## 集合

Collection

ArrayList、LinkedList、Vector（线程安全）

ArrayList：底层是数组，初始化时，数据量是0，当add时候，默认变成10，扩容机制是上一次容量的1.5倍，特性查询比较快（通过脚标去查），删除效率低

LinkedList：底层是头结点和尾节点的双向链表，提供了2个方法，头插法LinkedFirst，尾插法LinkedLast，特性非常适用于经常增加和删除的场景，查询效率慢(他会对各个节点进行比较得到查询数据)

Vector：线程安全，底层和arrayList一样是数组，和ArrayList的区别在于，大部分方法都是被synchronized关键字所修饰，扩容机制是上一次容量的2倍

### set

HashSet

### map

HashMap、ConcurrentHashMap线程安全、HashTable线程安全

hashmap：

1.7版本--底层是数组+单链表

1.8版本--数组+单链表+红黑树

1、单链表和红黑色之间的转换，单链表长度>=8，并且哈希桶数组长度>=64，会将单链表转换成红黑树形式进行存储，红黑树节点数量<=6，会重新转换成单链表

2、hash桶数组默认是16个，阈值默认是0.75，当插入16\*0.75=12个元素的时候(简单理解为，已经有12个数组被占用)，就会触发扩容

3、扩容机制是上一次容量的2倍，然后重新进行新的hash运算(hash值 & （length -1）),填充到新的hash桶数组

4、链表扩容的数据迁移是将相同下标的节点组合在一起，放到新数组里面；红黑色是直接转移根节点

### ConcurrentHashMap

线程安全，实现分段锁表

1.7版本--底层分片数组，为了保障线程安全，用到了Segment锁，他继承于ReentrantLock

1.8版本--数组+单链表+红黑树，使用synchronized+CAS操作来保证线程安全，里面有2把锁，第一把锁在数据插入时，防止插入时数据并发，第二把锁在链表树化，防止红黑色里面根节点旋转导致根节点转移。

### HashTable

线程安全，全局锁表

## CAS

1、compare and swap的缩写，中文翻译成比较并替换

2、CAS(内存值V，旧的预期值A，要修改的新值B)

3、相当于轻量级的加锁过程，在并发量不是特别大的情况下，当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做

CAS存在问题：

1、CPU开销大；自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销

2、只能保证一个共享变量原子操作；对多个共享变量操作时，循环CAS就无法保证操作的原子性

3、ABA-标志位 时间戳；如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

## synchronized

synchronized关键字在编译之后会在同步代码块前后加入2个指令：

1、monitorenter(锁)

2、monitorexit(解锁)

### 偏向锁加锁：

1、将线程id写入到锁对象的对象头中

2、其他线程进入并获取锁之后，锁对象会结束偏向状态，膨胀成轻量锁或重量锁，有资源竞争重量锁，无资源竞争轻量锁

### 轻量级锁加锁：

1、执行monitorenter(锁)

2、新线程进来会在虚拟栈中为当前线程开辟一个空间叫Lock Record

3、将锁对象的Mark Word写入里面

4、CAS尝试将锁对象的Mark Word轻量级锁指针指向当前Lock Record（作用是让其他线程知道，该object monitor已被占用），成功获取锁，失败则挂起线程

5、然后lock record里的owner指针指向object mark word（作用是为了在接下里的运行过程中，识别哪个对象被锁住了）

6、执行monitorexit(解锁)

### 重量级锁同步代码块加锁：synchronized(o)

1、执行monitorenter(锁)

2、偏向锁膨胀成重量锁的过程，偏向锁会先进行撤销偏向成无锁01，此时锁对象的监视器锁monitor为0

3、新的线程获取锁，锁对象的monitor监视器锁会+1（锁支持重入），当前线程成为monitor的owner（所有者），并将monitor的指针写入到锁对象头

4、修改锁对象标志位为10，此时锁就膨胀成重量锁

5、若其他线程已经占有monitor的所有权，那么当前尝试获取monitor的所有权的线程会被阻塞，直到monitor的进入数变为0，才能重新尝试获取monitor的所有权。

6、执行monitorexit(解锁)

7、同步代码块执行完，加了几次锁释放几次，将monitor清零，完成释放锁

### 重量级锁同步方法加锁：public synchronized void test()

JVM使用ACC\_SYNCHRONIZED标识来实现，即JVM通过在方法访问标识符(flags)中加入ACC\_SYNCHRONIZED来实现同步功能。

## volatile关键字

CPU和内存之间的线程效率是差好多数量级的，但为了保证他们之间的计算，不影响CPU的计算，中间会有好多LLV缓存，我们线程在这个缓存中去工作，首先他取数据会在主内存取到工作内存中去计算，计算完之后在传给主内存，此时就出现了多线程间变量的可见性，java为了解决这种问题，他就抽象出了JMM这种模型，被volatile修饰的变量，保证变量能在多线程之间的可见性，也就是变量被更新了，能立刻刷新到主内存中，其他线程能立刻取到最新的变量数据，这是总线嗅探机制，但不保证原子性，简单的说就是保证线程修改了变量能及时被其他线程看到，但不保证多个线程改变同一个变量之后数据的完整。

如何保障数据原子性：使用synchronized关键字、ReentrantLock加锁、AtomicXXX类（多个AtomicXXX类连续调用不能构成原子性）来保证数据原子性

### 总线嗅探机制

每个处理器通过嗅探在总线上传播的数据来检查自己的缓存值是不是过期了，如果处理器发现自己缓存行对应的内存地址呗修改，就会将当前处理器的缓存行设置无效状态

### 总线风暴：

由于volatile的mesi缓存一致性协议需要不断的从主内存嗅探和cas不断循环无效交互导致总线带宽达到峰值 解决办法：部分volatile和cas使用synchronize

mesi缓存一致性协议：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

资料：https://zhuanlan.zhihu.com/p/137193948

## JMM内存模型

JMM内存模型Java Memory Model是为了屏蔽各种硬件和操作系统对内存访问的差异性，简单理解就是为了保证程序在不同的操作系统和硬件，执行的结果是一样的。

Java的一切可见性问题都是因为编译器优化、内存优化、指令优化导致，硬件的底层已经保证了可见性，可以理解Java不存在可见性问题（伪可见性）。

https://hllvm-group.iteye.com/group/topic/34932#post-232535

### JMM的happends-before原则

1、如果操作A和操作B属于同一个线程，那么说明操作A的操作对操作B可见

2、unlock hb lock 线程A的unlock操作，在线程B的lock操作前操作

3、如果操作A是对volatile变量的写操作，操作B是对同一个变量的读操作，那么操作Ahappends-before操作B

4、happends-before具有传递性，操作A hb 操作B，操作B hb 操作C，操作A hb 操作C

## SQL调优

### Sql优化

1、首先考虑where及order条件建立索引

2、少使用is null处理，用=0处理

3、少使用!= <> 判断

4、少使用or做条件，用union all做数据合并

5、少使用in/not in，能使用betwen就使用

6、少使用%在前的模糊查询

7、少使用对字段进行表达式操作num/2=100 --> num=100\*2

8、少使用函数

9、不要写一些没意义的查询，比如1=1

10、用exists代替in

11、索引不是越多越好，虽然查询快，但是插入和修改会降低

12、尽量使用数字型字段，因为字符需要对每个字段进行比较

13、尽量用varchar代替char

14、少使用\*，用具体字段和删除临时表

16、大数据备份用select into 新表名 from 旧表名 代替 create table

17、数据超过1万条数据，考虑改写脚本

### 索引选择

1. 没主键，需要创建聚族索引
2. 有主键，查询还是慢，创建非聚族索引
3. 非聚族索引规则：
4. 尽量使用where、order by、group by后面的字段来创建
5. 查询最频繁的一列放在左边
6. 查询重复值最少的放在左边
7. 组合索引的字段尽量不要超过3个

Sql调优最基本的是表要有主键，因为有主键的表MySQL会创建聚族索引，聚族索引的好处是它的主键和数据行是在一行的，在explain查询语句的时候会发现，它的type级别是const，这是很高的一种级别。当有主键的时候，如果一条SQL语句很慢，可以去查看他是否创建了相应的索引，建立索引要尽量选择where、order by、group by后面的字段或者join in连接的字段作为索引列，这些索引列也要排序，要符合最左匹配原则，你选的时候要根据他们的索引选择器，你的非重复的数据行和重复的数据行 中间的排列，大的放左，小的放右这样的形式，这种多列的索引列，要去建立联合索引而非单个索引。

如果数据查询的非常频繁，可以考虑给它使用覆盖索引，覆盖索引是可以直接在索引中查询到数据，相对来说还是很快的。

聚族索引（数据需要确保唯一）：它确切来说应该是一种数据结构，它是将它的主键和它的数据行放在一块的，找到了索引就找到了数据。

非聚族索引（数据不需要确保唯一）：就是我们说的普通索引，他的索引存放的是叶子节点，叶子节点包含他的主键、索引列、事务ID等信息，它在查询的时候，其实是通过它的索引列查询它的主键，在通过主键去回表查询，还是通过聚族索引查的，这样的速度会相对慢点

### 事务

脏读：一个事务读取了另一个事务改写但未提交的数据，如果这些数据被回滚，则读到的数据时无效的。

不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。在事务1中，第一次读到了A记录数据，中间事务2修改了A记录数据，第二次读到的A记录数据和第一次读到的A记录数据不一致。

幻读：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发生有些原来没有的记录。事务1修改所有A表数据，事务2增加了A表数据，事务1的用户就会发现A表还有未修改的数据。

InnoDB的特性就是好它支持事务，支持事务的时候，为了保证事务的并发度，它提供了一个隔离级别，有4种隔离级别：

1、Read Uncommitted：结果什么都不能保证

2、Read Committed：解决事务之间的脏读，但是不能保证不可重复读和幻读

3、Repeatable Read：默认事务，保证不发生脏读、不可重复读，不保证幻读

4、Serializable：事务最高级别，串行化，一个个事务串行排列。事务一个接一个执行，等待前一个事务执行完，后面的事务才可以顺序执行，保证不发生脏读、不可重复读、幻读。

## 线程池的运行机制

线程池他是有一个核心线程数，当线程运行的时候，如果没有设置成预启动加载，首发线程数为0，当提交一个新任务的时候，他会首先建立一个核心线程去执行任务，如果此时一直来任务，之前线程没有执行完，那么就会一直创建核心线程，当达到最大核心线程数时，如果都还在忙，那么此时就会放到BlockingQueue里面作为节点，如果BlockingQueue队列也放满了，而却核心线程都还在忙，那就会去建立新线程，它叫做非核心线程，若一直创建，数量达到非核心线程数max access，此时就会触发一个拒绝策略。

线程构造，在创建线程池的时候，需要提供线程的threadFactory，一定要指定他名称，并且可以设置为守护线程，其他线程执行完，主线程也执行完，JVM才会停止运行。

### JDK内置了四种拒绝策略

1. AbortPolicy，直接抛出异常，这是默认策略
2. DiscardPolicy，悄无声息的丢弃任务
3. DiscardOldestPolicy，丢弃最早的未执行任务
4. CallerRunsPolicy，用调用者所在的线程来执行任务，这种方式会影响新任务提交速度

### 阻塞队列

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的无界阻塞队列，也可以理解为有界队列，因为他的默认容量是Integer.MAX\_VALUE。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个支持延时获取元素的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列