附：内容导航、学习参考、笔记说明

Content Navigator

二级标题	一级知识点	二级知识点
入门	JavaScript 的组成和特点	
	三种编写位置	
	基本语法	注释、大小写、分号等
	字面量、变量、常量	变量声明与赋值
		变量内存结构
		标识符命名规范
数据类型	基本数据类型	数值、大整数、字符串、 布尔值、空值、未定义、符号
	数据类型转换	转换为字符串：`xxx.toString()`、`String()`
		转换为数值：`Number()`、 `str.parseInt()`、`str.parseFloat()`
		转换为布尔值：`Boolean()`
运算符	算术运算符	`+`、`-`、`*`、`/`、`**`、`%`
	赋值运算符	`=`、`??=`、`+=`、`-=`、`*=`、`/=`、`%=`、`**=`
	正负运算符	`+`、`-`
	自增自减运算符	`++`、`--`
	逻辑运算符	`!`、`&&`、`||`
	关系运算符	`>`、`>=`、`<`、`<=`
	相等运算符	`==`、`===`、`!=`
	条件运算符	`condition ? 表达式1 : 表达式2`
流程控制	代码块	`let` Vs. `var`
	条件判断	`if`
	条件分支	`if-else`、`if-else if-else`、`switch`
	循环	`while`、`do-while`、`for`
	break、continue	

二级标题	一级知识点	二级知识点
对象	对象的创建	三种方式
	对象的属性	增删改查
	属性名与属性值	
	对象的检查	`typeof`、`in`
	枚举属性	`for-in`
	可变类型 Vs. 不可变类型	
函数	函数的定义	函数声明、函数表达式、箭头函数
	函数的调用	
	函数的检查	`typeof`
	函数的参数	实参与形参、参数的默认值、参数类型等
	函数的返回值	`return`
	作用域	全局作用域、局部作用域、块作用域、函数作用域、作用域链
	方法	
	window 对象	
	`let` Vs. `var` Vs. `function` Vs. `window`	
	提升	变量、函数、函数中的变量
	立即执行函数	
	函数中的 this	对于函数声明和函数表达式
		对于箭头函数
	严格模式	
面向对象	面向对象编程思想	
	类	类的创建、类的实例化、`instanceof`
	属性	实例属性、静态属性
	方法	实例方法、静态方法
	构造函数	
	面向对象的三大特点	

二级标题	一级知识点	二级知识点
	封装	属性私有化、读写权限控制
	继承	`extends`
	多态	
	对象的内存结构	对象自身、原型对象
	原型的访问	
	原型的特点	
	原型的修改	
	`instanceof` Vs. `in` Vs. `hasOwnProperty` Vs. `hasOwn`	
	旧类	
	`new` 运算符的底层原理	
	对象分类	内建、宿主、自定义
数组	数组的创建	构造函数、字面量
	数组元素的操作	增删改查
	数组的长度属性	`length`
	数组 Vs. 对象	
	数组的遍历	`for`、`for-of`
	浅拷贝与深拷贝	`arr.slice()`、`structuredClone(arr)`
	数组的浅拷贝	`arr.slice()`、`[...arr]`
	对象的浅拷贝	`structuredClone(obj)`、 `Object.assign(obj_target, obj_old)`、 `[...obj]`
	非破坏性方法	`Array.isArray()`、`.at()`、`.concat()`、`.indexOf()`、`.lastIndexOf()`、`.join()`、`.slice()`
	破坏性方法	`.push()`、`.pop()`、`.unshift()`、`.shift()`、`.splice()`、`.reverse()`
	回调函数	
	高阶函数	
	闭包	三要素、原理、生命周期

二级标题	一级知识点	二级知识点
	递归	
	高阶方法	`.sort()`、`.forEach()`、`.filter()`、`.map()`、`.reduce()`
	`arguments`	
	可变参数	
	`this` 的绑定	`call`、`apply`、`bind`

Reference

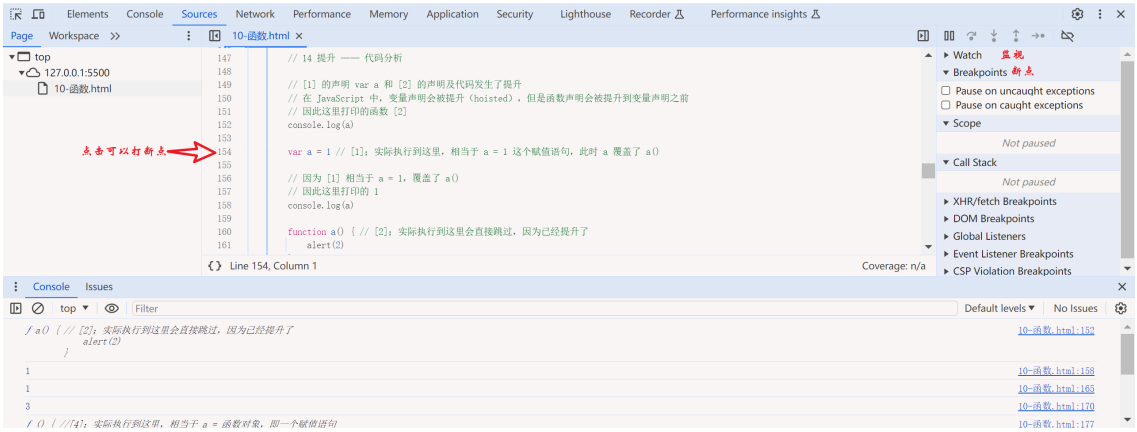
- 参考视频：https://www.bilibili.com/video/BV1mG411h7aD/
- 参考文档：JavaScript Guide - JavaScript | MDN

Note on Usage

- 更加详细全面内容见 代码文件夹，以其中内容为准
- 除了 “4-流程控制” 外，其他部分的笔记都可以在 代码文件夹 中找到，并附上了相关示例

附：调试 debug

1. 我们可以通过开发者模式的源代码（Source）选项卡实现对 JS 的调试



2. 打断点的两种方式
    - 在源代码中，通过 debugger 在代码中打一个断点，让代码在该位置暂停执行

      ```
      debugger
      ```

    - 开发者模式的源代码（Source）选项卡中点击对应的行序号可以打一个断点
3. 可以通过开发者模式的源代码（Source）选项卡右上角的 Watch 实现对变量的监视

1 入门

1.1 简要介绍

1. JavaScript 组成
    - ECMAScript：JavaScript 的语言标准和规范，定义了语法、数据类型、关键字等核心概念。
    - DOM (Document Object Model)：用于操作网页内容的 API，提供了一系列方法和属性来访问和修改 HTML 文档。
    - BOM (Browser Object Model)：用于与浏览器交互的 API，提供了一些浏览器特定的对象和方法，如 `window`、`screen`、`navigator` 等。
2. JavaScript 特点
    - 解释型语言：JavaScript 代码无需编译，由浏览器或 Node.js 等运行环境直接解释执行。
    - 类 C/Java 语法结构：JavaScript 语法借鉴了 C 和 Java 的语法特点，使用大括号、分号等常见语法元素。
    - 动态语言：JavaScript 是一种动态类型语言，变量可以存储任意类型的值，不需要提前声明变量类型。
    - 基于原型的面向对象：JavaScript 不使用传统的类和继承机制，而是通过原型链实现对象的创建和继承。
    - 事件驱动：JavaScript 是一种事件驱动的语言，通过事件处理程序响应用户交互和浏览器事件。
    - 跨平台：JavaScript 可以在不同的浏览器和操作系统上运行，具有较好的跨平台性。
    - 单线程：JavaScript 是一种单线程语言，但可以通过异步编程技术来实现并发操作。
    - 函数式编程：JavaScript 支持函数式编程范式，可以将函数作为参数传递和返回。

1.2 Hello World

1-HelloWorld

1. `alert("Hello World")`
    - 当执行这段代码时，浏览器会弹出一个对话框，显示 "Hello World"。
    - alert() 是 JavaScript 中的一个内置函数，用于在浏览器中弹出一个警告框，显示指定的消息。
2. `console.log("Hello World")`
    - 当执行这段代码时，消息 "Hello World" 会在浏览器的开发者工具控制台中输出。
    - console.log() 是 JavaScript 中的一个内置方法，用于在浏览器的开发者工具控制台中输出指定的消息。
3. `document.write("Hello World")`
    - 当执行这段代码时，浏览器会在网页的 `<body>` 标签中输出 "Hello World"。
    - document.write() 是 JavaScript 中的一个方法，用于在当前网页的 `<body>` 标签中输出指定的内容。

    > 注：`document.write()` 在某些情况下可能会导致一些问题，例如在页面加载完成后再调用它会清空整个页面内容。

1.3 编写位置

2-JS编写位置

1. 位置一：写在 `<script>` 标签中，不需要写 `type` 属性。

```
<script>
    alert("Hello World");
</script>
```

2. 通过 `<script>` 标签引入外部 JavaScript 文件，有助于代码组织和复用。

```
<script src="./js/external.js"></script>
```

3. 位置三：通过在元素的指定属性中编写 JavaScript 代码，可以根据触发条件执行相应的操作。

```
<button onclick="alert('点击成功')">点击我</button>
<button onmousemove="alert('触摸成功')">触摸我</button>
<a href="javascript:alert('海棠');">点击报花名</a>
<a href="javascript:;">点击我什么也不做</a>
```

1.4 基本语法

3-JS语法

1. 多行注释：`/* 这里是注释内容 */`
2. 单行注释：`// 这里是注释内容`
3. JS 严格区分大小写。
4. JS 代码中多个空格和换行会被忽略。
5. JS 中每条语句都应该以分号（;）结尾。虽然 JS 具有自动添加分号的机制，但为了代码的清晰和可读性，建议每条语句都添加分号。

1.5 字面量、变量和常量

4-字面量、变量和常量

1. **字面量**：固定的值，例如 1、2、"Hello"、true、null 等。
2. **变量**：存储着数据的内存地址，可以通过变量更方便地访问数据。变量中存储的数据可以随意修改。
    - **变量的声明**：
        - 推荐使用 `let`：`let 变量名`
        - 旧的声明方法，不推荐使用 `var`：`var 变量名`
    - **变量的赋值**：`变量名 = 值`，值可以是字面量或其他表达式。
    - **声明与赋值的结合**：`let 变量名 = 值` 或 `var 变量名 = 值`
3. **变量的内存结构**：在 JavaScript 中，变量存储的内容取决于它们的数据类型。基本类型存储的是实际的值，而引用类型存储的是指向值的地址（即引用）。
4. **常量**：只能赋值一次的变量。
    - 使用 `const` 关键字声明。

    - 常量的变量名通常大写。
5. **标识符**：JavaScript 中所有可以自主命名的内容，如变量名、函数名、类名等，需要遵循命名规范。
    - 只能含有字母、数字、下划线 `_`、美元符号 `$`，且不能以数字开头。
    - 不能是 JavaScript 中的关键字和保留字，也不建议使用内置的函数名或类名。
    - 通常使用驼峰命名法（首字母小写，其他每个单词开头大写，如 `maxLength`）；类名使用大驼峰命名法（首字母大写，其他每个单词开头大写，如 `MaxLength`）；常量名全部字母大写（如 `MAX_LENGTH`）。

2 数据类型

2.1 基本数据类型

5-基本数据类型

1. **数值 Number**：整数或浮点数。
    - JS 中的数值并不是无限大的，当超过一定范围后会显示近似值，因此进行高精度运算时要注意。
    - `Infinity` 表示无穷大，`NaN`（Not a Number）表示非法数值。
    - 其他进制数字的表示（输出到控制台会变为十进制）：
        - 二进制 `0b` 开头
        - 八进制 `0o` 开头
        - 十六进制 `0x` 开头
2. **大整数 BigInt**：用于表示一些比较大的整数，以 `n` 结尾，可表示的数字范围是无限大（受限于内存）。
3. **字符串 String**：
    - JS 中使用单引号或双引号表示字符串。
    - JS 中可以使用转义字符（`\`）显示一些有特殊含义的字符：
        - `\"` 显示 `"`
        - `\'` 显示 `'`
        - `\\` 显示 `\`
        - `\t` 显示制表符
        - `\n` 显示换行符
    - 模板字符串是一种特殊的字符串，使用反引号（`）来表示，其特点是可以嵌入变量，通过 `${变量名}` 的方式，在字符串中根据变量名输出变量的值。
4. **布尔值 Boolean**：用于进行逻辑判断，只有 `true` 和 `false` 两个可选值。
5. **空值 Null**：表示空对象，只有 `null` 一个可选值；不能使用 `typeof` 运算符检查空置，使用 `typeof` 运算符检查空值时会返回 `object`。
6. **未定义 Undefined**：当一个变量没有赋值时，其数据类型就是 `undefined`，只有 `undefined` 一个可选值。
7. **符号 Symbol**：用于创建一个唯一的表示。

注：七种原始的数据类型是各种数据的基石，其在 JS 中是不可变类型，一旦创建就不能修改！

> 给已经赋值过的一个变量，赋一个新值，并不是修改了该变量原本的值，只是创建了一个新的值而已。

2.2 数据类型转换

6-类型转换

1. **转换为字符串（toString）**
   - 方式一：调用 `toString` 方法。
     - `Null` 和 `Undefined` 数据类型没有 `toString` 方法。
   - 方式二：调用 `String` 函数。
     - 对于有 `toString` 方法的数据类型调用该函数，本质上还是调用 `toString` 方法。
     - 对于 `Null` 和 `Undefined` 数据类型，直接转换为 `"null"` 和 `"undefined"`。
   - 注：字符串转换的本质是根据原有数据类型的数值，创建一个 `String` 数据类型的数值。
2. **转换为数值（toNumber）**
   - 方式一：调用 `Number` 函数
     - 如果传入的是 `String` 数据类型：
       - 如果字符串内容是一个合法数字，则自动转换为对应数字。
       - 如果字符串内容不是一个合法数字，则转换为 `NaN`。
       - 如果字符串是空串或纯空格的字符串，则转换为 `0`。
     - 如果传入的是 `Boolean` 数据类型：`true` 转换为 `1`，`false` 转换为 `0`。
     - 如果传入的是 `Null` 数据类型：转换为 `0`。
     - 如果传入的是 `Undefined` 数据类型：转换为 `NaN`。
   - 方式二：调用 `parseInt` / `parseFloat` 函数
     - 这两个函数适用于字符串转换为数值的情况，都是将字符串从左到右解析，读取连续的所有有效字符（第一个字符必须是数字，否则返回 `NaN`）。
     - `parseInt` 将一个字符串转换为一个整数。
     - `parseFloat` 将字符串转换为一个浮点数。
     - 如果传入参数不是字符串类型，则会先将传入数据转换为字符串类型，然后再进一步处理；基于这一特性，`parseInt` 函数可以用于浮点数取整。
3. **转换为布尔值（toBoolean）**：调用 `Boolean` 函数。
   - 如果传入的是 `Number` 类型：`0` 或 `NaN` 转换为 `false`；其余都是 `true`（包括 `Infinity`）。
   - 如果传入的是 `String` 类型：空串 `""` 转换为 `false`；其余都是 `true`。
   - 如果传入的是 `Null` 或 `Undefined` 类型：都转换为 `false`。
   - 如果传入的是 `Object`：转换为 `true`。

3 运算符

7-运算符

1. **算术运算符**：加法（+）、减法（-）、乘法（*）、除法（/）、幂（**）、模/取余（%）

> 注：JS 是一门弱类型语言，在运算时会进行自动类型转换
>
> - 对于算术运算，除了字符串加法（加法运算至少一个操作数是字符串），其他所有运算，如果操作数是非数值类型，则会先自动转换为数值再运算（to Number）
> - 任意一个值和字符串做加法运算时，会先将该值转换为字符串，然后进行拼串操作；任意类型的值可以通过和空串（""）做加法运算，实现类型转换，原理与 String 函数相同，但是更加简洁（to String）

2. **赋值运算符**：`=`、`??=`、`+=`、`-+`、`*=`、`/=`、`%=`、`**=`
   - 赋值运算符用于将一个值赋给一个变量，或者说是，将符号右边的值赋给左边的变量
   - `??=` 表示空赋值，只有变量的值为 null 或 undefined 时才对变量进行赋值
   - `a += n` ↔ `a = a + n`；`a -=n` ↔ `a = a - n`；..... 其余以此类推
3. **正负运算符**：正号（+）、负号（-）
   - 正负运算符是一元运算符，`+` 表示不改变数值的符号，`-` 表示数值符号位取反

     > 注：我们可以通过正负运算符实现自动类型转换，因为当对非数值类型进行正负运算时，会先将其转换为数值然后再运算；任意类型的值可以通过正号运算符转换为数值类型，原理与 Number 函数相同（to Number）

4. **自增自减运算符**：自增（++）、自减（--）
   - 自增自减运算符是一元运算符，并且与其他运算符不同的是，运算后变量的值会变化（自增会增 1，自减会减 1），而其他运算符只是负责计算出一个结果，由赋值运算符改变变量的值
   - 自增分为前自增（++i）和后自增（i++）；无论哪种，运算后 i 的值都会增加 1；++i 的值是 i 增加后的新值，i++ 的值是 i 增加前的旧值
   - 自减分为前自减（--i）和后自减（i--）；无论哪种，运算后 i 的值都会减少 1；--i 的值是 i 减少后的新值，i-- 的值是 i 减少前的旧值
5. **逻辑运算符**
   - 逻辑非 `!`：逻辑非运算符是一元运算符，可以对一个布尔值进行取反操作

     > 注：我们可以通过逻辑非运算符实现自动类型转换，当我们对一个非布尔值进行逻辑非运算时，会先将其转换为布尔值后在进行取反操作；任意类型的值可以通过两次取反转换为布尔类型（to Boolean）

   - 逻辑与 `&&`
     - 逻辑与运算符是二运运算符，当两个操作数都为 true 时，运算结果才为 true，否则为 false
     - 逻辑与运算是短路的与运算，如果第一个值为 false，则不会看第二个值，直接返回 false
     - 非布尔值进行与运算，会先转换为布尔值然后再运算，但是最终会返回原值
       - 第一个值为 false，则返回第一个值
       - 第一个值为 true，则返回第二个值
   - 逻辑或 `||`
     - 逻辑或运算符是二运运算符，当两个操作数都为 false 时，运算结果才为 false，否则为 true

        - 逻辑或运算是短路的或运算，如果第一个值为 true，则不会看第二个值，直接返回 true
        - 非布尔值进行或运算，会先转换为布尔值然后再运算，但是最终会返回原值
            - 第一个值为 true，则返回第一个值
            - 第一个值为 false，则返回第二个值
6. **关系运算符**：>、>=、<、<=
    - 关系运算符用于检查两个值之间是否满足某种关系，满足则返回 true，否则返回 false

    > 注1：对非数值类型进行关系运算时，一般会先将其转换为数值类型后再进行关系运算
    >
    > 注2：如果俩个操作数都是字符串，则关系运算时，不会将字符串转换为数值，而是按照字符的 Unicode 编码逐位比较；可以利用这个特点对字符串按照字母进行排序

7. **相等运算符**：相等（==）、全等（===）、不相等（!=）、不全等（!==）
    - 相等，用于比较两个值是否相等：当比较两个不同类型的值，会先将其转换为相同的类型（通常转换为数值类型）再进行比较，类型转换后值相同则会返回 true，否则返回 false
        - `NaN` 不与任何值相等，包括自身
        - `null == undefined` 结果是 true
    - 全等，用于比较两个值是否全等：与相等不同的是，全等不会进行自动类型转换，两个值类型不同则直接返回 false，只有在类型相同值时才返回 true
        - `null === undefined` 结果是 false
    - 不等，用于检查两个值是否不相等：对相等结果进行取反，涉及自动类型转换
    - 不全等，用于检查两个值是否不全等：对全等结果进行取反，不涉及自动类型转换
8. **条件运算符（三目运算符）**
    - 语法：`条件表达式 ? 表达式1 : 表达式2`
    - 运算规则：条件表达式运算结果为 true，则执行表达式1；false，则执行表达式2
9. 运算符**优先级**：括号 () 的优先级最大

4 流程控制

8-流程控制

1. 代码块：我们可以使用 {} 来创建代码块，用于对代码进行分组
    - 同一组代码块中的代码，要么都执行，要么都不执行
    - let 和 var 的区别在于
        - let 声明的变量具有块作用域，var 声明的变量不具有块作用域
        - 变量具有块作用域，说明在代码块外无法访问代码块内声明的变量
2. 流程控制语句可以用于改变程序执行的顺序，分为以下三类
    - 条件判断语句
    - 条件分支语句
    - 循环语句：循环的三个必要条件包括初始化表达式、条件表达式、更新表达式

    > 初始化表达式用于初始化变量；条件表达式用于设置循环运行的条件；更新表达式用于修改变量的值

3. if 语句

```
if(条件表达式){
  语句
}
```

如果条件表达式结果为 `true`，则执行其后代码块中的语句；为 `false` 则不执行

> 如果不加 {}，则条件表达式只管辖其下一行语句，类似 C 语言，只作用于一行代码；为了避免错误，可以一直加上 {}，表示其中语句是否执行与条件表达式的值有关
>
> 如果条件表达式是非布尔值的其他数据类型，则会自动转换为布尔值在进行判断

4. if-else 语句

```
if(条件表达式){
  语句
}else{
  语句
}
```

如果条件表达式结果为 `true`，则执行 `if` 后边代码块中的语句；为 `false` 则执行 `else` 后边代码块的语句

> 同样的 if 和 else 后边的 {} 可以取消，此时条件表达式为 true 时，只执行其后的一行代码；为 false 时，只执行 else 后边的一行代码

5. if-else if-else 语句

```
if(条件表达式){
  语句
}else if(条件表达式){
  语句
}else if(条件表达式){
  语句
}...else if(条件表达式){
  语句
}else{
  语句
}
```

如果条件表达式1结果为 `true`，则执行语句1；为 `false` 则继续向下检查，第一个检查到为 `true` 的条件表达式，执行其后的语句；如果所有条件表达式为 `false`，则执行 `else` 后边的代码块中的语句

> 只有一个代码块会被最终执行！
>
> 同样的 if、else if、else 后边的 {} 可以取消，此时如果其对应的条件成立，只执行其后的一行代码

6. switch 语句

```
switch(表达式){
  case 表达式1:
    语句
    break
  case 表达式2:
```

```
    语句
    break
  case 表达式3:
    语句
    break
  ...
  case 表达式n:
    语句
    break
  default:
    语句
    break
}
```

`switch` 语句执行时，会依次将 `switch` 后的表达式和 `case` 后的表达式进行比较；如果比较结果为 `true`，则自对应的 `case` 处开始执行代码块中的所有语句；如果比较结果全是 `false`，则执行 `default` 后的语句

> 因为比较为 true 的 case 后的语句都会执行，因此使用 break 跳出 switch 语句，来避免执行其他的 case

7. while 循环

```
while (条件表达式) {
  循环体
}
```

条件表达式判断为 `true` 时，执行循环体；为 `false` 时跳出循环

8. do-while 循环

```
do {
  循环体
} while (条件表达式)
```

先执行循环体，然后判断条件表达式是否为 `true`，如果为 `true`，则继续执行循环体，否则结束循环

> do-while 和 while 区别
>
> - do-while 先判断再执行（可以确保循环体至少执行一次）
> - while 先执行再判断

9. for 循环

```
for (初始化表达式; 条件表达式; 更新表达式) {
  循环体
}
```

初始化表达式在循环的整个生命周期中只执行一次（第一个执行）；条件表达式为 `true` 时执行循环体，并在循环体执行后执行更新表达式；一旦条件表达式为 `false`，则结束循环

> for 循环中的三个表达式都可以省略；如果都省略则为 for(;;){}，表示死循环

> 初始化表达式中建议使用 let 关键字声明变量，此时变量为 for 循环中的局部变量，只有在 for 循环内部才能访问；如果用 var 声明变量，则为全局变量，在 for 循环外也可以访问

10. break 和 continue
    - `break` 可以终止 `switch` 语句或最近的循环语句
    - `continue` 用于跳过当前循环的剩余代码，继续下一次循环

5 对象

[9-对象]

1. 什么是对象？对象是 JS 中的一种复合数据类型，相当于一个容器，可以存储各种不同类型的数据
2. 对象的创建
    - `let obj_name = new Object()`
    - `let obj_name = Object()`
    - `let obj_name = {}`（字面量方式创建对象，可以直接添加属性）

```
// 举例：使用字面量方式直接创建有属性的对象
let person = {
    name : "孙悟空", // 第一种方式 => 属性名 : 属性值
    age : 18,
    ["gender"] : "男", // 第二种方式 => ["属性名"] : 属性值
    [mySymbol] : "特殊的属性" // 第三种方式 => [变量名] : 属性值
}
```

3. 对象的属性：对象中存储的数据，称之为属性
    - 添加属性
        - `obj_name.field_name = field_value`

        > 此时属性名相当于字符串（但是不用引号括起），因此属性名可以是任何值（甚至关键字、保留字），但是建议遵循标识符命名规范。

        - `obj_name[field_name] = field_value`

        > 此时属性名可以写字符串（用引号括起的），也可以是变量。我们可以使用符号（Symbol）变量作为属性名添加属性，此时必须使用相同的 Symbol 才能取出属性，通常用于不希望被外界访问的属性。

    - 修改属性
        - `obj_name.field_name = new_field_value`
        - `obj_name[field_name] = new_field_value`
    - 读取属性
        - `obj_name.field_name`
        - `obj_name[field_name]`

        > 如果不存在该属性，则会返回 undefined

    - 删除属性
        - `delete obj_name.field_name`

        - `delete obj_name[field_name]`

4. 属性名与属性值
    - 属性名
        - 句点（.）调用时：相当于字符串（不用引号括起），直接写就行
        - 中括号（[]）调用时：字符串（需要引号括起）；变量名
    - 属性值：任意数据类型，也可以是一个对象
5. 对象的检查
    - 可以使用 `typeof` 运算符检查对象的类型，返回 `object`
    - 可以用 `in` 运算符检查对象中是否含有某个属性，有则返回 `true`，没有则返回 `false`，语法为：`变量名 in 对象名`
6. 枚举属性

```
for (let field_name in obj_name) {
    语句
}
```

我们可以通过 for-in 语句，获取对象中的所有属性（不包括符号属性，符号属性不可枚举）

> for-in 语句的循环体循环的次数取决于可枚举的属性的数量，每次循环时，都会有一个属性名赋值 field_name 变量，此时只可以通过 obj_name[field_name] 的方式访问属性值，不可用句点访问，因为此时 field_name 是一个变量

```
const Jack = {
    name: "Jack",
    age: 19,
    gender: "Male",
    height: 188,
    weight: 78
}

for (let field in Jack) {
    console.log(`${field} is ${Jack[field]}`);
}
```

7. 对象 —— 可变类型
    - 可变类型与不可变类型
        - 原始值属于不可变类型，一旦创建就无法修改；此外，内存中不会创建重复的原始值
        - 对象属于可变类型，创建完成后可以任意地添加删除修改对象中的属性
    - 对象比较（相等或全等）的本质是对象的内存地址的比较
    - 如果有两个变量同时指向一个对象，此时如果通过一个变量修改对象，另一个变量也会产生影响
    - 通常使用 `const` 关键字声明存储对象的变量，这是为了避免因为修改变量指向的对象而提高代码的复杂度，注意此时对象仍然是可修改的

6 函数

10-函数

1. 什么是函数（Function）？函数是一个存储代码的对象，具有其他对象所有的功能，并且可以在需要时调用这些代码
2. 函数的定义
    - 函数声明

      ```
      function fun_name([参数]) {
          语句
      }
      ```

    - 函数表达式

      ```
      const 变量 = function([参数]) {
          语句
      }
      ```

    - 箭头函数

      ```
      const 变量 = ([参数]) => {
          语句
      }
      ```

    - 说明
        - 函数表达式和箭头函数都是匿名函数，返回一个函数对象给对应的变量，然后通过该变量调用函数
        - 使用 `const` 关键字声明存储函数对象的变量，是为了避免变量的重复赋值
        - 箭头函数的函数体如果只有一行语句，则可以省略花括号
        - 箭头函数有且仅有一个参数，则可以省略括号
3. 函数的调用 `函数对象()`
    - 对于函数声明， fun_name 就是一个函数对象
    - 对于匿名函数，变量就是一个函数对象
4. 函数的检查
    - 使用 typeof 运算符检查函数对象时会返回 function
5. 实参与形参
    - 形式参数（形参）：定义函数时，在函数中指定的数量不等的参数，以逗号分隔，相当于在函数内部声明了变量但是没赋值
    - 实际参数（实参）：调用函数时，在函数的 () 中传递数量不等的参数，以逗号分隔。 -
    - 参数传递：函数调用时传递的实参会赋值给函数中对应的形参
        - 实参数量 = 形参数量：对应的实参赋值给对应的形参
        - 实参数量 > 形参数量：多余实参不予使用
        - 实参数量 < 行参数量：多余形参值为 `undefined`

> JavaScript 不会检查函数参数的数据类型，从而减少报错，但是我们在实际使用中应该手动进行类型判断，从而确保代码实现符合预期； 如希望参数是数值类型时，可以先将传入的实参转换为数值类型，再使用 `isNaN` 函数进行合法性判断。

6. 参数的默认值 `形参名=默认值`
    - 定义参数时，可以为其指定默认值，在没有传入对应的实参时生效
7. 函数的参数类型
    - 对象作为函数参数：形参的修改会影响到实参
    - 对象作为函数形参的默认值：每次调用函数，都会通过默认值创建一个新的对象
    - 函数也可以作为函数参数
8. 函数的返回值 `return 返回值`
    - 函数体中，可以通过 return 关键在来指定函数的返回值
    - 任何值（包括对象和函数等）都可以作为函数的返回值
    - return 后如果不跟任何值，则相当于返回 undefined
    - 如果不使用 return 关键字，则函数的返回值仍然是 undefined
    - return 有使函数立即结束的作用
    - 箭头函数的返回值：如果函数只有返回值这一条语句，则可以省略花括号和关键字 return；但是如果返回的是一个对象字面量，则需要用括号括起，避免对象字面量的花括号和函数体的花括号产生混淆

```
(a, b) => a + b
() => ({name: "Tom"})
```

```
function sum1(a, b) {
    return a + b
}

const sum2 = function (a, b) {
    return a + b
}

const sum3 = (a, b) => a + b

let a = 12
let b = 13

console.log(`${a}+${b}=${sum2(a, b)}`)
```

9. 全局作用域和局部作用域
    - 作用域（scope）：指的是一个变量的可见区域，分为全局作用域和局部作用域
    - 全局作用域
        - 在网页运行时创建，网页关闭时销毁
        - 所有“直接”编写在 `script` 标签中的变量都位于全局作用域中
        - 全局作用域中的变量是全局变量，可以在任意位置访问
    - 局部作用域
        - 块作用域
            - 在代码执行时创建，执行完毕后销毁
            - 块作用域中的变量是局部变量，只能在块内部访问，外部无法访问

- 函数作用域
  - 在函数调用时创建，调用完毕后销毁
  - 函数的每次调用都会产生一个全新的函数作用域
  - 函数作用域中定义的变量是局部变量，只能在函数内部访问，外部无法访问

10. 作用域链

    当我们要使用一个变量时，JavaScript 解释器会优先在当前作用域中寻找变量，如果找到则直接使用，如果未找到则会向上一层作用域中继续寻找，直到找到全局作用域。 如果一直找到全局作用域都未找到变量，则会报错 "xxx is not defined"

```
let variable = 15 // 1

function f1() {
    let variable = 16 // 2

    function f2() {
        let variable = 17 // 3
        console.log(variable) // 寻找顺序 3 → 2 → 1，找到则使用，找不到则继续向外找
    }
}
```

11. 对象的函数 —— 方法（method）

    当一个对象的属性指向一个函数，此时就称这个函数是该对象的方法，调用该函数称之为调用对象的方法

```
let person = new Object()

person.name = "yiTu"
person.gender = "Male"
person.sayHello = () => console.log(`Hello, I'm ${person.name}`)

person.sayHello()
```

12. window 对象

    - 浏览器为我们提供了一个可以直接访问的 `window` 对象，表示浏览器窗口，通过该对象可以对浏览器窗口进行各种操作；同时 `window` 对象还负责存储 JavaScript 中的内置对象和浏览器中的宿主对象
    - `window` 对象的属性可以通过 `window` 对象访问，也可以直接访问，如 `window.console.log("Hello")`。
    - 函数可以认为是 `window` 对象的方法，如 `window.alert("Hello")`。

13. let、var、function、window 声明区别

    - `var` 声明变量和 `let` 声明变量的区别是：`var` 声明的变量不具有块作用域。
    - 全局作用域中使用 `var` 声明的变量，都会作为 `window` 对象的属性保存（`var a = 10` 等价于 `window.a = 10`）。
    - 全局作用域中使用 `function` 声明的函数，都会作为 `window` 对象的方法保存。
    - 全局作用域中使用 `let` 声明的变量，不会存储在 `window` 对象中，而是存储在一个特定的地方。

- 如果 `window.a` 和 `let a` 两种方式声明同名变量，则在后续代码中，访问的是 `let` 声明的变量。
- `var` 虽然没有块作用域，但是有函数作用域。
- 在局部作用域中，如果没有使用 `var` 或 `let` 声明变量，直接使用 `a = 10` 这样的方式，则变量会自动成为 `window` 对象的属性，也就是全局变量（不建议使用）。

14. 提升
    - 变量的提升
        - 使用 `var` 声明的变量，会在所有代码开始执行前被声明（假设 19 行：`var a = 1`，那么 JavaScript 解释器会将 `var a` 提升到所有代码的最前边，19 行的代码此时就相当于一个简单赋值 `a = 1`），因此我们可以在变量实际声明位置前就访问该变量。
        - 使用 `let` 声明的变量，也存在提升，但是 JavaScript 解释器禁止在代码中变量声明的实际位置前访问该变量。
    - 函数的提升：使用函数声明（方式一）创建的函数，会在所有代码执行前被创建，因此我们可以在函数声明前调用函数。
    - 函数中变量的提升：函数中声明的变量，也会在函数所有代码开始执行前提升到函数最开始的位置（类似的 `let` 声明的变量不能提前访问；`var` 声明的变量可以提前访问）。
    - 提升的意义：提升的意义在于为变量和函数预先分配内存空间，避免因为频繁内存分配降低代码效率。
    - 注：直接通过 `a = 10`，相当于 `window.a = 10` 的方式，此时变量不存在提升。

```
// [1] 的声明 var a 和 [2] 的声明及代码发生了提升
// 在 JavaScript 中，变量声明会被提升（hoisted），但是函数声明会被提升到变量声明之前
// 因此这里打印的函数 [2]
console.log(a)

var a = 1 // [1]：实际执行到这里，相当于 a = 1 这个赋值语句，此时 a 覆盖了 a()

// 因为 [1] 相当于 a = 1，覆盖了 a()
// 因此这里打印的 1
console.log(a)

function a() { // [2]：实际执行到这里会直接跳过，因为已经提升了
    alert(2)
}

// 因为 [2] 跳过了，因此这里 a 不变，打印结果仍为 1
console.log(a)

var a = 3 // [3]：实际执行到这里，相当于 a = 3 这个赋值语句

// 因为 [3] 相当于重新赋值，因此这里 a 打印为 3
console.log(a)

var a = function() { //[4]：实际执行到这里，相当于 a = 函数对象，即一个赋值语句
    alert(4)
}

// 因为 [4] 相当于重新赋值，因此这里的 a 打印为函数 [4]
console.log(a)
```

```
var a // [5]；实际执行到这里跳过，因为 var a 早都因为 [1] 提升了变量 a 的声明

// 因为 [5] 跳过了，因此这里 a 不变，打印结果仍为函数 [4]
console.log(a);
```

15. 代码编写规范：实际开发中减少在全局作用域中编写代码，而尽量在局部作用域中编写；可以使用 {} 来创建块作用域
16. 立即执行函数（IIFE）

```
(function([参数]){
    语句
}());
```

类似函数表达式的调用，但是整体外边需要括起，并且必须以 ; 结束

- IIFE 是匿名函数，并且只会调用一次
- IIFE 不会被提升
- IIFE 可以用于创建一个一次性的函数作用域，从而避免变量的冲突问题

```
(function () {
    console.log("hello world")
}());
```

17. 函数中的 this
    - 函数在执行时，JavaScript 解释器每次都会传进一个隐含的参数 `this`，`this` 指向一个对象。
    - 对于函数声明和函数表达式创建的函数，`this` 指向的对象会根据函数调用方式的不同而不同：
        - 以函数形式调用，如 `fun_name()`，此时 `this` 指向的是 `window` 对象。
        - 以方法形式调用，如 `obj_name.method_name()`，此时 `this` 指向的是调用该方法的对象 `obj_name`。
    - 对于箭头函数，`this` 由外层作用域决定，与其调用方式无关：
        - 箭头函数的外层作用域是全局作用域，则 `this` 指向 `window` 对象。
        - 箭头函数的外层作用域是对象 `obj_name`，则 `this` 指向 `obj_name` 对象。

```
const stu1 = {
    name: "jack",
    gender: "male",
    height: 178,
    weight: 78,
    performance: "B",
    details: function () {
        console.log(`My name is ${this.name}, and I'm ${this.height} cm tall,
${this.weight} kg weight, Performance grade is ${this.performance}`);
    }
}

const stu2 = {
    name: "Jerry",
```

```
    gender: "female",
    height: 157,
    weight: 48,
    performance: "A",
    details() {
        console.log(`My name is ${this.name}, and I'm ${this.height} cm tall, ${this.weight} kg weight, Performance grade is ${this.performance}`);
    }
}

stu1.details()

stu2.details()
```

18. 箭头函数语法总结
    - 函数体只有返回值语句时
        - 无参：`() => 返回值`
        - 单参：`a => 返回值`
        - 多参：`(a, b, c, ...) => 返回值`
        - 返回值为对象：`() => ({...})`
    - 函数体有多行语句时

```
() => {
    语句
    return 返回值
}
```

19. 严格模式
    - JS 运行代码的模式有两种，正常模式和严格模式
    - 正常模式：默认情况下代码都运行在正常模式中

> 语法检查不严格，能不报错就不报错；代码运行性能较差

- 严格模式：使用 `"use strict"` 可以在当前作用域下开启严格模式

```
> 语法检查变严格，禁止了一些语法，更容易报错；代码运行性能提升
```

- 在实际开发中推荐使用严格模式，一方面避免产生一些隐藏问题，另一方面可以提升代码的运行性能

7 面向对象

11-面向对象

7.1 基本语法

1. 面向对象编程（OOP，Object Oriented Programming）
    - 面向对象编程是一种软件开发范式，它通过将问题领域中的实体抽象为对象，以及对象之间的交互来解决问题

- 每个对象都有其自己的数据（即属性）和行为（即方法），例如，一个汽车可以被抽象为一个对象，其属性可以包括颜色、速度和品牌，而方法可以包括加速、减速等行为
- 在面向对象编程中，一切皆对象！

2. 类（Class）
   - 什么是类?
     - 因为使用 Object 创建对象，既无法区分不同类型的对象，也不方便批量地创建对象，因此 JS 通过类来解决这个问题
     - 类是对象的模板，可以将对象中的属性和方法直接定义在类中，当我们定义好一个类后，就可以直接通过类来创建对象
     - 如果某个对象是由某个类所创建，我们称这个对象是这个类的实例
   - 类的创建
     - 方法一 `class 类名 {}`
     - 方法二 `const 类名 = class {}`
     - 注：类名使用大驼峰命名法
   - 通过类创建对象：`new 类名()`，这种方式称之为调用构造函数创建对象
   - `instanceof` 关键字：我们可以使用该关键字检查某个对象是否是由某个类创建的，是则返回 true，否则为 false
   - 注：类的代码块默认就是严格模式
3. 属性（Field）：类中我们可以定义两种属性，分别是实例属性和静态属性（类属性）
   - 实例属性 `field_name = value`

     > 实例属性只能通过实例访问；不需要使用任何关键字去声明属性

   - 静态属性 `static field_name = value`

     > 静态属性，又称类属性，只能通过类去访问；需要使用 static 关键字去声明属性

4. 方法（Method）：与属性类似，类中我们也可以定义两种方法，分别是实例方法和静态方法（类方法）
   - 实例方法
     - 方式一 `method_name = function() {}`
     - 方式二 `method_name() {}`
     - 说明

       > 实例方法只能通过实例访问
       >
       > 不需要使用任何关键字去声明指向方法的变量
       >
       > 实例方法中的 this 指向的是当前实例
       >
       > 方式二定义的方法，不可以通过打印实例显示，但是实例可以常调用

```
        - 静态方法

            - 方式一 `static method_name = function() {}`

            - 方式二 `static method_name() {}`

            - 说明

                > 静态方法，又称类方法，只能通过类去访问
                >
                > 需要使用 static 关键字去声明属性指向方法的变量
                >
                > 静态方法中的 this 指向的是当前类
```

```
class Person {
    name = "Tom Sun"
    age = 21

    static belongTo = "Journey to West"
    static powerLevel = "highest"

    sayDetails() {
        console.log("I'm " + this.name + ", " + this.age + " years old")
    }

    static sayOrigin() {
        console.log("We'are from " + this.belongTo + ", and at the " +
this.powerLevel + " power level")
    }
}

const p = new Person()

console.log(p)
console.log(Person)
console.log(p.belongTo) // undefined
console.log(Person.belongTo) // Journey to West

p.sayDetails() // I'm Tom Sun, 21 years old
Person.sayOrigin() // We'are from Journey to West, and at the highest power level
```

7.2 构造函数

构造函数（constructor）

1. 是什么?
    - `constructor` 是类中的一个特殊的方法，称之为构造函数（或构造方法）。
    - 我们使用类创建对象时，构造函数会自动执行。
    - 通过 `new class_name(field1, field2, ...)` 创建对象时，其实是调用了空的构造函数，并将参数传递给构造函数中。

- 构造函数中的 `this` 指向的是当前创建的实例对象，因此我们可以在构造函数中对实例属性进行赋值。
  通过构造函数，我们可以实现创建的每个对象都有各自的属性值。

2. 语法：
   假设 `new class_name(field1, field2, ...)`，则此时构造函数语法如下：

```
class class_name {
    constructor(field1, field2, ...) {
        this.field1 = field1; // this.field1 是实例属性，field1 是外界传进来的赋值给构造函数的形参
        this.field2 = field2;
    }
}
```

```
class Person {
    constructor(name, age, gender) {
        this.name = name
        this.age = age
        this.gender = gender
    }

    sayDetails() {
        console.log("I'm " + this.name + ", " + this.age + " years old, " + this.gender)
    }
}

const p1 = new Person("孙悟空", "519", "male")
const p2 = new Person("猪八戒", "234", "male")
const p3 = new Person("孙悟空", "135", "male")

p1.sayDetails() // I'm 孙悟空, 519 years old, male
p2.sayDetails() // I'm 猪八戒, 234 years old, male
p3.sayDetails() // I'm 孙悟空, 135 years old, male
```

7.3 面向对象的三大特性

1. 面向对象的三大特点：封装、继承、多态
   - 封装（Encapsulation） —— 安全
     - 是什么?
       - 封装是指将数据（对象的属性）和操作数据的方法（对象的行为）捆绑在一起的机制。
       - 通过封装，对象的内部细节被隐藏起来，只有被允许的操作才能够访问和修改对象的状态。
       - 可以保证数据的安全性，同时降低了对象的耦合度，使得代码更易于理解和维护。
       - 还可以提高代码的重用性，因为对象的内部实现细节可以被隐藏，而只暴露必要的接口给外部使用。
     - 封装的实现

        - 属性私有化：通过在属性名前加上 #，将需要保护的数据设置为私有，确保只能在类内部使用。
        - 属性读写权限控制：提供 getter 和 setter 方法来操作属性，getter 提供读取权限，setter 提供修改权限。可以在方法中对属性的值进行验证，如数据类型判断、大于 0 判断等。
    - 继承（Inheritance） —— 扩展
        - 是什么?
            - 继承是指一个类（子类）可以从另一个类（父类）中继承属性和方法的机制。
            - 通过继承，子类可以重用父类的代码，并且可以在其基础上进行扩展和定制。
            - 可以减少代码的重复性，提高了代码的可维护性和可扩展性。
        - 好处
            - 符合 OCP（开闭原则）：软件实体应该对扩展开放，对修改关闭。
            - 可以建立类之间的层次关系，使得代码的结构更加清晰和易于理解。
        - 通过继承定义一个类：`sub_class_name extends super_class_name {}`
    - 多态（Polymorphism） —— 灵活
        - 是什么?
            - 多态是指同一种操作可以在不同的对象上具有不同的行为的特性。
            - 通过多态，可以实现基于对象类型的动态调度，即在运行时根据对象的实际类型来调用相应的方法。
            - 多态通常通过方法重写（Override）和方法重载（Overload）来实现，在编译时或运行时确定具体调用的方法。
2. 封装（Encapsulation）
    - 引：对象可以看作是一个存储不同属性的容器，而对象不仅要负责存储属性，还要确保数据的安全，而封装就是确保数据安全。

- 封装的实现
  - 属性私有化：通过在属性名前加上 #，将需要保护的数据设置为私有，确保只能在类内部使用。

> - 属性读写权限控制：提供 `getter` 和 `setter` 方法来操作属性，`getter` 提供读取权限，`setter` 提供修改权限。

3. 继承（Inheritance）
    - 引：一个类继承另一个类时，可以理解为将另一个类中的所有代码复制到了当前类中；被继承的类称之为父类，继承的类称之为子类。
    - 好处
        - 符合 OCP（开闭原则）：软件实体应该对扩展开放，对修改关闭。
        - 可以建立类之间的层次关系，使得代码的结构更加清晰和易于理解。
    - 在子类中的操作
        - 可以通过创建同名方法来重写（overwrite）父类的方法。
        - 重写构造函数时，第一行代码必须是 `super()`，表示调用父类的构造函数，并且可以向 `super()` 传递一些必要的参数。

        - 通过 `super.method_name()` 的方式，调用父类中的方法。
    4. 多态（Polymorphism）
        - 引：因为 JS 不会检查函数参数的类型，因此如果对象作为参数，则无需是指定的类型，只要满足某些条件即可。
        - 多态为程序的编写提供了灵活性。
    5. 对象的内存结构
        - 是什么？对象在内存中存储其属性、方法的区域有两个地方，分别是对象自身和原型对象（prototype）。
- 查找优先级：对象自身→原型对象，当访问对象中的属性或方法时，会优先访问对象自身的存储区域，如果找不到，再去原型对象的存储区域中寻找。

7.4 原型

```
# 原型

- **引**
  > - 原型是 JavaScript 中一种重要的机制，用于实现对象之间的继承和属性共享。
  > - 每个对象都有一个原型（prototype），它是一个指向另一个对象的引用。
  > - 当试图访问对象的属性时，如果该属性不存在于对象本身，则 JavaScript 引擎会沿着原型链向上查找，直到找到该属性或者到达原型链的末端（Object.prototype）。

- **访问一个对象的原型对象**
  > - 方式一：`obj_name.__proto__`
  > - 方式二：`Object.getPrototypeOf(obj_name)`

- **原型链**
  > - 一个对象有原型对象，而原型对象也有原型对象，直到 Object 的原型对象没有原型。
  > - 原型链就是 JS 查找属性或方法的一种路径，如果当前对象中存在所需属性或方法，则使用，否则去其原型对象中找。

# 原型的特点

- **同类对象的原型对象是同一个**
- **可以将原型对象理解为一个公共区域**
- **JS 的继承本质上是原型继承；子类的原型是一个父类的实例**
- **原型解决了对象中的某些值是对象所独有的，而有些值对所有对象都是相同的的问题**

# 原型的修改

- **语法：`class_name.prototype`**
- **最好通过类修改原型，一方面一旦修改则修改了所有实例的原型，另一方面无需创建实例即可修改**
- **注意**
  > - 原型尽量不要手动修改。
  > - 原型修改不要通过实例对象（`obj_name.__proto__`）去修改，最好通过类去修改（`class_name.prototype`）。

# instanceof/in/hasOwnProperty/hasOwn

- **instanceof [关键字]**
  > - 用于检查一个对象是否是一个类的实例，本质是检查该对象的原型链上是否有该类的实例。

- **in [关键字]**
```

```
  > - 用于检查一个属性是否存在于一个对象之中，这里的存在既包含对象自身，也包含其原型对象。

- **hasOwnProperty [内置方法]**
  > - 用于检查一个对象自身是否含有某个属性，不推荐使用。

- **hasOwn [Object 的静态方法]**
  > - 用于检查一个对象自身是否含有某个属性，推荐使用。
```

7.5 旧类

```
# 旧类

- **构造函数、公共属性、静态属性、创建实例等语法**
  ```javascript
  function ClassName(param1, param2) {
      this.field1 = param1;
      this.field2 = param2;
  }

  ClassName.staticField = "Static Field";

  ClassName.prototype.methodName = function() {
      // Method implementation
  };

  var instance = new ClassName(value1, value2);
```

7.6 new 运算符

When a function is called with the `new` keyword in JavaScript, it behaves as a constructor. Here's a breakdown of what happens when `new` is used:

1. **Creates a blank, plain JavaScript object:**

   ```
   var newInstance = {};
   ```

2. **Points newInstance's `[[Prototype]]` to the constructor function's `prototype` property:**

   ```
   newInstance.__proto__ = current_class_name.prototype;
   ```

   This step establishes the prototype chain, allowing the new instance to inherit properties and methods from the constructor function's prototype.

3. **Executes the constructor function with the given arguments, binding newInstance as the `this` context:**
   This means that inside the constructor function, references to `this` will point to the newly created instance (`newInstance`).

4. **Returns the newly created object instance (`newInstance`):**
   If the constructor function returns a non-primitive value, that value will be the result of the `new` expression. Otherwise, if the constructor function doesn't explicitly return anything or returns a primitive value, the `newInstance` object will be returned instead.
   ```

In summary, the `new` keyword facilitates the creation of new object instances from constructor functions in JavaScript, setting up the prototype chain and executing the constructor function with the appropriate context.

7.7 对象的分类

- 内建对象：由 ES 标准定义的对象，如 Object、Function、String、Number、...
- 宿主对象：由浏览器提供的对象，包含 BOM 和 DOM
- 自定义对象：开发者自己创建的对象

8 数组

12-数组

8.1 基本语法

```
## 数组 (Array)

- **定义**:
  - 数组是一种复合数据类型，用于存储多个不同类型的数据（为了性能考虑，通常确保数组中存储的数据类型相同）。
  - 数组中的数据是有序的，每个数据都有一个唯一的索引，可以通过索引来操作数据。
  - 数组中存储的数据称为元素，索引从0开始计数。
  - 使用 `typeof` 关键字检查数组对象时，返回结果为 "object"。

- **创建数组**:
  - 方式一：使用构造函数 `const arr = new Array()`
  - 方式二：使用数组字面量
    - `const arr = []`
    - `const arr = [12, 34, 56, 56, 12]`

- **元素操作** (假设 `arr` 是数组对象，`index` 是元素索引，`element` 是元素取值):
  - 读取元素：`arr[index]`（如果 `index` 非法，不会报错，而是返回 `undefined`）。
  - 添加元素：`arr[index] = element`（若添加前数组中索引为 `index` 的元素不存在）。
  - 修改元素：`arr[index] = element`（若添加前数组中索引为 `index` 的元素已存在）。

- **数组长度属性 (`length`)**:
  - 数组长度 `length` 等于数组的最大索引值加 1。
  - 可以通过 `arr[arr.length] = element` 向数组末尾追加元素。
  - `length` 属性是可修改的：若修改后的 `length` 比数组长度大，则会增加空元素；若小，则会截掉尾部元素以满足 `length`。

- **数组与对象的区别**:
  - 数组中的数据有序，对象中的数据无序。
  - 数组中的数据称为元素，对象中的数据称为属性。
```

8.2 数组的遍历

```
## 数组的遍历

- **方式一：for 循环**
- **方式二：for-of 循环**
  - 可以遍历所有可迭代对象，如数组、字符串等。
  - **语法**：`for (变量 of 可迭代对象) { 语句 }`
  - **执行流程**：for-of 循环体执行的次数为可迭代对象中元素的个数，每次循环从可迭代对象中按顺序取出一个元素赋给变量。
```

8.3 浅拷贝与深拷贝

```
## 浅拷贝与深拷贝

- **浅拷贝（Shallow Copy）**：
  - 通常对对象的拷贝都是浅拷贝，顾名思义，浅拷贝只对对象的浅层进行复制（或者说只复制一层）。
  - 如果对象存储的数据是原始值，此时拷贝的深浅与否并不重要，作用相同；只有对象中存储的数据是对象时，才会有所差异。
  - 浅拷贝只对对象本身进行复制，不会复制对象中的属性/元素。
  - 对于数组对象，可以使用实例方法 `arr.slice()` 返回一个 `arr` 的浅拷贝。
  - **举例**：
    - **原始对象**：
      ![原始对象](https://cdn.jsdelivr.net/gh/Nasir1423/blog-img@main/20240330235024.png)
    - **浅拷贝后的对象**：
      ![浅拷贝后的对象](https://cdn.jsdelivr.net/gh/Nasir1423/blog-img@main/20240330235129.png)

- **深拷贝（Deep Copy）**：
  - 不仅复制对象本身，还要复制对象中的属性/元素，但通常不使用深拷贝因为性能问题。
  - 可以使用函数 `structuredClone(arr)` 返回一个 `arr` 的深拷贝。
  - **举例**：
    - **原始对象**：
      ![原始对象](https://cdn.jsdelivr.net/gh/Nasir1423/blog-img@main/20240330235024.png)
    - **深拷贝后的对象**：
      ![深拷贝后的对象](https://cdn.jsdelivr.net/gh/Nasir1423/blog-img@main/20240331000307.png)
```

8.4 数组的浅拷贝

```
## 数组的浅拷贝

- **方式一：使用数组的实例方法 `slice()`**
  - 使用 `slice()` 方法复制一个对象。注意，复制会生成一个新的对象。
  - **语法**：`const arr_new = arr_old.slice()`

- **方式二：使用展开运算符（...）**
  - 使用展开运算符 `[...]` 复制一个对象。
  - **语法**：`const arr_new = [...arr_old]`

- **关于展开运算符（...）：**
```

  - 可以将一个数组中的所有元素展开到另一个数组中，或者作为函数的参数传递。
  - `...arr` 表示将数组中的所有元素展开。
  - `"孙悟空", ...arr, "唐僧"` 表示将数组中的所有元素展开，并在前后各加上一个元素。

8.5 对象的浅拷贝

```
## 对象的深拷贝

- **方式一：使用函数 `structuredClone(obj)`**
  - 使用 `structuredClone(obj)` 函数创建一个对象的深拷贝。
  - **语法**：`const obj_new = structuredClone(obj_old)`

## 对象的浅拷贝

- **方式一：使用 Object 的静态方法 `Object.assign(obj_target, obj_old)`**
  - 使用 `Object.assign(obj_target, obj_old)` 方法将 `obj_old` 对象中的属性复制到 `obj_target` 对象中。
  - **语法**：`const obj_target = Object.assign(obj_target, obj_old)`
  - **注**：`Object.assign` 方法用于给 `obj_target` 中添加 `obj_old` 的属性，`obj_old` 会覆盖掉 `obj_target` 中的同名属性，而不同名属性不会影响。

- **方式二：使用展开运算符（...）**
  - 使用展开运算符 `[...]` 复制一个对象。
  - **语法**：`const arr_new = [...arr_old]`
```

8.6 数组的常用方法

非破坏性

```
## 数组的常用非破坏性方法（即不会影响原数组，而是返回一个新的数组）

### 静态方法（Array 是数组类）

- **`Array.isArray(obj)`**：
  - 检查一个对象是否是数组对象，是则返回 true，否则返回 false

### 实例方法（假设 arr 是一个数组对象）

- **`arr.at(index)`**：
  - 根据索引返回数组中的指定元素；索引可指定为负数，如 -1 表示倒数第一个元素

- **`arr.concat(arr1, arr2, ..., arrn)`**：
  - 连接两个或多个数组并返回连接结果

- **`arr.indexOf(element, [start])`**：
  - 返回元素在数组中第一次出现的索引

- **`arr.lastIndexOf(element, [start])`**：
  - 返回元素在数组中最后一次出现的索引
  - `arr.indexOf` 和 `arr.lastIndexOf` 都是用于返回数组中某个元素“第”一次出现的索引，找不到则返回 -1
  - 两个方法都接收最多两个参数，第一个参数表示要查询的元素（必选），第二个参数表示查询的起始位置（可选）
  - `arr.indexOf` 从前向后查询，`arr.lastIndexOf` 从后向前查询
```

- **`arr.join([sep])`**：
  - 将数组中的元素连接为字符串并返回；默认连接符为逗号，可以通过可选的 sep 参数指定一个字符串作为连接符

- **`arr.slice([start], [stop])`**：
  - 返回一个数组的切片，从 start 位置开始，到 stop 位置结束（不包含 stop 位置的元素）
  - 截取的起始位置 start 可以省略，省略则表示从第一个元素开始截取；结束位置 stop 可以省略，省略则表示一直截取到最后一个元素
  - 使用区间表示，则 [start, stop) 为实际截取的数组
  - start、stop 取值可以为负数，如 -1 表示倒数第一个元素
  - 如果将 start、stop 都省略，此时 `arr.slice()` 表示对数组进行浅拷贝

破坏性

```
## 数组的常用破坏性方法（实例方法，假设 arr 是一个数组对象）

- **`arr.push(element1, element2, ..., elementn)`**：
  - 向数组的末尾添加一个或多个元素，并返回数组长度

- **`arr.pop()`**：
  - 删除并返回数组的最后一个元素

- **`arr.unshift(element1, element2, ..., elementn)`**：
  - 向数组的头部添加一个或多个元素，并返回数组长度

- **`arr.shift()`**：
  - 删除并返回数组的第一个元素
  - 记忆 unshift 和 shift：队列数据结构中 shift 操作表示移除队列的第一个元素，而 JS 中 shift 表示移除数组的第一个元素，并将数组长度减 1；与之对应的 unshift 则意为向数组开头添加元素

- **`arr.splice(start, number, element1, element2, ..., elementn)`**：
  - 使用指定元素，替换数组中的若干元素，并返回数组中被替换掉的元素
  - `start` 表示数组元素被替换的开始位置；`number` 表示从 `start` 开始数组被替换掉元素的数量；`element1->n` 表示替换数组元素的新元素
  - `splice` 方法可以实现数组元素的删除（即用空去替换数组中的若干元素）
  - `splice` 方法可以实现数组元素的替换（即用指定元素替换数组中的若干元素）
  - `splice` 方法可以实现向数组中插入新元素（即用指定的元素在 `start` 位置替换 0 个数组原本的元素）

- **`arr.reverse()`**：
  - 表示反转数组
```

8.7 回调函数

```
## 回调函数（callback）

回调函数是指将函数作为参数传递的一种机制。当一个函数被作为参数传递给另一个函数时，被传入的这个函数即称为回调函数。

- 通过回调函数，可以使得调用其的函数的功能变得十分灵活，甚至可以适用于不同的数据类型和不同的问题。
- 一般而言，传入函数的回调函数是匿名函数，而不是先创建一个函数对象，再将创建的函数对象作为回调函数传入。
```

示例

```javascript
// 示例：filter 函数使用回调函数来根据条件过滤数组元素
function filter(array, callback) {
    const result = [];
    for (const item of array) {
        if (callback(item)) {
            result.push(item);
        }
    }
    return result;
}

// 使用 filter 函数并传入不同的回调函数来过滤数组元素
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = filter(numbers, function (item) {
    return item % 2 === 0;
});
console.log("Even numbers:", evenNumbers);

const students = [
    { name: "Alice", score: 85 },
    { name: "Bob", score: 90 },
    { name: "Charlie", score: 75 }
];
const highScoringStudents = filter(students, function (student) {
    return student.score >= 80;
});
console.log("High-scoring students:", highScoringStudents);
```

8.8 高阶函数

```
## 高阶函数

高阶函数是指如果一个函数的参数或返回值是函数，则这个函数就称为高阶函数。

- 将函数作为参数（此时这个作为参数的函数就是回调函数），可以对另一个函数动态传递代码，示例见 [回调函数](#回调函数)。
- 将函数作为返回值，可以动态地生成一个新函数，在不修改原函数的基础上，增加其他功能，符合 OCP 开闭原则。
```

8.9 闭包

```
## 闭包

- **是什么？** 闭包是能访问到外部函数作用域中变量的函数。
- **有啥用？** 可以隐藏一些不希望被他人访问的内容。
- **三要素？**
  - 函数的嵌套；
  - 内部函数需要引用外部函数中的变量；
  - 内部函数要作为返回值返回。
```

- **注意：**
  - 闭包主要用于隐藏一些不希望被外部访问的内容，因此闭包的创建会占据一定的内存空间。
  - 相较于类，因为闭包没有原型，会比较浪费内存，一般只调用一次外部函数创建一个闭包。

闭包的原理：词法作用域

函数的作用域在函数创建时就已经确定，与调用的位置无关。

闭包的生命周期

- **创建：** 外部函数调用时产生；外部函数的每次调用都会产生一个全新的闭包。
- **销毁：** 内部函数丢失时销毁；内部函数被垃圾回收，此时闭包才会消失。

8.10 递归

递归

- **思想：** 将大问题拆解为小问题，通过小问题的解决，去解决大问题。
- **两要素：**
  - 基线条件（递归的终止条件）；
  - 递归条件（问题的拆分方式）。

递归与循环

递归思路更加清晰，但是循环的执行性能较好。

8.11 数组的高阶方法

数组的方法（高阶）

arr.sort([cb]) [破坏性方法]

用于对数组进行排序。

- 参数：`cb` 即回调函数，可以通过传入一个回调函数指定排序规则
  - `(a, b) => a - b`，表示升序排列
  - `(a, b) => b - a`，表示降序排列
- 默认情况：不传入任何参数时，`sort` 会默认按照 `Unicode` 编码（字符串比较）对数组元素进行升序排列，此时数值数组调用 `sort` 可能会得到错误结果。

arr.forEach(cb) [非破坏性方法]

用于遍历数组。

- 参数：`cb` 即回调函数，可以通过传入一个回调函数指定遍历规则
  - 数组中有几个元素，回调函数就会被调用几次
  - 回调函数的每次调用都接受三个参数，分别为 `element`、`index`、`array`，表示当前元素、当前元素的索引、被遍历的数组

arr.filter(cb) [非破坏性方法]

用于将数组中符合条件的元素保存到一个新的数组并返回。

- 参数：`cb` 即回调函数，可以通过传入一个回调函数指定选择规则

  - 数组中有几个元素，回调函数就会被调用几次
  - 回调函数的每次调用都接受一个参数，即 element，表示当前元素
  - filter 根据回调函数的返回值（布尔值）决定是否将该元素添加到新数组中

arr.map(cb) [非破坏性方法]

用于根据当前数组以一定方式生成一个新数组。

- 参数：cb 即回调函数，可以通过传入一个回调函数指定生成规则
  - 数组中有几个元素，回调函数就会被调用几次
  - 回调函数的每次调用都接受一个参数，即 element，表示当前元素
  - map 根据回调函数的返回值作为新数组中的元素

arr.reduce(cb, [initial]) [非破坏性方法]

用于将一个数组中的所有元素整合为一个值。

- 参数：cb 即回调函数，可以通过传入一个回调函数指定整合规则；initial 表示整合初始值
  - 回调函数的每次调用都接受两个参数，即 a 和 b
- 函数执行规则（假设没指定 initial）
  - 回调函数第一次调用 a=arr[0] b=arr[1]
  - 回调函数第二次调用 a=cb(arr[0], arr[1]) b=arr[2]
  - 回调函数第三次调用 a=cb(cb(arr[0], arr[1]), arr[2]) b=arr[3]
  - ......
  - 回调函数第 n 次调用 a=cb(cb(cb(...),arr[n-2]), arr[n-1]) b=arr[n]
  - 即除了第一次调用以外，a = 回调函数上次调用的结果，b = 下一个未参与计算的数组元素
  - 如果设定了 initial，则第一次调用 a=initial b=arr[0]；第二次调用 a=cb(initial, arr[0]) b=arr[1]；...

8.12 arguments

arguments

- `arguments` 是函数中的一个隐含参数（`this` 也是函数中的一个隐含参数），用于存储传递进函数的实参（无论用户是否给函数定义了形参，定义了多少形参），可以直接通过该对象访问实参。
- `arguments` 是一个类数组（伪数组）对象
  - 和数组相似：可以通过索引读取元素；有 `length` 属性；可以使用 `for-of` 语句遍历内容。
  - 与数组不同：不是数组对象；不能调用数组相关方法。
- 不足之处：无法根据函数形式了解调用函数是否需要传递参数；因为是类数组对象，无法调用数组的方法。

8.13 可变参数

可变参数

- 语法：`...参数名`
- 作用：表示可以接受任意数量的实参，并将其存储到一个以参数名为名字的数组。
- 注意
  - 与函数的隐含参数 `arguments` 不同，可变参数的名字可以自己指定。
  - 同样的，可变参数就是一个数组，可以直接使用数组的方法。
  - 可变参数可以配合其他参数一起使用，但是需要将可变参数写到最后；而 `arguments` 参数不能配合其他参数一起使用。

8.14 this 的绑定

```
# call、apply、bind

- `fun.call(this_pointer, arg1, arg2, ...)`：通过 `call` 的方式调用一个函数，其中第一个参数用于指定函数中的 `this`，后续参数为传递给函数的实参。
- `fun.apply(this_pointer, args)`：通过 `apply` 的方式调用一个函数，其中第一个参数用于指定函数中的 `this`，第二个参数为传递给函数的数组。
- `fun.bind(this_pointer, arg1, arg2, ...)`：通过 `bind` 的方式创建一个新的函数，其中第一个参数表示为新函数绑定的 `this`，之后的参数表示为新函数绑定的实参。
  - **注意**：一旦绑定了 `this` 或实参，则无法再次修改。

# 函数的 this 总结

- 函数形式调用：`this` 是 `window`。
- 方法形式调用：`this` 是调用方法的对象。
- 构造函数（constructor）中，`this` 是新创建的实例对象。
- 箭头函数没有自己的 `this`，其 `this` 由外层作用域决定。
- 使用 `call`、`apply` 调用函数，可以通过第一个参数指定函数的 `this`。
- 使用 `bind` 返回函数，可以通过第一个参数指定函数的 `this`（无法修改）。
- 箭头函数没有自己的 `this`，因此也无法通过 `call`、`apply`、`bind` 修改其 `this`（箭头函数也没有 `arguments`）。
- 通过 `call`、`apply` 的第一个参数，可以指定调用的函数的 `this`。
- 通过 `bind` 返回的函数，无法再次改变其 `this`。
```

9 内建对象

9.1 解构赋值

13-解构赋值[内建对象].html

```
### 解构赋值

1. **数组的解构赋值**
    - 语法：`[arg1, arg2, ...] = [value1, value2, ...]` 或 `[arg1, arg2, ...] = arr`
    - 变量与值数量不一的情况：
        - 左边变量数 > 右边值的数：多余的变量被赋值为 `undefined`
        - 左边变量数 < 右边值的数：多余的值被忽略
    - 变量默认值：如果左边存在多余的变量，则可以指定默认值
    - 使用 `...arg` 获取右边多余的值，类似于可变参数
    - 可以快速交换两个变量的值

2. **二维数组的解构赋值**
    - 二维数组：一个数组的元素还是数组
    - 解构赋值：使用数组嵌套的形式进行解构赋值

3. **对象的解构赋值**
    - 语法：`[name, age, ...] = [name: "张三", age: "李四", ...]` 或 `[arg1, arg2, ...] = obj`
    - 变量别名：使用 `{name: a, age: b}` 可以指定变量别名
    - 变量默认值：如果没有找到对应的属性，则使用默认值
```

更多笔记

```
### 9.2 对象序列化

### 9.3 Map、Set 对象

### 9.4 Math 工具类

### 9.5 Date 对象

### 9.6 包装类

### 9.7 字符串常用放啊

### 9.8 正则表达式

### 9.9 字符串正则方法

### 9.10 垃圾回收

## 10 DOM
```