ES6

- 1 简介
- 2 let const 命令
  - let 命令
    - 基本用法
      - let 声明的变量只在其所在代码块内有效
      - 变量应该在声明语句之后才可以使用
    - 不存在变量提升
      - 在声明之前就使用这些变量 就会报错
    - 暂时性死区
      - “暂时性死区”意味着 typeof 不再是个百分之百安全的操作
    - 不允许重复声明
  - 块级作用域
    - 为什么需要块级作用域
      - 内层变量可能会覆盖外层变量
      - 用来计数的循环变量泄露为全局变量
    - ES6 的块级作用域
      - 内部作用域可以定义外层作用域的同名变量
    - 块级作用域与函数声明
      - 为了在块级作用域内声明函数
      - 函数声明类似于 var，即会提升到全局作用域或函数作用域的头部。
      - 函数声明还会提升到所在的块级作用域的头部
    - do 表达式
      - 使得块级作用域可以发为表达式，即可以返回值
  - const 命令
    - 基本用法
      - 声明只读的常量。一旦声明，常量的值就不能改变
    - 本质
      - 声明常量，就必须立即初始化
      - 保证的不是变量的值不得改动，而是变量指向的那个内存地址不得改动。
      - 对象冻结后变动，所以原则可以为其添加新属性。
      - 如果真的想将对象冻结，应该使用 Object.freeze 方法
    - ES6 声明变量的六种方法
      - var 命令和 function 命令
      - let和const 命令
      - 使用 import 命令和 class命令
  - 顶层对象的属性
    - global 对象
      - ES5 的顶层对象
        - 浏览器环境中指的是 window 对象，在 Node 环境中指的是 global 对象
        - var 命令和 function 命令声明的全局变量依旧是顶层对象的属性
        - let 命令、const 命令、class 命令声明的全局变量不属于顶层对象的属性
        - 在浏览器中，顶层对象是 window，但 Node 和 Web Worker 没有 window
        - 在浏览器和 Web Worker 中，self 也指向顶层对象，但是 Node 没有 self
        - Node 中，顶层对象是 global，但其他环境都不支持。
      - 获取顶层对象
- 3 变量的解构赋值
  - 数组的解构赋值
    - 基本用法
      - let [a, b, c] =[1, 2, 3];
    - 默认值
      - let [foo = true]=[];
    - 对象的解构赋值
      - 变量必须与属性同名才能取到正确的值
    - 字符串的解构赋值
      - const [a , b , c , d , e] = 'hello ';
    - 数值和布尔值的解构赋值
      - 如果等号右边是数值和布尔值，则会先转为对象
      - 只要等号右边的值不是对象或数组，就先将其转为对象
      - undefined null 无法转为对象，所以对它们进行解构赋值都会报错
    - 函数参数的解构赋值
  - 圆括号问题
    - 不能使用圆括号的情况
      - 变量声明语句
        - let [x] = [1];
        - let { x = { y p } } = { x: { y: 2 } };
        - var
      - 函数参数
        - function f ([[x]]) { return x ; }
        - function f([x , [x]]) {return x ;}
      - 赋值语句的模式
        - ({p : p }) = {p: 42 };
        - ([ ]) = [5]
        - [(b)] = [2];
        - ({p: [d]}) = {p: [42]};
      - 赋值语句的非模式部分
        - 可以使用圆括号的情况
          - [x, y] = [y , x];
          - 交换变量的值
  - 用途
    - 从函数返回多个值
    - 函数参数的定义
    - 提取 JSON 数据
    - 函数参数的默认值
    - 遍历 Map 结构
    - 输入模块的指定方法
    - const { SourceMapConsumer, SourceNode } = require ("source-map");
- 4 字符串的扩展
  - 字符的 Unicode 表示法
    - codePoint(x)
      - 能够正确处理 4 个字节储存的字符
    - String.fromCodePoint()
      - 可以识别大于 0xFFFF 的字符
    - 字符串的遍历接口 for ... of 循环遍历
    - 字符串的遍历器接口
      - 能识别 4 个字节的字符
    - 直接输入 4 个字节的字符
    - 码点表示法
    - 可以正确处理大于 0xFFFF 的字符
    - 字符串的遍历器接口
    - 大于 0xFFFF 的字符
    - at 方法
      - 返回正确的码点值
      - 大于 0xFFFF 的字符
      - 0x FF FF 的字符
    - 字符串的遍历器接口
    - 合成字母
    - normalize()
  - includes()、startsWith()、endsWith()
    - includes()：表示是否找到了参数字符串
    - startsWith()：表示参数字符串是否在原字符串的头部
    - endsWith()：表示参数字符串是否在原字符串的尾部
    - 支持第二个参数 表示开始搜索的位置
    - endsWith 只对前 n 个字符
  - repeat()
    - 返回一个新字符串，表示将原字符串重复 n 次。
    - 'x ' .repeat (3) // 'xxx '
    - 'hello ' .repeat(2) // 'hellohello'
    - 'na ' .repeat(0) // ''
    - 'na ' .repeat(2.9) // 'nana '
    - 参数如果是小数，会被取整
    - 参数是负数或者 Infinity，会报错
  - padStart()、padEnd()
    - 'x ' .padStart(5,' ab ') // 'abxbx '
    - 'x ' .padStart(5,'ab ') // 'xabab '
    - 'x ' .padEnd(5,'ab ') // 'xabab '
    - 'x ' .padEnd(5,' ab ') // 'xbxbx '
    - 第一个参数是小长度，第二个参数是用来补全的字符串
    - 省略第二个参数，会用空格来补全
    - 用途
      - 补全指定位数
      - 提示字符串格式
  - 模板字符串
    - 简介
    - String.raw()
      - 模板字符串的用途
      - String.raw ({ raw:'test ' }, 0, 1, 2); // 't0e1s2t '
    - 模板字符串的限制
      - 模板字符串默认会将字符串转义，导致无法嵌入其他语言
- 5 正则的扩展
  - RegExp 构造函数
    - var regex = new RegExp ('xyz ', 'i ');
    - var regex = New RegExp (/xyz/i);
  - 字符串的正则方法
    - match ( ) 、replace ( ) 、search ( ) 和split ( ) 。
  - u 修饰符
    - 可以正确处理大于字符的 UTF-16 编码。
    - 匹配的语义将会发生一个巨变开始
  - y 修饰符
    - 匹配的位置必须是剩余部分的开始
    - 表示是否设置了 修饰符
    - sticky 属性
    - 会返回正则表达式的修饰符
    - flags 属性
  - s 修饰符：dotAll 模式
    - 会返回匹配结果的开始位置
    - 在匹配结果上调用 exec 方法
    - 匹配的数组
    - s/g 和 y 都是全局匹配
  - Unicode 属性类
    - 简介
    - 其他匹配部分
    - 匹配模式修饰符
    - 引用
- 6 数值的扩展
  - 二进制和八进制表示法
  - Number.isFinite()、Number.isNaN()
    - 区别于全局的 isFinite() 和 isNaN() 方法，它们首先将参数强制转换为数值，而这两个新方法对非数值一律返回 false。
  - Number.parseInt()、Number.parseFloat()
    - 将全局方法 parseInt() 和 parseFloat() 移植到 Number 对象上面，这样做是为了逐步减少全局性方法，使得语言逐步模块化的计划。
  - Number.isInteger()
    - 用于判断一个数值是否是一个整数。
    - 注意，在 JavaScript 内部，整数和浮点数是同样的储存方法，所以 25 和 25.0 被视为同样的值。
    - 如果参数不是数值，Number.isInteger() 一律返回 false。
    - 如果数据精度太高，+1 或 -1 就会失去精度。这时候，Number.isInteger() 对于数值的判断可能会不精确，这是必须加以防范的。
  - Number.EPSILON
    - 表示 1 与大于 1 的最小浮点数之间的差。对于 64 位浮点数来说，大于 1 的最小浮点数相当于二进制的 1.00..001，小于 1 的最小浮点数相当于二进制的 0.11..111，二者的差是 2 ^ -52，这个值很小，会受到浮点数的精度的影响。
    - 引入 EPSILON 的目的，在于为浮点数计算，设置一个误差可以忍受的范围。比如，5.551115123167 - 5.551115123165 的结果是 0.000000000002，按照 EPSILON 的值，小于 2 ^ -52 的结果，就可以认为是 0。
    - Number.EPSILON * Math.pow(2, 52) // 2.2204460492503130808472633361816E-16
    - 2 ^ -52 * 2 ^ 53 // 2
    - 0.1 + 0.2 - 0.3 // 5.551115123167 e-17
    - 5.551115123167 e-17 < 2 ^ -52 // true
    - 5.551115123167 e-17 < Number.EPSILON // true
  - Number.isSafeInteger()
    - 安全整数和 Number.isInteger()
    - Math.trunc()
      - 用于去除一个数的小数部分，返回整数部分。
      - 对于空值和无法截取整数的值，返回 NaN。
      - 内部使用 Number 方法将其转为数值
    - Math.sign()
      - 判断一个数到底是正数、负数、还是零
      - 参数为正数，返回 +1
      - 参数为负数，返回 -1
      - 参数为 0，返回 0。
      - 参数为 -0，返回 -0。
      - 其他值，返回 NaN
    - Math.cbrt()
      - 用于计算一个数的立方根（开方运算）
      - 对非数值，会先将其转换为数值。
      - 等价于 Math.pow(x, 1/3)。
    - Math.clz32()
      - 返回一个数的 32 位无符号整数形式有多少个前导 0。
      - 对于小数，只有整数部分会发生作用。
    - Math.imul()
      - 返回两个数以 32 位带符号整数形式相乘的结果，返回的也是一个 32 位的带符号的整数。
      - 主要用于一些高性能的数学计算，比如底层算法和硬件检测的算法。
    - Math.fround()
      - 返回一个数的单精度浮点数形式。
      - 对于 NaN，返回 NaN。
    - Math.hypot()
      - 返回所有参数的平方和的平方根。
      - 如果参数不是数值，则会先转为数值。这等价于 Math.sqrt(x * x + y * y + z * z)。
    - Math.expm1()
      - 返回 e^x - 1，即 Math.exp(x) - 1。
    - Math.log1p()
      - 返回 1+x，即 Math.log(1 + x)。如果 x 小于 -1，则返回 NaN。
    - Math.log10()
      - 返回以 10 为底的 x 的对数。如果 x 小于 0，则返回 NaN。
    - Math.log2()
      - 返回以 2 为底的 x 的对数。如果 x 小于 0，则返回 NaN。
    - 对数方法
    - Math.sinh(x)
    - Math.cosh(x)
    - Math.tanh(x)
    - Math.asinh(x)
    - Math.acosh(x)
    - Math.atanh(x)
    - 双曲函数方法
    - Math.signbit()
      - 来判断一个数的正负，包括某些参数是 -0、它使会返回 -0。
    - 指数运算符
      - 新增了一个指数运算符**。
      - 2 ** 3 //8
    - Integer 数据类型
      - 创建 Integer 数据类型
      - 将现有数据类型与 Number（ ）一致
- 7 函数的扩展
  - 函数参数的默认值
    - 基本用法
      - 与解构赋值默认值的结合使用
    - 参数默认值的位置
      - 通常情况下，定义了默认值的参数，应该是尾参数。因为这样比较容易看出来，到底省略了哪些参数。如果非尾部的参数设置默认值，比较容易看出来，到底省略了哪些参数。
    - 函数的 length 属性
      - 指定了默认值后，函数的 length 属性将返回没有指定默认值的参数个数。也就是说，指定了默认值后的参数不计入 length 属性。
      - 如果设置了默认值的参数是尾参数，则 length 属性会忽略后面的参数。
      - 这样一来，从语义上来说，length 属性的返回值，就是期望传入的参数个数。
    - 作用域
      - 一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域。
      - 等初始化结束，这个作用域就消失了。这种语法行为的好处是，函数的默认值表达式，不会受到函数内部的变量的影响。
    - rest 参数
      - 用于获取函数的多余的参数，这样就不需要使用 arguments 对象了。
      - 形式为“...变量名”，由于 rest 参数是一个真正的数组，数组特有的方法都可以用于这个参数。而 arguments 对象则不是。
    - 严格模式
      - 只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部的严格模式，就不能显式指定了。
    - name 属性
      - 函数的 name 属性返回该函数的函数名。
      - 对匿名函数赋值的变量，ES6 会写出变量的名字。
      - 对具名函数赋值的变量，ES6 会写出函数的名字。
      - 对于 Function 构造函数赋值的函数，name 属性返回 anonymous。
      - 对于 bind 方法生成的函数，name 属性返回 bound 加上原函数的名字。
      - 对于 Function 构造函数赋值的函数，name 属性返回 anonymous。
    - 箭头函数
      - 基本用法
        - var f = v => v ;
      - 注意事项
        - 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象。
        - 不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。
        - 不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
        - 不可以使用 yield 命令，因此箭头函数不能用作 generator 函数。
        - 箭头函数根本没有自己的 this，导致内部的 this 就是外层代码块的 this。正是因为它没有 this，所以也不能用作构造函数。
      - 嵌套的箭头函数
      - 箭头函数的应用
        - 绑定 this
        - 尾调用优化
        - 尾递归
        - 尾递归优化实现
        - 函数参数的尾逗号
- 8 数组的扩展
  - 扩展运算符
    - 含义
      - 如同 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列
    - 替代数组的 apply 方法
      - Math .max (... [14, 3, 77])
    - 扩展运算符的应用
      - 合并数组
        - [... arr1 , ... arr2 , ... arr3]
      - 与解构赋值结合
        - const [first, ... rest] =[1 , 2 , 3 , 4 , 5];
      - 函数的返回值
      - 字符串
        - [... 'hello '] === ['h' ,'e' ,'l' ,'l' ,'o']
      - 实现了 Iterator 接口的对象
      - Map 和Set 结构、 Generator 函数
  - Array. from()
    - 用于将两类对象转为真正 数组
    - arguments 对象
    - 伪数组
- 10 Symbol
  - 概述
    - Symbol 函数前不能使用 new 命令，否则会报错。
    - Symbol 值不是对象，所以不能添加属性
    - Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述
    - Symbol 值不能与其他数据进行运算，否则会报错
  - 作为属性的 Symbol
    - Symbol 值作为对象属性名 不能使用点运算符
    - Symbol 值作为属性名时，该属性是公 属性，不是私有属性
  - 属性名的遍历
    - Object .getOwnPropertySymbols 方法可以获取指定对象的所有 Symbol 属性名
    - Reflect.ownKeys 方法可以返回所有类型的键名，包括常规键名和Symbol 键名
  - Symbol.for()、Symbol. keyFor()
    - Symbol.for()
      - var s1 = Symbol.for ('foo ');
      - var s2 = Symbol.for ('foo ');
      - s1 === s2 // true
      - 接收有没有以该参数作为名称的 Symbol 值。如果有，就直接返回一个 Symbol 值，否则就新建并返回一个 以该字符串为名称的 Symbol 值
    - Symbol. keyFor()
      - var s1 = Symbol.for (" foo ") ;
      - SymbolsKeyFor(s1 //foo'
      - 返回一个已登记的 Symbol 类型键的 key。
    - 区别
      - Symbol.for()与 Symbol()这两种写法都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，而后者不登
      - var s2 = Symbol (" foo") ;
      - SymbolsKeyFor(s2) // undefined
      - Symbol.for 为 Symbol 值登记的名字是全局环境的，可以在不同的 iframe或serviceworker 中取到同一个值
  - 内置的 Symbol 值
    - Symbol.hasInstance
      - 判断该对象是否为某个构造函数的实例
    - Symbol.isConcatSpreadable
      - 表示该对象使用Array . prototype . concat()时是否可以展开
    - Symbol.species
    - Symbol.match
    - ...
- 11 Set Map 数据结构
  - Set
    - 基本用法
      - const s =new Set() ;
      - 还原的数据类型唯一的
      - [2 , 3 , 5 , 4 , 5 , 2 , 2] .forEach(x => s . add(x) ;
    - Set 实例的属性和方法
      - Set . prototype. constructor 构造函数，默认就是 Set 函数
      - Set . prototype . size 返回 Set 实例的成员总数
      - 操作方法
        - add (value) ：添加某个值，返回 Set 结构本身
        - delete(value) ：删除某个值，返回一个布尔值，表示删除是否成功
        - has (va lue) ：返回 个布尔值，表示参数是否为 Set 的成员
        - clear ( ) ：清除所有成员，没有返回值。
      - 遍历操作
        - keys( ) ：返回键名的遍历器。
        - values( ) ：返回键值的遍历器
        - entries ( ) ：返回键名和键值的遍历器
        - forEach( ) ：使用回调函数遍历每个成员
        - Set 结构没有键名，只有键值（或者键名和键值是同一个值），所以keys 方法和 values法的行为完全一致。
      - 应用
        - let union = new Set([... a, ... b]);
        - let intersect= new Set([... a] .filter(x <> b . has(x));
        - let difference = new Set([... a].filter(x <> !b .has(x));
        - 并集
        - 交集
        - 差集
  - WeakSet
    - 含义
      - WeakSet结构与 Set 类似，也是不重复的值的集合
    - 区别
      - WeakSet 的成员只能是对象，而不能是其他类型的值
      - WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用
      - WeakSet 不能遍历，因为成员都是弱引用，随时可能消失，遍历机制无法保证成员存在（类似forEach 方法）
    - 语法
      - const wst = new WeakSet();
      - 成为WeakSet的成员只能是普通的对象，而不是基本身
      - 方法
        - WeakSet.prototype.add(value) ：WeakSet 实例添加一个新成员
        - WeakSet.prototype.delete (value) ：删除 WeakSet 实例 指定成 员
        - WeakSet.prototype.has value) ：返回一个布尔值 表示某个值是否在 WeakSet中
    - “键”名固定不变字符串，各种类型的键（包括对象）都可以当作键
    - 如果 Map 的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，Map 就将其视为一个键，包括 +0和 - 0
    - 只有对同一个对象的引用，Map 结构才将其视为同一个键
    - Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键
    - 实例的属性和操作方法
      - size 属性
        - size 属性返回 Map 结构的成员总数
      - set(key, value)
        - set 方法设置 key 所对应的键值，然后返回整个 Map 结构
      - get(key)
        - get 方法读取 key 对应的键值，如果找不到 key，则返回 undefined
      - has(key)
        - 返回 一个布尔值，表示某个键是否在 Map 数据 结构中
      - delete(key)
        - 删除某个键，返回 true。如果删除失败，则返回 false
      - clear( )
        - clear 方法清除所有成员，没有返回值
    - 遍历方法
      - keys ( ) ：返回键名的遍历器
      - values ( ) ：返回键值的遍历器
      - entries ( ) ：返回所有成员的遍历器
      - forEach ( ) ：遍历 Map 的所有成员
      - Map 结构默认为数组结构的比较快速的方法是给它扩展运算符 (...)
    - 与其他数据结构的互相转换
      - Map 转为数组
        - 使用扩展运算符 (...)
      - 数组转为 Map
        - 将数组传入 Map 构造函数就可以转为 Map
      - Map 转为对象
        - 如果 Map 的所有键都是字符串，则可以转为对象
      - 对象转为 Map
      - Map 转为 JSON
      - JSON 转为 Map
    - WeakMap
      - 含义
        - WeakMap 只接受对象作为键名
        - WeakMap 的键名所指向的对象不计入垃圾回收机制
      - WeakMap 的语法
        - 只有 个方法可用：get ( ) 、set ( ) 、has ( ) 、delete ( )
      - WeakMap 的用途
        - 部署私有属性
- 12 Proxy
  - 概述
    - 可以理解成在目标对象前架设一个“拦截”层，外界对该对象的访问都必须先通过这层拦截
    - var proxy= new Proxy(target, handler) ;
    - 实例的方法
      - get(target, propKey, receiver)
      - set(target, propKey, value, receiver)
      - has(target, propKey)
      - deleteProperty(target, propKey)
      - ownKeys(target)
      - getOwnPropertyDescriptor(target, propKey)
      - defineProperty(target, propKey, propDesc)
      - preventExtensions(target)
      - getPrototypeOf(target)
      - isExtensible(target)
      - setPrototypeOf(target, proto)
      - apply(target, object, args)
      - construct(target, args)
  - Proxy 实例的方法
  - Proxy.revocable()
    - Proxy.revocable 方法返回 一个可取消 Proxy 实例。
    - 由于 Proxy.revocable() 是个工厂函数，每次调用都会返回一个新的 Proxy 实例，并且旧的 revoke 函数也会生成一个新的。
    - 场景
      - 目标对象不允许直接访问，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
    - this 问题
      - 在 Proxy 代理的情况下，目标对象内部的 this 关键字会指向 Proxy 代理
- 13 Reflect
  - 概述
    - 将Object 对象的 属于语言内部的方法（比如 Object.defineProperty)放到 Reflect 对象上。
    - 现阶段，某些方法同时在 Object 和 Reflect 对象上部署，未来的新方法将只部署在 Reflect 对象上。
    - 修改某些 Object 方法的返回结果，让其变得更合理。
    - Object.defineProperty (obj, name, desc) 在无法定义属性时会抛出一个错误，而Reflect. defineProperty(obj, name, desc) 则会返回 false
    - 让Object 操作都变成函数行为。
    - Reflect . has(Object,'assign ') // true
    - Reflect 对象的方法与 Proxy 对象的方法一一对应
- 14 Promise 对象
  - Promise 的含义
    - 特点
      - 对象的状态不受外界影响。
      - 一旦状态改变就不会再变
    - 缺点
      - 一旦新建它就会立即执行，无法取消
      - 如果不设置回调函数，Promise 内部抛出的错误不会反应到外部
  - 基本用法
    - Promise.prototype.then()
      - 作用是为 Promise 实例添加状态改变时的回调函数
      - 第一个参数是 Resolved 状态的回调函数
      - 第二个参数（可选）是 Rejected 状态的回调函数
    - Promise.prototype.catch()
      - 可以捕获前面 then方法的错误
      - 建议使用 catch 方法，而不使then 方法的第二个参数
      - 没有使用 catch 方法指定错误处理的回调函数，Promise 对象抛出的错误不会传递到外层代码，即不会有任何反应
      - 浏览器会默认打印出错误信息
    - Promise.all()
      - 用于将多个 Promise 实例包装成一个新的 Promise 实例
      - var p = Promise.all ([p1 , p2 , p3]) ;
      - 如果不是promise就先调用下面讲到的 Promise.resolve 方法
      - 只有p1 p2 p3状态都变成 Fulfilled 时状态才会变成 Fulfilled
      - p1 p2 p3 初始回调函数成一个数组，传递给p 的回调函数
      - 只要p1 p2 p3 中有一个 被 rejected ，p的状态就变成 Rejected
      - 如果作为参数的 Promise 实例自身定义了 catch 方法，那么它被 rejected 时并不会触发Promise.all()的 catch 方法
    - Promise.race()
      - 只要 p1 p2 p3 中有一个实例率先改变状态，p的状态就跟着改变
      - 率先改变的 Promise 实例的返回值将传递给p 的回调函数
    - Promise.resolve()
      - 将现有对象转为 Promise 对象
      - 参数是一个 Promise 实例
        - 将不做任何修改，原封不动地返回这个实例
      - 参数是一个 thenable 对象
        - 指的是具有 then 方法的对象
        - 会将这个对象转为 Promise 对象，然后立即执行 thenable 对象的then 方法。
      - 参数不是具有 then 方法的对象或根本不是对象
        - 返回一个新的 Promise 对象，状态为 Resolved。
      - 不带有任何参数
        - 直接返回 Resolved 状态的Promise 对象
        - 立即 resolve Promise 对象是在本轮 “事件循环” 结束时
    - Promise.reject()
      - 生成 Promise 对象的实例，状态为 Rejected ，回调函数会立即执行。
      - 参数会原封不动地作为 reject 回调函数或抛错方法的参数
    - 两个有用的实验方法
      - done()
        - 总是处于回调函数的末端，保证抛出任何可能出现的错误，全局抛出
      - finally()
        - 该函数不管怎样都会调执行
    - Promise.try()
      - 模拟了 try 代码块，就像promise.catch 模拟 catch 代码
- 15 Iterator和 for... of 循环
  - Iterator（ 遍历器 ）的概念
    - 为各种不同的数据结构提供统一的访问机制
    - 作用
      - 各种数据结构提供 一个统 的、简便的访问接口
      - 使数据结构的成员能按某种次序排列
      - for ... of
    - 遍历过程
      - 创建 个指针对象，指向当前数据结构的起始位置。
      - 第一次调用指针对象的 next 方法，可以将指针指向数据结构的第一个成员
      - 第二次调用指针对象的 next 方法，指针指向数据结构的第 二个成员
      - 不断调用指针对象的 next 方法，直到它指向数据结构的结束位置
    - 默认 Iterator 接口
- 18 async 函数
  - 含义
    - Generator 函数的语法糖
    - 内置执行器
    - 更好的语义
    - 更广泛的适用性
    - 返回值是 Promise
      - async 函数的返回值是 Promise 对象
      - Generator 函数的返回值是 Iterator 对象
    - 注意事项
  - 用法
    - 返回 Promise 对象
      - async 函数内部 return 语句返回的值，会成为 then 方法回调函数的参数
      - async 函数内部抛出错误会导致返回的 Promise 对象变为 reject 状态
      - 抛出的错误对象会被 catch 方法回调函数接收到
    - Promise 对象的状态变化
      - async 函数返回的 promise 对象必须等到内部所有 await 命令后面的 Promise 对象执行完才会发生状态改变
      - 除非遇到 return 语句或者抛出错误。也就是说，只有async 函数内部的异步操作执行完，才会出发async函数返回的promise的resolve方法，开始执行 then 方法指定的回调函数。
    - 语法
      - await 命令
        - await 命令后面是 promise 对象，如果不是，会被转成一个Promise 对象。
        - 只要一个 await 语句后面的 Promise 变为 reject ，那么整个 async 函数都会中断执行
      - 错误处理
        - 将第一个await 放在 try ... catch 结构里面，这样不管这个异步操作是否成功，第二个 await 都会执行
      - 注意点
        - await 命令后面的 Promise 对象会出现 resolved 或 rejected
        - 把 await 命令放在 try ... catch 代码块中
    - async函数的实现原理
      - 原理就是将 Generator 函数和自动执行器包装在 一个函数里
    - 其它 async 函数
    - 其它编程语言比较

XMind:ZEN 试用模式