附录B SQL3 概 览

B.1 引言

在本书出版之际，SQL3有可能会成为标准（“SQL/99”）。然而，本书对SQL的讨论并不是建立在SQL3的基础之上的，部分原因已经在第4章中给出解释，另外还有部分其他的原因：那就是SQL3有时非常的麻烦。但是，至少要有一个对SQL3的主要特征的概述，因此本书给出了这个附录。

SQL3几乎包括SQL/92的所有的内容，当然除了一些小的、已经删除的下列特征（在本书中对这些特征没有进行讨论）：SQLCODE，无符号整数（而不是ORDER BY子句中的列名）、标识符的介绍、用户定义的一些字符集、校勘和翻译，以及一些其他的内容。本书主要介绍那些1992年以来增加的特征，为了方便起见，将这些特征称为“SQL3”。在这些新特征中，最明显的是用户定义的数据类型和与其相关的问题，在B.2~B.5中将对这些内容进行讨论。其他主要的特征将在B.6节中做简要的探讨。注意：要了解B.2~B.5主题的详细内容可参见[3.3]。

在详细介绍SQL3之前，首先解释一下SQLJ[4.6]。“SQLJ”是为了综合SQL和Java而提出的一个计划，很多大家熟悉的开发商参与了这个计划。该计划的第0部分解决如何在Java程序中处理嵌入式SQL；第1部分处理如何从SQL激活Java，如，调用一个用Java写的存储过程；第2部分解决如何将Java类作为SQL数据类型，如，将SQL表中定义的列作为基础。这些活动跟SQL3中的技术是不一样的，但是，SQLJ的第0部分已经作为包括10个部分的SQL标准草案[4.22]的第一部分发布（至少在美国已经发布），而且SQLJ的第1部分和第2部分也会在SQL3正式发行之后相继发布。

一个SQL/CLI的最新版本也可能会在这个时间出现。

下面是对SQL3的一些说明：首先，虽然“#”不是SQL3的一个有效字符，但是却经常在例子中的列名前使用，本书中的其它部分也是这样使用的；使用“；”作为语句结束符，以前的例子中也是这么使用的。第二，接下来的讨论将非常的彻底。最后，需要说明，到SQL3批准之时，某些细节肯定还要有所改变。

B.2 新数据类型

正如在前面的小节中所说的，SQL3的最明显的特征是可以处理数据类型[⊖]。这些数据类型有新的可嵌入的类型，更确切地说，是新的可嵌入的标量类型；还有新的可嵌入的类型生成器，SQL3称之为类型构造器；另外，CREATE TYPE语句允许用户定义自己的类型。当然，还有相应的DROP TABLE语句允许用户删除自定义的类型。下面依次介绍这些特征。

⊖ 增加一个几乎看不见的SQL类型的“域”（见第4章和第5章）是非常有诱惑力的，虽然它一般情况下会被忽视掉。

嵌入的标量类型

SQL3可以支持的三种新的标量类型：

- BOOLEAN：BOOLEAN是一个真值类型；支持一般的布尔操作符 NOT、AND和OR，一般来说，标量表达式可以出现的位置就可以写布尔表达式。然而，需要注意： SQL支持3个真值，而不是像第 18章中所说的是两个，对应的关键字是 TRUE、FALSE和UNKNOWN。然而事实却是： BOOLEAN类型仅包括两个值，而不是三个值；因为UNKNOWN对应的是null。例如，一个类型为BOOLEAN的变量若赋值为UNKNOWN，实际上是对其赋值为null，这是非常不正确的。这是一个严重的错误，跟“需要赋值为 0的numeric类型却赋值为null”的错误是一样的。

  SQL3的另一个特殊之处是，它不能在附录 A中所说的<condition primary>中引用一个简单的布尔变量。例如，如果 B是一个BOOLEAN型的变量，那么一个 WHERE B的WHERE子句就是不正确的。

  SQL在增加了BOOLEAN类型的同时，还增加了两个新的聚集操作符： EVERY（而不是ALL）和ANY。这两种操作的参数都是一个BOOLEAN类型的列，该列的BOOLEAN值通常是由一个表达式得到的，例如，在WHERE子句中的WHERE ANY(QTY>200)。如果列为空，两个操作符都返回unknown(或是null)[⊖]；如果列为非空，那么当列中的每一个值都为true时，EVERY返回true；否则，返回false。当列中的每一个值都是 false时；ANY返回false；否则为true。对于其他的SQL聚集操作符，在聚集操作符之前将去掉所有的空值。
- CLOB（character large object）：这种类型可以是长度没有限制的变长字符串。对应的是定位机制，该机制跟游标机制有些相似，可以将许多一般的字符串操作符不能处理的字符串分开处理。可以支持的有“ = ”和LIKE。
- BLOB（binary large object）：跟字符串的的机制类似，但是现在是“八位字节”的串，而不是字符串。

生成的数据类型

SQL3的类型生成器有： REF，ARRAY和ROW。然而，定义“ REF类型”的唯一方法却是隐含的，当通过CREATE TYPE定义“结构类型”时，就定义了一个 REF类型。因此此处略去对 REF类型的讨论。而 ARRAY和ROW类型实际上并不能真正定义，因为没有单独的CREATE ARRAY TYPE和CREATE ROW TYPE语句；在调用CREATE TABLE语句内置的相关类型的生成器时，将使用这些类型。例如，下面是一个 ARRAY类型使用的示例：

```
CREATE TABLE SALES
     ( ITEM# CHAR(5),
       QTY   INTEGER ARRAY [12],
     PRIMARY KEY ( ITEM# ) ) ;
```

这里的QTY列是array值的；一个QTY的值是一个包括12个元素的元组，其中的每一个元素都是INTEGER类型的。注意：SQL数组只能是一维的，而且数组中的元素不能是数组。

下面是一个在SALES表中的查询：

```
SELECT ITEM#
FROM   SALES
WHERE  QTY [3] > 100 ;
```

⊖ 跟all-or-any条件的行为相比较(见附录A)。

该例可“取出三月份的销售额超过100的记录的记录号”，注意下面的参考书目。下面的例子可以往表中插入一行：

```
INSERT INTO SALES ( ITEM#, QTY )
VALUES ( 'X4320',
         ARRAY [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ;
```

(注意语句中的array)。

ROW类型是类似的。例如：

```
CREATE TABLE CUST
     ( CUST# CHAR(3),
       ADDR  ROW ( STREET CHAR(50),
                   CITY   CHAR(25),
                   STATE  CHAR(2),
                   ZIP    CHAR(5) )
     PRIMARY KEY ( CUST# ) ) ;
```

查询示例：

```
SELECT CUST#
FROM   CUST
WHERE  ADDR.STATE = 'CA' ;
```

插入示例：

```
INSERT INTO CUST ( CUST#, ADDR )
VALUES ( '001', ROW ( '1600 Pennsylvania Ave.',
                      'Washington', 'DC', '20500' ) ) ;
```

DISTINCT类型

使用新的CREATE TYPE 语句可以定义一个用户自定义的数据类型：该数据类型或者是DISTINCT类型，或者是结构类型。此处没有将在前面已经介绍过的产生的数据类型作为用户自定义的数据类型。在这一小节，只讨论 DISTINCT类型。一个 DISTINCT类型（将DISTINCT大写，是为了强调它是一个特殊的专用词）是一个受限的用户定义的特殊的类型。特别地，它在实际执行时必须包含一个内置的标量类型。下面是定义 DISTINCT类型的语法：

```
CREATE TYPE <type name>
    AS <builtin scalar type name> FINAL
     [ <cast options> ]
     [ <method specification commalist> ] ;
```

下面是一个例子：

```
CREATE TYPE WEIGHT AS NUMERIC(5, 1) FINAL;
```

说明：

1) WEIGHT类型继承了可以应用于基本数据类型（例如，NUMERIC类型）的操作符。然而，需要注意，WEIGHT只能与同类型的数据值具有可比性，而不能与其他的任何数据类型的数据相比。例如，如果WT是一个WEIGHT类型的SQL变量，下面的比较就是合法的：

```
Wt>14.7
```

然而，如果假设已经设定了适当的 <case option>（忽略细节），那么下面的比较也是合法的：

```
WT>CAST(14.7 AS WEIGHT)
CAST(WT AS NUMERIC)>14.7
```

而且，这两个CAST类型可以分别简略为 WEIGHT（14.7）和NUMERIC（WT）类型。注意，这里的函数名WEIGHT和NUMERIC是通过类型定义器在<case option>中生成的，

而不能通过被定义的类型的名字或基本类型的名字来标识。

2) 赋值运算的解释也是类似的。例如，WEIGHT类型的值只能赋到一个WEIGHT类型的变量中，而且其他类型的数据不能赋到这里面。

3) WEIGHT类型不能自动继承其他的基本数据类型的运算。然而，<method specification>（例子中没有）可以通过类型定义器定义在WEIGHT类型的变量和值上使用的“方法”，这可参见下面的部分。还可以定义在WEIGHT类型的值和变量上操作的SQL程序和函数。例如，可以定义一个名为ADDW的函数，该函数可将两个重量相加的结果放到第三个值中，因此允许用户使用这样的表达式：

```
ADDW(WT1, WT2)
ADDW(WT1, WEIGHT(14.7))
```

4) FINAL标识写在最后。这可参见下面的小节。

结构类型

另一种用户定义的数据类型是结构类型。下面是几个例子[⊖]：

```
CREATE TYPE POINT AS (X FLOAT, Y FLOAT) FINAL;
CREATE TYPE LINESEG AS (BEGIN POINT, END POINT) FINAL;
```

说明：

1) POINT类型有两个属性X和Y（不要跟在本书的第二部分中的元组和关系属性相混淆）；同样，LINESEG类型有BEGIN和END两个属性。每个属性可为任意数据类型。

2) 然而，在定义结构的数据类型时提到的这种属性构成了该类型的值的物理实现，而不是像第5章中所说的这只是一个“可能的表述”。这样，在这个例子中的点就是根据它们的笛卡儿积物理实现的。

3) 每定义一个属性时，会自动定义一个observer（观察运算符）和一个mutator（变化运算符），不很严格地说，可以分别称之为取值（get）和赋值（set）。这样就需要在语法上使用点这个符号[⊜]。例如，假定Z、P和LS分别是SQL的FLOAT、POINT和LINESEG类型的变量。下面的表达都是合法的。

```
P.X;                    /*取得点P的X部分的值*/
LS.BEGIN.X;             /*取得线段LS的起始点的X部分的值*/
SET P.X=Z;              /*将点P的X部分的值赋给Z*/
SET LS.BEGIN.X=Z        /*将线段LS的起始点的X部分的值赋为Z*/
```

4) 因为没有定义任何附加的操作符，所以这些数据类型唯一的其他操作符是等于操作符和赋值操作符，这两种操作符可以适用于任何的数据类型。特别注意，POINT和LINESEG这两个选择符（见第5章）不是自动定义的，因此没有POINT和LINESEG这两个单词的出现。

5) FINAL标识符表示要定义这种类型的任何一个子类型都不会成功（见B.3）；这些类型都是叶类型，而且以后也只能作为叶类型。

6) 所有的POINT和LINESEG类型（或者一般的结构类型）都是封装的吗？这个问题的回答依赖于具体的内容。例如，当一个结构类型用做某列的类型时，答案就是yes。然而，

⊖ 实际上，第二个例子是不能正确执行的，因为BEGIN和END都是保留字。

⊜ 注意，此处的“mutator”并不是第5章中所说的真正的mutator（例如，一个更新操作）。因为它定义要返回一个值。这些内容超出了本附录的讨论范围。详细内容可参见[3.3]。

当用做某表的类型时（参见B.4部分），答案就是no。

注意：之所以在第一种情况说“是……的一种类型”的原因是，即使在这种情况下，get和set操作符也能按照已经声明的类型有效地显示其属性，因此这两种操作符还不能忽略。也许在第一种情况下使用“假封装”或者“伪标量”更合适。

下面是定义结构类型的一般的语法，该结构类型不是一个子类型（参见B.3子类型部分的介绍）：

```
CREATE   TYPE<type name>
         AS (<attribute definition commalist>)
            [<ref type implementation>]
            [[NOT] INSTANTIABLE]
            [NOT] FINAL
            [<method specification commalist>]
```

说明：

1) <attribute definition commalist>不能为空。

2) <ref type implementation>是可选的，将在B.4中介绍。

3) NOT INSTANIABLE表示这个类型是一个假类型。见第19章。缺省值是INSTANTIABLE。

4) NOT FINAL的意思是该类型可以有适当的子类型。FINAL则表明不能有子类型。

5) 对于一个给定的结构类型*T*，可以使用的操作符有：

   a. 上面介绍过的属性的observer（观察运算符）和mutator（变化运算符）；

   b. 赋值操作符“=”和可能的“<”(所有语句的最后两个操作符是通过单独的CREATE ORDERING FOR *T*定义的，但是至于为什么要通过排序来定义“=”还很不清楚)；

   c. 需要使用*T*类型或者*T*类型的子类型（参见B.3节）的数据做参数的程序和函数；

   d. 将*T*类型或任何*T*的子类型（参见B.3）作为特殊的目标参数的方法。注意：这里的“方法”是指传统的实体意义（见第24章）上的方法，即，这些操作符将一个参数特殊对待。如果方法*M*是定义在类型*T*上的，而且*X*是类型*T*的一个表达式，那么表达式*X*.*M*就是激活*X*上的*M*方法，为了简单起见，此处假定*M*没有参数。

   使用<method specification>可以定义第19章和第24章中提到的“描述签名”。然而，<method specification>中也包括了许多其他的定义，这些定义有很多是可执行的，但是却没有放到适当的位置（按照作者的观点来看）。执行该方法的实际代码在其他地方定义。

B.3 类型继承

因为SQL3仅适用于结构类型，所以SQL3的类型继承不是很直接。例如，它不支持嵌入的类型、生成的类型[⊖]或DISTINCT类型的继承。另外，SQL3也不支持多继承。在这一节中，只考虑“伪标量”结构类型的类型继承，“伪标量”类型可以被看做是其他的相应的类型；在B.5节中将介绍没有封装的结构类型。

第19章中的类型继承模型和SQL3的“伪标量”类型继承之间，在逻辑上的最大不同之处在于：

⊖ 不包括可能的行类型。在行类型的继承上有很多问题，但是因为这个问题超出了讨论的范围，在此不详加介绍。

- SQL3除了支持行为继承外，还支持结构继承。
- SQL3不能很充分地区分值和变量；特别地，也不能区分值的替换和变量的替换。
- SQL3不能支持类型约束，因此也就不能支持约束定义的功能。
- SQL3使用更新操作符或变化操作符mutator来实现无条件的继承。

因为这些不同，SQL3允许“非环状的循环”和类似的情况（这些介绍可参见第19章）。这些不同还导致了有些应该赋值为true的SQL3的比较赋值了false。例如，某一个有相等半轴的ELLIPSE类型在现实世界中就是一个圆。

下面显示了如何在SQL3中表示椭圆和圆：

```
CREATE TYPE ELLIPSE
    AS (A LENGTH, B LENGTH, CTR POINT)
       NOT FINAL;
CREATE TYPE CIRCLE UNDER ELLIPSE
    AS (R LENGTH)    ——这是不现实的！（可参见下面的解释）
        NOT FINAL;
```

注意：按照CIRCLE类型的定义，其物理实现包括四个部分：从ELLIPSE继承而来的A、B和CTR，以及只在CIRCLE类型中声明的R。当然，对于任一给定的圆，这四部分中有三部分是有相同的值。换一种定义方法，为类型CIRCLE定义一个方法R，而不是属性R，则CIRCLE类型就跟类型ELLIPSE有了相同的物理实现，这样，该物理实现就包含了尽可能少的冗余。值得注意的是，如果C是声明的CIRCLE类型的一个变量，那么C.R的赋值对第一种定义来说是合法的；但是对于第二种定义来说就是非法的。于是，如果支持对C.R的赋值，那么，一般来说，该赋值就会产生一个C.R的圆，而不是C.A的圆。

下面是在第19章中所说的类型继承模型和SQL3中所支持的类型继承模型之间的不同之处：

- SQL3使用“直接(direct)”（就像是用直接的子类型）代替“立即(immediate)”。
- SQL3使用“最大超类型(maximal supertype)”代替“根类型(root type)”。
- SQL支持类似于TREAT DOWN的TREAT操作符。例如，SQL3分析器对TREAT_DOWN_AS_CIRCLE（E）的分析就是TREATE AS CIRCLE。
- SQL3还支持“TREAT UP”操作符，例如，考虑下面的操作符：

```
AREA (C AS ELLIPSE)
```

即使变量C声明的是CIRCLE类型，但是“AS ELLIPSE”的说明使得AREA操作符的ELLIPSE版本被激活。值得注意的是“TREAT UP”这个操作符只能在一定的上下文中使用。特别地，可以像上面的例子那样用“TREAT UP”操作符来处理一个参数，使其成为用户定义的一个操作符。需要说明的是提供这种功能有可能会使用户混淆模型和其物理实现，因为用户没有必要知道AREA有两种定义。

- SQL3还支持*X*.SPECIFICTYPE形式的方法，该方法将字符串参数*X*转化为某一个特定的类型返回。
- SQL3的IS_*T*(*X*)和IS_MS_*T*(*X*)类似：

```
X IS OF (T)
X IS OF (ONLY T)
```

B.4 参考类型

在第24章曾经说到实体系统，该系统只包含了一个称为适当的数据类型支持的好的概念

(如果将类型继承单独算的话，也可以说是两个好的概念)。可以看到，虽然SQL3对这个概念中提到的一些好的功能的支持还很不理想，但是它确实包括了其中很多好的功能。然而，SQL3还包含了很多相关的非常不好的功能。实际上， SQL3非常类似地处理第一类根本性错误（将表和类等同）和第二类根本性错误（混淆了指针和表），这样做的目的是非常不明确的，至少对作者来说是这样；因此，与其说SQL3是一种比较模糊的概念，还不如说它是“像对象”的。

尽管如此，本节还要在下面对其相关特征的实际情况做简要的解释。首先， SQL3允许定义基表，这些基表不仅可以是一般的列的集合，而且还可以是用户定义的类型。例如：

```
CREATE TYPE DEPT_TYPE
   AS (DEPT# CHAR(3),
       DNAME CHAR(25),
       BUFGET MONEY)
REF IS SYSTEM GENERATED;..
CREATE TABLE DEPT OF DEPT_TYPE
       (REF IS DEPT_ID SYSTEM GENERATED,
       PRIMARY KEY(DEPT#),
       UNIQUE (DEPT_ID))...;
```

说明：

1) 给定了结构类型*T*的定义，系统自动生成一个相关的名为 REF（*T*）的REF类型的参考类型。类型REF（*T*）的值就是一些基表 ㊀ 中的类型为*T*的行的参照，而这些基表的类型已经定义为*T*（见下面的第3点）。注意：*T*也可以在其他上下文中使用，如，可以作为某些变量*V*或某些列*C*的类型。然而，REF(*T*)的类型的值却不可以这么使用。

2) 在CREATE TYPE语句中的REF IS SYSTEM GERERATED表示跟REF类型相关的实际的值是系统生成的。其他的选项，例如， REF IS USER GENERATED也是可用的，但是在此处忽略这些细节。此处， REF IS SYSTEM GENERATED是默认的。

3) 可以通过CREATE TABLE定义基表为某一结构类型中的一个，即 OF。然而，此处的“是……的一种”(“OF”)用得不是很恰当，因为实际上， 基表不是这种类型的，而且它的行也不是这种类型的 ㊁ 。特别地，可以在表中定义一些附加列，这些列可以是结构类型。实际上，至少要声明这样的一列，也就是说，必须有一列是 REF类型的。定义这样的列的语法不同于定义一般的列，其语法是：

```
REF IS <column name> SYSTEM GENERATED
```

这多余的列用来标识基表各行的唯一的 ID。上面的例子中明确地指明了列是唯一的，起始UNIQUE(<column name>)也可以是隐含的。某一给定行的ID是在该行插入时赋值的，一直到该行 ㊂ 删除时才删除该ID。注意：此处SYSTEM GENERATED是需要重复的。虽然要进行特殊的考虑，但是，在一个 INSERT或UPDATE操作中，SYSTEM GENERATED也可以是一个目标列，此处忽略细节说明。在这一点上需要多做一点说明，为了得到“唯一的ID”的功能，此处首先考虑的是将表定义为某结构类型的一种，

㊀ 或者有可能是视图，视图情况下的细节超出了本书的范围。

㊁ 因此要特别注意，如果某个操作符*OP*的参数*P*的声明类型是结构类型*T*，那么，该基表的一行定义为“OF”类型*T*的基表的行就不能相应地激活*OP*操作符。

㊂ “该行”只能表示为“有特殊ID的行”。注意不要混淆值和变量。

而不是用一般的方法定义一个适当的列。但是这种选择折衷方案的原因还不是很清楚。

4) 按照B.2节所说的，当一种结构类型如 DEPT_TYPE作为定义基表的基础时，虽然在上下文中有时候认为它是封装的，但是在本书中却认为这种类型不是封装的。例如，在上例中表DEPT有四列，但是如果认为DEPT_TYPE进行了封装，该表就仅有两列了。

5) 前面已经知道， DEPT表的主码是DEPT#。假定DEPT的行有唯一的ID（“ 参照 ”），那么，在需要的情况下，就可以将其作为主码，如下：

```
CREATE TABLE DEPT OF DEPT_TYPE
       (REF IS DEPT_ID SYSTEM GENERATED;
       PRIMARY KEY(DEPT_ID);
       UNIQHE (DEPT#))...;
```

下面扩展该例，并给出另外一个基表 EMP，如：

```
CREATE TABLE EMP
       (EMP#        CHAR(5),
       ENAME        CHAR(25),
       SALARY       MONEY,
       DEPT_ID      REF(DEPT_TYPE) SCOPE DEPT
                    REFERENCES ARE CHECKED
                    ON DELETE CASCADE,
       PRIMARY       KEY (EMP#));..
```

一般来说，基表 EMP将包含一个外码列 DEPT#，该列参照的是部门中的部门号。此处有一个参照列DEPT_ID，该列号没有很明确地声明为外码列，但是通过“ references ”参照了部门。SCOPE DEPT指出了参照表。REFERENCES ARE CHECKED则表示可支持参照完整性。REFERENCES ARE NOT CHECKED允许虚参照；但是为什么要标识这个选项却不是很明确。ON DELETE……声明了一个删除规则，该删除规则跟一般的外码删除规则类似（支持相同的选项）。然而，需要注意，如果基表 DEPT 中的DEPT_ID列是不能更新的（见上面的第 3点），那么也就没有相应的ON UPDATE规则。

考虑这个数据库的示例查询。首先用 SQL3实现“ 查找雇员E1所在的部门号 ”功能：

```
SELECT EX.DEPT_ID⊖DEPT# AS DEPT#
FROM EMP EX
WHERE EX.EMP#='E1';
```

注意SELECT子句中逆参照的操作 ⊖ ，表达式EX.DEPT_ID→DEPT#返回DEPT_ID值所在的DEPT行中的DEPT#的值。还需要注意该子句中有时还需要 AS标识，如果省略了AS标识，结果列将给出一个“ 执行依赖 ”的名字。值得注意的是该查询的执行效率可能要低于用 SQL所写的查询的执行效率，因为 SQL查询中只存取一个表，而不是存取两个表。之所以说明这一点，是因为使用参照在一般情况下都会提高执行能力。

顺便说一下，如果查询已经得到雇员 E1的部门（而不仅是部门号），则逆参照的操作就不一样了：

```
SELECT DEREF (EX.DEPT_ID) AS DEPT
FROM   EMP EX
WHER   EX.EMP#='E1'
```

⊖ 大部分支持逆参照操作的语言支持参照操作，但是 SQL3不支持。

进一步地讲，显式的DEREF将返回一个封装的值，而不是一个行值。

下面给出另外一个例子：列出部门 D1中的所有的雇员号（注意 WHERE子句中的逆参照操作）：

```
SELECT EX.EMP#
FROM   EMP EX
WHERE  EX.DEPT_ID -> DEPT# = 'D1' ;
```

下面是一个INSERT的例子（一个职工记录的插入）

```
INSERT INTO EMP ( EMP#, DEPT_ID )
       VALUES ( 'E5', ( SELECT DX.DEPT_ID
                        FROM   DEPT DX
                        WHERE  DX.DEPT# = 'D2' ) ) ;
```

B.5 子表和超表

考虑下面的结构类型：

```
CREATE TYPE EMP_TYPE
  AS ( EMP# ..., DEPT# ... )
     REF IS SYSTEM GENERATED
     NOT FINAL ... ;

CREATE TYPE PGMR_TYPE UNDER EMP_TYPE
  AS ( LANG ... )
     NOT FINAL ... ;
```

注意，PGMR_TYPE没有REF IS...子句，它直接从它的直接子类型 EMP_TYPE中继承一个子句。

考虑下面的基表的定义：

```
CREATE TABLE EMP OF EMP_TYPE
     ( REF IS EMP_ID SYSTEM GENERATED,
       PRIMARY KEY ( EMP# ),
       UNIQUE ( EMP_ID ) ) ;

CREATE TABLE PGMR OF PGMR_TYPE UNDER EMP ;
```

注意基表PGMR的定义中对UNDER EMP的规定，也要注意该基表定义中省略了REF IS...、PRIMARY KEY和UNIQUE的规定。基表 PGMR和EMP分别是一个子表和相应的直接超表；PGMR在继承了 EMP的列的基础上增加了一个 LANG列。因此，程序不可能只在 EMP表中有一行，而是在两个表中都有一行。因此， PGMR表中的一行必在EMP表中有对应，反之则不然。

这些表中的数据操作符的行为如下：

- SELECT：在EMP表中执行正常的SELECT。只有当PGMR中除LANG列之外还真正含有EMP的列时，SELECT才起作用。注意：在一个 <table expression>中使用ONLY限定词可以排除该表的子表中有参照的那些行。例如，只有 EMP表中的行在PGMR中没有参照时，SELECT ...FROM ONLY EMP操作才起作用。
- INSERT：正常地向EMP中插入。正常地向 PGMR中插入新行，会在 EMP和PGMR中都插入新行。
- DELETE：在EMP中执行DELETE则会从EMP和(当这些行碰巧对应程序员时)PGMR中都删除一行。在PGMR中执行DELETE，也会在EMP和PGMR中都删除。
- UPDATE：当通过PGMR更新LANG列时，只需要更新 PGMR表。但是当通过 EMP和PGMR更新其他的列时，会更新两个表。

注意一些前面没有明确提到的内容：

- 假设雇员Joe成为了一个程序员。如果只简单地将Joe插入到PGMR中，系统就会试图在EMP中也插入Joe的一行，当然，这种试图会失败。正确的做法应该是，必须从EMP中删除Joe的行，然后在PGMR中插入某一合适的行。
- 相反，假定雇员Joe停止了程序员的工作。必须将Joe的相应记录要么从EMP中删除，要么从PGMR中删除。无论指定的是哪个表，结果都是在两个表中都删除记录。然后往EMP中插入一个合适的行。

所有这些跟类型继承有什么关系呢？可以看到，答案是没有什么关系。没有什么关系是指子类型和它的直接父类型没有关系。当然这不包括SQL3中分别定义结构类型时，需要一个子表和该子表的直接父表的的情况。至于说为什么在SQL3中会这样，原因还不是很清楚。可以看到在这个安排中没有替换，还要看到包括的类型没有很明确地“封装”。

那么提出子表和子表的父表的概念带来的是什么呢？现在看来，至少在模型[⊖]方面是没有什么用处的。如果子表和其父表作为一个单独的表存在在磁盘中，很明显可以看出执行的效率可以提高。但是，这样的考虑在模型方面没有任何帮助。换句话说，父子表关系必须依赖于“父子”结构类型的原因还不清楚，而且SLQ3支持这个特征的原因也是不清楚的。

B.6 其他特征

创建表

SQL3支持CREATE TABLE中的LIKE选项。这个选项允许一个新创建的基表的某些或者所有的列可以从已经存在的表中得到。需要注意，并不是任意的表的表达式都是可以的。

表表达式

在第5章中描述了WITH子句；该子句的目的是为一定的表达式定义快捷名称。SQL3有类似的构造功能。但是它仅限于用在表的表达式中。下面是一个例子：

```
WITH LONG_TERM_EMPS AS
   ( SELECT *
     FROM   EMP
     WHERE  DATE_HIRED < DATE '1980-01-01' )

SELECT EMP#, ( LONG_TERM_CT - 1 ) AS #_OF_FELLOW_LONG_TERM_EMPS
FROM   LONG_TERM_EMPS AS L1,
     ( SELECT DEPT#, COUNT(*) AS LONG_TERM_CT
       FROM   LONG_TERM_EMPS
       GROUP  BY DEPT# ) AS L2
WHERE  L1.DEPT# = L2.DEPT# ;
```

（取出那些在1979年或者更早的时候来到公司的雇员的雇员号及其同部门的其他这样的雇员总数。）

SQL3的WITH子句可专门用在递归查询中。例如，给定表PARENT_OF，下面的递归查询返回以（*a*，*b*）成对的人的列表，（*a*，*b*）表示*a*是*b*的前驱。ANCESTOR_OF的定义中参照了ANCESTOR_OF它自己。

```
WITH RECURSIVE ANCESTOR_OF ( ANCESTOR, DESCENDANT )
AS ( SELECT PARENT, CHILD
```

[⊖] 或许应该告诉读者本书在这个问题上更倾向的方法：充分利用视图（参见13.5节末尾的例子）。

```
        FROM    PARENT_OF
        UNION
        SELECT A.PARENT, P.CHILD
        FROM    ANCESTOR_OF AS A, PARENT_OF AS P
        WHERE   A.CHILD = P.PARENT )

SELECT *
FROM    ANCESTOR_OF ;
```

条件表达式

SQL3提出了一个新的DISTINCT条件来检验两行是否相同，不要将其与UNIQUE条件（附录A）相混淆。假定要检验的行是*Left*和*Right*；*Left*和*Right*必须包括相同数量(设为*n*)的组成部分。假定*Left*和*Right*的第*i*个部分分别为*Li*和*Ri*，而且*Li*的类型和*Ri*的类型匹配，那么当且仅当对所有的*i*来说，*Li*=*Ri*或者*Li*和*Ri*均为空时，表达式将返回false；否则返回true。换句话说，当且仅当*Left*和*Right*不是重复的时候，它们才是不同的。注意DISTINCT条件不可能得到unknown的值。

SQL3还提供了一个新的SIMILAR条件，该条件是用于字符串匹配的，跟LIKE类似。即，检验一个给定的字符串是否与先前给定的模式相符合。SIMILAR和LIKE的不同之处是前者支持的可能性的范围更大（“通配符”）。语法是：

从本质上来说，此处对<pattern>和<escape>的要求跟LIKE中的类似，但是<pattern>可支持更多的附加的特殊字符，除了LIKE中的“_”和“%”之外，还支持“*”、“+”、“-”和其他的一些字符。其目的是尽可能地支持能在正规语言中出现的表达式的字符。注意：SIMILAR的很多规则来自POSIX的规则。

在本小节的结束部分，特别强调，给定已经存在的新的嵌入类型 BOOLEAN，条件表达式是一种真正的特殊类型的标量表达式，它们也确实是一致的。

完整性

SQL3支持RESTRICT<referential action>，它跟NO ACTION（见第8章中两者的不同的介绍）类似，但是不相同。SQL3还支持触发过程；特别地，SQL3还有CREATE TRIGGER语句，可以使用该语句定义一个触发器，即，可以定义一个事件和行为的组合。但要注意：

- 一个事件是指在某一个给定的表上的 INSERT、UPDATE（某些列是可选的）或者DELETE操作。
- 一个行为是指当这些事件发生前后要执行的动作，它实际上是一个程序。

更确切地说，行为包含可选的条件表达式（默认为true），当事件发生时，当且仅当条件为true时，这个SQL程序才会执行。用户可以定义该动作什么时候执行，可以是每一个事件发生的时候触发，也可以是该事件所联系的表中的每一行触发该行为。另外，还可以指明是在事件发生前还是事件发生后触发该行为，这样就可以支持简单的转换的约束。

视图更新

SQL3扩展了SQL中对视图更新的支持，包括支持“ UNION ALL ”视图，以及一对一和一对多连接视图。对游标也做了同样的扩展。

事务管理

SQL3增加了几个新的事务管理特征：

- START TRANSACTION语句，这跟第14章中的SET TRANSACTION的操作数一样。
- DECLARE CURSOR中的WITH HOLD选项（见第14章）。
- 支持保存点（参考书目[14.11]中的注释）的概念。

安全性

SQL3支持某些特定列的SELECT优先权，SELECT(*x*)允许在一个<table expression>中参照某一个给定名称的表的某一特定的列 *x*。SQL3还支持用户定义的角色，ACCG就是一个很好的例子，ACCG表示会计系的每一个人。另外，一旦创建，角色就可以授权，如同用户 ID一样。而且这些角色还可以将权限授给其他的用户 ID或其他的角色。

空缺信息

在这个专题下仅讨论一个问题，即，因为空值的存在使得新的类型功能更复杂了。例如，假定*V*是某一结构类型 *T*的一个变量。那么 *V*的某一部分就有可能为空，在这种情况下， V=V返回的就是unknown，而且条件表达式 V IS NULL 将返回false。实际上，如果（(V=V）IS NOT TRUE）IS TRUE为true，那么*V*可能为空或者是一个空的部分。

决策支持

SQL3支持在第21章中所说的 GROUP BY 子句中的 GROUPING SETS、ROLLUP和CUBE选项。