1.倒排索引

倒排索引是为了根据关键字来获取文档信息而出现的，可以帮助用户快速定位目标信息，是基于信息的关键字来构建的。

实现

1. 根据文章内容，先分词获得所有的关键词。
2. 统计所有关键词在所有文章中出现的频率和次数，然后建立一个倒排索引文件，关键字是按字符顺序排列的，然后出现在不同文章中的位置按照频率排序。

```
关键词              文章号[出现频率]                出现位置
guangzhou           1[2]                        3，6
he                  2[1]                        1
i                   1[1]                        4
live                1[2]                        2，5，
                    2[1]                        2
shanghai            2[1]                        3
tom                 1[1]                        1
```

2.说一下线程得创建方式

- 继承 Thread 类，重写 run 方法
- 实现 Runnable 接口，实现 run 方法
- 实现 Callable 接口，实现 call 方法
- 线程池ThreadPoolExecutor

  ①. newFixedThreadPool(int nThreads)

  创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

  ②. newCachedThreadPool()

  可缓存的线程池，线程池无限大。如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

  ③. newSingleThreadExecutor()

  这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

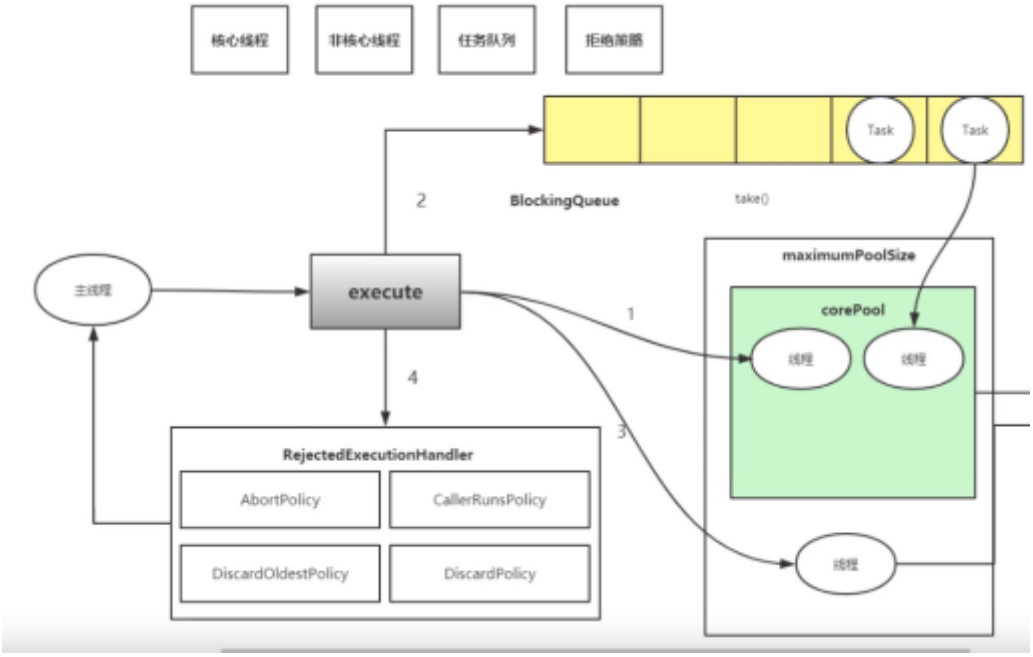
  ④. newScheduledThreadPool(int corePoolSize)

  创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

3.说一下线程池的使用和流程：

线程池的7个参数：①线程池中的核心线程数；②线程池中最大线程数；③闲置超时时间；④超时时间的单位；⑤线程池中的任务队列；⑥线程工厂；⑦拒绝策略

流程是：一般阻塞队列中的任务会先放到核心线程中进行处理，如果核心线程都在工作且任务队列满了，就放到非核心线程中处理；否则就放到任务队列中，如果任务队列也满了就会创建非核心线程来处理。如果非核心线程也都在工作了，那么还有任务来请求处理就会被拒绝或者等待。



4.说一下线程池参数的队列具体是什么。

阻塞队列：

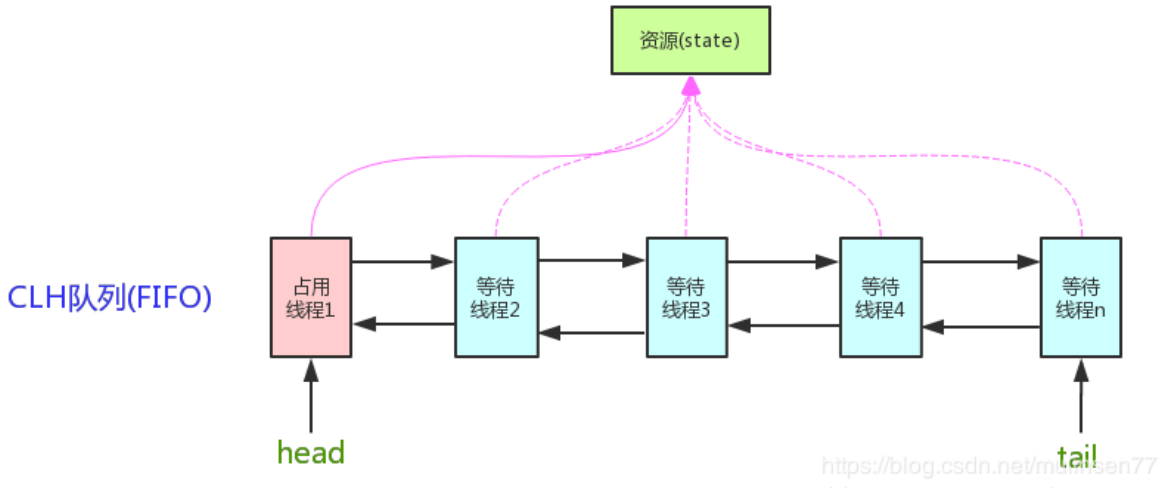
①ArrayBlockingQueue；②LinkedBlockingQuene；③SynchronousQuene；④PriorityBlockingQueue

5.LinkedBlockingQuene这个具体实现说说，什么时候阻塞。

LinkedBlockingQueue链表是基于单项链表结构的队列，按照先进先出(FIFO)的原则取出数据。使用了两个重入锁(Reetrantlock，put和take各一个)来维护链表的添加和删除； 节点。

AQS原理

存储结构：一个volatile修饰的int类型状态值（用于锁的状态变更），一个双向链表（用于存储等待中的线程）



1. 一个线程通过CAS乐观锁原理尝试去修改state值。

2. 如果成功，设置当前线程为工作线程；否则，会尝试再次通过CAS获取一次锁，接着将当前线程加入上面锁的双向链表（等待队列）中，最后不断自旋查询是否可以获取state锁。
3. 释放锁时，工作线程修改state值，然后设置工作线程为空，同时更新下一个链表中的线程等待节点。

6.i++操作线程安全吗？具体怎么改成线程安全。

不安全，AutomicInteger，不能用volatile（不能保证原子性），用synchronized也可以但是因为是悲观锁，效率没有乐观锁好。因为悲观锁会阻塞线程，被阻塞的线程唤醒需要OS来操作，那么就会从用户态切换到内核态，这个切换需要额外的时间开销。

7.AutomicInteger底层用了什么原理.

CAS，乐观锁原理，版本对比。

8.CAS会造成什么问题，怎么解决。

ABA问题，并让我举例描述一下。

例如A线程从内存中取出了1，这时候B线程也从内存中取出了1，但是A线程的执行时间远远小于B线程的执行时间，A线程先把1变成2写回到内存中，又把2变成1再次写回到内存中，这时候B线程开始进行CAS操作的时候发现内存中的值依然是1，然后线程B操作也会成功。但是在这期间B线程操作虽然完成了，但是并不代表这个过程是没有问题的 。

使用AtomicStampedReference类（这里忘记提加版本号这个解决办法了）

```
AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(num, version);
```

9.锁都了解哪些。使用ReentrantLock应该注意什么。

Synchronized,ReentrantLock,Semaphore,Atomic。

注意释放锁。

10.Java是怎么实现平台无关的？以及自己对jvm的了解

JAVA源文件会被编译成能被JAVA虚拟机执行的字节码文件，而JVM就是通过执行字节码文件，解释成具体平台上的机器指令来实现跨平台指令的。

首先通过类加载器（ClassLoader）会把 Java 代码转换成字节码，运行时数据区（Runtime Data Area）再把字节码加载到内存中，而字节码文件只是 JVM 的一套指令集规范，并不能直接交个底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

11.事务的四大特性以及四个隔离级别

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。

2. 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。

3. 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。

4. 持久性（Durability）

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

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。（更新数据时加入了共享锁即读锁，只能读不能写，解决丢失修改）

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。（读数据时加入了共享锁但是读操作完了以后就释放共享锁，使得不能重复读，更新数据时加入了排他锁即写锁，解决脏读的问题）

可重复读（REPEATABLE READ）

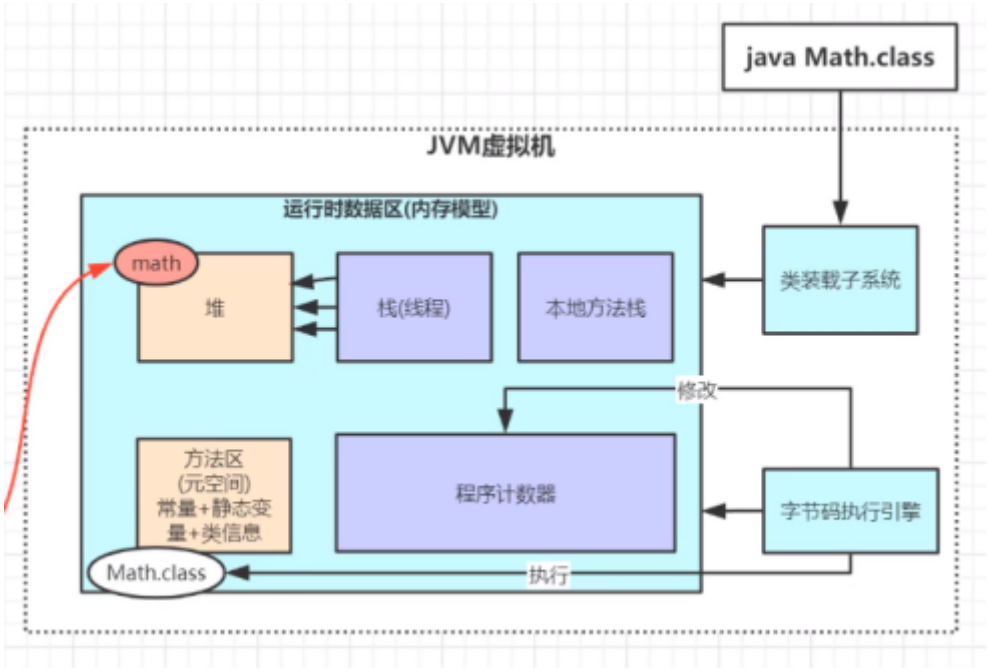
保证在同一个事务中多次读取同一数据的结果是一样的。（读数据时加入了共享锁，一直到事务执行完毕才释放共享锁，这样就可以解决不可重复读问题，更新数据时加入了排他锁即写锁。）

可串行化（SERIALIZABLE）

强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题。

该隔离级别需要加锁实现，因为要使用加锁机制保证同一时间只有一个事务执行，也就是保证事务串行执行。（读数据时对**整个数据表**加入了共享锁，更新数据时对**整个数据表**加入了排他锁即写锁，解决幻影读问题）

12.Java内存分配

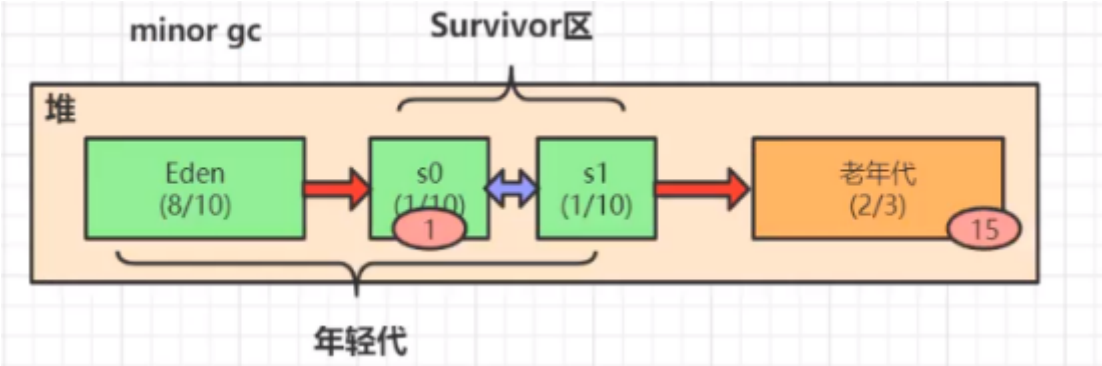


动态连接是一个将符号引用解析为直接引用的过程。当java虚拟机执行字节码时，如果它遇到一个操作码，这个操作码第一次使用一个指向另一个类的符号引用

那么虚拟机就必须解析这个符号引用。在解析时，虚拟机执行两个基本任务

1.查找被引用的类，（如果必要的话就装载它）

2.将符号引用替换为直接引用，这样当它以后再次遇到相同的引用时，它就可以立即使用这个直接引用，而不必花时间再次解析这个符号引用了。



13.算法一：a和b的值互换，不开辟新空间

```
public static  void getchange1(){
      int x=3,y=9;
      x=x^y;
      y=x^y;
      x=x^y;
      System.out.println("getchange1:x="+x+";y="+y);
   }
   public static void getchange2(){
       int x=3,y=9;
       x=x+y;
       y=x-y;
       x=x-y;
       System.out.println("getchange2:x="+x+";y="+y);
   }
   public static void getchange3(){
       int x=3,y=9;
```

```
        x=x-y;
        y=x+y;
        x=y-x;
        System.out.println("getchange3:x="+x+";y="+y);
    }
    public static void getchange4(){
        int x=3,y=9;
        x=x*y;
        y=x/y;
        x=x/y;
        System.out.println("getchange4:x="+x+";y="+y);
    }
```

14.算法二：1000个数里面求最大的第几个数（topK问题）

```
/** 输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是
1,2,3,4
 */
public static ArrayList<Integer> GetLeastNumbers(int[] input,int k){
    ArrayList<Integer> resultList = new ArrayList<Integer>();
    if (input.length<=0 || k<= 0 ){
      return resultList;
    }
    int h = 1;
    PriorityBlockingQueue<Integer> queue = new PriorityBlockingQueue<Integer>(
           h, new Comparator<Integer>(){
                public int compare(Integer o1,Integer o2){
                 return o2.compareTo(o1);
                }
            }
    );

    for (int i = 0;i<input.length;i++){
        if(i<k){
            queue.add(input[i]);
        }else {
            if (queue.peek()>input[i]){
                queue.poll();
                queue.add(input[i]);
            }
        }
    }
    while (!queue.isEmpty() && queue.size()<=k){
        resultList.add(queue.poll());
    }
    Collections.reverse(resultList);
    return resultList;
}
```

15.静态变量与成员变量

1.两个变量的生命周期不同

成员变量随着对象的创建而存在，随着对象的被回收而释放；

静态变量随着类的加载而存在，随着类的消失而消失；

2.调用方式不同

成员变量只能被对象调用；

静态变量能被对象调用，还能被类名调用；

3.别名不同

成员变量也称为实例变量；

静态变量被称为类变量；

4.数据存储位置不同

成员变量数据存储在堆内存的对象中，所以也叫对象的特有数据；

静态变量数据存储在方法区的静态区，所以也叫对象的共享数据；

16.线程池 五个核心参数

1、corePoolSize：核心线程数
* 核心线程会一直存活，及时没有任务需要执行
    * 当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理
    * 设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭

2、queueCapacity：任务队列容量（阻塞队列）
* 当核心线程数达到最大时，新任务会放在队列中排队等待执行

3、maxPoolSize：最大线程数
* 当线程数>=corePoolSize，且任务队列已满时。线程池会创建新线程来处理任务
    * 当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常

4、 keepAliveTime：线程空闲时间
* 当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize
    * 如果allowCoreThreadTimeout=true，则会直到线程数量=0

5、allowCoreThreadTimeout：允许核心线程超时
6、rejectedExecutionHandler：任务拒绝处理器

```
* 两种情况会拒绝处理任务：
    - 当线程数已经达到maxPoolSize，切队列已满，会拒绝新任务
    - 当线程池被调用shutdown()后，会等待线程池里的任务执行完毕，再shutdown。如果在调用
shutdown()和线程池真正shutdown之间提交任务，会拒绝新任务
    * 线程池会调用rejectedExecutionHandler来处理这个任务。如果没有设置默认是
AbortPolicy，会抛出异常
    * ThreadPoolExecutor类有几个内部实现类来处理这类情况：
    - AbortPolicy 丢弃任务，抛运行时异常
    - CallerRunsPolicy 主线程执行任务
    - DiscardPolicy 忽视，什么都不会发生
    - DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务
    * 实现RejectedExecutionHandler接口，可自定义处理器
```

17.mybatis中的#和$的区别

1、传入的参数在SQL中显示不同

#传入的参数在SQL中显示为字符串（当成一个字符串），会对自动传入的数据加一个双引号。

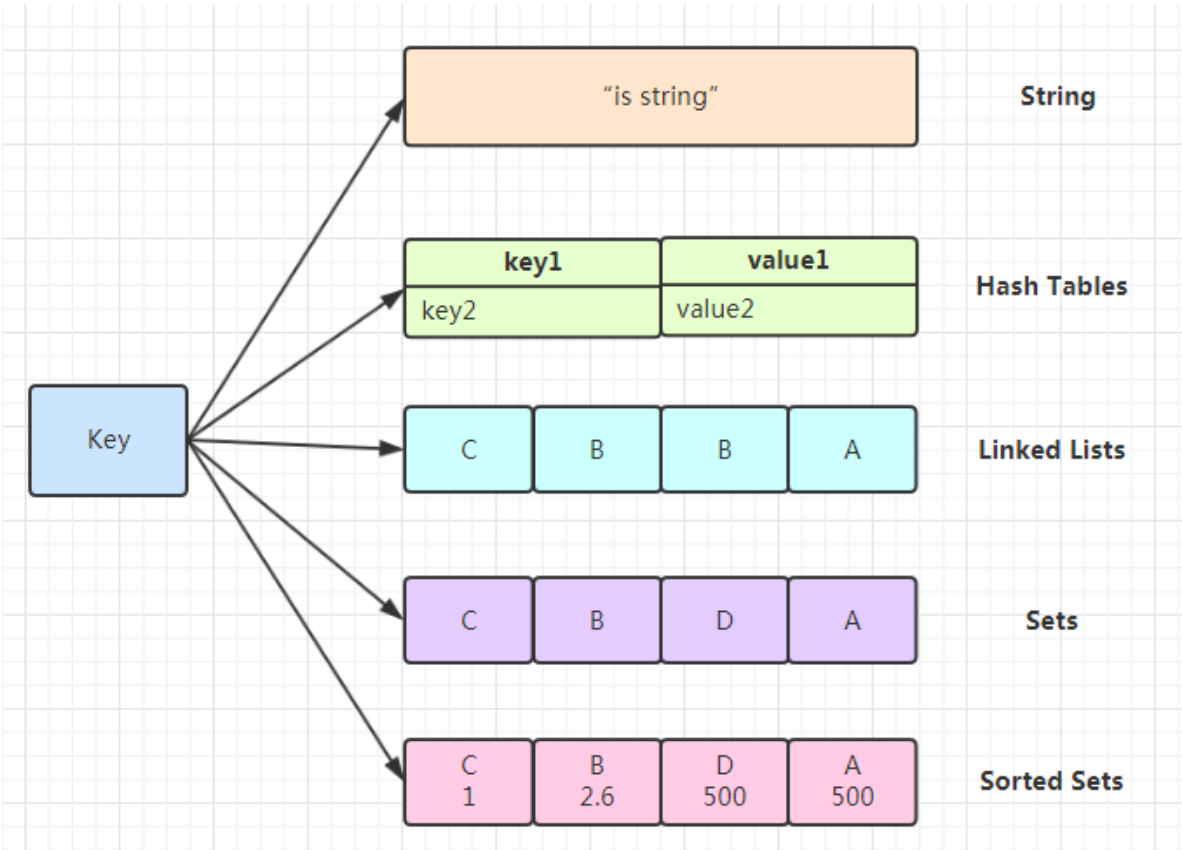
$传入的参数在SqL中直接显示为传入的值

所以#可以防止SQL注入的风险（语句的拼接）

2.$方式一般用于传入数据库对象，例如传入表名，列名。

3.一般能用#的就别用$.

18.redis数据结构



19.redis配置方式

20.sql语句查询 包含 ab值的查询。

```
select * from table1 where num like '%[ab]%'
```

21.MD5,DES,RSA这三种算法的区别。

- DES是对称密码算法，就是加密密钥能够从解密密钥中推算出来，反过来也成立。密钥较短，加密处理简单，加解密速度快，适用于加密大量数据的场合。
- RSA是非对称算法，加密密钥和解密密钥是不一样的，或者说不能由其中一个密钥推导出另一个密钥。密钥尺寸大，加解密速度慢，一般用来加密少量数据，比如DES的密钥。
- SHA1 和 MD5 是散列算法，将任意大小的数据映射到一个较小的、固定长度的唯一值。加密性强的散列一定是不可逆的，这就意味着通过散列结果，无法推出任何部分的原始信息。任何输入信息的变化，哪怕仅一位，都将导致散列结果的明显变化，这称之为雪崩效应。散列还应该是防冲突的，即找不出具有相同散列结果的两条信息。具有这些特性的散列结果就可以用于验证信息是否被修改。MD5 比 SHA1 大约快 33%。

22.一个文件中 755中，7的含义。

- 第一位7，代表文件所有者拥有的权限为可读（4）+可写（2）+可执行（1）
- 第二位5，代表文件所有者同组用户的权限为可读（4）+不可写（0）+可执行（1）
- 第三位5，代表公共用户的权限为可读（4）+不可写（0）+可执行（1）
- 755表示该文件所有者对该文件具有读、写、执行权限，该文件所有者所在组用户及其他用户对该文件具有读和执行权限。

23.linux查看文件，一般用什么命令?

- cat 查看内容较少的文件，不支持翻页。
- less 查看大文件，上下键翻页，q退出。
- head -n 显示头n行
- tail -n 显示尾n行

24.讲一讲泛型与反射.

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

Java反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

Java 泛型在运行的时候是会进行类型擦除，泛型信息只存在于代码编译阶段。即对于List和List在运行阶段都是List.class,泛型信息被擦除了。

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
assert(l1.getClass() == l2.getClass()); //结果为真
```

```
//指定获得Bean中list属性, public List<String> list;
Field list = Bean.class.getField("list");
// 属性对应的Class如果是List或其子类
if (List.class.isAssignableFrom(list.getType())) {
    //获得 Type
    Type genericType = list.getGenericType();
    //ParameterizedType
    if (genericType instanceof ParameterizedType) {
        //获得泛型类型,如果是map<String,Integer>的话0代表Key,1代表value
        Type type = ((ParameterizedType) genericType).getActualTypeArguments()
[0];
    }
}
```

25.Spring中的Context

Spring的核心是容器，而容器并不唯一，框架本身就提供了很多个容器的实现。

大概分为两种类型：一种是不常用的BeanFactory，这是最简单的容器，只能提供基本的DI功能；

还有一种就是继承了BeanFactory后派生而来的**应用上下文**，其抽象接口也就是ApplicationContext，它能提供更多企业级的服务，例如解析配置文本信息等等，这也是应用上下文实例对象最常见的应用场景。有了上下文对象，我们就能向容器注册需要Spring管理的对象了。

只要将你需要IOC容器替你管理的对象基于xml也罢，java注解也好，总之你要将需要管理的对象（Spring中我们都称之问bean）、bean之间的协作关系配置好，然后利用应用上下文对象加载进我们的Spring容器，容器就能为你的程序提供你想要的对象管理服务了。

26.JVM中 堆和栈的区别。

栈主要作用是用来存放线程的局部变量，其次还有操作数栈（存放一些数字，如用于赋值、逻辑/算术运算等操作），动态链接和方法出口（记录的是调用这个方法的线程执行位置，以便这个方法执行完毕后可以继续执行当前线程的下一行代码）。

堆主要存放的是一些new的对象。

区别

1. 栈内存存储的是局部变量而堆内存存储的是实体；
2. 栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短；
3. 栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。

27.多线程如何实现资源共享?

因为一个线程只能启动一次，通过Thread实现线程时，线程和线程所要执行的任务是捆绑在一起的。也就使得一个任务只能启动一个线程，不同的线程执行的任务是不相同的，不能让两个线程共享彼此任务中的资源。

一个任务可以启动多个线程，通过Runnable方式实现的线程，实际是开辟一个线程，将任务传递进去，由此线程执行。可以实例化多个 Thread对象，将同一任务传递进去，也就是一个任务可以启动多个线程来执行它。这些线程执行的是同一个任务，所以他们的资源是共享。

28.了解CPU的时间容片吗?

时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。而不会造成CPU资源浪费。在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。但在微观上：由于只有一个CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

在Linux的内核处理过程中，每一个进程默认会有一个固定的时间片来执行命令（默认为1/100秒），这段时间内进程被分配到CPU，然后独占使用。如果使用完，同时未到时间片的规定时间，那么就主动放弃CPU的占用，如果到时间片尚未完成工作，那么CPU的使用权也会被收回，进程将会被中断挂起等待下一个时间片。

29.垃圾回收和算法

一般有两种方法垃圾回收来判断：

- 引用计数器：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；
- 可达性分析：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

  GC roots: 一般为方法区中常量引用/类中静态引用的对象、栈中局部变量引用的对象等。

算法：

- 标记-清除算法：会引起内存碎片。
- 标记-整理法：首先标记出所有需要回收的对象，让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。
- 引用计数法
- 复制算法

回收器：

- 新生代回收器：Serial、ParNew、Parallel Scavenge
- 老年代回收器：Serial Old、Parallel Old、CMS
- 整堆回收器：G1

新生代垃圾回收器一般采用的是**复制算法**，复制算法的优点是效率高，缺点是内存利用率低；

老年代回收器一般采用的是**标记-整理**的算法进行垃圾回收。

30.redis数据结构，持久化方式。机器宕机怎么办，丢失数据怎么办

cglib

- RDB（Redis Database）：指定的时间间隔能对你的数据进行快照存储。

  优点：

  一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

  缺点：

  1） 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

  2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

- AOF（Append Only File）：每一个收到的写命令都通过write函数追加到文件中。

  优点：

  由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心

  缺点：

  对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

31.mybatis 如何获取自增ID

keyProperty="id" 和useGeneratedKeys="true"

```
<insert id="xxx" keyProperty="id" useGeneratedKeys="true" parameterType="Users">
         insert  into users (name ,phone)
         values (#{name},#{phone});
        </insert>
```

32.mybatis 一二级缓存

一级缓存是SqlSession级别的缓存，是一直开启的，我们关闭不了它；

一级缓存失效的情况：

1. sqlSession不同
2. sqlSession相同，查询条件不同
3. sqlSession相同，两次查询之间执行了增删改操作！
4. sqlSession相同，手动清除一级缓存

二级缓存

- 一个会话查询一条数据，这个数据就会被放在当前会话的一级缓存中；
- 如果当前会话关闭了，这个会话对应的一级缓存就没了；但是我们想要的是，会话关闭了，一级缓存中的数据被保存到二级缓存中；
- 新的会话查询信息，就可以从二级缓存中获取内容；
- 不同的mapper查出的数据会放在自己对应的缓存（map）中；

1、开启全局缓存 【mybatis-config.xml】

```
<setting name="cacheEnabled" value="true"/>
```

2、去每个mapper.xml中配置使用二级缓存，这个配置非常简单；【xxxMapper.xml】

```
<cache/>

官方示例=====>查看官方文档
<cache
 eviction="FIFO"
 flushInterval="60000"
 size="512"
 readOnly="true"/>
这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。
```

33.设计模式工厂模式讲讲

简单工厂模式：一个抽象的接口，多个抽象接口的实现类，一个工厂类，用来实例化抽象的接口

工厂方法模式：有四个角色，抽象工厂模式，具体工厂模式，抽象产品模式，具体产品模式。不再是由一个工厂类去实例化具体的产品，而是由抽象工厂的子类去实例化产品

抽象工厂模式：与工厂方法模式不同的是，工厂方法模式中的工厂只生产单一的产品，而抽象工厂模式中的工厂生产多个产品

34.redis数据如何和mysql的数据保持一致

读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：数据库和缓存更新，就容易出现缓存(Redis)和数据库（MySQL）间的数据一致性问题。

不管是先写MySQL数据库，再删除Redis缓存；还是先删除缓存，再写库，都有可能出现数据不一致的情况。举一个例子：

1.如果删除了缓存Redis，还没有来得及写库MySQL，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。

2.如果先写了库，在删除缓存前，写库的线程宕机了，没有删除掉缓存，则也会出现数据不一致情况。

方案：

1.第一种方案：采用延时双删策略

1. 先删除缓存
2. 再写数据库

3. 休眠500毫秒
4. 再次删除缓存

2、第二种方案：异步更新缓存(基于订阅binlog的同步机制)

一旦MySQL中产生了新的写入、更新、删除等操作，就可以把binlog相关的消息推送至Redis，Redis再根据binlog中的记录，对Redis进行更新。其实这种机制，很类似MySQL的主从备份机制，因为MySQL的主备也是通过binlog来实现的数据一致性。

35.集合中 arrayList 和linkedlist有什么区别?

最明显的区别是 ArrrayList底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。

36.hashMap 为什么设置初始化为16

length为2的整数次幂的话，位运算以后分布会均匀，因为（len-1）的二进制位为1111的这种形式进行位运算才分布均匀。

capacity 为 2 的整数次幂的话，为偶数，这样 capacity-1 为奇数，奇数的最后一位是 1，这样便保证了 h&(capacity-1) 的最后一位可能为 0，也可能为 1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性；

37.hashCode 怎么对应桶的位置?

key % (length-1)

38. 线程同步有哪些方法?

信号量，互斥量，临界区

39.线程同步使用哪些锁?

synchronized，volatile，lock

40.有一个场景，现在一张表有几十万的数据，然后10个线程，对它并发计算，然后计算完了之后通知，该怎么设计？(每个线程处理每个线程的事情，然后做个通知)

41.说说分布式锁?

一般的sychronized加锁只能对同一个JVM中的进程下加锁，但是如果一个WEB项目是部署在不同的机器上，对于不同的机器，实际上是不同的JVM了，那么sychronized加锁就不起作用了，需要用分布式锁来满足开发需求。

通过对可能并发执行的代码加入分布式锁，利用redis中的setnx命令来加分布式锁，首先判断分布式锁是否有被加锁，没的话，这个进程可以执行接下来的代码，否则不可以执行。最终业务代码执行完毕以后必须要删除这个setnx锁。

```
Setnx lock true  ->  加分布式锁

Setnx lock false  -> 已经加了分布式锁，不能再加了

Get lock -> 显示lock状态为true

........ 执行代码..........

Del lock  ->  删除分布式锁
```

问题：

1、简易分布式锁可能会因为中间的业务代码抛出异常导致分布式锁没有被删除出错，可以用**try finally**解决。

2、但是如果机器重启的话，也会因为锁没有删除出错，**可以在加分布式锁时设置锁的存活时间（比如10s）**，过期会自动释放锁。

3、如果一个线程业务的执行时间比锁的存活时间更长，比如15s，那么这个线程在执行到一半时锁就会被释放，可是业务还没执行完，还需要执行5s。此时如果第二个线程申请加锁也能加成功，并且第一个线程业务结束时释放的锁会是第二个线程加的锁，那么redis锁就失效了，没有了锁的意义。**解决方法：可以将锁的值设置为一个UUID值，每次释放锁的时候判断锁的值是否是这个线程一开始设定的UUID值，如果是的话，代表是这个线程加的分布式锁，可以释放。**

4、如果业务执行的时间（15s）大于锁的存活时间（10s），除了会出现一个线程释放另一个线程的锁的问题（问题3），还会出现多个线程共同执行一部分同步代码的问题。**解决方法：可以在执行业务前设置一个后台分线程，每隔一段时间（3s）判断上述的锁是否还存活，如果存活的话，重新设置存的存活时间为10s。**

42.synchronized和Lock的区别。lock底层实现原理

- 首先synchronized是关键字，Lock是个java类；
- synchronized是JVM提供的，Lock是JDK提供的;
- synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；
- synchronized会自动加锁解锁，Lock需在手动加锁解锁；
- 用synchronized不能退出等待，而lock如果获取不到锁可以停止等待；
- Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

底层是AQS原理。

43.悲观锁和乐观锁说一下

悲观锁

当我们要对一个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。

乐观锁

乐观锁的“乐观情绪”体现在，它认为数据的变动不会太频繁。因此，它允许多个事务同时对数据进行变动。乐观锁通常是通过在表中增加一个版本(version)或时间戳(timestamp)来实现，其中，版本最为常用。事务在从数据库中取数据时，会将该数据的版本也取出来(v1)，当事务对数据变动完毕想要将其更新到表中时，会将之前取出的版本v1与数据中最新的版本v2相对比，如果v1=v2，那么说明在数据变动期间，没有其他事务对数据进行修改，此时，就允许事务对表中的数据进行修改，并且修改时version会加1，以此来表明数据已被变动。如果，v1不等于v2，那么说明数据变动期间，数据被其他事务改动了，此时不允许数据更新到表中，一般的处理办法是通知用户让其重新操作。不同于悲观锁，乐观锁是人为控制的。

44.算法：数组找出平衡点，前一部分和等于后一部分的和

```
public static int balance_point_opt(ArrayList<Integer> a){

        int left_sum = 0;
        int right_sum = 0;
        for(int i=1; i<a.size();i++)
            right_sum += a.get(i);

        // 下标为1,2,3,...,a.length-2时的情形
        for(int i=1; i<a.size()-1;i++){
            left_sum += a.get(i-1);
            right_sum -= a.get(i);
            if(left_sum == right_sum)
                return i;
        }

        return -1;
    }
```

45.cglib动态代理

动态代理：

当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

普通的动态代理：

首先必须定义一个接口，还要有一个InvocationHandler(将实现接口的类的对象传递给它)处理类。再有一个工具类Proxy(习惯性将其称为代理类，因为调用他的newInstance()可以产生代理对象,其实他只是一个产生代理对象的工具类）。利用到InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

cglib动态代理

1.业务类，不需要实现接口

```
package com.test.cglib;

public class People {
    public void show(){
        System.out.println("我是芙蓉 ...");
    }
}
```


2.代理类，需要实现MethodInterceptor接口

```
package com.test.cglib;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class CglibTest implements MethodInterceptor {
    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
        System.out.println("增强之前 ...");
        Object o1 = methodProxy.invokeSuper(o, objects);
        System.out.println("增强之后 ...");
        return o1;
    }

    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(People.class);
        // 设置回调方法
        CglibTest cglibTest = new CglibTest();
        enhancer.setCallback(cglibTest);
        People people = (People)enhancer.create();
        people.show();
    }
}
```


3.测试结果

```
增强之前 ...
我是芙蓉 ...
增强之后 ...

Process finished with exit code 0
```


46.spring事务

JDBC事务

```
<bean
id="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTran
sactionManager">
       <property name="dataSource" ref="dataSource" />
</bean>
```

配置好事务管理器后我们需要去配置事务的通知

```
<!--配置事务通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
   <tx:attributes>
       <!--配置哪些方法使用什么样的事务,配置事务的传播特性-->
       <tx:method name="add" propagation="REQUIRED"/>
       <tx:method name="delete" propagation="REQUIRED"/>
       <tx:method name="update" propagation="REQUIRED"/>
       <tx:method name="search*" propagation="REQUIRED"/>
       <tx:method name="get" read-only="true"/>
       <tx:method name="*" propagation="REQUIRED"/>
   </tx:attributes>
</tx:advice>
```

配置AOP

导入aop的头文件!

```
<!--配置aop织入事务-->
<aop:config>
   <aop:pointcut id="txPointcut" expression="execution(* com.kuang.dao.*.*
(..))"/>
   <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

47.用户态和系统态

为什么要有这两种状态：在计算机系统中，通常运行着两类程序：系统程序和应用程序。CPU 的指令集分为特权指令（启动I/O设备指令、程序状态字指令）和非特权指令（逻辑运算指令，存数取数指令）两类。对于那些危险的指令，只允许操作系统使用，应用程序只能使用那些不会造成灾难的指令。

为了保证系统程序不被应用程序有意或无意地破坏，为计算机设置了两种状态:

系统态(也称为管态或核心态)，操作系统在系统态运行——运行操作系统程序

用户态(也称为目态)，应用程序只能在用户态运行——运行用户程序

在实际运行过程中，处理机会在系统态和用户态间切换。

用户态->内核态：

1.系统调用：用户进程通过系统调用申请使用操作系统提供的服务程序来完成工作，比如read()、fork()等。系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现的。

2.中断：当外围设备完成用户请求的操作后，会想CPU发送中断信号。这时CPU会暂停执行下一条指令（用户态）转而执行与该中断信号对应的中断处理程序（内核态）

3.异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

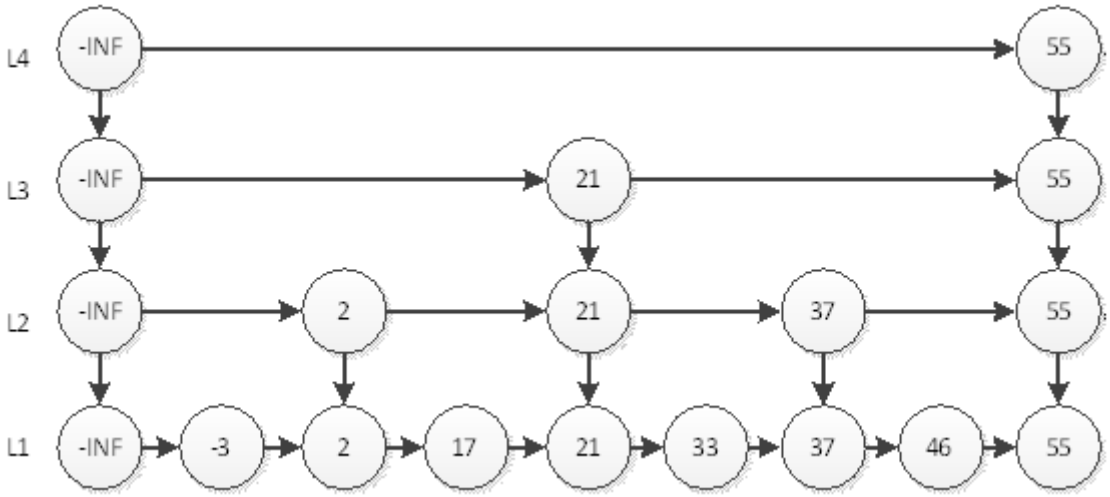
内核态->用户态：设置程序状态字PSW为用户态。

48.交换机属于哪一层

数据链路层

49.skiplist跳表

出现的原因，如果我需要查询一个链表的最后一个，那么需要查找n次。



我们可以看到，最耗时的访问46需要6次查询。即L4访问55，L3访问21、55，L2访问37、55，L1访问46。我们直觉上认为，这样的结构会让查询有序链表的某个元素更快。**O（logn），如果要做到严格*O(logn)*，上层结点个数应是下层结点个数的 1/2 。**

先讨论插入，我们先看理想的跳跃表结构，L2层的元素个数是L1层元素个数的1/2，L3层的元素个数是L2层的元素个数的1/2，以此类推。从这里，我们可以想到，只要在插入时尽量保证上一层的元素个数是下一层元素的1/2，我们的跳跃表就能成为理想的跳跃表。那么怎么样才能在插入时保证上一层元素个数是下一层元素个数的1/2呢？很简单，抛硬币就能解决了！假设元素X要插入跳跃表，很显然，L1层肯定要插入X。那么L2层要不要插入X呢？我们希望上层元素个数是下层元素个数的1/2，所以我们有1/2的概率希望X插入L2层，那么抛一下硬币吧，正面就插入，反面就不插入。那么L3到底要不要插入X呢？相对于L2层，我们还是希望1/2的概率插入，那么继续抛硬币吧！以此类推，元素X插入第n层的概率是(1/2)的n次。这样，我们能在跳跃表中插入一个元素了。

50.ConcurrentHashmap若全部数据都在一个segment怎么改进

段的个数设置为2次幂，使得映射均匀。

51.volatile关键字

通过**缓存一致性**来解决变量同步的问题，当一个用volatile关键字修饰的变量被修改时，底层会向CPU发出一个lock前缀的汇编命令，它会锁定修改这个变量的这块内存区域的缓存（即这个线程的工作内存），然后立即写回到主存中，同时这个写回内存的操作会引起其他CPU里缓存了这个内存的数据无效（即其他线程拷贝了这部分地址的数据会失效）。其他CPU是通过**总线嗅探机制**感知到数据的变化从而让自己的缓存的数据失效。当CPU发现自己需要用到的数据失效以后就会重新从主内存read数据。

52.OOM（Out Of Memory）内存泄露，出现原因，解决方法

原因：

① 分配的少了：比如虚拟机本身可使用的内存（一般通过启动时的VM参数指定）太少。

② 应用用的太多，并且用完没释放，浪费了。此时就会造成内存泄露或者内存溢出。

方法：

1. java.lang.OutOfMemoryError: Java heap space ——>java堆内存溢出。一般由于内存泄露或者堆的大小设置不当引起。对于内存泄露，需要通过内存监控软件查找程序中的泄露代码，而堆大小可以通过虚拟机参数-Xms（设定程序启动时占用内存大小）,-Xmx（设定程序运行期间最大可占用的内存大小）等修改。
2. java.lang.OutOfMemoryError: PermGen space ——>java永久代溢出，即方法区溢出了，因为大量的Class信息存储于方法区。此种情况可以通过更改方法区的大小来解决，使用类似-XX:PermSize=64m -XX:MaxPermSize=256m的形式修改
3. java.lang.StackOverflowError ——> 不会抛OOM error，但也是比较常见的Java内存溢出。一般是由于程序中存在死循环或者深度递归调用造成的，栈大小设置太小也会出现此种溢出。可以通过虚拟机参数-Xss来设置栈的大小

53.工厂方法模式和抽象工厂模式区别

54.装饰模式，适配器模式

装饰模式：通过给一个方法增加一些新的功能，而不修改其原有的功能。

适配器模式：输入的内容不满足程序的要求，用适配器模式转为合适的输入格式。

55.锁，自旋锁、轻量级锁、偏向锁

锁从宏观上分类，分为悲观锁与乐观锁

锁重量级细化分为四种：自旋锁，轻量级锁，偏向锁，重量级锁。前三种是乐观锁，后一种是悲观锁。

自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。执行一段无意义的循环即可（自旋）来等待，且不会释放CPU资源。

偏向锁，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，这种情况下，就会给线程加一个偏向锁。如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

轻量级锁，是由偏向锁升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁竞争的时候，偏向锁就会升级为轻量级锁。

重量级锁synchronized，synchronized关键字并非一开始就该对象加上重量级锁，也是从偏向锁，轻量级锁，再到重量级锁的过程。当系统检查到锁是重量级锁之后，会把等待想要获得锁的线程进行阻塞，被阻塞的线程不会消耗cpu。但是阻塞或者唤醒一个线程时，都需要操作系统来帮忙，这就需要从用户态转换到内核态，而转换状态是需要消耗很多时间的，有可能比用户执行代码的时间还要长。

56.Sping中IOC怎么将类注入到里面去，并且怎么实例化对象

1. 构造器注入
2. set注入（要求被注入的属性，必须有set方法）
3. p命名（set注入）和c命名（构造器注入）注入

57.AOP的切点怎么切

execution(*packagename.ClassName.MethodName(..))

*代表返回类型任意

..代表参数个数任意

```
 <aop:pointcut id="pointcut" expression="execution(*
com.kuang.service.UserServiceImpl.*(..))"/>
```

58.Sleep()和wait()的区别，使用wait()方法后，怎么唤醒线程

sleep()：方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

wait()：wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池（不会竞争锁），同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程进入到锁池去竞争锁。

用notify()或者notifyAll()

59.用过哪些线程池，说一下常见的四种线程池和区别

①. newFixedThreadPool(int nThreads)

创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

②. newCachedThreadPool()

可缓存的线程池，线程池无限大。如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

③. newSingleThreadExecutor()

这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

④. newScheduledThreadPool(int corePoolSize)

创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

60.redis对于hash数据结构，如果要删除数据，那么redis的底层是怎么处理的

61.Redis缓存是怎么运用的

62.索引的优点与缺点

优点:

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
第二，可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。
第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
第四，在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

缺点:

第一，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。
第二，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。
第三，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。

63.提高并发的以及响应速度

1.页面渲染可以开启缓存，关闭日志

2.nginx实现动静分离

3.数据库优化就是加索引

4.把自己写的业务逻辑优化 比如多级查询就一次性查出所有数据 然后对数据进行查询 避免多次访问数据库