文章
姚 鑫 · 十一月 12, 2021 阅读大约需 9 分钟

第七十四章 SQL命令 SET TRANSACTION

第七十四章 SQL命令 SET TRANSACTION

设置事务的参数。

大纲

```
SET TRANSACTION [%COMMITMODE commitmode]

SET TRANSACTION [transactionmodes]
```

参数

- %COMMITMODE commitmode - 可选-指定向数据库提交事务的方式。
  取值包括EXPLICIT、IMPLICIT和NONE。
  默认为IMPLICIT。
- transactionmodes - 可选—指定事务的隔离模式和访问模式。
  可以将隔离模式、访问模式或这两种模式的值指定为逗号分隔的列表。隔离模式的有效值为ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, and ISOLATION LEVEL READ VERIFIED。
  默认为 “ ISOLATION LEVEL READ UNCOMMITTED ” 。
  访问模式的有效值为 “ READ ONLY ” 和 “ READ WRITE ” 。
  注意，只有隔离级别READ COMMITTED与读写模式READ WRITE兼容。

描述

SET TRANSACTION语句为当前进程设置控制SQL事务的参数。
这些参数在下一个事务开始时生效，并在当前进程持续期间或直到显式重置为止。
它们不会在事务结束时自动重置为默认值。

单个SET TRANSACTION语句可用于设置提交模式参数或事务模式参数，但不能同时设置两者。

可以使用START TRANSACTION命令设置相同的参数，该命令既可以设置参数，也可以开始一个新事务。
也可以使用方法调用设置参数。

SET TRANSACTION不会开始一个事务，因此不会增加$TLEVEL事务级别计数器。

SET TRANSACTION可以在动态SQL(包括SQL Shell)和嵌入式SQL中使用。

%COMMITMODE

%COMMITMODE关键字允许您指定是否执行自动事务承诺。
可供选择的选项有:

- IMPLICIT隐式:自动事务承诺是开启的(默认)。
  当程序发出数据库修改操作(INSERT、UPDATE或DELETE)时，SQL自动启动一个事务。
  事务将继续进行，直到操作成功完成并SQL自动提交更改，或者操作无法在所有行上成功完成并SQL自动回滚整个操作。
  每个数据库操作(INSERT、UPDATE或DELETE)构成一个单独的事务。
  成功完成数据库操作将自动清除回滚日志、释放锁并减少$TLEVEL。
  不需要COMMIT语句。
  这是默认设置。
- EXPLICIT:关闭自动事务承诺。
  当程序发出第一个数据库修改操作(INSERT、UPDATE或DELETE)时，SQL自动启动一个事务。
  该交易将继续进行，直到明确达成协议。
  成功完成后，发出COMMIT语句。
  如果数据库修改操作失败，则发出ROLLBACK语句将数据库恢复到事务开始之前的位置。
  在EXPLICIT模式下，每个事务的数据库操作数是用户定义的。
- NONE:没有自动事务处理。
  除非由START transaction语句显式调用，否则不会初始化事务。
  必须通过发出COMMIT或ROLLBACK语句显式地结束事务。
  因此，事务中是否包含数据库操作以及事务中数据库操作的数量都是用户定义的。

TRUNCATE TABLE不会在自动启动的事务中发生。
如果需要对TRUNCATE TABLE进行日志记录和回滚，则必须显式指定START TRANSACTION，并以显式COMMIT或rollback结束。

可以使用GetOption(“ AutoCommit ”)方法确定当前进程的%COMMITMODE设置，如下面的ObjectScript示例所示:

```
ClassMethod SetTransaction()
{
    s stat = $SYSTEM.SQL.SetOption("AutoCommit",$RANDOM(3),.oldval)
    if stat '= 1 {
        w "SetOption failed:"
        d $System.Status.DisplayError(stat)
        q
    }
    s x = $SYSTEM.SQL.GetOption("AutoCommit")
    if x = 1 {
        w "%COMMITMODE IMPLICIT (default behavior):",!,
        "???????????????",!,
        "???????"
    } elseif x = 0 {
        w "%COMMITMODE NONE:",!,
        "????????",!,
        "????START TRANSACTION?????",!,
        "?COMMIT?ROLLBACK?????"
    } else {
        w "%COMMITMODE EXPLICIT:",!,
        "???????????",!,
        "??????;????",!,
        "?????COMMIT?ROLLBACK" }
}
```

%COMMITMODE可以在ObjectScript中使用SetOption()方法设置，如下set status=$SYSTEM.SQL.Util.SetOption("AutoCommit"，intval，.oldval)。
可用的方法值为0 (NONE)、1 (IMPLICIT)和2 (EXPLICIT)。

隔离级别

可以为发出查询的进程指定“ 隔离级别 ”。
“ 隔离级别 ”选项允许指定正在进行的更改是否可用于查询的读访问。
如果另一个并发进程正在执行对表的插入或更新，并且对表的更改在事务中，那么这些更改正在进行中，并且可能会回滚。
通过为正在查询该表的流程设置ISOLATION
LEVEL，可以指定是否希望在查询结果中包含或排除这些正在进行的更改。

- READ UNCOMMITTED表示所有更改都可以立即用于查询访问。
  这包括随后可能被回滚的更改。
  READ UNCOMMITTED确保查询将在不等待并发插入或更新进程的情况下返回结果，并且不会因为锁定超时错误而失败。
  然而，READ UNCOMMITTED的结果可能包括未提交的值;
  这些值在内部可能不一致，因为插入或更新操作只部分完成，这些值可能随后被回滚。
  如果查询进程不在显式事务中，或者事务没有指定隔离级别，则READ UNCOMMITTED是默认值。
  READ UNCOMMITTED与READ - WRITE访问不兼容;
  试图在同一语句中同时指定这两个变量会导致SQLCODE -92错误。
- READ VERIFIED声明来自其他事务的未提交数据立即可用，并且不执行锁操作。
  这包括随后可能被回滚的更改。
  然而，与READ UNCOMMITTED不同的是，READ VERIFIED事务将重新检查任何可能因未提交或新提交的数据而失效的条件，这将导致不满足查询条件的输出。
  由于这种条件重新检查，READ VERIFIED比READ
  UNCOMMITTED更准确，但效率更低，应该只在可能发生对条件检查的数据的并发更新时使用。
  READ VERIFIED与READ - WRITE访问不兼容;
  试图在同一语句中同时指定这两个变量会导致SQLCODE -92错误。
- READ COMMITTED表示只有那些已经提交的更改可以用于查询访问。
  这确保了在数据库上以一致的状态执行查询，而不是在进行一组更改时执行，这组更改随后可能会回滚。
  如果请求的数据已被更改，但更改尚未提交(或回滚)，则查询将等待事务完成。
  如果在等待该数据可用时发生锁定超时，则会发出SQLCODE -114错误。

READ UNCOMMITTED还是READ VERIFIED?

下面的例子演示了READ UNCOMMITTED和READ VERIFIED之间的区别:

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

查询优化器可能首先选择从Name索引中收集所有RowID包含的符合>= 'M'条件的Name。
收集之后，每次访问一个RowID，以检索Name和SSN字段用于输出。
并发运行的更新事务可以将一个RowID
72的Person的Name字段从“ Smith ”更改为“ Abel ”，该字段位于查询的rowwid集合和它对表的逐行访问之间。
在本例中，索引中的RowID集合将包含不再符合Name >= 'M'条件的行的RowID。

READ UNCOMMITTED查询处理假设Name >=
'M'条件已经被索引满足，并且将输出从索引中收集的每个RowID在表中出现的任何Name。
因此，在本例中，它将输出一个名称为'Abel'的行，该行不满足条件。

READ VERIFIED查询处理注意到，它正在从表中为output (Name)检索一个字段，该字段参与了之前应该由索引满足的条件，然后重新检查条件，以防在检查索引之后字段值发生变化。
在重新检查时，它注意到该行不再满足条件，并将其从输出中删除。
只有输出所需的值才会重新检查其条件:在本例中，SELECT SSN FROM Person WHERE Name >=
'M'将输出RowID为72的行。

READ COMMITTED 异常

当ISOLATION LEVEL read committed生效时，可以通过设置ISOLATION LEVEL read committed或SetOption()方法，如下SET status=$SYSTEM.SQL.Util.SetOption("IsolationMode"，1，.oldval)。

SQL只能检索已提交数据的更改。
然而，也有一些明显的例外:

- 查询永远不会返回已删除的行，即使删除该行的事务正在进行，且删除可能随后回滚。
  ISOLATION LEVEL READ COMMITTED确保插入和更新处于一致状态，而不是删除。
- 如果查询包含聚合函数，则聚合结果将返回数据的当前状态，而与指定的隔离级别无关。
  因此，聚合结果中包含正在进行的插入和更新(随后可能回滚)。
  正在进行的删除(随后可能会回滚)不包括在聚合结果中。
  这是因为聚合操作需要访问表中的许多行数据。
- 包含DISTINCT子句或GROUP BY子句的SELECT查询不受隔离级别设置的影响。
  包含这些子句之一的查询将返回数据的当前状态，包括可能随后回滚的正在进行的更改。
  这是因为这些查询操作需要访问表中的许多行数据。
- 带有%NOLOCK关键字的查询。

注意:在使用ECP(企业缓存协议)的IRIS实现上，与READ UNCOMMITTED相比，使用READ COMMITTED可能会导致明显的性能下降。
在定义包含ECP的事务时，开发人员应该权衡READ UNCOMMITTED的优越性能和READ COMMITTED的更高数据准确性。

有效隔离级别

可以使用set TRANSACTION(不启动事务)、START TRANSACTION(设置隔离模式并启动事务)或SetOption(“ IsolationMode ”)方法调用为进程设置隔离级别。

指定的隔离级别保持有效，直到由SET TRANSACTION、START TRANSACTION或SetOption(“ IsolationMode ”)方法调用显式重置。
由于COMMIT或ROLLBACK仅对数据更改有意义，而对数据查询没有意义，因此COMMIT或ROLLBACK操作对ISOLATION LEVEL设置没有影响。

在查询开始时有效的 “ 隔离级别 ” 在查询期间仍然有效。

可以使用GetOption(“ IsolationMode ”)方法调用确定当前进程的隔离级别。
还可以使用SetOption(“ IsolationMode ”)方法调用为当前进程设置隔离模式。
这些方法将READ UNCOMMITTED(默认值)指定为0,READ COMMITTED指定为1,READ VERIFIED指定为3。
指定任何其他数值将保持隔离模式不变。
如果将隔离模式设置为当前隔离模式，则不会发生错误或更改。
以下示例显示了这些方法的使用:

```
ClassMethod SetTransaction1()
{
    w $SYSTEM.SQL.GetOption("IsolationMode")," ??",!
    &sql(
        START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE
    )
    w $SYSTEM.SQL.GetOption("IsolationMode")," TART TRANSACTION??",!
    d $SYSTEM.SQL.SetOption("IsolationMode",0,.stat)
    if stat=1 {
        w $SYSTEM.SQL.GetOption("IsolationMode")," after IsolationMode=0 call",!
    } else { WRITE "Set IsolationMode error"
    }
    &sql(COMMIT)
}
```

隔离模式和访问模式必须始终兼容。
更改访问模式将更改隔离模式，示例如下:

```
ClassMethod SetTransaction2()
{
    w $SYSTEM.SQL.GetOption("IsolationMode")," default",!
    &sql(
        SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE
    )
    w $SYSTEM.SQL.GetOption("IsolationMode")," after SET TRANSACTION",!
    &sql(START TRANSACTION READ ONLY)
    w $SYSTEM.SQL.GetOption("IsolationMode")," after changing access mode",!
    &sql(COMMIT)
}
```

示例

下面的嵌入式SQL示例使用两个SET TRANSACTION语句来设置事务参数。
注意，SET TRANSACTION不会增加事务级别($TLEVEL)。
START TRANSACTION命令启动一个事务并增加$TLEVEL:

```
ClassMethod SetTransaction3()
{
    &sql(SET TRANSACTION %COMMITMODE EXPLICIT)
    w !,"????????, SQLCODE=",SQLCODE
    w !,"????=",$TLEVEL
    &sql(SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED)
    w !,"????????, SQLCODE=",SQLCODE
    w !,"????=",$TLEVEL
    &sql(START TRANSACTION)
    w !,"????, SQLCODE=",SQLCODE
    w !,"????==",$TLEVEL
    &sql(SAVEPOINT a)
    w !,"?????, SQLCODE=",SQLCODE
    w !,"????==",$TLEVEL
    &sql(COMMIT)
    w !,"????, SQLCODE=",SQLCODE
    w !,"????==",$TLEVEL
}
```

```
DHC-APP>d ##class(PHA.TEST.SQLCommand).SetTransaction3()

????????, SQLCODE=0
????=0
????????, SQLCODE=0
????=0
????, SQLCODE=0
????==1
?????, SQLCODE=0
????==2
????, SQLCODE=0
????==0
```

#SQL #Caché

源
URL:
https://cn.community.intersystems.com/post/%E7%AC%AC%E4%B8%83%E5%8D%81%E5%9B%9B%E7%AB%A0-sql%E5%91%BD%E4%BB%A4-set-transaction