1.什么是原型链继承？什么是借用构造函数继承？什么是组合继承?

2.就原型链继承、借用构造函数继承、组合继承各写一个例子

3.测试新学内容。

js继承之原型链继承

面向对象编程都会涉及到继承这个概念，JS中实现继承的方式主要是通过原型链的方法。

一、构造函数、原型与实例之间的关系

　　每创建一个函数，该函数就会自动带有一个 prototype 属性。该属性是个指针，指向了一个对象，我们称之为 原型对象。什么是指针？指针就好比学生的学号，原型对象则是那个学生。我们通过学号找到唯一的那个学生。假设突然，指针设置 null, 学号重置空了，不要慌，对象还存在，学生也没消失。只是不好找了。

　　原型对象上默认有一个属性 constructor，该属性也是一个指针，指向其相关联的构造函数。

　　通过调用构造函数产生的实例，都有一个内部属性，指向了原型对象。所以实例能够访问原型对象上的所有属性和方法。

　　所以三者的关系是，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，而实例都包含一个指向原型对象的内部指针。通俗点说就是，实例通过内部指针可以访问到原型对象，原型对象通过constructor指针，又可以找到构造函数。

　　下面看一个例子：

function Dog (name) {

this.name = name;

this.type = 'Dog';

}

Dog.prototype.speak = function () {

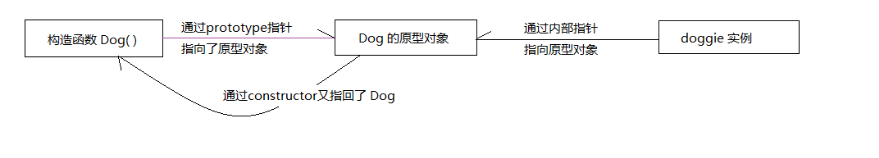
　　alert('wang');

}

var doggie = new Dog('jiwawa');

doggie.speak(); //wang

　　以上代码定义了一个构造函数 Dog(), Dog.prototype 指向的原型对象，其自带的属性construtor又指回了 Dog，即 Dog.prototype.constructor == Dog. 实例doggie由于其内部指针指向了该原型对象，所以可以访问到 speak方法。



　　Dog.prototype 只是一个指针，指向的是原型对象，但是这个原型对象并不特别，它也只是一个普通对象。假设说，这时候，我们让 Dog.protptype 不再指向最初的原型对象，而是另一个类 （Animal）的实例，情况会怎样呢？

二、原型链

　　前面我们说到，所有的实例有一个内部指针，指向它的原型对象，并且可以访问原型对象上的所有属性和方法。doggie实例指向了Dog的原型对象，可以访问Dog原型对象上的所有属性和方法；如果Dog原型对象变成了某一个类的实例 aaa，这个实例又会指向一个新的原型对象 AAA，那么 doggie 此时就能访问 aaa 的实例属性和 AA A原型对象上的所有属性和方法了。同理，新的原型对象AAA碰巧又是另外一个对象的实例bbb，这个实例bbb又会指向新的原型对象 BBB，那么doggie此时就能访问 bbb 的实例属性和 BBB 原型对象上的所有属性和方法了。

　　这就是JS通过原型链实现继承的方法了。看下面一个例子：

//定义一个 Animal 构造函数，作为 Dog 的父类

function Animal () {

this.superType = 'Animal';

}

Animal.prototype.superSpeak = function () {

alert(this.superType);

}

function Dog (name) {

this.name = name;

this.type = 'Dog';

}

//改变Dog的prototype指针，指向一个 Animal 实例

Dog.prototype = new Animal();

//上面那行就相当于这么写

//var animal = new Animal();

//Dog.prototype = animal;

Dog.prototype.speak = function () {

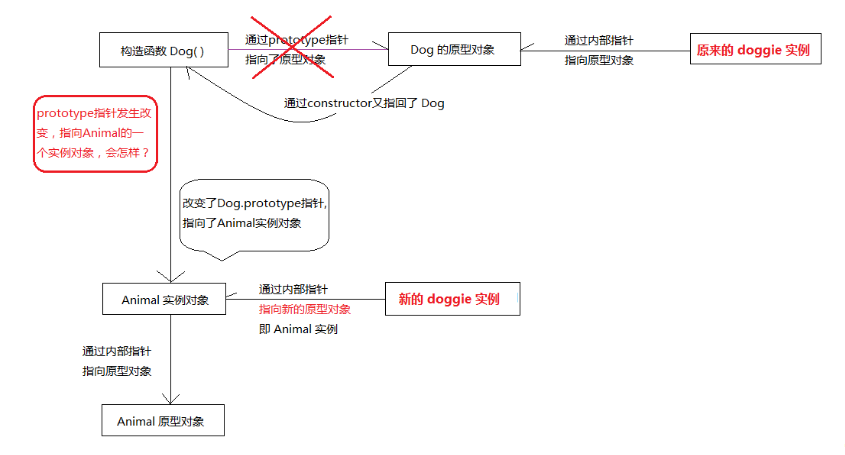
　　alert(this.type);

}

var doggie = new Dog('jiwawa');

doggie.superSpeak(); //Animal

　　解释一下。以上代码，首先定义了一个 Animal 构造函数，通过new Animal()得到实例，会包含一个实例属性 superType 和一个原型属性 superSpeak。另外又定义了一个Dog构造函数。然后情况发生变化，代码中加粗那一行，将Dog的原型对象覆盖成了 animal 实例。当 doggie 去访问superSpeak属性时，js会先在doggie的实例属性中查找，发现找不到，然后，js就会去doggie 的原型对象上去找，doggie的原型对象已经被我们改成了一个animal实例，那就是去animal实例上去找。先找animal的实例属性，发现还是没有 superSpeack, 最后去 animal 的原型对象上去找，诶，这才找到。



　　这就说明，我们可以通过原型链的方式，实现 Dog 继承 Animal 的所有属性和方法。

　　总结来说：就是当重写了Dog.prototype指向的原型对象后，实例的内部指针也发生了改变，指向了新的原型对象，然后就能实现类与类之间的继承了。（但是如果在重写原型对象之前，产生的实例，其内部指针指向的还是最初的原型对象。

js继承之构造函数继承

一、原型链的缺点

1.1 单纯的原型链继承最大的一个缺点，在于对原型中引用类型值的误修改。

　　先看一个例子：

　　 //父类：人

function Person () {

this.head = '脑袋瓜子';

}

//子类：学生，继承了“人”这个类

function Student(studentID) {

this.studentID = studentID;

}

Student.prototype = new Person();

var stu1 = new Student(1001);

console.log(stu1.head); //脑袋瓜子

stu1.head = '聪明的脑袋瓜子';

console.log(stu1.head); //聪明的脑袋瓜子

var stu2 = new Student(1002);

console.log(stu2.head); //脑袋瓜子

　　以上例子，我们通过重写 Student.prototype 的值为 Person 类的一个实例，实现了 Student 类对 Person 类的继承。所以 ，stu1 能访问到父类 Person 上定义的 head 属性，打印值为“脑袋瓜子”。我们知道，所有的 Student 实例都共享着原型对象上的属性。那么，如果我在 stu1 上改变了 head 属性值，是不是会影响原型对象上的 head 值呢？看我上面的代码就知道，肯定是不会。stu1 的 head 值确实是改变了，但是我重新实例化的对象 stu2 的 head 值仍旧不变。

　　这是因为，当实例中存在和原型对象上同名的属性时，会自动屏蔽原型对象上的同名属性。stu1.head = "聪明的脑袋瓜子" 实际上只是给 stu1 添加了一个本地属性 head 并设置了相关值。所以当我们打印 stu1.head 时，访问的是该实例的本地属性，而不是其原型对象上的 head 属性（它因和本地属性名同名已经被屏蔽了）。

　　刚才我们讨论的这个 head 属性是一个基本类型的值，可如果它是一个引用类型呢？这其中又会有一堆小九九。

　　其实原型对象上任何类型的值，都不会被实例所重写/覆盖。在实例上设置与原型对象上同名属性的值，只会在实例上创建一个同名的本地属性。

　　但是，原型对象上引用类型的值可以通过实例进行修改，致使所有实例共享着的该引用类型的值也会随之改变。

　　再看下面这个例子：

//父类：人

function Person () {

this.head = '脑袋瓜子';

this.emotion = ['喜', '怒', '哀', '乐']; //人都有喜怒哀乐

}

//子类：学生，继承了“人”这个类

function Student(studentID) {

this.studentID = studentID;

}

Student.prototype = new Person();

var stu1 = new Student(1001);

console.log(stu1.emotion); //['喜', '怒', '哀', '乐']

stu1.emotion.push('愁');

console.log(stu1.emotion); //["喜", "怒", "哀", "乐", "愁"]

var stu2 = new Student(1002);

console.log(stu2.emotion); //["喜", "怒", "哀", "乐", "愁"]

　　我们在刚才的 Person 类中又添加了一个 emotion 情绪属性，人都有喜怒哀乐嘛。尤其需要注意的是，这是一个引用类型的值。这时，stu1 认为他还很“愁”，所以就通过 stu1.emotion.push ( ) 方法在原来的基础上增加了一项情绪，嗯，打印出来“喜怒哀乐愁”，没毛病。可是 stu2 是个乐天派，他咋也跟着一起愁了呢？！肯定不对嘛～

　　这就是单纯的原型链继承的缺点，如果一个实例不小心修改了原型对象上引用类型的值，会导致其它实例也跟着受影响。

　　因此，我们得出结论，原型上任何类型的属性值都不会通过实例被重写，但是引用类型的属性值会受到实例的影响而修改。

1.2 原型链不能实现子类向父类中传参。这里就不细说了。

二、借用构造函数

2.1 实现原理

　　在解决原型对象中包含引用类型值所带来问题的过程中，开发人员开始使用一种叫做借用构造函数的技术。实现原理是，在子类的构造函数中，通过 apply ( ) 或 call ( )的形式，调用父类构造函数，以实现继承。

//父类：人

function Person () {

this.head = '脑袋瓜子';

this.emotion = ['喜', '怒', '哀', '乐']; //人都有喜怒哀乐

}

//子类：学生，继承了“人”这个类

function Student(studentID) {

this.studentID = studentID;

Person.call(this);

}

//Student.prototype = new Person();

var stu1 = new Student(1001);

console.log(stu1.emotion); //['喜', '怒', '哀', '乐']

stu1.emotion.push('愁');

console.log(stu1.emotion); //["喜", "怒", "哀", "乐", "愁"]

var stu2 = new Student(1002);

console.log(stu2.emotion); //["喜", "怒", "哀", "乐"]

　　细心的同学可能已经发现了，该例子与上面的例子非常相似，只是去掉了之前通过 prototype 继承的方法，而采用了 Person.call (this) 的形式实现继承。别忘了，函数只不过是一段可以在特定作用域执行代码的特殊对象，我们可以通过 call 方法指定函数的作用域。

　　（题外话：也许有的同学对 this 的指向还不完全清楚，我是这么理解的：谁调用它，它就指向谁。）

　　在 stu1 = new Student ( ) 构造函数时，是 stu1 调用 Student 方法，所以其内部 this 的值指向的是 stu1, 所以 Person.call ( this ) 就相当于Person.call ( stu1 )，就相当于 stu1.Person( )。最后，stu1 去调用 Person 方法时，Person 内部的 this 指向就指向了 stu1。那么Person 内部this 上的所有属性和方法，都被拷贝到了 stu1 上。stu2 也是同理，所以其实是，每个实例都具有自己的 emotion 属性副本。他们互不影响。说到这里，大家应该清楚一点点了吧。

　　总之，在子类函数中，通过call ( ) 方法调用父类函数后，子类实例 stu1, 可以访问到 Student 构造函数和 Person 构造函数里的所有属性和方法。这样就实现了子类向父类的继承，而且还解决了原型对象上对引用类型值的误修改操作。

2.2 缺点

　　这种形式的继承，每个子类实例都会拷贝一份父类构造函数中的方法，作为实例自己的方法，比如 eat()。这样做，有几个缺点：

　　1. 每个实例都拷贝一份，占用内存大，尤其是方法过多的时候。（函数复用又无从谈起了，本来我们用 prototype 就是解决复用问题的）

　　2. 方法都作为了实例自己的方法，当需求改变，要改动其中的一个方法时，之前所有的实例，他们的该方法都不能及时作出更新。只有后面的实例才能访问到新方法。

//父类：人

function Person () {

this.head = '脑袋瓜子';

this.emotion = ['喜', '怒', '哀', '乐']; //人都有喜怒哀乐

this.eat = function () {

console.log('吃吃喝喝');

}

this.sleep = function () {

console.log('睡觉');

}

this.run = function () {

console.log('快跑');

}

}

　　所以，无论是单独使用原型链继承还是借用构造函数继承都有自己很大的缺点，最好的办法是，将两者结合一起使用，发挥各自的优势。我将在下一篇文章作出解释。

js之结合原型链继承和借用构造函数继承

一、回顾借用构造函数继承的缺点

先看我们之前在借用构造函数继承中最后用到的代码：

//父类：人

function Person () {

this.head = '脑袋瓜子';

this.emotion = ['喜', '怒', '哀', '乐']; //人都有喜怒哀乐

this.eat = function () {

console.log('吃吃喝喝');

}

this.sleep = function () {

console.log('睡觉');

}

this.run = function () {

console.log('快跑');

}

}

//子类：学生，继承了“人”这个类

function Student(studentID) {

this.studentID = studentID;

Person.call(this);

}

//Student.prototype = new Person();

var stu1 = new Student(1001);

console.log(stu1.emotion); //['喜', '怒', '哀', '乐']

stu1.emotion.push('愁');

console.log(stu1.emotion); //["喜", "怒", "哀", "乐", "愁"]

var stu2 = new Student(1002);

console.log(stu2.emotion); //["喜", "怒", "哀", "乐"]

　　在这段代码中，我们通过借用构造函数继承，保证了 stu1 和 stu2 都有各自的父类属性副本，从而使得各自 emotion 互不影响。但同时带来的问题是，stu1 和 stu2 都拷贝了 Person 类中的所有属性和方法，而在 Person 类中，像 eat ( ), sleep ( ), run ( ) 这类方法应该是公用的，而不需要添加到每个实例上去，增大内存，尤其是这类方法较多的时候。

二、结合使用两种继承模式

　　所以我们想到，是否能把这些方法挂载到父类的原型对象上去，实现方法复用，然后子类通过原型链继承，就能调用这些方法啦？～

//父类：人

function Person () {

this.head = '脑袋瓜子';

this.emotion = ['喜', '怒', '哀', '乐']; //人都有喜怒哀乐

}

//将 Person 类中需共享的方法放到 prototype 中，实现复用

Person.prototype.eat = function () {

console.log('吃吃喝喝');

}

Person.prototype.sleep = function () {

console.log('睡觉');

}

Person.prototype.run = function () {

console.log('快跑');

}

//子类：学生，继承了“人”这个类

function Student(studentID) {

this.studentID = studentID;

Person.call(this);

}

Student.prototype = new Person(); //此时 Student.prototype 中的 constructor 被重写了，会导致 stu1.constructor === Person

Student.prototype.constructor = Student; //将 Student 原型对象的 constructor 指针重新指向 Student 本身

var stu1 = new Student(1001);

console.log(stu1.emotion); //['喜', '怒', '哀', '乐']

stu1.emotion.push('愁');

console.log(stu1.emotion); //["喜", "怒", "哀", "乐", "愁"]

var stu2 = new Student(1002);

console.log(stu2.emotion); //["喜", "怒", "哀", "乐"]

stu1.eat(); //吃吃喝喝

stu2.run(); //快跑

console.log(stu1.constructor); //Student

　　首先，我们将 Person 类中需要复用的方法提取到 Person.prototype 中，然后设置 Student 的原型对象为 Person 类的一个实例，这样 stu1 就能访问到 Person 原型对象上的属性和方法了。其次，为保证 stu1 和 stu2 拥有各自的父类属性副本，我们在 Student 构造函数中，还是使用了 Person.call ( this ) 方法。如此，结合原型链继承和借用构造函数继承，就完美地解决了之前这二者各自表现出来的缺点。