

Table of Contents generated with DocToc

- 深入JavaScript
 - 深入JavaScript运行原理
 - 浏览器工作原理
 - JavaScript引擎
 - JavaScript的执行过程
 - JS执行函数的方法
 - 变量环境和记录
 - 作用域提升面试题
 - JS的内存管理和闭包
 - 认识内存管理
 - JS的内存管理
 - JS的垃圾回收
 - 常见的GC算法 – 引用计数
 - 常见的GC算法 – 标记清除
 - 闭包
 - JS中函数是一等公民
 - JS中闭包的定义
 - 闭包的访问过程
 - 闭包的执行过程
 - 闭包的内存泄露
 - AO不使用的属性
 - JS函数的this指向
 - this指向
 - 规则一：默认绑定
 - 规则二：隐式绑定
 - 规则三：显示绑定
 - call、apply、bind
 - 内置函数的绑定思考
 - 规则四：显示绑定
 - 规则优先级
 - this规则之外 – 忽略显示绑定
 - this规则之外 - 间接函数引用
 - this规则之外 – ES6箭头函数
 - 面试题
 - JS函数式编程
 - 实现apply、call、bind
 - call
 - apply

- bind
- arguments
- 箭头函数不绑定arguments，没有显式原型
- JavaScript纯函数
 - 纯函数的案例
 - 纯函数的优势
- JavaScript柯里化
 - 柯里化的结构
 - 柯里化作用
 - 自动柯里化函数
- 组合函数
- JS额外知识补充
 - with语句
 - eval函数
 - 严格模式
- JS面向对象
 - 对属性操作的控制
 - 数据属性描述符
 - 存取属性描述符
 - 同时定义多个属性
 - 对象方法补充
 - 创建多个对象的方案
 - 工厂模式
 - 构造函数
 - 对象的原型
 - 函数的原型 prototype
 - 重写原型对象
 - 面向对象的特性 – 继承
 - JavaScript原型链
 - 通过原型链实现继承
 - 借用构造函数继承
 - 原型式继承函数
 - 寄生式继承函数
 - 寄生组合式继承
 - 对象的方法补充
 - 原型继承关系
 - 类
 - 继承内置类
 - 类的混入 mixin
 - 阅读源码

- JavaScript中的多态
- ES6-ES12新增
 - ES6
 - 字面量的增强
 - 解构 Destructuring
 - let/const
 - 函数默认值
 - 函数的剩余参数
 - 展开语法
 - 数值的表示
 - Symbol
 - Set的常见方法
 - WeakSet使用
 - Map的基本使用
 - WeakMap的使用
 - ES7
 - Array Includes
 - 指数(乘方) exponentiation运算符
 - ES8
 - Object values
 - Object entries
 - String Padding
 - Trailing Commas
 - Object Descriptors
 - ES10
 - flat flatMap
 - Object fromEntries
 - trimStart trimEnd
 - ES11
 - BigInt
 - Nullish Coalescing Operator
 - Optional Chaining
 - Global This
 - for..in标准化
 - ES12
 - FinalizationRegistry
 - WeakRefs
 - logical assignment operators
- Proxy-Reflect vue2-vue3响应式原理
 - Proxy

- Proxy基本使用
- Proxy的set和get捕获器
- Proxy所有捕获器
- Proxy的construct和apply
- Reflect
 - Receiver的作用
 - Reflect的construct
- 响应式
 - 响应式函数的实现
 - 响应式依赖的收集
 - 监听对象的变化
 - 对象的依赖管理
 - 正确的依赖收集
 - 对Depend重构
 - 创建响应式对象
 - Vue2响应式原理
- Promise
 - Promise使用
 - Executor
 - resolve不同值的区别
 - then方法
 - catch方法
 - finally方法
 - Promise的类方法
 - Promise 实现
- Iterator-Generator
 - 迭代器
 - 可迭代对象
 - 原生迭代器对象
 - 可迭代对象的应用
 - 自定义类的迭代
 - 生成器
 - 生成器传递参数 – next函数
 - 生成器提前结束 – return 函数
 - 生成器抛出异常 – throw 函数
 - 生成器替代迭代器
 - 对生成器的操作
 - 异步处理方案
- await-async-事件循环
 - 异步函数 async function

- 进程和线程
 - 操作系统的工作方式
 - 浏览器中的JavaScript线程
 - 浏览器的事件循环
 - 宏任务和微任务
- Node的事件循环
 - Node事件循环的阶段
 - Node的宏任务和微任务
 - Node事件循环的顺序
- 错误处理方案
 - throw关键字
 - Error类型
 - 异常的处理
 - 异常的捕获
- 模块化
 - CommonJS规范和Node关系
 - exports导出
 - require细节
 - 模块的加载过程
 - CommonJS规范缺点
 - AMD规范
 - require.js的使用
 - CMD规范
 - SeaJS的使用
 - ES Module
 - import meta
 - ES Module的解析流程
- 包管理工具详解 npm、yarn、cnpm、npx
 - npm
 - 常见的属性
 - npm install
 - npm install 原理
 - package-lock.json
 - npm其他命令
 - yarn工具
 - cnpm工具
 - npx工具
 - 局部命令的执行
 - npm发布自己的包
- JSON-数据存储

- JSON基本语法
- JSON序列化
 - Stringify的参数
 - parse方法
 - 使用JSON序列化深拷贝
- Storage/indexedDB/cookie
 - Storage
 - localStorage和sessionStorage的区别
 - Storage常见的方法和属性
 - 封装Storage
 - IndexedDB
 - IndexedDB的连接数据库
 - IndexedDB的数据库操作
 - Cookie
 - cookie常见的属性
 - 客户端设置cookie
 - cookie 缺点
- BOM 和 DOM
 - BOM
 - Window
 - EventTarget
 - Location对象
 - history对象常见属性和方法
 - DOM和架构
 - Node节点
 - Element
- 事件监听
 - 事件流
 - 事件对象event
- 防抖和节流
 - 防抖 debounce 函数

深入JavaScript

深入JavaScript运行原理

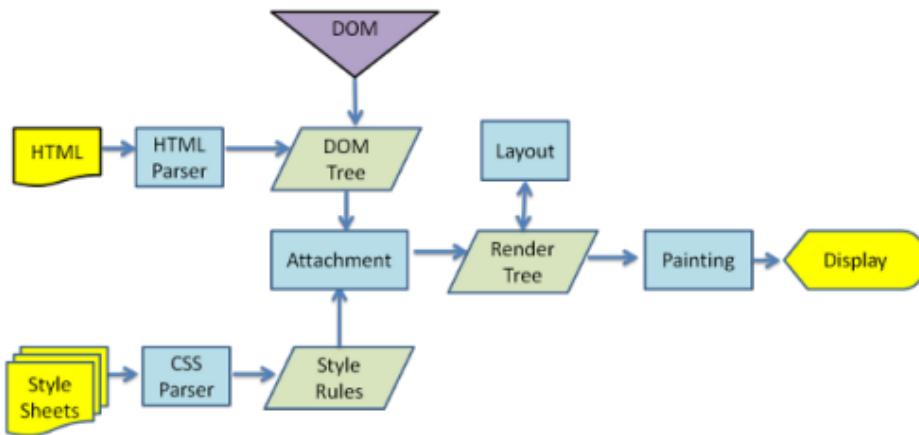
浏览器工作原理

1. 不同的浏览器有不同的内核组成

1. Gecko：早期被Netscape和Mozilla Firefox浏览器使用；
2. Trident：微软开发，被IE4~IE11浏览器使用，但是Edge浏览器已经转向Blink；
3. Webkit：苹果基于KHTML开发、开源的，用于Safari，Google Chrome之前也在使用；
4. Blink：是Webkit的一个分支，Google开发，目前应用于Google Chrome、Edge、Opera等；

2. 浏览器内核指的是浏览器的排版引擎

3. HTML解析的时候遇到了JavaScript标签，会停止解析HTML，而去加载和执行JavaScript代码



JavaScript引擎

1. 为什么需要JavaScript引擎呢？

- 我们前面说过，高级的编程语言都是需要转成最终的机器指令来执行的
- 事实上我们编写的JavaScript无论你交给浏览器或者Node执行，最后都是需要被CPU执行的
- 但是CPU只认识自己的指令集，实际上是机器语言，才能被CPU所执行
- 所以我们需要JavaScript引擎帮助我们将JavaScript代码翻译成CPU指令来执行

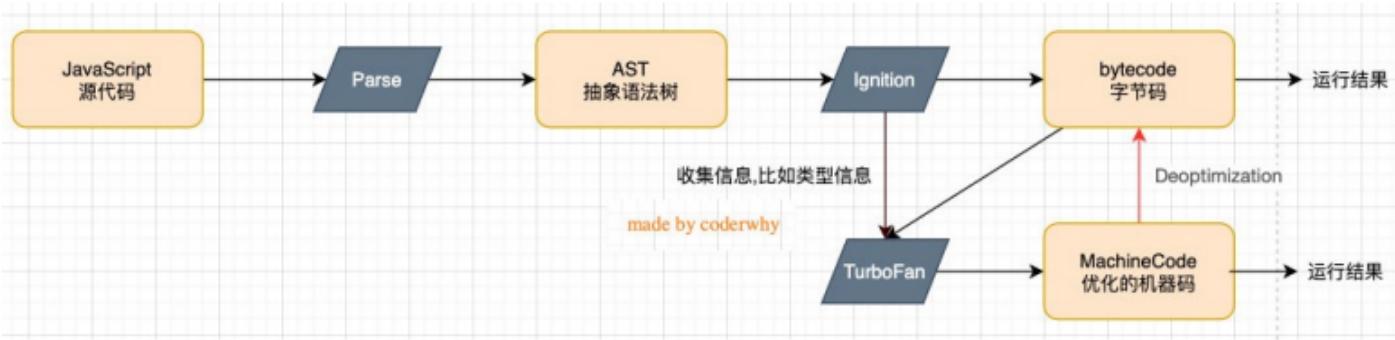
2. 浏览器内核和JS引擎的关系

这里我们先以WebKit为例，WebKit事实上由两部分组成的

- WebCore：负责HTML解析、布局、渲染等等相关的工作
- JavaScriptCore：解析、执行JavaScript代码

3. V8引擎的原理

V8是用C++编写的Google开源高性能JavaScript和WebAssembly引擎，它用于Chrome和Node.js等，可以独立运行，也可以嵌入到任何C++应用程序中。



1. Parse 模块会将JavaScript代码转换成 AST (抽象语法树) , 这是因为解释器并不直接认识JavaScript代码
 - 如果函数没有被调用, 那么是不会被转换成AST的
 - Parse 的V8官方文档: <https://v8.dev/blog/scanner>
2. Ignition 是一个解释器, 会将 AST 转换成 ByteCode (字节码)
 - 同时会收集 TurboFan 优化所需要的信息 (比如函数参数的类型信息, 有了类型才能进行真实的运算)
 - 如果函数只调用一次, Ignition 会执行解释执行 ByteCode
 - Ignition 的V8官方文档: <https://v8.dev/blog/ignition-interpreter>
3. TurboFan 是一个编译器, 可以将字节码编译为CPU可以直接执行的机器码
 - 如果一个函数被多次调用, 那么就会被标记为热点函数, 那么就会经过 TurboFan 转换成优化的机器码, 提高代码的执行性能
 - 但是, 机器码实际上也会被还原为 ByteCode , 这是因为如果后续执行函数的过程中, 类型发生了变化 (比如sum函数原来执行的是number类型, 后来执行变成了string类型) , 之前优化的机器码并不能正确的处理运算, 就会逆向的转换成字节码
 - TurboFan 的V8官方文档: <https://v8.dev/blog/turbofan-jit>
4. V8执行的细节

Blink将源码交给V8引擎, Stream获取到源码并且进行编码转换;

 - Scanner 会进行词法分析 (lexical analysis) , 词法分析会将代码转换成tokens;
 - 接下来 tokens 会被转换成AST树, 经过 Parser 和 PreParser :
 - Parser 就是直接将tokens转成AST树架构;
 - PreParser 称之为预解析, 为什么需要预解析呢?
 - 这是因为并不是所有的JavaScript代码, 在一开始时就会被执行。那么对所有的JavaScript代码进行解析, 必然会影响网页的运行效率;
 - 所以V8引擎就实现了 Lazy Parsing (延迟解析) 的方案, 它的作用是将不必要的函数进行预解析, 也就是只解析暂时需要的内容, 而对函数的全量解析是在函数被调用时才会进行;
 - 比如我们在一个函数outer内部定义了另外一个函数inner, 那么inner函数就会进行预解析;
 - 生成 AST树后, 会被 Ignition 转成 字节码 (bytecode) , 之后的过程就是代码的执行过程

JavaScript的执行过程

1. 初始化全局对象

js引擎会在执行代码之前，会在堆内存中创建一个全局对象：Global Object (GO)

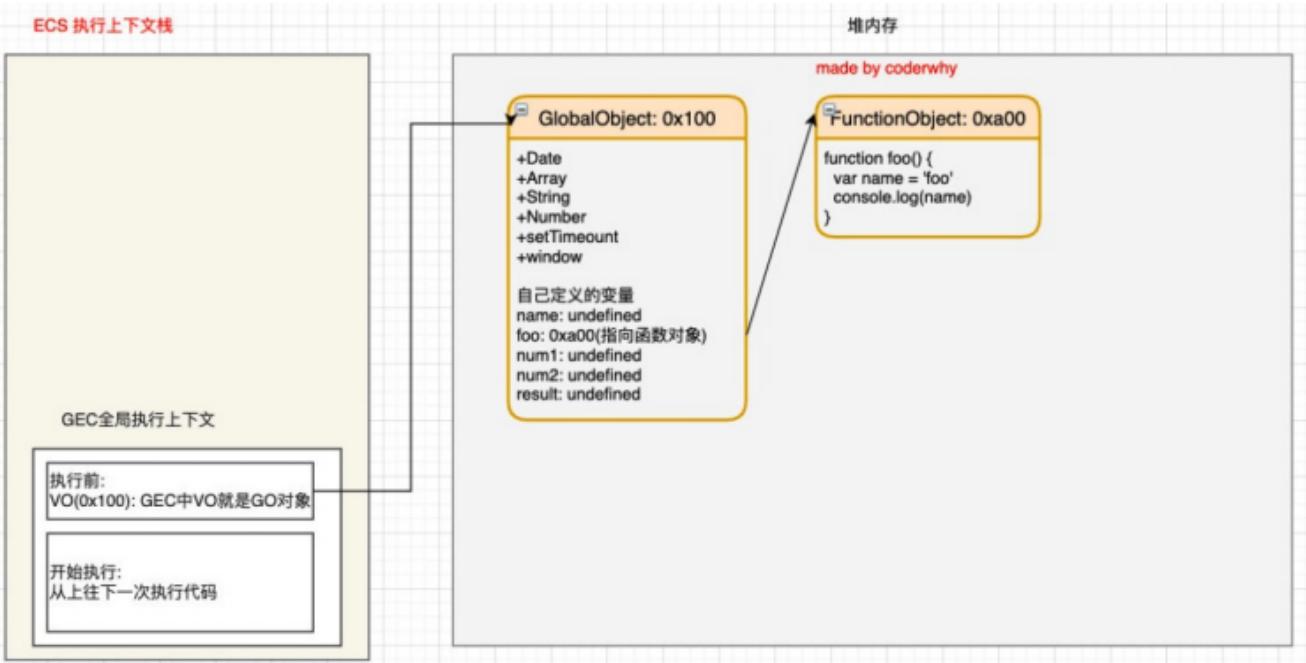
- 该对象 所有的作用域 (scope) 都可以访问；
- 里面会包含Date、Array、String、Number、setTimeout、setInterval等等；
- 其中还有一个window属性指向自己

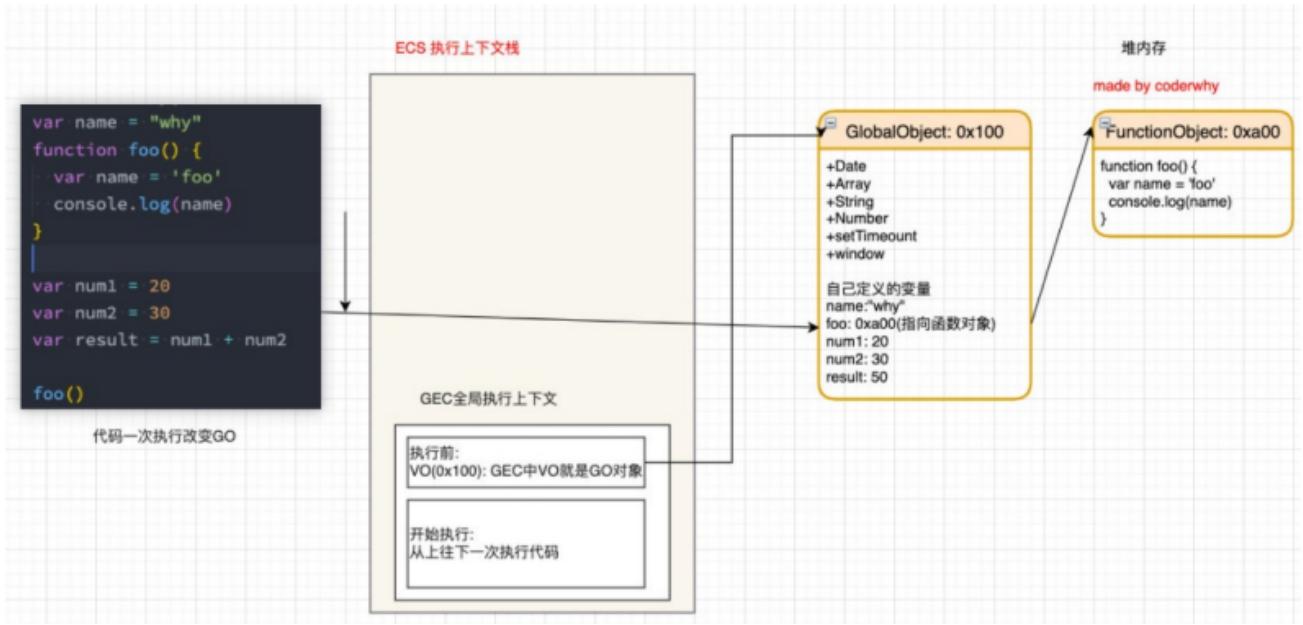


2. 执行上下文栈（调用栈）

js引擎内部有一个执行上下文栈（Execution Context Stack，简称ECS），它是用于执行代码的调用栈。那么现在它要执行谁呢？执行的是全局的代码块：

- 全局的代码块为了执行会构建一个 Global Execution Context (GEC)；
- GEC会被放入到ECS中执行；
- GEC被放入到ECS中里面包含两部分内容：
 - 第一部分：在代码执行前，在parser转成AST的过程中，会将全局定义的变量、函数等加入到GlobalObject中，但是并不会赋值；
 - 这个过程也称之为变量的作用域提升 (hoisting)
 - 第二部分：在代码执行中，对变量赋值，或者执行其他的函数



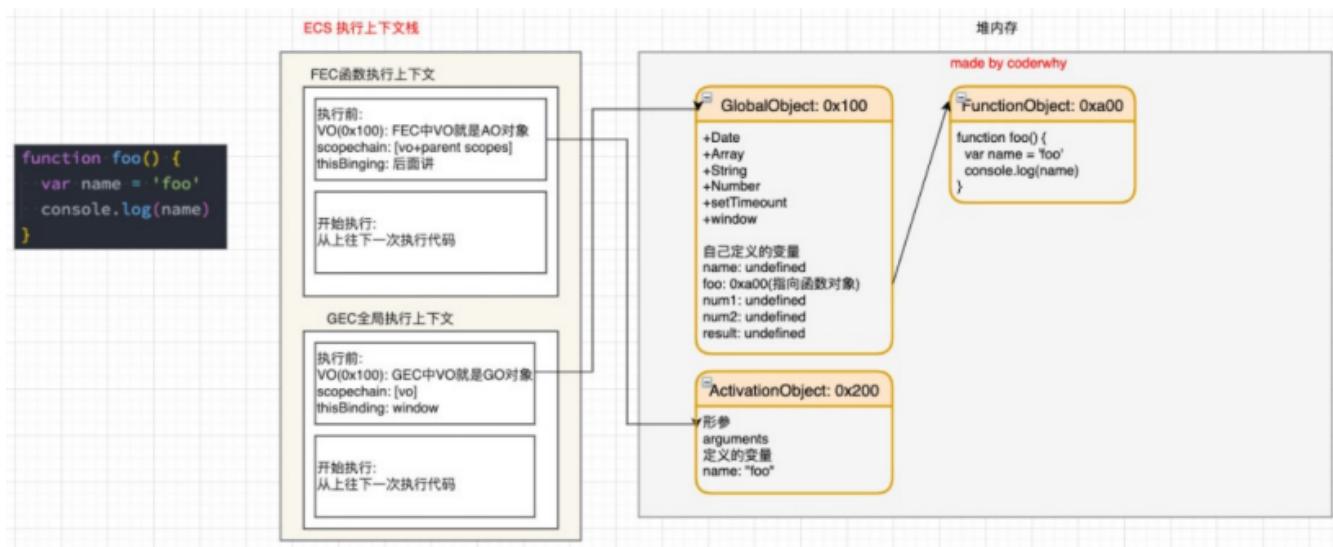
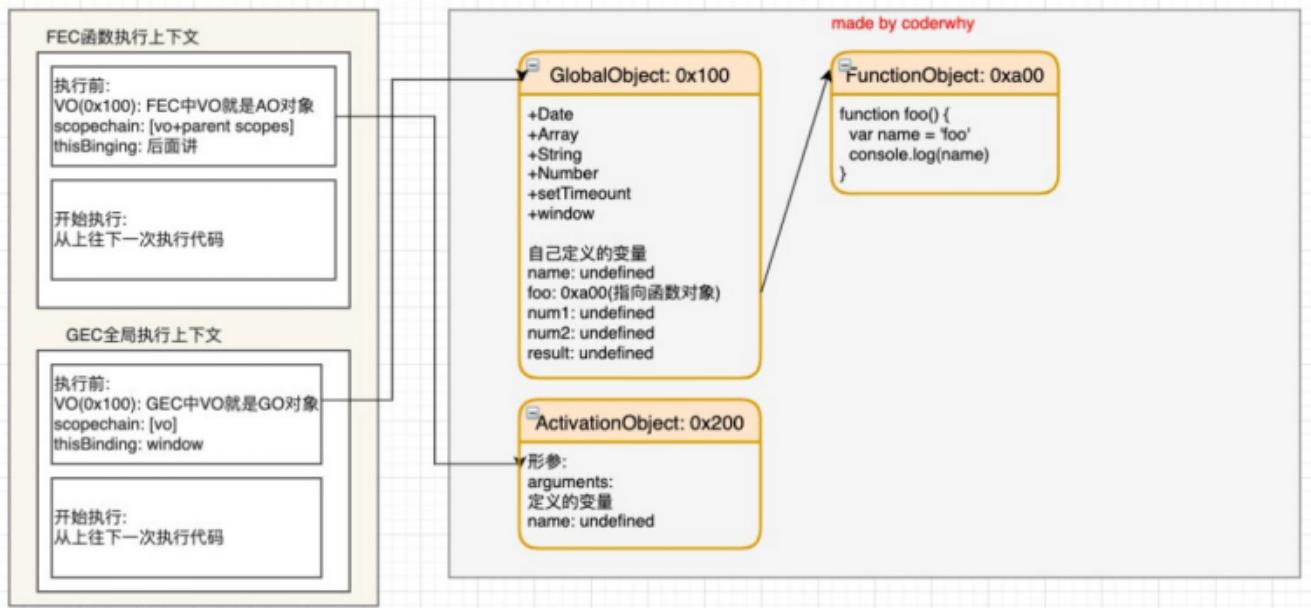


JS执行函数的方法

- 在执行的过程中执行到一个函数时，就会根据函数体创建一个函数执行上下文（Functional Execution Context，简称FEC），并且压入到 EC Stack 中。
- FEC 中包含三部分内容：
 - 第一部分：在解析函数成为 AST 树结构时，会创建一个 Activation Object (AO)
 - AO 中包含 形参、 arguments、 函数定义 和 指向函数对象、 定义的变量
 - 第二部分：作用域链：由 VO（在函数中就是AO对象） 和 父级VO 组成，查找时会一层层查找
 - 第三部分： this 绑定的值

Functional Execution Context
+ VO: 形参/arguments/function/变量
+ Scope Chain: VO/Parent VO
+ thisValue: 根据不同情况绑定this

ECS 执行上下文栈



变量环境和记录

1. 上面都是基于早期ECMA的版本规范：

Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function code, parameters are added as properties of the variable object.

每一个执行上下文会被关联到一个变量环境 (variable object, VO)，在源代码中的变量和函数声明会被作为属性添加到VO中。

对于函数来说，参数也会被添加到VO中。

2. 在最新的ECMA的版本规范中，对于一些词汇进行了修改

Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.

每一个执行上下文会关联到一个变量环境 (VariableEnvironment) 中，在执行代码中变量和函数的声明会作为环境记录 (Environment Record) 添加到变量环境中。

对于函数来说，参数也会被作为环境记录添加到变量环境中。

- 在最新的ECMA标准中，变量对象VO已经有另外一个称呼了变量环境VE

作用域提升面试题

```

var n = 100
function foo() {
  n = 200
}
console.log(n) // 100

function foo() {
  console.log(n) // undefined
  var n = 200
  console.log(n) // 200
}
var n = 100
foo()

var n = 100
// 函数作用域链取决于定义的位置，而不是调用的位置
function foo1() {
  console.log(n) // 100
}
function foo2() {
  var n = 200
  console.log(n) // 200
  foo1()
}
foo2()
console.log(n) // 100

var a = 100
function foo() {
  console.log(a) // undefined
  return // 在 return 前 变量已被定义
  var a = 100
}
foo()

function foo() {
  var a = b = 100 // var a = 100; b = 100
}
foo()
// 全局作用域中找不到 a
console.log(a) // a is not defined
console.log(b) // 100

```

JS的内存管理和闭包

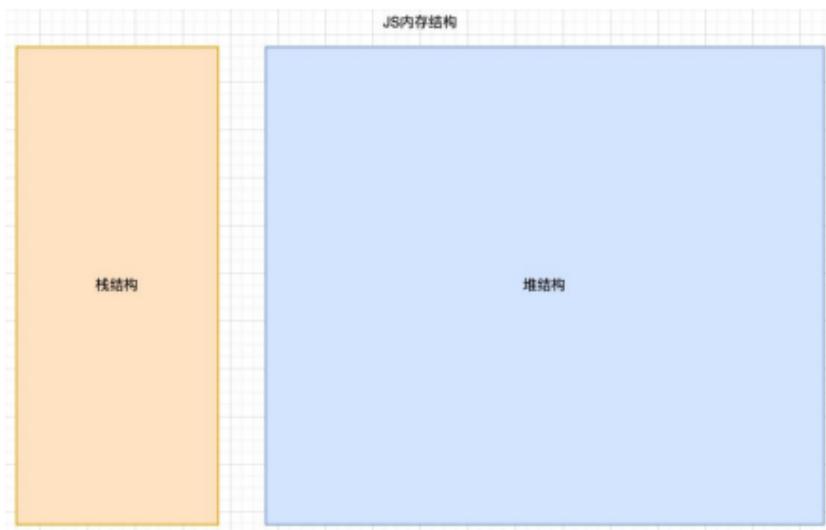
认识内存管理

- 不管什么样的编程语言，在代码的执行过程中都是需要给它分配内存的，不同的是某些编程语言需要我们自己手动的管理内存，某些编程语言会可以自动帮助我们管理内存：
- 不管以什么样的方式来管理内存，内存的管理都会有如下的生命周期：
 - 第一步：分配申请你需要的内存（申请）；
 - 第二步：使用分配的内存（存放一些东西，比如对象等）；
 - 第三步：不需要使用时，对其进行释放；
- 不同的编程语言对于第一步和第三步会有不同的实现：
 - 手动管理内存：比如C、C++，包括早期的OC，都是需要手动来管理内存的申请和释放的（`malloc` 和 `free` 函数）
 - 自动管理内存：比如Java、JavaScript、Python、Swift、Dart等，它们有自动帮助我们管理内存；
- JavaScript通常情况下是不需要手动来管理的

JS的内存管理

JavaScript会在定义变量时为我们分配内存。

- JS对于 基本数据类型 内存的分配会在执行时，直接在栈空间进行分配；
- JS对于 复杂数据类型 内存的分配会在堆内存中开辟一块空间，并且将这块空间的指针返回值变量引用



JS的垃圾回收

- 因为内存的大小是有限的，所以当内存不再需要的时候，我们需要对其进行释放，以便腾出更多的内存空间。
- 在手动管理内存的语言中，我们需要通过一些方式自己来释放不再需要的内存，比如 `free` 函数：
 - 但是这种管理的方式其实非常的低效，影响我们编写逻辑的代码的效率

- 并且这种方式对开发者的要求也很高，并且一不小心就会产生内存泄露
- 所以大部分现代的编程语言都是有自己的垃圾回收机制：
 - 垃圾回收的英文是 Garbage Collection，简称 GC
 - 对于那些不再使用的对象，我们都称之为是垃圾，它需要被回收，以释放更多的内存空间
 - 而我们的语言运行环境，比如Java的运行环境JVM，JavaScript的运行环境js引擎都会内存 垃圾回收器
 - 垃圾回收器我们也会简称为 gc，所以在很多地方你看到 gc 其实指的是垃圾回收器
- 但是这里又出现了另外一个很关键的问题： GC 怎么知道哪些对象是不再使用的呢？
 - 这里就要用到GC的算法了

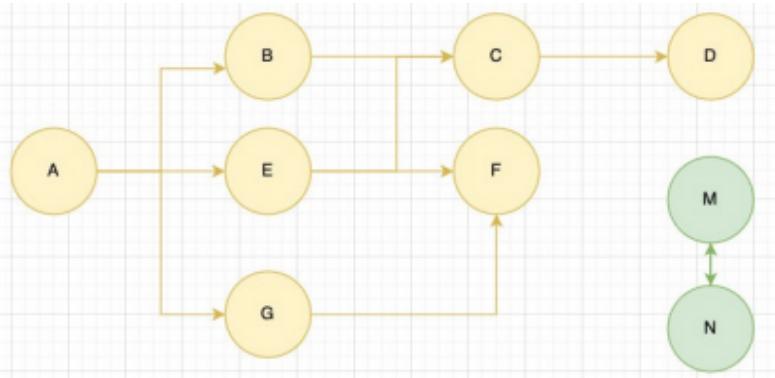
常见的GC算法 – 引用计数

- 引用计数：
 - 当一个对象有一个引用指向它时，那么这个对象的引用就 +1，当一个对象的引用为 0 时，这个对象就可以被销毁掉
- 这个算法有一个很大的弊端就是会产生循环引用



常见的GC算法 – 标记清除

- 这个算法是设置一个 根对象 (root object)，垃圾回收器会定期从这个根开始，找所有从根开始有引用到的对象，对于哪些没有引用到的对象，就认为是不可用的对象
- 这个算法可以很好的解决循环引用的问题
- JS引擎比较广泛的采用的就是标记清除算法，当然类似于V8引擎为了进行更好的优化，它在算法的实现细节上也会结合一些其他的算法



闭包

JS中函数是一等公民

- 在JavaScript中，函数是非常重要的，并且是一等公民：
 - 那么就意味着函数的使用是非常灵活的
 - 函数可以作为另外一个函数的参数，也可以作为另外一个函数的返回值来使用
- 自己编写高阶函数
- 使用内置的高阶函数

JS中闭包的定义

1. 在计算机科学中对闭包的定义（维基百科）：
 - 闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures）
 - 是在支持头等函数的编程语言中，实现词法绑定的一种技术
 - 闭包在实现上是一个结构体，它存储了一个函数和一个关联的环境（相当于一个符号查找表）
 - 闭包跟函数最大的区别在于，当捕捉闭包的时候，它的自由变量会在补充时被确定，这样即使脱离了捕捉时的上下文，它也能照常运行
- 闭包的概念出现于60年代，最早实现闭包的程序是 Scheme，那么我们就可以理解为什么JavaScript中有闭包：
 - 因为JavaScript中有大量的设计是来源于Scheme的
2. MDN对JavaScript闭包的解释：
 - 一个函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）
 - 也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域
 - 在JavaScript中，每当创建一个函数，闭包就会在函数创建的同时被创建出来
3. 总结：
 - 一个普通的函数function，如果它可以访问外层作用于的自由变量，那么这个函数就是一个闭包
 - 从广义的角度来说：JavaScript中的函数都是闭包
 - 从狭义的角度来说：**JavaScript中一个函数，如果访问了外层作用于的变量，那么它是一个闭包**

闭包的访问过程

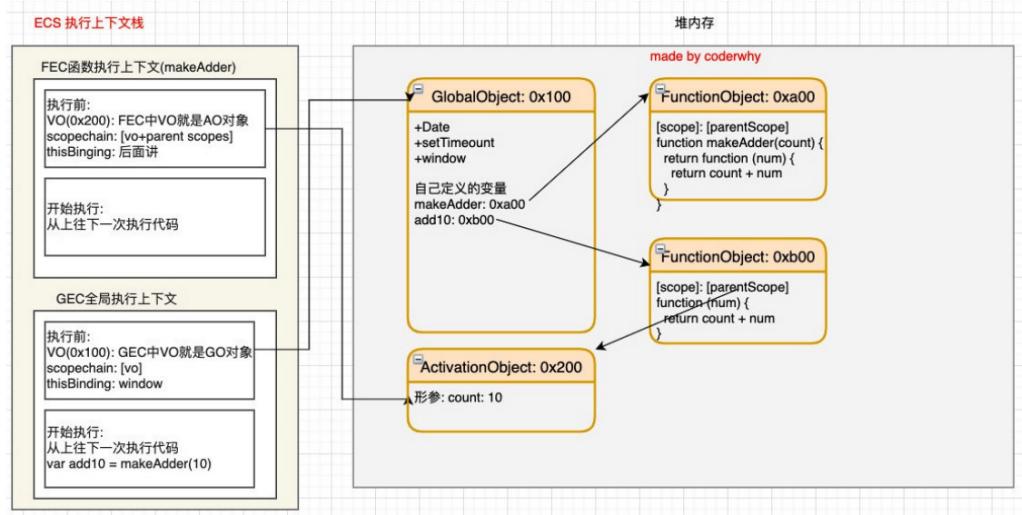
如果我们编写了如下的代码，它一定是形成了闭包的：

```

function makeAdder(count) {
  return function (num) {
    return count + num
  }
}

var add10 = makeAdder(10)
console.log(add10(5))

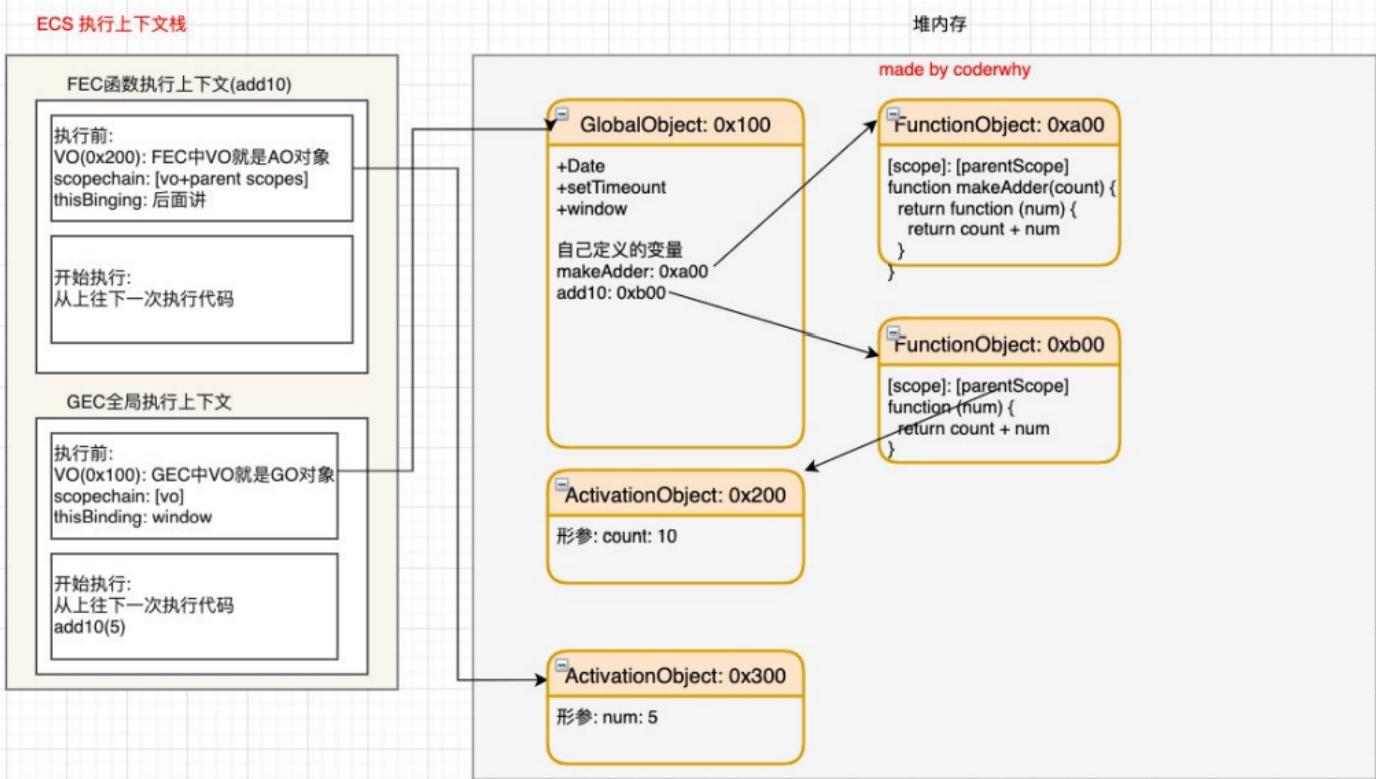
```



闭包的执行过程

当函数继续执行

- 这个时候makeAdder函数执行完毕，正常情况下我们的AO对象会被释放
- 但是在0xb00的函数中有作用域引用指向了这个AO对象，所以它不会被释放掉



闭包的内存泄露

闭包是有内存泄露的

- 在上面的案例中，如果后续我们不再使用 add10 函数了，那么该函数对象应该要被销毁掉，并且其引用着的父

作用域 A0 也应该被销毁掉

- 但是目前因为在全局作用域下 add10 变量对 0xb00 的函数对象有引用，而 0xb00 的作用域中 A0 (0x200) 有引用，所以最终会造成这些内存都是无法被释放的
- 所以我们经常说的闭包会造成内存泄露，其实就是刚才的引用链中的所有对象都是无法释放的
- 解决
 - 当将add10设置为null时，就不再对函数对象0xb00有引用，那么对应的AO对象0x200也就不可达了
 - 在GC的下一次检测中，它们就会被销毁掉

```
add10 = null
```

AO不使用的属性

- js引擎会自动清除没有被引用的属性

JS函数的this指向

this指向

1. this 在全局作用于下指向 window
2. 函数中使用
 - 所有的函数在被调用时，都会创建一个执行上下文：
 - 这个上下文中记录着函数的 调用栈、 AO对象 等
 - this 也是其中的一条记录
3. 函数在调用时，JavaScript会默认给this绑定一个值
 - this的绑定和定义的位置（编写的位置）没有关系
 - this的绑定和调用方式以及调用的位置有关系
 - this是在运行时被绑定的
4. 绑定规则
 1. 默认绑定
 2. 隐式绑定
 3. 显示绑定
 4. new 绑定

规则一：默认绑定

独立的函数调用我们可以理解成函数没有被绑定到某个对象上进行调用

独立函数调用使用默认绑定

```
// 1. 案例一:  
function foo() {  
  console.log(this);  
}  
  
foo();
```

```
// 2. 案例二:  
function test1() {  
  console.log(this);  
  test2();  
}  
  
function test2() {  
  console.log(this);  
  test3();  
}  
  
function test3() {  
  console.log(this);  
}  
  
test1();
```

```
// 3. 案例三:  
function foo(func) {  
  func()  
}  
  
var obj = {  
  name: "why",  
  bar: function() {  
    console.log(this);  
  }  
}  
  
foo(obj.bar);
```

规则二：隐式绑定

通过某个对象进行调用使用隐式绑定

```
// 1. 通过对象调用  
function foo() {  
  console.log(this); // obj对象  
}  
  
var obj = {  
  name: "why",  
  foo: foo  
}  
  
obj.foo();
```

```
function foo() {  
  console.log(this);  
}  
  
var obj1 = {  
  name: "obj1",  
  foo: foo  
}  
  
var obj2 = {  
  name: "obj2",  
  obj1: obj1  
}  
  
obj2.obj1.foo();
```

```
function foo() {  
  console.log(this);  
}  
  
var obj1 = {  
  name: "obj1",  
  foo: foo  
}  
  
// 将obj1的foo赋值给bar  
var bar = obj1.foo;  
bar();
```

规则三：显示绑定

隐式绑定有一个前提条件：

- 必须在调用的对象内部有一个对函数的引用（比如一个属性）
- 如果没有这样的引用，在进行调用时，会报找不到该函数的错误
- 正是通过这个引用，间接的将 this 绑定到了这个对象上

如果我们不希望在 对象内部 包含这个函数的引用，同时又希望在这个对象上进行强制调用，该怎么做呢？

- JavaScript所有的函数都可以使用 call 和 apply 方法（和Prototype有关）
 - 区别：第一个参数是相同的，后面的参数， apply 为数组， call 为参数列表
 - 这两个函数的第一个参数都要求是一个对象，这个对象就是给this准备的
 - 在调用这个函数时，会将this绑定到这个传入的对象上

- 因为上面的过程，我们明确的绑定了this指向的对象，所以称之为 **显示绑定**

call、apply、bind

- 通过 call 或者 apply 绑定 this 对象
 - 显示绑定后， this 就会明确的指向绑定的对象

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

foo.call(window) //window
foo.call({name: 'hidari'}) // {name: 'hidari'}
foo.call(123) // Number(123)
```

- 如果我们希望一个函数总是显示的绑定到一个对象上，可以用 bind

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

var obj = {
  name: 'hidari'
}

var bar = foo.bind(obj)
bar() // obj对象
```

内置函数的绑定思考

- 有些时候，我们会调用一些JavaScript的内置函数，或者一些第三方库中的内置函数。
 - 这些内置函数会要求我们传入另外一个函数；
 - 我们自己并不会显示的调用这些函数，而且JavaScript内部或者第三方库内部会帮助我们执行；
 - 这些函数中的 this 又是如何绑定的呢
- setTimeout、数组的 forEach、 div 的点击

```
setTimeout(function() {
  console.log(this) // window
}, 1000)
```

```
var names = ['abc','cba']
var obj = { name: 'hidari' }
names.forEach(function (item) {
  console.log(this) // 三次 obj 对象
}, obj) // obj 为 this 绑定对象 thisArg 参数
```

```
var box = document.querySelector('.box')
box.onclick = function () {
  console.log(this === box)
}
```

规则四：显示绑定

- JavaScript中的函数可以当做一个类的构造函数来使用，也就是使用 new 关键字
- 使用 new 关键字来调用函数是，会执行如下的操作：
 1. 创建一个全新的对象
 2. 这个新对象会被执行 prototype 连接
 3. 这个新对象会绑定到函数调用的 this 上（this 的绑定在这个步骤完成）
 4. 如果函数没有返回其他对象，表达式会返回这个新对象

```
// 创建 Person
function Person(name) {
  console.log(this) // Person {}
  this.name = name // Person { name: 'hidari' }
}
var p = new Person('hidari')
console.log(p)
```

规则优先级

1. 默认规则的优先级最低

毫无疑问，默认规则的优先级是最低的，因为存在其他规则时，就会通过其他规则的方式来绑定 this

2. 显示绑定优先级高于隐式绑定
3. new 绑定优先级高于隐式绑定
4. new 绑定优先级高于 bind
 - new 绑定和 call、apply 是不允许同时使用的，所以不存在谁的优先级更高
 - new 绑定可以和 bind 一起使用，new 绑定优先级更高

this规则之外 – 忽略显示绑定

如果在显示绑定中，我们传入一个 null 或者 undefined，那么这个显示绑定会被忽略，使用默认规则：

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

var obj = { name: 'hidari'}

foo.call(null) // window
foo.call(undefined) // window

var bar = foo.bind(null)
bar() // window
```

this规则之外 - 间接函数引用

另外一种情况，创建一个函数的 间接引用，这种情况使用默认绑定规则

- 赋值 (`obj2.foo = obj1.foo`) 的结果是 `foo` 函数
- `foo` 函数被直接调用，那么是默认绑定

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

var obj1 = {
  name: 'obj1'
  foo: foo
}

var obj2 = { name: 'obj2'}

obj1.foo() // obj1
(obj2.foo = obj1.foo)() // window
```

this规则之外 – ES6箭头函数

箭头函数不使用this的四种标准规则（也就是不绑定this），而是根据外层作用域来决定this

面试题

```
var name = "window";

var person = {
  name: "person",
  sayName: function () {
    console.log(this.name);
  }
};

function sayName() {
  var sss = person.sayName;
  sss(); // window: 独立函数调用
  person.sayName(); // person: 隐式调用
  (person.sayName)(); // person: 隐式调用
  (b = person.sayName)(); // window: 赋值表达式(独立函数调用)
}

sayName();
```

```
var name = 'window'

var person1 = {
  name: 'person1',
  foo1: function () {
    console.log(this.name)
  },
  foo2: () => console.log(this.name),
  foo3: function () {
    return function () {
      console.log(this.name)
    }
  },
  foo4: function () {
    return () => {
      console.log(this.name)
    }
  }
}

var person2 = { name: 'person2' }

person1.foo1(); // person1(隐式绑定)
person1.foo1.call(person2); // person2(显示绑定优先级大于隐式绑定)

person1.foo2(); // window(不绑定作用域,上层作用域是全局)
person1.foo2.call(person2); // window

person1.foo3()(); // window(独立函数调用)
person1.foo3.call(person2)(); // window(独立函数调用)
person1.foo3().call(person2); // person2(最终调用返回函数式, 使用的是显示绑定)

person1.foo4()(); // person1(箭头函数不绑定this, 上层作用域this是person1)
person1.foo4.call(person2)(); // person2(上层作用域被显示的绑定了一个person2)
person1.foo4().call(person2); // person1(上层找到person1)
```

```
var name = 'window'

function Person (name) {
  this.name = name
  this.foo1 = function () {
    console.log(this.name)
  },
  this.foo2 = () => console.log(this.name),
  this.foo3 = function () {
    return function () {
      console.log(this.name)
    }
  },
  this.foo4 = function () {
    return () => {
      console.log(this.name)
    }
  }
}

var person1 = new Person('person1')
var person2 = new Person('person2')

person1.foo1() // person1
person1.foo1.call(person2) // person2(显示高于隐式绑定)

person1.foo2() // person1 (上层作用域中的this是person1)
person1.foo2.call(person2) // person1 (上层作用域中的this是person1)

person1.foo3()() // window(独立函数调用)
person1.foo3.call(person2)() // window
person1.foo3().call(person2) // person2

person1.foo4()() // person1
person1.foo4.call(person2)() // person2
person1.foo4().call(person2) // person1

var obj = {
  name: "obj",
  foo: function() {

  }
}
```

```

var name = 'window'

function Person (name) {
  this.name = name
  this.obj = {
    name: 'obj',
    foo1: function () {
      return function () {
        console.log(this.name)
      }
    },
    foo2: function () {
      return () => {
        console.log(this.name)
      }
    }
  }
}

var person1 = new Person('person1')
var person2 = new Person('person2')

person1.obj.foo1()() // window
person1.obj.foo1.call(person2)() // window
person1.obj.foo1().call(person2) // person2

person1.obj.foo2()() // obj
person1.obj.foo2.call(person2)() // person2
person1.obj.foo2().call(person2) // obj

```

```

// 上层作用域的理解
var obj = {
  name: "obj",
  foo: function() {
    // 上层作用域是全局
  }
}

function Student() {
  this.foo = function() {}
}

```

JS函数式编程

实现apply、call、bind

call

```

// apply/call/bind的用法

// 给所有的函数添加一个hycall的方法
Function.prototype.hycall = function(thisArg, ...args) {
    // 在这里可以去执行调用的那个函数(foo)
    // 问题：得可以获取到是哪一个函数执行了hycall
    // 1. 获取需要被执行的函数
    var fn = this

    // 2. 对thisArg转成对象类型(防止它传入的是非对象类型)
    thisArg = (thisArg !== null && thisArg !== undefined) ? Object(thisArg) : window

    // 3. 调用需要被执行的函数
    thisArg.fn = fn
    var result = thisArg.fn(...args)
    delete thisArg.fn

    // 4. 将最终的结果返回出去
    return result
}

function foo() {
    console.log("foo函数被执行", this)
}

function sum(num1, num2) {
    console.log("sum函数被执行", this, num1, num2)
    return num1 + num2
}

// 系统的函数的call方法
foo.call(undefined)
var result = sum.call({}, 20, 30)
// console.log("系统调用的结果:", result)

// 自己实现的函数的hycall方法
// 默认进行隐式绑定
// foo.hycall({name: "why"})
foo.hycall(undefined)
var result = sum.hycall("abc", 20, 30)
console.log("hycall的调用:", result)

// var num = {name: "why"}
// console.log(Object(num))

```

apply

```
// 自己实现hyapply
Function.prototype.hyapply = function(thisArg, argArray) {
    // 1.获取到要执行的函数
    var fn = this

    // 2.处理绑定的thisArg
    thisArg = (thisArg !== null && thisArg !== undefined) ? Object(thisArg): window

    // 3.执行函数
    thisArg.fn = fn
    var result
    // if (!argArray) { // argArray是没有值(没有传参数)
    //     result = thisArg.fn()
    // } else { // 有传参数
    //     result = thisArg.fn(...argArray)
    // }

    // argArray = argArray ? argArray: []
    argArray = argArray || []
    result = thisArg.fn(...argArray)

    delete thisArg.fn

    // 4.返回结果
    return result
}

function sum(num1, num2) {
    console.log("sum被调用", this, num1, num2)
    return num1 + num2
}

function foo(num) {
    return num
}

function bar() {
    console.log("bar函数被执行", this)
}

// 系统调用
// var result = sum.apply("abc", 20)
// console.log(result)

// 自己实现的调用
// var result = sum.hyapply("abc", [20, 30])
// console.log(result)

// var result2 = foo.hyapply("abc", [20])
// console.log(result2)

// edge case
bar.hyapply()
```

bind

```
Function.prototype.hybind = function(thisArg, ...argArray) {
    // 1. 获取到真实需要调用的函数
    var fn = this

    // 2. 绑定this
    thisArg = (thisArg !== null && thisArg !== undefined) ? Object(thisArg): window

    function proxyFn(...args) {
        // 3. 将函数放到thisArg中进行调用
        thisArg.fn = fn
        // 特殊：对两个传入的参数进行合并
        var finalArgs = [...argArray, ...args]
        var result = thisArg.fn(...finalArgs)
        delete thisArg.fn

        // 4. 返回结果
        return result
    }

    return proxyFn
}

function foo() {
    console.log("foo被执行", this)
    return 20
}

function sum(num1, num2, num3, num4) {
    console.log(num1, num2, num3, num4)
}

// 系统的bind使用
var bar = foo.bind("abc")
bar()

// var newSum = sum.bind("aaa", 10, 20, 30, 40)
// newSum()

// var newSum = sum.bind("aaa")
// newSum(10, 20, 30, 40)

// var newSum = sum.bind("aaa", 10)
// newSum(20, 30, 40)

// 使用自己定义的bind
// var bar = foo.hybind("abc")
// var result = bar()
// console.log(result)

var newSum = sum.hybind("abc", 10, 20)
var result = newSum(30, 40)
```

arguments

- `arguments` 是一个对应于传递给函数的参数的类数组(array-like)对象。
- `array-like` 意味着它不是一个数组类型，而是一个对象类型：
 - 但是它却拥有数组的一些特性，比如说 `length`，比如可以通过 `index` 索引来访问；
 - 但是它却没有数组的一些方法，比如 `forEach`、`map` 等
- `arguments` 转成 `array`

```

function foo(num1, num2) {
    // 1.自己遍历
    var newArr = []
    for (var i = 0; i < arguments.length; i++) {
        newArr.push(arguments[i])
    }
    console.log(newArr)

    // 2.arguments转成array类型
    // 2.1.自己遍历arguments中所有的元素

    // 2.2.Array.prototype.slice将arguments转成array
    var newArr2 = Array.prototype.slice.call(arguments)
    console.log(newArr2)

    var newArr3 = [].slice.call(arguments)
    console.log(newArr3)

    // 2.3.ES6的语法
    var newArr4 = Array.from(arguments)
    console.log(newArr4)
    var newArr5 = [...arguments]
    console.log(newArr5)
}

foo(10, 20, 30, 40, 50)

// 额外补充的知识点：Array中的slice实现
Array.prototype.hyslice = function(start, end) {
    var arr = this
    start = start || 0
    end = end || arr.length
    var newArray = []
    for (var i = start; i < end; i++) {
        newArray.push(arr[i])
    }
    return newArray
}

var newArray = Array.prototype.hyslice.call(["aaaa", "bbb", "cccc"], 1, 3)
console.log(newArray)

var names = ["aaa", "bbb", "ccc", "ddd"]
names.slice(1, 3)

```

箭头函数不绑定arguments，没有显式原型

箭头函数是不绑定 arguments 的，所以我们在箭头函数中使用 arguments 会去上层作用域查找

箭头函数是没有显式原型的，所以不能作为构造函数，使用 new 来创建对象

```

// 1.案例一:
var foo = () => {
  console.log(arguments)
}

// foo()

// 2.案例二:
function foo() {
  var bar = () => {
    console.log(arguments)
  }
  return bar
}

var fn = foo(123)
fn()

// 3.案例三:
var foo = (num1, num2, ...args) => {
  console.log(args)
}

foo(10, 20, 30, 40, 50)

```

JavaScript纯函数

- 函数式编程中有一个非常重要的概念叫纯函数，JavaScript符合函数式编程的范式，所以也有纯函数的概念
 - 在 react 开发中纯函数是被多次提及的
 - 比如 react 中组件就被要求像是一个纯函数（为什么是像，因为还有 class 组件），redux 中有一个 reducer 的概念，也是要求必须是一个纯函数
- 纯函数的维基百科定义：
 - 在程序设计中，若一个函数符合以下条件，那么这个函数被称为纯函数：
 - 此函数在相同的输入值时，需产生相同的输出
 - 函数的输出和输入值以外的其他隐藏信息或状态无关，也和由I/O设备产生的外部输出无关
 - 该函数不能有语义上可观察的函数副作用，诸如“触发事件”，使输出设备输出，或更改输出值以外物件的内容等
- 总结：
 - 确定的输入，一定会产生确定的输出
 - 函数在执行过程中，不能产生副作用
 - 副作用表示在执行一个函数时，除了返回函数值之外，还对调用函数产生了附加的影响，比如 修改了全局变量，修改参数 或者 改变外部的存储

纯函数的案例

对数组操作的两个函数：

- `slice` : `slice` 截取数组时不会对原数组进行任何操作,而是生成一个新的数组
- `splice` : `splice` 截取数组, 会返回一个新的数组, 也会对原数组进行修改
- `slice` 就是一个纯函数, 不会修改传入的参数

```
var names = ["abc", "cba", "nba", "dna"]

// slice只要给它传入一个start/end, 那么对于同一个数组来说, 它会给我们返回确定的值
// slice函数本身它是不会修改原来的数组
// slice -> this
// slice函数本身就是一个纯函数
var newNames1 = names.slice(0, 3)
console.log(newNames1)
console.log(names)

// ["abc", "cba", "nba", "dna"]
// splice在执行时, 有修改掉调用的数组对象本身, 修改的这个操作就是产生的副作用
// splice不是一个纯函数
var newNames2 = names.splice(2)
console.log(newNames2)
console.log(names)
```

纯函数的优势

- 写的时候保证了函数的纯度, 只是单纯实现自己的业务逻辑即可, 不需要关心传入的内容是如何获得的或者依赖其他的外部变量是否已经发生了修改
- 在用的时候, 确定输入内容不会被任意篡改, 并且自己确定的输入, 一定会有确定的输出
- React 中就要求我们无论是函数还是 class 声明一个组件, 这个组件都必须像纯函数一样, 保护它们的 props 不被修改

JavaScript柯里化

维基百科的解释：

- 在计算机科学中, 柯里化 (英语: Currying) , 又译为卡瑞化或加里化
- 是把接收多个参数的函数, 变成接受一个单一参数 (最初函数的第一个参数) 的函数, 并且返回接受余下的参数, 而且返回结果的新函数的技术
- 柯里化声称 “如果你固定某些参数, 你将得到接受余下参数的一个函数”

总结：

- 只传递给函数一部分参数来调用它, 让它返回一个函数去处理剩余的参数
- 这个过程就称之为柯里化

柯里化的结构

```

function add(x, y, z) {
  return x + y + z
}

var result = add(10, 20, 30)
console.log(result)

function sum1(x) {
  return function(y) {
    return function(z) {
      return x + y + z
    }
  }
}

var result1 = sum1(10)(20)(30)
console.log(result1)

// 简化柯里化的代码
var sum2 = x => y => z => {
  return x + y + z
}

console.log(sum2(10)(20)(30))

var sum3 = x => y => z => x + y + z
console.log(sum3(10)(20)(30))

```

柯里化作用

1. 让函数的职责单一

- 在函数式编程中，我们其实往往希望一个函数处理的问题尽可能的单一，而不是将一大堆的处理过程交给一个函数来处理
- 那么我们是否就可以将每次传入的参数在单一的函数中进行处理，处理完后在下一个函数中再使用处理后的结果

2. 复用参数逻辑：

- `makeAdder` 函数要求我们传入一个 `num` (并且如果我们需要的话，可以在这里对 `num` 进行一些修改)；
- 在之后使用返回的函数时，我们不需要再继续传入 `num` 了

```
// function sum(m, n) {
//   return m + n
// }

// // 假如在程序中,我们经常需要把5和另外一个数字进行相加
// console.log(sum(5, 10))
// console.log(sum(5, 14))
// console.log(sum(5, 1100))
// console.log(sum(5, 555))

function makeAdder(count) {
  count = count * count

  return function(num) {
    return count + num
  }
}

// var result = makeAdder(5)(10)
// console.log(result)
var adder5 = makeAdder(5)
adder5(10)
adder5(14)
adder5(1100)
adder5(555)
```

- 打印日志中的柯里化

```

function log(date, type, message) {
  console.log(`[${date.getHours()}]:[${date.getMinutes()}]:[${type}]:[${message}]`)
}

// log(new Date(), "DEBUG", "查找到轮播图的bug")
// log(new Date(), "DEBUG", "查询菜单的bug")
// log(new Date(), "DEBUG", "查询数据的bug")

// 柯里化的优化
var log = date => type => message => {
  console.log(`[${date.getHours()}]:[${date.getMinutes()}]:[${type}]:[${message}]`)
}

// 如果我现在打印的都是当前时间
var nowLog = log(new Date())
nowLog("DEBUG")("查找到轮播图的bug")
nowLog("FETURE")("新增了添加用户的功能")

var nowAndDebugLog = log(new Date())("DEBUG")
nowAndDebugLog("查找到轮播图的bug")
nowAndDebugLog("查找到轮播图的bug")
nowAndDebugLog("查找到轮播图的bug")
nowAndDebugLog("查找到轮播图的bug")

var nowAndFetureLog = log(new Date())("FETURE")
nowAndFetureLog("添加新功能~")

```

自动柯里化函数

```

// 柯里化函数的实现hyCurrying
function hyCurrying(fn) {
  function curried(...args) {
    // 判断当前已经接收的参数的个数，可以参数本身需要接受的参数是否已经一致了
    // 1.当已经传入的参数 大于等于 需要的参数时，就执行函数
    if (args.length >= fn.length) {
      // fn(...args)
      // fn.call(this, ...args)
      return fn.apply(this, args)
    } else {
      // 没有达到个数时，需要返回一个新的函数，继续来接收的参数
      function curried2(...args2) {
        // 接收到参数后，需要递归调用curried来检查函数的个数是否达到
        return curried.apply(this, args.concat(args2))
      }
      return curried2
    }
  }
  return curried
}

```

组合函数

- 组合 (Compose) 函数是在JavaScript开发过程中一种对函数的使用技巧、模式：
 - 比如我们现在需要对某一个数据进行函数的调用，执行两个函数fn1和fn2，这两个函数是依次执行的
 - 那么如果每次都需要进行两个函数的调用，操作上就会显得重复
 - 可以将这两个函数组合起来，自动依次调用
 - 这个过程就是对函数的组合，我们称之为 **组合函数 (Compose Function)**

```
function hyCompose(...fns) {  
    var length = fns.length  
    // 遍历所有的参数 如果不是函数 报错  
    for (var i = 0; i < length; i++) {  
        if (typeof fns[i] !== 'function') {  
            throw new TypeError("Expected arguments are functions")  
        }  
    }  
  
    // 取出所有函数 依次调用  
    function compose(...args) {  
        var index = 0  
        var result = length ? fns[index].apply(this, args) : args  
        while(++index < length) {  
            result = fns[index].call(this, result)  
        }  
        return result  
    }  
    return compose  
}  
  
function double(m) {  
    return m * 2  
}  
  
function square(n) {  
    return n ** 2  
}  
  
var newFn = hyCompose(double, square)  
console.log(newFn(10))
```

JS额外知识补充

with语句

with语句 扩展一个语句的作用域链

```

"use strict";

var message = "Hello World"
// console.log(message)

// with语句：可以形成自己的作用域
var obj = {name: "why", age: 18, message: "obj message"}

function foo() {
  function bar() {
    with(obj) {
      console.log(message)
      console.log("-----")
    }
  }
  bar()
}

foo()

var info = {name: "kobe"}
with(info) {
  console.log(name)
}

```

- 不建议使用with语句，因为它可能是混淆错误和兼容性问题的根源

eval函数

- eval是一个特殊的函数，它可以将传入的字符串当做JavaScript代码来运行

```

var jsString = 'var message = "Hello World"; console.log(message);'

var message = "Hello World"
console.log(message)

eval(jsString)

```

- 不建议在开发中使用 eval：
 - eval 代码的可读性非常的差（代码的可读性是高质量代码的重要原则）
 - eval 是一个字符串，那么有可能在执行的过程中被刻意篡改，那么可能会造成被攻击的风险
 - eval 的执行必须经过JS解释器，不能被JS引擎优化

严格模式

- 在 ECMAScript5 标准中，JavaScript提出了 严格模式 的概念（Strict Mode）：
 - 严格模式很好理解，是一种具有限制性的JavaScript模式，从而使代码隐式的脱离了懒散（sloppy）模式

- 支持严格模式的浏览器在检测到代码中有严格模式时，会以更加严格的方式对代码进行检测和执行
- 严格模式对正常的JavaScript语义进行了一些限制：
 - 严格模式通过 抛出错误 来消除一些原有的 静默 (silent) 错误
 - 严格模式让JS引擎在执行代码时可以进行更多的优化（不需要对一些特殊的语法进行处理）
 - 严格模式禁用了在 ECMAScript 未来版本中可能会定义的一些语法
- 严格模式支持粒度话的迁移：
 - 可以支持在js文件中开启严格模式
 - 也支持对某一个函数开启严格模式
- 严格模式通过在文件或者函数开头使用 `use strict` 来开启。
- 严格模式限制：
 1. 无法意外的创建全局变量
 2. 严格模式会使引起 静默失败(silently fail,注:不报错也没有任何效果) 的赋值操作抛出异常
 3. 严格模式下试图删除不可删除的属性
 4. 严格模式不允许函数参数有相同的名称
 5. 不允许 `o` 的八进制语法
 6. 在严格模式下，不允许使用 `with`
 7. 在严格模式下， `eval` 不再为上层引用变量
 8. 严格模式下， `this` 绑定不会默认转成对象

JS面向对象

JavaScript支持多种编程范式的，包括函数式编程和面向对象编程：

- JavaScript中的对象被设计成一组属性的无序集合，像是一个哈希表，有 `key` 和 `value` 组成；
- `key` 是一个标识符名称，`value` 可以是任意类型，也可以是其他对象或者函数类型；
- 如果值是一个函数，那么我们可以称之为是对象的方法
- 创建对象的两种方式

```

// 创建一个对象，对某一个人进行抽象(描述)
// 1.创建方式一：通过new Object()创建
var obj = new Object()
obj.name = "why"
obj.age = 18
obj.height = 1.88
obj.running = function() {
  console.log(this.name + "在跑步~")
}

// 2.创建方式二：字面量形式
var info = {
  name: "kobe",
  age: 40,
  height: 1.98,
  eating: function() {
    console.log(this.name + "在吃东西~")
  }
}

```

对属性操作的控制

- 如果我们想要对一个属性进行比较精准的操作控制，那么我们就可以使用属性描述符
 - 通过属性描述符可以精准的添加或修改对象的属性
 - 属性描述符需要使用 `Object.defineProperty` 来对属性进行添加或者修改
- `Object.defineProperty()` 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象
 - 可接收三个参数：
 - `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
 - 返回值：被传递给函数的对象
- 属性描述符的类型有两种：
 - 数据属性 (Data Properties) 描述符 (Descriptor)
 - 存取属性 (Accessor Properties) 描述符 (Descriptor)

	<code>configurable</code>	<code>enumerable</code>	<code>value</code>	<code>writable</code>	<code>get</code>	<code>set</code>
数据描述符	可以	可以	可以	可以	不可以	不可以
存取描述符	可以	可以	不可以	不可以	可以	可以

数据属性描述符

数据数据描述符有如下四个特性：

- `[[Configurable]]`：表示属性是否可以通过 `delete` 删除属性，是否可以修改它的特性，或者是否可以将它修改为存取属性描述符；
 - 当我们直接在一个对象上定义某个属性时，这个属性的 `[[Configurable]]` 为 `true`
 - 当我们通过属性描述符定义一个属性时，这个属性的 `[[Configurable]]` 默认为 `false`
- `[[Enumerable]]`：表示属性是否可以通过 `for-in` 或者 `Object.keys()` 返回该属性
 - 当我们直接在一个对象上定义某个属性时，这个属性的 `[[Enumerable]]` 为 `true`
 - 当我们通过属性描述符定义一个属性时，这个属性的 `[[Enumerable]]` 默认为 `false`
- `[[Writable]]`：表示是否可以修改属性的值
 - 当我们直接在一个对象上定义某个属性时，这个属性的 `[[Writable]]` 为 `true`
 - 当我们通过属性描述符定义一个属性时，这个属性的 `[[Writable]]` 默认为 `false`
- `[[value]]`：属性的 `value` 值，读取属性时会返回该值，修改属性时，会对其进行修改
 - 默认情况下这个值是 `undefined`

存取属性描述符

数据数据描述符有如下四个特性：

- `[[Configurable]]`：表示属性是否可以通过 `delete` 删除属性，是否可以修改它的特性，或者是否可以将它修改为存取属性描述符
 - 和数据属性描述符是一致的
 - 当我们直接在一个对象上定义某个属性时，这个属性的 `Configurable` 为 `true`
 - 当我们通过属性描述符定义一个属性时，这个属性的 `Configurable` 默认为 `false`
- `[[Enumerable]]`：表示属性是否可以通过 `for-in` 或者 `Object.keys()` 返回该属性
 - 和数据属性描述符是一致的
 - 当我们直接在一个对象上定义某个属性时，这个属性的 `[[Enumerable]]` 为 `true`
 - 当我们通过属性描述符定义一个属性时，这个属性的 `[[Enumerable]]` 默认为 `false`
- `[[get]]`：获取属性时会执行的函数。默认为 `undefined`
- `[[set]]`：设置属性时会执行的函数。默认为 `undefined`

同时定义多个属性

`Object.defineProperties()` 方法直接在一个对象上定义 多个 新的属性或修改现有属性，并且返回该对象。

```

var obj = {
  // 私有属性(js里面是没有严格意义的私有属性)
  _age: 18,
  _eating: function() {},
  set age(value) {
    this._age = value
  },
  get age() {
    return this._age
  }
}

Object.defineProperties(obj, {
  name: {
    configurable: true,
    enumerable: true,
    writable: true,
    value: "why"
  },
  age: {
    configurable: true,
    enumerable: true,
    get: function() {
      return this._age
    },
    set: function(value) {
      this._age = value
    }
  }
})

obj.age = 19
console.log(obj.age)

console.log(obj)

```

对象方法补充

- 获取对象的属性描述符：
 - `getOwnPropertyDescriptor`
 - `getOwnPropertyDescriptors`
- 禁止对象扩展新属性： `preventExtensions`
 - 给一个对象添加新的属性会失败（在严格模式下会报错）
- 密封对象，不允许配置和删除属性： `seal`
 - 实际是调用 `preventExtensions`
 - 并且将现有属性的 `configurable:false`
- 冻结对象，不允许修改现有属性： `freeze`
 - 实际上是调用 `seal`
 - 并且将现有属性的 `writable: false`

创建多个对象的方案

- 如果我们现在希望创建一系列的对象：比如 Person 对象
 - 包括张三、李四、王五、李雷等等，他们的信息各不相同
 - 那么采用什么方式来创建比较好呢

工厂模式

- 工厂模式其实是一种常见的设计模式
- 通常我们会有一个工厂方法，通过该工厂方法我们可以产生想要的对象

```
// 工厂模式：工厂函数
function createPerson(name, age, height, address) {
  var p = {}
  p.name = name
  p.age = age
  p.height = height;
  p.address = address

  p.eating = function() {
    console.log(this.name + "在吃东西~")
  }

  p.running = function() {
    console.log(this.name + "在跑步~")
  }

  return p
}

var p1 = createPerson("张三", 18, 1.88, "广州市")
var p2 = createPerson("李四", 20, 1.98, "上海市")
var p3 = createPerson("王五", 30, 1.78, "北京市")

// 工厂模式的缺点(获取不到对象最真实的类型)
console.log(p1, p2, p3)
```

构造函数

- 工厂方法创建对象有一个比较大的问题：我们在打印对象时，对象的类型都是 Object 类型
 - 但是从某些角度来说，这些对象应该有一个他们共同的类型
- 什么是构造函数
 - 构造函数也称之为 构造器 (constructor)，通常是我们创建对象时会调用的函数
 - 在其他面向的编程语言里面，构造函数是存在于类中的一个方法，称之为构造方法
 - 但是JavaScript中的构造函数有点不太一样
- JavaScript中的构造函数是怎么样的？
 - 构造函数也是一个普通的函数，从表现形式来说，和普通的函数没有任何区别
 - 那么如果这么一个普通的函数被使用 new 操作符来调用了，那么这个函数就称之为是一个构造函数

- new操作符调用的作用
 - 如果一个函数被使用new操作符调用了，那么它会执行如下操作：
 1. 在内存中创建一个新的对象（空对象）
 2. 这个对象内部的 [[prototype]] 属性会被赋值为该构造函数的 prototype 属性
 3. 构造函数内部的 this，会指向创建出来的新对象
 4. 执行函数的内部代码（函数体代码）
 5. 如果构造函数没有返回非空对象，则返回创建出来的新对象
 - 构造函数也是有缺点的，它在于我们需要为每个对象的函数去创建一个函数对象实例

对象的原型

JavaScript当中每个对象都有一个特殊的内置属性 [[prototype]]，这个特殊的对象可以指向另外一个对象。

- 那么这个对象有什么用呢？
 - 当我们通过引用对象的属性 key 来获取一个 value 时，它会触发 [[Get]] 的操作
 - 这个操作会首先检查该属性是否有对应的属性，如果有的话就使用它
 - 如果对象中没有改属性，那么会访问对象 [[prototype]] 内置属性指向的对象上的属性
- 那么如果通过字面量直接创建一个对象，这个对象也会有这样的属性吗？如果有，应该如何获取这个属性呢？
 - 答案是有的，只要是对象都会有这样的一个内置属性
- 获取的方式有两种：
 - 方式一：通过对对象的 __proto__ 属性可以获取到（但是这个是早期浏览器自己添加的，存在一定的兼容性问题）
 - 方式二：通过 Object.getPrototypeOf 方法可以获取到

函数的原型 prototype

所有的函数都有一个 prototype 的属性

- 因为函数是一个函数，才有 prototype 属性
- new 操作符

new 关键字的步骤：

1. 在内存中创建一个新的对象（空对象）
 2. 这个对象内部的 [[prototype]] 属性会被赋值为该构造函数的 prototype 属性
- 那么也就意味着我们通过 Person 构造函数创建出来的所有对象的 [[prototype]] 属性都指向 Person.prototype

```
var p1 = new Person()
```

```
var p2 = new Person()
```

function Person() {}

prototype

Person函数的原型对象

constructor

p1对象

__proto__

p2对象

__proto__

function foo() {}

prototype

foo函数的原型对象

constructor

f1对象

__proto__

新的对象

constructor

why

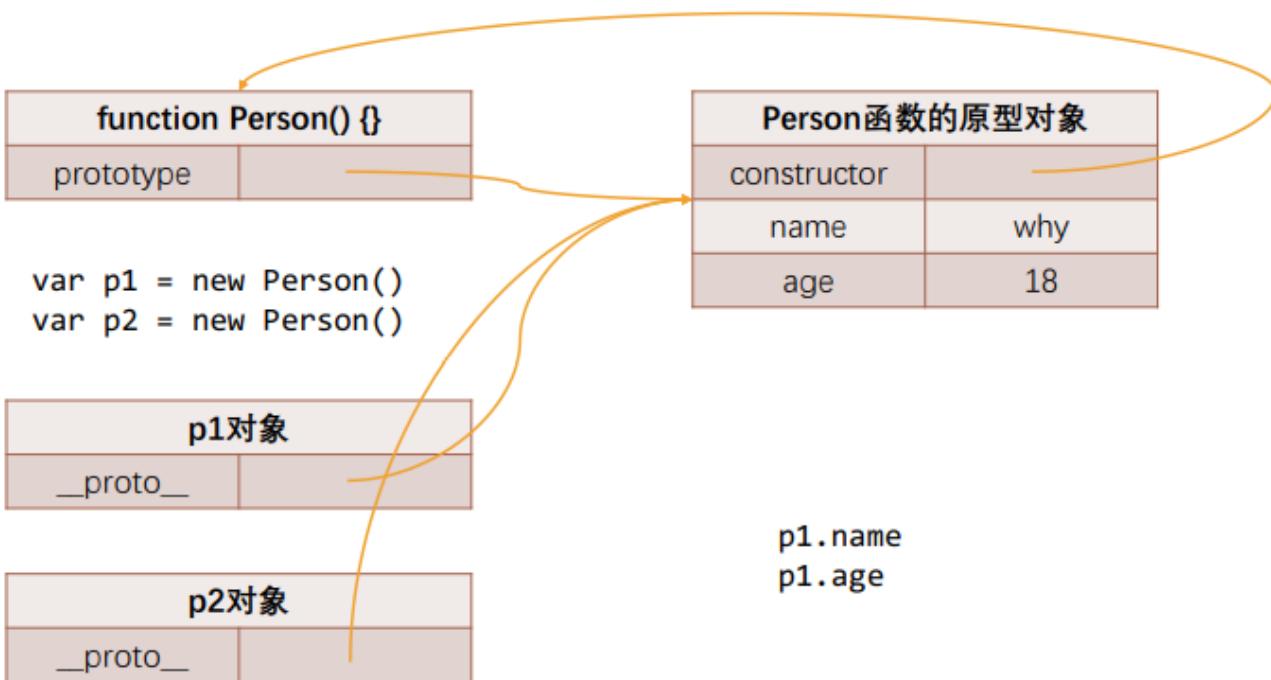
name

18

age

1.88

height



- `constructor` 属性

- 事实上原型对象上面是有一个属性的： `constructor`
- 默认情况下原型上都会添加一个属性叫做 `constructor`，这个 `constructor` 指向当前的函数对象

重写原型对象

- 如果我们需要在原型上添加过多的属性，通常我们会重新整个原型对象：

```
function Person() {}

Person.prototype = {
  name: 'hidari'
}
```

- 前面我们说过，每创建一个函数，就会同时创建它的 `prototype` 对象，这个对象也会自动获取 `constructor` 属性；
 - 而我们这里相当于给 `prototype` 重新赋值了一个对象，那么这个新对象的 `constructor` 属性，会指向 `Object` 构造函数，而不是 `Person` 构造函数了
- 如果希望 `constructor` 指向 `Person`，那么可以手动添加：
- 上面的方式虽然可以，但是也会造成 `constructor` 的 `[[Enumerable]]` 特性被设置了 `true`。
 - 默认情况下，原生的 `constructor` 属性是不可枚举的。
 - 如果希望解决这个问题，就可以使用我们前面介绍的 `Object.defineProperty()` 函数了

```

Person.prototype = {
  constructor: Person,
  name: 'hidari'
}
Object.defineProperty(Person.prototype, "constructor", {
  enumerable: false,
  value: Person
})

```

面向对象的特性 – 继承

面向对象有三大特性：封装、继承、多态

- 封装：我们前面将属性和方法封装到一个类中，可以称之为封装的过程；
- 继承：继承是面向对象中非常重要的，不仅仅可以减少重复代码的数量，也是多态前提（纯面向对象中）；
- 多态：不同的对象在执行时表现出不同的形态；

JavaScript原型链

- 从一个对象上获取属性，如果在当前对象中没有获取到就会去它的原型上面获取

```

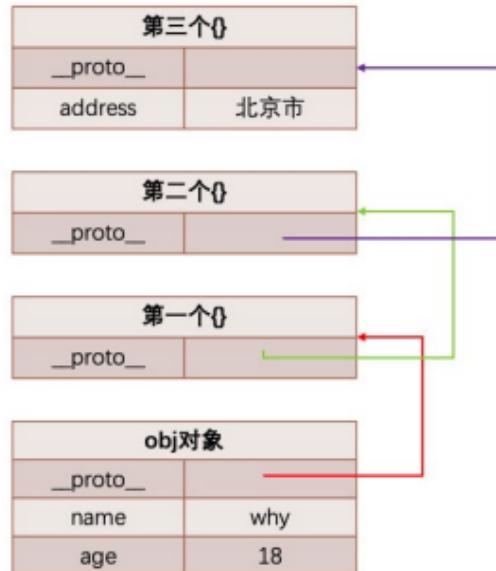
var obj = {
  name: "why",
  age: 18
}

obj.__proto__ = {}

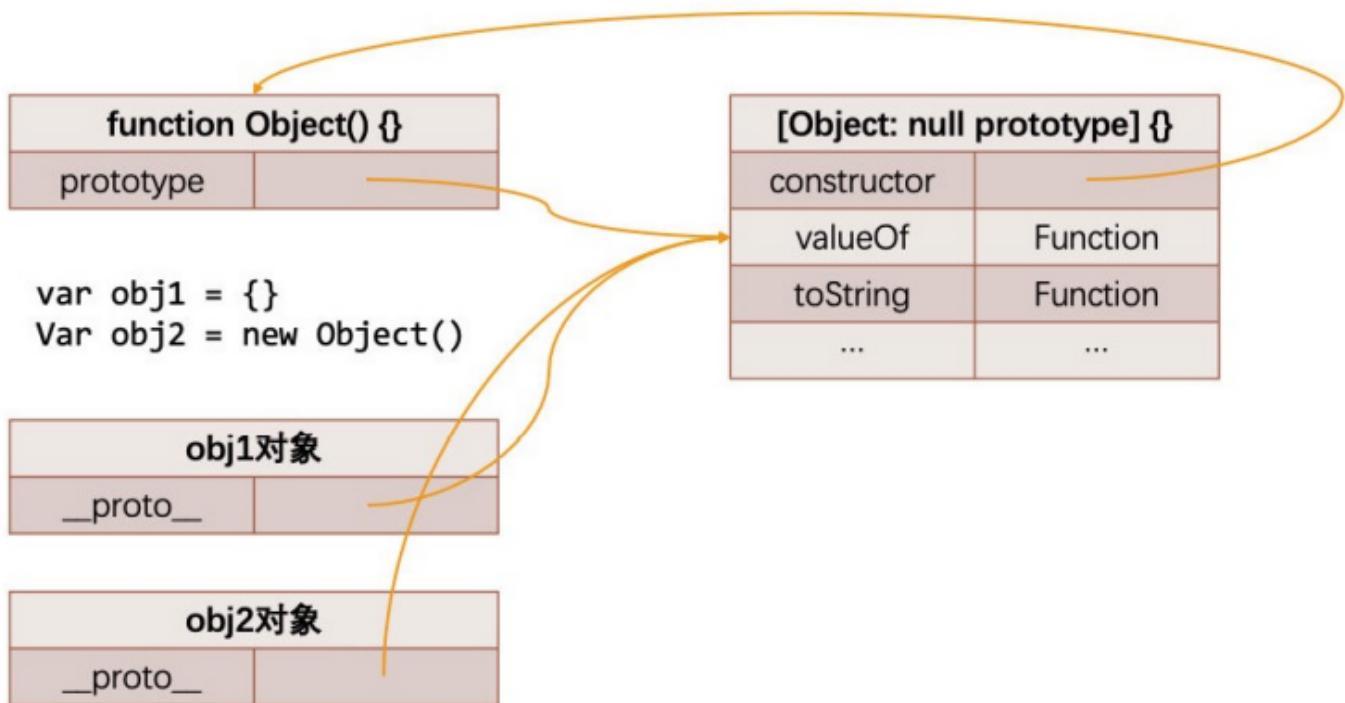
obj.__proto__.__proto__ = {}

obj.__proto__.__proto__.__proto__ = {
  address: "北京市"
}

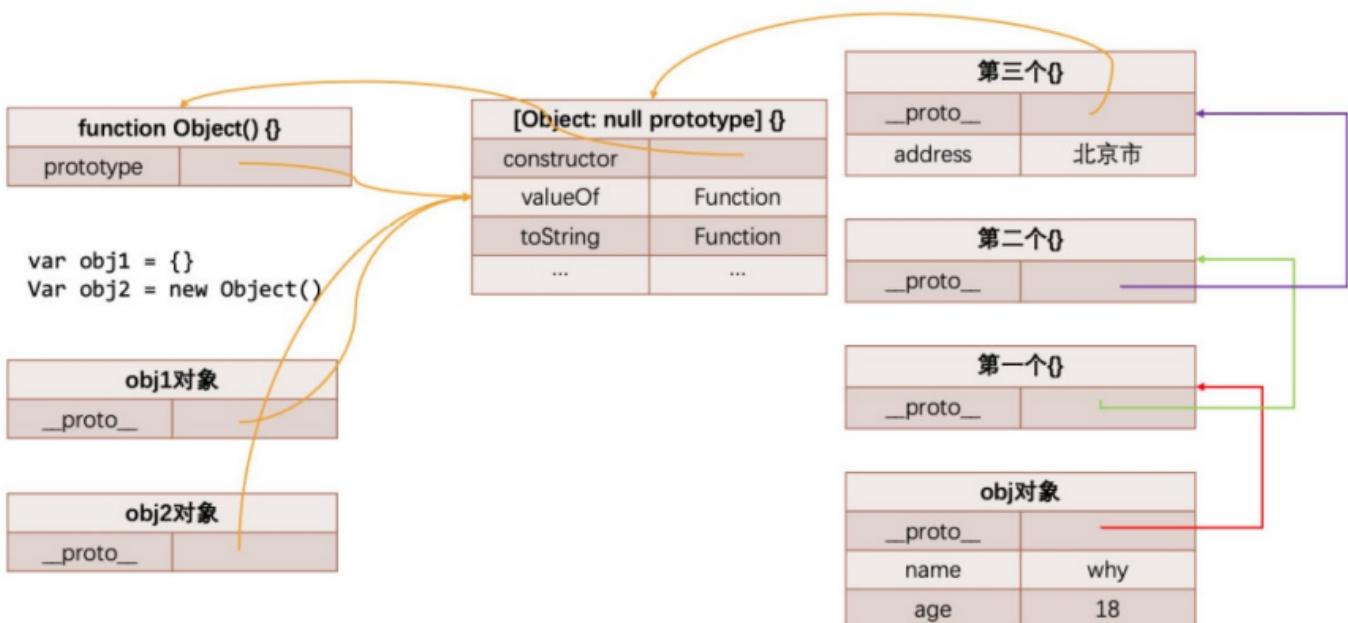
```



- Object的原型
- [Object: null prototype] {} 原型特殊的地方：
 - 特殊一：该对象有原型属性，但是它的原型属性已经指向的是 null，也就是已经是顶层原型了
 - 特殊二：该对象上有很多默认的属性和方法
- 创建Object对象的内存图



- 原型链关系的内存图



- Object是所有类的父类

```

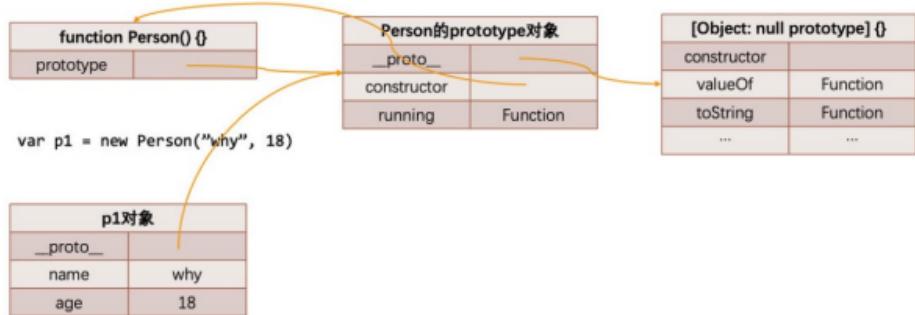
function Person(name, age) {
  this.name = name
  this.age = age
}

Person.prototype.running = function() {
  console.log(this.name + "running~")
}

var p1 = new Person("why", 18)

console.log(p1)
console.log(p1.valueOf())
console.log(p1.toString())

```



通过原型链实现继承

- 如果我们现在需要实现继承，那么就可以利用原型链来实现了：
 - 目前stu的原型是p对象，而p对象的原型是Person默认的原型，里面包含running等函数；
 - 注意：步骤4和步骤5不可以调整顺序，否则会有问题

```
// 1. 定义父类构造函数：公共属性和方法
function Person() {
  this.name = "why"
  this.friends = []
}

// 2. 父类原型上添加内容
Person.prototype.eating = function() {
  console.log(this.name + " eating~")
}

// 3. 定义子类构造函数：特有属性和方法
function Student() {
  this.sno = 111
}

// 4. 创建父类对象 作为子类的原型对象
var p = new Person()
Student.prototype = p

// 5. 在子类原型上添加内容
Student.prototype.studying = function() {
  console.log(this.name + " studying~")
}

// name/sno
var stu = new Student()

// console.log(stu.name)
// stu.eating()

// stu.studying()

// 原型链实现继承的弊端：
// 1.第一个弊端：打印stu对象，继承的属性是看不到的
console.log(stu.name)

// 2.第二个弊端：创建出来两个stu的对象
var stu1 = new Student()
var stu2 = new Student()

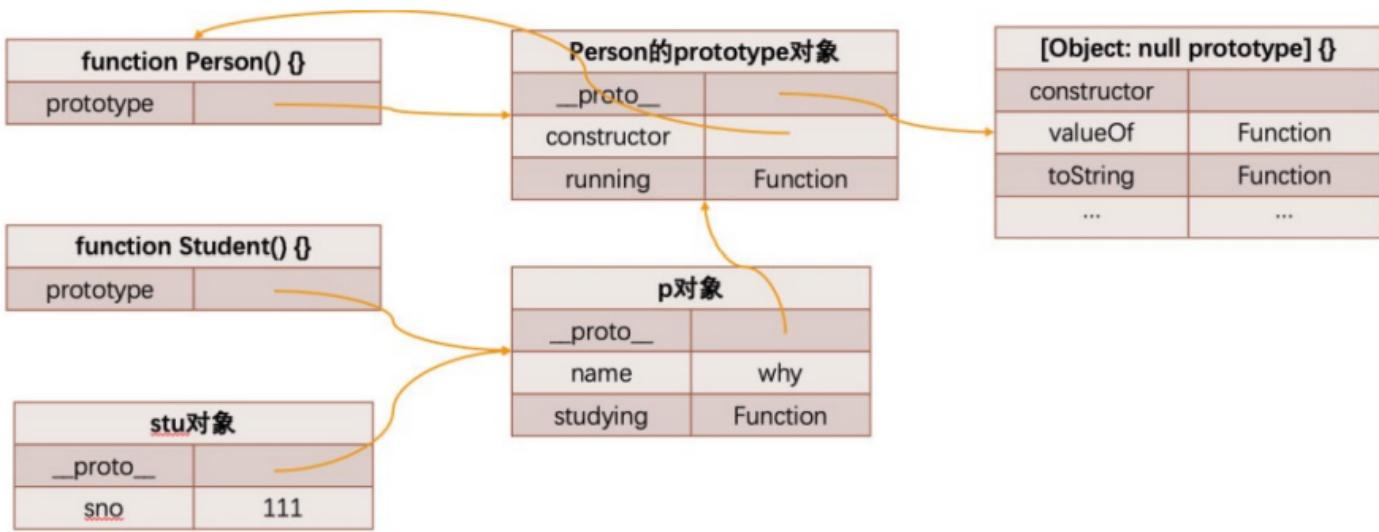
// 直接修改对象上的属性，是给本对象添加了一个新属性
stu1.name = "kobe"
console.log(stu2.name)

// 获取引用，修改引用中的值，会相互影响
stu1.friends.push("kobe")

console.log(stu1.friends)
console.log(stu2.friends)
```

```
// 3.第三个弊端：在前面实现类的过程中都没有传递参数
var stu3 = new Student("lilei", 112)
```

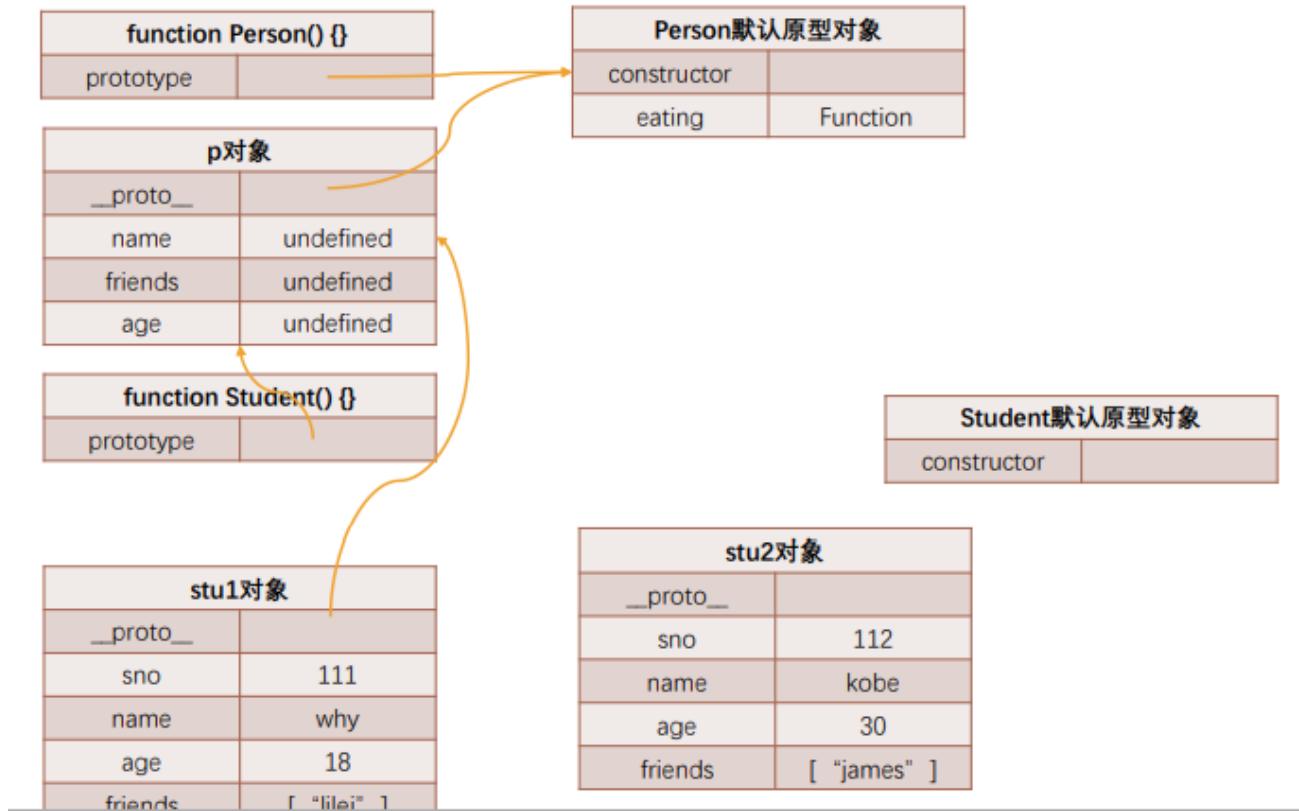
- 继承创建子类的内存图



- 原型链继承的弊端：某些属性其实是保存在p对象上的；
 1. 我们通过直接打印对象是看不到这个属性的；
 2. 这个属性会被多个对象共享，如果这个对象是一个引用类型，那么就会造成问题；
 3. 不能给Person传递参数，因为这个对象是一次性创建的（没办法定制化）

借用构造函数继承

- 借用继承的做法非常简单：在子类型构造函数的内部调用父类型构造函数。
 - 因为函数可以在任意的时刻被调用；
 - 因此通过apply()和call()方法也可以在新创建的对象上执行构造函数



- 组合借用继承的问题

- 组合继承最大的问题就是无论在什么情况下，都会调用两次父类构造函数。
 - 一次在创建子类原型的时候
 - 另一次在子类构造函数内部(也就是每次创建子类实例的时候)
- 所有的子类实例事实上会拥有两份父类的属性
 - 一份在当前的实例自己里面(也就是person本身的)，另一份在子类对应的原型对象中(也就是 `person.__proto__` 里面)

原型式继承函数

JavaScript想实现继承的目的：重复利用另外一个对象的属性和方法

```

var obj = {
  name: "why",
  age: 18
}

var info = Object.create(obj)

// 原型式继承函数
function createObject1(o) {
  var newObj = {}
  Object.setPrototypeOf(newObj, o)
  return newObj
}

function createObject2(o) {
  function Fn() {}
  Fn.prototype = o
  var newObj = new Fn()
  return newObj
}

// var info = createObject2(obj)
var info = Object.create(obj)
console.log(info)
console.log(info.__proto__)

```

寄生式继承函数

寄生式继承的思路是结合原型类继承和工厂模式的一种方式
即创建一个封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再将这个对象返回

```

var personObj = {
  running: function() {
    console.log("running")
  }
}

function createStudent(name) {
  var stu = Object.create(personObj)
  stu.name = name
  stu.studying = function() {
    console.log("studying~")
  }
  return stu
}

var stuObj = createStudent("why")
var stuObj1 = createStudent("kobe")
var stuObj2 = createStudent("james")

```

寄生组合式继承

理想的组合继承

- 组合继承是比较理想的继承方式, 但是存在两个问题:
 - 问题一: 构造函数会被调用两次: 一次在创建子类型原型对象的时候, 一次在创建子类型实例的时候.
 - 问题二: 父类型中的属性会有两份: 一份在原型对象中, 一份在子类型实例中

寄生式继承将这两个问题给解决掉

- 当我们在子类型的构造函数中调用父类型 `.call(this, 参数)` 这个函数的时候, 就会将父类型中的属性和方法复制一份到了子类型中. 所以父类型本身里面的内容, 我们不再需要
- 这个时候, 我们还需要获取到一份父类型的原型对象中的属性和方法.
- 能不能直接让子类型的原型对象 = 父类型的原型对象呢?
- 不要这么做, 因为这么做意味着以后修改了子类型原型对象的某个引用类型的时候, 父类型原生对象的引用类型也会被修改.
- 我们使用前面的寄生式思想就可以了

```
// 定义 createObject 函数 返回 object 的实例
function createObject(o) {
  function Fn() {}
  Fn.prototype = o
  return new Fn()
}

/**
 * 继承式核心函数
 * @param SubType 子函数
 * @param SuperType 父函数
 */
function inheritPrototype(SubType, SuperType) {
  SubType.prototype = Object.create(SuperType.prototype)
//  SubType.prototype = createObject(SuperType.prototype)
  Object.defineProperty(SubType.prototype, "constructor", {
    enumerable: false,
    configurable: true,
    writable: true,
    value: SubType
  })
}

function Person(name, age, friends) {
  this.name = name
  this.age = age
  this.friends = friends
}

inheritPrototype(Student, Person)
```

对象的方法补充

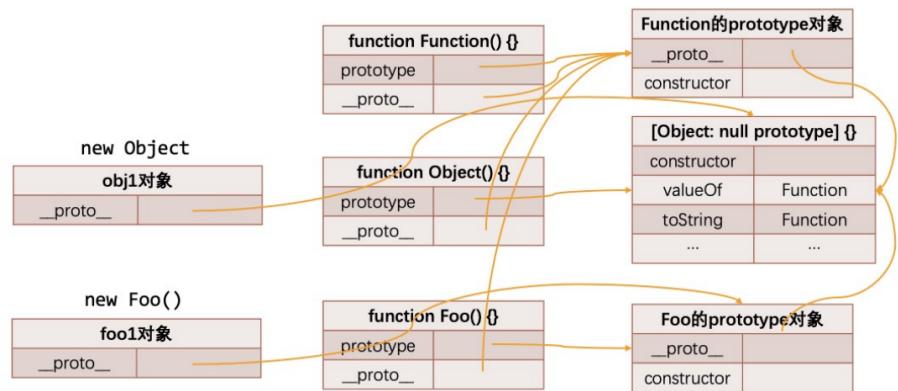
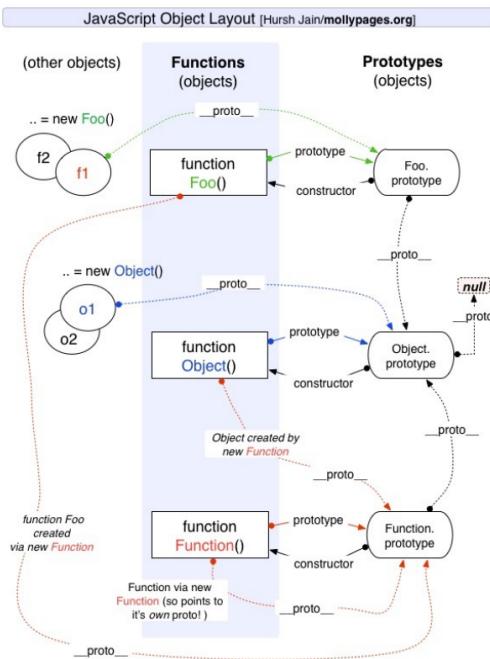
1. `hasOwnProperty` : 对象是否有某一个属于自己的属性 (不是在原型上的属性)
2. `in / for in` 操作符: 判断某个属性是否在某个对象或者对象的原型上
3. `instanceof` : 用于检测构造函数的 `prototype`, 是否出现在某个实例对象的原型链上
4. `isPrototypeOf` : 用于检测某个对象, 是否出现在某个实例对象的原型链上

原型继承关系

`function Function(){} 的 prototype 和 __proto__ 都是 Function`

proto => 因为函数也是对象 指向对象

`prototype =>` 因为是一个函数 所以有 `prototype` 属性 指向 Function



类

- 类的构造函数
- 当我们通过 `new` 关键字操作类的时候, 会调用这个 `constructor` 函数, 并且执行如下操作:

1. 在内存中创建一个新的对象 (空对象)
2. 这个对象内部的 `[[prototype]]` 属性会被赋值为该类的 `prototype` 属性
3. 构造函数内部的 `this`, 会指向创建出来的新对象
4. 执行构造函数的内部代码 (函数体代码)
5. 如果构造函数没有返回非空对象, 则返回创建出来的新对象

• 类的实例方法

- 对于实例的方法, 我们是希望放到原型上的, 这样可以被多个实例来共享
- 类中定义的方法, 直接放在原型上

```

class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
    this._address = "广州市"
  }

  // 普通的实例方法
  // 创建出来的对象进行访问
  // var p = new Person()
  // p.eating()
  eating() {
    console.log(this.name + " eating~")
  }

  running() {
    console.log(this.name + " running~")
  }
}

```

- 类的访问器方法

- 讲对象的属性描述符时有讲过对象可以添加 `setter` 和 `getter` 函数的，那么类也是可以的

```

class Person {
  // 类的访问器方法
  get address() {
    console.log("拦截访问操作")
    return this._address
  }

  set address(newAddress) {
    console.log("拦截设置操作")
    this._address = newAddress
  }
}

```

- 类的静态方法

- 静态方法通常用于定义直接使用类来执行的方法，不需要有类的实例，使用`static`关键字来定义

```

class Person {
  // 类的静态方法(类方法)
  // Person.createPerson()
  static randomPerson() {
    var nameIndex = Math.floor(Math.random() * names.length)
    var name = names[nameIndex]
    var age = Math.floor(Math.random() * 100)
    return new Person(name, age)
  }
}

```

- **super关键字：**
 - 注意：在子（派生）类的构造函数中使用this或者返回默认对象之前，必须先通过super调用父类的构造函数！
 - super的使用位置有三个：子类的构造函数、实例方法、静态方法

```
// 调用 父对象/父类 的构造函数  
super([arguments])  
  
// 调用 父对象/父类 的方法  
super.functionOnparent([arguments])
```

继承内置类

```
class HYArray extends Array {  
    firstItem() {  
        return this[0]  
    }  
  
    lastItem() {  
        return this[this.length-1]  
    }  
}  
  
var arr = new HYArray(1, 2, 3)  
console.log(arr.firstItem())  
console.log(arr.lastItem())
```

类的混入 mixin

- JavaScript的类只支持单继承：也就是只能有一个父类
 - 在开发中我们需要在一个类中添加更多相似的功能时，可以使用混入（mixin）

```
class Person {  
}  
  
function mixinRunner(BaseClass) {  
    class NewClass extends BaseClass {  
        running() {  
            console.log("running~")  
        }  
    }  
    return NewClass  
}  
  
function mixinEater(BaseClass) {  
    return class extends BaseClass {  
        eating() {  
            console.log("eating~")  
        }  
    }  
}  
  
// 在JS中类只能有一个父类：单继承  
class Student extends Person {  
}  
  
var NewStudent = mixinEater(mixinRunner(Student))  
var ns = new NewStudent()  
ns.running()  
ns.eating()
```

阅读源码

1. ES6转ES5的代码

```
class Person {  
    constructor(name, age) {  
        this.name = name  
        this.age = age  
    }  
  
    eating() {  
        console.log(this.name + " eating~")  
    }  
}
```

```

// babel转换
"use strict";

function _classCallCheck(instance, Constructor) {
    if (!(instance instanceof Constructor)) {
        throw new TypeError("Cannot call a class as a function");
    }
}

function _defineProperties(target, props) {
    for (var i = 0; i < props.length; i++) {
        var descriptor = props[i];
        descriptor.enumerable = descriptor.enumerable || false;
        descriptor.configurable = true;
        if ("value" in descriptor) descriptor.writable = true;
        Object.defineProperty(target, descriptor.key, descriptor);
    }
}

function _createClass(Constructor, protoProps, staticProps) {
    if (protoProps) _defineProperties(Constructor.prototype, protoProps);
    if (staticProps) _defineProperties(Constructor, staticProps);
    return Constructor;
}

// /*#__PURE__*/ 纯函数
// webpack 压缩 tree-shaking
// 这个函数没副作用
var Person = /*#__PURE__*/ (function () {
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }

    _createClass(Person, [
        {
            key: "eating",
            value: function eating() {
                console.log(this.name + " eating~");
            }
        }
    ]);
}

return Person;
})();

```

2. ES6转ES5的继承

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

    running() {
        console.log(this.name + " running~")
    }

    static staticMethod() {

    }
}

class Student extends Person {
    constructor(name, age, sno) {
        super(name, age)
        this.sno = sno
    }

    studying() {
        console.log(this.name + " studying~")
    }
}
```

```
// babel转换后的代码
"use strict";

function _typeof(obj) {
  "@babel/helpers - typeof";
  if (typeof Symbol === "function" && typeof Symbol.iterator === "symbol") {
    _typeof = function _typeof(obj) {
      return typeof obj;
    };
  } else {
    _typeof = function _typeof(obj) {
      return obj &&
        typeof Symbol === "function" &&
        obj.constructor === Symbol &&
        obj !== Symbol.prototype
      ? "symbol"
      : typeof obj;
    };
  }
  return _typeof(obj);
}

function _inherits(subClass, superClass) {
  if (typeof superClass !== "function" && superClass !== null) {
    throw new TypeError("Super expression must either be null or a function");
  }
  subClass.prototype = Object.create(superClass && superClass.prototype, {
    constructor: { value: subClass, writable: true, configurable: true }
  });
  // 目的静态方法的继承
  // Student.__proto__ = Person
  if (superClass) _setPrototypeOf(subClass, superClass);
}

// o: Student
// p: Person
// 静态方法的继承
function _setPrototypeOf(o, p) {
  _setPrototypeOf =
    Object.setPrototypeOf ||
    function _setPrototypeOf(o, p) {
      o.__proto__ = p;
      return o;
    };
  return _setPrototypeOf(o, p);
}

function _createSuper(Derived) {
  var hasNativeReflectConstruct = _isNativeReflectConstruct();
  return function _createSuperInternal() {
    var Super = _getPrototypeOf(Derived),
      result;
    if (hasNativeReflectConstruct) {
      var NewTarget = _getPrototypeOf(this).constructor;
```

```

        result = Reflect.construct(Super, arguments, NewTarget);
    } else {
        result = Super.apply(this, arguments);
    }
    return _possibleConstructorReturn(this, result);
};

}

function _possibleConstructorReturn(self, call) {
    if (call && (_typeof(call) === "object" || typeof call === "function")) {
        return call;
    } else if (call !== void 0) {
        throw new TypeError(
            "Derived constructors may only return object or undefined"
        );
    }
    return _assertThisInitialized(self);
}

function _assertThisInitialized(self) {
    if (self === void 0) {
        throw new ReferenceError(
            "this hasn't been initialised - super() hasn't been called"
        );
    }
    return self;
}

function _isNativeReflectConstruct() {
    if (typeof Reflect === "undefined" || !Reflect.construct) return false;
    if (Reflect.construct.sham) return false;
    if (typeof Proxy === "function") return true;
    try {
        Boolean.prototype.valueOf.call(
            Reflect.construct(Boolean, [], function () {})
        );
        return true;
    } catch (e) {
        return false;
    }
}

function _getPrototypeOf(o) {
    _getPrototypeOf = Object.setPrototypeOf
        ? Object.getPrototypeOf
        : function _getPrototypeOf(o) {
            return o.__proto__ || Object.getPrototypeOf(o);
        };
    return _getPrototypeOf(o);
}

function _classCallCheck(instance, Constructor) {
    if (!(instance instanceof Constructor)) {
        throw new TypeError("Cannot call a class as a function");
    }
}

```

```
        }
    }

    function _defineProperties(target, props) {
        for (var i = 0; i < props.length; i++) {
            var descriptor = props[i];
            descriptor.enumerable = descriptor.enumerable || false;
            descriptor.configurable = true;
            if ("value" in descriptor) descriptor.writable = true;
            Object.defineProperty(target, descriptor.key, descriptor);
        }
    }

    function _createClass(Constructor, protoProps, staticProps) {
        if (protoProps) _defineProperties(Constructor.prototype, protoProps);
        if (staticProps) _defineProperties(Constructor, staticProps);
        return Constructor;
    }

    var Person = /*#__PURE__*/ (function () {
        function Person(name, age) {
            _classCallCheck(this, Person);

            this.name = name;
            this.age = age;
        }

        _createClass(Person, [
            {
                key: "running",
                value: function running() {
                    console.log(this.name + " running~");
                }
            }
        ]);
    });

    return Person;
})();

var Student = /*#__PURE__*/ (function (_Person) {
    // 实现之前的寄生式继承的方法(静态方法的继承)
    _inherits(Student, _Person);

    var _super = _createSuper(Student);

    function Student(name, age, sno) {
        var _this;

        _classCallCheck(this, Student);

        // Person不能被当成一个函数去调用
        // Person.call(this, name, age)

        debugger;
    }
})
```

```

// 创建出来的对象 {name: , age: }
_this = _super.call(this, name, age);
_this.sno = sno;
// {name: , age:, sno: }
return _this;
}

_createClass(Student, [
{
  key: "studying",
  value: function studying() {
    console.log(this.name + " studying~");
  }
}
]);
};

return Student;
})(Person);

```

```

var stu = new Student()

// Super: Person
// arguments: ["why", 18]
// NewTarget: Student
// 会通过Super创建出来一个实例，但是这个实例的原型constructor指向的是NewTarget
// 会通过Person创建出来一个实例，但是这个实例的原型constructor指向的Person
Reflect.construct(Super, arguments, NewTarget);

```

JavaScript中的多态

- 面向对象的三大特性：封装、继承、多态
- 维基百科对多态的定义：多态（英语：polymorphism）指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。
- 总结：不同的数据类型进行同一个操作，表现出不同的行为，就是多态的体现。
- 那么从上面的定义来看，JavaScript是一定存在多态的

```

// 多态：当对不同的数据类型执行同一个操作时，如果表现出来的行为(形态)不一样，那么就是多态的体现。
function calcArea(foo) {
  console.log(foo.getArea())
}

var obj1 = {
  name: "why",
  getArea: function() {
    return 1000
  }
}

class Person {
  getArea() {
    return 100
  }
}

var p = new Person()

calcArea(obj1)
calcArea(p)

// 也是多态的体现
function sum(m, n) {
  return m + n
}

sum(20, 30)
sum("abc", "cba")

```

ES6-ES12新增

ES6

字面量的增强

字面量的增强主要包括下面几部分：

- 属性的简写：Property Shorthand
- 方法的简写：Method Shorthand
- 计算属性名：Computed Property Names

```
var name = "why"
var age = 18

var obj = {
  // 1.property shorthand(属性的简写)
  name,
  age,

  // 2.method shorthand(方法的简写)
  foo: function() {
    console.log(this)
  },
  bar() {
    console.log(this)
  },
  baz: () => {
    console.log(this)
  },

  // 3.computed property name(计算属性名)
  [name + 123]: 'hehehehe'
}
```

解构 Destructuring

ES6中新增了一个从数组或对象中方便获取数据的方法，称之为解构Destructuring。

- 我们可以划分为：数组的解构和对象的解构。
- 数组的解构：
 - 基本解构过程
 - 顺序解构
 - 解构出数组
 - 默认值

```
var names = ["abc", "cba", "nba"]
// var item1 = names[0]
// var item2 = names[1]
// var item3 = names[2]

// 对数组的解构: []
var [item1, item2, item3] = names
console.log(item1, item2, item3)

// 解构后面的元素
var [, , itemz] = names
console.log(itemz)

// 解构出一个元素,后面的元素放到一个新数组中
var [itemx, ...newNames] = names
console.log(itemx, newNames)

// 解构的默认值
var [itema, itemb, itemc, itemd = "aaa"] = names
console.log(itemd)
```

- 对象的解构:

- 基本解构过程
- 任意顺序
- 重命名
- 默认值

```

var obj = {
  name: "why",
  age: 18,
  height: 1.88
}

// 对象的解构: {}
var { name, age, height } = obj
console.log(name, age, height)

var { age } = obj
console.log(age)

var { name: newName } = obj
console.log(newName)

var { address: newAddress = "广州市" } = obj
console.log(newAddress)

function foo(info) {
  console.log(info.name, info.age)
}

foo(obj)

function bar({name, age}) {
  console.log(name, age)
}

bar(obj)

```

let/const

- **const** 关键字:
 - **const** 关键字是 **constant** 的单词的缩写，表示常量、衡量的意思；
 - 它表示保存的数据一旦被赋值，就不能被修改；
 - 但是如果赋值的是引用类型，那么可以通过引用找到对应的对象，修改对象的内容；
- **let/const** 作用域提升

```
console.log(foo) // ReferenceError: Cannot access 'foo' before initialization
```

```
let foo = "foo"
```

- 这些变量会被创建在包含他们的词法环境被实例化时，但是是不可以访问它们的，直到词法绑定被求值
- 在执行上下文的词法环境创建出来的时候，变量事实上已经被创建了，只是这个变量是不能被访问的
- **let**、**const** 没有进行作用域提升，但是会在解析阶段被创建出来

- Window 对象添加属性
 - 在全局通过 var 来声明一个变量，事实上会在 window 上添加一个属性
 - let、const 不会给 window 上添加任何属性
 - V8 中，变量被保存到 VariableMap 中
 - window 对象是早期的 GO 对象，在最新的实现中其实是浏览器添加的全局对象，并且一直保持了 window 和 var 之间值的相等性
- let/const 的块级作用域
 - 在 ES6 中新增了块级作用域，并且通过 let、const、function、class 声明的标识符是具备块级作用域的限制的：
 - 但是我们会发现函数拥有块级作用域，但是外面依然可以访问的：
 - 这是因为引擎会对函数的声明进行特殊的处理，允许像 var 那样进行提升
- 暂时性死区
 - 在一个代码中，使用 let、const 声明的变量，在声明之前，变量都是不可以访问的

函数默认值

- 参数的默认值我们通常会将其放到最后
- 外默认值会改变函数的 length 的个数，默认值以及后面的参数都不计算在 length 之内了

函数的剩余参数

- 如果最后一个参数是 ... 为前缀的，那么它会将剩余的参数放到该参数中，并且作为一个数组
- 剩余参数和 arguments 的区别
 - 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参
 - arguments 对象不是一个真正的数组，而 rest 参数是一个真正的数组，可以进行数组的所有操作
 - arguments 是早期的 ECMAScript 中为了方便去获取所有的参数提供的一个数据结构，而 rest 参数是 ES6 中提供并且希望以此来替代 arguments 的
- 剩余参数必须放到最后一个位置，否则会报错

展开语法

- 展开语法(Spread syntax):
 - 可以在函数调用/数组构造时，将数组表达式或者 string 在语法层面展开；
 - 还可以在构造字面量对象时，将对象表达式按 key-value 的方式展开
- 展开语法的场景：
 - 在函数调用时使用；
 - 在数组构造时使用；
 - 在构建对象字面量时，也可以使用展开运算符，这个是在 ES2018 (ES9) 中添加的新特性
- 注意：展开运算符其实是一种浅拷贝

数值的表示

- 在 ES6 中规范了二进制和八进制的写法

```
const num1 = 100 // 十进制  
  
// b -> binary  
const num2 = 0b100 // 二进制  
// o -> octonary  
const num3 = 0o100 // 八进制  
// x -> hexadecimal  
const num4 = 0x100 // 十六进制
```

- 在ES2021新增特性：数字过长时，可以使用_作为连接符

```
// 大的数值的连接符(ES2021 ES12)  
const num = 10_000_000_000_000_000
```

Symbol

Symbol 是ES6中新增的一个基本数据类型，翻译为符号

用来生成一个独一无二的值

- 为什么需要 Symbol
 - 在ES6之前，对象的属性名都是字符串形式，那么很容易造成属性名的冲突
 - 比如原来有一个对象，我们希望在其中添加一个新的属性和值，但是我们在不确定它原来内部有什么内容的情况下，很容易造成冲突，从而覆盖掉它内部的某个属性
- Symbol 值是通过 Symbol 函数来生成的，生成后可以作为属性名
 - 也就是在ES6中，对象的属性名可以使用字符串，也可以使用 Symbol 值
- Symbol 即使多次创建值，它们也是不同的
- 我们也可以在创建 Symbol 值的时候传入一个描述 description：这个是ES2019 (ES10) 新增的特性
- 想创建相同的Symbol
 - 可以使用 Symbol.for 方法
 - 可以通过 Symbol.keyFor 方法来获取对应的 key

```

// 3.Symbol值作为key
// 3.1.在定义对象字面量时使用
const obj = {
  [s1]: "abc",
  [s2]: "cba"
}

// 3.2.新增属性
obj[s3] = "nba"

// 3.3.Object.defineProperty方式
const s4 = Symbol()
Object.defineProperty(obj, s4, {
  enumerable: true,
  configurable: true,
  writable: true,
  value: "mba"
})

console.log(obj[s1], obj[s2], obj[s3], obj[s4])
// 注意：不能通过.语法获取
// console.log(obj.s1)

// 4.使用Symbol作为key的属性名，在遍历/Object.keys等中是获取不到这些Symbol值
// 需要Object.getOwnPropertySymbols来获取所有Symbol的key
console.log(Object.keys(obj))
console.log(Object.getOwnPropertyNames(obj))
console.log(Object.getOwnPropertySymbols(obj))
const sKeys = Object.getOwnPropertySymbols(obj)
for (const sKey of sKeys) {
  console.log(obj[sKey])
}

// 5.Symbol.for(key)/Symbol.keyFor(symbol)
const sa = Symbol.for("aaa")
const sb = Symbol.for("aaa")
console.log(sa === sb)

const key = Symbol.keyFor(sa)
console.log(key)
const sc = Symbol.for(key)
console.log(sa === sc)

```

Set的常见方法

- Set常见的属性：
 - size：返回Set中元素的个数
- Set常用的方法：
 - add(value)：添加某个元素，返回Set对象本身
 - delete(value)：从set中删除和这个值相等的元素，返回boolean类型
 - has(value)：判断set中是否存在某个元素，返回boolean类型

- `clear()`: 清空set中所有的元素，没有返回值
- `forEach(callback, [, thisArg])`: 通过`forEach`遍历set
- 另外Set是支持for of的遍历的

WeakSet使用

- 和Set的区别
 - 区别一： `WeakSet` 中只能存放对象类型，不能存放基本数据类型
 - 区别二： `WeakSet` 对元素的引用是弱引用，如果没有其他引用对某个对象进行引用，那么GC可以对该对象进行回收
- `WeakSet` 常见的方法：
 - `add(value)`：添加某个元素，返回 `WeakSet` 对象本身
 - `delete(value)`：从 `WeakSet` 中删除和这个值相等的元素，返回 `boolean` 类型
 - `has(value)`：判断 `WeakSet` 中是否存在某个元素，返回 `boolean` 类型
- `WeakSet` 不能遍历
 - 因为 `WeakSet` 只是对对象的弱引用，如果我们遍历获取到其中的元素，那么有可能造成对象不能正常销毁。
 - 所以存储到 `WeakSet` 中的对象是没办法获取的

```
// WeakSet的应用场景
const personSet = new WeakSet()
class Person {
  constructor() {
    personSet.add(this)
  }

  running() {
    if (!personSet.has(this)) {
      throw new Error("不能通过非构造方法创建出来的对象调用running方法")
    }
    console.log("running~", this)
  }
}

let p = new Person()
p.running()
p = null

p.running.call({name: "why"})
```

Map的基本使用

- 对象存储映射关系只能用字符串（ES6新增了Symbol）作为属性名（key）
- 某些情况下我们可能希望通过其他类型作为 key，比如对象，这个时候会自动将对象转成字符串来作为 key
- Map 常见的属性：

- `size` : 返回 Map 中元素的个数
- Map常见的方法:
 - `set(key, value)` : 在 Map 中添加 key 、 value , 并且返回整个 Map 对象
 - `get(key)` : 根据 key 获取 Map 中的 value
 - `has(key)` : 判断是否包括某一个 key , 返回 Boolean 类型
 - `delete(key)` : 根据key删除一个键值对, 返回 Boolean 类型
 - `clear()` : 清空所有的元素
 - `forEach(callback, [, thisArg])` : 通过 forEach 遍历 Map
- Map 也可以通过 `for of` 进行遍历

WeakMap的使用

- 和Map的区别
 - 区别一: WeakMap的key只能使用对象, 不接受其他的类型作为key
 - 区别二: WeakMap的key对对象想的引用是弱引用, 如果没有其他引用引用这个对象, 那么GC可以回收该对象
- WeakMap常见的方法有四个:
 - `set(key, value)` : 在Map中添加 key 、 value , 并且返回整个Map对象
 - `get(key)` : 根据key获取Map中的 value
 - `has(key)` : 判断是否包括某一个key, 返回 Boolean 类型
 - `delete(key)` : 根据key删除一个键值对, 返回 Boolean 类型
- WeakMap的应用
 - WeakMap也是不能遍历的
 - 因为没有forEach方法, 也不支持通过for of的方式进行遍历

```
// 应用场景(vue3响应式原理)
const obj1 = {
  name: "why",
  age: 18
}

function obj1NameFn1() {
  console.log("obj1NameFn1被执行")
}

function obj1NameFn2() {
  console.log("obj1NameFn2被执行")
}

function obj1AgeFn1() {
  console.log("obj1AgeFn1")
}

function obj1AgeFn2() {
  console.log("obj1AgeFn2")
}

const obj2 = {
  name: "kobe",
  height: 1.88,
  address: "广州市"
}

function obj2NameFn1() {
  console.log("obj2NameFn1被执行")
}

function obj2NameFn2() {
  console.log("obj2NameFn2被执行")
}

// 1.创建WeakMap
const weakMap = new WeakMap()

// 2.收集依赖结构
// 2.1.对obj1收集的数据结构
const obj1Map = new Map()
obj1Map.set("name", [obj1NameFn1, obj1NameFn2])
obj1Map.set("age", [obj1AgeFn1, obj1AgeFn2])
weakMap.set(obj1, obj1Map)

// 2.2.对obj2收集的数据结构
const obj2Map = new Map()
obj2Map.set("name", [obj2NameFn1, obj2NameFn2])
weakMap.set(obj2, obj2Map)

// 3.如果obj1.name发生了改变
// Proxy/Object.defineProperty
obj1.name = "james"
```

```
const targetMap = weakMap.get(obj1)
const fns = targetMap.get("name")
fns.forEach(item => item())
```

ES7

Array Includes

- 在ES7之前，如果我们想判断一个数组中是否包含某个元素，需要通过 `indexof` 获取结果，并且判断是否为 -1
- 在ES7中，我们可以通过 `includes` 来判断一个数组中是否包含一个指定的元素，根据情况，如果包含则返回 `true`，否则返回 `false`

指数(乘方) exponentiation运算符

- 在ES7之前，计算数字的乘方需要通过 `Math.pow` 方法来完成
- 在ES7中，增加了 `**` 运算符，可以对数字来计算乘方

ES8

Object values

- 之前我们可以通过 `Object.keys` 获取一个对象所有的 key
- 在ES8中提供了 `Object.values` 来获取所有的 value 值

Object entries

- 通过 `Object.entries` 可以获取到一个数组，数组中会存放可枚举属性的键值对数组

```
const obj = {
  name: "why",
  age: 18
}

console.log(Object.entries(obj)) // [ [ 'name', 'why' ], [ 'age', 18 ] ]
const objEntries = Object.entries(obj)
objEntries.forEach(item => {
  console.log(item[0], item[1]) // name why // age 18
})

console.log(Object.entries(["abc", "cba", "nba"])) // [ [ '0', 'abc' ], [ '1', 'cba' ], [ '2', 'nba' ] ]
console.log(Object.entries("abc")) // [ [ '0', 'a' ], [ '1', 'b' ], [ '2', 'c' ] ]
```

String Padding

- 某些字符串我们需要对其进行前后的填充，来实现某种格式化效果，ES8中增加了 `padStart` 和 `padEnd` 方法，分别是对字符串的首尾进行填充的

```
const message = "Hello World"

const newMessage = message.padStart(15, "*").padEnd(20, "-")
console.log(newMessage) // ****Hello World----
```

- 应用场景

```
const cardNumber = "321324234242342342341312"
const lastFourCard = cardNumber.slice(-4)
const finalCard = lastFourCard.padStart(cardNumber.length, "*")
console.log(finalCard) // *****1312
```

Trailing Commas

- 在ES8中，我们允许在函数定义和调用时多加一个逗号

```
function foo(m, n,) {
}

foo(20, 30,)
```

Object Descriptors

ES8中增加了另一个对对象的操作是 `Object.getOwnPropertyDescriptors`

`Object.getOwnPropertyDescriptors(obj)` 方法用来获取一个对象的所有自身属性的描述符

- 浅拷贝一个对象
 - `Object.assign()` 方法将所有可枚举（`Object.propertyIsEnumerable()` 返回 `true`）和自有（`Object.hasOwnProperty()` 返回 `true`）属性从一个或多个源对象复制到目标对象，返回修改后的对象。

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

- 针对深拷贝，需要使用其他办法，因为 `Object.assign()` 只复制属性值。

- 假如源对象是一个对象的引用，它仅仅会复制其引用值
- 原型链上的属性和不可枚举属性不能被复制
- `Object.assign()` 方法只能拷贝源对象的可枚举的自身属性，同时拷贝时无法拷贝属性的特性们，而且访问器属性会被转换成数据属性，也无法拷贝源对象的原型，该方法配合 `Object.create()` 方法可以实现上面说的这些

```
Object.create(
  Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj)
)
```

- 创建子类
 - 创建子类的典型方法是定义子类，将其原型设置为超类的实例，然后在该实例上定义属性。这么写很不优雅，特别是对于 getters 和 setter 而言。相反，您可以使用此代码设置原型：

```
function superclass() {}
superclass.prototype = {
  // 在这里定义方法和属性
};

function subclass() {}
subclass.prototype = Object.create(superclass.prototype, Object.getOwnPropertyDescriptors({
  // 在这里定义方法和属性
}));
```

ES10

flat flatMap

- `flat()` 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回
- `flatMap()` 方法首先使用映射函数映射每个元素，然后将结果压缩成一个新数组
 - 注意一： `flatMap` 是先进行 `map` 操作，再做 `flat` 的操作
 - 注意二： `flatMap` 中的 `flat` 相当于深度为 1

```

// 1.flat的使用
const nums = [10, 20, [2, 9], [[30, 40], [10, 45]], 78, [55, 88]]
const newNums = nums.flat()
console.log(newNums) // [ 10, 20, 2, 9, [ 30, 40 ], [ 10, 45 ], 78, 55, 88 ]

const newNums2 = nums.flat(2)
console.log(newNums2)
// [
//   10, 20, 2, 9, 30,
//   40, 10, 45, 78, 55,
//   88
// ]

```



```

// 2.flatMap的使用
const nums2 = [10, 20, 30]
const newNums3 = nums2.flatMap(item => {
  return item * 2
})
const newNums4 = nums2.map(item => {
  return item * 2
})

console.log(newNums3) // [ 20, 40, 60 ]
console.log(newNums4) // [ 20, 40, 60 ]

```



```

// 3.flatMap的应用场景
const messages = ["Hello World", "hello lyh", "my name is coderwhy"]
const words = messages.flatMap(item => {
  return item.split(" ")
})

console.log(words)
// [
//   'Hello', 'World',
//   'hello', 'lyh',
//   'my', 'name',
//   'is', 'coderwhy'
// ]

```

Object fromEntries

- 可以通过 `Object.entries` 将一个对象转换成 `entries`，那么如果我们有一个 `entries` 了，如何将其转换成对象呢
- ES10提供了 `Object.fromEntries` 来完成转换

```

const obj = {
  name: "why",
  age: 18,
  height: 1.88
}

const entries = Object.entries(obj)
console.log(entries)

const newObj = {}
for (const entry of entries) {
  newObj[entry[0]] = entry[1]
}

// 1. ES10中新增了Object.fromEntries方法
const newObj1 = Object.fromEntries(entries)

console.log(newObj1)

```

- 应用场景

```

// 2. Object.fromEntries的应用场景
const queryString = 'name=why&age=18&height=1.88'
const queryParams = new URLSearchParams(queryString)
for (const param of queryParams) {
  console.log(param)
}

const paramObj = Object.fromEntries(queryParams)
console.log(paramObj)

```

trimStart trimEnd

- 单独去除前面或者后面，ES10中给我们提供了 trimStart 和 trimEnd

ES11

BigInt

- 在早期的JavaScript中，我们不能正确的表示过大的数字：
 - 大于 MAX_SAFE_INTEGER 的数值，表示的可能是不正确的
- ES11中，引入了新的数据类型 BigInt，用于表示大的整数
 - BigInt 的表示方法是在数值的后面加上 n

```
// ES11之前 max_safe_integer
const maxInt = Number.MAX_SAFE_INTEGER
console.log(maxInt) // 9007199254740991
console.log(maxInt + 1)
console.log(maxInt + 2)

// ES11之后: BigInt
const bigInt = 900719925474099100n
console.log(bigInt + 10n)

const num = 100
console.log(bigInt + BigInt(num))

const smallNum = Number(bigInt)
console.log(smallNum)
```

Nullish Coalescing Operator

- Nullish Coalescing Operator 增加了空值合并操作符

```
// ES11: 空值合并运算 ???

const foo = undefined
// const bar = foo || "default value"
const bar = foo ?? "defualt value"

console.log(bar)
```

Optional Chaining

- 可选链也是ES11中新增一个特性，主要作用是让我们的代码在进行 `null` 和 `undefined` 判断时更加清晰和简洁

```
// ES11提供了可选链(Optional Chainling)
console.log(info.friend?.girlFriend?.name)
```

Global This

- 之前我们希望获取JavaScript环境的全局对象，不同的环境获取的方式是不一样的
 - 比如在浏览器中可以通过 `this`、`window` 来获取
 - 比如在Node中我们需要通过 `global` 来获取
- 那么在ES11中对获取全局对象进行了统一的规范： `globalThis`

```
// 获取某一个环境下的全局对象(Global Object)

// 在浏览器下
console.log(window)
console.log(this)

// 在node下
console.log(global)

// ES11
console.log(globalThis)
```

for..in标准化

for...in 是用于遍历对象的 key 的

```
// for...in 标准化: ECMA
const obj = {
  name: "why",
  age: 18
}

for (const item in obj) {
  console.log(item)
}
```

ES12

FinalizationRegistry

- FinalizationRegistry 对象可以让你在对象被垃圾回收时请求一个回调。
 - FinalizationRegistry 提供了这样的一种方法：当一个在注册表中注册的对象被回收时，请求在某个时间点上调用一个清理回调。（清理回调有时被称为 finalizer）；
 - 你可以通过调用 register 方法，注册任何你想要清理回调的对象，传入该对象和所含的值；

```
// ES12: FinalizationRegistry类
const finalRegistry = new FinalizationRegistry((value) => {
  console.log("注册在finalRegistry的对象，某一个被销毁", value)
})

let obj = { name: "why" }
let info = { age: 18 }

finalRegistry.register(obj, "obj")
finalRegistry.register(info, "value")

obj = null
info = null
```

WeakRefs

- 如果我们默认将一个对象赋值给另外一个引用，那么这个引用是一个强引用：
 - 如果我们希望是一个弱引用的话，可以使用 `WeakRef`

```
// ES12: WeakRef类
// WeakRef.prototype.deref:
// > 如果原对象没有销毁，那么可以获取到原对象
// > 如果原对象已经销毁，那么获取到的是undefined
const finalRegistry = new FinalizationRegistry((value) => {
  console.log("注册在finalRegistry的对象，某一个被销毁", value)
})

let obj = { name: "why" }
let info = new WeakRef(obj)

finalRegistry.register(obj, "obj")

obj = null

setTimeout(() => {
  console.log(info.deref()?.name) // why
  console.log(info.deref() && info.deref().name) // why
}, 1000)
```

logical assignment operators

```
// 1. ||= 逻辑或赋值运算
let message = "hello world"
message = message || "default value"
message ||= "default value"
console.log(message)
```

```

// 2.&&= 逻辑与赋值运算
// &&
const obj = {
  name: "why",
  foo: function() {
    console.log("foo函数被调用")
  }
}

// obj.foo && obj.foo()

// &&=
let info = {
  name: "why"
}

// 1.判断info
// 2.有值的情况下，取出info.name
info = info && info.name
info &&= info.name
console.log(info)

// 3.??= 逻辑空赋值运算
let message = 0
message ??= "default value"
console.log(message)

```

Proxy-Reflect vue2-vue3响应式原理

Proxy

- 监听对象的操作
 - 需求：有一个对象，我们希望监听这个对象中的属性被设置或获取的过程
 - 利用 `Object.defineProperty` 的存储属性描述符来对属性的操作进行监听
 - 缺点：`Object.defineProperty` 设计的初衷，不是为了去监听截止一个对象中所有的属性的，，如果我们想监听更加丰富的操作，比如新增属性、删除属性，那么 `Object.defineProperty` 是无能为力的

Proxy基本使用

- 在ES6中，新增了一个 `Proxy` 类，是用于帮助我们创建一个代理的：
 - 也就是说，如果我们希望监听一个对象的相关操作，那么我们可以先创建一个代理对象（`Proxy`对象）
 - 之后对该对象的所有操作，都通过代理对象来完成，代理对象可以监听我们想要对原对象进行哪些操作

- 可以将上面的案例用Proxy来实现一次：
 - 首先，我们需要 new Proxy对象，并且传入需要侦听的对象以及一个处理对象，可以称之为 handler

```
const p = new Proxy(target, handler)
```
 - 其次，我们之后的操作都是直接对Proxy的操作，而不是原有的对象，因为我们需要在 handler 里面进行侦听

Proxy的set和get捕获器

- 如果我们想要侦听某些具体的操作，那么就可以在 handler 中添加对应的捕捉器（Trap）
- set 和 get 分别对应的是函数类型；
 - set 函数有四个参数：
 - target : 目标对象（侦听的对象）
 - property : 将被设置的属性 key
 - value : 新属性值
 - receiver : 调用的代理对象
 - get 函数有三个参数：
 - target : 目标对象（侦听的对象）
 - property : 被获取的属性key
 - receiver : 调用的代理对象

```

const obj = {
  name: "why",
  age: 18
}

const objProxy = new Proxy(obj, {
  // 获取值时的捕获器
  get: function(target, key) {
    console.log(`监听到对象的${key}属性被访问了`, target)
    return target[key]
  },
  // 设置值时的捕获器
  set: function(target, key, newValue) {
    console.log(`监听到对象的${key}属性被设置值`, target)
    target[key] = newValue
  }
})

console.log(objProxy.name)
// why
// 监听到对象的name属性被访问了 { name: 'why', age: 18 }
console.log(objProxy.age)
// 监听到对象的age属性被访问了 { name: 'why', age: 18 }
// 18

objProxy.name = "kobe"
objProxy.age = 30

console.log(obj.name)
console.log(obj.age)
// 监听到对象的name属性被设置值 { name: 'why', age: 18 }
// 监听到对象的age属性被设置值 { name: 'kobe', age: 18 }
// kobe
// 30

```

Proxy所有捕获器

- `handler.getPrototypeOf()`
 - `Object.getPrototypeOf` 方法的捕捉器。
- `handler.setPrototypeOf()`
 - `Object.setPrototypeOf` 方法的捕捉器。
- `handler.isExtensible()`
 - `Object.isExtensible` 方法的捕捉器。
- `handler.preventExtensions()`
 - `Object.preventExtensions` 方法的捕捉器。
- `handler.getOwnPropertyDescriptor()`
 - `Object.getOwnPropertyDescriptor` 方法的捕捉器。
- `handler.defineProperty()`

- `Object.defineProperty` 方法的捕捉器。
- `handler.ownKeys()`
 - `Object.getOwnPropertyNames` 方法和 `Object.getOwnPropertySymbols` 方法的捕捉器。
- `handler.has()`
 - `in` 操作符的捕捉器。
- `handler.get()`
 - 属性读取操作的捕捉器。
- `handler.set()`
 - 属性设置操作的捕捉器。
- `handler.deleteProperty()`
 - `delete` 操作符的捕捉器。
- `handler.apply()`
 - 函数调用操作的捕捉器。
- `handler.construct()`
 - `new` 操作符的捕捉器。

```

const obj = {
  name: "why", // 数据属性描述符
  age: 18
}

// 变成一个访问属性描述符
// Object.defineProperty(obj, "name", {
//   })

const objProxy = new Proxy(obj, {
  // 获取值时的捕获器
  get: function(target, key) {
    console.log(`监听到对象的${key}属性被访问了`, target)
    return target[key]
  },

  // 设置值时的捕获器
  set: function(target, key, newValue) {
    console.log(`监听到对象的${key}属性被设置值`, target)
    target[key] = newValue
  },

  // 监听in的捕获器
  has: function(target, key) {
    console.log(`监听到对象的${key}属性in操作`, target)
    return key in target
  },

  // 监听delete的捕获器
  deleteProperty: function(target, key) {
    console.log(`监听到对象的${key}属性in操作`, target)
    delete target[key]
  }
})

// in操作符
console.log("name" in objProxy)

// delete操作
delete objProxy.name

```

Proxy的construct和apply

捕捉器中 construct 和 apply，是应用于函数对象的

```

function foo() {
}

const fooProxy = new Proxy(foo, {
  apply: function(target, thisArg, argArray) {
    console.log("对foo函数进行了apply调用")
    return target.apply(thisArg, argArray)
  },
  construct: function(target, argArray, newTarget) {
    console.log("对foo函数进行了new调用")
    return new target(...argArray)
  }
})

fooProxy.apply({}, ["abc", "cba"])
new fooProxy("abc", "cba")

```

Reflect

- Reflect 也是ES6新增的一个API，它是一个对象，字面的意思是反射。
 - 它主要提供了很多操作 JavaScript 对象的方法，有点像 Object 中操作对象的方法；
 - 比如 Reflect.getPrototypeOf(target) 类似于 Object.getPrototypeOf()；
 - 比如 Reflect.defineProperty(target, propertyKey, attributes) 类似于 Object.defineProperty()
- 如果我们有 Object 可以做这些操作，那么为什么还需要有 Reflect 这样的新增对象呢？
 - 这是因为在早期的ECMA规范中没有考虑到这种对 对象本身 的操作如何设计会更加规范，所以将这些 API 放到了 Object 上面；
 - 但是 Object 作为一个构造函数，这些操作实际上放到它身上并不合适；
 - 另外还包含一些类似于 in、 delete 操作符，让JS看起来有一些奇怪
 - 所以在ES6中新增了 Reflect，让我们这些操作都集中到了 Reflect 对象上；
- Object 和 Reflect 对象之间的API关系，可以参考MDN文档：
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect/Comparing_Reflect_and_Object_methods
- Reflect的常见方法
 - Reflect.getPrototypeOf(target)
 - 类似于 Object.getPrototypeOf()。
 - Reflect.setPrototypeOf(target, prototype)
 - 设置对象原型的函数。返回一个 Boolean，如果更新成功，则返回 true。
 - Reflect.isExtensible(target)
 - 类似于 Object.isExtensible()
 - Reflect.preventExtensions(target)
 - 类似于 Object.preventExtensions()。返回一个 Boolean。
 - Reflect.getOwnPropertyDescriptor(target, propertyKey)

- 类似于 `Object.getOwnPropertyDescriptor()`。如果对象中存在该属性，则返回对应的属性描述符，否则返回 `undefined`
- `Reflect.defineProperty(target, propertyKey, attributes)`
 - 和 `Object.defineProperty()` 类似。如果设置成功就会返回 `true`
- `Reflect.ownKeys(target)`
 - 返回一个包含所有自身属性（不包含继承属性）的数组。（类似于 `Object.keys()`，但不会受 `enumerable` 影响）
- `Reflect.has(target, propertyKey)`
 - 判断一个对象是否存在某个属性，和 `in` 运算符的功能完全相同。
- `Reflect.get(target, propertyKey[, receiver])`
 - 获取对象身上某个属性的值，类似于 `target[name]`。
- `Reflect.set(target, propertyKey, value[, receiver])`
 - 将值分配给属性的函数。返回一个 `Boolean`，如果更新成功，则返回 `true`。
- `Reflect.deleteProperty(target, propertyKey)`
 - 作为函数的 `delete` 操作符，相当于执行 `delete target[name]`。
- `Reflect.apply(target, thisArgument, argumentsList)`
 - 对一个函数进行调用操作，同时可以传入一个数组作为调用参数。
和 `Function.prototype.apply()` 功能类似。
- `Reflect.construct(target, argumentsList[, newTarget])`
 - 对构造函数进行 `new` 操作，相当于执行 `new target(...args)`。
- `Reflect` 的使用
 - 可以将之前 `Proxy` 案例中对原对象的操作，都修改为 `Reflect` 来操作

```

const obj = {
  name: "why",
  age: 18
}

const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    console.log("get-----")
    return Reflect.get(target, key)
  },
  set: function(target, key, newValue, receiver) {
    console.log("set-----")
    target[key] = newValue

    const result = Reflect.set(target, key, newValue)
    if (result) {
    } else {
    }
  }
})

objProxy.name = "kobe"
console.log(objProxy.name)

```

Receiver的作用

- 在使用 getter、setter 的时候有一个 receiver 的参数
- 如果我们的源对象 (obj) 有 setter、getter 的访问器属性，那么可以通过 receiver 来改变里面的 this

```
const obj = {  
  _name: "why",  
  get name() {  
    return this._name  
  },  
  set name(newValue) {  
    this._name = newValue  
  }  
}  
  
const objProxy = new Proxy(obj, {  
  get: function(target, key, receiver) {  
    // receiver是创建出来的代理对象  
    console.log("get方法被访问-----", key, receiver)  
    console.log(receiver === objProxy)  
    return Reflect.get(target, key, receiver)  
  },  
  set: function(target, key, newValue, receiver) {  
    console.log("set方法被访问-----", key)  
    Reflect.set(target, key, newValue, receiver)  
  }  
})  
  
// console.log(objProxy.name)  
objProxy.name = "kobe"
```

Reflect的construct

```
function Student(name, age) {  
  this.name = name  
  this.age = age  
}  
  
function Teacher() {  
}  
  
const stu = new Student("why", 18)  
console.log(stu)  
console.log(stu.__proto__ === Student.prototype)  
  
// 执行Student函数中的内容，但是创建出来对象是Teacher对象  
const teacher = Reflect.construct(Student, ["why", 18], Teacher)  
console.log(teacher)  
console.log(teacher.__proto__ === Teacher.prototype)
```

响应式

可以自动响应数据变量的代码机制，我们就称之为是响应式的

- m有一个初始化的值，有一段代码使用了这个值
- 那么在m有一个新的值时，这段代码可以自动重新执行

响应式函数的实现

- 响应式函数设计
 - 当数据发生变化时，自动去执行某一个函数
- 问题：在开发中我们是有很多的函数的，我们如何区分一个函数需要响应式，还是不需要响应式呢
 - 下面的函数中 foo 需要在obj的name发生变化时，重新执行，做出相应
 - bar函数是一个完全独立于obj的函数，它不需要执行任何响应式操作

```
function foo() {  
    let newName = obj.name  
    console.log(obj.name)  
}  
function bar() {  
    const res = 20 + 30  
    console.log(res)  
}
```

- 怎么区分呢？
- 封装一个新的函数 watchFn
 - 传入到 watchFn 的函数，就是需要响应式的

```
const reactiveFns = []  
function watchFn(fn) {  
    reactiveFns.push(fn)  
    fn()  
}  
  
watchFn(function() {  
    let newName = obj.name  
    console.log(obj.name)  
})  
  
watchFn(function() {  
    console.log('my name is' + obj.name)  
})
```

响应式依赖的收集

- 目前我们收集的依赖是放到一个数组中来保存的，但是这里会存在数据管理的问题：
 - 我们在实际开发中需要监听很多对象的响应式
 - 这些对象需要监听的不只是一个属性，它们很多属性的变化，都会有对应的响应式函数

- 我们不可能在全局维护一大堆的数组来保存这些响应函数
- 所以我们要设计一个类，这个类用于管理某一个对象的某一个属性的所有响应式函数：
 - 相当于替代了原来的简单 reactiveFns 的数组

```

class Depend {
  constructor() {
    this.reactiveFns = []
  }

  addDepend(reactiveFn) {
    this.reactiveFns.push(reactiveFn)
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
const depend = new Depend()
function watchFn(fn) {
  depend.addDepend(fn)
}

// 对象的响应式
const obj = {
  name: "why", // depend对象
  age: 18 // depend对象
}

watchFn(function() {
  const newName = obj.name
  console.log("你好啊, 李银河")
  console.log("Hello World")
  console.log(obj.name)
})

watchFn(function() {
  console.log(obj.name, "demo function -----")
})

obj.name = "kobe"
depend.notify()

```

监听对象的变化

- 监听对象的变量：
 - 方式一：通过 Object.defineProperty 的方式（vue2采用的方式）
 - 方式二：通过 new Proxy 的方式（vue3采用的方式）

- 先以 Proxy 的方式来监听

```
class Depend {
  constructor() {
    this.reactiveFns = []
  }

  addDepend(reactiveFn) {
    this.reactiveFns.push(reactiveFn)
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
const depend = new Depend()
function watchFn(fn) {
  depend.addDepend(fn)
}

// 对象的响应式
const obj = {
  name: "why", // depend对象
  age: 18 // depend对象
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, newValue, receiver) {
    Reflect.set(target, key, newValue, receiver)
    depend.notify()
  }
})

watchFn(function() {
  const newName = objProxy.name
  console.log("你好啊, 李银河")
  console.log("Hello World")
  console.log(objProxy.name) // 100行
})

watchFn(function() {
  console.log(objProxy.name, "demo function -----")
})

watchFn(function() {
  console.log(objProxy.age, "age 变化是需要执行的----1")
})
```

```

watchFn(function() {
  console.log(objProxy.age, "age 变化是需要执行的----2")
})

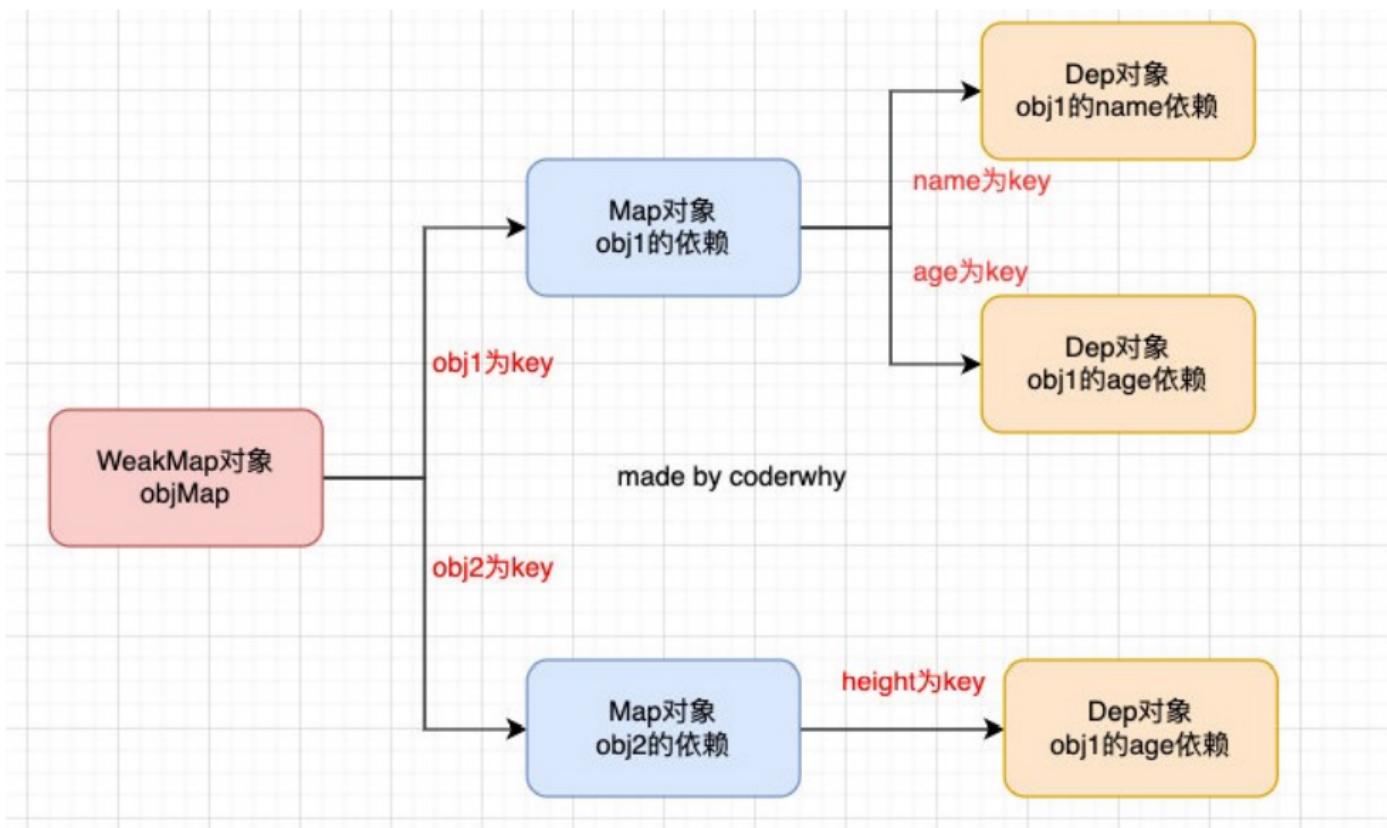
objProxy.name = "kobe"
objProxy.name = "james"
objProxy.name = "curry"

objProxy.age = 100

```

对象的依赖管理

- 目前是创建了一个 Depend 对象，用来管理对于 name 变化需要监听的响应函数：
 - 但是实际开发中我们会有不同的对象，另外会有不同的属性需要管理
 - 我们如何可以使用一种数据结构来管理不同对象的不同依赖关系呢
- 通过 WeakMap 管理这种响应式的数据依赖



- 对象依赖管理的实现
 - 可以写一个 getDepend 函数专门来管理这种依赖关系

```
class Depend {
  constructor() {
    this.reactiveFns = []
  }

  addDepend(reactiveFn) {
    this.reactiveFns.push(reactiveFn)
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
const depend = new Depend()
function watchFn(fn) {
  depend.addDepend(fn)
}

// 封装一个获取depend函数
const targetMap = new WeakMap()
function getDepend(target, key) {
  // 根据target对象获取map的过程
  let map = targetMap.get(target)
  if (!map) {
    map = new Map()
    targetMap.set(target, map)
  }

  // 根据key获取depend对象
  let depend = map.get(key)
  if (!depend) {
    depend = new Depend()
    map.set(key, depend)
  }
  return depend
}

// 对象的响应式
const obj = {
  name: "why", // depend对象
  age: 18 // depend对象
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, newValue, receiver) {
    Reflect.set(target, key, newValue, receiver)
  }
})
```

```
// depend.notify()  
const depend = getDepend(target, key)  
depend.notify()  
}  
})
```

正确的依赖收集

- 之前收集依赖的地方是在 `watchFn` 中：
 - 但是这种收集依赖的方式我们根本不知道是哪一个 `key` 的哪一个 `depend` 需要收集依赖
 - 你只能针对一个单独的 `depend` 对象来添加你的依赖对象
- 正确的应该是在哪里收集呢？应该在我们调用了 `Proxy` 的 `get` 捕获器时
 - 因为如果一个函数中使用了某个对象的 `key`，那么它应该被收集依赖

```
class Depend {
  constructor() {
    this.reactiveFns = []
  }

  addDepend(reactiveFn) {
    this.reactiveFns.push(reactiveFn)
  }

  notify() {
    console.log(this.reactiveFns)
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
let activeReactiveFn = null
function watchFn(fn) {
  activeReactiveFn = fn
  fn()
  activeReactiveFn = null
}

// 封装一个获取depend函数
const targetMap = new WeakMap()
function getDepend(target, key) {
  // 根据target对象获取map的过程
  let map = targetMap.get(target)
  if (!map) {
    map = new Map()
    targetMap.set(target, map)
  }

  // 根据key获取depend对象
  let depend = map.get(key)
  if (!depend) {
    depend = new Depend()
    map.set(key, depend)
  }
  return depend
}

// 对象的响应式
const obj = {
  name: "why", // depend对象
  age: 18 // depend对象
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    // 根据target.key获取对应的depend
```

```
const depend = getDepend(target, key)
// 给depend对象中添加响应函数
depend.addDepend(activeReactiveFn)

return Reflect.get(target, key, receiver)
},
set: function(target, key, newValue, receiver) {
  Reflect.set(target, key, newValue, receiver)
  // depend.notify()
  const depend = getDepend(target, key)
  depend.notify()
}
})
```

对Depend重构

- 有两个问题：
 - 问题一：如果函数中有用到两次 key，比如 name，那么这个函数会被收集两次；
 - 问题二：我们并不希望将添加 reactiveFn 放到 get 中，以为它是属于 Dep 的行为
- 所以我们需要对 Depend 类进行重构：
 - 解决问题一的方法：不使用数组，而是使用 Set
 - 解决问题二的方法：添加一个新的方法，用于收集依赖

```
// 保存当前需要收集的响应式函数
let activeReactiveFn = null

/**
 * Depend优化：
 * 1> depend方法
 * 2> 使用Set来保存依赖函数，而不是数组[]
 */

class Depend {
  constructor() {
    this.reactiveFns = new Set()
  }

  // addDepend(reactiveFn) {
  //   this.reactiveFns.add(reactiveFn)
  // }

  depend() {
    if (activeReactiveFn) {
      this.reactiveFns.add(activeReactiveFn)
    }
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
function watchFn(fn) {
  activeReactiveFn = fn
  fn()
  activeReactiveFn = null
}

// 封装一个获取depend函数
const targetMap = new WeakMap()
function getDepend(target, key) {
  // 根据target对象获取map的过程
  let map = targetMap.get(target)
  if (!map) {
    map = new Map()
    targetMap.set(target, map)
  }

  // 根据key获取depend对象
  let depend = map.get(key)
  if (!depend) {
    depend = new Depend()
    map.set(key, depend)
  }
}
```

```
return depend
}

// 对象的响应式
const obj = {
  name: "why", // depend对象
  age: 18 // depend对象
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    // 根据target.key获取对应的depend
    const depend = getDepend(target, key)
    // 给depend对象中添加响应函数
    // depend.addDepend(activeReactiveFn)
    depend.depend()

    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, newValue, receiver) {
    Reflect.set(target, key, newValue, receiver)
    // depend.notify()
    const depend = getDepend(target, key)
    depend.notify()
  }
})
```

创建响应式对象

- 我们目前的响应式是针对于obj一个对象的，我们可以创建出来一个函数，针对所有的对象都可以变成响应式对象

```
// 保存当前需要收集的响应式函数
let activeReactiveFn = null

/**
 * Depend优化：
 * 1> depend方法
 * 2> 使用Set来保存依赖函数，而不是数组[]
 */

class Depend {
  constructor() {
    this.reactiveFns = new Set()
  }

  // addDepend(reactiveFn) {
  //   this.reactiveFns.add(reactiveFn)
  // }

  depend() {
    if (activeReactiveFn) {
      this.reactiveFns.add(activeReactiveFn)
    }
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
function watchFn(fn) {
  activeReactiveFn = fn
  fn()
  activeReactiveFn = null
}

// 封装一个获取depend函数
const targetMap = new WeakMap()
function getDepend(target, key) {
  // 根据target对象获取map的过程
  let map = targetMap.get(target)
  if (!map) {
    map = new Map()
    targetMap.set(target, map)
  }

  // 根据key获取depend对象
  let depend = map.get(key)
  if (!depend) {
    depend = new Depend()
    map.set(key, depend)
  }
}
```

```

    return depend
}

function reactive(obj) {
  return new Proxy(obj, {
    get: function(target, key, receiver) {
      // 根据target.key获取对应的depend
      const depend = getDepend(target, key)
      // 给depend对象中添加响应函数
      // depend.addDepend(activeReactiveFn)
      depend.depend()

      return Reflect.get(target, key, receiver)
    },
    set: function(target, key, newValue, receiver) {
      Reflect.set(target, key, newValue, receiver)
      // depend.notify()
      const depend = getDepend(target, key)
      depend.notify()
    }
  })
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = reactive({
  name: "why", // depend对象
  age: 18 // depend对象
})

const infoProxy = reactive({
  address: "广州市",
  height: 1.88
})

watchFn(() => {
  console.log(infoProxy.address)
})

infoProxy.address = "北京市"

const foo = reactive({
  name: "foo"
})

watchFn(() => {
  console.log(foo.name)
})

foo.name = "bar"

```

Vue2响应式原理

- Vue3主要是通过 Proxy 来监听数据的变化以及收集相关的依赖的

- Vue2中通过 `Object.defineProperty` 的方式来实现对象属性的监听

可以将 `reactive` 函数进行如下的重构：

- 在传入对象时，我们可以遍历所有的key，并且通过属性存储描述符来监听属性的获取和修改
- 在 `setter` 和 `getter` 方法中的逻辑和前面的 `Proxy` 是一致的

```
// 保存当前需要收集的响应式函数
let activeReactiveFn = null

/**
 * Depend优化：
 * 1> depend方法
 * 2> 使用Set来保存依赖函数，而不是数组[]
 */

class Depend {
  constructor() {
    this.reactiveFns = new Set()
  }

  // addDepend(reactiveFn) {
  //   this.reactiveFns.add(reactiveFn)
  // }

  depend() {
    if (activeReactiveFn) {
      this.reactiveFns.add(activeReactiveFn)
    }
  }

  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}

// 封装一个响应式的函数
function watchFn(fn) {
  activeReactiveFn = fn
  fn()
  activeReactiveFn = null
}

// 封装一个获取depend函数
const targetMap = new WeakMap()
function getDepend(target, key) {
  // 根据target对象获取map的过程
  let map = targetMap.get(target)
  if (!map) {
    map = new Map()
    targetMap.set(target, map)
  }

  // 根据key获取depend对象
  let depend = map.get(key)
  if (!depend) {
    depend = new Depend()
    map.set(key, depend)
  }
}
```

```
return depend
}

function reactive(obj) {
// {name: "why", age: 18}
// ES6之前，使用Object.defineProperty
Object.keys(obj).forEach(key => {
  let value = obj[key]
  Object.defineProperty(obj, key, {
    get: function() {
      const depend = getDepend(obj, key)
      depend.depend()
      return value
    },
    set: function(newValue) {
      value = newValue
      const depend = getDepend(obj, key)
      depend.notify()
    }
  })
})
return obj
}

// 监听对象的属性变量: Proxy(vue3)/Object.defineProperty(vue2)
const objProxy = reactive({
  name: "why", // depend对象
  age: 18 // depend对象
})

const infoProxy = reactive({
  address: "广州市",
  height: 1.88
})

watchFn(() => {
  console.log(infoProxy.address)
})

infoProxy.address = "北京市"

const foo = reactive({
  name: "foo"
})

watchFn(() => {
  console.log(foo.name)
})

foo.name = "bar"
foo.name = "hhh"
```

Promise

Promise使用

实例：

- 我们调用一个函数，这个函数中发送网络请求（我们可以用定时器来模拟）
- 如果发送网络请求成功了，那么告知调用者发送成功，并且将相关数据返回过去
- 如果发送网络请求失败了，那么告知调用者发送失败，并且告知错误信息

两个主要的问题：

- 第一，我们需要自己来设计回调函数、回调函数的名称、回调函数的使用等
- 第二，对于不同的人、不同的框架设计出来的方案是不同的，那么我们必须耐心去看别人的源码或者文档，以便可以理解它这个函数到底怎么用

我们来看一下 Promise 的 API 是怎么样的：

- Promise 是一个类，可以翻译成承诺
- 当我们需要给予调用者一个承诺：待会儿我会给你回调数据时，就可以创建一个 Promise 的对象
- 在通过 new 创建 Promise 对象时，我们需要传入一个回调函数，我们称之为 executor
 - 这个回调函数会被立即执行，并且给传入另外两个回调函数 resolve、reject
 - 当我们调用 resolve 回调函数时，会执行 Promise 对象的 then 方法传入的回调函数
 - 当我们调用 reject 回调函数时，会执行 Promise 对象的 catch 方法传入的回调函数

Promise 使用过程，可以将它划分成三个状态：

- 待定（pending）：初始状态，既没有被兑现，也没有被拒绝
 - 当执行 executor 中的代码时，处于该状态
- 已兑现（fulfilled）：意味着操作成功完成
 - 执行了 resolve 时，处于该状态
- 已拒绝（rejected）：意味着操作失败
 - 执行了 reject 时，处于该状态

Executor

Executor 是在创建 Promise 时需要传入的一个回调函数，这个回调函数会被立即执行，并且传入两个参数

- 通常我们会在 Executor 中确定我们的 Promise 状态：
 - 通过 resolve，可以兑现（fulfilled）Promise 的状态，我们也可以称之为已决议（resolved）
 - 通过 reject，可以拒绝（reject）Promise 的状态
- 注意：一旦状态被确定下来，Promise 的状态会被锁死，该 Promise 的状态是不可更改的
- 在我们调用 resolve 的时候，如果 resolve 传入的值本身不是一个 Promise，那么会将该 Promise 的状态变成 兑现（fulfilled）

- 在之后我们去调用 `reject` 时，已经不会有任何的响应了（并不是这行代码不会执行，而是无法改变 `Promise` 状态）

resolve不同值的区别

- 情况一：如果 `resolve` 传入一个普通的值或者对象，那么这个值会作为 `then` 回调的参数；
- 情况二：如果 `resolve` 中传入的是另外一个 `Promise`，那么这个新 `Promise` 会决定原 `Promise` 的状态；
- 情况三：如果 `resolve` 中传入的是一个对象，并且这个对象有实现 `then` 方法，那么会执行该 `then` 方法，并且根据 `then` 方法的结果来决定 `Promise` 的状态

then方法

`then` 方法是 `Promise` 对象上的一个方法：它其实是放在 `Promise` 的原型上的 `Promise.prototype.then`

- `then` 方法接受两个参数：
 - `fulfilled` 的回调函数：当状态变成 `fulfilled` 时会回调的函数
 - `reject` 的回调函数：当状态变成 `reject` 时会回调的函数
- 一个 `Promise` 的 `then` 方法是可以被多次调用的：
 - 每次调用我们都可以传入对应的 `fulfilled` 回调；
 - 当 `Promise` 的状态变成 `fulfilled` 的时候，这些回调函数都会被执行
- `then` 方法本身是有返回值的，它的返回值是一个 `Promise`，所以我们可以进行链式调用：
 - 但是 `then` 方法返回的 `Promise` 到底处于什么样的状态呢？
 - `Promise` 有三种状态，那么这个 `Promise` 处于什么状态呢？
 - 当 `then` 方法中的回调函数本身在执行的时候，那么它处于 `pending` 状态
 - 当 `then` 方法中的回调函数返回一个结果时，那么它处于 `fulfilled` 状态，并且会将结果作为 `resolve` 的参数
 - 情况一：返回一个普通的值
 - 情况二：返回一个 `Promise`
 - 情况三：返回一个 `thenable` 值
 - 当 `then` 方法抛出一个异常时，那么它处于 `reject` 状态

catch方法

- `catch`方法也是 `Promise` 对象上的一个方法：它也是放在 `Promise` 的原型上的 `Promise.prototype.catch`
- 一个 `Promise` 的 `catch` 方法是可以被多次调用的：
 - 每次调用我们都可以传入对应的 `reject` 回调
 - 当 `Promise` 的状态变成 `reject` 的时候，这些回调函数都会被执行
- 事实上 `catch` 方法也是会返回一个 `Promise` 对象的，所以 `catch` 方法后面我们可以继续调用 `then` 方法或者 `catch` 方法：
- `catch` 传入的回调在执行完后，默认状态依然会是 `fulfilled` 的

finally方法

- `finally` 是在ES9 (ES2018) 中新增的一个特性：表示无论 `Promise` 对象无论变成 `fulfilled` 还是 `reject` 状态，最终都会被执行的代码。
- `finally` 方法是不接收参数的，因为无论前面是 `fulfilled` 状态，还是 `reject` 状态，它都会执行

Promise的类方法

- 有时候我们已经有一个现成的内容了，希望将其转成 `Promise` 来使用，这个时候我们可以使用 `Promise.resolve` 方法来完成。
 - `Promise.resolve` 的用法相当于 `new Promise`，并且执行 `resolve` 操作
 - `resolve` 参数的形态：
 - 情况一：参数是一个普通的值或者对象
 - 情况二：参数本身是 `Promise`
 - 情况三：参数是一个 `thenable`
- `Promise.reject` 的用法相当于 `new Promise`，只是会调用 `reject`：
 - `Promise.reject` 传入的参数无论是什么形态，都会直接作为 `reject` 状态的参数传递到 `catch` 的
- `Promise.all`：
 - 它的作用是将多个 `Promise` 包裹在一起形成一个新的 `Promise`；
 - 新的 `Promise` 状态由包裹的所有 `Promise` 共同决定：
 - 当所有的 `Promise` 状态变成 `fulfilled` 状态时，新的 `Promise` 状态为 `fulfilled`，并且会将所有 `Promise` 的返回值组成一个数组；
 - 当有一个 `Promise` 状态为 `reject` 时，新的 `Promise` 状态为 `reject`，并且会将第一个 `reject` 的返回值作为参数
- `allSettled` 方法
 - `all` 方法有一个缺陷：当有其中一个 `Promise` 变成 `reject` 状态时，新 `Promise` 就会立即变成对应的 `reject` 状态。
 - 那么对于 `resolved` 的，以及依然处于 `pending` 状态的 `Promise`，我们是获取不到对应的结果的
 - 在ES11 (ES2020) 中，添加了新的API `Promise.allSettled`
 - 该方法会在所有的 `Promise` 都有结果 (`settled`)，无论是 `fulfilled`，还是 `reject` 时，才会有最终的状态
 - 并且这个 `Promise` 的结果一定是 `fulfilled` 的
 - `allSettled` 的结果是一个数组，数组中存放着每一个 `Promise` 的结果，并且是对应一个对象的
 - 这个对象中包含 `status` 状态，以及对应的 `value` 值
- `race` 方法
 - 如果有一个 `Promise` 有了结果，我们就希望决定最终新 `Promise` 的状态，那么可以使用 `race` 方法
 - `race` 是竞技、竞赛的意思，表示多个 `Promise` 相互竞争，谁先有结果，那么就使用谁的结果
- `any` 方法
 - `any` 方法是ES12中新增的方法，会等到一个 `fulfilled` 状态，才会决定新 `Promise` 的状态
 - 如果所有的 `Promise` 都是 `reject` 的，那么也会等到所有的 `Promise` 都变成 `rejected` 状态
 - 如果所有的 `Promise` 都是 `reject` 的，那么会报一个 `AggregateError` 的错误

Promise 实现

1. 简单总结手写 Promise

- 一. Promise 规范

<https://promisesaplus.com/>

- 二. Promise 类设计

```
class HYPromise {}
```

```
function HYPromise() {}
```

- 三. 构造函数的规划

```
class HYPromise {  
    constructor(executor) {  
        // 定义状态  
        // 定义resolve、reject回调  
        // resolve执行微任务队列：改变状态、获取value、then传入执行成功回调  
        // reject执行微任务队列：改变状态、获取reason、then传入执行失败回调  
  
        // try catch  
        executor(resolve, reject)  
    }  
}
```

- 四. then 方法的实现

```
class HYPromise {  
    then(onFulfilled, onRejected) {  
        // this.onFulfilled = onFulfilled  
        // this.onRejected = onRejected  
  
        // 1.判断onFulfilled、onRejected，会给默认值  
  
        // 2.返回Promise resolve/reject  
  
        // 3.判断之前的promise状态是否确定  
        // onFulfilled/onRejected直接执行（捕获异常）  
  
        // 4.添加到数组中push(() => { 执行 onFulfilled/onRejected 直接执行代码})  
    }  
}
```

- 五. catch 方法

```
class HYPromise {
  catch(onRejected) {
    return this.then(undefined, onRejected)
  }
}
```

- 六. finally

```
class HYPromise {
  finally(onFinally) {
    return this.then(() => {onFinally()}, () => {onFinally()})
  }
}
```

- 七. resolve/reject
- 八. all/allSettled

核心：要知道new Promise的resolve、reject在什么情况下执行

all:

- 情况一：所有的都有结果
- 情况二：有一个 reject

allSettled :

- 情况：所有都有结果，并且一定执行 resolve
- 九. race/any

race :

- 情况：只要有结果

any :

- 情况一：必须等到一个 resolve 结果
- 情况二：都没有 resolve，所有的都是 reject

2.

```
// ES6 ES2015
// https://promisesaplus.com/
const PROMISE_STATUS_PENDING = 'pending'
const PROMISE_STATUS_FULFILLED = 'fulfilled'
const PROMISE_STATUS_REJECTED = 'rejected'

// 工具函数
function execFunctionWithCatchError(execFn, value, resolve, reject) {
  try {
    const result = execFn(value)
    resolve(result)
  } catch(err) {
    reject(err)
  }
}

class HPromise {
  constructor(executor) {
    this.status = PROMISE_STATUS_PENDING
    this.value = undefined
    this.reason = undefined
    this.onFulfilledFns = []
    this.onRejectedFns = []

    const resolve = (value) => {
      if (this.status === PROMISE_STATUS_PENDING) {
        // 添加微任务
        queueMicrotask(() => {
          if (this.status !== PROMISE_STATUS_PENDING) return
          this.status = PROMISE_STATUS_FULFILLED
          this.value = value
          this.onFulfilledFns.forEach(fn => {
            fn(this.value)
          })
        });
      }
    }

    const reject = (reason) => {
      if (this.status === PROMISE_STATUS_PENDING) {
        // 添加微任务
        queueMicrotask(() => {
          if (this.status !== PROMISE_STATUS_PENDING) return
          this.status = PROMISE_STATUS_REJECTED
          this.reason = reason
          this.onRejectedFns.forEach(fn => {
            fn(this.reason)
          })
        });
      }
    }

    try {
      executor(resolve, reject)
    }
  }
}
```

```
        } catch (err) {
          reject(err)
        }
      }

      then(onFulfilled, onRejected) {
        const defaultOnRejected = err => { throw err }
        onRejected = onRejected || defaultOnRejected

        const defaultOnFulfilled = value => { return value }
        onFulfilled = onFulfilled || defaultOnFulfilled

        return new HYPromise((resolve, reject) => {
          // 1.如果在then调用的时候，状态已经确定下来
          if (this.status === PROMISE_STATUS_FULFILLED && onFulfilled) {
            execFunctionWithCatchError(onFulfilled, this.value, resolve, reject)
          }
          if (this.status === PROMISE_STATUS_REJECTED && onRejected) {
            execFunctionWithCatchError(onRejected, this.reason, resolve, reject)
          }

          // 2.将成功回调和失败的回调放到数组中
          if (this.status === PROMISE_STATUS_PENDING) {
            if (onFulfilled) this.onFulfilledFns.push(() => {
              execFunctionWithCatchError(onFulfilled, this.value, resolve, reject)
            })
            if (onRejected) this.onRejectedFns.push(() => {
              execFunctionWithCatchError(onRejected, this.reason, resolve, reject)
            })
          }
        })
      }

      catch(onRejected) {
        return this.then(undefined, onRejected)
      }

      finally(onFinally) {
        this.then(() => {
          onFinally()
        }, () => {
          onFinally()
        })
      }

      static resolve(value) {
        return new HYPromise((resolve) => resolve(value))
      }

      static reject(reason) {
        return new HYPromise((resolve, reject) => reject(reason))
      }

      static all(promises) {
```

```
// 问题关键：什么时候要执行resolve，什么时候要执行reject
return new HYPromise((resolve, reject) => {
    const values = []
    promises.forEach(promise => {
        promise.then(res => {
            values.push(res)
            if (values.length === promises.length) {
                resolve(values)
            }
        }, err => {
            reject(err)
        })
    })
})
}

static allSettled(promises) {
    return new HYPromise((resolve) => {
        const results = []
        promises.forEach(promise => {
            promise.then(res => {
                results.push({ status: PROMISE_STATUS_FULFILLED, value: res})
                if (results.length === promises.length) {
                    resolve(results)
                }
            }, err => {
                results.push({ status: PROMISE_STATUS_REJECTED, value: err})
                if (results.length === promises.length) {
                    resolve(results)
                }
            })
        })
    })
}
}

static race(promises) {
    return new HYPromise((resolve, reject) => {
        promises.forEach(promise => {
            // promise.then(res => {
            //     resolve(res)
            // }, err => {
            //     reject(err)
            // })
            promise.then(resolve, reject)
        })
    })
}
}

static any(promises) {
    // resolve必须等到有一个成功的结果
    // reject所有的都失败才执行reject
    const reasons = []
    return new HYPromise((resolve, reject) => {
        promises.forEach(promise => {
```

```
promise.then(resolve, err => {
  reasons.push(err)
  if (reasons.length === promises.length) {
    reject(new AggregateError(reasons))
  }
})
})
})
}
}
```

Iterator-Generator

迭代器

迭代器是帮助我们对某个数据结构进行遍历的对象

- 在JavaScript中，迭代器也是一个具体的对象，这个对象需要符合迭代器协（iterator protocol）：
 - 迭代器协议定义了产生一系列值（无论是有限还是无限个）的标准方式
 - 那么在js中这个标准就是一个特定的 `next` 方法
- `next` 方法有如下的要求：
 - 一个无参数或者一个参数的函数，返回一个应当拥有以下两个属性的对象：
 - `done` (boolean)
 - 如果迭代器可以产生序列中的下一个值，则为 `false`。（这等价于没有指定 `done` 这个属性。）
 - 如果迭代器已将序列迭代完毕，则为 `true`。这种情况下，`value` 是可选的，如果它依然存在，即为迭代结束之后的默认返回值。
 - `value`
 - 迭代器返回的任何 JavaScript 值。`done` 为 `true` 时可省略。

可迭代对象

- 当一个对象实现了 `iterable protocol` 协议时，它就是一个可迭代对象
- 这个对象的要求是必须实现 `@@iterator` 方法，在代码中我们使用 `Symbol.iterator` 访问该属性
- 当一个对象变成一个可迭代对象的时候，进行某些迭代操作，比如 `for...of` 操作时，其实就会调用它的 `@@iterator` 方法

```

// 创建一个迭代器对象来访问数组
const iterableObj = {
  names: ["abc", "cba", "nba"],
  [Symbol.iterator]: function() {
    let index = 0
    return {
      next: () => {
        if (index < this.names.length) {
          return { done: false, value: this.names[index++] }
        } else {
          return { done: true, value: undefined }
        }
      }
    }
  }
}
// for...of可以遍历的东西必须是一个可迭代对象

```

原生迭代器对象

事实上我们平时创建的很多原生对象已经实现了可迭代协议，会生成一个迭代器对象的：

- String、Array、Map、Set、arguments 对象、NodeList 集合

可迭代对象的应用

- for ... of、展开语法 (spread syntax)、yield*、解构赋值 (Destructuring_assignment)
- 创建一些对象
时：new Map([Iterable])、new WeakMap([iterable])、new Set([iterable])、new WeakSet([iterable])
- 一些方法的调用：Promise.all(iterable)、Promise.race(iterable)、Array.from(iterable)

自定义类的迭代

- 在面向对象开发中，我们可以通过 class 定义一个自己的类，这个类可以创建很多的对象
- 如果我们也希望自己的类创建出来的对象默认是可迭代的，那么在设计类的时候我们就可以添加上 @@iterator 方法
- 迭代器在某些情况下会在没有完全迭代的情况下中断
 - 比如遍历的过程中通过 break、continue、return、throw 中断了循环操作
 - 比如在解构的时候，没有解构所有的值
 - 这个时候我们想要监听中断的话，可以添加 return 方法
- 案例：创建一个 classroom 的类
 - 教室有自己的位置、名称、当前教室的学生
 - 这个教室可以进来新学生（push）
 - 创建的教室对象是可迭代对象

```

// 案例：创建一个教室类，创建出来的对象都是可迭代对象
class Classroom {
    constructor(address, name, students) {
        this.address = address
        this.name = name
        this.students = students
    }

    entry(newStudent) {
        this.students.push(newStudent)
    }

    [Symbol.iterator]() {
        let index = 0
        return {
            next: () => {
                if (index < this.students.length) {
                    return { done: false, value: this.students[index++] }
                } else {
                    return { done: true, value: undefined }
                }
            },
            return: () => {
                console.log("迭代器提前终止了~")
                return { done: true, value: undefined }
            }
        }
    }
}

const classroom = new Classroom("3幢5楼205", "计算机教室", ["james", "kobe", "curry", "why"])
classroom.entry("lilei")

for (const stu of classroom) {
    console.log(stu)
    if (stu === "why") break
}

function Person() {

}

Person.prototype[Symbol.iterator] = function() {
}

```

生成器

生成器函数需要在 `function` 的后面加一个符号： *
 生成器函数可以通过 `yield` 关键字来控制函数的执行流程

生成器函数的返回值是一个Generator (生成器) :

生成器事实上是一种特殊的迭代器

生成器传递参数 – next函数

在调用 `next` 函数的时候，可以给它传递参数，那么这个参数会作为上一个 `yield` 语句的返回值

注意：也就是说我们是为本次的函数代码块执行提供了一个值

生成器提前结束 – return 函数

- 还有一个可以给生成器函数传递参数的方法是通过 `return` 函数
 - `return` 传值后这个生成器函数就会结束，之后调用 `next` 不会继续生成值了

生成器抛出异常 – throw 函数

- 除了给生成器函数内部传递参数之外，也可以给生成器函数内部抛出异常：
 - 抛出异常后我们可以在生成器函数中捕获异常
 - 但是在 `catch` 语句中不能继续 `yield` 新的值了，但是可以在 `catch` 语句外使用 `yield` 继续中断函数的执行

```
function* foo() {
  console.log("代码开始执行~")

  const value1 = 100
  try {
    yield value1
  } catch (error) {
    console.log("捕获到异常情况:", error)

    yield "abc"
  }

  console.log("第二段代码继续执行")
  const value2 = 200
  yield value2

  console.log("代码执行结束~")
}

const generator = foo()

const result = generator.next()
generator.throw("error message")
```

生成器替代迭代器

```
// 1.生成器来替代迭代器
function* createArrayIterator(arr) {
    // 3.第三种写法 yield*
    yield* arr

    // 2.第二种写法
    // for (const item of arr) {
    //     yield item
    // }

    // 1.第一种写法
    // yield "abc" // { done: false, value: "abc" }
    // yield "cba" // { done: false, value: "abc" }
    // yield "nba" // { done: false, value: "abc" }
}
```

- 自定义类迭代 – 生成器实现

```
// 2. 创建一个函数，这个函数可以迭代一个范围内的数字
// 10 20
function* createRangeIterator(start, end) {
  let index = start
  while (index < end) {
    yield index++
  }

  // let index = start
  // return {
  //   next: function() {
  //     if (index < end) {
  //       return { done: false, value: index++ }
  //     } else {
  //       return { done: true, value: undefined }
  //     }
  //   }
  // }
}

const rangeIterator = createRangeIterator(10, 20)
console.log(rangeIterator.next())
console.log(rangeIterator.next())
console.log(rangeIterator.next())
console.log(rangeIterator.next())
console.log(rangeIterator.next())
```

- 自定义类迭代 – 生成器实现

```
// 3.class案例
class Classroom {
    constructor(address, name, students) {
        this.address = address
        this.name = name
        this.students = students
    }

    entry(newStudent) {
        this.students.push(newStudent)
    }

    foo = () => {
        console.log("foo function")
    }

    // [Symbol.iterator] = function*() {
    //     yield* this.students
    // }

    *[Symbol.iterator]() {
        yield* this.students
    }
}

const classroom = new Classroom("3幢", "1102", ["abc", "cba"])
for (const item of classroom) {
    console.log(item)
}
```

对生成器的操作

- 既然生成器是一个迭代器，那么我们可以对其进行如下的操作：

```
const namesIterator1 = createArrayIterator(names)
for (const item of namesIterator1) {
    console.log(item)
}

const namesIterator2 = createArrayIterator(names)
const set = new Set(namesIterator2)
console.log(set)

const namesIterator3 = createArrayIterator(names)
Promise.all(namesIterator3).then(res => {
    console.log(res)
})
```

异步处理方案

```
// request.js
function requestData(url) {
    // 异步请求的代码会被放入到executor中
    return new Promise((resolve, reject) => {
        // 模拟网络请求
        setTimeout(() => {
            // 拿到请求的结果
            resolve(url)
        }, 2000);
    })
}

// 需求:
// 1> url: why -> res: why
// 2> url: res + "aaa" -> res: whyaaa
// 3> url: res + "bbb" => res: whyaaabbb

// 1.第一种方案: 多次回调
// 回调地狱
requestData("why").then(res => {
    requestData(res + "aaa").then(res => {
        requestData(res + "bbb").then(res => {
            console.log(res)
        })
    })
})
}

// 2.第二种方案: Promise中then的返回值来解决
requestData("why").then(res => {
    return requestData(res + "aaa")
}).then(res => {
    return requestData(res + "bbb")
}).then(res => {
    console.log(res)
})

// 3.第三种方案: Promise + generator实现
function* getData() {
    const res1 = yield requestData("why")
    const res2 = yield requestData(res1 + "aaa")
    const res3 = yield requestData(res2 + "bbb")
    const res4 = yield requestData(res3 + "ccc")
    console.log(res4)
}

function* getDepartment() {
    const user = yield requestData("id")
    const department = yield requestData(user.departmentId)
}

// 1> 手动执行生成器函数
const generator = getData()
generator.next().value.then(res => {
```

```

generator.next(res).value.then(res => {
  generator.next(res).value.then(res => {
    generator.next(res)
  })
})
})

// 2> 自己封装了一个自动执行的函数
function execGenerator(genFn) {
  const generator = genFn()
  function exec(res) {
    const result = generator.next(res)
    if (result.done) {
      return result.value
    }
    result.value.then(res => {
      exec(res)
    })
  }
  exec()
}

// execGenerator(getData)
// execGenerator(getDepartment)

// 3> 第三方包co自动执行
// TJ: co/nvm/commander(coderwhy/vue cli)/express/koa(egg)
// const co = require('co')
// co(getData)

// 4.第四种方案: async/await
async function getData() {
  const res1 = await requestData("why")
  const res2 = await requestData(res1 + "aaa")
  const res3 = await requestData(res2 + "bbb")
  const res4 = await requestData(res3 + "ccc")
  console.log(res4)
}

getData()

```

await-async-事件循环

异步函数 async function

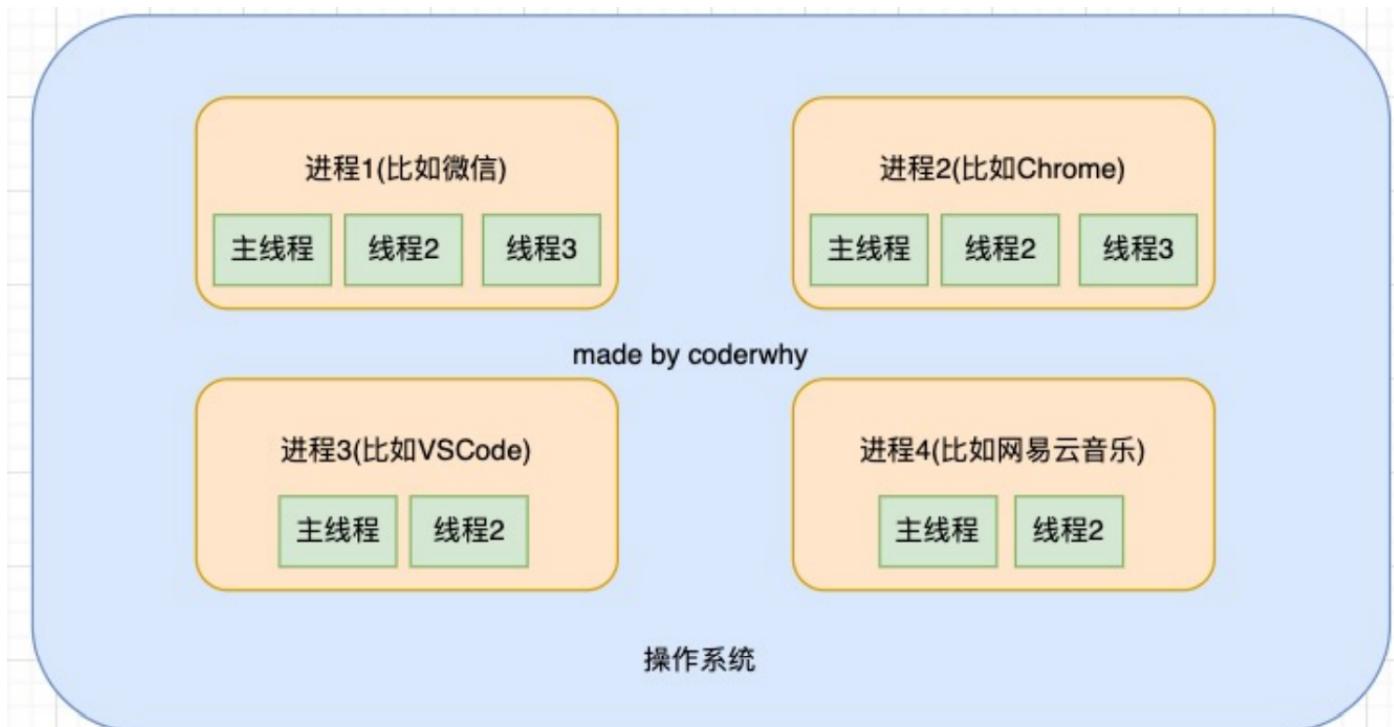
- `async` 关键字用于声明一个异步函数:
 - `async` 是 `asynchronous` 单词的缩写, 异步、非同步
 - `sync` 是 `synchronous` 单词的缩写, 同步、同时
- 异步函数的执行流程

- 异步函数的内部代码执行过程和普通的函数是一致的，默认情况下也是会被同步执行
- 异步函数有返回值时，和普通函数会有区别：
 - 情况一：异步函数也可以有返回值，但是异步函数的返回值会被包裹到 `Promise.resolve` 中
 - 情况二：如果我们的异步函数的返回值是 `Promise`，`Promise.resolve` 的状态会由 `Promise` 决定
 - 情况三：如果我们的异步函数的返回值是一个对象并且实现了 `thenable`，那么会由对象的 `then` 方法来决定
- 如果我们在 `async` 中抛出了异常，那么程序并不会像普通函数一样报错，而是会作为 `Promise` 的 `reject` 来传递
- `async` 函数另外一个特殊之处就是可以在它内部使用 `await` 关键字，而普通函数中是不可以的
 - 通常使用 `await` 是后面会跟上一个表达式，这个表达式会返回一个 `Promise`
 - 那么 `await` 会等到 `Promise` 的状态变成 `fulfilled` 状态，之后继续执行异步函数
 - 如果 `await` 后面是一个普通的值，那么会直接返回这个值
 - 如果 `await` 后面是一个 `thenable` 的对象，那么会根据对象的 `then` 方法调用来决定后续的值
 - 如果 `await` 后面的表达式，返回的 `Promise` 是 `reject` 的状态，那么会将这个 `reject` 结果直接作为函数的 `Promise` 的 `reject` 值

进程和线程

1. 线程和进程是操作系统中的两个概念：

- 进程 (process)：计算机已经运行的程序，是操作系统管理程序的一种方式
- 线程 (thread)：操作系统能够运行运算调度的最小单位，通常情况下它被包含在进程中
- 2. 解释：
 - 进程：我们可以认为，启动一个应用程序，就会默认启动一个进程（也可能是多个进程）
 - 线程：每一个进程中，都会启动至少一个线程用来执行程序中的代码，这个线程被称之为为主线程
 - 所以我们 also 可以说进程是线程的容器



操作系统的的工作方式

- 操作系统是如何做到同时让多个进程（边听歌、边写代码、边查阅资料）同时工作呢？
 - 这是因为CPU的运算速度非常快，它可以快速的在多个进程之间迅速的切换
 - 当我们进程中的线程获取到时间片时，就可以快速执行我们编写的代码
 - 对于用户来说是感受不到这种快速的切换的

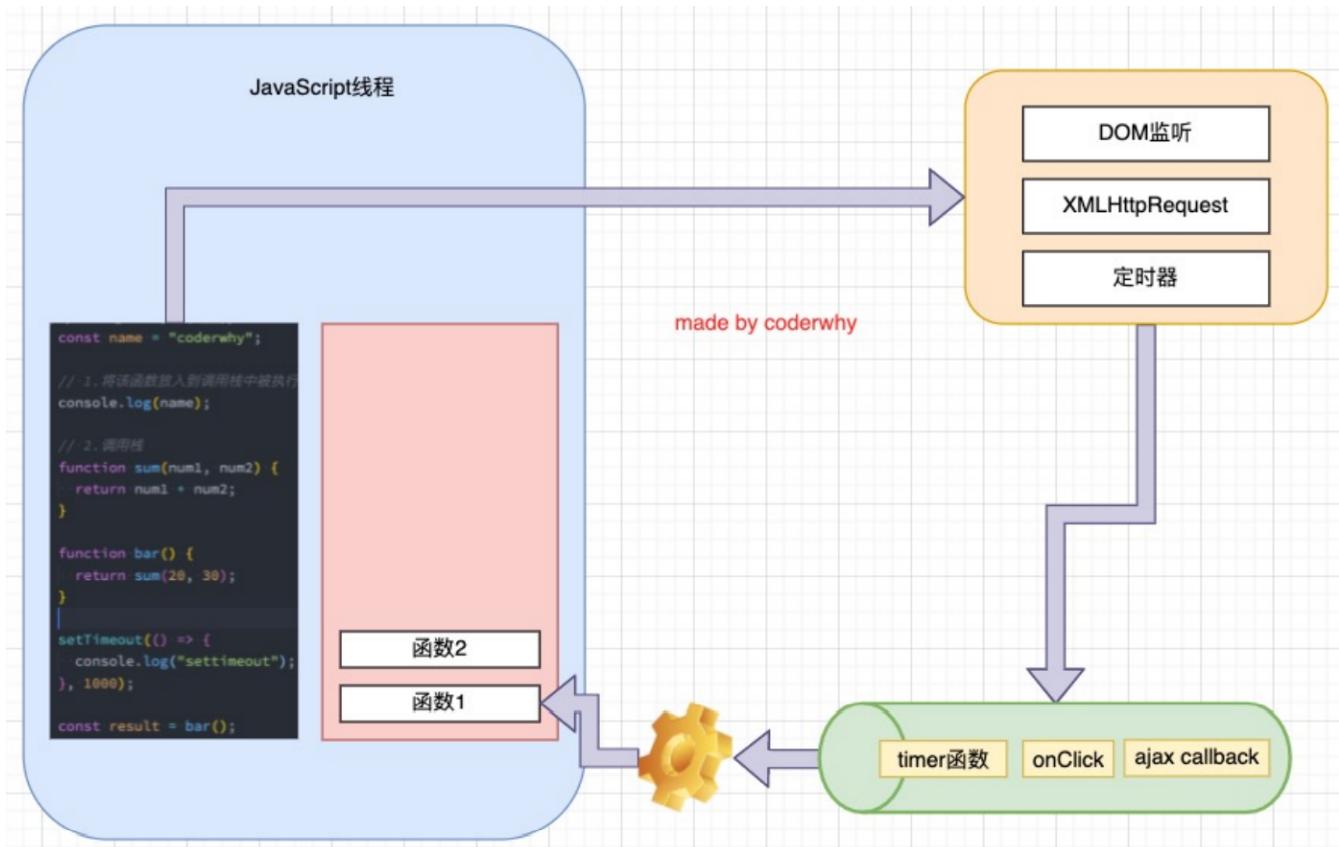
浏览器中的JavaScript线程

JavaScript是单线程的，但是JavaScript的线程应该有自己的容器进程：浏览器或者Node

- 目前多数的浏览器其实都是多进程的，当我们打开一个tab页面时就会开启一个新的进程，这是为了防止一个页面卡死而造成所有页面无法响应，整个浏览器需要强制退出；
 - 每个进程中又有很多的线程，其中包括执行JavaScript代码的线程
- JavaScript的代码执行是在一个单独的线程中执行的
 - 这就意味着JavaScript的代码，在同一个时刻只能做一件事
 - 如果这件事是非常耗时的，就意味着当前的线程就会被阻塞
- 所以真正耗时的操作，实际上并不是由JavaScript线程在执行的：
 - 浏览器的每个进程是多线程的，那么其他线程可以来完成这个耗时的操作
 - 比如网络请求、定时器，我们只需要在特性的时候执行应该有的回调即可

浏览器的事件循环

- 如果在执行JavaScript代码的过程中，有异步操作呢？
 - 函数会被放到入调用栈中，执行会立即结束，并不会阻塞后续代码的执行



宏任务和微任务

- 事件循环中并非只维护着一个队列，事实上是有两个队列：
 - 宏任务队列 (macrotask queue) : ajax、 setTimeout、 setInterval、 DOM监听、 UI Rendering 等
 - 微任务队列 (microtask queue) : Promise的then回调、 Mutation Observer(观察器，在指定的 DOM 发生变化时被调用) API、 queueMicrotask() 等
- 那么事件循环对于两个队列的优先级是怎么样的呢？
 1. main script中的代码优先执行 (编写的顶层script代码)；
 2. 在执行任何一个宏任务之前 (不是队列，是一个宏任务)，都会先查看微任务队列中是否有任务需要执行
 - 也就是宏任务执行之前，必须保证微任务队列是空的；
 - 如果不为空，那么就优先执行微任务队列中的任务 (回调)

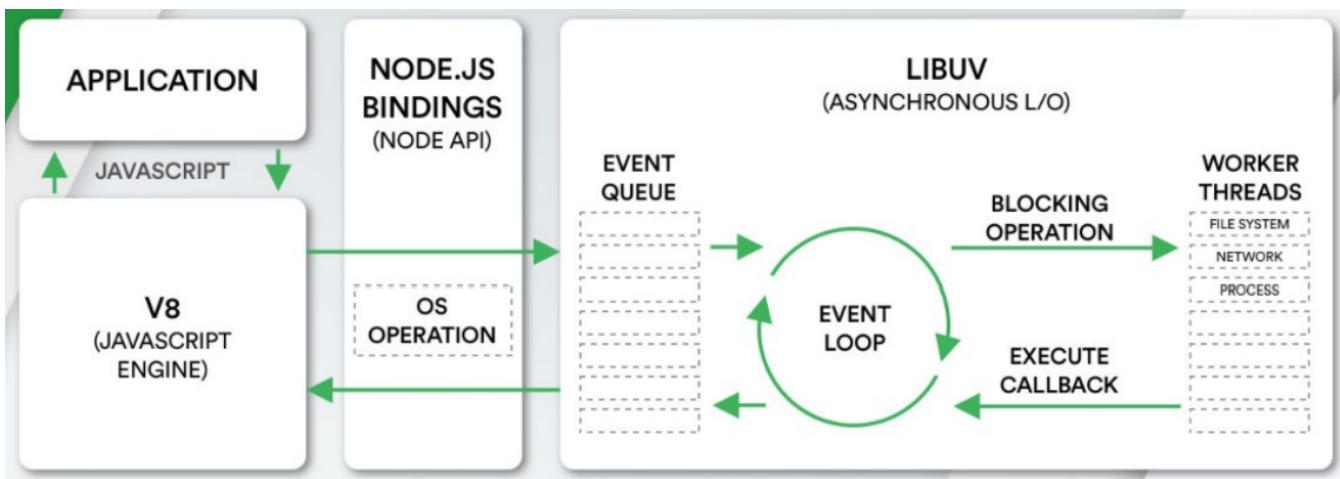
```
Promise.resolve().then(() => {
  console.log(0);
  // 1.直接return一个值 相当于resolve(4)
  // return 4

  // 2.return thenable的值 多加一次微任务
  return {
    then: function(resolve) {
      // 大量的计算
      resolve(4)
    }
  }

  // 3.return Promise
  // 不是普通的值，多加一次微任务
  // Promise.resolve(4)，多加一次微任务
  // 一共多加两次微任务
  return Promise.resolve(4)
}).then((res) => {
  console.log(res)
})
```

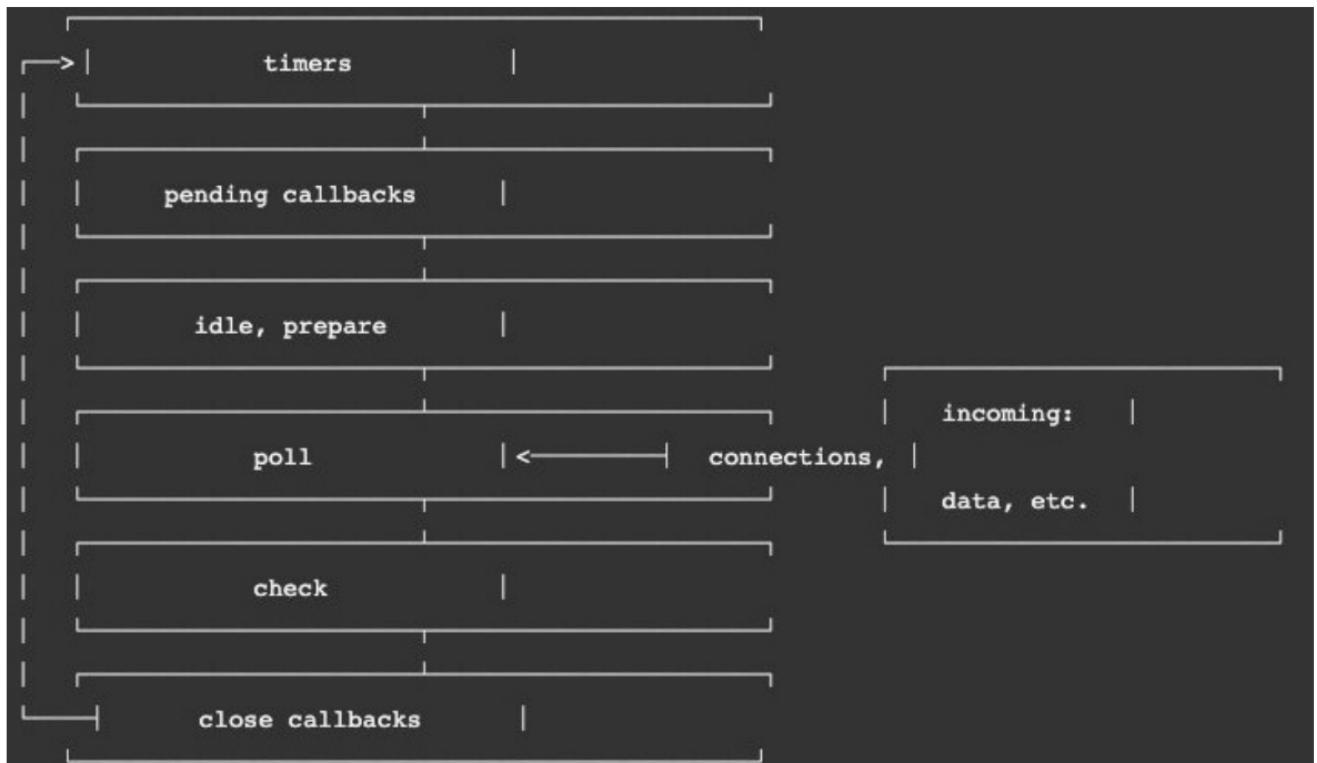
Node的事件循环

- 浏览器中的 EventLoop 是根据HTML5定义的规范来实现的，不同的浏览器可能会有不同的实现，而 Node 中是由libuv实现的
- 这里我们来给出一个 Node 的架构图：
 - libuv中主要维护了一个 EventLoop 和 worker threads (线程池)
 - EventLoop 负责调用系统的一些其他操作：文件的IO、 Network、 child-processes 等
 - libuv是一个多平台的专注于 异步IO 的库，它最初是为Node开发的，但是现在也被使用到Luvit、Julia、pyuv等其他地方



Node事件循环的阶段

- 事件循环像是一个桥梁，是连接着应用程序的JavaScript和系统调用之间的通道：
 - 无论是我们的文件IO、数据库、网络IO、定时器、子进程，在完成对应的操作后，都会将对应的结果和回调函数放到事件循环（任务队列）中；
 - 事件循环会不断的从任务队列中取出对应的事件（回调函数）来执行
- 但是一次完整的事件循环Tick分成很多个阶段：
 - 定时器 (Timers)：本阶段执行已经被 `setTimeout()` 和 `setInterval()` 的调度回调函数。
 - 待定回调 (Pending Callback)：对某些系统操作（如TCP错误类型）执行回调，比如TCP连接时接收到ECONNREFUSED
 - idle, prepare：仅系统内部使用
 - 轮询 (Poll)：检索新的 I/O 事件；执行与 I/O 相关的回调
 - 检测 (check)：`setImmediate()` 回调函数在这里执行
 - 关闭的回调函数：一些关闭的回调函数，如：`socket.on('close', ...)`



Node的宏任务和微任务

- 现从一次事件循环的Tick来说，Node的事件循环更复杂，它也分为微任务和宏任务：
 - 宏任务 (macrotask) : setTimeout、setInterval、IO事件、setImmediate、close事件
 - 微任务 (microtask) : Promise的then回调、process.nextTick、queueMicrotask
- 但是，Node中的事件循环不只是微任务队列和宏任务队列：
 - 微任务队列：
 - next tick queue: process.nextTick
 - other queue: Promise的then回调、queueMicrotask
 - 宏任务队列：
 - timer queue: setTimeout、setInterval
 - poll queue: IO事件
 - check queue: setImmediate
 - close queue: close事件

Node事件循环的顺序

在每一次事件循环的tick中，会按照如下顺序来执行代码：

1. next tick microtask queue
2. other microtask queue
3. timer queue
4. poll queue
5. check queue
6. close queue

错误处理方案

throw关键字

- throw 表达式就是在 throw 后面可以跟上一个表达式来表示具体的异常信息：
- throw 关键字可以跟上：
 - 基本数据类型：比如 number、string、Boolean
 - 对象类型：对象类型可以包含更多的信息

Error类型

- Error包含三个属性：
 - message : 创建Error对象时传入的 message
 - name : Error的名称，通常和类的名称一致

- stack：整个Error的错误信息，包括函数的调用栈，当我们直接打印Error对象时，打印的就是 stack
- Error有一些自己的子类：
 - RangeError：下标值越界时使用的错误类型
 - SyntaxError：解析语法错误时使用的错误类型
 - TypeError：出现类型错误时，使用的错误类型

异常的处理

- 一个函数抛出了异常，调用它的时候程序会被强制终止：这是因为如果我们在调用一个函数时，这个函数抛出了异常，但是我们并没有对这个异常进行处理，那么这个异常会继续传递到上一个函数调用中
- 而如果到了最顶层（全局）的代码中依然没有对这个异常的处理代码，这个时候就会报错并且终止程序的运行

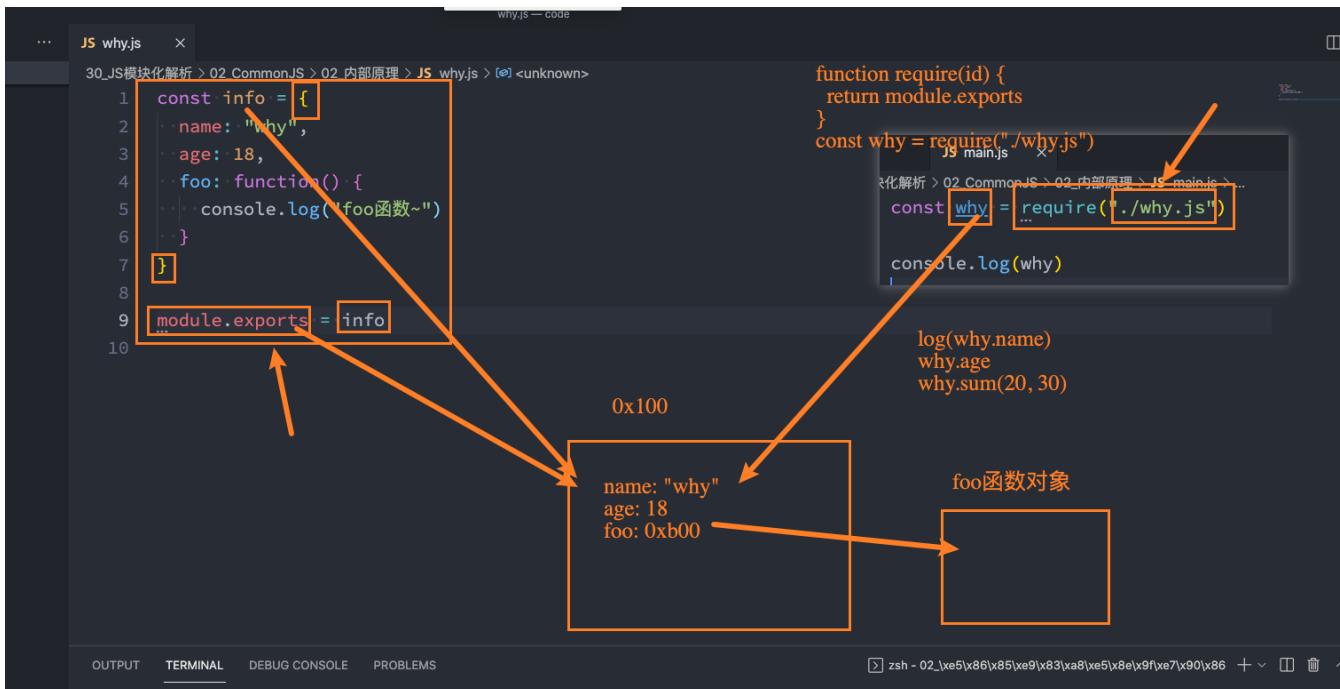
异常的捕获

- 在ES10 (ES2019) 中， catch 后面绑定的 error 可以省略。
- 如果有一些必须要执行的代码，我们可以使用 finally 来执行：
 - finally 表示最终一定会被执行的代码结构；
 - 注意：如果 try 和 finally 中都有返回值，那么会使用 finally 当中的返回值

模块化

CommonJS规范和Node关系

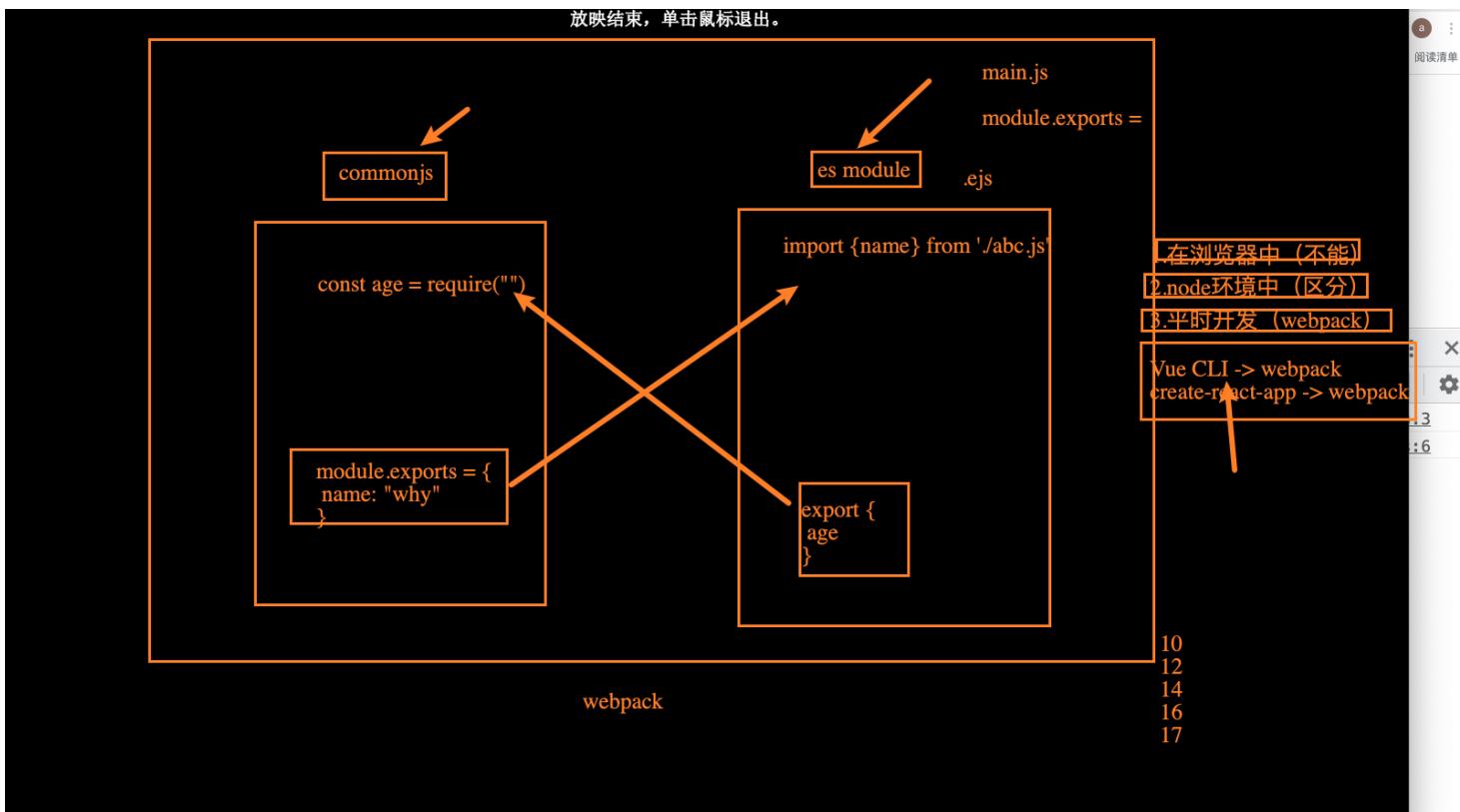
- CommonJS是一个规范，最初提出来是在浏览器以外的地方使用，并且当时被命名为ServerJS，后来为了体现它的广泛性，修改为CommonJS，平时我们也会简称为CJS。
 - Node是CommonJS在服务器端一个具有代表性的实现
 - Browserify是CommonJS在浏览器中的一种实现
 - webpack打包工具具备对CommonJS的支持和转换
- 所以，Node中对CommonJS进行了支持和实现，让我们在开发node的过程中可以方便的进行模块化开发：
 - 在Node中每一个js文件都是一个单独的模块
 - 这个模块中包括CommonJS规范的核心变量： exports 、 module.exports 、 require
 - 我们可以使用这些变量来方便的进行模块化开发
- 模块化的核心是导出和导入，Node中对其进行了实现：
 - exports 和 module.exports 可以负责对模块中的内容进行导出
 - require 函数可以帮助我们导入其他模块（自定义模块、系统模块、第三方库模块）中的内容



exports导出

注意：`exports`是一个对象，我们可以在这个对象中添加很多个属性，添加的属性会导出

- `module.exports` 和 `exports` 关系
- 维基百科中对CommonJS规范的解析：
 - CommonJS中是没有 `module.exports` 的概念的
- 但是为了实现模块的导出，Node中使用的是 `Module` 的类，每一个模块都是 `Module` 的一个实例，也就是 `module`
 - 所以在Node中真正用于导出的其实根本不是 `exports`，而是 `module.exports`
 - 因为 `module` 才是导出的真正实现者
- 为什么 `exports` 也可以导出呢？
 - 这是因为 `module` 对象的 `exports` 属性是 `exports` 对象的一个引用
 - 也就是说 `module.exports = exports = main` 中的 `bar`



require细节

`require` 是一个函数，可以帮助我们引入一个文件（模块）中导出的对象

`require` 的查找规则 <https://nodejs.org/dist/latest-v14.x/docs/api/modules.html>

- 导入格式如下: `require(X)`
- 情况一: `X` 是一个Node核心模块，比如 `path`、`http`
 - 直接返回核心模块，并且停止查找
- 情况二: `X` 是以 `./` 或 `../` 或 `/` (根目录) 开头的
 - 第一步: 将 `X` 当做一个文件在对应的目录下查找
 - 如果有后缀名，按照后缀名的格式查找对应的文件
 - 如果没有后缀名，会按照如下顺序
 - 直接查找文件 `X`
 - 查找 `X.js` 文件
 - 查找 `X.json` 文件
 - 查找 `X.node` 文件
 - 第二步: 没有找到对应的文件，将 `X` 作为一个目录
 - 查找目录下面的 `index` 文件
 - 查找 `X/index.js` 文件
 - 查找 `X/index.json` 文件
 - 查找 `X/index.node` 文件
 - 如果没有找到，那么报错: `not found`

- 情况三：直接是一个X（没有路径），并且X不是一个核心模块
 - 逐层向上查找
 - 如果上面的路径中都没有找到，那么报错：not found

模块的加载过程

- 结论一：模块在被第一次引入时，模块中的js代码会被运行一次
- 结论二：模块被多次引入时，会缓存，最终只加载（运行）一次
 - 因为每个模块对象 `module` 都有一个属性：`loaded`
 - 为 `false` 表示还没有加载，为 `true` 表示已经加载
- 结论三：如果有循环引入，那么加载顺序是图结构
 - 图结构在遍历的过程中，有深度优先搜索（DFS，depth first search）和广度优先搜索（BFS，breadth first search）
 - Node采用的是深度优先算法

CommonJS规范缺点

CommonJS加载模块是同步的：

- 同步的意味着只有等到对应的模块加载完毕，当前模块中的内容才能被运行
- 这个在服务器不会有什么问题，因为服务器加载的js文件都是本地文件，加载速度非常快

如果将它应用于浏览器呢？

- 浏览器加载js文件需要先从服务器将文件下载下来，之后再加载运行
- 那么采用同步的就意味着后续的js代码都无法正常运行，即使是一些简单的DOM操作

所以在浏览器中，我们通常不使用CommonJS规范

- 在webpack中使用CommonJS，会将我们的代码转成浏览器可以直接执行的代码
- 在早期为了可以在浏览器中使用模块化，通常会采用 AMD 或 CMD
 - 前一方面现代的浏览器已经支持ES Modules，另一方面借助于webpack等工具可以实现对CommonJS或者ES Module代码的转换
 - AMD和CMD目前使用非常少

AMD规范

- AMD主要是应用于浏览器的一种模块化规范
 - AMD是Asynchronous Module Definition（异步模块定义）的缩写
 - 采用的是异步加载模块
 - 上AMD的规范还要早于CommonJS，但是CommonJS目前依然在被使用，而AMD使用的较少
- 规范只是定义代码的应该如何去编写
 - AMD实现的比较常用的库是 `require.js` 和 `curl.js`

require.js的使用

- 第一步：下载 require.js
 - 下载地址：<https://github.com/requirejs/requirejs>
 - 找到其中的 require.js 文件
- 第二步：定义HTML的script标签引入require.js和定义入口文件：
 - data-main属性的作用是在加载完src的文件后会加载执行该文件

```
<script src="./lib/require.js" data-main="./index.js"></script>
```

CMD规范

- CMD规范也是应用于浏览器的一种模块化规范：
 - CMD 是Common Module Definition（通用模块定义）的缩写
 - 它也采用了异步加载模块，但是它将CommonJS的优点吸收了过来
 - 但是目前CMD使用也非常少了
- CMD也有自己比较优秀的实现方案：
 - SeaJS

SeaJS的使用

- 第一步：下载 SeaJS
 - 下载地址：<https://github.com/seajs/seajs>
 - 找到dist文件夹下的 sea.js
- 第二步：引入 sea.js 和使用主入口文件
 - seajs 是指定主入口文件的

ES Module

| 采用ES Module将自动采用严格模式：use strict

import meta

- import.meta 是一个给JavaScript模块暴露特定上下文的元数据属性的对象。
 - 它包含了这个模块的信息，比如说这个模块的URL
 - 在ES11 (ES2020) 中新增的特性

ES Module的解析流程

- ES Module是如何被浏览器解析并且让模块之间可以相互引用的呢？
 - <https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/>
- ES Module的解析过程可以划分为三个阶段：
 - 阶段一：构建 (Construction)，根据地址查找js文件，并且下载，将其解析成模块记录 (Module Record)；

- 阶段二：实例化（Instantiation），对模块记录进行实例化，并且分配内存空间，解析模块的导入和导出语句，把模块指向对应的内存地址。
- 阶段三：运行（Evaluation），运行代码，计算值，并且将值填充到内存地址中

包管理工具详解 npm、yarn、cnpm、npx

npm

- 每一个项目都会有一个对应的配置文件 `package.json` (`npm init -y`)
 - 这个配置文件会记录着你项目的名称、版本号、项目描述等
 - 也会记录着你项目所依赖的其他库的信息和依赖库的版本号
- 安装的依赖版本出现：`^2.0.3`或`~2.0.3`，这是什么意思呢？
 - npm的包通常需要遵从semver版本规范：
 - semver: <https://semver.org/lang/zh-CN/>
 - npm semver: <https://docs.npmjs.com/misc/semver>
 - semver版本规范是X.Y.Z：
 - X主版本号 (major)：当你做了不兼容的 API 修改（可能不兼容之前的版本）
 - Y次版本号 (minor)：当你做了向下兼容的功能性新增（新功能增加，但是兼容之前的版本）
 - Z修订号 (patch)：当你做了向下兼容的问题修正（没有新功能，修复了之前版本的bug）
 - ^和~的区别：
 - `^x.y.z`：表示x是保持不变的，y和z永远安装最新的版本
 - `~x.y.z`：表示x和y保持不变的，z永远安装最新的版本

常见的属性

- 必须填写的属性：`name`、`version`
 - `name` 是项目的名称
 - `version` 是当前项目的版本号
 - `description` 是描述信息，很多时候是作为项目的基本描述
 - `author` 是作者相关信息（发布时用到）
 - `license` 是开源协议（发布时用到）
- `private` 属性：
 - `private` 属性记录当前的项目是否是私有的
 - 当值为 `true` 时，npm是不能发布它的，这是防止私有项目或模块发布出去的方式
- `main` 属性：
 - 设置程序的入口。
 - 在发布一个模块的时候会用到
 - 比如我们使用 `axios` 模块 `const axios = require('axios')`
 - 实际上是找到对应的 `main` 属性查找文件的
- `scripts` 属性

- `scripts` 属性用于配置一些脚本命令，以键值对的形式存在；
- 配置后我们可以通过 `npm run` 命令的key来执行这个命令；
- `npm start` 和 `npm run start` 是等价的
 - 对于常用的 `start`、`test`、`stop`、`restart` 可以省略掉 `run` 直接通过 `npm start` 等方式运行；
- `dependencies` 属性
 - `dependencies` 属性是指定无论开发环境还是生成环境都需要依赖的包
 - 通常是我们项目实际开发用到的一些库模块 `vue`、`vuex`、`vue-router`、`react`、`react-dom`、`axios` 等等
- `devDependencies` 属性
 - 一些包在生成环境是不需要的，比如 `webpack`、`babel` 等；
 - 这个时候我们会通过 `npm install webpack --save-dev`，将它安装到 `devDependencies` 属性中
- `peerDependencies` 属性
 - 还有一种项目依赖关系是对等依赖，也就是你依赖的一个包，它必须是以另外一个宿主包为前提的
 - 比如 `element-plus` 是依赖于 `vue3` 的，`ant design` 是依赖于 `react`、`react-dom`
- `engines` 属性
 - `engines` 属性用于指定Node和NPM的版本号
 - 在安装的过程中，会先检查对应的引擎版本，如果不符合就会报错
 - 事实上也可以指定所在的操作系统 `"os" : ["darwin", "linux"]`，只是很少用到
- `browserslist` 属性
 - 用于配置打包后的JavaScript浏览器的兼容情况，参考
 - 否则我们需要手动的添加 `polyfills` 来让支持某些语法
 - 也就是说它是为 `webpack` 等打包工具服务的一个属性

npm install

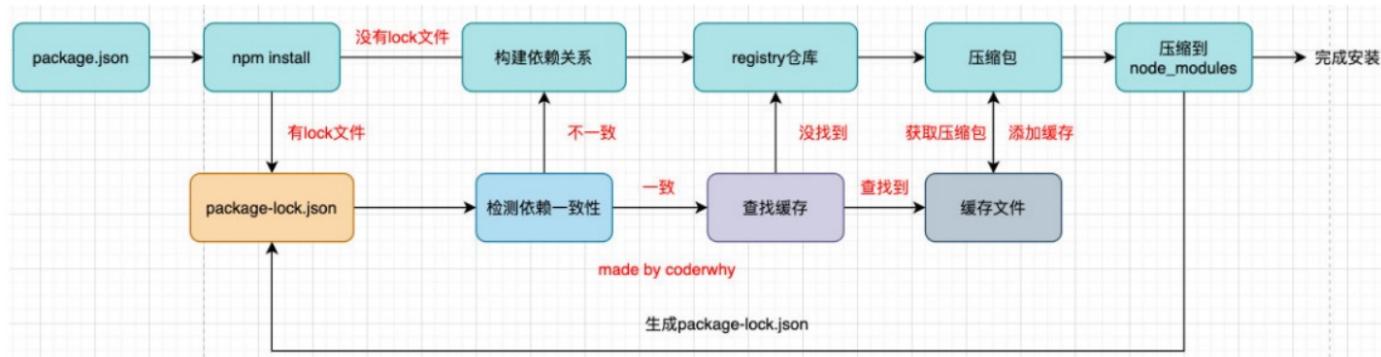
- 安装npm包分两种情况：
 - 全局安装 (`global install`)： `npm install webpack -g`
 - 项目 (局部) 安装 (`local install`)： `npm install webpack`
- 全局安装
 - 全局安装是直接将某个包安装到全局：
 - 比如yarn的全局安装：
 - 但是很多人对全局安装有一些误会：
 - 通常使用npm全局安装的包都是一些工具包： `yarn`、`webpack` 等
 - 并不是类似于 `axios`、`express`、`koa` 等库文件
 - 所以全局安装了之后并不能让我们在所有的项目中使用 `axios` 等库
- 局部安装分为开发时依赖和生产时依赖：

```

# 安装开发和生产依赖
npm install axios
npm i axios
# 开发依赖
npm install webpack --save-dev
npm install webpack -D
npm i webpack -D
# 根据package.json中的依赖包
npm install

```

npm install 原理



- npm install 会检测是有 package-lock.json 文件:
 - 没有lock文件
 - 分析依赖关系，这是因为我们可能包会依赖其他的包，并且多个包之间会产生相同依赖的情况
 - 从 registry 仓库中下载压缩包（如果我们设置了镜像，那么会从镜像服务器下载压缩包）
 - 获取到压缩包后会对压缩包进行缓存（从npm5开始有的）
 - 将压缩包解压到项目的 node_modules 文件夹中（前面我们讲过，require的查找顺序会在该包下面查找）
 - 有lock文件
 - 检测lock中包的版本是否和 package.json 中一致（会按照semver版本规范检测）
 - 不一致，那么会重新构建依赖关系，直接会走顶层的流程
 - 一致的情况下，会去优先查找缓存
 - 没有找到，会从 registry 仓库下载，直接走顶层流程
 - 查找到，会获取缓存中的压缩文件，并且将压缩文件解压到 node_modules 文件夹中

package-lock.json

- package-lock.json 文件解析：
 - name：项目的名称
 - version：项目的版本
 - lockfileVersion：lock文件的版本
 - requires：使用requires来跟踪模块的依赖关系
 - dependencies：项目的依赖
 - 当前项目依赖axios，但是axios依赖 follow-redirects

- axios 中的属性如下：
 - version 表示实际安装的axios的版本
 - resolved 用来记录下载的地址， registry 仓库中的位置
 - requires 记录当前模块的依赖
 - integrity 用来从缓存中获取索引，再通过索引去获取压缩包文件

npm其他命令

<https://docs.npmjs.com/cli/v8/commands>

- 卸载某个依赖包：

```
npm uninstall package
npm uninstall package --save-dev
npm uninstall package -D
```

- 强制重新build

```
npm rebuild
```

- 清除缓存

```
npm cache clean
```

yarn工具

yarn是由Facebook、Google、Exponent 和 Tilde 联合推出了一个新的JS包管理工具，为了弥补 npm 的一些缺陷而出现

cnpm工具

- 查看npm镜像：

```
npm config get registry # npm config get registry
```

- 直接设置npm的镜像：

```
npm config set registry https://registry.npm.taobao.org
```

- 如果并不希望将npm镜像修改
 - 可以使用cnpm，并且将cnpm设置为淘宝的镜像：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
cnpm config get registry # https://r.npm.taobao.org/
```

npx工具

- npx是npm5.2之后自带的一个命令。
 - npx的作用非常多，但是比较常见的是使用它来调用项目中的某个模块的指令
- 以webpack为例：
 - 全局安装的是webpack5.1.3
 - 项目安装的是webpack3.6.0
- 如果在终端执行 webpack --version
 - 显示结果会是 webpack 5.1.3，事实上使用的是全局的
 - 原因：在当前目录下找不到webpack时，就会去全局找，并且执行命令

局部命令的执行

- 如何使用项目（局部）的webpack，常见的是两种方式：
 - 方式一：明确查找到node_module下面的webpack
 - 方式二：在 scripts定义脚本，来执行webpack
- 方式一：在终端中使用如下命令（在项目根目录下）

```
./node_modules/.bin/webpack --version
```

- 方式二：修改package.json中的scripts

```
"scripts": {
  "webpack": "webpack --version"
}
```

- 方式三：使用npx

```
npx webpack --version
```

- npx的原理非常简单，它会到当前目录的node_modules/.bin目录下查找对应的命令

npm发布自己的包

- 注册npm账号：
 - <https://www.npmjs.com/>
 - 选择 sign up
- 在命令行登录：

```
npm login
```

- 修改 package.json
- 发布到 npm registry 上

```
npm publish
```

- 更新仓库：
 - 1. 修改版本号(最好符合semver规范)
 - 2. 重新发布

JSON-数据存储

- JSON的由来

JSON是一种非常重要的数据格式，全称是JavaScript Object Notation（JavaScript对象符号）

- 其他的传输格式：
 - XML：在早期的网络传输中主要是使用XML来进行数据交换的，但是这种格式在解析、传输等各方面都弱于JSON，所以目前已经很少在被使用了；
 - Protobuf：另外一个在网络传输中目前已经越来越多使用的传输格式是protobuf，但是直到2021年的3.x版本才支持JavaScript，所以目前在前端使用的较少

JSON基本语法

- JSON的顶层支持三种类型的值：

1. 简单值：数字（Number）、字符串（String，不支持单引号）、布尔类型（Boolean）、null类型（不支持undefined）
2. 对象值：由key、value组成，key是字符串类型，并且必须添加双引号，值可以是简单值、对象值、数组值
3. 数组值：数组的值可以是简单值、对象值、数组值

JSON序列化

- 某些情况下我们希望将JavaScript中的复杂类型转化成JSON格式的字符串，这样方便对其进行处理：
 - 比如我们希望将一个对象保存到localStorage中
 - 但是如果我们将一个对象直接存放在一个对象，这个对象会被转化成[object Object]格式的字符串，并不是我们想要的结果
- 在ES5中引用了JSON全局对象，该对象有两个常用的方法：
 - stringify方法：将JavaScript类型转成对应的JSON字符串

- `parse` 方法：解析JSON字符串，转回对应的JavaScript类型

Stringify的参数

JSON.stringify() 方法将一个 JavaScript 对象或值转换为 JSON 字符串：

- Stringify的参数 `replacer`

- 如果指定了一个 `replacer` 函数，则可以选择性地替换值

```
JSON.stringify(obj, (key, val) => val)
```

- 如果指定的 `replacer` 是数组，则可选择性地仅包含数组指定的属性

```
JSON.stringify(obj, ['name', 'age'])
```

- Stringify的参数 `space`

- 格式化JSON，前面拼接上第三个参数的内容

```
JSON.stringify(obj, null, '---')
```

- 如果对象本身包含 `toJSON` 方法，那么会直接使用 `toJSON` 方法的结果

parse方法

JSON.parse() 方法用来解析JSON字符串，构造由字符串描述的JavaScript值或对象

- 提供可选的 `reviver` 函数用以在返回之前对所得到的对象执行变换(操作)

使用JSON序列化深拷贝

- JSON.stringify() 生成的新对象和之前的对象并不是同一个对象
 - 相当于是进行了一次深拷贝
- 这种方法对函数无能为力
 - 因为 `stringify` 并不会对函数进行处理

Storage/indexedDB/cookie

Storage

- WebStorage主要提供了一种机制，可以让浏览器提供一种比 cookie 更直观的 key、 value 存储方式：
- `localStorage`：本地存储，提供的是一种永久性的存储方法，在关闭掉网页重新打开时，存储的内容依然保留；
- `sessionStorage`：会话存储，提供的是本次会话的存储，在关闭掉会话时，存储的内容会被清除

localStorage和sessionStorage的区别

- 验证一：关闭网页后重新打开，`localStorage` 会保留，而 `sessionStorage` 会被删除
- 验证二：在页面内实现跳转，`localStorage` 会保留，`sessionStorage` 也会保留
- 验证三：在页面外实现跳转（打开新的网页），`localStorage` 会保留，`sessionStorage` 不会被保留

Storage常见的方法和属性

- 属性：
 - `Storage.length`：只读属性
 - 返回一个整数，表示存储在Storage对象中的数据项数量
- 方法：
 - `Storage.key()`：该方法接受一个数值 `n` 作为参数，返回存储中的第`n`个 `key` 名称
 - `Storage.getItem()`：该方法接受一个 `key` 作为参数，并且返回 `key` 对应的 `value`
 - `Storage.setItem()`：该方法接受一个 `key` 和 `value`，并且将会把 `key` 和 `value` 添加到存储中
 - 如果 `key` 存储，则更新其对应的值
 - `Storage.removeItem()`：该方法接受一个 `key` 作为参数，并把该 `key` 从存储中删除
 - `Storage.clear()`：该方法的作用是清空存储中的所有 `key`

封装Storage

```

class HYCache {
  constructor(isLocal = true) {
    this.storage = isLocal ? localStorage: sessionStorage
  }

  setItem(key, value) {
    if (value) {
      this.storage.setItem(key, JSON.stringify(value))
    }
  }

  getItem(key) {
    let value = this.storage.getItem(key)
    if (value) {
      value = JSON.parse(value)
      return value
    }
  }

  removeItem(key) {
    this.storage.removeItem(key)
  }

  clear() {
    this.storage.clear()
  }

  key(index) {
    return this.storage.key(index)
  }

  length() {
    return this.storage.length
  }
}

const localCache = new HYCache()
const sessionCache = new HYCache(false)

export {
  localCache,
  sessionCache
}

```

IndexedDB

- 什么是 IndexedDB 呢?
 - DB这个词，说明它其实是一种数据库（Database），通常情况下在服务器端比较常见
 - 在实际的开发中，大量的数据都是存储在数据库的，客户端主要是请求这些数据并且展示
 - 有时候我们可能会存储一些简单的数据到本地（浏览器中），比如token、用户名、密码、用户信息等，比较少存储大量的数据

- 那么如果确实有**大量的**数据需要存储，这个时候可以选择使用 IndexedDB
- IndexedDB 是一种**底层的**API，用于在客户端存储大量的结构化数据。
 - 它是一种事务型数据库系统，是一种基于JavaScript面向对象数据库，有点类似于NoSQL（非关系型数据库）
 - IndexDB本身就是基于事务的，我们只需要指定数据库模式，打开与数据库的连接，然后检索和更新一系列事务即可

IndexedDB的连接数据库

- 第一步：打开IndexedDB的某一个数据库
 - 通过 `IndexedDB.open(数据库名称, 数据库版本)` 方法
 - 如果数据库不存在，那么会创建这个数据
 - 如果数据库已经存在，那么会打开这个数据库
- 第二步：通过监听回调得到数据库连接结果
 - 数据库的 `open` 方法会得到一个 `IDBOpenDBRequest` 类型
 - 我们可以通过下面的三个回调来确定结果：
 - `onerror`：当数据库连接失败时
 - `onsuccess`：当数据库连接成功时回调
 - `onupgradeneeded`：当数据库的version发生变化并且高于之前版本时回调
 - 通常我们在这里会创建具体的存储对
象：`db.createObjectStore(存储对象名称, { keypath: 存储的主键 })`
 - 我们可以通过 `onsuccess` 回调的 `event` 获取到db对象：`event.target.result`

IndexedDB的数据库操作

- 我们对数据库的操作要通过事务对象来完成：
 - 第一步：通过db获取对应存储的事务 `db.transaction(存储名称, 可写操作)`
 - 第二步：通过事务获取对应的存储对象 `transaction.objectStore(存储名称)`
- 增删改查操作：
 - 新增数据 `store.add`
 - 查询数据
 - 方式一： `store.get(key)`
 - 方式二：通过 `store.openCursor` 拿到游标对象
 - 在 `request.onsuccess` 中获取 `cursor: event.target.result`
 - 获取对应的 `key: cursor.key`
 - 获取对应的 `value: cursor.value`
 - 可以通过 `cursor.continue` 来继续执行
 - 修改数据 `cursor.update(value)`
 - 删除数据 `cursor.delete()`

```
// 打开数据(和数据库建立连接)
const dbRequest = indexedDB.open("why", 3)
dbRequest.onerror = function(err) {
  console.log("打开数据库失败~")
}
let db = null
dbRequest.onsuccess = function(event) {
  db = event.target.result
}
// 第一次打开/或者版本发生升级
dbRequest.onupgradeneeded = function(event) {
  const db = event.target.result

  console.log(db)

  // 创建一些存储对象
  // keyPath => 表的主键
  db.createObjectStore("users", { keyPath: "id" })
}

class User {
  constructor(id, name, age) {
    this.id = id
    this.name = name
    this.age = age
  }
}

const users = [
  new User(100, "why", 18),
  new User(101, "kobe", 40),
  new User(102, "james", 30),
]

// 获取bt�s, 监听点击
const bt�s = document.querySelectorAll("button")
for (let i = 0; i < bt�s.length; i++) {
  bt�s[i].onclick = function() {
    const transaction = db.transaction("users", "readwrite")
    console.log(transaction)
    const store = transaction.objectStore("users")

    switch(i) {
      case 0:
        console.log("点击了新增")
        for (const user of users) {
          const request = store.add(user)
          request.onsuccess = function() {
            console.log(`${user.name}插入成功`)
          }
        }
        transaction.oncomplete = function() {
          console.log("添加操作全部完成")
        }
    }
  }
}
```

```
break
case 1:
  console.log("点击了查询")

  // 1.查询方式一(知道主键，根据主键查询)
  // const request = store.get(102)
  // request.onsuccess = function(event) {
  //   console.log(event.target.result)
  // }

  // 2.查询方式二:
  const request = store.openCursor()
  request.onsuccess = function(event) {
    const cursor = event.target.result
    if (cursor) {
      if (cursor.key === 101) {
        console.log(cursor.key, cursor.value)
      } else {
        cursor.continue()
      }
    } else {
      console.log("查询完成")
    }
  }
  break
case 2:
  console.log("点击了删除")
  const deleteRequest = store.openCursor()
  deleteRequest.onsuccess = function(event) {
    const cursor = event.target.result
    if (cursor) {
      if (cursor.key === 101) {
        cursor.delete()
      } else {
        cursor.continue()
      }
    } else {
      console.log("查询完成")
    }
  }
  break
case 3:
  console.log("点击了修改")
  const updateRequest = store.openCursor()
  updateRequest.onsuccess = function(event) {
    const cursor = event.target.result
    if (cursor) {
      if (cursor.key === 101) {
        const value = cursor.value;
        value.name = "curry"
        cursor.update(value)
      } else {
        cursor.continue()
      }
    }
  }
}
```

```
        } else {
            console.log("查询完成")
        }
    }
break
}
}
}
```

Cookie

Cookie (复数形态Cookies) , 类型为“小型文本文件，某些网站为了辨别用户身份而存储在用户本地终端 (Client Side) 上的数据。

浏览器会在特定的情况下携带上cookie来发送请求，我们可以通过cookie来获取一些信息

- Cookie总是保存在客户端中，按在客户端中的存储位置，Cookie可以分为内存Cookie和硬盘Cookie。
 - 内存Cookie由浏览器维护，保存在内存中，浏览器关闭时Cookie就会消失，其存在时间是短暂的
 - 硬盘Cookie保存在硬盘中，有一个过期时间，用户手动清理或者过期时间到时，才会被清理
- 如果判断一个cookie是内存cookie还是硬盘cookie呢？
 - 没有设置过期时间，默认情况下cookie是**内存cookie**，在关闭浏览器时会自动删除
 - 有设置过期时间，并且过期时间不为0或者负数的cookie，是**硬盘cookie**，需要手动或者到期时，才会删除

cookie常见的属性

- cookie的生命周期：
 - 默认情况下的cookie是**内存cookie**，也称之为**会话cookie**，也就是在浏览器关闭时会自动被删除
 - 我们可以通过设置 `expires` 或者 `max-age` 来设置过期的时间
 - `expires`：设置的是 `Date.toUTCString()`，设置格式是 `;expires=date-in-GMTString-format`
 - `max-age`：设置过期的秒钟 `;max-age=max-age-in-seconds` (例如一年为 $606024*365$)
- cookie的作用域：(允许cookie发送给哪些URL)
- Domain：指定哪些主机可以接受cookie
 - 如果不指定，那么默认是 `origin`，不包括子域名。
 - 如果指定Domain，则包含子域名。例如，如果设置 `Domain=mozilla.org`，则 Cookie 也包含在子域名中 (如 `developer.mozilla.org`)。
- Path：指定主机下哪些路径可以接受cookie
 - 例如，设置 `Path=/docs`，则以下地址都会匹配：
 - `/docs`
 - `/docs/Web/`
 - `/docs/Web/HTTP`

客户端设置cookie

- js直接设置和获取cookie：

```
console.log(document.cookie)
```

- 这个cookie会在会话关闭时被删除掉
 - 设置过期时间就是本地cookie，未设置就是内存cookie
- 设置cookie，同时设置过期时间（默认单位是秒钟）

```
document.cookie = 'name=hidari;max-age=10'
```

cookie 缺点

1. 将 cookie 附加到每次http请求中
2. 明文传输，存在安全风险
3. 有大小限制（4kb）
4. 客户端cookie浏览器自动添加 => 服务器，ios，安卓，小程序 => 需要手动添加 cookie

BOM 和 DOM

BOM

JavaScript有一个非常重要的运行环境就是浏览器，而且浏览器本身又作为一个应用程序需要对其本身进行操作，所以通常浏览器会有对应的对象模型（BOM，Browser Object Model）。

我们可以将BOM看成是连接JavaScript脚本与浏览器窗口的桥梁。

- BOM主要包括一下的对象模型：
 - window：包括全局属性、方法，控制浏览器窗口相关的属性、方法
 - location：浏览器连接到的对象的位置（URL）
 - history：操作浏览器的历史
 - document：当前窗口操作文档的对象
 - window对象在浏览器中有两个身份：
 - 身份一：全局对象。
 - 我们知道ECMAScript其实是一个全局对象的，这个全局对象在Node中是global
 - 在浏览器中就是window对象
 - 身份二：浏览器窗口对象。
 - 作为浏览器窗口时，提供了对浏览器操作的相关的API

Window

1. 全局对象
- 在浏览器中，window对象就是之前经常提到的全局对象，也就是我们之前提到过GO对象
 - 比如在全局通过var声明的变量，会被添加到GO中，也就是会被添加到window上

- 比如window默认给我们提供了全局的函数和类： setTimeout、 Math、 Date、 Object等
- 通过var声明的变量， 全局提供的类和方法
- window对象作为JavaScript语言本身所拥有的一些特性

2. 窗口对象

<https://developer.mozilla.org/zh-CN/docs/Web/API/Window>

- 第一：包含大量的属性， localStorage、 console、 location、 history、 screenX、 scrollX 等等（大概60+个属性）
- 第二：包含大量的方法， alert、 close、 scrollTo、 open 等等（大概40+个方法）
- 第三：包含大量的事件， focus、 blur、 load、 hashchange 等等（大概30+个事件）
- 第四：包含从EventTarget继承过来的方法， addEventListener、 removeEventListener、 dispatchEvent 方法

3. 常见的属性、方法、事件

```

// 1.常见的属性
console.log(window.screenX)
console.log(window.screenY)

window.addEventListener("scroll", () => {
  console.log(window.scrollX, window.scrollY)
})

console.log(window.outerHeight)
console.log(window.innerHeight)

// 2.常见的方法
const scrollBtn = document.querySelector("#scroll")
scrollBtn.onclick = function() {
  // 1.scrollTo
  window.scrollTo({ top: 2000 })

  // 2.close
  window.close()

  // 3.open
  window.open("http://www.baidu.com", "_self")
}

// 3.常见的事件
window.onload = function() {
  console.log("window窗口加载完毕~")
}

window.onfocus = function() {
  console.log("window窗口获取焦点~")
}

window.onblur = function() {
  console.log("window窗口失去焦点~")
}

const hashChangeBtn = document.querySelector("#hashchange")
hashChangeBtn.onclick = function() {
  location.hash = "aaaa"
}
window.onhashchange = function() {
  console.log("hash发生了改变")
}

```

EventTarget

默认事件监听: <https://developer.mozilla.org/zh-CN/docs/Web/Events>

- Window继承自EventTarget，所以会继承其中的属性和方法：
 - addEventListener：注册某个事件类型以及事件处理函数

- removeEventListener : 移除某个事件类型以及事件处理函数
- dispatchEvent : 派发某个事件类型到EventTarget上

```
const clickHandler = () => {
  console.log("window发生了点击")
}

window.addEventListener("click", clickHandler)
window.removeEventListener("click", clickHandler)

window.addEventListener("coderwhy", () => {
  console.log("监听到了coderwhy事件")
})

window.dispatchEvent(new Event("coderwhy"))
```

Location对象

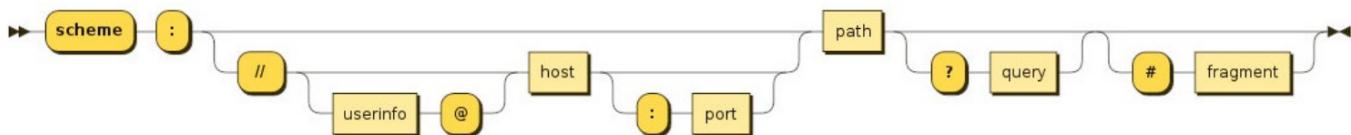
1. 常见的属性

| Location对象用于表示window上当前链接到的URL信息。

- 常见的属性：

- href : 当前window对应的超链接URL, 整个URL
- protocol : 当前的协议
- host : 主机地址
- hostname : 主机地址(不带端口)
- port : 端口
- pathname : 路径
- search : 查询字符串
- hash : 哈希值
- username : URL中的username (很多浏览器已经禁用)
- password : URL中的password (很多浏览器已经禁用)

2. 常见的方法



- assign : 赋值一个新的URL, 并且跳转到该URL中
- replace : 打开一个新的URL, 并且跳转到该URL中 (不同的是不会在浏览记录中留下之前的记录)
- reload : 重新加载页面, 可以传入一个Boolean类型

```
console.log(window.location)

// 当前的完整的url地址
console.log(location.href)

// 协议protocol
console.log(location.protocol)

// 几个方法
location.assign("http://www.baidu.com")
location.href = "http://www.baidu.com"

location.replace("http://www.baidu.com")
location.reload(false)
```

history对象常见属性和方法

- history 对象允许我们访问浏览器曾经的会话历史记录。
- 有两个属性：
 - length : 会话中的记录条数
 - state : 当前保留的状态值
- 有五个方法：
 - back() : 返回上一页, 等价于 history.go(-1)
 - forward() : 前进下一页, 等价于 history.go(1)
 - go() : 加载历史中的某一页
 - pushState() : 打开一个指定的地址
 - replaceState() : 打开一个新的地址, 并且使用 replace

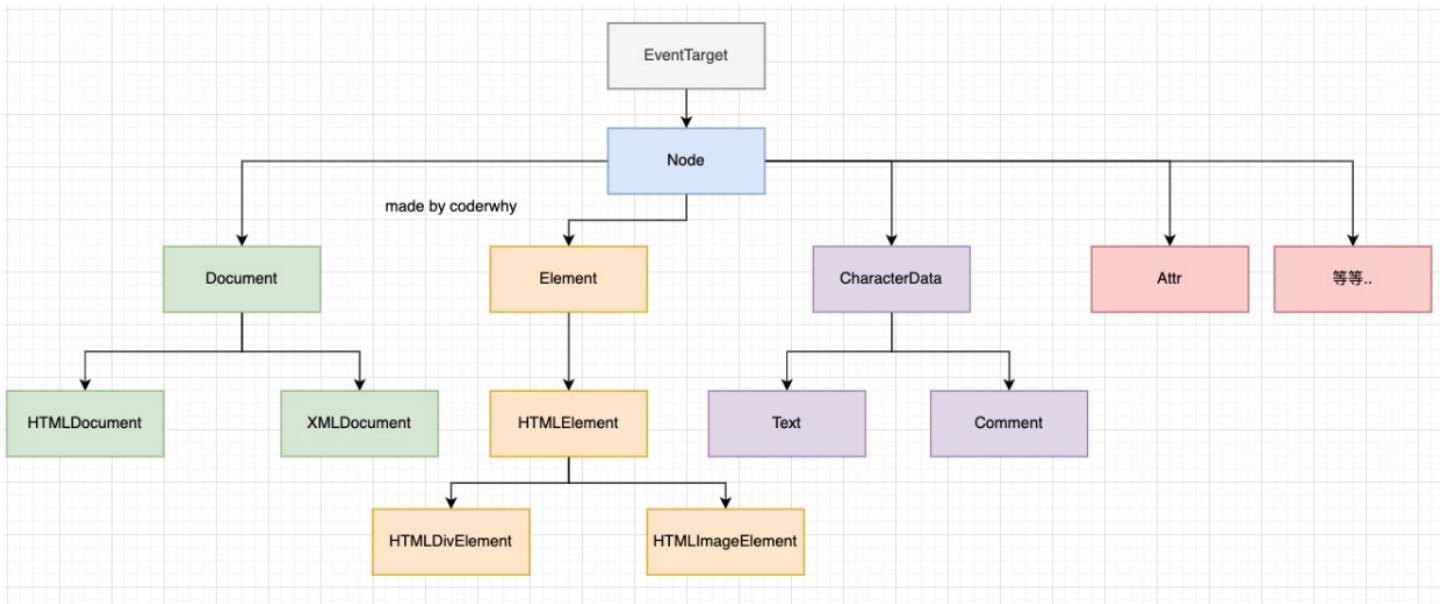
```
const jumpBtn = document.querySelector("#jump")

jumpBtn.onclick = function() {
  location.href = "./demo.html"

  // 跳转(不刷新网页)
  history.pushState({name: "coderwhy"}, "", "/detail")
  history.replaceState({name: "coderwhy"}, "", "/detail")
}
```

DOM和架构

JavaScript通过Document Object Model (DOM, 文档对象模型) 操作元素
DOM给我们提供了一系列的模型和对象, 让我们可以方便的来操作Web页面



Node节点

所有的DOM节点类型都继承自Node接口。

<https://developer.mozilla.org/zh-CN/docs/Web/API/Node>

- Node有几个非常重要的属性：
 - nodeName : node节点的名称
 - nodeType : 可以区分节点的类型
 - nodeValue : node节点的值
 - childNodes : 所有的子节点

```
const divEl = document.querySelector("#box")
const spanEl = document.querySelector(".content")
```

```
// 常见的属性
console.log(divEl.nodeName, spanEl.nodeName)
console.log(divEl.nodeType, spanEl.nodeType)
console.log(divEl.nodeValue, spanEl.nodeValue)
```

```
// childNodes
const spanChildNodes = spanEl.childNodes
const textNode = spanChildNodes[0]
console.log(textNode.nodeValue)
```

```
// 常见的方法
const strongEl = document.createElement("strong")
strongEl.textContent = "我是strong元素"
divEl.appendChild(strongEl)
```

```
// 注意事项: document对象
document.body.appendChild(strongEl)
```

Document

```
// 常见的属性
console.log(document.body)
console.log(document.title)
document.title = "Hello World"

console.log(document.head)
console.log(document.children[0])

console.log(window.location)
console.log(document.location)
console.log(window.location === document.location)

// 常见的方法
// 创建元素
const imageEl = document.createElement("img")
const imageEl2 = new HTMLImageElement()

// 获取元素
const divEl1 = document.getElementById("box")
const divEl2 = document.getElementsByTagName("div")
const divEl3 = document.getElementsByName("title")
const divEl4 = document.querySelector(".content")
const divEl5 = document.querySelectorAll(".content")
```

Element

div、p、span等元素在DOM中表示为Element元素

```
const divEl = document.querySelector("#box")
```

```
// 常见的属性
console.log(divEl.id)
console.log(divEl.tagName)
console.log(divEl.children)
console.log(divEl.className)
console.log(divEl.classList)
console.log(divEl.clientWidth)
console.log(divEl.clientHeight)
console.log(divEl.offsetLeft)
console.log(divEl.offsetTop)

// 常见的方法
const value = divEl.getAttribute("age")
console.log(value)
divEl.setAttribute("height", 1.88)
```

事件监听

浏览器在某个时刻可能会发生一些事件，比如鼠标点击、移动、滚动、获取、失去焦点、输入内容等等一系列的事件

我们需要以某种方式（代码）来对其进行响应，进行一些事件的处理

在Web当中，事件在浏览器窗口中被触发，并且通过绑定到某些元素上或者浏览器窗口本身，那么我们就可以

给这些元素或者window窗口来绑定事件的处理程序，来对事件进行监听

- 事件监听方式一：在script中直接监听
- 事件监听方式二：通过元素的 `on` 来监听事件
- 事件监听方式三：通过EventTarget中的 `addEventListener` 来监听

事件流

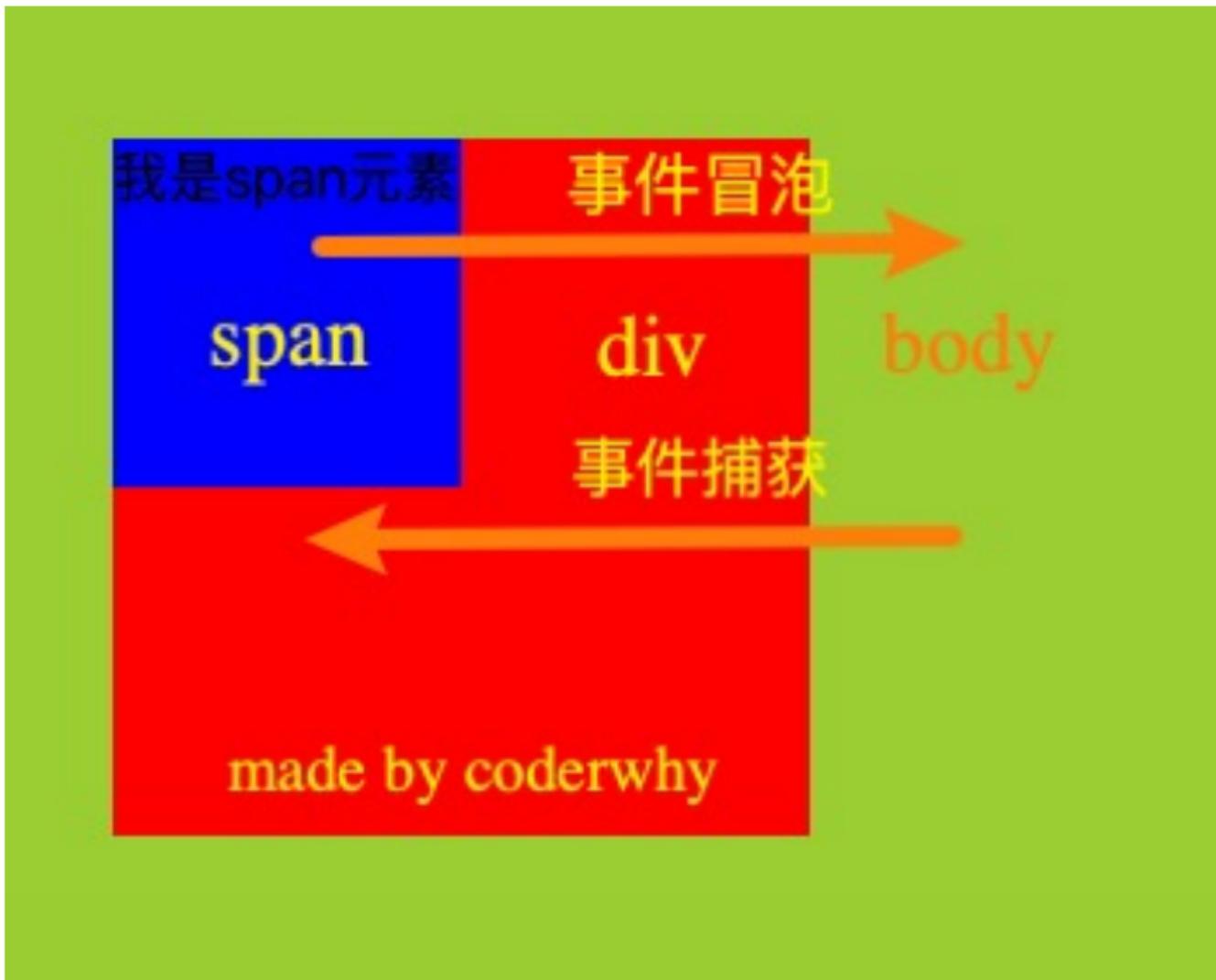
1. 事件冒泡和事件捕获

默认情况下事件是从最内层的span向外依次传递的顺序，这个顺序我们称之为事件冒泡（Event Bubble）

另外一种监听事件流的方式就是从外层到内层（body -> span），这种称之为事件捕获（Event Capture）

2. 冒泡和捕获的顺序

如果我们同时有事件冒泡和时间捕获的监听，那么会优先监听到事件捕获的



事件对象event

- 当一个事件发生时，就会有和这个事件相关的很多信息：
 - 比如事件的类型是什么，你点击的是哪一个元素，点击的位置是哪里等等相关的信息；
 - 那么这些信息会被封装到一个Event对象中；
 - 该对象给我们提供了想要的一些属性，以及可以通过该对象进行某些操作；
- 常见的属性：
 - `type`：事件的类型；
 - `target`：当前事件发生的元素；
 - `currentTarget`：当前处理事件的元素；
 - `offsetX`、`offsetY`：点击元素的位置；
- 常见的方法：
 - `preventDefault`：取消事件的默认行为；
 - `stopPropagation`：阻止事件的进一步传递；
- 事件类型：<https://developer.mozilla.org/zh-CN/docs/Web/Events>

防抖和节流

防抖 debounce 函数