基础

数据结构

基本数据类型有哪些，各站多少字节

1、byte----->1字节
2、short---->2字节
3、int------>4字节
4、long----->8字节
5、char----->2字节
6、float---->4字节
7、double--->8字节
8、boolen--->1bit

自定义注解

> 元注解
>
> 元注解的作用就是负责注解其他注解
>
> - @Target
> - @Retention
> - @Documented
> - @Inherited
>
> @Target说明了Annotation所修饰的对象范围，取值(ElementType)有：
>
> 1.CONSTRUCTOR:用于描述构造器
> 2.FIELD:用于描述域
> 3.LOCAL_VARIABLE:用于描述局部变量
> 4.METHOD:用于描述方法
> 5.PACKAGE:用于描述包
> 6.PARAMETER:用于描述参数
> 7.TYPE:用于描述类、接口(包括注解类型) 或enum声明
>
> @Retention用于描述注解的声明周期（被描述的注解什么范围内有效），取值（RetentionPoicy）有：
>
> 1.SOURCE:在源文件中有效（即源文件保留）
> 2.CLASS:在class文件中有效（即class保留）
> 3.RUNTIME:在运行时有效（即运行时保留）
>
> @Documented标记后，可以被例如javadoc此类的工具文档化
>
> @Inherited阐述了某个被标注的类型是被继承的
>
> https://www.cnblogs.com/peida/archive/2013/04/24/3036689.html.

HashMap

数据结构：散列桶数组 + 链表

一般简单的HashMap，大致流程，key值经过计算打散到散列数组对应位置，以链表形式挨个比对完成对应操作

主要了解：**扰动函数、初始化容量、负载因子、扩容方法以及链表和红黑树转换的使用**

扰动函数

散列数组的index，为什么不直接Key的hashCode值取余length，而是多了一次扰动计算

> **String的hashCode方法，为什么选择31作为乘积值?**
>
> 31为奇质数，偶数会导致乘积运算时数据溢出；31 = 2<<5-1,乘积运算可以使用位移提升性能；
>
> hash碰撞概率小（运算过程在int取值范围内），基本稳定。

HashMap，key通过Hash计算索引位置，然后找到对应元素或链表或树，为什么不直接取余(&(size-1))?

key的hashCode，进行一次**扰动计算**，哈希值右移16位（默认初始化大小16长度），与原哈希做异或，这样混合了原哈希值中的高位和地位，增大了**随机性**，进一步**减少了哈希碰撞**，分布的更均匀

初始化容量和负载因子

负载因子默认0.75，就是说当阀值容量占了3/4 时赶紧扩容，减少 Hash 碰撞

扩容元素拆分

扩容方式就是需要把元素拆分到新的数组中。

拆分元素的过程中，原 jdk1.7 中会需要重新计算哈希值，但是到 jdk1.8 中已经进行优化，不在需要重新计算

基本的数据操作功能：存储、删除、获取、遍历

- 存储删除操作可能会遇到哈希碰撞（equals，替换或者插入（树化）），可能会遇到扩容，扩容后，原来因为哈希碰撞存放的链表或者红黑树，都需要进行拆分存放

ArrayList

数据结构：数组

创建一个集合，初始容量不指定时为0，在第一次添加元素时，集合会扩容为10；继续添加至超过上限，会继续扩容为原来的3/2，也就是15

Arrays.asList构建的集合，不能再添加，不能再删除

JVM性能调优

JVM参数配置最佳实践

在一个机器中JVM进程占用总内存(堆，元空间，堆外内存，CodeCache)一般不建议超过总内存的80%

JVM参数	说明	4c8g	8c16g
-Xms	初始堆内存大小	4g	10g
-Xmx	最大堆内存大小	4g	10g
-Xmn	新生代空间大小	2g	5g
-XX:MetaspaceSize	初始元空间大小	384m	512m
-XX:MaxMetaspaceSize	最大元空间大小	384m	512m
-XX:MaxDirectMemorySize	最大堆外内存大小	1g	1g
-XX:ReservedCodeCacheSize	CodeCache(本地方法)大小	256m	256m

不分机器的通用配置

-XX:SurvivorRatio=10，survivor区大小为新生代的1/12

-XX:ParallelGCThreads=${CPU_COUNT}，并发GC线程数（Young GC 在jdk1.8默认使用多线程线性收集器）

-XX:+UseConcMarkSweepGC ， old区使用CMS GC

-XX:CMSMaxAbortablePrecleanTime=5000， CMS GC的回收超时时间，避免GC太久

-XX:CMSInitiatingOccupancyFraction=80 ，Old区达到80%触发CMS GC，如果不设置，JVM会自适应，效果不好

-XX:UserCMSInitiatingOccupancyOnly，必须配置这个，上面这个Fraction配置才会生效

-XX:+CMSScavengeBeforeRemark，执行GMS GC的remark前先Young GC一次，缩短remark时间（会有跨代引用，老年代进行GC Roots追踪时，同样会扫描年轻代）

-XX:+ExplicitGCInvokesConcurrent， 调用System.gc()时触发CMS GC而不是Full GC

-XX:HeapDumpOnOutOfMemoryError，OOM时自动jmap dump内存

-XX:HeapDumpPath=/home/admin/logs/java.hprof，OOM时dump内存的位置

-Xloggc:/home/admin/logs/gc.log，GC log位置

-XX:+PrintGCDetails，打印GC详细信息

-XX:+PrintGCDeteStamps，将GC时间戳改为人类可识别的时间刻

-XX:+UseGCLogFileRotation，开启GC日志文件轮询

-XX:GCLogFileSize=50M，每个GC日志文件最大大小

-XX:NumberOfGCLogFiles=5，保留的GC日志文件个数

JVM垃圾收集器

Serial（串行）收集器（**新生代采用复制算法，老年代采用标记-整理算法**）

ParNew 收集器（ Serial 收集器的多线程版本）

Parallel Scavenge 收集器（与ParNew几乎一样，关注点吞吐量，高效率利用CPU）**JDK1.8 默认收集器**

CMS（Concurrent Mark Sweep）收集器，最短回收停顿时间为目标的收集器。注重用户体验，“标记-清除”算法(并发收集、低停顿)，参考https://blog.csdn.net/zqz_zqz/article/details/70568819

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征

ZGC（The Z Garbage Collector）是JDK 11中推出的一款低延迟垃圾回收器(在 ZGC 中出现 Stop The World 的情况会更少，目标10ms)

强引用，软引用，弱引用，虚引用

软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生

程序里面最快的缓存就是直接在内存里面构建缓存。但是如果我们使用强引用构建缓存的话，一旦缓存过大无法回收就会导致内存溢出。这个时候我么可以使用软引用构建缓存。

```
// 软引用
Caffeine.newBuilder().softValues().build();
```

如何根据JVM定位OOM问题

todo

缓存是否可以使用堆外内存 https://www.jianshu.com/p/17e72bb01bf1

OHC，全称为 **off-heap-cache**，即堆外缓存，是一款基于Java 的 key-value 堆外缓存框架

与堆内空间不同，堆外空间不影响GC，由应用程序自身负责分配与释放内存

两种分配堆外内存的方法，Unsafe和NIO；

caffinitas，OHC是将Java对象序列化后存储在堆外，因此用户需要实现 org.caffinitas.ohc.CacheSerializer 类，OHC会运用其实现类来序列化和反序列化对象。

https://www.cnblogs.com/liang1101/p/13499781.html

https://my.oschina.net/u/4384701/blog/3386255

堆外内存，其实就是不受JVM控制的内存。相比于堆内内存有几个**优势**：
1 减少了垃圾回收的工作，因为垃圾回收会暂停其他的工作（可能使用多线程或者时间片的方式，根本感觉不到）
2 加快了复制的速度。因为堆内在flush到远程时，会先复制到直接内存（非堆内存），然后在发送；而堆外内存相当于省略掉了这个工作。
而福之祸所依，自然也有不好的一面：
1 堆外内存难以控制，如果内存泄漏，那么很难排查
2 堆外内存相对来说，不适合存储很复杂的对象。一般简单的对象或者扁平化的比较适合。

Netty在直接内存上的封装

在NIO的框架下，很多框架会采用DirectByteBuffer来操作，这样分配的内存不再是在java heap上，而是在C heap上，经过性能测试，可以得到非常快速的网络交互，在大量的网络交互下，一般速度会比HeapByteBuffer要快速好几倍。

避免了在Java 堆和Native 堆中来回复制数据

线程安全

多线程顺序执行

- join方法等待线程销毁，阻塞主线程
- CountDownLatch倒数计时器

```
CountDownLatch countDownLatch1 = new CountDownLatch(1);
countDownLatch1.await();
doSomething();
countDownLatch2.countDown();
```

- 单线程线程池，newSingleThreadExecutor()

java线程通信方式

- **“共享内存”式的通信**
- **Object类的 wait() 和 notify() 方法**，Condition的await()和signal()方法
- **管道通信**就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

线程间的协作

wait/notify/sleep/yield/join

sleep方法只是暂时让出CPU的执行权

yield方法的作用是暂停当前线程，以便其他线程有机会执行，不过不能指定暂停的时间，并且也不能保证当前线程马上停止。yield方法只是将Running状态转变为Runnable状态（很少用）

join方法的作用是父线程等待子线程执行完成后再执行，换句话说就是将异步执行的线程合并为同步的线程。

https://www.cnblogs.com/paddix/p/5381958.html

jvm内存模型，jvm线程模型

java内存模型

得出线程安全讲的是什么？线程是什么，什么又叫做安全？

todo

https://blog.csdn.net/u014730165/article/details/81981154

ThreadLocal会有什么问题

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，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。

由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，**发生GC时弱引用Key会被回收，而Value不会回收。**

当线程没有结束，但是ThreadLocal已经被回收，则可能导致线程中存在ThreadLocalMap的键值对，**造成内存泄露。**（`ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在`）

避免泄露两种方法

- 使用完线程共享变量后，显示调用ThreadLocalMap.remove方法清除线程共享变量；
- JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了

线程池的核心参数 https://blog.csdn.net/lveex/article/details/78261173

- corePollSize：核心线程数。在创建了线程池后，线程中没有任何线程，等到有任务到来时才创建线程去执行任务。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中。
- maximumPoolSize：最大线程数。表明线程中最多能够创建的线程数量。
- keepAliveTime：空闲的线程保留的时间，TimeUnit：空闲线程的保留时间单位
- BlockingQueue：阻塞队列，存储等待执行的任务。参数有ArrayBlockingQueue（有界队列）、LinkedBlockingQueue（无界队列，可产生OOM）、SynchronousQueue可选。
- ThreadFactory：线程工厂，用来创建线程，新的线程都是由ThreadFactory创建的，系统默认使用的是Executors.defaultThreadFactory创建的，用它创建出来的线程的优先级、组等都是一样的，并且他都不是守护线程。我们也可以使用自定义的线程创建工厂，并对相关的值进行修改
- RejectedExecutionHandler：队列已满，而且任务量大于最大线程的异常处理策略
    - ThreadPoolExecutor.AbortPolicy：丢弃任务并抛出RejectedExecutionException异常
    - ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常
    - ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
    - ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

分IO密集型与CPU密集型的，配置差异，CPU密集型核心线程数与CPU核数一致，IO可以多一点

Synchonize vs ThreadLocal区别

都是解决线程安全问题，Sync解决共享变量并发访问的线程安全的方式是：线性挨个执行；
ThreadLocal是利用线程间的隔离，独立享有各自的内存达到线程安全

悲观锁乐观锁，单纯讲这个没用结合Synchronize关键字的锁升级将悲观乐观

> 悲观锁：假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁；

> 传统的关系型数据库里边就用到了很多悲观锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中`synchronized`和`ReentrantLock`等独占锁就是悲观锁思想的实现
>
> 乐观锁：假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现；
>
> **乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于**write_condition机制**，其实都是提供的乐观锁。在Java中`java.util.concurrent.atomic`包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的

Java6以上版本对synchronized的优化

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步

- 代码块同步: 通过使用monitorenter和monitorexit指令实现的
- 同步方法: ACC_SYNCHRONIZED修饰

多线程下**synchronized的加锁**就是对**同一个对象的对象头中的MarkWord**中的变量进行CAS操作

锁可以升级, 但不能降级. 即: 无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁是单向的

偏向锁，不存在多线程竞争，利用epoch表示一个偏向锁的时间戳判断，默认启用

> 所谓“自旋”，就是让线程去执行一个无意义的循环，循环结束后再去重新竞争锁，如果竞争不到继续循环，循环过程中线程会一直处于running状态，但是基于JVM的线程调度，会出让时间片，所以其他线程依旧有申请锁和释放锁的机会。
>
> 自旋锁省去了阻塞锁的时间空间（队列的维护等）开销，但是长时间自旋就变成了“忙式等待”，忙式等待显然还不如阻塞锁。所以自旋的次数一般控制在一个范围内，例如10,100等，在超出这个范围后，自旋锁会升级为阻塞锁

轻量级锁，发生多线程竞争则升级到轻量级锁，MarkWord膨胀。**线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋获取锁，当自旋获取锁仍然失败（一般有个范围）时，表示存在其他线程竞争锁(两条或两条以上的线程竞争同一个锁)，则轻量级锁会膨胀成重量级锁**

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗, 和执行非同步代码方法的性能相差无几.	如果线程间存在锁竞争, 会带来额外的锁撤销的消耗.	适用于只有一个线程访问的同步场景
轻量级锁	竞争的线程不会阻塞, 提高了程序的响应速度	如果始终得不到锁竞争的线程, 使用自旋会消耗CPU	追求响应时间, 同步快执行速度非常快
重量级锁	线程竞争不适用自旋, 不会消耗CPU	线程堵塞, 响应时间缓慢	追求吞吐量, 同步快执行时间速度较长

https://www.cnblogs.com/wuqinglong/p/9945618.html

https://www.cnblogs.com/brithToSpring/p/13307598.html

HashMap为什么线程不安全

- 在JDK1.7中，当并发执行扩容操作时会造成环形链和数据丢失的情况
- 在JDK1.8中，在并发执行put操作时会发生数据覆盖的情况

ConcurrentHashMap为什么去掉Segment

- 分段锁的优势在于保证在操作不同段 map 的时候可以并发执行，操作同段 map 的时候，进行锁的竞争和等待。这相对于直接对整个map同步synchronized是有优势的。
- 缺点在于**分成很多段时会比较浪费内存空间**(不连续，碎片化); 操作map时竞争同一个分段锁的概率非常小时，分段锁反而会造成更新等操作的长时间等待; 当某个段很大时，**分段锁的性能**会下降

排序算法

- 快排
- 归并
    - 分，治 https://www.cnblogs.com/chengxiao/p/6194356.html
    - Arrays.sort()采用的就是TimeSort算法， 归并算法的优化版
- 堆

常用的加密算法

加密算法分 **对称加密** 和 **非对称加密**，其中对称加密算法的加密与解密 **密钥相同**，非对称加密算法的加密密钥与解密 **密钥不同**，此外，还有一类 **不需要密钥** 的 **散列算法**

> 常见的 **对称加密** 算法主要有 `DES`、`3DES`、`AES` 等，常见的 **非对称算法** 主要有 `RSA`、`DSA` 等，**散列算法** 主要有 `SHA-1`、`MD5` 等

https://blog.csdn.net/baidu_22254181/article/details/82594072

http与https区别，https加密算法的过程，用到了哪些，具体是如何加密的

HTTP： 直接通过明文在浏览器和服务器之间传递信息。

HTTPS： 采用 对称加密 和 非对称加密 结合的方式来保护浏览器和服务端之间的通信安全。
对称加密算法加密数据+非对称加密算法交换密钥+数字证书验证身份=安全

> HTTPS其实是有两部分组成：HTTP + SSL / TLS，也就是在HTTP上又加了一层处理加密信息的模块。服务端和客户端的信息传输都会通过TLS进行加密，所以传输的数据都是加密后的数据。

深度不够

谦逊一点就好，无关其他，各方面的事情不用也不可能样样精通，尽量表现出所知所学即可，也表现出谦逊好学。