# 前面课程笔记链接

笔记1：es6学习笔记1-12节

笔记2：es6学习笔记13-18节

笔记3：es6学习笔记19节

# 17.es6的Generator函数

## 1，什么的Generator

|  |
| --- |
|  |

## 注意：定义生成器函数的时候需要在函数名前面加一个\*，否则报错

## 2.课堂笔记

### 基本案例

|  |
| --- |
| //简单用法          function \*Gen() {              console.log("first");              yield              console.log('second');              yield              console.log("third");            }          let gen = Gen()          // gen.next()          // gen.next()          // gen.next()          // gen.next()          console.log(gen.next()); //{value: undefined, done: false} 因为上面的yield后面没有参数，所以是undefined          console.log(gen.next()); //{value: undefined, done: false} 因为上面的yield后面没有参数，所以是undefined          console.log(gen.next()); //{value: undefined, done: false} 因为上面的yield后面没有参数，所以是undefined          console.log("================================");          //yield后面可以跟参数          function\* func(){              console.log("one");              yield '1'; //会返回一个对象有一个value属性和一个done属性              console.log("two");              yield '2';              console.log("three");              return '3';          }            let f = func()          // f.next()          // f.next()          // f.next()          console.log(f.next()); //{value: '1', done: false}          console.log(f.next()); //{value: '2', done: false}          console.log(f.next()); //{value: '3', done: false}          console.log(f.next()); //{value: undefined, done: true} |
|  |

### next方法是有返回值的，它返回一个对象，有value和done属性，如果没有设置，value是undefined，done表明是否到达迭代器是否迭代完成

### 生成器也可以用for..of来遍历，但是return后面的值for..of无法遍历出来。用next方法能够取出来

|  |
| --- |
| function\* func(){              // console.log("one");              yield '1'; //会返回一个对象有一个value属性和一个done属性              // console.log("two");              yield '2';              // console.log("three");              return '3';          }            let f = func()            for(let f of func()){              console.log(f);          } |
|  |

### next方法也是可以接收参数的，next函数的参数实际上会传递给yield

|  |
| --- |
| //next()方法里面可以接收参数，然后他是传递给yield的           function \*GenFun() {              let ret1 = yield "123"              console.log("inside myGen",ret1);              let ret2 = yield "456"              console.log("inside myGen",ret2);          }          let myGen = GenFun()          let res1 = myGen.next("hello")          console.log(res1);          let res2 = myGen.next("world")          console.log(res2);          let res3=myGen.next("girls")          console.log(res3); |
|  |

#### 需要注意的是：第一次调用next方法传递的参数会丢失，因为当参数进入函数内部，在赋值给ret1之前就遇到yield，程序暂停，把yield后面的数据返回，然后第二次执行next，就把这一次的参数传递给了ret1，以此类推，第一次的就丢失了

## 生成器应用小案例，把上面Ajax的代码拿过来，然后在生成器函数里面调用

|  |
| --- |
| <script>          //1.封装自己的ajax          function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(xhr.responseText)                          } else {                              reject(xhr.responseText)                          }                      }                    }              })            }       // 生成器        function \*Gen() {          let res = yield myAjax("1.json")          console.log(res);        }        let g = Gen()        g.next()      </script> |

### 此时直接调用会报错，因为我们还没有1.json文件

|  |
| --- |
|  |

### 然后我们在文件夹下面新建一个1.json

|  |
| --- |
|  |

### 然后我们再来调用，注意因为myAjax返回的是一个Promise对象，但是他是在返回对象的value对应的值，所以需要首页返回值.value.then来获取数据

|  |
| --- |
| <script>          //1.封装自己的ajax          function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(xhr.responseText)                          } else {                              reject(xhr.responseText)                          }                      }                    }              })            }       // 生成器        function \*Gen() {          let res = yield myAjax("1.json")          console.log(res);        }        let g = Gen()      //   console.log(g.next());     g.next().value.then(res=>console.log(res)); |
|  |

### 然后我们想把这个值回传到生成器里面，方便我们做进一步处理，需要两次next方法

|  |
| --- |
| <script>          //1.封装自己的ajax          function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(xhr.responseText)                          } else {                              reject(xhr.responseText)                          }                      }                    }              })            }       // 生成器        function \*Gen() {          let res = yield myAjax("1.json")          console.log("回传到Gen里面：",res);        }        let g = Gen()        g.next(g.next().value.then(res=>console.log(res)))      </script> |
|  |

### 然后我们可以进行下一次产出，把第一次的结果当作参数传递进去，需要把代码修改一下，我们需要添加一个2.json

|  |
| --- |
|  |
| <script>          //1.封装自己的ajax          function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(xhr.responseText)                          } else {                              reject(xhr.responseText)                          }                      }                    }              })            }       // 生成器        function \*Gen() {          let res = yield myAjax("1.json")          console.log("第一次请求结果:",res);          let res2 =  yield myAjax("2.json",res)          console.log("第二次请求结果:",res2);        }        let g = Gen()      g.next().value.then(data=>{          g.next(data).value.then(res=>{              g.next(res)          })      })      </script> |
|  |

#### 上面其实就是es8的async 和await的实现原理，不过还是挺麻烦的，下面有一个自动代码

|  |
| --- |
|  |

## 生成器参考文档1

|  |
| --- |
| Generator 函数组成Generator 有两个区分于普通函数的部分：1.是在 function 后面，函数名之前有个 \* ；2.函数内部有 yield 表达式。 **其中 \* 用来表示函数为 Generator 函数，yield 用来定义函数内部的状态。**  **function\* func(){**  **console.log("one");**  **yield '1';**  **console.log("two");**  **yield '2';**  **console.log("three");**  **return '3';**  **}** 执行机制 **调用 Generator 函数和调用普通函数一样，在函数名后面加上()即可，但是 Generator 函数不会像普通函数一样立即执行，而是返回一个指向内部状态对象的指针，所以要调用遍历器对象Iterator 的 next 方法，指针就会从函数头部或者上一次停下来的地方开始执行。**  **f.next();**  **// one**  **// {value: "1", done: false}**    **f.next();**  **// two**  **// {value: "2", done: false}**    **f.next();**  **// three**  **// {value: "3", done: true}**    **f.next();**  **// {value: undefined, done: true}**  **第一次调用 next 方法时，从 Generator 函数的头部开始执行，先是打印了 one ,执行到 yield 就停下来，并将yield 后边表达式的值 '1'，作为返回对象的 value 属性值，此时函数还没有执行完， 返回对象的 done 属性值是 false。**  **第二次调用 next 方法时，同上步 。**  **第三次调用 next 方法时，先是打印了 three ，然后执行了函数的返回操作，并将 return 后面的表达式的值，作为返回对象的 value 属性值，此时函数已经结束，多以 done 属性值为true 。**  **第四次调用 next 方法时， 此时函数已经执行完了，所以返回 value 属性值是 undefined ，done 属性值是 true 。如果执行第三步时，没有 return 语句的话，就直接返回 {value: undefined, done: true}。**  **函数返回的遍历器对象的方法**  **next 方法**  **一般情况下，next 方法不传入参数的时候，yield 表达式的返回值是 undefined 。当 next 传入参数的时候，该参数会作为上一步yield的返回值。**  **function\* sendParameter(){**  **console.log("start");**  **var x = yield '2';**  **console.log("one:" + x);**  **var y = yield '3';**  **console.log("two:" + y);**  **console.log("total:" + (x + y));**  **}**  **next不传参**  **var sendp1 = sendParameter();**  **sendp1.next();**  **// start**  **// {value: "2", done: false}**  **sendp1.next();**  **// one:undefined**  **// {value: "3", done: false}**  **sendp1.next();**  **// two:undefined**  **// total:NaN**  **// {value: undefined, done: true}** next传参 **var sendp2 = sendParameter();**  **sendp2.next(10);**  **// start**  **// {value: "2", done: false}**  **sendp2.next(20);**  **// one:20**  **// {value: "3", done: false}**  **sendp2.next(30);**  **// two:30**  **// total:50**  **// {value: undefined, done: true}**  **除了使用 next ，还可以使用 for... of 循环遍历 Generator 函数生产的 Iterator 对象。**  **return 方法**  **return 方法返回给定值，并结束遍历 Generator 函数。**  **return 方法提供参数时，返回该参数；不提供参数时，返回 undefined 。**  **function\* foo(){**  **yield 1;**  **yield 2;**  **yield 3;**  **}**  **var f = foo();**  **f.next();**  **// {value: 1, done: false}**  **f.return("foo");**  **// {value: "foo", done: true}**  **f.next();**  **// {value: undefined, done: true}**  **throw 方法**  **throw 方法可以再 Generator 函数体外面抛出异常，再函数体内部捕获。**  **var g = function\* () {**  **try {**  **yield;**  **} catch (e) {**  **console.log('catch inner', e);**  **}**  **};**    **var i = g();**  **i.next();**    **try {**  **i.throw('a');**  **i.throw('b');**  **} catch (e) {**  **console.log('catch outside', e);**  **}**  **// catch inner a**  **// catch outside b**  **遍历器对象抛出了两个错误，第一个被 Generator 函数内部捕获，第二个因为函数体内部的catch 函数已经执行过了，不会再捕获这个错误，所以这个错误就抛出 Generator 函数体，被函数体外的 catch 捕获。** yield\* 表达式 **yield\* 表达式表示 yield 返回一个遍历器对象，用于在 Generator 函数内部，调用另一个 Generator 函数。**  **function\* callee() {**  **console.log('callee: ' + (yield));**  **}**  **function\* caller() {**  **while (true) {**  **yield\* callee();**  **}**  **}**  **const callerObj = caller();**  **callerObj.next();**  **// {value: undefined, done: false}**  **callerObj.next("a");**  **// callee: a**  **// {value: undefined, done: false}**  **callerObj.next("b");**  **// callee: b**  **// {value: undefined, done: false}**    **// 等同于**  **function\* caller() {**  **while (true) {**  **for (var value of callee) {**  **yield value;**  **}**  **}**  **}** 使用场景实现 Iterator为不具备 Iterator 接口的对象提供遍历方法。 **function\* objectEntries(obj) {**  **const propKeys = Reflect.ownKeys(obj);**  **for (const propKey of propKeys) {**  **yield [propKey, obj[propKey]];**  **}**  **}**    **const jane = { first: 'Jane', last: 'Doe' };**  **for (const [key,value] of objectEntries(jane)) {**  **console.log(`${key}: ${value}`);**  **}**  **// first: Jane**  **// last: Doe**  **Reflect.ownKeys() 返回对象所有的属性，不管属性是否可枚举，包括 Symbol。**  **jane 原生是不具备 Iterator 接口无法通过 for... of遍历。这边用了 Generator 函数加上了 Iterator 接口，所以就可以遍历 jane 对象了。** |

# 18.es6的class语法，其实是构造函数Function的语法糖

## 课堂笔记

### 1.类的基本语法

|  |
| --- |
|  |
|  |

### class里面也可以使用属性表达式

|  |
| --- |
|  |

### 还可以使用Symbol

|  |
| --- |
|  |
|  |

### 调用Symbol方法

|  |
| --- |
|  |
|  |

### class的get和set拦截

|  |
| --- |
|  |
|  |

#### 问题：为什么没有location属性？，有，我们点击这个对象，发现有location 但是他没有值

|  |
| --- |
|  |

#### 那为什么它没有值？

#### 其实我们不要给location赋初始值，如果在get和set方法里面访问this.location会引发死循环

#### 我们拦截的目的是想使用这个属性来渲染页面，或者关联其他属性，因为set和get已经有默认处理，我们不需要干预，我们只是想在这里做一点别的操作，我们把代码改一下

|  |
| --- |
| <body>      <ul id="list">      </ul>      <script>          //拦截类对象的get和set方法          class Person{              constructor(name,age,id){                  this.name = name                  this.age = age                  this.ele = document.querySelector(`#${id}`)              }              //需要被拦截的方法              get location(){                  console.log("get method");                  // return this.location //还不能直接return,会进入死循环                }              set location(value){                  console.log("set method:",value);                }              get html(){                 return this.ele.innerHTML              }              set html(value){                 this.ele.innerHTML = value.map(item=>`<li>${item}</li>`).join("")              }          }          let p = new Person("Kerby",18,"list")          // p.location = "tianjing"          // console.log(p.location);          p.html=["Jackline","Mark","Linda"]      </script>  </body> |
|  |

### 类的静态属性和方法，有两种方式定义，一种方法是类名.新属性 = 属性值，另外一个是在类的内部使用static关键字设置

|  |
| --- |
|  |

#### 注意：js中类的静态属性和静态方法只能通过类名来访问，类的实例访问不了

#### 另外一种方法

|  |
| --- |
|  |

### 注意，在类是静态方法中用this.属性名同样是不能够获取类是实例属性的，因为在类中，this是指向类的名的

|  |
| --- |
|  |
|  |

### 类的继承

#### 一个类继承另外一个类，它会把父类的静态方法和属性也一起继承过来

|  |
| --- |
| <body>      <ul id="list">      </ul>      <script>          //拦截类对象的get和set方法          class Person{              static alias =this.name              static str = function () {                  return `class[${this.name}]`              }              constructor(name,age,id){                  this.name = name                  this.age = age                  this.ele = document.querySelector(`#${id}`)              }              //需要被拦截的方法              get location(){                  console.log("get method");                  // return this.location //还不能直接return,会进入死循环                }              set location(value){                  console.log("set method:",value);                }              get html(){                 return this.ele.innerHTML              }              set html(value){                 this.ele.innerHTML = value.map(item=>`<li>${item}</li>`).join("")              }          }          let p = new Person("Kerby",18,"list")          // p.location = "tianjing"          // console.log(p.location);          p.html=["Jackline","Mark","Linda"]          console.log(p.html);          console.log("testing static property and methods:");          //添加静态方法的另外一种写法          // Person.alias="Person class"          // Person.str = function () {          //     return `class[${this.alias}]`          // }          console.log("alias:",Person.alias);          console.log(Person.str()); //会输出Person类的全部代码          console.log("=====================");          //类的继承          class Student extends Person{              constructor(name,age,score){                  super(name,age) //调用父类的构造函数                  this.score = score              }              showScore(){                  console.log(`Hi,my name is ${this.name},my score is `);              }          }          console.log(Student.str());          console.log(Student.alias);      </script>  </body> |
|  |

#### 如果你在子类的constractor里面没有调用父类构造函数，就会报错

|  |
| --- |
|  |
|  |

#### 子类可以重写父类的方法

|  |
| --- |
|  |
|  |

#### 如果子类重写了父类方法，也可以使用super关键字来调用父类的方法

|  |
| --- |
|  |
|  |

#### 子类也可以覆盖父类的静态方法和静态属性

|  |
| --- |
|  |
|  |

#### 在类的静态方法里面可以用this获取类的静态属性

### 课堂案例，利用面向对象的思想来渲染页面

|  |
| --- |
| <body>      <div class="box1">          <h1></h1>          <ul></ul>      </div>      <div class="box2">          <h1></h1>          <img src="" alt="图片"  width="100px">          <ul></ul>      </div>      <script>          let data1 = {              title:"体育",              list:["体育-1","体育-2","体育-3"]          }          let data2 = {              title:"综艺",              url:"https://pic.maizuo.com/usr/movie/5011ee407fb407d47e333a3935ec33d1.jpg?x-oss-process=image/quality,Q\_70",              list:["综艺-1","综艺-2","综艺-3"]          }          class CreateBox{              constructor(select,data){                 this.box = document.querySelector(select)                 this.title = data.title                 this.list = data.list                 //初始化完成后马上进行渲染                 this.render()              }              render(){                  let h1 = this.box.querySelector("h1")                  let ul  = this.box.querySelector("ul")                  h1.innerHTML = this.title                  ul.innerHTML = this.list.map(item=>`<li>${item}</li>`).join("")              }          }          new CreateBox(".box1",data1)          class CreateBox2 extends CreateBox{              constructor(select,data){                  super(select,data)                  this.url = data.url                  this.render()              }              render(){                  super.render()                  let img = this.box.querySelector("img")                  img.src = this.url              }          }          new CreateBox2(".box2",data2)      </script>  </body> |
|  |

## class参考文档

## class

class是一个语法糖，其底层还是通过 构造函数 去创建的。所以它的绝大部分功能，ES5 都可以做到。新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayName = function() {

return this.name;

}

const xiaoming = new Person('小明', 18);

console.log(xiaoming);

上面代码用ES6的class实现，就是下面这样

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayName() {

return this.name;

}

}

const xiaoming = new Person('小明', 18)

console.log(xiaoming);

// { name: '小明', age: 18 }

console.log((typeof Person));

// function

console.log(Person === Person.prototype.constructor);

// true

constructor方法，这就是构造方法，this关键字代表实例对象。 类的数据类型就是函数，类本身就指向构造函数。

定义类的时候，前面不需要加 function, 而且方法之间不需要逗号分隔，加了会报错。

类的所有方法都定义在类的prototype属性上面。

class A {

constructor() {}

toString() {}

toValue() {}

}

// 等同于

function A () {

// constructor

};

A.prototype.toString = function() {};

A.prototype.toValue = function() {};

在类的实例上面调用方法，其实就是调用原型上的方法。

let a = new A();

a.constructor === A.prototype.constructor // true

### constructor 方法

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

class A {

}

// 等同于

class A {

constructor() {}

}

constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。

class A {

constructor() {

return Object.create(null);

}

}

console.log((new A()) instanceof A);

// false

### 类的实例

实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

### 注意：

1. class不存在变量提升

复制代码

new A(); // ReferenceError

class A {}

因为 ES6 不会把类的声明提升到代码头部。这种规定的原因与继承有关，必须保证子类在父类之后定义。

{

let A = class {};

class B extends A {}

}

上面的代码不会报错，因为 B继承 A的时候，A已经有了定义。但是，如果存在 class提升，上面代码就会报错，因为 class 会被提升到代码头部，而let命令是不提升的，所以导致 B 继承 A 的时候，Foo还没有定义。

1. this的指向 类的方法内部如果含有this，它默认指向类的实例。但是，必须非常小心，一旦单独使用该方法，很可能报错。

## 静态方法

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。 如果在一个方法前，加上 static 关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为"静态方法"。

class A {

static classMethod() {

return 'hello';

}

}

A.classMethod();

console.log(A.classMethod());

// 'hello'

const a = new A();

a.classMethod();

// TypeError: a.classMethod is not a function

A 类的classMethod 方法前有 static关键字，表明这是一个静态方法，可以在 A 类上直接调用，而不是在实例上调用 在实例a上调用静态方法，会抛出一个错误，表示不存在改方法。

如果静态方法包含this关键字，这个this指的是类，而不是实例。

复制代码

class A {

static classMethod() {

this.baz();

}

static baz() {

console.log('hello');

}

baz() {

console.log('world');

}

}

A.classMethod();

// hello

静态方法classMethod调用了this.baz，这里的this指的是A类，而不是A的实例，等同于调用A.baz。另外，从这个例子还可以看出，静态方法可以与非静态方法重名。

父类的静态方法，可以被子类继承。

class A {

static classMethod() {

console.log('hello');

}

}

class B extends A {}

B.classMethod() // 'hello'

## 静态属性

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。 写法是在实例属性的前面，加上static关键字。

class MyClass {

static myStaticProp = 42;

constructor() {

console.log(MyClass.myStaticProp); // 42

}

}

## 继承

Class 可以通过extends关键字实现继承

class Animal {}

class Cat extends Animal { };

上面代码中 定义了一个 Cat 类，该类通过 extends关键字，继承了 Animal 类中所有的属性和方法。 但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个Animal类。 下面，我们在Cat内部加上代码。

class Cat extends Animal {

constructor(name, age, color) {

// 调用父类的constructor(name, age)

super(name, age);

this.color = color;

}

toString() {

return this.color + ' ' + super.toString(); // 调用父类的toString()

}

}

constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

子类必须在 constructor 方法中调用 super 方法，否则新建实例就会报错。 这是因为子类自己的this对象，必须先通过 父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用super方法，子类就得不到this对象。

class Animal { /\* ... \*/ }

class Cat extends Animal {

constructor() {

}

}

let cp = new Cat();

// ReferenceError

Cat 继承了父类 Animal，但是它的构造函数没有调用super方法，导致新建实例报错。

如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

class Cat extends Animal {

}

// 等同于

class Cat extends Animal {

constructor(...args) {

super(...args);

}

}

另一个需要注意的地方是，es5 的构造函数在调用父构造函数前可以访问 this, 但 es6 的构造函数在调用父构造函数(即 super)前不能访问 this。

class A {

constructor(x, y) {

this.x = x;

this.y = y;

}

}

class B extends A {

constructor(x, y, name) {

this.name = name; // ReferenceError

super(x, y);

this.name = name; // 正确

}

}

上面代码中，子类的constructor方法没有调用super之前，就使用this关键字，结果报错，而放在super方法之后就是正确的。

父类的静态方法，也会被子类继承。

class A {

static hello() {

console.log('hello world');

}

}

class B extends A {

}

B.hello() // hello world

## super

super这个关键字，既可以当作函数使用，也可以当作对象使用

### super作为函数调用

super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。

class A {}

class B extends A {

constructor() {

super();

}

}

子类B的构造函数之中的super()，代表调用父类的构造函数。这是必须的，否则 JavaScript 引擎会报错。

注意，super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B的实例，因此super()在这里相当于A.prototype.constructor.call(this)。

class A {

constructor() {

// new.target 指向正在执行的函数

console.log(new.target.name);

}

}

class B extends A {

constructor() {

super();

}

}

new A() // A

new B() // B

在super()执行时，它指向的是子类B的构造函数，而不是父类A的构造函数。也就是说，super()内部的this指向的是B。

### super作为对象调用

**在普通方法中，指向父类的原型对象；** **在静态方法中，指向父类**。

#### super对象在普通函数中调用

复制代码

class A {

p() {

return 2;

}

}

class B extends A {

constructor() {

super();

console.log(super.p()); // 2

}

}

let b = new B();

上面代码中，子类B当中的super.p()，就是将super当作一个对象使用。这时，super在普通方法之中，指向A.prototype，所以super.p()就相当于A.prototype.p()。

这里需要注意，由于super指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过super调用的。

复制代码

class A {

constructor() {

this.p = 2;

}

}

class B extends A {

get m() {

return super.p;

}

}

let b = new B();

b.m // undefined

上面代码中，p是父类A实例的属性，super.p就引用不到它。

如果属性定义在父类的原型对象上，super就可以取到。

复制代码

class A {}

A.prototype.x = 2;

class B extends A {

constructor() {

super();

console.log(super.x) // 2

}

}

let b = new B();

上面代码中，属性x是定义在A.prototype上面的，所以super.x可以取到它的值。

#### super对象在静态方法中调用

用在静态方法之中，这时super将指向父类，而不是父类的原型对象。

复制代码

class Parent {

static myMethod(msg) {

console.log('static', msg);

}

myMethod(msg) {

console.log('instance', msg);

}

}

class Child extends Parent {

static myMethod(msg) {

super.myMethod(msg);

}

myMethod(msg) {

super.myMethod(msg);

}

}

Child.myMethod(1); // static 1

const child = new Child();

child.myMethod(2); // instance 2

上面代码中，super在静态方法之中指向父类，在普通方法之中指向父类的原型对象。

另外，在子类的静态方法中通过super调用父类的方法时，方法内部的this指向当前的子类，而不是子类的实例。

复制代码

class A {

constructor() {

this.x = 1;

}

static print() {

console.log(this.x);

}

}

class B extends A {

constructor() {

super();

this.x = 2;

}

static m() {

super.print();

}

}

B.x = 3;

B.m() // 3

上面代码中，静态方法B.m里面，super.print指向父类的静态方法。这个方法里面的this指向的是B，而不是B的实例。

## 总结

* class是一个语法糖，其底层还是通过 构造函数 去创建的。
* 类的所有方法都定义在类的prototype属性上面。
* 静态方法：在方法前加static，表示该方法不会被实例继承，而是直接通过类来调用。
* 静态属性：在属性前加static，指的是 Class 本身的属性，而不是定义在实例对象（this）上的属性。
* es5 的构造函数在调用父构造函数前可以访问 this, 但 es6 的构造函数在调用父构造函数(即 super)前不能访问 this。
* super
  + 作为函数调用，代表父类的构造函数
  + 作为对象调用，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

## 再来几道题检查一下

### 1. 下面代码输出什么

复制代码

class Person {

constructor(name) {

this.name = name

}

}

const member = new Person("John")

console.log(typeof member)

答案：object

解析： 类是构造函数的语法糖，如果用构造函数的方式来重写Person类则将是：

复制代码

function Person() {

this.name = name

}

通过new来调用构造函数，将会生成构造函数Person的实例，对实例执行typeof关键字将返回"object"，上述情况打印出"object"。

### 2. 下面代码输出什么

复制代码

class Chameleon {

static colorChange(newColor) {

this.newColor = newColor

return this.newColor

}

constructor({ newColor = 'green' } = {}) {

this.newColor = newColor

}

}

const freddie = new Chameleon({ newColor: 'purple' })

freddie.colorChange('orange')

答案：TypeError

解析： colorChange 是一个静态方法。静态方法被设计为只能被创建它们的构造器使用（也就是 Chameleon），并且不能传递给实例。因为 freddie 是一个实例，静态方法不能被实例使用，因此抛出了 TypeError 错误。

### 3.下面代码输出什么

复制代码

class Person {

constructor() {

this.name = "Lydia"

}

}

Person = class AnotherPerson {

constructor() {

this.name = "Sarah"

}

}

const member = new Person()

console.log(member.name)

答案："Sarah"

解析： 我们可以将类设置为等于其他类/函数构造函数。 在这种情况下，我们将Person设置为AnotherPerson。 这个构造函数的名字是Sarah，所以新的Person实例member上的name属性是Sarah。

# 19.es6module语法

## 模块化出现之前js开发的痛点

### 1.以前的js中，所有在一个js文件里面定义的函数外面都能够调用，毫无保密可言

### 2.我们新建一个目录结构如下，在moduleDemo文件夹下面新建两个js文件，都有test方法，我们在module.html文件中把这两个js文件都导入，然后在一个script标签在调用test，调用的是2.js的test方法，因为js的规则是同名属性后面的覆盖前面的

|  |
| --- |
|  |
|  |
|  |
|  |

### 3.此时引入js文件的顺序会影响函数的调用，如果你在引入之前调用，会报错

### 解决这些问题的办法就是使用模块化开发

## 模块化语法的引入

### 1.模块化基本语法，在script标签的src=“xx.js” 后面添加type=”module”

|  |
| --- |
|  |

#### 注意,此时你直接调用js文件里面的方法会报错，说方法没有定义，因为此时它不是全局的了，

|  |
| --- |
|  |
|  |

#### 那怎么才能调用到这个方法，这种引入方法是没有办法的，因为它把整个js都封闭起来了

### 2.改进引入，其实在1.js和2.js中需要导出东西的，我们来导出一下

|  |
| --- |
|  |
|  |

#### 然后在没有src属性并且类型是module的script标签里面导入对应的方法，没有导出的方法不能导入

|  |
| --- |
|  |
|  |

#### 其实导入的时候名字不必和导出的名字一样

|  |
| --- |
|  |
|  |

#### 注意export default只能写一次，所以如果需要导出多个函数需要把他们封装成一个对象

|  |
| --- |
|  |

#### 此时在需要导入的地方用对象来接收

|  |
| --- |
| <body>      <div>hello</div>      <script type="module">            import obj from "./1.js"          import test2 from "./2.js"            //模块化          /\*              异步加载              私密不漏              重名不怕              依赖不乱          \*/      //    test()         obj.func1()         test2()      </script>  </body> |
|  |

#### 这种导入方法就是：在哪里需要就在哪里导入

### 3.还有另外一种导入导出方法，先修改一下js文件子类是1-2.js 和2-2.js

#### 1-2.js

|  |
| --- |
| let div = document.querySelector("div")  console.log(div);  function func1() {      console.log("inside 1.js func1...");  }  function func2() {      console.log("inside func2...");  }  function test() {      console.log("1.js---test method");  }  export  {      func1,      func2,      test  } |

#### 2-2.js

|  |
| --- |
| function test() {      console.log("2.js---test method");  }  function fun1() {      console.log("inside 2.j2 fun1...");  }  export  {      test,      fun1  } |

### 然后我们在module2.html里面导入这两个文件，此时就能够使用解构赋值的方式来导入

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>es6中的module语法2</title>  </head>  <body>      <script type="module">          //第二种导入导出js方法          import {func1,func2 } from './1-2.js'          import {fun1,test} from './2-2.js'          func1()          fun1()          test()        </script>  </body>  </html> |

### 这种导入方法支持改名

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>es6中的module语法2</title>  </head>  <body>      <div>hello js</div>      <script type="module">          //第二种导入js方法          import {func1,func2,test as test1} from './1-2.js'          import {fun1,test} from './2-2.js'          func1()          fun1()          test()          test1()      </script>  </body>  </html> |
|  |

#### 使用这种方法导入，接收变量的名称必须和导出那里的变量名称相同否则报错，这种方法通过了改名机制来解决命名冲突，而且可以根据需要导入，别人导出了10给函数，我可以只导入其中的一个来用

### 这种写法还有一个变种，可以这么写import \* as obj from ‘./1-2.js’

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>es6中的module语法2</title>  </head>  <body>      <div>hello js</div>      <script type="module">          //第二种导入js方法          // import {func1,func2,test as test1} from './1-2.js'          import \* as obj from './1-2.js'          import {fun1,test} from './2-2.js'          obj.func1()          obj.test()          fun1()          test()      </script>  </body>  </html> |
|  |

### 用于第二种导入方式的导出方式还可以这么写

|  |
| --- |
|  |

### 4.export default 和export可以同时使用，不过记住，export default只能写一次，export可以多次

#### 如我们新一个4.js，内容如下

|  |
| --- |
| export function info(){      console.log("this is 4.js");  }  export default function(){        console.log("i was inside 4.js");      console.log("i was exported using export default");  } |

#### 混合导出也需要混合导入，比较麻烦，但是比较灵活

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>es6中的module语法2</title>  </head>  <body>      <div>hello js</div>      <script type="module">          //第二种导入js方法          // import {func1,func2,test as test1} from './1-2.js'          import \* as obj from './1-2.js'          import {fun1,test} from './2-2.js'          import {info} from './4.js'  //导入使用export导出的函数          import my from './4.js' //导入使用export default导出的函数          obj.func1()          obj.test()          fun1()          test()          console.log("=================");          info()          my()      </script>  </body>  </html> |
|  |

### 混合导入还可以这么写

|  |
| --- |
| <body>      <div>hello js</div>      <script type="module">          //第二种导入js方法          // import {func1,func2,test as test1} from './1-2.js'          import \* as obj from './1-2.js'          import {fun1,test} from './2-2.js'          // import {info} from './4.js'  //导入使用export导出的函数          // import my from './4.js' //导入使用export default导出的函数          import my,{info} from './4.js'          obj.func1()          obj.test()          fun1()          test()          console.log("=================");         info()         my()      </script>  </body> |

#### 效果是一样的

|  |
| --- |
|  |

#### 其实react中有很多这样子导入的写法

### 课堂案例，把上一节class实例的面向对象案例拿过来我们练习一下，我们新建一个module-project文件夹，在整个文件夹里面新建一个module案例.html文件，把上一节的代码粘贴过来，然后把两个class文件抽取出来放到一个class.js文件夹中，然后需要把html文件的script标签的类型改为module再导入这两个类

#### class.js

|  |
| --- |
| class CreateBox{      constructor(select,data){         this.box = document.querySelector(select)         this.title = data.title         this.list = data.list         //初始化完成后马上进行渲染         this.render()      }      render(){          let h1 = this.box.querySelector("h1")          let ul  = this.box.querySelector("ul")          h1.innerHTML = this.title          ul.innerHTML = this.list.map(item=>`<li>${item}</li>`).join("")      }  }  class CreateImgBox extends CreateBox{      constructor(select,data){          super(select,data)          this.url = data.url          this.render()      }      render(){          super.render()          let img = this.box.querySelector("img")          img.src = this.url      }  }  export {      CreateBox,      CreateImgBox  } |

#### module案例.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>类继承案例-模块化写法</title>  </head>  <body>      <div class="box1">          <h1></h1>          <ul></ul>      </div>      <div class="box2">          <h1></h1>          <img src="" alt="图片"  width="100px">          <ul></ul>      </div>      <script type="module">          import {CreateBox,CreateImgBox} from './class.js'          let data1 = {              title:"体育",              list:["体育-1","体育-2","体育-3"]          }          let data2 = {              title:"综艺",              url:"https://pic.maizuo.com/usr/movie/5011ee407fb407d47e333a3935ec33d1.jpg?x-oss-process=image/quality,Q\_70",              list:["综艺-1","综艺-2","综艺-3"]          }          new CreateBox(".box1",data1)          new CreateImgBox(".box2",data2)      </script>  </body>  </html> |

#### 效果是一样的

|  |
| --- |
|  |

# 20 .nodejs中的模块化语法

## 1.es6的import和nodejs的import是不一样

|  |
| --- |
|  |

## 课堂笔记，前提是你已经安装了nodejs

### 1.我们新建一个nodejs\_import\_export文件夹，在里面新建一个index.js文件和一个1.js文件

|  |
| --- |
|  |

#### 1.js导出函数

|  |
| --- |
| function test() {      console.log("test function inside 1.js");  }  module.exports = test |

#### index.js，导入1.js中的函数

|  |
| --- |
| let test = require('./1')  test() |

#### 然后打开终端，运行node index.js,效果如下

|  |
| --- |
|  |

#### 注意。如果你导出两个函数，但是你导入时只是用一个变量来接收，这样子后面的函数就覆盖前面的函数

|  |
| --- |
|  |
|  |

#### 那么怎么办？也可以导出对象的

|  |
| --- |
|  |

#### 然后这么导入

|  |
| --- |
|  |
|  |

#### 还有另外一种写法

|  |
| --- |
|  |

#### 这种写法的导入方法

|  |
| --- |
|  |
|  |

#### 还是一样的导入方法，

#### 还可以使用解构赋值导入，如我们新一个2.js文件里面有一个Person类和一个printPerson方法

##### 2.js

|  |
| --- |
| class Person{      constructor(name,gender,age){          this.name = name          this.age = age          this.gender =gender      }      intro(){        console.log(`hello,my name is ${this.name},i am ${this.gender} and i am ${this.age} years old`);      }  }  function printPerson(p) {      console.log(`Person[name=${p.name},gender=${p.gender},age=${p.age}]`);  }  exports.Person = Person  exports.printPerson = printPerson |

#### 然后我们在index.js中使用解构赋值导入

##### index.js

|  |
| --- |
| let {Person,printPerson} = require('./2')  let p2 = new Person("Lily","female",30)  p2.intro()  printPerson(p2) |

#### 效果如下

|  |
| --- |
|  |

#### 现在，在nodejs里面也可以支持es6的导入导出，不过你需要在你的文件夹下面新建一个package.json把整个项目变为nodejs项目，也就是在项目根目录下面执行npm init，

|  |
| --- |
|  |

#### 然后打开package.json文件，添加一个type，值是module

|  |
| --- |
|  |

#### 然后你就可以在index.js里面使用es6的模块化语法导入了，不过前提是你的js文件有模块化导出，我们新建一个3.js，内容如下

|  |
| --- |
|  |

#### 然后我们在index.js里面使用模块化导入

|  |
| --- |
| //使用es6模块化导入  import { Student,printStudent } from "./3.js";  let s = new Student("李连杰","male",60)  s.intro()  console.log("==============");  printStudent(s) |

#### 效果

|  |
| --- |
|  |

#### 需要注意：一旦你把他改为模块化语法了，原来的COMMONJS规范又不支持了，所以还是建议你不要改为es6模块化，因为node\_module里面的都是COMMONJS规范的文件。上面的例子只需要了解即可

#### 还有一种方法，不需要package.json,就是把需要使用es6模块化的文件的后缀名改为.mjs

|  |
| --- |
|  |

#### vuejs和reactjs是两种规范都支持，但是你不能混合使用，只能使用其中一种