第6章　数据库触发器

命名PL/SQL块的第四种类型是触发器。触发器类在某些方面类似于子程序，但它们之间也有明显地区别。我们在本章将介绍如何创建不同类型的触发器以及讨论触发器的一些应用。

6.1　触发器的类型

触发器类似于函数和过程，它们都是具有声明部分、执行部分和异常处理部分的命名PL/SQL块。像包一样，触发器必须在数据库中以独立对象的身份存储，并且不能与包和块具有本地关系。我们在前两章中已经讲过，过程是显式地通过过程调用从其他块中执行的，同时，过程调用可以传递参数。与之相反，触发器是在事件发生时隐式地运行的，并且触发器不能接收参数。运行触发器的方式叫做激发（firing）触发器，触发事件可以是对数据库表的DML（INSERT、UPDATE或DELETE）操作或某种视图的操作(View)。Oracle8i把触发器功能扩展到了可以激发系统事件，如数据库的启动和关闭，以及某种DDL操作。我们将在本章的后几节讨论触发事件的详细内容。

触发器可以用于下列情况：

- 维护在表创建阶段通过声明限制无法实现的复杂完整性限制。
- 通过记录修改内容和修改者来审计表中的信息。
- 在表内容发生变更时，自动通知其他程序采取相应的处理。
- 在订阅发布环境下，发布有关各种事件的信息。

有三种主要的触发器类型：DML、替代触发器和系统触发器。在下面几节中，我们将逐一介绍这些触发器类型。在本章后面“创建触发器”一节中还将详细讨论这些触发器。

注意　Oracle8i允许使用PL/SQL语言或可以作为外部例程调用的其他语言来编制触发器。有关触发器的详细介绍，请参阅6.2.4节和第10章的内容。

6.1.1　DML触发器

DML触发器可以由DML语句激发，并且由该语句的类型决定DML触发器的类型。可以定义DML触发器进行INSERT，UPDATE，DELETE操作。这类触发器可以在上述操作之前或之后激发，除此之外，它们也可以在行或语句操作上激发。

作为例子，让我们假设要跟踪不同专业的统计信息，其中包括已注册学生的数量和已得到的总分。我们要把这些结果存储在表major_stats中：

节选自在线代码relTables.sql

```
CREATE TABLE major_stats (
  major         VARCHAR2(30),
  total_credits NUMBER,
```

```
  total_students NUMBER);
```

为了保持表major_stats中的数据处于更新状态，我们可以创建一个每次表 students被修改时自动更新表major_stats的触发器。下面所示的 UpdateMajorStats就是实现上述功能的触发器。在表students上进行任何DML操作之后，该触发器将启动运行。该触发器的代码要查询表 students并使用当前的统计信息更新表 major_stats。

节选自在线代码UpdateMajorStats.sql

```
CREATE OR REPLACE TRIGGER UpdateMajorStats
  /* Keeps the major_stats table up-to-date with changes made
     to the students table. */
  AFTER INSERT OR DELETE OR UPDATE ON students
DECLARE
  CURSOR c_Statistics IS
    SELECT major, COUNT(*) total_students,
           SUM(current_credits) total_credits
      FROM students
      GROUP BY major;
BEGIN
  /* First delete from major_stats. This will clear the
     statistics, and is necessary to account for the deletion
     of all students in a given major. */
  DELETE FROM major_stats;
  /* Now loop through each major, and insert the appropriate row into
     major_stats. */
  FOR v_StatsRecord in c_Statistics LOOP
    INSERT INTO major_stats (major, total_credits, total_students)
      VALUES (v_StatsRecord.major, v_StatsRecord.total_credits,
              v_StatsRecord.total_students);
  END LOOP;
END UpdateMajorStats;
```

语句触发器可以激发多种类型的触发语句。例如， UpdateMajorStats可以激发INSERT，UPDATE，DELETE语句。触发事件说明了一个或多个激发触发器的 DML操作。

6.1.2 替代触发器

Oracle 8及更高版本

Oracle8提供的这种替代触发器（Instead-of trigger）只能定义在视图上（可以是关系或对象）。与DML触发器不同，DML触发器是在DML操作之外运行的，而替代触发器则代替激发它的 DML语句运行。替代触发器是行一级的。例如，请看下面的视图 classes_rooms:

节选自在线代码insteadOf.sql

```
CREATE OR REPLACE VIEW classes_rooms AS
  SELECT department, course, building, room_number
  FROM rooms, classes
  WHERE rooms.room_id = classes.room_id;
```

如下所示，直接执行对该视图的插入操作是非法的。这是因为该视图是两个表的联合，而插入操作要求对两个现行表进行修改，下面的SQL *Plus会话显示了插入操作过程：

节选自在线代码insteadOf.sql

```
SQL> INSERT INTO classes_rooms (department, course, building,
                                room_number)
  2    VALUES ('MUS', 100, 'Music Building', 200);
INSERT INTO classes_rooms (department, course, building, room_number)
                                                *
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

然而，我们可以创建一个替代触发器来实现正确的插入操作，也就是来更新现行表：

节选自在线代码insteadOf.sql

```
CREATE TRIGGER ClassesRoomsInsert
  INSTEAD OF INSERT ON classes_rooms
DECLARE
  v_roomID rooms.room_id%TYPE;
BEGIN
  -- First determine the room ID
  SELECT room_id
    INTO v_roomID
    FROM rooms
    WHERE building = :new.building
    AND room_number = :new.room_number;
  -- And now update the class
  UPDATE CLASSES
    SET room_id = v_roomID
    WHERE department = :new.department
    AND course = :new.course;
END ClassesRoomsInsert;
```

有了触发器ClassesRoomsInsert，INSERT语句就可以执行正确的更新操作。

注意 在上面的程序中，触发器ClassesRoomsInsert没有做任何错误检查。我们将在本章后的程序中加入错误检查和处理代码。

6.1.3 系统触发器

Oracle 8i及更高版本

Oracle8i提供了第三种触发器，这种系统触发器在发生如数据库启动或关闭等系统事件时激发，而不是在执行DML语句时激发。系统触发器也可以在DDL操作时，如表的创建中激发。例如，假设我们要记录对象创建的时间，我们可以通过创建下面的表来实现上述记录功能：

节选自在线代码LogCreations.sql

```
CREATE TABLE   ddl_creations (
  user_id            VARCHAR2(30),
  object_type        VARCHAR2(20),
```

```
  object_name         VARCHAR2(30),
  object_owner        VARCHAR2(30),
  creation_date       DATE);
```

一旦该表可以使用，我们就可以创建一个系统触发器来记录相关信息。在每次 CREATE语句对当前模式进行操作之后，触发器 LogCreations就记录在ddl_creations中创建的对象的有关信息。

节选自在线代码LogCreations .sql

```
CREATE OR REPLACE TRIGGER LogCreations
  AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ddl_creations (user_id, object_type, object_name,
                             object_owner, creation_date)
    VALUES (USER, SYS.DICTIONARY_OBJ_TYPE, SYS.DICTIONARY_OBJ_NAME,
            SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;
```

6.2 创建触发器

所有触发器，不管其类型如何，都可以使用相同的语法创建。下面是创建触发器的通用语法：

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF} triggering_event
  referencing_clause
  [WHEN trigger_condition]
  [FOR EACH ROW]
  trigger_body;
```

其中，trigger_name是触发器的名称，triggering_event说明了激发触发器的事件（也可能包括特殊的表或视图），trigger_body是触发器的代码。referencing_clause用来引用正在处于修改状态下的行中的数据，如果在 WHEN子句中指定trigger_condition的话，则首先对该条件求值。触发器主体只有在该条件为真值时才运行。我们在下面几节中将演示不同类型的触发器案例。

注意 触发器主体不能超过32K。如果触发器长度超过了该限制，就要把该体内的某些代码放到单独编译的包或存储子程序中，并从触发器主体中调用这些代码。

6.2.1 创建DML触发器

DML触发器是由对数据库表进行 INSERT、UPDATE、DELETE操作而激发的触发器。该类触发器可以在上述操作之前或之后激发运行，也可以按每个变更行激发一次，或每个语句激发一次进行。这些条件的组合形成了触发器的类型。总共有 12种可能的触发类型：3种语句×2种定时×2级。例如，下面所有的说明都是合法的 DML触发器类型：

- 更新语句之前
- 插入行之后
- 删除行之前

表6-1总结了DML触发器的各种选择项。除此之外，触发器也可以由给定表中的一个以上的DML语句，如INSERT和UPDAE而激发。触发器中的任何代码将随触发语句一起作为同一事务的一部分运行。

可以对一个表定义任意数量的触发器，其中可以包括一个以上的给定 DML类型。例如，可以定义两个删除语句之后的触发器。所有同类型的触发器将按顺序激发。（下一节讨论触发器的激发顺序。）

注意 在PL/SQL2.1版（Oracle7的7.1版）之前的版本下，每种类型的触发器只能在表上定义一个。也就是说，最多有 12个触发器。因此，初始化参数 COMPATIBLE就必须是7.1或更高以便复制一个表上的同类触发器。

DML触发器的触发事件说明了激发触发器的表的名称（以及列）。在Oracle8i下，触发器可以在嵌套表的列上激发。本书的第 14章提供了更多的触发器内容。

表6-1 DML触发器类型

类 别	值	说 明
语句	INSERT、DELETE、UPDATE	定义何种DML语句激发触发器
定时	之前或之后	定义触发器是在语句运行前或运行后激发
级	行或语句	如果触发器是行级触发器，该触发器就对由触发语句变更的每一行激发一次。如果触发器是语句级的触发器，则该触发器就在语句之前或之后激发一次。行级触发器是按触发器定义中的FOR EACH ROW子句表示的

1. DML 触发器激发顺序

触发器是在DML语句运行时激发的。下面是执行 DML语句的算法步骤：

1) 如果有语句之前级触发器的话，先运行该触发器。

2) 对于受语句影响每一行：

a. 如果有行之前级触发器的话，运行该触发器。

b. 执行该语句本身。

c. 如果有行之后级触发器的话，运行该触发器。

3) 如果有语句之后级触发器的话，运行该触发器。

为了说明上面的算法，假设我们在表 classes上创建了所有四种 UPDATE触发器，即之前，之后，语句和行级。我们将创建三个行前触发器和两个语句后触发器，其代码如下：

节选自在线代码firingOrder .sql

```
CREATE SEQUENCE trig_seq
  START WITH 1
  INCREMENT BY 1;

CREATE OR REPLACE PACKAGE TrigPackage AS
  -- Global counter for use in the triggers
  v_Counter NUMBER;
END TrigPackage;
```

```
CREATE OR REPLACE TRIGGER ClassesBStatement
  BEFORE UPDATE ON classes
BEGIN
  -- Reset the counter first.
  TrigPackage.v_Counter := 0;
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Statement: counter = ' || TrigPackage.v_Counter);

  -- And now increment it for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBStatement;

CREATE OR REPLACE TRIGGER ClassesAStatement1
  AFTER UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
  VALUES (trig_seq.NEXTVAL,
    'After Statement 1: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesAStatement1;

CREATE OR REPLACE TRIGGER ClassesAStatement2
  AFTER UPDATE ON classes
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'After Statement 2: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesAStatement2;

CREATE OR REPLACE TRIGGER ClassesBRow1
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Row 1: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow1;

CREATE OR REPLACE TRIGGER ClassesBRow2
```

```
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Row 2: counter = ' || TrigPackage.v_Counter);
  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow2;

CREATE OR REPLACE TRIGGER ClassesBRow3
  BEFORE UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Row 3: counter = ' || TrigPackage.v_Counter);
  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesBRow3;

CREATE OR REPLACE TRIGGER ClassesARow
  AFTER UPDATE ON classes
  FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'After Row: counter = ' || TrigPackage.v_Counter);

  -- Increment for the next trigger.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END ClassesARow;
```

假设我们现在提交下列UPDATE**语句：**

```
UPDATE classes
  SET num_credits = 4
  WHERE department IN ('HIS', 'CS');
```

该语句对四行有影响。语句之前级和之后触发器将各自运行一次，而行之前级和之后触发器则每个运行四次。如果在上述操作之后，我们从表 temp_table**进行选择的话，下面就是得到的结果：**

节选自在线代码firingOrder .sql

```
SQL> SELECT * FROM temp_table
  2 ORDER BY num_col;
  NUM_COL CHAR_COL
--------- ----------------------------------
        1 Before Statement: counter = 0
```

```
2 Before Row 3: counter = 1
3 Before Row 2: counter = 2
4 Before Row 1: counter = 3
5 After Row: counter = 4
6 Before Row 3: counter = 5
7 Before Row 2: counter = 6
8 Before Row 1: counter = 7
9 After Row: counter = 8
10 Before Row 3: counter = 9
11 Before Row 2: counter = 10
12 Before Row 1: counter = 11
13 After Row: counter = 12
14 Before Row 3: counter = 13
15 Before Row 2: counter = 14
16 Before Row 1: counter = 15
17 After Row: counter = 16
18 After Statement 2: counter = 17
19 After Statement 1: counter = 18
```

当每个触发器被激发时，该触发器将可以看到由其前期触发器实现的变更，以及到目前为止由DML语句对数据实现的变更。触发器的激发可以通过由每个触发器打印的记数器值来判断。（请看第5章中有关使用包变量的说明。）

同类触发器的激发顺序没有明确的定义。如前面所示的例子中，每个触发器都将可以看到其前面的触发器实施的变更。如果顺序非常重要的话，可以把所有的操作组合在一个触发器中。

注意 当你为表创建快照日志时，Oracle将自动地为该表创建一个AFTER ROW触发器，该触发器在每个DML语句后更新日志文件。如果要创建额外的 AFTER ROW触发器的话，你一定要避免与系统创建的触发器发生冲突。除此之外，数据库系统还对触发器和快照有其他的限制。

2. 行级触发器的相关标识

行级触发器是按触发语句所处理的行激发的。在触发器内部，我们可以访问正在处理中的行的数据。这种访问是通过两个相关的标识符： :old和:new实现的。相关标识符是一种特殊的PL/SQL连接变量（bind variable）,在该标识符前面的冒号说明它们是使用在嵌套 PL/SQL中的宿主变量意义上的连接变量，而不是一般的 PL/SQL变量。 PL/SQL编译器将把这种变量按记录类型处理。

triggering_table%ROWTYPE

其中，triggering_table是定义触发器所在的表。因此，下面的引用：

：new.field

将只有在该字段位于触发表中时才是合法的。表 6-2总结了标识符:old和:new的含义。尽管从语法上将，这两个标识符都按记录类型处理，但实际上不是这样的（该问题将在下文“伪记录”中介绍。）。正是这个原因，标识符:old和:new也被称为伪记录。

表6-2 :old和:new相关标识符

触发语句	标识符:old	标识符:new
INSERT	无定义-所有字段为空NULL	该语句结束时将插入的值
UPDAE	更新前行的原始值	该语句结束时将更新的值
DELETE	行删除前的原始值	无定义-所有字段为空NULL

注意 标识符:old对INSERT语句无定义，而标识符 :new对DELETE语句无定义。PL/SQL编译器不会对在INSERT语句中使用的:old和在DELETE语句中使用的:new标识符报错，编译的结果将使该字段为空。

Oracle 8i及更高版本

Oracle8i定义了另外一个相关标识符 -:parent。如果触发器定义在嵌套表中的话，标识符:old和:new就引用嵌套表中的行，而 :parent则引用其父表的当前行。有关详细信息，请参阅Oracle文档资料。

使用:old和:new相关标识符 下面所示的触发器GenerateStudentID使用了标识符:new。该触发器是一个INSERT之前触发器，其目的是使用从序列 studengt_seuence中生成的值来填写 ID字段。

节选自在线代码GenerateStudentID .sql

```
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  FOR EACH ROW
BEGIN
  /* Fill in the ID field of students with the next value from
     student_sequence. Since ID is a column in students, :new.ID
     is a valid reference. */
SELECT student_sequence.NEXTVAL
    INTO :new.ID
    FROM dual;
END GenerateStudentID;
```

触发器GenerateStudentID实际上是修改:new.ID的值，这就是:new标识符的用途之一，即当该语句实际运行时，在 :new中的任何值都将被使用。使用触发器 GenerateStudentID，我们可以发布如下所示的INSERT语句：

节选自在线代码GenerateStudentID .sql

```
INSERT INTO students (first_name, last_name)
  VALUES ('Lolita', 'Lazarus');
```

该语句可以正确执行。尽管我们没有为主键列 ID指定值（该值是语句需要的），但触发器可以提供它。事实上，即使我们为该 ID指定了一个值，该值也将被忽略不记，因为触发器将改变该值。如果我们执行下面的命令：

节选自在线代码GenerateStudentID .sql

```
INSERT INTO students (ID, first_name, last_name)
  VALUES (-7, 'Zelda', 'Zoom');
```

该ID列将被来自于student_sequence.NEXTVAL的值填充，而不是值-7.

按上面的操作结果，我们不能在行级触发器之后改变 :new，其原因是该语句已被处理了。总的来说，对:new的修改只能在行级触发器之前修改。:old具有只读属性，只能读入。

记录：new和:old只在行级触发器内部合法。如果企图引用在语句级触发器之内的： new或:old的话，编译器将报错。由于语句级的触发器只运行一次，即使存在很多被语句处理过的行的话，new和:old也没有定义。编译器不知道该引用那一行。

伪记录 尽管:new和:old从语法上讲按triggring_table%ROWTYPE的记录来处理，但实际处理却不一样。这样一来，应该是合法的记录操作对于 :new和：old来说就变成了非法操作。例如，这两个记录不能按全记录赋值。而只用在其内部的个别字段可以赋值。下面的程序可以说明上述问题：

节选自在线代码pseudoRecords .sql

```
CREATE OR REPLACE TRIGGER TempDelete
  BEFORE DELETE ON temp_table
  FOR EACH ROW
DECLARE
  v_TempRec temp_table%ROWTYPE;
BEGIN
  /* This is not a legal assignment, since :old is not truly
     a record. */
  v_TempRec := :old;

  /* We can accomplish the same thing, however, by assigning
    the fields individually. */
  v_TempRec.char_col := :old.char_col;
  v_TempRec.num_col := :old.num_col;
END TempDelete;
```

除此之外，:old和：new记录不能传递到接收triggring_table%ROWTYEP的过程或函数中。

引用子句 我们可以根据需要使用子句REFERENCING来为:new和:old指定一个不同的名称。该子句可以在触发事件之后，WHEN子句之前使用，其语法如下：

```
REFERENCING [OLD AS :old_name] [NEW AS :new_name]
```

在触发器体内，我们可以使用 :old_name和:new_name来代替:old和:new。下面是触发器GenerateStudentID的另一种版本，该版本使用REFERECING来把:new作为:new_student引用。

节选自在线代码GenerateStudentID .sql

```
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  REFERENCING new AS new_student
  FOR EACH ROW
BEGIN
  /* Fill in the ID field of students with the next value from
     student_sequence. Since ID is a column in students,
     :new_student.ID is a valid reference. */
  SELECT student_sequence.nextval
    INTO :new_student.ID
    FROM dual;
```

```
END GenerateStudentID;
```

3. WHEN子句

WHEN子句只适用于行级触发器。如果使用该子句的话，触发器体将只对满足由 WHEN子句说明的条件的行执行。WHEN子句的语法是：

```
WHEN trigger_condition
```

其中，trigger_condition是逻辑表达式。该表达式将为每行求值。 :new和：old记录可以在trigger_condition内部引用，但不需使用冒号。该冒号只在触发器体内有效。例如，触发器CheckCredits的体只在当前的学生得到的学分超出20时才运行：

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
  WHEN (new.current_credits > 20)
BEGIN
  /* Trigger body goes here. */
END;
```

触发器CheckCredits也可写为下列代码：

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
BEGIN
  IF :new.current_credits > 20 THEN
     /* Trigger body goes here. */
  END IF;
END;
```

4. 触发器谓语：INSERTING、UPDATING和DELETING

6.1.1节中讨论的触发器 UpdateMajorStats就是INSERT、UPDATE和DELETE触发器。在这种触发器的内部（为不同的 DML语句激发的触发器）有三个可用来确认执行何种操作的逻辑表达式。这些表达式的谓语是 INSERTING、UPDATING、DELETING。下面说明了每个谓词的含义。

表达式谓语	状　态
INSERTING	如果触发语句是INSERT的话，则为真值（TRUE），否则为FALSE
UPDATING	如果触发语句是UPDATE的话，则为真值（TRUE），否则为FALSE
DELETING	如果触发语句是DELETE的话，则为真值（TRUE），否则为FALSE

注意 Oracle8i定义了额外的可以从触发器体内调用的函数，这种函数类似于触发器表达式谓语。有关详细内容，请看6.2.3节中的“事件属性函数”。

触发器LogRSChanges使用上述表达式的谓语来记录表 registered_students发生的所有变化。除了记录这些信息外，它还记录对表进行变更的用户名。该触发器的记录存放在表 RS_audit中，其结构如下：

节选自在线代码relTables .sql

```
CREATE TABLE RS_audit (
```

```
  change_type      CHAR(1)        NOT NULL,
  changed_by       VARCHAR2(8)    NOT NULL,
  timestamp        DATE NOT       NULL,
  old_student_id   NUMBER(5),
  old_department   CHAR(3),
  old_course       NUMBER(3),
  old_grade        CHAR(1),
  new_student_id   NUMBER(5),
  new_department   CHAR(3),
  new_course       NUMBER(3),
  new_grade        CHAR(1)
  );
```

触发器LogRSChanges**的创建语句如下：**

节选自在线代码LogRSChanges.sql

```
CREATE OR REPLACE TRIGGER LogRSChanges
  BEFORE INSERT OR DELETE OR UPDATE ON registered_students
  FOR EACH ROW
DECLARE
  v_ChangeType CHAR(1);
BEGIN
  /* Use 'I' for an INSERT, 'D' for DELETE, and 'U' for UPDATE. */
  IF INSERTING THEN
    v_ChangeType := 'I';
  ELSIF UPDATING THEN
    v_ChangeType := 'U';
  ELSE
    v_ChangeType := 'D';
  END IF;
    /* Record all the changes made to registered_students in
    RS_audit. Use SYSDATE to generate the timestamp, and
    USER to return the userid of the current user. */
  INSERT INTO RS_audit
    (change_type, changed_by, timestamp,
    old_student_id, old_department, old_course, old_grade,
    new_student_id, new_department, new_course, new_grade)
  VALUES
    (v_ChangeType, USER, SYSDATE,
    :old.student_id, :old.department, :old.course, :old.grade,
    :new.student_id, :new.department, :new.course, :new.grade);
END LogRSChanges;
```

触发器常用于进行数据检查，就象触发器 LogRSChanges**的功能那样。然而，检查还只是数据库的一部分用途，触发器还可用于更灵活和更用户化的记录。我们还可以对触发器** LogRSChanges**进行修改，例如，使用它来记录仅由某些人做的修改。我们还可以使用该触发器来检查是否用户有权变更数据并在没有授权的情况下引发异常（使用** RAISE_APPLICATION_ERROR**）。**

6.2.2 创建替代触发器

Oracle 8及更高版本

DML触发器是除去执行INSERT，UPDATE或DELETE操作外（在这些语句之前或之后）还要被激活运行的触发器，而替代触发器则被激发来代替执行 DML语句。除此之外，替代触发器还可以定义在视图上，而 DML触发器只能定义在表上。替代触发器的用途有两类：

- 允许对无法变更的视图进行修改。
- 修改视图中嵌套表列的列。

我们将在本节讨论第一种应用，其他信息请看本书第 14章。

注意 在Oracle8i的8.1.5版中，替代触发器功能只能在其企业版中使用。在将来的版本中，有可能在其他的版本中也提供该功能。

1. 可变更的与不可变更的视图

可变更视图是可以发布 DML命令的视图。一般来说，视图如果不包括下列命令参数的话就是一个可变更视图：

- Set operators(UNION,UNION ALL,MINUS)
- Aggregate functions(SUM,AVG,etc.)
- GROUP BY,CONNECT BY,或START WITH clauses
- 操作数DISTINCT
- 联合

然而，也确实有包括联合的视图是可以变更的。总的来说，如果对该视图的 DML操作每次只变更基表，并且DML操作满足了表6-3的条件，那么，该联合视图也是可变更的。如果一个视图是不可变更的，则我们可以在其上写一个替代触发器来执行正确的操作，从而使该视图可变更。如果需要进行额外处理的话，替代触发器也可以写在可变更视图上。

表6-3引用了保留字表。如果一个表与另一个表联合后，其原始表中的关键字在联合后的表中仍然是关键字的话，该表就是一个关键字保留表（key-preserved）。

表6-3 可变更的联合视图

DML操作	可执行条件
INSERT	该语句没有显式或隐式地引用非保留字表的列
UPDATE	更新的列映射到保留字表的列中
DELETE	在联合中仅有一个保留字表

2. 替代触发器案例

请考虑我们在6.1.2节中介绍过的的视图classes_rooms：

节选自在线代码insteadOf.sql

```
CREATE OR REPLACE VIEW classes_rooms AS
  SELECT department, course, building, room_number
  FROM rooms, classes
  WHERE rooms.room_id = classes.room_id;
```

如上所见，对该视图的 INSERT操作是合法的，尽管可以合法地对该视图执行 UPDATE或

DELETE操作，但这些命令不能实现正确的操作。例如，从 classes_rooms发布的DELETE命令将会删除表classes中对应的行，什么是 classes_rooms的正确DML操作呢？这与程序的逻辑要求有关。假设这些命令有下列含义：

操　作	含　义
INSERT	把新近插入的班级赋予新插入的教室。该操作将导致对表 classes的更新
UPDATE	变更赋予班级的教室。该操作将导致 classes或rooms的更新，取决于表 classes_rooms的哪一列有变更
DELETE	从删除的班级中清除教室的ID。该操作将导致表classes的更新，将ID置为空NULL

下面所示的触发器 ClassesRoomsInstead实施了上述规则并允许对表 classes_romms执行正确的DML操作。

节选自在线代码ClassesRoomInstead.sql

```
CREATE OR REPLACE TRIGGER ClassesRoomsInstead
  INSTEAD OF INSERT OR UPDATE OR DELETE ON classes_rooms
  FOR EACH ROW
DECLARE
  v_roomID rooms.room_id%TYPE;
  v_UpdatingClasses BOOLEAN := FALSE;
  v_UpdatingRooms BOOLEAN := FALSE;

  -- Local function that returns the room ID, given a building
  -- and room number. This function will raise ORA-20000 if the
  -- building and room number are not found.
  FUNCTION GetRoomID(p_Building IN rooms.building%TYPE,
                     p_Room IN rooms.room_number%TYPE)
  RETURN rooms.room_id%TYPE IS

  v_RoomID rooms.room_id%TYPE;
BEGIN
  SELECT room_id
    INTO v_RoomID
    FROM rooms
    WHERE building = p_Building
    AND room_number = p_Room;
RETURN v_RoomID;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20000, 'No matching room');
END getRoomID;

-- Local procedure that checks whether the class identified by
-- p_Department and p_Course exists. If not, it raises
-- ORA-20001.
```

```
PROCEDURE VerifyClass(p_Department IN classes.department%TYPE,
                      p_Course IN classes.course%TYPE) IS
    v_Dummy NUMBER;
BEGIN
    SELECT 0
      INTO v_Dummy
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001,
      p_Department || ' ' || p_Course || ' doesn''t exist');
  END verifyClass;

BEGIN
  IF INSERTING THEN
    -- This essentially assigns a class to a given room. The logic
    -- here is the same as the "updating rooms" case below: First,
    -- determine the room ID:
      v_RoomID := GetRoomID(:new.building, :new.room_number);
      -- And then update classes with the new ID.
      UPDATE CLASSES
        SET room_id = v_RoomID
        WHERE department = :new.department
        AND course = :new.course;

ELSIF UPDATING THEN
  -- Determine if we are updating classes, or updating rooms.
  v_UpdatingClasses := (:new.department != :old.department) OR
                       (:new.course != :old.course);
  v_UpdatingRooms := (:new.building != :old.building) OR
                     (:new.room_number != :old.room_number);
IF (v_UpdatingClasses) THEN
  -- In this case, we are changing the class assigned for a
  -- given room. First make sure the new class exists.
    VerifyClass(:new.department, :new.course);

    -- Get the room ID,
    v_RoomID := GetRoomID(:old.building, :old.room_number);

    -- Then clear the room for the old class,
    UPDATE classes
      SET room_ID = NULL
      WHERE department = :old.department
      AND course = :old.course;
```

```
    -- And finally assign the old room to the new class.
    UPDATE classes
      SET room_ID = v_RoomID
      WHERE department = :new.department
      AND course = :new.course;
  END IF;

  IF v_UpdatingRooms THEN
    -- Here, we are changing the room for a given class. This
    -- logic is the same as the "inserting" case above, except
    -- that classes is updated with :old instead of :new.
    -- First, determine the new room ID.
    v_RoomID := GetRoomID(:new.building, :new.room_number);

    -- And then update classes with the new ID.
    UPDATE CLASSES
      SET room_id = v_RoomID
      WHERE department = :old.department
      AND course = :old.course;
  END IF;

ELSE
   -- Here, we want to clear the class assigned to the room,
  -- without actually removing rows from the underlying tables.
  UPDATE classes
      SET room_ID = NULL
      WHERE department = :old.department
      AND course = :old.course;
  END IF;
END ClassesRoomsInstead;
```

注意 子句FOR EACH ROW是替代触发器的选择项。不管该子句是否存在，所有的替代触发器都是行级的。

触发器ClassesRoomsInstead使用触发器判定来决定将要执行的 DML操作，并且采取相应的动作。图6-1演示了表classes、rooms和classes_rooms的原始内容。假设我们发布了下列 INSERT 命令：

节选自在线代码ClassesRoomInstead.sql

```
INSERT INTO classes_rooms
  VALUES ('MUS', 100, 'Music Building', 200);
```

该触发器对表 classes进行更新以便反映新的教室。新的 classes表如图6-2所示。现在我们假设发布了下列UPDATE命令：

节选自在线代码ClassesRoomInstead.sql

```
UPDATE classes_rooms
```

```
SET department = 'NUT', course = 307
WHERE building = 'Building 7' AND room_number = 201;
```

我们看到表classes又一次被更新，更新后的表反映了新的变更。从该表中我们可以看到，历史系101课程没有分配教室，营养系307课程分配到了原历史系101课程的教室。这种变化反映在图6-3所示的表中。最后，假设我们发布如下的DELETE命令：

节选自在线代码ClassesRoomInstead.sql

```
DELETE FROM classes_rooms
  WHERE building = 'Building 6';
```

classes

Dept	Course	Room ID
HIS	101	20000
CS	101	20001
ECN	203	20002
CS	102	20003
HIS	301	20004
MUS	410	20005
MUS	100	
ECN	101	20007
NUT	307	20008

rooms

Room ID	Building	Room Number
20000	Building 7	201
20001	Building 6	101
20002	Building 6	150
20003	Building 6	160
20004	Building 6	170
20005	Music Building	100
20006	Music Building	200
20007	Building 7	300
20008	Building 7	310

classes_rooms

Dept	Course	Building	Room Number
HIS	101	Building 7	201
CS	101	Building 6	101
ECN	203	Building 6	150
CS	102	Building 6	160
HIS	301	Building 6	170
MUS	410	Music Building	100
ECN	101	Building 7	300
NUT	307	Building 7	310

图6-1 表classesrooms和classes_rooms的原始内容

classes

Dept	Course	Room ID
HIS	101	20000
CS	101	20001
ECN	203	20002
CS	102	20003
HIS	301	20004
MUS	410	20005
MUS	**100**	**20006**
ECN	101	20007
NUT	307	20008

rooms

Room ID	Building	Room Number
20000	Building 7	201
20001	Building 6	101
20002	Building 6	150
20003	Building 6	160
20004	Building 6	170
20005	Music Building	100
20006	Music Building	200
20007	Building 7	300
20008	Building 7	310

classes_rooms

Dept	Course	Building	Room Number
HIS	101	Building 7	201
CS	101	Building 6	101
ECN	203	Building 6	150
CS	102	Building 6	160
HIS	301	Building 6	170
MUS	410	Music Building	100
ECN	101	Building 7	300
NUT	307	Building 7	310
MUS	**100**	**Music Building**	**200**

图6-2 执行插入操作后的各表的内容

对表classes的更新操作把原在6楼教室的room_ID设置为空。更新后的表如图6-4所示。请注

意，经过前面所有的DML语句，表rooms保持没有变更，只有表classses进行了更新。

classes

Dept	Course	Room ID
NUT	307	20000
CS	101	20001
ECN	203	20002
CS	102	20003
HIS	301	20004
MUS	410	20005
MUS	100	20006
ECN	101	20007
HIS	101	

rooms

Room ID	Building	Room Number
20000	Building 7	201
20001	Building 6	101
20002	Building 6	150
20003	Building 6	160
20004	Building 6	170
20005	Music Building	100
20006	Music Building	200
20007	Building 7	300
20008	Building 7	310

classes_rooms

Dept	Course	Building	Room Number
HIS	101	Building 7	201
CS	101	Building 6	101
ECN	203	Building 6	150
CS	102	Building 6	160
HIS	301	Building 6	170
MUS	410	Music Building	100
ECN	101	Building 7	300
NUT	307	Building 7	201
MUS	100	Music Building	200

图6-3 更新后的表的内容

classes

Dept	Course	Room ID
NUT	307	20000
CS	101	
ECN	203	
CS	102	
HIS	301	
MUS	410	20005
MUS	100	20006
ECN	101	20007
HIS	101	

rooms

Room ID	Building	Room Number
20000	Building 7	201
20001	Building 6	101
20002	Building 6	150
20003	Building 6	160
20004	Building 6	170
20005	Music Building	100
20006	Music Building	200
20007	Building 7	300
20008	Building 7	310

classes_rooms

Dept	Course	Building	Room Number
MUS	410	Music Building	100
ECN	101	Building 7	300
NUT	307	Building 7	201
MUS	100	Music Building	200

图6-4 删除操作后表的内容

6.2.3 创建系统触发器

Oracle 8i及更高版本

正如我们在前几节所看到的，DML和替代触发器都在（或代替）DML事件，即INSERT、UPDATE、DELETE语句上激活。而系统触发器可以在两种不同的事件即DDL或数据库上激活。DDL事件包括CREATE、ALTER或DROP语句，而数据库事件包括服务器的启动或关闭，用户的登录或退出，以及服务器错误。创建系统触发器的语法如下：

```
CREATE [OR REPLACE] TRIGGER [ schema.] trigger_name
  {BEFORE | AFTER}
  { ddl_event_list| database_event_list}
  ON {DATABASE | [ schema.]SCHEMA}
  [ when_clause]
```

```
trigger_body;
```

其中，ddl_event_list是一个或多个DDL事件（事件之间用OR分隔），database_event_list是一个或多个数据库事件（事件之间用‘ OR ’分隔）。

表6-4说明了DDL和数据库事件的种类以及这些事件出现的时机（之前或以后）。系统不支持替代系统触发器，TRUNCATE没有对应的数据库事件。

注意 创建系统触发器必须具有系统权限ADMINISTER DATABASE TRIGGER。本章6.2.4节中的“触发器权限”提供了详细信息。

表6-4 系统DLL和数据库事件

事 件	允许时机	说 明
启动	之后	实例启动时激活
关闭	之前	实例关闭时激活。如果数据库非正常关闭（如关闭故障），则该事件不激活
服务器错误	之后	只要有该类错误就激活
登录	之后	在用户成功连接数据库后激活
注销	之前	在用户注销开始时激活
创建	之前，之后	在创建模式对象之前或之后激活
撤消	之前，之后	在创建模式对象撤消之前或之后激活
变更	之前，之后	在创建模式对象变更之前或之后激活

1. 数据库与模式触发器的比较

系统触发器可以在数据库级或模式级定义。数据库级的触发器不管触发事件何时发生都将激活，而模式级触发器只有在指定的模式的触发事件发生时才会激活。关键字 DATABASE和SCHEMA决定了给定系统触发器的等级。如果没有用关键字 SCHEMA来说明模式，则以触发器所属的模式为默认模式。例如，假设我们在作为数据库的联机用户 UserA时，创建了下列的触发器：

节选自在线代码DatabaseSchema.sql

```
CREATE OR REPLACE TRIGGER LogUserAConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (1, 'LogUserAConnects fired!');
END LogUserAConnects;
```

字符串LogUserAConnects将在UserA与数据库建立连接时记录在表 temp_table中。我们可以通过创建下面的触发器在用户 UserB与数据库建立连接时为用户 UserB做相同的记录：

节选自在线代码DatabaseSchema.sql

```
CREATE OR REPLACE TRIGGER LogUserBConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (2, 'LogUserBConnects fired!');
END LogUserBConnects;
```

最后，我们可以在以 example的身份与数据库建立连接时创建下面的触发器。由于 LogAll Connects触发器是数据库级触发器，所以它可以把所有的数据库连接都记录在数据库中。

节选自在线代码DatabaseSchema.sql

```
CREATE OR REPLACE TRIGGER LogAllConnects
  AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO example.temp_table
    VALUES (3, 'LogAllConnects fired!');
END LogAllConnects;
```

注意 我们必须首先创建UserA和UserB,并在运行上面的例子前把相应的权限赋予这些用户。请看程序DatabaseSchema.sql中的实现代码。

现在我们可以在SQL *Plus会话中看到不同触发器的影响。

节选自在线代码DatabaseSchema.sql

```
SQL> connect UserA/UserA
Connected.
SQL> connect UserB/UserB
Connected.
SQL> connect example/example
Connected.
SQL>
SQL> SELECT * FROM temp_table;
   NUM_COL CHAR_COL
---------- ------------------------------------------------------------
         3 LogAllConnects fired!
         2 LogUserBConnects fired!
         3 LogAllConnects fired!
         3 LogAllConnects fired!
         1 LogUserAConnects fired!
```

LogAllConnects触发器被激活了三次（每个连接激活一次），而LogUserAConnects和LogUserBConnects按预期只激活了一次。

注意 STARTUP和SHUTDOWN触发器只与数据库级有关。虽然在模式级创建它们是合法的，但它们不会被激活。

2. 事件属性函数

系统触发器有几个内部的属性函数可供使用。这些函数类似于我们在前一节介绍的触发器参数（INSETING，UPDATING，DELETING），这些参数允许触发器体获得有关触发事件的信息。尽管从其他的 PL/SQL块中调用这些函数是合法的（在系统触发器中不必这样调用），但有时这些函数也会返回我们不希望的结果。表 6-5对这些事件属性函数做了说明。

我们在本章的开始部分介绍的触发器 LogCreations中使用了这些属性函数。与触发器参数不同，事件属性函数是 SYS拥有的独立 PL/SQL函数。系统没有为这些函数指定默认的替代名称，所以为了识别这些函数，在程序中必须在它们的前面加上前缀 SYS。

节选自在线代码LogCreations.sql

```
CREATE OR REPLACE TRIGGER LogCreations
  AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ddl_creations (user_id, object_type, object_name,
                             object_owner, creation_date)
    VALUES (USER, SYS.DICTIONARY_OBJ_TYPE, SYS.DICTIONARY_OBJ_NAME,
            SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;
```

表6-5 事件属性函数

属性函数	数据类型	可应用的系统事件	说 明
SYSEVENT	VARCHAR2（20）	所有事件	返回激活触发器的系统事件
INSTANCE_NUM	NUMBER	所有事件	返回当前实例号。在不运行OPS的情况下，该号为1
DATABASE_NAME	VARCHAR2(50)	所有事件	返回当前数据库名
SERVER_ERROR	NUMBER	SERVERERROR	接收一个NUMBER类型的参数，返回由该参数所指示的错误堆栈中相应位置的错误。错误堆栈的顶部对应于位置1
IS_SEVERERROR	BOOLEAN	SERVERERROR	接收一个错误号作为参数，如果所指示的Oracle错误返回在堆栈中，则返回真值（TRUR）
LOGIN_USER	VARCHAR2(30)	所有事件	返回激活触发器的用户的userid
DICTIONARY_OBJ_TYPE	VARCHAR2(20)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的类型
DICTIONARY_OBJ_NAME	VARCHAR2(30)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的名称
DICTIONARY_OBJ_OWNER	VARCHAR2(30)	CREATE, DROP, ALTER	返回激活触发器的DDL操作使用的字典对象的拥有者
DES_ENCRYPTED_PASSWORD	VARCHAR2(30)	CREATE用户或 ALTER	返回正在创建或变更用户的使用DES加密的口令

3. 使用SERVERERROR事件

事件SERVERERROR可以用于跟踪数据库中发生的错误。其错误代码可以使用触发器内部的SERVER_ERROR属性函数取出。该函数可以让用户确定堆栈中的错误码。然而，该函数不能返回与该错误码相关的错误信息。

上述缺点可以通过使用过程 DBMS_UTILITY.FORMAT_ERROR_STACK来解决。尽管触发器本身不会引发错误，但借助于该过程，我们可以使用 PL/SQL来访问错误堆栈。下面是演示上述过程的例子，该程序将错误记录在下面的表中：

节选自在线代码LogErrors.sql

```
CREATE TABLE error_log (
  timestamp     DATE,
  username      VARCHAR2(30),
  instance      NUMBER,
  database_name VARCHAR2(50),
  error_stack   VARCHAR2(2000)
  );
```

我们可以创建一个如下所示的插入表error_log的触发器：

节选自在线代码LogErrors.sql

```
CREATE OR REPLACE TRIGGER LogErrors
  AFTER SERVERERROR ON DATABASE
BEGIN
  INSERT INTO error_log
    VALUES (SYSDATE, SYS.LOGIN_USER, SYS.INSTANCE_NUM, SYS.
            DATABASE_NAME, DBMS_UTILITY.FORMAT_ERROR_STACK);
END LogErrors;
```

最后，我们可以生成几个错误并来看过程LogErrors怎样来记录这些错误信息。请注意可以捕捉SQL中的错误、运行时PL/SQL错误和编译时的PL/SQL错误。

节选自在线代码LogErrors.sql

```
SQL> SELECT * FROM non_existent_table;
SELECT * FROM non_existent_table
              *
ERROR at line 1:
ORA-00942: table or view does not exist
SQL> BEGIN
  2 INSERT INTO non_existent_table VALUES ('Hello!');
  3 END;
  4 /
  INSERT INTO non_existent_table VALUES ('Hello!');
              *
ERROR at line 2:
ORA-06550: line 2, column 15:
PLS-00201: identifier 'NON_EXISTENT_TABLE' must be declared
ORA-06550: line 2, column 3:
PL/SQL: SQL Statement ignored
SQL> BEGIN
  2 -- This is a syntax error!
  3 DELETE FROM students
  4 END;
  5 /
END;
*
ERROR at line 4:
ORA-06550: line 4, column 1:
PLS-00103: Encountered the symbol "END" when expecting one of the
following:
. @ ; return RETURNING_ <an identifier>
<a double-quoted delimited-identifier> partition where
The symbol ";" was substituted for "END" to continue.
SQL> SELECT *
  2 FROM error_log;

TIMESTAMP USERNAME INSTANCE DATABASE
```

```
--------- -------- --------- --------
ERROR_STACK
------------------------------------------------------------
30-AUG-99 EXAMPLE          1 V815
ORA-00942: table or view does not exist

30-AUG-99 EXAMPLE          1 V815
ORA-06550: line 2, column 15:
PLS-00201: identifier 'NON_EXISTENT_TABLE' must be declared
ORA-06550: line 2, column 3:
PL/SQL: SQL Statement ignored

30-AUG-99 EXAMPLE          1 V815
ORA-06550: line 4, column 1:
PLS-00103: Encountered the symbol "END" when expecting one of
  the following:

    . @ ; return RETURNING_ <an identifier>
    <a double-quoted delimited-identifier> partition where
The symbol ";" was substituted for "END" to continue.
```

4. 系统触发器和事务

系统触发器的事务特性与触发事件有关。系统触发器可以作为基于触发器正常结束时提交的独立事务激活，也可以作为当前用户事务的一部分激活。触发器 STARTUP，SHUTDOWN，SEVERERROR和LOGON都是由独立事务激活的，而 LOGOFF和DDL触发器则作为当前事务的一部分被激活。

需要注意的是，触发器实现的任务将被提交处理。在使用 DDL触发器的情况下，当前事务（也就是CREATE、ALTER或DROP语句）将自动提交。触发器 LOGOFF的操作也将作为会话中最后事务的一部分提交。

注意 由于系统触发器一般都要提交，因此把这类触发器声明为自主事务是没有意义的。请看本书第11章介绍自主事务的内容。

5. 系统触发器和WHEN字句

就象DML触发器一样，系统触发器可以使用 WHEN子句来指定触发器激活条件。然而，对每一种系统触发器所指定的条件类型有如下限制：

- STARTUP和SHUTDOWN触发器不能带有任何条件。
- SERVERERROR触发器可以使用ERRNO测试来检查特定的错误。
- LOGON 和LOGOFF触发器可以使用USERID或USERNAME测试来检查用户标识或用户名。
- DDL触发器可以检查正在修改对象的名称和类型。

6.2.4 其他触发器问题

我们在本节将讨论有关触发器的最后一些问题。其中包括触发器名称的命名空间（ Name-

space），使用触发器的各种限制，和不同的触发器体。本节的最后将讨论与触发器有关的权限问题。

1. 触发器名称

触发器的命名空间不同于其他子程序的命名空间。所谓命名空间就是一组合法的可供对象作为名字使用的标识符。过程，包和表都共享同一个命名空间，这就是说，在一个数据库模式范围内，同一命名空间内的所有的对象必须具有唯一的名称。例如，把同一个名字同时赋予一个过程和包就是非法的。

然而，触发器隶属于一个独立的命名空间。也就是说，触发器可以有与表或过程相同的名称。在一个模式范围内，然而，给定的名称只能用于一个触发器。例如，我们可以创建一个建立在表major_stats上叫做major_stats触发器，当但是，如果再创建一个叫做 major_stats的过程就是非法操作，下面的SQL *Plus会话演示了上述规则：

节选自在线代码Samename.sql

```
SQL> -- Legal, since triggers and tables are in different namespaces.
SQL> CREATE OR REPLACE TRIGGER major_stats
  2    BEFORE INSERT ON major_stats
  3 BEGIN
  4    INSERT INTO temp_table (char_col)
  5      VALUES ('Trigger fired!');
  6 END major_stats;
  7 /
Trigger created.
SQL> -- Illegal, since procedures and tables are in the same namespace.
SQL> CREATE OR REPLACE PROCEDURE major_stats AS
  2 BEGIN
  3    INSERT INTO temp_table (char_col)
  4      VALUES ('Procedure called!');
  5 END major_stats;
  6 /
CREATE OR REPLACE PROCEDURE major_stats AS
*
ERROR at line 1:
ORA-00955: name is already used by an existing object
```

提示 尽管触发器和表可以共用一个名称，但我们不推荐使用这种命名方法。最好是给每个触发器和表都赋予一个标识其功能的唯一的名称，也可以在触发器的名称加上如TRG_之类的前缀。

1. 对触发器的限制

触发器的体是一个PL/SQL块。（Oracle8i允许其他类型的触发器体，下节将讨论。）除去下面的限制外，在PL/SQL块中可以使用的语句也可以使用在触发器的体中：

- 触发器不能发布任何事务控制语句，如 COMMIT、ROLLBACK、SAVEPOINT或SET TRANSACTION。PL/SQL编译器允许触发器体中出现上述控制语句，但当该触发器激活时，将出现错误提示。这是因为该触发器是作为触发语句的执行部分被激活，并且与触发语句位于同一个事务中。当触发语句被提交或重新运行时，则该触发器所做的工作也将被

提交或重新开始。（在Oracle8i下，我们可以创建作为自动事务运行的触发器，这时，触发器所做的工作就可以独立于触发语句的状态而独立提交或返回起始点。本书第 11章对自动事务做了详细介绍。）

- 与上一条类似，由触发器体调用的任何过程或函数都不能发布任何事务控制命令（除非在Oracle8i下把它们声明为自动类型）。
- 触发器体不能声明任何LONG或LONG RAW变量。同样， :new和:old也不能引用定义触发器所用表的LONG 或LONG RAW类型的列。
- 在Oracle8及更高版本中，触发器体内的代码可以引用和使用 LOB（大型对象）的列，但不能修改该列的值。上述限制也适用于对象列。

除此之外，还有对触发器访问的表的操作限制。根据触发器类型和对该表的限制，有时，表可能要进行调整。

3. 触发器体

Oracle 8i 及更高版本

在Oracle8i之前的版本中，触发器的体必须是 PL/SQL块。而在Oracle8i下，触发器的体可以由调用语句组成，所调用的过程子程序可以是 PL/SQL存储子程序，或是C语言使用的包（wrapper）,以及Java程序。借助于这种选择，我们可以创建一个其基础代码是用Java编制的触发器。例如，假设我们要把数据库的连接和断开信息记录在下面的表中：

节选自在线代码relGTables.sql

```
CREATE TABLE connect_audit (
  user_name VARCHAR2(30),
  operation VARCHAR2(30),
  timestamp DATE);
```

现在，我们用下面的包来记录连接和断开信息：

节选自在线代码LogPkg.sql

```
CREATE OR REPLACE PACKAGE LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2);
  PROCEDURE LogDisconnect(p_UserID IN VARCHAR2);
END LogPkg;
CREATE OR REPLACE PACKAGE BODY LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2) IS
  BEGIN
    INSERT INTO connect_audit (user_name, operation, timestamp)
      VALUES (p_USerID, 'CONNECT', SYSDATE);
END LogConnect;
PROCEDURE LogDisconnect(p_UserID IN VARCHAR2) IS
BEGIN
  INSERT INTO connect_audit (user_name, operation, timestamp)
      VALUES (p_USerID, 'DISCONNECT', SYSDATE);
  END LogDisconnect;
END LogPkg;
```

包LongPkg.LogConnect和LogPkg.LogDisconnect都以用户名作为其入口参数，并在表connect_audit中插入一行。最后，我们可以从触发器 LOGON和LOGOFF中按下列代码调用这两

个触发器：

节选自在线代码LogConnects.sql

```
CREATE OR REPLACE TRIGGER LogConnects
  AFTER LOGON ON DATABASE
  CALL LogPkg.LogConnect(SYS.LOGIN_USER)
/

CREATE OR REPLACE TRIGGER LogDisconnects
  BEFORE LOGOFF ON DATABASE
  CALL LogPkg.LogDisconnect(SYS.LOGIN_USER)
/
```

注意 由于LogConnect和LogDisconnect都是系统数据库触发器，（与模式相反），创建上述触发器必须具有系统权限ADMINISTER DATABASE TRIGGER。

LogConnect和LogDisconnect的触发器体只是简单的调用语句，用来指出待执行的过程，当前用户是它们的唯一入口参数。在前面的例子中，语句CALL的目标是标准的PL/SQL打包过程。然而，语句CALL的目标也可以是简单的C语言使用的包或Java外部例程使用的已包装。例如，假设我们把下面的Java类载入到数据库中：

节选自在线代码Loger.sql

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class Logger {
  public static void LogConnect(String userID)
  throws SQLException {
    // Get default JDBC connection
    Connection conn = new OracleDriver().defaultConnection();

    String insertString =
      "INSERT INTO connect_audit (user_name, operation, timestamp)" +
      " VALUES (?, 'CONNECT', SYSDATE)";

    // Prepare and execute a statement that does the insert
    PreparedStatement insertStatement =
      conn.prepareStatement(insertString);
    insertStatement.setString(1, userID);
    insertStatement.execute();
  }

  public static void LogDisconnect(String userID)
  throws SQLException {
    // Get default JDBC connection
    Connection conn = new OracleDriver().defaultConnection();

    String insertString =
      "INSERT INTO connect_audit (user_name, operation, timestamp)" +
```

```
      " VALUES (?, 'DISCONNECT', SYSDATE)";

    // Prepare and execute a statement that does the insert
    PreparedStatement insertStatement =
      conn.prepareStatement(insertString);
    insertStatement.setString(1, userID);
    insertStatement.execute();
  }
}
```

如果我们接着再创建一个如下所示的作为该类包装的包LogPkg:

节选自在线代码LogPkg2.sql

```
CREATE OR REPLACE PACKAGE LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2);
  PROCEDURE LogDisconnect(p_UserID IN VARCHAR2);
END LogPkg;

CREATE OR REPLACE PACKAGE BODY LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2) IS
    LANGUAGE JAVA
    NAME 'Logger.LogConnect(java.lang.String)';

PROCEDURE LogDisconnect(p_UserID IN VARCHAR2) IS
    LANGUAGE JAVA
    NAME 'Logger.LogDisconnect(java.lang.String)';
END LogPkg;
```

我们可以使用相同的触发器来实现上述要求。读者请看本书第10章有关外部例程及如何把Java过程载入到数据库中的内容的介绍。

注意 触发器参数如INSERTING、UPDATING和DELETING，以及:new和:old相关标识符（还有:parent），只有在触发器体是完整的PL/SQL块而不是CALL语句的情况下才可以使用。

4. 触发器权限

下面的表6-6说明了五个适用于触发器的系统权限。除此之外，触发器的拥有者必须还具有对其引用的对象的权限。由于触发器是已经编译的对象（从Oracle7的7.3版本开始），所有权限都必须直接授权，不能通过角色授权.

表6-6 与触发器有关的系统权限

系统权限	说明
CREATE TRIGGER	授予在其自己的模式下创建触发器的权限
CREATE ANY TRIGGER	授予在任何模式（除了SYS外）下创建触发器的权限。在数据字典表中不推荐使用创建触发器
ALTER ANY TRIGGER	授予在任何模式（除了SYS外）下启用、禁用或编译数据库触发器的权限。注意，如果没有授予GREATE ANY TRIGGER权限，用户不能改变发触发器代码

（续）

系统权限	说 明
DROP ANY TRIGGER	授予在任何模式下（除去SYS）撤消数据库触发器的权限
ADMINISTER DATABASE TRIGGER	授予创建或变更数据库系统触发器的权限（与当前模式相反）。所授的权限必须具有CREATE TRIGGER或CREATE ANY TRIGGER

6.2.5 触发器与数据字典

与存储子程序类似，数据字典视图包括了有关触发器和其执行状态的信息。这些视图必须在触发器创建或撤消时进行更新。

1. 数据字典视图

当创建了一个触发器时，其源程序代码存储在数据库视图user_triggers中。该视图包括了触发器体，WHEN子句，触发表，和触发器类型。例如，下面的查询返回有关UpdateMajorStats的信息：

```
SQL> SELECT trigger_type, table_name, triggering_event
  2    FROM user_triggers
  3    WHERE trigger_name = 'UPDATEMAJORSTATS';
TRIGGER_TYPE     TABLE_NAME     TRIGGERING_EVENT
---------------- -------------- --------------------------
AFTER STATEMENT  STUDENTS INSERT OR UPDATE OR DELETE
```

有关数据字典视图的详细介绍，请看附录C。

2. 撤消和禁止触发器

与过程和包相类似，触发器也可以被撤消。实现撤消功能的命令如下：

```
DROP TRIGGER triggername;
```

其中，triggername是触发器的名称。该命令把指定的触发器从数据字典中永久性地删除。类似于子程序，子句OR REPLACE可用在触发器的CREATE语句中。在这种情况下，要创建的触发器已存在的话，则先将其删除。

与过程和包不同的是，触发器可以被禁止使用。当触发器被禁止时，它仍将存储在数据字典中，但不再激活。禁止触发器的语句如下：

```
ALTER TRIGGER triggername{DISABLE | ENABLE};
```

其中，triggername是触发器的名称。当创建触发器时，所有触发器的默认值都是允许状态（ENABLED）。语句ALTER TRIGGER可以禁止，或再允许任何触发器。例如，下面的代码先禁止接着再允许激活触发器UpdateMajorStats：

```
SQL> ALTER TRIGGER UpdateMajorStats DISABLE;
Trigger altered.

SQL> ALTER TRIGGER UpdateMajorStats ENABLE;
Trigger altered.
```

在使用命令ALTER TABLE的同时加入ENABLE ALL TRIGGERS或DISABLE ALL

TRIGGERS子句可以将指定表的所有触发器禁止或允许。例如：

```
SQL> ALTER TABLE students
  2    ENABLE ALL TRIGGERS;
Table altered.
SQL> ALTER TABLE students
  2    DISABLE ALL TRIGGERS;
Table altered.
```

视图user_triggers的status列包括有ENABLED或DISABLED两个字符串用来指示触发器的当前状态。禁止一个触发器将不从其数据字典中删除。

3. p-code触发器

当包或子程序存储在数据字典中时，其编译后的中间代码将同该对象的源代码一起存储，对于触发器来说情况也一样。这就是说触发器不需重新编译就可调用，并且其相关信息也得到保存。因此，触发器也可以向包和子程序那样自动无效。当触发器处于无效状态时，该触发器将在下次激活时自动重编译。

在Oracle7的7.3版前，系统对触发器的处理是不同的。在数据字典中唯一存储的是触发器的源代码，而没有中间代码。结果，触发器每次从数据字典中载入时都要进行编译。虽然这样处理并不影响触发器的定义和使用，但降低了触发器的运行效率。

6.3 变异表

系统对触发器将要访问的表和列有一些限制。为了定义这些限制，有必要来理解什么是变异表（mutating table）和约束表(constraining table)，变异表就是当前被DML语句修改的表。对触发器来说，变异表就是触发器在其上进行定义的表；由于执行 DELETE CASCADE引用完整性约束而需要更新的表也是变异表。（有关引用完整性约束的详细信息，请看 Oracle服务器参考手册。）约束表是一种需要实施引用完整性约束而读入的表。为了说明这些定义，请看表registered_students,该表的结构如下：

节选自在线代码relTables.sql

```
CREATE TABLE registered_students (
  student_id NUMBER(5) NOT NULL,
  department CHAR(3) NOT NULL,
  course NUMBER(3) NOT NULL,
  grade CHAR(1),
  CONSTRAINT rs_grade
    CHECK (grade IN ('A', 'B', 'C', 'D', 'E')),
  CONSTRAINT rs_student_id
    FOREIGN KEY (student_id) REFERENCES students (id),
  CONSTRAINT rs_department_course
    FOREIGN KEY (department, course)
    REFERENCES classes (department, course)
);
```

表registered_students有两个声明的引用完整性约束。在这种约束下，对表 registered-stuelents来说，表students和classes都是约束表。由于这些限制，表 classes和studengts也需要由

DML语句修改或查询。同样，表 registered_students自身在DML语句的对其操作期间也是变异表。

触发器体中的SQL语句不能进行下列操作：

- 读或修改触发语句的任何变异表，其中包括触发表本身。
- 读或修改触发表的约束表中的主关键字，唯一关键字和外部关键字列。除此之外的其他列可以修改。

上述限制适用于所有的行级触发器，这些限制只在语句触发器作为 DELETE CASCADE操作的结果激活时适用于语句触发器。

注意 如果INSERT语句只影响一行的话，则在该行的之前和之后触发器将不把触发表作为变异表对待，这是在行级触发器可能载入或修改触发表时的唯一案例。下面一类的语句

```
INSERT INTO table SELECT ...
```

总是把触发表作为变异表对待，即使其子查询仅返回一行也是如此。

作为例子，请考虑下面的触发器 CascadeRSInserts。尽管该触发器修改表 students和classes，但由于修改的表students和classes中的列不是关键字列，所以修改是合法的。下面，我们将分析一个非法的触发器案例。

节选自在线代码CascadeRSInserts .sql

```
CREATE OR REPLACE TRIGGER CascadeRSInserts
  /* Keep the registered_students, students, and classes
    tables in synch when an INSERT is done to registered_students. */
  BEFORE INSERT ON registered_students
  FOR EACH ROW
DECLARE
  v_Credits classes.num_credits%TYPE;
BEGIN
  -- Determine the number of credits for this class.
  SELECT num_credits
    INTO v_Credits
    FROM classes
    WHERE department = :new.department
    AND course = :new.course;

  -- Modify the current credits for this student.
  UPDATE students
    SET current_credits = current_credits + v_Credits
    WHERE ID = :new.student_id;

  -- Add one to the number of students in the class.
  UPDATE classes
    SET current_students = current_students + 1
    WHERE department = :new.department
    AND course = :new.course;
```

```
END CascadeRSInserts;
```

6.3.1 变异表案例介绍

假设我们要把每个专业的学生名额限制在五个。我们可以通过使用表 studengts**上的之前插入或更新行级触发器来实现这种限制，下面是该触发器的代码：**

节选自在线代码LimitMajors .sql

```
CREATE OR REPLACE TRIGGER LimitMajors
  /* Limits the number of students in each major to 5.
    If this limit is exceeded, an error is raised through
    raise_application_error. */
  BEFORE INSERT OR UPDATE OF major ON students
  FOR EACH ROW
DECLARE
  v_MaxStudents CONSTANT NUMBER := 5;
  v_CurrentStudents NUMBER;
BEGIN
  -- Determine the current number of students in this
  -- major.
  SELECT COUNT(*)
    INTO v_CurrentStudents
    FROM students
    WHERE major = :new.major;

  -- If there isn't room, raise an error.
  IF v_CurrentStudents + 1 > v_MaxStudents THEN
    RAISE_APPLICATION_ERROR(-20000,
      'Too many students in major ' || :new.major);
  END IF;
END LimitMajors;
```

初看，该触发器好象可以实现需要的功能。然而，如果我们更新表 students**并激活该触发器，我们会得到下面的输出：**

节选自在线代码LimitMajors .sql

```
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10003;
UPDATE students
 *
ERROR at line 1:
ORA-04091: table EXAMPLE.STUDENTS is mutating, trigger/function
           may not see it
ORA-06512: at line 7
ORA-04088: error during execution of trigger 'EXAMPLE.LIMITMAJORS'
```

由于触发器LimitMajor**查询其自己的触发表（该表是变异表），所以导致了错误**ORA-4091**。另外，错误**ORA-4091**是在该触发器激活时引发的，而不是在创建时引发的。**

6.3.2 变异表错误的处理

表students由于使用了行级触发器而成了变异表，这就使我们不能在该行级触发器中对该表进行查询，而只能在语句级触发器使用查询语句。然而，我们还不能只是简单地把触发器LimitMajor修改为语句级触发器，这是因为我们需要在触发器体中使用 :new.major的值。解决该问题的方法是创建两个触发器，即一个行级触发器和一个语句级触发器。在行级触发器中，我们记录:new.major的值，但我们不查询表 students；而查询任务由语句级触发器来实现并使用行触发器记录的值。

至于如何记录:new.major的值。一种方法是在包的内部使用 PL/SQL表来记录。用这种方法，我们可以在每次更新时保存多个值。同样，每个会话也得到其自己打包变量的实例，因此，我们就不必担心由于不同会话进行同时更新操作而引起的麻烦。实现上述方案的包 student_data,以及改进的触发器RLimitMajors和SLimiMajors的程序如下：

节选自在线代码mutating .sql

```
CREATE OR REPLACE PACKAGE StudentData AS
  TYPE t_Majors IS TABLE OF students.major%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE t_IDs IS TABLE OF students.ID%TYPE
    INDEX BY BINARY_INTEGER;

  v_StudentMajors t_Majors;
  v_StudentIDs t_IDs;
  v_NumEntries BINARY_INTEGER := 0;
END StudentData;

CREATE OR REPLACE TRIGGER RLimitMajors
  BEFORE INSERT OR UPDATE OF major ON students
  FOR EACH ROW
BEGIN
  /* Record the new data in StudentData. We don't make any
     changes to students, to avoid the ORA-4091 error. */
  StudentData.v_NumEntries := StudentData.v_NumEntries + 1;
  StudentData.v_StudentMajors(StudentData.v_NumEntries) :=
    :new.major;
  StudentData.v_StudentIDs(StudentData.v_NumEntries) := :new.id;
END RLimitMajors;

CREATE OR REPLACE TRIGGER SLimitMajors
  AFTER INSERT OR UPDATE OF major ON students
DECLARE
  v_MaxStudents CONSTANT NUMBER := 5;
  v_CurrentStudents NUMBER;
  v_StudentID students.ID%TYPE;
  v_Major students.major%TYPE;
BEGIN
  /* Loop through each student inserted or updated, and verify
```

```
      that we are still within the limit. */
  FOR v_LoopIndex IN 1..StudentData.v_NumEntries LOOP
    v_StudentID := StudentData.v_StudentIDs(v_LoopIndex);
    v_Major := StudentData.v_StudentMajors(v_LoopIndex);

    -- Determine the current number of students in this major.
    SELECT COUNT(*)
      INTO v_CurrentStudents
      FROM students
      WHERE major = v_Major;

    -- If there isn't room, raise an error.
    IF v_CurrentStudents > v_MaxStudents THEN
      RAISE_APPLICATION_ERROR(-20000,
        'Too many students for major ' || v_Major ||
        ' because of student ' || v_StudentID);
    END IF;
  END LOOP;

  -- Reset the counter so the next execution will use new data.
  StudentData.v_NumEntries := 0;
END LimitMajors;
```

注意 请注意要在运行前面的脚本之前，一定要撤消不正确的 LimitMajors触发器。

现在，我们可以通过更新表 student**来测试这一系列触发器直到我们的程序中出现了过多的历史专业为止**

节选自在线代码mutating .sql

```
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10003;
1 row updated.
SQL> UPDATE students
  2    SET major = 'History'
  3    WHERE ID = 10002;
1 row updated.

SQL> UPDATE students
  2 SET major = 'History'
  3 WHERE ID = 10009;
UPDATE students
 *
ERROR at line 1:
ORA-20000: Too many students for major History because of student 10009
ORA-06512: at "EXAMPLE.SLIMITMAJORS", line 19
ORA-04088: error during execution of trigger 'EXAMPLE.SLIMITMAJORS'
```

上面的结果就是我们期望的功能。当行级触发器读入或修改变异表（ mutating table**）时，**

上述技术可以引发错误信息ORA-4091。为了避免在行级触发器中进行非法处理，我们把该处理推迟到一个之后的语句级触发器实现，这样一来该处理就是合法的了。打包的PL/SQL表是用来存储被修改的行的。

下面是应用该技术时要注意的几个问题：

- 由于PL/SQL表是位于包中，所以这些表对于行级触发器和语句级触发器都是可见的，确保变量的全局性的唯一方法是把要定义的全局变量放在包中。
- 我们在该程序中使用了计数器变量StudentsData.v_NumEntries，当其所在的包首次创建时，该变量的初始值为0。然后，该变量的值由行级触发器修改。语句级触发器对该变量进行引用并在处理结束后将该变量设置为0。上述的步骤是必须的，因为只有这样该会话发布的UPDATE语句将实现正确的结果。
- 对触发器SLimitMajors进行最多学生数量的检查将稍有变化。由于该触发器是语句之后触发器，所以变量v_CurrentStudents将在执行插入或更新命令后保存某专业的学生数量。因此，在触发器LimitMajor中对变量v_CurrentStudents+1 的检查也被变量v_CurrentStudents代替。
- 我们也可以使用数据库表来代替 PL/SQL表。我个人不喜欢这种技术，这是由于发布UPDATE的多个同时会话之间可能有相互用（在Oracle8i中，我们可以使用临时表）。打包的PL/SQL表是众多会话之间唯一的，因此解决了上面的问题。

6.4 小结

正如我们在本章所看到的，触发器是继PL／SQL与Oracle数据库之后的重要工具。我们可以使用该工具来加强比正常的引用完整性约束条件更复杂的数据约束。 Oracle8i把触发器扩展到了除了在表或视图上进行ＤＭＬ操作以外的事件。本章介绍的触发器结束了我们在前三章中对命名PL/SQL块的讨论。我们将在第13章中看到的另一种命名PL/SQL块是对象类型体。下面一章将讨论PL/SQL中的内置包问题。