# JS

## String

### API

**注意：**

所有字符串 API 都无权修改原字符串，

#### 1. 大小写转换

`str.toUpperCase()` 将字符串所有字母转为 大写

`str.toLowerCase()` 将字符串所有字母转为 小写

使用场景：不区分大小写进行判断的时候

```
var code = "EaS6";
do{
    var codeInput = prompt("请输入验证码：" + code);
    if(codeInput.toUpperCase() !== code.toUpperCase()){
        alert("验证码错误，请重试");
    }else{
        break;
    }
}while(true);
document.write("验证通过");
```

#### 2. 获取指定位置字符

str[i] => `str.charAt(i)`

获取指定位置字符的 unicode 号

`str.charCodeAt(i);`

将 unicode 号反向转回字符

`String.fromCharCode(i);`

#### 3. 选取子字符串

`str.slice(strrt, end);` 优点：支持附属参数，表示倒数

`str.substring(start, end);` 缺点：不支持附属参数 解决：str.length-n 代替 -n

`str.substr(start, n)` n 表示个数

**4. 查找关键词：4 种**

**1. 查找一个固定关键词位置**

`var i = str.indexOf(keyWord, from);`

在 str 中，从 from 位置开始，找下一个 keyWord 所在位置

如果没有 from，默认从 0 位置开始

返回值：找到的第一个关键词的第一个字的下标位置

如果找不到，返回 -1

```
// 查找所有的关键词首字符下标
var str = "no zuo no die, no can no bibi";
  var i = -1;
do{
  var i = str.indexOf("no", i ++);
  if(i != -1){
    console.log(i);
  }else{
    break;
  }
}while(true);
```

`var i = str.lastIndexOf(keyWord);`

在 str 中找最后一个 keyWord 的位置

```
var url = "http://tmooc.cn/imgs/logo.png";
var i = url.lastIndexOf("/");
var dot = url.lastIndexOf(".");
// 找出 url 中的 文件名 和 扩展名
var fileName = url.slice(i + 1);;
var extend = url.slice(dot);
console.log(fileName, extend);
```

问题：只能查找一种固定的关键词

解决：正则表达式

**2. 判断是否包含符合正则的关键词**

`var i = str.search(/regexp/);`

在 str 中查找第一个符合正则要求关键词位置，找不到，返回 `-1`

问题：正则默认区分大小写

解决：在第二个 `/` 后加 `i` `var i = str.search(/regexp/i);`

注意：只能找 **第一个** 关键词的 **位置**

**3. 查找所有关键词的内容**

`var arr = str.match(/正则/i);`

在 str 中查找符合正则表达式要求的关键词，并将内容放入 arr 中

语法：

1. 默认只找第一个，但会返回位置

    arr : [0: 关键词, index: 位置]

2. 查找所有关键词的内容

注意：只能获得关键词内容，无法获取位置

**4. 即查找每个关键词的内容，又获得每个关键词的位置**

---

**5. 替换关键词**

1. 简单替换

   将所有敏感词都替换为统一的值

   `var str = str.replace(/正则/ig, value);`

2. 高级替换

   根据关键词的不同，动态选择不同的替换值

```
str = str.replace(/正则/ig, function(){
    return kw.toUpperCase();
});
```

---

**6. 切割**

切割：用指定 **分隔符** 将字符串分割为多段 **子字符串**

1. 简单切割：分隔符是固定的

`str.split("");`

固定套路：打散字符串为字符数组

1. 复杂切割：分隔符不是固定的

`str.split(/ /);`

---

# 正则表达式

> regular expression

概念：规定一个字符串中字符出现规律的规则

使用场景：

1. 模糊查找多种敏感词
    1. 验证格式

形式：

1. 一个关键词自身，就是一个最简单的正则表达式
2. 字符集：

   规定一位字符上多个备选字符列表的集合

   只要一位字符上有多种字时

   语法：[备选字符列表]

   强调：一个字符集只能匹配一位字符

   简写：如果备选字符列表中部分字符连接，可用 `-` 省略中间字符

   常用简写：一位数字：[0-9]

   一位小写字母：[a-z]

   一位大写字母：[A-Z]

   一位字母：[A-Za-z]

   一位汉字：[\u4e00-\u9fa5]

   除了： `[^xxx]` ：除了 `xxx` 都选中

   EX: 车牌号：`[\u4e00-\u9fa5][A-Z]·[A-Z0-9][A-Z0-9][A-Z0-9][A-Z0-9][A-Z0-9]`

---

3. 预定义字符集

   对常用字符集的进一步简写

   1. `\d` 一位数字
   2. `\w` 一位字母，数字或 _
   3. `\s` 一位空字符：空格，tab，...
   4. `.` 通配符

---

字符集的问题：只能固定一个字符，无法灵活定义数量；

解决：量词

1. 量词

   定义：规定一位字符出现次数的规则

   when：只要规定字符出现的次数

   how：字符集量词

   强调：一个量词默认仅修饰相邻的前一个字符集

分类：

1. 有明确数量范围：

   {m, n} 至少 m 个，最多 n 个

   {m,} 至少 m 个

{m} 只有 m 个

2. 没有明确数量范围

? 可有可无，最多一个

* 可有可无，多了无限

+ 至少一个，多了无限

---

1. 选择和分组
    1. 选择： 或 |

       规则1 | 规则2
    2. 分组：(多个规则)

       作用：将多个规则用 ()包裹为一组

       why：量词默认只修饰相邻的一个字符集

       when：只要希望一个量词同时修饰多个字符集的时候

EX: 身份证号：15位数字 + 2位数字 + 1位数字或X

`\d{15}(\d{2}[0-9X])?`

EX: 手机号： +86 | 0086 + 空字符 + 1 + [3-8] + 9位数字

`(\+86|0086)?\s*1[3-8]\d{9}`

---

1. 匹配特殊位置

字符串开头：`^`

字符串结尾：`$`

when：仅希望匹配开头或结尾的内容时

EX: "  zhang  dong  "

`^\s+` 仅匹配开头的空字符

`\s+$` 仅匹配结尾的空字符

`^\s+|\s+$` 即匹配开头，又匹配结尾的空字符

> 单词边界： \b 包括：空格，标点，开头，结尾

1. 匹配一个独立的单词时 `\b[a-z]+\b`
2. 仅匹配单词开头或结尾的字母 `\b[a-z]` `[a-z]\b`

## 定义

专门封装一条正则表达式，并提供用正则表达式执行验证和查找的 API

## 何时

只要用正则表达式执行验证和查找功能，都要用正则表达式对象

## 如何

#### 创建方式

**1. 直接量（字面量）**

`var reg = /RegExp/;`

何时：如果正则表达式是固定的，不需要动态的生成

**2. new**

`var reg = new RegExp("RegExp", "ig");`

何时：如果正则表达式需要动态生成

---

---

## API

**1. 验证**

`var bool = reg.test(str);`

用 reg 验证 str 的格式是否符合要求

> 问题：默认，只要 str 中部分和 reg 匹配，就返回 `true`；
>
> 解决：`var reg = /^正则$/;`

---

**2. 查找关键词**

`var arr = reg.exec(str);`

在 str 中查找下一个符合 reg 要求的关键词的内容和位置

> 强调：一次只能找一个关键词
>
> 返回值：arr:[0: 关键词, index: 位置]

> exec 可自动找下一个：
>
> 每个 reg 对象上有一个lastIndex 属性，开始为 0，记录着下次查找的位置
>
> 每次找到关键词，exec 自动修改 reg.lastIndex = 本次的位置 + 关键词长度

---

# Math

**定义**

封装数学计算的常量和 API 的对象

**何时**

只要执行数学相关计算

**API**

**1. 取整**

`Math.ceil();` 向上取整：只要超过，就取下一个整数

`Math.floor();` 向下取整：只要超过，就舍弃小数部分

`parseInt(str)` 能去掉数字后非数字字符（单位）

`Math.round();` 四舍五入

| | Math.round(); | n.toFixed(d); |
|---|---|---|
| 灵活 | 只能取整 | 可按任意小数位数取整 |
| 返回值 | Number 类型 | String 类型 |

---

**2. 乘方和开平方**

`Math.pow(num, n);` 乘方

`Math.sqrt();` 开平方

---

**3. 最大值和最小值**

`Math.max();` 最大值

`Math.min();` 最小值

> 问题：不支持获得数组中的最大值和最小值
>
> 解决：Math.max(...arr); // ... ：ES6中的 spread 操作，打散数组为单个元素

---

**4. 随机数**

0 <= `Math.random();` < 1

公式： min ~ max

parseInt(Math.random() * (max - min + 1) + min);

---

**5. 三角函数 反正切**

`Math.atan(对边 / 临边);`

> 局限：只能在正负 45deg 内进行计算
>
> 解决：`Math.atan2( 对边长, 临边长);`

---

## Date

---

定义：封装一个时间，并提供操作时间的 API

场景：只要在程序中存储时间或计算时间

创建方式：

- 创建日期对象，并自动获取当前系统时间

  var now = new Date();

  **强调**：只能获得客户端系统时间 => 用处不大，了解

- 创建日期对象并封装自定义时间

  var birth = new Date("year/month/day h : m : s");

- 将毫秒数转化为日期对象 new Date();

  日期对象本质：日期对象中实际保存的是 **1970 年 1 月 1 日** 至今的毫秒数

  缘由：

  用文字表示时间的两个问题

  字符串不便于计算

  受时区影响

  用毫秒数的优点：

```
便于计算

不会受时区的影响

问题：

人看不懂，且没有 API

解决：

用 new Date(); 将毫秒数转为日期对象，并使用日期的对象提供的简化 API 操作时间

var date = new Date(ms);
```

- 复制一个日期对象

  缘由：日期计算都是直接修改原日期对象

  场景：如果希望同时保留计算前后的两个时间时

  方式：var date2 = new Date(date1);

---

---

## API

### 1. 8 个单位

FullYear Month Date Day(周几)

Hours Minutes Seconds Milliseconds

### 2. get / set

每个单位上都有一对 get / set 方法

其中：date.getXXX(); 获取一个单位上的值

date.setXXX(num); 修改一个单位的数值

特例：Day 没有 set

**3. 取值范围**

| 单位 | 范围 | --=-说明 |
| --- | --- | --- |
| Month | 0 ~ 11 | 计算机中的月份值比现实中小 1 |
| Date | 1 ~ 31 | |
| Day | 0 ~ 6 | |
| Hours | 0 ~ 23 | |
| Minutes / Seconds | 0 ~ 59 | |

## 计算

1. 两日期对象可 **相减** ：结果是 **毫秒差**
2. 对任意单位 **加减** ：

   3 步：

   1. 取值： var now = new Date(); var d = date.getDate();
   2. 计算： d += 7;
   3. 改回去：now.setDate(d);

   简写：

   date.setDate(date.getDate() + 7);

   福利：

   setXXX(); 可自动调整时间进制

   问题：

   setXXX(); 直接修改原日期对象

   解决：

   先复制日期对象副本，再用副本计算
3. 日期格式化：

| API | 描述 |
| --- | --- |
| date.toString() | 转为当地时间的完整格式 |
| date.toLocaleString() | 转为当地时间的简化版格式 |
| date.toLocaleDateString() | 当地时间格式，仅保留日期部分 |
| date.toLocaleTimeString() | 当地时间，仅保留时间部分 |

---

## Error

> - 什么是：
>
>   当错误发生时，自动创建，封装错误信息的对象
> - 什么是错误：
>
>   程序执行过程中，导致程序无法继续执行的异常情况
> - 什么是错误处理：
>
>   即使程序出错，也不会中断推出的一种机制
> - 为什么要进行错误处理：
>
>   程序只要出错，都会闪退
> - 场景：
>
>   只要希望，即使程序出错，也不会中断退出
> - 形式：**try catch**
>
>   ```
>   try{
>       //可能出错的正确代码
>   }catch(e){ //只有出错时才触发
>       //错误处理代码
>       // 提示错误信息，保存进度，记录日志
>   }
>   ```
>
> - **问题**：会额外创建错误对象，降低效率和浪费内存
> - 解决：用 if else 提前预防错误发生

主动抛出错误：

throw new Error("错误提示");

---

## Function

---

## 基本介绍

定义：专门封装一段代码段的对象

场景：只要重用一段代码时

创建形式：

- 函数声明

  function 函数声明(参数列表){

  　　//函数体

  　　//返回值

  }

> 问题：声明提前（hoist）

- 函数表达式

  var 函数名 = function(){}

> 优点：避免声明提前

- new

  var 函数名 = new Function("参数", "参数", ... , "函数体");

## 重载

定义：相同函数名不同参数列表的多个函数，

　　在调用时，可根据传入参数的不同，自动选择匹配的函数执行

场景：只要一个函数根据不同的参数，执行不同的任务时

作用：减少 API 的数量，减轻调用者的负担

形式：JavaScript 默认不支持重载

原因：不允许多个同名函数同时存在

解决：arguments

　　定义：函数内自动创建的，收集所有传入函数参数值的 **类数组对象**

　　vs 数组：

　　　　相同： 1. 下标　　2. length　　3. 遍历

　　　　不同： 类型不同　API 不通用

　　场景：不确定传入函数的参数个数时

## 匿名函数

定义：函数创建后，不保存到任何变量的函数

场景：函数只调用一次时

作用：节约内存 =》匿名函数调用后自动释放

创建方式：

- 回调：创建函数后，自己不调用，而是给另一个函数去调用

  EX：arr.sort(function(a, b){return a - b;});

  arr.replace(/RegExp/ig,function(){return value;});

- 自调：函数定义后，立即调用自己

> 问题：禁止使用全局变量
>
> 理由：全局污染，内存泄漏
>
> 解决：用匿名函数自调包裹所有变量和操作

## 闭包（Closure）

定义：即重用变量，又保护变量不被篡改的机制

原因：全局变量和局部变量都有不可兼得的优缺点

| | **优点** | **缺点** |
|---|---|---|
| 全局变量 | 随处可用，可重复使用 | 造成全局污染和内存泄漏 |
| 局部变量 | 仅函数内可用 | 不可重用 |

何时：如果希望即重用一个变量，又保护变量不被篡改时

如何：3 步

- 用外层函数，包裹内层函数和受保护的变量
- 外层函数返回内层函数到外部
- 调用外层函数，获得内层函数引用，保存在变量中

## OOP

### 概念

程序中都是用对象结构来描述现实中一个具体事物

### 对象

> 用途：描述现实中一个具体事物的程序结构

> 本质：内存中存储多个数据和函数的存储空间
>
> 作用：便于大量数据的维护
>
> 场景：今后，几乎所有程序都是面向对象的
>
> 特点：封装，继承，多态

---

## 封装

创建一个对象，集中存储一个事物的属性和功能

作用：便于大量数据的维护

场景：只要使用面向对象方式编程，都要先封装对象，再使用对象操作数据

创建形式： 3 种

**1. 直接量：**

```
var obj = {
            属性名 : 属性值 ,
            方法名 : function(){
                this.prototype
            },
            ...
}
```

问题：对象自己的方法，要用自己的属性

错误：直接写属性名

因为不带任何前缀的普通变量，只能在作用域对象中找，无权擅自进入某个对象中查找

正确：即使对象

强调：即使对象自己的方法，想访问对象自己的属性，也要加 this

访问对象成员：成员 = 属性 + 方法

Obj.prototype;

对象.方法()

---

**2. new**

var obj = new Object();

obj.prototype = value;

揭示：起始 js 底层一切都是 **关联数组**

> 1. 可随时添加新属性
> 2. 如果访问不存在的成员，不报错，返回 undefined

> 3. obj.prototype = obj["prototype"];

如何选择：

1. 如果属性名时固定的，不需要动态生成，则首选 obj.prototype
2. 如果属性名需要动态生成，则只能用 obj["prototype"]

> 4. 都能用 for in 循环遍历
>
> 问题：一次只能创建一个对象
>
> 反复创建多个结构功能相同的对象时，代码冗余
>
> 解决：**构造函数**

---

**3. 构造函数**

概念：描述一类对象统一结构的函数

场景：如果需要反复创建多个相同结构的对象时

步骤：2 步

> 1. 定义构造函数

function 类型名(属性参数列表){

this.prototype  = values;

...

}

> 2. 调用构造函数反复创建多个新对象

var obj = new 类型名(属性值列表);

new：4 件事

> 1. 创建新的空对象
> 2. 让新对象自动继承构造函数的原型对象
> 3. 调用构造函数，为新对象添加新属性
>
> new 可将构造函数中的 this =》 新对象
>
> 通过为新对象强行赋值新属性的方式
>
> 返回新对象的地址保存在变量中

> 问题：反复创建方法，浪费内存
>
> 解决：继承

---

## 继承（inherit）

---

概念：父对象的成员，子对象无需重复创建，就可直接使用

作用：重用代码，节约内存

场景：多个对象拥有相同的方法定义时

形式：

继承原型对象

原型对象：集中存储多个子对象相同成员的父对象

场景：只要希望有成员可被多个子对象继承使用

语法：

> 创建：不用自己创建
>
> 在定义构造函数时，附赠一个原型对象
>
> 继承：不用手动 设置继承关系
>
> new 创建子对象时，自动让子对象继承父对象
>
> 将公有的方法添加到原型对象中：
>
> 构造函数.prototype.方法 = function(){}

**原型链** ： 由多级父对象逐级继承形成的链式结构

---

## 内置对象

概念：ES标准规定的，浏览器厂商已经实现的，我们直接使用的对象 / 类型

包括：11 个

| String | Number | Boolean | | |
|---|---|---|---|---|
| Array | Date | RegExp | Math | |
| Error | | | | |
| Function | Object | | | |
| global | | | | |

> global (全局作用域对象，在浏览器中，用window代替)
>
> 特例：Math 不能 new，因为不是构造函数
>
> global 也不能 new，因为已经是全局作用域对象
>
> 其余类型都由 2 部分组成：
>
> 1. 构造函数：创建该类型的子对象
> 2. 原型对象：保存该类型所有子对象共享的 API

---

## 包装类型

String Number Boolean

概念：专门封装一个原始类型的值，并提供操作原始类型值的 API 的特殊类型

作用：原始类型的值本身什么 **功能 / 属性** 都没有

场景：只要用原始类型的值调用函数或访问属性时，都会自动使用包装类型

语法：

EX:

```
var n = 345.678; n.toFixed(2);
// typeof(n); // number
// new Number(n).toFixed(2); // 345.68  => delete
```

bishi：判断一个对象是不是数组类型，共有集中方式

typeof 可区分原始类型，对象和函数，不能细致区分对象的具体类型

1. 判断 **原型对象**

   obj.__proto__ === Array.prototype

   Array.prototype.isPrototypeOf(obj)

<img src="../img/isArray.png">

1. 判断 **构造函数**

   obj.constructor === Array

   obj instanceof Array

**问题**：不够严谨

1. 判断对象的 class （内部）属性

   每个对象都有一个隐藏的 class 属性，只在创建对象时保存对象的类型名，之后不随继承关系改变

问题：无法用 . 直接访问

解决：只有顶级父对象 Obejct.prototype中的toString()方法才能输出对象的 class

问题：各类型的原型对象中都重写了 toString() 方法，子对象默认无法调用顶级父对象中的 toString()

解决： call()

语法： 函数名.call(对象)

作用：让对象强行调用调用本来调不到的

1. Array.isArray(obj);

| | 实例方法 | 静态方法（static） |
| --- | --- | --- |
| 概念 | 保存在类型的原型对象中，只允许当前类型的子对象使用。其他类型的子对象不能使用 | 不定义在原型对象中，任何对象都可以使用的方法 |
| 场景 | 只允许当前类型的子对象使用 | 任何对象都可使用 |
| 存储位置 | 所有实例方法都定义在原型对象中 | 直接添加到构造函数对象上 |
| 调用 | 必须先创建该类型的子对象，再用子对象调用 | 不用创建任何子对象，用构造函数直接调用 |

## 多态

概念：同一个函数在不同情况下表现出不同的状态

形式：2 种

1. 重载
2. **重写 (override)**

   概念：如果子对象觉得从父对象继承来的成员不好用！

   可在子对象本地定义同名成员，覆盖父对象同名成员

   原因：子对象觉得从父对象继承来的成员不好用

   场景：如果子对象觉得从父对象继承来的成员不好用

   语法：在子对象本地定义同名成员，覆盖父对象同名成员

## 自定义继承

1. 只修改一个子对象的父对象

   child.__proto__ = father;

   Object.setPrototypeOf(child, father);
2. 修改所有子对象的父对象

   构造函数.prototype = father;

   **强调**：注意时机，定义构造函数后，立即更换！
3. 两种类型间的继承：

问题：两种类型，拥有部分相同的属性结构和方法定义

解决：定义一个公共的抽象父类型

语法： 2 步

1. 定义抽象父类型

   1. 定义父类型构造函数，包含相同的属性结构

> 2. 定义父类型原型对象，包含相同的方法定义

1. 让子类型继承抽象父类型

> 1. 在子类型构造函数中借用父类型构造
>
> 问题：直接调用父类型构造函数，无法为子对象添加父类型中的属性
>
> 原因：直接调用一个函数，其中 this 默认指向 window
>
> 解决：call
>
> > 函数.call(替换 this 的对象, 对象, ...);
> >
> > 将函数内的 this 替换为指定的对象，并传入参数
>
> 总结：将来，只要函数中的 this 不是想要的，都可用 call 替换
>
> 2. 让子类型原型对象继承父类型原型对象

---

## ES5

严格模式：比普通 js 运行要求更严格的模式

原因：js 语言本身有广受诟病的缺陷

场景：今后所有 js 程序，都要运行在严格模式下

如何：2 种

1. 新项目： <script> 或 js 文件顶部：`"use strict"`
2. 旧项目维护：逐个模块向严格模式迁移

---

在 function 函数内部： `"use strict"`

包括：

1. 禁止给未声明的变量赋值

   > 解决：内存泄漏，全局污染

2. 静默失败升级为错误

   > 静默失败：执行不成功，也不报错

3. 普通函数调用中的 this 不在默认指向 window

   > 解决：内存泄漏，全局污染

4. 不建议使用 arguments arguments.callee arguments.caller

   > arguments.callee 指参数列表自身函数引用，比如匿名函数自调
   >
   > arguments.caller 参数列表自身函数执行的环境

---

# 保护对象

## 保护属性

ES5中对象属性分为：

> 内部属性：仅在对象内保存，不能用 `.` 直接访问的属性
>
> 命名属性：可用 `.` 随意访问的属性，分两类
>
> > 数据属性：实际存储属性值的属性
> >
> > 访问器属性：不实际存储属性值，仅提供对另一个数据属性的保护

**保护数据属性：**

数据属性包含四大特性：{

value : 保存属性值 ,

writable : 控制是否可修改 ,

enumerable : 控制是否可被 for in 遍历 ,

configurable : 控制是否可被删除 && 控制是否可修改前两个特性

}

> **强调**
>
> 1. 通常修改 writable 和 enumerable 时，都将configurable 改为 false，双保险
> 2. enumerable 只能防止不被遍历访问到，用 `.` 还是可以被访问
>
> 获取一个属性的四大特性：

```
var obj = Object.getOwnPropertyDescriptor();
```

> 修改一个属性的四大特性：

```
Object.defineProperty(obj, "属性名", {
    四大特性 ： true / false
});
```

> 同时修改多个属性的四大特性：

```
Object.defineProperty(obj, {
    属性名 ： {四大特性之一 ： true / false, 四大特性之一 ： true / false, ...},
    属性名 ： {四大特性之一 ： true / false, 四大特性之一 ： true / false, ...},
    属性名 ： {四大特性之一 ： true / false, 四大特性之一 ： true / false, ...},
    ...
});
```

问题：无法用自定义规则保护属性

解决：访问器属性

---

**访问器属性**

概念：不存储属性值，仅提供对其他数据属性的保护

场景：用自定义规则保护属性时

语法：

前提：有一个被保护的数据属性

四大特性

```
{
  get : function(){return 从受保护的数据属性中取值},
  set : function(val){
    if(true){
      将 val 的值保存给受保护的数据属性
    }else{
      Error
    }
  },
  enumerable : ture,
  configurable : false
}
```

访问器属性只能用 Object.defineProperty 或 Object.defineProperties 添加，不能用直接量添加

用法：

访问器属性的用法和普通属性完全一样：

试图获取属性值时，自动调用 get 方法，

试图修改属性值时，自动调用 set 方法

其中 val 自动获得新值

## 保护结构

---

3 个级别

**1. 防扩展：禁止添加新属性**

`Object.preventExtensions(obj)`

> 原理：
>
> 每个对象中都隐藏着一个内部属性：extensible = true
>
> Object.preventExtensions(obj) => extensible = false;

**2. 密封：在禁止扩展同时禁止删除现有属性**

`Object.seal(obj);`

> 原理：
>
> 1. extensible = false
> 2. configurable = false

**3. 冻结：在密封基础上，禁止修改属性的值**

`Object.freeze(obj)`

> 原理：
>
> 1. extensible = false;
> 2. configurable = false;
> 3. writable = false;

## Object.create()

概念：仅基于一个现有父对象，创建子对象，并继承父对象

场景：如果没有构造函数，也想创建子对象时

语法：var child = Object.create(father, {

// 同 defineProperty

});

## call, apply, bind

概念：替换函数中不想要的 this

- call , apply

  概念：调用一个函数，并 **临时替换** this 未指定对象

  语法：函数.call(对象 , 参数值 , ...)

> call vs apply ? apply 要求传入的多个参数值 必须放在一个数组中整体传入

- bind

  概念：基于原函数，创建一个新函数，并永久绑定 this 和部分参数

## ES6

### let

代替 var, 用于声明提前

var 的问题 ： 会被声明提前

let 的优点：不会被声明提前

> 原理：let 会检查，在 let 之前，不允许使用声明的变量
>
> 重要作用：
>
> 为程序添加了块级作用域：
>
> 块级作用域：if else while for do...while switch
>
> 问题：传统 js 没有块级作用域，块级变量很可能泄漏到外部，造成污染
>
> 解决：let 可将上面的 if .. 编程块级作用域，内部的变量只能在块内使用
>
> 原理：let 其实是 **匿名函数自调**

## 参数增强

默认值：default

function fun(参数1 ， 参数2 ， ..., 参数n = 默认值){}

强调：有默认值的参数必须写在参数列表的末尾

原理：fromi = fromi || 0;

**剩余参数：rest**

作用：代替 arguments

arguments 问题：

1. 不是纯正的数组类型
2. 只能获取所有参数，不能部分获取

场景：不确定参数个数

形式：function fun(参数1, 参数2, ... , ...数组){}

优点：

rest 能获得纯正的数组

可部分获得参数

**散播：spread**

调用函数 传参时，打散数组类型参数为单个值：

fun(参数1, 参数2, 参数3, ...){}

> 场景：如果函数要求单个参数值传入，而给定的参数是一个数组时

> 替代：fun.apply(null, 数组);

---

### 箭头函数

简化一切回调函数或匿名函数自调

> 1. 去掉 `function`，改为 `=>`
> 2. 更简化：如果函数体只有一条执行语句，可省略 `{}`
>
> 如果这条执行语句时 `return` ，一起省略
>
> 如果只有一个参数，可省略 `()`
>
> > 强调：将普通函数改为箭头函数后，函数内的 this 与 外部的 this 通用、
> >
> > 总结：如果希望 函数内外的 this 不通用，不能使用箭头函数 简化！比如：事件处理函数

---

### 模板字符串

代替 `+` 拼接字符串

> 语法：将完整字符串包裹在 `` 中
>
> 作用：内部支持换行等格式
>
> 支持动态内容：${ 变量或表达式 }

## 解构（destruct）

---

简化对象属性 / 数组元素的使用：

> 场景：只要从对象 / 数组中挑选个别需要的属性或元素使用时
>
> 形式：3 种

### 数组解构

从数组中挑出想要的元素来使用，下标对下标：

var [变量1, 变量2, ...] = [值1, 值2, ...]

变量 1 = 值 1，变量 2 = 值 2， ...

### 对象解构

从对象中挑出想要的属性或方法来使用，属性对属性，方法对方法：

var {属性1:变量1, 属性2: 变量2,...} = {属性1:值1, 属性2: 值2,...}

变量1 = 值1, 变量2 = 值2, ...

### 函数参数解构

问题：普通函数参数必须连续传入，不能间隔或跳跃

解决：参数解构

形式：2 步

1. 定义函数时，将参数列表定义为对象格式
2. 传参时，将所有参数值，放入对象中传入

作用：可以选择传入部分参数，且和顺序无关

---

## for of

最简化的遍历数组或类数对象的方法

遍历数组： 3 种

```
// 1.
for(var i = 0; i < arr.length; i ++){
      var elem = arr[i];
   }
// 2.
arr.forEach((elem, i, arr) => {
    ...
   })
// 3.
for(var elem of arr){
  ...
}
```

> 注意：
>
> > for...of 只能遍历数组/类数组对象
> >
> > for...in 专门遍历关联数组和对象属性

---

### 封装：

> 1. 直接量
>
>  var obj = {
>
>  属性名 : 变量名 , // 如果变量名 == 属性名，只写一个即可
>
>  函数名(){}
>
>  }
>
> 2. 构造函数
>
>  用 class 类型名 {} 包裹构造函数和原型对象方法
>
>  构造函数名提升为类型名，构造函数律变更为 constructor

> 直接定义在 class 中的方法，默认就保存进原型对象中

---

### 继承：

两种类型间正式的继承

1. 用 extends 父类型，代替 Object.setPrototypeOf()
2. 用 super

## Promise

作用：代替所有异步回调（ex: ajax）

异步回调：在不影响主程序情况下，等待一段时间后，调用回调函数或触发事件

问题：callback hell

因为提前传入回调函数参数，导致嵌套层级很深

解决：Promise

封装一段任务，并提供一个向后继续执行

错误处理：

```
return new Promise(function(open, err){
  // 如果正确执行，则打开
  open() -> .then()
  // 否则，打开
  err("error msg") -> .catch()
}).catch((err) => { throw err })
```

> 说明：只需要在最后一个 .then

等待多项任务：

```
Promise.all([
  fun(),
  fun(),
]).then();
```

---