# 第五章　数据库完整性

数据是数据库的核心，数据的正确和完整与否将直接影响数据库的使用。

数据的完整性指数据的正确性和相容性。

数据的完整性和安全性是两个不同的概念。

1、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。

2、数据的完整性

- 防止数据库中存在不符合语义的数据（不正确的数据）；
- 防范对象：不合语义的、不正确的数据。

# 本章主要内容

- 数据完整性概述
- 索引
- 在SQL Server 2008中实现数据完整性
- 其他相关知识

# 一、数据库完整性概述

1、为了维护数据的完整性，DBMS必须能够：

（1）提供定义完整性约束条件的机制

完整性约束条件 = 完整性规则，由SQL的DDL语句来实现；

（2）提供完整性检查的机制

检查数据是否满足完整性约束条件的机制；

（3）违约处理

若用户操作违反完整性约束条件，采取一定的动作，如拒绝执行（NO ACTION）、级联执行或其他操作；

2、数据库完整性约束机制分类

完整性约束条件作用的对象有**关系**、**元组**、**列**三种；可分为：

（1）**实体完整性**：实体完整性是为了保证表中的数据唯一；

（2）**参照完整性**：参照完整性用于确保相关联的表间的数据应保持一致，避免因一个表的记录修改，造成另一个表的内容变为无效的．一般来说，参照完整性是通过外键和主键来维护的；

（3）**域完整性**：域完整性可以保证数据的取值在有效的范围内；

（4）**用户定义完整性**：由用户自行定义的，不同于前面3种的完整性，也可以说一种强制数据定义；

（1）实体完整性

实体（Entity）：是数据逻辑模型中的一个概念。现实世界是一组实体的基本对象和这些对象的联系（Relation）构成的。在数据库中，我们可以理解一条记录就是一个实体。

实体完整性：在现实世界中，任何一个实体都有区别于其他实体的特征，即在数据库中，所有的记录都应该有惟一的标识，这就是实体完整性的含义（规则2.1：实体完整性规则）。

常见的实体完整性机制包括：

主键（Primary Key）；惟一码（Unique）；

（2）参照完整性

参照完整性是指在两个表的主键和外键之间数据的完整性，其含义包括：

- 参照完整性保证被参照表和参照表之间数据的一致性；
- 可以防止数据丢失或者无意义的数据；
- 可以禁止在从表中插入被参照表中不存在的关键字的记录。

参照完整性的常见实现机制包括：

外键（Foreign Key）、检查（Check）、触发器（Trigger）；存储过程（Stored Procedure）

（3）域完整性

域：指列（字段），所以域完整性就是指列的完整性；

域完整性要求列（字段）的数据具有正确的数据类型、格式和有效的数值范围。

域完整性的常见实现机制包括：

- 默认值（Default）
- 检查（Check）
- 数据类型（Data type）
- 规则（Rule）

（4）用户自定义完整性

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；可直接由RDBMS提供，而不必由应用程序承担；系统将实现数据完整性的要求直接定义在表上或列上。

常见的实现机制包括：

规则（Rule）、创建数据表时的所有约束（Constraint）、触发器（Trigger）、存储过程（Stored Produre）；

# 二、索引

## 1、索引的分类

索引：根据表中一列或若干列按照一定顺序建立的列值与记录行之间的对应关系表。

作用：

- 快速存取数据；
- 保证数据记录的唯一性；
- 实现表与表之间的参照完整性；
- 在使用ORDER BY、GROUP BY子句进行数据检索时，利用索引可以减少排序和分组的时间。

SQL Server 2008支持在表中任何列（包括计算列）上定义索引，按索引的组织方式分为聚集索引和非聚集索引。

（1）聚集索引（clustered index）

将数据行的键值在表内排序并存储对应的数据记录，使得数据表物理顺序与索引顺序一致。（索引中键值的逻辑顺序决定了表中相应行的物理顺序。）

聚集索引确定表中数据的物理顺序，一个表只能包含一个聚集索引，该索引可以包含多个列（组合索引）。

（2）非聚集索引（nonclustered index）

非聚集索引完全独立于数据行的结构。数据存储在一个位置，索引存储在另一位置，索引带有指针指向数据的存储位置。

非聚集索引中的项目按索引值的顺序存储，表中的信息按另一种顺序存储。索引的逻辑顺序与磁盘上的物理存储顺序不同。

一个表中可以有一个或多个非聚集索引。

聚集索引与非聚集索引的区别

聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。

- 聚集索引表记录的排列顺序与索引的排列顺序一致，查询速度快，对表进行修改会引起数据顺序的重组，速度较慢。
- 非聚集索引指定了表中记录的逻辑顺序，记录的物理顺序和索引的顺序不一致，添加记录不会引起数据顺序的重组。
- SQL Server默认在主键上建立聚集索引。

## 如何选择聚集索引与非聚集索引

| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| --- | --- | --- |
| 列经常被分组排序 | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 一个或极少不同值 | 不应 | 不应 |
| 小数目的不同值 | 应 | 不应 |
| 大数目的不同值 | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
| 频繁修改索引列 | 不应 | 应 |

## 2、创建索引

```
CREATE [ UNIQUE ]                                  /*指定索引是否唯一*/
[ CLUSTERED | NONCLUSTERED ]                       /*索引的组织方式*/
INDEX index_name                                   /*索引名称*/
ON <object> ( column [ ASC | DESC ] [ ,...n ] )    /*索引定义的依据*/
```

EG：为KCB表的课程名列创建索引。

```
CREATE INDEX kc_name_ind
ON KCB(Cname);
```

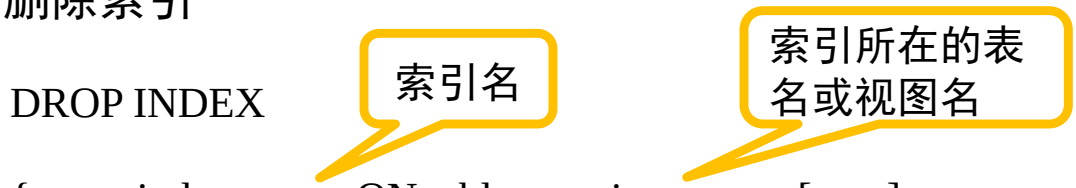
EG：根据KCB表的课程号列创建唯一聚集索引。（先将KCB的主键删除）

```
CREATE UNIQUE CLUSTERED INDEX kc_id_ind
ON KCB(C_ID);
```

EG：根据CJB表的学号和课程号创建复合索引。

```
CREATE INDEX CJB_ind
ON CJB(Stu_ID,C_ID);
```

3、删除索引




```
DROP INDEX
{       index_name ON table_or_view_name […n ]
        | table_or_view_name.index_name […n]
}
```

EG：删除表KCB的一个索引名为kc_name_ind的索引。

```
IF EXISTS(SELECT name FROM sysindexes WHERE name='kc_name_ind')
DROP INDEX KCB.kc_name_ind;
```

EG：删除CJB中名为CJB_ind的索引。

```
DROP INDEX CJB.CJB_ind;
```

# 三、在SQL Server 2008中实现数据完整性

SQL Server 2008提供了完善的数据完整性机制，主要包括约束、默认值和规则3类；

- 创建及管理约束及约束对象
- 默认值约束及默认值对象
- 创建及管理规则对象

1、创建及管理约束及约束对象

- 主键Primary key约束
- 惟一值Unique约束
- 外键Foreign key约束
- 检查Check约束

（1）创建及删除主键约束及惟一值约束

- 单属性构成的码的两种说明方法：

  定义为列级约束条件，定义为表级约束条件

- 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

创建约束方式：

- 创建表时同时创建primary key约束或unique约束
- 修改表时同时创建primary key约束或unique约束

① 创建表时同时创建primary key约束或unique约束

CREATE TABLE table_name

(column_name data_type (NULL| NOT NULL)

[ [CONSTRAINT constraint_name]

{ PRIMARY KEY | unique}

[, ...n]

EG：创建XSB1表，并对学号字段创建PRIMARY KEY约束，对姓名字段定义UNIQUE约束

```
CREATE TABLE PXSCJ..XSB1
(
Stu_ID char(6) not null PRIMARY KEY,
Sname char(8) not null UNIQUE,
Ssex bit null ,
Sdate date null,
Major char(12) null,
Tcredit int null,
Remark varchar(500) null
);
```

```
CREATE TABLE PXSCJ..XSB1
(
Stu_ID char(6) not null,
Sname char(8) not null UNIQUE,
Ssex bit null ,
Sdate date null,
Major char(12) null,
Tcredit int null,
Remark varchar(500) null,
PRIMARY KEY(Stu_ID)
);
```

EG：创建XSB2表，并对学号字段创建PRIMARY KEY约束，对姓名、出生时间字段定义UNIQUE约束。

```
CREATE TABLE PXSCJ..XSB1
(
Stu_ID char(6) not null,
Sname char(8) not null UNIQUE,
Ssex bit null ,
Sdate date null UNIQUE,
Major char(12) null,
Tcredit int null,
Remark varchar(500) null,
PRIMARY KEY(Stu_ID)
);
```

```
CREATE TABLE PXSCJ..XSB4
(
Stu_ID char(6) not null,
Sname char(8) not null,
Ssex bit null ,
Sdate date null,
Major char(12) null,
Tcredit int null,
Remark varchar(500) null,
PRIMARY KEY(Stu_ID),
UNIQUE(Sname,Sdate)
);
```

EG：创建CJB1表，并对学号、课程号字段创建PRIMARY KEY约束

```
CREATE TABLE PXSCJ..CJB1
(
Stu_ID char(6) not null,
C_ID char(3) not null,
Grade int null
primary key(Stu_ID,C_ID)
);
```

实体完整性检查：

插入或对主码列进行更新操作时，RDBMS（关系数据库管理系统）按照实体完整性规则自动进行检查。

- 检查主码值是否唯一，如果不唯一则拒绝插入或修改；
- 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；
- 检查方法：全表扫描、索引。

② 通过修改表时同时创建primary key约束或unique约束

```
ALTER TABLE table_name
        ADD [CONSTRAINT constraint_name]    /*为约束命名*/
        primary key | unique
        CLUSTERED | NONCLUSTERED          /*定义约束的索引类型*/
        (column [,...n])
```

定义约束

约束命名，若省略则系统自动创建

EG：向XSB1中添加一个“身份证号码”字段，对该字段定义UNIQUE约束，对出生时间字段定义UNIQUE约束。

```
ALTER TABLE XSB1
ADD 身份证号码 char(20)
CONSTRAINT SFZ_UK UNIQUE (身份证号码);
ALTER TABLE XSB1
ADD CONSTRAINT CJSJ_UK UNIQUE (Sdate);
```

③ 删除PRIMARY KEY约束和UNIQUE约束

使用ALTER TABLE的DROP子句。

ALTER TABLE table_name

DROP CONSTRAINT constraint_name [,…n]

EG：删除XSB1中的SFZ_UK,CJSJ_UK约束。

```
ALTER TABLE XSB1
DROP CONSTRAINT SFZ_UK,CJSJ_UK
```

一个表只能有一个 PRIMARY KEY 约束，并且 PRIMARY KEY 约束中的列不能接受空值。

如果对多列定义了 PRIMARY KEY 约束，则一列中的值可能会重复，但来自 PRIMARY KEY 约束定义中所有列的任何值组合必须唯一。

可以对一个表定义多个 UNIQUE 约束，但只能定义一个 PRIMARY KEY 约束。

若要修改 PRIMARY KEY 、UNIQUE 约束，必须首先删除现有的 UNIQUE 约束，然后用新定义重新创建。

（2）创建及管理外键约束

对于两个相关联的表（主表与从表）进行数据插入和删除时，通过参照完整性保证他们之间数据的一致性。

利用FOREIGN KEY 定义从表的外码，PRIMARY KEY 或者 UNIQUE定义主表中的主码或惟一码（不允许为空），可实现主表与从表之间的参照完整性。

FOREIGN KEY 约束并不仅仅可以与另一表的 PRIMARY KEY 约束相链接，它还可以定义为引用另一表的 UNIQUE 约束。

定义表间参照关系：先定义主表主码（或惟一码），再对从表定义外码约束。

## 可能破坏参照完整性的情况及违约处理

| 被参照表 | 参照表 | 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 |

当主码外码发生不一致时，系统可采取以下策略：

- 拒绝(NO ACTION)执行（默认策略）
- 级联(CASCADE)操作
- 设置为空值（SET-NULL）：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

## ① 创建表时同时定义FOREIGN KEY约束

```
CREATE TABLE    table_name
( <column_definition>
    [ CONSTRAINT constraint_name ]
    [FOREIGN KEY][ ( column [ ,...n ] ) ]
    REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ NOT FOR REPLICATION ]
)
```

FOREIGN KEY 约束引用的表的名称

FOREIGN KEY 约束所引用的表中的一列或多列

指定参照动作

不强制执行约束

NO ACTION：数据库引擎将引发错误，不采取动作；

CASCADE：如果在父表中删除或更新了一行，则将在引用表中删除或更新相应的行；

SET NULL：如果删除或更新了父表中的相应行，则将构成外键的所有值设置为 NULL。若要执行此约束，外键列必须可为空值；

SET DEFAULT：如果删除或更新了父表中的相应行，则会将构成外键的所有值都设置为其默认值。若要执行此约束，所有外键列都必须有默认定义。

EG：创建stu表，要求stu表中所有的学生学号都必须出现在XSB表中。

```
CREATE TABLE stu
(Stu_ID char(6) NOT NULL FOREIGN KEY(Stu_ID) REFERENCES XSB(Stu_ID),
 xm char(8) NOT NULL,
 cssj datetime NULL
);
```

EG：创建point表，要求表中所有的学号、课程号组合都必须出现在CJB表中。

```
CREATE TABLE point
(Stu_ID char(6) NOT NULL,
 C_ID char(3) NOT NULL,
 cj int NULL,
 FOREIGN KEY(Stu_ID,C_ID) REFERENCES CJB(Stu_ID,C_ID)
ON DELETE NO ACTION);
```

```
CREATE TABLE point
(Stu_ID char(6) NOT NULL FOREIGN
KEY(Stu_ID) REFERENCES CJB(Stu_ID),
 C_ID char(3) NOT NULL FOREIGN KEY(C_ID)
REFERENCES CJB(C_ID),
 cj int NULL
);
```

② 通过修改表定义外键约束

使用ALTER TABLE语句的ADD子句可以定义FOREIGN KEY约束，语法格式与定义PRIMARY KEY、UNIQUE约束类似。

EG：若KCB为主表，课程号为主键；CJB为从表，请将CJB的课程号定义为外键。

```
ALTER TABLE CJB
ADD CONSTRAINT kc_foreign
FOREIGN KEY (C_ID) REFERENCES KCB(C_ID);
```

③ 删除表间的参照关系

删除表间的参照关系，是指删除从表的FOREIGN KEY约束。

语法格式与删除PRIMARY KEY、UNIQUE约束的格式类似。

EG：删除对CJB表的课程号字段定义的外键约束。

```
ALTER TABLE CJB
DROP CONSTRAINT kc_foreign;
```

（3）创建及删除检查约束

通过限制列可接受的值，强制域的完整性。

可以将多个 CHECK 约束应用于单个列，也可以通过在表级创建 CHECK 约束，将一个 CHECK 约束应用于多个列。

CHECK约束在执行INSERT语句或UPDATE时起作用，当用户在向表中插入或更新数据时，由SQL Server检查新行中带有该约束的列值必须满足约束条件。

必须先删除现有的 CHECK 约束，然后使用新定义重新创建，才能修改 CHECK 约束。

① 创建数据表时创建CHECK约束

在创建表时使用CHECK约束表达式定义CHECK约束。

CHECK ( logical_expression )

EG：创建student表，只包括学号和性别两列，性别只能包含男或女。

```
CREATE TABLE student
( 学号 char(6) NOT NULL,
  性别 char(1) NOT NULL CHECK(性别 IN('男','女'))
);
```

EG：创建student1表，只包括学号和出生日期两列，出生日期必须大于1980年1月1日，并命名CHECK约束。

```
CREATE TABLE student1
( 学号 char(6) NOT NULL,
  出生日期 datetime NOT NULL,
  CONSTRAINT DF_student1_cjsj CHECK(出生日期>'1980-1-1')
);
```

EG：创建表student2，有学号、最好成绩和平均成绩三列，要求最好成绩必须不高于100分，并且大于平均成绩。

```
CREATE TABLE student2
( 学号 char(6) NOT NULL,
  最好成绩 INT NOT NULL,
  平均成绩 INT NOT NULL,
  CHECK(最好成绩<=100),CHECK(最好成绩>平均成绩)
 );
```

或者：

```
CREATE TABLE student5
( 学号 char(6) NOT NULL,
  最好成绩 INT NOT NULL CHECK(最好成绩<=100),
  平均成绩 INT NOT NULL ,
  CHECK(最好成绩>平均成绩)
 );
```

② 修改表时创建CHECK约束

ALTER TABLE table_name

[ WITH { CHECK | NOCHECK } ] ADD

[<column_definition>]

[ CONSTRAINT constraint_name ] CHECK(logical_expression)

是否用新添加的约束进行验证，默认为WITH CHECK

EG：修改CJB，增加成绩字段的CHECK约束。

```
ALTER TABLE CJB
ADD CONSTRAINT cj_constraint CHECK(Grade>=0 AND Grade<=100);
```

③ 删除CHECK约束

使用ALTER TABLE语句的DROP子句删除CHECK约束。

ALTER TABLE  table_name

DROP CONSTRAINT check_name

EG：删除CJB表成绩字段的CHECK约束。

```
ALTER TABLE CJB
DROP CONSTRAINT cj_constraint;
```

# 四、其他相关知识

1、存储过程

2、触发器

1、存储过程

存储过程是数据库对象之一（数据库的子程序），是一组为了完成特定功能的SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

使用存储过程的优点：

- 只需编译一次，以后即可多次执行，提高了系统的性能；
- 在服务器端运行，执行速度快；
- 用户可以被授予权限执行存储过程，而不必直接对存储过程中引用的对象具有权限，确保了数据库的安全；
- 自动完成需预先执行的任务，方便了用户的使用。

SQL Server中的存储过程分为：

- 系统存储过程
  - 由SQL Server提供的存储过程，可以作为命令执行；
  - 定义在系统数据库master中，以“sp_”作为前缀；
  - 通过系统存储过程，可以实现许多管理性或信息性的活动（如了解数据库对象、数据库信息）。
- 扩展存储过程

在SQL Server环境之外，使用编程语言（如C语言）创建的外部例程所形成的动态链接库，加载到SQL Server系统中，按照使用系统存储过程的方法执行。

- 用户存储过程

在SQL Server2008中，用户定义的T-SQL存储过程中包含一组T-SQL语句集合，可以接受和返回用户提供的参数。

## （1）创建存储过程

存储过程名

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
```

指定参数名称

```
                                                          /*定义过程名*/
    [ { @parameter [ type_schema_name. ] data_type }      /*定义参数的类型*/
        [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]   /*定义参数的属性*/
    ] [ ,...n ]
```

指定作为输出参数支持的结果集

指示参数是输出参数

指示不能在过程的主体中更新或修改参数。

```
[ WITH <procedure_option> [ ,...n ] ]                 /*定义存储过程的处理方式*/
[ FOR REPLICATION ]
```

指定不能在订阅服务器上执行为复制创建的存储过程。

```
AS { <sql_statement> [;][ ...n ] | <method_specifier> }   /*执行的操作*/
[;]
```

要包含在过程中的一个或多个 T-SQL 语句。

```
<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
```

指示 SQL Server 将 CREATE PROCEDURE 语句的原始文本转换为模糊格式。

指示数据库引擎不缓存该过程的计划，该过程在运行时编译。

对于存储过程要注意以下几点：

- 用户定义的存储过程只能在当前数据库中创建（临时存储过程除外，临时存储过程总是在系统数据库tempdb中创建）；
- 成功执行CREATE PROCEDURE语句后，存储过程名称存储在sysobjects系统表中，而该语句的文本存储在syscomments中；
- SQL Server启动时可以自动执行一个或多个存储过程，这些存储过程必须由系统管理员在master数据库中创建，并在sysadmin固定服务器角色下作为后台过程执行，这些过程不能有任何输入参数。

## （2）执行存储过程

```
[ { EXEC | EXECUTE } ]
    { [ @return_status = ]
      { module_name [ ;number ] | @module_name_var }
      [ [ @parameter = ] { value    | @variable [ OUTPUT ] | [ DEFAULT ] } ]
      [ ,...n ]
      [ WITH RECOMPILE ]
    }
[;]
```

可选的整型变量，存储模块的返回状态。

局部定义的变量名

要调用的存储过程的名称。

创建语句中定义的参数名。

局部变量，保存OUTPUT参数返回的值

（3）举例

① 简单存储过程

EG：返回081101号学生的成绩情况。

```
CREATE PROC student_info
AS
SELECT CJB.*
FROM CJB
WHERE CJB.Stu_ID='081101';
```

执行存储过程：

```
EXEC student_info
```

## ② 带参数的存储过程

EG：从PXSCJ数据库的三个表中查询某人指定课程的成绩和学分。

```
CREATE PROC student_info1 @name char(8),@cname char(16)
AS
SELECT XSB.Stu_ID,XSB.Sname,KCB.Cname,CJB.Grade,KCB.Credit
FROM XSB,KCB,CJB
WHERE XSB.Stu_ID=CJB.Stu_ID AND KCB.C_ID=CJB.C_ID
AND XSB.Sname=@name AND KCB.Cname=@cname;
```

执行存储过程：

```
EXEC student_info1 '王林','计算机基础';
```

```
EXEC student_info1 @name='王林',@cname='计算机基础';
```

```
EXEC student_info1 @name='王林','计算机基础';
```

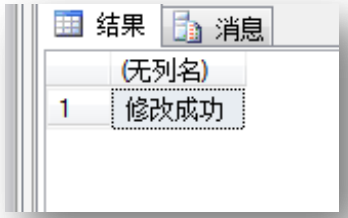
### ③ 带OUTPUT参数的存储过程

EG：创建一个存储过程do_insert，作用是向XSB表中插入一行数据。创建另外一个存储过程do_action，在其中调用第一个存储过程，并根据条件处理该行数据，处理后输入相应的信息。

```
CREATE PROC do_insert  AS
INSERT INTO XSB VALUES('091299','示例',1,'1991-5-5','软件工程',50,NULL);
GO
CREATE PROC do_action @X bit,@str char(8) OUTPUT  AS
BEGIN
EXEC do_insert  IF @X=0  BEGIN  UPDATE XSB SET Sname='LIU',Ssex=0
WHERE Stu_ID='091299‘  SET @str='修改成功‘  END
ELSE  IF @X=1  BEGIN  DELETE FROM XSB WHERE Stu_ID='091299'
SET @str='删除成功‘  END
END
```

执行存储过程：

```
DECLARE @str char(8)
EXEC do_action 0,@str OUTPUT
```




在存储过程执行时使用的OUTPUT参数需要用DECLARE命令在之前定义。

## ④ 带通配符参数的存储过程

EG：从三个表的连接中返回指定学生的学号、姓名、所选课程名称及该课程的成绩。

该存储过程在参数中使用了模式匹配，如果没有提供参数，则使用预设的默认值。

```
CREATE PROC st_info @name varchar(30)='李%'
AS
SELECT a.Stu_ID,a.Sname,c.Cname,b.Grade
FROM XSB a INNER JOIN CJB b ON a.Stu_ID=b.Stu_ID INNER JOIN KCB c ON c.C_ID=b.C_ID
WHERE Sname LIKE @name;
```

执行存储过程：

```
EXEC st_info
```

或

```
EXEC st_info '王%'
```

结果 消息

| | Stu_ID | Sname | Cname | Grade |
|---|---|---|---|---|
| 1 | 081104 | 李欣 | 计算机基础 | 90 |
| 2 | 081104 | 李欣 | 程序设计与语言 | 84 |
| 3 | 081104 | 李欣 | 离散数学 | 65 |
| 4 | 081107 | 李德军 | 计算机基础 | 78 |
| 5 | 081107 | 李德军 | 程序设计与语言 | 80 |
| 6 | 081107 | 李德军 | 离散数学 | 68 |
| 7 | 081241 | 李莎 | 计算机基础 | 74 |

结果 消息

| | Stu_ID | Sname | Cname | Grade |
|---|---|---|---|---|
| 1 | 081102 | 王洪泽 | 程序设计与语言 | 78 |
| 2 | 081102 | 王洪泽 | 离散数学 | 78 |
| 3 | 081109 | 王晓峰 | 计算机基础 | 66 |
| 4 | 081109 | 王晓峰 | 程序设计与语言 | 83 |
| 5 | 081109 | 王晓峰 | 离散数学 | 70 |
| 6 | 081201 | 王敏 | 计算机基础 | 80 |
| 7 | 081201 | 王敏 | 程序设计与语言 | 85 |
| 8 | 081202 | 王林 | 计算机基础 | 65 |
| 9 | 081203 | 王玉民 | 计算机基础 | 87 |

（4）修改存储过程

ALTER { PROC | PROCEDURE } [schema_name.] procedure_name [ ;

number ]

[ { @parameter [ type_schema_name. ] data_type }

[ VARYING ] [ = default ] [ [ OUT [ PUT ]

] [ ,...n ]

[ WITH <procedure_option> [ ,...n ] ]

[ FOR REPLICATION ]

AS

{ <sql_statement> [ ...n ] | <method_specifier> }

使用ALTER PROC命令可以修改已经存在的存储过程并保留以前赋予的许可，更改后，存储过程的权限和启动属性保持不变。各参数含义与CREATE PROC相同。

EG：修改存储过程student_info1，将第一个参数改成学生的学号。

```
ALTER PROC student_info1
@number char(6),@canme char(16)
AS
SELECT Stu_ID,Cname,Grade
FROM CJB,KCB
WHERE CJB.Stu_ID=@number AND KCB.Cname=@canme;
```

EG：创建名为select_students的存储过程，在默认情况下，该存储过程可查询所有学生信息。当该存储过程需更改为能检索计算机专业的学生信息时，用ALTER PROC重新定义该存储过程。

- 创建select_students存储过程

```
CREATE PROC select_students
AS SELECT * FROM XSB ORDER BY XSB.Stu_ID;
```

- 修改select_students存储过程

```
ALTER PROC select_students
AS
SELECT * FROM XSB WHERE XSB.Major='计算机'
ORDER BY XSB.Stu_ID;
```

（5）删除存储过程

DROP { PROC | PROCEDURE } { [ schema_name. ] procedure } [ ,...n ]

使用DROP PROCEDURE 语句可以永久的删除存储过程。

删除存储过程之前必须确认该存储过程没有任何依赖关系。

EG：删除PXSCJ数据库中的student_info 存储过程。

```
IF EXISTS(SELECT name FROM sysobjects WHERE name='student_info')
DROP PROC student_info;
```

2、触发器

触发器是一种特殊类型的存储过程，它在指定的表中的数据发生变化时自动生效。

触发器和普通的存储过程的区别是：触发器是当对某一个表进行操作。诸如：update、insert、delete这些操作的时候，系统会自动调用执行该表上对应的触发器。

（1）触发器的类型

在SQL Server 2008中，按照触发事件的不同可以将触发器分为：

- DML触发器
- DDL触发器

① DML触发器

当数据库中发生数据操纵语言（DML）事件时将调用DML触发器；

DML触发器分为三种类型：INSERT、UPDATE和DELETE；

- 方便地保持数据库中数据的完整性；
- 实现多个表间数据的一致性。

② DDL触发器

当数据库中发生数据定义语句（DDL）事件时将调用DDL触发器，如CREATE、ALTER、DROP等关键字开头的语句；

- 防止对数据库架构进行某些修改；
- 希望数据库中发生某些变化以利于相应数据库架构中的更改；
- 记录数据库架构中的更改或事件。

（2）创建DML触发器

触发器名

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }                                    /*指定操作对象*/
[ WITH    ENCRYPTION ]                                 /*说明是否采用加密方式*/
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]                                /*说明该触发器不用于复制*/
AS { sql_statement    [ ; ] [ ,...n ] | EXTERNAL NAME assembly_name.class_name.method_name }
```

用触发器中的操作代替触发语句的操作

指定激活触发器的语句类型

再添加一个现有类型的触发器

触发器的SQL语句

① 触发器中使用的特殊表

- inserted表：当向表中插入数据时，INSERT触发器触发执行，新的纪录插入到触发器表和inserted表中；
- deleted表：用于保存已从表中删除的纪录，当触发一个DELETE触发器时，被删除的纪录存放到deleted表中。

当对定义了UPDATE触发器的表记录修改时，表中原记录移动deleted表中，修改过的纪录插入到inserted表中。

inserted表和deleted表都是临时表，只可以在触发器语句中使用SELECT语句查询这两个表。

② 几点说明：

- CREATE TRIGGER语句只能应用到一个表中；
- DML触发器只能在当前数据库中创建；
- 在同一CREATE TRIGGER语句中，可以为多种操作（如INSERT和UPDATE）定义相同的触发器操作；
- 不能对临时表和系统表创建DML触发器；

③ 创建INSERT触发器

EG：创建触发器，当向CJB表中插入一个学生的成绩时，将XSB表中该学生的总学分加上添加的课程的学分。

```
CREATE TRIGGER cjb_insert
ON CJB AFTER INSERT
AS
BEGIN
DECLARE @num char(6),@kc_num char(3)
DECLARE @xf int
SELECT @num=CJB.Stu_ID,@kc_num=CJB.C_ID FROM inserted
SELECT @xf=KCB.Credit FROM KCB WHERE KCB.C_ID=@kc_num
UPDATE XSB SET Tcredit=Tcredit+@xf WHERE Stu_ID=@num
PRINT '修改成功'
END;
```

④ 创建UPDATE触发器

EG：创建触发器，当修改XSB表中的学号时，同时也要将CJB表中的学号修改成相应的学号（假设XSB表和CJB表之间没有定义外键约束）。

```
CREATE TRIGGER xsb_update
ON XSB AFTER UPDATE
AS
BEGIN
DECLARE @old_num char(6),@new_num char(6)
SELECT @old_num=Stu_ID FROM deleted
SELECT @new_num=Stu_ID FROM inserted
UPDATE CJB SET Stu_ID=@new_num WHERE Stu_ID=@old_num
END;
```

⑤ 创建DELETE触发器

EG：在删除XSB表中的一条学生记录时将CJB表中该学生的相应记录也删除。

```
CREATE TRIGGER xsb_delete
ON XSB AFTER DELETE
AS
BEGIN
DELETE FROM CJB WHERE Stu_ID IN(SELECT Stu_ID FROM deleted)
END;
```

（3）创建DDL触发器

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH    ENCRYPTION]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement    [ ; ] [ ,...n ]
| EXTERNAL NAME    assembly_name.class_name.method_name    [ ; ] }
```

将当前DDL触发器应用于当前服务器

DDL触发器的有效事件

DDL触发器的有效事件组

EG：创建PXSCJ数据库作用域的DDL触发器，当删除一个表时，提示禁止该操作，然后回滚删除表的操作。

```
CREATE TRIGGER safety
ON DATABASE
AFTER DROP_TABLE
AS
PRINT'不能删除该表'
ROLLBACK TRANSACTION;
```

用于回滚之前所做的修改，将数据库恢复到原来的状态

（4）删除触发器

```
DROP TRIGGER [schema_name.]trigger_name [ ,...n ] [ ; ]          /*删除 DML 触发器*/
DROP TRIGGER trigger_name [ ,...n ] ON { DATABASE | ALL SERVER } [ ; ]
                                                                 /*删除 DDL 触发器*/
```

EG：删除DML触发器xsb_delete。

```
IF EXISTS(SELECT name FROM sysobjects WHERE name='xsb_delete')
DROP TRIGGER xsb_delete;
```

EG：删除DDL触发器safety。

```
DROP TRIGGER safety ON DATABASE;
```

删除DDL触发器，需要使用ON关键字指定在数据库作用域还是服务器作用域。