锁机制详解

锁是计算机协调多个进程或线程并发访问某一资源的机制。

在数据库中，除了传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供需要用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。

锁分类

- 从性能上分为**乐观锁**(用版本对比或CAS机制)和**悲观锁**，乐观锁适合读操作较多的场景，悲观锁适合写操作较多的场景，如果在写操作较多的场景使用乐观锁会导致比对次数过多，影响性能
- 从对数据操作的粒度分，分为**表锁、页锁、行锁**
- 从对数据库操作的类型分，分为**读锁和写锁**(都属于悲观锁)，还有**意向锁**

读锁（共享锁，S锁(**S**hared)）：针对同一份数据，多个读操作可以同时进行而不会互相影响，比如：

```
select * from T where id=1 lock in share mode
```

写锁（排它锁，X锁(e**X**clusive)）：当前写操作没有完成前，它会阻断其他写锁和读锁，数据修改操作都会加写锁，查询也可以通过for update加写锁，比如：

```
select * from T where id=1 for update
```

意向锁（Intention Lock）：又称**I锁**，针对**表锁**，主要是为了提高加表锁的效率，是mysql数据库自己加的。当有事务给表的数据行加了共享锁或排他锁，同时会给表设置一个标识，代表已经有行锁了，其他事务要想对表加表锁时，就不必逐行判断有没有行锁可能跟表锁冲突了，直接读这个标识就可以确定自己该不该加表锁。特别是表中的记录很多时，逐行判断加表锁的方式效率很低。而这个标识就是意向锁。

意向锁主要分为：

意向共享锁，IS锁，对整个表加共享锁之前，需要先获取到意向共享锁。

意向排他锁，IX锁，对整个表加排他锁之前，需要先获取到意向排他锁。

表锁

每次操作锁住整张表。**开销小，加锁快**；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低；一般用在整表数据迁移的场景。

基本操作

```
--建表SQL
CREATE TABLE `mylock` (
 `id` INT (11) NOT NULL AUTO_INCREMENT,
 `NAME` VARCHAR (20) DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE = MyISAM DEFAULT CHARSET = utf8;

--插入数据
INSERT INTO`test`.`mylock` (`id`, `NAME`) VALUES ('1', 'a');
 INSERT INTO`test`.`mylock` (`id`, `NAME`) VALUES ('2', 'b');
 INSERT INTO`test`.`mylock` (`id`, `NAME`) VALUES ('3', 'c');
 INSERT INTO`test`.`mylock` (`id`, `NAME`) VALUES ('4', 'd');
```

```
--手动增加表锁
lock table 表名称 read(write),表名称2 read(write);
--查看表上加过的锁
show open tables;
--删除表锁
```

```
6 unlock tables;
```

页锁

只有**BDB存储引擎支持页锁**，页锁就是在页的粒度上进行锁定，锁定的数据资源比行锁要多，因为一个页中可以有多个行记录。当我们使用页锁的时候，会出现数据浪费的现象，但这样的浪费最多也就是一个页上的数据行。页锁的开销介于表锁和行锁之间，会出现死锁。锁定粒度介于表锁和行锁之间，并发度一般。

行锁

每次操作锁住一行数据。**开销大，加锁慢**；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度最高。

InnoDB相对于MYISAM的最大不同有两点：

- **InnoDB支持事务（TRANSACTION）**
- **InnoDB支持行级锁**

注意，InnoDB的行锁实际上是针对索引加的锁(在索引对应的索引项上做标记)，不是针对整个行记录加的锁。并且该索引不能失效，否则会从行锁升级为表锁。(**RR级别会升级为表锁，RC级别不会升级为表锁**)

比如我们在RR级别执行如下sql

```
1 select * from account where name = 'lilei' for update; --where条件里的name字段无索引
```

则其它Session对该表任意一行记录做修改操作都会被阻塞住。

PS：关于RR级别行锁升级为表锁的原因分析

因为在RR隔离级别下，需要解决不可重复读和幻读问题，所以在遍历扫描聚集索引记录时，为了防止扫描过的索引被其它事务修改(不可重复读问题) 或 间隙被其它事务插入记录(幻读问题)，从而导致数据不一致，所以MySQL的解决方案就是把所有扫描过的索引记录和间隙都锁上，这里要注意，并不是直接将整张表加表锁，因为不一定能加上表锁，可能会有其它事务锁住了表里的其它行记录。

间隙锁(Gap Lock)

间隙锁，锁的就是两个值之间的空隙，**间隙锁是在可重复读隔离级别下才会生效。**

上节课讲过，Mysql默认级别是repeatable-read，有幻读问题，间隙锁是可以解决幻读问题的。

假设account表里数据如下：

id	name	balance
1	lilei	0
2	hanmei	10000
3	lucy	80000
10	zhuge	666
20	zhangsan	2000

那么间隙就有 id 为 (3,10)，(10,20)，(20,正无穷) 这三个区间，在Session_1下面执行如下sql：

```
1 select * from account where id = 18 for update;
```

则其他Session没法在这个(10,20)这个间隙范围里插入任何数据。

如果执行下面这条sql：

```
1 select * from account where id = 25 for update;
```

则其他Session没法在这个(20,正无穷)这个间隙范围里插入任何数据。

也就是说，只要在间隙范围内锁了一条不存在的记录会锁住整个间隙范围，不锁边界记录，这样就能防止其它Session在这个间隙范围内插入数据，就解决了可重复读隔离级别的幻读问题。

临键锁(Next-key Locks)

Next-Key Locks是行锁与间隙锁的组合。

总结：

MyISAM在执行查询语句SELECT前，会自动给涉及的所有表加读锁，在执行update、insert、delete操作会自动给涉及的表加写锁。
InnoDB在执行查询语句SELECT时(非串行隔离级别)，不会加锁。但是update、insert、delete操作会加行锁。
另外，读锁会阻塞写，但是不会阻塞读。而写锁则会把读和写都阻塞。

Innodb存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一下，但是在整体并发处理能力方面要远远优于MYISAM的表级锁定的。当系统并发量高的时候，Innodb的整体性能和MYISAM相比就会有比较明显的优势了。
但是，Innodb的行级锁定同样也有其脆弱的一面，当我们使用不当的时候，可能会让Innodb的整体性能表现不仅不能比MYISAM高，甚至可能会更差。

锁等待分析

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

```
show status like 'innodb_row_lock%';

对各个状态量的说明如下：
Innodb_row_lock_current_waits: 当前正在等待锁定的数量
Innodb_row_lock_time: 从系统启动到现在锁定总时间长度
Innodb_row_lock_time_avg: 每次等待所花平均时间
Innodb_row_lock_time_max：从系统启动到现在等待最长的一次所花时间
Innodb_row_lock_waits: 系统启动后到现在总共等待的次数

对于这5个状态变量，比较重要的主要是：
Innodb_row_lock_time_avg （等待平均时长）
Innodb_row_lock_waits （等待总次数）
Innodb_row_lock_time（等待总时长）
```

尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手制定优化计划。

查看INFORMATION_SCHEMA系统库锁相关数据表

```
-- 查看事务
select * from INFORMATION_SCHEMA.INNODB_TRX;
-- 查看锁，8.0之后需要换成这张表performance_schema.data_locks
select * from INFORMATION_SCHEMA.INNODB_LOCKS;
-- 查看锁等待，8.0之后需要换成这张表performance_schema.data_lock_waits
select * from INFORMATION_SCHEMA.INNODB_LOCK_WAITS;

-- 释放锁，trx_mysql_thread_id可以从INNODB_TRX表里查看到
kill trx_mysql_thread_id
```

```
10
11 -- 查看锁等待详细信息
12 show engine innodb status;
```

死锁问题分析

```
1 set tx_isolation='repeatable-read';
2 Session_1执行：select * from account where id=1 for update;
3 Session_2执行：select * from account where id=2 for update;
4 Session_1执行：select * from account where id=2 for update;
5 Session_2执行：select * from account where id=1 for update;
6 查看近期死锁日志信息：show engine innodb status;
```

大多数情况mysql可以自动检测死锁并回滚产生死锁的那个事务，但是有些情况mysql没法自动检测死锁，这种情况我们可以通过日志分析找到对应事务线程id，可以通过kill杀掉。

锁优化实践

- 尽可能让所有数据检索都通过索引来完成，避免无索引行锁升级为表锁
- 合理设计索引，尽量缩小锁的范围
- 尽可能减少检索条件范围，避免间隙锁
- 尽量控制事务大小，减少锁定资源量和时间长度，涉及事务加锁的sql尽量放在事务最后执行
- 尽可能用低的事务隔离级别

MVCC多版本并发控制机制

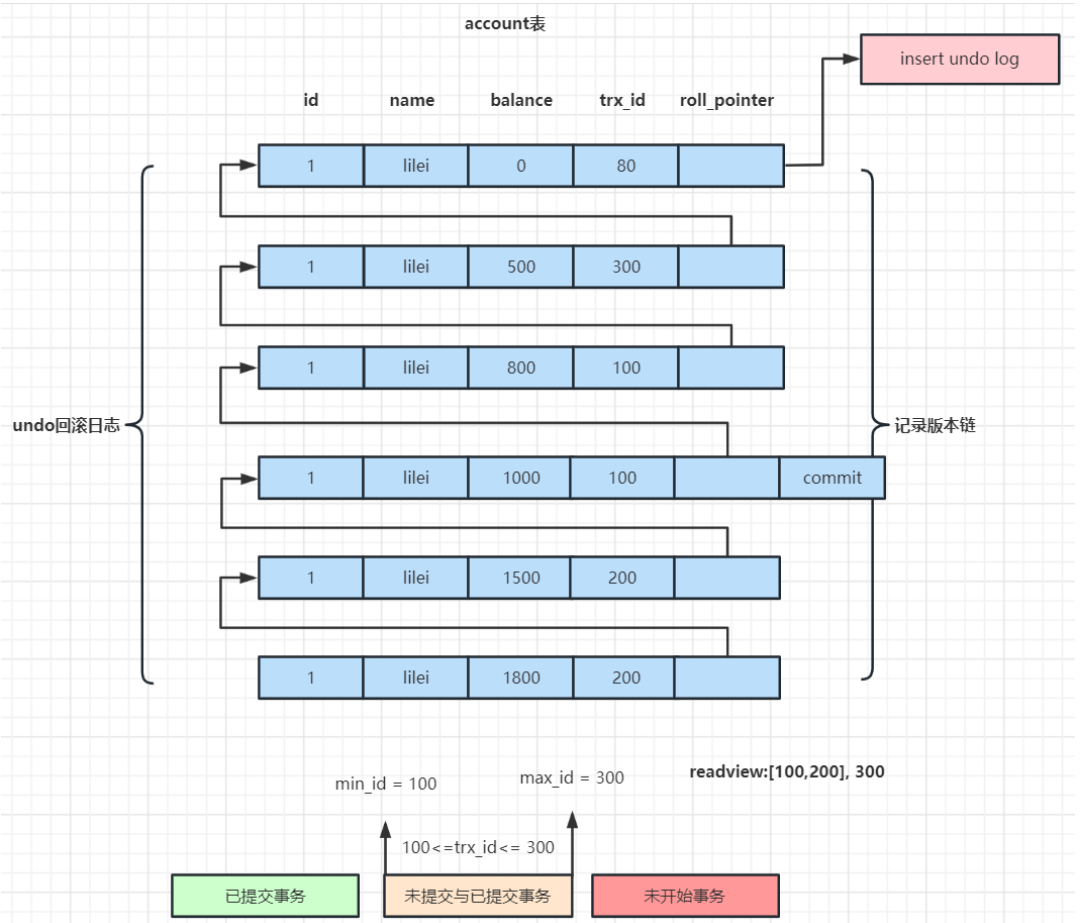
Mysql在可重复读隔离级别下如何保证事务较高的隔离性，我们上节课给大家演示过，同样的sql查询语句在一个事务里多次执行查询结果相同，就算其它事务对数据有修改也不会影响当前事务sql语句的查询结果。

这个隔离性就是靠MVCC(**Multi-Version Concurrency Control**)机制来保证的，对一行数据的读和写两个操作默认是不会通过加锁互斥来保证隔离性，避免了频繁加锁互斥，而在串行化隔离级别为了保证较高的隔离性是通过将所有操作加锁互斥来实现的。

Mysql在读已提交和可重复读隔离级别下都实现了MVCC机制。

undo日志版本链与read view机制详解

undo日志版本链是指一行数据被多个事务依次修改过后，在每个事务修改完后，Mysql会保留修改前的数据undo回滚日志，并且用两个隐藏字段trx_id和roll_pointer把这些undo日志串联起来形成一个历史记录版本链(见下图，需参考视频里的例子理解)



在**可重复读隔离级别**，当事务开启，执行任何查询sql时会生成当前事务的**一致性视图read-view，**该视图在事务结束之前永远都不会变化(**如果是读已提交隔离级别在每次执行查询sql时都会重新生成read-view**)，这个视图由执行查询时所有未提交事务id数组（数组里最小的id为min_id）和已创建的最大事务id（max_id）组成，事务里的任何sql查询结果需要从对应版本链里的最新数据开始逐条跟read-view做比对从而得到最终的快照结果。

版本链比对规则：

1. 如果 row 的 trx_id 落在绿色部分(trx_id<min_id)，表示这个版本是已提交的事务生成的，这个数据是**可见**的；
2. 如果 row 的 trx_id 落在红色部分(trx_id>max_id)，表示这个版本是由将来启动的事务生成的，是**不可见**的(若 row 的 trx_id 就是当前自己的事务是可见的）；
3. 如果 row 的 trx_id 落在黄色部分(min_id <=trx_id<= max_id)，那就包括两种情况

   a. 若 row 的 trx_id 在**视图数组**中，表示这个版本是由还没提交的事务生成的，**不可见**(若 row 的 trx_id 就是当前自己的事务是可见的)；

   b. 若 row 的 trx_id 不在**视图数组**中，表示这个版本是已经提交了的事务生成的，**可见。**

课上演示MVCC可见性算法的操作示例，大家可以对照着示例理解：

RR隔离级别MVCC可见性算法示例						RC隔离级别MVCC可见性算法示例
#事务 100	#事务 200	#事务 300	# select 1	# select 2	balance	# select 3
begin;	begin;	begin;	begin;	begin;	0	
update test set age = '18' where id = 1;						
	update test set age = '20' where id = 6;					
		update account set balance = balance+500 where id =1;				
		commit;			500	
			select balance from account where id = 1; --readview:[100,200] 300 结果是500			select balance from account where id = 1; --readview:[100,200] 300 结果是500
update account set balance = balance+300 where id =1;						
update account set balance = balance+200 where id =1;						
commit;					1000	
			select balance from account where id = 1; --readview:[100,200] 300 结果是500	select balance from account where id = 1; --readview:[200] 300 结果是1000		select balance from account where id = 1; --readview:[200] 300 结果是1000
	update account set balance = balance+500 where id =1;					
	update account set balance = balance+300 where id =1;					
	commit;				1800	
			select balance from account where id = 1; --readview:[100,200] 300 结果是500 select balance from account where id = 2; --readview:[100,200] 300	select balance from account where id = 1; --readview:[200] 300 结果是1000		

对于删除的情况可以认为是update的特殊情况，会将版本链上最新的数据复制一份，然后将trx_id修改成删除操作的trx_id，同时在该条记录的头信息（record header）里的（deleted_flag）标记位写上true，来表示当前记录已经被删除，在查询时按照上面的规则查到对应的记录如果delete_flag标记位为true，意味着记录已被删除，则不返回数据。

关于readview和可见性算法的原理解释

readview和可见性算法其实就是记录了sql查询那个时刻数据库里提交和未提交所有事务的状态。

要实现RR隔离级别，事务里每次执行查询操作readview都是使用第一次查询时生成的readview，也就是都是以第一次查询时当时数据库里所有事务提交状态来比对数据是否可见，当然可以实现每次查询的可重复读的效果了。

要实现RC隔离级别，事务里每次执行查询操作readview都会按照数据库当前状态重新生成readview，也就是每次查询都是跟数据库里当前所有事务提交状态来比对数据是否可见，当然实现的就是每次都能查到已提交的最新数据效果了。

注意：begin/start transaction 命令并不是一个事务的起点，在执行到它们之后的第一个修改操作或加排它锁操作(比如select...for update)的语句，事务才真正启动，才会向mysql申请真正的事务id，mysql内部是严格按照事务的启动顺序来分配事务id的。

总结：

MVCC机制的实现就是通过read-view机制与undo版本链比对机制，使得不同的事务会根据数据版本链对比规则读取同一条数据在版本链上的不同版本数据。

```
1 文档：06-VIP-Mysql锁机制与优化实践以及MVCC底层原理剖析
2 链接：http://note.youdao.com/noteshare?id=91f18a2472ba9712d3749a6cad827d24&sub=82B026B480674333AF3FA9DA3A374815
```