Web开发技术-JavaScript

面向对象的程序设计和函数表达式

内容提要

5 面向对象的程序设计

5.1 面向对象程序设计

5.2 创建对象

- 原型模式
- 组合使用构造函数模式和原型模式

5.3 继承

- 原型链

5.1 面向对象程序设计

面向对象（Object-oriented，OO）的三个基本特征

- 封装（Encapsulation）
- 继承（Inheritance）
- 多态（Polymorphism）

5.2 创建对象

回顾：创建自定义对象（1.3.3 节）

new Object() 方法

```
var person = new Object();
person.name = “Shaorong Wang”;
person.corporation = “BJFU”;

person.getName = function() {
  alert(this.name);
}
```

对象字面量方法（推荐）

```
var person = {
  name: “Shaorong Wang”,
  corporation: “BJFU”,
  getName: function() {
    alert(this.name);
  }
}
```

无法满足批量创建对象的需求。
需要将对象进行抽象、封装。

5.2.1 原型模式

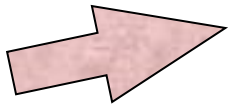
原型对象（Prototype）

- 每创建一个对象时，JavaScript 都会自动为该对象创建一个 prototype 属性，该属性是指向一个原型对象的指针。
- 原型对象中包含可以由特定类型的**所有**实例共享的属性和方法。
- 可以通过定义原型对象的属性，实现对象的封装。

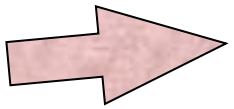
5.2.1 原型模式

使用 object.prototype 属性定义原型对象的属性和方法。

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();          // Shaorong Wang

var person2 = new Person();
person2.getName();          // Shaorong Wang
```



新建 Person 实例 1



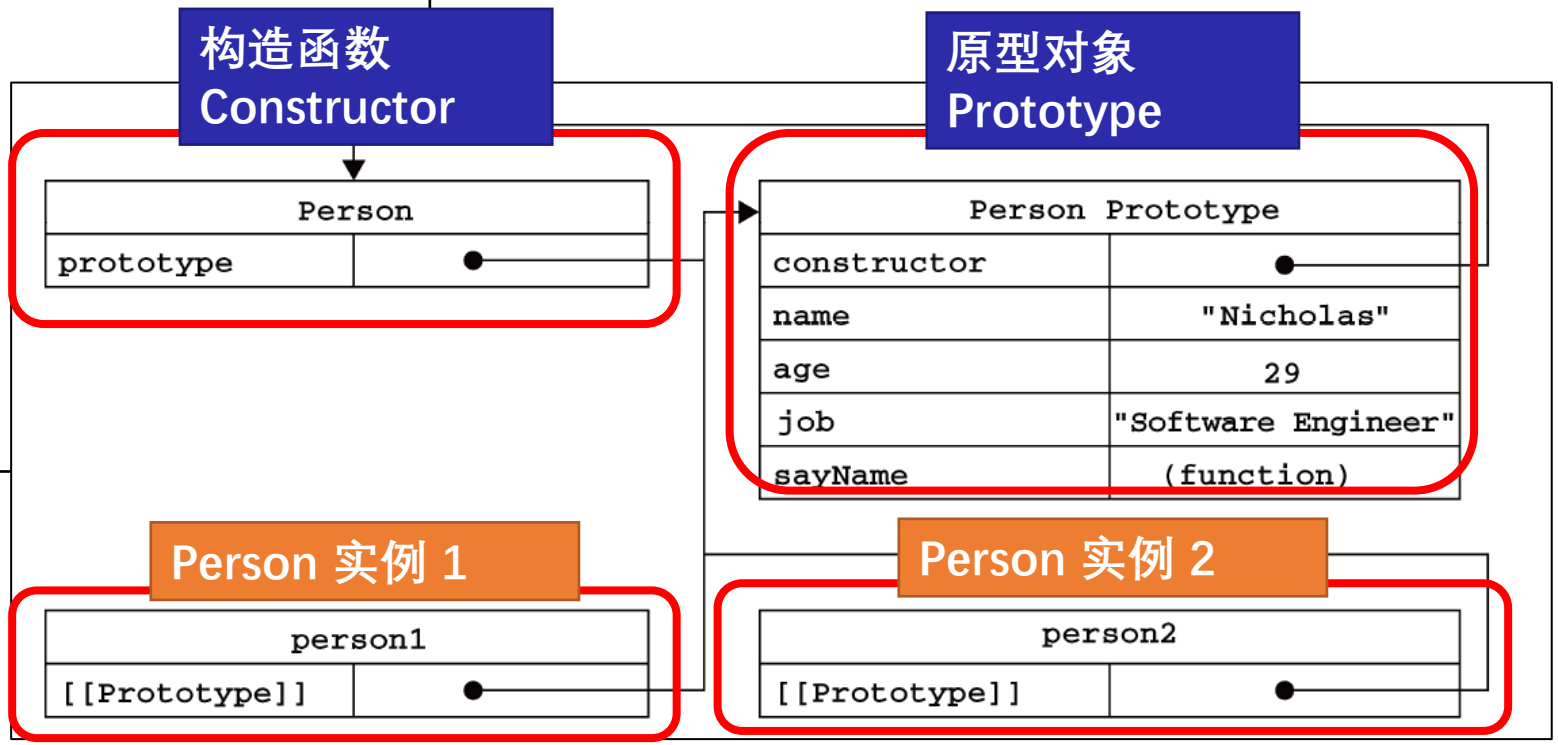
新建 Person 实例 2

Demo 2.12

5.2.1 原型模式

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();

var person2 = new Person();
person2.getName();
```

创建了 4 个对象

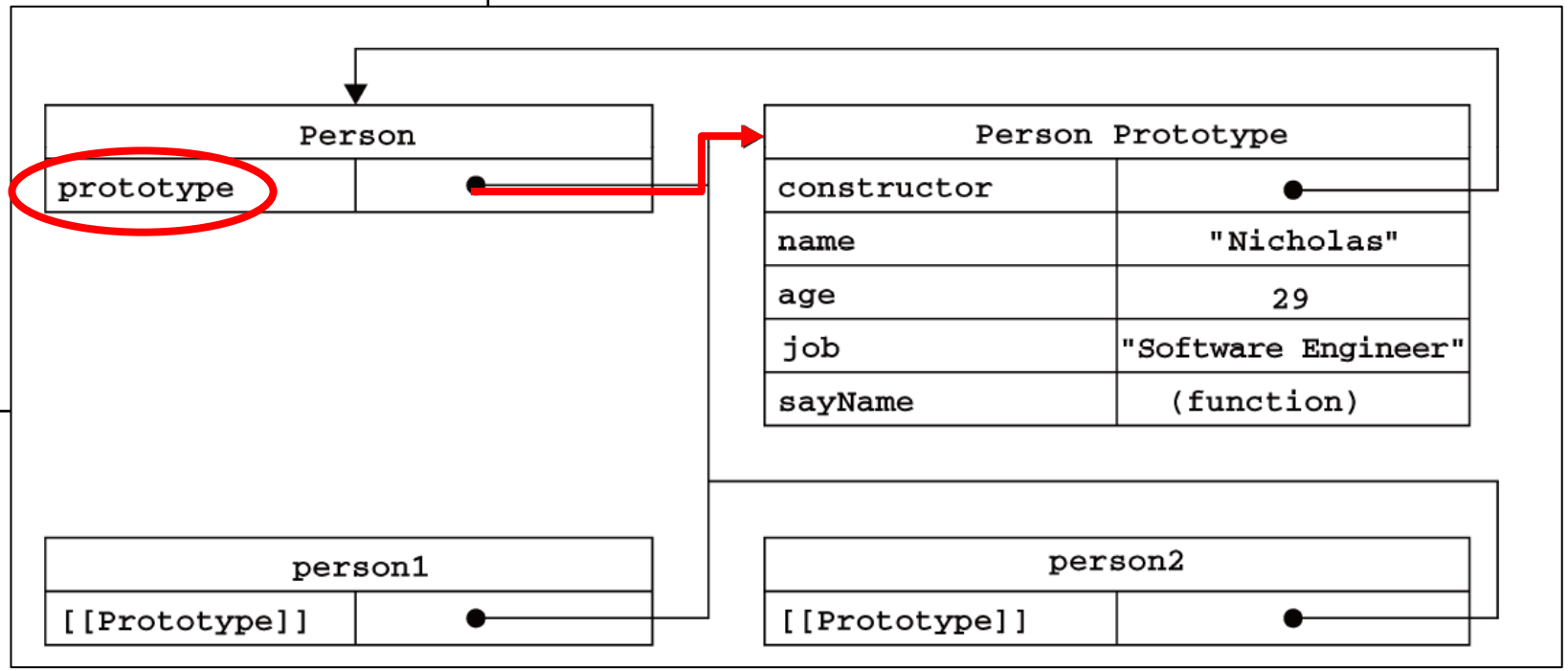


5.2.1 原型模式

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();

var person2 = new Person();
person2.getName();
```

1 只要创建新对象，就会根据一组特定的规则为该对象创建一个 Prototype 属性。

这个属性指向函数的原型对象。

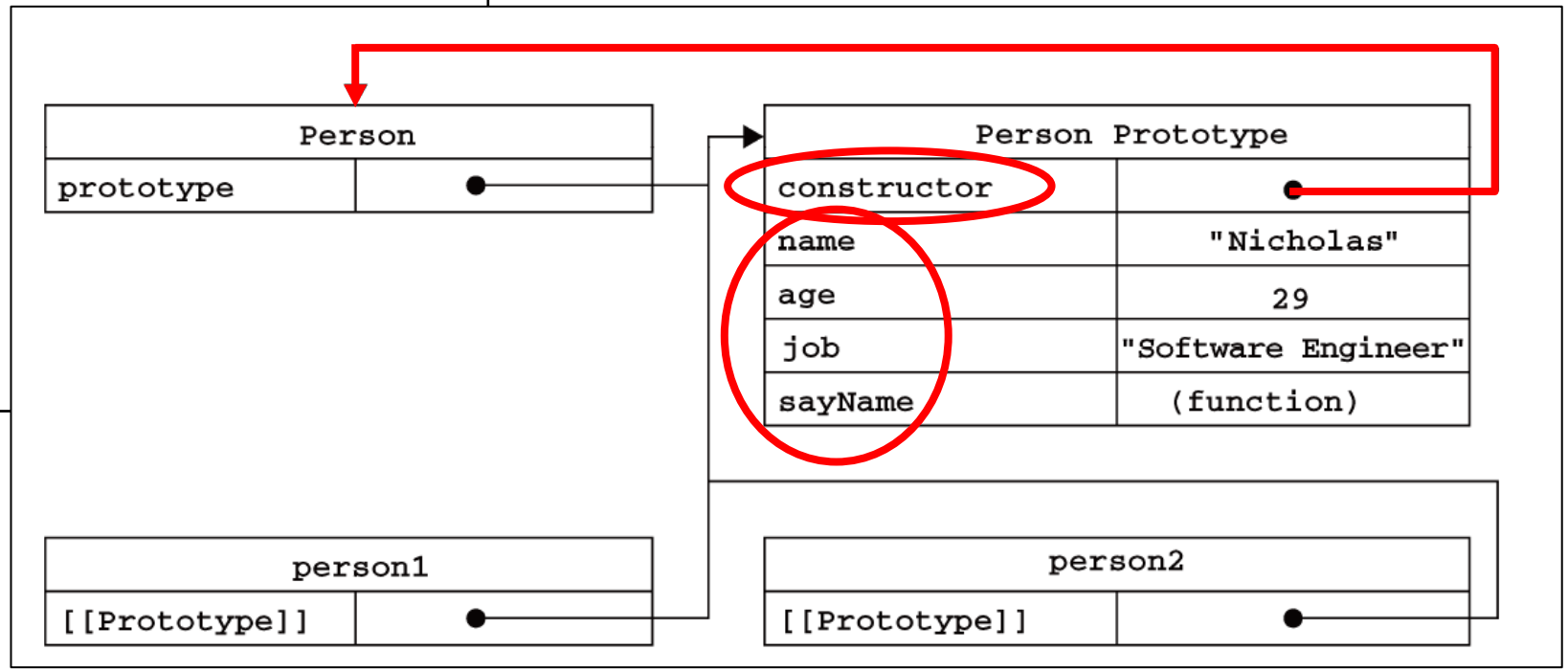


5.2.1 原型模式

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();

var person2 = new Person();
person2.getName();
```

2 原型对象具有 constructor 属性，指向 prototype 属性所在函数。

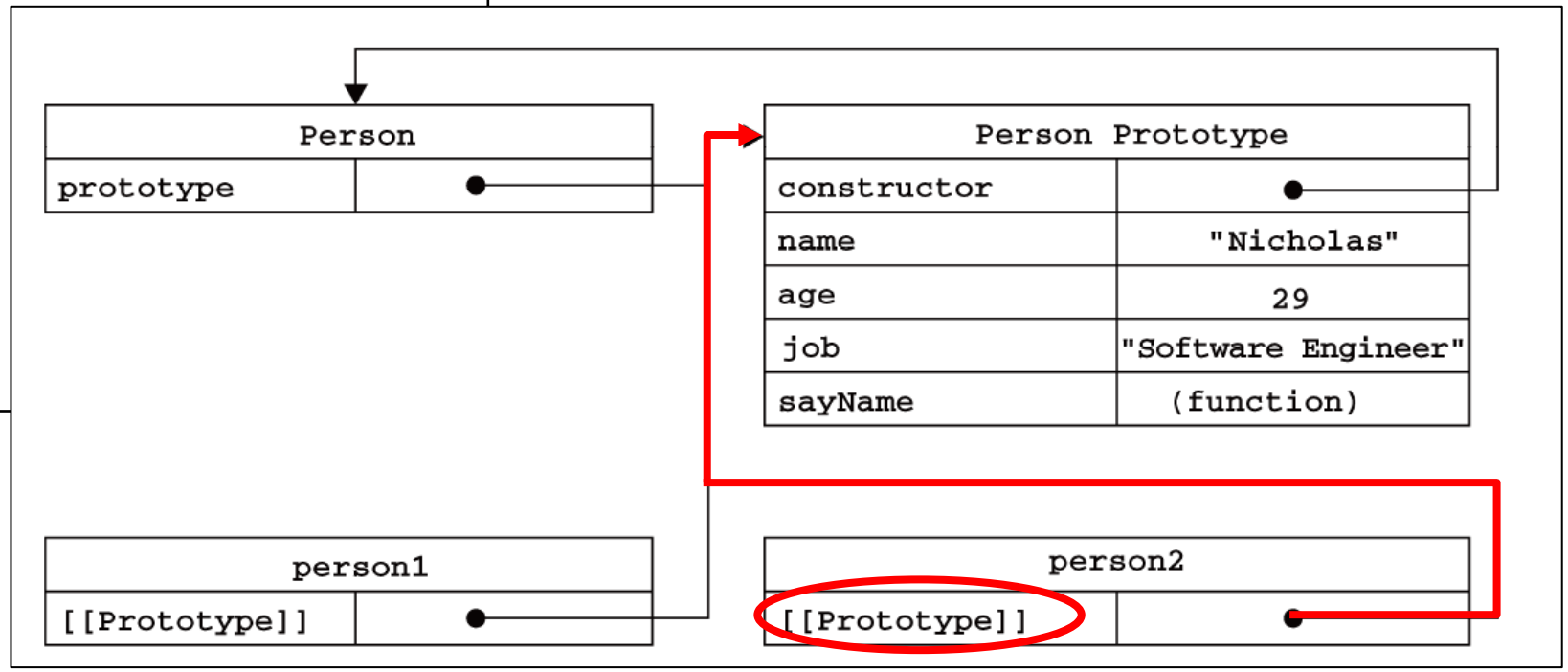
可以通过构造函数为原型添加其他属性和方法



5.2.1 原型模式

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();

var person2 = new Person();
person2.getName();
```

3 创建实例后，**实例**内部将自动包含一个 **[[Prototype]]** 指针作为 [[内部属性]]，**指向构造函数的原型对象**。



5.2.1 原型模式

属性的搜索顺序

- 先搜索对象实例本身，再搜索原型对象的属性。
- 若在实例中添加一个与原型对象实例同名的属性，该属性将**屏蔽**原型对象中的属性。

若使用 delete 操作符将新属性删除，原型中的属性将再次生效

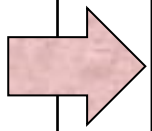
```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}

var person1 = new Person();
person1.getName();     // Shaorong Wang

person1.name = “Nicholas”;
person1.getName();    // Nicholas
```

5.2.1 原型模式

直接通过对象字面量重写原型对象

- 减少不必要输入，实现更好封装

```
function Person() { }

Person.prototype.name = "Shaorong Wang";
Person.prototype.corporation = "BJFU";
Person.prototype.getName = function() {
  alert(this.name);
}
```



```
function Person() { }

Person.prototype = {
  name: “Shaorong Wang”,
  corporation: “BJFU”,
  getName: function() {
    alert(this.name);
  }
}
```

5.2.1 原型模式

直接通过对象字面量重写原型对象

- 减少不必要输入，实现更好封装

```
function Person() { }

Person.prototype = {
  name: “Shaorong Wang”,
  corporation: “BJFU”,
  getName: function() {
    alert(this.name);
  }
}
```

- Person.prototype **指向**的也是**新对象**
- 创建这个新对象时会生成该新对象的原型对象
- 新对象的 constructor 属性不再指向 Person，而指向默认的 Object 构造函数

```
Person.prototype = {
  constructor: Person,
  name: “Shaorong Wang”,
  corporation: “BJFU”,
  getName: function() {
    alert(this.name);
  }
}
```

可以将 constructor 属性修改为适当的值

5.2.2 组合使用构造函数模式和原型模式

实现合理封装的同时自定义实例属性

```
function Person(name, corporation) {
  this.name = name;
  this.corporation = corporation;
}

Person.prototype = {
  constructor: Person,
  getName: function() {
  alert(this.name);
  }
}

var person1 = new Person("Shaorong Wang", "BJFU");
var person2 = new Person("Nicholas", "PKU");

person1.getName();          // Shaorong Wang
person2.getName();          // Nicholas
```

封装的最常用方法

Demo 2.13

5.3 继承

使用原型链实现继承

- 将新对象的**原型对象**指向**被继承类的新实例**实现**继承。**
- **注意：创建新实例时会创建一个 `[[Prototype]]` 属性的新对象。**

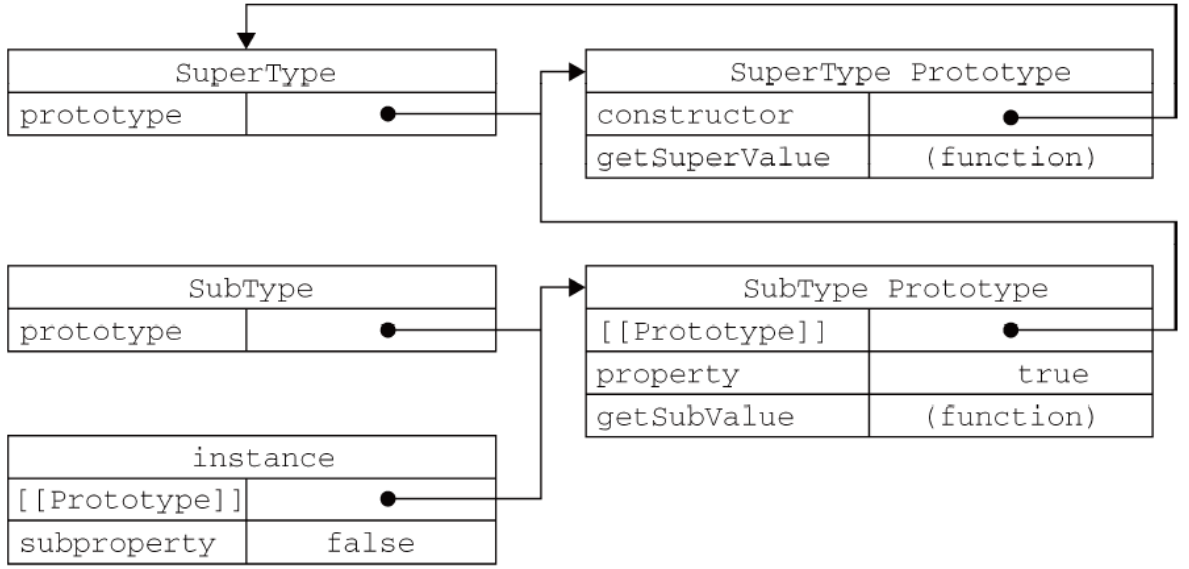
- 此时，新对象的原型对象 A 的 `[[Prototype]]` 属性变为一个指向被继承对象的原型对象 B 的指针，而原型对象 B 中包含着指向构造这个原型函数 B 的指针 `constructor` 对象。
- 新对象可以继续在原型对象中添加属性或方法。
- 层层递进，形成实例与原型对象的链条，为原型链。这样新的实例可以访问多次继承而来的原型对象的属性和方法。

5.3 继承

使用原型链实现继承

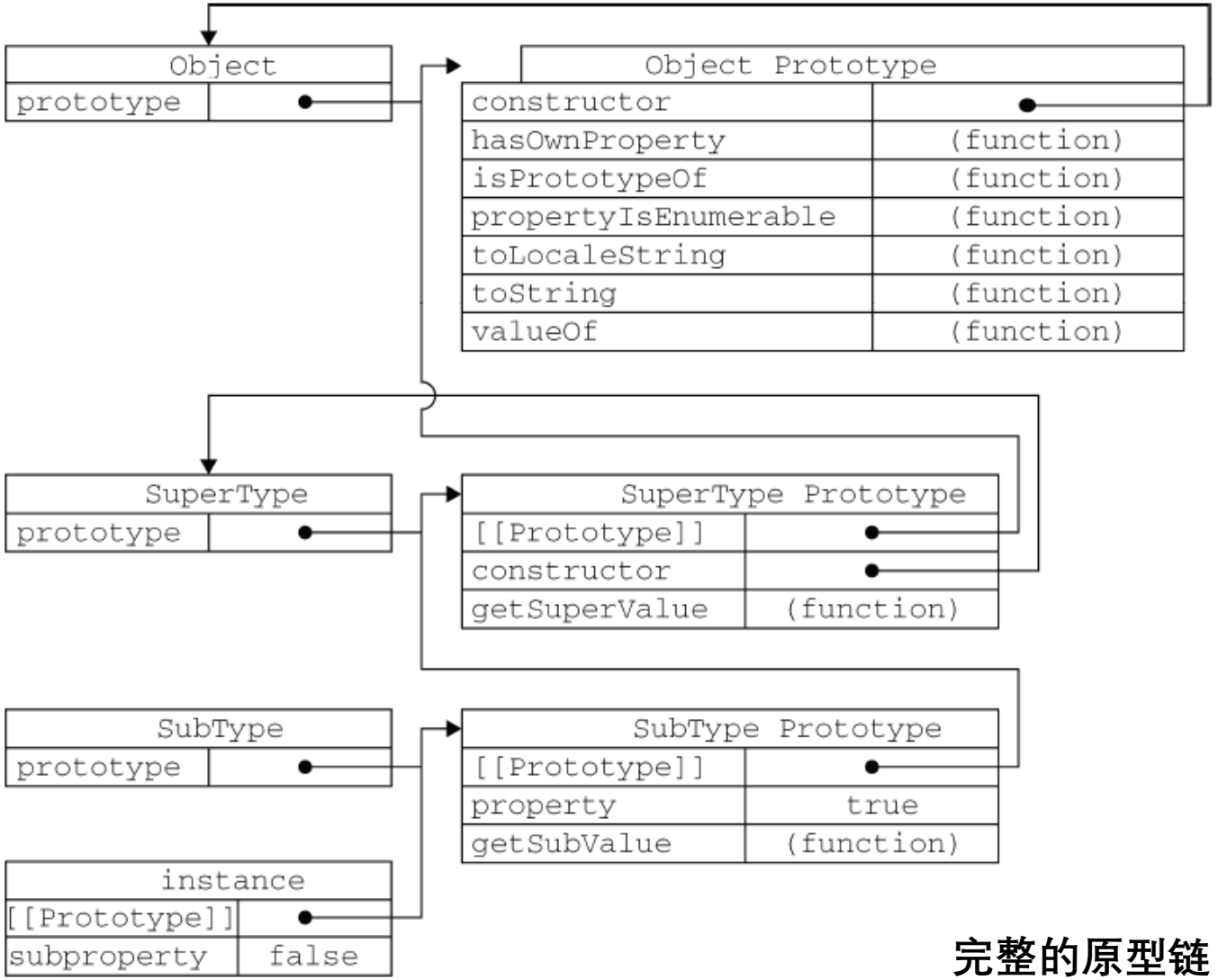
Demo 2.14

```
function SuperType() {
  this.property = true;
}

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

function SubType() {
  this.subproperty = false;
}
```




```
SubType.prototype = new SuperType();

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

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

5.3 继承


Object
prototype
Object Prototype
constructor
hasOwnProperty (function)
isPrototypeOf (function)
propertyIsEnumerable (function)
toLocaleString (function)
toString (function)
valueOf (function)
SuperType
prototype
SuperType Prototype
[[Prototype]]
constructor
getSuperValue (function)
SubType
prototype
SubType Prototype
[[Prototype]]
property true
getSubValue (function)
instance
[[Prototype]]
subproperty false

完整的原型链

6 函数表达式

回顾：定义函数的两种方法（4.1.1 节）

函数声明

```
function functionName(arg0, arg1) {
    // 函数体
}
```

享受 JavaScript 的函数声明提升机制：
在解析器首次读取代码时即将函数声明添加到执行环境中。
函数声明可以放在函数调用语句后面。

函数表达式：函数是对象

```
var functionName = function(arg1, arg2) {
    // 函数体
}
```

创建一个函数并赋值给变量 functionName
创建的函数是**匿名函数**，匿名函数的 name 属性是空字符串。
函数调用要在函数赋值后进行。否则报错。

6 函数表达式

作用域

- 当某个函数被调用时，会创建一个单独的执行环境及相应的作用域。
- 该作用域可读父作用域的变量和本作用域内的变量。
- 函数执行结束后，本作用域和作用域内变量**被销毁**。

```
var flag = true;
var setFlag = function() {
  alert(flag);   // true
  var newFlag = false;
}
setFlag();
alert(newFlag);  // Error: newFlag is not defined
```

Demo 2.15

6 函数表达式

使用匿名函数实现闭包

闭包：有权访问在另一个函数作用域中变量的函数

```
function createComparisionFunction(propertyName) {
  return function(object1, object2) {
    var value1 = object1[propertyName];
    var value2 = object2[propertyName];
    if (value1 < value2) {
      return -1;
    } else if (value1 > value2) {
      return 1;
    } else {
      return 0;
    }
  }
};
```

```
/* 按 corporation 排序 */
var compareNames =
createComparisionFunction("corporation");

var persons = [{ name: "Shaorong", corporation: "BJFU" },
{ name: "Nicholas", corporation: "PKU" }]

persons.sort(compareNames);
console.log(persons);
```