

# 数据库建模1

---

- 数据模型的组成和抽象过程
- 对象定义语言ODL
  - 基本ODL和高级ODL
  - 设计原则
- 关系模型
  - 基本概念
  - ODL到关系模型的转换

# 数据模型的组成要素

◆ **数据模型** 是描述客观事物特征的模型，包括事物本身的特性和事物之间的联系。

数据模型通常由数据结构、数据操作和数据的约束条件三个要素组成。

## ◆ **数据结构**

数据结构用于描述系统的静态特性。

数据结构是所研究的对象的特性的集合，它是刻画一个数据模型性质最重要的方面。

数据结构有层次结构、网状结构、关系结构和面向对象的数据结构四种类型，按照这四种结构命名的数据模型分别称为层次模型、网状模型、关系模型和面向对象模型。



# 数据模型的组成要素

◆ **数据操作** 数据操作用于描述系统的动态特性。数据操作是对数据库中各种数据操作的集合，包括操作及相应的操作规则。

数据模型必须定义操作的确切含义、操作规则以及实现操作的语言。

◆ **数据的约束条件** 数据的约束条件是一组完整性规则的集合。

**完整性规则**是给定的数据模型中，数据及其联系所具有的制约和依存规则，用以限定符合数据模型的数据库中的数据，以保证数据的正确、有效、相容和一致性。

数据模型应提供定义完整性约束条件的机制，反映在具体应用中，就是所涉及的数据必须遵守的特定的语义约束条件。



# 数据的抽象

---

由于计算机不能直接处理现实世界中的具体事物，所以人们必须将具体事物转换成计算机能够处理的数据。

在数据库中，用**数据模型**来抽象、表示和处理现实世界中的数据。

数据库即是模拟现实世界中某应用环境（一个企业、单位或部门）所涉及的数据的集合，它不仅要反映数据本身的内容，而且要反映数据之间的**联系**。



# 数据的抽象

为了把现实世界中的具体事物抽象、组织为某一DBMS支持的数据模型，在实际的数据处理过程中，

- 首先将现实世界的事物及联系抽象成信息世界的信息模型；
- 然后再抽象成计算机世界的数据模型。

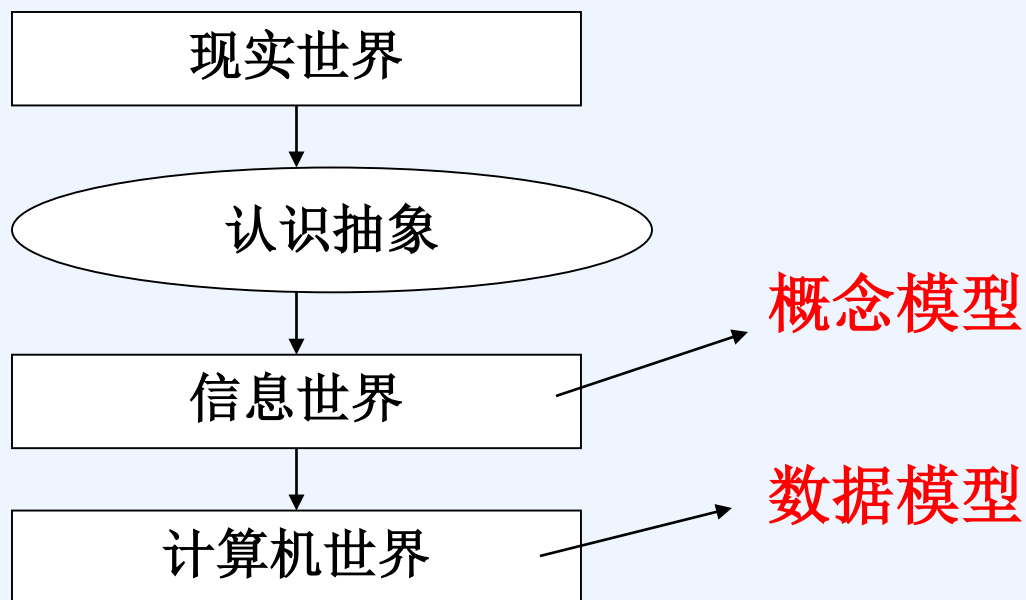
信息模型并不依赖于具体的计算机系统，不是某一个DBMS所支持的数据模型，它是计算机内部数据的抽象表示，是**概念模型**。

概念模型经过抽象，转换成计算机上某一DBMS支持的**数据模型**。所以说，数据模型是现实世界的两级抽象的结果。



# 数据的抽象

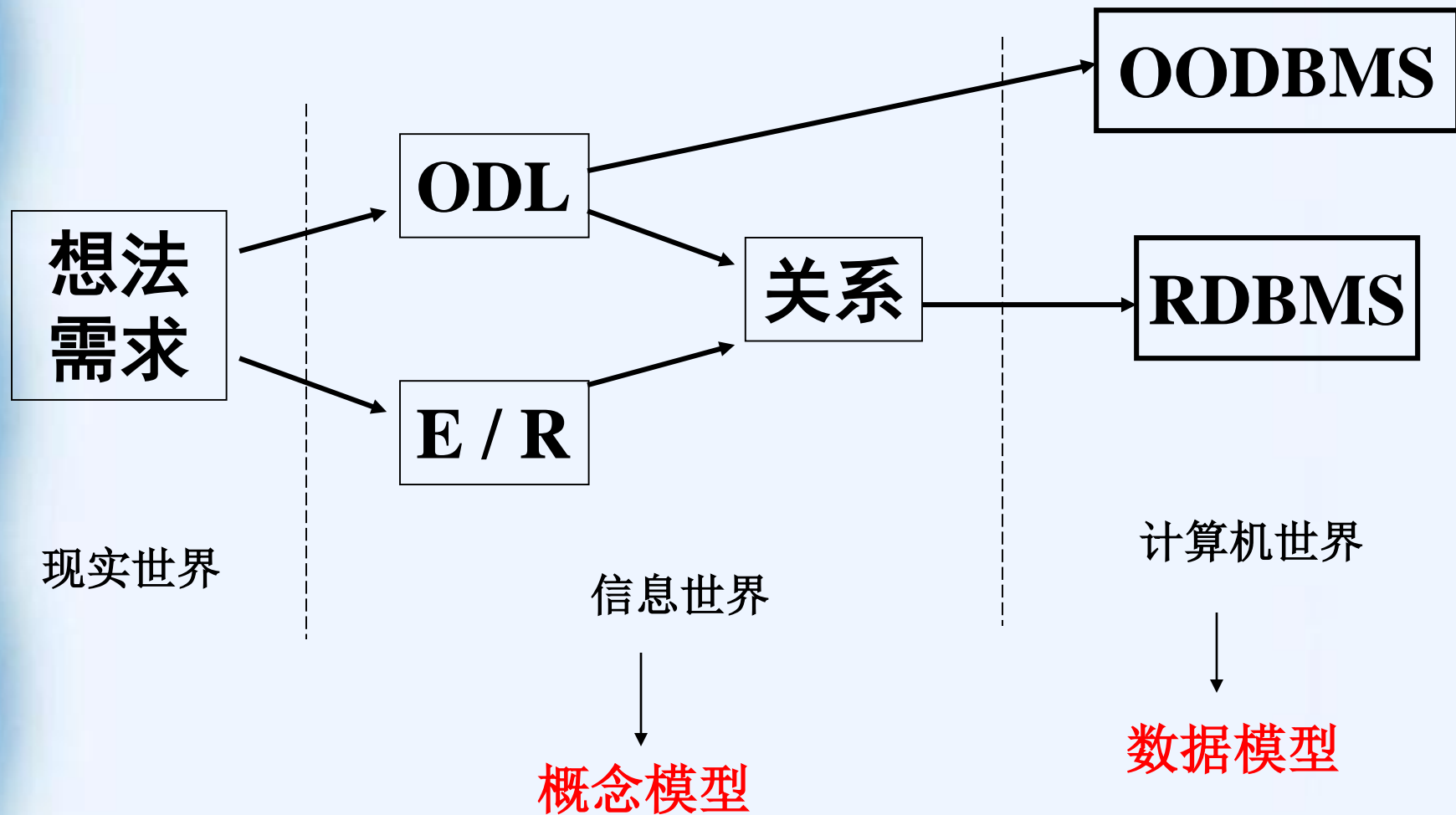
在数据处理中，数据加工经历了现实世界、信息世界和计算机世界三个不同的世界，经历了两级抽象和转换。这一过程如图所示。



数据处理的抽象和转换过程



# 数据的抽象



# 对象定义语言

**ODL(Object Definition Language,对象定义语言)**,是用面向对象的术语说明数据库中数据的结构的一种标准语言。

**ODL**的主要用途是书写面向对象数据库的设计,进而将其直接转换成面向对象数据库管理系统(**OODBMS**)中的数据结构。

在面向对象的设计中,把准备建立数据库模型的世界看作由**对象**组成,而对象是某种可观察的实体。假定对象有唯一的**对象标识**(**OID,Object IDentity**),各个不同的对象之间通过对象标识来区别。





# 对象定义语言 例

---

```
interface Movie{    //Movie 类的ODL说明
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film
        {color,blackAndwhite} filmType;
}
```



# ODL的类

class

entity sets

ODL中以**类**作为一个信息单元，类似于E/R图中的实体集，也就是关系模型中的关系(表)。

为了组织信息，将具有相似特性的对象归为一**类**。相似特性是指：

- ◆属于同一类的对象所表示的现实世界在概念上应该是类似的。也就是说，属于同一类的对象在现实世界中应属于同一概念。
- ◆属于一类的对象其特性必须相同，即具有相同的属性。




# ODL的类

在ODL中，形式最简单的类的说明应包括：

- 关键字**interface** 表示类说明的开始。
- 类的名字 表示某种具有共同特性的对象的集合的变量。
- 用花括号括起来的类的特性表(包括属性和联系)

```
interface 类的名字 {  
    <特性表>;  
}
```

 在不同的类中，属性的定义可以用相同名称，但却具有不同的类型。



# 对象定义语言 例

```
interface Movie{    //Movie类的ODL说明  
    attribute string title;  
    attribute integer year;  
    attribute integer length;  
    attribute enum Film  
        {color,blackAndwhite} filmType;  
}
```

枚举



# ODL中的数据类型

在ODL中，属性和联系的说明需要提供数据类型，ODL中数据类型包括基本类型、聚集类型和结构。

◆ **基本类型** 包括两大类：

- **原子类型** 整型int、浮点型float、字符char、字符串string、布尔型bool和枚举型enum。

布尔型的取值是“真”或“假”。

枚举型是名字的列表，它和整型是同义词。

在类Movie的说明中就有一个枚举型的属性filmType，效果上就是把名字color和blackAndwhite定义成整数0和1的同义词。

- **接口类型** 即定义好的某个类。

例如定义了类Movie后，Movie就代表了一个接口类型。我们把类也看作是基本类型。



# ODL中的数据类型

◆ **聚集类型**: 指通过以下四种类型构造符将基本类型组合而成的新的数据类型。(假设T是任意一种基本类型):

- **集合 Set<T>**表示由类型为T的元素构成的集合。元素无序、不重复。
- **包 Bag<T>**表示由类型为T的元素构成的包或多重集。元素无序、可重复。
- **列表 List<T>**表示类型为T的元素构成的有序列表。元素有序、可重复。string类型可以看作是List<char>类型的简化形式。

例 {1,2,1}和{2,1,1}不是集合，是相同的包，不同的列表。

- **数组** 如果i是一个整数，则**Array<T,i>**表示由i个类型为T的元素组成的数组。

例 Array<char,10>



# ODL中的数据类型

◆ **结构** 如果 $T_1, T_2, \dots, T_n$ 是类型,  $F_1, F_2, \dots, F_n$ 是域名, 则

$$\text{Struct } N \{ T_1 F_1, T_2 F_2, \dots, T_n F_n \}$$

表示名为 $N$ 的一个结构, 是由 $n$ 个类型分别为 $T_1, T_2, \dots, T_n$ 的元素构成的, 第 $i$ 个域名是 $F_i$ , 类型是 $T_i$ 。

**Struct**也是一种类型构造符。



# ODL中的数据类型的规则

## ◆ 属性的数据类型

- 原子类型

类型构造符指set, bag, list等

- 类型构造符**单次**应用于**原子类型**或**结构**所构成的聚集类型或**结构**

- **不能是接口类型**

## ◆ 联系的数据类型

- **接口类型**

如set{actors}

- 类型构造符**单次**应用于**接口类型**所构成的聚集类型

- **不能是原子类型**

- **不能是结构**

! 联系和属性的数据类型都**不能是聚集类型的两次应用。**





# ODL中的数据类型 例

? 判断下面数据类型应用于属性是否合法:

- integer ✓
- Struct N{string field1, integer field2} ✓
- List<real> ✓  
Struct N{struct 1{integer int1, integer int2}, struct 2{float fl1, float fl2}}是合法的
- Array<Struct N {string field1, integer field2}> ✓
- List<Movie> (假设Movie是已经定义好的类) ✗

? 判断下面数据类型应用于联系是否合法:

- Struct N{Movie field1, Star field2} ✓
- Set<integer> ✗
- Set<Array<Star>> (假设Star是已经定义好的类) ✗

不允许两次聚集



# ODL的属性

- ◆属性：利用某个数据类型的值来描述对象的某个方面的特征。
- ◆属性说明的四个基本要素：
  - 关键字`attribute` 表示属性说明的开始。
  - 属性的数据类型 表示属性的取值空间。
  - 属性的名字 表示描述对象某方面特征的变量。
  - 分号 `;` 表示属性说明的结束。



# ODL的属性 例Movie和Star

interface Movie{

原子  
类型

attribute string title; 属性名

attribute integer year;

attribute integer length;

attribute enum Film

{color,blackAndwhite}filmType;

}

interface Star {

attribute string name;

attribute Struct Addr 类型名 属性名

{string street,string city} address;

}



# ODL的属性 例

为某公司设计数据库，要求保存以下信息：

雇员：雇员号、姓名、年龄、地址、所在部门。

```
interface employee{  
    attribute      int      employeeID;  
    attribute      string   name;  
    attribute      int      age;  
    attribute      string   address;  
    attribute      string   department;  
}
```



# 联系与反向联系

◆联系与反向联系：联系是描述类的一个对象与同一类或不同类的其他对象之间的关系特性。联系与反向联系总是成对出现的。

◆联系与反向联系说明的基本要素包括：

- 关键字 **relationship** 表示联系说明的开始。
- 联系的数据类型 表示该关系特性取值的域。
- 联系的名字 表示该关系特性的名字。
- 关键字 **inverse** 表示该联系对应的反向联系。

对应interface的名字

- 类名::反向联系的名字 反向联系的名字之前加上类的名字以及两个冒号::表示引用。



这里的类名与定义该关系特性取值的域中的类名一致。

- 分号 ; 表示联系与反向联系说明的结束。



# ODL的联系 例

记录电影的影星。

```
interface Movie {  
    attribute string    title;  
    attribute integer   year;  
    attribute integer   length;  
    attribute enum Film {color,blackAndWhite} filmType;  
    relationship Set<Star> stars 联系名  
        inverse Star::starredIn 反向联系名  
}  
  
interface Star {  
    attribute string    name;  
    attribute Struct    Addr  
        {string street,string city} address;  
    relationship Set<Movie> starredIn  
        inverse Movie::stars;  
}
```



# 联系与反向联系

◆联系的规则：如果类C的联系R把类C的对象x和类D的一个对象或多个对象 $y_1, y_2, \dots, y_n$ 的集合相连，

那么R的反向联系，即类D中对应的联系，就把类D的每个对象 $y_i$ 和类C的对象x(或还有其他的对象)相连。

◆注：联系中C和D也可以同类，即一个联系可以在逻辑上连接自己。




# 自身的联系 例

设计一个家谱数据库中的类**Person**，要求：

- 记录每个人的姓名。
- 记录血缘关系的联系：母亲、父亲和孩子。

对这个数据库用**ODL**进行描述。

思考：需要几对联系和反向联系？





# 自身的联系

**interface Person {**  
    **attribute string name;**  
    **relationship Person motherOf**  
        **inverse Person::childrenOfFemale ;**  
    **relationship Person fatherOf**  
        **inverse Person::childrenOfMale ;**  
    **relationship Set<Person> children**  
        **inverse Person::parentsOf ;**  
    **relationship Set<Person> childrenOfFemale**  
        **inverse Person::motherOf ;**  
    **relationship Set<Person> childrenOfMale**  
        **inverse Person::fatherOf ;**  
    **relationship Set<Person> parentsOf**  
        **inverse Person::children ;**  
**}**



# 联系的多重性

◆联系及其反向联系的唯一性要求称为**联系的多重性**。

◆联系的三种类型：

- 从类C到类D的**多对多联系** 在联系中，至少C中有一个对象和D的对象的集合有关，而在反向联系中，至少D中有一个对象和C的对象的集合有关。
- 从类C到类D的**多对一联系** 在联系中，每个C的对象都和唯一的D的对象有关，而在反向联系中，至少D中有一个对象和C的对象的集合有关。
- 从类C到类D的**一对一联系** 在联系中，每个C的对象都和唯一的D的对象有关，而在反向联系中，每个D的对象都和唯一的C的对象有关。



# 联系的多重性 例

引入类Studio的说明，包含了制片公司与电影之间的联系。  
interface Studio{

attribute string name;

attribute string address;

**relationship** Set<Movie> owns **inverse** Movie::ownedBy;  
}

这样，相应的类Movie也要作相应的改动：

interface Movie{

attribute string title;

attribute integer year;

attribute integer length;

attribute enum Film{color,blackAndwhite} filmType;

relationship Set<Star> stars **inverse** Star::starredIn;

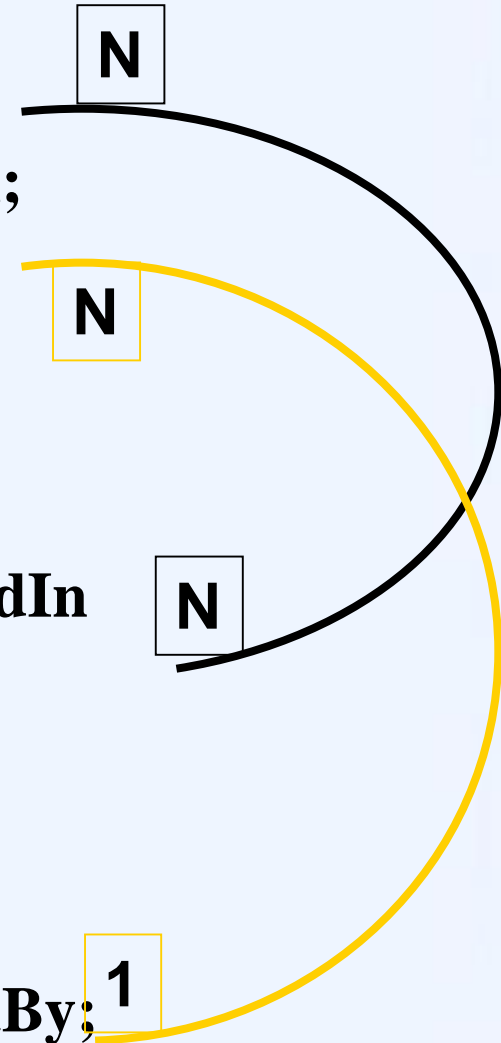
**relationship** Studio ownedby **inverse** Studio::owns;

}



# 多重性的说明 例

- interface Movie{  
.....  
relationship Set<Star> stars  
inverse Star::staredIn;  
relationship Studio ownedBy  
inverse Studio::owns;  
}
- interface Star{  
.....  
relationship Set<Movie> staredIn  
inverse Movie::stars;  
}
- interface Studio{  
.....  
relationship Set<Movie> owns  
inverse Movie::ownedBy;  
}



# ODL的联系 例 relationship

为某公司设计数据库，要求保存以下信息：

雇员：雇员号、姓名、年龄、地址、所在部门。

部门：名称、雇员、经理、所销售的商品。

商品：名称、制造商、价格、型号、编号。

制造商：名称、地址、商品名。

```
interface GuYuan{
```

```
attribute int  guyuanID;   attribute string name;
```

```
attribute int  age;   attribute string address;
```

```
relationship dep workIn inverse dep::myworker;
```

```
relationship dep headOf inverse dep::header;}
```

```
interface dep{
```

```
attribute string name;
```

```
relationship Set<GuYuan> myworker inverse GuYuan::workIn;
```

```
relationship GuYuan header inverse GuYuan::headOf;
```

```
relationship Set<shangP> forsale inverse shangP::toDep;}
```



# ODL的联系

雇员：雇员号、姓名、年龄、地址、所在部门。

部门：名称、雇员、经理、所销售的商品。

商品：名称、制造商、价格、型号、编号。

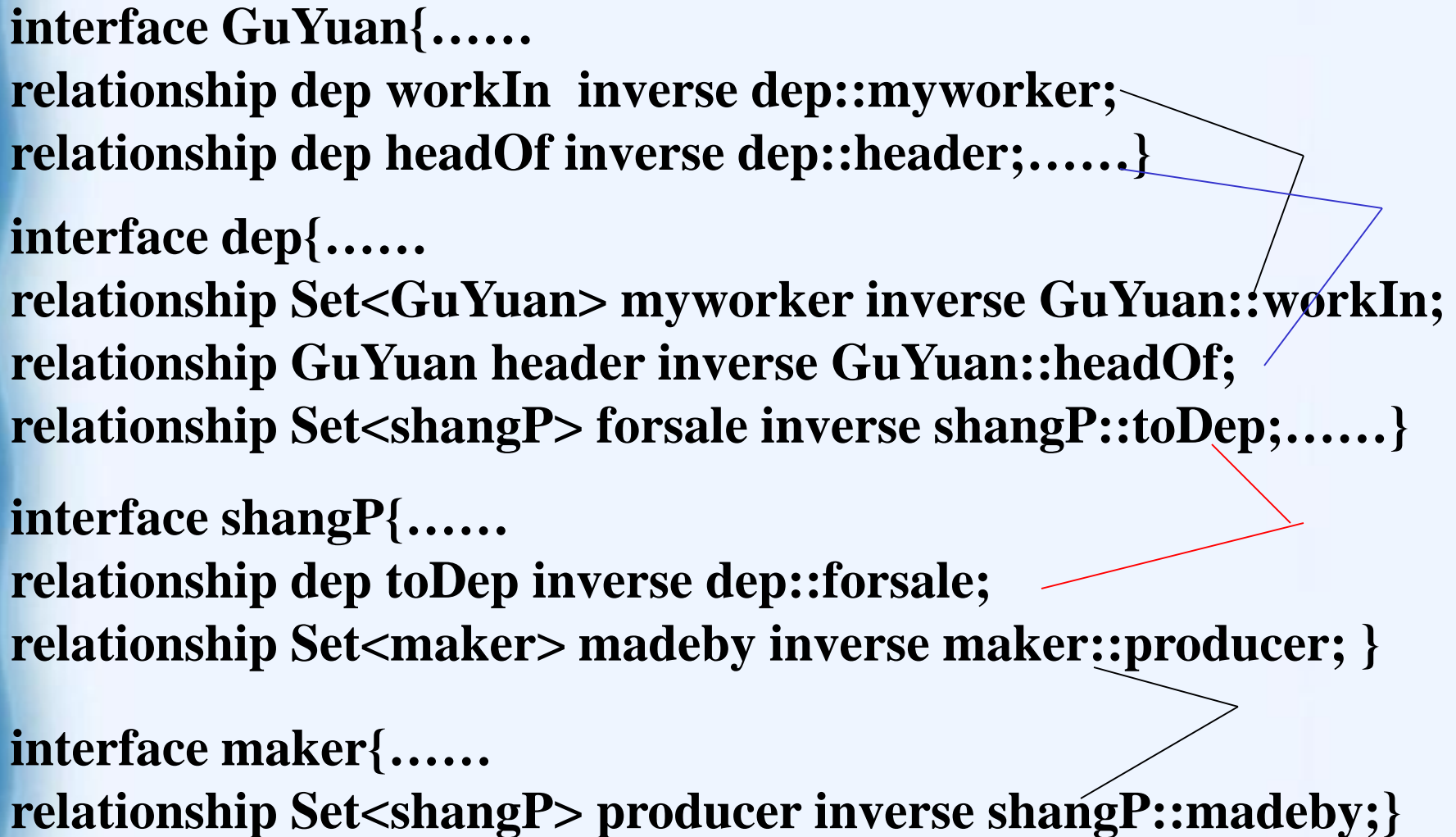
制造商：名称、地址、商品名。

```
interface shangP{  
  attribute string name;  
  attribute float price;  
  attribute string modelID;  
  attribute string shangpID;  
  relationship dep toDep inverse dep::forsale;  
  relationship Set<maker> madeby inverse maker::producer;}  
  
interface maker{  
  attribute string name;  
  attribute string address;  
  relationship Set<shangP> producer inverse shangP::madeby;}
```



# ODL的联系

```
interface GuYuan{.....  
relationship dep workIn inverse dep::myworker;  
relationship dep headOf inverse dep::header;.....}  
  
interface dep{.....  
relationship Set<GuYuan> myworker inverse GuYuan::workIn;  
relationship GuYuan header inverse GuYuan::headOf;  
relationship Set<shangP> forsale inverse shangP::toDep;.....}  
  
interface shangP{.....  
relationship dep toDep inverse dep::forsale;  
relationship Set<maker> madeby inverse maker::producer; }  
  
interface maker{.....  
relationship Set<shangP> producer inverse shangP::madeby;}
```



# 联系类型之间的内涵

! 多对一联系是多对多联系的特例，一对一联系是多对一联系的特例。

! 多对多的联系 $R$ ，指联系 $R$ 有多对多的自由度。随着对象的改变，有时它可能成为多对一的联系甚至一对一的联系。

同样地，多对一的联系 $R$ 有时可能成为一对一的联系。



# ODL中的子类

- **子类和超类** ODL中，如果某一类对象具有另外一类对象所有的特性，同时又有某些其他特性，此时该类为另外一类的子类。另外一类称为该类的超类。

- 子类的说明方法：

假设类C是类D的子类，则在类C的说明中，在类名C后面加上冒号和类D的名字。即

```
interface 子类名： 超类名 {  
    <特性表>;  
}
```



子类定义中的特性表中不包含超类中已含有的特性。



子类继承其超类的所有特性。



# ODL中的子类 例

假设要记录卡通片的配音演员。

**Cartoon**为**Movie**的子类，用以下的ODL说明：

```
interface Cartoon:Movie{  
    relationship Set<Star> voice inverse Star::voiceof;  
}
```

在**Star**中增加

```
relationship Set<Cartoon> voiceof inverse Cartoon::voice;
```



# 子类的多重继承

◆ODL的子类具有多重性，其含义包括：

- 一个类可以有多个子类。
- 子类可以有子类，形成类的层次，每个类都继承祖先的特性。
- 一个子类可以有多个超类。

说明格式：将这多个超类用逗号隔开放在冒号后。

```
interface 子类名： 超类名1,超类名2,.....{  
    <特性表>;  
}
```

? 这时子类的特性包含哪些？

如果超类1和超类2中有重名的属性，则在继承时应注意。



# 子类的多重继承 例

- Movie的一个子类Cartoon,

```
interface Cartoon:Movie{relationship Set<Star> voices...;}
```

- 再假设保存谋杀片信息,

```
interface Murder:Movie{attribute string Weapon;}
```

- 有的电影既是卡通片，又是谋杀片，则可以说明另一个子类Cartoon-Murder来描述这种情况，说明如下：

```
interface Cartoon-Murder:Cartoon,Murder {}
```

 类Cartoon-Murder的特性包含哪些？

包括Movie的所有特性，Cartoon中新增的一个联系，Murder中新增的一个属性。



# ODL中的子类 例

用ODL描述军舰数据库。

- 1、记录每艘军舰的信息，包括名称、排水量、类型。
- 2、记录下列特殊类型的军舰：
  - 炮舰：携带大型火炮的军舰，记录主炮的数量和口径。
  - 航空母舰：记录飞行甲板的长度和分派给它们的航空大队的集合。
  - 潜艇：记录最大安全深度。
  - 攻击型航空母舰：既是炮舰又是航空母舰。



# ODL中的子类

```
interface Ship {  
    attribute string name;  
    attribute float displacement;  
    attribute string type;}
```

军舰信息：名称、排水量、类型

```
interface Gunship: Ship {  
    attribute integer numberOfGuns;  
    attribute float bore;}
```

炮舰：记录主炮的数量和口径

```
interface Carrier: Ship {  
    attribute float deckLength;  
    attribute Set<string> airGroups;}
```

航空母舰：甲板的长度和航空大队的集合

```
interface Submarine: Ship {  
    attribute float maxSafeDepth;}
```

潜艇：最大安全深度

```
interface BattleCarrier: Gunship, Carrier {}
```

攻击型航空母舰

? 类BattleCarrier的特性包含哪些?



# ODL中键码的说明

## ◆ ODL中键码的说明要素包括:

- 圆括号
- 关键字key或keys
- 构成键码的属性或属性集列表

## ◆ 格式如下:

(keys (键码1之属性1,键码1之属性2,...),键码2之属性)

! 如果键码由多个属性组成，则属性表必须用括号括起来。

! 键码说明必须紧跟在接口名之后，在左花括号之前说明。

! 如果有多个键码，则各个键码之间用逗号隔开。



# ODL中键码的说明 例

1) **interface Movie (key (title,year))**{

**//接着为属性和联系表**

2) **(key empID,ssNo)**

**?**与**(key (empID,ssNo))**的含义是不同的。

3) 记录学生信息，并为其选择合适的键码。

```
interface Student(key studentID){  
    attribute string name;  
    attribute string studentID;  
    attribute integer age;  
    .....  
    }
```



# 设计原则

- **真实性** 数据库的设计应当忠于现实社会。
- **避免冗余** 任何事物只表达一次。冗余会带来危害：
  - 占用更多的空间。
  - 易造成数据不一致。
- **对简单性的考虑** 避免在设计中引入过多的不必要的元素，以节省空间，避免出错。
- **选择合适的元素类型**
  - 是使用**属性**还是使用**类**要根据实际情况确定。一般来说，属性比类或联系更为简单。但是，如果某个事物除名字信息外还有其他信息，就需要使用类。

# 关系模型的术语

❖ **关系模型**：以二维表的形式来表示数据的相关特性以及数据之间的联系的一种数据结构。

❖ **关系(表)**：关系模型中的每个二维表称为关系。关系是现实世界中的信息在数据库中的数据表示。表是含有数据库中所有数据的数据库对象。  
relation

❖ **元组**：表格中的一行，如学生表中的一个学生记录即为一个元组。

❖ **属性**：表格中的一列，相当于记录中的一个字段，如学生表中有五个属性(学号，姓名，性别，年龄，系别)。

❖ **键码**：可唯一标识元组的属性或属性集，如学生表中学号可以唯一确定一个学生，为学生关系的键码。

❖ **域**：属性的取值范围，通常通过属性的数据类型以及各种约束来实现。如学生表中年龄的域是(14~40),性别的域是(男、女)。domain



# 关系模型的术语

❖ **分量**：每一行对应的列的属性值，即元组中的一个属性值，如学号、姓名、年龄等均是一个分量。

❖ **关系模式**：对关系的描述，一般表示为：`relation schema`

关系名(属性1, 属性2, ....., 属性n)

其中加下划线表示这两个属性的集合构成键码。

如：学生(学号, 姓名, 性别, 年龄)。

在设计关系模型时，一般给出关系模式。

❖ **实例**：将给定关系中元组的集合称为该关系的实例。实例会随时间发生变化。

❖ **当前实例**：目前该关系中元组的集合。



# 关系模型的术语

关系名称：学生基本信息

键码

属性

学号	姓名	性别	年龄
197184013	张三	男	24
197184022	李四	女	25
100184013	王五	男	27

元组

关系模式为：

学生(学号, 姓名, 性别, 年龄)



# 关系数据库

❖ 关系具有如下特性：

- 关系中不允许出现相同的元组。

- 关系中元组的顺序(即行序)是无关紧要的，在一个关系中  
可以任意交换两行的次序。

根据关系的这个性质，可以改变元组的顺序使其具有某种  
排序，可以提高查询速度。

学号	姓名	生日	分数
200101184022	王涛	1983/3/9	84
200201184358	于栋	1982/10/15	95

A red curved arrow on the right side of the table points from the first data row to the second data row. Another red curved arrow below the table points from the second data row back to the first data row, indicating that the rows can be swapped.

- 关系中属性的顺序是无关紧要的，即列的顺序可以任意  
交换。交换时，应连同属性名一起交换。

- 同一属性名下的各个属性值必须来自同一个域，是同一  
类型的数据。



# 关系数据库

- 关系中各个属性必须有不同的名字，不同的属性可来自同一个域。

姓名	职业	兼职
张强	教师	辅导员
王丽	工人	教师
刘宁	教师	辅导员

- 属性值可以为空值，表示“未知”或“不可使用”。
- 关系中每一分量必须是不可分的数据项，或者说所有属性值都是原子的，即是一个确定的值，而不是值的集合。

必须都是原子的

姓名	<del>籍贯</del>	
	省	市 / 县
张强	吉林	长春
王丽	山西	大同

姓名	省	市 / 县
张强	吉林	长春
王丽	山西	大同



# 关系模型的优缺点

## ❖ 关系模型的优点

- 它有较强的数学理论根据(关系代数)。
- 数据结构简单、清晰，用户易懂易用。
- 关系模型的存取路径对用户透明，从而具有更高的数据独立性、更好的安全保密性，也简化了程序员的工作和数据库建立和开发的工作。

## ❖ 关系模型的缺点

- 由于存取路径对用户透明，查询效率往往不太好，因此，为了提高性能，必须对用户的查询表示进行优化，增加了开发数据库管理系统的负担。

# 从ODL到关系模型

---

- ❖ 属性的转换
- ❖ 联系与反向联系的转换
- ❖ 子类的转换
- ❖ 键码的处理



# 属性的转换

- ❖ 首先，用与类名相同的名字建立一个关系名。
- ❖ ODL中的属性转换到关系模型时，需要根据数据类型来确定转换方式。
  - 原子类型的属性：每个属性对应于关系的一个属性，关系的属性名和相应类的属性名相同。
- 枚举类型实际上是前几个整数的别名列表。因此，枚举类型可以用整型的属性来表示。
  - 非原子类型的属性需要区别对待。

由于关系模型的属性要求具有原子类型，不能是聚集类型，因此对类中非原子类型的属性，必须经过转换才能对应关系中的属性。



# 非原子属性的转换

## 1、结构

结构的域本身都是原子类型，转换的方法：

- 将结构的每个域都转换成关系的一个属性。
- 如果两个结构有相同的域名，则必须在关系中使用新的属性名，以示区别。

**例** 考虑ODL模型

```
interface Star{  
    attribute string name;  
    attribute Struct Addr{string street,string city} address;}
```

转换成关系模式为Star(name,street,city)



# 非原子属性的转换

**2、集合** 如果ODL中属性A是集合，则转换到关系模型的表示方法是：

模型中的属性名称与ODL中一致，属性的数据类型也与ODL中一致，**只是ODL中的每个对象都会对应关系模型中的多个元组，元组的数量就是对应集合属性的取值数量。**

**使用这种方法由ODL转换成关系，如果除了属性A和键码外还有其他属性，则会产生冗余信息。**

**例** 假设有ODL如下：

```
interface Star{  
    attribute string name;  
    attribute Set<Struct Addr{string street,string city}> address;  
    attribute date birthdate;}
```

转换成关系模式为：Star(name,street,city,birthdate)



# 集合类型的转换 例

关系模式Star(name,street,city,birthdate),下图为该关系的一个实例:

name	street	city	birthdate
Carrie Fisher	123 Maple St.	Hollywood	9/9/77
Carrie Fisher	5 Locust Ln.	Malibu	9/9/77
Mark Hamill	456 Oak Rd.	Brentwood	8/9/76

ODL中的一个对象

显然, 其中的出生日期重复出现, 产生冗余。



# 非原子属性的转换

**3、包** 由于包中允许出现相同的元素，不能简单地在一个关系中引入n个相同的元组，

而要在关系模式中增加一个计数的属性count，用以表明每个元组中对应的属性是包的多少次成员。

例Bag<int>，则转换到关系模式中变成两个属性，原属性和count(为int类型)。

**4、列表** 列表可以增加一个新属性position(位置)来表示，用来表明在列表中的位置。

例List<int>，则转换到关系模式中变成两个属性：原属性和position(为int类型)。

**5、数组** 定长的数组可以用带有数组位置标志的属性表示。

例Array<Struct Addr{ string street,string city },2>则转换到关系模式中变成street1,city1,street2,city2。



# 联系的转换

❖ **对单值联系**，把该联系看成是一个属性，该属性由与之联系的类的键码属性或属性集合构成。

转换成关系模型时，用与之联系的类中构成键码的属性集来代替这个单值联系作为关系的属性。

❖ **对多值联系**，即当联系是某个类的聚集类型时，同样需要首先找出相关类的键码，然后与数据类型为集合的属性的处理方法一样，需要为每个值建立一个元组。

这种方法带来了冗余。因为对应于集合的每个成员，该关系的其他属性都将它们的值重复一次。



# 多值联系的转换

◆假设类C转换成关系模式，那么对应的关系中的属性构成包括：

- 类C的所有属性
- 类C中所有单值联系对应的类的键码属性
- 类C的多值联系对应的类的键码属性

◆有时，一个类可能有多个多值联系。在这种情况下，表示类中的一个对象所需的元组数会爆炸性增长。

假设类C有 $k$ 个多值联系 $R_1, R_2, \dots, R_k$ ，类C的某特定对象O通过联系 $R_1$ 与 $n_1$ 个对象相联，通过 $R_2$ 与 $n_2$ 个对象相联，.....。

那么在类C对应的关系中，共有 $n_1 \times n_2 \times \dots \times n_k$ 个元组对应于ODL中设计的对象O。



# 多值联系的转换 例

假设类C有一个单值属性X和两个多值联系 $R_1$ 和 $R_2$ ，这两个联系将类C分别与键码属性为Y和Z的两个类相联。

现在，考虑类C的一个对象c,假设对象c的属性集X的值为x，并且c通过联系 $R_1$ 与键码值分别为y1， y2的两个对象相联，通过 $R_2$ 与键码值分别为z1， z2， z3的三个对象相联。

于是，对象c在对应于类C的关系中需要用六个元组表示,也就是Y对应的键码与Z对应的键码的所有可能组合。

(x,y1,z1)	(x,y1,z2)	(x,y1,z3)	(x,y2,z1)
(x,y2,z2)	(x,y2,z3)		





# 多值联系的转换 例

```
interface GuYuan{  
  attribute int  guyuanID;   attribute string name;  
  attribute int  age;   attribute string address;  
  relationship dep workIn  inverse dep::myworker;  
  relationship dep headOf inverse dep::header;}
```

**GuYuan(guyuanID, name, age, address, depname1, depname2)**

```
interface dep{  
  attribute string name;  
  relationship Set<GuYuan> myworker inverse GuYuan::workIn;  
  relationship GuYuan header inverse GuYuan::headOf;  
  relationship Set<shangP> forsale inverse shangP::toDep;}
```

**dep(name, guyuanID, guyuanID2, shangpID)**



# 联系单向表示的选择原则

**!** ODL中联系与反向联系转换到关系模式时，二者只需选择一个进行转换，另一个则舍弃。其原则是

➤如果是多对多或一对一的联系，两个类中任选一个进行该联系的转换都可以。

➤如果是多对一的联系，则选择包括“多”的联系的类，即该类中的多个对象对应于另一类的一个对象。这样做可以避免冗余。



# 联系的转换 例

```
interface GuYuan{  
  attribute int  guyuanID;   attribute string name;  
  attribute int  age;   attribute string address;  
  relationship dep workIn inverse dep::myworker;  
  relationship dep headOf inverse dep::header;}  

```

```
GuYuan(guyuanID, name, age, address, depname1, depname2)
```

```
interface dep{  
  attribute string name;  
  relationship Set<GuYuan> myworker inverse GuYuan::workIn;  
  relationship GuYuan header inverse GuYuan::headOf;  
  relationship Set<shangP> forsale inverse shangP::toDep;}  

```

```
dep(name, guyuanID, guyuanID2, shangpID)
```



# 联系的转换 例

~~dep(name, guyuanID, guyuanID2, shangpID)~~

```
interface shangP{  
  attribute string name;  
  attribute float price;  
  attribute string modelID;  
  attribute string shangpID;  
  relationship dep toDep inverse dep::forsale;  
  relationship Set<maker> madeby inverse maker::producer; }
```

shangP(name, price, modelID, shangpID, depname, makername)

```
interface maker{  
  attribute string name;  
  attribute string address;  
  relationship Set<shangP> producer inverse shangP::madeby;}
```

maker(name, address)



# 联系的转换 例

```
interface Movie (key (title,year)){  
    attribute string title;  
    attribute integer year;  
    attribute integer length;  
    attribute enum Film {color,blackAndWhite} filmType;  
    relationship Set<Star> stars inverse Star::starredIn;  
    relationship Studio ownedBy inverse Studio::owns;}  
interface Studio(key name){  
    attribute string name;  
    attribute string address;  
    relationship Set<Movie> owns inverse Movie::ownedBy;}  
interface Star (key name){  
    attribute string name;  
    attribute Struct Addr {string street,string city} address;  
    relationship Set<Movie> starredIn inverse Movie::stars;}
```

Movie(title,year,length,filmType,studioName,starName)

Studio(name,address)

Star(name,street,city)

Star(name,street,city,title,year)



# 联系的转换

下面是Movie关系的一个实例

title	year	length	filmType	studioName	starName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	EmilioEstevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

用Studio的键  
码来代替单值  
联系ownedBy

用Star的键码  
来代替多值联  
系stars



# ODL子类到关系的转换

---

将ODL子类转换成关系模式应该遵循下面的原则：

- 为每个子类建立自己的关系。
- 这个关系应该包括子类自己定义的特性以及它从超类继承的全部特性。



# 子类到关系的转换 例

假设除了类Movie外，还有以下三个类，

```
interface Cartoon:Movie{relationship Set<Star> voices...;}
```

```
interface Murder:Movie{attribute string Weapon;}
```

```
interface Cartoon-Murder:Cartoon,Murder {}
```

将这四个类转换成关系，这四个关系的模式如下：

**Movie(title,year,length,filmType,studioName,starName)**

**Cartoon(title,year,length,filmType,studioName,starName,voice)**

**Murder(title,year,length,filmType,studioName,starName,weapon)**

**Cartoon-Murder(title,year,length,filmType,studioName,starName,voice,weapon)**





# ODL子类到关系的转换

■解决办法：如果在元组中允许使用NULL值，就可以用单个关系来表示类的分层结构，

该关系拥有的属性对应于子类的分层结构中所有的特性，包括子类的和超类的。

如果不是这个对象的类所拥有的特性，则该元组对应的属性就取值为NULL。



# 用NULL值合并关系 例

直接从ODL转换到关系时，得到

**Movie(title,year,length,filmType,studioName,starName)**

**Cartoon(title,year,length,filmType,studioName,starName,  
voice)**

**Murder(title,year,length,filmType,studioName,  
starName,weapon)**

**Cartoon-Murder(title,year,length,filmType,studioName,  
starName,voice,weapon)**

利用NULL来合并，则得到一个关系，其模式为：

**Movie(title,year,length, filmType,  
studioName,starName,voice,weapon)**

这种方法使得我们可以在一个关系中查找来自分层结构中所有类的一个对象的所有信息。

# 由ODL导出的关系的键码

由ODL转换成关系时键码的处理：

❖ 在ODL中没有定义键码并且无法断定哪些属性可以作为键码时，必须在相应的关系中引入一个属性，代替该类对象中的对象标识。如果在ODL中定义了键码，则直接继续下一步。

❖ 转换后关系的键码需要考虑该类的联系的情况。

➤ 假设类C指向某个类D的联系R是一对一或者多对一，C的键码依然是相应关系的键码。

➤ 假设类C指向类D的联系R是“多对多”的联系，则需要把D的键码加入到C的键码中，来组成关系的键码。如果类C有多个多对多联系，则所有这些多值联系所连接的类的键码都必须加入到转换后的关系C的键码中，得到的结果才是C对应的关系的键码。



## 键码的转换 例

```
interface Movie {attribute string title;attribute integer year;
    attribute integer length;
    attribute enum Film { color, blackAndWhite } filmType;
    relationship Set < Star > stars inverse Star::starredIn; }
interface Star {attribute string name;
    attribute Struct Addr { string street,string city } address;
    relationship Set<Movie> starredIn inverse Movie::stars;
```

假设在ODL定义中，不能肯定Name是影星的键码，则在转换到关系时，可以建立一个属性“cert#”(证书号),以此作为Star的键码，则对应的Star关系模式为

Star(cert#,name,street,city,title,year)

同时，由于Star和Movie间存在着多对多联系，则对应的Movie关系的模式为

Movie(title,year,length,filmType,~~cert#~~)



# 键码的转换 例

! 对于自身到自身的联系，转换后的关系需要为每一次自身联系提供一次键码(此时需要经过改名)。

```
interface Person (key name){  
attribute string name;  
relationship Person motherOf inverse Person::childrenOfFemale;  
relationship Person fatherOf inverse Person::childrenOfMale;  
relationship Set<Person> children inverse Person::parentsOf;  
relationship Set<Person> childrenOfFemale inverse  
Person::motherOf;  
relationship Set<Person> childrenOfMale inverse  
Person::fatherOf;  
relationship Set<Person> parentsOf inverse Person::children;}  
Person(name,nameofmother,nameoffather,nameofchildren)
```

name1,nameofchildren1

name2,nameofchildren2



# ODL转换成关系模式1

```
interface Customer (key ssNo) {attribute string name;  
    attribute string addr;  
    attribute string phone;  
    attribute integer ssNo;  
    relationship Set<Account> ownsAccts  
        inverse Account::ownedBy;}
```

```
interface Account (key number) {attribute integer number;  
    attribute string type;  
    attribute real balance;  
    relationship Customer ownedBy  
        inverse Customer::ownsAccts;}
```

<p><b>Customer(ssNo, name, address, phone)</b> <b>Account(number, type, balance, ssNo)</b></p>
--



# ODL转换成关系模式2

```
interface Ship (key name) {attribute string name;  
    attribute integer displacement;  attribute string type;}  
interface Gunship:Ship {attribute integer numberOfGuns;  
    attribute integer bore;}  
    interface Carrier:Ship {attribute integer deckLength;  
        attribute Set<string> airGroups;}  
interface Submarine:Ship {attribute integer maxSafeDepth;}  
interface BattleCarrier:Gunship,Carrier {}  
Ship(name, displacement, type)  
Gunship(name, displacement, type, numberOfGuns, bore)  
Carrier(name, displacement, type, deckLength, airGroups)  
Submarine(name, displacement, type, maxSafeDepth)  
BattleCarrier(name, displacement, type, numberOfGuns,  
bore,deckLength, airGroups)  
allship(name, displacement, type, numberOfGuns,  
bore,deckLength, airGroup , maxSafeDepth)
```

