原型模式不仅是一种设计模式，它还是一种**编程范式**（programming paradigm），是 JavaScript 面向对象系统实现的根基。

在原型模式下，当我们想要创建一个对象时，会先找到一个对象作为原型，然后通过**克隆原型**的方式来创建出一个与原型一样（共享一套数据/方法）的对象。在 JavaScript 里，`Object.create` 方法就是原型模式的天然实现——准确地说，只要我们还在借助 `Prototype` 来实现对象的创建和原型的继承，那么我们就是在应用原型模式。

有的设计模式资料中会强调，原型模式就是拷贝出一个新对象，认为在 JavaScript 类里实现了深拷贝方法才算是应用了原型模式。这是非常典型的对 JAVA/C++ 设计模式的生搬硬套，更是对 JavaScript 原型模式的一种误解。事实上，在JAVA中，确实存在原型模式相关的克隆接口规范。但在 JavaScript 中，我们使用原型模式，并不是为了得到一个副本，而是为了得到与构造函数（类）相对应的类型的实例、实现数据/方法的共享。克隆是实现这个目的的方法，但克隆本身并不是我们的目的。

以类为中心的语言和以原型为中心的语言

相信很多小伙伴读到这儿还会有些迷惑：使用 JavaScript 以来，我确实离不开 `Prototype`，按照上面的说法，也算是原型模式重度用户了。但这个原型模式用得我一脸懵逼啊——难道我还有除了 `Prototype` 以外的选择?

Java 中的类

作为 JavaScript 开发者，我们确实没有别的选择 —— 毕竟开头我们说过，原型模式是 JavaScript 这门语言面向对象系统的根本。但在其它语言，比如 JAVA 中，类才是它面向对象系统的根本。所以说在 JAVA 中，我们可以选择不使用原型模式 —— 这样一来，所有的实例都必须要从类中来，当我们希望创建两个一模一样的实例时，就只能这样做（假设实例从 Dog 类中来,必传参数为姓名、性别、年龄和品种）：

```
Dog dog = new Dog('旺财', 'male', 3, '柴犬')

Dog dog_copy = new Dog('旺财', 'male', 3, '柴犬')
```

没错，我们不得不把一模一样的参数传两遍，非常麻烦。而原型模式允许我们通过调用克隆方法的方式达到同样的目的，比较方便，所以 Java 专门针对原型模式设计了一套接口和方法，在必要的场景下会通过原型方法来应用原型模式。当然，在更多的情况下，Java 仍以“实例化类”这种方式来创建对象。

所以说在以类为中心的语言中，原型模式确实不是一个必选项，它只有在特定的场景下才会登场。

JavaScript 中的“类”

这时有一部分小伙伴估计要炸毛了：啥？？？JavaScript 只能用 `Prototype`？我看你还活在上世纪，ES6 早就支持类了！现在我们 JavaScript 也是以类为中心的语言了。

这波同学的思想非常危险，因为 ES6 的类其实是原型继承的语法糖:

> ECMAScript 2015 中引入的 JavaScript 类实质上是 JavaScript 现有的基于原型的继承的语法糖。类语法不会为 JavaScript 引入新的面向对象的继承模型。 ——MDN

当我们尝试用 class 去定义一个 Dog 类时：

```
class Dog {
  constructor(name ,age) {
   this.name = name
   this.age = age
  }

  eat() {
    console.log('肉骨头真好吃')
  }
}
```

其实完全等价于写了这么一个构造函数:

```
function Dog(name, age) {
  this.name = name
  this.age = age
}

Dog.prototype.eat = function() {
  console.log('肉骨头真好吃')
}
```

所以说 JavaScript 这门语言的根本就是原型模式。在 Java 等强类型语言中，原型模式的出现是为了实现类型之间的解耦。而 JavaScript 本身类型就比较模糊，不存在类型耦合的问题，所以说咱们平时**根本不会刻意地去使用原型模式**。因此我们此处不必强行把原型模式当作一种设计模式去理解，把它作为一种编程范式来讨论会更合适。

谈原型模式，其实是谈原型范式

原型编程范式的核心思想就是**利用实例来描述对象，用实例作为定义对象和继承的基础**。在 JavaScript 中，原型编程范式的体现就是**基于原型链的继承**。这其中，对原型、原型链的理解是关键。

原型

在 JavaScript 中，每个构造函数都拥有一个 `prototype` 属性，它指向构造函数的原型对象，这个原型对象中有一个 construtor 属性指回构造函数；每个实例都有一个 `__proto__` 属性，当我们使用构造函数去创建实例时，实例的 `__proto__` 属性就会指向构造函数的原型对象。

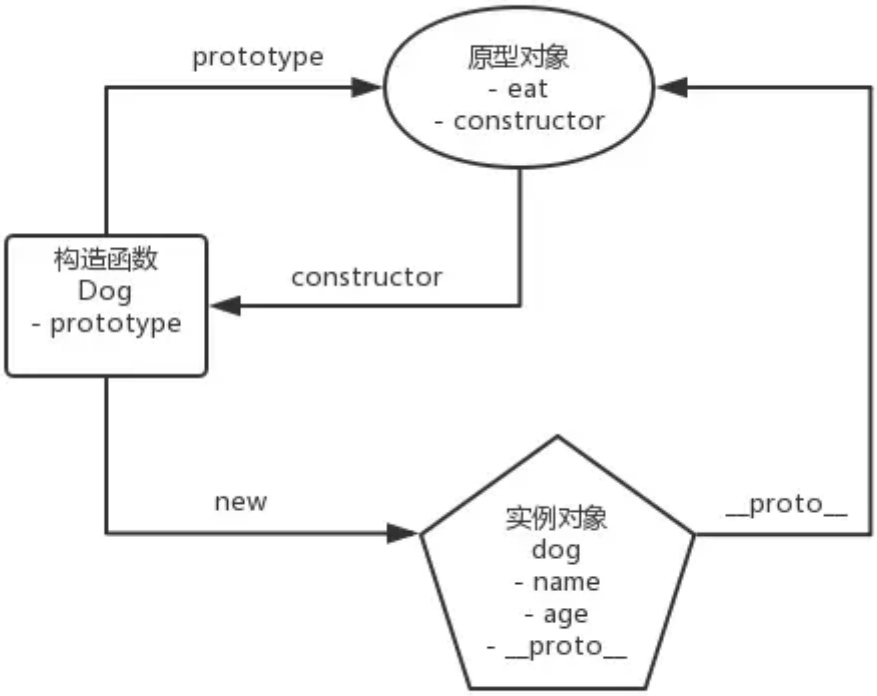
具体来说，当我们这样使用构造函数创建一个对象时：

```
// 创建一个Dog构造函数
function Dog(name, age) {
  this.name = name
  this.age = age
}

Dog.prototype.eat = function() {
  console.log('肉骨头真好吃')
}

// 使用Dog构造函数创建dog实例
const dog = new Dog('旺财', 3)
```

这段代码里的几个实体之间就存在着这样的关系：



原型链

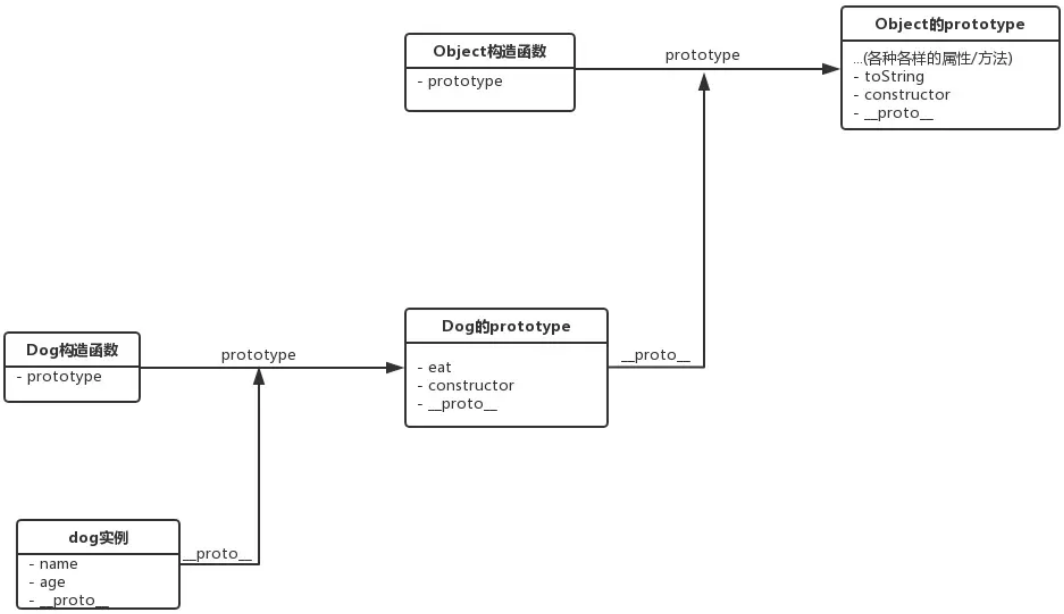
现在我在上面那段代码的基础上，进行两个方法调用:

```
// 输出"肉骨头真好吃"
dog.eat()

// 输出"[object Object]"
dog.toString()
```

明明没有在 dog 实例里手动定义 eat 方法和 toString 方法，它们还是被成功地调用了。这是因为当我试图访问一个 JavaScript 实例的属性/方法时，它首先搜索这个实例本身；当发现实例没有定义对应的属性/方法时，它会转而去搜索实例的原型对象；如果原型对象中也搜索不到，它就去搜索原型对象的原型对象，这个搜索的轨迹，就叫做原型链。

以我们的 eat 方法和 toString 方法的调用过程为例，它的搜索过程就是这样子的：



楼上这些彼此相连的 `prototype`，就组成了一个原型链。 注： 几乎所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例，除了 `Object.prototype` （当然，如果我们手动用 `Object.create(null)` 创建一个没有任何原型的对象，那它也不是 Object 的实例）。

以上为大家介绍了原型、原型链等 JavaScript 中核心的基础知识。这些不仅是基础中的基础，也是面试中的重点。此外在面试中，一些面试官可能会刻意混淆 JavaScript 中原型范式和强类型语言中原型模式的区别，当他们这么做的时候不一定是因为对语言、对设计模式的理解有问题，而很有可能是为了考察你**对象的深拷贝**。

对象的深拷贝

这类题目的发问方式又很多，除了“模拟 JAVA 中的克隆接口”、“JavaScript 实现原型模式”以外，它更常见、更友好的发问形式是“请实现JS中的深拷贝”。

实现 JavaScript 中的深拷贝，有一种非常取巧的方式 —— JSON.stringify：

```
const liLei = {
    name: 'lilei',
    age: 28,
    habits: ['coding', 'hiking', 'running']
}

const liLeiStr = JSON.stringify(liLei)
const liLeiCopy = JSON.parse(liLeiStr)

liLeiCopy.habits.splice(0, 1)
console.log('李雷副本的habits数组是', liLeiCopy.habits)
console.log('李雷的habits数组是',  liLei.habits)
```

丢进控制台检验一下，我们发现引用类型也被成功拷贝了，副本和本体相互不干扰，正合我意~

但是注意，这个方法存在一些局限性，比如无法处理 function、无法处理正则等等——只有当你的对象是一个严格的 JSON 对象时，可以顺利使用这个方法。在面试过程中，大家答出这个答案没有任何问题，但不要仅仅答这一种做法。

深拷贝没有完美方案，每一种方案都有它的边界 case。而面试官向你发问也并非是要求你破解人类未解之谜，多数情况下，他只是希望考查你对**递归**的熟练程度。所以递归实现深拷贝的核心思路，大家需要重点掌握（解析在注释里）：

```
function deepClone(obj) {
    // 如果是 值类型 或 null，则直接return
    if(typeof obj !== 'object' || obj === null) {
        return obj
    }

    // 定义结果对象
    let copy = {}

    // 如果对象是数组，则定义结果数组
    if(obj.constructor === Array) {
        copy = []
    }

    // 遍历对象的key
    for(let key in obj) {
        // 如果key是对象的自有属性
        if(obj.hasOwnProperty(key)) {
            // 递归调用深拷贝方法
            copy[key] = deepClone(obj[key])
        }
    }

    return copy
}
```

调用深拷贝方法，若属性为值类型，则直接返回；若属性为引用类型，则递归遍历。这就是我们在解这一类题时的核心的方法。

拓展阅读

深拷贝在命题时，可发挥的空间主要在于针对不同数据结构的处理，比如除了考虑 Array、Object，还需要考虑一些其它的数据结构（Map、Set 等）；此外还有一些极端 case（循环引用等）的处理等等。深拷贝的实现细节，这里为大家推荐两个阅读材料：

- **jQuery中的extend方法源码**
- **深拷贝的终极探索**

想要在深拷贝这一命题上拿高分的同学，不妨点开一看，相信你的收获会比你想象中更多~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）