js高级

浏览器的运行机制（重构和回流）

浏览器的渲染过程：加载html文件，将内容渲染成dom树，css加载，由css parser形成css的规则，然后叠加在一起成为render tree，根据layout再进行布局，然后painting

V8引擎：javascript的源代码通过parser转换为AST抽象语法树，然后通过ignition将抽象语法树转换为字节码，或者通过turboFan来进行机器码的优化

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；

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

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；

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；执行上下文栈（调用栈）

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

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：这个我们后续会详细解析

在最新的ECMA标准中，我们前面的变量对象VO已经有另外一个称呼了变量环境VE，然后在执行代码中变量和函数的声明会作为环境记录（Environment Record）添加到环境变量中

ES6

在instantiate过程中LE和VE是指向一个对象的

我一开始想，是否与块作用域相关，作用域需要利用两个环境的切换吗？我去阅读块作用域相关的规范，发现执行到块作用域的时候，大致流程如下：

会创建一个新的 newEnv，记忆 oldEnv。

使当前执行上下文的 LE 指向 newEnv。

执行代码

退出，使当前执行上下文的 LE 指向 oldEnv。

block 代码的执行比函数调用轻，不建立新的执行上下文，这样就完成了 block 中代码的执行。我们可以发现这个过程不依赖于 LE 和 VE 两个环境，其实如果只存在一个环境也可以实现块作用域的效果。那么 LE 和 VE 这两个环境的设计到底是为什么呢？

为了某些情况：

当strict mode为false的时候，为了保证eval的行为正确，LE会进行分岔

js的内存管理和闭包

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；

常见的GC算法–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；p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；

MDN对javascript的闭包解释：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）；

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；但是因为在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：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；

但是v8对内存释放做了一些优化，所以AO中没有被用到的变量就会被释放

this的指向问题

这个问题非常容易回答，在浏览器中测试就是指向window

绑定一：默认绑定；独立函数调用

绑定二：隐式绑定；通过某个对象进行调用的

绑定三：显示绑定；使用call和apply和bind方法，call的参数是剩余参数，apply的参数是数组，bind是返回绑定了this的函数

绑定四：new绑定；

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；

this规则之外-间接函数引用：

箭头函数并不绑定this对象，那么this引用就会从上层作用于中找到对应的this，箭头函数无法显示绑定this

JS函数增强知识点补充

narray-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；

```
// 1.转化方式一:
var length = arguments.length
var arr = []
for (var i = 0; i < length; i++) {
  arr.push(arguments[i])
}
console.log(arr)

// 2.转化方式二
var arr1 = Array.prototype.slice.call(arguments);
var arr2 = [].slice.call(arguments)
console.log(arr1)
console.log(arr2)

// 3.转化方式三: ES6之后
const arr3 = Array.from(arguments)
const arr4 = [...arguments]
console.log(arr3)
console.log(arr4)
```

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

纯函数和柯里化

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”，修改了全局的变量

纯函数的好处：

因为你可以安心的编写和安心的使用；

你在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；

你在用的时候，你确定你的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

柯里化：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；

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；

组合函数：将这两个函数组合起来，自动依次调用

其他知识补充

with语句可以形成自己的作用域

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。

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

在严格模式下, 自执行函数(默认绑定)会指向undefined

JS面向对象

可以使用for in遍历对象的key

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

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。

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

```
"use strict"

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

var address = "北京市"

Object.defineProperty(obj, 'address', {
  configurable: true,
  enumerable: true,
  get: function() {
    return address
  },
  set: function(value) {
    address = value
  }
})

console.log(obj.address)
obj.address = "广州市"
console.log(obj.address)
```

可以同时定义多个属性

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 = {
  _age: 18
}

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

getOwnPropertyDescriptor

```
// 获取某一个特性属性的属性描述符
console.log(Object.getOwnPropertyDescriptor(obj, "name"))
console.log(Object.getOwnPropertyDescriptor(obj, "age"))

// 获取对象的所有属性描述符
console.log(Object.getOwnPropertyDescriptors(obj))
```

创建对象的方案-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

认识构造函数

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；（后面详细讲）；

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；

但是构造函数就没有缺点了吗？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；

认识对象的原型

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；

获取的方式有两种：

方式一：通过对象的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）；

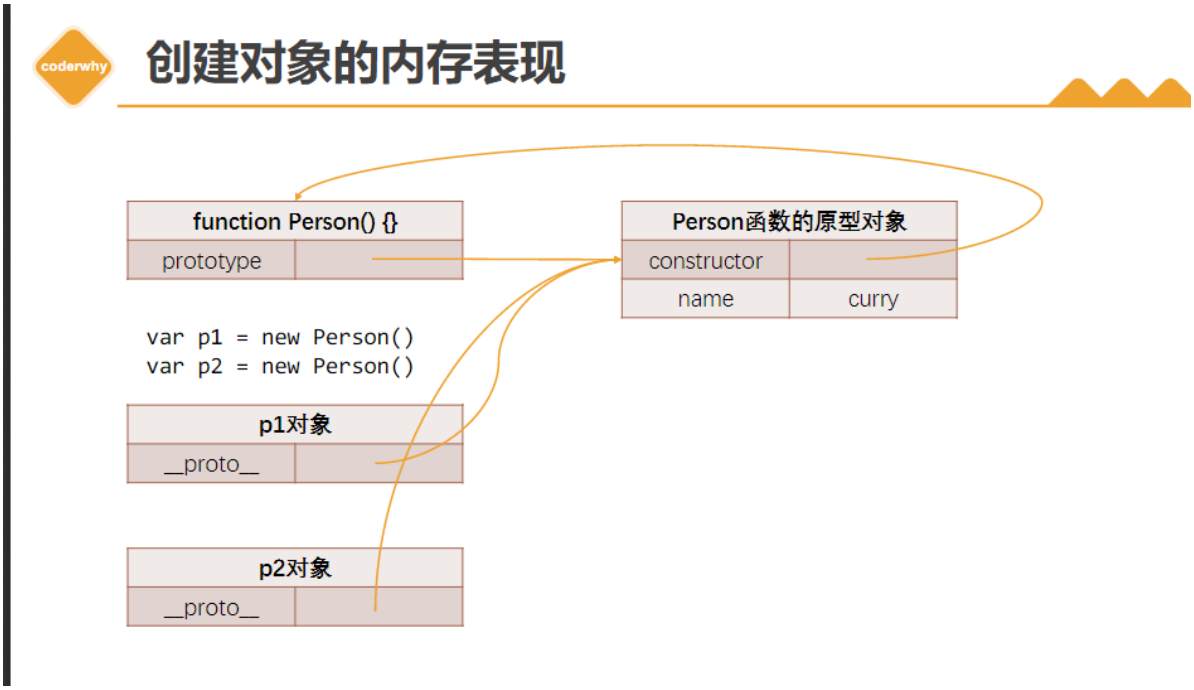
方式二：通过Object.getPrototypeOf方法可以获取到；

函数的原型prototype

所有的函数都有一个prototype的属性

prototype上都有一个construct属性，指向当前的函数对象

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



我们在上一个构造函数的方式创建对象时，有一个弊端：会创建出重复的函数，比如running、eating这些函数

那么有没有办法让所有的对象去共享这些函数呢?p

可以，将这些函数放到Person.prototype的对象上即可

可以给函数的prototype原型对象上添加属性，这样在new出来的新对象上就可以有这些属性

类的三大特性

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

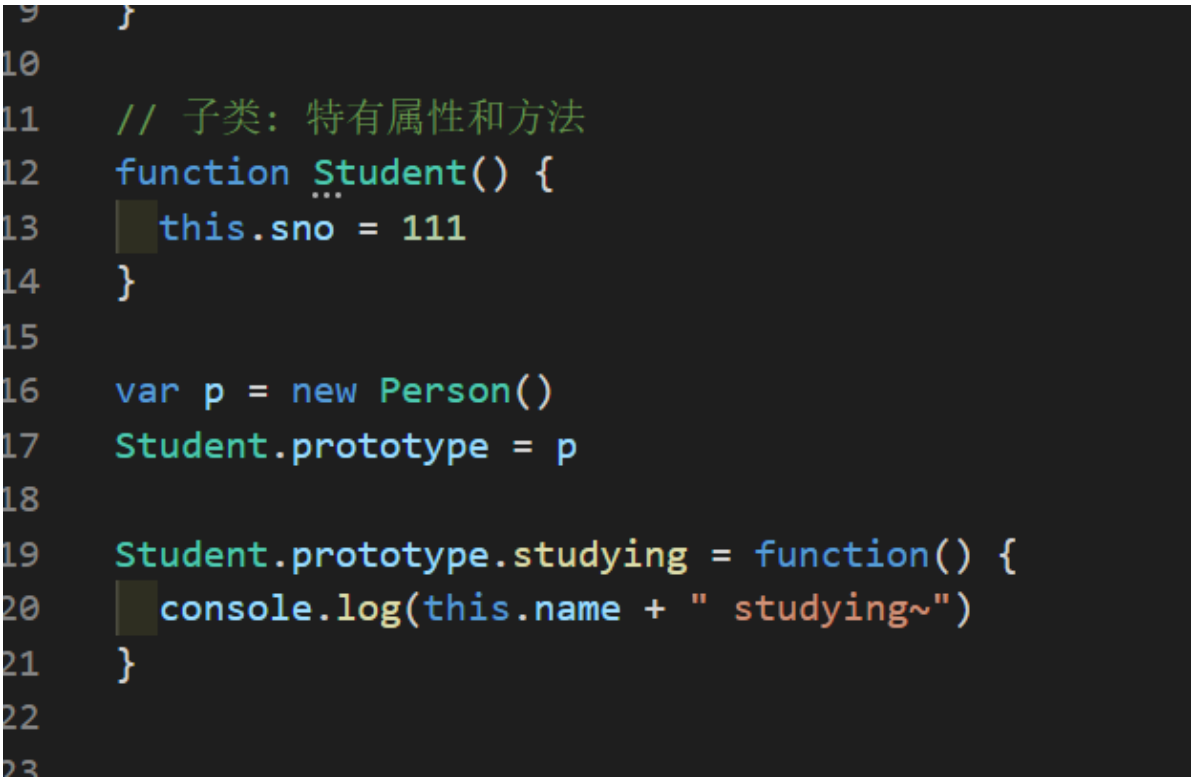
原型链

从我们上面的Object原型我们可以得出一个结论：原型链最顶层的原型对象就是Object的原型对象

每个函数都有一个prototype属性，然后这个prototype是一个对象，它是Object创建的，所以它的**proto**等于Object的prototype

通过原型链实现继承

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




```
// 父类: 公共属性和方法
function Person() {
  this.name = "why"
  this.friends = []
}

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

// 子类: 特有属性和方法
function Student() {
  this.sno = 111
}

var p = new Person()
Student.prototype = p

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)

```

弊端:

第一，我们通过直接打印对象是看不到这个属性的；

第二，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；

第三，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

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

```
// 父类: 公共属性和方法
function Person(name, age, friends) {
  // this = stu
  this.name = name
  this.age = age
  this.friends = friends
}

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

// 子类: 特有属性和方法
function Student(name, age, friends, sno) {
  Person.call(this, name, age, friends)
  // this.name = name
```

```
  // this.age = age
  // this.friends = friends
  this.sno = 111
}

var p = new Person()
Student.prototype = p

Student.prototype.studying = function() {
  console.log(this.name + " studying~")
}


// name/sno
var stu = new Student("why", 18, ["kobe"], 111)

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

// stu.studying()


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

// 2.第二个弊端: 创建出来两个stu的对象
var stu1 = new Student("why", 18, ["lilei"], 111)
var stu2 = new Student("kobe", 30, ["james"], 112)

// // 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("lucy")

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)



// 强调: 借用构造函数也是有弊端:
// 1.第一个弊端: Person函数至少被调用了两次
// 2.第二个弊端: stu的原型对象上会多出一些属性, 但是这些属性是没有存在的必要
```

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。

另外，如果你仔细按照我的流程走了上面的每一个步骤，你会发现：所有的子类实例事实上会拥有两份父类的属性

父类原型赋值给子类

直接将Student的原型改为Person的原型

但是如果每个类都向原型中定义方法，那么原型就会越来越大

```
// 父类: 公共属性和方法
function Person(name, age, friends) {
  // this = stu
  this.name = name
  this.age = age
  this.friends = friends
}

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

// 子类: 特有属性和方法
function Student(name, age, friends, sno) {
  Person.call(this, name, age, friends)
  // this.name = name
  // this.age = age
  // this.friends = friends
  this.sno = 111
}

// 直接将父类的原型赋值给子类, 作为子类的原型
Student.prototype = Person.prototype

Student.prototype.studying = function() {
  console.log(this.name + " studying~")
}


// name/sno
var stu = new Student("why", 18, ["kobe"], 111)
console.log(stu)
stu.eating()

```

原型式继承函数

创建一个新的对象，这个对象的原型为父类的原型

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

//将obj作为返回的对象的原型对象
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

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

缺点：每次创建一个新的对象就会重新创建相同的函数

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

//缺点：每次创建一个新的对象就会重新创建相同的函数
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

```
function createObject(o) {
  function Fn() {}
  Fn.prototype = o
  return new Fn()
}

function inheritPrototype(SubType, SuperType) {
  SubType.prototype = Object.create(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
}

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

Person.prototype.eating = function() {
  console.log("eating~")
}


function Student(name, age, friends, sno, score) {
  Person.call(this, name, age, friends)
  this.sno = sno
  this.score = score
}

inheritPrototype(Student, Person)

Student.prototype.studying = function() {
  console.log("studying~")
}

var stu = new Student("why", 18, ["kobe"], 111, 100)
console.log(stu)
stu.studying()
stu.running()
stu.eating()

console.log(stu.constructor.name)


```

对象的方法补充

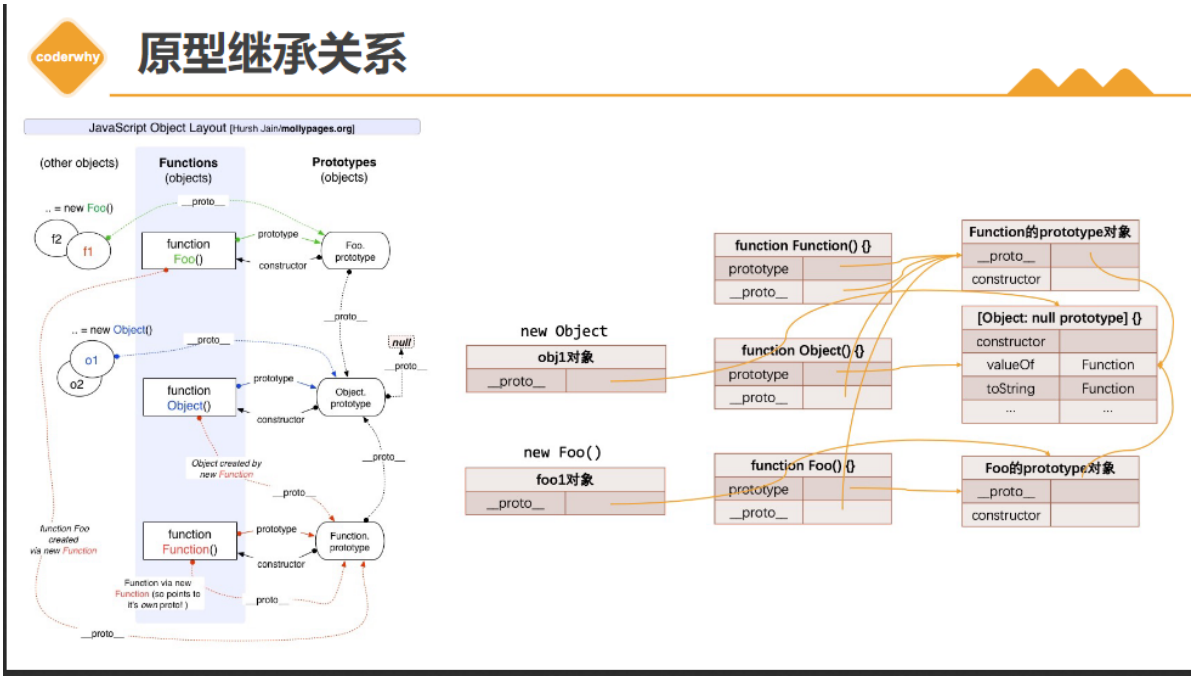
hasOwnProperty: 对象是否有某一个属于自己的属性（不是在原型上的属性）

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

instanceof: 用于检测构造函数的pototype，是否出现在某个实例对象的原型链上

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



JS中实现类的混入效果

```
class Person {}

function mixinRunner(BaseClass) {
  class NewClass extends BaseClass {
    running() {
      console.log("running~");
    }
  }
  return NewClass;
}

function mixinEater(BaseClass) {
  return class extends BaseClass {
    // 匿名类
    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();

```

ES6-ES12

字面量的增强

```
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'
}

obj.baz()
obj.bar()
obj.foo()

// obj[name + 123] = "hahaha"
console.log(obj)

```

解构

```
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)

```

```
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

```

// console.log(foo)
// var foo = "foo"

// Reference(引用)Error: Cannot access 'foo' before initialization(初始化)
// let/const他们是没有作用域提升
// foo被创建出来了, 但是不能被访问
// 作用域提升: 能提前被访问
console.log(foo)
let foo = "foo"

```

模板字符串

```
// ES6之前拼接字符串和其他标识符
const name = "why"
const age = 18
const height = 1.88

```

```
// console.log("my name is " + name + ", age is " + age + ", height is " +
height)

// ES6提供模板字符串 ``
const message = `my name is ${name}, age is ${age}, height is ${height}`
console.log(message)

const info = `age double is ${age * 2}`
console.log(info)

function doubleAge() {
  return age * 2
}

const info2 = `double age is ${doubleAge()}`
console.log(info2)

```

函数的默认参数

```
// ES5以及之前给参数默认值
/**
 * 缺点:
 *  1.写起来很麻烦, 并且代码的阅读性是比较差
 *  2.这种写法是有bug
 */
// function foo(m, n) {
//   m = m || "aaa"
//   n = n || "bbb"

//   console.log(m, n)
// }

// 1.ES6可以给函数参数提供默认值
function foo(m = "aaa", n = "bbb") {
  console.log(m, n)
}

// foo()
foo(0, "")

// 2.对象参数和默认值以及解构
function printInfo({name, age} = {name: "why", age: 18}) {
  console.log(name, age)
}

printInfo({name: "kobe", age: 40})

// 另外一种写法
function printInfo1({name = "why", age = 18} = {}) {
  console.log(name, age)
}

printInfo1()

// 3.有默认值的形参最好放到最后
function bar(x, y, z = 30) {
```

```
  console.log(x, y, z)
}

// bar(10, 20)
bar(undefined, 10, 20)

// 4.•有默认值的函数的length属性
function baz(x, y, z, m, n = 30) {
  console.log(x, y, z, m, n)
}

console.log(baz.length)
```

函数的剩余参数

```
// function foo(...args, m, n) {
//   console.log(m, n)
//   console.log(args)

//   console.log(arguments)
// }

// foo(20, 30, 40, 50, 60)


// rest paramaters必须放到最后
// Rest parameter must be last formal parameter

// function foo(m, n = m + 1) {
//   console.log(m, n)
// }

// foo(10);
```

箭头函数

```
// function foo() {

// }

// console.log(foo.prototype)
// const f = new foo()
// f.__proto__ = foo.prototypee

// 箭头函数没有this，没有arguments，没有原型prototype
var bar = () => {
  console.log(this, arguments);
};

console.log(bar.prototype);

// bar is not a constructor
const b = new bar();
```

展开语法

```
const names = ["abc", "cba", "nba"]
const name = "why"
const info = {name: "why", age: 18}

// 1.函数调用时
function foo(x, y, z) {
  console.log(x, y, z)
}

// foo.apply(null, names)
foo(...names)
foo(...name)

// 2.构造数组时
const newNames = [...names, ...name]
console.log(newNames)

// 3.构建对象字面量时ES2018(ES9)
const obj = { ...info, address: "广州市", ...names }
console.log(obj)
```

Symbol的基本使用

```
// 1.ES6之前, 对象的属性名(key)
// var obj = {
//   name: "why",
//   friend: { name: "kobe" },
//   age: 18
// }


// obj['newName'] = "james"
// console.log(obj)


// 2.ES6中Symbol的基本使用
const s1 = Symbol()
const s2 = Symbol()

console.log(s1 === s2)

// ES2019(ES10)中, Symbol还有一个描述(description)
const s3 = Symbol("aaa")
console.log(s3.description)


// 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的基本使用

```
// 10, 20, 40, 333
// 1.创建Set结构
const set = new Set()
set.add(10)
set.add(20)
set.add(40)
set.add(333)

set.add(10)

// 2.添加对象时特别注意:
set.add({})
set.add({})

const obj = {}
set.add(obj)
set.add(obj)

// console.log(set)
```

```

// 3.对数组去重(去除重复的元素)
const arr = [33, 10, 26, 30, 33, 26]
// const newArr = []
// for (const item of arr) {
//   if (newArr.indexOf(item) !== -1) {
//     newArr.push(item)
//   }
// }

const arrSet = new Set(arr)
// const newArr = Array.from(arrSet)
// const newArr = [...arrSet]
// console.log(newArr)

// 4.size属性
console.log(arrSet.size)

// 5.Set的方法
// add
arrSet.add(100)
console.log(arrSet)

// delete
arrSet.delete(33)
console.log(arrSet)

// has
console.log(arrSet.has(100))

// clear
// arrSet.clear()
console.log(arrSet)

// 6.对Set进行遍历
arrSet.forEach(item => {
  console.log(item)
})

for (const item of arrSet) {
  console.log(item)
}
```

WeakSet的基本使用

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

```

const weakSet = new WeakSet()

// 1.区别一: 只能存放对象类型
// TypeError: Invalid value used in weak set
// weakSet.add(10)

// 强引用和弱引用的概念(看图)
```

```
// 2.区别二: 对对象是一个弱引用
let obj = {
  name: "why"
}

// weakSet.add(obj)

const set = new Set()
// 建立的是强引用
set.add(obj)

// 建立的是弱引用
weakSet.add(obj)

// 3.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

```
// 1.JavaScript中对象中是不能使用对象来作为key的
const obj1 = { name: "why" }
const obj2 = { name: "kobe" }

// const info = {
//   [obj1]: "aaa",
//   [obj2]: "bbb"
// }

// console.log(info)

// 2.Map就是允许我们对象类型来作为key的
// 构造方法的使用
const map = new Map()
map.set(obj1, "aaa")
map.set(obj2, "bbb")
map.set(1, "ccc")
console.log(map)

```

```
const map2 = new Map([[obj1, "aaa"], [obj2, "bbb"], [2, "ddd"]])
console.log(map2)

// 3.常见的属性和方法
console.log(map2.size)

// set
map2.set("why", "eee")
console.log(map2)

// get(key)
console.log(map2.get("why"))

// has(key)
console.log(map2.has("why"))

// delete(key)
map2.delete("why")
console.log(map2)

// clear
// map2.clear()
// console.log(map2)

// 4.遍历map
map2.forEach((item, key) => {
  console.log(item, key)
})

for (const item of map2) {
  console.log(item[0], item[1])
}

for (const [key, value] of map2) {
  console.log(key, value)
}

```

WeakMap

```

const obj = {name: "obj1"}
// 1.WeakMap和Map的区别二:
const map = new Map()
map.set(obj, "aaa")

const weakMap = new WeakMap()
weakMap.set(obj, "aaa")

// 2.区别一: 不能使用基本数据类型
// weakMap.set(1, "ccc")

// 3.常见方法
// get方法
console.log(weakMap.get(obj))

// has方法
```

```
console.log(weakMap.has(obj))

// delete方法
console.log(weakMap.delete(obj))
// WeakMap { <items unknown> }
console.log(weakMap)

```

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。

```
const names = ["abc", "cba", "nba", "mba", NaN]

if (names.indexOf("cba") !== -1) {
  console.log("包含abc元素")
}

// ES7 ES2016
if (names.includes("cba", 2)) {
  console.log("包含abc元素")
}

if (names.indexOf(NaN) !== -1) {
  console.log("包含NaN")
}

if (names.includes(NaN)) {
  console.log("包含NaN")
}

```

指数(乘方)exponentiation运算符

```
const result1 = Math.pow(3, 3)
// ES7: **
const result2 = 3 ** 3
console.log(result1, result2)
```

Object values/Object keys

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

console.log(Object.keys(obj))
console.log(Object.values(obj))

// 用的非常少
console.log(Object.values(["abc", "cba", "nba"]))
console.log(Object.values("abc"))

```

Object entries

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

console.log(Object.entries(obj))
const objEntries = Object.entries(obj)
objEntries.forEach(item => {
  console.log(item[0], item[1])
})

console.log(Object.entries(["abc", "cba", "nba"]))
console.log(Object.entries("abc"))
```

String Padding

```
const message = "Hello World"

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

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

```

Object.getOwnPropertyDescriptors

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)

// const newNums2 = nums.flat(2)
// console.log(newNums2)

// 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)
// console.log(newNums4)

// 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)
```

Object fromEntries

```
// 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 newObj = Object.fromEntries(entries)

// console.log(newObj)


// 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)
}
```

```
27
28 const paramObj = Object.fromEntries(queryParams)
29 console.log(paramObj)
```

trimStart trimEnd

```
1 const message = "    Hello World    "
2
3 console.log(message.trim())
4 console.log(message.trimStart())
5 console.log(message.trimEnd())
```

BigInt

```
1  // ES11之前 max_safe_integer
2  const maxInt = Number.MAX_SAFE_INTEGER
3  console.log(maxInt) // 9007199254740991
4  console.log(maxInt + 1)
5  console.log(maxInt + 2)
6
7  // ES11之后: BigInt
8  const bigInt = 900719925474099100n
9  console.log(bigInt + 10n)
10
11 const num = 100
12 console.log(bigInt + BigInt(num))
13
14 const smallNum = Number(bigInt)
15 console.log(smallNum)
```

Nullish Coalescing Operator

```
1 // ES11: 空值合并运算 ??
2
3 const foo = undefined;
4 // const bar = foo || "default value"
5 const bar = foo ?? "defualt value";
6
7 console.log(bar);
8
9 // ts 是 js 的超级
```

Optional Chaining

```
1  const info = {
2    name: "why",
3    // friend: {
4    //   girlFriend: {
5    //     name: "hmm"
6    //   }
7    // }
8  }
9
```

```
10
11 // console.log(info.friend.girlFriend.name)
12 // if (info && info.friend && info.friend.girlFriend) {
13 //   console.log(info.friend.girlFriend.name)
14 // }
15
16 // ES11提供了可选链(Optional Chainling)
17 console.log(info.friend?.girlFriend?.name)
18
19 console.log('其他的代码逻辑')
```

GlobalThis

在之前我们希望获取JavaScript环境的全局对象，不同的环境获取的方式是不一样的

那么在ES11中对获取全局对象进行了统一的规范：globalThis

```
1  // 获取某一个环境下的全局对象(Global Object)
2
3  // 在浏览器下
4  // console.log(window)
5  // console.log(this)
6
7  // 在node下
8  // console.log(global)
9
10 // ES11
11 console.log(globalThis)
```

FinalizationRegistry

FinalizationRegistry对象可以让你在对象被垃圾回收时请求一个回调。

```
1  // ES12: FinalizationRegistry类
2  // 用来监听某个对象是否被销毁了
3  const finalRegistry = new FinalizationRegistry((value) => {
4    console.log("注册在finalRegistry的对象, 某一个被销毁", value);
5  });
6
7  let obj = { name: "why" };
8  let info = { age: 18 };
9
10 finalRegistry.register(obj, "obj");
11 finalRegistry.register(info, "value");
12
13 obj = null;
14 info = null;
```

WeakRefs

```
1  // ES12: WeakRef类
2  // WeakRef.prototype.deref:
3  // > 如果原对象没有销毁, 那么可以获取到原对象
4  // > 如果原对象已经销毁, 那么获取到的是undefined
5  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)
  console.log(info.deref() && info.deref().name)
}, 10000)

```

逻辑赋值运算符

```
// 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

首先，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

其次，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

在ES6中，新增了一个Proxy类，这个类从名字就可以看出来，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


```
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
```

对函数对象的操作

```
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也是ES6新增的一个API，它是一个对象，字面的意思是反射。

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;

Reflect的使用:

```
const objProxy = new Proxy(obj, {
  has: function(target, key) {
    return Reflect.has(target, key)
  },
  set: function(target, key, value) {
    return Reflect.set(target, key, value)
  },
  get: function(target, key) {
    return Reflect.get(target, key)
  },
  deleteProperty: function(target, key) {
    return Reflect.deleteProperty(target, key)
  }
})
```

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里面的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)
```

响应式原理

我们目前是创建了一个Depend对象，用来管理对于name变化需要监听的响应函数

我们可以写一个getDepend函数专门来管理这种依赖关系

vue3的响应式原理:

```
// 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的响应式原理:

```
// 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

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

上面Promise使用过程，我们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；Promise的代码结构

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：

```
/**
 * resolve(参数)
 *  1> 普通的值或者对象  pending -> fulfilled
 *  2> 传入一个Promise
 *    那么当前的Promise的状态会由传入的Promise来决定
 *    相当于状态进行了移交
 *  3> 传入一个对象，并且这个对象有实现then方法(并且这个对象是实现了thenable接口)
 *    那么也会执行该then方法，并且又该then方法决定后续状态
 */

// 1.传入Promise的特殊情况
const newPromise = new Promise((resolve, reject) => {
  // resolve("aaaaaa")
  reject("err message")
})

new Promise((resolve, reject) => {
  // pending -> fulfilled
  resolve(newPromise)
}).then(res => {
  console.log("res:", res)
}, err => {
  console.log("err:", err)
})

// 2.传入一个对象，这个兑现有then方法
new Promise((resolve, reject) => {
  // pending -> fulfilled
  const obj = {
    then: function(resolve, reject) {
```

```
      // resolve("resolve message")
      reject("reject message")
    }
  }
  resolve(obj)
}).then(res => {
  console.log("res:", res)
}, err => {
  console.log("err:", err)
})

// eatable/runable
const obj = {
  eat: function() {

  },
  run: function() {

  }
}
```

then方法-多次调用

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-返回值

then方法本身是有返回值的，它的返回值是一个Promise，所以我们可以进行如下的链式调用

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；

```
// Promise有哪些对象方法
// console.log(Object.getOwnPropertyDescriptors(Promise.prototype))

const promise = new Promise((resolve, reject) => {
  resolve("hahaha");
});

// 1.同一个Promise可以被多次调用then方法
// 当我们的resolve方法被回调时，所有的then方法传入的回调函数都会被调用
// promise.then(res => {
//   console.log("res1:", res)
// })

// promise.then(res => {
//   console.log("res2:", res)
```

```
// })

// promise.then(res => {
//   console.log("res3:", res)
// })

// 2.then方法传入的 "回调函数: 可以有返回值
// then方法本身也是有返回值的, 它的返回值是Promise

// 1> 如果我们返回的是一个普通值(数值/字符串/普通对象/undefined), 那么这个普通的值被作为一个新的Promise的resolve值
// promise.then(res => {
//   return "aaaaaa"
// }).then(res => {
//   console.log("res:", res)
//   return "bbbbbb"
// })

// 2> 如果我们返回的是一个Promise,同样会用Promise进行包装
promise.then(res => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(111111)
    }, 3000)
  })
}).then(res => {
  console.log("res:", res)
})

// 3> 如果返回的是一个对象, 并且该对象实现了thenable
promise
  .then((res) => {
    return {
      then: function (resolve, reject) {
        resolve(222222);
      },
    };
  })
  .then((res) => {
    console.log("res:", res);
  });
```

catch方法-多次调用

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;

catch方法-返回值

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

```
// const promise = new Promise((resolve, reject) => {
//   resolve()
```

```
//   // reject("rejected status")
//   // throw new Error("rejected status")
// })

// 1.当executor抛出异常时, 也是会调用错误(拒绝)捕获的回调函数的
// promise.then(undefined, err => {
//   console.log("err:", err)
//   console.log("----------")
// })

// 2.通过catch方法来传入错误(拒绝)捕获的回调函数
// promise/a+规范
// promise.catch(err => {
//   console.log("err:", err)
// })
// promise.then(res => {
//   // return new Promise((resolve, reject) => {
//   //   reject("then rejected status")
//   // })
//   throw new Error("error message")
// }).catch(err => {
//   console.log("err:", err)
// })


// 3.拒绝捕获的问题(前面课程)
// promise.then(res => {

// }, err => {
//   console.log("err:", err)
// })
// const promise = new Promise((resolve, reject) => {
//   reject("111111")
//   // resolve()
// })

// promise.then(res => {
// }).then(res => {
//   throw new Error("then error message")
// }).catch(err => {
//   console.log("err:", err)
// })

// promise.catch(err => {

// })

// 4.catch方法的返回值
const promise = new Promise((resolve, reject) => {
  reject("111111")
})

promise.then(res => {
  console.log("res:", res)
}).catch(err => {
  console.log("err:", err)
  return "catch return value"
}).then(res => {
```

```
  console.log("res result:", res)
}).catch(err => {
  console.log("err result:", err)
})

```

finally

```
const promise = new Promise((resolve, reject) => {
  // resolve("resolve message")
  reject("reject message")
})

promise.then(res => {
  console.log("res:", res)
}).catch(err => {
  console.log("err:", err)
}).finally(() => {
  console.log("finally code execute")
})
```

Promise的类方法-resolve

Promise.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

另外一个类方法是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

```
// 创建多个Promise
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(11111)
  }, 1000);
})

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(22222)
  }, 2000);
})

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(33333)
  }, 3000);
})

// 需求: 所有的Promise都变成fulfilled时, 再拿到结果
// 意外: 在拿到所有结果之前, 有一个promise变成了rejected, 那么整个promise是rejected
Promise.all([p2, p1, p3, "aaaa"]).then(res => {
  console.log(res)
}).catch(err => {
  console.log("err:", err)
})

```

Promise.allSettled

在ES11（ES2020）中，添加了新的APIPromise.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；

```
// 创建多个Promise
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(11111)
  }, 1000);
})

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(22222)
  }, 2000);
})

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(33333)
  }, 3000);
})

// allSettled
Promise.allSettled([p1, p2, p3]).then(res => {
```

```
  console.log(res)
}).catch(err => {
  console.log(err)
})

```

Promise.race

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

```
// 创建多个Promise
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(11111)
  }, 3000);
})

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(22222)
  }, 500);
})

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(33333)
  }, 1000);
})

// race: 竞技/竞赛
// 只要有一个Promise变成fulfilled状态，那么就结束
// 意外:
Promise.race([p1, p2, p3]).then(res => {
  console.log("res:", res)
}).catch(err => {
  console.log("err:", err)
})

```

Promise.any

any方法是ES12中新增的方法，和race方法是类似的：

any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

```
// 创建多个Promise
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    // resolve(11111)
    reject(1111)
  }, 1000);
})

```

```
const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(22222)
  }, 500);
})

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    // resolve(33333)
    reject(3333)
  }, 3000);
})

// any方法
Promise.any([p1, p2, p3]).then(res => {
  console.log("res:", res)
}).catch(err => {
  console.log("err:", err.errors)
})
```

迭代器和生成器

迭代器（iterator），是确使用户可在容器对象（container，例如链表或数组）上遍访的对象，使用该接口无需关心对象的内部实现细节

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议（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

```
// 编写的一个迭代器
const iterator = {
  next: function() {
    return { done: true, value: 123 }
  }
}

// 数组
const names = ["abc", "cba", "nba"]

// 创建一个迭代器对象来访问数组
let index = 0

const namesIterator = {
  next: function() {
    if (index < names.length) {
```

```
      return { done: false, value: names[index++] }
    } else {
      return { done: true, value: undefined }
    }
  }
}

console.log(namesIterator.next())
console.log(namesIterator.next())
console.log(namesIterator.next()) // { done: false, value: "nba" }
console.log(namesIterator.next()) // { done: true, value: undefined }
console.log(namesIterator.next()) // { done: true, value: undefined }
console.log(namesIterator.next()) // { done: true, value: undefined }
console.log(namesIterator.next())
console.log(namesIterator.next())
console.log(namesIterator.next())

```

什么又是可迭代对象呢?

它和迭代器是不同的概念;

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;

```
// 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 }
        }
      }
    }
  }
}

// iterableObj对象就是一个可迭代对象
// console.log(iterableObj[Symbol.iterator])

// 1.第一次调用iterableObj[Symbol.iterator]函数
// const iterator = iterableObj[Symbol.iterator]()
// console.log(iterator.next())
// console.log(iterator.next())
// console.log(iterator.next())
// console.log(iterator.next())

// // 2.第二次调用iterableObj[Symbol.iterator]函数
// const iterator1 = iterableObj[Symbol.iterator]()
// console.log(iterator1.next())
// console.log(iterator1.next())
// console.log(iterator1.next())
```

```
// console.log(iterator1.next())

// 3.for...of可以遍历的东西必须是一个可迭代对象
// const obj = {
//   name: "why",
//   age: 18
// }

for (const item of iterableObj) {
  console.log(item)
}
```

可迭代对象的应用场景

```
// 1.for of场景

// 2.展开语法(spread syntax)
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 }
        }
      }
    }
  }
}

const names = ["abc", "cba", "nba"]
const newNames = [...names, ...iterableObj]
console.log(newNames)

const obj = { name: "why", age: 18 }
// for (const item of obj) {

// }
// ES9(ES2018)中新增的一个特性: 用的不是迭代器
const newObj = { ...obj }
console.log(newObj)


// 3.解构语法
const [ name1, name2 ] = names
// const { name, age } = obj 不一样ES9新增的特性

// 4.创建一些其他对象时
const set1 = new Set(iterableObj)
const set2 = new Set(names)

const arr1 = Array.from(iterableObj)

// 5.Promise.all
```

```
Promise.all(iterableObj).then(res => {
  console.log(res)
})
```

自定义类的迭代实现

实现@@iterator函数

```
// class Person {

// }

// const p1 = new Person()
// const p2 = new Person()
// const p3 = new Person()

// 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() {

}
```

迭代器的中断

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；

那么这个时候我们想要监听中断的话，可以添加return方法

生成器

生成器是ES6中新增的一种函数控制、使用的方案，它可以让我们更加灵活的控制函数什么时候继续执行、暂停执行等

首先，生成器函数需要在function的后面加一个符号：*

其次，生成器函数可以通过yield关键字来控制函数的执行流程：

最后，生成器函数的返回值是一个Generator（生成器）：

```
// 当遇到yield时候值暂停函数的执行
// 当遇到return时候生成器就停止执行
function* foo() {
  console.log("函数开始执行~");

  const value1 = 100;
  console.log("第一段代码:", value1);
  yield value1;

  const value2 = 200;
  console.log("第二段代码:", value2);
  yield value2;

  const value3 = 300;
  console.log("第三段代码:", value3);
  yield value3;

  console.log("函数执行结束~");
  return "123";
}

// generator本质上是一个特殊的iterator
const generator = foo();
console.log("返回值1:", generator.next());
console.log("返回值2:", generator.next());
console.log("返回值3:", generator.next());
console.log("返回值3:", generator.next());
console.log("返回值3:", generator.next());
```

生成器的next传递参数

```
function* foo(num) {
  console.log("函数开始执行~")

  const value1 = 100 * num
  console.log("第一段代码:", value1)
  const n = yield value1

  const value2 = 200 * n
  console.log("第二段代码:", value2)
  const count = yield value2

  const value3 = 300 * count
  console.log("第三段代码:", value3)
  yield value3

  console.log("函数执行结束~")
  return "123"
}

// 生成器上的next方法可以传递参数
const generator = foo(5)
// console.log(generator.next())
// // 第二段代码, 第二次调用next的时候执行的
// console.log(generator.next(10))
// console.log(generator.next(25))
```

生成器提前结束-return函数

```
function* foo(num) {
  console.log("函数开始执行~")

  const value1 = 100 * num
  console.log("第一段代码:", value1)
  const n = yield value1

  const value2 = 200 * n
  console.log("第二段代码:", value2)
  const count = yield value2

  const value3 = 300 * count
  console.log("第三段代码:", value3)
  yield value3

  console.log("函数执行结束~")
  return "123"
}

const generator = foo(10)

console.log(generator.next())

// 第二段代码的执行, 使用了return
// 那么就意味着相当于在第一段代码的后面加上return, 就会提前终端生成器函数代码继续执行
console.log(generator.return(15))
```

```
console.log(generator.next())
console.log(generator.next())
console.log(generator.next())
console.log(generator.next())
console.log(generator.next())
console.log(generator.next())
```

生成器抛出异常-throw函数

```
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

事实上我们还可以使用yield*来生产一个可迭代对象：

这个时候相当于是一种yield的语法糖，只不过会依次迭代这个可迭代对象，每次迭代其中的一个值

```
// 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" }
}

// const names = ["abc", "cba", "nba"]
// const namesIterator = createArrayIterator(names)
```

```
18
19  // console.log(namesIterator.next())
20  // console.log(namesIterator.next())
21  // console.log(namesIterator.next())
22  // console.log(namesIterator.next())
23
24  // 2.创建一个函数, 这个函数可以迭代一个范围内的数字
25  // 10 20
26  function* createRangeIterator(start, end) {
27    let index = start;
28    while (index < end) {
29      yield index++;
30    }
31
32    // let index = start
33    // return {
34    //   next: function() {
35    //     if (index < end) {
36    //       return { done: false, value: index++ }
37    //     } else {
38    //       return { done: true, value: undefined }
39    //     }
40    //   }
41    // }
42  }
43
44  const rangeIterator = createRangeIterator(10, 20);
45  console.log(rangeIterator.next());
46  console.log(rangeIterator.next());
47  console.log(rangeIterator.next());
48  console.log(rangeIterator.next());
49  console.log(rangeIterator.next());
50
51  // 3.class案例
52  class Classroom {
53    constructor(address, name, students) {
54      this.address = address;
55      this.name = name;
56      this.students = students;
57    }
58
59    entry(newStudent) {
60      this.students.push(newStudent);
61    }
62
63    foo = () => {
64      console.log("foo function");
65    };
66
67    // [Symbol.iterator] = function*() {
68    //   yield* this.students
69    // }
70
71    *[Symbol.iterator]() {
72      yield* this.students;
73    }
74  }
75
```

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

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)
// }
```

```
51
52  // 1> 手动执行生成器函数
53  const generator = getData()
54  generator.next().value.then(res => {
55    generator.next(res).value.then(res => {
56      generator.next(res).value.then(res => {
57        generator.next(res)
58      })
59    })
60  })
61
62  // 2> 自己封装了一个自动执行的函数
63  // function execGenerator(genFn) {
64  //   const generator = genFn()
65  //   function exec(res) {
66  //     const result = generator.next(res)
67  //     if (result.done) {
68  //       return result.value
69  //     }
70  //     result.value.then(res => {
71  //       exec(res)
72  //     })
73  //   }
74  //   exec()
75  // }
76
77  // execGenerator(getData)
78  // execGenerator(getDepartment)
79
80  // 3> 第三方包co自动执行
81  // TJ: co/n(nvm)/commander(coderwhy/vue cli)/express/koa(egg)
82  // const co = require('co')
83  // co(getData)
84
85
86  // 4.第四种方案: async/await
87  async function getData() {
88    const res1 = await requestData("why")
89    const res2 = await requestData(res1 + "aaa")
90    const res3 = await requestData(res2 + "bbb")
91    const res4 = await requestData(res3 + "ccc")
92    console.log(res4)
93  }
94
95  getData()
96
```

异步函数async function

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。

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；

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。

await关键字有什么特点呢?

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;

```
async function foo() {
  console.log("foo function start~");

  console.log("中间代码~");

  // 异步函数中的异常, 会被作为异步函数返回的Promise的reject值的
  throw new Error("error message");

  console.log("foo function end~");
}

// 异步函数的返回值一定是一个Promise
foo().catch((err) => {
  console.log("coderwhy err:", err);
});

console.log("后续还有代码~~~~~");
```

```
// 1.await更上表达式
function requestData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // resolve(222)
      reject(1111);
    }, 2000);
  });
}

// async function foo() {
//   const res1 = await requestData()
//   console.log("后面的代码1", res1)
//   console.log("后面的代码2")
//   console.log("后面的代码3")

//   const res2 = await requestData()
//   console.log("res2后面的代码", res2)
// }

// 2.跟上其他的值
// async function foo() {
//   // const res1 = await 123
//   // const res1 = await {
//   //   then: function(resolve, reject) {
```

```
//   //     resolve("abc")
//   //   }
//   // }
//   const res1 = await new Promise((resolve) => {
//     resolve("why")
//   })
//   console.log("res1:", res1)
// }

// 3.reject值
// 直接将reject里面的值封装成为一个新的promise进行返回
async function foo() {
  const res1 = await requestData();
  console.log("res1:", res1);
}

foo().catch((err) => {
  console.log("err:", err);
});
```

事件循环

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；

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

宏任务和微任务

宏任务队列（macrotaskqueue）：ajax、setTimeout、setInterval、DOM监听、UI Rendering等

微任务队列（microtask queue）：Promise的then回调、Mutation Observer API、queueMicrotask()等

1.mainscript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）；

异常处理

```
// class HYError {
//   constructor(errorCode, errorMessage) {
//     this.errorCode = errorCode
//     this.errorMessage = errorMessage
```

```
//   }
// }

function foo(type) {
  console.log("foo函数开始执行")

  if (type === 0) {
    // 1.抛出一个字符串类型(基本的数据类型)
    // throw "error"

    // 2.比较常见的是抛出一个对象类型
    // throw { errorCode: -1001, errorMessage: "type不能为0~" }

    // 3.创建类, 并且创建这个类对应的对象
    // throw new HYError(-1001, "type不能为0~")

    // 4.提供了一个Error
    // const err = new Error("type不能为0")
    // err.name = "why"
    // err.stack = "aaaa"

    // 5.Error的子类
    const err = new TypeError("当前type类型是错误的~")

    throw err

    // 强调: 如果函数中已经抛出了异常, 那么后续的代码都不会继续执行了
    console.log("foo函数后续的代码")
  }

  console.log("foo函数结束执行")
}

foo(0)

console.log("后续的代码继续执行~")


// function test() {
//   console.log("test")
// }

// function demo() {
//   test()
// }

// function bar() {
//   demo()
// }

// bar()
```

```
function foo(type) {
  if (type === 0) {
    throw new Error("foo error message~")
  }
}
```

```
6
7  // 1.第一种是不处理，bar函数会继续将收到的异常直接抛出去
8  function bar() {
9    // try {
10   foo(0)
11   //   console.log("bar函数后续的继续运行")
12   // } catch(err) {
13   //   console.log("err:", err.message)
14   //   // alert(err.message)
15   // } finally {
16   //   console.log("finally代码执行~, close操作")
17   // }
18 }
19
20 function test() {
21   try {
22     bar()
23   } catch (error) {
24     console.log("error:", error)
25   }
26 }
27
28 function demo() {
29   test()
30 }
31
32
33 // 两种处理方法:
34 // 1.第一种是不处理，那么异常会进一步的抛出，直到最顶层的调用
35 // 如果在最顶层也没有对这个异常进行处理，那么我们的程序就会终止执行，并且报错
36 // foo()
37
38 // 2.使用try catch来捕获异常
39
40 try {
41   demo()
42 } catch (err) {
43
44 }
45
46 console.log("后续的代码执行~")
```

Js的模块化

事实上模块化开发最终的目的是将程序划分成一个个小的结构；

这个结构中编写属于自己的逻辑代码，有自己的作用域，不会影响到其他的结构；

这个结构可以将自己希望暴露的变量、函数、对象等导出给其结构使用；

也可以通过某种方式，导入另外结构中的变量、函数、对象等

为了让JavaScript支持模块化，涌现出了很多不同的模块化规范：AMD、CMD、CommonJS等

但是，我们其实带来了新的问题：

第一，我必须记得每一个模块中返回对象的命名，才能在其他模块使用过程中正确的使用；

第二，代码写起来混乱不堪，每个文件中的代码都需要包裹在一个匿名函数中来编写；

第三，在没有合适的规范情况下，每个人、每个公司都可能会任意命名、甚至出现模块名称相同的情况

我们需要制定一定的规范来约束每个人都按照这个规范去编写模块化的代码；

这个规范中应该包括核心功能：模块本身可以导出暴露的属性，模块又可以导入自己需要的属性；

JavaScript社区为了解决上面的问题，涌现出一系列好用的规范，接下来我们就学习具有代表性的一些规范。

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；

CommonJs

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；

但是，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exports = module.exports中的bar；module.exports

```
const name = "why"
const age = 18
function sum(num1, num2) {
  return num1 + num2
}

// 源码
// module.exports = {}
// exports = module.exports

// 第二种导出方式
// exports.name = name
// exports.age = age
// exports.sum = sum

// 这种代码不会进行导出
// exports = {
//   name,
//   age,
//   sum
// }

// 这种代码不会进行导出
// exports.name = name
// exports.age = age
// exports.sum = sum

module.exports = {

```

```
}

// 最终能导出的一定是module.exports
```

require细节

1> 直接查找文件X

2> 查找X.js文件

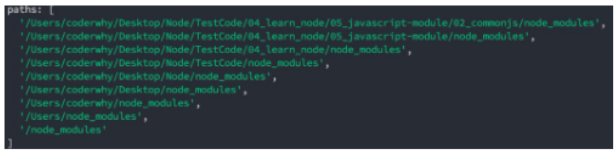
3> 查找X.json文件

4> 查找X.node文件

```
// 情况一: 核心模块
// const path = require("path")
// const fs = require("fs")

// path.resolve()
// path.extname()

// fs.readFile()

// 情况二: 路径 ./ ../ /
const abc = require("./abc")

// console.log(abc.name)

// 情况三: X不是路径也不是核心模块
const axios = require("axios")

// axios.get()

console.log(module.paths)

```

情况三

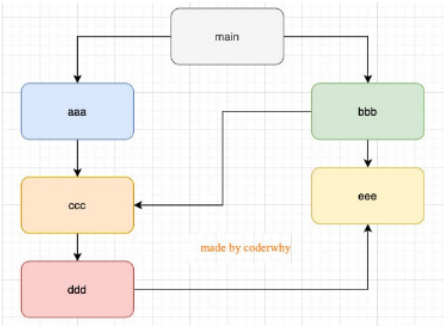
- 情况三：直接是一个X（没有路径），并且X不是一个核心模块
- /Users/coderwhy/Desktop/Node/TestCode/04_learn_node/05_javascript-module/02_commonjs/main.js中编写 require('why')



- 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**
  - 为什么只会加载运行一次呢？
  - 这是因为每个模块对象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，那么加载顺序是什么？**
- 如果出现右图模块的引用关系，那么加载顺序是什么呢？
  - 这个其实是一种数据结构：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：main -> aaa -> ccc -> ddd -> eee ->bbb



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：**
  - 当然在webpack中使用CommonJS是另外一回事；
  - 因为它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**：
  - 但是目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已经使用非常少了，所以这里我们进行简单的演练；

ES Module

ES Module和CommonJS的模块化有一些不同之处：

一方面它使用了import和export关键字；

另一方面它采用编译期的静态分析，并且也加入了动态引用的方式

```
// 1.第一种方式: export 声明语句
// export const name = "why"
// export const age = 18

// export function foo() {
//   console.log("foo function")
// }

// export class Person {

// }

```

```
// 2.第二种方式: export 导出 和 声明分开
const name = "why"
const age = 18
function foo() {
  console.log("foo function")
}

export {
  name,
  age,
  foo
}

// 3.第三种方式: 第二种导出时起别名
// export {
//   name as fName,
//   age as fAge,
//   foo as fFoo
// }
```

```
// 1.导入方式一: 普通的导入
// import { name, age, foo } from "./foo.js"
// import { fName, fAge, fFoo } from './foo.js'

// 2.导入方式二: 起别名
// import { name as fName, age as fAge, foo as fFoo } from './foo.js'

// 3.导入方式三: 将导出的所有内容放到一个标识符中
import * as foo from './foo.js'

console.log(foo.name)
console.log(foo.age)
foo.foo()

const name = "main"

console.log(name)
console.log(age)

```

```
// 1.导出方式一:
// import { add, sub } from './math.js'
// import { timeFormat, priceFormat } from './format.js'

// export {
//   add,
//   sub,
//   timeFormat,
//   priceFormat
// }

// 2.导出方式二:
// export { add, sub } from './math.js'
// export { timeFormat, priceFormat } from './format.js'

// 3.导出方式三:
```

```
export * from './math.js'
export * from './format.js'
```

```
const name = "why"
const age = 18

const foo = "foo value"

// 1.默认导出的方式一:
export {
  // named export
  name,
  // age as default,
  // foo as default
}

// 2.默认导出的方式二: 常见
export default foo

// 注意: 默认导出只能有一个
```

import函数

ES Module的解析过程

ESModule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（ModuleRecord）；

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