对象、类与面向对象编程

继承

接口继承:继承方法签名
实现继承:继承实际的方法
ES函数没有签名，只能实现继承。

原型链

ECMA-262 把原型链定义为 ECMAScript 的主要继承方式
基本思想：通过原型继承多个引用类型的属性和方法

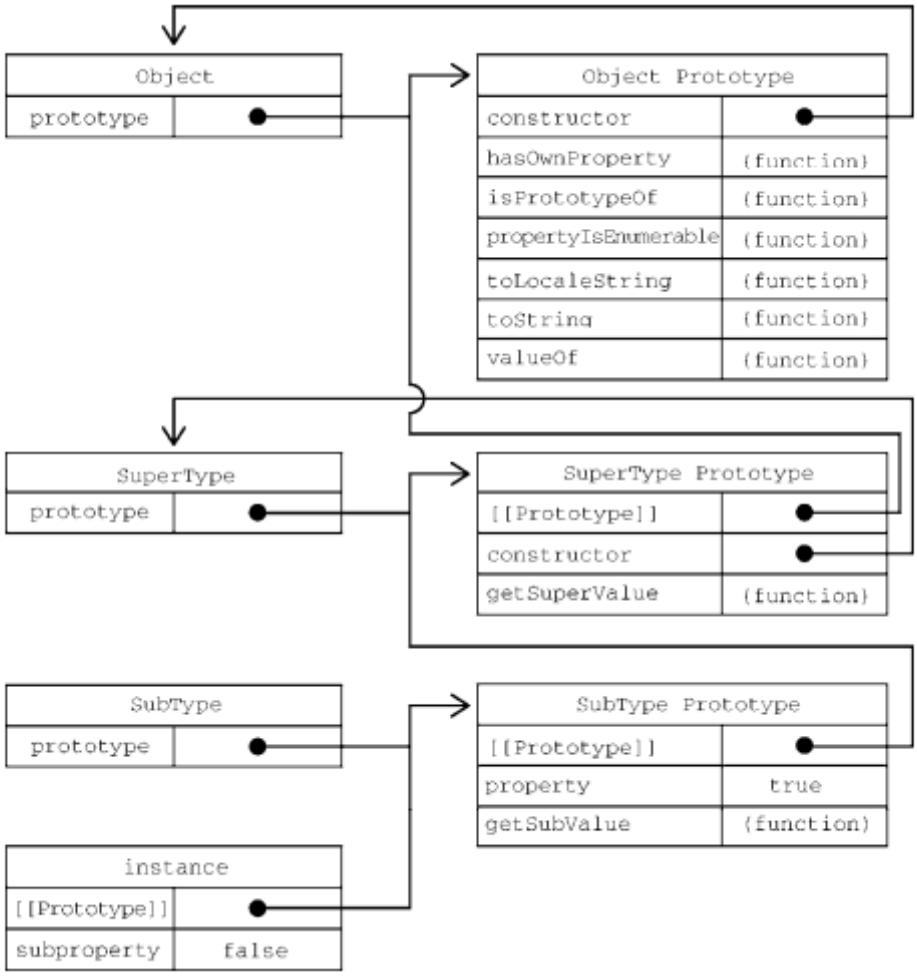
- 每个构造函数(函数)都有一个原型对象prototype
- prototype有constructor属性指向构造函数
- 实例有一个内部指针指向prototype
- 浏览器一般用__proto__暴露内部指针

原型链：原型是另一个构造函数的实例

```
function SuperType() {
 this.property = true;
}
SuperType.prototype.getSuperValue = function() {
 return this.property;
};
function SubType() {
 this.subproperty = false;
}
// 继承 SuperType
SubType.prototype = new SuperType();
SubType.prototype.getSubValue = function () {
 return this.subproperty;
};
let instance = new SubType();
console.log(instance.getSuperValue()); // true
```

默认原型

所有引用类型都继承自Object
即，最终会指向Object的Prototype



原型与继承关系

原型与实例的关系可以用两种防暑确定

- instanceof操作符

```
console.log(instance instanceof Object); // true
console.log(instance instanceof SuperType); // true
console.log(instance instanceof SubType); // true
```

- isPrototypeOf()
  prototypeObj.isPrototypeOf(object)
  在参数的原型链上搜寻prototypeObj

```
console.log(Object.prototype.isPrototypeOf(instance)); // true
console.log(SuperType.prototype.isPrototypeOf(instance)); // true
console.log(SubType.prototype.isPrototypeOf(instance)); // true
```

关于方法

子类有时候需要覆盖父类的方法，或者增加父类没有的方法。
这些方法必须在原型赋值之后再添加到原型上

```
function SuperType() {
 this.property = true;
}
SuperType.prototype.getSuperValue = function() {
 return this.property;
};
function SubType() {
 this.subproperty = false;
}
// 继承 SuperType
SubType.prototype = new SuperType();
// 新方法
SubType.prototype.getSubValue = function () {
 return this.subproperty;
};
// 覆盖已有的方法
SubType.prototype.getSuperValue = function () {
 return false;
};
let instance = new SubType();
console.log(instance.getSuperValue()); // false
```

对象字面量方式创建原型会破坏原型链，相当于重写了原型链

```
function SuperType() {
 this.property = true;
}
SuperType.prototype.getSuperValue = function() {
 return this.property;
};
function SubType() {
 this.subproperty = false;
}
// 继承 SuperType
SubType.prototype = new SuperType();
// 通过对象字面量添加新方法，这会导致上一行无效
SubType.prototype = {
 getSubValue() {
 return this.subproperty;
 },
 someOtherMethod() {
 return false;
 }
};
let instance = new SubType();
console.log(instance.getSuperValue()); // 出错!
```

原型链的问题

与原型的问题一样，主要问题是引用值的问题

盗用构造函数

盗用构造函数又称对象伪装，经典继承
解决引用值的问题
思路：子类构造函数中调用父类构造函数

```
function SuperType() {
 this.colors = ["red", "blue", "green"];
}
function SubType() {
 // 继承 SuperType
 SuperType.call(this);
}
let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"
let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green"
```

优点

解决引用值问题
可以传参

```
function SuperType(name){
 this.name = name;
}
function SubType() {
 // 继承 SuperType 并传参
 SuperType.call(this, "Nicholas");
 // 实例属性
 this.age = 29;
}
let instance = new SubType();
console.log(instance.name); // "Nicholas";
console.log(instance.age); // 29
```

缺点

与构造函数模式缺点一模一样，方法得写在构造函数内，无法复用
因为通过call而不是 new，所以子类不能继承父类的方法，如果要继承父类方法，父类的构造函数内必须定义方法

组合继承

又称伪经典继承
综合原型链，盗用构造函数
使用最多的继承模式

```
function SuperType(name){
 this.name = name;
 this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function() {
```

```
 console.log(this.name);
};
function SubType(name, age){
 // 继承属性
 SuperType.call(this, name);
 this.age = age;
}
// 继承方法
SubType.prototype = new SuperType();
SubType.prototype.sayAge = function() {
 console.log(this.age);
};
let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"
instance1.sayName(); // "Nicholas";
instance1.sayAge(); // 29
let instance2 = new SubType("Greg", 27);
console.log(instance2.colors); // "red,blue,green"
instance2.sayName(); // "Greg";
instance2.sayAge(); // 27
```

原型式继承

目标：不自定义类型也可以通过原型实现对象之间的信息共享

原型链： 原型是另一个构造函数的实例
原型式继承：有一个对象，想要在这个对象的基础上再创建一个新对象

适合不需要单独创建构造函数，但是仍然需要在对象间共享信息的场合。引用值始终会在相关对象间共享

ES5把原型式继承概念规范化
Object.create
Object.create(proto，[propertiesObject])
proto：新创建对象的原型对象
propertiesObject：可选，参照Object.defineProperties()的第二个参数

```
function object(o) {
 function F() {}
 F.prototype = o;
 return new F();
}


let person = {
 name: "Nicholas",
 friends: ["Shelby", "Court", "Van"]
};
let anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");
let yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");
console.log(person.friends); // "Shelby,Court,Van,Rob,Barbie"
```

```
let person = {
 name: "Nicholas",
 friends: ["Shelby", "Court", "Van"]
};
let anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");
let yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");
console.log(person.friends); // "Shelby,Court,Van,Rob,Barbie"


let person = {
 name: "Nicholas",
 friends: ["Shelby", "Court", "Van"]
};
let anotherPerson = Object.create(person, {
 name: {
 value: "Greg"
 }
});
console.log(anotherPerson.name); // "Greg"
```

寄生式继承

创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象

```
function createAnother(original){
 let clone = object(original); // 通过调用函数创建一个新对象
 clone.sayHi = function() { // 以某种方式增强这个对象
 console.log("hi");
 };
 return clone; // 返回这个对象
}

let person = {
 name: "Nicholas",
 friends: ["Shelby", "Court", "Van"]
};
let anotherPerson = createAnother(person);
anotherPerson.sayHi(); // "hi"
```

原有的基础上，只能加了sayHi属性

寄生式组合继承

最佳模式！！
解决组合式继承 调用2次 父类构造函数

组合式继承分成两步：继承父类属性，继承父类方法

继承父类方法：son.prototype = new Father()

寄生式组合继承：通过寄生的方式继承父类方法

```
function SuperType(name) {
 this.name = name;
 this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function() {
 console.log(this.name);
};
function SubType(name, age){
 SuperType.call(this, name); // 第二次调用 SuperType()
 this.age = age;
}
SubType.prototype = new SuperType(); // 第一次调用 SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function() {
 console.log(this.age);
};
```

通过寄生来实现

```
function inheritPrototype(subType, superType) {
    let prototype = object(superType.prototype); // 创建对象
    prototype.constructor = subType; // 增强对象
    subType.prototype = prototype; // 赋值对象
}
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function () {
    console.log(this.name);
};
function SubType(name, age) {
    SuperType.call(this, name);
    this.age = age;
}
inheritPrototype(SubType, SuperType);
SubType.prototype.sayAge = function () {
    console.log(this.age);
};
```