事务的四大特性

A账户 10000 -2000 8000+2000=10000 8000+2000 写入buffer Pool（内存缓冲池） Redo Log 环形日志 磁盘

B账户 5000 +2000 7000

原子性（Atomicity）

也就是我们刚才说的不可再分，也就意味着我们对数据库的一系列的操作，要么都是成功，要么都是失败，不可能出现部分成功或者部分失败的情况，以刚才提到的转账的场景为例，一个账户的余额减少，对应一个账户的增加，这两个一定是同时成功或者同时失败的。全部成功比较简单，问题是如果前面一个操作已经成功了，后面的操作失败了，怎么让它全部失败呢？这个时候我们必须要回滚。

原子性，在 InnoDB 里面是通过 undo log （回滚日志，撤销日志）来实现的，它记录了数据修改之前的值（逻辑日志），一旦发生异常，就可以用 undo log 来实现回滚操作。

一致性（consistent）

指的是数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。比如主键必须是唯一的，字段长度符合要求。

除了数据库自身的完整性约束，还有一个是用户自定义的完整性。

举例：

1.比如说转账的这个场景，A 账户余额减少 1000，B 账户余额只增加了 500，这个时候因为两个操作都成功了，按照我们对原子性的定义，它是满足原子性的， 但是它没有满足一致性，因为它导致了会计科目的不平衡。

2.还有一种情况，A 账户余额为 0，如果这个时候转账成功了，A 账户的余额会变成-1000，虽然它满足了原子性的，但是我们知道，借记卡的余额是不能够小于 0 的，所以也违反了一致性。用户自定义的完整性通常要在代码中控制。

隔离性（isolation）

有了事务的定义以后，在数据库里面会有很多的事务同时去操作我们的同一张表或者同一行数据，必然会产生一些并发或者干扰的操作，对隔离性就是这些很多个的事务，对表或者 行的并发操作，应该是透明的，互相不干扰的。通过这种方式，我们最终也是保证业务数据的一致性。

持久性（Durable）

我们对数据库的任意的操作，增删改，只要事务提交成功，那么结果就是永久性的，不可能因为我们重启了数据库的服务器，它又恢复到原来的状态了。 Redo Log 二阶段提交的

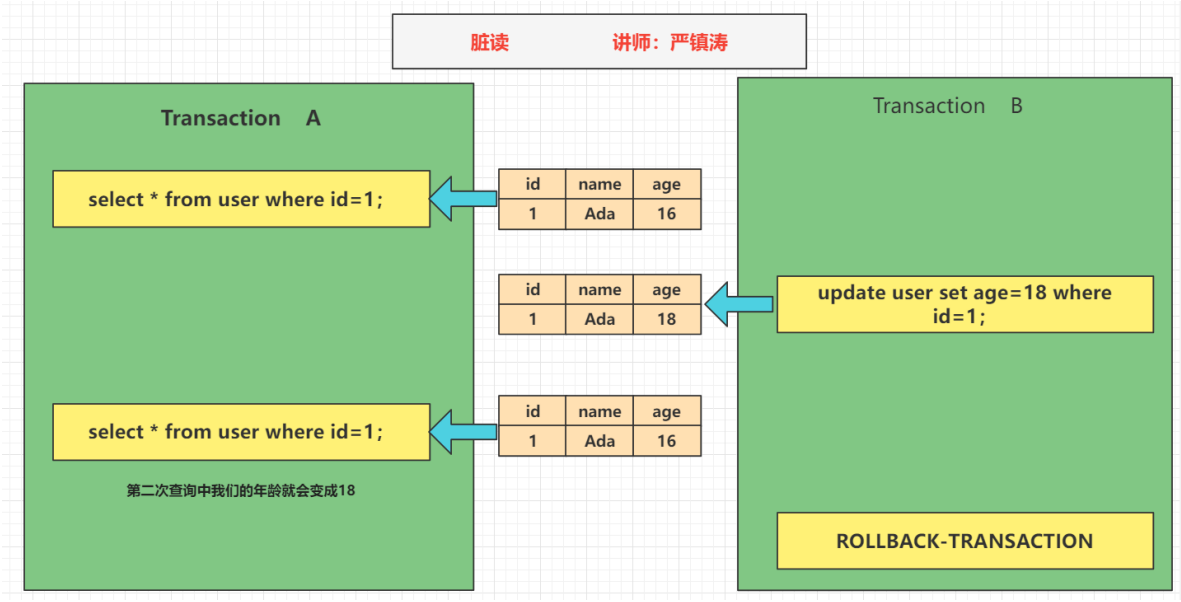
持久性怎么实现呢？数据库崩溃恢复（crash-safe）是通过什么实现的？持久性是通过 redo log 来实现的，我们操作数据的时候，会先写到内存的 buffer pool 里面，同时记录 redo log，如果在刷盘之前出现异常，在重启后就可以读取 redo log的内容，写入到磁盘，保证数据的持久性。

总结：原子性，隔离性，持久性，最后都是为了实现一致性

事务并发会带来什么问题?

当很多事务并发地去操作数据库的表或者行的时候，如果没有我们刚才讲的事务的Isolation 隔离性的时候，会带来哪些问题呢?

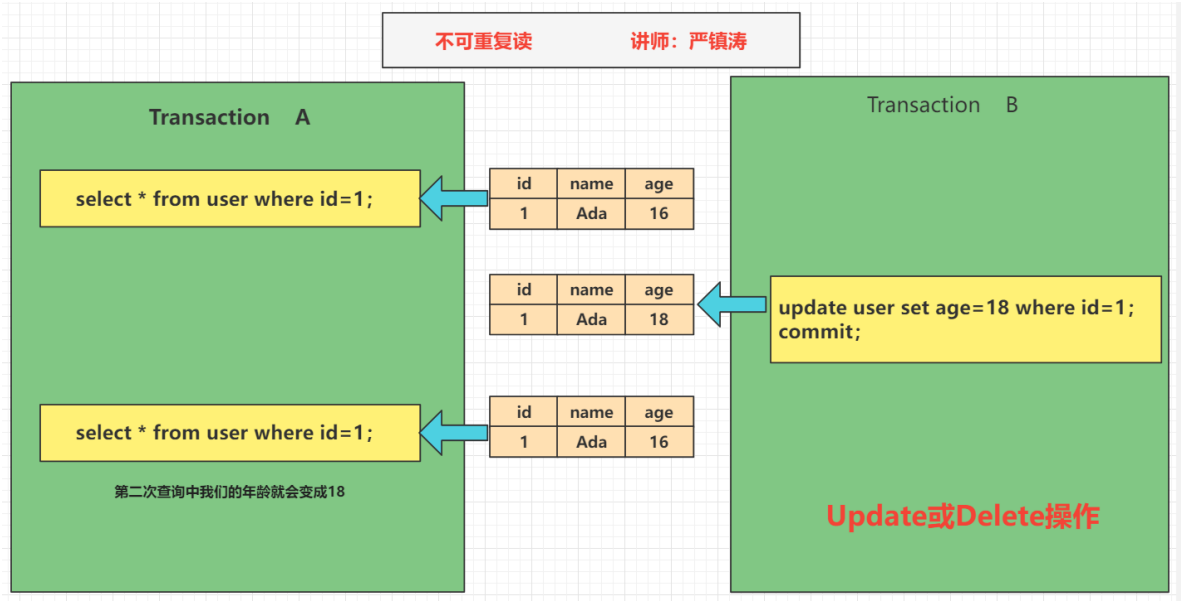
脏读



大家看一下，我们有两个事务，一个是 Transaction A，一个是 Transaction B，在第一个事务里面，它首先通过一个 where id=1 的条件查询一条数据，返回 name=Ada，age=16 的这条数据。然后第二个事务呢，它同样地是去操作 id=1 的这行数据，它通过一个 update 的语句，把这行 id=1 的数据的 age 改成了 18，但是大家注意，它没有提交。

这个时候，在第一个事务里面，它再次去执行相同的查询操作，发现数据发生了变化，获取到的数据 age 变成了 18。那么，这种在一个事务里面，由于其他的时候修改了数据并且没有提交，而导致了前后两次读取数据不一致的情况，这种事务并发的问题，我们把它定义成脏读。

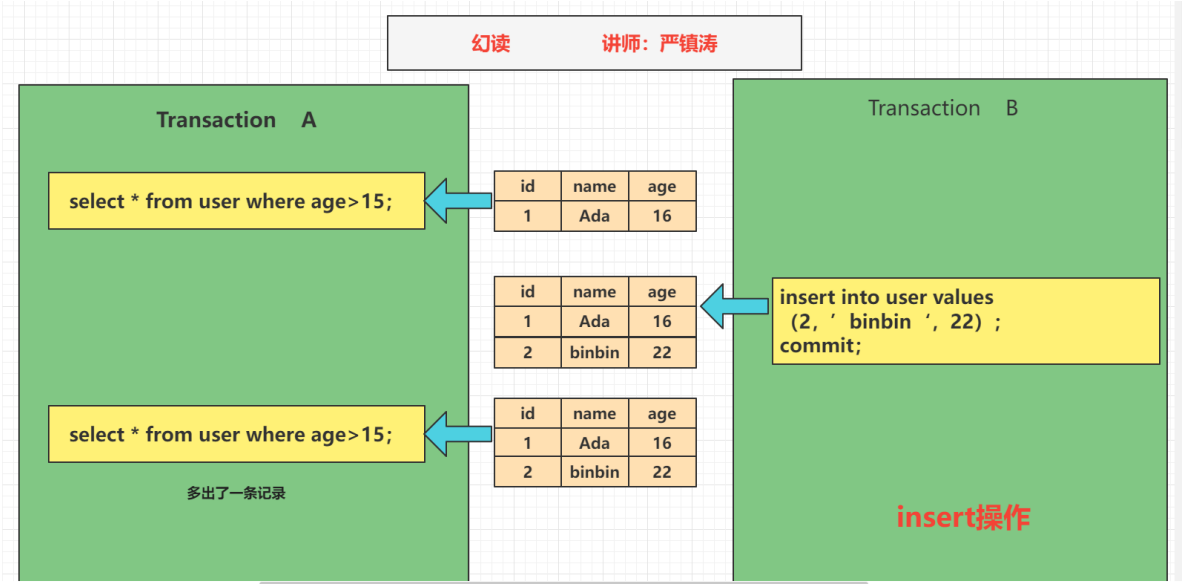
不可重复读



同样是两个事务，第一个事务通过 `id=1` 查询到了一条数据。然后在第二个事务里面执行了一个 `update` 操作，这里大家注意一下，执行了 `update` 以后它通过一个 `commit`提交了修改。然后第一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况，就像这里，`age` 到底是等于 `16` 还是 `18`，那么这种事务并发带来的问题，我们把它叫做不可重复读。

幻读

在第一个事务里面我们执行了一个范围查询，这个时候满足条件的数据只有一条。在第二个事务里面，它插入了一行数据，并且提交了。重点：插入了一行数据。在第一个事务里面再去查询的时候，它发现多了一行数据。



一个事务前后两次读取数据数据不一致，是由于其他事务插入数据造成的，这种情况我们把它叫做幻读。

总结：

不可重复读是修改或者删除，幻读是插入。

无论是脏读，还是不可重复读，还是幻读，它们都是数据库的读一致性的问题，都是在一个事务里面前后两次读取出现了不一致的情况。

如何解决数据的读一致性问题

两大方案:

LBCC

第一种，既然要保证前后两次读取数据一致，那么读取数据的时候，锁定我要操作的数据，不允许其他的事务修改就行了。这种方案叫做基于锁的并发控制 Lock Based Concurrency Control（LBCC）。

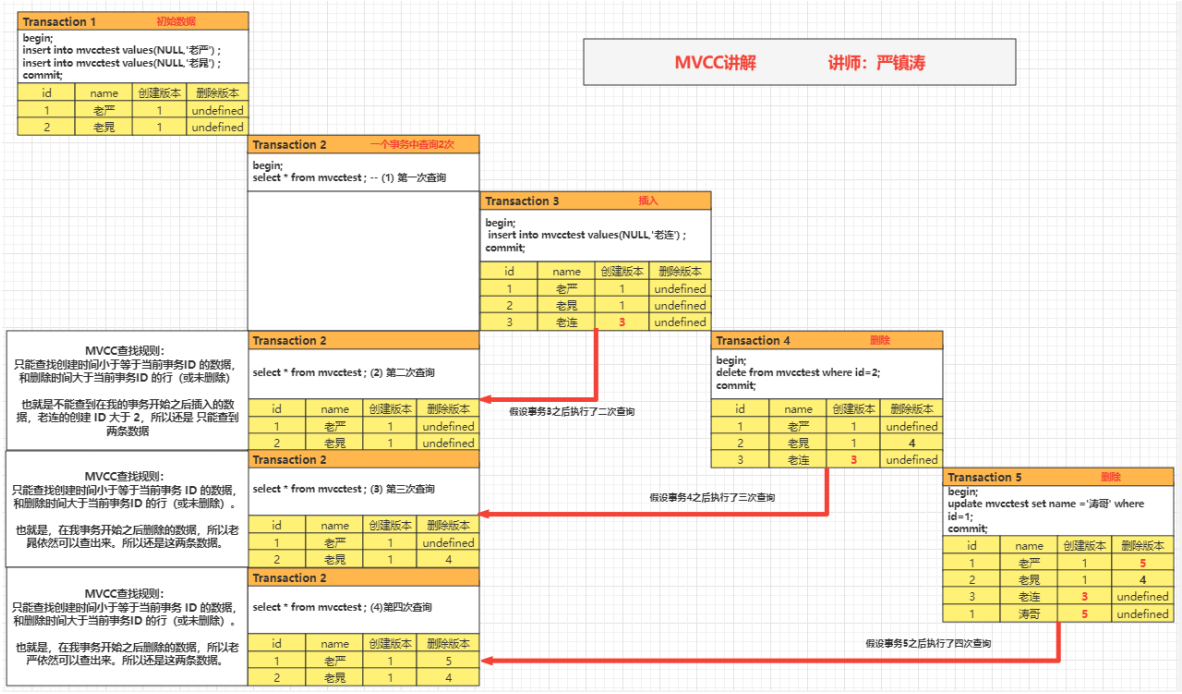
如果仅仅是基于锁来实现事务隔离，一个事务读取的时候不允许其他时候修改，那就意味着不支持并发的读写操作，而我们的大多数应用都是读多写少的，这样会极大地影响操作数据的效率。

MVCC

https://dev.mysql.com/doc/refman/5.7/en/innodb-multi-versioning.html

另一种解决方案，如果要让一个事务前后两次读取的数据保持一致，那么我们可以在修改数据的时候给它建立一个备份或者叫快照，后面再来读取这个快照就行了。这种方案我们叫做多版本的并发控制 Multi Version Concurrency Control（MVCC）

MVCC 的核心思想是： 我可以查到在我这个事务开始之前已经存在的数据，即使它在后面被修改或者删除了。在我这个事务之后新增的数据，我是查不到的。



通过以上演示我们能看到，通过版本号的控制，无论其他事务是插入、修改、删除，第一个事务查询到的数据都没有变化。

在 InnoDB 中，MVCC 是通过 Undo log 实现的。

Oracle、Postgres 等等其他数据库都有 MVCC 的实现。

需要注意，在 InnoDB 中，MVCC 和锁是协同使用的来实现隔离性的，这两种方案并不是互斥的。

第一大类解决方案是锁，锁又是怎么实现读一致性的呢？

MVCC为什么不能完全解决幻读

MVCC（多版本并发控制）通过快照读机制实现可重复读隔离级别，但其设计特性决定了无法彻底消除幻读问题，具体表现如下：

1. 快照读与当前读的隔离性差异

- **快照读**（如普通SELECT）基于事务开始时创建的Read View，仅读取历史版本数据，可屏蔽其他事务的插入操作。
- **当前读**（如带锁的SELECT FOR UPDATE、INSERT/UPDATE/DELETE操作）直接读取最新提交数据，若其他事务在两次查询之间插入新数据，仍会导致幻读。
- **示例**：事务A首次快照读未命中某条件，事务B插入符合该条件的数据并提交，事务A执行当前读时会读取到新数据，导致幻读。

2. 范围查询的边界漏洞

- MVCC对单行数据的版本控制有效，但在 **范围查询**（如 `WHERE id > 100`）场景中，事务无法感知其他事务在查询范围内插入的新行。
- **示例**：事务A查询 `id > 100` 的记录（快照读），事务B插入 `id=200` 并提交，事务A再次执行相同查询（当前读）会包含新插入的 `id=200`。

3. 插入操作无历史版本依赖

- MVCC依赖数据行的版本链回溯历史，但**新插入的行**没有历史版本，其他事务的插入操作会绕过MVCC的快照隔离机制。
- **示例**：事务A未开启间隙锁时，事务B插入新行并提交，事务A即使处于RR隔离级别，仍可能通过当前读感知到新行。

4. 隔离级别与锁机制的依赖

- 在MySQL的RR隔离级别下，**间隙锁（Gap Lock）** 才是彻底解决幻读的关键，而非MVCC本身。MVCC仅通过快照读减少锁冲突，但无法阻止其他事务插入新数据。
- **示例**：若事务A未对查询范围加间隙锁，事务B仍可插入新数据，导致事务A后续操作出现幻读。

10　20　30　40

出现幻读的案例：

案例一：快照读与当前读混合引发的幻读

场景复现

1. **事务A（可重复读隔离级别）**
   - **T1**：执行快照读 `SELECT * FROM book WHERE id > 100` → 结果为空（基于事务开始时的Read View）。
   - **T2**：事务B插入一条 `id=200` 的数据并提交。
   - **T3**：事务A执行当前读 `SELECT * FROM book WHERE id > 100 FOR UPDATE` → 结果包含 `id=200`（直接读取最新提交数据）。

结论

- MVCC通过快照读屏蔽了事务B插入的新数据（T1结果为空），但当前读（如 `FOR UPDATE`）会绕过MVCC，直接读取最新数据，导致事务A感知到新插入的 `id=200`，产生幻读。

案例二：无间隙锁保护的范围查询

场景复现

1. **事务A（可重复读隔离级别）**
   - **T1**：快照读 `SELECT * FROM book WHERE id BETWEEN 100 AND 300` → 结果为空。
   - **T2**：事务B插入 `id=150` 并提交。
   - **T3**：事务A执行更新操作 `UPDATE book SET name='test' WHERE id BETWEEN 100 AND 300` → 更新语句触发当前读，实际修改了事务B插入的 `id=150`。

=**结论**=

- MVCC在快照读时未感知新插入的 `id=150`，但事务A后续的更新操作（当前读）会操作新数据，导致逻辑冲突。若事务A未对查询范围加间隙锁，MVCC无法阻止新数据的插入。

案例三：插入操作绕过MVCC版本链

场景复现

1. **事务A（可重复读隔离级别）**
   - **T1**：快照读 `SELECT * FROM book WHERE name='java'` → 结果为空。
   - **T2**：事务B插入 `name='java'` 的新记录并提交。
   - **T3**：事务A执行插入操作 `INSERT INTO book (name) VALUES ('java')` → 触发主键冲突（因事务B已插入相同数据）。

结论

- MVCC的快照读未感知事务B的插入操作，但事务A的插入操作需校验唯一约束（如主键），此时直接读取最新数据，发现冲突。新插入数据无历史版本，MVCC无法通过版本链回溯规避此问题。

结论：

MVCC通过快照读解决了不可重复读和部分幻读问题，但**无法完全消除幻读的根本原因**在于：

1. **当前读与快照读的混合使用**导致数据可见性差异；
2. **新插入数据无历史版本**，直接绕过MVCC隔离机制；
3. **范围查询的天然漏洞**=需依赖锁机制（如间隙锁）弥补。

实际生产中，需结合 **间隙锁+MVCC**=（如MySQL RR隔离级别）或提升至**串行化隔离级别**，才能完全规避幻读

如何分析执行计划

https://dev.mysql.com/doc/refman/5.7/en/explain-output.html

我们先创建三张表。一张课程表，一张老师表，一张老师联系方式表（没有任何索引）。

我们先创建三张表。一张课程表，一张老师表，一张老师联系方式表（没有任何索引）。

```
DROP TABLE
IF
    EXISTS course;

CREATE TABLE `course` ( `cid` INT ( 3 ) DEFAULT NULL, `cname` VARCHAR ( 20 )
DEFAULT NULL, `tid` INT ( 3 ) DEFAULT NULL ) ENGINE = INNODB DEFAULT CHARSET =
utf8mb4;

DROP TABLE
IF
    EXISTS teacher;

CREATE TABLE `teacher` ( `tid` INT ( 3 ) DEFAULT NULL, `tname` VARCHAR ( 20 )
DEFAULT NULL, `tcid` INT ( 3 ) DEFAULT NULL ) ENGINE = INNODB DEFAULT CHARSET =
utf8mb4;

DROP TABLE
IF
    EXISTS teacher_contact;

CREATE TABLE `teacher_contact` ( `tcid` INT ( 3 ) DEFAULT NULL, `phone` VARCHAR
( 200 ) DEFAULT NULL ) ENGINE = INNODB DEFAULT CHARSET = utf8mb4;

INSERT INTO `course`
VALUES
    ( '1', 'mysql', '1' );

INSERT INTO `course`
VALUES
    ( '2', 'jvm', '1' );

INSERT INTO `course`
VALUES
    ( '3', 'juc', '2' );

INSERT INTO `course`
VALUES
    ( '4', 'spring', '3' );
```

```
INSERT INTO `teacher`
VALUES
    ( '1', 'bobo', '1' );

INSERT INTO `teacher`
VALUES
    ( '2', '老严', '2' );

INSERT INTO `teacher`
VALUES
    ( '3', 'dahai', '3' );

INSERT INTO `teacher_contact`
VALUES
    ( '1', '13688888888' );

INSERT INTO `teacher_contact`
VALUES
    ( '2', '18166669999' );

INSERT INTO `teacher_contact`
VALUES
    ( '3', '17722225555' );
```

explain 的结果有很多的字段，我们详细地分析一下。

先确认一下环境：

```
select version();
show variables like '%engine%';
```

4.3.1 id

id 是查询序列编号。

id 值不同

id 值不同的时候，先查询 id 值大的（先大后小）。

```
-- 查询 mysql 课程的老师手机号
EXPLAIN SELECT
    tc.phone
FROM
    teacher_contact tc
WHERE
    tcid = ( SELECT tcid FROM teacher t WHERE t.tid = ( SELECT c.tid FROM course
c WHERE c.cname = 'mysql' ) );
```

查询顺序：course c--teacher t--teacher_contact tc。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null	(Null)	(Nu	6	16.67	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where

先查课程表，再查老师表，最后查老师联系方式表。子查询只能以这种方式进行，只有拿到内层的结果之后才能进行外层的查询。

id 值相同（从上往下）

```
-- 查询课程 ID 为 2，或者联系表 ID 为 3 的老师
EXPLAIN SELECT
    t.tname,
    c.cname,
    tc.phone
FROM
    teacher t,
    course c,
    teacher_contact tc
WHERE
    t.tid = c.tid
    AND t.tcid = tc.tcid
    AND ( c.cid = 2 OR tc.tcid = 3 );
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	100	(Null)
1	SIMPLE	c	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where; Using join buffer
1	SIMPLE	tc	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	33.33	Using where; Using join buffer

id 值相同时，表的查询顺序是

从上往下顺序执行。例如这次查询的 id 都是 1，查询的顺序是 teacher t（3 条）——course c（4 条）——teacher_contact tc（3 条）。

既有相同也有不同

如果 ID 有相同也有不同，就是 ID 不同的先大后小，ID 相同的从上往下。

4.3.2 select type 查询类型

这里并没有列举全部（其它：DEPENDENT UNION、DEPENDENT SUBQUERY、MATERIALIZED、UNCACHEABLE SUBQUERY、UNCACHEABLE UNION）。

下面列举了一些常见的查询类型：

SIMPLE

简单查询，不包含子查询，不包含关联查询 union。

```
EXPLAIN SELECT * FROM teacher;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	teacher	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	100	(Null)

再看一个包含子查询的案例：

```
-- 查询 mysql 课程的老师手机号
EXPLAIN SELECT
    tc.phone
FROM
    teacher_contact tc
WHERE
    tcid = ( SELECT tcid FROM teacher t WHERE t.tid = ( SELECT c.tid FROM course
c WHERE c.cname = 'mysql' ) );
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	33.33	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where

PRIMARY

子查询 SQL 语句中的主查询，也就是最外面的那层查询。

SUBQUERY

子查询中所有的内层查询都是 SUBQUERY 类型的。

DERIVED

衍生查询，表示在得到最终查询结果之前会用到临时表。例如：

```
-- 查询 ID 为 1 或 2 的老师教授的课程
EXPLAIN SELECT
    cr.cname
FROM
    ( SELECT * FROM course WHERE tid = 1 UNION SELECT * FROM course WHERE tid =
2 ) cr;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	100	(Null)
2	DERIVED	course	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where
3	UNION	course	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where
(Nu	UNION RESULT	<union2,3>	(Null)	ALL	(Null)	(Null	(Null)	(Nu	(Null)	(Null)	Using temporary

对于关联查询，先执行右边的 table（UNION），再执行左边的 table，类型是DERIVED

UNION

用到了 UNION 查询。同上例。

UNION RESULT

主要是显示哪些表之间存在 UNION 查询。<union2,3>代表 id=2 和 id=3 的查询存在 UNION。同上例。

4.3.3 type 连接类型

https://dev.mysql.com/doc/refman/5.7/en/explain-output.html#explain-join-types

所有的连接类型中，上面的最好，越往下越差。

在常用的链接类型中：system > const > eq_ref > ref > range > index > all

这里并没有列举全部（其他：fulltext、ref_or_null、index_merger、unique_subquery、index_subquery）。

以上访问类型除了 all，都能用到索引。

const

主键索引或者唯一索引，只能查到一条数据的 SQL。

```
DROP TABLE
IF
    EXISTS single_data;
CREATE TABLE single_data ( id INT ( 3 ) PRIMARY KEY, content VARCHAR ( 20 ) );
INSERT INTO single_data
VALUES
    ( 1, 'a' );
EXPLAIN SELECT
    *
FROM
    single_data a
WHERE
    id = 1;
```

system

system 是 const 的一种特例，只有一行满足条件。例如：只有一条数据的系统表。

```
EXPLAIN SELECT * FROM mysql.proxies_priv;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	proxies_priv	(Null)	system	(Null)	(Null	(Null)	(Nul	1	100

eq_ref

通常出现在多表的 join 查询，表示对于前表的每一个结果,，都只能匹配到后表的一行结果。一般是唯一性索引的查询（UNIQUE 或 PRIMARY KEY）。

eq_ref 是除 const 之外最好的访问类型。

先删除 teacher 表中多余的数据，teacher_contact 有 3 条数据，teacher 表有 3条数据。

```
DELETE
```

```
FROM
    teacher
WHERE
    tid IN ( 4, 5, 6 );
COMMIT;
-- 备份
INSERT INTO `teacher`
VALUES
    ( 4, '老严', 4 );
INSERT INTO `teacher`
VALUES
    ( 5, 'bobo', 5 );
INSERT INTO `teacher`
VALUES
    ( 6, 'seven', 6 );
COMMIT;
```

为 teacher_contact 表的 tcid（第一个字段）创建主键索引。

```
-- ALTER TABLE teacher_contact DROP PRIMARY KEY;
ALTER TABLE teacher_contact ADD PRIMARY KEY(tcid);
```

为 teacher 表的 tcid（第三个字段）创建普通索引。

```
-- ALTER TABLE teacher DROP INDEX idx_tcid;
ALTER TABLE teacher ADD INDEX idx_tcid (tcid);
```

执行以下 SQL 语句：

```
select t.tcid from teacher t,teacher_contact tc where t.tcid = tc.tcid;
```

tcid
1
2
3

此时的执行计划（teacher_contact 表是 eq_ref）：

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
SIMPLE	t	(Null)	index	idx_tcid	idx_tcic	5	(Null)	3
SIMPLE	tc	(Null)	eq_ref	PRIMARY	PRIMAI	4	gupao	1

小结：

以上三种 system，const，eq_ref，都是可遇而不可求的，基本上很难优化到这个状态。

ref

查询用到了非唯一性索引，或者关联操作只使用了索引的最左前缀。

例如：使用 tcid 上的普通索引查询：

```
explain SELECT * FROM teacher where tcid = 3;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	teacher	(Null)	ref	idx_tid	idx_tid	5	const	1	100

range

索引范围扫描。

如果 where 后面是 between and 或 <或 > 或 >= 或 <=或 in 这些，type 类型就为 range。

不走索引一定是全表扫描（ALL），所以先加上普通索引。

```
-- ALTER TABLE teacher DROP INDEX idx_tid;
ALTER TABLE teacher ADD INDEX idx_tid (tid);
```

执行范围查询（字段上有普通索引）：

```
EXPLAIN SELECT * FROM teacher t WHERE t.tid <3;
-- 或
EXPLAIN SELECT * FROM teacher t WHERE tid BETWEEN 1 AND 2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	t	(Null)	range	idx_tid	idx_tid	5

IN 查询也是 range（字段有主键索引）

```
EXPLAIN SELECT * FROM teacher_contact t WHERE tcid in (1,2,3);
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	t	(Null)	range	PRIMARY	PRIMARY	4

index

Full Index Scan，查询全部索引中的数据（比不走索引要快）。

```
EXPLAIN SELECT tid FROM teacher;
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	teacher	(Null)	index	(Null)	idx_tid	5

all

Full Table Scan，如果没有索引或者没有用到索引，type 就是 ALL。代表全表扫描。

小结：

一般来说，需要保证查询至少达到 range 级别，最好能达到 ref。

ALL（全表扫描）和 index（查询全部索引）都是需要优化的。

4.3.4 possible_key、key

可能用到的索引和实际用到的索引。如果是 NULL 就代表没有用到索引。

possible_key 可以有一个或者多个，可能用到索引不代表一定用到索引。

反过来，possible_key 为空，key 可能有值吗？

表上创建联合索引：

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone (name,phone);
```

执行计划（改成 select name 也能用到索引）：

```
explain select phone from user_innodb where phone='126';
```

id	select_type	table	type	possible_keys	key
1	SIMPLE	user_innodb	index	(Null)	comidx_name_phone

结论：是有可能的（这里是覆盖索引的情况）。

如果通过分析发现没有用到索引，就要检查 SQL 或者创建索引。

4.3.5 key_len

索引的长度（使用的字节数）。跟索引字段的类型、长度有关。

表上有联合索引：KEY

`comidx_name_phone(name,phone)`

```
explain select * from user_innodb where name ='jim';
```

4.3.6 rows

MySQL 认为扫描多少行才能返回请求的数据，是一个预估值。一般来说行数越少越好。

4.3.7 filtered

这个字段表示存储引擎返回的数据在 server 层过滤后，剩下多少满足查询的记录数量的比例，它是一个百分比。

4.3.8 ref

使用哪个列或者常数和索引一起从表中筛选数据。

4.3.9 Extra

执行计划给出的额外的信息说明。

using index

用到了覆盖索引，不需要回表。

```
EXPLAIN SELECT tid FROM teacher ;
```

using where

使用了 where 过滤，表示存储引擎返回的记录并不是所有的都满足查询条件，需要在 server 层进行过滤（跟是否使用索引没有关系）。

```
EXPLAIN select * from user_innodb where phone ='13866667777';
```

id	select_type	table	type	possible_keys	key
1	SIMPLE	user_innodb	index	(Null)	comidx_name_phone

using filesort

不能使用索引来排序，用到了额外的排序（跟磁盘或文件没有关系）。需要优化。（复合索引的前提）

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone (name,phone);
```

```
EXPLAIN select * from user_innodb where name ='jim' order by id;
```

（order by id 引起）

select_type	table	p	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	user_innodb		ref	comidx_name_phone	comidx	1023	const	1	100	Using index condition;Using filesort

using temporary

用到了临时表。例如（以下不是全部的情况）：

1、distinct 非索引列

```
EXPLAIN select DISTINCT(tid) from teacher t;
```

```
2、group by 非索引列
```

```
EXPLAIN select tname from teacher group by tname;
```

```
3、使用 join 的时候，group 任意列
```

```
EXPLAIN select t.tid from teacher t join course c on t.tid = c.tid group by
t.tid;
```

```
需要优化，例如创建复合索引。
```

总结一下：

模拟优化器执行 SQL 查询语句的过程，来知道 MySQL 是怎么处理一条 SQL 语句的。通过这种方式我们可以分析语句或者表的性能瓶颈。

分析出问题之后，就是对 SQL 语句的具体优化。

数据库死锁怎么办，你会怎么处理?

数据库死锁是多个事务因互相等待对方释放资源而无法继续执行的阻塞现象。以下是常见死锁场景及处理方案：

一、常见死锁场景

1. **交叉资源访问顺序**
   - **场景**：事务A先锁资源X，再请求资源Y；事务B先锁资源Y，再请求资源X。
   - **原因**：事务对资源的加锁顺序不一致。
2. **长事务占用资源**
   - **场景**：事务A长时间持有锁未提交，事务B等待该锁时自身持有的锁也被其他事务等待。
   - **原因**：未及时提交/回滚事务，导致锁长期占用。
3. **批量操作锁升级**
   - **场景**：批量更新/删除未走索引，导致行锁升级为表锁，阻塞其他事务。
   - **原因**：索引缺失或SQL优化不足，引发全表扫描。
4. **间隙锁（Gap Lock）冲突**
   - **场景**：事务A锁定某范围的间隙，事务B尝试插入该范围内的数据。
   - **常见于**：MySQL的 `REPEATABLE READ` 隔离级别。

二、死锁处理方案

1. 自动处理机制

- **死锁检测**
  数据库（如InnoDB）自动检测死锁，强制回滚代价较小的事务。

```
SHOW ENGINE INNODB STATUS; -- 查看MySQL死锁日志
```

- **锁超时设置**
  设置事务等待锁的超时时间，超时后自动回滚。

```
SET innodb_lock_wait_timeout = 30; -- MySQL设置锁超时时间（秒）
```

2. 代码与设计优化

- **统一资源访问顺序**
  确保所有事务按固定顺序访问资源（如按主键排序更新）。
- **短事务原则**
  减少事务执行时间，尽快提交或回滚，避免长事务占用锁。
- **优化SQL与索引**
  - 为查询条件添加索引，避免全表扫描。
  - 避免批量操作锁大量数据，分批处理。
  - 使用 `SELECT ... FOR UPDATE` 时明确指定索引。
- **降低隔离级别**
  使用 `READ COMMITTED` 代替 `REPEATABLE READ`，减少间隙锁的使用（需权衡数据一致性）。

3. 程序容错机制

- **重试策略**
  捕获死锁异常（如MySQL的 `1213` 错误码），在代码层重试事务。

```
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Service;

@Service
public class AccountService {

    @Retryable(
        value = {DeadlockLoserDataAccessException.class},
        maxAttempts = 3,
        backoff = @Backoff(delay = 1000, multiplier = 2)
    )
    @Transactional
    public void transferWithRetry(int fromId, int toId, BigDecimal amount) {
        // 转账业务逻辑
        jdbcTemplate.update("UPDATE account SET balance = balance - ? WHERE
id = ?", amount, fromId);
        jdbcTemplate.update("UPDATE account SET balance = balance + ? WHERE
id = ?", amount, toId);
    }
}
```

- **乐观锁**
  通过版本号或时间戳避免显式加锁（适用于冲突较少的场景）。

```
UPDATE table SET col = new_val, version = version + 1
WHERE id = 1 AND version = old_version;
```

三、排查工具

1. **数据库日志**
   - MySQL：`SHOW ENGINE INNODB STATUS`
   - PostgreSQL：`pg_locks`视图 + `deadlock_timeout`参数
2. **监控工具**
   - Prometheus + Grafana监控锁等待。
   - 第三方工具（如Percona Toolkit、pt-deadlock-logger）。

总结

预防死锁的核心是规范资源访问顺序、优化事务设计。处理时优先依赖数据库的自动回滚机制，并通过日志分析根因。在代码层添加重试逻辑和监控，可显著降低死锁对业务的影响。

MySQL InnoDB 锁的基本类型

官网：https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html

锁的基本模式——共享锁

第一个行级别的锁就是我们在官网看到的 Shared Locks （共享锁），我们获取了一行数据的读锁以后，可以用来读取数据，所以它也叫做读锁。而且多个事务可以共享一把读锁。那怎么给一行数据加上读锁呢？

我们可以用 select lock in share mode;的方式手工加上一把读锁。

释放锁有两种方式，只要事务结束，锁就会自动事务，包括提交事务和结束事务。

锁的基本模式——排它锁

第二个行级别的锁叫做 Exclusive Locks（排它锁），它是用来操作数据的，所以又叫做写锁。只要一个事务获取了一行数据的排它锁，其他的事务就不能再获取这一行数据的共享锁和排它锁。

排它锁的加锁方式有两种，第一种是自动加排他锁，可能是同学们没有注意到的：

我们在操作数据的时候，包括增删改，都会默认加上一个排它锁。

还有一种是手工加锁，我们用一个 FOR UPDATE 给一行数据加上一个排它锁，这个无论是在我们的代码里面还是操作数据的工具里面，都比较常用。

释放锁的方式跟前面是一样的。

锁的基本模式——意向锁

意向锁是由数据库自己维护的。

也就是说，当我们给一行数据加上共享锁之前，会自动在这张表上面加一个意向共享锁。

当我们给一行数据加上排他锁之前，会自动在这张表上面加一个意向排他锁。

反过来说：

如果一张表上面至少有一个意向共享锁，说明有其他的事务给其中的某些数据行加上了共享锁。

如果一张表上面至少有一个意向排他锁，说明有其他的事务给其中的某些数据行加上了排他锁。

那么这两个表级别的锁存在的意义是什么呢？第一个，我们有了表级别的锁，在 `InnoDB` 里面就可以支持更多粒度的锁。它的第二个作用，我们想一下，如果说没有意向锁的话，当我们准备给一张表加上表锁的时候，我们首先要做什么？是不是必须先要去判断有没其他的事务锁定了其中了某些行？如果有的话，肯定不能加上表锁。那么这个时候我们就要去扫描整张表才能确定能不能成功加上一个表锁，如果数据量特别大，比如有上千万的数据的时候，加表锁的效率是不是很低？

但是我们引入了意向锁之后就不一样了。我只要判断这张表上面有没有意向锁，如果有，就直接返回失败。如果没有，就可以加锁成功。所以 `InnoDB` 里面的表锁，我们可以把它理解成一个标志。就像火车上厕所有没有人使用的灯，是用来提高加锁的效率的。

mysql面试的时候怎么去聊　1天基本的原理　案例　原理

mysql性能优化　中间件　标准　SQL标准 不能执行时间超过1S　业务结合数据库进行的

延迟关联　游标分页　索引分析　覆盖索引