属性

# 属性

## 概述

属性是对产品性质的描述，是区分产品差异性的集合。 在电商中，属性通常分为关键属性、 销售属性、 非关键属性、 商品属性。

（1） 关键属性：能够确认唯一“产品”的属性。 关键属性可以是一个属性，也可以是多个属性的组合。 例如：通过手机的“品牌”、 “型号”两个属性组合就能确定唯一的产品，这两个就是关键属性；通过服装的“品牌”， “货号”两个属性组合能确定唯一的产品，所以这两个也是关键属性。

（2） 销售属性：也称为**规格属性**。 其是组成**SKU**的特殊属性，它会影响买家的购买和卖家的库存管理。 例如服装的“颜色”、 “套餐”和“尺码”，都是销售属性。

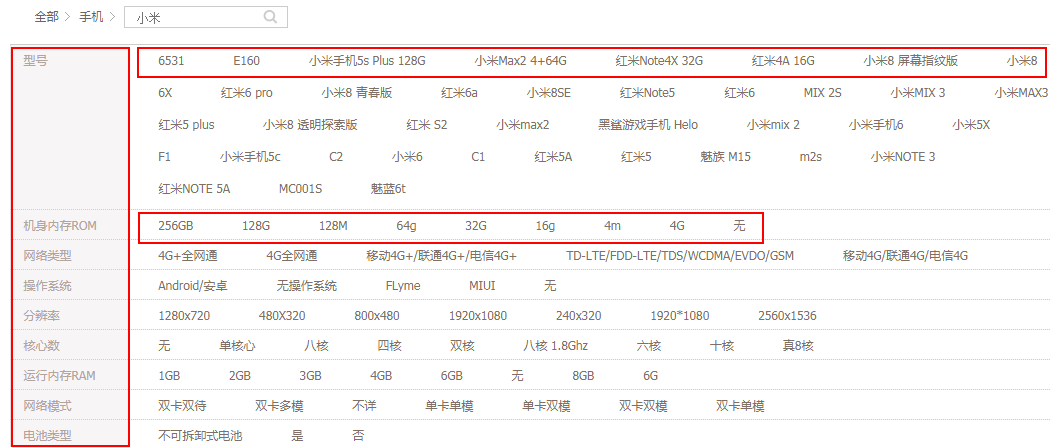
（3） 非关键属性：除关键属性、 销售属性外的其他属性。 非关键属性不一定是非必填项，有时为了商品信息完整，也会被设为必填项。注意，产品的非关键属性并不包括商品属性。

（4） 商品属性：表示商品的特有特征，比如新旧程度、 保修方式等，不能作为产品的属性。

## 作用

**属性的定义对于良好的消费体验有着至关重要的意义，对搜索、 索引、 筛选都能起到至关重要的作用。 商品搜索能力，除了标题、 类目，很大程度依赖于商品属性，条件筛选的基础数据也是商品属性和规格属性。 完善商品属性对于实现良好用户体验至关重要。**

淘宝小米手机搜索页面，属性显示如下：



同学们想想，数据表应该如何设计为好？？？

在设计表之前，我们先来理清一下，它和其他类目及品牌之间的关系吧。

## 关系

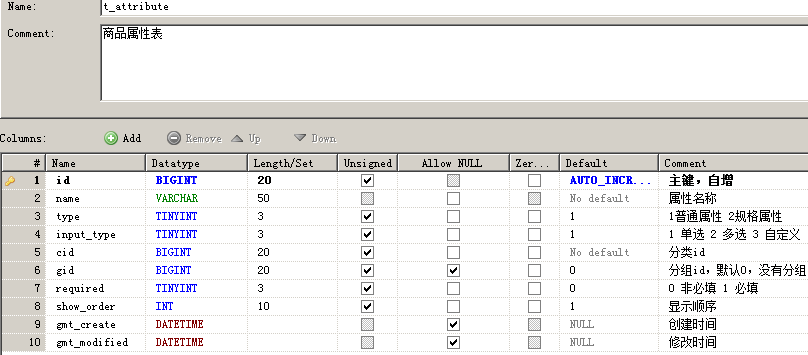
理清分类、品牌和属性之间的关系并不难，同学们可以想下自己购买商品时，搜索商品的流程，一般是进首页，然后选择分类，再是品牌，然后再通过属性，基本上就能搜索到比较准确的商品了。

注意的地方：其中一个属性，大部分情况下都是有多个固定值的，少数是用户自定义的，通过前面的淘宝页面大家也能看出来。

还有一种特殊情况，在商品详情页会存在属性分组显示的情况，如下图：



# 表结构



# 代码

## 新增



|  |
| --- |
| @PostMapping("/add")  **public** ServiceMessage add(@RequestBody Attribute attribute) {  **try** {  attributeService.add(attribute);  **return** ServiceMessage.*ok*();  } **catch** (Exception e) {  e.printStackTrace();  **return** ServiceMessage.*error*();  }  } |

## 查询



|  |
| --- |
| <select id=*"selectByAttribute"* parameterType=*"com.go9.goods.pojo.Attribute"* resultMap=*"BaseResultMap"*>  select  <include refid=*"Base\_Column\_List"* />  from t\_attribute  <where>  <if test=*"type != null"*>  type = #{type,jdbcType=TINYINT}  </if>    <if test=*"cid != null"*>  and cid = #{cid,jdbcType=BIGINT}  </if>  </where>  </select> |

# 讨论（扩展）

前面跟大家零散的提及了一些表的设计经验，这里再做个简单的总结吧。直接就讲重点了。大家都知道，关系型数据库表本质是一个二维数据结构，和我们面向对象中的“对象”结构不匹配，所以为了对上层的应用开发人员隐藏这种不匹配，从而简化编程，提高编程效率，就出现了“**ORM**”这种框架，例如：经典的**Hibernate**。然而这个框架有个很大的问题，就是对表的结构有太多的限制。因为这个框架属于非常老的框架，在设计之初是认为表的设计结构是遵循三大范式的（很久以前，那时候还没有电商类型的项目，大部分项目都是传统的**ERP**项目）。那么，何为**三大范式**呢？？这个这里不做过多讨论，可以简单的描述如下：

**1NF:字段不可分;**

**2NF:有主键，非主键字段依赖主键;**

**3NF:非主键字段不能相互依赖;**

**解释:**

**1NF:原子性 字段不可再分,否则就不是关系数据库;**

**2NF:唯一性 一个表只说明一个事物;**

**3NF:每列都与主键有直接关系，不存在传递依赖;**

有兴趣的同学可以再去找找相关的论文，仔细学习。这个你要是真想去研究，也是一天两天讲不完的，现在还出现了扩展范式等等。

提示：关系型数据库表设计的难点不是表结构本身，可能是对关系的分析和处理上，但也是有一个基本的设计原则的，总结如下：

**表之间的关系，一般分为三种：一对一，一对多（多对一），多对多**

**属于一对一关系的表很少，有也可以合并成一张表的，例如：学生、身份证及学号的关系。如果做成多张表，关联关系使用外键。一对多（多对一）关系最常见，大家最熟悉，也是通过外键来关联关系。那么多对多呢？这个麻烦点，一般都是通过扩展一张额外的关联表来表达这种关系的。**

这种严格的范式设计最大的缺点就是，关联表多了，关联查询就多了，在传统型的项目里，高并发的场景很少，基本不会出现，所以查询的速度快慢就变得不重要。架构师在技术选型时，更多的考虑的不是性能，而是开发速度，所以那个时候Hibernate这种框架受到欢迎，也是有它的道理的。然而时代变了，现在就算不是大型的电商项目，也经常会出现一些对性能要求较高的场景，这时候有必要对表的设计做出改变，要根据具体的业务场景去设计表，而不是教条的死报着范式去设计表了，也就是对设计人员要求越来越高了。

**提示：永远没有最好的设计，只有最合适的设计。**

就我们当前做的这个go8项目来说，除了前面设计品牌表时，加入了冗余的类名字段外。为了减少表之间的关联查询，我们去除了不少多对多关系时的扩展关联表。前提是在减少了关联表之后，这种关系依然能被推导的出来。如果不行，也只能关联。以后我们还有其他方法来提高性能。但是这里还是要说一下，能在数据库层面解决的性能问题，尽量就在数据库层解决掉。因为引入任何第三方的解决方案，解决问题的同时，同样会带来新的问题，也就意味着会产生更多的潜在bug。

还是举个例子来说明我们的设计过程吧，就拿产品属性、属性值来说，属性和属性值的关系应该是一对多的关系，比如，颜色属性值有红色，黄色等等。但是如果算上分类的话，就是多对多的关系，比如：一个红色可以是手机颜色，也可以是苹果颜色。这时应该再建立一张额外的**属性和属性值的关系关联表，**这样一个颜色值，例如红色，就能被重复使用，不会出现数据冗余的现象。但是go8数据库里并没有出现这个关联表，只有属性和属性值2张表。这样导致的结果就是，在属性值表里，会出现重复的，比如，“红色”这样的属性值。这是**另一种数据冗余方式**，是被动的，和前面主动在品牌表里强加分类名称冗余是不一样的。数据冗余的坏处就是增加了磁盘的存储空间，好处就是减少了关联查询，提高了查询速度。这就是典型的一种性能优化方式：

**用空间换时间！！**