# 自我介绍

# 基础

# 集合

## Java中常用的集合有哪些？

**常用的集合主要在java.util包下面：**

主要接口有List,Set,Map,Queue,Deque,SortedMap,SortedSet

List:

线程不安全：ArrayList, LinkedList, AttributeList, RoleList

线程安全：CopyOnWriteArraylist ,Stack, Vector

Set:

线程不安全：EnumSet, HashSet, LinkedHashSet, TreeSet

线程安全：ConcurrentSkipListSet, CopyOnWriteArraySet

Map:

线程不安全：EnumMap, HashMap, IdentityHashMap, LinkedHashMap, Properties, TreeMap

线程安全：ConcurrentHashMap, ConcurrentSkipListMap, Hashtable

SortedMap继承于map:

ConcurrentSkipListMap, TreeMap

SortedSet继承于set:

ConcurrentSkipListSet, TreeSet

queue:

1.ArrayDeque, （数组双端队列）

2.PriorityQueue, （优先级队列）

3.ConcurrentLinkedQueue, （基于链表的并发队列）

4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）

5.ArrayBlockingQueue, （基于数组的并发阻塞队列）

6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）

8.PriorityBlockingQueue, （带优先级的无界阻塞队列）

9.SynchronousQueue （并发同步阻塞队列）

Deque:

ArrayDeque, LinkedBlockingDeque, LinkedList

## Java的数据结构相关的类实现原理？

Java常用的数据包括数组，链表，树。有些类是基于单一的数据结构实现的，比如ArrayList，Vector是基于数组实现的，LinkedList是基于链表实现的。有些类是基于复合数据结构实现的，比如HashMap是基于数组+链表或数组+树的数据结构实现的，LinkedHashMap更是在HashMap的基础上再加上一层链表数据结构来维护集合的顺序。

## 讲讲自己对HashMap的理解？

HashMap是以key-value键值对存储数据的集合，JDK1.7以前，HashMap底层使用的是数组+单向链表的数据结构（在数据结构中，一般称之为“链表散列“），其主要包含put，get方法，put方法用于存储数据，get用于获取数据。当使用put方法时，首先会根据key的hashcode与数组大小长度-1进行逻辑与运算，也就是通过求模计算出该数据在数组中存储的位置。遍历该位置下的链表，使用equals判断这个链表是否包含这个key，如果存储则替换该entry对象的值，并将旧的value返回。如果不存在则将新数据放在链表的首位。存储完成以后会判断数组的大小是否超过阀值，超过则对数组进行扩容，并遍历整个hashmap，将数据放入新的数组中。get方法则相对简单，只是通过hashcode找到数组对应位置，遍历数组，使用equals方法查找是否包含对应的值，有则返回，无则返回null。

JDK1.8之后HashMap数据结构做了改动，在链表数据达到8个时会转换为使用红黑树的数据结构存储，put，get方法也做了相应的改动。put方法存储数据时会先判断当前数据结构是否为红黑树，是则通过红黑树的方式存储数据，而且扩容的时候也有些改变，1.7是遍历整个数组，并将链表的数据一个一个重新计算hashcode到新的数组中，但1.8则是先生成新的两个链表，再放到新的数组中。

**hashmap扩容过程，1.7 1.8的区别**

1.7

Index的计算方式：return h & (length-1);

扩容方式：循环遍历所有的node节点，再一个个获取位置存入新的数组，扩容时也是扩为原来的两倍。

1.8

Index的计算方式 : h & (n-1)

扩容时方式：遍历数组，将每个桶先分成两个桶再插入到新的数组中

1.8新增：h = (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);将高位参与运算是为了在table长度比较小时，减少碰撞。



如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞

## ？HashMap为什么用红黑树，红黑树的特点？

HashMap使用红黑树，无非也是为了加快检索速度。

## HashMap是否线程安全？

HashMap非线程安全的，如何线程安全的使用 HashMap。这个无非就是以下三种方式：Hashtable ConcurrentHashMap Synchronized Map

Synchronized Map：

SynchronizedMap：synchronizedMap() 方法后会返回一个 SynchronizedMap 类的对象，而在 SynchronizedMap 类中使用了 synchronized 同步关键字来保证对 Map 的操作是线程安全的。

## HashMap和HashTable的区别

Hashtable是线程安全的，基本上每个方法都使用了synchronized来保证线程安全性。

1、hashMap去掉了HashTable?的contains方法，但是加上了containsValue()和containsKey()方法

2、Hashtable不允许 null 值(key 和 value 都不可以)，HashMap允许 null 值(key和value都可以)。

3、两者的遍历方式大同小异，Hashtable仅仅比HashMap多一个elements方法。

4、HashTable使用Enumeration，HashMap使用Iterator

## ConcurrentHashMap 是如何实现的？

**1.7、1.8 实现有何不同？为什么这么做？**

base1.7

1.7仍然是数组加链表，相比于HashMap，ConcurrentHashMap数组与entry很多核心数据都使用了volatile修饰.原理上来说：ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理。因为volatile无法保证原子性， put 操作时仍然需要加锁处理。首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁。如果重试的次数达到了 MAX\_SCAN\_RETRIES 则改为阻塞锁获取，保证能获取成功。

由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

base1.8

1.8抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。也将 1.7 中存放数据的 HashEntry 改为 Node，但作用都是相同的。其中的 val next 都用了 volatile 修饰，保证了可见性。

## ConcurrentHashMap的size方法是如何实现的？

看源码是使用baseCount与CounterCell类进行累加，baseCount是一个volatile变量，ConcurrentHashMap的put方法后有个addCount方法会对这个变量做 CAS 加法。而CounterCel在没有并发的情况下，使用一个 baseCount volatile 变量就足够了，当并发的时候，CAS 修改 baseCount 失败后，就会使用 CounterCell 类了，会创建一个这个对象，通常对象的 volatile value 属性是 1。在计算 size 的时候，会将 baseCount 和 CounterCell 数组中的元素的 value 累加，得到总的大小，但这个数字仍旧可能是不准确的。因为都是并发插入的。

**如果你觉得size方法返回值不准确，那如果让你自己实现，你觉得应该怎么实现呢？**

Size是因为有并发插入操作才会使得其不准确，如果要让其准确返回，除非启用单线程做累加操作，或者可以用AtomicInteger变量进行记录。

## ConcurrentHashMap能完全替代HashTable吗？

HashTable虽然性能上不如ConcurrentHashMap，但并不能完全被取代，两者的迭代器的一致性不同的，HashTable的迭代器是强一致性的，而ConcurrentHashMap是弱一致的。 ConcurrentHashMap的get，clear，iterator 都是弱一致性的。 Doug Lea 也将这个判断留给用户自己决定是否使用ConcurrentHashMap。

那么什么是强一致性和弱一致性呢？

get方法是弱一致的，是什么含义？可能你期望往ConcurrentHashMap底层数据结构中加入一个元素后，立马能对get可见，但ConcurrentHashMap并不能如你所愿。换句话说，put操作将一个元素加入到底层数据结构后，get可能在某段时间内还看不到这个元素，若不考虑内存模型，单从代码逻辑上来看，却是应该可以看得到的。

下面将结合代码和java内存模型相关内容来分析下put/get方法。put方法我们只需关注Segment#put，get方法只需关注Segment#get，在继续之前，先要说明一下Segment里有两个volatile变量：count和table；HashEntry里有一个volatile变量：value。

总结:ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。

## HashMap和WeakHashMap的区别

HashMap的key是对实际对象的强引用,而WeakHashMap的key是对实际对象的弱引用。弱引用（WeakReference）的特性是：当gc线程发现某个对象只有弱引用指向它，那么就会将其销毁并回收内存。WeakReference也会被加入到引用队列queue中。

理解了相关概念之后，对WeakHashMap的实际应用感到很好奇。然后发现tomcat的源码里，实现缓存时会用到WeakHashMap。

## ArrayList与Vector的区别

ArrayList 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的。

## TreeMap和LinkedHashMap是如何保证它的顺序的？

TreeMap只能根据key来排序，是不能根据value来排序的，默认是根据key的自然排序来组织（比如integer的大小，String的字典排序），自定义排序的话需要实现Comparator的compare方法。

LinkedHashMap底层维护了一个链表来保证数据之间的顺序。

# Jvm

# 数据结构

# 锁

# 多线程

## 线程，进程

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。进程之间是隔离的，进程是拥有资源的基本单位， 线程是CPU调度的基本单位。

## Java中线程具有五种状态：

初始化；可运行；运行中；阻塞；销毁



但是，线程不同状态之间的转化是谁来实现的呢？是JVM吗？

并不是。JVM需要通过操作系统内核中的TCB（Thread Control Block）模块来改变线程的状态，这一过程需要耗费一定的CPU资源。

**一个线程会因为以下原因而放弃CPU：**

时间片用完了，java虚拟机让当前线程暂时放弃CPU,转到就绪状态,使其它线程获得运行机会。

当前线程因为某些原因而进入阻塞状态

线程结束运行

**线程在以下情况会停止：**

run方法正常执行完毕

run方法执行过程中抛出一个未捕获的异常

调用stop方法(不推荐使用)

**进程的停止，当一个进程中所有的前台线程停止后，该进程结束。**

**什么时候会出现僵死进程**

线程无法结束的时候会僵死进程，一般是出现死锁。

## 做过的项目介绍，项目中的线程池怎么实现的

案例：kafka消费时使用线程池接收数据。

对账的时候使用线程池操作。

**线程池使用：**

**Java中线程池的实现方式**

单个线程: Executors.newSingleThreadExecutor();

缓存线程: Executors.newCachedThreadPool();

固定线程Executors.newFixedThreadPool(2);

定时线程: Executors.newScheduledThreadPool(3);

**ThreadPoolExecutor**

ThreadPoolExecutor 继承 AbstractExecutorService；AbstractExecutorService 实现 ExecutorService， ExecutorService 继承 Executor

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {...}

corePoolSize : 核心线程数的大小

maximumPoolSize : 线程池中允许的最大线程数

keepAliveTime : 空闲线程允许的最大的存活时间

unit : 存活时间的单位

workQueue : 阻塞任务队列

threadFactory : 线程工厂用来创建线程

handler : 拒绝策略，针对当队列满了时新来任务的处理方式

**线程池的执行过程：**

在刚刚创建线程池的时候，内部线程的数量是 0，当首个任务进行添加的时候，会根据参数的配置进行线程的创建，并随着任务数的增加，会逐渐创建新的线程直到线程数达到核心线程的大小。这时，如果再添加任务会将任务放置到阻塞队列当中,如果队列装不下了这时会判断当前线程的数量是否达到了最大线程数，如果未达到，新添加的任务会由最大重新创建线程并立马执行。如果队列满了，而且已经达到最大线程数了，这时再添加任何就会由拒绝策略来处理，默认的拒绝策略是抛出异常。核心线程数不会被关闭，会一直等待任务到来。

**使用线程池的优点是什么**

虽然线程是个轻量级的东西， 但是对于互联网应用来说，如果每个用户的请求都创建一个线程，那会非常得多，服务器也是难于承受， 再说了，众多的线程去竞争CPU，不断切换，也会让CPU调度不堪重负，很多线程将不得不等待。

线程可以预先创建，当线程池的线程刚创建时，让他们进入阻塞状态：等待某个任务的到来。 如果任务来了，那就好办，唤醒其中一个线程，让它拿到任务去执行即可。

## 线程池原理

原文：<https://mp.weixin.qq.com/s/LExpStGw7XXp0M1E_T0yKA>

https://mp.weixin.qq.com/s/DJPuYVhRGRXkkUGMuDdrSw

根据源码可以发现整个线程池大致分为 3 个部分，1. 是创建 worker 线程，2. 添加任务到 workQueue; 3.worker 线程执行具体任务

线程池每个线程会包装成worker，用来循环执行take任务。

第一步是生产者，也就是任务提供者他执行了一个 execute() 方法，本质上就是往这个内部队列里放了一个任务。

之前已经创建好了的 Worker 线程会执行一个 while 循环 ---> 不停的从这个 内部队列里获取任务。(这一步是竞争的关系，都会抢着从队列里获取任务，由这个队列内部实现了线程安全。)

获取得到一个任务后，其实也就是拿到了一个 Runnable 对象(也就是 execute(Runnabletask) 这里所提交的任务)，接着执行这个 Runnable 的 run() 方法，而不是 start()，为什么线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？线程池这个场景中却恰好就是要利用它只是一个普通方法调用。如果我们往一个核心、最大线程数为 2 的线程池里丢了 1000 个任务，那么它会额外的创建 1000 个线程，同时每个任务都是异步执行的，一下子就执行完毕了。

所谓线程池本质是一个hashSet。多余的任务会放在阻塞队列中。

只有当阻塞队列满了后，才会触发非核心线程的创建。所以非核心线程只是临时过来打杂的。直到空闲了，然后自己关闭了。

线程池提供了两个钩子（beforeExecute，afterExecute）给我们，我们继承线程池，在执行任务前后做一些事情。

线程池原理关键技术：锁（lock,cas）、阻塞队列、hashSet（资源池）

**当线程池在执行一个任务时抛出未捕获异常，这个任务还会执行吗？**



右图放大后的那一块，也就是内部队列并没有其他线程往里边丢任务执行 execute() 方法。

而一旦发生未捕获的异常后， Worker1 被回收，顺带的它所调度的线程 task1（这个task1 也就是在执行一个 while 循环消费左图中的那个队列） 也会被回收掉。

新创建的 Worker2 会取代 Worker1 继续执行 while 循环从内部队列里获取任务，但此时这个队列就一直会是空的，所以也就是处于 Waiting 状态。

原文：<https://mp.weixin.qq.com/s/SpxYhfExN-GwTYflkhvKRg>

因此：这个任务不会再继续执行了

线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？

**线程池遇到未捕获异常会怎样？**

Java中线程执行的任务接口java.lang.Runnable如果 run() 方法中抛出了RuntimeExceptio将会怎么处理了？通常java.lang.Thread对象运行设置一个默认的异常处理方法：java.lang.Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler)而这个默认的静态全局的异常捕获方法是直接输出异常堆栈。当然，我们可以覆盖此默认实现，只需要一个自定义的java.lang.Thread.UncaughtExceptionHandler接口实现即可。

而在线程池中却比较特殊。默认情况下，线程池 java.util.concurrent.ThreadPoolExecutor 会Catch住所有异常， 当任务执行完成(java.util.concurrent.ExecutorService.submit(Callable))获取其结果 时(java.util.concurrent.Future.get())会抛出此RuntimeException。也就是说当我们想线程池 ThreadPoolExecutor(java.util.concurrent.ExecutorService)提交任务时， 如果不理会任务结果（Feture.get()），那么此异常将被线程池吃掉。可以通过继承ThreadPoolExecutor并覆盖ThreadPoolExecutor.afterExecute 方法，我们才能捕获到任务的异常(RuntimeException)。

线程池有未抓取异常时不会导致后续任务中断，但是没有异常信息的输出，难以定位问题，对此，有三种方案：

）自己封装一个RunnableWrapper、CallableWrapper，将传入的任务包裹其中，在run方法中try-catch，并统一对未抓取异常进行异常处理/输出；

）继承ThreadPoolExecutor的方式来创建线程池，重写其afterExecute(Thread, Throwable)方法，判断第二个参数非空时获得其异常信息，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

）创建线程池时，传入自定义的ThreadFactory，在其创建线程时，使用setUncaughtExceptionHandler(..)设置每个线程的未抓取异常处理句柄，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

**executor与submit的区别**

1、接收的参数不一样

2、submit有返回值，而execute没有

3、submit方便Exception处理：意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

## 线程间通信

**线程间有哪些通信方式？**

可以使用wait,notify等待或唤醒其它线程，可以使用join等待其它线程执行完毕，可以使用volatile标识的变量来当作共享变量，可以使用CountDownLatch，CyclicBarrier来等待唤起其它任务，使用Semaphore限制某段代码块的并发数。

**wait,notify为什么必须同步代码块执行的，否则会抛出IllegalMonitorStateException。**

因为使用wait,notify时，当前线程必须拥有对象监视器。使用wait方法时，线程释放对象监视器的所有权，等待直到另一个线程通过调用notify或notifyAll来通知等待对象监视器的线程们并唤醒。在某一时刻，只有一个线程占有对象监视器。该方法必须同步执行的，否则会抛出IllegalMonitorStateException。

其次，由于CPU执行是随机的，所以我们必须对流程进行管理，也就是同步，通过在同步块中并结合 wait 和 notify 方法，我们可以手动对线程的执行顺序进行调整。

比如：假设有2个线程，分别是生产者和消费者，他们有各自的任务。

1.1生产者检查条件（如缓存满了）-> 1.2生产者必须等待

2.1消费者消费了一个单位的缓存 -> 2.2重新设置了条件（如缓存没满） -> 2.3调用notifyAll()唤醒生产者

我们希望的顺序是： 1.1->1.2->2.1->2.2->2.3

但是由于CPU执行是随机的，可能会导致 2.3 先执行，1.2 后执行，这样就会导致生产者永远也醒不过来了！

**wait,notify的原理**

不知道，因为看源码都是使用了native方法，没去看了。

**Semaphore有什么作用？**

Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

**CyclicBarrier和CountDownLatch的区别？**

两个看上去有点像的类，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

（1）CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

（2）CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

（3）CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

**Volatile的作用**

使用Volatile修饰的变量具有可见性，在虚拟机内存中有主内存和工作内存的概念，每一条线程对应一个工作内存，一般线程需要使用某个变量时会先从主内存中复制该变量到工作内存中，操作完成后再写回主内存中，使用volatile修饰后，当一个线程修改了变量的值，新的值会立刻同步到主内存当中。而其他线程读取这个变量的时候，也会从主内存中拉取最新的变量值。使用volatile还能防止指令重排。

**Volatile的原理**

工作内存Work Memory其实就是对CPU寄存器和高速缓存的抽象，或者说每个线程的工作内存也可以简单理解为CPU寄存器和高速缓存。

那么当写两条线程Thread-A与Threab-B同时操作主存中的一个volatile变量i时，Thread-A写了变量i，那么：

Thread-A发出LOCK#指令

发出的LOCK#指令锁总线（或锁缓存行），同时让Thread-B高速缓存中的缓存行内容失效

Thread-A向主存回写最新修改的i

Thread-B读取变量i，那么：

Thread-B发现对应地址的缓存行被锁了，等待锁的释放，缓存一致性协议会保证它读取到最新的值

由此可以看出，volatile关键字的读和普通变量的读取相比基本没差别，差别主要还是在变量的写操作上。

**竞态条件**：当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

**CountDownLatch原理**

countDownLatch.await()方法时会当前线程就会进入了一个死循环当中，在这个死循环里面，会不断的进行判断，通过调用tryAcquireShared方法，不断判断我们上面说的那个计数器，看看它的值是否为0了，如果是为0的话，tryAcquireShared就会返回1，然后跳出循环。需要注意的是，说是在不停的循环，其实也并非在不停的执行for循环里面的内容，因为在后面调用parkAndCheckInterrupt（）方法时，在这个方法里面是会调用 LockSupport.park(this);，来禁用当前线程的。

**？CyclicBarrier原理**

其核心代码是 ReentrantLock 以及 Condition 的共享唤醒线程，多个线程竞争锁，保证计数器parties为原子操作，然后当parties执行为0时候，执行方法，此时所有阻塞的线程继续执行

**？Semaphore原理**

Semaphore有两种模式，公平模式和非公平模式。公平模式就是调用acquire的顺序就是获取许可证的顺序，遵循FIFO；而非公平模式是抢占式的，也就是有可能一个新的获取线程恰好在一个许可证释放时得到了这个许可证，而前面还有等待的线程。其内部是基于AQS的共享模式，AQS的状态表示许可证的数量，在许可证数量不够时，线程将会被挂起；而一旦有一个线程释放一个资源，那么就有可能重新唤醒等待队列中的线程继续执行。