1. **Mysql**
2. **基本查询语句**

**开启慢查询日志查询参数**：

**long\_query\_time、 slow\_query\_log、slow\_query\_log\_file**

**File SORT**

**单路算法：一次磁盘IO,读数据到sort\_buffer，进行排序**

**多路算法：多次磁盘IO**

**相关参数：**

**sort\_buffer\_size：缓冲区大小**

**max\_length\_for\_sort\_data：当查询字段大小之和小于它，排序字段不是text|BLOB就会使用单路排序**

**查询语句：**

**SHOW VARIABLES LIKE "long\_query\_time";**

**SHOW VARIABLES Like "slow\_query\_log";**

**SHOW VARIABLES LIKE "slow\_query\_log\_file";**

**SHOW VARIABLES LIKE "%slow\_query%";**

**SET GLOBAL long\_query\_time=1;**

1. **Mysql的索引**

**定义：索引是一个排好序数据结构，目的是提高sql查询速度，查找+排序**

**优点：查找+排序，降低了磁盘IO消耗查询，降低了cpu消耗**

**缺点：索引占存储空间、查找快，但是插入和修改慢，需要经常变更**

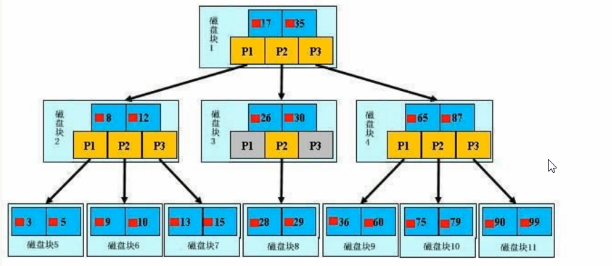
**基本语句**

**CREATE [unique] index indexname ON mytable(col\_name)**

**ALTER mytable ADD [UNIQUE] INDEX indexname ON (colname)**

**DROP INDEX indexname ON mytable**

**SHOW INDEX FROM table\_name\G**

**B+树是索引的基本结构？**

**默认主键索引，聚集索引，包含索引和数据，若无主键则mysql自带row\_id**

**Sql慢的原因：sql写的不合理；索引失效；关联join太多表**

**那些需要索引：**

**1、主键索引 2、频繁查询的字段 3、join的字段外键关系索引**

**4、频繁更新的不适合建立索引 5.where中经常用到的建索引**

**6、单值索引、组合索7、排序字段order by 8、统计分组字段group by**

**那些不要建索引：**

1. **记录少 2.经常增删改 3.某个列包含很多重复，枚举字段**

**EXPLAIN 查询计划 + sql**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **id** | **select\_type** | **table** | **type** | **possible\_keys** | **key** | **ref** | **rows** | **extra** |

**Id 语句加载顺序Id相同，顺序加载上到下；id不同，id大先执行**

**select\_type 查询的类型，普通、联合查询、嵌套查询、子查询**

**SIMPLE：简单的select查询，不包括子查询和union**

**PRIMARY :查询中包含任何复杂的子部分。最外层的标记为**

**SUBQUERY：select中或where中包含了子查询**

**DERIVED：from列表子查询标记为DERIVED衍生表执行子查询临时表**

**UNION：若第二个SELECT出现在UNION之后则被标记为UNION若UNION包含在from子查询中，外层SELECT被标记为DERIVED**

**UNION RESULT: 从UNION表获取结果的SELECT**

**type**

**system >const>eq\_ref>ref>range>index()>ALL(全表扫描)**

**system：表只有一条记录，const的特例**

**const：通过一次索引就找到了，用于主键索引，或者唯一索引，只匹配一行数据**

**eq\_ref：唯一性索引扫描，对于每个索引都只有一个记录匹配**

**ref: 非唯一索引扫描、返回匹配某个单值的所有行**

**rang：检索指定范围的行，使用索引选择行，范围between ，>,<**

**index： 全索引扫描**

**ALL：全表扫描**

**possible\_keys 现实可能用到的索引**

**key 实际查询的过程使用索引，null没有使用使用索引**

**理论可能用到索引，实际没有用， 索引失效**

**理论用不到，实际上用到了 索引覆盖了**

**key\_len 索引中使用字节的数**

**ref 索引那一列被使用了，并使用了什么值**

**rows 检索需要查询多少行**

**extra 额外信息**

**Using filesort mysql无法按照索引排序，使用外部排序（最左前缀）**

**Using temp mysql创建临时表，排序order by,group by（最左前缀）**

**Using index select数据列使用覆盖索引避免访问数据行直接从索引取**

**同时using where 使用索引查找**

**没有using where 没有使用索引**

**Using where、Using joinbuffer、Impossi where、Distinct 第一个匹配**

**避免索引失效**

**全值索引**

**最佳左前缀原则**

**不要在索引列上计算函数，类型转换**

**范围右边的索引失效**

**尽量使用覆盖索引**

**!= 和<>索引失效**

**Is null 和 is not null**

**Like中左边的%不走索引，非要用%XXXX%覆盖索引**

1. **Mysql优化 （建表、建索引、查询语句的优化）**

**A：建表**

**1. 原则：尽量使用整型表示字符串**

**存储IP**

**INET\_ATON(str)，address to number**

**INET\_NTOA(number)，number to address**

**2. 原则：定长和非定长数据类型的选择**

**decimal不会损失精度，存储空间会随数据的增大而增大。double占用固定空间，较大数的存储会损失精度。非定长的还有varchar、text**

**定长char，非定长varchar、text（上限65535，其中varchar还会消耗1-3字节记录长度，而text使用额外空间记录长度）**

**4. 原则：尽可能选择小的数据类型和指定短的长度**

**5. 原则：尽可能使用 not null**

**非null字段的处理要比null字段的处理高效些！且不需要判断是否为null。**

**6.原则：单表字段不宜过多**

**7. 存储引擎选择**

**InnoDB 支持事物acid，支持外键，支持行锁，适用于并发场景**

**MYISM 表锁，对增加和查询较快 ，并发性差**

**InnoDB只有在通过索引条件检索数据时使用行级锁，否则使用表锁，行锁变表锁**

**B：建立索引**

**C：查询语句优化尽量使用索引**

1. **Mysql中的锁**

**锁的类型： 读锁（共享锁）、写锁（排他锁）、间隙锁gap**

**show open tables**

**lock table mytbl1 read, mytbl2 write**

**unlock tables**

**读锁（锁表）： session1自己可以读、不能写、不能读其他表，**

**session2可以读其他，可以读该表，但是写会阻塞**

**写锁（锁表）： session1，可以读写，不能读写其他表**

**session2 不可以读写该枷锁的表，会被阻塞**

**show status like ‘table%’**

**Table\_locks\_immediate 产生表级锁定的次数**

**Table\_locks\_waited 表级锁定等待的次数**

**锁的粒度**：**行锁、表锁**

**事务的ACID，**

**事务的问题， 丢失更新、脏读、不可重复读、幻读**

**事务的隔离级别 读未提交、 读已提交、可重复读、可序列化**

**Show variables like “tx\_isolation” 查看事务隔离级别**

**Set autocommit=0； 默认提交关闭**

**事务内**

1. **读自己的修改，读己之所写，别的事务是读不到你的修改**
2. **事务修改一行，未提交前，其他事务修改同一行会阻塞**
3. **事务修改一行，未提交前，其他事务修改另一行无阻塞**

**无索引行锁失效为表锁**

**索引失效性能下降行锁变表锁**

**间隙锁的危害：范围锁定然后修改，即使没有某个索引也会被锁定，其他的session你不能插入**

**锁定某一行，for update**

**Show status like “innodb\_row\_lock%“**

1. **Mysql存储引擎**

|  |  |  |
| --- | --- | --- |
| **对比** | **MyISAM** | **InnoDB** |
| **主外键** | **no** | **yes** |
| **事务** | **no** | **yes** |
| **行表锁** | **表锁** | **行锁（高并发）** |
| **缓存** | **缓存索引，不缓存数据** | **索引和数据内存要求高、内存会决定性影响性能** |
| **表空间** | **小** | **大** |
| **关注点** | **性能** | **事务** |
| **适合** | **经常插入、查询场景** | **增删改查** |
| **文件格式** | **数据和索引是分别存储的，数据.MYD，索引.MYI** | **数据和索引是集中存储的，.ibd** |

1. **Redis**
2. **Redis是单线程的为什么速度快**

**redis是基于内存的，内存的读写速度非常快；**

**redis是单线程的，省去了很多上下文切换线程的时间;**

**redis多路复用技术，可以处理并发的连接。非阻塞IO 内部实现采用epoll，采用了epoll+自己实现的简单的事件框架。epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性，绝不在io上浪费一点时间。**

**String的最大大小512m，Set的最大个数2的23次方-1**

1. **Redis的持久化RDB / AOF**

**RDB：持久化方式能够在指定的时间间隔能对你的数据进行快照存储.**

**AOF：持久化方式记录每次对服务器写的操作, AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大.**

**你也可以通过调用 SAVE或者 BGSAVE ， 手动让 Redis 进行保存，**

**save 60 1000**

**redis-check-aof –fix**

1. **Redis的实现原理epoll**
2. **Redis的集群主从，哨兵模式、**
3. **过期删除策略，内存淘汰策略**

**过期删除策略：设置了过期时间的key如何进行删除**

**惰性过期：只有当访问这个过期的key的时候才会把这个key删除**

**定期过期：**

**具体就是Redis每秒10次做的事情：**

**1测试随机的20个keys进行相关过期检测。**

**2删除所有已经过期的keys。**

**3如果有多于25%的keys过期，重复步奏1.**

**Redis key的过期时间和永久有效分别怎么设置？EXPIRE和PERSIST命令。**

**内存淘汰策略：内存不够的时候淘汰相关key**

**全局的键空间选择性移除**

**noeviction：新写入操作会报错。**

**allkeys-lru：在键空间中，移除最近最少使用的key。**

**allkeys-random：在键空间中，随机移除某个key。**

**设置过期时间的键空间选择性移除**

**volatile-lru：在设置了过期时间的键空间中，移除最近最少使用的key。**

**volatile-random：在设置了过期时间的键空间中，随机移除某个key。**

**volatile-ttl：在设置了过期时间的键空间中，有更早过期时间的key优先移除**

1. **Redis和 Memcached**

|  |  |  |
| --- | --- | --- |
| **参数** | **Redis** | **Memcached** |
| **数据类型** | **5种数据类型，并且和相应的本地方法** | **String** |
| **类型** | **内存数据库，nosql** | **内存数据库，nosql** |
| **网络IO** | **1.单线程NIO多路复用epoll** | **1.多线程NIO** |
| **持久化** | **RDB、AOF** | **不支持** |

1. **Redis如何做大量数据插入？**

**Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。单个命令从client到server，因为一个个的插入会有大量的时间浪费在每一个命令往返时间上**

**cat data.txt | redis-cli –pipe**

**(cat data.txt; sleep 10) | nc localhost 6379 > /dev/null**

**这个命令不会对data..txt中的命令进行校验**

1. **Redis的缓存问题，缓存穿透、缓存击穿、缓存雪崩？**

**缓存穿透**：**是指缓存和数据库中都没有的数据，请求都落到数据库上**

**1.从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点。**

**2.采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力**

**接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截**

**缓存击穿**：**一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力**

**1.设置热点数据永远不过期。2.加互斥锁，互斥锁**

**缓存雪崩：指缓存同一时间大面积的失效，后面的请求都会落到数据库上**

**1.缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。**

**2.一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。**

**3.给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。**

1. **Redis的事务**

**事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。**

**事务开始 MULTI**

**命令入队**

**事务执行 EXEC**

通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务，从事务状态中退出

**WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令**

**事务的原子性：命令放入队列中发生错误则全部不执行，命令放入队列后在运行时错误，错误的不执行，正确的可以执行**

1. **Redis的的集群主从复制、哨兵模式、集群？**
2. **主从复制、读写分离（降低m读的压力，s数据的复制和同步m来处理）**

**1.开启多个redis的实例**

**2.使用slaveof ip port / replicaof <masterip> <masterport>**

**info replication查看主从复制信息**

**当从库和主库建立MS关系后，会向主数据库发送SYNC命令**

**主库接收到SYNC命令后 (RDB持久化过程)，并将期间接收到的写命令缓存起来**

**当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从Redis**

**从Redis接收到后，会载入快照文件并且执行收到的缓存的命令**

**主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致**

**哨兵模式+主从结构**

**三、JVM与高并发多线程**

1. **多线程与锁的机制**

**线程创建的几种方法**

**线程池使用及其参数**

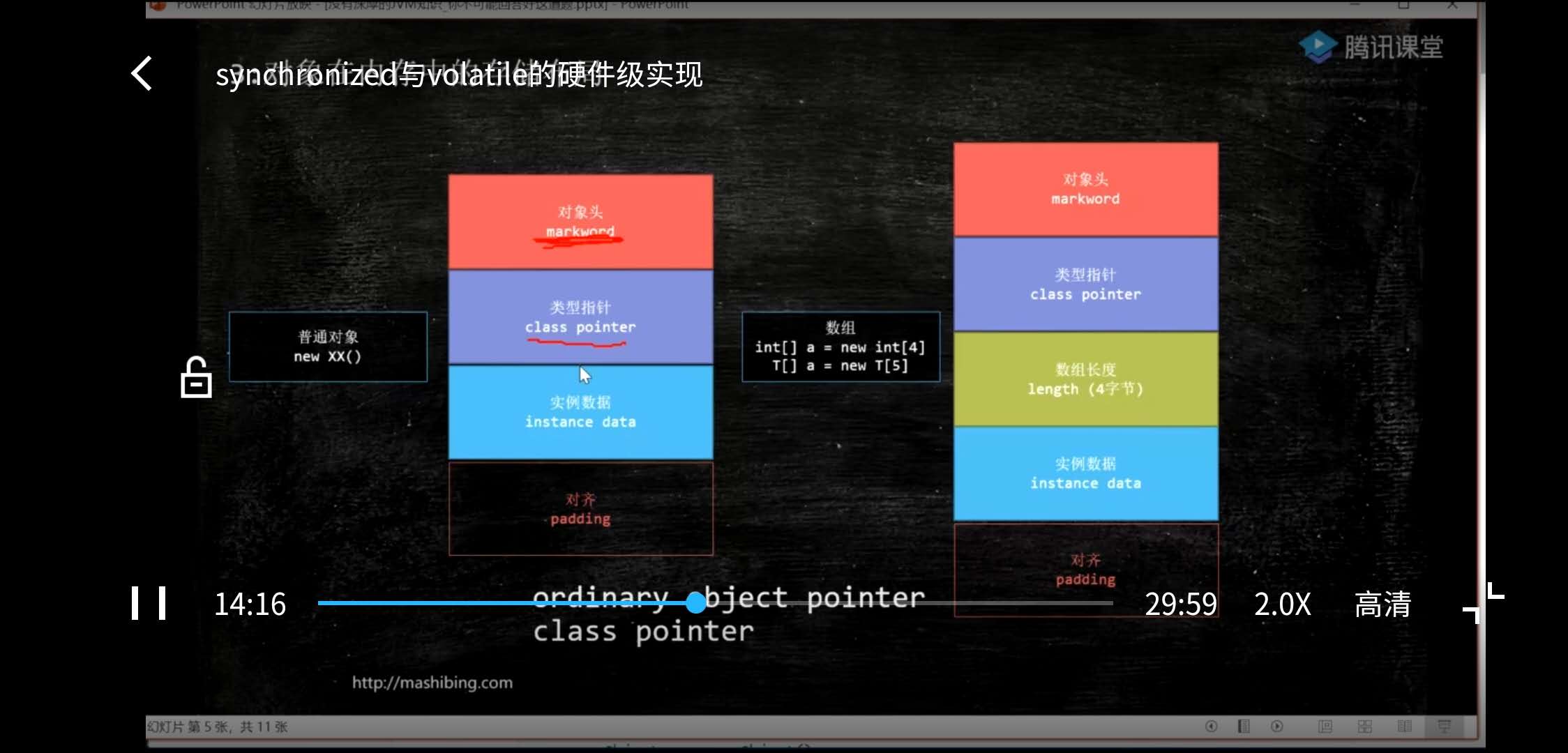
**锁的概念，Synchonized** 和**Lock**

**synchronized同步块使用了monitorenter和monitorexit指令实现同步，对一个对象的监视器(monitor)进行获取，这个过程是排他的，也就是说同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。**

**线程执行到monitorenter指令时，会尝试获取对象所对应的monitor所有权，也就是尝试获取对象的锁，而执行monitorexit，就是释放monitor的所有权。**

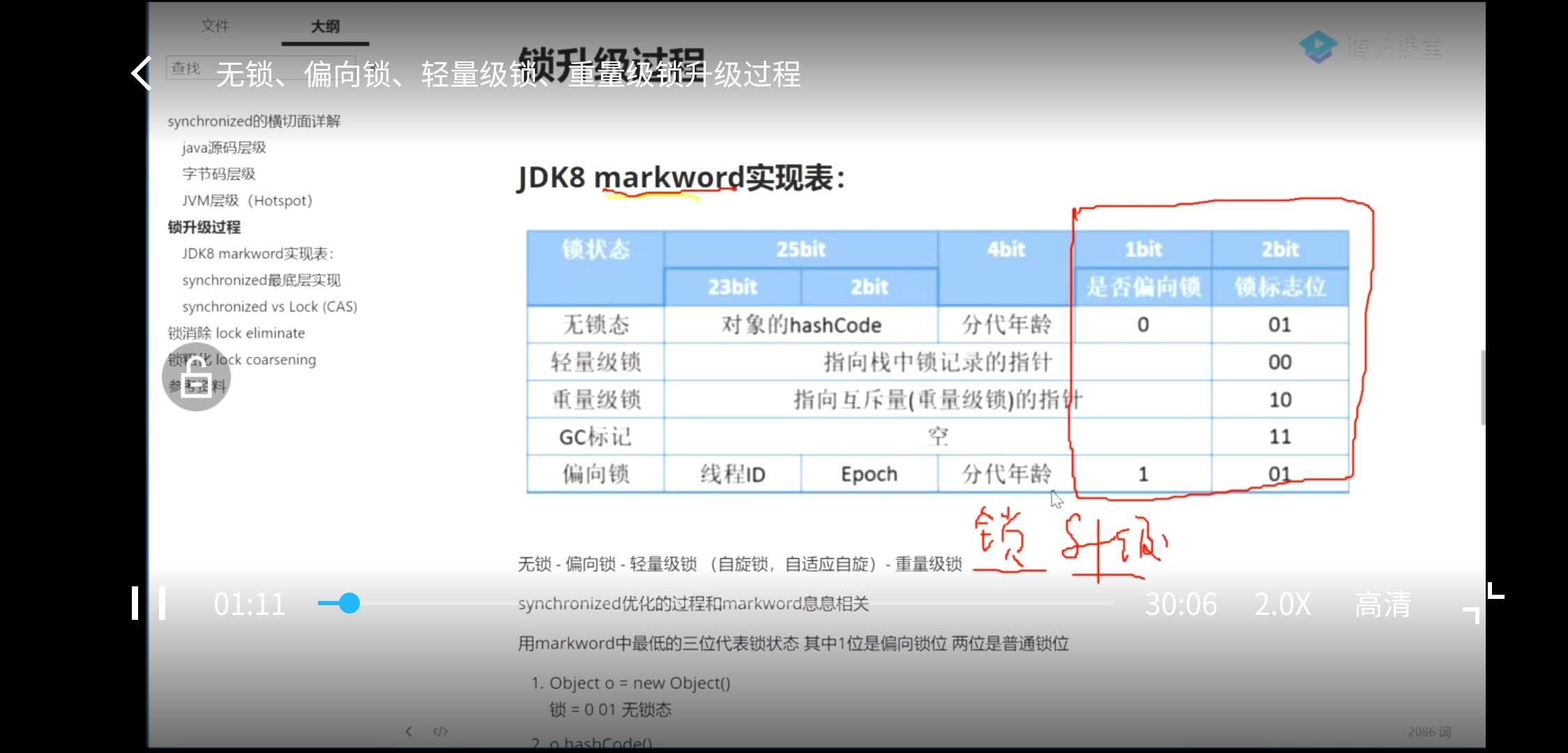
**Java对象头 8个字节 ， 任何一个obj必须被8整除**

对象三块区域**：对象头**（Mark Word （8）、Class Metadata Address（4））、**实例数据**和**对齐填充**；Java对象头是实现synchronized的锁对象的基础。synchronized使用的锁对象是存储在Java对象头里。它是轻量级锁和偏向锁的关键。



**Mark Word**

Mark Word用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的 锁、偏向线程 ID、偏向时间戳等等。Java对象头一般占有两个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit）。



**Class Metadata Address**

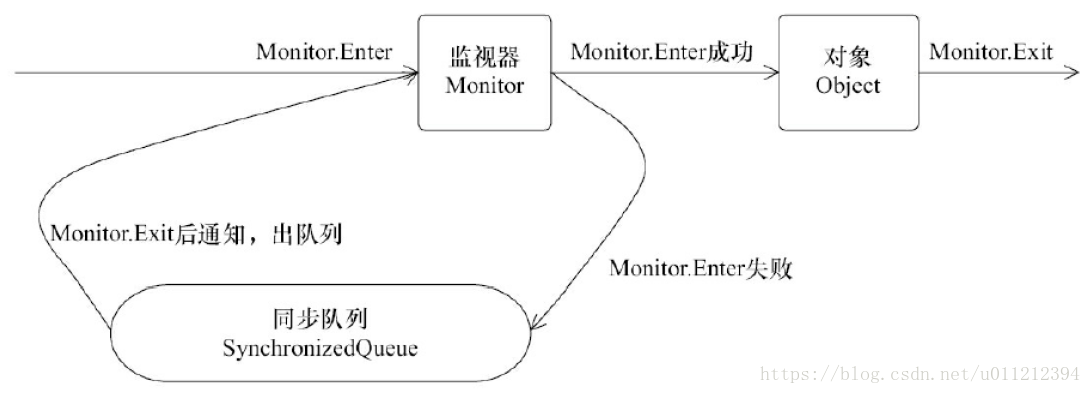
类型指针，即是对象指向它的类的元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

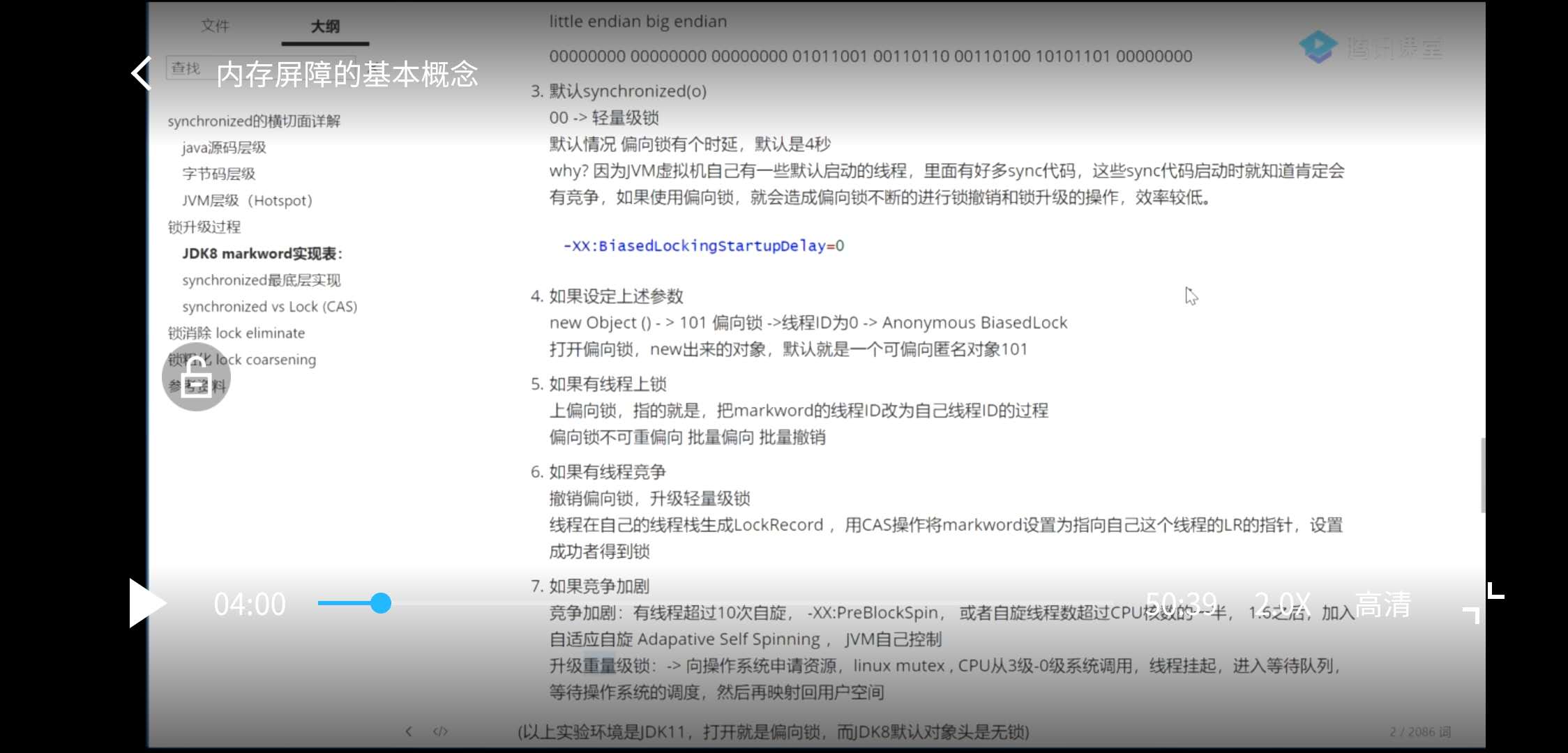
**synchronized锁的优化**

无锁->偏向锁->轻量级锁（自旋锁，无锁）->重量级锁

**Monitor**

Monitor内置于每一个Object对象中拿到即可以进行操作，没有拿到则需要阻塞等待。



  
**a.偏向锁**

HotSpot的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。

偏向锁直接把自己的线程ID写入marketword

只要有任意一个线程来抢相同对象的锁就升级

**b.轻量级锁（自旋锁、CAS）**

**线程栈中锁记录指针**

**竞争加剧，如果某个线程的10次自旋，-XX：PreBlockSpin，或者自旋线程个数超过CPU核数一半，jdk1.6加入自适应自旋，JVM自己控制**

**c.重量级锁**

**volatile**

**当对volatile变量执行写操作后，JMM会把工作内存中的最新变量值强制刷新到主内存**

**写操作会导致其他线程中的缓存无效**

**volatile是通过编译器在生成字节码时，在指令序列中添加“内存屏障”来禁止指令重排序的**

**硬件层面的“内存屏障”：**

**sfence：即写屏障(Store Barrier)，在写指令之后插入写屏障，能让写入缓存的最新数据写回到主内存，以保证写入的数据立刻对其他线程可见**

**lfence：即读屏障(Load Barrier)，在读指令前插入读屏障，可以让高速缓存中的数据失效，重新从主内存加载数据，以保证读取的是最新的数据。**

**mfence：即全能屏障(modify/mix Barrier )，兼具sfence和lfence的功能**

**lock 前缀：lock不是内存屏障，而是一种锁。执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。**

**JMM层面的“内存屏障”：**

**LoadLoad屏障： 对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。**

**StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。**

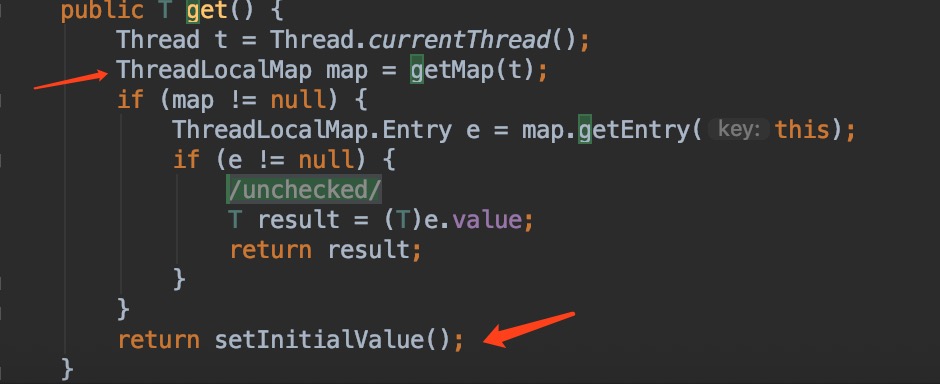
**LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。**

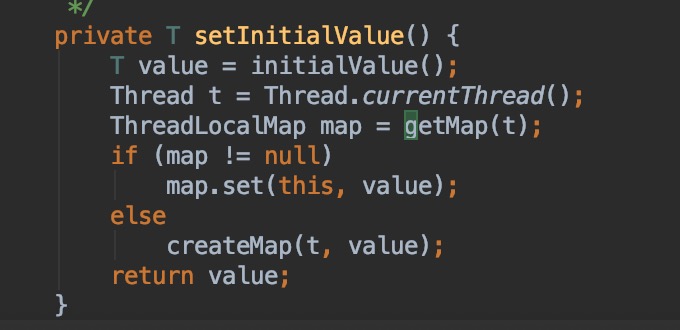
**StoreLoad屏障： 对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。**

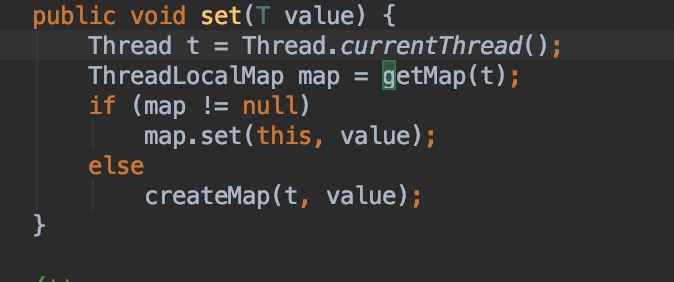
javap -v TestVolatile.class

**ThreadLocal 原理**

1. **Thead 类中有个变量。ThreadLocalMap threadlocals**
2. **ThreadLocalMap 底层是个Entry数组， Entry是个key –value对，key是this（ThreadLocal变量），value是线程需要操作的变量**







**HashMap原理**

1. 阻塞队列BlockQueue
2. CopyOnWriteArralist / ConcurrentHashMap 底层原理
3. CountDownLatch/。。。。。。

**如何选择线程个数**

1. 异常处理
2. 异常的种类
3. 异常如何处理
4. 自定义异常
5. **RPC**
6. RPC的概念及RPC的实现
7. **分布式锁 redis/redission/rdlock zookeeper**

public ResponseResult grabOrder(int orderId , int driverId){ //生成key String lock = "order\_"+(orderId+""); /\* \* 情况一，如果锁没执行到释放，比如业务逻辑执行一半，运维重启服务，或 服务器挂了，没走 finally，怎么办？ \* 加超时时间 \*/// boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(lock.intern(), driverId+"");// if(!lockStatus) {// return null;// }  
 /\* \* 情况二：加超时时间,会有加不上的情况，运维重启 \*/// boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(lock.intern(), driverId+"");// stringRedisTemplate.expire(lock.intern(), 30L, TimeUnit.SECONDS);// if(!lockStatus) {// return null;// }  
 /\* \* 情况三：超时时间应该一次加，不应该分2行代码， \* \*/ boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(lock.intern(), driverId+"", 30L, TimeUnit.SECONDS); if(!lockStatus) { return null; }  
 try { System.out.println("用户:"+driverId+" 执行抢单逻辑");  
 boolean b = orderService.grab(orderId, driverId); if(b) { System.out.println("用户:"+driverId+" 抢单成功"); }else { System.out.println("用户:"+driverId+" 抢单失败"); }  
 } finally { /\*\* \* 这种释放锁有，可能释放了别人的锁。 \*/// stringRedisTemplate.delete(lock.intern());  
 /\*\* \* 下面代码避免释放别人的锁 \*/ if((driverId+"").equals(stringRedisTemplate.opsForValue().get(lock.intern()))) { stringRedisTemplate.delete(lock.intern()); } } return null; }

1. **BIO/NIO/AIO**

**.网络编程中的BIO**

*/\*\*  
 \* BIO 同步阻塞IO的多线程写法  
 \*  
 \* 同步： A调用B，B执行返回结果给A，A继续执行  
 \* 非同步： A调用B，B直接返回一个-1或者null， A继续执行  
 \*  
 \*  
 \* 阻塞： A调用B，如果B没有返回给我，我就挂起线程，停止在这里不会继续执行  
 \* 非阻塞： A调用B，B直接就有个返回结果，我继续执行，当B返回  
 \*  
 \*  
 \* socket => fd 文件描述符 3  
 \* bind （3，8090）  
 \* listen 3  
 \* while（true）  
 \* accept （3， =5 阻塞  
 \*  
 \*  
 \* 优势：1.接受很多客户端连接  
 \* 缺点：1.线程多浪费内存  
 \* 2.cpu调度消耗时间片  
 \*  
 \* 根源：accept， recv  
 \*/*public class Server\_BIO {  
 public static void main(String[] args) throws IOException {  
  
 ServerSocket serverSocket = new ServerSocket(8090);  
 System.*out*.println("step1: new ServerSocket(8090)");  
  
 while(true){  
  
 // 该处阻塞在accept()等待链接  
 Socket client = serverSocket.accept();  
 System.*out*.println("step2: client \t"+client.getPort());  
  
 new Thread(()->{  
 try {  
 InputStream inputStream = client.getInputStream();  
 byte[] bytes = new byte[1024];  
 // BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));  
 while(inputStream.read(bytes)!=-1){  
 // read是阻塞的  
 System.*out*.println(new String(bytes));  
 }  
 } catch (IOException e) {  
 }  
 }).start();  
 }  
 }  
}

**.网络编程中的NIO**

*/\*\*  
 \*NIO 非阻塞IO 调用方法一定有返回，或者NULL或者有连接client  
 \*  
 \* 优点：1.不需要很多线程连接  
 \* 缺点：1.当有1w个客户端连接，但是只有一个输入IO，这样我要循环1w次去系统调用recv，循环内核空间切换用户空间1w次  
 \*  
 \*  
 \*/*public class Server\_NIO {  
 public static void main(String[] args) throws Exception {  
 LinkedList<SocketChannel> clients = new LinkedList<>();  
  
 ServerSocketChannel ss = ServerSocketChannel.*open*();  
 ss.bind(new InetSocketAddress(9090));  
 ss.configureBlocking(true); // OS的NIO  
 //设置非阻塞  
  
  
 while(true){  
 // 接受客户端的连接  
 Thread.*sleep*(1000);  
 SocketChannel client = ss.accept();  
 // accept 调用内核了 ： 1.没有客户端连接就返回NULL或者-1，  
 // 2.BIO会停止在这里，无返回，NIO是返回null，继续执行下去  
 // 3.有客户端连接就返回fd 5 client  
  
 if (client == null){  
 System.*out*.println("null................");  
 }else{  
 client.configureBlocking(false);  
 int port = client.socket().getPort();  
 System.*out*.println("client.....port: " + port);  
 clients.add(client);  
 }  
  
 ByteBuffer buffer = ByteBuffer.*allocateDirect*(4096);  
 // 遍历已经连接进来的客户能不能读写数据  
  
 for (SocketChannel c:clients){  
 int num = c.read(buffer);  
  
 // 这个read 发送recv的系统调用  
 if (num>0){  
 buffer.flip();  
 byte[] aaa = new byte[buffer.limit()];  
 buffer.get(aaa);  
 String b = new String(aaa);  
 System.*out*.println(c.socket().getPort()+":" + b);  
 buffer.clear();  
 }  
 }  
 }  
 }  
}

**.网络编程中的AIO**

**4. 代理模式 静态代理、动态代理、cglib代理模式**

**红黑树：**

**性质：**

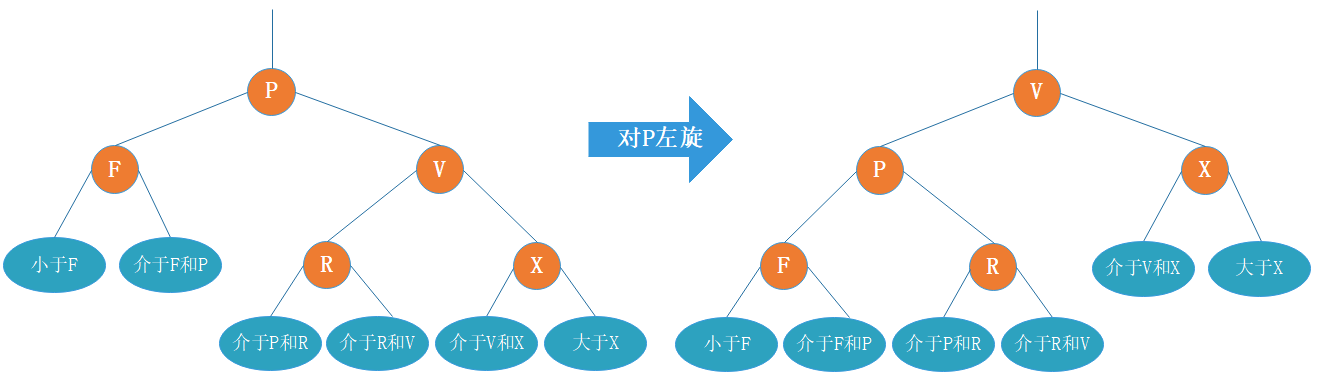
1. **所有节点不是黑色就是红色**
2. **根节点和叶子结点（Null）必须为黑色**
3. **每个红色节点的子节点必须是黑色**
4. **从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点**

性质3、4导致路径上不能有两个连续的红色节点确保了这个结果。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。[3]

**操作：**

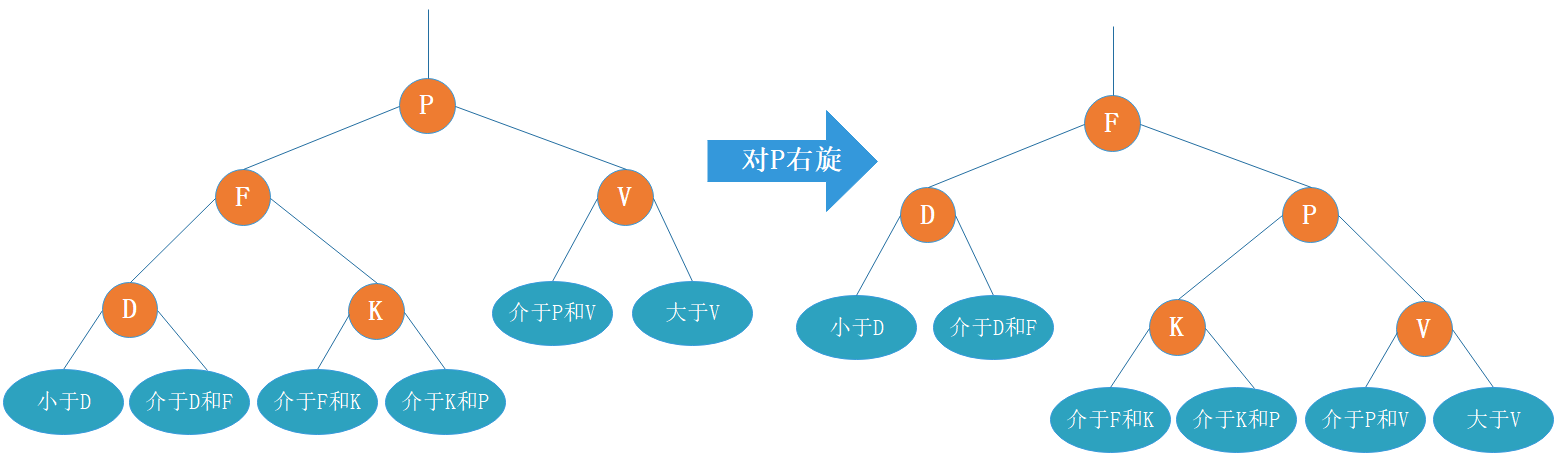
1. **变色**
2. **左旋**

以某个结点作为支点(旋转结点)，其**右子结点**变为旋转结点的父结点，右子结点的**左**子结点变为旋转结点的**右子结点**，其左子结点保持不变。



1. **右旋**

以某个结点作为支点(旋转结点)，其**左子结点**变为旋转结点的父结点，左子结点的**右**子结点变为旋转结点的**左子结点**，其右子结点保持不变。



旋转操作是**局部**的。另外可以看出旋转能保持红黑树平衡的一些端详了：当一边子树的结点少了，那么向另外一边子树“借”一些结点；当一边子树的结点多了，那么向另外一边子树“租”一些结点。

但要保持红黑树的性质，结点不能乱挪，还得靠变色了。怎么变？具体情景又不同变法，后面会具体讲到，现在只需要记住**红黑树总是通过旋转和变色达到自平衡**。