6. 高阶数据库模型（数据库概念设计）

实体-联系模型 (entity-relationship model, ER model)：可通过画图得到 ER 图。

- **实体** (entity)：类似于面向对象中的对象；
- **实体集** (entity set, ES)：许多相似实体的集合，图中用矩形表示；
- **属性** (attribute)：实体集中实体的的性质，图中用椭圆表示；
- **联系** (relationship)：联系两个或多个实体集的，用于描述若干实体集的实体之间存在联系，图中用菱形表示。
- 图中的边连接 ES 及其属性、关系以及其连接的 ES；ER 图描述数据库的模式。
- **二元联系** (binary relationship)：连接两个 ES 的联系，分为如下若干种，在 ER 图中，“多”用不带箭头的连线表示；“一”代表着最多一个，用箭头表示；恰好一个用圆箭头表示。
  - 多对多： many-to-many；
  - 一对多/多对一：one-to-many, many-to-one；
  - 一对一：one-to-one。
- **多元联系**：连接多个 ES 的联系（共同关联）；如老师、课程以及开课院系形成了一个三元联系。
  - 三元联系与三个二元联系不等价！
  - 如何转化为二元联系？人为增加一个虚构的抽象实体集 (connecting ES)，将这个实体集与原有的每个实体集中分别创建联系。
- 一个 ES 可能参与不止一个关系，给边带上标签以区分不同的意义。
- **子类** (subclass)：实体集的子集（特例），其可能满足一些特殊性质。
- **isa 关系** (isa relationship)：联系两个实体集，表示一个是另一个的子类，用三角形表示，是一个一对一的关系，只允许树形结构，同时不允许多重继承（如助教既是老师，又是学生）。
  - isa 关系组成了一棵树；
  - 子类具有父类具有的所有属性，同时具有祖先类具有的所有属性（但是不在子类中显示，在祖先类中显示）；
  - 子类参与父类参与的所有关系，同时参与祖先类参与的所有关系（但是不在子类中参与，在祖先类中参与）；
  - **与面向对象不同**：面向对象的子类中具有祖先类的所有信息，不需要到祖先类中进行查询。
- **ER 模型要求实体集必须有键值 (key)，即没有两个元素有着相同的键值**，当然有时候一个实体集的键值可能在另一个实体集中。如果有超过一个键值，则用下划线表示主键值。子类需要继承父类的键值。
- **引用完整性** (Referential Integrity)：利用圆箭头来限制多对一/一对一关系（必须存在）。
- **度限制** (Degree Constraint)：限制使用这个连接的实体数量，用标签标注在边上。如每个院系不能有超过 200 个学生，可以利用联系边上的标签 “≤ 200” 来标注。
- **弱实体集** (Weak ES, WES)：某个实体集的所有属性都不足以构成其自身的键值，需要当前实体集连接的其他 ES 的某些属性来构成自身的键值，那么当前实体集称为弱实体集。
  - **来源**：unit-of 或 belong-to 结构（如班级与学校，摄制组与电影制片厂等），或是由于消除多元联系引入的虚拟的抽象实体集。

- **键值**：来自本身实体集的零个或多个属性，以及来自于**支撑实体集** (supporting ES) 的若干属性。
- **要求**：
  - 弱实体集必须与其支撑实体集中有支撑联系 (supporting relationship) R；且 R 是多对（恰好）一 (many-to-exactly-one) 的二元关系；
  - 支撑实体集将其键值提供给当前弱实体集；当然支撑实体集可能本身也是 WES，那么其键值可能由其他实体集提供；
  - 一个弱实体集可能由多个支撑实体集支撑，他们中的任何之一都可以支持当前弱实体集；他们任何之一的键值都可能出现在弱实体集的键值中。
- 在 ER 图中，弱实体集用双框矩形表示；支撑关系用双框菱形表示；键值用下划线表示。

设计准则

- 数据库应该反映现实；
- 数据库应该避免重复（一件事情只说一遍）；
- 数据库应该尽可能简单（避免多余的不必要的元素）；
- 数据库应该选择合理的关系（如果一个关系可以被其他关系推出，那么这个关系是不必要的）；
- 数据库应该正确选择所用的信息：
  - 一个东西需要被当成实体集，当且仅当其需要很多信息来描述，否则应该被当作属性；
  - 多元联系与多个二元联系的选择（根据实际情况）。
- 数据库不应过度使用弱实体集。

将 E/R 模型转化为关系模型的一般规则

- 实体集：关系；实体集的属性：关系的属性；
- 弱实体集：关系，其中包含支撑实体集的键值属性；
- 联系：关系；关系的属性包含该联系所连接的多个实体集的所有键值，同时包含联系自身可能带有的属性；
  - 如果一个联系中含有同一个实体集的两个元素，需要对属性进行改名；
  - 对于多对一 (many-to-one) 的联系，按照如上操作得到了关系 R_1, R_2, R_3（分别代表两个实体集和联系），一般可以将 R_1 与 R_2 合并。如“学生 住在 寝室”，按照如上操作得到了学生、住在、寝室关系；由于每个学生在仅“住在”一个寝室，可以将 R_2 合并至 R_1，直接在 R_1 中加入 R_3 的键值即可，可以省去联系转化而来的关系；
  - 对于一对一 (one-to-one) 的联系，可以将联系与任意一个实体集所转化而成的关系合并；
  - 弱实体集参加的联系所转化的关系需要使用其完整的键值（可能存在于支撑实体集中）；支撑联系所产生的关系不需要存在。
- 子类：
  - **E/R Style**：为每个参与 isa 关系的实体集构造一个关系，其中包含根的键值以及当前实体集的特殊属性。不需要为 isa 构造任何关系。例如 `Students(sno, name, age, dept)` 以及 `Graduates(sno, advisor)`，将当前实体集到根所涉及到的所有关系自然连接即可得到当前实体集的所有信息。
  - **O/O Style**：为每个参与 isa 关系的实体集构造一个关系，其中包含从根到当前实体集的所有属性（所有相关属性）。例如 `Students(sno, name, age, dept)` 以及 `Graduates(sno, name, age, dept, advisor)`。
  - **Single Relation**：用**一个关系**来存储整个关系树，包含其中所有可能的属性；因此某些实体

的某些属性可能为 NULL。

7. ODL 语言

面向对象数据库：改进现有的面向对象编程语言来允许程序员创建持久对象（数据库）。

ODL 类型 (ODL type)：ODL 的类型系统包含：

- 基本类型：如整型、浮点型、字符型、字符串型、枚举型等等，或是类的名称；
- 用类型构造子定义的结构类型：如集合 `Set<T>`，可重集 `Bag<T>`，列表 `List<T>`，数组 `Array<T, i>`，字典 `Dictionary<KeyType, RangeType>` 等；或结构体 `Struct N {T1 F1, T2 F2, ..., Tn Fn}`。
- **属性的类型**：
  1. 基本类型；
  2. 用类型构造子对 1（或 2，即重复应用）应用得到的类型；
- **关系的类型**：
  1. 基本类型
  2. 用类型构造子对 1 应用得到的类型；

> 【例】
>
> ```
> class Teacher {
>   attribute string name;
>   attribute Struct Addr {
>     string street,
>     string city,
>     int zip
>   } address;
>   attribute Enum Degrees {
>     bachelor, master, doctorate
>   } degree;
>   relationship ...;
> }
> ```

ODL 类 (ODL class)：一个 ODL 类定义包含：类名称，可选的键值定义以及属性、联系描述等等。

```
class <className> {
  <list of properties>
}
```

不同性质用分号隔开，下面是一些性质的定义。

- 属性是带有类型的元素，通常不包含类，即

  ```
  attribute <type> <name>
  ```

  属性类型 `<type>` 可以是：简单类型（整型、字符串等等）；结构体、集合等等。

- 联系连接着一个实例与其他类的一个或多个实例，即

```
relationship <type> <name> inverse <relationship>;
```

联系必须成对出现，用 `inverse <relationship>` 定义联系的另一方面。即假设类 C 与类 D 有联系 R，那么类 D 必须与类 C 有联系 S，且 R 与 S 必须完全互逆。

联系类型 `<type>` 可以有以下几种：

  - 一个类，如 C；那么一个有着此联系的实例仅能与 C 中的一个实例联系；
  - 一个集合，如 `Set<C>`；那么一个有着此联系的实例联系着多个 C 中实例；
  - 其他，如 `Bag<C>`，`List<C>`，`Array<C>`；分别表示可重集、有序表、长度限定的数组。

> 【例】学生选课的数据库。
>
> ```
> class Student { ...
>   relationship Set<Course> takes inverse Course::takenBy;
>   relationship Course favorite inverse Course::fans;
> }
> class Course { ...
>   relationship Set<Student> takenBy inverse Student::takes;
>   relationship Set<Student> fans inverse Student::favorite;
> }
> ```

注：ODL 不支持三路或更多路的联系，我们可以通过建立连接类（类似 ER 图中的方法）来建立多路联系。

- **连接类** (Connecting Classes)
  - **多路联系的情况**：假设我们想要构造一个连接类 X, Y, Z 的联系 R；那么构造一个类 C，其中的元素代表着 X, Y, Z 中实例的三元组 (x, y, z)；我们再构造三个从 (x, y, z) 到 x, y, z 的多对一的联系即可。
  - **联系包含着属性的情况**：如“学生-课程-成绩”的情况，在“学生-课程”之间增加一个连接类 `SCG`，其中包含“成绩”属性；然后构造两个从 `SCG` 到学生、课程的多对一联系即可。

    ```
    class SCG {
      attribute int grade;
      relationship Student theS inverse Student::toSCG;
      relationship Course theC inverse Course::toSCG;
    }
    ```

    然后在 `Student`、`Course` 中分别进行修改并加入联系即可。
- **子类** (subclass)：继承了所有父类 (superclass) 的性质，并且可能会有更多的性质，如

```
class GradStud extends Student {
  attribute string advisor;
}
```

多路继承：一个子类可以从多个父类进行继承，如

```
class TA extends Student:Teacher {...}
```

- **键值** (keys)：允许一个类有多个不一样的键值，如

```
class Student (key sno) {...}
class Teacher (key tno, ID) {...}
class Classes (key (school, number), (number, tno)) {...}
```

ODL 与关系模型的转化

- **一般规则**：类对应关系，属性对应属性，联系对应连接关系，实例对应元组。
- **一般类型**：对应成一般类型的属性，可能需要加入人造的键值。
- **结构体类型**：将结构体展开，结构体中每一个元素代表一个属性；如

```
class R { ...
  attribute Struct S {T1 F1, T2 F2, T3 F3} a;
  ...
}
```

  可以转化成 $R(\dots, F_1, F_2, F_3, \dots)$，其中 T_1, T_2, T_3 是原子类型。如果两个结构体中的元素有相同的名字，则进行换名。
- **集合类型**：为集合中的每一个元素加入一个元组，可能会出现重复（BCNF/4NF violations），为避免重复也可以为每个集合新建一个关系来单独表示，即可符合 4NF/BCNF 的要求。
- **可重集类型**：类似集合类型表示，同时加入一个属性 `count` 表示该元素出现的次数；由于抽象关系模型不允许重复，因此我们不能加入两个相同的元组来描述可重集。
- **有序表类型**：类似集合类型表示，同时加入一个属性 `order` 表示该元素在表中出现的位置即可。
- **数组类型**：由于数组长度为 L 固定，可以直接加入 L 个属性来代表数组属性；如果数组中有结构体类型，则继续将结构体展开即可。
- **字典类型**：类似一个 `(key, value)` 组的集合，利用集合进行转化即可。
- **其他包含结构体的类型**：先进行对应转换，然后再对结构体展开即可。
- **联系到关系的转化**：建立一个新的关系来连接两个类的键值；只需要对一对互逆的联系构造一个关系即可。对于单值联系，可以直接将单值一方的键值复制到另一方即可，不需要新建一个联系。
- **子类到关系的转化**：类似于 E/R 图转化的 OO-style，对于一个子类创建一个新的关系，注意其有继承等特性，任意连通块都可能成为一个新的关系。