第二部分　关系数据模型

关系数据模型是现代数据库技术的基础，它使这一领域具有了科学性。任何有关数据库技术基础的书籍，如果没有包括关系模型，那它是不全面的。同样，如果没有深入理解关系模型，而声称是数据库领域的专家，那是无法让人接受的。其实关系模型并不非常难理解。但是，需要强调的是：它是基础，并且在人们所能预见的时期内它仍然是。

在第3章曾经提到，关系模型考虑了数据的三个主要方面：数据结构、数据操作和数据的完整性。在本书的这一部分，依次讨论这三个方面：第5章讨论结构，第6章和第7章讨论操作，第8章讨论完整性（有两章关于操作，是因为模型的操作可以通过两个截然不同而又等价的方法来实现，它们是关系代数和关系演算）。最后，第9章介绍视图。

关系模型不是一个静态的事物，理解这一点非常重要——过去几年里它不断发展和进化，未来将仍然如此。[⊖] 接下来的内容反映了作者和在这个领域里的其他工作者的思想（特别是，如同在序言提到的，参考了《第三次宣言》[3.3]一书的观点）。本书尽量使内容相对地更加完备、准确，叙述风格尽量采用了授课形式。但读者不要把这些当作一成不变的东西。

需重申的是，关系模型并不难理解，但它是理论，大多数理论都带有它自己的专门术语。（由于3.3节已说明的原因）关系模型在这方面也不例外。在本书这一部分，要用到专门术语。不可否认，最初这些术语会带来一点混乱，并且会成为理解的障碍。因此，如果读者在理解上有些困难，需要耐心些：一旦熟悉了有关的术语，就会发现内容非常简单。

接下来的五章篇幅很长（这一部分完全可单独成书）。但是，长度正反映了这部分内容的重要性。用一两页的篇幅来概述这部分内容也是完全可能的；事实上，关系模型的一个主要优点就是其基本内容非常容易阐述和理解。然而，用过短的篇幅来介绍是不合理的，因为它不能说明关系模型的应用广泛性。本书这部分的长篇论述，不应看作是模型复杂性的表现，而应看作是该模型作为无数深远发展之基础的重要性和成功之处的诠释。

最后，关于SQL的一点说明。在本书的第一部分已解释过，SQL是标准关系数据库语言，市场上几乎每一种数据库产品都支持它（或者，更准确一点，是支持它的变种［4.21］）。由此而来的结果是，不包含SQL的现代数据库书籍是不能称其为全面的。接下来关于关系模型的章节中，在用到SQL的地方也将顺便讨论它的有关特性（第4章覆盖SQL的基本内容）。

第5章　域、关系和基本关系变量

5.1　引言

正如第3章所讲的，关系模型可以认为有三个基本部分，即：数据结构、数据完整性以及

[⊖] 在这方面，它类似于数学（数学也不是静止不变的，它随时间而发展）。实际上，关系模型本身可以看作是数学的一个分支。

数据操作。每一部分都有各自的专门术语。最重要的结构化术语如图5-1所示（所用的供应商样本关系取自图3-8所示的供应商-零件数据库，并加以扩展以显示应用的数据类型或域）。涉及到的术语有：关系（relation）、元组（tuple）、势（cardinality）、属性（attribute）、度（或目）（degree）、域（domain）以及主码（primary key）。

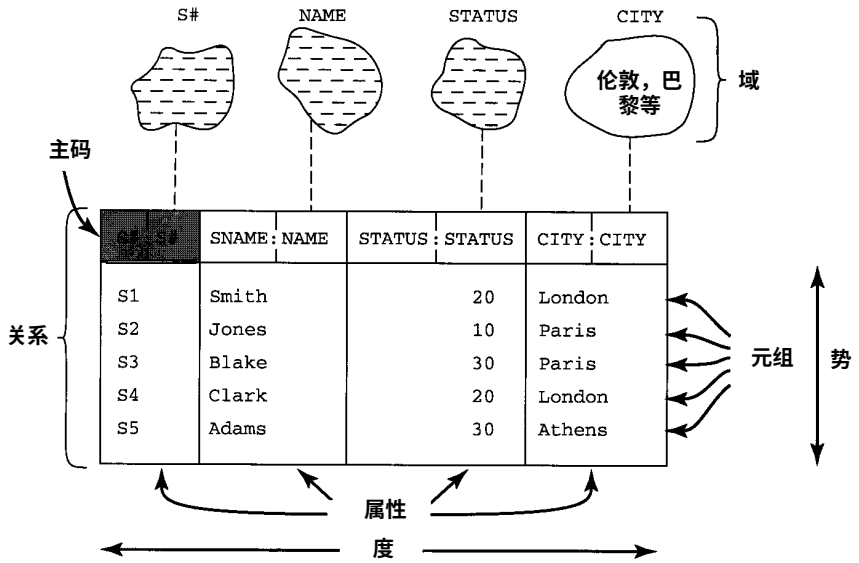

图5-1 结构化术语

在这些术语中，关系和主码，读者已经通过第3章熟悉了。我们先简单介绍其余的术语，然后在接下来的章节里，将给出正式的定义。简单地说，如果把关系看作一张表，那么一个元组就是这张表的一行，一个属性就是一列；元组的数目称为势，属性的数目称作度；域是值的集合，关系中属性的值取自域。例如，图5-1中标为S#的域是所有供应商号码的集合，并且供应商关系中出现的每一个S#的值都取自这个集合（同样在供货关系中每一个S#的值也来自这个集合，参见图3-8）。

图5-2是前面所提到术语的一个小结。图中显示的“等价（equivalence）”都只是近似的（正式的关系术语有准确的定义，然而非正式的“相等”只是粗略简单的定义）。例如，关系和表实际上并不是同一件事物，尽管在实践中常把它们认为是相等的，如同在本书第一部分所看到的。

下面，开始较为正式的论述。

正式的关系术语	非正式的等价术语
relation (关系)	table
tuple (元组)	row or record
cardinality (势)	number of rows
attribute (属性)	column or field
degree (度)	number of columns
primary key (主码)	unique identifier
domain (域)	pool of legal values

图5-2 结构化的术语（小结）

5.2 域

一个域就是一个数据类型（简称为类型），它可能是一种系统定义的类型，如INTERGER或CHAR。更常见的是用户自己定义的类型，如供应商-零件数据库中的S#或P#或WEIGHT或QTY。实际上，可以把“类型”和“域”这两个术语混用，在本书中就是这样做的（我们更喜欢“类型”这个称呼；当用到“域”时，主要考虑了历史的原因）。

什么是一个类型呢？它就是相关类型的所有值的集合。例如，类型INTERGER是所有整数的集合；类型S#是所有供应商号码的集合；等等。伴随着给定的类型，就会出现操作符的问题：即何种操作符能合法地应用于给定类型的值上；也就是说，该类型的值只能被定义在相应类型上的操作符操作。例如，对于INTEGER（系统定义）：

- 系统提供操作符“=”、“<”，等等，用来比较整数大小。
- 系统同时提供操作符“+”、“*”，等等，执行整数上的算术操作。
- 系统不提供“||”（连接）、SUBSTR（取子串）等操作符用在整数上进行字符串操作。换句话说，整数不支持字符串操作。

在一个能提供正确类型支持的系统上，用户可以定义自己的类型，例如：S#类型。并且，能定义“=”、“<”等操作符，用来比较供应商号码。然而，不能定义“+”、“*”等操作符，即：这种类型不支持供应商号码上的算术操作（其实对两个供应商号码相加或相乘是毫无意义的）。

因此，我们严格区分一种类型本身和这种类型的值在系统内部的物理表示[㊀]。（事实上，类型是一个模型问题，而物理表示是一种实现问题）。例如，供应商号码可以被物理地表现为字符串，但这并不意味着可以在供应商号码上进行字符串操作；仅当在类型上定义了相应的操作符，才能执行这些操作。当然，给一种类型定义操作符，依赖于该类型的具体含义和语义，而不依赖于类型的值被物理表示的方式，实际上，物理表示应该对用户是隐藏的。

到目前为止，其实我们谈论的就是程序设计语言中的强类型问题。不同的作者对这个词语有稍微不同的定义；然而当使用的时候，它有相同的含义：（a）每一个值都属于一个类型；（b）无论何时执行一种操作，系统都要检查操作对象是否是相关操作的正确类型。例如，考虑下面的表达式：

```
P.WEIGHT + SP.QTY     /* part weight plus shipment quantity  */
P.WEIGHT * SP.QTY     /* part weight times shipment quantity */
```

第一个表述没有意义，系统会拒绝它。第二个是正确的，它代表发货里面所有零件的总重量。因此，在重量和数量上定义的操作符包括“*”，但不包含“+”。

再看几个例子，这里包括了比较操作（实际是相等比较）：

```
P. WEIGHT = SP. QTY
P. CITY = S.CITY
```

同样，第一个表达式无意义，而第二个有意义。因此，我们给重量和数量定义的操作符不应包含“=”，但对城市来说可以包含[㊁]。（事实上，我们同参考书[3.3]一样有充分理由相信：

㊀ 类型有时称为抽象数据类型（ADT），这是为了强调类型必须与它们的实现区分开。这里并不使用这个术语，因为它可能使人误解为有些类型不是抽象的，我们强调类型必须永远与其实现相区别。

㊁ 顺便提一下，关于类型上应用何种操作符是合法的问题，以前很多数据库文献（包括本书的前期版本）只考虑了比较操作符如“=”和“>”，而忽略了如“+”和“*”等其他操作符。

做相等比较的操作符“ = ”可被用在任何一种类型上；它永远可用来比较同一种类型的两个值是否相等）。

到目前为止，还没有谈及关于构成一种类型的值的本质，实际上，它们可以是任何形式的。一般把它们简单化为数字、字符串等，但是，关系模型并没有把它们局限于如此简单的形式。它允许使用关于音频的域、图的域、视频的域、工程设计图的域、建筑蓝图的域、几何点的域（等等）。仅有的要求是：域的值必须只能被定义为在相关域上的操作符所操作（物理表示必须隐藏起来）。

以上内容是非常重要的，由于在很多情况下被误解，因此，重申如下：

对数据类型的支持与对关系模型的支持是互不相关的

1. 值的类型划分

每一个值属于特定的类型。换句话说，如果 *v*是一个值，那么 *v*可以认为带有一种标记，能显示“ *v*是一个整数 ”、或“ *v*是一个供应商号码 ”、或“ *v*是一个几何符号 ”（等等）。注意，根据定义，任何给定的值仅属于一种类型[⊖] ，并且不能改变类型（意思是，不同的类型永远不能一起使用，即它们没有共同的值）。

一种给定的类型可以是标量的，也可以是非标量（ nonscalar）的。一个非标量的类型可以显式定义为有用户可见的分量（ component）。从这个意义上来说，关系类型是非标量的；例如，图5-1所示的关系是某种关系类型，并且这个类型有用户可见的分量（它们是：属性 S#、SNAME、STATUS和CITY）。与此相反，标量类型没有用户可见的分量。要点如下：

1) 在本节的后半部分，接触的所有类型都将是标量的，因此只要提“ 类型 ”就是指一个标量的类型。
2) 当然，一个给定标量值的物理表示（即一个标量类型的值）可能是相当复杂的。特别地，它可有分量（但是那些分量是用户不可见的）。例如，在适当的情况下，一个标量值的物理表示由字符串的数组组成。
3) 因为没有用户可见的分量，标量类型有时被说成是封装的，有时说成是原子的。我们不想采用以上的说法，因为过去它们给大家带来了很多的混乱。确切地说，它们给以下的区别带来混乱：模型和实现之间，类型和物理表示之间（并且这是经常的）。
4) 后面会看到，标量类型有所谓的“ 可能表示（ possible representation）”，反过来，可能表示却有用户可见的分量。不要被以下的事实搞混：这些分量不是类型的分量，它们是可能表示的分量。类型本身仍是标量的。

2. 类型定义

贯穿本书，将使用Tutorial D语言（或者，这种语言的一个简单变种），这种语言在第3章中举例说明时用到过。大体上说， Tutorial D类似于Pascal语言。接下来会一点点介绍它的特征。很明显，需要做的首要的事情，是为用户定义自己的类型提供一种方法：

```
TYPE  <type name>  <possible representation>...
```

通过下面的例子，为供应商和零件数据库定义类型：

```
TYPE  S#        POSSREP CHAR ;
TYPE  NAME      POSSREP CHAR ;
```

⊖ 除非支持类型继承。在第19章之前，忽略这种可能性。

```
TYPE   P#          POSSREP (CHAR) ;
TYPE   COLOR       POSSREP (CHAR) ;
TYPE   WEIGHT      POSSREP (RATIONAL) ;
TYPE   QTY         POSSREP (INTEGER) ;
```

解释：

1) 在第3章中，供应商的STATUS属性和供应商和零件的CITY属性是用固有类型定义的，而不是用户定义的类型，因此，对这些属性没有类型的定义。

2) 从前面可知，物理表示对用户是隐藏的。因此，类型定义不涉及到物理表示。但是，这些物理表示作为概念 / 内部的映射（conceptual / internal mapping）必须详细说明（参看第2章2.6节）。

   我们要求每一类型至少有一个相关的“可能表示”，原因将在下节中讨论。例如，类型S#、NAME、P#以及COLOR的值可能用字符串表示（可以是任意长度），类型QTY的值可用整数表示，类型WEIGHT的值可用有理数表示。注意：在这本书中（和［3.3］一样），我们喜欢用更确切的有理数(RATIONAL)，而不是所熟悉的实数(REAL)。

3) 采用的语法规则：（a）没有命名的可能表示继承相关类型的名称；（b）没有命名的可能表示的分量继承相关可能类型的名称。例如，为类型 QTY定义的唯一的可能表示也叫做QTY，可能表示的唯一分量也叫QTY。

4) 一旦定义了一种新的类型，系统就会为这种新的类型在目录（ catalog）中记一个条目（entry）进行描述（如果需要复习关于目录的知识，可参考第 3章3.6节）。操作符的定义也是类似情况（参看下面两节）。

5) 大家可能已注意到，上面的类型定义没有具体指定组成类型的实际值！这项功能用相关联的“类型约束”来实现（在上面的语法中，用省略号……指出），这要在第8章学习。当然，如果不再需要一个类型，可用如下方式删除：

```
DROP  TYPE  < type  name >
```

其中的*<type name>*必须指明是一个用户定义的类型，而不是系统固有的类型。该操作会使目录中描述该类型的条目被删除掉，从而意味着这一类型对系统是不可知的。注意：如果要被删除的类型在别的地方正被用到 —— 特别是有些关系的属性定义基于此类型，那么 DROP TYPE将会失败。

3. 可能表示

为了说明“可能表示”这个概念的重要性，考虑一个稍微复杂的例子：

```
TYPE POINT /* geometric points */
     POSSREP CARTESIAN ( X RATIONAL, Y RATIONAL )
     POSSREP POLAR ( R RATIONAL, THETA RATIONAL ) ;
```

首先看到，类型POINT 有两个不同的可能表示CARTESIAN 和POLAR，这表明，几何点可以用笛卡尔或极坐标来表示（当然，在特定的系统中，物理表示可能是笛卡尔坐标，或极坐标，也可能是完全不同的另外一种形式）。每一种可能表示有两个分量。特别注意，这个例子不同于前面的例子，因为这个例子中可能表示和它们的分量都给定了具体的名字。

每一个可能表示的说明，都可能引起下面几种操作符的自动 ⊖ 定义：

⊖ 这里“自动”的含义是：(a) 无论何种代理—可能是系统，也可能是用户 —— 只要定义了可能表示，就要定义相应的操作符；(b) 只要那些操作符没有定义完，可能表示的定义就没结束。

- 选择子(selector)操作符（与可能表示有相同的名字）。它允许用户，通过给定可能表示的每个分量的值，指定或选择类型的一个值；
- THE_操作符（对应于每个可能表示的分量）的集合。用户可以利用它来获得类型值的可能表示的分量。

例如，这里有在类型POINT上用选择子和THE_操作符进行操作的几个例子：

```
CARTESIAN ( 5.0, 2.5 )
/* denotes the point with x = 5.0, y = 2.5 */

CARTESIAN ( XXX, YYY )
/* denotes the point with x = XXX, y = YYY -- */
/* XXX and YYY are variables of type RATIONAL */

POLAR ( 2.7, 1.0 )
/* denotes the point with r = 2.7, theta = 1.0 */

THE_X ( P )
/* returns the x coordinate of point P */
/* -- P is a variable of type POINT    */

THE_R ( P )
/* returns the r coordinate of point P */
```

注意：选择子（或者说，选择子调用）是对大家所熟悉的 *literal*(文字)的概括。

为了搞清楚前面所说的在实际系统中如何工作，假设点的物理表示用笛卡尔坐标（尽管物理表示没有必要用所讲的任何一种来表示）。于是系统提供某种受高度保护的操作符——下面用斜体的伪代码表示——用以有效地表现物理表示，并且，类型的定义者会用这些操作符完成CARTESIAN和POLAR的操作[⊖]。例如：

```
OPERATOR CARTESIAN ( X RATIONAL, Y RATIONAL )
                                  RETURNS ( POINT ) ;
   BEGIN ;
      VAR P POINT ;
      X component of physical representation of P  :=  X ;
      Y component of physical representation of P  :=  Y ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;

OPERATOR POLAR ( R RATIONAL, THETA RATIONAL )
                                  RETURNS ( POINT ) ;
   RETURN ( CARTESIAN ( R * COS ( THETA ),
                        R * SIN ( THETA ) ) ) ;
END OPERATOR ;
```

在上面的代码中，POLAR的定义用到了CARTESIAN，也用到了（假设是内置的）操作符SIN和COS。POLAR的定义也可以直接用那些受保护的操作符来表示，如下：

```
OPERATOR POLAR ( R RATIONAL, THETA RATIONAL )
                                  RETURNS ( POINT ) ;
   BEGIN ;
      VAR P POINT ;
      X component of physical representation of P
                                     :=  R * COS ( THETA ) ;
      Y component of physical representation of P  :=  Y ;
                                     :=  R * SIN ( THETA ) ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;
```

类型定义者也会用那些受保护的操作符描述THE_操作符，如下所示：

```
OPERATOR THE_X ( P POINT ) RETURNS ( RATIONAL ) ;
   RETURN ( X component of physical representation of P ) ;
END OPERATOR ;
```

⊖ 显然，类型定义者是一般规则的例外。这个规则就是：用户涉及不到物理表示。

```
OPERATOR THE_Y ( P POINT ) RETURNS ( RATIONAL ) ;
   RETURN ( Y component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_R ( P POINT ) RETURNS ( RATIONAL ) ;
   RETURN ( SQRT ( THE_X ( P ) ** 2 + THE_Y ( P ) ** 2 ) ) ;
END OPERATOR ;

OPERATOR THE_THETA ( P POINT ) RETURNS ( RATIONAL ) ;
   RETURN ( ARCTAN ( THE_Y ( P ) / THE_X ( P ) ) ) ;
END OPERATOR ;
```

从上面可看到，THE_R和THE_THETA 的定义利用了 THE_X和THE_Y，同时用到了SQRT和ARCTAN操作符（假定是内置）。同样，THE_R和THE_THETA也能直接通过受保护的操作符来定义（详细的实现作为一个练习）。

关于POINT的例子就到此为止。然而，所有上面讨论的内容也可以应用于简单的类型——例如QTY，理解这一点很重要。这里有一些例子：

```
QTY ( 100 )

QTY ( Q )

QTY ( ( Q1 - Q2 ) * 2 )
```

这里有几个有关 THE_操作符调用的例子：

```
THE_QTY ( Q )

THE_QTY ( ( Q1 - Q2 ) * 2 )
```

特别强调的是，值都是有类型的，因此对“某种货物的数量是 100”的说法，严格来讲就是不正确的。这里的数量是类型 QTY的一个值，而不是INTEGER的一个值。因此，对于供货量应该更准确地说是QTY（100），而不是简单的100。然而，在非正式的上下文中，不用如此精确，这样就用100替代QTY（100），以图方便。在图3-8 ⊖ （供应商和零件数据库）和图4-5（供应商-零件-工程数据库）中用到了这样的简写。

下面给出最后一个关于类型定义的例子，LINESEG（线段）：

```
TYPE LINESEG POSSREP ( BEGIN POINT , END POINT);
```

一种给定的可能表示当然可以利用用户定义的类型来定义，而不是像前面的例子那样只使用系统定义的类型。

4. 操作符定义

现在来看一下怎样定义操作符。下面有几个例子。第一个是在内置类型 RATIONAL（有理数）上的用户定义的操作符：

```
OPERATOR ABS ( Z RATIONAL ) RETURNS ( RATIONAL ) ;
   RETURN ( CASE
               WHEN Z ≥ 0.0 THEN +Z
               WHEN Z < 0.0 THEN -Z
            END CASE ) ;
END OPERATOR ;
```

操作符ABS（绝对值）就使用了一个类型为 RATIONAL的参数Z定义，返回了同样类型的一个结果（换句话说，一个 ABS调用——例如ABS(AMT1-AMT2)——就是类型RATIONAL的一个表达式）。

下面的一个例子，DIST (两点间距离)，包含了许多用户定义的类型：

⊖ 在SQL中也做同样的要求，但原因不一样：因为 SQL不支持用户定义类型。参看附录 B。

```
OPERATOR DIST ( P1 POINT, P2 POINT ) RETURNS ( LENGTH ) ;
   RETURN ( WITH THE_X ( P1 ) AS X1 ,
                 THE_X ( P2 ) AS X2 ,
                 THE_Y ( P1 ) AS Y1 ,
                 THE_Y ( P2 ) AS Y2 :
            LENGTH ( SQRT ( ( X1 - X2 ) ** 2
                          + ( Y1 - Y2 ) ** 2 ) ) ) ;
END OPERATOR ;
```

假设LENGTH是一个使用RATIONAL可能表示的用户定义类型。注意WITH子句的使用使某些表述更简便了。

下面的例子是类型POINT的“ = ”比较操作符：

```
OPERATOR EQ ( P1 POINT, P2 POINT ) RETURNS ( BOOLEAN ) ;
   RETURN ( THE_X ( P1 ) = THE_X ( P2 ) AND
            THE_Y ( P1 ) = THE_Y ( P2 ) ) ;
END OPERATOR ;
```

RETURN语句的表述中对类型RATIONAL使用了系统内置操作符“ = ”。为简单起见，假定符号“ = ”可以被用在等价的操作符中（可在所有类型上，包括 POINT）；我们不考虑这样的插入符号在实际中怎样说明，因为这只是一个语法的问题。

下面是类型QTY上的“ < ”操作符：

```
OPERATOR LT ( Q1 QTY, Q2 QTY ) RETURNS ( BOOLEAN ) ;
   RETURN ( THE_QTY ( Q1 ) < THE_QTY ( Q2 ) ) ;
END OPERATOR ;
```

RETURN语句的表述中对型INTEGER使用了系统内置操作符“ < ”。同样，假定这一操作符用于所有可排序的类型上，而不只是类型 QTY（实际上，从定义看，一个可排序的类型就是一个“ < ”能应用的类型。一个不能排序类型的简单例子是类型 POINT）。

最后是一个关于更新操作符定义的例子（前面所有的例子中都是只读的操作符）。[⊖] 可以看到，定义中包含了一个 UPDATES说明，而不是RETURNS说明；更新操作符没有返回值，并且必须被显式的CALL调用[3.3]。

```
OPERATOR REFLECT ( P POINT ) UPDATES ( P ) ;
   BEGIN ;
      THE_X ( P )  :=  - THE_X ( P ) ;
      THE_Y ( P )  :=  - THE_Y ( P ) ;
      RETURN ;
   END ;
END OPERATOR ;
```

REFLECT操作符能有效地移动笛卡尔坐标的点（*x*,*y*）到相反的位置（-*x*,-*y*）；它通过适当修改点的参数来实现。注意这个例子中“ THE_伪变量 ”的使用。THE_伪变量是THE_操作符在目标位置的一个调用（特别地，在一个赋值的左边）。这样的一个调用实际上是指定 —— 优于简单地返回值 —— 参数的特定分量（应用的可能表示）。例如，在REFLECT的定义中下面的赋值

```
THE_X ( P )  :=  ... ;
```

实际上是根据参数P给参数变量的分量X赋了一个值。当然，任何被更新操作符修改的参数变量（特殊的通过给 THE_伪变量赋值）必须特别地作为一个变量指定，而不是作为一般的表达式。

伪变量可以嵌套，例如：

```
VAR LS LINESEG ;

THE_X ( THE_BEGIN ( LS ) )  :=  6.5 ;
```

[⊖] 只读 (read-only) 和更新 (update) 操作符也被分别看作是observer和mutator，特别是在对象系统中（参看本书的第六部分）。

最后，如果一个操作符不再被使用，必须能够删除掉，例如：

```
DROP  OPERTAOR  REFLECT;
```

被删除的操作符必须是用户定义的，而不是系统内置的。

5. 类型转换

考虑下面的类型定义：

```
TYPE  S#  POSSREP  (CHAR);
```

缺省的情况下，这里的可能表示为名字 S#，相应的选择子操作符同样如此。因此，下面是一个有效的选择子调用：

```
S# ('S1');
```

(它返回某个供应商号码)。注意，选择子 S#可能被认作是把字符串转换为供应商号码的类型转换符；类似地，选择子 P#可能被认作是把字符串转换为零件号码的类型转换符； QTY选择子可能被认作是把整数转换为数量的类型转换符；等等。

第3章中有几个例子，在零件号码和字符串之间进行了比较。例如，例 3.4包含了下面的WHERE子句：

```
... WHERE  P#='P2'
```

比较式的左边是 P#类型，右边是CHAR类型；表面上看，比较会出现一个类型错误（实际上是一个编译期间的类型错误)。然而，实际的情况是：系统认为它能利用 P#这个类型转换符把CHAR转换为P#，因此系统有效地重写了比较式：

```
... WHERE P#=P#('P2')
```

这样表达式就是合法的了。

通过这种方式调用类型转换符即为强制转换，在整个第 3章大量使用了它。然而，实际中我们都知道强制会带来程序错误。因为这个原因，本书以后的部分不再允许强制转换，操作双方必须是合适的类型。特别地，我们认为：

- 在“ = ”、“ < ”和“ > ”上的比较双方必须是同一类型；
- 赋值符(“ ：= ”)的左右两方必须是同一类型。

当然，允许定义通常名为“ CAST ”的操作符，允许在需要的地方被显式调用以进行转换，例如：

```
CAST_AS_RATIONAL(5)
```

前面已经指出，选择子 —— 至少那些带有一个参数的 —— 可以认为是显式的转换符。

6. 小结

这一节中讲的对类型的完全支持蕴涵了很多重要的东西，现简要地概括如下：

- 首先且最重要的是，它意味着系统（ a）确切知道哪一个表达式是合法的，(b）知道合法表达式的结果的类型。
- 它也意味着一个给定的数据库中所有类型的集合是一个封闭集 —— 也就是说，每一个合法表达式结果的类型对系统是可知的。特别地，如果比较式是合法的表达式，这个类型的封闭集必须包含布尔型或真值型！
- 特别地，由于系统知道每一个合法表达式的结果，因此它知道哪一个赋值是合法的，也知道哪一个比较是合法的。

最后说明一点，我们已经知道域是一个任意复杂的数据类型，它可以是系统定义的，也

可以是用户定义的，并且它的值只能被定义在有关类型上的操作符操作（并且它的内部表示对用户是隐藏的）。而对对象的系统而言，可以看出最基本的对象概念，即对象类，就是一种由系统或用户定义的任意复杂的数据类型，其值只能通过定义在有关类型上的操作符操作（并且它的内部表示对用户是隐藏的）……换句话说，域和对象类是同样的事情！这是把这两项技术（关系和对象）结合在一起的关键。第25章将详细论述这一重要问题。

5.3 关系值

回顾第3章，发现有必要仔细区分关系值和关系变量。在这一节讨论关系值，下一节讨论关系变量。首先，这里有一个关系的精确定义：

- 给定一个集合，它包含 n 个类型或域 Ti（i=1,2,… n），这些域没有必要各不相同。r 如果包含如下定义的表头（heading）和主体（body）[⊖]，那它就是一个关系。
  a) 表头是具有 n 个形式为 Ai : Ti 的属性的集合，其中：Ai（必须各不相同）是 r 的属性名；Ti 是相应类型的名字（i=1,2,…,n）。
  b) 主体是一个包含 m 个元组 t 的集合，其中，t 依次是形式为 Ai : vi 的分量的集合，vi 是类型 Ti 的值，即元组 t 在属性 Ai 上的值（i=1,2,…,n）。

m 和 n 分别被称为关系的势和度。

由定义引发出以下的要点：

1) 按照关系的表状结构，表头是列名和相应类型名的行，主体是数据行的集合（参看下面进一步的论述）。
2) 属性 Ai 可被说成是类型 Ti 的或定义在类型 Ti 上。注意，来自同一或不同关系的不同属性可以是同一类型（参看第3章图3-8及第4章图4-5和图4-6，那里有这方面的几个示例）。
3) 在任何情况下，都会存在给定类型的某些这样的值：它们不出现在数据库的属于此类型的任何属性中。例如，P8可能是一个有效的零件号码，但它没有出现在图 3-8中。
4) 就像定义中所说的，数 n —— 关系中属性的数目——称为度（或目(arity)）。度数为1的关系称为单目关系，度数为2的关系称为双目关系，度数为3的关系称为三目关系……，度数为 n 的关系称为 n 目关系。一般地讲，对应任意的非负整数 n，关系模型被认为是带有 n 目关系值。
5) 元组有时表述为 n 元组（因此可以说4元组、5元组，等等）。然而，通常去掉前缀“ n- ”。
6) 如果说关系 r 有表头 H，即是精确地说关系 r 属于类型 RELATION{H}（类型的名字精确地称为 RELATION{H}）。进一步的内容可参考 5.4节。

让我们通过举例的方式看一下图 5-1中的表（现在有意不称它为关系）是如何对应前面的定义的。

- 首先，表有四个类型，即供应商号（ S#）、姓名(NAME)、状态值(STATUS)和城市名(CITY)。注意：当在纸上把一个关系画作一张表时，经常不特意提及上面的类型（就像在第3章中所看到的），但是必须在概念上明白：它们总是存在的。
- 其次，表有两部分 —— 它有一行列的名称，且有一个数据行的集合。首先考虑列名所在的那一行：

```
(S#, SNAME, STATUS, CITY )
```

⊖ 表头在文献中也称为(关系) (schem或scheme)。表头也称为内涵(intention)，主体是外延(extention)。

这一行实际上代表了下面有序对的集合：

```
{  S#        :  S#       ,
   SNAME     :  NAME     ,
   STATUS    :  STATUS   ,
   CITY      :  CITY     }
```

每一对的前面部分是一个属性名称，后面部分是相应的类型名称。因此，可以认为列头所在的一行实际上代表了定义中的表头。注意：就像已经指明的，实际中通常可以把表头看作是一个属性名称的集合（也就是类型名称经常省略掉），除了特别需要说明的时候。这一做法比较随便，但却方便，在下面的论述中经常采用这种方式。

- 表的剩余部分，它由一个数据行的集合组成。现在考虑集合中的一行：

```
( S1, Smith, 20, London )
```

这一行实际上代表了下面有序对的集合：

```
{ S#       :  S#( 'S1' )          ,
SNAME      :  NAME( 'Smith' )     ,
STATUS     :  20                  ,
CITY       :  'London'            }
```

每一对的前面部分是一个属性名称，后面部分是相应的属性值。在非正式的论述中，经常省略属性名称，这是因为有一个约定：表中的每一个值的属性实际上就是值所对应列的顶部出现的属性；进一步讲，值实际上属于相关的类型，即属性所属的类型。例如，值S1 —— 确切地说是S#('S1') —— 是属性S#的一个值，并且它是相关类型“供应商号码”（也可以称为S#）的一个值。因此可以说每一行实际代表了一个元组，这符合定义说明。

从上面所讲的内容，可以得出这样的结论：根据定义，图 5-1中的表可以被看作是描述关系的图片 —— 假设看待图片的方式一样（也就是说，假设对图片的解释规则一样）。换句话说，都认为其中存在一定的类型；每一列对应一个类型；每一行代表一个元组；每一个属性的值是对应的类型的值；等等。只有都基于以上的解释规则，才可以称一张表是一个合理的关系。

现在知道了一张表和一个关系并不是完全相同的事物（尽管在前面的章节中假设它们是）。确切地说，一个关系就是定义所描述的一个比较抽象的对象，一张表是建立在这一抽象事物之上的一张具体的图（经常在纸上）。它们不完全一样。当然，它们非常相像，至少在非正式的场合中通常说它们是同一事物。但是，当力求精确的时候，就不得不将它们区分开来。

注意：如果理解这一思想（关系和表的区别）有些困难，下面的内容将会有所帮助。首先，不可否认的是，关系模型有这样一个主要优点：它的基本的抽象对象 —— 关系在纸上有一个简单的表示；这个简单的表示使得关系系统比较容易使用和理解，同时对关系系统运作方式的解释变得比较容易。然而，表状的表达方式显示了许多不正确的东西。例如，它明确地显示了表的行 —— 关系的元组 —— 有着某种从上到下的次序，但实际上没有这个次序（参看下一节）。

关系值的特性

关系值具有某些特性，所有这些特性都是上节给出的关系定义的直接结果，它们都非常重要。首先简要地列出这些特性，然后再详细讨论。在任意给定的关系中：

- 没有重复的元组；
- 元组从上而下的排列没有次序；

• 属性从左至右的排列没有次序；

• 每个元组只包含每个属性的一个值。

1. 没有重复的元组

这个特性由以下的事实得出：关系的主体是一个数学集合（元组的）；数学里的集合是不包含重复元素的。

注意：实际上很明显，“重复元组”的概念没有意义。考虑一个带有属性 S#和CITY的关系（意思是供应商S#在城市CITY中）。假设关系包含一个元组，此元组表达了这样一个事实：供应商S1在伦敦。如果这个关系有此元组的一个重复值（假设是可能的），那它只是把以上的事实简单地又说了一遍。但如果事实本身就是正确的，再说一遍也不会使它更正确！

顺便说一下，第一个特性恰恰说明了关系和表不是同一事物，因为表（一般来说）可以含有重复的行 —— 如果没有规则去防止 —— 而关系不能包含任何重复的元组（因为如果一个关系含有重复的元组，那根据定义它就不是一个关系）。遗憾的是，SQL中允许表包含重复的行。这里不过多阐述这其中的原因（参看文献［5.3］和［5.6］有较全面的论述）；就目前来说，我们认定关系模型没有重复值，因此在这本书中，将确保不会出现重复的值（这一点主要针对对SQL的讨论。对关系模型则不必特别在意）。

2. 元组从上到下没有次序

这个特性也由以下的事实得出：关系的主体是一个数学集合；数学里的集合是无序的。例如，在图5-1中元组也可以以相反的次序显示，但关系仍是原来的关系。因此没有譬如“第5个元组”、“第97个元组”、“第1个元组”等类似的说法，也不能说“下一个元组”；换句话说，元组没有地址的概念，也没有“下一个”的概念。文献［5.6］在讲“没有重复的元组”的特性时已经提到了上面的观点，并说明了为何“元组没有次序”的思想也非常重要（确实，这两个特性是关联的）。

注意：在数据库和主语言（例如C或COBOL）之间的接口中确实需要“下一个”的概念（参看第4章SQL游标和ORDER BY 的讨论）。但那是主语言而不是关系模型提出的要求。实际上，主语言需要把无序的集合转换成（元组的）有序的队列或数组，这样诸如“取下一个元组”的操作才有意义。注意，这样的功能只是形成了应用程序接口的一部分，对最终用户是看不到的。

本节的前面已经说过，第二个特性也可以用来说明关系和表不是同一事物，因为表的行有自上而下的次序，而关系的元组却没有。

3. 属性从左至右没有次序

这个特性是从以下的事实而来：关系的表头是（属性的）一个集合。例如，在图5-1中，属性也可以采用SNAME、CITY、STATUS、S#的次序，而表达的仍是同一关系，至少对涉及的关系模型是如此 ⊖ 因此，没有诸如“第一个属性”、“第二个属性”等的概念，也没有“下一个属性”的概念；换言之，属性总是用名字来提及，从不用位置。这样，错误以及晦涩的程序出现的机会就减少了。例如，从一个属性到另一个属性的“flopping over（顺推）”就无法再去破坏系统。而在许多程序系统中，逻辑上分散但物理上邻接的地址可能被故意或非故意地通过各种途径破坏。

属性次序的问题也揭示了关系的具体表现即表，显示了不正确的东西：表的列很明显有从左到右的次序，但关系的属性没有。

⊖ 数学关系，不像关系模型中关系的属性，有从左到右的属性次序。

4. 每个元组只包含每个属性的一个值

这个特性从元组的定义直接而来——一个元组就是n个分量或形如$Ai:vi$ (i=1,2,…, n)的有序对的集合。一个满足这个特性的关系称为是规范化的，也就是满足第一范式。

注意：这个特殊的特性实际上是很明显的，特别是自从所有的关系按照定义规范化后！然而，这个特性有几个重要的结论，参见：(a) 参考文献［5.10］的注解，(b) 第18章有关丢失信息的部分，(c) 接下来的章节。

关系值属性

在5.2节中说过，构成类型的值可以是任何种类的。因此，可以构造一种特殊的类型，它的值是关系——从而可以产生这样一种关系：它的属性的值是关系。换言之，即关系之间是可以嵌套的。例如图 5-3所示的关系 S_SP；它的属性 PQ是关系型的。供应商 S5供应的零件的一个空集用一个空的集合表示（准确地说，被一个空的关系表示）。

S_SP

S#	SNAME	STATUS	CITY	PQ
S1	Smith	20	London	P# QTY: P1 300; P2 200; .. ...; P6 100
S2	Jones	10	Paris	P# QTY: P1 300; P2 400
..	.....	..	.....	.....
S5	Adams	30	Athens	P# QTY:

图5-3 具有关系型属性的关系

这里提及关系型属性的问题的主要原因是：过去，大多数关系数据库文献（包括本书的早期版本）声称这样的可能性是不合法的（现在，它们中的大多数仍这样认为）。例如，下面是本书前期版本的一段摘录：

> 注意所有列值是不可分的……也就是说，在表的每一行和每一列的交叉处只能有一个数据值，绝不能是一组值。因此，在表 EMP中，有

DEPT#	EMP#
D1	E1
D1	E2
..	..

而不是

DEPT#	EMP#
D1	E1,E2
..	..

> 在第二张表中列 EMP#就是一个通常称为“重复组”(repeating group）的例子。一个重复组就是一列……它在一行中含有多个值（通常，不同的行有不同数量的值)。

关系数据库不允许重复值；上面的第二张表在关系系统中是不允许的。

在同一本书的后面又写到：

［域］只含有不可分的数据项……［因此，］关系不含有重复值。满足这个条件的关系称为是规范化的，或者说符合第一范式……在有关关系模型的书籍中所说的关系总是规范化的。

上面的说法是不正确的（至少不是全部正确）：它们产生于作者对类型和谓词本质的误解。原因将在第11章（11.6节）讨论，这些错误不会在实际中引起严重的错误；但无论如何，仍然要对那些被误导的读者道歉。前一个版本中关于“关系模型中的关系总是规范化的”的说法是正确的。进一步的讨论，请参看第11章。

关系值和它们的解释

总结这一节如下：正如第3章3.4节中所说的：（a）任何给定关系的表头可看作是一个谓词；（b）关系的元组可以被认为是对谓词的真命题，它通过替换谓词的合适参数和占位符而得到（“实例化谓词”）。实际上，封闭世界假说 (也称封闭世界解释)认为，如果另外一个有效的元组 —— 也就是符合关系表头的元组 —— 没有出现在关系的主体中，那我们认为相应的命题是错误的。意思就是关系的主体包含了所有且仅包含对应于真实命题的元组。

5.4 关系变量

回顾第3章可以知道：关系变量分两种类型：基本关系变量和视图（也分别称为实的和虚的关系变量）。在本节中，只考虑基本关系变量；视图将在第9章中讨论。

1. 基本关系变量的定义

下面是定义一个基本关系变量的语法：

```
VAR <relvar name>BASE <relation type >
                <candidate key definition list>
               [<foreign key definition list>];
```

*<relation type>*采用如下形式

```
RELATION {<attribute commalist>}
```

其中每一个*<attribute>*以如下有序对的形式出现：

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

关于*<candidate key definition list>*和可选项*<foreign key definition list>*的说明见下面。

注意：术语逗号列表在第4章4.6节中已定义。假如*<xyz>*代表了一个任意的语法类别（就是出现在BNF产生式规则的左边的项）。于是，表达式*<xyz list>*代表了零个或多个*<xyz>*的序列，其中每对相邻的*<xyz>*之间用一个或多个空格隔开。

通过例子顺便给出供应商和零件数据库（图3-9）基本关系变量的定义：

```
VAR S BASE RELATION
    { S#      S#,
      SNAME   NAME,
      STATUS  INTEGER,
      CITY    CHAR }
    PRIMARY KEY { S# } ;

VAR P BASE RELATION
    { P#      P#,
```

```
        PNAME    NAME,
        COLOR    COLOR,
        WEIGHT   WEIGHT,
        CITY     CHAR }
      PRIMARY KEY { P# } ;

VAR SP BASE RELATION
      { S#       S#,
        P#       P#,
        QTY      QTY }
      PRIMARY KEY { S#, P# }
      FOREIGN KEY { S# } REFERENCES S
      FOREIGN KEY { P# } REFERENCES P ;
```

解释：

1) **这三个基本关系变量的（关系）类型如下：**

```
RELATION { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR }

RELATION { P# P#, PNAME NAME, COLOR COLOR,
                              WEIGHT WEIGHT, CITY CHAR }

RELATION { S# S#, P# P#, QTY QTY }
```

注意这些关系类型中的每一个实际上就是一个类型，并且可以在所有的正常场合中使用——特殊地，可以作为关系属性的类型（实际上，“RELATION**”（类型）是一个类型发生器，它允许定义任意的关系类型，例如：在传统的程序设计语言中，“**array**”(数组)是一个类型发生器）。**

2) **给定关系变量的所有可能的关系值，无论是基本的或其他的，都属于同一个关系类型（即在关系变量定义中直接或间接指定的关系类型），并且因此而具有同一个表头。特别地，任意给定的基本关系变量的初始值是相应关系类型的一个空的关系。**

3) **已经为关系值定义的术语**heading**（表头）、**body**（主体）、**attribute**（属性）、**tuple**（元组）、**cardinality**（势）和**degree**（度），也能通过合理的方式应用于关系变量（所有的关系变量，不仅是基本的）。**

4) **对一个新定义的基本关系变量，系统会在目录中生成一个描述此关系变量的条目。**

5) **候选码的定义将在第**8**章中详细介绍。在此之前，简单地假定每个基本关系变量的定义只包含一个如下特殊形式的定义：**

```
PRIMARY KEY {<attribute name commalist>}
```

6) **外码的定义也将在第**8**章中介绍。**

最后，给出删除一个基本关系变量的语法：

```
DROP VAR <relvar name>;
```

这一操作首先把某基本关系变量的值置为一个空的关系（不严格地说，就是删掉了基本关系变量中的所有元组），然后删掉基本关系变量在目录中的条目。于是这个关系变量对系统来说就是不可知的了。注意：为了简单起见，假定如果待删除的关系变量正在其他的地方被使用（例如，它被别处的视图定义所引用），那么DROP**操作会失败。详见第**9**章的讨论。**

2. 关系变量更新

关系模型包括关系赋值操作（即修改），以便给关系变量赋值（特指基本关系变量）。下面是 Tutorial D**语法：**

```
<relvar name>:=<relational expression>;
```

对<*relational expression*>**计算求值，求得的关系结果赋给** <*relvar name*>**指定的关系变量，以**

代替变量中以前的值。当然，关系变量和涉及到的关系必须是同一类型（也就是具有相同的表头）。

举例，假设有一个与供应商关系变量S类型相同的关系变量R：

```
VAR R BASE RELATION
    { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR } ... ;
```

下面是几个合法的关系赋值：

```
• R  :=  S ;
• R  :=  S WHERE CITY = 'London' ;
• R  :=  S MINUS ( S WHERE CITY = 'Paris' ) ;
```

这些例子中的每一个可以被认为（a）检索了式子右边指定的关系；（b）修改了式子左边指定的关系变量。

现在修改一下第二和第三个例子，用关系变量S替换左边的关系变量R：

```
• S  :=  S WHERE CITY = 'London' ;
• S  :=  S MINUS ( S WHERE CITY = 'Paris' ) ;
```

可以看到这两个赋值都有效地改变成了关系变量S——一个删除了所有不在伦敦的供应商，另一个删除了所有在巴黎的供应商。为了方便，Tutorial D支持显式的插入、删除以及修改操作，但是每个操作的定义是为了某个关系赋值的方便。这里有几个例子：⊖

```
• INSERT INTO S
  RELATION { TUPLE { S# S# ( 'S6' ),
                     SNAME NAME ( 'Smith' ),
                     STATUS 50,
                     CITY 'Rome' } } ;
```

赋值等价于

```
S  :=  S UNION
       RELATION { TUPLE { S# S# ( 'S6' ),
                          SNAME NAME ( 'Smith' ),
                          STATUS 50,
                          CITY 'Rome' } } ;
```

顺便说一下，如果对应于供应商S6的元组已经出现在元组变量S中，赋值就会成功。在实际中，可能希望通过扩展INSERT的赋值操作来确保此事不会发生；然而为简单起见，我们忽略了这一点。类似的事情同样适用于DELETE和UPDATE。

```
• DELETE S WHERE CITY = 'Paris' ;
```

它等价于以下赋值：

```
S  :=  S MINUS ( S WHERE CITY = 'Paris' ) ;
```

```
• UPDATE S WHERE CITY = 'Paris'
                 STATUS := 2 * STATUS,
                 CITY   := 'Rome' ;
```

这一等价的赋值在UPDATE操作中会稍微复杂一点，在这里省略了细节。参看文献[3.3]。

下面给出了INSERT、DELETE以及UPDATE操作语法的一个简单的小结，以供参考：

```
• INSERT INTO <relvar name> <relational expression> ;
• DELETE <relvar name> [ WHERE <boolean expression> ] ;
```

⊖ 在INSERT例子中的表达式 RELATION{…}是一个关系选择子调用（实际上，括号里面的表达式TUPLE{…}是元组选择子调用，在里面括号中的表达式是标量选择子调用）。参考第6章6.3节和6.4节。

- UPDATE <relvar name> [WHERE <boolean expression>]
  <attribute update commalist> ;

<*attribute update*> 采用如下的形式：

```
<attribute name> : =<expression>
```

<*boolean expression*>的语法无需再说明，不过会在第6章中给出详细说明。

在结束这一节之前，强调一点：关系的赋值（包括INSERT、UPDATE和DELETE）都是集合级的操作。例如，（不严格地讲）UPDATE修改了目标关系变量的一个元组的集合。在非正式的情况下，经常说修改某个元组，但是一定要明白：

1) 实际上是在谈论修改一个元组的集合，而这个集合的势恰好是1；

2) 有时，修改一个势为1的元组的集合是不可能的！

例如，假设供应商元组变量满足完整性约束（参考第8章），即供应商S1和S4必须具有相同的状态值。于是，任何企图改变这两个供应商之一的状态值的单元组修改必然要失败。相反两者必须同时修改，就像如下所示：

```
UPDATE S WHERE S# = S# ( 'S1' ) OR S# = S# ( 'S4' )
         STATUS := some value ;
```

不得不承认，“修改一个元组”的说法是相当草率的。元组就像关系值一样是值的概念，它们不能被修改（按照定义，改变一个值是不能做到的）。为了能够“修改元组”，需要引进元组变量的概念，这个概念不属于关系模型的范畴！因此在说“修改元组 *t*”的时候，实际的意思是用一个元组替换了另一个元组。谈及“修改属性 *A*”（在元组中）也有类似的情况。在本书中，将继续采用“修改元组”或“修改元组的属性”的表述方式——这样在实际中是很方便的，但是必须明确：这样的用法只是为了简便，它在某种程度上是不严格的。

5.5 SQL的支持

与前面章节的内容有关的SQL语句如下：

```
CREATE DOMAIN     CREATE TABLE     INSERT
ALTER DOMAIN      ALTER TABLE      UPDATE
DROP DOMAIN       DROP TABLE       DELETE
```

INSERT、UPDATE和DELETE在第4章中已经讨论过了。下面看一下其余的几个语句。

1. 域

正如第4章中所说的，SQL中域的概念与真正的关系中域（类型）的概念大不相同；实际上，这两个概念相去甚远，在SQL中使用别的名字会更好些。SQL中域的主要目的只是简单地允许通过在基本表定义中给一个内置类型起一个名字，用于简化某些列的定义。这里有几个例子：

```
CREATE DOMAIN S# CHAR(5) ;
CREATE DOMAIN P# CHAR(6) ;

CREATE TABLE S  ( S# S#, ... ) ;
CREATE TABLE P  ( P# P#, ... ) ;
CREATE TABLE SP ( S# S#, P# P#, ... ) ;
```

下面列出了真正的域和SQL中的域之间的区别，以供参考：

- 正如已经说过的，SQL的域实际上只是一个符合语法的简写；它们根本不是真正的数据类型，并且不是用户定义的类型。
- SQL域中的值不能有任意的内部复杂性，它们受复杂性的约束，这些约束就是内置类

型。

- 一个SQL的域必须根据一个内部类型来定义，而不能是其他的用户定义的域。
- 实际上，一个SQL的域必须只能根据一个内部类型来定义。因此，如果要在SQL中定义类似于5.2节中的POINT和LINESEG的域是不可能的。
- 一个SQL的域不能有超过一个的可能表示。实际上，SQL的域不能正确区分类型和（物理）表示。例如，SQL的域S#和P#通过类型CHAR定义，于是SQL会允许在供应商号码和零件号码上执行字符串操作。
- SQL域中没有强类型，并且只有极少的类型正确性的检查。例如，前面给出了S#和P#的定义，下面的SQL操作会安全通过类型检查，尽管这是不应该的：

```
SELECT *
FROM   SP
WHERE  S# = P# ;
```

注意：或许可以通过另一个途径来说明这件事情，即SQL仅仅支持8个真正的关系域，这就是下面8个“基本类型”：

- 数字
- 字符串
- 位串
- 日期
- 时间
- 时间戳
- 年/月间隔
- 天/时间间隔

可以说类型检查是执行了，但只是在这8个类型的基础上。因此，如果把一个数字和一个位串进行比较，是非法的；但比较两个数字却是合法的，即使这两个数字代表不同的事物——例如分别是INTEGER和FLOAT。

- SQL不允许用户在一个给定的域上定义操作符。相反，精确地说，可用的操作符只能是那些作用在相应的表示上的内置操作符。

下面是SQL的CREATE DOMAIN语句的语法：

```
CREATE DOMAIN <domain name> <builtin type name>
            [ <default spec> ]
            [ <constraints> ] ;
```

解释：

1) 第4章4.2节列有合法<*builtin type name*>的清单。
2) 有些域没有明确的缺省值（参考下一节），任选项<*default spec*>为定义在这些域上的每一列指定了缺省的值。它的形式是“DEFAULT <*default*>”，其中<*default*>是一个文字、无参（niladic）内置操作符或空值。[⊖] 注意：一个无参操作符就是不带操作数的操作符（CURRENT_DATE就是一个例子）。
3) 任选项<*constraint*>指定作用在相关域上的完整性约束。在第8章将会讨论这方面的问题。

一个已存在的SQL域可以在任何时候利用ALTER DOMAIN来进行修改。并且，ALTER

[⊖] “SQL支持空值对”的详细讨论见第18章。

DOMAIN允许为已存在的或将被删除的域定义一个新的 <*default spec*>。同时也允许为已存在或将被删除的域定义一个新的完整性约束。这些问题的细节非常复杂且已超出了本书的范围；有关问题可以参考［4.19］。

最后，一个已存在的域可以通过命令 DROP DOMAIN被删除掉，语法是：

```
DROP DOMAIN <domain name><option>;
```

其中<*option*>可以是RESTRICT或CASCADE。总的思想如下：

- 如果指定了RESTRICT，并且待删的域被别的地方使用，则 DROP失败。
- 如果指定了CASCADE，则DROP会成功，并且通过各种“级联”(cascade)方式。例如，定义在域上的列会被认为是直接定义在域的数据类型上。

再次声明一遍，因为涉及到的细节非常复杂，这里忽略不讲。若想进一步了解，请参考［4.19］。

2. 基本表

在详细讲解基本表之前，对SQL表有如下两点说明：

- 第一，SQL表允许包含重复的行。因此它们没有必要有主码（或任何侯选码）。
- 第二，SQL表的列有从左至右的次序。例如，在供应商表 S中，列S#可能是第一列，SNAME可能是第二列，等等。

现在讨论基本表：基本表通过语句 CREATE TABLE来定义（注意：这里的TABLE特指基本表，ALTER TABLE和DROP TABLE中的TABLE也是基本表）。语法如下：

```
CREATE TABLE <base table name>
          ( <base table element commalist> ) ;
```

其中，<*base table element*>是一个<*column definition*>或一个<*constraint*>。<*constraint*>是指作用在有关基本表上的完整性约束；详细的讨论将在第 8章进行。<*column definition*>（至少有一个）一般采用如下形式：

```
<column name> <type or domain name> { <default spec> }
```

这里的<*type or domain name*>或者是一个内部类型名，或者是一个域名，任选项 <*default spec*>为列指定一个缺省值，应用中它的优先级高于域级指定的任何缺省值。注意： <*default spec*>定义一个缺省值，或简称缺省，如果用户在 INSERT中没有为某列提供一个明确的值，则定义的缺省值会自动赋给该列。在第 4章4.6节的“不包含游标的操作”中举有例子。如果一个给定的列没有自己的缺省值，并且没有从它所依赖的域上继承一个值，那它就隐含地有缺省值NULL —— NULL是“缺省的缺省值”。

有关CREATE TABLE的例子，参看第4章图4-1。

一个已存在的基本表可以在任何时候通过 ALTER TABLE来修改。修改命令支持以下的操作：

- 增加一个新列；
- 可以为已存在的列定义一个新的缺省值（如果原先有缺省值，则被替换）；
- 删除现存列的缺省值；
- 删除现存的列；
- 可以指定一个新的完整性约束；
- 删除现存的完整性约束。

下面只给出第一种情况的例子：

```
ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1;
```

这一行语句给供应商基本表增加了一个DISCOUNT列（整型）。于是此表由4列扩展到5列，第5列的初始值都是-1。

最后，一个存在的基本表可以通过DROP TABLE来删除，语法是：

```
DROP TABLE <base table name><option>;
```

其中<*option*>是RESTRICT或CASCADE（与DROP DOMAIN中相同）。如果指定了RESTRICT，并且基本表在视图定义或完整性约束中被使用，则删除失败；如果指定了CASCADE，则删除会成功（删除了表和其中的所有行），并且任何相关的视图定义和完整性约束也会被删除。

5.6 小结

在本章中，对关系模型的各部分有了一个完整的理解。一个域就是一个数据类型（可能是内置的或系统定义的，更一般的是用户定义的）；它提供了：（a）一个值的集合（相关类型的所有可能的值），不同关系中的不同属性从中取出各自的实际值；（b）一个操作符的集合（包括只读的和可修改的），供相关类型上的值和变量进行操作。一个给定类型上的值可以是任何种类的——数字、字符串、日期、时间、音频记录、地图、视频记录、几何点，等等。类型约束着操作，一个给定操作上的操作对象必须符合操作上的正确类型（强类型）。强类型是一个非常好的设想，因为它能使某些逻辑错误被捕捉到，并且是在编译的时候而不是运行的时候。注意对关系操作（尤其是连接、并等）来说，强类型有重要的含义，在下章中将会讨论到。

类型可以是标量的或者是非标量的。标量类型就是没有用户可见分量的类型。关系模型中最重要的非标量类型是关系类型（看下一段），它通过关系类型发生器（RELATION type generator）定义。我们严格区分类型和它的物理表示（类型是一个模型问题，物理表示是实现问题）。然而，我们规定每个标量类型至少有一个可能表示。每个可能表示产生如下的自动定义：（a）一个选择子操作符，和（b）对应可能表示的每个分量的THE_操作符（包括THE_伪变量）。我们支持显式的类型转换，但不支持隐含的强制转换。同时支持对标量类型定义任意数目的操作符，并在每个类型上定义相等操作符（按照有关语义）。

现在转到关系：我们区分关系值和关系变量。一个关系有两个部分：表头和主体；表头是属性的集合，主体是与表头对应的元组的集合。表头中属性的数目叫做度，主体中元组的数目称为基数或势。一个关系有一个关系类型：即类型RELATION{*H*}，其中*H*是相应的表头。一个关系可被看作是一张表，表的列代表属性，行代表元组，但这种表示只是近似的。同时，所有的关系满足四个重要的特性：

- 不含有重复的元组；
- 元组从上到下没有次序；
- 属性从左至右没有次序；
- 每个元组的每个属性只有一个值（也就是说，所有的关系是规范化的，或者说满足第 1 范式）。

从第3章可知，关系的表头可认为是一个谓词，而主体可看作是一个真命题，这通过用正

确类型的值替换谓词的占位符或参数来实现。如果一个有效的元组（即符合关系的表头）没有出现在关系的主体中，根据封闭世界假设，可以说相应的命题是错误的。

因为一个类型的值可以是任意种类的，因此带有关系型属性的关系是合法的（在第 6章和第11章将会看到，这一点是很有用的）。

关系变量分两种类型：基本关系变量和视图。前面已经知道了怎样用 Tutorial D定义基本关系变量，也知道了怎样为这样的关系变量的属性定义类型。在本书中将用 Tutorial D语言进行举例说明。

注意：读者可能已经发现书中只讨论了标量类型上的用户操作符，而没有在关系类型上讨论。原因是大多数的关系操作符 —— 选择、投影、连接、关系赋值等 —— 实际上已经内置在模型中，不再需要用户定义（而且，因为这些操作能应用于所有类型的关系，所以它们是通用的，这只是不严格的说法）。然而，如果系统提供了定义的方法，就没有理由不去扩展这些操作。

最后，概括一下 SQL中有关（SQL意义下的）域和基本表的定义的方法。特别地，我们指出：

- SQL 中的域不是类型；
- SQL中的表（基本的或其他的）不是关系，因为（ a）它们允许重复的行；（b）它们的列有从左到右的次序。实际上，它们甚至可以有两个或多个相同名字的列（对基本表或视图名，SQL中是不允许重名的）。例如，考虑下面例子中通过 SQL查询得到的表：

```
SELECT *
FROM  S,  P ;
```

练习

5.1 根据第3章图3-6中给定的部门和雇员数据库：

a) 按照本章中正式的关系的形式为目录中的各成员重新命名。

b) 目录的结构怎样根据类型（域）去扩展？

c) 针对扩展的目录写一个查询，查询包含类型为 EMP#的属性的所有关系变量的名字。

d) 在一个没有用户定义类型的 SQL系统中，怎样去实现 c？

5.2 目录关系变量本身定义在何种域之上？

5.3 给定基本关系变量 PART_STRUCTURE（参看第4章图4-6）

a) 用本章所讲的 Tutorial D写出一个关系变量定义和适当的类型定义。

b) 假设这个关系变量包含在练习 5.1的部门和雇员数据库中，为反映出对 a的答案，写出系统必须在目录上的修改。

c) 用 Tutorial D写出合适的 DROP操作，以撤消 b中对目录所做的修改。

5.4 利用本章所介绍的 Tutorial D的知识，为图 4-5中供应商-零件-工程项目数据库，写出一个合适的定义（参看第4章练习）。

5.5 在5.2节曾经指出：“某种货物的数量（ quantity）”这种说法从严格意义上来讲是不正确的（数量是类型 QTY的一个值，不是类型 INTEGER的值）。因此，图 4-5是相当草率的，它假设了把数量看作整数是正确的。在练习 5.4的基础上，给出正确引用图 4-5中各种标量值的途径。

5.6 根据练习5.4的答案，判断下列标量表达式哪些是正确的？对于正确的，说明结果的类型；其他的，给出产生预期效果的合法表达式。

a) `J.CITY = P.CITY`

b) `JNAME || PNAME`

c) `QTY * 100`

d) `QTY + 100`

e) `STATUS + 5`

f) `J.CITY < S.CITY`

g) `COLOR = P.CITY`

h) `J.CITY = P.CITY || 'burg'`

5.7 给标量类型CIRCLE一个合理的类型定义。何种选择子和THE_操作符能应用于这个类型？并且（a）定义一个只读操作符的集合，用来计算直径、圆周和给定圆的面积；（b）定义一个更新操作符，对一个给定圆的半径加倍（严格地说是修改一个给定的CIRCLE变量，使半径变成原先的两倍）。

5.8 域和类型有时可理解为变量，如同关系变量。例如，合法的雇员编号有可能从三位扩展到四位，因此，可能需要修改“所有雇员编号的集合”。对此进行讨论。

5.9 一个关系定义为一个属性的集合和一个元组的集合。数学中空集是相当重要的；实际上，若结果和定理等对*n*个元素的集合是正确的，则经常期望当*n*=0时也正确。一个关系的元组集合可以是空集吗？或属性集可以是空集吗？

5.10 一个关系变量有时好像就是一个传统的计算机文件。只是元组代替了记录，属性代替了字段。对此进行讨论。

5.11 我们已经知道，数据定义操作使目录被修改。但目录只是一个关系变量的集合，可不可用通常的更新操作符INSERT、UPDATE和DELETE来适当修改目录？讨论。

5.12 写出Tutorial D中DROP操作的过程，这个操作删除所有用户在供应商-零件-工程数据库中创建的信息。

参考文献和简介

下面的大多数文章都讲了关系模型的各个方面，而不只是结构方面。

5.1 E. F. Codd: “A Relational Model of Data for Large Shared Data Banks,” *CACM 13*, No. 6 (June 1970). Republished in *Milestones of Research-Selected Papers 1958-1982* (*CACM 25th Anniversary Issue*), *CACM*26, No. 1 (January 1983). See also the earlier version “Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks,” IBM Research Report RJ599 (August 19th. 1969). *Note*: That earlier version was Codd's very first publication on the relational model.

尽管30年过去了，它仍经得起再次阅读。当然，自从首次发表后，里面的许多思想已经得到进一步精炼，但其中的大多数变动是在原基础上的发展，而不是对原来的否定。实际上，这篇文章里还有许多想法至今仍没有去研究。

谈一个术语的问题。这篇文章中，Codd使用了术语“time-varying-relation”，而不是我们所说的“关系变量（relation variables）”。但“time-varying-relation”并不是一个好的表述。首先，relations是值，它不简单地随时间而变动（vary with time）。其次，如果

在某种程序设计语言中说：

```
DECLARE N INTEGER;
```

不称N为一个随时间而变的整数（time-varying integer），而叫它一个整型变量。因此，在本书中使用了关系变量，没用“ time-varying-relation ”；然而，大家至少要知道有这种说法存在。

5.2 E. F. Codd: *The Relational Model for Database Management Version* 2. Reading, Mass.: Addison-Wesley (1990).

在20世纪80年代后期，Codd花了他的大部分时间来修订和扩展最初的模型（他现在称之为“关系模型版本1”或RM / V1），这本书就是他劳动的成果。里面描述了关系模型版本2（RM / V2）。RM / V1和RM / V2的本质区别是：RM / V1为数据库的一个特殊领域规划了一个抽象的蓝图（主要是基础领域），而RM / V2则是对整个系统规划了一个蓝图。因此，RM / V1只包含三个部分——结构、完整性和操作；RM / V2则有十八个部分，不仅包含了最初的三部分，还包含了目录、授权、命名、分布式数据库和数据库管理的许多其他方面。下面列出了所有的十八个部分，以供参考：

A 授权	M 操纵
B 基本运算符	N 命名
C 目录	P 保护
D DBMS设计原理	Q 量词
E DBA的命令	S 结构
F 函数	T 数据类型
I 完整性	V 视图
J 指示器	X 分布式数据库
L 语言设计原理	Z 高级运算

然而这本书的思想并没有被全部接受[5.7~5.8]。这里只谈一点。本章中我们说域之间是不允许比较的，拿供应商和零件来说，S.S# = SP.P#是无效的，因为比较的双方类型不同；因此，试图通过匹配供应商和零件号码来连接供应商和发货（shipment）的努力就会失败。Codd就此提出了一种关系代数操作DCO（domain check override，域检查替换），允许不同类型的值之间进行比较。例如，对于上面提到的连接，尽管S.S#和SP.P#的类型不同，但连接是基于匹配表示，而不是匹配类型。

但这里面存在一个问题。整个DCO思想建立在混淆了类型和表示的基础之上。对域类型的辨认给出了我们想要的域检查和类似于DCO的功能。例如，下面的表达式在供应商号和零件号之间构建了一个表示级的比较：

```
THE_S# ( S# ) = THE_P# ( P# )
```

（两个操作对象都是CHAR类型）。因此，要求5.2节讨论的机制能给出所有想要的功能，并且要清晰，不混乱。现在来说，没有必要加进类似于“ DCO连接 ”的新操作符以增加混乱。

5.3 Hugh Darwen: “The Duplicity of Duplicate Rows,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

这篇文章进一步支持了[5.6]中的观点：禁止重复的行。该文不仅新颖地说明了这个

观点，而且提出了许多新的思想。它强调一个基本观点：讨论两个事物是否是重复的，实质上是要对所讨论的事物有一个明确的相等判断标准。

5.4 Hugh Darwen: "Relation-Valued Attributes," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

5.5 Hugh Darwen: "The Nullologist in Relationland," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

按照Darwen的说法，空值理论(Nullology)就是对"无的研究"(study of nothing)，即对空集的研究。集合在关系理论中无所不在，一个很重要的情况是：如果集合是空的，结果会怎样。这是经常要碰到的基本问题。

到本章为止，与上述思想直接有关的是该书的第2节("没有行的表")和第3节("没有列的表")。同时参看练习5.9的答案。

5.6 C. J. Date: "Why Duplicate Rows Are Prohibited," in *Relational Database Writings 1985-1989*. Reading. Mass.: Addison-Wesley (1990).

该书为"禁止重复行"的观点用例子做了广泛的论证。特别地，该书指出了重复的行对优化构成重要障碍（第17章）。同时参考[5.3]。

5.7 C. J. Date: "Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/Vl)." in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

总结和批评了Codd的RM / V1，并提供了一个可选择的定义。在转向"第2版"之前，完善"第1版"是很重要的。注意：本书描述的关系模型建立在该书"重新构建"的版本之上（[3.3]中有更进一步的叙述）。

5.8 C. J. Date: "A Critical Review of the Relational Model Version 2 (RM/V2)," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

总结和批评了Codd的RM / V2[5.2]。

5.9 C. J. Date: " 30 Years of Relational" (series of twelve articles), *Intelligent Enterprise 1*, Nos. 1 -3 and 2, Nos. 1 -9 (October 1998 onward). *Note*: Most installments after the first are published in the onlne portion of the magazine at *www.intelligententerprise.com.*

Codd在20世纪70年代发表的作品对关系作出了贡献，这些文章对此进行了仔细的、公正的回顾和评价。具体地说，它们详细地检查了下面的论文（也涉及了其他的一部分）：

- Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks (the first version of reference [5.1]);
- A Relational Model of Data for Large Shared Data Banks [5.1];
- Relational Completeness of Data Base Sublanguages [6.1];
- A Data Base Sublanguage Founded on the Relational Calculus [7.1];
- Further Normalization of the Data Base Relational Model [10.4];
- Extending the Relational Database Model to Capture More Meaning [13.6];
- Interactive Support for Nonprogrammers: The Relational and Network Approaches [25.8].

5.10 Mark A. Roth, Henry F. Korth, and Abraham Silberschatz: "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS 13*, No. 4 (December 1988).

多年来，有的学者提出了NF^2关系，其中NF^2=NFNF=“non first normal form”(非第一范式形式)。实际上，关于NF^2没有一个准确的概念，比较让人接受的是下面这样理解：设*nr*是一个NF^2关系，设*nr*中属性*A*的类型为*T*。于是*nr*的一个元组*t*可以包含属性*A*上的任意多个值，不同的元组所含的数目可以是不同的 ⊖ (因此属性*A*是一个重复组（repeating group）属性 —— 参见本书前一版中的有关论述，及本章 5.3节引述)。注意“关系”*nr*因此而不是规范化的（它的每个元组在每个属性上的值不只一个）；实际上，就关系模型来说，它根本就不是一个关系。尽管如此，该文还是讨论了 NF^2的思想。具体来说，它对NF^2关系定义了一个关系演算和一个关系代数（第 6、7章)，并且证明了两者的等价性。它同时给出有关这个领域中的其他工作的大量参考文献。

部分练习答案

5.1 a）明显的变化归纳如下：

原来的	新的
TABLE	RELVAR
COLUMN	ATTRIBUTE
TABNAME	RVNAME
COLCOUNT	DEGREE
ROWCOUNT	CARDINALITY 缩写为(CARD)
COLNAME	ATTRNAME

同时，关系变量需要一个额外的属性，上面的值用来指明相应的关系变量是基本关系变量还是视图。目录结构因此有如下形式：

RELVAR	RVNAME	DEGREE	CARD	RVKIND	...
ATTRIBUTE	RVNAME	ATTRNAME	.....		

b) 我们需要一个新的包含每一个类型条目的目录关系变量（TYPE)，同时在ATTRIBUTE关系变量中增加一个新的属性（TYPENAME)，用以指出关系变量中每个属性的类型。目录结构因此有如下形式：

TYPE	TYPENAME	.....			
RELVAR	RVNAME	DEGREE	CARD	RVKIND	...
ATTRIBUTE	RVNAME	ATTRNAME	TYPENAME	.....	

作为一个辅助的练习，你可以去思考在主码和外码方面目录会有哪些变化。

c) `( ATTRIBUTE WHERE TYPENAME = NAME ( 'EMP#' ) ) { RVNAME } ;`

注意这里的NAME选择子调用（见练习5.2的答案)。

d) 在一个不支持用户定义类型的系统上，不可能去目录中查询这种类型！—— 只能够对属性进行查询。一般来说，如果可能的话，把属性定义成和类型相同的名字是比较好的，或至少包含类型的名字作为属性名称的一部分。如果数据库定义者遵循这样的规

⊖ 有些作者不承认这个定义，他们把 NF^2 关系定义为包含至少一个关系属性的任何关系。我们有自己的理由认为这样的关系实际上是第一范式。

则，则这样的属性查询可能能解决问题。

5.2 这个问题不可能给出一个确定的答案。下面是一些建议：

TYPENAME	*is defined on domain*	NAME
RVNAME		NAME
ATTRNAME		NAME
DEGREE		NONNEG_INTEGER
CARD		NONNEG_INTEGER
RVKIND		RVKIND

这些域假设定义如下：

- NAME是一个包含所有合法名字的集合。
- NONNEG_INTEGER是包含所有非负整数的集合。
- RVKIND是集合{“Base”,“View”}。

注意：这里偏离了刚才所说的原则！——这个原则是：如果可能的话，属性和相应的类型应该用相同的名称。这个练习说明了，如果关系变量在类型固定之前定义，就会发生偏离（这针对所有的关系变量，而不仅是目录）。

5.3

a)
```
TYPE P#  POSSREP ( CHAR ) ;
TYPE QTY POSSREP ( INTEGER ) ;

VAR PART_STRUCTURE BASE RELATION
     { MAJOR_P# P#,
       MINOR_P# P#,
       QTY      QTY }
     PRIMARY KEY { MAJOR_P#, MINOR_P# } ;
```

b) 只列出了目录条目（图5-4）。注意关系变量RELVAR的CARD值也需要增加一个。图中PART_STRUCTURE的基数是0而不是7（尽管图4-6中，显示为7），因为关系变量刚建的时候假设为空。

TYPE

TYPENAME	.....
P#	.....
QTY	.....

RELVAR

RVNAME	DEGREE	CARD	RVKIND	...
PART_STRUCTURE	3	0	Base	...

ATTRIBUTE

RVNAME	ATTRNAME	TYPENAME	.....
PART_STRUCTURE	MAJOR_P#	P#	.....
PART_STRUCTURE	MINOR_P#	P#	.....
PART_STRUCTURE	QTY	QTY	.....

图5-4 关系变量PART_STRUCTURE的目录内容

c)
```
DROP VAR PART_STRUCTURE ;
DROP TYPE QTY ;
DROP TYPE P# ;
```

5.4
```
TYPE S#     POSSREP ( CHAR ) ;
TYPE NAME   POSSREP ( CHAR ) ;
TYPE P#     POSSREP ( CHAR ) ;
TYPE COLOR  POSSREP ( CHAR ) ;
TYPE WEIGHT POSSREP ( RATIONAL ) ;
TYPE J#     POSSREP ( CHAR ) ;
TYPE QTY    POSSREP ( INTEGER ) ;

VAR S BASE RELATION
     { S#     S#,
       SNAME  NAME,
       STATUS INTEGER,
```

```
        CITY    CHAR }
      PRIMARY KEY { S# } ;

VAR P BASE RELATION
      { P#      P#,
        PNAME   NAME,
        COLOR   COLOR,
        WEIGHT  WEIGHT,
        CITY    CHAR }
      PRIMARY KEY { P# } ;

VAR J BASE RELATION
      { J#      J#,
        JNAME   NAME,
        CITY    CHAR }
      PRIMARY KEY { J# } ;

VAR SPJ BASE RELATION
       { S#      S#,
         P#      P#,
         J#      J#,
         QTY     QTY }
       PRIMARY KEY { S#, P#, J# }
       FOREIGN KEY { S# } REFERENCES S
       FOREIGN KEY { P# } REFERENCES P
       FOREIGN KEY { J# } REFERENCES J ;
```

5.5 对每个属性给出了一个典型的值。首先，关系变量 S

```
S#      :  S# ( 'S1' )
SNAME   :  NAME ( 'Smith' )
STATUS  :  20
CITY    : 'London'
```

关系变量P：

```
P#      :  P# ( 'P1' )
PNAME   :  NAME ( 'Nut' )
COLOR   :  COLOR ( 'Red' )
WEIGHT  :  WEIGHT ( 12.0 )
CITY    : 'London'
```

关系变量J：

```
J#      :  J# ( 'J1' )
JNAME   :  NAME ( 'Sorter' )
CITY    : 'Paris'
```

5.6 a) 合法；布尔型（BOOLEAN）。

b) 不合法；NAME (THE_NAME (JNAME) || THE_NAME (PNAME)。注意：这里的思想是连接字符串表达式，然后把结果转换回类型 NAME。当然，如果连接的结果不能转换为合法的名称，转换就会出现类型错。

c) 合法：QTY。

d) 不合法：QTY+QTY(100)。

e) 合法：STATUS。

f) 合法：BOOLEAN。

g) 不合法：THE_COLOR(COLOR)=P.CITY。

h) 合法。

5.7

```
TYPE LENGTH POSSREP ( RATIONAL ) ;
TYPE POINT  POSSREP ( X RATIONAL, Y RATIONAL ) ;
TYPE CIRCLE POSSREP ( R LENGTH, CTR POINT ) ;
            /* R is (the length of) the radius of the circle */
            /* and CTR is the center                         */
```

应用于类型CIRCLE的唯一的选择子是：

```
CIRCLE ( r, ctr )
/* returns the circle with radius r and center ctr */
```

THE_操作符为：

```
THE_R ( c )
/* returns the length of the radius of circle c */

THE_CTR ( c )
/* returns the point that is the center of circle c */
```

a)
```
OPERATOR DIAMETER ( C CIRCLE ) RETURNS ( LENGTH ) ;
   RETURN ( 2 * THE_R ( C ) ) ;
END OPERATOR ;

OPERATOR CIRCUMFERENCE ( C CIRCLE ) RETURNS ( LENGTH ) ;
   RETURN ( 3.14159 * DIAMETER ( C ) ) ;
END OPERATOR ;

OPERATOR AREA ( C CIRCLE ) RETURNS ( AREA ) ;
   RETURN ( 3.14159 * ( THE_R ( C ) ** 2 ) ) ;
END OPERATOR ;
```

假设：(a) length乘上一个整数或有理数，结果是 length；(b) length乘length，结果是面积（area）(其中AREA是一个用户定义的类型)。

b)
```
OPERATOR DOUBLE_R ( C CIRCLE ) UPDATES ( C ) ;
   BEGIN ;
      THE_R ( C )  :=  2 * THE_R ( C ) ;
      RETURN ;
   END ;
END OPERATOR ;
```

5.8 下面是有关的论述。首先，定义一个类型的操作并没有创造一个值的集合；那些值已经存在了。因此，所有的类型定义操作符实际上是引进了一个供集合里的值引用的名称。同样，DROP TYPE也没有真正删除有关的值，它只是删去了 TYPE引进的名称。“修改一个存在的类型”意味着先删除存在的类型名称，然后再重新定义这个名称，使它指向一个不同的集合。“alter type”(更改类型）是上述过程的一个简写。

5.9 元组为空集的关系是很正常的（类似于没有记录的关系)。就像5.4节所讲的，每个基本关系变量都以这样的关系作为初始值。通常指这样的关系为空关系，尽管有点不精确。

虽然可能不很明显，但属性集为空的关系也是合理存在的！事实上，这样的关系非常重要——就像空集之于集合理论的重要性，或零之于算术的重要性。

为了略微仔细地说明这个思想，首先必须考虑一个没有属性的关系是否能包含元组。事实上，这样的关系至多能包含一个元组——即0元组（没有分量的元组）；它的元组数不能超过一个，因为0元组相互之间是重复的。因此，度是零的关系只有两个，其一是包含一个元组，另一个根本没有元组。这两个关系如此重要，以至于它们有了昵称（根据 Darwen[5.5]）——称第一个为TABLE_DEE，第二个为TABLE_DUM，或缩写为DEE和DUM（DEE是只有一个元组的关系，DUM是空元组)。

现在不是深入讨论这个问题的时机；只说明一点就够了：这两个关系非常重要的原因之一是DEE对应于true（或yes)，DUM对应于false（或no)。第6、8章的练习和答案给了更深入的探讨。[5.5]中有进一步的讨论。

5.10 可以认为元组类似于记录（具体的值，不是类型)，属性类似于字段（类型，不是具体值)。然而，这些对应只是近似的。一个关系变量不应该被看作只是一个文件，它更像是一个具有约束的文件（看 5.3节“关系的特性”)。即它对用户来说具有简洁的数据结构，处理数据的操作符是简单的，有用户接口。

5.11 原则来说是可以的，通过 INSERT、UPDATE和DELETE操作符可以修改目录。然而，

允许这样的操作有着潜在的危险，破坏（无意或有意）系统正常工作所需的目录信息就此而变得非常容易。例如，假设DELETE操作

```
DELETE RELVAR WHERE RVNAME = NAME('EMP') ;
```

在部门和雇员目录上被允许。结果是从RELVAR关系变量中移去了对关系变量EMP的描述。就这个系统来说，关系变量EMP从此不存在了——即系统不再知道这个关系变量的信息。因此，接下来所有存取此关系变量的努力都会失败。

大多数实际产品对在目录上的UPDATE、DELETE和INSERT操作，或者（a）根本不允许（通常情况下）；或者（b）只授权给级别非常高的用户（或许只给DBA）；目录的修改通过数据定义语句来进行。例如，定义关系变量EMP会产生：（a）在关系变量RELVAR中，为EMP做一个条目;（b）在关系变量ATTRIBUTE中增加四个条目，对应EMP的四个属性（然而这也会引起一系列其它的问题，这里与用户无关）。因此，定义一个新的事物（如新的类型或基本关系变量）某种程度上就是对目录进行INSERT操作。同样，DROP类似于DELETE；SQL提供了不少ALTER语句（如ALTER TABLE），这样就可以通过多种方法来修改目录条目，ALTER类似于UPDATE。

注意：目录也包含了目录变量本身的条目。然而，这些条目不是通过数据定义操作来创建的。它们在系统安装过程中自动创建。

5.12
```
DROP VAR SPJ, S, P, J ;

DROP TYPE S#, NAME, P#, COLOR, WEIGHT, J#, QTY ;
```

假设对DROP VAR和DROP TYPE作了扩展，从而允许把几个变量和类型通过一次操作删除。