

# 对象关系映射

## Part.1 理论基础

张子阳

[www.tracefact.net](http://www.tracefact.net)  
[jimmy\\_dev@163.com](mailto:jimmy_dev@163.com)

# 引言

大多数情况下，大家都在使用面向对象的思想进行程序开发与设计。与此同时，和我们打交道最多的数据库莫过于关系数据库了。而如何使这两个不同领域的对象相互协作自然成了我们日常讨论最多的话题之一。

作为本系列的第一篇文章，我主要向大家介绍了理解对象关系映射的一些预备知识和基础概念。主要包括：一对一关系、面向对象基础、关系基础 并对 对象与关系之间存在的差异作了简要的讨论。

对象关系映射的英文全名为：Object Relational Mapping，简称为：ORM。为了简便，本文下面提到对象关系映射，均使用 ORM 。

## 一对一关系

在讲述对象关系映射 ORM 之前，我们首先需要了解一下什么是一对一关系 以及 如何实现一对一关系。一对一关系也叫做超类/子类关系。

在实际的开发中，大家最熟悉的数据库关系可能就是一对多了。比如说一个分类下有很多篇随笔，又比如一篇随笔下有很多条评论。不知不觉中，对于需要使用一对一来实现的情况，也不假思索地使用一对多的方式去实现了。

我们来看这样一个例子，为了演示我做了简化：

假如我们正在设计一个论坛的用户表 User，那么一般来说需要这样几个字段：UserId，用户 ID；Name，用户名称；Password，密码；Email，电子邮件。我们还要设置管理员和版主，所以我们继续添加字段：AdminPwd，后台密码；Level，用户级别：0，普通会员；1，版主；2，管理员。

这个表的建立脚本是这样的：

```
Create Table [User]
(
    UserId          Int Identity(1,1),
    Name            Varchar(30)      Not Null,
    Password        Varchar(50)      Not Null,
    Email           Varchar(50)      Not Null,
    AdminPwd        Varchar(50)      Null,
    Level           TinyInt          Not Null Default 0 Constraint ck_UserLevel
Check(Level between 0 and 2)        --0,普通会员; 1,版主; 2,管理员

    Constraint pk_User Primary Key(UserId)    -- 建立主键
)
```

很快就会发现，用户与管理员的比例往往是 1000:1。这时候对于 99.9%的用户来说，它的 AdminPwd 字段都是 Null，Level 字段是 0。于是，为了避免这种不必要的空间浪费，我们将管理员抽象出来，新建一个 Admin 表。

这时候，问题出来了，很多人都是采用这种方式建立的：

```
Create Table [User]
(
    UserId          Int Identity(1,1),
    Name            Varchar(30)      Not Null,
    Password        Varchar(50)      Not Null,
```

```

    Email          Varchar(50)          Not Null,
)
Constraint pk_User Primary Key(UserId)    -- 建立主键
)
Create Table Admin
(
    AdminId      Int Identity(1,1),
    AdminPwd     Varchar(50)           Null,
    Level        TinyInt               Not Null Default 1 Constraint ck_UserLevel
Check(Level between 1 and 2),
    FKUserId     Int                  Not Null
)
Constraint pk_Admin Primary Key(AdminId)  -- 建立主键
)

```

在 Admin 表中，不少朋友想当然地用 FKUserId 去对应 User 表中的 UserId，以此来实现管理员与用户的对应关系。实际上，这是典型的一对多关系的做法。因为，你完全可以让两个不同的管理员拥有着相同的 FKUserId 字段值，也就是对应 User 表中的同一个用户。这时候，已经破坏了数据一致性。而且，还产生了一个没有任何意义的字段：AdminId。

那么，合理的方案是什么呢？那就是采用一对一关系，具体的实现脚本如下：

```

Create Table [User]
(
    UserId      Int Identity(1,1),          -- 主表的主键
    Name        Varchar(30)               Not Null,
    Password    Varchar(50)               Not Null,
    Email       Varchar(50)               Not Null,
)
Constraint pk_User Primary Key(UserId)
)
Create Table Admin
(
    UserId      Int,                      -- 不能是自增型
    AdminPwd    Varchar(50)               Null,
    Level       TinyInt                   Not Null Default 1 Constraint ck_UserLevel
Check(Level between 1 and 2), -- 1,版主; 2,管理员
)
Constraint pk_Admin Primary Key(UserId)   -- 建立从表主键
)
Alter Table Admin -- 建立从表外键
Add Constraint fk_Admin_User Foreign Key(UserId) References User (UserId)
On Delete No Action On Update No Action

```

我们看看现在与之前有什么区别？

1. AdminId 字段重命名为了 UserId，为什么这么做？我的原则是：如果一个表中的字段与另一个表中的字段含义相同，那么它们的名字也相同，外键包含的字段视情况而定。
2. UserId 字段不再是自增型了，这个很容易理解：它要与 User 表中的 UserId 对应，以表示是同一个人。User 表中的 UserId 已经是自增型了，它当然不能再自增了，不然如何对应？
3. 取消了 FKUserId 字段，原因很简单，因为 UserId 现在不光是主键，也是外键，用来对应 User 表的 UserId 字段。

在这个实例中：

- UserId 是主键，保证了在 Admin 表中它只能有一个，不能重复。
- UserId 是外键，保证了它一定对应一条已经存在于 User 表中的记录，也就是一个用户。

我们现在总结一下，一对一关系实施的特点：

1. 主表的主键为自增型或者其他任何能唯一标识记录的类型。
2. 从表的主键(包含的字段)与主表的主键(包含的字段)名称与含义相同。
3. 从表的外键(包含的字段)与从表的主键(包含的字段)相同。
4. 从表的外键(包含的字段)对应主表的主键(包含的字段)。

## 面向对象基础

### 类型

个人认为，类型(Type)是面向对象思想中最重要的一个概念，许多其他的概念都是由它推演出来的。而很多朋友以为定义了一个类(Class)就是定义了一个类型，这样的理解有所偏差。

类型是由类的接口决定的，当我们提到某个对象是什么类型的时候，实际上说的是它有什么样的接口，而并不关心其接口是如何实现的。反过来思考，就是：类型是详细定义了、对象所要支持的接口。

**NOTE:**如果一个对象实现了类型定义的全部接口，那么我们就说：这个对象实现了这个类型，说得顺口一点，就是：某某类型的对象。如果一个对象实现了很多个类型的接口，那么它就同时是很多个类型。

GOF 四人组在其著作《设计模式》第 13 页中这样提到：A type is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operation defined in the interface "Window".

翻译过来是：类型是标识一种特定接口的名称。如果有一个对象，它满足定义在"Window"接口中的全部要求，那么我们就说这个对象具有"Window"类型。

### 联系

类型并不是孤立存在的，一个类型可以与其他类型存在联系。如果两个类型之间存在着联系，那么某个类型的对象就可以转换为另一个类型的对象。

**NOTE:**大家整天在说面向对象的多态性，奇怪为什么一个类的实例可以转换成它的接口类型，而接口什么都没有做，只不过定义了一堆方法和属性签名而已。

现在明白了吧？这是因为类型间的联系，接口定义了类型，而类(class)与之联系。

### 类

类定义了对象(反过来思考，就是类的实例)的具体实现。它定义了对象拥有什么样的类型，说得通俗一点，就是类实现了接口(抽象类是个特例)。

### 继承

继承可以是类型继承，也可以是类继承。

当是类型继承的时候：如果一个对象是类型 B，而类型 B 继承自类型 A，那么我们可以说：此对象也是类型 A。

当是类继承的时候：我们说一个类继承了另一个类的实现，同时，继承类并不丧失对其基类的覆盖能力。

## 状态

状态这一概念相对来说比较好理解，它通常指的是对象拥有的值的集合。

## 行为

对象的行为是指：

- 一个对象所提供的操作的集合，也就是它的接口。
- 对象对于调用它的其他对象所做出的回应，通常来说就是方法的返回值。
- 操作对于对象本身所造成的影响。

对于一个对象的所有互动和了解都是通过它的行为来完成。

## 封装

封装使得对象成为一个不依赖于其他对象的独立个体。外界对其内部的实现是不可见的。举个例子，当你与一个封装好了的对象进行交互，你只能通过它所提供的接口，而对于接口以内的具体实现是不能也不应该知道的。

**NOTE:** 本章仅为后面章节的说明提供预备知识，内容比较浅显，如果您对面向对象程序设计感兴趣，可以参考相关书籍。

## 关系基础

本章讲述的是关系(Relation)，尽管有的概念与之前面向对象的概念有几分相似，但请不要试图用面向对象的思想去理解关系中的概念。

## 关系

关系定义了一组属性，这组属性组合起来构成一个有意义的事物，以我们之前讨论过的一对一关系为例，它之中的一个关系就是：User {Name, Email, Gender, BirthDay}。

## 属性

属性是关系的组成部分，它提供了区分不同关系的依据。举个例子，为什么用户关系 User {Name, Email, Gender, BirthDay} 和 汽车关系 Car {Name, Color, Speed, Price} 不同？不是因为“User”的意思翻译过来是“用户”，“Car”翻译过来是“汽车”，而是因为它们的属性不同。

属性还定义了属性值的范围。例如，User 关系的属性 Gender (性别)，它的取值范围就只能可能是{男, 女}，不可能再有其它值。

## 值域

简单地理解，值域就是一种数据类型，或者说，是值的一个取值范围。例如上面提到的 Gender 属性，它的值域就是{男, 女}。

## 元组

就如用说“汽车”一样，关系是在很高层次上的一个抽象，并不代表具体的事物。只有当关系的所有属性都有了其值域内的特定值时，它才能代表一个实际的事物。

当我们给一个关系的属性都赋予了其值域内的某个特定值以后，就构成了一个元组。

我们再以上面的 User 关系为例，它的一个元组是：

```
<User Name="张子阳" Gender="男" Email="jimmy_dev@163.com" BirthDay="1982-12-8">
```

对于多个元组来说，如果它们的所有属性值都相同，那么我们就说它们是同一个元组。

## 关系值

一组元组就构成了关系值，我们再以 User 为例，它的一个关系值是：

```
{  
<User Name="张子阳" Gender="男" Email="jimmy_dev@163.com" BirthDay="1982-12-8">  
<User Name="王五" Gender="男" Email="Wang5@cnblogs.com" BirthDay="1983-7-17">  
<User Name="李四" Gender="女" Email="Li4@cnblogs.com" BirthDay="1984-5-20">  
}
```

对于关系值来说：

- 不存在重复的元组。
- 元组的先后顺序是无关紧要的。

## 关系和表

在关系数据库中，关系通常表现为表的形式，例如上面的 User 关系，通常在数据库中表达成一张类似下面的 User 表。

User 表

Name	Gender	Email	BirthDay
张子阳	男	jimmy_dev@163.com	1982-12-8
李四	女	li4@cnblogs.com	1983-7-17
王五	男	wang5@cnblogs.com	1984-5-20

## 关系系统和数据库系统的术语对应

数据库系统	关系系统
表(Table)	关系(Relation)
行(Row)	元组(Tuple)
列(Column)	属性(Attribute)
数据类型(Data Type)	值域(Domain)
列值(Column Value)	属性值(Attribute Value)

## 基关系

现在请回想一下本文第一章所讲述的一对一关系,在前面的范例中,相对于 Admin 关系来说, User 关系就是一个基关系。

## 继承关系

子关系是相对于基关系而言的,相对于 User 来说, Admin 关系就是一个继承关系。

**NOTE:** 本章仅为后面章节的说明提供预备知识,内容比较浅显,如果您对关系系统感兴趣,可以参考《离散数学》和《关系数据库系统概论》。

## 对象 vs 关系

如果我们现在要对“汽车”这一概念进行建模,它有四个属性: Name(车名)、Head(车头)、Body(车身)、Rear(车尾)。

## 面向对象的做法

我们首先定义一个 Car 类:

```
public class Car{
    public string Name;
    public string Head;
    public string Body;
    public string Rear;

    public Car(name, head, body, rear){
        this.Name = name;
        this.Head = head;
        this.Body = body;
        this.Rear = rear;
    }
}
```

当我们需要一辆宝马(BMW)、一辆奔驰(MB)、一辆奥迪(Audi)的时候:

```
Car BMW = new Car("宝马", "宝马车头", "宝马车身", "宝马车尾");
Car MB = new Car("奔驰", "奔驰车头", "奔驰车身", "奔驰车尾");
Car Audi = new Car("奥迪", "奥迪车头", "奥迪车身", "奥迪车尾");
```

**NOTE:** 上面这个例子实际并不合适,你可以将 CarHead、CarBody 想象成还拥有各自属性的实体类。

## 关系数据库的做法

建四张表,分别为: CarName、CarHead、CarBody、CarRear,然后将三辆车分成四个部分,将每个部分分别放置到合适的表中,需要用的时候,再进行组装。

## 问题提出

如果仅仅只是如此而已,问题似乎很容易解决:让类的属性去对应数据库中表的字段。

现在,请回想之前面向对象基础那一节的内容,对象不仅仅只有属性而已,还包括 行为、

封装、继承、联系 等复杂特性。它们并不能够在数据库中找到良好的对应，这样的问题该如何解决呢？

限于篇幅，本文就介绍的这里，更多的内容将在后续文章中讲述。

## 总结

在本文中，我们首先了解了什么是一对一关系 以及 如何实现一对一关系。

随后我们对 面向对象 和 关系系统 的基础概念和理论做了复习和回顾。

最后，我们结合之前介绍的知识，提出了在软件开发中面向对象与关系系统之间存在的差异。

希望本文能给你带来帮助。