1. redis的淘汰机制

Redis清除key的方式：定期删除+惰性删除

定期删除：redis设定每隔100ms**随机(因为大量数据全部遍历影响性能)**抽取设置过期时间的key，对其进行检查，如果已经过期就删除

惰性删除：每获取key时会对key进行判断是否还存活，如果已经过期了删除

注：redis中过期的key 不会马上删除，因为定期删除可能正好每抽取到它，我们也没有访问到它触发惰性删除。

redis配置文件中可以设置maxmemory,内存的最大使用量,到达限度时会执行内存淘汰机制。



1. Springboot的自动装配原理
   1. SpringBoot启动时加载主配置类，开启了自动配置功能@EnableAutoConfiguration
   2. @EnableAutoConfiguration作用：将类路径下META-INF/spring.factories里面配置的所有EnableAutoConfiguration的值加入到了容器中
   3. 每一个自动配置类进行自动配置功能
   4. 所有在配置文件中能配置的属性都是在xxxxPropertites类中封装着，配置文件能配置什么就可以参照某个功能对应的这个属性类；
2. Dubbo支持哪些通信协议和序列化协议?
   * 1. dubboo协议(默认)，单一长连接，NIO异步通信，基于hessian作为序列化协议。适用于传输数据量很小的高并发场景。
     2. rmi协议：java二进制序列化，多个短连接，适和消费者和提供者数量差不多。适用于文件的传输(用的少)
     3. Hessian协议：hessian序列化协议，多个短连接，适用于提供者数量比消费者数量还多。适用于文件的传输(用的少)
     4. http协议：json序列化
     5. Webservice：SOAP文本序列化。
3. ES读写数据过程：
   1. 写数据过程：
      1. 客户端通过hash选择一个node发送请求，这个node被称做coordinating node（协调节点），
      2. 协调节点对docmount进行路由，将请求转发给到对应的primary shard
      3. primary shard 处理请求，将数据同步到所有的replica shard
      4. 此时协调节点，发现primary shard 和所有的replica shard都处理完之后，就反馈给客户端。
   2. 读数据过程：
      1. 客户端发送get请求到任意一个node节点，然后这个节点就称为协调节点，
      2. 协调节点对document进行路由，将请求转发到对应的node，此时会使用随机轮询算法，在primary shard 和replica shard中随机选择一个，让读取请求负载均衡，
      3. 接收请求的node返回document给协调节点，
      4. 协调节点，返回document给到客户端。
   3. 搜索过程：
      1. 客户端发送请求到协调节点，
      2. 协调节点将请求发送到所有的shard对应的primary shard或replica shard ；
      3. 每个shard将自己搜索到的结果返回给协调节点，返回的结果是document.id或者自己自定义id，然后协调节点对数据进行合并排序操作，最终得到结果。
      4. 最后协调节点根据id到各个shard上拉取实际的document数据，最后返回给客户端。
4. Jdk1.8新特性：
   1. Lambda表达式
   2. 函数式接口
   3. 方法引用和构造器调用
   4. Stream API
   5. 接口中的默认方法和静态方法
   6. 新时间日期API
   7. 在jdk1.8中对hashMap等map集合的数据结构优化。hashMap数据结构的优化：

原来的hashMap采用的数据结构是哈希表（数组+链表），hashMap默认大小是16，一个0-15索引的数组，如何往里面存储元素，首先调用元素的hashcode  
方法，计算出哈希码值，经过哈希算法算成数组的索引值，如果对应的索引处没有元素，直接存放，如果有对象在，那么比较它们的equals方法比较内容  
如果内容一样，后一个value会将前一个value的值覆盖，如果不一样，在1.7的时候，后加的放在前面，形成一个链表，形成了碰撞，在某些情况下如果链表  
无限下去，那么效率极低，碰撞是避免不了的  
加载因子：0.75，数组扩容，达到总容量的75%，就进行扩容，但是无法避免碰撞的情况发生  
在1.8之后，在数组+链表+红黑树来实现hashmap，当碰撞的元素个数大于8时 & 总容量大于64，会有红黑树的引入  
除了添加之后，效率都比链表高，1.8之后链表新进元素加到末尾  
ConcurrentHashMap (锁分段机制)，concurrentLevel,jdk1.8采用CAS算法(无锁算法，不再使用锁分段)，数组+链表中也引入了红黑树的使用

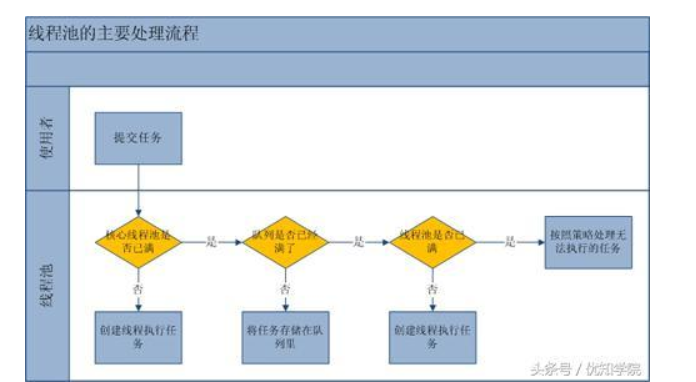
1. 分布式锁的实现方式:
   1. 数据库解决方案：加排他锁
      1. 数据库建一张表，字段方法名并且作为唯一性，当一个方法执行时插入，则相当于获得锁，其他线程将无法访问，方法执行完则释放锁。
      2. 使用select \* from user u where username = '' for update 来对记录加上排他锁。操作完成后使用commit命令释放锁。
   2. 基于缓存实现：例如使用redis实现分布式锁
      1. 使用setnx 在key不存在时设置值，实现加锁，其他线程无法在创建该key的锁
      2. 设置成功的，就等于拿到了锁，执行业务
      3. 执行业务之后，使用del删除锁
      4. 为了防止执行业务过程中出现异常，第3步没有执行，第1步执行完毕时， 使用expire 设置过期时间，防止出现死锁等现象，让多个请求等待）
   3. zookeeper实现分布式锁：
      1. 首先zookeeper中我们可以创建一个持久化节点
      2. 然后再在该持久化节点下创建自己的临时顺序节点
      3. 获取所有的1中创建的持久化节点下的所有子节点，并排序
      4. 判读自己创建的节点是否最小值（第一位）
      5. 如果是，则获取得到锁，执行自己的业务逻辑，最后删除这个临时节点。
      6. 如果不是最小值，则需要监听自己创建节点前一位节点的数据变化，并阻塞。
      7. 当前一位节点被删除时，我们需要通过递归来判断自己创建的节点是否在是最小的，如果是则执行5）；如果不是则执行6）（就是递归循环的判断)
2. 线程池原理：

预先启动一些线程，线程无限循环从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已，如此反复。

1. 线程池的大小配置(根据任务的类型来配置线程池大小)：
   1. 如果是**CPU密集型任务**，就需要尽量压榨CPU，参考值可以设为**NCPU+1**
   2. 如果是**IO密集型任务**，参考值可以设置为**2\*NCPU**

只是一个参考值，具体的设置还需要根据实际情况进行调整

1. 线程池处理流程：
   1. 判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。
   2. 线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
   3. 判断**线程池里的线程**是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。



1. 线程池状态(RUNNING、SHURDOWN、STOP、TERMINATED。)：
   1. 线程池创建后处于RUNNING状态。
   2. 调用shutdown后处于SHUTDOWN状态，线程池不能接受新的任务，会等待缓冲队列的任务完成。
   3. 调用shutdownNow后处于STOP状态，线程池不能接受新的任务，并尝试终止正在执行的任务。
   4. 当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。
2. Java提供的四种线程池实现：
   1. newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
   2. newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
   3. newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
   4. newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
3. 创建线程的方法:
   1. 继承Thread类
   2. 实现Runnable接口
   3. 实现Callable接口通过FutureTask包装类来创建Thread线程
   4. 使用ExecutorService管理前三种方式来实现有返回结果的多线程。
4. maven依赖jar包版本冲突的解决：
   1. 第一声明原则：

在pom.xml配置文件中，如果有两个名称相同版本不同的依赖声明，那么先写的会生效。所以，先声明自己要用的版本的jar包即可。

* 1. 路径近者优先：

直接依赖优先于传递依赖，如果传递依赖的jar包版本冲突了，那么可以自己声明一个指定版本的依赖jar，即可解决冲突。

* 1. 排出原则：

传递依赖冲突时，可以在不需要的jar的传递依赖中声明排除，从而解决冲突。

* 1. 版本锁定原则(最常使用)：

在配置文件pom.xml中先声明要使用哪个版本的相应jar包，声明后其他版本的jar包一律不依赖。解决了依赖冲突。

1. Springboot项目打成war包放到tomcat容器中进行，需要入口类继承哪个类？在哪个springboot的jar包中?
   1. 需要入口继承SpringBootServletInitializer类，并重写方法configure。
   2. 在springboot的web包中