对象、类与面向对象

创建对象

使用Object构造函数或对象字面量可以方便地创建对象，但是有明显不足：创建具有同样接口地多个对象，需要重复编写很多代码

概述

ES5.1并没有正式支持面向对象的解构，比如类或继承
但是可以运用原型式继承可以模拟相同的行为
ES6正式支持类和继承。ES6的类旨在完全涵盖之前规范设计的基于原型的继承模式
ES6的类本质上是ES5.1构造函数加原型继承的语法糖

工厂模式

设计模式

```
function createPerson(name, age, job) {
 let o = new Object();
 o.name = name;
 o.age = age;
 o.job = job;
 o.sayName = function() {
 console.log(this.name);
 };
 return o;
}
let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");
```

解决了创建多个类似对象重复的问题，但是不能解决对象标示问题（TS解决）

构造函数模式

```
function Person(name, age, job){
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = function() {
 console.log(this.name);
 };
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");
person1.sayName(); // Nicholas
person2.sayName(); // Greg
```

```
//constructor指向构造函数Person
console.log(person1.constructor == Person); // true
console.log(person2.constructor == Person); // true

// instanceof 来判断类型
console.log(person1 instanceof Object); // true
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Object); // true
console.log(person2 instanceof Person); // true
```

与工厂模式的区别：

- 没有显式创建对象
- 属性和方法直接给了this
- 没有return

构造函数首字母大写（习惯）

new操作符干了什么？

- 内存中创建一个新对象
- 新对象的 [[Prototype]] 特性赋值为构造函数的prototype
- 构造函数内部this指向新对象
- 执行构造函数内部代码（给新对象添加属性）
- 构造函数返回非空对象，则返回对象；否则返回新对象

instanceof操作符
variable instanceof constructor
所有自定义对象都继承与Object 所以instanceof Object也是true

构造函数不一定要写成函数声明的形式

```
let Person = function(name, age, job) {
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = function() {
 console.log(this.name);
 };
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");
person1.sayName(); // Nicholas
person2.sayName(); // Greg
console.log(person1 instanceof Object); // true
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Object); // true
console.log(person2 instanceof Person); // true
```

不传参可以不写括号

```
function Person() {
 this.name = "Jake";
 this.sayName = function() {
 console.log(this.name);
 };
}
let person1 = new Person();
let person2 = new Person;
```

构造函数也是函数

构造函数与函数唯一区别 调用方式不同

```
function Person() {
 this.name = "Jake";
 this.sayName = function() {
 console.log(this.name);
 };
}

// 作为构造函数
let person = new Person("Nicholas", 29, "Software Engineer");
person.sayName(); // "Nicholas"
// 作为函数调用
Person("Greg", 27, "Doctor"); // 添加到 window 对象
window.sayName(); // "Greg"
// 在另一个对象的作用域中调用
let o = new Object();
Person.call(o, "Kristen", 25, "Nurse");
o.sayName(); // "Kristen"
```

构造函数的问题

构造函数定义的方法会在每个实例上都创建一遍
使用构造函数定义的方法

```
function Person(name, age, job){
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = new Function("console.log(this.name)"); // 逻辑等价
}
```

每个实例的方法都是Function的不同实例!

```
console.log(person1.sayName == person2.sayName); // false
```

解决方案
把函数定义转移到构造函数外

```
function Person(name, age, job){
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = sayName;
}
function sayName() {
 console.log(this.name);
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");
person1.sayName(); // Nicholas
person2.sayName(); // Greg
```

原型模式

如果使用上述方法解决相同逻辑函数重复定义问题会产生新问题
如果一个对象有很多方法，全局作用域就有很多方法
解决方案：原型模式

prototype属性

- 每个函数都会创建prototype属性
- 是一个对象
- 包含应该由特定引用类型的实例共享的属性和方法
- 通过调用构造函数创建的对象的原型

```
function Person() {}
Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
 console.log(this.name);
};
let person1 = new Person();
person1.sayName(); // "Nicholas"
let person2 = new Person();
person2.sayName(); // "Nicholas"
console.log(person1.sayName == person2.sayName); // true
```

理解原型

创建一个函数，就会按照特定规则为这个函数创建一个prototype属性（指向原型对象）
默认情况下，所有原型对象，自动获得constructor属性

自定义构造函数时，原型对象只会获得constructor属性，其他所有的方法都继承Object

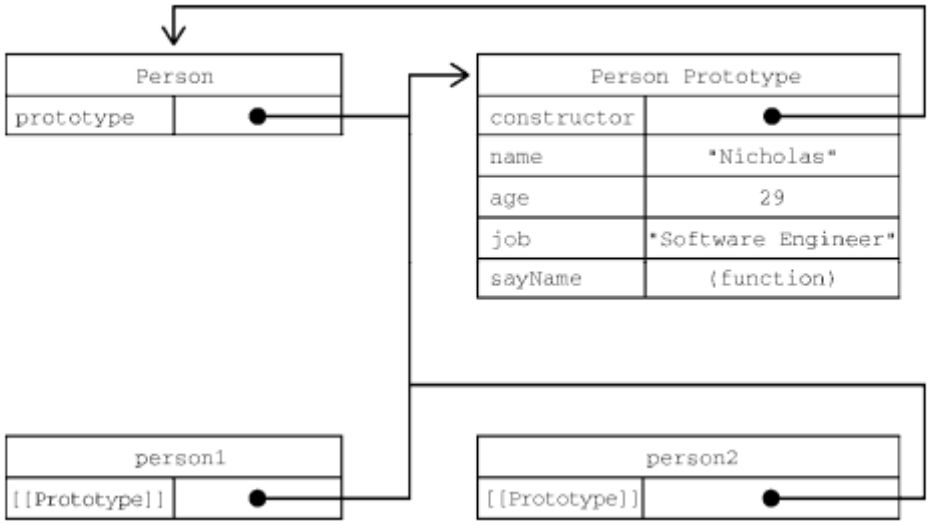
每调用构造函数创建一个实例，实例内部[[prototype]]指针就会被赋值为构造函数的原型对象

js没有访问[[prototype]]的特定方式，Firefox、Safari、Chrome会在每个对象上暴露__proto__属性

实例与构造函数原型由直接联系
实例与构造函数之间没有直接联系

```
/**
 * 构造函数可以是函数表达式
 * 也可以是函数声明，因此以下两种形式都可以：
 * function Person() {}
 * let Person = function() {}
 */
function Person() {}
/**
 * 声明之后，构造函数就有了一个
 * 与之关联的原型对象：
 */
console.log(typeof Person.prototype);
console.log(Person.prototype);
// {
// constructor: f Person(),
// __proto__: Object
// }
/**
 * 如前所述，构造函数有一个 prototype 属性
 * 引用其原型对象，而这个原型对象也有一个
 * constructor 属性，引用这个构造函数
 * 换句话说，两者循环引用：
 */
console.log(Person.prototype.constructor === Person); // true
/**
 * 正常的原型链都会终止于 Object 的原型对象
 * Object 原型的原型是 null
 */
console.log(Person.prototype.__proto__ === Object.prototype); // true
console.log(Person.prototype.__proto__.constructor === Object); // true
console.log(Person.prototype.__proto__.__proto__ === null); // true
console.log(Person.prototype.__proto__);
// {
// constructor: f Object(),
// toString: ...
// hasOwnProperty: ...
// isPrototypeOf: ...
// ...
// }
let person1 = new Person(),
 person2 = new Person();
/**
 * 构造函数、原型对象和实例
 * 是 3 个完全不同的对象：
 */
console.log(person1 !== Person); // true
console.log(person1 !== Person.prototype); // true
console.log(Person.prototype !== Person); // true
/**
 * 实例通过__proto__链接到原型对象，
 * 它实际上指向隐藏特性[[Prototype]]
 *
 * 构造函数通过 prototype 属性链接到原型对象
 *
 * 实例与构造函数没有直接联系，与原型对象有直接联系
 */
console.log(person1.__proto__ === Person.prototype); // true
```

```
conosle.log(person1.__proto__.constructor === Person); // true
/**
 * 同一个构造函数创建的两个实例
 * 共享同一个原型对象：
 */
console.log(person1.__proto__ === person2.__proto__); // true
/**
 * instanceof 检查实例的原型链中
  * 是否包含指定构造函数的原型：
 */
console.log(person1 instanceof Person); // true
console.log(person1 instanceof Object); // true
console.log(Person.prototype instanceof Object); // true
```



总结一下

- 每个函数都有prototype属性，指向函数的原型对象
- 每个对象有__proto__属性，指向对象的原型
- 正常的原型链终止于Object的原型对象
- Object原型的原型是null

```
function a() {

}
console.log(a.prototype.__proto__ == Object.prototype) // true
console.log(Object.prototype.__proto__)// null
```

ES有Object.getPrototypeOf(),返回参数内部[[prototype]]值

```
function a(){
}
console.log(a.prototype)
console.log(Object.getPrototypeOf(new a())==a.prototype) // true
```

Object还有setPrototypeOf()方法，可以向实例的私有特性[[prototype]]写入新值
很消耗性能，可以Object.create(),并指定原型

```
let biped = {
 numLegs: 2
};
let person = {
 name: 'Matt'
};
Object.setPrototypeOf(person, biped);
console.log(person.name); // Matt
console.log(person.numLegs); // 2
console.log(Object.getPrototypeOf(person) === biped); // true

// Object.create()
let person = Object.create(biped);
person.name = 'Matt';
console.log(person.name); // Matt
console.log(person.numLegs); // 2
console.log(Object.getPrototypeOf(person) === biped); // true
```

原型层级

访问属性时，先访问实例，如果实例没有则查找原型
constructor属性在原型上，通过实例也可以访问（查找原型获得）
不能通过实例重写原型的值
实例添加一个与原型对象中同名的属性，实例的属性会遮蔽原型对象的属性，只有delete才能解除这种遮蔽

Object.hasOwnProperty()就是用来判断是实例属性还是原型属性

原型和in操作符

in操作符两种使用方式:

- 单独使用
  可以通过对象访问指定属性时返回true (会查原型)

```
function Person() {}
Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
 console.log(this.name);
};
let person1 = new Person();
let person2 = new Person();
console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true
person1.name = "Greg";
console.log(person1.name); // "Greg"，来自实例
console.log(person1.hasOwnProperty("name")); // true
console.log("name" in person1); // true
console.log(person2.name); // "Nicholas"，来自原型
console.log(person2.hasOwnProperty("name")); // false
console.log("name" in person2); // true
```

```
delete person1.name;
console.log(person1.name); // "Nicholas"，来自原型
console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true
```

判断某个属性是否是原型属性
即不在实例上，且能访问到，使用hasOwnProperty()和in操作符

- for-in循环中使用
  返回可枚举的属性，包括实例属性和原型属性
  Object.keys()方法，对象为参数，返回所有可枚举的属性名称的字符串数组
  Object.getOwnPropertyNames()，获得对象的所有属性(不查原型)
  ES6新增符号后，需要获取符号属性：Object.getOwnPropertySymbols()

```
let k1 = Symbol('k1'),
 k2 = Symbol('k2');
 let o = {
 [k1]: 'k1',
 [k2]: 'k2'
};
console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

属性枚举顺寻

for-in 循环和 Object.keys()的枚举顺序是不确定的,取决于js引擎

Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()和 Object.assign()的枚举顺序是确定性的。先以升序枚举数值键，然后以插入顺序枚举字符串和符号键

对象迭代

ES2017新增静态方法，将对象内容转换为无序的可迭代的格式
Object.values()和 Object.entries()，接收一个对象，返回它们内容的数组
Object.values()返回对象值的数组，Object.entries()返回键/值对的数组。

```
const o = {
 foo: 'bar',
 baz: 1,
 qux: {}
};
console.log(Object.values(o));
// ["bar", 1, {}]
console.log(Object.entries((o)));
// [["foo", "bar"], ["baz", 1], ["qux", {}]]
```

非字符串属性会被转为字符串输出。
浅拷贝
符号属性会被忽略

其他原型语法

字面量赋值prototype,会修改constructor属性，指向Object

```
let friend = new Person();
console.log(friend instanceof Object); // true
console.log(friend instanceof Person); // true
console.log(friend.constructor == Person); // false
console.log(friend.constructor == Object); // true
```

解决方案，带上constructor属性

```
function Person() {
}
Person.prototype = {
 constructor: Person,
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",
 sayName() {
 console.log(this.name);
 }
};
```

但是这样constructor属性就是显式声明，[[Enumerable]]默认为true
解决方案 Object.defineProperty

```
function Person() {}
Person.prototype = {
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",
 sayName() {
 console.log(this.name);
 }
};
// 恢复 constructor 属性
Object.defineProperty(Person.prototype, "constructor", {
 enumerable: false, //也可以不写，默认就是false
 value: Person
});
```

原型的动态性

原型向上搜索过程是动态的，即使实例在修改原型之前存在，任何时候对原型对象所做的修改也会在实例上反映出来

```
let friend = new Person();
Person.prototype.sayHi = function() {
 console.log("hi");
```

```
};
friend.sayHi(); // "hi"，没问题！
```

这与重写原型不一样

```
function Person() {}
let friend = new Person();
Person.prototype = {
 constructor: Person,
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",
 sayName() {
 console.log(this.name);
 }
};
friend.sayName(); // 错误
```

调用构造函数生成实例自动赋值的prototype是原型的指针，如果重写prototype，则是一个新对象，地址变了，所以报错

原生对象原型

修改原生对象原型可以给原生类型定义新的方法

```
String.prototype.startsWith = function (text) {
 return this.indexOf(text) === 0;
};
let msg = "Hello world!";
console.log(msg.startsWith("Hello")); // true
```

不建议这么做，很容易造成命名冲突！

原型的问题

弱化了向构造函数传参的能力
共享特性，引用值

```
function Person() {}
Person.prototype = {
 constructor: Person,
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",
 friends: ["Shelby", "Court"],
  sayName() {
 console.log(this.name);
 }
};
let person1 = new Person();
let person2 = new Person();
person1.friends.push("Van");
```

```
console.log(person1.friends); // "Shelby,Court,Van"
console.log(person2.friends); // "Shelby,Court,Van"
console.log(person1.friends === person2.friends); // true
```