

JavaScript ES6实现继承

王红元 coderwhy

目录

content



1 原型继承关系图

2 class方式定义类

3 extends实现继承

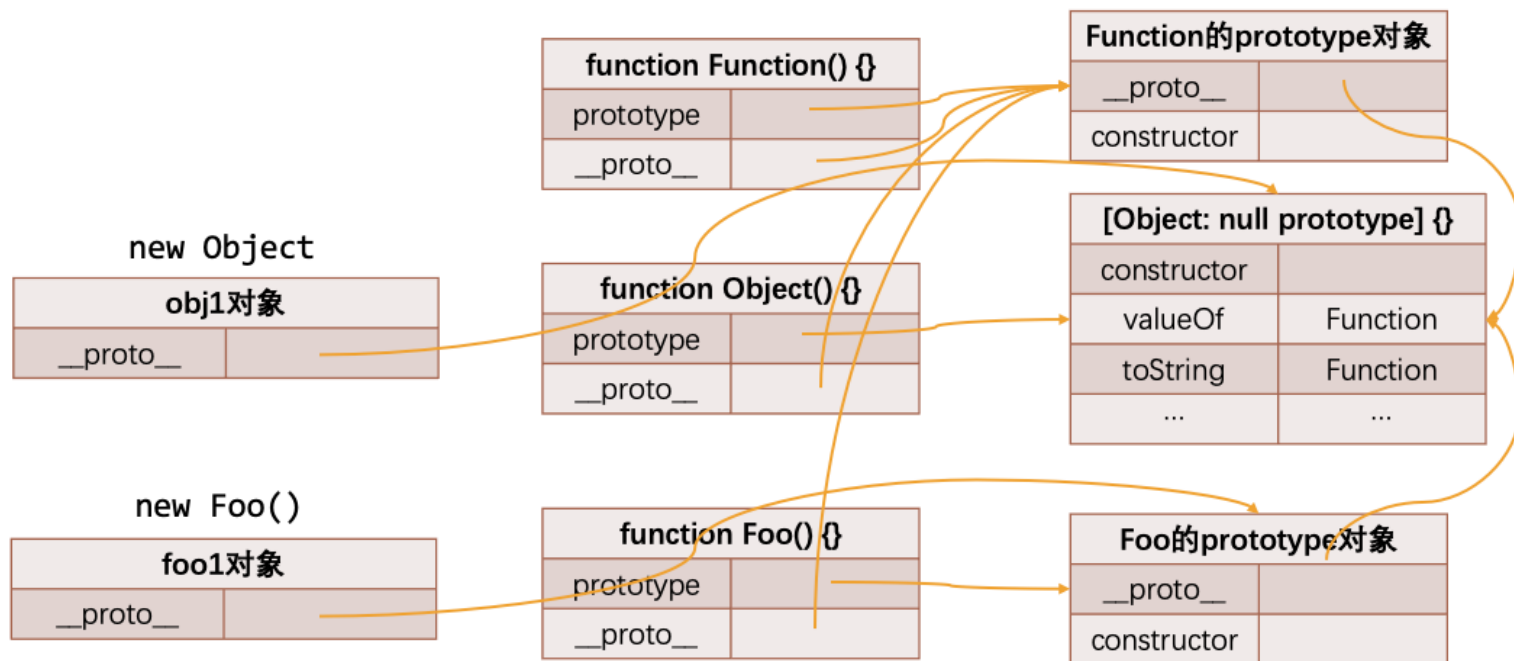
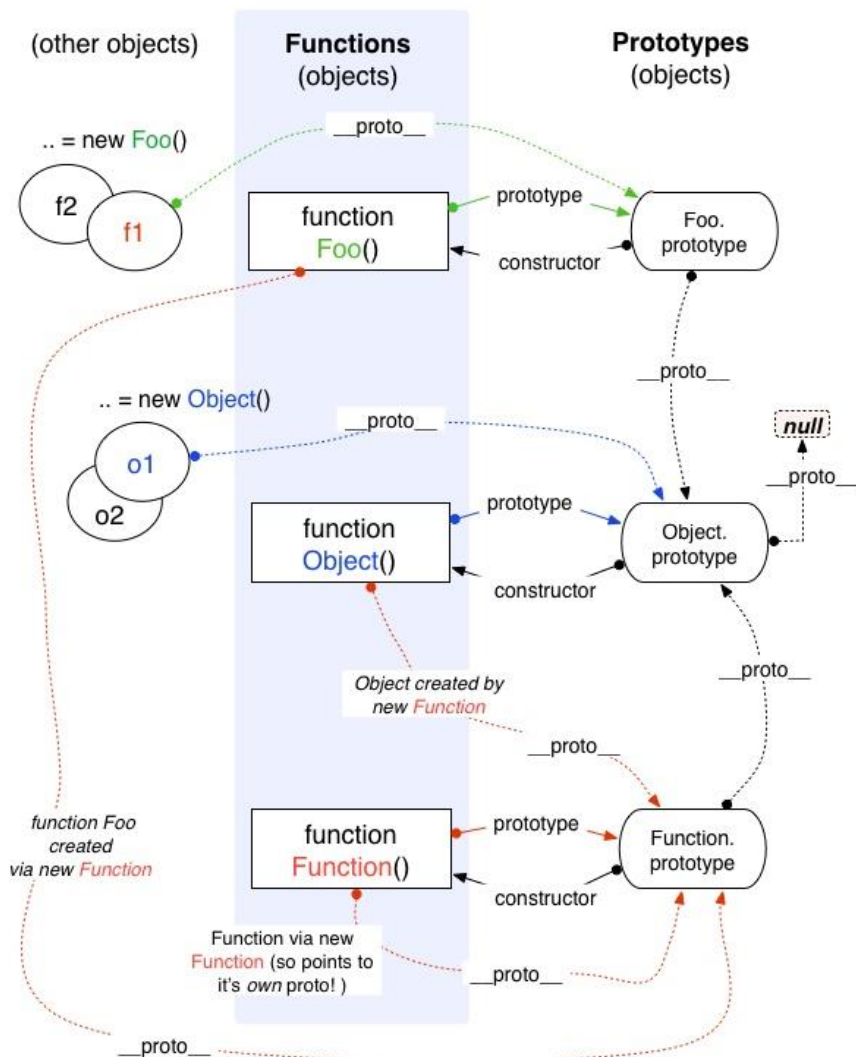
4 Babel的ES6转ES5

5 面向对象多态理解

6 ES6对象的增强

原型继承关系

JavaScript Object Layout [Hursh Jain/mollypages.org]



创建对象的内存表现

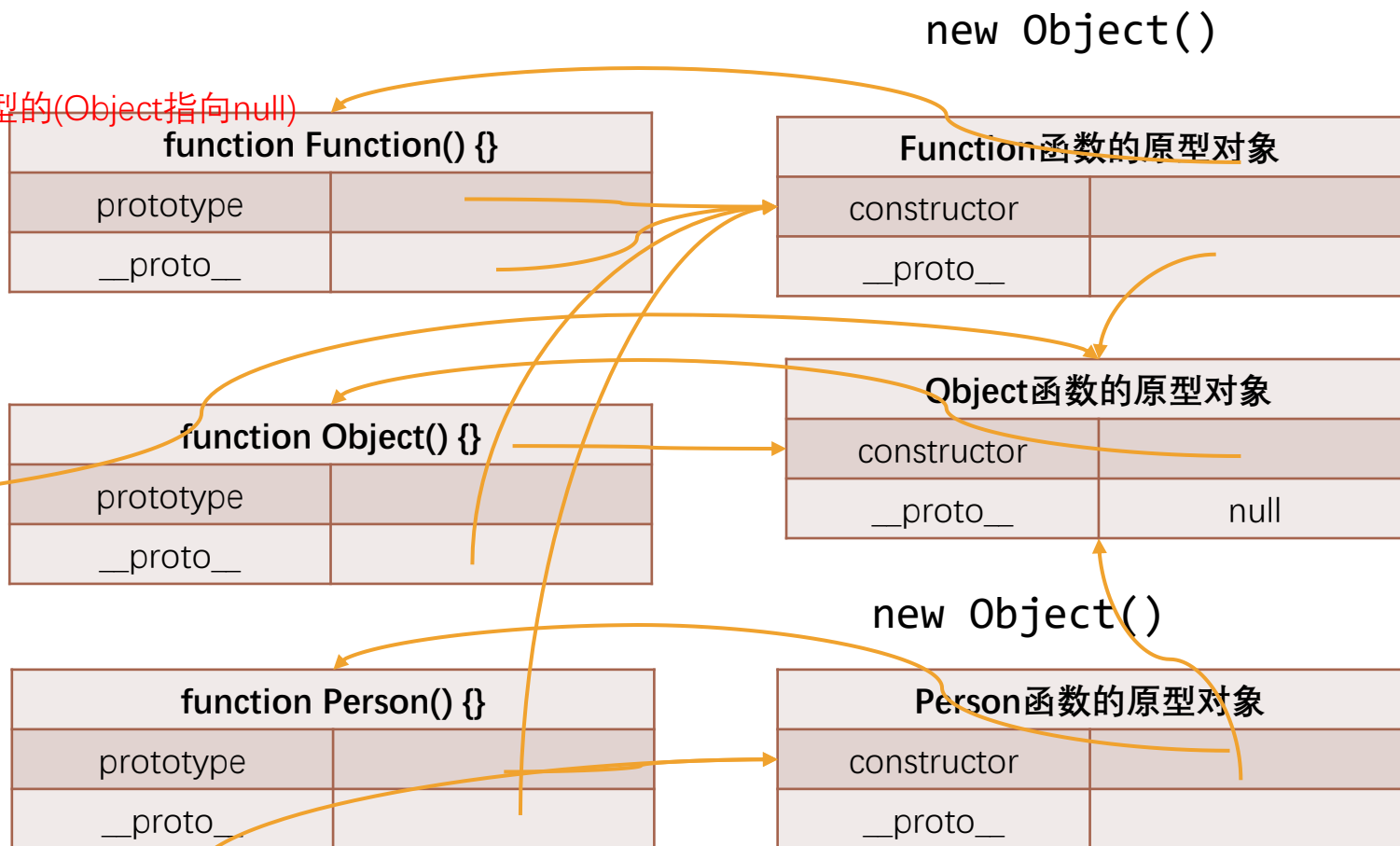
- 1.p1是Person的实例对象
 - 2.obj是Object的实例对象
 - 3.Function/Object/Foo都是Function的实例对象
 - 4.原型对象默认创建时, 隐式原型都是指向Object的显式原型的(Object指向null)
- * 推导另外一个结论: Object是Person/Function的父类

```
var obj = {}
```

obj对象	
__proto__	

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

p1对象	
__proto__	



认识class定义类

- 我们会发现，按照前面的构造函数形式创建 **类**，不仅仅和编写普通的函数过于相似，而且代码并不容易理解。
 - 在ES6（ECMAScript2015）新的标准中使用了class关键字来直接定义类；
 - 但是类本质上依然是前面所讲的构造函数、原型链的语法糖而已；
 - 所以学好了前面的构造函数、原型链更有利于我们理解类的概念和继承关系；
- 那么，如何使用class来定义一个类呢？
 - 可以使用两种方式来声明类：类声明和类表达式；

```
class Person {  
}  
  
var Student = class {  
}
```

- 注意：类中定义的多个内容不需要使用，进行分割；

类和构造函数的异同

- 我们来研究一下类的一些特性：你会发现它和我们的构造函数特性其实是一致的；

```
var p = new Person

console.log(Person) // [class Person]
console.log(Person.prototype) // {}
console.log(Person.prototype.constructor) // [class Person]

console.log(p.__proto__ === Person.prototype) // true

console.log(typeof Person) // function
```

类的构造函数

■ 如果我们希望在创建对象的时候给类传递一些参数，这个时候应该如何做呢？

- 每个类都可以有一个自己的构造函数（方法），这个方法的名称是固定的`constructor`；
- 当我们通过`new`操作符，操作一个类的时候会调用这个类的构造函数`constructor`；
- 每个类只能有一个构造函数，如果包含多个构造函数，那么会抛出异常；

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

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

类的实例方法

- 在上面我们定义的属性都是直接放到了this上，也就意味着它是放到了创建出来的新对象中：
 - 在前面我们说过对于实例的方法，我们是希望放到原型上的，这样可以被多个实例来共享；
 - 这个时候我们可以直接在类中定义；

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


类的访问器方法

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

```
class Person {  
  constructor(name) {  
    this._name = name  
  }  
  
  set name(newName) {  
    console.log("调用了name的setter方法")  
    this._name = newName  
  }  
  
  get name() {  
    console.log("调用了name的getter方法")  
    return this._name  
  }  
}
```

类的静态方法

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

```
class Person {  
    constructor(age) {  
        this.age = age  
    }  
  
    static create() {  
        return new Person(Math.floor(Math.random() * 100))  
    }  
}
```

ES6类的继承 - extends

- 前面我们花了很大的篇幅讨论了在ES5中实现继承的方案，虽然最终实现了相对满意的继承机制，但是过程却依然是非常繁琐的。
- 在ES6中新增了使用**extends**关键字，可以方便的帮助我们实现继承：

```
class Person {  
  
}  
  
class Student extends Person {  
  .  
  .  
}
```

super关键字

■ Class为我们的方法中还提供了super关键字：

- 执行 `super.method(...)` 来调用一个父类方法。
- 执行 `super(...)` 来调用一个父类 constructor（只能在我们的 constructor 中）

■ 注意：在子（派生）类的构造函数中使用this或者返回默认对象之前，必须先通过super调用父类的构造函数！

■ super的使用位置有三个：子类的构造函数、实例方法、静态方法；

```
// 调用父对象/父类的构造函数
super([arguments]);

// 调用父对象/父类上的方法
super.functionOnParent([arguments]);
```

继承内置类

- 我们也可以让我们的类继承自内置类，比如Array：

```
class HYArray extends Array {  
  · lastItem() {  
    · · return this[this.length-1]  
  · }  
}  
  
var array = new HYArray(10, 20, 30)  
console.log(array.lastItem())
```

类的混入mixin

■ JavaScript的类只支持单继承：也就是只能有一个父类

- 那么在开发中我们我们需要在一个类中添加更多相似的功能时，应该如何来做呢？
- 这个时候我们可以使用混入（mixin）；

```
function mixinRunner(BaseClass) {  
  return class extends BaseClass {  
    running() {  
      console.log("running~")  
    }  
  }  
}  
  
function mixinEater(BaseClass) {  
  return class extends BaseClass {  
    eating() {  
      console.log("eating~")  
    }  
  }  
}
```

```
class Person {  
  
}  
  
class NewPerson extends mixinEater(mixinRunner(Person)) {  
  ...  
}  
  
var np = new NewPerson()  
np.eating()  
np.running()
```

在react中的高阶组件

Users > coderwhy > Desktop > React > 课堂 > code > 10_react-redux > src > utils > JS connect.js > ...

```
1  import React, { PureComponent } from "react";
2  import { StoreContext } from '../context';
3
4  export function connect(mapStateToProps, mapDispatchToProps) {
5    return function enhanceHOC(WrappedComponent) {
6      class EnhanceComponent extends PureComponent {
7        constructor(props, context) {
8          super(props, context);
9          this.state = {
10            storeState: mapStateToProps(context.getState())
11          }
12        }
13        componentDidMount() {
14          this.unsubscribe = this.context.subscribe(() => {
15            this.setState({
16              storeState: mapStateToProps(this.context.getState())
17            })
18          })
19        }
20        componentWillUnmount() {
21          this.unsubscribe();
22        }
23        render() {
24          return <WrappedComponent {...this.props}
25            {...mapStateToProps(this.context.getState())}
26            {...mapDispatchToProps(this.context.dispatch)} />
27        }
28      }
```

babel可以将ES6转成ES5代码

babeljs.io/repl#?browsers=defaults&build=&builtins=false&corejs=false&spec=false&loose=false&code_lz=MYGwhgzAeAKCmAnCB7AdtA3gKGtY6EALogK7BEqIAUaYAtvADTRgDm8AlFmEQAsAlhAB0dRtAC80CfF55BI0e3jTWHXgF9svM...

BABEL Docs Setup Try it out Videos Blog Search Donate Team GitHub

SETTINGS

- ☐ Evaluate
- ☒ Line Wrap
- ☒ Prettify
- ☐ File Size
- ☐ Time Travel

Source Type: Module

TARGETS

defaults

PRESETS

- ☒ react
- ☐ flow
- ☐ typescript
- ☐ stage-3
- ☒ stage-2
- ☐ stage-1
- ☐ stage-0

OPTIONS

React Runtime: Classic

Decorators version: 2021-12

Decorators before export: ☐

ENV PRESET

- ☒ Enabled

ELECTRON 1.8 ☐

NODE 10.13 ☐

BUILT-INS core-js 2 Entry ☐

SPEC ☐

```
1 class Person {
2   constructor(name, age) {
3     this.name = name
4     this.age = age
5   }
6
7   running() {
8     console.log("running")
9   }
10 }
11
12 class Student extends Person {
13   constructor(name, age, sno, score) {
14     super(name, age)
15     this.sno = sno
16     this.score = score
17   }
18 }
19
20 var stu = new Student()
21
22 var p = new Person()
```

```
1 "use strict";
2
3 function _typeof(obj) {
4   "@babel/helpers - typeof";
5   return (
6     _typeof =
7       "function" == typeof Symbol && "symbol" == typeof Symbol.iterator
8         ? function (obj) {
9             return typeof obj;
10           }
11         : function (obj) {
12             return obj &&
13               "function" == typeof Symbol &&
14                 obj.constructor == Symbol &&
15                 obj !== Symbol.prototype
16                 ? "symbol"
17                 : typeof obj;
18           }),
19     _typeof(obj)
20   );
21 }
22
23 function _inherits(subClass, superClass) {
24   if (typeof superClass !== "function" && superClass !== null) {
25     throw new TypeError("Super expression must either be null or a function");
26   }
27   subClass.prototype = Object.create(superClass && superClass.prototype, {
28     constructor: { value: subClass, writable: true, configurable: true }
29   });
30   Object.defineProperty(subClass, "prototype", { writable: false });
31   if (superClass) _setPrototypeOf(subClass, superClass);
32 }
33
34 function _setPrototypeOf(o, p) {
35   _setPrototypeOf = Object.setPrototypeOf ||
36     Object.setPrototypeOf.bind()
37     ; function _setPrototypeOf(o, p) {
38     o.__proto__ = p;
39     return o;
40   };
41   return _setPrototypeOf(o, p);
42 }
43
44 function _createSuper(Derived) {
45   var hasNativeReflectConstruct = isNativeReflectConstruct();
```


JavaScript中的多态

■ 面向对象的三大特性：封装、继承、多态。

□ 前面两个我们都已经详细解析过了，接下来我们讨论一下JavaScript的多态。

■ JavaScript有多态吗？

□ 维基百科对多态的定义：**多态**（英语：polymorphism）指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。

□ 非常的抽象，个人的总结：不同的数据类型进行同一个操作，表现出不同的行为，就是多态的体现。

■ 那么从上面的定义来看，JavaScript是一定存在多态的。

```
function sum(a, b) {  
  console.log(a + b)  
}  
  
sum(10, 20)  
sum("abc", "cba")
```

字面量的增强

■ ES6中对 **对象字面量** 进行了增强，称之为 Enhanced object literals（增强对象字面量）。

■ 字面量的增强主要包括下面几部分：

- 属性的简写：**Property Shorthand**

- 方法的简写：**Method Shorthand**

- 计算属性名：**Computed Property Names**

解构Destructuring

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

- 解构赋值 是一种特殊的语法，它使我们可以将数组或对象“拆包”至一系列变量中。

- 我们可以划分为：数组的解构和对象的解构。

- 数组的解构：

- 基本解构过程
 - 顺序解构
 - 解构出数组：...语法
 - 默认值

```
var [name1 = "aaa", name2, ...newNames] = names
```

- 对象的解构：

- 基本解构过程
 - 任意顺序
 - 重命名
 - 默认值

```
var {name : varName = "why", ...rest} = object
```

解构的应用场景

■ 解构目前在开发中使用是非常多的：

- 比如在开发中拿到一个变量时，自动对其进行解构使用；
- 比如对函数的参数进行解构；

```
async getPageListDataAction({ commit }, payload: IPagePayload) {  
  const pageName = payload.pageName  
  const pageUrl = `/${pageName}/list`  
  if (pageUrl.length === 0) return  
  const { totalCount, list } = await getPageList(pageUrl, payload.queryInfo)
```

```
instance.update = effect(function componentEffect() {  
  // 组件没有被挂载，那么挂载组件  
  if (!instance.isMounted) {  
    let vnodeHook: VNodeHook | null | undefined  
    const { el, props } = initialVNode  
    const { bm, m, parent } = instance
```