

第 15 章 事务处理

具备事务处理功能是数据库区别于文件系统的重要特征，事务处理也是大型 DBMS 的核心部分，并发控制基于事务实现，数据库出现故障后的实例或介质恢复也都以事务为单位。

本章主要包括：

- 事务概念
- ACID 属性
- 事务控制命令
- 客户端的事务模式
- DDL 与 DCL 的处理方式
- 事务隔离级别
- SQL Server 的多版本数据技术

15.1 事务概念

事务是若干操作的集合，这个集合中的所有操作作为一个整体要么都完成要么都取消。

事务处理首先在美国航空的 SABRE 系统(由 IBM 开发)和 IBM 的 IMS 系统中实现，至今已应用于几乎所有的大中型数据库产品中。

15.2 ACID 属性

ACID 属性由 Jim Gray 和 Andreas Reuter 在 1980 年代初提出和完善。

关系数据库中的事务应该满足 ACID 属性，ACID 是四个英文词的词首字母，分别翻译为原子性，一致性，隔离性，持久性，其含义如下：

- Atomicity: 事务中的操作要么都完成，要么都取消，这些操作作为一个整体不可分。
- Consistency: 事务把数据库从一个一致状态转到另一个一致状态。一致状态是指数据库中的数据都是正确的，要保证事务一致性往往需要程序员的参与。
- Isolation: 事务对数据的修改直到提交后才对其他事务可见。
- Durability: 事务提交成功后，即使发生系统故障，其效果在数据库中也是永久的。由于性能原因，事务提交不是把修改的数据写入磁盘，需要某种机制保证这个特性。

15.3 控制命令

Oracle 和 SQL Server 中的主要事务控制命令如下：

- commit: 结束一个事务，且使事务中的修改永久化。
- rollback: 结束一个事务，把数据库恢复到事务开始之前的状态。
- savepoint/save transaction: 创建一个标志。Oracle 使用前者，SQL Server 使用后者。
- rollback to <savepoint>/rollback transaction <savepoint>: 与 savepoint/save transaction 结合使用，使事务回滚到指定位置。Oracle 使用前者，SQL Server 使用后者。
- set transaction: 设置事务属性，如 set transaction isolation level 设置隔离级别。

15.3.1 commit 背后

不同数据库产品执行 commit 操作的原理是类似的。

在上面的 ACID 属性中提到, commit 操作并不是把事务中被修改的数据写入磁盘永久保存。假如 100 个事务修改同一个对象, 若 commit 时就把其修改结果写入磁盘, 这 100 个事务会写入磁盘 100 次, 效率被无谓降低了, 只要把最后的修改结果写入磁盘就可以了。既然 commit 操作不会把事务中的修改结果写入磁盘, 那么它又做了哪些事情呢?

事务提交后, 主要发生了三件事情:

- 释放事务产生的锁。
- 把重做缓冲区数据写入磁盘。
- 把事务提交信息写入重做日志文件, 以表示进行数据库恢复时要达到的最后状态。

以上任务主要是第二项相对消耗资源。事务提交前, 重做缓冲区数据可能已经被写入多次(事务提交只是重做缓冲区内容写入磁盘的条件之一), 因此事务提交时间是很快的, 与事务中的操作数量及修改的数据量基本没关系。

事务提交之前, 其修改的部分数据可能已经被写入磁盘, 而提交之后, 可能部分被修改的数据尚未写入磁盘, 若发生断电等故障, 导致数据库意外关闭, 数据库处于不一致状态。数据库重新启动时, 若处于不一致状态, 会进行实例恢复, 把已经提交, 但尚未写入磁盘的数据应用联机重做日志文件写入数据文件(redo), 再把未提交却已写入磁盘的数据回滚(undo), 使数据库重新处于一致状态。

15.3.2 rollback 背后

执行 rollback 操作时, 除了释放锁以外, Oracle 使用 undo 表空间的 undo 数据替换被事务修改的数据, SQL Server 使用重做日志中的 undo 数据替换被事务修改的数据。

15.4 客户端的事务模式

根据事务的开始和结束方式, 客户端的事务模式可以分为:

- 自动提交模式
- 隐式模式
- 显式模式

我们下面分别说明其含义以及在 Oracle 的 SQL*Plus 和 SQL Server 的 sqlcmd 中, 各种模式是如何设置和使用的。

15.4.1 自动提交模式

自动提交模式是指客户端中执行的每一条 SQL 命令自动构成一个事务, 这个命令执行后自动提交, 下一条 SQL 命令又构成另外一个事务。

自动提交模式可以让事务在最短的时间内结束, 一般存在读写操作相互等待的 DBMS 会使用这种方式, 以最大限度地降低等待的发生。

Oracle 的多版本数据技术使得读取操作不必加锁, 从而不会发生读写等待。

默认情况下, SQL Server 未使用多版本数据技术, 会产生读写等待的情况, 为了降低等待的发生或缩短等待时间, 其 sqlcmd 客户端默认采用自动提交模式。

另外, SQL Server 的锁使用内存结构来管理, 如果事务持续时间过长, 可能会导致锁的个数增多, 从而耗费过多内存, 采用自动提交模式, 可以尽快释放锁占用的内存资源。Oracle 的锁不使用内存管理, 也不存在这种尽快释放锁的需要。

15.4.2 隐式模式

隐式模式下，客户端中执行的第一个 SQL 命令就开始了一个事务，直到执行 commit 或 rollback 命令才结束。

因为使用了多版本数据技术，Oracle 的读写操作不会相互等待，另外 Oracle 的锁也未使用内存结构管理，Oracle 不存在尽快结束事务的需要，其 SQL*Plus 默认采用隐式事务模式。

15.4.3 显式模式

显式模式以 begin transaction 作为事务开始标志，以 commit 或 rollback 作为结束标志。Oracle 的 SQL*Plus 不支持这种模式。

SQL Server 的 sqlcmd 在自动提交模式下，执行 begin transaction 就开启了显式模式，并开始了事务，直到 commit 或 rollback 才结束。

15.4.4 设置事务模式

在 SQL*Plus 中设置 autocommit 环境变量为 on，即可处于自动提交事务模式：

```
SQL> show autocommit
autocommit OFF
SQL> set autocommit on
```

设置 autocommit 为 on 后，在 SQL*Plus 中执行 SQL 语句的效果与 SQL Server 的 sqlcmd 默认的自动提交事务模式相同。

要恢复至隐式事务模式，只要重置 autocommit 为 off。

在 sqlcmd 中设置 implicit_transactions 环境变量为 on，即可处于隐式事务模式：

```
1> set implicit_transactions on
2> go
```

SQL Server 的 implicit_transactions 变量与 SQL*Plus 的 autocommit 的功能相同，但开启后的效果正好相反。设置 implicit_transactions 为 off，则 sqlcmd 重新处于自动提交模式，。

15.5 DDL 及 DCL 语句的处理方式

在 Oracle 中，执行 DDL 或 DCL 语句的前后会自动附加一个 commit 操作，这样，即使 DDL 或 DCL 语句失败，其前面的操作已经提交，如果 DDL 或 DCL 语句成功执行，其本身也作为一个事务提交了，执行 rollback 命令不会有任何效果。

我们首先验证在 DDL 语句执行之后，Oracle 自动执行了 commit 操作。

在连接 1 中，先查询 dept 表的初始数据：

```
SQL> conn scott/tiger
已连接。
SQL> select * from dept where deptno=40;
```

DEPTNO	DNAME	LOC
40	OPERATIONS	BOSTON

对 dept 表执行 update 操作：

```
SQL> update dept set loc='Beijing' where deptno=40;
```

然后创建表 t:

```
SQL> create table t(a int);
```

在连接 2 中, 查询表 t:

```
SQL> conn scott/tiger
已连接。
SQL> select * from t;
未选定行。
```

结果说明表 t 已经存在, 即连接 1 中的建表操作执行之后 Oracle 执行了 commit。
如何验证建表操作执行之前 Oracle 也发出了 commit 操作? 继续上面的实验过程。
在连接 2 中, 再查询 dept 表的内容:

```
SQL> select * from dept where deptno=40;
```

DEPTNO	DNAME	LOC
40	OPERATIONS	Beijing

我们发现, 连接 1 中的 update 操作也提交了, 那么如何验证这个效果是由建表操作之前的 commit 引起的, 而不是建表之后的 commit 引起的呢?

依照上面实验过程, 要验证 DDL 语句执行之前发出了 commit 操作, 只要让 DDL 语句在执行时出错即可, 这样, 其之前的 commit 操作会发出, 而之后的 commit 操作由于 DDL 本身执行出错, 不会执行。

要让建表语句执行出错, 只要新表的名称与当前模式下的某个表相同就可以了, 这样, 建表语句没有语法错误, 执行时 Oracle 才会发现与数据库中已有的对象名称重复(如果 SQL 语句出现语法错误, 对其所在的事务不会产生任何影响, 即不会执行 rollback 操作, Oracle 和 SQL Server 对这种情况的处理方式是相同的)。

在连接 1 中, 对 dept 表执行 update 操作:

```
SQL> update dept set loc='Guangzhou' where deptno=40;
```

然后创建 dept 表:

```
SQL> create table dept(a int);
create table dept(a int)
*
第 1 行出现错误:
ORA-00955: 名称已由现有对象使用
```

在连接 2 中, 查询连接 1 修改的记录:

```
SQL> select * from dept where deptno=40;
```

DEPTNO	DNAME	LOC
40	OPERATIONS	Guangzhou

我们发现, 连接 1 中的 update 操作已经提交, 从而验证了在建表操作执行之前, Oracle 发出了 commit 操作。

关于 DCL 的验证过程与此类似, 这里不再赘述。

SQL Server 对 DDL 或 DCL 语句并未像 Oracle 那样处理, 而是把 DDL 及 DCL 语句与 DML 语句一样对待, 也就是执行 DDL 或 DCL 之前及之后都不会发出 commit 操作, 从而 DDL 及 DCL 语句也可以回滚。

如果 DDL 或 DCL 语句在执行过程中失败，会自动发出 rollback 操作，与 Oracle 相比，这种处理方式稍显不友好。

这可以由下面实验过程验证。

开始显式事务后，查询 dept 表的初始值：

```
1> begin transaction
2> go
1> select * from dept where deptno=50
2> go
```

DEPTNO	DNAME	LOC
50	OPERATIONS	Beijing

执行 update 操作，修改以上记录：

```
1> update dept set loc='Guangzhou' where deptno=50
2> go
```

假定已经存在表 t，再次创建表 t：

```
1> create table t(a int)
2> go
```

消息 2714，级别 16，状态 6，服务器 LAW_X240，第 1 行
数据库中已存在名为 't' 的对象。

重新查询 dept 表的记录，我们发现上面的 update 操作已经被回滚：

```
1> select * from dept where deptno=50
2> go
```

DEPTNO	DNAME	LOC
50	OPERATIONS	Beijing

如果建表操作成功执行呢？

重新进行上面实验过程，这次让建表操作成功执行：

```
1> begin tran
2> go
1> update dept set loc='Guangzhou' where deptno=50
2> go
```

创建表 deptnew：

```
1> create table deptnew(a int)
2> go
```

然后执行 rollback 命令：

```
1> rollback
2> go
```

查询前面被修改的记录：

```
1> select * from dept where deptno=50
2> go
```

DEPTNO	DNAME	LOC
50	OPERATIONS	Beijing

我们发现，update 操作已经被回滚。

再查询新建的 deptnew 表：

```
1> select * from deptnew
2> go
消息 208, 级别 16, 状态 1, 服务器 LAW_X240, 第 1 行
对象名 'deptnew' 无效。
```

从上面错误信息，我们可以看到 deptnew 表不存在了，显然建表操作也被回滚。
关于 DCL 的验证与此类似，这里不再赘述。

15.6 事务隔离级别

设置隔离级别可解决数据库应用中的脏读和不可重复读问题，这是并发控制的典型问题。

15.6.1 脏读和不可重复读问题

脏读是指一个连接读取了其他尚未提交的事务修改的数据。脏读破坏了事务的原子性，读取到的是事务进行过程中的中间结果，若此结果与事务结束时的结果不同，则此连接读取的是错误数据，如果以此错误数据为依据继续执行另外的任务，则可能造成一连串的错误。

下面以超市收银为例说明脏读的产生过程。

一顾客购买了 2 支钢笔、10 个笔记本，总价 100 元，收银员在连接 A 完成收银操作，超市采购员在连接 B 查询商品库存，确定某种商品是否需要进货。假定收银开始前，钢笔库存量为 100，笔记本库存量为 300，下面是两个连接在不同时间进行的操作：

表 15-1 操作顺序

操作时间	连接 A	连接 B
t1	扫描商品条形码，计算总价	
t2	修改钢笔库存量：100-2=98	
t3	修改笔记本库存量：300-10=290	
t4		查询钢笔库存量：98
t5		查询笔记本库存量：290
t6	超市银行账号余额+100	
t7	顾客银行卡账号余额-100	
t8	顾客银行卡余额小于 100，付款失败	
t9	以上修改操作全部撤销	
t10	钢笔库存量恢复 100	
t11	笔记本库存量恢复 300	
t12	操作结束	

由上述过程可以看到，因为连接 2 在事务未结束的时候读取了其中间结果，导致其读取的数据与最终结果不同，如果以此数据为依据决定是否进货显然是错误的。

不可重复读是指一个事务中的查询操作因为分为多个步骤，导致其结果既包括了这个事务开始之前的数据，也包括了这个事务开始之后的数据，从而在最后得到了错误的查询结果。

下面示例中的数据包括三个银行账号 acc1，acc2 及 acc3，其余额分别为 100,200,300，如下表所示：

表 15-2 各账号初始值

账号	余额
acc1	100

acc2	200
acc3	300

用户 A 查询银行三个账号的余额总和，先设置 sum 变量为 0，然后依次查询三个账号余额，累加至 sum 变量。用户 B 执行转账操作，由 acc3 账号转账 100 至 acc1。

两个用户的执行步骤如下表所示：

表 15-3 操作步骤

操作时间	用户 A	用户 B
t1	设置 sum=0	
t2	查询 acc1 账号的余额：100	
t3	100 累计至 sum：sum=100	
t4	查询 acc2 账号的余额：200	
t5	200 累加至 sum：sum=300	
t6		由 acc3 转账 100 至 acc1
t7		修改 acc1 账号余额为 200
t9		修改 acc3 账号余额为 200
t10		提交事务
t11	查询 acc3 账号余额：200	
t12	200 累加至 sum：sum=500	

用户 A 的多次查询操作不存在脏读的情况，但最后得到的总和却是错误的。这里出现错误的原因在于最后的总和既包括了事务开始之前的数据，也包括了事务开始之后、由其他事务提交的结果。

15.6.2 SQL 标准中的事务隔离级别

事务具备隔离性，可以通过设置事务隔离级别控制一个事务与其他事务的隔离程度。

在 SQL 标准中，事务隔离级别有四种，从低到高分别为：

- read uncommitted：事务可以读取到其他事务未提交的修改结果。
- read committed：事务只能读取到其他事务提交后的修改结果。
- repeatable read：如果在事务中进行两次查询，则第一次查询的结果在第二次查询中保证不会发生改变，但第二次查询结果中的记录条数可能会多于第一次查询。
- serializable：事务中的两次相同查询的结果相同(本事务中的修改操作结果除外)。

一个事务隔离级别越高，用户越不会感觉到其他并发用户的存在。

Oracle 数据库提供了这四种事务隔离级别中的 read committed 及 serializable，SQL Server 数据库则提供了所有这四种隔离级别。

连接的隔离级别设置为 read uncommitted，SQL Server 的读取操作不会使用共享锁，但会产生脏读。

设置为 read committed 可以解决脏读问题，但会产生不可重复读，另外，在数据库默认设置下，SQL Server 在 read committed 隔离级别的的读取操作会使用共享锁，导致产生读写等待，这是 SQL Server 支持 read uncommitted 的原因。

为了解决 read committed 下产生的不可重复读问题，只需把隔离级别设置为 serializable，但此级别下，SQL Server 的读取操作会锁住整个表，所以 SQL Server 支持 repeatable read 级别，可以不锁住整个表，也降低不可重复读的发生，但此级别不能杜绝不可重复读。

Oracle 的读取操作不会使用锁，也没有支持 read uncommitted 和 repeatable read 这两种隔离级别的必要。

Oracle 的 SQL*Plus 与 SQL Server 的 sqlcmd 的默认隔离级别都是 read committed，可以

满足大部分需要。

Oracle 和 SQL Server 设置隔离级别可以执行 `set transaction isolation level` 命令实现，下面是在 Oracle 的 SQL*Plus 中设置 `read committed` 隔离级别：

```
SQL> set transaction isolation level read committed;
```

Oracle 查询当前连接的隔离级别可以执行下面命令：

```
SQL> select
  2  case bitand(t.flag, power(2, 28))
  3      when 0 then 'read committed'
  4      else 'serializable'
  5  end as isolation_level
  6  from v$transaction t
  7  join v$session s on t.addr = s.taddr
  8  and s.sid = sys_context('USERENV', 'SID')
  9  /

ISOLATION_LEVEL
-----
read committed
```

SQL Server 查询当前连接的隔离级别，可以执行 `dbcc useroptions` 命令，除了显示当前的隔离级别外，也显示了其他选项的值：

```
C:\Windows\system32>sqlcmd -dlaw -Y23
1> dbcc useroptions
2> go
Set Option              Value
-----
textsize                 4096
language                 简体中文
dateformat               ymd
datefirst                7
lock_timeout             -1
ansi_null_dflt_on        SET
ansi_warnings            SET
ansi_padding             SET
ansi_nulls               SET
concat_null_yields_null SET
isolation level          read committed

(11 行受影响)
DBCC 执行完毕。如果 DBCC 输出了错误信息，请与系统管理员联系。
```

15.6.3 read committed 隔离级别

先看 Oracle 的处理方式。

在一个连接中设置 `read committed` 后，这个连接只能读到其他用户提交后的数据，如果读取的数据已经被其他连接中的一个事务修改，但是这个事务还未提交，则只能读到这些数据在这个事务开始之前的状态。

下面我们做一个实验验证上述结论。

在连接 1 中，以 `scott` 用户修改 `dept` 表中 `deptno` 为 10 的记录的 `loc` 列值，其初始值为：

```
SQL> select * from dept where deptno=10;
```

DEPTNO	DNAME	LOC
--------	-------	-----


```
-----  
10 ACCOUNTING NEW YORK
```

对其执行 update 操作:

```
SQL> update dept set loc='Dallas'where deptno=10;
```

在这个连接中查询确认修改的结果:

```
SQL> select * from dept where deptno=10;
```

```
DEPTNO DNAME      LOC  
-----  
10 ACCOUNTING Dallas
```

在连接 2 中执行上述同样的查询:

```
SQL> select * from dept where deptno=10;
```

```
DEPTNO DNAME      LOC  
-----  
10 ACCOUNTING NEW YORK
```

我们发现, 查询结果是连接 1 中 update 操作所在事务开始之前的数据。连接 1 对数据修改后, 其修改之前的旧数据存入了 undo 表空间, 连接 2 查到的是这些 undo 数据。

返回连接 1, 执行 commit 命令, 提交事务:

```
SQL> commit;
```

在连接 2 中, 重新执行刚才的查询, 则可以查到新结果:

```
SQL> select * from dept where deptno=10;
```

```
DEPTNO DNAME      LOC  
-----  
10 ACCOUNTING Dallas
```

再看 SQL Server 对 read committed 隔离级别的处理方式。

一个连接设置 read committed 隔离级别后, 若查询其他连接还未提交的修改结果, 则会发生等待。

与前面实验过程类似, 在连接 1 中, 开始一个事务, 并在其中修改 dept 表中 deptno 为 10 的记录, 先确认其当前值:

```
1> begin tran  
2> go  
1> select * from dept where deptno=10  
2> go
```

```
DEPTNO DNAME      LOC  
-----  
10 ACCOUNTING NEW YORK
```

执行 update 操作:

```
1> update dept set loc='Dallas'where deptno=10  
2> go
```

在连接 2 中, 执行下面命令查询上面修改的数据:

```
1> select * from dept where deptno=10
```

```
2> go
```

可以发现，没有任何查询结果出现，而是等待。
回到连接 1，提交事务：

```
1> commit
2> go
```

连接 2 已经不再等待，查询结果自动显示出来：

```
1> select * from dept where deptno=10;
2> go
DEPTNO DNAME          LOC
-----
10 ACCOUNTING Dallas
```

15.6.4 serializable 隔离级别

先看 Oracle 的处理方式。

在客户端中设置事务隔离级别为 `serializable` 后，事务中的任何一个查询的结果都是这个事务开始之前的状态，假定此事务为 A，在 A 事务开始后，即使有其他连接提交了事务，在这些事务中修改的数据也不会被 A 事务内的查询查到，这是 `serializable` 隔离级别与 `read committed` 隔离级别显著的不同。

显然，设置 `serializable` 隔离级别后，事务中的任意两个相同查询的结果也是相同的。

我们通过下面的实验验证上述说法。

在连接 1 设置 `serializable` 隔离级别：

```
SQL> set transaction isolation level serializable;
```

执行第一次查询：

```
SQL> select * from dept where deptno=10;

DEPTNO DNAME          LOC
-----
10 ACCOUNTING Dallas
```

在连接 2 中，修改刚才查到的记录的 `loc` 字段值：

```
SQL> update dept set loc='NEW YORK' where deptno=10;
```

然后提交：

```
SQL> commit;
```

回到连接 1，执行相同的查询，我们发现其结果与第一次是相同的：

```
SQL> select * from dept where deptno=10;

DEPTNO DNAME          LOC
-----
10 ACCOUNTING Dallas
```

在连接 1 中，执行 `commit` 命令以结束事务：

```
SQL> commit;
```

再次执行上面的查询：

```
SQL> select * from dept where deptno=10;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK

刚才的事务已经执行 `commit` 命令而结束，这个查询在连接 1 中又开始了一个新事务，可以发现，查询结果已经是连接 2 提交的修改结果。

连接 1 中执行的第二次查询的结果来自 `undo` 表空间，如果有其他多个连接对连接 1 中第一次查询的数据进行了多次修改，则 `undo` 表空间会保存这所有多个版本的数据。

再考察 SQL Server 的情形。

在客户端连接中设置 `serializable` 隔离级别后，在这个连接的事务(假定为 A 事务)中执行一个查询后，会对被查询的表附加共享锁(S 锁，若表上没有聚簇索引)或范围共享锁(Range-S 锁，若表上存在聚簇索引)，因为其他用户对这个表的数据进行修改(update、delete、insert)会导致对此表附加 IX 锁，而 IX 锁与 S 锁互斥，所以这些修改操作会一直等到事务 A 结束才能执行。设置 `serializable` 隔离级别后，事务中查询到的数据不能被其他事务修改，这个事务中的两次查询的结果显然是相同的。

我们通过实验说明以上结论。

在连接 1 设置 `serializable` 隔离级别，事务开始后，执行第一次查询：

```
1> set transaction isolation level serializable
2> go
1> begin tran
2> select * from dept where deptno=10
3> go
DEPTNO DNAME      LOC
-----
10 ACCOUNTING Dallas
```

在连接 2 中，修改 `dept` 表的数据：

```
1> delete from dept where deptno=10
2> go
```

可以发现，这个连接发生了等待。

回到连接 1 执行第二次相同的查询，结果与第一次相同：

```
1> select * from dept where deptno=10
2> go
DEPTNO DNAME      LOC
-----
10 ACCOUNTING Dallas
```

15.7 SQL Server 的多版本数据技术

从 2005 版本开始，SQL Server 开始支持多版本技术，以解决备受诟病的读写等待问题，但是这个功能直到 2016 版本，默认并未开启。下面说明 SQL Server 如何使用多版本技术。

15.7.1 设置 `read_committed_snapshot` 改变 `read committed` 效果

在数据库中设置 `read_committed_snapshot` 选项参数为 `on`，当处于 `read committed` 隔离级别时，会启用行版本控制功能的 `read committed` 隔离级别。

启用行版本控制功能的 `read committed` 后，与 Oracle 类似，当一个用户修改表中的数据

时，这些数据的旧映像会存入 tempdb 中，这类似于 Oracle 的 undo 表空间功能，如果一个处于 read committed 隔离级别的连接查询的数据正在被其他用户修改，则这个连接会查询位于 tempdb 中的旧映像数据，读取操作不再需要使用锁，不再发生等待。以下是简单测试。

如果不启用 read_committed_snapshot 参数，则一个查询读到其他连接未提交的修改结果时，会发生等待。

在连接 1 开始一个事务，并修改 dept 表：

```
1> begin tran
2> update dept set loc='beijing' where deptno=30
3> go
```

在连接 2 中设置 read committed 隔离级别：

```
1> set transaction isolation level read committed
2> go
```

继续在连接 2 执行下面查询，则发生等待：

```
1> select * from dept
2> go
```

在连接 1 中启用 read_committed_snapshot 参数(注意，设置之前要退出连接 2，否则会发发生等待)：

```
1> alter database hr set read_committed_snapshot on
2> go
```

然后执行下面查询：

```
1> select * from dept where deptno=10
2> go
deptno  dname      loc
-----  -
10  ACCOUNTING NEW YORK
```

在连接 2 中，开始一个事务，并对 dept 表执行 update 操作，但不提交：

```
1> begin tran
2> update dept set loc='beijing' where deptno=10
3> go
```

回到连接 1，执行第二次相同的查询：

```
1> select * from dept where deptno=10
2> go
deptno  dname      loc
-----  -
10  ACCOUNTING NEW YORK
```

可以发现，第三条记录的 loc 字段的值与第一次的查询结果相同，第二次查询结果来自于 tempdb 的 version store 中的旧版本数据，我们可以从下面的查询结果发现，这时 version store 中已经有数据了：

```
1> select version_store_reserved_page_count from sys.dm_db_file_space_usage
2> go
version_store_reserved_page_count
-----
```

设置 “read_committed_snapshot” 为 on 后，在 read committed 隔离级别下，执行 dbcc useroptions 命令的显示结果为 “read committed snapshot”。

15.8.2 设置 allow_snapshot_isolation 改变 serializable 效果

设置 allow_snapshot_isolation 参数为 on 后，可以在连接中设置 snapshot 隔离级别，设置这个隔离级别后，与 Oracle 的 serializable 效果相同，即事务中的读取操作不会使用锁，只从 tempdb 数据库读取事务开始之前的旧版本数据，以下是简单测试。

设置 allow_snapshot_isolation 为 on:

```
1> alter database hr set allow_snapshot_isolation on
2> go
1> begin tran
2> go
```

执行第一次查询:

```
1> select * from dept where deptno=10
2> go
deptno  dname      loc
-----  -
10  ACCOUNTING  NEW YORK
```

接下来，启动另外一个连接，在其中修改 dept 表的数据:

```
1> update dept set loc='beijing' where deptno=10
2> go
```

我们知道，这个连接是默认的自动提交模式，所以上面的 update 操作已经作为一个事务提交了。回到连接 1，执行同样的查询，我们发现第二次的查询结果未发生任何改变:

```
1> select * from dept where deptno=10
2> go
deptno  dname      loc
-----  -
10  ACCOUNTING  NEW YORK
```

显然，上面连接 2 中的 update 操作导致产生了多版本数据，而连接 1 的第二次的查询结果是从 tempdb 中的 version store 的旧版本数据得来的，这从下面对 version store 的查询结果也可以验证:

```
1> select version_store_reserved_page_count from sys.dm_db_file_space_usage
2> go
version_store_reserved_page_count
-----
```

8