MySQL基础

- **什么是MySQL** — MySQL 是一种关系型数据库，主要用于持久化存储我们的系统中的一些数据比如用户信息
- **数据库设计步骤**
  - 需求分析 — 分析用户的需求，包括数据、功能和性能需求。
  - 概念结构设计 — 主要采用 E-R 模型进行设计，包括画 E-R 图。
  - 逻辑结构设计 — 通过将 E-R 图转换成表，实现从 E-R 模型到关系模型的转换。
  - 物理结构设计 — 主要是为所设计的数据库选择合适的存储结构和存取路径。
  - 数据库实施 — 包括编程、测试和试运行
  - 数据库运行和维护 — 系统的运行与数据库日常维护
- **有了关系型数据库，为什么还需要NOSQL** — NOSQL数据库无需提前设计表结构，数据可以根据需要自由地存储和组织，而且相对于关系型数据库，NOSQL高效灵活，非常适合那些复杂、高变化、高并发量的场景中
- **关系型数据库和非关系型数据库的区别**
  - 关系型数据库 (RDBMS) — 适合需要复杂查询、事务支持和强一致性的应用，优点是数据一致性强、支持复杂的 SQL 查询，缺点是扩展性相对较弱。
  - 非关系型数据库 (NoSQL) — 适合需要高可用性、可扩展性和处理大数据量的应用，优点是扩展性强、支持多种数据模型，缺点是复杂查询支持较弱，通常只保证最终一致性
- **char和varchar的区别**
  - char是一种定长的数据类型，它的长度固定且在存储时会自动在结尾添加空格来将字符串填满指定的长度。char的长度范围是0-255
  - archar是一种可变长度的数据类型，它只会存储实际的字符串内容，不会填充空格。
- **ACID特性**
  - 原子性（Atomicity）
    - 事务是一个不可分割的工作单位，事务中包括的操作要么全部完成，要么全部不完成。
    - 实现方式：通过数据库管理系统的日志机制和回滚机制来确保原子性。当事务在执行过程中发生错误时，系统可以利用日志文件将事务回滚到开始前的状态。
  - 一致性（Consistency）
    - 事务必须使数据库从一个一致性状态变换到另一个一致性状态，即事务执行前后，数据库都必须保持一致性。
    - 实现方式：数据库管理系统通过一系列的检查和约束来确保数据的一致性，如触发器、完整性约束等。同时，系统也会在事务执行前后对数据进行验证，以确保数据的一致性。
  - 隔离性（Isolation）
    - 在并发环境中，多个事务并发执行时，一个事务的执行不应影响其他事务。隔离性确保了每个事务都在独立的、隔离的环境中执行。
    - 实现方式：数据库管理系统通过锁机制和MVCC（多版本并发控制）等技术来实现事务的隔离性。锁机制可以防止多个事务同时修改同一数据，而MVCC则允许多个事务同时读取同一数据而不会相互干扰。
  - 持久性（Durability）
    - 一旦事务提交，则其结果就是永久性的。即使发生系统崩溃或重启等故障，数据库也能恢复到提交后的状态。
    - 实现方式：数据库管理系统通过日志机制和恢复机制来确保事务的持久性。当事务提交时，系统会将事务的修改记录到日志文件中。如果系统发生故障，恢复机制可以利用日志文件来恢复数据库到故障前的状态。
- **drop、delete、truncate区别**
  - 用法不同
    - 删除表结构（整个表） — drop(丢弃数据): drop table 表名 ，直接将表都删除掉，在删除表的时候使用。
    - 只删除数据不删除表结构
      - truncate (清空数据) : truncate table 表名 ，只删除表中的数据，再插入数据的时候自增长 id 又从 1 开始，在清空表中数据的时候使用。
      - delete（删除数据）：delete from 表名 where 列名=值，删除某一行的数据，如果不加 where 子句和truncate table 表名作用类似。
  - 属于不同的数据库语言
    - DDL(数据定义语言)语句 - 对数据库内部的对象进行创建、删除、修改的操作语言。 — truncate / drop
    - DML （数据库操作语言）- 是指对数据库中表记录的操作，主要包括表记录的插入、更新、删除和查询，是开发人员日常使用最频繁的操作。 — delete
  - 执行速度不同（drop>truncate>delete）
    - delete命令执行的时候会产生数据库的binlog日志，而日志记录是需要消耗时间的，但是也有个好处方便数据回滚恢复。
    - truncate命令执行的时候不会产生数据库日志，因此比delete要快。除此之外，还会把表的自增值重置和索引恢复到初始大小等。
    - drop命令会把表占用的空间全部释放掉。
    - tips：更多关注使用场景而不是效率
- **数据库是如何删除的?**
  - 数据库删除是怎么删的？业务都是逻辑删除，表里有个字段叫status，正常是1，删除就变为-1，当他已经被删除。再去查数据的时候只查status为1。如何再去定期清理呢？status为-1并且已经过了一个固定的时间，比如一个月之类的。创建了一些定时任务，了解一下定时任务框架有哪些。
  - 如何删除一段时间内的数据库？ — 创建了一些定时任务，了解一下定时任务框架有哪些 — Java 定时任务框架
    - Quartz
      - 功能强大的任务调度框架，支持复杂的任务计划和并发任务执行。
      - 支持 Cron 表达式、持久化任务信息、集群等高级功能。
    - Spring
      - 集成在 Spring 框架中，简化了定时任务的配置和管理。
      - 支持 Cron 表达式、固定间隔执行等。
    - ScheduledExecutorService
      - Java 标准库中的定时任务执行器，属于 java.util.concurrent 包。
      - 提供了简单的 API，用于调度任务在给定的延迟后执行或定期执行。

函数依赖类型	描述	例子
函数依赖（FD）	如果知道了属性X的值，就能确定属性Y的值	学号→姓名。知道学生的学号，就能确定学生的姓名。
部分函数依赖（Partial FD）	如果X→Y，且X的某个真子集X'也可以确定Y，则Y部分函数依赖于X	学号+科目→姓名。只需要学号部分就能确定姓名，不需要知道科目。
完全函数依赖（Full FD）	如果X→Y，且X的任何真子集都不能单独确定Y，则Y完全函数依赖于X	学号+科目→成绩。必须同时知道学号和科目才能确定成绩。
传递函数依赖（Transitive FD）	如果X→Y，Y→Z，且X不直接决定Z，则Z传递函数依赖于X	员工ID→部门，部门→部门位置。员工ID通过部门间接决定部门位置。

- **SQL中PK、UK、CK、FK、DF是什么意思**
  - PK：Primary Key ，主键约束
    - 主键列的值必须是唯一的，且不能为Null
    - 一个表中只能有一个主键，但主键可以由多个列组成（复合主键）
  - UK：Unique Key， 唯一约束 — 在同一列或列组合中不能有两行具有相同的值。唯一约束允许NULL值（除非另有规定），且一个表可以有多个唯一约束
  - CK： check()， 检查约束 — 限制列中的值必须满足特定的条件
  - FK：Foreign Key， 外键约束
    - 确保一个表中的数据引用另一个表中的唯一数据行
    - 外键引用的主键值发生变化或被删除，数据库将根据外键约束的配置执行相应的操作（如级联更新或删除）
  - DF：default ，默认约束
    - 果在插入记录时未指定该列的值，则会自动填充为默认值
    - 例如，可以为日期字段指定当前日期为默认值
- **什么是数据库范式，为什么要反范式**
  - 数据库范式
    - 1NF — 每个字段不可再分
    - 2NF — 每个字段都完全依赖于主键 — 函数依赖
      - 函数依赖
      - 部分函数依赖
      - 完全函数依赖
      - 传递函数依赖
    - 3NF — 在 2NF 的基础之上，消除了非主属性对于键的传递函数依赖
  - 为什么要反范式 — 完全遵守数据库三范式，会丢失一些读取性能
    - 表中不能有任何冗余字段，这就使得查询的时候就会经常有多表关联查询，这无疑是比较耗时的
    - 比如我们可以在表中增加一些冗余字段，方便我们进行数据查询
    - 本质上是 — 用空间换时间
- **为什么大厂不建议使用多表join**
  - 最主要的原因就是join的效率比较低
  - MySQL 8.0.18版本以前使用嵌套循环的方式来实现关联查询
    - 简单点说就是要通过两层循环，用第一张表做外循环，第二张表做内循环，外循环的每一条记录跟内循环中的记录作比较，符合条件的就输出
    - 有2张表join的话，复杂度最高是O(n^2)，3张表则是O(n^3)...随着表越多，表中的数据量越多，JOIN的效率会呈指数级下降
- **order by 是怎么实现的** — 取决于优化器的选择，如果优化器认为走索引更快，那么就会用索引排序，否则，就会使用filesort (执行计划中extra中提示：using filesort)
- **count(1)、count(*) 与 count(列名)**
  - count() 是一个聚合函数，函数的参数不仅可以是字段名，也可以是其他任意表达式，该函数作用是**统计符合查询条件的记录中，函数指定的参数不为 NULL 的记录有多少个**
  - 假设 count() 函数的参数是数字 1 这个表达式
    - 这条语句是统计「 t_order 表中，1 这个表达式不为 NULL 的记录」有多少个
    - 1 这个表达式就是单纯数字，它永远都不是 NULL，所以上面这条语句，其实是在统计 t_order 表中有多少个记录
  - count(*) 其实等于 count(0)，也就是说，当你使用 count(*) 时，MySQL 会将 * 参数转化为参数 0 来处理
  - count(*) 执行过程跟 count(1) 执行过程基本一样，没有什么差距
    - 如果有多个二级索引的时候，优化器会使用key_len最小的二级索引进行扫描。
    - 只有当没有二级索引的时候，才会采用主键索引来进行统计
  - count(字段) 执行过程是怎样的 — 会采用全表扫描的方式来计数，所以它的执行效率是比较差的
  - 小结
    - count(1)、 count(*)、 count(主键字段)在执行的时候，如果表里存在二级索引，优化器就会选择二级索引进行扫描。
    - 所以，如果要执行 count(1)、 count(*)、 count(主键字段) 时，尽量在数据表上建立二级索引，这样优化器会自动采用 key_len 最小的二级索引进行扫描，相比于扫描主键索引效率会高一些。
    - 再来，就是不要使用 count(字段) 来统计记录个数，因为它的效率是最差的，会采用全表扫描的方式来统计。如果非要统计表中该字段不为 NULL 的记录个数，建议给这个字段建立一个二级索引
  - 如何优化count（*） — 面对大表的记录统计，有更好的办法
    - 近似值 — 如果你的业务对于统计个数不需要很精确，比如搜索引擎在搜索关键词的时候，给出的搜索结果条数是一个大概值
      - 这时，我们就可以使用 show table status 或者 explain 命令来表进行估算
      - 执行 explain 命令效率是很高的，因为它并不会真正的去查询
    - 额外表保存计数值
      - 如果是想精确的获取表的记录总数，我们可以将这个计数值保存到单独的一张计数表中。
      - 当我们在数据表插入一条记录的同时，将计数表中的计数字段 + 1。也就是说，在新增和删除操作时，我们需要额外维护这个计数表
- **limit 0,100和limit 10000000,100一样吗**
  - 不一样，这是MySQL中典型的深度分页的问题
  - MySQL的limit m n工作原理就是先读取前面m+n条记录，然后抛弃前m条，然后返回后面n条数据，所以m越大，偏移量越大，性能就越差
  - 所以，limit 10000000,100 — 要先读取10000100条数据，然后再抛弃前面的10000000条
- **MySQL一条SQL语句的执行过程**
  - 使用连接器，通过客户端/服务器通信协议与 MySQL 建立连接。并查询是否有权限
  - Mysql8.0之前检查是否开启缓存，开启了 Query Cache 且命中完全相同的 SQL 语句，则将查询结果直接返回给客户端；
  - 由解析器（分析器）进行语法分析和语义分析，并生成解析树。
    - 如查询是select、表名users、条件是age='18' and name='Hollis'
    - 预处理器则会根据 MySQL 规则进一步检查解析树是否合法。比如检查要查询的数据表或数据列是否存在等。
  - 由优化器生成执行计划。根据索引看看是否可以优化
  - 执行器来执行SQL语句，这里具体的执行会操作MySQL的存储引擎来执行 SQL 语句，根据存储引擎类型，得到查询结果。若开启了 Query Cache，则缓存，否则直接返回
- **MySQL字段类型**
  - 数值类型
    - 整型
      - TINYINT: 一个非常小的整数，占用1个字节的存储空间。有符号范围是-128到127，无符号范围是0到255
      - SMALLINT: 一个小的整数，占用2个字节的存储空间。有符号范围是-32,768到32,767，无符号范围是0到65,535
      - MEDIUMINT: 一个中等大小的整数，占用3个字节的存储空间
      - INT或INTEGER: 一个标准的整数，占用4个字节的存储空间
      - BIGINT: 一个大整数，占用8个字节的存储空间
    - 浮点型
      - FLOAT: 单精度浮点数，占用4个字节的存储空间。它可以表示大约7位有效数字的数值。
      - DOUBLE: 双精度浮点数，占用8个字节的存储空间。它可以表示大约15位有效数字的数值。
      - REAL: 实际上，REAL是FLOAT的同义词，除非在特定的SQL模式下有所不同。通常，它们都被视为单精度浮点数
    - 定点型
      - DECIMAL(M,N) 或 NUMERIC(M,N): 定点数，其中M表示数字的总位数（包括小数点两侧），N表示小数点右侧的位数
      - DECIMAL类型用于存储精确的数值，如财务数据，因为它不会引入浮点数的舍入误差
  - 字符类型CHAR、VARCHAR、TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT、TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB 等
    - 常用
      - Char — 固定长度
      - Varchar
        - 可变长度
        - VARCHAR(10)存储超过 10 个字符时，就需要修改表结构才可以。
    - Text类 — 数据库规范通常不推荐使用 BLOB 和 TEXT 类型
    - BLOB类
  - 日期时间类型 — YEAR、TIME、DATE、DATETIME 和 TIMESTAMP — DATETIME 类型没有时区信息，TIMESTAMP 和时区有关
  - Null和''（空字符串）的区别（也可以说为什么不适用NULL为列默认值）
    - NULL 代表一个不确定的值,就算是两个 NULL,它俩也不一定相等
    - NULL 会影响聚合函数的结果。*会输出带null结果
    - 查询 NULL 值时，必须使用 IS NULL 或 IS NOT NULLl 来判断，而不能使用 =、!=、 <、> 之类的比较运算符。而''是可以使用这些比较运算符的
    - ''的长度是 0，是不占用空间的，而NULL 是需要占用空间的
- **MySQL存储引擎**
  - 默认使用InnoDB — 因为只有InnoDB引擎支持事务
    - 需要主键
    - 支持外检
  - 架构使用插件式架构，支持多种存储引擎（存储引擎是基于表的）
  - innodb Mysiam区别
    - 事务支持:
      - InnoDB 支持事务，它提供了提交、回滚和崩溃恢复功能，使得它适合执行多步骤的更新操作。
      - MyISAM 不支持事务，适合读取密集型的应用，但不适合需要高度可靠性的系统。
    - 表锁定与行锁定:
      - InnoDB 提供行级锁定，这意味着在多用户同时访问时，性能更优，冲突更少。
      - MyISAM 仅支持表级锁定，当一个用户在进行写操作时，其他用户必须等待整个表解锁才能进行读写操作。
    - 外键约束:
      - InnoDB 支持外键，这对于保持数据库的引用完整性非常有用。
      - MyISAM 不支持外键。
    - 数据恢复:
      - InnoDB 更适合需要高可靠性的系统，因为它支持事务日志（Redo Log），有助于崩溃恢复。
      - MyISAM 更难以从崩溃中恢复，因为它不支持事务日志。
    - 存储限制:
      - MyISAM 适合处理大量的数据，因为它支持的表的大小通常只受限于操作系统的文件大小限制。
      - InnoDB 通常有更大的存储开销，因为它需要存储事务和撤销信息。

数据库基础知识

- 数据库概念
  - 数据库 — 简称DB，数据库管理系统管理数据的总和
  - 数据库管理系统 — 是一种操纵和管理数据库的大型软件，通常用于建立使用和维护数据库
  - 数据库系统 — DBS，通常由软件、数据库和数据管理员DBA构成
  - 数据库管理员 — 全面管理和控制数据库系统
- 数据库元素
  - 元组tuple — 关系是一张表，一个元组就是每行（即数据库的每条记录），也可以叫行
  - 码 — 能识别唯一标识实体的属性，对应表中的列
  - 候选码 — 若关系中的某一属性或属性组的值能唯一的标识一个元组，而其任何、子集都不能再标识，则称该属性组为候选码。
  - 主码/主键 — 主码也叫主键。主码是从候选码中选出来的。 一个实体集中只能有一个主码，但可以有多个候选码
  - 外码/外键
    - 如果一个表中的一个属性是另一个关系中的主键，这个属性就是外键
    - 外键的好处
      - 保证了数据库数据的一致性和完整性；
      - 级联操作方便，减轻了程序代码量；
  - 主键和外键的区别
    - 主键(主码)：主键用于唯一标识一个元组，不能有重复，不允许为空。一个表只能有一个主键。
    - 外键(外码)：外键用来和其他表建立联系用，外键是另一表的主键，外键是可以有重复的，可以是空值。一个表可以有多个外键。
  - 主属性（Prime Attribute） — 候选键中出现过的属性就是主属性。
  - 非主属性 — 不包含在任何一个候选码中的属性称为非主属性。
- ER图
  - 实体
  - 属性
  - 联系
- **外键和级联?**
  - 缺点
    - 外键与级联更新适用于单机低并发，不适合分布式、高并发集群；
    - 对分库表不友好
    - 外键影响数据库的插入速度
    - 增加了复杂性，每次都要考虑外键约束
  - 优点
    - 保证了数据库数据的一致性和完整性；
    - 级联操作方便，减轻了程序代码量；

MySQL数据库说说你了解的关键字

- 数据定义语言（DDL）关键字
- CREATE：用于创建数据库、表、索引等对象。
- ALTER：用于修改已存在的数据库对象，如表的结构。
- DROP：用于删除数据库、表、索引等对象。
- 数据查询语言（DQL）关键字
- SELECT：用于从数据库表中检索数据。
- FROM：指定查询数据的来源表。
- WHERE：用于过滤记录，只选择满足条件的记录。
- GROUP BY：用于根据一个或多个列对结果集进行分组。
- HAVING：在SQL的GROUP BY子句之后进行条件过滤，与WHERE不同，HAVING用于对分组后的数据进行过滤。
- ORDER BY：用于对结果集进行排序，可以指定升序（ASC）或降序（DESC）。
- JOIN：用于结合两个或多个表的行，基于某些相关的列之间的关系。
- 数据操纵语言（DML）关键字
- INSERT：用于向表中插入新记录。
- UPDATE：用于修改表中的现有记录。
- DELETE：用于从表中删除记录。
- 数据控制语言（DCL）及其他关键字
- GRANT：用于赋予用户特定的权限。
- REVOKE：用于收回之前赋予用户的权限。
- 通用SQL关键字和函数
- BETWEEN：用于在指定范围内选择值，常与数字和时间范围查询一起使用。
- DISTINCT：用于从结果集中删除重复的行。
- COUNT, SUM, AVG, MIN, MAX：聚合函数，用于对一组值执行计算，并返回单个值。
- IS NULL 和 IS NOT NULL：用于测试列中的值是否为NULL。
- LIKE：用于在搜索中使用模式匹配。
- IN：用于测试某个值是否在给定的列表中。
- CASE：用于在SQL语句中添加条件逻辑。
- 此外，还有一些其他重要的关键字，如UNSIGNED（用于定义非负数值的列），UNION（用于合并两个或多个SELECT语句的结果集），以及CURSOR（用于声明和处理游标）等。

MySQL锁

- 锁
  - 加锁机制
    - 总：是两种并发控制机制，用于处理多个事务同时访问和修改相同数据时的数据一致性问题
    - 什么是悲观锁
      - 借助数据库锁机制在修改数据之前先锁定，再修改的方式被称之为悲观并发控制，很保守
      - 悲观锁的流程
        - 在对记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。
        - 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。具体响应方式由开发者根据实际需要决定。
        - 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
        - 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常
    - 什么是乐观锁
      - 乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。
      - 乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本
    - MySQL的实现方式
      - 乐观锁在MySQL中通常是通过版本号（Version）或时间戳（Timestamp）来实现的。
        - 版本号机制：
          - 在数据表中增加一个版本号字段，如version，用于表示数据被修改的次数。
          - 当数据被修改时，version值会自动加一。
          - 当线程尝试更新数据时，会同时读取并检查version值。
          - 提交更新时，只有当前读取到的version值与数据库中的值相等时，才会执行更新操作。
          - 如果version值不匹配，说明数据已被其他事务修改，此时会重试更新操作，直到成功。
        - 时间戳机制：
          - 类似于版本号，可以在表中增加一个时间戳字段。
          - 每次更新数据时，都会比较时间戳以确定数据是否被其他事务修改
      - 悲观锁在MySQL中通常是通过使用SELECT ... FOR UPDATE语句来实现的。
        - 开始事务：使用BEGIN、START TRANSACTION等语句开始一个新的事务。
        - 锁定资源：使用SELECT ... FOR UPDATE语句查询并锁定需要的资源。这会阻止其他事务修改被锁定的数据，直到当前事务完成。
        - 执行操作：在资源被锁定期间，执行所需的数据修改操作。
        - 提交或回滚事务：操作完成后，使用COMMIT提交事务以永久保存更改，或者使用ROLLBACK回滚事务以撤销更改并释放锁。
  - 锁粒度
    - 全局锁：一般用作数据备份，可能造成业务停滞，可以通过可重复读解决，flush tables with read lock 释放 unlock tables
    - 表级锁
      - 表锁
        - 读锁：lock tables table read
        - 写锁：lock tables table write
        - 释放：unlock tables 释放当前会话中的所有表锁，或者会话结束后
        - 并发度很低，一旦加上将对这张表的任何操作都不允许，除了会限制别的线程读写，还会限制本线程接下来的读写
      - 元数据锁：不需要显式的使用MDL，我们对数据库表进行操作时，会自动为这个表加上MDL，对一张表进行CURD操作时，加的是MDL读锁，对一张表做结构变更操作时，加的时DML写锁
      - 意向锁：意向锁的目的是为快速判断表里是否有记录被加锁。不如就需要遍历表中的所有记录。在InnoDB引擎的表里对某些记录加上共享/独占锁之前 需要先在表级别加上一个意向独占/共享锁
      - AUTO-INC锁：在插入数据时，会加一个表级别的AUTO-INC锁，等插入语句执行完了后，才会把AUTO-INC锁释放，自增值的唯一性和连续性
        - 注意：这是一种特殊的表锁机制（**自增长**），锁不再是一个事务提交后释放，而是再执行完插入语句后立即释放。一个事务在持有AUTO-INC锁的过程中，其它事务的如果要向该表插入语句都会被阻塞，从而保证插入数据时，被AUTO-INCREMENT修饰的字段的值是连续递增的。
        - 但是再大数据量插入的情况下，会影响插入性能。所以从MySql5.1.22版本开始。提供了一种轻量级的锁。它是直接给该字段赋值一个自增的值，就把这个轻量级锁释放了，而不需要等待整个插入语句执行完才释放锁
        - **高并发情况下自增主键会不会重复，为什么** — MySQL通过AUTO-INC锁机制确保了自增主键的唯一性
    - 页锁 — 介于表锁和行锁之间的锁
    - 行级锁
      - 什么是行锁：行锁是一种粒度更细的锁机制，用于保护数据库表中的行级记录。行锁可以在并发时提供更高的并发性能，减少锁冲突的概率（innodb才支持），**行锁是基于索引的**，而不是整个表。**只有通过索引访问数据时，InnoDB才会使用行级锁。如果没有查询字段没有使用索引，那么InnoDB会升级为表级锁**
      - 几种实现(间隙锁和临键锁都是用来解决幻读问题的，在已提交读的情况下会失效)
        - Record Lock记录锁
          - 读取时，通过select ... lock in share mode加读锁 通过 select ... for update加写锁
          - 记录锁就是仅仅把一条记录锁上，方便来做悲观锁
        - Gap Lock间隙锁
          - 锁定的是一个范围，但是不包含范围本身。其根本目的是为了防止改变原有状态，防止在一个区间里面再插入元素。所以对一个范围加锁，解决幻读现象，幻读就是一个范围内的记录突然多了或少了。
          - 例如select * from t where id = 3 for update 或者 select * from t where id > 1 and id < 6 for update就会把(1,6)区间锁定。注意：间隙锁虽然存在X型间隙锁和S型间隙锁，但是并没有什么区别 。间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系，因为间隙锁的目的只是为了防止插入幻影记录而提出的
        - Next-key Lock临键锁 — Record + Gap Lock的组合，锁定一个范围，并且**锁定右范围本身**，是**间隙加上它右边记录组成的左开右闭区间** 注意不同于间隙锁 因为这是包含记录锁的，所以相同范围内的X和X锁是会互斥的
      - 插入意向锁：一个事务再插入一条记录的时候，需要判断插入位置是否已经被其它事务加了间隙锁。如果有的话，插入操作就会发生阻塞，知道拥有间隙锁的那个事务提交为止，在此期间，会生成一个插入意向锁，表明有事务想在某个区间插入新记录，但是现在处于等待状态
  - 兼容性
    - 共享锁：也叫读锁
    - 排他锁：也叫写锁
- 死锁（可能有情景题）
  - 阻塞的概念
    - 因为不同锁之间的兼容性关系，在有些时刻一个事务中的锁需要等待另一个事务中的锁释放它所占用的资源
    - 阻塞并不是一件坏事，其是为了确保事务可以并发且正常地运行
  - 概念 — 是指两个或两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。若无外力作用，事务都将无法推进下去
  - 出现原因
    - 资源竞争 — 多个事务试图同时访问相同的资源，如数据库表、行、页或锁。但是它们请求资源的顺序不同，导致互相等待
    - 未释放资源 — 事务在使用完资源后未及时释放资源，导致其他事务无法获得所需的资源。这可能是由于程序中的错误或异常情况引起的
    - 不同事务的执行速度不同 — 如果一个事务在获取资源后执行速度很慢，而其他事务需要等待该事务释放资源，那么可能会导致其他事务超时，从而发生死锁
    - 操作的数据量过大 — 在持有锁的同时，又请求获取更多的锁，导致互相等待
  - 解决（避免）办法 — 形成死锁的必要条件：互斥、占有并等待、不可抢占和循环等待
    - 减少锁的数量 — 选择较低的隔离级别可以减少锁的数量 — 隔离级别越低，事务请求的锁越少或保持锁的时间就越短
    - 减少锁的时长
      - 加快事务的执行速度，降低执行时间，也能减少死锁发生的概率
      - 通过**优化查询**、**减少事务中操作的数据量**或者**只在必要时请求锁**来实现
    - 固定顺序访问 — 数据事务在访问同一张表时，应该以相同的顺序获取锁，这样可以避免死锁的发生
    - 减少操作的数据量 — 尽量减少事务操作的数据量，尽量减少事务的持有时间，这样可以降低死锁的发生几率
    - 破坏不可抢占条件 — 设置优先级并允许高优先级的进程抢占资源
    - 破坏循环等待条件 — 确保所有事务以相同的顺序请求资源可以避免产生循环等待的环路。例如，如果事务总是先锁定表A，然后锁定表B，循环等待的可能性就会大大减少。
  - Java应用层的解决办法
    - 避免死锁
      - 锁顺序 — 确保所有线程获取多个锁的顺序一致。这是避免循环等待条件发生的一个有效方法
      - 超时尝试 — 在尝试获取锁时使用带超时的尝试（例如，使用tryLock方法），如果在指定时间内无法获取所有所需的锁，则释放已持有的锁并重新尝试或回退
      - 锁分割 — 将大的锁分割成几个小锁（如果可能的话），这样可以减少不同线程之间因为锁竞争导致的阻塞
    - 解决死锁
      - 手动干预 — 通过重启应用或终止部分线程来打破死锁。这是最简单但也是最粗暴的方法，可能会导致数据不一致
      - 死锁恢复机制 — 在检测到死锁后，自动触发某些恢复机制，例如回滚事务或释放某些锁，以打破死锁循环
      - 使用并发库 — 尽可能使用Java提供的高级并发库（如java.util.concurrent中的类），这些库通常通过封装复杂的同步逻辑来减少死锁的可能性
- **数据库读写冲突怎么做**
  - 锁定（Locking）
    - 排他锁（Exclusive Locks）：当一个事务在一个数据项上加上排他锁时，只允许该事务对该数据项进行读写操作。其他事务必须等待直到锁被释放。
    - 共享锁（Shared Locks）：允许多个事务同时读取同一个数据项，但在共享锁定的数据项上，任何事务都不能进行写操作。如果一个事务需要写入，它必须等待直到所有的共享锁被释放，并且获得排他锁。
  - 乐观并发控制 — 在乐观并发控制中，事务在执行过程中不会立即锁定数据，而是在事务提交时检查数据是否被其他事务修改。如果在事务读取数据后和提交之前，数据被其他事务修改，当前事务会被回滚，并可能重新尝试。
  - 多版本并发控制 — MVCC允许每个读操作访问数据的一个快照，而不是最新的版本。这意味着读操作可以不受写操作的影响继续进行，因为它们操作的是不同版本的数据。这种方式特别适用于读操作远多于写操作的数据库。
  - 两阶段锁定协议 — 这是保证事务序列化的一种方法，分为两个阶段：加锁阶段和解锁阶段。在加锁阶段，事务可以获得任意数量的锁，但一旦释放了任何锁，它就不能再获得新的锁。这种方法可以防止某些类型的并发相关的问题，但可能导致死锁。
  - 时间戳 — 每个事务都有一个唯一的时间戳。使用时间戳来决定事务的优先级，早的事务优先于晚的事务。如果一个事务尝试访问一个已经被具有较新时间戳的事务修改的数据项，那么较早的事务可能会被回滚。