PostgreSQL 与 MS SQLServer在过程语言中的差异

# 数据类型

## 数字类型

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sql server | |  |  | | --- | --- | | Numeric/  decimal | 精确数值型 从-10 的38 次方-1，到10 的38 次方-1 | | bit | 整型 其值只能是0、1 或空值 | | int | 整型 -2 的31 次方到 2 的31 次方 | | smallint | 整型 -2 的15 次方到 2 的15 次方 | | tinyint | 0 到255 之间的整数 | | float | 浮点数 | | real | 浮点数 | |
| postgreSQL | |  |  | | --- | --- | | Numeric(p,s)/  Decimal(p,s) | 任意精度数值，p必须为正数，s可以为零或者正数 | | smallint | 2字节小范围整数，-32768到+32767 | | Integer/  Int | 4字节常用的整数，-2147483648到+2147483647 | | bigint | 8字节大范围的整数，-9223372036854775808到9223372036854775807 | | real | 4字节变精度，不精确6位十进制数字精度 | | double precision | 8字节变精度，不精确15位十进制数字精度 | | serial | 4字节自增整数，1到+2147483647 | | bigserial | 8字节大范围的自增整数，1到9223372036854775807 | |

## 字符类型

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sql server | |  |  | | --- | --- | | char | 定长，最大8000 | | varchar | 变长，SQL Server2005后可以通过 varchar(MAX)来允许最大存储2G 的数据 | | text | 最多存储有2G 字符 | | nchar | 定长，最大4000 国际化字符 | | nvarchar | 变长。 | | ntext | 最多存储有1G 国际化字符 | |
| postgreSQL | |  |  | | --- | --- | | character varying(n) varchar(n) | 变长，有长度限制 | | character(n)  char(n) | 定长，不足补空白 | | text | 变长，无长度限制 | | "char" | 1 字节，单字节内部类型 | | name | 64字节，用于对象名的内部类型 | |

## 日期类型

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sql server | |  |  | | --- | --- | | time | 格式: hh:mm:ss[.nnnnnnn],  00:00:00.0000000 到 23:59:59.9999999,  固定5个字节 | | date | 格式: YYYY-MM-DD,  0001-01-01 到 9999-12-31,  固定3个字节 | | datetime2 | 格式: YYYY-MM-DD hh:mm:ss[.fractional seconds],  0001-01-01 到 9999-12-31,  精度小于3时为6个字节；精度为3和4时为7个字节。所有其他精度则需要8个字节。 | |
| postgreSQL | |  |  | | --- | --- | | timestamp[(p)] [without time zone] | 包括日期和时间  8字节  精确到1毫秒/14位 | | timestamp[(p)] with time zone | 日期和时间  8字节  精确到1毫秒/14位 | | interval[(p)] | 时间间隔  12 字节  精确到1毫秒/14位 | | date | 只用于日期  4字节  精确到1天 | | time[(p)] [without time zone] | 只用于一日内时间  8字节  精确到1毫秒/14位 | |

## 伪类型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| postgreSQL | |  |  | | --- | --- | | void | 应用于过程语言(PL/pgSQL)编写的函数  表示一个函数不返回数值 | | record | 应用于过程语言(PL/pgSQL)编写的函数  标识一个函数返回一个未声明的行类型 | |

# 变量定义与赋值

|  |  |
| --- | --- |
| Sql server | 3> DECLARE  4> **@testvalue AS VARCHAR(20);**  5> BEGIN  6> **SET @testvalue = 'First Test!';**  7> PRINT( @testvalue );  8> END;  9> go |
| postgreSQL | SQL> DECLARE  2 **testvalue VARCHAR(20);**  3 BEGIN  4 **testvalue := 'First Test!';**  5 RAISE NOTICE 'Testvalue is %', testvalue;  6 END; |

# IF/ELSE控制结构

|  |  |
| --- | --- |
| Sql server | 1> DECLARE  2> @testvalue AS INT;  3> BEGIN  4> SET @testvalue = 100;  5>  6> IF @testvalue > 100  7> BEGIN  8> PRINT( '100+' );  9> END  10> ELSE IF @testvalue = 100  11> BEGIN  12> PRINT( '100' );  13> END  14> ELSE  15> BEGIN  16> PRINT( '100-' );  17> END  18>  19> END;  20> go |
| postgreSQL | IF ... THEN … END IF;  IF ... THEN ... ELSE … END IF;  IF ... THEN ... ELSE IF … THEN … END IF;  IF ... THEN ... ELSIF ... THEN ... ELSE … END IF;  IF ... THEN ... ELSEIF ... THEN ... ELSE … END IF;  （注：ELSEIF 是 ELSIF 的别名）  1 DECLARE  2 testvalue INT;  3 BEGIN  4 testvalue := 100;  5  6 IF testvalue > 100 THEN  7 RAISE NOTICE '100+';  8 ELSIF testvalue = 100 THEN  9 RAISE NOTICE '100';  10 ELSE  11 RAISE NOTICE '100-';  12 END IF;  13  14 END; |

# case when 结构

|  |  |
| --- | --- |
| Sql server  注：没有decode函数 | Case具有两种格式。简单Case函数和Case搜索函数。  --简单Case函数  CASE sex  WHEN '1' THEN '男'  WHEN '2' THEN '女'  ELSE '其他' END  --Case搜索函数  CASE  WHEN sex = '1' THEN '男'  WHEN sex = '2' THEN '女'  ELSE '其他' END |
| postgreSQL  注：没有decode函数 | 条件:  case s.xxx  when 'A'  then (case when S.×× is null then '' else '' end)  when 'B'  then( case when S.×× is null then '' else '' end)  when 'C'  then( case when S.×× is null then '' else '' end)  else s.order\_class end; |

# 循环结构

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sql server | |  |  | | --- | --- | | WHILE | 1> DECLARE  2> @testvalue AS INT;  3> BEGIN  4> SET @testvalue = 0;  5>  6> WHILE @testvalue < 5  7> BEGIN  8> SET @testvalue = @testvalue + 1;  9> PRINT ( @testvalue );  10> END  11>  12> END;  13> go | | FOR | 不支持 | | Break 与 Continue | 1> DECLARE  2> @testvalue AS INT;  3> BEGIN  4> SET @testvalue = 0;  5> WHILE @testvalue < 5  6> BEGIN  7> SET @testvalue = @testvalue + 1;  8> IF @testvalue = 2  9> BEGIN  10> CONTINUE;  11> END;  12> IF @testvalue = 4  13> BEGIN  14> BREAK;  15> END;  16> PRINT ( @testvalue );  17> END  18> END;  19> go | |
| postgreSQL | |  |  | | --- | --- | | WHILE | [ <<label>> ]  WHILE expression LOOP  statements  END LOOP [ label ];  例如:  WHILE amount\_owed > 0 AND gift\_certificate\_balance > 0 LOOP  -- 可以在这里做些计算  END LOOP;  WHILE NOT BOOLEAN\_expression LOOP  -- 可以在这里做些计算  END LOOP; | | LOOP | [ <<label>> ]  LOOP  statements  END LOOP [ label ]; | | EXIT | EXIT [ label ] [ WHEN expression ];  例如:  Loop 循环  If … then 条件判断  Exit ; 条件成立，则退出循环。  End if;  End loop; | | CONTINUE | CONTINUE [ label ] [ WHEN expression ];  例如:  LOOP  一些计算  EXIT WHEN count > 100;  CONTINUE WHEN count < 50;  一些在count 数值在 [50 .. 100] 里面时候的计算  END LOOP; | | FOR（整数变种） | [ <<label>> ]  FOR name IN [ REVERSE ] expression .. expression LOOP  statements  END LOOP [ labal ];  例子∶  FOR i IN 1..10 LOOP --表示1循环到10  --这里可以放一些表达式  RAISE NOTICE 'i IS %', i;  END LOOP;  FOR i IN REVERSE 10..1 LOOP  --这里可以放一些表达式  END LOOP; | |

# 存储过程（函数）

## 创建时的基本定义与调用

|  |  |  |
| --- | --- | --- |
| 存储过程 | Sql server | 1> CREATE PROCEDURE HelloWorld AS  2> BEGIN  3> PRINT 'Hello World';  4> END;  5> go  1> DECLARE @RC int  2> EXECUTE @RC = HelloWorld  3> PRINT @RC  4> go  Hello World  0 |
| postgreSQL  注：  如果没有返回值，则使用：**returns void**  **$body$**可以简写为**$$** | Create or replace **function** 过程名(参数名 参数类型,…..) **returns** 返回值类型 as  **$body$**  **//声明变量**  Declare  变量名 变量类型；  如：  flag Boolean;  变量赋值方式（变量名 类型 ：=值；）  如：  Str text :=值; / str text; str :=值；  Begin  函数体；  End;  **$body$**  **Language plpgsql;** |
| 函数 | Sql server  注：如果没有那个dbo.  可能报错 | 3> CREATE FUNCTION HelloWorld4()  4> RETURNS VARCHAR(20)  5> AS  6> BEGIN  7> RETURN 'Hello World!';  8> END  9> go  1> SELECT dbo.HelloWorld4()  2> go  --------------------  Hello World! |
| postgreSQL  **注：函数可以重构；根据参数类型自动匹配调用对应的函数。** | （同存储过程，postgresql中没有procedure） |

## 修改存储过程（函数）

|  |  |
| --- | --- |
| Sql server | 5> **ALTER PROCEDURE** HelloWorld AS  6> BEGIN  7> PRINT 'Hello World V2.0';  8> END;  9> go  1> DECLARE @RC int  2> EXECUTE @RC = HelloWorld  3> PRINT @RC  4> go  Hello World V2.0  0 |
| postgreSQL  注: 修改后重新编译即可 | **CREATE OR REPLACE FUNCTION** isint(v\_str varchar)  RETURNS int  AS $$  DECLARE  V\_NUM numeric;  BEGIN  --**直接修改函数体**  END;  $$ LANGUAGE plpgsql; |

## 参数定义

|  |  |
| --- | --- |
| Sql server  注：  如果参数的  VARCHAR 不指定长  度，那么 返回可能  就是  HelloE! | 1> CREATE PROCEDURE HelloWorld1  2> **@UserName VARCHAR(10)**  3> AS  4> BEGIN  5> PRINT 'Hello' + @UserName + '!';  6> END;  7> go  1> DECLARE @RC int;  2> EXECUTE @RC = HelloWorld1 'Edward' ;  3> PRINT @RC;  4> go  HelloEdward!  0 |
| postgreSQL | CREATE OR REPLACE FUNCTION isint(**v\_str varchar**)  RETURNS int  AS $$  DECLARE  V\_NUM numeric;  BEGIN  --函数体  END;  $$ LANGUAGE plpgsql; |

## IN、OUT、IN OUT

|  |  |
| --- | --- |
| Sql server  注：  SQL Server 的  OUTPUT 需要写在  变量数据类型后  面。  SQL Server 没有  IN OUT 关键字  OUTPUT 已经相当  于 IN OUT 了。 | 1> CREATE PROCEDURE HelloWorld2  2> @UserName VARCHAR(10),  3> @OutVal VARCHAR(10) **OUTPUT**,  4> @InoutVal VARCHAR(10) **OUTPUT**  5> AS  6> BEGIN  7> PRINT 'Hello ' + @UserName + @InoutVal + '!';  8> SET @OutVal = 'A';  9> SET @InoutVal = 'B';  10> END;  11> go  1>  2> DECLARE @RC int, @OutVal VARCHAR(10), @InoutVal VARCHAR(10);  3> BEGIN  4> SET @InoutVal = '~Hi~';  5> EXECUTE @RC = HelloWorld2 'Edward', @OutVal **OUTPUT**, @InoutVal  **OUTPUT**;  6> PRINT @RC;  7> PRINT '@OutVal=' + @OutVal;  8> PRINT '@InoutVal=' + @InoutVal;  9> END  10> go  Hello Edward~Hi~!  0  @OutVal=A  @InoutVal=B |
| postgreSQL  注： Out 需要写在变量数据类型前面；可以省去returns部分。 | CREATE OR REPLACE FUNCTION p\_gisrs2iodn\_vendor(  v\_vendorNameCn VARCHAR(255),  **OUT** n\_vendorId BIGINT)  AS $$  declare  n\_seq BIGINT;  BEGIN  --函数部分  END;  $$ LANGUAGE plpgsql; |

## 参数的默认值

|  |  |
| --- | --- |
| Sql server | 1> CREATE PROCEDURE HelloWorld3  2> @UserName VARCHAR(10),  3> @Val1 VARCHAR(20) **= ' Good Moning,',**  4> @Val2 VARCHAR(20) **= ' Nice to Meet you'**  5> AS  6> BEGIN  7> PRINT 'Hello ' + @UserName + @Val1 + @Val2 + '!';  8> END;  9> go  1>  2> DECLARE @RC int;  3> BEGIN  4> EXECUTE @RC = **HelloWorld3 'Edward';**  5> PRINT @RC;  6> EXECUTE @RC = **HelloWorld3 'Edward', ' Good Night,';**  7> PRINT @RC;  8> EXECUTE @RC = **HelloWorld3 'Edward', ' Good Night,', ' Bye';**  9> PRINT @RC;  10> END  11> go  Hello Edward Good Moning, Nice to Meet you!  0  Hello Edward Good Night, Nice to Meet you!  0  Hello Edward Good Night, Bye!  0 |
| postgreSQL  注： 在8.4版本上测试可行 | CREATE OR REPLACE FUNCTION dropprocedure(  v\_prc\_like varchar **default '\_return\_ids'**)  RETURNS void  AS $$  DECLARE  v\_cnt int;  v\_sql varchar(255);  cur record;  BEGIN  --函数部分  END;  $$ LANGUAGE plpgsql; |

## 返回结果集的函数

|  |  |
| --- | --- |
| Sql server  SQL Server 通过  RETURN TABLE 来  实现。 | 1> CREATE FUNCTION getHelloWorld()  2> **RETURNS TABLE**  3> AS  4> RETURN  5> SELECT 'Hello' AS A, 'World' AS B;  6> GO  1> SELECT \* FROM getHelloWorld();  2> go  A B  ----- -----  Hello World  (1 行受影响) |
| postgreSQL  注：行变量类型使用record来定义。 | CREATE OR REPLACE FUNCTION f\_querydata(v\_col varchar(255))  Returns **SETOF record**  AS $$  DECLARE  v\_sql text;  **v\_rec record;**  Begin  v\_sql :=  'SELECT \* FROM TABLE\_NAME WHERE colname = ' || v\_col;  **FOR v\_rec IN execute v\_sql** Loop --**隐性游标, 使用FOR可以避免定义多个变量。**  **RETURN NEXT v\_rec;**  END LOOP;    END;  $$ LANGUAGE plpgsq; |

## 在存过中访问或调用函数的方法

|  |  |
| --- | --- |
| postgreSQL | 第一：returns void 的函数： PERFORM p\_testfun(v1,v2,…);  第二：returns type 的函数： 例如：  --函数  CREATE OR REPLACE FUNCTION p\_testfun(v1 BIGINT, v2 VARCHAR(255))  RETURNS varchar  AS $$  Declare  v\_rn varchar2(255);  BEGIN  --处理  return v\_rn;  END;  $$ LANGUAGE plpgsql;  --调用  Declare  v\_tt varchar(255);  Begin  v\_tt := p\_testfun(v1,v2);  End;  第三：使用单个out参数定义返回的函数： 例如：  --函数  CREATE OR REPLACE FUNCTION p\_testfun(v1 BIGINT, **out** v2 VARCHAR(255))  AS $$  Declare  v\_rn varchar2(255);  BEGIN  --处理  v2 := v\_rn;  END;  $$ LANGUAGE plpgsql;  --调用  Declare  v\_tt varchar(255);  Begin  v\_tt := p\_testfun(v1);  End;  第四：使用多个out参数定义返回的函数：  例如：  --函数  CREATE OR REPLACE FUNCTION p\_testfun(v1 BIGINT, **out** v2 st\_geometry, **out** v3 numeric(28, 20), **out** v4 numeric(28, 20))  AS $$  Declare  v\_rn varchar2(255);  BEGIN  --处理  END;  $$ LANGUAGE plpgsql;  --调用  Declare  v\_tt1 st\_geometry;  v\_tt2 numeric(28, 20);  v\_tt2 numeric(28, 20);  Begin  select t1.v2, t1.v3, t1.v4  into v\_tt1, v\_tt2, v\_tt3  from p\_testfun(v1) as t1;  End; |

# cursor游标定义、使用

|  |  |
| --- | --- |
| Sql server  注： | CREATE TABLE test\_main (  id INT,  value VARCHAR(10),  PRIMARY KEY(id)  );  INSERT INTO test\_main(id, value) VALUES (1, 'ONE');  INSERT INTO test\_main(id, value) VALUES (2, 'TWO');  INSERT INTO test\_main(id, value) VALUES (3, 'THREE');  1> DECLARE  2> @id INT, @value VARCHAR(10);  3> BEGIN  4> -**- 定义游标.**  **5> DECLARE c\_test\_main CURSOR FAST\_FORWARD FOR**  **6> SELECT id, value FROM test\_main;**  7> **-- 打开游标.**  **8> OPEN c\_test\_main;**  9> -- 填充数据.  10> FETCH NEXT FROM c\_test\_main INTO @id, @value;  11> -- 假如检索到了数据，才处理.  12> WHILE **@@fetch\_status = 0**  13> BEGIN  14> PRINT @value;  16> FETCH NEXT FROM c\_test\_main INTO @id, @value;  17> END;  18> **-- 关闭游标**  **19> CLOSE c\_test\_main;**  **20> -- 释放游标.**  **21> DEALLOCATE c\_test\_main;**  22> END;  23> go  ONE  TWO  THREE |
| postgreSQL  注：行变量类型使用record来定义。 | CREATE OR REPLACE FUNCTION f\_querydata(v\_col varchar(255))  Returns SETOF record  AS $$  DECLARE  v\_sql text;  **v\_rec record;**  Begin  v\_sql :=  'SELECT \* FROM TABLE\_NAME WHERE colname = ' || v\_col;  **FOR v\_rec IN execute v\_sql** Loop  **RETURN NEXT v\_rec;**  END LOOP;    END;  $$ LANGUAGE plpgsq; |

# 隐性游标for语句的使用

|  |  |
| --- | --- |
| Sql server | 不支持 |
| postgreSQL  注：行变量类型使用record来定义。 | CREATE OR REPLACE FUNCTION f\_querydata(v\_col varchar(255))  Returns SETOF record  AS $$  DECLARE  v\_sql text;  **v\_rec record;**  Begin  v\_sql :=  'SELECT \* FROM TABLE\_NAME WHERE colname = ' || v\_col;  **FOR v\_rec IN execute v\_sql** Loop --**隐性游标, 使用FOR可以避免定义多个变量。**  RETURN NEXT v\_rec;  END LOOP;    END;  $$ LANGUAGE plpgsq; |

# 日志输出RAISE NOTICE语句的使用

|  |  |
| --- | --- |
| Sql server | PRINT ' Hello World!';  PRINT 'Hello ' + @UserName + @Val1 + @Val2 + '!'; --变量 |
| postgreSQL | RAISE NOTICE ' Hello World!';  RAISE NOTICE '%: 对象不是表!', $1; --函数参数  RAISE NOTICE 'SEQUENCE: **%** START WITH : **%**', **v\_seq\_name, v\_maxid+1; --变量** |

# 异常处理EXCEPTION的使用

|  |  |
| --- | --- |
| Sql server  错误级别从 0 到  25， 其中 19 到  25 是重大错误级  别。  注：小于 0 的级别  被解释为 0，大于  25 的级别被解释  为 25。  19 到 25 级别的  错误， 只能由  sysadmin 角色的  成员用 WITH LOG  选项指定。  19 到 25 级别的  错误，将记录到错  误日志和应用程序  日志。  20 到 25 级别的  错误，被认为是致  命的；遇到致命的  级别错误，客户端  的连接将在收到消  息后终止。 | 1> DECLARE  2> @test\_val INT;  **3> BEGIN TRY**  4> -- 导致一个 除零错误  5> SET @test\_val = 1024 / 0;  6> PRINT(STR(@test\_val));  **7> END TRY**  **8> BEGIN CATCH**  9> PRINT('Exception Happen!');  **10> END CATCH**  11> go  Exception Happen!  1> DECLARE  2> @test\_val INT;  3> BEGIN TRY  4> -- 导致一个 除零错误  5> SET @test\_val = 1024 / 0;  6> PRINT(STR(@test\_val));  7> END TRY  8> BEGIN CATCH  9> PRINT('错误代码 = ' + STR(**ERROR\_NUMBER()**));  10> PRINT('错误严重级别 = ' + STR(**ERROR\_SEVERITY()**));  11> PRINT('错误状态代码 = ' + STR(**ERROR\_STATE()**));  12> PRINT('错误信息 = ' + **ERROR\_MESSAGE()**);  13> END CATCH  14> go  错误代码 = 8134  错误严重级别 = 16  错误状态代码 = 1  错误信息 = 遇到以零作除数错误。 |
| postgreSQL | CREATE OR REPLACE FUNCTION p\_GET\_EXCEPTION\_TEST(v\_flag int)  RETURNS void  AS $$  DECLARE  --定义变量  BEGIN  BEGIN  --函数部分  **EXCEPTION** **--捕获异常**  **WHEN OTHERS THEN --异常处理**  RAISE NOTICE 'SQLSTATE: %', **SQLSTATE**; **--异常内部代码**  RAISE NOTICE 'SQLERRM: %', **SQLERRM**; **--异常详细信息**  END;  END;  $$ LANGUAGE plpgsql; |

# like条件语句的使用

|  |  |
| --- | --- |
| Sql server | 语法  match\_expression [ NOT ] LIKE pattern [ ESCAPE escape\_character ]  match\_expression [ NOT ] LIKE pattern [ ESCAPE escape\_character ]参数  match\_expression  任何字符串数据类型的有效 SQL Server 表达式。  pattern  match\_expression 中的搜索模式，可以包含下列有效  SQL Server 通配符。  通配符 描述示例  % 包含零个或更多字符的任意字符串。  \_（下划线） 任何单个字符。  [ ] 指定范围 ([a-f]) 或集合 ([abcdef]) 中的任何单个字符。  [^] 不属于指定范围  WHERE au\_lname LIKE 'de[^l]%' 将查找以 de 开始且其后的字母不为 l 的所有作者的姓氏。  escape\_character  字符串数据类型分类中的所有数据类型的任何有效SQL Server表达式。  escape\_character 没有默认值，且必须仅包含一个字符。  结果类型  Boolean  结果值  如果 match\_expression 匹配指定模式，LIKE 将返回 TRUE。 |
| postgreSQL | 语法:  string LIKE pattern [ESCAPE escape-character]  string NOT LIKE pattern [ESCAPE escape-character]  每个 pattern 定义一个字符串的集合。如果该 string 包含在 pattern 代表的字符串集合里，那么 LIKE 表达式返回真。和我们想像的一样，如果 LIKE 返回真，那么 NOT LIKE 表达式将返回假，反之亦然。一个等效的表达式是 NOT (string LIKE pattern) 。  如果 pattern 不包含百分号或者下划线，那么该模式只代表它本身；这时候 LIKE 的行为就像等号操作符。在 pattern 里的下划线(\_)匹配任何单个字符；而一个百分号(%)匹配零或多个任何字符。  一些例子：  'abc' LIKE 'abc' true  'abc' LIKE 'a%' true  'abc' LIKE '\_b\_' true  'abc' LIKE 'c' false  LIKE 模式匹配总是覆盖整个字符串。要匹配在字符串内部任何位置的序列，该模式必须以百分号开头和结尾。  要匹配下划线或者百分号本身，在 pattern 里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜杠，但是你可以用 ESCAPE 子句指定一个。要匹配逃逸字符本身，写两个逃逸字符。  请注意反斜杠在字符串文本里已经有特殊含义了，所以如果你写一个包含反斜杠的模式常量，那你就要在 SQL 语句里写两个反斜杠。因此，写一个匹配单个反斜杠的模式实际上要在语句里写四个反斜杠。你可以通过用 ESCAPE 选择一个不同的逃逸字符来避免这样；这样反斜杠就不再是 LIKE 的特殊字符了。但仍然是字符文本分析器的特殊字符，所以你还是需要**两个反斜杠。**  我们也可以通过写成 ESCAPE '' 的方式关闭逃逸机制，这时，我们就不能关闭下划线和百分号的特殊含义。  关键字 ILIKE 可以用于替换 LIKE ，令该匹配就当前的区域设置是大小写无关的。这个特性不是 SQL 标准，是 PostgreSQL 扩展。  操作符 ~~ 等效于 LIKE ，而 ~~\* 等效于 ILIKE 。还有 !~~ 和 !~~\* 操作符分别代表 NOT LIKE 和 NOT ILIKE 。所有这些操作符都是 PostgreSQL 特有的.  例子: 查找数据库中表的名字(**查找以tbl\_开头的表的名字)**  select tablename from pg\_tables  where tablename **like 'tbl!\_%'ESCAPE '!';**  利用escape转义: 上面的sql语句中就是将!作为转义字符,作用是将"\_"转换成普通的"\_" |

# 事务概念与sqlserver的差异

|  |  |
| --- | --- |
| Sql server | BeginTrans用于开始一个事物；RollbackTrans用于回滚事务；CommitTrans用于提交所有的事务处理结果，即确认事务的处理。  事务处理可以将一组操作视为一个整体，只有全部语句都成功执行后，事务处理才算成功；若其中有一个语句执行失败，则整个处理就算失败，并恢复到处里前的状态。  BeginTrans和CommitTrans用于标记事务的开始和结束，在这两个之间的语句，就是作为事务处理的语句。判断事务处理是否成功，可通过连接对象的Error集合来实现，若Error集合的成员个数不为0，则说明有错误发生，事务处理失败。Error集合中的每一个Error对象，代表一个错误信息。 |
| postgreSQL | 如果一个错误被 EXCEPTION 捕获，PL/pgSQL 函数的局部变量保持错误发生的时候的原值， 但是所有该块中想固化在数据库中的状态都回滚。作为一个例子，让我们看看下面片断：  INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');  BEGIN  UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';  x := x + 1;  y := x / 0;  EXCEPTION  WHEN division\_by\_zero THEN  RAISE NOTICE 'caught division\_by\_zero';  RETURN x;  END;  当控制到达给 y 赋值的地方的时候，它会带着一个 division\_by\_zero 错误失败。 这个错误将被 EXCEPTION 子句捕获。而在 RETURN 语句里返回的数值将是 x 的增量值。 但是，在该块之前的 INSERT 将不会回滚，因此最终的结果是数据库包含 Tom Jones 而 不是 Joe Jones。 |

# 动态sql的使用

|  |  |
| --- | --- |
| Sql server  注：这里简单介绍，详细应用请查看相关文档 | 1 :普通SQL语句可以用exec执行  Select \* from tableName exec('select \* from tableName') exec sp\_executesql N'select \* from tableName' -- 请注意字符串前一定要加N  2:字段名，表名，数据库名之类作为变量时，必须用动态SQL  declare @fname varchar(20) set @fname = 'FiledName' --Select @fname from tableName -- 错误,不会提示错误，但结果为固定值FiledName,并非所要。 exec('select ' + @fname + ' from tableName') -- 请注意 加号前后的单引号的边上加空格  --当然将字符串改成变量的形式也可 declare @fname varchar(20) set @fname = 'FiledName' --设置字段名  declare @s varchar(1000) set @s = 'select ' + @fname + ' from tableName' exec(@s) -- 成功 --exec sp\_executesql @s -- 此句会报错  declare @s Nvarchar(1000) -- 注意此处改为nvarchar(1000) (必须为ntext或nchar哐nvarchar类型，不能是varchar类型)  set @s = 'select ' + @fname + ' from tableName' exec(@s) -- 成功 exec sp\_executesql @s -- 此句正确 |
| postgreSQL | CREATE or Replace FUNCTION func\_getnextid(  tablename varchar(240),  idname varchar(20) default 'id')  RETURNS integer AS $funcbody$  Declare  sqlstring varchar(240);  currentId integer;  Begin  sqlstring:= **'select max("' || idname || '") from "' || tablename || '";';**  **EXECUTE sqlstring into currentId;**  if currentId is NULL or currentId = 0 then  return 1;  else  return currentId + 1;  end if;  End;  $funcbody$ LANGUAGE plpgsql; |

# select into 变量或行/记录赋值语句的使用

|  |  |
| --- | --- |
| Sql server |  |
| postgreSQL | --1，单变量  declare  v\_cnt int;  BEGIN  select count(\*) **into v\_cnt** from pg\_class;  --处理  END;  --2，行/记录  DECLARE  **users\_rec RECORD;**  BEGIN  **SELECT INTO users\_rec \*** FROM users WHERE user\_id=3;  IF **users\_rec.homepage** IS NULL THEN  -- 用户没有输入主页，返回"http://"  RETURN 'http://';  END IF;  END; |

# 获取结果状态rowcount

|  |  |
| --- | --- |
| Sql server | create table #t (ID int IDENTITY, --自增字段 yhm\_id int,  yhm\_name varchar(40))  --向临时表中写入数据 declare @iRecordCount int  insert into #t  select yhm\_id,yhm\_name from dbo.[yhm]  order by yhm\_id  --select \* from dbo.[t]  --取得记录总数 set @iRecordCount=**@@rowcount** |
| postgreSQL | **第一个方法是:** 使用 GET DIAGNOSTICS，它的形式如下：  GET DIAGNOSTICS variable = item [ , ... ] ;     这条命令允许我们检索系统状态标识符。每个 item 是一个关键字，表示一个将要赋予该特定变量的状态值（该变量应该和要接收的数值类型相同）。 当前可用的状态项有 ROW\_COUNT， 最后一个发送给 SQL 引擎的 SQL 命令处理的行的数量，和 RESULT\_OID，最后一条 SQL 命令插入的最后一行的 OID。请注意 RESULT\_OID 只有在一个INSERT命令之后才有用。  一个例子：  CREATE OR REPLACE FUNCTION p\_GET\_TEST()  RETURNS void  AS $$  DECLARE  N\_ROWS INT;  BEGIN  UPDATE qxs\_test SET TNAME='BBB' WHERE TNAME='bbb';  **GET DIAGNOSTICS N\_ROWS = ROW\_COUNT;**  RAISE NOTICE 'ROW\_COUNT: %', N\_ROWS;  EXCEPTION  WHEN OTHERS THEN  RAISE NOTICE 'SQLSTATE: %', SQLSTATE;  RAISE NOTICE 'SQLERRM: %', SQLERRM;  END;  $$ LANGUAGE plpgsql;  **另外一个判断命令效果的方法是:**  一个类型为 boolean 的特殊变量 FOUND。 FOUND在每个 PL/pgSQL 函数里开始都为假。它被下列语句设置：  一个SELECT INTO语句如果返回一行则设置 FOUND 为真，如果没有返回行则设置为假。  一个PERFORM语句如果生成（或抛弃）一行，则设置 FOUND 为真，如果没有生成行则为假。  如果至少影响了一行，那么UPDATE，INSERT，和DELETE语句设置 FOUND 为真，如果没有行受影响则为假。  FETCH语句如果返回行则设置 FOUND 为真， 如果不返回行则为假。  一个FOR语句如果迭代了一次或多次，则设置 FOUND 为真，否则为假。这个规律适用于所有FOR语句的三种变体 （整数FOR循环，记录集的FOR循环，以及动态记录集FOR循环）。 只有在FOR循环退出的时候才设置 FOUND； 在循环执行的内部，FOUND 不被FOR语句修改， 但是在循环体里它可能被其他语句的执行而修改。  FOUND 是每个 PL/pgSQL 里的局部变量； 它的任何修改只影响当前的函数。  一个例子：  CREATE OR REPLACE FUNCTION p\_gisrs2iodn\_vendor(v\_vendorNameCn VARCHAR(255),  OUT n\_vendorId BIGINT)  AS $$  declare  n\_seq BIGINT;  BEGIN  if v\_vendorNameCn is null then  begin  n\_vendorId := null;  end;  else  begin  select ID into n\_vendorId  from XB\_VENDOR  where NAMECN = v\_vendorNameCn;    **if not found then**  begin  n\_seq := nextval('s\_xb\_vendor');    insert into XB\_VENDOR  (ID, NAMECN)  values  (n\_seq, v\_vendorNameCn);    n\_vendorId := n\_seq;  end;  end if;  end;  end if;  END;  $$ LANGUAGE plpgsql; |

# sequence的使用

|  |  |
| --- | --- |
| Sql server | 如果大家在ORACLE里面用惯了Sequence，要在SqlServer里实现Sequence，就会发现没有现成的Sequence对象可以Create了。那应该怎么办呢？  当然这点小问题是难不倒我们程序员的，“max+1啊”，有人会说这样的方式。是的，这种方式实现起来也比较简单。当然你也许还会说，最Sql的方式是采用identity列，自增的方式去增加就ok了。但是这样的列如果要实现“YYYYMMDD”+“部门编号（如ZX）”+Sequence值 （例如：20100707ZX000002）的格式就不行了。还是老老实实用存储过程来取得一个Sequence值吧，使用起来也就随心所欲了。  网上常见的一个存储过程为 为当前库所有用到Sequence建一张表，譬如“AllSequence”，内含四个字段【名字，起点值，递增值，当前值】，创建Sequence的时候就创建一条记录，获取Sequence的时候就从对应行的当前值按递增值增加得到。  在并发性请求不那么高的系统里，这种过程是没有问题的。但是并发性请求一旦处于某个量级以后，这个过程常常会遇到问题。  以下为一个改进方法，适合于高吞吐量的访问请求，比如每秒数千次的请求：  -- 假定要给T\_0101001创建一个Sequence  -- 创建表SeqT\_0101001  create table SeqT\_0101001(  -- ID列为自增列  SeqID int identity(1,1) primary key,  -- Sequence值  SeqVal varchar(1)  )  -- 创建从SeqT\_0101001表获取最新Sequence值的存储过程  create procedure P\_GetNewSeqVal\_SeqT\_0101001  as  begin  -- 声明新Sequence值变量  declare @NewSeqValue int  -- 设置插入、删除操作后的条数显示取消  set NOCOUNT ON  -- 插入新值到SeqT\_0101001表  insert into SeqT\_0101001 (SeqVal) values ('a')  -- 设置新Sequence值为插入到SeqT\_0101001表的标识列内的最后一个标识值  set @NewSeqValue = scope\_identity()  --如果组合的Sequence值<999999999(大于的话数值会溢出)，如:“YYYY”+5位的Sequence值(201019988)，可以采用下面的方式来实现  --set @NewSeqValue = Convert(char(4),Getdate(),112) + right('00000'+CAST(scope\_identity() AS varchar(5)),5)  --这样直接Print @NewSeqVal 就可以得到类似“201019988”的Sequence值  -- 删除SeqT\_0101001表(不显示被锁行)  delete from SeqT\_0101001 WITH (READPAST)  -- 返回新Sequence值  return @NewSeqValue  end;  --使用Sequence  Declare @NewSeqVal int  Exec @NewSeqVal = P\_GetNewSeqVal\_SeqT\_0101001  Print @NewSeqVal  Print Convert(char(8),Getdate(),112) +'ZX'+ right('000000'+CAST(@NewSeqVal AS varchar(5)),6) |
| postgreSQL | Sequence是数据库系统按照一定规则自动增加的数字序列。这个序列一般作为代理主键（因为不会重复），没有其他任何意义。  CREATE SEQUENCE yuanyao\_id\_seq  INCREMENT 1  MINVALUE 1  MAXVALUE 9223372036854775807  START 617  CACHE 1;  参数:  TEMPORARY 或 TEMP  如果声明了这个修饰词，那么该序列对象只为这个会话创建， 并且在会话结束的时候自动删除。在临时序列存在的时候， 同名永久序列是不可见的（在同一会话里），除非它们是用模式修饰的名字引用的。  name  将要创建的序列号名（可以用模式修饰）。  increment  可选子句 INCREMENT BY increment 要创建一个新的值，应该向当前序列值上增加什么。一个正数将生成一个递增的序列， 一个负数将生成一个递减的序列。缺省值是一（1）。  minvalue  NO MINVALUE  可选的子句 MINVALUE minvalue 决定一个序列可生成的最小值。 如果没有声明这个子句或者声明了 NO MINVALUE，那么就使用缺省。 缺省分别是递增序列为 1 递减为 -263-1。  maxvalue  NO MAXVALUE  使用可选子句 MAXVALUE maxvalue 决定序列的最大值。 如果没有声明这个子句或者声明了 NO MAXVALUE，那么就使用缺省。 缺省的分别是递增为 -263-1，递减为 -1。  start  可选的 START WITH start 子句 使序列可以从任意位置开始。缺省初始值是递增序列为 minvalue 递减序列为 maxvalue.  cache  CACHE cache 选项使序列号预分配并且为快速访问存储在内存里面。 最小值（也是缺省值）是1（一次只能生成一个值, 也就是说没有缓存）这也是缺省。  CYCLE  NO CYCLE  可选的CYCLE关键字可用于使序列到达 最大值（maxvalue） 或 最小值（minvalue） 时可复位并继续下去。如果达到极限，生成的下一个数据将分别是 最小值（minvalue） 或 最大值（maxvalue）。  如果声明了可选的关键字 NO CYCLE， 那么在序列达到其最大值之后任何对 nextval 的调用都强返回一个错误。 如果既没有声明 CYCLE 也没有声明 NO CYCLE， 那么 NO CYCLE 是缺省。    **例子:**  --创建一个叫 serial 的递增序列，从101开始：  CREATE SEQUENCE serial START 101;  --从此序列中选出下一个数字：  SELECT nextval('serial');  nextval  -------  114  --在一个 INSERT 中使用此序列：  INSERT INTO distributors VALUES (nextval('serial'),'nothing');  --修改初始值  ALTER SEQUENCE serial RESTART WITH 1; |

# 递归查询（connectby() 或with recursive）

|  |  |
| --- | --- |
| Sql server | 参考：<http://www.soaspx.com/dotnet/sql/mssql/sql2005/sqlservr2005_20120525_9197.html>  --查询子节点  with  RTD1 as(  select id ,pid from UserGroup  ),  RTD2 as(  select \* from RTD1 where id=6  union all  select RTD1.\* from RTD2 inner join RTD1  on RTD2.id=RTD1.PID  )  select \* from RTD2  --查询父节点  with  RTU1 as(  select id ,pid from UserGroup  ),  RTU2 as(  select \* from RTU1 where id=26  union all  select RTU1.\* from RTU2 inner join RTU1  --on myT2.id=myT.PID  on RTU2.PID=RTU1.ID  )  select \* from RTU2 |
| postgreSQL  **注：在使用connectby()函数时，要注意查询表中的递归查询的id和parentid字段 与 生成结果中的key和parentkey字段是有差异的，在理解业务时要加以区分。** | 方法一：connectby()函数  PostgreSQL8.3以前是用connectby()函数来做递归 查 询 。 connectby() 函数是 contrib/tablefunc模 块 下的功能，默认是没有安装的，需要自己安装。  方法二：with recursive语句  PostgreSQL8.4中当然可以用connectby()函数来做递归 查 询，且系统默认安装了这个功能。下面简单介绍一下 PostgreSQL8.4增加的新功能递归SQL查询的用法。  --取得一棵树的所有孩子节点  WITH RECURSIVE r AS (  SELECT \* FROM tree WHERE id = 1  union ALL  SELECT tree.\* FROM tree, r WHERE tree.parent = r.id  )  SELECT \* FROM r ORDER BY id;  **注：可以参考我们系统中的三个函数，代码中有两种方法的实现代码**  1，getoltcode  2，getobdcodelist  3，f\_getobdlevel |

# 系统对象名（表名、索引等）是小写

|  |  |
| --- | --- |
| postgreSQL | --例如：  --查询所有以'p\_gis'打头的存过  select \* from pg\_proc where proname like 'p\\\_gis%';  如果用大写'P\_gis'，查询无结果  --所有以'xb\_'打头的所有表  select \* from pg\_tables where tablename like 'xb\\\_%'; |

# 常用字符串操作

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sql server | |  |  |  | | --- | --- | --- | | + | 字串连接 | 'SQL' + ' server' = ' SQL server' | | charindex(char\_expr, expression) | 字符串查找 |  | | substring(expression, start, length) | 取子字串 |  | | upper(char\_expr) | 把字串转化为大写 |  | | lower(char\_expr) | 把字串转化为小写 |  | | len(expression) | 字符（而不是字节）个数 |  | | http://blog.sina.com.cn/s/blog\_50cc0ffd0100c775.html | | | |
| postgreSQL | |  |  |  | | --- | --- | --- | | || | 字串连接 | 'Post' || 'greSQL' = PostgreSQL | | position(substring in string) | 指定的子字串的位置 | position('om' in 'Thomas') = 3 | | substring(string [from int] [for int]) | 抽取子字串 | substring('Thomas' from 2 for 3) = 'hom' | | replace(string text, from text, to text) | 把字串string里出现地所有子字串 from 替换成子字串 to。 | replace('abcdefabcdef', 'cd', 'XX') = 'abXXefabXXef' | | upper(string) | 把字串转化为大写 | upper('tom') = 'TOM' | | lower(string) | 把字串转化为小写 | lower('TOM') = 'tom' | | length(string text) | string 中字符的数目 | length('jose') = 4 | | http://www.php100.com/manual/PostgreSQL8/ | | | |

# 字串连接，以及null字符的处理

|  |  |
| --- | --- |
| Sql server | --**sqlserver 2012之前的版本**  /\* 处理not null字符 \*/  DECLARE @t1 varchar(10) = 'a' DECLARE @t2 varchar(10) = 'b' DECLARE @t3 varchar(10) = 'c' DECLARE @t4 varchar(10) = 'd' DECLARE @N1 int = 2012 SELECT @t1+@t2+@t3+@t4+convert(varchar(10),@N1 ) as result go  /\* 处理 null字符 \*/  DECLARE @t1 varchar(10) = 'a' DECLARE @t2 varchar(10) = 'b' DECLARE **@t3 varchar(10) = null** DECLARE @t4 varchar(10) = 'd' DECLARE @N1 int = 2012  SELECT @t1+@t2+**ISNULL(@t3,'')**+@t4+convert(varchar(10),@N1 ) as result GO  如果这里不使用isnull判断，那么我们获得的值将为null, 与我们期望的大相径庭.  ---**sqlserver 2012中** SELECT CONCAT('a','b','c','d','2012') as result go /\* 处理not null字符 \*/  DECLARE @t1 varchar(10) = 'a' DECLARE @t2 varchar(10) = 'b' DECLARE @t3 varchar(10) = 'c' DECLARE @t4 varchar(10) = 'd' DECLARE @N1 int = 2012  SELECT CONCAT(@t1, @t2, @t3, @t4, @N1) as result go /\* 处理 null字符 \*/  DECLARE @t1 varchar(10) = 'a' DECLARE @t2 varchar(10) = 'b' DECLARE **@t3 varchar(10) = null** DECLARE @t4 varchar(10) = 'd' DECLARE @N1 int = 2012  SELECT **CONCAT(@t1, @t2, @t3, @t4, @N1)** as result go |
| postgreSQL | 函数：string || string 说明：String concatenation 字符串连接操作 例子：'Post' || 'greSQL'＝PostgreSQL  **/\* 处理 null字符 \*/**  select a||b from table1; **当a或b其中一个为null时，该查询返回null，切记切记！！！！！**  替代方法一：  select coalesce(a,'') || coalesce(b,'') from table1;  替代方法二：  在public schema中创建函数concat  create or replace function concat(text, text)  returns text as  $body$select coalesce($1,'') || coalesce($2,'')$body$  language 'sql' volatile;  alter function concat(text, text) owner to postgres;  函数：string || non-string or non-string || string 说明：String concatenation with one non-string input 字符串与非字符串类型进行连接操作 例子：'Value: ' || 42 = Value: 42  postgressql数据库中有A，B两个表，都有字段name。 在left join的on中有A.name=B.name， A.name 和 B.name 都有值切相等的时候，可以显示。但是当A.name和B.name都为null的时候，A.name=B.name的值就取不出来了，请问这on条件该怎么写？  postgresql中有COALESCE函数，可以替换null，可以使用这个函数来试下。 COALESCE(A.name,'') = COALESCE(B.name,'') 这样就会把null替换为0 那么就会值相同了，出来的结果中，要注意看下是不是正确的。 |

# 类型转换

|  |  |
| --- | --- |
| Sql server | 参考：<http://www.blogjava.net/kxx129/archive/2006/08/23/65340.html>  在一般情况下，SQL Server 会自动完成数据类型的转换，例如，可以直接将字符数据类型或表达式与DATATIME 数据类型或表达式比较当表达式中用了INTEGER、 SMALLINT或TINYINT 时，SQL Server 也可将INTEGER 数据类型或表达式转换为SMALLINT数据类型或表达式，这称为隐式转换。如果不能确定SQL Server 是否能完成隐式转换或者使用了不能隐式转换的其它数据类型，就需要使用数据类型转换函数做显式转换了。此类函数有两个：  ·CAST（）  CAST（） 函数语法如下：  CAST （<expression> AS <data\_ type>[ length ]）  ·CONVERT（）  CONVERT（） 函数语法如下：  CONVERT （<data\_ type>[ length ]， <expression> [， style]）  提醒： 1、data\_type为SQL Server系统定义的数据类型，用户自定义的数据类型不能在此使用。  2、length用于指定数据的长度，缺省值为30。  3、把CHAR或VARCHAR类型转换为诸如INT或SAMLLINT这样的INTEGER类型、结果必须是带正号（+）或负号（-）的数值。  4、TEXT类型到CHAR或VARCHAR类型转换最多为8000个字符，即CHAR或VARCHAR数据类型是最大长度。  5、IMAGE类型存储的数据转换到BINARY或VARBINARY类型，最多为8000个字符。  6、把整数值转换为MONEY或SMALLMONEY类型，按定义的国家的货币单位来处理，如人民币、美元、英镑等。  7、BIT类型的转换把非零值转换为1，并仍以BIT类型存储。  8、试图转换到不同长度的数据类型，会截短转换值并在转换值后显示“+”，以标识发生了这种截断。  用CONVERT（） 函数的style 选项能以不同的格式显示日期和时间。style 是将DATATIME 和SMALLDATETIME 数据转换为字符串时所选用的由SQL Server 系统提供的转换样式编号，不同的样式编号有不同的输出格式。 |
| postgreSQL | 一个类型转换声明一个从一种数据类型到另外一种数据类型的转换。 PostgreSQL 接受两种等效的类型转换语法：  **CAST ( expression AS type )**  **expression::type**  CAST 语法遵循 SQL；:: 的语法是 PostgreSQL 传统用法。  如果对一个已知类型的值表达式应用转换，它代表一个运行时类型转换。 只有在定义了合适的类型转换操作的情况下，该转换才能成功。 请注意这一点和用于常量的转换略有区别，如 Section 4.1.2.5 所示。 一个应用于某个未修饰的字串文本的转换表示给一个字串文本数值赋予一个初始化类型， 因此它对于任何类型都会成功（如果字串文本的内容符合该数据类型的输入语法接受。）  如果对于一个值表达式生成的数值对某类型而言不存在混淆的情况， 那么我们可以省略明确的类型转换（比如，在给一个表字段赋值的时候）； 在这样的情况下，系统将自动附加一个类型转换。 不过，自动转换只适用于那些系统表中标记着 "OK to apply implicitly" 的转换函数。 其它转换函数必须用明确的转换语法调用。 这些限制是为了避免一些怪异的转换被应用。  我们也可以用函数样的语法声明一个类型转换：  **typename ( expression )**  不过，这个方法只能用于那些名字同时也是有效函数名字的类型。 比如，double precision 就不能这么用， 但是等效的 float8 可以。同样，interval， time，和 timestamp 如果加了双引号也只能这么用， 因为存在语法冲突。因此，函数样的类型转换会导致不一致， 所以可能应该避免在新应用中这么用。 （函数样语法实际上就似乎一个函数调用。如果使用两种标准转换语法做运行时转换， 那么它将在内部调用一个已注册得函数执行转换。通常， 这种转换函数和它们得输出类型同名，但是这个要点可不是那些可以移植的程序可以依赖的东西。） |

# 数字类型转换成格式字符串

|  |  |
| --- | --- |
| Sql server | 参考：<http://blog.csdn.net/studyzy/article/details/4491127>  SQL Server中的数值类型分为两种，一种是精确的数值类型，具体的数据类型有：bit、tinyint、smallint、int、bigint、 smallmoney、money和decimal，这些数据类型能够精确的表明某以数值；另一种是近似的数值类型，具体就是float和real。浮点 数据为近似值，因此，并非数据类型范围内的所有值都能精确地表示。  有些时候我们需要将这些数值类型转换为字符串类型，用到的转换函数就 是cast和convert，这两个函数的作用都是进行类型转换，只不过语法格式不同。据说在转换时还是有一定的区别的，不过我个人更习惯于使用 convert函数，应该这个函数一方面更像是个函数的语法，另一方面在做时间和数值转换成字符串时还可以指定转换的格式。  对于精确数值的数据类型，转换出来的字符串就是我们存储的数值。比如：  declare @i int  set @i=123456789  print 'test:'+convert(varchar(20),@i)  输出就是：test:123456789  而如果对于近似数值的数据类型，那么可就没有那么简单了。  declare @i float  set @i=123456789  print 'test:'+convert(varchar(20),@i)  输出结果：test:1.23457e+008  输出的结果是使用科学计数法来表示的,再看看可否通过指定转换样式来指定不使用科学计数法呢?帮助文档中说到float 或 real 转换为字符数据时的 style 值:  0（默认值）最大为 6 位数。根据需要使用科学记数法。  1 始终为 8 位值。始终使用科学记数法。  2 始终为 16 位值。始终使用科学记数法。  我们的值是123456789,超过了6位数.所以不管是0还是1,2结果都会使用科学计数法来表示。那么要怎么样才能将我们的数据不转换成科学计数法而输出呢？比较简单的办法就是将近似数据转换为精确数据，然后再将精确数据转换成字符串。  同样以上面的例子为例， 进行两次数据类型的转换如下：  declare @i float  set @i=123456789  print 'test:'+convert(varchar(20),convert(int,@i))  输出：test:123456789  如果是有小数，那么我们要转换出小数位的话，可以转换成decimal，转换出的小数位数在定义decimal时指定。比如要输出4位小数，那么转换代码是：declare @i float  set @i=123456789.12  print 'test:'+convert(varchar(20),convert(decimal(18,4),@i))  输出：test:123456789.1200 |
| postgreSQL | 参考：<http://openblogs.blog.163.com/blog/static/189733135201121011441496/>  <http://www.php100.com/manual/PostgreSQL8/>  to\_char(int, text) text 把整数转换成字符串 to\_char(125, '999')  to\_char(double precision, text) text 把实数/双精度数转换成字符串 to\_char(125.8::real, '999D9')  to\_char(numeric, text) text 把数字转换成字符串 to\_char(-125.8, '999D99S')  数值格式化的模版模式:  9 带有指定数值位数的值  0 带前导零的值  . (句点) 小数点  , (逗号) 分组(千)分隔符  PR 尖括号内负值  S 带符号的数值(使用区域设置)  L 货币符号(使用区域设置)  D 小数点(使用区域设置)  G 分组分隔符(使用区域设置)  MI 在指明的位置的负号(如果数字 < 0)  PL 在指明的位置的正号(如果数字 > 0)  SG 在指明的位置的正/负号  RN 罗马数字(输入在 1 和 3999 之间)  TH 或 th 序数后缀  V 移动指定位(小数)(参阅注解)  EEEE 科学记数(尚未实现)  数字格式化的用法须知：  使用 SG, PL, MI 生成的符号并不挂在数字上面；比如，to\_char(-12, 'S9999') 生成 ' -12' ；to\_char(-12, 'MI9999') 生成 '- 12' 。Oracle里的实现不允许在 9 前面使用 MI ，而是要求 9 在 MI 前面。  9 声明和 9 的个数相同的数字位数的数值。如果某个数值位没有数字，则输出一个空白。  TH 不会转换小于零的数值，也不会转换小数。  PL, SG, TH 是 PostgreSQL 扩展。  V 方便地把输入值乘以 10^n ，这里 n 是跟在 V 后面的数字。to\_char 不支持把 V 与一个小数点组合在一起使用(也就是说 99.9V99 是不允许的)。 |

# 日期类型转换成格式字符串

|  |  |
| --- | --- |
| Sql server | 参考：<http://blog.csdn.net/studyzy/article/details/4491127>  SQL Server中的数值类型分为两种，一种是精确的数值类型，具体的数据类型有：bit、tinyint、smallint、int、bigint、 smallmoney、money和decimal，这些数据类型能够精确的表明某以数值；另一种是近似的数值类型，具体就是float和real。浮点 数据为近似值，因此，并非数据类型范围内的所有值都能精确地表示。  有些时候我们需要将这些数值类型转换为字符串类型，用到的转换函数就 是cast和convert，这两个函数的作用都是进行类型转换，只不过语法格式不同。据说在转换时还是有一定的区别的，不过我个人更习惯于使用 convert函数，应该这个函数一方面更像是个函数的语法，另一方面在做时间和数值转换成字符串时还可以指定转换的格式。  对于精确数值的数据类型，转换出来的字符串就是我们存储的数值。比如：  declare @i int  set @i=123456789  print 'test:'+convert(varchar(20),@i)  输出就是：test:123456789  而如果对于近似数值的数据类型，那么可就没有那么简单了。  declare @i float  set @i=123456789  print 'test:'+convert(varchar(20),@i)  输出结果：test:1.23457e+008  输出的结果是使用科学计数法来表示的,再看看可否通过指定转换样式来指定不使用科学计数法呢?帮助文档中说到float 或 real 转换为字符数据时的 style 值:  0（默认值）最大为 6 位数。根据需要使用科学记数法。  1 始终为 8 位值。始终使用科学记数法。  2 始终为 16 位值。始终使用科学记数法。  我们的值是123456789,超过了6位数.所以不管是0还是1,2结果都会使用科学计数法来表示。那么要怎么样才能将我们的数据不转换成科学计数法而输出呢？比较简单的办法就是将近似数据转换为精确数据，然后再将精确数据转换成字符串。  同样以上面的例子为例， 进行两次数据类型的转换如下：  declare @i float  set @i=123456789  print 'test:'+convert(varchar(20),convert(int,@i))  输出：test:123456789  如果是有小数，那么我们要转换出小数位的话，可以转换成decimal，转换出的小数位数在定义decimal时指定。比如要输出4位小数，那么转换代码是：declare @i float  set @i=123456789.12  print 'test:'+convert(varchar(20),convert(decimal(18,4),@i))  输出：test:123456789.1200 |
| postgreSQL | 参考：<http://openblogs.blog.163.com/blog/static/189733135201121011441496/>  <http://www.php100.com/manual/PostgreSQL8/>  to\_char(timestamp, text) text 把时间戳转换成字符串 to\_char(current\_timestamp, 'HH12:MI:SS')  to\_char(interval, text) text 把时间间隔转为字符串 to\_char(interval '15h 2m 12s', 'HH24:MI:SS')  to\_date(text, text) date 把字符串转换成日期 to\_date('05 Dec 2000', 'DD Mon YYYY')  to\_number(text, text) numeric 把字符串转换成数字 to\_number('12,454.8-', '99G999D9S')  to\_timestamp(text, text) timestamp with time zone 把字符串转换成时间戳 to\_timestamp('05 Dec 2000', 'DD Mon YYYY')  to\_timestamp(double precision) timestamp with time zone 把 UNIX 纪元转换成时间戳 to\_timestamp(200120400)  用于日期/时间格式化的模式模式 描述  HH 一天的小时数(01-12)  HH12 一天的小时数(01-12)  HH24 一天的小时数(00-23)  MI 分钟(00-59)  SS 秒(00-59)  MS 毫秒(000-999)  US 微秒(000000-999999)  SSSS 午夜后的秒(0-86399)  AM 或 A.M. 或 PM 或 P.M. 上下午标识(大写)  am 或 a.m. 或 pm 或 p.m. 上下午标识(小写)  Y,YYY 带逗号的年(4 和更多位)  YYYY 年(4 和更多位)  YYY 年的后三位  YY 年的后两位  Y 年的最后一位  IYYY ISO年(4 位或更多位)  IYY ISO年的最后三位  IY ISO年的最后两位  I ISO年的最后一位  BC 或 B.C. 或 AD 或 A.D. 纪元标识(大写)  bc 或 b.c. 或 ad 或 a.d. 纪元标识(小写)  MONTH 全长大写月份名(空白填充为 9 字符)  Month 全长混合大小写月份名(空白填充为 9 字符)  month 全长小写月份名(空白填充为 9 字符)  MON 大写缩写月份名(3 字符)  Mon 混合大小写缩写月份名(3 字符)  mon 小写缩写月份名(3 字符)  MM 月份数(01-12)  DAY 全长大写日期名(空白填充为 9 字符)  Day 全长混合大小写日期名(空白填充为 9 字符)  day 全长小写日期名(空白填充为 9 字符)  DY 缩写大写日期名(3 字符)  Dy 缩写混合大小写日期名(3 字符)  dy 缩写小写日期名(3 字符)  DDD 一年里的日(001-366)  DD 一个月里的日(01-31)  D 一周里的日(1-7 ；周日是 1)  W 一个月里的周数(1-5)(第一周从该月第一天开始)  WW 一年里的周数(1-53)(第一周从该年的第一天开始)  IW ISO一年里的周数(第一个星期四在第一周里)  CC 世纪(2 位)(20 世纪从 2001-01-01 开始)  J 儒略日(自公元前 4712 年 1 月 1 日来的天数)  Q 季度  RM 罗马数字的月份(I-XII ；I=JAN)(大写)  rm 罗马数字的月份(I-XII ；I=JAN)(小写)  TZ 时区名(大写)  tz 时区名(小写)  日期/时间格式化的模板模式修饰词修饰符 描述 例子  FM 前缀 填充模式(抑制填充空白和零) FMMonth  TH 后缀 大写顺序数后缀 DDTH  th 后缀 小写顺序数后缀 DDth  FX 前缀 固定格式全局选项(见用法须知) FX Month DD Day  TM 前缀 翻译模式(基于 lc\_messages 显示本地化的日期和月份名) TMMonth  SP 后缀 拼写模式(还未实现) DDSP |