目录

[1. 什么是字符集和编码？ASCII、UTF-8、UTF-16、UTF-32又是什么？ 4](#_Toc78408008)

[2. MySQL的逻辑架构 4](#_Toc78408009)

[3. 说一下 MySQL 执行一条查询语句的内部执行过程、一条SQL查询语句在MySQL中如何执行的？（看上图）日志系统 4](#_Toc78408010)

[4. SQL优化 6](#_Toc78408011)

[5. 怎么看执行计划（explain），如何理解其中各个字段的含义 7](#_Toc78408012)

[6. 关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？ 7](#_Toc78408013)

[7. 数据库停掉重启需要进行什么操作？ 7](#_Toc78408014)

[8. MySql 的limit字段（limit 1000000加载很慢的话，你是怎么解决的呢？） 8](#_Toc78408015)

[9. select for update有什么含义，会锁表还是锁行还是其他 8](#_Toc78408016)

[10. MySQL的GROUP BY 和 JOIN 8](#_Toc78408017)

[9](#_Toc78408018)

[11. 数据库的索引类型 9](#_Toc78408019)

[12. 聚集索引与非聚集索引的区别 9](#_Toc78408020)

[13. 说一下事务是怎么实现的 10](#_Toc78408021)

[14. 事务隔离：为什么你改了我还看不见？（重点如何实现事物的隔离性） 10](#_Toc78408022)

[15. 事务是什么锁的种类和区别 11](#_Toc78408023)

[16. 全局锁和表锁:给表加个字段怎么有这么多障碍 11](#_Toc78408024)

[17. 行锁功过：怎么减少行锁对性能的影响 12](#_Toc78408025)

[18. 事务到底是隔离的还是不隔离的？ 12](#_Toc78408026)

[19. MySQL怎么建立索引，怎么建立主键索引，怎么删除索引？ 13](#_Toc78408027)

[20. MySQL为什么有时候会选择错索引？ 13](#_Toc78408028)

[21. 索引的优缺点，什么时候使用索引，什么时候不能使用索引（重点） 13](#_Toc78408029)

[22. 聚集索引与非聚集索引的区别 14](#_Toc78408030)

[23. 数据库索引的分类（功能分类和底层分类）索引的结构，以及为什么，索引的优劣势 14](#_Toc78408031)

[24. 深入浅出索引（上、下） 23](#_Toc78408032)

[25. 索引失效可能会导致什么？索引为什么失效？ 24](#_Toc78408033)

[26. Hash索引和B+树索引区别是什么？你在设计索引是怎么抉择的？ 24](#_Toc78408034)

[27. 什么是最左前缀原则？什么是最左匹配原则？ 24](#_Toc78408035)

[28. 数据库自增主键可能遇到什么问题？ 24](#_Toc78408036)

[29. MySql中的数据是存在哪的？又是如何存储的呢？ 24](#_Toc78408037)

[30. MySQL中使用什么存储引擎 25](#_Toc78408038)

[31. Mysql的两种存储引擎——InnoDB、MyISAM（以插件的形式设计、使用场景） 25](#_Toc78408039)

[32. InnoDB中B+树有几层 26](#_Toc78408040)

[33. 为什么MySQL使用B+树 27](#_Toc78408041)

[34. 隔离级别 28](#_Toc78408042)

[35. MySql如何解决死锁问题 29](#_Toc78408043)

[36. 间隙锁、行锁、表锁 29](#_Toc78408044)

[37. 封锁 29](#_Toc78408045)

[38. 读写锁的作用、为什么要用 30](#_Toc78408046)

[39. 乐观锁和悲观锁（使用场景） 31](#_Toc78408047)

[40. MySQL的行级表和oracle的行级表底层实现 33](#_Toc78408048)

[41. IEG光子工作室 33](#_Toc78408049)

[42. 数据库如何做到持久化 33](#_Toc78408050)

[43. redolog和undolog记录有什么区别 33](#_Toc78408051)

[44. 最大连接数的设定有什么影响 33](#_Toc78408052)

[45. 主从库的同步方法 33](#_Toc78408053)

[46. Mvcc的实现原理（多版本并发控制，只说了个大概） 33](#_Toc78408054)

[47. Mvcc和乐观锁的区别 34](#_Toc78408055)

[48. Next-Key Locks（解决MVCC的幻读问题） 34](#_Toc78408056)

[49. Redis和MySQL的区别 35](#_Toc78408057)

[50. 如果是一个群消息，你要这么处理？因为每个人读取的数据的进度是不一样的（如何用redis实现多人消息队列） 36](#_Toc78408058)

[51. Redis持久化 36](#_Toc78408059)

[52. 假设我们有一个场景，比如说唯一游戏里面我们会有几千万甚至上亿的玩家上一个角色的数据，我们会有一个角色表。表里面角色数量太多情况下对于严重的影响数据的查找以及修改，如果让你来设计，你怎么对这个角色表做一个设计，尽可能的去做，不会因为角色数量的增加而影响了我们一个读写的速率。 39](#_Toc78408060)

[53. Redis是一个单进程架构为什么快？ 39](#_Toc78408061)

[54. 如果大量的key同一时间过期怎么办？ 40](#_Toc78408062)

[55. 跳跃表时间复杂度？使用场景 40](#_Toc78408063)

[56. 为什么redis用跳跃表不用红黑树（https://www.cnblogs.com/cjjjj/p/12751487.html） 40](#_Toc78408064)

[57. 平衡二叉树、二叉搜索树、B树、B+树和红黑树的区别 41](#_Toc78408065)

[58. 互斥锁和自旋锁的区别和应用场景 41](#_Toc78408066)

[59. 为什么采用B+树这么一个结构 41](#_Toc78408067)

[60. 41](#_Toc78408068)

[61. 41](#_Toc78408069)

[62. Redis的数据结构 41](#_Toc78408070)

[63. 41](#_Toc78408071)

[64. 41](#_Toc78408072)

[65. Mysql实战 41](#_Toc78408073)

[66. 在高并发情况下，如何做到安全的修改同一行数据？ 41](#_Toc78408074)

[67. MySql数据库cpu飙升的话，要怎么处理呢 41](#_Toc78408075)

[68. MySql的主从延迟，你怎么解决 42](#_Toc78408076)

[69. 如果某个表有近千万数据，CRUD比较慢，如何优化。 42](#_Toc78408077)

[70. 如果让你做分库与分表的设计，简单说说你会怎么做？ 43](#_Toc78408078)

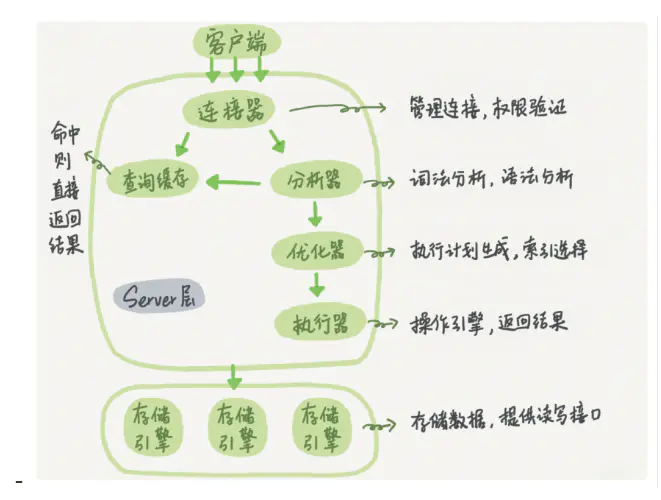
[71. 43](#_Toc78408079)

[72. 说一下 MySQL 执行一条查询语句的内部执行过程 43](#_Toc78408080)

1. 什么是字符集和编码？ASCII、UTF-8、UTF-16、UTF-32又是什么？

<https://mp.weixin.qq.com/s/jgD4L0e6978Ubgv-CMb7JA>

1. MySQL的逻辑架构



Mysql逻辑架构图主要分三层：

（1）第一层负责连接处理，授权认证，安全等等

（2）第二层负责编译并优化SQL

（3）第三层是存储引擎。

1. 说一下 MySQL 执行一条查询语句的内部执行过程、一条SQL查询语句在MySQL中如何执行的？（看上图）日志系统

* 连接器：客户端先通过连接器连接到 MySQL 服务器。
* 缓存：连接器权限验证通过之后，先查询是否有查询缓存，如果有缓存（之前执行过此语句）则直接返回缓存数据，如果没有缓存则进入分析器。
* 分析器：分析器会对查询语句进行语法分析和词法分析，判断 SQL 语法是否正确，如果查询语法错误会直接返回给客户端错误信息，如果语法正确则进入优化器。
* 优化器：优化器是对查询语句进行优化处理，例如一个表里面有多个索引，优化器会判别哪个索引性能更好。**多表关联（join）的时候，决定各个表的连接顺序**
* 执行器：优化器执行完就进入执行器，执行器就开始执行语句进行查询比对了，直到查询到满足条件的所有数据，然后进行返回。

**先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限会先查询缓存(MySQL8.0 版本以前)。**

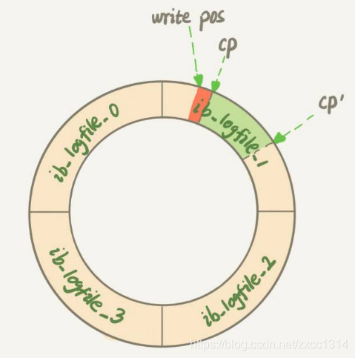
**如果没有缓存，分析器进行词法分析，提取 sql 语句中 select 等关键元素，然后判断 sql 语句是否有语法错误，比如关键词是否正确等等。**

**最后优化器确定执行方案进行权限校验，如果没有权限就直接返回错误信息，如果有权限就会调用数据库引擎接口，返回执行结果。**

日志系统

redo log

MySQL WAL 技术，先写日志，再写磁盘。保证掉电重启，数据不丢失（crash-safe）。  
redo log 是 InnoDB 引擎特有的日志。  
当记录更新时，Innodb 先记录 redo log 再更新内存，这时更新就算完成。引擎往往会在系统空闲时刷盘。



redo log 是实现了类似环形缓冲区，一个指针 write pos 是当前记录的位置，另一个指针 checkpoint 是当前要擦除的位置，write pos 和checkpoint 之间是空闲部分。如果 write pos 快追上 checkpoint 时，代表缓冲区快满了，需要暂停刷盘。（CP’见12讲）

innodb\_flush\_log\_at\_trx\_commit参数：

0：log buffer将每秒一次地写入log file中，并且log file的flush(刷到磁盘)操作同时进行。该模式下在事务提交的时候，不会主动触发写入磁盘的操作。

1：每次事务提交时MySQL都会把log buffer的数据写入log file，并且flush(刷到磁盘)中去，该模式为系统默认。

2：每次事务提交时MySQL都会把log buffer的数据写入log file，但是flush(刷到磁盘)操作并不会同时进行。该模式下，MySQL会每秒执行一次 flush(刷到磁盘)操作。

bin log（归档日志）

Server层日志。bin log 日志只能用于归档，没有crash-safe能力。  
三个用途:

1. 恢复：**利用bin log日志恢复数据库数据**
2. 复制：**主从同步**
3. 审计：通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入攻击

常见格式：



sync\_binlog参数：

0：当事务提交后，Mysql仅仅是将binlog\_cache中的数据写入binlog文件，但不执行fsync之类的磁盘 同步指令通知文件系统将缓存刷新到磁盘，而让Filesystem自行决定什么时候来做同步，这个是性能最好的。

n：在进行n次事务提交以后，Mysql将执行一次fsync之类的磁盘同步指令，同志文件系统将binlog文件缓存刷新到磁盘。

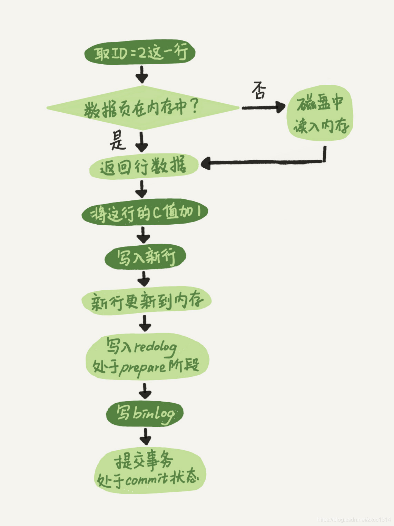
不同点:

redo log 是物理日志，记录的是“在某个数据页上做了什么修改”；binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这一行的 c 字段加 1 ”。

redo log 是循环写的，空间固定会用完；binlog 是可以追加写入的。binlog 文件到一定大小，会切换到下一个文件。

**update执行过程：**

mysql> update T set c=c+1 where ID=2;



两阶段提交

1 prepare阶段 2 写binlog 3 commit

当在2之前崩溃时

重启恢复：后发现没有commit，回滚。备份恢复：没有binlog 。

当在3之前崩溃

重启恢复：虽没有commit，但满足prepare和binlog完整，所以重启后会自动commit。备份：有binlog 。

1. SQL优化

**优化表结构**

1. 尽量使用数字型字段

若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

1. 尽可能的使用varchar代替char

变长字段存储空间小，可以节省存储空间

1. 当索引列大量重复数据时，可以把索引删除掉

比如有一列是性别，几乎只有男、女、未知，这样的索引是无效的

**优化查询**

1. 应尽量避免在where子句中使用!=或<>操作符
2. 应尽量避免在where子句中使用or来连接条件
3. 任何查询也不要出现select \*
4. 避免在where子句中对字段进行null值判断

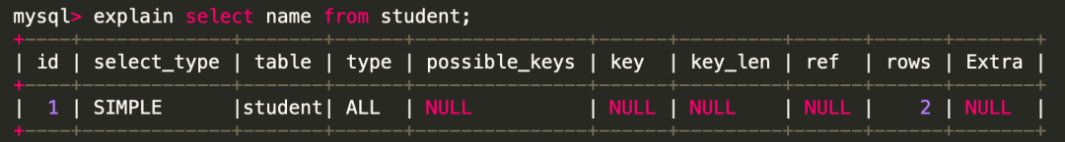
**索引优化**

1. 对作为**查询条件**和**order by**的字段建立索引
2. 避免建立过多的索引，**多使用组合索引**

**分库分表读写分离**

1. 怎么看执行计划（explain），如何理解其中各个字段的含义

在 select 语句之前增加 explain 关键字，会返回**执行计划**的信息。



（1）id 列：是 select 语句的序号，MySQL将 select 查询分为**简单查询**和**复杂查询**。

（2）select\_type列：表示对应行是是**简单还是复杂**的查询。

（3）table 列：表示 explain 的一行正在**访问哪个表**。

（4）type 列：**最重要的列之一**。表示**关联类型或访问类型**，即 MySql 决定如何查找表中的行。从最优到最差分别为：system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

（5）possible\_keys 列：显示**查询可能使用哪些索引**来查找。

（6）key 列：这一列显示 mysql 实际**采用哪个索引来优化对该表的访问**。

（7）key\_len 列：显示了mysql在**索引里使用的字节数**，通过这个值可以算出具体使用了索引中的哪些列。

（8）ref 列：这一列显示了在key列记录的索引中，表查找值所用到的列或常量，常见的有：const（常量），func，NULL，字段名。

（9）rows 列：这一列是 mysql 估计要读取并检测的行数，注意这个不是结果集里的行数。

（10）Extra 列：显示额外信息。比如有 Using index、Using where、Using temporary等。

1. undo log、redo log、bin log

undo log

重做日志保证了事务的持久性，保证能够在宕机后恢复事务的数据，那么另外一种情况就是事务在需要回滚的时候怎么办？这时候就是undo\_log的作用了，它保证了事务的一致性。

对于undo\_log来说，简单理解就是做了逆向操作。

比如insert一条数据，就对应生成delete，update语句则生成相反的更新语句，这样做到将数据修改回之前的状态。

redo log

redo\_log按照字面翻译称为重做日志，是InnoDB存储引擎特有的，用于保证事务的原子性和持久性。怎么理解呢？简单来说就是保存我们执行的更新语句的记录，如果服务器或者Mysql宕机，通过redo\_log可以恢复更新的数据。

按照上述流程来举例的话，比如update user set age=20 where id=1这样的简单更新SQL，我们不管执行引擎怎么拿到的数据，不管是从缓冲池拿的还是磁盘拿到的，这条现在数据都在缓冲池里面，然后去缓冲池的数据把age改成10。

缓冲池内存中的数据已经更新好了，那么接下来就该开始写redo\_log了，只是redo\_log也不是直接写文件的，一般都是这样对吧，直接写的话性能太差了，所以就有redo\_log\_buffer叫做redo\_log缓冲。



在写redo\_log的时候先把数据写到redo\_log缓冲区，然后异步写入磁盘，很显然，极端情况下会有丢失数据的可能。

控制这个刷盘策略的的参数叫做innodb\_flush\_log\_at\_trx\_commit。

这个参数有3个值：0|1|2，默认的话是1。

0代表提交事务时不会写入磁盘，这样的话性能当然最好，但是在Mysql宕机的情况会丢失上一秒的事务的数据。

1代表提交事务一定会进行一次刷盘，同步当然性能最差，但是也最安全。

2代表写入文件系统的缓存，不进行刷盘。这个选项性能略差于1，Mysql宕机的话对数据没有任何影响，只有在操作系统宕机才会丢失数据，这种情况下默认Mysql每秒会执行一次刷盘。

使用0或者2虽然提高了性能，但是变相的也丧失了事务的持久性。

bin log

binlog称为二进制日志，大家都很熟悉，记录了改变数据库的那些SQL语句，对于这里来说，更新语句当然是了。

通过不同于redo\_log是独属于存储引擎独有的东西，binlog则是Mysql本身产生的日志。

不同于redo\_log是物理日志，binlog和undo\_log都属于逻辑日志。

这有什么区别呢？

简单来说，逻辑日志可以认为就是存储的SQL本身，而物理日志看看redo\_log存储的是啥就知道了，关于page\_id页ID，offset偏移量啊这些东西，记录的是对页的修改。

另外物理日志可以保证幂等性，而逻辑日志则不一定能，除非本身SQL就是幂等的。

上面我们提到了redo\_log的刷盘策略，binlog就和它非常类似了，控制参数是sync\_binlog。

默认值为0，相当于是innodb\_flush\_log\_at\_trx\_commit的值为2，由文件系统控制，同样如果服务器宕机，binlog丢失，当然我们也可以改成1，就和redo\_log的效果是一样，每1次事务提交都同步写入磁盘。

1. 关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

我们平时写Sql时，都要养成用explain分析的习惯。慢查询的统计，运维会定期统计给我们

优化慢查询思路：

* 分析语句，是否加载了不必要的字段/数据
* 分析 SQL 执行句话，是否命中索引等
* 如果 SQL 很复杂，优化 SQL 结构
* 如果表数据量太大，考虑分表

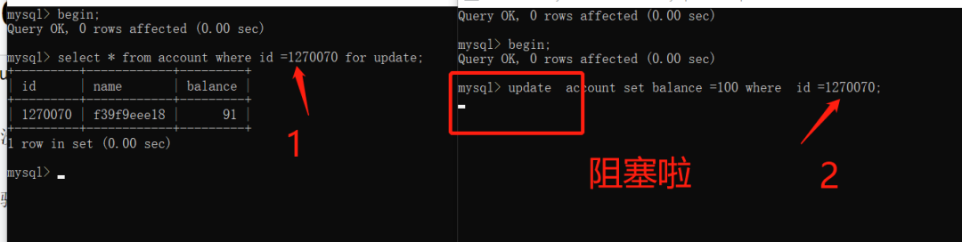
1. 数据库停掉重启需要进行什么操作？
2. MySql 的limit字段（limit 1000000加载很慢的话，你是怎么解决的呢？）
3. select for update有什么含义，会锁表还是锁行还是其他

#### select for update 含义

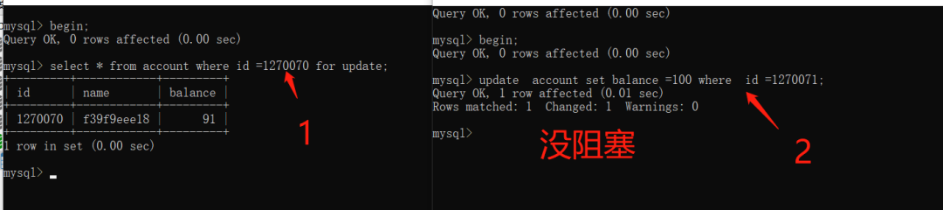
select查询语句是不会加锁的，但是select for update除了有查询的作用外，还会加锁呢，而且它是悲观锁哦。至于加了是行锁还是表锁，这就要看是不是用了索引/主键啦。

没用索引/主键的话就是表锁，否则就是是行锁

id为主键，select for update 1270070这条记录时，再开一个事务对该记录更新，发现更新阻塞啦，其实是加锁了。如下图：



我们再开一个事务对另外一条记录1270071更新，发现更新成功，因此，如果查询条件用了索引/主键，会加行锁~

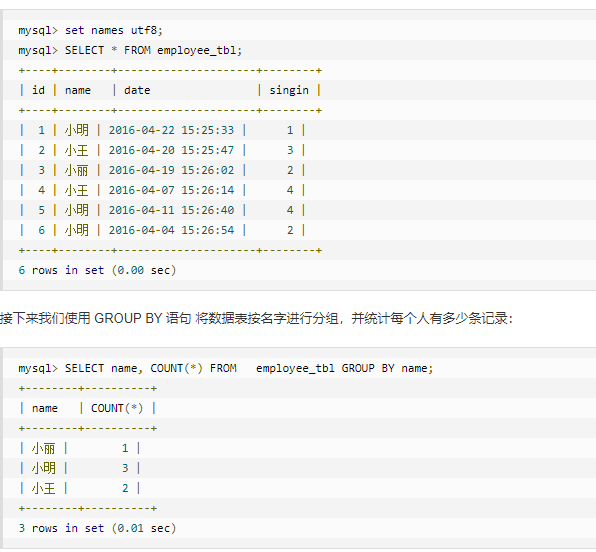


我们继续一路向北吧，换普通字段balance吧，发现又阻塞了。因此，没用索引/主键的话，select for update加的就是表锁



1. MySQL的GROUP BY 和 JOIN

GROUP BY 语句根据一个或多个列对结果集进行分组。



JOIN等同于INNER JOIN,即内连接，不满足ON条件的会直接过滤掉，不显示

多表关联查询



1. 数据库的索引类型

数据库的索引类型分为逻辑分类和物理分类

逻辑分类：

* 主键索引 当关系表中定义主键时会自动创建主键索引。每张表中的主键索引只能有一个，要求主键中的每个值都唯一，即不可重复，也不能有空值。
* 唯一索引 数据列不能有重复，可以有空值。一张表可以有多个唯一索引，但是每个唯一索引只能有一列。如身份证，卡号等。
* 普通索引 一张表可以有多个普通索引，可以重复可以为空值
* 全文索引 可以加快模糊查询，不常用

物理分类：

* 聚集索引（聚簇索引） 数据在物理存储中的顺序跟索引中数据的逻辑顺序相同，比如以ID建立聚集索引，数据库中id从小到大排列，那么物理存储中该数据的内存地址值也按照从小到大存储。一般是表中的主键索引，如果没有主键索引就会以第一个非空的唯一索引作为聚集索引。一张表只能有一个聚集索引。
* 非聚集索引 数据在物理存储中的顺序跟索引中数据的逻辑顺序不同。非聚集索引因为无法定位数据所在的行，所以需要扫描两遍索引树。第一遍扫描非聚集索引的索引树，确定该数据的主键ID，然后到主键索引（聚集索引）中寻找相应的数据。

1. 聚集索引与非聚集索引的区别

可以按以下四个维度回答：

（1）一个表中只能拥有一个聚集索引，而非聚集索引一个表可以存在多个。

（2）聚集索引，索引中键值的逻辑顺序决定了表中相应行的物理顺序；非聚集索引，索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。

（3）索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。

（4）聚集索引：物理存储按照索引排序；非聚集索引：物理存储不按照索引排序；

1. 说一下事务是怎么实现的

事务指的是满足ACID特性的一组操作，可以通过Commit提交一个事务，也可以使用Rollback进行回滚。

1. 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志（Undo Log）来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可

1. 一致性（Consistency）

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对同一个数据的读取结果都是相同的。

1. 隔离性（Isolation）

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

1. 持久性（Durability）

一旦事务提交，则其所做的修改将永远保存到数据库中。即时系统发生崩溃，事务执行的结果也不能丢失。

系统发生崩溃可以用重做日志（Redo Log）进行恢复，从而实现持久性。与回滚日志记录数据的逻辑修改不同，重做日志记录的是数据页的物理修改。

事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：

<https://guobinhit.blog.csdn.net/article/details/61200815?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7Edefault-7.base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7Edefault-7.base>

重点！

事务就是一组逻辑操作的集合。实现事务就是要保证可靠性和并发隔离，或者说，能够满足ACID特性的机制。而这些主要是靠日志恢复和并发控制实现的。

* 日志恢复：数据库里有两个日志，一个是redo log，一个是undo log。redo log记录的是已经成功提交的事务操作信息，用来恢复数据，保证事务的持久性。undo log记录的是事务修改之前的数据信息，用来回滚数据，保证事务的原子性。
* 并发控制：并发控制主要靠读写锁和MVCC（多版本并发控制）来实现。读写锁包括共享锁和排他锁，保证事务的隔离性。**MVCC通过为数据添加时间戳**来实现。

1. 事务隔离：为什么你改了我还看不见？（重点如何实现事物的隔离性）

回滚段

rollback segment称为回滚段，每个回滚段中有1024个undo log segment。每个undo操作在记录的时候占用一个undo log segment。

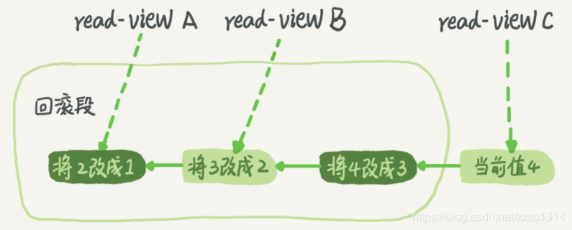
undo log有两个作用：**提供回滚**和**多个行版本控制(MVCC)**。

在数据修改的时候，不仅记录了redo，还记录了相对应的undo，如果因为某些原因导致事务失败或回滚了，可以借助该undo进行回滚。

undo log和redo log记录物理日志不一样，它是逻辑日志。可以认为当delete一条记录时，undo log中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。

事务隔离的实现

以可重复读（RR）为例，每条记录在更新的时候都会同时记录一条回滚操作。



不同时刻启动的事务会有**不同的 read-view**。**同一条记录在系统中可以存在多个版本，就是数据库的多版本并发控制（MVCC）**。

当系统里没有比这个回滚日志更早的 read-view 的时候，回滚日志会被删除。所以要避免长事务。

1. 事务是什么锁的种类和区别
2. 全局锁和表锁:给表加个字段怎么有这么多障碍

mysql锁大致可以分成全局锁、表级锁和行锁三类

全局锁全局锁的典型使用场景是，做全库逻辑备份。FTWRL命令：Flush tables with read lock;

官方自带的逻辑备份工具是 mysqldump，当 mysqldump 使用参数–single-transaction 的时候，导数据之前就会启动一个事务，来确保拿到一致性视图。

但当引擎不支持事务时，只能使用FTWRL 命令了。不推荐不使用 set global readonly=true，readonly会被其他逻辑使用（比如判断主从），readonly发生异常会保持该状态。

表级锁

MySQL 里面表级别的锁有两种：一种是表锁，一种是元数据锁（meta data lock，MDL)。

表锁的语法是 lock tables … read/write。

MDL不需要显式使用，在访问一个表的时候会被自动加上。

当对一个表做**增删改查操作的时候，加 MDL 读锁**；当要**对表做结构变更操作的时候，加 MDL 写锁**。



当一个长事务还没提交，进行表结构变更操作，会导致后面的事务block。当客户端有重试机制时，新起session请求，会导致库的线程很快就会爆满。

如何安全地给小表加字段？

避免长事务。

在 alter table 语句里面设定等待时间。

MariaDB 已经合并了 AliSQL 的这个功能，所以这两个开源分支目前都支持 DDL NOWAIT/WAIT n 这个语法。

1. 行锁功过：怎么减少行锁对性能的影响

行锁

MySql 行锁由引擎层实现（引擎层）

两阶段锁

行锁需要事务结束时才释放，这就是两阶段锁

所以需要合理安排事务中sql执行顺序，尽量把容易冲突的更新语句放在后面

死锁和死锁检测

1. 设置超时时间，innodb\_lock\_wait\_timeout
2. 死锁检测，发现死锁主动回滚某个事物，innodb\_deadlock\_detect默认on. 假设1000个同时更新一行，则死锁检测操作就是100万量级的。即使没有死锁，检测也会消耗大量的CPU资源

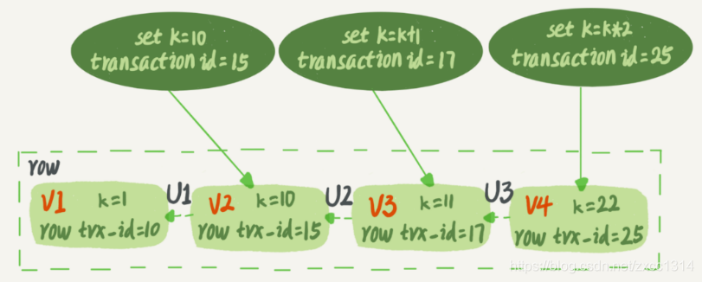
解决方案：

1. 业务不会出现死锁，可以临时关闭
2. 在客户端控制并发
3. 修改MySql源码，并发进入引擎之前排队
4. 将一行数据改为多行，如将一个余额账户分为多个，但在数据减少操作时需考虑小于0的情况
5. 事务到底是隔离的还是不隔离的？

Innodb里面每个事务都有唯一的事务ID，叫做transaction id。它是在事务开始的时候向innodb的事务系统申请的，是按申请顺序严格递增。

而每行数据也都有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本，并且把transaction id赋值给这个数据版本的事务ID，记为row trx\_id。

也就是说，数据表中的一行记录，其实可能有多个版本（row），每个版本有自己的row trx\_id



上图中三个虚线箭头就是undo log

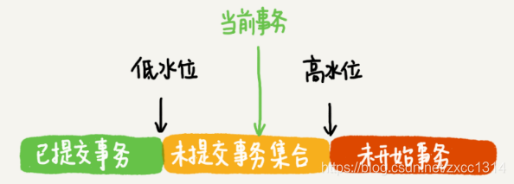
某个事务建立快照，只需根据transaction id。只认事务启动时小于数据版本的数据，**除自己更新的数据**。

快照实现

InnoDB在每个事务启动瞬间，构造了数组保存了当前启动但未提交的事务ID。

数组ID最小值为低水位，当前系统最大事务ID+1为高水位。

数组和高水位，组成了当前事务的一致性事务（read-view）。



黄色部分需分为以下两种情况，因为有可能大于低水位的某个事务已经提交：

若 row trx\_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见；

若 row trx\_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

select read-view创建在03 | 事务隔离中提过了，就不写了。

更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”（current read）。

如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。

不同隔离级别：

对于可重复读，查询**只承认在事务启动前就已经提交完成的数据**

对于读提交，查询**只承认在语句启动前就已经提交完成的数据**

而当前读，总是读取已经提交完成的最新版本。

1. MySQL怎么建立索引，怎么建立主键索引，怎么删除索引？

MySQL建立索引有两种方式：用alter table或者create index。

alter table table\_name add primary key(column\_list) #添加一个主键索引

alter table table\_name add index (column\_list) #添加一个普通索引

alter table table\_name add unique (column\_list) #添加一个唯一索引

create index index\_name on table\_name (column\_list) #创建一个普通索引

create unique index\_name on table\_name (column\_list) #创建一个唯一索引

Mysql删除索引同样也有两种方式：alter table 和 drop index

alter table table\_name drop index index\_name #删除一个普通索引

alter table table\_name drop primary key #删除一个主键索引

drop index index\_name on table table\_name

1. MySQL为什么有时候会选择错索引？

平常不断地删除历史数据和新增数据的场景，mysql有可能会选错索引。

优化器的逻辑

索引选择异常和处理

force index 强行选择一个索引

修改语句，引导Mysql使用我们期望的索引

新建索引，或者删除误用的索引

1. 索引的优缺点，什么时候使用索引，什么时候不能使用索引（重点）

https://www.cnblogs.com/wezheng/p/8399305.html

1、主键自动建立唯一索引；

2、经常作为查询条件在WHERE或者ORDER BY 语句中出现的列要建立索引；

3、作为排序的列要建立索引；

4、查询中与其他表关联的字段，外键关系建立索引

5、高并发条件下倾向组合索引；

6、用于聚合函数的列可以建立索引，例如使用了max(column\_1)或者count(column\_1)时的column\_1就需要建立索引

索引的缺点：

1、创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

2、索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

3、当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

哪些列不适合建索引？

1.很少查询的列

2.更新很频繁的列

3.数据值的取值比较少的列（比如性别）

1. 数据库索引的分类（功能分类和底层分类）索引的结构，以及为什么，索引的优劣势

前文：我以为我对MySQL索引很了解 ，直到我遇到了阿里的面试官。

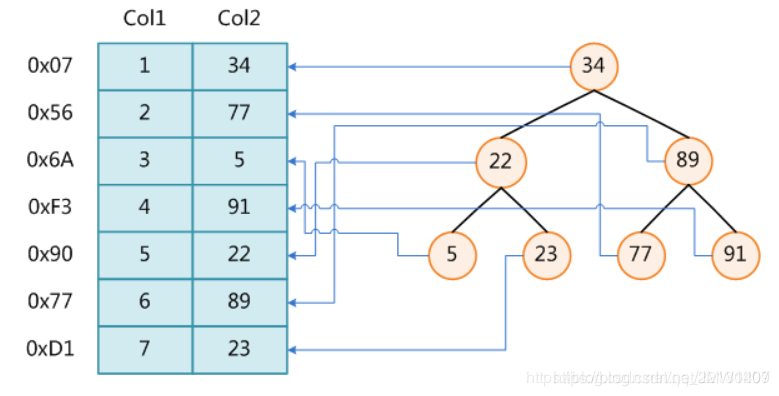
什么是索引：索引其实就是一种数据结构，能够帮助我们快速的检索数据库中的数据。

对于一本书而言，通过目录里，我们可以很快的找到自己想要的章节对应的位置。同样，在数据十分庞大的时候，索引可以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。

索引的定义和特征

定义：

1. 索引是一个排序的列表 ，在这个列表中存储着索引的值和包含这个值得数据所在行的物理地址
2. 索引是一种数据结构，数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B+树
3. 索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录
4. 索引是一个文件，它是要占物理空间的。



根据col2列建立索引，key是索引字段的值，value是值所在的磁盘文件地址，例如77就是一个key索引，0x56就是对应的value，通过这个地址，找到你需要查询的数据。

特征：

1. 索引的好处
2. 提高数据检索速度，降低数据库IO成本。就是通过缩小表中需要查询的记录的数目从而加快搜索速度。
3. 降低数据排序的成本，降低CPU消耗，之所以查的快，是因为先将数据排好序了。
4. 索引的坏处
5. 占用存储空间：索引实际上是一张表，记录了主键和索引字段，一般以索引文件的形式存储在磁盘上。（数据表和每一个索引都要占一定的空间）
6. 降低更新表的速度：表的速度发生了变化时，对应的索引也需要一起变更（所以不适用增删改频繁的数据库）
7. 创建索引和维护索引要消耗时间

数据量少的不适合加索引、更新比较频繁也不适合加索引、区分度低也不适合加索引

索引的分类（功能上分类）

从功能上来说，分为6种：普通索引、唯一索引、主键索引、复合索引、外键索引、全文索引

1. 普通索引：最基本的索引，没有任何约束



1. 唯一索引：与普通索引类似，但具有唯一性约束。在表上一个或者多个字段组合建立的索引，这个字段的值组合起来在表中不可以重复。一张表可以建立任意多个唯一索引，但一般只建立与一个。



1. 主键索引：特殊的唯一索引，不允许有空值。

唯一索引允许null值，而主键列不允许为null值，一张表最多建立一个主键，也可以不建立主键



1. 复合索引：将多个列组合在一起创建的索引，可以覆盖多个列。

如：姓-名-电话号码。电话薄中的内容先按姓氏的拼音排序，相同姓氏再按名字的拼音排序，这相当于（姓、名）上建立了一个复合索引



1. 外键索引：只InnoDB类型的表才可以使用外键索引，保证数据的一致性、完整性和实现级联操作。
2. 全文索引：MySQL自带的全文索引只能用于InnoDB、MyISAM，并且只能对英文进行全文检索，一般使用全文索引引擎（ES，Solr）

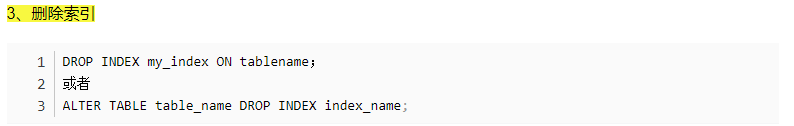


注意：主键就是唯一索引，但是唯一索引不一定是主键，唯一索引可以为空，但是空值只能有一个，主键不能为空。

MySQL下索引的基本操作







索引的底层实现原理：

可以从几个维度去看这个问题，**查询是否够快**，**效率是否稳定，存储数据多少，以及查找磁盘次数**

1. 二叉查找树（BST）：左子树节点均小于父节点，右子树节点均大有父节点

查找时间复杂度为O(logn)

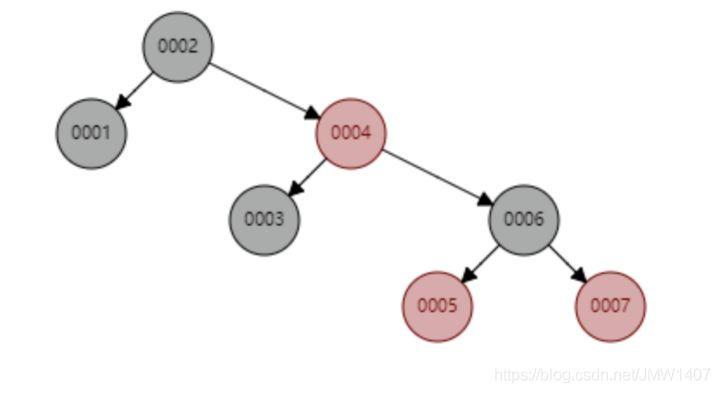
不足之处：主键一般默认都是自增的，产生不平衡状态

普通二叉查找树有个致命缺点：极端情况下会退化为线性链表，二分查找也会退化为遍历查找，时间复杂度退化为O（n），检索性能下降

1. 红黑树

红黑树：这是一颗会自动调整树形态的树结构，比如当二叉树处于一个不平衡状态时，红黑树就会自动左旋右旋以及节点变色，调整树的形态，使其保持基本的平衡状态（时间复杂度为O（logN）），也就保证了查找效率不会明显降低。

如从 1 到 7升序插入数据节点，如果是普通的二叉查找树则会退化成链表，但是红黑树则会不断调整树的形态，使其保持基本平衡状态。如下图所示。下面这个红黑树下查找 id=7 的所要比较的节点数为 4，依然保持二叉树不错的查找效率。



不足之处：红黑树顺序插入1~16个节点时，查找id=16需要比较的节点树为6次。观察一下这个树的形态，是不是当数据是顺序插入时，树的形态就一直处于“右倾”的趋势呢？

从根本上看，红黑树并没有完全解决二叉查找树逐渐右倾这个趋势，虽然没有二叉查找树退化为线性链表那么夸张，但是数据库中的基本主键自增操作，主键一般都是数百万数千万，效率也很低。

1. AVL树

自平衡二叉树。因为AVL是个绝对平衡的二叉树，左右子树高度差小于等于1

因此在调整二叉树的形态上消耗的性能会更多。当树的某个位置被删除节点，需要经过一系列左旋右旋操作维持高度平衡。

查找性能O（logN），不存在极端的低效查找情况

* 从查找效率而言，AVL 树查找的速度要高于红黑树的查找效率（AVL 树是 4 次比较，红黑树是 6 次比较）。
* 从树的形态看来，AVL 树不存在红黑树的“右倾”问题。

不足之处：数据库查询数据的瓶颈在于磁盘 IO，就是从磁盘读取 1B 数据和 1KB 数据所消耗的时间是基本一样的。

AVL 树的每一个树节点只存储了1个数据，我们一次磁盘 IO 只能取出来一个节点上的数据加载到内存里，那比如查询 id=7 这个数据我们就要进行磁盘 IO 三次，这是多么消耗时间的。设计数据库索引时首先考虑怎么减少磁盘 IO 的次数。

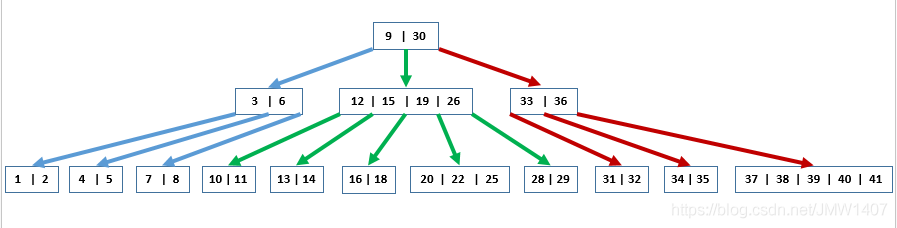
综上所述：  
BST、红黑树、AVL突出的问题，我们需要维持基本结构平衡、需要考虑树的深度过深，可以在一个树节点上尽可能多地存储数据，一次磁盘 IO 就多加载点数据到内存，这就是 B 树，B+树的的设计原理了。

1. B树（多路查找二叉树）

B树是为实现高效的磁盘存取而设计的多叉平衡搜索树

就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下（为什么会出现这种情况，待会在外部存储器-磁盘中有所解释），那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的想法就是：采用多叉树结构

B-Tree:一种自平衡的树（所有的叶子节点拥有相同的高度）类型的数据结构。但是和其它树比如红黑树，AVL树只有两个孩子：左孩子和右孩子不同，B-Tree的子节点多余或者等于两个孩子



总结：

优秀的检索速度，时间复杂度：B树的查找性能等于O（h\*logN）,其中h为树高，n为每个节点关键词的个数；

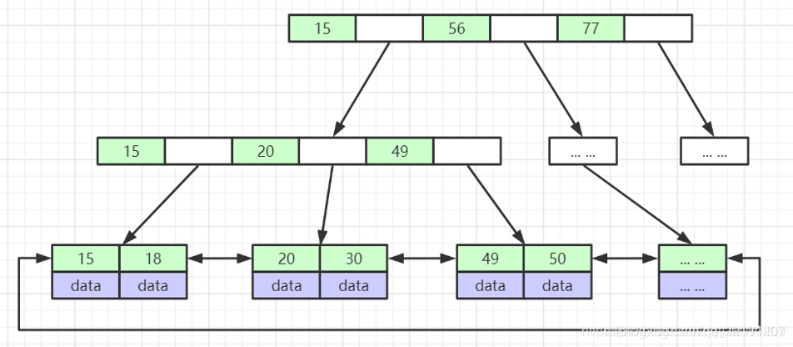
尽可能少的磁盘IO，加快了检索速度；

MySQL里也没有用这种，而是用到了它的变种B+树，具体原因下面介绍完B+树进行对比

1. **B+树**

**B+树主体特征和B树一样，主要不一样的特征如下：**

* 1、有n棵子树的结点中含有n个关键字，每个关键字不保存数据，只用来索引，所有数据都保存在叶子节点，可以让每个节点保存更多的索引。（特别重要）
* 2、所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
  + 也就是所有数据（这里其实是数据的地址）都放在叶子节点，并且所以叶子节点用指针（双箭头）串起来，按顺序排列，奠定了范围查找的基础（特别重要）
* 3、所有的非终端结点可以看成是索引部分，结点中仅含其子树(根结点)中的最大(或最小)关键字。



**已上图为例简述B+树的查找过程，比如查找30：**

**1、从根节点开始。进行第1次磁盘IO，把它load到内存去。拿30去内存里做比对，通过二分查找，快速定位，发现在15和56之间**

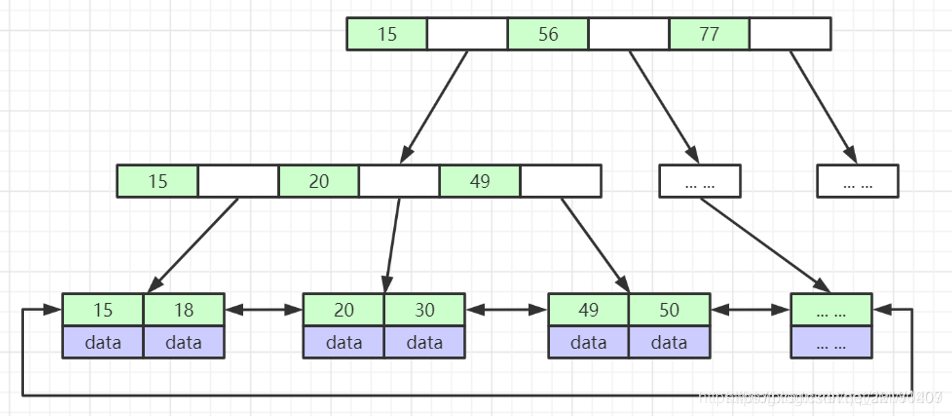
**2、进行第2次磁盘IO找到这一页数据，我们把它加载到内存，发现介于20-49之间**

**3、进行第3次磁盘IO，把这个叶子结点load到内存进行比对，发现找到，再把30对应的磁盘文件地址去磁盘上找这一行数据。**

或许你会问，可不可以把所有索引元素的数据都放到根节点上呢？这样岂不是1次磁盘IO，加载到内存中比对即可吗？

答：肯定是不行的，因为这样在数据量巨大的情况下，会瞬间使内存使用率过高，把内存撑爆，并且也不会很快，而且1次磁盘IO也弄不了这么多数据。

1. MySQL索引为什么选择B+树
2. 树的高度决定：因为B+树节点不放数据，所以存放更多的索引节点

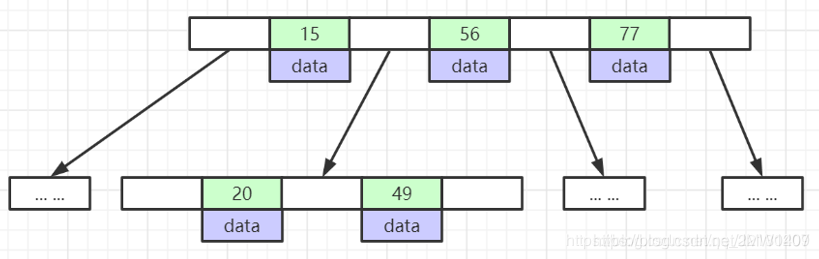


上图是B+树的结构：

mysql认为每个结点就是1个页，默认1页的大小是16KB。那么我们可以估算下3层高度能存储多少数据。

一张表假设用bigint做主键，8B大小，1个地址大概是6B。那么1页是16KB，可以存放16KB/14B=1170个结点。第二层同理1170个，那么叶子结点，最多也就1KB，这个叶结点16个。那么一共放1170x1170x16=2190万。这样的数据量也就用了3行。经过了3次磁盘IO就找到了元素。

mysql会把根节点常驻内存，那就更快了。高版本会把所有非叶子结点加载到内存，那么可能也就1次磁盘IO把叶子结点加到内存。



**我们估算B树存储2000多万数据得多高？**  
每个结点16个元素，默认每个结点还是16KB，1个结点1KB大小，16/1 = 16，16^7 = 268 435 456，可以看到存储相同数据量，B+树仅用3层,B树最少7层。

1. B+树可以很好的支持范围查找

假设我们找大于20小于50范围内的数据，从根节点出发，找到叶子结点的20，沿着指针方向找到50就可以了。

B树没有指针，找完20，会继续从根节点往下找，效率会低得多。

1. 哈希表

不足之处：范围查找（致命之处）用哈希算法实现的索引，范围查找怎么做呢？简单的思路：一次把所有数据找出来加载到内存，然后再在内存里筛选筛选目标范围内的数据。但是这个范围查找的方法也太笨重了，没有一点效率而言。

**索引的分类（实现上分类）**

为了更好理解一下内容，首先对数据库存储数据文件的形式简要说明。

Mysql 底层数据引擎以插件形式设计，最常见的是 Innodb 引擎和 Myisam 引擎。

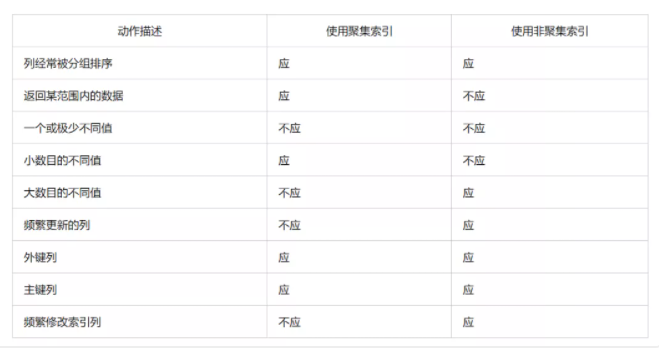


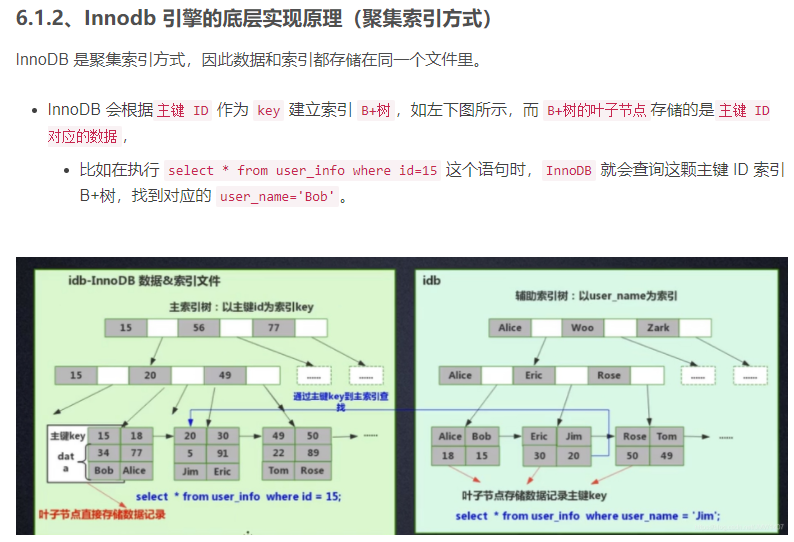
MyISAM 引擎把数据和索引分开了，一人一个文件，这叫做非聚集索引方式；  
Innodb 引擎把数据和索引放在同一个文件里了，这叫做聚集索引方式。

聚集索引：

定义：

聚集（clustered）索引，也叫聚簇索引。





1、当我们为表里某个字段加索引时 InnoDB 会怎么建立索引树呢？

比如我们要给 user\_name 这个字段加索引：

InnoDB 会建立 user\_name 索引 B+树，节点里存的是 user\_name 这个 KEY，叶子节点存储的数据的是主键 KEY。

叶子存储的是主键 KEY！！！拿到主键 KEY 后，InnoDB 才会去主键索引树里根据刚在 user\_name 索引树找到的主键 KEY查找到对应的数据。

2、为什么 InnoDB 只在主键索引树的叶子节点存储了具体数据，但是其他索引树却不存具体数据呢，而要多此一举先找到主键，再在主键索引树找到对应的数据呢?

因为 InnoDB 需要节省存储空间。一个表里可能有很多个索引，InnoDB都会给每个加了索引的字段生成索引树，如果每个字段的索引树都存储了具体数据，那么这个表的索引数据文件就变得非常巨大（数据极度冗余了）。

从节约磁盘空间的角度来说，真的没有必要每个字段索引树都存具体数据，通过这种看似“多此一举”的步骤，在牺牲较少查询的性能下节省了巨大的磁盘空间，这是非常有值得的。

3、 MyISAM 查询性能比 InnoDB 更高？

**聚簇索引（InnoDB）**

辅助索引的叶子节点的data存储的是主键的值；

主索引的叶子节点的data存储的是数据本身，也就是说数据和索引存储在一起；

索引查询到的地方就是数据（data）本身，那么索引的顺序和数据本身的顺序就是相同的；

**非聚簇索引（MyISAM ）**

主索引和辅助索引的叶子节点的data都是存储的数据的物理地址，也就是说索引和数据并不是存储在一起的，数据的顺序和索引的顺序并没有任何关系，也就是索引顺序与数据物理排列顺序无关。

综上特点：

* MyISAM 直接找到物理地址后就可以直接定位到数据记录；
* InnoDB 查询到叶子节点后，还需要再查询一次主键索引树，才可以定位到具体数据。
* 等于 MyISAM 一步就查到了数据，但是 InnoDB 要两步，那当然 MyISAM 查询性能更高。

非聚集索引方式



非聚集索引，分成普通索引，唯一索引，全文索引

* MyISAM 用的是非聚集索引方式，即数据和索引落在不同的两个文件上。
* MyISAM 在建表时以主键作为 KEY 来建立主索引 B+树，树的叶子节点存的是对应数据的物理地址。
  + 拿到这个物理地址后，就可以到 MyISAM 数据文件中直接定位到具体的数据记录了

当我们为某个字段添加索引时，我们同样会生成对应字段的索引树，该字段的索引树的叶子节点同样是记录了对应数据的物理地址，然后也是拿着这个物理地址去数据文件里定位到具体的数据记录。

1. 深入浅出索引（上、下）

索引的常见模型

哈希表，不适合做区间搜索。

有序数组，只适合静态数据，插入麻烦。

二叉搜索树，N叉树。

<https://zhuanlan.zhihu.com/p/29118331>

InnoDB 的索引模型

在 MySQL 中，索引是在存储引擎层实现的。

以主键顺序存在B+树中。

主键索引（聚簇索引） 的叶子节点存的是整行数据。主键查询主需要扫描主键索引。

非主键索引（二级索引）的叶子节点内容是主键的值。通过二级索引需要扫描二级索引树，找到主键后再扫描主键索引。该过程称为回表。

索引维护

当插入到索引树最后，只需直接插入。

但当插入到索引树中间，需要逻辑上挪动后面的数据，空出位置，并且当数据页满时，需要申请一个新的数据页，然后挪动部分数据过去（页分裂）。

当相邻两个页由于删除了数据，利用率很低之后，会将数据页做合并。

自增索引（追加操作，都不涉及到挪动其他记录，也不会触发叶子节点的分裂）

业务逻辑的字段做主键，则往往不容易保证有序插入，这样写数据成本相对较高。

二级索引的叶子节点为主键，业务字段做主键时会占大量存储空间。

什么时候可以使用业务字段做主键？ 只有一个索引；该索引必须是唯一索引。

覆盖索引当查询值已经在二级索引上时，不需要回表。最左前缀原则联合索引合理安排顺序，可以少维护索引，或者减少存储空间。

索引下推MySQL 5.6 引入的索引下推优化，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

1. 索引失效可能会导致什么？索引为什么失效？

* 查询条件包含or，可能导致索引失效
* 如何字段类型是字符串，where时一定用引号括起来，否则索引失效
* like通配符可能导致索引失效。
* 联合索引，查询时的条件列不是联合索引中的第一个列，索引失效。
* 在索引列上使用mysql的内置函数，索引失效。
* 对索引列运算（如，+、-、\*、/），索引失效。
* 索引字段上使用（！= 或者 < >，not in）时，可能会导致索引失效。
* 索引字段上使用is null， is not null，可能导致索引失效。
* 左连接查询或者右连接查询查询关联的字段编码格式不一样，可能导致索引失效。
* mysql估计使用全表扫描要比使用索引快,则不使用索引。

1. Hash索引和B+树索引区别是什么？你在设计索引是怎么抉择的？

* B+ 树可以进行范围查询，Hash 索引不能。
* B+ 树支持联合索引的最左侧原则，Hash 索引不支持。
* B+ 树支持 order by 排序，Hash 索引不支持。
* Hash 索引在等值查询上比 B+ 树效率更高。
* B+ 树使用 like 进行模糊查询的时候，like 后面（比如%开头）的话可以起到优化的作用，Hash 索引根本无法进行模糊查询。

1. 什么是最左前缀原则？什么是最左匹配原则？

最左前缀原则，就是最左优先，在创建多列索引时，要根据业务需求，where 子句中使用最频繁的一列放在最左边。

当我们创建一个组合索引的时候，如 (a1,a2,a3)，相当于创建了（a1）、(a1,a2)和(a1,a2,a3)三个索引，这就是最左匹配原则。

1. 数据库自增主键可能遇到什么问题？

* 使用自增主键对数据库做分库分表，可能出现诸如主键重复等的问题。解决方案的话，简单点的话可以考虑使用UUID哈
* 自增主键会产生表锁，从而引发问题
* 自增主键可能用完问题。

1. MySql中的数据是存在哪的？又是如何存储的呢？

关于Mysql这种关系型数据库，里面保存的数据最终都是要持久化到磁盘文件上面的。磁盘文件里存放的物理格式就是数据页（关于数据页，如果不太理解先忽略，后续文章单独介绍），数据页中存放的是一行一行的记录，但是对于数据页中的每一行数据他又是怎么存储的呢？

MySQL中存储有3种：

**server层格式：**与存储引擎无关，Binlog存储常用的一种 (Bin Log 我们前面已经详细介绍过了，这个是MySql主从复制的一个很重要的文件)**索引元组格式：**InnoDB存取过程记录的中间状态，是InnoDB在内存中存储的格式 （换句话说我们的增删改的操作都是在内存中执行的，这个只是一种临时状态）**物理存储格式：**记录在物理页面中的存储格式，即compact格式，与索引元组格式一一对应。（这个是数据在磁盘存储的真正的格式）

InnoDB 储存引擎支持有四种行储存格式：

COMPACT、Redundant、Dynamic 和 COMPRESSED。默认为COMPACT。

略

1. MySQL中使用什么存储引擎

InnoDB、MyISAM

1. Mysql的两种存储引擎——InnoDB、MyISAM（以插件的形式设计、使用场景）

区别：

* + - 1. InnoDB支持事务，MyISAM不支持，对于innodb每一条sql语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条sql语句放在begin和commit之间，组成一个事务
      2. Innodb支持外键，而myisam不支持。对一个包含外键的innodb表转为myisam会失败
      3. Innodb是聚集索引，使用B+ Tree作为索引结构，数据文件是和（主键）索引绑定在一起的（表数据文件本身就是按B+ Tree组织的一个索引结构），必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询。先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键过大，其它索引都会很大。

  MyISAM是非聚集索引，也是使用B+Tree作为索引结构，索引和数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。       也就是说：InnoDB的B+树主键索引的叶子节点就是数据文件，辅助索引的叶子节点是主键的值；而MyISAM的B+树主键索引和辅助索引的叶子节点都是数据文件的地址指针。

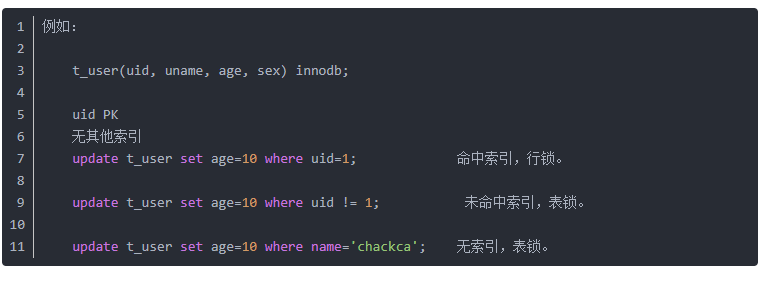
* + - 1. Innodb不保存表的具体行数，执行select count(\*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快（注意不能加有任何WHERE条件）

那么为什么innodb没有了这个变量呢？

因为InnoDB的事务特性，在同一时刻表中的行数对于不同的事务而言是不一样的，

* + - 1. **Innodb不支持全文索引，而MyISAM支持全文索引，在涉及全文索引领域的查询效率上MyISAM速度更快高；PS：5.7以后的InnoDB支持全文索引了**
      2. **MyISAM表格可以被压缩后进行查询操作**
      3. Innodb支持表、行（默认）级锁，而myisam支持表级锁

Innodb的行锁是实现在索引上的，而不是锁在物理行记录上。潜台词是，如果访问没有命中索引，也无法使用行锁，将要退化为表锁。



* + - 1. Innodb表必须具有唯一索引（如主键）（用户没有指定的话自己找/生产一个隐藏列Row\_id来充当默认主键），而myisam可以没有
      2. Innodb存储文件有frm、ibd，而myisam是frm、myd、myi

Innodb：frm是表定义文件，ibd是数据+索引文件

Myisam：frm是表定义文件，myd是数据文件，myi是索引文件

使用场景：

1. 是否要支持事务，如果要请选择innodb，如果不需要可以考虑myisam
2. 如果表中绝大多数都只是查询，可以考虑myisam，如果既有读也有写，请使用innodb
3. 系统崩溃后，myisam恢复起来更困难，能否接受
4. **MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。**

Innodb为什么推荐使用自增ID作为主键：

自增ID可以保证每次插入时，B+索引是向右扩展的，可以避免B+树频繁合并和分裂。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差。

Innodb引擎的四大特性：

插入缓冲、二次写、自适应哈希索引、预读

1. InnoDB中B+树有几层



这里我们先假设 B+ 树高为 2，即存在一个根节点和若干个叶子节点，那么这棵 B+ 树的存放总记录数为：**根节点指针数 \* 单个叶子节点记录行数。**

上文我们已经说明单个叶子节点（页）中的记录数 =16K/1K=16。（这里假设一行记录的数据大小为 1k，实际上现在很多互联网业务数据记录大小通常就是 1K 左右）。

那么现在我们需要计算出非叶子节点能存放多少指针？

其实这也很好算，我们假设主键 ID 为 bigint 类型，长度为 8 字节，而指针大小在 InnoDB 源码中设置为 6 字节，这样一共 14 字节，我们一个页中能存放多少这样的单元，其实就代表有多少指针，即 16384/14=1170。

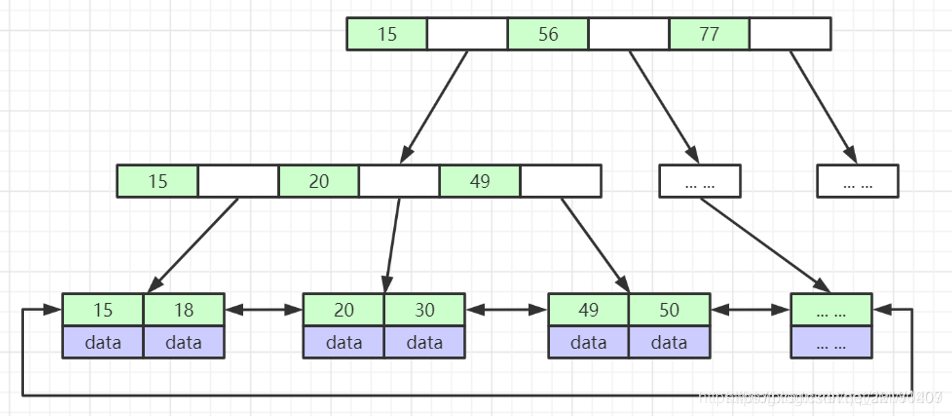
那么可以算出一棵高度为 2 的 B+ 树，能存放 1170\*16=18720 条这样的数据记录。

根据同样的原理我们可以算出一个高度为 3 的 B+ 树可以存放： 1170*1170*16=21902400 条这样的记录。

**所以在 InnoDB 中 B+ 树高度一般为 1-3 层，它就能满足千万级的数据存储。**

**在查找数据时一次页的查找代表一次 IO，所以通过主键索引查询通常只需要 1-3 次 IO 操作即可查找到数据。**

1. 为什么MySQL使用B+树
2. 树的高度决定：因为B+树节点不放数据，所以存放更多的索引节点

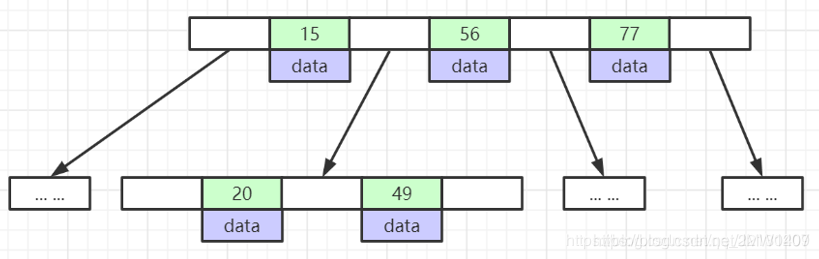


上图是B+树的结构：

mysql认为每个结点就是1个页，默认1页的大小是16KB。那么我们可以估算下3层高度能存储多少数据。

一张表假设用bigint做主键，8B大小，1个地址大概是6B。那么1页是16KB，可以存放16KB/14B=1170个结点。第二层同理1170个，那么叶子结点，最多也就1KB，这个叶结点16个。那么一共放1170x1170x16=2190万。这样的数据量也就用了3行。经过了3次磁盘IO就找到了元素。

mysql会把根节点常驻内存，那就更快了。高版本会把所有非叶子结点加载到内存，那么可能也就1次磁盘IO把叶子结点加到内存。



**我们估算B树存储2000多万数据得多高？**  
每个结点16个元素，默认每个结点还是16KB，1个结点1KB大小，16/1 = 16，16^7 = 268 435 456，可以看到存储相同数据量，B+树仅用3层,B树最少7层。

1. B+树可以很好的支持范围查找

假设我们找大于20小于50范围内的数据，从根节点出发，找到叶子结点的20，沿着指针方向找到50就可以了。

B树没有指针，找完20，会继续从根节点往下找，效率会低得多。

1. 隔离级别

什么是事务的隔离性呢？

隔离性是指，多个用户的并发事务访问同一个数据库时，一个用户的事务不应该被其他用户的事务干扰，多个并发事务之间要相互隔离。

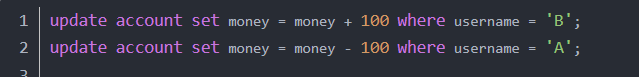
如果不考虑隔离性，会发生什么事呢？

1. 脏读

脏读是指一个事务读取了未提交事务执行过程中的数据。

当一个事务的操作正在多次修改数据，而在事务还未提交的时候，另外一个并发事务来读取了数据，就会导致读取到的数据并非是最终持久化数据之后的数据，这个数据就是脏读的数据。

最典型的例子就是银行转账，从A账户转账100到B账户，脚本命令为



在这个事务执行过程中，另外一个事务读取结果发现B账户中的钱已经到账，提示B钱已到账，B就进行下一步的操作。但是最终转账事务失败，导致操作回滚。实际上B并未受到钱，但是进行了下一步操作，造成了损失，这就是脏读。

1. 不可重复读

不可重复读是指对于数据库中的某个数据，一个事务执行过程中多次查询返回不同查询结果，这就是在事务执行过程中，数据被其他事务提交修改了。

不可重复度同脏读的区别在于，脏读是一个事务读取了另一未完成的事务执行过程中的数据，而不可重复读是一个事务的执行过程中，另一事务提交并修改了当前事务正在读取的数据。在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据

1. 幻读

在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。新增的记录。

数据库事务的隔离级别有4个，有低到高依次为

Read uncommitted、Read committed、Repeatable read、Serializable

这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。



1. 读未提交（Read uncommitted）
2. **已提交读（Read commited）**
3. **可重复读（repeatable read）**
4. 可串行化（Serializable）

<https://blog.csdn.net/FG2006/article/details/6937413?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-3.control&dist_request_id=1328767.60839.16176261993584247&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-3.control>（故事场景）

1. MySql如何解决死锁问题

遇到过。我排查死锁的一般步骤是酱紫的：

（1）查看死锁日志 show engine innodb status;

（2）找出死锁Sql

（3）分析sql加锁情况

（4）模拟死锁案发

（5）分析死锁日志

（6）分析死锁结果

1. 间隙锁、行锁、表锁

Mysql中表锁分两种：表共享读锁、表独占写锁

1. 封锁

封锁粒度：

Mysql提供了两种封锁粒度：行级锁以及表级锁

MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；BDB存储引擎采用的是页面锁（page-level locking），但也支持表级锁；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。

应该尽量之锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获得锁、释放锁、以及检查锁的状态）都会增加系统开销。因此封锁粒度越小，系统开销就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

封锁类型：

1. 读写锁

互斥锁（exclusive）：简称为X锁，又称写锁。

共享锁（shared）：简称为S锁，又称读锁。

有以下两个规定：

一个事务对数据对象A加了X锁，就可以对A进行读取和更新。加锁期间其它事务不能对A加任何锁

一个事务对数据对象a加了S锁，可以对A进行读取操作，但是不能更新操作。加锁期间其它事务能对A加s锁，但是不能加X锁。

1. 意向锁

使用意向锁（intention locks）可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务T想要对表A加X锁，就需要先检测是否有其它事务对表A或者表A中的任意一行加了锁，那么就需要对表A的每一行都检测一次，这是非常耗时的。

意向锁在原来的X/S锁之上引入了IX/IS都是表锁，用来表示一个事务想要在表中的某个数据上加X锁或S锁。有以下规定

一个事务在获得某个数据行对象的S锁之前，必须先获得表的IS锁或者更强的锁

一个事务在获得某个数据行对象的X锁之前，必须先获得表的IX锁

通过引入意向锁，事务T想要对表A加X锁，只需要先检测是否有其它事务对表A加了X/IX/S/IS锁，如果加了就表示有其它事务正在使用这个表或者表中的某一行的锁，因此事务T加X锁失败。



解释如下：

任意IS/IX锁之间都是兼容的，因为它们只表示想要对表加锁，而不是真正加锁

这里兼容关系针对的是表级锁，而表级锁IX锁和行级的X锁兼容，两个事务可以对两个数据加X锁（事务T1想要对数据行R1加X锁，事务T2想要对同一个表的数据行R2加X锁，两个事务都需要对该表加IX锁，但是IX锁是兼容的，并且IX锁与行级的X锁也是兼容的，因此两个事务都能加锁成功，对同一个表中的数据行做修改。）

1. 读写锁的作用、为什么要用

多个线程同时读一个资源类没有任何问题,所以为了满足并发量,读取共享资源应该可以同时进行

但是,如果有一个线程想去写共享资源来,就不应该在有其他线程可以对该资源进行读或写;

读-读 可以共存,读-写 不能共存, 写-写 可以工作;

写操作:原子+独占,整个过程必须是一个完整的,中间不许被分割,不许被打断;

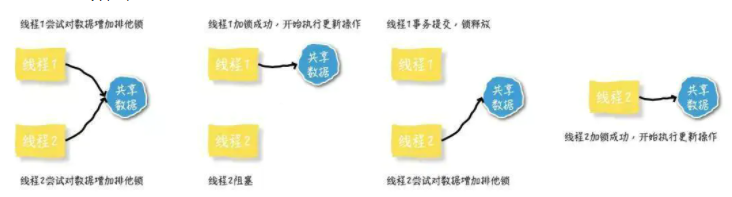
1. 乐观锁和悲观锁（使用场景）

当程序中可能出现[并发](https://www.jianshu.com/p/e8d674931af4" \t "_blank)的情况时，就需要保证在并发情况下数据的准确性，以此确保当前用户和其他用户一起操作时，所得到的结果和他单独操作时的结果是一样的。这种手段就叫做并发控制。并发控制的目的是保证一个用户的工作不会对另一个用户的工作产生不合理的影响。

实现并发控制的主要手段大致可以分为乐观并发控制和悲观并发控制两种。

首先要明确：无论是悲观锁还是乐观锁，都是人们定义出来的概念，可以认为是一种思想。其实不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像 hibernate、[tair](https://www.jianshu.com/p/72fc26ec5216" \t "_blank)、memcache 等都有类似的概念。所以，不应该拿乐观锁、悲观锁和其他的数据库锁等进行对比。**乐观锁比较适用于读多写少的情况(多读场景)，悲观锁比较适用于写多读少的情况(多写场景)。**

悲观锁（Pessimistic Lock）



理解：

当要对数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制，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。

要使用悲观锁，必须关闭 [MySQL](https://www.jianshu.com/p/359c3fe50c7f) 数据库的自动提交属性。因为 MySQL 默认使用 autocommit 模式，也就是说，当执行一个更新操作后，MySQL 会立刻将结果进行提交。(sql语句：set autocommit=0)

**之所以叫做悲观锁，是因为这是一种对数据的修改持有悲观态度的并发控制方式。总是假设最坏的情况，每次读取数据的时候都默认其他线程会更改数据，因此需要进行加锁操作，当其他线程想要访问数据时，都需要阻塞挂起。**

1. 传统的关系型数据库使用这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。
2. Java 里面的同步 [synchronized](https://www.jianshu.com/p/c8f997e7f75c) 关键字的实现。

悲观锁主要分为共享锁与排他锁：

 共享锁【shared locks】又称为读锁，简称S锁。顾名思义，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

 排他锁【exclusive locks】又称为写锁，简称X锁。顾名思义，排他锁就是不能与其他锁并存，如果一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务是可以对数据行读取和修改。

说明：

悲观并发控制实际上是“先取锁再访问”的保守策略，为数据处理的安全提供了保证。但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会。另外还会降低并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数据。

乐观锁（Optimistic Locking）：



理解：

乐观锁是相对悲观锁而言的，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，则返回给用户错误的信息，让用户决定如何去做。乐观锁适用于读操作多的场景，这样可以提高程序的吞吐量。

乐观锁机制采取了更加宽松的加锁机制。乐观锁是相对悲观锁而言，也是为了避免数据库幻读、业务处理时间过长等原因引起数据处理错误的一种机制，但乐观锁不会刻意使用数据库本身的锁机制，而是依据数据本身来保证数据的正确性。乐观锁的实现：

1. CAS 实现：Java 中java.util.concurrent.atomic包下面的原子变量使用了乐观锁的一种 CAS 实现方式。
2. 版本号控制：一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数。当数据被修改时，version 值会+1。当线程A要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值与当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。
3. 使用条件限制实现乐观锁。比如：在售、暂停出售、产品库存。

乐观锁，大多是基于**数据版本(Version)记录机制**实现。何谓数据版本？即为**数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。**

说明：

乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。

如何选择：

在乐观锁与悲观锁的选择上面，主要看下两者区别以及使用场景就可以了。

1. 响应效率：如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁。乐观锁并未真正加锁，直到提交的时候才会去锁定，效率高。一旦锁的粒度掌握不好，更新失败的概率就会比较高，容易发生业务失败。
2. 冲突频率：如果冲突频率非常高，建议使用悲观锁，保证成功率。冲突频率大，选择乐观锁会需要多次重试才能成功，代价比较大。
3. 重试代价：如果重试代价大，建议采用悲观锁。悲观锁依赖数据库锁，效率低。更新失败的概率比较低。
4. 乐观锁如果有人在你之前更新了，你的更新应当是被拒绝的，可以让用户重新操作。悲观锁则会等一个更新完成。这也是区别

随着互联网[三高架构(](https://www.jianshu.com/p/58589a7dc5c6" \t "_blank)**[高并发、高性能、高可用](https://www.jianshu.com/p/58589a7dc5c6" \t "_blank)**[)](https://www.jianshu.com/p/58589a7dc5c6" \t "_blank)的提出，悲观锁已经越来越少的被应用到生产环境中了，尤其是并发量比较大的业务场景。

乐观锁实现方式：

* 1. 版本号控制机制
  2. CAS（compare and swap）算法

1. MySQL的行级表和oracle的行级表底层实现
2. IEG光子工作室
3. 数据库如何做到持久化
4. redolog和undolog记录有什么区别

s

1. 最大连接数的设定有什么影响
2. 主从库的同步方法
3. Mvcc的实现原理（多版本并发控制，只说了个大概）

什么是MVCC？

**MVCC**

**MVCC**，全称Multi-Version Concurrency Control，即**多版本并发控制**。MVCC是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问，在编程语言中实现事务内存。

**MVCC**在**MySQL InnoDB**中的实现主要是为了提高数据库并发性能，用更好的方式去处理读-写冲突，做到即使有读写冲突时，也能做到不加锁，非阻塞并发读

什么是MySQL innodb下的当前读和快照读

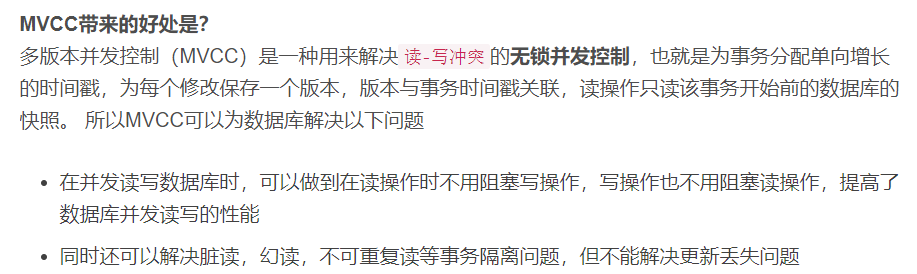
当前读

像select lock in share mode(共享锁), select for update ; update, insert ,delete(排他锁)这些操作都是一种当前读，为什么叫当前读？就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁

快照读

像不加锁的select操作就是快照读，即不加锁的非阻塞读；快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读；**之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于多版本并发控制，即MVCC,可以认为MVCC是行锁的一个变种，但它在很多情况下，避免了加锁操作，降低了开销；既然是基于多版本，即快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本**

**说白了MVCC就是为了实现读-写冲突不加锁，而这个读指的就是快照读, 而非当前读，当前读实际上是一种加锁的操作，是悲观锁的实现**



总之，MVCC就是因为大牛们，不满意只让数据库采用悲观锁这样性能不佳的形式去解决读-写冲突问题，而提出的解决方案，所以在数据库中，因为有了MVCC，所以我们可以形成两个组合：

MVCC + 悲观锁

MVCC解决读写冲突，悲观锁解决写写冲突

MVCC + 乐观锁

MVCC解决读写冲突，乐观锁解决写写冲突

这种组合的方式就可以最大程度的提高数据库并发性能，并解决读写冲突，和写写冲突导致的问题

1. Mvcc和乐观锁的区别

多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 这样在读操作不用阻塞写操作，写操作不用阻塞读操作的同时，避免了脏读和不可重复读

乐观并发控制（OCC）是一种用来解决写-写冲突的无锁并发控制，认为事务间争用没有那么多，所以先进行修改，在提交事务前，检查一下事务开始后，有没有新提交改变，如果没有就提交，如果有就放弃并重试。乐观并发控制类似自选锁。乐观并发控制适用于低数据争用，写冲突比较少的环境。

多版本并发控制可以结合基于锁的并发控制来解决写-写冲突，即MVCC+2PL，也可以结合乐观并发控制来解决写-写冲突。

1. Next-Key Locks（解决MVCC的幻读问题）

InnoDB默认next-key Locks

在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。MVCC 不能解决幻影读问题，Next-Key Locks 就是为了解决这个问题而存在的。在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

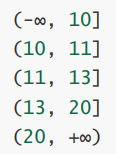
**InnoDB有三种行锁的算法：**

1. Record Locks : 单个行记录上的锁。 锁定一个记录上的索引，而不是记录本身。 如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚簇索引，因此 Record Locks 依然可以使用。
2. Gap Locks : 间隙锁，锁定一个范围，但不包括记录本身。GAP锁的目的，是为了防止 同一事务的两次当前读，出现幻读的情况。 锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就 不能在 t.c 中插入 15。



1. Next-Key Locks : 1+2，锁定一个范围，并且锁定记录本身。对于行的查询，都是采用该方法，主要目的是解决幻读的问题。

它是 Record Locks 和 Gap Locks 的结合，**不仅锁定一个记录上的索引，也锁定索引之间 的间隙**。它锁定一个前开后闭区间，例如一个索引包含以下值：10, 11, 13, and 20，那么 就需要锁定以下区间：



1. Redis和MySQL的区别
2. 类型上

从类型上来说，MySQL是关系型数据库，redis是缓存数据库

1. 作用上

MySQL用于持久化的存储数据到硬盘，功能强大，但是速度较慢

Redis用于存储使用较为频繁的数据到缓存中，读取数据快

1. 需求上

MySQL和redis因为需求的不同，一般都是配合使用

1.mysql和redis的数据库类型

mysql是关系型数据库，主要用于存放持久化数据，将数据存储在硬盘中，读取速度较慢。

redis是NOSQL，即非关系型数据库，也是缓存数据库，即将数据存储在缓存中，缓存的读取速度快，能够大大的提高运行效率，但是保存时间有限

2.mysql的运行机制

mysql作为持久化存储的关系型数据库，相对薄弱的地方在于每次请求访问数据库时，都存在着I/O操作，如果反复频繁的访问数据库。第一：会在反复链接数据库上花费大量时间，从而导致运行效率过慢；第二：反复的访问数据库也会导致数据库的负载过高，那么此时缓存的概念就衍生了出来。

3.缓存

缓存就是数据交换的缓冲区（cache），当浏览器执行请求时，首先会对在缓存中进行查找，如果存在，就获取；否则就访问数据库。

缓存的好处就是读取速度快

4.redis数据库

redis数据库就是一款缓存数据库，用于存储使用频繁的数据，这样减少访问数据库的次数，提高运行效率。

redis适合放一些频繁使用，比较热的数据，因为是放在内存中，读写速度都非常快，一般会应用在下面一些场景

排行榜、计数器、消息队列推送、好友关注、粉丝

首先要知道mysql存储在**磁盘**里，redis存储在**内存**里，redis既可以用来做持久存储，也可以做缓存，而目前大多数公司的存储都是mysql + redis，mysql作为主存储，redis作为辅助存储被用作缓存，加快访问读取的速度，提高性能

1. 如果是一个群消息，你要这么处理？因为每个人读取的数据的进度是不一样的（如何用redis实现多人消息队列）
2. Redis持久化

为什么需要持久化？

Redis对数据的操作都是基于内存的，当遇到了进程退出、服务器宕机等意外情况，如果没有持久化机制，那么Redis中的数据将会丢失无法恢复。有了持久化机制，Redis在下次重启时可以利用之前持久化的文件进行数据恢复。理解和掌握Redis的持久机制，对于Redis的日常开发和运维都有很大帮助，也是在大厂面试经常被问到的知识点。Redis支持的两种持久化机制：

RDB：把当前数据生成快照保存在硬盘上。

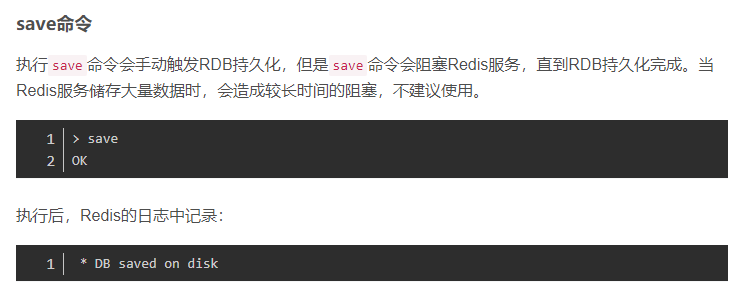
AOF：记录每次对数据的操作到硬盘上。

RDB持久化：

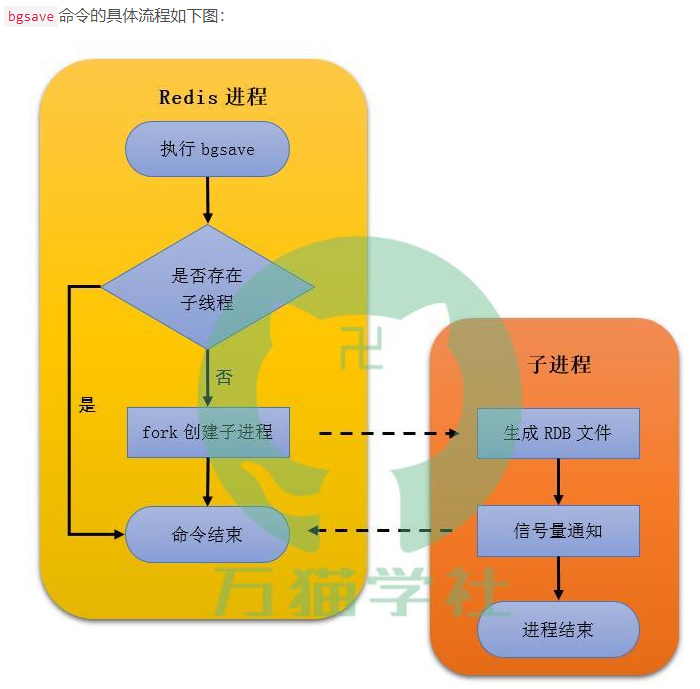
RDB（Redis database）持久化是把**当前redis中全部数据生成快照保存在硬盘上**。RDB持久化可以手动触发，也可以自动触发。

手动触发：

Save和bgsave命令都可以手动触发RDB持久化。







1. 执行bgsave命令，Redis进程先判断当前是否存在正在执行的RDB或AOF子进程，如果存在就是直接结束。

2. Redis进程执行fork操作创建子进程，在fork操作的过程中Redis进程会被阻塞。

3. Redis进程fork完成后，bgsave命令就结束了，自此Redis进程不会被阻塞，可以响应其他命令。

4. 子进程根据Redis进程的内存生成快照文件，并替换原有的RDB文件。

5. 子进程通过信号量通知Redis进程已完成。

自动触发：

除了执行以上命令手动触发以外，Redis内部可以自动触发RDB持久化。自动触发的RDB持久化都是采用bgsave的方式，减少Redis进程的阻塞。那么，在什么场景下会自动触发呢？

1. 在配置文件中设置了save的相关配置，如sava m n，它表示在m秒内数据被修改过n次时，自动触发bgsave操作。
2. 当从节点做全量复制时，主节点会自动执行bgsave操作，并且把生成的RDB文件发送给从节点。
3. 执行debug reload命令时，也会自动触发bgsave操作。
4. 执行shutdown命令时，如果没有开启AOF持久化也会自动触发bgsave操作。

RDB优点：

RDB文件是一个紧凑的二进制压缩文件，是redis在某个时间点的全部数据快照。所以使用RDB恢复数据速度远远比AOF快，非常适合备份、全量复制、灾难恢复等场景。

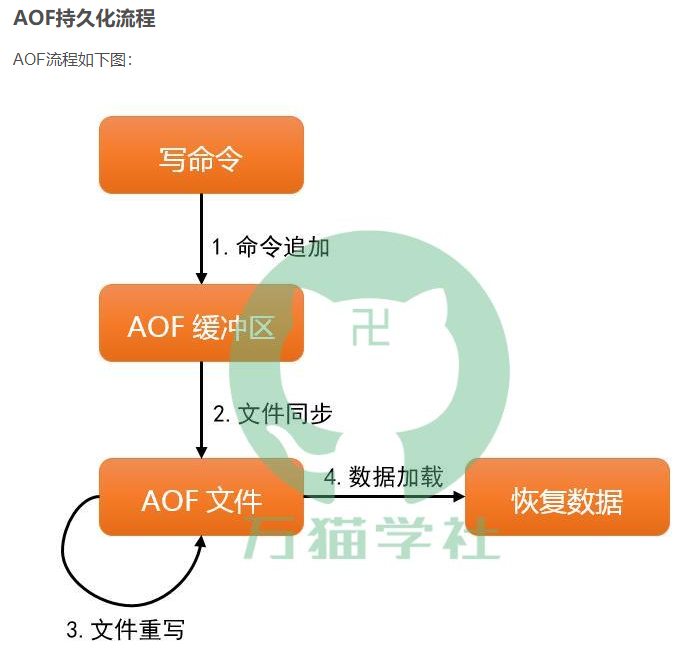
RDB缺点：

每次进行bgsave操作都要执行fork操作创建子经常，属于重量级操作，频繁执行成本过高，所以无法做到实时持久化，或者秒级持久化。

另外，由于Redis版本的不断迭代，存在不同格式的RDB版本，有可能出现低版本的RDB格式无法兼容高版本RDB文件的问题。

AOF持久化

APF（Append Only File）持久化是每次写命令追加写入日志中，当需要恢复数据时重新执行AOF文件中的命令就可以了。AOF解决了数据持久化的实时性，也是目前主流的Redis持久化方式。

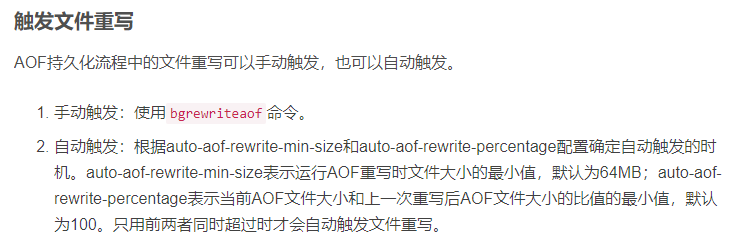


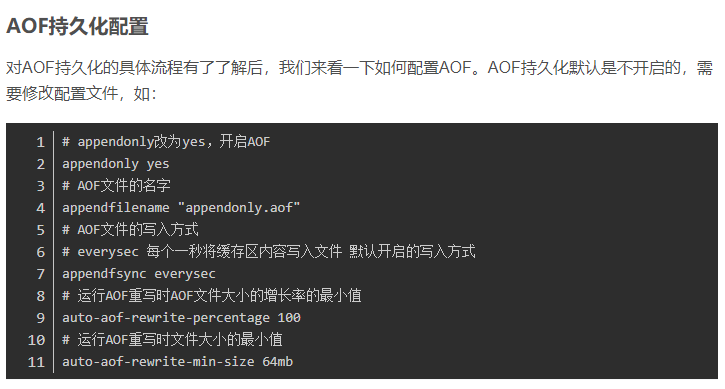
1. 命令追加（append）：所有写命令都会被追加到AOF缓冲区（aof\_buf）中。
2. 文件同步（sync）：根据不同策略将AOF缓存区同步到AOF文件中
3. 文件重写（rewrite）：定期对AOF文件进行重写，以达到压缩的目地。
4. 数据加载（load）：当需要恢复数据时，重新执行AOF文件中的命令。

AOF中，文件同步策略：

AOF持久化流程中的文件同步有以下几个策略：

1. **always**：每次写入缓存区都要同步到AOF文件中，硬盘的操作比较慢，限制了Redis高并发，不建议配置。
2. **no**：每次写入缓存区后不进行同步，同步到AOF文件的操作由操作系统负责，每次同步AOF文件的周期不可控，而且增大了每次同步的硬盘的数据量。
3. **eversec**：每次写入缓存区后，由专门的线程每秒钟同步一次，做到了兼顾性能和数据安全。是建议的同步策略，也是默认的策略。





1. 假设我们有一个场景，比如说唯一游戏里面我们会有几千万甚至上亿的玩家上一个角色的数据，我们会有一个角色表。表里面角色数量太多情况下对于严重的影响数据的查找以及修改，如果让你来设计，你怎么对这个角色表做一个设计，尽可能的去做，不会因为角色数量的增加而影响了我们一个读写的速率。
2. Redis是一个单进程架构为什么快？

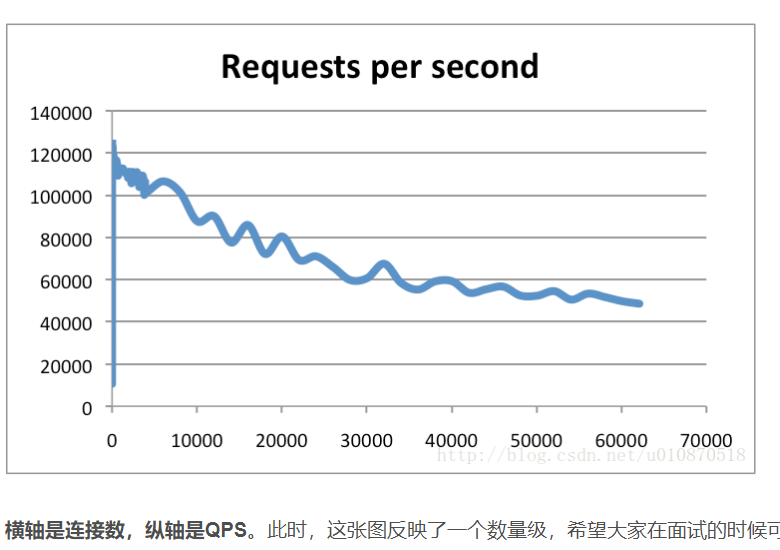
什么是Redis？

它可以用作：**数据库、缓存和消息中间件**。

Redis 将数据储存在内存里面，读写数据的时候都不会受到硬盘 I/O 速度的限制，所以速度极快。

Redis采用的是基于内存的采用的是**单进程单线程**模型的 **KV 数据库。**

**Redis到底有多快？**



QPS（每秒内查询次数）

为什么redis那么快

1. 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；
2. 数据结构接单，对数据操作也简单，redis中的数据结构是专门设计的
3. 采用单线程，避免了不必要上下文切换和竞争条件，也不存在多进程或者多线程切换而消耗CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁导致性能消耗
4. 使用多路I/O复用模型，非阻塞IO（epoll）
5. 使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，redis直接构建了VM机制，因为一般系统调用系统函数，会浪费一定的时间去移动和请求。

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量。

为什么Redis是单线程的？

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

1. 如果大量的key同一时间过期怎么办？

Redis 基于 Reactor 模式开发了自己的网络事件处理器-文件事件处理器

Redis的过期策略

Redis采用定期删除和惰性删除两种策略

1. 定期删除：指的是redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。 注意，**这里可不是每隔100ms就遍历所有的设置过期时间的key，那样就是一场性能上的**灾难。实际上redis是每隔100ms随机抽取一些key来检查和删除的，定期删除可能会导致很多key到了时间却没有被删除的情况
2. 惰性删除：就是在你操作key时候，redis会检查一下这个key有没有设置过期时间，如果设置了过期时间并且过期了，那么redis此时就会删除这个key，不会给你返回任何数据
3. 跳跃表时间复杂度？使用场景
4. 为什么redis用跳跃表不用红黑树（https://www.cnblogs.com/cjjjj/p/12751487.html）

Redis只在两个地方用到了跳跃表，一个是实现有序集合键（zset），另一个是在集群节点中用作内部数据结构，除此之外，跳表在Redis里面没有其他用途。

1. skiplist和各种平衡树（如AVL、红黑树等）的元素是有序排列的，而哈希表不是有序的。因此，在哈希表上只能做单个key的查找，不适宜做范围查找。所谓范围查找，指的是查找那些大小在指定的两个值之间的所有节点。
2. 在做范围查找的时候，平衡树比skiplist操作要复杂。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在skiplist上进行范围查找就非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。
3. 平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。
4. 从内存占用上来说，skiplist比平衡树更灵活一些。一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势。
5. 查找单个key，skiplist和平衡树的时间复杂度都为O(log n)，大体相当；而哈希表在保持较低的哈希值冲突概率的前提下，查找时间复杂度接近O(1)，性能更高一些。所以我们平常使用的各种Map或dictionary结构，大都是基于哈希表实现的。
6. 从算法实现难度上来比较，skiplist比平衡树要简单得多。
7. 平衡二叉树、二叉搜索树、B树、B+树和红黑树的区别
8. 互斥锁和自旋锁的区别和应用场景
9. 为什么采用B+树这么一个结构

12. Redis的数据结构
13. **String：这个是最基础类型，支持最常规的set/get操作，value可以是String也可以是数字，一般用于做些计数功能的缓存**
14. **hash：这里value存放是结构化的对象，可以很方便的操作其中的字段，比如存放用户登陆的信息，设置缓存过期时间模拟session效果**
15. **list：可以用于时间简单的消息队列功能，同时可以使用lrange命令实现分页，性能非常好**
16. **set：一般用于实现去重功能，比如实现交集、并集、差集，比如可以用set实现共同好友功能**
17. **sorted set：有序集合，其中多了一个score参数用于排序，可以实现排行榜的功能**
19. MySql 分库分表

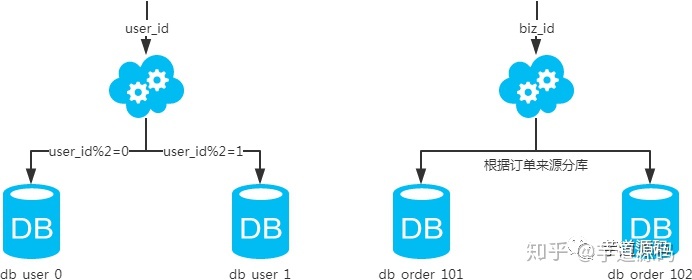
## 数据库瓶颈

不管是IO瓶颈，还是CPU瓶颈，最终都会导致数据库的活跃连接数增加，进而逼近甚至达到数据库可承载活跃连接数的阈值。在业务Service来看就是，可用数据库连接少甚至无连接可用。接下来就可以想象了吧（并发量、吞吐量、崩溃）。

IO瓶颈第一种：磁盘读IO瓶颈，热点数据太多，数据库缓存放不下，每次查询时会产生大量的IO，降低查询速度 -> 分库和垂直分表。第二种：网络IO瓶颈，请求的数据太多，网络带宽不够 -> 分库。

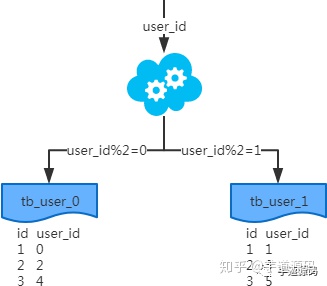
CPU瓶颈第一种：SQL问题，如SQL中包含join，group by，order by，非索引字段条件查询等，增加CPU运算的操作 -> SQL优化，建立合适的索引，在业务Service层进行业务计算。第二种：单表数据量太大，查询时扫描的行太多，SQL效率低，CPU率先出现瓶颈 -> 水平分表。

## 水平分库



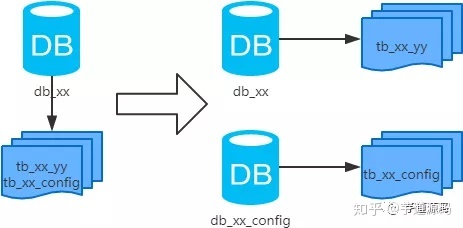
概念：以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。结果：每个库的结构都一样；每个库的数据都不一样，没有交集；所有库的并集是全量数据；场景：系统绝对并发量上来了，分表难以根本上解决问题，并且还没有明显的业务归属来垂直分库。分析：库多了，io和cpu的压力自然可以成倍缓解。

## 水平分表



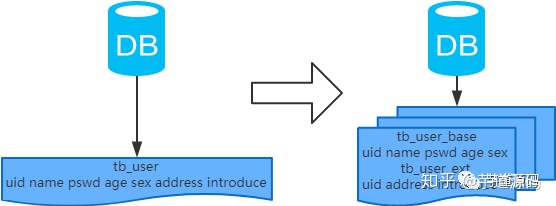
概念：以字段为依据，按照一定策略（hash、range等），将一个表中的数据拆分到多个表中。结果：每个表的结构都一样；每个表的数据都不一样，没有交集；所有表的并集是全量数据；场景：系统绝对并发量并没有上来，只是单表的数据量太多，影响了SQL效率，加重了CPU负担，以至于成为瓶颈。分析：表的数据量少了，单次SQL执行效率高，自然减轻了CPU的负担。

## 垂直分库



概念：以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。结果：每个库的结构都不一样；每个库的数据也不一样，没有交集；所有库的并集是全量数据；场景：系统绝对并发量上来了，并且可以抽象出单独的业务模块。分析：到这一步，基本上就可以服务化了。例如，随着业务的发展一些公用的配置表、字典表等越来越多，这时可以将这些表拆到单独的库中，甚至可以服务化。再有，随着业务的发展孵化出了一套业务模式，这时可以将相关的表拆到单独的库中，甚至可以服务化。

## 垂直分表



概念：以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。结果：每个表的结构都不一样；每个表的数据也不一样，一般来说，每个表的字段至少有一列交集，一般是主键，用于关联数据；所有表的并集是全量数据；场景：系统绝对并发量并没有上来，表的记录并不多，但是字段多，并且热点数据和非热点数据在一起，单行数据所需的存储空间较大。以至于数据库缓存的数据行减少，查询时会去读磁盘数据产生大量的随机读IO，产生IO瓶颈。分析：可以用列表页和详情页来帮助理解。垂直分表的拆分原则是将热点数据（可能会冗余经常一起查询的数据）放在一起作为主表，非热点数据放在一起作为扩展表。这样更多的热点数据就能被缓存下来，进而减少了随机读IO。拆了之后，要想获得全部数据就需要关联两个表来取数据。但记住，千万别用join，因为join不仅会增加CPU负担并且会讲两个表耦合在一起（必须在一个数据库实例上）。关联数据，应该在业务Service层做文章，分别获取主表和扩展表数据然后用关联字段关联得到全部数据。

1. Mysql实战
2. 在高并发情况下，如何做到安全的修改同一行数据？

悲观锁和乐观锁的思想重温

1. UUID是如何保证唯一性的（分布式ID的实现）

首先，即便是虚拟机的话MAC地址也是不一样的。另外你说的统一时间还是个宏观的概念，这个仅 仅是决定了UUID生产串中的某一部分相同而已，因为为了保证UUID的唯一性，规范定义了包括网 卡MAC地址、时间戳、名字空间（Namespace）、随机或伪随机数、时序等元素。 当然，你要说UUID是不是绝对的不会出现重复的，这个也不能这样说的（我下面会提到）

UUID具有以下涵义：

经由一定的算法机器生成

为了保证UUID的唯一性，规范定义了包括网卡MAC地址、时间戳、名字空间（Namespace）、随机或伪随机数、时序等元素，以及从这些元素生成UUID的算法。UUID的复杂特性在保证了其唯一 性的同时，意味着只能由计算机生成。

非人工指定

非人工识别 UUID是不能人工指定的，除非你冒着UUID重复的风险。UUID的复杂性决定了“一般人“不能直 接从一个UUID知道哪个对象和它关联。

在特定的范围内重复的可能性极小

UUID的生成规范定义的算法主要目的就是要保证其唯一性。但这个唯一性是有限的，只在特定的范围内才能得到保证，这和UUID的类型有关（参见UUID的版本）。

UUID的版本 UUID具有多个版本，每个版本的算法不同，应用范围也不同。

UUID Version 1：基于时间的UUID 基于时间的UUID通过计算当前时间戳、随机数和机器MAC地址得到。

由于在算法中使用了MAC地 址，这个版本的UUID可以保证在全球范围的唯一性。但与此同时，使用MAC地址会带来安全性问 题，这就是这个版本UUID受到批评的地方。如果应用只是在局域网中使用，也可以使用退化的算 法，以IP地址来代替MAC地址－－Java的UUID往往是这样实现的（当然也考虑了获取MAC的难 度）。

UUID Version 2：DCE安全的UUID DCE（Distributed Computing Environment）

安全的UUID和基于时间的UUID算法相同，但会把 时间戳的前4位置换为POSIX的UID或GID。这个版本的UUID在实际中较少用到。

UUID Version 3：基于名字的UUID（MD5） 基于名字的UUID通过计算名字和名字空间的MD5散列值得到。这个版本的UUID保证了：相同名字空间中不 同名字生成的UUID的唯一性；不同名字空间中的UUID的唯一性；相同名字空间中相同名字的UUID重复生成 是相同的。

空间中不同名字生成的UUID的唯一性；不同名字空间中的UUID的唯一性；相同名字空间中相同名 字的UUID重复生成是相同的。

UUID Version 4：随机UUID 根据随机数，或者伪随机数生成UUID。

这种UUID产生重复的概率是可以计算出来的，但随机的东 西就像是买彩票：你指望它发财是不可能的，但狗屎运通常会在不经意中到来。

UUID Version 5：基于名字的UUID（SHA1） 和版本3的UUID算法类似，只是散列值计算使用SHA1（Secure Hash Algorithm 1）算法。

从UUID的不同版本可以看出， Version 1/2适合应用于分布式计算环境下，具有高度的唯一性； Version 3/5适合于一定范围内名字唯一，且需要或可能会重复生成UUID的环境下； 至于Version 4，个人的建议是最好不用（虽然它是最简单最方便的）。 通常我们建议使用UUID来标识对象或持久化数据，但以下情况最好不使用UUID： 映射类型的对象。比如只有代码及名称的代码表。 人工维护的非系统生成对象。比如系统中的部分基础数据。 对于具有名称不可重复的自然特性的对象，最好使用Version 3/5的UUID。比如系统中的用户。如果用户的 UUID是Version 1的，如果你不小心删除了再重建用户，你会发现人还是那个人，用户已经不是那个用户了。 （虽然标记为删除状态也是一种解决方案，但会带来实现上的复杂性。）

1. MySql数据库cpu飙升的话，要怎么处理呢

排查过程：

（1）使用top 命令观察，确定是mysqld导致还是其他原因。（2）如果是mysqld导致的，show processlist，查看session情况，确定是不是有消耗资源的sql在运行。（3）找出消耗高的 sql，看看执行计划是否准确， 索引是否缺失，数据量是否太大。

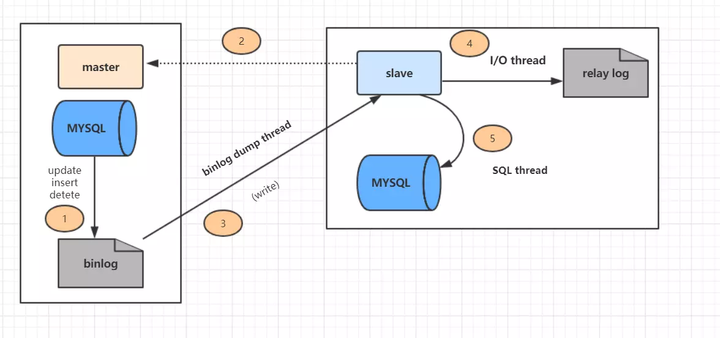
处理：

（1）kill 掉这些线程(同时观察 cpu 使用率是否下降)， （2）进行相应的调整(比如说加索引、改 sql、改内存参数) （3）重新跑这些 SQL。

其他情况：

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等

1. MySql的主从延迟，你怎么解决



* 步骤一：主库的更新事件(update、insert、delete)被写到binlog
* 步骤二：从库发起连接，连接到主库。
* 步骤三：此时主库创建一个binlog dump thread，把binlog的内容发送到从库。
* 步骤四：从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log
* 步骤五：还会创建一个SQL线程，从relay log里面读取内容，从Exec\_Master\_Log\_Pos位置开始执行读取到的更新事件，将更新内容写入到slave的db

**主从同步延迟的原因**

一个服务器开放Ｎ个链接给客户端来连接的，这样有会有大并发的更新操作, 但是从服务器的里面读取binlog的线程仅有一个，当某个SQL在从服务器上执行的时间稍长 或者由于某个SQL要进行锁表就会导致，主服务器的SQL大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

**主从同步延迟的解决办法（待完善，不是什么好方法）**

* 主服务器要负责更新操作，对安全性的要求比从服务器要高，所以有些设置参数可以修改，比如sync\_binlog=1，innodb\_flush\_log\_at\_trx\_commit = 1 之类的设置等。
* 选择更好的硬件设备作为slave。
* 把一台从服务器当度作为备份使用， 而不提供查询， 那边他的负载下来了， 执行relay log 里面的SQL效率自然就高了。
* 增加从服务器喽，这个目的还是分散读的压力，从而降低服务器负载。

1. 如果某个表有近千万数据，CRUD比较慢，如何优化。

#### 分库分表

* 某个表有近千万数据，可以考虑优化表结构，分表（水平分表，垂直分表），当然，你这样回答，需要准备好面试官问你的分库分表相关问题呀，如
* 分表方案（水平分表，垂直分表，切分规则hash等）
* 分库分表中间件（Mycat，sharding-jdbc等）
* 分库分表一些问题（事务问题？跨节点Join的问题）
* 解决方案（分布式事务等）

分库分表，首先得知道**瓶颈在哪里**，然后**才能合理地拆分**（**分库还是分表？水平还是垂直？分几个？**）。**且不可为了分库分表而拆分**。**选key很重要，既要考虑到拆分均匀，也要考虑到非partition key的查询**。只要能满足需求，**拆分规则越简单越好**。

#### 索引优化

除了分库分表，优化表结构，当然还有所以索引优化等方案~

1. 如果让你做分库与分表的设计，简单说说你会怎么做？
3. 说一下 MySQL 执行一条查询语句的内部执行过程