[1. ACID 3](#_Toc462774330)

[1-1. 定义 3](#_Toc462774331)

[1-2. Atomicity 3](#_Toc462774332)

[1-3. Consistency 3](#_Toc462774333)

[1-4. Isolation 3](#_Toc462774334)

[1-5. Durability 4](#_Toc462774335)

[2. SQL Server 2008 R2 支持的事务隔离级别 4](#_Toc462774336)

[2-1. 小结 4](#_Toc462774337)

[2-2. 详细说明 5](#_Toc462774338)

[2-3. 为什么会支持有并发问题的隔离级别 5](#_Toc462774339)

[3. SQL Server 的每个Query都是一个独立的Session 5](#_Toc462774340)

[4. 事务隔离级别是针对Session的而不是整个数据库的 6](#_Toc462774341)

[4-1. 如何更改 6](#_Toc462774342)

[4-2. 如何查看当前session的事务隔离级别 6](#_Toc462774343)

[4-3. 如何查看所有session的事务隔离级别 6](#_Toc462774344)

[5. 事务的隔离级别内部是如何实现的 6](#_Toc462774345)

[5-1. 是通过锁机制 6](#_Toc462774346)

[5-2. 隔离级别与锁持续的时间 6](#_Toc462774347)

[6. 并发带来的几个问题就解决方法 7](#_Toc462774348)

[6-1. 脏读(Dirty reads (Uncommitted Dependency)) 7](#_Toc462774349)

[6-1-1. 提升隔离级别至 Read Commited 7](#_Toc462774350)

[6-2. 不可重复读(non-repeatable read) 7](#_Toc462774351)

[6-2-1. 提升隔离级别至 Repeatable Read 7](#_Toc462774352)

[6-3. 幻影读(phantom read) 7](#_Toc462774353)

[6-3-1. 提升隔离级别至Serilizable 7](#_Toc462774354)

[6-4. 更新丢失(Update Lost) 7](#_Toc462774355)

[6-4-1. 一种是用乐观的锁机制，即基于时间戳的解决方案 8](#_Toc462774356)

[6-4-2. 将update 写成一句而不是分成读写两部分 9](#_Toc462774357)

[7. 模拟并发问题的一个有用的函数 Waitfor 12](#_Toc462774358)

[8. SQL Server 锁机制 12](#_Toc462774359)

[8-1. 独占锁，共享锁和更新锁 12](#_Toc462774360)

[8-2. 悲观锁和乐观锁 12](#_Toc462774361)

[8-3. 锁的力度 13](#_Toc462774362)

# ACID

## 定义

In computer science, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction.

ACID是针对数据库事务的，所以ACID更准确的说话是数据库事务的ACID特性。所以原子性，一致性，隔离性，持久性都是指的数据库事务。

一个事务指一个逻辑的操作，比如一个账户的转账操作。物理上可能涉及到多张表的多个操作

For example, **a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction**.

## Atomicity

Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

原子性，要么全成功，要么全失败。强调的是事务能包含多个操作，一个事务包含多个操作，这些操作要么全部执行，要么全都不执行。

## Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

任何事务执行前后，数据库都应该从一个有效状态迁移到下一个有效状态，即要满足数据完整性要求。（比如主键限制，外键限制，级联删除等）。主要针对的是数据库写操作，因为读不影响数据库状态

## Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. **Providing isolation is the main goal of concurrency control**. Depending on concurrency control method (i.e. if it uses strict - as opposed to relaxed - serializability), **the effects of an incomplete transaction might not even be visible to another transaction**.

主要是为处理并发问题而设计的属性，

## Durability

The durability property ensures that **once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors**. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

事务一旦提交，就不能再丢失。

# SQL Server 2008 R2 支持的事务隔离级别

## 小结



可串行化(Serializable) 同可重复读相比较，主要是避免了幻影读的问题。就是当你已读的情况下，lock别的事务的insert操作。 事务串行化执行 （什么是事务串行化执行？并不是真的所有事务都单线程了，只能一个一个的执行。而是从效果上来看好像是串行化执行的。任何可能彼此产生影响的事务都不能并行执行，比如一个读了，另一个要更新和插入。但是两个读，多个根本就是不同的表，有个毛的关系）

可重复读(Repeatable reads) lock 的可能是自己，也可能是别人。如果自己已经读了，那别人试图写的时候就被lock了。如果别人已经写了，而且尚未提交或者rollback， 那么自己再试图去读的时候，自己就被lock了。那如果别人写的是记录2，是我试图读的是同一张表的记录1 呢，还会被锁么，这要取决于锁的粒度了。如果是整张表的锁，那么不行。如果是行锁，那么可以。

为什么repeatable reads 会产生幻影读的问题。如果别人先更改，无论是insert 还是update，在别人commit或者rollback之前，你再试图读取的时候都会让自己被lock。

但是针对第二个场景，如果你先读，别人的update 操作是会被lock的，但是别人的insert操作是可以执行的。这时，如果别人提交了，那么你再读的话，是可以读到新提交的值的，这就产生了幻影读的问题。

当然，如果你先读，然后别人insert，在别人提交或者rollback之前，如果你再读的话，自己会被lock的，因为已经不符合可重复读的限制了。

提交读(Read committed) 。read committed 不是说你第二次读的时候，如果别人还没提交，你就读到之前的值。而是你第二次读的时候，如果别人已经update了，但是还没有提交，你就只能被block了，你只能等着，要么对方提交，要么rollback之后，才能继续往下执行。 Read committed lock 的是自己

未提交读(Read uncommitted)

Serializable 是最高级别的事务隔离级别。

Snapshot(快照) 和 Read Commited Snapshot 是基于rowversion（行版本控制）的两个事务隔离级别

Snapshot 的话，如果别人先更新了，你去读是要被lock的。因为别人加了锁。

它解决的只是当你先读了，别人再更新的情况下，别人不用被lock了，因为你读的是一个版本，别人更新的是另一个。这样不管别人更新完之后有没有提交，你都可以再次读，两次返回的结果都是一样的。（对于read committed snapshot 场景下，如果你先读了，别人更新，这时没问题的，别人不会被lock，但是当你试图再次去读的时候，你就要被lock了，只有等别人commit或者rollback之后，你才能继续往下执行）

还是在别人先更新的情况下，如果你再试图更新，先你会被block，因为别人更新的时候会先加锁，然后如果别人commit或者rollback了， 你会直接报错，而不是继续执行。

Snapshot 是一个可以使用set transaction isolation level snapshot 设置的事务隔离级别，但是用之前先要在整个数据库级别打开snapshot开关(Alter database dbname set allow\_snapshot\_isolation on)， 数据库会在temp db 做一些初始化的工作来允许snapshot， 基于版本

Read committed snapshot 是一个隔离级别，也需要在用之前在数据库级别打开开关。

Alter database dbname set read\_committed\_snapshot on. 但是， 不能用set transaction isolation level read committed snapshot， 只能用set transaction isolation level read committed。 也就是在这个开关打开的前提下，read committed 默认使用snapshot的形式实现的。

## 详细说明

<http://blog.itpub.net/13651903/viewspace-1082730/>

<http://www.cnblogs.com/chillsrc/archive/2013/04/20/3032787.html>

http://www.cnblogs.com/chenmh/p/3998614.html

## 为什么会支持有并发问题的隔离级别

1. 并非所有系统的数据库都是并发访问的,虽然这种情况比较少（比如自己搭个模拟的环境学学SQL啥的），当然怎么设置都可以，不存在并发问题（企业级的应用都应该是考虑并发的）
2. 有时候是在准确性和运行效率的一种Balance，折衷。

# SQL Server 的每个Query都是一个独立的Session

互相之间的事务隔离级别互不影响，可以用开多个Query窗口的方式模拟并发操作

# 事务隔离级别是针对Session的而不是整个数据库的

数据库有自己的默认的事务隔离级别（SQL Server 是Read Commited），但是每个connection 都可以自己更改的事务隔离级别。

## 如何更改

set transaction isolation level read uncommitted

go

## 如何查看当前session的事务隔离级别

SELECT transaction\_isolation\_level

FROM sys.dm\_exec\_sessions

where session\_id = @@SPID

SELECT CASE transaction\_isolation\_level

WHEN 0 THEN 'Unspecified'

WHEN 1 THEN 'ReadUncommitted'

WHEN 2 THEN 'ReadCommitted'

WHEN 3 THEN 'Repeatable'

WHEN 4 THEN 'Serializable'

WHEN 5 THEN 'Snapshot' END AS TRANSACTION\_ISOLATION\_LEVEL

FROM sys.dm\_exec\_sessions

where session\_id = @@SPID

## 如何查看所有session的事务隔离级别

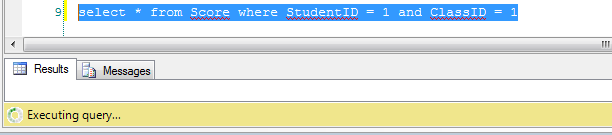
SELECT \*

FROM sys.dm\_exec\_sessions

# 事务的隔离级别内部是如何实现的

## 是通过锁机制

事务内的每一条语句在执行之前，SQL Server 都会试图去取得被影响对象的锁。如果取不到，SQL Server 就会等待。正如同下面



## 隔离级别与锁持续的时间

在基于锁的并发控制中，隔离级别决定了锁的持有时间。

"C"-表示锁会持续到事务提交。 "S" –表示锁持续到当前语句执行完毕。



# 并发带来的几个问题就解决方法

## 脏读(Dirty reads (Uncommitted Dependency))

脏读不是说读取了旧的数据，恰恰是读了最新的数据，就是读了别人还没有提交的数据。

这个同Cache机制里面的脏读还不太一样，那个应该是读了旧的数据

因为别的事务最终由可能是rollback而不是commit更新

### 提升隔离级别至 Read Commited

从而避免脏读

## 不可重复读(non-repeatable read)

在一次事务中，当**一行数据**获取两遍得到不同的结果表示发生了“不可重复读(non-repeatable read)”.

还有就是读完之后，在执行下一次读之前，有别的事务更新了数据并提交

所以不可重复读至少表名事务需要进行两次以上的读，如果只进行一次读，那么不管隔离级别怎样，应该都不会出现不可重复读的险象。

脏读算不算不可重复读的范畴，要看脏读之后是否还有后续的读。如果脏读之后直接更新而没有二次读，有可能的影响是Update Lost（更新丢失），而不会有不可重复读的现象。

### 提升隔离级别至 Repeatable Read

## 幻影读(phantom read)

在事务执行过程中，当两个完全相同的查询语句执行得到不同的结果集。这种现象称为“幻影读(phantom read)”

“幻影读(phantom read)”是不可重复读(Non-repeatable reads)的一种特殊场景：当事务1两次执行SELECT ... WHERE检索一定范围内数据的操作中间，事务2在这个表中创建了(如INSERT)了一行新数据，这条新数据正好满足事务1的“WHERE”子句。

### 提升隔离级别至Serilizable

## 更新丢失(Update Lost)

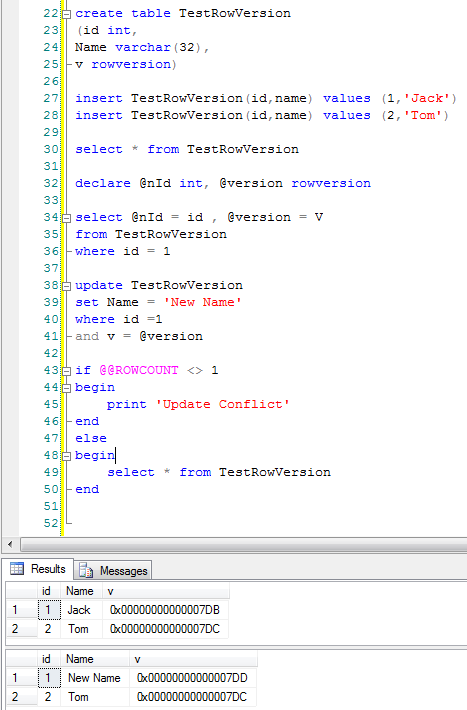
一个事务对某条数据的更新被另一个事务的对同一条数据的更新所覆盖，从而造成更新丢失。

其中一个场景是一个页面先load过来一批数据，然后过了一段时间之后，用户选中一条进行更新。而在这之前数据可能已经被另一个用户更新过了。

还有一种场景是两个事物同时在做update 操作，但在update之前先要经过一段计算算出要更新的值。假若第一个事务在计算值时，第二个事务已经更新完毕并提交，那么第一个事务再去更新时，第二个事务的值就会丢失。

### 一种是用乐观的锁机制，即基于时间戳的解决方案

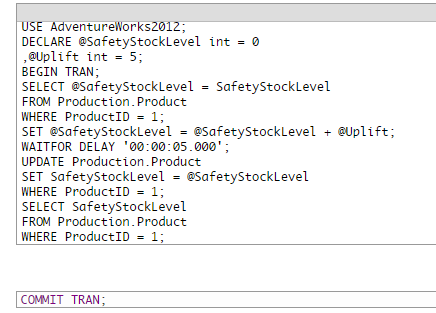
The usual way to handle this is to include a column of data type TIMESTAMP (ROWVERSION) in the table and use optimistic locking to verify that the row has not changed since you selected the row to be updated.

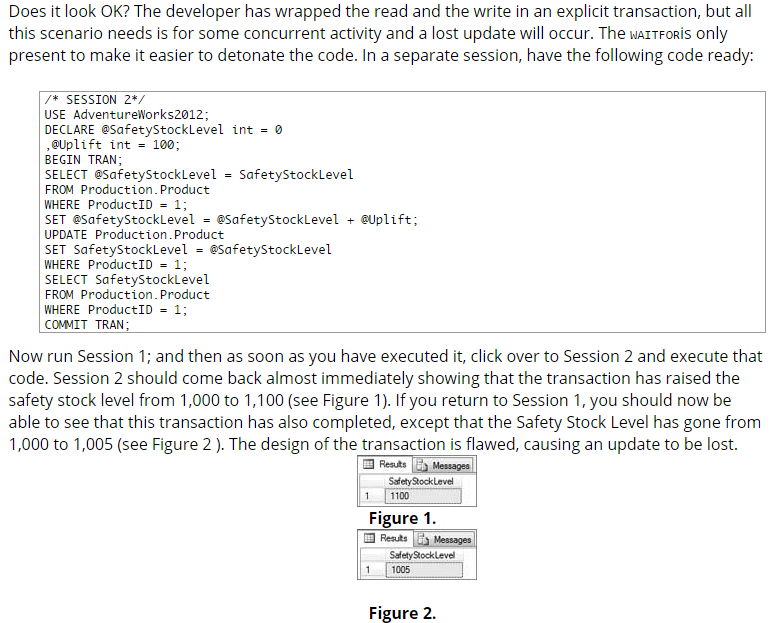


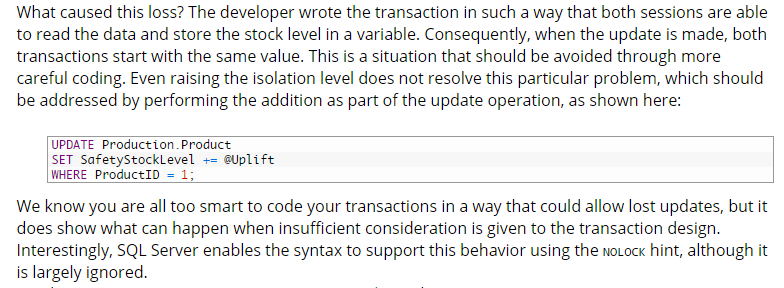
### 将update 写成一句而不是分成读写两部分

原因是当写成一句时，当SQL Server 试图去Update时，会先去加锁，然后才会读现在的值，然后再相加。另一个事务无法在同时获得更新锁

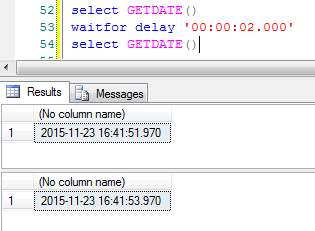
A lost update occurs when two processes read the same data and then try to update the data with a different value. Consider a scenario in which you and your partner have the romantic notion of a joint bank account. On pay day, your respective employers both deposit your salaries into the joint account. To perform the update, each process reads the data. At the time of the payments, all is well in the world and you have an outstanding balance of $10,000. Each process therefore reads $10,000 as its starting point. Your employer attempts to update the $10,000 figure with your monthly salary of $2,000, but at the same time your partner’s employer updates the sum with his or her salary of $4,000. Your partner’s salary is added just before yours, updating the $10,000 balance to $14,000. Your payment then runs and updates the $10,000 balance to $12,000. A look at the ATM shows $12,000. The first update has been lost, and even worse, it represented the bigger update! This situation is one that the SQL Server platform handles automatically, regardless of the isolation level. However, database developers can introduce this behavior themselves by performing an update in two steps, rather than one. Consider this example:







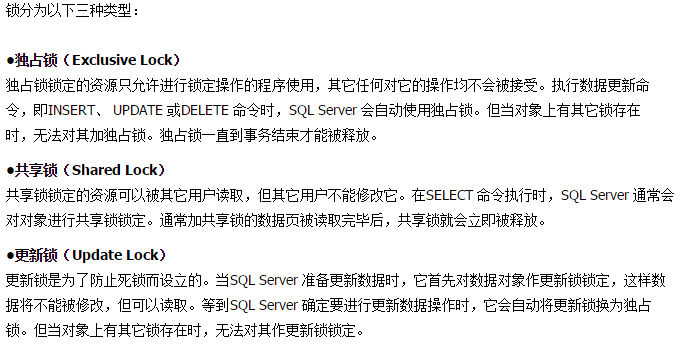
# 模拟并发问题的一个有用的函数 Waitfor



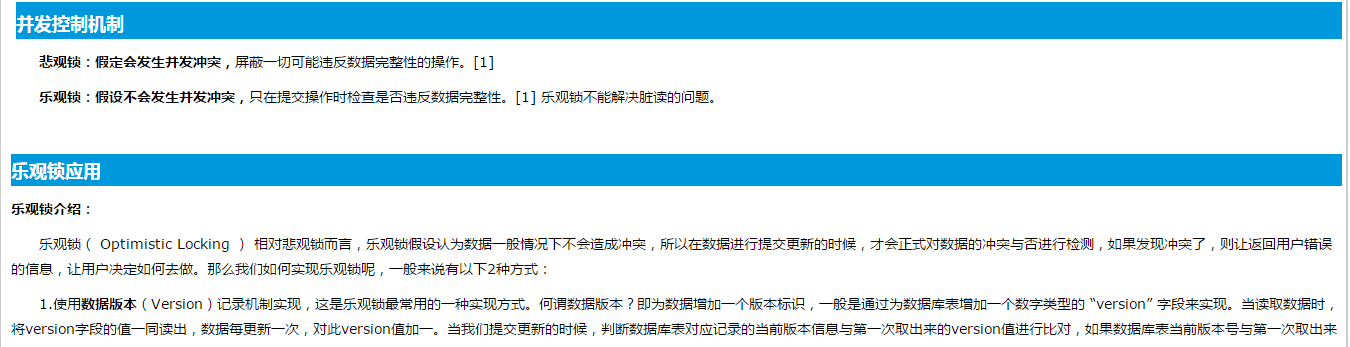
# SQL Server 锁机制

锁的类别有两种分法

## 独占锁，共享锁和更新锁



## 悲观锁和乐观锁







## 锁的力度