**DB2锁**

**锁的工作原理**

**我们知道DB2通过使用锁 把事务彼此隔离开来。锁是一种用来将数据资源与单个事务关联起来的机制，其用途是当某个资源与拥有它的事务关联在一起时，控制其他事务如何与该资源进行交互。（我们称与被锁定的资源关联的事务持有 或拥有该锁。）DB2**[**数据库**](http://lib.csdn.net/base/14)**管理程序用锁来禁止事务访问其他事务写入的未提交数据（除非使用了未提交的读隔离级别），并禁止其他事务在拥有锁的事务使用限制性隔离级别时对这些 行进行更新。一旦获取了锁，在事务终止之前，就一直持有该锁；该事务终止时释放锁，其他事务就可以使用被解锁的数据资源了。**

**如果一个事务尝试访问数据资源的方式与另一个事务所持有的锁不兼容（稍后我们将研究锁兼容性），则该事务必须等待，直到拥有锁的事务终止为止。这被称为锁等待 事件。当锁等待事件发生时，尝试访问数据资源的事务所做的只是停止执行，直到拥有锁的事务终止和不兼容的锁被释放为止。**

**锁的属性**

**所有的锁都有下列基本属性：  
object：object 属性标识要锁定的数据资源。DB2 数据库管理程序在需要时锁定数据资源（如表空间、表和行）。  
size：size 属性指定要锁定的数据资源部分的物理大小。锁并不总是必须控制整个数据资源。例如，DB2 数据库管理程序可以让应用程序独占地控制表中的特定行，而不是让该应用程序独占地控制整个表。  
duration：duration 属性指定持有锁的时间长度。事务的隔离级别通常控制着锁的持续时间。  
mode：mode 属性指定允许锁的拥有者执行的访问类型，以及允许并发用户对被锁定数据资源执行的访问类型。这个属性通常称为锁状态。**

**锁状态**

锁状态确定允许锁的拥有者执行的访问类型，以及允许并发用户对被锁定数据资源执行的访问类型。表 1 说明了可用的锁状态，按照控制递增的次序排列。  
**表 1. 锁状态**

锁状态（模式）适用对象描述

|  |  |  |
| --- | --- | --- |
| 意向无（Intent None，IN） | 表空间和表 | 锁的拥有者可以读取被锁定表中的数据（包括未提交数据），但不能更改这些数据。在这种模式中，锁的拥有者不获取行级锁；因此，其他并发应用程序可以读取和更改表中的数据。 |
| 意向共享（Intent Share，IS） | 表空间和表 | 锁的拥有者可以读取被锁定表中的数据，但不能更改这些数据。同样，因为锁的拥有者不获取行级锁；所以，其 他并发的应用程序可以读取和更改表中的数据。（当事务拥有表上的意向共享锁时，就在它所读取的每个行上进行共享锁定。）当事务没有表达更新表中行的意图 时，就获取这种锁。（SELECT FOR UPDATE、UPDATE ... WHERE 和 INSERT 语句表达更新的意图。） |
| 下一键共享（Next Key Share，NS） | 行 | 锁拥有者和所有并发的事务都可以读（但不能更改）被锁定行中的数据。这种锁用来在使用读稳定性或游标稳定性事务隔离级别读取的数据上代替共享锁。 |
| 共享（Share，S） | 表和行 | 锁 拥有者和任何其他并发的事务都可以读（但不能更改）被锁定的表或行中的数据。只要表不是使用共享锁锁定的，那么该表中的单个行可以使用共享锁锁定。但是， 如果表是用共享锁定的，则锁拥有者不能在该表中获取行级的共享锁。如果表或行是用共享锁锁定的，则其他并发事务可以读取数据，但不能对它进行更改。 |
| 意向互斥（Intent Exclusive，IX） | 表空间和表 | 锁 拥有者和任何其他并发的应用程序都可以读取和更改被锁定表中的数据。当锁拥有者从表中读取数据时，它在所读取的每一行上获取一个共享锁，而在它更新的每一 行上获取更新锁和互斥锁。其他并发的应用程序可以读取和更新被锁定的表。当事务表达更新表中行的意图时，就获取这种锁。 |
| 带意向互斥的共享（Share With Intent Exclusive，SIX） | 表 | 锁拥有者可以读取和更改被锁定表中的数据。锁拥有者在它更新的行上获取互斥锁，但不在它读取的行上获取锁；因此，其他并发的应用程序可以读取但不能更新被锁定表中的数据。 |
| 更新（Update，U） | 表和行 | 锁的拥有者可以更新被锁定表中的数据，而且锁的拥有者在它所更新的任何行上自动获得互斥锁。其他并发的应用程序可以读取但不能更新被锁定表中的数据。 |
| 下一键弱互斥（Next Key Weak Exclusive，NW） | 行 | 锁的拥有者可以读取但不能更新被锁定的行。当在非编目表的索引中插入行时，在表中的下一行上获得这种锁。 |
| 互斥（Exclusive，X） | 表和行 | 锁的拥有者可以读取和更改被锁定的表或行中的数据。如果获取了互斥锁，则只允许使用未提交的读隔离级别的应用程序访问被锁定的表或行。对于用 INSERT、UPDATE 和／或 DELETE 语句操作的数据资源，将获取互斥锁。 |
| 弱互斥（Weak Exclusive，WE） | 行 | 锁的拥有者可以读取和更改被锁定的行。当向非编目表中插入行时，该行上将获得这种锁。 |
| 超级互斥（Super Exclusive，Z） | 表空间和表 | 锁的拥有者可以更改表、删除表、创建索引或删除索引。当事务尝试执行上述任何一种操作时，表上就自动获得这种锁。在释放这个锁之前，不允许其他并发事务读取或更新该表。 |

**如何获取锁**

在大多数情况下，DB2 数据库管理程序在需要锁时隐式地获取它们，因此这些锁在 DB2 数据库管理程序的控制之下。除了使用未提交读隔离级别的情况外，事务从不需要显式地请求锁。实际上，惟一有可能被事务显式地锁定的数据库对象是表。图 6 说明了用何种逻辑确定为所引用的对象获取什么类型的锁。

**图 6. 如何获取锁**

DB2 数据库管理程序总是尝试获取行级锁。但是，可以通过执行特殊形式的 ALTER TABLE 语句来修改这种行为，如下所示：

ALTER TABLE [TableName] LOCKSIZE TABLE

其中的 TableName 标识一个现有表的名称，所有事务在访问它时都要获取表级锁。

也可以通过执行 LOCK TABLE 语句，强制 DB2 数据库管理程序为特定事务在表上获取表级锁，如下所示：

LOCK TABLE [TableName] IN [SHARE | EXCLUSIVE] MODE 

其中的 TableName 标识一个现有表的名称，对于这个表应该获取表级锁（假定其他事务在该表上没有不兼容的锁）。如果在执行这个语句时指定了共享（SHARE）模式，就会获得 一个允许其他事务读取（但不能更改）存储在表中的数据的表级锁；如果执行时指定了互斥（EXCLUSIVE）模式，就会获得一个不允许其他事务读取或修改 存储在表中的数据的表级锁。

**锁兼容性**

如果数据资源上的一种锁状态允许在同一资源上放置另一个锁，就认为这两种锁（或两种状态）是兼容的 。 每当一个事务持有数据资源上的锁，而第二个事务请求同一资源上的锁时，DB2 数据库管理程序会检查两种锁状态以判断它们是否兼容。如果锁是兼容的，则将锁授予第二个事务（假定没有其他事务在等待该数据资源）。但是，如果锁不兼容， 则第二个事务必须等待，直到第一个事务释放它的锁为止，然后它才可以获取对资源的访问权并继续处理。（如果资源上有多个与新请求的锁不兼容的锁，则第二个 事务必须等到它们全部被释放为止。）请参阅 IBM DB2 9 Administration Guide: Performance 文档（或在 DB2 信息中心搜索 Lock type compatibility 主题）以获取关于各个锁之间是否兼容的特定信息。

**锁转换**

当事务尝试访问它已经持有锁的数据资源，但是所需的访问模式需要比已持有的锁更严格的锁时，则所持有的锁的状态更改成更严格的状态。将已经持有的锁的状态更改成更严格状态的操作称为锁转换 。发生锁转换是因为一个事务同一时间内只能在一个数据资源上持有一个锁。

在大多数情况下，对行级锁执行锁转换，转换过程相当简单。例如，如果持有共享（S）或更新（U）行级锁，但是需要 互斥（X）锁，则所持有的锁将被转换成互斥（X）锁。但意向互斥（IX）锁和共享（S）锁是特例，因为无法确定其中哪个更严格。因此，如果持有其中一种行 级锁但又需要另一种锁，则所持有的锁将转换成带意向互斥的共享（SIX）锁。假定所请求的锁状态更严格的话，则类似的转换会使所请求的锁状态成为持有锁的 新状态。（仅当所持有的锁可以增加其严格性时，才会发生锁转换。）在转换了锁状态之后，锁处于所获取的最高状态，直到持有该锁的事务终止为止。

**锁升级**

所有的锁都需要存储空间；因为可用空间并不是无限的，所以 DB2 数据库管理程序必须限制锁可以使用的空间（这是通过maxlocks 数据库配置参数完成的）。为了防止特定数据库代理超过已建立的锁空间限制，当获取的（任意类型的）锁过多时，会自动执行称为锁升级 的过程。锁升级是一种转换，它将同一表内几个单独的行级锁转换成一个单一的表级锁。因为锁升级是在内部处理的，所以惟一可从外部检测到的结果可能只是对一个和多个表的并发访问减少了。

以下是锁升级的工作原理：当事务请求锁而锁存储空间已满时，就选择与该事务相关联的一个表，让它获取一个表级锁， 释放该表的所有行级锁（从而在锁列表**[数据结构](http://lib.csdn.net/base/31" \t "_blank" \o "算法与数据结构知识库)**中让出空间），并将表级锁添加到锁列表。如果这个过程所释放的空间不够，则选择另一个表，重复这个过程，直到 释放了足够的可用空间为止。这时，事务将获取所请求的锁并继续执行。但是，如果在该事务的所有行级锁都已经升级之后，仍然没有获得必要的可用锁空间，则 （通过 SQL 错误编码）要求事务提交或回滚它启动以来所做的所有更改，然后事务终止。

**锁超时**

每当一个事务在特定数据资源（例如，表或行）上持有锁时，直到持有锁的事务终止并释放它所获取的所有锁之前，其他 事务对该资源的访问都可能被拒绝。如果没有某种锁超时检测机制，则事务可能无限期地等待锁的释放。例如，有可能出现这种情况：一个事务在等待另一个用户的 应用程序所持有的锁被释放，而该用户离开了他或她的工作站，但忘了执行一些允许应用程序终止拥有锁的事务的交互。显然，此类情况会导致极差的应用程序性 能。要避免发生此类情况时阻碍其他应用程序的执行，可以在数据库的配置文件中指定锁超时值（通过 locktimeout 数据库配置参数）。该参数控制任何事务等待获取所请求的锁的时间。如果在指定的时间间隔过去之后还未获得想要的锁，则等待的应用程序接收一个错误，并回滚请求该锁的事务。分布式事务应用程序环境尤其容易产生此类情况；可以通过使用锁超时避免它们。

**死锁**

尽管可以通过建立锁超时来避免一个事务无限期地等待另一个事务释放锁的情况，但是锁超时无法解决两个或更多事务对锁的争用。这种情况称为死锁 或死锁循环 。 说明死锁的发生原因的最佳方式是举例说明：假定事务 1 在表 A 上获取了互斥（X）锁，而事务 2 在表 B 上获取了互斥（X）锁。现在，假定事务 1 尝试在表 B 上获取互斥（X）锁，而事务 2 尝试在表 A 上获取互斥（X）锁。这两个事务的处理都将被挂起，直到同意第二个锁请求为止。但是，因为在任何一个事务释放它目前持有的锁（通过执行或回滚操作）之前， 这两个事务的锁请求都不会被同意，而且因为这两个事务都不能释放它目前持有的锁（因为它们都已挂起并等待锁），所以它们都陷入了死锁循环。图 7 说明了这个死锁场景。  
**图 7. 死锁循环**

当死锁循环发生时，除非某些外部代理进行干涉，否则所涉及的所有事务将无限期地等待释放锁。在 DB2 UDB 中，用于处理死锁的代理是称为死锁检测器 的异步系统后台进程。死锁检测器的惟一职责是定位和解决在锁定子系统中找到的任何死锁。每个数据库有自己的死锁检测器，它在数据库初始化过程中激活。激活 之后，死锁检测器在大多数时间处于 “休眠” 状态，但会以预置的时间间隔被 “唤醒”，以确定锁定子系统中是否存在死锁循环。如果死锁检测器在锁定子系统中发现死锁，则随机选择死锁涉及的一个事务，终止并回滚它。选择的事务收到一 个 SQL 错误编码，它所获得的所有锁都被释放；这样，剩下的事务就可以继续执行了，因为死锁循环已经被打破了。

**锁粒度**

正如先前提到的，每当一个事务在特定数据资源上持有锁时，在持有锁的事务终止之前，其他事务对该资源的访问都可能 被拒绝。因此，为了进行优化以获取最大的并发性，行级锁通常比表级锁更好，因为它们所限制访问的资源要小得多。但是，因为所获取的每个锁都需要一定数量的 处理时间和存储空间，才能获取锁并进行管理，所以单个表级锁需要的开销比几个单独的行级锁少。除非另外指定，否则默认情况下获取行级锁。

可以通过使用 ALTER TABLE ... LOCKSIZE TABLE、ALTER TABLE ... LOCKSIZE ROW 和 LOCK TABLE 语句控制锁的粒度 （即， 获取行级锁还是表级锁）。ALTER TABLE ... LOCKSIZE TABLE 语句提供了确定粒度的全局方法，它使得所有访问特定表中行的事务都获取表级锁。另一方面，LOCK TABLE 语句允许在单个事务级别获取表级锁。在使用这两种语句时，事务在需要锁时就获取单个共享（S）或互斥（X）表级锁。结果，因为只需获取和释放一个表级锁， 而不是多个不同的行级锁，所以锁定性能通常会提高。但是，在使用表级锁时，如果长时间运行的事务获取互斥而不是共享表级锁，那么并发性会降低。

**事务和锁定**

从锁定的角度来看，所有事务通常归为以下几类之一：

**只读**：这是指只读性的事务，它们包含 SELECT 语句（它们本质上就是只读的）、指定了 FOR READ ONLY 子句的 SELECT 语句或意义虽不明确但因为在预编译和／或绑定过程中指定了 BLOCKING 选项而看作是只读的 SQL 语句。  
**倾向于更改**：这是指有可能进行更改的事务，它们包含指定了 FOR UPDATE 子句的 SELECT 语句或者那些意义虽不明确但因为 SQL 预编译器解释它的方式而看作是倾向于进行更改的 SQL 语句。  
**更改**：这是指一定会进行更改的事务，它们包含 INSERT、UPDATE 和／或 DELETE 语句，但不包括 UPDATE ... WHERE CURRENT OF ... 或 DELETE ... WHERE CURRENT OF ... 语句。  
游标控制：这是指包含 UPDATE ... WHERE CURRENT OF ... 和 DELETE ... WHERE CURRENT OF ... 语句的事务。

只读事务通常使用意向共享（IS）和／或共享（S）锁。另一方面，倾向于更改的事务将更新（U）、意向互斥 （IX）和互斥（X）锁用于表，将共享（S）、更新（U）和互斥（X）锁用于行。更改事务往往使用意向互斥（IX）和／或互斥（X）锁，而游标控制的事务 通常使用意向互斥（IX）和／或互斥（X）锁。

当 SQL 语句准备执行时，DB2 优化器研究各种满足该语句请求的方法，并估计每种方法所涉及的执行成本。然后，DB2 优化器根据这一评估选择它认为最优的访问计划（访问计划指定满足 SQL 请求所需的操作，以及执行这些操作的顺序）。访问计划可以使用两种方法之一来访问表中的数据：通过直接地读取表（称为执行表 或关系扫描 ）；或通过读取该表上的索引，然后检索特定索引项所引用的表行（称为执行索引扫描 ）。

DB2 优化器选择的访问路径（通常是根据数据库的设计确定的）会对所获取锁的数目和所使用的锁状态产生显著的影响。例如，当使用索引扫描来查找特定行时，DB2 数据库管理程序极有可能获取一个或多个意向共享（IS）行级锁。但是，如果使用表扫描，因为必须依次扫描整个表来找到特定行，所以 DB2 数据库管理程序可能会选择获取单个共享（S）表级锁。

**DB2隔离级别**

要维护数据库的一致性和数据完整性，同时又允许多个应用程序同时访问一个数据库，将这样的特性称为并发性。 DB2 数据库尝试强制实施并发性的方法之一是使用隔离级别，它决定在第一个事务访问数据时，如何对其他事务锁定或隔离该事务所使用的数据。 DB2 使用下列隔离级别来强制实施并发性：

* 可重复读 (Reapeatable Read，RR)
* 读稳定性 (Read Stability，RS)
* 游标稳定性 (Cursor Stability，CS)
* 未提交的读 (Uncommitted Read，UR)

隔离级别是根据称为现象 (Phenomena) 的三个禁止操作序列来声明的：

* 脏读 (Dirty Read)：在事务 A 提交修改结果之前，其他事务即可看到事务Ａ的修改结果。
* 不可重复读 (Non-Repeatable Read)：在事务Ａ提交之前，允许其他事务修改和删除事务Ａ涉及的数据，导致事务Ａ中执行同样操作的结果集变小。
* 幻像读 (Phantom Read)：事务Ａ在提交查询结果之前，其他事务可以插入或者更改事务 A 涉及的数据，导致事务 A 中执行同样操作的结果集增大。 数据库并发性 ( 可以同时访问同一资源的事务数量 ) 因隔离级别不同而有所差异，可重复读隔离级别可以防止所有现象，但是会大大降低并发性。未提交读隔离级别提供了最大的并发性，但可能会造成“脏读”、“幻像读”或“不可重复读”现象。 DB2 默认的隔离级别是 CS 。

**WebSphere中数据源的缺省隔离级别为 RS**

**RR**

可重复读隔离级别是最严格的隔离级别。在使用它时，一个事务的操作结果完全与其他并发事务隔离，脏读、不可重复读、幻像读都不会发生。当使用可重复读隔离级别时，在事务执行期间会共享 (S) 锁定该事务以任何方式引用的所有行，在该事务中多次执行同一条 SELECT 语句，得到的结果数据集总是相同的。因此，使用可重复读隔离级别的事务可以多次检索同一行集，并可以对它们执行任意操作，直到提交或回滚操作终止事务。但是，在事务提交前，不允许其他事务执行会影响该事务正在访问的任何行的插入、更新或删除操作。为了确保这种行为，需要锁定该事务所引用的每一行—— 而不是仅锁定被实际检索或修改的那些行。因此，如果一个表中有 1000 行，但只检索两行，则整个表 (1000 行，而不仅是被检索的两行 ) 都会被锁定。

如果使用这种隔离级别，不管你从表中读多少数据，整个表上都加 S 锁，直到该事务被提交或回滚，表上的锁才会被释放。这样可以保证在一个事务中即使多次读取同一行，都会得到相同结果集。另外，在同一事务中如果以同样的搜索标准重新打开已被处理过的游标，那么得到的结果集不会改变。可重复读相对于读稳定性而言，加锁的范围更大：对于读稳定性，应用程序只对符合要求的所有行加锁；而对于重复读，应用程序将对整个表都加 S 锁。 可重复读会锁定应用程序在工作单元中引用的整个表。利用可重复读，一个应用程序在打开游标的相同工作单元内发出一个 SELECT 语句两次，每次都返回相同的结果。利用可重复读隔离级别，不可能出现丢失更新、脏读和幻像读的情况。 在该工作单元完成之前，“可重复读”应用程序可以多次检索和操作这些行。但是，在该工作单元完成之前其他应用程序均不能更新、删除或插入可能会影响结果表的行。**“可重复读”应用程序不能查看其他应用程序未提交的更改。**

**RS**

读稳定性隔离级别没有可重复读隔离级别那么严格；因此，它没有将事务与其他并发事务的效果完全隔离。读稳定性隔离级别可以防止脏读和不可重复读，但是可能出现幻像读。在使用这个隔离级别时，只锁定事务实际检索和修改的行。因此，如果一个表中有1000行，但只检索两行(通过索引扫描)，则只锁定被检索的两行(而不是所扫描的1000行)。因此，如果在同一个事务中发出同一个SELECT语句两次或更多次，那么每次产生的结果数据集可能不同。

与可重复读隔离级别一样，在读稳定性隔离级别下运行的事务可以检索一个行集(ROWSSET)，并可以对它们执行任意操作，直到事务终止。在这个事务存在期间，其他事务不能执行那些会影响这个事务检索到的行集的更新或删除操作，但是可以执行插入操作。如果插入的行与第一个事务的查询的选择条件匹配，那么这些行可能作为幻像出现在后续产生的结果数据集中。**其他事务对其他行所作的更改，在提交之前是不可见的**。

如果使用这种隔离级，那么在一个事务中将有N+1个锁，其中N是所有被读取(通过索引扫描)过的行的数目，这些行上都会被加上NS锁，在表上加上1个IS锁。这些锁直到该事务被提交或回滚才会被释放。这样可以保证在一个事务中即使多次读取同一行，得到的值也不会改变。但是使用这种隔离级别，

在一个事务中，如果使用同样的搜索标准重新打开已被处理过的游标，则结果集可能改变(可能会增加某些行，这些行被称为幻影行(Phantom))。这是因为RS隔离级别不能阻止通过插入或更新操作在结果集中加入新行。注意：

NS是下一键共享锁，此时锁拥有者和所有并发的事务都可以读(但不能更改)被锁定行中的数据。这种锁用来在使用读稳定性或游标稳定性事务隔离级别读取的数据上代替共享锁。

读稳定性(RS)只锁定应用程序在工作单元中检索的那些行。它确保在某个工作单元完成之前，在该工作单元运行期间的任何限定行读取不被其他应用程序进程更改，且确保不会读取由另一个应用程序进程所更改的任何行，直至该进程提交了这些更改。也就是说，不可能出现“不可重复读”情形。

“读稳定性”隔离级别的其中一个目标是提供较高并行性以及数据的稳定视图，为了有助于达到此目标，优化器确保在发生锁定升级前不获取表级锁定。

“读稳定性”隔离级别最适用于包括下列所有特征的应用程序：

* 在并发环境下运行。
* 需要限定某些行在工作单元运行期间保持稳定。
* 在工作单元中不会多次发出相同的查询，或者在同一工作单元中发出多次查询时并不要求该查询获得相同的回答。

**CS**

游标稳定性隔离级别在隔离事务效果方面非常宽松。它可以防止脏读；但有可能出现不可重复读和幻像读。这是因为在大多数情况下，游标稳定性隔离级别只锁定事务声明并打开的游标当前引用的行。 当使用游标稳定性隔离级别的事务通过游标从表中检索行时，其他事务不能更新或删除游标所引用的行。但是，如果被锁定的行本身不是用索引访问的，那么其他事务可以将新的行添加到表中，以及对被游标锁定行前后的行进行更新或删除操作。所获取的锁一直有效，直到游标重定位或事务终止为止 ( 如果游标重定位，原来行上的锁就被释放，并获得游标现在引用的行上的锁 ) 。此外，如果事务修改了它检索到的任何行，那么在事务终止之前，其他事务不能更新或删除该行，即使在游标不再位于被修改的行时。与可重复读和读稳定性隔离级别一样，其他事务在其他行上进行的更改，在这些更改提交之前对于使用游标稳定性隔离级别的事务 ( 这是默认的隔离级别 ) 是不可见的。我们还用上面那个例子，一个表中有 1000 行数据，我们只检索其中两行数据。那么对于可重复读隔离级别会锁住整个表，对于读稳定性隔离级别会对读到的数据 ( 两行 ) 加锁，而对于游标稳定性隔离级别只对游标当前所在那一行加锁，游标所在行的前一行和下一行都不加锁。

如果使用这种隔离级，那么在一个事务中只有两个锁：结果集中只有正在被读取的那一行 ( 游标指向的行 ) 被加上 NS 锁，在表上加 IS 锁。其他未被处理的行上不加锁。这种隔离级别只能保证正在处理的行的值不会被其他并发的程序所改变。该隔离级别是 DB2 默认的隔离级别。 游标稳定性 (CS) 当在行上定位游标时会锁定任何由应用程序的事务所访问的行。此锁定在读取下一行或终止事务之前有效。但是，如果更改了某一行上的任何数据，那么在对数据库提交更改之前必须挂起该锁定。 对于具有“游标稳定性”的应用程序已检索的行，当该行上有任何可更新的游标时，任何其他应用程序都不能更新或删除该行。“游标稳定性”应用程序不能查看其他应用程序的未提交操作。 使用“游标稳定性”，可能会出现不可重复读和幻像读现象。“游标稳定性”是默认隔离级别，应在需要最大并行性，但只看到其他应用程序中的已提交行的情况下才使用。

**UR**

未提交读隔离级别是最不严格的隔离级别。实际上，在使用这个隔离级别时，仅当另一个事务试图删除或更改被检索的行所在的表时，才会锁定一个事务检索的行。因为在使用这种隔离级别时，行通常保持未锁定状态，所以脏读、不可重复读和幻像读都可能会发生。因此，未提交读隔离级别通常用于那些访问只读表和视图的事务，以及某些执行?SELECT?语句的事务(只要其他事务的未提交数据对这些语句没有负面效果)。

顾名思义，其他事务对行所做的更改在提交之前对于使用未提交读隔离级别的事务是可见的。但是，此类事务不能看见或访问其他事务DDL(CREATE、ALTER和DROP)语句所创建的表、视图或索引，直到那些事务被提交为止。类似地，如果其他事务删除了现有的表、视图或索引，那么仅当进行删除操作的事务终止时，使用未提交读隔离级别的事务才能知道这些对象不再存在了。

一定要注意一点：当运行在未提交读隔离级别下的事务使用可更新游标时，该事务的行为和在游标稳定性隔离级别下运行一样，并应用游标稳定性隔离级别的约束。

*1、ur(Uncommited Read)：就是俗称“脏读“，在没有提交数据的时候能够读到更新的数据；即select数据时，无需等待正在对数据进行的update操作。*

*2、cs(Cursor Stability)：在使用游标稳定性隔离级别时，一个事务的结果集中只有正在被读取的那一行(游标指向的行)会被加上NS锁，其他未被处理的行上不被加锁。这种隔离级别只能保证正在被处理的行的值不会被其他并发的程序所改变。该隔离级别是DB2默认的隔离级别。可以避免脏读的发生，但无法避免不可重复读和幻读的情况发生。*

*3、rs(Read Stability)：在一个事务中进行查询时，不允许读取其他事务update结果集中的数据，允许读取到其他事务提交的新增数据。会锁定结果集中的所有行，会解决不可重复读问题，但无法解决幻读的问题。*

*4、rr(Repeatable Read)：在一个事务中进行查询时，不允许任何对这个查询表的数据修改。对该表的操作彻底为序列化的形式，效率最低，可避免任何问题。*

**如何获取锁 ?**

从图 6-1 中我们可以看到，数据库首先判断该 SQL 语句是采用全表扫描还是索引扫描。如果是全表扫描，那么会在整个表上加表级别的锁；如果是读操作，那么获取表级 S 锁；如果是 DML(INSERT、UPDATE 和 DELETE) 操作，那么获取表级 X 锁。假设 SQL 语句采用的是索引扫描，如果是读操作，在读取的行上加 NS 锁，同时在表上加 IS 锁；如果是 DML 操作，那么在操作的行上加 X 锁，同时在表上加 IX 锁。

**读锁和写锁?**

在 DB2 数据库中有两种主要类型的锁：读锁 (S) 和写 (X) 锁。 一般来说读锁是在如下情况下加的： NS 是在 RS 和 CS 隔离级别下对读取到的行加的锁。而 S 锁是在 RR 隔离级别下对读取到的表 ( 使用全表扫描 ) 或行 ( 使用索引扫描 ) 加的锁。 U 锁是在“ select \* from t1 for update ”情况下加的锁。这些锁都是在读取 (SELECT) 期间加的锁。

一般来说写锁是在如下情况下加的： Z 锁是超级排它锁，它不允许任何隔离级别的读取，一般是在数据物理结构发生改变的情况下加的锁。例如：CREATE、ALTER、DROP、离线 REORG 和离线 LOAD 期间会加 Z 锁。 X 锁是在做 INSERT、UPDATE 和 DELETE 期间加的锁，它允许使用 UR 隔离级别进行未提交读取。 NW 锁表示当一行被插入到索引中的时候，该行的下一行会被加上该锁。锁的拥有者可以读但不能更改锁定行。该锁与 X 锁类似，只是与 NS 锁兼容。

# [MySQL的并发控制与加锁分析](https://www.cnblogs.com/yelbosh/p/5813865.html)

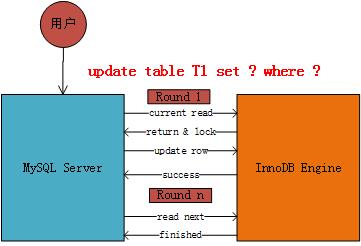
**1.1 MVCC：Multi-Version Concurrent Control 多版本并发控制**

　　MVCC是为了实现数据库的并发控制而设计的一种协议。从我们的直观理解上来看，要实现数据库的并发访问控制，最简单的做法就是**加锁访问**，即读的时候不能写（允许多个西线程同时读，即**共享锁**，S锁），写的时候不能读（一次最多只能有一个线程对同一份数据进行写操作，即**排它锁**，X锁）。这样的加锁访问，其实并不算是真正的并发，或者说它只能实现并发的读，因为它最终实现的是**读写串行化**，这样就大大降低了数据库的读写性能。加锁访问其实就是和MVCC相对的LBCC，即基于锁的并发控制（Lock-Based Concurrent Control），是四种隔离级别中级别最高的Serialize隔离级别。为了提出比LBCC更优越的并发性能方法，MVCC便应运而生。

**几乎所有的RDBMS都支持MVCC**。它的最大好处便是，**读不加锁，读写不冲突**。在MVCC中，读操作可以分成两类，快照读（Snapshot read）和当前读（current read）。快照读，读取的是记录的可见版本（可能是历史版本，即最新的数据可能正在被当前执行的事务并发修改），不会对返回的记录加锁；而当前读，读取的是记录的最新版本，并且会对返回的记录加锁，保证其他事务不会并发修改这条记录。在MySQL InnoDB中，简单的select操作，如 select \* from table where ? 都属于快照读；属于当前读的包含以下操作：

1. select \* from table where ? lock in share mode; （加S锁）
2. select \* from table where ? for update; （加X锁，下同）
3. insert, update, delete操作

 　　针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，**加锁也是一条一条进行的**。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后再读取下一条加锁，直至读取完毕。需要注意的是，以上需要加X锁的都是当前读，而普通的select（除了for update）都是快照读，每次insert、update、delete之前都是会进行一次当前读的，这个时候会上锁，防止其他事务对某些行数据的修改，从而造成数据的不一致性。我们广义上说的幻读现象是通过MVCC解决的，意思是通过MVCC的快照读可以使得事务返回相同的数据集。如下图所示：



注意，我们一般说在MyISAM中使用表锁，因为MyISAM在修改数据记录的时候会将整个表锁起来；而InnoDB使用的是行锁，即我们以上所谈的MVCC的加锁问题。但是，并不是InnoDB引擎不会使用表锁，比如在alter table的时候，Innodb就会将该表用表锁锁起来。

**1.2 隔离级别**

　　在SQL的标准中，定义了四种隔离级别。每一种级别都规定了，在一个事务中所做的修改，哪些在事务内和事务间是可见的，哪些是不可见的。低级别的隔离可以执行更高级别的并发，性能好，但是会出现脏读和幻读的现象。首先，我们从两个基础的概念说起：

**脏读（dirty read）**：两个事务，一个事务读取到了另一个事务未提交的数据，这便是脏读。

**幻读（phantom read）**：两个事务，事务A与事务B，事务A在自己执行的过程中，执行了两次相同查询，第一次查询事务B未提交，第二次查询事务B已提交，从而造成两次查询结果不一样，这个其实被称为**不可重复读**；如果事务B是一个会影响查询结果的insert操作，则好像新多出来的行像幻觉一样，因此被称为幻读。**其他事务的提交会影响在同一个事务中的重复查询结果。**

　　下面简单描述一下SQL中定义的四种标准隔离级别：

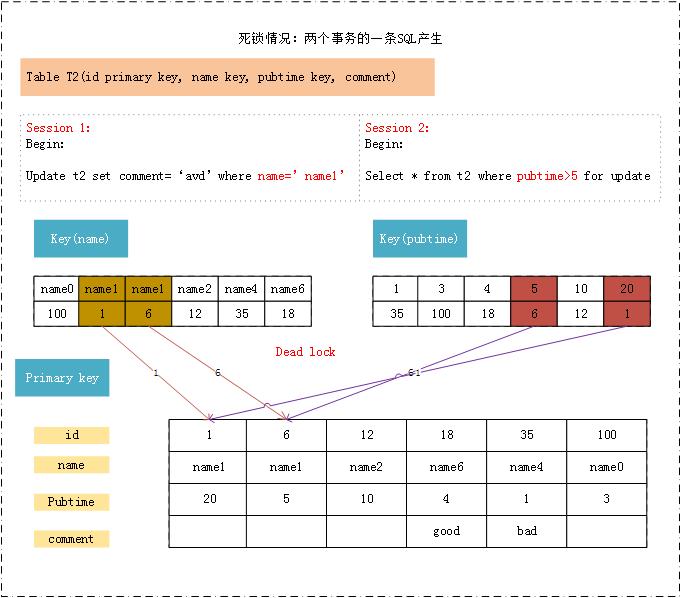
1. **READ UNCOMMITTED (未提交读)** ：隔离级别：0. 可以读取未提交的记录。会出现脏读。
2. **READ COMMITTED (提交读)** ：隔离级别：1. 事务中只能看到已提交的修改。不可重复读，会出现幻读。（在InnoDB中，会加行所，但是不会加间隙锁）该隔离级别是大多数数据库系统的默认隔离级别，但是MySQL的则是RR。
3. **REPEATABLE READ (可重复读)** ：隔离级别：2. **在InnoDB中是这样的**：RR隔离级别保证对读取到的记录加锁 (记录锁)，同时保证对读取的范围加锁，新的满足查询条件的记录不能够插入 (间隙锁)，因此不存在幻读现象。但是标准的RR只能保证在同一事务中多次读取同样记录的结果是一致的，而无法解决幻读问题。InnoDB的幻读解决是依靠MVCC的实现机制做到的。
4. **SERIALIZABLE （可串行化）**：隔离级别：3. 该隔离级别会在读取的每一行数据上都加上锁，退化为基于锁的并发控制，即LBCC。

 　　需要注意的是，MVCC只在RC和RR两个隔离级别下工作，其他两个隔离级别都和MVCC不兼容。

**1.3 死锁**

　　死锁是指两个或者多个事务在同一资源上相互作用，并请求锁定对方占用的资源，从而导致恶性循环的现象。**当多个事务试图以不同的顺序锁定资源时**，就可能产生死锁。多个事务同时锁定同一个资源时，也会产生死锁。且看下面的两个产生死锁的例子：





 　　第一个死锁很好理解，而第二个死锁，由于在主索引（聚簇索引表）上仍旧是对两条记录进行了不同顺序的加锁，因此仍旧会造成死锁。死锁的发生与否，并不在于事务中有多少条SQL语句，死锁的关键在于：**两个(或以上)的Session加锁的顺序不一致**。因此，我们通过分析加锁细节，可以判断所写的sql是否会发生死锁，同时发生死锁的时候，我们应该如何处理。

**1.4 InnoDB的MVCC实现机制**

　　MVCC可以认为是行级锁的一个变种，它可以在很多情况下避免加锁操作，因此开销更低。MVCC的实现大都都实现了非阻塞的读操作，写操作也只锁定必要的行。InnoDB的MVCC实现，是通过保存数据在某个时间点的快照来实现的。**一个事务，不管其执行多长时间，其内部看到的数据是一致的**。也就是事务在执行的过程中不会相互影响。下面我们简述一下MVCC在InnoDB中的实现。

　　InnoDB的MVCC，**通过在每行记录后面保存两个隐藏的列来实现：一个保存了行的创建时间，一个保存行的过期时间（删除时间），当然，这里的时间并不是时间戳，而是系统版本号，每开始一个新的事务，系统版本号就会递增**。在RR隔离级别下，MVCC的操作如下：

1. select操作。a. **InnoDB只查找版本早于（包含等于）当前事务版本的数据行**。可以确保事务读取的行，要么是事务开始前就已存在，或者事务自身插入或修改的记录。b. **行的删除版本要么未定义，要么大于当前事务版本号**。可以确保事务读取的行，在事务开始之前未删除。
2. insert操作。将新插入的行保存当前版本号为行版本号。
3. delete操作。将删除的行保存当前版本号为删除标识。
4. update操作。变为insert和delete操作的组合，insert的行保存当前版本号为行版本号，delete则保存当前版本号到原来的行作为删除标识。

　　由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是**将删除版本号小于当前系统版本的行删除**，这个过程叫做purge。

**1.5 一个简单SQL的加锁分析**

　　在MySQL的InnoDB中，都是基于聚簇索引表的。而且普通的select操作都是基于快照读，是不需要加锁的。那么我们在分析其他的sql语句的时候，如何分析加锁细节？下面我们以一个简单的delete操作的SQL为例，进行一个详细的阐述。且看下面的SQL：

　　delete from t1 where id=10;

　　如果对这条SQL进行加锁分析，那么MySQL是如何加锁的呢？一般情况下，我们直观的感受是：**会在id=10的记录上加锁**。但是，这样轻率的下结论是片面的，要想确定MySQL的加锁情况，我们还需要知道更多的条件。还需要知道哪些条件呢？比如：

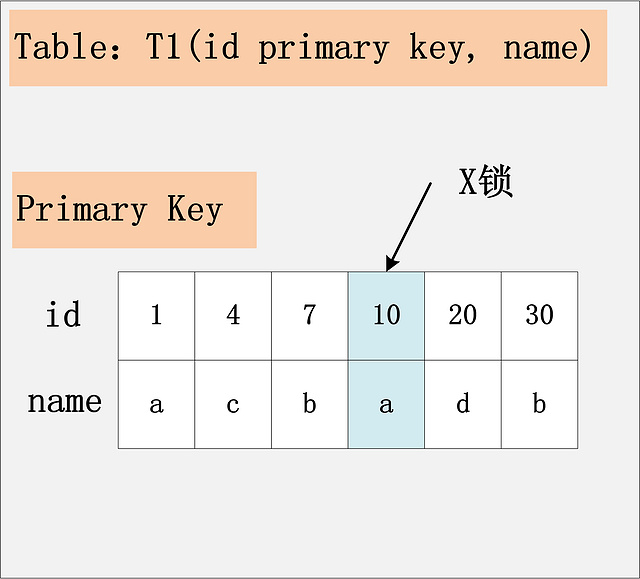
1. id列是不是主键？
2. 系统的隔离级别是什么？
3. id非主键的话，其上有建立索引吗？
4. 建立的索引是唯一索引吗？
5. 该SQL的执行计划是什么？索引扫描？全表扫描？

　　接下来，我将这些问题的答案进行组合，然后按照从易到难的顺序，逐个分析每种组合下，对应的SQL会加哪些锁。

* **组合1**：id列是主键，RC隔离级别
* **组合2**：id列是二级唯一索引，RC隔离级别
* **组合3**：id列是二级非唯一索引，RC隔离级别
* **组合4**：id列上没有索引，RC隔离级别
* **组合5**：id列是主键，RR隔离级别
* **组合6**：id列是二级唯一索引，RR隔离级别
* **组合7**：id列是二级非唯一索引，RR隔离级别
* **组合8**：id列上没有索引，RR隔离级别
* **组合9**：Serializable隔离级别

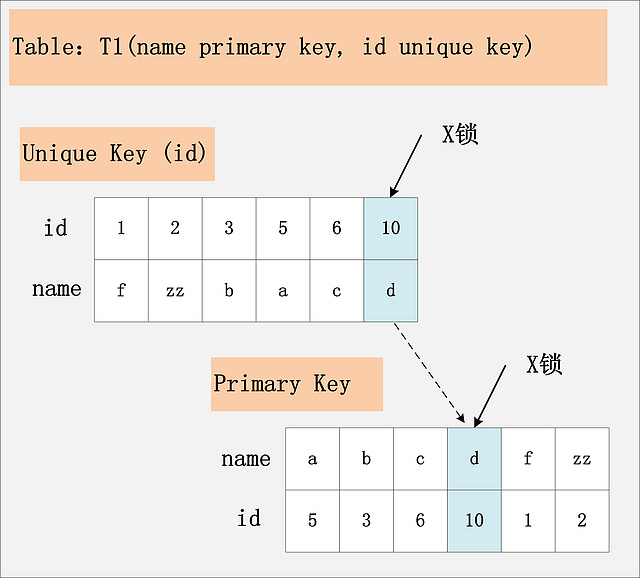
　　组合1：id列是主键，RC隔离级别

　　当id是主键的时候，我们只需要在该id=10的记录上加上x锁即可。如下图所示：



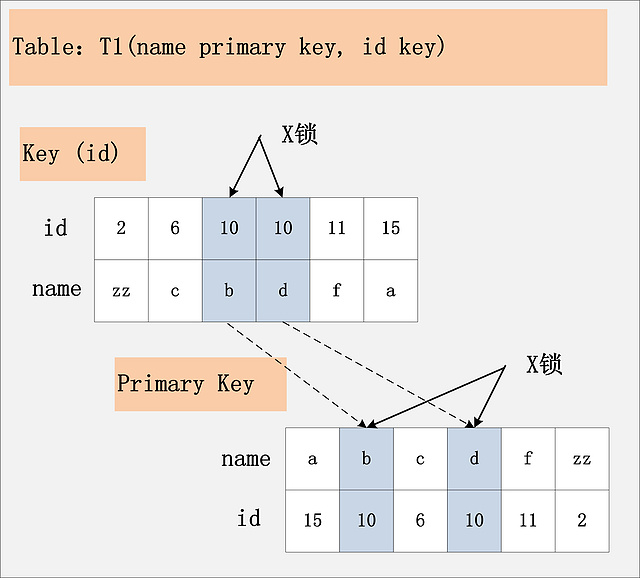
　　组合2：id列是二级唯一索引，RC隔离级别

　　在这里我先解释一下聚簇索引和普通索引的区别。在InnoDB中，主键可以被理解为聚簇索引，聚簇索引中的叶子结点就是相应的数据行，具有聚簇索引的表也被称为聚簇索引表，数据在存储的时候，是按照主键进行排序存储的。我们都知道，数据库在select的时候，会选择索引列进行查找，索引列都是按照B+树（多叉搜索树）数据结构进行存储，找到主键之后，再回到聚簇索引表中进行查询，这叫回表查询。那我们自然会问，**当使用索引进行查询的时候，与索引相对应的记录会被上锁吗？会的**。如果id是唯一索引，那么只给该唯一索引所对应的索引记录上x锁；如果id是非唯一索引，那么所对应的所有的索引记录上都会上x锁。如下图所示：



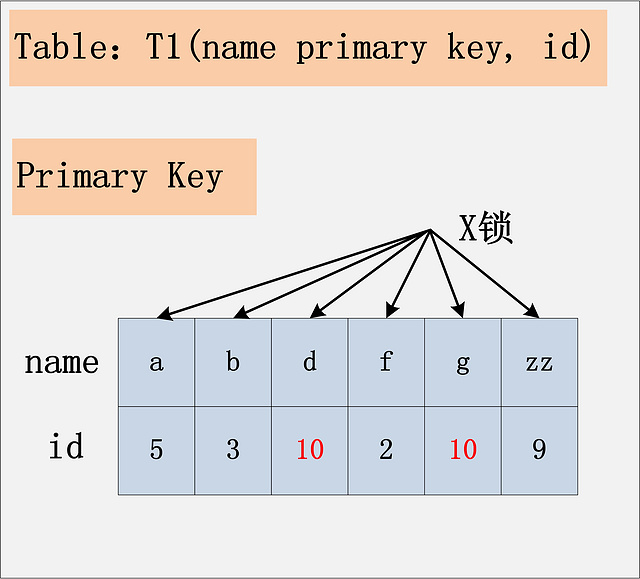
　　组合3：id列是二级非唯一索引，RC隔离级别

　　解释同上，如下图：



　　组合4：id列上没有索引，RC隔离级别

 　  由于id列上没有索引，因此只能走聚簇索引，进行全部扫描。有人说会在表上加X锁；有人说会在聚簇索引上，选择出来的id = 10 的记录加上X锁。真实情况如下图：

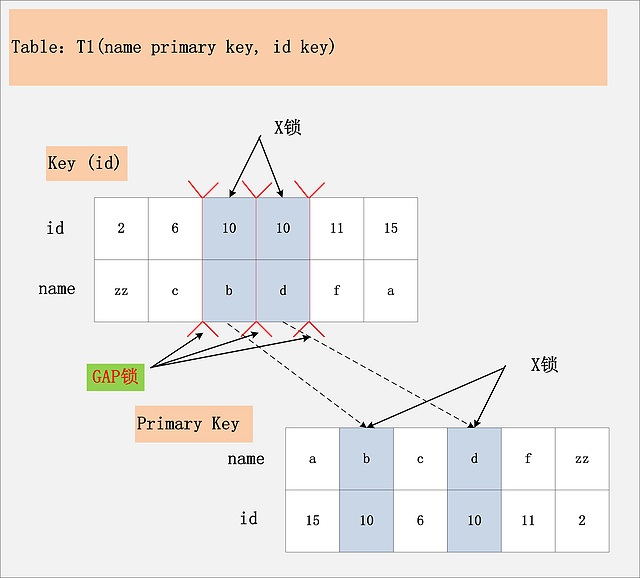


　　若id列上没有索引，SQL会走聚簇索引的全扫描进行过滤，由于过滤是由MySQL Server层面进行的。因此每条记录，无论是否满足条件，都会被加上X锁。但是，为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了2PL的约束（同时加锁同时放锁）。

　　组合5，6同以上（因为只有一条结果记录，只能在上面加锁）

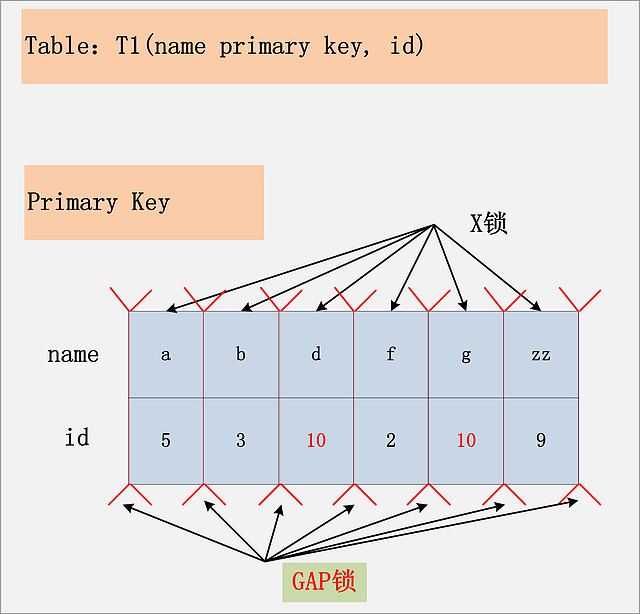
　　组合7：id列是二级非唯一索引，RR隔离级别

 　　在RR隔离级别下，为了防止幻读的发生，会使用Gap锁。这里，你可以把Gap锁理解为，不允许在数据记录前面插入数据。首先，通过id索引定位到第一条满足查询条件的记录，加记录上的X锁，加GAP上的GAP锁，然后加主键聚簇索引上的记录X锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录X锁，但是仍旧需要加GAP锁，最后返回结束。如下图所示：



　　组合8：id列无索引，RR隔离级别

　　在这种情况下，聚簇索引上的所有记录，都被加上了X锁。其次，聚簇索引每条记录间的间隙(GAP)，也同时被加上了GAP锁。如下图：



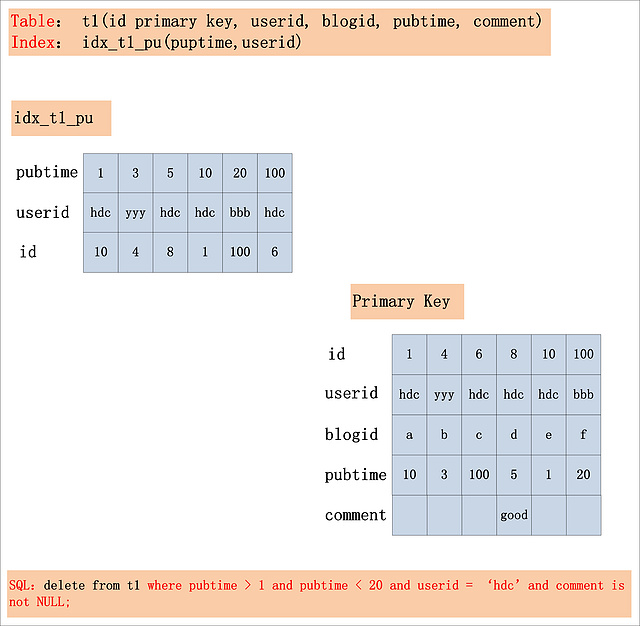
　　但是，MySQL是做了相关的优化的，就是所谓的semi-consistent read。semi-consistent read开启的情况下，对于不满足查询条件的记录，MySQL会提前放锁，同时也不会添加Gap锁。

　　组合9：Serializable隔离级别

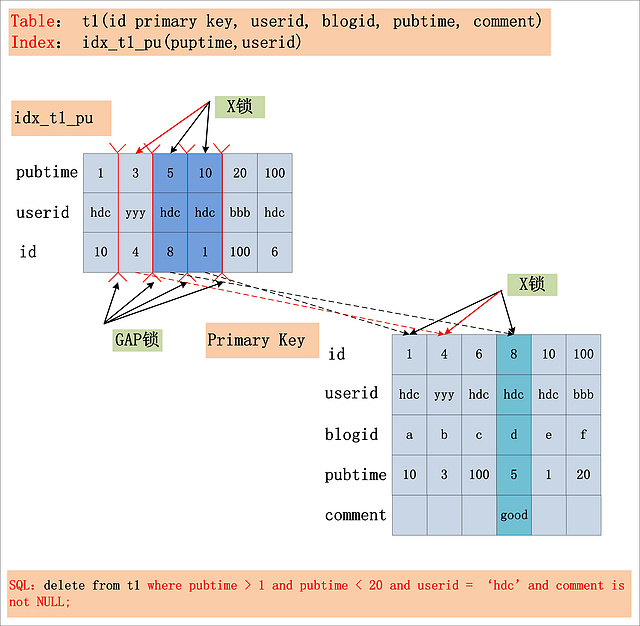
　　和RR隔离级别一样。

**1.6 一个复杂的SQL的加锁分析**

 　这里我们只是列出一个结论，因为要涉及到MySQL的where查询条件的分析，因此这里先不做详细介绍，我会在之后的博客中详细说明。如下图：



　　结论：在RR隔离级别下，针对一个复杂的SQL，首先需要提取其where条件。Index Key确定的范围，**需要加上GAP锁**；Index Filter过滤条件，视MySQL版本是否支持ICP，若支持ICP，则不满足Index Filter的记录，不加X锁，否则需要X锁；Table Filter过滤条件，无论是否满足，都需要加X锁。加锁的结果如下所示：



**总结**

本文只是对MVCC的一些基础性的知识点进行了详细的总结，参考了网上和书上比较多的资料和实例。希望能对各位的学习有所帮助。

# 乐观锁和悲观锁的实例讲解

悲观锁：正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）的修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

以常用的mysql InnoDB存储引擎为例：加入商品表items表中有一个字段status，status=1表示该商品未被下单，status=2表示该商品已经被下单，那么我们对每个商品下单前必须确保此商品的status=1。假设有一件商品，其id为10000；如果不使用锁，那么操作方法如下：

//查出商品状态   
select status from items where id=10000;   
//根据商品信息生成订单   
insert into orders(id,item\_id) values(null,10000);   
//修改商品状态为2   
update Items set status=2 where id=10000;

上述场景在高并发环境下可能出现问题：   
前面已经提到只有商品的status=1是才能对它进行下单操作，上面第一步操作中，查询出来的商品status为1。但是当我们执行第三步update操作的时候，有可能出现其他人先一步对商品下单把Item的status修改为2了，但是我们并不知道数据已经被修改了，这样就可能造成同一个商品被下单2次，使得数据不一致。所以说这种方式是不安全的。   
使用悲观锁来实现：在上面的场景中，商品信息从查询出来到修改，中间有一个处理订单的过程，使用悲观锁的原理就是，当我们在查询出items信息后就把当前的数据锁定，直到我们修改完毕后再解锁。那么在这个过程中，因为items被锁定了，就不会出现有第三者来对其进行修改了。   
注：要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。我们可以使用命令设置MySQL为非autocommit模式：

set autocommit=0;

设置完autocommit后，我们就可以执行我们的正常业务了。具体如下：

//开始事务

begin;/begin work;/start transaction; (三者选一就可以)

//查询出商品信息

select status from items where id=10000 for update;

//根据商品信息生成订单

insert into orders (id,item\_id) values (null,10000);

//修改商品status为2

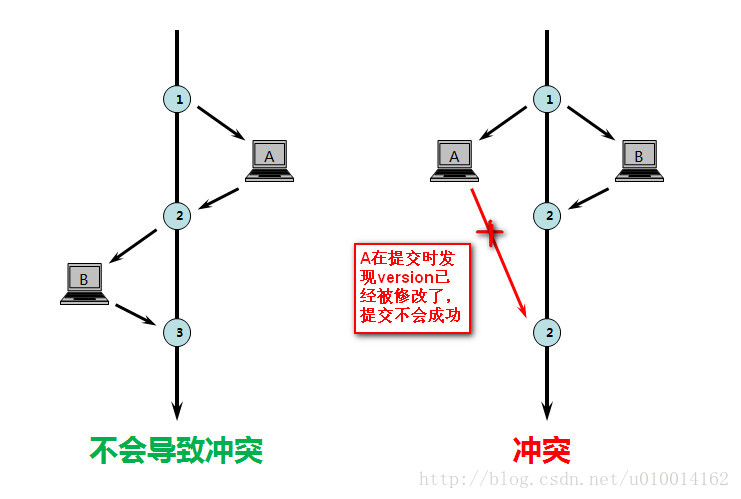
update items set status=2 where id=10000;

//提交事务

commit;/commit work;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

注：上面的begin/commit为事务的开始和结束，因为在前一步我们关闭了mysql的autocommit，所以需要手动控制事务的提交，在这里就不细表了。   
上面的第一步我们执行了一次查询操作：select status from items where id=10000 for update;与普通查询不一样的是，我们使用了select…for update的方式，这样就通过数据库实现了悲观锁。此时在items表中，id为10000的 那条数据就被我们锁定了，其它的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。   
注：需要注意的是，在事务中，只有SELECT … FOR UPDATE 或LOCK IN SHARE MODE 同一笔数据时会等待其它事务结束后才执行，一般SELECT … 则不受此影响。拿上面的实例来说，当我执行select status from items where id=10000 for update;后。我在另外的事务中如果再次执行select status from items where id=10000 for update;则第二个事务会一直等待第一个事务的提交，此时第二个查询处于阻塞的状态，但是如果我是在第二个事务中执行select status from items where id=10000;则能正常查询出数据，不会受第一个事务的影响。   
上面我们提到，使用select…for update会把数据给锁住，不过我们需要注意一些锁的级别，MySQL InnoDB默认Row-Level Lock，所以只有明确地指定主键，MySQL 才会执行Row lock (只锁住被选取的数据) ，否则MySQL 将会执行Table Lock (将整个数据表单给锁住)。除了主键外，使用索引也会影响数据库的锁定级别。   
悲观锁并不是适用于任何场景，它也有它存在的一些不足，因为悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。如果加锁的时间过长，其他用户长时间无法访问，影响了程序的并发访问性，同时这样对数据库性能开销影响也很大，特别是对长事务而言，这样的开销往往无法承受。所以与悲观锁相对的，我们有了乐观锁，乐观锁的概念如下：   
- 乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。那么我们如何实现乐观锁呢，一般来说有以下2种方式：   
1.使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值+1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。用下面的一张图来说明：



如上图所示，如果更新操作顺序执行，则数据的版本（version）依次递增，不会产生冲突。但是如果发生有不同的业务操作对同一版本的数据进行修改，那么，先提交的操作（图中B）会把数据version更新为2，当A在B之后提交更新时发现数据的version已经被修改了，那么A的更新操作会失败。   
2.乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。   
以mysql InnoDB存储引擎为例，还是拿之前的例子商品表items表中有一个字段status，status=1表示该商品未被下单，status=2表示该商品已经被下单，那么我们对每个商品下单前必须确保此商品的status=1。假设有一件商品，其id为10000；   
下单操作包括3步骤：

//查询出商品信息

select (status,version) from items where id=#{id}

//根据商品信息生成订单

//修改商品status为2

update items set status=2,version=version+1 where id=#{id} and version=#{version};

* 1
* 2
* 3
* 4
* 5
* 6

为了使用乐观锁，我们需要首先修改items表，增加一个version字段，数据默认version可设为1；   
其实我们周围的很多产品都有乐观锁的使用，比如我们经常使用的分布式存储引擎Tair，Tair中存储的每个数据都有版本号，版本号在每次更新后都会递增，相应的，在Tair put接口中也有此version参数，这个参数是为了解决并发更新同一个数据而设置的，这其实就是乐观锁；   
很多情况下，更新数据是先get，修改get回来的数据，然后put回系统。如果有多个客户端get到同一份数据，都对其修改并保存，那么先保存的修改就会被后到达的修改覆盖，从而导致数据一致性问题,在大部分情况下应用能够接受，但在少量特殊情况下，这个是我们不希望发生的。   
比如系统中有一个值”1”, 现在A和B客户端同时都取到了这个值。之后A和B客户端都想改动这个值，假设A要改成12，B要改成13，如果不加控制的话，无论A和B谁先更新成功，它的更新都会被后到的更新覆盖。Tair引入的乐观锁机制避免了这样的问题。刚刚的例子中，假设A和B同时取到数据，当时版本号是10，A先更新，更新成功后，值为12，版本为11。当B更新的时候，由于其基于的版本号是10，此时服务器会拒绝更新，返回version error，从而避免A的更新被覆盖。B可以选择get新版本的value，然后在其基础上修改，也可以选择强行更新。   
当然了，乐观锁也是要精心挑选的，主要的目的就是避免锁的失败率过高又要规避ABA问题。关于锁力度太大导致接口操作失败率过高。   
商品库存扣减时，尤其是在秒杀、聚划算这种高并发的场景下，若采用version号作为乐观锁，则每次只有一个事务能更新成功，业务感知上就是大量操作失败。   
若挑选以库存数作为乐观锁

update item   
set   
quantity=quantity-#sub\_quantity#   
where   
item\_id = #id#   
and quantity-#sub\_quantity# > 0

通过挑选乐观锁，可以减小锁力度，从而提升吞吐～   
乐观锁需要灵活运用,现在互联网高并发的架构中，受到fail-fast思路的影响，悲观锁已经非常少见了。