2022/5/2

10时46分

舟谱数据：

# 开始

观察身边四五十岁的党员干部或成功人士他们就是活字典

超强的自律和自控能力

博识多学、善于思考、强于表达

说话慢条斯理，有自己的节奏和超强的目的性

有自己的御用口头禅

宠辱不惊，喜怒不形于色

特别懂尊重人，礼仪礼数特别周到

善于倾听，有一种超强的亲和力

和谁都能聊几句，并且聊得舒服有意义

知行合一，心口不一

便宜不爱占，大亏不会吃

心思缜密细腻，能够感受与感知别人的情绪变化

对人对事从不轻易明确自己的态度

总能是自己立于不败之地

断舍离三字悟的透彻，用的恰当

吃过苦，遭过罪，受过委屈，看透了人性，明白人心

不会冒犯他人，也不允许他人冒犯，有不怒自威的气场

语气里，眼神中，看不出情绪变化与内心的状态

有一定的未卜先知能力

分寸拿捏的游刃有余

豁达圆通，凡事处理的井井有条，安排的妥妥当当

有看破不说破的修养，什么人什么事心有明镜

独立，从不把自己交到别人手中，让别人主宰自己

有一眼望穿你的小心思，也一言不发的修养

有实力，有能力，没有好为人师这一习惯

识大局，懂大局，有大智大勇

身上看不到那么多儿女情长

朋友圈看不到名言警句，只有普通的柴米油盐

看不到她掉眼泪

善于拒绝别人和请人办事

城府深到一定程度可上善若水，亦或是心狠手辣，狼心狗肺

# 第一节：java基础：

* hashmap、treemap、linkedmap使用，以及特点

 Map.Entry是Map声明的一个内部接口，此接口为泛型，定义为Entry<K,V>。它表示Map中的一个实体（一个key-value对）。接口中有getKey(),getValue方法。

遍历方法：两大类：entryset（iterator while实现，直接for :（foreach遍历）实现）；元素实现（Map.keyset遍历实现,Map.values遍历实现（只有值））

//第一种：普遍使用，二次取值  
  System.out.println("通过Map.keySet遍历key和value：");  
  for (String key : map.keySet()) {  
   System.out.println("key= "+ key + " and value= " + map.get(key));  
  }  
    
  //第二种  
  System.out.println("通过Map.entrySet使用iterator遍历key和value：");  
  Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();  
  while (it.hasNext()) {  
   Map.Entry<String, String> entry = it.next();  
   System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());  
  }  
    
  //第三种：推荐，尤其是容量大时  
  System.out.println("通过Map.entrySet遍历key和value");  
  for (Map.Entry<String, String> entry : map.entrySet()) {  
   System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());  
  }

  //第四种  
  System.out.println("通过Map.values()遍历所有的value，但不能遍历key");  
  for (String v : map.values()) {  
   System.out.println("value= " + v);  
  }  
 }

1.HashMap:据键的HashCode值存储数据,访问速度快,效率最高,键可(一个)空，值可为(多)空；线程不安全

Hashtable:与hashmap相似,线程安全

2.TreeMap:可以排序,遍历出也有序,可自定义排序,实现SortMap接口,基于红黑树 （平衡二叉树）实现,键、值都不能为null，HashMap通过hashcode对其内容进行快速查找，而 TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap。HashMap通常比TreeMap效率要高一些，一个是哈希表，一个是二叉树，建议多使用HashMap，在需要排序的Map时候才用TreeMap。

<https://www.cnblogs.com/chenmo-xpw/p/4922641.html>

<https://www.cnblogs.com/LiaHon/p/11221634.html>

3.LinkedHashMap :HashMap的子类,内部维持双向链表,保留了插入的顺序new LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

第一个参数代表初始容量，第二个参数代表加载因子，第三个参数代表排序模式，true代表访问顺序，false代表插入顺序。

* hashmap删除方式（安全）遍历方法 遍历删除

可以推测，由于我们在遍历HashMap的元素过程中删除了当前所在元素，下一个待访问的元素的指针也由此丢失了（hashMap的中有一个域modCount，每次对集合进行修改（增添元素，删除元素……）时都会modCount++迭代HashMap的HashIterator中有一个变量expectedModCount，该变量会初始化和modCount相等，但如果接下来如果集合进行修改modCount改变，就会造成expectedModCount!=modCount，此时就会抛出java.util.ConcurrentModificationException异常过程类似下图（与ArrayList类似）原文链接：https://blog.csdn.net/tttzzztttzzz/article/details/87556048）。正确的方式是使用迭代器的remove()方法进行删除。

* 与hashtable的实现方法 区别

hashmap ：JDK1.8 之前 HashMap 由 数组+链表 组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。

HashMap内部维护了一个存储数据的Entry数组，HashMap采用链表解决冲突，每一个Entry本质上是一个单向链表。当准备添加一个key-value对时，首先通过hash(key)方法计算hash值，然后通过indexFor(hash,length)求该key-value对的存储位置，计算方法是先用hash&0x7FFFFFFF后，再对length取模，这就保证每一个key-value对都能存入HashMap中，当计算出的位置相同时，由于存入位置是一个链表，则把这个key-value对插入链表头。

      HashMap中key和value都允许为null。key为null的键值对永远都放在以table[0]为头结点的链表中。

<https://www.cnblogs.com/williamjie/p/9099141.html>

<https://www.jianshu.com/p/d2c14a10266e>

冲突解决：

链表法和开放地址法。链表法就是将相同hash值的对象组织成一个链表放在hash值对应的槽位；开放地址法是通过一个探测算法，当某个槽位已经被占据的情况下继续查找下一个可以使用的槽位。java.util.HashMap采用的链表法的方式，链表是单向链表。形成单链表的核心代码如下：<https://www.cnblogs.com/peizhe123/p/5790252.html>

区别：(7条)

<https://www.cnblogs.com/williamjie/p/9099141.html>

HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。

HashMap 的 key 和 value 都允许为 null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable遇到 null，直接返回 NullPointerException。

Hashtable 方法是同步，而HashMap则不是。我们可以看一下源码，Hashtable 中的几乎所有的 public 的方法都是 synchronized 的，而有些方法也是在内部通过 synchronized 代码块来实现。所以有人一般都建议如果是涉及到多线程同步时采用 HashTable，没有涉及就采用 HashMap，但是在 Collections 类中存在一个静态方法：synchronizedMap()，该方法创建了一个线程安全的 Map 对象，并把它作为一个封装的对象来返回因为hashtable,concurrenthashmap它们是用于多线程的，并发的 ，如果map.get(key)得到了null，不能判断到底是映射的value是null,还是因为没有找到对应的key而为空，而用于单线程状态的hashmap却可以用containKey（key） 去判断到底是否包含了这个null。

扩容方式的不同

Hash值不同

hashtable为什么不能null:

没有containKey方法故不能判断具体是key为空还是value为空---:hashtable为什么就不能containKey(key)

一个线程先get(key)再containKey(key)，这两个方法的中间时刻，其他线程怎么操作这个key都会可能发生，例如删掉这个key

* list对象深度拷贝方法

1.浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是**内存地址**（引用类型），拷贝的就是内存地址 ，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。

2.深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。

分为两种情况： 数组内容为person等对象类，数组内容为不可变String类。，，，插播一句平时我们用的arr数组也是地址引用，一个函数里改变了传的这个值，其也会改变。

比如List<String>这样的情况。我上面使用的是List<Person>，它和List<String>的区别就在于Person类和String类的区别，Person类提供了破坏数据的2个setter方法。因此，在浅复制的情况下，源数据被修改破坏之后，使用相同引用指向该数据的目标集合中的对应元素也就发生了相同的变化。

因此，在需求要求必须深复制的情况下，要是使用上面提到的方法，请确保List<T>中的T类对象是不易被外部修改和破坏的。

<https://www.iteye.com/blog/bijian1013-2358367>

首先已经知道浅拷贝的原因是两个开辟的空间同时指向了同一个顺序表而导致对其中一个进行操作时，另一个也会受到影响。而当把对象序列化并存储后，再将其反序列化（反序列化返回的是一个对象），这时候反序列化得到的对象的存储位置已经与原对象不同了，也就是在反序列化后产生了两个一毛一样的对象，但它们并不是同一个。

<https://www.jianshu.com/p/5a31266c6adc>

<https://blog.csdn.net/zhchyun2008/article/details/89487500>

* 用到的锁有哪些

排他锁和共享锁

synchronized就是一个排他锁，ReentrantLock也是一个排它锁，而ReentrantReadWriteLock则是一个读共享锁，写排他锁

乐观锁和悲观锁

乐观锁和悲观锁是一种锁的思想，乐观锁对并发数据修改持乐观态度，通过CAS（Compare and Swap，<https://www.jianshu.com/p/ab2c8fce878b>即比较再交换，CAS是一种无锁算法，CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值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，，是CPU指令级的操作，只有一步原子操作，所以非常快，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1，那么A→B→A就会变成1A→2B→3A。从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。加版本标志）非加锁方式尝试修改数据，而悲观锁对数据修改持保守态度，修改都要加排它锁，也常用于和数据库隔离级别结合使用

分段锁

分段锁就是将数据分段上锁，把锁进一步细粒度化，有助于提升并发效率。具体应用如下：HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

分段锁也是一种锁思想，对数据分段加锁已提高并发效率，比如jdk8之前的ConcurrentHashMap，jdk8后采用CAS+synchronized。通过hashCode计算到索引后对数据分段加锁<https://www.jianshu.com/p/e694f1e868ec>

自旋锁

通过不断的轮询来尝试获取锁，是一种占用CPU时间的非阻塞锁，当锁的等待时间短，效率会很高

公平锁和非公平锁

同步锁按照线程申请锁的顺序，非同步则不保证。synchronized和ReentrantLock都是非同步锁。ReentrantLock可在构造方法里指定为同步锁

可重入锁

又叫递归锁，具体是当外部申请锁的操作获得了锁，内部申请锁的操作也会获得锁，有点类似与进程中破坏请求保持原则来预防死锁一样，要么全部申请成功，要么全部失败

锁消除

JVM会加锁的代码进行逃逸分析，当发现是单线程时，会去掉代码所加的锁，以达到优化。

synchronized是JVM层次实现的，在高并发的情况下性能不如代码层次实现的Lock高效，但是synchronized一直在被优化，现在差距已经不大了，是官方推荐的方式。

————————————————

线程池中的四个有什么区别cachedthread

一.FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。有核心的线程

二．CachedThreadPool的特点就是在线程池空闲时，即线程池中没有可运行任务时，它会释放工作线程，从而释放工作线程所占用的资源。但是，但当出现新任务时，又要创建一新的工作线程，又要一定的系统开销。并且，在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。只有非核心的线程

SignalThreadPool 单例线程池 一个人慢慢的干活 只有一个核心线程，就是一个孤家寡人，通过指定的顺序将任务一个个丢到线程，都乖乖的排队等待执行，不处理并发的操作，不会被回收。

ScheduledThreadPoll

这个线程池就厉害了，是唯一一个有延迟执行和周期重复执行的线程池。它的核心线程池固定，非核心线程的数量没有限制，但是闲置时会立即会被回收

Future用来保存Callable异步运算的结果

FutureTask封装Future的实体类

* Threadlocal作用

ThreadLocal是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只有在指定的线程中可以获取到存储的数据，对于其他线程来说则无法取到数据，，线程为作用域并且不同线程具有不同的数据的副本时，就可以考虑用ThreadLocal；

* 原子性实现

Java中的原子操作zhidao包括：

1）除long和double之外的基本类型的赋值操作

2）所有引用reference的赋值操作

3）java.concurrent.Atomic.\* 包中所有类的一切操作

CAS机制实现

锁机制实现

* Concurrent包用过哪些

Atomic love、lock

* Jvm中参数设置，理解有多少

新生代分为三个区域，一个Eden（伊甸园）区和两个Survivor（幸存）区，它们之间的比例为（8：1：1），这个比例也是可以修改的。通常情况下，对象主要分配在新生代的Eden区上，少数情况下也可能会直接分配在老年代中。Java虚拟机每次使用新生代中的Eden和其中一块Survivor（From），在经过一次Minor GC后，将Eden和Survivor中还存活的对象一次性地复制到另一块Survivor空间上（这里使用的复制算法进行GC），最后清理掉Eden和刚才用过的Survivor（From）空间。将此时在Survivor空间存活下来的对象的年龄设置为1，以后这些对象每在Survivor区熬过一次GC，它们的年龄就加1，当对象年龄达到某个年龄（默认值为15）时，就会把它们移到老年代中。在新生代中进行GC时，有可能遇到另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代；

年老代里存放的都是存活时间较久的，大小较大的对象，因此年老代使用标记整理算法。当年老代容量满的时候，会触发一次Major GC（full GC），回收年老代和年轻代中不再被使用的对象资源

永久代

指内存的永久保存区域，主要存放Class和Meta（元数据）的信息。

Class在被加载的时候被放入永久区域。它和和存放实例的区域不同，GC不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常。

在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。

元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入java堆中. 这样可以加载多少类的元数据就不再由MaxPermSize控制, 而由系统的实际可用空间来控制。

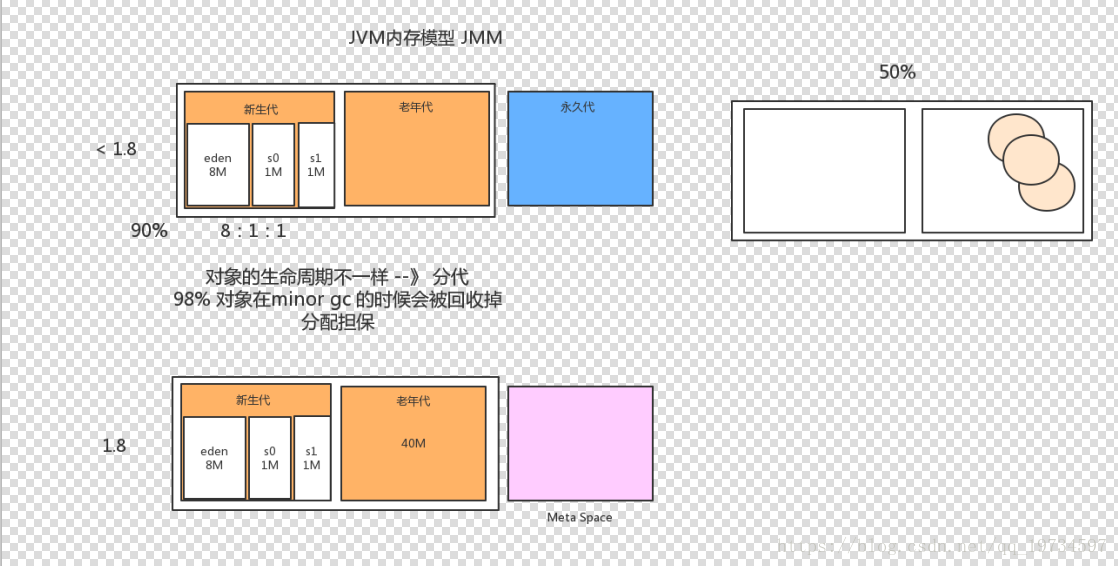
永久代的垃圾收集主要回收两部分内容（一般不适用，性价比低，）：废弃常量和无用的类。判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”：

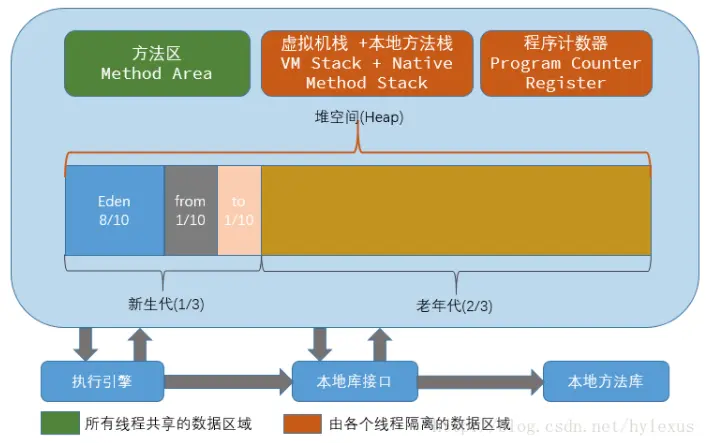
1该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。

2加载该类的ClassLoader已经被回收。

3该类对应的java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

　元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存





在 Jdk6 以及以前的版本中，字符串的常量池是放在堆的Perm区的，Perm区是一个类静态的区域，主要存储一些加载类的信息，常量池，方法片段等内容，默认大小只有4m，一旦常量池中大量使用 intern 是会直接产生java.lang.OutOfMemoryError:PermGen space错误的。

在 jdk7 的版本中，字符串常量池已经从Perm区移到正常的Java Heap区域了。为什么要移动，Perm 区域太小是一个主要原因，

Jdk8已经直接取消了Perm区域，而新建立了一个元区域。应该是jdk开发者认为Perm区域已经不适合现在 JAVA 的发展了。

作者：小小少年Boy

链接：https://www.jianshu.com/p/c1ac5e7a5f87

来源：简书

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

* **Major GC和Full GC区别**
* Full GC：收集young gen、old gen、perm gen
* Major GC：有时又叫old gc，只收集old gen
* **Full GC触发机制：**  
  　　（1）调用System.gc时，系统建议执行Full GC，但是不必然执行  
  　　（2）老年代空间不足  
  　　（3）方法区空间不足  
  　　（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存  
  　　（5）由Eden区、survivor space1（From Space）区向survivor space2（To Space）区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小  
  　　　　当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载。

Xmx xms 堆最大最小

Xss栈值

-Xmn、-XX:NewRatio、-XX:SurvivorRatio：

-Xmn设置新生代大小

-XX:NewRatio新生代（eden+2\*s）和老年代（不包含永久区）的比值

例如：4，表示新生代:老年代=1:4，即新生代占整个堆的1/5

-XX:SurvivorRatio（幸存代）设置两个Survivor区和eden的比值

例如：8，表示两个Survivor:eden=2:8，即一个Survivor占年轻代的1/10

<https://blog.csdn.net/qq_19734597/article/details/80958817>

<https://www.cnblogs.com/pony1223/p/8661219.html>

1栈空间 栈空间是每个线程私有的区域。栈里面的主要内容是栈帧，而栈帧存放的是局部变量表，局部变量表的内容是：局部变量、参数。决定了函数调用的深度，每个线程都有独立的栈空间，局部变量、参数 分配在栈上。

2堆空间 新生代 老年代 元数据组（<1.7永久代） -Xmx（最大堆空间）和 –Xms（最小堆空间）应该保持一个什么关系，可以让系统的性能尽可能的好呢？

根据实际事情调整新生代和幸存代的大小

官方推荐新生代占堆的3/8

幸存代占新生代的1/10

在OOM时，记得Dump出堆，确保可以排查现场问题

* JDK8中map流
* 设计模式23中设计模式

对比：<https://www.cnblogs.com/pony1223/p/7608955.html>

设计模式有哪些：

<https://blog.csdn.net/perfectmatch_G/article/details/96429773>

单例模式（创建）：懒汉式-用到对象才对其实例化，线程不安全，可以使用synchronized关键字解决；饿汉式-构造时即产生相应实例，线程安全<https://blog.csdn.net/yubujian_l/article/details/81455524>

工厂设计模式（创建）：普通工厂设计模式-定义一个工厂类返回相关具体类（根据字符串）方法，具体类实现了同一个接口，工厂方法设计模式-多工厂，不再根据字符串返回对象，而是实现了具体点的工厂，根据不同工厂区分不同。

模板设计模式（行为）：简单来说是有抽象类定义好一个骨架，里面有一些子类必须实现的方法，子类来实现具体的方法，模板设计模式是一种行为设计模式，一般是准备一个抽象类，将部分逻辑以具体方法或者具体的构造函数实现，然后声明一些抽象方法，这样可以强制子类实现剩余的逻辑。不同的子类以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板设计模式能达成的功能。<https://www.cnblogs.com/yefengyu/p/10520531.html>

代理设计模式（结构）：为某个对象提供一个代理，以控制对这个对象的访问。 代理类和委托类有**共同的**父类或父接口，这样在任何使用委托类对象的地方都可以用代理对象替代。代理类负责请求的预处理、过滤、将请求分派给委托类处理、以及委托类执行完请求后的后续处理；分为静态代理与动态代理，静态的为两个类区实现一个接口，其中一个是代理类（辅助业务）一个是被代理类（具体业务），代理类中方法可以操作被代理实例，做一些环绕动作，如可以利用代理对象为被代理对象添加额外的功能，以此来拓展被代理对象的功能。可以用于计算某个方法执行时间，在某个方法执行前后记录日志等操作；动态代理模式：Java通过java.lang.reflect包中的三个类Proxy、InvocationHandler、Method来支持动态代理。动态代理常用于有若干个被代理的对象，且为每个被代理对象添加的功能是相同的（例如在每个方法运行前后记录日志）。动态代理的代理类不需要我们编写，由Java自动产生代理类源代码并进行编译最后生成代理对象。

创建动态代理对象的步骤：

1. 指明一系列的接口来创建一个代理对象

2. 创建一个调用处理器（InvocationHandler）对象

3. 将这个代理指定为某个其他对象的代理对象

4. 在调用处理器的invoke（）方法中采取代理，一方面将调用传递给真实对象，另一方面执行各种需要的操作

<https://blog.csdn.net/yubujian_l/article/details/81455524>

适配器模式：简单来说解释适配，C想掉B中的方法，但是没有相关的接口（重复写或者不容许写），但是可以调A中的方法，所以写一个A的适配器可以适配B，由此C就可以掉B了。<https://www.cnblogs.com/qianguyihao/p/3930271.html>

重点说下模板设计模式实现

* 设计原则：六大设计原则

<https://www.cnblogs.com/pony1223/p/7594803.html>

* 职责单一：一个类只负责一项职责
* 里式替换：子类可以扩展父类的功能，但尽量不能改变父类原有的功能
* 依赖倒置：对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合，核心思想为面向接口编程。
* 接口隔离：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。
* 迪米特（最少知道）：一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息
* 开闭原则：开闭原则就是说对扩展开放，对修改关闭
* **用抽象构建框架，用实现扩展细节的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。**
* Object中方法说明：

1 registerNatives() //私有方法

2 getClass() //返回此 Object 的运行类。

3 hashCode() //用于获取对象的哈希值。

4 equals(Object obj) //用于确认两个对象是否“相同”。

5 clone() //创建并返回此对象的一个副本。

6 toString() //返回该对象的字符串表示。

7 notify() //唤醒在此对象监视器上等待的单个线程。

8 notifyAll() //唤醒在此对象监视器上等待的所有线程。

以下三个方法都必须在synchronized 同步关键字所限定的作用域中调用 其原理为ObjectMonitor再起作用，wait()方法会释放所占有的ObjectMonitor对象，而notify()和notifyAll()并不会释放所占有的ObjectMonitor对象，它们的主要工作是将相应的线程从\_WaitSet转移到\_EntryList中，然后等待竞争获取锁。其实真正释放ObjectMonitor对象的时间点是在执行monitorexit指令，一旦释放ObjectMonitor对象后，\_EntryList中ObjectWaiter节点所保存的线程就可以竞争ObjectMonitor对象进行加锁操作了。ObjectMonitor

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

10 wait(long timeout, int nanos) //在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

11 wait() //用于让当前线程失去操作权限，当前线程进入等待序列

12 finalize() //当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

List转set 数组

<https://www.jianshu.com/p/717bc27141c4?tdsourcetag=s_pctim_aiomsg>

Arrays.asList方法返回的是数组的一个视图

视图意味着，对这个list的操作都会反映在原数组上，而且这个list是定长的，不支持add、remove等改变长度的方法。

# 第二节：应用

#### Spring

AOP，日志原理，切入办法

注解

接口实现，Springboot如何在spring上创建，

Bean生命周期，初始化加载的实现有哪些

Spring事务 隔离

微服务之间熔断机制，失联策略

#### Redis

多个微服务订单个数唯一性如何实现

#### Mysql

用到过那些调优

大量的数据如何存储的

1. 日志区别

Logstash log4j 等等

#### Kafka

如何保障有效性，时序性

Redis击穿处理

数据库防注入。。。

## MyBatis 面试题

* 1、什么是 Mybatis？
* 2、Mybaits 的优点：
* 3、MyBatis 框架的缺点：
* 4、MyBatis 框架适用场合：
* 5、MyBatis 与 Hibernate 有哪些不同？
* 6、#{}和${}的区别是什么？
* 7、当实体类中的属性名和表中的字段名不一样 ，怎么办 ？
* 8、 模糊查询 like 语句该怎么写?

**ZooKeeper 面试题**

ZooKeeper 是一个开源的分布式协调服务，由雅虎创建，是 Google Chubby 的开源实现。分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、配置维护，名字服务、分布式同步、分布式锁和分布式队列等功能。

* zookeeper 负载均衡和 nginx 负载均衡区别
* Zookeeper Watcher 机制--数据变更通知
* 客户端注册 Watcher 实现
* zookeeper 是如何保证事务的顺序一致性的？
* Zookeeper 对节点的 watch监听通知是永久的吗？为什么不是永久的?
* zk 节点宕机如何处理？

**Dubbo 面试题**

Spring Eureka 从开源转变为闭源，Consul 正在崛起，而 Dubbo 又开始重新更新。目前市场上仍有不少公司使用dubbo我们也需要继续学习。

* 为什么要用 Dubbo？
* 默认使用的是什么通信框架，还有别的选择吗?
* 服务调用是阻塞的吗？
* 一般使用什么注册中心？还有别的选择吗？
* 默认使用什么序列化框架，你知道的还有哪些？
* Dubbo 的整体架构设计有哪些分层?

**Elasticsearch 面试题**

* elasticsearch 了解多少，说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段 。
* elasticsearch 索引数据多了怎么办，如何调优，部署
* elasticsearch 是如何实现 master 选举的
* Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法
* 详细描述一下 Elasticsearch 更新和删除文档的过程。
* Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法？

(文末附面试答案)

**Redis面试题**

随着应用研发技术的不断成熟，Redis 缓存技术已经成为后台研发同学必备的能力之一。在很多公司面试的过程中，都必不可少地考察 Redis 知识点的掌握。

* 使用 Redis 有哪些好处？
* Redis 是单进程单线程的？
* Redis 的持久化机制是什么？各自的优缺点？
* Redis 常见异常问题和解决方案（雪崩，穿透...）
* redis 过期键的删除策略？
* 为什么 redis 需要把所有数据放到内存中？
* Redis 集群的主从复制模型是怎样的？
* Redis key 的过期时间和永久有效分别怎么设置？

**MySQL 面试题**

如今互联网行业用的最多就是 MySQL，然而对于高级 Web 面试者，尤其对于寻找 30k 下工作的求职者，很多 MySQL 相关知识点基本都会涉及，如果面试中，你的相关知识答的模糊和不切要点，基本大多稍好公司的面试官，基本也不会对你有兴趣，毕竟稍微量大的项目，最终都会让你去深入数据库，而且最终的大多瓶颈首先也会出现在数据库。

* MySQL 相关存储引擎。
* MySQL 索引实现原理。
* MySQL 数据库锁。
* MySQL 的 SQL 调优。
* MySQL 分表分库全局 ID。

**Java 并发编程面试题**

* 在 java 中守护线程和本地线程区别？
* 什么是多线程中的上下文切换？
* Java 中用到的线程调度算法是什么？
* 什么是线程组，为什么在 Java 中不推荐使用？
* 在 Java 中 Executor 和 Executors 的区别？
* 并发编程三要素？
* 什么是线程池？有哪几种创建方式？

**Spring 面试题**

* 1. 谈谈对 Spring IoC 的理解？
* 2. 谈谈对 Spring DI 的理解？
* 3. BeanFactory 接口和 ApplicationContext 接口不同点是什么？
* 4. 请介绍你熟悉的 Spring 核心类，并说明有什么作用？
* 5. 介绍一下 Spring 的事务的了解？
* 6. 介绍一下 Spring 的事务实现方式？
* 7.什么是 Spring 的依赖注入？

(文末附面试答案)

**JVM面试题**

* JVM三大性能调优参数，JVM 几个重要的参数
* JVM调优
* JVM内存管理，JVM的常见的垃圾收集器，Ｇ１垃圾收集器。GC调优，Minor GC ，Full GC 触发条件
* Java内存模型
* Java垃圾回收机制
* 描述JVM中一次Full gc过程
* Java 中都有哪些引用类型？
* JVM中类加载机制，类加载过程，什么是双亲委派模型？，类加载器有哪些  
  如何判断是否有内存泄露？定位 Full GC 发生的原因，有哪些方式？

**Spring Cloud 面试题**

* 什么是Spring Cloud？
* 使用Spring Cloud有什么优势？
* 服务注册和发现是什么意思？Spring Cloud如何实现？
* 负载平衡的意义什么？
* 什么是Hystrix？它如何实现容错？
* 什么是Hystrix断路器？我们需要它吗？
* 什么是Netflix Feign？它的优点是什么？
* 什么是Spring Cloud Bus？我们需要它吗？

**Kafka面试题**

* Kafka 与传统 MQ 消息系统之间有三个关键区别？
* 讲一讲 kafka 的 ack 的三种机制？
* 消费者故障，出现活锁问题如何解决？
* kafka 分布式（不是单机）的情况下，如何保证消息的顺序消费？
* kafka 如何不消费重复数据？比如扣款，我们不能重复的扣。

多线程开发过程中遇到的问题

讲一讲索引

<https://www.jianshu.com/p/d90f6b028d0e>

数据库中的挑战

Kafka有别于其他MQ

项目中遇到的问题

百度（3轮技术面）：

**1.synchrnoized和reentrantlock的底层实现及重入的底层原理**

**monitorenter monitorexit AQS。 重入原理：entry count，state**

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

**Java 中的并发锁大致分为隐式锁和显式锁两种。隐式锁就是我们最常使用的 synchronized 关键字，显式锁主要包含两个接口：Lock 和 ReadWriteLock，主要实现类分别为ReentrantLock 和ReentrantReadWriteLock，这两个类都是基于AQS(AbstractQueuedSynchronizer) 实现的。还有的地方将 CAS 也称为一种锁，在包括 AQS 在内的很多并发相关类中，CAS 都扮演了很重要的角色。**

**原文链接：https://blog.csdn.net/ly199108171231/java/article/details/88098614**

**2.锁的四种状态及升级过程**

**3.CMS和G1的异同**

**4.G1什么时候引发Full GC**

**5.除了CAS，原子类，syn，Lock还有什么线程安全的方式**

**final**

**6.HashMap和Hashtable的异同**

**7.允许null键的map你知道哪些**

**HashMap，LinkedHashMap，WeakHashMap**

**8.null键放在HashMap的哪里**

**底层数组的0号位置**

**9.为什么hashtable的扩容是2倍+1**

**从除留余数法，hashtable初始容量方面回答**

**10.红黑树的插入时间复杂度**

**11.解决哈希冲突的方式**

**12.现有1T的数据，内存只有1G，该怎么对他们排序**

**外部排序（多路归并）**

**13.Tomcat为什么要重写类加载器**

**14.tcp握手挥手过程及其状态转换**

**15.mysql隔离级别**

**16.B树和B+树的区别**

**17.你知道哪些设计模式，它们在JDK源码中是怎么体现的**

**18.Java运行时数据区**

**19.说一个最熟悉的垃圾回收算法**

**20.吞吐量优先和响应时间优先的回收器是哪些**

**parallel scavenge和CMS**

**21.类加载全过程**

**22.线程池7个参数**

**23.CAS的ABA问题怎么解决**

**AtomicStampedReference，还可用带boolean版本戳的AtomicMarkableReference**

**24.Java内存模型**

**JMM共享内存模型以及8个原子操作指令**

**25.什么叫做阻塞队列的有界和无界**

**26.cookie和session介绍一下**

**27.说一下反射，反射会影响性能吗**

**28.谈一下AQS吧**

**29.为什么你说AQS的底层是CAS+volatile**

**state是volatile修饰的，并且设置state的方法除了有setState，还有compareAndSetState**

**30.JUC包里的同步组件主要实现了AQS的哪些主要方法**

**tryAcquire, tryRelease, tryAcquireShared, tryReleaseShared, isHeldExclusively**

**coding：leetcode+剑指offer原题共5题**

**美团（3轮技术面）：**

**1.ConcurrentHashMap底层原理**

**2.手写一个LRU**

**用LinkedHashMap**

**3.HashMap底层数据结构**

**4.为什么用红黑树不用普通的AVL树**

**5.为什么在8的时候链表变成树**

**6.为什么在6的时候从树退回链表**

**7.线程池7个参数，该怎么配置最好**

**8.说一下volatile**

**9.volatile的可见性和禁止指令重排序怎么实现的**

**可见性：缓存一致性协议**

**禁止指令重排序：JMM模型里有8个指令完成数据的读写，通过其中load和store指令相互组合成的4个内存屏障实现禁止指令重排序**

**10.CAS是什么**

**11.PriorityQueue底层是什么，初始容量是多少，扩容方式呢**

**最小堆，**

**11，若原始大小<64，则扩容为原来的2倍+2，不然就扩容为原来的1.5倍**

**12.HashMap的容量为什么要设置为2的次幂**

**13.你知道跳表吗，什么场景会用到**

**ConcurrentSkipListMap，用在多线程下需要自定义排序顺序时**

**14.CopyOnWriteArrayList知道吗，迭代器支持fail-fast吗**

**线程安全ArrayList，写时复制，迭代器是采用快照风格，不支持fail-fast**

**15.innodb的底层数据结构**

**16.为什么用B+树不用B树**

**17.为什么用B+树不用红黑树**

**18.coding：无序数组怎么寻找第k大的数，写一个二叉树层次遍历**

**19.不知道大小的数据流取其中100个数，怎样的取法能最随机**

**20.n个物品每个物品都有一定价值，分给2个人，怎么分两个人的价值差最小**

**21.假设百度每个页面能放100个网页，每个页面都有一个评分，怎样快速找到第8页的所有网页**

**顺丰（2轮技术面）：**

**1.线程池的设计里体现了什么设计模式**

**2.说说你了解什么设计模式，知道责任链设计模式吗**

**3.wait/notify体现了什么设计模式**

**4.线程池7个参数**

**5.谈一下spring事务传播**

**6.谈一下IOC底层原理**

**7.怎么判断内存泄漏**

**8.怎么在日志里排查错误，该用哪些Linux命令**

**9.mysql原子性和持久性怎么保证**

**undolog，redolog**

**10.怎么解决幻读**

**MVCC+间隙锁**

**11.innodb和myisam区别**

**12.索引分类**

**13.对象的创建过程**

**14.对象在内存中的存储布局**

**对象头，类元指针，实例数据，对齐填充**

**15.对象头具体包括什么**

**16.对象怎么定位**

**直接指针，句柄**

**17.堆的划分**

**18.对象怎么分配**

**栈上分配->TLAB->老年代->新生代**

**19.具体讲一下CMS流程**

**20.锁的四种状态**

**21.Object obj = new Object()这句话在内存里占用了多少内存**

**引用变量4Byte，Object对象的对象头12Byte，由于对齐填充增值16Byte，共20Byte**

**22.synchronized和reentrantLock的异同**

**京东（2轮技术面）：**

**1.总体说一下集合框架**

**2.你怎么看待接口和抽象类**

**3.索引的分类**

**4.主键索引的设计应该采用B-tree索引还是hash索引**

**5.设计模式说5，6个**

**6.谈一谈DDD面向领域编程**

**7.说一下hibernate一级缓存和二级缓存**

**8.说一下你了解的MQ**

**9.谈一谈你对高并发的理解，你会从什么角度设计高并发程序**

**10.JUC包里的限流该怎么做到**

**Semaphore / guava ratelimiter**

**11.索引不适用的条件**

**12.说一下NIO和AIO**

**13.AIO里用到什么设计模式**

**观察者**

**14.说一下select，poll，epoll**

**15.谈一下TCP的拥塞控制**

**16.你知道什么是as-if-serial语义吗，它和happen-before语义有什么区别**

**本质没啥区别，前者是谈的单线程程序，后者谈的多线程程序**

**17.Executors创建线程池的方式**

**18.CachedThreadPool里面用的什么阻塞队列**

**SynchronousQueue**

**19.那你知道LinkedTransferQueue吗，和SynchronousQueue有什么区别**

**20.你还知道什么阻塞队列，能具体说说它们的特点吗**

**ArrayBlockingQueue，LinkedBlockingQueue，LinkedBlockingDeque，PriorityBlockingQueue，DelayQueue**

**21.线程池的线程数怎么设置比较好**

**22.你知道新出的LongAdder吗，和AtomicLong有什么区别**

**AtomicLong相当于多线程竞争一次修改value的机会**

**LongAdder把value拆成多个值放在cell数组里，相当于多线程竞争多次修改value的机会，性能自然上升**

**23.那你知道LongAccumulator吗**

**不仅支持加法，还支持自定义的其他操作**

**阿里（7轮技术面）：**

**阿里淘系 七面 工程项目经验基本为0 所以被死磕Java**

**总结起来：面试官喜欢抛出一个大点，看你能讲多广多深**

**一面: 90mins**

**讲项目**

**讲一个集合框架整体框架**

**分Collection和Map两大类全部讲一遍，每一个接口和对应实现类，他们类各自的特点，性质，基本参数，能讲多深讲多深**

**arraylist和hashset区别**

**为什么hashmap要在8的时候树华**

**根据泊松分布概率（源码注释里写了）**

**hashmap线程安全的方式**

**hashtable和concurrenthashmap的各自特点，能讲多深讲多深**

**为什么hashtable被弃用了，cch1.7之前和1.8之后的区别**

**锁的分类**

**乐观锁、悲观锁、自旋锁、读写锁、排它锁、共享锁、分段锁等等各自特点，以及它们在java中具体的实现方式**

**Spring IOC的底层实现**

**XML+dom4j+工厂+单例**

**索引不适用的条件**

**索引列上有函数，不满足最左前缀，使用了不等号，使用了范围查询等等**

**索引的分类**

**B-Tree索引，Hash索引，全文索引，单值索引、唯一索引、复合索引、聚簇索引、非聚簇索引等等，以及它们各自的特点**

**二面：40mins**

**讲项目**

**线程池了解多少**

**executors创建的几种线程池，直接new ThreadPoolExecutor，7个参数**

**线程池拒绝策略分别使用在什么场景**

**Spring AOP的底层实现**

**动态代理，newProxyInstance，cglib，ASM**

**讲一下代理模式**

**动态代理，静态代理**

**你都了解什么设计模式，他们在JDK中如何体现的**

**工厂，责任链，观察者，建造，代理，单例，原型等等在JDK中对应的体现。。。**

**三面coding：2h**

**千万级数据量的list找一个数据（多线程），抢红包**

**四面：50mins**

**讲项目**

**讲一下JVM堆内存管理**

**栈上分配->TLAB->新声代、老年代->可达性分析->GC算法->所有垃圾回收器及其优缺点和特点**

**那到底多大的对象会被直接扔到老年代**

**G1两个region不是连续的，而且之间还有可达的引用，我现在要回收其中一个，另一个会被怎么处理**

**听说过CMS的并发预处理和并发可中断预处理吗**

**平常怎么学习的和职业规划**

**五面coding：2h**

**百万级int数据量的一个array求和（fork/join）**

**六面交叉面：20mins**

**深挖科研项目**

**七面hr：20min**

**聊人生**

**阿里面**

1、请描述synchrnoized和reentrantlock的底层实现及重入的底层原理

2、请描述锁的四种状态和升级过程

3、CAS的ABA问题如何解决

4、请谈一下AQS，为什么AQS的底层是CAS + volatile

5、请谈一下你对volatile的理解

6、volatile的可见性和禁止指令重排序是如何实现的

7、CAS是什么

8、请描述一下对象的创建过程

9、对象在内存中的内存布局

10、DCL单例为什么要加volatile

11、解释一下锁的四种状态

12、Object 0 = new Object()在内存中占了多少字节?

13、请描述synchronized和ReentrantLock的异同

14、聊聊你对as-if-serial和happens-before语义的理解

15、你了解ThreadLocal吗?你知道ThreadLocal中 如何解决内存泄漏问题吗?

16、请描述一下锁的分类以及JDK中的应用

17、自旋锁一定比重量级锁效率高吗?

a/b

i++线程安全吗，为什么不安全

volitate

线程执行一部分再唤起另一半的方法

Spring 拦截器

项目中Pullpush

# 第三节：实战

### 博客搜集

进程与线程的区别与联系？

进程就是程序的一次动态执行过程，通俗来讲，进程就是正在运行的程序，它是系统进行资源分配和调用的独立单位。每一个进程都有它自己的内存空间和系统资源。但进程的开启是非常耗费时间的，所以有必要对其进行进一步的划分以提高性能。一个进程可以同时有多个线程，相当于一个程序中同时进行多个任务，多个线程共享同一个进程的资源(堆内存和方法区)。所有的线程一定要依附于进程才能够存在，一旦进程消失，线程一定也会消失。多线程的作用不是提高执行速度，而是为了提高应用程序的使用率。线程是处理器调度和分派的基本单位，而且多线程具有随机性，抢占CPU执行权的概率完全是随机的。

线程的操作状态

任何线程一般都具有五种状态，即：创建，就绪，运行，堵塞与终止。

创建

在程序中用构造方法创建一个线程对象后，新的线程就处于新建状态。此时的线程就已经拥有内存空间和其他资源了，但还处于不可运行状态。

就绪

新建线程对象后，调用该线程的\*\*start()\*\*方法就可以启动线程。当线程启动时，线程就进入了就绪状态。此时，线程将进入线程队列排队，等待CPU服务，这表明线程已经具备了运行条件。

运行

当就绪状态的线程被调用并且获得处理器资源时，线程就已经进入运行状态。此时，会自动调用线程的\*\*run()\*\*方法。run()方法中定义了线程的操作与功能。

堵塞

一个正在运行状态的线程在某些特殊情况下，比如被人为挂起或需要执行耗时的输入输出操作时，将让出CPU，并暂时中止自己的执行，进入堵塞状态。在可执行的状态下，如果调用sleep()，suspend()，\*\*wait()\*\*等方法，线程都将进入阻塞状态。堵塞时，线程不能进入排队队列，只有当引起阻塞的院系被消除后，线程将重新进入就绪状态。

终止

线程调用stop方法时或者run方法结束后，就处于终止状态。处于终止状态的线程不具有继续运行的能力。

并行与并发的区别？

并发是逻辑上同时发生，指在某一个时间内同时运行多个程序。

并行是物理上同时发生，指在某一个时间点同时运行多个程序。

当有多个线程在操作时,如果系统只有一个CPU,则它根本不可能真正同时进行一个以上的线程，它只能把CPU运行时间划分成若干个时间段,再将时间 段分配给各个线程执行，在一个时间段的线程代码运行时，其它线程处于挂起状。这种方式我们称之为并发(Concurrent)。

当系统有一个以上CPU时,则线程的操作有可能非并发。当一个CPU执行一个线程时，另一个CPU可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。

Java虚拟机是多线程的吗？

java命令会启动java虚拟机，相当于启动了一个应用程序，相当于启动了一个进程。虚拟机会开启一个主线程去寻找main方法，所以说main方法是运行在主线程中的。但是虚拟机在工作时还会启动垃圾回收机制，也就相当于开启了另一个线程。所以说，我们的JVM虚拟机是多线程的。

多线程的实现方式有哪几种？如何选择？

在Java中，要想实现多线程，就必须依靠一个线程的主类，在主类中重写run()方法作为线程的主体。不管是以三种方式中的哪一个实现多线程，都是为了定义这个主类。

三种实现方式的选择：尽量避免继承Thread类，优先考虑实现接口（Runnable或Callable）的方法。因为Java采用的是单继承的模式，继承Thread类就会带来这种局限性，没法再继承其他类；另外，实现接口可以更方便的实现数据共享的概念。

申明一点，多线程启动的唯一方法就是Thread类中的start()方法。

1. 继承Thread类

线程主体类：

public class MyThread extends Thread {

// 重写run方法，作为线程的主操作方法

@Override

public void run() {

...

}

}

开启多线程：

MyThread threadA = new MyThread("ThreadA");

MyThread threadB = new MyThread("ThreadB");

threadA.start();

threadB.start();

2. 实现Runnable接口

Thread类也是Runnable类的接口。使用Runnable接口实现多线程：

避免了单继承带来的局限性；

可以更好的实现数据共享。

主体类

public class MyThread implements Runnable {

// 重写run方法，作为线程的主操作方法

@Override

public void run() {

...

}

}

开启多线程：要用到Thread类的有参构造方法（public Thread(Runnable target)）

MyThread mt1 = new MyThread();

new Thread(mt1).start(); // 多线程调用同一个Runnable对象，就可以实现数据共享

new Thread(mt1).start();

3. 实现Callable接口

使用Runnable接口实现的多线程可以避免单继承的局限，但是Runnable接口存在一个问题就是没有办法返回run方法的操作结果（public void run()）。为了解决这个问题，从JDK1.5开始，引入了这个接口java.util.concurrent.Callable:

@FunctionalInterface

public interface Callable<V> { V call() throws Exception; }

这个接口中只定义了一个\*\*call()\*\*方法，而且在call()方法上可以实现线程操作数据的返回，返回类型由Callable接口上的泛型决定。但是注意，Callable接口并非Runnable接口的子类，意味着不能直接传入Thread构造器并开启线程。

为了开启线程并获取这个返回值，靠Thread类是不可以的。为了解决这个问题，从JDK1.5起，引入了java.util.concurrent.FutureTask<V>类，定义如下：

public class FutureTask<V> extends Object implements RunnableFuture<V>{...}

1

FutureTask类提供了Callable接口子类的构造方法，并定义了专门的方法来获取多线程中的返回值：public FutureTask(Callable<V> callable)，public V get()。

同时，FutureTask是Runnable接口子类，可以使用public Thread(Runnable target)构造。所以开启Callable对象线程的步骤一般是 callable - FutureTask(callable) - Thread(futuretask).start()。

主体类

import java.util.concurrent.Callable;

public class MyThread implements Callable<String> {

private int ticket = 10;

@Override

public String call() throws Exception {

for (int i = 0; i < 100; i++) {

if (this.ticket > 0)

System.out.println("ticket=" + this.ticket--);

}

return "售完";

}

}

开启多线程并获取返回值：

// 实例化多线程对象

MyThread myThread1 = new MyThread();

MyThread myThread2 = new MyThread();

// 使用public FutureTask(Callable<V> callable)实例化FutureTask

FutureTask<String> task1 = new FutureTask(myThread1);

FutureTask<String> task2 = new FutureTask(myThread2);

// FutureTask是Runnable接口子类，可以使用public Thread(Runnable target)构造

new Thread(task1).start();

new Thread(task2).start();

// 调用获取返回值

String msg1 = task1.get();

String msg2 = task2.get();

System.out.println("线程1返回的结果是：" + msg1 + "\t线程2返回的结果是：" + msg2);

启动一个线程是调用 run()方法还是 start()方法？

多线程启动的唯一方法就是Thread类中的start()方法。start()方法里面会调用一个start0()的方法，而且这个方法是用native声明的。java中调用本机操作系统提供的函数的技术叫做JNI（Java Native Interface ），这个技术离不开特定的操作系统，因为多线程必须由操作系统来分配资源。这项操作是根据JVM负责根据不同的操作系统实现的。start()方法使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，所以线程并不会会立即运行。而run()方法是线程启动后要进行回调（callback）的方法。

线程类Thread的常用方法？

获取和设置线程名称：public final String getName()，public final void setName(String name)

获取当前执行的线程：public static Thread currentThread()；

线程的优先级设置：public final void setPriority(int newPriority)；（1-10，默认是5）优先级只能说明抢占CUP执行权的概率大一些，并不能保证一定优先执行。

线程休眠：public static void sleep(long millis) ；单位是毫秒。

线程加入：public final void join()； 该线程执行完毕再执行其他线程。

线程礼让：public static void yield() ；暂停当前正在执行的线程对象（时间相当短），并执行其他线程。

线程守护：public final void setDaemon(boolean on) ；传入true设置为守护线程，必须在启动线程前调用，主线程死亡后，守护线程均死亡。

中断线程：public void interrupt() ；当线程调用wait()，sleep()方法的时候处于阻塞状态，可以通过这个方法清除阻塞状态。并不能中断正在运行的线程。

同步线程及线程调度相关的方法？

wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；

notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

JDK5 通过 Lock 接口提供了显示的锁机制， Lock 接口中定义了加锁（lock()方法）和解锁（unLock()方法），增强了多线程编程的灵活性及对线程的协调。

wait() 和 sleep() 方法的不同？

wait()是Object类的方法，而sleep()是Thread类的方法；

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法(锁代码块和方法锁)；

wait，notify（唤醒单个线程）和notifyAll（唤醒所有线程）（都是Object类的方法）只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用(使用范围)；

sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常；

wait 通常被用于线程间交互， sleep 通常被用于暂停执行。

定时器Timer和TimerTask?

定时器是一个应用十分广泛的线程工具，可用于调度多个定时任务以后台线程的方式执行。定时器在实际开发中应用场景不多，一般由第三方框架实现。在Java中，可以通过Timer + TimerTask 来实现定时器功能。

Timer用于在后台线程中计划执行任务，可安排任务执行一次，或者定期重复执行。

TimerTask是一个抽象类，实现了Runnable接口。它的子类代表一个可以被Timer计划的任务，具体的任务在TimerTask中run方法中实现（TimerTask相当于专门用来制定定时任务的Runnable对象）。

定时器的启动必须通过Timer的schedule()方法（相当于start方法开启线程）：

schedule(TimerTask task, long delay); 指定任务执行的延迟时间

schedule(TimerTask task,long delay,long period); 指定任务执行的延迟时间和周期

schedule(TimerTask task, Date time); 指定任务执行的准确时间

schedule(TimerTask task, Date firstTime, long period);指定任务执行的准确时间和周期

线程的互斥与同步？

由于多线程执行的异步性，会给系统造成混乱，比如当多个线程急用共享变量，表格，链表时，可能会导致数据处理出错。因此线程同步的主要任务是使并发执行的各线程之间能够有效的共享资源和相互合作，从而使程序的执行具有可再现性。

当线程并发执行时，由于资源共享和线程协作，使用线程之间会存在以下两种制约关系：

间接相互制约。一个系统中的多个线程必然要共享某种系统资源，如共享 CPU，共享 I/O 设备，所谓间接相互制约即源于这种资源共享，打印机就是最好的例子，线程 A 在使用打印机时，其它线程都要等待。

直接相互制约。这种制约主要是因为线程之间的合作，如有线程 A 将计算结果提供给线程 B 作进一步处理，那么线程 B 在线程 A 将数据送达之前都将处于阻塞状态。

间接相互制约可以称为互斥，直接相互制约可以称为同步。同步包括互斥，互斥其实是一种特殊的同步。

死锁是怎样产生的？如何解决或者预防？

所谓死锁，指两个线程都在等待彼此先完成，造成程序的停滞，一般程序的死锁都在运行期产生的。

死锁产生的必要条件：

互斥条件：线程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个线程所占有。此时若有其他线程请求该资源，则请求线程只能等待。

不剥夺条件：线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走，即只能由获得该资源的线程自己来释放（只能是主动释放)。

请求和保持条件：线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

循环等待条件：存在一种线程资源的循环等待链，链中每一个线程已获得的资源同时被链中下一个线程所请求。

如何避免死锁：

加锁顺序（线程按照一定的顺序加锁）- 在主线程里使用 join() 方法按顺序执行子线程

加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）- 使用 lock.tryLock(5000, TimeUnit.MILLISECONDS) 等待5秒获取不到就放回false，进一步操作就是放弃请求，释放锁。

Java多线程同步操作如何实现？

在多线程高并发编程的时候，最关键的问题就是保证临界区的对象的安全访问。临界区指的是一个访问共用资源的程序片段，而这些共用资源又无法同时被多个线程访问的特性。

Java中实现同步的方式：

使用synchronized关键字

同步代码块：利用synchronized包装的代码块，但是需要指定同步对象，一般指定为this。这个对象习惯叫做监视器，它本身就是锁。

@Override

public void run() {

synchronized (this) {

...

}

}

Java的内置锁：每个java对象都可以用做一个实现同步的锁，这些锁成为内置锁。 线程进入同步代码块或方法的时候会自动获得该锁，在退出同步代码块或方法时会释放该锁。获得内置锁的唯一途径就是进入这个锁的保护的同步代码块或方法。java内置锁是一个互斥锁，这就是意味着最多只有一个线程能够获得该锁。

同步方法：利用synchronized定义的方法。同步代码块的锁对象其实可以为任意一个对象，但同步方法的锁对象只能是this。还有一种静态同步方法的锁对象是当前类对应的字节码文件对象。

@Override

public void run() { //调用同步方法 }

// 同步方法

private synchronized void A() {...}

使用Volatile关键字修饰变量

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：a.保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的；b.禁止进行指令重排序（在执行程序时为了提高性能，编译器和处理器通常会对指令做重排序，多线程下会影响正确性）。

注意 volatile 没有原子性，仅仅实现了对变量操作的可见性。volatile 本质是在告诉 JVM 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取。

使用并发库下的Lock锁

Lock比使用synchronized方法和语句可以获得的更广泛的锁定操作。使用Lock锁需要自己手动释放锁，灵活性更高。

所谓重入锁，是针对同一个线程而言的，指已经获得锁的情况下可以再次请求获取锁，但相应的也要释放。

ReentrantLock类是可重入、互斥、实现了Lock接口的锁， 它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力。

private static final Lock lock = new ReentrantLock(); //ReentrantLock是Lock的实现类

@Override

public void run() {

lock.lock(); // 上锁，请求不成功则等待

try{

//处理任务

}catch(Exception ex){

//处理异常

}finally{

lock.unlock(); //释放锁

}

}

使用并发编程库下的atomic包下的具有原子性的数据类型

java.util.concurrent 下的 atomic 包提供了一系列的操作简单，性能高效，并能保证线程安全的类去更新基本类型变量，数组元素，引用类型以及更新对象中的字段类型。其实现基于CAS非阻塞算法。

AtomicBoolean：以原子更新的方式更新boolean；

AtomicInteger：以原子更新的方式更新Integer;

AtomicLong：以原子更新的方式更新Long；

AtomicIntegerArray：原子更新整型数组中的元素；

AtomicLongArray：原子更新长整型数组中的元素；

AtomicReferenceArray：原子更新引用类型数组中的元素

AtomicReference：原子更新引用类型；

AtomicReferenceFieldUpdater：原子更新引用类型里的字段；

AtomicMarkableReference：原子更新带有标记位的引用类型；

AtomicIntegeFieldUpdater：原子更新整型字段类；

AtomicLongFieldUpdater：原子更新长整型字段类；

AtomicStampedReference：原子更新引用类型，这种更新方式会带有版本号。

线程局部变量ThreadLocal

参考文章1

参考文章2

ThreadLocal是一个本地线程副本变量工具类。

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。

同步与ThreadLocal是解决多线程中数据访问问题的两种思路，前者是数据共享的思路，后者是数据隔离的思路，同步是一种以时间换空间的思想，ThreadLocal是一种空间换时间的思想。

ThreadLocal类提供了三个public方法：

get()方法用于获取当前线程的副本变量值。

set()方法用于保存当前线程的副本变量值。

remove()方法移除当前前程的副本变量值。

常见操作是将ThreadLocal设置为public static修饰的全局共享的，在多线程中的run()里面调用ThreadLoca对象的set(T value)方法，T是泛型，value就表示要设置的变量。

set(T value)方法会自动获取当前线程的 ThreadLocalMap 对象，然后往这个 map 中插入一条记录， key 其实是 ThreadLocal 对象（一个线程可能会遇到多个 ThreadLocal 形式的变量，用以区分）， value 的 set 方法传进去的值。

在线程结束时可以调用 remove() 方法，这样会更快释放内存，不调用也可以，因为线程结束后也可以自动释放相关的 ThreadLocal 变量。

每个线程都有一个自己的ThreadLocal.ThreadLocalMap对象，ThreadLocalMap是ThreadLocal的内部类，没有实现Map接口，用独立的方式实现了Map的功能，在ThreadLocalMap中也是用Entry来保存K-V结构数据的。但是Entry中key只能是ThreadLocal对象，这点被Entry的构造方法已经限定死了。ThreadLocalMap中使用了独特的开发地址法来解决hash冲突。

什么是公平锁和非公平锁？

公平锁就是指线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出顺序。

非公平锁是一种获取锁的抢占机制，是随机获取锁的，和公平锁的区别就是先来的不一定先得到锁，导致某些线程可能一直拿不到锁，所以是不公平的。

如何控制某个方法允许并发访问线程的个数？

Semaphore（信号量） 用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。还可以用来实现某种资源池限制，或者对容器施加边界。

Semaphore两个构造器：

Semaphore(int permits)

创建一个 Semaphore与给定数量的许可证，非公平锁。

Semaphore(int permits, boolean fair)

创建一个 Semaphore与给定数量的许可证，fair为true时为公平锁。

通过构造器可以指定锁的个数。

获取锁：acquire() 如果请求不到就一直阻塞，直到请求通过或者线程被中断。

释放锁：release() 使用完毕，让出资源。

Java中多线程之间的通信是如何实现的？

以共享变量的形式（在共享对象供设置信号，通过获取信号和改变信号来通信）

wait / notify 机制

多线程共享数据如何实现？

分为两种情况：

多个线程行为一致，共同操作一个数据源。

如果每个线程执行的代码相同，可以使用同一个 Runnable 对象，共享数据直接定义在这个 Runnable 对象中。之后创建多个Thread对象传入同一个Runnable对象开启线程即可。卖票系统就是这么做的。

多个线程行为不一致，共同操作一个数据源。

如果每个线程执行的代码不同，这时候需要用不同的 Runnable 对象，比如存款和取款。有两种方式实现数据共享：

a. 将共享数据封装在另外一个对象中逐一传递给各个 Runnable 对象（Runnable的构造器接收）。每个Runnable对象通过传入的这个对象来操作共享数据。这样容易实现针对该数据进行的各个操作的互斥和通信。

b. 将这些 Runnable 对象作为某一个类中的内部类，共享数据作为这个外部类中的成员变量，外部类实现针对共享数据的操作方法来供内部类Runnable对象调用，以便实现对共享数据进行的各个操作的互斥和通信。

synchronized 和 volatile 关键字的区别？

volatile实现了针对变量操作的可见性和禁止指令重排序，其本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；

synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。

volatile 仅能实现变量的修改可见性，并不能保证原子性；synchronized 则可以保证变量的修改可见性和原子性

volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化（指令重排序）。

什么是线程池，如何使用？

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，使用完之后再归还给线程池。线程池作用就是限制系统中执行线程的数量，这样可以节省了开辟子线程的时间，提高的代码执行效率。

在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法：

public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的线程池。保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

调用他们的 execute 方法即可。

常用的线程池有哪些？

参照上个问题。

线程池可以带来哪些好处？

线程池的关键在于限制系统中执行线程的数量，合理使用线程池可以带来以下好处：

降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

线程池的启动策略？

线程池刚刚创建好的时候，里面并没有线程。任务队列是作为参数传进来的，执行任务前，线程池会进行一系列的判断，而不是立刻执行任务。当使用execute方法提交一个任务时：

如果正在运行的线程数量小于 corePoolSize，那么马上创建线程并运行这个任务；

如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入任务队列等待被执行。

如果任务队列已满，而正在运行的线程数小于 maximumPoolSize ，创建新线程并执行这个任务；

如果任务队列已满，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常，告诉调用者“我不能再接受任务了”。

当一个线程的任务执行完之后，它会从任务队列中取下一个任务来执行；当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于

corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

非阻塞算法CAS

乐观锁与悲观锁

悲观锁：假设并发环境是悲观的，如果发生并发冲突，就会破坏一致性，所以要通过独占锁彻底禁止冲突发生。共即享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：假定并发环境是乐观的，虽然有可能发生并发冲突，但冲突可发现且不会造成损害，所以，可以不加任何保护，等发现并发冲突后再决定放弃操作还是重试。乐观锁适用于多读的应用类型，这样可以提高吞吐量，在Java中java.util.concurrent.atomic包下面的原子变量类就是使用CAS（乐观锁的一种实现）实现的。

乐观锁的设计往往比较复杂，因此，复杂场景下还是多用悲观锁。首先保证正确性，有必要的话，再去追求性能。

“使用 CAS 控制并发”与“使用乐观锁”并不等价。 CAS 只是一种手段，既可以实现乐观锁，也可以实现悲观锁。乐观、悲观只是一种并发控制的策略。

为什么要使用CAS算法

在多线程高并发编程的时候，最关键的问题就是保证临界区（指的是一个访问共用资源的程序片段，而这些共用资源又无法同时被多个线程访问的特性）的对象的安全访问。

对于并发控制而言，锁是一种悲观策略，会阻塞线程执行。

CAS(compare and swap)基于乐观策略，有如下优势：

天生免疫死锁 （根本就无锁，何来死锁）

更优越的性能：使用无锁的方式没有所竞争带来的开销，也没有线程间频繁调度带来的开销。

CAS无锁实现原理

CAS的实现往往需要硬件的支持，多数处理器都都实现了一个 CAS 指令，实现“Compare And Swap”的语义（这里的 swap 是“换入”，也就是 set），构成了基本的乐观锁。

CAS 包含 3 个操作数：内存位置（V）、预期原值（A）和新值（B） 。

A表示要进行比较的预期旧值 ，B是拟写入的新值 ，当且仅当位置 V 的值等于 A 时， CAS 才会通过原子方式用新值 B 来更新位置 V 的值；如果V不等于A，就说明值已经被其他线程修改过了，直接将V值返回。

CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新 值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需担心线程安全问题。

版本问题

有一个特殊情况是，比如 V 本来是 = A的，但是经过了 A - B - A的过程，就会误以为其他线程没有改动过这个值，从而影响下一步决策。一个解决方法就是给值加上版本号，比如1A - 1B - 2A 这种，这样就很容易判断了。

Java的线程并发库java.util.concurrent

java.util.concurrent自JDK 5之后加入Java平台，使得Java下的并发编程变得更加简单，强大。java.util.concurrent 包含许多线程安全、测试良好、高性能的并发构建块：

java.util.concurrent.atomic (多线程的原子性操作提供的工具类）

java.util.concurrent.lock (多线程的锁机制)

Executors 线程池工厂类

工厂和工具方法Executor ， ExecutorService ， ScheduledExecutorService ，ThreadFactory和Callable在此包中定义。

线程池的作用

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

线程池作用就是限制系统中执行线程的数量。

减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为因为消耗过多的内存导致服务器死机，或过少导致效率低下。

Executors 创建线程池

Executors 类提供工厂方法用来创建不同类型的线程池：

Executor 执行器

Executor 是 Java 线程池的顶级接口，仅仅提供了execute(Runnable command)方法用来提交任务，且没有返回值。

ExecutorService 执行器服务

ExecutorService接口继承了Executor接口，提供了线程生命周期管理的方法，常用该接口来实现和管理多线程。

ExecutorService接口的实现类有：

ThreadPoolExecutor 可调整线程池中存活的线程数量

ScheduledThreadPoolExecutor 具备定时功能

ExecutorService 的创建

可以选择实例化其实现类，也可以选择使用Executors的静态方法（上述）。

ExecutorService 任务的提交

ExecutorService 中的execute(Runnable command)方法与Executor并无区别，但是由于无法接受返回值，ExecutorService提供了特有的提交任务的方法submit，是基于其父接口Executor的execute(Runnable command)扩展而来。不同于execute只能接受Runnable对象，submit方法可以接收Callable对象，这就意味着submit可以接收线程执行完毕的返回值（返回值用Future类封装）。submit方法的定义如下：

<T> Future<T>submit(Callable task)

Future<?> submit(Runnable task)

<T> Future<T> submit(Runnable task, T result)

如果传入的是Runnable对象，则返回的Future为null，但是可以用它来检测run()方法是否执行完毕。

ExecutorService 中同样提供了针对Callable任务列表的提交方法，然后可以等待全部任务或者部分执行完毕：

<T> List<Future<T>> invokeAll(Collection<? extends Callable> tasks)

<T> T invokeAny(Collection<? extends Callable> tasks)

invokeAny() 方法要求一系列的 Callable 或者其子接口的实例对象。调用这个方法并不会返回一个 Future，但它返回其中一个 Callable 对象的结果。无法保证返回的是哪个 Callable 的结果 – 只能表明其中一个已执行结束。如果其中一个任务执行结束(或者抛了一个异常)，其他 Callable 将被取消。

invokeAll() 方法将调用你在集合中传给 ExecutorService 的所有 Callable 对象。 invokeAll() 返回一系列的 Future 对象，通过它们你可以获取每个 Callable 的执行结果。

记住，一个任务可能会由于一个异常而结束，因此它可能没有 “成功”。无法通过一个 Future 对象来告知我们任务是否是正常结束还是因为异常而结束。

ExecutorService 关闭任务

使用 shutdown() 和 shutdownNow() 可以关闭线程池。二者的区别：

shutdown 只是将空闲的线程 interrupt 了， shutdown()之前提交的任务可以继续执行直到结束。

shutdownNow 是 interrupt 所有线程， 因此大部分线程将立刻被中断。之所以是大部分，而不是全部 ，是因为 interrupt()方法能力有限。

ThreadPoolExecutor 线程池执行者

ThreadPoolExecutor类是 ExecutorService 接口的一个实现。ThreadPoolExecutor 中的连接池大小可以动态变化。

池中线程的数量由以下变量决定：corePoolSize 和maximumPoolSize，称为核心线程数和最大线程数。这两个值可以通过有参构造指定，也可以通过set方法改变。

当一个新任务被提交时，如果池中正在运行的线程数小于corePoolSize，那么将会有一个新的线程被创建去执行这个任务；如果正在运行的线程数大于corePoolSize但是小于maximumPoolSize，任务进入队列等待分配线程，并且只有在任务队列已满的情况下才会去创建新线程。

这种机制的核心在于维护核心线程数的数量。当核心线程数和最大线程数相等时，等同于设置了一个固定大小的线程池。

ScheduledThreadPoolExecutor定时线程执行者

ScheduledExecutorService 是 ExecutorService 接口的子接口。它能够将任务延后执行，或者间隔固定时间多次执行，完全可以用来代替定时器。

ScheduledThreadPoolExecutor 是它的实现类。

ScheduledExecutorService的创建：

通过Executors的内置静态方法 public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) 来构造；

直接构造ScheduledExecutorService的实现类对象（ScheduledThreadPoolExecutor）来实例化。

ScheduledExecutorService 的任务执行：

 schedule (Callable task, long delay, TimeUnit timeunit)指定延迟后单次执行Future任务，timeunit为延迟参数时间

 schedule (Runnable task, long delay, TimeUnit timeunit)指定延迟后单次执行任务，timeunit为延迟参数时间

 scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)指定延迟和间隔时间，周期性执行任务。上一个任务开始后计算间隔时间开启下一次任务。

 scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit) 创指定延迟和间隔时间，周期性执行任务。上一个任务结束后计算间隔时间开启下一次任务。

ScheduledExecutorService 的关闭：

没有什么特别的方法，都使用继承自ExecutorService的 shutdown() 或 shutdownNow()方法来关闭线程。

并发队列

常用的并发队列有阻塞队列和非阻塞队列。前者使用锁实现，后者使用CAS非阻塞算法实现。都是Java并发库Java util.concurrent 下的重要组成部分。

并发队列之BlockingQueue阻塞队列

BlockingQueue 提供了线程安全的队列访问方式：

当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；

从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。

并发包下很多高级同步类的实现都是基于 BlockingQueue 实现的。BlockingQueue是一个接口，所有方法使用内部锁或其他形式的并发控制在原子上实现其效果。

应用场景

BlockingQueue实现被设计为主要用于生产者 - 消费者队列，但另外支持Collection接口。BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景：

一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

BlockingQueue的方法

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

方法\处理方式 抛出异常 返回特殊值 true / false 一直阻塞 超时退出

插入方法 add(e) offer(e) put(e) offer(e,time,unit)

移除方法 remove() poll() take() poll(time,unit)

检查方法 element() peek() 不可用 不可用

BlockingQueue不接受null元素。 使用其实现类尝试插入null值时会抛出NullPointerException异常。

BlockingQueue的实现类

 ArrayBlockingQueue： ArrayBlockingQueue 是采用数组实现的有界阻塞线程安全队列。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

 LinkedBlockingQueue： LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。

 DelayQueue： DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口。只有在延迟期满时才能从中提取元素。该队列的头部是延迟期满后保存时间最长的 Delayed 元素。DelayQueue 内部是使用 PriorityQueue 实现的。DelayQueue = BlockingQueue +PriorityQueue + Delayed。可以理解为DelayQueue 是一个使用优先队列（PriorityQueue）实现的BlockingQueue，优先队列的比较基准值是时间。

 PriorityBlockingQueue ： PriorityBlockingQueue 是一个无界的并发队列。它使用了和类java.util.PriorityQueue 一样的排序规则 。无法向这个队列中插入null值 。所有插入到PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。PriorityBlockingQueue 始终保证出队的元素是优先级最高的元素，并且可以定制优先级的规则，内部通过使用一个二叉树最小堆算法来维护内部数组，这个数组是可扩容的，当当前元素个数>=最大容量时候会通过算法扩容。

 SynchronousQueue： SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素（没有容器）。一个生产线程，当它生产产品（即put 的时候），如果当前没有人想要消费产品(即当前没有线程执行 take)，此生产线程必须阻塞，等待一个消费线程调用 take 操作， take 操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

介绍几个实现类常用的方法：

boolean offer(E e) 将元素插入队列末尾，并返回true | false

E peek() 获取队列头的元素，但是不删除它，如果队列为空则返回 null 。

E poll() 获取队列头的元素并从队列中删除它，如果队列为空则返回 null 。

boolean remove(Object o) 从该队列中删除指定元素的单个实例（如果存在）。

boolean contains(Object o) 如果此队列包含指定的元素，则返回 true 。

int size() 返回此队列中的元素数。

并发队列之非阻塞队列

与阻塞队列相反，非阻塞队列的执行并不会被阻塞，无论是消费者的出队，还是生产者的入队。在底层，非阻塞队列使用的是 CAS(compare and swap)来实现线程执行的非阻塞。

ConcurrentLinkedQueue 非阻塞无界链表队列

ConcurrentLinkedQueue 是一个线程安全的队列，基于链表结构实现，是一个无界队列，采用的也是先进先出（FIFO）入队规则。ConcurrentLinkedQueue 使用 CAS 非阻塞算法实现使用 CAS 解决了当前节点与 next 节点之间的安全链接和对当前节点值的赋值。由于使用 CAS 没有使用锁，所以获取 size 的时候有可能进行 offer， poll 或者 remove 操作，导致获取的元素个数不精确，所以在并发情况下 size 函数不是很有用。

ConcurrentLinkedQueue 中有两个 volatile 类型的 Node 节点分别用来存在列表的首尾节点，其中 head 节点存放链表第一个 item （为 null） 的节点， tail 则并不是总指向最后一个节点。 Node 节点内部则维护一个变量 item 用来存放节点的值， next 用来存放下一个节点，从而链接为一个单向无界列表。

public ConcurrentLinkedQueue() {

head = tail = new Node<E>(null);

}

初始化时候会构建一个 item 为 NULL 的空节点作为链表的首尾节点。

如何实现线程安全?

可知入队出队函数都是操作 volatile 变量： head， tail。所以要保证队列线程安全只需要保证对这两个 Node 操作的可见性和原子性，由于 volatile 本身保证可见性，所以只需要看下多线程下如果保证对着两个变量操作的原子性：对于 offer 操作是在 tail 后面添加元素，也就是调用 tail.casNext 方法，而这个方法是使用的 CAS 操作，只有一个线程会成功，然后失败的线程会循环一下，重新获取 tail，然后执行 casNext 方法。对于 poll 也是这样的。

ConcurrentHashMap 非阻塞 Hash 集合

ConcurrentHashMap 是 Java 并发包中提供的一个线程安全且高效的 HashMap 实现， ConcurrentHashMap在并发编程的场景中使用频率非常之高。

HashMap的死循环问题

HashMap的死循环问题-参考博客

HashMap的底层数据结构是数组+链表，数组（table）充当索引，链表解决冲突。

put(key,value):

判断table是否为null或者size是否为0，如果是就resize；

根据hash算法计算key值然后返回索引 i ，看table[i]处是否为null，如果为空，就把node<key,value>放入table[i]中；

如果table处不为null，就要进一步判断key值是否一致，如果一致，覆盖掉value；

如果key值不一致，就会产生冲突，解决冲突的方式是链表法，先判断是否table[i]为Tree节点，如果是，就插入到红黑树中；

如果不是，就直接插入到链表中。

插值结束，进行扩容判断，如果size太大，达到了capacity （默认16）的0.75（默认加载因子），就resize()扩容至当前的2倍。

为什么扩容大小是 2 的幂？

——HashMap在根据hash值来计算索引时，为了尽可能的使元素分布均匀且提高运算效率，使用了位运算：

static int indexFor(int h, int length) {

return h & (length-1);

}

h是通过K的hashCode最终计算出来的哈希值，length是目前容量。使用 & 运算（同为1得1），当容量是2^n时，h & (length - 1) == h % length （取余），得到的结果即为索引 i。

hash冲突的解决方法

可以看出，如果有元素根据hash值计算出了相同的索引 i ，并且key值不一致，就会产生冲突。冲突的元素会以链表的形式放在同一索引下，这种解决方法叫做链表法，另一种解决冲突的方式是开放地址法，由ThreadLocal采用。

在最坏的情况下，如果所有元素都存在冲突，那么HashMap就会变成由数组变为链表，复杂度由 O(1)变成O(n)，性能变差。JDK8对此进行了改进，当链表的长度超过8时，此后就会变成红黑树结构（复杂度为O(logn)），使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap希望key值最好是实现了Comparable接口的，这样它可以按照顺序来进行插入。

resize的重要性

从上文分析看出，resize()方法对于HashMap来说非常重要。因为table的大小直接影响冲突发生的概率，越小就越容易发生冲突，链表长度越大，查找时性能越差。

resize意味着需要重新建一张大小是当前2倍的新table，然后遍历数组，遍历链表，把元素再挨个重新放入新table中。这是一个相当耗费资源的事情。JDK1.8以前的版本中不管是put还是resize使用的都是头插法（作者认为越靠后插入的元素被查找的概率越大，放在前面可以提高查找效率），也就是说，新元素会放在链表的头部。

死循环的产生

头插法是造成多线程下HashMap产生死循环的原因，死循环是在get()时发生的，但是却是在put后扩容时产生错误的。

问题出在如果是多线程同时操作同一个hashMap，在都需要resize的情况下，由于头插法会改变next指针，多线程操作就有可能导致新table中某处形成一个循环链表，也就是链表尾部的元素的next直接指向头部元素而非null值。

HashMap的get(key)会根据key值先返回value所在索引 i ,然后去遍历 table[i] 处的链表（如果有的话），直到找出key对应的value值。

但是如果这个链表是循环链表，而查找的key值不存在，就会发生死循环（next一直不为null，相当于遍历无限长的全是重复元素的链表），导致cpu 空转。

JDK1.8之后对这一问题进行了修复，作者统一使用尾插法（不管是put还是resize），HashMap死锁的问题得到解决。

为什么不用HashTable?

HashTable是线程安全版的HashMap，但是，也仅仅是给所有的关键方法加上synchronized关键字，相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

ConcurrentHashMap 底层原理

HashTable 性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁只负责锁一段数据，这样在多线程访问不同段的数据时，就不会存在锁竞争了，这样便可 以有效地提高并发效率。这就是ConcurrentHashMap 所采用的"分段锁"思想：

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。 Segment 是一种可重入锁 ReentrantLock ，扮演锁的角色，HashEntry 用于存储键值对数据。

整个 ConcurrentHashMap 由一个 Segment 数组组成。Segment 是子哈希表，也是数组+链表结构。一个 Segment 里维护了一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素。

当对 HashEntry 的数据进行修改时，必须首先获得它对应的 Segment 锁。并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。所以，对于同一个 Segment 的操作才需考虑线程同步，不同的 Segment则无需考虑。以默认的 concurrencyLevel = 16 来说，相当于可以同时支持16个线程并发。

HashEntry是目前我们提到的最小的逻辑处理单元了，其内部维护着链表结构：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

//其他省略

}

Segment 类似哈希表，也会有负载因子loadFactor，和阈值 threshold ，以及特有的 concurrencyLevel 变量。这些值都可以选择根据构造方法指定，否则使用默认的 0.75，16，16。

public ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel){}

1

Segment 数组的大小 ssize 是由 concurrentLevel 来决定的，但是却不一定等于concurrentLevel，ssize 一定是大于或等于 concurrentLevel 的最小的 2 的次幂，这种处理方式与HashMap中定义initialCapacity类似，都是为了通过按位与的散列算法来定位 Segment 的 index。

ConcurrentHashMap 的 get 方法没有加锁 ，其中涉及到的共享变量都使用 volatile 修饰， volatile 可以保证内存可见性，所以不会读取到过期数据。

ConcurrentHashMap 的 put 方法加锁，只不过是锁粒度更细。

总的来说，ConcurrentHashMap 作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比HashTable 的全表锁在性能上的提升非常之大。

java.util.concurrent.lock 包

不同于内置同步和监视器，lock框架允许更灵活地使用锁（锁是用于通过多个线程控制对共享资源的访问的工具）和条件。本包下有三大接口：

Lock 接口：Lock实现提供比使用synchronized方法和语句可以获得的更广泛的锁定操作。锁提供对共享资源的独占访问：一次只能有一个线程可以获取锁，并且对共享资源的所有访问都要求首先获取锁。主要的实现是 ReentrantLock。

ReadWriteLock 接口： 读写锁允许访问共享数据时的并发性高于互斥锁所允许的并发性。 它利用了这样一个事实：一次只有一个线程（ 写入线程）可以修改共享数据，在许多情况下，任何数量的线程都可以同时读取数据（读取线程）。

Condition 接口： 描述了可能会与锁有关联的条件变量。这些变量在用法上与使用 Object.wait 访问的隐式监视器类似，但提供了更强大的功能。需要特别指出的是，单个 Lock 可能与多个 Condition 对象关联。 Lock替换synchronized方法和语句的使用， Condition取代了对象监视器的使用。

java.util.concurrent.atomic 包

java.util.concurrent 下的 atomic 包提供了一系列的操作简单，性能高效，并能保证线程安全的类去更新基本类型变量，数组元素，引用类型以及更新对象中的字段类型。atomic包下的这些类都是采用的是乐观锁策略去原子更新数据。

atomic类是通过自旋CAS操作volatile变量实现的。

在 java 的内存模型中每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值 load 到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。

要保证多线程操作最后得到正确的变量值，就要保证操作的原子性，atomic 的存在意义就在于此。

————————————————

版权声明：本文为CSDN博主「等一次另眼相看」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qq_44238142/article/details/100087735>

### 小米

* 1. 手写string转int
  2. Jdk8 新特性
  3. ConcurrentHashmap 为什么放弃自旋锁，Hashtable HashMap使用场景、区别
  4. JVM参数解释 xms 。。。
  5. Sychonise volitate lock 区别使用场景，具体在业务的哪里实现了
  6. 线程池中使用的synchonise，如何保证同步
  7. Spring事务声明编程方式区别、事务类型、aop使用场景代理jdk

Cglib区别使用场景、ioc单利多例，mvvc隔离级别，spring使用版本、beanfactory-applicationcontext区别

* + 1. Mysql 引擎区别、效率为什么会有不一样、分表分库原理、主从复制延时问题、 关注那些计划字段、innodb效率快在哪里为什么
    2. Zookeeper选举问题
    3. Rpc HTTP区别，使用到的框架
    4. 业务中的并发性是怎么解决的，10万台机器怎么采集的

### 跟谁学

1. https请求流程
2. countdownLatch AQS 自旋锁在哪里实现
3. 怎样开启一个线程池、线程池参数，怎样开启一个线程、线程实现方式、
4. Redis数据结构 底层实现 string实现 知道哪些
5. 最大不重复字符串
6. 项目：telemetry 代码自动修复工具 课程展示

### 头条面试

volatile 底层实现 场景 与c++区别 可见性与禁止指令重排序

类加载过程/gc过程、与python区别/jvm分区

<https://www.songma.com/news/txtlist_i41009v.html>

加载 链接（校验、准备、解析）初始化 使用 回收<https://www.cnblogs.com/williamjie/p/11167902.html>

kafka / rocketmq 两者区别、如何实现一个广播消息

zookeeper 为什么使用、kafka为什么离不开它

factory/singleton 设计模式

1. 实现一个单向链表局部翻转，每隔k个节点做翻转

h-1-2-3-4-5-nil k=2

= h-2-1-4-3-5-nil

h-1-2-nil k=10

= h-2-1-nil

* 1. hashMap 怎么存储、hashcode怎么计算、链表使用情况、
  2. AQS原理
  3. Redis排序集合实现原理
  4. 数据库隔离级别、幻读、脏读、已提交可重复区别
  5. 正则匹配\* 的实现

### 滴滴

卡顿、简历、自信、自己的项目

Hashmap扩容时的过程、产生链表环路的原因

CMS parral serise 区别 并行的方式

Spring bean周期 xml读取用到哪些接口、aop实现原理

Redis 为什么比较高速、并发度是怎么样的

[https://mp.weixin.qq.com/s?src=11&timestamp=1620473674&ver=3056&signature=24KSqSV2c6EMoBbQ-GFQaD6Rl2hWAGkQo6O21XIXw73uPScdli5t4LAE3AAXHarwH3wvif\*KFch3wH\*6y-ki\*lr0s9RUEJzmixnw5ArUeD7yacSN8tRrb8slkaM27DV7&new=1](https://mp.weixin.qq.com/s?src=11&timestamp=1620473674&ver=3056&signature=24KSqSV2c6EMoBbQ-GFQaD6Rl2hWAGkQo6O21XIXw73uPScdli5t4LAE3AAXHarwH3wvif*KFch3wH*6y-ki*lr0s9RUEJzmixnw5ArUeD7yacSN8tRrb8slkaM27DV7&new=1)

Kafka使用场景

项目 网络 知识管理 路径存储

一个接口依赖3个下游接口，且下游接口比较耗时，使用并发方式请求下游，将结果聚合之后返回

### 蚂蚁

Redis 写的源码过程

Spring bytype byname 冲突解决 覆盖 抛异常 <https://blog.csdn.net/lisongjia123/article/details/52041945>

Redis Epoll实现

epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：

1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)

2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

3）调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

原文链接：<https://blog.csdn.net/wxy941011/article/details/80274233>

遇到问题解决

采集问题、解析正则问题、ansible模板问题

源码怎么看、为什么看的

Spring/redis/kafka要搞清

Spring 配置信息是怎么加载进去的

注册发现失败的原因、服务间 问题与挑战

源码怎么看的

正则底层原理

NFA自动机引擎 贪婪模式（\*会回溯） 懒惰模式（？） 独立模式（+不会回溯） 回溯 字符逐一匹配 DFA在匹配过程中是跟正则表达式无关的，而 NFA 对于不同但效果相同的正则表达式，匹配过程是完全不同的。

### 京东物流

1. 数组与链表区别

数组：连续内存、预留空间，读取效率高o1，增改效率低on，

链表：内存随机不预留，增改效率高o1，读取效率低on;

Vector、ArrayList都是以数组的形式存储在内存中，所以查询效率高，新增和删除效率不高，但是Vector被Synchronized修饰，所以线程是安全的，ArraryList线程不安全。LinkedList则以链表的形式进行存储，所以查询效率底，新增和删除效率高，并且线程不安全。

1. LinkedHashMap使用场景

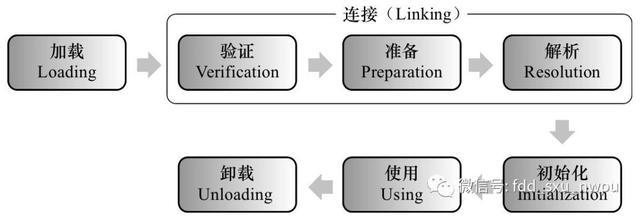
实现接口：Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

双端队列和栈来使用，双向链表实现、添加删除性能优越（使用优点）

1. Static方法中为什么不能有变量

静态方法是属于类的，动态方法属于实例对象，在类加载的时候就会分配内存，可以 通过类名直接去访问，非静态成员（变量和方法）属于类的对象，所以只有该对象初始化之后才存在，然后通过类的对象去访问，也就是说如果我们在静态方法中调用非静态成员变量会超前，可能会调用了一个还未初始化的变量。因此编译器会报错。

<http://baijiahao.baidu.com/s?id=1660679841519186374&wfr=spider&for=pc>



**加载：类名->二进制流->静态存储结构转化到运行时数据结构->堆中类对象**

**验证：确保加载类正确性**

**准备：静态变量分配内存并初始化值**

**解析：常亮池中引用转换为直接引用**

**初始化：普通变量赋值**

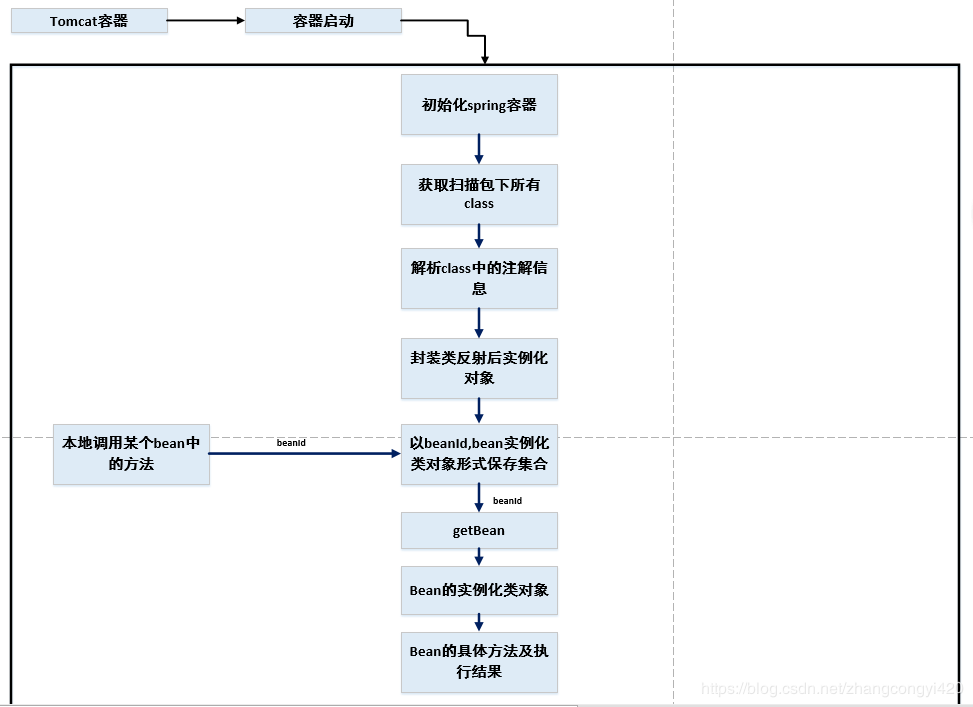
**->->**

1. IOC原理

反射机制 xml解析 实例存储结构

以前有调用者进行实例化像现在交有程序（spring）进行控制注入、所以完成了反转

<https://blog.csdn.net/ai_haibin/article/details/82455322>



<https://blog.csdn.net/zhangcongyi420/article/details/89419715>从示意图可以看出，当web容器启动的时候，spring的全局bean的管理器会去xml配置文件中扫描的包下面获取到所有的类，并根据你使用的注解，进行对应的封装，封装到全局的bean容器中进行管理，一旦容器初始化完毕，beanID以及bean实例化的类对象信息就全部存在了，（扫描后判断注解，有注解则实例化封装到concurrentMap中）现在我们需要在某个service里面调用另一个bean的某个方法的时候，我们只需要依赖注入进来另一个bean的Id即可，调用的时候，spring会去初始化完成的bean容器中获取即可，如果存在就把依赖的bean的类的实例化对象返回给你，你就可以调用依赖的bean的相关方法或属性等

1. SpringMVC 拦截器与AOP区别

<https://blog.csdn.net/fly910905/article/details/86537648>

1. 作用层面

拦截器只对action负责，作用层面一般位于Controller层

Spring AOP主要是拦截对Spring管理的Bean的访问，一般作用与Service层

2、联系与区别

联系：Spring AOP和拦截器一样，都是AOP的实现方式的一种，均使用代理模式实现。

区别：1、拦截器和过滤器有点相似，是链式的处理模式，这样有一个缺点就是，每次请求，都会访问action的上下文，不够灵活

2、Spring AOP的注解有@Before、@After、@AfterReturning、@AfterThrowing、@Around，可以更灵活的配置要监听处理的Bean

1. mybatis一级二级缓存

默认一级缓存开启并且不能关闭、一级至sqlsession的缓存，同一session中两个相同查询，第二次会直接在缓存中读取，一般能存1024条，但是中间有 commit操作时，一级缓存清除。二级缓存☞可跨sqlsession的缓存，是mapper级别的缓存，表级缓存开销大、多表更新操作会产生脏数据，所以不推荐使用。

1. mysql主从复制 延时问题

<https://blog.csdn.net/hao_yunfeng/article/details/82392261>

解决从库复制延迟的问题：

1. 优化网络

2. 升级Slave硬件配置

3. Slave调整参数，关闭binlog，修改innodb\_flush\_log\_at\_trx\_commit参数值

4. 升级MySQL版本到5.7，使用并行复制

解决数据丢失的问题：

1. 半同步复制

从MySQL5.5开始，MySQL已经支持半同步复制了，半同步复制介于异步复制和同步复制之间，主库在执行完事务后不立刻返回结果给客户端，需要等待至少一个从库接收到并写到relay log中才返回结果给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一个TCP/IP往返耗时的延迟。

2. 主库配置sync\_binlog=1，innodb\_flush\_log\_at\_trx\_commit=1

sync\_binlog的默认值是0，MySQL不会将binlog同步到磁盘，其值表示每写多少binlog同步一次磁盘。

innodb\_flush\_log\_at\_trx\_commit为1表示每一次事务提交或事务外的指令都需要把日志flush到磁盘。

（性能受到影响）

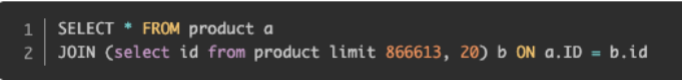
1. mysql分页使用 limit 效率低下原因 sql优化

这种分页查询方式会从数据库第一条记录开始扫描，所以越往后，查询速度越慢，而且查询的数据越多，也会拖慢总查询速度。

使用ID限定优化 或使用子查询(有临时表出现) 或使用联接查询join（最快）

select \* from orders\_history where type=2 and id between 1000000 and 1000100 limit 100;

select \* from orders\_history where type=8 and id>=(select id from orders\_history where type=8 limit 100000,1) limit 100;



我们都知道，利用了索引查询的语句中如果只包含了那个索引列（覆盖索引），那么这种情况会查询很快。

因为利用索引查找有优化算法，且数据就在查询索引上面，不用再去找相关的数据地址了，这样节省了很多时间。

另外Mysql中也有相关的索引缓存，在并发高的时候利用缓存就效果更好了。

在我们的例子中，我们知道id字段是主键，自然就包含了默认的主键索引。

<https://www.cnblogs.com/wanghongsen/p/12616558.html>

为什么 <https://www.jianshu.com/p/ba30a7c0c20b> 按顺序读取所有，然后取某一个页出来

1. cpu飙升100% 解决方案
2. springcloud 注册发现 故障推送 监控方法

### 京东2

**为什么要双亲委派**

Booststrap extentsion system 由类与类加载器才能在jvm中确定其唯一性，因此都委托到上一个加载器中就会保证

**Irs offerset**

Lsr kafka中所有的副本叫做Ar（Assigned Repllicas），同步的副本叫做lsr，未同步的叫做osr，AR = lsr + osr

Offset为消费者消费时记录的偏移量，有at most once （最多一次，fetch到消息后就会发送），at last once （最少一次，消费者成功消费后才会更新offset）

Redis 分布式锁问题 有节点挂掉后怎么处理，过期，恢复后怎么判定仍未主节点

**Schronozed原理**

https://blog.csdn.net/weixin\_38481963/article/details/88384493---带待深化

编译后会有表级ACC\_SHCHONIZED ，然后monitor监控 enter excit 底层使用互斥锁（mutex），相应的状态记录在对象头中（Mark word）锁升级为无锁 -》偏向锁 -》 轻量级锁 -》 重量级锁

**Lock原理**

<https://blog.csdn.net/qq_33945246/article/details/104037001>

有三个锁，分别为reentrantlock reentrantwritereadlock中两个静态锁 readlock writelock

各种持有sync对象，其继承于AQS，利用AQS的state机制与获取释放机制来实现锁得功能

**AQS原理**

简单说来是：AQS 基于 CLH 队列，用 volatile 修饰共享变量 state，线程通过 CAS 去改变状态符，成功则获取锁成功，失败则进入等待队列，等待被唤醒。

CLH: CLH是一种基于单向链表的高性能、公平的自旋锁。申请加锁的线程通过前驱节点的变量进行自旋。在前置节点解锁后，当前节点会结束自旋，并进行加锁

所以lock的存储结构为：一个 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），本质为：使用CAS 来获取状态值修改，如果当场没获取到，会将该线程放在线程等待链表中。

公平非公平区别：非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

**Springcloud feign作用 路由 cloudsop总线机制**

Feign是一个http请求调用的轻量级框架，也可以说是声明式WebService客户端

底层通过动态代理将调用接口的请求生成requestTemplate,再生成request，再交由client均衡处理

Mysql可重复读原理 索引优化

MVCC多版本并发控制(Multi-Version Concurrency Control)

Nio io 原理

Io模型是，户线程不断查询内部线程缓冲区是否有数据，若果有则等待复制到用户缓冲区接触blocking返回，没有则立即返回（整个过程看起来像是非阻塞的）

Java模型使用的是用多路复用器，select/epoll,(Linux的epoll),,,,这种新的系统调用系统可以监视多个文件描述符,一旦一个就绪,内核就会通知相应用户进程进行调用,平时会轮训那个处于就绪状态,epoll会主动感知那个就绪二减少轮训带来的cpu浪费,,本质上讲IO读写过程也是阻塞的. https://www.cnblogs.com/crazymakercircle/p/10225159.html

Fork jion

Kafka 写过程读过程leader follower怎么完成的怎么区分的

零拷贝原理 DMA

### 美团

1. Hashmap为什么不是线程安全的，concurrentHashmap为什么是安全的，底层实现 100个固定数怎么初始化

<https://www.jianshu.com/p/e2f75c8cce01> put时数据不一致、get时引起死循环

<https://blog.csdn.net/qq_41737716/article/details/90549847> 各个操作安全

底层实现，初始化数值以2的N次方进行存储，100 -> 128 加上扩容因子 128\*0.75=96，故初始化256最好。

1. FullGC定位，性能调优
2. Kafka大体结构 消费者组的作用，高吞吐的底层实现原理 消息不重复消费原理

<https://www.cnblogs.com/songanwei/p/9202803.html>

<https://blog.csdn.net/camel84/article/details/82861053>

顺序读写 零拷贝 分区 数据压缩 批量发送 <https://blog.csdn.net/yamaxifeng_132/article/details/86695657>

1. Redis与memcache区别 使用场景 挂掉怎么办 大量键值过期怎么办 aof rdb为什么会有这两种方式区别在哪里

缓存雪崩

1. 设计模式知道哪些 单例安全性的解决
2. 高并发的场景使用

### 苏宁

1. Kafka存储时间
2. Char强转int
3. ACDI特性
4. 事务嵌套 7种
5. 索引优化
6. 符合索引优化
7. Sql优化
8. Dobbou 框架有spring区别 rpc 与rest区别
9. 负载均衡如何实现 springcloud全套讲清楚

### T3出行

1. 前后端数据机交互过程 DNS ngix tomcat 序列化问题
2. Tomcat模式
3. Mybatis设计模式 链式模式
4. Mvvc模式功能
5. 非聚簇索引查询过程 回表查询
6. 缓存 一致性解决
7. Redis主从方式 怎样分离 分布式锁
8. Px ex nx区别 数据一致性问题
9. Cap
10. Zookeeper Eruka实现区别
11. Reentralock实现原理
12. 线程池操作过程
13. 阻塞队列的区别 为什么这样做

### 拼多多

为什么使用工厂模式，具体优势有哪些

Hash不平衡处理办法

数据湖概念与数据库云平台区别

Redis数据持久化

# 第四节：新挑战

### 小米

工作难点是什么 难在哪里

技术框架那个比较熟

Redis数据结构

Redis 存储方式优缺点

Redis主从复制、断点续传

断电续传：master中有个backlog文件记录文件传输的位移offset，当传输断掉之后，会继续重此店续传，当找不到了会重新传

https://www.cnblogs.com/mrmirror/p/13583715.html

Redis使用优缺点

Redis

Mysql索引方式

二叉树特点

中间件开发过吗

分布式锁实现

<https://blog.csdn.net/wuzhiwei549/article/details/80692278>

Setnx隐患

如何设计高并发系统

网络项目中的大量数据怎么处理的

### 快手

**实际的问题，现实中现实的**

**图谱的关系，实现了一个什么样子的功能，有挑战的地方 课程买卖**

**题目数量 怎么分库的，之后怎么访问的**

**类加载机制 双亲委派原理，为什么要双亲委派**

**Java 泛型原理**

Java 虚拟机会将泛型的类型变量擦除，并替换为限定类型（没有限定的话，就用 Object）。可以使用关键字 extends 限定子类，也可以使用关键字 super 限定父类。

List<? extends Number> list = new ArrayList<>()

Java中是强类型的，就是说任何变量在具体使用时，必须明确类型。上述list在定义时，表述其中的每一项值都是Number本身或者是其子类，但是在add时（list.add( new Long(1) ) 看似没错），但编译器它却不知道这个list到底放的是啥类型，有可能是ArrayList<Short>,ArrayList<Test> 等，因为不确定所以就不允许你add一个long或者short。

从这个list定义中，在获取值时，编译器知道的是里面放的是Number类型或者是其子类型值，因此Number n = (Number)list.get(0), Long l = (Long)list.get(0)可以编译通过。

**Redis 热键 解决方案 string数据结构**

https://www.toutiao.com/a6694032493512753671/

As .热键问题：

描述：热键被大量客户端访问，导致大量网络流量集中在一台[Redis](https://cloud.tencent.com/product/crs?from=10680)服务器上，服务器宕机。

解决方法：将热键分散到不同的redis服务器上

2.缓存击穿：

描述：热键被大量客户端访问，热键如果过期，可能导致大量网络流量打到 DB 服务器上，导致数据库服务器宕机

解决方法：

对于热键，访问之前如果不存在的话使用分布式锁（应用线程访问不到，一般返回false，就让应用线程 自己执行等待逻辑）

异步刷新这个键的过期时间，延迟

在DB服务器和 redis服务器间多设置一层缓存

3.缓存穿透：

描述：一个键根本就不存在，但是还是被大量的访问，轻则增大数据库的压力，重则数据库宕机

解决方法：布隆过滤器，过滤掉不存在的key请求

4.缓存雪崩：

描述：大量的热键过期，或者redis数据库宕机，导致大量网络流量打到 DB 上，DB宕机

解决方法：

* 多设一层缓存
* 避免大量热键同时过期
* 当redis宕机的时候，要给网络限流

**数据库 主从复制方式 原理 讲一讲innoDB 隔离机制 幻读 间隙锁**

主从复制，首先master主机会在有数据进行更新的时候将更新事件写入到binlog文件中，通过log dump线程通知slave端，从机会通过IO线程将主机中binlog文件位置读取回来并存入本机ready log中，当检测到中继日志（readylog）有变化时会使用sql Thread将数据进行同步，，，同步策略有：同步（slave全部答复）半同步（slave有一个答复）异步（不等待答复）延迟（落后）

主从复制有不同的复制策略，对于不同的场景的适应性也不同，对于数据的实时性要求很高，要求强一致性，可以采用同步复制策略，但是这样就会性能就会大打折扣。

若是主从复制采用异步复制，要求数据最终一致性，性能方面也会好很多。只能说，对于数据延迟的解决方案没有最好的方案，就看你的业务场景中哪种方案使比较适合的。

行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表锁。

最常使用的MYISAM与INNODB都支持表级锁定（行锁失效时使用表锁）。

https://blog.csdn.net/chenyiminnanjing/article/details/82714341

并行事务的四大问题：

1.更新丢失：和别的事务读到相同的东西，各自写，自己的写被覆盖了。（谁写的快谁的更新就丢失了）

2.脏读：读到别的事务未提交的数据。（万一回滚，数据就是脏的无效的了）

3.不可重复读：两次读之间有别的事务修改。

4.幻读：两次读之间有别的事务增删。

对应隔离级别

1.READ UNCOMMITTED：读未提交，不处理。

2.READ COMMITTED：读已提交，只读提交的数据，无脏读；

3.REPEATABLE READ：可重复读，加行锁，两次读之间不会有修改，无脏读无重复读；

4.SERIALIZABLE: 串行化，加表锁，全部串行，无所有问题。

**100 的生产者、消费者 数据池实现 IntStream.rangeClosed(1, 20)**

**深度不够**

**表达有问题**

**二面**

数据库分表

长链变短链的实现方式

<https://blog.csdn.net/youanyyou/article/details/104305989/>

Java锁 synchronized ReentryLock区别,各自原理，内部公平锁非公平锁的实现，monitor 字节头内容，锁膨胀， 轻量级重量级区别

Redis

GC cms原理，使用什么方式 新生代Parremel,,,根节点寻找方式，怎么过滤的，，,怎么看到的

什么时候OMM

注册中心 原理

RPC使用的什么

排行榜实现，定时更新怎么办，数据库 redis大量访问办法， 谷歌内部cofffiin实现原理，过期策略，，order原理，，还有哪些方法

Kafka中间件保证消息不丢失 磁盘坏掉后怎么办

父类 静态代码块，静态变量，变量，构造方法

子类 静态代码块，静态变量，变量，构造方法

new 子类，执行顺序

一个文本文件，共4列，以“|”分割。第一列到第四列，分别表示“帐号|性别|联系方式|注册省份”。其中第一列帐号和第四列注册省份为非空字段，其他列均可能为空。编写程序实现从该文件中读出内容，统计在每个省份注册帐号的数量，将结果按降序排列输出结果。

别的方法

**三面**

**在互联网中的优势 项目优势 你在项目中做了什么 突出的地方**

**背景潜力**

**山脉计算**

### 百度

**监控怎么使用的，用的什么，**

<https://blog.csdn.net/haponchang/article/details/93469050>

<https://zhuanlan.zhihu.com/p/28851786>

基础框架、系统层、应用层、业务层、客户端的监控

1、日志监控

2、Metrics监控

3、调用链监控

4、报警系统

5、健康检查

调用链

通过收集日志，对系统和各个服务的运行状态进行监控（ELK）

通过收集量度（Metrics），对系统和各个服务的性能进行监控(Prometheus 容器)

通过分布式追踪，追踪服务请求是如何在各个分布的组件中进行处理的细节（Zipkin和Jaeger）

Zipkin + sleuth

基于Opentracing开发分布式实时数据追踪系统：span(基本调用单元) trace（完整链路）spancontexttrace（上下文信息）

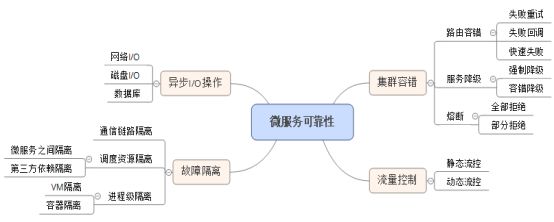
zipkin ：拦截请求，发送（HTTP，MQ）数据至zipkin服务，需要的话可以持久化到数据库。

SkyWalking：使用 agent的形式去接入业务系统,这样就不需要在业务系统里添加任何的日志代码,也能记录到对应的调用数据库。

**可靠性设计**

微服务化之后，系统分布式部署，传统单个流程的本地 API 调用被拆分成多个微服务之间的跨网络调用，由于引入了网络通信、序列化和反序列化等操作，系统发生故障的概率提高了很多。

微服务健康度检测 注册中心； 网络IO 磁盘IO 异步非阻塞；故障隔离 线程池隔离、相应式编程 第三方依赖隔离 vm容器隔离 链路隔离；容错 路由 服务 熔断； 流控 静态 动态



**数据库主从同步延时怎么办 binlog readylog**

主库产生的ddl dml 语句到binlog中，从库读取后是单sql线程执行的，并且是无顺序执行的，效率较慢而且会与从库查询碰撞遇到死锁问题。

最简单的减少slave同步延时的方案就是在架构上做优化，尽量让主库的DDL快速执行。还有就是主库是写，对数据安全性较高，比如sync\_binlog=1，innodb\_flush\_log\_at\_trx\_commit = 1 之类的设置，而slave则不需要这么高的数据安全，完全可以讲sync\_binlog设置为0或者关闭binlog，innodb\_flushlog也可以设置为0来提高sql的执行效率。另外就是使用比主库更好的硬件设备作为slave。flush logs命令的作用就是关闭当前使用的binary log，然后打开一个新的binary log文件，文件的序号加1,，，,内存日志， innodb\_flush\_log\_at\_trx\_commit是将事务日志从innodb log buffer写入到redo log中，sync\_binlog是将二进制日志文件刷新到磁盘上。<https://blog.csdn.net/thundermeng/article/details/50448614>

硬件

缓存

分库分压力

[https://mp.weixin.qq.com/s?src=11&timestamp=1617623501&ver=2990&signature=wVux-R2vXpy9QonXcg\*zcshxvtQYNXkhtPU2JKYC2h8fbpcz1Vw6vqPe\*ZjUfLtt1X-FQiHdPDhT-vmHBkt5HFNIWAaxlcqIY1D1lyiTbpS5j24dNkOZpfd0HoAlUohZ&new=1](https://mp.weixin.qq.com/s?src=11&timestamp=1617623501&ver=2990&signature=wVux-R2vXpy9QonXcg*zcshxvtQYNXkhtPU2JKYC2h8fbpcz1Vw6vqPe*ZjUfLtt1X-FQiHdPDhT-vmHBkt5HFNIWAaxlcqIY1D1lyiTbpS5j24dNkOZpfd0HoAlUohZ&new=1)

**Redis主从同步方式，aof rdb区别，复制时用哪种，实现，单线程为什么快，到底快在哪里 数据raft一致性**

初次同步会使用rdb方式进行同步，过程中主节点会记录写命令rof然后同步给从节点。：sync->bgsave->rdb->缓冲区->psync

发送 SLAVEOF 命令可以进行主从同步，比如：SLAVEOF 127.0.0.6379

主从同步有同步和命令传播 2 个步骤。

同步：将从服务器的数据库状态更新成主服务器当前的数据库状态（一个消耗资源的操作）

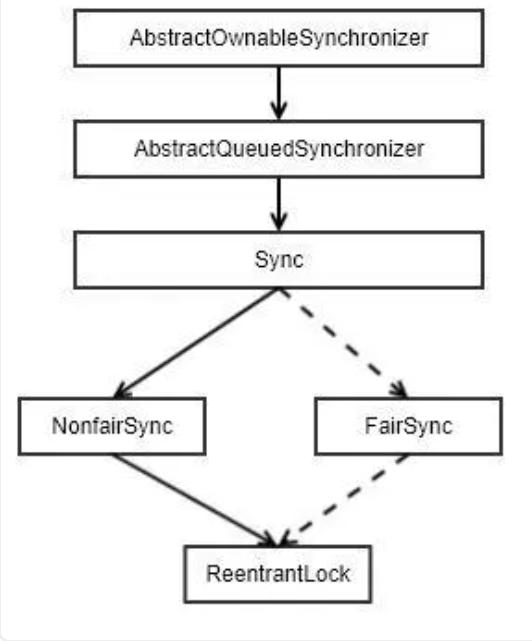
命令传播：当主服务器数据库状态被修改后，导致主从服务器数据库状态不一致，此时需要让主从数据同步到一致的过程

主从同步分初次复制和断线后重复制两种情况

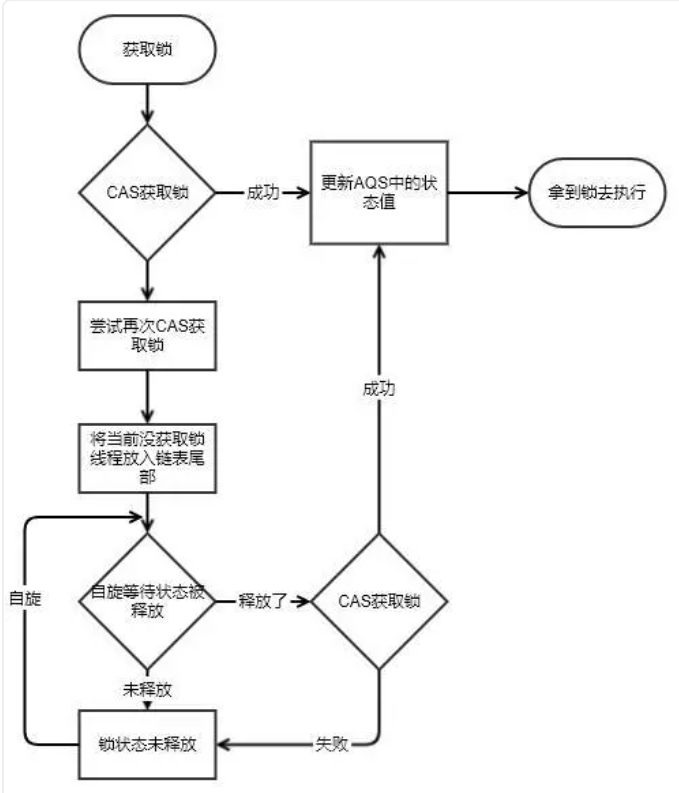
从 2.8 版本开始，在出现断线后重复制情况时，主服务器会根据复制偏移量、复制积压缓冲区和 run id，来确定执行完整重同步还是部分重同步

2.8 版本使用 psync 命令来代替 sync 命令去执行同步操作。目的是为了解决同步（sync 命令）的低效操作

**用过哪些锁，Lock实现**

****

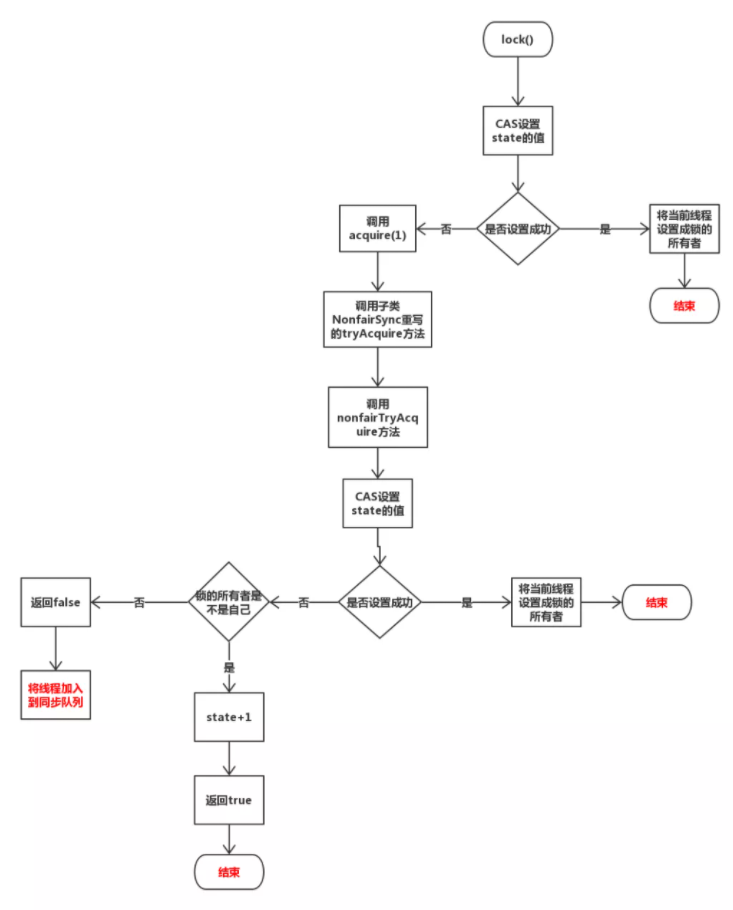
**锁的存储结构就两个东西:"双向链表" + "int类型状态"。**



基本过程是：尝试cas改变state状态，成功则获取到锁，未成功则未获取到到再次尝试，再没有这加入双向链表尾部，自旋等待cas获取，释放锁就是对AQS中的状态值State进行修改。同时更新下一个链表中的线程等待节点。

AQS即AbstractQueuedSynchronizer的缩写，这个是个内部实现了**两个队列**的抽象类，分别是**同步队列**和**条件队列**。其中**同步队列**是一个双向链表，里面储存的是处于等待状态的线程，正在排队等待唤醒去获取锁，而**条件队列**是一个单向链表，里面储存的也是处于等待状态的线程，只不过这些线程唤醒的结果是加入到了同步队列的队尾，AQS所做的就是管理这两个队列里面线程之间的**等待状态-唤醒**的工作。  
在同步队列中，还存在2种模式，分别是**独占模式**和**共享模式**，这两种模式的区别就在于AQS在唤醒线程节点的时候是不是传递唤醒，这两种模式分别对应**独占锁**和**共享锁**。

lock方法调用CAS方法设置state的值，如果state等于期望值0(代表锁没有被占用)，那么就将state更新为1(代表该线程获取锁成功)，然后执行setExclusiveOwnerThread方法直接将该线程设置成锁的所有者。如果CAS设置state的值失败，即state不等于0，代表锁正在被占领着，则执行acquire(1)，即下面的步骤。

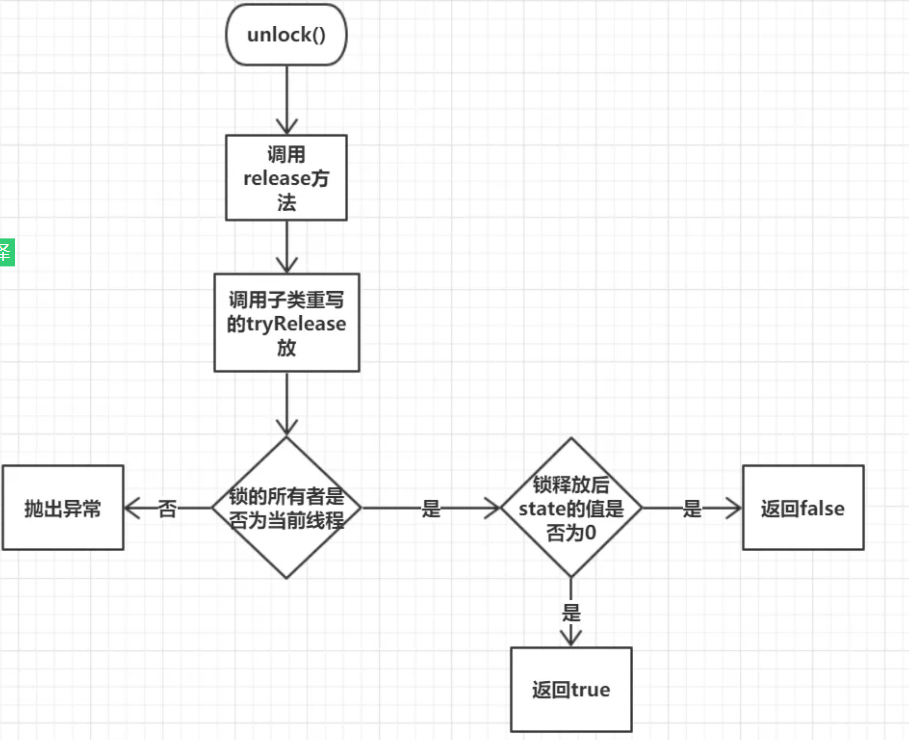
**非公平获取：**nonfairTryAcquire方法首先调用getState方法获取state的值，如果state的值为0(之前占领锁的线程刚好释放了锁)，那么用CAS这是state的值，设置成功则将该线程设置成锁的所有者，并且返回true。如果state的值不为0，那就调用getExclusiveOwnerThread方法查看占用锁的线程是不是自己，如果是的话那就直接将state + 1，然后返回true。如果state不为0且锁的所有者又不是自己，那就返回false，然后线程会进入到同步队列中。

**非公平释放：**1.判断当前线程是不是锁的所有者，如果是则进行步骤2，如果不是则抛出异常。

2.判断此次释放锁后state的值是否为0，如果是则代表锁有没有重入，然后将锁的所有者设置成null且返回true，然后执行步骤3，如果不是则代表锁发生了重入执行步骤4。

3.现在锁已经释放完，即state=0，唤醒同步队列中的后继节点进行锁的获取。

4.锁还没有释放完，即state!=0，不唤醒同步队列。

****

**公平锁：**

1. 获取状态的state的值，如果state=0即代表锁没有被其它线程占用(但是并不代表同步队列没有线程在等待)，执行步骤2。如果state!=0则代表锁正在被其它线程占用，执行步骤3。
2. **判断同步队列是否存在线程(节点)，如果不存在则直接将锁的所有者设置成当前线程，且更新状态state，然后返回true。**
3. **判断锁的所有者是不是当前线程，如果是则更新状态state的值，然后返回true，如果不是，那么返回false，即线程会被加入到同步队列中**

通过步骤2**实现了锁获取的公平性，即锁的获取按照先来先得的顺序，后来的不能抢先获取锁，非公平锁和公平锁也正是通过这个区别来实现了锁的公平性。**

**公平既是保证了同步队列中先到的节点先获取，非公平是大家被唤醒后一起抢谁抢到是谁的。**

**等待通知机制：类似于object wait notify**

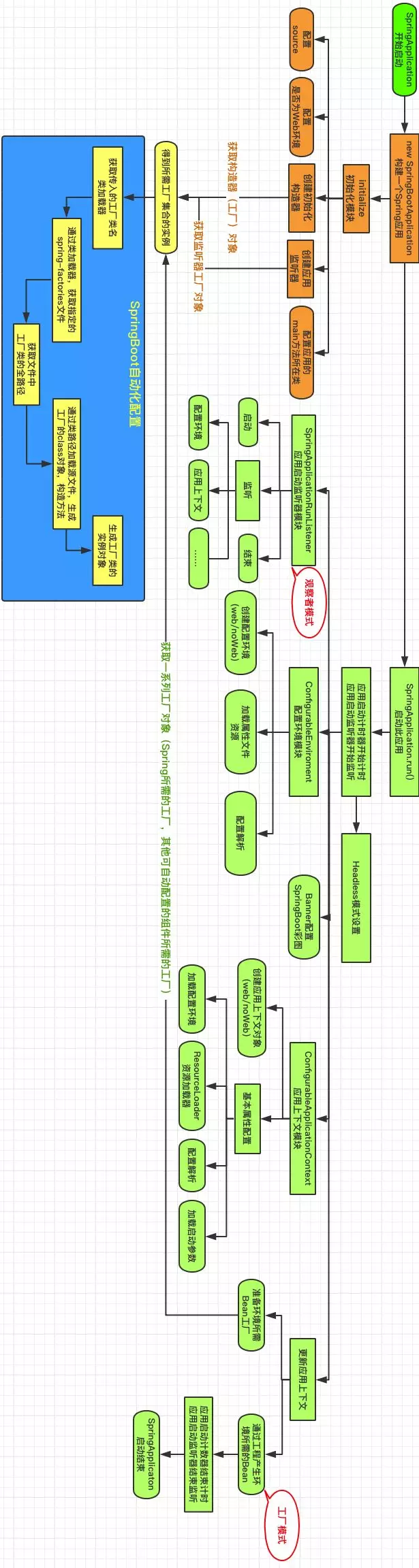
**线程执行condition.await()方法，将节点1从同步队列转移到条件队列中。会释放锁。**

**线程执行condition.signal()方法，将节点1从条件队列中转移到同步队列。**

因为只有在同步队列中的线程才能去获取锁，所以通过Condition对象的wait和signal方法能实现等待/通知机制。

**Springboot启动类，启动流程**

[**https://mp.weixin.qq.com/s?src=11&timestamp=1618101366&ver=3001&signature=wzoVnXhvkNHdPFfvQkMHw550cdS0MLM3R6UU7QSV2MmzFOCnQN3vRw0pAIBsAVPECEfp-TbOfnBDkt2n4AOO9qxgg44O93TJtHiPqDJh7aav2MNEmtSUJBbLN6dDcEfi&new=1**](https://mp.weixin.qq.com/s?src=11&timestamp=1618101366&ver=3001&signature=wzoVnXhvkNHdPFfvQkMHw550cdS0MLM3R6UU7QSV2MmzFOCnQN3vRw0pAIBsAVPECEfp-TbOfnBDkt2n4AOO9qxgg44O93TJtHiPqDJh7aav2MNEmtSUJBbLN6dDcEfi&new=1)

****

public ConfigurableApplicationContext run(String... args) {

// 开启一个定时器

StopWatch stopWatch = new StopWatch();

stopWatch.start();

ConfigurableApplicationContext context = null;

Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();

configureHeadlessProperty();

// 获取监听器

SpringApplicationRunListeners listeners = getRunListeners(args);

// 执行Spring应用刚启动时的前置事件

listeners.starting();

try {

ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);

// 创建并配置环境变量

ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);

// 配置忽略BeanInfo

configureIgnoreBeanInfo(environment);

// 获取打印Banner对象

Banner printedBanner = printBanner(environment);

// 创建上下文对象，根据项目类型创建不同的，主要为Servlet与Reactive(WebFlux),refreshContext

context = createApplicationContext();

// 加载spring.factories文件

exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class,

new Class[] { ConfigurableApplicationContext.class }, context);

// 上下文准备工作，主要为上下文的后置处理、注册banner对象、懒加载处理等

prepareContext(context, environment, listeners, applicationArguments, printedBanner);

// 刷新上下文，springboot启动的核心方法

refreshContext(context);

// 刷新上下文的后置方法

afterRefresh(context, applicationArguments);

// 停止计时器

stopWatch.stop();

if (this.logStartupInfo) {

new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);

}

listeners.started(context);

callRunners(context, applicationArguments);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, listeners);

throw new IllegalStateException(ex);

}

try {

// 执行Spring应用正在运行事件

listeners.running(context);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, null);

throw new IllegalStateException(ex);

}

return context;

}

refreshContext方法是整个项目启动的核心方法，其执行对象源于createApplicationContext,如果当前应用类型为Servlet，其返回的上下文类型为：org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext，不过最终起作用的还是抽象类里的refresh方法org.springframework.context.support.AbstractApplicationContext#refresh。

在AbstractApplicationContext类中提供了onRefresh方法，其作用是在特定的上下文子类中初始化其他特殊操作。在ServletWebServerApplicationContext类中重写了该方法，其内容如下：

在AbstractApplicationContext类中提供了onRefresh方法，其作用是在特定的上下文子类中初始化其他特殊操作。在ServletWebServerApplicationContext类中重写了该方法，其内容如下：

其中createWebServer核心的方法又为factory.getWebServer(getSelfInitializer())，该方法会根据当前上下文环境去获取不同的servlet服务器工厂(org.springframework.boot.web.servlet.server.ServletWebServerFactory)。

[**https://jiangliuhong.top/2021/03/18/java/spring/springboot\_qi\_dong\_guo\_cheng\_fen\_xi/**](https://jiangliuhong.top/2021/03/18/java/spring/springboot_qi_dong_guo_cheng_fen_xi/)

**RPC框架使用过哪些**

**与socket区别：**RPC（远程过程调用）采用客户机/服务器模式实现两个进程之间相互通信。socket是RPC经常采用的通信手段之一，RPC是在Socket的基础上实现的，它比socket需要更多的网络和系统资源。除了Socket，RPC还有其他的通信方法，比如：http、操作系统自带的管道等技术来实现对于远程程序的调用。微软的Windows系统中，RPC就是采用命名管道进行通信。好比http是普通话，rpc是团伙内部黑话

与rest区别：REST API 和 RPC 都是在 Server端 把一个个函数封装成接口暴露出去，以供 Client端 调用，不过 REST API 是基于 HTTP协议的，REST致力于通过http协议中的POST/GET/PUT/DELETE等方法和一个可读性强的URL来提供一个http请求。而 RPC 则可以不基于 HTTP协议

因此，如果是后端两种语言互相调用，用 RPC 可以获得更好的性能（省去了 HTTP 报头等一系列东西），应该也更容易配置。如果是前端通过 AJAX 调用后端，那么用 REST API 的形式比较好（因为无论如何也避不开 HTTP 这道坎）。RPC主要是(可以是)基于TCP/IP协议的，而HTTP服务主要是基于HTTP协议的，我们都知道HTTP协议是在传输层协议TCP之上的，所以效率来看的话，RPC当然是要更胜一筹啦.restful是一种开发风格。

Dobbo

gRPC是Google开发的高性能、通用的开源RPC框架，其由Google主要面向移动应用开发并基于HTTP/2.0协议标准而设计，基于ProtoBuf(Protocol Buffers)序列化协议开发，且支持众多开发语言。本身它不是分布式的，所以要实现上面的框架的功能需要进一步的开发。

Thrift

**什么是好代码**

**代码规范**

**代码风格**

**好看（简约）+安全（易于写出测试用例）+高效（算法、数据结构得当）**

**设计原则，解耦、六大设计原则 23模式等等**

**设计模式用过哪些，实例比枚举类更好的实现方式**

[**https://mp.weixin.qq.com/s?src=11&timestamp=1617800657&ver=2994&signature=98Zzxiwei0aow-FN6zV8eOHlXQZ2qsqir8cEzrc4itouXVAvbMu8YvlxwM-Sw6v3W4MQxOnrcg\*G8kDSXAFGXMwTJbI1VT3Hca15-5vZglZK-Q2qYMrhXhIv0ITyD6BW&new=1**](https://mp.weixin.qq.com/s?src=11&timestamp=1617800657&ver=2994&signature=98Zzxiwei0aow-FN6zV8eOHlXQZ2qsqir8cEzrc4itouXVAvbMu8YvlxwM-Sw6v3W4MQxOnrcg*G8kDSXAFGXMwTJbI1VT3Hca15-5vZglZK-Q2qYMrhXhIv0ITyD6BW&new=1)

反射破坏 反序列化破坏

**领域驱动设计**

Userinterface用户界面层 负责构建用户界面并管理与领域模型的交互。典型的应用是MVC模式。

application应用层 允许视图层与领域层协作的中间层。注意：这是容易聚集比较散乱的行为，诸如“事务脚本”风格的代码。

domain领域层 领域层通常有着丰富的行为并且富有表达力，注意到仓库和工厂也是领域层的一部分。但是，ORM框架会使得仓库层将它的部分功能委托给底层的基础设施。

infrastructure基础设施 主要处理特定技术领域的决策，侧重于实现而非意图。注意到虽然领域实例可以在这一层被创建，但是与这层进行交互的往往是仓库层，以获得对这些对象的引用。

我们的目标是设计好层和接口，并且让这些接口做好层与层之间的通讯。此外，要让使用域层的代码控制事务边界

**高并发的设计**

**本公司的java技术怎么样**

[**https://mp.weixin.qq.com/s?src=11&timestamp=1618043554&ver=2999&signature=werkG1CWBUJv4AFbzUDXu5pvS\*KPeFl98EwIvjlkMaVRUgeXiX4DCIWpGmqMLj2g25na0Rn5tRhu75XAItDOwZUlwN1vg\*488kbV4JfqyvVNX0NV--zWK4gDFKpNAkkm&new=1**](https://mp.weixin.qq.com/s?src=11&timestamp=1618043554&ver=2999&signature=werkG1CWBUJv4AFbzUDXu5pvS*KPeFl98EwIvjlkMaVRUgeXiX4DCIWpGmqMLj2g25na0Rn5tRhu75XAItDOwZUlwN1vg*488kbV4JfqyvVNX0NV--zWK4gDFKpNAkkm&new=1)

分表和在用途上不一样，分表是为了承接超大规模的表，单机放不下那种。分区的话则一般都是放在单机里的，用的比较多的是时间范围分区，方便归档。

InnoDb数据存储类型：段 -》 区 -》 页，，，一个表空间中有多个段，可有数据段，索引段、回滚段，每个段中有多个区，，，每个区的大小是固定的为1M，默认有64页（16Kb），一次申请3~4段，32个碎片页 + 64连续页，页中数据页、系统页

不是上面的区，分区的意思是指将同一表中不同行的记录分配到不同的物理文件中，几个分区就有几个.idb文件，不是我们刚刚说的区。MySQL数据库的分区是局部分区索引，一个分区中既存了数据，又放了索引。RANGE分区是实战最常用的一种分区类型，行数据基于属于一个给定的连续区间的列值被放入分区。RANGE分区主要用于日期列的分区，比如交易表啊，销售表啊等。

<https://blog.csdn.net/jiang7701037/article/details/81839899>

分区，一个逻辑表，多个物理存储，只是数据块变为了多个，**表描述并没有**，，，主要优点是缩小查询范围

分区和分表的测重点不同，分表重点是存取数据时，如何提高mysql**并发能力上**；而分区呢，如何突破**磁盘的读写能力**，从而达到提高mysql性能的目的。

分表和分区不矛盾，可以相互配合的，对于那些大访问量，并且表数据比较多的表，我们可以采取分表和分区结合的方式..大访问量的可以分表，小访问量大数据可以分区。

<https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247493366&idx=3&sn=55c684e0a22dddd3f54b9fc6033cf043&chksm=eb5063c0dc27ead610a788fc39746f3406ffeaee19917e9429a5e11f02f186f3ab30222dfd9f&scene=21#wechat_redirect>

分库主要是多实例

[https://mp.weixin.qq.com/s?src=11&timestamp=1618043554&ver=2999&signature=werkG1CWBUJv4AFbzUDXu5pvS\*KPeFl98EwIvjlkMaVRUgeXiX4DCIWpGmqMLj2g25na0Rn5tRhu75XAItDOwZUlwN1vg\*488kbV4JfqyvVNX0NV--zWK4gDFKpNAkkm&new=1](https://mp.weixin.qq.com/s?src=11&timestamp=1618043554&ver=2999&signature=werkG1CWBUJv4AFbzUDXu5pvS*KPeFl98EwIvjlkMaVRUgeXiX4DCIWpGmqMLj2g25na0Rn5tRhu75XAItDOwZUlwN1vg*488kbV4JfqyvVNX0NV--zWK4gDFKpNAkkm&new=1)

分布式数据库中间件分为两种，proxy和客户端式架构。proxy模式有MyCat、DBProxy等，客户端式架构有TDDL、Sharding-JDBC等

柔性事务是目前主流的方案，TCC模式就属于柔性事务。

对于分布式事务问题每家公司有自己的实现，华为用saga，阿里用TXC，蚂蚁用DTX，支持FMT模式和TCC模式。

### 美团

**挑战在哪里**

**Omm问题解决，，，之间的关系**

**Mysql慢查询问题解决，，，逻辑，前后步骤**

**Janusgraph的es查询为什么会快，，，属性存在里面问什么会慢**

**Cse相对spring优点在哪里**

**A b 有序数组排序**

**Nio 为什么快**

**解决问题的步骤，前后顺序**

**大佬的面向，想听什么，，，是个重视什么的人**

多聊 多讲

### 京东物流

Mvvc

Linkedlist Arraylist 暂用内存情况比较

一般来说，LinkedList对内存的占用也是比较大的，毕竟每个Node都维护着前后指向地址的节点，数据量大的话会占用不少内存空间，但是，如果刚好数据量超过ArrayList默认的临时值时，ArrayList占用的空间也是不小的，因为扩容的原因会浪费将近原来数组一半的容量，不过，因为ArrayList的数组变量是用transient关键字修饰的，如果集合本身需要做序列化操作的话，ArrayList这部分多余的空间不会被序列化。

原文链接：https://blog.csdn.net/haozige888/article/details/107797713

arraylist初始化10扩1.5并不是一开始就有10个数据而是初始化化数组的大小 照样占用空间 transient

liknedList 双向链表 查询时 index 总链表长度的/2 那头近从哪儿开始查

ConcurrentHashMap 原理

多线程 8个线程为什么这样限定 最大线程池 threadpoolExecutor 线程池慢后怎么办

Mysql 引擎 事务隔离级别 多版本控制 mvvc用在哪里 sql优化 执行计划关键字慢日志 回表 如何避免

Spring 通信模式 事务传播机制

Springboot 自动装配文件 原理 与Spring区别

Dobble 使用

Redis 使用场景 雪崩 解决方法 分布式锁

设计模式 使用场景 行为与结构区别

MQ 场景 两个数据库为什么不调用接口 为什么用消息队列

### 滴滴

业务技术框架

Jvm 内存结构 分别存储什么数据，私有共享 1.7 1.8区别 元数据 堆外存储怎么预防OMM 数据那些放常量

**JVM内存结构，和Java虚拟机的运行时区域有关。  
Java内存模型，和Java的并发编程有关。  
Java对象模型，和Java对象在虚拟机中的表现形式有关。**

装入jvm过程：加载（字节码 双亲委派 启动类加载器 –>扩展类加载器->应用程序系统类加载器）

链接(验证 结构分析 链接)

初始化(静态属性赋值 静态块的执行)

方法区:类加载信息,常量,静态变量,常量池 线程共享 1.8后元空间替代方法区,其不在jvm内存中而是在本地内存中(堆外内存)

堆区:对象实例 非静态成员变量 最大区域 共享,,,,,因而根据分代又分为新生代(eden区幸存from to区)老年代永生代(元空间取代1.8)

栈区（虚拟栈）:线程状态 局部变量（基础类型，引用类型只存引用，指向存在堆上） 操作数 返回地址等 私有区域

程序计数器: 记录程序执行行数 每个线程都有一个 私有区域

本地方法栈: 本地方法 nativemethod

共享：堆 方法区/元空间

私有：虚拟栈 本地方法栈 计数器

Jvm 编译运行 + 解释运行 混合使用 有JIT 即时编译器 interpreter解释器进行运行

编译执行需要生成编译后的机器码文件，而解释执行时逐句解释执行.

二次编译 : 前端编译(javac) .java->.class 后端编译: 字节码->机器码(jit)

java程序--(编译javac)-->字节码文件.class-->类装载子系统化身为反射类Class--->运行时数据区--->(解释执行+JIT编译器编译)-->操作系统(Win，Linux，Mac JVM)

永久代就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。 也就是说，永久代是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久代这一说法。

整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制 -XX: metaspaceSize设置

新生代15岁晋升老年代 -XX:MaxTenuringThreshold

JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池

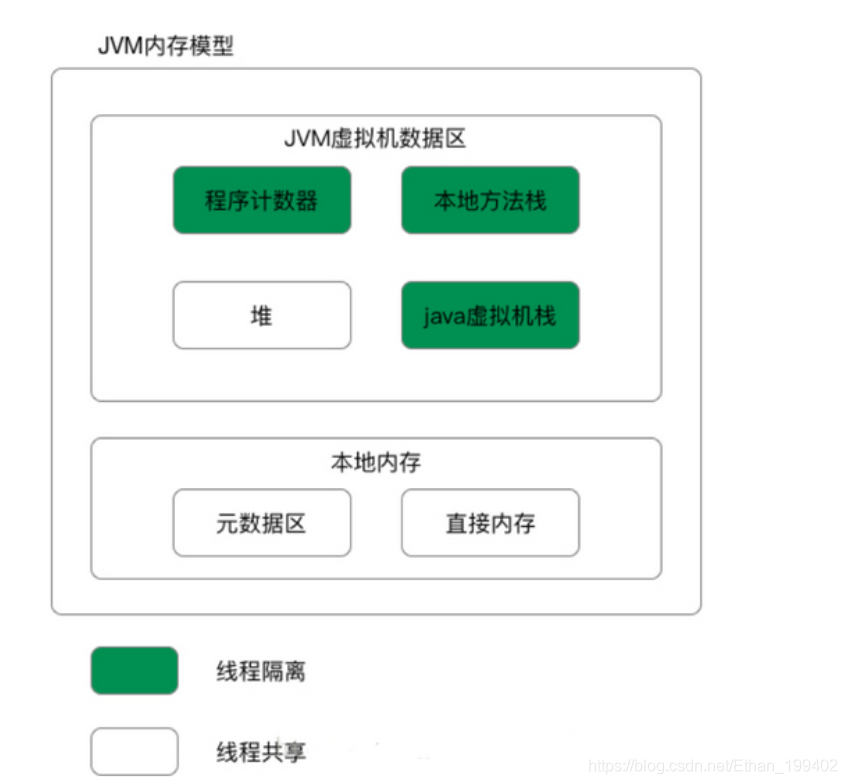
类加载检查 内存分配 初始化零值 设置对象头 执行init方法

对象内存：对象头（自身运行数据（哈希码 gc年代 锁状态）类型指针（指向元数据类））

对象定位：使用句柄池（reference -句柄池-具体实例地址） 使用直接指针(-直接到具体地址),,,,使用句柄的好处是对象移动时reference本身不用变,直接指针的好处是速度快

java 基本类型的包装类的大部分都实现了常量池技术，即 Byte,Short,Integer,Long,Character,Boolean；这 5 种包装类默认创建了数值[-128，127] 的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象,而String常量池双引号申明的会直接只用池中数据,没有时会新建.

堆外内存



使用堆外内存，就是为了能直接分配和释放内存

关于堆内内存以及相应的内存回收策略，在之前的文章《JVM内存管理和垃圾回收》、《JVM垃圾回收器、内存分配与回收策略》中已有介绍。对于堆外内存是指分配的对象直接受操作系统管理的JVM内存堆以外的空间。同时因为这部分区域直接受操作系统的管理，别的进程可以直接通过操作系统对其进行访问，减少了从JVM中复制数据的过程。

堆外内存适用生命周期较长的对象，具有以下特点：

可以很方便的自主开辟很大的内存空间，对于大内存有良好的伸缩性

减少垃圾回收带来的系统停顿时间

在进程间可以共享对象，减少JVM间的复制过程

适合那些分配次数少，读写操作频繁的场景

但也存在如下缺点：

容易出现内存泄漏，并且很难排查

堆外内存的数据结构不直观，当存储结构复杂的对象时，会浪费大量的时间对其进行串行化

Unsafe和NIO包下ByteBuffer。

也就是说NIO直接内存的回收，需要依赖于System.gc()。如果我们的应用中使用了java nio中的direct memory，那么使用-XX:+DisableExplicitGC一定要小心，存在潜在的内存泄露风险。

Full GC一般发生在年老代垃圾回收或者代码调用System.gc的时候，依靠年老代垃圾回收触发 Full GC，进而实现堆外内存的回收显然具有太大的不确定性。如果年老代一直不进行垃圾回收，那么堆外内存就得不到回收，机器的物理内存可能就会被慢慢耗光。为了避免这种情况发生，可以通过参数-XX:MaxDirectMemorySize来指定最大的直接内存大小，当其使用达到了阈值的时候将调用System.gc来做一次Full GC，从而完成可控的堆外内存回收。这样做的问题在于，堆外内存的回收依赖于代码调用 System.gc，先捕获到异常，再在Catch块里面通过System.gc()命 令来触发垃圾收集。但如果Java虚拟机再打开了-XX：+DisableExplicitGC开关，禁止了人工触发垃圾 收集的话，根本不会触发Full GC，这样在使用Netty等 NIO 框架时需注意是否会因为这个参数导致直接内存的泄露

————————————————

版权声明：本文为CSDN博主「Ethan\_199402」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/Ethan_199402/article/details/110431404>

5.2.1 堆内存泄漏

Top jstack jps

通过 jmap -dump:format=b,file=xx pid 命令 Dump 内存堆栈，然后使用 MemoryAnalyzer 工具对内存占用进行分析，查找内存泄漏点，然后结合代码进行分析，定位内存泄漏的具体原因，示例如下所示：

5.2.2 堆外内存泄漏

建议策略如下：

排查下业务代码，看使用堆外内存的地方是否存在忘记释放问题。

如果使用到了 Netty 的 TLS/SSL/openssl，建议到 Netty 社区查下 BUG 列表，看是否是 Netty 老版本已知的 BUG，此类 BUG 通过升级 Netty 版本可以解决。

如果上述两个步骤排查没有结果，则可以通过 google-perftools 工具协助进行堆外内存分析。

————————————————

版权声明：本文为CSDN博主「跨过山河大海」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qq_33814088/article/details/88225997>

Hashmap 1.7 .18区别 红黑树为什么快 hash如何均匀分布

Put、get操作 链表最坏情况on 红黑树olgn

经典常见的平衡二叉搜索树是 AVL 树和红黑树。

平衡是为了防止二叉树退化成链表 （当我们对一棵平衡二叉搜索树进行插入、删除的时候，很可能会让这棵树变得失衡(最坏可能导致所有祖先结点失衡，但是父结点和非祖先结点都不可能失衡为了达到平衡，需要对树进行旋转。而红黑树能够达到自平衡，靠的也就是左旋、右旋和变色。）

AVL自平衡二叉树

红黑树 不追求"完全平衡"，，，因而insert delete效率高，而更平衡的avl树则查询效率高

B树 多路二叉树

B+树 非叶子结点不存储关键信息,故每个块能存储更多的叶子结点,,,叶子结点件有链表,适用于范围查找.

扩容后位置计算不一致:1.7直接使用hash值与需要进行的二进制数进行& 1.8扩容前位置+扩容大小 1.7头插法 1.8尾插法

http 数据存储，rest restful存在哪个区域 怎么传输 netty rpc

http 基于http协议（应用层） rpc基于tcpip协议（传输层）

流行rpc框架 grpc thrift dubbo

客户端（Client），服务的调用方。

服务端（Server），真正的服务提供者。

客户端存根，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方。

服务端存根，接收客户端发送过来的消息，将消息解包，并调用本地的方法。

要求双端使用相同技术

五层协议结构里面是没有表示层和会话层的。应该说它们和应用层合并了

一般来说，RPC 服务主要是针对大型企业的，而 HTTP 服务主要是针对小企业的，因为 RPC 效率更高，而 HTTP 服务开发迭代会更快。RPC并没有规定数据传输格式，这个格式可以任意指定，不同的RPC协议，数据格式不一定相同。Http中还定义了资源定位的路径，RPC中并不需要。最重要的一点：RPC需要满足像调用本地服务一样调用远程服务，也就是对调用过程在API层面进行封装。Http协议没有这样的要求，因此请求、响应等细节需要我们自己去实现。，

1)Request

　　　　Method(get,post) ---请求方式

　　　　URL-------请求地址

　　　　Header------请求头

　　　　Body--------请求体

　　2)Response

　　　　Status Code-------状态码

　　　　Header--------响应头

　　　　Body--------响应体

Rest特点：

1. 使用客户/服务器模型。客户和服务器之间通过一个统一的接口来互相通讯。
2. 层次化的系统。在一个REST系统中，客户端并不会固定地与一个服务器打交道。
3. 无状态。在一个REST系统中，服务端并不会保存有关客户的任何状态。也就是说，客户端自身负责用户状态的维持，并在每次发送请求时都需要提供足够的信息。
4. 可缓存。REST系统需要能够恰当地缓存请求，以尽量减少服务端和客户端之间的信息传输，以提高性能。
5. 统一的接口。一个REST系统需要使用一个统一的接口来完成子系统之间以及服务与用户之间的交互。这使得REST系统中的各个子系统可以独自完成演化。

https = http + ssl/tsl

**TCP传输的三次握手、四次挥手策略**

**Tcp基于连接传输 UDP面向非连接**

正向代理即是客户端代理, 代理客户端, 服务端不知道实际发起请求的客户端.

反向代理即是服务端代理, 代理服务端, 客户端不知道实际提供服务的服务端

Rest:表现层状态转移，是一种软件风格，不是标准，URL定位资源，HTTP动词（GET，POST，PUT，DELETE）描述操作，rest构建的API就是restful风格。像修改删除rest风格使用的是post操作，而restful风格是用put delete操作。

服务器处理相应主要有两种设计模式，线程驱动（传统同步阻塞IO模型，tomcat） 事件驱动）（维护一个事件池，监听事件）

JDK 已经给我们提供了 NIO 的包，也是使用了 Reactor 模型来实现的异步非阻塞模式，那为什么还要使用netty，实际上大家不使用的原因是因为它太难控制

编写一个 Socket 通信都有哪些步骤呢？

* 1. 创建一个 ServerSocket，监听并绑定一个端口。

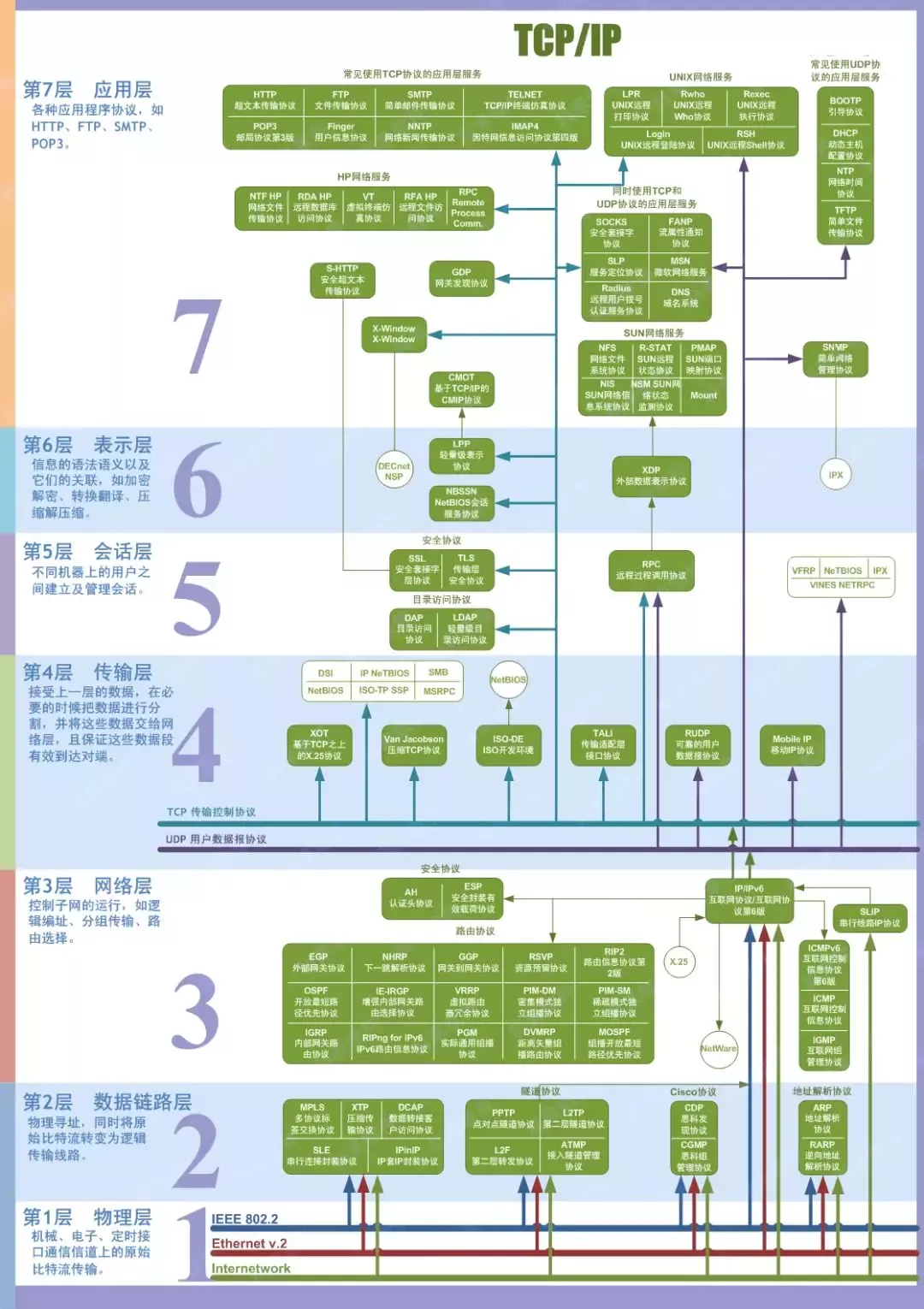
一系列客户端来请求这个端口。

服务器使用 Accept，获得一个来自客户端的 Socket 连接对象。

启动一个新线程处理连接：①读 Socket，得到字节流；②解码协议，得到 HTTP 请求对象；③处理 HTTP 请求，得到一个结果，封装成一个 HTTPResponse 对象；④编码协议，将结果序列化字节流；④写 Socket，将字节流发给客户端。

继续循环步骤 3。

总体来讲channel就是双向管道处理各种io缓冲区中buffer流，然后注册到selector上，受selector管理，



多线程几种实现方式 3

四种

* 1. 继承Thread类，重写run方法  
     2.实现Runnable接口，重写run方法，实现Runnable接口的实现类的实例对象作为Thread构造函数的target  
     3.通过Callable和FutureTask创建线程  
     4.通过线程池创建线程
* 1.Runnable接口的话，可以避免单继承的局限性，具有较强的健壮性。
* 2.Runnable可以实现资源的共享，同时处理同一资源。
* 3.Thread类的线程间都是独立运行的，资源不共享。
* 4.继承Thread类不再被其他类继承（java不存在多继承）

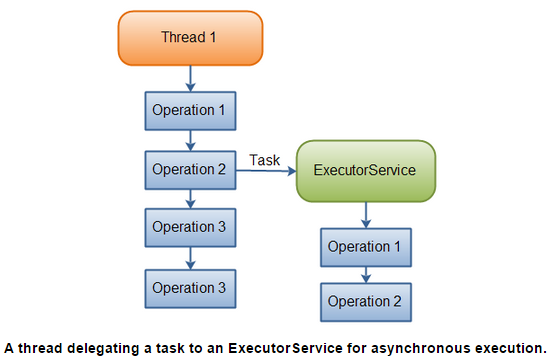
t．jion方法，，，主线程main函数等待子线程结果后再继续执行，原理如下所示：

Join方法实现是通过wait（小提示：Object 提供的方法）。 当main线程调用t.join时候，main线程会获得线程对象t的锁（wait 意味着拿到该对象的锁),调用该对象的wait(等待时间)，直到该对象唤醒main线程 ，比如退出后。这就意味着main 线程调用t.join时，必须能够拿到线程t对象的锁。

ExecutorService是一个接口，Executors是一个类，里面简化了ThreadPoolExecutor中的参数进行初始化，返回接口类型。

其中ExecutorService中有多个接口方法，submit返回future类型，execute不返回只接受Runnable类型，，，，还有一些终止方法等

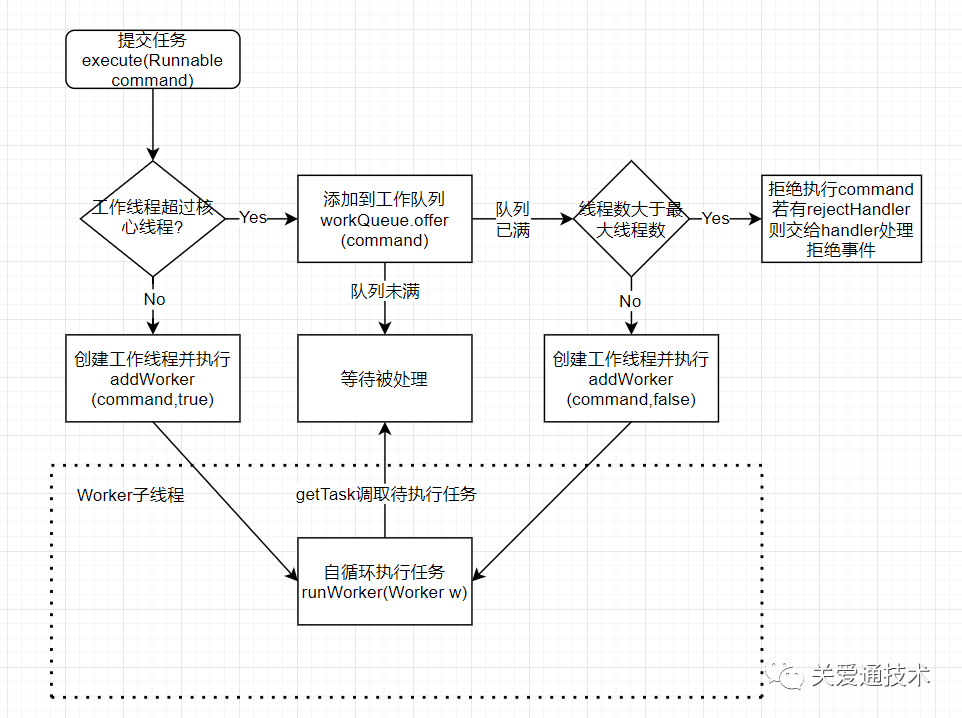
ThreadPoolExecutor是ExecutorService的一个实现类，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。



Runnable Callable均是函数式接口，均有注解@FunctionalInterface，故只有一个抽象方法，（Comparator也是），**@FunctionalInterface标记在接口上，“函数式接口”是指仅仅只包含一个抽象方法的接口**。JDK8接口中的[静态方法和默认方法](http://blog.csdn.net/aitangyong/article/details/54134385)，都不算是抽象方法。

Future也是接口

线程池实现的几种方式，拒绝测试有几种具体实现，直接拒绝



1、int corePoolSize 核心线程数  
2、int maximumPoolSize 最大线程数  
3、long keepAliveTime + TimeUnit unit 空闲线程的存活时间  
4、BlockingQueue<Runnable> workQueue 任务队列  
5、ThreadFactory threadFactory 线程工厂，用来创建新线程  
6、RejectedExecutionHandler handler 处理被拒绝的任务

7、TimeUnit 等待时间单位

workQueue

LikedBlockingQueue 无界缓存队列

SynchronousQueue 无缓冲队列

ArrayBlockingQueue 有界缓存队列

DelayedWorkQueue 延时队列

RejectExecutionHandler 官方有四种，也可自定义策略

CallerRunsPolicy 调用者运行策略 ---交由提交者处理任务，具有负反馈性质（占中提交者时间，避免连续提交新线程）

AbortPolicy 终止策略 直接抛出异常

DiscardPolicy 丢弃策略

DiscardOldestPolicy 弃老策略 弹出队列头元素

•newFixedThreadPool：该方法返回一个固定数量的线程池，线程数不变，当有一个任务提交时，若线程池中空闲，则立即执行，若没有，则会被暂缓在一个任务队列中，等待有空闲的线程去执行。new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>()) 多个核心线程并且由于使用的是LinkedBlockingQueue，在资源有限的时候容易引起OOM异常

•newSingleThreadExecutor: 创建一个线程的线程池，若空闲则执行，若没有空闲线程则暂缓在任务队列中。new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>())) 因为LinkedBlockingQueue是长度为Integer.MAX\_VALUE的队列，可以认为是无界队列，因此往队列中可以插入无限多的任务，在资源有限的时候容易引起OOM异常

•newCachedThreadPool：new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>());最终会创建非核心线程来执行任务。对于非核心线程空闲60s时将被回收。无界队列OMM

返回一个可根据实际情况调整线程个数的线程池，不限制最大线程数量，若用空闲的线程则执行任务，若无任务则不创建线程。并且每一个空闲线程会在 60 秒后自动回收。

•newScheduledThreadPool: 创建一个可以指定线程的数量的线程池，但是这个线程池还带有延迟和周期性执行任务的功能，类似定时器 (corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS, new DelayedWorkQueue()) 也容易OMM

数据库 聚簇 非聚簇 存储结构B+数为什么这么快 底层结构 查询次数 树的层数大于4层 怎么优化 left join 走子集

核心原因是受限于磁盘i/o读取速度。mysql一般用于存储比较大的数据，使用的都是机械硬盘。机械硬盘一次数据读取的时间是毫秒级的，和内存读取远远不在一个量级。

如果使用二叉树这种多层级结构，会导致磁盘的多次读取，每读取下一层数据，都是一次磁盘重新寻址。所以B树，B+树 这种多叉树的优势有体现出来了。一个4层的B+树，基本能覆盖上亿数据的查找。

减少IO---B+跟B树不同B+树的非叶子节点不保存关键字记录的指针，只进行数据索引，这样使得B+树每个非叶子节点所能保存的关键字大大增加；这个也就是为什么说查找单个节点，B+树查询会更稳定，B树查询可能会更快的原因。

范围查找---而B+树这种数据结构能更好满足这种范围查询输出

join，其实就是“inner join”，为了简写才写成join

<https://www.cnblogs.com/wxmdevelop/p/13803298.html>

join on and 不会过滤结果记录条数，只会根据and后的条件是否显示 B表的记录，A表的记录一定会显示。1、 on条件是在生成临时表时使用的条件，它不管on中的条件是否为真，都会返回左边表中的记录。2、where条件是在临时表生成好后，再对临时表进行过滤的条件。这时已经没有left join的含义（必须返回左边表的记录）了，条件不为真的就全部过滤掉。

left join的目的无非是把两个模型上的数据合并到一个模型上，这是需要运算的，数据量越大运算量越大，更何况，数据库在left join的设计上，表现并不理想。

那么如何优化left join：

1、条件中尽量能够过滤一些行将驱动表变得小一点，用小表去驱动大表 （where）

2、右表的条件列一定要加上索引（主键、唯一索引、前缀索引等），最好能够使type达到range及以上（ref,eq\_ref,const,system）

Binlog日志有几种模式 按行级 按执行命令级 混合级

RowLevel 行模式 每一行数据变化都记录避免存储过程或函数无法正确复制问题，日志超大

StateMentLevel 默认 记录sql语句，互联网中也常用

Mixed 自动混合模式 根据情况选择使用哪一种 ，，，update delete 使用行模式，上面避免的可以避免

SpringBoot enableaotuconfiguration metainfo信息加载后怎么注入bean

<https://blog.csdn.net/zxc123e/article/details/80222967>

@SpringBootApplication 中三个重要注解

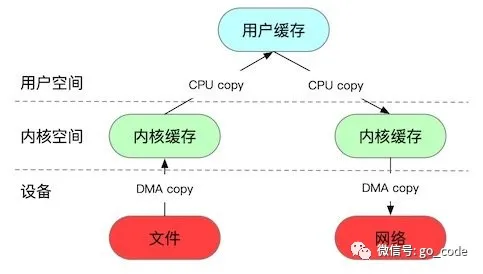
@Configuration（@SpringBootConfiguration点开查看发现里面还是应用了@Configuration）----其用于定义配置类，可替换xml配置文件，被注解的类内部包含有一个或多个被@Bean注解的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义，初始化Spring容器。

@ComponentScan -----在配置类上添加 @ComponentScan 注解。该注解默认会扫描该类所在的包下所有的配置类，作用就是根据定义的扫描路径，把符合扫描规则的类装配到spring容器中，注解定义如下。相当于之前的 <context:component-scan>

@EnableAutoConfiguration -----可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器,其中里面最关键的要属@Import(AutoConfigurationImportSelector.class)，其通过 SpringFactoriesLoader.loadFactoryNames()把 spring-boot-autoconfigure.jar/META-INF/spring.factories中每一个xxxAutoConfiguration文件都加载到容器中，SpringFactoriesLoader是一个抽象类，类中定义的静态属性定义了其加载资源的路径public static final String FACTORIES\_RESOURCE\_LOCATION = "META-INF/spring.factories" ，里面的三个方法作用依次是loadFactories方法首先获取类加载器，然后调用loadFactoryNames方法获取所有的指定资源的名称集合、接着调用instantiateFactory方法实例化这些资源类并将其添加到result集合中。最后调用AnnotationAwareOrderComparator.sort方法进行集合的排序。

零拷贝

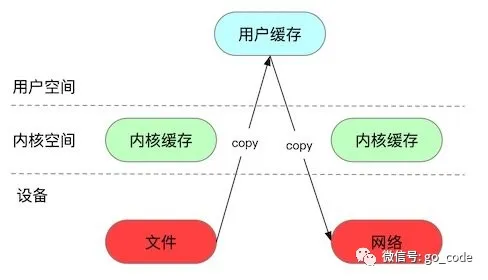
传统IO过程要有四次拷贝四次上下文切换



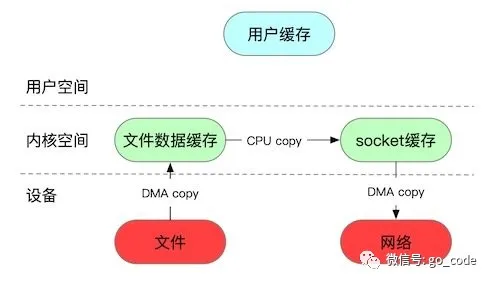
DMA（Direct Memory Access）：直接存储器访问。DMA是一种无需CPU的参与，让外设和系统内存之间进行双向数据传输的硬件机制。使用DMA可以使系统CPU从实际的I/O数据传输过程中摆脱出来，从而大大提高系统的吞吐率。硬件技术

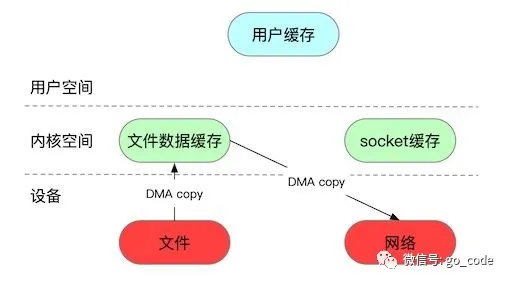
零拷贝并不是完全取消掉拷贝只是减少拷贝，主要有三种形式：

* 直接I/O：数据直接跨过内核，在用户地址空间与I/O设备之间传递，内核只是进行必要的虚拟存储配置等辅助工作；如Linux中的open函数



* 避免内核和用户空间之间的数据拷贝：当应用程序不需要对数据进行访问时，则可以避免将数据从内核空间拷贝到用户空间；如mmap，通过内存映射的机制，把内核中的部分内存空间映射到用户空间的内存，用户空间和内核空间共享一块相同的物理内存，从而提供用户进程对内存直接访问的能力。有了mmap的支持，数据从文件中读取到内核空间之后，就不会再拷贝到用户空间， 但是上下文切换照样。



，，sendfile也是如此，其更厉害，优化之后“socket缓存”中只存储要发送的数据在“文件数据缓存”中的位置和偏移量，在实际发送时，根据位置和偏移量直接将“文件数据缓存”中的数据拷贝到网卡设备中，又省掉了一次拷贝操作。

Sockmap更牛逼了，直接socket到sockt数据转发。使用场景比较特殊。

* copy on write：写时拷贝技术，数据不需要提前拷贝，而是当需要修改的时候再进行部分拷贝。

直接内存操作的DMA模块有声卡 网卡 磁盘 显卡等

Kafka-----Java实现的零拷贝：则是通过java.nio.channels.FileChannel中的transferTo方法来实现的。transferTo方法底层是基于操作系统的sendfile这个system call来实现的（不再需要拷贝到用户态了）

不要慌 有个过程从数据结构 jvm 网路传输 多线程 spring套装 中间件 数据库 分布式解决方案等，，，，有几个答的好就行，刷题一定要。

### 每日优鲜

教育项目描述 ES 图库同步情况，异常处理 事务 写入tps

Mysql 底层索引结构

MVCC(Multi-Version Concurrency Control)多版本并发控制 (印象笔记)

MVCC是为了实现事务的隔离性，通过版本号，避免同一数据在不同事务间的竞争，可以把它当成基于多版本号的一种乐观锁。当然，这种乐观锁只在事务级别提交读和可重复读有效。READ UNCOMMITTED 总是读取最新的数据行, 而不是符合当前事务版本的数据行。而 SERIALIZABLE 则会对所有读取的行都加锁。

MVCC的实现依赖于：**三个隐藏字段**，**Read View**、**Undo log** 来实现。

**三个隐式字段**：rowId（DB\_ROW\_ID） 事务号（DB\_TRX\_ID） 回滚指针(DB\_ROLL\_PY)

DB\_TRX\_ID（6字节）

DB\_TRX\_ID表示最近一次对本行记录行做修改（insert|update|delete）的事务ID。这儿注意，delete操作，innodb也认为是一个update操作，会更新一个删除为delete\_bit，将行表示为deleted，并非真正的删除。

DB\_ROLL\_PTR（7字节）

DB\_ROLL\_PTR表示回滚指针，指向当前记录行的undo log信息。

DB\_ROW\_ID（6字节）

当表没有设置主键或唯一非空索引时，innodb便会使用这个row\_id自动生成聚簇索引。如果表有主键或唯一索引，聚簇索引就不会包含这个字段，所以这个字段跟MVCC关系不大。

**Read View**即读视图，和快照、snapshot是一个意思.。里面有几个字段low\_limit\_id，目前出现过的最大的事务ID+1，即下一个将被分配的事务ID。 up\_limit\_id：活跃事务列表trx\_ids中最小的事务ID，如果trx\_ids为空，则up\_limit\_id 为 low\_limit\_id。trx\_ids：Read View创建时其他未提交的活跃事务ID列表。意思就是创建Read View时，将当前未提交事务ID记录下来，后续即使它们修改了记录行的值，对于当前事务也是不可见的。creator\_trx\_id：当前创建事务的ID，是一个递增的编号。这个编号并不是数据行里的DB\_ROW\_ID。

**Undo log**中存储的是老版本数据，当一个事务需要读取记录行时，如果当前记录行不可见，可以顺着undo log链找到满足其可见性条件的记录行版本。

当插入的是一条新数据时，记录上对应的回滚段指针为NULL 更新记录时，原记录将被放入到undo表空间中，并通过DB\_ROLL\_PT指向该记录。

作用：MVCC使得数据库读不会对数据加锁，select不会加锁，提高了数据库的并发处理能力；借助MVCC，数据库可以实现RC，RR等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I-隔离性。

TCP三次握手

第一次握手

客户主动（active open）去connect服务器，并且发送SYN 假设序列号为J,

服务器是被动打开(passive open)

第二次握手

服务器在收到SYN后，它会发送一个SYN以及一个ACK（应答）给客户，

ACK的序列号是 J+1表示是给SYN J的应答，新发送的SYN K 序列号是K

第三次握手

客户在收到新SYN K, ACK J+1 后，也回应ACK K+1 以表示收到了，

然后两边就可以开始数据发送数据了

————————————————

版权声明：本文为CSDN博主「jun2016425」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/jun2016425/article/details/81506353

规划 目标

### 华泰证券

二级缓存区别，guava缓存多实例，数据一致性问题（多节点访问）

本地缓存 远程缓存 网络带宽 距离用户越近的地方缓存越快

1惰性加载**结合**消息机制来更新缓存数据，也就是：当导购服务的配置发生变化时，通知业务网关重新拉取数据，更新缓存。

2适当调大 LoadigCache 的线程池参数，并在线程池埋点，监控线程池的使用情况，当线程繁忙时能发出告警，然后动态修改线程池参数。

Redis 雪崩 击穿 穿透 解决方案 分布式锁优缺点ZK

击穿：**单个Key**在缓存中查不到而去数据库中查，在高并发的情况下造成数据库崩溃。主要愿意是并发访问。

解决方案：1、通过synchronized+双重检查机制：某个key只让一个线程查询，阻塞其它线程出查询，切点阻塞

* 1. 设置value永不过期，定时刷新value值（可以接受的情况下）
  2. 互斥锁（常用），原理与1相同，也像是分布式锁，利用setnx设置获取锁，然后回设缓存。

雪崩：**多个key**并发查询，缓存中不存在，而去查数据库导致崩溃的现象。主要原因是出现原因: 1 key同时失效2 redis本身崩溃了

解决方案：1在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。(跟击穿的第一个方案类似，但是这样是避免不了其它key去查数据库，只能减少查询的次数)2可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存3不同的key，设置不同的过期时间，具体值可以根据业务决定，让缓存失效的时间点尽量均匀4做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。(这种方式复杂点)

穿透： 一般是出现这种情况是因为恶意频繁查询才会对系统造成很大的问题: key缓存并且**数据库不存在**，所以每次查询都会查询数据库从而导致数据库崩溃。

解决方案：1布隆过滤器2将击透的key缓存起来，但是时间不能太长，下次进来是直接返回不存在，但是这种情况无法过滤掉动态的key，就是说每次请求进来都是不同额key，这样还是会造成这个问题

DMQ使用场景 为什么这样用直接使用接口可以吗 Kafak 为什么快， 一致性、可靠性、多线程消费partition 有序性

不用接口是为了解耦----减少一个系统的修改另一个调用系统或被调用系统也跟着修改的局面 还有可以缓冲

Kafka特点： 异步解耦 系统缓冲 削峰作用 数据流处理

**一致性**：一致性主要是说不论是老的 Leader 还是新选举的 Leader，Consumer 都能读到一样的数据1. 只有 High Water Mark 以上的消息才支持 Consumer 读取，而 High Water Mark 取决于 ISR 列表里面偏移量最小的分区，对应于上图的副本2，这个很类似于木桶原理。

**可靠性**：1. Kafka 的分区多副本架构是 Kafka 可靠性保证的核心，把消息写入多个副本可以使 Kafka 在发生崩溃时仍能保证消息的持久性。

2.product 往broke 消息确认机制 ack 0消息发送出去 1leader收到消息并写入分区 all所有同步副本都会收到消息 ；同步 异步，异步高效 同步可靠

3. 当 Leader 挂掉了，而且 unclean.leader.election.enable=false 的情况下，Kafka 会从 ISR 列表中选择第一个 follower 作为新的 Leader，因为这个分区拥有最新的已经 committed 的消息。

**有效性**：1.同一partition中消息只能别同一消费者组中的一个实例消费

2. 为了保证一个消费者中多个线程去处理时，不会使得消息的顺序被打乱，则可以在消费者中，消息分发至不同的线程时，**加一个队列，**消费者去做hash分发，将需要放在一起的数据，分发至同一个队列中，最后多个线程从队列中取数据。写 N 个内存 queue，具有相同 key 的数据都分发到同一个内存 queue；然后对于 N 个线程，每个线程分别消费一个内存 queue 即可，这样就能保证顺序性。

分布式 优化方案 高并发设计

微服务优缺点 内部构造 注册中心 Springcloud 组成

Spring生命周期 数据预加载方式

Rpc 方式 底层使用框架 netty吗

数据库多实例访问 多session管理，底层索引

Java hashMap->安全 JUC包函数（3）锁 线程池

拦截器 过滤器区别

Limit数据分页变慢后处理方法

# 第五节：面试官

[**https://blog.csdn.net/weixin\_36380516/article/details/114811866**](https://blog.csdn.net/weixin_36380516/article/details/114811866)