作者：AddOneG

JS高级技巧(简洁版)

高级函数

由于在 JS 中，所有的函数都是对象，所以使用函数指针十分简单，也是这些东西使 JS 函数有趣且强大

安全的类型检测

JS 内置的类型检测机制并不是完全可靠的

typeof

操作符返回一个字符串，表示未经计算的操作数的类型，在大多数情况下很靠谱，但是当然还有例外

正则表达式

```javascript
typeof /s/ === 'function'; // Chrome 1-12 , 不符合 ECMAScript 5.1
typeof /s/ === 'object'; // Firefox 5+ , 符合 ECMAScript 5.1
```

NULL

```javascript
typeof null === 'object'; // 从一开始出现JavaScript就是这样的
```

> 在 JavaScript 最初的实现中， JavaScript 中的值是由一个表示类型的标签和实际数据值表示的。对象的类型标签是 0。由于 null 代表的是空指针（大多数平台下值为 0x00 ），因此， null 的类型标签也成为了 0， typeof null 就错误的返回了 object

instanceof

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

语法

> object instanceof constructor(要检测的对象 instanceof 构造函数)

但是在浏览器中，我们的脚本可能需要在多个窗口之间进行交互。多个窗口意味着多个全局环境，不同的全局环境拥有不同的全局对象，从而拥有不同的内置类型构造函数。这可能会引发一些问题。

```javascript
[] instanceof window.frames[0].Array        //false
```

因为 `Array.prototype !== window.frames[0].Array.prototype` ，因此你必须使用 `Array.isArray(myObj)` 或者 `Object.prototype.toString.call(myObj) === "[object Array]"` 来判断 `myObj` 是否是数组

解决以上两个问题的方案就是 `Object.prototype.toString`

Object.prototype.toString

方法返回一个表示该对象的字符串

可以通过 `toString()` 来获取每个对象的类型。为了每个对象都能通过 `Object.prototype.toString()` 来检测，需要以 `Function.prototype.call()` 或者 `Function.prototype.apply()` 的形式来调用，传递要检查的对象作为第一个参数，称为 `thisArg`

```javascript
var toString = Object.prototype.toString;

toString.call(new Date); // [object Date]
toString.call(new String); // [object String]
toString.call(Math); // [object Math]
toString.call(/s/); // [object RegExp]
toString.call([]); // [object Array]

//Since JavaScript 1.8.5
toString.call(undefined); // [object Undefined]
toString.call(null); // [object Null]
```

作用域安全的构造函数

构造函数其实就是一个使用 new 操作符调用的函数。当使用 new 调用时，构造函数内用到的 this 对象会指向新创建的对象实例

```javascript
function Person(name, age){
    this.name = name;
    this.age = age;
}

let person = new Person("addone", 20);

person.name // addone
```

当你使用 new 操作符的时候，就会创建一个新的 Person 对象，同时分配这些属性，但是如果你没有使用 new

```javascript
let person = Person("addone", 20);

person1.name // Cannot read property 'name' of undefined
window.name // addone
```

这是因为 this 是在执行时确认的，当你没有使用 new ，那么 this 在当前情况下就被解析成了 window ，属性就被分配到 window 上了

作用域安全的构造函数在进行更改前，首先确认 this 对象是正确类型的实例，如果不是，就创建新的对象并且返回

```javascript
function Person(name, age){
    if(this instanceof Person){
        this.name = name;
        this.age = age;
    }else{
        return new Person(name, age);
    }
}

let person1 = new Person("addone", 20);
person1.name // addone

let person2 = Person("addone", 20);
person2.name // addone
```

this instanceof Person 检查了 this 对象是不是 Person 的实例，如果是则继续，不是则调用 new

惰性载入函数

假如你要写一个函数，里面有一些判断语句

```javascript
function foo(){
    if(a != b){
        console.log('aaa')
    }else{
        console.log('bbb')
    }
}
```

如果你的 `a` 和 `b` 是不变的，那么这个函数不论执行多少次，结果都是不变的，但是每次执行还要进行 `if` 判断，这就造成了不必要的浪费。

惰性载入表示函数执行的分支只会发生一次，这里有两种解决方式。

在函数被调用时再处理函数

```javascript
function foo(){
    if(a != b){
        foo = function(){
            console.log('aaa')
        }
    }else{
        foo = function(){
            console.log('bbb')
        }
    }
    return foo();
}
```

这样进入每个分支后都会对 `foo` 进行赋值，覆盖了之前的函数，之后每次调用 `foo` 就不会再执行 `if` 判断

在声明函数时就指定适当的函数

```javascript
var foo = (function foo(){
    if(a != b){
        return function(){
            console.log('aaa')
        }
```

```
    }else{
        return function(){
            console.log('bbb')
        }
    }
})();
```

这里创建一个匿名，自执行的函数，用来确定应该使用哪一个函数来实现。

> 惰性函数的优点就是只在第一次执行分支时牺牲一点点性能

函数绑定

请使用 `fun.bind(thisArg[, arg1[, arg2[, ...]]])`

thisArg

当绑定函数被调用时，该参数会作为原函数运行时的 `this` 指向。当使用 `new` 操作符调用绑定函数时，该参数无效

arg1,arg2,...

当绑定函数被调用时，这些参数将置于实参之前传递给被绑定的方法

返回

由指定的this值和初始化参数改造的原函数拷贝

一个例子

javascript

```
let person = {
   name: 'addone',
   click: function(e){
       console.log(this.name)
   }
}

let btn = document.getElementById('btn');
EventUtil.addHandle(btn, 'click', person.click);
```

这里创建了一个 person 对象，然后将 person.click 方法分配给 DOM 按钮的事件处理程序，当你点击按按钮时，会打印出 undefiend ，原因是执行时 this 指向了 DOM 按钮而不是 person

解决方案： 将 this 强行指向 person

```javascript
EventUtil.addHandle(btn, 'click', person.click.bind(person));
```

函数柯里化

函数柯里化是把接受多个参数的函数转变成接受单一参数的函数

```javascript
function add(num1, num2){
    return num1 + num2;
}
function curryAdd(num2){
    return add(1, num2);
}
add(2, 3) // 5
curryAdd(2) // 3
```

> 这个例子用来方便理解柯里化的概念

下面是创建函数柯里化的通用方式

```javascript
function curry(fn){
    var args = Array.prototype.slice.call(arguments, 1);
    return function(){
        let innerArgs = Array.prototype.slice.call(arguments);
        let finalArgs = args.concat(innerArgs);
        return fn.apply(null, finalArgs);
    }
}
```

第一个参数是要进行柯里化的函数，其他参数是要传入的值。这里使用 Array.prototype.slice.call(arguments, 1) 来获取第一个参数后的所有参数(外部)。在返回的函数中，同样调用 Array.prototype.slice.call(arguments) 让 innerArgs 来存放所有的参数(内部)，然后用 concat 将内部外部参数组合，用 apply 传递给函数

```javascript
function add(num1, num2){
    return num1 + num2;
}
let curryAdd1 = curry(add, 1);
curryAdd1(2); // 3

let curryAdd2 = curry(add, 1, 2);
curryAdd2(); // 3
```

防篡改对象

Javascript 中任何对象都可以被同一环境中运行的代码修改，所以开发人员有时候需要定义**防篡改对象（tamper-proof object)** 来保护自己

不可扩展对象

默认情况下所有对象都是可以扩展的(添加属性和方法)

```javascript
let person = { name: 'addone' };
person.age = 20;
```

第二行为 person 对象扩展了 age 属性，当然你可以阻止这一行为，使用 Object.preventExtensions()

```javascript
let person = { name: 'addone' };
Object.preventExtensions(person);
person.age = 20;

person.age // undefined
```

你还可以用 Object.isExtensible() 来判断对象是不是可扩展的

```javascript
let person = { name: 'addone' };
Object.isExtensible(person); // true

Object.preventExtensions(person);
Object.isExtensible(person); // false
```

> 请记住这是**不可扩展!!**，即不能添加属性或方法

密封的对象

密封对象不可扩展，且不能删除属性和方法

```javascript
let person = { name: 'addone' };
Object.seal(person);

person.age = 20;
delete person.name;

person.age // undefined
person.name // addone
```

相对的也有 `Object.isSealed()` 来判断是否密封

```javascript
let person = { name: 'addone' };
Object.isExtensible(person); // true
Object.isSealed(person); // false

Object.seal(person);
Object.isExtensible(person); // false
Object.isSealed(person); // true
```

冻结的对象

这是最严格的防篡改级别，冻结的对象即不可扩展，又密封，且不能修改

```javascript
let person = { name: 'addone' };
Object.freeze(person);

person.age = 20;
delete person.name;
person.name = 'addtwo'

person.age // undefined
person.name // addone
```

同样也有 `Object.isFrozen` 来检测

```javascript
let person = { name: 'addone' };
Object.isExtensible(person); // true
Object.isSealed(person); // false
Object.isFrozen(person); // false

Object.freeze(person);
Object.isExtensible(person); // false
Object.isSealed(person); // true
Object.isFrozen(person); // true
```

> 以上三种方法在严格模式下进行错误操作均会导致抛出错误

高级定时器

阅读前提

大概理解 `setTimeout` 的基本执行机制和 `js` 事件机制

重复的定时器

当你使用 `setInterval` 重复定义多个定时器的时候，可能会出现**某个定时器代码在代码再次被添加到执行队列之前还没有完成执行**，导致定时器代码连续执行多次。

机智 `Javascript` 引擎解决了这个问题，使用 `setInterval()` 的时候，仅当没有该定时器的其他代码实例时，才会将定时器代码添加到队列中。但这还会导致一些问题:

- 某些间隔被跳过
- 间隔可能比预期的小

为了避免这个两个问题，你可以使用链式 `setTimeout()` 调用

```javascript
setTimeout(function(){
    TODO();

    setTimeout(arguments.callee, interval);
}, interval)
```

`arguments.callee` 获取了当前执行函数的引用，然后为其设置另外一个定时器，这样就确保在下一次定时器代码执行前，必须等待指定的间隔。

Yielding Processes

浏览器对长时间运行的脚本进行了制约，如果代码运行超过特定的时间或者特定语句数量就不会继续执行。

如果你发现某个循环占用了大量的时间，那么对于下面这两个问题

- 该处理是否必须同步完成?
- 数据是否必须按顺序完成?

如果你的两个答案都是"否"，那么你可以使用一种叫做**数组分块(array chunking)** 的技术。基本思路是为要处理的项目创建一个队列，然后使用定时器取出下一个要出处理的项目进行处理，然后再设置另一个定时器。

```javascript
function chunk(array, process, context){
    setTimeout(function(){
        // 取出下一个项目进行处理
        let item = array.shift();
        process.call(item);

        if(array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100)
}
```

这里接受三个参数，要处理的数组，处理的函数，运行该函数的环境(可选)，这里设置间隔 `100ms` 是个效果不错的选择

> 如果你一个函数需要50ms以上时间完成，那么最好看看能否将任务分割成一系列可以使用定时器的小任务

函数节流(Throttle)

节流的目的是防止某些操作执行的太快。比如在调整浏览器大小的时候会出发 `onresize` 事件，如果在其内部进行一些 `DOM` 操作，这种高频率的更爱可能会使浏览器崩溃。为了避免这种情况，可以采取函

数节流的方式。

```javascript
function throttle(method, context){
    clearTimeout(method.tId);
    method.tId = setTimeout(function(){
        method.call(context);
    }, 100)
}
```

这里接受两个参数，要执行的函数，执行的环境。执行时先清除之前的定时器，然后将当前定时器赋值给方法的 `tId` ，之后调用 `call` 来确定函数的执行环境。

一个应用的例子

```javascript
function resizeDiv(){
    let div = document.getElementById('div');
    div.style.height = div.offsetWidth + "px";
}

window.onresize = function(){
    throttle(resizeDiv);
}
```

> 这个就不用讲了吧2333

文章参考于《JavaScript高级程序设计(第三版)》

如果你觉得我的理解有问题或者整理的太简略，那么我强烈安利你自己去读一下这本书~