this指向什么，完全取决于函数在哪里调用。

17.谈谈你对JS执行上下文栈和作用域链的理解。

执行上下文就是当前 JavaScript 代码被解析和执行时所在环境, JS执行上下文栈可以认为是一个存储函数调用的栈结构，遵循先进后出的原则。

- JavaScript执行在单线程上，所有的代码都是排队执行。
- 一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。
- 每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行-完成后，当前函数的执行上下文出栈，并等待垃圾回收。
- 浏览器的JS执行引擎总是访问栈顶的执行上下文。
- 全局上下文只有唯一的一个，它在浏览器关闭时出栈。

作用域链: 无论是 LHS 还是 RHS 查询，都会在当前的作用域开始查找，如果没有找到，就会向上级作用域继续查找目标标识符，每次上升一个作用域，一直到全局作用域为止。

注：如果查找的目的是对变量进行赋值，那么就会使用LHS 查询；如果目的是获取变量的值，就会使用RHS 查询

18.什么是闭包？闭包的作用是什么？闭包有哪些使用场景？

闭包就是能够访问其他函数内部变量的函数；j s语言特有的“链式作用作域“， 子级对象对一层一层的向上寻找父级的变量， 所以父级的变量子级都能访问， 反之，父级却不能访问到子级的变量。 那要是想要访问到怎么办呢？ 那就只能在子级里面在定一个函数， 然后return出来，那么 就可以访问了局部变量了。

- 闭包的优点：可以重复使用变量，并且不会造成变量污染，变量可以保持存在内存里面。
- 闭包的缺点：占用很多的内存，会导致网页性能变差，在IE下容易造成内存泄露。在外面也能改父级的变量。

http://www.ruanyifeng.com/blog/2009/08/learning_javascript_closures.html

19 谈谈原型链的理解

Javascript 里面每一个对象都有自己的原型prototype， 这个原型里面有可以继承的的属性和方法。而且每一个对象都有自己的**proto**这个隐原型只想自己的原型（prototype）。而原型又有自己的隐原型，只想自己的原型。就这样一直向上查找，直到最后的原型的隐原型指向null。也就是达到了原型链的终点。

https://javascript.ruanyifeng.com/oop/prototype.html

19.call、apply有什么区别？call,aplly和bind的内部是如何实现的?

call 和 apply 的功能相同，区别在于传参的方式不一样:

- call(对象, a,b,c,)

  第一个参数是绑定的this值

  第二个参数之后，都是参数

  立即调用

- apply(对象, [1,2,3])

  第一个参数就是绑定的this值

  第二参数是一个数组 参数数组

  立即调用

- bind(对象)

  第一个参数是绑定的this值

  第二个以后的参数都是实参

  不会立即调用， 而是返回一个新的函数。供之后调用

  call 的实现

  - 新加一个函数， 让当前调用的函数的数的this指向增新的函数
  - 执行新函数
  - 删除新函数
  - 返回结果

```
Function.prototype.myCall = function (thisObj, ...args) {
  if (typeof this !== "function") {
    throw new Error("错误");
  }
  thisObj.fn = this; // (this就是当前调用的函数)
  const result = thisObj.fn(...args); // 执行新加的函数
  delete thisObj.fn;
  return result;
};
const obj = {
  name: "shuliqi",
};
function getPerosion(age) {
  console.log(this.name, age);
}
getPerosion.myCall(obj, 12); // shuliqi 12
```

https://www.jianshu.com/p/af945ea77b44

```
// 实现的原理
const obj = {
  name: 'shuliqi',
  getName: function() {
    console.log(this.name); // shuliqi
  }
```

```
}
obj.getName(); // this的指向是隐式指向，  指定调用它的函数


Function.prototype.MyCall = function(thisObj, ...args) {
  const fn = Symbol('函的函数名'); // 为了防止与obj里面的函数重名
  thisObj[fn] = this; // this就是指向当前调用MyCall方法的函数;
  thisObj[fn](...args); // 调用call的时候是会立即调用的，所以执行
  delete thisObj.fn; // 记得删除新家的函数
}
const obj = {
  name: 'shuliqi'
}
function getName(age) {
  console.log(this.name, age);
}
getName.MyCall(obj, 18)
```

apply的实现

和call 实现的差不多， 只是传入的参数不同而已

```
  Function.prototype.myCall = function (thisObj, args) {
    if (typeof this !== "function") {
      throw new Error("错误");
    }
    thisObj.fn = this; // (this就是当前调用的函数)
    let result;
    if (args) {
      result = thisObj.fn(args);
    } else {
      result = thisObj.fn();
    }
    delete thisObj.fn;
    return result;
  };
  const obj = {
    name: "shuliqi",
  };
  function getPerosion(age) {
    console.log(this.name, age);
  }
  getPerosion.myCall(obj, [12, 13]); // shuliqi [ 12, 13 ]
}
```

https://www.jianshu.com/p/af945ea77b44

```
Function.prototype.MyApply = function(thisObj, args) {
  const fn = Symbol();
  thisObj[fn] = this;  // this就是指向当前调用MyApply方法的函数;
  thisObj[fn](...args) // 调用MyApply的时候是会立即调用的，所以执行
  delete thisObj[fn];  // 记得删除新加的函数
}

const obj = {
  name: 'shuliqi'
}
function getName(age) {
  console.log(this.name, age);
}
getName.MyApply(obj, [18])
```

bind 的实现

```
  Function.prototype.myBind = function (thisObj, ...args1) {
    if (typeof this !== "function") {
      throw Error("调用的应该是一个函数");
    }
    const _this = this;
    function Fn() {}
    Fn.prototype = this.prototype;
    let bound = function (...args2) {
      _this.apply(this instanceof Fn ? this : thisObj,
args1.concat(args2));
    };
    bound.prototype = new Fn();
    return bound;
  };
  const obj = {
    name: "shuliqi",
  };
  function test(name) {
    console.log("my is name:", this.name);
```

```
    this.age = 12;
    this.getPerson = function () {
      console.log(name, this.age);
    };
}
  const newTest = test.myBind(obj, "shuliqi");
  newTest(); // my is test
  const myNewTest = new newTest();
  myNewTest.getPerson(); // shuliqi 12
```

https://blog.csdn.net/tangzhl/article/details/79669461

20.什么是函数柯里化？实现 sum(1)(2)(3) 返回结果是1,2,3之和

https://shuliqi.github.io/archives/page/2/

21.截流和防抖函数

https://shuliqi.github.io/2018/04/16/Debounce%E5%92%8CThrottle%E7%9A%84%E5%8E%9F%E7%90%86%E5%8F%8A%E5%AE%9E%E7%8E%B0/

22.继承

原型链继承：

特点：实例是子类的实例也是父类的实例，父类新增的原型方法/属性，子类都能够访问，并且原型链继承简单易于实现

缺点：

1、新实例无法向父类构造函数传参。
2、继承单一。
3、所有新实例都会共享父类实例的属性。（原型上的属性是共享的，一个实例修改了原型属性，另一个实例的原型属性也会被修改！）

```
// 父级构造函数
const parent = function () {
  this.age = 12;
  this.getName = function () {
    console.log(this.name);
  };
};
// 子级构造函数
const child = function () {
  this.name = "shuliqi";
};
```

```
// 原型链继承
child.prototype = new parent(); // 重要的点：让新实例的原型等于父类的实例

const person = new child();//. 构造实例

person.getName(); // shuliqi
console.log(person.age); // 12
console.log(person instanceof parent); // true
```

构造函数方式继承：

```
// 父级构造函数
  const parent = function (age) {
    this.age = age;
    this.getName = function () {
      console.log(this.name);
    };
  };
  // 子级构造函数
  const child = function () {
    // 用.call()和.apply()将父类构造函数引入子类函数（在子类函数中做了父类函数的自执行（复制））
    parent.call(this, 12); // 主要
    this.name = "shuliqi";
  };

  const person = new child();

  person.getName(); // shuliqi
  console.log(person.age); // 12
  console.log(person instanceof parent); // false
```

优点：

1. 可以继承多个构造函数属性（call多个）。
2. 解决了原型链继承缺点1、2、3。
3. 在子实例中可向父实例传参

缺点：

1. 无法实现构造函数的复用。（每次用每次都要重新调用）
2. 每个新实例都有父类构造函数的副本，臃肿。

组合继承（组合原型链继承和借用构造函数继承）（常用）

```
{
  // 父级构造函数
  const parent = function (age) {
    this.age = age;
    this.getName = function () {
      console.log(this.name);
    };
  };
  // 子级构造函数
  const child = function (age) {
    parent.call(this, age); // 主要
    this.name = "shuliqi";
  };
  child.prototype = new parent();

  const person = new child(12);

  person.getName(); // shuliqi
  console.log(person.age); // 12
  console.log(person instanceof parent); // true
}
```

特点：

1. 可以继承父类原型上的属性，可以传参，可复用。
2. 每个新实例引入的构造函数属性是私有的。

缺点：调用了两次父类构造函数（耗内存）

实例继承（原型式继承）

```
  // 父级构造函数
  const parent = function () {
    this.age = 12;
    this.getName = function () {
      console.log(this.name);
    };
  };

  // 封装一个容器函数，用来输出对象和承载继承的原型
  const content = function (obj) {
    function F() {}
    F.prototype = obj; // 继承传入的参数
    return new F(); // 返回构造函数
  };

  const obj = new parent(); // 拿到父类的实例
```

```
const person = content(obj);
console.log(person.age); // 12
```

重点：用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象。object.create()就是这个原理。

特点：类似于复制一个对象，用函数来包装

缺点：

1. 所有实例都会继承原型上的属性。
2. 无法实现复用。（新实例属性都是后面添加的）

寄生式继承

```
// 父级构造函数
const parent = function () {
  this.age = 12;
  this.getName = function () {
    console.log(this.name);
  };
};

// 封装一个容器函数，  用来输出对象和承载继承的原型
const content = function (obj) {
  function F() {}
  F.prototype = obj; // 继承传入的参数
  return new F(); // 返回构造函数
};
const obj = new parent(); // 拿到父类的实例

const sub = function () {
  const person = content(obj);
  person.haha = "asdhas";
  return person;
};

const person = sub(obj);
console.log(person.age, person.haha); // 12, asdhas
```

重点：就是给原型式继承外面套了个壳子。

23. promise

Promise

24.加载

异步加载js的方法

1. **defer**：defer 属性规定是否对脚本执行进行延迟，直到页面加载为止。只有ie支持

   有的 js 脚本会用 document.write 方法来创建当前的文档内容，其他脚本就不一定是了

   如果您的脚本不会改变文档的内容，可将 defer 属性加入到 标签中，

   ```
   <script type="text/javascript" defer="defer">
   </script>
   ```

2. **async**: html5的属性，属性规定一旦脚本可用，则会异步执行。async 属性仅适用于外部脚本（只有在使用 src 属性时）

   ```
   <script type="text/javascript" src="demo_async.js" async="async"></script>
   ```

注意：

有多种执行外部脚本的方法：

- 如果 async="async"：脚本相对于页面的其余部分异步地执行（当页面继续进行解析时，脚本将被执行）
- 如果不使用 async 且 defer="defer"：脚本将在页面完成解析时执行
- 如果既不使用 async 也不使用 defer：在浏览器继续解析页面之前，立即读取并执行脚本
- 同时存在defer和async，那么defer的优先级比较高；脚本将在页面完成时执行。

3. 动态加载js

图片的懒加载和预加载

- 预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。
- 懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

25. 事件流

标准事件流： 捕获事件（true）—目标事件—冒泡事件（false）

IE事件流：冒泡事件

一个元素绑定了一个捕获事件，绑定了一个冒泡事件，执行几次？ 顺序?

```
<div id="one">
  one
  <div id="two">
    two
    <div id="three">
      three
      <div id="four">
        four
      </div>
    </div>
  </div>
</div>
```

```
document.getElementById('one').addEventListener('click', function() {
  console.log('one')
}, true)
document.getElementById('two').addEventListener('click', function() {
  console.log('two 冒泡')
}, false)
document.getElementById('two').addEventListener('click', function() {
  console.log('two 捕获')
}, true)

document.getElementById('three').addEventListener('click', function() {
  console.log('three')
}, true)
document.getElementById('four').addEventListener('click', function() {
  console.log('four')
}, true)
```

two 元素即绑定了冒泡 也绑定了捕获。

- 标准事件流

如果是目标发生了事件：则目标元素先绑定了什么就先执行什么，其他元素是先捕获后冒泡

如果不是目标事件发生了事件：则先执行捕获或执行冒泡

例如：点击two： 输出：one , two 冒泡, two 捕获

例如：点击four： 输出：one，two 捕获， three， four， two 冒泡

- IE：后绑定的会覆盖钱绑定的事件

事件流的例子

IE与DOM事件的区别

- 事件流不一样：

  DOM: 捕获事件（true）， 目标事件， 冒泡事件（false）

IE: 冒泡事件

- 监听函数的参数不一致

  DOM: addEventListener(type, fn, false/true)

  IE: attachevent(type, fn)

  type 表示事件类型， fn 表示事件触发的函数

- this取值也不一致

  DOM: 严格模式下（undefined）， 非严格模式（当前调用的对象）

  IE: window

- 取消冒泡不一致

  DOM: e.stopPropagatioon()

  IE: window.event.canceBubble = true

- 取消默认行为

  DOM: e.preventDefualt ()

  IE: window.event.returnValue = false

- 获取事件源不一致

  DOM: e.target

  IE: window.event.srcElement

普通事件和事件绑定有什么区别?

普通事件：只支持单一事件， 后面绑定的事件会覆盖当前的事件。

事件绑定 ： 支持绑定多个事件，不会被覆盖， 绑定几个就会有多少个?

mouseover 和 mouseenter的区别

mouseover: 鼠标移入某个元素或者某个元素的子元素都会触发，有冒泡的过程， 对应的取消事件是mouseout

mouseenter :鼠标移除本身元素触发， 不存在冒泡过程， 对应的取消事件是mouseleave

事件委托（事件代理）

原理：利用冒泡原理，把事件加到父级上面， 让父级来执行

好处：提高性能， 新增加的元素也会有该事件

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

```
document.getElementById('ul').addEventListener('click', function(e) {
  const event = e || window.evebt;
  const target = event.target || event.srcElemnt;
  console.log(target.nodeName)
  if (target.nodeName === 'LI') {
    target.style.color = 'red';
  }
}, false)
```

事件代理在捕获阶段的实际应用

可以在父元素层面阻止事件向子元素传播，也可代替子元素执行某些操作。

26.跨域

跨域的原因：浏览器的同源策略（端口，协议，域名）限制了不是同一个源的脚本不能访问另一个源的资源。

为什么有这个限制：如果没有同源策略，浏览器很容易收到XSS, CSFR等攻击.

XSS攻击通常指的是通过利用网页开发时留下的漏洞，通过巧妙的方法注入恶意指令代码到网页，使用户加载并执行攻击者恶意制造的网页程序

CSRF, 跨站请求伪造，攻击者盗用了你的身份，以你的名义发送恶意请求

同源策略限制的行为

- cookie, local storage, indexdb 无法获取
- ajax 请求无法发出
- DOM 和 JS 对象无法被捕获到

解决跨域的方法

- **Jsonp 方式**

  原理:script标签不受同源策略的影响，可以动态创建script标签

```
const script = document.createElement('script');
script.type = "text/javascript"
script.src = 'shuliqi.com:80/?name=shuliqi&age=12&callback=handelCallback';
document.body.appendChild(script)
function handelCallback() {
  // 处理后端返回的
}
```

缺点：只支持get请求

- **document.domain + iframe 跨域**

  限制条件： 同主域， 不同子域

  shuliqi.com/a.html

```
<iframe src="shuliqi.com/b.html"></iframe>
<script>
  document.domain = "do main";
  var nage = "shuliqi";
</script>
```

  shuliqi.com/b.html

```
<script>
  document.domain = "do main";
  // 获取腹肌窗口的变量
  alert(window.parent.name)
</script>
```

- **location.hash + ifrema**

  可以在不同域名跨域了

  原理：hash的改变不会导致页面的刷新

  实现： A域下：a.html， c.html。B 域下： b html，现在 a.html 想和b.html 交换信息， 由于同源策略是不可以修改信息的，就a.html 传给b.html 页面的 hash 值，b.html 页面获取到， 但是a.html与b.html是不同域的，不能进行通信， 没办法获取啊，html的变量，函数等。所以需要与a.html同域的c.html做一个代理

  A域：a.html

```
<iframe id="iframe",  src="B.com/b.html"></iframe>
<script>
  const iframe = document.getelementById('ifame');
```

```
    setTimout(() => {
      // 向b.html传递hash值
      iframe.src = iframe.src + "#user=name"
    }, 1000)

    // 开放给c.html的回调函数
    function handleCallback(res) {
      console.log('data from c.html' + res)
    }

  </script>
```

B域：b.html

```
<iframe id="iframe",  src="A.com/C.html"></iframe>
<script>
  const iframe = document.getelementById('ifame');
  // 监听a.html传过来的hash值
  window.onhashchange = function() {
     // 向C.html传递hash值
     iframe.src = iframe.src + location.hash;
  }
</script>
```

A域： c.html

```
<script>
  window.onhashchange = function() {
    window.parent.parent.handleCallback('我调用a.html的回道函数' +
location.hash.replace('#nahe="hahahah"'))
  }
</script>
```

缺点： url 暴露出来了， 而且hash的长度是有限制的。

- **window.name +iframe**

  原理：window.name 在不同的页面，甚至不同域加载之后依然存在，并且可以支持非常长的name值

  A域a.html proxy.html B域的b.html 。

  实现：b.html 设置window.name。a.html 创建iframe, src指向b.html。但是a.html 和b.html 不同域名。无法通过ifame.contentWindow.name 获取 那么值。 所以就借助proxy.html。在rsc.指向b.html之后，迅速改src指向proxy.html。这样就是同域的了。就可以获取name值了。每次触发onload时间后，重置src，相当于重新载入页面，又触发onload事件，于是就不停地刷新了（但是需要的数据还是能输出的

  a.html

```
<script>
  let state = 0;
  const ifame = window.createElement('ifame');
  ifame.src = 'b.html'
   // onload事件会触发2次，第1次加载跨域页，并留存数据于window.name
  ifame.onload = function() {
      if (state === 0) {
        // 第1次onload(跨域页)成功后，切换到同域代理页面
          state = 1;
          iframe.contentWindow.location = 'proxy.html';
      } else if(state ===1) {
        // 第2次onload(同域proxy页)成功后，读取同域window.name中数据
         var data = JSON.parse(iframe.contentWindow.name);
          console.log(data);
          iframe.contentWindow.document.write('');
          iframe.contentWindow.close();
          document.body.removeChild(iframe);
      }
  }
  document.body.appendChild(iframe);
</script>
```

proxy.html 是一个空页面

b.html

```
<script>
  window.name = "shuliqi"
</script>
```

- **post.message()**

只支持到IE8及以上的IE浏览器，其他现代浏览器当然没有问题。

不受同源策略的限制

1. 接收数据方：

```
/*
 * postMessage.js中的数据放在的消息队列里，监听message获取到消息
 *
 */
window.addEventListener("message",function(event){
    if(event.origin!="http://127.0.0.1:8848"){
        alert("跨域访问，不接受");
    }else{//同源的地址
        console.log("event.origin:",event.origin);
        console.log(event.data);
        document.querySelector("#result").innerHTML=event.data;
    }
```

```
})
```

2. 发送数据方：

```
var userData =[{
    "id":0,
    "name":"张三",
}];
window.postMessage(JSON.stringify(userData));
```

- CORS： 跨域资源共享

  前端和后端一起配合， 具体说来就是在header中加入origin请求头字段，同样，在响应头中，返回服务器设置的相关CORS头部字段，Access-Control-Allow-Origin字段为允许跨域请求的源。请求时浏览器在请求头的Origin中说明请求的源，服务器收到后发现允许该源跨域请求，则会成功返回

请求方式使用下列方法之一：GET，HEAD，POST

Content-Type 的值仅限于下列三者之一：text/plain，multipart/form-data

27.ajax

实现一个ajax

```
const xhr = new XMLHttpRequest();
// 必须在调用 open()之前指定 onreadystatechange 事件处理程序才能确保跨浏览器兼容性
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    if (xhr.status === 200 || xhr.status <= 300 || status === 304) {
      console.log('成功拿到数据：', responseText)
    } else {
      console.log('出错了：'status)
    }
  }
}
//  第三个参数表示异步请求
xhr.open('get','https://i2', true)
xhr.send(null)
```

ajax的状态

0 未初始化。表示还没有调用open()函数

1.初始化。已经调用open()函数，但是还没有调用send()函数

2.发送。已经调用send()函数，但是还没收到响应

3.接收。接收到部分响应

4.完成。接收到全部反应

将原生的ajax封装成promise

```
const ajax = (method, url, async, data) => {
  return new Promise((resolve, reject) => {
    const xhr = XMLHttpRequest();
    xhr.onreadystatechange = () => {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          resolve(JSOM.parse(xhr.responseText);
        } else if(xhr.status > 400) {
            reject("发生错误");
        }
      }
    }
    xhr.open(method, url, async);
    xhr.send(data || null);
  })
}
```

28. 垃圾回收

什么是内存泄漏

内存泄漏就是指：不再使用的内存， 没有得到及时的释放。没有得到释放的结果就是导致系统变慢。

什么是垃圾回收

垃圾回收机制就是间歇的不定期的对这些不在使用的变量， 释放它们占用的内存。

变量的生命周期

全局变量在浏览器关闭之后会被清除， 局部变量会在函数执行完毕之后被释放。

垃圾回收的方式

标记清除

垃圾收集齐会在当所有的变量进入环境的时候都标记一下，例如"进入环境"， **去除环境中的变量以及被环境中的变量引用的变量**。再此之后再被标记上的就是待释放内存的变量。

引用计数

29 eval 是什么

eval()方法就是解析并且执行js字符串

- 性能很差： js引擎会在编译阶段对很多项的优化， 其中有一些项目就是对代码进行静态的词法分析。预先确定变量和函数的位置，在执行的时候才能找到标识符，
- 无法在词法分析的阶段明确知道eval()会接收到什么代码。所以无法做词法分析。
- 欺骗作用域：在严格的模式下。 Eval（）在执行代码的时候室友自己的词法作用域的。意味着其中的变量无法修改所在的作用域。

30 如何监听对象的属性

ES5 :Object.defineProperty()

new Proxy()

31 如何实现私有变量

```
function Pclass() {
  //注意：这里不用this
  const name = "私有变量";
  this.getName = () => {
    console.log("私有获取name的方法：", name);
  };
  this.age = "变量age，外部可以访问";
}
const pclass = new Pclass();
console.log(pclass.name); // undefined
console.log(pclass.getName()); // 私有获取name的方法：  私有变量
console.log(pclass.age); // 变量age，外部可以访问
```

32 new 操作符都干了什么

1. 创建一个新的函数
2. 将构造函数的作用域指向新的函数（将this值指向新的函数）fn**proto** = Fn.prototype
3. 自执行构造函数的代码（是为了把属性和方法给新的函数）call(). apply方式
4. 返回新的函数

33 数组去重

includes方式

```
function unique(arr) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    if (!result.includes(arr[i])) {
      // 如果结果集里面没有这个选项，则加上ß
      result.push(arr[i]);
    }
  }
  return result;
}
console.log(unique([1, "1", 2, 3, 3, 5, 5, 6, 6])); // [ 1, '1', 2, 3, 5, 6 ]
```

indexOf

indexOf方式和includes()是一样的

ES6 Set()

```
function unique(arr) {
  return Array.from(new Set(arr));
}
console.log(unique([1, "1", 2, 3, 3, 5, 5, 6, 6])); // [ 1, '1', 2, 3, 5, 6 ]
```

或者

```
function unique(arr) {
  return [...new Set(arr)];
}
console.log(unique([1, "1", 2, 3, 3, 5, 5, 6, 6])); // [ 1, '1', 2, 3, 5, 6 ]
```

ES6 map()

```
function unique(arr) {
  const map = new Map();
  const resulte = arr.filter((a) => {
    if (!map.has(a)) {
      return map.set(a, 1);
    }
  });
  return resulte;
}
console.log(unique([1, "1", 2, 3, 3, 5, 5, 6, 6])); // [ 1, '1', 2, 3, 5, 6 ]
```

34 展开数组

es6： flat()

```
function flatArr(arr = []) {
  return arr.flat(Infinity);
}
console.log(flatArr([1, [2, 3, [4]], 5, 6, [7, 8], [[9, [10, 11], 12], 13],
14]))
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

es6： rest操作符号

```
function flatArr(arr) {
  while(arr.some((item) => Array.isArray(item))) {
    arr = [].concat(...arr)
  }
  return arr;
}
console.log(flatArr([1, [2, 3, [4]], 5, 6, [7, 8], [[9, [10, 11], 12], 13],
14]))
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

递归方式

```
function flatArr(arr) {
  let result = [];
  arr.forEach((a) => {
    if(Array.isArray(a)) {
      // 如果还是数组，  则就递归
      result = result.concat(flatArr(a))
    } else {
      result.push(a)
    }
  })
  return result;
}
console.log(flatArr([1, [2, 3, [4]], 5, 6, [7, 8], [[9, [10, 11], 12], 13],
14]))
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

toString()

```
function flatArr(arr) {

  return arr.toString().split(',').map((a) => Number(a));
}
console.log(flatArr([1, [2, 3, [4]], 5, 6, [7, 8], [[9, [10, 11], 12], 13],
14]))
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

reduce

```
function flatArr(arr) {
    return arr.reduce((pre, next) => {
      return pre.concat(Array.isArray(next) ? flatArr(next) : next);
    }, []);
  }
  console.log(
    flatArr([1, [2, 3, [4]], 5, 6, [7, 8], [[9, [10, 11], 12], 13], 14])
  );
  // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

35 实现拖拽

js实现

直接上例子 拖拽实现

其中需要注意的点：

- 要拖拽的元素必须的是绝对定位的

```
#box{
    position: absolute; // 必须是绝对定位的
    width: 200px;
    height: 200px;
    background: red;
}
```

- 为了防止目标拖拽元素拖动太快，目标移动不够快，可能或失去焦点，所以mosemove绑定在document上

Htm5 : draggable

Html5 的draggable 属性设置为true 可实现拖拽

直接上例子吧

draggable

其中只需要：

```
<div class="box" draggable="true"></div>
```

拖拽元素的时候，被拖拽元素会触发以下事件

- **dragstart**
- **drag**
- **dragend**

36.实现一个只会执行一次的函数（once）

```
function once(fun) {
  let done = false;
  return () => {
    if (!done) {
      fun.apply(null, arguments);
      done = true;
    }
  }
}

function test() {
  console.log("我是test");
}
const myName = once(test);
myName(); // 我是test
myName(); // ps:不会输出什么
myName(); // ps:不会输出什么
myName(); // ps:不会输出什么
myName(); // ps:不会输出什么
```

37 sleep

promise 实现

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, time);
  });
}

sleep(1000).then(() => {
  console.log("过了1000ms 我才执行的");
});
```

async, await（Generator函数的改进版本）

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, time);
  });
}

async function test() {
  await sleep(1000);
  console.log("过了1000ms 我才执行的");
}
test();
```

Genrator 函数

```
function* sleep(time) {
  yield new Promise((resolve, reject) => {
    setTimeout(resolve, time);
  });
}

sleep(1000)
  .next()
  .value.then(() => {
    console.log("过了1000ms 我才执行的");
  });
// 其中使用.value 才是返回一个genertor 接口
```

38 .promise

Promise

40. 事件循环

自己写的博客

js的任务大致分为同步宏任务，异步微任务， 异步宏任务，

js 的主线程会首先执行同步的宏任务，如果遇到异步（微任务和宏任务）的任务，就挂起来， 有结果的时候就放在异步微任务队列和异步宏任务队列，同步宏任务执行完， 就去执行异步微任务的里面的任务，最后执行异步宏任务里面的队列。

时间循环也就是javascriptde的运行机制，运行机制也就是为了解决js 实现异步操作。

宏任务： 全部的javscript代码(同步)， setTimeout(异步)， setInterval(异步)。I/O,浏览器的render等

微任务：promise，process，nextTICK

事件循环的机制：同步宏任务--> 微任务promise---> 微任务 process，nexttick ----> 异步宏任务

```
console.log("1");
setTimeout(function() {
   console.log('2')
});
console.log('3');
```

输出的结果， 1， 3， 2

同步宏任务 1， 3 ----> 微任务（无）----> 异步宏任务（2）

```
  process.nextTick(function () {
    console.log("1");
  });

  new Promise(function (resolve) {
    console.log("2");
    resolve();
  }).then(function () {
    console.log("3");
    setTimeout(function () {
      console.log("4");
    });
  });

  new Promise(function (resolve) {
    setTimeout(function () {
      console.log("6");
    });
    resolve();
  }).then(function () {
    setTimeout(function () {
      console.log("7");
      new Promise(function (resolve) {
        setTimeout(function () {
          console.log("8");
        });
        resolve();
      }).then(function () {
        setTimeout(function () {
          console.log("9");
        });
      });
    });
```

```
  });
  console.log("10");



  // 2 10 1 3 6 4 7 8 9
```

```
{
  function f() {
    setTimeout(() => {
      console.log(5);
      Promise.resolve().then(() => {
        console.log(6);
      });
    });

    new Promise((resolve, reject) => {
      console.log(1);
      resolve(1);
    }).then(() => {
      console.log(2);
      Promise.resolve().then(() => {
        console.log(3);
      });
      setTimeout(() => {
        console.log(4);
      });
      Promise.resolve().then(() => {
        console.log(7);
      });
    });
  }
  f();
  // 1 2 3 7  5 6 4
}
```

41 es6 的一些小拓展

- 函数的默认值

  e s5 是是没有默认值的，es6 参数可以写成默认参数

  ```
  function add (x = 1, y = 2) {
    return x + y;
  }
  ```

- rest参数： ...变量名， 用来获取函数多余的参数。rest是一个数组， 将剩余的变量存入数组里面

```
function add(a, ...rest) {
  console.log(a); // 1
  console.log(Array.isArray(rest)); // true
  console.log(rest); // 2,3,4,5,6,7,8
}
add(1, 2, 3, 4, 5, 6, 7, 8);
```

- 扩展运算符（...）相当于是...rest 的逆运算。将数据转换成以逗号分割的参数序列

```
console.log(...[1, 2, 3, 4, 5, 6, 7]); // 1 2 3 4 5 6 7
```

- 箭头函数：

```
const add = () => { return 12};
```

注意点：

  - 箭头函数的this 在创建的时候就确定了， 就是定义时所在的对象，而不是在在调用时所被调用的对象
  - 箭头函数不能使用new 命令
  - 将有函数不能使用argumengts, 可以使用...rest来替代
  - 不可以使用yield 命令， 所以肩头函数不能写成generator函数

- 变量的结构赋值

```
const [a, b, c] = [1, 2, 3];
console.log(a, b, c); // 1 2 3
```

- Set 数据结构

Set类似数组， 但是没有重复的值

```
// 不会有重复的值
console.log(new Set([1, 1, 2, 2, 3, 3, 3, 4, 4, 5])); // 1 2 3 4 5

// newSet的方法
const setArr = new Set([1, 1, 2, 2, 3, 3, 4]);
console.log(setArr.add(9));
console.log(setArr.delete(1));
console.log(setArr.has(2));
setArr.clear();
console.log(setArr);
```

  - add: 添加一个元素， 返回整个set
  - delete： 删除一个元素， 返回布尔值，true 表示删除成功
  - has： 判断是否有有某个元素， 返回布尔值， true表示有

    - clear： 清空所有的元素， 没有返回值

- Map 数据结构

```
const obj = { name: "shuliqi" };
const mapObj = new Map();
mapObj.set(obj, "hahah");
mapObj.set({ name: "haha" }, 11111);
mapObj.get(obj);
mapObj.delete(obj);
mapObj.has({ name: "haha" });
mapObj.clear();

console.log(mapObj);‘
```

- 增加了会计作用域：let， const

  let定义的变量只有在会计作用域内有效，

    - let 不存在变量提升，
    - 不可以重复声明
    - 存在暂时性死区

  const 用来定一个常量， 一旦定义了就不可以改变了。

42. post 和get 的区别

- post 的数据是放在body里面的比较安全。但是用户刷新是没用的。get的数据是明文在url里面的，比较不安全， 但是用户可以刷新继续使用
- get能传送的数据量只有2 - 8k， post 能传送的比较多点

44 document.onload 和document.ready的区别

- doument.onload:

  指dom元素和其它元元素都加载完成（包括图片和多媒体）

- Document.ready

  指dom元素家在完成

45 js如何创建一个对象

- 使用内置对象

```
const obj = new Object();
const string = new String("sds");
```

- 使构造函数

```
const obj = {
  this.name = "shuliqi",
  this.age = 12
};
// 或者
```

- 使用对象字面量

```
const obj = {
  name: "shuliq",
  age = 12;
}
```

46. [1,2,3].map(parseInt)

返回的结果为： 1， NaN, NaN。

```
[1,2,3].map((value, index, arr) => {})
```

map的回调函数传入的是三个参数，

- 第一个参数是当前的循环的值
- 第二值是当前的值的下标
- 第三个值是数组本身

```
parseInt(string, radix)
```

- 第一个参数是要转换的字符，
- 第二个参数是要转成的进制

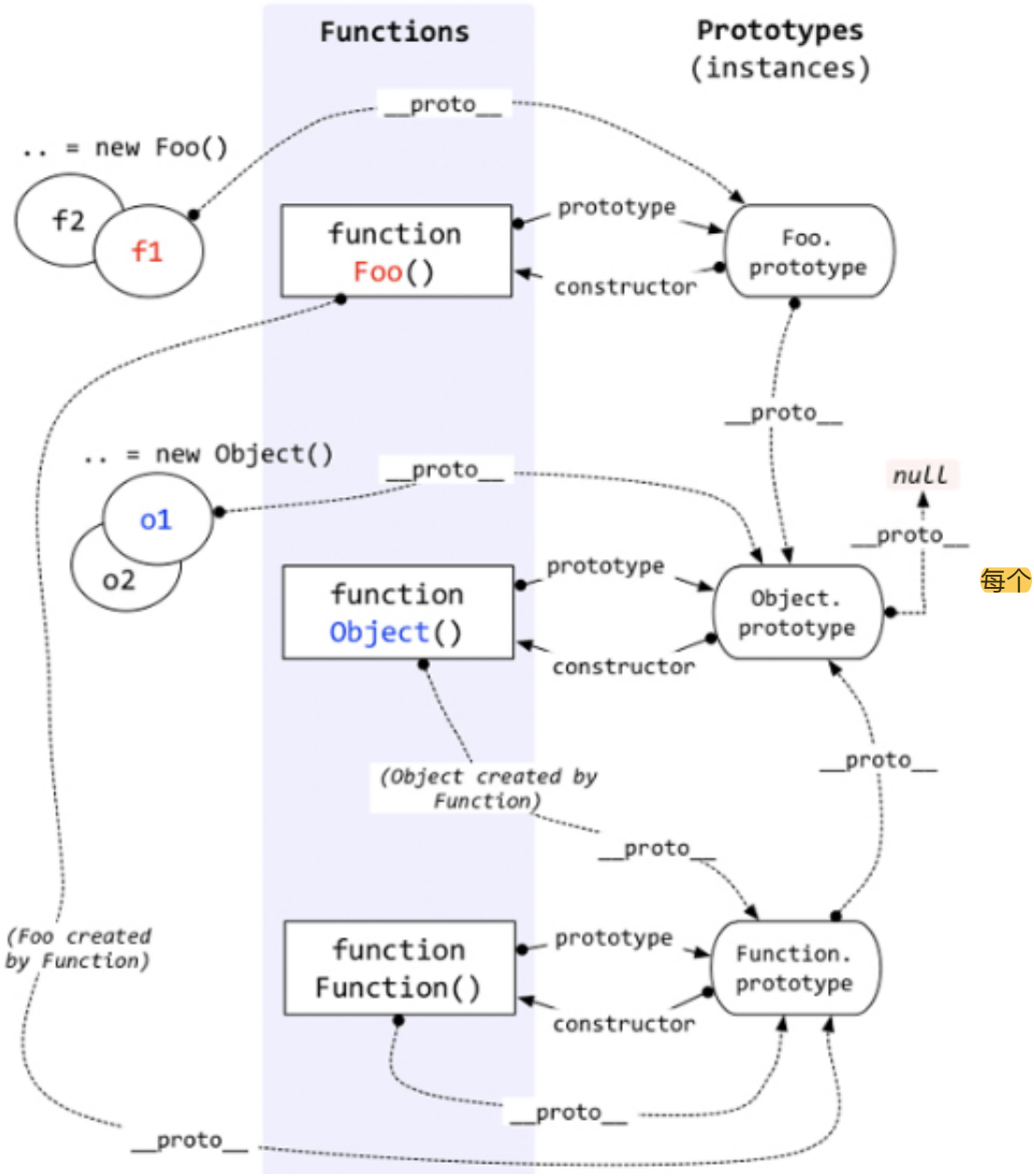
  这个值的范围是 2-36。如果不在这个范围则输出NaN， 默认是0 表示10 进制

所以这个解析的过程就变成了

```
parseInt(1, 0) // o 表示默认10进制， 输出1
parseInt(2, 1) // 1 不在2-36 范围， 输出NaN
parseInt(3, 2) //  3 不在2进制的范围里面， 输出NaN
```

二进制 就是以1 和0 开头的， 因为逢2 就进一位了。ßß

47 原型链的理解



每个

每个函数都有一个内置的属性（prototype），它是一个指针， 指向这个函数的原型

每个对象都有一个内置的属性（**proto**）。 指向的是构造它的函数的原型。

在查找对象的属性或者方法的时候， 会现在自身的属性有咩有属性和方法，如果没有则沿着**proto**继续其构造函数的的prototype。而其构造函数的prototype 也是一个对象，也有隐原型。就这样一层层的往上找。 直到找到最终的构造函数额prototype。Funtion.prototype.而Funtion.prototype 也是一个对象， 它是由Object.prototype构造的。所以的它的隐原型指向了Object.prototype。 而Object.prototype的隐原型指向了null。这样就到达了原型链的终点

48. 深拷贝实现

```
function deepCopy(obj) {
  if (obj === null || typeof obj !== "object") {
    return obj;
  }

  let result = Array.isArray(obj) ? [] : {};
  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      result[key] = deepCopy(obj[key]);
    }
  }
  return result;
}
```

50. 前端性能优化

- 尽量减少请求数量。 合理利用缓存
- css Spities。将图片合起来， 这样就会减少很多请求
- 懒加载图片： 第一次只加载第一屏幕的图片，到达屏幕底部的时候再加载新的。
- css 放在html 的头部， js 加载放在尾部
- 尽量少用with， eval（）等
- 少用css表达式等
- 为了减少回流和重绘：
  - 尽量少用table 部署，
  - 改变样式尽量使用class 来一次性改变
  - 将动画运用到position 为absolute 和fixed 上面
  - 避免使用多层次的内敛样式，
  - 少用css表达式（calc（）等）
- cdn 做负载均衡

51. vue 和react 的区别

52 Proxy 和defineProperty的区别

Proxy 是对整个对象进行拦截，

defineProperty 是只能对某个属性进行拦截

Proxy有更多的拦截的方式（get， set， has，apply， defineProperty等13 种）

defineProperty 只有get， set，configurable, enumerable， writable，value

55 CDN 是干什么的，有什么意义呢?

56 前端常见的安全问题有哪些?

前端安全问题

57 对缓存的理解

前端缓存

按照位置来分的话：

- memory catch： 存在内存中， 因此不是一个永久的存储，浏览器关闭就会不在了， 一把几乎所有的请求都会memory catch；
- disk catch: 存在硬盘上， 是一个永久的， 但是浏览器会有自己的算法去调没有用的缓存。这个是需设置http头的各种字段来实现的， 我们也叫http 缓存。

按照缓存的不同来分：

- 强缓存： 强缓存的主要的目的就是减少请求。 如果命中了强缓存， 就不会发出请求了。一般有两个字段设置： Expires， Catch-Control。

```
const express = require('express')
const fs = require('fs');
const app = express();
app.get('/1/css', (res, req) => {
  fs.readFile(patch, (err, data) = >{
    req.setHeader('Expires', 'Thu Dec 05 2019 23:51:08 GMT+0800 (CST');
    req.setHeader('Catch-Control', 'public max-age=600');
    req.sed(data.toString())
  })
})
```

后端设置了强缓存， 那么当前端要发出请求时呢， 先去看当前的时间有没有超过缓存时间，如果没有超过呢， 就不去请求， 而而是直接使用缓存。

- 对比缓存： 对比缓存的只要目的主要是减少响应体，来加快传输的过程。主要用到的两对字段。Last-Modified/ If-Modified-Since。 Etag/If-None-Match

后端的设置

```
// 对比缓存 [if-modified-since, Last-Modified, ]
app.use('/1.js', (req, res) => {
  const jsPath = path.join(__dirname, './public/javascripts/1.js');
```

```
    // 获取文件1.js的信息
    fs.stat(jsPath, (err, stat) => {
      // 获取文件内容被修改的时间 modify time
      let lastModified = stat.mtime.toUTCString();
     // 判断 if-modified-since 的时间与资源的最后修改时间是否一致
      if (req.headers['if-modified-since'] === lastModified) {
        // 设置响应状态码
        res.writeHead(304, 'not modified');
       // 响应体为空，减少传输时间
        res.end();
      } else {
        // 读取文件
        fs.readFile(jsPath, (err, content) => {
          // 设置Last-Modified
          res.setHeader('Last-Modified', lastModified);
          // 设置响应状态码
          res.writeHead(200, 'ok');
          // 响应体为空，减少传输时间
          res.end(content);
        })
      }
    })
});
```

前端第一次请求的时候， 根据相应头的Last-Modified，Etag 我们知道这个内容缓存。当我们再发出请求的时候我们在请求头在分别把If-Modified-Since. If-None-Match 字段再返回给后端。值就是上一次响应头两个字段返回给我们的值。 如果后端判断文件内容没有改变， 则直接返回状态码为304。没有响应体给前端。

58 TypeScript

typescript

59 正则

字符集

-
-
- \w： 匹配 [a-zA-G0-9]的任意字符
- \W：匹配[^a-zA-G0-9]的任意字符
- \d: 匹配一个数字， [0-9]
- \D: 匹配一个非数字 [^0-9]
- \s: 匹配空白符号

- \S: 匹配非空白字符

重复

- {n, m}: 至少匹配n次， 但是不超过m次
- {n, }: 至少匹配n次
- {n}: 匹配n次
- ? : 匹配 0次或则1 次
- *: 匹配0次或则 多次
- +: 匹配1 次或则多次

组合

() ： 提取匹配字符串的，表达式中有几个()就有几个相应的匹配字符串

```
const str = "17885257632";
const reg = /^(\d{3})\d*(\d{4})$/;
console.log(str.match(reg)); // [ '17885257632', '178', '7632', index: 0,
input: '17885257632' ]
```

选择

|

```
  const str = "shuliqi";
  // 匹配s开头 或者l结尾的
  const reg = /^([s])|([l])$/g;
  console.log(str.match(reg)); // [ 's' ]
```

指定位置匹配

- ^: 匹配开头
- $: 匹配结尾
- \b: 匹配一个单词的边界
- \B: 匹配一个非单词的边界

60 电话号码中间四位变星号

```
function phone(phone) {
  const arr = ("" + phone).split("");
  arr.splice(3, 4, "****");
  return arr.join("");
}
console.log(phone(17885251632));
```

使用正则表达式

```
function phone(num) {
  return ("" + num).replace(/^(\d{3})\d*(\d{4})$/, "$1****$2");
}
console.log(phone(13991367972));
```

```
function phone(num) {
  const str = "" + num;
  return str.substr(0, 3) + "****" + str.substr(7);
}
console.log(phone(17885257632)); // 178****7632
```

```
function phone(num) {
  const str = "" + num;
  return str.slice(0, 3) + "****" + str.substr(7);
}
console.log(phone(17885257632)); // 178****7632
```

```
function phone(num) {
  const str = "" + num;
  return str.substring(0, 3) + "****" + str.substr(7);
}
console.log(phone(17885257632)); // 178****7632
```

61 字符串的方法

charAt(index)

返回一个字符串指定下标的字符， 如果index 值不在0 和 index 之间，则返回一个空的字符串。字符串的第一个下标是0

```
var str = "123456";
console.log(str.charAt(0)); // 1
console.log(str.charAt(-1)); // “”不在0 - 6 中间， 返回空
console.log(str.charAt(12)); // “”不在0 - 6 中间， 返回空
```

charCodeAt(index)

返回一个字符串指定下标的值的unicode 值。 如果index 值的范围不在0 和 length之间。 则返回NAN。

```
const str = "12345677";
console.log(str.charCodeAt(0)); // 49
console.log(str.charCodeAt(-1)); // NAN  不在0 - 6 中间， 返回空
console.log(str.charCodeAt(12)); // NAN  不在0 - 6 中间， 返回空
```

concat(str1, str2,...)

用于拼接字符串concat 方法会把所有的参数转成字符串，然后按照顺序来拼接。注意， 拼接不会改变原字符串。

使用 + 会更好

```
const str1 = "shuliqi";
const str2 = "23";
console.log(str1.concat(str2)); // shuliqi23
console.log(str1); // shuliqi // 原字符串不被改变
console.log(str2); // 23 //  原字符串不被改变
```

fixed()

把字符串显示为打字机字体

```
const str = "shuliqi";
console.log(str.fixed()); // <tt>shuliqi</tt>
```

indexOf(searchStr, index)

用于检索某个指定的字符串在字符串中首次出现的位置. 如果找不到就返回-1

- searchStr：要检索的字符串
- index： 开始检索的位置。 可选， 如果没有是从0 开始检索的，index 的取值在0 和length -1

```
const str = "shuliqi";
// 查找u, 从下标为3 开始查找
console.log(str.indexOf("u", 3)); // -1
console.log(str.indexOf("u")); // 2
```

lastIndexof()

个指定的字符串值最后出现的位置，在一个字符串中的指定位置从后向前搜索

```
const str = "shuliqi";
// 查找u, 从下标为3， 从后面 开始查找
console.log(str.lastIndexOf("u", 3)); // -1
console.log(str.lastIndexOf("u")); // 2
```

match()

检索指定的字符串或者匹配一个或者多个正则匹配的字符串， 和indexOf ， lastIndexOf 差不多。只是返回的是存放指定结果的数组， 而不是下标。

```
var str = "Hello world!";
console.log(str.match("world")); // ['world', index: 6, input: 'Hello world!']
const str1 = "shuliwqi 1ask 3";
console.log(str1.match(/\d+/g)); // [ '1', '3' ]
console.log(str1.match("asdhajg")); // null
```

replace(regexp/substr, replacement)

- regexp/substr:要替换的字符串或者是符合匹配正则的字符串
- replacement：要替换成新的新的字符串

replacement 可以是字符串，也可以是函数。如果它是字符串，那么每个匹配都将由字符串替换。但是 replacement 中的 $ 字符具有特定的含义。如下表所示，它说明从模式匹配得到的字符串将用于替换。

字符	替换文本
$1、$2、...、$99	与 regexp 中的第 1 到第 99 个子表达式相匹配的文本。
$&	与 regexp 相匹配的子串。
$`	位于匹配子串左侧的文本。
$'	位于匹配子串右侧的文本。
$$	直接量符号。

```
const str = "shul 1 iqi1 ";
console.log(str.replace(/\d/g, "haha")); // shul haha iqihaha
console.log(str.replace("shu", "na")); // nal 1 iqi1
```

search(reg)

从头检索与正则表达式匹配的字符串， 找到返回第一个匹配的字符串的下标。 枷锁不-1

```
const str = "s 11 hul 1111 iqi12123";
// 匹配第一个数字
console.log(str.search(/\d/i)); // 2
// 匹配第一个【a-z0=9, A-Z】
console.log(str.search(/\w/i)); // 0
// 匹配 1 或则 2  字符串
console.log(str.search(/[12]/)); // 2
console.log(str.search(/\d{5,}/)); // 17
```

slice(start, end)

返回指定返回的字符串, 不不包含end。 接受负数

```
const str = "123shuliq";
console.log(str.slice(2, 4)); // 3s
console.log(str); // 123shuliq  说明原字符串不会改变
console.log(str.slice(-2, -1)); // i  接受负数
```

substr(start, n)

从起始位置开始，提取n个字符 ， 如果没有指定n, 则从起始位置到最后。 如果起始位置是负数的话，那么是从字符串后头开始计算的。

```
const str = "123shuliq";
console.log(str.substr(2, 4)); // 3shu
console.log(str.substr(2)); // 3shuliq
console.log(str.substr(-2)); // iq  可接受负数
```

substring

和slice 和substr 一样， 只是接受负数

```
const str = "123shuliq";
console.log(str.substring(2, 4)); // 3s
console.log(str); // 123shuliq  说明原字符串不会改变
console.log(str.substring(-2, -1)); // 没有返回值的
```

toLowerCase()

将字符串字母转成小写

```
const str = '123SHULIQ'
console.log(str); // 123shuliq  说明原字符串不会改变
console.log(str.toLowerCase()); // 123shuliq
```

toUpperCase()

将字符串字母转成大写

```
const str = "123shuliq";
console.log(str.toUpperCase()); // 123SHULIQ
```