零 基础知识

1. 单词

abort	终止
interceptor	拦截器
pending	未确定的
resolved	已解决的
rejected	拒绝的
iterable	可迭代的
parse	解析
radix	基数
r	radius 半径
invoke	调用
orient	朝向
extend	延伸，扩展
derive	源于，派生
closure	闭包
Regular Expression	正则表达式
sibling	兄弟姐妹
previous	以前的
interval	间隔
navigator	导航员(浏览器类型)
property	特性
extension	扩展
configurable	可配置的
writable	可写的
enumrable	可枚举的
seal	密封
exponent	指数
fraction	小数
epsilon	希腊字母中表示极小值的那个

2. 常见的语法名称的术语

1.2.1. Numeric Literals（数值语法）

- 了解什么形式的才是js中的数值语法

Syntax

NumericLiteral ::
DecimalLiteral
BinaryIntegerLiteral
OctalIntegerLiteral
HexIntegerLiteral

DecimalLiteral ::
DecimalIntegerLiteral . *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits* *ExponentPart*$_{opt}$
DecimalIntegerLiteral *ExponentPart*$_{opt}$

DecimalIntegerLiteral ::
0
NonZeroDigit *DecimalDigits*$_{opt}$

DecimalDigits ::
DecimalDigit
DecimalDigits *DecimalDigit*

DecimalDigit :: **one of**
0 1 2 3 4 5 6 7 8 9

NonZeroDigit :: **one of**
1 2 3 4 5 6 7 8 9

ExponentPart ::
ExponentIndicator *SignedInteger*

ExponentIndicator :: **one of**
e E

SignedInteger ::
DecimalDigits
+ *DecimalDigits*
− *DecimalDigits*

BinaryIntegerLiteral ::
0b *BinaryDigits*
0B *BinaryDigits*

BinaryDigits ::
BinaryDigit
BinaryDigits *BinaryDigit*

BinaryDigit :: **one of**
0 1

OctalIntegerLiteral ::
0o *OctalDigits*
00 *OctalDigits*

OctalDigits ::
OctalDigit
OctalDigits *OctalDigit*

OctalDigit :: **one of**
0 1 2 3 4 5 6 7

HexIntegerLiteral ::
0x *HexDigits*
0X *HexDigits*

HexDigits ::

HexDigit
HexDigits *HexDigit*

HexDigit :: **one of**
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

The *SourceCharacter* immediately following a *NumericLiteral* must not be an *IdentifierStart* or *DecimalDigit*.

NOTE For example: **3in** is an error and not the two input elements **3** and **in**.

A conforming implementation, when processing strict mode code, must not extend, as described in B.1.1, the syntax of *Nur*
LegacyOctalIntegerLiteral, nor extend the syntax of *DecimalIntegerLiteral* to include *NonOctalDecimalIntegerLiteral*.

-
- 注意事项
  - DecimalIntegerLiteral
    - 0
    - 第一位非0的整数
  - DecimalDights
    - 由整数构成（没要求第一位非0）
  - DecimalDight
    - 差一个s
    - 0~9任意一个
  - ExpontPart
    - 指数部分
    - 科学计数法

1.2.2. 重点掌握十进制的语法

DecimalLiteral ::
DecimalIntegerLiteral . *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits* *ExponentPart*$_{opt}$
DecimalIntegerLiteral *ExponentPart*$_{opt}$

-

1.2.3. StringNumericLiteral

- strnumberliteral
- 了解什么是字符型的数值语法

Syntax

StringNumericLiteral :::
StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral* *StrWhiteSpace*$_{opt}$

StrWhiteSpace :::
StrWhiteSpaceChar *StrWhiteSpace*$_{opt}$

StrWhiteSpaceChar :::
WhiteSpace
LineTerminator

StrNumericLiteral :::
StrDecimalLiteral
BinaryIntegerLiteral
OctalIntegerLiteral
HexIntegerLiteral

StrDecimalLiteral :::
StrUnsignedDecimalLiteral
\+ *StrUnsignedDecimalLiteral*
− *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral :::
Infinity
DecimalDigits . *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits* *ExponentPart*$_{opt}$
DecimalDigits *ExponentPart*$_{opt}$

All grammar symbols not explicitly defined above have the definitions used in the Lexical Grammar

NOTE 2 Some differences should be noted between the syntax of a *StringNumericLiteral* and a

- A *StringNumericLiteral* may include leading and/or trailing white space and/or
- A *StringNumericLiteral* that is decimal may have any number of leading **0** digits.
- A *StringNumericLiteral* that is decimal may include a **+** or **-** to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only white space is converted t
- **Infinity** and **-Infinity** are recognized as a *StringNumericLiteral* but not as a

- **.1 Runtime Semantics: MV's** #
- 注意事项（与正常的数值语法相比）
  - 左右可以包含任意数量的空格或者换行符，但转化的时候可以被忽略
  - 整数的左侧可以有任意数量的0，转化时会被忽略
  - 可以用+、-号表示符号
  - 若字符串为空或者只有空格，则会返回+0
  - 字符串型有Infinity和-Infinity，Numeric Literals

3 typeof, instanceof

- typeof只能分辨基本数据类型和object
- instanceof可以用于判断对象类型，即某个对象是不是某个构造对象的实例

4. 用于调试的一些函数

4.1. console.log()与console.dir()

- console.log()会在浏览器控制台打印出信息
- console.dir()可以显示一个对象的所有属性和方法

5 Object.prototype.toString.call(object)

- 每个对象都默认有toString方法
- 如果此方法在自定义对象中未被覆盖，toString() 返回 "[object type]"，其中 type 是对象的类型，返回的类型是string类型
- 应用
- 判断对象类型

```
// 获取strObject的构造函数类型
var objectType = Object.prototype.toString.call(strObject);
console.log(objectType.substring(8, objectType.length - 1));
```

6 理解这段话

- 有些函数具有 this 引用，有时候这些 this 确实会指向调用位置的对象引用。但是这种用法从本质上来说并没有把一个函数变成一个“方法”，因为 this 是在运行时根据调用位置动态绑定的，所以函数和对象的关系最多也只能说是间接关系。
- 函数并不是在定义时成为方法，而是在被调用时根据调用位置的不同成为方法（不太准确但可以作为理解）
- 即使你在对象的文字形式中声明一个函数表达式，这个函数也不会“属于”这个对象——它们只是对于相同函数对象的多个引用
- 结论
  - 函数是某个对象的方法并不是在定义时决定的，而是在调用时确定的（隐式绑定的时候）

7 Object的一些常用函数

7.1 Object.prototype.toString

- 每个对象都默认有toString方法
- 如果此方法在自定义对象中未被覆盖，toString() 返回 "[object type]"，其中 type 是对象的类型，返回的类型是string类型
- 应用
- 判断对象类型

```
// 获取strObject的构造函数类型
var objectType = Object.prototype.toString.call(strObject);
console.log(objectType.substring(8, objectType.length - 1));
```

7.2 Object.keys(obj)

- ES6新增的方法

```
Object.keys(obj)
```

- 参数
  - obj: 要返回其枚举自身属性的对象
- 返回值
  - 一个表示给定对象的所有可枚举属性的字符串数组

7.3 Object.assign()

```
Object.assign(target, ...sources)
```

- 参数
  - target: 目标对象
  - soures：源对象
- 返回值
  - 目标对象
- 浅拷贝
  - bject.assign()拷贝的是（可枚举）属性值。假如源值是一个对象的引用，它仅仅会复制其引用值。

```
const newObj = Object.assign({}, oldObj);
```

8 浮点数双精度的科学计数法

- 浮点数是用二进制的科学计数法来表示，由于js采用的是双精度的浮点数，以下就以双精度浮点数及64位二进制数来说明
- 双精度浮点数的10进制的精确有效位
  - 所以只能保证前15位是精确数字，第16位只是部分精确。
    - 大于等于5：不精确，会被舍弃掉
    - 小于5：精确，保留

-
- sign(63): 符号位
- exponent(52~62): 指数位
  - 取值范围
    - 原本表示的区间为[0, 2047]
    - 计算最后结果时结果的区间[0-1023, 2047 - 1023]
      - 去掉c = 0且c为最大值: [-1022, 1023]
  - 为什么要计算最后结果时减去指数偏移量
    - 为了使得改指数形式可以表示出负数
  - 为什么减去的偏移量为1023
    - 使得最高为可以类似表示为符号位
- fraction(0~51):小数点后的尾数

    - 由于是小数部分，所以2的平方根
    - 取值范围:
        - 原本表示的区间：
            - $[0, 1-2^{-52}]$
        - 计算结果表示的范围
            - $[1, 2-2^{-52}]$
    - 为什么在计算最后的小数部分的时候结果要加1
        - 在使用二进制科学计算法表示该数时，要保证二进制形式的小数的整数部分为1，所以实际上fraction表示的只是尾数部分
- 表示正数最大值：
    - $exponent = 2046$ ， $fraction = 1-2^{-52}$
    - 用10进制表示约为
        - 2×10^{308}
- 表示正数最小值
    - $exponent = 1$ ， $fraction = 0$
    - 用10进制表示约为
        - 2×10^{-308}
- 当exponent=0和exponent=2047的特殊含义

exponent	fraction	result
=0	=0	(+/-)0
=0	!=0	denormalized number 非规格化数
=2047	=0	(+/-)无穷
=2047	=0	NAN

5.1 将浮点数转化为二进制科学计数法遇到的问题

- 一些小数的尾数部分无法使用小数的二进制进行表示，只能近似的进行表示
- 规格化成浮点数的步骤：
    - 将整数部分和小数部分分别格式化成二进制数
    - 整数部分只保留1
    - 根据计算公式求得对应的exponent和fraction

```
类型1：
10.25
第一步：用二进制形式表示
// 小数部分可以用有限的二进制数表示
1010.01
第二步：整数部分只保留1
1.01001 * 2 ^ 3;
第三步：根据公式计算
exponent = 3 + 1023 = 1026
fraction = 01001

类型2：
10.4
第一步：用二进制形式表示
// 小数部分无法使用有限的二进制数表示，则要用近似的表示方法，二进制至少要保留52位
1010.01100 01100 ...(保留52位)
第二步：整数部分只保留1
1.010 01100 01100 ...(保留52位) * 2 ^ 3
第三步：根据公式计算
exponent = 3 + 1023 = 1026
fraction = 010 01100 01100 ...(保留52位)
```

- 由于有些小数并不能用一一对应的二进制形式表示，所以会导致这些小数在表达式的计算中会出现误差，该误差的数量级为
    - $2^{-52} = 2.2 \times 10^{-16}$
    - 导致10进制小数只能保证15位有效数字，第16位就开使为部分精确

5.2 常见的解释原因问题

```
例1：
>>> 1.4 - 1.1
0.2999999999999998
为什么不是0.3？
/*
1.4和1.1两个数都不能被精确表示，用[4]的链接可以将浮点数字面量转换成真实存储的值，可以看到1.4被近似成了1.3999999999999999，1.1被近似成了
1.1000000000000008所以两数相减得到的不是0.3而是0.2999999999999998。
*/
例2：
4.0 + 1e+16 - 1e+16
>>> 4.0
没错。
5.0 + 1e+16 - 1e+16
>>> 4.0
为什么5变成了4？
/*
所以在例2中 ，恰巧第16位有效数字是部分精确的，4可以被精确表示，因为最后一位并不能表示5，所以出现了浮点数误差。
*/
4.0 + 1e+17 - 1e+17
>>> 0.0
为什么结果是0.0？ 4去哪了？
/*
4已经在第17位，超出了双精度浮点数的最大有效位数，就被忽略了，所以有[公式] 。
*/
```

9 一些js的简单表示

- Object.prototype.func ==> Object#func

10 toString函数的一些易错点

- 基本数据类型
  - null, undefined, NaN -- "null", "undefined", "NaN"
  - boolean -- true: "true", false: "false"
  - number, symbol: 为对应的字符串形式
- 符合数据类型
  - 数组
    - [null] --> ""(空数组)
    - [undefined] --> ""（空数组)
    - [NaN] --> NaN
    - 其他为对应的字符串形式
  - 其他仍然遵照ToString类型

11 区别实例对象与函数对象

- 实例对象（简称xx对象）：通过new创建一个新的实例对象
- 函数对象：将函数当作一个对象使用（函数既可以当作一个函数使用，也可以看作一个对象使用）
- 如何分析一个语句角度
  - 从语法：判断是函数还是对象
  - 从功能

```
function Fn() {

}
const fn = new Fn()
// 构造函数的确定，是由后面调用决定的，即后面有没有new
console.log(Fn.prototype)
// 从这可以将Fn理解为一个对象
// 如何看出它的对象特点
// 该函数有属性和方法时，可以看作为对象
Fn.bind({})
// 所有的函数当时Function的实例，所以对应有bind方法
$("#test")
// 此时$是一个函数
$.get("/test")
// 此时$是一个对象
```

12 看懂数据类型

```
a.b.c()
// 括号前面的必然是函数即a.b.c为函数
// 其中a,b为对象
```

13 两种类型的回调函数

- 特征
  - 自己拟定
  - 没有调用但是会执行

2.1 同步回调函数

- 程序会按照顺序来调用

```
const arr = [1, 2, 3];
arr.forEach(item => {
    console.log(item)
})
console.log(arr)
// 先输出item，在输出arr
```

2.2 异步回调函数

- 会根据代码启动异步，将回调函数放入队列之中
  - 等触发了异步后再调用

```
setTimeout(() => {
    ...
}, 0)
// 执行完后面的函数再执行该异步
```

14 js常见的内置类型之Error

```
new Error(content)
// 显示content的错误
```

14.1 Error的类型

14.1.1 ReferenceError

- 引用变量不存在

14.1.2 TypeError

- 数据类型不正确的错误

14.1.3 RangeError

- 数据值不再所允许的范围内

14.1.4 SyntaxError

- 语法错误

14.2 错误处理

14.2.1 try{执行代码}catch(error){错误处理}

- 捕获错误,即被动进行错误处理

```
try {
    let d;
    console.log(d.xxx)
}
catch(error) {
    console.log(error.message);
    // console.log(error.stack)
}
console.log("hello_world");
// 仍能正常的运行
```

14.2.2 throw error

- 抛出错误, 主动进行错误处理,并显示自己的语段

```
if(Date.now() % 2 === 0)
    alert("执行完毕")
else{
   throw new Error("不能执行")
}
```

14.3 错误对象的属性

14.3.1 message

- 错误相关的信息

14.3.2 stack

- 函数调用栈的记录信息

15 理解apply, call, bind

15.1 func.apply(thisArg, [args])

15.1.1 理解apply的作用

- 将func中this所指的属性和方法添加到thisArg的this所指的属性和方法之中，并执行func
  - 注意：此时的func所用的this就是thisArg的this即可以理解为thisArg在运行func的内容

```
function fn1() {
  this.name = "hello"
}

function fn2() {
  this.word = "world";
}
fn2Object = new fn2();
fn1.apply(fn2Object)
// 执行fn1的代码，并将fn1的this所指的属性和方法加入到fn2Object
//
console.dir(fn2Object)
console.dir(fn2)
```

15.2 bind(func1, func2)

- 返回值是一个函数wrap

2.3.1 源码的理解

```
// bind函数的理解
// 语法上：返回一个函数wrap
// 功能上：若是调用warp函数并传入参数
//          fn根据传递的参数执行并且thisArg将继承fn的属性和方法
//          之后返回值：undefine
function bind(fn, thisArg) {
  return function wrap() {
    //根据传入的参数来调用fn
    var args = new Array(arguments.length);
    console.log(args);
    for (var i = 0; i < args.length; i++) {
      args[i] = arguments[i];
    }
    //用apply来实现绑定
    return fn.apply(thisArg, args);
  };
};

function fn1(name) {
  this.name = name
}

function fn2() {
  this.word = "world";
}
fn2Object = new fn2();
console.log(bind(fn1, fn2Object)("hello"))
console.dir(fn2)
/*
undefined
fn2
 name: "hello"word: "world"
*/
```

一 javascript的基础认识

1. 解释型语言

- 编译型语言
  - C语言->编译成2进制代码（与操作系统有关）-> 执行
- 解释型语言
  - 一行一行读取，一行一行执行

2. 动态类型语言

- 动态类型的语言
  - 不能确定一个变量的类型，所以可以在代码执行的过程中动态修改
- 静态类型的语言
  - 在代码执行之前可以确定一个变量（标识符）准确的类型，并且不能修改

3. 应用场景

- 网页的交互
- 服务器开发(node.js)
- 命令行工具(node.js)
- 桌面程序，VSCode使用TypeScript开发
- App(React Native)
- 游戏开发(cocos2d-js)
- 小程序开发

4. 编写位置

4.1. 在HTML元素中属性直接执行js代码（不建议）

- 事件触发属性的双引号内

```
<a href="" onclick="alert('hello_world')>我是a元素</a>
```

- 非事件触发属性(需要指明javascript)

```
<a href="javascript: alert('hello_world')" >我是a元素</a>
```

4.2. 书写到script标签中

- 一般位于body的最下面

4.3. 从外部引入js文件

- 引入方法（仍放在body的最后面）

```
<script src="..."><script>
```

4.4. 注意事项

- 在外联js文件时，script标签中不能有js代码，且不用写type类型(text/javascript)
- 由于html文档加载顺序的原因，推荐将js代码放在body子元素的最后一行
  - html元素加载后才能够与js代码进行关联
- js代码严格区分大小写
  - 但HTML和CSS代码不区分大小写

5. js代码的注释

- 单行注释

```
//hello_world
```

- 多行注释

```
/*
hello_world
*/
```

- 文档注释

```
/*
* test 函数
*/
```

6. js的内置数据类型数据类型

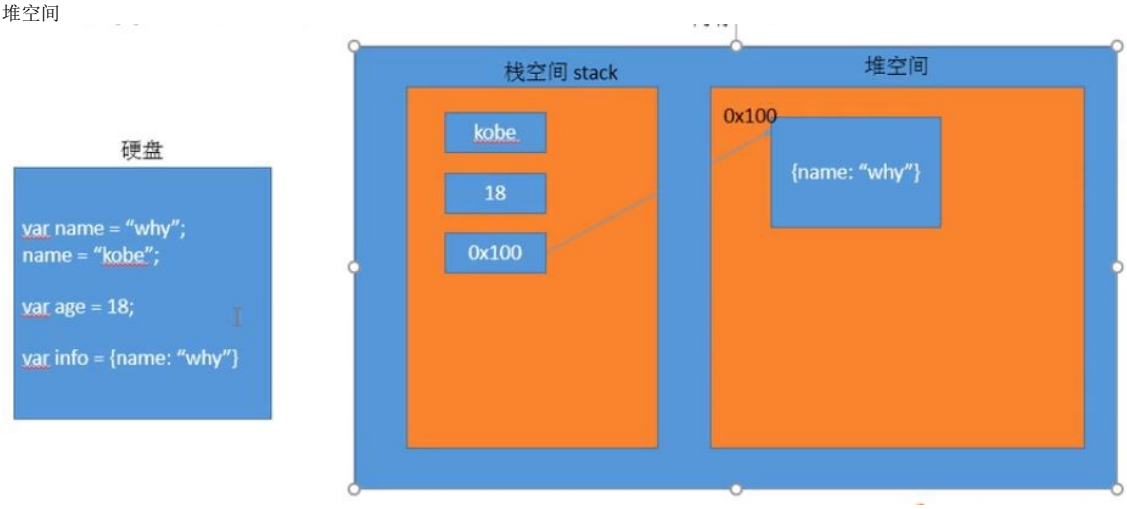
- 基本数据类型
  - number
  - string

    - boolean
    - undefined
    - null
    - symbol
- 复合数据类型
    - object

7. 变量存储的本质

7.1 内存的分类

- 栈空间
- 堆空间



-

7.2 代码运行时，各部分存储的位置

- 代码存储到硬盘中
- 代码运行时，其中定义的变量是存储在内存中
- 基本数据类型的变量所对应的值是直接存储在内存栈空间中，当变量的值改变时，是直接修改栈空间中对应位置的变量的值
- 复合数据类型的变量所对应的值是存储在内存栈空间中，但是存储的变量的值是所在堆空间的地址
    - 变量的获取实际上是通过栈空间中存储的对应堆空间的内存地址来找到相应的存储位置

二 js内置数据类型与语法解析

1 js的内置类型

- 一般认为有七种内置类型
    - null, undefined, boolean, number, string, symbol(符号类型，ES6新增的), object,

1.1 使用typeof来分辨不同的内置类型

- typeof的实质
    - 得到的不是变量的类型，而是变量所持值的类型
    - js的变量是没有类型的
- 结果：返回一个string数据的值

```
typeof undefined === "undefined"; // true
typeof true === "boolean"; // true
typeof 42 === "number"; // true
typeof "42" === "string"; // true
typeof { life: 42 } === "object"; // true
// ES6中新加入的类型
typeof Symbol() === "symbol"; // true

// 特殊
function foo() {
    // ...
}
typeof foo === "function" // true
// null比较特殊
typeof null === "object"  //true
// typeof对于只声明而没有赋值的变量返回undefined，对于没有声明的变量也是返回undefined且不报错
var a;
typeof a === "undefined"  //true
typeof b === "undefined" //true
```

- 注意
    - typeof还可以用于分辨function，虽然只是object的子类型
    - null比较特殊，由于历史遗留的原因被typeof认为是object，应该使用以下的方法来判断一个值是否是null

```
!a && typeof a === "object"
// 若a=null, 返回true
```

1.2 基本类型的理解

1.2.1 undefined

- 已在作用域中声明但没有赋值的变量，值是undefined

1.2.1.1 undefined和undeclared的区别

- undeclared是连声明都没有的变量,js错误描述是

```
ReferenceError: b is not defined
```

1.3 总结

- JavaScript 有七种内置类型：null、undefined、boolean、number、string、object 和symbol，可以使用 typeof 运算符来查看。
- 变量没有类型，但它们持有的值有类型。类型定义了值的行为特征。
- undefined 是值的一种。undeclared 则表示变量还没有被声明过。
- JavaScript 却将它们混为一谈，在我们试图访问 "undeclared" 变量时这样报错：ReferenceError: a is not defined，并且 typeof 对 undefined 和 undeclared 变量都返回"undefined"。

2 js的内置值类型

2.1 js的数组类型(Array)

- 实质上也是一种对象，只是键值为十进制数字类型而已
  - 如果一个[]类型变量的键值能转化为十进制，则其会被视为数组
- 在 JavaScript 中，数组可以容纳任何类型的值，可以是字符串、数字、对象（object），甚至是其他数组
- 对数组声明后即可向其中加入值，不需要预先设定大小
- “稀疏”数组的空白部分
  - 区别于空白部分被赋值为undefined
  - 对象执行Get操作时没有从中找到对应的属性，因此返回一个undefined，而并不是空白部分被赋值为undefined

```
var a = [];
a[1] = 2;
a[4] = 4;
console.log( a[2] ) // undefined
```

2.1.1 类数组转化为数组类型

- 类数组
  - 有length属性
  - 一组通过数字索引的值
  - 例子
    - 一些 DOM 查询操作会返回 DOM 元素列表
    - 通过 arguments 对象（类数组）将函数的参数当作列表来访问（ES6废除）

2.1.1.1 Array#slice.call(arrayLike)

- 将类数组强制转化为数组类型，并将数组作为返回值

2.1.1.2 [].slice.call(arrayLike)

- 将类数组强制转化为数组类型,并将数组作为返回值

```
function foo() {
    var arr = Array.prototype.slice.call(arguments);
    console.log(arr);
    console.log(arr instanceof Array)
}
foo("a", "b", "c");
// ["a", "b", "c"]
// true
```

2.1.1.3 Array.from(arrayLike[,func]);

- 将类数组强制转化为数组类型,并将数组作为返回值
- ES6新增的方法

```
function foo() {
    var arr = Array.from(arguments);
    console.log(arr);
    console.log(arr instanceof Array)
}
foo("a", "b", "c");
// ["a", "b", "c"]
// true
```

2.2 字符串类型(string)

- string类型的数值在调用String的方法时，会先自动被初始化String对象，再调用对应的函数
- string类型使用String对象进行包装

2.2.1 区分字符串和字符数组

- 都具有length属性, indexOf和concat方法

```
var a = "hello";
var b = ["h", "e", "l", "l", "o"];

console.log(a.length);   // 5
console.log(b.length);   // 5

console.log(a.indexOf("o")); // 4
console.log(b.indexOf("o")); //4

a = a.concat(" world");
b = b.concat([" ", "w", "o", "r", "l", "d"]);

console.log(a)   // "hello world"
console.log(b)
```

- 但是区别
  - 以前访问对应的字符应该使用charAt(index)来进行访问
  - js中字符串中的值是不可变的
    - 字符串不可变是无法对字符串内部的值进行修改
      - 需要通过字符串的成员函数对字符串进行修改，但是实质上为创建并返回一个新的字符��，而不是在原字符串上进行修改。
  - 字符数组中的值是可变的

```
var a = "hello"
var b = ["h", "e", "l", "l", "o"];

a[0] = "H";
b[0] = "H"

console.log(a)  //"hello"
console.log(b)  //["H", "e", "l", "l", "o"]
```

2.2.2 字符串借用数组类型的函数来实现功能

- 各个字母中间插进其他字符

2.2.2.1 Array.prototype.join.call(string, 中间插入的值)

- 返回一个中间加入值的新的字符串

2.2.2.2 Array.prototype.map.call(string, func).join("")

- 返回一个中间加入值的新的字符串

```
var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
return v.toUpperCase() + ".";
} ).join( "" );
c; // "f-o-o"
d; // "F.O.O."
```

- 实现字符串的reserve
  - reverve函数是将本身颠倒，而不仅仅是返回一个颠倒的字符串
  - 字符串中没有reserve函数
  - 数组的reserver函数是将本身颠倒，当使用call让字符串调用时，由于字符串本身的值时不变的，所以不能这种方法
- 可以使用以下步骤
  - 将字符串转化称字符数组
  - 将字符数组颠倒
  - 再转化为字符串

```
var a = a.split("")
         .reserve()
         .join("")
// 但对简单的字符串却完全适用
```

2.3 数字类型(number)

- js只有一个数字类型number
  - js没有真正意义上的整数，实际上是没有小数点的小数
  - 使用双精度的格式来表示number
    - 以10进制的形式
      - 前15位为精确位
      - 第16位为部分精确位
- number类型的数值在调用Number的方法时，会先自动被初始化Number对象，再调用对应的函数
- 常见的数字表示法
  - 科学计数法: 1.23e-10
  - 二进制: 0b
  - 八进制: 0o
  - 十六进制：0x

2.3.1 数字类型的较小值问题

```
console.log( 0.1 + 0.2 === 0.3) // false
// 原因：0.1和0.2都无法规范为有限的浮点数形式
// 前15位为精确的，第16位部分精确
```

- 使表达式与实际数值相等的判断方法

2.3.1.1 二进制的误差

- $2^{-52} = 2.2 \times 10^{-16}$

```
var a = 0.1 + 0.2;
var b = 0.3;
console.log(Math.abs(b - a) < Math.pow(2, -52))
```

2.3.1.2 Number.EPSILON

- ES6将上述误差值保存在Number.EPSILON
- 应用：判断是技术和表达式之间的大小

```
var a = 0.1 + 0.2;
var b = 0.3;
console.log(Math.abs(b - a) < Number.EPSLION)
```

2.3.2 Number常用的属性和方法

2.3.2.1 Number.EPSILON

- 表达式和实际数之间的最大误差

2.3.2.2 Number.MAX_SAFE_INTEGER

- 被安全呈现的最大整数(正)

2.3.2.3 Number.MIN_SAFE_INTEGER

- 被安全呈现的最小整数（负）
- 与上面一起构成了整数的安全范围

2.3.2.4 Number#toExponential()

- 返回对应数值的科学计数法的表示

2.3.2.5 Number#toFixed(num)

- 返回保留num位小数的值

2.3.2.6 Number#toPrecision(num)

- 返回保留num位有效数字的值

2.3.2.6 Number.isInteger(num)

- 判断某个数是不是整数

2.3.2.7 Number.isSafeInteger(num)

- 判断某个数是不是在安全的整数

2.3.3 实现32为有符号整数

```
number | 0
```

- 因为位运算符只适用于32位数

2.3.4 特殊数字(number)

2.3.4.1 不是数字的数字(NaN)

- NaN是指“不是一个数字”，但是仍是一种number类型
- 实质上是一个警戒值，用于指出数字类型中的错误
  - 无效数值
  - 失败数值
  - 坏数值

2.3.4.1.1 NaN与==、===

- 数值NaN是唯一一个非自反
  - x === x不成立

```
console.log(NaN == NaN) //false
console.log(NaN === NaN) //false
```

2.3.4.1.2 判断一个值是不是NaN的方法 -- Number.isNaN

- 运用isNaN，舍弃
- 只适用于ES6

```
var a = NaN;
console.log(Number.isNaN(a))
```

- 内部实现的步骤（利用自反的性质）

```
if(!Number.isNaN) {
    Number.isNaN = function(n) {
        return n !== n
    }
}
```

2.3.4.2 无穷数

- Infinity -Infinity
- js中有固定的属性进行定义
  - Number.INFINITY
  - Number.NAGATIVE_INFINITY

2.3.4.2.1 js中除以0为无穷值

```
console.log(1 / 0) // Infinity
console.log(-1 / 0) // -Infinity
```

- 可以用于分辨(+/-)0

2.3.4.2.2 js中当所表示的值超过Number.MAX_VALUE时，为无穷值

2.3.4.2.3 js中无穷除以无穷为NaN

2.3.4.3 零值(+/-)

- 包含+0和-0

2.3.4.3.1 -0出现的场合

- 自己赋值
- 在一些运算中 0 * 负数 | 0 / 负数
  - 加减法不会得到-0

2.3.4.3.2 常见函数对-0的处理

- -0转化为字符串(不保留符号)

```
var a = -0;
a.toString(); // "0"
a + ""; // "0"
String( a ); // "0"

JSON.stringify( a ); // "0"
```

- 字符串-0转化为数值

```
+"-0"; // -0
Number( "-0" ); // -0
JSON.parse( "-0" ); // -0
```

2.3.4.3.3 -0与0之间的大小关系

- ==、===结果都返回true

2.3.4.3.4 如何判断数值是-0

- 法一：构建函数
  - 利用-0与-Inifity的关系

```
function isNegZero(n) {
    n = Number(n);
    return n === 0 && 1 / n === -Inifity
}
```

2.3.4.3 特殊等式Object.is(v1, v2)

- 经常用于判断-0以及NaN，其他一般用==或者===
- 源码

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 判断是否是-0
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
            }
        // 判断是否是NaN
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其他情况
        return v1 === v2;
    };
}
```

2.4 不是值的值(undefined,null)

2.4.1 undefined和null的异同

- 相同
  - undefined和null：类型只有一个值，即是undefined/null,既是类型也是值
- 不同
  - null表示空值(empty value)，undefined表示没有值（missing value）
    - 空值表示赋过值，但此时值为空
    - 没有值表示没有赋过值
  - null是一个特殊的关键字，不能进行赋值,undefined可以作为标识符，当作变量来使用和赋值

2.4.2 作为标识符和变量的undefined

- 在非严格模式下可以把undefined作为标识符进行赋值，而严格模式下会报TypeError的错误
- 在非严格和严格模式下可以声明一个局部变量undefined,用它进行计算
- 虽然undefined可以作为标识符或者变量，但是永远不要重新定义undefined

```
function foo() {
    undefined = 2;
    console.log(undefined)
}
foo();
// 2
function foo() {
    "use strict";
    undefined = 2; // TypeError!
}
foo();
function foo() {
    "use strict";
    var undefined = 2;
    console.log( undefined ); // 2
}
foo();
```

- 与undefined相关的运算符--void
- void运算符可以得到undefined标识符的返回值
  - void + 表达式
    - 返回值为内置undefined的值即undefined
      - 作用：使得表达式的返回值为undefined

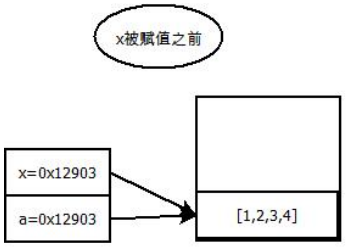
```
var a = 42;
console.log( void a, a ); // undefined 42
```

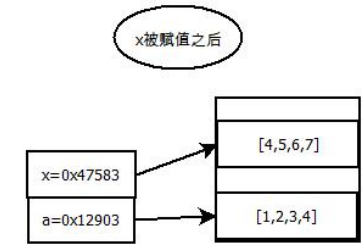
3 数据类型和赋值，引用

- 变量的赋值和参数的传递两种形式
  - 值赋值
    - 重新申请一块存储区域，并将值变为和赋的值相同
  - 引用复制
    - 只将存储值的地址赋给变量
- 规则
  - 简单值（即标量基本类型值，scalar primitive）总是通过值复制的方式来赋值 / 传递，包括null、undefined、字符串、数字、布尔和 ES6 中的 symbol。
  - 复合值（compound value）——对象（包括数组和封装对象）和函数，则总是通过引用复制的方式来赋值 / 传递。

3.1 函数的参数的值传递方式

- 函数参数的传递方式仍然符合上面的规则，但是有要注意的地方
- 对引用复制的参数执行赋值操作
  - 函数内的参数由于赋值之后指向其他存储地址，但是原来的参数指向的地址并没有改变





  - 若想要让函数参数指向的是相应的地址，在函数操作的时候不能对传入的参数执行赋值操作

```
function foo(x) {
    x.push( 4 );
    x; // [1,2,3,4]
    // 然后
    x = [4,5,6];
    x.push( 7 );
    x; // [4,5,6,7]
}
var a = [1,2,3];
foo( a );
a; // 是[1,2,3,4]，不是[4,5,6,7]
```

- 数组可以采用arr.slice()使得传递的是另一个地址，从而不改变原地址的数组
  - 即可以通过传递副本的方法使得不改变原本的值

3.2 函数的参数的传递封装对象（对基本类型进行封装）

- 进行表达式的运算时，结果会返回对应的封装对象而不是基本类型
- 同样不能进行赋值操作，否则会指向新的内存地址

```
function foo(x) {
    x = x + 1;
    x; // 3
}
var a = 2;
var b = new Number( a ); // Object(a)也一样
foo( b );
console.log( b ); // 是2，不是3
```

3.3 总结

- JavaScript 中的数组是通过数字索引的一组任意类型的值。字符串和数组类似，但是它们的行为特征不同，在将字符作为数组来处理时需要特别小心。
- JavaScript 中的数字包括“整数”和“浮点型”。
- null 类型只有一个值 null，undefined 类型也只有一个值 undefined。所有变量在赋值之前默认值都是 undefined。void 运算符返回 undefined。
- 数字类型有几个特殊值，包括 NaN（意指“not a number”，更确切地说是“invalid number”）、+Infinity、-Infinity 和 -0。
- 简单标量基本类型值（字符串和数字等）通过值复制来赋值 / 传递，而复合值（对象等）通过引用复制来赋值 / 传递。
- JavaScript 中的引用和其他语言中的引用 / 指针不同，它们不能指向别的变量 / 引用，只能指向值。

4 原生函数

- 常见的原生函数 --- 用于构造封装对象
  - String()
  - Number()
  - Boolean()
  - Array()
  - Object()
  - Function()
  - RegExp()
  - Date()
  - Error()
  - Symbol()——ES6 中新加入的！

4.1 基本类型的封装对象

- 我们可以先对基本类型封装成对象以来调用对应的方法，但是js并不这样建议，直接使用封装对象来“提前优化”代码反而会降低执行效率。因为返回值都是基本类型，你这样又要进行封装
- 基本类型在调用相应的函数时，
  - js会自动对基本数据类型进行封装后再调用相应的函数

4.2 拆封基本类型的封装对象

- obj.valueOf()z
  - 返回相应的基本类型
- 隐式拆封
  - 在表达式中，有时候需要进行隐式拆分

4.3 原生函数作为构造函数构建对象进行封装

- 一般情况下，我们采用的是字面量对数据类型进行定义（包括复杂基本类型）
- 利用new 原生函数构建的是一个对象而不是对应的值

4.3.1 Array作为构造函数

- 不要只用一个数字最为Array的构造参数
- Array可以带不带new都可以

```
var a = new Array( 1, 2, 3 );
// 参数列表， arr
a; // [1, 2, 3]
var b = [1, 2, 3];
b; // [1, 2, 3]
```

4.3.1.1 为什么不能只用一个数字作为Array的构造参数

- 该参数会作为该Array的预设长度
  - 创建出的数组是空数组，但是有长度
- 如若一个数组没有任何单元，但它的 length 属性中却显示有单元数量，这样奇特的数据结构会导致一些怪异的行为, 可以用以下的方法来解决这种行为

```
var a = Array.apply( null, { length: 3 } );
// 通过该方法来构建长度为3的参数
// 相当于Array(undefined, undefined, undefined)
```

4.3.2 RegExp作为构造函数的情况

- 强烈建议使用常量形式（如 /^a*b+/g）来定义正则表达式
- 但是定义动态正则表达式时
  - RegExp("pattern","flags")
    - pattern可以是string，reg类型
    - flags
      - g: 全局匹配
      - i：忽略大小写

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );
var matches = someText.match( namePattern );
```

4.3.3 Date和Error作为构造函数

- 由于没有对应的常量形式作为体代，所以经常作为构造函数。
- 创建日期对象必须使用 new Date()。Date(..) 可以带参数，用来指定日期和时间.

4.3.3.1 Date用于根据时间创建随机数

- Date获取从1970年到现在的毫秒数
  - Date.now()
  - new Date() .getTime()

```
console.log(Date.now)

if(!Date.now) {
    Date.now = function() {
        return new Date().getTime();
    }
}
```

4.3.3.2 Error作为构造函数

- 带不带new都可以
- 创建错误对象（error object）主要是为了获得当前运行栈的上下文
  - 栈上下文信息包括函数调用栈信息和产生错误的代码行号，以便于调试（debug）。

4.3.4 Symbol作为构造函数

- 可以自定义Symbol
  - 在自定义Symbol属性时，千万不要用new符号
- 可以使用一些已经预定好的symbol
  - Symbol.create
  - Symbol.iterator
- ES6 中新加入了一个基本数据类型 ——符号（Symbol）。
- 作用
  - 符号是具有唯一性的特殊值（并非绝对），用它来命名对象属性不容易导致重名。

4.3.4.1 Symbol类型作为属性名

- 符号可以用作属性名，但无论是在代码还是开发控制台中都无法查看和访问它的值，只会显示为诸如 Symbol(Symbol.create) 这样的值。它却主要用于私有或特殊属性。属于无法枚举的属性
  - 无法通过Object.keys()获得对应的属性
- 自定义
  - 虽然然符号实际上并非私有属性（通过 Object.getOwnPropertySymbols(..) 便可以公开获得对象中的所有符号）

```
var mysym = Symbol( "my own symbol" );
mysym; // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym; // "symbol"
```

- 使用已经定义好的Symbol

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

4.3.4.1.1 访问对象的Symbol属性

- 无法使用Object.keys()得到Symbol属性值，但可以用

    - 通过 Object.getOwnPropertySymbols(..) 便可以公开获得对象中的所有符号
- 访问属性值只能用[]的形式

```
var mysym = Symbol( "my own symbol" );
mysym; // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym; // "symbol"

var a = { };
a[mysym] = "foobar";
Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

4.4 Object#toString.call(obj)

- 所有 typeof 返回值为 "object" 的对象（如数组）都包含一个内部属性 [[Class]]（我们可以把它看作一个内部的分类，而非传统的面向对象意义上的类）。
- 返回值为字符串[object, 内部类]
- obj为基本类型
    - null和undefined
        - 虽然没有对应的内部类，但仍然会返回[object, null] [object, undefined]
    - 其他基本类型
        - 得到一个对象的内部属性[object [Class]]
            - 即会先将用new进行自动封装
- obj为符合类型
    - 得到一个对象的内部属性[object [Class]]
    - Class: 构造函数的名称

```
// 1. undefined和null
Object.prototype.toString.call( null );
// "[object Null]"
Object.prototype.toString.call( undefined );
// "[object Undefined]"
// 2. 基本类型
Object.prototype.toString.call( "abc" );
// "[object String]"
Object.prototype.toString.call( 42 );
// "[object Number]"
Object.prototype.toString.call( true );
// "[object Boolean]"

// 3. 复合类型
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"
Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

4.5 原生函数的原型

- 原生构造函数有自己的 .prototype 对象，如 Array.prototype、String.prototype
    - 字符串值封装为字符串对象之后，就能访问 String.prototype 中定义的方法
- 所有的函数都可以调用 Function.prototype 中的 apply(..)、call(..) 和 bind(..),来时实现对具体的对象调用对应的方法
- 原生函数的原型对象只能由其实例来进行调用

```
String#indexOf(..)
在字符串中找到指定子字符串的位置。
String#charAt(..)
获得字符串指定位置上的字符。
String#substr(..)、String#substring(..) 和 String#slice(..)
获得字符串的指定部分。
String#toUpperCase() 和 String#toLowerCase()
将字符串转换为大写或小写。
String#trim()
去掉字符串前后的空格，返回新的字符串。

// 以上方法并不改变原字符串的值，而是返回一个新字符串
```

4.5.1 有些原型对象并不一定是对象

- 对应类型的空值
    - Function.prototype
        - 函数类型，且为空函数
    - Array.prototype
        - Array类型，且为空数组
    - RegExp.prototype
        - RegExp类型，且为空正则表达式
- 可以用于作为默认值
    - 从 ES6 开始，我们不再需要使用 vals = vals || .. 这样的方式来设置默认 值（参见第 4 章），因为默认值可以通过函数声明中的内置语法来设置

```
var vals = Array.prototype;
var fn = Function.prototype;
var rx = RegExp.prototype;
```

4.6 总结

- JavaScript 为基本数据类型值提供了封装对象，称为原生函数（如 String、Number、Boolean等）。它们为基本数据类型值提供了该子类型所特有的方法和属性（如：String#trim() 和Array#concat(..)）
- 对于简单标量基本类型值，比如 "abc"，如果要访问它的 length 属性或 String.prototype 方法，JavaScript 引擎会自动对该值进行封装（即用相应类型的封装对象来包装它）来实现对这些属性和方法的访问。

5 类型转换

- JavaScript 中的强制类型转换总是返回标量基本类型值

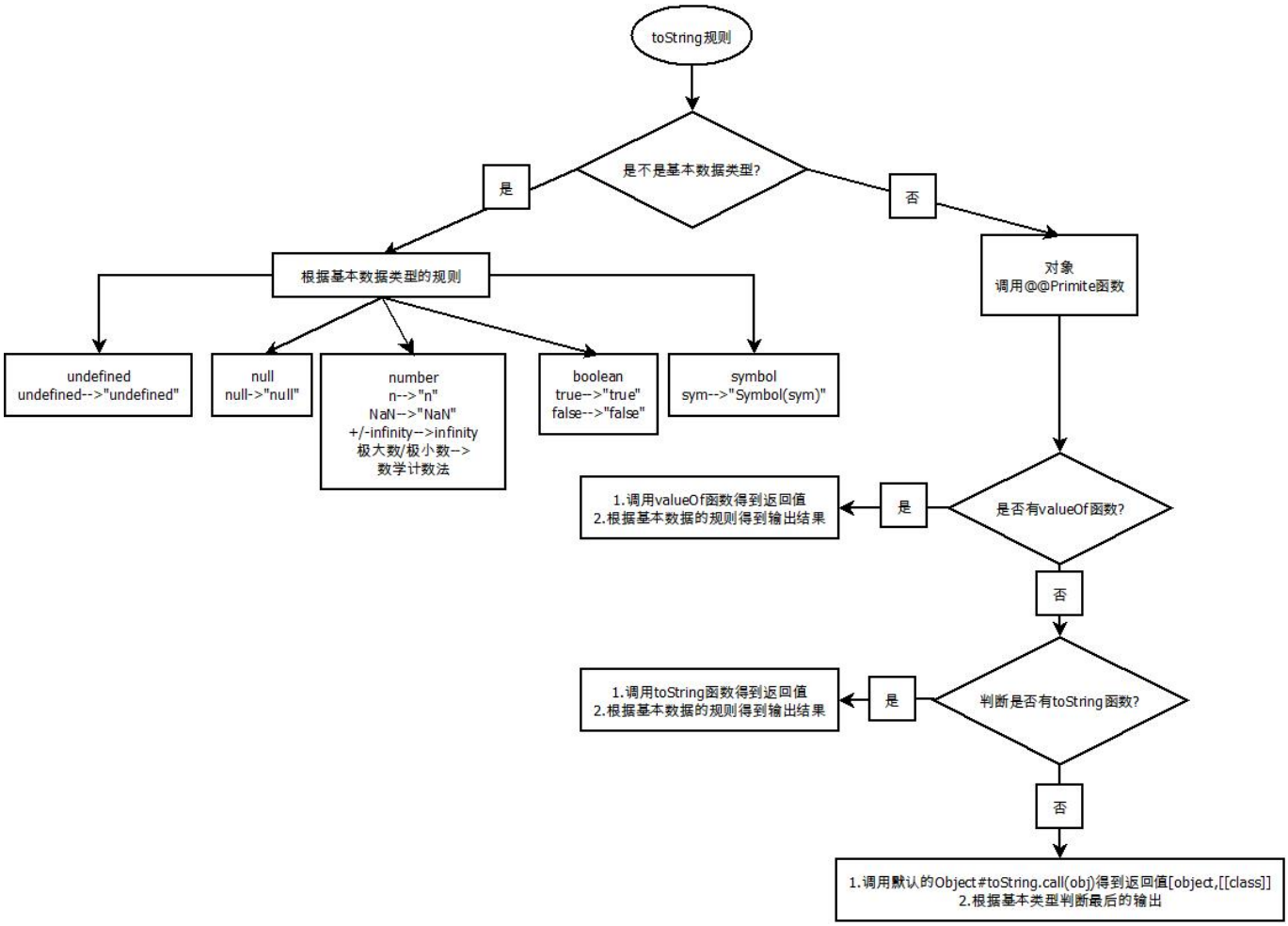
5.1 类型转换和强制类型转换

- 类型转换
  - 显示类型转换
  - 发生在静态类型语言的编译阶段
- 强制类型转换
  - 隐式类型转换
  - 动态类型语言的运行时

5.2 数值，字符串，布尔值之间的类型转化规则

5.2.1 ToString规则

- 非字符串到字符串的类型转换
-



- 规则：
  - 基本数据类型：
    - null, undefined
      - null --> "null"，undefined --> "undefined"
    - boolean
      - true --> "true", false --> "false"
    - number
      - n --> "n"
      - NaN --> "NaN"
      - 极大数和极小数-->转化为科学计数法
    - symbol

        - sym --> "Symbol(sym)"
    - 对象
        - 调用抽象函数ToPrimitive()，是否有valueOf()
            - 有valueOf（封装对象）：放回对应的valueOf的值，然后按照基本数据类型的规则进行转化
            - 没有valueOf（非封装对象），是否有toString函数
                - 返回toString的值
                - 若没有toString函数，则默认调用有Object.prototype.toString.call(obj), 返回"[object, [[Class]]]"

```
var a = {};
console.log(String(a));
    // [object Object]
var b = {
    c: 2,
    d: 3,
    toString() {
        var str = "";
        Object.keys(this).forEach(key => {
            if(typeof this[key] !== "function")
                str = str + key + ": " + this[key] + " ";
        })
        return str;
    }
}
console.log(String(b));
    // c: 2 d: 3
```

- 注意：
    - [null] --> ""（空字符串）
    - [undefined] --> ""（空字符串）

5.2.1.1 JSON.stringify(obj)

- 不属于强制转换类型，但是结果是返回一个string类型
- (1) string，number，null，boolean的 JSON.stringify(..) 规则与 ToString 基本相同。
- (2) 若要转化的是对象时
    - 判断是否有toJSON函数，调用toJSON函数得到结果

```
JSON.stringify( 42 ); // "42"
JSON.stringify( "42" ); // ""42"" （含有双引号的字符串）
JSON.stringify( null ); // "null"
JSON.stringify( true ); // "true"
```

- 不安全的JSON值
    - undefined、function、symbol（ES6+）和包含循环引用（对象之间相互引用，形成一个无限循环）的对象都不符合 JSON结构标准
        - 在对象中遇到 undefined、function 和 symbol 时会自动将其忽略
        - 在数组中遇到 undefined、function 和 symbol 时则会返回 null

```
JSON.stringify( undefined ); // undefined
JSON.stringify( function(){} ); // undefined
JSON.stringify(
    [1,undefined,function(){},4]
); // "[1,null,null,4]"
JSON.stringify(
    { a:2, b:function(){} }
); // "{"a":2}"
```

5.2.1.1.1 JSON.stringify(obj)针对于对象

- 自定义JSON的输出
    - 条件
        - 如果要对含有非法 JSON 值的对象做字符串化，或者对象中的某些值无法被序列化时，就需要定义 toJSON() 方法来返回一个安全的 JSON 值(不会报错)
        - 某些属性不想被JSON化
- 定义obj.toJSON()函数
    - 返回一个能够被字符串化的安全的 JSON 值，再由 JSON.stringify(..) 对其进行字符串化
        - 返回的可以是安全的JSON值，即任何类型

```
var a = {
    b: 5,
    c: 4
}
// 只JSON化对象和对应的b属性
a.toJSON = function() {
    return {
        b: this.b
    }
}
console.log(JSON.stringify(a))
//
```

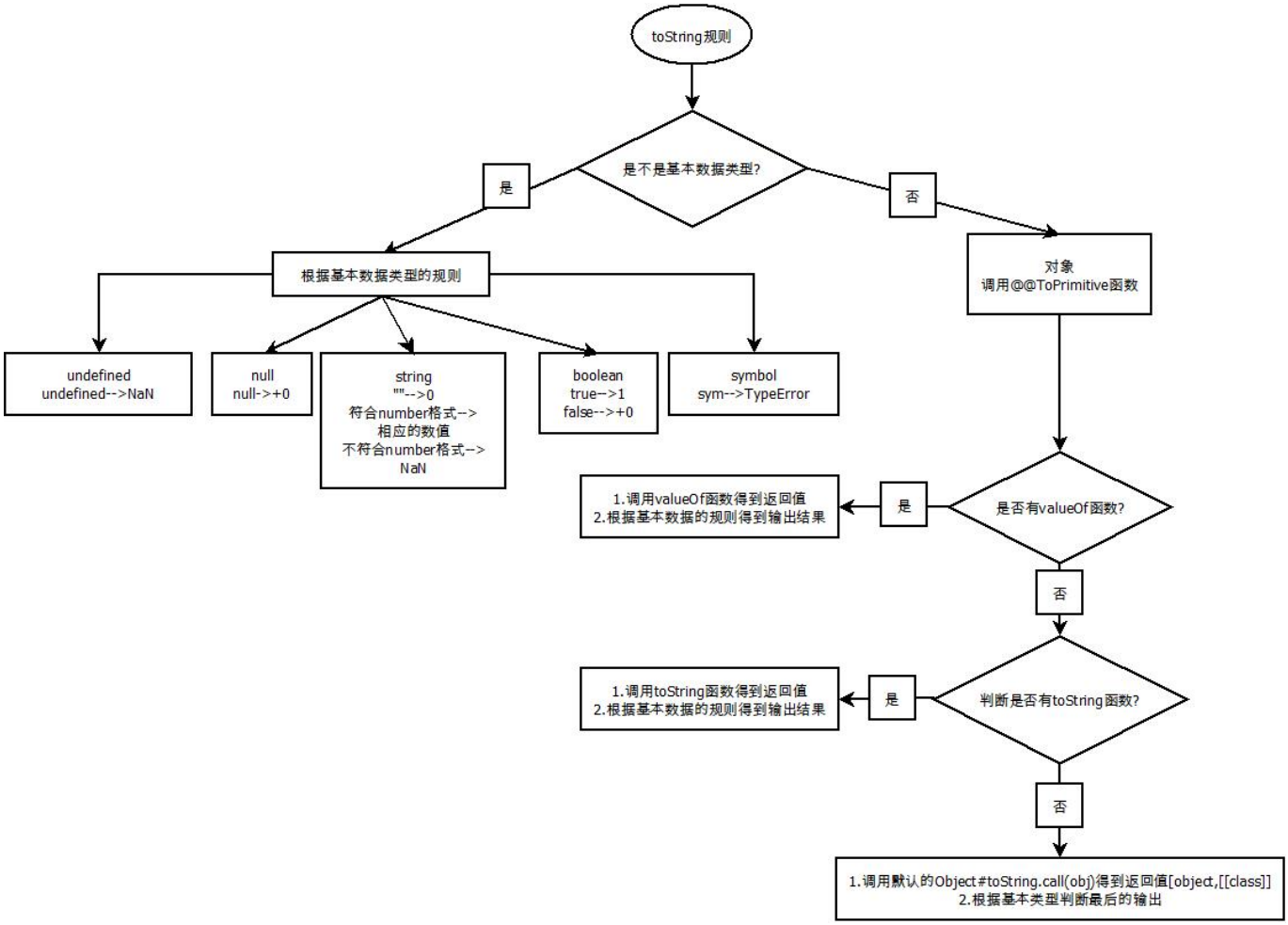
- 使用JSON.stringify(obj[, replacer[, space]])
  - 参数replacer
    - 如果 replacer 是一个数组，那么它必须是一个字符串数组，其中包含序列化要处理的对象的属性名称，除此之外其他的属性则被忽略。
    - 如果 replacer 是一个函数，它会对对象本身调用一次，然后对对象中的每个属性各调用一次，每次传递两个参数，键和值
  - 参数space(string/number)
    - 键值前面个的空格的值

```
var a = {
    b: 42,
    c: "42",
    d: [1,2,3]
};
JSON.stringify( a, ["b","c"] ); // "{"b":42,"c":"42"}"
JSON.stringify( a, function(k,v){
    if (k !== "c") return v;
} );
// "{"b":42,"d":[1,2,3]}"
```

```
var a = {
    b: 42,
    c: "42",
    d: [1,2,3]
};
JSON.stringify( a, null, "-----" );
// "{
// -----"b": 42,
// -----"c": "42",
// -----"d": [
// ----------1,
// ----------2,
// ----------3
// -----]
// }"
```

5.2.2 ToNumber规则

- 非数字类型到数字类型的转化
-

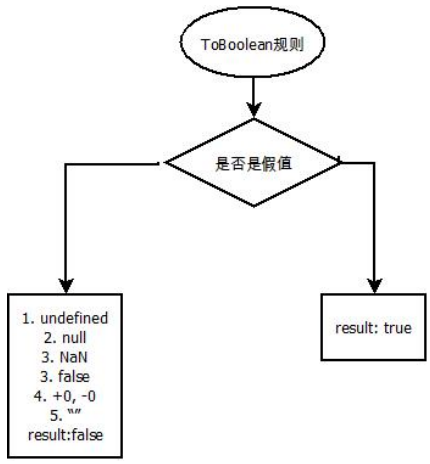


;

- 规则
  - 基本数据类型
    - true 转换为 1，false 转换为 0。
    - undefined 转换为 NaN，null 转换为 0
    - string
      - 符合Number语法规则的string，转化为对应的数字
      - 不符合语法规则返回NaN
      - 空字符串，返回0
  - 对象
    - 调用抽象操作 ToPrimitive, 检查是否有valueOf()
      - 封装对象(有valueOf函数)
        - 被转换为相应的基本类型值，如果返回的是非数字的基本类型值，遵循基本数据类型转化
      - 非封装对象（无valueOf函数），检查是否有toString()函数
        - 如果有，返回toString函数的值，并且调用string的转化规则
        - 如果没有，则返回TypeError

```
var a = [1, 3, 4]
console.log(Number(a)) // NaN
/**
*   无valueOf函数
*   有toString函数："1,3,4"
*   string不符合Number语法：返回NaN
*/
var b = {a: 1, b: 2}   // NaN
/**
*   无valueOf函数
*   有toString函数:Object#toString.call(a) [object, object]
*   string不符合Number语法：返回NaN
*/
var c = "134"
console.log(console.log(c)) // 134

var d = []
/**
*   无valueOf函数
*   有toString函数：""
*   string为空，返回0
*/
```

5.2.3 ToBoolean规则




```
var a = new Boolean(false);
console.log(Boolean(a));
// true
var b = [];
console.log(Boolean(b))
// true
```

4.2.3.1 假值

- 假值对象（对以下的情况进行封装的对象）
- undefined
- null
- false
- +0、-0 和 NaN
- ""

5.3 显式强制类型转换

5.3.1 其他类型转换为字符串

4.3.1.1 String(obj)

- 运用了ToString规则进行转换为string

4.3.1.2 obj.toString()

- 若obj本身有toString函数，则可以用这种形式，返回相应的string
- 若没有，则采用Object#toString.call(obj)，结果返回[object, [[Class]]]

5.3.2 其他类型转换为数值

5.3.2.1 Number(obj)

- 运用了ToNumber规则进行转换为number

5.3.2.2 obj.toNumber()

- 若obj本身有toString函数，则可以用这种形式，返回相应的string

5.3.2.3 parseInt(obj, radix), parseFloat(obj)

- 注意ES5以前要注意写radix
- 和Number以及obj.toNumber要区别开来
- 其他过程和toNumber规则类似
- 区别在于对于string基本类型的处理
  - 解析允许字符串中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停止。而转换不允许出现非数字字符，否则会失败并返回 NaN

```
var a = [1,4,5]
console.log(Number(a));
// NaN
console.log(parseInt(a));
// 1
```

- 特殊例子

```
parseInt( 0.000008 ); // 0 ("0" 来自于 "0.000008")
parseInt( 0.0000008 ); // 8 ("8" 来自于 "8e-7")
parseInt( false, 16 ); // 250 ("fa" 来自于 "false")
parseInt( parseInt, 16 ); // 15 ("f" 来自于 "function..")
parseInt( "0x10" ); // 16
parseInt( "103", 2 ); // 2
```

5.3.3 其他类型转换为布尔类型

5.3.3.1 Boolean()

- 运用ToBoolean规则(不常用)

5.3.3.2 !!

- 运用ToBoolean规则，将其他类型转化为boolean，实际上已经是隐式转化了

5.3.4 特殊符号的类型转化

5.3.4.1 +

- 若+的左右运算数中都是number类型，则正常进行加法运算
- 若+的左右运算数中存在string类型，则非string类型会进行隐式转化，再进行拼接
- 若+的左右运算数中除了上述的两种情况，则两个运算数都会隐式转化为string类型，再进行拼接。

5.3.4.1.1 运用+号进行字符串的转化

```
console.log(typeof 123 + '')
// string
```

5.3.4.2 ~ + 整数

- ~符号的隐含操作（补码）
  - 转换为32为有符号的整数
  - -(x + 1)
- 应用，在查找是否有对应的index过程中, 可以将这样形式的判断转化为真假值的判断
  - 如果有，则返回相应的index
  - 没有，则返回-1

```
function foo(a, value) {
    if(~a.indexOf(value)) {
        console.log(value + " in a");
    }
    else {
        console.log("Not in");
    }
}
var a = [1,3,4]

foo(a, 3);
foo(a, 5);
// 3 in a
// Not in
```

5.4 隐式强制类型转换

5.4.1 常见的隐式转化

5.4.1.1 在有运算符+

- 隐式和显示是相对的，不是绝对的
- +的操作符有一个是字符串类型
  - 其他类型会先运用toString规则隐式转化为对应的string类型，再进行拼接

```
var a = []
console.log([] + "")
// ""
var b = {};
console.log(b + "")
// [object, Object]
```

5.4.1.2 在有运算符-

- -的操作符有一个是Number类型
  - 其他类型会先运用toNumber规则隐式转化为对应的number，再进行计算

```
console.log(100 - "95");
// 5
console.log(100 - 'abc');
// NaN
```

5.4.1.3 隐式转化为Boolean类型

- (1) if (..) 语句中的条件判断表达式。
- (2) for (.. ; .. ; ..) 语句中的条件判断表达式（第二个）。
- (3) while (..) 和 do..while(..) 循环中的条件判断表达式。
- (4) ? : 中的条件判断表达式。
- (5) 逻辑运算符 ||（逻辑或）和 &&（逻辑与）左边的操作数（作为条件判断表达式）。
- (6) ||
  - 注意不是返回true或者false，而是true变量的返回值
  - 短路操作
  - 返回值是第一个true的对应的值
    - 即先转化为boolean类型判断真假，再返回相应的值
- &&
  - 短路操作
  - 常用于判断是否存在某个属性，从而能执行相应的函数

5.4.1.4 符号类型的隐式转化

- 不符合规范

```
var a = Symbol("hello")
console.log(String(a))
// "Symbol(hello)"
console.log(a + "")
// TypeError
```

5.5 宽松相等和严格相等与隐式转化

- 区别
  - == 允许在相等比较中进行强制类型转换，而 === 不允许

5.5.1 ==隐式类型转化的一般规则

- 方法（判断两侧的数据类型）
  - 第一步：左右两侧都是对象类型，则判断对象 === 对象
  - 第二步（以下方法存在优先级）
    - 1. 存在对象类型，调用toString(obj)
    - 2. 存在字符串类型，调用toNumebr(str)
    - 3. 存在布尔类型，调用toNumber(boolean)
  - 第三步
    - 若左右两侧的基本类型相同，则判断左 === 右
    - 若存在undefine，null，NaN则根据特殊规则进行判断
  - 第四步：循环步骤第二步，第三步知道左右两端的类型相同为止
  - 采用查表的方式

	被比较值 B						
		Undefined	Null	Number	String	Boolean	Object
被比较值 A	Undefined	`true`	`true`	`false`	`false`	`false`	`IsFalsy(B)`
	Null	`true`	`true`	`false`	`false`	`false`	`IsFalsy(B)`
	Number	`false`	`false`	`A === B`	`A === ToNumber(B)`	`A=== ToNumber(B)`	`A== ToPrimitive(B)`
	String	`false`	`false`	`ToNumber(A) === B`	`A === B`	`ToNumber(A) === ToNumber(B)`	`ToPrimitive(B) == A`
	Boolean	`false`	`false`	`ToNumber(A) === B`	`ToNumber(A) === ToNumber(B)`	`A === B`	ToNumber(A) == ToPrimitive(B)
	Object	`false`	`false`	`ToPrimitive(A) == B`	`ToPrimitive(A) == B`	ToPrimitive(A) == ToNumber(B)	`A === B`

在上面的表格中，`ToNumber(A)` 尝试在比较前将参数 A 转换为数字，这与 +A（单目运算符+）的效果相同。`ToPrimitive(A)`通过尝试调用 A 的`A.toString()` 和 `A.valueOf()` 方法，将参数 A 转换为原始值（Primitive）。

```
2 == [2]; // true
/**
* 数值 == 对象
* 2 == ToString([2]) --> "2"
* 数值 == 字符串
* 2 == ToNumber("2") --> 2
*    true
*/
"" == [null]; // true
/**
* 字符串 == 对象
* "" == ToString([null]) --> ""
*    true
*/

42 == "43"; // false
/**
* 数值 == 字符串
* 42 == ToNumber("43") --> 43s
*    false
*/
42 == "foo"; // false
/**
* 数值 == 字符串
* 42 == ToNumber("foo") --> NaN
*    false
*/
"true" == true; // false
/**
*  字符串 == 布尔值
* ToNumber("true") --> NaN == ToNumber(true) --> 1
*    false
*/
"foo" == [ "foo" ]; // true
/**
*  字符串 == 对象
* "foo" == ToString(["foo"]) --> "foo"
*    true
*/
```

4.5.1.1 字符串和数字的宽松相等

(1) 如果 Type(x) 是数字，Type(y) 是字符串，则返回 x == ToNumber(y) 的结果。 (2) 如果 Type(x) 是字符串，Type(y) 是数字，则返回 ToNumber(x) == y 的结果。

- 利用ToNumber规则将字符串转化为数字进行判断相等

4.5.1.2 其他类型和布尔类型

(1) 如果 Type(x) 是布尔类型，则返回 ToNumber(x) == y 的结果； (2) 如果 Type(y) 是布尔类型，则返回 x == ToNumber(y) 的结果。

- 利用ToNumber将布尔类型转化为数字进行判断（可能还需要再接着转化）
- 千万不要出现a == true, b == false的形式，而是

```
// 这样的显式用法没问题：
if (a) {
// ..
}
// 这样的显式用法更好：
if (!!a) {
// ..
}
// 这样的显式用法也很好：
if (Boolean( a )) {
// ..
}
```

4.5.1.3 null和undefined的比较

(1) 如果 x 为 null，y 为 undefined，则结果为 true。 (2) 如果 x 为 undefined，y 为 null，则结果为 true。

- 条件判断 a == null 仅在 doSomething() 返回非 null 和 undefined 时才成立

```
var a = doSomething();
if (a == null) {
// ..
}
// 相当于
var a = doSomething();
if (a === undefined || a === null) {
    // ..
}
```

4.5.1.4 对象和非对象的比较

(1) 如果 Type(x) 是字符串或数字，Type(y) 是对象，则返回 x == ToPrimitive(y) 的结果； (2) 如果 Type(x) 是对象，Type(y) 是字符串或数字，则返回 ToPromitive(x) == y 的结果。

- 即将对象转化为基本类型再进行比较

4.5.1.5 特殊情况

```
"0" == null; // false
/**
*   字符串 == null
*   ToNumber("0")--> 0 == null
*   false
*/
"0" == undefined; // false
/**
*   字符串 == undefined
*   ToNumber("0")--> 0 == undefined
*   false
*/
"0" == false; // true -- 晕!
/**
*   字符串 == 布尔
*   "0" == ToNumber(false) -> 0
*   字符串 == 数值
*   ToNumber("0")--> 0 == 0
*   True
*/
"0" == NaN; // false
/**
*   字符串 == NaN
*   ToNumber("0")--> 0 == NaN
*   false
*/
"0" == 0; // true
/**
*   字符串 ==数值
*   ToNumber("0")--> 0 == 0
*   true
*/
"0" == ""; // false
/**
*   字符串 == 字符串
*   false
*/

false == null; // false
/**
*   布尔值 == null
*   ToNumber(false)--> 0 == null
*   false
*/
false == undefined; // false
/**
*   布尔值 == undefined
*   ToNumber(false)--> 0 == undefined
*   false
*/
false == NaN; // false
/**
*   布尔值 == NaN
*   ToNumber(false)--> 0 == NaN
*   false
*/
false == 0; // true -- 晕!
/**
*   布尔值 == 数值
*   ToNumber(false)--> 0 == 0
*   true
*/
false == ""; // true -- 晕!
/**
*   布尔值 == 字符串
*   ToNumber(false)--> 0 == ""
*   数值 == 字符串
*   0 == ToNumber("") --> 0
*   true
*/
false == []; // true -- 晕!
/**
*   布尔值 == 对象
*   ToNumber(false)--> 0 == []
*   数值 == 字符串
*   0 == ToNumber([]) --> 0
*   true
*/
false == {}; // false
/**
```

```
*   布尔值 == 对象
*   ToNumber(false)--> 0 == {}
*   数值 == 字符串
*   0 == ToNumber([]) --> [object, Object] --> NaN
*   false
*/

"" == null; // false
"" == undefined; // false
"" == NaN; // false
"" == 0; // true -- 晕!
// toNumber("") -> 0
"" == []; // true -- 晕!
// toNumber("") -> 0 toNumber([]) == 0
"" == {}; // false
// toNumber("") -> 0 toNumber([])

0 == null; // false
0 == undefined; // false
0 == NaN; // false
0 == []; // true -- 晕!
/**
*   数值 == 对象
*   0 == ToNumber([]) --> 0
*   true
*/
0 == {}; // false
/**
*   数值 == 对象
*   0 == ToNumber({}) --> [object, Object] --> NaN
*   false
*/

[] == ![] // true
// !先执行，将其转化为对应的相反的布尔值：false, [] == false
// 都ToNumber([]) --> 0 == ToNumber(false) == 0
```

5.5.2 抽象关系比较

- === 是没有对应的抽象关系比较的
- a </> b额隐式强制转化
  - 比较双方都是字符串类型
    - 如果比较双方都是字符串，则按字母顺序来进行比较
  - 其他情况
    - 双方首先调用 ToPrimitive，如果结果出现非字符串，就根据 ToNumber 规则将双方强制类型转换为数字来进行比较
- />=, <=的在js中的处理方式
- a >= b 的result为 !(a < b)
- a <= b 的result为 !(a > b)

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // false
a == b; // false
a > b; // false

a <= b; // true
/*
*  因为a>b为false，所以取反为true
*/
a >= b; // true
```

5.5.3 == 使用的注意事项

- 如果两边的值中有true或false，千万不要使用==
- 如果两边的值中有[],""或者0, 尽量不要使用==

5.6 总结

- 强制类型转换：包括显式和隐式
- 显式强制类型转换明确告诉我们哪里发生了类型转换，有助于提高代码可读性和可维护性。
- 隐式强制类型转换则没有那么明显，是其他操作的副作用。感觉上好像是显式强制类型转换的反面，实际上隐式强制类型转换也有助于提高代码的可读性。

三 js的基础语法

1. 数组

- 在js获取一个不存在的索引值，不会保存，结果为undefined

1.1. 创建数组的两种方法

- 方法1 new Array(obj) / Array(obj)
  - obj
    - 参数列表
      - 不推荐只有一个参数,因为该参数是长度，不符合规则
    - array
- 方法2 Array.of(obj) --- ES6
  - obj
    - 参数列表
    - 只有一个参数，该参数指的是第一个元素
  - array
- 方法3：字面量

```
let arr1 = new Array(1, 2, 3);
let arr2 = Array.of(3);
let arr3 = [1, 2, 4];

console.log(arr1);
console.log(arr2);
console.log(arr3);

// [1, 2, 3]
// [3]
// [1, 2, 4]
```

1.2 数组元素的增删改查

1.2.1 数组元素的添加

1.2.1.1 利用索引直接添加

- js数组可以直接通过索引添加数组，其他没有被添加的数组则默认为undefined

1.2.1.2 Array#push(...val)

- 向数组的后面添加val元素，返回undefined

1.2.1.3 Array#unshift(...val)

- 向数组的前面添加val元素，返回undefined

1.2.1.4 Array#fill(val[start[, end]]);

- 填充[start, end)的元素为val
  - 若没有start，则填充全部
  - 若没有end，则从start开始填充到最后

1.2.2 数组元素的删除

1.2.1.2 Array#pop()

- 删除数组的最后一个元素，返回该元素，若数组为空，则返回undefined

1.2.1.3 Array#unshift()

- 删除数组的最前面一个元素，返回该元素，若数组为空，则返回undefined

1.2.3 数组元素的修改

1.2.4 数组元素的访问

1.2.4.1 [index]

- 通过索引访问第index元素

1.2.4.2 Array#indexOf（val)

- 返回值为val的元素的序列
  - 若不存在，则返回-1

1.3 数组的常用函数

1.3.1 Array.length

- 返回数组当前的长度

1.3.2 Array#join([separator])

- 参数
  - sparator: 指定一个字符串来分隔数组的每个元素
    - 默认值为','
- 返回用其分割的字符串

1.3.3 Array#toString()

- 返回以逗号分割的字符串

1.3.4 Array#slice([begin, [end]])

- 参数
  - 没有begin：截取整一个数组
  - begin: 从begin开始截取该数组
    - 若没有end
      - begin为正数，从begin开始到最后
      - begin为负数，从倒数第i个到最后一个
  - end
    - 到end结束，不包括end
- 返回[start,end)截取的数组

1.3.4.1 使用Array#slice将类数组转化为数组

```
function list() {
  return Array.prototype.slice.call(arguments);
  // return [].slice.call(arguments)
}

var list1 = list(1, 2, 3); // [1, 2, 3]
```

1.3.5 Array#splice(start[, deleteCount[, item1[, item2[, ...]]]])

- 通过删除或替换现有元素或者原地添加新的元素来修改数组
- 返回
  - 数组形式返回原数组中被修改的内容。
- 参数
  - start：开始的index
  - deletCounet：删除的数量
  - item[1-n]:添入的元素

1.3.5.1 Array#splice的应用

```
  // 1. 用于替换数组中的第index元素
  // Array#splice(index, 1, replacement)
  let arr = [1, 2, 4];
  console.log("arr: ", arr);
  arr.splice(1, 1, 3);
  console.log("arr: ", arr);
// [1, 2, 4]  [1, 3, 4]

  // 2. 用于向index插入元素
  // Array#splice(index, 0, addition)
  let arr1 = [1, 2, 4];
  console.log("arr1: ", arr1);
  arr1.splice(1, 0, 3);
  console.log("arr1: ", arr1);
// [1, 2, 4]  [1, 5]

  // 3. 删除index之后(包含index）的元素
  // Array#splice(index, Array.lenght - index)
  let arr2 = [1, 2, 4];
  console.log("arr2: ", arr2);
  arr2.splice(1, arr2.lenght - 1);
  console.log("arr2: ", arr2);
// [1, 2, 4]  [1]
```

1.3.6 Array#sort([func])

- 排序后的数组。请注意，数组已原地排序，并且不进行复制。
- func(ele1, ele2)
  - 按照func函数进行排列
    - return 1 -- ele1,ele2的位置需要改变
    - return -1 -- ele1,ele2的位置是不需要改变

```
class Foo {
  constructor(value) {
    this.value = value;
  }
}
const arr = [new Foo(3), new Foo(5), new Foo(4)];
/*无法进行排序
arr.sort();
console.log(arr);
*/
arr.sort((ele1, ele2) => {
  if(ele1.value < ele2.value)
    return -1;
  else
    return 1;
});
console.log(arr);

// 3, 4, 5
```

1.3.7 Array#reverse()

- 返回颠倒后的数组

1.3.8 Array#concat(value1[, value2[, ...[, valueN]]])

- 参数
  - value[1-n]: 值或者数组
- 返回连接后的数组
  - 使用的是浅拷贝

1.4 数组的函数编程

1.4.1 Array#forEach(callback(currentValue [, index [, array]])[, thisArg])

- 遍历数组对象
- 返回undefined
  - 里面的修改会影响原arr
- 参数
  - callback
    - 遍历的回调函数
    - 注意三者的顺序
  - thisArg
    - 回调函数中this的指向

1.4.2 Array#map(callback(currentValue [, index [, array]])[, thisArg])

- 遍历数组对象并返回经过回调函数筛选（即结果为true）的数组
  - 里面的修改不会影响原arr
- 参数
  - callback
    - 遍历的回调函数
    - 注意三者的顺序
  - thisArg
    - 回调函数中this的指向

1.4.3 Array#filter(callback(currentValue [, index [, array]])[, thisArg])

- 遍历数组对象并返回经过回调函数筛选后的数组
  - 里面的修改不会影响原arr
- 参数
  - callback
    - 遍历的回调函数，必须返回布尔值
    - 注意三者的顺序
  - thisArg
    - 回调函数中this的指向

1.4.4 Array#some(callback(element[, index[, array]])[, thisArg])

- 遍历数组对象并返回经过回调函数后的布尔值
  - 若里面每一个都满足回调函数的条件，则返回true
  - 里面的修改不会影响原arr
- 参数
  - callback
    - 遍历的回调函数，必须返回布尔值
    - 注意三者的顺序
  - thisArg
    - 回调函数中this的指向

1.4.5 Array#every(callback(element[, index[, array]])[, thisArg])

- 遍历数组对象并返回经过回调函数后的布尔值
  - 若里面存在满足回调函数的条件，则返回true
  - 里面的修改不会影响原arr
- 参数
  - callback
    - 遍历的回调函数，必须返回布尔值
    - 注意三者的顺序
  - thisArg
    - 回调函数中this的指向

1.4.6 Array#reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

- 遍历数组对象并归约数组，并返回最后的结果
  - 里面的修改不会影响原arr
- 参数
  - callback
    - 遍历的回调函数，必须返回布尔值
    - 第一个参数为preValue
  - initialValue
    - 初始值，默认为0

```
function sum(...argv) {
  return argv.reduce((preValue, currentValue) => preValue + currentValue, 0);
}

function mul(...argv) {
  return argv.reduce((preValue, currentValue) => preValue * currentValue, 1);
}
const arr = [1, 2, 3, 4, 5];
console.log(sum(...arr));
console.log(mul(...arr));
//   15 120
```

1.5. 遍历数组

- 方法1：
  - Array#forEach((value, index) => {...})
- 方法2：
  - for...in...
    - 遍历index
- 方法3：
  - for...of...
    - 遍历value
- 方法4
  - for(let i = 0;i < arr.length;i ++)

1.6. 注意事项

1.6.1. 数组名的赋值

- 不要直接给数组名赋值，否则数组指向的堆的内存会被垃圾处理代码清空

2. 对象

2.1. 面向对象的编程

2.1.1 面向过程的编程(POP)(Process-oriented programming)

- 分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候在一个一个依次调用即可

2.1.2. 面向对象编程(OOP)(Oject-oriented programming)

- 把事务分解成一个个对象，然后对象之间分工与合作
- 特点
  - 封装性
  - 继承性
  - 多态性

2.2. 面向对象编程的思维特点

- 抽取（抽象）对象共用的属性和行为组成（封装）成一个类（模板）
- 对类进行实例化，获取类的对象

2.3. 对象的基本组成

2.3.1. 属性

事物的特征，在对象中用属性来表示

2.3.2. 方法

对象的行为，在对象中用方法来表示

2.4 对象

2.4.1 对象的定义形式

- 声明/文字形式
  - 添加多个键值对
- 构造形式
  - 逐个添加属性

```
// 文字形式
var obj = {
    // ...
}
// 构造形式
var obj = new Object();
myObj.key = value;
```

2.4.2 对象的分类

- 根据对象的定义形式，可以分为两类

2.4.2.1 普通对象

- 方法1：直接通过字面量进行定义
- 方法2：通过new Object()来进行定义
  - 实际上也可以成为Object的实例对象，但这里暂且不这么认为

```
let obj = {
  a: 4;
}

let obj1 = new Object();
```

2.4.2.2 类的实例对象

- 方法1：通过new和构造函数创建对象
- 方法2：对象委派
  - Object.create(o)

```
let obj = new Foo();

let obj = Object.create(Foo.prototype)
// 实例对象的__proto__指向类的原型对象
```

2.4.3 对象的属性访问和修改

- 在对象中，属性名永远都是字符串。如果你使用 string（字面量）以外的其他值作为属性名，那它首先会被转换为一个字符串,即使是数字也不例外

2.4.3.1 属性访问

- 只能使用对应的属性值才能进行访问
- 若不是访问而是赋值
  - 可以用于添加新的属性和值（必须有两者）

```
var myObject - {
    a: 2
}
console.log(myObject.a)
```

2.4.3.2 键值访问

- 要求键值的规范
  - 而 ["..."] 语法可以接受任意 UTF-8/Unicode 字符串作为属性名
  - 可以使用字符串的变量来进行访问
  - ES6增加了可计算属性名，即可以使用表达式来作为[]内的字符串
- 若不是访问而是赋值
  - 可以用于添加新的属性和值（必须有两者）

```
var myObject - {
    a: 2
}
console.log(myObject["a"])
```

2.4.4 理解对象属性访问的默认操作

2.4.4.1 [Get]

- myObject.a 在 myObject 上实际上是实现了 [[Get]] 操作
  - 对象默认的内置 [[Get]] 操作首先在当前对象中查找是否有名称相同的属性，如果找到就会返回这个属性的值。
  - 如果没有找到遍历可能存在的 [[Prototype]] 链，也就是原型链
  - 还是没有找到，放回undefined
- 这种方法和访问变量时是不一样的。如果你引用了一个当前词法作用域中不存在的变量，并不会像对象属性一样返回 undefined，而是会抛出一个 ReferenceError 异常：

2.4.4.2 [[Put]]（用于属性的新增和修改）

- [[Put]] 算法大致会检查下面这些内容。
  - 1. 属性是否是访问描述符？如果是并且存在 setter 就调用 setter。
  - 2. 属性的数据描述符中 writable 是否是 false？如果是，在非严格模式下静默失败，在严格模式下抛出 TypeError 异常。
  - 3. 如果都不是，将该值设置为属性的值。

2.4.4.3 如何判断是否具有某个属性

- 方法一：使用属性访问，若实例对象和原型链中没有该属性，则返回undefined

```
obj.a //返回undefined
```

- 使用
- 方法二：使用in，若实例对象和原型链中没有该属性，则返回false

```
prop in obj
```

- 方法3 若实例对象中没有该属性，则返回false
  - 注意不检查原型链
  - Object.prototype.hasOwnProperty.call(obj, prop) 或者
  - obj.hasOwnProperty(prop)

```
obj.hasOwnProperty(prop)
```

2.4.5 对象的复制

- 对象的存储
  - obj[key]即对应的value
    - 基本类型：存储该值
    - 复杂类型：存储该复杂类型数据所在堆中的地址

2.4.5.1 浅复制

- 基本数据类型有进行复制
  - Object.assign(target, ...sources)
- 复杂数据类型只是指向对应的堆的地址，没有对其进行赋值

2.4.5.2 深复制

```
const newObj = {}
function deepCopy(oldObj, newObj) {
    Object.keys(oldObj).forEach((key, index) => {
        let value = oldObj[key];
        if(value instanceof Array) {
            newObject[key] = [];
            deepCopy(value, newObject[key]);
        }
        else if(value instanceof Object) {
            newObject[key] = {};
            deepCopy(value, newObject[key]);
        }
        else {
            newObject[key] = value;
        }
    })
}
```

2.4.6 对象的属性描述符

- 对象属性对应的值实际上是一个对象
  - value
    - 该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。
    - 默认为undefined
  - 后面三个对应了该属性的一些特性（即属性描述符）

    - writable
        - 当且仅当该属性的 writable 键值为 true 时，属性的值，也就是上面的 value，才能被赋值运算符改变。
        - 默认值为false
    - configurable
        - 当且仅当该属性的 configurable 键值为 true 时，该属性的描述符才能够被改变，同时该属性也能从对应的对象上被删除。
        - 默认值为false
    - enumerable
        - 当且仅当该属性的 enumerable 键值为 true 时，该属性才会出现在对象的枚举属性中。
        - 默认值为false
    - get
        - 属性的 getter 函数。当访问该属性时，会调用此函数。执行时不传入任何参数，但是会传入 this 对象（由于继承关系，这里的this并不一定是定义该属性的对象）。该函数的返回值会被用作属性的值。
        - 默认为undefined
    - set
        - 属性的 setter 函数。当属性值被修改时，会调用此函数。该方法接受一个参数（也就是被赋予的新值），会传入赋值时的 this 对象。
        - 默认为undefined
- 若在不允许的情况下对相应的属性描述符或者属性值进行修改，则会返回TypeError

```
{
    value: 2,
    writable: true/false;
    configurable: true/false;
    enumerable: true/false;
    // 在访问该属性时调用
    get() {
        // ...
        return
    }
    // 在修改该属性时调用
    set() {
        // ...
    }
}
```

2.4.6.1 对对象属性特性的操作

2.4.6.1.1 Object.defineProperty()

- 给obj添加/修改一个普通的属性并显式指定/修改了一些特性

```
Object.defineProperty(obj, prop, descriptor)
```

- 参数
    - obj：要定义属性的对象
    - prop: 要定义或修改的属性的名称或 Symbol
    - descriptor: 要定义或修改的属性描述符。(使用对象)
- 返回值
    - 被传递给函数的对象。

```
var obj = {};
Object.definePropety(obj, "a", {
    value: 2,
    writable: false,
    configurable: true,
    enumerable: true
})
```

2.4.6.2 对象属性的不变性

2.4.6.2.1 对象常量

- 将对象属性特性
    - writable: false
    - configurable: false

2.4.6.3 对象的不变性

2.4.6.3.1 Object.preventExtensions(obj)

- 禁止对一个对象增添新的属性并且保存已有的属性
    - Object.preventExtensions()仅阻止添加自身的属性。但其对象类型的原型依然可以添加新的属性。

```
Object.preventExtensions(obj);
Object.isExtensions(obj)
// 判断一个对象是否是不可扩展的对象
```

- 参数

    - obj：将要变得不可扩展的对象。
- 返回值
    - 已经不可扩展的对象。

2.4.6.3.2 Object.seal(obj)

- **proto** ( ) 属性的值也会不能修改。
- Object.preventExtensions(obj) + configurable: false

```
Object.seal(obj);
```

- 参数
    - obj：将要变得不可扩展的对象。
- 返回值
    - 已经不可扩展的对象。

2.4.6.3.3 Object.freeze(obj)

- **proto** ( ) 属性的值也会不能修改。
- Object.seal(obj) + wriable: false

```
Object.freeze(obj);
```

- 参数
    - obj：将要变得不可扩展的对象。
- 返回值
    - 已经不可扩展的对象。

2.4.6.4 不可枚举属性的问题

- 常见的不可枚举的属性
    - symbol属性
    - constructor属性
    - **proto**, prototype
    - 可枚举属性值为false

2.4.6.4.1 obj.propertyIsEnumerable(prop)

- 会检查给定的属性名是否直接存在于对象中

```
obj.propertyIsEnumerable(prop);
```

2.4.6.4.2 Object.getPropertySymbols(obj)

- 以数组形式返回对象中的symbol属性

```
Object.getPropertySymbols(obj)
```

2.4.7 对象的遍历

- 利用枚举函数

2.4.7.1 Object.keys(obj)

- 会返回一个数组，包含所有可枚举属性
- 属性只是当前对象的属性，不包含原型链

2.4.7.2 Object.getOwnPropertyNames(obj)

- 会查找对象直接包含的属性(包括不可枚举类型)
- 属性只是当前对象的属性，不包含原型链

2.4.7.3 for...in...(key)

- 可以用来遍历对象的可枚举属性列表（包括 [[Prototype]] 链）

2.4.7.4 for...of...(value)

- 遍历属性的值
- 本质
    - 循环首先会向被访问对象请求一个迭代器对象，然后通过调用迭代器对象的next() 方法来遍历所有返回值

2.4.8 总结

- JavaScript 中的对象有字面形式（比如 var a = { .. }）和构造形式（比如 var a = new Array(..)）。字面形式更常用

- 对象是 6 个（或者是 7 个，取决于你的观点）基础类型之一。对象有包括 function 在内的子类型，不同子类型具有不同的行为，比如内部标签 [object Array] 表示这是对象的子类型数组。
- 对象就是键 / 值对的集合。可以通过 .propName 或者 ["propName"] 语法来获取属性值。访问属性时，引擎实际上会调用内部的默认 [[sGet]] 操作（在设置属性值时是 [[Put]]），[[Get]] 操作会检查对象本身是否包含这个属性，如果没找到的话还会查找 [[Prototype]]链
- 属性的特性可以通过属性描述符来控制，比如 writable 和 configurable。此外，可以使用Object.preventExtensions(..)、Object.seal(..) 和 Object.freeze(..) 来设置对象（及其属性）的不可变性级别。
- 属性不一定包含值——它们可能是具备 getter/setter 的“访问描述符”。此外，属性可以是可枚举或者不可枚举的，这决定了它们是否会出现在 for..in 循环中。
- 你可以使用 ES6 的 for..of 语法来遍历数据结构（数组、对象，等等）中的值，for..of会寻找内置或者自定义的 @@iterator 对象并调用它的 next() 方法来遍历数据值。

2.5 类

2.5.1. 类的构造

2.5.1.1 通过函数构造类（构造函数）

- 对象属性
  - 定义
    - this.attrName = ...
    - 子类会创建一个新的内存空间来存储这些属性
- 类的原型对象方法
  - 定义： className.prototype.funcName = function(argv) {...}
  - 置于原型链之中, 在定义时，可以通过this调用其他函数
  - 通过实例对象才能调用的方法（实际上是通过原型链进行调用）
    - classObj.funcName(argv)
- 类方法
  - 定义：
    - className.funcName = function(argv) {...}
  - 通过类名能够调用的方法, 不能被实例对象所调用
    - className.funcName(argv)

```
function Foo(name) {
    this.name = name;
}
Foo.sayKeyWord = function() {
  console.log("hello world");
}
Foo.prototype.sayName = function() {
  console.log(this.name)
}

const obj = new Foo("bulumrcai");
obj.sayName();
Foo.sayKeyWord();

console.log("sayKeyWord can be excuted by obj :", obj.sayKeyWord);
console.log("sayName can be excuted by Foo :", Foo.sayName);
/*
bulumrcai
hello world
sayKeyWord can be excuted by obj : undefined
sayName can be excuted by Foo : undefined
*/
```

2.5.1.2 通过class构造类

- 对象属性（在构造函数constructor中定义）
  - 定义
    - this.attrName = ...
    - 子类会创建一个新的内存空间来存储这些属性
- 类的原型对象方法
  - 直接在class内部定义：
    - funcName(argv) {...}
  - 置于原型链之中, 在定义时，可以通过this调用其他函数
  - 通过实例对象才能调用的方法（实际上是通过原型链进行调用）
    - classObj.funcName(argv)
- 类方法
  - 定义(通过static进行定义）：
    - static funcName(argv) {...}
  - 通过类名能够调用的方法, 不能被实例对象所调用
    - className.funcName(argv)

```
function Foo {
  constructor() {
    this.name = name;
  }
  static sayKeyWord() {
    console.log("hello world");
  }
  sayName() {
    console.log(this.name);
  }
}

const obj = new Foo("bulumrcai");
obj.sayName();
Foo.sayKeyWord();

console.log("sayKeyWord can be excuted by obj :", obj.sayKeyWord);
console.log("sayName can be excuted by Foo :", Foo.sayName);
/*
bulumrcai
hello world
sayKeyWord can be excuted by obj : undefined
sayName can be excuted by Foo : undefined
*/
```

2.5.2 类的实例对象（通过new构造的对象）

- 通过new + 构造函数（class）创建原型对象
  - new使构造函数中会做的四件事情：
    - 创建一个新的空的对象
    - 这个新对象会执行原型链接
      - obj.**proto** = Object.create(funcName.prototype)
    - 这个新的对象会绑定到函数调用的this
    - 如果函数没有返回其他对象，那么表达式中的函数调用会自动返回这个新的对象

2.5.2.1. 类的实例对象的成员类型(实例成员，静态成员)

- 构造函数中的属性和方法被称为成员
- 实例成员
  - 构造函数内部通过this添加的成员（this.sayName = functino(){})或这通过原型对象添加的成员
    - 通过this创建的成员在新的实例对象创建时会再存储一次
    - 通过原型对象添加的成员只需存储一次，可以通过原型链进行访问
  - 实例成员只能通过实例化的对象来访问，不可以通过函数来访问实例成员
- 静态成员
  - 在构造函数本身上添加成员
  - 静态成员只能通过函数来访问，不能通过对象实例来访问

2.5.3 类的实例对象的实质

- 创建新的存储空间来存储构造函数中的this指向的成员
- 将实例对象的__proto__对象指向构造函数的prototype构成原型链

2.5.4. 类的实例对象的方法查询规则

- 在实例对象查找对应的方法
- 若没有找到，通过原型链查找对应的方法

2.5.5 类的继承

- 类的继承的实质
  - 子类构造函数继承了父类构造函数的this指向的静态成员
  - 子类构造函数的prototype对象的__proto__指向父类构造函数的prototype
- 创建的子类实例对象的特点
  - 子类实例对象会为构造函数中的this分配新的存储空间以及赋相应的值
  - 子类实例对象的__proto__会指向Son.prototype,其中Son.prototype的__proto__属性则会指向Father.prototype（父类的原型对象）以及Son.prototype的新增的原型对象的方法

2.5.5.1. 原型链

- 所有普通的原型链最终都会指向内置的 Object.prototype
- 每个对象都有__proto__属性
- 当对象是通过构造函数的形式来进行构建的时候，实质上构建的过程包括将对象的__proto__指向构造函数的原型对象（funcName.prototype)
- 构造函数和构造函数之间的继承关系
  - 作为子类的构造函数的原型对象的__proto__属性会指向作为父类的构造函数的原型对象，一般通过以下的方法进行构建

```
Son.prototype = Object.create(Father.prototype);
Son.constructor = Son;
```

2.5.5.1.1 实例对象是如何通过原型链来寻找相应的方法?

- 从实例对象开始的__proto__指向的原型对象
- 第一类：实例对象
  - 实例对象的__proto__属性指向构造函数的原型对象

```
实例对象
  {
    // ...其他成员
    __proto__: { //构造函数的原型对象
      __proto__: 父类原型对象,
      constructor: 构造函数的名字,
      ...
      // ......该构造函数原型的方法
    }
  }
```

- 第二类：构造函数的原型对象
  - 构造函数的原型对象的__proto__属性指向父类的原型对象
- 第三类：父类的原型对象
  - 父类原型对象的__proto__属性指向其父类的原型对象

```
// 父类原型对象
  {
    __proto__: 父类原型对象,
    constructor: 构造函数的名字,
      ...
    // ......该构造函数原型的方法
  }
```

- 后面的都是第三类的循环
- 2.7.1.1.2 实例对象的原型链查询
- 第一步：实例对象会先在属性__proto__对象中寻找相应的方法
- 第二步：若没有找到相应的方法，则会寻找该__proto__对象中属性__proto__所指向的另一个对象中寻找方法
- 第三步：循环第二步直至找到为止
- 这一个过程即形成了原型链查询
-

2.5.5.2 原型链引起的属性设置和屏蔽

2.5.5.2.1 属性屏蔽

- 若当前对象myObject有属性prop，其原型链上也有属性prop
  - 在访问属性prop的时候，访问的是myObject上的属性值而不是原型链上的属性
- 此时称该prop为屏蔽属性

2.5.5.2.2 属性设置的过程（myObject.prop = value)

- 遍历myObject的属性，
  - 若该属性在myObject在遍历的属性中，则修改myObject的属性值（要考虑是否writable）
  - 若不在myObject的属性，则往其原型链寻找
    - 遍历原型链的属性
      - 若该属性不在原型链上，则向myObject添加该属性以及对应的属性值
      - 若该属性在原型链上
        - 若该属性在原型链上不是只读的类型，则在myObject上添加该属性以及对应的属性值，此时该属性起到了屏蔽的作用
        - 若该属性在原型链上且是只读类型，由于继承的原因，myObject的属性也是只读类型，返回TypeError
        - 如果在原型链上存在该属性并且它是一个 setter。foo 不会被添加到（或者说屏蔽于）myObject，也不会重新定义该属性这个 setter。
- 属性设置的过程

2.5.5.2.3 属性屏蔽和属性设置的易错点--隐式屏蔽

- 在属性值进行++的过程，会有一个属性设置的过程，即给当前对象新增一个属性

```
    var obj = {
        a: 2
    }
    var newObj = Object.create(obj)

    console.log(newObj.a)

    newObj.a ++;
    // newObj.a = newObj.a + 1 相当于一个设置属性的过程
    console.log(obj.a)
    console.log(newObj.a)
    // 2
    // 2
    // 3
```

- 若想要使obj的2增加，必须使用obj.a++

2.5.5.3 通过构造函数实现类的继承

- 称为组合继承
- 类的继承的实质
  - 子类构造函数继承了父类构造函数的this指向的静态成员，借用call来实现显示地改变函数中this指向的对象
  - 子类构造函数的prototype对象的__proto__指向父类构造函数的prototype，通过Object.create(Father.prototype)来构建__proto__属性指向Father.prototype的对象

```
function Father(name) {
  this.name = name;
}
Father.welcome = function() {
  console.log("welcome");
}
Father.prototype.sayName = function() {
  console.log(this.name);
}

function Son(name, label) {
  Father.call(this, name);
  this.label = label;
}
Son.prototype = Object.create(Father.prototype);
// Object.create(Father.prototype)
// 创建了一个对象，该对象的__proto__属性指向Father.prototype
Son.prototype.constructor = Son;

Son.prototype.sayLabel = function() {
  console.log(this.label);
}
console.dir(Son);

const son = new Son("bulumrcai", "hello world");
console.dir(son);
son.sayLabel();
son.sayName();
// Son.welcome(); // TypeError
// 若要正常使用静态成员，则需要加上
Son.prototype.constructor.__proto__ = Father;
Son.welcome();
```

- ;

2.5.5.2.1 为什么不能使用Son.prototype.proto = Father.prototype来实现原型链的继承

2.5.5.2.2 为什么不能使用Son.prototype = new Father()

- new Father()会创建一个Father的实例对象

```
{
  // ...Father中this指向的属性
  __proto__: Father.prototype
}
```

- 会发现该对象的__proto__确是指向Father.prototype,但是会引入Father的其他一些属性，可能会造成意想不到的错误

2.5.5.2.3 为什么不能使用Son.prototype = Father

- Father是一个函数而不是一个对象
- Father中的__proto__属性指向的是Object.prototype而不是Father.prototype

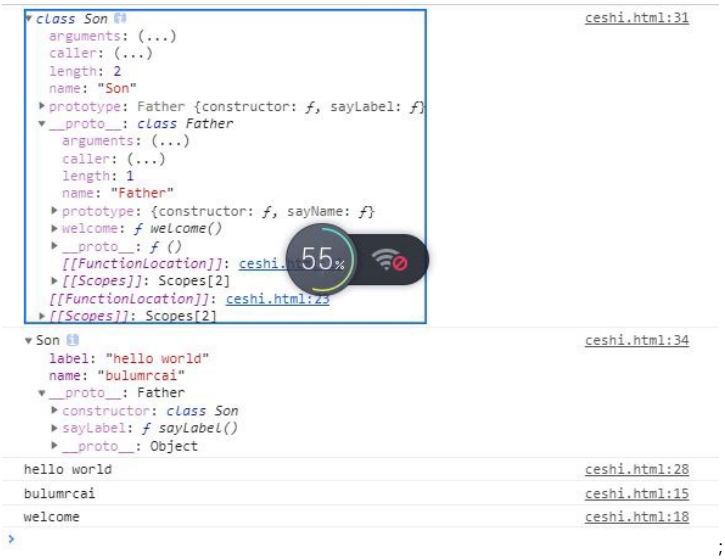
2.5.5.2.4 如何实现父类静态成员的继承？

- 为什么通过原型链不能继承
  - 静态成员并不位于原型对象之中，因此不能使用原型链继承
- 经过研究，我发现该静态成员实际上位于构造函数原型对象的constructor的__proto__属性之中，因此用一下的方法来时实现静态成员的继承

```
Son.prototype.constructor.__proto__ = Father;
```

2.5.5.4 通过class中的extends实现继承

```
class Father {
  constructor(name) {
    this.name = name;
  }
  sayName() {
    console.log(this.name);
  }
  static welcome() {
    console.log("welcome");
  }
}

class Son extends Father {
  constructor(name, label) {
    super(name);
    this.label = label;
  }
  sayLabel() {
    console.log(this.label);
  }
}
console.dir(Son);

const son = new Son("bulumrcai", "hello world");
console.dir(son);
son.sayLabel();
son.sayName();
Son.welcome(); // 能够正常的使用
```



- ;

2.5.5.3.1 class中的super对象

- 在子类的构造函数中，super指的是父类的constuctor函数
  - 调用的时候必须在前面，通过super()进行调用
- 在子类的其他函数中，super指的是原型链上的对象
  - 通过super.funcName()进行调用

2.5.5.3.2 class中的static函数

- 即静态成员，只能通过函数名进行调用，且不位于原型链之中。
- 实际上是位于constructor的__proto__属性之中

2.5.6 类的基本认识

- JavaScript 中实际上没有类，由于类是一种设计模式，所以你可以用一些方法（本章之后会介绍）近似实现类的功能。
- 构造函数
  - 类实例是由一个特殊的类方法构造的，这个方法名通常和类名相同，被称为构造函数
  - 类构造函数属于类，而且通常和类同名，构造函数大多需要用 new 来调
- 类的继承
  - 父类
  - 子类
  - 父母的基因特性会被复制给孩子
  - 子类会包含父类行为的原始副本，但是也可以重写所有继承的行为甚至定义新行为。
- 多态
  - 子类可以重写父类中名字相同的方法，且可以进行调用
  - 在继承链的不同层次中一个方法名可以被多次定义，当调用方法时会自动选择合适的定义
    - 会使用实例化对象的方法
- 混入

2.5.7 类的易混点

- js中并没有所谓的类，只有对象，而是一种类似类的形式
- 所有的函数都会有一个名为prototype的公有并且不可枚举的属性，该属性的值是一个对象，该对象的__proto__属性的值会指向另一个对象
- 所有的对象都具有一个名为__proto__的不可枚举的属性，且该属性指向另一个对象

2.5.7.1 类的实例对象和类的关系

2.5.7.1.1 类的实例的本质

- 若根据类的构造函数创建一个实例时,该实例对象的__proto__会指向该构造函数的prototype

```
function Foo() {
    // ...
}

var obj = new Foo();

console.log(obj.__proto__ === Foo.prototype)
console.log(Object.getPrototypeOf(obj) === Foo.prototype)
// true true
```

2.5.7.1.2 原型继承

- 该实例对象的__proto__会指向该构造函数的prototype，这即是原型继承
- 对于js中的类的实例，只是通过原型相互关联，而形成了该类的实例，但本质上是一个对象

2.5.7.1.3 类的实例对象的不可枚举属性constructor本质

- 在类/构造函数进行声明的时候，就会有默认的.prototype.constructor的不可枚举属性，该属性指向本身

```
function Foo() {
    // ...
}
console.log(Foo === Foo.prototype.constructor)
// true
```

- 在通过new创建实例时，由于原型继承，所以实例中的不可枚举属性constructor实际上就是通过Get方法寻找的原型链上的constructor

```
function Foo() {
    // ...
}
Foo.prototype.constructor = {};
var obj = new Foo();
console.log(obj.constructor === Foo)
// false
```

2.5.7.2 构造函数的本质

- js中的构造函数实质上是带有new的函数调用，即原本的函数不是构造函数，当且仅当使用new时，函数调用才会变成构造函数的调用
- 调用的时候函数才能称为构造函数

2.5.7.3 instanceof判断的实质（对象与函数）

- 实际上判断的是对象的原型链中是不是有指向函数的原型对象的对象，而不能判断对象与对象之间的关联

```
console.log(obj instanceof Father)
// obj的原型链中是否有指向Fahter.prototype
```

2.5.7.4 判断某个对象是否出现在另一个对象的原型链之中

```
c.isPrototypeOf(b)
// c对象是否出现在b对象的原型链中

var obj = {};
var newObj = Object.create(obj);
/*
    {
        __proto__: obj
    }
*/
console.log(obj.isPrototypeOf(newObj))
// true
```

2.5.7.5 获取某个对象的原型链

2.5.7.5.1 Object.getPrototypeOf(obj)

- 标准的方法

2.5.7.5.2 obj.proto

- 在某些浏览器上并不适用
- 对象__proto__的实质：

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
    return Object.getPrototypeOf( this );
    },
    set: function(o) {
    // ES6 中的 setPrototypeOf(..)
    Object.setPrototypeOf( this, o );
    return o;
    }
    } );
```

2.5.7.6 总结

- 如果要访问对象中并不存在的一个属性，[[Get]] 操作（参见第 3 章）就会查找对象内部原型链关联的对象
- 关联两个对象最常用的方法是使用 new 关键词进行函数调用，即使类和实例的关系

2.5.8 注意

- 在生成实例时，new不能够省略，后面的类名一定要加上小括号
- ES6类没有变量提升，必须先定义类，才能通过类实例化对象
- 对象的属性和value的访问方法：
  - 已知具体的属性key
    - obj.key
  - 属性attr是一个变量：一般是用keys进行遍历的时候
    - obj[key]

2.5.8.1 会检查所有在原型链上的属性

- ...in..
  - for let key in obj
  - prop in obj

3 函数

3.1. 函数的定义

3.1.1. 函数的命名写法（命名函数）

```
function 函数名() {
  函数封装的代码
  ...
}

函数名()
```

3.1.2. 函数的表达式写法（匿名函数）

- 命名函数表达式

```
const foo = function 函数名() {
  函数封装代码
  ...
}

foo();
```

- 匿名函数表达式（没有函数名）

```
const foo = function() {
  函数封装代码
  ...
}

foo();
```

3.2. 函数的参数传递

3.2.1. 形参

- 定义函数时，小括号中的参数，是用来接收参数用的，在函数的内部作为变量

3.2.2. 实参

- 调用函数时，小括号中的参数，是用来把数据传递到函数的内部用的

3.2.3. 形参变量个数和实参变量个数的关系

- 形参变量个数>实参变量个数 没赋值的默认为undefined
- 形参变量个数<实参变量个数 用arguments对象进行处理

3.2.3.1. arguments对象(现在很少用了)

3.2.3.1.1. 特点1

- 是一个object类型，而不是一个数组(Array类型)，但是和数组的用法类似
- 伪数组
  - 具有数组的length属性
  - 按照索引的方式进行存储的
  - 它没有真正数组的一些方法 pop() push()等等

3.2.3.1.2. 特点2

- 是所有非箭头函数中都可用的局部变量

3.2.3.1.3. 特点3

- 存放着所有的调用者传递的参数，从0位置开始，依次存放
  - 如果调用者传入的参数多余函数接收的参数，可以通过arguments去获取所有的传入的参数

```
function print(val) {
  console.log(val);
  console.log(arguments)
}

print("hello", "world")
```

- 

- 实现多个数的相加

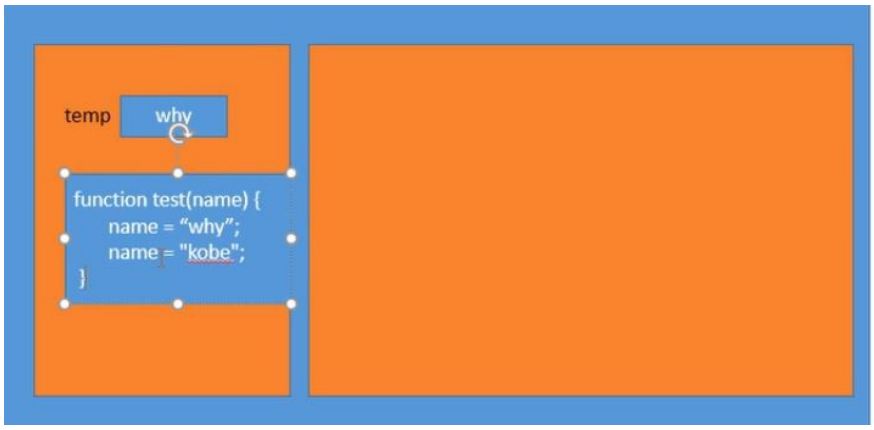
```
fuction sum() {
  arguments.reduce((prevalue, value) => prevalue + value, 0)
}
```

3.2.4. 函数参数的值传递和引用传递

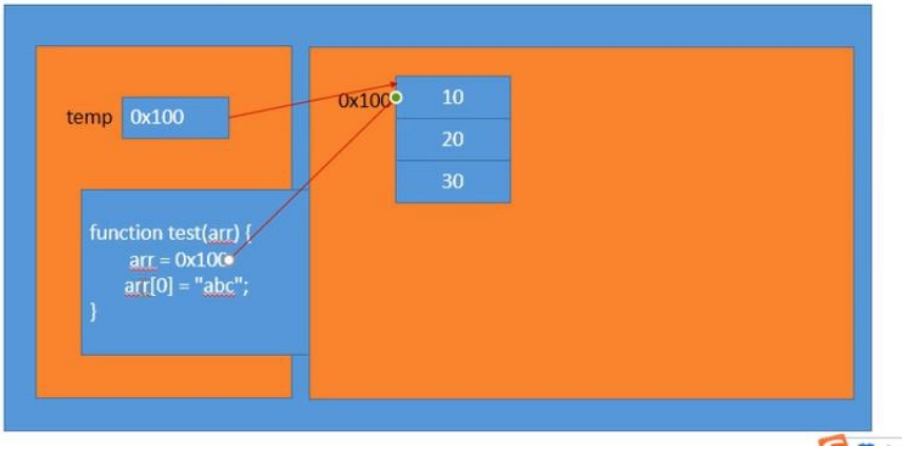
- 参数传递的本质
  - 实参将栈中的值赋给了函数中对应的形参，形参在函数的栈中开辟新的空间来存储该值

3.2.4.1. 值传递

- 适用：
  - 基本数据类型
- 特点：
  - 形参改变值不能使得实参的值改变
- 原因：基本类型的数值存储在栈中，开辟新空间来存储该值实际上就是就是对应的数值，所以值改变并不会导致对应的实参的值改变

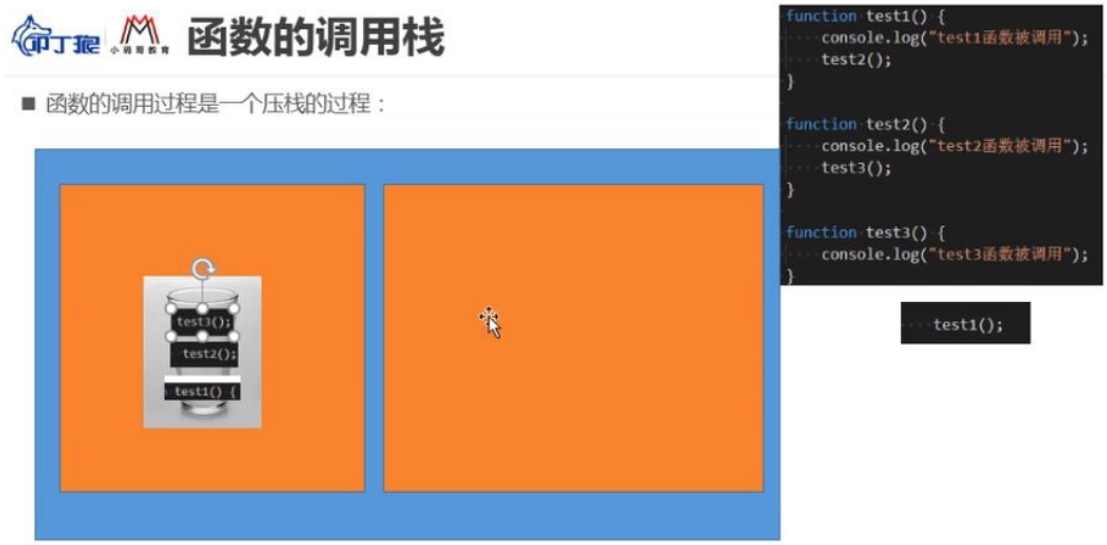
- 

3.2.4.2. 引用传递

- 适用
  - 复合数据类型(object)
- 特点：
  - 形参改变值的内容会使得实参的值改变
- 原因：复合数据类型存储在栈中的值是对应存储内容的堆的地址（即指针），参数传递实际上传的是地址，所以一旦形参改变了内容，则会导致实参指向的内容改变
- 

3.3. 函数的调用

3.3.1. 函数的调用栈

- 函数的调用过程是一个压栈的过程
- 

- 视频地址
- 在开发中尽量避免使用递归
  - 递归如果没有写好约束条件，意味着会无限调用
  - 递归调用非常占用栈空间的内存

3.3.2. 普通函数的调用

- funcName();
- funcName.call();

3.3.3 函数的特殊的调用形式

- 标签模板字面量的调用

3.3.4. 通过立即调用函数进行调（IIFE）

- 函数定义的同时立即实行函数
- 特点：执行完函数会立即销毁，对应的函数名在调用之后也会变得没有意义
- 作用
  - 创建一个独立的执行上下文环境，可以避免外界的访问修改内部的变量
- 例子见笔记

3.3.4.1. 函数声明的立即调用函数规范

- 可以使用立即调用函数，只是需要在定义加上一个小括号后在加上参数列表
- 立即调用函数的写法1：

```
(function (arg1...) {
  函数封装代码
  ...
} )(arg1...)
```

- 立即调用函数的写法2

```
(function (arg1...) {
  函数封装代码
  ...
}(arg1...) )
```

3.2.5.4.2. 函数的表达式写法可以在后面增加参数列表从而实现函数的立即实行

- 返回的是函数执行的最终结果

```
 const foo = function 函数名(arg1...) {
  函数封装代码
  ...
}(arg1...)

 const foo = function (arg1...) {
  函数封装代码
  ...
}(arg1...)
```

- foo的结果不是一个函数，而是函数的返回值

3.4. 函数的返回值

- 如果函数中没有使用return语句，那么函数有默认返回值undefined

四 JS的知识补充

1 作用域

1.1 理解作用域

1.1.1 认识三个对话的部分

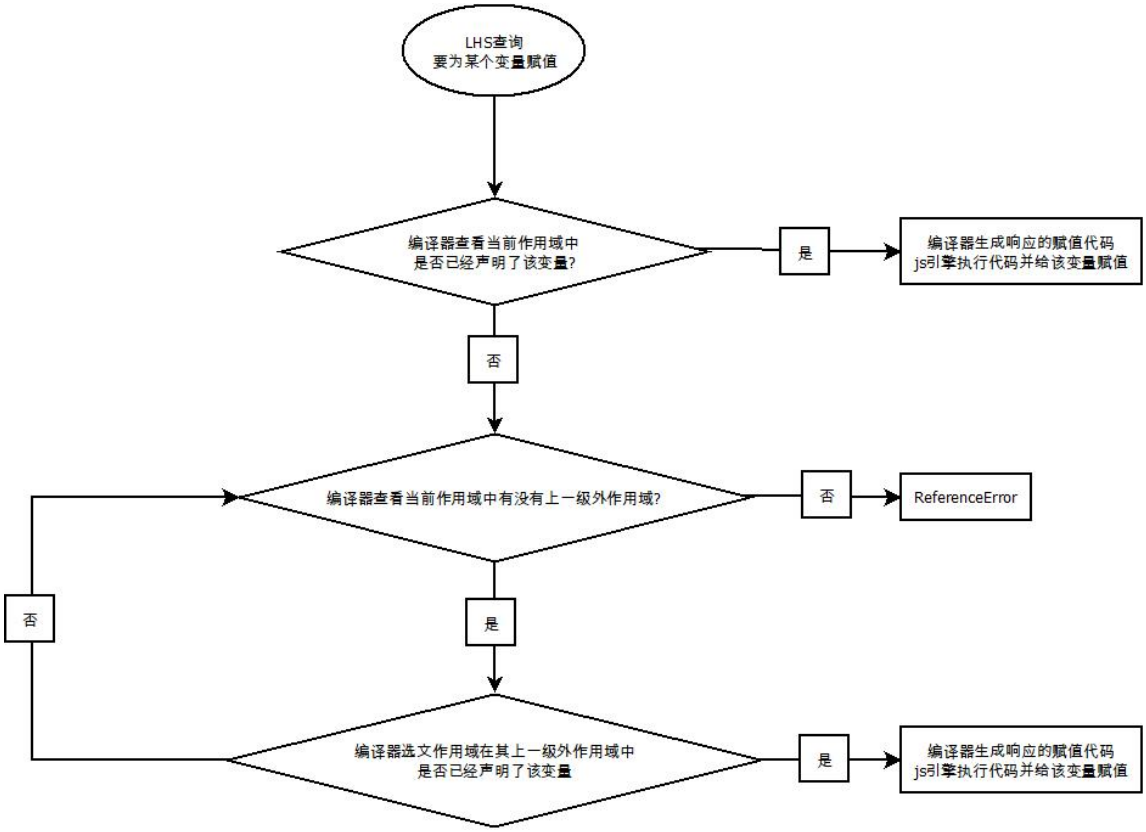
- 引擎
  - 从头到尾负责整个js程序的编译及执行过程
- 编译器
  - 负责语法分析即代码生成等等
- 作用域
  - 负责收集并维护所有声明的表示符组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限

1.1.2 对话的过程

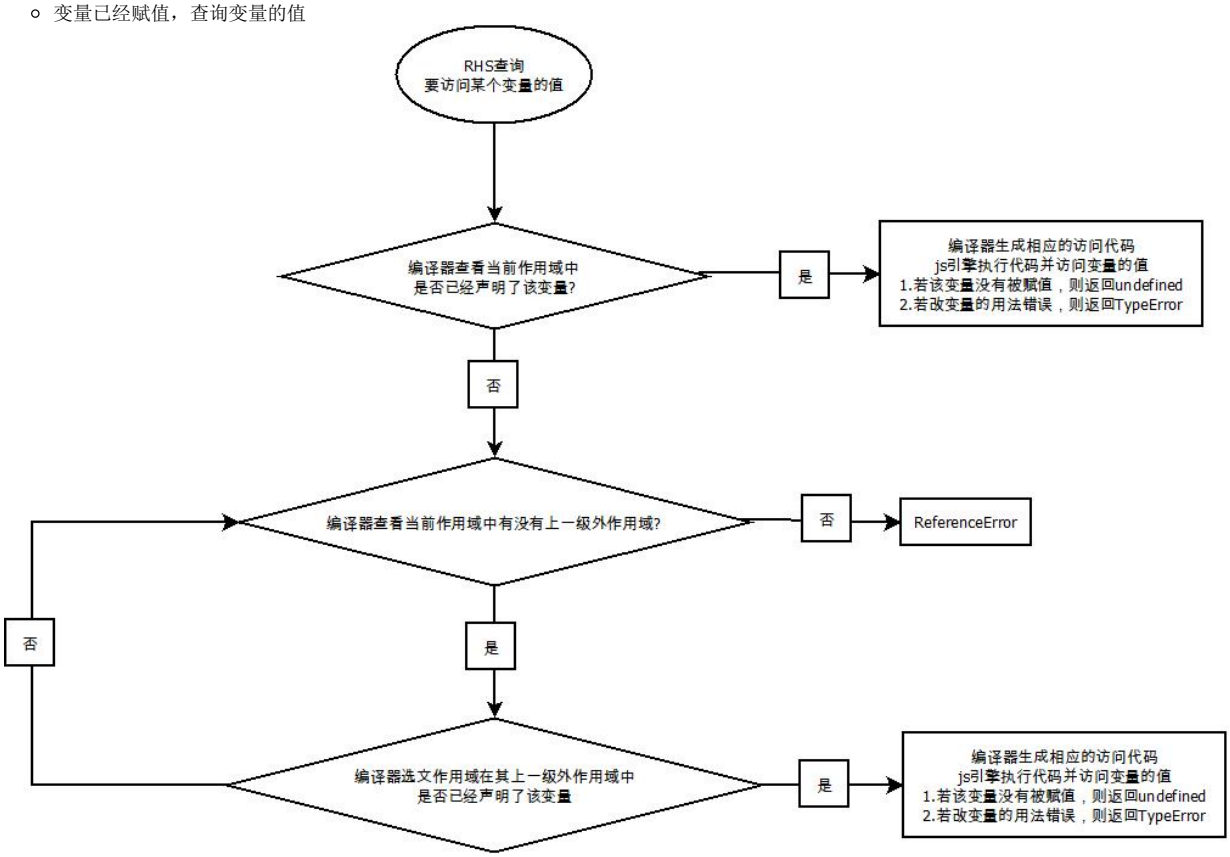
- var a = 2的三个部分的对话
  - 第一步：编译器查看当前作用域中是否有声明a
    - 若有声明，则忽略声明变量过程
    - 若没有声明，在当前作用域下声明该变量
  - 第二步：编译器生成相应的赋值代码，JS引擎执行赋值代码给a赋值

1.1.3 js引擎查询变量的方法

- LHS查询
  - 查询变量是否已经声明后并给它赋值



- RHS查询
  - 变量已经赋值，查询变量的值




```
function foo(a) {
    console.log(a)
}

foo(2);
// a = 2 为LHS查询，参数的传递
// console.log(a) 为RHS查询
```

- 练习

```
function foo(a) {
  var b = a;
  return a + b;
}
var c = foo(2)
/**
* 三次LHS查询
*   1 调用函数时，a需要进行LHS引用
*   2 b需要进行LHS引用
*   3 c需要进行LHS引用
* 四次RHS查询
*   1 调用foo函数：js引擎询问在该作用域当中有没有foo函数
*   2 赋值b时，js询问a的值
*   3 在return时，询问a,b的值
*   4 赋值c时，js询问函数的返回值
*/
```

- 认清语法LHS和RHS的重要性

```
function foo(a) {
    console.log(a + b);
    b = a;
}

foo(2)
/*
*b是一个未声明的变量，所以进行RHS查询时无法找到，是一个undefined类型，抛出ReferenceError
*若RHS查询对变量进行不符合规范的引用，如变量不是函数全被调用成函数，抛出TypeError
* 对b进行LHS查询，由于b没有声明，若在非严格模式下，js引擎会自动帮你创建一个全局变量
*若严格模式下，js引擎会抛出ReferenceError
*/
```

1.1.4 理解变量的访问

- 要分清楚是LHS还是RHS从而确认变量的访问作用域

1.2 作用域嵌套

- 当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。
  - 以{}符号作为一个块
- 遍历作用域链的过程
  - 在当前作用域中无法找到某个变量时，引擎就会在外层嵌套的作用域中继续查找，直到找到该变量，或抵达最外层的作用域（也就是全局作用域）为止
- 注意：
- var的特殊特性；非块级作用域
  - 对于var而言，只有函数的嵌套才存在错用于的嵌套
- 内部的作用域能够访问外部作用域变量，而外部作用域不能访问内部作用域的变量

1.3 总结

- 作用域是一套规则，用于确定在何处以及如何查找变量（标识符）
- 如果查找的目的是对变量进行赋值，那么就会使用 LHS 查询；如果目的是获取变量的值，就会使用 RHS 查询。
- JavaScript 引擎首先会在代码执行前对其进行编译，在这个过程中，像 var a = 2 这样的声明会被分解成两个独立的步骤：
  - 首先，var a 在其作用域中声明新变量。这会在最开始的阶段，也就是代码执行前进行。
  - 接下来，a = 2 会查询（LHS 查询）变量 a 并对其进行赋值。
- 遍历作用域链
  - LHS 和 RHS 查询都会在当前执行作用域中开始，如果有需要（也就是说它们没有找到所需的标识符），就会向上级作用域继续查找目标标识符，这样每次上升一级作用域（一层楼），最后抵达全局作用域（顶层），无论找到或没找到都将停止。
- 异常抛出
  - 不成功的 RHS 引用会导致抛出 ReferenceError 异常。（没有进行声明）
  - 不成功的 LHS 引用会导致自动隐式地创建一个全局变量（非严格模式下），该变量使用 LHS 引用的目标作为标识符，或者抛出 ReferenceError 异常（严格模式下）。

1.4 词法作用域

- 词法作用域是一种静态作用域，即某个变量在声明的时候就已经确定了其作用域范围，该变量的作用域即是该变量的词法作用域

1.4.1 欺骗词法

1.4.1.1 eval()

```
function foo(str, a) {
    eval(str); // 欺骗
    console.log(a, b);
}
var b = 2;
foo("var b = 3;", 1) // 1, 3
// 当 console.log(..) 被执行时，会在 foo(..) 的内部同时找到 a 和 b，但是永远也无法找到外部的 b
```

- 默认情况下，如果 eval(..) 中所执行的代码包含有一个或多个声明（无论是变量还是函数），就会对 eval(..) 所处的词法作用域进行修改。

- 严格模式下,eval(..) 在运行时有其自己的词法作用域，意味着其中的声明无法修改所在的作用域。

```
function foo(str) {
    "use strict"
    eval(str)
    console.log(a);
}
foo("var a = 2");
// ReferenceError: a is not defined
```

1.4.2 总结

- 词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。编译的词法分析阶段基本能够知道全部标识符在哪里以及是如何声明的，从而能够预测在执行过程中如何对它们进行查找。

1.5 函数作用域

- 函数作用域的含义是指，属于这个函数的全部变量都可以在整个函数范围内使用及复用
- 气泡：在某个作用域下的所有变量

```
function foo(a) {
    var b = 2;

    function bar() {
        // ...
    }
    var c = 3;
}
// 全局作用域气泡：foo
// foo函数作用域气泡：a, b, bar, c
// 所以全局无法访问a, b, bar, c
```

1.5.1 作用1：隐藏内部实现

```
function doSomething(a) {
    b = a + doSomethingElse(a * 2);
    console.log(b * 3);
}
function doSomethingElse(a) {
    return a - 1;
}

var b;
doSomething(2);
// 在这个代码片段中，变量 b 和函数 doSomethingElse(..) 应该是 doSomething(..) 内部具体实现的“私有”内容
// 由于b在外部没有用到，所以最好是作为函数作用域内的变量，而不是放在外面
```

1.5.2 作用2：规避冲突

```
function foo() {
    function bar(a) {
        i = 3;
        console.log(a + i);
    }

    for(var i = 0;i < 10;i ++) {
        bar(i * 2);
    }
}
// 会陷入无限的循环
// bar(..) 内部的赋值表达式 i = 3 意外地覆盖了声明在 foo(..) 内部 for 循环中的 i
// 解决方法：在bar函数内部内部声明一个i变量或者其他变量
// 内部作用域的声明的变量由于嵌套的访问的原因会被优先采用，且不会影响外部同名变量的值
```

1.5.3 作用3：全局命名空间

- 声明一个对象，在对象内部声明变量和函数，该对象就被称为命名空间

1.5.4 函数声明和函数表达式

- 区分函数表达式和函数声明
  - 函数的声明必须用具名函数
  - 有LRS的引用就可以称之为函数表达式，此时可以用匿名函数或者具名函数

```
// 赋值表达式
const a = function() {
    // ...
}
// 函数的参数
function foo(def) {
    def();
}
foo(function() {
    // ...
})
// 立即执行函数表达式
(function(args){
    // ...
})(args)
(function(args) {
    // ...
}(args))
```

1.5.4 立即执行函数表达式

- IIFE(Immediately Invoked Function Expression)

```
// 法1
    (function(args){
        // ...
    })(args)
// 法2
    (function(args) {
        // ...
    }(args))
```

1.6 块作用域

1.6.1 var声明的变量不具有块级作用域

- 在用var声明变量的时候，var并不具有块级作用域

```
    {
        var a = 10;
    }
    console.log(a);
    // 10
    // 函数及作用域
    function foo() {
        var a = 10;
    }
    console.log(a);
    // undefined referenceError
```

1.6.2 try...catch的err具有块级作用域

```
    try{
        // ...
    }
    catch(err) {
        console.log(err)
    }
    console.log(err);
    // undefined referenceError
```

1.6.3 let/const声明的变量具有块级作用域

- let/const具有块级作用域
- 使用 let 进行的声明不会在块作用域中进行提升。声明的代码被运行之前，声明并不“存在”

1.6.3.1 在if语句中

```
    var foo = true;
    if(foo) {
        let bar = foo * 2;
        console.log(bar);
    }
    console.log(bar);
    // 2
    // undefined referenceError
```

1.6.3.2 垃圾回收中

(看书补充)

1.6.3.3 let循环

- for 循环头部的 let 不仅将 i 绑定到了 for 循环的块中，事实上它将其重新绑定到了循环的每一个迭代中，确保使用上一个循环迭代结束时的值重新进行赋值。

```
    for(let i = 0;i < 10;i ++) {
        console.log(i);
    }
    console.log(i)
    // undefined referenceError
```

- for循环的转化理解

```
    {
        let j;
        for(j = 0;j < 10;j ++) {
            let i = j;
            console.log(i);
        }
    }
```

1.6.4 总结

- 函数是 JavaScript 中最常见的作用域单元。本质上，声明在一个函数内部的变量或函数会在所处的作用域中“隐藏”起来，这是有意为之的良好软件的设计原则。
- 块作用域指的是变量和函数不仅可以属于所处的作用域，也可以属于某个代码块（通常指 { .. } 内部）。
- ES3 开始，try/catch 结构在 catch 分句中具有块作用域。
- 在 ES6 中引入了 let 关键字（var 关键字的表亲），用来在任意代码块中声明变量。if(..) { let a = 2; } 会声明一个劫持了 if 的 { .. } 块的变量，并且将变量添加到这个块中。

1.7 声明的提升

- 任何声明在某个作用域内的变量，都将附属于这个作用域。
- 包括变量和函数在内的所有声明都会在任何代码被执行前首先被处理。

1.7.1 编译器对代码的处理

- 声明和代码执行的先后关系

```
    var a = 2;
    // 实际上执行的过程
    var a; // 声明阶段
    a = 2; // 执行阶段
```

- 练习

```
// 练习一
    a = 2;
    var a;
    console.log(a);

    // 等价于:
    var a;
    a = 2;
    console.log(a);
    // 2
```

```
// 练习二
    console.log(a);
    var a = 2;
    // 等价于
    var a;
    console.log(a);
    a = 2;
    // undefine;
    // 2
```

1.7.2 提升

- 当执行的变量在代码执行的时候还没有被声明时,要想到变量提升
- 无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。
- 将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提升

```
foo();
function foo() {
    console.log( a );
    var a = 2;
}
// 等价于
function foo() {
    var a;
    console.log( a );
    a = 2;
}
foo();
// undefined;
```

- 注意
  - 只有声明本身会被提升，而赋值或其他运行逻辑会留在原地。
  - 只有函数声明会被提升,函数表达式无法进行提升, 包含具名表达式也无法进行提升

```
foo();
var foo = function() {
    // ...
}
// TypeError
// 等价于
var foo;
foo();
foo = function() {
    // ...
}
```

```
foo();
bar();
var foo = function bar() {
    // ...
}
// TypeError
// ReferenceError 没有声明bar()
```

1.7.3 函数声明和变量声明的优先级

- 变量和函数声明同时提升时, 函数声明优先级比较高。

```
foo();
var foo = 2;
function foo() {
    console.log(1);
}
// 1
```

1.7.4 函数声明不具有块级作用域

- 一个普通块内部的函数声明通常会被提升到所在作用域的顶部
  - 函数的声明并不具有块级作用域的作用

```
foo();
var a = true;
if(a) {
    function foo() {
        console.log( "a" );
    }
}
else{
    function foo() {
        console.log( "b" );
    }
}
// b
```

1.7.5 总结

- 我们习惯将 var a = 2; 看作一个声明，而实际上 JavaScript 引擎并不这么认为。它将 var a和 a = 2 当作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。
- 无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提升。
- 声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升
- 要注意避免重复声明，特别是当普通的 var 声明和函数声明混合在一起的时候，否则会引起很多危险的问题！

1.8 作用域的闭包

- 某个函数在其他作用域(不是本身的词法作用域)被调用时,仍然可以访问其词法作用域内的变量(相应变量的值被缓存了下来),即仍然可以引用其所在的词法作用域,该词法作用域就称为闭包.

1.8.1 理解闭包

- 非闭包

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    bar();
}

foo();
```

- 由于bar在foo作用域中被执行,不存在闭包的效果

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    return bar;
}

var baz = foo();
baz(); // 根据调用的位置，该baz应该访问的是baz内部定义的变量，但是它仍可以访问foo内部的变量
```

- 由于bar在foo中定义,即bar是foo的作用域气泡,即bar正常情况下只能在foo的作用域范围内被执行
- 但是,在该例子中bar在自己定义的词法作用域以外的地方执行。
  - 原因: 由于闭包的原因,bar仍然可以访问它定义时的作用域

1.8.2 闭包的实质

- foo() 执行后，通常会期待 foo() 的整个内部作用域都被销毁, 但闭包阻止了该内部作用域被销毁, 即该作用域仍然被bar所使用, 即bar() 依然持有对该作用域的引用，而这个引用就叫作闭包。
- 某个函数被执行的时候，本来只能访问该函数本身的作用域内部的声明的变量或者全部变量，但是实质上，其可以访问声明位置的作用域内部的变量（声明位置的作用域会被缓存起来）
- 闭包起效果的条件是，该函数声明位置的函数又被执行

1.8.3 闭包的通常形式

- 通过return返回对应的函数

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    return bar;
}

var baz = foo();
baz();
// 2
```

- 通过参数传递的形式

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    baz(bar);
}
function baz(fn) {
    fn();
}
foo();
// 2
// bar在baz函数作用域执行,仍然调用其词法作用域的内容
function wait(args) {
    setTimeout(function timer() {
    // ...有用到args
    }, 1000)
}
// 在其他作用域调用了timer函数,但仍然可以访问到args
```

- 通过外部变量传递

```
var baz;
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    baz = bar;
}
foo();
baz();
// 2
// bar在全局作用域执行,仍然调用其词法作用域的内容
```

1.8.4 闭包的运用

1.8.4.1 循环和闭包

```
    for(var i = 0;i <= 5;i ++) {
        setTimeout(function() {
            console.log(i);
        }, 1000)
    }
//  6 6 6 6 6 6
```

- 原因:在执行完同步代码之后,此时的i为6,而异步执行的setTimeout的回调函数满足条件后被调用时,显然是输出6
- 解决方法
  - 法1: 运用立即函数作用域的闭包给每个循环创建一个新的作用域
  - 法2: 利用块级作用域

```
    for(var i = 0;i <= 5;i ++) {
        (function() {
            var j = i;
            setTimeout(function() {
            console.log(j);
            }, 1000)
        })()
    }
```

```
    for(var i = 0;i <= 5;i ++) {
        (function(j) {
            setTimeout(function() {
            console.log(j);
            }, 1000)
        })(i)
    }
```

```
    for(let i = 0;i <= 5;i ++) {
        setTimeout(function() {
            console.log(i);
        }, 1000)
    }
```

1.8.4.2 实现模块

- 必须有外部的封闭函数，该函数必须至少被调用一次（每次调用都会建一个新的模块实例）。
- 封闭函数必须返回至少一个内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态。
- 模块的基本形式

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];
    function doSomething() {
        console.log(something);
    }
    function doAnother() {
        console.log(another);
    }
    return {
        doSomething: doSomething,
        doAnother: doAnother
    }
}

var CoolModule = foo();
CoolModule.doSomething();
CoolModule.doAnther();
```

- doSomething() 和 doAnother() 函数具有涵盖模块实例内部作用域的闭包
- IIFE模块

```
var args = undefined;
var module1 = (function(args) {
    var something = "cool";
    var another = [1, 2, 3];
    function doSomething() {
        console.log(something);
    }
    function doAnother() {
        console.log(another);
    }
    return {
        doSomething: doSomething,
        doAnother: doAnother
    }
})(args)
module1.doSomething();
module1.doAnother();
```

1.8.5 总结

- 当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。
- 模块有两个主要特征：
  - （1）为创建内部作用域而调用了一个包装函数；
  - （2）包装函数的返回值必须至少包括一个对内部函数的引用，这样就会创建涵盖整个包装函数内部作用域的闭包。

2 this的理解

- this是在运行时进行绑定,而不是编写时绑定,它的上下文取决于函数调用时的各种条件.
- this的绑定和函数声明的位置没有任何关系,只取决于函数的调用方式

2.1 this的实质

- 当一个函数被调用时,会创建一个活动记录(有时候被称为执行上下文).这个记录会包含在调用栈中,函数调用栈中有函数的调用方法,传入的参数等信息.this就是记录其中一个属性,会在函数执行的过程中用到.

2.2 确定this所指的对象的方法

2.2.1 第一步：寻找调用位置和当前的调用栈

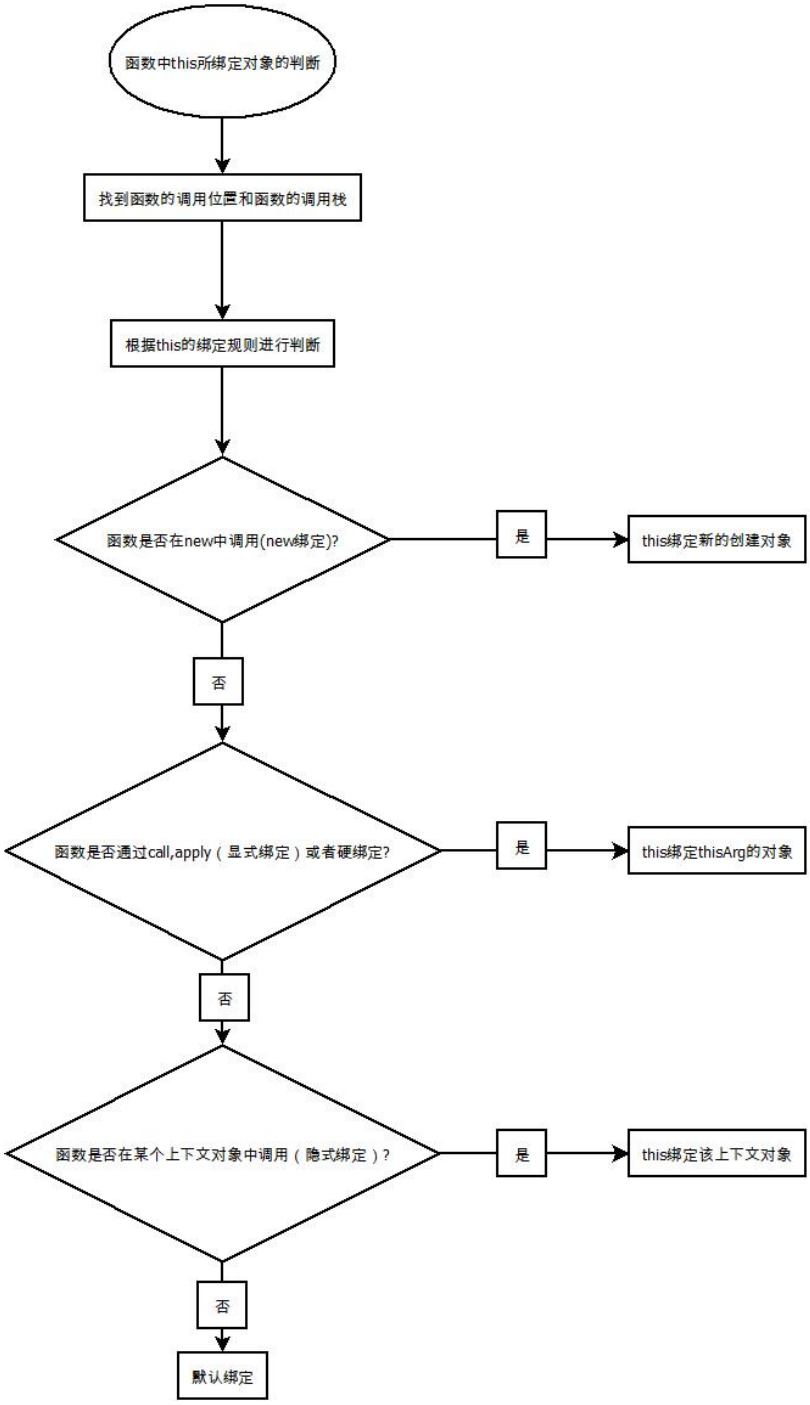
- 调用位置:函数在代码中被调用的位置

```
function baz() {
    /*
     * 第二步
     * 当前的调用栈是:baz
     * 当前的调用位置是全局作用域
    */
    console.log("baz");
    bar(); // 第三步: bar的调用位置
}
function bar() {
    /*
     * 第四步
     * 当前的调用栈是:baz -> bar
     * 当前的调用位置是baz中
    */
    console.log("bar");
    foo(); // 第五步: foo的调用位置
}
function foo() {
    /*
     * 第六步
     * 当前的调用栈是:baz -> bar -> foo
     * 当前的调用位置是bar中
    */
    console.log("foo");
}

baz();  // 第一步:baz的调用位置
```

- 调用栈的倒数第二个即为调用位置

2.2.2 第二步：根据this的绑定规则来确定绑定的对象

- this的绑定规则
  - 函数是否在 new 中调用（new 绑定）？如果是的话 this 绑定的是新创建的对象。
    - var bar = new foo()
  - 函数是否通过 call、apply（显式绑定）或者硬绑定调用？如果是的话，this 绑定的是指定的对象。
    - var bar = foo.call(obj2)
  - 函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this 绑定的是那个上下文对象。
    - var bar = obj1.foo()
  - 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 undefined，否则绑定到全局对象。
    - var bar = foo()



2.2.2.1 默认绑定

- 默认this指向全局对象(window)
- 常见于独立函数的调用
  - 直接使用不带任何修饰的函数引用进行调用的，因此只能使用默认绑定

```
function foo() {
    console.log(this.a);
}
var a = 2;
foo();

// 2
```

- 注意
  - 在严格模式下,this是没有指向全局对象的,而是绑定到undefined

```
function foo() {
    "use strict";
    console.log(this.a);
}
var a = 2;
foo();
// TypeError: this is undefined
```

2.2.2.2 隐式绑定

- 是调用位置是否有上下文对象，或者说是否被某个对象拥有或者包含，隐式绑定规则会把函数调用中的 this 绑定到这个上下文对象
- 指向调用的对象

```
function foo() {
    console.log(this.a);
}
var obj2 = {
    a: 42,
    foo: foo
}
var obj1 = {
    a:2,
    obj2:obj2
}
obj1.obj2.foo()
// 42
```

2.2.2.2.1 隐式丢失

- 使用函数别名进行调用
  - 虽然 bar 是 obj.foo 的一个引用，但是实际上，它引用的是 foo 函数本身，因此此时的bar() 其实是一个不带任何修饰的函数调用，因此应用了默认绑定。

```
// 相当于这种形式
    function foo() {
        console.log(this === window);
        console.log(this.a);
    }
    var obj1 = {
        a: 2,
        foo: foo
    }
    var a = "hello world"
    var bar = obj1.foo;
    bar();
    //true "hello world"
```

```
var obj1 = {
    a: 2,
    foo: function() {
        console.log(this === window);
        console.log(this.a);
    }
}
var a = "hello world"
var bar = obj1.foo;
bar();
//true "hello world"
```

- 传入回调函数的参数
  - 参数传递其实就是一种隐式赋值，因此我们传入函数时也会被隐式赋值，与上面的结果一样

```
function foo() {
    console.log(this === window);
    console.log(this.a);
}
var obj1 = {
    a: 2,
    foo: foo
}
function doFunc(fn) {
    fn();
}
var a = "hello world"
doFunc(obj1.foo);
//true "hello world"
// setTimeout的回调函数也有类似的作用
```

2.2.2.3 显示绑定

- 方法1：使用call, apply进行显示调用
- 方法2：使用bind进行硬绑定
  - 使得obj1中的this指向foo的this,
    - 若foo中有this的属性,方法,则都变成了obj1的方法
    - foo的this是obj1的this
  - 解决使用函数别名进行调用
- 方法3：API调用的上下文
  - 第三方库的许多函数，以及 JavaScript 语言和宿主环境中许多新的内置函数，都提供了一个可选的参数，通常被称为“上下文”（context），其作用和 bind(..) 一样，确保你的回调函数使用指定的 this

```
function foo() {
    console.log(this === window);
    console.log(this.a);
}
var obj1 = {
    a: 2,
    foo: foo
}
var a = "hello world"
var bar = foo.bind(obj1);
bar();
// false 2
```

```
var obj = {
    id: "awesome"
};
[1, 2, 3].forEach( function(el) {
    console.log( el, this.id );
}, obj );
/*
1 awesome
2 awesome
3 awesome
*/
```

2.2.2.4 new绑定

- 四步走
  - 创建一个新的空的对象
  - 这个新对象会执行原型链接
    - obj.**proto** = Object.create(funcName.prototype)
  - 这个新的对象会绑定到函数调用的this
  - 如果函数没有返回其他对象，那么表达式中的函数调用会自动返回这个新的对象

2.3 this的绑定例外

2.3.1 被忽略的this

- 如果你把 null 或者 undefined 作为 this 的绑定对象传入 call、apply 或者 bind，这些值在调用时会被忽略，实际应用的是默认绑定规则
- 在使用call,apply,bind有时候只需要运用传输参数,此时的this应该指向一个空的对象,使用以下的方法更加安全
  - 创建一个“DMZ”（demilitarized-zone，非军事区）对象——它就是一个空的非委托的对象

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}
var ø = Object.create(null);
// 把数组展开成参数
foo.apply( ø, [2, 3] ); // a:2, b:3
// 使用 bind(..) 进行柯里化
var bar = foo.bind( ø, 2 );
ar( 3 );
```

2.3.2 间接引用(包括函数参数的传递)

- 创建一个函数的“间接引用”，在这种情况下，调用这个函数会应用默认绑定规则。

```
function foo() {
    console.log(this.a);
}
var a = 2;
var o = {
    a: 3,
    foo: foo
}
var p = {
    a: 4
}
o.foo()
(p.foo = o.foo) ()
// 间接引用是默认绑定
// 3
// 2
```

```
const obj = {
  foo() {
    console.log(this);
  }
}
function baz(func){
  func();
}
obj.foo();
baz(obj.foo);
// obj
// window
```

- 解决方法
  - 运用硬绑定
  - 运用软绑定

2.3.3 软绑定(obj.softBind(..))

- 硬绑定会大大降低函数的灵活性，使用硬绑定之后就无法使用隐式绑定或者显式绑定来修改 this
- 可以给默认绑定指定一个全局对象和 undefined 以外的值，那就可以实现和硬绑定相同的效果，同时保留隐式绑定或者显式绑定修改 this 的能力。
- obj.softBind(..)
  - 会对指定的函数进行封装，首先检查调用时的 this，如果 this 绑定到全局对象或者 undefined，那就把指定的默认对象 obj 绑定到 this，否则不会修改 this。

```
function foo() {
    console.log("name: " + this.name);
}
var obj = { name: "obj" },
obj2 = { name: "obj2" },
obj3 = { name: "obj3" };
var fooOBJ = foo.softBind( obj );
fooOBJ(); // name: obj
obj2.foo = foo.softBind(obj);
// 此时为默认绑定
obj2.foo(); // name: obj2 <---- 看！！！
fooOBJ.call( obj3 ); // name: obj3 <---- 看!
setTimeout( obj2.foo, 10 );
// name: obj <---- 应用了软绑定
```

2.3.4 箭头函数

- 箭头函数会继承外层函数调用的 this 绑定（无论 this 绑定到什么）

2.4 总结

- 如果要判断一个运行中函数的 this 绑定，就需要找到这个函数的直接调用位置,然后再运用规则
- this的规则

1. 由 new 调用？绑定到新创建的对象。
2. 由 call 或者 apply（或者 bind）调用？绑定到指定的对象。
3. 由上下文对象调用？绑定到那个上下文对象。
4. 默认：在严格模式下绑定到 undefined，否则绑定到全局对象。

- 一定要注意，有些调用可能在无意中使用默认绑定规则。如果想“更安全”地忽略 this 绑定，你可以使用一个 DMZ 对象，比如 ø = Object.create(null)，以保护全局对象。
- ES6 中的箭头函数并不会使用四条标准的绑定规则，而是根据当前的词法作用域来决定this，具体来说，箭头函数会继承外层函数调用的 this 绑定（无论 this 绑定到什么）

3 有关类的三种不同设计模式

3.1 使用构造函数的模式

```
function Father(name) {
    this.name = name;
}
Father.prototype.getName = function() {
    return this.name;
}

// 继承
function Son(name, label) {
    this.label = label;
    Father.call(this, name);
}
Son.prototype = Object.create(Father.prototype);
Son.prototype.getLabel = function() {
    return this.label;
}
Son.prototype.printAll = function() {
    console.log(this.getName(), this.label)
}

var son = new Son("hello world", 2);
son.printAll();
console.log(son.getName());
console.dir(son);
```

3.2 使用class模式

```
class Father {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}
class Son extends Father {
    constructor(name, label) {
     super(name);
     // super指的是同名的父类函数
        this.label = label;
    }
    getLabel() {
        return this.label;
    }
    printAll() {
        console.log(this.getName(), this.label)
    }
}
var son = new Son("hello world", 2);
son.printAll();
console.log(son.getName());
console.dir(son)
```

3.3 使用事件委派的模式

```
var Father = {
    fatherInit(name) {
        this.name = name;
    },
    getName() {
        return this.name;
    }
}
var Son = Object.create(Father);
Son.sonInit = function(name, label) {
    this.label = label;
    this.fatherInit(name);
}
Son.getLabel = function() {
    return this.label;
}
Son.printAll = function() {
    console.log(this.getName(), this.label);
}
var son = Object.create(Son);
son.sonInit("hello world", 2)
son.printAll();
console.log(son.getName());
```

```
    var Father = {
        fatherInit(name) {
            this.name = name;
        },
        getName() {
            return this.name;
        }
    }
    var Son = {
        sonInit(name, label) {
            this.label = label;
            this.fatherInit(name);
        },
        getLabel() {
            return this.label;
        },
        printAll() {
            console.log(this.getName(), this.label);
        }
    }
    Object.setPrototypeOf(Son, Father)
    var son = Object.create(Son);
    son.sonInit("hello world", 2)
    son.printAll();
    console.log(son.getName());
```

```
     // new创建实例的本质
     function fun(name) {
      this.name = name;
     }
     // 第一步：创建一个原型指向构造函数原型对象的对象
     var obj = Object.create(fun.prototype);
     // 第二步：将构造函数的this指向该原型对象
        // 由于此时执行了一次构造函数，所以会创建一个新的作用域，该作用域中的this指向该原型对象
    fun.call(obj, "hello");
    // 第三步：
    console.log(obj)

    var newObj = new fun("hello");
    console.log(newObj)
    // 继承
    function Father(name) {
     this.name = name;
    }
    function Son(name, label) {
     this.label = label;
     Father.call(this, name);
     // 继承Father的this指向的属性
    }
    // 子类的原型对象是指的原型是指向父类原型的对象
    Son.prototype = Object.create(Father.prototype)
```

4 语法

4.1 语句和表达式

- “句子”（sentence）是完整表达某个意思的一组词，由一个或多个“短语”（phrase）组成， 它们之间由标点符号或连接词（and 和 or 等）连接起来。短语可以由更小的短语组成。有 些短语是不完整的，不能独立表达意思；有些短语则相对完整，并且能够独立表达某个意 思。
  - 语句相当于句子，表达式相当于短语，运算符则相当于标点符号和连接词。

```
    var a = 3 * 6;
    var b = a;
    b;
    /*
    *  这三行代码都是包含表达式的语句。var a = 3 * 6 和 var b = a 称为“声明语句”
（declaration statement），因为它们声明了变量（还可以为其赋值）。
    *   a = 3 * 6 和 b = a（不带 var）叫作“赋值表达式”。
    *   b既是语句有时表达式，称为表达式语句
    */
```

4.1.1 语句的结果值的隐式返回

- 每一个语句都默认有返回值，一般为undefine
  - 赋值语句(赋值表达式）的返回值为被赋值的数的值
- 对于代码块，也默认有返回值
  - 最后一个执行语句的返回值
  - 但是由于语法规则规定不允许我们获得语句的结果值并将其赋值给另一个变量
  - ES7规定可以使用do关键词实现

```
var a;
console.log(a = 42);
// 42

console.log(var b = 42);
// undefined

var c = do {
    if(false) {
        var d;
        d = 100;
    }
}
console.log(c)
```

五 js的DOM操作

1. DOM(document Object Model)

- 文档对象模型(DOM树)
- document
  - 表示整个html网页文档
- object
  - 表示将文档的每一个部分转化为一个对象
- model
  - 使用模型来表示对象之间的关系，以方便获取对象
    - 使用的是树的模型

1.2. DOM树

- DOM把一个文档表示为一棵家谱树，树是由节点构成的

1.3. 节点(node)

- 构成网页的基本元素

1.3.1. 节点的分类

1.3.1.1. 文档节点(document)

- 浏览器已经为我们提供了文档节点，可以在页面中直接使用
- 文档节点代表的是整个网页

1.3.1.1. 元素节点

- 元素标签

1.3.1.1. 属性节点

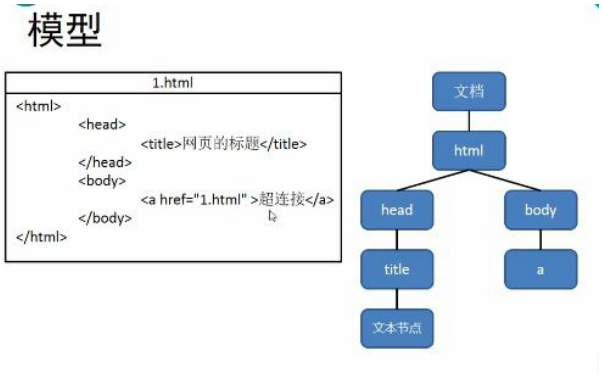
- 元素的属性

1.3.1.1. 文本节点

- 标签内部的文本内容

1.3.2. 节点的属性

- DOM定义了node的接口以及许多中节点类型来表示XML几点的多个方面



2 DOM方法

2.1 DOM document属性和方法

2.1.1 document获取元素对象的方法

2.1.1.1 document.getElementsByClassName(names)

- 返回类名names元素对象的类数组
- 参数
  - names：类名，是一个字符串
- ie8及以下不支持

2.1.1.2 document.getElementsByTagName(tagNames)

- 返回类名tagNames元素对象的类数组
- 参数
  - tagNames：标签名，是一个字符串

2.1.1.3 document.getElementById(id)

- 返回一个id的元素对象
- 参数
  - id：id名，是一个字符串

2.1.1.4 document.getElementsByName(names)

- 返回属性name值为names元素对象的类数组
- 参数
  - names：name属性的值，是一个字符串

2.1.1.5 document.querySelector(cssSelector)

- 返回与指定的选择器组匹配的元素的后代的第一个元素对象。

2.1.1.6 document.querySelectorAll(cssSelector)

- 返回与指定的选择器组匹配的元素的后代的所有元素对象组成的类数组。

2.1.2 document常用的属性

2.1.2.1 document.body

- 返回标签名为body的元素对象

2.1.2.2 document.all

- 以类数组的形式返回整个文档的元素对象

2.1.2.3 document.doucumentElement

- 返回标签名为html的元素对象

2.1.3 doucment与增加元素对象相关的属性

2.1.3.1 document.createElement(tagName)

- 创建一个标签名为tagName的元素对象

2.2 DOM Element方法

2.2.1 element获取元素对象的方法

2.2.1.1 element.getElementsByClassName(names)

- 返回类名names元素对象的类数组

- 参数
  - names：类名，是一个字符串

2.2.1.2 element.getElementsByTagName(tagNames)

- 返回类名tagNames元素对象的类数组
- 参数
  - tagNames：标签名，是一个字符串

2.1.1.3 element.querySelector(cssSelector)

- 返回与指定的选择器组匹配的元素的后代的第一个元素对象。
- IE8及以下浏览器不支持

2.1.1.4 element.matches(cssSelector)

- 判断当前元素对象是否与cssSelector匹配

2.1.1.4 element.querySelectorAll(cssSelector)

- 返回与指定的选择器组匹配的元素的后代的所有元素对象组成的类数组。
- IE8及以下浏览器不支持

2.2.2 element与父元素对象，子元素对象，兄弟元素对象有关的属性和方法

2.2.2.1 element.parentNode

2.2.2.1.1 parentNode.childElementCount

- 返回一个当前 ParentNode 所含有的后代数量。

2.2.2.1.2 parentNode.children

- 返回一个包含 ParentNode 所有后代元素对象的类数组，忽略所有非元素子节点。

2.2.2.1.3 parentNode.firstElementChild

- 返回第一个后代元素对象
- IE8及以下浏览器不支持

2.2.2.1.4 parentNode.lastElementChild

- 返回最后一个后代元素对象
- IE8及以下浏览器不支持

2.2.2.1.5 parentNode.append(...node)

- 在子元素对象的最后插入一个节点
- 参数
  - node
    - 元素对象

2.2.2.1.5 parentNode.prepend(node)

- 在子元素对象的最钱买你插入一个节点
- 参数
  - node
    - 元素对象

2.2.2.2 element.childNode

2.2.2.2.1 childNode.remove()

- 删除当前元素对象

2.2.2.3 element.previousElementSibling（只读）

- 该元素的上一个兄弟元素对象，若返回null，则说明不存在

2.2.2.4 element.nextElementSibling（只读）

- 该元素的下一个兄弟元素对象，若返回null，则说明不存在

2.2.2.5 element.closest(selectors)

- 获取离当前元素对象最近的具有能够匹配selectors的祖先元素

2.2.3 element与增加元素对象相关的属性和方法

2.2.3.1 parentNode.append(...node)

- 在子元素对象的最后插入一个节点
- 参数

    - node
        - 元素对象

2.2.3.2 parentNode.prepend(node)

- 在子元素对象的最钱买你插入一个节点
- 参数
    - node
        - 元素对象

2.2.3.3 parentNode.replaceChildren(newChild, oldChild)

- 用新的子类对象来代替旧的子类对象

2.2.3.4 childNode.replaceWith(...node)

- 用node来代替当前的childNode

2.2.3.5 childNode.remove()

- 删除当前的childNode元素对象

2.2.3.6 element.insertAdjacentElement(position, newElement)

- 将newElement插入相当于当前元素对象某个位置
- 参数
    - position（理解：begin可以看成当前元素对象的开始标签，end可以看成当前元素对戏那的结束标签）
        - "beforebegin": 在当前元素对象本身的前面
        - "afterbegin": 在当前元素对象中，第一个子元素对象的前面
        - "beforeend": 在当前元素对象中，最后一个子元素对象的后面
        - "afterend": 在当前元素对象本身的后面
    - newElement
        - 新插入的元素

2.2.3.6 element.insertAdjacentHTML(position, text)

- 将text解析成一个新的元素对象，将该对象插入相当于当前元素对象某个位置
- 参数
    - position（理解：begin可以看成当前元素对象的开始标签，end可以看成当前元素对戏那的结束标签）
        - "beforebegin": 在当前元素对象本身的前面
        - "afterbegin": 在当前元素对象中，第一个子元素对象的前面
        - "beforeend": 在当前元素对象中，最后一个子元素对象的后面
        - "afterend": 在当前元素对象本身的后面
    - text
        - 以字符串形式表示的HTML

2.2.4 element与元素属性相关的属性及方法

2.2.4.1 element.nodeName/tagName

2.2.4.2 element.id

2.2.4.3 element.attributes（只读）

- 返回一个与该元素相关的所有属性的类数组

2.2.4.4 element.innerHTML

- 仅获取元素内容的HTML表示形式或替换元素的内容

2.2.4.5 element.outerHTML

- 获取描述元素（包括其后代）的序列化HTML片段。（包括自身的标签）

2.2.4.1.1 用于替换当前的元素对象

2.2.4.6 element.innerText

- 返回当前元素及其后代的“渲染”文本内容
    - 显示最终渲染完的内容，display:none无法获取

2.2.4.7 element.className

- 返回标签的类名值，以空格的形式分开

2.2.4.8 element.classList（只读）

- 相比将 element.className 作为以空格分隔的字符串来使用，classList 是一种更方便的访问元素的类列表的方法。
- 虽然是只读属性，但是可以使用以下的方法来进行修改

2.2.4.3.1 classList.add(...classValue)

- 类值增加

2.2.4.3.2 classList.remove(...classValue)

- 类值删除

2.2.4.3.3 classList.toggle(classValue)

- 类值的切换
  - 有则删去
  - 没有则添加

2.2.4.9 element.getAttribute(attrName:string)（只读）

- 返回当前元素指定属性名的值

2.2.4.10 element.getAttributeNames()

- 以数组的形式返回所有具有的属性名称
- 将 getAttributeNames() 与 getAttribute() 组合使用， 是一种有效替代 Element.attributes 的使用方法.

```
// 遍历elements的元素
for(let name of element.getAttributeNames())
{
  let value = element.getAttribute(name);
  console.log(name, value);
}
```

2.2.4.11 element.hasAttribute(attrName:string)

- 判断是否具有attrName属性，返回一个布尔值

2.2.4.11 element.removeAttribute(attrName:string)

- 删去attrName属性，不存在也不会报错

2.2.4.12 element.setAttribute(attrName:string, attrValue:string)

- 设置当前属性的值

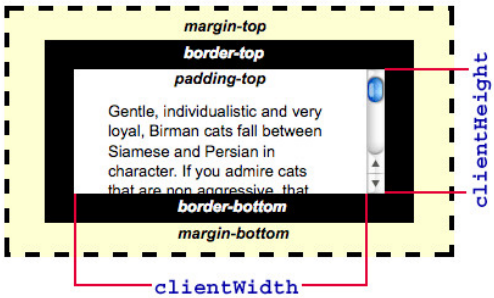
2.2.4.13 element.toggleAttribute(attrName[, force])

- 存在某个属性则增加，不存在则删除

2.2.5 element与样式相关的属性和方法

2.2.5.1 element.client[Height, Width]（只读）

- 该元素对象的宽度/高度，返回的是number类型



  - 组成包括content和padding

2.2.5.2 element.client[Left, Top]（只读）

- element为非静态元素对象，获取其Left和Top的偏移量
  - 子: 以border为边界
  - 父: 以content为边界

2.2.5.3 element.scroll[Height, Width]（只读）

- 一个元素内容高度的度量，包括由于溢出导致的视图中不可见内容。
  - 若没有滚动条，则值与element.client[Height, Width]相同
- 包括content, padding, ::before, ::after

2.2.5.4 element.scroll[Left, Top]

- 读取或设置当前元素滚动条到元素左边/上边的距离
  - 如果元素不能滚动（比如：元素没有溢出），那么scroll[Left, Top] 的值是0。
  - 如果给scroll[Left, Top] 设置的值小于0，那么scroll[Left, Top] 的值将变为0。
  - 如果给scroll[Left, Top] 设置的值大于元素内容最大宽度，那么scroll[Left, Top]的值将被设为元素最大宽度。

2.2.5.5 element.[attName] （获取内联样式属性）

- 以字符串的形式返回attrName的属性值
  - 导致相应的单位也会返回

2.2.5.5.1 element.value

- 一般用于获取或设置input的value

2.2.5.6 element.style.[attrName]

- 用于设置内联样式的style属性
  - 若在样式表中有以-连接，则应该采取驼峰的形式
  - 注意是有单位的字符串
- 该样式的设置优先级为内联样式的优先级

2.2.5.6.1 element.style.backgroundColor

- 用于设置背景的颜色

2.2.5.7 实现判断滚动条是否到底

```
// 当垂直滚动条滚动到底的时候有
element.clientHeight === element.scrollHeight - element.scrollTop
```

2.2.4.10

2.3 DOM的版本适应的问题

6.1.2.3. 获取元素节点对象的行内样式属性值

- 元素节点对象.属性
- class属性不能使用这种方式

6.1.2.3.3. 获取元素节点对象的样式表中的样式

6.1.2.3.3.1. 由于兼容性，需要自定义一个函数

```
function getStyle(obj, name) {
  // 判断浏览器版本太麻烦，用什么进行判断？
  // 正常浏览器
  if(window.getComputedStyle)
    return getComputedStyle(obj, null)[name];
  else
    return obj.currentStyle[name];
  // ie8浏览器Style

}
```

6.1.2.3.3.2. getComputedStyle(elementObject, pseudoElements)

- window方法
- pseudoElements = null
  - 返回了存储elementObject当前的样式的对象
- pseudoElements 伪元素
  - 返回了存储elementObject伪元素当前的样式的对象

6.1.2.3.3.3. objectElement.currentStyle.样式名

- 获取当前元素对象显示的样式而不仅仅是内联或样式
  - 只能ie
- 区别：ie返回的auto，则上面一个放回的是具体的数值
- 兼容ie8

6.1.2.4.1. 以父元素为对象的操作

6.1.2.4.1.1. document.createElement(tagStr)

- 创建元素的标签对象

6.1.2.4.1.2. document.createTextNode(str)

- 创建并初始化文本节点对象

6.1.2.4.1.3. fatherNodeObject.appendChild(childNodeObject)

- 向父节点对象内部的最后面增加子节点对象
- 父节点为对象

6.1.2.4.1.4. fatherNodeObject.insertBefore(newchildObject, oldchildObject)

- 向旧的子节点对象的前面添加新的子节点对象
- 父节点为对象

6.1.2.4.1.5. fatherNodeObject.replaceChild(newchildObject, oldchildObject)

- 替换旧的子节点对象为新的子节点对象

6.1.2.4.1.6. fatherNodeObject.removeChild(childObject)

- 删除子节点队形

6.1.2.4.1.7. 元素对象的增加的步骤（使用createElement）

```
/*
  1. 创建子元素对象
  2. 给子元素对象赋值
  3. 添加到父元素对象中
*/
let childElement = createElement('div');

let text = createTextNode('hello_world');
childElement.appendChild(text);

fatherElement.appendChild(childElement);
```

6.1.2.4.1.8. 使用innerHTML对元素进行增加

- 缺点:会把整一块替换

```
fatherElement.innerHTML += childHTML
```

6.1.2.4.1.9. 两种方式结合对元素进行添加(推荐)

```
let childElement = createElement('div');
childElement.innerHTML = 'hello_world'  //一般是比较固定的格式
fatherElement.appendChild(childElement);
```

6.1.2.4.1.10. 元素对象的删除的步骤（经常使用的）

```
// 找到父亲节点之后对其进行删除
  childObject.parentNode.removeChild(childObject);
```

3 DOM事件

- 文档或浏览器窗口中发生的一些特定交互的瞬间

3.0 事件的一些基本常识

3.0.1 事件的公共属性和方法

3.0.1.1 event.target

- event.target返回的是一个元素对象
  - 若该事件不是冒泡事件，则event.target会捕获到当前绑定的元素对象
  - 若该事件是冒泡事件，则event.target会捕获到最底层的触发该事件的元素对象

3.0.1.2 event.currentTarget

- 与是不是冒泡事件无关，只是返回当前触发的元素

3.0.1.3 event.bubbles

- 表明当前事件是否会向DOM树上层元素冒泡
  - event.stopPropagation()

3.0.1.4 event.cancelable

- 表明该事件是否可以被取消
  - event.preventDefault()

3.0.1.5 event.type

3.0.1.6 event.stopPropagation()

- 以前是用event.cancelBubble = true,现在用这个进行代替

- 阻止捕获和冒泡阶段中在该元素的当前事件中的进一步传播。

3.0.1.7 event.preventDefault()

- 取消当前元素触发事件的默认行为
  - 在on[eventName]的表达式形式定义的回调函数中，可以使用return false来取消事件的默认行为

```
elementObject.onclick = function() {
  ...
  return false;
}
```

3.0.2 事件与元素的绑定和取消的方法方法

3.0.2.0 在内联样式中增加属性on[eventName] = "callback()"

3.0.2.1 element.on[eventName] = function() {}

- 表达式的形式的事件绑定
- 缺点：
  - 只能同时为一个元素的一个事件绑定一个响应函数
  - 不能绑定多个，若绑定了多个，则后面的会覆盖前面的

3.0.2.2 element.on[eventName] = null

- 表达式形式的事件取消绑定

3.0.2.3 element.addEventListener(eventName:string, func, [true || false])

- 参数
  - 事件的字符串，不要on
  - 回调函数，当事件触发
  - 是否在捕获阶段触发事件，需要一个布尔值，默认是false
- 优点
  - 可以同时为一个元素对象的同一个事件绑定多个响应函数
    - 当事件被触发时，响应函数会按照绑定的顺序进行响应
- 缺点
  - 这种方法不支持ie8及以下

3.0.2.4 element.removeEventListener(eventName:string, func, [true || false])

3.0.2.5 element.attachEvent(on[eventName]:string, func)

- 适用于ie8及以下浏览器
- 参数
  - 事件的字符串，要on
  - 回调函数，当事件触发
- 优点
  - 可以同时为一个元素对象的同一个事件绑定多个响应函数
    - 当事件被触发时，响应函数会按照绑定的顺序的倒序进行响应（后绑定先执行）

3.0.2.5 通用的事件绑定函数

```
/*
* 参数
* 1. obj 要绑定的事件对象
* 2. eventName: 事件名称
* 3. callback: 回调函数
*/
// addEventListener()中的callback函数中的this，是绑定事件对象
// attachEvent()中的allback函数中的this是window， 所以需要修改callback的this对象
function bind(obj, eventName, callback){
  if(obj.addEventListener)
    obj.addEventListener( eventName, callback, false);
  else
    obj.attachEvent('on' + eventName,function() {
    callback.call(obj); //将call的this指向obj
  });
}
// callback.bind(obj)
```

3.0.3 事件的传播

- 事件的传播途径有两种
  - 以非冒泡的形式实现事件的传播
  - 以冒泡的形式实现事件的传播

3.0.3.1 非冒泡事件

- 假设当前绑定事件的元素有子元素，且此时事件触发

- 非冒泡事件执行
  - 并不会向其子元素传播触发的事件
  - 该元素触发了事件

3.0.3.2 冒泡事件

- 假设当前绑定事件的元素有子元素，且此时事件触发
- 冒泡事件的执行过程
  - 该元素检查是否触发事件的回调函数
  - 向其子元素传递该事件，导致其子元素都会检查是否触发事件的回调函数（捕获事件）
  - 执行一次该事件触发的回调函数
    - 若子元素满足回调函数的条件，则执行该子元素的事件触发回调函数，其父元素的回调函数不能再执行
    - 若子元素都不满足回调函数的条件，则执行该元素的回调函数

3.0.3.2.1 冒泡事件的传播

- ie8及以下没有捕获阶段
- 将冒泡事件分成三个阶段
  - 捕获阶段
    - 在捕获阶段时，从外层的祖先元素，向目标元素进行事件捕获，但是默认此时不会触发事件
  - 目标阶段
    - 捕获到目标元素，捕获结束，开始在目标元素上触发事件
  - 冒泡阶段
    - 事件从目标元素向它的祖先元素传递，只会触发一次相应的事件

	nodeName	nodeType	nodeValue
文档节点	#document	9	null
元素节点	标签名	1	null
属性节点	属性名	2	属性值
文本节点	#text	3	★文本内容

- addEventListener的第三个参数
  - 默认值为false
    - 即事件的回调函数在冒泡阶段才会被触发，即子元素触发优先于祖先元素触发
  - 若设置为true
    - 即事件的回调函数在捕获阶段被触发，即祖先元素触发优先于子元素的触发

3.0.3.2.2 event.stopPropagation()的应用

- 阻止捕获和冒泡阶段中在该元素的当前事件中的进一步传播。
- 解释
  - 若其父元素触发的事件是一个冒泡事件，当该冒泡事件被触发时，会向子元素传播
  - 在向子元素传播时，会根据子元素触发事件中是否有event.stopPropagation()以及是否触发了相同事件决定是否往下传播
    - 若存在，则不传播且不触发任何回调函数
    - 若不存在，则继续往下传播，直至底层再冒泡起来执行
- 使用event.stopPropagation()实现冒泡事件click父元素后不向子元素传播
- 例子见下面

3.0.3.2.3 利用冒泡事件实现事件委派

- 指将冒泡事件统一绑定给元素的共同祖先，当冒泡事件被触发，会通过事件冒泡传递给子元素且触发相应的回调函数
  - 减少事件的绑定从而提高性能

3.1 剪贴版事件（ClipboardEvent）

事件名称	事件基本信息	事件触发条件
copy	Bubbles: Yes Cancelable: Yes Target: 获得焦点的元素（即是能够编辑内容的元素） 返回的接口类型(eventType):keyboardEvent	当用户通过浏览器UI（例如，使用 Ctrl/⌘+C 键盘快捷方式或从菜单中选择“复制”）
cut	Bubbles: Yes Cancelable: Yes Target: element 返回的接口类型(eventType):keyboardEvent	在将选中内容从文档中删除并将其添加到剪贴板后触发。
paste	Bubbles: Yes Cancelable: Yes Target: element 返回的接口类型(eventType):keyboardEvent	当用户在浏览器用户界面发起“粘贴”操作时，会触发paste事件。

3.1.1 剪贴板事件的属性和方法(ClipboardEvent)

事件属性	属性的功能

事件属性	属性的功能
ClipboardEvent.clipboardData	是一个 DataTransfer 对象，它包含了由用户发起的 cut 、 copy 和 paste 操作影响的数据,可以通过.getData(fomat)方法得到数据(fomat: "text/plain"，text/"uri-list")

3.2 焦点事件(FocusEvent)

- 你点击某个input的时候,页面中的输入框就会变成可以输入的样子,这就叫做获取焦点

事件名称	事件基本信息	事件触发条件
blur	Bubbles: No Cancelable: No Target: element 返回的接口类型(eventType):FocusEvent	当一个元素失去焦点的时候 blur 事件被触发
focus	Bubbles: No Cancelable: No Target: element 返回的接口类型(eventType):FocusEvent	focus事件在元素获取焦点时触发
focusout	Bubbles: Yes Cancelable: No Target: element 返回的接口类型(eventType):FocusEvent	当元素即将失去焦点时，focusout 事件被触发
focusin	Bubbles: Yes Cancelable: No Target: element 返回的接口类型(eventType):FocusEvent	focusin事件在元素获取焦点时触发

3.2.1 焦点事件的属性和方法(FocusEvent)

事件属性	属性的功能
FocusEvent.relatedTarget	补充

- focus/blur 和 focusin/focusout区别
  - 前两个不触发冒泡事件，而后两个触发冒泡事件

3.3 键盘事件

事件名称	事件基本信息	事件触发条件
keyup	Bubbles: Yes Cancelable: Yes Target: 键盘 返回的接口类型(eventType):KeyboardEvent	当一个按钮按下后被释放时触发
keydown	Bubbles: Yes Cancelable: Yes Target: 元素 返回的接口类型(eventType):KeyboardEvent	当一个按钮被按下时触发

3.3.1 键盘事件的属性和方法(KeyboardEvent)

事件属性	属性的功能
KeyboardEvent.ctrlKey（只读）	返回一个Boolean, 如果按键事件ctrl被按下，则为true
KeyboardEvent.ctrlKey（只读）	返回一个Boolean, 如果按键事件产生ctrl被按下，则为true
KeyboardEvent.AltKey（只读）	返回一个Boolean, 如果按键事件产生Alt被按下，则为true
KeyboardEvent.shiftKey（只读）	返回一个Boolean, 如果按键事件产生shift被按下，则为true
KeyboardEvent.metaKey（只读）	返回一个Boolean, 如果按键事件产生窗口按钮被按下，则为true
KeyboardEvent.code（只读）	返回一个DOMString, code代表事件触发的物理按键
KeyboardEvent.repeat（只读）	返回一个Boolean, 如果按键一直被按住，则返回true

3.4 鼠标事件

事件名称	事件基本信息	事件触发条件
click	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮在一个元素对象上按下和放开时触发
dblclick	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮在一个元素对象上短时间内两次按下和放开时触发

事件名称	事件基本信息	事件触发条件
mousedown	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮在一个元素对象上被（连续）按下时触发
mouseup	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮在一个元素对象上被松开时触发
mouseenter	Bubbles: No Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮进入一个元素对象的区域时触发
mouseleave	Bubbles: No Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮离开一个元素对象的区域时触发
mouseover	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮进入一个元素对象的区域时触发
mouseout	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮离开一个元素对象的区域时触发
mousemove	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):MouseEvent	当定点设备的按钮在一个元素对象上移动时触发

- mouseenter和mouseover的区别
- mouseenter不能产生冒泡事件
- mouseover有冒泡事件

3.4.1 鼠标事件的属性和方法(MouseEvent)

- 鼠标事件

事件属性	属性的功能
MouseEvent.ctrlKey（只读）	返回一个Boolean, 如果鼠标事件ctrl被按下，则为true
MouseEvent.ctrlKey（只读）	返回一个Boolean, 如果鼠标事件产生ctrl被按下，则为true
MouseEvent.AltKey（只读）	返回一个Boolean, 如果鼠标事件产生Alt被按下，则为true
MouseEvent.shiftKey（只读）	返回一个Boolean, 如果鼠标事件产生shift被按下，则为true
MouseEvent.metaKey（只读）	返回一个Boolean, 如果鼠标事件产生窗口按钮被按下，则为true
MouseEvent.detail（只读）	返回在短时间内元素对象被连续点击的次数（long)
MouseEvent.screenX（只读）	返回点击位置对应屏幕的x轴坐标
MouseEvent.screenY（只读）	返回点击位置对应屏幕的y轴坐标
MouseEvent.clientX（只读）	返回点击位置对应浏览器窗口的x轴坐标，当存在滚动条时，只表示在当前窗口的位置
MouseEvent.clientY（只读）	返回点击位置对应浏览器窗口的y轴坐标，当存在滚动条时，只表示在当前窗口的位置
MouseEvent.movementX（只读）	它提供了当前事件和上一个mousemove事件之间鼠标在水平方向上的移动值。相当于currentEvent.movementX = currentEvent.screenX - previousEvent.screenX
MouseEvent.movementY（只读）	它提供了当前事件和上一个mousemove事件之间鼠标在垂直方向上的移动值。相当于currentEvent.movementY = currentEvent.screenY - previousEvent.screenY
MouseEvent.pageX（只读）	返回的相对于整个文档的x（水平）坐标，要和clientX区分开来，当具有滚动条时，表示的是整个页面的鼠标位置，对于ie8及以下不兼容
MouseEvent.pageY（只读）	返回的相对于整个文档的y（水平）坐标，要和clientY区分开来，当具有滚动条时，表示的是整个页面的鼠标位置，对于ie8及以下不兼容
MouseEvent.button（只读）	返回点击按钮(long)，0为鼠标左键，1为鼠标中间的按钮，2为鼠标右键
MouseEvent.buttons（只读）	返回多个点击按钮,通过加号进行计算（long），1为鼠标的左键，2为鼠标的右键，4为鼠标的滚轮，8为浏览器的后退按钮，16为浏览器的前进按钮

3.5 触摸事件

事件名称	事件基本信息	事件触发条件

事件名称	事件基本信息	事件触发条件
touchcancel	Bubbles: Yes Cancelable: No Target: Element 返回的接口类型(eventType):TouchEvent	当触摸点被中断时会触发 touchcancel 事件，中断方式基于特定实现而有所不同（例如， 创建了太多的触摸点）
touchstart	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):TouchEvent	当一个或多个触摸点与触控设备表面接触时触发touchstart 事件
touchmove	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):TouchEvent	当一个或多个触摸点在触控设备表面移动时触发touchstart 事件
touchend	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):TouchEvent	当一个或多个触摸点与触控设备表面离开时触发touchstart 事件

3.4.0 触摸事件的类型

- 触摸事件可以分为三类，可以用TouchEvent.type来查看当前的触摸事件的类型
  - touchstart
    - 当用户在触摸平面上放置了一个触点时触发
  - touchmove
    - 当用户在触摸平面上移动触点时触发
  - touchend
    - 当一个触点被用户从触摸平面上移除（即用户的一个手指或手写笔离开触摸平面）时触发。当触点移出触摸平面的边界时也将触发
  - touchcancel
    - 当触点由于某些原因被中断时触发。有几种可能的原因如下（具体的原因根据不同的设备和浏览器有所不同）：
      - 由于某个事件出现而取消了触摸：例如触摸过程被弹窗打断。
      - 触点离开了文档窗口，而进入了浏览器的界面元素、插件或者其他外部内容区域。
      - 当用户产生的触点个数超过了设备支持的个数，从而导致 TouchList 中最早的 Touch 对象被取消。

3.4.1 触摸事件的属性和方法(TouchEvent)

事件属性	属性的功能
TouchEvent.ctrlKey（只读）	返回一个Boolean, 如果触摸事件ctrl被按下，则为true
TouchEvent.ctrlKey（只读）	返回一个Boolean, 如果触摸事件产生ctrl被按下，则为true
TouchEvent.AltKey（只读）	返回一个Boolean, 如果触摸事件产生Alt被按下，则为true
TouchEvent.shiftKey（只读）	返回一个Boolean, 如果触摸事件产生shift被按下，则为true
TouchEvent.metaKey（只读）	返回一个Boolean, 如果触摸事件产生窗口按钮被按下，则为true
TouchEvent.changedTouches（只读）	返回一个发生改变的元素对象的类数组, 包含所有被触发的对象（因为可以多只手指来触发）
TouchEvent.targetTouches（只读）	返回一个包含所有触点的底层的元素对象的类数组, 包含所有被触发的对象（因为可以多只手指来触发）
TouchEvent.touches（只读）	返回一个触点的元素对象的类数组, 包含所有被触发的对象（因为可以多只手指来触发）

3.4.2 触摸事件的注意事项

- 触摸事件和鼠标事件会同时被触发（目的是让没有对触摸设备优化的代码仍然可以在触摸设备上正常工作）。如果你使用了触摸事件，可以调用 event.preventDefault() 来阻止鼠标事件被触发

3.6 滚轮事件（WheelEvent)

事件名称	事件基本信息	事件触发条件
wheel	Bubbles: Yes Cancelable: Yes Target: Element 返回的接口类型(eventType):WheelEvent	当鼠标的滚轮被滑动时触发

3.6.1 滚轮事件的属性和方法(WheelEvent)

- 继承了鼠标事件的属性和方法

事件属性	属性的功能
deltaX	水平滚动量以WheelEvent.deltaMode 为单位
deltaY	垂直滚动量以WheelEvent.deltaMode 为单位
deltaZ	Z轴滚动量以WheelEvent.deltaMode 为单位

3.6.2 wheel的历史

- 在之前，该事件是以onmousewheel来触发的
- 有一个.wheelDelta事件属性的注意事项
- 获取事件中滚轮滚动的方向
  - 正值：向上，与数值无关
  - 负值：向下,与数值无关
- 属性中火狐不支持
  - 使用event.detail
    - 正值，向下
    - 负值，向上
- 实现兼容判断方向

```
if(event.wheelDelta > 0 || event.detail < 0){
  ... //向上
}
else{
  向下
}
```

3.7 事件的一些要学会的应用

3.7.1 冒泡事件实现子盒和父盒的颜色变化

- 实现功能
  - 点击子盒：盒子颜色变成蓝色
  - 点击父盒：盒子颜色变成黑色

3.7.1.1 利用event.stopPropagation()

- 关键点：
  - 在子盒的触发事件中增加阻止冒泡的作用
    - 当父盒的事件触发且可以满足子盒往下传播的条件时，会被阻止冒泡所阻止，从而导致父盒的回调函数不能被触发

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
      .container {
        width: 500px;
        height: 500px;
        background-color: #ff0000;
      }
      .inner {
        width: 100px;
        height: 100px;
        background-color: #00ff00;
      }
    </style>
</head>
<body>
  <!--
    当鼠标进入.inner时，inner变成2s时间蓝色，而container不变色
    当鼠标进入.container时，container变成黑色
   -->
      <div class="container">
        <div class="inner"></div>
      </div>
    <script type="text/javascript">
      window.onload = function() {
        function changeColor(event, color) {
          alert(event.target.className);
          event.target.style.backgroundColor = color;
        }
        let inner = document.querySelector(".inner");
        let container = document.querySelector(".container");
        inner.addEventListener("click", function(event) {
          event = event || window.event;
          event.stopPropagation();
          changeColor(event, "#0000ff");
        })
        // 使用mouseenter绑定inner -- 非冒泡，且event.target为绑定的事件
        container.addEventListener("click", function(event) {
          event = event || window.event;
          changeColor(event, "#000000");
        })

      }
   </script>
</body>
</html>
```

3.7.1.2 利用冒泡事件的出发元素的不同（即event.target的不同）

- 在冒泡事件中
  - event.target所得到的元素对象为触发该事件回调函数的事件

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
      .container {
        width: 500px;
        height: 500px;
        background-color: #ff0000;
      }
      .inner {
        width: 100px;
        height: 100px;
        background-color: #00ff00;
      }
    </style>
</head>
<body>
  <!--
    当鼠标进入.inner时，inner变成2s时间蓝色，而container不变色
    当鼠标进入.container时，container变成黑色
   -->
      <div class="container">
        <div class="inner"></div>
      </div>
    <script type="text/javascript">
      window.onload = function() {
        function changeColor(event, color) {
          alert(event.target.className);
          event.target.style.backgroundColor = color;
        }
        let container = document.querySelector(".container");
        container.addEventListener("click", function(event) {
          event = event || window.event;
          if(event.target === container)
            changeColor(event, "#000000");
          else
            changeColor(event, "#0000ff");
        })

      }
   </script>
</body>
</html>
```

3.7.2 实现全选框

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="option">
        <form action="https://www.baidu.com/s?" method="get" target="_blank">
            <span>你的爱好运动是?</span>
                <input type="checkbox" id="all-select"><label for="all-select">全选/全不选</label>
            <div class="hobbies">
                <input type="checkbox" id="football" name="wd" value="football"><label for="football">足球</label>
                <input type="checkbox" id="basketball" name="wd" value="basketball"><label for="basketball">篮球</label>
                <input type="checkbox" id="badminton" name="wd" value="badminton"><label for="badminton">羽毛球</label>
                <input type="checkbox" id="ping-pong" name="wd" value="ping-pong"><label for="ping-pong">乒乓球</label>
            </div>
            <div class="buttons">
                <button id="btnSelectAll" type="button">全选</button>
                <button id="btnSelectNone" type="button">全不选</button>
                <button id="btnSelectNot" type="button">反选</button>
                <button type="submit">提交</button>
            </div>
        </form>
    </div>
    <script>
        const btnSelectAll = document.querySelector('#option .buttons #btnSelectAll');
        const btnSelectNone = document.querySelector('#option .buttons #btnSelectNone');
        const btnSelectNot = document.querySelector('#option .buttons #btnSelectNot');
        const selectAll = document.querySelector('#option #all-select');
        const inputsHobbies = document.querySelectorAll('#option .hobbies input');
        // 思路
        /*
            1. 将各个按钮的功能和input选择框单项联系起来
            2. input选择框单项和各个按钮的功能联系起来
            3. 总结和提取相应的函数以简化代码
        */
        /*
        *给某一个按钮绑定一个点击事件，用于操作所有的选择框
        *button: 绑定的按钮
        *func:  操作选择框内容的函数
        */
        function btnClick(button, func) {
            button.onclick = function() {
                // 对input框进行操作
                inputsHobbies.forEach(func);
                // 判断上面的全选框的结果
                // flag用于判断是不是全选
                let flag = true;
                inputsHobbies.forEach(btn => flag = btn.checked && flag);
                selectAll.checked = flag;
            }
        }
        /*绑定三个按钮以及全选择框以及4个input*/
        btnClick(btnSelectAll, inputHobbies => inputHobbies.checked = true)
        btnClick(btnSelectNone, inputHobbies => inputHobbies.checked = false)
        btnClick(btnSelectNot, inputHobbies => inputHobbies.checked = !inputHobbies.checked)
        btnClick(selectAll, inputHobbies => inputHobbies.checked = selectAll.checked)

        /*由于不需要对每个input执行函数，所以使用了空的函数*/
        inputsHobbies.forEach(inputHobbies => btnClick(inputHobbies, ()=>{}))
        // 箭头函数的this是在定义外函数
    </script>
</body>
</html>
```

3.7.3 实现元素的拖拽

- 当鼠标在该拖拽元素按下时，开始拖拽(onmousedown)
- 当鼠标移动时，被拖拽元素(onmousemove)
- 当鼠标松开时，拖拽元素固定在当前位置(onmouseup)

3.7.3.1 拖拽时鼠标位于左上角

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box1 {
            width: 100px;
            height: 100px;
            background-color: #ff0000;
            position: absolute;
        }
        .box2 {
            width: 100px;
            height: 100px;
            background-color: #00ff00;
            position: absolute;
            left: 200px;
            top: 200px;
        }
    </style>
</head>
<body>
    <div class="box1"></div>
    <div class="box2"></div>
    <!-- 相同的absolute的层叠现象，后面的覆盖先前的 -->
    <script>
        window.onload = function() {
            let box1 = document.querySelector('.box1');
            box1.onmousedown = function() {
                // alert('开始拖拽')
                // 不能给box1绑定，因为鼠标会脱离对应的边缘
                document.onmousemove = function(event) {
                    event = event || window.event;
                    const x = event.clientX
                    const y = event.clientY;
                    const scrollTop = document.body.scrollTop || document.documentElement.scrollTop;
                    box1.style.left = x + 'px';
                    box1.style.top = y + scrollTop + 'px';
                }
                // box1.onmouseup
                document.onmouseup = function() {
                   document.onmousemove = null;
                   document.onmoveup = null;
                }
            }
        }
        /* 问题：
            1. 当拖拽元素移动到第二个非拖拽元素的位置时，由于层叠的原因会被覆盖。此时松开鼠标，并没有实现元素的位置固定
             原因：鼠标绑定的是box1，当移动到第二个元素时，松开时触发的是第二个元素mouseup的事件而不是box1，所以无效
             改box1 -> document
            2. 改成document要注意取消该事件。(没有存在意义的事件要取消)

*/
    </script>
</body>
</html>
```

3.7.3.2 拖拽位置位于鼠标位置刚开始点击的位置

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box1 {
            width: 100px;
            height: 100px;
            background-color: #ff0000;
            position: absolute;
            /* 相对于body进行定位 */
        }
        .box2 {
            width: 100px;
            height: 100px;
            background-color: #00ff00;
            position: absolute;
            left: 200px;
            top: 200px;
        }
    </style>
</head>
<body>
    <div class="box1"></div>
    <div class="box2"></div>
    <!-- 相同的absolute的层叠现象，后面的覆盖先前的 -->
    <script>
        window.onload = function() {
            let box1 = document.querySelector('.box1');
            box1.onmousedown = function(event) {
                // alert('开始拖拽')
                // 不能给box1绑定，因为鼠标会脱离对应的边缘
                // offsetX， offsetY为元素和鼠标之间的相对距离
                // offsetLeft为定位元素相对于祖先非static的left，且返回一个数值
                // .style是用于设置元素的位置，进行计算时记得使用元素对象的属性而不是style的属性
                let offsetX = event.clientX - this.offsetLeft;
                let offsetY = event.clientY - this.offsetTop;

                document.onmousemove = function(event) {
                    event = event || window.event;
                    let x = event.clientX
                    let y = event.clientY;
                    let scrollTop = document.body.scrollTop || document.documentElement.scrollTop;
                    box1.style.left = x - offsetX + 'px';
                    box1.style.top = y + scrollTop - offsetY + 'px';
                }
                // box1.onmouseup
                document.onmouseup = function() {
                   document.onmousemove = null;
                   document.onmouseup = null;
                }
            }
        }
    </script>
</body>
</html>
```

3.7.3.3 取消浏览器的默认行为造成拖拽的bug

- 当我们拖拽一个网页的内容时，浏览器会默认去搜索引擎中搜索内容，这时回导致出现拖拽的bug
- 方法1: 利用return false取消浏览器的默认行为
  - 该方法对ie8及以下没有作用

```
box.onmousedown = function() {
  ...
  return false;
}
```

- 方法2：利用elementObject.setCapture()和elementObject.releaseCapture();
- 当一个元素设置了setCapture(),这个元素会把下一次全部相同的事件捕获到自己的身上
- 当mousedown的时候设置，当mouseup时释放

```
box.onmousedown = function() {
  elementObject.setCapture && elementObject.setCapture();
  ...
  ...
  box.onmouseup = function() {
    ...
    elementObject.releaseCapture && elementObject.releaseCapture();
  }
}
```

- 方法3： 兼容性写法(利用短路)

```
  box.onmousedown = function() {
    elementObject.setCapture();
    ...
    ...
    box.onmouseup = function() {
      ...
      elementObject.releaseCapture();
    }
    return false;
  }
```

3.7.3.4 总结一个拖拽函数

```
function drag(obj) {
  obj.onmousedown = function(event) {
      // 解决浏览器默认搜索的bug
  obj.setCapture && obj.setCapture();
      // 实现浏览器兼容
  event = event || window.event;
      // 得到相对位置，注意在计算过程中是不用style中的值
  let offsetX = event.clientX - this.offsetLeft;
  let offsetY = event.clientY - this.offsetTop;
  document.onmousemove = function(event) {
      let mouseX = event.clientX;
      let mouseY = event.clientY;
      // 解决下方出现滚动条的问题
      let scrollTop = document.body.scrollTop || document.documentElement.scrollTop;
      obj.style.left = mouseX - offsetX + 'px';
      obj.style.top = mouseY + scrollTop - offsetY + 'px';
  }
  // document防止延迟时鼠标脱离范围导致无法触发
  document.onmouseup = function() {
      // 对于document的方法，在结束触发时要记得消除；
      document.onmousemove = null;
      obj.releaseCapture && obj.releaseCapture();
      document.onmouseup = null;
  }
  // 同样是取消浏览器的默认行为
  return false;
}
}
```

3.7.4 滚轮事件的运用

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box {
            min-height: 50px;
            height: 100px;
            width: 100px;
            background-color: #ff0000;
        }
    </style>
</head>
<body>
    <!-- 鼠标向下滚动时，div高变大，向上滚动时变小 -->
    <div class="box"></div>
    <script>
        const box = document.querySelector('.box');

        function bind(obj, eventName, callback) {
            if(obj.addEventListener) {
                obj.addEventListener(obj, eventName, callback);
            }
            else {
                obj.attachEvent(eventName, function() {
                    callback.call(obj);
                })
            }
        }
        // onmousewheel中火狐并不支持，只能使用addEventListener进行绑定，且对应的事件名称为'DOMMouseScroll'
        box.onmousewheel = function (event) {
            // 火狐中并没有event.wheelDelta属性，只有event.detail;
            if(event.wheelDelta > 0 || event.detail < 0){
                box.style.height = box.clientHeight - 10 + 'px';
            }
            else {
                box.style.height = box.clientHeight + 10 + 'px';
            }
            // 当触发该响应函数时，取消滑动鼠标滚轮时浏览器滚动条的默认行为
            event.preventDefault && event.preventDefault();
            // ie浏览器并不支持这种方式来取消浏览器的默行为
            return false;
        }

        bind(box,'DOMMouseScroll', box.onmousewheel);

    </script>
</body>
</html>
```

- 注意
  - 取消浏览器的默认行为的两种方法
  - onmousewheel在两种浏览器的不同写法，以及共用写法。

3.7.5 键盘事件的运用

- 利用event.keyCode获得对应键盘的编码

3.7.5.1 实现两个按键的判断

```
// ctrl+y的判断
  document.onkeydown = function(event) {
    if(event.ctrlKey && event.keyCode === 17) {
      ...
    }
  }
```

3.6.7.2 限制input框的输入内容

```
// 当input被取消浏览器的默认行为时，则输入的内容不会出现在文本框之中
  input.onkeydown = function(event) {
    ...
    return false;
  }
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div>
        <input type="text">
    </div>
    <script>
        // 禁止input框输入数字
        let inputElement = document.querySelector('input');
        inputElement.onkeydown = function(event) {
            event = event || window.event;
            // 利用return false取消默认行为的方法
            if(event.keyCode >= 48 && event.keyCode <= 57)
                return false;
        }
    </script>
</body>
</html>
```

3.6.7.3 div元素的移动

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box {
            width: 50px;
            height: 50px;
            background-color: #ff0000;
            position: absolute;
        }
    </style>
</head>
<body>
    <div class="box"></div>
    <script>
        let step = 5;
        window.onload = function() {
            const dir = {
                38: [0, -1],
                39: [1, 0],
                40: [0, 1],
                37: [-1, 0]
            }
            let box = document.querySelector('.box');
            document.onkeydown = function(event) {
                if (event.shiftKey && event.keyCode === 38) {
                    step += 5;
                    return;
                }
                if (event.shiftKey && event.keyCode === 40) {
                    step = step >= 10 ? step - 5 : 5;
                    return;
                }
                event = event || window.event;
                // 上：38 右：39 下：40 左：37
                console.log(box.offsetLeft)
                box.style.left = box.offsetLeft + step*dir[event.keyCode][0] + 'px';
                box.style.top = box.offsetTop + step*dir[event.keyCode][1] + 'px';
            }
        }
    </script>
</body>
</html>
```

3.8 与事件相关的兼容性问题

3.8.1 获取样式表中属性值的兼容性问题

```
  function getStyle(obj, name) {
  // 判断浏览器版本太麻烦，用什么进行判断？
  // 正常浏览器
    if(window.getComputedStyle)
      return getComputedStyle(obj, null)[name];
    else
      return obj.currentStyle[name];
  // ie8浏览器Style
}
```

3.8.2 event参数的传递

- 为了解决兼容性的问题，事件在事件函数中传递event常用以下形式
  - ie8以下不传递该参数

```
  elenmentObject.onevent = function(event) {
    const event = event || window.event
  }
```

3.8.3 绑定事件的兼容性写法

```
/*
* 参数
* 1. obj 要绑定的事件对象
* 2. eventName：事件名称
* 3. callback：回调函数
*/
// addEventListener()中的callback函数中的this，是绑定事件对象
// attachEvent()中的allback函数中的this是window， 所以需要修改callback的this对象
  function bind(obj, eventName, callback){
    if(obj.addEventListener)
      obj.addEventListener( eventName, callback, false);
    else
      obj.attachEvent('on' + eventName,function() {
        callback.call(obj); //将call的this指向obj
      });
  }
// callback.bind(obj)
```

6.1.3.1. elementObject.clientWidth, elementObject.clientHeight

- 返回元素对象的可见宽度和可见高度
  - 即包括content和padding（包含滚动条）
  - 由于是计算过来的，所以readonly，不能够修改
- 只是返回相应的数字而不带px，可以直接用于计算

6.1.3.2. elementObject.offsetWidth, elementObject.offsetHeight

- 返回元素对象的整个宽度和整个高度
  - 即包括content + padding + border
  - 由于计算来的，所以readonly
- 只是返回相应的数字

6.1.3.3. elementObject.offsetParent

- 获取当前元素的定位父元素对象
  - 获取到的离当前元素最近的非static祖先元素
- 若没有，则默认为body

6.1.3.4. elementObject.offset[Left,right], elementObject.offset[Top,buttom]

- 当前元素相对于其定位元素(offsetParent(非static))的水平偏移量/垂直偏移量
  - 包含父元素的padding+top/left

6.1.3.5. elementObject.scrollWidth, elementObject.scrollHeight

- 在父元素有overflow:auto/scroll下
  - 获取滚动区域整个宽度/高度

6.1.3.6. elementObject.scrollLeft, elementObject.scrollTop

- 在父元素有overflow:auto/scroll下
  - 获取滚动条滚动的距离（水平/垂直）

6.1.3.7. elementObject.scrollHeight, elementObject.scrollTop和elementObject.clientHeight的结合使用

- 实现判断滚动条是否已经滚动到底

```
// 当垂直滚动条滚动到底的时候有
elementObject.clientHeight == elementObject.scrollHeight - elementObject.scrollTop
```

6.2.6.4. elementObject.onmousewheel

- 鼠标在该元素对象中的滚轮触发该事件
  - 火狐不兼容
  - 在火狐中，需要使用DMMMouseScroll来绑定事件
    - 必须使用addEventListener来进行绑定
    - 可以使用bind函数来实现绑定

```
elementObject.onmousewheel = function() {};
bind(elementObject,'DOMMouseScroll', elementObject.onmousewheel);
```

3.8. 文档的加载

3.8.1 浏览器加载页面的顺序

- 浏览器加载页面时，是按照自上向下的顺序加载的，读取到一行就运行一行
  - js代码写到页面下面的原因就是为了可以在页面加载完毕以后再执行js代码

3.8.2 onload事件

- 一张图片或页面加载完成之后才执行
- window.onload = function() {...}
  - 页面加载完成之后才会执行对应的函数

```
window.onload = function() {
  btn.onclick = () => alert("点击了一下");
}
```

六 js的BOM操作

6.1. BOM(brower Object Model)

- 浏览器对象模型
  - BOM可以使我们通过js来操作浏览器

6.2. BOM的对象

- 在浏览器当中都是作为window的对象的属性保存的，可以直接使用window对象来使用，也可以直接使用

6.2.1. Window

- 代表的是整个浏览器的窗口，同时window也是网页的全局对象

6.2.1.1. Window方法

6.2.1.1.1. alert(str)

- 弹出对话框，提示str内容

6.2.1.1.2. prompt(str)

- 弹出对话框，提示str内容要求输入，并返回输入值

6.2.1.1.3. comfirm(str)

- 弹出对话框，显示str内容，并有确认和取消按钮，返回布尔值

6.2.1.1.4. setInterval(callback, time)

- 定时函数
- 将一个函数每隔一段时间调用一次,time单位是毫秒（多次）
- 返回一个Number数据定时器id，这个数字作为定时器的唯一标识（id）

6.2.1.1.5. clearInterval(intervalId)

- 用来关闭intervalId的定时器
- 内部可以添加任何类型的值，若所添加的不是intervalId则什么也不做

6.2.1.1.6. setTimeout(callback, time)

- 延时调用一个函数不马上执行而是隔一段时间再执行，只能实行一次

6.2.1.1.7. clearTimeout(timeoutId)

- 关闭一个演延时调用函数

6.2.1.1.8. 延时调用和定时调用的关系

- 延时调用和定时调用之间是可以相互替代的

6.2.2. Navigator

- 代表的当前浏览器的信息，通过该对象可以用来识别不同的浏览器

6.2.2.1. Navigator的属性

- 由于历史原因，Navigator对象中的大部分属性都已经不能帮助我们识别浏览器

7.2.2.1.1. navigator.userAgent

- 一般我们只会使用userAgent用来判断浏览器的信息
- 返回的是一个字符串，这个字符串中包含有用来秒速浏览器信息的内容，不同浏览器会有不同的userAgent

```
// 适用于判断大部分浏览器名称
  const ua = navigator.userAgent;
  // i正则表达式忽略大小写
  if(/firefox/i.test(ua))
      alert("我是火狐");
  else if(/chrome/i.test(ua))
      alert("我是chrome")
  else if(/msie/i.test(ua))
      alert("我是ie10及以下")
  else if("ActiveXObject" in window)
      console.log("你是ie11")

  // 如果无法通过UserAgent不能判断，可以通过一些浏览器中特有的对象来识别
  // 不如ActiveXobject
  if("ActiveXObject" in window)
      console.log("你是ie")
  else
      console.log("你不是ie")
```

6.2.3. Location

- 代表当前浏览器的地址栏信息，通过locatino可以获取地址按信息或者操作浏览器跳转页面

6.2.3.1. localtion

- 可以得到当前页面的完成路径
  - 给location赋值，则可以实现跳转

7.2.3.1.1. location实现元素对象的类似a标签的属性

- 若给location赋值一个完整路径或者相对路径，则页面会自动跳转到该路径，并且生成了相应的历史记录

6.2.3.2. location的属性

属性	描述
hash	设置或返回从井号（#）开始的 URL（锚）。
host	设置或返回主机名和当前 URL 的端口号。
hostname	设置或返回当前 URL 的主机名。
href	设置或返回完整的 URL。
pathname	设置或返回当前 URL 的路径部分。
port	设置或返回当前 URL 的端口号。
protocol	设置或返回当前 URL 的协议。
search	设置或返回从问号（?）开始的 URL（查询部分）。

  ;

6.2.3.3. Location的方法

6.2.3.3.1. location.assign(URL);

- 跳转到URL，和直接赋值的作用相同

6.2.3.3.2. location.reload();

- 刷新页面，和刷新页面的按钮作用相同，不清空缓存
- location.reload(true)
  - 强制清空缓存并刷新页面

6.2.3.3.3. location.replace(URL)

- 跳转到URL。且不留下历史记录

6.2.4. History

- 代表浏览器的立是记录，可以通过该对象来操作浏览器的历史记录，由于隐私原因，只能操作浏览器向前或向后跳转，而且该操作只在当次访问时有效。
- 可以用来进行浏览器的向前向后翻页

6.2.4.1. History的属性

6.2.4.1.1. history.length

- 获取到当次访问页面的数量（关闭窗口就重新计数）

6.2.4.2. History的方法

6.2.4.2.1. history.back()

- 可以用来回退到上一个页面，作用和浏览器的回退效果一样

6.2.4.2.2. history.forward()

- 可以用来前进到上一个页面，作用和浏览器的回退效果一样

6.2.4.2.3. history.go(n)

- 可以用来跳转到指定的页面
  - n为正数，向前前进n个页面
  - n为负数，向后回退n个页面

6.2.5. Screen

- 代表用户的屏幕的信息，通过该对象可以获取到用户的显示器相关的信息

6.2.6. Bom的应用

6.2.6.1. 图片切换

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        body {
            text-align: center;
        }
        .box {
            width: 500px;
            height: 500px;
            padding: 20px;
            background: #eaeaea;
            margin: 0 auto;
        }
        .box img{
            width: 500px;
            height: 500px;

        }
    </style>
</head>
<body>
    <div class="box">
        <img src="../img/ex1.JPG" alt="图片">
    </div>
    <button class="begin">开始图片的切换</button>
    <button class="end">终止图片的切换</button>
    <script>
        window.onload = function() {
            const imgUrl = [
                '../img/ex1.JPG',
                '../img/ex2.JPG',
                '../img/ex3.JPG',
                '../img/ex4.JPG',
                '../img/ex5.JPG',
                '../img/ex6.JPG',
                '../img/ex7.JPG',
                '../img/ex8.JPG'
            ]
            let index = 0;
            function imgChange(img, imgUrl) {
                img.src = imgUrl[index];
                index = (index + 1) % imgUrl.length;
            }
            const img = document.querySelector('.box img')
            let id;
            const btn1 = document.querySelector('button.begin');
            btn1.onclick = function() {
                clearInterval(id);
                id = setInterval(() => {
                    imgChange(img, imgUrl);
                }, 1000);
                // 点击多次之后会越来越快的原因：
                /*每点击一个按钮，就会添加一个定时器，所以会开很多个定时器，所以切换速度变快了*/
                // 无法停下来的原因：只能清除最后一个定时器
                // 解决方案：在同一个元素对象开启一个定时器之前先清除先前的定时器。
            }
            const btn2 = document.querySelector('button.end');
            btn2.onclick = function() {
                clearInterval(id);
            }
        }
    </script>
</body>
</html>
```

6.2.6.2. 解决div移动第一个键和第二个键之间的延迟问题（防误触）

- 触发机制使用一个定时器函数用于减少按键的间隔
- 方向由事件决定，速度由定时器决定

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box {
            width: 50px;
            height: 50px;
            background-color: #ff0000;
            position: absolute;
        }
    </style>
</head>
<body>
    <div class="box"></div>
    <script>
        let curValue = 0;
        let step = 5;
        let box = document.querySelector('.box');
        const dir = {
                38: [0, -1],
                39: [1, 0],
                40: [0, 1],
                37: [-1, 0]
            }
        window.onload = function() {
            function Operation(curValue) {
                if(!dir[curValue])
                    return;
                // 上：38 右：39 下：40 左：37
                box.style.left = box.offsetLeft + step*dir[curValue][0] + 'px';
                box.style.top = box.offsetTop + step*dir[curValue][1] + 'px';
                // return 0;
            }
            // 触发机制使用一个定时器函数用于减少按键的间隔
            // 方向由事件决定，速度由定时器决定
            setInterval(function(){
                Operation(curValue)
             }, 30)
            // 不能用清除interval，再重新建立一个interval会造成延迟的过程
            document.onkeydown = function(event) {
                event = event || window.event
                if (event.shiftKey && event.keyCode === 38) {
                    step += 5;
                    return;
                }
                if (event.shiftKey && event.keyCode === 40) {
                    step = step >= 10 ? step - 5 : 5;
                    return;
                }
                curValue = event.keyCode;
            }
            document.onkeyup = function() {
                curValue = 0;
            }
        }
    </script>
</body>
</html>
```

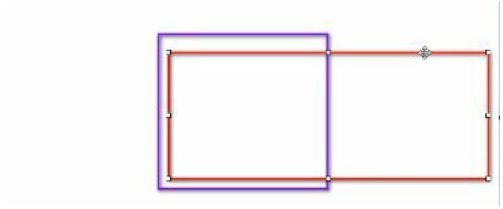
6.2.6.3. 构造一个简单的动画函数

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .box1 {
            position: absolute;
            width: 100px;
            height: 100px;
            background-color: #ff0000;
            left: 0px;
            top: 40px;
        }
    </style>
</head>
<body>
    <button class="btn1">点击往右走</button>
    <button class="btn2">点击往左走</button>
    <button class="btn3">点击往上走</button>
    <button class="btn4">点击往下走</button>
    <button class="btn5">增加宽度</button>
```

```
<button class="btn6">增加高度</button>
<button class="stop">暂停</button>
<div class="box1"></div>
<script>
    window.onload = function() {
        /*
        * 作用：以字符串的形式返回元素对象对应的cssStyle
        * 参数：
        *       obj:元素对象
        *       cssStyle:string 要取得的样式
        * 返回：对象元素样式的字符串表示(:string)
        * 注意：对应的属性一定要进行初始化，不然会返回auto
        */
        function getStyle(obj, cssStyle) {
            if(window.getComputedStyle)
                return getComputedStyle(obj, null)[cssStyle];
            else //IE8及以下没有getComputedStyle方法
                return obj.currentStyle[cssStyle];
        }
        function clearObjectInterval(obj) {
            clearInterval(obj.intervalId);
            obj.intervalId = undefined;
        }
        let box1 = document.querySelector('.box1');
        let btn1 = document.querySelector('.btn1');
        let btn2 = document.querySelector('.btn2');
        let btn3= document.querySelector('.btn3');
        let btn4 = document.querySelector('.btn4');
        let btn5 = document.querySelector('.btn5');
        let btn6 = document.querySelector('.btn6')
        let btnStop = document.querySelector('.stop');

        // let intervalId;
        // 利用在对象中单独定义一个intervalId，从而绑定对象的intervalId,从而可以进行复用
        // let speed = 10;
        /*
        *  可以执行简单动画的函数
        *  参数：
        *       obj: 要执行动画的对象
        *       speed: 运动的速度
        *       target:运动的终止目标
        *       attr: 要执行运动的属性
        *       callback:动画执行完后调用的函数，可以利用该性质实现自动旋转等功能
        */
        function autoMove(obj, speed, target, attr, callback=function(){}) {
            // 清除定时器
            clearInterval(obj.intervalId)

            if(attr === 'left' || attr === 'right') {
                if(attr === 'left')
                    speed = -speed;
                attr = 'left';
            }
            if(attr === 'top' || attr === 'buttom') {
                if(attr === 'top')
                    speed = -speed;
                attr = 'top';
            }
            // 定义定时器
            obj.intervalId = setInterval(function() {
                let currentPositon = parseInt(getStyle(obj, attr));
                let nextPosition = currentPositon + speed;
                obj.style[attr] = speed < 0 ? nextPosition < target ? target + 'px': nextPosition + 'px'
                                            : nextPosition > target ? target + 'px': nextPosition + 'px';
                if(nextPosition === target) {
                    clearObjectInterval(obj)
                    callback();
                }
            }, 30)


        }
        btn1.onclick = function() {
            autoMove(box1, 10, 800, 'right',function(){alert("到底了")});
        }
        btn2.onclick = function() {
            autoMove(box1, 10, 0, 'left');
        }
        btn3.onclick = function() {
            autoMove(box1, 10, 40, 'top');
        }
        btn4.onclick = function() {
            autoMove(box1, 10, 800, 'buttom');
        }
        btn5.onclick = function() {
            autoMove(box1, 10, 800, 'width');
        }
        btn6.onclick = function() {
            autoMove(box1, 10, 800, 'height');
        }
```

```
            btnStop.onclick = function() {
                clearObjectInterval(box1);
            }

            // btn1.onclick = function() {
            //     // 关闭先前的定时器
            //     clearInterval(intervalId);
            //     // 打开新的定时器
            //     intervalId =  setInterval(function() {
            //         // 若直接使用offsetLeft, offsetTop无法和所要接收的style形成一一对应的关系，所以可以使用getStlye()的方法得到对应的样式
            //         let currentPosition = parseInt(getStyle(box1, 'left'));
            //         let nextPosition = currentPosition + speed;
            //         // box终止的位置
            //         if(nextPosition > 800)
            //             nextPosition = 800;
            //         box1.style.left = nextPosition + 'px';
            //     }, 30)
            // }
            // btn2.onclick = function() {
            //      // 关闭先前的定时器
            //     clearInterval(intervalId);
            //     // 打开新的定时器
            //     intervalId = setInterval(function() {
            //         // 若直接使用offsetLeft, offsetTop无法和所要接收的style形成一一对应的关系，所以可以使用getStlye()的方法得到对应的样式
            //         let currentPosition = parseInt(getStyle(box1, 'left'));
            //         let nextPosition = currentPosition +(-speed) ;
            //         // box终止的位置
            //         if(nextPosition < 0)
            //             nextPosition = 0;
            //         box1.style.left = nextPosition + 'px';
            //     }, 30)
            // }
            // 重复的部分有点多，可以提取成一个对象或者函数，注意到函数的共用变量intervalId，可以作为返回值

        }
    </script>
</body>
</html>
```

6.2.6.4. 轮播图效果的实现

- 原理
  - 

七 js的迭代器

八 Promise

1 Promised的基本理解

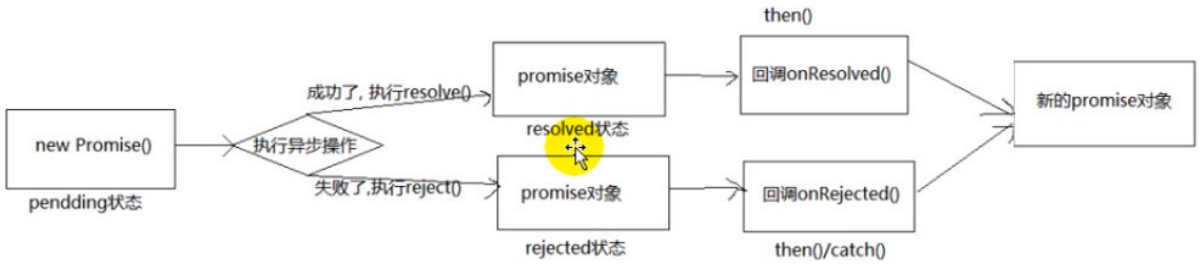
1.1 promise的理解

- 通常情况下, 要实现异步调用的返回, 必须判断异步调用有没有执行结束, 然后通过callback函数来获取数据
- 抽象表达
  - Promise是js中进行异步调用的解决办法
- 具体表达
  - 从语法上来说:Promise是一个构造函数
  - 从功能上来说:Promse对象用于封装一个异步操作的方法并可以获取其结果

1.2 promise的状态

- pending
  - 未确定状态
- fulfilled
  - 完成状态
    - 当调用了resolved函数, 就会从pending -> fulfilled转化
- rejected
  - 拒绝状态
    - 当调用了rejected函数, 就会从pending-> rejected转化

1.3. promise的基本使用流程



-

1.4. 为什么使用promise

1.4.1 指定回调函数的方式更加灵活

- 传统的回调函数的定义必须在异步任务之前
- promise回调函数的定义可以在程序的任意位置
  - promise => 启动异步而任务, 返回promise对象 => 给promise对象绑定回调函数
  - 甚至在异步任务完成之后的任意时刻都进行回调.

1.4.2 支持链式调用,可以解决回调地狱的问题

1.4.2.1 回调地狱

- 回调函数的嵌套调用,外部回调函数异步执行的结果是嵌套的回调函数的前提
  - 难以阅读而且很难找到错误

```
// 涉及到多个异步操作, 而且异步操作是串联执行
// 串联执行: 下一个异步任务是以前面的异步任务的结果作为条件调用的
    doFirstSomething(function(firstResult, failCallBack) {
        doSecondSomething(function(secondResult, failCallBack){
            doThirdSomething(function(thirdResult, failCallBack) {
                console.log(thirdResult)
            })
        })
    })
```

1.4.2.2 利用Promise的链式调用解决回调地狱的问题

```
    doFirstSomething().then(function(firstResult) {
        return doSecondSomething()
        // 必须返回一个promise类型
    })
    .then(function(secondResult) {
        return doThirdSomething()
    })
    .then(function(thirdResult) {
        console.log(thirdResult);
    })
    .catch(failCallBack)
```

1.4.2.3 解决回调地狱的最终方案

- 使用async和await

```
    async function request() {
        try {
            const firstResult = await doFirstSomething();
            const secondResult = await doSecondSomething(firstResult);
            const thirdResult = await doThirdSomething(secondResult);
            console.log(thirdResult);
        }
        catch {
            failCallBack();
        }

    }
```

1.5 promise的API

- 语法和前后端交互的方法
- (prototype)函数原型对象上的方法只能由实例对象进行调用
  - 因为有原型链

5.1 基本语法

```
new Promise(function(resolve, reject) {...}/*executor*/)
```

- executor是带有 resolve 和 reject 两个参数的函数
- Promise构造函数执行时立即调用executor 函数， resolve 和 reject 两个函数作为参数传递给executor
- executor函数在Promise构造函数返回所建promise实例对象前被调用

5.2 函数对象的方法

5.2.1 Promise.all(iterable))

- 作用: 用于将多个pending状态的promise实例对象的结果统一起来
- 返回一个Promise实例
  - 如果传入的参数是一个空的可迭代对象，则返回一个已完成（already resolved）状态的Promise
  - 如果传入的参数不包含任何 promise，则返回一个异步完成（asynchronously resolved）
  - 如果有一个promise的运行最终状态是非fullfilled, 则会回调reject的结果
- 参数
  - iterable
    - 以数组的形式, items为一般为多个处于pending状态的promise的实例对象

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```

5.2.2 Promise.race(iterable)

- 作用: 用于将多个pending状态promise实例进行竞争,只是取第一个执行成功的值
- 返回一个promise对象
- 参数
  - iterable
    - 以数组的形式, items为一般为多个处于pending状态的promise的实例对象

5.2.3 Promise.resolve(value)

- 作用:产生一个成功值为value的promise
- 返回一个以给定值解析后的Promise实例对象
- 参数
  - value
  - 将被Promise对象解析的参数也
    - 可以是一个Promise对象
    - 是一个thenable。
      - 即将被传入then方法中的参数

```
// 产生一个成功值为2的promise
const p1 = new Promise((resolve, reject) => {
    resolve(2)
})
// 实际上可以看作是语法糖
const p1 = Promise.resolve(2)
```

5.2.4 Promise.reject(reason)

- 返回一个失败的promise的实例对象

5.3 实例对象的方法

- 函数原型对象的方法只能由实例对象进行调用

5.3.1 Promise.prototype.then(onFulfilled[, onRejected])

- 作用: 当函数处于fulfilled状态时调用的函数, 指定一个异步调用的回调函数
-
- 返回一个新的promise
- 参数
  - onResolved函数: 成功的回调函数 (value) => {}
  - onRejected函数: 失败的回调函数 (reason) => {}

```
primiseObject.then(value => {
  // fulfillment
}, reason => {
  // rejection
});
```

5.3.2 Promise.prototype.catch(onRejected)

- 相当于then的语法糖, 相当于then(undefined, onRejected)
- 返回一个新的promise对象
- 参数
  - onRejected函数: 失败的回调函数 (reason) => {}

1.6 promise的几个关键问题

1.6.1 如何改变promise的状态

- resolve(value): 如果当前是pending状态=>fulfilled状态
- reject(reason): 如果当前是pending状态=>rejected状态
- 抛出异常: 如果当前是pending状态=>rejected状态

```
    const p = new Promise((resolve, rejected) =>{
        //resolve(1) //promise变为fulfilled状态
        //reject(2) // promise变为rejected的状态
        throw new Error("出错了") //promise变为rejected的状态, reason为Error
    })
```

1.6.2 什么时候可以得到数据?

- 指定回调函数且promise的状态为fulfilled时,会异步调用回调函数
- 一般情况
  - 先指定回调函数, 再改变状态
- 特殊情况
  - 先改变状态,再指定回调函数

```
    const p = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(2);
        }, 1000)
    })
    setTimeout(() => {
        p.then((value) => {
            console.log(value);
        })
    }, 2000)
```

1.6.3 理解promise中的同步异步

```
// 同步: new Promise内部的函数, setTimeout函数, then函数
// 异步: setTimeout的回调函数, then的回调函数
// 异步是要放入一个队列待执行的
    const p = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(1)
        }, 1000)
    }).then(value => {
        console.log(value);
    })
```

1.6.4 promise.then()返回新的promise的结果状态由什么决定的

- 由then指定的回调函数的返回结果决定
  - 如果抛出异常,新promise会变为rejected, reason为抛出的异常
  - 如果返回的是非promise的任意值, 新的promise变为resolved, value为返回的值(包括没有返回值的undefined)
  - 如果返回的是另一个新的promise, 此promise的状态结果即为返回的状态

```
new Promise((resolve, reject) => {
    resolve(2);
})
.then(value => {
    // return 2;                  //fulfilled
    // return Promise.resolve(3);  //fulfilled
    // return Promise.reject(2)    //rejected
    // throw 5  //rejected
})
.then(value => {
    console.log(value)
})
```

1.6.5 promise如何串联多个异步任务

- promise的then()返回一个新的promise, 可以看成then()的链式调用
- 通过then的链式调用串联多个同步/异步任务

1.6.6 promise的异常穿透

- 当使用promise的链式调用时,可以在最后指定失败的回调
- 前面任何操作出了异常,都会传到最后是吧的回调中进行处理
  - 实际上.catch管理的并不是new的对象,而是通过then创建的一个个新对象,所以有穿透的概念
- 定义方法:
  - 所有的OnRejected的回调函数都是throw reason

```
new Promise((resolve, reject) => {
    resolve(2);
})
.then(value => {
    // return 2;                  //fulfilled
    // return Promise.resolve(3);  //fulfilled
    // return Promise.reject(2)    //rejected
    // throw 5  //rejected
}, reason => throw reason)
.then(value => {
    console.log(value)
}, reason => throw reason)
.catch(reason => {
    console.log(reason)
})
```

1.6.7 中断promise链

- 当使用promise的链式调用时, 在中间中断,不再调用后面的回调函数
- 办法: 在回调函数的后面返回一个pending状态的promise对象
  - return new Promise((resolve, reject) = > {})

2 手写promise

2.1 定义整体结构

- 要注意哪些是Promise的函数，哪些是Promise.prototype函数

2.2. 定义构造函数

- Promise构造函数的基本变量
  - this.status
    - 表示promise状态
    - 默认值是"pending"
  - this.data
    - undefined
    - 存储resolve， reject传进来的数据
  - this.callbacks
    - 存储异步调用时的回调函数
    - item是object
- 返回新promise的过程

```
const p = new Promise((resolve, reject) => {
  ...
})
```

- excutor为在类的构造中需要调用的函数，所以在构造函数中需要执行excutor
  - 由传入的参数知，其接收两个函数形式的参数
- 这两个函数形式的参数的实参为Promise中已经定义好的函数
  - 两个函数在类内部定义，在类外部被调用
- resolve/reject执行的过程
  - 改变promise的状态

- 将data赋给类中的data
- 异步执行指定的回调函数
  - 由于是异步执行，肯定会在同步执行后再执行
- 注意
  - 在类中定义这两个函数式this的指向
    - 因为是在外部调用的

```
// ES5中定义模块
(function(window) {
    function MyPromise(excutor) {
        const that = this;
        this.status = 'pending';
        this.data = undefined;
        this.callbacks = [];
        // 由于resolve是在外调用的,所以this对象是window
        // 利用一个that来使其指向对象
        function resolve(value) {
            // 如果当前状态不是pending, 直接结束
            if(that.status !== 'pending')
                return;
            // 将状态改为resolved
            that.status = 'fulfilled'
            // 保存value数据
            that.data = value;
            // 如果有待执行callback函数, 立即异步执行回调函数'
            // 为了放入异步执行的队列,使用setTimeout
            setTimeout(() => {
                if(that.callbacks.length > 0) {
                    that.callbacks.forEach(callbackObj => {
                        callbackObj.onResolved(that.data);
                    })
                }

            })
        }
        function reject(reason) {
             // 如果当前状态不是pending, 直接结束
            if(that.status !== 'pending')
                return;
            that.status = 'rejected';

            that.data = reason;

            setTimeout(() => {
                that.callbacks.forEach(callbackObj => {
                    callbackObj.onRejected(that.data);
                })
            })
        }
        // 立即执行excutor
        // 抛出异常,利用catch捕获异常
        // 由于传进来的excutor是一个函数, 所以直接调用即可
        try {
            excutor(resolve, reject);
        }
        catch(error) {
            reject(error)
        }


    }

    // 返回一个成功的指定结果promise
    MyPromise.resolve = function(value) {
        return new MyPromise((resolve, reject) => {
            if(value instanceof MyPromise) {
                // 当是MyPromsie的类型时, 根据promise的结果来确定当前新的promise的状态
                value.then(resolve, reject);
            }
            else {
                resolve(value)
            }
        })
    }
    // 返回一个失败的指定结果promise
    MyPromise.reject = function(reason) {
        return new MyPromise((resolve, reject) => {
            reject(reason)
        })
    }
    // 只有当所有的promise都成功时才成功
    MyPromise.all = function(promises) {
        const values = []; //用于存放成功的value值
        let resolveCount = 0;  //用于判断成功异步调用resolve的次数, 当次数达到和promises相同时, 就是全部成功
        return new MyPromise((resolve, reject) => {
            promises.forEach((promise, index) => {
                // 将非promise的转化为promise
                MyPromise.resolve(promise).then(
                    value => {
```

```
                    resolveCount ++;
                    values[index] = value;
                    if(resolveCount == promises.length)
                        resolve(values);
                },
                reason => {
                    reject(reason);
                    // reject可以被调用多次，但是前面在reject中不是pending直接结束
                    // 所以只是第一次调用
                }
            )
        })
    })
};
// 由第一个promise完成时决定
MyPromise.race = function(promises) {
    return new MyPromise((resolve, reject) => {
        promises.forEach((promise, index) => {
            MyPromise.resolve(promise).then(
                value => {
                    resolve(value);
                },
                reason => {
                        reject(reason);
                }
            )
        })
    })
};

// Promise原型对象的then()
// 指定成功和失败的回调函数，返回一个新的promise对象
MyPromise.prototype.then = function(onResolved, onRejected) {
    const that = this;
    if(!typeof onResolved === "function")
        onResolved = value => value;
    // 解决异常穿透的问题
    if(!typeof onRejected === "function")
        onRejected = reason => {throw reason}
    // 返回一个新的promise且promise由回调函数的执行结果确定
    return new Promise((resolve, reject) => {
        function handler(callback) {
            try {
                const result = callback(that.data);
                if(result instanceof MyPromise) {
                    // 实际上是直接执行了return的promise的值传递给新的promise的reject和resolve
                    result.then(
                        value => resolve(value),
                        reason => reject(reason)
                    )
                    // result.then(resolve, reject)
                    // 可以写成简洁版的原因，resolve在内部调用，传入的只是该函数的定义，而该函数就是promise内部已经定义的函数
                    // 里面的that指向的是当前的promise

                }
                // 利用.then取出其结果
                else {
                    resolve(result)
                }
            }
            catch(error) {
                reject(error)
            }
        }
        // 假设当前状态还是pending状态,则加入回调函数
        if(that.status === "pending") {
            that.callbacks.push({
                onResolved(value) {
                    handler(onResolved)
                },
                onRejected(value) {
                    handler(onRejected)
                }
            })
        }
        // 假设当前状态是"fulfilled", 则异步执行回调函数
        else if(that.status === "fulfilled") {
            setTimeout(() => {
                 // 决定返回promise的状态
                //  1 onResolved抛出错误:
                //  2 onResolved返回一个非promise的值
                //  3 onResolved返回一个promise对象
                //          - 需要promise异步调用完后再得到相应的结果
                handler(onResolved)
            })
        }
        else {
            setTimeout(() => {
                handler(onRejected);
            })
        }
```

```
        })


    };

    // Promise原型对象的catch()
    // 指定失败的回调函数, 返回一个新的promise对象
    MyPromise.prototype.catch = function(onRejected) {
        return this.then(undefined, onRejected)
    };


    // 向外暴露Promise函数
    window.MyPromise = MyPromise;
})(window)
```

3 async和await使用

3.1 async 函数

- async函数的返回值是一个新的promise对象
- async函数返回的promise的结果由函数执行的结果决定

```
async function fn() {
    return 1;
  }
async function fn2() {
    throw 2;
  }
async function fn3() {
    return Promise.reject(3)
}
fn().then(
    value => console.log("onResolved1: " + value),
    reason => console.log("onRejected1: " + reason)
)
fn2().then(
    value => console.log("onResolved2: " + value),
    reason => console.log("onRejected2: " + reason)
)
fn3().then(
    value => console.log("onResolved3: " + value),
    reason => console.log("onRejected3: " + reason)
)
/*
  输出结果
  onResolved1: 1
  onRejected2: 2
  onRejected3: 3
*/
```

3.2 await 表达式

- 表达式
  - 若是promise
    - 取得promise对象的value值(即成功的结果)
    - 若想得到失败的结果,只能使用try, catch
  - 若不是promise
    - 则取得表达式中本身的值

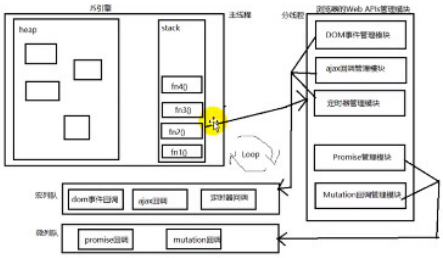
```
  async function test() {
    try {
      const value = fn3()
    }
    catch(error) {
      console.log("发生异常:" + error)
    }
  }
test();
// 发生异常:3
```

3.3 注意

- await必须写在async函数中,但async函数中可以没有await
- 如果await的promise失败,会抛出异常,需要通过try...catch进行捕获

4 宏队列和微队列

4.1 原理图



- JS中用来存储待执行的回调函数的队列包裹2个不同特定的队列
  - 宏队列
    - 用来保存代执行的宏任务
      - 定时器回调函数, DOM事件的回调函数
  - 微队列
    - 用来保存待执行的微任务
      - promise的回调函数/MutationObserver回调函数

4.2 js的异步任务的触发函数什么时候放入了对应的异步执行队列?

- 当异步调用触发时，会交给相应的线程单独去维护异步任务，等待某个时机（计时器结束、网络请求成功、用户点击DOM）完成时，然后事件触发线程将异步对应的回调函数 加入到消息队列中，回调函数在响应的消息队列中的等待被执行。

4.3 js的异步执行流程

- JS引擎首先必须先执行所有的初始化同步任务代码
- 每次准备取出第一个宏任务前,都要将所有的微任务一个一个取出执行

```
setTimeout(() => {     //会放入宏队列
    console.log("timeout callback1()");
    Promise.resolve(3).then(    //会放入微队列
        value => {
            console.log("Promise onResolved3() " + value);
        }
    )
}, 0)

setTimeout(() => { //会放入宏队列
    console.log("timeout callback2()");
}, 0)

Promise.resolve(1).then( //会放入微队列
    value => {
        console.log("Promise onResolved1() " + value);
    }
)

Promise.resolve(2).then( //会放入微队列
    value => {
        console.log("Promise onResolved2() " + value);
    }
)

/** 输出结果
 * Promise onResolved1() 1
 * Promise onResolved2() 2
 * timeout callback1()
 * Promise onResolved3() 3
 * timeout callback2()
 * /
```

5 常见面试题

5.1 认清楚哪些是异步回调函数,哪些是同步代码

```
  new Promise((resolve, reject) => {
    console.log(2); //同步代码
    resolve()
  }).then(
    value => {
      console.log(1); //异步回调函数,且插入微队列
    }
  )
```

```
async function async1() {
  console.log('async1 start');
  await async2();
  /*
  相当于
  async2().then(
    value => {
      console.log(async1 end)
    }
  )
  */
  console.log('async1 end');
}
async function async2() {
  console.log('async2');
}

console.log('script start');
setTimeout(function() {
    console.log('setTimeout');
}, 0);
async1();
new Promise(function(resolve) {
    console.log('promise1');
    resolve();
  }).then(function() {
    console.log('promise2');
});
console.log('script end');

/**
  [setTimeout]
  [promise2]
  script start
  async1 start
  async2
  async1 end
  promise1
  script end
  promise2
  setTimeout
*/
```

九 axios

1 http的理解

1.1 http请求交互的基本过程

1. 前后应用从浏览器端向服务器发送 HTTP 请求(请求报文)
2. 后台服务器接收到请求后，调度服务器应用处理请求，向浏览器端返回 HTTP 响应(响应报文)
3. 浏览器端接收到响应，解析显示响应体/调用监视回调

1.2 http请求报文

-

1.2.1 请求报文的组成

1.2.1.1 请求行

1.1.1.1.1 method(请求方法)

- 控制服务器端的增删改查
  - GET: 请求指定的页面信息，并返回实体主题 --服务端数据查询
  - POST: 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改 --服务端数据的修改/增加
  - DELETE：请求服务器删除Request-URL所标识的资源 --服务端数据的删除
  - PUT：向指定资源位置上传其最新内容 --服务端数据的增加
- 其他
  - HEAD: 向服务器索与GET请求相一致的响应，只不过响应体将不会被返回
  - TRACE: 回显服务器收到的请求，主要用于测试或诊断
  - CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

1.1.1.1.2 URL

1.1.1.1.3 协议版本

- http的版本类型

1.2.1.2 请求头

- 请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号":"分隔

1.2.1.2.1 User-Agent

- 产生浏览器类型

1.2.1.2.2 Accipt

- 客户端可识别的相应内容类型列表

1.2.1.2.3 Accept-Language

- 客户端可接收的语言类型

1.2.1.2.4 Accept-Encoding

- 客户端可接受的编码压缩格式

1.2.1.2.5 Accept-Charset

- 客户端可接受的应答的字符集

1.2.1.2.6 Host

- 请求主机名，允许多个域名同处一个IP，即虚拟主机

1.2.1.2.7 connection

- 连接方式
  - close
  - keepalive

1.2.1.2.8 Cookie

- 存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie

1.2.1.3 空行

- 最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头

1.2.1.4 请求包体

- 存不存在与方法的选择有关
- get方法中
  - 不存在请求包体
- 其他方法包含以下属性
  - Content-Type

- Content-Length

1.2.2 请求方法与请求包体Content-type以及axios的config的关系

请求方法 请求包体中的Content-type axios的config写法 发送请求的显示方式 GET 无 axios的config写法 POST application/json axios的config写法 请求体中的数据将会以json字符串的形式发送到后端 POST application/x-www-form-urlencoded axios的config写法 请求体中的数据会以普通表单形式（键值对）发送到后端 POST multipart/form-data axios的config写法 般用来上传文件，指定传输数据为二进制数据

1.3 http响应报文

1.3.1 响应报文的组成

1.3.1.1 状态行

- 状态行由 HTTP 协议版本字段、状态码和状态码的描述文本 3 个部分组成，他们之间使用空格隔开;

1.3.1.1.1 http协议字段

1.3.1.1.2 状态码

状态码	含义
1xx	表示服务器已接收了客户端请求，客户端可继续发送请求
2xx	表示服务器已成功接收到请求并进行处理;
3xx	表示服务器要求客户端重定向;
4xx	表示客户端的请求有非法内容
5xx	表示服务器未能正常处理客户端的请求而出现意外错误;

1.3.1.1.3 状态码描述文本

状态码	含义
200 OK	表示客户端请求成功;
400 Bad Request	表示客户端请求有语法错误，不能被服务器所理解;
401 Unauthonzed	表示请求未经授权，该状态代码必须与 WWW-Authenticate 报头域一起使用;
403 Forbidden	表示服务器收到请求，但是拒绝提供服务，通常会在响应正文中给出不提供服务的原因;
404 Not Found	请求的资源不存在，例如，输入了错误的URL;
500 Internal Server Error	表示服务器发生不可预期的错误，导致无法完成客户端的请求;
503 Service Unavailable	表示服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常;

1.3.1.2 响应头部

1.3.1.2.1 Location

- 用于重定向接受者到一个新的位置。
  - 例如：客户端所请求的页面已不存在原先的位置，为了让客户端重定向到这个页面新的位置，服务器端可以发回Location响应报头后使用重定向语句，让客户端去访问新的域名所对应的服务器上的资源;

1.3.1.2.2 Server

- Server 响应报头域包含了服务器用来处理请求的软件信息及其版本。
  - 它和 User-Agent 请求报头域是相对应的，前者发送服务器端软件的信息，后者发送客户端软件(浏览器)和操作系统的信息。

1.3.1.2.3 Vary

- 指示不可缓存的请求头列表

1.3.1.2.4 Vary

- 指示不可缓存的请求头列表

1.3.1.2.5 Connection

- 连接方式
- close
- keepalive

1.3.1.3 空行

1.3.1.4 响应包体

1.4 基础知识补充

1.4.1 Connection在不同报文中的作用

- 对于请求来说
  - close(告诉 WEB 服务器或者代理服务器，在完成本次请求的响应后，断开连接，不等待本次连接的后续请求了)
  - keepalive(告诉WEB服务器或者代理服务器，在完成本次请求的响应后，保持连接，等待本次连接的后续请求);
- 对于响应来说
  - close(连接已经关闭)
  - keepalive(连接保持着，在等待本次连接的后续请求);
  - Keep-Alive：如果浏览器请求保持连接，则该头部表明希望WEB 服务器保持连接多长时间(秒);例如：Keep-Alive：300;

2 服务器提供给客户端的API分类

2.1 分类依据

- 服务器提供的API端口允许客户端做CRUD操作的权限
  - 即对应的两端交互的接口不同类型请求的响应类型

2.2 类型

2.2.1 REST API（restful）

- 发送请求运行CRUD哪个操作由请求方式决定
- 同一个请求路径可以同时进行多个操作
  - 同时进行get或post
- 请求方式可以有GET/POST/PUT/DELETE

2.2.2 非 REST API(restless)

- 请求方式不决定请求的CRUD操作
- 一个请求路径只能对应一个操作
- 一般只有GET/POST

3 搭建具有REST API的简单服务器用于测试

- 网址
- 注意
  - db.json必须是在根目录下
    - db.json
      - posts: 发布文章
      - comments: 评论文章
      - profile: 我的信息

4 AJAX编程的基础--XHR

4.1. XHR的基本定义

- XMLHttpRequest（XHR）对象用于与服务器交互。通过 XMLHttpRequest 可以在不刷新页面的情况下请求特定 URL，获取数据
- XMLHttpRequest 在 AJAX 编程中被大量使用。
  - AJAX技术：向服务器请求数据但不进行页面的跳转，而只是获取其中的数据
- 显然XMLHttpRequest是异步调用

4.2 XHR的基本使用

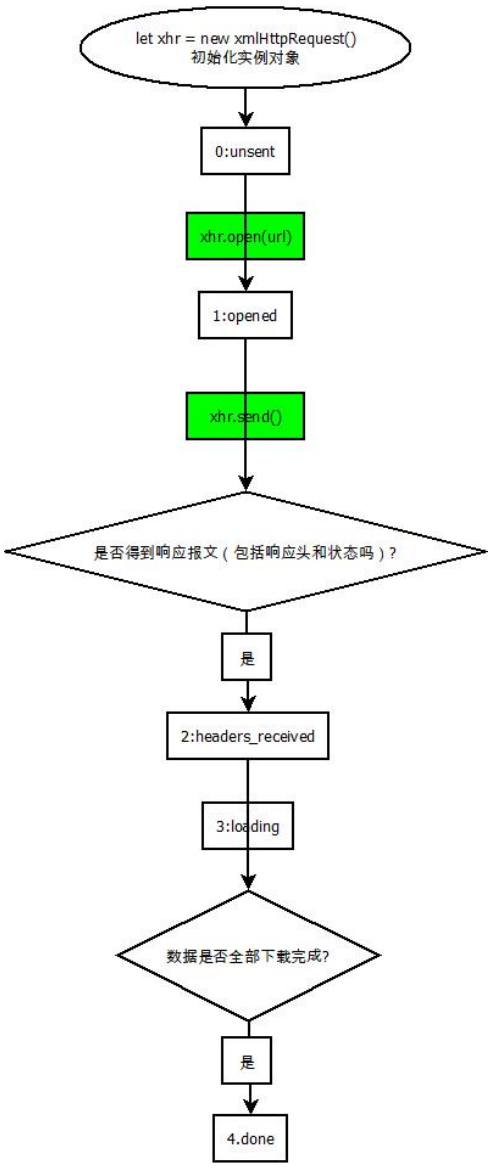
4.2.1 XHR的构造函数

- 该构造函数用于初始化一个 XMLHttpRequest 实例对象。在调用下列任何其他方法之前，必须先调用该构造函数，或通过其他方式，得到一个实例对象。

```
const request = new XmlHttpRequest()
```

4.2.3 XHR实例对象从建立到接收到数据状态以及变化

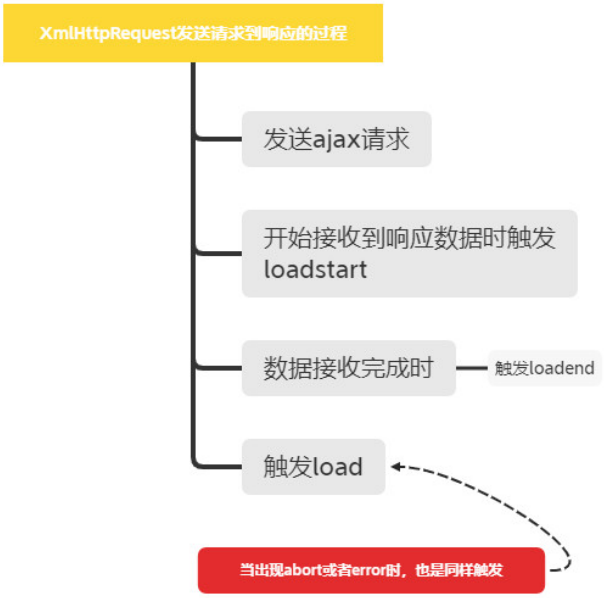
0	UNSENT	Client has been created. open() not called yet.
1	OPENED	open() has been called.
2	HEADERS_RECEIVED	send() has been called, and headers and status are available.
3	LOADING	Downloading; responseText holds partial data.
4	DONE	The operation is complete.



- 分界的操作
  - open, send

4.2.2 XHR实例对象接收数据时事件

- 事件的基本模型



loadstart	接收到响应数据时触发。
load	XMLHttpRequest请求成功完成时触发。
loadend	当请求结束时触发, 无论请求成功 (load) 还是失败 (abort 或 error)。

abort	request被中止时触发（例如调用XHR.abort()）
error	当request遭遇错误时触发。
progress	当请求接收到更多数据时，周期性地触发。
timeout	在预设时间内没有接收到响应时触发。

- 注意
  - load和loadend的区别：即使loadend只有在成功接收到了数据才会触发
  - progress: 由于接收响应的过程并不是连续的过程，每当再接收到一块数据时就会触发

4.2.3 XHR的属性和方法

4.2.3.1 与接收到响应结果相关的属性

4.2.3.1.1 XmlHttpRequest#responseType(只读)

- 用于决定响应正文的格式
- responseType取值以及对应的类型

取值	对应的类型
""(空字符串)	response采用默认类型 DOMString，与设置为 text 相同。
arraybuffer	response 是一个包含二进制数据的 JavaScript ArrayBuffer。
blob	response 是一个包含二进制数据的 Blob 对象 。
document	response 是一个 HTML Document 或 XML XMLDocument，
json	response 是一个 JavaScript 对象。这个对象是通过将接收到的数据类型视为 JSON 解析得到的。
text	response 是一个以 DOMString 对象表示的文本。

4.2.3.1.2 XmlHttpRequest#response(只读)

- 返回响应的正文，响应报文的类型由XmlHttpRequest#responseType决定

```
  var url = 'somePage.html'; //一个本地页面

  function load(url, callback) {
    var xhr = new XMLHttpRequest();

  xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
      callback(xhr.response);
    }
  }

  xhr.open('GET', url, true);
  xhr.send('');
}
```

4.2.3.1.3 XmlHttpRequest#status / statusText（只读）

- 返回响应的状态码 / 状态码的描述

4.2.3.1.4 XmlHttpRequest#responseURL(只读)

- 返回响应的序列化URL
  - 如果URL为空则返回空字符串。
  - 如果URL有锚点，则位于URL # 后面的内容会被删除。
  - 如果URL有重定向， responseURL 的值会是经过多次重定向后的最终 URL

4.2.3.1.5 XmlHttpRequest#responseText(只读)

- DOMString 是XMLHttpRequest 返回的纯文本的值。当DOMString 为null时，表示请求失败了。当DOMString 为""时，表示这个请求还没有被send()

4.2.3.1.6 XmlHttpRequest#responseXML(只读)

- 返回一个包含请求检索的HTML或XML的Document
  - responseXML 对于任何其他类型的数据以及 data: URLs 为 null。

4.2.3.1.7 XmlHttpRequest#getAllResponseHeaders()

- 以字符串的形式返回所有用 , 分隔的响应头，如果没有收到响应，则返回 null。

4.2.3.1.8 XmlHttpRequest#getResponseHeader(name)

- 返回包含指定响应头文本的字符串。
- 参数
  - 一个字符串，表示要返回的报文项名称

4.2.3.2 与状态相关的属性

4.2.3.2.1 XmlHttpRequest#readyState(只读)

- 返回一个XMLHttpRequest实例对象(client)当前所处的状态

4.2.3.2.2 XmlHttpRequest#onreadystatechange = callback

- 只要 readyState 属性发生变化，就会调用相应的处理函数
- 注意
  - 当一个 XMLHttpRequest 请求被 abort() 方法取消时，其对应的 readystatechange 事件不会被触发。
  - 不能用于同步的requests对象之中
    - 因为产生阻塞所以无法探查得到

```
var xhr = new XMLHttpRequest();
console.log('UNSENT', xhr.status);

xhr.open('GET', '/server', true);
console.log('OPENED', xhr.status);

xhr.onprogress = function () {
  console.log('LOADING', xhr.status);
};

xhr.onload = function () {
  console.log('DONE', xhr.status);
};

xhr.send(null);

/**
 * 输出如下：
 *
 * UNSENT（未发送） 0
 * OPENED（已打开） 0
 * LOADING（载入中） 200
 * DONE（完成） 200
 */
```

4.2.3.3 与请求超时相关的属性和方法

4.2.3.3.1 XmlHttpRequest#timeout

- 用于设置/读取该实例对象请求的超时时间

4.2.3.3.2 XmlHttpRequest#ontimeout = callback

- 超时触发callback

4.2.3.4 特殊的事件触发有关方法

4.2.3.4.1 XmlHttpRequest#abort()

- 如果请求已被发出，则立刻中止请求。

4.2.3.5 发送请求有关的属性和方法

4.2.3.5.1 XmlHttpRequest#open(method, url[, async[, user[, password]]])

- 初始化一个请求，并没有发送
- 参数
  - method
    - 请求方式
  - url
    - 请求地址
  - async
    - true: 异步
    - false：同步
      - send()代码会发生阻塞
  - user
    - 用户名
  - password
    - 密码

4.2.3.5.2 XmlHttpRequest#setRequestHeader(headerName, value)

- 设置HTTP请求头部的方法。
  - 此方法必须在open() 方法和 send()之间调用。如果多次对同一个请求头赋值，只会生成一个合并了多个值的请求头
- 键值对应
  - Content-Type 文本的格式

4.2.3.5.3 XmlHttpRequest#send(body)

- 向服务器发送已经初始好的ajax请求
- body

- 若method为get，则为null，因为get没有请求体
- 请求体
  - string 使用键值的方式：类似于foo=bar&lorem=ipsum
  - object 使用对象的方式
  - blob对象类型
  - Int8Array 对象类型
  - document (json等) 对象类型

```
var xhr = new XMLHttpRequest();
xhr.open("POST", '/server', true);

//发送合适的请求头信息
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

xhr.onload = function () {
    // 请求结束后,在此处写处理代码
};
xhr.send("foo=bar&lorem=ipsum");
```

5 利用XHR简单实现axios

```
/**
 * axios的特点
 *  1. 函数的返回值是一个promise，成功的结果为response，异常的返回error
 *  2. 能处理多种类型的请求：GET/POST/PUT/DELETE
 *  3. 函数的参数为一个配置对象
 *       {
 *            url       请求地址
 *            method    请求方式
 *            params    GET/DELETE请求的query参数
 *            data      POST或DELETE请求的请求体参数
 *        }
 *  4. 响应的数据专户为对象或数组：JSON.parse(request.response)
 */

// 由于传入的是一个对象，该形参运用的是对象的结构语法
function axios({
    url,
    method="GET",
    params={},
    data={}
}) {
    return new Promise((resolve, reject) => {
        // 1. 执行异步的ajax请求
        method = method.toUpperCase();
        // 创建xhr请求
        const request = XMLHttpRequest();
        // 初始化xhr
        request.open(method, url)
        // 发送请求
        switch(method) {
            case "GET" || "PUT":
                // 处理query形式的参数，并拼接到url上面 id=1&...
                let queryString = '';
                Object.keys(params).forEach(key => {
                    queryString += `${key}=${params[key]}&`
                })
                if(queryString){
                    queryString = queryString.substr(0, queryString.length - 1);
                    url += '?' + ueryString;
                }
                request.send(null);
                break;
            case "POST" || "DELETE":
                // 告诉请求体是json格式
                request.setRequestHeader("Content-Type", "application/json;charset=utf-8")
                request.send(JSON.stringify(data))
                break;
        }
        // 绑定状态改变的监听来判断运行结果
        request.onreadystatechange = function() {
            if(request.readyState !== 4)
                return;
            //  如果状态码在200-300之间，代表成功，否则失败
            const {status, statusText} = request;
            // 2.1 如果请求成功，调用resolve()

            // 2.2 如果请求失败,调用reject()
            if(status >= 200 && status < 300) {
                const response = {
                    data: JSON.parse(request.response),
                    status,
                    statusText
                }
                resolve(response);
            }

            else{
                reject(new Error("request error status is" + status))
            }
        }

    })
}
```

6 axios的使用

6.1 axios的基本定义

- 是一个基于 promise 的ajax请求库，可以用在浏览器和node.js 中。

6.2 axios的特征

- 从浏览器中创建XMLHttpRequests
- 从 node.js 创建 http 请求
- 支持 Promise API（基于promise的异步ajax请求库）
- 支持拦截请求和响应

- 转换请求数据和响应数据（就是json.parse)
- 支持取消请求
- 自动转换 JSON 数据
- 客户端支持防御 XSRF

6.3 axios的基本使用

6.3.0 ajax请求和普通的http请求的区别

- 对于服务端，ajax请求和普通的http请求没有区别，区别在于浏览器
- 浏览器发送请求时
  - ajax请求是由浏览器中的ajax引擎进行发送
  - ajax请求是一种特殊的http请求，只有XHR或fetch发出的才是ajax请求，其他都是非ajax请求
- 浏览器接收到响应报文时
  - 一般请求：浏览器一般会直接显示响应体的数据，也就是我们所说的刷新/跳转页面
  - ajax请求：浏览器不会对界面进行更新操作，只是调用监视的回调函数并传入响应的相关数据

6.3.1 axios的config对应的参数和含义

6.3.2 axios发送请求的三种不同方式

6.3.2.1 将axios视为一个对象去向服务器端发送请求

```
  //发送get请求
axios.get(url[, config])
// 发送delete请求
axios.delete(url[, config])
// 发送post请求
axios.post(url[, data[, config]])
// 发送put请求
axios.put(url[, data[, config]])
```

6.3.2.1 axios.get(url[,config])

- 向服务端查询数据，返回类型为Promise类型
- 参数
  - config
    - 是一个对象
- 由于get方式没有请求体，所以在进行查询时需要设置config的params
  - params是一个键值的对象

```
// 为给定 ID 的 user 创建请求
axios.get('/user?ID=12345')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });

// 可选地，上面的请求可以这样做
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

6.3.2.2 axios.post(url ,data)

- 向服务器端修改数据或者增加数据，返回一个Promise
- 由于post有请求体，所以可以直接传入data，而不需要设置config中的params

```
  axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

6.3.2.3 axios.put(url ,data)

- 对服务器中相应的id数据进行修改或者更新，返回一个Promise类型
- 由于put有请求体，所以可以直接传入data，而不需要设置config中的params

```
axios.put('/user/1', {
  firstName: 'Fred',
  lastName: 'Flintstone'data
})
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
});
```

6.3.2.2 将axios视为一个函数去向服务器端发送请求

- 缺点：只能有一个异步的ajax请求

6.3.3.1 axios(config)

- 返回一个promise对象

```
axios({
  method: "post",
  url: "xxx"
  // ...
})
```

6.3.2.3 将axios视为一个构造函数去向服务器端发送请求

6.3.2.3.1 axios.create(config)

- 返回一个axios的实例

```
// 一般用于首先配置默认请求
const instance = axios.create({
  baseURL: 'https://...'
})
// 使用instance发送请求
instance({
  url:'/xxx'
})
```

6.3.2.3.2 axios实例的方法

- 和全局的axios没有区别

```
axios#request(config)
axios#get(url[, config])
axios#delete(url[, config])
axios#head(url[, config])
axios#options(url[, config])
axios#post(url[, data[, config]])
axios#put(url[, data[, config]])
axios#patch(url[, data[, config]])
axios#getUri([config])
```

6.3.3 axios的全局配置

6.3.3.1 axios.defaults.configKey = value

- 设置全局axios的配置

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

6.3.4 axios的拦截器

6.3.4.1 axios.interceptors.request.use(func(config) {}, function(error) {})

- 在ajax请求发送之前对config做一些处理

```
axios.interceptors.request.use(function (config) {
  // Do something before request is sent
  return config;
}, function (error) {
  // Do something with request error
  return Promise.reject(error);
});
```

- 为什么需要返回config
  - 在axios中，拦截器和request请求是以链的形式串联起来的，若不返回config，则返回undefine，相当于向下一个函数传递了config为undefined，所以会报错

6.3.4.2 axios.interceptors.response.use(func(response) {}, function(error) {})

- 在axios获得response对response进行一些处理

```
// Add a response interceptor
axios.interceptors.response.use(function (response) {
    // Any status code that lie within the range of 2xx cause this function to trigger
    // Do something with response data
    return response;
  }, function (error) {
    // Any status codes that falls outside the range of 2xx cause this function to trigger
    // Do something with response error
    return Promise.reject(error);
  });
```

- 为什么需要返回response
  - 在axios中，拦截器和request请求是以链的形式串联起来的，若不返回response，则返回undefine，相当于向下一个函数传递了response为undefined，所以会报错

6.3.4.3 拦截器的运行流程

```
const axios = require("axios")
// 必须return config和reponse，否则报错
//请求拦截器
axios.interceptors.request.use(
    config => {
        console.log("request interceptor1 onResolved");
        return config;
    },
    error => {
        Promise.reject(error)
    }
)
axios.interceptors.request.use(
    config => {
        console.log("request interceptor2 onResolved");
        return config;
    },
    error => {
        Promise.reject(error)
    }
)
// 响应拦截器
axios.interceptors.response.use(
    response => {
        console.log("request interceptor3 onResolved");
        return response;
    },
    error => {
        Promise.reject(error)
    }
)
axios.interceptors.response.use(
    response => {
        console.log("request interceptor4 onResolved");
        return response;
    },
    error => {
        Promise.reject(error)
    }
)
axios.get("http://localhost:3000/posts").then(
    response => {
        console.log("取得的数据为" + response.data)
    }
)
/**
 *  结果
 * request interceptor2 onResolved
 * request interceptor1 onResolved
 * request interceptor3 onResolved
 * request interceptor4 onResolved
 * 取得的数据为[object Object],[object Object]
 */
```

- 一般的执行流程：
  - 请求拦截器 -> 异步request请求 -> 响应拦截器 -> then的回调函数
- 请求拦截器
  - 栈
- 响应拦截器
  - 队列

6.3.5 axios的reponse

- 回调函数中的reponse参数的结构

```
  {
  // `data` 由服务器提供的响应
  data: {},

  // `status` 来自服务器响应的 HTTP 状态码
  status: 200,

  // `statusText` 来自服务器响应的 HTTP 状态信息
  statusText: 'OK',

  // `headers` 服务器响应的头
  headers: {},

  // `config` 是为请求提供的配置信息
  config: {}
}
```

6.3.6 axios的取消

- 作用：取消未完成的请求

6.3.6.1 方法1 使用内部的函数

```
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

axios.get('/user/12345', {
  cancelToken: source.token
}).catch(function (thrown) {
  if (axios.isCancel(thrown)) {
    console.log('Request canceled', thrown.message);
  } else {
    // handle error
  }
});

axios.post('/user/12345', {
  name: 'new name'
}, {
  cancelToken: source.token
})

// cancel the request (the message parameter is optional)
source.cancel('Operation canceled by the user.');
```

6.3.6.2 方法2：创建一个对象

```
const CancelToken = axios.CancelToken;
let cancel;

axios.get('/user/12345', {
  cancelToken: new CancelToken(function executor(c) {
    // An executor function receives a cancel function as a parameter
    cancel = c;
  })
});

// cancel the request
cancel();
```

6.3.6.3 应用

6.3.6.3.1 用于请求当前的请求

```
  if(typeof cancel === 'function')
    cancel();
  // 之后在then的回调函数中
  // 将cancel设置为null，若已经执行好的，cancel为null,所以不会再执行
```

6.3.6.3.2 用于请求上一个请求

```
let cancel = c;
btn1.onclick = function() {
  if(typeof cancel === "function")
    cancel("取消上一个请求")
  axios({
    url:"url1",
    method: "get",
    cancelToken: new cancelToken(c => {
      cancel = c;
    })
  }).then((resolve, reject) => {
    response => {
      ...
      cancel = null;
    },
    error => {
      if(isCancel(error)) {
        ...
      }
      else{
        ...
        cancel = null;
      }
    }
  })
}
btn2.onclick = function() {
  if(typeof cancel === "function")
    cancel("取消上一个请求")
  axios({
    url:"url2",
    method: "get",
    cancelToken: new cancelToken(c => {
      cancel = c;
    })
  }).then((resolve, reject) => {
    response => {
      ...
      cancel = null;
    },
    error => {
      if(isCancel(error)) {
        ...
      }
      else{
        ...
        cancel = null;
      }
    }
  })
}
```

- 为什么在reject的回调函数中要分情况，而isCancel不能赋值为null？
  - 异步函数的机制是浏览器将异步任务交给操作系统中的线程去执行，当达到触发条件时，在将其加入宏队列或者微队列中执行
  - 执行的步骤
    - 在执行完全部同步代码后，当用户点击btn1时，触发了点击事件，执行了axios的同步代码，将cancel赋值为函数之后发出请求
    - 当用户点击了btn2时，触发了btn2事件，于是将其回调函数加入了宏队列，此时执行了其回调函数中的同步代码，取消btn1中的请求，同时将cancel赋值为btn2请求的cancel函数。由于请求被取消，所以触发了axios取消的事件
      - 若此时部分情况将cancel赋值为null，则会将第二个请求的cancel函数赋值为null，进而使得后面的取消无效
    - 重点是理解同步任务和异步任务是如何执行的，执行的顺序是什么

7 axios源码分析

7.1 axios与Axios的关系

- 从语法上来说
  - axios不是Axios的实例
- 从功能上来说
  - axios是Axios的实例
  - axios既有Axios原型上的属性和方法，又有Axios实例对象的属性和方法
- axios是Axios.prototype.request函数bind()返回的函数
- axios作为对象有Axios原型对象上的所有方法，有Axios对象的所有属性
- 源码

```
  /**
 * Create an instance of Axios
 *
 * @param {Object} defaultConfig The default config for the instance
 * @return {Axios} A new instance of Axios
 */
function createInstance(defaultConfig) {
  // 1. 创建了一个Axios的实例
  var context = new Axios(defaultConfig);
  /*
  module.exports = function bind(fn, thisArg) {
    return function wrap() {
      var args = new Array(arguments.length);
      for (var i = 0; i < args.length; i++) {
        args[i] = arguments[i];
      }
      return fn.apply(thisArg, args);
    };
  };
  */

    // 2.1 instance是一个函数
    // 2.2 当instance被调用的时候，Axios.prototype.request的this会指向context对象（即context拥有了其所有属性和方法），并且由context来执行request
    // 2.3 该方法实现了axios(config)时执行了request
  var instance = bind(Axios.prototype.request, context);
  // Copy axios.prototype to instance
  // 3.1 Axios.prototype对象中的属性和方法绑定给instance，可以发现get，post等实际上也是使用了request
  // 3.2 该方法实现了axios.get(),axios.request()等方法
  utils.extend(instance, Axios.prototype, context);

  // 4.1 Copy context to instance
  // 4.2 使得axios是Axios的一个实例，具有default和interceptor的特殊属性
  utils.extend(instance, context);

  return instance;
  // instance是Axios的一个实例，且有Axios原型对象的属性和方法，而且又是一个函数，可以直接调用原型对象中的request方法
}
```

- axios的指向

```
  var axios = createInstance(defaults);
```

7.2 axios与instance的区别

```
  const instance = axios.create(config)
```

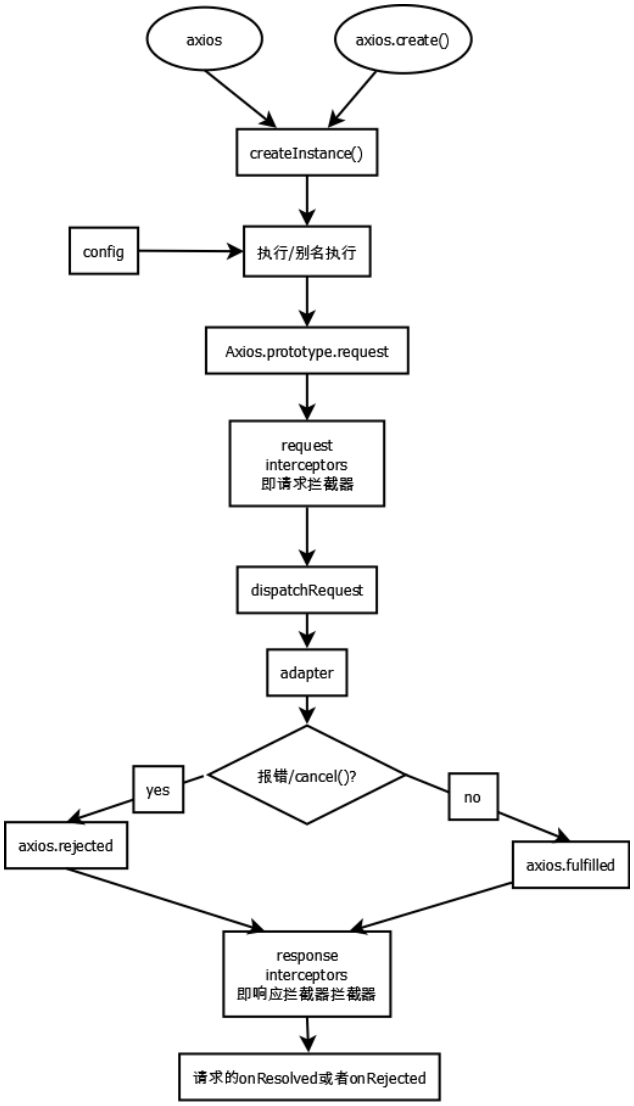
- 相同点
  - 都是一个能发送给任意请求的函数:request(config)
  - 都有发特定请求的各种方法get()/post()/put()/delete()
  - 都有默认配置和拦截器属性defaluts/interceptor
- 不同
  - 默认匹配值很可能不一样
  - instance后面没有添加一些方法
    - create()/CancelToken()/all
- 源码

```
// Factory for creating new instances
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};
```

- axios特有的属性和方法

```
axios.Axios = Axios;

// Factory for creating new instances
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};

// Expose Cancel & CancelToken
axios.Cancel = require('./cancel/Cancel');
axios.CancelToken = require('./cancel/CancelToken');
axios.isCancel = require('./cancel/isCancel');

// Expose all/spread
axios.all = function all(promises) {
  return Promise.all(promises);
};
axios.spread = require('./helpers/spread');
```

7.3 axios执行的流程图



- request(config)
  - 将请求拦截器/dispatchRequest()/响应拦截器通过promise链串连起来, 返回promise
  - dispatchRequest()
    - 转换请求数据，调用xhrAdapter() => 请求返回后转换响应函数数据，返回promise
    - xhrAdapter()
      - 创建xhr对象，根据config进行相应的设置，发送特定的请求，并接收响应的数据，返回promise。
- request(config)源码了解

```
Axios.prototype.request = function request(config) {
  /*eslint no-param-reassign:0*/
  // Allow for axios('example/url'[, config]) a la fetch API
  if (typeof config === 'string') {
    config = arguments[1] || {};
    config.url = arguments[0];
  } else {
    config = config || {};
  }

  config = mergeConfig(this.defaults, config);

  // Set config.method
  // 1. 判断请求的类型
  if (config.method) {
    config.method = config.method.toLowerCase();
  } else if (this.defaults.method) {
    config.method = this.defaults.method.toLowerCase();
  } else {
    config.method = 'get';
  }

  // 2. 创建保存请求/响应拦截回调函数的数组
  /*
  数组的中间为发送请求的函数
  数组的左边为请求拦截器的回调函数（成功或失败）
  数组的右边为响应拦截器的回调函数
  */
  var chain = [dispatchRequest, undefined];
  var promise = Promise.resolve(config);

  // 3. 后添加的请求拦截器回调函数加入到数组的前面
  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });
  // 4. 后添加的响应拦截器回调函数加入到数组的后面
  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });
  // 5. 通过promise的then()串联起所有的请求拦截器/请求方法/响应拦截器
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift());
  }
  // 返回用来指定我们的onResolved和onRejected的promise
  return promise;
};
```

- dispatchRequest()源码了解

```
/*
  transformRequest: [function transformRequest(data, headers) {
    normalizeHeaderName(headers, 'Accept');
    normalizeHeaderName(headers, 'Content-Type');
    if (utils.isFormData(data) ||
      utils.isArrayBuffer(data) ||
      utils.isBuffer(data) ||
      utils.isStream(data) ||
      utils.isFile(data) ||
      utils.isBlob(data)
    ) {
      return data;
    }
    if (utils.isArrayBufferView(data)) {
      return data.buffer;
    }
    // 如果是Params数据，则转化为urlencoded类型
    if (utils.isURLSearchParams(data)) {
      setContentTypeIfUnset(headers, 'application/x-www-form-urlencoded;charset=utf-8');
      return data.toString();
    }
    //// 如果是对象，则转换为json类型的数据
    if (utils.isObject(data)) {
      setContentTypeIfUnset(headers, 'application/json;charset=utf-8');
      return JSON.stringify(data);
    }
    return data;
  }],
 */
module.exports = function dispatchRequest(config) {
  throwIfCancellationRequested(config);
  // 1. 发请求前对设置，数据等进行处理
  // Ensure headers exist
  config.headers = config.headers || {};

  // Transform request data
  config.data = transformData(
    config.data,
```

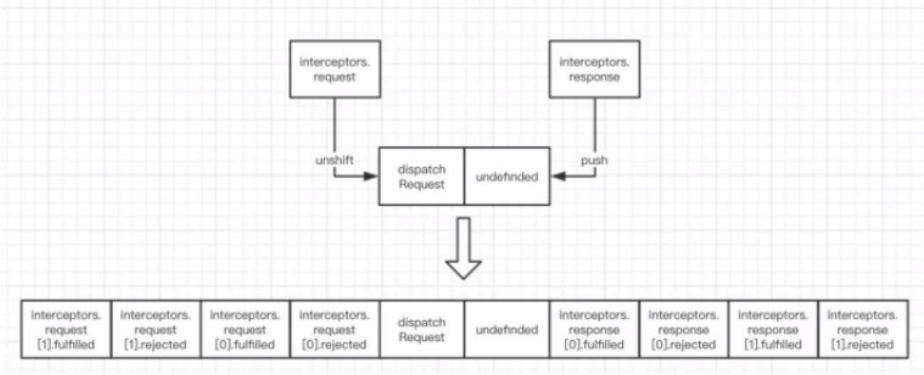
```
    config.headers,
    // 1. 对请求头和类型进行转化
    config.transformRequest

  );

  // Flatten headers
  config.headers = utils.merge(
    config.headers.common || {},
    config.headers[config.method] || {},
    config.headers
  );

  utils.forEach(
    ['delete', 'get', 'head', 'post', 'put', 'patch', 'common'],
    function cleanHeaderConfig(method) {
      delete config.headers[method];
    }
  );

  var adapter = config.adapter || defaults.adapter;
  // 2. 发送请求
  return adapter(config).then(function onAdapterResolution(response) {
    throwIfCancellationRequested(config);
    /*
    transformResponse: [function transformResponse(data) {
    if (typeof data === 'string') {
      try {
        data = JSON.parse(data);
      } catch (e) { }
    }
      return data;
    }],
    */

    // 3. 对请求的数据的格式进行转换
    response.data = transformData(
      response.data,
      response.headers,
      config.transformResponse
    );

    return response;
  }, function onAdapterRejection(reason) {
    if (!isCancel(reason)) {
      throwIfCancellationRequested(config);

      // Transform response data
      if (reason && reason.response) {
        reason.response.data = transformData(
          reason.response.data,
          reason.response.headers,
          config.transformResponse
        );
      }
    }

    return Promise.reject(reason);
  });
};
```

- xhrAdapter()源码了解

```
module.exports = function xhrAdapter(config) {
  return new Promise(function dispatchXhrRequest(resolve, reject) {
    var requestData = config.data;
    var requestHeaders = config.headers;

    if (utils.isFormData(requestData)) {
      delete requestHeaders['Content-Type']; // Let the browser set it
    }

    if (
      (utils.isBlob(requestData) || utils.isFile(requestData)) &&
      requestData.type
    ) {
      delete requestHeaders['Content-Type']; // Let the browser set it
    }
    // 1. 创建xml的对象
    var request = new XMLHttpRequest();

    // HTTP basic authentication
    if (config.auth) {
      var username = config.auth.username || '';
      var password = unescape(encodeURIComponent(config.auth.password)) || '';
      requestHeaders.Authorization = 'Basic ' + btoa(username + ':' + password);
    }

    var fullPath = buildFullPath(config.baseURL, config.url);
```

```
  // 2.配置xml的对象的参数
  request.open(config.method.toUpperCase(), buildURL(fullPath, config.params, config.paramsSerializer), true);

  // Set the request timeout in MS
  request.timeout = config.timeout;

  // Listen for ready state
  // 3. 设置状态监听器来监视党情的状态
  request.onreadystatechange = function handleLoad() {
    // 3.1 如果状态不为4
    if (!request || request.readyState !== 4) {
      return;
    }

    // The request errored out and we didn't get a response, this will be
    // handled by onerror instead
    // With one exception: request that using file: protocol, most browsers
    // will return status as 0 even though it's a successful request
    if (request.status === 0 && !(request.responseURL && request.responseURL.indexOf('file:') === 0)) {
      return;
    }

    // Prepare the response
    // 3.2 接收响应的数据
    var responseHeaders = 'getAllResponseHeaders' in request ? parseHeaders(request.getAllResponseHeaders()) : null;
    var responseData = !config.responseType || config.responseType === 'text' ? request.responseText : request.response;
    var response = {
      data: responseData,
      status: request.status,
      statusText: request.statusText,
      headers: responseHeaders,
      config: config,
      request: request
    };

    // 3.3 根据reponse的status来确定请求是成功还是失败
    settle(resolve, reject, response);

      /*
      validateStatus: function validateStatus(status) {
        return status >= 200 && status < 300;
      }
      // 若status位于200~300之间，则证明请求是成功的
      */
    /*
    function settle(resolve, reject, response) {
      var validateStatus = response.config.validateStatus;

      if (!response.status || !validateStatus || validateStatus(response.status)) {
        resolve(response);
      } else {
        reject(createError(
          'Request failed with status code ' + response.status,
          response.config,
          null,
          response.request,
          response
        ));
      }
    */
    // Clean up request
    request = null;
  };

  // Handle browser request cancellation (as opposed to a manual cancellation)
  // 3.4 绑定中断的事件
  request.onabort = function handleAbort() {
    if (!request) {
      return;
    }

    reject(createError('Request aborted', config, 'ECONNABORTED', request));

    // Clean up request
    request = null;
  };
.................
// 4. 发送请求
request.send(requestData);
```

7.4 axios如何把interceptor和request串联起来

- 


```
Axios.prototype.request = function request(config) {
  /*eslint no-param-reassign:0*/
  // Allow for axios('example/url'[, config]) a la fetch API
  if (typeof config === 'string') {
    config = arguments[1] || {};
    config.url = arguments[0];
  } else {
    config = config || {};
  }

  config = mergeConfig(this.defaults, config);

  // Set config.method
  // 1. 判断请求的类型
  if (config.method) {
    config.method = config.method.toLowerCase();
  } else if (this.defaults.method) {
    config.method = this.defaults.method.toLowerCase();
  } else {
    config.method = 'get';
  }

  // 2. 创建保存请求/响应拦截回调函数的数组
  /*
  数组的中间为发送请求的函数
  数组的左边为请求拦截器的回调函数（成功或失败）
  数组的右边为响应拦截器的回调函数
  */
  var chain = [dispatchRequest, undefined];
  var promise = Promise.resolve(config);

  // 3. 后添加的请求拦截器回调函数加入到数组的前面
  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });
  // 4. 后添加的响应拦截器回调函数加入到数组的后面
  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });
  // 5. 通过promise的then()串联起所有的请求拦截器/请求方法/响应拦截器
  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift());
  }
  // 返回用来指定我们的onResolved和onRejected的promise
  return promise;
};
```

- 请求拦截器后添加先执行

7.5 axios是如何取消request的请求的

- 通过创建一个cancelToken的对象进行取消
- 源码
  - 在xml函数中有abort()方法来取消请求

```
// 若配置中存在cancelToken则调用request.abort();
if (config.cancelToken) {

      // Handle cancellation
      config.cancelToken.promise.then(function onCanceled(cancel) {
        if (!request) {
          return;
        }

        request.abort();
        reject(cancel);
        // Clean up request
        request = null;
      });
    }

    if (!requestData) {
      requestData = null;
    }
```

```
function CancelToken(executor) {
  if (typeof executor !== 'function') {
    throw new TypeError('executor must be a function.');
  }

  // 通过一个全局的resolve，使的可以在外部调用当前的promise
  var resolvePromise;
  this.promise = new Promise(function promiseExecutor(resolve) {
    resolvePromise = resolve;
  });
  //保存当前的对象为token
  var token = this;
  // 执行器：
  // 1. 为外面的形参定义成一个函数，以便于外面的函数进行调用
  // 2. 执行外面传入的函数体
  executor(function cancel(message) {
    // 即若是调用cancel则会将其变为resolve
    // 如果有reason，则证明已经取消完成了
    if (token.reason) {
      // Cancellation has already been requested
      return;
    }
    // 将token的reason指定为一个Cancel对象
    token.reason = new Cancel(message);
    // 成功的回调，成功才调用cancel
    resolvePromise(token.reason);
  });
}
CancelToken.prototype.throwIfRequested = function throwIfRequested() {
  if (this.reason) {
    throw this.reason;
  }
};

/**
 * Returns an object that contains a new `CancelToken` and a function that, when called,
 * cancels the `CancelToken`.
 */
CancelToken.source = function source() {
  var cancel;
  var token = new CancelToken(function executor(c) {
    cancel = c;
  });
  return {
    token: token,
    cancel: cancel
  };
};
```

- 当配置了cancelToken对象时，保存cancel函数
  - 创建一个用于将来中断请求的cancelPromise
  - 并定义了一个用于取消请求的cancel函数
  - 将cance函数传递出来
- 调用cancel()取消请求
- 调用cancel()取消请求
  - 执行cancel函数，传入错误信息message
  - 内部会让cancelPromise变为成功，且成功的值为一个cancelPromise
  - 在cancelPromise的成功回调中断请求，并让发请求的promise失败，失败的reason为Cancel对象

十 ES6增加的内容总结

1. 类的新的定义方式

1.1. class

```
class Foo {
  constructor(a, b) {
    this.x = a;
    this.y = b;
  }

  gimmeXY() {
    return this.x * this.y;
  }
}
```

- class Foo表明创建一个具名函数
- constructor为构造函数，在new的时候会执行
- 类的方法可以使用字面量的简洁语法，类的方法默认是不可以枚举的。
- 内部不需要逗号来分割成员

1.1.1. extends和super

1.1.1.1 extends

- ES6类通过extends提供了一个语法糖，用来在两个函数原型之间建立[[Prototype]]委托链接，被误称为"继承"或表示为"原型继承"

```
class Father {
  // ...
}
class Son extends Father {
  // ...
}
```

1.1.1.2 super

- 功能与位置有关
  - 在构造器中，super表示父类的构造函数（new father())
    - 必须在前面进行调用
  - 在方法中，super表示父类的原型father.prototype

```
class Foo {
  constructor() {
    this.a = 1;
  }
}

class Bar extends Foo {
  constructor() {
    super();
    this.b = 2;
    // super(); 报错，必须放在第一行
  }
}
```

1.1.2 new.target

- 指向new实际上直接调用的构造器, 与继承无关
  - 其他函数的new.target则为undefined

```
class Foo {
  constructor() {
    console.log("Func: ", new.target.name)
  }
}

class Bar extends Foo {
  constructor() {
    super();
    console.log("Func: ", new.target.name)
    // super(); 报错，必须放在第一行
  }
}

var a = new Foo();
// Func:  Foo
var b = new Bar();
// Func:  Bar
// Func:  Bar
```

1.1.3 static

- 直接定义的函数是连接到函数的原型
- static定义的函数并不在类的原型链上，而是在函数构造器之间的双向/并行链上
  - 无法通过对象的实例来访问，但可以通过函数名来访问

```
class Foo {
  static cool() {
    console.log("cool");
  }
  wow() {
    console.log("wow")
  }
};

class Bar extends Foo {
  static awesome() {
    super.cool();
    console.log("awesome");
  }

  neat() {
    super.wow();
    console.log("neat");
  }
}

Foo.cool();
// cool

Bar.cool();
// cool 访问到的是Foo.cool
Bar.awesome();
// cool awesome
// 为什么能够访问到?
/*
  static成员能用函数名称访问到，所以访问aewsome
  为什么可以访问super.cool()?
  cool()也是static，可以用函数名访问到
*/
var b = new Bar();
b.neat();
// wow neat

console.log(b.awesome);
// undefined
console.log(b.cool)
// undefined
```

2 块级作用域

- {...}代表一个块级作用域

2.1 let

2.1.1. let 关键字的特点

- 块级作用域
- 不存在变量提升
- TDZ错误（暂时性死区错误）

```
{
  console.log(a); // 1
  var a = 2;
}
console.log(a)    // 2
// undefined
// 2
// 1 说明a已经被声明，所以var有变量提升
// 2 说明var不存在块级作用域
{
  console.log(b);
  let b = 2;    // 3
}
  console.log(b)  // 4
  // Reference Error
  // Reference Error
  // 3 说明let不存在变量提升，此时为TDZ错误
  // 4 说明let为块级作用域，块外无法访问块内的变量定义
```

2.1.2 let 关键字与for

- 在ES6的for循环中，计数变量一般使用let

```
var arr = [];
for(var i = 0;i < 2;i ++){
  arr[i] = function() {
    console.log(i);
  }
}
arr[0]();
arr[1]();
// 2 2
var arr = [];
for(let i = 0;i < 2;i ++){
  arr[i] = function() {
    console.log(i);
  }
}
arr[0]();
arr[1]();
// 0 1
```

2.2 const

2.2.1 特点

- 具有块级作用域
- 没有变量提升
- 声明常量时必须赋值
- 常量赋值后，值不能修改
  - 基本数据类型
    - 字面值不能修改
  - 复合数据类型
    - 对应的存储地址不能改，但结构内的内容是可以改的
    - 理解
      - const指的是不能修改内存中对应栈的值，而复杂数据类型中对应栈的值存储的是对应堆中的数据的地址，即数据实际上是存在内存的堆中，所以可以修改，但是复杂数据类型不能重新赋值

2.3 块级作用域函数(有待查证)

- 在ES6前，函数声明不存在块级作用域
- 在ES6后，函数的声明具有块级作用域，同时还有变量提升

```
{
  function foo() {
    console.log("A");
  }
  foo();
}
foo();
// A
// ReferenceError
```

3 函数的参数的默认值

- 在ES6前，函数的参数默认值需要通过||来使得参数取到默认值
- 使用参数值的注意事项
  - 当对应实参为空或者是undefined时，使用默认的参数
  - 当对应的实参为null时，会强制转化为0
- 函数的参数可以看作一个作用域

```
// ES6前
  function foo(x, y) {
    x = x != undefined ? x : 31;
    y = y != undefined ? y : 1;
    console.log(x + y);
  }
  foo(undefined, 3);
  // 34
// ES6
  function foo(x = 31, y = 1) {
    console.log(x + y);
  }
    foo(undefined, 3);
  // 34
```

4 对象字面量的扩展

- 注意是用于对象中的属性和方法定义

4.1 简洁属性(key值和value的变量名相同)

- 前提：key值和value的变量名相同

```
  var x = 2, y = 3;
// 非简洁写法：
  var o = {
    x: x,
    y: y
  }
// 间接写法
  var o = {
    x,
    y
  }
```

4.2 简洁方法

- 注意：
  - 简洁方法对应的是函数未命名的方法，即x,y,z并不是它们函数的名称，而是它们的key值
    - 无法直接使用函数名称来进行调用，需要在前面增加对象名或者this进行调用
      - o.x()
      - this.x()
  - 如果要使用递归的方法，最好采用有有名称函数的调用。

```
// 老方法
  var o = {
    x: function() {
      // ...
    },
    y: function() {
      // ...
    }
    z: *function() {
      // ...
    }
  }
// ES6
  var o = {
    x() {
      // ...
    },
    y() {
      // ...
    }
    * z() {
      // ...
    }//生成器
  }
```

```
  var o = {
    x: function x(val) {
      if(val < 0)
        return 0;
      else {
        return val + x(val - 1)
      }
    }
  }
  // 函数内的x函数指的就是本身
```

4.3 使用计算属性名

- 在ES6以前，若想要以一个计算属性名作为key，则需要单独累出
- ES6提供了可以在对象字面量内部使用定义计算属性名
  - []内部使用计算表达式作为属性值，也可以是计算属性值([Symbol.iterator])

```
// ES6之前
  var prefix = "user_";
  var o = {
  baz: function(..){ .. }
  };
  o[ prefix + "foo" ] = function(..){ .. };
  o[ prefix + "bar" ] = function(..){ .. };
// ES6
  var prefix = "user_";
  var o = {
  baz: function(..){ .. },
    [ prefix + "foo" ]: function(..){ .. },
    [ prefix + "bar" ]: function(..){ .. }
  };
```

4.4 关联原型

- Object.setPrototypeOf(son, father)
  - 使得son的原型为father
  - 用于事件的委派

```
var o1 = {
  // ...
}
var o2 = {
  // ...
}
Object.setPrototypeOf(o2, o1);
// o2.__proto__ = o1 类似于o1将事件委托给了o2
```

4.5 super对象

- super对象只允许在简洁方法中出现，而不允许在普通函数表达式中出现
  - super代指原型链的对象，常用于调用原型链中的函数（同名或者非同名）

```
var o1 = {
  foo() {
    console.log("o1.foo");
  };
}
var o2 = {
  foo() {
    super.foo();
    console.log("o2.foo");
  }
}
```

5 解构赋值

- 方便从数组，对象中提取值
- 特征
  - 数组或对象的字面量写在赋值函数的前面（左边）
  - 数组或对象的字面量作为形参

5.1 数组解构

- 变量的位置和数组值的位置一一对应

```
let arr = [1, 3, 4]
let [a, b, c] = arr;
// 1，  3，4
```

5.2 对象解构

- 从被解构对象中找到与key值相匹配的value值，并进行赋值

```
// 简洁属性的方法
  let person = {name: "bulumrcai", age: 23, height: 175}
  let {age, name, height} = person;
  //age 23 name bulumrcai height 175

// 正常匹配的方法
  let person = {name: "bulumrcai", age: 23, height: 175}
  let {name: myname} = person;
  // myname: bulumrcai
  // myname实际上是类似于重新命名为。。。
```

5.3 解构赋值的两种写法

5.3.1 用于解构的变量未声明变量未声明

```
let arr = [1, 3, 4]
let [a, b, c] = arr;

let person = {name: "bulumrcai", age: 23, height: 175}
let {age, name, height} = person;
```

5.3.2 用于解构的变量已经声明(前面无需加声明类型)

```
let a, b,c;
let arr =[1, 3, 4];
[a, b, c] = arr;

let myName;
let person = {name: "bulumrcai", age: 23, height: 175};
{name: myName} = person;
```

5.4 用于解构变量的数量

- 太少
- 刚刚好
- 太多
  - 多出的变量会被赋值为undefined

5.5 重复赋值

- 当对象出现复数个相同的key值，会对用于解构的变量赋值

```
var a = {b: 4};
var {b: c, b: d} = a;
console.log(c, d);
// 4 4
```

5.6 解构参数（当数组或对象的字面量作为形参）

```
function f3([ x, y, ...z], ...w) {
  console.log( x, y, z, w );
}
f3( [] ); // undefined undefined [] []
f3( [1,2,3,4], 5, 6 ); // 1 2 [3,4] [5,6]
```

5.7 用于解构变量的默认值

当对应的参数的值为undefined时，会使用默认的参数

```
var arr1 = [1, 2, 3, 12];
var arr2 = [4, 5, 6];
var [a = 3, b = 6, c = 9, d = 12 ] = arr1;
var [ x = 5, y = 10, z = 15, w = 20] = arr2;
console.log(a, b, c, d);
console.log(x, y, z, w);
//  1 2 3 12
// 4 5 6 20
```

5.7.1 解构的默认值和形参的默认值（难点）

- 根据触发填充默认值的条件进行区分
  - 解构的默认值：
    - 当被解构的数组/对象中没有相应值触发
  - 形参的默认值：
    - 当形参为空或者为undefined触发

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
  console.log( x, y );
}
f6();
/**
  * 没有传入参数，则使用形参的默认值即
  * {x = 10} = {}, {y} = {y: 10}
  * 从{}无法找到x对应的value值，所以使用解构的默认值即x = 10
  * {y: 10}找到y对应的值，所以y=10
*/
f6( {}, {} );
/**
  * 传入参数{}, {}
  * {x = 10} = {}, {y} = {}
  * 从{}无法找到x对应的value值，所以使用解构的默认值即x = 10
  * {}无法找到y对应的值，又没有解构的默认值，所以y=undefined
*/
```

5.8 与其他用法的结合应用

- 使用实现不使用临时变量使两个变量的值交换

```
var x = 10, y = 20;
[y, x] = [x, y];
// 解释
// 赋值表达式右边先执行，则有新构建一个数组[10, 20];
// 左边为数组的形式，显然使用了数组的解构，所以对应位置被赋予对对应的值，即
// y = 10, x = 20;
```

- 与扩展运算符结合实现截取数组

```
var a = [1,3,4,6,7]
var [,,...b] = a;
console.log(b);
// [4, 6, 7]
```

6 展开运算符（...)

- ...后面跟上的实际上不是一个数组，而是一个iterable类型的变量
  - ...后面跟上array类型时，实际上数组要先做一步处理
    - arr[Symbol.iterator]()，利用生成器函数来生成对应的iterable变量
- rest/spread --> 展开/收集运算符

6.1 ... 用于iteratble前(展开运算符)

- 实际上是iterable类型,作用是将变量展开为各个独立的值
  - ...[1,2,4] => 1, 2, 4
- 作用
  - 用于实参的输入
  - 用于合并数组(基本代替了concat函数的作用)

```
// 用于合并数组
  var a = [2, 3, 4];
  var b = [1, ...a, 5];
  console.log(b);
  // [1, 2, 3 ,4 ,5]
```

6.2 ...用于函数的形参中（剩余参数）

- 这种参数被称为剩余参数,对应变量的返回值是一个真正的数组
  - 用于替代arguments（类数组），而...后面的形参为真正的数组

```
// 作为剩余参数，且剩余参数为真正的数组
  function sum(...args) {
    let res = args.reduce((preValue, value) => {
      return preValue + value;
    }, 0)
    return res;
  }

// 用于实参的输入
  const a = [1, 2, 3, 4, 5];
  console.log(sum(...a));
  // 15
```

- 允许我们讲一个不定数量的参数以数组的形式传递给函数

```
  function sum(...argv) {
    return argv.reduce((preValue, curValue) => preValue + curValue, 0);
  }
  console.log(sum(1,3))
  console.log(sum(1,2,4))
  // 4, 7
```

- 剩余参数和解构赋值配合使用

```
  const arr = [1, 3, 4, 5];
  const [arvg, ...list] = arr;
  console.log(arvg, list)
  // 1, [3, 4, 5]
```

7 模板字面量

- 返回值是一个字符串
- 利用`作为界定符声明字符串的字面量

- 优点
- 可以分散在多行进行字符串字面量的定义
- 字面量中字符串的换行符会被保留

7.1 插入字符串字面量

- 可以在字面量中增加${js表达式} --- 插入表达式
  - 会返回当中js表达式的返回值，并将对应的返回值加入到字符串之中
  - js表达式可以是函数的调用，在线函数表达式调用，设置插入字符串的字面量
- ${js表达式}的作用域
  - 对应的词法作用域

```
function upper(s) {
  return s.toUpperCase();
}
var who = "reader";
var text = `
A very ${upper("warm")} welcome
to all of you ${upper(`${who}s`)}!
`
console.log(text);
/*

A very WARM welcome
to all of you READERS!

*/
```

7.2. 标签模板字面量

- 一种特殊的函数调用形式

```
function funcName(strings, ...values) {
  // ...
}
// 函数调用
funcName`...`
```

- funcName...
  - 一种特殊的函数调用
    - 不需要()，而是`进行函数的调用
- 参数
  - strings: 由普通的字符串组成的数组，得到的是以插入表达式分割的字符
  - ...values: 由插入表达式结果组成的数组

```
function foo(strings, ...values) {
  console.log(strings);
  console.log(values);
}

var desc = "awesome";

foo`Everything is ${desc}!`;
```

- 常用于对模板字面量所组成的字符串进行改造
- 用于将字符串中的数字改造成美元

```
function dollabillsyall(strings, ...values) {
  return strings.reduce(function(preValue, currentValue, index) {
    if(index > 0) {
      if(typeof values[index - 1] === "number") {
        preValue += `$${values[index-1].toFixed(2)}`;
      }
      else {
        preValue += values[index];
      }
    }
    return preValue + currentValue;
  }, "")
}

var amt1 = 11.99, amt2 = amt1 * 1.08, name = "Kyle";
var text = dollabillsyall
`Thanks for you purchase, ${name}! your
product cost was ${amt1}, which with tax
comes out ${amt2}
`
console.log(text)
/*
Thanks for you purchase, 11.99! your
product cost was $11.99, which with tax
comes out $12.95
*/
```

7.3. raw字符串

- 在参数strings返回的数组中，该数组有一个属性raw
  - 原始数据，即其中的换行符等是以转义的形式表示出来，而不是被视为一个换行符
- String.raw(strings, ...values)
  - js提供了一个模板字符串函数
  - 返回一个转义字符没有被变成相应格式的原始字符串

```
String.raw`...`
```

```
console.log(String.raw`hello\nworld`)
console.log(`hello\nworld`)
/*
  hello\nworld
  hello
  world
*/
```

8 箭头函数

```
(argv1, argv2...) => {};
// 相当于
function(argv1, argv2...) {};
```

- 通常会把箭头函数赋值给一个变量，然后再进行调用
- 特点
  - 当函数没有返回值的时候，会默认返回undefined
- 目的
  - 简化匿名函数定义的语法

8.1 箭头函数的特殊规则

8.1.1 只有一个变量

- 括号可以省略

```
let fn = value => {console.log(value)}
```

8.1.2. 只有一行代码

- 大括号可以省略
- 并默认返回对应表达式的执行结果：如布尔表达式则返回true或false

```
let = value => value > 100;
```

8.2 箭头this指向

- 箭头函数不绑定this，this关键字将指向定义位置中的this
  - 能产生局部作用域的函数中定义

```
obj = {name: "bulumrcai"};
function fn() {
  console.log(this);
  return () => console.log(this)
}

const fun = fn.call(obj);
fun();
// obj obj
```

- 在不能产生局部作用域的对象定义(this并不是指向对象的this)

```
obj = {
        name: "bulumrcai"
        sayhello: () => console(this.age)
      };
// undefined this指向的是window
```

8.3 箭头函数的应用注意事项

- 用于使代码更短，对于长的代码，不要考虑使用箭头函数

9 for...of...循环

- of后面的参数是iterable
- 默认具有iterable的内置对象
- Array, String, Generator, collection, typedArray

9.1 for...of和for...in...的区别

- for...of...为变量赋的值是value
- for...in...为变量赋的值是index

```
var a = ['a', 'b', 'c', 'd', 'e'];

for(let val of a) {
  console.log(val);
}
// a b c d e
for(let index in a) {
  console.log(index);
}
// 0 1 2 3 4
```

```
// for...of... 等价代码
  var a = ['a', 'b', 'c', 'd', 'e'];

  for(var val, ret, it = a[Symbol.iterator]();(ret = it.next()) && !ret.done;) {
    val = ret.value;
    console.log( val )
  }
```

10 数字字面量的拓展

- 0b 二进制
- 0o 八进制
- 0x 十六进制
- .toString(radix)
  - 转化为对应进制的字符串

11 Unicode编码（后面再补充）

12 符号（symbol）

12.1 symbol的创建(自定义)

- 不需要使用new符号。不是一个构造器，也不会创建一个对象
- 与string，number等方法类似

12.2 symbol常用的方法

12.2.1 typeof

- 返回值为symbol

12.2.2 instanceof

- 与string，number的字面量不是对应String, Number的实例一样，symbol也不是Symbol的实例
- 返回值为false

12.2.3 valueOf

- 返回symbol字面量

12.2.4 Symbol#toString

- 返回相应的symbol的字符串格式

12.2.5 Symbol.for(desc)

- 在全局符号注册表中搜索，如果有描述文字相同的符号，则返回该symbol
- 若没有，则新建一个该desc的符号，并且返回

12.2.6 Symbol.keyFor(symbol)

- 提取符号的注册文本的描述(desc)

12.3 符号的应用

12.2.3.1 符号的注册

- 符号一般是进行全局注册
  - 可以通过Symbol.for的方法确定唯一性desc

```
const evet = Symbol.for("event");
// 在全局符号注册表中搜索，如果有描述文字相同的符号，则返回该symbol
// 若没有，则新建一个该desc的符号，并且返回
```

12.2.3.2 作为对象属性的符号

- 会用一种特殊的方式存储，使得这个属性不出现在这个对象的一般枚举属性之中
- 需要使用Object.getPropertySymbols(obj)才能取得对应的符号属性

```
var o = {
  foo: 42,
  [ Symbol("bar") ]: "hello world",
  baz: true
}

console.log(Object.getOwnPropertyNames( o ));
console.log(Object.getOwnPropertySymbols( o ));
```

12.2.3.3 内置符号

- iterator
  - Array
  - 在表示时@@iterator来指代内置的符号

13 Map数据结构

- 为什么要使用Map数据结构
  - 对象只能使用字符串作为键，而Map实现了可以使用任意的变量作为键
    - 无法用[]的形式进行访问

13.1 Map的构造

```
var map = new Map(obj);
```

- obj
  - 为空
  - 为一个iterable,正好是entries的返回值，格式和下面的数组一样
  - 为一个数组 [[key, value], [key,value]]
  - 为一个Map

13.2 Map的方法

13.2.1 增删改查

13.2.1.1 Map#set(key, value)

- 给Map对象添加/修改相应的键值

13.2.1.2 Map#delete(key)

- 删除Map对象中对应的键

13.2.1.3 Map#clear(key)

- 清空Map对象中的键

13.2.1.4 Map#has(key)

- 返回boolean，判断是否有给定的键

13.2.1.5 Map#get(key)

- 返回key对应的value值

13.2.2 Map#size()

- 返回Map对象的大小

13.2.3 Map#entries()

- 返回一个iterator
  - 经过展开后:[...Map#entries()]为数组后的格式为 [[key, value], [key,value]]

13.2.4 Map#values()

- 返回一个iterator
  - 经过展开后:[...Map#entries()]为数组后的格式为 [value1, value2...]

13.2.5 Map#keys()

- 返回一个iterator
  - 经过展开后:[...Map#entries()]为数组后的格式为 [key1, key2...]

13.3 WeakMap方法

- 用法类似于Map
- 区别
  - 只有增改查和大小的操作，不能机型删除，暴露key，value等操作
- 优势

14 Set数据结构

- Set的唯一性使得内部元素不能进行类型转化来判断是否相等。

14.1. Set的构造

```
var set = new Set(obj)
```

- obj
  - obj可以为空
  - 可以是一个iterator
  - 可以是一个数组
  - 可以是一个set对象实例

14.2. Set的方法

14.2.1 增删改查

14.2.1.1 Set#add(key, value)

- 给Map对象添加/修改相应的键值

14.2.1.2 Set#delete(key)

- 删除Map对象中对应的键

14.2.1.3 Set#clear(key)

- 清空Map对象中的键

14.2.1.4 Set#has(key)

- 返回boolean，判断是否有给定的键

14.2.2 Set#size()

- 返回Set的大小

14.2.3 Set#entries()

- 返回一个iterator
  - 经过展开后:[...Set#entries()]为数组后的格式为 [[value1, value1], [value2,value2]]

14.2.4 Set#keys()

- 返回一个iterator
  - 经过展开后:[...Set#keys()]为数组后的格式为 [value1, value2...]

14.2.5 Set#values()

- 返回一个iterator
  - 经过展开后:[...Set#values()]为数组后的格式为 [value1, value2...]

15 Array的扩展方法

15.1. Array.of()

- 成为数组推荐的函数形式构造器
  - 解决了单个参数导致只有长度而没有赋值的情况

```
var arr = Array.of(obj)
```

- obj
  - ...val: 任意数量的参数
  - array

15.2. Array.from(arraylike,func?)

- 将iterable或者是类数组转化为数组
  - 返回一个数组
- func可以对数组中的元素进行处理
  - 类似于map的用法
- 优点
  - 避免空槽
    - 空白的地方会自动填充undefined

```
let arraylike = {
  '0': 'a',
  '1': 'b',
  length: 2
}

// ES6以前，调用数组的赋值操作
const arr = Array.prototype.slice.call(arraylike);
const arr = [].slice.call(arraylike)
// ES6
let arr = Array.from(arrayLike);
console.log(arr);
// [a, b]
```

15.3. Array.from和Array.of对于子类的影响

- 可以通过extends来继承Array
- from, of都是用访问它的构造器来构造数组

```
class MyArray extends Array {
  // ...
}

MyArray.of(1, 3) instanceof MyArray; // true
MyArray.of(1, 3) instanceof Array;  // true

Array.of(MyArray.of(1, 3)) instanceof MyArray // false
```

15.4. Array#copyWithin(target, start[, end])

- 不会增加数组的长度，到达数组结尾复制就会停止
- 当target > start时，赋值算法会反向进行
- 当为负数的时候，相当于倒序(最后一个的下标为-1)
- 返回一个被自身数据替换的数组
- 参数
  - target 要复制到的索引

- start 开始复制的索引（包括）
- end 赋值结束的索引（不包括）

```
console.log([1, 2, 3, 4, 5].copyWithin(3, 0, 1));
// [1, 2, 3, 1, 5]
console.log([1, 2, 3, 4, 5].copyWithin(3, -3, -2));
// [1, 2, 3, 3, 5]
console.log([1, 2, 3, 4, 5].copyWithin(2, 1));
// [1, 2, 2, 3, 4]
// 反向赋值，保证不是[1, 2, 2, 2, 2]
```

15.5. Array#fill(value[, start, end])

- 返回一个被填充了相应val的数组
- 若只有参数value，则默认全部填充为value
- 参数
  - val 要填充的值
    - 若只有一个参数，则默认填充全部
  - start 填充的开始索引（包括）
  - end 填充的结束索引（不包括）
- 常用于初始化

```
const arr = Array(26).fill(0);
console.log(arr)
// 全部被填充为0
```

15.6. Array#find(func)

- 返回一个自定义是否匹配的值
- func
  - 返回值是布尔值，当为true时，find会返回相应的value

15.7. Array#findIndex(func)

- 为什么不适用Array#indexOf(val)
  - Array#indexOf(val)是不改变类型的进行比较
  - findIndex由于func的缘故可以自定义的进行比较
- 返回一个自定义的匹配索引
- func
  - 返回值是布尔值，当为true时，find会返回相应的index
- Array#indexOf(value)
  - 若找到对应的值，则返回相应的索引
  - 没有找到则返回-1
  - 采用的是严格匹配===
- 需要严格匹配则使用Array#indexOf

15.8. Array#entries()

- 返回一个iterator
  - 经过展开后:[...Set#values()]为数组后的格式为 [[index , value], [index , value]...]

15.9. Array#keys()

- 返回一个iterator
  - 经过展开后:[...Set#values()]为数组后的格式为 [value1, value2...]

15.10. Array#values()

- 返回一个iterator
  - 经过展开后:[...Set#values()]为数组后的格式为 [0, 1...]

16. Object的扩展方法

16.1. Object.is()

- 执行比===更严格的值比较
- 常用于区分
  - NaN
  - +/- 0

16.2. Object.getPropertySymbols(obj)

- 获得obj中的不可枚举的symbol的数组

16.3. Object.setPrototypeOf(son, father)

- 一般用于行为委托

16.4. Object.assign(target, ...source)

- 返回一个浅复制的对象
- target对象创建source相应的键，然后赋值为source中对应的值

17. Number的扩展方法

17.1 Number.EPSILON

- 任意两个值之间的最小差2^-52，常用于极小值的判断

17.2 Number.MAX_SAFE_INTEGER

- 安全的无歧义的最大整数:2^53 - 1

17.3 Number.MIN_SAFE_INTEGER

- 安全的无歧义的最小整数:-(2^-53 - 1)

17.4 Number.isNaN(val)

- 用于判断是不是为NaN值

17.5 Number.isFinite(val)

- 判断是不是有限的数字
- 没有进行类型转化

17.6 Number.isInteger(val)

- 判断val是否是整数

```
function isFloat(x) {
  return Number.inFinite(x) && !Number.isInteger(x)
}
```

17.7 Number.isSafeInteger(val)

- 判断整数是否在安全范围内

18. String的扩展方法

18.1 针对Unicode的函数（之后补充）

18.1.1 String.fromCodePoint()

18.1.2 String#codePointAt()

18.1.3 String.nomalize()

18.2 String.raw()

- 与模板字符串字面量一起使用

18.3 String#repeat(n)

- 返回重复了n次的字符

```
console.log("foo".repeat(3));
// foofoofoo
```

18.4 新的索引方法

18.4.1 String#startsWith(str[, index])

- 查看第index位是否是以str为开始，index默认值为0

18.4.2 String#endsWith(str[, index])

- 查看第index位是否是以str为结束，index默认值为0

18.4.3 String#includes(str[, index])

- 查看第index位开始是否包含str，index默认值为0

```
  var palindrome = "step on no pets";
  palindrome.startsWith( "step on" ); // true
  palindrome.startsWith( "on", 5 ); // true

  palindrome.endsWith( "no pets" ); // true
  palindrome.endsWith( "no", 10 ); // true

  palindrome.includes( "on" ); // true
  palindrome.includes( "on", 6 ); // false
```

```
{

  method: 'get', // default


  // `transformRequest` allows changes to the request data before it is sent to the server
  // This is only applicable for request methods 'PUT', 'POST', 'PATCH' and 'DELETE'
  // The last function in the array must return a string or an instance of Buffer, ArrayBuffer,
  // FormData or Stream
  // You may modify the headers object.
  transformRequest: [function (data, headers) {
    // Do whatever you want to transform the data

    return data;
  }],

  // `transformResponse` allows changes to the response data to be made before
  // it is passed to then/catch
  transformResponse: [function (data) {
    // Do whatever you want to transform the data

    return data;
  }],

  // `headers` are custom headers to be sent
  headers: {'X-Requested-With': 'XMLHttpRequest'},

  // `params` are the URL parameters to be sent with the request
  // Must be a plain object or a URLSearchParams object
  params: {
    ID: 12345
  },

  // `paramsSerializer` is an optional function in charge of serializing `params`
  // (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
  paramsSerializer: function (params) {
    return Qs.stringify(params, {arrayFormat: 'brackets'})
  },

  // `data` is the data to be sent as the request body
  // Only applicable for request methods 'PUT', 'POST', 'DELETE , and 'PATCH'
  // When no `transformRequest` is set, must be of one of the following types:
  // - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
  // - Browser only: FormData, File, Blob
  // - Node only: Stream, Buffer
  data: {
    firstName: 'Fred'
  },

  // syntax alternative to send data into the body
  // method post
  // only the value is sent, not the key
  data: 'Country=Brasil&City=Belo Horizonte',

  // `timeout` specifies the number of milliseconds before the request times out.
  // If the request takes longer than `timeout`, the request will be aborted.
  timeout: 1000, // default is `0` (no timeout)

  // `withCredentials` indicates whether or not cross-site Access-Control requests
  // should be made using credentials
  withCredentials: false, // default

  // `adapter` allows custom handling of requests which makes testing easier.
  // Return a promise and supply a valid response (see lib/adapters/README.md).
  adapter: function (config) {
    /* ... */
  },

  // `auth` indicates that HTTP Basic auth should be used, and supplies credentials.
  // This will set an `Authorization` header, overwriting any existing
  // `Authorization` custom headers you have set using `headers`.
  // Please note that only HTTP Basic auth is configurable through this parameter.
  // For Bearer tokens and such, use `Authorization` custom headers instead.
  auth: {
    username: 'janedoe',
    password: 's00pers3cret'
  },
```

```
  // `responseType` indicates the type of data that the server will respond with
  // options are: 'arraybuffer', 'document', 'json', 'text', 'stream'
  //   browser only: 'blob'
  responseType: 'json', // default

  // `responseEncoding` indicates encoding to use for decoding responses (Node.js only)
  // Note: Ignored for `responseType` of 'stream' or client-side requests
  responseEncoding: 'utf8', // default

  // `xsrfCookieName` is the name of the cookie to use as a value for xsrf token
  xsrfCookieName: 'XSRF-TOKEN', // default

  // `xsrfHeaderName` is the name of the http header that carries the xsrf token value
  xsrfHeaderName: 'X-XSRF-TOKEN', // default

  // `onUploadProgress` allows handling of progress events for uploads
  // browser only
  onUploadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `onDownloadProgress` allows handling of progress events for downloads
  // browser only
  onDownloadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `maxContentLength` defines the max size of the http response content in bytes allowed in node.js
  maxContentLength: 2000,

  // `maxBodyLength` (Node only option) defines the max size of the http request content in bytes allowed
  maxBodyLength: 2000,

  // `validateStatus` defines whether to resolve or reject the promise for a given
  // HTTP response status code. If `validateStatus` returns `true` (or is set to `null`
  // or `undefined`), the promise will be resolved; otherwise, the promise will be
  // rejected.
  validateStatus: function (status) {
    return status >= 200 && status < 300; // default
  },

  // `maxRedirects` defines the maximum number of redirects to follow in node.js.
  // If set to 0, no redirects will be followed.
  maxRedirects: 5, // default

  // `socketPath` defines a UNIX Socket to be used in node.js.
  // e.g. '/var/run/docker.sock' to send requests to the docker daemon.
  // Only either `socketPath` or `proxy` can be specified.
  // If both are specified, `socketPath` is used.
  socketPath: null, // default

  // `httpAgent` and `httpsAgent` define a custom agent to be used when performing http
  // and https requests, respectively, in node.js. This allows options to be added like
  // `keepAlive` that are not enabled by default.
  httpAgent: new http.Agent({ keepAlive: true }),
  httpsAgent: new https.Agent({ keepAlive: true }),

  // `proxy` defines the hostname and port of the proxy server.
  // You can also define your proxy using the conventional `http_proxy` and
  // `https_proxy` environment variables. If you are using environment variables
  // for your proxy configuration, you can also define a `no_proxy` environment
  // variable as a comma-separated list of domains that should not be proxied.
  // Use `false` to disable proxies, ignoring environment variables.
  // `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and
  // supplies credentials.
  // This will set an `Proxy-Authorization` header, overwriting any existing
  // `Proxy-Authorization` custom headers you have set using `headers`.
  proxy: {
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'mikeymike',
      password: 'rapunz3l'
    }
  },

  // `cancelToken` specifies a cancel token that can be used to cancel the request
  // (see Cancellation section below for details)
  cancelToken: new CancelToken(function (cancel) {
  }),

  // `decompress` indicates whether or not the response body should be decompressed
  // automatically. If set to `true` will also remove the 'content-encoding' header
  // from the responses objects of all decompressed responses
  // - Node only (XHR cannot turn off decompression)
  decompress: true // default

}
```

config

设置的名称	设置的作用	设置的取值	设置的要求
baseURL	基础Url	string: 基础网址	无
url	网址	string: 网址	无
method	请求方法	string: 网址	get post delete