第二部分 进阶内容

第7章 迭代器与生成器

（ES6新增）

7.1迭代器

1. 实现Iterable接口（可迭代协议）要求同时具备两种能力：支持迭代的自我识别能力和创建实现Iterable接口的对象的能力；
2. 在函数调用的时候，浏览器每次都会传递进两个隐式参数：函数的上下文对象this；封装实参的对象arguments
3. 很多内置类型都实现了`Iterable`接口：字符串、数组、映射、集合、arguments对象、NodeList等DOM集合类型；

```
let str='abc';
let arr=['a','b','c'];
let map=new Map().set('a',1).set('b',2).set('c',3);
let set=new Set().add('a').add('b').add('c');
let els=document.querySelectorAll('div');
console.log(str[Symbol.iterator]);//ƒ [Symbol.iterator]() { [native code] }
console.log(str[Symbol.iterator]());//String Iterator  //__proto__: String
Iterator
```

4. next()方法每次调用，都会返回IteratorResult对象，对象包含两个属性：done和value;`done:true`状态表示耗尽；

7.2生成器

1. 生成器：生成器是一种返回迭代器的函数，通过function关键字后的星号(*)来表示，函数中会用到新的关键字yield。
2. yield关键字可以让生成器停止和开始执行；通过yield关键字退出的生成器函数会处于`done:false`状态，通过return关键字退出的生成器函数会处于`done:true`状态；yeild关键字只能在生成器函数内部使用；
3. 生成器拥有在一个函数内暂停和恢复代码执行的能力，比如，使用生成器可以自定义迭代器和实现协程。

   对于生成器函数产生的迭代器来说，这个值就是生成器函数返回的值；

   图数据结构非常适合递归遍历，而递归器恰好非常合用；
4. 一个实现`Interator`接口的对象一定有next()方法，可选的`return()`和`throw()`

```
function* foo(x) {
    var y = 2 * (yield(x + 1));
    var z = yield(y/3);
    return (x + y + z);
}
let a = foo(5);
// console.log(a.next());//{value:6,done:flase}
// console.log(a.next());//{value:NAN,done:flase}
// console.log(a.next());//{value:NAN,done:true}
console.log(a.next());//{value:6,done:flase}
console.log(a.next(12));//{value:8,done:flase}
console.log(a.next(13));//{value:42,done:true}

/*没有提供参数时，第二次调用next()，导致y的值为2*undefined（即NaN）， 第三次调用z为undefined ,返回对象的value属性等于 5+NaN+undefined 为NaN.
提供参数时，第二次调用next()，导致y的值24，第三次调用，使得z的值为13 ，所有返回对象的value值为5+24+13=42
由于next()的参数表示上一条yield语句的返回值，所有第一次使用next方法时，传递的参数是无效的。
*/
```

第八章 对象、类与面向对象编程

1、理解对象

属性分为两种：

- 数据属性（保存数据值的位置）；（4个特性：[[Configurable]]、[[Enumberable]]、[[Writable]]、[[Value]]）
- 访问器属性（包含getter和setter函数，非必需）；（4个特性：[[Configurable]]、[[Enumberable]]、[[Get]]、[[Set]]）

为了将某个特性标识为内部特性，规范会用两个中括号将特性的名称括起来，如[[Enumerable]]；

2、理解对象创建过程

2.1 原型：

- **构造函数、实例、原型三者之间的关系**：每个构造函数都有一个原型对象，原型有一个constructor属性指向构造函数，而实例有一个内部指针指向原型；
- 原型中包含的引用值会在所有实例中**共享**，这也是为什么属性通常会在构造函数中定义而不会在原型上的原因。
- 同一个构造函数创建的两个实例共享同一个原型对象；
- `Object.getPrototypeOf()`返回参数的内部特性[[Prototype]]的值；
- `Object.setPrototypeOf()`和`Object.creat()`；
- `hasOwnProperty()`确定某个属性是否在实例上；
- `Object.getOwnPropertyDescriptor()`只对实例属性有效；

```
console.log(Person.prototype.constructor===Person);  //true
```

```
console.log(Person.prototype._proto_._proto_===null);//true
//Object原型的原型是null
```

- `Object.getOwnPropertyDescriptors`用法：
  - 是配合`Object.create()`方法，将对象属性克隆到一个新对象上面。这属于浅拷贝；
  - 实现一个对象继承另一个对象；
  - 用来实现Mixin（混入模式）；

2.2 构造函数首字母一般大写（约定俗成）；

构造函数的问题：其定义的方法会在每个实例上都创建一遍，所以可以把函数定义放在构造函数外部，这样虽然解决了**相同逻辑重复定义**的问题，但是是**全局作用域**也被搞乱了，这个新问题可以用过**原型模式**解决，使用原型定义的属性和方法是由所有实例共享的；但是由于原型的共享问题，开发中通常并不单独使用；开发中常用的继承模式是寄生组合式继承；

2.3 对象迭代

ES2017新增静态方法：`Object.values()`返回对象值的数组和`Object.entries()`返回键值对的数组；

3、理解继承：

- 接口继承和**实现继承**，前者继承方法签名，后者继承实际的方法；js只支持实现继承，通过原型链实现；
- 继承主要有五种方法：原型链、借用构造函数、组合继承、寄生式继承、寄生组合式继承；
- **寄生组合式继承的基本思路：**不必为了指定子类型的原型而调用超类型的构造函数，我们所需的无非就是超类型原型的一个副本而已。本质上，就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。

```
//寄生组合式继承的基本模式如下所示：
function inheritPrototype(subType, superType){
    var prototype = object(superType.prototype);  // 创建对象
    prototype.constructor = subType;  // 增强对象
    subType.prototype = prototype;  // 指定对象
}
function SuperType(){
```

```
    // 超类型的构造函数
}
function SubType(){
    SuperType.call(this);  // 借用构造函数
}
inheritPrototype(SubType, SuperType);
var instance = new SubType();
```

4、理解类

4.1类定义主要有两种方式：类声明、类表达式

```
class Person{}//类声明
const Animal=class{}//类表达式
```

4.2类可以包含构造函数方法、实例方法、获取函数、设置函数、静态类方法；默认情况下，类定义的代码都是在严格模式下执行的；

4.3使用new调用类的构造函数会执行下列操作：

- 在内存中创建一个新对象；
- 这个新对象内部的[[Prototype]]指针被赋值为构造函数的prototype属性；
- 构造函数内部的this被赋值为这个新对象（即this指向新对象）；
- 执行构造函数内部的代码（给新对象添加属性）；
- 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象；

4.4类构造函数和构造函数的主要区别是：调用类构造函数必须使用new操作符，而普通构造函数如果不使用new调用，那么会以的this（通常是window）作为内部对象。调用类构造函数时，如果忘了使用new则会抛出错误；

4.5 类继承

- ES6支持单继承，使用extends关键字，就可以继承任何拥有[[Construct]]和原型的对象；
- 派生类的方法可以通过super关键字引用它们的原型，这个关键字只能在派生类中使用，而且仅限于类构造函数、实例方法和静态方法内部。
- 使用super要注意的几个问题：
    - super只能在派生类构造函数和静态方法中使用；
    - 不能单独使用super关键字；
    - 调用super()会调用父类构造函数，并将返回的实例赋值给this；
    - super()的行为如同调用构造函数，如果需要给父类构造函数传参，则需要手动传入；
    - 如果没有定义类构造函数，在实例化派生类时会调用super(),而且会传入所有传给派生类的参数；
    - 在类构造函数中，不能在调用super()之前引用this；
    - 如果在派生类中显式定义了构造函数，则要么必须在其中调用了super()，要么必须在其中返回一个对象；

第九章 代理(Proxy) 和 反射(Reflection)

代理、捕获器方法、反射API方法（只要在代理上调用，所有捕获器都会拦截它们对应的反射API操作）；

1、代理

定义：是一种可以拦截并改变底层JS引擎操作的包装器。代理对象可以作为抽象的目标对象来使用（代理是目标对象的抽象）；

（1）代理是使用Proxy构造函数创建的；接受两个参数：目标对象（target）和处理程序对象（handler）

```
const proxy=new Proxy(target,handler);//缺少任何一个参数都会抛出TypeError
//handler实际上是一个对象，其记录了需要代理的行为，以及代理的方法。
```

（2）最简单的代理是空代理；（在代理对象上执行的任何操作实际上都会应用到目标对象上）；给目标属性赋值会反映在两个对象上，因为两个对象访问的是同一个值；给代理属性赋值会反映在两个对象上，因为这个复制会转移到目标对象上；

（3）使用代理的主要目的是可以定义捕获器（trap）；捕获器就是在处理程序对象中定义的"基本操作的拦截器"；所有捕获器都可以访问相应的参数，基于这些参数可以重建被捕获方法的原始行为，实际上，开发者可以通过调用全局Reflect对象上（封装了原始行为）的同名方法来轻松创建；

代理的问题与不足：

this问题（方法中的this通常指向调用这个方法的对象）；代理与内置引用类型（如Array）的实例通常可以很好的协同，但是有些ES的内置类型可能会依赖代理无法控制的机制，结果导致在代理上调用某些方法会出错，如Date类型；

2、反射机制

反射机制指的是程序在运行时能够获取自身的信息。（代理可以捕获13种不同的基本操作）

代理陷阱	覆写的特性	默认特性
get	读写一个属性值	Reflect.get()
set	写入一个属性	Reflect.set()
has	in操作	Reflect.has()
deleteProperty	delete操作符	Reflect.deleteProperty()
getAPrototypeof	Object.getAPrototypeof ()	Reflect.getAPrototypeof ()
setAPrototypeof	Object.setAPrototypeof ()	Reflect.setAPrototypeof ()
isExtensible	Object.isExtensible()	Reflect.isExtensible()
preventExtensions	Object.preventExtensions()	Reflect.preventExtensions()
getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor()	Reflect.getOwnPropertyDescriptor()
defineaProperty	Object.defineaProperty()	Reflect.defineaProperty()
ownKeys	Object.keys()、Object.getOwnPropertyNames()和Object.getOwnPropertySysmbols()	Reflect.ownKeys()
apply	调用一个函数	Reflect.apply()
construct	用new调用一个函数	Reflect.construct()

```
//get()捕获器：这段代码允许添加新的属性，并且此后可以正常读取该属性的值，但当读取的属性并不存在时，程序抛出了一个错误，而不是将其默认为undefined。
let proxyObj = new Proxy(targetObj, {
```

```
  set: set,
  get: get
});

/* 定义 get 陷阱函数 */
function get(trapTarget, key, receiver) {
  if (!(key in receiver)) {
    throw new TypeError("Property " + key + " doesn't exist.");
  }
  return Reflect.get(trapTarget, key, receiver);
}

console.log(proxyObj.count); // 123
console.log(proxyObj.newcount) // TypeError: Property newcount doesn't exist.
```

3、捕获器不变式：

其因方法不同而异，但通常都会防止程序的行为必须遵循“捕获器不变式（trap invariant）”；

4、代理的应用场景：

跟踪属性访问、隐藏属性、阻止修改或删除属性、函数参数验证、构造函数参数验证、数据绑定、观察对象等；

第十章 函数

使用闭包实现私有变量

1、函数定义

```
//函数声明（有提升）
function sum(num1,num2){
    return num1+num2;
}
//函数表达式（无提升）
let sum=(num1,num2){
    return num1+num2;
};
//匿名函数
function(){}//报错
(function(){})//不会报错
(function(){})()//立即执行
//箭头函数：不能使用arguments、super、new.target,也不能用作构造函数，且其没有prototype属性；
let sum=(a,b)=>{return a+b;};
//使用Function构造函数
let sum=new Function("num1","num2","return num1+num2");//不推荐
```

- arguments对象是一个类数组对象，其长度是根据传入参数的个数确定的，而不是定义函数时给出的命名参数个数确定的；
- js无重载，定义的同名的两个参数，后定义的会覆盖先定义的；
- 暂时性死区：前面定义的参数不能引用后面定义的；

2、函数内部

函数内部对象：arguments和this；属性：new.target（ES6新增）

- arguments.callee():是指向arguments对象所在函数的指针；
- 在事件回调或定时回调中调用某个函数时，此时将回调函数写成箭头函数就可以解决this指向问题，因为在箭头函数中，this指向的是定义箭头函数的上下文；
- 严格模式下，调用函数如果没有指定上下文对象，this不会指向window，会变成undefined；除非使用apply和call把函数指定给一个对象；

3、尾调用优化

ES6新增内存管理机制，让js引擎在满足条件时可以重用栈帧；尾调用：即外部函数的返回值是一个内部函数的返回值；

尾调用优化的条件：

- 代码在严格模式下执行；（之所以要求严格模式，主要是因为在非严格模式下，函数调用中允许使用f.arguments和f.caller，而它们都会引用外部函数的栈帧）
- 外部函数的返回值实对尾调用函数的调用；
- 尾调用函数返回后不需要执行额外的逻辑；
- 尾调用不是引用外部函数作用域中自由变量的闭包；

4、特权方法

特权方法是能够访问函数私有变量（及私有函数）的公有方法；

特权方法可以使用构造函数或原型模式通过自定义类型中实现，也可以使用模块模式或模块增强模式在单例对象上实现；

```
//创建方式有两种：在构造函数中实现（闭包）
function MyObject(){
    let privateVariable=10;
    function privateFunction(){
        return false;
    }
    this.publicMethod=function(){
        privateVariable++;
        return privateFunction();
    };
}
//通过使用私有作用域定义私有变量和函数来实现
(function (){
    //私有变量和私有函数
    let privateVariable=10;
    function privateFunction(){
        return false;
    }
    //构造函数
    MyObject=function(){};
   MyObject.prototype.publicMethod=function(){
        privateVariable++;
        return privateFunction();
    };
})();
```

第十一章 期约与异步函数

ES6期约模式在ES函数中的应用：异步函数（语法关键字：async/await）是ES8规范新增的；**异步函数是JavaScript项目中使用最广泛的特性之一**；

1 、异步编程

同步行为对应内存中顺序执行的处理器指令；

异步行为类似于系统中断，即当前进程外部的实体可以触发代码执行；

2、期约

2.1 ES6新增的引用类型Promise()期约，期约是有状态的对象（期约的状态是私有的；）：

- 待定（pending）；【期约最初始的状态】”表示尚未开始或者正在执行中“
- 兑现（fulfilled或解决resolved）；”表示已经成功“
- 拒绝（rejected）；”表示没有成功完成“

2.2 执行器函数的职责：

- 初始化期约的异步行为；
- 控制状态的最终转换；【控制期约状态的转换通过调用它的两个两个函数参数resolve（）和reject（）实现的】

  resolve（）会把状态切换回兑现；调用reject会把状态切换回拒绝；

2.3 Promise. resolve（）：实例化一个解决的期约；如果传入的参数本身是一个期约，那么它的行为类似于一个空包装（幂等方法）；

Promise. reject（）实例化一个拒绝的期约并抛出一个异步错误；如果给它传入一个期约对象，则这个期约会成为它返回的拒绝期约的理由；

```
let p1=new Promise((resolve,reject)=>resolve());
let p2=Promise.resolve();//p1等价于p2
```

2.4 期约实例的方法是连接外部同步代码与内部异步代码之间的桥梁。

`Promise.prototype.then()`：这个`then()`方法接收最多两个参数: `onResolved`处理程序和`onRejected`处理程序。这两个参数均可选，如果提供，则会在期约分别进入"兑现"和"拒绝"状态时执行。（这两个操作是互斥的，因为因为期约只能转换为最终状态一次）；

```
let p1 = new Promise(() => {});
let p2 = p1.then();
setTimeout(console.log, 0, p1 === p2); // false：因为then该方法返回一个新的期约实例
```

`Promise.prototype.catch()`和`Promise.prototype.finally()`

2.5 期约的组合通过两种方式实现：

- 期约连锁：一个期约接着一个期约地拼接；

  期约连锁可以构建有向非循环图；期约的处理程序是按照他们添加的顺序执行的；
- 期约合成：多个期约组合为一个期约；
  - `Promise.all（）`：会在一组期约全部解决之后在解决；该静态方法会接受一个可迭代对象，返回一个新期约；

- `Promise.race()`：返回一个包装期约，是一组集合中最先解决或拒绝的期约的镜像；该静态方法会接受一个可迭代对象，返回一个新期约；

3、异步函数

又称”`async/await`“：旨在解决利用异步结构组织代码的问题；

- 关键字async用于声明异步函数，如果异步函数使用了`return`关键字返回了值（如果没有return则会返回undefined），这个值会被Promise.resolve()包装成一个期约对象。异步函数始终返回期约对象。
- 关键字await可以暂停异步函数代码的执行，等待期约来解决；其必须在异步函数中使用，不能在顶级上下文如标签或模块中使用；