

# 数据库系统概论



计算机与信息学院  
人工智能学院



➤完整性约束

➤存储过程和触发器（高级SQL）

# 1、完整性约束

保证数据库中的数据正确，有意义

- 1) **定义完整性约束**: PK、FK、UNIQUE、CHECK、NOT NULL、...
- 2) **完整性检查**: 一般在INSERT、UPDATE、DELETE语句执行时开始检查
- 3) **违约处理**: NO ACTION/CASCADE

- 实体完整性

Insert/对PK进行Update时，检查

1)PK是否唯一，不唯一：拒绝

2)主属性是否为空，只要有一个为空：拒绝

- 参照完整性

| 被参照表（例如 <b>Student</b> ） | 参照表（例如 <b>SC</b> ） | 违约处理          |
|--------------------------|--------------------|---------------|
| 可能破坏参照完整性                | 插入元组               | 拒绝            |
| 可能破坏参照完整性                | 修改外码值              | 拒绝            |
| 删除元组                     | 可能破坏参照完整性          | 拒绝/级联删除/设置为空值 |
| 修改主码值                    | 可能破坏参照完整性          | 拒绝/级联修改/设置为空值 |

## [例] 显式说明参照完整性的违约处理示例

CREATE TABLE SC

( Sno CHAR(8),

Cno CHAR(5),

Grade SMALLINT, /\*成绩\*/

Semester CHAR(5), /\*选课学期\*/

Teachingclass CHAR(8), /\*学生选修某一门课所在的教学班\*/

**PRIMARY KEY(Sno,Cno),** /\*在表级定义实体完整性, Sno、Cno都不能取空值\*/

**FOREIGN KEY (Sno) REFERENCES Student(Sno)** /\*在表级定义参照完整性\*/

**ON DELETE CASCADE** /\*当删除Student表中的元组时, 级联删除SC表中相应的元组\*/

**ON UPDATE CASCADE,** /\*当更新Student表中的sno时, 级联更新SC表中相应的元组\*/

**FOREIGN KEY (Cno) REFERENCES Course(Cno)** /\*在表级定义参照完整性\*/

**ON DELETE NO ACTION** /\*当删除Course 表中元组造成与SC表不一致时, 拒绝删除\*/

**ON UPDATE CASCADE** /\*当更新Course表中Cno时, 级联更新SC表中相应的元组的Cno\*/

);

- 自定义完整性: insert, update(属性), 检查

```
CREATE TABLE Student
```

```
(Sno CHAR(9) PRIMARY KEY (Sno),
```

```
, Sname CHAR(8) NOT NULL,
```

```
Ssex CHAR(2) CHECK (Ssex IN ('男' , '女' )) ,
```

```
Sage SMALLINT CHECK (ssage>=0 AND ssage <=200) ,
```

```
Sdept CHAR(20),
```

```
CHECK (Ssex='女' OR Sname NOT LIKE 'Ms.%')
```

```
)
```

- 完整性约束命名

CONSTRAINT <完整性约束名> <完整性约束>

```
CREATE TABLE Student
(Sno CHAR(8)
CONSTRAINT C1 CHECK (Sno BETWEEN '10000000' AND '29999999'),
Sname CHAR(20)
CONSTRAINT C2 NOT NULL,
Sbirthdate Date
CONSTRAINT C3 CHECK (Sbirthdate > '1980-1-1'),
Ssex CHAR(6)
CONSTRAINT C4 CHECK (Ssex IN ( '男','女')),
Smajor VARCHAR(40),
CONSTRAINT StudentKey PRIMARY KEY(Sno)
);
```



- 修改完整性约束

**ALTER TABLE** <表名>

**DROP CONSTRAINT** <约束名>

**ADD CONSTRAINT** <约束名>



**ALTER** TABLE Student

**DROP** CONSTRAINT C1;

ALTER TABLE Student

**ADD** CONSTRAINT C1 CHECK (Sno BETWEEN '900000' AND '999999');

ALTER TABLE Student

**DROP** CONSTRAINT C3;

ALTER TABLE Student

**ADD** CONSTRAINT C3 CHECK (Sbirthdate >'1985-1-1');



- 复杂的约束

- 断言 (**ASSERTION**) : 涉及多个表或聚集操作的复杂完整性约束, 复杂的断言可能会导致系统在检测和维护上的开销较高

**Create assertion** asse\_sc\_db\_num

**Check** (60>=(select count(\*) from sc, course where sc.cno=course.cno

and cname='数据库')

大多数DBMS支持Trigger(触发器): 实现复杂的约束

## 2、过程化SQL

SQL：非过程化的查询语言，缺少流程控制能力，难以实现复杂的业务逻辑控制

- 增加过程控制语句

✓ Oracle： PL/SQL（Procedural Language/SQL）

✓ SQLServer： T(transact)-SQL

.....

存储过程（存储函数）：第八章 数据库编程

触发器

### 3、触发器

➤触发器是与表相关的特殊存储过程，在满足特定条件时，会被触发执行

- ✓ 定义在基本表上
- ✓ 当基本表被修改（增删修）时，会触发定义在其上的触发器
- ✓ 实现较为复杂的完整性约束

1) 在SQL: 1999标准中引入，很多DBMS很早就支持Trigger，语法格式各不相同；

2) Trigger的基本原理对不同实现都适用

**CREATE TRIGGER** <触发器名>

<触发时机><触发事件> ON <表名>

REFERENCING NEW|OLD AS<变量>

FOR EACH {ROW | STATEMENT}

[WHEN <触发条件>]<触发动作体>

触发时机: before、after、for

触发事件: insert、delete、update

**CREATE TRIGGER** Trig-show

AFTER insert ON student

select \* from student

## CREATE TRIGGER <触发器名>

<触发时机><触发事件> ON <表名>

REFERENCING NEW|OLD AS<变量>

FOR EACH {ROW | STATEMENT}

[WHEN <触发条件>]<触发动作体>

- **Row**: 行级触发, 对于触发事件作用的每一行, 执行一次触发动作体
- **Statement**: 语句级触发: 触发动作体执行一次

行级触发:

- ✓ 对于触发事件作用的每一行, 在触发事件发生之前该行称为**OLD**, 之后该行称为**NEW**
- ✓ 可以使用**OLD**和**NEW**来引用触发事件发生前后的元组的值

行级触发:

- ✓ 对于触发事件作用的每一行, 在触发事件发生之前该行称为OLD, 之后该行称为NEW
- ✓ 可以使用OLD和NEW来引用触发事件发生前后的元组的值

例如: Insert一条元组:

Delete一条元组:

Update一条元组



# 例

当对表SC的Grade属性进行修改时，若分数增加了10%，则将此次操作记录到另一个表SC\_U (Sno 、 Cno 、 Oldgrade、 Newgrade )

**CREATE TRIGGER** <触发器名>

<触发时机><触发事件> ON <表名>

REFERENCING NEW|OLD AS<变量>

FOR EACH {ROW | STATEMENT}

[WHEN <触发条件>]<触发动作体>

```
CREATE TRIGGER SC_T
  AFTER UPDATE ON SC
  REFERENCING
    OLD AS OldTuple,
    NEW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.Grade >= 1.1 * OldTuple.Grade)
  BEGIN
    INSERT INTO SC_U (Sno,Cno,OldGrade,NewGrade)
    VALUES(OldTuple.Sno,OldTuple.Cno,OldTuple.Grade,NewTuple.Grade)
  END
```

# 例

例：定义一个BEFORE行级触发器，为教师表Teacher定义完整性规则“教授的工资不得低于4000元，如果低于4000元，自动改为4000元”

**CREATE TRIGGER** <触发器名>

<触发时机><触发事件> ON <表名>

REFERENCING NEW|OLD AS<变量>

FOR EACH {ROW | STATEMENT}

[WHEN <触发条件>]<触发动作体>

```
CREATE TRIGGER Insert_Or_Update_Sal
    BEFORE INSERT OR UPDATE ON Teacher
    REFERENCING NEW ROW AS newtuple
    FOR EACH ROW
    BEGIN
        IF (newtuple.Job='教授') AND (newtuple.Sal < 4000) THEN
            newtuple.Sal :=4000;
        END IF;
    END;
```

**CREATE TRIGGER** <触发器名>

<触发时机><触发事件> ON <表名>

REFERENCING NEW|OLD AS<变量>

**FOR EACH** {ROW | **STATEMENT**}

[WHEN <触发条件>]<触发动作体>

- **语句级触发**：触发动作体将执行一次

Old table: 触发事件执行之前的表

New table: 触发事件执行之后的表

# 例

对表Student的新增操作（批量新增），所增加的学生个数记录到表T2中

```
CREATE TRIGGER Student_Count
```

```
AFTER INSERT ON Student
```

```
REFERENCING NEWTABLE AS Delta
```

```
FOR EACH STATEMENT /*执行完INSERT语句后下面的触发动作体执行一次*/
```

```
BEGIN
```

```
    INSERT INTO StudentInsertLog (Numbers)
```

```
    SELECT COUNT(*) FROM Delta
```

```
END
```

# 执行触发器

触发器的执行，是由触发事件激活，并由数据库服务器自动执行

一个数据表上可能定义了多个触发器，遵循执行顺序：

- (1) 执行该表上的BEFORE触发器
- (2) 激活触发器的SQL语句
- (3) 执行该表上的AFTER触发器

多个before/after触发器：按触发器创建的时间顺序执行/触发器名称的字母排序

删除触发器: Drop trigger <触发器名> ON <表名>



触发器的应用场景：

数据完整性：例如学生表的增删，则对应的班级表人数进行修改；

数据审计：以检查数据变化过程是否合法、规范

.....

- 1) 对数据库中每行数据的修改都会调用触发器。因此可能会导致数据库性能的降低，所以要避免编写过多的触发器
- 2) 数据迁移过程中，触发器所带来的问题

## 两个关系模式

1) student (sno, sname, clsno, ssex)

2) class (clsno, clsname, total)

Clsno: 班级号

Total: 班级总人数

要求:

1) 定义关系模式

2) 定义约束: pk, fk, student (clsno) 非空

3) 定义trigger (行级、after) : student新增一条学生记录时, class中的total+1