高级Day 02---理解原型

一、什么是原型

> 原型是Javascript中的继承的继承，JavaScript的继承就是基于原型的继承。

1.1 函数的原型对象

在JavaScript中，我们创建一个函数A(就是声明一个函数), 那么浏览器就会在内存中创建一个对象B，而且每个函数都默认会有一个属性 prototype 指向了这个对象(即：prototype的属性的值是这个对象)。这个对象B就是函数A的原型对象，简称函数的原型。这个原型对象B 默认会有一个属性 constructor 指向了这个函数A (意思就是说：constructor属性的值是函数A)。

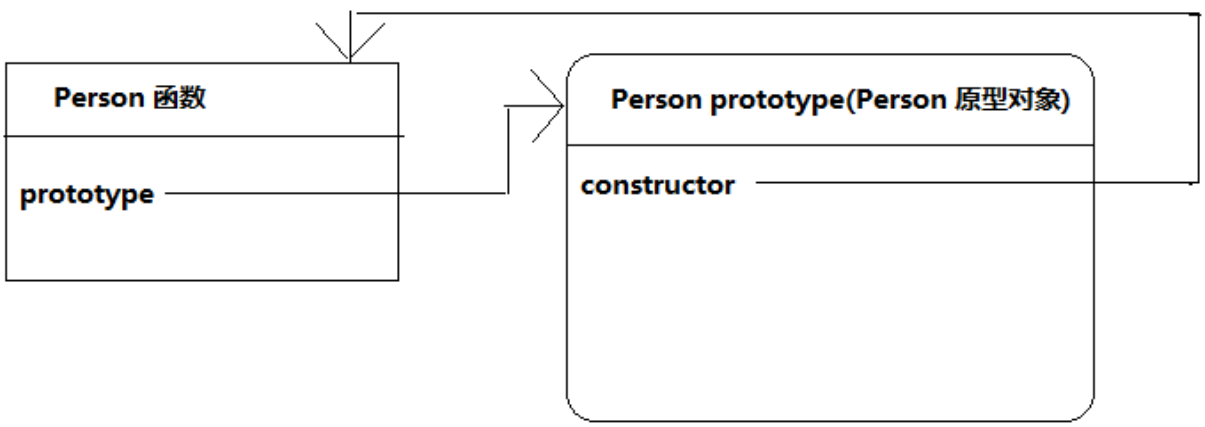
看下面的代码：

```
<body>
    <script type="text/javascript">
        /*
            声明一个函数，则这个函数默认会有一个属性叫 prototype 。而且浏览器会自动按照一定的规则
            创建一个对象，这个对象就是这个函数的原型对象，prototype属性指向这个原型对象。这个原型对象
            有一个属性叫constructor 执行了这个函数

            注意：原型对象默认只有属性：constructor。其他都是从Object继承而来，暂且不用考虑。
        */
        function Person () {

        }
    </script>
</body>
```

> 下面的图描述了声明一个函数之后发生的事情：



1.2 使用构造函数创建对象

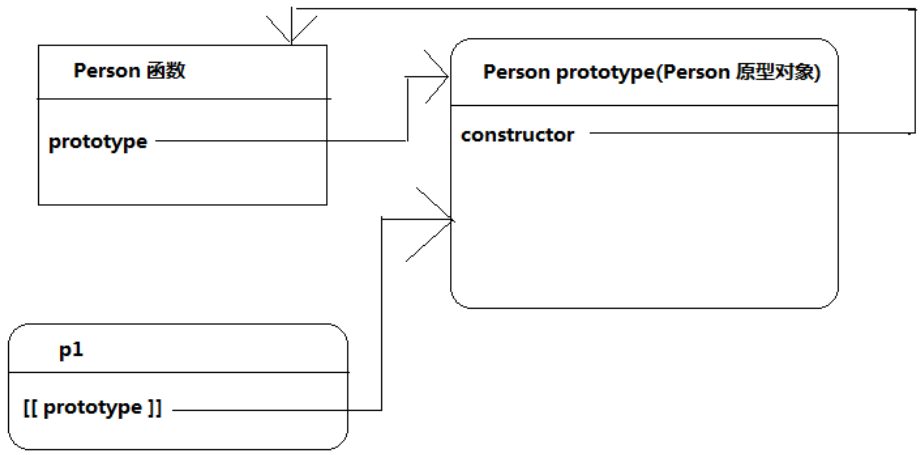
当把一个函数作为构造函数 (理论上任何函数都可以作为构造函数) 使用new创建对象的时候，那么这个对象就会存在一个默认的不可见的属性，来指向了构造函数的原型对象。 这个不可见的属性我们一般用 [[prototype]] 来表示，只是这个属性没有办法直接访问到。

看下面的代码：

```
<body>
    <script type="text/javascript">
        function Person () {

        }
        /*
            利用构造函数创建一个对象，则这个对象会自动添加一个不可见的属性 [[prototype]]，而且这个属性指向了构造函数的原型对象。
        */
        var p1 = new Person();
    </script>
</body>
```

> 观察下面的示意图：



> 说明：

1. 从上面的图示中可以看到，创建p1对象虽然使用的是Person构造函数，但是对象创建出来之后，这个p1对象其实已经与Person构造函数没有任何关系了，p1对象的[[prototype]]属性指向的是Person构造函数的原型对象。
2. 如果使用new Person()创建多个对象，则多个对象都会同时指向Person构造函数的原型对象。
3. 我们可以手动给这个原型对象添加属性和方法，那么p1,p2,p3...这些对象就会共享这些在原型中添加的属性和方法。
4. 如果我们访问p1中的一个属性name，如果在p1对象中找到，则直接返回。如果p1对象中没有找到，则直接去p1对象的[[prototype]]属性指向的原型对象中查找，如果查找到则返回。(如果原型中也没有找到，则继续向上找原型的原型---原型链。 后面再讲)。
5. 如果通过p1对象添加了一个属性name，则p1对象来说就屏蔽了原型中的属性name。 换句话说：在p1中就没有办法访问到原型的属性name了。
6. 通过p1对象只能读取原型中的属性name的值，而不能修改原型中的属性name的值。 p1.name = "李四"; 并

不是修改了原型中的值，而是在p1对象中给添加了一个属性name。

> 看下面的代码：

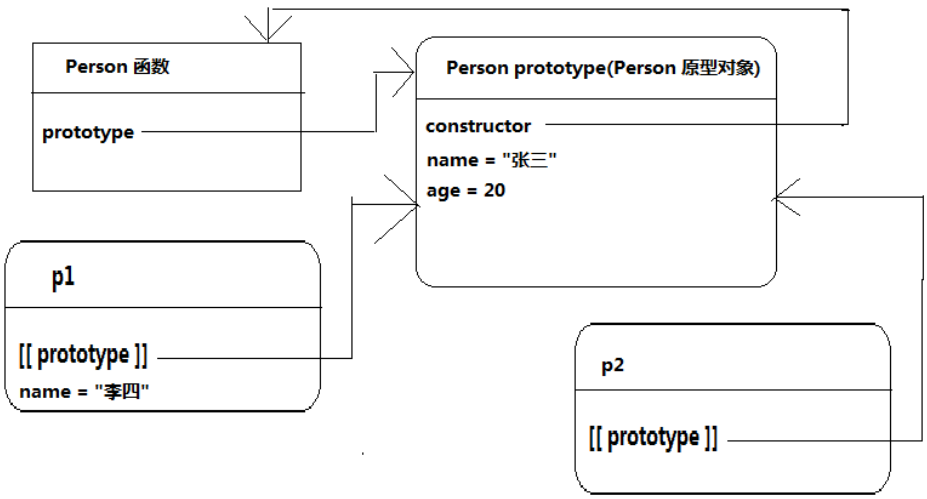
```
<body>
    <script type="text/javascript">
        function Person () {
        }
        // 可以使用Person.prototype 直接访问到原型对象
        //给Person函数的原型对象中添加一个属性 name并且值是 "张三"
        Person.prototype.name = "张三";
        Person.prototype.age = 20;

        var p1 = new Person();
        /*
            访问p1对象的属性name，虽然在p1对象中我们并没有明确的添加属性name，但是
            p1的 [[prototype]] 属性指向的原型中有name属性，所以这个地方可以访问到属性name
            就值。
            注意：这个时候不能通过p1对象删除name属性，因为只能删除在p1中删除的对象。
        */
        alert(p1.name);  // 张三

        var p2 = new Person();
        alert(p2.name);  // 张三  都是从原型中找到的，所以一样。

        alert(p1.name === p2.name);  // true

        // 由于不能修改原型中的值，则这种方法就直接在p1中添加了一个新的属性name，然后在p1中无法再访问到
        //原型中的属性。
        p1.name = "李四";
        alert("p1: " + p1.name);
        // 由于p2中没有name属性，则对p2来说仍然是访问的原型中的属性。
        alert("p2:" + p2.name);  // 张三
    </script>
</body>
```



二、与原型有关的几个属性和方法

2.1 prototype属性

prototype 存在于构造函数中 (其实任意函数中都有，只是不是构造函数的时候prototype我们不关注而已)，他指向了这个构造函数的原型对象。

参考前面的示意图。

2.2 constructor属性

constructor属性存在于原型对象中，他指向了构造函数

看下面的代码：

```
<script type="text/javascript">
    function Person () {
    }
    alert(Person.prototype.constructor === Person); // true
    var p1 = new Person();
    //使用instanceof 操作符可以判断一个对象的类型。
    //typeof一般用来获取简单类型和函数。而引用类型一般使用instanceof，因为引用类型用typeof 总是返回objece。
    alert(p1 instanceof Person);    // true
</script>
```

> 我们根据需要，可以Person.prototype 属性指定新的对象，来作为Person的原型对象。
>
> 但是这个时候有个问题，新的对象的constructor属性则不再指向Person构造函数了。

看下面的代码：

```
<script type="text/javascript">
    function Person () {

    }
    //直接给Person的原型指定对象字面量。则这个对象的constructor属性不在执行Person函数
    Person.prototype = {
        name:"志玲",
        age:20
    };
    var p1 = new Person();
    alert(p1.name);  // 志玲

    alert(p1 instanceof Person); // true
    alert(Person.prototype.constructor === Person); //false
    //如果constructor对你很重要，你应该在Person.prototype中添加一行这样的代码：
    /*
    Person.prototype = {
        constructor : Person    //让constructor重新指向Person函数
    }
    */
</script>
```

2.3 __proto__ 属性(注意：左右各是2个下划线)

用构造方法创建一个新的对象之后，这个对象中默认会有一个不可访问的属性 [[prototype]] , 这个属性就指向了构造方法的原型对象。

但是在个别浏览器中，也提供了对这个属性[[prototype]]的访问(chrome浏览器和火狐浏览器。ie浏览器不支持)。访问方式：p1.__proto__

但是开发者尽量不要用这种方式去访问，因为操作不慎会改变这个对象的继承原型链。

```
<script type="text/javascript">
    function Person () {

    }
    //直接给Person的原型指定对象字面量。则这个对象的constructor属性不在执行Person函数
    Person.prototype = {
        constructor : Person,
        name:"志玲",
        age:20
    };
    var p1 = new Person();

    alert(p1.__proto__ === Person.prototype);    //true

</script>
```

2.4 hasOwnProperty() 方法

大家知道，我们用去访问一个对象的属性的时候，这个属性既有可能来自对象本身，也有可能来自这个对象的[[prototype]]属性指向的原型。

那么如何判断这个对象的来源呢？

hasOwnProperty方法，可以判断一个属性是否来自对象本身。

```
<script type="text/javascript">
    function Person () {

    }
    Person.prototype.name = "志玲";
    var p1 = new Person();
    p1.sex = "女";
    //sex属性是直接在p1属性中添加，所以是true
    alert("sex属性是对象本身的：" + p1.hasOwnProperty("sex"));
    // name属性是在原型中添加的，所以是false
    alert("name属性是对象本身的：" + p1.hasOwnProperty("name"));
    //  age 属性不存在，所以也是false
    alert("age属性是存在于对象本身：" + p1.hasOwnProperty("age"));

</script>
```

> 所以，通过hasOwnProperty这个方法可以判断一个对象是否在对象本身添加的，但是不能判断是否存在于原型中，因为有可能这个属性不存在。
>
> 也即是说，在原型中的属性和不存在的属性都会返回fasle。
>
> 如何判断一个属性是否存在于原型中呢？

2.5 in 操作符

in操作符用来判断一个属性是否存在于这个对象中。但是在查找这个属性时候，现在对象本身中找，如果对象找不到再去原型中找。换句话说，只要对象和原型中有一个地方存在这个属性，就返回true

```
<script type="text/javascript">
    function Person () {

    }
    Person.prototype.name = "志玲";
    var p1 = new Person();
    p1.sex = "女";
    alert("sex" in p1);      // 对象本身添加的，所以true
    alert("name" in p1);     //原型中存在，所以true
    alert("age" in p1);      //对象和原型中都不存在，所以false

</script>
```

> 回到前面的问题，如果判断一个属性是否存在于原型中：
>
> 如果一个属性存在，但是没有在对象本身中，则一定存在于原型中。

```
<script type="text/javascript">
    function Person () {
    }
    Person.prototype.name = "志玲";
    var p1 = new Person();
    p1.sex = "女";

    //定义一个函数去判断原型所在的位置
    function propertyLocation(obj, prop){
        if(!(prop in obj)){
            alert(prop + "属性不存在");
        }else if(obj.hasOwnProperty(prop)){
            alert(prop + "属性存在于对象中");
        }else {
            alert(prop + "对象存在于原型中");
        }
    }
    propertyLocation(p1, "age");
    propertyLocation(p1, "name");
    propertyLocation(p1, "sex");
</script
```

三、组合原型模型和构造函数模型创建对象

3.1 原型模型创建对象的缺陷

原型中的所有的属性都是共享的。也就是说，用同一个构造函数创建的对象去访问原型中的属性的时候，大家都是访问的同一个对象，如果一个对象对原型的属性进行了修改，则会反映到说有的对象上面。

但是在实际使用中，每个对象的属性一般是不同的。张三的姓名是张三，李四的姓名是李四。

但是，这个共享特性对方法(属性值是函数的属性)又是非常合适的。所有的对象共享方法是最佳状态。这种特性在c#和Java中是天生存在的。

3.2 构造函数模型创建对象的缺陷

在构造函数中添加的属性和方法，每个对象都有自己独有的一份，大家不会共享。这个特性对属性比较合适，但是对方法又不太合适。因为对所有对象来说，他们的方法应该是一份就够了，没有必要每人一份，造成内存的浪费和性能的低下。

```
<script type="text/javascript">
    function Person() {
        this.name = "李四";
        this.age = 20;
        this.eat = function() {
            alert("吃完东西");
        }
    }
    var p1 = new Person();
    var p2 = new Person();
    //每个对象都会有不同的方法
    alert(p1.eat === p2.eat); //fasle
</script>
```

> 可以使用下面的方法解决：

```
<script type="text/javascript">
    function Person() {
        this.name = "李四";
        this.age = 20;
        this.eat = eat;
    }
    function eat() {
        alert("吃完东西");
    }
    var p1 = new Person();
    var p2 = new Person();
    //因为eat属性都是赋值的同一个函数，所以是true
    alert(p1.eat === p2.eat); //true
</script>
```

> 但是上面的这种解决方法具有致命的缺陷：封装性太差。使用面向对象，目的之一就是封装代码，这个时候为了性能又要把代码抽出对象之外，这是反人类的设计。

3.3 使用组合模式解决上述两种缺陷

原型模式适合封装方法，构造方法模式适合封装属性，综合两种模式的优点就有了组合模式。

```
<script type="text/javascript">
    //在构造方法内部封装属性
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    //在原型对象内封装方法
    Person.prototype.eat = function (food) {
        alert(this.name + "爱吃" + food);
    }
    Person.prototype.play = function (playName) {
        alert(this.name + "爱玩" + playName);
    }

    var p1 = new Person("李四", 20);
    var p2 = new Person("张三", 30);
    p1.eat("苹果");
    p2.eat("香蕉");
    p1.play("志玲");
    p2.play("凤姐");
</script>
```

四、动态原型模式创建对象

前面讲到的组合模式，也并非完美无缺，有一点也是感觉不是很完美。把构造方法和原型分开写，总让人感觉不舒服，应该想办法把构造方法和原型封装在一起，所以就有了动态原型模式。

```

```