**SQLSERVER数据库的数据库对象**

**Database Objects**

# 01、数据库(Databases)

具体请参考：<https://msdn.microsoft.com/zh-cn/library/ms176061(v=sql.100).aspx>

-- 新建;

CREATE DATABASE database\_name

[ ON

[ PRIMARY ] [ <filespec> [ ,...n ]

[ , <filegroup> [ ,...n ] ]

[ LOG ON { <filespec> [ ,...n ] } ]

]

[ COLLATE collation\_name ]

[ WITH <external\_access\_option> ]

]

[;]

-- 附加;

CREATE DATABASE database\_name

ON <filespec> [ ,...n ]

FOR { ATTACH [ WITH <service\_broker\_option> ]

| ATTACH\_REBUILD\_LOG }

[;]

<filespec>::=

{

(

NAME =logical\_file\_name,

FILENAME = { 'os\_file\_name' | 'filestream\_path' }

[ , SIZE =size [ KB | MB | GB | TB ] ]

[ , MAXSIZE = { max\_size [ KB | MB | GB | TB ] | UNLIMITED } ]

[ , FILEGROWTH =growth\_increment [ KB | MB | GB | TB | % ] ]

) [ ,...n ]

}

<filegroup> ::=

{

FILEGROUP filegroup\_name [ CONTAINS FILESTREAM ] [ DEFAULT ]

<filespec> [ ,...n ]

}

<external\_access\_option> ::=

{

[ DB\_CHAINING { ON | OFF } ]

[ , TRUSTWORTHY { ON | OFF } ]

}

<service\_broker\_option> ::=

{

ENABLE\_BROKER

| NEW\_BROKER

| ERROR\_BROKER\_CONVERSATIONS

}

-- 快照;

CREATE DATABASE database\_snapshot\_name

ON

(

NAME = logical\_file\_name,

FILENAME ='os\_file\_name'

) [ ,...n ]

AS SNAPSHOT OF source\_database\_name

[;]

# 02、数据库关系图/图表(Database Diagrams)

图表其实就是数据库表之间的关系示意图(**数据库关系图**)；

利用它可以编辑表与表之间的关系；

# 03、表(Tables)

数据库中的表与我们日常生活中使用的表格类似，它也是由行(Row)和列(Column)组成的；

列由同类的信息组成，每列又称为一个字段，每列的标题称为字段名；

行包括了若干列信息项；

一行数据称为一个或一条记录，它表达有一定意义的信息组合；

一个数据库表由一条或多条记录组成，没有记录的表称为空表；

每个表中通常都有一个主关键字，用于惟一地确定一条记录；

CREATE TABLE

[ database\_name . [ schema\_name ] . | schema\_name . ] table\_name

( { <column\_definition> | <computed\_column\_definition>

| <column\_set\_definition> }

[ <table\_constraint> ] [ ,...n ] )

[ ON { partition\_scheme\_name ( partition\_column\_name ) | filegroup

| "default" } ]

[ { TEXTIMAGE\_ON { filegroup | "default" } ]

[ FILESTREAM\_ON { partition\_scheme\_name | filegroup

| "default" } ]

[ WITH ( <table\_option> [ ,...n ] ) ]

[ ; ]

# 04、视图(Views, VW)

视图看上去同表似乎一模一样，具有一组命名的字段和数据项，但它其实是一个虚拟的表，在数据库中并不实际存在；

视图是由查询数据库表产生的，它限制了用户能看到和修改的数据；

视图可以用来控制用户对数据的访问，并能简化数据的显示，即通过视图只显示那些需要的数据信息；

CREATE VIEW [ schema\_name . ] view\_name [ (column [ ,...n ]) ]

[ WITH <view\_attribute> [ ,...n ] ]

AS select\_statement

[ WITH CHECK OPTION ] [ ; ]

<view\_attribute> ::=

{

[ ENCRYPTION ]

[ SCHEMABINDING ]

[ VIEW\_METADATA ]

}

# 05、同义词(Synonyms, SY)

可以为下列对象类型创建同义词：

程序集(CLR)存储过程、程序集(CLR)表值函数、程序集(CLR)标量函数、程序集聚合(CLR)聚合函数、复制筛选过程、扩展存储过程、SQL标量函数、SQL表值函数、SQL内联表值函数、SQL存储过程、视图、表；

-- 定义(表);

CREATE SYNONYM SY\_MyProduct

FOR dbo.Product;

GO

-- 用法;

SELECT ProductID, Name

FROM SY\_MyProduct

WHERE ProductID < 5;

GO

-- 定义(标量值函数);

CREATE FUNCTION dbo.OrderDozen(@OrderAmt int)

RETURNS int

WITH EXECUTE AS CALLER

AS

BEGIN

IF @OrderAmt % 12 <> 0

BEGIN

SET @OrderAmt += 12 - (@OrderAmt % 12)

END

RETURN(@OrderAmt);

END;

GO

-- 定义(同义词);

CREATE SYNONYM dbo.SY\_CorrectOrder

FOR dbo.OrderDozen;

GO

-- 用法;

DECLARE @Amt int

SET @Amt = 15

SELECT @Amt AS OriginalOrder, dbo.SY\_CorrectOrder(@Amt) AS ModifiedOrder;

# 06、可编程性(Programmability)

## 06.01、存储过程(Stored Procedures, UP)

存储过程是为完成特定的功能而汇集在一起的一组SQL 程序语句，经编译后存储在数据库中的SQL 程序；

### 06.01.01、系统存储过程

**常用系统存储过程：**

-- 查看数据库;

EXEC SP\_DATABASES;

-- 查看表;

EXEC SP\_TABLES;

-- 查看列;

EXEC SP\_COLUMNS OrderBasic;

-- 查看索引;

EXEC SP\_HELPINDEX OrderBasic;

-- 查看约束;

EXEC SP\_HELPCONSTRAINT OrderBasic;

-- 查看存储过程;

EXEC SP\_STORED\_PROCEDURES;

-- 查看定义语句;

EXEC SP\_HELPTEXT 'SP\_STORED\_PROCEDURES';

-- 重命名(表/列/索引等数据库对象);

EXEC SP\_RENAME Table1, Table2;

-- 重命名数据库;

EXEC SP\_RENAMEDB MyDB1, MyDB2;

-- 更改登录名的默认数据库;

EXEC SP\_DEFAULTDB 'master', 'MyDB1';

-- 数据库帮助;

EXEC SP\_HELPDB;

-- 查询数据库信息;

EXEC SP\_HELPDB master;

### 06.01.02、用户存储过程(自定义存储过程)

具体请参考：<https://msdn.microsoft.com/en-us/library/ms187926(v=sql.110).aspx>

-- 创建存储过程(简单);

CREATE PROCEDURE dbo.UP\_Employee\_GETS

AS

BEGIN

SET NOCOUNT ON;

SELECT \* FROM dbo.Employee;

END

GO

-- 执行存储过程;

EXECUTE dbo.UP\_Employee\_GETS;

GO

-- 创建存储过程(返回多数据集);

CREATE PROCEDURE dbo.UP\_Person\_Customer\_GETS

AS

BEGIN

SELECT TOP(10) \* FROM dbo.Person;

SELECT TOP(10) \* FROM dbo.Customer;

END

GO

-- 执行存储过程;

EXECUTE dbo.UP\_Person\_Customer\_GETS;

GO

-- 创建存储过程(输入参数);

CREATE PROCEDURE dbo.UP\_OrderItem\_GETS

(

@Quantity INT

)

AS

BEGIN

SET NOCOUNT ON;

SELECT \* FROM dbo.OrderItem WHERE Quantity >= @Quantity;

END

GO

-- 执行存储过程;

EXECUTE dbo.UP\_OrderItem\_GETS 10;

EXECUTE dbo.UP\_OrderItem\_GETS @Quantity = 10;

GO

-- 创建存储过程(输入参数);

CREATE PROCEDURE dbo.UP\_Employee\_GETS2

(

@FirstName NVARCHAR(50),

@LastName NVARCHAR(50)

)

AS

BEGIN

SET NOCOUNT ON;

SELECT \* FROM dbo.Employee WHERE FirstName = @FirstName AND LastName = @LastName;

END

GO

-- 执行存储过程;

EXECUTE dbo.UP\_Employee\_GETS2 N'Pilar', N'Ackerman';

EXECUTE dbo.UP\_Employee\_GETS2 @FirstName = N'Pilar', @LastName = N'Ackerman';

GO

-- 创建存储过程(输出参数);

CREATE PROCEDURE dbo.UP\_Product\_GETSXXXX

(

@Product VARCHAR(40),

@MaxQty INT OUTPUT

)

AS

BEGIN

SET NOCOUNT ON;

SELECT @MaxQty = MAX(Quantity) FROM dbo.OrderItem;

END

-- 执行存储过程;

DECLARE @MaxQty INT;

EXECUTE UP\_Product\_GETSXXXX '', @MaxQty OUTPUT;

PRINT @MaxQty;

GO

-- 创建存储过程(输出游标参数);

CREATE PROCEDURE dbo.UP\_CurrencyCursor

(

@CurrencyCursor CURSOR VARYING OUTPUT

)

AS

BEGIN

SET NOCOUNT ON;

SET @CurrencyCursor = CURSOR

FORWARD\_ONLY STATIC FOR

SELECT CurrencyCode, Name FROM dbo.Currency;

OPEN @CurrencyCursor;

END

GO

-- 执行存储过程;

DECLARE @MyCursor CURSOR;

EXEC dbo.UP\_CurrencyCursor @CurrencyCursor = @MyCursor OUTPUT;

WHILE(@@FETCH\_STATUS = 0)

BEGIN

FETCH NEXT FROM @MyCursor;

END

CLOSE @MyCursor;

DEALLOCATE @MyCursor;

GO

-- 创建存储过程(TRY+CATCH+TRAN);

CREATE PROCEDURE dbo.UP\_WorkOrder\_DELETE

(

@WorkOrderID INT

)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

BEGIN TRAN

DELETE FROM dbo.WorkOrderRouting WHERE WorkOrderID = @WorkOrderID;

DELETE FROM dbo.WorkOrder WHERE WorkOrderID = @WorkOrderID;

COMMIT

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0 ROLLBACK

DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT;

SELECT @ErrorMessage = ERROR\_MESSAGE(), @ErrorSeverity = ERROR\_SEVERITY();

RAISERROR(@ErrorMessage, @ErrorSeverity, 1);

END CATCH;

END

GO

-- 执行存储过程;

EXEC dbo.UP\_WorkOrder\_DELETE 13;

### 06.01.03、扩展存储过程

具体请参考：<http://blog.csdn.net/xmzhaoym/article/details/8745178>

## 06.02、函数(Functions, UF)

SQL Server并不将用户限制在定义为 Transact-SQL语言一部分的内置函数上，而是允许用户创建自己的用户定义函数；

函数是由一个或多个Transact-SQL语句组成的子程序，可用于封装代码以便重新使用；

### 06.02.01、表值函数(Table-Valued Functions, 返回表)

|  |  |  |
| --- | --- | --- |
| **函数类型** | **定义** | **示例** |
| **内联表值函数** | 内联表值函数在概念上与带参数的视图类似； | -- 定义;  CREATE FUNCTION dbo.UF\_SalesByCustomer(@CustomerID int)  RETURNS TABLE AS  RETURN  (  SELECT  P.ProductID, P.Name, SUM(SD.LineTotal) AS Total  FROM  dbo.Production.Product AS P LEFT JOIN  dbo.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID LEFT JOIN  dbo.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID  WHERE SH.CustomerID = @CustomerID  GROUP BY P.ProductID, P.Name  );  GO  -- 用法;  SELECT \* FROM dbo.UF\_SalesByCustomer(30052); |
| **多语句表值函数** | 多语句表值函数允许多条语句在表变量中创建结果集来返回； | -- 定义;  CREATE FUNCTION dbo.UF\_SalesByCustomerMS(@CustomerID int)  RETURNS @table TABLE  (  ProductID int PRIMARY KEY NOT NULL,  ProductName nvarchar(50) NOT NULL,  TotalSales numeric(38,6) NOT NULL,  TotalInventory int NOT NULL  ) AS  BEGIN  INSERT INTO @table  SELECT  P.ProductID, P.Name, SUM(SD.LineTotal) AS Total, 0  FROM  Production.Product AS P LEFT JOIN  dbo.SalesOrderDetail SD ON SD.ProductID = P.ProductID LEFT JOIN  dbo.SalesOrderHeader SH ON SH.SalesOrderID = SD.SalesOrderID  WHERE SH.CustomerID = @CustomerID  GROUP BY P.ProductID, P.Name;  UPDATE @table SET TotalInventory = dbo.UF\_GetTotalInventoryStock(ProductID);  RETURN;  END  GO  -- 用法;  SELECT \* FROM dbo.UF\_SalesByCustomerMS(30052); |

### 06.02.02、标量值函数(Scalar-Valued Functions, 返回值)

具体请参考(示例)：<http://blog.csdn.net/ycl295644/article/details/50764003>

-- 定义;

CREATE FUNCTION dbo.UF\_GetStudentName(@StudentId INT)

RETURNS NVARCHAR(50) AS

BEGIN

DECLARE @studentName NVARCHAR(50)

SELECT

@studentName = StudentName

FROM

dbo.StudentInfo

WHERE StudentId = @StudentId

END

GO

-- 用法;

SELECT dbo.UF\_GetStudentName(1);

-- 定义;

CREATE FUNCTION UF\_DataFormat(@strDate datetime)

RETURNS NVARCHAR(20) AS

BEGIN

DECLARE @date NVARCHAR(20)

SET @date = DATENAME(YY, @strDate) + '年'+ CONVERT(NVARCHAR, MONTH(@strDate)) + '月' + CONVERT(NVARCHAR, DAY(@strDate)) + '日'

RETURN @date

END

GO

-- 用法;

SELECT dbo.UF\_DataFormat('2017-01-10 02:30:01');

### 06.02.03、聚合函数(Aggregate Functions)

具体请参考：<http://www.cnblogs.com/volts0302/p/5545401.html>

### 06.02.03、系统函数(System Functions)

具体请参考：D01.SQLSERVER数据库的T-SQL基本语法(T-SQL Basic Syntax).docx的05.03、系统函数(System Functions)小节；

## 06.03、数据库触发器(Database Triggers, TR)

A、触发器是SQL Server提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，当对一个表进行操作(Insert/Update/Delete)时就会激活它执行，触发器经常用于加强数据的完整性约束和业务规则等；

B、触发器与存储过程的区别是运行方式的不同，触发器不能执行EXECUTE语句调用，而是在用户执行T-SQL语句时自动触发执行而存储过程需要用户，应用程序或者触发器来显示地调用并执行；

C、触发器触发时，系统自动在内存中创建DELETED表或INSERTED表，只读，不允许修改，触发器执行完成后，自动删除；

具体请参考：<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql>

### 06.03.01、DML触发器(DML Trigger)

-- 语法;

CREATE [ OR ALTER ] TRIGGER [ schema\_name . ]trigger\_name

ON { table | view }

[ WITH <dml\_trigger\_option> [ ,...n ] ]

{ FOR | INSTEAD OF | AFTER }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

AS { sql\_statement [ ; ] [ ,...n ] }

<dml\_trigger\_option> ::=

[ NATIVE\_COMPILATION ]

[ SCHEMABINDING ]

[ EXECUTE AS Clause ]

**FOR与INSERT/UPDATE/DELETE的矩阵：**

|  |  |  |  |
| --- | --- | --- | --- |
| **FOR** | **INSERT** | **UPDATE** | **DELETE** |
| **INSTEAD OF(替换)** | INSTEAD OF INSERT | INSTEAD OF UPDATE | INSTEAD OF DELETE |
| **AFTER** | AFTER INSERT | AFTER UPDATE | AFTER DELETE |

**临时表DELETED/INSERTED：**

|  |  |  |
| --- | --- | --- |
| **操作** | **DELETED表** | **INSERTED表** |
| **INSERT** | 无 | 存放被新增的记录 |
| **UPDATE** | 存放更新前的记录 | 存放更新后的记录 |
| **DELETE** | 存放被删除的记录 | 无 |

-- (准备测试环境)创建主表;

CREATE TABLE dbo.OrderBasic(

PurchaseId BIGINT NOT NULL,

ItemQty INT NULL,

CONSTRAINT PK\_OrderBasic PRIMARY KEY CLUSTERED(PurchaseId)

)

-- 创建从表;

CREATE TABLE dbo.OrderItem(

ItemId BIGINT NOT NULL,

Quantity INT NULL,

CONSTRAINT PK\_OrderItem PRIMARY KEY CLUSTERED(ItemId)

)

GO

#### 06.03.01.A、测试AFTER INSERT场景

-- AFTER INSERT;

CREATE TRIGGER TR\_OrderItem\_AFTER\_INSERT

ON OrderItem AFTER INSERT

AS

BEGIN

DECLARE @ItemQty INT;

SELECT @ItemQty = SUM(Quantity) FROM OrderItem;

IF NOT EXISTS (SELECT \* FROM OrderBasic)

BEGIN

INSERT INTO OrderBasic VALUES(1, 0);

END

UPDATE OrderBasic SET ItemQty = @ItemQty;

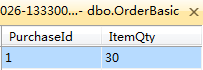
END

GO

-- 测试AFTER;

INSERT INTO dbo.OrderItem VALUES(1, 10);

INSERT INTO dbo.OrderItem VALUES(2, 20);



-- 附: 回滚插入等事务;

CREATE TRIGGER TR\_OrderItem\_AFTER\_INSERT

ON OrderItem AFTER INSERT

AS

BEGIN

DECLARE @ItemQty INT;

SELECT @ItemQty = Quantity FROM INSERTED;

IF(@ItemQty >= 100)

BEGIN

RAISERROR('数量超过100件上限.', 1, 1);

ROLLBACK TRAN;

END

END

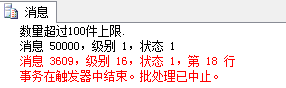
GO

DROP TRIGGER TR\_OrderItem\_AFTER\_INSERT

-- 测试(失败);

INSERT INTO dbo.OrderItem VALUES(1, 50);

INSERT INTO dbo.OrderItem VALUES(2, 100);



#### 06.03.01.B、测试AFTER UPDATE场景

-- AFTER UPDATE;

CREATE TRIGGER TR\_OrderItem\_AFTER\_UPDATE

ON OrderItem AFTER UPDATE

AS

BEGIN

DECLARE @ItemQty1 INT, @ItemQty2 INT;

-- 更新前的;

SELECT @ItemQty1 = Quantity FROM DELETED;

-- 更新后的;

SELECT @ItemQty2 = Quantity FROM INSERTED;

-- 重新核算;

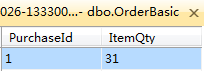
UPDATE OrderBasic SET ItemQty = ItemQty - @ItemQty1 + @ItemQty2;

END

GO

-- 测试AFTER;

UPDATE dbo.OrderItem SET Quantity = 11 WHERE ItemId = 1;



#### 06.03.01.C、测试AFTER DELETE场景

-- AFTER DELETE;

CREATE TRIGGER TR\_OrderItem\_AFTER\_DELETE

ON OrderItem AFTER DELETE

AS

BEGIN

DECLARE @ItemQty INT;

-- 更新前的;

SELECT @ItemQty = Quantity FROM DELETED;

-- 重新核算;

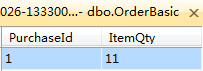
UPDATE OrderBasic SET ItemQty = ItemQty - @ItemQty;

END

GO

-- 测试AFTER;

DELETE FROM dbo.OrderItem WHERE ItemId = 2;



#### 06.03.01.D、测试INSTEAD OF INSERT场景

-- INSTEAD OF INSERT;

CREATE TRIGGER TR\_OrderItem\_INSTEADOF\_INSERT

ON OrderItem INSTEAD OF INSERT

AS

BEGIN

DECLARE @ItemQty INT;

SELECT @ItemQty = Quantity FROM INSERTED;

IF(@ItemQty >= 100)

BEGIN

RAISERROR('数量超过100件上限.', 1, 1);

RETURN;

END

INSERT INTO dbo.OrderItem SELECT \* FROM INSERTED;

END

GO

-- 测试(成功);

INSERT INTO dbo.OrderItem VALUES(1, 50);

-- 测试(失败);

INSERT INTO dbo.OrderItem VALUES(2, 100);

### 06.03.02、DDL触发器(DDL Trigger)

具体请参考：<https://docs.microsoft.com/en-us/sql/relational-databases/triggers/ddl-triggers>

### 06.03.03、登录触发器(Logon Trigger)

具体请参考：<https://docs.microsoft.com/en-us/sql/relational-databases/triggers/logon-triggers>

### 06.03.04、管理触发器

-- 查看某表的触发器;

EXEC SP\_HELPTRIGGER 'OrderItem';



-- 查看表与触发器的对应关系;

SELECT

tb2.name AS TableName, tb1.name AS TriggerName

FROM

dbo.sysobjects tb1 LEFT JOIN

dbo.sysobjects tb2 ON tb1.parent\_obj = tb2.id

WHERE tb1.type = 'TR';



-- 禁用触发器;

DISABLE TRIGGER TR\_OrderItem\_INSTEADOF\_INSERT ON dbo.OrderItem;

DISABLE TRIGGER ALL ON dbo.OrderItem;

-- 启用触发器;

ENABLE TRIGGER TR\_OrderItem\_INSTEADOF\_INSERT ON dbo.OrderItem;

ENABLE TRIGGER ALL ON dbo.OrderItem;

-- 删除触发器;

DROP TRIGGER TR\_OrderItem\_INSTEADOF\_INSERT;

## 06.04、程序集(Assemblies)

具体请参考：<http://blog.csdn.net/xmzhaoym/article/details/8745178>

## 06.05、类型(Types)

### 06.05.01、系统数据类型(System Data Types)

具体请参考：D01.SQLSERVER数据库的T-SQL基本语法(T-SQL Basic Syntax).docx的06、系统数据类型的对照表(System Data Types)小节；

### 06.05.02、用户定义数据类型(User-Defined Data Types)

具体请参考：<https://stackoverflow.com/questions/9558220/sql-server-2005-user-defined-data-type-tempdb-and-permissions>

### 06.05.03、用户定义表类型(User-Defined Table Types)

具体请参考：<http://www.cnblogs.com/-Apple/p/3549889.html>

### 06.05.04、用户定义类型(User-Defined Types)

具体请参考：<https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-types/registering-user-defined-types-in-sql-server>

### 06.05.05、XML架构集合(XML Schema Collections)

具体请参考：<https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-schema-collections-sql-server>

## 06.06、规则(Rules, RL)

A、规则是对数据库表中数据信息的限制，它限定的是表的列；

B、**规则已被抛弃**，MicroSoft不准备后续支持这个特性，推荐使用**CHECK约束(CHECK CONSTRAINT)**；

-- 创建表;

CREATE TABLE PersonInfo1(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

Age1 INT,

CONSTRAINT PK\_PersonInfo1 PRIMARY KEY(PersonId)

);

-- 创建表;

CREATE TABLE PersonInfo2(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

Age2 INT,

CONSTRAINT PK\_PersonInfo2 PRIMARY KEY(PersonId)

);

-- 创建规则;

CREATE RULE RL\_Age

AS @Age > 18;

-- 绑定规则;

EXEC SP\_BINDRULE 'RL\_Age', 'PersonInfo1.Age1';

EXEC SP\_BINDRULE 'RL\_Age', 'PersonInfo2.Age2';

-- 测试插入(失败);

INSERT INTO PersonInfo1 VALUES(1, 'LastName', 'FirstName', 17);

INSERT INTO PersonInfo2 VALUES(1, 'LastName', 'FirstName', 17);

-- 解绑规则;

EXEC SP\_UNBINDRULE 'PersonInfo1.Age1';

-- 删除规则(失败: 还有1个绑定在PersonInfo2.Age2, 必须解绑所有表列, 才可删除规则);

DROP RULE RL\_Age;

-- 解绑规则;

EXEC SP\_UNBINDRULE 'PersonInfo2.Age2';

-- 删除规则(成功);

DROP RULE RL\_Age;

## 06.07、默认值(Defaults, DF)

A、默认值类似于DEFAULT约束；

B、默认值-DEFAULT约束的关系与规则-CHECK约束的关系差不多；

C、区别在于它们被追加到表中的方式和对用户自定义数据类型的默认值(是对象而不是约束)支持；

D、**默认值已被抛弃**，MicroSoft不准备后续支持这个特性，推荐使用**DEFAULT约束(DEFAULT CONSTRAINT)**；

-- 创建表;

CREATE TABLE PersonInfo1(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

Age1 INT,

CONSTRAINT PK\_PersonInfo1 PRIMARY KEY(PersonId)

);

-- 创建表;

CREATE TABLE PersonInfo2(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

Age2 INT,

CONSTRAINT PK\_PersonInfo2 PRIMARY KEY(PersonId)

);

-- 创建默认值;

CREATE DEFAULT DF\_Age

AS 18;

-- 绑定默认值;

EXEC SP\_BINDEFAULT 'DF\_Age', 'PersonInfo1.Age1';

EXEC SP\_BINDEFAULT 'DF\_Age', 'PersonInfo2.Age2';

-- 测试插入(成功);

INSERT INTO PersonInfo1(PersonId, LastName, FirstName) VALUES(1, 'LastName', 'FirstName');

INSERT INTO PersonInfo2(PersonId, LastName, FirstName) VALUES(1, 'LastName', 'FirstName');

-- 解绑默认值;

EXEC SP\_UNBINDEFAULT 'PersonInfo1.Age1';

-- 删除默认值(失败: 还有1个绑定在PersonInfo2.Age2, 必须解绑所有表列, 才可删除默认值);

DROP DEFAULT DF\_Age;

-- 解绑默认值;

EXEC SP\_UNBINDEFAULT 'PersonInfo2.Age2';

-- 删除默认值(成功);

DROP DEFAULT DF\_Age;

## 06.08、计划指南(Plan Guides)

具体请参考：<https://docs.microsoft.com/zh-cn/sql/relational-databases/performance/plan-guides>

## 06.09、序列(Sequences, SEQ)

场景：有时我们需要在多表之间，实现ID的一致性，在SQL SERVER里面就会有一定的麻烦，通常我们会使用额外使用一张temp表来映射这些ID的关系，然后再从中取序列来完成；

ORACLE中有SEQUENCE，SQL SERVER 2012也增加了SEQUENCE新特性，是一个基于schema的对象(数据库级)，所以可以被多表调用；

-- 语法;

CREATE SEQUENCE [schema\_name . ] sequence\_name

[ AS [ built\_in\_integer\_type | user-defined\_integer\_type ] ]

[ START WITH <constant> ]

[ INCREMENT BY <constant> ]

[ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]

[ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]

[ CYCLE | { NO CYCLE } ]

[ { CACHE [ <constant> ] } | { NO CACHE } ]

[ ; ]

-- 用法;

CREATE SEQUENCE [dbo].[Sequence-20170821-171032]

AS [bigint]

START WITH 1

INCREMENT BY 1

MINVALUE -9223372036854775808

MAXVALUE 9223372036854775807

CACHE

GO

-- 检查序列;

SELECT \* FROM sys.sequences;

-- 创建表1;

CREATE TABLE [dbo].[scott\_emp1](

[empid] [bigint] NOT NULL,

[sal] [decimal](18, 4) NULL,

CONSTRAINT [PK\_scott\_emp1] PRIMARY KEY CLUSTERED

(

[empid] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

-- 创建表2;

CREATE TABLE [dbo].[scott\_emp2](

[empid] [bigint] NOT NULL,

[sal] [decimal](18, 4) NULL,

CONSTRAINT [PK\_scott\_emp2] PRIMARY KEY CLUSTERED

(

[empid] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

-- 插入(++);

INSERT INTO [dbo].[scott\_emp1] VALUES(NEXT VALUE FOR [dbo].[Sequence-20170821-171032], 1000.00);

-- 插入(++);

INSERT INTO [dbo].[scott\_emp2] VALUES(NEXT VALUE FOR [dbo].[Sequence-20170821-171032], 1000.00);

-- 查询(++);

SELECT NEXT VALUE FOR [dbo].[Sequence-20170821-171032];

# 07、索引(Indexes, IX)

具体请参考：D04.SQLSERVER数据库的索引(Database Index).docx

# 08、键(Keys)

## 08.01、主键/主键约束(Primary Keys, PK)

具体请参考：09.01、主键约束(Primary Keys, PK)；

## 08.01、外键/外键约束(Foreign Keys, FK)

具体请参考：09.02、外键约束(Foreign Keys, FK)；

## 08.01、代理键/人造键(Surrogate Keys/Artificial Keys)

|  |  |
| --- | --- |
| **类型** | **示例** |
| **IDENTITY** | -- 语法(INT/TINYINT/SMALLINT/BIGINT/DECIMAL/NUMERIC);  IDENTITY [ ( seed, increment ) ] [ NOT FOR REPLICATION ]  -- 创建表;  CREATE TABLE dbo.Test1  (  TestId INT NOT NULL IDENTITY(1, 1),  CONSTRAINT PK\_Test1 PRIMARY KEY(TestId)  ) |
| **ROWGUIDCOL** |  |

# 09、约束(Constraints)

## 09.01、主键约束(Primary Keys, PK)

|  |  |
| --- | --- |
| **类型** | **示例** |
| **主键(单字段)** | -- 创建表时增加主键约束1(随机命名: PK\_\_Customer1\_\_A4AE64D893809AD9);  CREATE TABLE Customer1(  CustomerId INT IDENTITY(1,1) NOT NULL PRIMARY KEY,  CustomerName NVARCHAR(50) NULL  );  -- 创建表时增加主键约束2;  CREATE TABLE Customer2(  CustomerId INT IDENTITY(1,1) NOT NULL,  CustomerName NVARCHAR(50) NULL,  CONSTRAINT PK\_Customer2 PRIMARY KEY(CustomerId)  );  -- 创建表;  CREATE TABLE Customer3(  CustomerId INT IDENTITY(1,1) NOT NULL,  CustomerName NVARCHAR(50) NULL  );  -- 修改表并增加主键约束3;  ALTER TABLE Customer3 ADD CONSTRAINT PK\_Customer3 PRIMARY KEY(CustomerId); |
| **复合主键** | -- 创建表时增加复合主键;  CREATE TABLE StockQty(  CompanyId BIGINT NOT NULL,  StoreId BIGINT NOT NULL,  DeskId BIGINT NOT NULL,  GoodsId NVARCHAR(50) NOT NULL,  StockType INT NOT NULL,  FinanceQty INT NULL,  AvailableQty INT NULL,  OutakeQty INT NULL,  LastTime DATETIME NULL,  CONSTRAINT PK\_StockQty PRIMARY KEY(CompanyId, GoodsId, StoreId, DeskId, StockType)  ); |

## 09.02、外键约束(Foreign Keys, FK)

-- 创建主表;

CREATE TABLE dbo.OrderBasic(

OrderId BIGINT NOT NULL,

State INT NULL,

CONSTRAINT PK\_OrderBasic PRIMARY KEY CLUSTERED(OrderId)

);

-- 创建明细表时添加外键约束1(随机命名: FK\_\_OrderItem\_\_Order\_\_114A936A);

CREATE TABLE dbo.OrderItem1(

ItemId BIGINT NOT NULL,

OrderId BIGINT NULL FOREIGN KEY REFERENCES dbo.OrderBasic(OrderId),

GoodsId NVARCHAR(50) NULL,

CONSTRAINT PK\_OrderItem1 PRIMARY KEY CLUSTERED(ItemId),

);

-- 创建明细表时添加外键约束2;

CREATE TABLE dbo.OrderItem2(

ItemId BIGINT NOT NULL,

OrderId BIGINT NULL,

GoodsId NVARCHAR(50) NULL,

CONSTRAINT PK\_OrderItem2 PRIMARY KEY CLUSTERED(ItemId),

CONSTRAINT FK\_OrderItem2\_OrderId FOREIGN KEY(OrderId) REFERENCES dbo.OrderBasic(OrderId),

);

-- 创建明细表;

CREATE TABLE dbo.OrderItem3(

ItemId BIGINT NOT NULL,

OrderId BIGINT NULL,

GoodsId NVARCHAR(50) NULL,

CONSTRAINT PK\_OrderItem3 PRIMARY KEY CLUSTERED(ItemId),

);

-- 修改明细表并增加主外键约束3;

ALTER TABLE dbo.OrderItem3 ADD CONSTRAINT FK\_OrderItem3\_OrderId FOREIGN KEY(OrderId) REFERENCES dbo.OrderBasic(OrderId);

**-- 测试插入明细表(异常: 主表没有对应的OrderId值, 所以出现异常);**

INSERT INTO dbo.OrderItem1 VALUES(1001, 9001, 'AAAAA');

-- 测试插入一行主表记录(成功);

INSERT INTO dbo.OrderBasic VALUES(9001, 1);

-- 测试插入明细表(成功);

INSERT INTO dbo.OrderItem1 VALUES(1001, 9001, 'AAAAA');

INSERT INTO dbo.OrderItem1 VALUES(1002, 9001, 'BBBBB');

INSERT INTO dbo.OrderItem1 VALUES(1003, 9001, 'CCCCC');

-- 测试删除明细表(成功);

DELETE dbo.OrderItem1 WHERE ItemId = 1001;

**-- 默认动作: 防止删除被引用的记录, SQLServer默认动作, 子行存在时限制父行被删除;**

**-- 测试删除主表记录(异常: 明细表还有对应的OrderId值的记录, 所以出现异常);**

DELETE dbo.OrderBasic WHERE OrderId = 9001;

**-- 级联动作: 进行自动删除和自动更新;**

-- 先删除掉旧的外键约束;

ALTER TABLE dbo.OrderItem1 DROP CONSTRAINT FK\_\_OrderItem\_\_Order\_\_114A936A;

-- 修改明细表并增加主外键约束;

ALTER TABLE dbo.OrderItem1

ADD CONSTRAINT FK\_OrderItem1\_OrderId FOREIGN KEY(OrderId) REFERENCES dbo.OrderBasic(OrderId)

**-- ON更新规范/删除规则: NO ACTION/CASCADE/SET NULL/SET DEFAULT;**

ON UPDATE NO ACTION **-- 默认修改时补级联更新依赖行;**

ON DELETE CASCADE **-- 删除时级联删除依赖行;**

**-- 测试删除主表记录(成功并删除明细表的外键关联的记录行);**

DELETE dbo.OrderBasic WHERE OrderId = 9001;

## 09.03、唯一约束(Uniques, AK)

唯一约束另名替换键(ALTERNATE KEY, AK)；

唯一约束与主键约束的区别：

A、主键约束不允许出现NULL值，任何索引的索引键都不允许包含null值，但唯一约束允许包含NULL值，但唯一约束把两个NULL值当作重复值，所以施加了唯一约束的每一列只允许包含一个NULL值；

B、创建主键时会自动创建聚集索引，除非当前表中已经含有了聚集索引或是创建主键时指定了NONCLUSTERED关键字；

C、创建唯一约束时会自动创建非聚集索引，除非你指定了CLUSTERED关键字并且当前表中还没有聚集索引；

D、每个表中只能有一个主键，但可以由多个唯一约束；

-- 创建表时增加唯一约束1;

CREATE TABLE UserInfo1(

UserId INT IDENTITY(1,1) NOT NULL,

UserName NVARCHAR(50) NULL,

CONSTRAINT PK\_UserInfo1 PRIMARY KEY(UserId),

CONSTRAINT AK\_UserInfo1 UNIQUE(UserName)

);

-- 创建表;

CREATE TABLE UserInfo2(

UserId INT IDENTITY(1,1) NOT NULL,

UserName NVARCHAR(50) NULL,

CONSTRAINT PK\_UserInfo2 PRIMARY KEY(UserId)

);

-- 修改表并增加唯一约束2;

ALTER TABLE UserInfo2 ADD CONSTRAINT AK\_UserInfo2 UNIQUE(UserName);

## 09.04、检查约束(Checks, CK)

CHECK约束的表达式：

A、区间：BETWEEN 1 AND 12

B、正则：LIKE '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'

C、集合：IN('UPS', 'FedEx', 'EMS')

D、值的比较：UnitPrice >= 0

E、字段比较：ShipDate >= OrderDate

F、函数：LEN(Password) BETWEEN 6 AND 20

G、等等…

-- 创建表时增加CHECK约束1;

CREATE TABLE PersonInfo1(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

ProvinceId INT,

CityId INT,

TownId INT,

State INT,

CONSTRAINT PK\_PersonInfo1 PRIMARY KEY(PersonId),

CONSTRAINT CK\_PersonInfo1 CHECK (ProvinceId > 0 AND CityId > 0 AND TownId > 0 AND State IN (1, 2, 3))

);

-- 创建表;

CREATE TABLE PersonInfo2(

PersonId INT NOT NULL,

LastName NVARCHAR(50) NULL,

FirstName NVARCHAR(50) NULL,

ProvinceId INT,

CityId INT,

TownId INT,

State INT,

CONSTRAINT PK\_PersonInfo2 PRIMARY KEY(PersonId)

);

-- 修改表并增加CHECK约束2;

ALTER TABLE PersonInfo2 ADD CONSTRAINT CK\_PersonInfo2 CHECK (ProvinceId > 0 AND CityId > 0 AND TownId > 0 AND State IN (1, 2, 3));

-- 场景1: 在创建约束时忽略检查之前的不满足数据(忽略老数据不作约束检查);

ALTER TABLE Account WITH NOCHECK ADD CONSTRAINT CK\_Account\_Age CHECK(Age > 18);

-- 场景2: 临时禁用已存在的约束(临时禁用再启用);

ALTER TABLE Account ADD CONSTRAINT CK\_Account\_Age CHECK(Age > 18);

-- 临时禁用已存在的约束;

ALTER TABLE Account NOCHECK CONSTRAINT CK\_Account\_Age;

-- 批量插入/导入数据;

INSERT INTO Account VALUES(25, '临时禁用已存在的约束', 17);

-- INSERT INTO...

-- 检索约束是否启用/禁用;

EXEC SP\_HELPCONSTRAINT Account;



-- 启用已禁用的约束;

ALTER TABLE Account CHECK CONSTRAINT CK\_Account\_Age;

## 09.05、默认值约束(Defaults, DF)

缺省值是当在表中创建列或插入数据时，对没有指定其具体值的列或列数据项赋予事先设定好的值；

-- 创建表;

CREATE TABLE UserInfo(UserId INT);

-- 增加字段;

ALTER TABLE UserInfo ADD [State] INT CONSTRAINT DF\_UserInfo\_State DEFAULT(0) NOT NULL;

-- 修改约束(先删再增);

ALTER TABLE UserInfo DROP CONSTRAINT DF\_UserInfo\_State;

ALTER TABLE UserInfo ADD CONSTRAINT DF\_UserInfo\_State DEFAULT(1) FOR [State]

-- 查询默认值(约束);

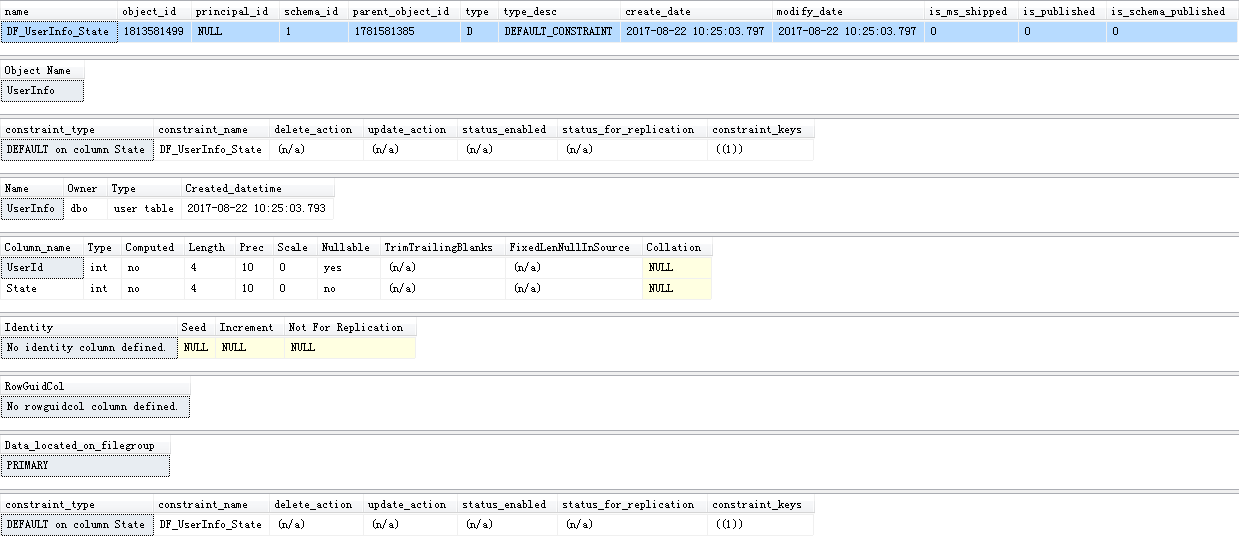
SELECT \* FROM sys.objects WHERE name = 'DF\_UserInfo\_State';

-- 查询表中的约束;

EXEC SP\_HELPCONSTRAINT 'UserInfo';

-- 查询某表信息;

EXEC SP\_HELP UserInfo;



# 10、游标(Cursors)

## 10.01、声明游标

-- 声明游标;

DECLARE cursor\_name CURSOR

[ LOCAL | GLOBAL ]

[ FORWARD\_ONLY | SCROLL ]

[ STATIC | KEYSET | DYNAMIC | FAST\_FORWARD ]

[ READ\_ONLY | SCROLL\_LOCKS | OPTIMISTIC ]

[ TYPE\_WARNING ]

FOR select\_statement

[ FOR UPDATE [ OF column\_name [ ,...n ] ] ]

[;]

|  |  |
| --- | --- |
| **选项** | **描述** |
| LOCAL | 作用域为局部； |
| GLOBAL(默认) | 作用域是全局； |
| FORWARD\_ONLY(单向) | 游标在数据集中从开始向结束的方向读取(FETCH NEXT)； |
| SCROLL(双向) | 游标在数据集中向任何方向读取，或任何位置移动； |
| STATIC(副本) | 将数据集的副本存入tempdb数据库中，任何对于底层表内数据的更改不会影响到游标的内容； |
| KEYSET(主键) | 将数据集的每一行的主键存入tempdb数据库中，当数据集中任何行改变或者删除时，@@FETCH\_STATUS=-2；  KEYSET无法探测新加入的数据； |
| DYNAMIC(随变) | 当底层表内数据更改时，游标的内容也随之得到反映，在下一次FETCH中，数据内容会随之改变； |
| FAST\_FORWARD(单向优化) | 启动性能优化的FORWARD\_ONLY、READ\_ONLY游标，如果指定了SCROLL或FOR\_UPDATE，则不能指定FAST\_FORWARD；  大多数情况下FAST\_FORWARD要比FORWARD\_ONLY性能略好； |
| READ\_ONLY(只读) | 游标只能读取数据，不能做任何更新操作； |
| SCROLL\_LOCKS(悲观锁) | 锁定数据集，防止其他事务更改，以确保本事务完整性； |
| OPTIMISTIC(乐观锁) | 不锁定数据集，当需要在游标中更新数据时，如果底层表内数据更新，则游标内数据更新不成功；  如果底层表内数据未更新，则游标内表数据可以更新； |

## 10.02、打开游标

-- 打开游标;

OPEN cursor\_name;

## 10.03、读取数据集

-- 读取数据集;

FETCH NEXT | PRIOR | FIRST | LAST | ABSOLUTE 10/-10 | RELATIVE 1/1

FROM cursor\_name

INTO @variable\_name1, @variable\_name2...

|  |  |
| --- | --- |
| **选项** | **描述** |
| NEXT(下) | 返回首次提取的第一行或下一行，并定位为当前位置； |
| PRIOR(上) | 返回首次提取的无返回或上一行，并定位为当前位置； |
| FIRST(头) | 返回第一行，并定位为当前位置； |
| LAST(尾) | 返回最后一行，并定位为当前位置； |
| ABSOLUTE(绝对) | 正数=游标头的跳N行，并定位为当前位置；  负数=游标尾的跳N行，并定位为当前位置；  0=无返回； |
| RELATIVE(相对) | 正数=当前行的跳N行，并定位为当前位置；  负数=当前行的跳N行，并定位为当前位置；  0=无返回； |

## 10.04、关闭游标

-- 关闭游标;

CLOSE cursor\_name;

## 10.05、释放游标

-- 释放游标;

DEALLOCATE cursor\_name;

## 10.06、游标示例

-- 创建主表;

CREATE TABLE dbo.OrderBasic(

PurchaseId bigint NOT NULL,

ItemQty int NULL,

CONSTRAINT PK\_OrderBasic PRIMARY KEY CLUSTERED(PurchaseId)

)

GO

-- 创建从表;

CREATE TABLE dbo.OrderItem(

ItemId bigint NOT NULL,

Quantity int NULL,

CONSTRAINT PK\_OrderItem PRIMARY KEY CLUSTERED(ItemId)

)

GO

-- 填充数据;

INSERT INTO dbo.OrderBasic VALUES(1, 0);

INSERT INTO dbo.OrderItem VALUES(1, 1000);

INSERT INTO dbo.OrderItem VALUES(2, 2000);

INSERT INTO dbo.OrderItem VALUES(3, 3000);

INSERT INTO dbo.OrderItem VALUES(4, 4000);

INSERT INTO dbo.OrderItem VALUES(5, 5000);

INSERT INTO dbo.OrderItem VALUES(6, 6000);

INSERT INTO dbo.OrderItem VALUES(7, 7000);

INSERT INTO dbo.OrderItem VALUES(8, 8000);

INSERT INTO dbo.OrderItem VALUES(9, 9000);

INSERT INTO dbo.OrderItem VALUES(10, 1001);

GO

-- 声明其他变量;

DECLARE @Quantity INT, @ItemQty INT;

SET @ItemQty = 0;

-- 声明游标;

DECLARE cursor\_OrderItem CURSOR LOCAL FORWARD\_ONLY FAST\_FORWARD READ\_ONLY FOR

SELECT Quantity FROM dbo.OrderItem ORDER BY ItemId DESC;

-- 打开游标;

OPEN cursor\_OrderItem;

-- 读取第一行;

FETCH NEXT FROM cursor\_OrderItem INTO @Quantity;

-- 遍历数据集;

WHILE @@FETCH\_STATUS = 0

BEGIN

-- 累加;

SET @ItemQty += @Quantity;

-- 读取下一行;

FETCH NEXT FROM cursor\_OrderItem INTO @Quantity;

END

-- 关闭游标;

CLOSE cursor\_OrderItem;

-- 释放游标;

DEALLOCATE cursor\_OrderItem;

-- 更新主表;

UPDATE dbo.OrderBasic SET ItemQty = @ItemQty WHERE PurchaseId = 1;

## 10.07、临时表替换游标的解决方案

-- 声明其他变量;

DECLARE @Quantity INT, @ItemQty INT, @ItemId BIGINT;

SET @ItemQty = 0;

-- 声明临时表;

SELECT ItemId, Quantity, Flag = 0

INTO #Temp\_OrderItem

FROM OrderItem ORDER BY ItemId DESC;

-- 读取第一行;

SELECT @ItemId = ItemId, @Quantity = Quantity FROM #Temp\_OrderItem WHERE Flag = 0;

-- 遍历数据集;

WHILE @ItemId > 0

BEGIN

-- 累加;

SET @ItemQty += @Quantity;

-- 更新处理标识;

UPDATE #Temp\_OrderItem SET Flag = 1 WHERE ItemId = @ItemId;

-- 复位;

SET @ItemId = 0;

-- 读取下一行;

SELECT @ItemId = ItemId, @Quantity = Quantity FROM #Temp\_OrderItem WHERE Flag = 0;

END

-- 删除临时表;

DROP TABLE #Temp\_OrderItem;

-- 更新主表;

UPDATE dbo.OrderBasic SET ItemQty = @ItemQty WHERE PurchaseId = 1;

# 11、存储(Storage)

## 11.01、全文目录(Full Text Catalogs)

A、全文搜索功能提供基于字符和二进制数据的智能单词(和短语)搜索，使用全文搜索的执行效率会比普通的LIKE查询好很多；

B、SQL Server 2008版本之后，全文搜索功能集成到数据库中，全文目录不再单独存储在文件系统中，全文索引和查询支持功能也不再依赖于SQL Server早期版本中的分离的MSFTESQL服务；

C、SQL Server 2008版本之后，数据库中全面集成非索引字(以前称为干扰词)，允许创建自己的非索引字表和相关非索引字，早期版本在数据库外部使用干扰词文件；

**全文目录(Full Text Catalog)：**

全文目录可以不包含全文索引，也可以包含数量不等的全文索引，从SQL Server 2008版本之后，全文目录存储在数据库中(在早期的版本中，它存储在SQL Server实例服务器的本地硬盘上)；

全文目录可包含用于在单个数据库内索引一个或多个表的全文索引；

**全文索引(Full Text Index)：**

全文索引能让我们更灵活地搜索非结构化的文本数据，并且提供了比LIKE查询好很多的性能；

-- 创建表(测试环境);

CREATE TABLE dbo.Article(

ArticleId BIGINT NOT NULL,

ArticleTitle NVARCHAR(50) NULL,

[Content] NVARCHAR(MAX) NULL,

CreateTime DATETIME NULL,

[State] INT NULL,

CONSTRAINT PK\_Article PRIMARY KEY CLUSTERED(ArticleId)

)

GO

-- 插入数据;

INSERT INTO dbo.Article VALUES(1, '麦谷科技王洪生：从产品差异到盈利差异的技术赋能', '宇宙旗舰价崩了 曝华为P10成交价52元', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(2, '分析师预测2019-2025年5G固定无线服务CA..', '花式自拍玩出新姿势 华为nova 2系列携手QQ..', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(3, '开学季换新机M3 青春版让你活力四射', '格莱美大师倾力助阵 华为nova 2系列开启古典文..', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(4, 'B&O发布了两款OLED电视 价格是索尼的三倍！', '手机依赖症测测你是几级患者', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(5, 'iPhone 8九月中旬发布，起步价6500', '看好你的女朋友 美图将一口气推三款手机：双摄', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(6, 'Vivo为何能从强手如云的手机市场脱颖而出？', '那些说安卓机越用越卡的 都是因为你不会用', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(7, '科技变化带来工作方式变化 戴尔商用将迎接全新挑战', '智能手机全球三甲缘何死磕旗舰机', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(8, '南京新百上半年扭亏为盈 同比增长136.52％', '三七互娱2017上半年网络游戏营收27.7亿', '2017-09-01 17:01:00', 1);

INSERT INTO dbo.Article VALUES(9, '华为Pay将进军欧洲市场 为高端用户市场铺路', '戴尔发布新型坚固平板电脑，运行Windows 7操..', '2017-09-01 17:01:00', 1);

GO

### 11.01.01、创建全文目录

-- 语法;

CREATE FULLTEXT CATALOG catalog\_name

[ ON FILEGROUP 'filegroup' ]

[ IN PATH 'rootpath']

[ WITH ACCENT\_SENSITIVITY = {ON|OFF} ]

[ AS DEFAULT ]

[ AUTHORIZATION owner\_name ]

|  |  |
| --- | --- |
| **参数** | **描述** |
| **ON FILEGROUP 'filegroup'** | 指定全文目录存放的文件组，未指定则使用数据库默认的文件组； |
| **IN PATH 'rootpath'** | SQL Server 2008版本之后被废弃的选项； |
| **WITH ACCENT\_SENSITIVITY = {ON|OFF}** | 在此全文目录中创建全文索引是否区分重音(重音字符和非重音字符)； |
| **AS DEFAULT** | 将该全文目录定义为默认全文目录，在数据库中创建的、未显式指定全文目录的全文索引将使用该全文目录； |
| **AUTHORIZATION owner\_name** | 该全文目录的所有者，可选择一个数据库用户或者是角色作为owner\_name； |

-- 创建全文目录;

CREATE FULLTEXT CATALOG FULLTEXT\_CATALOG\_1

WITH ACCENT\_SENSITIVITY = ON

AS DEFAULT;

### 11.01.02、修改全文目录

-- 语法;

ALTER FULLTEXT CATALOG catalog\_name

{ REBUILD [WITH ACCENT\_SENSITIVITY = {ON|OFF} ] | REORGANIZE | AS DEFAULT }

|  |  |
| --- | --- |
| **参数** | **描述** |
| **REBUILD** | 重新生成目录(重新填充)； |
| **WITH ACCENT\_SENSITIVITY = {ON|OFF}** | (必须与REBUILD同时使用)在此全文目录中创建全文索引是否区分重音(重音字符和非重音字符)； |
| **REORGANIZE** | 整理碎片； |
| **AS DEFAULT** | 将该全文目录定义为默认全文目录，在数据库中创建的、未显式指定全文目录的全文索引将使用该全文目录； |

-- 重新生成;

ALTER FULLTEXT CATALOG FULLTEXT\_CATALOG\_1

REBUILD WITH ACCENT\_SENSITIVITY = OFF;

-- 设为默认;

ALTER FULLTEXT CATALOG FULLTEXT\_CATALOG\_1

AS DEFAULT;

-- 整理碎片;

ALTER FULLTEXT CATALOG FULLTEXT\_CATALOG\_1

REORGANIZE;

### 11.01.03、查询全文目录状态

-- 当前数据库中全文目录;

SELECT \* FROM sys.fulltext\_catalogs;



-- 当前数据库中所有的全文索引;

SELECT OBJECT\_NAME(object\_id) table\_name, change\_tracking\_state\_desc, stoplist\_id FROM sys.fulltext\_indexes;



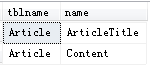
-- 当前数据库中所有的全文索引的列;

SELECT OBJECT\_NAME(ic.object\_id) tblname, c.name

FROM

sys.fulltext\_index\_columns ic INNER JOIN

sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id;



-- 全文目录的区分重音标识(0=不区分重音, 1=区分重音);

SELECT FULLTEXTCATALOGPROPERTY('FULLTEXT\_CATALOG\_1','AccentSensitivity');

-- 全文目录的大小(单位=MB);

SELECT FULLTEXTCATALOGPROPERTY('FULLTEXT\_CATALOG\_1','IndexSize');

-- 全文目录的重组状态(0=未进行重组, 1=正在进行重组);

SELECT FULLTEXTCATALOGPROPERTY('FULLTEXT\_CATALOG\_1','MergeStatus');

-- 全文目录的填充状态(0=空闲, 1=正在填充, 2=暂停, 7=正在生成索引, 8=磁盘满);

SELECT FULLTEXTCATALOGPROPERTY('FULLTEXT\_CATALOG\_1','PopulateStatus');

### 11.01.04、删除全文目录

-- 语法;

DROP FULLTEXT CATALOG catalog\_name;

-- 删除全文目录(失败: 无法删除全文目录'FULLTEXT\_CATALOG\_1', 因为它含有全文检索);

DROP FULLTEXT CATALOG FULLTEXT\_CATALOG\_1;

-- 先删除全文索引(全文检索);

DROP FULLTEXT INDEX ON dbo.Article;

-- 再删除全文目录(成功, 但警告: 正在删除全文目录'FULLTEXT\_CATALOG\_1', 当前它被设置为默认值);

DROP FULLTEXT CATALOG FULLTEXT\_CATALOG\_1;

### 11.01.05、创建全文索引

A、要建立全文索引的表列，它的数据类型必须符合：VARCHAR/NVARCHAR/CHAR/NCHAR/XML/VARBINARY/TEXT/NTEXT/IMAGE；

B、表必须存在主键或至少一个UNIQUE约束的列；

-- 语法;

CREATE FULLTEXT INDEX ON table\_name

[ ( { column\_name [ TYPE COLUMN type\_column\_name ] [ LANGUAGE language\_term ] } [ ,...n] ) ]

KEY INDEX index\_name [ ON fulltext\_catalog\_name] [ WITH [ ( ] <with\_option> [ ,...n] [ ) ] ]

[;]

<with\_option>::=

{

CHANGE\_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO POPULATION ] } |

STOPLIST [ = ] { OFF | SYSTEM | stoplist\_name }

}

|  |  |
| --- | --- |
| **参数** | **描述** |
| **TYPE COLUMN type\_column\_name** | 使用什么文件类型来保存指定的表列，表列的数据类型为VARBINARY(MAX)/IMAGE时，才有必要指定全文索引的存储格式(FILEEXTENSION)； |
| **LANGUAGE language\_term** | (可选项)指定语言，可在sys.fulltext\_languages表中查询所有支持的语言；  SELECT \* FROM sys.fulltext\_languages; |
| **ON fulltext\_catalog\_name** | 指定存储在哪个全文目录，如果某全文目录AS DEFAULT，则不用指定(可选)； |
| **CHANGE\_TRACKING** | 指定全文服务如何检测数据变化； |
| **CHANGE\_TRACKING = MANUAL** | 手动跟踪更改，或SQL Server代理按计划跟踪更改； |
| **CHANGE\_TRACKING = AUTO** | 自动跟踪更改(全文索引随表数据的更改而自动刷新更改)； |
| **CHANGE\_TRACKING = OFF** | 禁用跟踪更改； |
| **CHANGE\_TRACKING = OFF, NO POPULATION** | 仅在执行ALTER FULLTEXT INDEX后才会被填充； |
| **STOPLIST** | 非索引字表包含被搜索忽略的字符串/词； |
| **STOPLIST = OFF** | 关闭非索引字表； |
| **STOPLIST = SYSTEM** | 系统默认非索引字表； |
| **STOPLIST = stoplist\_name** | 用户自定义的非索引字表； |

-- 创建全文索引;

CREATE FULLTEXT INDEX ON dbo.Article

(

ArticleTitle LANGUAGE 2052,

[Content] LANGUAGE 2052

)

KEY INDEX PK\_Article

ON FULLTEXT\_CATALOG\_1

WITH CHANGE\_TRACKING AUTO, STOPLIST = SYSTEM;

-- 查询分词;

SELECT \* FROM dbo.Article WHERE FREETEXT(\*, '宇宙 Vivo 科技');

SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '科技');

### 11.01.06、修改全文索引

-- 语法;

ALTER FULLTEXT INDEX ON table\_name

{

ENABLE | DISABLE | SET CHANGE\_TRACKING { MANUAL | AUTO | OFF } |

ADD ( column\_name [ TYPE COLUMN type\_column\_name ] [ LANGUAGE language\_term ] [,...n] ) [ WITH NO POPULATION ] |

DROP ( column\_name [,...n] ) [WITH NO POPULATION ] |

START { FULL | INCREMENTAL | UPDATE } POPULATION |

{ STOP | PAUSE | RESUME } POPULATION |

SET STOPLIST { OFF| SYSTEM | stoplist\_name } [WITH NO POPULATION]

}

|  |  |
| --- | --- |
| **参数** | **描述** |
| **ENABLE** | 启用全文索引(全文检索)； |
| **DISABLE** | 禁用全文索引(全文检索)，即使禁用，但之前的旧全文索引记录仍然可以根据条件进行查询； |
| **SET CHANGE\_TRACKING { MANUAL | AUTO | OFF }** | 手动跟踪更改/自动跟踪更改/禁用跟踪更改； |
| **ADD ( column\_name**  **[ TYPE COLUMN type\_column\_name ]**  **[ LANGUAGE language\_term ] [,...n] )**  **[ WITH NO POPULATION ]** | 添加需要作全文索引的列； |
| **DROP ( column\_name [,...n] )**  **[WITH NO POPULATION ]** | 删除全文索引中的列； |
| **START { FULL | INCREMENTAL | UPDATE } POPULATION** | FULL(启用完全填充)：重新刷新所有全文索引；  INCREMENTAL(启用增量填充)：刷新最后一次填充后的修改的行(表中必须存在一个timestamp列)；  UPDATE：刷新最后一次填充后的所有插入/更新/删除的行； |
| **{ STOP | PAUSE | RESUME } POPULATION** | 停止/暂停填充； |
| **SET STOPLIST { OFF| SYSTEM | stoplist\_name } [WITH NO POPULATION]** | 关闭/系统默认/用户自定义非索引字表； |

-- 增加表列到全文索引;

ALTER FULLTEXT INDEX ON dbo.Article

ADD(Title2);

-- 完全填充;

ALTER FULLTEXT INDEX ON dbo.Article

START FULL POPULATION;

-- 删除全文索引中的表列;

ALTER FULLTEXT INDEX ON dbo.Article

DROP(Title2);

### 11.01.07、删除全文索引

-- 语法;

DROP FULLTEXT INDEX ON table\_name;

-- 删除全文索引;

DROP FULLTEXT INDEX ON dbo.Article;

### 11.01.08、搜索

|  |  |
| --- | --- |
| **匹配方式** | **示例** |
| **FREETEXT** | -- 查询分词(按变形单词/字面单词/同义单词的匹配方式);  SELECT \* FROM dbo.Article WHERE FREETEXT(\*, '宇宙 Vivo 科技');  SELECT \* FROM dbo.Article WHERE FREETEXT(ArticleTitle, '科技'); |
| **CONTAINS** | -- 查询分词(按字面单词的匹配方式);  SELECT \* FROM dbo.Article WHERE CONTAINS(\*, '"宇宙" OR "Vivo" OR "科技"');  SELECT \* FROM dbo.Article WHERE CONTAINS(\*, '"戴尔" AND "科技"');  SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '科技'); |
| **FREETEXTTABLE** | 根据分词按排名次序，返回搜索结果; |
| **CONTAINSTABLE** | 根据加权模式匹配，返回搜索结果; |

## 11.02、分区方案(Partition Schemes)

具体请参考：D05.SQLSERVER数据库的海量数据处理机制(Massive Data Processing Mechanism).docx；

## 11.03、分区函数(Partition Functions)

具体请参考：D05.SQLSERVER数据库的海量数据处理机制(Massive Data Processing Mechanism).docx；

## 11.04、全文非索引字表(Full Text Stoplists)

### 11.04.01、创建全文非索引字表

-- 语法;

CREATE FULLTEXT STOPLIST stoplist\_name

[ FROM { [ database\_name. ] source\_stoplist\_name } | SYSTEM STOPLIST ]

[ AUTHORIZATION owner\_name ];

|  |  |
| --- | --- |
| **参数** | **描述** |
| **FROM { [ database\_name. ] source\_stoplist\_name }** | 引用某数据库的既有的全文非索引字表； |
| **SYSTEM STOPLIST** | 复制系统默认的全文非索引字表； |
| **AUTHORIZATION owner\_name** | 所有者，可选择一个数据库用户或者是角色作为owner\_name； |

-- 创建空的全文非索引字表;

CREATE FULLTEXT STOPLIST STOPLIST\_1;

### 11.04.02、修改全文非索引字表

-- 语法;

-- language\_term: SELECT \* FROM sys.fulltext\_languages;

ALTER FULLTEXT STOPLIST stoplist\_name

{

ADD 'stopword' LANGUAGE language\_term |

DROP { 'stopword' LANGUAGE language\_term | ALL LANGUAGE language\_term | ALL }

};

-- 添加新的非索引字到字表中;

ALTER FULLTEXT STOPLIST STOPLIST\_1

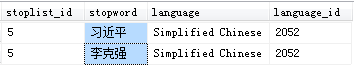
ADD '习近平' LANGUAGE 2052;

ALTER FULLTEXT STOPLIST STOPLIST\_1

ADD '李克强' LANGUAGE 2052;

-- (检验)查询当前数据库的非索引字表;

SELECT \* FROM sys.fulltext\_stopwords;



-- 删除非索引字表中的某非索引字;

ALTER FULLTEXT STOPLIST STOPLIST\_1

DROP '李克强' LANGUAGE 2052;

-- (还原测试环境)先删除全文索引(全文检索);

DROP FULLTEXT INDEX ON dbo.Article;

-- (还原测试环境)再删除全文目录(成功, 但警告: 正在删除全文目录'FULLTEXT\_CATALOG\_1', 当前它被设置为默认值);

DROP FULLTEXT CATALOG FULLTEXT\_CATALOG\_1;

-- 创建全文目录;

CREATE FULLTEXT CATALOG FULLTEXT\_CATALOG\_1

WITH ACCENT\_SENSITIVITY = ON;

-- 创建全文索引(绑定非索引字到字表);

CREATE FULLTEXT INDEX ON dbo.Article

(

ArticleTitle LANGUAGE 2052,

[Content] LANGUAGE 2052

)

KEY INDEX PK\_Article

ON FULLTEXT\_CATALOG\_1

WITH CHANGE\_TRACKING AUTO, STOPLIST = STOPLIST\_1;

-- (检验绑定)当前数据库的全文非索引字表;

SELECT \* FROM sys.fulltext\_stoplists;



-- (检验绑定)当前数据库中所有的全文索引;

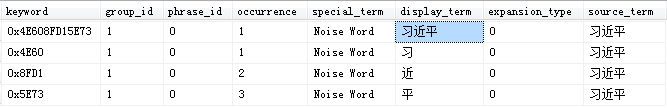
SELECT OBJECT\_NAME(object\_id) table\_name, change\_tracking\_state\_desc, stoplist\_id FROM sys.fulltext\_indexes;



-- (检验绑定)新非索引字是否已被全文引擎识别;

-- 语法: SELECT \* FROM sys.dm\_fts\_parser('query\_string', lcid, stoplist\_id, accent\_sensitivity);

SELECT \* FROM sys.dm\_fts\_parser('习近平', 2052, 5, 1);



-- (检验绑定)匹配返回空的数据集(记录);

SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '习近平');

SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '习');

SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '近');

SELECT \* FROM dbo.Article WHERE CONTAINS(ArticleTitle, '平');

### 11.04.03、删除全文非索引字表

-- 删除非索引字表(失败: 无法删除全文非索引字表'STOPLIST\_1', 因为当前至少有一个全文检索正在使用它);

DROP FULLTEXT STOPLIST STOPLIST\_1;

-- 解除绑定;

ALTER FULLTEXT INDEX ON dbo.Article

SET STOPLIST SYSTEM;

-- 删除非索引字表(成功);

DROP FULLTEXT STOPLIST STOPLIST\_1;

## 11.05、搜索属性列表(Search Property Lists)

具体请参考：<https://docs.microsoft.com/zh-cn/sql/relational-databases/search/search-document-properties-with-search-property-lists>

# …12、安全性(Security)

## 12.01、用户(Users)

触…行；

## 12.02、角色(Roles)

触…行；

## 12.03、架构(Schemas)

触…行；

## 12.04、非对称秘钥(Asymmetric Keys)

触…行；

## 12.05、证书(Certificates)

触…行；

## 12.06、对称秘钥(Symmetric Keys)

触…行；

## 12.07、数据库审核规范(Database Audit Specifications)

触…行；