第8章 完 整 性

8.1 引言

数据库中完整性(integrity)一词指数据的正确性和相容性。正像第3章所说的，一个给定的数据库可能受到多个完整性约束的限制，可能是简单的也可能是复杂的。比如，在供应商-零件例子中，供应商号码可能是 *Snnnn*（*nnnn* 为1到4位整数），并且这一编号是唯一的；状态值可能在 1~100的范围内；伦敦的供应商状态值必须是 20；发货量必须是 50的倍数；颜色为“红”的零件必须被存放在伦敦；等等。通常，DBMS都被告知存在这样的约束，当然，也能在一定程度上实现这样的约束（一般是通过拒绝违反这些约束的更新请求来实现）。例如（再次引用 Tutorial D）：

```
CONSTRAINT SC3
    IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

（状态值必须在 1~100范围内）。这个约束被命名为 SC3（供应商约束 3，suppliers constraint 3）；这个约束将以此名在系统目录（system catalog）中注册，当系统对企图违反这个约束的操作作出响应时，这个名字将在诊断提示信息中出现。此约束本身被指定为一个布尔表达式，并且，其值不可为假。

注意：为明确起见，我们采用 Tutorial D 的代数形式；所以，这个布尔表达式常常采取（虽然并非一成不变）如下的形式：IS_EMPTY(...)，意思是数据库中没有违反这个约束的元组（参见第 6章，6.9节）。这个例子的演算描述为[⊖]：

```
CONSTRAINT SC3
    FORALL SX ( SX.STATUS ≥ 1 AND SX.STATUS ≤ 100 ) ;
```

(SX 是定义在供应商上的一个范围变量)。

另外我们注意到以演算约束形式存在的布尔表达式必须是一个封闭的合式公式 (参见第7章，7.2节)，并且常（但并非一成不变）采用 FORALL *x*(...)的形式。这样，这个例子约束所有的供应商状态值必须在指定的范围内。实际上，对于一个数据库系统，只要检查新插入或更新的“供应商”记录就可以了，而不用检查所有的供应商记录。

当声明一个新的约束时，系统首先要确保当前系统满足这个约束。如果当前系统不满足这个约束，这个新约束就被拒绝；反之，则被接受（也就是被存储在目录中），从此新的约束开始生效。此后，DBMS就要监视每一个插入的“供应商”记录和会引起已存在供应商记录状态值变动的操作。

当然，我们也要有能够解除已存在约束的方法。

```
DROP CONSTRAINT <constraint name> ;
```

例如：

⊖ 实际上，用演算而不用代数形式的公式表示约束看起来要简单一些（特别是复杂的约束）。为了与本书其他章节一致，这里只讨论代数形式，读者可以试着将下面的代数形式转化为演算形式。

```
DROP CONSTRAINT SC3 ;
```

注意：正像下面的讨论所指出的，我们将尤其关注声明完整性支持。遗憾的是，即使现在，也几乎没有什么产品能在这方面做得很好。但在这一点上是不断进步的，虽然目前一些数据库产品（尤其是非关系数据库）还特别强调一种与此相反的方法——也就是对约束的“过程支持”（procedural support），通过存储过程和触发过程来实现[⊖]。但是，建议一个DBMS提供对约束的声明支持（declarative support），因为多达百分之九十的数据库定义中包含约束。一个提供对约束的声明支持的数据库将在相当程度上减轻程序员的负担并使他们的生产率更高。对约束的声明支持是相当重要的。

在进一步讨论这个问题之前，可以说：在关系模型中，完整性是在这些年里变化最大的（可能应该说“演化”而不是“变化”）。正像第3章中提到的，最初，主码和外码（简称为“码”）是人们关注的重点，逐渐地，完整性约束的重要性——普遍存在，且具有毫无争议的重要性——得到人们的普遍理解和广泛重视。同时，关于码的一些讨厌的细节问题不断出现。本章的结构体现了这一重点的转变。我们先介绍完整性约束（仅在一定程度上讨论），然后讨论“码”，因为在应用中，“码”这一概念还有相当的重要性。

“约束”的分类

根据参考文献[3.3]，大体上完整性约束可分为四大类：类型（域）约束、属性约束、关系变量约束和数据库约束。

- 类型约束指明给定类型的合法取值。注意：在本章中，我们用“类型”表示一个标量类型。关系类型也受类型约束的制约，但是，这种关系类型的约束只是表示：关系类型是通过标量类型来定义，而标量类型受到约束，这一逻辑结果就是关系类型也受类型约束制约。
- 属性约束说明属性的合法取值。
- 关系变量约束说明关系变量的合法取值。
- 数据库约束说明数据库的合法取值。

这四种类型将在8.2~8.4节中分别加以详细讨论。

8.2 类型约束

实质上，类型约束只是（或者逻辑上等同于）给定类型的合法取值的列举。下面是一个例子，是类型“WEIGHT”的类型约束：

```
TYPE WEIGHT POSSREP ( RATIONAL )
     CONSTRAINT THE_WEIGHT ( WEIGHT ) > 0.0 ;
```

这里采取一个显而易见的约定，通过这种约定，类型约束可以利用适当的类型名来指定此类型的任意值；这样，上述例子约束“重量”为：“重量”可表示为大于0的合理数值，任何将“重量”赋值为不满足此约束的表达式将失败。注意：关于“POSSREP”和“THE_”的含义，请参见第5章。

⊖ 存储过程和触发过程是可以被应用程序调用的预编译过程。例子中可能包括用户定义的操作符ABS、DIST、REFLECT，等等，这些已在5.2节中讨论过（“操作符定义”小节）。这些过程在逻辑上可以被看作是DBMS的功能扩展（在客户机/服务器系统中，通常是在服务器端执行）。在8.8节、本章最后的参考文献注释和第20章中我们还会对“过程”有进一步的讨论。

重申一下，类型约束只是构成这个类型的取值的描述。在 Tutorial D中，将这一约束和其数据类型定义绑定在一起，通过所用的类型名识别它们（这样，只有删除一个类型本身时，才能把这个约束删除）。

现在，应该已经明确：任何一个能够产生类型“ WEIGHT ”的值的表达式，都是通过某个对“ WEIGHT ”类型的选择子调用实现的。这样，这个表达式违反“ WEIGHT ”的类型约束的唯一原因就是选择子调用违反了这一约束。类型约束总是可以被认为是 —— 至少概念上是 —— 在选择子请求被执行过程中检查的。结果，我们可以说类型约束是被立即执行的。这样，所有关系变量不能为其任何一个属性得到一个与其类型不适应的取值（在支持类型约束的系统中）。

下面是类型约束的另一个例子：

```
TYPE POINT POSSREP CARTESIAN ( X RATIONAL, Y RATIONAL )
     CONSTRAINT ABS ( THE_X ( POINT ) ) ≤ 100.0 AND
                ABS ( THE_Y ( POINT ) ) ≤ 100.0 ;
```

理论上，这个类型检查是在 CARTESIAN选择子调用的执行过程中被实施的。注意这里使用了用户定义的操作符 ABS（参见第5章，5.2节）。

第三个例子：

```
TYPE ELLIPSE POSSREP ( A LENGTH, B LENGTH, CTR POINT )
     CONSTRAINT THE_A ( ELLIPSE ) ≥ THE_B ( ELLIPSE ) ;
```

这里，A、B、CTR分别代表椭圆的长半轴长度 *a*和短半轴长度 *b*及中点 *ctr*。假设标量 E 被定义为类型 ELLIPSE，其长半轴值为 5，短半轴值为 4。

考虑如下赋值：

```
THE_B ( E )  :=  LENGTH ( 6.0 ) ;
```

注意：出于简便考虑，这个例子是基于标量变量和标量赋值的；但我们可以方便地使之基于关系变量和关系赋值。

显然，上述赋值将失败。但这个赋值语句本身并没有错误，错误也是发生在选择子调用的内部（虽然在这个赋值中没有直接的请求），因为这个赋值语句实际上是下面语句的缩写[⊖]：

```
E  :=  ELLIPSE ( THE_A ( E ), LENGTH ( 6.0 ), THE_CTR ( E ) ) ;
```

这是因为右面的选择子调用语句失败。

8.3 属性约束

属性约束只是将某个属性指定为确定类型的声明。比如，供应商关系变量定义：

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

在这个关系变量中，属性 S#、SNAME、STATUS、CITY的取值分别被规定为类型 S#、NAME、INTEGER和CHAR。也就是说，属性约束是这些属性定义的一部分，通过属性的

⊖ 换句话说 —— 虽然在第5章中我们没有明确表示出来 —— THE_这个伪变量逻辑上是不必要的。也就是说，任何给 THE_伪变量的赋值，在逻辑上总是等同于（实际上是被定义为一个缩写）将一个选择子调用的结果赋值给一个常规变量。

名字我们可以将之识别出来。显然，只有当删除一个属性本身时才能删除属性约束（实际上这意味着删除包含它们的关系变量）。

注意：原则上，当一操作向数据库中引入某属性的新值时，若此值与其类型不相容，则此操作会被简单地拒绝。实际上，如果系统中已经实施了上一节中讲到的类型约束，这种情况就不会发生。

8.4 关系变量约束

关系变量约束是对单个关系变量的约束（仅用被讨论的关系变量来表示，虽然在某些方面，这可能相当复杂）。下面有一些例子：

```
CONSTRAINT SC5
    IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ;
```

（“伦敦的供应商状态值必须是 20”）。

```
CONSTRAINT PC4
    IS_EMPTY ( P WHERE COLOR = COLOR ( 'Red' ) )
                   AND CITY ≠ 'London' ) ;
```

（“红色零件必须被存储在伦敦”）。

```
CONSTRAINT SCK
    COUNT ( S ) = COUNT ( S { S# } ) ;
```

（“供应商编号是唯一的”，或者更正式一点说，“{S#}是供应商的候选码 —— 参见 8.8 节）。

```
CONSTRAINT PC7
    IF NOT ( IS_EMPTY ( P ) ) THEN
       COUNT ( P WHERE COLOR = COLOR ( 'Red' ) ) > 0
    END IF ;
```

（“如果有零件，则其中至少有一种是红色的”）。注意，这个例子与我们见过的都不同，因为删除操作后的结果可能违反此约束。

关系变量约束通常是在执行某项操作后被立即检查的（实际上，是作为可能带来违反此约束后果的指令的一部分来执行）。这样，任何向特定关系变量赋值的语句，如果违反某个关系变量约束，将会被拒绝。

8.5 数据库约束

数据库约束是使两个或更多的关系变量相互联系的约束。下面有一些例子：

```
CONSTRAINT DBC1
    IS_EMPTY ( ( S JOIN SP )
               WHERE STATUS < 20 AND QTY > QTY ( 500 ) ) ;
```

（“如果供应商的状态值小于 20，那么他提供的零件数不能大于 500”）。练习：为实施约束DBC1，DBMS应对更新操作进行怎样的检测？

```
CONSTRAINT DBC2 SP { S# } ⊆ S { S# } ;
```

（发货关系变量中的每一个供应商编号，都要在供应商关系变量中存在；回忆第 6章，我们用⊆来表示“子集”）。因为关系变量 S中的属性 S#构成了参照“供应商”的候选码，这个约束就成为从“发货”到“供应商”的必要的参照约束（也就是说，在关系变量 SP中的{S#}，是“发货量”相对于“供应商”的外码）。详细内容参见 8.8节。

```
CONSTRAINT DBC3 SP { P# } = P { P# } ;
```

(“每一种零件，至少要有一次发货记录”)。注意：显然，每一次发货至少要对应一种零件，这是因为在关系变量中的 {P#}是参照“零件”的候选码，并且存在一个从“发货”到“零件”的参照约束。这里就不把后一个约束写出来了。详细内容参见 8.8节。

后两个例子用来说明：数据库约束检查是不能被立即执行的，而必须被延迟到事务结束，也就是提交（COMMIT）时（关于COMMIT，参照第3章）。相反，假设“检查”是可以被立即执行的，并且假设目前在“供应商”和“零件”中没有记录。这时，插入一条零件记录，失败，因为违反约束 DBC3；反之，插入一条出货记录，失败，因为违反约束 DBC2 ⊖。

如果提交时违反数据库约束，事务被回滚。

8.6 黄金法则

注意：本节内容非常重要，但不幸的是，在实际中它并没有得到广泛支持，甚至没有被很好地理解，尽管它是很直观的。

在第3章的3.4节，我们解释了对关系如何指定一个谓词，以及元组如何表示一个由此谓词导出的正确的命题。在第 5章的5.3节，提到了“封闭世界假设”，意思是，如果某个元组没在某个特定的关系中出现，就可以认为相应的命题是错误的。

到现在为止，我们还没有强调过这样一点，但我们应该明白“关系变量也有一个谓词”，也就是说，每一个关系都存在谓词，它们指明了这些关系变量的合法取值。再次考虑供应商关系变量 S，它的谓词是：

> 带有确定的供应商号码的供应商有特定的名字，状态值，位于确定的城市中；而且，不能同时有两个供应商的号码相同。

这段描述既不详细也不完整，但目前够用了。

一个给定关系变量的谓词的作用是充当此关系变量更新的准则，它指出了对于此关系变量的更新、插入和删除是否可以被执行，这一点应该是很清楚的。比如，插入一个已经存在的供应商编号的请求会被拒绝。

理想上，DBMS应该知道并且理解谓词的意思，因此它能正确处理任何更新操作。当然，这一目标还没有实现。比如， DBMS无法得知某个“供应商”处于“某个城市”中是什么意思，它也无法由“供应商”的谓词推理得知元组

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

满足它，而元组

```
{ S#     :  S# ( 'S6' )      ,
  SNAME  :  NAME ( 'Smith' ) ,
  STATUS :  50               ,
  CITY   : 'Rome'            }
```

不满足。实际上，如果用户要用后一个元组插入，系统所要作的就是确定这个元组没有违反任何现存的完整性约束。如果是这样的话，系统将接受此插入，并且以后把它当作

⊖ 实际上，参考文献 [3.3]中提出一种赋值的复杂形式，它允许在同一个操作中对“零件”和“供应商”进行插入。如果 DBMS支持这种赋值，数据库约束就可以被立即检查。

正确的命题。

重申一下，系统不能（也不可能）百分之百地知道并理解供应商谓词，但它确实能知道它的大致意思。准确地说，它知道用于“供应商”的完整性约束。而且，我们可以定义在关系变量供应商上（更普遍地说，任何关系变量）的新的谓词，使它和已有的关系变量约束成“AND”的逻辑关系。比如，关系变量的谓词是这样的：

```
( IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ) AND
( IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ) AND
( COUNT ( S ) = COUNT ( S { S# } ) )
```

（当然，系统知道属性 S#、SNAME、STATUS、CITY的类型分别是 S#、NAME、INTEGER 和CHAR）

注意，如果这样，对于一个给定的关系变量就有两个有效的谓词：非正式谓词或外部谓词（external predicate），它是用户理解而系统不能理解的；正式谓词或内部谓词，它是系统和用户都能理解的。当然，内部约束才是我们所说的“关系变量谓词”，而且对于它，更新操作无论何时发生，系统都要进行检查。从现在开始，用“谓词”（当和某个关系变量相连时）来专门表示内部谓词，而不再显式说明。

现在，可以将黄金法则定义如下（至少法则第一个版本可以这样定义）：

> 如果一个更新操作将使一个关系变量处于违反自身某个谓词的状态，这样的更新是被禁止的。

应该强调一下，这里的关系变量不一定是一个基本关系变量——黄金法则适用于所有的关系变量，基本的或是导出的。下一章再讨论这个问题。

需要指出，正像关系变量有一个捆绑在一起的谓词一样，每一个数据库也有一个捆绑在一起的数据库谓词，它与此数据库上的其它谓词形成逻辑“与”的关系。现在，可以将黄金法则扩展如下[⊖]：

> 如果一个更新操作将使一个关系变量处于违反自身某个约束的状态，这样的更新是被禁止的；相应地，如果一个更新事务将使一个数据库处于违反自身某个约束的状态，这样的更新事务是被禁止的。

8.7 静态约束和动态约束

到目前为止，讨论的约束都是静态约束：静态约束是关于数据库正确状态的约束。但有时要考虑变化过程的约束——也就是说，关于数据库从一种正确状态到另一种正确状态转移的约束。例如，一个关于人口的数据库，有一些对于“婚姻状况”的动态约束。例如，下面的状态转移是有效的：

- 未婚到已婚
- 已婚到丧偶
- 已婚到离异
- 丧偶到已婚

（等等），而下面的是非法的：

- 未婚到丧偶

⊖ 如果支持同时用多条语句赋值，就可以用黄金法则的原来的那种简单形式。

- 未婚到离异
- 离异到丧偶

等等。回到供应商-零件数据库中，有另外一个例子（不允许供应商的状态值减小）：

```
CONSTRAINT TRC1 IS_EMPTY
   ( ( ( S' { S#, STATUS } RENAME STATUS AS STATUS' )
       JOIN S { S#, STATUS } )
     WHERE STATUS' > STATUS ) ;
```

解释：引入如下的约定：在更新操作进行之前，关系变量名（这里是 S'）已被理解为指向对应的关系变量。这样，本例中的约束就可以这样理解： (a)将更新前后的关系变量在供应商号上进行连接； (b)找到旧元组值大于新元组值的元组； (c)结果集必须为空（因为连接建立在供应商编号上，任何旧值大于新值的元组表示状态值减小）。

注意：约束 TRC1是关系变量上的动态约束（它只适用于单个的关系变量，这里是“供应商”），而且检查是立即执行的。下面有一个数据库动态约束的例子 (所有供应商对某一种零件供应的总数量不能减少)：

```
CONSTRAINT TRC2 IS_EMPTY
    ( ( ( SUMMARIZE SP' PER S'{ S# } ADD SUM ( QTY ) AS SQ' )
        JOIN
        ( SUMMARIZE SP  PER S { S# } ADD SUM ( QTY ) AS SQ  ) )
      WHERE SQ' > SQ ) ;
```

约束TRC2是一个数据库动态约束（它涉及两个关系变量，供应商和发货）；因此，检查被延迟到提交时，关系变量名 S'和SP'在开始事务时就被理解为关系变量 S和SP。

在静态约束与动态约束的概念中，对类型约束和属性约束没有意义。

8.8 码

关系模型一直强调码的重要性，虽然它只是一般情况下的一个特殊情况（重要的一种）。在本节中，我们将关注一下码这个概念。

注意：虽然码的基本思想很简单，但不幸的是，这里多了一个复杂的因素：空值（NULL）。对于外码允许空值的可能性在一定程度上引起混乱。空值本身就是一个很广的话题，但在这里要详细地讨论它是不合适的。出于教学的目的，这里暂且忽略空值；第18章专门讲述空值时，再讨论它在此处的影响（实际上我们有一种偏见，觉得空值是一个错误，不应该被引入，但在一本讨论数据库原理的书中，把它完全忽略掉也是错误的）。

1. 候选码

设*R*为一个关系变量，由定义， *R*的属性集具有唯一性(uniqueness)的特性，也就是，在任一时刻，没有两个元组是重复的。实际上， *R*的属性集合的某个真子集也具有唯一性；比如在供应商关系变量 S中，S#就具有此特性，这样就得到候选码的定义：

设*K*是*R*的一个属性集，当且仅当 *K*满足下列特性时，它是 *R*的候选码 ⊖：

a) 唯一性：*R*上没有两个不同的元组，在 *K*上有相同的值。

b) 不可约性：*K*没有一个真子集也具有唯一性。

注意，每一个关系变量至少有一个候选码。唯一性是无需解释的；关于不可约性这一

⊖ 观察到这个定义只用于关系变量（关系变量），可以给关系值作出一个相似的定义 [3.3]，但关系变量要更为重要。

点，如果指定了不具有不可约性的候选码，系统将无法得知事件的真实状态，于是不能正确实施相应的完整性约束。比如，定义 {S#，CITY}而不是{S#}为候选码，系统将无法实施供应商编号在全数据库内是唯一的这一“全程”约束，它只能实施较弱的“在城市内唯一”的“局部”约束。因此要求候选码不包括任何与唯一识别的目的无关的其它属性㊀。

顺便提一下，前面讲的不可约性在字面上的意思是最小性（ minimality），（包括本书的前几版）。但最小性也不是一个准确的词语，因为如果说候选码 *K1*是最小化的，并不意味着无法找到另一个候选码 *K2*，而*K2*包含更少的元素；有可能 *K1*包含四个元素而 *K2*仅有两个。所以还用“不可约性”这个词。

在Tutorial D中，用如下语法

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

指定这个关系变量的候选码。下面有一些例子：

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

注意：前几章中，在定义中带有“ PRIMARY KEY ”的语句，而不是用“ KEY ”。请参考“主码和替换码”一小节，有更多的讨论。

```
VAR SP BASE RELATION
     { S#      S#,
       P#      P#,
       QTY     QTY }
     KEY { S#, P# } ... ;
```

这个例子显示了一个带有一个复合候选码的关系变量（也就是候选码涉及一个以上的属性）。与此相对的是简单候选码。

```
VAR ELEMENT BASE RELATION { NAME    NAME,
                            SYMBOL  CHAR,
                            ATOMIC# INTEGER }
                          KEY { NAME }
                          KEY { SYMBOL }
                          KEY { ATOMIC# } ;
```

这个例子显示了一个同时有几个不同的（简单）候选码的情况。

```
VAR MARRIAGE BASE RELATION { HUSBAND                 NAME,
                             WIFE                    NAME,
                             DATE /* of marriage */  DATE }
/* assume no polyandry, no polygyny, and no husband and */
/* wife marry each other more than once ...             */
                     KEY { HUSBAND, DATE }
                     KEY { DATE, WIFE }
                     KEY { WIFE, HUSBAND } ;
```

这个例子显示了有多个复合（交错的）候选码的情况。

当然，如在 8.4节中指出的，候选码定义只是关系变量约束的缩写形式，因为从视图的语法角度来看，候选码的概念是重要的，所以这种缩写很有用。尤其是候选码提供了一种在关系模型中的元组级的寻址方式 —— 系统中唯一精确指示某些指定元组的方法就是通过候选码。比如：

㊀ 候选码要求唯一性的另一个原因是与外码相一致。指向可约的候选码（如果可能）的外码也具有可约性，而且含有这样一个外码的关系变量显然是违反规范化准则的（参见第 11章）。

```
S WHERE S# = S# ( 'S3' )
```

可以保证至多产生一个元组（更准确地说，它产生了至多包含一个元组的一个关系）。相反，表达式：

```
S WHERE CITY = 'Paris'
```

得到多个元组。可以说，候选码对于关系系统的正确操作的重要性，是可以和内存地址对于计算机正确操作的重要性相提并论的。结果为：

1) 没有候选码的关系变量 —— 也就是允许重复元组的关系变量 —— 在某些环境中会出现异常现象。
2) 不支持候选码概念的系统有时会表现得像一个非关系系统，虽然它所处理的关系变量，是真正的关系变量，而且不允许重复元组。

上面所说的“反常”和“像非关系系统”的现象与视图的更新和优化有关（分别参见第 9 章和第 17章）。

再指出几点，然后结束这一小节的讨论：

- 候选码的超集是超码（superkey）。属性集{S#，CITY}是关系变量 S的超码。超码具有唯一性，但不具有不可约性（当然，候选码可以看作是超码的一个特例）。
- 如果*SK* 是关系变量 *R*的超码，*A*是*R*的一个属性，那么函数依赖*SK*→*A*在*R*中为真（这一概念在第 10章中进行深入讨论）。实际上可以定义 *SK*上的一个子集为超码，而使函数依赖 *SK*→*A*对*R*上的任意属性 *A*为真。
- 最后，请注意不要把逻辑上的概念“候选码”和物理中的概念“唯一索引”相混淆（虽然后者常被用来实施前者）。换言之，候选码上不一定要有索引（或任何其它的物理存取路径）。但实际上可能会有这样的存取路径，有还是没有并不在关系模型的讨论范围之内。

2. 主码和替换码

很明显，一个关系变量可以有多个候选码。这种情况下，此关系模型就要求 —— 至少是对于基本关系变量 —— 它们中的一个被选为主码，其它的就被称为替换码（alternate key）。如在“元素”例子中，我们选取 {SYMBOL}作为主码；{NAME}和{ATOMIC#}就是替换码。在只有一个候选码的情况下，对于基本关系变量关系模型，要求这个候选码被指定为主码，因此，基本关系变量总是有一个主码。选取一个候选码作为主码（当存在选择时），在多数情况下是一个好主意，但并不是所有的情况。在参考文献 [3.3]中有这方面的详细讨论；这里，我们只说明一种情况，在这种情况中哪一个候选码被选为主码是任意的（引用 Codd的[8.8]中的话“以简便性为原则，但这是在关系模型研究之外的”）。在本书中，有时会定义主码，有时不会（当然，我们总是会定义一个候选码）。

3. 外码

简单地说，外码就是定义在关系变量 *R2*上的一组属性集，它们的值要求与关系变量 *R1*上的候选码的值相匹配。比如，考虑在关系变量 SP中的属性集{S#}（仅包含一个属性）。显然，如果一个 {S#}的值要在关系变量 SP中出现，仅当同样的值出现在关系变量 S的唯一候选码 {S#}中（不能让不存在的供应商有一个发货量）。相应地，一个对 {P#}的给定的值要能在关系变量 SP中出现，仅当同样的值出现在关系变量 P的唯一候选码 {P#}中（不能让不存在的零件有一个发货量）。下面用一些例子来解释这些定义：

• 设*R2*是关系变量。那么 *R2*的外码是它的一个属性集，比如说 *FK*，则：

a) 存在带有候选码 *CK*的关系变量 *R1*（*R1*，*R2*可能相同），且

b) 无论何时，*R2*中的任一个 *FK*值都在 *R1*的*CK*中有一个相同的值。

这里有如下要点：

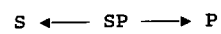
1) 这个定义要求在外码中出现的每一个值都要在相应的候选码中出现（通常是主码，但这不是一成不变的）。注意，相反的情况不是必需的；也就是说，在对应于某个外码所对应的候选码中出现的值不一定要在相应的外码中出现。比如在“供应商-零件”例子中（参见表 3-8），供应商号 S5在关系变量 S中出现，而关系变量 SP中没有，因为 S5目前没有供货。

2) 外码是简单的还是复合的取决于对应的候选码是简单的还是复合的。

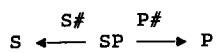
3) 外码所含的属性与相应的候选码中的属性有相同的名字和类型。

4) 术语：外码表示一个对包含相应候选码的某个元组的参照（被参照元组）。确保数据库中不含有无效外码的问题被称为参照完整性问题。外码值必须与相对应的候选码的值相匹配的约束被称为参照约束。含有外码的关系变量被称为参照关系变量，含有相应候选码的关系变量称为被参照关系变量。

5) 参照图：考虑“供应商-零件”例子。人们可以把其中的参照约束用下面的参照图表示出来：

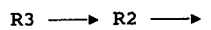


每一个箭头表示一个外码，这个外码从此关系变量出发，参照箭头指向的关系变量中的某个候选码。注意，为清晰起见，用构成候选码的属性名来标识参照表是可行的 ㊀。比如：

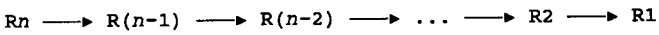


在本书中，只有当省略标识会引起混淆的情况下才进行标识。

6) 某个关系变量可以是参照关系变量，也可以是被参照关系变量。如在 *R2*的例子中：



为此，引入参照路径(referential path)这一术语。设有关系 *Rn*, *R*(*n*-1),...*R2*, *R1*, 存在从 *Rn*到*R*(*n*-1)的参照约束，从 *R*(*n*-1)到*R*(*n*-2)的参照约束，.....，从*R2*到*R1*的参照约束：



那么，从 *Rn*到*R1*的箭头链表示从 *Rn*到*R1*的参照路径。

7) 外码定义中的 *R2*与*R1*不必是不同的。也就是说，外码可以参照关系变量自身的候选码。考虑下面的外码定义（一会儿我们再解释语法，但它应该是可以看懂的）：

```
VAR EMP BASE RELATION
          { EMP# EMP#, ..., MGR_EMP# EMP#, ... }
    PRIMARY KEY { EMP# }
    FOREIGN KEY { RENAME MGR_EMP# AS EMP# } REFERENCES EMP ;
```

这里，MGR_EMP#代表此雇员（用 EMP#标识）的经理的雇员号；比如，雇员 E4这个元组，它的 EGR_EMP#属性值为 E3，它表示指向雇员号为 E3的元组的参照（注意，为了实现上面第三段中的要求，这里要给外码中的某个属性重命名）。这样的关

㊀ 也可以给外码起名，然后用这些名字来标识箭头。

系变量可以称为是自参照（self-referencing）的。练习：给这个例子增加一些数据。

8) 像EMP这样的自参照的关系变量只是一种更一般的情形的特例 —— 也就是，这里可以有一个参照的环。在关系变量 *Rn*,*R*(*n*-1)..., *R2*,*R1*中，当从存在 *Rn*到*R*(*n*-1), 从*R*(*n*-1)到*R*(*n*-2),....., 从*R2*到*R1*，从*R1*回到*Rn*的参照时，就形成了一个参照的环。更简洁地说，如果存在一个从关系变量 *Rn*回到它自身的参照路径，这里就有一个参照的环。

```
Rn ──► R(n-1) ──► R(n-2) ──► ... ──► R2 ──► R1 ──► Rn
```

9) “外码-候选码”的匹配可以比作是使数据库结合在一起的粘合剂。它的另一个说法是：它们的匹配表示了元组之间的某种关系。仔细观察，并不是所有的关系都通过码表现出来。比如，在供应商和零件之间有“地点相同”的关系，由 S和P的城市属性表现出来；当某个供应商和某种零件放在同一城市中时，就说它们有地点相同的关系，但这并不通过码之间的关系表现出来。

10) 由于历史原因，外码只定义给基本关系变量。这一情况引起了一些问题（参见第9章，9.2节关于互换性准则的讨论）。这里我们不实施这样的限制；但是，为简单起见，我们把讨论限于基本关系变量（这里有些区别）。

11) 关系模型起初是要求外码参照的，不仅是候选码，还要主码（参见 [8.8]）。由于这是不必要的，后来放弃了这些限制，虽然实践中它会是一种好习惯 [8.13]。实际中我们经常仍会遵守这一规则。

12) 除了外码的概念，关系模型还包括下列规则：

- 参照完整性：数据库不能含有无匹配的外码[⊖]。

术语：“未匹配外码”表示在被参照表中不含有相应的外码；换句话说，约束只是简单的要求：如果 *B*参照*A*，则*A*必须存在。

下面是外码定义的语法：

```
FOREIGN KEY { <item commalist > } REFERENCES <relvar name>
```

这一句出现在参照关系变量定义中。注意：

- 每一个*<item>*是属性名，或者是这种形式的表达式

```
RENAME <attribute name> AS <attribute name>
```

（对于RENAME这种情况，请参见前面“自参照”的例子中的关系变量 EMP）。

- *<relvar name>*指被参照的关系变量。

在前面已经给出了很多例子（如第 3章中的图3-9所示）。注意：正如8.5节中指出的，外码定义只是一些数据库约束（在自参照的关系变量中，是关系变量约束，）的缩写 —— 除非外码定义被扩展，以至包括某些“参照行为”（referential action）,那时它就不只是完整性约束了。参见下两小节。

4. 参照行为

考虑下面的语句：

```
DELETE S WHERE S# = S# ( 'S1' ) ;
```

假设DELETE能像所描述的那样做，也就是它删除供应商元组 S1，既不多，也不少。假设

⊖ 参照完整性可以被看作是一个元约束 (metaconstraint)：意思是给定的数据库必然服从一定的完整性约束，特别对现在讨论的数据库，在数据库中不可以违反这些准则。顺便提一下，数据库也经常含有另外一个元约束：实体完整性，它与空值有关。在第 18章再讨论这个问题。

S1对应一些发货元组（见图 3-8），并且程序没有删除这些元组，当系统检查从发货到供应商的参照完整性时，就会发生错误。

注意：因为这里的参照约束是数据库约束，检查将被延迟到 COMMIT时进行 —— 至少理论上是这样（实际上检查可能是在 DELETE后就进行，但这时发生的违反约束不一定会产生错误；它只是表示在 COMMIT时系统还要进行一次检查）。

然而，替代的方法是存在的，在某些情况下它可能更合适。那就是让系统进行一次适当的补偿性操作以保证全部结果满足约束。在此例中，显然的补偿操作是同时自动删除“发货”元组。为达到这一目的，将外码定义扩展如下：

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                      ON DELETE CASCADE ;
```

ON DELETE CASCADE 定义了指定外码的删除规则，CASCADE是此规则的参照行为。这些说明的意思是：在供应商关系变量上的删除操作应“级联”（cascade）删除相匹配的发货元组。

另一个常见的参照行为是 RESTRICT（与关系代数中的 *restrict*操作符没有关系）。这个例子中，RESTRICT表示删除被限制为只当不存在相应的元组时才进行（否则被拒绝）。在外码定义中忽略参照行为相当于指定行为是“没有动作”（NO ACTION），意思是 DELETE只按所描述的去作，不多作也不少作（当指定为 NO ACTION时，删除了一个有匹配的发货元组的供应商元组，就会引起违反参照完整性）。几个要点：

1) 删除不是唯一需要参照行为的操作。比如，当存在相匹配的发货元组时对供应商编号更新。显然，和删除规则一样，需要一个更新规则。通常，DELETE和UPDATE是相同的：
   - CASCADE：在对应的发货元组上级联更新。
   - RESTRICT：被限制为只当没有相匹配的发货元组时才可更新（否则被拒绝）。
   - NO ACTION：更新只按所描述的那样操作。
2) 当然，并不是只有 CASCADE、RESTRICT和NO ACTION三种参照行为，它们只是常用的三种。原则上，可以有多种不同的反应，比如，对某个供应商元组的删除，有：
   - 信息被写到文档数据库。
   - 与被删除元组相对应的发货元组被转移到其它的供应商元组中。

   等等。不可能为所有可预见的结果定义一个操作，通常可以通过 CALL *proc*(...)的形式来指定参照行为，其中 *proc*是一个用户自定义的过程。

   注意：此过程的执行应被认为是引起此完整性检查的事务的一部分，更进一步，此过程之后要再进行完整性检查（此过程要明显地不会引起违反完整性约束）。
3) 设*R2*、*R1*分别为参照关系变量和被参照关系变量：

   R2 ⟶ R1

   设使用的删除规则为 CASCADE。那么在 *R1*上删除某个元组也就暗示在 *R2*上删除某些相应的元组。假设 *R2*相应地被另一个关系变量 *R3*所参照：

   R3 ⟶ R2 ⟶ R1

   那么蕴含的对 *R2*的删除就好像在 *R2*上直接删除一样；也就是说，它也遵守定义

在其参照约束上的删除规则。如果蕴含删除失败（由于 *R2*到*R3*的删除规则或其它原因），则整个操作失败，数据库恢复原状。等等，如此递归地进行，可到达任意层次。

如果*R2*的外码中有被 *R3*的候选码所参照的属性，则相似的方法也适用于级联更新（CASCADE UPDATE）。

4) 从前面所讲的可知，从视图的逻辑观点来看，数据库更新是原子性的（ atomic）—— 或者全做或者全部不做。即使是在多个关系变量上的多个更新操作，也会因CASCADE的参照行为而实现原子性。

5. 触发过程

大家可能已经意识到（如前面“ 用户定义过程 ”小节所述），参照行为的概念已经使我们超越完整性约束，而到达了触发过程的范畴。触发过程（通常称为触发器）是当指定的事件发生或触发条件满足时被自动引用的过程。典型的触发条件是数据库的更新操作，但也可能是指定的意外情况发生时（特别是违反了某个完整性约束），或经过了一定时间后。 CASCADE参照行为提供了一个简单的触发过程的例子。

通常，触发过程可被用于更广的范围，而不只是本章所讲的完整性（在 [8.1]中可以找到这些应用的列表）。关于它有很多可探讨的，这已超出了本章的范围（更多信息，请参见[8.22]）。这里只是说，触发过程在很多方面都有用，但在数据库完整性方面它不是最好的，原因很明显（可能的话，声明的方法会更好）。注意：这些评论不是说参照行为不好。虽然参照行为确实会引发了某些过程，但这些过程至少应该是公开声明的。

8.9 SQL 对完整性的支持

SQL的完整性分类与前面 8.1~8.5节中讨论的有很大的不同。首先，它把约束分为三大类：

- 域约束（ domain constraint）
- 基本表约束 (base table constaint)
- 一般约束 (general constraint，即断言)

但是，域约束不同于我们所说的类型约束，基本表约束不同于关系变量约束，断言也不同于数据库约束。实际上：

- SQL并不真正支持类型约束（因为它并不真正支持类型，除了几个内置的之外）。
- SQL中的域约束是我们不想见到的属性约束的一般形式（回忆一下， SQL风格的域不是所说的关系意义上的域）。
- SQL的基本表约束和断言（它们之间可以有效地相互转化）勉强算是所说的关系变量约束和数据库约束的等价形式。

可以看出， SQL不支持动态约束，目前也不支持触发过程，虽然在 SQL3中已提供了某些支持（见附录 B）。

1. 域约束

SQL风格的域约束是一个可用于所有定义在这个域上的列的约束。下面有一个例子：

```
CREATE DOMAIN COLOR CHAR(6) DEFAULT '???'
   CONSTRAINT VALID_COLORS
      CHECK ( VALUE IN
            ( 'Red', 'Yellow', 'Blue', 'Green', '???' ) ) ;
```

假定像下面这样为基本表 P创建表：

```
CREATE TABLE P ( ... , COLOR COLOR, ... ) ;
```

那么，如果用户插入一个零件记录而不提供颜色（ COLOR）值，颜色值将被缺省地置为“ ??? ”。或者，用户指定了一个非法的颜色值，则操作失败，系统将产生一个含有约束VALID_COLORS名字的诊断信息。

现在，看8.2节中的一个例子，这里，域约束只是域上的值的列举，而VALID_COLORS例子也遵守这一形式。通常， SQL允许域约束包含任意复杂的布尔表达式，这样，域 *D*的合法取值就可以是依赖于在某个表 *T*中出现的值。你可能会考虑这种没有保证的许可究竟意味着什么。

2. 基本表约束

SQL的基本表约束可以是下面的任一种：

- 候选码定义
- 外码定义
- “ 检查约束 ” 定义

下面将详细讨论每种情况。注意，这些定义都可以在前面加上“ CONSTRAINT <*constraint name*> ”，由此为新约束起一个名字（对域约束也可以这样，如在 VALID_COLORS中所见）。为简化，后面将忽略这一选项。

1) 候选码：候选码定义形式为

```
UNIQUE ( <column name commalist> )
```

或

```
PRIMARY KEY ( <column name commalist> )
```

<*column name commalist*>两种情况都不为空。一个基本表只能指定一个主码，但可以指定多个 UNIQUE。当是主码时，指定的列会被认为是非空的，即使“ 非空 ” 没有被明确表示出来（参见检查约束的讨论）。

2) 外码：外码定义形式为

```
FOREIGN KEY ( <column name commalist> )
   REFERENCES <base table name> [ ( <column name
 [ ON DELETE <referential action> ]
 [ ON UPDATE <referential action> ]
```

<*referential action*>可以是 NO ACTION（缺省）、CASCADE、SET DEFAULT 或 SET NULL。对于SET DEFAULT和SET NULL的讨论将在第 18章中讲述。当外码所参照的候选码不是主码时，第二个 <*column name commalist*>是必需的。注意：在以逗号分隔列的表中，外码-主码匹配是以列的位置“（从左到右）”，而不是列名为基础的。

3) 检查约束：检查约束的定义形式为

```
CHECK ( <conditional expression> )
```

当在基本表 *T*中插入一个新行 *r*时，如果检查约束的条件表达式对于 *r*判断为 false，则认为违反了 *T*上的检查约束。注意：条件表达式是布尔表达式的 SQL形式，在附录 A中有详细解释。尤其注意（在当前的上下文中）条件表达式可以相当复杂 —— 可以不只是局限于 *T* 的条件，而是参照于数据库中任何成份的条件。你可能又会考虑这种没有保证的许可意味着什么。

下面是CREATE TABLE的一个例子，它包含基本表约束的三种形式：

```
CREATE TABLE SP
     ( S# S# NOT NULL, P# P# NOT NULL, QTY QTY NOT NULL,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       FOREIGN KEY ( P# ) REFERENCES P
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       CHECK ( QTY > 0 AND QTY < 5001 ) ) ;
```

这里假设域 S#、P#和QTY已经被定义， S#和P#已被显式定义为表 S和P的主码。而且，通过运用下面的缩写形式

```
CHECK ( <column name> IS NOT NULL )
```

检查约束定义可以被在列定义中的一个简单的 NOT NULL描述来代替。在这个例子中，用三个NOT NULL代替了繁琐的检查约束定义。

下面是一个看上去有点怪的结论：当某个基本表是空时，则其上的 SQL基本表约束总是被认为是满足的 —— 即使约束为“ 表必须是非空的 ”。

3. 断言

本节剩余部分，我们来关注第三种情况：一般约束或断言。断言是通过 CREATE ASSERTION语法定义的：

```
CREATE ASSERTION <constraint name>
       CHECK ( <conditional expression> ) ;
```

下面是DROP ASSERTION的语法：

```
DROP ASSERTION <constraint name> ;
```

注意：与本书中所讲过的其它 SQL DROP操作（DROP DOMAIN、DROP TABLE、DROP TABLE和DROP VIEW）的语法不同，删除断言不提供 RESTRICT和CASCADE选项。

下面是CREATE ASSERTION的例子：

1) 供应商的状态值至少是 5：

```
CREATE ASSERTION IC13 CHECK
       ( ( SELECT MIN ( S.STATUS ) FROM S ) > 4 ) ;
```

2) 零件的重量为正：

```
CREATE ASSERTION IC18 CHECK
       ( NOT EXISTS ( SELECT * FROM P
                      WHERE  NOT ( P.WEIGHT > 0.0 ) ) ) ;
```

3) 所有的红色零件被存放在伦敦：

```
CREATE ASSERTION IC99 CHECK
       ( NOT EXISTS ( SELECT * FROM P
                      WHERE  P.COLOR = 'Red'
                      AND    P.CITY <> 'London' ) ) ;
```

4) 发货的总重量（零件重量乘以个数）不超过 20 000：

```
CREATE ASSERTION IC46 CHECK
       ( NOT EXISTS ( SELECT * FROM P, SP
                      WHERE  P.P# = SP.P#
                      AND   ( P.WEIGHT * SP.QTY ) > 20000.0 ) ) ;
```

5) 供应商状态小于 20时提供的零件数不能超过 500：

```
CREATE ASSERTION IC95 CHECK
       ( NOT EXISTS ( SELECT * FROM S, SP
                      WHERE  S.STATUS < 20
                      AND    S.S# = SP.S#
                      AND    SP.QTY > 500 ) ) ;
```

4. 延迟的检查

SQL的完整性约束分类方案在何时检查完整性上也与本书所讲的不同。在本书中，数据库约束在COMMIT时检查，其它约束是立即检查。在SQL中，约束可以被定义为“可延迟的”和“不可延迟的”；如果约束是可延迟的，则又可分为“最初延迟”（INITIALLY DEFERRED）和“最初立即”（INITIALLY IMMEDIATE）。它用来定义事务开始时约束的状态。不可延迟约束通常是立即执行的，但可延迟约束的“可延迟”是由下面的语句动态打开或关闭的：

```
SET CONSTRAINTS <constraint name commalist> <option> ;
```

其中<*option*>可以是IMMEDIATE或DEFERRED。下面是一个例子：

```
SET CONSTRAINTS IC46, IC95 DEFERRED ;
```

可延迟约束只有当处于“IMMEDIATE”状态时才会被检查。设置可延迟约束到立即状态会使约束被立即检查。检查失败，则“SET IMMEDIATE”也失败。COMMIT强迫所有可延迟约束为SET IMMEDIATE状态；如果任何完整性检查失败，则事务回滚。

8.10 小结

本章讨论了完整性这一重要的概念。完整性问题是保证数据库中数据的正确性和相容性的问题（我们对这一问题的声明式的解决方法更感兴趣）。现在，大家可能已经认识到，完整性在上下文中是“在语义学中的理解”：完整性约束（尤其是关系变量和数据库谓词）表示了数据正确的含意。正如在第8.6节中所说的，完整性是很重要的概念。

完整性约束被分为四类：

- 类型约束定义了指定数据类型（域）的合法值，在调用相应的选择子时被检查。
- 属性约束定义了指定属性的合法值，任何情况不能违反（设类型约束已被检查）。
- 关系变量约束定义了指定关系变量的合法值，当关系变量被更新时检查。
- 数据库约束定义了指定数据库的合法取值，在COMMIT时检查。

某个关系变量上的关系变量约束和它们之间的逻辑与（AND）是这个关系变量的（内部）关系变量谓词。关系变量谓词是关系变量在系统中被理解的形式，也是关系变量上更新被接受的标准。黄金法则描述了如果一个更新操作将使关系变量处于违反其上某个谓词的状态，则此操作是不可进行的。整个数据库处于受数据库谓词制约的状态，如果一个事务操作将使数据库处于违反其上某个谓词的状态，则此事务操作是不可进行的。

接下来，本章简要地描述了动态约束的基本思想（其它约束是静态约束）；然后讨论候选码、主码、替换码和外码的重要概念。候选码满足唯一性和不可约性，而且每个关系变量至少有一个候选码（没有例外）。外码的值必须与相对应的候选码的值相匹配的约束称为参照约束；本章深入研究了参照完整性的几个含意，包括参照行为（特别是CASCADE）。接下来讨论了触发过程这一概念。

最后，本章讨论了SQL中相应的各个方面。SQL支持域约束、基本表约束和断言（一般约束），而且，SQL中的基本表约束包含对码的特殊支持。

练习

8.1 用8.2~8.5节中介绍的语法，写出“供应商-工程”数据库中的一些完整性约束：

a. 合法的城市有： London, Paris, Rome, Athens, Oslo, Stockholm, Madrid, and Amsterdam.

b. 合法的供应商编号为大于两个字母的字符串，其中第一个字母为“ S”，其余部分为从0到9999范围内的十进制整数。

c. 红色零件重量小于 50磅。

d. 任何两个工程不能位于同一城市。

e. 任一时刻， Athens至多只能有一位供应商。

f. 每一次发货的数量不能超过所有发货的数量的平均值的两倍。

g. 供应商状态值最高的供应商和状态值最低的供应商不能在同一城市中。

h. 每个工程所在的城市中至少有一位供应商。

i. 每个工程所在城市的中至少有一位向此工程提供零件的供应商。

j. 至少有一种红色的零件。

k. 供应商状态平均值大于 18。

l. 在London的供应商必须提供编号为 P2的零件。

m. 至少有一种红色零件的单重小于 50磅。

n. 在London的供应商提供的零件种类要多于在 Paris的供应商提供的零件种类。

o. 在London的供应商提供的零件总数量要多于在 Paris的供应商提供的零件总数量。

p. 发货的数量不能被减小（ UPDATE）到小于它原先的一半。

q. 位于Athens的供应商只能搬到 London 或Paris，London的供应商只能搬到 Paris。

8.2 对练习8.1中的每一个答案，说出是关系变量约束，还是数据库约束。

8.3 对练习8.1中的每一个答案，举出违反约束的操作的例子。

8.4 设CHAR(5)、CHAR(3)分别指长度为 5和3 的字符串，那么它们是几种数据类型，一种还是两种？

8.5 设*A*和*B*是两个关系变量，说出下面每一种情况的候选码：

a. `A WHERE ...`

b. `A {...}`

c. `A TIMES B`

d. `A UNION B`

e. `A INTERSECT B`

f. `A MINUS B`

g. `A JOIN B`

h. `EXTEND A ADD exp AS Z`

i. `SUMMARIZE A PER B ADD exp AS Z`

j. `A SEMIJOIN B`

k. `A SEMIMINUS B`

假设*A*和*B*满足上述的每一种操作（如，在 UNION操作下它们是同一种类型）。

8.6 设*R*是一个*n*度的关系变量，则 *R*可能有的候选码的最大数是多少？

8.7 设*R*是一个关系变量，它的仅有的合法值为 0度关系DEE和DUM，请指出它的候选码

取值。

8.8 本章主要讨论了 DELETE和UPDATE的外码操作规则，但没有讨论 INSERT时外码的变化，为什么？

8.9 用图4-5中“供应商-零件-工程”例子中的数据，说出下列操作会引起什么结果。

a. 更新工程 J7，置CITY值为 New York;

b. 更新零件 P5，置P#值为 P4；

c. 更新供应商 S5，置S#为S8，此时的参照行为是 RESTRICT；

d. 删除供应商 S3，此时的参照行为是 CASCADE；

e. 删除零件 P2，此时的参照行为是 RESTRICT；

f. 删除工程 J4，此时的参照行为是 CASCADE；

g. 更新供货量 S1-P1-J1，置S#为S2；

h. 更新供货量 S5-P5-J5，置J#为J7；

i. 更新供货量 S5-P5-J5，置J#为J8；

j. 插入供货量记录 S5-P6-J7；

k. 插入供货量记录 S4-P7-J6；

l.插入供货量记录 S1-P2-jjj (jjj表示缺省工程号)。

8.10 某个教育数据库包含一个公司室内培训的信息。对于每一门课程，数据库中都有它的先导课程的信息；对于每一门开设的课程，数据库中都有它的教师和学生的信息；数据库中也包含雇员的信息。如下所示：

```
COURSE      { COURSE#, TITLE }
PREREQ      { SUP_COURSE#, SUB_COURSE# }
OFFERING    { COURSE#, OFF#, OFFDATE, LOCATION }
TEACHER     { COURSE#, OFF#, EMP# }
ENROLLMENT  { COURSE#, OFF#, EMP#, GRADE }
EMPLOYEE    { EMP#, ENAME, JOB }
```

PREREQ 关系变量中，SUP_COURSE#是SUB_COURSE #的先导课程。为这个数据库画一个参照图，写出相应的数据库定义（即写出类型集和关系变量的定义）。

8.11 下面表示一个包含“部门”和“雇员”信息的数据库：

```
DEPT  { DEPT#, ..., MGR_EMP#, ... }
EMP   { EMP#, ..., DEPT#, ... }
```

每个部门有一个经理（ MGR_EMP#）；每一雇员属于某个部门（ DEPT#)。画出参照图，写出数据库定义。

8.12 下面表示一个包含“雇员”和“程序员”信息的数据库：

```
EMP   { EMP#, ..., JOB, ... }
PGMR  { EMP#, ..., LANG, ... }
```

每一程序员都是雇员，反之却不然。画出参照图，写出数据库定义。

8.13 本章没有讨论这样一种情况：当试图删除已存在的关系变量约束、类型约束或其它约束时，会发生什么现象。请你回答一下。

8.14 写出练习 8.1的SQL表示。

8.15 比较SQL的完整性支持和本章讲述的完整性机制。

参考文献和简介

8.1 Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom: “Static Analysis Techniques for

Predicting the Behavior of Active Database Rules," *ACM TODS 20*, No. 1 (March 1995).

这篇论文继续参考文献 [8.2]、[8.5]中的工作，讨论“专家数据库系统”（这里称为主动数据库系统）。尤其是它探讨了 IBM Starburst原型的规则机制（参见 [17.50]、[25.14], [25.17], [25.21~25.22]和[8.22]）。

8.2 Elena Baralis and Jennifer Widom: " An Algebraic Approach to Rule Analysis in Expert Database Systems," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

根据该文，“专家数据库系统”是支持“条件 /行为规则”的数据库（用本文中的术语，条件是触发条件，行为是触发过程）。这个体系存在的问题是：它的行为是不可预见的。这篇论文提供的方法可以在执行前检测存在的规则是否能结束和汇合（termination and confluence）。结束表示保证此程序不会永远运行下去；汇合表示数据库的最终状态与规则的检查顺序无关。

8.3 Philip A. Bernstein, Barbara T. Blaustein, and Edmund M. Clarke: " Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (October 1980).

对某一特定类型提出了一个有效的完整性实施方法。有这样一个例子：集合 *A*中的每一个值必须小于集合 *B*中的每一个值”。可以观察到：给出的约束在逻辑上等同于另一个约束“集合 *A*中的最大值必须小于集合 *B*中的最小值”。将这一类约束组织起来，用隐藏的变量来自动表示最大值与最小值的关系，无论对 *A*还是*B*，用这种方式，系统都能有效地将实施约束时进行比较的次数减少到一次 —— 但代价是要维护隐藏的变量。

8.4 O. Peter Buneman and Erik K. Clemons: " Efficiently Monitoring Relational Databases," *ACM TODS 4*, No. 3 (September 1979).

该文谈到了有效实施触发过程（这里称为 alerters）的方法 —— 尤其是在判断触发条件是否满足这一问题上。其中给出了检测可能不满足触发条件的更新的一种方法（“避免”算法）。也讨论了当避免运算失败时减少处理的技巧：对相关元组的一个子集判断是否满足触发条件。

8.5 Stefano Ceri and Jennifer Widom: " Deriving Production Rules for Constraint Maintenance," Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

描述了基于 SQL的约束定义语言，并给出了识别可能违反约束的操作的算法（在参考文献 [8.11]中给出了这种算法的初步介绍，这种算法的存在表明：无需显式地通知DBMS何时需要进行完整性检查。本书没有为读者提供这方面的信息）。该文也介绍了优化和正确性方面的知识。

8.6 Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca: "Automatic Generation of Production Rules for Integrity Maintenance," *ACM TODS 19*, 3 (September 1994).

该文是在参考文献 [8.5]的基础上进行讨论的，介绍当数据库因违反完整性而造成损坏时如何进行自动修复。约束与下面的内容一起被编译成为产生式规则（production rule）。

1) 可能违反约束的一系列操作。

2) 当违反约束时会被判断为真的布尔表达式（主要是原约束的否定形式）。

3) SQL的修复过程。

该文也包括了一些相关工作的概述。

8.7 Roberta Cochrane, Hamid Pirahesh, and Nelson Mattos: " Integrating Triggers and Declarative Constraints in SQL Database Systems," Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai (Bombay), India (September 1996).

" 小心定义触发和声明约束之间的相互作用，以避免它们之间不一致的行为，并为用户提供这种相互作用的可理解的模型。这篇论文定义了这样一种模型 "。此模型已在DB2中实现，并被接受为未来 SQL标准的模型（ SQL3）（见附录 B）。

8.8 E. F. Codd: " Domains, Keys, and Referential Integrity in Relational Databases," *InfoDB 3*, No. 1 (Spring 1988).

该文讨论域、主码和外码。由于 Codd是这三个概念的创造者，此文显然具有权威性；但仍有一些问题未作解答。对于必须选取一个候选码作为主码的要求，作者给出如下描述：" 不提供这一准则的支持好像是在用这样一种寻址方式的计算机……每次特定的事件发生时，它都改变基数（如遇到的地址是素数时）"，但如果我们接受这一论点，那为什么不用一种任何情况下都清晰的寻址方式呢？通过供应商号来寻址供应商和通过零件号来寻址零件，这样不觉得好用吗？更不用说发货了，它涉及复合的寻址方式（实际上，对于全程唯一的寻址方法有很多内容。参见 13章中对 [13.16]的评论）。

8.9 C. J. Date: " Referential Integrity," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981). Republished in revised form in *Relational Database: Selected Writings*. Reading, Mass: Addison-Wesley (1986).

该文介绍参照行为（尤其是 CASCADE 和RESTRICT），即本章 8.8节中讨论的。注意：该文的初版（ VLDB 1981）和修正版的区别是在于初版遵从参考文献 [13.6]，允许一个外码参照多个关系变量，由于参考文献 [8.10]中所说的原因，修正版中将之取消了。

8.10 C. J. Date: " Referential Integrity and Foreign Keys" (in two parts), in *Relational Database Writings 1985-1989*. Reading, Mass: Addison-Wesley (1990).

该文的第一部分讨论了参照完整性的概念，并提供了更好的定义方式（和基本原理）。第二部分提供了进一步讲述，并给出了应用的办法；尤其是讨论了外码重叠；复合外码中部分为空；以及有共同边界的参照路径（如不同的参照路径有相同的起点和终点）时引起的问题。注意：该文有小部分与参考文献 [8.13]相抵触。

8.11 C. J. Date: "A Contribution to the Study of Database Integrity," in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

" 本文试图为完整性问题建立一个结构，提出一个完整性的分类方案，用这一方案来澄清数据完整性概念蕴含的重要性；概述了用一门具体语言来明确表达完整性约束的方法，指出了几个需要进一步研究的领域 "。本章的部分内容是基于此论文的早期版本，但它的分类方案应该被本章 8.2~8.5节中的修正方案取代。

8.12 C. J. Date: " Integrity," Chapter l l of C. J. Date and Colin J. White, *A Guide to DB2* (4th edition) [4.20]. Reading, Mass: Addison-Wesley (1993).

IBM的DB2产品确实提供了声明主码、外码的支持（实际上，它只是最早提供这一功能的产品之一）。正像该文中说的， DB2提供这一功能并不是因为实施中有什么不便，而是为了保证可预测的行为。给出一个简单的例子。设关系变量 *R*现在只含有两个元组，它们的主码分别为 1和2，考虑更新请求“将它们的主码都乘 2”，正确的结果是*R*的两个元组主码分别为 2和4。如果系统先将 2更新为4（将2用4替换），然后更新主码1，则操作成功；相反，如果系统试图先更新 1（将它用2代替），就会违反唯一性的约束，请求失败（数据库恢复原状）。换言之，请求的结果是不可预测的。为避免这种不可预测性， DB2简单地将会引起多种可能结果的情况设为不合法。不幸的是，这种限制是过于严格的（ [8.17]）。

注意，像前面的例子中所说的， DB2是进行“飞行中检查”的（ inflight checking）——也就是说，在每一次单个元组的更新后进行检查。这种飞行中检查是逻辑错误的（参见第5章5.4节中关于更新操作的讨论）；它是因为实现方面的原因才这样做的。

8.13 C. J. Date: “ The Primacy of Primary Keys: An Investigation,” in *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).

该文是基于这样一种观点“使一个候选码比其它的更重要并不是一个好主意”。

8.14 M. M. Hammer and S. K. Sarin: “ Efficient Monitoring of Database Assertions,” Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas (May/June 1978).

该文描述了能够产生一种完整性检查过程的算法，这种算法比以前的在一次更新后简单地强制进行检查效率更高。在编译时，这一检查就被包含到应用目标代码中。有时，可能会发现无需运行时检查，即使需要，也有很大的可能通过多种途径显著减少数据库存取次数。

8.15 Bruce M. Horowitz: “A Run-Time Execution Model for Referential Integrity Maintenance,” Proc. 8th IEEE Int. Conf. on Data Engineering, Phoenix, Ariz. (February 1992).

我们都知道下列三种结构的某些组合：

1) 参照结构(通过参照约束相关联的关系变量的集合)；
2) 外码的DELETE和UPDATE规则；
3) 数据库中的实际数据值。

能导致相冲突的情况，也会在实施时引起一些不可预期的行为（详细内容请参见[8.10]）。有三种方法来解决这一问题：

a) 将之留给用户解决；
b) 让系统检测并拒绝运行时可能会引起冲突的结构定义；
c) 让系统检测并拒绝运行时的错误。

方法a是不可取的， b又过分小心 [8.12, 8.17]；Horowitz建议方法c，该文给出了一些运行时要遵守的规则，并证明了它们的正确性，但这些运行时检测的性能没有被考虑。

Horowitz是SQL/92定义小组中活跃的一员，其标准中参照完整性部分有效地暗示此文中的建议应该得到支持。

8.16 Victor M. Markowitz: “ Referential Integrity Revisited: An Object-Oriented Perspective,” Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

该文的标题“面向对象的观点”表明了作者开放的立场：参照约束支持了面向对象结构的关系表示。该文并不是真正讨论面向对象的，但它从实体/关系图出发，展示了一个会产生数据库定义的算法，在此算法中，[8.10]中定义的一些问题（如交迭的码）将保证不会出现。

此文从参照完整性的观点讨论了三种商业产品（DB2、Ingres和Sybase，大约是在1990年）。DB2提供了声明支持，功能是有限制的；Sybase和Ingres提供过程支持（分别通过触发器和规则），限制要比DB2少，但较难用（虽然Ingres的技术上声称优于Sybase）。

8.17 Victor M. Markowitz: “Safe Referential Integrity Structures in Relational Databases,” Proc. 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain (September 1991).

该文提出了两个完全条件，以保证[8.10]和[8.15]讨论的问题不会出现。此文也考虑了在DB2、Sybase和Ingres中实现这些条件时所涉及的问题（也是在1990年左右）。对于DB2，该文表明了其中一些由于安全原因而实施的执行限制在逻辑上是不必要的，而其它一些限制是不合适的（如DB2仍允许一些不安全的状态）。对于Sybase和Ingres，此文说它们提供的过程无法检测出不安全的，或不正确的参照定义。

8.18 Ronald G. Ross: *The Business Rule Book: Classifying, Delining, and Modeling Rules* (Version 3.0). Boston, Mass.: Database Research Group (1994).

见[8.19]的注释。

8.19 Ronald G. Ross: *Business Rule Concepts.* Houston, Tx.: Business Rule Solutions Inc. (1998).

在近年的商业化产品中，都提供了大量的商业规则支持；一些业界人士开始提议将这些规则作为设计和建立数据库和应用程序的更好的基础（更好，是相对于已确立的方法，如“实体/完整性模型”，“对象模型”，“语义模型”等）。我们一致认为，商业规则实际上是对用户更友好的关于谓词、命题和完整性的其他方面的说明方法。Ross是最早提出商业规则这一方法的，对于热心于此的人，他的书值得一读。[8.18]读起来有些费力，[8.19]是一个简明教程。

8.20 M. R. Stonebraker and E. Wong: “Access Control in a Relational Data Base Management System by Query Modification,” Proc. ACM National Conf. (1974).

在University Ingres prototype[7.11]一书中最先提出了一种关于完整性约束的有趣的实施方法（也用于安全性约束——参见第6章），此方法基于请求修改（request modification）。完整性约束通过DEFINE INTEGRITY语句定义。

```
DEFINE INTEGRITY ON <relvar name> IS <boolean expression>
```

例如：

```
DEFINE INTEGRITY ON S IS S.STATUS > 0
```

设用户*U*试图进行下列操作：

```
REPLACE S ( STATUS = S.STATUS - 10 )
WHERE   S.CITY = "London"
```

然后Ingres自动修改REPLACE语句为

```
REPLACE S ( STATUS = S.STATUS - 10 )
WHERE   S.CITY = "London"
AND     ( S.STATUS - 10 ) > 0
```

当然，被修改过的操作不会违反完整性约束。

此方法的缺陷是并非所有的约束都可以用这种简单的方法加以调整；实际上，QUEL只支持布尔表达式是一个简单限制条件的约束。然而，即使有限的支持，也不能在实际系统中得到体现。

8.21 A. Walker and S. C. Salveter: "Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates," State University of New York, Stony Brook, N.Y.: Technical Report 81/026 (June 1981).

讲述一个技巧，用来自动修改"事务模板"（也就是事务原代码）为相应的安全模板（safe template）——安全，是指经修改后的模板不会违反已声明的完整性约束，它是通过向原模板中加入查询和测试来实现的。在运行时，如果测试失败，则事务被拒绝，并产生一个错误消息。

8.22 Jennifer Widom and Stefano Ceri (eds.): *Active Database Systems: Triggers and Rules for Advanced Database Processing.* San Francisco, Calif.: Morgan Kaufmann (1996).

关于"主动数据库"（也就是能对特定事件进行指定行为的数据库系统——换言之，支持存储过程的数据库系统）的有用的研究纲要和自学教程，其中包括了几个原型系统，尤其是有 IBM Research 的 Starburst（参见 [17.50], [25.14], [25.17], [25.2]~[25.22]）。该书也总结了 SQL/92、SQL3（早期版本）和一些商业产品（Oracle、Informix、Ingres等）的相关内容，还包括一个有用的参考书目录。

部分练习答案

8.1

a.
```
TYPE CITY POSSREP ( CHAR )
CONSTRAINT THE_CITY ( CITY ) = 'London'
        OR THE_CITY ( CITY ) = 'Paris'
        OR THE_CITY ( CITY ) = 'Rome'
        OR THE_CITY ( CITY ) = 'Athens'
        OR THE_CITY ( CITY ) = 'Oslo'
        OR THE_CITY ( CITY ) = 'Stockholm'
        OR THE_CITY ( CITY ) = 'Madrid'
        OR THE_CITY ( CITY ) = 'Amsterdam' ;
```

可简写为

a.
```
TYPE CITY POSSREP ( CHAR )
CONSTRAINT THE_CITY ( CITY ) IN { 'London' , 'Paris'     ,
                                  'Rome'   , 'Athens'    ,
                                  'Oslo'   , 'Stockholm' ,
                                  'Madrid' , 'Amsterdam' } ;
```

b.
```
TYPE S# POSSREP ( CHAR ) CONSTRAINT
     SUBSTR ( THE_S# ( S# ), 1, 1 ) = 'S' AND
     CAST_AS_INTEGER ( SUBSTR ( THE_S# ( S# ), 2 ) ≥ 0 AND
     CAST_AS_INTEGER ( SUBSTR ( THE_S# ( S# ), 2 ) ≤ 9999 ;
```

这里有子字符串操作符 SUBSTR和一个直观的转换操作 CAST_AS_INTEGER

c.
```
CONSTRAINT C IS_EMPTY ( P WHERE WEIGHT ≥ WEIGHT ( 50.0 ) ) ;
```

d.
```
CONSTRAINT D COUNT ( J ) = COUNT ( J { CITY } ) ;
```

e.
```
CONSTRAINT E COUNT ( S WHERE CITY = 'Athens' ) ≤ 1 ;
```

f.
```
CONSTRAINT F
    IS_EMPTY ( ( EXTEND SPJ ADD 2 * AVG ( SPJ, QTY )
```

```
                                    AS X ) WHERE QTY > X ) ;
```

g.
```
CONSTRAINT G
    IS_EMPTY ( ( S WHERE STATUS = MIN ( S { STATUS } ) ) JOIN
               ( S WHERE STATUS = MAX ( S { STATUS } ) ) ) ;
```

实际上，这里的术语“最高状态供应商 (highest status supplier)”和“最低状态供应商(lowest status supplier)”，并不是很好的定义，因为状态值不是唯一的。我们将这一要求解释为，如果 *Sx*和*Sy*分别是具有最高和最低状态值的供应商，则他们不能在一个城市中。注意，当最高值与最低值相等时，显然违反约束 —— 特别是，当只有一名供应商时将违反约束。

h.
```
CONSTRAINT H IS_EMPTY ( J { CITY } MINUS S { CITY } ) ;
```

i.
```
CONSTRAINT I IS_EMPTY ( J WHERE NOT ( TUPLE { CITY CITY } IN
                        ( J { J# } JOIN SPJ JOIN S ) { CITY } ) ) ;
```

j.
```
CONSTRAINT J NOT ( IS_EMPTY
                   ( P WHERE COLOR = COLOR ( 'Red' ) ) ) ;
```

如果没有任何零件时显然将违反约束，更好的表示方法为：

```
CONSTRAINT J IS_EMPTY ( P ) OR NOT ( IS_EMPTY
                   ( P WHERE COLOR = COLOR ( 'Red' ) ) ) ;
```

k.
```
CONSTRAINT K IF NOT ( IS_EMPTY ( S ) )
                THEN AVG ( S, STATUS ) > 18
              END IF ;
```

没有供应商元组时，系统也会试图检查约束， IF测试是为了避免这类错误。

l.
```
CONSTRAINT L IS_EMPTY
              ( ( S WHERE CITY = 'London' ) { S# } MINUS
                ( SPJ WHERE P# = P# ( 'P2' ) ) { S# } ) ;
```

m.
```
CONSTRAINT M IS_EMPTY ( P ) OR NOT
            ( IS_EMPTY ( P WHERE WEIGHT < WEIGHT ( 50.0 ) ) ) ;
```

n.
```
CONSTRAINT N
COUNT ( ( ( S WHERE CITY = 'London') JOIN SPJ ) { P# } ) >
COUNT ( ( ( S WHERE CITY = 'Paris' ) JOIN SPJ ) { P# } ) ;
```

o.
```
CONSTRAINT O
    SUM ( ( ( S WHERE CITY = 'London') JOIN SPJ ), QTY ) >
    SUM ( ( ( S WHERE CITY = 'Paris' ) JOIN SPJ ), QTY ) ;
```

p.
```
CONSTRAINT P IS_EMPTY
            ( ( SPJ JOIN ( SPJ' RENAME QTY AS QTY' ) )
              WHERE QTY > 0.5 * QTY' ) ;
```

q.
```
CONSTRAINT Q IS_EMPTY (
  ( S JOIN ( S' WHERE CITY = 'Athens' ) ) WHERE
    CITY ≠ 'Athens' AND CITY ≠ 'London' AND CITY ='Paris' )
         AND IS_EMPTY (
  ( S JOIN ( S' WHERE CITY = 'London' ) ) WHERE
    CITY ≠ 'London' AND CITY ='Paris' ) ;
```

作为辅助练习，可以不用代数形式，而用演算形式定义上述约束。

8.2 前两个是类型约束、 C，D，E，F，G，J，K，M，P，Q是关系变量约束，其余的是数据库约束。

8.3
a. CITY selector invocation.
b. S# selector invocation.
c. INSERT on P, UPDATE on part WEIGHT.
d. INSERT on J, UPDATE on project CITY.

e. INSERT on S, UPDATE on supplier CITY.
f. INSERT or DELETE on SPJ, UPDATE on shipment QTY.
g. INSERT or DELETE on S, UPDATE on supplier STATUS.
h. INSERT on J, DELETE on S, UPDATE on supplier or project CITY.
i. INSERT on J, DELETE on or SPJ, UPDATE on supplier or project
j. INSERT or DELETE on P, UPDATE on part CITY.
k. INSERT or DELETE on S, UPDATE on supplier STATUS.
l. INSERT on S, DELETE on SPJ, UPDATE on supplier CITY or shipment S# or P#.
m. INSERT or DELETE on P, UPDATE on part WEIGHT.
n. INSERT or DELETE on S or SPJ, UPDATE on supplier CITY or shipment S# or P#.
o. INSERT or DELETE on S or SPJ, UPDATE on supplier CITY or shipment S# or P# or QTY.
p. UPDATE on shipment QTY.
q. UPDATE on supplier CITY.

8.4 1种。作为完整性约束，“（5）”和“（3）”的定义形式是最好的。这种方式所期望的结果是：如果X和Y被定义为CHAR（5）和CHAR（3），则它们之间的比较是合法的，也就是说，它们并不与“被比较的两个变量是同一类型”的要求相背。

8.5 以下提供了一组答案。

a. 任何对*A*的限制都是基于*A*的候选码的。
b. 如果投影包括*A*的任意一个候选码*K*，则*K*是投影的一个候选码。否则候选码是投影的所有属性的组合（通常是这样）。
c. *A*的候选码*KA*和*B*的候选码*KB*的组合是*A*乘*B*的积的候选码。
d. 所有属性的集合是唯一的候选码（通常是这样）。
e. 留作练习（交集不是最基本的）。
f. *A*中的每个候选码都是A MINUS B的差的候选码。
g. 一般情况留作练习（自然连接不是基本的）。特别情况下，如果*A*的连接属性是*A*的候选码，则*B*中的候选码就是连接后的候选码。
h. *A*的任意扩充后的候选码与*A*的候选码相同。
i. *A* per *B*任意合计后的候选码就是*B*的候选码。
j. *A*的每一个候选码都是A SEMIJOIN B的一个候选码。
k. *A*的每一个候选码都是A SEMIMINUS B的一个候选码。

上述的内容在某些情况下是可以修改的，如：

- {S#，P#，J#}不是“限制SPJ WHERE S#=S#{ 'S1' }”的候选码，但{P#，J#}是。
- 如果*A*包含{*X*，*Y*，*Z*}和单一的候选码*X*，并满足函数依赖*Y*→*Z*（参见第10章），则*Y*是*A*在*Y*和*Z*上的投影的候选码。
- 如果*A*和*B*都是*C*的限制，则*C*的候选码都是*A* UNION *B*的候选码。等等。候选码推理的全部内容在[10.6]中有详细讨论。

8.6 设*m*是大于或等于*n*/2的最小整数，如果*m*个属性的每一个不同的集合是候选码，或*n*是奇数而且*m*-1个属性的每一个不同的集合是候选码，则*R*可能有最大个数的候选码。无论哪种情况，最大数都为*n*!/(*m*!*(*n*-*m*)!)。注意，关系变量ELEMENT和MARRIAGE都是有最大数目的候选码的例子。

8.7 *R*只有一个候选码，就是属性空集 {}（有时记作 Φ）。注意：对于空候选码，值得花些笔墨。如果关系变量 *R*仅有的合法值是 DEE和DUM，则它不能有属性，显然它仅有的候选码也不包含属性。并非只有无属性的关系变量才会有这样的候选码。如果 Φ是*R*的候选码，则：

- 它一定是*R*的唯一的候选码，因为任何其它的属性集都是 Φ的超集，这就违反候选码的不可约性（如果主码是必需的，则它是主码）。
- *R*被限制为最多含有一个元组，因为对属性空集，所有元组都有相同的值（即 0 元组）。

语法中允许这样的声明，如：

```
VAR R BASE RELATION { ... }
    PRIMARY KEY { } ;
```

也允许声明不含属性的关系变量——也就是，仅有合法值为 DEE和DUM的关系变量。

```
VAR R BASE RELATION { }
    PRIMARY KEY { } ;
```

回到空候选码这个问题：如果候选码可以为空，则相应的外码也可以为空。参考文献[5.5]详细讨论了这一问题。

8.8 不存在显式的外码“ INSERT规则 ”，因为在参照关系变量上了插入（或更新）受到参照完整性基本规则的限制，也就是要有无匹配的外码值的要求。以供应商-零件为例：

- 当试图插入发货元组时，只有当此元组的供应商编号在 S中存在，而且零件编号在P中存在时，请求才会成功。
- 当试图更新发货元组时，只有当更新后元组的供应商编号在 S中存在，而且零件编号在P中存在时，请求才会成功。

注意：前面的叙述适用于参照关系变量，而（显式的） DELETE、UPDATE规则适用于被参照关系变量。因此，当谈论插入规则时，就像它和删除规则和更新规则一样，这是令人感到迷惑的。实际上，在具体语法中不提供显式的插入规则的支持还有其它理由。

8.9 a. 接受
b. 拒绝（违反候选码唯一性）
c. 拒绝（违反限制说明）
d. 接受（供应商 S3和它的所有发货被删除）
e. 拒绝（违反限制说明）
f. 接受（工程 J4和它的所有发货被删除）
g. 接受
h. 拒绝（违反候选码唯一性）
i. 拒绝（违反参照完整性）
j. 接受
k. 拒绝（违反参照完整性）
l. 拒绝（违反参照完整性 —— 关系变量 J中缺省工程号 *jjj*不存在）

8.10 参照图见图 8-1，下面是数据库的定义。为简便起见，我们不定义任何类型约束，除

非在此类型定义上的 POSSREP充当priori约束。

```
TYPE COURSE# POSSREP ( CHAR ) ;
TYPE TITLE   POSSREP ( CHAR ) ;
TYPE OFF#    POSSREP ( CHAR ) ;
TYPE OFFDATE POSSREP ( DATE ) ;
TYPE CITY    POSSREP ( CHAR ) ;
TYPE EMP#    POSSREP ( CHAR ) ;
TYPE NAME    POSSREP ( NAME ) ;
TYPE JOB     POSSREP ( CHAR ) ;
TYPE GRADE   POSSREP ( CHAR ) ;

VAR COURSE BASE RELATION
     { COURSE# COURSE#,
       TITLE   TITLE }
     PRIMARY KEY { COURSE# } ;

VAR PREREQ BASE RELATION
     { SUP_COURSE# COURSE#,
       SUB_COURSE# COURSE# }
     PRIMARY KEY { SUP_COURSE#, SUB_COURSE# }
     FOREIGN KEY { RENAME SUP_COURSE# AS COURSE# }
                                      REFERENCES COURSE
                                      ON DELETE CASCADE
                                      ON UPDATE CASCADE
     FOREIGN KEY { RENAME SUB_COURSE# AS COURSE# }
                                      REFERENCES COURSE
                                      ON DELETE CASCADE
                                      ON UPDATE CASCADE ;

VAR OFFERING BASE RELATION
     { COURSE#  COURSE#,
       OFF#     OFF#,
       OFFDATE  OFFDATE,
       LOCATION CITY }
     PRIMARY KEY { COURSE#, OFF# }
     FOREIGN KEY { COURSE# } REFERENCES COURSE
                             ON DELETE CASCADE
                             ON UPDATE CASCADE ;

VAR EMPLOYEE BASE RELATION
     { EMP#  EMP#,
       ENAME NAME,
       JOB   JOB }
     PRIMARY KEY { EMP# } ;
VAR TEACHER BASE RELATION
     { COURSE# COURSE#,
       OFF#    OFF#,
       EMP#    EMP# }
     PRIMARY KEY { COURSE#, OFF#, EMP# }
     FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
                                   ON DELETE CASCADE
                                   ON UPDATE CASCADE
     FOREIGN KEY { EMP# } REFERENCES EMPLOYEE
                         ON DELETE CASCADE
                         ON UPDATE CASCADE ;

VAR ENROLLMENT BASE RELATION ENROLLMENT
     { COURSE# COURSE#,
       OFF#    OFF#,
       EMP#    EMP#,
       GRADE   GRADE }
     PRIMARY KEY { COURSE#, OFF#, EMP# }
     FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
                                   ON DELETE CASCADE
                                   ON UPDATE CASCADE
     FOREIGN KEY { EMP# } REFERENCES EMPLOYEE
                         ON DELETE CASCADE
                         ON UPDATE CASCADE ;
```

几个要点：

1) 在TEACHER中单个的属性集{COURSE#}和在ENROLLMENT中属性集{COURSE#}都可被看作外码，它们都参照 COURSE。但是，如果从 TEACHER到

OFFERING和从 ENROLLMENT到OFFERING的参照约束能被正确地维护，则从TEACHER到COURSE，和从 ENROLLMENT到COURSE的参照约束会被自动维护。详见[8.10]。

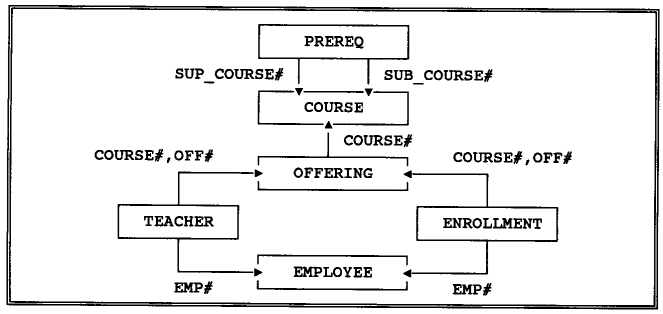
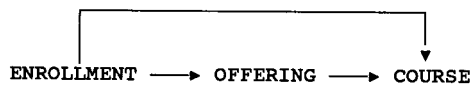


图8-1 教育数据库中的参照图

2) OFFERING是一个既是参照关系变量、又是被参照关系变量的例子。其中有从ENROLLMENT到OFFERING的参照约束（也是从 TEACHER开始的）和从OFFERING到COURSE的参照约束。

ENROLLMENT ⟶ OFFERING ⟶ COURSE

3) 注意：从 ENROLLMENT到COURSE有两条不同的参照路径：一条是直接的（在ENROLLMENT中的外码{COURSE#}），另一条通过OFFERING（在ENROLLMENT中的外码 {COURSE#，OFF #}，以及在 OFFERING中的外码{COURSE#}）。



但这两条路径并非完全独立（下面两个参照的组合蕴含着上面的那个）。详细论述参见[8.10]。

4) 也有从PREREQ到COURSE的两条不同的参照路径，但这两条路径是独立的。

8.11 参照图见图 8-2。注意，在数据库中含有参照环（有自身到自身的参照路径）。除此之外，数据库定义是很直观的。我们省略细节。

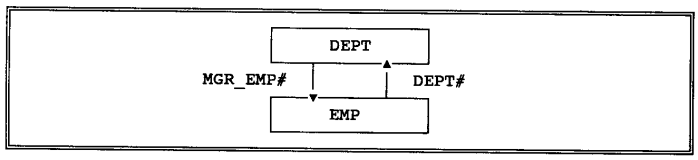


图8-2 包含一个循环的参照图

8.12 这里只写出关系变量定义（要点）：

```
VAR EMP BASE RELATION
      { EMP# ... ,
        ..... ,
        JOB  ... }
      PRIMARY KEY { EMP# } ;

VAR PGMR BASE RELATION
      { EMP# ... ,
        ..... ,
```

```
    LANG ... }
PRIMARY KEY { EMP# }
FOREIGN KEY { EMP# } REFERENCES EMP
        ON DELETE CASCADE
        ON UPDATE CASCADE ;
```

几个要点：

1) **此例表明这样一个事实：外码也可以是候选码。关系变量** EMP**包含所有的雇员，关系变量** PGMR**包含是程序员的雇员；因此，在** PGMR**中出现的雇员号也一定会在**EMP**中出现（反之却不然）。**PGMR**中的主码也是外码，参照** RMP**中的主码。**

2) **注意，在这个例子中还有另一个要维护的约束，这就是：当且仅当雇员的** JOB**值是**programmer (**程序员**)**时，此雇员要在** PGMR**中出现。这显然不是参照约束。**

8.14 **注意解法** a**和**b**。下面只是练习** 8.1**的近似的解法。**

a.
```
CREATE DOMAIN CITY CHAR(15) VARYING
   CONSTRAINT VALID_CITIES
       CHECK ( VALUE IN ( 'London', 'Paris', 'Rome',
                          'Athens', 'Oslo', 'Stockholm',
                          'Madrid', 'Amsterdam' ) ) ;
```

b.
```
CREATE DOMAIN S# CHAR(5) VARYING
CONSTRAINT VALID_S# CHECK
       ( SUBSTRING ( VALUE FROM 1 FOR 1 ) = 'S' AND
  CAST ( SUBSTRING ( VALUE FROM 2 ) AS INTEGER ) >= 0 AND
  CAST ( SUBSTRING ( VALUE FROM 2 ) AS INTEGER ) <= 9999 ) ;
```

c.
```
CREATE ASSERTION SQL_C CHECK
          ( P.COLOR <> 'Red' OR P.WEIGHT < 50.0 ) ;
```

d.
```
CREATE ASSERTION SQL_D CHECK
         ( NOT EXISTS ( SELECT * FROM J JX WHERE
                EXISTS ( SELECT * FROM J JY WHERE
                       ( JX.J# <> JY.J# AND
                         JX.CITY = JY.CITY ) ) ) ) ;
```

e.
```
CREATE ASSERTION SQL_E CHECK
          ( ( SELECT COUNT(*) FROM S
              WHERE  S.CITY = 'Athens' ) < 2 ) ;
```

f.
```
CREATE ASSERTION SQL_F CHECK
          ( NOT EXISTS ( SELECT *
                         FROM   SPJ SPJX
                         WHERE  SPJX.QTY > 2 *
                              ( SELECT AVG ( SPJY.QTY )
                                FROM   SPJ SPJY ) ) ) ;
```

g.
```
CREATE ASSERTION SQL_G CHECK
          ( NOT EXISTS ( SELECT * FROM S SX WHERE
                EXISTS ( SELECT * FROM S SY WHERE
                         SX.STATUS = ( SELECT MAX ( S.STATUS )
                                       FROM   S ) AND
                         SY.STATUS = ( SELECT MIN ( S.STATUS )
                                       FROM   S ) AND
                         SX.STATUS <> SY.STATUS AND
                         SX.CITY = SY.CITY ) ) ) ;
```

h.
```
CREATE ASSERTION SQL_H CHECK
          ( NOT EXISTS ( SELECT * FROM J WHERE
            NOT EXISTS ( SELECT * FROM S WHERE
                         S.CITY = J.CITY ) ) ) ;
```

i.
```
CREATE ASSERTION SQL_I CHECK
          ( NOT EXISTS ( SELECT * FROM J WHERE
            NOT EXISTS ( SELECT * FROM S WHERE
```

```
                    S.CITY = J.CITY AND
                    EXISTS ( SELECT * FROM SPJ
                             WHERE  SPJ.S# = S.S#
                             AND    SPJ.J# = J.J# ) ) ) )
```

j.
```
CREATE ASSERTION SQL_J CHECK
        ( NOT EXISTS ( SELECT * FROM P )
          OR EXISTS ( SELECT * FROM P
                      WHERE  P.COLOR = 'Red' ) ) ;
```

k.
```
CREATE ASSERTION SQL_K CHECK
        ( ( SELECT AVG ( S.STATUS ) FROM S ) > 10 ) ;
```

如果供应商表为空，则 SQL的AVG操作返回空（非正常），条件表达式判断为“未知”，约束被认为没有违反。详细内容请见第 18章。

l.
```
CREATE ASSERTION SQL_L CHECK
        ( NOT EXISTS ( SELECT * FROM S
                       WHERE  S.CITY = 'London'
                       AND    NOT EXISTS
                            ( SELECT * FROM SPJ
                              WHERE  SPJ.S# = S.S#
                              AND    SPJ.P# = 'P2' ) ) ) ;
```

m.
```
CREATE ASSERTION SQL_M CHECK
        ( NOT EXISTS ( SELECT * FROM P
                       WHERE  P.COLOR = 'Red' )
          OR EXISTS ( SELECT * FROM P
                      WHERE  P.COLOR = 'Red'
                      AND    P.WEIGHT < 50.0 ) ) ;
```

n.
```
CREATE ASSERTION SQL_N CHECK
        ( ( SELECT COUNT(*) FROM P
            WHERE  EXISTS ( SELECT * FROM SPJ WHERE
                   EXISTS ( SELECT * FROM S WHERE
                          ( P.P# = SPJ.P# AND
                            SPJ.S# = S.S# AND
                            S.CITY = 'London' ) ) ) ) >
          ( SELECT COUNT(*) FROM P
            WHERE  EXISTS ( SELECT * FROM SPJ WHERE
                   EXISTS ( SELECT * FROM S WHERE
                          ( P.P# = SPJ.P# AND
                            SPJ.S# = S.S# AND
                            S.CITY = 'Paris' ) ) ) ) ) ;
```

o.
```
CREATE ASSERTION SQL_O CHECK
        ( ( SELECT SUM ( SPJ.QTY ) FROM SPJ
            WHERE  ( SELECT S.CITY FROM S
                     WHERE  S.S# = SPJ.S# ) = 'London' ) >
          ( SELECT SUM ( SPJ.QTY ) FROM SPJ
            WHERE  ( SELECT S.CITY FROM S
                     WHERE  S.S# = SPJ.S# ) = 'Paris' ) ) ;
```

p. 无法实现。

q. 无法实现。