InnoDB锁问题  
InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁。行级锁与表级锁本来就有许多不同之处，另外，事务的引入也带来了一些新问题。下面我们先介绍一点背景知识，然后详细讨论InnoDB的锁问题。  
背景知识  
1．事务（Transaction）及其ACID属性  
事务是由一组SQ语句组成的逻辑处理单元，事务具有以下4个属性，通常简称为事务的ACID属性。  
     原子性（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。  
      一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。  
     隔离性（Isoation）：[**数据库**](http://lib.csdn.net/base/14)系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。  
     持久性（Durabe）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。  
银行转帐就是事务的一个典型例子。  
2．并发事务处理带来的问题  
相对于串行处理来说，并发事务处理能大大增加数据库资源的利用率，提高数据库系统的事务吞吐量，从而可以支持更多的用户。但并发事务处理也会带来一些问题，主要包括以下几种情况。  
     更新丢失（ost Update）：当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题－－最后的更新覆盖了由其他事务所做的更新。例如，两个编辑人员制作了同一文档的电子副本。每个编辑人员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改副本的编辑人员覆盖另一个编辑人员所做的更改。如果在一个编辑人员完成并提交事务之前，另一个编辑人员不能访问同一文件，则可避免此问题。  
     脏读（Dirty Reads）：一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做"脏读"。  
     不可重复读（Non-Repeatabe Reads）：一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变、或某些记录已经被删除了！这种现象就叫做“不可重复读”。  
     幻读（Phantom Reads）：一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”。  
3．事务隔离级别  
在上面讲到的并发事务处理带来的问题中，“更新丢失”通常是应该完全避免的。但防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据加必要的锁来解决，因此，防止更新丢失应该是应用的责任。  
“脏读”、“不可重复读”和“幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。数据库实现事务隔离的方式，基本上可分为以下两种。  
  一种是在读取数据前，对其加锁，阻止其他事务对数据进行修改。  
  另一种是不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot)，并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好象是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据多版本并发控制（MutiVersion Concurrency Contro，简称MVCC或MCC），也经常称为多版本数据库。  
数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上 “串行化”进行，这显然与“并发”是矛盾的。同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。  
为了解决“隔离”与“并发”的矛盾，ISO/ANSI SQ92定义了4个事务隔离级别，每个级别的隔离程度不同，允许出现的副作用也不同，应用可以根据自己的业务逻辑要求，通过选择不同的隔离级别来平衡 “隔离”与“并发”的矛盾。表20-5很好地概括了这4个隔离级别的特性。  
  
 最后要说明的是：各具体数据库并不一定完全实现了上述4个隔离级别，例如，Oracle只提供Read committed和Serializable两个标准隔离级别，另外还提供自己定义的Read only隔离级别；SQL Server除支持上述ISO/ANSI SQL92定义的4个隔离级别外，还支持一个叫做“快照”的隔离级别，但严格来说它是一个用MVCC实现的Serializable隔离级别。[**MySQL**](http://lib.csdn.net/base/14) 支持全部4个隔离级别，但在具体实现时，有一些特点，比如在一些隔离级别下是采用MVCC一致性读，但某些情况下又不是，这些内容在后面的章节中将会做进 一步介绍。

获取InnoDB行锁争用情况

可以通过检查InnoDB\_row\_lock状态变量来分析系统上的行锁的争夺情况：

1. mysql> show status like 'innodb\_row\_lock%';
2. +-------------------------------+-------+
3. | Variable\_name                 | Value |
4. +-------------------------------+-------+
5. | InnoDB\_row\_lock\_current\_waits | 0     |
6. | InnoDB\_row\_lock\_time          | 0     |
7. | InnoDB\_row\_lock\_time\_avg      | 0     |
8. | InnoDB\_row\_lock\_time\_max      | 0     |
9. | InnoDB\_row\_lock\_waits         | 0     |
10. +-------------------------------+-------+
11. 5 rows in set (0.01 sec)

如果发现锁争用比较严重，如InnoDB\_row\_lock\_waits和InnoDB\_row\_lock\_time\_avg的值比较高，还可以通过设置InnoDB Monitors来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。具体方法如下：

1. mysql> CREATE TABLE innodb\_monitor(a INT) ENGINE=INNODB;
2. Query OK, 0 rows affected (0.14 sec)

 然后就可以用下面的语句来进行查看：

1. mysql> Show innodb status\G;
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. Type: InnoDB
4. Name:
5. Status:

 监视器可以通过发出下列语句来停止查看：

1. mysql> DROP TABLE innodb\_monitor;
2. Query OK, 0 rows affected (0.05 sec)

 设置监视器后，在SHOW INNODB STATUS的显示内容中，会有详细的当前锁等待的信息，包括表名、锁类型、锁定记录的情况等，便于进行进一步的分析和问题的确定。打开监视器以后，默认情况下每15秒会向日志中记录监控的内容，如果长时间打开会导致.err文件变得非常的巨大，所以用户在确认问题原因之后，要记得删除监控表以关闭监视器，或者通过使用“--console”选项来启动服务器以关闭写日志文件。  
InnoDB的行锁模式及加锁方法  
InnoDB实现了以下两种类型的行锁。  
  共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。  
  排他锁（X)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。  
另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。  
  意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。  
  意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。  
上述锁模式的兼容情况具体如表20-6所示。  
  
 如果一个事务请求的锁模式与当前的锁兼容，InnoDB就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。  
意向锁是InnoDB自动加的，不需用户干预。对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；对于普通SELECT语句，InnoDB不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。  
  共享锁（S）：SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE。  
  排他锁（X)：SELECT \* FROM table\_name WHERE ... FOR UPDATE。  
用SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用SELECT... FOR UPDATE方式获得排他锁。  
在如表20-7所示的例子中，使用了SELECT ... IN SHARE MODE加锁后再更新记录，看看会出现什么情况，其中actor表的actor\_id字段为主键。  
  
  
  
当使用SELECT...FOR UPDATE加锁后再更新记录，出现如表20-8所示的情况。

InnoDB行锁实现方式  
InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！  
在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。下面通过一些实际例子来加以说明。  
（1）在不通过索引条件查询的时候，InnoDB确实使用的是表锁，而不是行锁。在如表20-9所示的例子中，开始tab\_no\_index表没有索引：

1. mysql> create table tab\_no\_index(id int,name varchar(10)) engine=innodb;
2. Query OK, 0 rows affected (0.15 sec)
3. mysql> insert into tab\_no\_index values(1,'1'),(2,'2'),(3,'3'),(4,'4');
4. Query OK, 4 rows affected (0.00 sec)
5. Records: 4  Duplicates: 0  Warnings: 0

 表20-9               InnoDB存储引擎的表在不使用索引时使用表锁例子

在如表20 -9所示的例子中，看起来session\_1只给一行加了排他锁，但session\_2在请求其他行的排他锁时，却出现了锁等待！原因就是在没有索引的情况下，InnoDB只能使用表锁。当我们给其增加一个索引后，InnoDB就只锁定了符合条件的行，如表20-10所示。  
创建tab\_with\_index表，id字段有普通索引：

1. mysql> create table tab\_with\_index(id int,name varchar(10)) engine=innodb;
2. Query OK, 0 rows affected (0.15 sec)
3. mysql> alter table tab\_with\_index add index id(id);
4. Query OK, 4 rows affected (0.24 sec)
5. Records: 4  Duplicates: 0  Warnings: 0

（2）由于MySQL的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。应用设 计的时候要注意这一点。在如表20-11所示的例子中，表tab\_with\_index的id字段有索引，name字段没有索引

1. mysql> alter table tab\_with\_index drop index name;
2. Query OK, 4 rows affected (0.22 sec)
3. Records: 4  Duplicates: 0  Warnings: 0
4. mysql> insert into tab\_with\_index  values(1,'4');
5. Query OK, 1 row affected (0.00 sec)
6. mysql> select \* from tab\_with\_index where id = 1;
7. +------+------+
8. | id   | name |
9. +------+------+
10. | 1    | 1    |
11. | 1    | 4    |
12. +------+------+
13. 2 rows in set (0.00 sec)

 （3）当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，另外，不论是使用主键索引、唯一索引或普通索引，InnoDB都会使用行锁来对数据加锁。  
在如表20-12所示的例子中，表tab\_with\_index的id字段有主键索引，name字段有普通索引：

1. mysql> alter table tab\_with\_index add index name(name);
2. Query OK, 5 rows affected (0.23 sec)
3. Records: 5  Duplicates: 0  Warnings: 0

 表20-12                                  InnoDB存储引擎的表使用不同索引的阻塞例子  
  
  
 （4）即便在条件中使用了索引字段，但是否使用索引来检索数据是由MySQL通过判断不同执行计划的代价来决定的，如果MySQL认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。关于MySQL在什么情况下不使用索引的详细讨论，参见本章“索引问题”一节的介绍。  
在下面的例子中，检索值的数据类型与索引字段不同，虽然MySQL能够进行数据类型转换，但却不会使用索引，从而导致InnoDB使用表锁。通过用explain检查两条SQL的执行计划，我们可以清楚地看到了这一点。  
例子中tab\_with\_index表的name字段有索引，但是name字段是varchar类型的，如果where条件中不是和varchar类型进行比较，则会对name进行类型转换，而执行的全表扫描。

1. mysql> alter table tab\_no\_index add index name(name);
2. Query OK, 4 rows affected (8.06 sec)
3. Records: 4  Duplicates: 0  Warnings: 0
4. mysql> explain select \* from tab\_with\_index where name = 1 \G
5. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
6. id: 1
7. select\_type: SIMPLE
8. table: tab\_with\_index
9. type: ALL
10. possible\_keys: name
11. key: NULL
12. key\_len: NULL
13. ref: NULL
14. rows: 4
15. Extra: Using where
16. 1 row in set (0.00 sec)
17. mysql> explain select \* from tab\_with\_index where name = '1' \G
18. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
19. id: 1
20. select\_type: SIMPLE
21. table: tab\_with\_index
22. type: ref
23. possible\_keys: name
24. key: name
25. key\_len: 23
26. ref: const
27. rows: 1
28. Extra: Using where
29. 1 row in set (0.00 sec)

 间隙锁（Next-Key锁）

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的 索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁 （Next-Key锁）。

举例来说，假如emp表中只有101条记录，其empid的值分别是 1,2,...,100,101，下面的SQL：

Select \* from  emp where empid > 100 for update;

是一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。

InnoDB使用间隙锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使 用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另外一方面，是为了满足其恢复和复制的需 要。有关其恢复和复制对锁机制的影响，以及不同隔离级别下InnoDB使用间隙锁的情况，在后续的章节中会做进一步介绍。

很显然，在使用范围条件检索并锁定记录时，InnoDB这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

还要特别说明的是，InnoDB除了通过范围条件加锁时使用间隙锁外，如果使用相等条件请求给一个不存在的记录加锁，InnoDB也会使用间隙锁！

在如表20-13所示的例子中，假如emp表中只有101条记录，其empid的值分别是1,2,......,100,101。

恢复和复制的需要，对InnoDB锁机制的影响

MySQL通过BINLOG录执行成功的INSERT、UPDATE、DELETE等更新数据的SQL语句， 并由此实现MySQL数据库的恢复和主从复制（可以参见本书“管理篇”的介绍）。MySQL的恢复机制（复制其实就是在Slave Mysql不断做基于BINLOG的恢复）有以下特点。

l  一是MySQL的恢复是SQL语句级的，也就是重新执行BINLOG中的SQL语句。这与Oracle数据库不同，Oracle是基于数据库文件块的。

l  二是MySQL的Binlog是按照事务提交的先后顺序记录的，恢复也是按这个顺序进行的。这点也与Oralce不同，Oracle是按照系统更新号 （System Change Number，SCN）来恢复数据的，每个事务开始时，Oracle都会分配一个全局唯一的SCN，SCN的顺序与事务开始的时间顺序是一致的。

从上面两点可知，MySQL的恢复 机制要求：在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读，这已经超过了ISO/ANSI SQL92“可重复读”隔离级别的要求，实际上是要求事务要串行化。这也是许多情况下，InnoDB要用到间隙锁的原因，比如在用范围条件更新记录时，无 论在Read Commited或是Repeatable Read隔离级别下，InnoDB都要使用间隙锁，但这并不是隔离级别要求的，有关InnoDB在不同隔离级别下加锁的差异在下一小节还会介绍。

另外，对于“insert  into target\_tab select \* from source\_tab where ...”和“create  table new\_tab ...select ... From  source\_tab where ...(CTAS)”这种SQL语句，用户并没有对source\_tab做任何更新操作，但MySQL对这种SQL语句做了特别处理。先来看如表20-14的例子。  
  
  
 在上面的例子中，只是简单地读 source\_tab表的数据，相当于执行一个普通的SELECT语句，用一致性读就可以了。ORACLE正是这么做的，它通过MVCC技术实现的多版本 数据来实现一致性读，不需要给source\_tab加任何锁。我们知道InnoDB也实现了多版本数据，对普通的SELECT一致性读，也不需要加任何 锁；但这里InnoDB却给source\_tab加了共享锁，并没有使用多版本数据一致性读技术！