数据库笔记

BV1X64y1x7HT - over

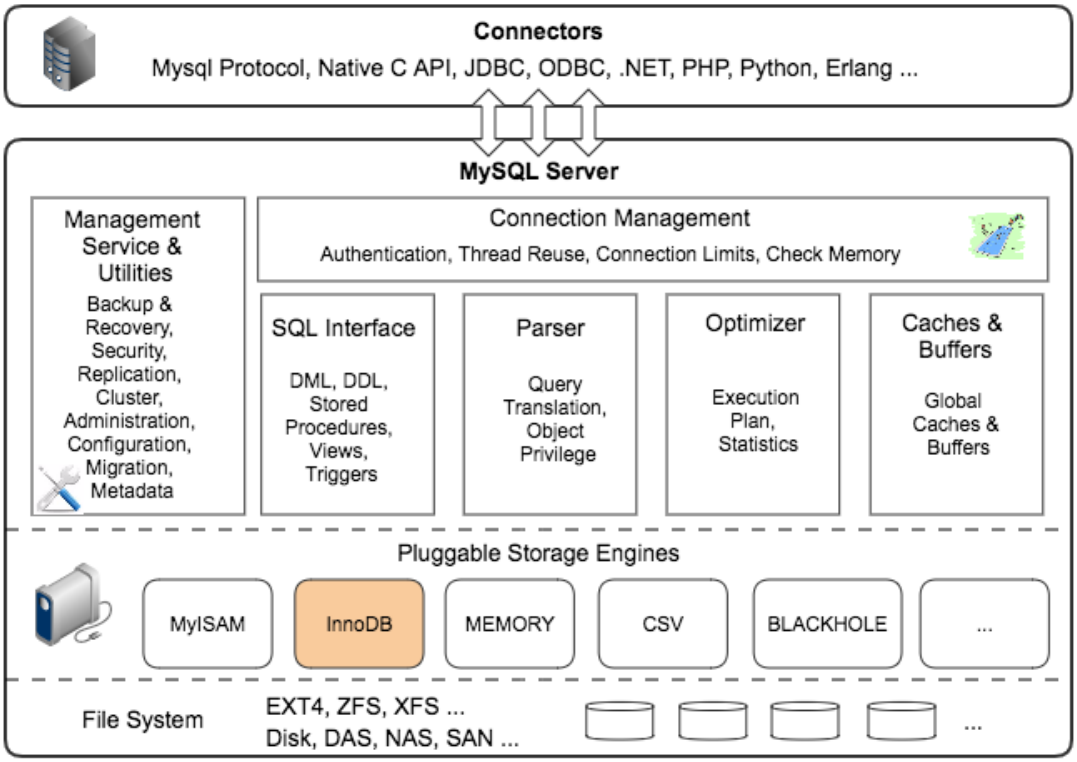
锁+主从复制等

BV1J5411A7ei

BV1eD4y1D7pR

MySQL架构

参考： **MySQL 性能优化**



存储引擎：MyISAM, InnoDB，支持的引擎可以使用`show engines`查看。

数据库后缀名为`MYI` -> MyISAM，`ibd` -> InnoDB

存储引擎	优点	缺点
InnoDB	5.5版本后MySQL默认数据库，支持事务，比MyISAM处理速度稍慢	非常复杂，性能较一些简单的引擎要差一点儿。空间占用比较多。
MyISAM	高速引擎，拥有极高的插入，查询速度	不支持事务，不支持行锁、崩溃后数据不容易修复
Archive	将数据压缩后存储，非常适合存储大量的独立的，作为历史记录的数据	只能进行插入和查询操作，非事务型
CSV	是基于CSV格式文件存储数据（应用于跨平台数据交换）	
Memory	内存存储引擎，拥有极高的插入，更新和查询效率，适用于临时表	占用和数据量成正比的内存空间，只在内存上保存数据，意味着数据可能会丢失，并发能力低下。不支持BLOB或TEXT类型的列
Falcon	一种新的存储引擎，支持事务处理，传言可能是InnoDB的替代者	

SQL基础

普通查询语句

🔘普通查询：

```
select A,B,C,D from t1,t2,t3 where p;
```

🔘去重查询：

```
select distinct A,B,C,D from t1,t2,t3 where p;
```

🔘字符串运算：

`%` 匹配任意字符（`NULL` 除外），`_` 匹配一个字符

排序：

```
select A,B,C,D from t1,t2,t3 where p order by A desc, B asc;    -- 默认升序
```

🔘限制行数：

```
select A,B,C,D from t1,t2,t3 where p limit m,n; -- 查询从第m行开始的n行数据
select A,B,C,D from t1,t2,t3 where p limit n offset m;
-- 注意 limit 1,1 为第二行
```

🔘范围值查询：

```
select * from customer where age between 10 and 20;
```

🔘空值检查：

❗注意：判断空值不能使用 `col = NULL`，而应该是 `col is NULL`

```
select * from toys where price is NULL;
```

逻辑操作符：`OR`，`AND`，`NOT`

范围操作符：`IN`，`LIKE`

🔘正则表达式匹配：

```
select * from customer where tel regexp ".*@qq\\.com";   -- 相当于like的正则表达式版
select * from customer where name binary regexp "Lisa[a-z]{1,3}";   -- 区分大小写的正则
```

🔘拼接字段：`concat()`

```
select concat(name,':', tel, '.') as info from customer where p
-- ————————————
--|    info    |
-- ————————————
--| John:53477.|
-- ————————————
```

🔘其他字符串函数：

- 删除空格：两边空格 `Trim(col)`，左边空格 `LTrim(col)`，右边空格 `RTrim(col)`
- 字母大小写：`Upper(col)` `Lower(col)`
- 返回字符串长度：`Length(col)`，对应长度为len的字符串 `Substring(col, pos, len)` 或者 `Substring(col, pos)`，这里的 `pos` 从1开始是第一个字符。

🔘使用子查询

> 多个select语句嵌套

举例：

```
select a from t1 where b in (select b from t2 where c = 'x'); -- 子查询过滤
select a (select count(*) from t2 where t1.a = t2.a) from t1; -- 子查询字段
```

列必须匹配：外围`where`语句中的列数目必须要和子查询中选择的列要相同，类型要兼容。

组合查询：

> 俗称交、并、差（MySQL不直接支持后两种）

并（`union`，`union all`）：

```
select a, b from t1 union select a, b from t2;      -- 自动去重
select a, b from t1 union all select a, b from t2;  -- 不去重
```

交：

```
select a, b from t1 inner join t2 using (a, b);     -- 保证这n列的内容相同即可
-- 等价于
select a, b from t1 inner join t2 on t1.a = t2.a and t1.b = t2.b;
```

差：

```
select a, b from t1 left join t2 using (a,b) where t2.col is NULL; -- a-b
```

聚集过滤

常见的聚集函数`MAX()` `MIN()` `AVG()` `SUM()`，都不将`NULL`值记录在内，`count(col)`也会忽略`NULL`值，`count(*)`除外。

分组过滤：

> 未出现在`group by`子句中的属性，在select子句中必须由聚集函数包围。

```
select age, avg(b) from users group by age; -- 字段b被聚集函数包围
select age, avg(b) from users group by age having age > 20;   -- group by 字段要使用having过滤
select age, avg(b) from users where b > 3 group by age having age > 20 order by age;
```

> `group by`子句必须出现在`where`子句之后，在`order by`子句之前！`where...group by...order by...`
>
> - `where`是在分组前过滤，`having`是在分组后过滤。
> - `order by`是对输出后的结果进行排序，而`group by`不对数据进行排序。

表联结(join)

> 有时候联结要比子查询快得多

内部联结：

```
select * from users inner join commodity on user.id = commodity.id;
```

外部联结：

```
-- left outer join 可简写为 left join
select * from users left join commodity on user.id = commodity.id;
select * from users right join commodity on user.id = commodity.id;
-- 等价于using,但是using在会自动省略相同的列
select * from users right join commodity using(id);
```

更改数据(增删改)

insert：

```
insert into t1 values (NULL, 'a', 1, true);     -- 方便但不安全，不能保证表结构改变后也是这个次序
insert into t1 (name, age, is_married) values ('a', 1, true);   -- 安全明确但繁琐
-- 一般不要使用没有明确列的insert的语句
```

insert插入多值：

```
-- 单条insert多数据插入要比多个单数据insert效率高
insert into t1 (name, age, is_married) values ('a', 1, true),
                                              ('b', 2, false),
                                              ('c', 3, false);
-- insert与select配合
insert into t1 (a, b, c) select (d, e, f) from t2 where p;
```

update:

```
update t1 set col='a' where p;
update ignore t1 set col='a' where p;   -- update过程中发生错误也继续执行。
-- update 支持case结构
update user set balance = case
      when age < 21 then balance + 100000
      when age >= 21 then balance * 2
      else balance + 9.99
      end
```

delete:

```
delete from t1 where p;
truncate t1;            -- 清空；实际上是删除表再重新创建一个表
```

创建和操作表

创建：

```
create table if not exists t1(
     id int not null auto_increment,                                    -- 设置自增
     name varchar(255) not null,
     age int not null check (age > 0),                                  -- 设置check语句
     vendor_id int not null,
     quantity int not null default 1,
     has_commodity bool null,                                           -- 可空
     update_time timestamp not null default CURRENT_TIMESTAMP,          -- 设置默认值
     primary key(id),                                                   -- 设置主键
     constraint vid_fk foreign key (vendor_id) references t2(id)        -- 设置外键
     index(name)                                                        -- 设置索引
)engine=innodb charset=utf8mb4 comment 'tl commment';                   -- 设置引擎和字符编码
```

更改表结构：

可参考： **MySQL ALTER TABLE Statement**

```
alter table t2 rename r1;                                          -- 重命名表
alter table t1 add name varchar(255);                              -- 增加一列
alter table t1 drop column name;                                   -- 删除一列
-- modify 列名 类型， change 旧列名 新列名 类型
alter table t1 modify name varchar(128), change age years int;     -- 更改列类型
alter table t1 add primary key (id);                               -- 添加主键
alter table t1 drop primary key;                                   -- 删除主键
alter table t1 add index(name),add unique(card_id);                -- 添加普通索引和唯一索引
alter table t1 drop index name                                     -- 删除索引;
-- 添加外键
alter table t1 add constraint vid_fk foreign key(vid) references vendor(id) on delete [options];
alter table t1 drop constraint vid_fk;                             -- 删除外键
```

删除表：

```
drop table t1;
```

重命名表：

```
rename table t1 to t2;
```

约束

🔘check子句：

```
create table if not exists t1(
     id int not null auto_increment,                               -- 设置自增
     name varchar(255) not null,
     age int not null check (age > 0)                              -- 设置check语句
)engine=innodb charset=utf8mb4;
```

🔘外键约束：

```
alter table t1 add constraint vid_fk foreign key(vid) references vendor(id) on delete [options];
```

这里的options包含： （MySQL默认为`restrict`或者`no action`）

- `cascade`：父表update/delete记录时，同步update/delete掉子表的匹配记录
- `set null`：父表update/delete记录时，将子表上匹配记录的列设为null (子表外键列不为not null)
- `no action`：如果子表中有匹配的记录，停止对父表对应候选键进行update/delete操作
- `restrict`：同no action, 都是立即检查外键约束

视图

优点：

- 重用SQL语句，简化复杂的SQL操作。
- 使用表的一部分数据，而不是整个表，可以保护数据。

操作视图：

```
-- 创建视图
create view uc_view as select * from users inner join commodity on user.id = commodity.id;
drop view uc_view;                                  -- 删除视图
create or replace view uc_view as select ...;       -- 不存在则创建，存在则替换
show create view uc_view;                           -- 展示视图
```

查询操作类似普通表的查询。

> 视图一般用于检索，不用于增删改。

存储过程

> 相当于函数，以后再填坑

触发器

使用场景：

- 检查用户电话格式是否正确，地址字符是否大写
- 每订购一个产品，从仓库中减少一个产品
- 删除用户时，再存档表中增加一个记录等

支持触发器的语句：`update`，`insert`，`delete`

创建触发器的4个条件：

- 唯一的触发器名字
- 触发器关联的表
- 触发器响应的活动`update`，`insert`，`delete`
- 触发器执行的时机`before`，`after`

> 每个表最多支持6个触发器，

```
-- 创建触发器，在插入表t1之后显示当前时间
create trigger insert_trigger after insert on t1 for each row select now() into @param;
select @param;       -- 显示insert记录后的结构,MySQL5之后不允许返回结果集
-- 删除触发器
drop trigger insert_trigger
-- 使用begin end包围
create trigger del_trigger before delete on orders o for each row
begin
    insert into archives(name, age, balance) values (o.name, o.age, o.balance)
end;
-- before update
create trigger up_tri before update on orders o for each row set o.name = Upper(o.name);
```

全局变量

```
show character set;                          -- 展示支持的所有字符
show variables like 'character%';            -- 查看默认的字符集

show global status like "Innodb_page_size"  -- 查看数据库B+树节点支持的大小
show variables like 'autocommit';            -- 是看是否提交事务
show engine innodb status;                   -- 可以查看锁的状态
select @@transaction_isolation;              -- 查看数据库当前隔离级别
```

账户安全管理

◍查看当前数据库的所有用户：

```
use mysql;
select user from user;
```

◍操作用户：

```
create user 'jack'@'host' identified by 'passwd';       -- 创建用户, % 表示可以任意主机
drop user jack;                                         -- 删除用户
```

◍授权：

```
show grants for root;                              -- 查看授权给root的信息
grant select on demo_db.demo_table to jack;        -- 授予jack在demo_db数据库demo_table上查询权限
grant all on demo_db.* to 'jack'@'%';              -- 授予jack在demo_db数据库上所有权限
grant all on *.* to 'jack'@'%';                    -- 授予jack所有权限
revoke all on *.* to 'jack'@'%';                   -- 撤销jack所有权限
SET PASSWORD FOR 'username'@'host' = PASSWORD('newpassword');   -- 更改用户密码
```

导入与导出

◍导入sql文件：

```
mysql -uroot -p -D customer < userinfo.sql      -- 向customer数据库导入表名为userinfo的数据
```

◍导入csv：

```
load data infile '/usr/local/user.csv'  -- CSV文件存放路径
into table student                      -- 要将数据导入的表名
fields terminated by ',' optionally enclosed by '"' escaped by '"'
lines terminated by '\r\n';
```

执行计划

explain关键字。

```
explain select * from user;
```

输出：

EXPLAIN Output Format

```
id, select_type, `table`, partitions, type,  possible_keys, `key`, key_len, ref, rows, filtered, Extra
1, 'SIMPLE',       'user',  null,          'ALL', null,                null, null,     null, 3,    100,
null
```

`type`：具体的值可以是`system`，`const`，`ref`，`range`，`index`，`all`。从左到右效率逐次降低。`all`代表全表扫描，效率很低，要避免出现`all`。

`key`：选择使用到的索引。

`Extra`：其他的关键信息。

索引

索引和实际的数据都是存储在磁盘中，在进行数据读取的时候会优先将索引读取到内存当中。

导入数据的时候不应该使用索引，应用所有数据索引的总时间 < 导入每行数据分别进行添加索引总时间

数据结构

索引的数据结构：二叉树、红黑树、Hash表、B树、**B+树**。

非B树存储当数据变多的时候树会变得非常高。

hash表无法进行范围查询，否则要进行挨个查询。

为什么使用B+树而不使用B树?

- B+树的磁盘读写代价更小。B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，通常B+树矮更胖，高度小查询产生的I/O更少。
- B+树查询效率更高。B+树使用双向链表串连所有叶子节点，区间查询效率更高（因为所有数据都在B+树的叶子节点，扫描数据库 只需扫一遍叶子结点就行了），但是B树则需要通过中序遍历才能完成查询范围的查找。
- B+树查询效率更稳定。B+树每次都必须查询到叶子节点才能找到数据，而B树查询的数据可能不在叶子节点，也可能在，这样就会造成查询的效率的不稳定。

如果B+树3-4层都存储满索引，数据量起码有千万级。

MySQL中的B+树索引：

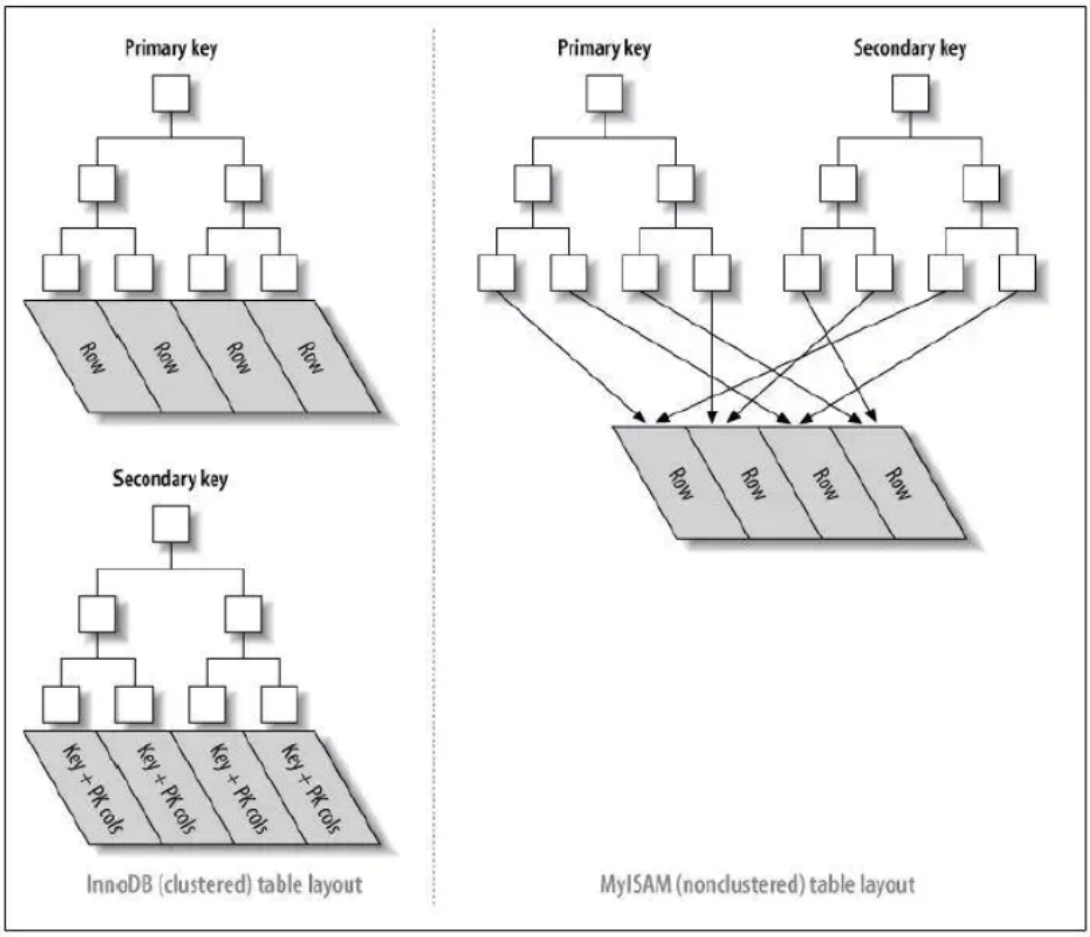
- 非叶子节点不存储data，只存储索引。
- 叶子节点包含索引字段和data

查看mysql中innodb叶节点数量大小：`show global status like "Innodb_page_size"`

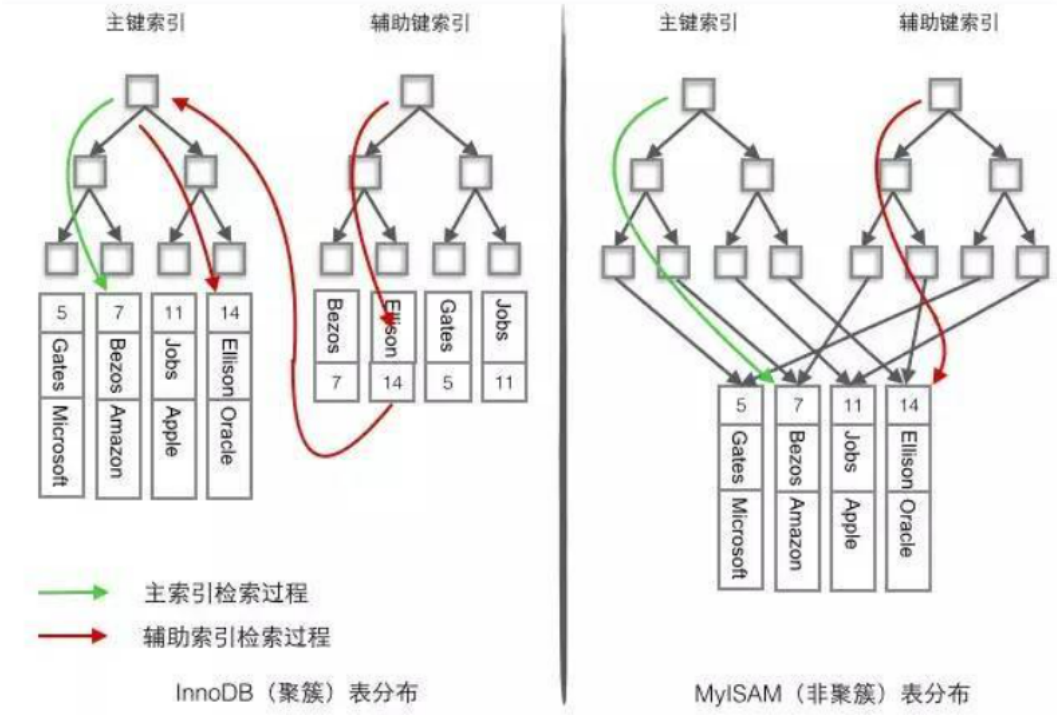
查看对应表中的索引：`show index from <table>`

聚簇索引和非聚簇索引

- 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据
- 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因



Id	Name	Company
5	Gates	Microsoft
7	Bezos	Amazon
11	Jobs	Apple
14	Ellison	Oracle



回表、索引覆盖、最左匹配、索引下推

- **回表**：

  id, name, age, gender. 其中id为自增主键索引，name为普通索引。

  `select * from user where name = "john"`。这个表有两个B+树。首先根据name查其B+树得到对应的id值，在根据id值查询id对应的B+树来检索对应的数据记录，这个过程叫做回表。**尽可能避免回表操作。** `select id from user where name = "john"`，无需回表。

- **索引覆盖**：

  `select id, name from user where name = "john"`。此操作根据name的值去查询name对应的B+树，能获取到id的值，不需要获取到所有列的值，因此不需要回表，这个过程叫做索引覆盖。使用`explain`会有`using index`的提示，**推荐使用。**

  在某些场景中，可以考虑将所有的列变成组合索引，就无需回表，加快查询速度。

  `select id, name, age from user where name = "john"` 则需要回表。

- **最左匹配：**

  首先解释一下**组合索引**或者**联合索引**：可以选择多个列来共同组成索引，但是要遵循最左匹配原则。

  id, name, age, gender. id为自增主键索引，name， age为联合索引。

  `select * from user where name = "john" and age = 12` -> 会走组合索引

  `select * from user where name = "john"` -> 会走组合索引

  `select * from user where age = 12` -> 不会走组合索引

  `select * from user where age = 12 and name = "john"` -> 会走组合索引，mysql自动优化顺序

- **索引下推**：

  `select * from user where name = "john" and age = 12`

  无索引下推：先根据name过滤并将数据拉取到server层，如何在server层对age进行过滤。

  有索引下推：直接根据两个条件进行过滤，筛选后再返回server层。

如何回答优化问题：

1. 加索引
2. 看执行计划`explain`语句
3. 优化sql语句，减少回表，增加索引覆盖，满足最左匹配等问题
4. 分库分表
5. 表结构设计优化

> "在实习或项目过程中做个一段sql优化，一般优化并不是出了问题才进行优化，在进行数据库建模或者数据库设计的时候要预先考虑到一些问题，比如字段的类型，字段的长度等，要创建合适的索引；在发生SQL问题的时候，还需要对SQL语句进行性能监控，观察他的执行计划，索引的创建和维护，SQL语句的调整、参数的设置多方面考虑"， 结合实际进行表达灵活变化。

事务，锁，MVCC

```
-- 查看事务是否自动提交
show variables like 'autocommit';

-- 设置是否自动提交
set autocommit = 0;     -- 0表示off，1表示on
```

事务

四个特性(ACID)：

可参考：**深入学习MySQL事务：ACID特性的实现原理**

◍原子性（Atomic）：由undolog实现

◍一致性（Consistency）：由其他三个特性共同实现，是最**根本**的。

◍隔离性（Isolation）：由MVCC实现

◍持久性（Durability）：由redolog实现，二阶段提交，WAL(write ahead log)，先写日志，再写数据。

并发事务的四个问题：

并发事务带来的问题

- 丢失数据（Lost to modify）

  指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

- 脏读（dirty read）

当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

Session-1(RU)	Session-2(RR)
begin; select salary from t1 where id = 1;(5000)	
	update t1 set salary=4500 where id = 1;
✕ select salary from t1 where id = 1; (4500)	
	rollback;
select salary from t1 where id = 1;(5000)	

处于Read-Uncommited模式下的会话一会出现脏读的情况。

- 不可重复读（unrepeatable read）

  指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。**这就发生了在一个事务内两次读到的数据是不一样的情况**，因此称为不可重复读。

  在RC隔离机制下，每次在进行快照读(select)的时候每次都会生成新的readview，在RR隔离机制下，只有第一次读的时候才会生成新的readview，来避免不可重复读的问题。

Session-1(RC)	Session-2(RR)
begin; select salary from t1 where id = 1;(5000)	
	update t1 set salary=4500 where id = 1;
select salary from t1 where id = 1;(5000)	
	commit;
✕ select salary from t1 where id = 1; (4500)	
commit;	

- 幻读（phantom read）

  幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，可能使用到了 `insert` `delete` `update` 语句更新了readview，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

  解决方法：添加表锁。

总结：

隔离级别	脏读	不可重复读	幻读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

锁：

```
show engine innodb status;  -- 可以查看锁的状态
```

分类：从对数据操作粒度上可分为表锁和行锁，从操作类型上可分为读锁（共享锁）和写锁（排他锁）。

🔘InnoDB行锁

对于update，delete，insert语句，innoDB会自动添加排他锁，普通select语句不会，可以使用以下语句加锁。

```
-- 共享锁
select * from t1 where col = 'condition' lock in share mode;
-- 排他锁
select * from t1 where col = 'condition' for update;
```

🔘行锁升级为表锁

```
-- 假定name为varchar类型，但是这里的1996为数字类型，会导致索引失效，行锁升级为表锁。
update t1 set age = 20 where name = 1996;
```

🔘间隙锁的影响

当使用范围条件查询而不是相等条件进行查询的时候，在请求共享或排他锁同时，innoDB会对符合条件的数据进行加锁，对于键值在条件范围内但不存在的记录，叫做“间隙”，InnoDB也会对这个间隙加锁，即间隙锁（next-key锁）

session-1	session-2
set autocommit=0;	set autocommit=0;
update t1 set name="Q" where id < 4;	
	insert into t1 (id, name)values(2, "ring");
commit;	

对于session-2中的`insert`语句中，其id<4，满足session1中间隙的条件，因此会阻塞直至session-1提交。

🔘查看InnoDB行锁的争用情况

```
show status like 'innodb_row_lock%'
```

```
mysql> show status like "innodb_row_lock%";
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_row_lock_current_waits | 0     |       # 当前为等待行锁的数量
| Innodb_row_lock_time          | 0     |       # 总锁定时长
| Innodb_row_lock_time_avg      | 0     |       # 每次等待平均时长
| Innodb_row_lock_time_max      | 0     |
| Innodb_row_lock_waits         | 0     |       # 系统启动后一共等待的次数
+-------------------------------+-------+
5 rows in set (0.00 sec)
```

MVCC

> Multi-Version Concurrency Control(多版本并发控制). 围绕InnoDB展开，不支持MyISAM。为了在并发控制的情况提高读写效率。

当前读：读取到的数据总是最新的数据。

什么时候会触发当前读?

```
select * from user lock in share mode;          -- select ... lock in share mode
select * from user for update;                    -- select ... for update
update ...
delete ...
insert ...
```

快照读：读取到的是历史版本的记录。

什么时候会触发快照读?

```
select ...    -- 朴素select语句
```

mvcc的组成部分

1. 隐藏字段：每一行的字段上都有用户不可见的字段
   - `DB_TRX_ID`：创建或者最后一次修改该记录的事务ID
   - `DB_ROW_ID`：即为隐藏主键，6字节的row id。
   - `DB_ROLL_PTR`：事务回滚指针，事务失败的话需要回滚到上一个历史记录，需要配合`undolog`使用。

name	age	gender	DB_TRX_ID	DB_ROW_ID	DB_ROLL_PTR
john	18	female	1	1	null

2. undolog(回滚日志)

   如果使用update语句，会将更改的数据保存到undolog中。一个undolog会存储多个历史记录，多个事务对同一条事务进行处理的时候，会形成一个历史记录链表。

   问题：当有多个历史记录的时候，可重复读读取的是哪一个历史记录?

3. readview(读视图)

   事务在进行快照读的时候产生的读视图。

其包含几个组件：

- `trx_list`：系统活跃的事务ID
- `up_limit_id`：列表中事务最小的ID
- `low_limit_id`：系统尚未分配的下一个事务ID

举例：
情况1：

事务1	事务2	事务3
begin;	begin;	begin;
		update ...; commit;
	快照读	

情况2：

事务1	事务2	事务3
begin;	begin; 快照读	begin;
		update ...; commit;
	快照读	

问题：在RR隔离级别下，这两种情况下是否能读到最新的值？

readview生成时机：

- 在RC隔离机制下，每次在进行快照读的时候每次都会生成新的readview。
- 在RR隔离机制下，只有第一次读的时候才会生成新的readview。

可见性原则算法：

> 当进行快照读的时候

1. 如果 `DB_TRX_ID` < `up_limit_id`， 则表示当前事务在此记录的事务发生之后开启，`DB_TRX_ID` 所在记录当然对当前事务可见，如果大于等于进入下一步；
2. 如果 `DB_TRX_ID` > `low_limit_id`，则表示当前事务还发生在此记录所提交的事务之前，`DB_TRX_ID` 所在的记录在 `readview` 生成之后才出现的，那么对当前事务不可见，如果小于进入下一步
3. 判断 `DB_TRX_ID` 是否仍在活跃事务中，如果在，则表示还在 `readview` 的生成时刻，还没有commit，修改的数据当前事务是不可见的，如果事务在 `readview` 生成之前 `commit` 是可见的，生成之后不可见。

分析：

- 当前 `trx_list` 为事务 `1, 2`， `up_limit_id` 为 `1`， `low_limit_id` 为 `4` （因为事务3已经提交）
- 所修改记录对应的隐藏字段 `DB_TRX_ID` 值为 `3` 。
- 因此在情况1下是可以读取到最新值，在情况2下不可以读到最新值。

> 如果事务中存在一次当前读的时候，下一次再进行快照读的时候会更新readview。这个现象叫“幻读”，在RR事务中需要配合间隙锁来解决幻读的问题。

隔离级别

参考：

彻底搞懂 MySQL 事务的隔离级别 - 阿里云社区

Innodb中的事务隔离级别和锁的关系 - 美团技术团队

> MySQL的事务隔离级别一共有四个，分别是读未提交(RU, read uncommitted)、读已提交(RC, read committed)、可重复读(RR, repeatable read)以及可串行化(Serializable)。
>
> MySQL的隔离级别的作用就是让事务之间互相隔离，互不影响，这样可以保证事务的一致性。

隔离级别比较与对性能的影响：可串行化>可重复读>读已提交>读未提交

MySQL默认的事务隔离级别是RR可重复读。

查看MySQL的事务级别：`select @@transaction_isolation;`

- READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

其他

面试问题：

面试官:讲讲mysql表设计要注意啥？ - 知乎

MySQL 面试题