数据库的安全性和完整性

- 一个完整的数据库里都保存什么？

- 关系（tables）是基本单位：关系模式和数据
- 视图（views）
- 约束关系(Constraints)：实体完整性、参照完整性、用户自定义完整性约束
- 触发器（Triggers）
- 存储过程(Stored Procedures)
- 存储函数(Functions)

本章重点内容

- SQL里如何对列、关系定义**约束规则**
- 什么是**触发器**，有什么用，如何设计
- 什么是**存储过程**、**存储函数**，如何设计使用？
- 如何管理数据库不同对象的访问权限？

数据库的完整性与安全性控制

---- 数据库完整性的概念

- 数据库完整性(DB Integrity)是指DBMS应保证DB在任何情况下的

  正确性、有效性和一致性

  - ✓广义完整性：语义完整性、并发控制、安全控制、DB故障恢复等
  - ✓狭义完整性：专指语义完整性，DBMS通常有专门的完整性管理机制与程序来处理语义完整性问题。(本讲义指语义完整性)

- 关系模型中有完整性要求
  - 实体完整性
  - 参照完整性
  - 用户自定义完整性

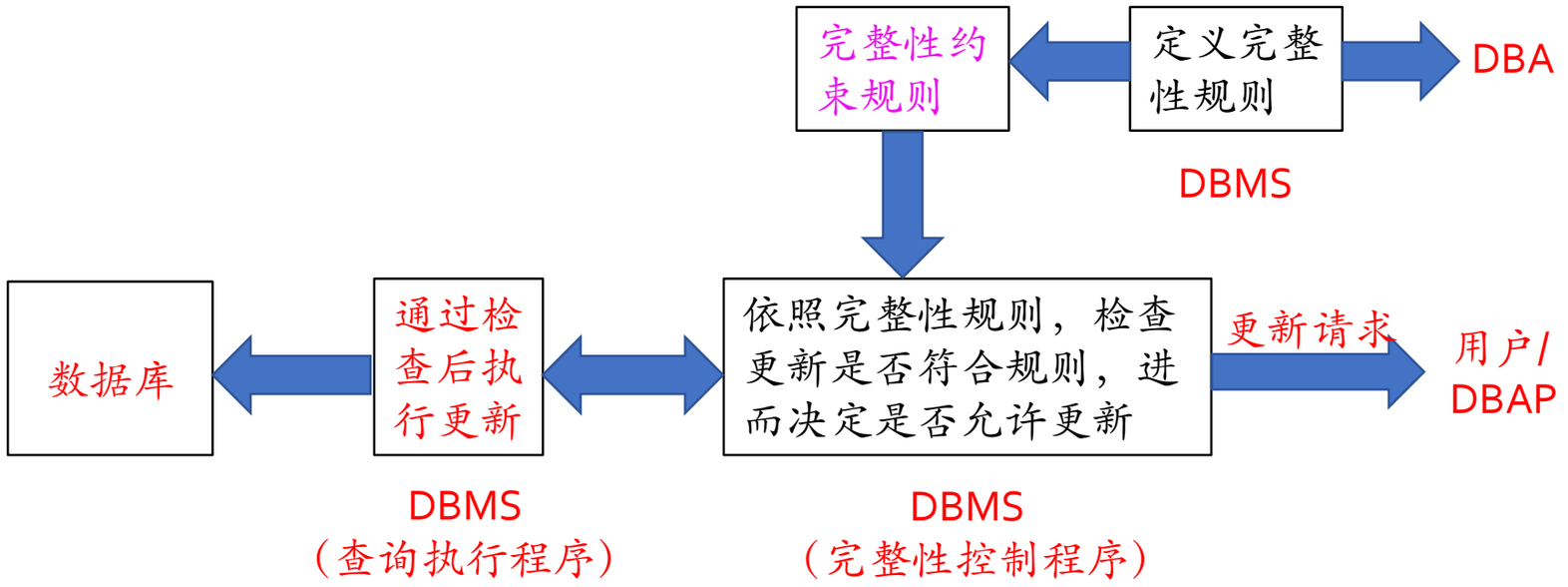
数据库的完整性与安全性控制

---- 数据库完整性的概念

- 为什么会引发数据库完整性的问题呢?
  - 不正当的数据库操作，如输入错误、操作失误、程序处理失误等
- 数据库完整性管理的作用
  - 防止和避免数据库中不合理数据的出现（salary<0）
  - DBMS应尽可能地自动防止DB中语义不合理现象
  - 如DBMS不能自动防止，则需要应用程序员和用户在进行数据库操作时处处加以小心，每写一条SQL语句都要考虑是否符合语义完整性，这种工作负担是非常沉重的，因此应尽可能多地让DBMS来承担。

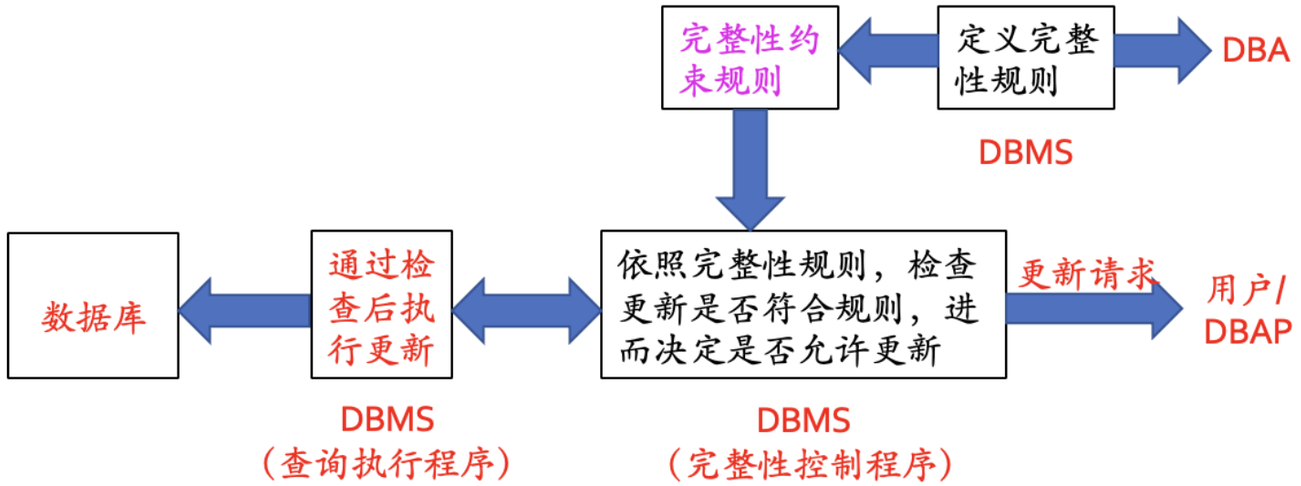
数据库的完整性与安全性控制

- DBMS怎样自动保证完整性呢?
  - DBMS允许用户定义一些完整性约束规则(用SQL-DDL来定义)
  - 当有DB更新操作时，DBMS自动按照完整性约束条件进行检查，以确保更新操作符合语义完整性



数据库的完整性与安全性控制

- 完整性约束条件(或称完整性约束规则)的一般形式 （Quad四元组）
  - Integrity Constraint::=（O，P，A，R)
    - ✓O—数据集合：约束的对象?
      - ❖ 列、多列(元组)、元组集合
    - ✓P—谓词条件：什么样的约束?
    - ✓A—触发条件：什么时候检查?
    - ✓R—响应动作：不满足时怎么办?



数据库的完整性与安全性控制

6<= chours/credit <=7

Course				
cno	cname	chours	credit	tno
001	数据库	40	6	001
003	数据结构	40	6	003
004	编译原理	40	6	001
005	C语言	30	4.5	003
002	高等数学	80	12	004

按约束对象分类

- 完整性约束条件的类别
  - 域完整性约束条件
    - ✓ 施加于某一列上，对给定列上所要更新的某一候选值是否可以接受 进行约束条件判断，这是孤立进行的。
  - 关系完整性约束条件
    - ✓ 施加于关系/table上，对给定table上所要更新的某一候选元组是否可以接受进行约束条件判断，或是对一个关系中的若干元组和另一个关系中的若干元组间的联系是否可以接受进行约束条件判断。

数据库的完整性与安全性控制

➢ 按约束来源分类

- **结构约束**：来自于模型的约束，例如函数依赖约束、主键约束(实体完整性)、外键约束(参照完整性)，只关心数值相等与否、是否允许空值等；
- **内容约束**：来自于用户的约束，如用户自定义完整性，关心元组或属性的取值范围。例如Student表的sage属性值在15岁至40岁之间等。

sno不允许有重复

sno不允许有空值

sage>15 AND sage<=40

dno必须在Dept表中存在

Student

sno	sname	ssex	sage	dno	sclass
21030101	张三	男	20	03	210301
21030102	张四	女	20	03	210301
21030103	张五	男	19	03	210301
21040201	王三	男	20	04	210402
21040202	王四	男	21	04	210402
21040203	王五	女	19	04	210402

数据库的完整性与安全性控制

➢ 按约束状态分类

- **静态约束：**

  要求DB在任一时候均应满足的约束；例如sage在任何时候都应满足大于0而小于150(假定人活最大年龄是150)。

- **动态约束：**

  要求DB从一状态变为另一状态时应满足的约束；

  例如工资只能升，不能降：工资可以是800元，也可以是1000元；可以从800元更改为1000元，但不能从1000元更改为800元。

salary不能从1000元更改为800元。

Teacher			
tno	tname	dno	salary
001	赵三	01	1200.00
002	赵四	03	1400.00
003	赵五	03	1000.00
004	赵六	04	1100.00

数据库的完整性与安全性控制

SQL语言支持如下几种约束

- 静态约束
  - 列完整性---域完整性约束：CHECK, UNIQUE, NOT NULL
  - 表完整性---关系完整性约束：主键、外键等
- 动态约束
  - 触发器

数据库的完整性与安全性控制

CREATE Table

- CREATE Table有三种功能：定义关系模式、定义完整性约束、定义物理存储特性
- 定义完整性约束条件：列完整性、表完整性

```
CREATE TABLE    tablename
    ( ( colname      datatype  [ DEFAULT { default_constant | NULL} ]
                    [ col_constr     {col_constr. . .} ]
              | , table_constr
  {,  { colname     datatype     [DEFAULT { default_constant | NULL} ]
                    [col_constr     {col_constr. . .} ]
              | , table_constr }
   . . . } );
```

数据库的完整性与安全性控制

➢ Col_constr列约束:

一种域约束类型，对单一列的值进行约束

```
{ NOT NULL |                                          //列值非空
    [ CONSTRAINT constraintname ]                     //为约束命名，便于以后修改或撤消
        { UNIQUE                                      //列值是唯一
        | PRIMARY KEY                                 //列为主键
        | CHECK    (search_cond)                      //列值满足条件,条件只能使用列当前值
        | REFERENCES      tablename      [(colname) ]
          [ON DELETE CASCADE] } }
```

//引用另一表tablename的列colname的值，如有ON DELETE CASCADE 或 ON DELETE SET NULL，则删除被引用表的某列值v 时，要将本表的该列值为v的记录删除或列值更新为null; 缺省为无操作 。

➢ Col_constr列约束：只能应用在单一列上，其后面的约束如UNIQUE, PRIMARY及search_cond只能是单一列唯一、单一列为主键、和单一列相关

数据库的完整性与安全性控制

- Col_constr列约束示例

```
CREATE  TABLE Student ( sno   char(8)  NOT NULL UNIQUE,
                        sname  char(10),
                        ssex char(2) CONSTRAINT ctssex CHECK (ssex='男'  OR  ssex='女' ),
                        sage  integer  CHECK (sage>=1  AND sage<150),
                       dno char(2)  REFERENCES  Dept(dno) ON DELETE CASCADE,
                      sclass  char(6) );
```

//假定ssex只能取{男， 女}, 1=<sage<=150, dno 是外键

Student					
sno	sname	ssex	sage	dno	sclass
21030101	张三	男	20	03	210301
21030102	张四	女	20	03	210301
21030103	张五	男	19	03	210301
21040201	王三	男	20	04	210402
21040202	王四	男	21	04	210402
21040203	王五	女	19	04	210402

Dept		
dno	dname	dean
01	机电	李三
02	能源	李四
03	计算机	李五
04	自动控制	李六

数据库的完整性与安全性控制

- table_constr表约束：

一种关系约束类型，对多列或元组的值进行约束

```
[ CONSTRAINT constraintname ]                    //为约束命名，便于以后修改或撤消
    { UNIQUE   (colname {, colname. . .})        //几列值组合一起是唯一
    | PRIMARY KEY    (colname {, colname. . .})      //几列联合为主键
    | CHECK (search_condition)                   //元组多列值共同满足条件
                                                 //条件中只能使用同一元组的不同列值
    | FOREIGN KEY (colname {, colname. . .})
            REFERENCES     tablename [(colname {, colname. . .})]  [ON
            DELETE CASCADE] }
                                    //引用另一列表tablename的若干列的值为外键
```

- table_constr表约束：是应用在关系上，即对关系的多列或元组进行约束，列约束是其特例

数据库的完整性与安全性控制

- table_constr表约束示例

```
CREATE  TABLE Student ( sno. char(8)  not null unique, sname  char(10),
        ssex char(2) constraint ctssex CHECK (ssex='男'  OR  ssex='女' ),
        sage  integer  CHECK (sage>1  AND sage<150),
        dno  char(2)  REFERENCES  Dept(dno) ON DELETE CASCADE,
        sclass  char(6),
        primary  key(sno) );

CREATE  TABLE  Course ( cno  char(3) ,  cname  char(12), chours integer,
          credit float(1) constraint ctcredit CHECK (credit >=0.0 AND  credit<=5.0 ),
          tno char(3) REFERENCES Teacher(tno) ON DELETE  CASCADE,
          primary key(cno),
          constraint  ctcc  CHECK(chours/credit = 20) );
                                        //假定严格约束20学时一个学分
```

数据库的完整性与安全性控制

➢表达外码:

```
CREATE  TABLE  SC ( sno  char(8) REFERENCES  student(sno) ON DELETE  CASCADE,
                    cno  char(3) REFERENCES  course(cno) ON DELETE  CASCADE ,
                   score float(1) constraint ctscore CHECK (score>=0.0 AND score<=100.0));

CREATE  TABLE  SC ( sno  char(8), cno  char(3),
                   score float(1) constraint ctscore CHECK (score>=0.0 AND  score<=100.0),
                   FOREIGH KEY (sno)  REFERENCES  student(sno) ON DELETE  CASCADE,
                   FOREIGH KEY(cno)  REFERENCES  course(cno) ON DELETE  CASCADE );

CREATE  TABLE  SC ( sno  char(8)  CHECK( sno  IN  (SELECT  sno  FROM Student)) ,
                    cno  char(3)  CHECK( cno  IN (SELECT  cno  FROM Course)) ,
                   score float(1) constraint ctscore CHECK (score>=0.0 AND  score<=100.0);
```

数据库的完整性与安全性控制

➢ CREATE Table中定义的表约束或列约束可以在以后根据需要进行撤消或追加。

撤消或追加约束的语句是 ALTER Table(不同系统可能有差异)

```
ALTER   TABLE   tblname
    [ADD   ( { colname datatype      [DEFAULT {default_const|NULL} ]
              [col_constr {col_constr...} ]  | , table_constr }
              {, colname ...}) ]
   [DROP  { COLUMN  columnname | (columnname {, columnname…})}]
   [MODIFY ( columnname  data-type
              [DEFAULT  {default_const | NULL} ] [      [ NOT ] NULL ]
              {, columnname . . .})]
   [ADD   CONSTRAINT  constr_name]
   [DROP   CONSTRAINT  constr_name]
   [DROP   PRIMARY KEY ] ;
```

数据库的完整性与安全性控制

- 例如，撤消SC表的ctscore约束(由此可见，未命名的约束是不能撤消)

```
ALTER  Table   SC
    DROP   CONSTRAINT  ctscore;
```

- 例如，若要再对SC表的score进行约束，比如分数在0~150之间，则可新增加一 个约束。

在Oracle中增加新约束，需要通过修改列的定义来完成

```
ALTER  Table   SC
    MODIFY ( score float(1) constraint nctscore CHECK (score>=0.0 AND
            score<=150.0) );
```

- 有些DBMS支持独立的追加约束, 注意书写格式可能有些差异

```
ALTER  Table   SC
    ADD  CONSTRAINT  nctscore    CHECK  (score>=0.0 AND score<=150.0) );
```

数据库的完整性与安全性控制

SQL语言支持如下几种约束

- 静态约束
  - 列完整性---域完整性约束
  - 表完整性---关系完整性约束
- 动态约束
  - 触发器

数据库的完整性与安全性控制

触发器Trigger

- 为实现动态约束以及多个元组之间的完整性约束，就需要触发器技术Trigger

- Trigger是一段程序（特殊的存储过程），用来实现过程性完整性约束。它在特定事件（例如对数据增、删、改等）发生时被自动触发执行。

- 而CREATE TABLE中的表约束和列约束基本上都是静态（非过程性）约束。

数据库的完整性与安全性控制

➢ 触发器Trigger

```
CREATE    TRIGGER   trigger_name   BEFORE | AFTER
{ INSERT | DELETE | UPDATE  [OF colname {, colname...}] }
ON  tablename  [REFERENCING corr_name_def {, corr_name_def...} ]
[FOR EACH ROW | FOR EACH STATEMENT]
        //对更新操作的每一条结果(前者)，或整个更新操作完成(后者)
    [WHEN   (search_condition)]          //检查条件，如满足执行下述程序
        {   statement                    //单行程序直接书写，多行程序要用下行方式
        | BEGIN ATOMIC statement; { statement;...} END }
```

➢ 触发器Trigger意义：当某一事件发生时(Before|After),对该事件产生的结 果(或是每一元组，或是整个操作的所有元组), 检查条件search_condition,

如果满足条件，则执行后面的程序段。条件或程序段中引用的变量可用corr_name_def来限定。

数据库的完整性与安全性控制

- 事件：BEFORE | AFTER { INSERT | DELETE | UPDATE …}
  - 当一个事件(INSERT, DELETE, 或UPDATE)发生之前Before或发生之后After触发
  - 操作发生，执行触发器操作需处理两组值：更新前的值和更新后的值，这两个值由corr_name_def的使用来区分
- corr_name_def的定义

```
{ OLD [ROW] [AS] old_row_corr_name          //更新前的旧元组命别名为
| NEW [ROW] [AS] new_row_corr_name          //更新后的新元组命别名为
| OLD TABLE [AS] old_table_corr_name        //更新前的旧Table命别名为
| NEW TABLE [AS] new_table_corr_name //更新后的新Table命别名为
}
```

- corr_name_def将在检测条件或后面的动作程序段中被引用处理

触发器分类

- 触发事件： INSERT | DELETE | UPDATE
- 运行时机：**BEFORE | AFTER**
  - 当一个事件(INSERT, DELETE, 或UPDATE)发生之前Before或发生之后After触发（6种组合）

例如：设计一个触发器当进行Teacher表更新元组时，使其工资只能升不能降，否则报错

触发器名字　触发时机　触发事件　事件的操作对象

```
CREATE TRIGGER  teacher_chgsal  BEFORE UPDATE OF salary  ON Teacher
    REFERENCING NEW x, OLD  y
    FOR  EACH  ROW  WHEN (x.salary < y.salary)
    BEGIN
        raise_application_error(-20003, 'invalid salary on update');
            //Oracle的错误处理函数
    END;
```

更新前后的元组分别用y, x指代（可自己决定命名）

检查条件，若满足则往下执行。可选。

所有要做的动作顺序执行

数据库的完整性与安全性控制

- 事件：BEFORE | AFTER { INSERT | DELETE | UPDATE …}
  - 当一个事件(INSERT, DELETE, 或UPDATE)发生之前Before或发生之后After触发
  - 操作发生，执行触发器操作需处理两组值：更新前的值和更新后的值，这两个值由corr_name_def的使用来区分
- corr_name_def的定义

```
{ OLD [ROW] [AS] old_row_corr_name          //更新前的旧元组命别名为
| NEW [ROW] [AS] new_row_corr_name          //更新后的新元组命别名为
| OLD TABLE [AS] old_table_corr_name        //更新前的旧Table命别名为
| NEW TABLE [AS] new_table_corr_name //更新后的新Table命别名为
}
```

- corr_name_def将在检测条件或后面的动作程序段中被引用处理

数据库的完整性与安全性控制

- 触发器Trigger示例一
- 设计一个触发器当进行Teacher表更新元组时，使其工资只能升不能降

```
CREATE TRIGGER  teacher_chgsal  BEFORE UPDATE OF salary  ON Teacher
    REFERENCING NEW x, OLD  y
    FOR  EACH  ROW  WHEN (x.salary < y.salary)  BEGIN
    raise_application_error(-20003, 'invalid salary on update');
    //Oracle的错误处理函数
    END;
```

Before Trigger

- Before类型的触发器会暂时阻止触发事件的执行。
- 待触发器内容执行完毕后，DBMS再继续执行触发事件。

- After类型的触发器则不影响触发动作的执行
- 待触发动作完成之后，再额外运行触发器内容。通常触发动作作用于触发事件对象以外的表或者列。

数据库的完整性与安全性控制

- 触发器Trigger示例二
- 假设student(sno, sname, sumcourse), sumcourse为该同学已学习课程的门数，以后每选修一门都要对其增1 。设计一个触发器完成上述功能

```
CREATETRIGGER  sumc  AFTER INSERT ON SC
REFERENCING NEW newi
 FOR EACH ROW
 BEGIN
        UPDATE    student
        SET sumcourse = sumcourse + 1
        WHERE sno =  newi.sno ;
END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例三
- 假设student(sno, sname, sage, ssex, sclass)中某一学生要变更其主码sno的值，如使其原来的98030101变更为99030131, 此时SC表中该同学已选课记录的sno也需自动随其改变。设计一个触发器完成上述功能

```
CREATE TRIGGER updSno AFTER UPDATE OF sno ON Student
 REFERENCING OLD oldi, NEW newi
 FOR EACH ROW
  BEGIN
        UPDATE SC
         SET sno = newi.sno
        WHERE sno = oldi.sno      ;
  END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例四
- 假设student(sno, sname, SumCourse), 当删除某一同学sno时，该同学的所有选课也都要删除。设计一个触发器完成上述功能

```
CREATE  TRIGGER  delSno  AFTER  DELETE  ON Student
REFERENCING OLD oldi
FOR EACH ROW
    BEGIN
        DELETE FROM SC WHERE sno = oldi.sno ;
    END;
```

只有OLD没有New, 为什么？

数据库的完整性与安全性控制

- 触发器Trigger示例五

- 假设student(sno, sname, sumcourse), 当删除某一同学sno时，该同学的所有选课中的sno都要置为空值。设计一个触发器完成上述功能

```
CREATE TRIGGER delSno AFTER DELETE ON Student
REFERENCING OLD oldi
FOR EACH ROW
    BEGIN
        UPDATE SC SET sno = Null WHERE sno= oldi.sno;
    END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例六

- 假设Dept(dno, dname, dean), 而dean一定是该系教师Teacher(tno, tname, dno, salary)中工资最高的教师。设计一个触发器完成上述功能

保证更新 Dept 后dean 工资是最高的

```
CREATE TRIGGER  updean BEFORE UPDATE OF dean ON Dept
REFERENCING  NEW  newi
     FOREACH  ROW WHEN ( newi.dean NOT IN
    (SELECT  Tname  FROM  Teacher  WHERE dno = newi.dno
AND   salary >=  ALL(SELECT salary FROM Teacher WHERE dno = newi.dno))
     BEGIN
 raise_application_error(-20003, 'invalid dean ON update');
     END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例六

- 假设Dept(dno, dname, dean), 而dean一定是该系教师Teacher(tno, tname, dno, salary)中工资最高的教师。设计一个触发器完成上述功能

保证更新 Dept 后dean 工资是最高的：当某系换系主任时，若新系主任非本息工资最高的教师（之一），则阻止此次更新操作

```
CREATE TRIGGER  updean BEFORE UPDATE OF dean ON Dept
REFERENCING  NEW  newi      //可以省去，直接用new和old指代新旧记录
    FOREACH  ROW WHEN ( newi.dean NOT IN
        (SELECT  Tname  FROM  Teacher  WHERE dno = newi.dno
  AND   salary >=  ALL(SELECT salary FROM Teacher WHERE dno = newi.dno))
      BEGIN
  raise_application_error(-20003, 'invalid dean ON update');
      END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例六

- 假设Dept(dno, dname, dean), 而dean一定是该系教师Teacher(tno, tname, dno, salary)中工资最高的教师。设计一个触发器完成上述功能

保证新的教师记录插入后Dean工资还是是最高的，否则制止此次插入操作

```
CREATE TRIGGER insertteacher BEFORE INSERT ON Teacher
    FOREACH ROW WHEN ( new.salary > ALL      //运行条件也可以用IF... ELSE写到主体内
    (SELECT salary FROM Teacher, Dept WHERE dno = new.dno AND
    Dept.dno=dno AND tname = dean) )
  BEGIN
     raise_application_error(-20003, 'invalid teacher ON insert');
  END;
```

数据库的完整性与安全性控制

- 触发器Trigger示例六

- 假设Dept(dno, dname, dean), 而dean一定是该系教师Teacher(tno, tname, dno, salary)中工资最高的教师。设计一个触发器完成上述功能

保证新的教师记录插入后Dean工资还是是最高的，否则制止此次插入操作

```
CREATE TRIGGER insertteacher BEFORE INSERT ON Teacher
FOR EACH ROW
BEGIN
   IF new.salary > ALL(SELECT salary FROM Teacher, Dept WHERE dno
    = new.dno AND Dept.dno=dno AND tname = dean)
  THEN
    raise_application_error(-20003, 'invalid teacher ON insert');
  END IF;
END;
```

数据库的完整性与安全性控制

➢ 触发器Trigger示例七

假设有如下的关系：

- Inventory（item，level），表示物品在仓库中的当前库存量。
- Minlevel（item，level），表示物品应该保持的最小库存量。
- Reorder（item，amount），表示当物品小于最小库存量的时候要订购的数量。
- Orders（item，amount），表示物品被订购的数量。

仅当物品库存量从大于最小值降到最小值以下的时候才下达一个订单，设计触发器完成以上功能。

数据库的完整性与安全性控制

➢ 触发器Trigger示例七

Inventory

Item	Level
1(矿泉水)	11→9
2(可乐)	50

Minlevel

Item	Level
1(矿泉水)	10
2(可乐)	20

Reorder

Item	amount
1(矿泉水)	100
2（可乐）	50

Order

Item	amount
1(矿泉水)	100

```
CREATE TRIGGER Reorder AFTER UPDATE OF level ON Inventory
    FOR EACH ROW
    WHEN new.level <=(SELECT level FROM Minlevel WHERE Minlevel.item=orow.item)
    AND old.level>(SELECT level FROM Minlevel WHERE Minlevel.item=orow.item)
    BEGIN
        INSERT INTO Orders (SELECT item, amount  FROM Reorder
                WHERE reorder.item=old.item);
    END;
```

数据库的完整性与安全性控制

➢ 触发器Trigger示例七

Inventory

Item	Level
1(矿泉水)	11→9
2(可乐)	50

Minlevel

Item	Level
1(矿泉水)	10
2(可乐)	20

Reorder

Item	amount
1(矿泉水)	100
2（可乐）	50

Order

Item	amount
1(矿泉水)	100

```
CREATE TRIGGER Reorder AFTER UPDATE OF level ON Inventory
FOR EACH ROW
BEGIN
    IF (new.level <= SELECT level FROM Minlevel WHERE Minlevel.item=old.item) AND
        (old.level > SELECT level FROM Minlevel WHERE Minlevel.item=old.item)
   THEN
        INSERT INTO Orders (SELECT item, amount  FROM Reorder WHERE reorder.item=old.item);
    END IF;
END;
```

互动小问题

- 假设有三张表

Student (sno, sname, dno, tot_cred),

SC (sno, cno, grade) // grade 等级分为A,B,C,F, null

Course (cno, cname, credit),

- 请设计一个触发器来使Student元组的tot_cred保持最新。当一门课通过，则对该同学的tot_cred加入相应学分。

Student (sno, sname, dno, tot_cred), SC (sno, cno, grade), Course (cno, cname, credit)

```
CREATE Trigger credits_earned after update of grade ON SC
For each row
WHEN new.grade IN ('A','B','C') AND (old.grade ='F' OR old.grade is null)
BEGIN
    UPDATE Student
    SET tot_cred= tot_cred +(SELECT credit From Course Where Cno=new.Cno)
    Where Sno= new.Sno;
END;
```

函数与过程

----存储过程和函数的概念

定义：类似于高级语言程序，过程化SQL程序也可以被命名和编译并保存在数据库中，称为存储过程（stored procedure）或存储函数（stored function），供其他过程化SQL调用。不同DBMS的语法和关键字有差别！

存储过程的优点：

- 创建时即完成SQL语句的语法分析与优化工作，运行效率更高，可在服务器端快速执行SQL
- 降低客户机与服务器之间的通信量，客户机只需发送调用存储过程的名字和参数，接受最终处理结果
- 方便实施企业规则，将企业规则的运算程序写为存储过程放入服务器，有利于集中控制和维护

存储过程或存储函数也是一类数据库的对象，需要有创建、删除等语句。

函数与过程

存储过程创建：

CREATE procedure 过程名（

[[IN|OUT|INOUT] 参数1 数据类型，

[[IN|OUT|INOUT] 参数2 数据类型，...]）/*存储过程首部*/

<过程化SQL块>；/*存储过程体，描述该存储过程的操作*/

IN：存储过程的输入参数，在被调用时需要指定参数值

OUT：输出参数，被调用后返回的参数值

INOUT：输入输出参数，调用时需要传入初始值，并会返回操作后的最终值

注：

- **如果不声明参数模式是输入还是输出参数，默认为IN**
- **过程化语句块：对基本SQL的扩展，在其基础上增加了一些描述过程控制的语句，如条件控制语句，循环控制语句，错误处理语句**

函数与过程

1. 建立一个存储过程，对一个指定名字的学生，其上过的指定老师的课中所有不及格成绩更新为60分。

```
CREATE PROCEDURE Update_Score_60 (IN studentname varchar(8), IN teachername varchar(8))
BEGIN
    UPDATE SC
    SET score = 60
    WHERE score<60 AND Sno IN (SELECT s.Sno FROM Student a JOIN SC ON s.sno=SC.sno
                                    JOIN Course c ON SC.cno = c.cno
                                    JOIN Teacher t ON t.tno= c.tno
                                WHERE s.sname = studentname AND t.tname = teachername);
END;
```

执行带输入参数的存储过程

```
CALL Update_Score_60('张三', '王五');
```

函数与过程

2. 创建带输出参数的存储过程, 查询某个分数的个数

```
CREATE PROCEDURE QueryGrade(s_grade DOUBLE,   OUT  grade_count  INT)
BEGIN
    SELECT COUNT(*) INTO grade_count FROM SC WHERE Score = QueryGrade .s_grade;
END;
```

执行带输出参数的存储过程，查询92分的数量

```
CALL QueryGrade (92，@grade_count);
SELECT @grade_count;
```

DECLARE <变量名><数据类型>;

SELECT <列名> INTO <变量名> FROM…

CALL <存储过程名>（参数1，参数2 …）

注意：以上语法仅限MySQL中使用。其他DBMS可能有不同语法
MySQL中，用户自定义的临时变量用@<变量名>表示。

函数与过程

----存储函数的概念

定义：存储函数也称为自定义函数，以区别于关系数据库管理系统的内置函数。存储函数与存储过程类似，都是持久性存储模块。存储函数的创建与存储过程也类似，不同之处是

1. 存储函数须指定返回的类型。
2. 函数可以直接用其名字调用，并将其嵌入查询中作为参数直接使用，存储过程则不行。

➢ 存储函数创建：

CREATE FUNCTION 函数名（参数1 数据类型，参数2 数据类型，...])

RETURNS <类型>

<过程化SQL块>;

函数与过程

SQL标准支持返回关系作为结果的函数，这种函数称为表函数（table function）。如下所示，该函数定义一个包含某特定系的所有教师的表：

```
CREATE FUNCTION instructor_of (dept_name varchar(20))
        RETURNS TABLE(
                ID varchar(5),
                name varchar(20),
                dept_name varchar(20),
                salary numeric(8,2))
RETURN TABLE
        (SELECT tno, tname , dno, salary
        FROM Dept, Teacher
        WHERE Dept. dname=instructor_of.dept_name AND Dept.dno= Teacher.dno);
```

这种函数可以在如下一个查询中使用：

```
SELECT* FROM table(instructor_of('机电'); //这个查询返回‘机电’系的所有教师的信息
```

函数与过程

函数和过程可以相互转换。

假定我们想要这样一个函数：给定一个系的名字，返回该系的教师数目。写法如下：

```
CREATE FUNCTION dept_count(dept_name varchart(20))
        RETURNS INTEGER
        BEGIN
          DECLARE d_count integer;
          SELECT count(*) INTO d_count FROM Teacher, Dept
                    WHERE Teacher.dno = Dept.dno AND Dept.dname=dept_name;
          return d_count;
        END
```

这种函数可以在如下一个查询中使用：

```
SELECT  dname
FROM Dept
WHERE dept_count(dname) > 12; //获取教师数大于12的所有系的名字
```

函数与过程

dept_count函数也可以写成一个过程：

```
CREATE PROCEDURE dept_count_proc(IN dept_name varchar(20), OUT d_count integer)
        BEGIN
                SELECT count(*) INTO d_count
                FROM Teacher, Dept
                WHERE Teacher.dno = Dept.dno AND Dept.dname=dept_name
        END
```

可以从另一个SQL过程中或者从嵌入式SQL中使用 call语句调用过程：

```
....
BEGIN
DECLARE d_count INT;
CALL dept_count_proc('计算机',d_count); //获取计算机系教师的数量
....
END;
```

数据库的完整性与安全性控制

- 数据库安全性是指DBMS能够保证使DB免受非法、非授权用户的使用、泄漏、更改或破坏的机制和手段
- 数据库安全性管理涉及许多方面
  - 社会法律及伦理方面：私人信息受到保护
  - 公共政策/制度方面：信息公开或非公开制度
  - 安全策略：政府、企业或组织所实施的安全性策略
  - 数据的安全级别：绝密(Top Secret)，机密(Secret)，秘密(Confidential) 和无分类(Unclassified)

数据库的完整性与安全性控制

- DBMS的安全机制：
  - 自主安全性机制：存取控制(Access Control)
    - ✓ 通过权限在用户之间的传递，使用户自主管理数据库安全性
  - 强制安全性机制：
    - ✓ 通过对数据和用户强制分类，使得不同类别用户能够访问不同类 别的数据
  - 推断控制机制（选看）：
    - ✓ 防止通过历史信息，推断出不该被其知道的信息；
    - ✓ 防止通过公开信息(通常是一些聚集信息)推断出私密信息(个体信 息)，通常在一些由个体数据构成的公共数据库中此问题尤为重要
  - 数据加密存储机制（选看）：
    - ✓通过加密、解密保护数据，密钥、加密/解密方法与传输

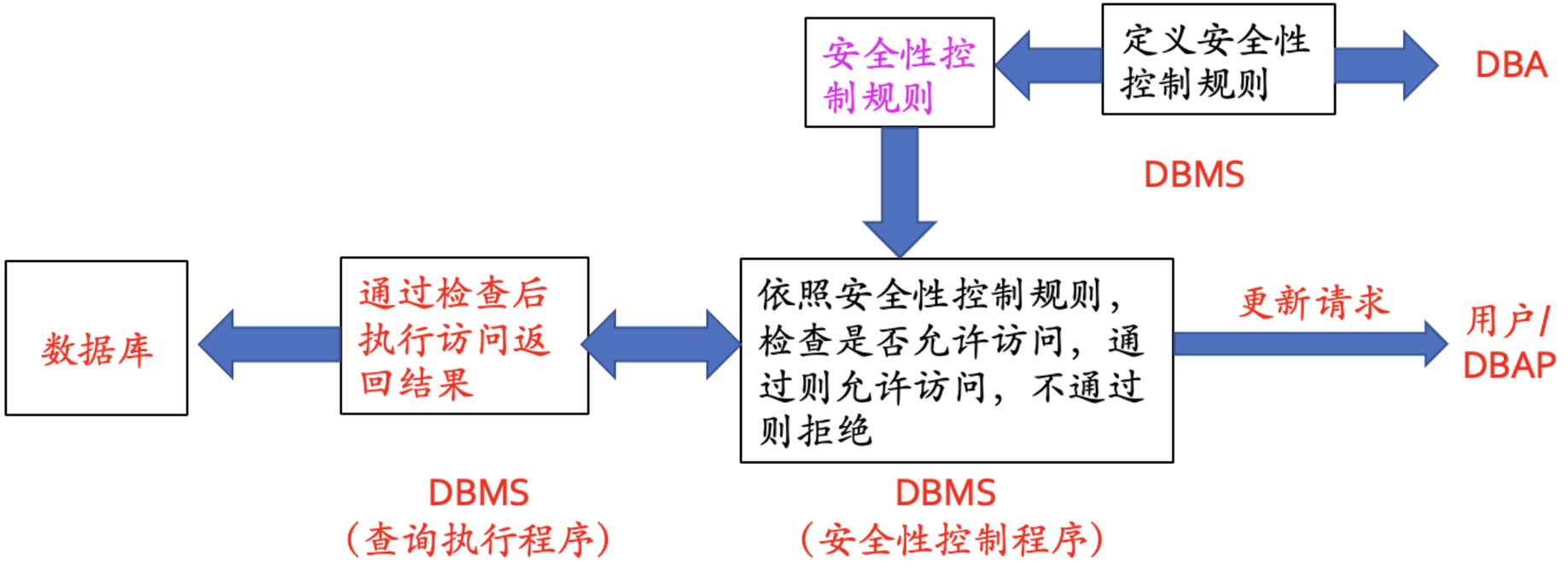
数据库的完整性与安全性控制

- 自主安全性机制
- 通常情况下，自主安全性是通过授权机制来实现的。用户在使用数据库前 必须由DBA处获得一个帐户，并由DBA授予该帐户一定的权限，该帐户的用 户依据其所拥有的权限对数据库进行操作；同时，该帐户用户也可将其所拥有的权利转授给其他的用户(帐户)，由此实现权限在用户之间的传播和控制。
  - 授权者：决定用户权利的人
  - 授权：授予用户访问的权利

数据库的完整性与安全性控制

DBMS如何实现自主安全性

- **DBMS允许用户定义安全规则（DCL）**
- 发生DB访问操作时，DBMS自动按照安全性控制规则进行检查，通过则可访问，否则拒绝访问



数据库的完整性与安全性控制

➢ DBMS将权利和用户(帐户)结合在一起，形成一个访问规则表，依据该规则 表可以实现对数据库的安全性控制

AccessRule ::= (S, O, t, P)

- ✓ S: 请求主体(用户)
- ✓ O: 访问对象
- ✓ t: 访问权利
- ✓ P: 谓词

❑ 用户多时，可以按用户组建立访问规则

❑ 访问对象可大可小(目标粒度Object granularity):属性/字段、记录/元 组、关系、数据库

❑ 权利：CREATE，增删改查等

❑ P：拥有权利需满足条件

数据库的完整性与安全性控制

- 访问权利被分成以下几种
  - (级别1)SELECT　　：读(读DB, Table, Record, Attribute, …)
  - (级别2)MODIFY　　：更新
    - ✓ INSERT　　：插入(插入新元组, …)
    - ✓ UPDATE　　：更新(更新元组中的某些值, …)
    - ✓ DELETE　　：删除(删除元组, …)
  - (级别3)CREATE　　：创建(创建表空间、模式、表、索引、视图等)
    - ✓ CREATE　　：创建
    - ✓ ALTER　　：更新
    - ✓ DROP　　：删除

- 级别高的权利自动包含级别低的权利。如某人拥有更新的权利，它也自动 拥有读的权利。在有些DBMS中，将级别3的权利称为帐户级别的权利，而将 级别1和2称为关系级别的权利。
- 超级用户(DBA) ➔ 帐户级别(程序员用户) ➔ 关系级别(普通用户)

数据库的完整性与安全性控制

- 数据库安全性访问规则示例
- 一个员工管理数据库

Employee(pno, pname, page, psex, psalary, dno, head)

有如下的安全性访问要求：

- 员工管理人员能访问该数据库的所有内容，具有所有权利
- 收发（前台）人员也需访问该数据库以确认某员工是哪一个部门的，便于工作，只能访问基本信息，只能读，其他信息不允许其访问
- 高级领导能访问该数据库的所有内容，但只能读
- 每个员工允许其访问自己的记录，以便查询自己的工资情况，只能读，但不能修改
- 部门领导，能够查询其所领导部门人员的所有情况，只能读，但不能修改

数据库的完整性与安全性控制

Employee(pno, pname, page, psex, psalary, dno, HEAD)

- 员工管理人员能访问该数据库的所有内容，具有所有权利
- 收发（前台）人员也需访问该数据库以确认某员工是哪一个部门的，便于工作，只能访问基本信息，只能读，其他信息不允许其访问
- 高级领导能访问该数据库的所有内容，但只能读
- 每个员工允许其访问自己的记录，以便查询自己的工资情况，只能读，但不能修改
- 部门领导，能够查询其所领导部门人员的所有情况，只能读，但不能修改

S	O	t	P
员工管理人员	Employee	读、删、插、改	
收发员	Employee（pname,dno）	读	
高级领导	Employee	读	
员工	Employee	读	pno=userid
部门领导	Employee	读	Head=userid

数据库的完整性与安全性控制

第1种：存储矩阵

数据对象 / 主体	O_1	O_2	O_3	…	O_n
S_1	All	All	All	….	All
S_2	Read	–	Read	….	–
…	…	…	…	…	….
S_n	–	Read	–	…	–

S	O	t	P
员工管理人员	Employee	读、删、插、改	
收发员	Employee（pname,dno）	读	
高级领导	Employee	读	
员工	Employee	读	pno=userid
部门领导	Employee	读	Head=userid

数据库的完整性与安全性控制

- 第2种：视图
  - 视图是安全性控制的重要手段
  - 通过视图可以限制用户对关系中某些数据项的存取，例如：
    - ✓视图1：CREATE EmpV1 as SELECT * FROM Employee （无意义）
    - ✓ 视图2：CREATE EmpV2 as SELECT pname, dno FROM Employee
  - 通过视图可将数据访问对象与谓词结合起来，限制用户对关系中某些元组的存取，例如：
    - ✓ 视图1： CREATE EmpV3 as SELECT * FROM Employee WHERE pno = :userid
    - ✓视图2： CREATE EmpV4 as SELECT * FROM Employee WHERE head = :userid
  - 用户定义视图后，视图便成为一新的数据对象，参与到存储矩阵或下表进行描述

数据库的完整性与安全性控制

SQL-DCL中关于安全性的命令

- 授权命令

GRANT {ALL PRIVILEGES | privilege {,privilege…}} ON
[TABLE] tablename | viewname
TO {public | user-id {, user-id…}} [WITH
GRANT OPTION];

- user-id，某一个用户帐户，由DBA创建的合法帐户
- public，允许所有有效用户使用授予的权利
- privilege是下面的权利
  - ✓ SELECT | INSERT | UPDATE | DELETE | ALL PRIVILEDGES
- WITH GRANT OPTION选项是允许被授权者传播这些权利

数据库的完整性与安全性控制

- 假定高级领导为Emp0001, 部门领导为Emp0021,员工管理员为Emp2001, 收发员为Emp5001(均为UserId, 也即员工的P#)

```
GRANT All Priviledges ON EmpV1 TO Emp2001;
GRANT SELECT ON EmpV1 TO Emp0001;
GRANT SELECT ON EmpV2 TO Emp5001;
GRANT SELECT ON EmpV3 TO public;
REVOKE SELECT ON employee FROM Emp5002；//废除权限:
```

- 授予视图访问的权利，并不意味着授予基本表访问的权利(两个级别：基本关系级别和视图级别); 授权者授予的权利必须是授权者已经拥有的权利

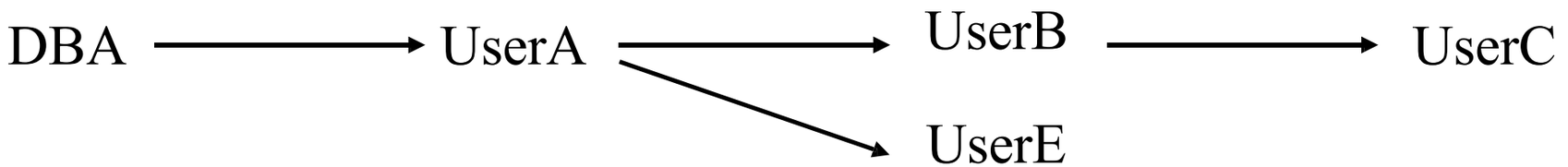
S	O	t	P
员工管理人员	Employee	读、删、插、改	
收发员	Employee（pname,dno）	读	
高级领导	Employee	读	
员工	Employee	读	pno=userid
部门领导	Employee	读	Head=userid

数据库的完整性与安全性控制

- 授权过程示例(续)

```
GRANT SELECT ON Employee TO UserB WITH GRANT OPTION;

GRANT SELECT ON Employee TO UserC WITH GRANT OPTION;
```



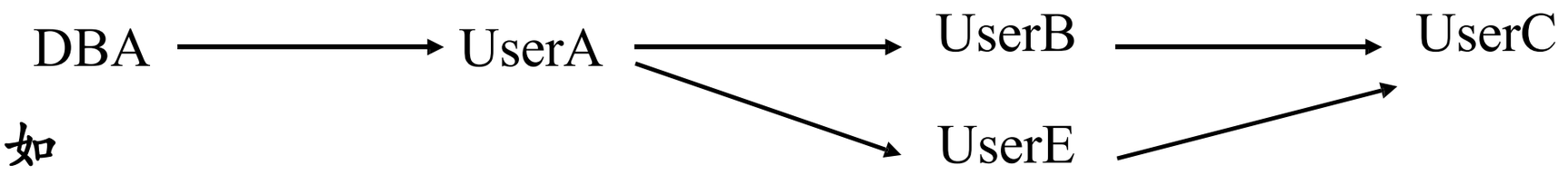
- 注意授权的传播范围问题:
- 传播范围包括两个方面：水平传播数量和垂直传播数量
  - 水平传播数量是授权者的再授权用户数目(树的广度)
  - 垂直传播数量是授权者传播给被授权者，再被传播给另一个被授权 者,...传播的深度(树的深度)
- 有些系统提供了传播范围控制，有些系统并没有提供，SQL标准中也并没有限制。

数据库的完整性与安全性控制

- 收回授权命令

  REVOKE {ALL privilEges | priv {, priv…} } ON tablename | viewname

  FROM {public | user {, user…} };

- 

例如

REVOKE SELECT ON employee FROM UserB;

- 当一个用户的权利被收回时，通过其传播给其他用户的权利也将被收回
- 如果一个用户从多个用户处获得了授权，则当其中某一个用户收回授权时，该用户可能仍保有权利。例如UserC从UserB和UserE处获得了授权，当UserB收回时，其还将保持UserE赋予其的权利。

数据库的完整性与安全性控制

- 强制安全性机制
- 强制安全性通过对数据对象进行安全性分级

  绝密(Top Secret), 机密(Secret),可信(Confidential)和无分类(Unclassified)
- 以及对用户也进行上述的安全性分级，从而强制实现不同级别用户访问不同级别数据的一种机制
- 访问规则如下：
  - 用户S, 不能读数据对象O, 除非Level(S)>=Level(O)
  - 用户S, 不能写数据对象O, 除非Level(S)<=Level(O)。