17. OR

从关系到对象关系：

- 支持更加复杂的属性类型（如嵌套关系）；
- 允许声明方法；
- 为元组增加了编号 Tuple ID，默认隐藏但是可对用户显示；
- 使用指向元组的指针进行引用。

嵌套关系

- 递归定义：一个基础类型可以是属性的类型；一个关系的类型可以是任何模式，此外，模式同样可以作为关系属性的类型。比如可以定义如下关系：`R(A, B, C(C1, C2, C3))`。
- 在嵌套关系中，关于上例中 `A, B` 等属性的重复已经不再存在，然而仍有关于上例中 `C1, C2, C3` 的重复，我们可以使用**引用**进行消除。因此，属性的类型也可以是给定模式中一个元组（或元组集合）的引用，写作 `A(*R)` 或 `A({*R})`。

面向对象、对象关系、关系：

- 对象 vs 元组：
  - 在 OO 模型中，对象是一个包括属性与联系 (relationship) 的结构体；
  - 在关系模型中，对象仅包括属性，联系通过另一个关系中的元组表示；
  - 在 OR 模型中，对象是一个属性的结构体，联系可以用引用成员的形式表示。
- 方法：OR 允许声明方法。
- 类型系统：都基于原子系统以及结构体（集合）构造子； OR 支持结构体的集合，也就是嵌套关系。
- 指针与 Obeject ID
  - 在 OO 模型中，OID 对用户不可见；
  - 在 OR 模型中，指针引用的值对用户可见，同时可以被使用。
- 后向兼容性：
  - 纯关系数据库仍然可以在 OR 模型中运行，不能在 OO 模型中运行；
  - 从 RDB 转化为 OODB 仍然是很低效的事情。

用户定义类型 (User-Defined Types, UDT)：SQL 中的 UDT 相当于 ODL 中的类。

- 用法：
  - 作为关系中属性的类型；
  - 作为关系的类型（其元组的类型），与 OO 的 class 类类似，有时也被称为**行类型**。
- **UDT 的声明**：
  - 对已有的类型重命名，为了进行更强的类型检查。

    ```
    CREATE TYPE name AS primitive-type;
    ```

  - 类 class 的 UDT：

```
CREATE TYPE name AS (attr-declarations) [method declarations];
```

与 ODL 相比，

- 对于键值的声明成为了表的一部分，而不是类型的一部分；
- 联系并不是通过性质来表现，而是通过一个单独的关系（联系构造的关系）或引用体现。

- 例：在这里，使用 `HeightType+WeightType` 可能会引发错误——更强的类型检查。

```
CREATE TYPE HeightType AS INTEGER:
CREATE TYPE WeightType AS INTEGER;
CREATE TYPE AddressType AS (
  street_num CHAR(50),
  city CHAR(20)
);
CREATE TYPE Person AS (
  name CHAR(30),
  address AddressType
);
```

- **UDT 中的方法**：需要进行方法的声明以及一个独立的定义。
  - **声明**

```
CREATE TYPE udt-name AS (...)
  METHOD method-name(arg1 type1, arg2 type2, ...)
  RETURNS return-type;
```

  - **定义**：需要在声明之外单独定义，但可不需要与声明保持连续。

```
CREATE METHOD method-name(arg1, type1, arg2, type2, ...)
RETURNS return-type FOR udt-name
BEGIN ... END;
```

  - 例：

```
CREATE TYPE AddressType AS (
  street_num CHAR(50),
  city CHAR(20)
) METHOD getNum() RETURNS CHAR(10);

CREATE METHOD getNum() RETURNS CHAR(10) FOR AddressType
BEGIN
  ...
END;
```

- 使用 UDT 建立表：

```
CREATE TABLE table-name OF UDT-name [(list-of-elements)];
```

- 元组可以看成 UDT 对象；
- `list-of-elements` 中声明键值、外键、检查等等，这里定义的事物仅对于这张表，而不是针对 UDT 对象——与 ODL 的不同之处之一。

例：

```
CREATE TABLE people OF Person (PRIMARY KEY name);
```

Typed Table：一个声明基于一些结构体类型的表称为 typed table。

- 其列对应于结构体类型中的名称与类型；
- Typed table 有一个额外的列称为 **自引用列 (self-referencing column)**，其类型是指向构成当前表的结构体类型的指针。
- **子表** (subtable)：如果一个 typed table TB1 与一个结构体类型 TP1 相关联，而这个结构体类型是另一个结构体类型 TP2 的子类，那么 TB1 可以被看成 TP2 所生成的 typed table TB2 的子表；
- **超表** (supertable)：反之，TB2 称为 TB1 的超表。

引用列：UDT 表可以存在引用列，作为当前元组的身份证明，类似于 OID；那么这个表是可引用的。

- 引用列的构造：一般由 DBMS 构造与维护，或使用主键 PK 进行构造。
- **带引用列的表的声明**

```
CREATE TABLE tablename OF typename
  (REF IS attribute-name how-generated);
```

  - `how-generated` 可以是 `SYSTEM GENERATED` 表示系统自动生成维护，也可以是 `DERIVED` 即使用 PK 进行生成维护。
- 例：

```
CREAT TABLE people OF Person
  (REF IS pID SYSTEM GENERATED, PRIMARY KEY (name));
```

引用：对属性 A，如果其引用了 UDT T 类型的表 R，那么可以如下声明：

```
A REF(T) SCOPE R
```

- `R` 是被引用的关系；
- 如果没有声明作用域 scope，则默认可以是任意的 T 类型关系。
- 例：

```
CREATE TYPE Department AS (
  name CHAR(30),
  head REF(Person) SCOPE people
);
```

多对多的联系：可以使用引用表达多对多的联系，例如学生选课信息：

```
CREATE TABLE SC (
  sref REF(StudentType) SCOPE Students,
  cref REF(CourseType) SCOPE Courses
);
```

OR 数据库的操作：支持所有 SQL 的内容（可能有一些为支持 UDT 引入的新语法）以及一些新操作：

- **指针指向的内容**：如果 `x` 有 `RET(T)` 类型，那么 `x` 指向 `T` 类型的某个对象 `t`，使用 `DEREF(x)` 得到 `t`，也可以使用 `x -> a` 得到 `t` 中属性 `a` 的值，如：

  ```
  SELECT DEREF(cref) FROM SC
  WHERE sref -> name = 'James Bond';
  ```

- **访问 UDT 的属性**：UDT 为每个属性 `x` 隐式地定义了一个访问方法 `x()`，可以通过 `t.x()` 访问 `t` 中属性 `x` 元素的值。在实践中，通常可以省略 `()`，如：

  ```
  SELECT s.name() FROM Students s WHERE s.sno() = '007';
  SELECT s.name FROM Students s WHERE s.sno = '007';
  ```

  注意这里元组的名称 `s` 是必要的。

- **生成器** (generator)：一个 UDT `T` 定义时，自动定义方法：生成器，即使用 `T()` 返回一个 `T` 类型的不包含元素的空对象；

  变异器 (mutator)：一个 UDT `T` 定义时，自动定义方法：编译器，即使用 `t.x(v)` 将 `t` 元组的 `x` 属性设置为 `v`。

  例如：

  ```
  DECLARE newPerson PersonType;
  SET newPerson = PersonType();
  newPerson.name('James');
  newPerson.address(newAddr);
  INSERT INTO people VALUES(newPerson);
  ```

- **序** (ordering)：我们经常需要比较或排序 UDT 类型的元素，于是需要定义他们之间的序。
  - 相等关系：全部对应属性相等为相等。

    ```
    CREATE ORDERING FOR T
      EQUALS ONLY BT STATE;
    ```

  - 全序：需要定义方法 `F(x1, x2)` 测试 `x1, x2` 的序；返回值小于 0 表示小于，等于 0 表示等于，大于 0 表示大于。

```
CREATE ORDERING FOR T
  ORDERING FULL BY RELATIVE WITH F;
```

- 其他比较方法：
  - 严格对象比较：两个对象相等当且仅当他们是同一个对象；
  - 自定义相等：自定义方法 `F` 测试两个对象是否相等，若是返回 TRUE，否则返回 FALSE。

    ```
    CREATE ORDERING FOR T
      EQUALS ONLY BY RELATIVE WITH F;
    ```

  - 自定义映射：将对象通过映射 `f` 得到一个实数，然后比较实数大小关系。