

数据库系统概论

An Introduction to Database Systems

第五章 数据库完整性



➡ 数据库的完整性

▢ 数据的正确性

- 是指数据是符合现实世界语义，反映了当前实际状况的

▢ 数据的相容性

- 是指数据库同一对象在不同关系表中的数据是符合逻辑的

➡ 为维护数据库的完整性，**DBMS**必须：

- ▢ 提供定义完整性约束条件的机制
- ▢ 提供完整性检查的方法
- ▢ 违约处理



➡ 数据的完整性和安全性是两个不同概念

☞ 数据的完整性


- 防止数据库中存在不符合语义的数据，也就是防止数据库中存在不正确的数据
- 防范对象：不合语义的、不正确的数据

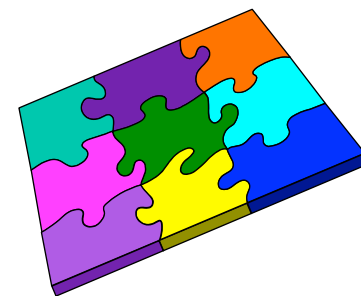
☞ 数据的安全性

- 保护数据库防止恶意的破坏和非法的存取
- 防范对象：非法用户和非法操作

第五章 数据库完整性

本章主要内容

- 实体完整性 
- 参照完整性
- 用户定义的完整性
- 完整性约束命名字句
- 域中的完整性限制*
- 断言
- 触发器
- 小结





5.1 实体完整性

5.1.1

实体完整性定义

5.1.2

实体完整性检查和违约处理

5.1.1 实体完整性定义

➡ 关系模型的实体完整性

- ▮ **CREATE TABLE**中用**PRIMARY KEY**定义

➡ 单属性构成的码有两种说明方法

- ▮ 定义为列级约束条件

- ▮ 定义为表级约束条件

➡ 对多个属性构成的码只有一种说明方法

- ▮ 定义为表级约束条件

5.1.1 实体完整性定义

[例1] 将**Student**表中的**Sno**属性定义为码

```
CREATE TABLE Student  
(Sno CHAR(9) PRIMARY KEY,
```

在列级定义主码

```
Sname CHAR(20) NOT NULL,
```

```
Ssex CHAR(2) ,
```

```
Sage SMALLINT,
```

```
Sdept CHAR(20));
```

在表级定义主码

```
CREATE TABLE Student
```

```
(Sno CHAR(9),
```

```
Sname CHAR(20) NOT NULL,
```

```
Ssex CHAR(2) ,
```

```
Sage SMALLINT,
```

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

```
PRIMARY KEY (Sno));
```

5.1.1 实体完整性定义

[例2] 将SC表中的Sno, Cno属性组定义为码

CREATE TABLE SC

(Sno CHAR(9) NOT NULL,

Cno CHAR(4) NOT NULL,

Grade SMALLINT,

PRIMARY KEY (Sno, Cno) /*只能在表级定

义主码*/

);



5.1 实体完整性

5.1.1

实体完整性定义

5.1.2

实体完整性检查和违约处理

5.1.2 实体完整性检查和违约处理

➡ 插入或对主码列进行更新操作时，**RDBMS**按照实体完整性规则自动进行检查。包括：

- ☐ 1. 检查主码值是否**唯一**，如果不唯一则拒绝插入或修改
- ☐ 2. 检查主码的各个属性是否为**空**，只要有一个为空就拒绝插入或修改

5.1.2 实体完整性检查和违约处理

➡ 检查记录中主码值是否唯一的一种方法是进行全表扫描

待插入记录

Key _i	F2 _i	F3 _i	F4 _i	F5 _i
------------------	-----------------	-----------------	-----------------	-----------------

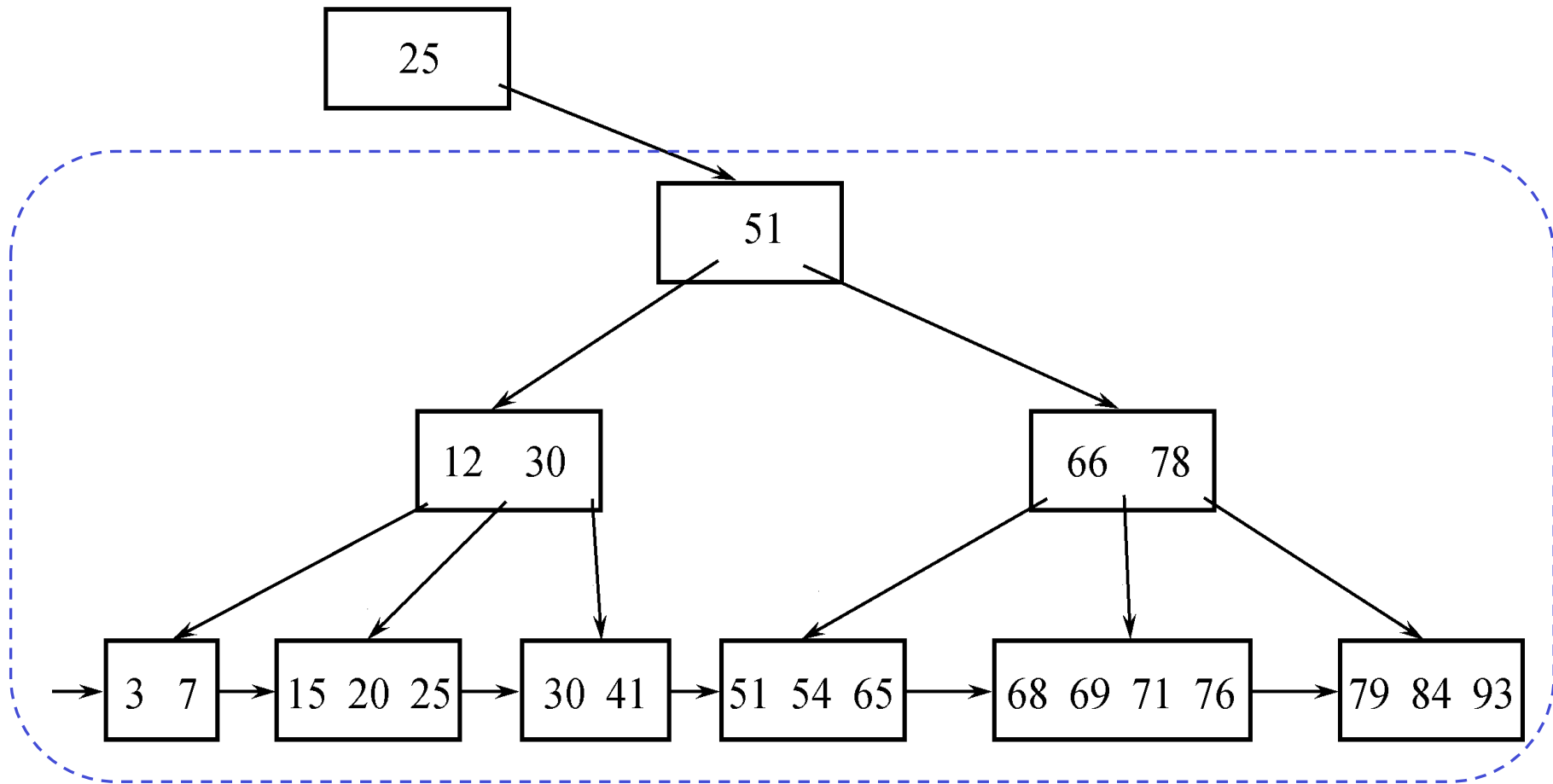
基本表

Key1	F21	F31	F41	F51
Key2	F22	F32	F42	F52
Key3	F23	F33	F43	F53
⋮				

5.1.2 实体完整性检查和违约处理

➡ 索引 -----提高效率

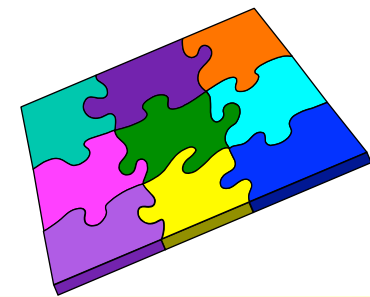
新记录的主码值



第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名字句
- 域中的完整性限制*
- 断言
- 触发器
- 小结





5.2 参照完整性

5.2.1

参照完整性定义

5.2.2

参照完整性检查和违约处理

5.2.1 参照完整性定义

➡ 关系模型的参照完整性定义

☞ 在**CREATE TABLE**中用**FOREIGN KEY**短语定义哪些列为外码

☞ 用**REFERENCES**短语指明这些外码参照哪些表的主码

※ 例如，关系**SC**中一个元组表示一个学生选修的某门课程的成绩，**(Sno, Cno)**是主码。**Sno, Cno**分别参照引用**Student**表的主码和**Course**表的主码

5.2.1 参照完整性定义

[例3] 定义SC中的参照完整性

```
CREATE TABLE SC
(Sno CHAR(9) NOT NULL,
 Cno CHAR(4) NOT NULL,
 Grade SMALLINT,
PRIMARY KEY (Sno, Cno),      /*在表级定
义实体完整性*/
FOREIGN KEY (Sno) REFERENCES Student(Sno),
/*在表级定义参照完整性*/
FOREIGN KEY (Cno) REFERENCES Course(Cno)
/*在表级定义参照完整性*/
);
```




5.2 参照完整性

5.2.1

参照完整性定义

5.2.2

参照完整性检查和违约处理

5.2.2 参照完整性检查和违约处理

可能破坏参照完整性的情况及违约处理

被参照表（例如Student）		参照表（例如SC）	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/级连删除/设置为空值
修改主码值	→	可能破坏参照完整性	拒绝/级连修改/设置为空值

■ 拒绝(**NO ACTION**)执行

■ 级联(**CASCADE**)操作

■ 设置为空值 (**SET-NULL**)

默认策略

- 对于参照完整性，除了应该定义外码，还应定义外码列是否允许空值

5.2.2 参照完整性检查和违约处理

[例4] 显式说明参照完整性的违约处理示例

CREATE TABLE SC

(Sno CHAR(9) NOT NULL,

Cno CHAR(4) NOT NULL,

Grade SMALLINT,

PRIMARY KEY (Sno, Cno) ,

FOREIGN KEY (Sno) REFERENCES Student(Sno)

ON DELETE CASCADE /*级联删除SC表中相应的元组*/

ON UPDATE CASCADE, /*级联更新SC表中相应的元组*/

FOREIGN KEY (Cno) REFERENCES Course(Cno)

ON DELETE NO ACTION

/*当删除course 表中的元组造成了与SC表不一致时拒绝删除*/

ON UPDATE CASCADE

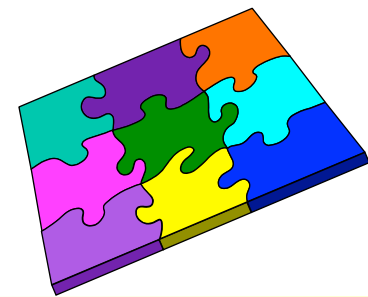
/*当更新course表中的cno时, 级联更新SC表中相应的元组*/

);

第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名字句
- 域中的完整性限制*
- 断言
- 触发器
- 小节



5.3 用户定义的完整性

- ➡ 用户定义的完整性就是针对**某一具体应用**的数据必须满足的语义要求
- ➡ **RDBMS**提供，而不必由应用程序承担

5.3 用户定义的完整性

5.3.1

属性上的约束条件的定义

5.3.2

属性上的约束条件检查和违约处理

5.3.3

元组上的约束条件的定义

5.3.4

元组上的约束条件检查和违约处理

5.3.1 属性上的约束条件的定义

➡ CREATE TABLE时定义

- 列值非空 (**NOT NULL**)

- 列值唯一 (**UNIQUE**)

- 检查列值是否满足一个布尔表达式 (**CHECK**)

5.3.1 属性上的约束条件的定义

1. 不允许取空值

[例5] 在定义**SC**表时，说明**Sno**、**Cno**、**Grade**属性不允许取空值。

CREATE TABLE SC

(Sno CHAR(9) NOT NULL,

Cno CHAR(4) NOT NULL,

Grade SMALLINT NOT NULL,

PRIMARY KEY (Sno, Cno),

/ 如果在表级定义实体完整性，隐含了**Sno**
，**Cno**不允许取空值，则在列级不允许取空值的定义就不必写了 * /*

) ;

5.3.1 属性上的约束条件的定义

2.列值唯一

[例6] 建立部门表**DEPT**，要求部门名称**Dname**列取值唯一，部门编号**Deptno**列为主码

CREATE TABLE DEPT

(Deptno NUMERIC(2),

Dname CHAR(9) UNIQUE NOT NULL, /*要求Dname列值唯一且不能取空值*/

Location CHAR(10),

PRIMARY KEY (Deptno)

);

5.3.1 属性上的约束条件的定义

3. 用CHECK短语指定列值应该满足的条件

[例7] **Student**表的**Ssex**只允许取“男”或“女”。

```
CREATE TABLE Student
(Sno CHAR(9) PRIMARY KEY,
 Sname CHAR(8) NOT NULL,
 Ssex CHAR(2) CHECK (Ssex IN ('男', '女')),
 /*性别属性Ssex只允许取'男'或'女'*/
 Sage SMALLINT,
 Sdept CHAR(20)
);
```

5.3 用户定义的完整性

5.3.1

属性上的约束条件的定义

5.3.2

属性上的约束条件检查和违约处理

5.3.3

元组上的约束条件的定义

5.3.4

元组上的约束条件检查和违约处理

5.3.2 属性上的约束条件检查和违约处理

- ➡ 插入元组或修改属性的值时，**RDBMS**检查属性上的约束条件是否被满足
- ➡ 如果不满足则操作被拒绝执行

5.3 用户定义的完整性

5.3.1

属性上的约束条件的定义

5.3.2

属性上的约束条件检查和违约处理

5.3.3

元组上的约束条件的定义

5.3.4

元组上的约束条件检查和违约处理

5.3.3 元组上的约束条件的定义

- ➡ 在**CREATE TABLE**时可以用**CHECK**短语定义元组上的约束条件，即**元组级的限制**
- ➡ 同属性值限制相比，元组级的限制可以设置**不同属性之间的**取值的相互约束条件

5.3.3 元组上的约束条件的定义

[例9] 当学生的性别是男时，其名字不能以**Ms.**打头。

```
CREATE TABLE Student
(Sno CHAR(9),
 Sname CHAR(8) NOT NULL,
 Ssex CHAR(2) CHECK (Ssex IN ('男', '女')),
 Sage SMALLINT,
 Sdept CHAR(20),
 PRIMARY KEY (Sno),
 CHECK (Ssex='女' OR Sname NOT LIKE 'Ms.%'))
/*定义了元组中Sname和 Ssex两个属性值之间的约束条件*/
```

- ✓ 性别是女性的元组都能通过该项检查，因为**Ssex**='女' 成立；
- ✓ 当性别是男性时，要通过检查则名字一定不能以**Ms.**打头

5.3.3 元组上的约束条件的定义

[例10] 如果是1948年以前制作的中国电影，则该电影不可能是彩色影片。

Movie(title,year,length,inColor,studioName,producerC#)

✧ **(year<1948) → (inColor = 'black-and-white')**

✧ **CHECK(year ≥ 1948 OR inColor = 'black-and-white')**

$$p \rightarrow q \equiv \neg p \vee q$$

CHECK (¬P1 OR P2)

5.3 用户定义的完整性

5.3.1

属性上的约束条件的定义

5.3.2

属性上的约束条件检查和违约处理

5.3.3

元组上的约束条件的定义

5.3.4

元组上的约束条件检查和违约处理

5.3.4 元组上的约束条件检查和违约处理

- ➡ 插入元组或修改属性的值时，**RDBMS**检查元组上的约束条件是否被满足
- ➡ 如果不满足则操作被拒绝执行

第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名子句
- 域中的完整性限制*
- 断言
- 触发器
- 小结



5.4 完整性约束命名子句

➡ 完整性约束命名子句

CONSTRAINT <完整性约束条件名><完整性约束条件>

☞ <完整性约束条件> 包括:

- **PRIMARY KEY**短语
- **FOREIGN KEY**短语
- **CHECK**短语
- **NOT NULL**
- **UNIQUE** 等等

5.4 完整性约束命名子句

[例10] 建立学生登记表**Student**，要求学号在**90000~99999**之间，姓名不能取空值，年龄小于**30**，性别只能是“男”或“女”。

```
CREATE TABLE Student
(Sno NUMERIC(6)
  CONSTRAINT C1 CHECK (Sno BETWEEN 90000 AND 99999),
  Sname CHAR(20)
  CONSTRAINT C2 NOT NULL,
  Sage NUMERIC(3)
  CONSTRAINT C3 CHECK (Sage < 30),
  Ssex CHAR(2)
  CONSTRAINT C4 CHECK (Ssex IN ('男', '女')),
  CONSTRAINT StudentKey PRIMARY KEY(Sno)
);
```

- ✓ 在**Student**表上建立了**5**个约束条件，包括主码约束（命名为**StudentKey**）以及**C1**、**C2**、**C3**、**C4**四个列级约束。

5.4 完整性约束命名子句

[例11]建立教师表**TEACHER**，要求每个教师的应发工资不低于**3000**元。应发工资是工资列**Sal**与扣除项**Deduct**之和。

```
CREATE TABLE TEACHER
(  Eno  NUMERIC(4) PRIMARY KEY    /*在列级定义主码*/
   Ename CHAR(10),
   Job   CHAR(8),
   Sal   NUMERIC(7, 2),
   Deduct NUMERIC(7, 2),
   Deptno NUMERIC(2),
CONSTRAINT TEACHERFKKey FOREIGN KEY (Deptno)
      REFERENCES DEPT (Deptno),
CONSTRAINT C1 CHECK (Sal + Deduct >= 3000)
);
```

5.4 完整性约束命名子句

➡ 使用**ALTER TABLE**语句修改表中的完整性限制

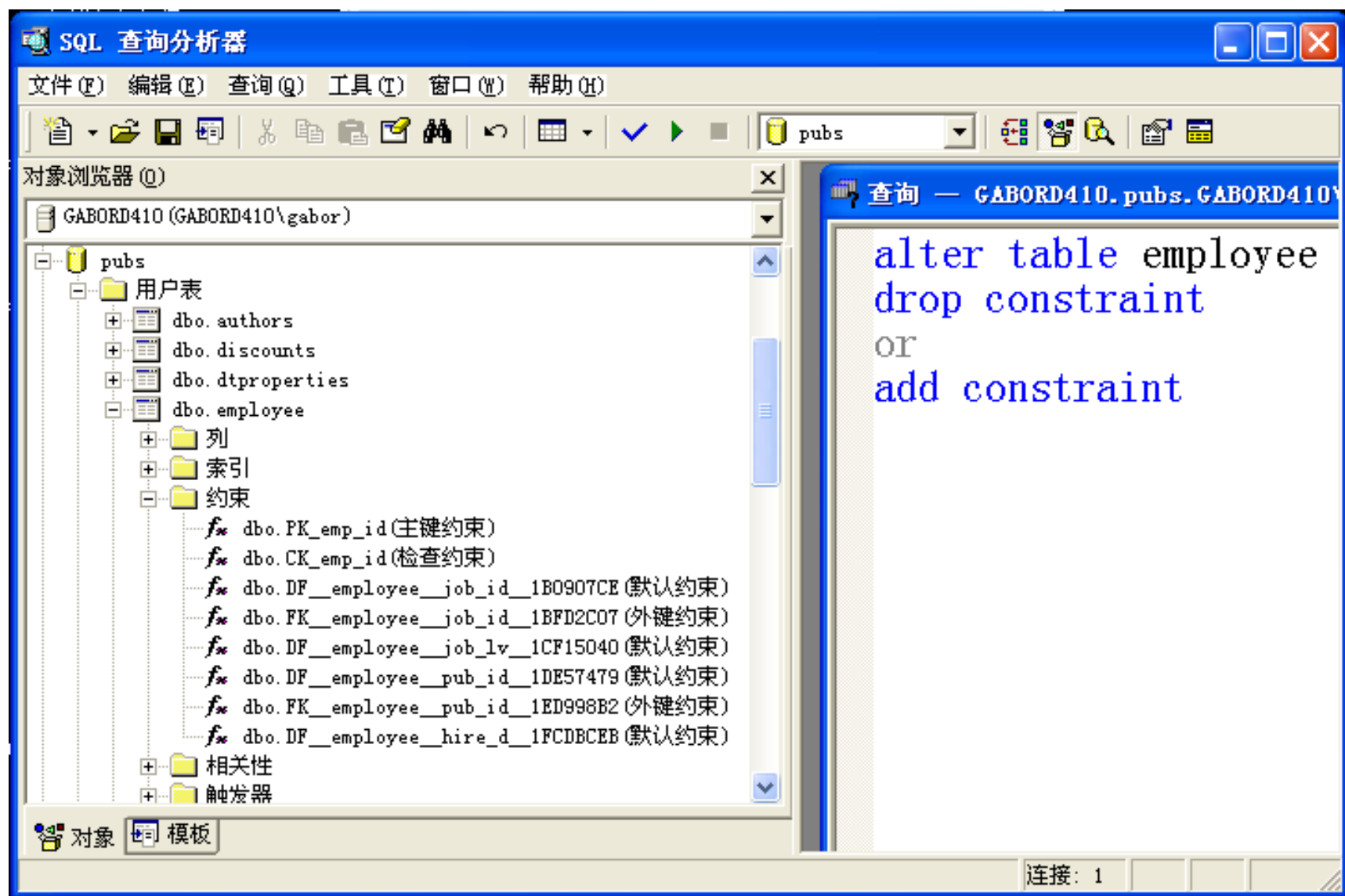
[例13] 修改表**Student**中的约束条件，要求学号改为在**900000~999999**之间，年龄由小于**30**改为小于**40**

■ 可以先删除原来的约束条件，再增加新的约束条件

```
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 (Sage < 40);
```

5.4 完整性约束命名子句



第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名子句
- 域中的完整性限制*
- 断言
- 触发器
- 小结



5.5 域中的完整性限制

➡ **SQL**支持域的概念，并可以用**CREATE DOMAIN**语句建立一个域以及该域应该满足的完整性约束条件。

[例14] 建立一个性别域，并声明性别域的取值范围

```
CREATE DOMAIN GenderDomain CHAR(2)  
CHECK (VALUE IN ('男', '女'));
```

这样 [例10] 中对**Ssex**的说明可以改写为
Ssex GenderDomain

5.5 域中的完整性限制

[例15] 建立一个性别域**GenderDomain**，并对其中的限制命名

```
CREATE DOMAIN GenderDomain CHAR(2)  
CONSTRAINT GD CHECK ( VALUE IN ('男', '女') );
```

[例16] 删除域**GenderDomain**的限制条件**GD**。

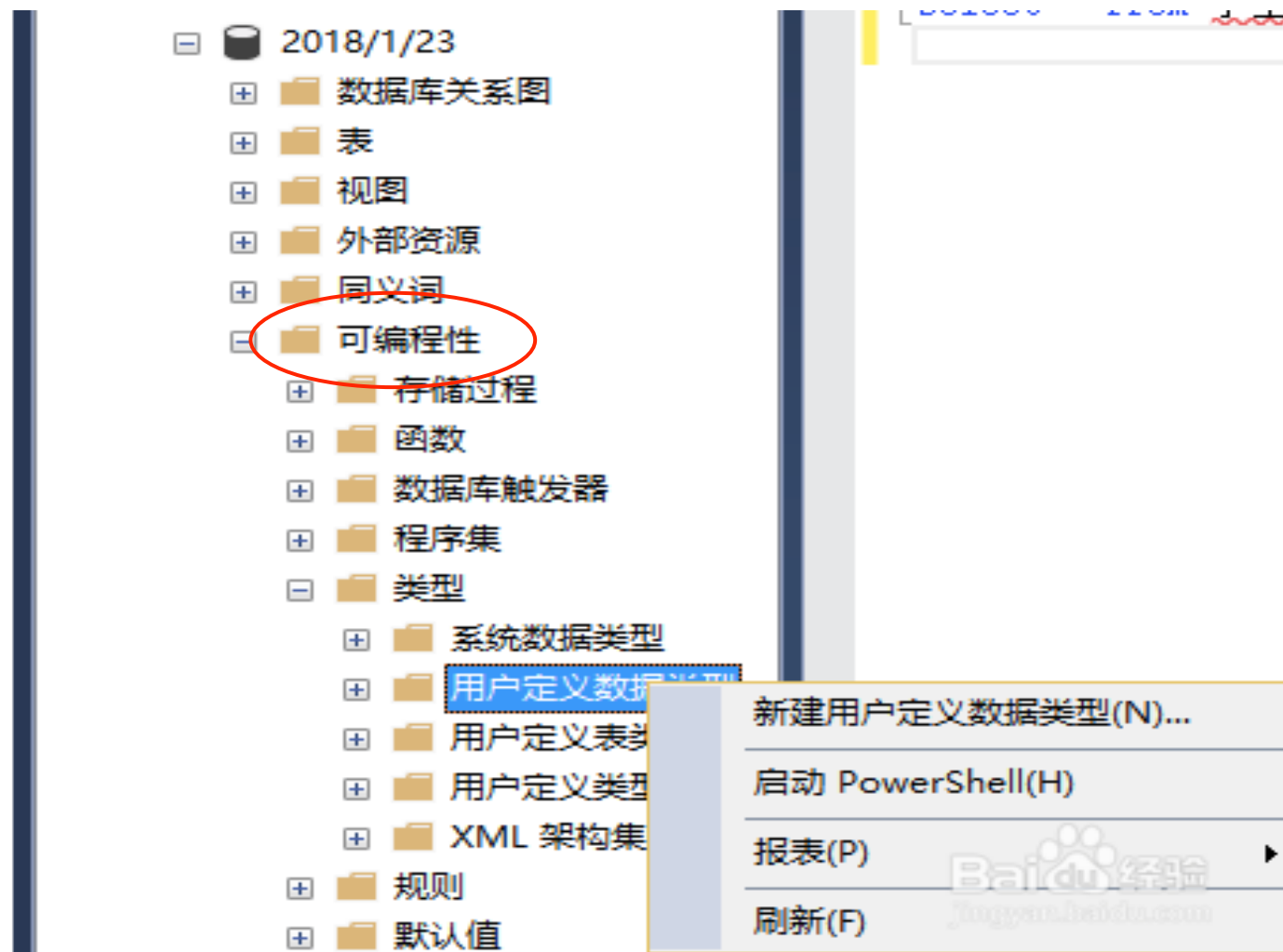
```
ALTER DOMAIN GenderDomain  
DROP CONSTRAINT GD;
```

[例17] 在域**GenderDomain**上增加限制条件**GDD**。

```
ALTER DOMAIN GenderDomain  
ADD CONSTRAINT GDD CHECK (VALUE IN ('1', '0') );
```

✓ 通过 [例16] 和 [例17]，就把性别的取值范围由('男', '女')改为 ('1', '0')

5.5 域中的完整性限制



5.5 域中的完整性限制

- ➔ **SQL Server**中可以用户自定义数据类型，其是系统提供的数据类型的别名。

sp_addtype 创建用户定义的数据类型。

☰ 语法

```
sp_addtype [ @typename = ] type,  
           [ @phystype = ] system_data_type  
           [ , [ @nulltype = ] 'null_type' ]  
           [ , [ @owner = ] 'owner_name' ]
```

- ➔ 在创建了用户数据类型之后，可以在 **CREATE TABLE** 或 **ALTER TABLE** 中使用它，也可以将默认值和规则绑定到用户定义的数据类型。

5.5 域中的完整性限制

➡ 例如：创建不允许空值的用户定义数据类型

下面的示例创建一个名为 **ssn**（社会保险号）的用户定义数据类型，它基于 **SQL Server** 提供的 **varchar** 数据类型。**ssn** 数据类型用于那些保存 **11** 位数字的社会保险号 (**999-99-9999**) 的列。该列不能为 **NULL**。

USE master

EXEC sp_addtype ssn, 'VARCHAR(11)', 'NOT NULL'

➡ 请注意，**varchar(11)** 由单引号引了起来，这是因为它包含了标点符号（圆括号）。

5.5 域中的完整性限制

- ➡ 例如，某公司要计算加拿大元、美元、欧元、日元的数量，则**DBA**可以建立如下数据类型：

```
Exec sp_addtype canadian_dollar 'decimal(11,2)'
```

```
Exec sp_addtype US_dollar 'decimal(11,2)'
```

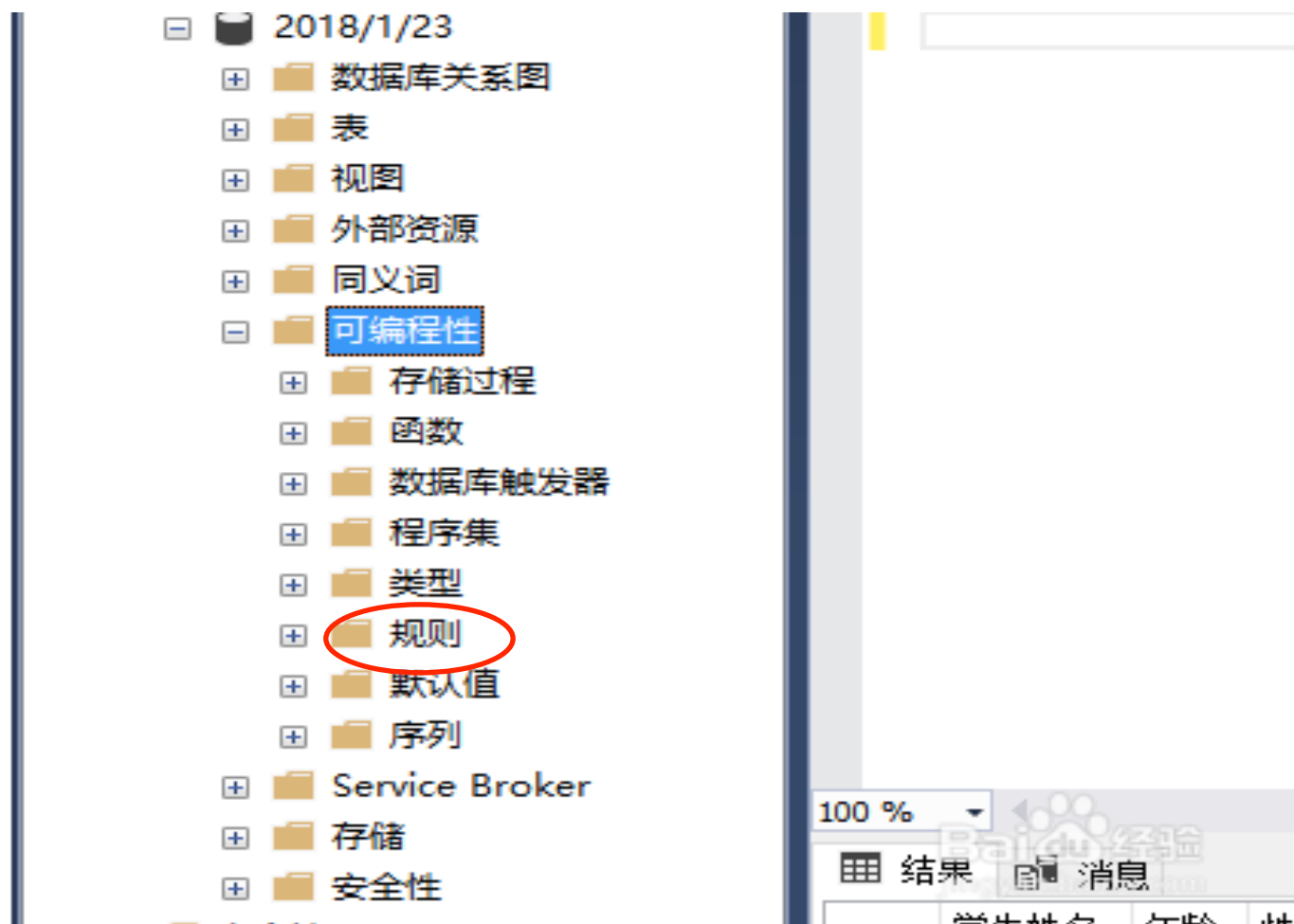
```
Exec sp_addtype Euro 'decimal(11,2)'
```

```
Exec sp_addtype Japanese_yen 'decimal(11,2)'
```

Total = cana_Account + US_Account

- ➡ 将相应列分别指定数据类型后，由于**DBMS**禁止在两种数据类型之间进行没有定义的操作，则上述求和操作会禁止。

5.5 域中的完整性限制



5.5 域中的完整性限制

➡ 规则（rule）约束

📖 Rule 的创建

```
CREATE RULE rule  
AS condition_expression
```

- **condition_expression** 是定义规则的条件。规则可以是 **WHERE** 子句中任何有效的表达式，并且可以包含诸如算术运算符、关系运算符和谓词（如 **IN**、**LIKE**、**BETWEEN**）之类的元素。

- 包含一个变量。每个局部变量的前面都有一个 **@** 符号。该表达式引用通过 **UPDATE** 或 **INSERT** 语句输入的值。

- 在创建规则时，可以使用任何名称或符号表示值，但第一个字符必须是 **@** 符号

📖 Rule 的绑定

```
sp_bindrule [ @rulename = ] 'rule' ,  
[ @objname = ] 'object_name'  
[ , [ @futureonly = ] 'futureonly_flag' ]
```

将规则绑定到列或用户定义的数据类型。

📖 Rule 的松绑

```
sp_unbindrule [ @objname = ] 'object_name'  
[ , [ @futureonly = ] 'futureonly_flag' ]
```

在当前数据库中为列或用户定义数据类型解除规则绑定。

5.5 域中的完整性限制

- ➡ 创建一个范围规则，用以限制插入该规则所绑定的列中的整数范围。

```
CREATE RULE range_rule  
AS @range >= 1000 AND @range < 20000
```

- ➡ 创建一个列表规则，用以将输入到该规则所绑定的列中的实际值限制为只能是该规则中列出的值。

```
CREATE RULE list_rule  
AS @list IN ('1389', '0736', '0877')
```

- ➡ 创建一个遵循这种模式的规则：任意两个字符的后面跟一个连字符和任意多个字符（或没有字符），并以 0 到 9 之间的整数结尾。

```
CREATE RULE pattern_rule  
AS @value LIKE '__- %[0-9]'
```

5.5 域中的完整性限制

➡ 将规则绑定到列

```
EXEC sp_bindrule 'range_rule', 'employees.[bonus]'
```

➡ 将规则绑定到用户定义的数据类型

```
EXEC sp_bindrule 'rule_ssn', 'ssn'
```

➡ 使用 `futureonly_flag`

```
USE master
```

```
EXEC sp_bindrule 'rule_ssn', 'ssn', 'futureonly'
```

因为已指定 *futureonly*，所以不影响类型 *ssn* 的现有列。

- ✓ 在 **CREATE TABLE** 语句中，类型 **ssn** 的列继承 **rule_ssn** 规则。
- ✓ 类型 **ssn** 的现有列也继承 **rule_ssn** 规则，除非为 **futureonly_flag** 指定了 **futureonly**
- ✓ 绑定到列的规则始终优先于绑定到数据类型的规则。

5.5 域中的完整性限制

- ➡ 为列解除规则绑定

EXEC sp_unbindrule 'employees.bonus'

- ➡ 为用户定义数据类型解除规则绑定

EXEC sp_unbindrule 'ssn'

这将为该数据类型的现有列和将来的列解除规则绑定。

- ➡ 使用 *futureonly_flag*

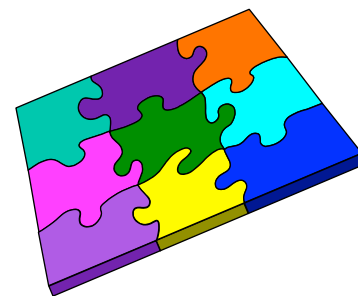
EXEC sp_unbindrule 'ssn', 'futureonly'

为用户定义数据类型 **ssn** 的解除规则绑定，现有的 **ssn** 列不受影响。

第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名子句
- 域中的完整性限制*
- 断言
- 触发器
- 小结



5.6 断言

- ➡ **SQL**中，可以使用 **CREATE ASSERTION**语句，通过声明性断言来指定更具一般性的约束。
 - ➡ 可以定义涉及多个表的或聚集操作的比较复杂的完整性约束。
 - ➡ 断言创建以后，任何对断言中所涉及的关系的操作都会触发关系数据库管理系统对断言的检查，任何使断言不为真值的操作都会被拒绝执行
-

5.6 断言

➡ 创建断言的语句格式

❏ **CREATE ASSERTION**<断言名><**CHECK** 子句>;

- 每个断言都被赋予一个名字，<**CHECK** 子句>中的约束条件与**WHERE**子句的条件表达式类似。

[例18] 限制数据库课程最多**60**名学生选修

CREATE ASSERTION ASSE_SC_DB_NUM

CHECK (60 >= (**select count**(*)

From Course,SC

Where SC.Cno=Course.Cno and

Course.Cname =' 数据库')

);

5.6 断言

[例19]限制每一门课程最多**60**名学生选修

```
CREATE ASSERTION ASSE_SC_CNUM1  
  CHECK (60 >= ALL (SELECT count(*)    注意ALL!!!  
                        FROM SC  
                        GROUP by cno)  
);
```

**/*此断言的谓词，涉及聚集操作count 和分组函数group by
的SQL语句*/**

5.6 断言

[例20]限制每个学期每一门课程最多**60**名学生选修

首先需要修改**SC**表的模式，增加一个“学期 (**TERM**)”属性

ALTER TABLE SC ADD TERM DATE;

然后，定义断言：

```
CREATE ASSERTION ASSE_SC_CNUM2  
CHECK (60 >= ALL (SELECT count(*)  
                FROM SC  
                GROUP by cno, TERM)  
);
```



5.6 断言

➡ 删除断言的语句格式为

❏ **DROP ASSERTION** <断言名>;

❏ 如果断言很复杂，则系统在检测和维护断言的开销较高，这是在使用断言时应该注意的

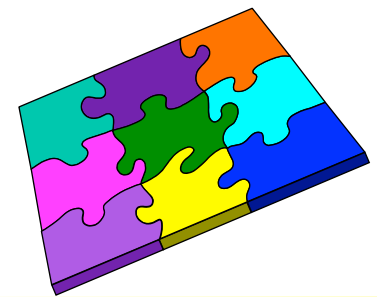
例：

DROP ASSERTION ASSE_SC_DB_NUM

第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名子句
- 域中的完整性限制*
- 断言
- 触发器
- 小结



➡ 触发器（**Trigger**）是用户定义在关系表上的一类由**事件驱动**的特殊过程

- ▣ 触发器保存在数据库服务器中
- ▣ 任何用户对表的增、删、改操作均由服务器自动激活相应的触发器
- ▣ 可以进行更为复杂的检查和操作，具有更精细和更强大的数据控制能力

5.7 触发器

5.6.1

定义触发器

5.6.2

激活触发器

5.6.3

删除触发器

5.7.1 定义触发器

➔ CREATE TRIGGER语法格式

创建者：表的**拥有者**

触发事件：

INSERT
DELETE
UPDATE

CREATE TRIGGER <触发器名>

{**BEFORE** | **AFTER**} <触发事件> **ON** <表名>

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

触发器的目标表

[**WHEN** <触发条件>]

触发器类型

- 行级触发器
- 语句级触发器

<触发动作体>

- 触发动作体可以是一个匿名**PL/SQ L**过程块
- 也可以是对已创建存储过程的调用

触发条件

- 触发条件为真
- 省略**WHEN**触发条件

5.7.1 定义触发器

➡ 例如,假设在**TEACHER**表上创建了一个**AFTER UPDATE**触发器。如果表**TEACHER**有**1000**行,执行如下语句:

UPDATE TEACHER SET Deptno=5;

- ☐ 如果该触发器为语句级触发器,那么执行完该语句后,触发动作只发生一次
- ☐ 如果是行级触发器,触发动作将执行**1000**次

5.7.1 定义触发器

[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下面表中：

SC_U (Sno, Cno, Oldgrade, Newgrade)

其中**Oldgrade**是修改前的分数，**Newgrade**是修改后的分数。

```
CREATE TRIGGER SC_T  
AFTER UPDATE OF Grade ON SC  
REFERENCING
```

```
    OLD row AS OldTuple,  
    NEW row AS NewTuple
```

```
FOR EACH ROW
```

```
WHEN (NewTuple.Grade >= 1.1*OldTuple.Grade)
```

```
    INSERT INTO SC_U (Sno,Cno,OldGrade,NewGrade)
```

```
    VALUES (OldTuple.Sno,OldTuple.Cno,OldTuple.Grade,NewTuple.Gr  
        ade)
```

指出引
用的变量

5.7.1 定义触发器

[例] 将每次对表**Student**的插入操作所增加的学生个数记录到表**StudentInsertLog**中。

CREATE TRIGGER Student_Count

AFTER INSERT ON Student

*/*指明触发器激活的时间是在执行INSERT后*/*

REFERENCING

NEW TABLE AS DELTA

FOR EACH STATEMENT

*/*语句级触发器, 即执行完INSERT语句后下面的触发动作体才执行一次*/*

INSERT INTO StudentInsertLog (Numbers)

SELECT COUNT(*) FROM DELTA

5.7.1 定义触发器

[例18] 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 Insert_Or_Update_Sal
  BEFORE INSERT OR UPDATE ON Teacher
  /*触发事件是插入或更新操作*/
  FOR EACH ROW                                /*行级触发器*/
  AS BEGIN                                    /*定义触发动作体，是PL/SQL过程块*/
    IF (new.Job='教授') AND (new.Sal < 4000) THEN
      new.Sal :=4000;
    END IF;
  END;
```

虚表 -----用于行级触发器
New: 引用UPDATE/INSERT事件之后的新值
Old: 引用UPDATE/DELETE事件之前的旧值

5.7.1 定义触发器

[例19] 定义**AFTER**行级触发器，当教师表**Teacher**的工资发生变化后就自动在工资变化表**Sal_log**中增加一条相应记录

首先建立工资变化表**Sal_log**

```
CREATE TABLE Sal_log
  (Eno      NUMERIC(4) references teacher(eno),
   Sal      NUMERIC(7, 2),
   Username char(10),
   Date     TIMESTAMP
  );
```



5.7.1 定义触发器

[例19] (续)

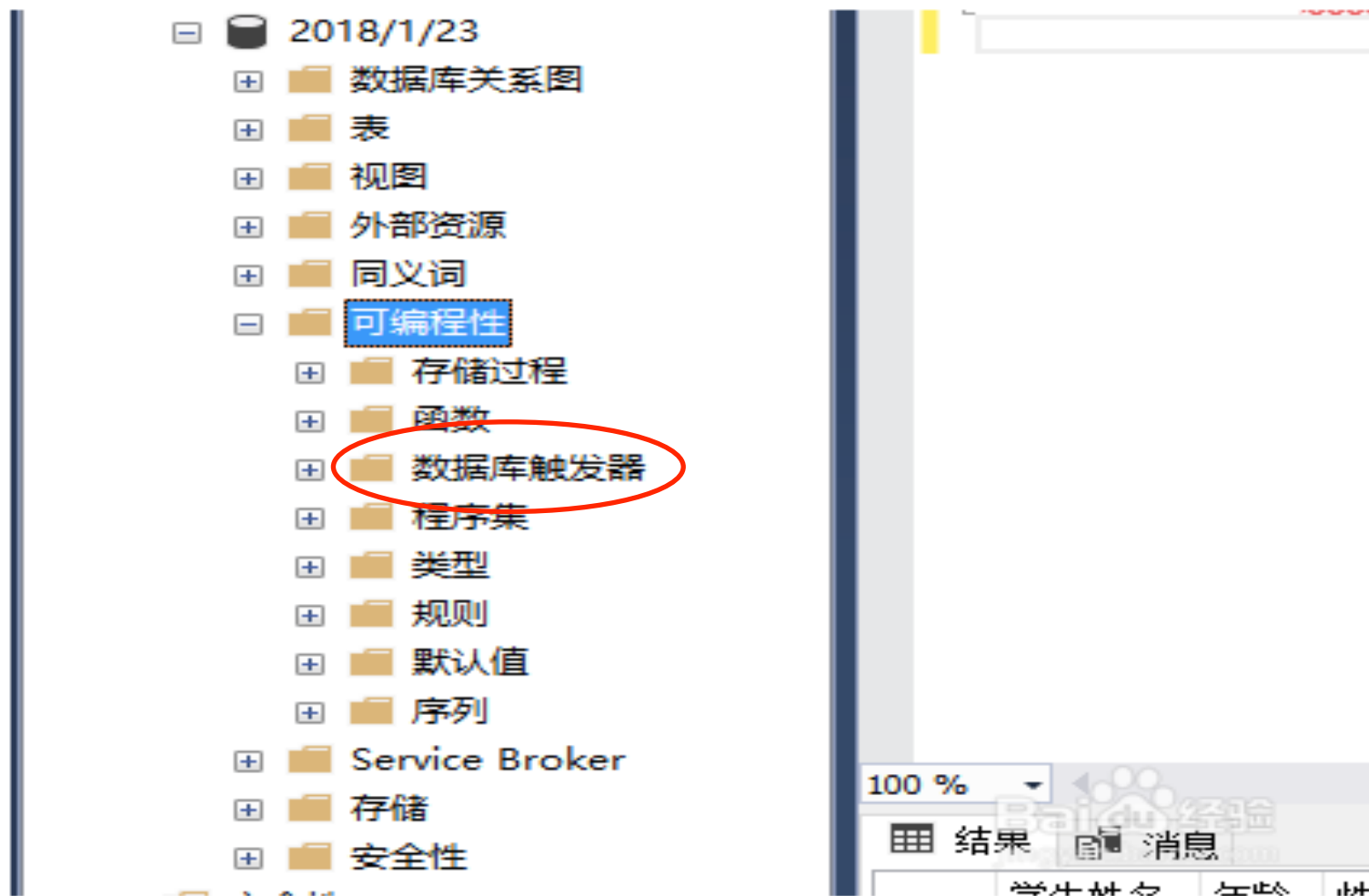
```
CREATE TRIGGER Insert_Sal
  AFTER INSERT ON Teacher          /*触发事件是INSERT*/
  FOR EACH ROW
  AS BEGIN
    INSERT INTO Sal_log VALUES(
      new.Eno, new.Sal, CURRENT_USER,
      CURRENT_TIMESTAMP);
  END;
```

5.7.1 定义触发器

[例19]（续）

```
CREATE TRIGGER Update_Sal
  AFTER UPDATE ON Teacher      /*触发事件是UPDATE */
  FOR EACH ROW
  AS BEGIN
    IF (new.Sal <> old.Sal) THEN INSERT INTO Sal_log VALUES(
      new.Eno, new.Sal, CURRENT_USER
    , CURRENT_TIMESTAMP);
    END IF;
  END;
```


5.7.1 定义触发器






5.7.1 定义触发器

触发器属性


常规

名称(N): 

 名称(N):   employee_insupd (dbo)

文本(T):

```
CREATE TRIGGER [TRIGGER NAME] ON [dbo].[employee]  
FOR INSERT, UPDATE, DELETE  
AS
```





检查语法(C) 删除(D) 另存为模板(S) 3, 3/3

确定 关闭 应用 帮助


触发器属性

常规

名称(N):   employee_insupd (dbo)

文本(T):

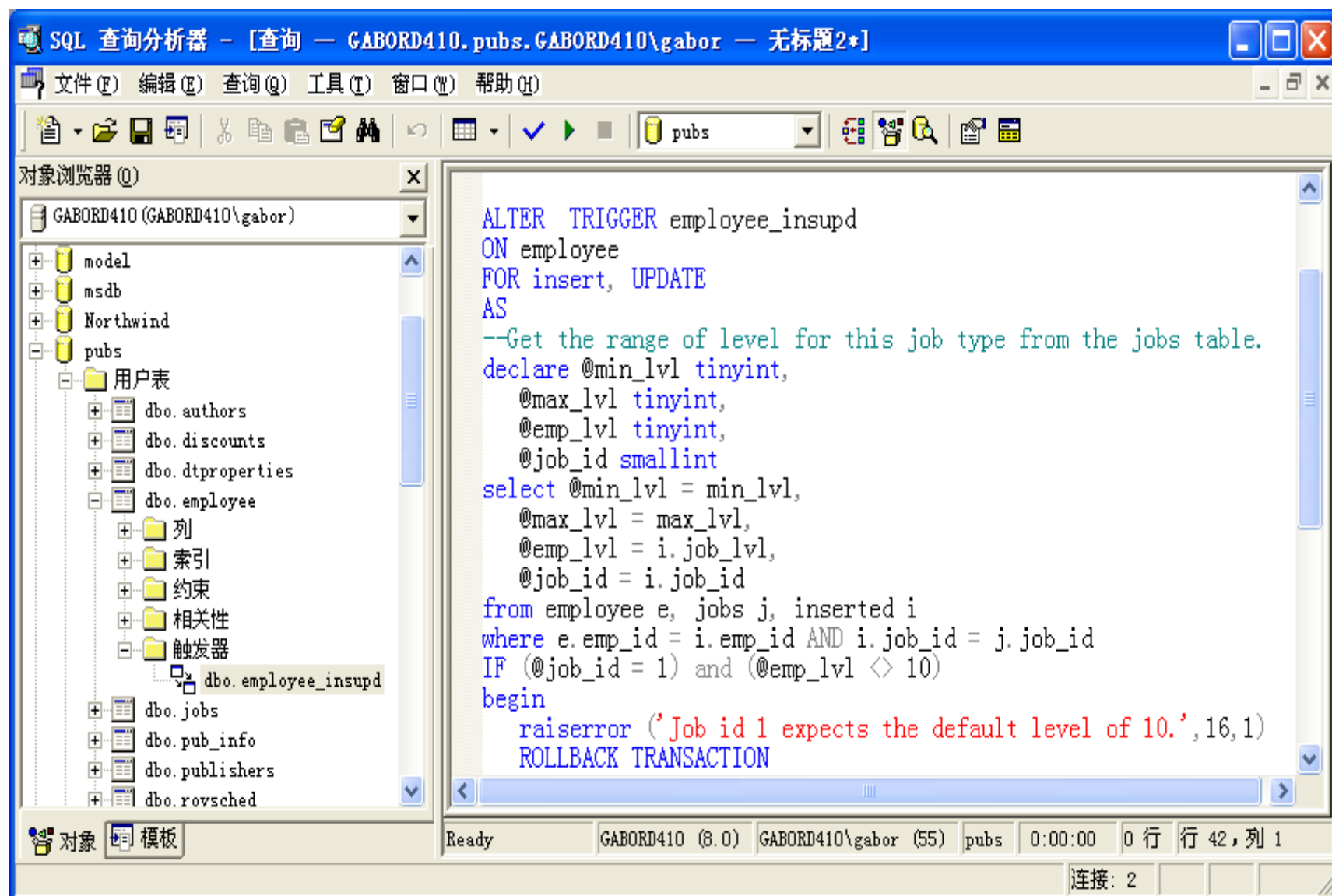
```
CREATE TRIGGER employee_insupd  
ON employee  
FOR insert, UPDATE  
AS  
--Get the range of level for this job type from the jobs table.  
declare @min_lvl tinyint,  
        @max_lvl tinyint,  
        @emp_lvl tinyint,  
        @job_id smallint  
select @min_lvl = min_lvl,  
       @max_lvl = max_lvl,  
       @emp_lvl = i.job_lvl
```



检查语法(C) 删除(D) 另存为模板(S) 3, 3/3

确定 关闭 应用 帮助

5.7.1 定义触发器



5.7.1 定义触发器

- ➔ SQL Server中，允许为任何给定的 INSERT、UPDATE 或 DELETE 语句创建多个触发器。

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
    { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [, ] [ UPDATE ] }
      [ WITH APPEND ]
      [ NOT FOR REPLICATION ]
    AS
      [ { IF UPDATE ( column )
        [ { AND | OR } UPDATE ( column ) ]
        [ ...n ]
        | IF ( COLUMNS_UPDATED ( ) { bitwise_operator }
          updated_bitmask )
          { comparison_operator } column_bitmask [ ...n ]
        } ]
      sql_statement [ ...n ]
    }
}
```

5.7.1 定义触发器

```
CREATE TRIGGER trigger_name  
ON { table | view }
```

```
{
```

```
{ { FOR | AFTER | INSTEAD OF } { [DELETE] [, ] [INSERT]  
  [, ] [UPDATE] }
```

```
AS
```

```
  sql_statement [ ...n ]  
  }
```

```
}
```

AFTER

- 触发器只有在触发 **SQL** 语句中指定的所有操作都已成功执行后才激发。
- 所有的引用级联操作和约束检查也必须成功完成后，才能执行此触发器。
- 如果仅指定 **FOR** 关键字，则 **AFTER** 是默认设置。
- 不能在视图上定义 **AFTER** 触发器。

INSTEAD OF

- 指定执行触发器而不是执行触发 **SQL** 语句，从而替代触发语句的操作。
- 在表或视图上，每个 **INSERT**、**UPDATE** 或 **DELETE** 语句最多可以定义一个 **INSTEAD OF** 触发器。然而，可以在每个具有 **INSTEAD OF** 触发器的视图上定义视图。

5.7.1 定义触发器

➡ 例如：使用带有提醒消息的触发器

当有人试图在 **titles** 表中添加或更改数据时，下例将向客户端显示一条消息。

```
USE pubs
CREATE TRIGGER reminder
ON titles
FOR INSERT, UPDATE
AS
RAISERROR (50009, 16, 10)
```

- 返回用户定义的错误信息并设系统标志，记录发生错误。通过使用 **RAISERROR** 语句，客户端可以从 **sysmessages** 表中检索条目，或者使用用户指定的严重度和状态信息动态地生成一条消息。这条消息在定义后就作为服务器错误信息返回给客户端。
- 消息 **50009** 是 **sysmessages** 中的用户定义消息。



5.7.1 定义触发器

➡ 例如: **instead of** 触发器的应用

```
create trigger f  
on tbl  
instead of delete  
as  
insert into Logs...
```

5.7.1 定义触发器

- ➡ 例如：我们看到许多注册系统在注册后都不能更改用户名，但这多半是由应用程序决定的，如果直接打开数据库表进行更改，同样可以更改其用户名，在触发器中利用回滚就可以巧妙地实现无法更改用户名。

```
create trigger tr  
on 表名  
for update  
as  
    if update(userName)  
        rollback tran
```

5.7.1 定义触发器

➡ **SQL Server**中，使用触发器需要用到两个虚拟表，即：
inserted和**deleted**

➡ 在执行**sql**命令时，两个表分别记录如下内容

▣ **inserted** 保存的是 **insert** 或 **update** 之后所影响的记录形成的表，

▣ **deleted** 保存的是 **delete** 或 **update** 之前所影响的记录形成的表。

Sql命令	deleted	inserted
insert	[不可用]	新插入的记录
update	被更新前的记录	被更新后的记录
delete	被删除的记录	[不可用]

5.7.1 定义触发器

➡ 例如:

```
create trigger tbl_delete
on tbl
for delete
as
    declare @title varchar(200)
    select @title=title from deleted
    insert into Logs(logContent)
values('删除了 title 为: ' + title + '的记录')
```

5.7 触发器

5.6.1

定义触发器

5.6.2

激活触发器

5.6.3

删除触发器

5.7.2 激活触发器

- ➡ 触发器的执行，是由触发事件激活的，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
 - ☞ 同一个表上的多个触发器激活时遵循如下的执行顺序：
 - (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。

5.7.2 激活触发器

[例20] 执行修改某个教师工资的SQL语句，激活上述定义的触发器。

UPDATE Teacher SET Sal=800 WHERE Ename='陈平';

执行顺序是：

- 执行触发器Insert Or Update Sal
- 执行SQL语句 “**UPDATE Teacher SET Sal=800 WHERE Ename='陈平';**”
- 执行触发器Insert Sal;
- 执行触发器Update_Sal

5.7 触发器

5.6.1

定义触发器

5.6.2

激活触发器

5.6.3

删除触发器

5.7.3 删除触发器

➡ 删除触发器的SQL语法:

DROP TRIGGER <触发器名> **ON** <表名>;

➡ 触发器必须是一个已经创建的触发器，并且只能由具有相应权限的用户删除。

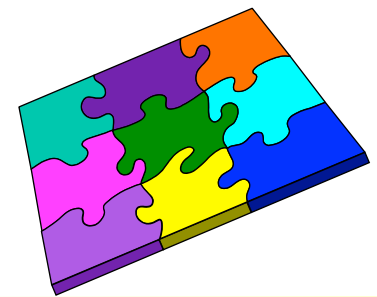
[例21] 删除教师表**Teacher**上的触发器**Insert_Sal**

DROP TRIGGER Insert_Sal ON Teacher;

第五章 数据库完整性

本章主要内容

- 实体完整性
- 参照完整性
- 用户定义的完整性
- 完整性约束命名字句
- 域中的完整性限制*
- 断言
- 触发器
- 小结



- ➡ 数据库的完整性是为了保证数据库中存储的数据是正确的
- ➡ **RDBMS**完整性实现的机制
 - ▣ 完整性约束定义机制
 - ▣ 完整性检查机制
 - ▣ 违背完整性约束条件时**RDBMS**应采取的动作

