JavaScript中的继承

- 原型链继承
- 原型式继承
- 构造继承
- 组合继承(****推荐)
- 寄生继承
- 寄生组合式继承(*****推荐)

1.原型链继承

ECMAScript将原型链作为实现继承的主要方法。基本思想是，利用原型让一个引用类型继承另一个引用类型的属性和
构造函数、原型、对象实例之间的关系：
每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而对象实例都包含一个指向原型对象的内
实现的本质是重写原型对象，代之以一个新类型的实例。*将父类的实例作为子类的原型*。基本模式：

```
<script>

    function SuperType() {
        this.property = true;
    }
    SuperType.prototype.getSuperValue = function() {
        return this.property;
    }

    function SubType() {
        this.subproperty = false;
    }
    //SubType继承自SuperType
    SubType.prototype = new SuperType();

    SubType.prototype.getSubValue = function() {
        return this.subproperty;
    }

    var instance = new SubType();
    alert(instance.getSuperValue()); // true

</script>
```

解析：定义2个类型SuperType和SubType，每个类型分别包含一个属性和方法。SubType的继承是通过创建SuperTy
instance指向SubType的原型，SubType的原型指向SuperType的原型。
调用instance.getSuperValue()会有3个步骤：1.搜索实例；2.搜索SubType.prototype;
3.搜索SuperType.prototype,最后一步找到该方法；找不

特点：非常纯粹的继承关系，实例是子类的实例，也是父类的实例
父类新增原型方法/原型属性，子类都能访问到
简单，易于实现

问题:要想为子类原型新增属性和方法，必须要在**new SuperType()**这样的语句之后执行
创建子类实例时，无法向父类构造函数传参
来自原型对象的所有属性被所有实例共享
无法实现多继承

2.原型式继承

核心：原型式继承就是借用构造函数的原型对象实现继承，子.prototype = 父.prototype
场景：可以在不必预先定义构造函数的情况下实现继承

```
<script>

    var person ={
                name: "nicholas",
                friends: ["ann", "bill", "candy"]
    };
    var anotherperson = Object.create(person);
    anotherperson.name = "gero";
    anotherperson.friends.push("rob");

    var yetperson = Object.create(person);
    yetperson.name = "gero";
    yetperson.friends.push("kill");

    alert(person.friends); // "ann, bill, candy, rob, kill"

</script>
```

问题:
父构造函数的原型对象和子构造函数的原型对象上的成员有共享问题
只能继承父构造函数的原型对象上的成员，不能继承父构造函数的实例对象的成员

3.构造继承

核心：使用**call**和**apply**借用其他构造函数的成员，即使用父类的构造函数来增强子类实例，等于是复制父类的实例属

```
<script>

    function SuperType() {
        this.color = ["red", "green"];
    }
    function SubType() {
        //继承SuperType
        SuperType.call(this);
    }
    var instance1 = new SubType();
    instance1.color.push("black");
    alert(instance1.color); // "red, green, black"
```

```
    var instance2 = new SubType();
    alert(instance2.color); // "red, green"

</script>
```

```
<script>

    function SuperType(name) {
        this.name = name;
    }
    function SubType() {
        //继承SuperType, 同时传递参数
        SuperType.call(this, "nicholas");

        //实例属性
        this.age = 22;
    }
    var instance = new SubType();
    alert(instance.name); // "nicholas"
    alert(instance.age); // 22

</script>
```

特点:

解决了原型链继承，子类实例共享父类引用属性的问题
创建子类实例时，可以向父类传递参数
可以实现多继承（apply多个父类对象）

问题: 实例并不是父类的实例，只是子类的实例
只能继承父类的实例属性和方法，不能继承原型属性/方法
无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

4.组合继承

核心: 原型链继承 + 构造函数，使用原型链实现对原型属性和方法的继承，使用构造函数来实现对实例属性的继承。

```
<script>

    function SuperType(name) {
        this.name = name;
        this.color = ["red", "green"];
    }
    SuperType.prototype.sayName = function() {alert(this.name);}

    function SubType(name, age) {
        //继承SuperType的属性,第二次调用父级SuperType()
        SuperType.call(this, name);
        this.age = age;
```

```
    }

    //继承SuperType的方法
    SubType.prototype = new SuperType(); // 第一次调用父级SuperType()
    SubType.prototype.constructor = SubType; //增强对象
    SubType.prototype.sayAge = function() {alert(this.age);}

    var instance1 = new SubType("nicholas", 22);
    instance1.color.push("black");
    alert(instance1.color); // "red, green, black"
    alert(instance1.name); // "nicholas"
    alert(instance1.age); // 22

    var instance2 = new SubType("gero", 28);
    alert(instance2.color); // "red, green, black"
    alert(instance2.name); // "gero"
    alert(instance2.age); // 28

</script>
```

特点：不存在引用属性共享问题
弥补了原型链继承和构造函数继承的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法
子类的实例，也是父类的实例
可传参 函数可复用
问题： 调用了两次父类构造函数，生成了两份实例(多消耗了一点内存)

5.寄生继承

核心：创建一个仅用于封装继承过程的函数，在函数内部以某种方式来增强对象，然后返回对象
场景：在主要考虑对象而不是自定义类型和构造函数的情况下

```
function createAnother(original) {
    var clone = object(original); //通过调用函数来创建函数
    clone.sayHi = function() {  //以某种方式来增强对象
        alert("hi");
    };
    return clone; //返回对象
}
```

解析：createAnother()函数接收了一个original参数，作为新对象的基础；把这个original对象传递给object(
将返回结果赋值给clone。然后为clone添加一个新方法sayHi()，最后返回clone对象。使用方式如下：

```
<script>

var person = {
    name: "nicholas",
    friends: ["ann", "bill"]
};
var anotherPerson = createAnother(person);
anotherPerson.sayHi(); // "hi"

</script>
```

解析： 基于person返回了一个新对象----anotherPerson，新对象不仅拥有person的所有属性和方法，还有自己的

6.寄生组合式继承

核心：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，
场景： 实现基于类型继承的最有效的方式
基本模式：

```
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
}
<script>

    function SuperType(name) {
        this.name = name;
        this.color = ["red", "green"];
    }
    SuperType.prototype.sayName = function() {alert(this.name);}

    function SubType(name, age) {
        //继承SuperType的属性
        SuperType.call(this, name);
        this.age = age;
    }
    inheritPrototype(SubType, SuperType);
    SubType.prototype.sayAge = function() {alert(this.age);}

</script>
```