# HTML

## 盒子模型

box-sizing:content-box;

width=width+padding+border+margin

Box-sizing:border-box;

width=width+margin

## html5新特性

https://www.cnblogs.com/vicky1018/p/7705223.html

1、语义化标签 2、增强型表单 3、视频和音频 4、Canvas绘图 5、SVG绘图 6、地理定位 7、拖放API（drag）

8、Web Worker 9、Web Storage 10、WebSocket

## 列举form标签4个常用的属性

| **accept-charset** | ***charset_list*** | **规定服务器可处理的表单数据字符集。** |
| --- | --- | --- |
| action | *URL* | 规定当提交表单时向何处发送表单数据。 |
| autocomplete | onoff | 规定是否启用表单的自动完成功能。 |
| enctype | 见说明 | 规定在发送表单数据之前如何对其进行编码。 |
| method | getpost | 规定用于发送 form-data 的 HTTP 方法。 |
| name | *form_name* | 规定表单的名称。 |
| novalidate | novalidate | 如果使用该属性，则提交表单时不进行验证。 |
| target | _blank_self_parent_top*framename* | 规定在何处打开 action URL。 |

# CSSS

# display

inherit、none、block、inline-block、table、inline-table、inherit、flex、grid、inline-grid

```
display: grid;
grid-template-rows指定的每个值可以创建每行的高度。
grid-template-rows指定的每个值可以创建每行的高度。
grid-row-gap创建行与行之间的间距
grid-column-gap创建列与列之间的间距
grid-template-rows: 10px 20px 30px 第一行10px 第二行20px 第三行30px
grid-template-columns: 1fr 1fr 2fr;
```

# clientWidth、scrollWidth与offsetWidth的区别详解

https://cloud.tencent.com/developer/article/1410065

**没滚动条的情况下**

```
clientWidth = content+padding
scrollWidth = content+padding
offsetWidth = content+padding+border
```

**有滚动条的情况下**

```
外面box的 clientWidth = content+padding-滚动条的宽度（大概是17px）
外面box的 scrollWidth = 里面box的总宽度（里面box的content+padding+border）+外面box一边的padding
外面box的 offsetWidth = 外面box自身的content+padding+border
```

# css实现多列等高容器布局，要求元素实际占用的高度以多列中较高的为准

# 重绘重排

首先是chome调试工具performance 录制刷新 可以看到页面元素重绘重排的过程

callTree task(可以看到重绘重排的每个步骤和时间)

1、获取dom 分割层

2、根据每层节点结算样式结果 Recalculte Style

3、为每个节点生成图形和位置 layout

4、将每个节点回执填充到当前帧的图层位图中 paint

5、将图层上传到gpu（显卡）bitmap 专门处理图像

6、根据符合要求的多个图层合并生成图像 给你看 composite layers

**重绘操作**

改变vidibility、outline、背景色等属性 box-shadow

**重排操作**

1. DOM元素的几何属性变化。
2. DOM树的结构变化。

> 当获取一些属性时，浏览器为取得正确的值也会触发重排。这样就使得浏览器的优化失效了。这些属性包括：offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight、getComputedStyle() (currentStyle in IE)。所以，在多次使用这些值时应进行缓存。

**减少重排次数和重排影响范围**

1、将多次改变样式属性的操作合并成一次操作。例如，

```
JS:
    var changeDiv = document.getElementById('changeDiv');
    changeDiv.style.color = '#093';
    changeDiv.style.background = '#eee';
    changeDiv.style.height = '200px';
    可以合并为:
CSS:
    div.changeDiv {
        background: #eee;
        color: #093;
        height: 200px;
    }
JS:
    document.getElementById('changeDiv').className = 'changeDiv';
```

2、 将需要多次重排的元素，position属性设为absolute或fixed，这样此元素就脱离了文档流，它的变化不会影响到其他元素。例如有动画效果的元素就最好设置为绝对定位。

3、 在内存中多次操作节点，完成后再添加到文档中去。例如要异步获取表格数据，渲染到页面。可以先取得数据后在内存中构建整个表格的html片段，再一次性添加到文档中去，而不是循环添加每一行。

4、 由于display属性为none的元素不在渲染树中，对隐藏的元素操作不会引发其他元素的重排。如果要对一个元素进行复杂的操作时，可以先隐藏它，操作完成后再显示。这样只在隐藏和显示时触发2次重排。

5、 在需要经常取那些引起浏览器重排的属性值时，要缓存到变量

Css3 transform GPU加速

淘宝：translate3d(-1560px, 0px, 0px); backface-visibility: hidden; left: 0px; opacity: 1;

# JS

## js运行过程

https://www.cnblogs.com/chengxs/p/10240163.html

### 语法分析

分析该js脚本代码块的语法是否正确，如果出现不正确会向外抛出一个**语法错误（syntaxError）**，停止该js代码的执行，然后继续查找并加载下一个代码块；如果语法正确，则进入到预编译阶段。

### 预编译

**运行环境：**

1、全局环境（js代码加载完毕后，进入到预编译也就是进入到全局环境）

2、函数环境（函数调用的时候，进入到该函数环境，不同的函数，函数环境不同）

3、eval环境（不建议使用，存在安全、性能问题）

每进入到一个不同的运行环境都会创建 一个相应的**执行上下文（execution context）**，那么在一段js程序中一般都会创建多个执行上下文，js引擎会以栈的数据结构对这些执行进行处理，形成**函数调用栈（call stack）**，栈底永远是**全局执行上下文（global execution context）**，栈顶则永远时当前的执行上下文。

**创建执行上下文**

1、创建变量对象（variable object）

2、创建作用域链（scope chain）

当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

```
当函数激活时，进入函数上下文，创建 VO/AO 后，就会将活动对象添加到作用链的前端。
Scope = [AO].concat([[Scope]]);
```

3、确定this的指向

**编译细节：**

1、创建AO对象

2、找行参和变量声明,将变量和形参名作为AO属性名，值为undefined

3、将实参和形参统一

4、在函数体里面找函数声明，赋值函数体

### 解释执行

## 防抖节流

### 防抖

```
function debounce(handler, delay) {
  var timer = null;
  return function () {
    var _self = this, _args = arguments;
    clearTimeout(timer);
    timer = setTimeout(function () {
      handler.apply(_self, _args);
    }, delay)
  }
}
```

### 节流

```
function throttle(handler, wait) {
  var lastTime = 0;
  return function (e) {
    var nowTime = new Date().getTime();
    if (nowTime - lastTime > wait) {
      handler.apply(this, arguments);
      lastTime = nowTime;
    }
  }
}
```

## js数组中filter、map、reduce、find等方法实现的原理

https://juejin.im/post/5c8b5c99f265da2da23d7350#heading-1

### map

```
Array.prototype.map = function(fn) {
  let newArr = [];
  for (let i = 0; i < this.length; i++) {
    newArr.push(fn(this[i]))
  };
  return newArr;
}
```

## set、weakSet、map、weakMap

集合

哈希结构 键值对对象

WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

这是因为垃圾回收机制依赖引用计数，如果一个值的引用次数不为`0`，垃圾回收机制就不会释放这块内存。结束使用该值之后，有时会忘记取消引用，导致内存无法释放，进而可能会引发内存泄漏。WeakSet 里面的引用，都不计入垃圾回收机制，所以就不存在这个问题。因此，WeakSet 适合临时存放一组对象，以及存放跟对象绑定的信息。只要这些对象在外部消失，它在 WeakSet 里面的引用就会自动消失。

由于上面这个特点，WeakSet 的成员是不适合引用的，因为它会随时消失。另外，由于 WeakSet 内部有多少个成员，取决于垃圾回收机制有没有运行，运行前后很可能成员个数是不一样的，而垃圾回收机制何时运行是不可预测的，因此 ES6 规定 WeakSet 不可遍历。

WeakMap 就是为了解决这个问题而诞生的，它的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。

## Generator

Generator 函数是一个状态机，封装了多个内部状态。执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

Generator 函数是一个普通函数，但是有两个特征。一是，`function`关键字与函数名之间有一个星号；二是，函数体内部使用`yield`表达式

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();
```

上面代码定义了一个 Generator 函数`helloWorldGenerator`，它内部有两个`yield`表达式（`hello`和`world`），即该函数有三个状态：hello，world 和 return 语句

调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。

```
hw.next()
// { value: 'hello', done: false }
hw.next()
// { value: 'world', done: false }
hw.next()
// { value: 'ending', done: true }
hw.next()
// { value: undefined, done: true }
```

## promise

https://juejin.im/post/5c41297cf265da613356d4ec#heading-8

```
//Promise 的三种状态  (满足要求 -> Promise的状态)
const PENDING = 'pending';
const FULFILLED = 'fulfilled';
const REJECTED = 'rejected';

class AjPromise {
  constructor(fn) {
    //当前状态
    this.state = PENDING;
    //终值
    this.value = null;
    //拒因
    this.reason = null;
    //成功态回调队列
    this.onFulfilledCallbacks = [];
    //拒绝态回调队列
    this.onRejectedCallbacks = [];

    //成功态回调
    const resolve = value => {
      console.log('2秒后')
      // 使用macro-task机制(setTimeout),确保onFulfilled异步执行,且在 then 方法被调用的那一轮事件循环       之后的新执行栈中执行。
      setTimeout(() => {
        if (this.state === PENDING) {
          // pending(等待态)迁移至 fulfilled(执行态),保证调用次数不超过一次。
          this.state = FULFILLED;
          // 终值
          this.value = value;
          this.onFulfilledCallbacks.map(cb => {
            console.log(this.onFulfilledCallbacks,'🍎')
            this.value = cb(this.value);
          });
```

```
        }
      });
    };
    //拒绝态回调
    const reject = reason => {
      // 使用macro-task机制(setTimeout),确保onRejected异步执行,且在 then 方法被调用的那一轮事件循环之                后的新执行栈中执行。 (满足要求 -> 调用时机)
      setTimeout(() => {
        if (this.state === PENDING) {
          // pending(等待态)迁移至 fulfilled(拒绝态),保证调用次数不超过一次。
          this.state = REJECTED;
          //拒因
          this.reason = reason;
          this.onRejectedCallbacks.map(cb => {
            this.reason = cb(this.reason);
          });
        }
      });
    };
    try {
      //执行promise
      fn(resolve, reject);
      console.log(resolve);
    } catch (e) {
      reject(e);
    }
  }
  then(onFulfilled, onRejected) {
    typeof onFulfilled === 'function' &&
this.onFulfilledCallbacks.push(onFulfilled);
    typeof onRejected === 'function' &&
this.onRejectedCallbacks.push(onRejected);
    // 返回this支持then 方法可以被同一个 promise 调用多次
    return this;
  }
}

new AjPromise((resolve, reject) => {
  setTimeout(() => {
    resolve(2);
  }, 2000);
}).then(res => {
  console.log(res);
  return res + 1;
}).then(res => {
  console.log(res);
});
```

# 面向切面

Aspect Oriented Programming(AOP),面向切面编程，AOP主要实现的目的是针对业务处理过程中的切面进行提取，它所面对的是处理过程中某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。

```
function test() {
    alert(2)
    return 'hello world!'
}

Function.prototype.before = function(fn) {
    var __self = this;
    return function() {
        fn.apply(this,arguments);
        console.log('this:',this);
        __self.apply(__self,arguments)
    }
}

Function.prototype.after = function(fn) {
    var __self = this;
    return function() {
        __self.apply(__self,arguments);
        fn.apply(this,arguments)
    }
}

test.before(function() {
    alert(1)
}).after(function() {
    alert(3)
})()
```

# 函数式编程

## 纯函数

对于相同的输入，永远得到相同的输出，而且没有可观察的副作用，也不依赖外部环境的状态。

```
let min = 19;
let checkage = age => age > min;
xs.splice(0,3)//这个就不是纯函数，因为它改变了原来的数组。

let min = 19;
let checkage = age => age > min;
xs.slice(0,3)
```

## 函数的柯里化

传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数

```
//柯里化之前
function add(x,y) {
    return x + y;
}
add(1,2)

//柯里化之后
function addx(y) {
    return functuon(x) {
        return x + y;
    }
}
addx(1)(2)

function foo(p1,p2){
    this.val = p1 + p2;
}
var bar = foo.bind(null,'p1');
var baz = new bar('p2')
console.log(baz.val);//p1p2
```

## 函数组合

纯函数以及如何把它柯里化写出来的洋葱代码h(g(f(x))),为了解决函数嵌套的问题，我们需要用到‘函数组合

这种风格能够帮助我们减少不必要的命名，让代码保持简洁和通用。

```
const f = str => str.toUpperCase().split(‘’);

var toUpperCase = world => word.toUpperCase();
var split = x => (str => str.split(x));
var f = compose(split(''),toUpperCase);
f('abcd efgh');
```

## 惰性函数

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

function addEvent (type, element, fun) {
    if (element.addEventListener) {
        addEvent = function (type, element, fun) {
            element.addEventListener(type, fun, false);
        }
    }
    else if(element.attachEvent){
        addEvent = function (type, element, fun) {
            element.attachEvent('on' + type, fun);
        }
    }
    else{
        addEvent = function (type, element, fun) {
            element['on' + type] = fun;
        }
    }
    return addEvent(type, element, fun);
}
```

可以看出，第一次调用addEvent会对浏览器做能力检测，然后，重写了addEvent。下次再调用的时候，由于函数被重写，不会再做能力检测。

```
function ajax() {
    var xhr = null;
    if(window.XMLHttpRequset) {
        xhr = new XMLHttpRequset()
    } else {
        xhr = new ActiveXObject('Microsoft.XMLHTTP');
    }
    ajax = xhr;
    return xhr;
}
```

## 高阶函数

## 尾调用优化

```
function sum(x){
  if(x == 1) {return 1;}
  return x + sum(x - 1)
}
sun(5)
function sum(x,total){
  if(x == 1) {return x + total;}
  return sun(x - 1,x + total)
}
sun(5,0)
```

## 事件委托

给父子元素用addEventListener()绑定同一个事件时，当触发子元素身上的事件，会先触发父元素，然后在传递给子元素，这种传播机制叫事件捕获。

当给父子元素的同一事件绑定方法时，触发了子元素身上的事件，执行完毕之后，也会触发父级元素的相同事件，这种传播机制叫事件冒泡。

attachEvent()和addEventListener()二者区别

1. attachEvent只用在iE8以下，addEventListener()适合标准浏览器
2. attachEvent的事件名带on 而addEventListener事件名不带on
3. attachEvent函数里面的this是window，而addEventListener函数里面的this是当前元素对象

attachEvent只有冒泡没有捕获addEventListener有冒泡也有捕获

ie低版本没有捕获

通过事件冒泡

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

**阻止事件冒泡：w3c的方法是e.stopPropagation()，IE则是使用e.cancelBubble = true;**

**取消默认事件：w3c的方法是e.preventDefault()，IE则是使用e.returnValue = false;**

## common.js cmd amd

Common.js 非浏览器规范

解决思路之一是，开发一个服务器端组件，对模块代码作静态分析，将模块与它的依赖列表一起返回给浏览器端。 这很好使，但需要服务器安装额外的组件，并因此要调整一系列底层架构。

**amd**

声明提前RequireJS

**cmd**

就近声明SeaJs 玉伯 注释问题//var xx = require('XX')

**umd**

## 元编程

**对编程语言的编程**

**Symbol.toPrimitive**是一个内置的 Symbol 值，它是作为对象的函数值属性存在的，当一个对象转换为对应的原始值时，会调用此函数。

```
const object1 = {
  [Symbol.toPrimitive](hint) {
    if (hint == 'number') {
      return 42;
    }
    return null;
  }
};

console.log(+object1);
// expected output: 42

let num = {
  [Symbol.toPrimitive]: ((i) => () => i++)(1)
}

if (num == 1 && num == 2 && num == 3) {
  console.log('一个值可以代表多个值');
}
```

---

**proxy**

```
let obj = {age:30};
const validator = {
  set(target,key,value) {
    if(typeof value!='number' || Number.isNaN(value)){
        throw new Error('年龄必须是数字');
    }
  }
}
const proxy = new Proxy(obj,validator);
proxy.age = 'haha';
```

**Reflect**

很多语言需要编译，编译过后类无法操作

**Tco 直接开启尾递归调用优化**

```
function test(i) {
  return test(i--,i)
  TCO_ENABLED = true;
}
test(5)
```

## 下列结果输出什么

https://blog.csdn.net/jackshiny/article/details/51941796

```
var a = { x: 1 };
var b = a;
a.x = a = { n: 1 };
console.log(a);
console.log(b);
```

## js创建对象的工厂模式、构造函数模式、原型模式、构造函数+原型模式

## instanceof 和 typeof 的实现原理

https://juejin.im/post/5b0b9b9051882515773ae714

typeof对于对象来说，除了函数都会显示object

typeof对于原始类型来说，除了null都可以显示正确类型

### Typeof

**js底层存储变量**

会在变量的机器码的低位1-3位存储其类型信息

- 000：对象
- 010：浮点数
- 100：字符串
- 110：布尔
- 1：整数

`null`：所有机器码均为0

`undefined`：用 −2^30 整数来表示

所以`typeof`在判断`null`的时候就出现问题了，由于`null`的所有机器码均为0，因此直接被当做了对象来看。

```
null instanceof null // TypeError: Right-hand side of 'instanceof' is not an object
```

### instanceof

```
function new_instance_of(leftVaule, rightVaule) {
  let rightProto = rightVaule.prototype; // 取右表达式的 prototype 值
  leftVaule = leftVaule.__proto__; // 取左表达式的__proto__值
  while (true) {
    if (leftVaule === null) {
      return false;
    }
    if (leftVaule === rightProto) {
      return true;
    }
    leftVaule = leftVaule.__proto__
  }
}
```

## JavaScript 执行机制

https://juejin.im/post/59e85eebf265da430d571f89

Micro-task:Promise、MutationObserver、Object.observe(废弃)，以及nodejs中的process.nextTick.

Macro-task:script(整体代码)、setImmediate、MessageChannel、setTimeout、I/O、UI rendering

## 找出3个数组共同的元素

```
let arr1 = [3,5,7,9,21];
let arr2 = [-10,3,8,79,5,9];
let arr3 = [11,5,16,42,3];

const intersect = (a, b) => {
  if (Array.isArray(a) && Array.isArray(b)) {
    return Array.from(new Set(a.filter(x => b.includes(x))));
  }
  return []
}

console.log(intersect(intersect(arr1,arr2),arr3))
```

## this指向

**this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，实际上this的最终指向的是那个调用它的对象**

**如果返回值是一个对象，那么this指向的就是那个返回的对象，如果返回值不是一个对象那么this还是指向函数的实例。**

```
function fn()
{
    this.user = '追梦子';
    return {};
}
var a = new fn;
console.log(a.user); //undefined
```

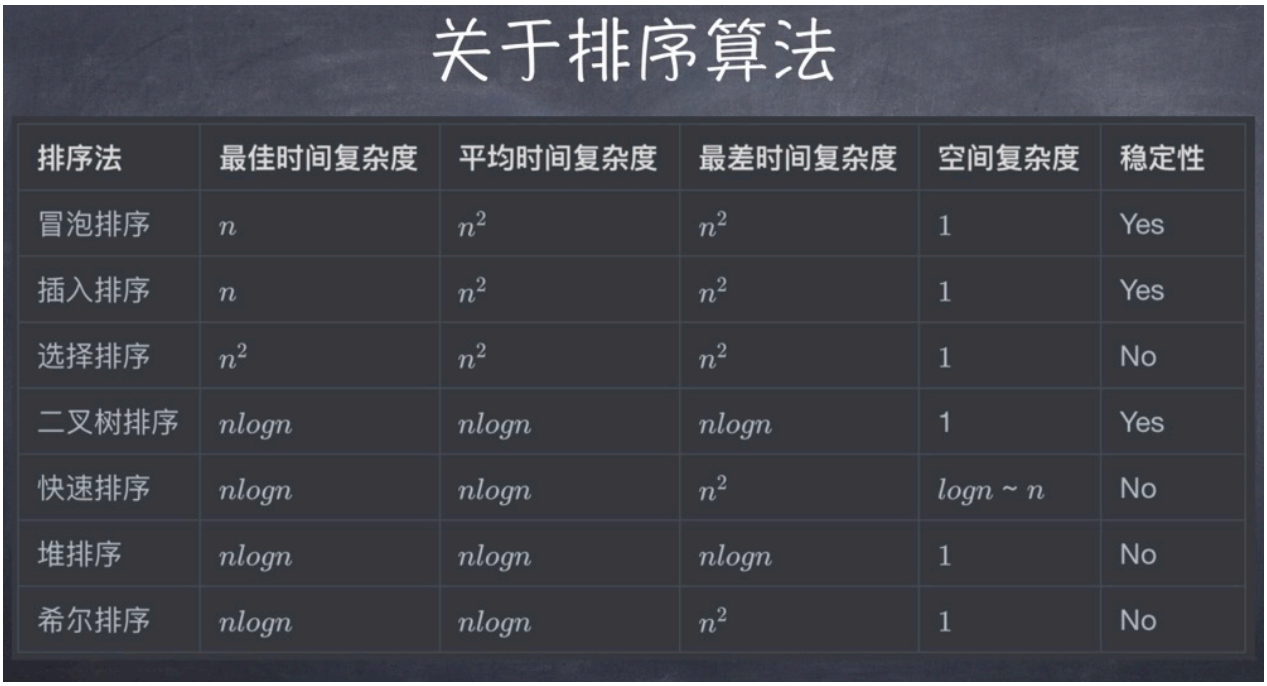
还有一点就是虽然null也是对象，但是在这里this还是指向那个函数的实例，因为null比较特殊。

```
function fn()
{
    this.user = '追梦子';
    return 1;
}
var a = new fn;
console.log(a.user); //追梦子

function fn()
{
    this.user = '追梦子';
    return undefined;
}
var a = new fn;
console.log(a.user); //追梦子

function fn()
```

```
{
    this.user = '追梦子';
    return null;
}
var a = new fn;
console.log(a.user); //追梦子
```

```
function Foo() {
  getName = function () { alert(1) };
  return this;
}
Foo.getName = function () { alert(2) };
Foo.prototype.getName = function () { alert(3) };
var getName = function () { alert(4) };
function getName() { alert(5) };
Foo.getName();//2
getName();//4
Foo().getName();//1 window.getName()
getName();//1 直接调用getName函数，相当于 window.getName() ，因为这个变量已经被Foo函数执行时修改了，遂结果与第三问相同，为1
new Foo.getName();//2 通过查上表可以得知点（.）的优先级高于new操作，遂相当于是:new (Foo.getName)();
new Foo().getName()
//第六问 new Foo().getName() ，首先看运算符优先级括号高于new，实际执行为(new Foo()).getName()
// 原题中，返回的是this，而this在构造函数中本来就代表当前实例化对象，遂最终Foo函数返回实例化对象。之后调用实例化对象的getName函数，因为在Foo构造函数中没有为实例化对象添加任何属性，遂到当前对象的原型对象（prototype）中寻找getName，找到了。
new new Foo().getName();//new ((new Foo()).getName)();
```

## 哪些排序算法不稳定




| 排序法 | 最佳时间复杂度 | 平均时间复杂度 | 最差时间复杂度 | 空间复杂度 | 稳定性 |
| --- | --- | --- | --- | --- | --- |
| 冒泡排序 | $n$ | $n^2$ | $n^2$ | 1 | Yes |
| 插入排序 | $n$ | $n^2$ | $n^2$ | 1 | Yes |
| 选择排序 | $n^2$ | $n^2$ | $n^2$ | 1 | No |
| 二叉树排序 | $nlogn$ | $nlogn$ | $nlogn$ | 1 | Yes |
| 快速排序 | $nlogn$ | $nlogn$ | $n^2$ | $logn \sim n$ | No |
| 堆排序 | $nlogn$ | $nlogn$ | $nlogn$ | 1 | No |
| 希尔排序 | $nlogn$ | $nlogn$ | $n^2$ | 1 | No |

## js获取当前页面中标签的种类

```
//方法一
var dom = document.querySelectorAll('*');
var str = [];
for(var i=0;i<dom.length;i++){
  if(str.indexOf(dom[i].nodeName)<0) {
    str.push(dom[i].nodeName)
  }
}
//方法二
var newArr = new
Set([...document.getElementsByTagName('*')].map(x=>x.tagName))
```

## js 实现trim()

```
String.prototype.trim = function() {
  return this.replace(/^(\s*)|(\s*)$/g)
}
```

## 判断参数是否是数组

```
Object.prototype.toString.call(arguments)
```

## js数组平铺

```
//方法一
let arr = [1, 2, 3, 4, [5, 5], [7, 8]];
arr = arr.join(',').split(',').map(item => Number(item));
//方法二
let newArr = [];
for(let item of arr) {
  newArr = newArr.concat(item)
}
//方法三 []可选。传递给函数的初始值
arr = arr.reduce((r,item)=>r.concat(item),[])
//方法四
arr.flat();
```

# 闭包

**当内部函数被保存到外部的时候，将会产生闭包。**

**闭包会导致原有作用域链不释放，造成内存泄露。**

## 实现公有变量 实现一个累加器

```
function add() {
  var count = 0;
  function demo () {
    count++;
    console.log(count)
  }
  return demo;
}
var counter = new add();
counter()//1
counter()//2
```

## 可以做缓存 保存数据结构

```
function test() {
  var num = 100;
  function a() {
    num++;
    console.log(num)
  }
  function b() {
    num--;
    console.log(num)
  }
  return [a,b]
}
var newArr = new test();
```

```
newArr[0]();
newArr[1]();
```

### 模块化开发 防止污染全局变量

```
var name = 'xiaozhang';
var init = (function () {
  var name = 'xiaoming';
  function callName() {
    console.log(name);
  }
  return function () {
    callName()
  }
}())
init()//xiaoming;
```

## New

https://github.com/mqyqingfeng/Blog/issues/13

```
function _new(fun) {
  var newObj = Object.create(fun.prototype);
  var returnObj = fun.call(newObj);
  if(typeof returnObj === 'object'){
    return returnObj
  }else{
    return newObj
  }
}
```

## call

```
var person = {
  age: 18
}

Function.prototype.newCall = function (context, args) {
  // this指向调用call的对象
  if (typeof this !== 'function') { // 调用call的若不是函数则报错
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this   // 将调用call函数的对象添加到context的属性中
  const result = context.fn(...[...arguments].slice(1)) // 执行该属性
  delete context.fn   // 删除该属性
  return result
```

```
}

function son(name) {
  console.log(this.age)
  console.log(name)
}

son.newCall(person, 'xiaoming')
```

## apply

```
Function.prototype.newCall = function (context, args) {
  console.log(args)
  // this指向调用call的对象
  if (typeof this !== 'function') { // 调用call的若不是函数则报错
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this   // 将调用call函数的对象添加到context的属性中
  // const result = context.fn(...[...arguments].slice(1)) // 执行该属性
  var result;
  if (args) {
    result = context.fn(...args)
  } else {
    result = context.fn()
  }
  delete context.fn   // 删除该属性
  return result
}
```

## bind

```
function foo() {
  this.b = 100;
  this.c = 200;
  return this.a;
}

var func = foo.bind({ a: 1 });
func();// 1
new func() // foo {b: 100, c: 200}
```

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function (oThis) {
    if (typeof oThis !== 'function') {
      throw new TypeError('试图绑定的内容不可调用')
    }
```

```
    var aArgs = Array.prototype.slice.call(arguments, 1);
    fToBind = this;
    fNOP = function () { };

    fBound = function () {
      return fToBind.apply(this instanceof fNOP ? this : oThis,

aArgs.concat(Array.prototype.slice.call(arguments)))
    }

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();

    return fBound;
  }
}
```

## 继承

```
function Person(color) {
    this.color = color;
}
Person.prototype.say = function () {
    console.log('Hello world!');
}

function chinese(color) {
    Person.call(this, color)
}


var __prototype = Object.create(Person.prototype);
__prototype.constructor = chinese;
chinese.prototype = __prototype;
chinese.prototype.kungfu = function () {
    console.log('yongchun!');
}
var xiaoming = new chinese('yellow');
console.log(xiaoming);
```

## 浅拷贝

var new_arr = arr.concat();

var new_arr = arr.slice();

```
var arr = [{old: 'old'}, ['old']];
var new_arr = arr.concat();
arr[0].old = 'new';
arr[1][0] = 'new';
console.log(arr) // [{old: 'new'}, ['new']]
console.log(new_arr) // [{old: 'new'}, ['new']]
```

我们会发现，无论是新数组还是旧数组都发生了变化，也就是说使用 concat 方法，克隆的并不彻底。

```
var shallowCopy = function(obj) {
  // 只拷贝对象
  if (typeof obj !== 'object') return;
  // 根据obj的类型判断是新建一个数组还是对象
  var newObj = obj instanceof Array ? [] : {};
  // 遍历obj，并且判断是obj的属性才拷贝
  for (var key in obj) {
    if (obj.hasOwnProperty(key)) {
      newObj[key] = obj[key];
    }
  }
  return newObj;
}
```

## 深拷贝

```
var deepCopy = function(obj) {
  if (typeof obj !== 'object') return;
  var newObj = obj instanceof Array ? [] : {};
  for (var key in obj) {
    if (obj.hasOwnProperty(key)) {
      newObj[key] = typeof obj[key] === 'object' ? deepCopy(obj[key]) :
obj[key];
    }
  }
  return newObj;
}
```

日期 正则拷贝

```
Date.prototype.clone=function(){
  return new Date(this.valueOf());
}

var date=new Date('2010');
var newDate=date.clone();
// newDate->  Fri Jan 01 2010 08:00:00 GMT+0800 (中国标准时间)
```

循环引用的问题

JSON.parse( JSON.stringify())不能拷贝函数

```
var arr = ['old', 1, true, ['old1', 'old2'], {old: 1}]
var new_arr = JSON.parse( JSON.stringify(arr));
console.log(new_arr);
```

## js实现拖拽

```
<div class="calculator" id="drag">🍎🍎🍎🍎🍎🍎🍎🍎🍎</div>
<script>
  window.onload = function () {
  //拖拽功能(主要是触发三个事件：onmousedown\onmousemove\onmouseup)
  var drag = document.getElementById('drag');
  //点击某物体时，用drag对象即可，move和up是全局区域，也就是整个文档通用，应该使用document对象而不是drag对象(否则，采用drag对象时物体只能往右方或下方移动)
  drag.onmousedown = function (e) {
    var e = e || window.event; //兼容ie浏览器
    var diffX = e.clientX - drag.offsetLeft; //鼠标点击物体那一刻相对于物体左侧边框的距离=点击时的位置相对于浏览器最左边的距离-物体左边框相对于浏览器最左边的距离
    var diffY = e.clientY - drag.offsetTop;
    /*低版本ie bug:物体被拖出浏览器可是窗口外部时，还会出现滚动条，
                  解决方法是采用ie浏览器独有的2个方法setCapture()\releaseCapture(),这两个方法，
                  可以让鼠标滑动到浏览器外部也可以捕获到事件，而我们的bug就是当鼠标移出浏览器的时候，
                  限制超过的功能就失效了。用这个方法，即可解决这个问题。注：这两个方法用于onmousedown和onmouseup中*/
    if (typeof drag.setCapture != 'undefined') {
      drag.setCapture();
    }
    document.onmousemove = function (e) {
      var e = e || window.event; //兼容ie浏览器
      var left = e.clientX - diffX;
      var top = e.clientY - diffY;
      //控制拖拽物体的范围只能在浏览器视窗内，不允许出现滚动条
      if (left < 0) {
        left = 0;
      } else if (left > window.innerWidth - drag.offsetWidth) {
        left = window.innerWidth - drag.offsetWidth;
      }
      if (top < 0) {
        top = 0;
      } else if (top > window.innerHeight - drag.offsetHeight) {
        top = window.innerHeight - drag.offsetHeight;
      }
      //移动时重新得到物体的距离，解决拖动时出现晃动的现象
```

```
      drag.style.left = left + 'px';
      drag.style.top = top + 'px';
    };
    document.onmouseup = function (e) { //当鼠标弹起来的时候不再移动
      this.onmousemove = null;
      this.onmouseup = null; //预防鼠标弹起来后还会循环（即预防鼠标放上去的时候还会移
动）
      //修复低版本ie bug
      if (typeof drag.releaseCapture != 'undefined') {
        drag.releaseCapture();
      }
    };
  };
};
</script>
```

# Vue

## vue实现原理

**vue2源码分析（一）**

首先要把那张图给讲明白

defineProperty兼容问题 以及解决方案提出来

**vue proxy**

新的数据监听系统带来了初始化速度加倍同时内存占用减半的效果

vue的初始化过程，有三座大山，分别为Observer、Compiler和Watcher，当我们new Vue的时候，会调用Observer，通过Object.defineProperty来对vue对象的data，computed或者props（如果是组件的话）的所有属性进行监听，同时通过compiler解析模板指令，解析到属性后就new一个Watcher并绑定更新函数到watcher当中，Observer和Compiler就通过属性来进行关联，如此，当Observer中的setter检测到属性值改变的时候，就调用属性对应的所有watcher，调用更新函数，从而更新到属性对应的dom。

**为啥慢**

1、**Object.defineProperty需要遍历所有的属性，这就造成了如果vue对象的data/computed/props中的数据规模庞大，那么遍历起来就会慢很多。** 2、**同样，如果vue对象的data/computed/props中的数据规模庞大，那么Object.defineProperty需要监听所有的属性的变化，那么占用内存就会很大。**

**Object.defineProperty VS Proxy**

defineProperty

1、无法监听es6的Set、WeakSet、Map、WeakMap的变化； 2、无法监听Class类型的数据； 3、属性的新加或者删除也无法监听； 4、数组元素的增加和删除也无法监听。

```
methods:{
  change(){
    this.aas[3] = 444;
    // 在全局对象上通知
    Vue.set(this.aas,3,this.aas[3])
  }
}
```

proxy

对IE不友好,所以vue3在检测到如果是使用IE的情况下（没错，IE11都不支持Proxy），会自动降级为Object.defineProperty的数据监听系统

# vue-router原理

Vue-router 监听浏览器的pushState(h5属性)，如果支持返回pushState，走/斜杠的形式，如果不支持走hash模式（监听hash change完事）

**第一步**

Vue是使用.use( plugins )方法将插件注入到Vue中。

use方法会检测注入插件VueRouter内的install方法，如果有，则执行install方法。

如果是在浏览器环境，在index.js内会自动调用.use方法。如果是基于node环境，需要手动调用。

```
if (inBrowser && window.Vue) {
  window.Vue.use(VueRouter)
}
```

**第二步**

Install解析 (对应目录结构的install.js)，该方法内主要做了以下三件事：

1、对Vue实例混入beforeCreate钩子操作（在Vue的生命周期阶段会被调用）

2、通过Vue.prototype定义router、route 属性（方便所有组件可以获取这两个属性）

3、Vue上注册router-link和router-view两个组件

# 组件通讯

https://segmentfault.com/a/1190000019208626?utm_source=tag-newest#articleHeader12

常见使用场景可以分为三类：

- 父子通信：

父向子传递数据是通过 props，子向父是通过 events（`$emit`）；通过父链 / 子链也可以通信（`$parent` / `$children`）；ref 也可以访问组件实例；provide / inject API；`$attrs/$listeners`

- 兄弟通信：

Bus；Vuex

- 跨级通信：

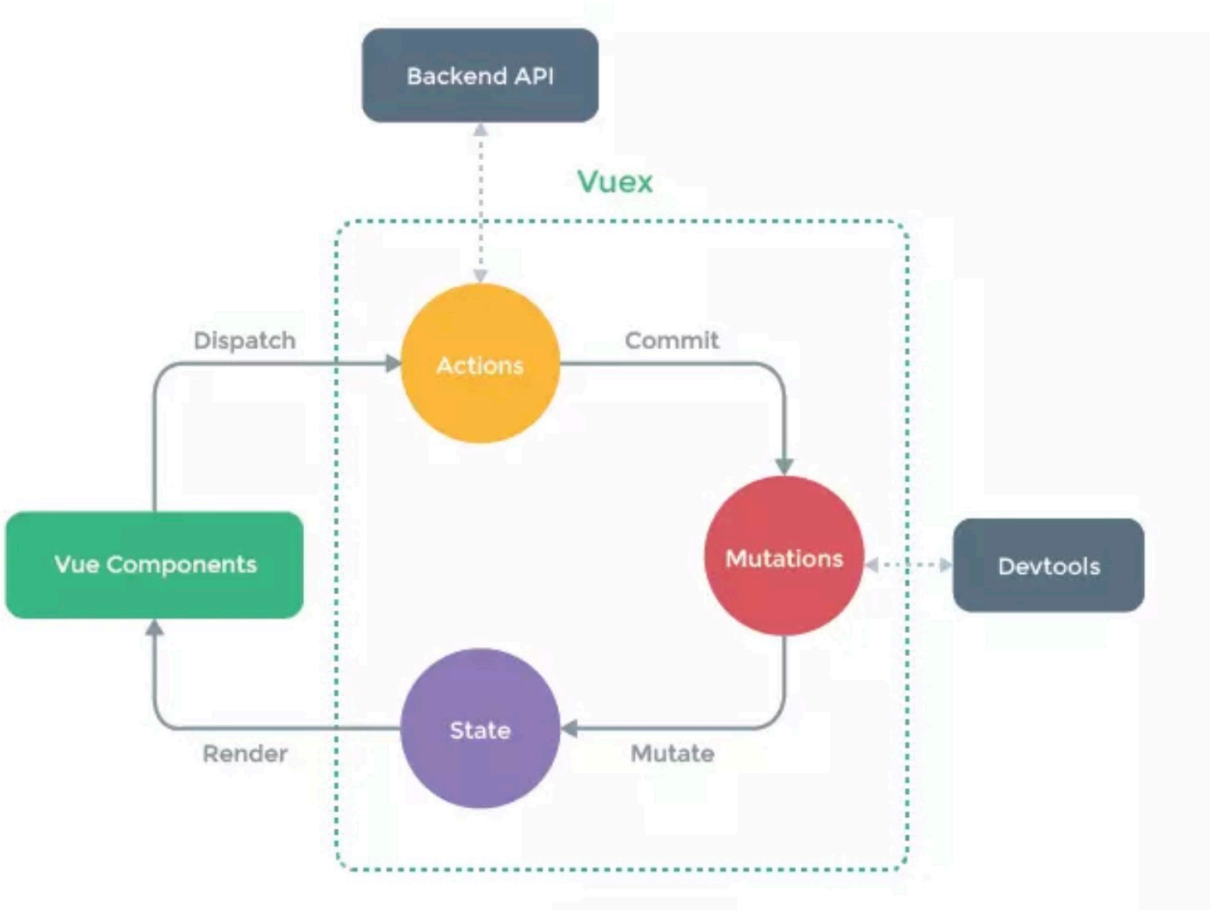
Bus；Vuex；provide / inject API、`$attrs/$listeners`

## vuex

容易忽略的api：mapState mapActions mapMutations

使用：https://juejin.im/post/58fffc52a22b9d0065b8db53#heading-15

源码：https://juejin.im/post/59f66bd7f265da432d275d30#heading-0




将数据层与组件层抽离,把数据层放到全局形成一个单一的Store，组件层变得更薄，专门用来进行数据的展示及操作。所有数据的变更都需要经过全局的Store来进行，形成一个单向数据流，使数据变化变得“可预测”。

Vuex运行依赖Vue内部数据双向绑定机制，需要new一个Vue对象来实现“响应式化”，所以Vuex是一个专门为Vue.js设计的状态管理库。

```
Vue.use(Vuex);

/*将store放入Vue创建时的option中*/
new Vue({
    el: '#app',
    store
});
```

1、vuex install 一件是防止Vuex被重复安装 另一件是执行applyMixin,将vuexinit混淆进Vue的beforeCreacte钩子中

2、vuexinit vuexInit会尝试从options中获取store，如果当前组件是根组件（Root节点），则options中会存在store。如果当前组件非根组件，则通过options中的parent获取父组件的$store引用。

所有的组件都获取到了同一份内存地址的Store实例，于是我们可以在每一个组件中通过this.$store愉快地访问全局的Store实例了

## watch computed

**watch擅长处理的场景：一个数据影响多个数据**

**computed擅长处理的场景：一个数据受多个数据影响**

# 正则

https://www.runoob.com/jsref/jsref-obj-regexp.html

https://juejin.im/post/5965943ff265da6c30653879#heading-9

## 匹配字符

**\d**就是`[0-9]`。表示是一位数字。记忆方式：其英文是digit（数字）。

**\D**就是`[^0-9]`。表示除数字外的任意字符。

**\w**就是`[0-9a-zA-Z_]`。表示数字、大小写字母和下划线。记忆方式：w是word的简写，也称单词字符。

**\W**是`[^0-9a-zA-Z_]`。非单词字符。

**\s**是`[ \t\v\n\r\f]`。表示空白符，包括空格、水平制表符、垂直制表符、换行符、回车符、换页符。记忆方式：s是space character的首字母。

**\S**是`[^ \t\v\n\r\f]`。 非空白符。

**.** 就是`[^\n\r\u2028\u2029]`。通配符，表示几乎任意字符。换行符、回车符、行分隔符和段分隔符除外。记忆方式：想想省略号...中的每个点，都可以理解成占位符，表示任何类似的东西。

**{m,}** 表示至少出现m次。

**{m}** 等价于{m,m}，表示出现m次。

**?** 等价于{0,1}，表示出现或者不出现。记忆方式：问号的意思表示，有吗？

**+** 等价于`{1,}`，表示出现至少一次。记忆方式：加号是追加的意思，得先有一个，然后才考虑追加。

***** 等价于`{0,}`，表示出现任意次，有可能不出现。记忆方式：看看天上的星星，可能一颗没有，可能零散有几颗，可能数也数不过来。

| | |
|---|---|
| **n+** | **匹配任何包含至少一个 n 的字符串。例如，/a+/ 匹配 "candy" 中的 "a"，"caaaaaaandy" 中所有的 "a"。** |
| n* | 匹配任何包含零个或多个 n 的字符串。例如，/bo*/ 匹配 "A ghost booooed" 中的 "boooo"，"A bird warbled" 中的 "b"，但是不匹配 "A goat grunted"。 |
| n? | 匹配任何包含零个或一个 n 的字符串。例如，/e?le?/ 匹配 "angel" 中的 "el"，"angle" 中的 "le"。 |
| n{X} | 匹配包含 X 个 n 的序列的字符串。例如，/a{2}/ 不匹配 "candy," 中的 "a"，但是匹配 "caandy," 中的两个 "a"，且匹配 "caaandy." 中的前两个 "a"。 |
| n{X,} | X 是一个正整数。前面的模式 n 连续出现至少 X 次时匹配。例如，/a{2,}/ 不匹配 "candy" 中的 "a"，但是匹配 "caandy" 和 "caaaaaaandy." 中所有的 "a"。 |
| n{X,Y} | X 和 Y 为正整数。前面的模式 n 连续出现至少 X 次，至多 Y 次时匹配。例如，/a{1,3}/ 不匹配 "cndy"，匹配 "candy," 中的 "a"，"caandy," 中的两个 "a"，匹配 "caaaaaaandy" 中的前面三个 "a"。注意，当匹配 "caaaaaaandy" 时，即使原始字符串拥有更多的 "a"，匹配项也是 "aaa"。 |
| n$ | 匹配任何结尾为 n 的字符串。 |
| ^n | 匹配任何开头为 n 的字符串。 |
| ?=n | 匹配任何其后紧接指定字符串 n 的字符串。 |
| ?!n | 匹配任何其后没有紧接指定字符串 n 的字符串。 |

## 贪婪匹配和惰性匹配

```
var regex = /\d{2,5}/g;
var string = "123 1234 12345 123456";
console.log( string.match(regex) );
// => ["123", "1234", "12345", "12345"]

var regex = /\d{2,5}?/g;
var string = "123 1234 12345 123456";
console.log( string.match(regex) );
// => ["12", "12", "34", "12", "34", "12", "34", "56"]
```

## 获取元素id

```
var regex = /id="[^"]*"/
var string = '<div id="container" class="main"></div>';
console.log(string.match(regex)[0]);
```

## 匹配位置

### ^ $

```
var result = "I\nlove\njavascript".replace(/^|$/gm, '#');
console.log(result);
/*
#I#
#love#
#javascript#
*/
```

### 千分位

```
var reg = /(?!^)(?=(\d{3})+$)/g;
var result = string1.replace(reg, ',')
console.log(result);
// => "12,345,678"
```

### 验证码

```
//同时包含具体两种字符
//比如同时包含数字和小写字母，可以用(?=.*[0-9])(?=.*[a-z])来做。
var reg = /(?=.*[0-9])(?=.*[a-z])^[0-9A-Za-z]{6,12}$/;
```

## 正则表达式括号的作用

### yyyy-mm-dd格式，替换成mm/dd/yyyy

```
var regex = /(\d{4})-(\d{2})-(\d{2})/;
var string = "2017-06-12";
var result = string.replace(regex, "$2/$3/$1");
//return RegExp.$2 + "/" + RegExp.$3 + "/" + RegExp.$1;`
console.log(result);
```

## 手机号

```
/^1(38|53|39)\d{8}$/
```

## trim()

```
/^(\s*)|(\s*)$/g
```

### 输出文件后缀

```
let str = './doucment/data/a.jpg';
let str2 = './Data/user/hafda.txt';
var reg = /(\.[^\.]+)$/;
let obj = str2.match(reg);
console.log(obj[1])
```

## 设计模式

### 订阅者

```
//主题对象
function Dep() {
  //订阅者列表
  this.subs = [];
}
//主题对象通知订阅者
Dep.prototype.notify = function () {
  //遍历所有的订阅者，执行订阅者提供的更新方法
  this.subs.forEach(function (sub) {
    sub.update();
  })
}
//订阅者
function Sub(x) {
  this.x = x;
}
//订阅者更新
Sub.prototype.update = function () {
  this.x = this.x + 1;
  console.log(this.x)
}
//发布者
var pub = {
  publish: function () {
    dep.notify();
  }
}
//主题对象实例
var dep = new Dep();
//新增2个订阅者
dep.subs.push(new Sub(1), new Sub(2));
//发布者发布更新
pub.publish();
```

## 单例

```
var getSingle = function (fn) {
  var result;
  return function () {
    return result || ( result = fn.apply(this, arguments) );
  }
};
```

# 算法

## 斐波那契

```
function Fb(n, n1 = 1, n2 = 1) {
  if (n <= 1) { return n1 };
  return Fb(n - 1, n2, n1 + n2);
}
console.log(Fb(6));
```

时间复杂度：**O(N)**

空间复杂度：**O(1)**

```
function fb3(n) {
  let num1 = 1,
      num2 = 2,
      num3 = 3;
  if (n < 1)
    return -1;
  if (n == 1 || n == 2)
    return 1;
  for (let i = 3; i < n; i++) {
    num3 = num1 + num2;
    num1 = num2;
    num2 = num3;
  }
  return num3
}
console.log(fb3(12));//144
```

## 反转二叉树

```
var obj = {
  'id': '4',
```

```
  'left': {
    'id': '2',
    'left': {
      'id': '1',
      'left': null,
      'right': null
    },
    'right': {
      'id': '3',
      'left': null,
      'right': null
    }
  },
  'right': {
    'id': '7',
    'left': {
      'id': '6',
      'left': null,
      'right': null
    },
    'right': {
      'id': '9',
      'left': null,
      'right': null
    }
  }
}
console.log(obj);

function invertTree(root) {
  if (root !== null) {
    var temp = root.left;
    root.left = root.right;
    root.right = temp;
    invertTree(root.left);
    invertTree(root.right);
  }
  return root
};
invertTree(obj)
```

# 排序

## 冒泡排序

```
//思路：先比较一轮一次，然后用for循环比较一轮多次，然后再加for循环比较多轮多次
//从大到小排序
```

```
//比较轮数
function bubbleSort(arr){
  var len = arr.length;
  for (var i = 0; i < len - 1; i++) {
    //每轮比较次数，次数=长度-1-此时的轮数
    for (var j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
        var temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      } //end if
    }//end for 次数
  } //end for 轮数
  return arr;
}
```

### 选择排序

```
function selectSort(arr) {
  var len = arr.length;
  var minIndex, temp;
  for (var i = 0; i < len - 1; i++) {
    minIndex = i;
    for (var j = i + 1; j < len; j++) {
      if (arr[j] < arr[minIndex]) { //寻找最小的数
        minIndex = j;               //将最小的索引保存
      }
    }
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
  return arr;
}
```

### 插入排序 (打牌)

```
function insertSort(arr) {
  var len = arr.length;
  var preIndex, current;
  for (var i = 1; i < len; i++) {
    preIndex = i - 1;
    current = arr[i];
    while (preIndex >= 0 && current < arr[preIndex]) {
      arr[preIndex+1] = arr[preIndex]
      preIndex--;
    }
    arr[preIndex+1] = current
```

```
  }
  return arr;
}
```

### 希尔排序

```
var arr = [3, 2, 4, 7, 1, 5, 9, 6, 8];
var gops = [5, 3, 1];
for(var g=0; g<gops.length;g++) {
  for(var i=gops[g]; i<arr.length; i++){
    var temp=arr[i];
    for(var j=i; j>=gops[g] && arr[j-gops[g]]>temp;j-=gops[g]){
      arr[j] = arr[j-gops[g]];
    }
    arr[j] = temp
  }
}
```

### 快速排序

```
function quickSort(arr) {
  if (arr.length == 0) {
    return [];
  }
  var pivot = arr[0];
  var lesser = [];
  var greater = [];
  for (var i = 1; i < arr.length; i++) {
    if (arr[i] < pivot) {
      lesser.push(arr[i])
    } else {
      greater.push(arr[i])
    }
  }
  return quickSort(lesser).concat(pivot,quickSort(greater));
}
```

## 数组去重

```
const arr = [];
// 生成[0, 100000]之间的随机数
for (let i = 0; i < 100000; i++) {
  arr.push(0 + Math.floor((100000 - 0 + 1) * Math.random()))
}

// ...实现算法

console.time('test');
arr.unique();
console.timeEnd('test');
```

## indexOf

```
Array.prototype.unique = function () {
  const newArray = [];
  this.forEach(item => {
    if (newArray.indexOf(item) === -1) {
      newArray.push(item)
    }
  })
  return newArray;
}

Array.prototype.unique = function() {
  return this.filter((item,index)=>{
    return this.indexOf(item) === index
  })
}

test1: 3766.324951171875ms
test2: 4887.201904296875ms
```

## sort

```
Array.prototype.unique = function () {
  const newArray = [];
  this.sort();
  for (let i = 0; i < this.length; i++) {
    if (this[i] != this[i + 1]) {
      newArray.push(this[i])
    }
  }
  return newArray;
}
test: 4300.39990234375ms
```

```
Array.prototype.unique = function () {
  const newArray = [];
  this.sort();
  for (let i = 0; i < this.length; i++) {
    if (this[i] != newArray[newArray.length-1]) {
      newArray.push(this[i])
    }
  }
  return newArray;
}
test1: 121.6259765625ms
test2: 123.02197265625ms
```

## hash

- 无法区分隐式类型转换成字符串后一样的值，比如 1 和 '1'
- 无法处理复杂数据类型，比如对象（因为对象作为 key 会变成 [object Object]）
- 特殊数据，比如 '**proto**'，因为对象的 **proto** 属性无法被重写

```
Array.prototype.unique = function () {
  const newArray = [];
  const tmp = {};
  for (let i = 0; i < this.length; i++) {
    if (!tmp[typeof this[i] + JSON.stringify(this[i])]) {
      tmp[typeof this[i] + JSON.stringify(this[i])] = 1;
      newArray.push(this[i]);
    }
  }
  return newArray;
}
```

## Map

```
Array.prototype.unique = function () {
  const tmp = new Map();
  return this.filter(item => {
    return !tmp.has(item) && tmp.set(item, 1)
  })
}
```

## includes

```
Array.prototype.unique = function () {
  const newArray = [];
  this.forEach(item => {
    if (!newArray.includes(item)) {
      newArray.push(item);
    }
  });
  return newArray;
}
```

## 找出数组中出现次数最多的元素

```
function findMost(arr) {
  var h = {};
  var maxNum = 0;
  var maxEle = null;
  for (var i = 0; i < arr.length; i++) {
    h[arr[i]] == undefined ? h[arr[i]] = 1 : h[arr[i]]++;
    if (h[arr[i]] > maxNum) {
      maxEle = arr[i];
      maxNum = h[arr[i]]
    }
  }
  console.log(maxEle, maxNum)
}

findMost(arr)
```

## 二分查找

```
function findValue(arr,value) {
  var low = 0,
      high = arr.length-1;
  while(low<=hight) {
    var mid = parseInt((low+high)/2);
    if(arr[mid] = value) {
      return mid;
    }else if(arr[mid]<value) {
      low = mid +1;
    }else if(arr[mid]>value) {
      high = mid -1
    }else{
      return -1;
    }
  }
}
```