数组的 includes 方法

```
function printAnimals(animal) {
  // if (animal === 'dog' || animal === 'cat') {
  //   console.log(`I have a ${animal}`)
  // }
  const animals = ['dog', 'cat', 'hamster']
  if (animals.includes(animal)) {
    console.log(animal)
  }
}

printAnimals('dog')
```

提前退出提前返回

```
function printAnimalsDetails(animal) {
  var result = null
  if (animal) {
    if (animal.type) {
      if (animal.name) {
        if (animal.gender) {
          result = `${animal.name} is a ${animal.gender} ${animal.type}`
        } else {
          result = 'no animal gender'
        }
      } else {
        result = 'no animal name'
      }
    } else {
      result = 'no animal type'
    }
  } else {
    result = 'no animal'
  }
  return result
}
```

```
const printAnimalsDetails = ({ type, name, gender } = {}) => {
  if (!type) return 'no animal type'
  if (!name) return 'no animal name'
  if (!gender) return 'no animal gender'
  return `${name} is a ${gender}  ${type}`
}
```

对象的字面量代替 switch 方法

```
// 基于颜色打印水果

// function printFruits(color) {
//   switch (color) {
//     case 'red':
//       return ['apple']
//     case 'yellow':
//       return ['banana']
//     default:
//       return []
//   }
// }

// console.log(printFruits(null))
// console.log(printFruits('yellow'))


// const

const fruitsColor = {
  red: ['apple'],
  yellow: ['banana']
}

function printFruits(color) {
  return fruitsColor[color] || []
}

console.log(printFruits(null))
console.log(printFruits('yellow'))
```

map 的使用

```
var obj1 = {
  name: 'zs'
}
var obj2 = {
  name: 'ls'
}

var obj3 = {
  [obj2]: '22',
  [obj1]: '11',
}

console.log(obj3)
```

```
const fruitsColor = new Map().set('red', ['apple']).set('yellow', ['banana']
)

function printFruits(color) {
  return fruitsColor.get(color) || []
}

console.log(printFruits(null))
console.log(printFruits('yellow'))
```

▸ {[object Object]: "11"}

Array.some 和 Array.every.、

```
// 上需求 检测是否所有的水果都是红色

const fruits = [
  { name: 'apple', color: 'red' },
  { name: 'banana', color: 'yellow' },
]


// function test() {
//   const isAllRed = fruits.every(f => f.color == 'red')
//   console.log(isAllRed)
// }

// test()
// 上需求 检测是否有红色的水果


function test() {
  const isAllRed = fruits.some(f => f.color == 'red')
  console.log(isAllRed)
}

test()
```

es6 对象解构基本用法及注意点

```
let person = {
  name: '向川',
  age: 20,
}
let { age: age, hobby, name: name1 } = person
console.log(name1, age, hobby);


// let name1 = person.name
// console.log(name1);

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

wps 秋招笔试题

```
// 实现一个方法 能够在控制台打印当前的时间 时间间隔是一秒 时间格式是 YYYY-MM-DD
hh:mm:ss
// wps秋招2021年笔试题

setInterval(() => {
  console.log('当前时间是' + timer());
}, 1000)
function timer() {
  // 能够在控制台打印当前的时间
  var date = new Date()
  var year = date.getFullYear()
  var moth = (date.getMonth() + 1).toString().padStart(2, "0")// 9 09
  var day = date.getDate().toString().padStart(2, "0") // 14 9 09

  var hour = date.getHours().toString().padStart(2, "0")
  var min = date.getMinutes().toString().padStart(2, "0")
  var hour = date.getSeconds().toString().padStart(2, "0")

  return `${year}-${moth}-${day} ${hour}:${min}:${hour}`

}
```

箭头函数注意点

注意点

- 函数体内的this
- 不可以当做构造函数
- arguments 是不可以的
- 箭头函数不可以当做generator函数

展开运算符操作数组

```
# 合并数组
```js
    var arr1 = ['a', 'b'];
    var arr2 = ['c'];
    var arr3 = ['d', 'e'];
    // ES5 的合并数组
    console.log(arr1.concat(arr2, arr3));
    // ES6 的合并数组
    console.log([...arr1, ...arr2, ...arr3]);
```

与解构赋值结合
```js
    const [first, ...rest] = [1, 2, 3, 4, 5];
    console.log(first); // 1
    console.log(rest); // [2, 3, 4, 5]
    const [first, ...rest] = [];
    console.log(first); // undefined
    console.log(rest); // []:
    const [first, ...rest] = ["foo"];
    console.log(first); // "foo"
    console.log(rest); // []

```

字符串转化成数组

```js

// 扩展运算符还可以将字符串转为真正的数组。
```

```
[...'hello']
// [ "h", "e", "l", "l", "o" ]
```
```

与迭代器接口的联系，展开运算符操作对象

```
# 展开运算符 展开的是 具有迭代器接口的有Symbol.iterator属性的
```js
    var arr = [1, 2, 3, 4]
    let iterator = arr[Symbol.iterator]()
    console.log(iterator.next());
    console.log(iterator.next());
    console.log(iterator.next());
    console.log(iterator.next());
    console.log(iterator.next());
```

展开运算符 对对象的运用 （ES7草案中的对象展开运算符）

- 配合对象的解构

```js
let {x,y,...z}={x:1,y:2,a:3,b:4};

```

- 另外还可以像数组那样将一个对象插入到另外一个对象当中
```

替代 apply

```
# 替代数组的apply的方法
```js
    function f(x, y, z) {
      return x + y + z
    }
    var args = [0, 1, 2]
    console.log(f.apply(null, args));
    console.log(f(...args));
```

- 求数组最大的元素
```js
  console.log(Math.max.apply(null, [1, 2, 3]));
  console.log(Math.max(...[1, 2, 3]));
```
- 将一个数组添加到另一个数组的尾部
```js
var arr1 = [0,1,2]
var arr2 = [3,4,5]
Array.prototype.push.apply(arr1,arr2)

var arr1 = [0,1,2]
var arr2 = [3,4,5]
arr1.push(...arr2)

```
```

```js
let z={a:3,b:4};
let n={x:1,y:2,...z};

```

- 还可以合并两个对象

```js
let a={x:1,y:2};
let b={z:3};
let ab={...a,...b};
ab //{x:1,y:2,z:3}

```

- 但是注意 不能单独对对象进行解构 因为对象没有部署 Symbol.iterator的接口

腾讯笔试题
- 我们能否以某种方式为下面的语句使用展开运算而不导致类型错误?

错误代码示例

```js
    let obj = { x: 1, y: 2, z: 3 }
```

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

// Symbol.iterator
// 为每一个对象定义了默认的迭代器。该迭代器可以被 for…of 循环使用。

// Reflect.ownKeys
// 静态方法 Reflect.ownKeys() 返回一个由目标对象自身的属性键组成的数组。

obj[Symbol.iterator] = function () {
  return {
    next: function () {
      let objArr = Reflect.ownKeys(obj)
      if (this.index < objArr.length - 1) {
        let key = objArr[this.index]
        this.index++
        return { value: obj[key] }
      } else {
        return { done: true }
      }
    },
    index: 0
  }
}
console.log([...obj]);
```

圣杯布局

1. DOM结构

```
<div id="container">
   <div id="center" class="column"></div>
   <div id="left" class="column"></div>
   <div id="right" class="column"></div>
 </div>
```

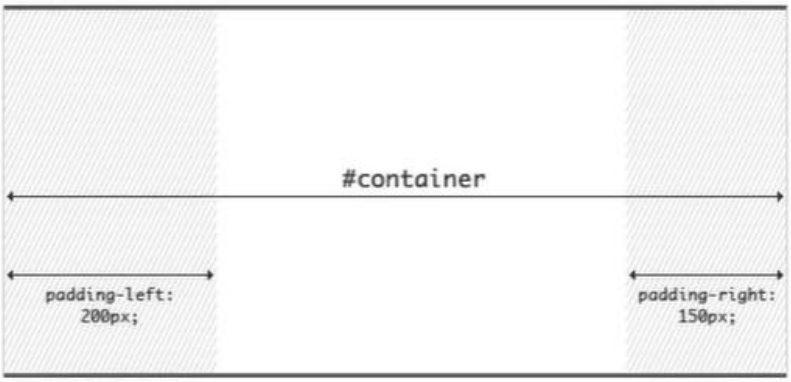
首先定义出整个布局的DOM结构，主体部分是由`container`包裹的`center`,`left`,`right`三列，其中`center`定义在最前面。

2. CSS代码

假设左侧的固定宽度为200px，右侧的固定宽度为150px，则首先在`container`上设置：

```
#container {
  padding-left: 200px;
  padding-right: 150px;
}
```

为左右两列预留出相应的空间，得到如下示意图：



随后分别为三列设置宽度与浮动

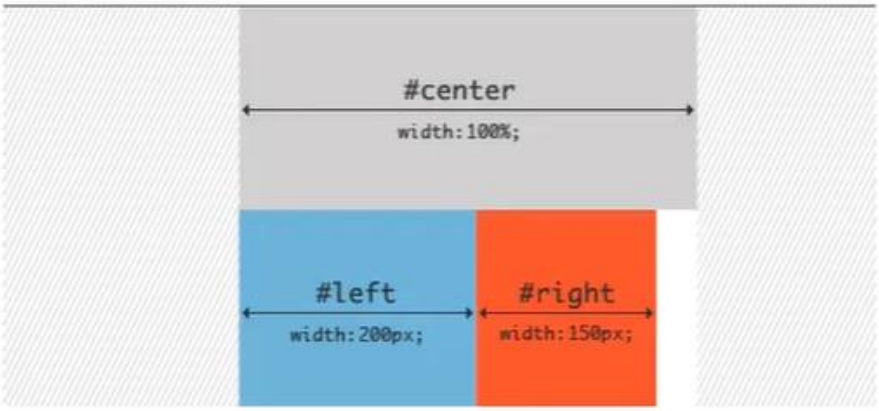
```
#container .column {
  float: left;
}

#center {
  width: 100%;
}

#left {
  width: 200px;
}

#right {
  width: 150px;
}
```

得到如下效果：

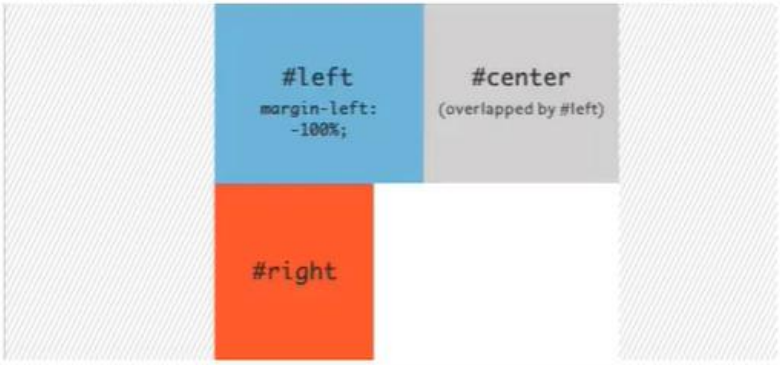


根据浮动的特性，由于`center`的宽度为100%，即占据了第一行的所有空间，所以`left`和`right`被“挤”到了第二行。

接下来的工作是将`left`放置到之前预留出的位置上，这里使用负外边距

```css
#left {
  width: 200px;
  margin-left: -100%;
}
```

得到：




```css
#left {
  width: 200px;
  height: 200px;
  background-color: pink;
  margin-left: -100%;
  position: relative;
  right: 200px;
}

#right {
  width: 150px;
  height: 200px;
  background-color: purple;
  margin-right: -150px;
}
```

flex 实现圣杯布局

```
.container {
  display: flex;
}

.left {
  width: 200px;
  height: 200px;
  background-color: pink;
}

.right {
  width: 150px;
  height: 200px;
  background-color: purple;
}

.center {
  flex: 1;
  height: 200px;
  background-color: powderblue;
}
```

区块

“编程风格”（programming style）指的是编写代码的样式规则。不同的程序员，往往有不同的编程风格。

有人说，编译器的规范叫做“语法规则”（grammar），这是程序员必须遵守的；而编译器忽略的部分，就叫“编程风格”（programming style），这是程序员可以自由选择的。这种说法不完全正确，程序员固然可以自由选择编程风格，但是好的编程风格有助于写出质量更高、错误更少、更易于维护的程序。

所以，编程风格的选择不应该基于个人爱好、熟悉程度、打字量等因素，而要考虑如何尽量使代码清晰易读、减少出错。你选择的，不是你喜欢的风格，而是一种能够清晰表达你的意图的风格。这一点，对于 JavaScript 这种语法自由度很高的语言尤其重要。

必须牢记的一点是，如果你选定了一种“编程风格”，就应该坚持遵守，切忌多种风格混用。如果你加入他人的项目，就应该遵守现有的风格。

缩进

行首的空格和 Tab 键，都可以产生代码缩进效果（indent）。

Tab 键可以节省击键次数，但不同的文本编辑器对 Tab 的显示不尽相同，有的显示四个空格，有的显示两个空格，所以有人觉得，空格键可以使得显示效果更统一。

无论你选择哪一种方法，都是可以接受的，要做的就是始终坚持这一种选择。不要一会使用 Tab 键，一会使用空格键。

区块

如果循环和判断的代码体只有一行，JavaScript 允许该区块（block）省略大括号。

```
if (a)
  b();
  c();
```

上面代码的原意可能是下面这样。

```
if (a) {
  b();
  c();
}
```

但是，实际效果却是下面这样。

```
if (a) {
  b();
}
  c();
```

因此，建议总是使用大括号表示区块。

另外，区块起首的大括号的位置，有许多不同的写法。最流行的有两种，一种是起首的大括号另起一行。

```
block
{
  // ...
}
```

另一种是起首的大括号跟在关键字的后面。

```
block {
  // ...
}
```

一般来说，这两种写法都可以接受。但是，JavaScript 要使用后一种，因为 JavaScript 会自动添加句末的分号，导致一些难以察觉的错误。

```
return
{
  key: value
};

// 相当于
return;
{
  key: value
};
```

上面的代码的原意，是要返回一个对象，但实际上返回的是`undefined`，因为 JavaScript 自动在`return`语句后面添加了分号。为了避免这一类错误，需要写成下面这样。

```
return {
  key : value
};
```

因此，表示区块起首的大括号，不要另起一行。

圆括号

圆括号（parentheses）在 JavaScript 中有两种作用，一种表示函数的调用，另一种表示表达式的组合（grouping）。

```
// 圆括号表示函数的调用
console.log('abc');

// 圆括号表示表达式的组合
(1 + 2) * 3
```

建议可以用空格，区分这两种不同的括号。

1. 表示函数调用时，函数名与左括号之间没有空格。
2. 表示函数定义时，函数名与左括号之间没有空格。
3. 其他情况时，前面位置的语法元素与左括号之间，都有一个空格。

按照上面的规则，下面的写法都是不规范的。

```
foo (bar)
return(a+b);
if(a === 0) {...}
function foo (b) {...}
function(x) {...}
```

上面代码的最后一行是一个匿名函数，`function`是语法关键字，不是函数名，所以与左括号之间应该要有一个空格。

使用分号的情况

行尾的分号

分号表示一条语句的结束。JavaScript 允许省略行尾的分号。事实上，确实有一些开发者行尾从来不写分号。但是，由于下面要讨论的原因，建议还是不要省略这个分号。

不使用分号的情况

首先，以下三种情况，语法规定本来就不需要在结尾添加分号。

（1）for 和 while 循环

```
for ( ; ; ) {
} // 没有分号

while (true) {
} // 没有分号
```

注意，`do...while`循环是有分号的。

```
do {
  a--;
} while(a > 0); // 分号不能省略
```

（2）分支语句：if，switch，try

```
if (true) {
} // 没有分号

switch () {
} // 没有分号

try {
} catch {
} // 没有分号
```

（3）函数的声明语句

```
function f() {
} // 没有分号
```

注意，函数表达式仍然要使用分号。

```
var f = function f() {
};
```

以上三种情况，如果使用了分号，并不会出错。因为，解释引擎会把这个分号解释为空语句。

分号的自动添加

除了上一节的三种情况，所有语句都应该使用分号。但是，如果没有使用分号，大多数情况下，JavaScript 会自动添加。

```
var a = 1
// 等同于
var a = 1;
```

这种语法特性被称为“分号的自动添加”（Automatic Semicolon Insertion，简称 ASI）。

因此，有人提倡省略句尾的分号。麻烦的是，如果下一行的开始可以与本行的结尾连在一起解释，JavaScript 就不会自动添加分号。

```
// 等同于 var a = 3
var
a
=
3

// 等同于 'abc'.length
'abc'
.length

// 等同于 return a + b;
return a +
b;

// 等同于 obj.foo(arg1, arg2);
obj.foo(arg1,
arg2);
```

```
// 等同于 3 * 2 + 10 * (27 / 6)
3 * 2
+
10 * (27 / 6)
```

选择语言

上面代码都会多行放在一起解释，不会每一行自动添加分号。这些例子还是比较容易看出来的，但是下面这个例子就不那么容易看出来了。

```
    // var li = document.getElementsByTagName('li');
    // (function () {
    //   console.log(123);
    // })()
```

```
x = y
(function () {
  // ...
})();

// 等同于
x = y(function () {...})();
```

下面是更多不会自动添加分号的例子。

```
// 引擎解释为 c(d+e)
var a = b + c
(d+e).toString();

// 引擎解释为 a = b/hi/g.exec(c).map(d)
// 正则表达式的斜杠，会当作除法运算符
a = b
/hi/g.exec(c).map(d);

// 解释为'b'['red', 'green'],
// 即把字符串当作一个数组，按索引取值
var a = 'b'
['red', 'green'].forEach(function (c) {
  console.log(c);
})

// 解释为 function (x) { return x }(a++)
// 即调用匿名函数，结果f等于0
var a = 0;
var f = function (x) { return x }
(a++)
```

只有下一行的开始与本行的结尾，无法放在一起解释，JavaScript 引擎才会自动添加分号。

```
if (a < 0) a = 0
console.log(a)

// 等同于下面的代码，
// 因为 0console 没有意义
if (a < 0) a = 0;
console.log(a)
```

另外，如果一行的起首是“自增”（`++`）或“自减”（`--`）运算符，则它们的前面会自动添加分号。

```
a = b = c = 1

a
++
b
--
c

console.log(a, b, c)
// 1 2 0
```

上面代码之所以会得到`1 2 0`的结果，原因是自增和自减运算符前，自动加上了分号。上面的代码实际上等同于下面的形式。

```
a = b = c = 1;
a;
++b;
--c;
```

如果`continue`、`break`、`return`和`throw`这四个语句后面，直接跟换行符，则会自动添加分号。这意味着，如果`return`语句返回的是一个对象的字面量，起首的大括号一定要写在同一行，否则得不到预期结果。

```
return
{ first: 'Jane' };

// 解释成
return;
{ first: 'Jane' };
```

由于解释引擎自动添加分号的行为难以预测，因此编写代码的时候不应该省略行尾的分号。

不应该省略结尾的分号，还有一个原因。有些 JavaScript 代码压缩器（uglifier）不会自动添加分号，因此遇到没有分号的结尾，就会让代码保持原状，而不是压缩成一行，使得压缩无法得到最优的结果。

另外，不写结尾的分号，可能会导致脚本合并出错。所以，有的代码库在第一行语句开始前，会加上一个分号。

```
;var a = 1;
// ...
```

上面这种写法就可以避免与其他脚本合并时，排在前面的脚本最后一行语句没有分号，导致运行出错的问题。

全局变量

JavaScript 最大的语法缺点，可能就是全局变量对于任何一个代码块，都是可读可写。这对代码的模块化和重复使用，非常不利。

因此，建议避免使用全局变量。如果不得不使用，可以考虑用大写字母表示变量名，这样更容易看出这是全局变量，比如`UPPER_CASE`。

变量声明

JavaScript 会自动将变量声明“提升”（hoist）到代码块（block）的头部。

```
if (!x) {
  var x = {};
}

// 等同于
var x;
if (!x) {
  x = {};
}
```

这意味着，变量`x`是`if`代码块之前就存在了。为了避免可能出现的问题，最好把变量声明都放在代码块的头部。

```
for (var i = 0; i < 10; i++) {
  // ...
}

// 写成
var i;
for (i = 0; i < 10; i++) {
  // ...
}
```

上面这样的写法，就容易看出存在一个全局的循环变量`i`。

另外，所有函数都应该在使用之前定义。函数内部的变量声明，都应该放在函数的头部。

with 语句

`with`可以减少代码的书写，但是会造成混淆。

```
with (o) {
  foo = bar;
}
```

上面的代码，可以有四种运行结果：

```
o.foo = bar;
o.foo = o.bar;
foo = bar;
foo = o.bar;
```

这四种结果都可能发生，取决于不同的变量是否有定义。因此，不要使用`with`语句。

相等和严格相等

JavaScript 有两个表示相等的运算符：“相等”（`==`）和“严格相等”（`===`）。

相等运算符会自动转换变量类型，造成很多意想不到的情况。

```
0 == ''// true
1 == true // true
2 == true // false
0 == '0' // true
false == 'false' // false
false == '0' // true
' \t\r\n ' == 0 // true
```

因此，建议不要使用相等运算符（`==`），只使用严格相等运算符（`===`）。

语句的合并

有些程序员追求简洁，喜欢合并不同目的的语句。比如，原来的语句是

```
a = b;
if (a) {
  // ...
}
```

他喜欢写成下面这样。

```
if (a = b) {
  // ...
}
```

虽然语句少了一行，但是可读性大打折扣，而且会造成误读，让别人误解这行代码的意思是下面这样。

```
if  (a === b) {
  // ...
}
```

建议不要将不同目的的语句，合并成一行。

自增和自减运算符

自增（`++`）和自减（`--`）运算符，放在变量的前面或后面，返回的值不一样，很容易发生错误。事实上，所有的`++`运算符都可以用`+= 1`代替。

```
++x
// 等同于
x += 1;
```

改用`+= 1`，代码变得更清晰了。

建议自增（`++`）和自减（`--`）运算符尽量使用`+=`和`-=`代替。

switch...case 结构

`switch...case`结构要求，在每一个`case`的最后一行必须是`break`语句，否则会接着运行下一个`case`。这样不仅容易忘记，还会造成代码的冗长。

而且，`switch...case`不使用大括号，不利于代码形式的统一。此外，这种结构类似于`goto`语句，容易造成程序流程的混乱，使得代码结构混乱不堪，不符合面向对象编程的原则。

```
function doAction(action) {
  switch (action) {
    case 'hack':
      return 'hack';
    case 'slash':
      return 'slash';
    case 'run':
      return 'run';
    default:
      throw new Error('Invalid action.');
  }
}
```

上面的代码建议改写成对象结构。

```
function doAction(action) {
  var actions = {
    'hack': function () {
      return 'hack';
    },
    'slash': function () {
      return 'slash';
    },
    'run': function () {
      return 'run';
    }
  };

  if (typeof actions[action] !== 'function') {
    throw new Error('Invalid action.');
  }

  return actions[action]();
}
```

因此，建议`switch...case`结构可以用对象结构代替。

web 安全-csrf 攻击

什么是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。

几种常见的攻击类型

- GET类型的CSRF：
  GET类型的CSRF利用非常简单，只需要一个HTTP请求，一般会这样利用：

```
<img src="http://bank.example/withdraw?amount=10000&for=hacker" >
```

在受害者访问含有这个img的页面后，浏览器会自动向http://bank.example/withdraw?account=xiaoming&amount=10000&for=hacker发出一次HTTP请求。bank.example就会收到包含受害者登录信息的一次跨域请求。

- POST类型的CSRF：

这种类型的CSRF利用起来通常使用的是一个自动提交的表单，如：

```
<form action="http://bank.example/withdraw" method=POST>
    <input type="hidden" name="account" value="xiaoming" />
    <input type="hidden" name="amount" value="10000" />
    <input type="hidden" name="for" value="hacker" />
</form>
<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。

POST类型的攻击通常比GET要求更加严格一点，但仍并不复杂。任何个人网站、博客，被黑客上传页面的网站都有可能是发起攻击的来源，后端接口不能将安全寄托在仅允许POST上面。

- 链接类型的CSRF

链接类型的CSRF并不常见，比起其他两种用户打开页面就中招的情况，这种需要用户点击链接才会触发。这种类型通常是在论坛中发布的图片中嵌入恶意链接，或者以广告的形式诱导用户中招，攻击者通常会以比较夸张的词语诱骗用户点击，例如：

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

由于之前用户登录了信任的网站A，并且保存登录状态，只要用户主动访问上面的这个PHP页面，则表示攻击成功。

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。
  CSRF通常是跨域的，因为外域通常更容易被攻击者掌控。但是如果本域下有容易被利用的功能，比如可以发图和链接的论坛和评论区，攻击可以直接在本域下进行，而且这种攻击更加危险。

防护策略

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。

讲了CSRF的两个特点：

CSRF（通常）发生在第三方域名。
CSRF攻击者不能获取到Cookie等信息，只是使用。
针对这两点，我们可以专门制定防护策略，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

同源检测

既然CSRF大多来自第三方网站，那么我们就直接禁止外域（或者不受信任的域名）对我们发起请求。

那么问题来了，我们如何判断请求是否来自外域呢?

在HTTP协议中，每一个异步请求都会携带两个Header，用于标记来源域名：

Origin Header
Referer Header
这两个Header在浏览器发起请求时，大多数情况会自动带上，并且不能由前端自定义内容。
服务器可以通过解析这两个Header中的域名，确定请求的来源域。

- 使用Origin Header确定来源域名
  在部分与CSRF有关的请求中，请求的Header中会携带Origin字段。字段内包含请求的域名（不包含path及query）。

如果Origin存在，那么直接使用Origin中的字段确认来源域名就可以。

但是Origin在以下两种情况下并不存在：

IE11同源策略： IE 11 不会在跨站CORS请求上添加Origin标头，Referer头将仍然是唯一的标识。最根本原因是因为IE 11对同源的定义和其他浏览器有不同，有两个主要的区别，可以参考MDN Same-origin_policy#IE_Exceptions

302重定向： 在302重定向之后Origin不包含在重定向的请求中，因为Origin可能会被认为是其他来源的敏感信息。对于302重定向的情况来说都是定向到新的服务器上的URL，因此浏览器不想将Origin泄漏到新的服务器上。

- 使用Referer Header确定来源域名
  根据HTTP协议，在HTTP头中有一个字段叫Referer，记录了该HTTP请求的来源地址。
  对于Ajax请求，图片和script等资源请求，Referer为发起请求的页面地址。对于页面跳转，Referer为打开页面历史记录的前一个页面地址。因此我们使用Referer中链接的Origin部分可以得知请求的来源域名。

这种方法并非万无一失，Referer的值是由浏览器提供的，虽然HTTP协议上有明确的要求，但是每个浏览器对于Referer的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不是很安全。在部分情况下，攻击者可以隐藏，甚至修改自己请求的Referer。

CSRF Token

前面讲到CSRF的另一个特征是，攻击者无法直接窃取到用户的信息（Cookie，Header，网站内容等），仅仅是冒用Cookie中的信息。

而CSRF攻击之所以能够成功，是因为服务器误把攻击者发送的请求当成了用户自己的请求。那么我们可以要求所有的用户请求都携带一个CSRF攻击者无法获取到的Token。服务器通过校验请求是否携带正确的Token，来把正常的请求和攻击的请求区分开，也可以防范CSRF的攻击。

原理
CSRF Token的防护策略分为三个步骤：

1. 将CSRF Token输出到页面中

首先，用户打开页面的时候，服务器需要给这个用户生成一个Token，该Token通过加密算法对数据进行加密，一般Token都包括随机字符串和时间戳的组合，显然在提交时Token不能再放在Cookie中了，否则又会被攻击者冒用。因此，为了安全起见Token最好还是存在服务器的Session中，之后在每次页面加载时，使用JS遍历整个DOM树，对于DOM中所有的a和form标签后加入Token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的HTML代码，这种方法就没有作用，还需要程序员在编码时手动添加Token。

2. 页面提交的请求携带这个Token

对于GET请求，Token将附在请求地址之后，这样URL 就变成 http://url?csrftoken=tokenvalue。 而对于 POST 请求来说，要在 form 的最后加上：

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

这样，就把Token以参数的形式加入请求了。

3. 服务器验证Token是否正确

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，验证过程是先解密Token，对比加密字符串以及时间戳，如果加密字符串一致且时间未过期，那么这个Token就是有效的。

这种方法要比之前检查Referer或者Origin要安全一些，Token可以在产生并放于Session之中，然后在每次请求时把Token从Session中拿出，与请求中的Token进行比对，但这种方法的比较麻烦的在于如何把Token以参数的形式加入请求。
下面将以Java为例，介绍一些CSRF Token的服务端校验逻辑，代码如下：

```java
HttpServletRequest req = (HttpServletRequest)request;
HttpSession s = req.getSession();
// 从 session 中得到 csrftoken 属性
String sToken = (String)s.getAttribute("csrftoken");
if(sToken == null){
    // 产生新的 token 放入 session 中
    sToken = generateToken();
    s.setAttribute("csrftoken",sToken);
    chain.doFilter(request, response);
} else{
    // 从 HTTP 头中取得 csrftoken
    String xhrToken = req.getHeader("csrftoken");
    // 从请求参数中取得 csrftoken
    String pToken = req.getParameter("csrftoken");
    if(sToken != null && xhrToken != null && sToken.equals(xhrToken)){
        chain.doFilter(request, response);
    }else if(sToken != null && pToken != null && sToken.equals(pToken)){
        chain.doFilter(request, response);
    }else{
        request.getRequestDispatcher("error.jsp").forward(request,response);
    }
}
```

代码源自IBM developerworks CSRF

总结

Token是一个比较有效的CSRF防护方法，只要页面没有XSS漏洞泄露Token，那么接口的CSRF攻击就无法成功。

但是此方法的实现比较复杂，需要给每一个页面都写入Token（前端无法使用纯静态页面），每一个Form及Ajax请求都携带这个Token，后端对每一个接口都进行校验，并保证页面Token及请求Token一致。这就使得这个防护策略不能在通用的拦截上统一拦截处理，而需要每一个页面和接口都添加对应的输出和校验。这种方法工作量巨大，且有可能遗漏。

验证码和密码其实也可以起到CSRF Token的作用哦，而且更安全。

双重Cookie验证

在会话中存储CSRF Token比较繁琐，而且不能在通用的拦截上统一处理所有的接口。

那么另一种防御措施是使用双重提交Cookie。利用CSRF攻击不能获取到用户Cookie的特点，我们可以要求Ajax和表单请求携带一个Cookie中的值。

双重Cookie采用以下流程：

在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串（例如csrfcookie=v8g9e4ksfhw）。
在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例POST https://www.a.com/comment?csrfcookie=v8g9e4ksfhw）。
后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝。
此方法相对于CSRF Token就简单了许多。可以直接通过前后端拦截的的方法自动化实现。后端校验也更加方便，只需进行请求中字段的对比，而不需要再进行查询和存储Token。

当然，此方法并没有大规模应用，其在大型网站上的安全性还是没有CSRF Token高，原因我们举例进行说明。

由于任何跨域都会导致前端无法获取Cookie中的字段（包括子域名之间），于是发生了如下情况：

如果用户访问的网站为www.a.com，而后端的api域名为api.a.com。那么在www.a.com下，前端拿不到api.a.com的Cookie，也就无法完成双重Cookie认证。
于是这个认证Cookie必须被种在a.com下，这样每个子域都可以访问。
任何一个子域都可以修改a.com下的Cookie。
某个子域名存在漏洞被XSS攻击（例如upload.a.com）。虽然这个子域下并没有什么值得窃取的信息。但攻击者修改了a.com下的Cookie。
攻击者可以直接使用自己配置的Cookie，对XSS中招的用户再向www.a.com下，发起CSRF攻击。
总结：
用双重Cookie防御CSRF的优点：

无需使用Session，适用面更广，易于实施。

Token储存于客户端中，不会给服务器带来压力。
相对于Token，实施成本更低，可以在前后端统一拦截校验，而不需要一个个接口和页面添加。
缺点：

Cookie中增加了额外的字段。
如果有其他漏洞（例如XSS），攻击者可以注入Cookie，那么该防御方式失效。
难以做到子域名的隔离。
为了确保Cookie传输安全，采用这种防御方式的最好确保用整站HTTPS的方式，如果还没切HTTPS的使用这种方式也会有风险。

Samesite Cookie属性

防止CSRF攻击的办法已经有上面的预防措施。为了从源头上解决这个问题，Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个 Cookie是个“同站 Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，分别是 Strict 和 Lax，下面分别讲解：

Samesite=Strict
这种称为严格模式，表明这个 Cookie 在任何情况下都不可能作为第三方 Cookie，绝无例外。比如说 b.com 设置了如下 Cookie：

Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
我们在 a.com 下发起对 b.com 的任意请求，foo 这个 Cookie 都不会被包含在 Cookie 请求头中，但 bar 会。举个实际的例子就是，假如淘宝网站用来识别用户登录与否的 Cookie 被设置成了 Samesite=Strict，那么用户从百度搜索页面甚至天猫页面的链接点击进入淘宝后，淘宝都不会是登录状态，因为淘宝的服务器不会接受到那个 Cookie，其它网站发起的对淘宝的任意请求都不会带上那个 Cookie。

Samesite=Lax
这种称为宽松模式，比 Strict 放宽了点限制：假如这个请求是这种请求（改变了当前页面或者打开了新页面）且同时是个GET请求，则这个Cookie可以作为第三方Cookie。比如说 b.com设置了如下Cookie：

Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
当用户从 a.com 点击链接进入 b.com 时，foo 这个 Cookie 不会被包含在 Cookie 请求头中，但 bar 和 baz 会，也就是说用户在不同网站之间通过链接跳转是不受影响了。但假如这个请求是从 a.com 发起的对 b.com 的异步请求，或者页面跳转是通过表单的 post 提交触发的，则bar也不会发送。

生成Token放到Cookie中并且设置Cookie的Samesite，Java代码如下：

```
 private void addTokenCookieAndHeader(HttpServletRequest httpRequest, HttpServletResponse
httpResponse) {
    //生成token
    String sToken = this.generateToken();
    //手动添加Cookie实现支持"Samesite=strict"
    //Cookie添加双重验证
    String CookieSpec = String.format("%s=%s; Path=%s; HttpOnly; Samesite=Strict",
this.determineCookieName(httpRequest), sToken, httpRequest.getRequestURI());
    httpResponse.addHeader("Set-Cookie", CookieSpec);
    httpResponse.setHeader(CSRF_TOKEN_NAME, token);
  }
```

代码源自OWASP Cross-Site_Request_Forgery #Implementation example

我们应该如何使用SamesiteCookie
如果SamesiteCookie被设置为Strict，浏览器在任何跨域请求中都不会携带Cookie，新标签重新打开也不携带，所以说CSRF攻击基本没有机会。

但是跳转子域名或者是新标签重新打开刚登陆的网站，之前的Cookie都不会存在。尤其是有登录的网站，那么我们新打开一个标签进入，或者跳转到子域名的网站，都需要重新登录。对于用户来讲，可能体验不会很好。

如果SamesiteCookie被设置为Lax，那么其他网站通过页面跳转过来的时候可以使用Cookie，可以保障外域连接打开页面时用户的登录状态。但相应的，其安全性也比较低。

另外一个问题是Samesite的兼容性不是很好，现阶段除了从新版Chrome和Firefox支持以外，Safari以及iOS Safari都还不支持，现阶段看来暂时还不能普及。

而且，SamesiteCookie目前有一个致命的缺陷：不支持子域。例如，种在topic.a.com下的Cookie，并不能使用a.com下种植的SamesiteCookie。这就导致了当我们网站有多个子域名时，不能使用SamesiteCookie在主域名存储用户登录信息。每个子域名都需要用户重新登录一次。

总之，SamesiteCookie是一个可能替代同源验证的方案，但目前还并不成熟，其应用场景有待观望。

防止网站被利用

前面所说的，都是被攻击的网站如何做好防护。而非防止攻击的发生，CSRF的攻击可以来自：

攻击者自己的网站。
有文件上传漏洞的网站。
第三方论坛等用户内容。
被攻击网站自己的评论功能等。
对于来自黑客自己的网站，我们无法防护。但对其他情况，那么如何防止自己的网站被利用成为攻击的源头呢?

严格管理所有的上传接口，防止任何预期之外的上传内容（例如HTML）。
添加Header X-Content-Type-Options: nosniff 防止黑客上传HTML内容的资源（例如图片）被解析为网页。
对于用户上传的图片，进行转存或者校验。不要直接使用用户填写的图片链接。
当前用户打开其他用户填写的链接时，需告知风险（这也是很多论坛不允许直接在内容中发布外域链接的原因之一，不仅仅是为了用户留存，也有安全考虑）。

CSRF其他防范措施

对于一线的程序员同学，我们可以通过各种防护策略来防御CSRF，对于QA、SRE、安全负责人等同学，我们可以做哪些事情来提升安全性呢?

CSRF测试
CSRFTester是一款CSRF漏洞的测试工具，CSRFTester工具的测试原理大概是这样的，使用代理抓取我们在浏览器中访问过的所有的连接以及所有的表单等信息，通过在CSRFTester中修改相应的表单等信息，重新提交，相当于一次伪造客户端请求，如果修改后的测试请求成功被网站服务器接受，则说明存在CSRF漏洞，当然此款工具也可以被用来进行CSRF攻击。
CSRFTester使用方法大致分下面几个步骤：

步骤1：设置浏览器代理
CSRFTester默认使用Localhost上的端口8008作为其代理，如果代理配置成功，CSRFTester将为您的浏览器生成的所有后续HTTP请求生成调试消息。

步骤2：使用合法账户访问网站开始测试
我们需要找到一个我们想要为CSRF测试的特定业务Web页面。找到此页面后，选择CSRFTester中的"开始录制"按钮并执行业务功能；完成后，点击CSRFTester中的"停止录制"按钮；正常情况下，该软件会全部遍历一遍当前页面的所有请求。

步骤3：通过CSRF修改并伪造请求
之后，我们会发现软件上有一系列跑出来的记录请求，这些都是我们的浏览器在执行业务功能时生成的所有GET或者POST请求。通过选择列表中的某一行，我们现在可以修改用于执行业务功能的参数，可以通过点击对应的请求修改query和form的参数。当修改完所有我们希望诱导用户form最终的提交值，可以选择开始生成HTML报告。

步骤4：拿到结果如有漏洞进行修复
首先必须选择"报告类型"。报告类型决定了我们希望受害者浏览器如何提交先前记录的请求。目前有5种可能的报告：表单、iFrame、IMG、XHR和链接。一旦选择了报告类型，我们可以选择在浏览器中启动新生成的报告，最后根据报告的情况进行对应的排查和修复。

CSRF监控
对于一个比较复杂的网站系统，某些项目、页面、接口漏掉了CSRF防护措施是很可能的。

一旦发生了CSRF攻击，我们如何及时的发现这些攻击呢?

CSRF攻击有着比较明显的特征：

跨域请求。
GET类型请求Header的MIME类型大概率为图片，而实际返回Header的MIME类型为Text、JSON、HTML。
我们可以在网站的代理层监控所有的接口请求，如果请求符合上面的特征，就可以认为请求有CSRF攻击嫌疑。我们可以提醒对应的页面和项目负责人，检查或者 Review其CSRF防护策略。

个人用户CSRF安全的建议

经常上网的个人用户，可以采用以下方法来保护自己：

使用网页版邮件的浏览邮件或者新闻也会带来额外的风险，因为查看邮件或者新闻消息有可能导致恶意代码的攻击。
尽量不要打开可疑的链接，一定要打开时，使用不常用的浏览器。

找出字符串出现最多的次数

```
var str = 'aaaaccccrrrrrr';
function get(str) {
  const arr = str.split('')
  return arr.reduce(function (pre, cur) {
    if (cur in pre) {
      pre[cur]++
    } else {
      pre[cur] = 1
    }
    return pre
  }, {})
}

console.log(get(str));

var maxObj = get(str)

var maxStr = null
var max = 0

for (var key in maxObj) {
  if (max < maxObj[key]) {
    max = maxObj[key]
    maxStr = key
  }
}

console.log(`出现次数最多的是${maxStr},出现了${max}次`);
```

获取区间的随机数

```
function randomNum(min, max) {
  var range = max - min
  var rand = Math.random()
  return min + Math.round(rand * range)
}

console.log(randomNum(1, 10));
```

事件委托

```
// 写一个事件委托
var ul = document.getElementById('ul')

ul.onclick = function (event) {
  event = event || window.event
  var target = event.target
  if (target.nodeName === 'LI') {
    alert(target.innerHTML)
  }
}
```

写一个函数-判断数据的类型

```
function getType(type) {
  var ty;
  if (typeof type === 'object') {
    // return 'obj'
    Object.prototype.toString.call(type) === '[object Object]' &&
    (ty = 'object')
    Object.prototype.toString.call(type) === '[object Array]' &&
    (ty = 'array')

  } else {
    typeof type === 'string' && (ty = 'string')
    typeof type === 'number' && (ty = 'number')
    typeof type === 'undefined' && (ty = 'undefined')
    typeof type === 'boolean' && (ty = 'boolean')
    typeof type === 'function' && (ty = 'function')
  }

  return ty
}

console.log(getType('string'));
console.log(getType(123));
console.log(getType(undefined));
console.log(getType(true));
console.log(getType({}));
console.log(getType([]));
console.log(getType(function () { }));
```

写一个函数-计算传入参数的和

```
// 需要你写一个函数 调用这个函数 来计算传入参数的和 （参数的个数是不确定的）

function getSum() {
  var length = arguments.length
  var s = 0
  for (var i = 0; i < length; i++) {
    if (!isNaN(arguments[i])) {
      s += Number(arguments[i])
    }
  }

  return s
}

console.log(getSum(1, '2', 3, '4', NaN));
```

单例模式理解

- 单例模式的定义是：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 单例模式是一种常用的模式，有一些对象我们往往只需要一个，比如线程池、全局缓存、浏 览器中的 window 对象等。在 JavaScript 开发中，单例模式的用途同样非常广泛。试想一下，当我 们单击登录按钮的时候，页面中会出现一个登录浮窗，而这个登录浮窗是唯一的，无论单击多少 次登录按钮，这个浮窗都只会被创建一次，那么这个登录浮窗就适合用单例模式来创建。

[#]js中的全局变量

- 全局变量不是单例模式，但在 JavaScript 开发中，我们经常会把全局变量当成单例来使用。

1. 如果 a 变量被声明在全局作用域下， 则我们可以在代码中的任何位置使用这个变量，全局变量提供给全局访问是理所当然的。这样就 满足了单例模式的两个条件。
2. 全局变量会很容易被覆盖存在问题
3. Douglas Crockford 多次把全局变量称为 JavaScript 中最糟糕的特性。在对 JavaScript 的创造者 Brendan Eich 的访谈中， Brendan Eich 本人也承认全局变量是设计上的失误，是在没有足够的时间思考一些东西的情况下导致的结果。

- 降低命名空间的污染

1. 使用命名空间 适当地使用命名空间，并不会杜绝全局变量，但可以减少全局变量的数量。 最简单的方法依然是用对象字面量的方式：

```
var namespace1 = {
a: function(){
  alert (1);
},
b: function(){
  alert (2);
}
};
```

1. 使用闭包封装私有变量 这种方法把一些变量封装在闭包的内部，只暴露一些接口跟外界通信：

```
var user = (function(){
  var __name = 'sven',
  __age = 29;
  return {
    getUserInfo: function(){
      return __name + '-' + __age;
    }
  }
})();
```

我们用下划线来约定私有变量**name** 和age，它们被封装在闭包产生的作用域中，外部是访问不到这两个变量的，这就避免了对全局的命令污染。

普通实现登录弹窗

```
<button id="login">登录</button>
<script>
  // 第一种解决方案 在页面加载完成的时候 创建好这个div 让它是隐藏的状态 当用户点击的时候 让这个登录的弹窗出来

  var loginLayer = (function () {
    var div = document.createElement('div')
    div.innerHTML = '我是登录的弹窗'
    div.style.display = 'none'
    document.body.append(div)
    return div
  })()

  document.getElementById('login').onclick = function () {
    loginLayer.style.display = 'block'
  }
</script>
```

单例模式实现

```
var createLoginLayer = (function () {
  var div = null
  return function () {
    if (!div) {
      div = document.createElement('div')
      div.innerHTML = '我是登录的弹窗'
      div.style.display = 'none'
      document.body.append(div)
    }
    return div
  }
})()


document.getElementById('login').onclick = function () {
  var loginLayer = createLoginLayer()
  loginLayer.style.display = 'block'
}
```

单一职责改进

```
// 单一职责 各自做各自的事情
var getSingle = function (fn) {
  var result = null
  return function () {
    // 是创建的业务逻辑 创建iframe 创建登录的弹窗 .....
    // result = fn()
    // return result
    return result || (result = fn())
  }
}

var createLoginLayer = function () {
  var div = document.createElement('div')
  div.innerHTML = '我是登录的弹窗'
  div.style.display = 'none'
  document.body.append(div)
  return div
}
var createSingleLogin = getSingle(createLoginLayer)

document.getElementById('login').onclick = function () {
  var loginLayer = createSingleLogin()
  loginLayer.style.display = 'block'
}
```

es6-class 类实现

```
class SingletonApple {
  constructor(name, creator, products) {
    //首次使用构造器实例
    if (!SingletonApple.instance) {
      this.name = name;
      this.creator = creator;
      this.products = products;
      //将this挂载到SingletonApple这个类的instance属性上
      SingletonApple.instance = this;
    }
    return SingletonApple.instance;
  }
}

let appleCompany = new SingletonApple('苹果公司', '乔布斯', ['iPhone', 'iMac', 'iPad',
'iPod']);
let copyApple = new SingletonApple('苹果公司', '阿辉', ['iPhone', 'iMac', 'iPad', 'iPod']);

console.log(appleCompany === copyApple);  //true
```

- ES6的静态方法优化代码
- 如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```
class SingletonApple {
  constructor(name, creator, products) {
      this.name = name;
      this.creator = creator;
      this.products = products;
  }
  //静态方法
  static getInstance(name, creator, products) {
    if(!this.instance) {
      this.instance = new SingletonApple(name, creator, products);
    }
    return this.instance;
  }
}

let appleCompany = SingletonApple.getInstance('苹果公司', '乔布斯', ['iPhone', 'iMac',
'iPad', 'iPod']);
let copyApple = SingletonApple.getInstance('苹果公司', '阿辉', ['iPhone', 'iMac', 'iPad',
'iPod'])
console.log(appleCompany === copyApple); //true
```

js-reduce 应用

js数组的reduce方法

```js
arr.reduce(function(prev,cur,index,arr){
...
}, init);

// 或者

arr.reduce(function(prev,cur,index,arr){
...
},);
```

- arr 表示将要原数组;
- prev 表示上一次调用回调时的返回值，或者初始值 init 如果没有初始值 那么prev的值是数组的第一项;
- cur 表示当前正在处理的数组元素;
- index 表示当前正在处理的数组元素的索引，若提供 init 值，则索引为0，否则索引为1;
- init 表示初始值。

常用的参数只有两个: prev 和 cur。

- 数组求和，求乘积

```js
var  arr = [1, 2, 3, 4];
var sum = arr.reduce((x,y)=>x+y)
var mul = arr.reduce((x,y)=>x*y)
console.log( sum ); //求和, 10
console.log( mul ); //求乘积, 24
```

- 计算数组中每个元素出现的次数

```js
let names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];

let nameNum = names.reduce((pre,cur)=>{
if(cur in pre){
    pre[cur]++
}else{
    pre[cur] = 1
}
return pre
},{})
console.log(nameNum); //{Alice: 2, Bob: 1, Tiff: 1, Bruce: 1}
```

- 数组去重

```js
let arr = [1,2,3,4,4,1]
let newArr = arr.reduce((pre,cur)=>{
    if(!pre.includes(cur)){
      return pre.concat(cur)
    }else{
      return pre
    }
},[])
console.log(newArr);// [1, 2, 3, 4]
```

- 将二维数组转化成一维数组

```js
let arr = [[0, 1], [2, 3], [4, 5]]
let newArr = arr.reduce((pre,cur)=>{
    return pre.concat(cur)
},[])
console.log(newArr); // [0, 1, 2, 3, 4, 5]

```

- 将多维数组转化成一维数组

```js
let arr = [[0, 1], [2, 3], [4,[5,6,7]]]
const newArr = function(arr){
    return arr.reduce((pre,cur)=>pre.concat(Array.isArray(cur)?newArr(cur):cur),[])
}
console.log(newArr(arr)); //[0, 1, 2, 3, 4, 5, 6, 7]
```

- 对象里的属性求和

```js
var result = [
    {
        subject: 'math',
        score: 10
    },
    {
        subject: 'chinese',
        score: 20
    },
    {
        subject: 'english',
        score: 30
    }
];

var sum = result.reduce(function(prev, cur) {
    return cur.score + prev;
}, 0);
console.log(sum) //60

```

cookies-sessionStorage-localStorage 的区别

1. cookie 是网站为了标示用户身份而储存在用户本地终端（Client Side）上的数据（通常经过加密）
2. cookie数据始终在同源的http请求中携带（即使不需要），记会在浏览器和服务器间来回 传递
   sessionStorage 和 localStorage 不会自动把数据发给服务器，仅在本地保存

存储大小：

1. cookie 数据大小不能超过4k
2. 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

有期时间：

1. 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
2. sessionStorage 数据在当前浏览器窗口关闭后自动删除
3. 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

隐式转换

```
console.log(1 + "true"); // '1true'
// String(1)+'true' = '1true'
console.log(1 + true); // 2
// 1 + Number(true) = 2
console.log(1 + undefined); // NaN
// 1 + Number(undefined) = 1 + NaN = NaN
console.log(1 + null); // 1
// 1 + Number(null) = 1+0 = 1
```

```
// 关系运算符
console.log("2" > 10); // false Number('2') 2 > 10 ? false
console.log("2" > "10"); // true
// 按照字符串对应的unicode编码来转换的

//多个字符从左往右依次比较
console.log("abc" > "b"); // false
console.log("abc" > "aad");  // true
console.log(undefined == undefined); // true
console.log(undefined == null); // true
console.log(null == null); // true
console.log(NaN == NaN); // false
```

```
// 腾讯2018年前端笔试题
console.log([] == 0); // true // Number([].valueOf().toString()) = 0
console.log(![] == 0); // true // !Boolean([])  ==  0 true


// 0 NaN undefined null '' false

// 1 关系运算符 将其他数据类型转换成数字
// 2 逻辑非 将其他数据类型用Boolean()转换

console.log([] == ![]); // true [] ''  [].valueOf().toString()  // false
false '' Numner('') == Number(false)
console.log([] == []); // false

console.log({} == !{}); // true???

// {}.valueOf().toString() = '[obejct Object]'

// false
console.log({} == {}); // false
```