面试题(吕德路)

1.类加载机制，双亲委派模型，垃圾回收算法，jvm垃圾回收机制
，CMS，jvm调优和性能优化,minorGC,FullGC,OOM解决方案，java虚引用，弱引用，软引用，强引用 及回收策略

2.java内存模型，volatile关键字实现，happen-before,内存屏障

3.java IO,BIO,NIO,AIO 底层及实现

4.单例模式，代理模式，观察者模式，策略模式，桥接模式，工厂模式，拦截器模式

5.同步和锁机制：lock,synchronized,runnable,callback等

6.并发编程：future,futureTask,Excutor,ExcutroService,线程池，AQS,秒杀
java.lang.Thread
java.lang.Runnable
java.util.concurrent.Callable
java.util.concurrent.locks.ReentrantLock
java.util.concurrent.locks.ReentrantReadWriteLock
java.util.concurrent.atomic.Atomic*
java.util.concurrent.Semaphore
java.util.concurrent.CountDownLatch
java.util.concurrent.CyclicBarrier
java.util.concurrent.ConcurrentHashMap
java.util.concurrent.Executors

7.spring bean创建流程，控制反转，依赖注入，AOP代理模式实现，CGLIB和InvocationHandler的区别，spring 源码

8.使用模板方法设计模式和策略设计模式实现IOC,不用synchronized和lock，实现线程安全的单例模式

9. redis,rabbatmq,memecache等常用中间件，底层实现，使用场景和原理

10. 集合类 List(ArrayList和Vector),Map(HashMap）底层原理及实现，hashMap和hashTable的区别，TreeMap和hashMap，LinkHashMap区别等

1.静态变量不能被序列化
2.序列化可以被加密解密(RMI远程过程调用字节流的不安全性，所以可以模糊序列化数据)，签名认证
3.父类不序列化，子类序列化，反序列化得到的父类字段为对象为null,int为0
4.序列化id不同，无法相互序列化和反序列化
5.加Transient的字段不能被序列化
6.序列化两次写入一个文件和一次写入这个文件，反序列化后两个对象是相等的，存储规则为 持有第一次写入文件的引用，占用内存
7.对象的字段第一次写入 然后序列化，再次对该字段赋值再次序列化，反序列化取第一次序列化的值，因为虚拟机根据引用关系知道已经有一个相同对象已经写入文件，因此只保存第二次写的引用，

所以读取时，都是第一次保存的对象。读者在使用一个文件多次 writeObject 需要特别注意这个问题。
8.序列化可以将代理放在流中

sublist 不能被序列化
static 不能被序列化
transient不能被序列化

1.bean的生命周期
2.cms工作原理
3.集群单点故障
4.第N大个数
5.netty
6.http
7session
8集群反向代理
9bean生命周期
10 aqs,concurrentHashMap
11.disruptor
12 分布式，rpc原理
13 redis—>memcached
14.怎么控制控制线程并发个数
信号量和线程池
15.excutors 处理线程异常,默认设置某个参数处理异常
16.linux环境的加载顺序
17.merge和rebase区别 rebase保证提交的顺序而merge不会保证
18.线程池最小最大，队列
19 关系型数据库和非关系型数据库的区别

非关系型数据库容易水平扩展
key-value,性能比较高
关系型支持 复杂查询和事务

20.executors怎么给线程命名
ThreadFactory

21 @service,@component—区分不清的组件用component
22 rpc调用成功，但是由于网络原因，http返回失败，这个问题怎么处理
1.分布式事务
2.补偿
3.全局唯一id号，幂等性
23.jstack处理死锁
24.redis分布式锁
25 executors队列深度

26.hashmap线程不安全原因
put hash map都对链表头进行插入，会造成问题
hashmap 元素引用造成死循环
27.concurrentHashMap怎么查找和插入一个元素
28.getdeclareMethod和getMethod
getDeclaredMethod*()获取的是类自身声明的所有方法，包含public、protected和private方法。
getMethod*()获取的是类的所有共有方法，这就包括自身的所有public方法，和从基类继承的、从接口实现的所有public方法。
29.http协议基于netty的实现
30.lock底层是怎么实现的
31.聚合索引和非聚合索引的区别

聚合索引-是叶子节点也是数据节点
非聚合索引 是叶子节点但是索引节点维持一个指针指向一个数据块

32.mongoDb回收磁盘空间
mongoShell中执行db.repairDatabase()

过滤器和拦截器区别
一、filter基于filter接口中的doFilter回调函数，interceptor则基于Java本身的反射机制；

二、filter是依赖于servlet容器的，没有servlet容器就无法回调doFilter方法，而interceptor与servlet无关；

三、filter的过滤范围比interceptor大，filter除了过滤请求外通过通配符可以保护页面、图片、文件等，而interceptor只能过滤请求，只对action起作用，在action之前开始，在action完成后结束（如被拦截，不执行action）；

四、filter的过滤一般在加载的时候在init方法声明，而interceptor可以通过在xml声明是guest请求还是user请求来辨别是否过滤；

五、interceptor可以访问action上下文、值栈里的对象，而filter不能；

六、在action的生命周期中，拦截器可以被多次调用，而过滤器只能在容器初始化时被调用一次

线程池
a.一个任务提交，如果线程池大小没达到corePoolSize，则每次都启动一个worker也就是一个线程来立即执行
b.如果来不及执行，则把多余的线程放到workQueue，等待已启动的worker来循环执行
c.如果队列workQueue都放满了还没有执行，则在maximumPoolSize下面启动新的worker来循环执行workQueue
d.如果启动到maximumPoolSize还有任务进来，线程池已达到满负载，此时就执行任务拒绝RejectedExecutionHandler

Reentrantlock

最恰当的例子就是厕所，排队上厕所的人是线程，厕所是要访问的资源，但是进入厕所需要拿到钥匙才能开锁，具体到程序行为就是很多人抢钥匙上厕所，这也就是synchronized的意思。
reentrantlock分公平与非公平，公平的跟synchronized效果一样，只不过提供了一些性能上还有其他高级需求的解决方案，但是非公平的情况下就表示，你拿到钥匙之后可以随意进出厕所，只要你不把钥匙交出去你就可以一直使用厕所资源，你不让别人用，这也就是所谓的不公平了
重入锁结构如下：

syn是aqs的一个抽象类

```
abstract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = -5179523762034025860L;

    /**
     * Performs {@link Lock#lock}. The main reason for subclassing
     * is to allow fast path for nonfair version.
     */
    abstract void lock();//锁方法，由其继承类实现

    /**
     * Performs non-fair tryLock.  tryAcquire is implemented in
     * subclasses, but both need nonfair try for trylock method.
     */
    final boolean nonfairTryAcquire(int acquires) {//独占锁尝试去获得锁
        final Thread current = Thread.currentThread();
        int c = getState();//当前锁的状态
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;//重入次数
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    protected final boolean tryRelease(int releases) {
        int c = getState() - releases;//重入次数--
        if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
        boolean free = false;
        if (c == 0) {//重入次数为0则 释放锁
            free = true;
            setExclusiveOwnerThread(null);
        }
        setState(c);
```

```
            return free;
        }

        protected final boolean isHeldExclusively() {
            // While we must in general read state before owner,
            // we don't need to do so to check if current thread is owner
            return getExclusiveOwnerThread() == Thread.currentThread();
        }

        final ConditionObject newCondition() {
            return new ConditionObject();
        }

        // Methods relayed from outer class

        final Thread getOwner() {
            return getState() == 0 ? null : getExclusiveOwnerThread();
        }

        final int getHoldCount() {
            return isHeldExclusively() ? getState() : 0;
        }

        final boolean isLocked() {
            return getState() != 0;
        }

        /**
         * Reconstitutes the instance from a stream (that is, deserializes it).
         */
        private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
            s.defaultReadObject();
            setState(0); // reset to unlocked state
        }
    }
```

非公平锁

```
 static final class NonfairSync extends Sync {
        private static final long serialVersionUID = 7316153563782823691L;

        /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         */
        final void lock() {
            if (compareAndSetState(0, 1))//cas操作,尝试去获得锁,这是一种抢占式的
                setExclusiveOwnerThread(Thread.currentThread());
            else
                acquire(1);//获取失败则进入队列挂起线程
 /*
     public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
```

```
    }*/
        }

        protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);
        }
    }
```

公平锁

```
 static final class FairSync extends Sync {
        private static final long serialVersionUID = -3000897897090466540L;

        final void lock() {
            acquire(1);
        }

        /**
         * Fair version of tryAcquire.  Don't grant access unless
         * recursive call or no waiters or is first.
         */
        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
          if (c == 0) {//区别一首先要判断队列中是否有等待的元素，如果有的话进入队列，如果没有cas
操作获取锁，然后操作系统进行cas操作，如果成功的话 则设置成当前线程
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {//获得后，重入次数自增，公平锁和非公平
锁下面逻辑是相同的
                int nextc = c + acquires;//重入次数
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
    }

//判断队列中是否有其他线程
public final boolean hasQueuedPredecessors() {
        // The correctness of this depends on head being initialized
        // before tail and on head.next being accurate if the current
        // thread is first in queue.
        Node t = tail; // Read fields in reverse initialization order
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
```

```
    }

//调用java底层—> cas操作（happen-before 原则，volatile 所有线程可见）->调用硬件里面的东西

 protected final boolean compareAndSetState(int expect, int update) {
        // See below for intrinsics setup to support this
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
    }
//AQS中方法
 public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

一个问题：文件有1亿数据，怎么快速判断数据存不存在

有些逻辑问题时长出现，怎么处理

问有一个w生产问题发生对的概率为一百万分之一 但是非常严重 怎么搞

bitmap

多线程
java中有几种方法可以实现一个线程?
如何停止一个正在运行的线程?
notify()和notifyAll()有什么区别?
sleep()和 wait()有什么区别?
什么是Daemon线程？它有什么意义?

当线程只剩下守护线程的时候，JVM就会退出.但是如果还有其他的任意一个用户线程还在，JVM就不会退出.控制jvm的退出
用户线程：就是我们平时创建的普通线程.
守护线程：主要是用来服务用户线程.

java如何实现多线程之间的通讯和协作?

锁
什么是可重入锁（ReentrantLock）？
当一个线程进入某个对象的一个synchronized的实例方法后，其它线程是否可进入此对象的其它方法?
synchronized和java.util.concurrent.locks.Lock的异同?
乐观锁和悲观锁的理解及如何实现，有哪些实现方式?
并发框架
SynchronizedMap和ConcurrentHashMap有什么区别?

CopyOnWriteArrayList可以用于什么应用场景?
线程安全
什么叫线程安全？servlet是线程安全吗?
同步有几种实现方法?
volatile有什么用？能否用一句话说明下volatile的应用场景?
请说明下java的内存模型及其工作流程。
为什么代码会重排序?

并发容器和框架

如何让一段程序并发的执行，并最终汇总结果?
如何合理的配置java线程池？如CPU密集型的任务，基本线程池应该配置多大？IO密集型的任务，基本线程池应该配置多大？用有界队列好还是无界队列好？任务非常多的时候，使用什么阻塞队列能获取最好的吞吐量?
如何使用阻塞队列实现一个生产者和消费者模型？请写代码。
多读少写的场景应该使用哪个并发容器，为什么使用它？比如你做了一个搜索引擎，搜索引擎每次搜索前需要判断搜索关键词是否在黑名单里，黑名单每天更新一次。
Java中的锁

如何实现乐观锁（CAS）？
CAS操作

如何避免ABA问题?
1. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是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。
读写锁可以用于什么应用场景?
什么时候应该使用可重入锁?
什么场景下可以使用volatile替换synchronized?
并发工具

如何实现一个流控程序，用于控制请求的调用次数?

1、sleep()

使当前线程（即调用该方法的线程）暂停执行一段时间，让其他线程有机会继续执行，但它并不释放对象锁。也就是说如果有synchronized同步快，其他线程仍然不能访问共享数据。注意该方法要捕捉异常。

例如有两个线程同时执行(没有synchronized)一个线程优先级为MAX_PRIORITY，另一个为MIN_PRIORITY，如果没有Sleep()方法，只有高优先级的线程执行完毕后，低优先级的线程才能够执行；但是高优先级的线程sleep(500)后，低优先级就有机会执行了。

总之，sleep()可以使低优先级的线程得到执行的机会，当然也可以让同优先级、高优先级的线程有执行的机会。

2、join()

join()方法使调用该方法的线程在此之前执行完毕，也就是等待该方法的线程执行完毕后再往下继续执行。注意该方法也需要捕捉异常。

3、yield()

该方法与sleep()类似，只是不能由用户指定暂停多长时间，并且yield（）方法只能让同优先级的线程有执行的机会。

4、wait()和notify()、notifyAll()

这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用。synchronized关键字用于保护共享数据，阻止其他线程对共享数据的存取，但是这样程序的流程就很不灵活了，如何才能在当前线程还没退出synchronized数据块时让其他线程也有机会访问共享数据呢？此时就用这三个方法来灵活控制。

wait()方法使当前线程暂停执行并释放对象锁标示，让其他线程可以进入synchronized数据块，当前线程被放入对象等待池中。当调用notify()方法后，将从对象的等待池中移走一个任意的线程并放到锁标志等待池中，只有锁标志等待池中线程能够获取锁标志；如果锁标志等待池中没有线程，则notify()不起作用。

notifyAll()则从对象等待池中移走所有等待那个对象的线程并放到锁标志等待池中。

注意 这三个方法都是java.lang.Object的方法。

二、run和start()

把需要处理的代码放到run()方法中，start()方法启动线程将自动调用run()方法，这个由java的内存机制规定的。并且run()方法必需是public访问权限，返回值类型为void。

三、关键字synchronized

该关键字用于保护共享数据，当然前提条件是要分清哪些数据是共享数据。每个对象都有一个锁标志，当一个线程访问到该对象，被Synchronized修饰的数据将被"上锁"，阻止其他线程访问。当前线程访问完这部分数据后释放锁标志，其他线程就可以访问了。

四、wait()和notify(),notifyAll()是Object类的方法，sleep()和yield()是Thread类的方法。

(1)、常用的wait方法有wait()和wait(long timeout);

void wait() 在其他线程调用此对象的 notify() 方法或者 notifyAll()方法前，导致当前线程等待。

void wait(long timeout)在其他线程调用此对象的notify() 方法 或者 notifyAll()方法，或者超过指定的时间量前，导致当前线程等待。

wait()后，线程会释放掉它所占有的“锁标志”，从而使线程所在对象中的其他shnchronized数据可被别的线程使用。

wait()h和notify()因为会对对象的“锁标志”进行操作，所以他们必需在Synchronized函数或者synchronized block 中进行调用。如果在non-synchronized 函数或 non-synchronized block 中进行调用，虽然能编译通过，但在运行时会发生IllegalMonitorStateException的异常。。

(2)、Thread.sleep(long millis)必须带有一个时间参数。

sleep(long)使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会被执行；

sleep(long)可使优先级低的线程得到执行的机会，当然也可以让同优先级的线程有执行的机会；

sleep(long)是不会释放锁标志的。

(3)、yield()没有参数

sleep 方法使当前运行中的线程睡眠一段时间，进入不可以运行状态，这段时间的长短是由程序设定的，yield方法使当前线程让出CPU占有权，但让出的时间是不可设定的。

yield()也不会释放锁标志。

实际上，yield()方法对应了如下操作；先检测当前是否有相同优先级的线程处于同可运行状态，如有，则把CPU的占有权交给次线程，否则继续运行原来的线程，所以yield()方法称为“退让”，它把运行机会让给了同等级的其他线程。

sleep 方法允许较低优先级的线程获得运行机会，但yield（）方法执行时，当前线程仍处在可运行状态，所以不可能让出较低优先级的线程此时获取CPU占有权。在一个运行系统中，如果较高优先级

的线程没有调用sleep方法，也没有受到I/O阻塞，那么较低优先级线程只能等待所有较高优先级的线程运行结束，方可有机会运行。

yield()只是使当前线程重新回到可执行状态，所有执行yield()的线程有可能在进入到可执行状态后马上又被执行，所以yield()方法只能使同优先级的线程有执行的机会。

sharding 分片，分区 解决磁盘Io，或者写的问题

slave 增加 解决读的问题

主库多线程，从库单线程

压测

tps 连接数

不要在数据库做运算

mysql不建议处理null，索引字段必须不为null

不要在数据库存储二进制字段

尽量 避免select *

对text和大字段进行拆分

mysql不支持函数索引

使用limit高效分页

explain 执行计划

如何创建索引

索引字段越小越好，varchar(10)>100

建组合索引比较好

a join b (a 小结果集)

select * 尽量不使用的原因，因为数据库的字段会发生变化，查询的字段变多，造成效率问题，所以当定表时 可以采用select * ,当数据库表不是定表或者数据库的字段易发生变化时，则不采用select *

不采用select *的目的是
减去不必要的字段，减少结果集字节数

固定结果集比较好

mysql主从同步问题

主多线程，从单线程导致时间的延迟

延迟问题解决办法：

数据库的更新一般是单表有序进行，多线程会导致表的操作的混乱

所以多表更新，一个进程执行一个表，确保表执行的操作是有序的

主库开启binlog的作用是什么，这个需要去考虑

集中时间内多人的读和写

常见场景：1浏览器->2站点->3服务->数据

1，2，3过滤　请求尽量拦截在上游

用户端请求拦截

浏览器：查询，购票后灰度　禁止用户重复提交请求

前端x秒内只能提交一次请求

80%的拦截

同一个uid,限制访问频度，做页面缓存，x秒内到达站点的请求，均返回同一个页面

同一个item的查询，列如手机车次，做页面缓存，x秒内达到站点层的请求，均返回同一个页面

服务的拦截

写请求，请求队列，每次有限的写请求去数据层，读请求走cache 过滤到 99.9%

优化细节

1亿请求大概200个请求到db

令牌桶限流

请求- 拦截器-令牌桶1有获得令牌 没有令牌 废弃请求 ->继续发送请求

令牌桶 没1/r秒的速度向桶中存放多少个令牌 大约是两个

两个页面->1.服务器异常 2.超时，请重新刷新

1：下订单扣库存，2：付款时扣库存（体验不好）

优化细节

减少页面链接数

减少页面大小

后端数据冗余

数据镜像

异步化

将请求拦截在系统上层

读多写少的多使用缓存

限流

数组一般1M放在老年代

永久代GC的原因：
1. 永久代空间已经满了
2. 调用了System.gc()
注意： 这种GC是full GC 堆空间也会一并被GC一次

GC有环怎么处理

根搜索算法
什么是根搜索算法

垃圾回收器从被称为GC Roots的点开始遍历遍历对象，凡是可以达到的点都会标记为存活，堆中不可到达的对象都会标记成垃圾，然后被清理掉。 GC Roots有哪些

类，由系统类加载器加载的类。这些类从不会被卸载，它们可以通过静态属性的方式持有对象的引用。注意，一般情况下由自定义的类加载器加载的类不能成为GC Roots
线程，存活的线程
Java方法栈中的局部变量或者参数
JNI方法栈中的局部变量或者参数
JNI全局引用
用做同步监控的对象
被JVM持有的对象，这些对象由于特殊的目的不被GC回收。这些对象可能是系统的类加载器，一些重要的异常处理类，一些为处理异常预留的对象，以及一些正在执行类加载的自定义的类加载器。但是具体有哪些前面提到的对象依赖于具体的JVM实现。
如何处理
基于引用对象遍历的垃圾回收器可以处理循环引用，只要是涉及到的对象不能从GC Roots强引用可到达，垃圾回收器都会进行清理来释放内存。

什么时候GC
. 旧生代空间不足
旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：
java.lang.OutOfMemoryError: Java heap space
为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。
2. Permanet Generation空间满
PermanetGeneration中存放的为一些class的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：
java.lang.OutOfMemoryError: PermGen space
为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。
3. CMS GC时出现promotion failed和concurrent mode failure
对于采用CMS进行旧生代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。
promotionfailed是在进行Minor GC时，survivor space放不下、对象只能放入旧生代，而此时旧生代也放不下造成的；concurrent mode failure是在执行CMS GC的过程中同时有对象要放入旧生代，而此时旧生代空间不足造成的。
应对措施为：增大survivorspace、旧生代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX:CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。
4. 统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间
这是一个较为复杂的触发情况，Hotspot为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。
例如程序第一次触发MinorGC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。

当新生代采用PSGC时，方式稍有不同，PS GC是在Minor GC后也会检查，例如上面的例子中第一次Minor GC后，PS GC会检查此时旧生代的剩余空间是否大于6MB，如小于，则触发对旧生代的回收。
除了以上4种状况外，对于使用RMI来进行RPC或管理的Sun JDK应用而言，默认情况下会一小时执行一次Full GC。可通过在启动时通过- java-Dsun.rmi.dgc.client.gcInterval=3600000来设置Full GC执行的间隔时间或通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc

如何解决单点故障

不支持跨区域负载
通过dnspod来完成故障自动切换。

haproxy 来解决单点故障-高可用

Keepalived组件：它可以检测web服务器的工作状态，如果该服务器出现故障被检测到，将其剔除服务器群中，直至正常工作后，keepalive会自动检测到并加入到服务器群里面。实现主备服务器发生故障时ip瞬时无缝交接。它是LVS集群节点健康检测的一个用户空间守护进程，也是LVS的引导故障转移模块（director failover）。Keepalived守护进程可以检查LVS池的状态。如果LVS服务器池当中的某一个服务器宕机了。keepalived会通过一 个setsockopt呼叫通知内核将这个节点从LVS拓扑图中移除。

1.插入数据库时，主键冲突，如何不报异常
答：insert replace/ignore into table
2.hashmap如何避免冲突
答：链表，先判断hashcode,再判断equals
hashcode相同，equals不一定相同---hashmap 链表
equals相同，hashcode一定相同
rehash ->扩容，重新分配index和进行拷贝，length->double
arrayList 值相同则hashcode相同
3.user-userId,address-addressId,addressName,userId

select username from user join useraddress on userId=userAddress.userId and addressName="sh"

select username from user a where exits (select userId from userAddress b where a.userId=b.userId and b.addressName="sh")
4.sql查询走几条索引
答：1条,原因，索引合并
5 加锁的三种方案
答：1.加redis锁
2.数据库字段unsigned
3.update的时候数据去减 为0的话则库存不足

6.mysql 单表查询走几条索引

1条--
7.clone浅拷贝与深拷贝

浅拷贝

x.clone()!=x but x.getClass()=x.clone().getClass()

深拷贝(deepCopy)

new x(x.data);

8.null不能做等值操作-见mysql