

# MySQL 锁机制



Part 01

前置概念

Part 02

锁分类

Part 03

锁详解

Part 04

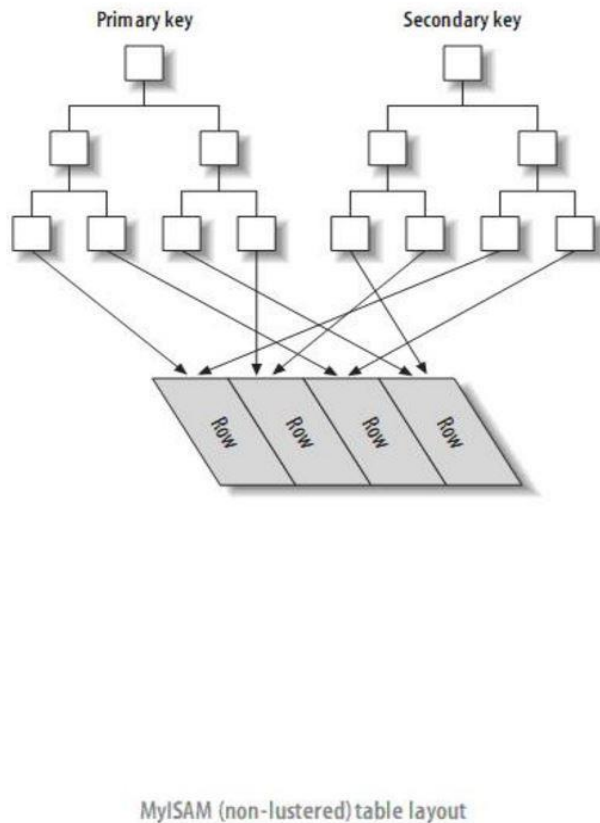
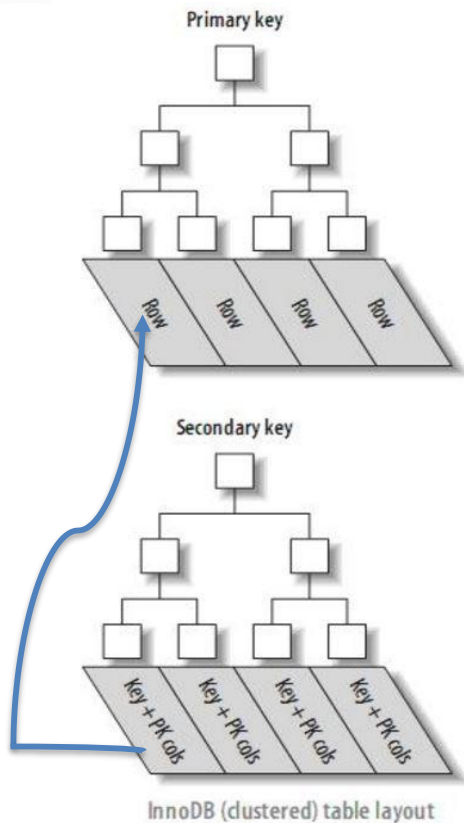
总结

# 01 PART

## 前置概念

- 数据库引擎
- 锁的对象
- 事务隔离级别

- MyISAM
  - 只支持表级锁
  - 不支持事务
  - 非聚簇索引
- InnoDB
  - 支持行级锁
  - 支持事务
  - 聚簇索引

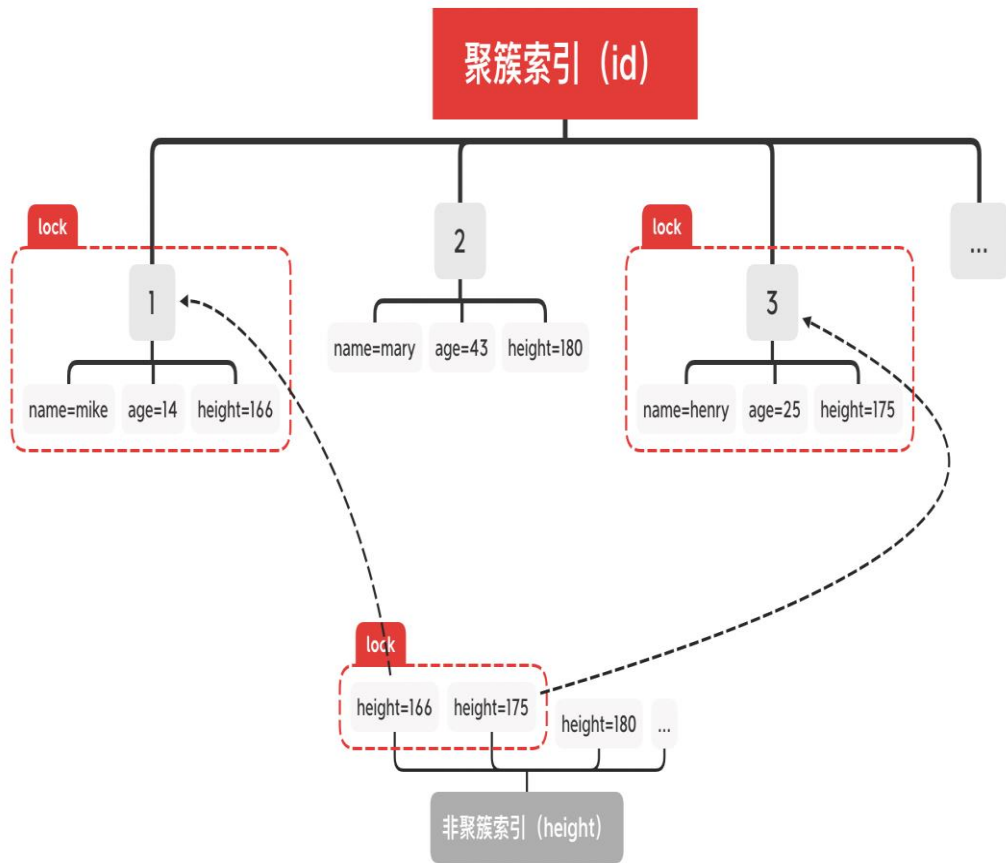


聚簇索引和非聚簇索引并不是单独的索引类型，而是一种数据存储方式。

innoDB 的行级锁是基于索引实现的，即：加锁的对象是索引而非具体的数据。

当加锁操作使用**聚簇索引**时，InnoDB 会锁住聚簇索引；而使用**非聚簇索引**时，InnoDB 会先锁住非主键索引，再锁定非聚簇索引锁对应的聚簇索引。

行级锁的加锁条件必须有对应的索引项，否则会退化为**表级锁**。



## 表信息

```
mysql> desc user_info;
```

Field	Type	Null	Key	Default	Extra
id	bigint(20) unsigned	NO	PRI	NULL	auto_increment
user_name	varchar(255)	NO	UNI		
age	int(20) unsigned	NO		NULL	
height	int(20) unsigned	NO	MUL	NULL	
weight	int(20) unsigned	NO		NULL	

5 rows in set (0.01 sec)

```
mysql> select * from user_info;
```

id	user_name	age	height	weight
2	a	8	166	50
3	b	12	177	60
4	c	16	188	70
5	d	19	199	80
6	e	27	185	90
7	f	37	160	100

6 rows in set (0.00 sec)

## 事务1

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user_info where age=8 for update;
+----+-----+-----+-----+-----+
| id | user_name | age | height | weight |
+----+-----+-----+-----+-----+
| 2 | a         | 8   | 166    | 50     |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

## 事务2

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update user_info set height=200 where id =3;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql> |
```

- 未提交读 (READ UNCOMMITTED)
- 提交读 (READ COMMITTED)
- 可重复读 (REPEATABLE READ)
- 串行化 (SERIALIZABLE)

InnoDB 通过 MVCC 实现了快照读

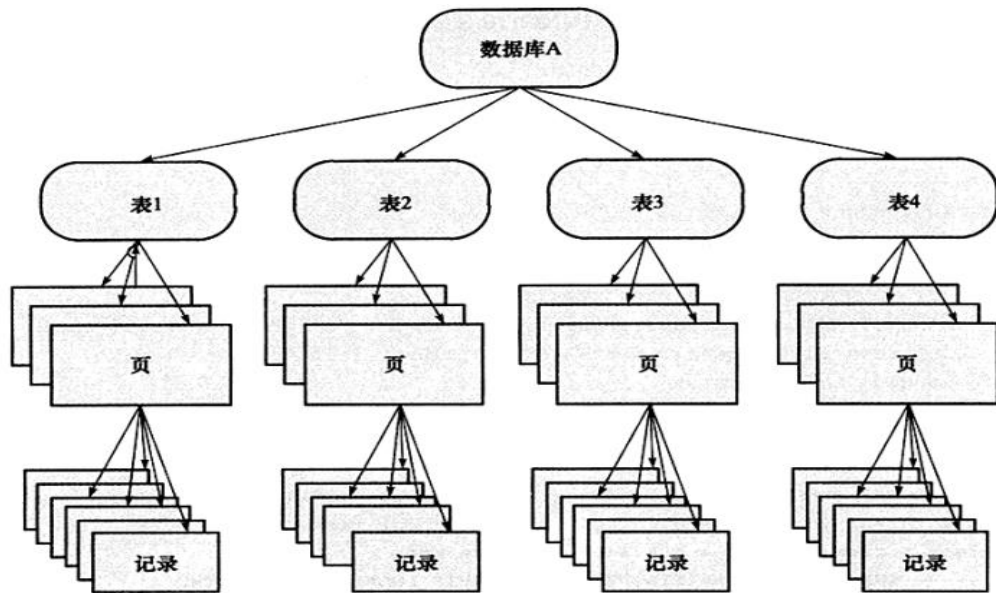
- MVCC (多版本并发控制)
- GAP LOCK (间隙锁)
- NEXT-KEY LOCK (临键锁)

## 02 PART

### 锁的类型

- 粒度
- 锁模式
- 阻塞范围

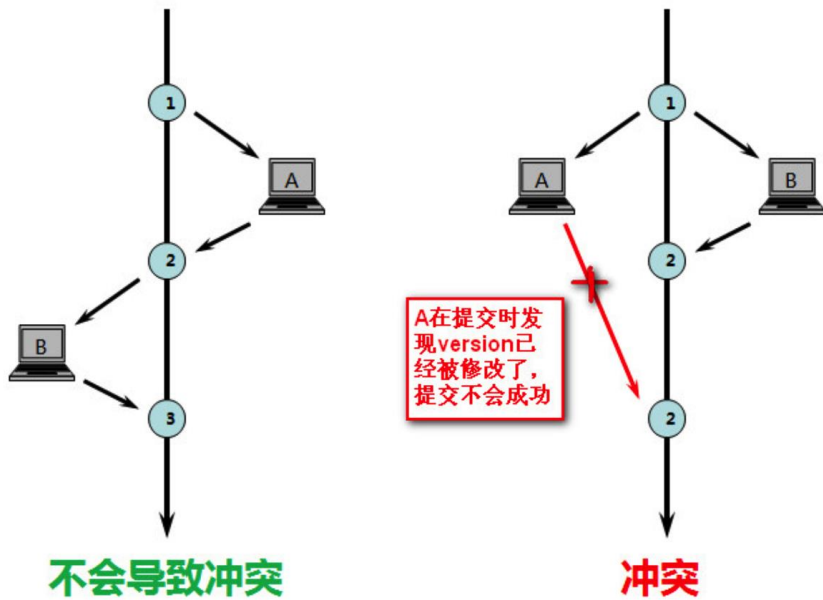


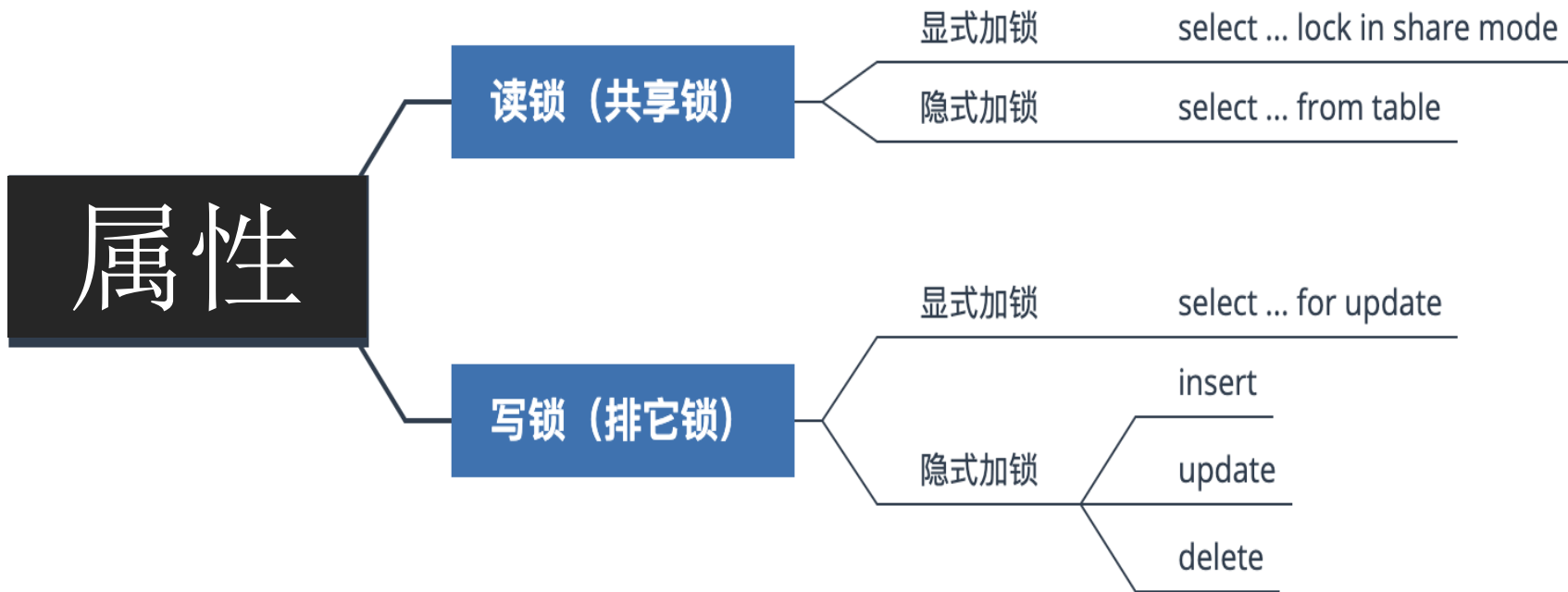


- 表级锁：锁住整张表
- 页级锁：锁住指定数据区间
- 行级锁：锁住某一条数据
- `LOCK TABLES tbl_name;`
- `SELECT ID FROM tbl_name WHERE age BETWEEN 1 AND 10 FOR UPDATE;`
- `SELECT ID FROM tbl_name WHERE age=12 FOR UPDATE;`

InnoDB 支持多粒度锁（multiple granularity locking），它允许行级锁与表级锁共存

- 乐观锁：先对数据进行操作，提交时再校验权限状态
- 悲观锁：先获取操作权限，再对数据进行操作





# 03 PART

## 详解 InnoDB 中的锁

- 共享锁
- 排它锁
- 记录锁
- 间隙锁
- 临键锁
- 自增锁
- 插入意向锁

共享锁又称读锁（S锁），当一个事务获取了某行数据的共享锁，其他事务亦可获取该行数据的共享锁，但不能获取该行数据的排他锁。即：共享锁间互相兼容，但与排它锁互斥。

**SELECT column FROM table ... LOCK IN SHARE MODE;** // 显式加锁，当前读

在可重复读的事务隔离级别下，所有未通过 **LOCK IN SHARE MODE** 显式加锁的 **SELECT** 语句都是快照读，快照读不会对所属的数据行加共享锁。

在获取某数据行的共享锁前，必须先获取该数据行所在表的意向共享锁。

排他锁又称写锁（X锁），当一个事务获取了某行数据的排他锁，其他事务既不可获取改行数据的共享锁，也不可获取该行数据的排它锁。即：排他锁与任何锁互斥。

**SELECT column FROM table ... FOR UPDATE;**

// 显式加锁

**UPDATE table SET aget=14 WHERE id=1;**

// 隐式加锁，DELETE、INSERT 语句也会隐式加排它锁

**DELETE FROM table WHERE id=1;**

**INSERT INTO table VALUES(1);**

在获取某数据行的排他锁前，必须先获取该数据行所在表的意向排他锁。



## 意向锁 (Intention Locks)

**意向共享锁** (intention shared lock, IS) : 事务有意向对表中的某些行加**共享锁** (S锁)

事务要获取某些行的 S 锁, 必须先获得表的 IS 锁:

```
SELECT column FROM table ... LOCK IN SHARE MODE;
```

// 执行后 **table** 会被增加意向共享锁

**意向排他锁** (intention exclusive lock, IX) : 事务有意向对表中的某些行加**排他锁** (X锁)

事务要获取某些行的 X 锁, 必须先获得表的 IX 锁:

```
SELECT column FROM table ... WHERE id=1 FOR UPDATE;
```

// 执行后 **table** 会被增加意向排他锁

## 意向锁 (Intention Locks)

意向锁是一种不与行级锁冲突表级锁

	意向共享锁 (IS)	意向排他锁 (IX)	共享表锁 (S)	排他表锁 (X)
意向共享锁 (IS)	兼容	兼容	兼容	互斥
意向排他锁 (IX)	兼容	兼容	互斥	互斥



## 意向锁 (Intention Locks)

意向锁的作用：

如果一个事务试图在表级别上添加共享或排它锁，则会受到由其他事务控制的表级别意向锁的阻塞。该事务在锁定该表前不必检查各个页或行锁，而只需检查表上的意向锁。

例如：

假设一张表中有 100w 条数据，事务 A 想要获取该表的表级排他锁，在没有意向锁的情况下，事务 A 需要逐个检查 100w 条数据中，是否有任意一行数据存在共享锁或排它锁。但引入意向锁后，获取表锁前只需检查该表是否存在意向锁，即可判断是否有获取表锁的权限。

## 记录锁 ( Record Locks )

记录锁定是对索引记录的锁定。

例：

SELECT \* FROM t WHERE id = 10 FOR UPDATE;

1. id 必须为主键列或唯一索引列
2. id = 10 的数据行必须实际存在
3. WHERE 子句必须为等值查询

Id (PK)	age
1	12
3	13
5	15

- 事务 A:
1. SELECT \* FROM t WHERE id IN(1,3,5) FOR UPDATE;
  2. SELECT \* FROM t WHERE id BETWEEN 1 AND 5 FOR UPDATE;
  3. SELECT \* FROM t WHERE id > 0 FOR UPDATE;

事务 B: INSERT INTO t VALUES(4,20);

## 记录锁 (Record Locks)

事务 A:

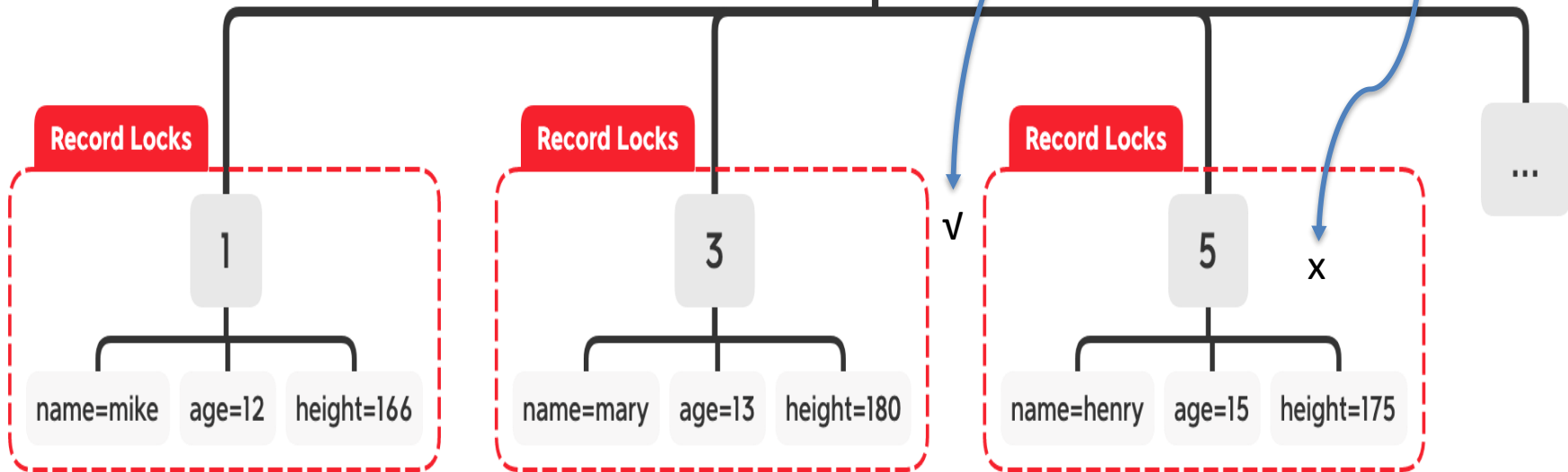
```
SELECT * FROM t WHERE id IN(1,3,5)  
FOR UPDATE;
```

索引 (id)

事务 B:

```
INSERT INTO t VALUES(4,20);
```

```
UPDATE t SET age=15  
WHERE id=5;
```



# 03 PART

## 间隙锁 ( Gap Locks )

事务 A:

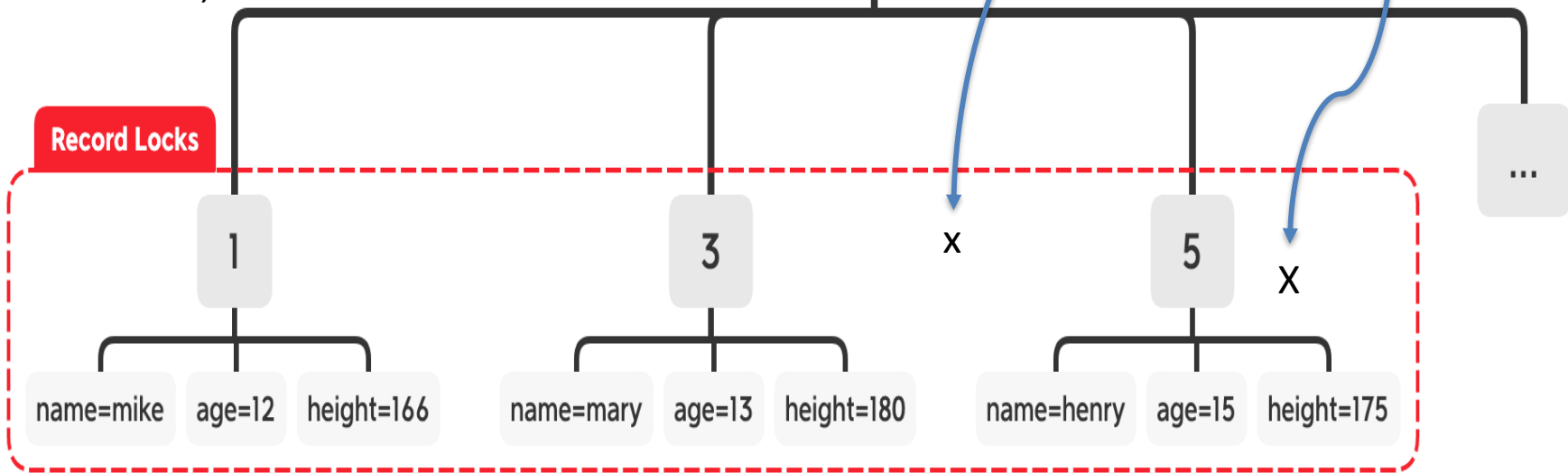
```
SELECT * FROM t WHERE id  
BETWEEN 1 AND 5  
FOR UPDATE;
```

事务 B:

```
INSERT INTO t VALUES(4,20);  
  
UPDATE t SET age=15  
WHERE id=5;
```

索引 (id)

Record Locks



## 间隙锁 ( Gap Locks )

间隙锁是对索引之间间隙的锁定。

亦或是对第一个索引记录之前，或最后一个索引记录之后的间隙的锁定

Id(PK)	age(UK)	height(key)
1	10	100
3	20	200
5	30	300

SELECT \* FROM t WHERE age = 15 FOR UPDATE;      age (10,20)

SELECT \* FROM t WHERE id = BETWEEN 1 AND 5 FOR UPDATE;      Id [1,5]

SELECT \* FROM t WHERE height=200;      height(100,300)

## 间隙锁 ( Gap Locks )

间隙锁存在的唯一目的是防止其他事务插入区间。

**间隙锁是可以共存的**，一个事务持有的间隙锁不会阻止另一事务对相同的间隙区间进行锁定。

事务1:

```
SELECT * FROM t WHERE age = 15 FOR UPDATE;
```

事务2:

```
SELECT * FROM t WHERE age = 15 FOR UPDATE;
```

```
INSERT INTO t VALUES(4,18,320);
```

成功

阻塞

Id(PK)	age(UK)	height(key)
1	10	100
3	20	200
5	30	300

## 临键锁 ( Next-Key Locks )

临键锁是索引记录上的记录锁和索引记录之前的间隙上的间隙锁的组合。  
即：临键锁控制一段左开右闭区间的数据。

Id(PK)	age(UK)	height(key)
1	10	100
3	20	200
5	30	300

height 列潜在的临键锁：  
 $(-\infty, 100]$   
 $(100, 200]$   
 $(200, 300]$   
 $(300, +\infty)$

所有对非唯一索引进行的加锁操作，都是基于临键锁实现的。

## 临键锁 ( Next-Key Locks )

Id(PK)	age(UK)	height(key)	comment
1	10	100	aa
3	20	200	bb
5	30	300	cc

最终：  
锁定 区间 (100,200]、(200,300)，  
锁定 id=3 的记录行

```
SELECT * FROM t WHERE height = 200 FOR UPDATE;
```

会分获取 Id = 3,5 的记录行的临键锁和记录行 id=1 的记录锁



## 临键锁 ( Next-Key Locks )

Id(PK)	age(UK)	height(key)	comment
1	10	100	aa
3	20	200	bb
5	30	300	cc

根据非唯一索引列 UPDATE 某条记录

```
UPDATE table SET comment = 'xx' WHERE height = 200;
```

根据非唯一索引列 锁住某条记录

```
SELECT * FROM table WHERE height = 200 FOR UPDATE;
```

在对非唯一索引进行加锁操作时，InnoDB 会获取该索引记录的**临键锁**和下一个区间的**间隙锁**。

最终锁定的区间为：  
(100,200] + (200,300)

同时也会锁定 id=3 的记录行

## 临键锁 ( Next-Key Locks )

Id(PK)	age(UK)	height(key)	comment
1	10	100	aa
3	20	200	bb
5	30	300	cc

间隙锁是可以共存的，  
一个事务持有的间隙锁  
不会阻止另一事务对相  
同的间隙区间进行锁定。

事务1:     SELECT \* FROM table WHERE height = 200 FOR UPDATE;

锁定 id=3 的记录 + 区间 (100,300)

事务2:     SELECT \* FROM table WHERE height = 300 FOR UPDATE;



锁定 id=5 的记录 + 区间 (200,+∞)

两条加锁语句申请了同一区间的间隙锁，但因间隙锁之间是不互斥的，故事务 2 不会被事务 1 阻塞

## 自增锁 ( Auto-INC Locks )

自增锁是在对含有 `AUTO_INC` 列的表进行插入操作时，产生的一种特殊的表锁。

在最简单的情况下，如果一个事务正在向表中插入值，且该表存在自增列，则任何其他事务在该表中进行的插入操作都会被阻塞。

1. `START TRANSACTION;`
2. `INSERT INTO t1 SELECT * FROM t2;`  加锁
3. `Query OK, 1 row affected`  锁释放

自增锁并不是在一个事务完成后才释放，而是在插入自增长值的 `SQL` 语句执行完成后立即释放。

## 自增锁 ( Auto-INC Locks )

自增插入分类:

插入类型	说明
insert-like	insert-like 指 所有的 插入 语句, 如 INSERT、REPLACE、INSERT...SELECT, REPLACE...SEECT、LOAD DATA 等
simple inserts	simple inserts 指能在插入前就确定插入行数的语句。这些语句包括 INSERT、REPLACE 等。需要注意的是: simple inserts 不包含 INSERT ...ON DUPLICATE KEY UPDATE 这类 SQL 语句
bulk inserts	bulk inserts 指在插入前不能确定得到插入行数的语句, 如 INSERT...SELECT, REPLACE...SELECT, LOAD DATA
mixed-mode inserts	mixed-mode inserts 指插入中有一部分的值是自增长的, 有一部分是确定的。如 INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d'); 也可以是指 INSERT ...ON DUPLICATE KEY UPDATE 这类 SQL 语句

## 自增锁 ( Auto-INC Locks )

在 InnoDB 中，可以通过 `innodb_automic_lock_mode` 干预自增时的加锁方式

`innodb_automic_lock_mode` 设置：

<code>innodb_autoinc_lock_mode</code>	说明
0	这是 MySQL5.1.22 版本之前自增长的实现方式，即通过表锁的 AUTO-INC Locking 方式。因为有了新的自增长实现方式，0 这个选项不应该是新版用户的首选项
1	这是该参数的默认值。对于“simple inserts”，该值会用互斥量（mutex）去对内存中的计数器进行累加的操作。对于“bulk inserts”，还是使用传统表锁的 AUTO-INC Locking 方式。在这种配置下，如果不考虑回滚操作，对于自增值列的增长还是连续的。并且在这种方式下，statement-based 方式的 replication 还是能很好地工作。需要注意的是，如果已经使用 AUTO-INC Locking 方式去产生自增长的值，而这时需要再进行“simple inserts”的操作时，还是需要等待 AUTO-INC Locking 的释放
2	在这个模式下，对于所有“INSERT-like”自增长值的产生都是通过互斥量，而不是 AUTO-INC Locking 的方式。显然，这是性能最高的方式。然而，这会带来一定的问题。因为并发插入的存在，在每次插入时，自增长的值可能不是连续的。此外，最重要的是，基于 Statement-Base Replication 会出现问题。因此，使用这个模式，任何时候都应该使用 row-base replication。这样才能保证最大的并发性能及 replication 主从数据的一致

## 自增锁 ( Auto-INC Locks )

T1: INSERT INTO t1 (c2) SELECT 1000 rows from another table ...

T2: INSERT INTO t1 (c2) VALUES ('xxx');

在上述 case 中，如果不使用表级锁，两条语句获得的自增量将会是不确定的。

....., 555, 556, 557, 558, 559, .....

不使用表锁

....., 997, 998, 999, 1000, 1001, .....

1, 2, 3, 4, 5, .....



使用表锁

## 插入意向锁 ( Gap Locks )

插入意向锁是在插入一条记录行前，由 **INSERT** 操作产生的一种特殊的间隙锁。

该锁用以表示插入意向，当多个事务在同一区间（Gap）插入位置不同的多条数据时，事务之间不需要互相等待。

假设存在两条值分别为 4 和 7 的记录，两个不同的事务分别试图插入值为 5 和 6 的两条记录，每个事务在获取插入行上独占的（排他）锁前，都会获取（4，7）之间的间隙锁，但是因为数据行之间并不冲突，所以两个事务之间并不会产生冲突（阻塞等待）。

## 插入意向锁 ( Gap Locks )

Id(PK)	age(UK)	height(key)
1	10	100
5	30	300

**T1: INSERT INTO users VALUES(3, 15, 150);**

**T2: INSERT INTO users VALUES(4, 16, 200);**

事务 B 是否会被事务 A 阻塞?      不会

锁区间: (1,5)

插入意向锁之间互不排斥，所以即使多个事务在同一区间插入多条记录，只要记录本身（主键、唯一索引）不冲突，那么事务之间就不会出现冲突等待。



## 查看当前存在的锁

```
LOCK_DATA: 0
***** 7. row *****
ENGINE: INNODB
ENGINE_LOCK_ID: 4788890152:2:4:7:140518948411760
ENGINE_TRANSACTION_ID: 6242
THREAD_ID: 57
EVENT_ID: 112
OBJECT_SCHEMA: test
OBJECT_NAME: user_info
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 140518948411760
LOCK TYPE: RECORD
LOCK_MODE: X
LOCK_STATUS: GRANTED
LOCK_DATA: 7
7 rows in set (0.00 sec)
```

X / S : 临键锁

X / S, Gap : 间隙锁

X / S, Rec\_not\_gap : 记录锁.

IX / S : 意向排他 / 共享锁

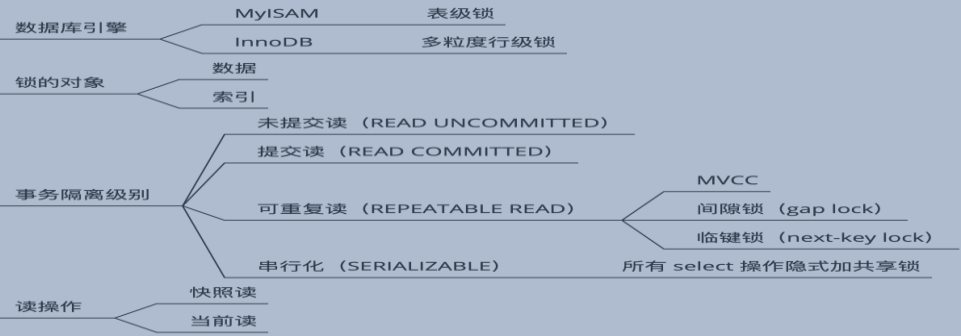
`SELECT * FROM performance_schema.data_locks\G`

# 04 PART

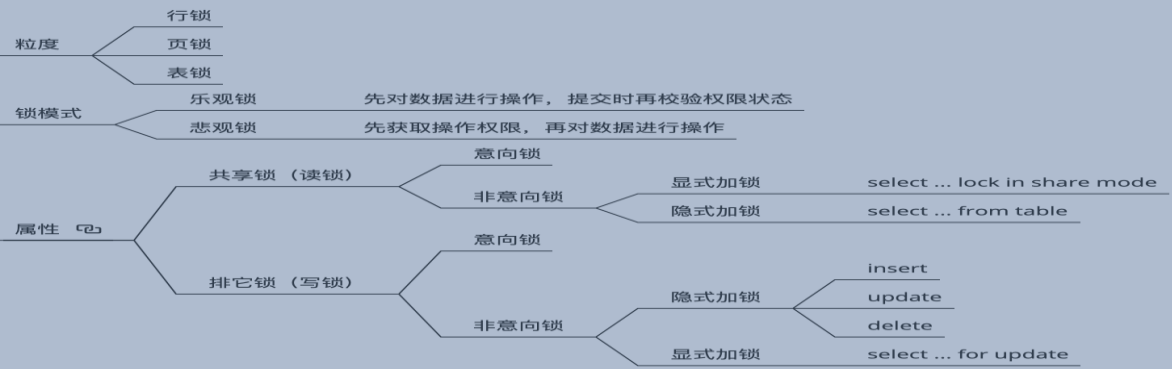
## • 总结

# MySQL 锁

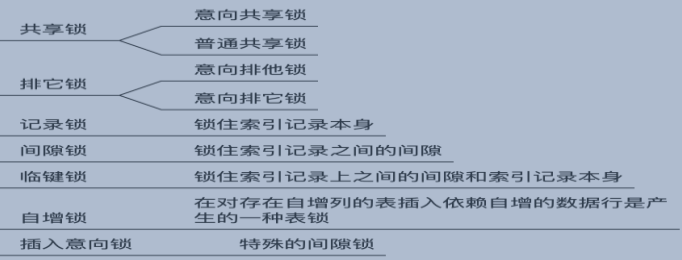
## 基础概念



## 锁分类



## 详解 InnoDB 中的锁



An abstract geometric design centered on a light gray circle containing the word "THANKS". This central circle is surrounded by two concentric thin gray circles. Several small dark gray dots are placed along these concentric circles, with some connected by thin curved lines, suggesting a stylized orbital or molecular structure. Additional dots of varying sizes are scattered in the background.

THANKS