一、 Lock 和 latch 介绍

Latch 锁的对象是数据库线程中的资源，分为 mutex 和 readwriteLock，用来保证并发过程中临近资源安全性，没有死锁检测。

Lock 锁的对象是事务，用来锁定数据库中的对象，如表、页、行。Lock 对象在事务 commit 或 rollback 后释放，有死锁机制。

表 6-1 lock 与 latch 的比较

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

对于 innodb 中的 latch，我们可以通过 show engine innodb mutex 来查询。Debug 模式下，可以看到更多信息。

至于 lock 信息，在可以通过 show engine innodb status 查询。具体死锁排查过程参见《mysql-5-事务-死锁-binlog》

二、 Lock 算法

行锁主要有 3 种方式

- 行锁 record lock：只锁定这一行
- 间隙锁 gap lock：锁定一个左开右闭区间范围
- next key lock：record lock + gap lock

这里详细描述下 next key lock。例如一个索引有 10，11，13，20 这 4 个值，那么 gap lock 区间为：

(-INF, 10]

(10, 11]

(11, 13]

(13, 20]

(20, +INF)

这里的区间必须是左开右闭区间。

需要注意的是，只有 RR 支持 next key lock 和 gap lock，RC 隔离等级的情况下，不符合 where 条件的行记录的间隙锁会被释放。

下面以 smstest 表为例来解释 RR 隔离级别下各个情况下 next key lock 情况。

```
CREATE TABLE `smstest` (
`sn` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增编号',
`phoneNo` int(16) NOT NULL ,
`channelType` int(11) NULL DEFAULT NULL COMMENT '通道识别',
`status` tinyint(4) NOT NULL COMMENT '短信转态,1.发送成功,2.发送失败,3.发送异常',
```

```
PRIMARY KEY (`sn`),
INDEX `in_p_index` (`phoneNo`) USING BTREE
)
ENGINE=InnoDB
DEFAULT CHARACTER SET=utf8 COLLATE=utf8_general_ci
COMMENT='短信发送成功记录表'
AUTO_INCREMENT=114
ROW_FORMAT=DYNAMIC;
```

Sn 为主键，phoneNo 为辅助索引。

初始数据如下：

sn	phoneNo	channelType	status
1	1	60	10
9	10	10	2
10	10	60	3
11	10	60	4
12	10	60	1
16	16	45	56
109	111	60	1

1. 唯一索引的 next key lock——行锁

如果 where 条件是唯一键（主键），则 next key lock 会降级成 record lock，只锁定 where 条件范围内的行。

例如：

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`sn`=12 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(13,110,1,1); commit;	不阻塞，可以直接插入，说明没有间隙锁，只有 record lock。
commit;		

2. 辅助索引的 next key lock

1) Where 条件=

如果 where 条件中是等于，此时会有左右两个 gap lock 以及 record lock 本身。例如：

select * from smstest t where t.`phoneNo`=16 for UPDATE;

其 next key lock 所在索引 phoneNo 的区间为：(10, 16), 16,(16, 111)。这与辅助索引本身结构有关，此时的 next key lock 如下：

PhoneNo 辅助索引	1	10	GAP LOCK	16 (RECORD LOCK)	GAP LOCK	111

Sn(主键)	1	9	10	11	12		16		109

注意 10 和 111 本身来说都不在间隙锁范围内，但是由于辅助索引的数据页中聚簇索引是顺序排列的，如果我们插入聚簇索引在间隙锁范围内的新数据，例如：

insert into SmsTest values(13,10,1,1); 第一列是 sn，第二列是 phoneNo

显然位于间隙锁范围内，此时任然会阻塞。反之，如果我们插入间隙锁外的数据，如：

insert into SmsTest values(8,10,1,1);

则可以直接插入。

下面来看例子：

间隙锁区间范围内

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`=16 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(8,10,1,1); commit;	间隙锁范围内，阻塞，等待超时
commit;		

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`=16 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(108,111,1,1); commit;	虽然 111 是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时
commit;		

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`=16 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(110,111,1,1); commit;	实际数据插入位置在间隙锁范围外，直接插入
commit;		

2) Where 条件 <

如果字段在 where 条件中是小于 x，那么首先会在索引中从 x 向右寻找最近的那个索引值 y，where col < x 等价于 where col < y。

例如下表中，对于 select * from newtable t where t.`phoneNo`<15 for UPDATE;
就相当于 select * from newtable t where t.`phoneNo`<16 for UPDATE;

sn	phoneNo	channelType	status
1	1	60	10
9	10	10	2
10	10	60	3
11	10	60	4
12	10	60	1
16	16	45	56
109	111	60	1

此时 next key lock 加锁范围如下，注意最左侧的 gap lock 位于负无穷。显然我们可以插入(17,16,1,1)，但是不能插入(15,16,1,1)

PhoneNo 辅助索引	GAP LOCK	1	10			GAP LOCK	16	111
Sn(主键)		1	9	10	11	12	16	109

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`<16 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(15,16,1,1); commit;	虽然 16 是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时
commit;		

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo<16 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(17,16,1,1); commit;	实际数据插入位置在间隙锁范围外，直接插入
commit;		

如果有小于等于，则间隙锁范围需要与 1)中的锁范围求并集合

3) Where 条件 >

如果字段在 where 条件中是大于 x，那么首先会在索引中从 x 向左寻找最近的那个索引值。

例如下表中，对于 select * from newtable t where t.`phoneNo`>14 for UPDATE;

就相当于 select * from newtable t where t.`phoneNo`>10 for UPDATE;

sn	phoneNo	channelType	status
1	1	60	10
9	10	10	2
10	10	60	3
11	10	60	4
12	10	60	1
16	16	45	56
109	111	60	1

此时 next key lock 加锁范围是如下，，注意最右侧的 gap lock 位于正无穷。显然我们可以插入(8,10,1,1)，但是不能插入(13,10,1,1)

PhoneNo 辅助索引	1	10				GAP LOCK	16	111	GAP LOCK
Sn(主键)	1	9	10	11	12		16	109	

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`>14 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(13,10,1,1); commit;	虽然 10 是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时
commit;		

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

T1	T2	结果及描述
start TRANSACTION;		
select * from smstest t where t.`phoneNo`>14 for UPDATE;		
	start TRANSACTION; insert into SmsTest values(8,10,1,1); commit;	实际数据插入位置在间隙锁范围外，直接插入
commit		

如果有小于等于，则间隙锁范围需要与 1)中的锁范围求并集合

3. 无索引情况下的 next key lock——锁表

where 条件中字段无索引情况下，间隙锁会扩大到整个表，即锁表！

T1	T2	结果及描述
start TRANSACTION;		
update SmsTest set channelType=10 where `status` = 2		
	start TRANSACTION; update SmsTest set channelType=11 where `status` = 5 commit;	Status 无索引，直接锁表
commit;		

三、 Lock 分类

1. 行锁与表锁

行锁和表锁主要有两种：

- 排他锁（X-lock）：写锁
- 共享锁（S-LOCK）：读锁、

表 6-3 排他锁和共享锁的兼容性

	X	S
X	不兼容	不兼容
S	不兼容	兼容

由于行锁和表锁的加锁对象不同，所以二者如何同步也就成了问题。SessionA 我们对一行数据加了 S 锁，sessionB 试图对该表整体加 X 锁，显然互斥。表 X 锁如何才能获取呢[1]？

step1：判断表是否已被其他事务用表锁锁表

step2：判断表中的每一行是否已被行锁锁住。

step2 中需要遍历全表，显然效率极低。所以我们在加行锁之前对表对象加入意向锁，这样就可以表征行锁加入对表锁的影响。在意向锁存在的情况下，事务 A 必须先申请表的意向共享锁，成功后再申请一行的行锁。

在意向锁存在的情况下，上面的判断可以改成：

step1：不变

step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务 B 申请表的写锁会被阻塞。

表 6-4　InnoDB 存储引擎中锁的兼容性

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

2. 一致性非锁定读

一致性非锁定读是指 innodb 通过多版本并发控制（MVCC）来读取当前时间数据库中的行数据。如果读取过程中正执行 DML 操作，读操作不会等待行锁释放，而是读取行的一个快照。

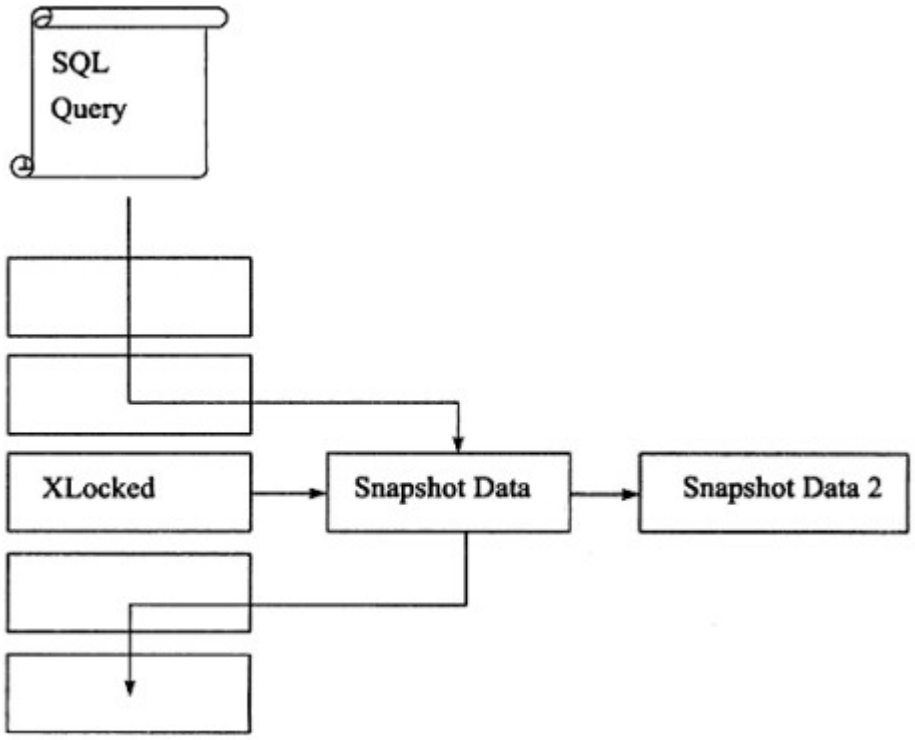


图 6-4　InnoDB 存储引擎非锁定的一致性读

显然，一致性非锁定读是对普通读写锁的极大优化，可以提高数据库查询效率。其中，snapshot 是通过 **undo 段**来完成的，undo 用于事务中回滚数据，因此快照本身没有额外开销。

1) Read Committed——当前读

Read Committed 级别下，总是进行当前读（最近一份快照），即读取最近一份 commited 数据，所以会有不可重复读的现象.

例子如下：

- ✓ sessionA：

```
Start transaction;
Select * from parent where id = 1;
```

```
mysql> SELECT* FROM parent WHERE id = 1;
+----+
| id  |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

✓ sessionB：

Start transaction;

udpate parent set id = 3 where id = 1;

此时 sessionB 尚未 commit。

✓ sessionA：

Select * from parent where id = 1;

```
+----+
| id  |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

✓ sessionB：

commit;

✓ sessionA：

Select * from parent where id = 1;

此时结果为空。说明 RC 级别下是读 commit 最新版本，即当前读。

2) Repeatable Read——快照读

Repeatable Read 级别下，总是进行快照度（第一次读的快照），所以不会有不可重复读的问题，且可以避免一般意义上的幻读。

例子如下：

✓ sessionA：

Start transaction;

Select * from parent where id = 1;

```
mysql> SELECT* FROM parent WHERE id = 1;
+----+
| id  |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

✓ sessionB：

Start transaction;

udpate parent set id = 3 where id = 1;

此时 sessionB 尚未 commit。

✓ sessionA：

Select * from parent where id = 1;

```
+----+
| id  |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

✓ sessionB：

commit;

✓ sessionA：

Select * from parent where id = 1;

```
+----+
| id  |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

此时结果任然为事务快事的快照。

表 6-8　示例执行的过程

时间	会话 A	会话 B
1	BEGIN	
2	SELECT * FROM parent WHERE id = 1;	
3		BEGIN
4		UPDATE parent SET id=3 WHERE id = 1;
5	SELECT * FROM parent WHERE id = 1;	
6		COMMIT;
7	SELECT * FROM parent WHERE id = 1;	
8	COMMIT	

3. 一致性锁定读

- Select … for update——X 锁
- Select … lock in share mode——S 锁

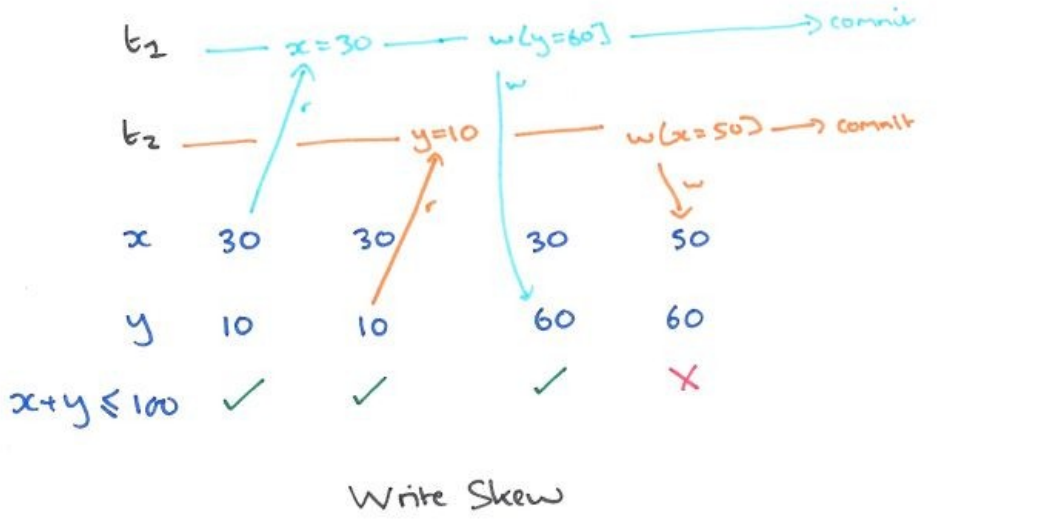
4. RR 级别下一致性非锁定读与幻读

RR 级别下既然有了 MVCC，是不是就可以避免幻读了呢？

1) 普通幻读消除

MySQL（innodb）的 RR 隔离级别实际上是 snapshot isolation，可以避免通常意义的幻读。snapshot isolation 的问题是无法处理如下的 read-write conflict（write skew）。

2) write skew 解决



我们约定，x + y <= 100. 如果都是快照读/当前读，t1 先读取 x=30，然后根据快照/当前版本判断，进行 y=60 更新满足 x + y <= 100。

t2 先读取 y = 10，在 t1 更新 y=60 之后，由于 t1 没有提交，此时 t2 依然可以满足

x+y<=100 的情况下修改 x=50。

之后 t1 和 t2 都可以成功 commit。但显然违背了我们之前的约束条件。

这种现象称为 write skew。由于 UPDATE 本身也是一种 read-write，如果执行 UPDATE 也会有 write skew 问题, 那对实际应用来说就太糟糕了。

Innodb 为了解决 RR 情况下的这个问题，强行把 read 分为快照读和当前读（Locking read），快照读就是普通的 read，当前读就是 select … for update。在 update 或 select … for update 时，Innodb 执行当前读，在过程中加入 record lock 和 gap lock（next key lock），相当于变相提升到了 Serializable 级别，从而消除 write skew。

3) next key lock 的必要性

上文中，update 或 select … for update 时，Innodb 执行当前读，在过程中加入 record lock 和 gap lock（next key lock）。

Record lock 就可以通过悲观锁实现目标资源的串行化，为啥 recordLock 还不够，还得使用 next key lock 呢？

在 update 或 select … for update 之后，会更新快照进行当前读，如果只有 record lock 没有 next key lock，在这个过程中可能会有 gap 内新数据插入，从而出现了原有幻读的现象。所以必须有 next key lock 来在更新快照后避免传统意义上的幻读出现。

4) 新的不一致读

虽然解决了 write skew，但是当前读（Locking read）和快照本身是矛盾，会出现**新的不一致读**。由于并没有 Serializable 的悲观锁，在两个不同的事务中仍然可以并发的执行 update 语句，update/select for update 执行当前读之后，会更新快照，导致与之前的快照读不一致的情况。

只有 Serializable 才能避免这种情况。

例如：mysql 正确处理的结果如下[4]：

```
CREATE TABLE char_encode (
    glyph CHAR(1) NOT NULL,
    codepoint TINYINT(3) NOT NULL
) ENGINE=InnoDB
INSERT INTO char_encode VALUES ('a', 97), ('b', 98);
```

✓ SESSION-1

序号	执行语句	结果
1	SHOW SESSION VARIABLE LIKE 'tx_isolation'	REPEATABLE-READ
2	START TRANSACTION;	
3	SELECT * FROM char_encode;	a | 97 b | 98

✓ SESSION-2

序号	执行语句	结果	解释
1.	SHOW SESSION VARIABLE LIKE 'tx_isolation'	REPEATABLE-READ	

2.	START TRANSACTION;		
3.	SELECT * FROM char_encode;	a | 97 b | 98	
4.	UPDATE char_encode SET codepoint = 100 WHERE glyph = 'a';	1 Rows affected;	
5.	SELECT * FROM char_encode;	a | 100 b | 98	如预期，一个事务能看到自己的本地改变
6.	COMMIT		

✓ SESSION-1

序号	执行语句	结果	解释
1.	SELECT * FROM char_encode WHERE glyph = 'a';	a | 97	完美，如预期，尽管 session-2 提交了，但是一致性非锁定读就是应该读自己事务中第一次读取时的快照
2.	UPDATE char_encode SET codepoint = codepoint + 1 WHERE glyph = 'a';	1 Rows affected;	
3.	SELECT * FROM char_encode WHERE glyph = 'a';	a | 101	刚才读才是 97，更新后应该是 97+1=98，为什么不是快照读了？因为 update 语句或 select for update 会更新快照，和之前的快照不一致

根据[4]中的官方解释，mysql 允许覆盖更新这种行为（不遵守 first commit win rule)，这让它产生了上面的这种幻像（但又不是幻读）。

4. 隔离级别与 binlog

一般来说，如果是 RC 隔离级别，我们必须将 binlog 设置为 Row。这不仅是数据记录更为详细，更重要的是 binlog 是在事务 commit 后才生产，由于 RC 没有 next-key lock 只有行锁，只能锁定具体的行而不是对满足条件的索引进行锁定，使用 statement 格式会生成和当前主库中不一致的 binlog，在执行 master-slave 复制时出错。而 RR 隔离级别利用 nextkey-lock 则可以避免这个情况。具体参见后文。

例如数据如下：

sn	phoneNo	channelType	status
1	1	60	8
8	10	1	1
9	10	10	2
10	10	60	3
11	10	1	1
12	10	60	1
13	10	1	1
15	16	1	1
16	16	45	56
100	100	1	1
109	111	60	1
113	113	1	1

1) RR——安全使用 binlog-Statement

对于 RR 隔离级别，由于在 tx1 已经对索引 phoneNo=16 前后加入了 next keylock 且未提交，tx2 此时试图将 sn=1（sn 也是辅助索引）的 phoneNo 修改为 16，对索引 phoneNo 而言是执行了插入 phoneNo=16 的操作，显然在 nextkey-lock 锁范围内，此时 tx2 阻塞等待 tx1 释放 next key lock。此时事务提交顺序 tx1, tx2。对应 statement 类型的 binlog 为：

Tx1: update smstest t set t.status = 9 where t.phoneNo='16'

Tx2: update smstest t set t.phoneNo='16' where t.sn='1';

tx1	tx2	解释
set tx_isolation='repeatable-read';		设置当前 session 事务隔离级别为 RR
BEGIN; update smstest t set t.status = 9 where t.phoneNo='16'		对索引 phoneNo=16 前后加入了 next keylock
	set tx_isolation='read- read'; BEGIN;	
	update smstest t set t.phoneNo='16' where t.sn='1';	tx2 此时试图将 sn=1（sn 也是辅助索引）的 phoneNo 修改为 16，对索引 phoneNo 是执行了插入 phoneNo=16 的操作，显然在 tx1 的 next keylock 锁范围内，tx2 阻塞等待 tx1 释放锁才能执行 commit
	commit;	阻塞
commit;		
		此时 tx2 commit 才成功

2) RC——数据库状态与 binlog-Statement 不一致

对于 RC 隔离级别，情况如下：

tx1	tx2	解释
set tx_isolation='read-committed';		设置当前 session 事务隔离

		级别为 RC
BEGIN; update smstest t set t.status = 9 where t.phoneNo='16'		对索引 phoneNo=16 只加入了行锁
	set tx_isolation='read-committed'; BEGIN; update smstest t set t.phoneNo='16' where t.sn='1';	tx2 此时试图将 sn=1（sn 也是辅助索引）的 phoneNo 修改为 16，对索引 phoneNo 是执行了插入 phoneNo=16 的操作，由于只有行锁直接成功
	commit;	成功
commit;		

由于是 RC 隔离级别，只有行锁，tx2 会直接执行成功，此时产生的 statement 的 binlog 如下：

Tx2: update smstest t set t.phoneNo='16' where t.sn='1';

Tx1: update smstest t set t.status = 9 where t.phoneNo='16'

显然 RC 隔离级别下，数据库中数据和 binlog 不是等价的。

master 数据库中实际数据如下：

sn	phoneNo	channelType	status
1	16	60	8
8	10	1	1
9	10	10	2
10	10	60	3
11	10	1	1
12	10	60	1
13	10	1	1
15	16	1	9
16	16	45	9
100	100	1	1
109	111	60	1
113	113	1	1

而按照上面的 binlog，在 slave 中所有 *phoneNo=16* 的 status 都等于 9.

所以 RC 情况下 binlog-format 只能是 row，不能是 statement。

5. 生产环境的隔离性

一般来说，普通电商业务并没有 RR 实际使用场景，所以一般生产环境使用 RC 隔离级别+row binlog，使用 row 类型 Binlog 还方便使用 canal 等中间件监听数据变化情况。当然，一般来说 RR 和 RC 在速度上差别不大，RR+row 类型 binlog 效率会慢很多。

参考

[1] InnoDB 的意向锁有什么作用？https://www.zhihu.com/question/51513268
[2] MySQL 中隔离级别 RC 与 RR 的区别 https://www.cnblogs.com/digdeep/archive/2015/11/16/4968453.html
[3] mysql 的 innodb 通过 nextkey lock 解决了幻读，为什么还说默认隔离级别是可重复读 https://www.zhihu.com/question/350352149/answer/865763594
[4] 既然 MySQL 中 InnoDB 使用 MVCC，为什么 REPEATABLE-READ 不能消除幻读？https://www.zhihu.com/question/334408495/answer/745098902
[5] mysql 解决 RR 下的 write skew https://bugs.mysql.com/bug.php?id=63870
[6] MySQL 隔离级别为读提交的时候为什么会出现的数据和日志不一致，还必须把 binlog 格式设置为 row？https://www.zhihu.com/question/344037151