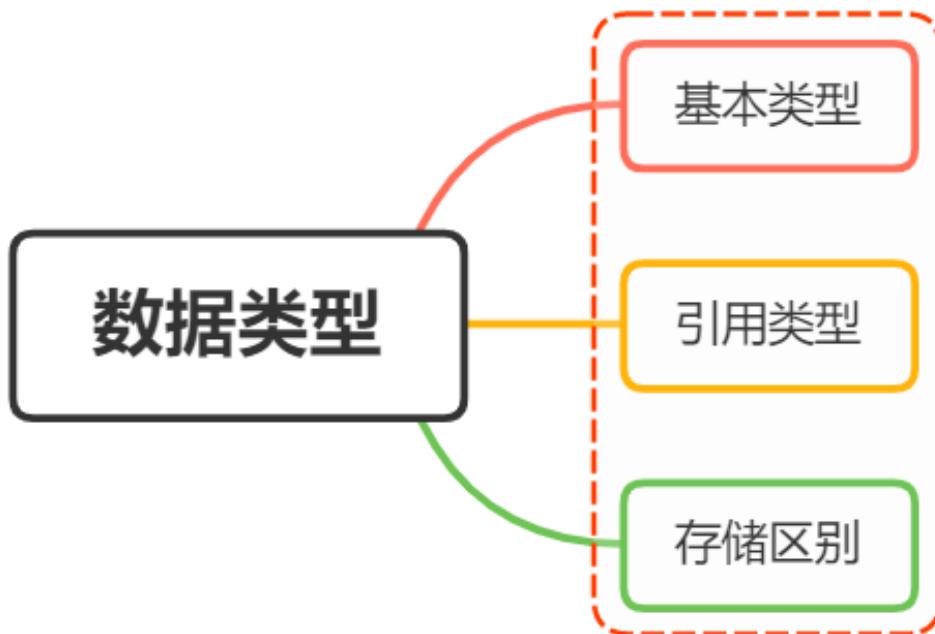


# JavaScript面试真题（35题）

## 1. 说说JavaScript中的数据类型？存储上的差别？



### 1.1. 前言

在 `JavaScript` 中，我们可以分成两种类型：

- 基本类型
- 复杂类型

两种类型的区别是：存储位置不同

### 1.2. 基本类型

基本类型主要为以下6种：

- Number
- String
- Boolean
- Undefined
- null

- symbol

### 1.2.1. Number

数值最常见的整数类型格式则为十进制，还可以设置八进制（零开头）、十六进制（0x开头）

```
1 let intNum = 55 // 10进制的55
2 let num1 = 070 // 8进制的56
3 let hexNum1 = 0xA //16进制的10
```

浮点类型则在数值汇总必须包含小数点，还可通过科学计数法表示

```
1 let floatNum1 = 1.1;
2 let floatNum2 = 0.1;
3 let floatNum3 = .1; // 有效，但不推荐
4 let floatNum = 3.125e7; // 等于 31250000
```

在数值类型中，存在一个特殊数值 `NaN`，意为“不是数值”，用于表示本来要返回数值的操作失败了（而不是抛出错误）

```
1 console.log(0/0); // NaN
2 console.log(-0/+0); // NaN
```

### 1.2.2. Undefined

`Undefined` 类型只有一个值，就是特殊值 `undefined`。当使用 `var` 或 `let` 声明了变量但没有初始化时，就相当于给变量赋予了 `undefined` 值

```
1 let message;
2 console.log(message == undefined); // true
```

包含 `undefined` 值的变量跟未定义变量是有区别的

JavaScript | 复制代码

```
1 let message; // 这个变量被声明了，只是值为 undefined
2
3 console.log(message); // "undefined"
4 console.log(age); // 没有声明过这个变量，报错
```

### 1.2.3. String

字符串可以使用双引号 ("") 、单引号 ('') 或反引号 (`) 标示

JavaScript | 复制代码

```
1 let firstName = "John";
2 let lastName = 'Jacob';
3 let lastName = `Jingleheimerschmidt`
```

字符串是不可变的，意思是一旦创建，它们的值就不能变了

JavaScript | 复制代码

```
1 let lang = "Java";
2 lang = lang + "Script"; // 先销毁再创建
```

### 1.2.4. Null

`Null` 类型同样只有一个值，即特殊值 `null`

逻辑上讲，`null` 值表示一个空对象指针，这也是给 `typeof` 传一个 `null` 会返回 `"object"` 的原因

JavaScript | 复制代码

```
1 let car = null;
2 console.log(typeof car); // "object"
```

`undefined` 值是由 `null` 值派生而来

JavaScript | 复制代码

```
1 console.log(null == undefined); // true
```

只要变量要保存对象，而当时又没有那个对象可保存，就可用 `null` 来填充该变量

## 1.2.5. Boolean

`Boolean` (布尔值) 类型有两个字面值: `true` 和 `false`

通过 `Boolean` 可以将其他类型的数据转化成布尔值

规则如下:

			JavaScript	复制代码
1	数据类型	转换为 <code>true</code> 的值	转换为 <code>false</code> 的值	
2	<code>String</code>	非空字符串	""	
3	<code>Number</code>	非零数值 (包括无穷值)	<code>0</code> 、 <code>NaN</code>	
4	<code>Object</code>	任意对象	<code>null</code>	
5	<code>Undefined</code>	<code>N/A</code> (不存在)	<code>undefined</code>	

## 1.2.6. Symbol

`Symbol` (符号) 是原始值, 且符号实例是唯一、不可变的。符号的用途是确保对象属性使用唯一标识符, 不会发生属性冲突的危险

			JavaScript	复制代码
1	<code>let genericSymbol = Symbol();</code>			
2	<code>let otherGenericSymbol = Symbol();</code>			
3	<code>console.log(genericSymbol == otherGenericSymbol); // false</code>			
4				
5	<code>let fooSymbol = Symbol('foo');</code>			
6	<code>let otherFooSymbol = Symbol('foo');</code>			
7	<code>console.log(fooSymbol == otherFooSymbol); // false</code>			

## 1.3. 引用类型

复杂类型统称为 `Object`, 我们这里主要讲述下面三种:

- `Object`
- `Array`
- `Function`

### 1.3.1. Object

创建 `object` 常用方式为对象字面量表示法, 属性名可以是字符串或数值

JavaScript | 复制代码

```
1 let person = {  
2     name: "Nicholas",  
3     "age": 29,  
4     5: true  
5 };
```

### 1.3.2. Array

JavaScript 数组是一组有序的数据，但跟其他语言不同的是，数组中每个槽位可以存储任意类型的数据。并且，数组也是动态大小的，会随着数据添加而自动增长

JavaScript | 复制代码

```
1 let colors = ["red", 2, {age: 20}]  
2 colors.push(2)
```

### 1.3.3. Function

函数实际上是对象，每个函数都是 `Function` 类型的实例，而 `Function` 也有属性和方法，跟其他引用类型一样

函数存在三种常见的表达方式：

- 函数声明

JavaScript | 复制代码

```
1 // 函数声明  
2 function sum (num1, num2) {  
3     return num1 + num2;  
4 }
```

- 函数表达式

JavaScript | 复制代码

```
1 let sum = function(num1, num2) {  
2     return num1 + num2;  
3 };
```

- 箭头函数

## 函数声明和函数表达式两种方式

```
1 let sum = (num1, num2) => {  
2     return num1 + num2;  
3 };
```

JavaScript | 复制代码

### 1.3.4. 其他引用类型

除了上述说的三种之外，还包括 `Date`、`RegExp`、`Map`、`Set` 等.....

## 1.4. 存储区别

基本数据类型和引用数据类型存储在内存中的位置不同：

- 基本数据类型存储在栈中
- 引用类型的对象存储于堆中

当我们把变量赋值给一个变量时，解析器首先要确认的就是这个值是基本类型值还是引用类型值

下面来举个例子

### 1.4.1. 基本类型

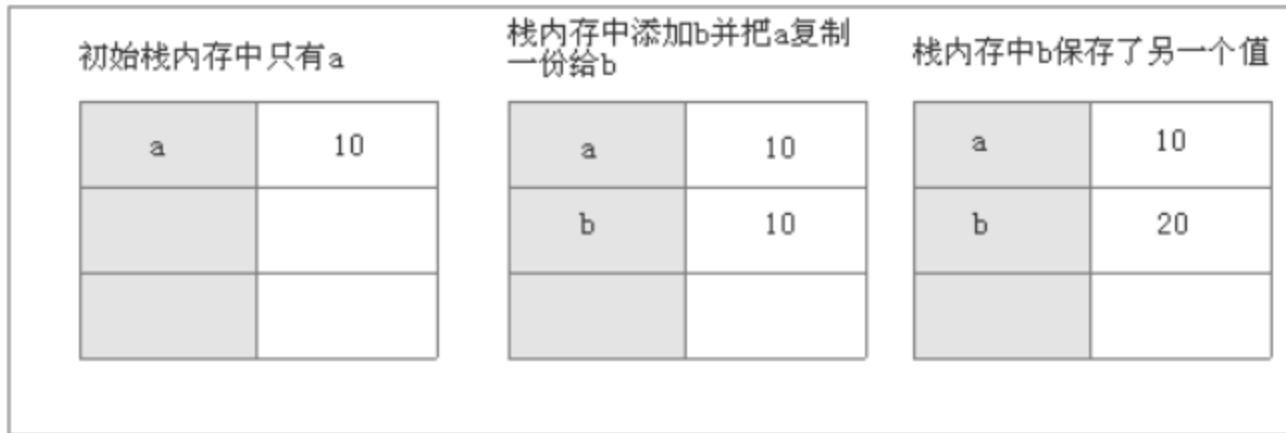
```
1 let a = 10;  
2 let b = a; // 赋值操作  
3 b = 20;  
4 console.log(a); // 10值
```

JavaScript | 复制代码

`a` 的值为一个基本类型，是存储在栈中，将 `a` 的值赋给 `b`，虽然两个变量的值相等，但是两个变量保存了两个不同的内存地址

下图演示了基本类型赋值的过程：

## 栈内存



### 1.4.2. 引用类型

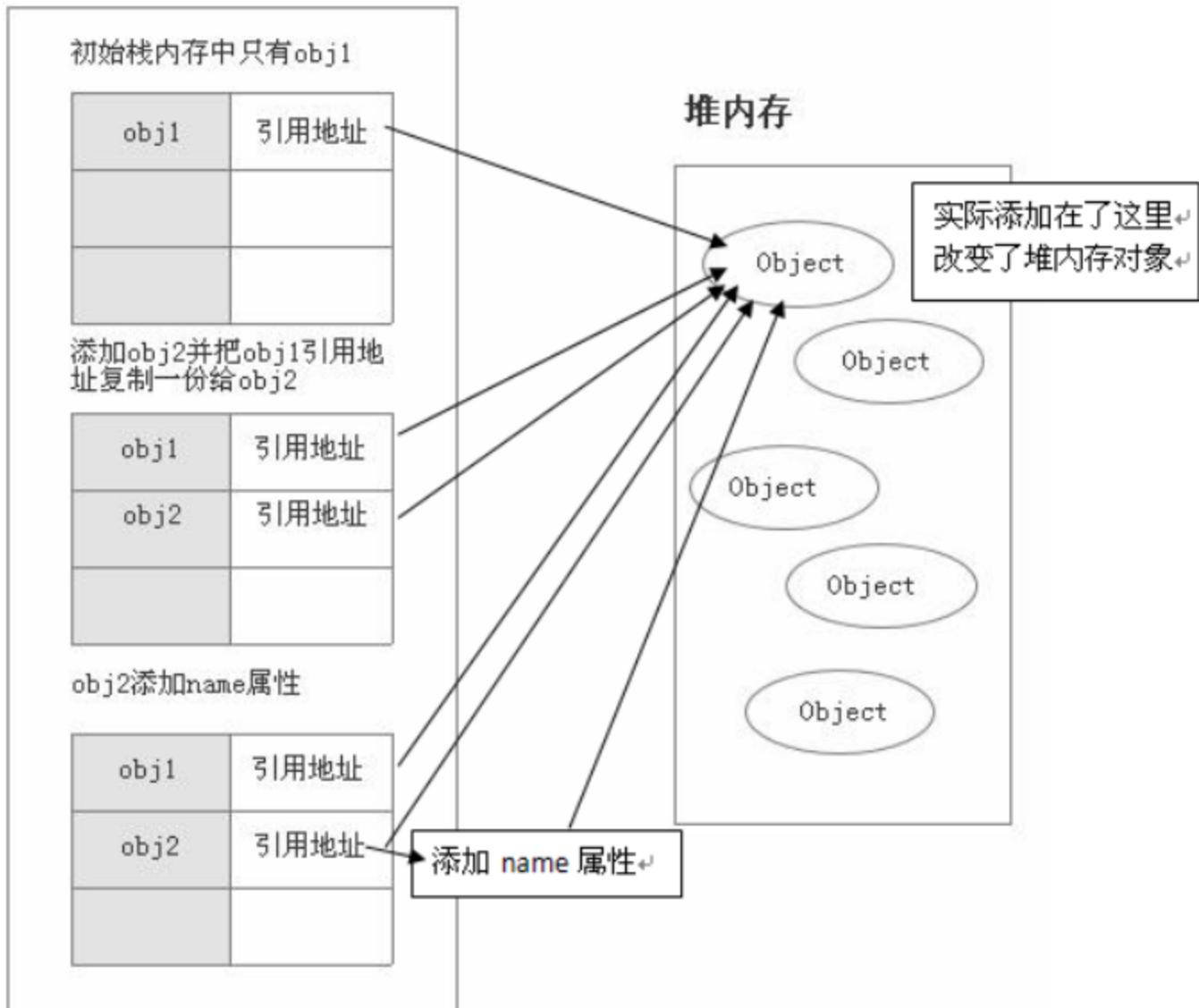
```
▼ JavaScript | 复制代码  
1 var obj1 = {}  
2 var obj2 = obj1;  
3 obj2.name = "Xxx";  
4 console.log(obj1.name); // xxx
```

引用类型数据存放在堆中，每个堆内存对象都有对应的引用地址指向它，引用地址存放在栈中。

`obj1` 是一个引用类型，在赋值操作过程汇总，实际是将堆内存对象在栈内存的引用地址复制了一份给了 `obj2`，实际上他们共同指向了同一个堆内存对象，所以更改 `obj2` 会对 `obj1` 产生影响

下图演示这个引用类型赋值过程

## 栈内存



## 1.5. 小结

- 声明变量时不同的内存地址分配：
  - 简单类型的值存放在栈中，在栈中存放的是对应的值
  - 引用类型对应的值存储在堆中，在栈中存放的是指向堆内存的地址
- 不同的类型数据导致赋值变量时的不同：
  - 简单类型赋值，是生成相同的值，两个对象对应不同的地址
  - 复杂类型赋值，是将保存对象的内存地址赋值给另一个变量。也就是两个变量指向堆内存中同一个对象

## 2. 说说你了解的js数据结构?

### 2.1. 什么是数据结构?

数据结构是计算机存储、组织数据的方式。

数据结构意味着接口或封装：一个数据结构可被视为两个函数之间的接口，或者是由数据类型联合组成的存储内容的访问方法封装。

我们每天的编码中都会用到数据结构

数组是最简单的内存数据结构

下面是常见的数据结构：

1. 数组 (Array)
2. 栈 (Stack)
3. 队列 (Queue)
4. 链表 (Linked List)
5. 字典
6. 散列表 (Hash table)
7. 树 (Tree)
8. 图 (Graph)
9. 堆 (Heap)

### 2.2. 数组 (Array)

数组是最最基本的数据结构，很多语言都内置支持数组。

数组是使用一块连续的内存空间保存数据，保存的数据的个数在分配内存的时候就是确定的。

在日常生活中，人们经常使用列表：待办事项列表、购物清单等。

而计算机程序也在使用列表，在下面的条件下，选择列表作为数据结构就显得尤为有用：

数据结构较为简单

不需要在一个长序列中查找元素，或者对其进行排序

反之，如果数据结构非常复杂，列表的作用就没有那么大了。

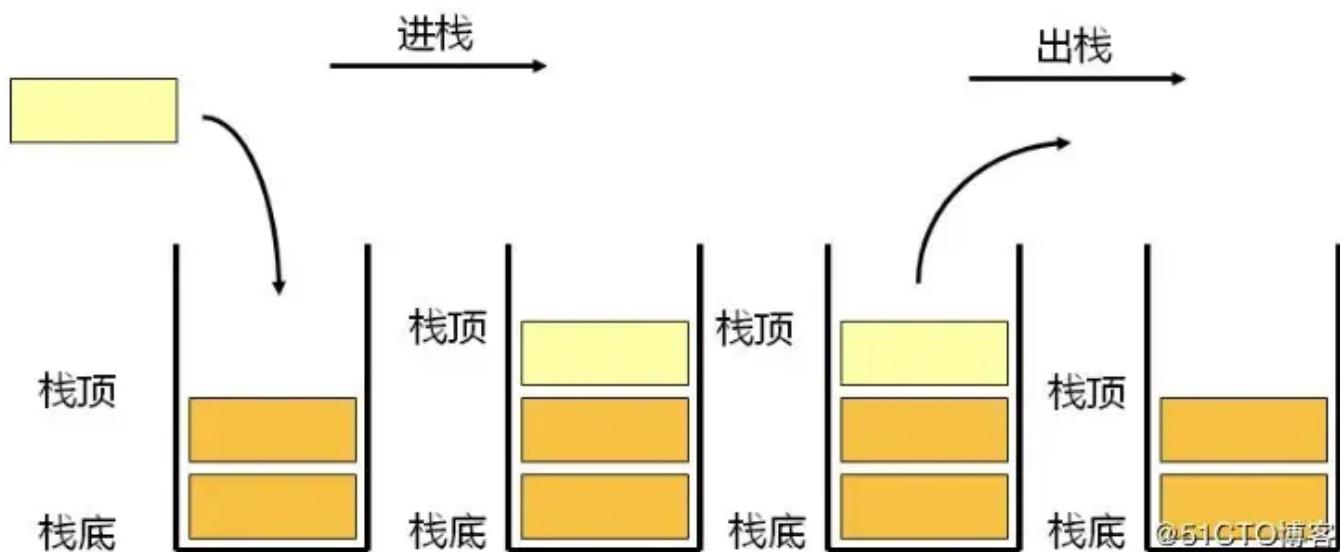
### 2.3. 栈 (Stack)

栈是一种遵循后进先出（LIFO）原则的有序集合

在栈里，新元素都接近栈顶，旧元素都接近栈底。

每次加入新的元素和拿走元素都在顶部操作

### - 后进先出 (Last In First Out)



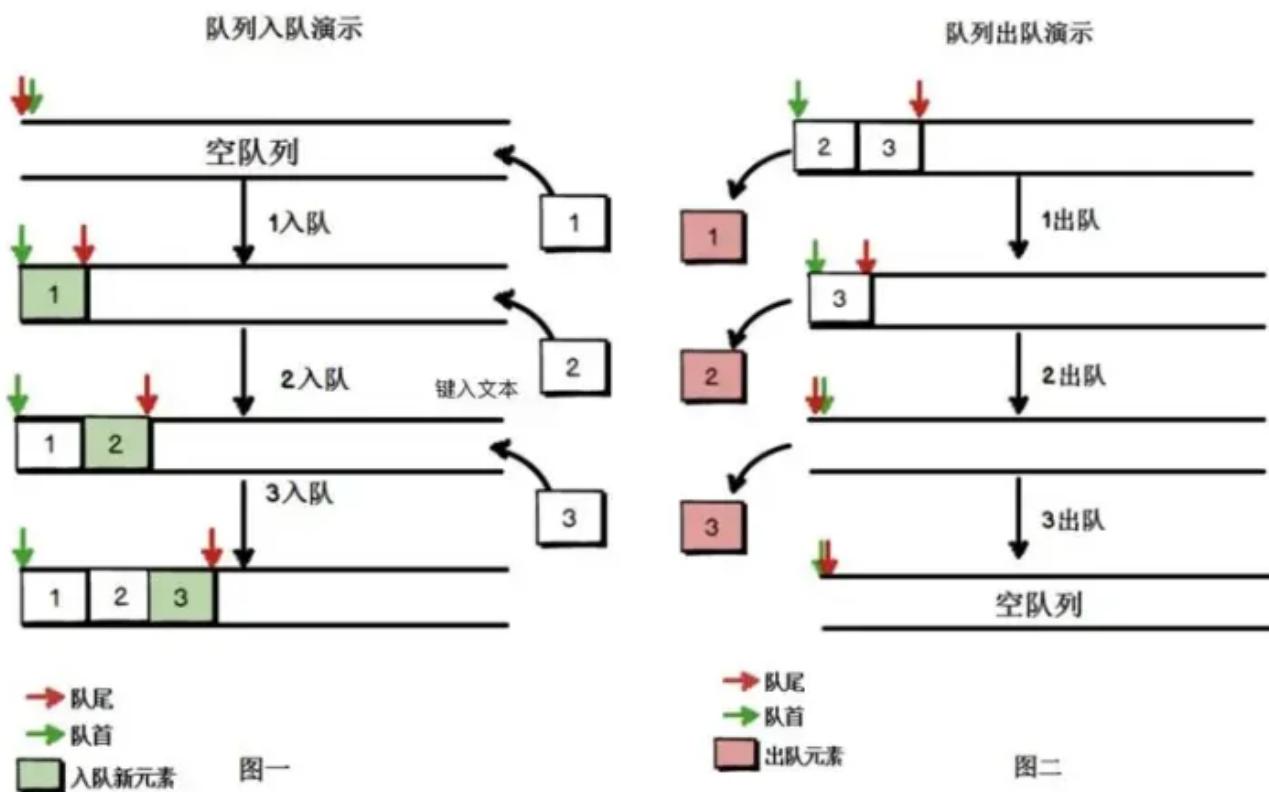
## 2.4. 队列 (Queue)

队列是遵循先进先出（FIFO，也称为先来先服务）原则的一组有序的项

队列在尾部添加新元素，并从顶部移除元素

最新添加的元素必须排在队列的末尾

### - 先进先出 (First In First Out)



## 2.5. 链表 (Linked List)

链表也是一种列表，已经设计了数组，为什么还需要链表呢？

JavaScript中数组的主要问题是，它们被实现成了对象，

与其他语言（比如C++和Java）的数组相对，效率很低。

如果你发现数组在实际使用时很慢，就可以考虑使用链表来代替它。

使用条件：

链表几乎可以在任何可以使用一维数组的情况下。

如果需要随机访问，数组仍然是更好的选择。

## 2.6. 字典

字典是一种以键-值对存储数据的数据结构，js中的Object类就是以字典的形式设计的。JavaScript可以通过实现字典类，让这种字典类型的对象使用起来更加简单，字典可以实现对象拥有的常见功能，并相应拓展自己想要的功能，而对象在JavaScript编写中随处可见，所以字典的作用也异常明显了。

## 2.7. 散列表

也称为哈希表，特点是在散列表上插入、删除和取用数据都非常快。

为什么要设计这种数据结构呢？

用数组或链表存储数据，如果想要找到其中一个数据，需要从头进行遍历，因为不知道这个数据存储到了数组的哪个位置。

散列表在JavaScript中可以基础数组去进行设计。

数组的长度是预先设定的，所有元素根据和该元素对应的键，保存在数组的特定位置，这里的键和对象的键是类型的概念。

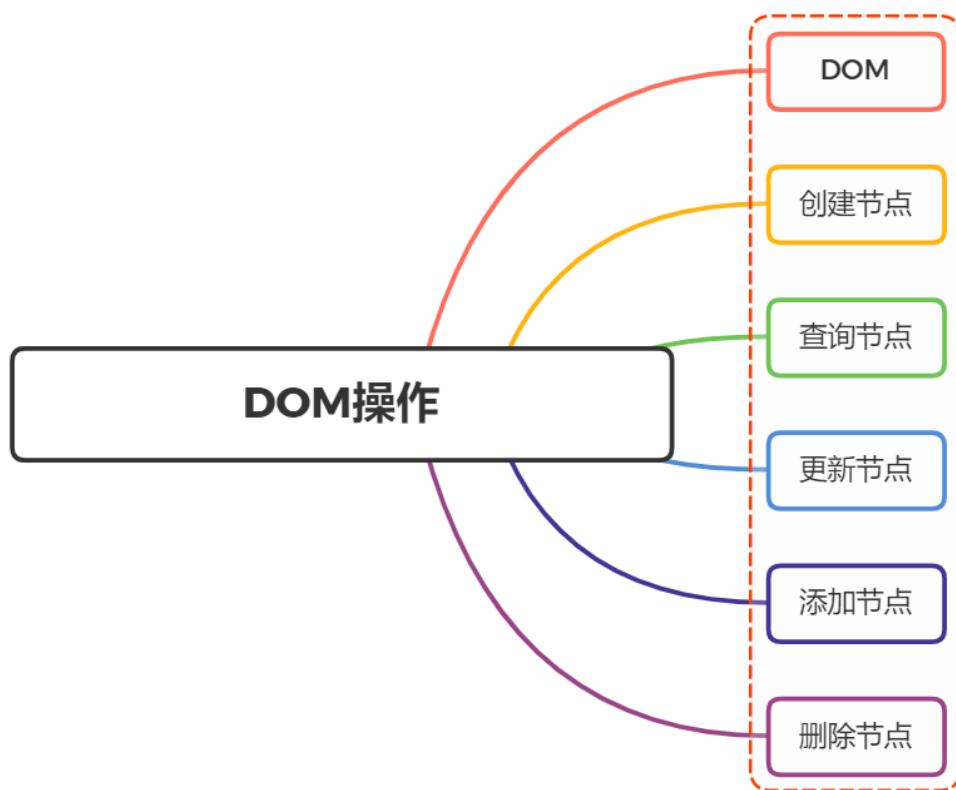
使用散列表存储数组时，通过一个散列函数将键映射为一个数字，这个数字的范围是0到散列表的长度。

即使使用一个高效的散列函数，依然存在将两个键映射为同一个值得可能，这种现象叫做碰撞。常见碰撞的处理方法有：开链法和线性探测法（具体概念有兴趣的可以网上自行了解）

使用条件：

可以用于数据的插入、删除和取用，不适用于查找数据

## 3. DOM常见的操作有哪些？



### 3.1. DOM

文档对象模型 (DOM) 是 `HTML` 和 `XML` 文档的编程接口

它提供了对文档的结构化的表述，并定义了一种方式可以使从程序中对该结构进行访问，从而改变文档的结构，样式和内容

任何 `HTML` 或 `XML` 文档都可以用 `DOM` 表示为一个由节点构成的层级结构

节点分很多类型，每种类型对应着文档中不同的信息和（或）标记，也都有自己不同的特性、数据和方法，而且与其他类型有某种关系，如下所示：

```
1 <html>
2   <head>
3     <title>Page</title>
4   </head>
5   <body>
6     <p>Hello World!</p>
7   </body>
8 </html>
```

`DOM` 像原子包含着亚原子微粒那样，也有很多类型的 `DOM` 节点包含着其他类型的节点。接下来我们先看看其中的三种：

```
1 <div>
2   <p title="title">
3     content
4   </p>
5 </div>
```

上述结构中，`div`、`p` 就是元素节点，`content` 就是文本节点，`title` 就是属性节点

## 3.2. 操作

日常前端开发，我们都离不开 `DOM` 操作

在以前，我们使用 `Jquery`，`zepto` 等库来操作 `DOM`，之后在 `vue`，`Angular`，`React` 等框架出现后，我们通过操作数据来控制 `DOM`（绝大多数时候），越来越少的去直接操作 `DOM`

但这并不代表原生操作不重要。相反，`DOM` 操作才能有助于我们理解框架深层的内容

下面就来分析 `DOM` 常见的操作，主要分为：

- 创建节点

- 查询节点
- 更新节点
- 添加节点
- 删除节点

### 3.2.1. 创建节点

#### 3.2.1.1. createElement

创建新元素，接受一个参数，即要创建元素的标签名

```
▼  
1 const divEl = document.createElement("div");  
JavaScript | ⌂ 复制代码
```

#### 3.2.1.2. createTextNode

创建一个文本节点

```
▼  
1 const textEl = document.createTextNode("content");  
JavaScript | ⌂ 复制代码
```

#### 3.2.1.3. createDocumentFragment

用来创建一个文档碎片，它表示一种轻量级的文档，主要是用来存储临时节点，然后把文档碎片的内容一次性添加到 `DOM` 中

```
▼  
1 const fragment = document.createDocumentFragment();  
JavaScript | ⌂ 复制代码
```

当请求把一个 `DocumentFragment` 节点插入文档树时，插入的不是 `DocumentFragment` 自身，而是它的所有子孙节点

#### 3.2.1.4. createAttribute

创建属性节点，可以是自定义属性

JavaScript | 复制代码

```
1 const dataAttribute = document.createAttribute('custom');
2 consle.log(dataAttribute);
```

## 3.2.2. 获取节点

### 3.2.2.1. querySelector

传入任何有效的 `css` 选择器，即可选中单个 `DOM` 元素（首个）：

JavaScript | 复制代码

```
1 document.querySelector('.element')
2 document.querySelector('#element')
3 document.querySelector('div')
4 document.querySelector('[name="username"]')
5 document.querySelector('div + p > span')
```

如果页面上没有指定的元素时，返回 `null`

### 3.2.2.2. querySelectorAll

返回一个包含节点子树内所有与之相匹配的 `Element` 节点列表，如果没有相匹配的，则返回一个空节点列表

JavaScript | 复制代码

```
1 const notLive = document.querySelectorAll("p");
```

需要注意的是，该方法返回的是一个 `NodeList` 的静态实例，它是一个静态的“快照”，而非“实时”的查询

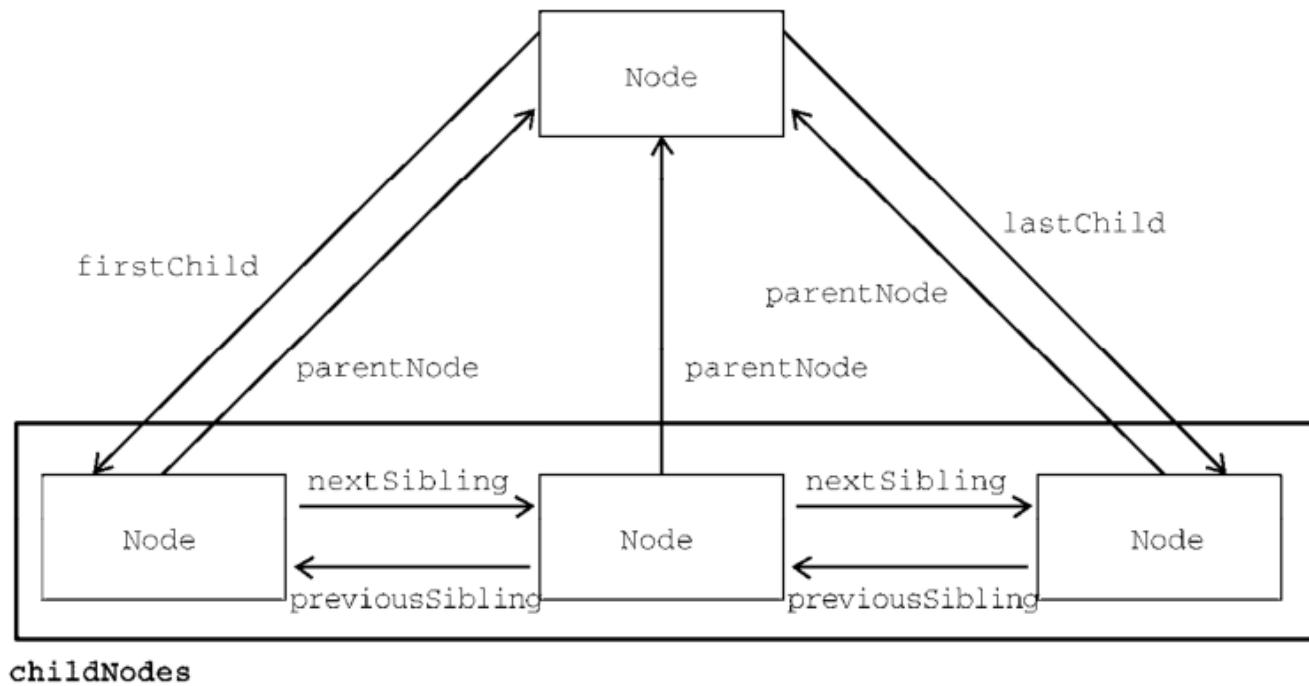
关于获取 `DOM` 元素的方法还有如下，就不一一述说

```

1 document.getElementById('id属性值'); 返回拥有指定id的对象的引用
2 document.getElementsByClassName('class属性值'); 返回拥有指定class的对象集合
3 document.getElementsByTagName('标签名'); 返回拥有指定标签名的对象集合
4 document.getElementsByName('name属性值'); 返回拥有指定名称的对象结合
5 document/element.querySelector('CSS选择器'); 仅返回第一个匹配的元素
6 document/element.querySelectorAll('CSS选择器'); 返回所有匹配的元素
7 document.documentElement; 获取页面中的HTML标签
8 document.body; 获取页面中的BODY标签
9 document.all['']; 获取页面中的所有元素节点的对象集合型

```

除此之外，每个 DOM 元素还有 `parentNode`、`childNodes`、`firstChild`、`lastChild`、`nextSibling`、`previousSibling` 属性，关系图如下图所示



### 3.2.3. 更新节点

#### 3.2.3.1. innerHTML

不但可以修改一个 DOM 节点的文本内容，还可以直接通过 HTML 片段修改 DOM 节点内部的子树

JavaScript | 复制代码

```
1 // 获取<p id="p">...</p>
2 var p = document.getElementById('p');
3 // 设置文本为abc:
4 p.innerHTML = 'ABC'; // <p id="p">ABC</p>
5 // 设置HTML:
6 p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';
7 // <p>...</p>的内部结构已修改
```

### 3.2.3.2. innerText、textContent

自动对字符串进行 HTML 编码，保证无法设置任何 HTML 标签

Plain Text | 复制代码

```
1 // 获取<p id="p-id">...</p>
2 var p = document.getElementById('p-id');
3 // 设置文本:
4 p.innerText = '<script>alert("Hi")</script>';
5 // HTML被自动编码，无法设置一个<script>节点:
6 // <p id="p-id">&lt;script&gt;alert("Hi")&lt;/script&gt;</p>
```

两者的区别在于读取属性时，`innerText` 不返回隐藏元素的文本，而 `textContent` 返回所有文本

### 3.2.3.3. style

DOM 节点的 `style` 属性对应所有的 CSS，可以直接获取或设置。遇到 - 需要转化为驼峰命名

JavaScript | 复制代码

```
1 // 获取<p id="p-id">...</p>
2 const p = document.getElementById('p-id');
3 // 设置CSS:
4 p.style.color = '#ff0000';
5 p.style.fontSize = '20px'; // 驼峰命名
6 p.style.paddingTop = '2em';
```

## 3.2.4. 添加节点

### 3.2.4.1. innerHTML

如果这个DOM节点是空的，例如，`<div></div>`，那么，直接使用`innerHTML = '<span>child</span>'`就可以修改DOM节点的内容，相当于添加了新的DOM节点

如果这个DOM节点不是空的，那就不能这么做，因为`innerHTML`会直接替换掉原来的所有子节点

### 3.2.4.2. appendChild

把一个子节点添加到父节点的最后一个子节点

举个例子

```
1 <!-- HTML结构 -->
2 <p id="js">JavaScript</p >
3 <div id="list">
4   <p id="java">Java</p >
5   <p id="python">Python</p >
6   <p id="scheme">Scheme</p >
7 </div>
```

添加一个`p`元素

```
1 const js = document.getElementById('js')
2 js.innerHTML = "JavaScript"
3 const list = document.getElementById('list');
4 list.appendChild(js);
```

现在HTML结构变成了下面

```
1 <!-- HTML结构 -->
2 <div id="list">
3   <p id="java">Java</p >
4   <p id="python">Python</p >
5   <p id="scheme">Scheme</p >
6   <p id="js">JavaScript</p > <!-- 添加元素 -->
7 </div>
```

上述代码中，我们是获取DOM元素后再进行添加操作，这个`js`节点是已经存在当前文档树中，因此这个节点首先会从原先的位置删除，再插入到新的位置

如果动态添加新的节点，则先创建一个新的节点，然后插入到指定的位置

JavaScript | 复制代码

```
1 const list = document.getElementById('list'),  
2 const haskell = document.createElement('p');  
3 haskell.id = 'haskell';  
4 haskell.innerText = 'Haskell';  
5 list.appendChild(haskell);
```

### 3.2.4.3. insertBefore

把子节点插入到指定的位置，使用方法如下：

JavaScript | 复制代码

```
1 parentElement.insertBefore(newElement, referenceElement)
```

子节点会插入到 `referenceElement` 之前

### 3.2.4.4. setAttribute

在指定元素中添加一个属性节点，如果元素中已有该属性改变属性值

JavaScript | 复制代码

```
1 const div = document.getElementById('id')  
2 div.setAttribute('class', 'white');//第一个参数属性名，第二个参数属性值。
```

## 3.2.5. 删除节点

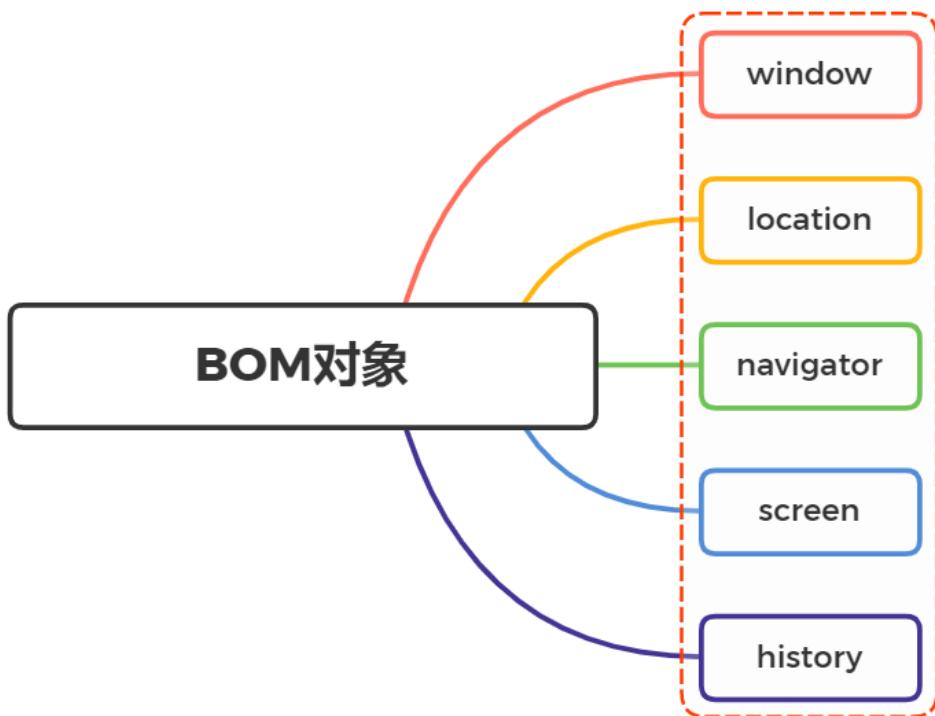
删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的 `removeChild` 把自己删掉

JavaScript | 复制代码

```
1 // 拿到待删除节点：  
2 const self = document.getElementById('to-be-removed');  
3 // 拿到父节点：  
4 const parent = self.parentElement;  
5 // 删除：  
6 const removed = parent.removeChild(self);  
7 removed === self; // true
```

删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置

## 4. 说说你对BOM的理解，常见的BOM对象你了解哪些？



### 4.1. 是什么

BOM (Browser Object Model)，浏览器对象模型，提供了独立于内容与浏览器窗口进行交互的对象。其作用就是跟浏览器做一些交互效果,比如如何进行页面的后退，前进，刷新，浏览器的窗口发生变化，滚动条的滚动，以及获取客户的一些信息如：浏览器品牌版本，屏幕分辨率。

浏览器的全部内容可以看成 DOM，整个浏览器可以看成 BOM。区别如下：

#### DOM

- 文档对象模型
- DOM 就是把「文档」当做一个「对象」来看待
- DOM 的顶级对象是 `document`
- DOM 主要学习的是操作页面元素
- DOM 是 W3C 标准规范

#### BOM

- 浏览器对象模型
- 把「浏览器」当做一个「对象」来看待
- BOM 的顶级对象是 `window`
- BOM 学习的是浏览器窗口交互的一些对象
- BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

### 4.2. window

Bom 的核心对象是 `window`，它表示浏览器的一个实例

在浏览器中，`window` 对象有双重角色，即是浏览器窗口的一个接口，又是全局对象

因此所有在全局作用域中声明的变量、函数都会变成 `window` 对象的属性和方法

```
1 var name = 'js每日一题';
2 function lookName(){
3     alert(this.name);
4 }
5
6 console.log(window.name); //js每日一题
7 lookName(); //js每日一题
8 window.lookName(); //js每日一题
```

关于窗口控制方法如下：

- `moveBy(x,y)`：从当前位置水平移动窗体x个像素，垂直移动窗体y个像素，x为负数，将向左移动窗体，y为负数，将向上移动窗体
- `moveTo(x,y)`：移动窗体左上角到相对于屏幕左上角的(x,y)点
- `resizeBy(w,h)`：相对窗体当前的大小，宽度调整w个像素，高度调整h个像素。如果参数为负值，将缩小窗体，反之扩大窗体
- `resizeTo(w,h)`：把窗体宽度调整为w个像素，高度调整为h个像素
- `scrollTo(x,y)`：如果有滚动条，将横向滚动条移动到相对于窗体宽度为x个像素的位置，将纵向滚动条移动到相对于窗体高度为y个像素的位置
- `scrollBy(x,y)`：如果有滚动条，将横向滚动条向左移动x个像素，将纵向滚动条向下移动y个像素

`window.open()` 既可以导航到一个特定的 `url`，也可以打开一个新的浏览器窗口

如果 `window.open()` 传递了第二个参数，且该参数是已有窗口或者框架的名称，那么就会在目标窗口加载第一个参数指定的URL

```
1 window.open('http://www.vue3js.cn','topFrame')
2 ==> <a href="" target="topFrame"></a>
```

`window.open()` 会返回新窗口的引用，也就是新窗口的 `window` 对象

```
1 const myWin = window.open('http://www.vue3js.cn','myWin')
```

`window.close()` 仅用于通过 `window.open()` 打开的窗口

新创建的 `window` 对象有一个 `opener` 属性，该属性指向打开他的原始窗口对象

## 4.3. location

`url` 地址如下：



JavaScript | 复制代码

```
1 http://foouser:barpassword@www.wrox.com:80/WileyCDA/?q=javascript#contents
```

`location` 属性描述如下：

属性名	例子	说明
hash	"#contents"	url中#后面的字符，没有则返回空串
host	www.wrox.com:80	服务器名称和端口号
hostname	www.wrox.com	域名，不带端口号
href	<a href="http://www.wrox.com:80/WileyCDA/?q=javascript#contents">http://www.wrox.com:80/WileyCDA/?q=javascript#contents</a>	完整url
pathname	"/WileyCDA/"	服务器下面的文件路径
port	80	url的端口号，没有则为空
protocol	http:	使用的协议
search	?q=javascript	url的查询字符串，通常为?后面的内容

除了 `hash` 之外，只要修改 `location` 的一个属性，就会导致页面重新加载新 `URL`

`location.reload()`，此方法可以重新刷新当前页面。这个方法会根据最有效的方式刷新页面，如果页面自上一次请求以来没有改变过，页面就会从浏览器缓存中重新加载

如果要强制从服务器中重新加载，传递一个参数 `true` 即可

## 4.4. navigator

`navigator` 对象主要用来获取浏览器的属性，区分浏览器类型。属性较多，且兼容性比较复杂

下表列出了 `navigator` 对象接口定义的属性和方法：

属性/方法	说 明
activeVrDisplays	返回数组，包含 <code>isPresenting</code> 属性为 <code>true</code> 的 <code>VRDisplay</code> 实例
appCodeName	即使在非 Mozilla 浏览器中也会返回 "Mozilla"
appName	浏览器全名
appVersion	浏览器版本。通常与实际的浏览器版本不一致
battery	返回暴露 <code>Battery Status API</code> 的 <code>BatteryManager</code> 对象
buildId	浏览器的构建编号
connection	返回暴露 <code>Network Information API</code> 的 <code>NetworkInformation</code> 对象
cookieEnabled	返回布尔值，表示是否启用了 cookie
credentials	返回暴露 <code>Credentials Management API</code> 的 <code>CredentialsContainer</code> 对象
deviceMemory	返回单位为 GB 的设备内存容量
doNotTrack	返回用户的“不跟踪”( <code>do-not-track</code> )设置
geolocation	返回暴露 <code>Geolocation API</code> 的 <code>Geolocation</code> 对象
getVRDisplays()	返回数组，包含可用的每个 <code>VRDisplay</code> 实例
getUserMedia()	返回与可用媒体设备硬件关联的流
hardwareConcurrency	返回设备的处理器核心数量
javaEnabled	返回布尔值，表示浏览器是否启用了 Java
language	返回浏览器的主语言
languages	返回浏览器偏好的语言数组

locks	返回暴露 Web Locks API 的 LockManager 对象
mediaCapabilities	返回暴露 Media Capabilities API 的 MediaCapabilities 对象
mediaDevices	返回可用的媒体设备
maxTouchPoints	返回设备触摸屏支持的最大触点数
mimeTypes	返回浏览器中注册的 MIME 类型数组
onLine	返回布尔值，表示浏览器是否联网
oscpu	返回浏览器运行设备的操作系统和（或）CPU
permissions	返回暴露 Permissions API 的 Permissions 对象
platform	返回浏览器运行的系统平台
plugins	返回浏览器安装的插件数组。在 IE 中，这个数组包含页面中所有<embed>元素
product	返回产品名称（通常是 "Gecko"）
productSub	返回产品的额外信息（通常是 Gecko 的版本）
registerProtocolHandler()	将一个网站注册为特定协议的处理程序
requestMediaKeySystemAccess()	返回一个期约，解决为 MediaKeySystemAccess 对象
sendBeacon()	异步传输一些小数据
serviceWorker	返回用来与 ServiceWorker 实例交互的 ServiceWorkerContainer
share()	返回当前平台的原生共享机制
storage	返回暴露 Storage API 的 StorageManager 对象
userAgent	返回浏览器的用户代理字符串
vendor	返回浏览器的厂商名称
vendorSub	返回浏览器厂商的更多信息
vibrate()	触发设备振动
webdriver	返回浏览器当前是否被自动化程序控制

## 4.5. screen

保存的纯粹是客户端能力信息，也就是浏览器窗口外面的客户端显示器的信息，比如像素宽度和像素高度

属性	说明
availHeight	屏幕像素高度减去系统组件高度（只读）
availLeft	没有被系统组件占用的屏幕的最左侧像素（只读）
availTop	没有被系统组件占用的屏幕的最顶端像素（只读）
availWidth	屏幕像素宽度减去系统组件宽度（只读）
colorDepth	表示屏幕颜色的位数；多数系统是 32（只读）
height	屏幕像素高度
left	当前屏幕左边的像素距离
pixelDepth	屏幕的位深（只读）
top	当前屏幕顶端的像素距离
width	屏幕像素宽度
orientation	返回 Screen Orientation API 中屏幕的朝向

## 4.6. history

`history` 对象主要用来操作浏览器 `URL` 的历史记录，可以通过参数向前，向后，或者向指定 `URL` 跳转

常用的属性如下：

- `history.go()`

接收一个整数数字或者字符串参数：向最近的一个记录中包含指定字符串的页面跳转，

```
▼
JavaScript | ⌂ 复制代码

1 history.go('maixiaofei.com')
```

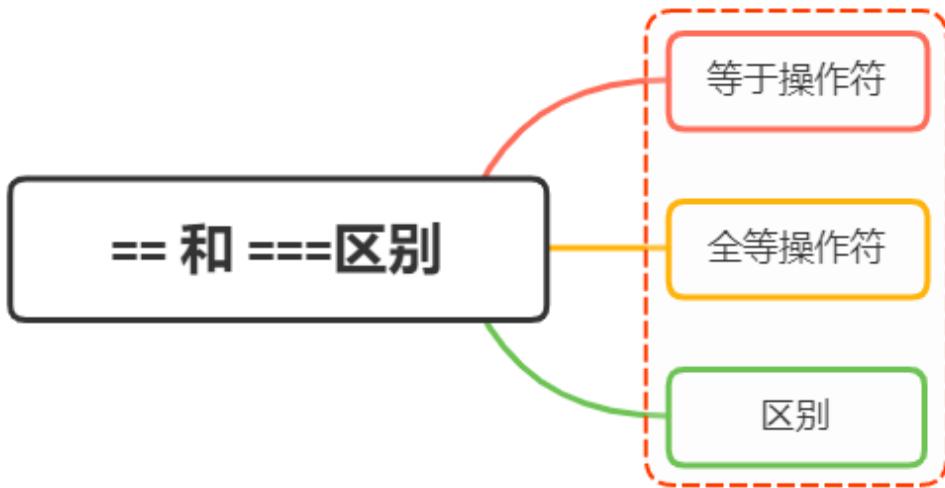
当参数为整数数字的时候，正数表示向前跳转指定的页面，负数为向后跳转指定的页面

```
▼
JavaScript | ⌂ 复制代码

1 history.go(3) //向前跳转三个记录
2 history.go(-1) //向后跳转一个记录
```

- `history.forward()`：向前跳转一个页面
- `history.back()`：向后跳转一个页面
- `history.length`：获取历史记录数

## 5. == 和 ===区别，分别在什么情况使用



## 5.1. 等于操作符

等于操作符用两个等于号（`==`）表示，如果操作数相等，则会返回 `true`

前面文章，我们提到在 `JavaScript` 中存在隐式转换。等于操作符（`==`）在比较中会先进行类型转换，再确定操作数是否相等

遵循以下规则：

如果任一操作数是布尔值，则将其转换为数值再比较是否相等

```
▼
JavaScript | ⌂ 复制代码
1 let result1 = (true == 1); // true
```

如果一个操作数是字符串，另一个操作数是数值，则尝试将字符串转换为数值，再比较是否相等

```
▼
JavaScript | ⌂ 复制代码
1 let result1 = ("55" == 55); // true
```

如果一个操作数是对象，另一个操作数不是，则调用对象的 `valueOf()` 方法取得其原始值，再根据前面的规则进行比较

```
▼
JavaScript | ⌂ 复制代码
1 let obj = {valueOf:function(){return 1}}
2 let result1 = (obj == 1); // true
```

`null` 和 `undefined` 相等

JavaScript | 复制代码

```
1 let result1 = (null == undefined); // true
```

如果有任一操作数是 `NaN`，则相等操作符返回 `false`

JavaScript | 复制代码

```
1 let result1 = (NaN == NaN); // false
```

如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 `true`

Plain Text | 复制代码

```
1 let obj1 = {name:"xxx"}  
2 let obj2 = {name:"xxx"}  
3 let result1 = (obj1 == obj2); // false
```

下面进一步做个小结：

- 两个都为简单类型，字符串和布尔值都会转换成数值，再比较
- 简单类型与引用类型比较，对象转化成其原始类型的值，再比较
- 两个都为引用类型，则比较它们是否指向同一个对象
- `null` 和 `undefined` 相等
- 存在 `NaN` 则返回 `false`

## 5.2. 全等操作符

全等操作符由 3 个等于号（`==`）表示，只有两个操作数在不转换的前提下相等才返回 `true`。即类型相同，值也需相同

JavaScript | 复制代码

```
1 let result1 = ("55" === 55); // false, 不相等，因为数据类型不同  
2 let result2 = (55 === 55); // true, 相等，因为数据类型相同值也相同
```

`undefined` 和 `null` 与自身严格相等

JavaScript | 复制代码

```
1 let result1 = (null === null) //true
2 let result2 = (undefined === undefined) //true
```

## 5.3. 区别

相等操作符（==）会做类型转换，再进行值的比较，全等运算符不会做类型转换

JavaScript | 复制代码

```
1 let result1 = ("55" === 55); // false, 不相等, 因为数据类型不同
2 let result2 = (55 === 55); // true, 相等, 因为数据类型相同值也相同
```

null 和 undefined 比较，相等操作符（==）为 true，全等为 false

JavaScript | 复制代码

```
1 let result1 = (null == undefined); // true
2 let result2 = (null === undefined); // false
```

## 5.4. 小结

相等运算符隐藏的类型转换，会带来一些违反直觉的结果

JavaScript | 复制代码

```
1 '' == '0' // false
2 0 == '' // true
3 0 == '0' // true
4
5 false == 'false' // false
6 false == '0' // true
7
8 false == undefined // false
9 false == null // false
10 null == undefined // true
11
12 '\t\r\n' == 0 // true
```

但在比较 null 的情况的时候，我们一般使用相等操作符 ==

JavaScript | 复制代码

```
1 const obj = {};
2
3 if(obj.x == null){
4     console.log("1"); //执行
5 }
```

等同于下面写法

JavaScript | 复制代码

```
1 if(obj.x === null || obj.x === undefined) {
2     ...
3 }
```

使用相等操作符（==）的写法明显更加简洁了

所以，除了在比较对象属性为 `null` 或者 `undefined` 的情况下，我们可以使用相等操作符（==），其他情况一律使用全等操作符

## 6. `typeof` 与 `instanceof` 区别



### 6.1. `typeof`

`typeof` 操作符返回一个字符串，表示未经计算的操作数的类型

使用方法如下：

JavaScript | 复制代码

```
1  typeof operand
2  typeof(operand)
```

`operand` 表示对象或原始值的表达式，其类型将被返回

举个例子

JavaScript | 复制代码

```
1  typeof 1 // 'number'
2  typeof '1' // 'string'
3  typeof undefined // 'undefined'
4  typeof true // 'boolean'
5  typeof Symbol() // 'symbol'
6  typeof null // 'object'
7  typeof [] // 'object'
8  typeof {} // 'object'
9  typeof console // 'object'
10 typeof console.log // 'function'
```

从上面例子，前6个都是基础数据类型。虽然 `typeof null` 为 `object`，但这只是 JavaScript 存在的一个悠久 Bug，不代表 `null` 就是引用数据类型，并且 `null` 本身也不是对象

所以，`null` 在 `typeof` 之后返回的是有问题的结果，不能作为判断 `null` 的方法。如果你需要在 `if` 语句中判断是否为 `null`，直接通过 `==null` 来判断就好

同时，可以发现引用类型数据，用 `typeof` 来判断的话，除了 `function` 会被识别出来之外，其余的都输出 `object`

如果我们想要判断一个变量是否存在，可以使用 `typeof`：（不能使用 `if(a)`，若 `a` 未声明，则报错）

JavaScript | 复制代码

```
1 if(typeof a != 'undefined'){
2     //变量存在
3 }
```

## 6.2. instanceof

`instanceof` 运算符用于检测构造函数的 `prototype` 属性是否出现在某个实例对象的原型链上

使用如下：

JavaScript | 复制代码

```
1 object instanceof constructor
```

`object` 为实例对象, `constructor` 为构造函数

构造函数通过 `new` 可以实例对象, `instanceof` 能判断这个对象是否是之前那个构造函数生成的对象

JavaScript | 复制代码

```
1 // 定义构建函数
2 let Car = function() {}
3 let benz = new Car()
4 benz instanceof Car // true
5 let car = new String('xxx')
6 car instanceof String // true
7 let str = 'xxx'
8 str instanceof String // false
```

关于 `instanceof` 的实现原理, 可以参考下面:

JavaScript | 复制代码

```
1 function myInstanceof(left, right) {
2     // 这里先用typeof来判断基础数据类型, 如果是, 直接返回false
3     if(typeof left !== 'object' || left === null) return false;
4     // getPrototypeOf是Object对象自带的API, 能够拿到参数的原型对象
5     let proto = Object.getPrototypeOf(left);
6     while(true) {
7         if(proto === null) return false;
8         if(proto === right.prototype) return true; // 找到相同原型对象, 返回true
9         proto = Object.getPrototypeOf(proto);
10    }
11 }
```

也就是顺着原型链去找, 直到找到相同的原型对象, 返回 `true`, 否则为 `false`

## 6.3. 区别

`typeof` 与 `instanceof` 都是判断数据类型的方法, 区别如下:

- `typeof` 会返回一个变量的基本类型, `instanceof` 返回的是一个布尔值
- `instanceof` 可以准确地判断复杂引用数据类型, 但是不能正确判断基础数据类型

- 而 `typeof` 也存在弊端，它虽然可以判断基础数据类型（`null` 除外），但是引用数据类型中，除了 `function` 类型以外，其他的也无法判断

可以看到，上述两种方法都有弊端，并不能满足所有场景的需求

如果需要通用检测数据类型，可以采用 `Object.prototype.toString`，调用该方法，统一返回格式 “[object Xxx]” 的字符串

如下

```
▼ JavaScript | 复制代码
1 Object.prototype.toString({})      // "[object Object]"
2 Object.prototype.toString.call({}) // 同上结果，加上call也ok
3 Object.prototype.toString.call(1)   // "[object Number]"
4 Object.prototype.toString.call('1') // "[object String]"
5 Object.prototype.toString.call(true) // "[object Boolean]"
6 Object.prototype.toString.call(function(){}) // "[object Function]"
7 Object.prototype.toString.call(null) // "[object Null]"
8 Object.prototype.toString.call(undefined) // "[object Undefined]"
9 Object.prototype.toString.call(/123/g) // "[object RegExp]"
10 Object.prototype.toString.call(new Date()) // "[object Date]"
11 Object.prototype.toString.call([]) // "[object Array]"
12 Object.prototype.toString.call(document) // "[object HTMLDocument]"
13 Object.prototype.toString.call(window) // "[object Window]"
```

了解了 `toString` 的基本用法，下面就实现一个全局通用的数据类型判断方法

```
▼ JavaScript | 复制代码
1 function getType(obj){
2     let type = typeof obj;
3     if (type !== "object") { // 先进行typeof判断，如果是基础数据类型，直接返回
4         return type;
5     }
6     // 对于typeof返回结果是object的，再进行如下的判断，正则返回结果
7     return Object.prototype.toString.call(obj).replace(/^\[object (\S+)\]\$/,
8         '$1');
```

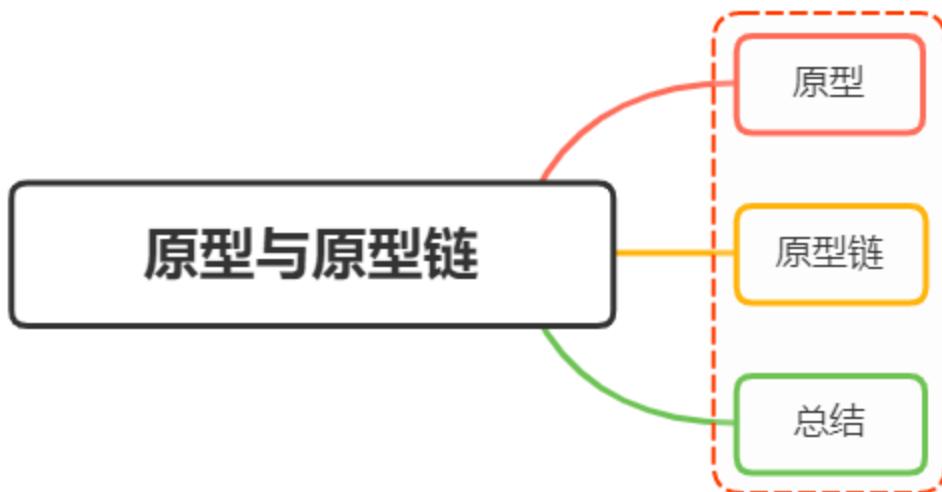
使用如下

```

1  getType([])      // "Array" typeof []是object, 因此toString返回
2  getType('123')   // "string" typeof 直接返回
3  getType(window)  // "Window" toString返回
4  getType(null)    // "Null"首字母大写, typeof null是object, 需toString来判断
5  getType(undefined) // "undefined" typeof 直接返回
6  getType()         // "undefined" typeof 直接返回
7  getType(function(){}) // "function" typeof能判断, 因此首字母小写
8  getType(/123/g)   // "RegExp" toString返回

```

## 7. JavaScript原型，原型链？有什么特点？



### 7.1. 原型

**JavaScript** 常被描述为一种基于原型的语言——每个对象拥有一个原型对象

当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾

准确地说，这些属性和方法定义在Object的构造器函数（constructor functions）之上  
的 `prototype` 属性上，而非实例对象本身

下面举个例子：

函数可以有属性。每个函数都有一个特殊的属性叫作原型 `prototype`

JavaScript | 复制代码

```
1 function doSomething(){}
2 console.log( doSomething.prototype );
```

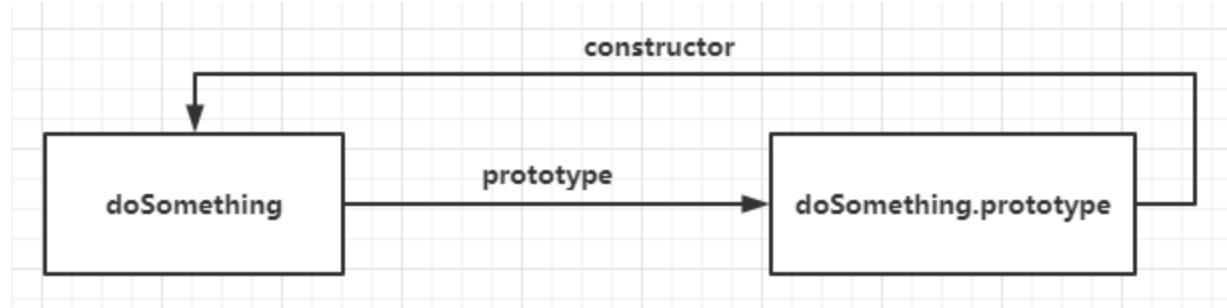
控制台输出

JavaScript | 复制代码

```
1 {
2     constructor: f doSomething(),
3     __proto__: {
4         constructor: f Object(),
5         hasOwnProperty: f hasOwnProperty(),
6         isPrototypeOf: f isPrototypeOf(),
7         propertyIsEnumerable: f propertyIsEnumerable(),
8         toLocaleString: f toLocaleString(),
9         toString: f toString(),
10        valueOf: f valueOf()
11    }
12 }
```

上面这个对象，就是大家常说的原型对象

可以看到，原型对象有一个自有属性 `constructor`，这个属性指向该函数，如下图关系展示



## 7.2. 原型链

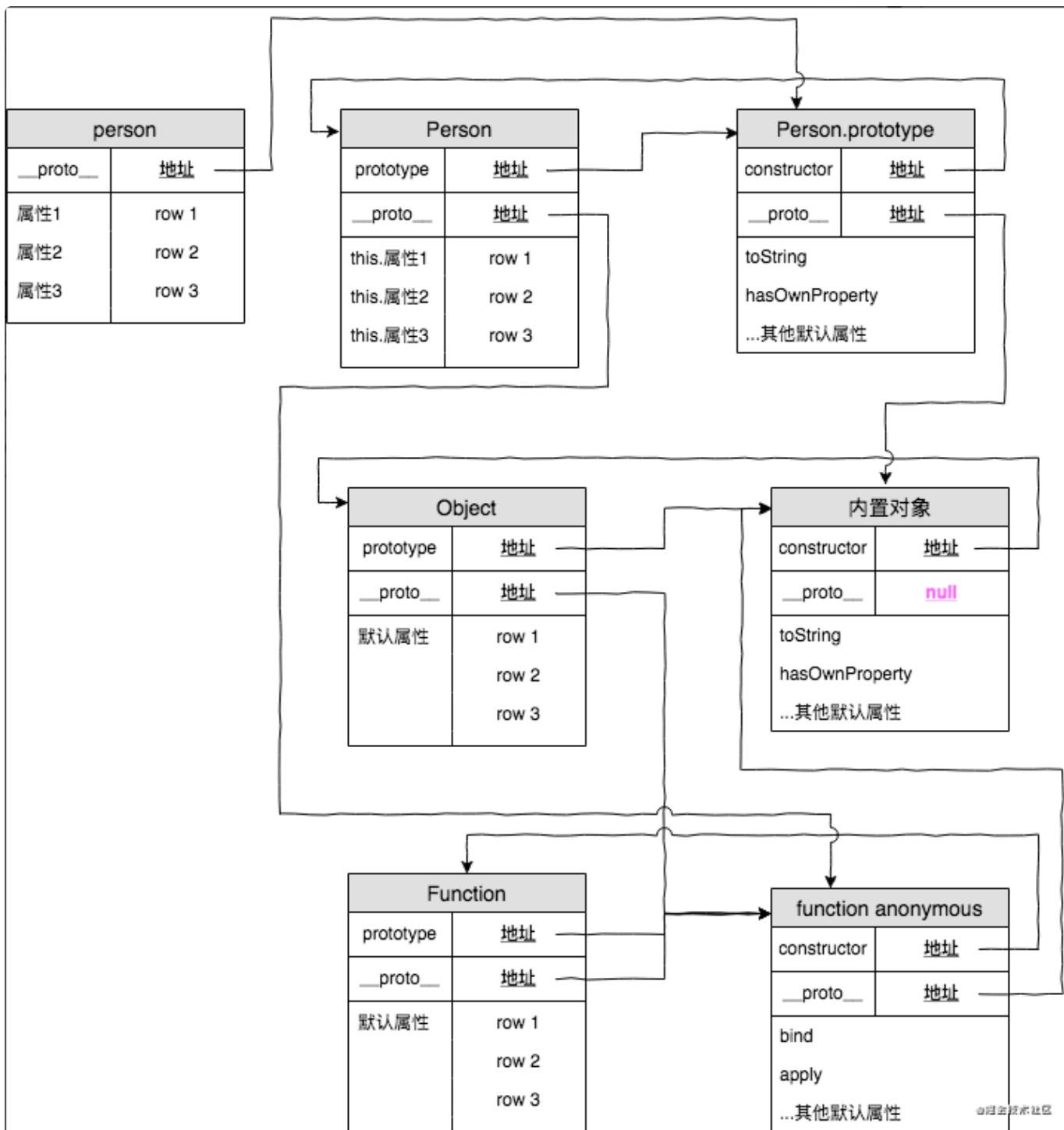
原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链 (prototype chain)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法

在对象实例和它的构造器之间建立一个链接（它是 `__proto__` 属性，是从构造函数的 `prototype` 属性派生的），之后通过上溯原型链，在构造器中找到这些属性和方法

下面举个例子：

```
1 function Person(name) {  
2     this.name = name;  
3     this.age = 18;  
4     this.sayName = function() {  
5         console.log(this.name);  
6     }  
7 }  
8 // 第二步 创建实例  
9 var person = new Person('person')
```

根据代码，我们可以得到下图



下面分析一下：

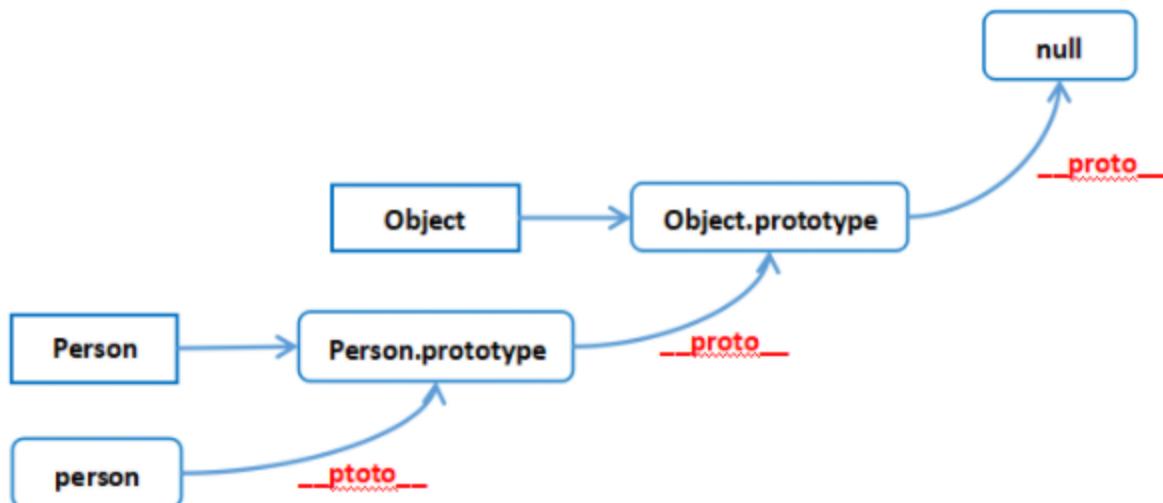
- 构造函数 `Person` 存在原型对象 `Person.prototype`
- 构造函数生成实例对象 `person`，`person` 的 `__proto__` 指向构造函数 `Person` 原型对象
- `Person.prototype.__proto__` 指向内置对象，因为 `Person.prototype` 是个对象，默认是由 `Object` 函数作为类创建的，而 `Object.prototype` 为内置对象
- `Person.__proto__` 指向内置匿名函数 `anonymous`，因为 `Person` 是个函数对象，默认由 `Function` 作为类创建

- `Function.prototype` 和 `Function.__proto__` 同时指向内置匿名函数 `anonymous`，这样原型链的终点就是 `null`

## 7.3. 总结

下面首先要看几个概念：

`__proto__` 作为不同对象之间的桥梁，用来指向创建它的构造函数的原型对象的



每个对象的 `__proto__` 都是指向它的构造函数的原型对象 `prototype` 的

```

▼
  JavaScript | 复制代码
1 person1.__proto__ === Person.prototype
  
```

构造函数是一个函数对象，是通过 `Function` 构造器产生的

```

▼
  JavaScript | 复制代码
1 Person.__proto__ === Function.prototype
  
```

原型对象本身是一个普通对象，而普通对象的构造函数都是 `Object`

```

▼
  JavaScript | 复制代码
1 Person.prototype.__proto__ === Object.prototype
  
```

刚刚上面说了，所有的构造器都是函数对象，函数对象都是 `Function` 构造产生的

```
1 Object.__proto__ === Function.prototype
```

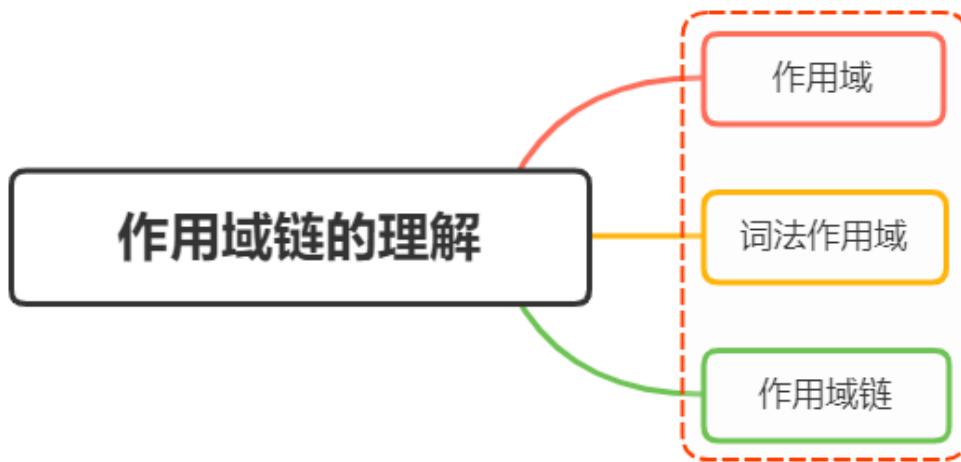
`Object` 的原型对象也有 `__proto__` 属性指向 `null`，`null` 是原型链的顶端

```
1 Object.prototype.__proto__ === null
```

下面作出总结：

- 一切对象都是继承自 `Object` 对象，`Object` 对象直接继承根源对象 `null`
- 一切的函数对象（包括 `Object` 对象），都是继承自 `Function` 对象
- `Object` 对象直接继承自 `Function` 对象
- `Function` 对象的 `__proto__` 会指向自己的原型对象，最终还是继承自 `Object` 对象

## 8. 说说你对作用域链的理解



### 8.1. 作用域

作用域，即变量（变量作用域又称上下文）和函数生效（能被访问）的区域或集合

换句话说，作用域决定了代码区块中变量和其他资源的可见性

举个例子

JavaScript | 复制代码

```
1 function myFunction() {  
2     let inVariable = "函数内部变量";  
3 }  
4 myFunction(); // 要先执行这个函数，否则根本不知道里面是啥  
5 console.log(inVariable); // Uncaught ReferenceError: inVariable is not defined
```

上述例子中，函数 `myFunction` 内部创建一个 `inVariable` 变量，当我们在全局访问这个变量的时候，系统会报错

这就说明我们在全局是无法获取到（闭包除外）函数内部的变量

我们一般将作用域分成：

- 全局作用域
- 函数作用域
- 块级作用域

### 8.1.1. 全局作用域

任何不在函数中或是大括号中声明的变量，都是在全局作用域下，全局作用域下声明的变量可以在程序的任意位置访问

JavaScript | 复制代码

```
1 // 全局变量  
2 var greeting = 'Hello World!';  
3 function greet() {  
4     console.log(greeting);  
5 }  
6 // 打印 'Hello World!'  
7 greet();
```

### 8.1.2. 函数作用域

函数作用域也叫局部作用域，如果一个变量是在函数内部声明的它就在一个函数作用域下面。这些变量只能在函数内部访问，不能在函数以外去访问

JavaScript | 复制代码

```
1 function greet() {  
2     var greeting = 'Hello World!';  
3     console.log(greeting);  
4 }  
5 // 打印 'Hello World!'  
6 greet();  
7 // 报错: Uncaught ReferenceError: greeting is not defined  
8 console.log(greeting);
```

可见上述代码中在函数内部声明的变量或函数，在函数外部是无法访问的，这说明在函数内部定义的变量或者方法只是函数作用域

### 8.1.3. 块级作用域

ES6引入了 `let` 和 `const` 关键字,和 `var` 关键字不同，在大括号中使用 `let` 和 `const` 声明的变量存在于块级作用域中。在大括号之外不能访问这些变量

JavaScript | 复制代码

```
1 {  
2     // 块级作用域中的变量  
3     let greeting = 'Hello World!';  
4     var lang = 'English';  
5     console.log(greeting); // Prints 'Hello World!'  
6 }  
7 // 变量 'English'  
8 console.log(lang);  
9 // 报错: Uncaught ReferenceError: greeting is not defined  
10 console.log(greeting);
```

## 8.2. 词法作用域

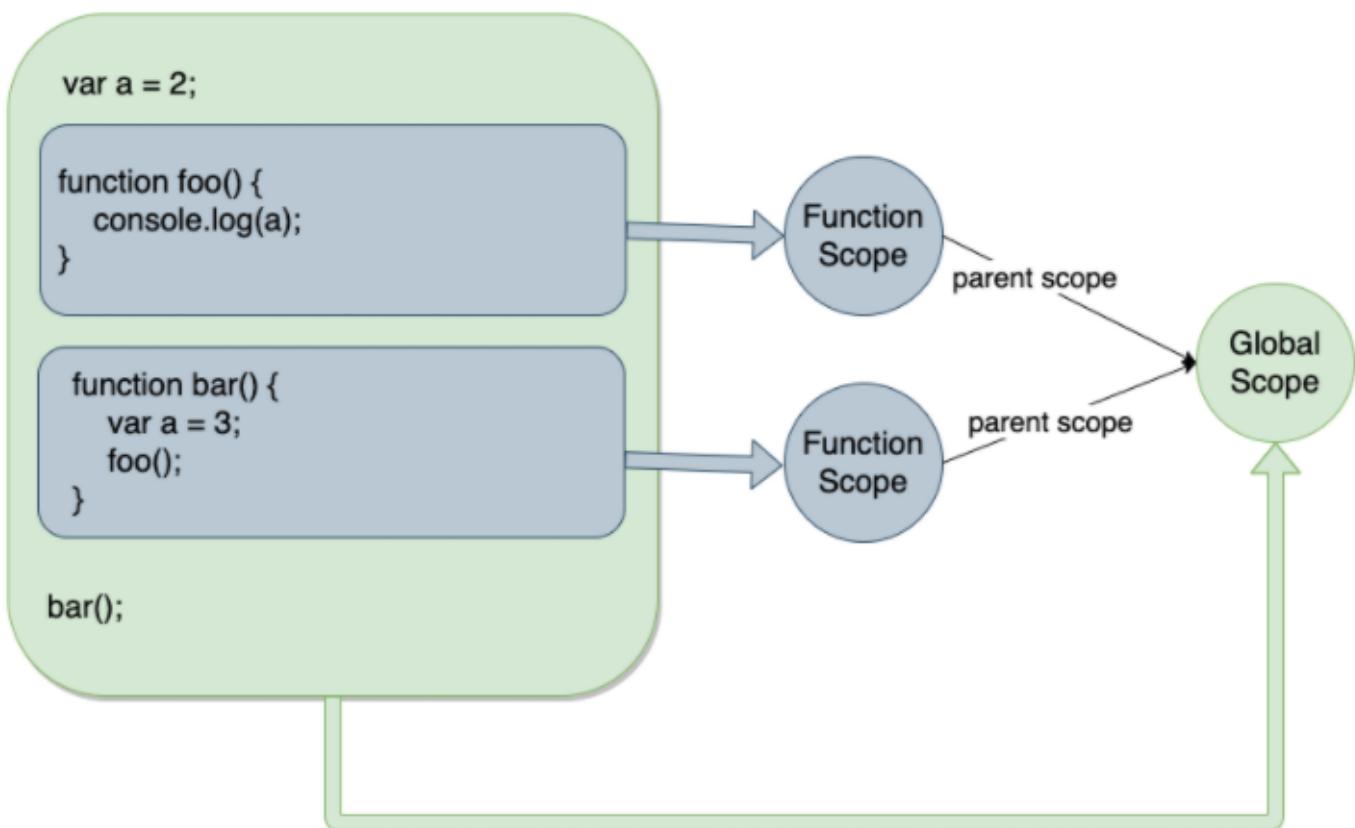
词法作用域，又叫静态作用域，变量被创建时就确定好了，而非执行阶段确定的。也就是说我们写好代码时它的作用域就确定了，`JavaScript` 遵循的就是词法作用域

```

1  var a = 2;
2  function foo(){
3      console.log(a)
4  }
5  function bar(){
6      var a = 3;
7      foo();
8  }
9  bar()

```

上述代码改变成一张图



由于 `JavaScript` 遵循词法作用域，相同层级的 `foo` 和 `bar` 就没有办法访问到彼此块作用域中的变量，所以输出2

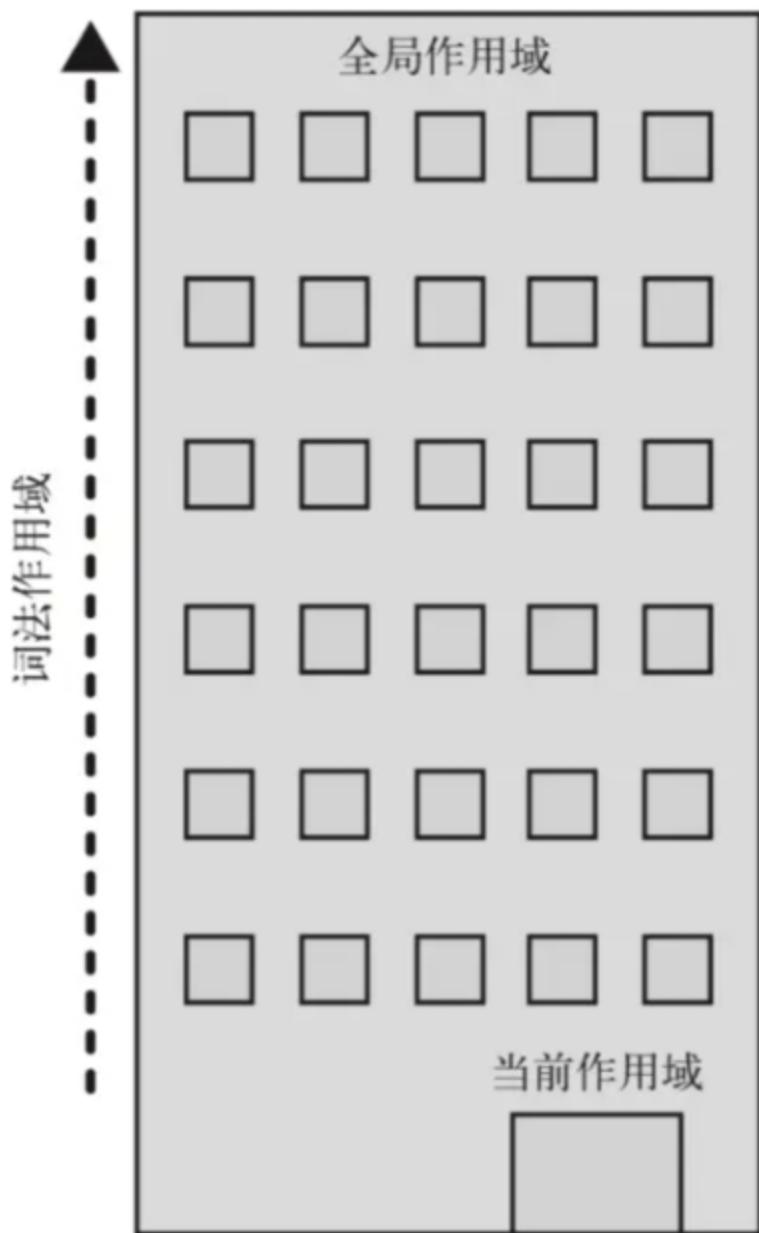
### 8.3. 作用域链

当在 `Javascript` 中使用一个变量的时候，首先 `Javascript` 引擎会尝试在当前作用域下去寻找该变量，如果没找到，再到它的上层作用域寻找，以此类推直到找到该变量或是已经到了全局作用域

如果在全局作用域里仍然找不到该变量，它就会在全局范围内隐式声明该变量(非严格模式下)或是直接报错

这里拿《你不知道的Javascript(上)》中的一张图解释：

把作用域比喻成一个建筑，这份建筑代表程序中的嵌套作用域链，第一层代表当前的执行作用域，顶层代表全局作用域



变量的引用会顺着当前楼层进行查找，如果找不到，则会往上一层找，一旦到达顶层，查找的过程都会停止

下面代码演示下：

```

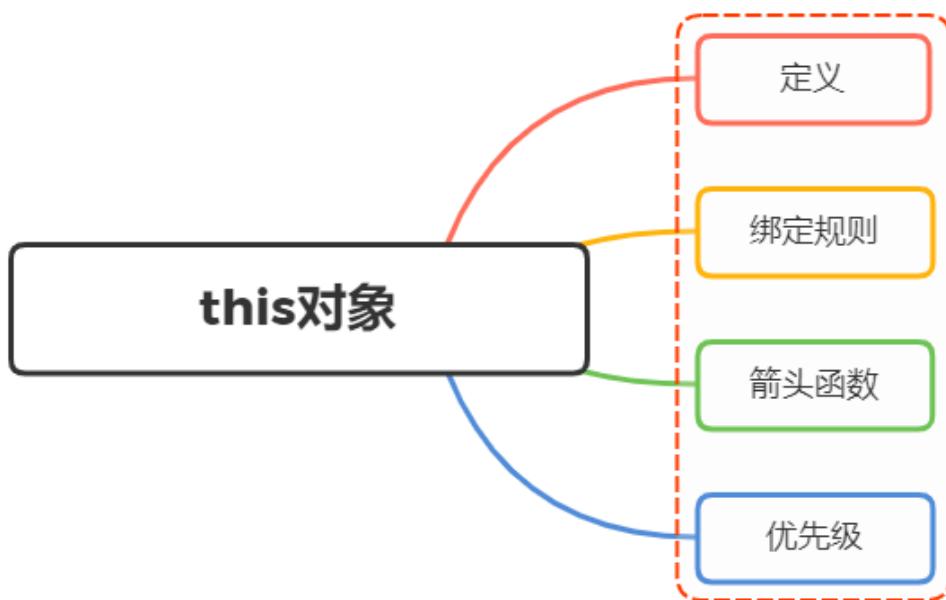
1  var sex = '男';
2  function person() {
3      var name = '张三';
4      function student() {
5          var age = 18;
6          console.log(name); // 张三
7          console.log(sex); // 男
8      }
9      student();
10     console.log(age); // Uncaught ReferenceError: age is not defined
11 }
12 person();

```

上述代码主要做了以下工作：

- `student` 函数内部属于最内层作用域，找不到 `name`，向上一层作用域 `person` 函数内部找，找到了输出“张三”
- `student` 内部输出 `sex` 时找不到，向上一层作用域 `person` 函数找，还找不到继续向上一层找，即全局作用域，找到了输出“男”
- 在 `person` 函数内部输出 `age` 时找不到，向上一层作用域找，即全局作用域，还是找不到则报错

## 9. 谈谈this对象的理解



## 9.1. 定义

函数的 `this` 关键字在 `JavaScript` 中的表现略有不同，此外，在严格模式和非严格模式之间也会有一些差别

在绝大多数情况下，函数的调用方式决定了 `this` 的值（运行时绑定）

`this` 关键字是函数运行时自动生成的一个内部对象，只能在函数内部使用，总指向调用它的对象

举个例子：

```
▼ JavaScript | 复制代码
1  function baz() {
2      // 当前调用栈是: baz
3      // 因此，当前调用位置是全局作用域
4
5      console.log( "baz" );
6      bar(); // <-- bar的调用位置
7  }
8
9  function bar() {
10     // 当前调用栈是: baz --> bar
11     // 因此，当前调用位置在baz中
12
13     console.log( "bar" );
14     foo(); // <-- foo的调用位置
15  }
16
17 function foo() {
18     // 当前调用栈是: baz --> bar --> foo
19     // 因此，当前调用位置在bar中
20
21     console.log( "foo" );
22  }
23
24 baz(); // <-- baz的调用位置
```

同时，`this` 在函数执行过程中，`this` 一旦被确定了，就不可以再更改

JavaScript | 复制代码

```
1 var a = 10;
2 var obj = {
3   a: 20
4 }
5
6 function fn() {
7   this = obj; // 修改this, 运行后会报错
8   console.log(this.a);
9 }
10
11 fn();
```

## 9.2. 绑定规则

根据不同的使用场合，`this` 有不同的值，主要分为下面几种情况：

- 默认绑定
- 隐式绑定
- new绑定
- 显示绑定

### 9.2.1. 默认绑定

全局环境中定义 `person` 函数，内部使用 `this` 关键字

JavaScript | 复制代码

```
1 var name = 'Jenny';
2 function person() {
3   return this.name;
4 }
5 console.log(person()); //Jenny
```

上述代码输出 `Jenny`，原因是调用函数的对象在浏览器中位 `window`，因此 `this` 指向 `window`，所以输出 `Jenny`

注意：

严格模式下，不能将全局对象用于默认绑定，`this`会绑定到 `undefined`，只有函数运行在非严格模式下，默认绑定才能绑定到全局对象

## 9.2.2. 隐式绑定

函数还可以作为某个对象的方法调用，这时 `this` 就指这个上级对象

```
1 function test() {  
2     console.log(this.x);  
3 }  
4  
5 var obj = {};  
6 obj.x = 1;  
7 obj.m = test;  
8  
9 obj.m(); // 1
```

这个函数中包含多个对象，尽管这个函数是被最外层的对象所调用，`this` 指向的也只是它上一级的对象

```
1 var o = {  
2     a:10,  
3     b:{  
4         fn:function(){  
5             console.log(this.a); //undefined  
6         }  
7     }  
8 }  
9 o.b.fn();
```

上述代码中，`this` 的上一级对象为 `b`，`b` 内部并没有 `a` 变量的定义，所以输出 `undefined`

这里再举一种特殊情况

```

1 var o = {
2     a:10,
3     b:{
4         a:12,
5         fn:function(){
6             console.log(this.a); //undefined
7             console.log(this); //window
8         }
9     }
10 }
11 var j = o.b.fn;
12 j();

```

此时 `this` 指向的是 `window`，这里的大家需要记住，`this` 永远指向的是最后调用它的对象，虽然 `fn` 是对象 `b` 的方法，但是 `fn` 赋值给 `j` 时候并没有执行，所以最终指向 `window`

### 9.2.3. new绑定

通过构建函数 `new` 关键字生成一个实例对象，此时 `this` 指向这个实例对象

```

1 function test() {
2     this.x = 1;
3 }
4
5 var obj = new test();
6 obj.x // 1

```

上述代码之所以能输出1，是因为 `new` 关键字改变了 `this` 的指向

这里再列举一些特殊情况：

`new` 过程遇到 `return` 一个对象，此时 `this` 指向为返回的对象

JavaScript | 复制代码

```
1 function fn()
2 {
3     this.user = 'xxx';
4     return {};
5 }
6 var a = new fn();
7 console.log(a.user); //undefined
```

如果返回一个简单类型的时候，则 `this` 指向实例对象

JavaScript | 复制代码

```
1 function fn()
2 {
3     this.user = 'xxx';
4     return 1;
5 }
6 var a = new fn();
7 console.log(a.user); //xxx
```

注意的是 `null` 虽然也是对象，但是此时 `new` 仍然指向实例对象

JavaScript | 复制代码

```
1 function fn()
2 {
3     this.user = 'xxx';
4     return null;
5 }
6 var a = new fn();
7 console.log(a.user); //xxx
```

## 9.2.4. 显示修改

`apply()`、`call()`、`bind()` 是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时 `this` 指的就是这第一个参数

JavaScript | 复制代码

```
1 var x = 0;
2 function test() {
3     console.log(this.x);
4 }
5
6 var obj = {};
7 obj.x = 1;
8 obj.m = test;
9 obj.m.apply(obj) // 1
```

关于 `apply`、`call`、`bind` 三者的区别，我们后面再详细说

### 9.3. 箭头函数

在 ES6 的语法中还提供了箭头函数语法，让我们在代码书写时就能确定 `this` 的指向（编译时绑定）

举个例子：

JavaScript | 复制代码

```
1 const obj = {
2     sayThis: () => {
3         console.log(this);
4     }
5 };
6
7 obj.sayThis(); // window 因为 JavaScript 没有块作用域，所以在定义 sayThis 的时候，里面的 this 就绑到 window 上去了
8 const globalSay = obj.sayThis;
9 globalSay(); // window 浏览器中的 global 对象
```

虽然箭头函数的 `this` 能够在编译的时候就确定了 `this` 的指向，但也需要注意一些潜在的坑

下面举个例子：

绑定事件监听

JavaScript | 复制代码

```
1 const button = document.getElementById('mngb');
2 button.addEventListener('click', ()=> {
3     console.log(this === window) // true
4     this.innerHTML = 'clicked button'
5 })
```

上述可以看到，我们其实是想要 `this` 为点击的 `button`，但此时 `this` 指向了 `window`

包括在原型上添加方法时候，此时 `this` 指向 `window`

JavaScript | 复制代码

```
1 Cat.prototype.sayName = () => {
2     console.log(this === window) //true
3     return this.name
4 }
5 const cat = new Cat('mm');
6 cat.sayName()
```

同样的，箭头函数不能作为构建函数

## 9.4. 优先级

### 9.4.1. 隐式绑定 VS 显式绑定

```
1 function foo() {
2     console.log( this.a );
3 }
4
5 var obj1 = {
6     a: 2,
7     foo: foo
8 };
9
10 var obj2 = {
11     a: 3,
12     foo: foo
13 };
14
15 obj1.foo(); // 2
16 obj2.foo(); // 3
17
18 obj1.foo.call( obj2 ); // 3
19 obj2.foo.call( obj1 ); // 2
```

显然，显示绑定的优先级更高

#### 9.4.2. new绑定 VS 隐式绑定

```

1  function foo(something) {
2      this.a = something;
3  }
4
5  var obj1 = {
6      foo: foo
7  };
8
9  var obj2 = {};
10
11 obj1.foo( 2 );
12 console.log( obj1.a ); // 2
13
14 obj1.foo.call( obj2, 3 );
15 console.log( obj2.a ); // 3
16
17 var bar = new obj1.foo( 4 );
18 console.log( obj1.a ); // 2
19 console.log( bar.a ); // 4

```

可以看到，new绑定的优先级 > 隐式绑定

### 9.4.3. new 绑定 VS 显式绑定

因为 new 和 apply、call 无法一起使用，但硬绑定也是显式绑定的一种，可以替换测试

```

1  function foo(something) {
2      this.a = something;
3  }
4
5  var obj1 = {};
6
7  var bar = foo.bind( obj1 );
8  bar( 2 );
9  console.log( obj1.a ); // 2
10
11 var baz = new bar( 3 );
12 console.log( obj1.a ); // 2
13 console.log( baz.a ); // 3

```

`bar` 被绑定到 `obj1` 上，但是 `new bar(3)` 并没有像我们预计的那样把 `obj1.a` 修改为 3。但是，`new` 修改了绑定调用 `bar()` 中的 `this`

我们可认为 `new` 绑定优先级 > 显式绑定

综上，`new` 绑定优先级 > 显示绑定优先级 > 隐式绑定优先级 > 默认绑定优先级

## 10. 说说`new` 操作符具体干了什么？



### 10.1. 是什么

在 `JavaScript` 中，`new` 操作符用于创建一个给定构造函数的实例对象

例子

```
1 function Person(name, age){  
2     this.name = name;  
3     this.age = age;  
4 }  
5 Person.prototype.sayName = function () {  
6     console.log(this.name)  
7 }  
8 const person1 = new Person('Tom', 20)  
9 console.log(person1) // Person {name: "Tom", age: 20}  
10 t.sayName() // 'Tom'
```

从上面可以看到：

- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数中的属性

- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数原型链中的属性（即实例与构造函数通过原型链连接了起来）

现在在构建函数中显式加上返回值，并且这个返回值是一个原始类型

```
1 function Test(name) {  
2     this.name = name  
3     return 1  
4 }  
5 const t = new Test('xxx')  
6 console.log(t.name) // 'xxx'
```

JavaScript | 复制代码

可以发现，构造函数中返回一个原始值，然而这个返回值并没有作用

下面在构造函数中返回一个对象

```
1 function Test(name) {  
2     this.name = name  
3     console.log(this) // Test { name: 'xxx' }  
4     return { age: 26 }  
5 }  
6 const t = new Test('xxx')  
7 console.log(t) // { age: 26 }  
8 console.log(t.name) // 'undefined'
```

JavaScript | 复制代码

从上面可以发现，构造函数如果返回值为一个对象，那么这个返回值会被正常使用

## 10.2. 流程

从上面介绍中，我们可以看到 `new` 关键字主要做了以下的工作：

- 创建一个新的对象 `obj`
- 将对象与构建函数通过原型链连接起来
- 将构建函数中的 `this` 绑定到新建的对象 `obj` 上
- 根据构建函数返回类型作判断，如果是原始值则被忽略，如果是返回对象，需要正常处理

举个例子：

```

1 function Person(name, age){
2     this.name = name;
3     this.age = age;
4 }
5 const person1 = new Person('Tom', 20)
6 console.log(person1) // Person {name: "Tom", age: 20}
7 t.sayName() // 'Tom'

```

流程图如下：

**const person1 = new Person('Tom', 20)**



```

const person1 = {
    __proto__ = Person.prototype;
    name = 'Tom';
    age = 20;
}

```

<https://chen-cong.blog.csdn.net>

## 10.3. 手写new操作符

现在我们已经清楚地掌握了 new 的执行过程

那么我们就动手来实现一下 new

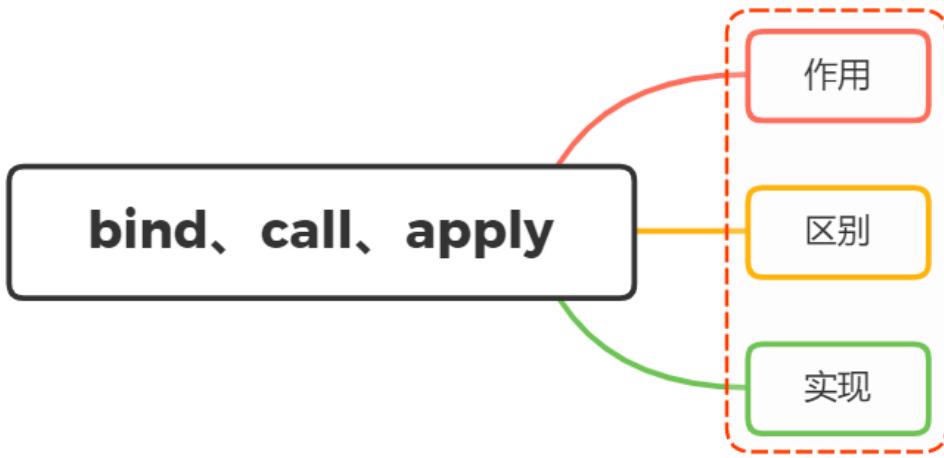
```
1 function mynew(Func, ...args) {  
2     // 1. 创建一个新对象  
3     const obj = {}  
4     // 2. 新对象原型指向构造函数原型对象  
5     obj.__proto__ = Func.prototype  
6     // 3. 将构建函数的this指向新对象  
7     let result = Func.apply(obj, args)  
8     // 4. 根据返回值判断  
9     return result instanceof Object ? result : obj  
10 }
```

测试一下

```
1 function mynew(func, ...args) {  
2     const obj = {}  
3     obj.__proto__ = func.prototype  
4     let result = func.apply(obj, args)  
5     return result instanceof Object ? result : obj  
6 }  
7 function Person(name, age) {  
8     this.name = name;  
9     this.age = age;  
10 }  
11 Person.prototype.say = function () {  
12     console.log(this.name)  
13 }  
14  
15 let p = mynew(Person, "huihui", 123)  
16 console.log(p) // Person {name: "huihui", age: 123}  
17 p.say() // huihui
```

可以发现，代码虽然很短，但是能够模拟实现 new

## 11. bind、call、apply 区别？如何实现一个bind？



## 11.1. 作用

`call`、`apply`、`bind` 作用是改变函数执行时的上下文，简而言之就是改变函数运行时的 `this` 指向

那么什么情况下需要改变 `this` 的指向呢？下面举个例子

```

1 var name = "lucy";
2 var obj = {
3   name: "martin",
4   say: function () {
5     console.log(this.name);
6   }
7 };
8 obj.say(); // martin, this 指向 obj 对象
9 setTimeout(obj.say,0); // lucy, this 指向 window 对象
  
```

从上面可以看到，正常情况 `say` 方法输出 `martin`

但是我们把 `say` 放在 `setTimeout` 方法中，在定时器中是作为回调函数来执行的，因此回到主栈执行时是在全局执行上下文的环境中执行的，这时候 `this` 指向 `window`，所以输出 `lucy`

我们实际需要的是 `this` 指向 `obj` 对象，这时候就需要该改变 `this` 指向了

```

1 setTimeout(obj.say.bind(obj),0); //martin, this指向obj对象
  
```

## 11.2. 区别

下面再来看看 `apply`、`call`、`bind` 的使用

### 11.2.1. `apply`

`apply` 接受两个参数，第一个参数是 `this` 的指向，第二个参数是函数接受的参数，以数组的形式传入

改变 `this` 指向后原函数会立即执行，且此方法只是临时改变 `this` 指向一次

```
▼ JavaScript | ⌂ 复制代码

1 function fn(...args){
2     console.log(this,args);
3 }
4 let obj = {
5     myname:"张三"
6 }
7
8 fn.apply(obj,[1,2]); // this会变成传入的obj，传入的参数必须是一个数组;
9 fn(1,2) // this指向window
```

当第一个参数为 `null`、`undefined` 的时候，默认指向 `window` (在浏览器中)

```
▼ JavaScript | ⌂ 复制代码

1 fn.apply(null,[1,2]); // this指向window
2 fn.apply(undefined,[1,2]); // this指向window
```

### 11.2.2. `call`

`call` 方法的第一个参数也是 `this` 的指向，后面传入的是一个参数列表

跟 `apply` 一样，改变 `this` 指向后原函数会立即执行，且此方法只是临时改变 `this` 指向一次

JavaScript | 复制代码

```
1 function fn(...args){  
2     console.log(this,args);  
3 }  
4 let obj = {  
5     myname:"张三"  
6 }  
7  
8 fn.call(obj,1,2); // this会变成传入的obj, 传入的参数必须是一个数组;  
9 fn(1,2) // this指向window
```

同样的，当第一个参数为 `null`、`undefined` 的时候，默认指向 `window` (在浏览器中)

JavaScript | 复制代码

```
1 fn.call(null,[1,2]); // this指向window  
2 fn.call(undefined,[1,2]); // this指向window
```

### 11.2.3. bind

bind方法和call很相似，第一参数也是 `this` 的指向，后面传入的也是一个参数列表(但是这个参数列表可以分多次传入)

改变 `this` 指向后不会立即执行，而是返回一个永久改变 `this` 指向的函数

JavaScript | 复制代码

```
1 function fn(...args){  
2     console.log(this,args);  
3 }  
4 let obj = {  
5     myname:"张三"  
6 }  
7  
8 const bindFn = fn.bind(obj); // this 也会变成传入的obj , bind不是立即执行需要执行一次  
9 bindFn(1,2) // this指向obj  
10 fn(1,2) // this指向window
```

### 11.2.4. 小结

从上面可以看到，`apply`、`call`、`bind` 三者的区别在于：

- 三者都可以改变函数的 `this` 对象指向
- 三者第一个参数都是 `this` 要指向的对象，如果没有这个参数或参数为 `undefined` 或 `null`，则默认指向全局 `window`
- 三者都可以传参，但是 `apply` 是数组，而 `call` 是参数列表，且 `apply` 和 `call` 是一次性传入参数，而 `bind` 可以分为多次传入
- `bind` 是返回绑定 `this` 之后的函数，`apply`、`call` 则是立即执行

## 11.3. 实现

实现 `bind` 的步骤，我们可以分解成为三部分：

- 修改 `this` 指向
- 动态传递参数

```
▼ JavaScript | 复制代码

1 // 方式一：只在bind中传递函数参数
2 fn.bind(obj,1,2)()
3
4 // 方式二：在bind中传递函数参数，也在返回函数中传递参数
5 fn.bind(obj,1)(2)
```

- 兼容 `new` 关键字

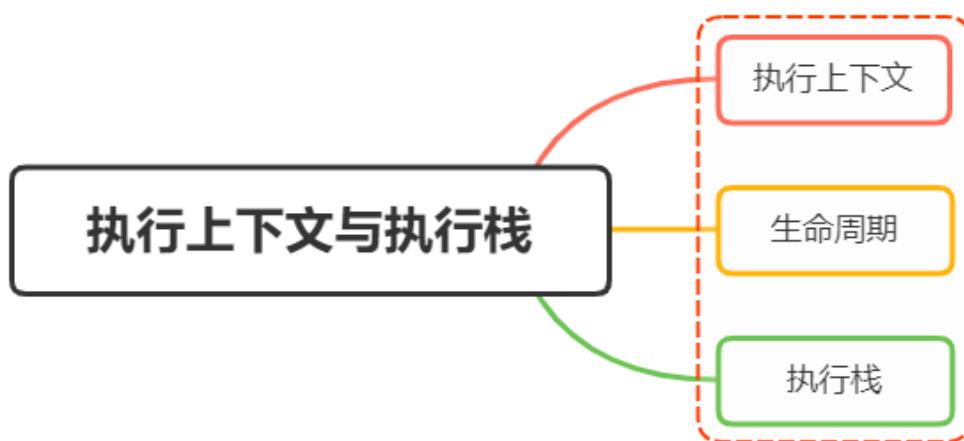
整体实现代码如下：

```

1 Function.prototype.myBind = function (context) {
2     // 判断调用对象是否为函数
3     if (typeof this !== "function") {
4         throw new TypeError("Error");
5     }
6
7     // 获取参数
8     const args = [...arguments].slice(1),
9         fn = this;
10
11    return function Fn() {
12
13        // 根据调用方式, 传入不同绑定值
14        return fn.apply(this instanceof Fn ? new fn(...args) : context,
15                      args.concat(...arguments));
16    }

```

## 12. JavaScript中执行上下文和执行栈是什么？



### 12.1. 执行上下文

简单的来说，执行上下文是一种对 `Javascript` 代码执行环境的抽象概念，也就是说只要有 `Javascript` 代码运行，那么它就一定是运行在执行上下文中。

执行上下文的类型分为三种：

- 全局执行上下文：只有一个，浏览器中的全局对象就是 `window` 对象，`this` 指向这个全局对象

- 函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文
- Eval 函数执行上下文：指的是运行在 eval 函数中的代码，很少用而且不建议使用

下面给出全局上下文和函数上下文的例子：

```
// global context

var sayHello = 'Hello';

function person() {           // execution context

    var first = 'David',
        last = 'Shariff';

    function firstName() { // execution context
        return first;
    }

    function lastName() { // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());
}

}
```

紫色框住的部分为全局上下文，蓝色和橘色框起来的是不同的函数上下文。只有全局上下文（的变量）能被其他任何上下文访问

可以有任意多个函数上下文，每次调用函数创建一个新的上下文，会创建一个私有作用域，函数内部声明的任何变量都不能在当前函数作用域外部直接访问

## 12.2. 生命周期

执行上下文的生命周期包括三个阶段：创建阶段 → 执行阶段 → 回收阶段

### 12.2.1. 创建阶段

创建阶段即当函数被调用，但未执行任何其内部代码之前

创建阶段做了三件事：

- 确定 `this` 的值，也被称为 `This Binding`
- `LexicalEnvironment`（词法环境）组件被创建
- `VariableEnvironment`（变量环境）组件被创建

伪代码如下：

```
▼ JavaScript | 复制代码

1 ExecutionContext = {
2   ThisBinding = <this value>,      // 确定this
3   LexicalEnvironment = { ... },    // 词法环境
4   VariableEnvironment = { ... },  // 变量环境
5 }
```

### 12.2.1.1. This Binding

确定 `this` 的值我们前面讲到，`this` 的值是在执行的时候才能确认，定义的时候不能确认

### 12.2.1.2. 词法环境

词法环境有两个组成部分：

- 全局环境：是一个没有外部环境的词法环境，其外部环境引用为 `null`，有一个全局对象，`this` 的值指向这个全局对象
- 函数环境：用户在函数中定义的变量被存储在环境记录中，包含了 `arguments` 对象，外部环境的引用可以是全局环境，也可以是包含内部函数的外部函数环境

伪代码如下：

JavaScript | 复制代码

```
1 GlobalExectionContext = { // 全局执行上下文
2   LexicalEnvironment: { // 词法环境
3     EnvironmentRecord: { // 环境记录
4       Type: "Object", // 全局环境
5       // 标识符绑定在这里
6       outer: <null> // 对外部环境的引用
7     }
8   }
9
10 FunctionExectionContext = { // 函数执行上下文
11   LexicalEnvironment: { // 词法环境
12     EnvironmentRecord: { // 环境记录
13       Type: "Declarative", // 函数环境
14       // 标识符绑定在这里 // 对外部环境的引用
15       outer: <Global or outer function environment reference>
16     }
17 }
```

### 12.2.1.3. 变量环境

变量环境也是一个词法环境，因此它具有上面定义的词法环境的所有属性

在 ES6 中，词法环境和变量环境的区别在于前者用于存储函数声明和变量（`let` 和 `const`）绑定，而后者仅用于存储变量（`var`）绑定

举个例子

JavaScript | 复制代码

```
1 let a = 20;
2 const b = 30;
3 var c;
4
5 function multiply(e, f) {
6   var g = 20;
7   return e * f * g;
8 }
9
10 c = multiply(20, 30);
```

执行上下文如下：

```
1 GlobalExectionContext = {
2
3   ThisBinding: <Global Object>,
4
5   LexicalEnvironment: { // 词法环境
6     EnvironmentRecord: {
7       Type: "Object",
8       // 标识符绑定在这里
9       a: < uninitialized >,
10      b: < uninitialized >,
11      multiply: < func >
12    }
13    outer: <null>
14  },
15
16  VariableEnvironment: { // 变量环境
17    EnvironmentRecord: {
18      Type: "Object",
19      // 标识符绑定在这里
20      c: undefined,
21    }
22    outer: <null>
23  }
24}
25
26 FunctionExectionContext = {
27
28   ThisBinding: <Global Object>,
29
30   LexicalEnvironment: {
31     EnvironmentRecord: {
32       Type: "Declarative",
33       // 标识符绑定在这里
34       Arguments: {0: 20, 1: 30, length: 2},
35     },
36     outer: <GlobalLexicalEnvironment>
37   },
38
39   VariableEnvironment: {
40     EnvironmentRecord: {
41       Type: "Declarative",
42       // 标识符绑定在这里
43       g: undefined
44     },
45     outer: <GlobalLexicalEnvironment>
```

```
46      }  
47 }
```

留意上面的代码，`let` 和 `const` 定义的变量 `a` 和 `b` 在创建阶段没有被赋值，但 `var` 声明的变量从在创建阶段被赋值为 `undefined`

这是因为，创建阶段，会在代码中扫描变量和函数声明，然后将函数声明存储在环境中

但变量会被初始化为 `undefined` (`var` 声明的情况下) 和保持 `uninitialized` (未初始化状态)(使用 `let` 和 `const` 声明的情况下)

这就是变量提升的实际原因

## 12.2.2. 执行阶段

在这阶段，执行变量赋值、代码执行

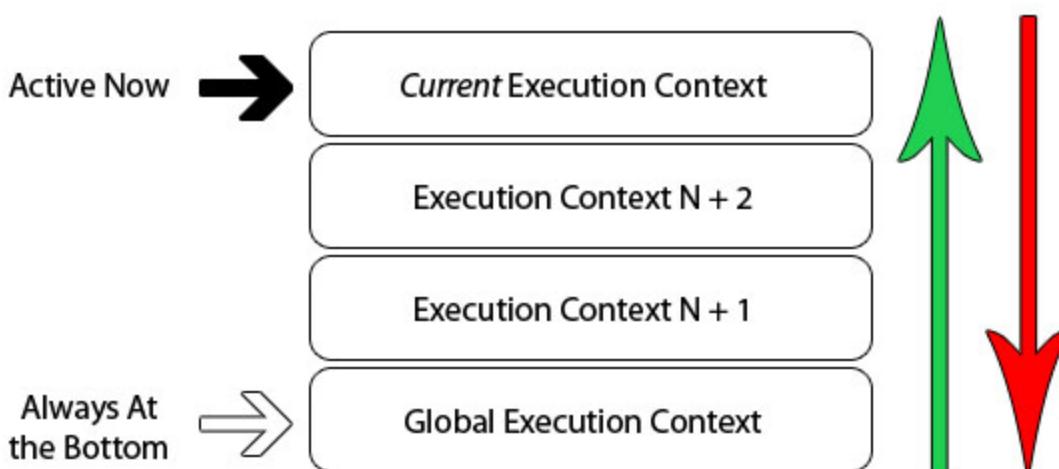
如果 `Javascript` 引擎在源代码中声明的实际位置找不到变量的值，那么将为其分配 `undefined` 值

## 12.2.3. 回收阶段

执行上下文出栈等待虚拟机回收执行上下文

## 12.3. 执行栈

执行栈，也叫调用栈，具有 LIFO (后进先出) 结构，用于存储在代码执行期间创建的所有执行上下文



当 `Javascript` 引擎开始执行你第一行脚本代码的时候，它就会创建一个全局执行上下文然后将它压到执行栈中

每当引擎碰到一个函数的时候，它就会创建一个函数执行上下文，然后将这个执行上下文压到执行栈中。引擎会执行位于执行栈栈顶的执行上下文(一般是函数执行上下文)，当该函数执行结束后，对应的执行上下文就会被弹出，然后控制流程到达执行栈的下一个执行上下文。

举个例子：

```
JavaScript | 复制代码
```

```
1 let a = 'Hello World!';
2 function first() {
3     console.log('Inside first function');
4     second();
5     console.log('Again inside first function');
6 }
7 function second() {
8     console.log('Inside second function');
9 }
10 first();
11 console.log('Inside Global Execution Context');
```

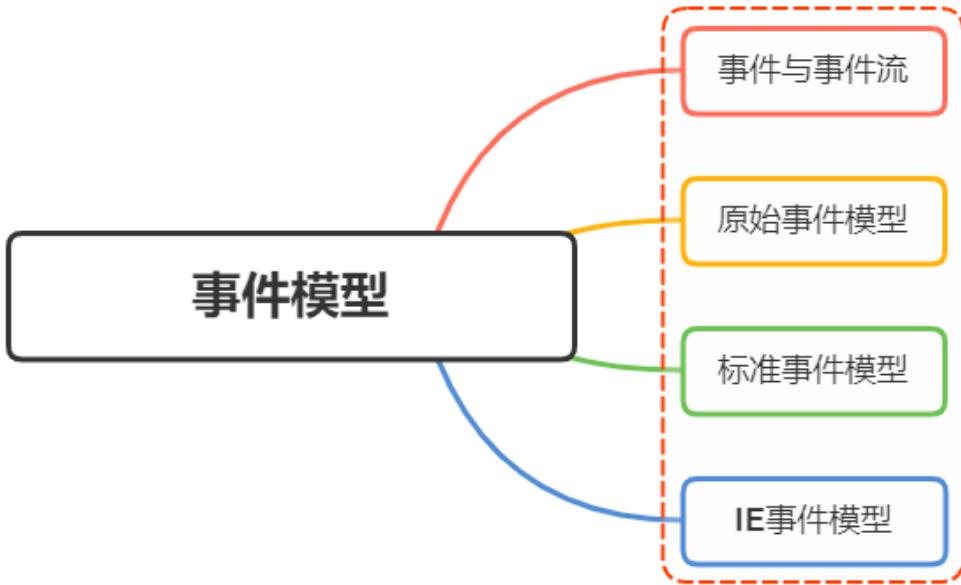
转化成图的形式



简单分析一下流程：

- 创建全局上下文请压入执行栈
- `first` 函数被调用，创建函数执行上下文并压入栈
- 执行 `first` 函数过程遇到 `second` 函数，再创建一个函数执行上下文并压入栈
- `second` 函数执行完毕，对应的函数执行上下文被推出执行栈，执行下一个执行上下文 `first` 函数
- `first` 函数执行完毕，对应的函数执行上下文也被推出栈中，然后执行全局上下文
- 所有代码执行完毕，全局上下文也会被推出栈中，程序结束

## 13. 说说JavaScript中的事件模型



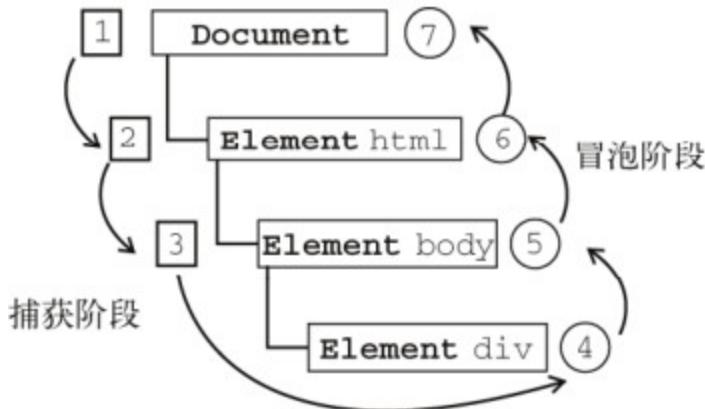
## 13.1. 事件与事件流

`javascript` 中的事件，可以理解就是在 `HTML` 文档或者浏览器中发生的一种交互操作，使得网页具备互动性，常见的有加载事件、鼠标事件、自定义事件等

由于 `DOM` 是一个树结构，如果在父子节点绑定事件时候，当触发子节点的时候，就存在一个顺序问题，这就涉及到了事件流的概念

事件流都会经历三个阶段：

- 事件捕获阶段(capture phase)
- 处于目标阶段(target phase)
- 事件冒泡阶段(bubbling phase)



事件冒泡是一种从下往上的传播方式，由最具体的元素（触发节点）然后逐渐向上传播到最不具体的那个节点，也就是 `DOM` 中最高层的父节点

HTML | 复制代码

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Event Bubbling</title>
6  </head>
7  <body>
8      <button id="clickMe">Click Me</button>
9  </body>
10 </html>
```

然后，我们给 `button` 和它的父元素，加入点击事件

JavaScript | 复制代码

```
1  var button = document.getElementById('clickMe');
2
3  button.onclick = function() {
4      console.log('1.Button');
5  };
6  document.body.onclick = function() {
7      console.log('2.body');
8  };
9  document.onclick = function() {
10     console.log('3.document');
11 };
12 window.onclick = function() {
13     console.log('4.window');
14 };
```

点击按钮，输出如下

JavaScript | 复制代码

```
1  1.button
2  2.body
3  3.document
4  4.window
```

点击事件首先在 `button` 元素上发生，然后逐级向上传播

事件捕获与事件冒泡相反，事件最开始由不太具体的节点最早接受事件，而最具体的节点（触发节点）最后接受事件

## 13.2. 事件模型

事件模型可以分为三种：

- 原始事件模型（DOM0级）
- 标准事件模型（DOM2级）
- IE事件模型（基本不用）

### 13.2.1. 原始事件模型

事件绑定监听函数比较简单，有两种方式：

- HTML代码中直接绑定

```
▼ JavaScript | 复制代码
1 <input type="button" onclick="fun()">
```

- 通过 JS 代码绑定

```
▼ JavaScript | 复制代码
1 var btn = document.getElementById('.btn');
2 btn.onclick = fun;
```

#### 13.2.1.1. 特性

- 绑定速度快

DOM0 级事件具有很好的跨浏览器优势，会以最快的速度绑定，但由于绑定速度太快，可能页面还未完全加载出来，以至于事件可能无法正常运行

- 只支持冒泡，不支持捕获
- 同一个类型的事件只能绑定一次

```
▼ JavaScript | 复制代码
1 <input type="button" id="btn" onclick="fun1()">
2
3 var btn = document.getElementById('.btn');
4 btn.onclick = fun2;
```

如上，当希望为同一个元素绑定多个同类型事件的时候（上面的这个 `btn` 元素绑定2个点击事件），是不被允许的，后绑定的事件会覆盖之前的事件

删除 `DOM0` 级事件处理程序只要将对应事件属性置为 `null` 即可

▼

JavaScript

复制代码

```
1 btn.onclick = null;
```

### 13.2.2. 标准事件模型

在该事件模型中，一次事件共有三个过程：

- 事件捕获阶段：事件从 `document` 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行
- 事件处理阶段：事件到达目标元素，触发目标元素的监听函数
- 事件冒泡阶段：事件从目标元素冒泡到 `document`，依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下：

▼

Plain Text

复制代码

```
1 addEventListener(eventType, handler, useCapture)
```

事件移除监听函数的方式如下：

▼

Plain Text

复制代码

```
1 removeEventListener(eventType, handler, useCapture)
```

参数如下：

- `eventType` 指定事件类型(不要加on)
- `handler` 是事件处理函数
- `useCapture` 是一个 `boolean` 用于指定是否在捕获阶段进行处理，一般设置为 `false` 与IE浏览器保持一致

举个例子：

JavaScript | 复制代码

```
1 var btn = document.getElementById('.btn');
2 btn.addEventListener('click', showMessage, false);
3 btn.removeEventListener('click', showMessage, false);
```

### 13.2.2.1. 特性

- 可以在一个 DOM 元素上绑定多个事件处理器，各自并不会冲突

JavaScript | 复制代码

```
1 btn.addEventListener('click', showMessage1, false);
2 btn.addEventListener('click', showMessage2, false);
3 btn.addEventListener('click', showMessage3, false);
```

- 执行时机

当第三个参数( useCapture )设置为 true 就在捕获过程中执行，反之在冒泡过程中执行处理函数

下面举个例子：

JavaScript | 复制代码

```
1 <div id='div'>
2   <p id='p'>
3     <span id='span'>Click Me!</span>
4   </p >
5 </div>
```

设置点击事件

JavaScript | 复制代码

```
1 var div = document.getElementById('div');
2 var p = document.getElementById('p');
3
4 function onClickFn (event) {
5   var tagName = event.currentTarget.tagName;
6   var phase = event.eventPhase;
7   console.log(tagName, phase);
8 }
9
10 div.addEventListener('click', onClickFn, false);
11 p.addEventListener('click', onClickFn, false);
```

上述使用了 `eventPhase`，返回一个代表当前执行阶段的整数值。1为捕获阶段、2为事件对象触发阶段、3为冒泡阶段

点击 `Click Me!`，输出如下

```
1 P 3  
2 DIV 3
```

JavaScript | 复制代码

可以看到，`p` 和 `div` 都是在冒泡阶段响应了事件，由于冒泡的特性，裹在里层的 `p` 率先做出响应  
如果把第三个参数都改为 `true`

```
1 div.addEventListener('click', onClickFn, true);  
2 p.addEventListener('click', onClickFn, true);
```

JavaScript | 复制代码

输出如下

```
1 DIV 1  
2 P 1
```

JavaScript | 复制代码

两者都是在捕获阶段响应事件，所以 `div` 比 `p` 标签先做出响应

### 13.2.3. IE事件模型

IE事件模型共有两个过程：

- 事件处理阶段：事件到达目标元素，触发目标元素的监听函数。
- 事件冒泡阶段：事件从目标元素冒泡到 `document`，依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下：

```
1 attachEvent(eventType, handler)
```

Plain Text | 复制代码

事件移除监听函数的方式如下：

Plain Text | 复制代码

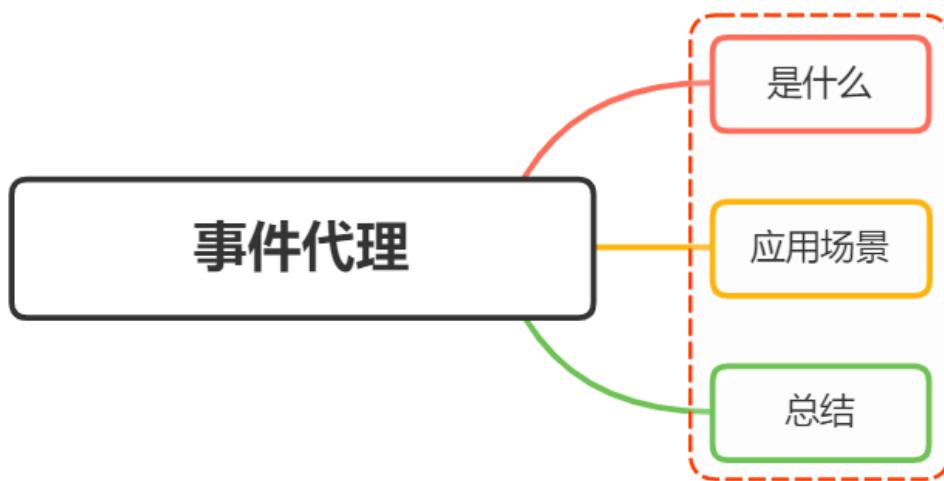
```
1 detachEvent(eventType, handler)
```

举个例子：

JavaScript | 复制代码

```
1 var btn = document.getElementById('.btn');
2 btn.attachEvent('onclick', showMessage);
3 btn.detachEvent('onclick', showMessage);
```

## 14. 解释下什么是事件代理？应用场景？



### 14.1. 是什么

事件代理，俗地来讲，就是把一个元素响应事件（`click`、`keydown` .....）的函数委托到另一个元素

前面讲到，事件流的都会经过三个阶段： 捕获阶段 -> 目标阶段 -> 冒泡阶段，而事件委托就是在冒泡阶段完成

事件委托，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，而不是目标元素

当事件响应到目标元素上时，会通过事件冒泡机制从而触发它的外层元素的绑定事件上，然后在外层元素上去执行函数

下面举个例子：

比如一个宿舍的同学同时快递到了，一种笨方法就是他们一个个去领取  
较优方法就是把这件事情委托给宿舍长，让一个人出去拿好所有快递，然后再根据收件人——分发给每个同学

在这里，取快递就是一个事件，每个同学指的是需要响应事件的 DOM 元素，而出去统一领取快递的宿舍长就是代理的元素

所以真正绑定事件的是这个元素，按照收件人分发快递的过程就是在事件执行中，需要判断当前响应的事件应该匹配到被代理元素中的哪一个或者哪几个

## 14.2. 应用场景

如果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件

```
1 <ul id="list">
2   <li>item 1</li>
3   <li>item 2</li>
4   <li>item 3</li>
5   .....
6   <li>item n</li>
7 </ul>
```

JavaScript | 复制代码

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的

```
1 // 获取目标元素
2 const lis = document.getElementsByTagName("li")
3 // 循环遍历绑定事件
4 for (let i = 0; i < lis.length; i++) {
5   lis[i].onclick = function(e){
6     console.log(e.target.innerHTML)
7   }
8 }
```

JavaScript | 复制代码

这时候就可以事件委托，把点击事件绑定在父级元素 ul 上面，然后执行事件的时候再去匹配目标元素

JavaScript | 复制代码

```
1 // 给父层元素绑定事件
2 document.getElementById('list').addEventListener('click', function (e) {
3     // 兼容性处理
4     var event = e || window.event;
5     var target = event.target || event.srcElement;
6     // 判断是否匹配目标元素
7     if (target.nodeName.toLowerCase === 'li') {
8         console.log('the content is: ', target.innerHTML);
9     }
10});
```

还有一种场景是上述列表项并不多，我们给每个列表项都绑定了事件

但是如果用户能够随时动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的

举个例子：

下面 `html` 结构中，点击 `input` 可以动态添加元素

HTML | 复制代码

```
1 <input type="button" name="" id="btn" value="添加" />
2 <ul id="ul1">
3     <li>item 1</li>
4     <li>item 2</li>
5     <li>item 3</li>
6     <li>item 4</li>
7 </ul>
```

使用事件委托

```

1  const oBtn = document.getElementById("btn");
2  const oUl = document.getElementById("ul1");
3  const num = 4;
4
5  //事件委托，添加的子元素也有事件
6  oUl.onclick = function (ev) {
7      ev = ev || window.event;
8      const target = ev.target || ev.srcElement;
9      if (target.nodeName.toLowerCase() == 'li') {
10          console.log('the content is: ', target.innerHTML);
11      }
12
13  };
14
15 //添加新节点
16 oBtn.onclick = function () {
17     num++;
18     const oLi = document.createElement('li');
19     oLi.innerHTML = `item ${num}`;
20     oUl.appendChild(oLi);
21 };

```

可以看到，使用事件委托，在动态绑定事件的情况下是可以减少很多重复工作的

### 14.3. 总结

适合事件委托的事件有： click , mousedown , mouseup , keydown , keyup , keypress

从上面应用场景中，我们就可以看到使用事件委托存在两大优点：

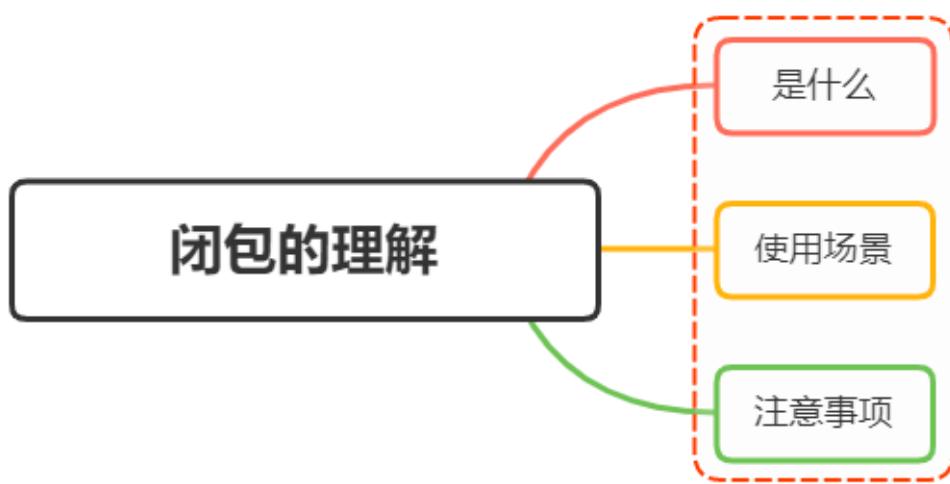
- 减少整个页面所需的内存，提升整体性能
- 动态绑定，减少重复工作

但是使用事件委托也是存在局限性：

- focus 、 blur 这些事件没有事件冒泡机制，所以无法进行委托绑定事件
- mousemove 、 mouseout 这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，因此也是不适合于事件委托的

如果把所有事件都用事件代理，可能会出现事件误判，即本不该被触发的事件被绑定上了事件

# 15. 说说你对闭包的理解？闭包使用场景



## 15.1. 是什么

一个函数和对其周围状态（lexical environment, 词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）

也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域

在 `JavaScript` 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来，作为函数内部与外部连接起来的一座桥梁

下面给出一个简单的例子

```
▼                                     JavaScript | ⌂ 复制代码
```

```
1 function init() {  
2     var name = "Mozilla"; // name 是一个被 init 创建的局部变量  
3     function displayName() { // displayName() 是内部函数，一个闭包  
4         alert(name); // 使用了父函数中声明的变量  
5     }  
6     displayName();  
7 }  
8 init();
```

`displayName()` 没有自己的局部变量。然而，由于闭包的特性，它可以访问到外部函数的变量

## 15.2. 使用场景

任何闭包的使用场景都离不开这两点：

- 创建私有变量
- 延长变量的生命周期

一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

下面举个例子：

在页面上添加一些可以调整字号的按钮

```
▼ JavaScript | 复制代码

1 function makeSizer(size) {
2   return function() {
3     document.body.style.fontSize = size + 'px';
4   };
5 }
6
7 var size12 = makeSizer(12);
8 var size14 = makeSizer(14);
9 var size16 = makeSizer(16);
10
11 document.getElementById('size-12').onclick = size12;
12 document.getElementById('size-14').onclick = size14;
13 document.getElementById('size-16').onclick = size16;
```

### 15.2.1. 柯里化函数

柯里化的目的在于避免频繁调用具有相同参数函数的同时，又能够轻松的重用

```
1 // 假设我们有一个求长方形面积的函数
2 function getArea(width, height) {
3     return width * height
4 }
5 // 如果我们碰到的长方形的宽老是10
6 const area1 = getArea(10, 20)
7 const area2 = getArea(10, 30)
8 const area3 = getArea(10, 40)
9
10 // 我们可以使用闭包柯里化这个计算面积的函数
11 function getArea(width) {
12     return height => {
13         return width * height
14     }
15 }
16
17 const getTenWidthArea = getArea(10)
18 // 之后碰到宽度为10的长方形就可以这样计算面积
19 const area1 = getTenWidthArea(20)
20
21 // 而且如果遇到宽度偶尔变化也可以轻松复用
22 const getTwentyWidthArea = getArea(20)
```

## 15.2.2. 使用闭包模拟私有方法

在 `JavaScript` 中，没有支持声明私有变量，但我们可以使用闭包来模拟私有方法

下面举个例子：

```

1 var Counter = (function() {
2     var privateCounter = 0;
3     function changeBy(val) {
4         privateCounter += val;
5     }
6     return {
7         increment: function() {
8             changeBy(1);
9         },
10        decrement: function() {
11            changeBy(-1);
12        },
13        value: function() {
14            return privateCounter;
15        }
16    }
17 })();
18
19 var Counter1 = makeCounter();
20 var Counter2 = makeCounter();
21 console.log(Counter1.value()); /* logs 0 */
22 Counter1.increment();
23 Counter1.increment();
24 console.log(Counter1.value()); /* logs 2 */
25 Counter1.decrement();
26 console.log(Counter1.value()); /* logs 1 */
27 console.log(Counter2.value()); /* logs 0 */

```

上述通过使用闭包来定义公共函数，并令其可以访问私有函数和变量，这种方式也叫模块方式

两个计数器 `Counter1` 和 `Counter2` 是维护它们各自的独立性的，每次调用其中一个计数器时，通过改变这个变量的值，会改变这个闭包的词法环境，不会影响另一个闭包中的变量

### 15.2.3. 其他

例如计数器、延迟调用、回调等闭包的应用，其核心思想还是创建私有变量和延长变量的生命周期

## 15.3. 注意事项

如果不是某些特定任务需要使用闭包，在其它函数中创建函数是不明智的，因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响

例如，在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。

原因在于每个对象的创建，方法都会被重新赋值

```
1 function MyObject(name, message) {  
2     this.name = name.toString();  
3     this.message = message.toString();  
4     this.getName = function() {  
5         return this.name;  
6     };  
7  
8     this.getMessage = function() {  
9         return this.message;  
10    };  
11}
```

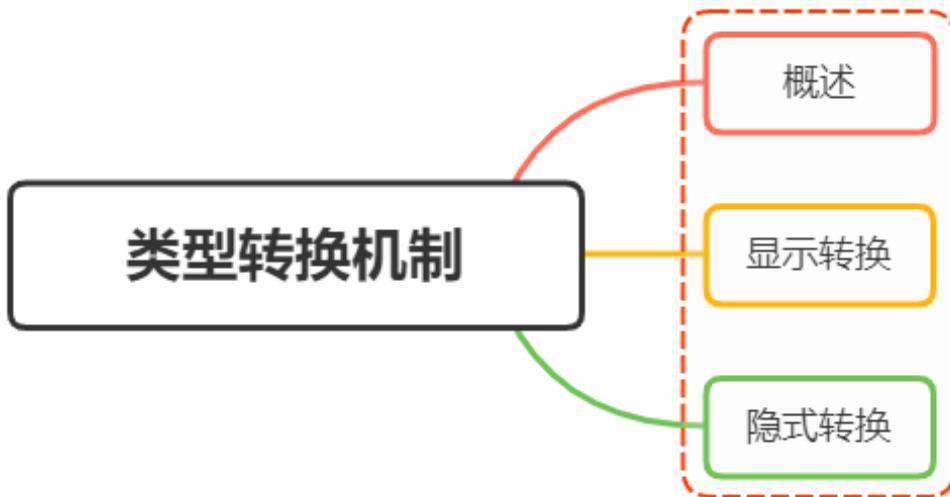
JavaScript | 复制代码

上面的代码中，我们并没有利用到闭包的好处，因此可以避免使用闭包。修改成如下：

```
1 function MyObject(name, message) {  
2     this.name = name.toString();  
3     this.message = message.toString();  
4 }  
5 MyObject.prototype.getName = function() {  
6     return this.name;  
7 };  
8 MyObject.prototype.getMessage = function() {  
9     return this.message;  
10};
```

JavaScript | 复制代码

## 16. 谈谈 JavaScript 中的类型转换机制



## 16.1. 概述

前面我们讲到，JS 中有六种简单数据类型：`undefined`、`null`、`boolean`、`string`、`number`、`symbol`，以及引用类型：`object`

但是我们在声明的时候只有一种数据类型，只有到运行期间才会确定当前类型

```
▼
JavaScript | ⌂ 复制代码
1 let x = y ? 1 : a;
```

上面代码中，`x` 的值在编译阶段是无法获取的，只有等到程序运行时才能知道

虽然变量的数据类型是不确定的，但是各种运算符对数据类型是有要求的，如果运算子的类型与预期不符合，就会触发类型转换机制

常见的类型转换有：

- 强制转换（显示转换）
- 自动转换（隐式转换）

## 16.2. 显示转换

显示转换，即我们很清楚可以看到这里发生了类型的转变，常见的方法有：

- `Number()`
- `parseInt()`
- `String()`
- `Boolean()`

## 16.2.1. Number()

将任意类型的值转化为数值

先给出类型转换规则：

原始值	转换结果
Undefined	NaN
Null	0
true	1
false	0
String	根据语法和转换规则来转换
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

JavaScript | 复制代码

```
1 Number(324) // 324
2
3 // 字符串：如果可以被解析为数值，则转换为相应的数值
4 Number('324') // 324
5
6 // 字符串：如果不可以被解析为数值，返回 NaN
7 Number('324abc') // NaN
8
9 // 空字符串转为0
10 Number('') // 0
11
12 // 布尔值：true 转成 1, false 转成 0
13 Number(true) // 1
14 Number(false) // 0
15
16 // undefined：转成 NaN
17 Number(undefined) // NaN
18
19 // null：转成0
20 Number(null) // 0
21
22 // 对象：通常转换成NaN（除了只包含单个数值的数组）
23 Number({a: 1}) // NaN
24 Number([1, 2, 3]) // NaN
25 Number([5]) // 5
```

从上面可以看到，`Number` 转换的时候是很严格的，只要有一个字符无法转成数值，整个字符串就会被转为 `NaN`

## 16.2.2. `parseInt()`

`parseInt` 相比 `Number`，就没那么严格了，`parseInt` 函数逐个解析字符，遇到不能转换的字符就停下来

JavaScript | 复制代码

```
1 parseInt('32a3') //32
```

## 16.2.3. `String()`

可以将任意类型的值转化成字符串

给出转换规则图：

原始值	转换结果
Undefined	'Undefined'
Boolean	'true' or 'false'
Number	对应的字符串类型
String	String
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

```
▼ JavaScript | ⌂ 复制代码  
1 // 数值：转为相应的字符串  
2 String(1) // "1"  
3  
4 //字符串：转换后还是原来的值  
5 String("a") // "a"  
6  
7 //布尔值：true转为字符串"true"，false转为字符串"false"  
8 String(true) // "true"  
9  
10 //undefined：转为字符串"undefined"  
11 String(undefined) // "undefined"  
12  
13 //null：转为字符串"null"  
14 String(null) // "null"  
15  
16 //对象  
17 String({a: 1}) // "[object Object]"  
18 String([1, 2, 3]) // "1,2,3"
```

## 16.2.4. Boolean()

可以将任意类型的值转为布尔值，转换规则如下：

数据类型	转换为 <code>true</code> 的值	转换为 <code>false</code> 的值
<code>Boolean</code>	<code>true</code>	<code>false</code>
<code>String</code>	非空字符串	<code>""</code> (空字符串)
<code>Number</code>	非零数值 (包括无穷值)	<code>0</code> 、 <code>NaN</code> (参见后面的相关内容)
<code>Object</code>	任意对象	<code>null</code>
<code>Undefined</code>	<code>N/A</code> (不存在)	<code>undefined</code>

实践一下：

▼
JavaScript
复制代码

```

1 Boolean(undefined) // false
2 Boolean(null) // false
3 Boolean(0) // false
4 Boolean(NaN) // false
5 Boolean('') // false
6 Boolean({}) // true
7 Boolean([]) // true
8 Boolean(new Boolean(false)) // true

```

## 16.3. 隐式转换

在隐式转换中，我们可能最大的疑惑是：何时发生隐式转换？

我们这里可以归纳为两种情况发生隐式转换的场景：

- 比较运算 (`==`、`!=`、`>`、`<`)、`if`、`while` 需要布尔值地方
- 算术运算 (`+`、`-`、`*`、`/`、`%`)

除了上面的场景，还要求运算符两边的操作数不是同一类型

### 16.3.1. 自动转换为布尔值

在需要布尔值的地方，就会将非布尔值的参数自动转为布尔值，系统内部会调用 `Boolean` 函数

可以得出个小结：

- `undefined`
- `null`
- `false`
- `+0`
- `-0`

- NaN

- ""

除了上面几种会被转化成 `false`，其他都被转化成 `true`

### 16.3.2. 自动转换成字符串

遇到预期为字符串的地方，就会将非字符串的值自动转为字符串

具体规则是：先将复合类型的值转为原始类型的值，再将原始类型的值转为字符串

常发生在 `+` 运算中，一旦存在字符串，则会进行字符串拼接操作

```
▼ JavaScript | ⌂ 复制代码
1 '5' + 1 // '51'
2 '5' + true // "5true"
3 '5' + false // "5false"
4 '5' + {} // "5[object Object]"
5 '5' + [] // "5"
6 '5' + function (){} // "5function (){}"
7 '5' + undefined // "5undefined"
8 '5' + null // "5null"
```

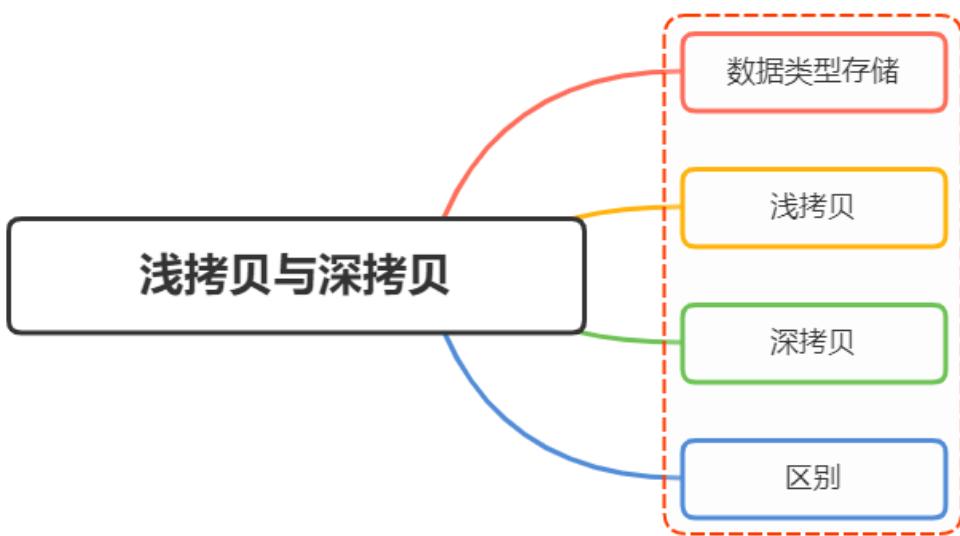
### 16.3.3. 自动转换成数值

除了 `+` 有可能把运算子转为字符串，其他运算符都会把运算子自动转成数值

```
▼ JavaScript | ⌂ 复制代码
1 '5' - '2' // 3
2 '5' * '2' // 10
3 true - 1 // 0
4 false - 1 // -1
5 '1' - 1 // 0
6 '5' * [] // 0
7 false / '5' // 0
8 'abc' - 1 // NaN
9 null + 1 // 1
10 undefined + 1 // NaN
```

`null` 转为数值时，值为 `0`。 `undefined` 转为数值时，值为 `Nan`

# 17. 深拷贝浅拷贝的区别？如何实现一个深拷贝？



## 17.1. 数据类型存储

前面文章我们讲到，`JavaScript` 中存在两大数据类型：

- 基本类型
- 引用类型

基本类型数据保存在栈内存中

引用类型数据保存在堆内存中，引用数据类型的变量是一个指向堆内存中实际对象的引用，存在栈中

## 17.2. 浅拷贝

浅拷贝，指的是创建新的数据，这个数据有着原始数据属性值的一份精确拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，拷贝的就是内存地址

即浅拷贝是拷贝一层，深层次的引用类型则共享内存地址

下面简单实现一个浅拷贝

JavaScript | 复制代码

```
1 function shallowClone(obj) {  
2     const newObj = {};  
3     for(let prop in obj) {  
4         if(obj.hasOwnProperty(prop)){  
5             newObj[prop] = obj[prop];  
6         }  
7     }  
8     return newObj;  
9 }
```

在 `JavaScript` 中，存在浅拷贝的现象有：

- `Object.assign`
- `Array.prototype.slice()`, `Array.prototype.concat()`
- 使用拓展运算符实现的复制

## 17.2.1. Object.assign

JavaScript | 复制代码

```
1 var obj = {  
2     age: 18,  
3     nature: ['smart', 'good'],  
4     names: {  
5         name1: 'fx',  
6         name2: 'xka'  
7     },  
8     love: function () {  
9         console.log('fx is a great girl')  
10    }  
11 }  
12 var newObj = Object.assign({}, fxObj);
```

## 17.2.2. slice()

JavaScript | 复制代码

```
1 const fxArr = ["One", "Two", "Three"]
2 const fxArrs = fxArr.slice(0)
3 fxArrs[1] = "love";
4 console.log(fxArr) // ["One", "Two", "Three"]
5 console.log(fxArrs) // ["One", "love", "Three"]
```

### 17.2.3. concat()

JavaScript | 复制代码

```
1 const fxArr = ["One", "Two", "Three"]
2 const fxArrs = fxArr.concat()
3 fxArrs[1] = "love";
4 console.log(fxArr) // ["One", "Two", "Three"]
5 console.log(fxArrs) // ["One", "love", "Three"]
```

### 17.2.4. 拓展运算符

JavaScript | 复制代码

```
1 const fxArr = ["One", "Two", "Three"]
2 const fxArrs = [...fxArr]
3 fxArrs[1] = "love";
4 console.log(fxArr) // ["One", "Two", "Three"]
5 console.log(fxArrs) // ["One", "love", "Three"]
```

## 17.3. 深拷贝

深拷贝开辟一个新的栈，两个对象属性完全相同，但是对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性

常见的深拷贝方式有：

- `_.cloneDeep()`
- `jQuery.extend()`
- `JSON.stringify()`
- 手写循环递归

### 17.3.1. `_.cloneDeep()`

JavaScript | 复制代码

```
1  const _ = require('lodash');
2  const obj1 = {
3      a: 1,
4      b: { f: { g: 1 } },
5      c: [1, 2, 3]
6  };
7  const obj2 = _.cloneDeep(obj1);
8  console.log(obj1.b.f === obj2.b.f); // false
```

### 17.3.2. `jQuery.extend()`

JavaScript | 复制代码

```
1  const $ = require('jquery');
2  const obj1 = {
3      a: 1,
4      b: { f: { g: 1 } },
5      c: [1, 2, 3]
6  };
7  const obj2 = $.extend(true, {}, obj1);
8  console.log(obj1.b.f === obj2.b.f); // false
```

### 17.3.3. `JSON.stringify()`

JavaScript | 复制代码

```
1  const obj2=JSON.parse(JSON.stringify(obj1));
```

但是这种方式存在弊端，会忽略 `undefined`、`symbol` 和 `函数`

```

1 const obj = {
2     name: 'A',
3     name1: undefined,
4     name3: function() {},
5     name4: Symbol('A')
6 }
7 const obj2 = JSON.parse(JSON.stringify(obj));
8 console.log(obj2); // {name: "A"}

```

### 17.3.4. 循环递归

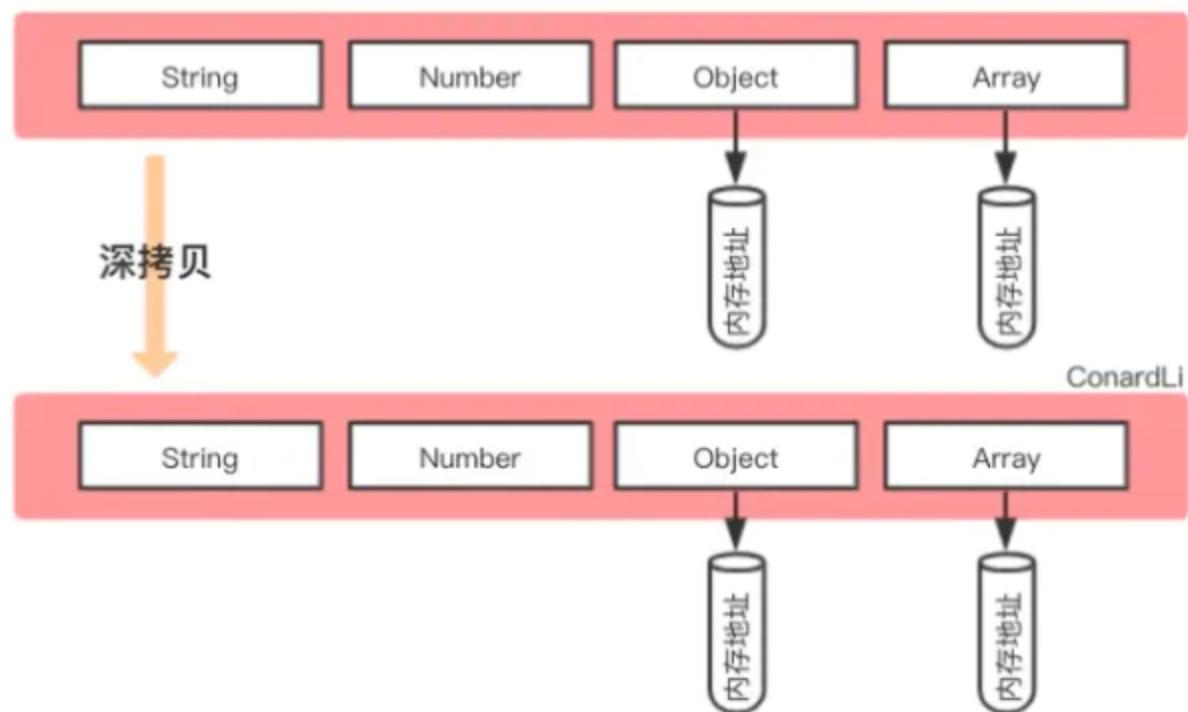
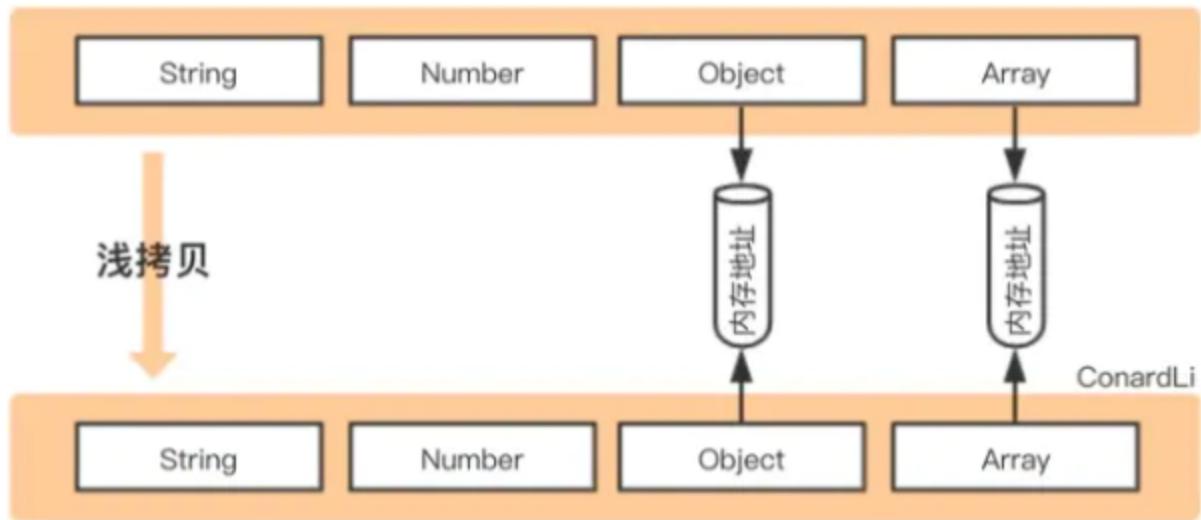
```

1 function deepClone(obj, hash = new WeakMap()) {
2     if (obj === null) return obj; // 如果是null或者undefined就不进行拷贝操作
3     if (obj instanceof Date) return new Date(obj);
4     if (obj instanceof RegExp) return new RegExp(obj);
5     // 可能是对象或者普通的值 如果是函数的话是不需要深拷贝
6     if (typeof obj !== "object") return obj;
7     // 是对象的话就要进行深拷贝
8     if (hash.get(obj)) return hash.get(obj);
9     let cloneObj = new obj.constructor();
10    // 找到的是所属类原型上的constructor,而原型上的 constructor指向的是当前类本身
11    hash.set(obj, cloneObj);
12    for (let key in obj) {
13        if (obj.hasOwnProperty(key)) {
14            // 实现一个递归拷贝
15            cloneObj[key] = deepClone(obj[key], hash);
16        }
17    }
18    return cloneObj;
19 }

```

## 17.4. 区别

下面首先借助两张图，可以更加清晰看到浅拷贝与深拷贝的区别



从上图发现，浅拷贝和深拷贝都创建出一个新的对象，但在复制对象属性的时候，行为就不一样

浅拷贝只复制属性指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存，修改对象属性会影响原对象

JavaScript | 复制代码

```
1 // 浅拷贝
2 const obj1 = {
3     name : 'init',
4     arr : [1,[2,3],4],
5 };
6 const obj3=shallowClone(obj1) // 一个浅拷贝方法
7 obj3.name = "update";
8 obj3.arr[1] = [5,6,7] ; // 新旧对象还是共享同一块内存
9
10 console.log('obj1',obj1) // obj1 { name: 'init', arr: [ 1, [ 5, 6, 7 ], 4 ] }
11 console.log('obj3',obj3) // obj3 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象

JavaScript | 复制代码

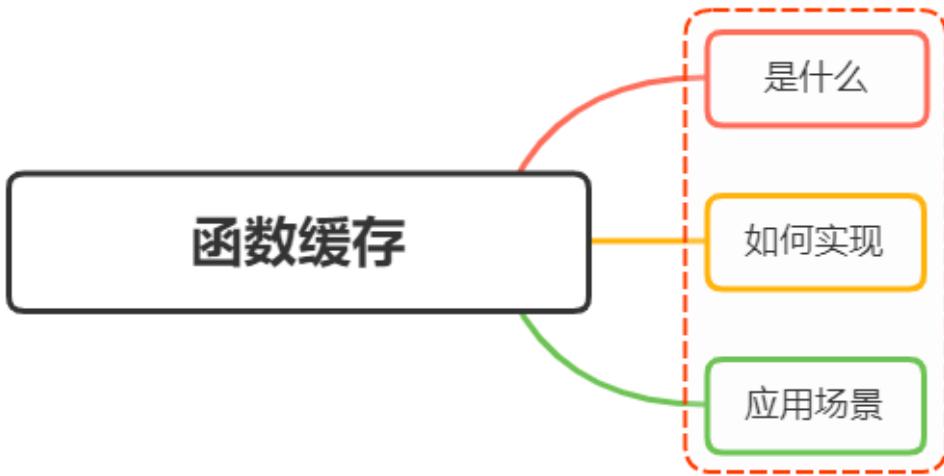
```
1 // 深拷贝
2 const obj1 = {
3     name : 'init',
4     arr : [1,[2,3],4],
5 };
6 const obj4=deepClone(obj1) // 一个深拷贝方法
7 obj4.name = "update";
8 obj4.arr[1] = [5,6,7] ; // 新对象跟原对象不共享内存
9
10 console.log('obj1',obj1) // obj1 { name: 'init', arr: [ 1, [ 2, 3 ], 4 ] }
11 console.log('obj4',obj4) // obj4 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

## 17.5. 小结

前提为拷贝类型为引用类型的情况下：

- 浅拷贝是拷贝一层，属性为对象时，浅拷贝是复制，两个对象指向同一个地址
- 深拷贝是递归拷贝深层次，属性为对象时，深拷贝是新开栈，两个对象指向不同的地址

## 18. Javascript中如何实现函数缓存？函数缓存有哪些应用场景？



## 18.1. 是什么

函数缓存，就是将函数运算过的结果进行缓存

本质上就是用空间（缓存存储）换时间（计算过程）

常用于缓存数据计算结果和缓存对象

```

▼
JavaScript | ⌂ 复制代码
1 const add = (a,b) => a+b;
2 const calc = memoize(add); // 函数缓存
3 calc(10,20); // 30
4 calc(10,20); // 30 缓存

```

缓存只是一个临时的数据存储，它保存数据，以便将来对该数据的请求能够更快地得到处理

## 18.2. 如何实现

实现函数缓存主要依靠闭包、柯里化、高阶函数，这里再简单复习下：

### 18.2.1. 闭包

闭包可以理解成，函数 + 函数体内可访问的变量总和

```

1 (function() {
2     var a = 1;
3     function add() {
4         const b = 2
5         let sum = b + a
6         console.log(sum); // 3
7     }
8     add()
9 })()

```

`add` 函数本身，以及其内部可访问的变量，即 `a = 1`，这两个组合在一起就形成了闭包

## 18.2.2. 柯里化

把接受多个参数的函数转换成接受一个单一参数的函数

```

1 // 非函数柯里化
2 var add = function (x,y) {
3     return x+y;
4 }
5 add(3,4) //7
6
7 // 函数柯里化
8 var add2 = function (x) {
9     /**返回函数**/
10    return function (y) {
11        return x+y;
12    }
13 }
14 add2(3)(4) //7

```

将一个二元函数拆分成两个一元函数

## 18.2.3. 高阶函数

通过接收其他函数作为参数或返回其他函数的函数

JavaScript | 复制代码

```
1 function foo(){
2     var a = 2;
3
4     function bar() {
5         console.log(a);
6     }
7     return bar;
8 }
9 var baz = foo();
10 baz(); //2
```

函数 `foo` 如何返回另一个函数 `bar`，`baz` 现在持有对 `foo` 中定义的 `bar` 函数的引用。由于闭包特性，`a` 的值能够得到

下面再看看如何实现函数缓存，实现原理也很简单，把参数和对应的结果数据存在一个对象中，调用时判断参数对应的数据是否存在，存在就返回对应的结果数据，否则就返回计算结果

如下所示

JavaScript | 复制代码

```
1 const memoize = function (func, content) {
2     let cache = Object.create(null)
3     content = content || this
4     return (...key) => {
5         if (!cache[key]) {
6             cache[key] = func.apply(content, key)
7         }
8         return cache[key]
9     }
10 }
```

调用方式也很简单

JavaScript | 复制代码

```
1 const calc = memoize(add);
2 const num1 = calc(100, 200)
3 const num2 = calc(100, 200) // 缓存得到的结果
```

过程分析：

- 在当前函数作用域定义了一个空对象，用于缓存运行结果
- 运用柯里化返回一个函数，返回的函数由于闭包特性，可以访问到 `cache`

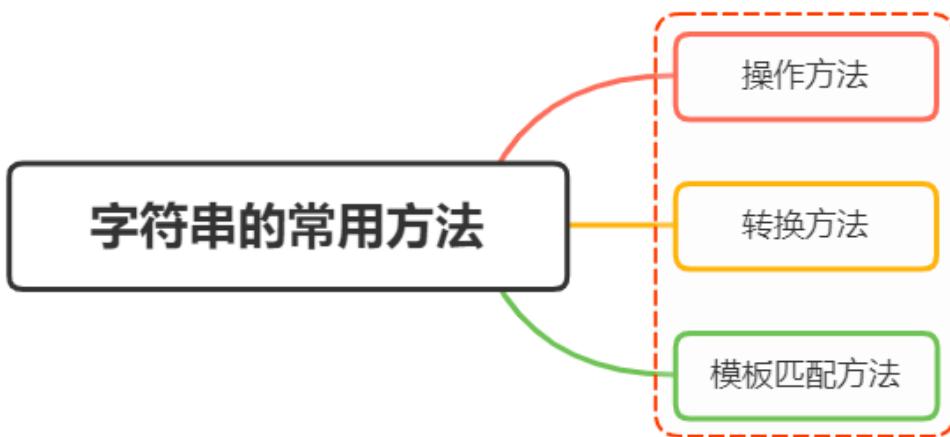
- 然后判断输入参数是不是在 `cache` 的中。如果已经存在，直接返回 `cache` 的内容，如果没有存在，使用函数 `func` 对输入参数求值，然后把结果存储在 `cache` 中

## 18.3. 应用场景

虽然使用缓存效率是非常高的，但并不是所有场景都适用，因此千万不要极端的将所有函数都添加缓存  
以下几种情况下，适合使用缓存：

- 对于昂贵的函数调用，执行复杂计算的函数
- 对于具有有限且高度重复输入范围的函数
- 对于具有重复输入值的递归函数
- 对于纯函数，即每次使用特定输入调用时返回相同输出的函数

## 19. JavaScript字符串的常用方法有哪些？



### 19.1. 操作方法

我们也可将字符串常用的操作方法归纳为增、删、改、查，需要知道字符串的特点是一旦创建了，就不可变

#### 19.1.1. 增

这里增的意思并不是说直接增添内容，而是创建字符串的一个副本，再进行操作

除了常用 `+` 以及 `{}$` 进行字符串拼接之外，还可通过 `concat`

### 19.1.1. concat

用于将一个或多个字符串拼接成一个新字符串

```
1 let stringValue = "hello ";
2 let result = stringValue.concat("world");
3 console.log(result); // "hello world"
4 console.log(stringValue); // "hello"
```

JavaScript | 复制代码

### 19.1.2. 删

这里的删的意思并不是说删除原字符串的内容，而是创建字符串的一个副本，再进行操作

常见的有：

- slice()
- substr()
- substring()

这三个方法都返回调用它们的字符串的一个子字符串，而且都接收一或两个参数。

```
1 let stringValue = "hello world";
2 console.log(stringValue.slice(3)); // "lo world"
3 console.log(stringValue.substring(3)); // "lo world"
4 console.log(stringValue.substr(3)); // "lo world"
5 console.log(stringValue.slice(3, 7)); // "lo w"
6 console.log(stringValue.substring(3,7)); // "lo w"
7 console.log(stringValue.substr(3, 7)); // "lo worl"
```

JavaScript | 复制代码

### 19.1.3. 改

这里改的意思也不是改变原字符串，而是创建字符串的一个副本，再进行操作

常见的有：

- trim()、trimLeft()、trimRight()
- repeat()
- padStart()、padEnd()
- toLowerCase()、toUpperCase()

### 19.1.3.1. trim()、trimLeft()、trimRight()

删除前、后或前后所有空格符，再返回新的字符串

```
1 let stringValue = " hello world ";
2 let trimmedStringValue = stringValue.trim();
3 console.log(stringValue); // " hello world "
4 console.log(trimmedStringValue); // "hello world"
```

JavaScript | 复制代码

### 19.1.3.2. repeat()

接收一个整数参数，表示要将字符串复制多少次，然后返回拼接所有副本后的结果

```
1 let stringValue = "na ";
2 let copyResult = stringValue.repeat(2) // na na
```

JavaScript | 复制代码

### 19.1.3.3. padEnd()

复制字符串，如果小于指定长度，则在相应一边填充字符，直至满足长度条件

```
1 let stringValue = "foo";
2 console.log(stringValue.padStart(6)); // " foo"
3 console.log(stringValue.padStart(9, ".")); // ".....foo"
```

JavaScript | 复制代码

### 19.1.4. toLowerCase()、toUpperCase()

大小写转化

```
1 let stringValue = "hello world";
2 console.log(stringValue.toUpperCase()); // "HELLO WORLD"
3 console.log(stringValue.toLowerCase()); // "hello world"
```

JavaScript | 复制代码

### 19.1.5. 查

除了通过索引的方式获取字符串的值，还可通过：

- `charAt()`
- `indexOf()`
- `startsWith()`
- `includes()`

#### 19.1.5.1. `charAt()`

返回给定索引位置的字符，由传给方法的整数参数指定

```
1 let message = "abcde";
2 console.log(message.charAt(2)); // "c"
```

JavaScript | 复制代码

#### 19.1.5.2. `indexOf()`

从字符串开头去搜索传入的字符串，并返回位置（如果没找到，则返回 `-1`）

```
1 let stringValue = "hello world";
2 console.log(stringValue.indexOf("o")); // 4
```

JavaScript | 复制代码

#### 19.1.5.3. `startsWith()`、`includes()`

从字符串中搜索传入的字符串，并返回一个表示是否包含的布尔值

```
1 let message = "foobarbaz";
2 console.log(message.startsWith("foo")); // true
3 console.log(message.startsWith("bar")); // false
4 console.log(message.includes("bar")); // true
5 console.log(message.includes("qux")); // false
```

JavaScript | 复制代码

## 19.2. 转换方法

### 19.2.1. `split`

把字符串按照指定的分割符，拆分成数组中的每一项

```
1 let str = "12+23+34"
2 let arr = str.split["+") // [12,23,34]
```

JavaScript | 复制代码

## 19.3. 模板匹配方法

针对正则表达式，字符串设计了几个方法：

- match()
- search()
- replace()

### 19.3.1. match()

接收一个参数，可以是一个正则表达式字符串，也可以是一个 RegExp 对象，返回数组

```
1 let text = "cat, bat, sat, fat";
2 let pattern = /.at/;
3 let matches = text.match(pattern);
4 console.log(matches[0]); // "cat"
```

JavaScript | 复制代码

### 19.3.2. search()

接收一个参数，可以是一个正则表达式字符串，也可以是一个 RegExp 对象，找到则返回匹配索引，否则返回 -1

```
1 let text = "cat, bat, sat, fat";
2 let pos = text.search(/at/);
3 console.log(pos); // 1
```

JavaScript | 复制代码

### 19.3.3. replace()

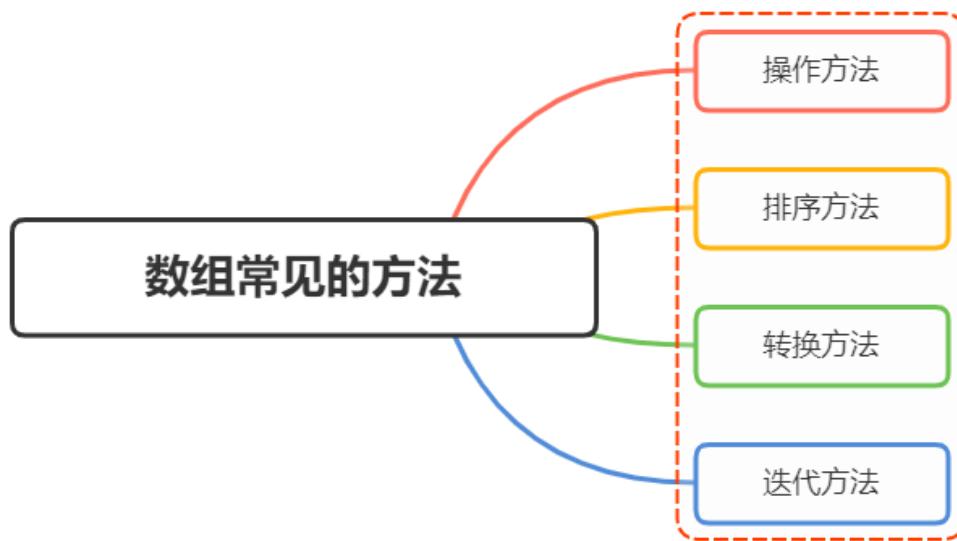
接收两个参数，第一个参数为匹配的内容，第二个参数为替换的元素（可用函数）

```

1 let text = "cat, bat, sat, fat";
2 let result = text.replace("at", "ond");
3 console.log(result); // "cond, bat, sat, fat"

```

## 20. 数组的常用方法有哪些？



### 20.1. 操作方法

数组基本操作可以归纳为 增、删、改、查，需要留意的是哪些方法会对原数组产生影响，哪些方法不会。下面对数组常用的操作方法做一个归纳。

#### 20.1.1. 增

下面前三种是对原数组产生影响的增添方法，第四种则不会对原数组产生影响。

- push()
- unshift()
- splice()
- concat()

##### 20.1.1.1. push()

`push()` 方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度

JavaScript | 复制代码

```
1 let colors = []; // 创建一个数组
2 let count = colors.push("red", "green"); // 推入两项
3 console.log(count) // 2
```

### 20.1.1.2. `unshift()`

`unshift()`在数组开头添加任意多个值，然后返回新的数组长度

JavaScript | 复制代码

```
1 let colors = new Array(); // 创建一个数组
2 let count = colors.unshift("red", "green"); // 从数组开头推入两项
3 alert(count); // 2
```

### 20.1.1.3. `splice()`

传入三个参数，分别是开始位置、0（要删除的元素数量）、插入的元素，返回空数组

JavaScript | 复制代码

```
1 let colors = ["red", "green", "blue"];
2 let removed = colors.splice(1, 0, "yellow", "orange")
3 console.log(colors) // red,yellow,orange,green,blue
4 console.log(removed) // []
```

### 20.1.1.4. `concat()`

首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组，不会影响原始数组

JavaScript | 复制代码

```
1 let colors = ["red", "green", "blue"];
2 let colors2 = colors.concat("yellow", ["black", "brown"]);
3 console.log(colors); // ["red", "green", "blue"]
4 console.log(colors2); // ["red", "green", "blue", "yellow", "black", "brown"]
```

## 20.1.2. 删

下面三种都会影响原数组，最后一项不影响原数组：

- `pop()`
- `shift()`
- `splice()`
- `slice()`

### 20.1.2.1. `pop()`

`pop()` 方法用于删除数组的最后一项，同时减少数组的 `length` 值，返回被删除的项

```
1 let colors = ["red", "green"]
2 let item = colors.pop(); // 取得最后一项
3 console.log(item) // green
4 console.log(colors.length) // 1
```

JavaScript | 复制代码

### 20.1.2.2. `shift()`

`shift()` 方法用于删除数组的第一项，同时减少数组的 `length` 值，返回被删除的项

```
1 let colors = ["red", "green"]
2 let item = colors.shift(); // 取得第一项
3 console.log(item) // red
4 console.log(colors.length) // 1
```

JavaScript | 复制代码

### 20.1.2.3. `splice()`

传入两个参数，分别是开始位置，删除元素的数量，返回包含删除元素的数组

```
1 let colors = ["red", "green", "blue"];
2 let removed = colors.splice(0,1); // 删除第一项
3 console.log(colors); // green,blue
4 console.log(removed); // red, 只有一个元素的数组
```

JavaScript | 复制代码

### 20.1.3. slice()

slice() 用于创建一个包含原有数组中一个或多个元素的新数组，不会影响原始数组

```
1 let colors = ["red", "green", "blue", "yellow", "purple"];
2 let colors2 = colors.slice(1);
3 let colors3 = colors.slice(1, 4);
4 console.log(colors) // red,green,blue,yellow,purple
5 console.log(colors2); // green,blue,yellow,purple
6 console.log(colors3); // green,blue,yellow
```

#### 20.1.3.1. 改

即修改原来数组的内容，常用 `splice`

#### 20.1.3.2. splice()

传入三个参数，分别是开始位置，要删除元素的数量，要插入的任意多个元素，返回删除元素的数组，对原数组产生影响

```
1 let colors = ["red", "green", "blue"];
2 let removed = colors.splice(1, 1, "red", "purple"); // 插入两个值，删除一个元素
3 console.log(colors); // red,red,purple,blue
4 console.log(removed); // green, 只有一个元素的数组
```

#### 20.1.3.3. 查

即查找元素，返回元素坐标或者元素值

- `indexOf()`
- `includes()`
- `find()`

#### 20.1.3.4. indexOf()

返回要查找的元素在数组中的位置，如果没找到则返回 -1

JavaScript | 复制代码

```
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 numbers.indexOf(4) // 3
```

### 20.1.3.5. includes()

返回要查找的元素在数组中的位置，找到返回 `true`，否则 `false`

JavaScript | 复制代码

```
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 numbers.includes(4) // true
```

### 20.1.3.6. find()

返回第一个匹配的元素

JavaScript | 复制代码

```
1 const people = [
2   {
3     name: "Matt",
4     age: 27
5   },
6   {
7     name: "Nicholas",
8     age: 29
9   }
10 ];
11 people.find((element, index, array) => element.age < 28) // // {name: "Mat
t", age: 27}
```

## 20.2. 排序方法

数组有两个方法可以用来对元素重新排序：

- `reverse()`
- `sort()`

### 20.2.1. reverse()

顾名思义，将数组元素方向反转

```
1 let values = [1, 2, 3, 4, 5];
2 values.reverse();
3 alert(values); // 5,4,3,2,1
```

JavaScript | 复制代码

## 20.2.2. sort()

sort()方法接受一个比较函数，用于判断哪个值应该排在前面

```
1 function compare(value1, value2) {
2   if (value1 < value2) {
3     return -1;
4   } else if (value1 > value2) {
5     return 1;
6   } else {
7     return 0;
8   }
9 }
10 let values = [0, 1, 5, 10, 15];
11 values.sort(compare);
12 alert(values); // 0,1,5,10,15
```

JavaScript | 复制代码

## 20.3. 转换方法

常见的转换方法有：

### 20.3.1. join()

join() 方法接收一个参数，即字符串分隔符，返回包含所有项的字符串

```
1 let colors = ["red", "green", "blue"];
2 alert(colors.join(","));
3 alert(colors.join("||"));
```

JavaScript | 复制代码

## 20.4. 迭代方法

常用来迭代数组的方法（都不改变原数组）有如下：

- some()
- every()
- forEach()
- filter()
- map()

### 20.4.1. some()

对数组每一项都运行传入的测试函数，如果至少有1个元素返回 true，则这个方法返回 true

```
▼ JavaScript | ⌂ 复制代码
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 let someResult = numbers.some((item, index, array) => item > 2);
3 console.log(someResult) // true
```

### 20.4.2. every()

对数组每一项都运行传入的测试函数，如果所有元素都返回 true，则这个方法返回 true

```
▼ JavaScript | ⌂ 复制代码
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 let everyResult = numbers.every((item, index, array) => item > 2);
3 console.log(everyResult) // false
```

### 20.4.3. forEach()

对数组每一项都运行传入的函数，没有返回值

```
▼ JavaScript | ⌂ 复制代码
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 numbers.forEach((item, index, array) => {
3     // 执行某些操作
4});
```

#### 20.4.4. filter()

对数组每一项都运行传入的函数，函数返回 `true` 的项会组成数组之后返回

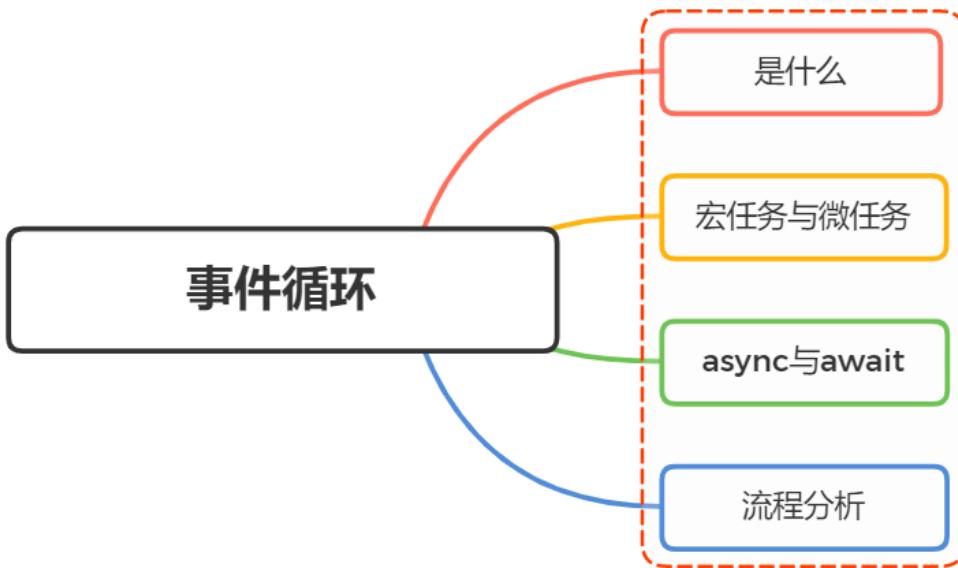
```
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 let filterResult = numbers.filter((item, index, array) => item > 2);
3 console.log(filterResult); // 3,4,5,4,3
```

#### 20.4.5. map()

对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组

```
1 let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2 let mapResult = numbers.map((item, index, array) => item * 2);
3 console.log(mapResult) // 2,4,6,8,10,8,6,4,2
```

## 21. 说说你对事件循环的理解



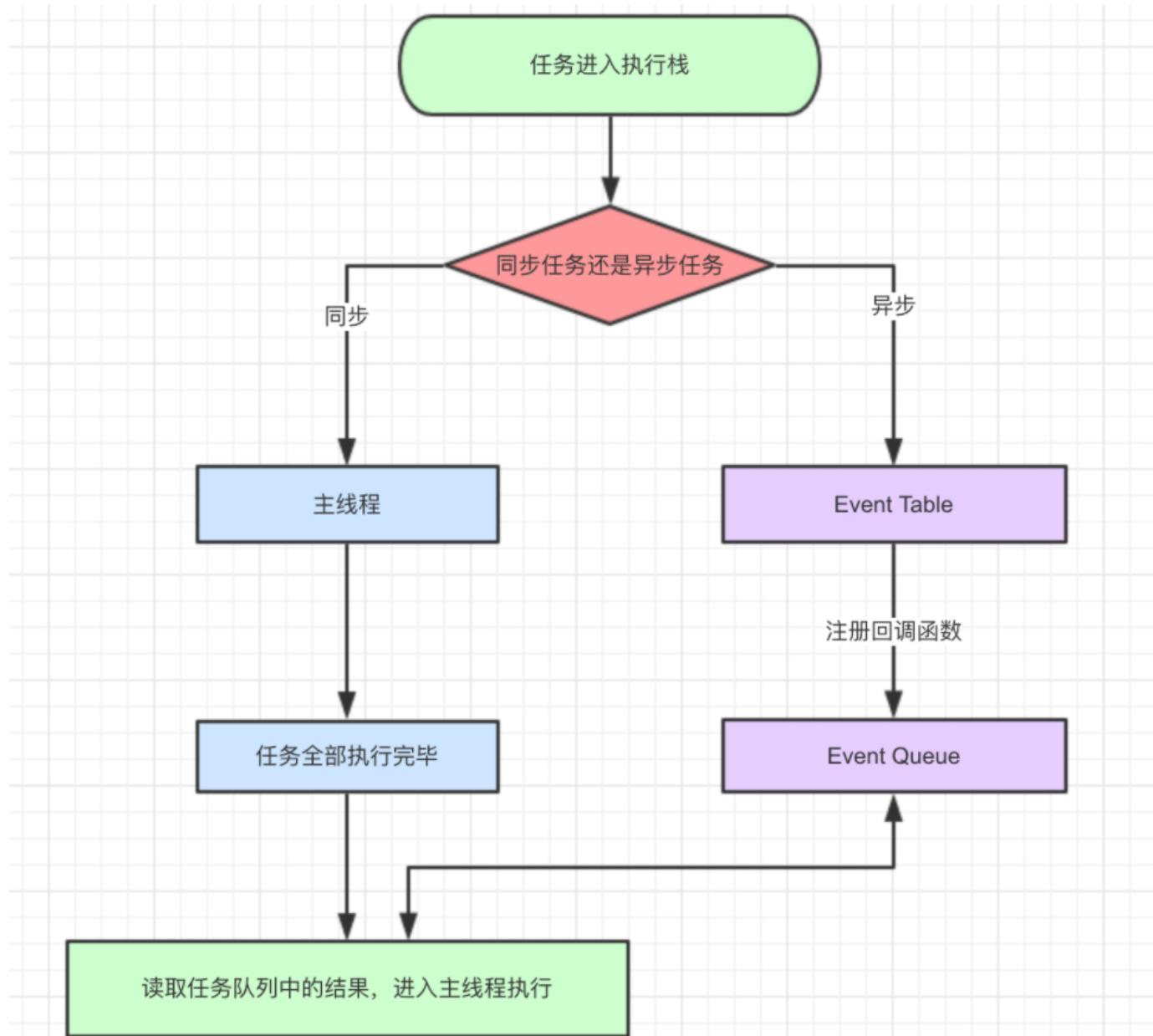
### 21.1. 是什么

首先，`JavaScript` 是一门单线程的语言，意味着同一时间内只能做一件事，但是这并不意味着单线程就是阻塞，而实现单线程非阻塞的方法就是事件循环

在 `JavaScript` 中，所有的任务都可以分为

- 同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行
- 异步任务：异步执行的任务，比如 `ajax` 网络请求，`setTimeout` 定时函数等

同步任务与异步任务的运行流程图如下：



从上面我们可以看到，同步任务进入主线程，即主执行栈，异步任务进入任务队列，主线程内的任务执行完毕为空，会去任务队列读取对应的任务，推入主线程执行。上述过程的不断重复就事件循环

## 21.2. 宏任务与微任务

如果将任务划分为同步任务和异步任务并不是那么的准确，举个例子：

```
1  console.log(1)
2
3  setTimeout(()=>{
4      console.log(2)
5  }, 0)
6
7  new Promise((resolve, reject)=>{
8      console.log('new Promise')
9      resolve()
10 })
11 })
12
13
14 console.log(3)
```

如果按照上面流程图来分析代码，我们会得到下面的执行步骤：

- `console.log(1)`，同步任务，主线程中执行
- `setTimeout()`，异步任务，放到 `Event Table`，0 毫秒后 `console.log(2)` 回调推入 `Event Queue` 中
- `new Promise`，同步任务，主线程直接执行
- `.then`，异步任务，放到 `Event Table`
- `console.log(3)`，同步任务，主线程执行

所以按照分析，它的结果应该是 `1 => 'new Promise' => 3 => 2 => 'then'`

但是实际结果是：`1 => 'new Promise' => 3 => 'then' => 2`

出现分歧的原因在于异步任务执行顺序，事件队列其实是一个“先进先出”的数据结构，排在前面的事件会优先被主线程读取

例子中 `setTimeout` 回调事件是先进入队列中的，按理说应该先于 `.then` 中的执行，但是结果却偏偏相反

原因在于异步任务还可以细分为微任务与宏任务

### 21.2.1. 微任务

一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前

常见的微任务有：

- Promise.then
- MutationObserver
- Object.observe (已废弃；Proxy 对象替代)
- process.nextTick (Node.js)

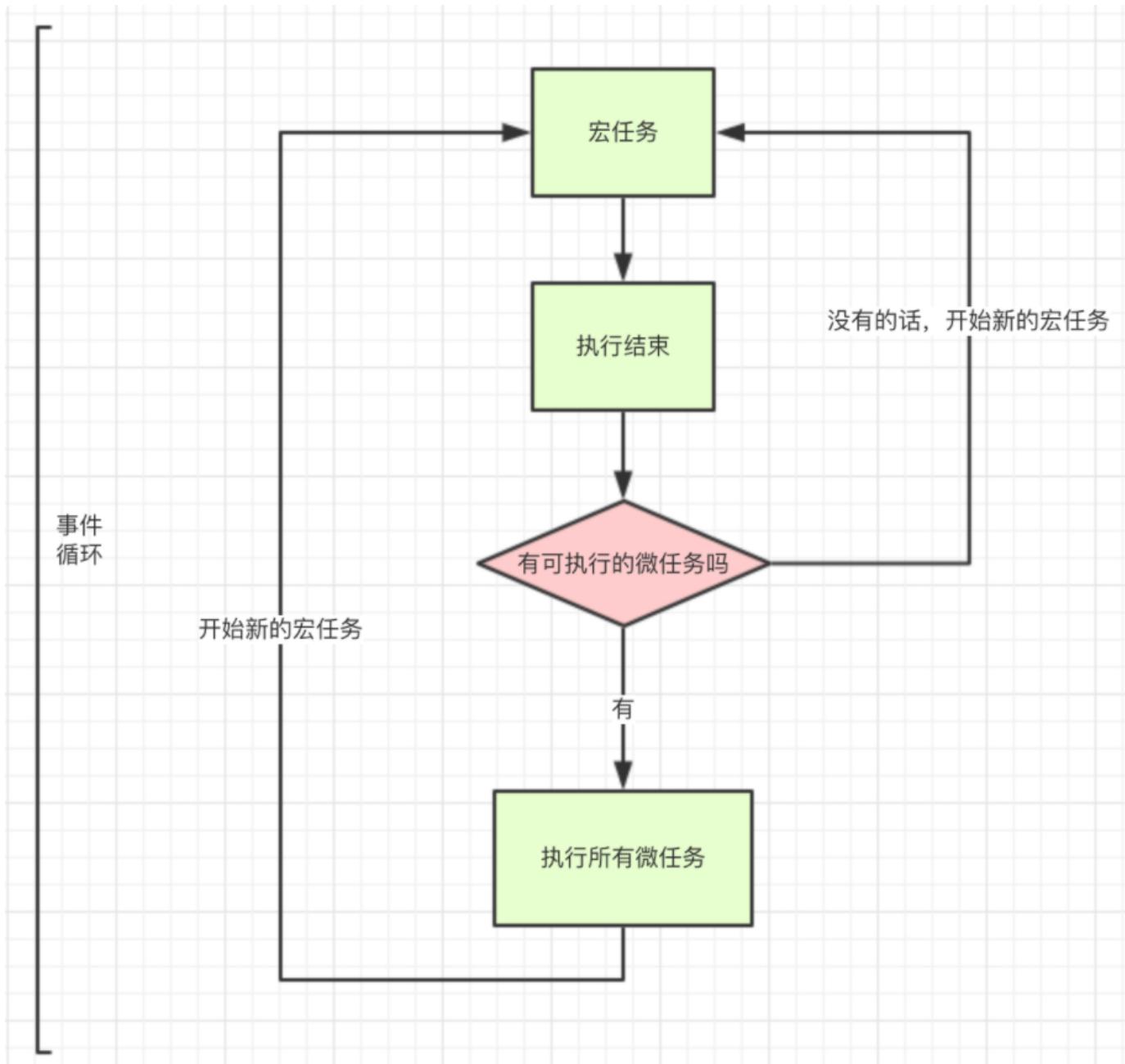
## 21.2.2. 宏任务

宏任务的时间粒度比较大，执行的时间间隔是不能精确控制的，对一些高实时性的需求就不太符合

常见的宏任务有：

- script (可以理解为外层同步代码)
- setTimeout/setInterval
- UI rendering/UI事件
- postMessage、MessageChannel
- setImmediate、I/O (Node.js)

这时候，事件循环，宏任务，微任务的关系如图所示



按照这个流程，它的执行机制是：

- 执行一个宏任务，如果遇到微任务就将它放到微任务的事件队列中
- 当前宏任务执行完成后，会查看微任务的事件队列，然后将里面的所有微任务依次执行完

[回到上面的题目](#)

JavaScript | 复制代码

```
1 console.log(1)
2 setTimeout(()=>{
3     console.log(2)
4 }, 0)
5 new Promise((resolve, reject)=>{
6     console.log('new Promise')
7     resolve()
8 }).then(()=>{
9     console.log('then')
10 })
11 console.log(3)
```

流程如下

JavaScript | 复制代码

```
1 // 遇到 console.log(1) , 直接打印 1
2 // 遇到定时器, 属于新的宏任务, 留着后面执行
3 // 遇到 new Promise, 这个是直接执行的, 打印 'new Promise'
4 // .then 属于微任务, 放入微任务队列, 后面再执行
5 // 遇到 console.log(3) 直接打印 3
6 // 好了本轮宏任务执行完毕, 现在去微任务列表查看是否有微任务, 发现 .then 的回调, 执行它,
    打印 'then'
7 // 当一次宏任务执行完, 再去执行新的宏任务, 这里就剩一个定时器的宏任务了, 执行它, 打印 2
```

## 21.3. `async`与`await`

`async` 是异步的意思, `await` 则可以理解为 `async wait`。所以可以理解 `async` 就是用来声明一个异步方法, 而 `await` 是用来等待异步方法执行

### 21.3.1. `async`

`async` 函数返回一个 `promise` 对象, 下面两种方法是等效的

JavaScript | 复制代码

```
1 function f() {
2     return Promise.resolve('TEST');
3 }
4
5 // asyncF is equivalent to f!
6 async function asyncF() {
7     return 'TEST';
8 }
```

### 21.3.2. await

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

JavaScript | 复制代码

```
1 async function f(){
2     // 等同于
3     // return 123
4     return await 123
5 }
6 f().then(v => console.log(v)) // 123
```

不管 `await` 后面跟着的是什么，`await` 都会阻塞后面的代码

JavaScript | 复制代码

```
1 async function fn1 (){
2     console.log(1)
3     await fn2()
4     console.log(2) // 阻塞
5 }
6
7 async function fn2 (){
8     console.log('fn2')
9 }
10
11 fn1()
12 console.log(3)
```

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

所以上述输出结果为: 1 , fn2 , 3 , 2

## 21.4. 流程分析

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

这里直接上代码：

The screenshot shows a browser's developer tools console with the following code and output:

```
1 -> async function async1() {
2     console.log('async1 start')
3     await async2()
4     console.log('async1 end')
5 }
6 -> async function async2() {
7     console.log('async2')
8 }
9 console.log('script start')
10 -> setTimeout(function () {
11     console.log('settimeout')
12 })
13 async1()
14 -> new Promise(function (resolve) {
15     console.log('promise1')
16     resolve()
17 }).then(function () {
18     console.log('promise2')
19 })
20 console.log('script end')
```

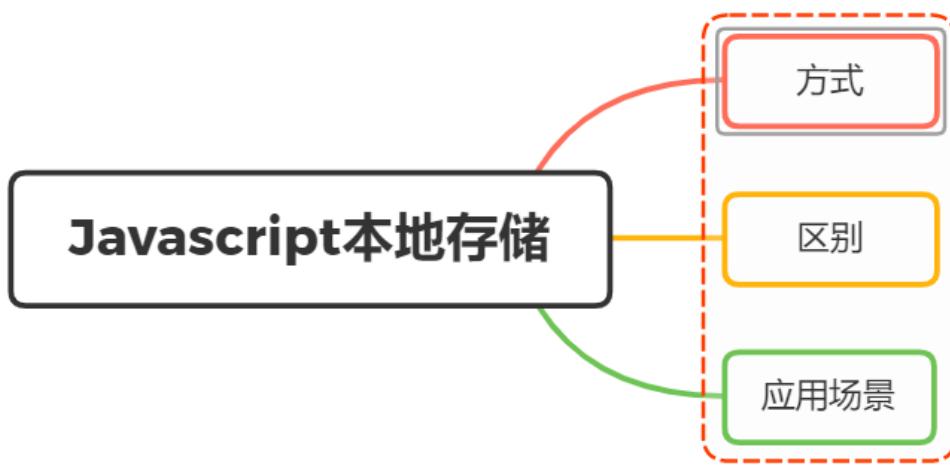
The console output is:  
script start  
promise1  
promise2  
script end

分析过程：

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

6. 继续执行下一个微任务，即执行 `then` 的回调，打印 `promise2`
  7. 上一个宏任务所有事都做完了，开始下一个宏任务，就是定时器，打印 `settimeout`
- 所以最后的结果是：`script start`、`async1 start`、`async2`、`promise1`、`script end`、`async1 end`、`promise2`、`settimeout`

## 22. Javascript本地存储的方式有哪些？区别及应用场景？



### 22.1. 方式

`javaScript` 本地缓存的方法我们主要讲述以下四种：

- cookie
- sessionStorage
- localStorage
- indexedDB

#### 22.1.1. cookie

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

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

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

关于 `cookie` 常用的属性如下：

- `Expires` 用于设置 Cookie 的过期时间

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

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

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

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

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

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

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

关于 `cookie` 的使用如下：

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

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

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

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

## 22.1.2. localStorage

HTML5 新方法，IE8及以上浏览器都兼容

## 22.1.3. 特点

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

下面再看看关于 `localStorage` 的使用

设置

```
▼ JavaScript | 复制代码
1 localStorage.setItem('username', 'cfangxu');
```

获取

```
▼ JavaScript | 复制代码
1 localStorage.getItem('username')
```

获取键名

```
▼ JavaScript | 复制代码
1 localStorage.key(0) // 获取第一个键名
```

删除

JavaScript | 复制代码

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

一次性清除所有存储

JavaScript | 复制代码

```
1 localStorage.clear()
```

`localStorage` 也不是完美的，它有两个缺点：

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

JavaScript | 复制代码

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

## 22.1.4. sessionStorage

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

## 22.1.5. 扩展的前端存储方式

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

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

### 22.1.5.1. 优点：

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

### 22.1.5.2. 缺点：

- 操作非常繁琐
- 本身有一定门槛

关于 `indexedDB` 的使用基本使用步骤如下：

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

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

## 22.2. 区别

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

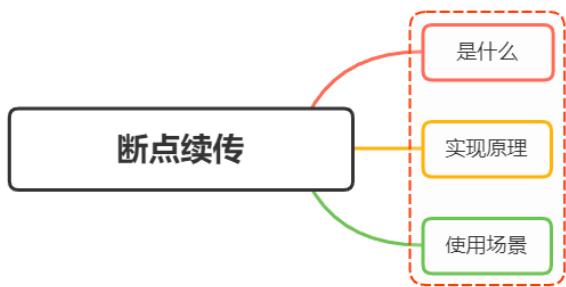
- 存储大小：`cookie` 数据大小不能超过 `4k`，`sessionStorage` 和 `localStorage` 虽然也有存储大小的限制，但比 `cookie` 大得多，可以达到 `5M` 或更大
- 有效时间：`localStorage` 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；`sessionStorage` 数据在当前浏览器窗口关闭后自动删除；`cookie` 设置的 `cookie` 过期时间之前一直有效，即使窗口或浏览器关闭
- 数据与服务器之间的交互方式，`cookie` 的数据会自动的传递到服务器，服务器端也可以写 `cookie` 到客户端；`sessionStorage`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

## 22.3. 应用场景

在了解了上述的前端的缓存方式后，我们可以看看针对不同场景的使用选择：

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

# 23. 大文件上传如何做断点续传?



## 23.1. 是什么

不管怎样简单的需求，在量级达到一定层次时，都会变得异常复杂

文件上传简单，文件变大就复杂

上传大文件时，以下几个变量会影响我们的用户体验

- 服务器处理数据的能力
- 请求超时
- 网络波动

上传时间会变长，高频次文件上传失败，失败后又需要重新上传等等

为了解决上述问题，我们需要对大文件上传单独处理

这里涉及到分片上传及断点续传两个概念

### 23.1.1. 分片上传

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

如下图



上传完之后再由服务端对所有上传的文件进行汇总整合成原始的文件

大致流程如下：

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

### 23.1.2. 断点续传

断点续传指的是在下载或上传时，将下载或上传任务人为的划分为几个部分

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

一般实现方式有两种：

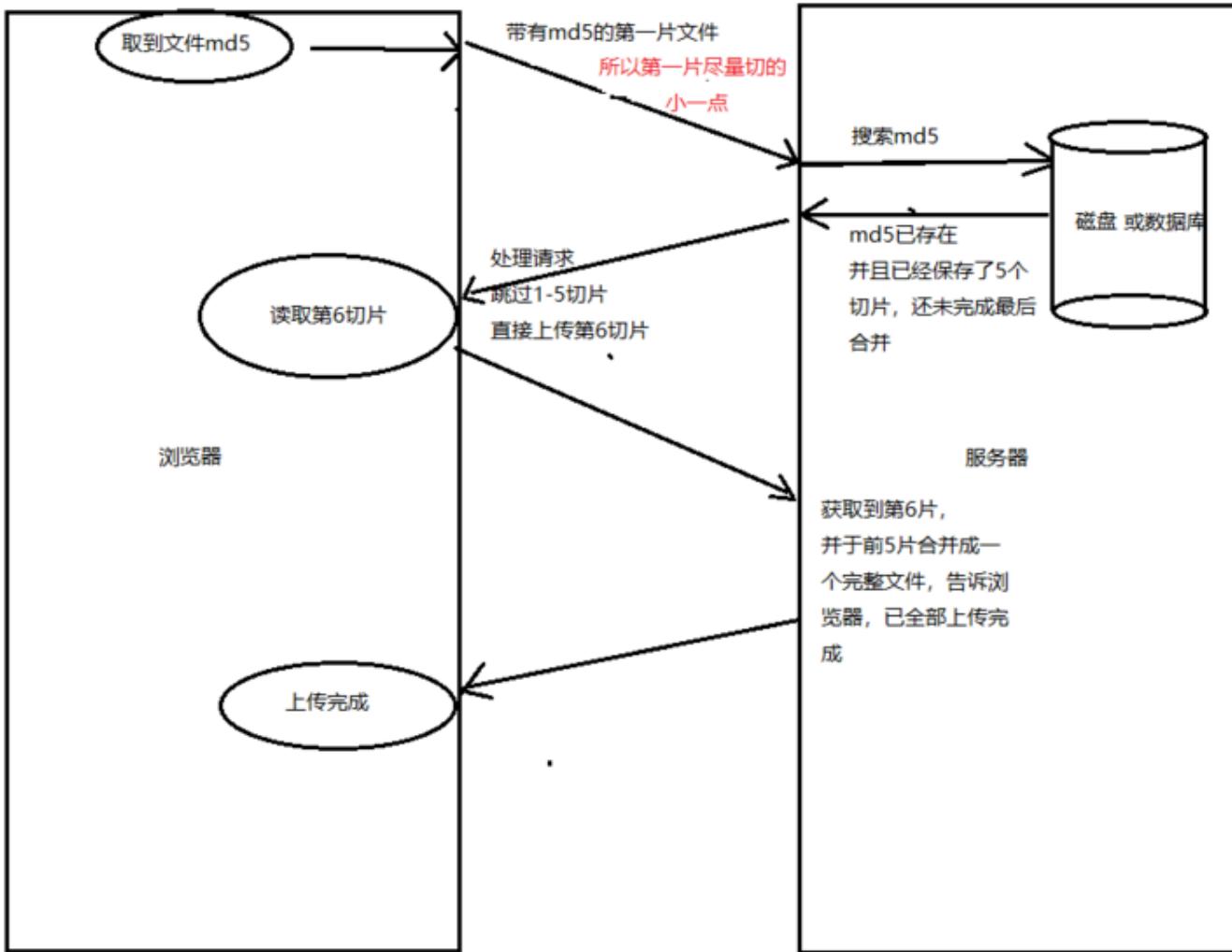
- 服务器端返回，告知从哪开始
- 浏览器端自行处理

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

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

## 23.2. 实现思路

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



下面的内容都是伪代码

读取文件内容：

```

1 const input = document.querySelector('input');
2 input.addEventListener('change', function() {
3     var file = this.files[0];
4 });

```

JavaScript | 复制代码

可以使用 `md5` 实现文件的唯一性

```

1 const md5code = md5(file);

```

JavaScript | 复制代码

然后开始对文件进行分割

JavaScript | 复制代码

```
1 var reader = new FileReader();
2 reader.readAsArrayBuffer(file);
3 reader.addEventListener("load", function(e) {
4     //每10M切割一段，这里只做一个切割演示，实际切割需要循环切割,
5     var slice = e.target.result.slice(0, 10*1024*1024);
6 });
```

h5上传一个（一片）

JavaScript | 复制代码

```
1 const formdata = new FormData();
2 formdata.append('0', slice);
3 //这里是有一个坑的，部分设备无法获取文件名称，和文件类型，这个在最后给出解决方案
4 formdata.append('filename', file.filename);
5 var xhr = new XMLHttpRequest();
6 xhr.addEventListener('load', function() {
7     //xhr.responseText
8 });
9 xhr.open('POST', '');
10 xhr.send(formdata);
11 xhr.addEventListener('progress', updateProgress);
12 xhr.upload.addEventListener('progress', updateProgress);
13
14 function updateProgress(event) {
15     if (event.lengthComputable) {
16         //进度条
17     }
18 }
```

这里给出常见的图片和视频的文件类型判断

```
1 function checkFileType(type, file, back) {
2 /**
3 * type png jpg mp4 ...
4 * file input.change=> this.files[0]
5 * back callback(boolean)
6 */
7     var args = arguments;
8     if (args.length != 3) {
9         back(0);
10    }
11    var type = args[0]; // type = '(png|jpg)' , 'png'
12    var file = args[1];
13    var back = typeof args[2] == 'function' ? args[2] : function() {};
14    if (file.type == '') {
15        // 如果系统无法获取文件类型, 则读取二进制流, 对二进制进行解析文件类型
16        var imgType = [
17            'ff d8 ff', //jpg
18            '89 50 4e', //png
19
20            '0 0 0 14 66 74 79 70 69 73 6F 6D', //mp4
21            '0 0 0 18 66 74 79 70 33 67 70 35', //mp4
22            '0 0 0 0 66 74 79 70 33 67 70 35', //mp4
23            '0 0 0 0 66 74 79 70 4D 53 4E 56', //mp4
24            '0 0 0 0 66 74 79 70 69 73 6F 6D', //mp4
25
26            '0 0 0 18 66 74 79 70 6D 70 34 32', //m4v
27            '0 0 0 0 66 74 79 70 6D 70 34 32', //m4v
28
29            '0 0 0 14 66 74 79 70 71 74 20 20', //mov
30            '0 0 0 0 66 74 79 70 71 74 20 20', //mov
31            '0 0 0 0 6D 6F 6F 76', //mov
32
33            '4F 67 67 53 0 02', //ogg
34            '1A 45 DF A3', //ogg
35
36            '52 49 46 46 x x x x 41 56 49 20', //avi (RIFF fileSize fileType
37            pe LIST)(52 49 46 46,DC 6C 57 09,41 56 49 20,4C 49 53 54)
38        ];
39        var typeName = [
40            'jpg',
41            'png',
42            'mp4',
43            'mp4',
44            'mp4',
```

```

45      'mp4',
46      'm4v',
47      'm4v',
48      'mov',
49      'mov',
50      'mov',
51      'ogg',
52      'ogg',
53      'avi',
54  ];
55  var sliceSize = /png|jpg|jpeg/.test(type) ? 3 : 12;
56  var reader = new FileReader();
57  reader.readAsArrayBuffer(file);
58  reader.addEventListener("load", function(e) {
59    var slice = e.target.result.slice(0, sliceSize);
60    reader = null;
61    if (slice && slice.byteLength == sliceSize) {
62      var view = new Uint8Array(slice);
63      var arr = [];
64      view.forEach(function(v) {
65        arr.push(v.toString(16));
66      });
67      view = null;
68      var idx = arr.join(' ').indexOf(imgType);
69      if (idx > -1) {
70        back(typeName[idx]);
71      } else {
72        arr = arr.map(function(v) {
73          if (i > 3 && i < 8) {
74            return 'x';
75          }
76          return v;
77        });
78        var idx = arr.join(' ').indexOf(imgType);
79        if (idx > -1) {
80          back(typeName[idx]);
81        } else {
82          back(false);
83        }
84      }
85    }
86  } else {
87    back(false);
88  }
89})
90});
91} else {
92  var type = file.name.match(/\.(\w+)$/) [1];

```

```
93         back(type);
94     }
95 }
```

调用方法如下

```
1 checkFileType('(mov|mp4|avi)', file, function(fileType){
2     // fileType = mp4,
3     // 如果file的类型不在枚举之列，则返回false
4 });
```

上面上传文件的一步，可以改成：

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

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

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

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

### 23.3. 使用场景

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

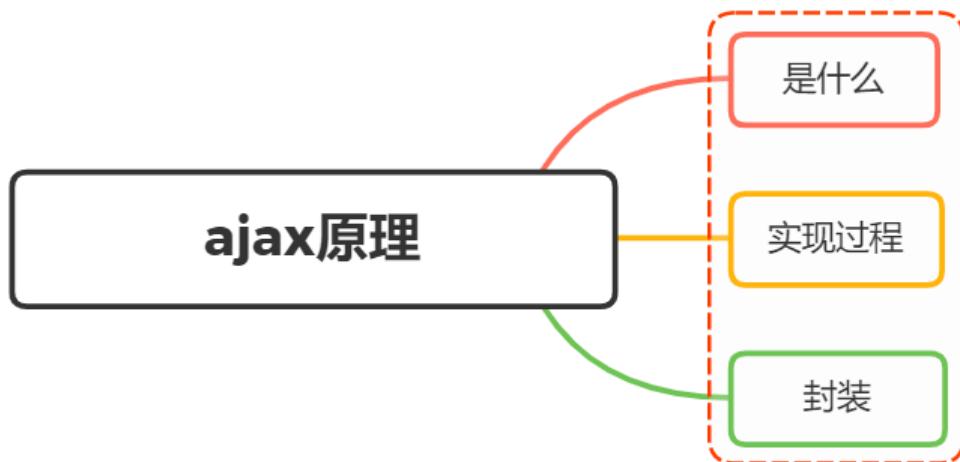
### 23.4. 小结

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

- 切片上传失败怎么办
- 上传过程中刷新页面怎么办

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

## 24. ajax原理是什么？如何实现？



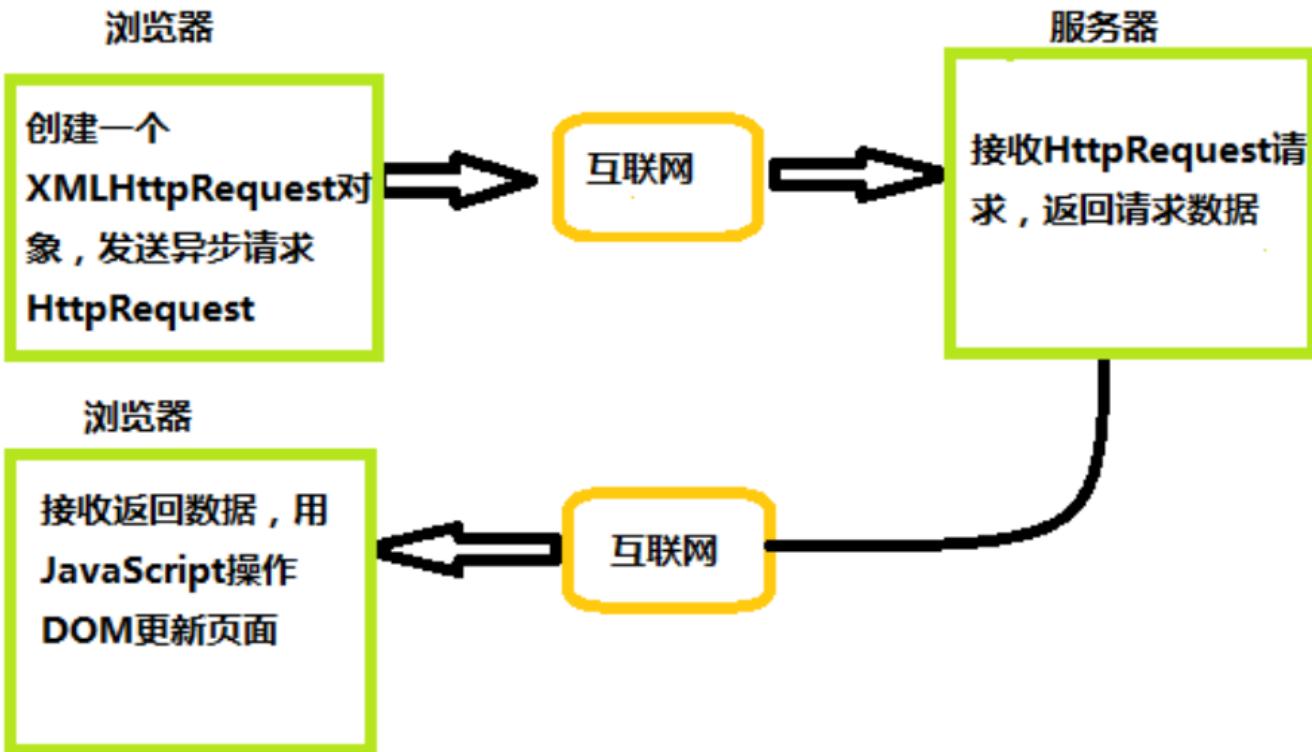
### 24.1. 是什么

AJAX 全称(Async Javascript and XML)

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

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

流程图如下：



下面举个例子：

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

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

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

## 24.2. 实现过程

实现 Ajax 异步交互需要服务器逻辑进行配合，需要完成以下步骤：

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

## 24.2.1. 创建 XMLHttpRequest 对象

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

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

JavaScript | 复制代码

## 24.2.2. 与服务器建立连接

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

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

JavaScript | 复制代码

参数说明：

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

## 24.2.3. 给服务端发送数据

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

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

JavaScript | 复制代码

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

如果使用 `GET` 请求发送数据的时候，需要注意如下：

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

## 24.2.4. 绑定 `onreadystatechange` 事件

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

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

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

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

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

举个例子：

```
▼ JavaScript | ⌂ 复制代码  
1 const request = new XMLHttpRequest()  
2 request.onreadystatechange = function(e){  
3   if(request.readyState === 4){ // 整个请求过程完毕  
4     if(request.status >= 200 && request.status <= 300){  
5       console.log(request.responseText) // 服务端返回的结果  
6     }else if(request.status >=400){  
7       console.log("错误信息: " + request.status)  
8     }  
9   }  
10 }  
11 request.open('POST','http://xxxx')  
12 request.send()
```

## 24.3. 封装

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

```
1 //封装一个ajax请求
2 function ajax(options) {
3     //创建XMLHttpRequest对象
4     const xhr = new XMLHttpRequest()
5
6
7     //初始化参数的内容
8     options = options || {}
9     options.type = (options.type || 'GET').toUpperCase()
10    options.dataType = options.dataType || 'json'
11    const params = options.data
12
13    //发送请求
14    if (options.type === 'GET') {
15        xhr.open('GET', options.url + '?' + params, true)
16        xhr.send(null)
17    } else if (options.type === 'POST') {
18        xhr.open('POST', options.url, true)
19        xhr.send(params)
20
21    //接收请求
22    xhr.onreadystatechange = function () {
23        if (xhr.readyState === 4) {
24            let status = xhr.status
25            if (status >= 200 && status < 300) {
26                options.success && options.success(xhr.responseText, xhr.r
esponseXML)
27            } else {
28                options.fail && options.fail(status)
29            }
30        }
31    }
32 }
```

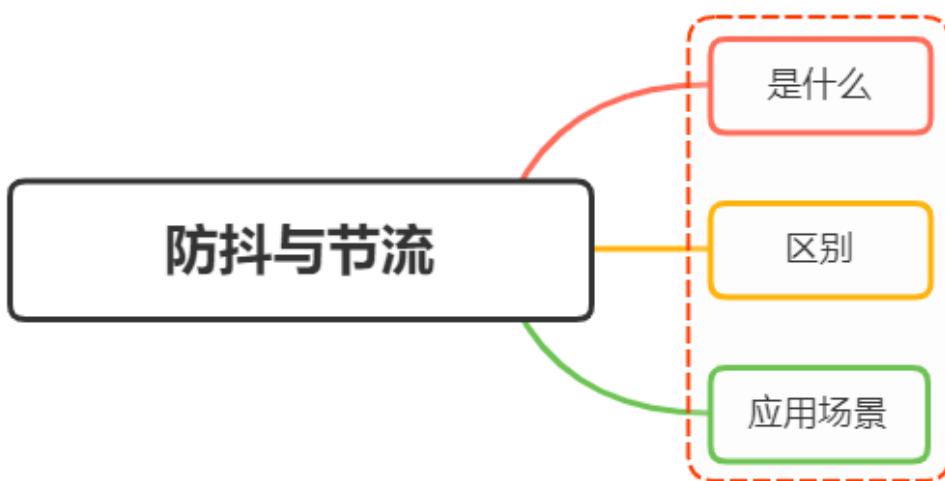
使用方式如下

```

1  ajax({
2      type: 'post',
3      dataType: 'json',
4      data: {},
5      url: 'https://xxxx',
6      success: function(text,xml){//请求成功后的回调函数
7          console.log(text)
8      },
9      fail: function(status){///请求失败后的回调函数
10         console.log(status)
11     }
12  })

```

## 25. 什么是防抖和节流？有什么区别？如何实现？



### 25.1. 是什么

本质上是优化高频率执行代码的一种手段

如：浏览器的 `resize`、`scroll`、`keypress`、`mousemove` 等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能

为了优化体验，需要对这类事件进行调用次数的限制，对此我们就可以采用 **防抖（debounce）** 和 **节流（throttle）** 的方式来减少调用频率

#### 25.1.1. 定义

- 节流: n 秒内只运行一次, 若在 n 秒内重复触发, 只有一次生效
- 防抖: n 秒后在执行该事件, 若在 n 秒内被重复触发, 则重新计时

一个经典的比喻:

想象每天上班大厦底下的电梯。把电梯完成一次运送, 类比为一次函数的执行和响应

假设电梯有两种运行策略 `debounce` 和 `throttle`, 超时设定为15秒, 不考虑容量限制

电梯第一个人进来后, 15秒后准时运送一次, 这是节流

电梯第一个人进来后, 等待15秒。如果过程中又有人进来, 15秒等待重新计时, 直到15秒后开始运送, 这是防抖

## 25.2. 代码实现

### 25.2.1. 节流

完成节流可以使用时间戳与定时器的写法

使用时间戳写法, 事件会立即执行, 停止触发后没有办法再次执行

```
▼ JavaScript | 复制代码

1  function throttled1(fn, delay = 500) {
2      let oldtime = Date.now()
3      return function (...args) {
4          let newtime = Date.now()
5          if (newtime - oldtime >= delay) {
6              fn.apply(null, args)
7              oldtime = Date.now()
8          }
9      }
10 }
```

使用定时器写法, `delay` 毫秒后第一次执行, 第二次事件停止触发后依然会再一次执行

JavaScript | 复制代码

```
1 function throttled2(fn, delay = 500) {
2     let timer = null
3     return function (...args) {
4         if (!timer) {
5             timer = setTimeout(() => {
6                 fn.apply(this, args)
7                 timer = null
8             }, delay);
9         }
10    }
11 }
```

可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下

JavaScript | 复制代码

```
1 function throttled(fn, delay) {
2     let timer = null
3     let starttime = Date.now()
4     return function () {
5         let curTime = Date.now() // 当前时间
6         let remaining = delay - (curTime - starttime) // 从上一次到现在，还
剩下多少多余时间
7         let context = this
8         let args = arguments
9         clearTimeout(timer)
10        if (remaining <= 0) {
11            fn.apply(context, args)
12            starttime = Date.now()
13        } else {
14            timer = setTimeout(fn, remaining);
15        }
16    }
17 }
```

## 25.2.2. 防抖

简单版本的实现

```

1  function debounce(func, wait) {
2      let timeout;
3
4      return function () {
5          let context = this; // 保存this指向
6          let args = arguments; // 拿到event对象
7
8          clearTimeout(timeout)
9          timeout = setTimeout(function(){
10              func.apply(context, args)
11          }, wait);
12      }
13  }

```

防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：

```

1  function debounce(func, wait, immediate) {
2
3      let timeout;
4
5      return function () {
6          let context = this;
7          let args = arguments;
8
9          if (timeout) clearTimeout(timeout); // timeout 不为null
10         if (immediate) {
11             let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发
12             timeout = setTimeout(function () {
13                 timeout = null;
14             }, wait)
15             if (callNow) {
16                 func.apply(context, args)
17             }
18         }
19         else {
20             timeout = setTimeout(function () {
21                 func.apply(context, args)
22             }, wait);
23         }
24     }
25 }

```

## 25.3. 区别

相同点：

- 都可以通过使用 `setTimeout` 实现
- 目的都是，降低回调执行频率。节省计算资源

不同点：

- 函数防抖，在一段连续操作结束后，处理回调，利用 `clearTimeout` 和 `setTimeout` 实现。函数节流，在一段连续操作中，每一段时间只执行一次，频率较高的事件中使用来提高性能
- 函数防抖关注一定时间连续触发的事件，只在最后执行一次，而函数节流一段时间内只执行一次

例如，都设置时间频率为500ms，在2秒时间内，频繁触发函数，节流，每隔 500ms 就执行一次。防抖，则不管调动多少次方法，在2s后，只会执行一次

如下图所示：

正常执行



函数防抖



函数节流



## 25.4. 应用场景

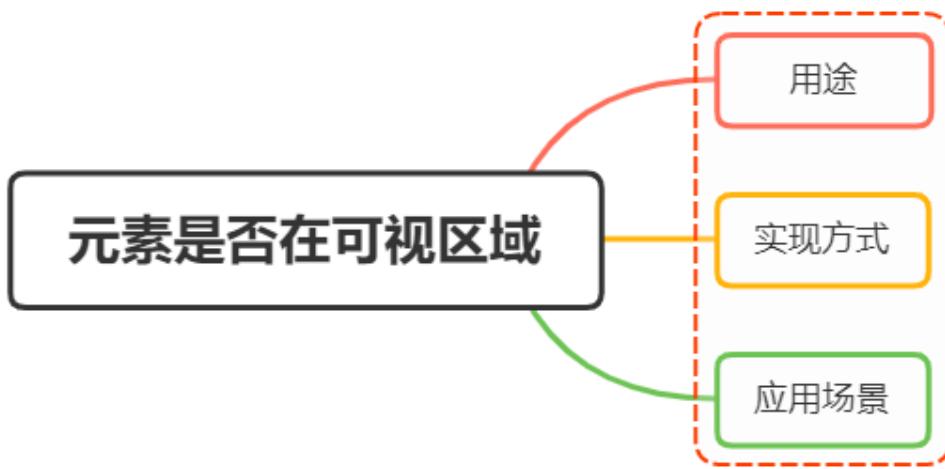
防抖在连续的事件，只需触发一次回调的场景有：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求
- 手机号、邮箱验证输入检测
- 窗口大小 `resize`。只需窗口调整完成后，计算窗口大小。防止重复渲染。

节流在间隔一段时间执行一次回调的场景有：

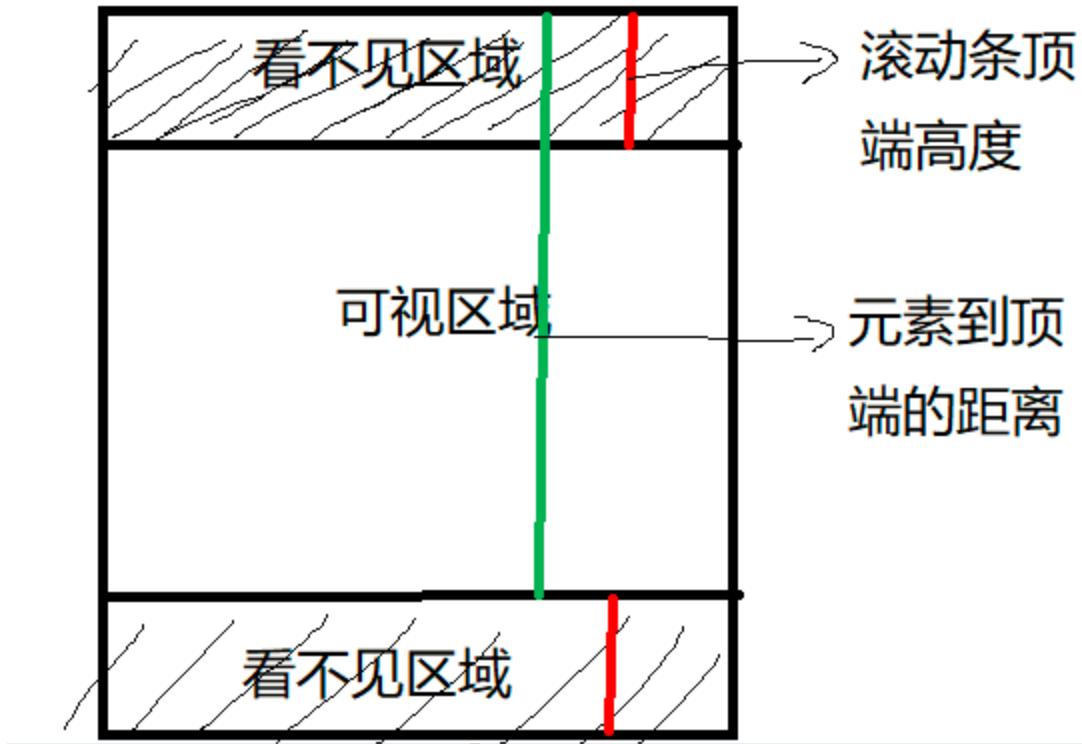
- 滚动加载，加载更多或滚到底部监听
- 搜索框，搜索联想功能

## 26. 如何判断一个元素是否在可视区域中？



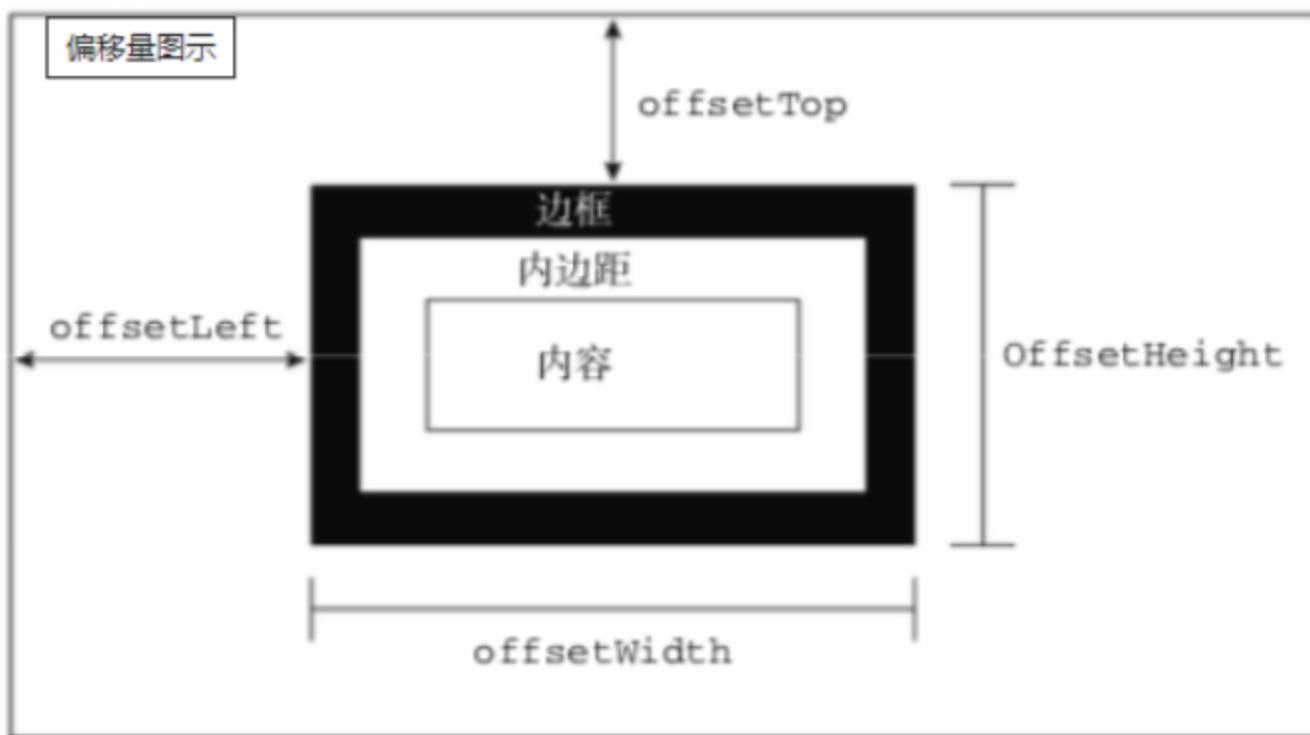
### 26.1. 用途

可视区域即我们浏览网页的设备肉眼可见的区域，如下图



`offsetTop`，元素的上外边框至包含元素的上内边框之间的像素距离，其他 `offset` 属性如下图所示：

### offsetParent



下面再来了解下 `clientWidth`、`clientHeight`：

- `clientWidth`：元素内容区宽度加上左右内边距宽度，即 `clientWidth = content + padding`
- `clientHeight`：元素内容区高度加上上下内边距高度，即 `clientHeight = content + padding`

这里可以看到 `client` 元素都不包括外边距

最后，关于 `scroll` 系列的属性如下：

- `scrollWidth` 和 `scrollHeight` 主要用于确定元素内容的实际大小
- `scrollLeft` 和 `scrollTop` 属性既可以确定元素当前滚动的状态，也可以设置元素的滚动位置
  - 垂直滚动 `scrollTop > 0`
  - 水平滚动 `scrollLeft > 0`
- 将元素的 `scrollLeft` 和 `scrollTop` 设置为 0，可以重置元素的滚动位置

#### 26.2.1.1. 注意

- 上述属性都是只读的，每次访问都要重新开始

下面再看看如何实现判断：

公式如下：

```
▼ JavaScript | 复制代码
1 el.offsetTop - document.documentElement.scrollTop <= viewPortHeight
```

代码实现：

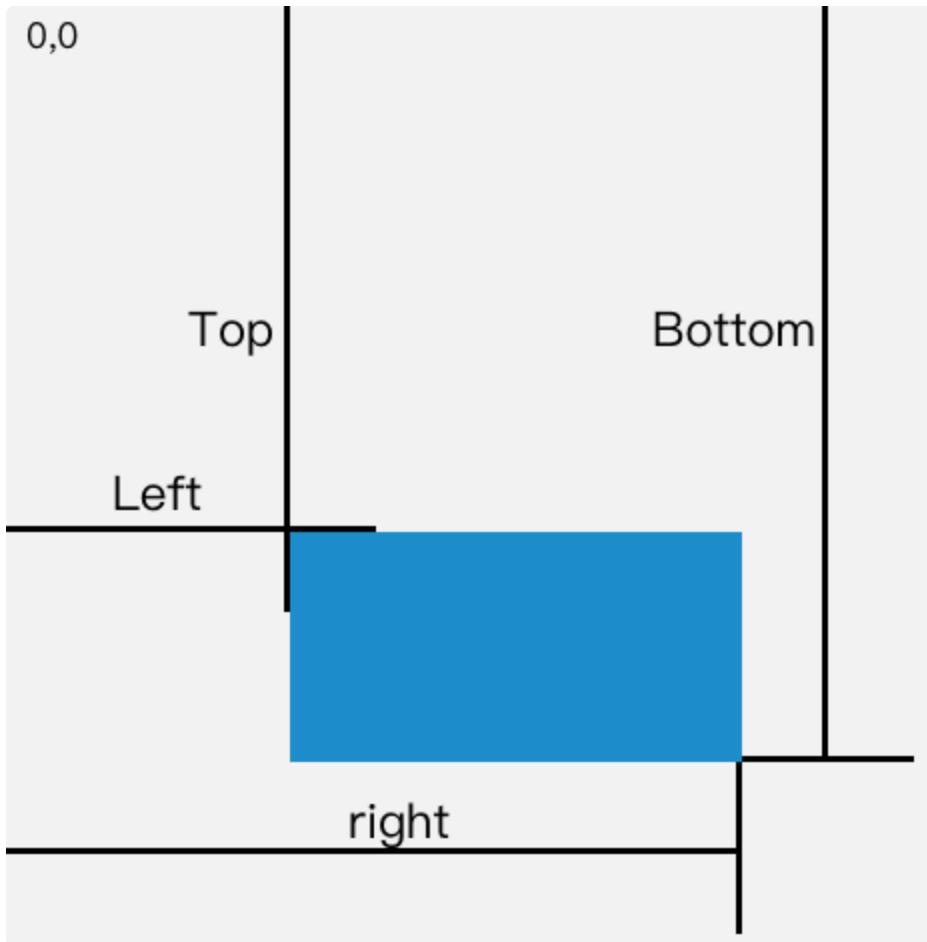
```
▼ JavaScript | 复制代码
1 ▾ function isInViewPortOfOne (el) {
2     // viewPortHeight 兼容所有浏览器写法
3     const viewPortHeight = window.innerHeight || document.documentElement.clientHeight || document.body.clientHeight
4     const offsetTop = el.offsetTop
5     const scrollTop = document.documentElement.scrollTop
6     const top = offsetTop - scrollTop
7     return top <= viewPortHeight
8 }
```

## 26.2.2. getBoundingClientRect

返回值是一个 `DOMRect` 对象，拥有 `left`，`top`，`right`，`bottom`，`x`，`y`，`width`，和 `height` 属性

```
▼ JavaScript | 复制代码
1 const target = document.querySelector('.target');
2 const clientRect = target.getBoundingClientRect();
3 console.log(clientRect);
4
5 // {
6 //   bottom: 556.21875,
7 //   height: 393.59375,
8 //   left: 333,
9 //   right: 1017,
10 //   top: 162.625,
11 //   width: 684
12 // }
```

属性对应的关系图如下所示：



当页面发生滚动的时候，`top` 与 `left` 属性值都会随之改变

如果一个元素在视窗之内的话，那么它一定满足下面四个条件：

- `top` 大于等于 0
- `left` 大于等于 0
- `bottom` 小于等于视窗高度
- `right` 小于等于视窗宽度

实现代码如下：

JavaScript | 复制代码

```
1 function isInViewPort(element) {
2     const viewWidth = window.innerWidth || document.documentElement.clientWidth;
3     const viewHeight = window.innerHeight || document.documentElement.clientHeight;
4     const {
5         top,
6         right,
7         bottom,
8         left,
9     } = element.getBoundingClientRect();
10
11    return (
12        top >= 0 &&
13        left >= 0 &&
14        right <= viewWidth &&
15        bottom <= viewHeight
16    );
17 }
```

### 26.2.3. Intersection Observer

`Intersection Observer` 即重叠观察者，从这个命名就可以看出它用于判断两个元素是否重叠，因为不用进行事件的监听，性能方面相比 `getBoundingClientRect` 会好很多

使用步骤主要分为两步：创建观察者和传入被观察者

#### 26.2.3.1. 创建观察者

JavaScript | 复制代码

```
1 const options = {
2     // 表示重叠面积占被观察者的比例，从 0 - 1 取值,
3     // 1 表示完全被包含
4     threshold: 1.0,
5     root:document.querySelector('#scrollArea') // 必须是目标元素的父级元素
6 };
7
8 const callback = (entries, observer) => { .... }
9
10 const observer = new IntersectionObserver(callback, options);
```

通过 `new IntersectionObserver` 创建了观察者 `observer`，传入的参数 `callback` 在重叠比例超过 `threshold` 时会被执行`

关于 `callback` 回调函数常用属性如下：

```
1 // 上段代码中被省略的 callback
2 const callback = function(entries, observer) {
3   entries.forEach(entry => {
4     entry.time;           // 触发的时间
5     entry.rootBounds;    // 根元素的位置矩形，这种情况下为视窗位置
6     entry.boundingClientRect; // 被观察者的位置矩形
7     entry.intersectionRect; // 重叠区域的位置矩形
8     entry.intersectionRatio; // 重叠区域占被观察者面积的比例（被观察者不是矩形时也按照矩形计算）
9     entry.target;         // 被观察者
10   });
11 }
```

JavaScript

复制代码

### 26.2.3.2. 传入被观察者

通过 `observer.observe(target)` 这一行代码即可简单的注册被观察者

```
1 const target = document.querySelector('.target');
2 observer.observe(target);
```

JavaScript

复制代码

### 26.2.4. 案例分析

实现：创建了一个十万个节点的长列表，当节点滚入到视窗中时，背景就会从红色变为黄色

Html 结构如下：

```
1 <div class="container"></div>
```

JavaScript

复制代码

css 样式如下：

CSS | 复制代码

```
1 .container {  
2     display: flex;  
3     flex-wrap: wrap;  
4 }  
5 .target {  
6     margin: 5px;  
7     width: 20px;  
8     height: 20px;  
9     background: red;  
10 }
```

往 `container` 插入1000个元素

JavaScript | 复制代码

```
1 const $container = $(".container");  
2  
3 // 插入 100000 个 <div class="target"></div>  
4 function createTargets() {  
5     const htmlString = new Array(100000)  
6         .fill('<div class="target"></div>')  
7         .join("");  
8     $container.html(htmlString);  
9 }
```

这里，首先使用 `getBoundingClientRect` 方法进行判断元素是否在可视区域

JavaScript | 复制代码

```
1 function isInViewport(element) {  
2     const viewWidth = window.innerWidth || document.documentElement.clientWidth;  
3     const viewHeight =  
4         window.innerHeight || document.documentElement.clientHeight;  
5     const { top, right, bottom, left } = element.getBoundingClientRect();  
6  
7     return top >= 0 && left >= 0 && right <= viewWidth && bottom <= viewHeight;  
8 }
```

然后开始监听 `scroll` 事件，判断页面上哪些元素在可视区域中，如果在可视区域中则将背景颜色设置为 `yellow`

JavaScript | 复制代码

```
1 $(window).on("scroll", () => {
2     console.log("scroll !");
3     $targets.each((index, element) => {
4         if (isInViewPort(element)) {
5             $(element).css("background-color", "yellow");
6         }
7     });
8 });
```

通过上述方式，可以看到可视区域颜色会变成黄色了，但是可以明显看到有卡顿的现象，原因在于我们绑定了 scroll 事件， scroll 事件伴随了大量的计算，会造成资源方面的浪费

下面通过 Intersection Observer 的形式同样实现相同的功能

首先创建一个观察者

JavaScript | 复制代码

```
1 const observer = new IntersectionObserver(getYellow, { threshold: 1.0 });
```

getYellow 回调函数实现对背景颜色改变，如下：

JavaScript | 复制代码

```
1 function getYellow(entries, observer) {
2     entries.forEach(entry => {
3         $(entry.target).css("background-color", "yellow");
4     });
5 }
```

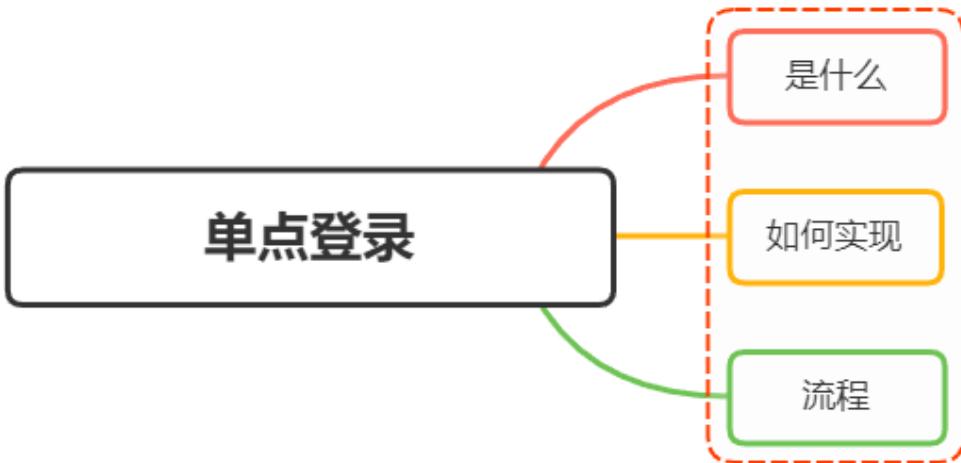
最后传入观察者，即 .target 元素

JavaScript | 复制代码

```
1 $targets.each((index, element) => {
2     observer.observe(element);
3 });
```

可以看到功能同样完成，并且页面不会出现卡顿的情况

## 27. 什么是单点登录？如何实现？



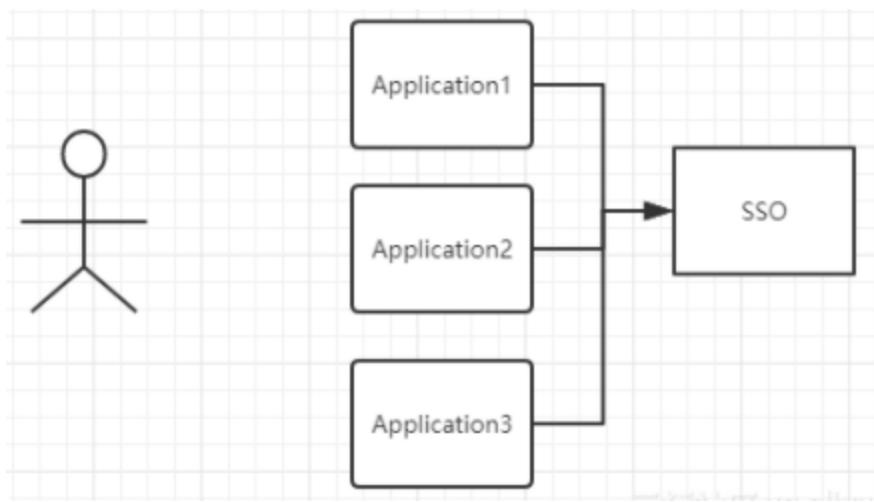
## 27.1. 是什么

单点登录 (Single Sign On) , 简称为 SSO, 是目前比较流行的企业业务整合的解决方案之一

SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统

SSO 一般都需要一个独立的认证中心 (passport) , 子系统的登录均得通过 `passport` , 子系统本身将不参与登录操作

当一个系统成功登录以后, `passport` 将会颁发一个令牌给各个子系统, 子系统可以拿着令牌会获取各自的受保护资源, 为了减少频繁认证, 各个子系统在被 `passport` 授权以后, 会建立一个局部会话, 在一定时间内可以无需再次向 `passport` 发起认证



上图有四个系统, 分别是 `Application1` 、 `Application2` 、 `Application3` 、和 `SSO` , 当 `Application1` 、 `Application2` 、 `Application3` 需要登录时, 将跳到 `SSO` 系统, `SSO` 系统完成登录, 其他的应用系统也就随之登录了

### 27.1.1. 举个例子

淘宝、天猫都属于阿里旗下，当用户登录淘宝后，再打开天猫，系统便自动帮用户登录了天猫，这种现象就属于单点登录

## 27.2. 如何实现

### 27.2.1. 同域名下的单点登录

`cookie` 的 `domain` 属性设置为当前域的父域，并且父域的 `cookie` 会被子域所共享。`path` 属性默认为 `web` 应用的上下文路径

利用 `Cookie` 的这个特点，没错，我们只需要将 `Cookie` 的 `domain` 属性设置为父域的域名（主域名），同时将 `Cookie` 的 `path` 属性设置为根路径，将 `Session ID`（或 `Token`）保存到父域中。这样所有的子域应用就都可以访问到这个 `Cookie`

不过这要求应用系统的域名需建立在一个共同的主域名之下，如 `tieba.baidu.com` 和 `map.baidu.com`，它们都建立在 `baidu.com` 这个主域名之下，那么它们就可以通过这种方式来实现单点登录

### 27.2.2. 不同域名下的单点登录(一)

如果是不同域的情况下，`Cookie` 是不共享的，这里我们可以部署一个认证中心，用于专门处理登录请求的独立的 `Web` 服务

用户统一在认证中心进行登录，登录成功后，认证中心记录用户的登录状态，并将 `token` 写入 `Cookie`（注意这个 `Cookie` 是认证中心的，应用系统是访问不到的）

应用系统检查当前请求有没有 `Token`，如果没有，说明用户在当前系统中尚未登录，那么就将页面跳转至认证中心

由于这个操作会将认证中心的 `Cookie` 自动带过去，因此，认证中心能够根据 `Cookie` 知道用户是否已经登录过了

如果认证中心发现用户尚未登录，则返回登录页面，等待用户登录

如果发现用户已经登录过了，就不会让用户再次登录了，而是会跳转回目标 `URL`，并在跳转前生成一个 `Token`，拼接在目标 `URL` 的后面，回传给目标应用系统

应用系统拿到 `Token` 之后，还需要向认证中心确认下 `Token` 的合法性，防止用户伪造。确认无误后，应用系统记录用户的登录状态，并将 `Token` 写入 `Cookie`，然后给本次访问放行。（注意这个 `Cookie` 是当前应用系统的）当用户再次访问当前应用系统时，就会自动带上这个 `Token`，应用系统验证 `Token` 发现用户已登录，于是就不会有认证中心什么事了

此种实现方式相对复杂，支持跨域，扩展性好，是单点登录的标准做法

### 27.2.3. 不同域名下的单点登录(二)

可以选择将 `Session ID` (或 `Token`) 保存到浏览器的 `LocalStorage` 中，让前端在每次向后端发送请求时，主动将 `LocalStorage` 的数据传递给服务端

这些都是由前端来控制的，后端需要做的仅仅是在用户登录成功后，将 `Session ID` (或 `Token`) 放在响应体中传递给前端

单点登录完全可以在前端实现。前端拿到 `Session ID` (或 `Token`) 后，除了将它写入自己的 `LocalStorage` 中之外，还可以通过特殊手段将它写入多个其他域下的 `LocalStorage` 中

关键代码如下：

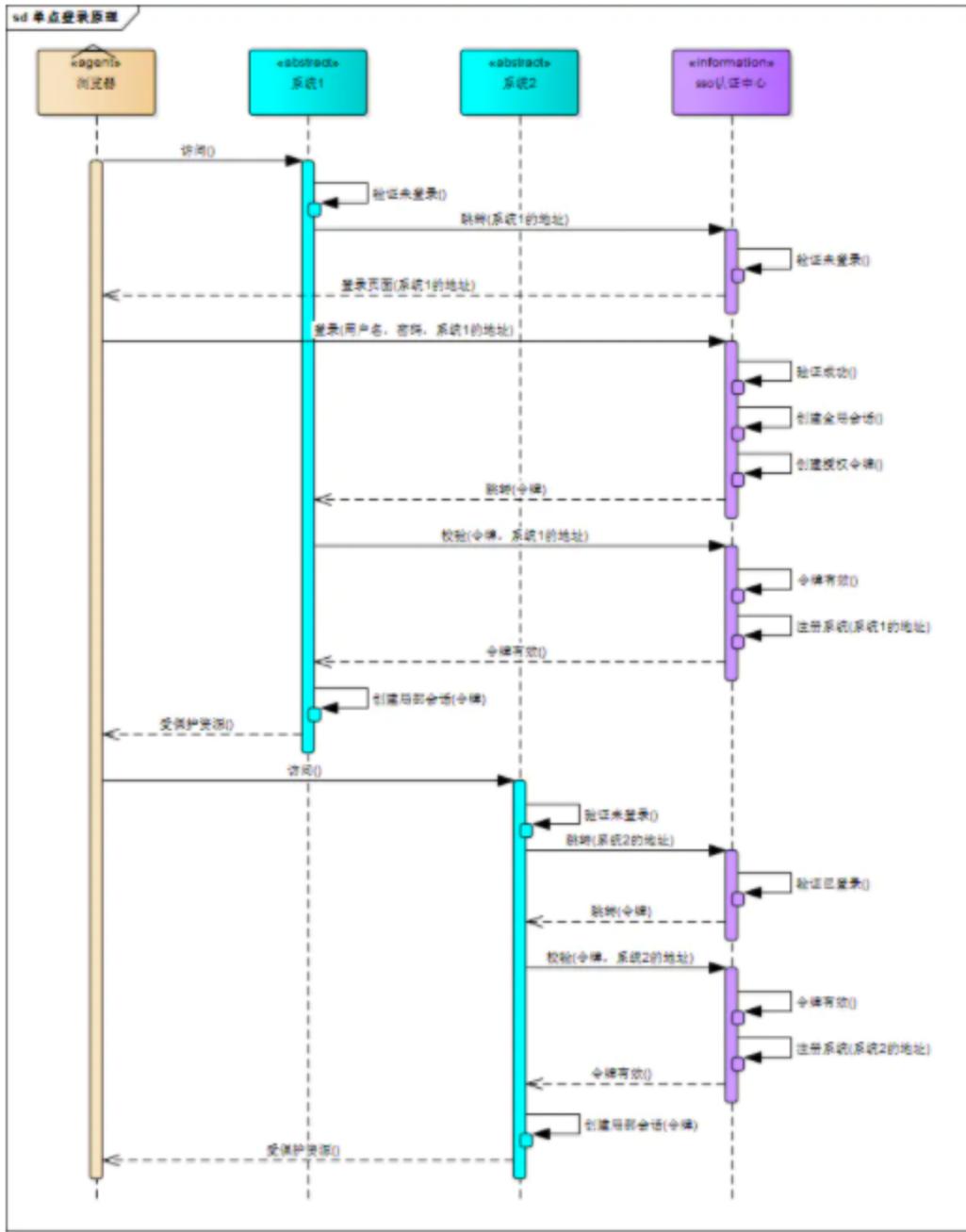
```
▼ JavaScript | 复制代码
1 // 获取 token
2 var token = result.data.token;
3
4 // 动态创建一个不可见的iframe，在iframe中加载一个跨域HTML
5 var iframe = document.createElement("iframe");
6 iframe.src = "http://app1.com/localstorage.html";
7 document.body.append(iframe);
8 // 使用postMessage()方法将token传递给iframe
9 setTimeout(function () {
10     iframe.contentWindow.postMessage(token, "http://app1.com");
11 }, 4000);
12 setTimeout(function () {
13     iframe.remove();
14 }, 6000);
15
16 // 在这个iframe所加载的HTML中绑定一个事件监听器，当事件被触发时，把接收到的token数据
17 // 写入localStorage
18 window.addEventListener('message', function (event) {
19     localStorage.setItem('token', event.data)
}, false);
```

前端通过 `iframe + postMessage()` 方式，将同一份 `Token` 写入到了多个域下的 `LocalStorage` 中，前端每次在向后端发送请求之前，都会主动从 `LocalStorage` 中读取 `Token` 并在请求中携带，这样就实现了同一份 `Token` 被多个域所共享

此种实现方式完全由前端控制，几乎不需要后端参与，同样支持跨域

## 27.3. 流程

单点登录的流程图如下所示：



- 用户访问系统1的受保护资源，系统1发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
- sso认证中心发现用户未登录，将用户引导至登录页面
- 用户输入用户名密码提交登录申请
- sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌
- sso认证中心带着令牌跳转会最初的请求地址（系统1）
- 系统1拿到令牌，去sso认证中心校验令牌是否有效
- sso认证中心校验令牌，返回有效，注册系统1
- 系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源

- 用户访问系统2的受保护资源
- 系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
- sso认证中心发现用户已登录，跳转回系统2的地址，并附上令牌
- 系统2拿到令牌，去sso认证中心校验令牌是否有效
- sso认证中心校验令牌，返回有效，注册系统2
- 系统2使用该令牌创建与用户的局部会话，返回受保护资源

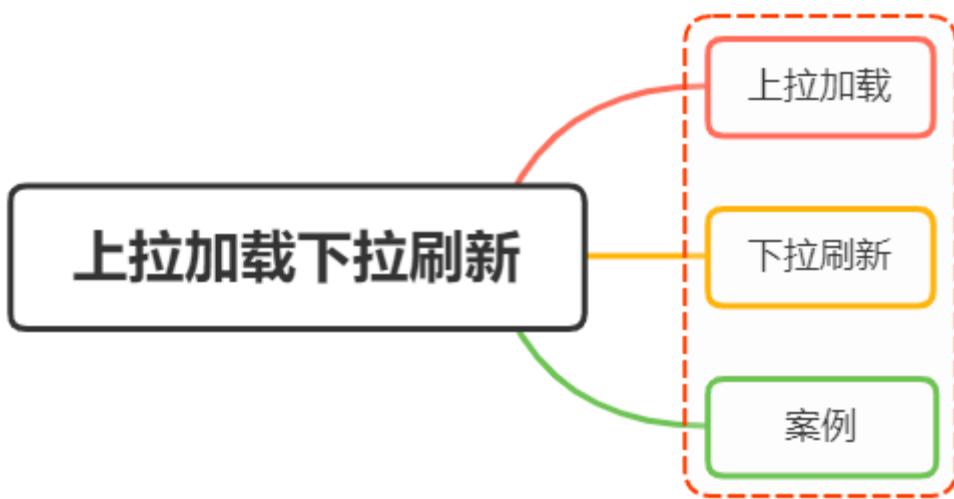
用户登录成功之后，会与 `sso` 认证中心及各个子系统建立会话，用户与 `sso` 认证中心建立的会话称为全局会话

用户与各个子系统建立的会话称为局部会话，局部会话建立之后，用户访问子系统受保护资源将不再通过 `sso` 认证中心

全局会话与局部会话有如下约束关系：

- 局部会话存在，全局会话一定存在
- 全局会话存在，局部会话不一定存在
- 全局会话销毁，局部会话必须销毁

## 28. 如何实现上拉加载，下拉刷新？



### 28.1. 前言

下拉刷新和上拉加载这两种交互方式通常出现在移动端中

本质上等同于PC网页中的分页，只是交互形式不同

开源社区也有很多优秀的解决方案，如 `iscroll`、`better-scroll`、`pulltorefresh.js` 库等等

这些第三方库使用起来非常便捷

我们通过原生的方式实现一次上拉加载，下拉刷新，有助于对第三方库有更好的理解与使用

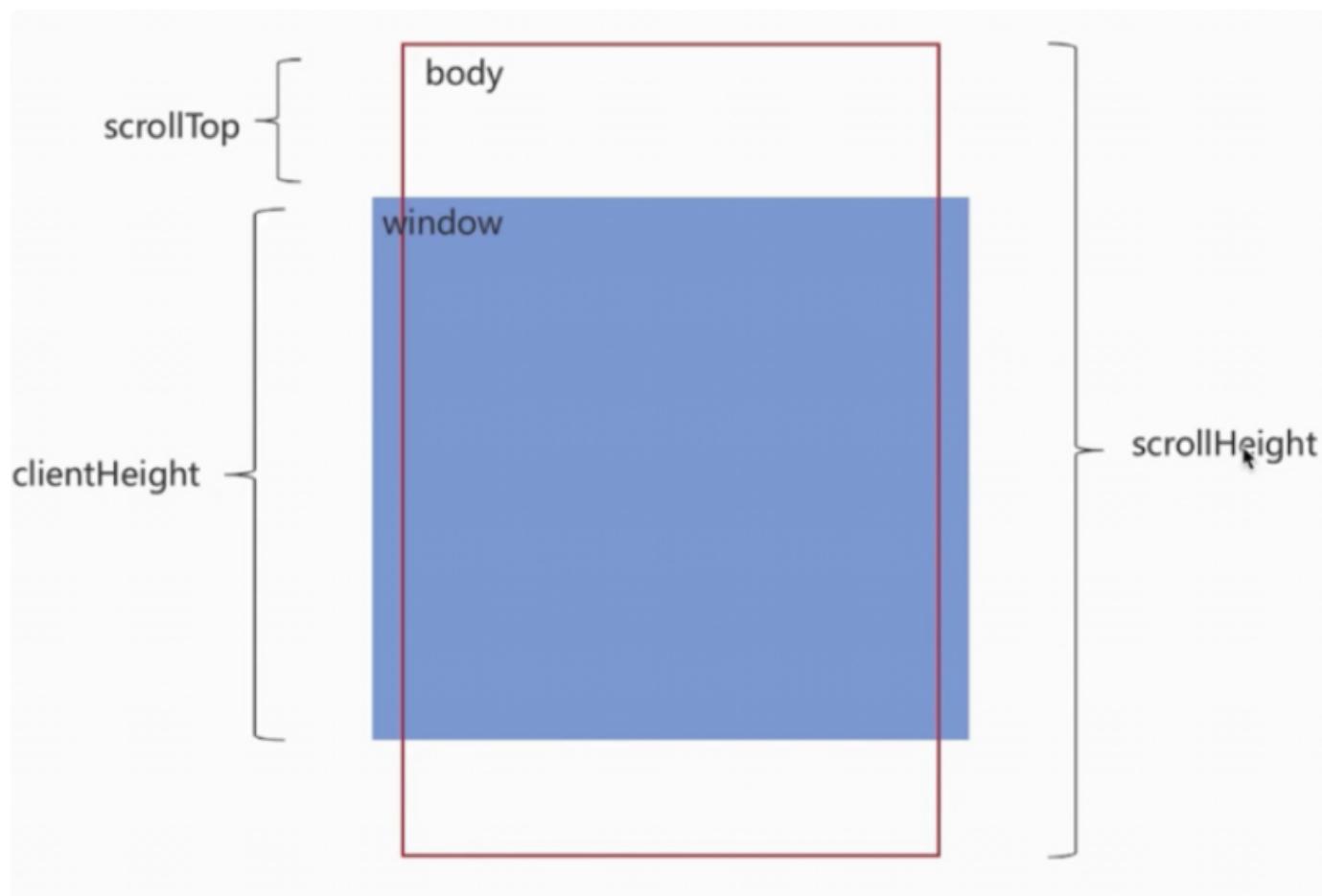
## 28.2. 实现原理

上拉加载及下拉刷新都依赖于用户交互

最重要的是要理解在什么场景，什么时机下触发交互动作

### 28.2.1. 上拉加载

首先可以看一张图



上拉加载的本质是页面触底，或者快要触底时的动作

判断页面触底我们需要先了解一下下面几个属性

- `scrollTop`：滚动视窗的高度距离 `window` 顶部的距离，它会随着往上滚动而不断增加，初始

值是0，它是一个变化的值

- `clientHeight` : 它是一个定值，表示屏幕可视区域的高度；
- `scrollHeight` : 页面不能滚动时也是存在的,此时`scrollHeight`等于`clientHeight`。`scrollHeight` 表示 `body` 所有元素的总长度(包括`body`元素自身的padding)

综上我们得出一个触底公式：

```
▼ JavaScript | 复制代码
1 scrollTop + clientHeight >= scrollHeight
```

简单实现

```
▼ JavaScript | 复制代码
1 let clientHeight = document.documentElement.clientHeight; //浏览器高度
2 let scrollHeight = document.body.scrollHeight;
3 let scrollTop = document.documentElement.scrollTop;
4
5 let distance = 50; //距离视窗还用50的时候，开始触发；
6
7 if ((scrollTop + clientHeight) >= (scrollHeight - distance)) {
8     console.log("开始加载数据");
9 }
```

## 28.2.2. 下拉刷新

下拉刷新的本质是页面本身置于顶部时，用户下拉时需要触发的动作

关于下拉刷新的原生实现，主要分成三步：

- 监听原生 `touchstart` 事件，记录其初始位置的值，`e.touches[0].pageY`；
- 监听原生 `touchmove` 事件，记录并计算当前滑动的位置值与初始位置值的差值，大于 `0` 表示向下拉动，并借助CSS3的 `translateY` 属性使元素跟随手势向下滑动对应的差值，同时也应设置一个允许滑动的最大值；
- 监听原生 `touchend` 事件，若此时元素滑动达到最大值，则触发 `callback`，同时将 `translateY` 重设为 `0`，元素回到初始位置

举个例子：

`Html` 结构如下：

JavaScript | 复制代码

```
1 <main>
2     <p class="refreshText"></p >
3     <ul id="refreshContainer">
4         <li>111</li>
5         <li>222</li>
6         <li>333</li>
7         <li>444</li>
8         <li>555</li>
9         ...
10    </ul>
11 </main>
```

监听 `touchstart` 事件，记录初始的值

JavaScript | 复制代码

```
1 var _element = document.getElementById('refreshContainer'),
2     _refreshText = document.querySelector('.refreshText'),
3     _startPos = 0, // 初始的值
4     _transitionHeight = 0; // 移动的距离
5
6 _element.addEventListener('touchstart', function(e) {
7     _startPos = e.touches[0].pageY; // 记录初始位置
8     _element.style.position = 'relative';
9     _element.style.transition = 'transform 0s';
10}, false);
```

监听 `touchmove` 移动事件，记录滑动差值

JavaScript | 复制代码

```
1 _element.addEventListener('touchmove', function(e) {
2     // e.touches[0].pageY 当前位置
3     _transitionHeight = e.touches[0].pageY - _startPos; // 记录差值
4
5 if (_transitionHeight > 0 && _transitionHeight < 60) {
6     _refreshText.innerText = '下拉刷新';
7     _element.style.transform = 'translateY('+_transitionHeight+'px)';
8
9 if (_transitionHeight > 55) {
10     _refreshText.innerText = '释放更新';
11 }
12 }
13}, false);
```

最后，就是监听 `touchend` 离开的事件

```
1 _element.addEventListener('touchend', function(e) {  
2     _element.style.transition = 'transform 0.5s ease 1s';  
3     _element.style.transform = 'translateY(0px)';  
4     _refreshText.innerText = '更新中...';  
5     // todo...  
6  
7 }, false);
```

从上面可以看到，在下拉到松手的过程中，经历了三个阶段：

- 当前手势滑动位置与初始位置差值大于零时，提示正在进行下拉刷新操作
- 下拉到一定值时，显示松手释放后的操作提示
- 下拉到达设定最大值松手时，执行回调，提示正在进行更新操作

## 28.3. 案例

在实际开发中，我们更多的是使用第三方库，下面以 `better-scroll` 进行举例：

HTML结构

```
1 <div id="position-wrapper">  
2     <div>  
3         <p class="refresh">下拉刷新</p>  
4         <div class="position-list">  
5             <!--列表内容-->  
6         </div>  
7         <p class="more">查看更多</p>  
8     </div>  
9 </div>
```

实例化上拉下拉插件，通过 `use` 来注册插件

▼ JavaScript | 复制代码

```
1 import BScroll from "@better-scroll/core";
2 import PullDown from "@better-scroll/pull-down";
3 import PullUp from '@better-scroll/pull-up';
4 BScroll.use(PullDown);
5 BScroll.use(PullUp);
```

实例化 `BetterScroll`，并传入相关的参数

```
1 let pageNo = 1, pageSize = 10, dataList = [], isMore = true;
2 var scroll = new BScroll("#position-wrapper", {
3     scrollY: true, // 垂直方向滚动
4     click: true, // 默认会阻止浏览器的原生click事件，如果需要点击，这里要设为true
5     pullUpLoad: true, // 上拉加载更多
6     pullDownRefresh: {
7         threshold: 50, // 触发pullingDown事件的位置
8         stop: 0 // 下拉回弹后停留的位置
9     }
10 });
11 // 监听下拉刷新
12 scroll.on("pullingDown", pullingDownHandler);
13 // 监测实时滚动
14 scroll.on("scroll", scrollHandler);
15 // 上拉加载更多
16 scroll.on("pullingUp", pullingUpHandler);
17
18 async function pullingDownHandler(){
19     dataList = [];
20     pageNo = 1;
21     isMore = true;
22     $(".more").text("查看更多");
23     await getlist(); // 请求数据
24     scroll.finishPullDown(); // 每次下拉结束后，需要执行这个操作
25     scroll.refresh(); // 当滚动区域的dom结构有变化时，需要执行这个操作
26 }
27 async function pullingUpHandler(){
28     if(!isMore){
29         $(".more").text("没有更多数据了");
30         scroll.finishPullUp(); // 每次上拉结束后，需要执行这个操作
31         return;
32     }
33     pageNo++;
34     await this.getlist(); // 请求数据
35     scroll.finishPullUp(); // 每次上拉结束后，需要执行这个操作
36     scroll.refresh(); // 当滚动区域的dom结构有变化时，需要执行这个操作
37 }
38 function scrollHandler(){
39     if(this.y > 50) $('.refresh').text("松手开始加载");
40     else $('.refresh').text("下拉刷新");
41 }
42 function getlist(){
43     // 返回的数据
44     let result = ....;
45     dataList = dataList.concat(result);
```

```
46     //判断是否已加载完
47     if(result.length<pageSize) isMore=false;
48     //将dataList渲染到html内容中
49 }
```

注意点：

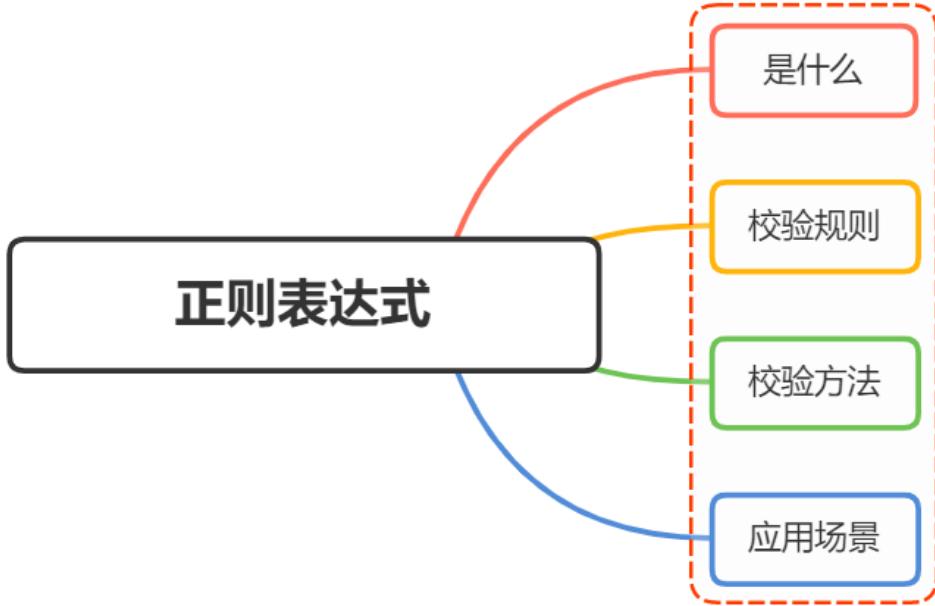
使用 `better-scroll` 实现下拉刷新、上拉加载时要注意以下几点：

- `wrapper` 里必须只有一个子元素
- 子元素的高度要比 `wrapper` 要高
- 使用的时候，要确定 `DOM` 元素是否已经生成，必须要等到 `DOM` 渲染完成后，再 `new BScroll()`
- 滚动区域的 `DOM` 元素结构有变化后，需要执行刷新 `refresh()`
- 上拉或者下拉，结束后，需要执行 `finishPullUp()` 或者 `finishPullDown()`，否则将不会执行下次操作
- `better-scroll`，默认会阻止浏览器的原生 `click` 事件，如果滚动内容区要添加点击事件，需要在实例化属性里设置 `click:true`

### 28.3.1. 小结

下拉刷新、上拉加载原理本身都很简单，真正复杂的是封装过程中，要考虑的兼容性、易用性、性能等诸多细节

## 29. 说说你对正则表达式的理解？应用场景？



## 29.1. 是什么

正则表达式是一种用来匹配字符串的强有力的武器

它的设计思想是用一种描述性的语言定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的

在 `JavaScript` 中，正则表达式也是对象，构建正则表达式有两种方式：

1. 字面量创建，其由包含在斜杠之间的模式组成

```

▼
JavaScript | 复制代码
1 const re = /\d+/g;

```

2. 调用 `RegExp` 对象的构造函数

```

▼
JavaScript | 复制代码
1 const re = new RegExp("\d+","g");
2
3 const rul = "\d+"
4 const re1 = new RegExp(rul,"g");

```

使用构建函数创建，第一个参数可以是一个变量，遇到特殊字符 `\` 需要使用 `\\` 进行转义

## 29.2. 匹配规则

常见的校验规则如下：

规则	描述
\	转义
^	匹配输入的开始
\$	匹配输入的结束
*	匹配前一个表达式 0 次或多次
+	匹配前面一个表达式 1 次或者多次。等价于 <code>{1,}</code>
?	匹配前面一个表达式 0 次或者 1 次。等价于 <code>{0,1}</code>
.	默认匹配除换行符之外的任何单个字符
x(?=y)	匹配'x'仅仅当'x'后面跟着'y'。这种叫做先行断言
(?<=y)x	匹配'x'仅当'x'前面是'y'.这种叫做后行断言
x(?:!y)	仅仅当'x'后面不跟着'y'时匹配'x'，这被称为正向否定查找
(?<!y)x	仅仅当'x'前面不是'y'时匹配'x'，这被称为反向否定查找
x y	匹配‘x’或者‘y’
{n}	n 是一个正整数，匹配了前面一个字符刚好出现了 n 次
{n,}	n是一个正整数，匹配前一个字符至少出现了n次
{n,m}	n 和 m 都是整数。匹配前面的字符至少n次，最多m次
[xyz]	一个字符集合。匹配方括号中的任意字符
[^xyz]	匹配任何没有包含在方括号中的字符
\b	匹配一个词的边界，例如在字母和空格之间
\B	匹配一个非单词边界

\d	匹配一个数字
\D	匹配一个非数字字符
\f	匹配一个换页符
\n	匹配一个换行符
\r	匹配一个回车符
\s	匹配一个空白字符，包括空格、制表符、换页符和换行符
\S	匹配一个非空白字符
\w	匹配一个单字字符（字母、数字或者下划线）
\W	匹配一个非单字字符

### 29.2.1. 正则表达式标记

标志	描述
g	全局搜索。
i	不区分大小写搜索。
m	多行搜索。
s	允许 . 匹配换行符。
u	使用 unicode 码的模式进行匹配。
y	执行“粘性(sticky)”搜索，匹配从目标字符串的当前位置开始。

使用方法如下：

JavaScript | 复制代码

```
1 var re = /pattern	flags;
2 var re = new RegExp("pattern", "flags");
```

在了解下正则表达式基本的之外，还可以掌握几个正则表达式的特性：

### 29.2.2. 贪婪模式

在了解贪婪模式前，首先举个例子：

JavaScript | 复制代码

```
1 const reg = /ab{1,3}c/
```

在匹配过程中，尝试可能的顺序是从多往少的方向去尝试。首先会尝试 `bbb`，然后再看整个正则是否能匹配。不能匹配时，吐出一个 `b`，即在 `bb` 的基础上，再继续尝试，以此重复

如果多个贪婪量词挨着，则深度优先搜索

JavaScript | 复制代码

```
1 const string = "12345";
2 const regex = /(\d{1,3})(\d{1,3})/;
3 console.log( string.match(regex) );
4 // => ["12345", "123", "45", index: 0, input: "12345"]
```

其中，前面的 `\d{1,3}` 匹配的是"123"，后面的 `\d{1,3}` 匹配的是"45"

### 29.2.3. 懒惰模式

惰性量词就是在贪婪量词后面加个问号。表示尽可能少的匹配

JavaScript | 复制代码

```
1 var string = "12345";
2 var regex = /(\d{1,3}?) (\d{1,3})/;
3 console.log( string.match(regex) );
4 // => ["1234", "1", "234", index: 0, input: "12345"]
```

其中 `\d{1,3}?` 只匹配到一个字符"1"，而后面的 `\d{1,3}` 匹配了"234"

### 29.2.4. 分组

分组主要是用过 `()` 进行实现，比如 `beyond{3}`，是匹配 `d` 字母3次。而 `(beyond){3}` 是匹配 `beyond` 三次

在 `()` 内使用 `|` 达到或的效果，如 `(abc | xxx)` 可以匹配 `abc` 或者 `xxx`

反向引用，巧用 `$` 分组捕获

```
▼ JavaScript | 复制代码

1 let str = "John Smith";
2
3 // 交换名字和姓氏
4 console.log(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

## 29.3. 匹配方法

正则表达式常被用于某些方法，我们可以分成两类：

- 字符串 (str) 方法： `match`、`matchAll`、`search`、`replace`、`split`
- 正则对象下 (regexp) 的方法： `test`、`exec`

方法	描述
<code>exec</code>	一个在字符串中执行查找匹配的RegExp方法，它返回一个数组（未匹配到则返回 <code>null</code> ）。
<code>test</code>	一个在字符串中测试是否匹配的RegExp方法，它返回 <code>true</code> 或 <code>false</code> 。
<code>match</code>	一个在字符串中执行查找匹配的String方法，它返回一个数组，在未匹配到时会返回 <code>null</code> 。
<code>matchAll</code>	一个在字符串中执行查找所有匹配的String方法，它返回一个迭代器（iterator）。
<code>search</code>	一个在字符串中测试匹配的String方法，它返回匹配到的位置索引，或者在失败时返回 <code>-1</code> 。
<code>replace</code>	一个在字符串中执行查找匹配的String方法，并且使用替换字符串替换掉匹配到的子字符串。

split

一个使用正则表达式或者一个固定字符串分隔一个字符串，并将分隔后的子字符串存储到数组中的 `String` 方法。

### 29.3.1. str.match(regexp)

`str.match(regexp)` 方法在字符串 `str` 中找到匹配 `regexp` 的字符。如果 `regexp` 不带有 `g` 标记，则它以数组的形式返回第一个匹配项，其中包含分组和属性 `index`（匹配项的位置）、`input`（输入字符串，等于 `str`）

```
1 let str = "I love JavaScript";
2
3 let result = str.match(/Java(Script)/);
4
5 console.log(result[0]); // JavaScript (完全匹配)
6 console.log(result[1]); // Script (第一个分组)
7 console.log(result.length); // 2
8
9 // 其他信息:
10 console.log(result.index); // 7 (匹配位置)
11 console.log(result.input); // I love JavaScript (源字符串)
```

如果 `regexp` 带有 `g` 标记，则它将所有匹配项的数组作为字符串返回，而不包含分组和其他详细信息。

```
1 let str = "I love JavaScript";
2
3 let result = str.match(/Java(Script)/g);
4
5 console.log(result[0]); // JavaScript
6 console.log(result.length); // 1
```

如果没有匹配项，则无论是否带有标记 `g`，都将返回 `null`

JavaScript | 复制代码

```
1 let str = "I love JavaScript";
2
3 let result = str.match(/HTML/);
4
5 console.log(result); // null
```

### 29.3.2. str.matchAll(regexp)

返回一个包含所有匹配正则表达式的结果及分组捕获组的迭代器

JavaScript | 复制代码

```
1 const regexp = /t(e)(st(\d?))/g;
2 const str = 'test1test2';
3
4 const array = [...str.matchAll(regexp)];
5
6 console.log(array[0]);
7 // expected output: Array ["test1", "e", "st1", "1"]
8
9 console.log(array[1]);
10 // expected output: Array ["test2", "e", "st2", "2"]
```

### 29.3.3. str.search(regexp)

返回第一个匹配项的位置，如果未找到，则返回 -1

JavaScript | 复制代码

```
1 let str = "A drop of ink may make a million think";
2
3 console.log( str.search( /ink/i ) ); // 10 (第一个匹配位置)
```

这里需要注意的是，`search` 仅查找第一个匹配项

## 29.4. str.replace(regexp)

替换与正则表达式匹配的子串，并返回替换后的字符串。在不设置全局匹配 `g` 的时候，只替换第一个匹配成功的字符串片段

JavaScript | 复制代码

```
1 const reg1=/javascript/i;
2 const reg2=/javascript/ig;
3 console.log('hello Javascript Javascript Javascript'.replace(reg1,'js'));
4 //hello js Javascript Javascript
5 console.log('hello Javascript Javascript Javascript'.replace(reg2,'js'));
6 //hello js js js
```

## 29.4.1. str.split(regexp)

使用正则表达式（或子字符串）作为分隔符来分割字符串

JavaScript | 复制代码

```
1 console.log('12, 34, 56'.split(/,\s*/)) // 数组 ['12', '34', '56']
```

## 29.4.2. regexp.exec(str)

`regexp.exec(str)` 方法返回字符串 `str` 中的 `regexp` 匹配项，与以前的方法不同，它是在正则表达式而不是字符串上调用的

根据正则表达式是否带有标志 `g`，它的行为有所不同

如果没有 `g`，那么 `regexp.exec(str)` 返回的第一个匹配与 `str.match(regexp)` 完全相同

如果有标记 `g`，调用 `regexp.exec(str)` 会返回第一个匹配项，并将紧随其后的位置保存在属性 `regexp.lastIndex` 中。下一次同样的调用会从位置 `regexp.lastIndex` 开始搜索，返回下一个匹配项，并将其后的位置保存在 `regexp.lastIndex` 中

JavaScript | 复制代码

```
1 let str = 'More about JavaScript at https://javascript.info';
2 let regexp = /javascript/ig;
3
4 let result;
5
6 while (result = regexp.exec(str)) {
7   console.log(`Found ${result[0]} at position ${result.index}`);
8   // Found JavaScript at position 11
9   // Found javascript at position 33
10 }
```

### 29.4.3. regexp.test(str)

查找匹配项，然后返回 `true/false` 表示是否存在

```
1 let str = "I love JavaScript";
2
3 // 这两个测试相同
4 console.log( /love/i.test(str) ); // true
```

JavaScript | 复制代码

## 29.5. 应用场景

通过上面的学习，我们对正则表达式有了一定的了解

下面再来看看正则表达式一些案例场景：

验证QQ合法性（5~15位、全是数字、不以0开头）：

```
1 const reg = /^[1-9][0-9]{4,14}$/;
2 const isvalid = patrn.exec(s)
```

JavaScript | 复制代码

校验用户账号合法性（只能输入5~20个以字母开头、可带数字、“\_”、“.”的字符串）：

```
1 var patrn=/[a-zA-Z]{1}([a-zA-Z0-9]|[_]){4,19}$/;
2 const isvalid = patrn.exec(s)
```

JavaScript | 复制代码

将 `url` 参数解析为对象

```
1 const protocol = '(?<protocol>https?:)';
2 const host = '(?<host>(?<hostname>[^/#?:]+)(?::(?<port>\d+))?)';
3 const path = '(?<pathname>(?:\/\/[^/#?]+)*\/\/?)';
4 const search = '(?<search>(?:\/\/[^#]*))';
5 const hash = '(?<hash>(?:#.*)?)';
6 const reg = new RegExp(`^${protocol}\/\/${host}${path}${search}${hash}$`);
7 function execURL(url){
8     const result = reg.exec(url);
9     if(result){
10         result.groups.port = result.groups.port || '';
11         return result.groups;
12     }
13     return {
14         protocol:'',
15         host:'',
16         hostname:'',
17         port:'',
18         pathname:'',
19         search:'',
20         hash:'',
21     };
22 }
23 console.log(execURL('https://localhost:8080/?a=b#xxxx'));
24 protocol: "https:"
25 host: "localhost:8080"
26 hostname: "localhost"
27 port: "8080"
28 pathname: "/"
29 search: "?a=b"
30 hash: "#xxxx"
```

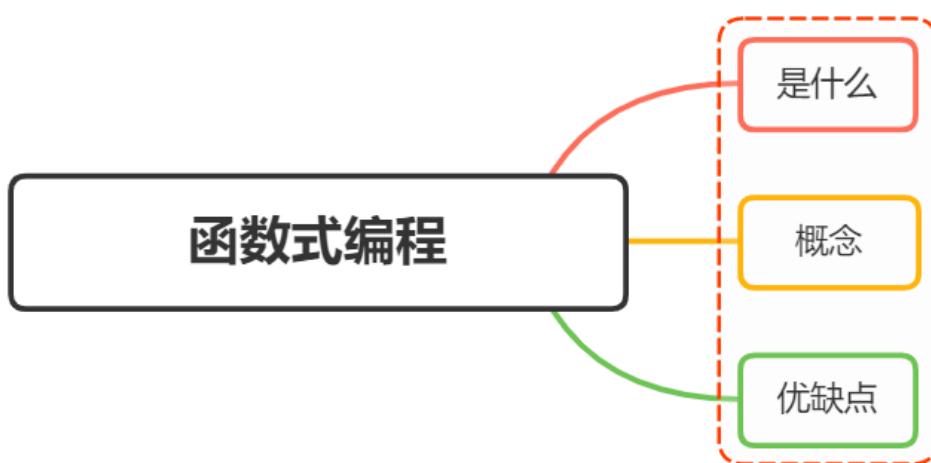
再将上面的 `search` 和 `hash` 进行解析

```

1  function execUrlParams(str){
2      str = str.replace(/^#?&/, '');
3      const result = {};
4      if(!str){ //如果正则可能配到空字符串，极有可能造成死循环，判断很重要
5          return result;
6      }
7      const reg = /(?:^|&)([^&=]*?)=?([^\&]*?)(?=|=|$)/y
8      let exec = reg.exec(str);
9      while(exec){
10         result[exec[1]] = exec[2];
11         exec = reg.exec(str);
12     }
13     return result;
14 }
15 console.log(execUrlParams('#'));// {}
16 console.log(execUrlParams('##'));//{'#':''}
17 console.log(execUrlParams('?q=3606&src=srp'));//{q: "3606", src: "srp"}
18 console.log(execUrlParams('test=a=b=c&&==&a='));//{test: "a=b=c", "":
"=", a: ""}

```

## 30. 说说你对函数式编程的理解？优缺点？



### 30.1. 是什么

函数式编程是一种"编程范式" (programming paradigm)，一种编写程序的方法论

主要的编程范式有三种：命令式编程，声明式编程和函数式编程

相比命令式编程，函数式编程更加强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而非设计一个复杂的执行过程

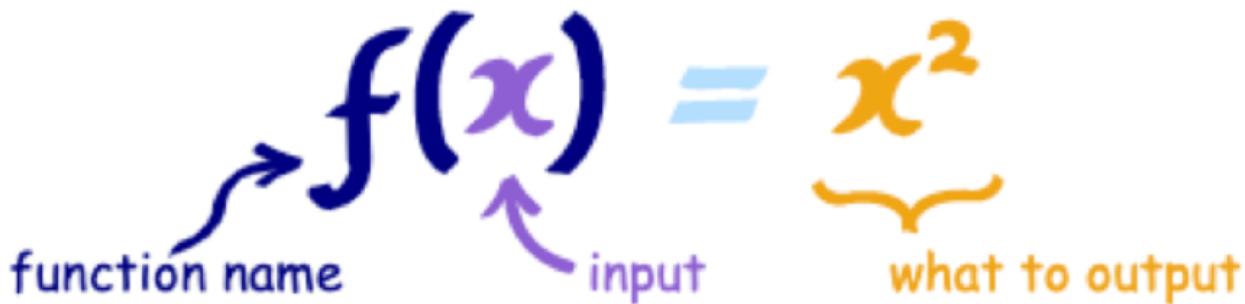
举个例子，将数组每个元素进行平方操作，命令式编程与函数式编程如下

```
1 // 命令式编程
2 var array = [0, 1, 2, 3]
3 for(let i = 0; i < array.length; i++) {
4     array[i] = Math.pow(array[i], 2)
5 }
6
7 // 函数式方式
8 [0, 1, 2, 3].map(num => Math.pow(num, 2))
```

JavaScript | 复制代码

简单来讲，就是要把过程逻辑写成函数，定义好输入参数，只关心它的输出结果

即是一种描述集合和集合之间的转换关系，输入通过函数都会返回有且只有一个输出值



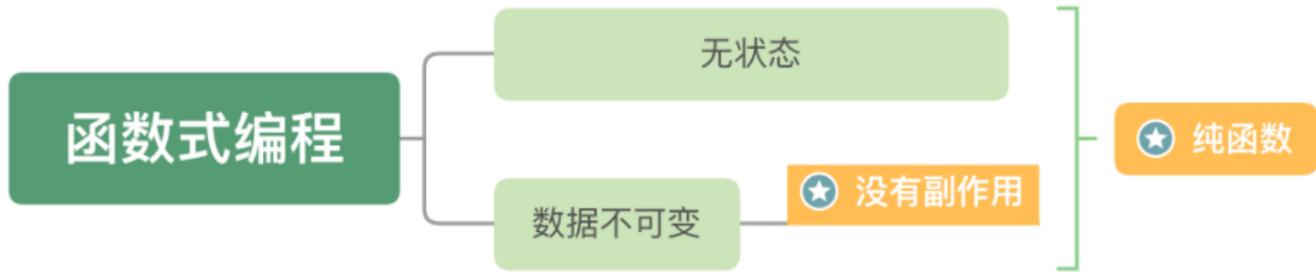
可以看到，函数实际上是一个关系，或者说是一种映射，而这种映射关系是可以组合的，一旦我们知道一个函数的输出类型可以匹配另一个函数的输入，那他们就可以进行组合

## 30.2. 概念

### 30.2.1. 纯函数

函数式编程旨在尽可能的提高代码的无状态性和不变性。要做到这一点，就要学会使用无副作用的函数，也就是纯函数

纯函数是对给定的输入返还相同输出的函数，并且要求你所有的数据都是不可变的，即纯函数=无状态+数据不可变



举一个简单的例子

```

▼
JavaScript | 复制代码

1 let double = value=>value*2;

```

特性：

- 函数内部传入指定的值，就会返回确定唯一的值
- 不会造成超出作用域的变化，例如修改全局变量或引用传递的参数

优势：

- 使用纯函数，我们可以产生可测试的代码

```

▼
JavaScript | 复制代码

1 ▶ test('double(2) 等于 4', () => {
2     expect(double(2)).toBe(4);
3 })

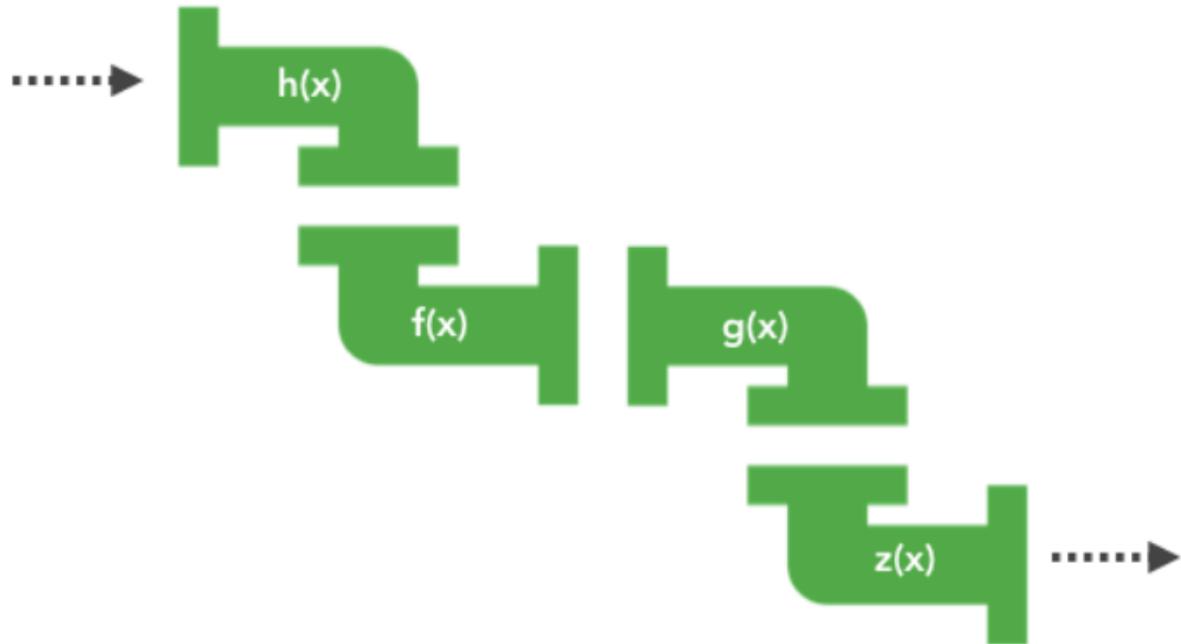
```

- 不依赖外部环境计算，不会产生副作用，提高函数的复用性
- 可读性更强，函数不管是否是纯函数 都会有一个语义化的名称，更便于阅读
- 可以组装成复杂任务的可能性。符合模块化概念及单一职责原则

### 30.2.2. 高阶函数

在我们的编程世界中，我们需要处理的其实也只有“数据”和“关系”，而关系就是函数

编程工作也就是在找一种映射关系，一旦关系找到了，问题就解决了，剩下的事情，就是让数据流过这种关系，然后转换成另一个数据，如下图所示



在这里，就是高阶函数的作用。高级函数，就是以函数作为输入或者输出的函数被称为高阶函数

通过高阶函数抽象过程，注重结果，如下面例子

```

1 const forEach = function(arr, fn){
2   for(let i=0; i<arr.length; i++){
3     fn(arr[i]);
4   }
5 }
6 let arr = [1,2,3];
7 forEach(arr, (item)=>{
8   console.log(item);
9 })

```

JavaScript | 复制代码

上面通过高阶函数 `forEach` 来抽象循环如何做的逻辑，直接关注做了什么

高阶函数存在缓存的特性，主要是利用闭包作用

JavaScript | 复制代码

```
1 const once = (fn)=>{
2     let done = false;
3     return function(){
4         if(!done){
5             fn.apply(this,fn);
6         }else{
7             console.log("该函数已经执行");
8         }
9         done = true;
10    }
11 }
```

### 30.2.3. 柯里化

柯里化是把一个多参数函数转化成一个嵌套的一元函数的过程

一个二元函数如下：

JavaScript | 复制代码

```
1 let fn = (x,y)=>x+y;
```

转化成柯里化函数如下：

JavaScript | 复制代码

```
1 const curry = function(fn){
2     return function(x){
3         return function(y){
4             return fn(x,y);
5         }
6     }
7 }
8 let myfn = curry(fn);
9 console.log( myfn(1)(2) );
```

上面的 `curry` 函数只能处理二元情况，下面再来实现一个实现多参数的情况

JavaScript | 复制代码

```
1 // 多参数柯里化;
2 const curry = function(fn){
3   return function curriedFn(...args){
4     if(args.length<fn.length){
5       return function(){
6         return curriedFn(...args.concat([...arguments]));
7       }
8     }
9     return fn(...args);
10  }
11 }
12 const fn = (x,y,z,a)=>x+y+z+a;
13 const myfn = curry(fn);
14 console.log(myfn(1)(2)(3)(1));
```

关于柯里化函数的意义如下：

- 让纯函数更纯，每次接受一个参数，松散解耦
- 惰性执行

### 30.2.4. 组合与管道

组合函数，目的是将多个函数组合成一个函数

举个简单的例子：

JavaScript | 复制代码

```
1 function afn(a){
2   return a*2;
3 }
4 function bfn(b){
5   return b*3;
6 }
7 const compose = (a,b)=>c=>a(b(c));
8 let myfn = compose(afn,bfn);
9 console.log( myfn(2));
```

可以看到 `compose` 实现一个简单的功能：形成了一个新的函数，而这个函数就是一条从 `bfn -> afn` 的流水线

下面再来看看如何实现一个多函数组合：

JavaScript | 复制代码

```
1 const compose = (...fns)=>val=>fns.reverse().reduce((acc,fn)=>fn(acc),val);
```

`compose` 执行是从右到左的。而管道函数，执行顺序是从左到右执行的

JavaScript | 复制代码

```
1 const pipe = (...fns)=>val=>fns.reduce((acc,fn)=>fn(acc),val);
```

组合函数与管道函数的意义在于：可以把很多小函数组合起来完成更复杂的逻辑

## 30.3. 优缺点

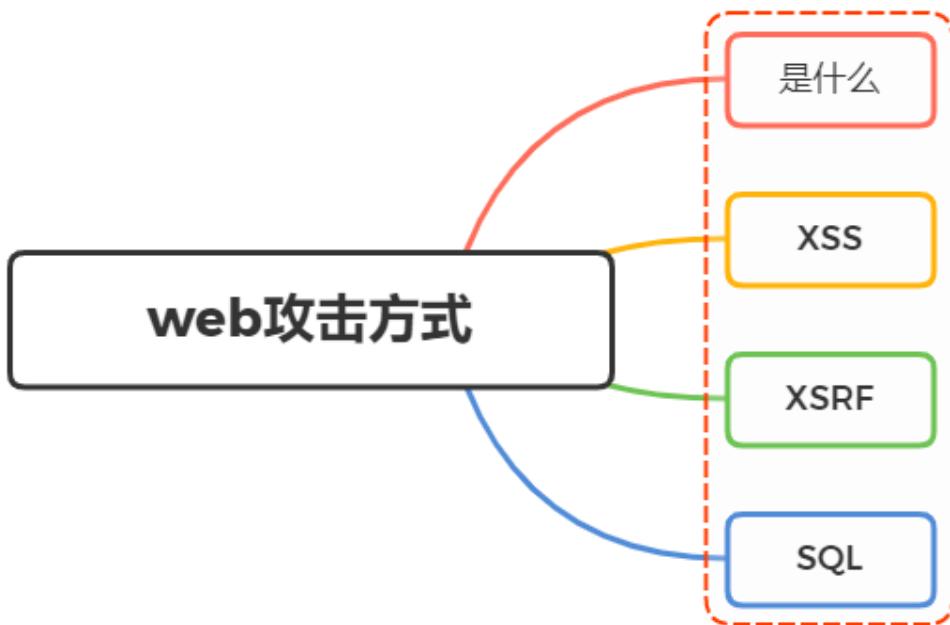
### 30.3.1. 优点

- 更好的管理状态：因为它的宗旨是无状态，或者说更少的状态，能最大化的减少这些未知、优化代码、减少出错情况
- 更简单的复用：固定输入->固定输出，没有其他外部变量影响，并且无副作用。这样代码复用时，完全不需要考虑它的内部实现和外部影响
- 更优雅的组合：往大的说，网页是由各个组件组成的。往小的说，一个函数也可能是由多个小函数组成的。更强的复用性，带来更强大的组合性
- 隐性好处。减少代码量，提高维护性

### 30.3.2. 缺点

- 性能：函数式编程相对于指令式编程，性能绝对是一个短板，因为它往往会对一个方法进行过度包装，从而产生上下文切换的性能开销
- 资源占用：在 JS 中为了实现对象状态的不可变，往往会创建新的对象，因此，它对垃圾回收所产生的压力远远超过其他编程方式
- 递归陷阱：在函数式编程中，为了实现迭代，通常会采用递归操作

## 31. web常见的攻击方式有哪些？如何防御？



## 31.1. 是什么

Web攻击 (WebAttack) 是针对用户上网行为或网站服务器等设备进行攻击的行为

如植入恶意代码，修改网站权限，获取网站用户隐私信息等等

Web应用程序的安全性是任何基于Web业务的重要组成部分

确保Web应用程序安全十分重要，即使是代码中很小的 bug 也有可能导致隐私信息被泄露

站点安全就是为保护站点不受未授权的访问、使用、修改和破坏而采取的行为或实践

我们常见的Web攻击方式有

- XSS (Cross Site Scripting) 跨站脚本攻击
- CSRF (Cross-site request forgery) 跨站请求伪造
- SQL注入攻击

## 31.2. XSS

XSS，跨站脚本攻击，允许攻击者将恶意代码植入到提供给其它用户使用的页面中

XSS 涉及到三方，即攻击者、客户端与 Web 应用

XSS 的攻击目标是为了盗取存储在客户端的 cookie 或者其他网站用于识别客户端身份的敏感信息。一旦获取到合法用户的信息后，攻击者甚至可以假冒合法用户与网站进行交互

举个例子：

一个搜索页面，根据 `url` 参数决定关键词的内容

HTML | 复制代码

```
1 <input type="text" value="<% getParameter("keyword") %>">
2 <button>搜索</button>
3 <div>
4 您搜索的关键词是: <%= getParameter("keyword") %>
5 </div>
```

这里看似并没有问题，但是如果按套路出牌呢？

用户输入 `"><script>alert('XSS');</script>"`，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

HTML | 复制代码

```
1 <input type="text" value=""><script>alert('XSS');</script>">
2 <button>搜索</button>
3 <div>
4 您搜索的关键词是: "><script>alert('XSS');</script>
5 </div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行，试想一下，如果是获取 `cookie` 发送对黑客服务器呢？

根据攻击的来源，`XSS` 攻击可以分成：

- 存储型
- 反射型
- DOM 型

### 31.2.1. 存储型

存储型 `XSS` 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等

### 31.2.2. 反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见

### 31.2.3. DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞

### 31.2.4. XSS的预防

通过前面介绍，看到 XSS 攻击的两大要素：

- 攻击者提交而恶意代码
- 浏览器执行恶意代码

针对第一个要素，我们在用户输入的过程中，过滤掉用户输入的恶劣代码，然后提交给后端，但是如果攻击者绕开前端请求，直接构造请求就不能预防了

而如果在后端写入数据库前，对输入进行过滤，然后把内容给前端，但是这个内容在不同地方就会有不同显示

例如：

一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`

在客户端中，一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)

在前端中，不同的位置所需的编码也不同。

- 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：



```
1 <div title="comment">5 &lt; 7</div>
```

The screenshot shows a code editor window with a single line of HTML code: <div title="comment">5 &lt; 7</div>. The code is displayed in a monospaced font. At the top right of the editor, there is a 'HTML' button and a 'Copy code' button.

- 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、`alert` 等

可以看到，过滤并非可靠的，下面就要通过防止浏览器执行恶意代码：

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等

如果用 `Vue/React` 技术栈，并且不使用 `v-html` / `dangerouslySetInnerHTML` 功能，就在前端 `render` 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseove` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免

```

1  <!-- 链接内包含恶意代码 -->
2  < a href="" "1</ a>
3
4  <script>
5  // setTimeout()/setInterval() 中调用恶意代码
6  setTimeout("UNTRUSTED")
7  setInterval("UNTRUSTED")
8
9  // location 调用恶意代码
10 location.href = 'UNTRUSTED'
11
12 // eval() 中调用恶意代码
13 eval("UNTRUSTED")

```

### 31.3. CSRF

CSRF (Cross-site request forgery) 跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求

利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的

一个典型的CSRF攻击有着如下的流程：

- 受害者登录a.com，并保留了登录凭证（Cookie）
- 攻击者引诱受害者访问了b.com
- b.com 向 a.com 发送了一个请求：a.com/act=xx。浏览器会默认携带a.com的Cookie
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
- a.com以受害者的名义执行了act=xx
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

`csrf` 可以通过 `get` 请求，即通过访问 `img` 的页面后，浏览器自动访问目标地址，发送请求

同样，也可以设置一个自动提交的表单发送 `post` 请求，如下：

JavaScript | 复制代码

```
1 <form action="http://bank.example/withdraw" method=POST>
2     <input type="hidden" name="account" value="xiaoming" />
3     <input type="hidden" name="amount" value="10000" />
4     <input type="hidden" name="for" value="hacker" />
5 </form>
6 <script> document.forms[0].submit(); </script>
```

访问该页面后，表单会自动提交，相当于模拟用户完成了一次 POST 操作

还有一种为使用 `a` 标签的，需要用户点击链接才会触发

访问该页面后，表单会自动提交，相当于模拟用户完成了一次POST操作

HTML | 复制代码

```
1 < a href="http://test.com/csrf/withdraw.php?amount=1000&for=hacker" target=
  "_blank">
2     重磅消息！ !
3 <a/>
```

### 31.3.1. CSRF的特点

- 攻击一般发起在第三方网站，而不是被攻击的网站。被攻击的网站无法防止攻击发生
- 攻击利用受害者在被攻击网站的登录凭证，冒充受害者提交操作；而不是直接窃取数据
- 整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”
- 跨站请求可以用各种方式：图片URL、超链接、CORS、Form提交等等。部分请求方式可以直接嵌入在第三方论坛、文章中，难以进行追踪

### 31.3.2. CSRF的预防

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性

防止 `csrf` 常用方案如下：

- 阻止不明外域的访问
  - 同源检测
  - Samesite Cookie
- 提交时要求附加本域才能获取的信息

- CSRF Token
- 双重Cookie验证

这里主要讲讲 `token` 这种形式，流程如下：

- 用户打开页面的时候，服务器需要给这个用户生成一个Token
- 对于GET请求，Token将附在请求地址之后。对于 POST 请求来说，要在 form 的最后加上

HTML | 复制代码

```
1 <input type="hidden" name="csrfmiddlewaretoken" value="tokenvalue"/>
```

- 当用户从客户端得到了Token，再次提交给服务器的时候，服务器需要判断Token的有效性

## 31.4. SQL注入

Sql 注入攻击，是通过将恶意的 `Sql` 查询或添加语句插入到应用的输入参数中，再在后台 `Sql` 服务器上解析执行进行的攻击



流程如下所示：

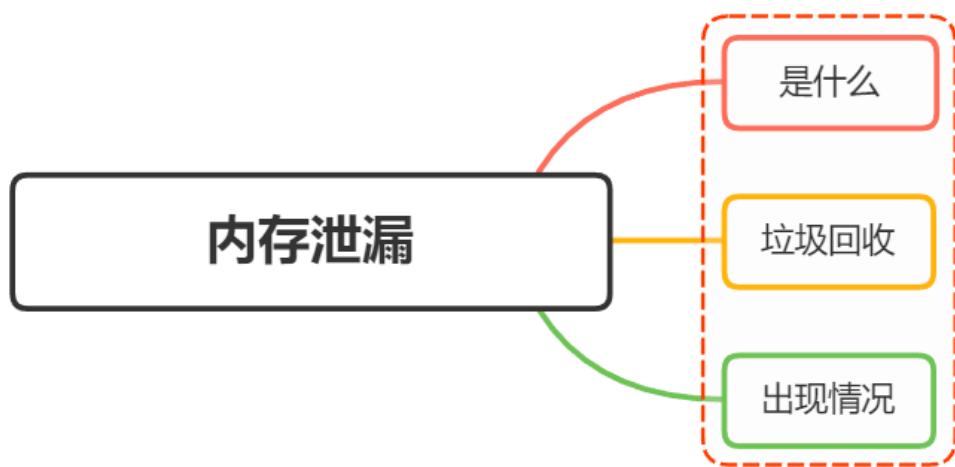
- 找出SQL漏洞的注入点
- 判断数据库的类型以及版本
- 猜解用户名和密码
- 利用工具查找Web后台管理入口
- 入侵和破坏

预防方式如下：

- 严格检查输入变量的类型和格式
- 过滤和转义特殊字符
- 对访问数据库的Web应用程序采用Web应用防火墙

上述只是列举了常见的 web 攻击方式，实际开发过程中还会遇到很多安全问题，对于这些问题，切记不可忽视

## 32. 说说 JavaScript 中内存泄漏的几种情况？

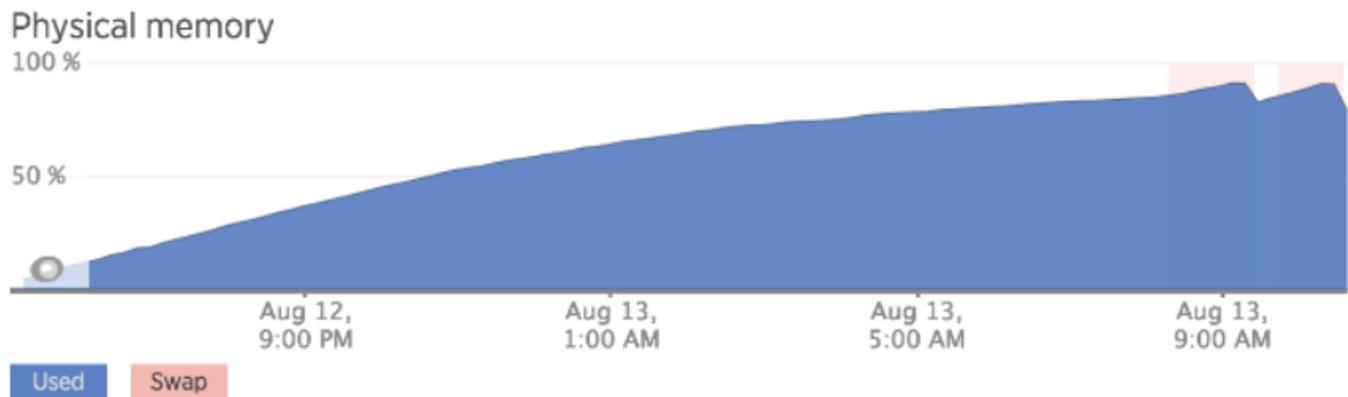


### 32.1. 是什么

内存泄漏（Memory leak）是在计算机科学中，由于疏忽或错误造成程序未能释放已经不再使用的内存，并非指内存物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费

程序的运行需要内存。只要程序提出要求，操作系统或者运行时就必须供给内存

对于持续运行的服务进程，必须及时释放不再用到的内存。否则，内存占用越来越高，轻则影响系统性能，重则导致进程崩溃



在 C 语言中，因为是手动管理内存，内存泄露是经常出现的事情。

```

1 char * buffer;
2 buffer = (char*) malloc(42);
3
4 // Do something with buffer
5
6 free(buffer);

```

上面是 C 语言代码，`malloc` 方法用来申请内存，使用完毕之后，必须自己用 `free` 方法释放内存。这很麻烦，所以大多数语言提供自动内存管理，减轻程序员的负担，这被称为"垃圾回收机制"

## 32.2. 垃圾回收机制

Javascript 具有自动垃圾回收机制 (GC: Garbage Collection)，也就是说，执行环境会负责管理代码执行过程中使用的内存

原理：垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存

通常情况下有两种实现方式：

- 标记清除
- 引用计数

### 32.2.1. 标记清除

JavaScript 最常用的垃圾回收机制

当变量进入执行环境时，就标记这个变量为“进入环境”。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境”

垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉

在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了

随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回它们的内存

举个例子：

```
1 var m = 0, n = 19 // 把 m,n,add() 标记为进入环境。
2 add(m, n) // 把 a, b, c标记为进入环境。
3 console.log(n) // a,b,c标记为离开环境，等待垃圾回收。
4 function add(a, b) {
5     a++
6     var c = a + b
7     return c
8 }
```

JavaScript | 复制代码

### 32.2.2. 引用计数

语言引擎有一张“引用表”，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是 0，就表示这个值不再用到了，因此可以将这块内存释放

如果一个值不再需要了，引用数却不为 0，垃圾回收机制无法释放这块内存，从而导致内存泄漏

```
1 const arr = [1, 2, 3, 4];
2 console.log('hello world');
```

JavaScript | 复制代码

上面代码中，数组 [1, 2, 3, 4] 是一个值，会占用内存。变量 arr 是仅有的对这个值的引用，因此引用次数为 1。尽管后面的代码没有用到 arr，它还是会持续占用内存

如果需要这块内存被垃圾回收机制释放，只需要设置如下：

```
1 arr = null
```

JavaScript | 复制代码

通过设置 arr 为 null，就解除了对数组 [1,2,3,4] 的引用，引用次数变为 0，就被垃圾回收了

### 32.2.3. 小结

有了垃圾回收机制，不代表不用关注内存泄露。那些很占空间的值，一旦不再用到，需要检查是否还存在对它们的引用。如果是的话，就必须手动解除引用

## 32.3. 常见内存泄露情况

意外的全局变量

```
1 function foo(arg) {  
2     bar = "this is a hidden global variable";  
3 }
```

另一种意外的全局变量可能由 `this` 创建：

```
1 function foo() {  
2     this.variable = "potential accidental global";  
3 }  
4 // foo 调用自己，this 指向了全局对象 (window)  
5 foo();
```

上述使用严格模式，可以避免意外的全局变量

定时器也常会造成内存泄露

```
1 var someResource = getData();  
2 setInterval(function() {  
3     var node = document.getElementById('Node');  
4     if(node) {  
5         // 处理 node 和 someResource  
6         node.innerHTML = JSON.stringify(someResource);  
7     }  
8 }, 1000);
```

如果 `id` 为 `Node` 的元素从 `DOM` 中移除，该定时器仍会存在，同时，因为回调函数中包含对 `someResource` 的引用，定时器外面的 `someResource` 也不会被释放

包括我们之前所说的闭包，维持函数内局部变量，使其得不到释放

JavaScript | 复制代码

```
1 function bindEvent() {  
2     var obj = document.createElement('XXX');  
3     var unused = function () {  
4         console.log(obj, '闭包内引用obj obj不会被释放');  
5     };  
6     obj = null; // 解决方法  
7 }
```

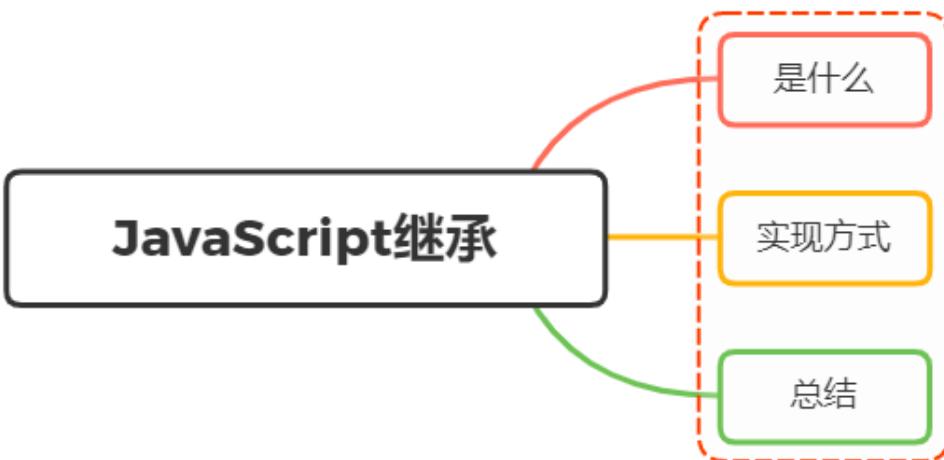
没有清理对 DOM 元素的引用同样造成内存泄露

JavaScript | 复制代码

```
1 const refA = document.getElementById('refA');  
2 document.body.removeChild(refA); // dom删除了  
3 console.log(refA, 'refA'); // 但是还存在引用能console出整个div 没有被回收  
4 refA = null;  
5 console.log(refA, 'refA'); // 解除引用
```

包括使用事件监听 addEventListener 监听的时候，在不监听的情况下使用 removeEventListener 取消对事件监听

## 33. Javascript如何实现继承?



### 33.1. 是什么

继承 (inheritance) 是面向对象软件技术当中的一个概念。

如果一个类别B“继承自”另一个类别A，就把这个B称为“A的子类”，而把A称为“B的父类别”也可以称“A是B的超类”

- 继承的优点

继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码

在子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能

虽然 JavaScript 并不是真正的面向对象语言，但它天生的灵活性，使应用场景更加丰富

关于继承，我们举个形象的例子：

定义一个类（Class）叫汽车，汽车的属性包括颜色、轮胎、品牌、速度、排气量等

```
▼ JavaScript | ⌂ 复制代码
1 class Car{
2   constructor(color,speed){
3     this.color = color
4     this.speed = speed
5     // ...
6   }
7 }
```

由汽车这个类可以派生出“轿车”和“货车”两个类，在汽车的基础属性上，为轿车添加一个后备厢、给货车添加一个大货箱

```
▼ JavaScript | ⌂ 复制代码
1 // 货车
2 class Truck extends Car{
3   constructor(color,speed){
4     super(color,speed)
5     this.Container = true // 货箱
6   }
7 }
```

这样轿车和货车就是不一样的，但是二者都属于汽车这个类，汽车、轿车继承了汽车的属性，而不需要再次在“轿车”中定义汽车已经有的属性

在“轿车”继承“汽车”的同时，也可以重新定义汽车的某些属性，并重写或覆盖某些属性和方法，使其获得与“汽车”这个父类不同的属性和方法

```
1 class Truck extends Car{  
2     constructor(color,speed){  
3         super(color,speed)  
4         this.color = "black" //覆盖  
5         this.Container = true // 货箱  
6     }  
7 }
```

从这个例子中就能详细说明汽车、轿车以及卡车之间的继承关系

## 33.2. 实现方式

下面给出 JavaScript 常见的继承方式：

- 原型链继承
- 构造函数继承（借助 call）
- 组合继承
- 原型式继承
- 寄生式继承
- 寄生组合式继承

### 33.2.1. 原型链继承

原型链继承是比较常见的继承方式之一，其中涉及的构造函数、原型和实例，三者之间存在着一定的关系，即每一个构造函数都有一个原型对象，原型对象又包含一个指向构造函数的指针，而实例则包含一个原型对象的指针

举个例子

JavaScript | 复制代码

```
1 function Parent() {
2     this.name = 'parent1';
3     this.play = [1, 2, 3]
4 }
5 function Child() {
6     this.type = 'child2';
7 }
8 Child.prototype = new Parent();
9 console.log(new Child())
```

上面代码看似没问题，实际存在潜在问题

JavaScript | 复制代码

```
1 var s1 = new Child();
2 var s2 = new Child();
3 s1.play.push(4);
4 console.log(s1.play, s2.play); // [1,2,3,4]
```

改变 `s1` 的 `play` 属性，会发现 `s2` 也跟着发生变化了，这是因为两个实例使用的是同一个原型对象，内存空间是共享的

### 33.2.2. 构造函数继承

借助 `call` 调用 `Parent` 函数

```
1 function Parent(){
2     this.name = 'parent1';
3 }
4
5 Parent.prototype.getName = function () {
6     return this.name;
7 }
8
9 function Child(){
10    Parent1.call(this);
11    this.type = 'child'
12 }
13
14 let child = new Child();
15 console.log(child); // 没问题
16 console.log(child.getName()); // 会报错
```

可以看到，父类原型对象中一旦存在父类之前自己定义的方法，那么子类将无法继承这些方法

相比第一种原型链继承方式，父类的引用属性不会被共享，优化了第一种继承方式的弊端，但是只能继承父类的实例属性和方法，不能继承原型属性或者方法

### 33.2.3. 组合继承

前面我们讲到两种继承方式，各有优缺点。组合继承则将前两种方式继承起来

```

1  function Parent3 () {
2      this.name = 'parent3';
3      this.play = [1, 2, 3];
4  }
5
6  Parent3.prototype.getName = function () {
7      return this.name;
8  }
9  function Child3() {
10     // 第二次调用 Parent3()
11     Parent3.call(this);
12     this.type = 'child3';
13 }
14
15 // 第一次调用 Parent3()
16 Child3.prototype = new Parent3();
17 // 手动挂上构造器，指向自己的构造函数
18 Child3.prototype.constructor = Child3;
19 var s3 = new Child3();
20 var s4 = new Child3();
21 s3.play.push(4);
22 console.log(s3.play, s4.play); // 不互相影响
23 console.log(s3.getName()); // 正常输出'parent3'
24 console.log(s4.getName()); // 正常输出'parent3'

```

这种方式看起来就没什么问题，方式一和方式二的问题都解决了，但是从上面代码我们也可以看到 `Parent3` 执行了两次，造成了多构造一次的性能开销

### 33.2.4. 原型式继承

这里主要借助 `Object.create` 方法实现普通对象的继承

同样举个例子

```
1 let parent4 = {
2     name: "parent4",
3     friends: ["p1", "p2", "p3"],
4     getName: function() {
5         return this.name;
6     }
7 };
8
9 let person4 = Object.create(parent4);
10 person4.name = "tom";
11 person4.friends.push("jerry");
12
13 let person5 = Object.create(parent4);
14 person5.friends.push("lucy");
15
16 console.log(person4.name); // tom
17 console.log(person4.name === person4.getName()); // true
18 console.log(person5.name); // parent4
19 console.log(person4.friends); // ["p1", "p2", "p3","jerry","lucy"]
20 console.log(person5.friends); // ["p1", "p2", "p3","jerry","lucy"]
```

这种继承方式的缺点也很明显，因为 `Object.create` 方法实现的是浅拷贝，多个实例的引用类型属性指向相同的内存，存在篡改的可能

### 33.2.5. 寄生式继承

寄生式继承在上面继承基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法

```
1 let parent5 = {
2     name: "parent5",
3     friends: ["p1", "p2", "p3"],
4     getName: function() {
5         return this.name;
6     }
7 };
8
9 function clone(original) {
10    let clone = Object.create(original);
11    clone.getFriends = function() {
12        return this.friends;
13    };
14    return clone;
15 }
16
17 let person5 = clone(parent5);
18
19 console.log(person5.getName()); // parent5
20 console.log(person5.getFriends()); // ["p1", "p2", "p3"]
```

其优缺点也很明显，跟上面讲的原型式继承一样

### 33.2.6. 寄生组合式继承

寄生组合式继承，借助解决普通对象的继承问题的 `Object.create` 方法，在前面几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对最优的继承方式

```

1  function clone (parent, child) {
2      // 这里改用 Object.create 就可以减少组合继承中多进行一次构造的过程
3      child.prototype = Object.create(parent.prototype);
4      child.prototype.constructor = child;
5  }
6
7  function Parent6() {
8      this.name = 'parent6';
9      this.play = [1, 2, 3];
10 }
11 Parent6.prototype.getName = function () {
12     return this.name;
13 }
14 function Child6() {
15     Parent6.call(this);
16     this.friends = 'child5';
17 }
18
19 clone(Parent6, Child6);
20
21 Child6.prototype.getFriends = function () {
22     return this.friends;
23 }
24
25 let person6 = new Child6();
26 console.log(person6); // {friends:"child5",name:"child5",play:[1,2,3],__proto__:Parent6}
27 console.log(person6.getName()); // parent6
28 console.log(person6.getFriends()); // child5

```

可以看到 person6 打印出来的结果，属性都得到了继承，方法也没问题

文章一开头，我们是使用 ES6 中的 extends 关键字直接实现 JavaScript 的继承

```

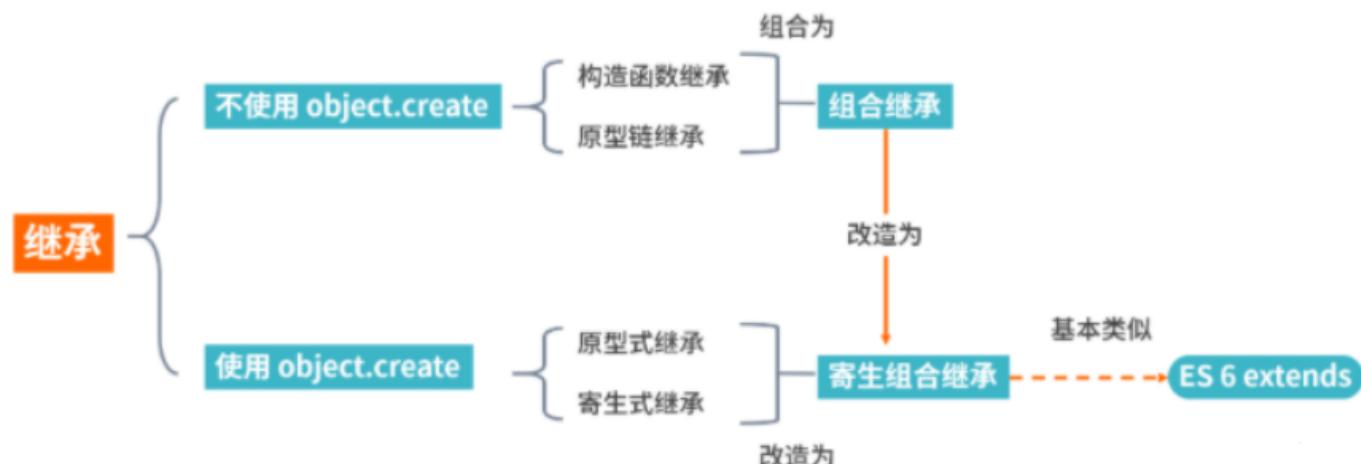
1  class Person {
2    constructor(name) {
3      this.name = name
4    }
5    // 原型方法
6    // 即 Person.prototype.getName = function() { }
7    // 下面可以简写为 getName() {...}
8    getName = function () {
9      console.log('Person:', this.name)
10   }
11 }
12 class Gamer extends Person {
13   constructor(name, age) {
14     // 子类中存在构造函数，则需要在使用“this”之前首先调用 super()。
15     super(name)
16     this.age = age
17   }
18 }
19 const asuna = new Gamer('Asuna', 20)
20 asuna.getName() // 成功访问到父类的方法

```

利用 `babel` 工具进行转换，我们会发现 `extends` 实际采用的也是寄生组合继承方式，因此也证明了这种方式是较优的解决继承的方式

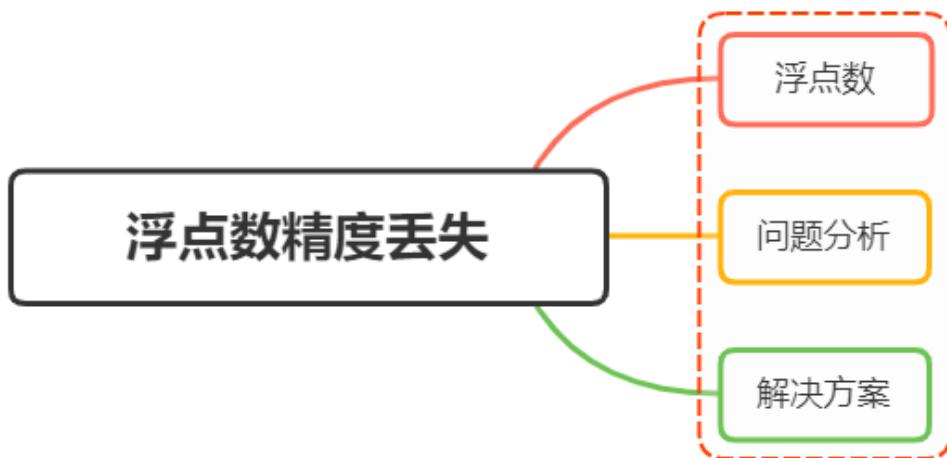
### 33.3. 总结

下面以一张图作为总结：



通过 `Object.create` 来划分不同的继承方式，最后的寄生式组合继承方式是通过组合继承改造之后的最优继承方式，而 `extends` 的语法糖和寄生组合继承的方式基本类似

## 34. 说说 Javascript 数字精度丢失的问题，如何解决？



### 34.1. 场景复现

一个经典的面试题

```
1 0.1 + 0.2 === 0.3 // false
```

为什么是 `false` 呢？

先看下面这个比喻

比如一个数  $1 \div 3 = 0.33333333\dots$

3会一直无限循环，数学可以表示，但是计算机要存储，方便下次取出来再使用，但 $0.333333\dots$  这个数无限循环，再大的内存它也存不下，所以不能存储一个相对于数学来说的值，只能存储一个近似值，当计算机存储后再取出时就会出现精度丢失问题

### 34.2. 浮点数

“浮点数”是一种表示数字的标准，整数也可以用浮点数的格式来存储

我们也可以理解成，浮点数就是小数

在 `JavaScript` 中，现在主流的数值类型是 `Number`，而 `Number` 采用的是 `IEEE754` 规范中64位双精度浮点数编码

这样的存储结构优点是可以归一化处理整数和小数，节省存储空间

对于一个整数，可以很轻易转化成十进制或者二进制。但是对于一个浮点数来说，因为小数点的存在，小数点的位置不是固定的。解决思路就是使用科学计数法，这样小数点位置就固定了

而计算机只能用二进制（0或1）表示，二进制转换为科学记数法的公式如下：

$$X = a * 2^e$$

其中，`a` 的值为0或者1，`e`为小数点移动的位置

举个例子：

27.0转化成二进制为11011.0，科学计数法表示为：

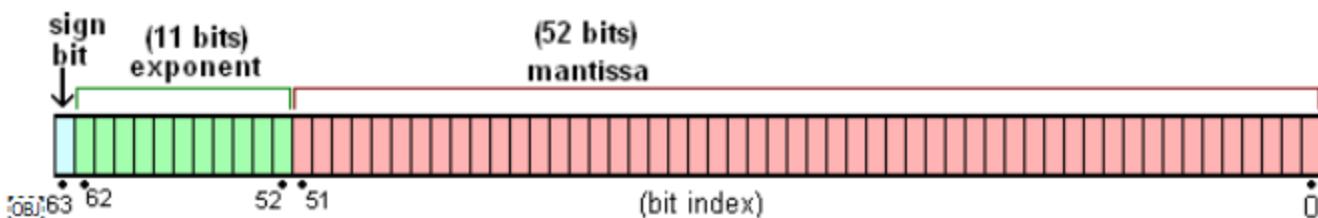
$$1.10110 * 2^4$$

前面讲到，`JavaScript` 存储方式是双精度浮点数，其长度为8个字节，即64位比特

64位比特又可分为三个部分：

- 符号位S：第1位是正负数符号位（sign），0代表正数，1代表负数
- 指数位E：中间的11位存储指数（exponent），用来表示次方数，可以为正负数。在双精度浮点数中，指数的固定偏移量为1023
- 尾数位M：最后的52位是尾数（mantissa），超出的部分自动进一舍零

如下图所示：



举个例子：

27.5 转换为二进制11011.1

11011.1转换为科学记数法  $1.10111 * 2^4$

符号位为1(正数)，指数位为4+，1023+4，即1027

因为它是十进制的需要转换为二进制，即 10000000011，小数部分为 10111，补够52位即：1011  
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000`

所以27.5存储为计算机的二进制标准形式(符号位+指数位+小数部分(阶数))，既下面所示

### 34.3. 问题分析

再回到问题上

```
1 0.1 + 0.2 === 0.3 // false
```

JavaScript |  复制代码

通过上面的学习，我们知道，在 javascript 语言中，0.1 和 0.2 都转化成二进制后再进行运算

JavaScript |  复制代码

所以输出 false

再来一个问题，那么为什么  $x=0.1$  得到  $0.1$ ？

主要是存储二进制时小数点的偏移量最大为52位，最多可以表达的位数

是  $2^{53}=9007199254740992$ ，对应科学计数尾数是  $9.007199254740992$ ，这也是 JS 最多能表示的精度

它的长度是 16，所以可以使用 `toPrecision(16)` 来做精度运算。超过的精度会自动做进位处理。

```
1 .1000000000000000555.toPrecision(16)  
2 // 返回 0.1000000000000000，去掉末尾的零后正好为 0.1
```

· JavaScript | 复制代码

但看到的  $0.1$  实际上并不是  $0.1$ ，不信你可用更高的精度试试：

JavaScript | 复制代码

```
1 0.1.toPrecision(21) = 0.10000000000000005551
```

如果整数大于 `9007199254740992` 会出现什么情况呢?

由于指数位最大值是1023, 所以最大可以表示的整数是 `2^1024 - 1`, 这就是能表示的最大整数。但你并不能这样计算这个数字, 因为从 `2^1024` 开始就变成了 `Infinity`

Plain Text | 复制代码

```
1 > Math.pow(2, 1023)
2 8.98846567431158e+307
3
4 > Math.pow(2, 1024)
5 Infinity
```

那么对于 `(2^53, 2^63)` 之间的数会出现什么情况呢?

- `(2^53, 2^54)` 之间的数会两个选一个, 只能精确表示偶数
- `(2^54, 2^55)` 之间的数会四个选一个, 只能精确表示4个倍数
- ... 依次跳过更多2的倍数

要想解决大数的问题你可以引用第三方库 `bignumber.js`, 原理是把所有数字当作字符串, 重新实现了计算逻辑, 缺点是性能比原生差很多

### 34.3.1. 小结

计算机存储双精度浮点数需要先把十进制数转换为二进制的科学记数法的形式, 然后计算机以自己的规则{符号位+(指数位+指数偏移量的二进制)+小数部分}存储二进制的科学记数法

因为存储时有位数限制 (64位), 并且某些十进制的浮点数在转换为二进制数时会出现无限循环, 会造成二进制的舍入操作(0舍1入), 当再转换为十进制时就造成了计算误差

## 34.4. 解决方案

理论上用有限的空间来存储无限的小数是不可能保证精确的, 但我们可以处理一下得到我们期望的结果

当你拿到 `1.4000000000000001` 这样的数据要展示时, 建议使用 `toPrecision` 凑整并 `parseFloat` 转成数字后再显示, 如下:

Plain Text | 复制代码

```
1 parseFloat(1.4000000000000001.toPrecision(12)) === 1.4 // True
```

封装成方法就是：

JavaScript | 复制代码

```
1 function strip(num, precision = 12) {
2     return +parseFloat(num.toPrecision(precision));
3 }
```

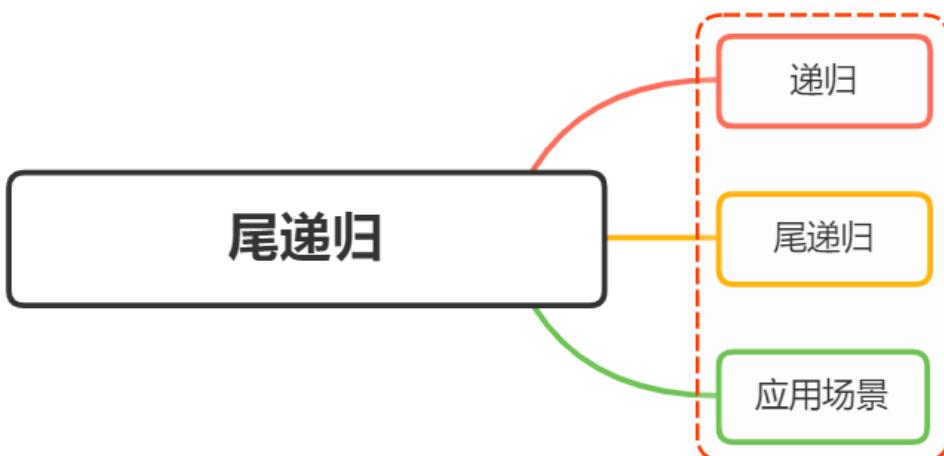
对于运算类操作，如 `+*/`，就不能使用 `toPrecision` 了。正确的做法是把小数转成整数后再运算。以加法为例：

JavaScript | 复制代码

```
1 /**
2  * 精确加法
3  */
4 function add(num1, num2) {
5     const num1Digits = (num1.toString().split('.')[1] || '').length;
6     const num2Digits = (num2.toString().split('.')[1] || '').length;
7     const baseNum = Math.pow(10, Math.max(num1Digits, num2Digits));
8     return (num1 * baseNum + num2 * baseNum) / baseNum;
9 }
```

最后还可以使用第三方库，如 `Math.js`、`BigDecimal.js`

## 35. 举例说明你对尾递归的理解，有哪些应用场景



## 35.1. 递归

递归（英语：Recursion）

在数学与计算机科学中，是指在函数的定义中使用函数自身的方法

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数

其核心思想是把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解

一般来说，递归需要有边界条件、递归前进阶段和递归返回阶段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回

下面实现一个函数 `pow(x, n)`，它可以计算 `x` 的 `n` 次方

使用迭代的方式，如下：

```
1 function pow(x, n) {
2     let result = 1;
3
4     // 再循环中，用 x 乘以 result n 次
5     for (let i = 0; i < n; i++) {
6         result *= x;
7     }
8     return result;
9 }
```

JavaScript | 复制代码

使用递归的方式，如下：

```
1 function pow(x, n) {
2     if (n == 1) {
3         return x;
4     } else {
5         return x * pow(x, n - 1);
6     }
7 }
```

JavaScript | 复制代码

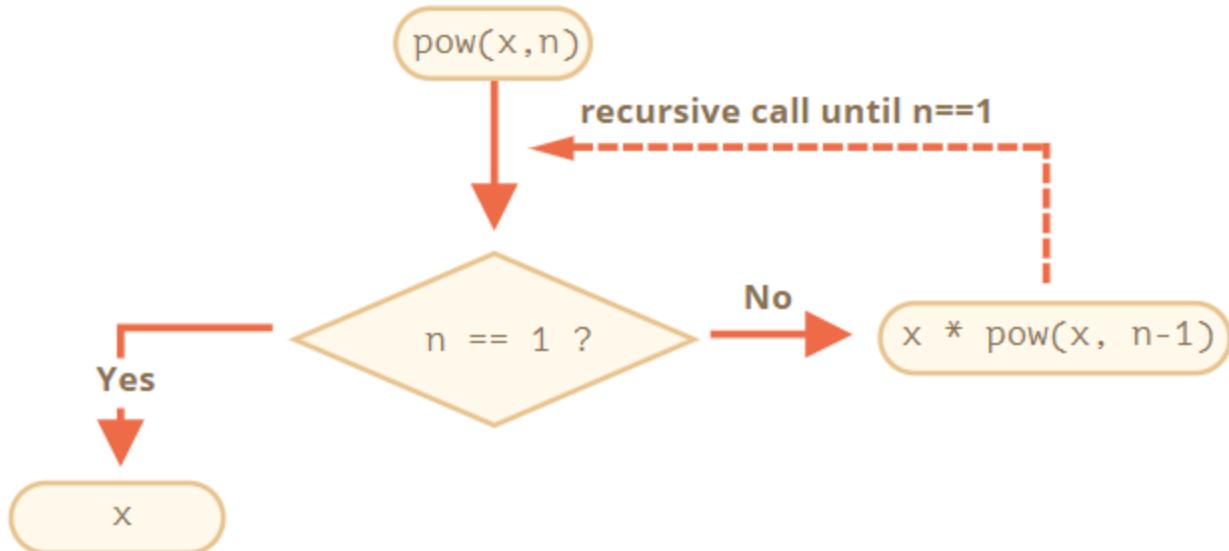
`pow(x, n)` 被调用时，执行分为两个分支：

```

1         if n==1 = x
2         /
3 pow(x, n) =
4         \
5         else     = x * pow(x, n - 1)

```

也就是说 `pow` 递归地调用自身 直到 `n == 1`



为了计算 `pow(2, 4)`，递归变体经过了下面几个步骤：

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

因此，递归将函数调用简化为一个更简单的函数调用，然后再将其简化为一个更简单的函数，以此类推，直到结果

## 35.2. 尾递归

尾递归，即在函数尾位置调用自身（或是一个尾调用本身的其他函数等等）。尾递归也是递归的一种特殊情形。尾递归是一种特殊的尾调用，即在尾部直接调用自身的递归函数

尾递归在普通尾调用的基础上，多出了2个特征：

- 在尾部调用的是函数自身
- 可通过优化，使得计算仅占用常量栈空间

在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储，递归次数过多容易造成栈溢出

这时候，我们就可以使用尾递归，即一个函数中所有递归形式的调用都出现在函数的末尾，对于尾递归来说，由于只存在一个调用记录，所以永远不会发生“栈溢出”错误

实现一下阶乘，如果用普通的递归，如下：

```
▼ JavaScript | 复制代码

1 function factorial(n) {
2     if (n === 1) return 1;
3     return n * factorial(n - 1);
4 }
5
6 factorial(5) // 120
```

如果 `n` 等于5，这个方法要执行5次，才返回最终的计算表达式，这样每次都要保存这个方法，就容易造成栈溢出，复杂度为  $O(n)$

如果我们使用尾递归，则如下：

```
▼ JavaScript | 复制代码

1 function factorial(n, total) {
2     if (n === 1) return total;
3     return factorial(n - 1, n * total);
4 }
5
6 factorial(5, 1) // 120
```

可以看到，每一次返回的就是一个新的函数，不带上一个函数的参数，也就不需要储存上一个函数了。尾递归只需要保存一个调用栈，复杂度  $O(1)$

### 35.3. 应用场景

数组求和

JavaScript | 复制代码

```
1 function sumArray(arr, total) {  
2     if(arr.length === 1) {  
3         return total  
4     }  
5     return sum(arr, total + arr.pop())  
6 }
```

使用尾递归优化求斐波那契数列

JavaScript | 复制代码

```
1 function factorial2 (n, start = 1, total = 1) {  
2     if(n <= 2){  
3         return total  
4     }  
5     return factorial2 (n -1, total, total + start)  
6 }
```

数组扁平化

JavaScript | 复制代码

```
1 let a = [1,2,3, [1,2,3, [1,2,3]]]  
2 // 变成  
3 let a = [1,2,3,1,2,3,1,2,3]  
4 // 具体实现  
5 function flat(arr = [], result = []) {  
6     arr.forEach(v => {  
7         if(Array.isArray(v)) {  
8             result = result.concat(flat(v, []))  
9         }else {  
10             result.push(v)  
11         }  
12     })  
13     return result  
14 }
```

数组对象格式化

```
1 let obj = {
2     a: '1',
3     b: {
4         c: '2',
5         D: {
6             E: '3'
7         }
8     }
9 }
10 // 转化为如下:
11 let obj = {
12     a: '1',
13     b: {
14         c: '2',
15         d: {
16             e: '3'
17         }
18     }
19 }
20
21 // 代码实现
22 function keysLower(obj) {
23     let reg = new RegExp("( [A-Z]+)", "g");
24     for (let key in obj) {
25         if (obj.hasOwnProperty(key)) {
26             let temp = obj[key];
27             if (reg.test(key.toString())) {
28                 // 将修改后的属性名重新赋值给temp，并在对象obj内添加一个转换后的属性
29                 temp = obj[key.replace(reg, function (result) {
30                     return result.toLowerCase()
31                 })] = obj[key];
32                 // 将之前大写的键属性删除
33                 delete obj[key];
34             }
35             // 如果属性是对象或者数组，重新执行函数
36             if (typeof temp === 'object' || Object.prototype.toString.call(temp) === '[object Array]') {
37                 keysLower(temp);
38             }
39         }
40     }
41     return obj;
42 }
```

## **35.4.**