**JAVA & JVM**

▶**String 不可变的原因**

String是最常用的类，大量创建，会很大内存开销，String pool去维护一份。

不可变性具备天生的线程安全

▶**泛型**

<? extends T> 表示该通配符所代表的类型是T类型的子类

<? super T> 表示该通配符所代表的包括T在内的任何T的父类

▶**反射的工作原理**

借助Class.java、 Constructor.java、 Method.java、 Field.java这四个类在程序运⾏

时动态访问和修改任何类的⾏为和状态

▶**动态代理**

运行时动态代理目标方法，植入处理逻辑。

实际上是通过JDK提供的反射机制（动态调用目标方法）实现的。

**动态代理的方式有两种：**

* JDK动态代理：利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。
* CGLIB动态代理：利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

**JDK动态代理和CGLIB字节码生成的区别？**

* JDK动态代理只能对实现了接口的类生成代理，而不能针对类。
* CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法。

因为是继承，所以该类或方法最好不要声明成final

▶**JDK1.8的新特性**

◇接口default方法

◇Lambda表达式

◇函数式接⼝

@FunctionalInterface

Supplier

Function

Consumer

◇⽅法与构造函数引⽤

Integer::valueOf

◇局部变量限制

◇Date Api更新

LocalDate/LocalTime/LocalDateTime

DateTimeFormatter

◇流

◇Objects⽅法新特性 StringJoiner是JDK8为了方便构造限定分隔符，甚至于给定前缀和后缀，而提供的一个类，示例代码如下：

StringJoiner sj = new StringJoiner(",","{","}");

sj.add("first").add("second").add("third");

System.out.println(sj.toString());

#### 运行结果

{first,second,third}

▶**包装类**

◇自动拆箱

将包装类的类型直接赋值给对应的基本数据类型的变量

例如： Integer i = 100; int a = i;

实际上底层调用了相应的xxxValue()方法：

Integer i = 100; int a = i.intValue();

在包装类进行与基本数据类型比较时 ，会存在自动拆箱的操作。

◇自动装箱

（-128~127）之内 调用相应包装类的valueOf()

不在（-128~127）之内 直接 new 对象

▶**为什么要序列化**

假如我有两个类，分别是A和B，B类中含有一个指向A类对象的引用， 现在我们对 两个类进行实 例化{ A a = new A(); B b = new B(); }。这时在内存中实际上分配了两个空间， 一个存储对象a，一个存 储对象b。接下来我们想将它们写入到磁盘的一个文件中去， 就在写入文件时出现了问题！因为对 象b包含对对象a的引用，所以系统会自动的将a 的数据复制一份到b中，这样的话当我们从文 件中恢复对象时(也就是重新加载到内存 中)时，内存分配了三个空间，而对象a同时在内存中存在 两份

▶**阻塞IO**

用户线程发出IO请求，会处于阻塞状态，交出CPU。当数据就绪之后，内核会将数据 拷贝到用 户线程，并返回结果给用户线程，用户线程才解除阻塞状态

▶**非阻塞IO**

用户线程发起IO操作，不需要等待，马上得到error结果，不断地询问内核数据是否就 绪，非阻 塞IO会一直占用 CPU。

▶**NIO**

NIO 主要有三大核心部分：Channel，Buffer, Selector。NIO 基于 Channel 和 Buffer(缓 冲区)进行操 作， 数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听 多 个通道的事件

▶**Set集合**

◇HashSet

基于哈希实现，支持快速查找，但不支持有序性操作，例如根据一个范围查找元素的操作。并 且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。

◇TreeSet

基于红黑树实现，支持有序性操作，但是查找效率不如 HashSet，HashSet 查找时间复杂度为 O(1)，TreeSet 则为 O(logn)；

◇LinkedHashSet

具有 HashSet 的查找效率，且内部使用链表维护元素的插入顺序。

▶**List集合**

◇ArrayList

基于数组实现，可以随机访问。当数组大小不满足时需要增加存储能力，就要将已经有数组的 数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行 复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

◇Vector

线程安全的ArrayList

◇LinkedList

基于双向循环链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如 此，LinkedList 还可以用作栈、队列和双端队列。

不要在 foreach 循环里进行元素的 remove/add 操作。 remove 元素请使用 Iterator 方式，如果并发操作，需要对 Iterator 对象加锁。

▶**BitSet**

数据结构：long数组，默认容量long[1] set方法：

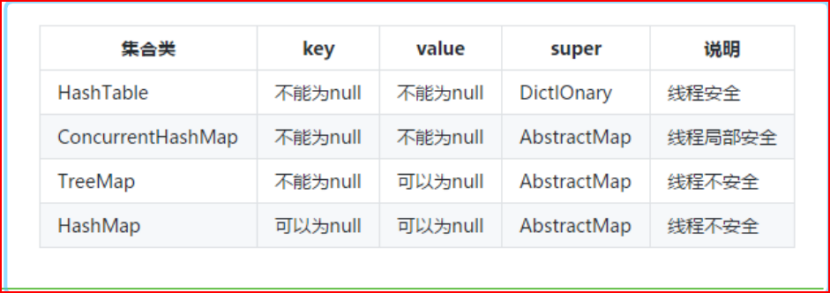
1，bitIndex（数字）除以64，得到wordIndex下标 // 小于64，wordIndex就是0

2，根据wordIndex，判断是否需要扩容

3，words[wordIndex] |= (1L << bitIndex)

问题：数据跨度比较大，创建时候会造成很大的内存浪费

▶**Map的key和value能否为空**



▶**HashMap**

◇原理

HashMap 底层采用一个 Entry[] 数组来保存所有的key-value对，put方法，hash算法来确定数 组上链表位置，equals方法确定链表中的元素位置，然后替换值；get方法，hash算法来确定数 组上链表位置，equals方法确定链表中的元素位置，取出值。（HashMap 最多只允许一条记录 的键为 null，允许多条记录的值为 null。）

◇线程不安全

1.7 会产生死循环(链表头插法造成的，1.8改尾插法)、数据覆盖的问题

1.8 中会有数据覆盖的问题

◇什么时候进行resize

entry[]的容量超过初始容量大小和loadFactor(装载因子)乘积时，就会进行数组扩容，扩容到初始容量2倍

◇1.8的优化

数组+链表改成了数组+链表或红黑树；

链表的插入方式从头插法改成了尾插法，简单说就是插入时，如果数组位置上已经有元素，1.7 将新元素放到数组中，原始节点作为新节点的后继节点，1.8遍历链表，将元素放置到链表的最 后；扩容的时候1.7需要对原数组中的元素进行重新hash定位在新数组的位置，1.8采用更简单 的判断逻辑，位置不变或索引+旧容量大小；

在插入时，1.7先判断是否需要扩容，再插入，1.8先进行插入，插入完成再判断是否需要扩容；

◇怎么解决这个线程不安全的问题？

HashTable

Collections.synchronizedMap

ConcurrentHashMap

▶**ConcurrentHashMap**

◇1.7原理

Segment + HashEntry + Unsafe

◇1.8原理

移除Segment，使锁的粒度更小，Synchronized + CAS + Node + Unsafe

•扩容

sizeCtl ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。

-1 代表table正在初始化

-N 表示有N-1个线程正在进行扩容操作

其余情况：

1、如果table未初始化，表示table需要初始化的大小。

2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍

扩容时候会判断这个值，如果超过阈值就要扩容，首先根据运算得到需要遍历的次数i，然后利 用tabAt方法获得i位置的元素f，初始化一个forwardNode实例fwd，如果f == null，则在table 中的i位置放入fwd，否则采用头插法的方式把当前旧table数组的指定任务范围的数据给迁移 到新的数组中，然后给旧table原位置赋值fwd。直到遍历过所有的节点以后就完成了复制工作， 把table指向nextTable，并更新sizeCtl为新数组大小的0.75倍 ，扩容完成。在此期间如果其他 线程的有读写操作都会判断head节点是否为forwardNode节点，如果是就帮助扩容。

扩容时读写操作如何进行

对于get读操作，如果当前节点有数据，还没迁移完成，此时不影响读，能够正常进行。

如果当前链表已经迁移完成，那么头节点会被设置成fwd节点，此时get线程会帮助扩容。

对于put/remove写操作，如果当前链表已经迁移完成，那么头节点会被设置成fwd节点， 此时写线程会帮助扩容，如果扩容没有完成，当前链表的头节点会被锁住，所以写线程会 被阻塞，直到扩容完成。

◇区别

•put（）

○1.7：先定位Segment，再定位桶，put全程加锁，get时候不加锁，没有获取锁的线程提前找 桶的位置，并最多自旋64次获取锁，超过则挂起。

○1.8：由于移除了Segment，类似HashMap，可以直接定位到桶，拿到first节点后进行判断，1、 为空则CAS插入；2、为-1则说明在扩容，则跟着一起扩容；3、else则加锁put（类似1.7）

注意：

如果key或者value为null，则抛出空指针异常；

如果table为null或者table的长度为0，则初始化table，调用initTable()方法。

计算当前键值的索引位置，如果Hash表中当前节点为null，则将元素直接插入。(注意，这里使用的就是前 面锁说的CAS操作)

如果当前位置的节点元素的hash值为-1，说明这是一个ForwaringNodes节点，即正在进行扩容。那么当前线 程加入扩容。

当前节点不为null，对当前节点加锁，将元素插入到当前节点。在Java8中，当节点长度大于8时，就将节点 转为树的结构。

•get（）

基本类似，由于value声明为volatile，保证了修改的可见性，因此不需要加锁。

▶**并发容器**

◇ConcurrentHashMap

对应的⾮并发容器： HashMap

⽬标：代替Hashtable、 synchronizedMap，⽀持复合操作

原理： JDK6中采⽤⼀种更加细粒度的加锁机制Segment“分段锁”， JDK8中采⽤CAS⽆锁算法。

◇CopyOnWriteArrayList

对应的⾮并发容器： ArrayList

⽬标：代替Vector、 synchronizedList

原理：利⽤⾼并发往往是读多写少的特性，对读操作不加锁，对写操作，先复制⼀份新的集合， 在新的集合上⾯修改，然后将新集合赋值给旧的引⽤，并通过volatile 保证其可⻅性，当 然写操作的锁是必不可少的了。

缺点：

内存占用问题：

如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去， 内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC

数据一致性问题：

只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的 数据，马上能读到，请不要使用CopyOnWrite容器

使用场景：

用于读多写少的并发场景。比如白名单，黑名单等场景。

◇CopyOnWriteArraySet

对应的⾮并发容器： HashSet

⽬标：代替synchronizedSet

原理：基于CopyOnWriteArrayList实现，其唯⼀的不同是在add时调⽤的是CopyOnWriteArrayList 的addIfAbsent⽅法，其遍历当前Object数组，如Object数组中已有了当前元素，则直接 返回，如果没有则放⼊Object数组的尾部，并返回。

◇ConcurrentSkipListMap

对应的⾮并发容器： TreeMap

⽬标：代替synchronizedSortedMap(TreeMap)

原理： Skip list（跳表）是⼀种可以代替平衡树的数据结构，默认是按照Key值升序的。

◇ConcurrentSkipListSet

对应的⾮并发容器： TreeSet

⽬标：代替synchronizedSortedSet

原理：内部基于ConcurrentSkipListMap实现

◇ConcurrentLinkedQueue

不会阻塞的队列

对应的⾮并发容器： Queue

原理：基于链表实现的FIFO队列（LinkedList的并发版本）

◇Queue

对应的⾮并发容器： Queue

特点：拓展了Queue，增加了可阻塞的插⼊和获取等操作

原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒

四种阻塞队列

•ArrayBlockingQueue：

（有界阻塞队列）基于数组、有界，按 FIFO（先进先出）原则对元素进行排序

•LinkedBlockingQueue：

（无界阻塞队列）基于链表，按FIFO（先进先出） 排序元素 吞吐量通常要高于ArrayBlockingQueue。Executors.newFixedThreadPool() 使用了这个队列

•SynchronousQueue：

（同步队列）不存储元素的阻塞队列 每个插入操作必须等到另一个线程调用移除操作，否 则插入操作一直处于阻塞状态 吞吐量通常要高于LinkedBlockingQueue Executors.newCachedThreadPool使用了这个队列

•PriorityBlockingQueue：

（优先级阻塞队列）具有优先级的、无限阻塞队列

▶**类加载的执行过程**

◇**加载（Loading）**

获取.class文件的二进制流

将类信息、静态变量、字节码、常量这些.class文件中的内容放入方法区中

◇**验证（Verification）**

确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求。文件格式验证 元数据验 证、字节码验证、符号引用验证

◇**准备（Preparation）**

准备阶段是正式为类变量分配内存并设置其初始值的阶段，这些变量所使用的内存都将在方法 区中分配。

**◇解析（Resolution）**

虚拟机将常量池中的符号引用替换为直接引用的过程。类或接口的解析、类方法解析、接口方 法解析、字段解析

◇**初始化（Initialization）**

初始化阶段就是执行类构造器()方法的过程

◇**使用（Using）**

◇**卸载（Unloading）**

▶**双亲委派**

◇**双亲委派是什么**

顶层父类加载器先加载class，无法加载此class，则会返回去交给子类加载器去尝试加载，若最 底层的子类加载器也没找到，则会抛出ClassNotFoundException。

◇**为啥要有双亲委派**

双亲委派确保只有JVM中有且仅有一份正常的java核心类。

◇**Jdbc为什么要破坏双亲委派模型？**

原生的JDBC中Driver驱动本身只是一个接口，并没有具体的实现，具体的实现是由不同数据库 类型去实现的。

原生的JDBC中的类需要由Application类加载器去进行类加载

◇**Tomcat为什么要破坏双亲委派模型？**

因为一个Tomcat可以部署N个web应用，但是每个web应用都有自己的classloader，互不干扰

比如web1里面有com.test.A.class，web2里面也有com.test.A.class，如果没打破双亲委派模型的 话，那么web1加载完后，web2在加载的话会冲突。

◇**如何破坏双亲委派模型**

重写ClassLoader下的loadClass方法，别重写findClass方法，因为loadClass是核心入口，将其 重写成自定义逻辑即可破坏双亲委派模型

▶**Jvm内存结构组成**

◇**Java堆（Heap）**

线程共享

Java虚拟机所管理的内存中最大的一块

目的就是存放对象实例

◇**JVM栈（JVM Stacks）**

线程私有

存储

局部变量表

操作栈

动态链接

方法出口

◇**方法区（Method Area）**

线程共享

存储

类信息

常量

静态变量

即时编译器编译后的代码

◇**程序计数器（Program Counter Register）**

线程私有

当前线程所执行的字节码的行号指示器

用来存放下一条指令的地址的。当执行一条指令时，首先需要根据PC中存放的指令地址，将指令由内存取到指令寄存器中，此过程 称为“取指令”。与此同时，PC中的地址或自动加1或由转移指针给出下一条指今的地址。此后经过分析指令，执行指令。完 成第一条指令的执行，而后根据PC取出第二条指令的地址，如此循环，执行每一条指令！

◇**本地方法栈（Native Method Stacks）**

线程私有

存储虚拟机使用到的Native方法

▶**元空间和永久代区别**

◇**存储位置不同**

永久代在物理上是堆的一部分

元空间属于本地内存

◇**存储内容不同**

在原来的永久代划分中，永久代用来存放类的元数据信息、静态变量以及常量池等

现在类的元数据信息存储在元空间中，静态变量和常量池等并入堆中

▶**Java8为什么要将永久代替换成Metaspace**

◇字符串存在永久代中，容易出现性能问题和内存溢出。

◇类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代 溢出，太大则容易导致老年代溢出。

◇永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

▶**新生代与老年代**

◇新生代和老年代的大小比列 1:2

◇eden、s0和s1比例 8:1:1

◇对象如何从eden、s0、s1到老年代过程

对象很大，直接在老年代创建对象。否则在eden区创建对象。后分析eden区、s0区、s1 区对象存活，如果存活，对象年龄+1，复制到s0区，否则清除； 新的一批对象在eden区创建， 再分析eden区、s0区、s1区对象存活，如果存活，对象复制到s1区，否则清除；如此往复对 象被复制15次依然存活，对象移植老年代，老年代采用的标记清除算法清除对象。

▶**跨代引用**

◇新生代中存在对老年代对象的引用，或者老年代中存在对新生代的引用

◇存在问题

YGC时，为了找到年轻代中的存活对象，不得不遍历整个老年代

◇解决方案

记忆集

◇记忆集

记录着收集区域被其他区域引用的数据（地址，对象，内存段）

缺点

•滞后性

•浪费一定的空间

•YGC时实际上对象E可以被回收，但是由于没发生FGC，老年代中的对象D仍存在对对象 E的引用，导致E无法被回收。

◇卡表

•记忆集的具体实现

•一个简单的字节数组结构

•数组的索引表示的是收集区域的一段内存区域，而元素的值就代表着对应的地址上是否被其他 区域所引用

▶**引用类型**

◇强引用

把一个对象赋给一个引用变量，引用变量为强引用；只要强引用存在，被引用的对象不被 回收

◇软应用

用 SoftReference类来实现；内存足够时不被回收，内存不足时被回收。

◇弱引用

用 WeakReference 类来实现；垃圾回收机制开始，总会回收该对象所占的内存。

弱引用 WeakHashMap了解下

◇虚引用

用 PhantomReference类来实现；它不能单独使用，必须和引用队列联合使用。虚 引用的主要 作用是跟踪对象被垃圾回收的状态。

▶**对象存活的判断机制**

◇**引用计数**

给对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引 用计数为 0 的对象可被回收。 （不怎么使用，堆内对象之间彼此引用没办法回收）

◇**可达性分析**

通过 GC Roots 作为起始点进行搜索，可达对象为可用，否则对象为不可达。不可达对象变为可 回收对象至少要经过两次可回收标记过程。

▶**几种垃圾回收算法**

◇**标记-清除**

将需要回收的对象进行标记，然后清除。

不足：效率不高，产生大量碎片，无法给大对象分配内存

◇**复制**

将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象 复制到另一块上面，然后再把使用过的内存空间进行一次清理。

不足：只是用了内存的一半

改进：分为 8：1：1 的 Eden 区和 survivor 区， YGC

◇**标记-整理**

让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

◇**分代收集**

一般将 Java 堆分为新生代和老年代。

新生代使用：复制算法

老年代使用：标记 - 清理 或者 标记 - 整理 算法

▶**什么是类加载器，类加载器有哪些?**

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)：

用来加载 java 核心类库，无法被 java 程序直接引用。

1. 扩展类加载器(extensions class loader):

它用来加载 Java 的扩展库。Java 虚拟机的实 现会提供一个扩展库目录。该类加载器在此目录里面 查找并加载 Java 类。

1. 系统类加载器（system class loader）：

它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

1. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

▶**垃圾收集器**



▶**CMS 收集器**

◇CMS（concurrent mark sweep）是以获取最短垃圾收集停顿时间为目标的收集器，CMS 收集器的关注点尽可能缩短 垃圾收集时用户线程的停顿时间,停顿时间越短就越适合与用户交互的程序,目前很大一部分的java应用几种在互联网 的B/S系统服务器上，这类应用尤其注重服务器的响应速度，系统停顿时间最短，给用户带来良好的体验，CMS收集器使用的算法是标记-清除算法实现的

◇CMS垃圾收集过程 整个过程4个步骤

初始标记

并发标记

重新标记

并发清除

其中 初始标记 和 重新标记 都需要stopTheWorld

◇缺点

CPU资源非常敏感

无法处理浮动垃圾

可能出现"Concurrent Mode Failure"失败而导致另一次Full GC的产生。

空间碎片

CMS是一款基于标记-清除算法实现的收集器，所有会有空间碎片的现象

▶**G1收集器**

◇G1是一款面向服务端应用的垃圾收集器，主要针对配备多核CPU及大容量内存的机器，以极高概 率满足 GC停顿时间的同时，还兼具高吞吐量的性能特征

◇G1收集器特点

•独特分区结构和算法

○G1把内存划分为多个独立的区域Region

○G1仍然保留分代思想,保留了新生代和老年代,但他们不再是物理隔离,而是一部分Region 的集合

○G1跟踪各个Region里面垃圾的价值大小,会维护一个优先列表,每次根据允许的时间来回 收价值最大的区域,从而保证在有限事件内高效的收集垃圾

•并行性

G1能够充分利用多CPU、多核环境硬件优势，尽量缩短STW

•内存整理

G1整体整体采用标记整理算法,局部是采用复制算法,不会产生内存碎片

•可预⻅性

G1的停顿可预测,能够明确指定在一个时间段内,消耗在垃圾收集上的时间不超过设置时间

◇Region区域



①使用G1收集器时，它将整个Java堆划分成约2048个大小相同的独立Region块，每个Region 块大小根据堆空间的实 际大小而定，为2的N次幂，即1MB, 2MB, 4MB, 8MB, 16MB,32MB。

② 虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了,它们都是一 部分Region (不需要连 续)的集合。通过Region的动态分配方式实现逻辑上的连续。

③ G1垃圾收集器还增加了一种新的内存区域，叫做Humongous内存区域，如图中的H块。主 要用于存储大对象，如 果超过1 .5个region,就放到H。一般被视为老年代

◇G1提供了两种GC模式

•Young GC和Mixed GC，两种均是完全Stop The World的。

•Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小， 来控制young GC 的时间开销。

•Mixed GC：选定所有年轻代里的Region，外加根据global concurrent marking统计得出收集收益 高的若干老年代 Region。在用户指定的开销目标范围内尽可能选择收益高的老年代Region。

◇G1垃圾回收四个阶段

初始标记 :和CMS一样只标记GC Roots直接关联的对象

并发标记 :进行GC Roots Traceing过程

最终标记 :修正并发标记期间,因程序运行导致发生变化的那一部分对象

筛选回收 :根据时间来进行垃圾价值最大化收集

▶**GC性能指标**

◇吞吐量

即CPU用于运行用户代码的时间与CPU总消耗时间的比值（吞吐量 = 运行用户代码时间 / ( 运 行用户代码时 间 + 垃圾收集时间 )）。

例如：虚拟机共运行100分钟，垃圾收集器花掉1分钟，那么吞吐量就是99%

◇暂停时间

执行垃圾回收时，程序的工作线程被暂停的时间

◇内存占用

Java堆所占内存的大小

◇收集频率

垃圾收集的频次

▶**堆GC收集器**

◇部分收集器

新生代收集（Minor GC / Young GC）: 只是新生代的垃圾收集

老年代收集 （Major GC / Old GC）: 只是老年代的垃圾收集 (CMS GC 单独回收老年代)

◇整堆收集（Full GC）

收集整个java堆和方法区的垃圾收集器

◇年轻代GC触发条件

年轻代空间不足,就会触发Minor GC， 这里年轻代指的是Eden代满，Survivor不满不会引发GC。 Minor GC会引发STW(stop the world) ,暂停其他用户的线程,等垃圾回收接收,用户的线程才恢复.

◇老年代GC(Major GC)触发机制

老年代空间不足时,会尝试触发MinorGC. 如果空间还是不足,则触发Major GC

如果Major GC , 内存仍然不足,则报错OOM

Major GC的速度比Minor GC慢10倍以上.

◇FullGC 触发机制

调用System.gc()

老年代空间不足

方法区空间不足

通过Minor GC进入老年代平均大小大于老年代可用内存

▶**GC调优策略**

◇监控GC的状态

使用各种JVM工具，查看当前日志，分析当前JVM参数设置，并且分析当前堆内存快照和gc 日志，根据实际的各区域内存划分和GC执行时间，觉得是否进行优化。

◇生成堆的dump文件，分析结果，判断是否需要优化

Minor GC执行时间不到50ms；

Minor GC执行不频繁，约10秒一次；

Full GC执行时间不到1s；

Full GC执行频率不算频繁，不低于10分钟1次；

◇Full GC 的成本远高于 Minor GC，尽可能将对象分配在新生代

“-Xmn”命令调节新生代大小，尽可能将对象分配在新生代；

-XX:PretenureSizeThreshold 可以设置直接进入老年代的对象大小；

-XX:MaxTenuringThreshold 设置对象进入老年代的年龄大小，减少老年代的内存占用，降低 full gc 发生的频率

▶**对象什么时候会进入老年代**

◇动态年龄的判断机制，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一 半，年龄大于或等于该年龄的对象就可以直接进入老年代。

◇在新生代中对象经过15次复制算法后会进入老年代

▶**频繁FullGC怎么排查？**

jstat -gcutil或者查看gc日志，查看内存回收情况

dump内存文件在具体分析，比如通过jmap命令jmap -dump:format=b,file=dumpfile pid，导出之后再 通过Eclipse Memory Analyzer等工具进行分析，定位到代码，修复

▶**CPU占用过高问题排查**

◇定位cpu过高进程

top

ps aux | grep PID

◇定位线程

ps -mp pid -o THREAD,tid,time | sort -rn

◇将需要的线程ID转换为16进制格式

printf "%x\n" tid

◇打印线程堆栈信息

jstack pid | grep tid -A 30

用户线程

通过该线程的堆栈信息查看其具体是在哪处用户代码处运行比较消耗CPU

Jvm线程

jstat -gcutil 进程号 统计间隔毫秒 统计次数（缺省代表一致统计） 命令监控当前系统的 GC状况

jmap -dump:format=b,file=filename 进程ID，导出某进程下内存heap输出到文件中。

▶**JMM内存模型**

◇什么是Java内存模型

所有的变量都存储在主内存中

线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都 必须在工作内存中进行

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作 内存和主存之间进行数据同步进行。

◇重排序类型

编译器重排序。 对于没有先后依赖关系的语句，编译器可以重新调整语句的执行顺序。

CPU指令重排序。 在指令级别，让没有依赖关系的多条指令并行。

CPU内存重排序。 指令的执行顺序和写入主内存的顺序不完全一致。

◇happen-before·

•如果A happen-before B，意味着A的执行结果必须对B可见，也就是保证跨线程的内存可见性。 A happen before B不代表A一定在B之前执行。因为，对于多线程程序而言，两个操作的执行 顺序是不确定的。happen-before只确保如果A在B之前执行，则A的执行结果必须对B可见。 定义了内存可见性的约束，也就定义了一系列重排序的约束。

•底层实现原理

内存屏障

禁止编译器重排序

禁止CPU指令重排序

禁止CPU内存重排序

◇volatile

•保证可见性，不保证原子性，禁止指令重排序

•作用

可见性

保证不同线程对变量操作的内存可见性

有序性

禁止指令重排序

•volatile如何保证内存可见性

被volatile修饰的变量一旦发生改变，会立即将缓存中的数据写入到内存当中。

其他cpu处理器通过嗅探在总线上传播的数据来查看自己的缓存中的数据是否过期，如果 过期，就会将缓存中对应的数据置为无效。

当处理器要对这个数据进行修改操作的时候，会重新从内存中将数据读取到缓存中。

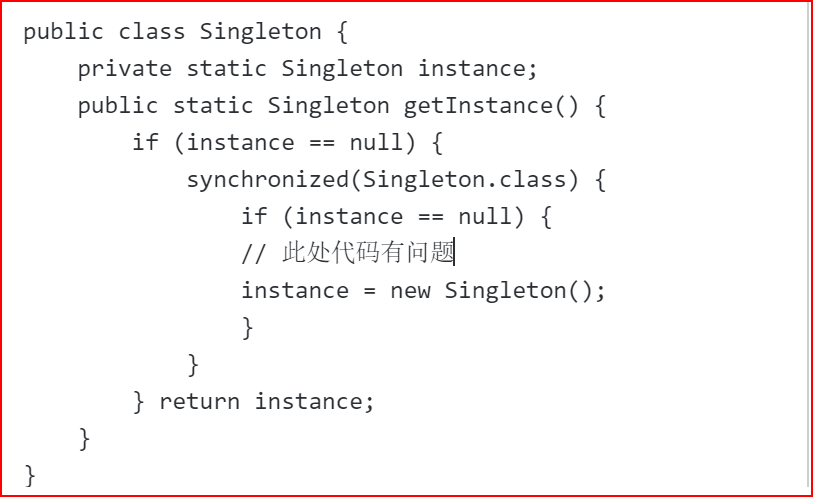
•volatile如何保证指令重排序

编译器和处理器都会执行指令重排的优化，如果在指令间插入一条内存屏障，那么内存屏障前 后的指令会被禁止重排序优化. 对volatile变量进行写操作时，会在写操作后面加入一条store 屏障指令，对volatile变量进行读操作时，会在读操作前加入一条load屏障指令。

◇DCL问题

何为DCL，DCL即Double Check Lock，双重检查锁定

示例：



上述的 instance = new Singleton(); 代码有问题：其底层会分为三个操作：

分配一块内存。

在内存上初始化成员变量。

把instance引用指向内存

操作2和操作3可能重排序，即先把instance指向内存，再初始化成员变量，因为二者并没有先 后的依赖关系。

volatile的三重功效：64位写入的原子性、内存可见性和禁止重排序。

**JAVA多线程**

▶**守护线程**

Jvm中所有的用户线程结束，守护线程结束。主要给用户线程提供服务

注意：守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候可能发生中断

▶**shutdown()与shutdownNow()的区别**

清空队列

shutdown()不会清空任务队列，会等所有任务执行完成

shutdownNow()清空任务队列,返回任务集合

中断线程

shutdown()只会中断空闲的线程

shutdownNow()会中断所有线程

▶**ForkJoinPool**

ForkJoinPool就是JDK7提供的一种“分治算法”的多线程并行计算框架。Fork意为分叉，Join意为合 并， 一分一合，相互配合，形成分治算法。此外，也可以将ForkJoinPool看作一个单机版的 Map/Reduce， 多个线程并行计算。

▶**工作窃取队列**

所谓工作窃取算法，是指一个Worker线程在执行完毕自己队列中的任务之后，可以窃取其他线程队 列中的任务来执行，从而实现负载均衡，以防有的线程很空闲，有的线程很忙。

如下几个操作

Worker线程在队列头部，通过对top指针执行加、减操作，实现入队或出队，这是单线程的。

其他Worker线程，在队列尾部，通过对base进行累加，实现出队操作，也就是窃取，这是多线 程的，需要通过CAS操作。

▶**ThreadLocal**

◇原理

线程调用全局ThreadLocal对象的set方法，相当于线程内部的ThreadLocalMap里面put一条记 录，key就是ThreadLocal对象,而value就是各自线程通过set方法传进去的值。

◇ThreadLocal为什么要使用弱引用

ThreadLocalMap中的key为一个threadlocal实例，如果key使用强引用，那么这个threadlocal 就会因为和entry存在强引用无法被回收！造成内存泄漏 ，除非线程结束，线程被回收了，map 也跟着回收。

Thread ThreadLocalMap Entry ThreadLocal

◇应用场景

数据库连接

Session管理

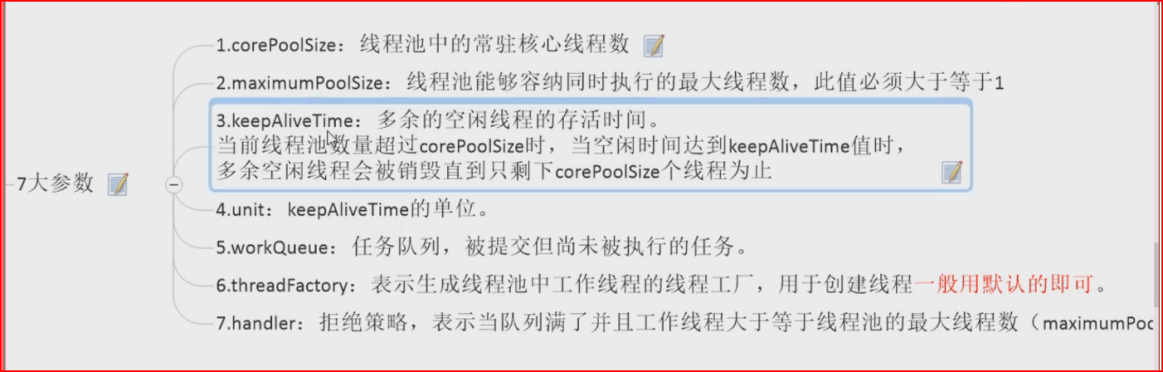
◇使用方法

每次使用完ThreadLocal都调用它的remove()方法清除数据

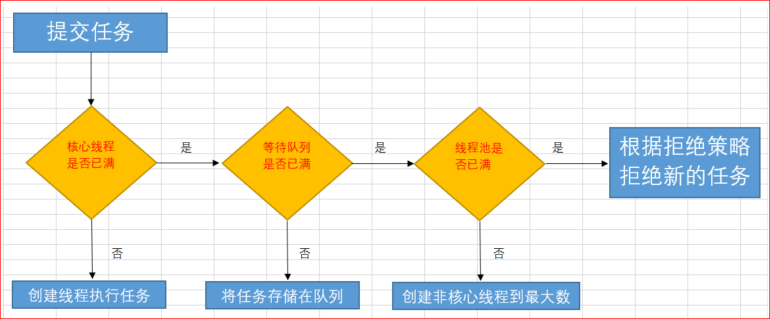
将ThreadLocal变量定义成private static，这样就一直存在ThreadLocal的强引用，也就能保证任 何时候都能通过ThreadLocal的弱引用访问到Entry的value值，进而清除掉 。

▶**线程池**

◇线程池七大参数



◇原理



如果线程池的poolsize还没有达到corePoolSize，那么就新增加一个线程处理新提交的任务；

如果poolsize已经达到了corePoolSize，就将新提交的任务提交到阻塞队列排队

如果队列容量已达上限，并且poolsize没有达到maximumPoolSize，那么就新增线程来处理任务；

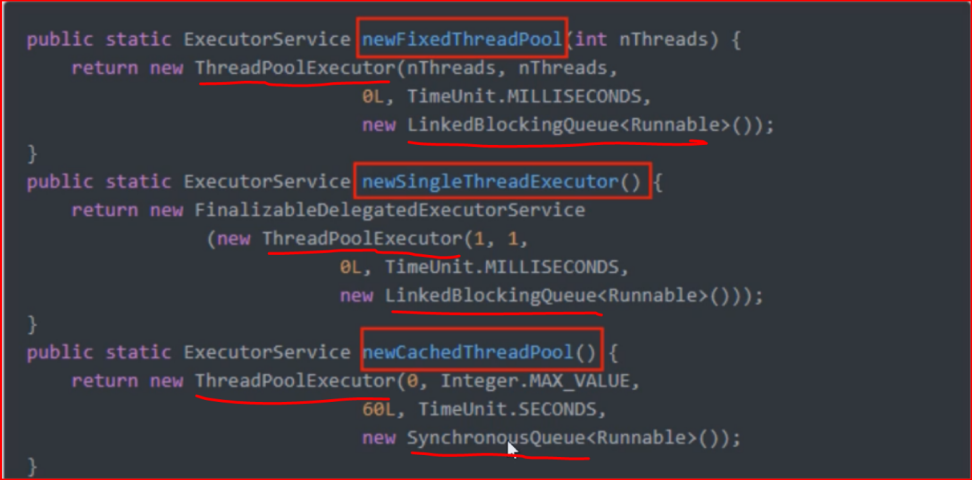
如果队列已满，并且当前线程数目也已经达到上限，此时需要拒绝新增加的任务。至于如何拒 绝处理新增的任务，取决线程池的饱和策略RejectedExecutionHandler。

如果线程无事可做，超过时间（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。

◇线程池怎么实现线程复用

通过取 Worker 的 firstTask 或者 getTask 方法从 workQueue 中取出了新任务，并直接调用 Runnable 的 run 方法来执行任务，每个线程都始终在一个大循环中，反复获取任务，然后执行 任务，从而实现了线程的复用。

◇几种java提供的线程池介绍



newFixedThreadPool：固定大小线程池，返回ThreadPoolExecutor，参数LinkedBlockingQueue

newSingleThreadExecutor：单线程化线程池，返回ThreadPoolExecutor，参数LinkedBlockingQueue

newCachedThreadPool：可回收缓存线程池，返回ThreadPoolExecutor，参数SynchronousQueue

◇不建议使用线程池默认的实现

newFixedThreadPool和newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至oom。

newCachedThreadPool和newScheduledThreadPool:

主要问题是线程数最大数是Integer.MAX\_VALUE，可能会创建非常多的线程，甚至oom

◇ThreadPoolExcutor构造参数介绍

corePoolSize, //核心线程的数量

maximumPoolSize, //最大线程数量

keepAliveTime, //超出核心线程数量以外的线程空余存活时间

TimeUnit , //存活时间的单位

BlockingQueue, //保存待执行任务的队列

ThreadFactory , //创建新线程使用的工厂

RejectedExecutionHandler // 当任务无法执行时的处理器

◇线程池四种拒绝策略

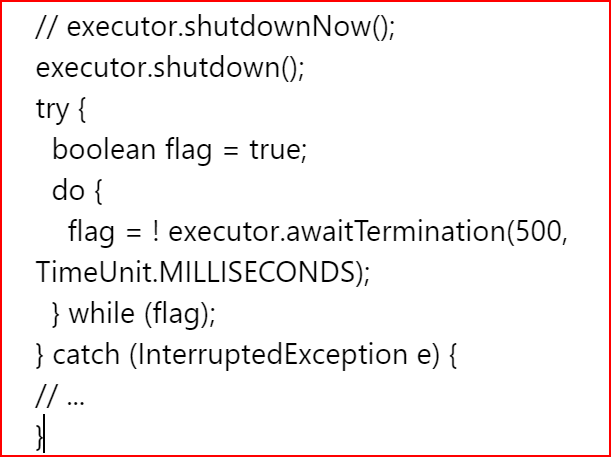
AbortPolicy：直接抛出异常。

CallerRunsPolicy：只用调用者所在线程来运行任务。

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

DiscardPolicy：不处理，丢弃掉。

◇正确关闭线程池的步骤



◇线程池配置合理线程数（maximumPoolSize）

一般公司都会选择自定义线程池

1. 代码查看公司服务器或阿里云是几核的

System.out.println(Runtime.getRuntime().availableProcessors());

1. 合理线程数配置之CPU密集型

一般公式：CPU核数+1个线程的线程池

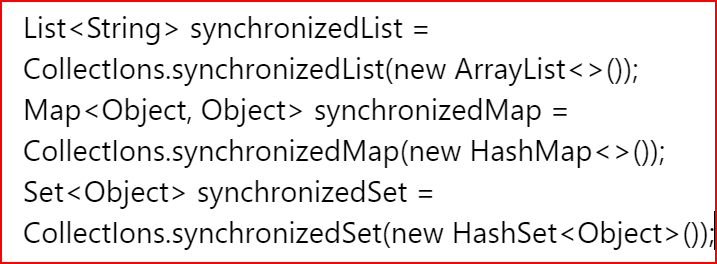
1. 合理线程数配置之IO密集型

方法一：CPU核数\*2

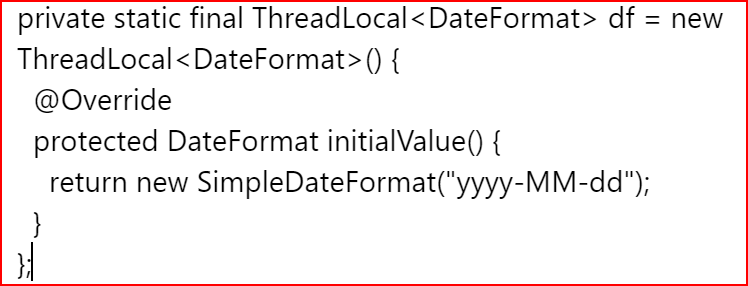
方法二：CPU核数/（1-阻系数）

阻塞系数在0.8~0.9之间

◇如何让线程不安全的集合类安全

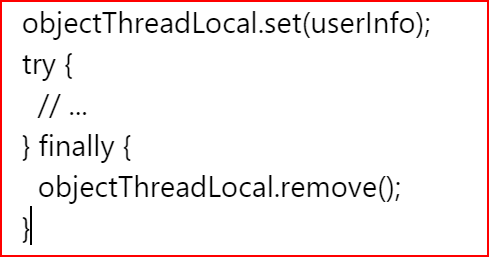


◇SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为static，必须加锁， 或者使用 DateUtils 工具类。



如果是 JDK8 的应用，可以使用 Instant 代替 Date， LocalDateTime 代替 Calendar， DateTimeFormatter 代替 SimpleDateFormat，官方给出的解释： simple beautiful strong immutable thread-safe。

◇必须回收自定义的 ThreadLocal 变量，尤其在线程池场景下，线程经常会被复用，如果不清理自定 义的 ThreadLocal变量，可能会影响后续业务逻辑和造成内存泄露等问题。尽量在代理中使用 try-finally 块进行回收。



**JAVA锁机制**

▶**CAS**

◇实现原理

当前值等于期待值，则更改值成为当前值

◇缺点

1、ABA问题

一个变量赋值A，后赋值B，再后改为A，CAS是认为这个变量没有变化的

解决：AtomicStampReference，使用时间戳，解决cas中出现的ABA问题

2、循环时间长，CPU开销大

3、只能保证一个共享变量的原子操作

◇底层原理

自旋：比较并交换，直到比较成功

底层靠Unsafe类保证原子性

◇AtomicStampedReference与AtomicReference的区别

AtomicStampedReference它内部不仅维护了对象值，还维护了一个时间戳。当 AtomicStampedReference对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳。当 AtomicStampedReference设置对象值时，对象值以及时间戳都必须满足期望值，写入才会成功。 因此，即使对象值被反复读写，写回原值，只要时间戳发生变化，就能防止不恰当的写入。

AtomicReference保证在修改对象引用时的线程安全性

▶**AQS**

◇抽象的队列式的同步器AbstractQueuedSynchronizer

◇AQS内部有3个对象，一个是state（用于计数器，类似gc的回收计数器），一个是线程标记（当 前线程是谁加锁的），一个是阻塞队列。

◇定义了两种资源方式

Exclusive（独占，只有一个线程能执行，如 ReentrantLock）

Share（共享，多个线程可同时执行，如 Semaphore/CountDownLatch）。

◇同步器的实现

以 ReentrantLock为例，初始state为0，未锁定状态。A线程lock时state+1；其他线程lock失 败，直到A线程unlock，state=0，其它线程才有机会获取该锁。当然，释放锁前A线程可以多 次获取锁，获取多少次就要释放多么次，保证state回到零态。

以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N。每个子线程执 行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark() 主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。