声明

* **const命令**：声明常量
* **let命令**：声明变量

作用

* 作用域
  + **全局作用域**
  + **函数作用域**：function() {}
  + **块级作用域**：{}
* 作用范围
  + var命令在全局代码中执行
  + const命令和let命令只能在代码块中执行
* 赋值使用
  + const命令声明常量后必须立马赋值
  + let命令声明变量后可立马赋值或使用时赋值
* 声明方法：var、const、let、function、class、import

重点难点

* 不允许重复声明
* 未定义就使用会报错：const命令和let命令不存在变量提升
* 暂时性死区：在代码块内使用let命令声明变量之前，该变量都不可用

### 解构赋值

* **字符串解构**：const [a, b, c, d, e] = "hello"
* **数值解构**：const { toString: s } = 123
* **布尔值解构**：const { toString: b } = true
* **对象解构**
  + 形式：const { x, y } = { x: 1, y: 2 }
  + 默认：const { x, y = 2 } = { x: 1 }
  + 改名：const { x, y: z } = { x: 1, y: 2 }
* **数组解构**
  + 规则：数据结构具有Iterator接口可采用数组形式的解构赋值
  + 形式：const [x, y] = [1, 2]
  + 默认：const [x, y = 2] = [1]
* **函数参数解构**
  + 数组解构：function Func([x = 0, y = 1]) {}
  + 对象解构：function Func({ x = 0, y = 1 } = {}) {}

应用场景

* 交换变量值：[x, y] = [y, x]
* 返回函数多个值：const [x, y, z] = Func()
* 定义函数参数：Func([1, 2])
* 提取JSON数据：const { name, version } = packageJson
* 定义函数参数默认值：function Func({ x = 1, y = 2 } = {}) {}
* 遍历Map结构：for (let [k, v] of Map) {}
* 输入模块指定属性和方法：const { readFile, writeFile } = require("fs")

重点难点

* 匹配模式：只要等号两边的模式相同，左边的变量就会被赋予对应的值
* 解构赋值规则：只要等号右边的值不是对象或数组，就先将其转为对象
* 解构默认值生效条件：属性值严格等于undefined
* 解构遵循匹配模式
* 解构不成功时变量的值等于undefined
* undefined和null无法转为对象，因此无法进行解构

### 字符串扩展

* **Unicode表示法**：大括号包含表示Unicode字符(\u{0xXX}或\u{0XXX})
* **字符串遍历**：可通过for-of遍历字符串
* **字符串模板**：可单行可多行可插入变量的增强版字符串
* **标签模板**：函数参数的特殊调用
* **repeat()**：把字符串重复n次，返回新字符串
* **includes()**：是否存在指定字符串
* **startsWith()**：是否存在字符串头部指定字符串
* **endsWith()**：是否存在字符串尾部指定字符串
* **String.raw()**：返回把字符串所有变量替换且对斜杠进行转义的结果
* **String.fromCodePoint()**：返回码点对应字符
* **codePointAt()**：返回字符对应码点(String.fromCodePoint()的逆操作)
* **normalize()**：把字符的不同表示方法统一为同样形式，返回新字符串(Unicode正规化)
* **matchAll()**：返回正则表达式在字符串的所有匹配

重点难点

* 以上扩展方法均可作用于由4个字节储存的Unicode字符上

### 数值扩展

* **二进制表示法**：0b或0B开头表示二进制(0bXX或0BXX)
* **八进制表示法**：0o或0O开头表示二进制(0oXX或0OXX)
* **Number.EPSILON**：数值最小精度
* **Number.MIN\_SAFE\_INTEGER**：最小安全数值(-2^53)
* **Number.MAX\_SAFE\_INTEGER**：最大安全数值(2^53)
* **Number.parseInt()**：返回转换值的整数部分
* **Number.parseFloat()**：返回转换值的浮点数部分
* **Number.isFinite()**：是否为有限数值
* **Number.isNaN()**：是否为NaN
* **Number.isInteger()**：是否为整数
* **Number.isSafeInteger()**：是否在数值安全范围内
* **Math.trunc()**：返回数值整数部分
* **Math.sign()**：返回数值类型(正数1、负数-1、零0)
* **Math.cbrt()**：返回数值立方根
* **Math.clz32()**：返回数值的32位无符号整数形式
* **Math.imul()**：返回两个数值相乘
* **Math.fround()**：返回数值的32位单精度浮点数形式
* **Math.hypot()**：返回所有数值平方和的平方根
* **Math.expm1()**：返回e^n - 1
* **Math.log1p()**：返回1 + n的自然对数(Math.log(1 + n))
* **Math.log10()**：返回以10为底的n的对数
* **Math.log2()**：返回以2为底的n的对数
* **Math.sinh()**：返回n的双曲正弦
* **Math.cosh()**：返回n的双曲余弦
* **Math.tanh()**：返回n的双曲正切
* **Math.asinh()**：返回n的反双曲正弦
* **Math.acosh()**：返回n的反双曲余弦
* **Math.atanh()**：返回n的反双曲正切

### 对象扩展

* **简洁表示法**：直接写入变量和函数作为对象的属性和方法({ prop, method() {} })
* **属性名表达式**：字面量定义对象时使用[]定义键([prop]，不能与上同时使用)
* **方法的name属性**：返回方法函数名
  + 取值函数(getter)和存值函数(setter)：get/set 函数名(属性的描述对象在get和set上)
  + bind返回的函数：bound 函数名
  + Function构造函数返回的函数实例：anonymous
* **属性的可枚举性和遍历**：描述对象的enumerable
* **super关键字**：指向当前对象的原型对象(只能用在对象的简写方法中method() {})
* **Object.is()**：对比两值是否相等
* **Object.assign()**：合并对象(浅拷贝)，返回原对象
* **Object.getPrototypeOf()**：返回对象的原型对象
* **Object.setPrototypeOf()**：设置对象的原型对象
* **\_\_proto\_\_**：返回或设置对象的原型对象

属性遍历

* 描述：自身、可继承、可枚举、非枚举、Symbol
* 遍历
  + for-in：遍历对象自身可继承可枚举属性
  + Object.keys()：返回对象自身可枚举属性的键组成的数组
  + Object.getOwnPropertyNames()：返回对象自身可继承可枚举非枚举属性的键组成的数组
  + Object.getOwnPropertySymbols()：返回对象Symbol属性的键组成的数组
  + Reflect.ownKeys()：返回对象自身可继承可枚举非枚举Symbol属性的键组成的数组
* 规则
  + 首先遍历所有数值键，按照数值升序排列
  + 其次遍历所有字符串键，按照加入时间升序排列
  + 最后遍历所有Symbol键，按照加入时间升序排列

### 数组扩展

* **扩展运算符(...)**：转换数组为用逗号分隔的参数序列([...arr]，相当于rest/spread参数的逆运算)
* **Array.from()**：转换具有Iterator接口的数据结构为真正数组，返回新数组
  + 类数组对象：包含length的对象、Arguments对象、NodeList对象
  + 可遍历对象：String、Set结构、Map结构、Generator函数
* **Array.of()**：转换一组值为真正数组，返回新数组
* **copyWithin()**：把指定位置的成员复制到其他位置，返回原数组
* **find()**：返回第一个符合条件的成员
* **findIndex()**：返回第一个符合条件的成员索引值
* **fill()**：根据指定值填充整个数组，返回原数组
* **keys()**：返回以索引值为遍历器的对象
* **values()**：返回以属性值为遍历器的对象
* **entries()**：返回以索引值和属性值为遍历器的对象
* **数组空位**：ES6明确将数组空位转为undefined(空位处理规不一，建议避免出现)

扩展应用

* 克隆数组：const arr = [...arr1]
* 合并数组：const arr = [...arr1, ...arr2]
* 拼接数组：arr.push(...arr1)
* 代替apply：Math.max.apply(null, [x, y]) => Math.max(...[x, y])
* 转换字符串为数组：[..."hello"]
* 转换类数组对象为数组：[...Arguments, ...NodeList]
* 转换可遍历对象为数组：[...String, ...Set, ...Map, ...Generator]
* 与数组解构赋值结合：const [x, ...rest/spread] = [1, 2, 3]
* 计算Unicode字符长度：Array.from("hello").length => [..."hello"].length

重点难点

* 使用keys()、values()、entries()返回的遍历器对象，可用for-of自动遍历或next()手动遍历

### 函数扩展

* **参数默认值**：为函数参数指定默认值
  + 形式：function Func(x = 1, y = 2) {}
  + 参数赋值：惰性求值(函数调用后才求值)
  + 参数位置：尾参数
  + 参数作用域：函数作用域
  + 声明方式：默认声明，不能用const或let再次声明
  + length：返回没有指定默认值的参数个数
  + 与解构赋值默认值结合：function Func({ x = 1, y = 2 } = {}) {}
  + 应用
    - 指定某个参数不得省略，省略即抛出错误：function Func(x = throwMissing()) {}
    - 将参数默认值设为undefined，表明此参数可省略：Func(undefined, 1)
* **rest/spread参数(...)**：返回函数多余参数
  + 形式：以数组的形式存在，之后不能再有其他参数
  + 作用：代替Arguments对象
  + length：返回没有指定默认值的参数个数但不包括rest/spread参数
* **严格模式**：在严格条件下运行JS
  + 应用：只要函数参数使用默认值、解构赋值、扩展运算符，那么函数内部就不能显式设定为严格模式
* **name属性**：返回函数的函数名
  + 将匿名函数赋值给变量：空字符串(**ES5**)、变量名(**ES6**)
  + 将具名函数赋值给变量：函数名(**ES5和ES6**)
  + bind返回的函数：bound 函数名(**ES5和ES6**)
  + Function构造函数返回的函数实例：anonymous(**ES5和ES6**)
* **箭头函数(=>)**：函数简写
  + 无参数：() => {}
  + 单个参数：x => {}
  + 多个参数：(x, y) => {}
  + 解构参数：({x, y}) => {}
  + 嵌套使用：部署管道机制
  + this指向固定化
    - 并非因为内部有绑定this的机制，而是根本没有自己的this，导致内部的this就是外层代码块的this
    - 因为没有this，因此不能用作构造函数
* **尾调用优化**：只保留内层函数的调用帧
  + 尾调用
    - 定义：某个函数的最后一步是调用另一个函数
    - 形式：function f(x) { return g(x); }
  + 尾递归
    - 定义：函数尾调用自身
    - 作用：只要使用尾递归就不会发生栈溢出，相对节省内存
    - 实现：把所有用到的内部变量改写成函数的参数并使用参数默认值

箭头函数误区

* 函数体内的this是定义时所在的对象而不是使用时所在的对象
* 可让this指向固定化，这种特性很有利于封装回调函数
* 不可当作构造函数，因此箭头函数不可使用new命令
* 不可使用yield命令，因此箭头函数不能用作Generator函数
* 不可使用Arguments对象，此对象在函数体内不存在(可用rest/spread参数代替)
* 返回对象时必须在对象外面加上括号

### 正则扩展

* **变更RegExp构造函数入参**：允许首参数为正则对象，尾参数为正则修饰符(返回的正则表达式会忽略原正则表达式的修饰符)
* **正则方法调用变更**：字符串对象的match()、replace()、search()、split()内部调用转为调用RegExp实例对应的RegExp.prototype[Symbol.方法]
* **u修饰符**：Unicode模式修饰符，正确处理大于\uFFFF的Unicode字符
  + 点字符(.)
  + Unicode表示法
  + 量词
  + 预定义模式
  + i修饰符
  + 转义
* **y修饰符**：粘连修饰符，确保匹配必须从剩余的第一个位置开始全局匹配(与g修饰符作用类似)
* **unicode**：是否设置u修饰符
* **sticky**：是否设置y修饰符
* **flags**：正则表达式的修饰符

重点难点

* y修饰符隐含头部匹配标志^
* 单单一个y修饰符对match()只能返回第一个匹配，必须与g修饰符联用才能返回所有匹配

### Symbol

* 定义：独一无二的值
* 声明：const set = Symbol(str)
* 入参：字符串(可选)
* 方法
  + **Symbol()**：创建以参数作为描述的Symbol值(不登记在全局环境)
  + **Symbol.for()**：创建以参数作为描述的Symbol值，如存在此参数则返回原有的Symbol值(先搜索后创建，登记在全局环境)
  + **Symbol.keyFor()**：返回已登记的Symbol值的描述(只能返回Symbol.for()的key)
  + **Object.getOwnPropertySymbols()**：返回对象中所有用作属性名的Symbol值的数组
* 内置
  + **Symbol.hasInstance**：指向一个内部方法，当其他对象使用instanceof运算符判断是否为此对象的实例时会调用此方法
  + **Symbol.isConcatSpreadable**：指向一个布尔值，定义对象用于Array.prototype.concat()时是否可展开
  + **Symbol.species**：指向一个构造函数，当实例对象使用自身构造函数时会调用指定的构造函数
  + **Symbol.match**：指向一个函数，当实例对象被String.prototype.match()调用时会重新定义match()的行为
  + **Symbol.replace**：指向一个函数，当实例对象被String.prototype.replace()调用时会重新定义replace()的行为
  + **Symbol.search**：指向一个函数，当实例对象被String.prototype.search()调用时会重新定义search()的行为
  + **Symbol.split**：指向一个函数，当实例对象被String.prototype.split()调用时会重新定义split()的行为
  + **Symbol.iterator**：指向一个默认遍历器方法，当实例对象执行for-of时会调用指定的默认遍历器
  + **Symbol.toPrimitive**：指向一个函数，当实例对象被转为原始类型的值时会返回此对象对应的原始类型值
  + **Symbol.toStringTag**：指向一个函数，当实例对象被Object.prototype.toString()调用时其返回值会出现在toString()返回的字符串之中表示对象的类型
  + **Symbol.unscopables**：指向一个对象，指定使用with时哪些属性会被with环境排除

数据类型

* **Undefined**
* **Null**
* **String**
* **Number**
* **Boolean**
* **Object**(包含Array、Function、Date、RegExp、Error)
* **Symbol**

应用场景

* 唯一化对象属性名：属性名属于Symbol类型，就都是独一无二的，可保证不会与其他属性名产生冲突
* 消除魔术字符串：在代码中多次出现且与代码形成强耦合的某一个具体的字符串或数值
* 遍历属性名：无法通过for-in、for-of、Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回，只能通过Object.getOwnPropertySymbols返回
* 启用模块的Singleton模式：调用一个类在任何时候返回同一个实例(window和global)，使用Symbol.for()来模拟全局的Singleton模式

重点难点

* Symbol()生成一个原始类型的值不是对象，因此Symbol()前不能使用new命令
* Symbol()参数表示对当前Symbol值的描述，相同参数的Symbol()返回值不相等
* Symbol值不能与其他类型的值进行运算
* Symbol值可通过String()或toString()显式转为字符串
* Symbol值作为对象属性名时，此属性是公开属性，但不是私有属性
* Symbol值作为对象属性名时，只能用方括号运算符([])读取，不能用点运算符(.)读取
* Symbol值作为对象属性名时，不会被常规方法遍历得到，可利用此特性为对象定义非私有但又只用于内部的方法

### Set

##### Set

* 定义：类似于数组的数据结构，成员值都是唯一且没有重复的值
* 声明：const set = new Set(arr)
* 入参：具有Iterator接口的数据结构
* 属性
  + **constructor**：构造函数，返回Set
  + **size**：返回实例成员总数
* 方法
  + **add()**：添加值，返回实例
  + **delete()**：删除值，返回布尔值
  + **has()**：检查值，返回布尔值
  + **clear()**：清除所有成员
  + **keys()**：返回以属性值为遍历器的对象
  + **values()**：返回以属性值为遍历器的对象
  + **entries()**：返回以属性值和属性值为遍历器的对象
  + **forEach()**：使用回调函数遍历每个成员

应用场景

* 去重字符串：[...new Set(str)].join("")
* 去重数组：[...new Set(arr)]或Array.from(new Set(arr))
* 集合数组
  + 声明：const a = new Set(arr1)、const b = new Set(arr2)
  + 并集：new Set([...a, ...b])
  + 交集：new Set([...a].filter(v => b.has(v)))
  + 差集：new Set([...a].filter(v => !b.has(v)))
* 映射集合
  + 声明：let set = new Set(arr)
  + 映射：set = new Set([...set].map(v => v \* 2))或set = new Set(Array.from(set, v => v \* 2))

重点难点

* 遍历顺序：插入顺序
* 没有键只有值，可认为键和值两值相等
* 添加多个NaN时，只会存在一个NaN
* 添加相同的对象时，会认为是不同的对象
* 添加值时不会发生类型转换(5 !== "5")
* keys()和values()的行为完全一致，entries()返回的遍历器同时包括键和值且两值相等

##### WeakSet

* 定义：和Set结构类似，成员值只能是对象
* 声明：const set = new WeakSet(arr)
* 入参：具有Iterator接口的数据结构
* 属性
  + **constructor**：构造函数，返回WeakSet
* 方法
  + **add()**：添加值，返回实例
  + **delete()**：删除值，返回布尔值
  + **has()**：检查值，返回布尔值

应用场景

* 储存DOM节点：DOM节点被移除时自动释放此成员，不用担心这些节点从文档移除时会引发内存泄漏
* 临时存放一组对象或存放跟对象绑定的信息：只要这些对象在外部消失，它在WeakSet结构中的引用就会自动消

重点难点

* 成员都是弱引用，垃圾回收机制不考虑WeakSet结构对此成员的引用
* 成员不适合引用，它会随时消失，因此ES6规定WeakSet结构不可遍历
* 其他对象不再引用成员时，垃圾回收机制会自动回收此成员所占用的内存，不考虑此成员是否还存在于WeakSet结构中

### Map

##### Map

* 定义：类似于对象的数据结构，成员键可以是任何类型的值
* 声明：const set = new Map(arr)
* 入参：具有Iterator接口且每个成员都是一个双元素数组的数据结构
* 属性
  + **constructor**：构造函数，返回Map
  + **size**：返回实例成员总数
* 方法
  + **get()**：返回键值对
  + **set()**：添加键值对，返回实例
  + **delete()**：删除键值对，返回布尔值
  + **has()**：检查键值对，返回布尔值
  + **clear()**：清除所有成员
  + **keys()**：返回以键为遍历器的对象
  + **values()**：返回以值为遍历器的对象
  + **entries()**：返回以键和值为遍历器的对象
  + **forEach()**：使用回调函数遍历每个成员

重点难点

* 遍历顺序：插入顺序
* 对同一个键多次赋值，后面的值将覆盖前面的值
* 对同一个对象的引用，被视为一个键
* 对同样值的两个实例，被视为两个键
* 键跟内存地址绑定，只要内存地址不一样就视为两个键
* 添加多个以NaN作为键时，只会存在一个以NaN作为键的值
* Object结构提供字符串—值的对应，Map结构提供值—值的对应

##### WeakMap

* 定义：和Map结构类似，成员键只能是对象
* 声明：const set = new WeakMap(arr)
* 入参：具有Iterator接口且每个成员都是一个双元素数组的数据结构
* 属性
  + **constructor**：构造函数，返回WeakMap
* 方法
  + **get()**：返回键值对
  + **set()**：添加键值对，返回实例
  + **delete()**：删除键值对，返回布尔值
  + **has()**：检查键值对，返回布尔值

应用场景

* 储存DOM节点：DOM节点被移除时自动释放此成员键，不用担心这些节点从文档移除时会引发内存泄漏
* 部署私有属性：内部属性是实例的弱引用，删除实例时它们也随之消失，不会造成内存泄漏

重点难点

* 成员键都是弱引用，垃圾回收机制不考虑WeakMap结构对此成员键的引用
* 成员键不适合引用，它会随时消失，因此ES6规定WeakMap结构不可遍历
* 其他对象不再引用成员键时，垃圾回收机制会自动回收此成员所占用的内存，不考虑此成员是否还存在于WeakMap结构中
* 一旦不再需要，成员会自动消失，不用手动删除引用
* 弱引用的只是键而不是值，值依然是正常引用
* 即使在外部消除了成员键的引用，内部的成员值依然存在

### Proxy

* 定义：修改某些操作的默认行为
* 声明：const proxy = new Proxy(target, handler)都为对象形式
* 入参
  + **target**：拦截的目标对象
  + **handler**：定制拦截行为
* 方法
  + **Proxy.revocable()**：返回可取消的Proxy实例(返回{ proxy, revoke }，通过revoke()取消代理)
* 拦截方式
  + **get()**：拦截对象属性读取
  + **set()**：拦截对象属性设置，返回布尔值
  + **has()**：拦截对象属性检查k in obj，返回布尔值
  + **deleteProperty()**：拦截对象属性删除delete obj[k]，返回布尔值
  + **defineProperty()**：拦截对象属性定义Object.defineProperty()、Object.defineProperties()，返回布尔值
  + **ownKeys()**：拦截对象属性遍历for-in、Object.keys()、Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()，返回数组
  + **getOwnPropertyDescriptor()**：拦截对象属性描述读取Object.getOwnPropertyDescriptor()，返回对象
  + **getPrototypeOf()**：拦截对象原型读取instanceof、Object.getPrototypeOf()、Object.prototype.\_\_proto\_\_、Object.prototype.isPrototypeOf()、Reflect.getPrototypeOf()，返回对象
  + **setPrototypeOf()**：拦截对象原型设置Object.setPrototypeOf()，返回布尔值
  + **isExtensible()**：拦截对象是否可扩展读取Object.isExtensible()，返回布尔值
  + **preventExtensions()**：拦截对象不可扩展设置Object.preventExtensions()，返回布尔值
  + **apply()**：拦截Proxy实例作为函数调用proxy()、proxy.apply()、proxy.call()
  + **construct()**：拦截Proxy实例作为构造函数调用new proxy()

应用场景

* Proxy.revocable()：不允许直接访问对象，必须通过代理访问，一旦访问结束就收回代理权不允许再次访问
* get()：读取未知属性报错、读取数组负数索引的值、封装链式操作、生成DOM嵌套节点
* set()：数据绑定(Vue数据绑定实现原理)、确保属性值设置符合要求、防止内部属性被外部读写
* has()：隐藏内部属性不被发现、排除不符合属性条件的对象
* deleteProperty()：保护内部属性不被删除
* defineProperty()：阻止属性被外部定义
* ownKeys()：保护内部属性不被遍历

重点难点

* 要使Proxy起作用，必须针对实例进行操作，而不是针对目标对象进行操作
* 没有设置任何拦截时，等同于直接通向原对象
* 属性被定义为不可读写/扩展/配置/枚举时，使用拦截方法会报错
* 代理下的目标对象，内部this指向Proxy代理

### Reflect

* 定义：保持Object方法的默认行为
* 方法
  + **get()**：返回对象属性
  + **set()**：设置对象属性，返回布尔值
  + **has()**：检查对象属性，返回布尔值
  + **deleteProperty()**：删除对象属性，返回布尔值
  + **defineProperty()**：定义对象属性，返回布尔值
  + **ownKeys()**：遍历对象属性，返回数组(Object.getOwnPropertyNames()+Object.getOwnPropertySymbols())
  + **getOwnPropertyDescriptor()**：返回对象属性描述，返回对象
  + **getPrototypeOf()**：返回对象原型，返回对象
  + **setPrototypeOf()**：设置对象原型，返回布尔值
  + **isExtensible()**：返回对象是否可扩展，返回布尔值
  + **preventExtensions()**：设置对象不可扩展，返回布尔值
  + **apply()**：绑定this后执行指定函数
  + **construct()**：调用构造函数创建实例

设计目的

* 将Object属于语言内部的方法放到Reflect上
* 将某些Object方法报错情况改成返回false
* 让Object操作变成函数行为
* Proxy与Reflect相辅相成

废弃方法

* Object.defineProperty() => Reflect.defineProperty()
* Object.getOwnPropertyDescriptor() => Reflect.getOwnPropertyDescriptor()

重点难点

* Proxy方法和Reflect方法一一对应
* Proxy和Reflect联合使用，前者负责拦截赋值操作，后者负责完成赋值操作

数据绑定：观察者模式

**const** observerQueue = **new** Set();

**const** observe = fn => observerQueue.add(fn);

**const** observable = obj => **new** Proxy(obj, {

set(tgt, key, val, receiver) {

**const** result = Reflect.set(tgt, key, val, receiver);

observerQueue.forEach(v => v());

**return** result;

}

});

**const** person = observable({ age: 25, name: "Yajun" });

**const** print = () => console.log(`${person.name} is ${person.age} years old`);

observe(print);

person.name = "Joway";

### Class

* 定义：对一类具有共同特征的事物的抽象(构造函数语法糖)
* 原理：类本身指向构造函数，所有方法定义在prototype上，可看作构造函数的另一种写法(Class === Class.prototype.constructor)
* 方法和关键字
  + **constructor()**：构造函数，new命令生成实例时自动调用
  + **extends**：继承父类
  + **super**：新建父类的this
  + **static**：定义静态属性方法
  + **get**：取值函数，拦截属性的取值行为
  + **set**：存值函数，拦截属性的存值行为
* 属性
  + **\_\_proto\_\_**：构造函数的继承(总是指向父类)
  + **\_\_proto\_\_.\_\_proto\_\_**：子类的原型的原型，即父类的原型(总是指向父类的\_\_proto\_\_)
  + **prototype.\_\_proto\_\_**：属性方法的继承(总是指向父类的prototype)
* 静态属性：定义类完成后赋值属性，该属性不会被实例继承，只能通过类来调用
* 静态方法：使用static定义方法，该方法不会被实例继承，只能通过类来调用(方法中的this指向类，而不是实例)
* 继承
  + 实质
    - ES5实质：先创造子类实例的this，再将父类的属性方法添加到this上(Parent.apply(this))
    - ES6实质：先将父类实例的属性方法加到this上(调用super())，再用子类构造函数修改this
  + super
    - 作为函数调用：只能在构造函数中调用super()，内部this指向继承的当前子类(super()调用后才可在构造函数中使用this)
    - 作为对象调用：在普通方法中指向父类的原型对象，在静态方法中指向父类
  + 显示定义：使用constructor() { super(); }定义继承父类，没有书写则显示定义
  + 子类继承父类：子类使用父类的属性方法时，必须在构造函数中调用super()，否则得不到父类的this
    - 父类静态属性方法可被子类继承
    - 子类继承父类后，可从super上调用父类静态属性方法
* 实例：类相当于实例的原型，所有在类中定义的属性方法都会被实例继承
  + 显式指定属性方法：使用this指定到自身上(使用Class.hasOwnProperty()可检测到)
  + 隐式指定属性方法：直接声明定义在对象原型上(使用Class.\_\_proto\_\_.hasOwnProperty()可检测到)
* 表达式
  + 类表达式：const Class = class {}
  + name属性：返回紧跟class后的类名
  + 属性表达式：[prop]
  + Generator方法：\* mothod() {}
  + Async方法：async mothod() {}
* this指向：解构实例属性或方法时会报错
  + 绑定this：this.mothod = this.mothod.bind(this)
  + 箭头函数：this.mothod = () => this.mothod()
* 属性定义位置
  + 定义在构造函数中并使用this指向
  + 定义在类最顶层
* **new.target**：确定构造函数是如何调用

原生构造函数

* **String()**
* **Number()**
* **Boolean()**
* **Array()**
* **Object()**
* **Function()**
* **Date()**
* **RegExp()**
* **Error()**

重点难点

* 在实例上调用方法，实质是调用原型上的方法
* Object.assign()可方便地一次向类添加多个方法(Object.assign(Class.prototype, { ... }))
* 类内部所有定义的方法是不可枚举的(non-enumerable)
* 构造函数默认返回实例对象(this)，可指定返回另一个对象
* 取值函数和存值函数设置在属性的Descriptor对象上
* 类不存在变量提升
* 利用new.target === Class写出不能独立使用必须继承后才能使用的类
* 子类继承父类后，this指向子类实例，通过super对某个属性赋值，赋值的属性会变成子类实例的属性
* 使用super时，必须显式指定是作为函数还是作为对象使用
* extends不仅可继承类还可继承原生的构造函数

私有属性方法

**const** name = Symbol("name");

**const** print = Symbol("print");

**class** **Person** {

**constructor**(age) {

**this**[name] = "Bruce";

**this**.age = age;

}

[print]() {

console.log(`${**this**[name]} is ${**this**.age} years old`);

}

}

继承混合类

**function** **CopyProperties**(target, source) {

**for** (**const** key **of** Reflect.ownKeys(source)) {

**if** (key !== "constructor" && key !== "prototype" && key !== "name") {

**const** desc = Object.getOwnPropertyDescriptor(source, key);

Object.defineProperty(target, key, desc);

}

}

}

**function** **MixClass**(...mixins) {

**class** **Mix** {

**constructor**() {

**for** (**const** mixin **of** mixins) {

CopyProperties(**this**, **new** mixin());

}

}

}

**for** (**const** mixin **of** mixins) {

CopyProperties(Mix, mixin);

CopyProperties(Mix.prototype, mixin.prototype);

}

**return** Mix;

}

**class** **Student** **extends** **MixClass**(**Person**, **Kid**) {}

### Module

* 命令
  + **export**：规定模块对外接口
    - **默认导出**：export default Person(导入时可指定模块任意名称，无需知晓内部真实名称)
    - **单独导出**：export const name = "Bruce"
    - **按需导出**：export { age, name, sex }(推荐)
    - **改名导出**：export { name as newName }
  + **import**：导入模块内部功能
    - **默认导入**：import Person from "person"
    - **整体导入**：import \* as Person from "person"
    - **按需导入**：import { age, name, sex } from "person"
    - **改名导入**：import { name as newName } from "person"
    - **自执导入**：import "person"
    - **复合导入**：import Person, { name } from "person"
  + **复合模式**：export命令和import命令结合在一起写成一行，变量实质没有被导入当前模块，相当于对外转发接口，导致当前模块无法直接使用其导入变量
    - **默认导入导出**：export { default } from "person"
    - **整体导入导出**：export \* from "person"
    - **按需导入导出**：export { age, name, sex } from "person"
    - **改名导入导出**：export { name as newName } from "person"
    - **具名改默认导入导出**：export { name as default } from "person"
    - **默认改具名导入导出**：export { default as name } from "person"
* 继承：默认导出和改名导出结合使用可使模块具备继承性
* 设计思想：尽量地静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量
* 严格模式：ES6模块自动采用严格模式(不管模块头部是否添加use strict)

模块方案

* **CommonJS**：用于服务器(动态化依赖)
* **AMD**：用于浏览器(动态化依赖)
* **CMD**：用于浏览器(动态化依赖)
* **UMD**：用于浏览器和服务器(动态化依赖)
* **ESM**：用于浏览器和服务器(静态化依赖)

加载方式

* **运行时加载**
  + 定义：整体加载模块生成一个对象，再从对象上获取需要的属性和方法进行加载(全部加载)
  + 影响：只有运行时才能得到这个对象，导致无法在编译时做静态优化
* **编译时加载**
  + 定义：直接从模块中获取需要的属性和方法进行加载(按需加载)
  + 影响：在编译时就完成模块加载，效率比其他方案高，但无法引用模块本身(**本身不是对象**)，可拓展JS高级语法(**宏和类型校验**)

加载实现

* **传统加载**：通过<script>进行同步或异步加载脚本
  + 同步加载：<script src=""></script>
  + Defer异步加载：<script src="" defer></script>(顺序加载，渲染完再执行)
  + Async异步加载：<script src="" async></script>(乱序加载，下载完就执行)
* **模块加载**：<script type="module" src=""></script>(默认是Defer异步加载)

CommonJS和ESM的区别

* CommonJS输出值的拷贝，ESM输出值的引用
  + CommonJS一旦输出一个值，模块内部的变化就影响不到这个值
  + ESM是动态引用且不会缓存值，模块里的变量绑定其所在的模块，等到脚本真正执行时，再根据这个只读引用到被加载的那个模块里去取值
* CommonJS是运行时加载，ESM是编译时加载
  + CommonJS加载模块是对象(即module.exports)，该对象只有在脚本运行完才会生成
  + ESM加载模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成

Node加载

* 背景：CommonJS和ESM互不兼容，目前解决方案是将两者分开，采用各自的加载方案
* 区分：要求ESM采用.mjs后缀文件名
  + require()不能加载.mjs文件，只有import命令才可加载.mjs文件
  + .mjs文件里不能使用require()，必须使用import命令加载文件
* 驱动：node --experimental-modules file.mjs
* 限制：Node的import命令目前只支持加载本地模块(file:协议)，不支持加载远程模块
* 加载优先级
  + 脚本文件省略后缀名：依次尝试加载四个后缀名文件(.mjs、.js、.json、node)
  + 以上不存在：尝试加载package.json的main字段指定的脚本
  + 以上不存在：依次尝试加载名称为index四个后缀名文件(.mjs、.js、.json、node)
  + 以上不存在：报错
* 不存在的内部变量：arguments、exports、module、require、this、\_\_dirname、\_\_filename
* CommonJS加载ESM
  + 不能使用require()，只能使用import()
* ESM加载CommonJS
  + 自动将module.exports转化成export default
  + CommonJS输出缓存机制在ESM加载方式下依然有效
  + 采用import命令加载CommonJS模块时，不允许采用按需导入，应使用默认导入或整体导入

循环加载

* 定义：脚本A的执行依赖脚本B，而脚本A的执行又依赖脚本B
* 加载原理
  + CommonJS：require()首次加载脚本就会执行整个脚本，在内存里生成一个对象缓存下来，二次加载脚本时直接从缓存中获取
  + ESM：import命令加载变量不会被缓存，而是成为一个指向被加载模块的引用
* 循环加载
  + CommonJS：只输出已经执行的部分，还未执行的部分不会输出
  + ESM：需开发者自己保证真正取值时能够取到值(可把变量写成函数形式，函数具有提升作用)

重点难点

* ES6模块中，顶层this指向undefined，不应该在顶层代码使用this
* 一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取
* export命令输出的接口与其对应的值是动态绑定关系，即通过该接口可获取模块内部实时的值
* import命令大括号里的变量名必须与被导入模块对外接口的名称相同
* import命令输入的变量只读(本质是输入接口)，不允许在加载模块的脚本里改写接口
* import命令命令具有提升效果，会提升到整个模块的头部，首先执行
* 重复执行同一句import语句，只会执行一次
* export default命令只能使用一次
* export default命令导出的整体模块，在执行import命令时其后不能跟大括号
* export default命令本质是输出一个名为default的变量，后面不能跟变量声明语句
* export default命令本质是将后面的值赋给名为default的变量，可直接将值写在其后
* export default命令和export {}命令可同时存在，对应复合导入
* export命令和import命令可出现在模块任何位置，只要处于模块顶层即可，不能处于块级作用域
* import()加载模块成功后，此模块会作为一个对象，当作then()的参数，可使用对象解构赋值来获取输出接口
* 同时动态加载多个模块时，可使用Promise.all()和import()相结合来实现
* import()和结合async/await来书写同步操作的代码

单例模式：跨模块常量

// 常量跨文件共享

// person.js

**const** NAME = "Bruce";

**const** AGE = 25;

**const** SEX = "male";

**export** { AGE, NAME, SEX };

// file1.js

**import** { AGE } **from** "person";

console.log(AGE);

// file2.js

**import** { AGE, NAME, SEX } **from** "person";

console.log(AGE, NAME, SEX);

默认导入互换整体导入

**import** Person **from** "person";

console.log(Person.AGE);

**import** \* **as** Person **from** "person";

console.log(Person.default.AGE);

### Iterator

* 定义：为各种不同的数据结构提供统一的访问机制
* 原理：创建一个指针指向首个成员，按照次序使用next()指向下一个成员，直接到结束位置(数据结构只要部署Iterator接口就可完成遍历操作)
* 作用
  + 为各种数据结构提供一个统一的简便的访问接口
  + 使得数据结构成员能够按某种次序排列
  + ES6创造了新的遍历命令for-of，Iterator接口主要供for-of消费
* 形式：for-of(自动去寻找Iterator接口)
* 数据结构
  + 集合：Array、Object、Set、Map
  + 原生具备接口的数据结构：String、Array、Set、Map、TypedArray、Arguments、NodeList
* 部署：默认部署在Symbol.iterator(具备此属性被认为可遍历的iterable)
* 遍历器对象
  + **next()**：下一步操作，返回{ done, value }(必须部署)
  + **return()**：for-of提前退出调用，返回{ done: true }
  + **throw()**：不使用，配合Generator函数使用

ForOf循环

* 定义：调用Iterator接口产生遍历器对象(for-of内部调用数据结构的Symbol.iterator())
* 遍历字符串：for-in获取索引，for-of获取值(可识别32位UTF-16字符)
* 遍历数组：for-in获取索引，for-of获取值
* 遍历对象：for-in获取键，for-of需自行部署
* 遍历Set：for-of获取值 => for (const v of set)
* 遍历Map：for-of获取键值对 => for (const [k, v] of map)
* 遍历类数组：包含length的对象、Arguments对象、NodeList对象(无Iterator接口的类数组可用Array.from()转换)
* 计算生成数据结构：Array、Set、Map
  + **keys()**：返回遍历器对象，遍历所有的键
  + **values()**：返回遍历器对象，遍历所有的值
  + **entries()**：返回遍历器对象，遍历所有的键值对
* 与for-in区别
  + 有着同for-in一样的简洁语法，但没有for-in那些缺点、
  + 不同于forEach()，它可与break、continue和return配合使用
  + 提供遍历所有数据结构的统一操作接口

应用场景

* 改写具有Iterator接口的数据结构的Symbol.iterator
* 解构赋值：对Set进行结构
* 扩展运算符：将部署Iterator接口的数据结构转为数组
* yield\*：yield\*后跟一个可遍历的数据结构，会调用其遍历器接口
* 接受数组作为参数的函数：for-of、Array.from()、new Set()、new WeakSet()、new Map()、new WeakMap()、Promise.all()、Promise.race()

### Promise

* 定义：包含异步操作结果的对象
* 状态
  + **进行中**：pending
  + **已成功**：resolved
  + **已失败**：rejected
* 特点
  + 对象的状态不受外界影响
  + 一旦状态改变就不会再变，任何时候都可得到这个结果
* 声明：new Promise((resolve, reject) => {})
* 出参
  + **resolve**：将状态从未完成变为成功，在异步操作成功时调用，并将异步操作的结果作为参数传递出去
  + **reject**：将状态从未完成变为失败，在异步操作失败时调用，并将异步操作的错误作为参数传递出去
* 方法
  + **then()**：分别指定resolved状态和rejected状态的回调函数
    - **第一参数**：状态变为resolved时调用
    - **第二参数**：状态变为rejected时调用(可选)
  + **catch()**：指定发生错误时的回调函数
  + **Promise.all()**：将多个实例包装成一个新实例，返回全部实例状态变更后的结果数组(齐变更再返回)
    - 入参：具有Iterator接口的数据结构
    - 成功：只有全部实例状态变成resolved，最终状态才会变成resolved
    - 失败：其中一个实例状态变成rejected，最终状态就会变成rejected
  + **Promise.race()**：将多个实例包装成一个新实例，返回全部实例状态优先变更后的结果(先变更先返回)
  + **Promise.resolve()**：将对象转为Promise对象(等价于new Promise(resolve => resolve()))
    - **Promise实例**：原封不动地返回入参
    - **Thenable对象**：将此对象转为Promise对象并返回(Thenable为包含then()的对象，执行then()相当于执行此对象的then())
    - **不具有then()的对象**：将此对象转为Promise对象并返回，状态为resolved
    - **不带参数**：返回Promise对象，状态为resolved
  + **Promise.reject()**：将对象转为状态为rejected的Promise对象(等价于new Promise((resolve, reject) => reject()))

应用场景

* 加载图片
* AJAX转Promise对象

重点难点

* 只有异步操作的结果可决定当前状态是哪一种，其他操作都无法改变这个状态
* 状态改变只有两种可能：从pending变为resolved、从pending变为rejected
* 一旦新建Promise对象就会立即执行，无法中途取消
* 不设置回调函数，内部抛错不会反应到外部
* 当处于pending时，无法得知目前进展到哪一个阶段
* 实例状态变为resolved或rejected时，会触发then()绑定的回调函数
* resolve()和reject()的执行总是晚于本轮循环的同步任务
* then()返回新实例，其后可再调用另一个then()
* then()运行中抛出错误会被catch()捕获
* reject()的作用等同于抛出错误
* 实例状态已变成resolved时，再抛出错误是无效的，不会被捕获，等于没有抛出
* 实例状态的错误具有冒泡性质，会一直向后传递直到被捕获为止，错误总是会被下一个catch()捕获
* 不要在then()里定义rejected状态的回调函数(不使用其第二参数)
* 建议使用catch()捕获错误，不要使用then()第二个参数捕获
* 没有使用catch()捕获错误，实例抛错不会传递到外层代码，即不会有任何反应
* 作为参数的实例定义了catch()，一旦被rejected并不会触发Promise.all()的catch()
* Promise.reject()的参数会原封不动地作为rejected的理由，变成后续方法的参数

### Generator

* 定义：封装多个内部状态的异步编程解决方案
* 形式：调用Generator函数(该函数不执行)返回指向内部状态的指针对象(不是运行结果)
* 声明：function\* Func() {}
* 方法
  + **next()**：使指针移向下一个状态，返回{ done, value }(入参会被当作上一个yield命令表达式的返回值)
  + **return()**：返回指定值且终结遍历Generator函数，返回{ done: true, value: 入参 }
  + **throw()**：在Generator函数体外抛出错误，在Generator函数体内捕获错误，返回自定义的new Errow()
* yield命令：声明内部状态的值(return声明结束返回的值)
  + 遇到yield命令就暂停执行后面的操作，并将其后表达式的值作为返回对象的value
  + 下次调用next()时，再继续往下执行直到遇到下一个yield命令
  + 没有再遇到yield命令就一直运行到Generator函数结束，直到遇到return语句为止并将其后表达式的值作为返回对象的value
  + Generator函数没有return语句则返回对象的value为undefined
* yield\*命令：在一个Generator函数里执行另一个Generator函数(后随具有Iterator接口的数据结构)
* 遍历：通过for-of自动调用next()
* 作为对象属性
  + 全写：const obj = { method: function\*() {} }
  + 简写：const obj = { \* method() {} }
* 上下文：执行产生的上下文环境一旦遇到yield命令就会暂时退出堆栈(但并不消失)，所有变量和对象会冻结在当前状态，等到对它执行next()时，这个上下文环境又会重新加入调用栈，冻结的变量和对象恢复执行

方法异同

* 相同点：next()、throw()、return()本质上是同一件事，作用都是让函数恢复执行且使用不同的语句替换yield命令
* 不同点
  + **next()**：将yield命令替换成一个值
  + **return()**：将yield命令替换成一个return语句
  + **throw()**：将yield命令替换成一个throw语句

应用场景

* 异步操作同步化表达
* 控制流管理
* 为对象部署Iterator接口：把Generator函数赋值给对象的Symbol.iterator，从而使该对象具有Iterator接口
* 作为具有Iterator接口的数据结构

重点难点

* 每次调用next()，指针就从函数头部或上次停下的位置开始执行，直到遇到下一个yield命令或return语句为止
* 函数内部可不用yield命令，但会变成单纯的暂缓执行函数(还是需要next()触发)
* yield命令是暂停执行的标记，next()是恢复执行的操作
* yield命令用在另一个表达式中必须放在圆括号里
* yield命令用作函数参数或放在赋值表达式的右边，可不加圆括号
* yield命令本身没有返回值，可认为是返回undefined
* yield命令表达式为惰性求值，等next()执行到此才求值
* 函数调用后生成遍历器对象，此对象的Symbol.iterator是此对象本身
* 在函数运行的不同阶段，通过next()从外部向内部注入不同的值，从而调整函数行为
* 首个next()用来启动遍历器对象，后续才可传递参数
* 想首次调用next()时就能输入值，可在函数外面再包一层
* 一旦next()返回对象的done为true，for-of遍历会中止且不包含该返回对象
* 函数内部部署try-finally且正在执行try，那么return()会导致立刻进入finally，执行完finally以后整个函数才会结束
* 函数内部没有部署try-catch，throw()抛错将被外部try-catch捕获
* throw()抛错要被内部捕获，前提是必须至少执行过一次next()
* throw()被捕获以后，会附带执行下一条yield命令
* 函数还未开始执行，这时throw()抛错只可能抛出在函数外部

首次next()可传值

**function** **Wrapper**(func) {

**return** **function**(...args) {

**const** generator = func(...args);

generator.next();

**return** generator;

}

}

**const** print = Wrapper(**function**\*() {

console.log(`First Input: ${**yield**}`);

**return** "done";

});

print().next("hello");

### 声明

* **共享内存和原子操作**：由全局对象SharedArrayBuffer和Atomics实现，将数据存储在一块共享内存空间中，这些数据可在JS主线程和web-worker线程之间共享

### 字符串扩展

* **padStart()**：把指定字符串填充到字符串头部，返回新字符串
* **padEnd()**：把指定字符串填充到字符串尾部，返回新字符串

### 对象扩展

* **Object.getOwnPropertyDescriptors()**：返回对象所有自身属性(非继承属性)的描述对象
* **Object.values()**：返回以值组成的数组
* **Object.entries()**：返回以键和值组成的数组

### 函数扩展

* **函数参数尾逗号**：允许函数最后一个参数有尾逗号

### Async

* 定义：使异步函数以同步函数的形式书写(Generator函数语法糖)
* 原理：将Generator函数和自动执行器spawn包装在一个函数里
* 形式：将Generator函数的\*替换成async，将yield替换成await
* 声明
  + 具名函数：async function Func() {}
  + 函数表达式：const func = async function() {}
  + 箭头函数：const func = async() => {}
  + 对象方法：const obj = { async func() {} }
  + 类方法：class Cla { async Func() {} }
* await命令：等待当前Promise对象状态变更完毕
  + 正常情况：后面是Promise对象则返回其结果，否则返回对应的值
  + 后随Thenable对象：将其等同于Promise对象返回其结果
* 错误处理：将await命令Promise对象放到try-catch中(可放多个)

Async对Generator改进

* 内置执行器
* 更好的语义
* 更广的适用性
* 返回值是Promise对象

应用场景

* 按顺序完成异步操作

重点难点

* Async函数返回Promise对象，可使用then()添加回调函数
* 内部return返回值会成为后续then()的出参
* 内部抛出错误会导致返回的Promise对象变为rejected状态，被catch()接收到
* 返回的Promise对象必须等到内部所有await命令Promise对象执行完才会发生状态改变，除非遇到return语句或抛出错误
* 任何一个await命令Promise对象变为rejected状态，整个Async函数都会中断执行
* 希望即使前一个异步操作失败也不要中断后面的异步操作
  + 将await命令Promise对象放到try-catch中
  + await命令Promise对象跟一个catch()
* await命令Promise对象可能变为rejected状态，最好把其放到try-catch中
* 多个await命令Promise对象若不存在继发关系，最好让它们同时触发
* await命令只能用在Async函数之中，否则会报错
* 数组使用forEach()执行async/await会失效，可使用for-of和Promise.all()代替
* 可保留运行堆栈，函数上下文随着Async函数的执行而存在，执行完成就消失

### 字符串扩展

* **放松对标签模板里字符串转义的限制**：遇到不合法的字符串转义返回undefined，并且从raw上可获取原字符串

### 对象扩展

* **扩展运算符(...)**：转换对象为用逗号分隔的参数序列({ ...obj }，相当于rest/spread参数的逆运算)

扩展应用

* 克隆对象：const obj = { \_\_proto\_\_: Object.getPrototypeOf(obj1), ...obj1 }
* 合并对象：const obj = { ...obj1, ...obj2 }
* 转换字符串为对象：{ ..."hello" }
* 转换数组为对象：{ ...[1, 2] }
* 与对象解构赋值结合：const { x, ...rest/spread } = { x: 1, y: 2, z: 3 }(不能复制继承自原型对象的属性)
* 修改现有对象部分属性：const obj = { x: 1, ...{ x: 2 } }

### 正则扩展

* **s修饰符**：dotAll模式修饰符，使.匹配任意单个字符(dotAll模式)
* **dotAll**：是否设置s修饰符
* **后行断言**：x只有在y后才匹配
* **后行否定断言**：x只有不在y后才匹配
* **Unicode属性转义**：匹配符合Unicode某种属性的所有字符
  + 正向匹配：\p{PropRule}
  + 反向匹配：\P{PropRule}
  + 限制：\p{...}和\P{...}只对Unicode字符有效，使用时需加上u修饰符
* **具名组匹配**：为每组匹配指定名字(?<GroupName>)
  + 形式：str.exec().groups.GroupName
  + 解构赋值替换
    - 声明：const time = "2017-09-11"、const regexp = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u
    - 匹配：time.replace(regexp, "$<day>/$<month>/$<year>")

### Promise

* **finally()**：指定不管最后状态如何都会执行的回调函数

### Async

* **异步迭代器(for-wait-of)**：，循环等待每个Promise对象变为resolved状态才进入下一步

### 字符串扩展

* **直接输入U+2028和U+2029**：字符串可直接输入行分隔符和段分隔符
* **JSON.stringify()改造**：可返回不符合UTF-8标准的字符串
* **trimStart()**：消除字符串头部空格，返回新字符串
* **trimEnd()**：消除字符串尾部空格，返回新字符串

### 对象扩展

* **Object.fromEntries()**：返回以键和值组成的对象(Object.entries()的逆操作)

### 数组扩展

* **flat()**：扁平化数组，返回新数组
* **flatMap()**：映射且扁平化数组，返回新数组(只能展开一层数组)

### 函数扩展

* **toString()改造**：返回函数原始代码(与编码一致)
* **catch()参数可省略**：catch()中的参数可省略

### Symbol

* **description**：返回Symbol值的描述

### 声明

* **globalThis对象**：作为顶层对象，指向全局环境下的this
* **do表达式**：封装块级作用域的操作，返回内部最后执行表达式的值(do{})
* **throw表达式**：直接使用throw new Error()，无需()或{}包括
* **!#命令**：指定脚本执行器(写在文件首行)

### 数值扩展

* **数值分隔符(\_)**：使用\_作为千分位分隔符(增加数值的可读性)
* **BigInt()**：创建任何位数的整数(新增的数据类型，使用n结尾)

### 对象扩展

* **链判断操作符(?.)**：是否存在对象属性(不存在返回undefined且不再往下执行)
* **空判断操作符(??)**：是否值为undefined或null，是则使用默认值

### 函数扩展

* **函数部分执行**：复用函数功能(?表示单个参数占位符，...表示多个参数占位符)
* **管道操作符(|>)**：把左边表达式的值传入右边的函数进行求值(f(x) => x |> f)
* **绑定运算符(::)**：函数绑定(左边是对象右边是函数，取代bind、apply、call调用)
  + bind：bar.bind(foo) => foo::bar
  + apply：bar.apply(foo, arguments) => foo::bar(...arguments)

### Proxy

* **Promise.try()**：不想区分是否同步异步函数，包装函数为实例，使用then()指定下一步流程，使用catch()捕获错误

### Realm

* 定义：提供沙箱功能，允许隔离代码，防止被隔离的代码拿到全局对象
* 声明：new Realm().global

### Class

* **静态属性**：使用static定义属性，该属性不会被实例继承，只能通过类来调用
* **私有属性**：使用#定义属性，该属性只能在类内部访问
* **私有方法**：使用#定义方法，该方法只能在类内部访问
* **装饰器**：使用@注释或修改类和类方法

### Module

* **import()**：动态导入(返回Promise)
  + 背景：import命令被JS引擎静态分析，先于模块内的其他语句执行，无法取代require()的动态加载功能，提案建议引入import()来代替require()
  + 位置：可在任何地方使用
  + 区别：require()是**同步加载**，import()是**异步加载**
  + 场景：按需加载、条件加载、模块路径动态化
* **import.meta**：返回脚本元信息

### Async

* **顶层Await**：允许在模块的顶层独立使用await命令(借用await解决模块异步加载的问题)