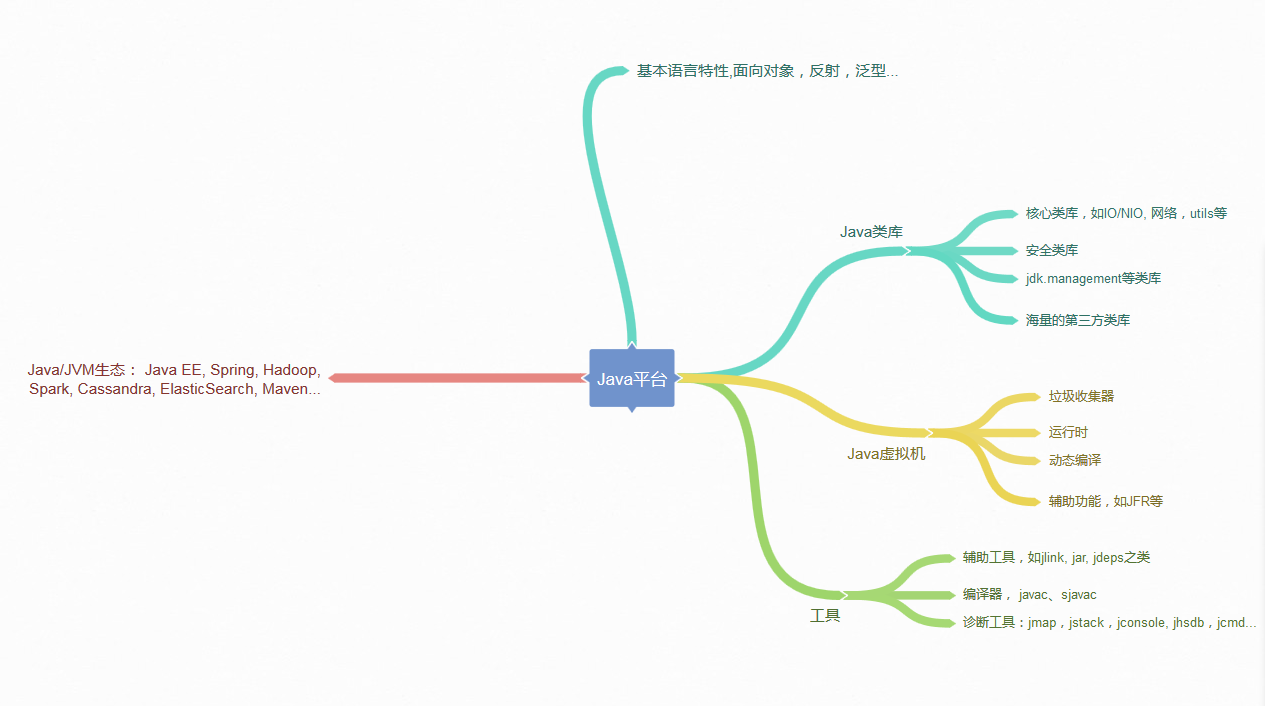
**【总结】Java核心技术36讲知识点大纲**

https://blog.csdn.net/sinat\_25295611/article/details/81592322

前段时间在极客时间上购买了杨晓峰老师的《Java核心技术36讲》

**Java核心技术36讲知识点总结大纲**

**1 Java平台的理解，**“Java 是解释执行”，这句话正确吗？



Java的特性，解释运行和编译运行

Java特性:

面向对象（封装，继承，多态）

平台无关性（JVM运行.class文件） 书写一次，到处运行”（Write once, run anywhere）

语言（泛型，Lambda）

类库（集合，并发，网络，IO/NIO）

JRE（Java运行环境，JVM，类库）

JDK（Java开发工具，包括JRE，javac，诊断工具）

Java 语言特性，包括泛型、Lambda 等语言特性；基础类库，包括集合、IO/NIO、网络、并发、安全等基础类库

Java是解析运行吗？

不正确！

1，Java源代码经过Javac编译成.class文件

2，.class文件经JVM解析或编译运行。

（1）解析:.class文件经过JVM内嵌的解析器解析执行。

（2）编译:存在JIT编译器（Just In Time Compile 即时编译器）把经常运行的代码作为"热点代码"编译与本地平台相关的机器码，并进行各种层次的优化。

（3）AOT编译器: Java 9提供的直接将所有代码编译成机器码执行。

类加载大致过程：加载、验证、链接、初始化

**2 Exception 和 Error 有什么区别**

1. 理解Java的异常体系的设计，Throwable ,Exception,Error 的关系
2. 理解ClassNotFoundException 与NoClassDefFoundError的区别
3. 遵循 Throw early, catch late 原则

**3 final、finally、finalize 有什么不同**

从安全、性能、垃圾收集方面考虑

1。final修饰的类，不可被继承，修饰的方法不可被重写，修饰的变量不可多次赋值。通过final能够得到性能上的优化，但是不明显，如果大量使用可能会干扰代码，不能表达出本来具有的含义。故不使用。匿名内部类，访问局部变量要求传入的参数，必须是final是要保证数据一致性问题。

2。finally。代码中总是会执行的代码段。除了退出虚拟机外。

3。finalize。在虚拟机回收改对象前进行调用。此种方式不可取。因为java虚拟机不知道在什么时候才对对象进行回收。

final修饰类，变量，方法等；目的是不可变，这样做使得安全、提高性能

copy-on-write 原则：

链接：http://www.cnblogs.com/hapjin/p/4840107.html

String类为什么是final的：

1、从设计安全)上讲，

1)、确保它们不会在子类中改变语义。String类是final类，这意味着不允许任何人定义String的子类。

换言之，

如果有一个String的引用，它引用的一定是一个String对象，而不可能是其他类的对象。

2)、String 一旦被创建是不能被修改的，

因为 java 设计者将 String 为可以共享的

2、从效率上讲：

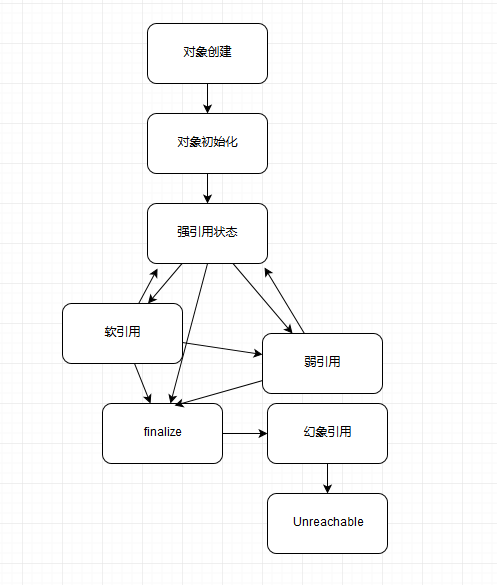
1)、设计成final，JVM才不用对相关方法在虚函数表中查询，而直接定位到String类的相关方法上，提高了执行效率。

2)、Java设计者认为共享带来的效率更高。

总而言之，就是要保证 java.lang.String 引用的对象一定是 java.lang.String的对象，而不是引用它的子孙类，这样才能保证它的效率和安全。

**4 强引用、软引用、弱引用、幻象引用有什么区别**

对象的可达性状态：强引用 -> 软引用 -> 弱引用 -(finalize)->幻想引用





1. 对象可达性状态的流转
2. 引用队列（ReferenceQueue）的使用
3. 诊断 JVM 引用情况
4. Reachability Fence 通过底层 API 来达到强引用的效果

牛人总结：

在Java语言中，除了基本数据类型外，其他的都是指向各类对象的对象引用；Java中根据其生命周期的长短，将引用分为4类。

1 强引用

特点：我们平常典型编码Object obj = new Object()中的obj就是强引用。通过关键字new创建的对象所关联的引用就是强引用。 当JVM内存空间不足，JVM宁愿抛出OutOfMemoryError运行时错误（OOM），使程序异常终止，也不会靠随意回收具有强引用的“存活”对象来解决内存不足的问题。对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应（强）引用赋值为 null，就是可以被垃圾收集的了，具体回收时机还是要看垃圾收集策略。

2 软引用

特点：软引用通过SoftReference类实现。 软引用的生命周期比强引用短一些。只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象：即JVM 会确保在抛出 OutOfMemoryError 之前，清理软引用指向的对象。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。后续，我们可以调用ReferenceQueue的poll()方法来检查是否有它所关心的对象被回收。如果队列为空，将返回一个null,否则该方法返回队列中前面的一个Reference对象。

应用场景：软引用通常用来实现内存敏感的缓存。如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

3 弱引用

弱引用通过WeakReference类实现。 弱引用的生命周期比软引用短。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。由于垃圾回收器是一个优先级很低的线程，因此不一定会很快回收弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

应用场景：弱应用同样可用于内存敏感的缓存。

4 虚引用

特点：虚引用也叫幻象引用，通过PhantomReference类来实现。无法通过虚引用访问对象的任何属性或函数。幻象引用仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

ReferenceQueue queue = new ReferenceQueue ();

PhantomReference pr = new PhantomReference (object, queue);

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取一些程序行动。

应用场景：可用来跟踪对象被垃圾回收器回收的活动，当一个虚引用关联的对象被垃圾收集器回收之前会收到一条系统通知。

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. 强引用：项目中到处都是。

2. 软引用：图片缓存框架中，“内存缓存”中的图片是以这种引用来保存，使得JVM在发生OOM之前，可以回收这部分缓存

3. 虚引用：在静态内部类中，经常会使用虚引用。例如，一个类发送网络请求，承担callback的静态内部类，则常以虚引用的方式来保存外部类(宿主类)的引用，当外部类需要被JVM回收时，不会因为网络请求没有及时回来，导致外部类不能被回收，引起内存泄漏

4. 幽灵引用：这种引用的get()方法返回总是null，所以，可以想象，在平常的项目开发肯定用的少。但是根据这种引用的特点，我想可以通过监控这类引用，来进行一些垃圾清理的动作。

**5 String、StringBuffer、StringBuilder有什么区别**

1. 字符串的设计与实现考量
2. 字符串缓存 intern()方法，由永久代移到堆中。
3. String 的演化，Java 9 中底层把 char 数组换成了 byte 数组，占用更少的空间
4. getBytes()方法指定字符编码

1 String

(1) String的创建机理

由于String在Java世界中使用过于频繁，Java为了避免在一个系统中产生大量的String对象，引入了字符串常量池。其运行机制是：创建一个字符串时，首先检查池中是否有值相同的字符串对象，如果有则不需要创建直接从池中刚查找到的对象引用；如果没有则新建字符串对象，返回对象引用，并且将新创建的对象放入池中。但是，通过new方法创建的String对象是不检查字符串池的，而是直接在堆区或栈区创建一个新的对象，也不会把对象放入池中。上述原则只适用于通过直接量给String对象引用赋值的情况。

举例：String str1 = "123"; //通过直接量赋值方式，放入字符串常量池

String str2 = new String(“123”);//通过new方式赋值方式，不放入字符串常量池

注意：String提供了inter()方法。调用该方法时，如果常量池中包括了一个等于此String对象的字符串（由equals方法确定），则返回池中的字符串。否则，将此String对象添加到池中，并且返回此池中对象的引用。

(2) String的特性

[A] 不可变。是指String对象一旦生成，则不能再对它进行改变。不可变的主要作用在于当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅度提高系统性能。不可变模式是一个可以提高多线程程序的性能，降低多线程程序复杂度的设计模式。

[B] 针对常量池的优化。当2个String对象拥有相同的值时，他们只引用常量池中的同一个拷贝。当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

2 StringBuffer/StringBuilder

StringBuffer和StringBuilder都实现了AbstractStringBuilder抽象类，拥有几乎一致对外提供的调用接口；其底层在内存中的存储方式与String相同，都是以一个有序的字符序列（char类型的数组）进行存储，不同点是StringBuffer/StringBuilder对象的值是可以改变的，并且值改变以后，对象引用不会发生改变;两者对象在构造过程中，首先按照默认大小申请一个字符数组，由于会不断加入新数据，当超过默认大小后，会创建一个更大的数组，并将原先的数组内容复制过来，再丢弃旧的数组。因此，对于较大对象的扩容会涉及大量的内存复制操作，如果能够预先评估大小，可提升性能。

唯一需要注意的是：StringBuffer是线程安全的，但是StringBuilder是线程不安全的。可参看Java标准类库的源代码，StringBuffer类中方法定义前面都会有synchronize关键字。为此，StringBuffer的性能要远低于StringBuilder。

3 应用场景

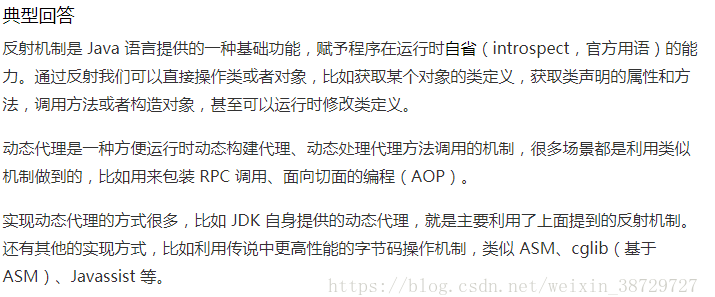
[A]在字符串内容不经常发生变化的业务场景优先使用String类。例如：常量声明、少量的字符串拼接操作等。如果有大量的字符串内容拼接，避免使用String与String之间的“+”操作，因为这样会产生大量无用的中间对象，耗费空间且执行效率低下（新建对象、回收对象花费大量时间）。

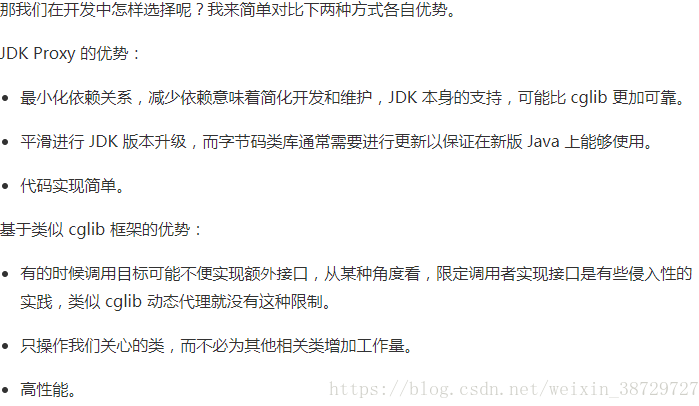
[B]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在多线程环境下，建议使用StringBuffer，例如XML解析、HTTP参数解析与封装。

[C]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在单线程环境下，建议使用StringBuilder，例如SQL语句拼装、JSON封装等。

**6 动态代理是基于什么原理**

1. 反射原理
2. 动态代理分类：JDK动态代理原理，cglib动态代理原理
3. 优缺点，适用场景





代理模式（通过代理静默地解决一些业务无关的问题，比如远程、安全、事务、日志、资源关闭……让应用开发者可以只关心他的业务）

    静态代理：事先写好代理类，可以手工编写，也可以用工具生成。缺点是每个业务类都要对应一个代理类，非常不灵活。

    动态代理：运行时自动生成代理对象。缺点是生成代理代理对象和调用代理方法都要额外花费时间。

        JDK动态代理：基于Java反射机制实现，必须要实现了接口的业务类才能用这种办法生成代理对象。新版本也开始结合ASM机制。

        cglib动态代理：基于ASM机制实现，通过生成业务类的子类作为代理类。

Java 发射机制的常见应用：动态代理（AOP、RPC）、提供第三方开发者扩展能力（Servlet容器，JDBC连接）、第三方组件创建对象（DI）

**7 int和Integer有什么区别**

1. 基本类型和包装类型
2. 包装类型的缓存
3. 适应场景，性能影响

[1] 基本类型均具有取值范围，在大数\*大数的时候，有可能会出现越界的情况。

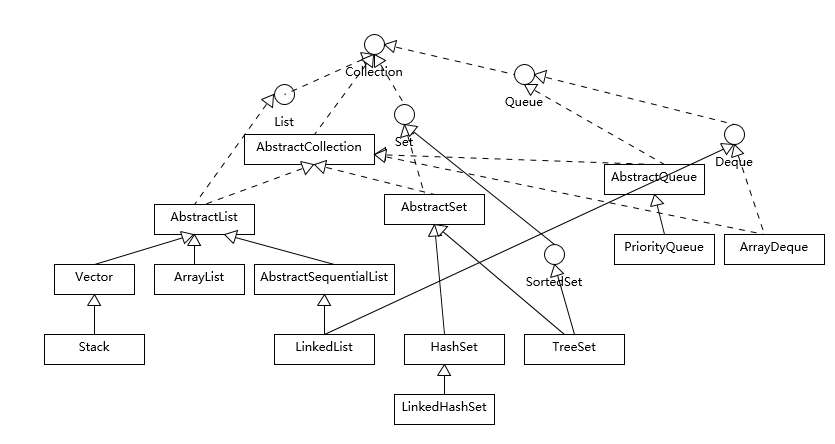
[2] 基本类型转换时，使用声明的方式。例：long result= 1234567890 \* 24 \* 365；结果值一定不会是你所期望的那个值，因为1234567890 \* 24已经超过了int的范围，如果修改为：long result= 1234567890L \* 24 \* 365；就正常了。

[3] 慎用基本类型处理货币存储。如采用double常会带来差距，常采用BigDecimal、整型（如果要精确表示分，可将值扩大100倍转化为整型）解决该问题。

[4] 优先使用基本类型。原则上，建议避免无意中的装箱、拆箱行为，尤其是在性能敏感的场合，

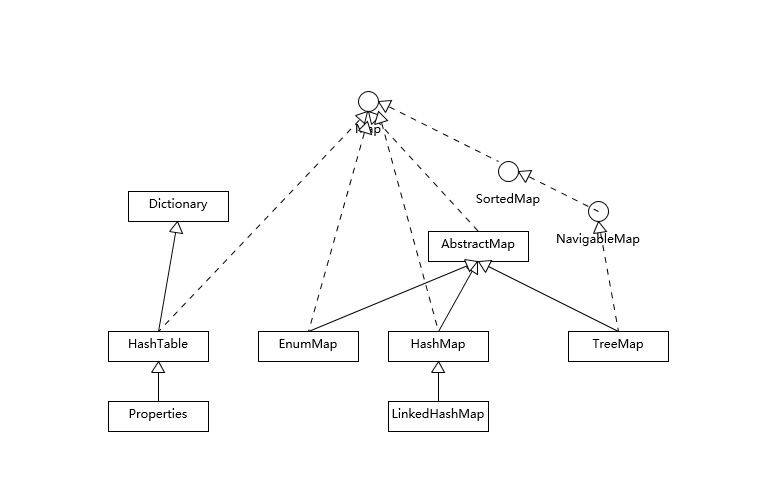
[5] 如果有线程安全的计算需要，建议考虑使用类型AtomicInteger、AtomicLong 这样的线程安全类。部分比较宽的基本数据类型，比如 float、double，甚至不能保证更新操作的原子性，可能出现程序读取到只更新了一半数据位的数值。

**8 对比Vector、ArrayList、LinkedList有何区别**



1. Collection 各种类的实现原理，使用场景
2. List 的各种实现，Set 的各种实现，Queue的各种实现
3. 理解Java提供的各种排序算法 Collections.sort() Arrays.sort()

**9 对比Hashtable、HashMap、TreeMap有什么不同**



1. HashMap 实现原理，源码分析
2. Java 8 HashMap 中的树化
3. 容量和负载因子
4. 解决哈希冲突的方法

三者均实现了Map接口，存储的内容是基于key-value的键值对映射，一个映射不能有重复的键，一个键最多只能映射一个值。

（1） 元素特性

HashTable中的key、value都不能为null；HashMap中的key、value可以为null，很显然只能有一个key为null的键值对，但是允许有多个值为null的键值对；TreeMap中当未实现 Comparator 接口时，key 不可以为null；当实现 Comparator 接口时，若未对null情况进行判断，则key不可以为null，反之亦然。

（2）顺序特性

HashTable、HashMap具有无序特性。TreeMap是利用红黑树来实现的（树中的每个节点的值，都会大于或等于它的左子树种的所有节点的值，并且小于或等于它的右子树中的所有节点的值），实现了SortMap接口，能够对保存的记录根据键进行排序。所以一般需要排序的情况下是选择TreeMap来进行，默认为升序排序方式（深度优先搜索），可自定义实现Comparator接口实现排序方式。

（3）初始化与增长方式

初始化时：HashTable在不指定容量的情况下的默认容量为11，且不要求底层数组的容量一定要为2的整数次幂；HashMap默认容量为16，且要求容量一定为2的整数次幂。

扩容时：Hashtable将容量变为原来的2倍加1；HashMap扩容将容量变为原来的2倍。

（4）线程安全性

HashTable其方法函数都是同步的（采用synchronized修饰），不会出现两个线程同时对数据进行操作的情况，因此保证了线程安全性。也正因为如此，在多线程运行环境下效率表现非常低下。因为当一个线程访问HashTable的同步方法时，其他线程也访问同步方法就会进入阻塞状态。比如当一个线程在添加数据时候，另外一个线程即使执行获取其他数据的操作也必须被阻塞，大大降低了程序的运行效率，在新版本中已被废弃，不推荐使用。

HashMap不支持线程的同步，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。如果需要同步（1）可以用 Collections的synchronizedMap方法；（2）使用ConcurrentHashMap类，相较于HashTable锁住的是对象整体， ConcurrentHashMap基于lock实现锁分段技术。首先将Map存放的数据分成一段一段的存储方式，然后给每一段数据分配一把锁，当一个线程占用锁访问其中一个段的数据时，其他段的数据也能被其他线程访问。ConcurrentHashMap不仅保证了多线程运行环境下的数据访问安全性，而且性能上有长足的提升。

(5)一段话HashMap

HashMap基于哈希思想，实现对数据的读写。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，让后找到bucket位置来储存值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap在每个链表节点中储存键值对对象。当两个不同的键对象的hashcode相同时，它们会储存在同一个bucket位置的链表中，可通过键对象的equals()方法用来找到键值对。如果链表大小超过阈值（TREEIFY\_THRESHOLD, 8），链表就会被改造为树形结构。

解决哈希冲突的常用方法有4种：

开放定址法

基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。

再哈希法

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key）  i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

链地址法

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

**10 如何保证集合是线程安全的? ConcurrentHashMap如何实现高效地线程安全**

1. HashTable 低效的加锁方式
2. Collections 提供的同步包装器
3. ConcurrentHashMap 的设计原理：

早期(Java 7)实现原理

* 分离锁 Segment 对数组进行分段锁定，基于 ReetrantLock
* HashEntry内部使用 volatile 保证可见性
* 一个 Segment 可以对应多个桶，先定位到Segment 再进行二次哈希定位具体链表
* size() 方法获取大小时会进行2次重试，不保证一定准确

Java 8 和之后版本 ConcurrentHashMap 变化：

* 总体结构类似于 HashMap，锁的粒度更细
* 初始化利用volatile的 sizeCtl检测是否发生竞争，若竞争则等待，然后CAS初始化
* 使用 CAS 操作进行无锁并发操作，key哈希得到桶的位置，为空进行CAS加新节点，否则使用 synchronized加锁，链表达到数量之后转换红黑树
* size() 方法利用 LongAdd 进行累计



4、redis的数据类型，以及每种数据类型的使用场景

回答：一共五种

(一)String

这个其实没啥好说的，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

(二)hash

这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。博主在做单点登录的时候，就是用这种数据结构存储用户信息，以cookieId作为key，设置30分钟为缓存过期时间，能很好的模拟出类似session的效果。

(三)list

使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。

(四)set

因为set堆放的是一堆不重复值的集合。所以可以做全局去重的功能。为什么不用JVM自带的Set进行去重？因为我们的系统一般都是集群部署，使用JVM自带的Set，比较麻烦，难道为了一个做一个全局去重，再起一个公共服务，太麻烦了。

另外，就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

(五)sorted set

sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作。另外，参照另一篇《分布式之延时任务方案解析》，该文指出了sorted set可以用来做延时任务。最后一个应用就是可以做范围查找。

**11 Java提供了哪些IO方式？ NIO如何实现多路复用**

1. 传统的IO类库结构
2. NIO 的基本组成和多路复用模型，Buffer,Channel,Selector
3. Socket IO 中 传统实现方式与 NIO 的区别
4. NIO 2 的基本组成
5. 区分同步和异步，阻塞和非阻塞

**12 Java有几种文件拷贝方式？哪一种最高效**

1. 使用 java.io 包下的XXXStream 拷贝 和 使用 NIO 中的FileChannel的 transferTo()
2. 理解用户态和内核态
3. Files.copy 标准库的底层使用的还是用户态的拷贝
4. NIO 中 Direct Buffer 堆外内存和垃圾收集（只有FullGC 时收集），使用场景
5. 高效文件拷贝的原则：   
   * 使用缓存，减少IO次数
   * transferTo 等机制，减少上下文切换和额外IO
   * 尽量减少不必要的转换，编解码，对象序列化之类

**13 谈谈接口和抽象类有什么区别**

1. 接口和抽象类的设计原则
2. 面向对象基本要素：抽象，封装，继承，多态
3. 面向对象的S.O.L.I.D原则：   
   * 单一职责
   * 开关原则
   * 里氏替换
   * 迪米特法则
   * 接口分离
   * 依赖反转

1 支持多重继承：接口支持；抽象类不支持；类不支持；

2 支持抽象函数：接口语义上支持；抽象类支持；类不支持；

3 允许函数实现：接口不允许；抽象类支持；类允许；

4 允许实例化：接口不允许；抽象类不允许；类允许；

5 允许部分函数实现：接口不允许；抽象类允许；类不允许。

6 定义的内容：接口中只能包括public函数以及public static final常量；抽象类与类均无任何限制。

7 使用时机：当想要支持多重继承，或是为了定义一种类型请使用接口；当打算提供带有部分实现的“模板”类，而将一些功能需要延迟实现请使用抽象类；当你打算提供完整的具体实现请使用类。

**14 谈谈你知道的设计模式**

设计模式分类：

* 创建型模式—对对象创建过程的各种问题和解决方案的总结：

工厂模式、单例模式、构建器模式、原型模式

* 结构型模式—针对软件设计结构的总结，关注于类、对象继承、组合方式的实践：

桥接模式、适配器模式、装饰者模式、代理模式、组合模式、外观模式、享元模式等

* 行为型模式—从类或对象之间交互、职责划分等角度总结的模式：

策略模式、解释器模式、命令模式、观察者模式、迭代器模式、模板方法模式、访问者模式

通过实例学习和理解设计模式

Spring 中常见的设计模式：

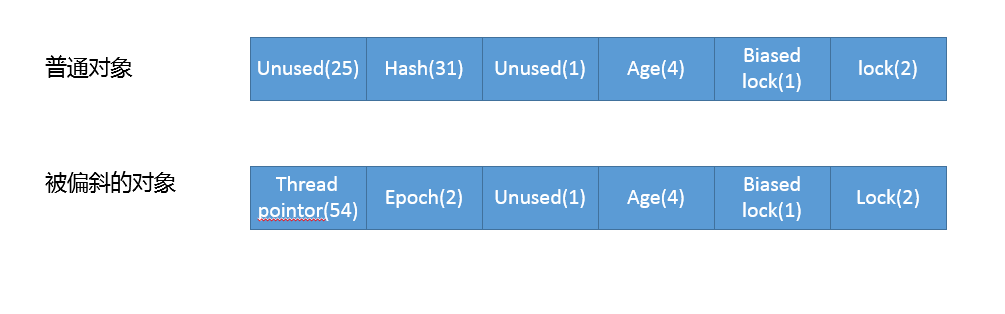
* BeanFactory 和 Application 应用工厂模式
* Bean 的创建过程中，提供了单例和原型模式
* AOP 中的代理模式，装饰器模式，适配器模式
* 各种事件监听器，观察者模式
* 类似JdbcTemplate 的模板模式

**15 synchronized 和 ReentrantLock 有什么区别**

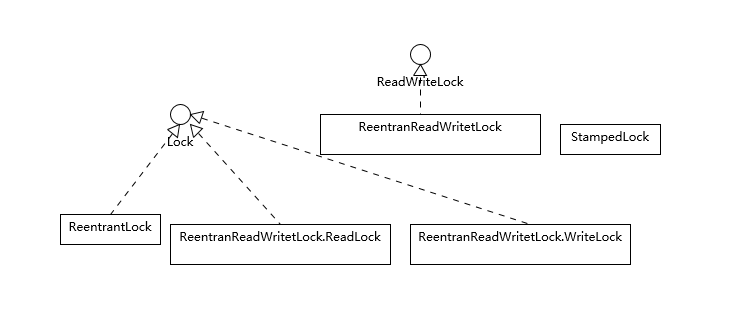
1. 理解线程安全，保证线程安全有哪些方案
2. 理解synchronized 和 ReentrantLock的机制和基本使用
3. 线程安全的特性：原子性，可见性，有序性
4. ReentrantLock的特点：公平锁，可重入锁，手动释放，超时等待，响应中断等
5. Condition 的应用：ArrayBlockingQueue 源码分析 lock 和 condition的作用

**16 synchronized底层如何实现？什么是锁的升级、降级**

1. synchronized 代码块由 monitorenter/monitorexit 指令实现，Monitor对象是同步的基本单元
2. Java 6之后 Monitor 的实现三种不同的锁：偏斜锁(BiasedLocking)、轻量级锁和重量级锁。升级降级就是三种不同锁的转换   
   * 没有竞争时使用偏向锁，JVM利用CAS在对象头的MarkWord设置当前线程ID
3. 当有线程竞争时，需要撤销revoke偏向锁，利用CAS获取锁，成功则用轻量级锁，失败升级为重量级锁



1. 不同的锁类型：



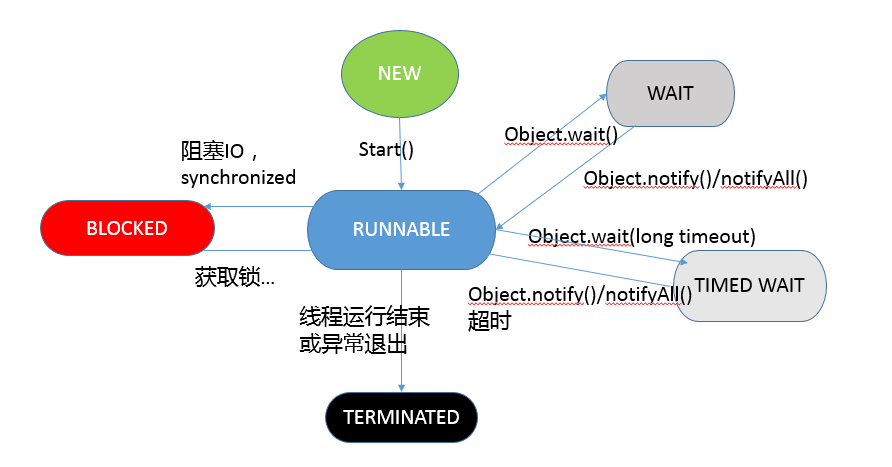
1. 读写锁，StampedLock

**17 一个线程两次调用start()方法会出现什么情况**

1. 理解线程的生命周期状态：java.lang.Thread.State:
   * NEW
   * RUNNABLE
   * BLOCKED :阻塞，线程被挂起，无法进入临界区
   * WAITING :进入临界区后，发现某条件未达到而等待
   * TIMED\_WAIT :进入临界区后，基于超时的等待
   * TERMINATED
2. 线程是什么，Java底层的实现方式:

操作系统调度的最小单元，Java线程映射到OS内核线程

1. 线程状态的切换



**18 什么情况下Java程序会产生死锁？如何定位、修复**

1. 写一个死锁程序
2. 诊断死锁的工具：
   * jstack
   * API：重量级操作ThreadMXBean mbean = ManagementFactory.getThreadMXBean();

Q:分布式环境可以用API实现吗

**19 Java并发包提供了哪些并发工具类**

1. java.util.concurrent 包下各类基本分类，使用

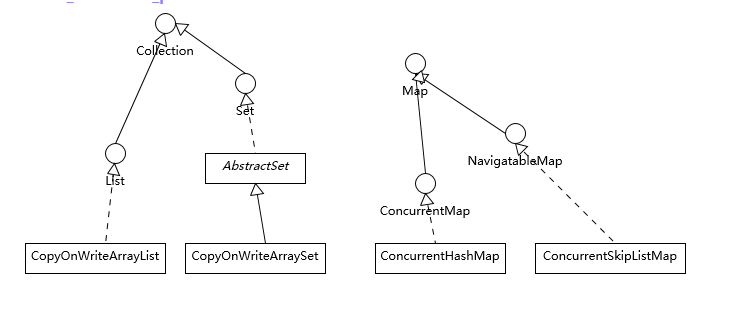
把握主要的组成部分：

* + 各种同步结构
  + CountDownLatch 无法重用，countDown/await ，基于事件
  + CyclicBarrier 可以重用，所有线程await() 之后自动重置，基于线程
  + Semaphore
  + Phase 与 CountDownLatch 类似，允许线程动态注册
  + 各种线程安全容器
  + 各种并发队列的实现和场景
  + Executor框架

1. 典型使用场景

车站出租车每次装5个人才能出发，再放入5个人进入

1. 并发容器



**20 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别**

ArrayBlockingQueue: 有界队列，一把锁，注意 add, offer , put 区别

SynchronousQueue: 容量为0，每个操作都要等待

PriorityBlockingQueue: 无界优先队列

LinkedBlockingQueue: 没有设置容量时为无界队列，头尾2把锁

ConcurrentLinkedQueue: 基于CAS无锁的并发队列，性能更好，可以提供更高吞吐量

**21 Java并发类库提供的线程池有哪几种？ 分别有什么特点**

1. Executors 工具类，ThreadPoolExecutor线程池核心类
2. Executors 提供的线程池：
   * newCachedThreadPool():
3. public static ExecutorService newCachedThreadPool() {
4. return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,
5. 60L, TimeUnit.SECONDS,
6. new SynchronousQueue<Runnable>());

}

* + 1
  + 2
  + 3
  + 4
  + 5

可根据实际情况调正线程池大小，需要注意的是线程数没有限制，空闲的线程缓存60秒

* + newFixedThreadPool(int nThreads)

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

* + 1
  + 2
  + 3
  + 4
  + 5

创建固定大小的线程池，工作队列使用LinkedBlockingQueue,使用时注意任务的堆积影响

* + newSingleThreadExecutor()

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

* + 1
  + 2
  + 3
  + 4
  + 5
  + 6

大小为1的线程池，同样要注意等待任务的堆积影响

* + newSingleThreadScheduledExecutor(): 线程池大小为1，可以做定时任务
  + newScheduledThreadPool(int corePoolSize)：可以指定线程数的定时任务线程池
  + newWorkStealingPool(int parallelism)：Java 8新增，构建ForkJoinPool ,并行处理任务

一般建议使用 ThreadPoolExecutor创建语义明确的线程池

**22 AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？**

1. AtomicXXX 的实现原理：CAS原理
2. CAS 有什么问题：   
   * ABA问题，解决方案：版本控制：AtomicStampedReference
   * 自旋次数（CAS的设计前提是竞争时短暂的）
3. 如果要使用CAS操作，可以使用 AtomicLongFieldUpdater 包装自己的数据
4. 理解AQS(AbstractQueuedSynchronizer)的原理,简单拆分：   
   * 表征状态的 volatile变量：state
   * 一个FIFO 的等待线程队列，处理线程竞争和等待
   * 各种基于CAS的基础操作方法，和等待实现的 tryAcquire 方法和 tryRelease 方法
5. 以 ReentrantLock为例理解 AQS 作为基础实现的作用

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

* 1
* 2
* 3
* 4
* 5

​ 理解 tryAcquire 和 acquireQueued 方法的实现

**23 请介绍类加载过程，什么是双亲委派模型**

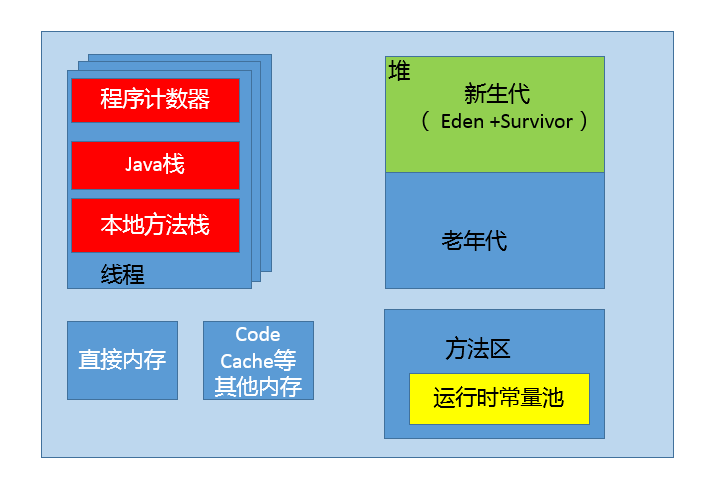
1. Java 的类加载过程主要三个步骤：加载、链接（验证、准备、解析）、初始化
2. 理解JVM各阶段做了什么工作
3. 类加载器与双亲委派模型
4. Java 9 中类加载器发生了哪些变化
5. 有方法可以降低类加载开销吗 ： AOT、AppCDS

**24 有哪些方法可以在运行时动态生成一个Java类**

1. 加深理解类加载机制，defineClass 等
2. 动态代理的原理，字节码操纵技术还有哪些，ASM, Javassist , cglib

**25 谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError**

1. JVM 内存划分，参考《深入理解Java虚拟机》
   * 程序计数器
   * Java虚拟机栈
   * 堆
   * 方法区
   * 运行时常量池
   * 本地方法栈



注意JDK版本，JDK 7前还有把方法区称为永久代，之后由元数据区取代,而字符串缓存和静态变量并没有转移到元数据区，是直接分配在了堆上

1. 常见 OOM 分析

**26 如何监控和诊断JVM堆内和堆外内存使用**

* 图形化工具：JConsole , VisualVM
* 命令行工具：jstat , jmap
* GC 日志
* Java Mission Control(JMC)

细化对各部分内存区域的理解，堆内结构的理解，参数调整

参数调整：

* 最大堆体积 -Xmx value
* 初始化最小堆体积 -Xms value
* 老年代和新生代的比例 -XX:NewRatio=value
* 新生代大小比例 -XX:SurviorRatio=value

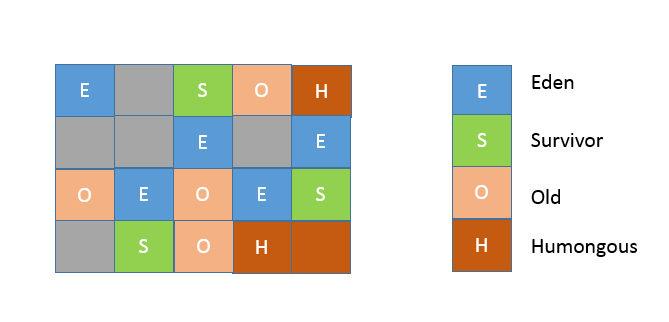
堆外内存

**27 Java常见的垃圾收集器有哪些**

1. 主流 Oracle JDK 的GC：
   * Serial GC :采用标记-整理（Mark-Compact）算法
   * ParNew GC : Serial GC的多线程实现
   * CMS (Concurrent Mark Sweep) GC : 基于标记-清除(Mark-Sweep)算法，存在内存碎片问题，设计目标是减少停顿
   * Parrallel GC ： 吞吐量优先的GC, JDK8 等为server 模式默认GC,新生代老年代并行
   * G1 GC : 兼顾吞吐量和停顿时间，JDK 9后默认GC
2. 如何判断一个对象是否可以回收？引用计数法和可达性分析
3. 常见垃圾收集算法：   
   * 复制算法
   * 标记-清除
   * 标记-整理
4. 垃圾收集的基本过程理解

**28 谈谈你的GC调优思路**

1. 明确GC 调优的目的：性能角度（内存占用footprint、延时latency、吞吐量throughput），其他方面
2. 一般调优思路：
   * 明确需求和问题，确定目标
   * 掌握JVM，GC的状态，是否有GC调优的必要
   * 应用的特征和 GC的类型是否符合
   * 分析确定需要调正的参数或软硬件
   * 验证目标是否达到
3. G1 GC 的内部结构和主要机制



内部划分为同等大小的 region（数值是在 1M 到 32M 字节之间的一个 2 的幂值数），年代成为逻辑概念，增加了Humongous，G1会将超过 region 50%大小的对象归类为Humongous对象，逻辑上属于老年代。

* + 新生代，G1 采用并行的复制算法，会发生 STW暂停
  + 老年代，大部分情况为并发标记，整理发生在新生代GC时捎带进行，而且是增量进行

查询有关资料学习

**29 Java内存模型中的happen-before是什么**

1. Java （JMM）内存模型的理解
2. 先行发生原则(Happen-before) 是Java内存模型中保证多线程操作可见性的机制
3. 理解 JMM 试图解决什么问题：
   * 简化多线程编程，保证程序可移植性
   * 隔离了不同处理器内存排序的区别
   * 解决多线程中可见性等问题
4. JMM内部实现依赖于内存屏障，通过禁止某些重排序的方式，提供内存可见性保证。
5. 线程安全的原子性，可见性，有序性的理解，同时理解(sychronized,volatile的原理)

**30 Java程序运行在Docker等容器环境有哪些新问题**

(由于本人在docker下使用java比较少，以下只摘录原专栏的主要点)

1. 早期 JDK 在Docker 环境下容易出现的基础问题：   
   * 未配置JVm堆，原数据区，直接内存等参数，可能超过容器限制出现OOM被容器kill
   * 会错误判断CPU资源，因Docker 限制了 CPU的核数
2. 容器对计算资源的管理方式是全新的，Docker通过CCroup实现
3. 解决方案：   
   * 根据容器设置好JVM的各种参数，包括CPU资源
   * 升级到新的JDK 9 ，10

**31 你了解Java应用开发中的注入攻击吗**

注入攻击的基本特征是程序允许攻击者将不可信的动态内容注入到程序中，并将其执行。

常见注入攻击（Java相关）：

* SQL注入
* 操作系统命令注入
* XML注入攻击

**32 如何写出安全的Java代码**

从Java语言的角度理解常见的攻击，如哈希碰撞等消耗资源的操作

从设计，开发，测试，部署等不同阶段考虑有哪些安全的策略

**33 后台服务出现明显“变慢”，谈谈你的诊断思路**

1. 服务器变慢的定义，突然变慢还是长时间，是否重现，“慢”的定义
2. 理清问题的症状，定位原因：   
   * 错误日志
   * 系统资源等情况的检查
   * 监控Java服务自身，GC日志是否异常
3. 业界最广泛的性能分析方法论：   
   * 自上而下。从应用顶层逐步深入到具体模块
   * 自下而上。从类似CPU这种底层往上分析
4. 首先排除功能性错误，然后使用典型的自上而下分析：   
   * 系统层面关注点：CPU，内存，IO（top,free，iostat等命令）
   * JVM层面，利用工具，JMC,JConsole，日志等

**34 有人说“Lambda能让Java程序慢30倍”，你怎么看**

1. 理解基准测试定义的范围和目标
2. 微基准测试
3. JMH的使用

**35 JVM优化Java代码时都做了什么**

1. JVM的优化分为：   
   * 运行时优化（runtime）
   * 即时编译器优化（JIT）–将热点代码以方法为单位转换成机器码
2. 从整体了解Java代码编译、执行的过程

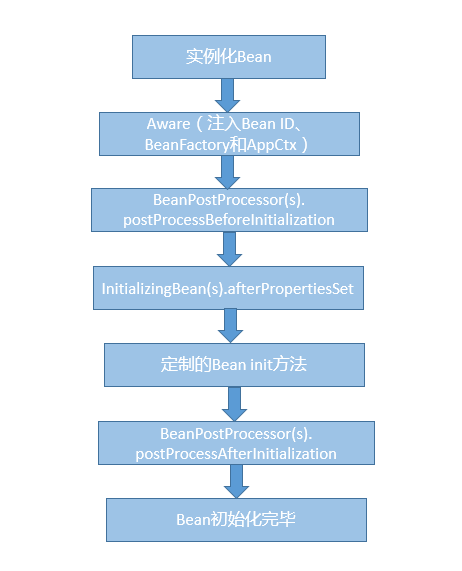
这部分内容建议参考 《深入理解Java虚拟机》及相关资料学习

**36 谈谈MySQL支持的事务隔离级别，以及悲观锁和乐观锁的原理和应用场景**

1. MySQL事务隔离级别：   
   * 读未提交
   * 读已提交
   * 可重复读
   * 串行化
2. 理解悲观锁 和 乐观锁的出发点及使用

**37 谈谈Spring Bean的生命周期和作用域**

1. Spring Bean生命周期
   * 初始化参考下图：



* + 销毁：依次调用 DisposableBean 的 destroy 方法和 Bean 自身定制的destroy 方法

1. Spring Bean 的作用域：
   * Singleton
   * Prototype
   * Request
   * Session
   * GlobalSession
2. Spring 的基础机制：控制反转（依赖注入），AOP及具体相关概念

**38 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗**

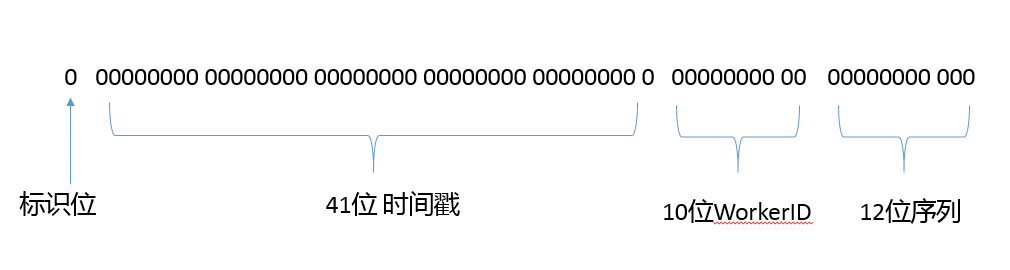
（Netty 还不太熟悉，以后补上总结）

Netty 在NIO 基础上进行了很多改进：

* Reactor模式实现的线程模型，利用EventLoop 创建新的机制
* Zero-Copy 机制，各种降低内存分配和回收的开销等
* 通信协议，序列化方面的优化

**39 谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响**

1. 分布式ID的定义，基本要求：
   1. 全局唯一，分布式系统中的唯一
   2. 有序性，通常要求是有序递增
2. 业界通常做法：
   1. 基于数据库自增的实现。扩展性和可靠性存在局限
   2. 基于 Twitter 的Snowflake的实现等，如下64位，通常使用Java中的long存储



* + - 头部是1位的正负标识
    - 41位的时间戳，通常使用System.currentTimeMills()
    - 10位WorkerID ：5位数据中心+5位机器ID
    - 最后12位是单位毫秒内可生成的序列号数目的理论极限
  1. Redis，Zookeeper，MangoDB等中间的解决方案
  2. 其他开源实现

1. Snowflake 中大都依赖于System.currentTimeMills()与夏/冬时令没有关系，不受影响
2. 分布式ID除了唯一和有序还会有其他要求，如有意义（业务相关），高可用性，紧凑等
3. 主流方案的优缺点分析
   1. 数据库自增方案的局限
   2. 时钟偏斜问题
   3. 等等

**专栏链接**：   
[https://time.geekbang.org/column/intro/82?code=IAMPy%2FqD3Mg2lpnKMAho6zjuewt%2FKEVD8gX%2FUCdV7Xk%3D](https://time.geekbang.org/column/intro/82?code=IAMPy/qD3Mg2lpnKMAho6zjuewt/KEVD8gX/UCdV7Xk=)