附　　录

附录A对SQL/92的语法和语义做了更加详尽的解释，以备参考之用。附录 B概要说明了SQL3 的主要特征，尤其是“实体”和“实体/关系”特征。附录C列出了书中用到的较为重要的缩写词和只取首字母的简略词，并解释了这些词的意思。

附录A　SQL表达式

A.1　引言

SQL语言的核心是SQL表达式，更确切地说，是SQL表、条件和标量表达式。该附录将按照SQL/92的标准详细说明这些表达式的语法和语义。然而，需要说明一点：因为标准术语不太恰当，所以在依据造句法进行的分类中和在 SQL语言的构造中所使用的名称跟 [4.22]的标准中使用的名称是不同的；实际上，本书中所使用的表表达式、条件表达式和标量表达式就不是标准术语。

A.2　表表达式

下面是一个<table expression>（表表达式）的BNF文法。除了一些处理空值（参见第 18章的第18.7节）的选项不能用该文法处理之外，其它的选项都可由该文法处理。注意：本章将继续使用第4章的4.6节中介绍的commalist（逗号列表）。

```
<table expression>
   ::=   <join table expression>
       | <nonjoin table expression>

<join table expression>
   ::=   <table reference> [ NATURAL ] JOIN <table reference>
                     [ ON <conditional expression>
                     | USING ( <column name commalist> ) ]
       | <table reference> CROSS JOIN <table reference>
       | ( <join table expression> )

<table reference>
   ::=   <table name> [ [ AS ] <range variable name>
                     [ ( <column name commalist> ) ] ]
       | ( <table expression> ) [ AS ] <range variable name>
                              [ ( <column name commalist> ) ]
       | <join table expression>

<nonjoin table expression>
   ::=   <nonjoin table term>
       | <table expression> UNION [ ALL ]
         [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
                     <table term>
       | <table expression> EXCEPT [ ALL ]
         [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
                     <table term>

<nonjoin table term>
   ::=   <nonjoin table primary>
       | <table term> INTERSECT [ ALL ]
         [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
```

```
                    <table primary>

<table term>
   ::=   <nonjoin table term>
       | <join table expression>

<table primary>
   ::=   <nonjoin table primary>
       | <join table expression>

<nonjoin table primary>
   ::=   TABLE <table name>
       | <table constructor>
       | <select expression>
       | ( <nonjoin table expression> )

<table constructor>
   ::=   VALUES <row constructor commalist>

<row constructor>
   ::=   <scalar expression>
       | ( <scalar expression commalist> )
       | ( <table expression> )

<select expression>
   ::=   SELECT [ ALL | DISTINCT ] <select item commalist>
            FROM <table reference commalist>
               [ WHERE <conditional expression> ]
                  [ GROUP BY <column name commalist> ]
                     [ HAVING <conditional expression> ]

<select item>
   ::=   <scalar expression> [ [ AS ] <column name> ]
       | [ <range variable name>. ] *
```

下面对<select expression>进行详细阐述，因为在实际使用中，该表达式是最重要的。一个<select expression>可以不很严格地看做是一个没有 JOIN、UNION、EXCEPT和INTERSECT操作的<table expression>。之所以说不很严格，是因为这些运算符可以出现在某些嵌套在<select expression>的表达式中。对JOIN、UNION、EXCEPT和INTERSECT的介绍可参见第7章的7.7节。

正如前面的文法中所写的，一个<select expression>按顺序包括：SELECT子句，FROM子句、可选的WHERE子句、GROUP BY子句和HAVING子句。下面依次来介绍这些子句。

1. SELECT子句

下面是SELECT子句的形式：

```
SELECT [ALL|DISTINCT]<select item commalist>
```

说明：

1) <select item commalist>不能为空（<select item>的详细介绍可参见下面）。

2) 如果没有指定是ALL还是DISTINCT，默认是ALL。

3) 此时假定已经执行完FROM、WHERE、GROUP BY 和HAVING子句。无论这些子句是给定的还是忽略的，执行完这些子句后，概念上来说得到一个表，这个表可能是一个“组”表（可参见后面的介绍），称这个表为*T1*，该表将会在后面用到。注，这个概念结果实际上是没有命名的。

4) 假定*T2*是通过在*T1*上执行指定的<select item>而从*T1*中得到的。

5) 假定*T3*是通过指定DISTINCT从*T2*或者是跟*T2*同样的表中消除冗余行后得到的。

6) *T3*是最后的结果。

现在对<select item>做一下解释。对<select item>，需要考虑两种情况，但是因为第二种

情况是第一种情况的<select item>的逗号列表的简写，因此，第一种情况更加基本。

第一种情况：<select item>的形式是：

```
<scalar expression> [[AS] <column name>]
```

<scalar expression>可以包含*T1*表的一个或多个列（见说明的第3点），当然这种包含不是必需的。对*T1*中的每一行，执行<scalar expression>将产生一标量结果。在*T1*表每一行上执行SELECT子句中的所有<select item>，将得到一个结果的逗号列表。这个逗号列表就是*T2*表（见上面说明的第4点）的一行。如果<select item>包括一个AS子句，那么，该子句中的没有限定的<column name>将赋值到*T2*对应的列上，当然，可以忽略可选的AS关键字，这不会影响执行结果。如果<select item>子句没有包含AS子句，就有两种情况需要考虑：(a) 如果它只包含了简单的可能有限定的<column name>，<column name >就作为*T2*中对应的列名；(b) 否则，*T2*中对应的列就没有有效的列名，实际执行中，将对*T2*中的列赋值一个执行依赖的列名[4.19，4.22]。

下面再做几点说明：

- 特别地，因为在AS子句中引入的列名是*T2*的列名，而不是*T1*的列名，所以在构造*T1*的WHERE、GROUP BY和HAVING子句时，就不能直接包含该列名。然而，它在任何情况下（特别地，在DECLARE CURSOR中）可以在一个相关连的ORDER BY子句中参照，也可以在一个包含<select expression>的<table expression>中参照使用。
- 如果<select item>包括一个聚集操作符，而且<select expression>不包含GROUP BY子句，那么：若*T1*的列没有作为聚集操作符的参数或者部分参数使用，则SELECT子句中的<select item>就不能参照*T1*中的任何列。

第二种情况：<select item>是如下的形式：

```
[<range variable name> . ] *
```

如果忽略了限定词，例如，<select item>只是没有限定的星号，那么这个<select item>就是SELECT子句中的唯一的<select item>了。这是一种按照从左往右的顺序列出*T1*中的所有的列名的简写方式。如果包含了限定词，如，<select item>包括了一个由范围变量名所限定的星号：*R*.*，那么<select item>就会按照从左往右的顺序列出跟范围变量*R*相联系的表的所有列的<column name>的逗号列表。注意，7.7节中曾经介绍过，一个表的名字经常会作为一个隐含的范围变量。这样，<select item>就经常是“*T*.*”的形式，而不是“*R*.*”的形式。

2. FROM子句

FROM子句的形式如下：

```
FROM <table reference commalist>
```

此处，<table reference commalist>不能为空。指定<table reference>分别对应表*A*，*B*，...，*C*。那么，FROM子句的执行结果就是一个等同于*A*，*B*，...，*C*做笛卡尔积后得到的表。注意：前面已经介绍过一个单表*T*的笛卡尔积仍然等于*T*（参见第6章的练习6.12的答案）；因此，FROM子句只包含一个<table reference>也是正确的。

3. WHERE子句

WHERE子句的形式如下：

```
WHERE <conditional expression>
```

假定增加了前面的FROM子句后执行得到的结果是*T*表。而WHERE子句的结果则是将*T*表中不

符合<conditional expression>的行去掉后得到的。如果忽略掉WHERE子句，结果就仍是*T*。

4. GROUP BY子句

GROUP BY子句的格式如下：

```
GROUP BY <column name commalist>
```

<column name commalist>不能为空。假定增加了前面FROM子句和WHERE子句之后，执行结果是*T*表。每一个在GROUP BY子句中的<column name>必须是*T*表的一个可选的有限定的列名。GROUP BY子句的结果是一个组表，如组的集合（每个组中又包括若干行）。通过概念上重新排列*T*表中的行使组数减小到最少，就得到了组表中的行，组表中的每一个组中的所有记录行在GROUP BY子句所指定的列组合上具有相同的值。因此要注意，这个结果不是一个“真正的表”，即不是一个记录行的表，而是由组组成的一个表。然而，如果没有相应的SELECT子句，GROUP BY子句也不会出现，SELECT语句就是从组表中得到真正的表（记录行的表）。因此，这种对关系框架的临时偏离对最后的结果没有很大的影响。

如果<select expression>包括一个GROUP BY子句，那么相应的SELECT子句的形式就会有限制。特别地，SELECT子句（包括星号简写的形式） 中的每一个<select item>在每个组中只有一个值。这样，如果 <select item>的聚集操作符的参数或者部分参数没有引用 T表的列，<select item>就不能参照GROUP BY子句中没有提到的T表的任何列。聚集操作符的功能是将一个组中标量值的集合减小到只有一个标量值。

5. HAVING子句

HAVING子句的格式如下：

```
HAVING <condition expression>
```

假定在增加了前面的FROM子句、WHERE子句和GROUP BY子句之后执行得到的结果是G表。如果没有GROUP BY子句，那么G就是在只执行FROM和WHERE子句的情况下得到的表，是一个只有一个组的表[⊖] 。换句话说，在这种情况下，有一个隐含的、概念上的 GROUP BY子句，该子句没有指定进行分组所需要参照的列。 HAVING 子句的结果就是将 G表中不满足条件<conditional expression>的组去掉后所得到的表。

下面再做几点说明：

- 如果省略了HAVING子句，但是仍然包括有 GROUP BY子句，执行的结果就是 G。如果HAVING子句和GROUP BY子句都省略了，结果就是只执行 FROM和WHERE子句后得到的*T*表，是没有分组的。
- HAVING子句中的任何<scalar expression>在每组中必须只能有一个值。这跟 SELECT子句中有GROUP BY子句时<scalar expression>的情况类似。

6. 综合示例

在讲解 <select expression> 的最后部分，给出一个复杂度适当的例子，通过该例可以更进一步阐明上面所讲的一些（但不是全部）内容。进行下面的查询：对于那些红色和蓝色的总供给量超过350的零件，列出其零件号、重量（以克计）、颜色和该零件的最大供给量。下面是一种可行的查询：

⊖ 虽然逻辑上应该说是每个组只有单个值，但是这就是标准的方式。如果 FROM和WHERE子句执行后产生一个空表，就不会有任何的组。

```
SELECT P.P#,
       'Weight in grams =' AS TEXT1,
       P.WEIGHT * 454 AS GMWT,
       P.COLOR,
       'Max quantity =' AS TEXT2,
       MAX ( SP.QTY ) AS MXQTY
FROM   P, SP
WHERE  P.P# = SP.P#
AND   ( P.COLOR = 'Red' OR P.COLOR = 'Blue')
AND    SP.QTY > 200
GROUP  BY P.P#, P.WEIGHT, P.COLOR
HAVING SUM ( SP.QTY ) > 350 ;
```

说明：需要注意<selection expression>**子句概念上的执行是按照其书写顺序一一进行的。当然，**SELECT**语句除外，因为它要在最后执行。因此，可以想像，在该例中，结果集是这样构造的：**

1) FROM：**执行**FROM**子句以生成一个新表，该表是表** P **和表**SP**的笛卡尔积。**

2) WHERE：**将第**1**步中不符合**WHERE**子句条件的记录行去掉。该例中，是去掉不满足下面条件的记录行：**

```
       P.P# = SP.P#
AND   ( P.COLOR = 'Red' OR P.COLOR = 'Blue')
AND    SP.QTY > 200
```

3) GROUP BY：**将第**2**步的结果按照**GROUP BY**子句中的列名分组。该例中，需要根据其分组的列名是：** P.P#、P.WEIGHT**和**P.COLOR**。注意，理论上来说只将** P.P#**作为分组列就已经足够了，因为每一个** P.COLOR**和**P.WEIGHT**都对应一个单值的零件号（例如，它们都是依赖于零件号的）。然而，**SQL**并不注意这个事实，当** P.WEIGHT**和**P.COLOR**省略时，系统会给出一个出错信息，这是因为它们在** SELECT**子句中有声明。对这一点的讨论，可参见**[10.6]**。**

4) HAVING：**那些不满足条件表达式。**

```
SUM(SP.QTY)>350
```

的组将要从步骤3**的结果中删除掉。**

5) SELECT：**第**4**步结果中的每个组产生一个只有一个结果的记录行。首先，从组中取得零件号、重量、最大需求量。第二，将重量转化为克来表示。第三，将字符串“** Weight in grams= **”和“** Max quatity= **”插入到记录行中适当的位置。注意，之所以说“适当的位置”，是因为在**SQL**中，表中的列有从左到右的顺序，如果这两个字符串没有出现在“恰当的位置”，那么它们的意义也就不大了。**

最后的结果是如下的形式：

P#	TEXT1	GMWT	COLOR	TEXT2	MXQTY
P1	Weight in grams =	5448	Red	Max quantity =	300
P5	Weight in grams =	5448	Blue	Max quantity =	400
P3	Weight in grams =	7718	Blue	Max quantity =	400

刚才描述的算法仅对 <select expression>**的执行进行了概念上的解释。从保证生成正确结果的方面来看，算法是正确的。然而，实际执行时却有可能不是很有效。例如，要使系统在第**1**步中非常正确地生成笛卡尔积，就是非常不可能的事情。这和第** 17**章的讨论都说明了为什么在关系系统中需要优化器。确实，** SQL**系统中的优化器的作用就在于：可以用比概念算法更少的时间来得到相同的结果。**

A.3 条件表达式

<conditional expression>跟<table expression>一样，在SQL语言中也出现得非常之多；当然，它们是专门用在WHERE子句中的，以在子查询处理过程中去掉某些不恰当的行[⊖]。本节将介绍这些表达式的一些重要特征。然而，请注意，这里并不是要对其进行详尽地解释。跟第18章的处理一样，在这里也忽略对空值的处理，如果考虑空值，<conditional expression>将要做很多的扩展处理。有的<conditional expression>格式对空值提供支持，当然这些是没有包括在这个附录中。这些问题都已经在第18章中进行了描述。

跟上节类似，首先给出一个BNF文法。接下来详细介绍以下几个问题：<like condition>、<match condition>、<all or any condition>和<unique condition>，对于其他的已在本书其余部分做过介绍的，或者内容非常浅显的，在此不再对其进行详细的解释。

```
<conditional expression>
   ::=   <conditional term>
       | <conditional expression> OR <conditional term>

<conditional term>
   ::=   <conditional factor>
       | <conditional term> AND <conditional factor>

<conditional factor>
   ::=   [ NOT ] <conditional primary>

<conditional primary>
   ::=   <simple condition> | ( <conditional expression> )

<simple condition>
   ::=   <comparison condition>
       | <in condition>
       | <like condition>
       | <match condition>
       | <all or any condition>
       | <exists condition>
       | <unique condition>

<comparison condition>
   ::=   <row constructor>
                 <comparison operator> <row constructor>

<comparison operator>
   ::=   = | < | <= | > | >= | <>

<in condition>
   ::=   <row constructor> [ NOT ] IN ( <table expression> )
       | <scalar expression> [ NOT ] IN
                          ( <scalar expression commalist> )

<like condition>
   ::=   <character string expression> [ NOT ] LIKE <pattern>
                                       [ ESCAPE <escape> ]

<match condition>
   ::=   <row constructor> MATCH UNIQUE ( <table expression> )

<all or any condition>
   ::=   <row constructor>
             <comparison operator> ALL ( <table expression> )
       | <row constructor>
             <comparison operator> ANY ( <table expression> )

<exists condition>
   ::=   EXISTS ( <table expression> )

<unique condition>
   ::=   UNIQUE ( <table expression> )
```

⊖ 在第18章中已经说明：条件表达式就是在书中所说的布尔表达式，在此再次提醒读者。

1. Like条件

Like条件一般用于简单的字符串的模式匹配。例如：检查一个字符串，看它是否与某一预定义的模式相匹配。其语法是：

```
<character string expressuin> [NOT] LIKE <pattern>
                               [ESCAPE] <escape>]
```

这里，<pattern>是任意的字符串表达式，<escape>是一个只有单个字符的字符串表达式。这是一个例子：

```
SELECT P.P# P.PNAME
  FROM P
WHERE P.NAME LIKE 'C%'
```

(列出那些零件名以C开头的零件的零件号和零件名)。结果为：

P#	PNAME
P5	Cam
P6	Cog

只要没有指定ESCAPE子句，<pattern>中的字符串按照如下的方式解释：

- 下划线“_”表示任意单个字符。
- 百分号“%”代表任何有序的*n*个字符（*n*可以为0）。
- 所有其他的字符代表其本身。

因此，在该例中，查询将返回表P中PNAME以C开头，之后可以有0到多个字符的记录行。

下面是一些例子：

```
ADDRESS LIKE '%Berkeley%'   —— 如果ADDRESS中包含“Berkeley”不管其出现的位置，返回真值

S# LIKE 'S__'               —— 如果S#正好为三个字符，且第一个为S，则返回真值

PNAME LIKE '%c___'          —— 如果PNAME为4个或多于4个字符，且倒数第4个为C，则返回真值

MYTEXT LIKE '=_%'           —— 如果MYTEXT包含一个下划线的字符(见下面)，则返回真值
       ESCAPE '='
```

在最后的例子中，字符“=”指定作为转义字符，转义字符的作用就是使特殊字符“_”和“%”不再具有特殊的意义，如果想使用这些特殊字符，就使用转义字符“=”。

最后，<like condition>

```
x NOT LIKEy[ESCAPEz]
```

在语义上跟下面的语句是相同的：

```
NOT (x LIKE y[ESCAPEz])
```

2. MATCH条件

<match condition>的格式是：

```
<row constructor> MATCH UNIQUE (<table expression>)
```

假定*r1*是执行<row constructor>后得到的一行，*T*是执行<table expression>后得到的一个表。

如果*T*有且只有一行，假定为*r2*，那么该<match condition>执行结果为真，下面的比较将返回真。

```
r1 = r2
```

下面是一个例子：

```
SELECT SP*
FROM   SP
WHERE  NOT (SP.S# MATCH UNIQUE (SELECT S.S#  FROM S));
```

（如果SP表的供应商不只对应供应商表中一个供应商，则列出其发货量 ）。当然，如果数据库正确，该查询就不会有什么结果，所以这样的一个查询对于检查数据库的完整性来说是有用的。需要注意，<in condition>语句也有可能达到这样的目的。

顺便说一下，UNIQUE可以从MATCH UNIQUE中省略，这样MATCH就跟IN非常类似，至少在没有空值的情况下是这样的。

3. All或Any条件

<all or any condition>的一般形式是：

```
<row constructor> <comparison operator> <qualifier>
                                    (<table expression>)
```

其中，<comparison operator>可以是任何的通常的运算符(=、<>，等)，<qualifier>是All或Any，一般来说，当且仅当对<table expression>给出的表中的所有行来说，没有ALL（即为ANY）的比较都为true时，<all or any condition>才为true。如果表为空，ALL条件将为true，但是ANY条件将为false。下面给出一个例子，“取得那些重量超过每一个蓝色零件的零件名”：

```
SELECT DISTINCT PX.PNAME
FROM P AS PX
WHERE PX.WEIGHT>ALL (SELECT PY.WEIGHT
                     FROM P AS PY
                     WHERE PY.COLOR='Blue');
```

使用一般的示例数据，结果如下：

PNAME
Cog

说明：嵌套的<table expression>返回蓝色零件的重量的集合。外层的SELECT返回那些重量超过上面的集合的零件名。当然，一般来说，结果的数目是不确定的（也有可能为 0）。

注意：需要注意一个单词的使用，母语为英语的读者更应该注意。 <all or any condition>跟一般的使用有很大的不同。自然的英语会使用“ any ”来表示“ every ”的查询，这样，大家在使用时就很容易使用“ >ANY ”而不是“ >ALL ”。这样的批评对ANY和ALL操作符来说都是适用的。

4. UNIQUE 条件

<unique condition>通常用来测试表中的每一行是不是唯一，例如，是不是有重复。其语法是：

```
UNIQUE (<table expression>)
```

如果<table expression>中表的各行都是不同的，那么该条件就赋值为 true，否则为false。

A.4 标量表达式

SQL的<scalar expression>本质上来说是很直接的。下面列出可在这样的表达式中使用的一些重要的操作符，并且对那些意思不很明显的操作符做出简要的解释。下面按照字母的排列顺序列出这些操作符：

```
arithmetic operators (+, -, *, /)     OCTET_LENGTH
BIT_LENGTH                            POSITION
CASE                                  SESSION_USER
CAST                                  SUBSTRING
CHARACTER_LENGTH                      SYSTEM_USER
concatenation (||)                    TRIM
CURRENT_USER                          UPPER
LOWER                                 USER
```

注意：因为聚集操作符返回一个标量结果，所以它也可以出现在 <scalar expression>中。如果括在圆括号中的<table expression>执行结果只有一行一列，那么它也可以作为一个标量值。

现在来详细介绍CASE和CAST操作符。

1. CASE操作

CASE操作根据特定的条件，返回特定的值的集合。例如：

```
CASE
   WHEN S.STATUS <  5 THEN 'Last resort'
   WHEN S.STATUS < 10 THEN 'Dubious'
   WHEN S.STATUS < 15 THEN 'Not too good'
   WHEN S.STATUS < 20 THEN 'Mediocre'
   WHEN S.STATUS < 25 THEN 'Acceptable'
   ELSE                    'Fine'
END
```

2. CAST操作

CAST将一标量值转为特定的标量数据类型。例如：

```
CAST ( P.WEIGHT AS FLOAT)
```

并不是所有的数据都是可转换的；例如，数字和位字符串之间的转换就是不允许的。可以参见[4.22]查看哪些具体的数据类型之间可以相互转化。