

基础知识

对象是包括属性与方法的数据类型，JS 中大部分类型都是对象如 `String/Number/Math/RegExp/Date` 等等。

传统的函数编程会有错中复杂的依赖很容易创造意大利式面条代码。

面向过程编程

```
1 let name = "向军";
2 let grade = [
3   { lesson: "js", score: 99 },
4   { lesson: "mysql", score: 85 }
5 ];
6 function average(grade, name) {
7   const total = grade.reduce((t, a) => t + a.score, 0);
8   return name + ":" + total / grade.length + "分";
9 }
10 console.log(average(grade, name));
11
```

面向对象编程

下面使用对象编程的代码结构清晰，也减少了函数的参数传递，也不用担心函数名的覆盖

```
1 let user = {
2   name: "后盾人",
3   grade: [
4     { lesson: "js", score: 99 },
5     { lesson: "mysql", score: 85 }
6   ],
7   average() {
8     const total = this.grade.reduce((t, a) => t + a.score, 0);
9     return this.name + ":" + total / grade.length + "分";
10  }
11 };
12 console.log(user.average());
13
```

#OOP

- 对象是属性和方法的集合即封装
- 将复杂功能隐藏在内部，只开放给外部少量方法，更改对象内部的复杂逻辑不会对外部调用造成影响即抽象
- 继承是通过代码复用减少冗余代码
- 根据不同形态的对象产生不同结果即多态

#基本声明

使用字面量形式声明对象是最简单的方式

```
1 let obj = {  
2   name: '后盾人',  
3   get:function() {  
4       return this.name;  
5   }  
6 }  
7 console.log(obj.get()); //后盾人  
8
```

属性与方法简写

```
1 let name = "后盾人";  
2 let obj = {  
3   name,  
4   get() {  
5       return this.name;  
6   }  
7 };  
8 console.log(obj.get()); //后盾人  
9
```

其实字面量形式在系统内部也是使用构造函数 `new Object` 创建的，后面会详细介绍构造函数。

```
1 let hd = {};  
2 let houdunren = new Object();  
3 console.log(hd, houdunren);  
4 console.log(hd.);  
5 console.log(houdunren.);  
6
```

#操作属性

使用点语法获取

```
1 let user = {  
2   name: "向军"  
3 };  
4 console.log(user.name);  
5
```

使用[] 获取

```
1 console.log(user["name"]);  
2
```

可以看出使用.操作属性更简洁，[]主要用于通过变量定义属性的场景

```
1 let user = {  
2   name: "向军"  
3 };  
4 let property = "name";  
5 console.log(user[property]);  
6
```

如果属性名不是合法变量名就必须使用扩号的形式了

```
1 let user = {};  
2 user["my-age"] = 28;  
3 console.log(user["my-age"]);  
4
```

对象和方法的属性可以动态的添加或删除。

```
1 const hd = {  
2   name: "后盾人"  
3 };  
4 hd.age = "10";
```

```
5  hd.show = function() {
6    return `${this.name}已经${this.age}岁了`;
7  };
8  console.log(hd.show());
9  console.log(hd);
10
11 delete hd.show;
12 delete hd.age;
13
14 console.log(hd);
15 console.log(hd.age); //undefined
16
```

#对象方法

定义在对象中的函数我们称为方法，下面定义了学生对象，并提供了计算平均成绩的方法

```
1  let lisi = {
2    name: "李四",
3    age: 22,
4    grade: {
5      math: 99,
6      english: 67
7    },
8    //平均成绩
9    avgGrade: function() {
10      let total = 0;
11      for (const key in this.grade) {
12        total += this.grade[key];
13      }
14      return total / this.propertyCount("grade");
15    },
16    //获取属性数量
17    propertyCount: function(property) {
18      let count = 0;
19      for (const key in this[property]) count++;
20      return count;
21    }
22  };
```

```
23 console.log(lisi.avgGrade());
24
```

一个学生需要手动创建一个对象，这显然不实际的，下面的构造函数就可以解决这个问题

#引用特性

对象和函数、数组一样是引用类型，即复制只会复制引用地址。

```
1 let hd = { name: "后盾人" };
2 let cms = hd;
3 cms.name = "hdcms";
4 console.log(hd.name); //hdcms
5
```

对象做为函数参数使用时也不会产生完全赋值，内外共用一个对象

```
1 let user = { age: 22 };
2 function hd(user) {
3   user.age += 10;
4 }
5 hd(user);
6 console.log(user.age); //32
7
```

对多的比较是对内存地址的比较所以使用 `==` 或 `===` 一样

```
1 let hd = {};
2 let xj = hd;
3 let cms = {};
4 console.log(hd == xj); //true
5 console.log(hd === xj); //true
6 console.log(hd === cms); //false
7
```

#this

`this` 指当前对象的引用，始终建议在代码内部使用 `this` 而不要使用对象名，不同对象的 `this` 只指向当前对象。

下例是不使用 `this` 时发生的错误场景

- 删除了

xj 变量，但在函数体内还在使用

xj变量造成错误

- 使用

this 后始终指向到引用地址，就不会有这个问题

```
1 let xj = {
2   name: "向军",
3   show() {
4     return xj.name;
5   }
6 };
7 let hd = xj;
8 xj = null;
9 console.log(hd.show()); //Error
10
```

改用this 后一切正常

```
1 let xj = {
2   name: "向军",
3   show() {
4     return this.name;
5   }
6 };
7 let hd = xj;
8 xj = null;
9 console.log(hd.show()); //Error
10
```

#展开语法

使用...可以展示对象的结构，下面是实现对象合并的示例

```
1 let hd = { name: "后盾人", web: "houdurnen.com" };
2 let info = { ...hd, site: "hdcms" };
3 console.log(info);
4
```

下面是函数参数合并的示例

```
1 function upload(params) {
2   let config = {
3     type: "*.jpeg,*.png",
4     size: 10000
5   };
6   params = { ...config, ...params };
7   console.log(params);
8 }
9 upload({ size: 999 });
10
```

#对象转换

#基础知识

对象直接参与计算时，系统会根据计算的场景在 `string/number/default` 间转换。

- 如果声明需要字符串类型，调用顺序为

`toString > valueOf`

- 如果场景需要数值类型，调用顺序为

`valueOf > toString`

- 声明不确定时使用

`default`，大部分对象的

`default` 会当数值使用

下面的数值对象会在数学运算时转换为 `number`

```
1 let houdunren = new Number(1);
2 console.log(houdunren + 3); //4
3
```

如果参数字符串运长时会转换为 `string`

```
1 let houdunren = new Number(1);
2 console.log(houdunren + "3"); //13
3
```

下面当不确定转换声明时使用 `default`，大部分`default`转换使用 `number` 转换。

```
1 let houdunren = new Number(1);
2 console.log(houdunren == "1"); //true
3
```

#Symbol.toPrimitive

内部自定义`Symbol.toPrimitive`方法用来处理所有的转换场景

```
1 let hd = {
2   num: 1,
3   [Symbol.toPrimitive]: function() {
4     return this.num;
5   }
6 };
7 console.log(hd + 3); //4
8
```

#valueOf/toString

可以自定义`valueOf` 与 `toString` 方法用来转换，转换并不限制返回类型。

```
1 let hd = {
2   name: "后盾人",
3   num: 1,
4   : function() {
5     console.log("valueOf");
6     return this.num;
7   },
8   : function() {
9     console.log("toString");
10    return this.name;
11  }
12 };
13 console.log(hd + 3); //valueOf 4
14 console.log(`${hd}向军`); //toString 后盾人向军
15
```


#解构赋值

解构是一种更简洁的赋值特性，可以理解为分解一个数据的结构，在数组章节已经介绍过。

- 建设使用

var/let/const 声明

#基本使用

下面是基本使用语法

```
1 //对象使用
2 let info = {name:'后盾人',url:'houdunren.com'};
3 let {name:n,url:u} = info
4 console.log(n); // 后盾人
5
6 //如果属性名与变量相同可以省略属性定义
7 let {name,url} = {name:'后盾人',url:'houdunren.com'};
8 console.log(name); // 后盾人
9
```

函数返回值直接解构到变量

```
1 function hd() {
2   return {
3     name: '后盾人',
4     url: 'houdunren.com'
5   };
6 }
7 let {name: n,url: u} = hd();
8 console.log(n);
9
```

函数传参

```
1 "use strict";
2 function hd({ name, age }) {
3   console.log(name, age); //向军大叔 18
4 }
5 hd({ name: "向军", age: 18 });
```

系统函数解构练习，这没有什么意义只是加深解构印象

```
1 const {random} =Math;
2 console.log(random());
3
```

#严格模式

非严格模式可以不使用声明指令，严格模式下必须使用声明。所以建议使用 let 等声明。

```
1 // "use strict";
2 ({name,url} = {name:'后盾人',url:'houdunren.com'});
3 console.log(name, url);
4
```

还是建议使用let等赋值声明

```
1 "use strict";
2 let { name, url } = { name: "后盾人", url: "houdunren.com" };
3 console.log(name, url);
4
```

#简洁定义

如果属性名与赋值的变量名相同可以更简洁

```
1 let web = { name: "后盾人",url: "houdunren.com" };
2 let { name, url } = web;
3 console.log(name); //后盾人
4
```

只赋值部分变量

```
1 let [,url]=['后盾人','houdunren.com'];
2 console.log(url);//houdunren.com
3
```

```
4 let {name}= {name:'后盾人',url:'houdunren.com'};  
5 console.log(name); //后盾人  
6
```

可以直接使用变量赋值对象属性

```
1 let name = "后盾人",url = "houdunren.com";  
2 //标准写法如下  
3 let hd = { name: name, url: url };  
4 console.log(hd); //{name: "后盾人", url: "houdunren.com"}  
5  
6 //如果属性和值变量同名可以写成以下简写形式  
7 let opt = { name, url };  
8 console.log(opt); //{name: "后盾人", url: "houdunren.com"}  
9
```

#嵌套解构

可以操作多层复杂数据结构

```
1 const hd = {  
2   name:'后盾人',  
3   lessons:{  
4     title:'JS'  
5   }  
6 }  
7 const {name,lessons:{title}} = hd;  
8 console.log(name,title); //后盾人 JS  
9
```

#默认值

为变量设置默认值

```
1 let [name, site = 'hdcms'] = ['后盾人'];  
2 console.log(site); //hdcms  
3  
4 let {name,url,user='向军大叔'}= {name:'后盾人',url:'houdunren.com'};
```

```
5 console.log(name,user);//向军大叔
6
```

使用默认值特性可以方便的对参数预设

```
1 function createElement(options) {
2   let {
3     width = '200px',
4     height = '100px',
5     backgroundColor = 'red'
6   } = options;
7
8   const h2 = document.createElement('h2');
9   h2.style.width = width;
10  h2.style.height = height;
11  h2.style.backgroundColor = backgroundColor;
12  document.body.appendChild(h2);
13 }
14 createElement({
15   backgroundColor: 'green'
16 });
17
```

#函数参数

数组参数的使用

```
1 function hd([a, b]) {
2   console.log(a, b);
3 }
4 hd(['后盾人', 'hdcms']);
5
```

对象参数使用方法

```
1 function hd({name,url,user='向军大叔'}) {
2   console.log(name,url,user);
3 }
4 hd({name:'后盾人','url':'houdunren.com'}); //后盾人 houdunren.com 向军大叔
```

对象解构传参

```

1 function user(name, { sex, age } = {}) {
2   console.log(name, sex, age); //向军大叔 男 18
3 }
4 user("向军大叔", { sex: "男", age: 18 });
5

```

#属性管理

#添加属性

可以为对象添加属性

```

1 let obj = {name: "后盾人"};
2 obj.site = "houdunren.com";
3 console.log(obj);
4

```

#删除属性

使用`delete` 可以删除属性（后面介绍的属性特性章节可以保护属性不被删除）

```

1 let obj = { name: "后盾人" };
2 delete obj.name;
3 console.log(obj.name); //undefined
4

```

#检测属性

`hasOwnProperty` 检测对象自身是否包含指定的属性，不检测原型链上继承的属性。

```

1 let obj = { name: '后盾人' };
2 console.log(obj.('name')); //true
3

```

下面通过数组查看

```
1 let arr = ["后盾人"];
2 console.log(arr);
3 console.log(arr.("length")); //true
4 console.log(arr.("concat")); //false
5 console.log("concat" in arr); //true
6
```

使用 `in` 可以在原型对象上检测

```
1 let obj = {name: "后盾人"};
2 let hd = {
3   web: "houdunren.com"
4 };
5
6 //设置hd为obj的新原型
7 Object.setPrototypeOf(obj, hd);
8 console.log(obj);
9
10 console.log("web" in obj); //true
11 console.log(obj.("web")); //false
12
```

#获取属性名

使用 `Object.getOwnPropertyNames` 可以获取对象的属性名集合

```
1 let hd = { name: '后盾人', year: 2010 }
2 const names = Object.getOwnPropertyNames(hd)
3 console.log(names)
4 // ["name", "year"]
5
```

#assign

以往我们使用类似 `jQuery.extend` 等方法设置属性，现在可以使用 `Object.assign` 静态方法从一个或多个对象复制属性

```
1 "use strict";
2 let hd = { a: 1, b: 2 };
3 hd = Object.assign(hd, { f: 1 }, { m: 9 });
4 console.log(hd); //{a: 1, b: 2, f: 1, m: 9}
5
```

#计算属性

对象属性可以通过表达式计算定义，这在动态设置属性或执行属性方法时很好用。

```
1 let id = 0;
2 const user = {
3   [`id-${id++}`]: id,
4   [`id-${id++}`]: id,
5   [`id-${id++}`]: id
6 };
7 console.log(user);
8
```

使用计算属性为文章定义键名

```
1 const lessons = [
2   {
3     title: "媒体查询响应式布局",
4     category: "css"
5   },
6   {
7     title: "FLEX 弹性盒模型",
8     category: "css"
9   },
10  {
11    title: "MYSQL多表查询随意操作",
12    category: "mysql"
13  }
14 ];
15 let lessonObj = lessons.reduce((obj, cur, index) => {
16   obj[`${cur["category"]}-${index}`] = cur;
17   return obj;
18 });
```

```
18  }, {}));
19  console.log(lessonObj); //{css-0: {...}, css-1: {...}, mysql-2: {...}}
20  console.log(lessonObj["css-0"]); //{title: "媒体查询响应式布局", category: "css"}
21
```

#传值操作

对象是引用类型赋值是传址操作，后面会介绍对象的深、浅拷贝操作

```
1  let user = {
2    name: '后盾人'
3  };
4  let hd = {
5    stu: user
6  };
7  hd.stu.name = 'hdcms';
8  console.log(user.name); //hdcms
9
```

#遍历对象

#获取内容

使用系统提供的 API 可以方便获取对象属性与值

```
1  const hd = {
2    name: "后盾人",
3    age: 10
4  };
5  console.log(Object.keys(hd)); //["name", "age"]
6  console.log(Object.values(hd)); //["后盾人", 10]
7  console.table(Object.entries(hd)); //[[ "name", "后盾人"], [ "age", 10]]
8
```

#for/in

使用for/in遍历对象属性


```
1  const hd = {
2    name: "后盾人",
3    age: 10
4  };
5  for (let key in hd) {
6    console.log(key, hd[key]);
7  }
8
```

#for/of

`for/of` 用于遍历迭代对象，不能直接操作对象。但 `Object` 对象的 `keys/` 方法返回的是迭代对象。

```
1  const hd = {
2    name: "后盾人",
3    age: 10
4  };
5  for (const key of Object.keys(hd)) {
6    console.log(key);
7  }
8
```

获取所有对象属性

```
1  const hd = {
2    name: "后盾人",
3    age: 10
4  };
5  for (const key of Object.values(hd)) {
6    console.log(key);
7  }
8
```

同时获取属性名与值

```
1  for (const array of Object.entries(hd)) {
2    console.log(array);
3  }
```

```
3  }  
4
```

使用扩展语法同时获取属性名与值

```
1  for (const [key, value] of Object.entries(hd)) {  
2    console.log(key, value);  
3  }  
4
```

添加元素 DOM 练习

```
1  let lessons = [  
2    { name: "js", click: 23 },  
3    { name: "node", click: 192 }  
4  ];  
5  let ul = document.createElement("ul");  
6  for (const val of lessons) {  
7    let li = document.createElement("li");  
8    li.innerHTML = `课程:${val.name},点击数:${val.click}`;  
9    ul.appendChild(li);  
10 }  
11 document.body.appendChild(ul);  
12
```

#对象拷贝

对象赋值时复制的内存地址，所以一个对象的改变直接影响另一个

```
1  let obj = {  
2    name: '后盾人',  
3    user: {  
4      name: 'hdcms'  
5    }  
6  }  
7  let a = obj;  
8  let b = obj;  
9  a.name = 'lisi';  
10 console.log(b.name); //lisi
```

#浅拷贝

使用`for/in`执行对象拷贝

```
1 let obj = {name: "后盾人"};
2
3 let hd = {};
4 for (const key in obj) {
5   hd[key] = obj[key];
6 }
7
8 hd.name = "hdcms";
9 console.log(hd);
10 console.log(obj);
11
```

`Object.assign` 函数可简单的实现浅拷贝，它是将两个对象的属性叠加后面对象属性会覆盖前面对象同名属性。

```
1 let user = {
2   name: '后盾人'
3 };
4 let hd = {
5   stu: Object.assign({}, user)
6 };
7 hd.stu.name = 'hdcms';
8 console.log(user.name); //后盾人
9
```

使用展示语法也可以实现浅拷贝

```
1 let obj = {
2   name: "后盾人"
3 };
4 let hd = { ...obj };
5 hd.name = "hdcms";
6 console.log(hd);
```

```
7 console.log(obj);  
8
```

#深拷贝

浅拷贝不会将深层的数据复制

```
1 let obj = {  
2   name: '后盾人',  
3   user: {  
4     name: 'hdcms'  
5   }  
6 }  
7 let a = obj;  
8 let b = obj;  
9  
10 function copy(object) {  
11   let obj = {}  
12   for (const key in object) {  
13     obj[key] = object[key];  
14   }  
15   return obj;  
16 }  
17 let newObj = copy(obj);  
18 newObj.name = 'hdcms';  
19 newObj.user.name = 'houdunren.com';  
20 console.log(newObj);  
21 console.log(obj);  
22
```

是完全的复制一个对象，两个对象是完全独立的对象

```
1 let obj = {  
2   name: "后盾人",  
3   user: {  
4     name: "hdcms"  
5   },  
6   data: []  
7 };
```

```

8
9 function copy(object) {
10   let obj = object instanceof Array ? [] : {};
11   for (const [k, v] of Object.entries(object)) {
12     obj[k] = typeof v == "object" ? copy(v) : v;
13   }
14   return obj;
15 }
16
17 let hd = copy(obj);
18 hd.data.push("向军");
19 console.log(JSON.stringify(hd, null, 2));
20 console.log(JSON.stringify(obj, null, 2));
21

```

#构造函数

对象可以通过内置或自定义的构造函数创建。

#工厂函数

在函数中返回对象的函数称为工厂函数，工厂函数有以下优点

- 减少重复创建相同类型对象的代码
- 修改工厂函数的方法影响所有同类对象

使用字面量创建对象需要复制属性与方法结构

```

1  const xj = {
2    name: "向军",
3    show() {
4      console.log(this.name);
5    }
6  };
7  const hd = {
8    name: "后盾人",
9    show() {
10     console.log(this.name);
11   }
12 };
13

```

使用工厂函数可以简化这个过程

```
1 function stu(name) {
2   return {
3     name,
4     show() {
5       console.log(this.name);
6     }
7   };
8 }
9 const lisi = stu("李四");
10 lisi.show();
11 const xj = stu("向军");
12 xj.show();
13
```

#构造函数

和工厂函数相似构造函数也用于创建对象，它的上下文为新的对象实例。

- 构造函数名每个单词首字母大写即

Pascal 命名规范

- this指当前创建的对象
- 不需要返回

this系统会自动完成

- 需要使用

new关键词生成对象

```
1 function Student(name) {
2   this.name = name;
3   this.show = function() {
4     console.log(this.name);
5   };
6   //不需要返回，系统会自动返回
7   // return this;
8 }
9 const lisi = new Student("李四");
10 lisi.show();
11 const xj = new Student("向军");
```

```
12 xj.show();
13
```

如果构造函数返回对象，实例化后的对象将是此对象

```
1 function ArrayObject(...values) {
2   const arr = new Array();
3   arr.push.apply(arr, values);
4   arr.string = function(sym = "|") {
5     return this.join(sym);
6   };
7   return arr;
8 }
9 const array = new ArrayObject(1, 2, 3);
10 console.log(array);
11 console.log(array.string("-"));
12
```

#严格模式

在严格模式下方法中的`this`值为 `undefined`，这是为了防止无意的修改 `window` 对象

```
1 "use strict";
2 function User() {
3   this.show = function() {
4     console.log(this);
5   };
6 }
7 let hd = new User();
8 hd.show(); //User
9
10 let xj = hd.show;
11 xj(); //undefined
12
```

#内置构造

JS 中大部分数据类型都是通过构造函数创建的。

```
1  const num = new Number(99);
2  console.log(num.());
3
4  const string = new String("后盾人");
5  console.log(string.());
6
7  const boolean = new Boolean(true);
8  console.log(boolean.());
9
10 const date = new Date();
11 console.log(date.() * 1);
12
13 const regexp = new RegExp("\\d+");
14 console.log(regexp.test(99));
15
16 let hd = new Object();
17 hd.name = "后盾人";
18 console.log(hd);
19
```

字面量创建的对象，内部也是调用了 `Object` 构造函数

```
1  const hd = {
2    name: "后盾人"
3  };
4  console.log(hd.); //f Object() { [native code] }
5
6  //下面是使用构造函数创建对象
7  const hdcms = new Object();
8  hdcms.title = "开源内容管理系统";
9  console.log(hdcms);
10
```

#对象函数

在JS中函数也是一个对象

```
1  function hd(name) {}
2
```



```
3 console.log(hd.());
4 console.log(hd.length);
5
```

函数是由系统内置的 `Function` 构造函数创建的

```
1 function hd(name) {}
2
3 console.log(hd.);
4
```

下面是使用内置构造函数创建的函数

```
1 const User = new Function(`name`, `
2   this.name = name;
3   this.show = function() {
4     return this.name;
5   };
6 `
7 );
8
9 const lisi = new User("李四");
10 console.log(lisi.show());
11
```

#抽象特性

将复杂功能隐藏在内部，只开放给外部少量方法，更改对象内部的复杂逻辑不会对外部调用造成影响即抽象。

下面的手机就是抽象的好例子，只开放几个按钮给用户，复杂的工作封装在手机内部，程序也应该如此。



#问题分析

下例将对象属性封装到构造函数内部

```
1 function User(name, age) {  
2   this.name = name;  
3   this.age = age;  
4   this.info = function() {  
5     return this.age > 50 ? "中年人" : "年轻人";  
6   };  
7   this.about = function() {  
8     return `${this.name}是${this.info()}`;  
9   };  
10 }  
11 let lisi = new User("李四", 22);  
12 console.log(lisi.about());  
13
```

#抽象封装

上例中的方法和属性仍然可以在外部访问到，比如 `info` 方法只是在内部使用，不需要被外部访问到这会破坏程序的内部逻辑。

下面使用闭包特性将对象进行抽象处理

```
1 function User(name, age) {
```

```
2 let data = { name, age };
3 let info = function() {
4     return data.age > 50 ? "中年人" : "年轻人";
5 };
6 this.message = function() {
7     return `${data.name}是${info()}`;
8 };
9 }
10 let lisi = new User("后盾人", 22);
11 console.log(lisi.message());
12
```

#属性特征

JS 中可以对属性的访问特性进行控制。

#查看特征

使用 `Object.getOwnPropertyDescriptor` 查看对象属性的描述。

```
1 "use strict";
2 const user = {
3     name: "向军",
4     age: 18
5 };
6 let desc = Object.getOwnPropertyDescriptor(user, "name");
7 console.log(JSON.stringify(desc, null, 2));
8
```

使用 `Object.getOwnPropertyDescriptors` 查看对象所有属性的描述

```
1 "use strict";
2 const user = {
3     name: "向军",
4     age: 18
5 };
6 let desc = Object.getOwnPropertyDescriptors(user);
7 console.log(JSON.stringify(desc, null, 2));
8
```

属性包括以下四种特性

特性	说明	默认值
configurable	能否使用 delete、能否需改属性特性、或能否修改访问器属性	true
enumerable	对象属性是否可通过 for-in 循环，或 Object.keys() 读取	true
writable	对象属性是否可修改	true
value	对象属性的默认值	undefined

#设置特征

使用 `Object.defineProperty` 方法修改属性特性，通过下面的设置属性 name 将不能被遍历、删除、修改。

```
1  "use strict";
2  const user = {
3    name: "向军"
4  };
5  Object.defineProperty(user, "name", {
6    value: "后盾人",
7    writable: false,
8    enumerable: false,
9    configurable: false
10 });
11
```

通过执行以下代码对上面配置进行测试，请分别打开注释进行测试

```
1  // 不允许修改
2  // user.name = "向军"; //Error
3
4  // 不能遍历
5  // console.log(Object.keys(user));
6
7  //不允许删除
8  // delete user.name;
9  // console.log(user);
```

```
10
11 //不允许配置
12 // Object.defineProperty(user, "name", {
13 //   value: "后盾人",
14 //   writable: true,
15 //   enumerable: false,
16 //   configurable: false
17 // });
18
```

使用 `Object.defineProperties` 可以一次设置多个属性，具体参数和上面介绍的一样。

```
1 "use strict";
2 let user = {};
3 Object.defineProperties(user, {
4   name: { value: "向军", writable: false },
5   age: { value: 18 }
6 });
7 console.log(user);
8 user.name = "后盾人"; //TypeError
9
```

#禁止添加

`Object.preventExtensions` 禁止向对象添加属性

```
1 "use strict";
2 const user = {
3   name: "向军"
4 };
5 Object.preventExtensions(user);
6 user.age = 18; //Error
7
```

`Object.isExtensible` 判断是否能向对象中添加属性

```
1 "use strict";
2 const user = {
3   name: "向军"
```

```
4  };
5  Object.preventExtensions(user);
6  console.log(Object.isExtensible(user)); //false
7
```

#封闭对象

`Object.seal()` 方法封闭一个对象，阻止添加新属性并将所有现有属性标记为 `configurable: false`

```
1  "use strict";
2  const user = {
3    name: "后盾人",
4    age: 18
5  };
6
7  Object.seal(user);
8  console.log(
9    JSON.stringify(Object.getOwnPropertyDescriptors(user), null, 2)
10 );
11
12 Object.seal(user);
13 console.log(Object.isSealed(user));
14 delete user.name; //Error
15
```

`Object.isSealed` 如果对象是密封的则返回 `true`，属性都具有 `configurable: false`。

```
1  "use strict";
2  const user = {
3    name: "向军"
4  };
5  Object.seal(user);
6  console.log(Object.isSealed(user)); //true
7
```

#冻结对象

`Object.freeze` 冻结对象后不允许添加、删除、修改属性，`writable`、`configurable` 都标记为 `false`

```
1 "use strict";
2 const user = {
3   name: "向军"
4 };
5 Object.freeze(user);
6 user.name = "后盾人"; //Error
7
```

`Object.isFrozen()` 方法判断一个对象是否被冻结

```
1 "use strict";
2 const user = {
3   name: "向军"
4 };
5 Object.freeze(user);
6 console.log(Object.isFrozen(user));
7
```

#属性访问器

getter 方法用于获得属性值，setter 方法用于设置属性，这是 JS 提供的存取器特性即使用函数来管理属性。

- 用于避免错误的赋值
- 需要动态监测值的改变
- 属性只能在访问器和普通属性任选其一，不能共同存在

#getter/setter

向对是地用户的年龄数据使用访问器监控控制

```
1 "use strict";
2 const user = {
3   data: { name: '后盾人', age: null },
4   set age(value) {
5     if (typeof value !== "number" || value > 100 || value < 10) {
6       throw new Error("年龄格式错误");
7     }
8     this.data.age = value;
9   },
10 }
```

```

10  get age() {
11      return `年龄是: ${this.data.age}`;
12  }
13 };
14 user.age = 99;
15 console.log(user.age);
16

```

下面使用 getter 设置只读的课程总价

```

1  let Lesson = {
2      lists: [
3          { name: "js", price: 100 },
4          { name: "mysql", price: 212 },
5          { name: "vue.js", price: 98 }
6      ],
7      get total() {
8          return this.lists.reduce((t, b) => t + b.price, 0);
9      }
10 };
11 console.log(Lesson.total); //410
12 Lesson.total = 30; //无效
13 console.log(Lesson.total); //410
14

```

下面通过设置站网站名称与网址体验[getter/setter](#)批量设置属性的使用

```

1  const web = {
2      name: "后盾人",
3      url: "houdunren.com",
4      get site() {
5          return `${this.name} ${this.url}`;
6      },
7      set site(value) {
8          [this.name, this.url] = value.split(",");
9      }
10 };
11 web.site = "后盾人,hdcms.com";
12 console.log(web.site);

```


下面是设置 token 储取的示例，将业务逻辑使用getter/setter处理更方便，也方便其他业务的复用。

```

1  let Request = {
2    get token() {
3      let con = localStorage.getItem('token');
4      if (!con) {
5        alert('请登录后获取token')
6      } else {
7        return con;
8      }
9    },
10   set token(con) {
11     localStorage.setItem('token', con);
12   }
13 };
14 // Request.token = 'houdunren'
15 console.log(Request.token);
16

```

定义内部私有属性

```

1  "use strict";
2  const user = {
3    get name() {
4      return this._name;
5    },
6    set name(value) {
7      if (value.length <= 3) {
8        throw new Error("用户名不能小于三位");
9      }
10     this._name = value;
11   }
12 };
13 user.name = "后盾人教程";
14 console.log(user.name);
15

```

#访问器描述符

使用 `defineProperty` 可以模拟定义私有属性，从而使用面向对象的抽象特性。

```
1 function User(name, age) {
2   let data = { name, age };
3   Object.defineProperty(this, {
4     name: {
5       get() {
6         return data.name;
7       },
8       set(value) {
9         if (value.trim() == "") throw new Error("无效的用户名");
10        data.name = value;
11      }
12    },
13    age: {
14      get() {
15        return data.name;
16      },
17      set(value) {
18        if (value.trim() == "") throw new Error("无效的用户名");
19        data.name = value;
20      }
21    }
22  });
23 }
24 let hd = new User("后盾人", 33);
25 console.log(hd.name);
26 hd.name = "向军1";
27 console.log(hd.name);
28
```

上面的代码也可以使用语法糖 `class` 定义

```
1 "use strict";
2 const DATA = Symbol();
3 class User {
4   (name, age) {
```

```

5     this[DATA] = { name, age };
6 }
7 get name() {
8     return this[DATA].name;
9 }
10 set name(value) {
11     if (value.trim() == "") throw new Error("无效的用户名");
12     this[DATA].name = value;
13 }
14 get age() {
15     return this[DATA].name;
16 }
17 set age(value) {
18     if (value.trim() == "") throw new Error("无效的用户名");
19     this[DATA].name = value;
20 }
21 }
22 let hd = new User("后盾人", 33);
23 console.log(hd.name);
24 hd.name = "向军1";
25 console.log(hd.name);
26 console.log(hd);
27

```

#闭包访问器

下面结合闭包特性对属性进行访问控制

- 下例中访问器定义在函数中，并接收参数 v
- 在 get() 中通过闭包返回 v
- 在 set() 中修改了 v，这会影响 get() 访问的闭包数据 v

```

1 let data = {
2     name: 'houdunren.com',
3 }
4 for (const [key, value] of Object.entries(data)) {
5     observer(data, key, value)
6 }
7
8 function observer(data, key, v) {

```

```
9   Object.defineProperty(data, key, {
10     get() {
11       return v
12     },
13     set(newValue) {
14       v = newValue
15     },
16   })
17 }
18 data.name = '后盾人'
19 console.dir(data.name) //后盾人
20
```

#代理拦截

代理（拦截器）是对象的访问控制，`setter/getter` 是对单个对象属性的控制，而代理是对整个对象的控制。

- 读写属性时代码更简洁
- 对象的多个属性控制统一交给代理完成
- 严格模式下

set 必须返回布尔值

#使用方法

```
1  "use strict";
2  const hd = { name: "后盾人" };
3  const proxy = new Proxy(hd, {
4    get(obj, property) {
5      return obj[property];
6    },
7    set(obj, property, value) {
8      obj[property] = value;
9      return true;
10   }
11 });
12 proxy.age = 10;
13 console.log(hd);
14
```

#代理函数

如果代理以函数方式执行时，会执行代理中定义 `apply` 方法。

- 参数说明：函数，上下文对象，参数

下面使用 `apply` 计算函数执行时间

```
1 function factorial(num) {
2   return num == 1 ? 1 : num * factorial(num - 1);
3 }
4 let proxy = new Proxy(factorial, {
5   apply(func, obj, args) {
6     console.time("run");
7     func.apply(obj, args);
8     console.timeEnd("run");
9   }
10 });
11 proxy.apply(this, [1, 2, 3]);
12
```

#截取字符

下例中对数组进行代理，用于截取标题操作

```
1 const stringDot = {
2   get(target, key) {
3     const title = target[key].title;
4     const len = 5;
5     return title.length > len
6       ? title.substr(0, len) + "...".repeat(3)
7       : title;
8   }
9 };
10 const lessons = [
11   {
12     title: "媒体查询响应式布局",
13     category: "css"
14   },
15   {
16     title: "FLEX 弹性盒模型",
17   }
18 ];
```

```

17     category: "css"
18   },
19   {
20     title: "MYSQL多表查询随意操作",
21     category: "mysql"
22   }
23 ];
24 const stringDotProxy = new Proxy(lessons, stringDot);
25 console.log(stringDotProxy[0]);
26

```

#双向绑定

下面通过代理实现vue 等前端框架的数据绑定特性特性。



```

1  <body>
2  <input type="text" v-model="title" />
3  <input type="text" v-model="title" />
4  <div v-bind="title"></div>
5  </body>
6  <script>
7  function View() {
8      //设置代理拦截
9      let proxy = new Proxy(
10         {},
11         {
12             get(obj, property) {},
13             set(obj, property, value) {
14                 obj[property] = value;
15                 document
16                 .querySelectorAll(

```

```

17         `[v-model="${property}"],[v-bind="${property}"]`
18     )
19     .forEach(el => {
20         el.innerHTML = value;
21         el.value = value;
22     });
23 }
24 }
25 );
26 //初始化绑定元素事件
27 this.run = function() {
28     const els = document.querySelectorAll("[v-model]");
29     els.forEach(item => {
30         item.addEventListener("keyup", function() {
31             proxy[this.getAttribute("v-model")] = this.value;
32         });
33     });
34 };
35 }
36 let view = new View().run();
37

```

#表单验证



```

1 <style>
2   body {
3     padding: 50px;
4     background: #34495e;

```

```
5   }
6   input {
7     border: solid 10px #ddd;
8     height: 30px;
9   }
10  .error {
11    border: solid 10px red;
12  }
13 </style>
14 <body>
15   <input type="text" validate rule="max:12,min:3" />
16   <input type="text" validate rule="max:3,isNumber" />
17 </body>
18 <script>
19   "use strict";
20   //验证处理类
21   class Validate {
22     max(value, len) {
23       return value.length <= len;
24     }
25     min(value, len) {
26       return value.length >= len;
27     }
28     isNumber(value) {
29       return /^d+$/ .test(value);
30     }
31   }
32
33   //代理工厂
34   function makeProxy(target) {
35     return new Proxy(target, {
36       get(target, key) {
37         return target[key];
38       },
39       set(target, key, el) {
40         const rule = el.getAttribute("rule");
41         const validate = new Validate();
42         let state = rule.split(",").every(rule => {
43           const info = rule.split(":");
44           return validate[info[0]](el.value, info[1]);
```



```

45         });
46         el.classList[state ? "remove":"add"]("error");
47         return true;
48     }
49     });
50 }
51
52 const nodes = makeProxy(document.querySelectorAll("[validate]"));
53 nodes.forEach((item, i) => {
54     item.addEventListener("keyup", function() {
55         nodes[i] = this;
56     });
57 });
58 </script>
59

```

#JSON

- json 是一种轻量级的数据交换格式，易于人阅读和编写。
- 使用

json 数据格式是替换

xml 的最佳方式，主流语言都很好的支持

json 格式。所以

json 也是前后台传输数据的主要格式。

- json 标准中要求使用双引号包裹属性，虽然有些语言不强制，但使用双引号可避免多程序间传输发生错误语言错误的发生。

#声明定义

基本结构

```

1  let hd = {
2      "title": "后盾人",
3      "url": "houdunren.com",
4      "teacher": {
5          "name": "向军大叔",
6      }
7  }
8  console.log(hd.teacher.name);

```

数组结构

```
1 let lessons = [  
2   {  
3     "title": '媒体查询响应式布局',  
4     "category": 'css',  
5     "click": 199  
6   },  
7   {  
8     "title": 'FLEX 弹性盒模型',  
9     "category": 'css',  
10    "click": 12  
11  },  
12  {  
13    "title": 'MYSQL多表查询随意操作',  
14    "category": 'mysql',  
15    "click": 89  
16  }  
17 ];  
18  
19 console.log(lessons[0].title);  
20
```

#序列化

序列化是将 `json` 转换为字符串，一般用来向其他语言传输使用。

```
1 let hd = {  
2   "title": "后盾人",  
3   "url": "houdunren.com",  
4   "teacher": {  
5     "name": "向军大叔",  
6   }  
7 }  
8 console.log(JSON.stringify(hd));  
9 //{"title":"后盾人","url":"houdunren.com","teacher":{"name":"向军大叔"}}  
10
```

根据第二个参数指定保存的属性

```
1 console.log(JSON.stringify(hd, ['title', 'url']));
2 //{"title":"后盾人","url":"houdunren.com"}
3
```

第三个是参数用来控制 TAB 数量，如果字符串则为前导字符。

```
1 let hd = {
2   "title": "后盾人",
3   "url": "houdunren.com",
4   "teacher": {
5     "name": "向军大叔",
6   }
7 }
8 console.log(JSON.stringify(hd, null, 4));
9
```

为数据添加 `toJSON` 方法来自定义返回格式

```
1 let hd = {
2   "title": "后盾人",
3   "url": "houdunren.com",
4   "teacher": {
5     "name": "向军大叔",
6   },
7   "toJSON": function () {
8     return {
9       "title": this.url,
10      "name": this.teacher.name
11    };
12  }
13 }
14 console.log(JSON.stringify(hd)); //{"title":"houdunren.com","name":"向军大叔"}
15
```

#反序列化

使用 `JSON.parse` 将字符串 `json` 解析成对象

```
1 let hd = {
2   "title": "后盾人",
3   "url": "houdunren.com",
4   "teacher": {
5     "name": "向军大叔",
6   }
7 }
8 let jsonStr = JSON.stringify(hd);
9 console.log(JSON.parse(jsonStr));
10
```

使用第二个参数函数来对返回的数据二次处理

```
1 let hd = {
2   title: "后盾人",
3   url: "houdunren.com",
4   teacher: {
5     name: "向军大叔"
6   }
7 };
8 let jsonStr = JSON.stringify(hd);
9 console.log(
10   JSON.parse(jsonStr, (key, value) => {
11     if (key == "title") {
12       return `[推荐] ${value}`;
13     }
14     return value;
15   })
16 );
17
```

#Reflect

Reflect 是一个内置的对象，它提供拦截 JavaScript 操作的方法

- Reflect并非一个构造函数，所以不能通过 `new` 运算符对其进行调用