# 自我介绍

12年毕业，14年的时候开始从事这一行。入行第一个接手的项目就是移动公司的软件应用商城，就是给用户下安卓应用的平台，当时做的是主页展示这一块，每到节假日会做一些营销活到，比如说玩个有些送流量这些。在这个公司做了两年多，然后跳槽外包到顺丰，进的IBU研发部，做一个代理商管理系统，这就是一个卖货系统，顺丰走不了的国际件都就是往国外发的货物基本上都通过这个系统向对应国家的货运公司下单，比如美国的UPS，英国的皇家邮政，下完单之后再打印他们的标签，再把货给他们走。后面就进来上一家公司，主要是给线下的商铺做贷款的。

# 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

## JVM虚拟机内存划分或者问内存模型

JDK1.7时JVM内存划分为程序计数器，虚拟机栈，本地方法栈，堆，方法区，JDK1.8将方法区移入到了直接内存中，称为元空间。

程序计数器主要有两个作用：字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。在多线程的情况下，程序计数器用于记录当前线程执行的位置。

虚拟机栈：用于描述Java方法执行的的状态，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

本地方法栈：虚拟机栈的作用非常相似，区别是：虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法。

堆：在虚拟机启动时创建，唯一目的就是存放对象实例，所有的对象实例以及数组都要在堆上分配。

方法区：是各个线程共享内存区域，用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

## Java堆是如何实现的？

新生代中由Eden和Survivor0，Survivor1组成，默认三者的比例是8：1：1，一般对象都在新生代的Eden区生成，新生代的回收机制采用复制算法，在MinorGC的时候，我们都留一个存活区用来存放存活的对象，真正进行的区域是Eden+其中一个存活区，当我们的对象时长超过一定年龄时（默认15，可以通过参数设置），将会把对象放入老生代，当然大的对象会直接进入老生代。老生代采用的回收算法是标记整理算法。

## 什么是Java内存模型

Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，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，而不能直接读写主内存。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。

而JMM就作用于工作内存和主存之间数据同步过程。他规定了如何做数据同步以及什么时候做数据同步。

## 双亲委派机制

**什么是双亲委派机制？**

双亲委派机制的前提是双亲委派模型，它要求除顶层启动类加载器外其余类加载器都应该有自己的父类加载器；类加载器之间通过复用关系来复用父加载器的代码。实现双亲委派模型的代码都集中在java.lang.ClassLoader的loadClass()方法中；首先会检查请求加载的类是否已经被加载过；若没有被加载过：递归调用父类加载器的loadClass()；父类加载器为空后就使用启动类加载器加载；如果父类加载器和启动类加载器均无法加载请求，则调用自身的加载功能。

**双亲委派模型的优点：**

Java类伴随其类加载器具备了带有优先级的层次关系，确保了在各种加载环境的加载顺序。

保证了运行的安全性，防止不可信类扮演可信任的类。

**什么情况下我们需要破坏双亲委派模型；**

双亲委派模型很好地解决了各个类加载器的基础类统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以被称为“基础”，是因为它们总是作为被调用代码调用的API。但是，如果基础类又要调用用户的代码，那该怎么办呢。

或者是由于用户对程序的动态性的追求导致的，例如OSGi的出现。在OSGi环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为网状结构。

## 对象在内存中的初始化过程，JVM 类加载机制

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。然后会对文件进行JVM规范校验，代码逻辑校验。校验通过以后，会在堆中为类的静态变量及成员变量分配内存，并将其初始化为默认值，接着执行构造代码块，对对象进行初始化，在堆中为类的对象分配内存。

## 类加载器

– Java 虚拟机自带的加载器

根类加载器（Bootstrap，使用C++编写，程序员无法在java代码中获得该类）

扩展类加载器（Extension，使用java代码实现）

系统类加载器（System，使用java代码实现）

– 用户自定义的类加载器

必须继承java.lang.ClassLoader类

用户可以定制类的加载方式

## JVM垃圾收集算法

1 引用计数

引用计数器采用分散式管理方式，通过计数器记录对象是否被引用。当计数器为0时，说明此对象已经不再被使用，可进行回收

2**复制（Copying）、**

**3标记-清除（Mark-Sweep）**

**标记失去引用的对象并回收，这个算法会产生碎片**

**4标记-整理（Mark-Compact）**

在标记清除的基础上整理内存空间

**计算对象是否引用**

**1.采用标记计数的方法：**给内存中的对象给打上标记，对象被引用一次，计数就加1，引用被释放了，计数就减一，当这个计数为0的时候，这个对象就可以被回收了。当然，这也就引发了一个问题：循环引用的对象是无法被识别出来并且被回收的。所以就有了第二种方法：

**2.采用根搜索算法：**从一个根出发，搜索所有的可达对象，这样剩下的那些对象就是需要被回收的判断完了哪些对象是没用的，这样就可以进行回收了

## 垃圾收集器

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收

答： 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

新生代的垃圾回收器包括Serial、ParNew、Parallel Scavenge，老年代的垃圾回收器包括CMS、Serial Old、Parallel Old。其中新生代的三种垃圾回收器都采用了复制算法，并且会触发STOP THE WORLD。CMS使用标记清除算法，Serial Old、Parallel Old使用标记整理算法。

**Serial收集器**是一个单线程收集器（-XX：+UseSerialGC），这个“单线程”不只是说它只会使用一个CPU或者一条线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它垃圾收集结束。

**ParNew收集器**是Serial收集器的多线程版本（-XX:+UseParNewGC -XX:ParallelGCThreads），它能够与CMS收集器配合工作，因此，在运行在Server模式下的虚拟机中，ParNew收集器是首选的新生代收集器。

这也是一个并行的新生代垃圾收集器，不同于其他收集器（以尽可能缩短垃圾收集时用户线程的停顿时间为目的），它是唯一一个以达到一个可控制的吞吐量为目标的垃圾收集器。

CMS收集器的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。

重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。

应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。

挑战

当我们在真实的应用中使用CMS收集器时，我们会面临两个主要的挑战，可能需要进行调优：

堆碎片;对象分配率高

CMS收集器并没有任何碎片整理的机制。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。

G1算法：

G1把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；

增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时 把之前用掉的部分中的存活对象再放到后面没有用的空间中

回收步骤：

初始标记（Initial Marking） stop the world

并发标记（Concurrent Marking）

最终标记暂停（Final Marking Pause） stop the world

存活对象计算及清除（Live Data Counting and Cleanup）

参考：<https://my.oschina.net/u/2302503/blog/1632775>

## GC机制和原理

**GC分哪两种， Minor GC 和Full GC有什么区别？什么时候会触发Full GC？**

Minor GC:

Eden区域满了，或者新创建的对象大小 > Eden所剩空间

当JVM无法为一个新的对象分配空间时会触发Minor GC，比如当Eden区满了。

执行Minor GC操作时，不会影响到永久代。

质疑常规的认知，所有的Minor GC都会触发“全世界的暂停(stop-the-world)”，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。

Major GC:清理永久代，但是由于很多MojorGC 是由MinorGC 触发的，所以有时候很难将MajorGC 和MinorGC区分开。

FullGC：是清理整个堆空间—包括年轻代和永久代。Full GC的时候会先触发Minor GC。

新生代直接晋升到老年代的大对象超过了老年代的剩余空间，引发Full GC。

Perm永久代空间不足会触发Full GC，可以让CMS清理永久代的空间。

System.gc()引起的Full GC，可以设置DisableExplicitGC来禁止调用System.gc引发Full GC

**分别采用什么算法？**

Eden跟Survivor0,Survivor1采用的垃圾回收方式是著名的**“停止-复制（Stop-and-copy）”**清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

老年代用的算法是标记-整理算法（串行收集器）

关于方法区即永久代的回收，永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的索引实例都已经被回收。2.加载类的ClassLoader已经被回收。3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

**YGC的时候，有些对象可能会发生晋升，如果晋升失败怎么处理？**

标记阶段时，会把对象和对应的对象头数据保存在两个栈中，如果晋升失败的话，就把该对象的对象头复原，然后提早进行一次老年代的回收，防止下次进行YGC的时候发生晋升失败

## JVM调优

调优工具：Jconsole,jProfile,visualVM

如何调优：

堆信息查看：

可查看堆空间大小分配（年轻代，年老代，持久代）

提供即时的垃圾回收功能

垃圾监控

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

年老代年轻代大小划分是否合理

内存泄漏

垃圾回收算法设置是否合理

查看FullGC的频率

Linux使用jstat命令查看jvm的GC情况，通常运行命令如下：

jstat -gc 12538 5000

即会每5秒一次显示进程号为12538的java进成的GC情况

性能调优建议： jvm调优没有一个固定模板配置说必须如何操作，它需要根据系统的情况不同对待。 但是可以有如下建议：

1、初始化内存和最大内存尽量保持一致，避免内存不够用继续扩充内存。最大内存不要超过物理内存，例如内存8g，你可以设置最大内存4g/6g但是不能超过8g否则加载类的时候没有空间会报错。

2、gc/full gc频率不要太高、每次gc时间不要太长、根据系统应用来定。

# 算法和数据结构

## 查找算法

有序数据查找：顺序查找算法，二分查找又称折半查找，插值查找，斐波那契查找

插值查找的核心思想就是基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。

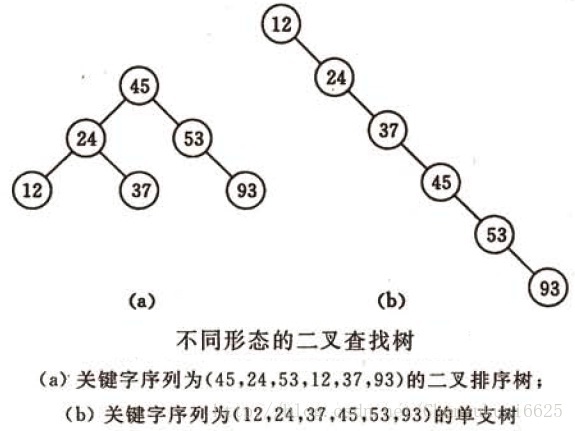
斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…，斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。

无序队列查找：索引查找（稠密索引，分块索引，倒排索引 ），树表查找（二叉树查找算法，平衡查找树之2-3查找树，平衡查找树之红黑树，B树和B+树）

## 树

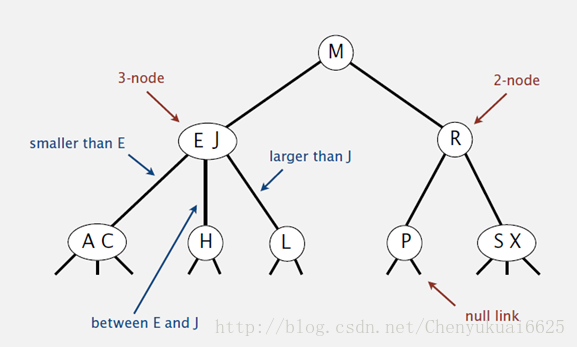
**二叉树查找**

有没有一种既可以使得插入和删除效率不错，又可以比较高效的实现查找的算法？这是有的，二叉树查找算法就可以实现这样的功能。



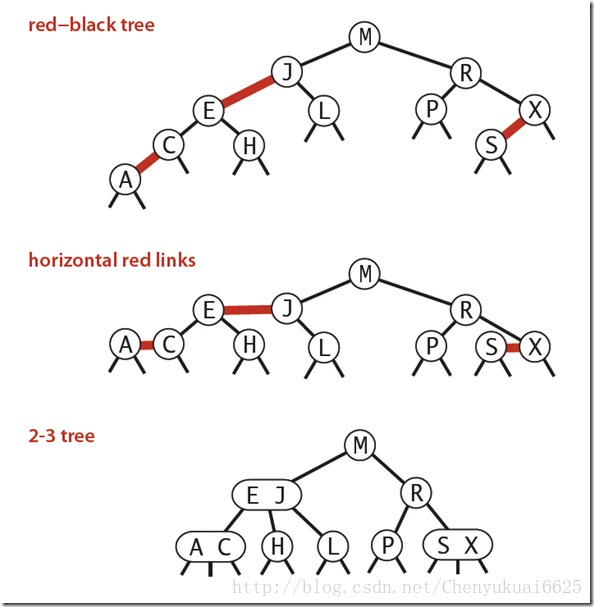
**平衡查找树之2-3查找树**

和二叉树不一样，2-3树中每个节点保存1个或者两个key值。对于普通的2节点(2-node)，它保存1个key和左右两个孩子（或没有孩子）。对应3节点(3-node)，保存两个Key和三个孩子（或没有孩子）



**平衡查找树之红黑树**

红黑树的思想就是对2-3查找树进行编码，尤其是对2-3查找树中的3-nodes节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个2-nodes节点来表示一个3-nodes节点。黑色链接用来链接普通的2-3节点。特别的，使用红色链接的两个2-nodes来表示一个3-nodes节点，并且向左倾斜，即一个2-node是另一个2-node的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。



**B树和B+树**

**B 树**可以看作是对2-3查找树的一种扩展，即他允许每个节点有M-1个子节点。

* 根节点至少有两个子节点
* 每个节点有M-1个key，并且以升序排列
* 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
* 其它节点至少有M/2个子节点

下图是一个M=4 阶的B树:



**B+**树是对B树的一种变形树，它与B树的差异在于：

* 有k个子结点的结点必然有k个关键码；
* 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
* 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

如下图，是一个B+树:



## ？红黑树是什么？红黑树的实现原理和应用场景,时间复杂度?

它的查找、插入和删除操作的时间复杂度是O(lgn)

## 排序算法

**排序算法大体可分为两种：**

一种是比较排序，时间复杂度O(nlogn) ~ O(n^2)，主要有：冒泡排序，选择排序，插入排序，归并排序，堆排序，快速排序等

另一种是非比较排序，时间复杂度可以达到O(n)，主要有：计数排序，基数排序，桶排序等。

排序算法稳定性的简单形式化定义为：**如果Ai = Aj，排序前Ai在Aj之前，排序后Ai还在Aj之前，则称这种排序算法是稳定的。**通俗地讲就是保证排序前后两个相等的数的相对顺序不变。



## 加密算法

加密算法可以归结为三大类：哈希算法、对称加密算法、非对称加密算法。

哈希算法：如MD5算法，SHA系列算法

对称加密算法：DES算法，3DES算法，AES算法

非对称加密算法：RSA算法

**对称加密**

采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。而因为加密和解密都使用同一个密钥，如何把密钥安全地传递到解密者手上就成了必须要解决的问题。

**非对称加密**

非对称加密算法需要两个密钥来进行加密和解密，分别是公钥和私钥。需要注意的一点，这个公钥和私钥必须是一对的，如果用公钥对数据进行加密，那么只有使用对应的私钥才能解密，反之亦然。由于加密和解密使用的是两个不同的密钥，因此，这种算法叫做非对称加密算法。

**HTTPS一般使用的加密与HASH算法**

非对称加密算法：RSA,DSA/DSS,DH

对称加密算法：AES,RC4,3DES

HSAH算法：MD5,SHA1,SHA256

AES，全称为“Advanced Encryption Standard”，中文名“高级加密标准”，在密码学中又称 Rijndael 加密法，是美国联邦政府采用的一种区块加密标准。AES 加密算法作为新一代的数据加密标准汇聚了强安全性、高性能、高效率、易用和灵活等优点。AES 设计有三个密钥长度：128，192，256 位。相对而言，AES 的 128 密钥比 DES 的 56 密钥强了 1021 倍。

**数字签名证书**非对称加密已经灰常安全了，但是还有一个破绽：  
服务器A公布了自己的公钥，我的电脑是用服务器A的公钥加密数据后再发给服务器A的；这时候服务器B侵入了我的电脑，把我用来加密的公钥换成了它的公钥，于是我发出去的数据就会被服务器B的私钥破解了。肿么防止公钥被篡改呢？  
对，我们想到了前面的消息摘要，服务器A把公钥丢给我的时候，同时去CA申请一份数字证书，其实主要就是公钥的消息摘要，有了这份证书，当我再用公钥加密的时候，我就可以先验证一下当前的公钥是否确定是服务器A发送给我的

# 锁

**锁的分类**

锁有哪些分类：悲观锁、乐观锁、独占锁、共享锁、公平锁、非公平锁、分布式锁、自旋锁

**jvm中有以下三种锁(由上到下越来越“重量级”)：偏向锁,轻量级锁,重量级锁**

偏向锁：适用于只有一个线程访问同步块场景。加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。

轻量级锁： 可以有多个线程交替进入临界区，在竞争不激烈的时候，稍微自旋等待一下就能获得锁。这也是CAS的实现方式。竞争的线程不会阻塞，提高了程序的响应速度。但是会消耗CPU

重量级锁：比如Synchronized它会利用java对象的监视器锁（monitor）实现，同一临界区内只允许一个线程操作。线程竞争不使用自旋，不会消耗CPU。

**java中锁的实现类有哪些？**

ReentrantLock,ReentrantReadWriteLock,StampedLock,AbstractQueuedSynchronizer

## volatile的使用场景

volatile一般用于状态标志，或者开销较低的“读－写锁”策略

**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。

## Synchronized实现原理

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

Synchronized是JVM实现的一种锁，其中锁的获取和释放分别是monitorenter和monitorexit指令，该锁在实现上分为了偏向锁、轻量级锁和重量级锁，其中偏向锁在1.6是默认开启的，轻量级锁在多线程竞争的情况下会膨胀成重量级锁，有关锁的数据都保存在对象头中

同步代码块是使用monitorenter和monitorexit指令实现的,同步方法（在这看不出来需要看JVM底层实现）依靠的是方法修饰符上的ACCSYNCHRONIZED实现。

## atomic的实现原理

以AtomicInteger为例，他的值是存在一个使用volatile修饰的int里面，由此保证该值的可见性，再通过自旋 + CAS（乐观锁），比较更新value值，由此保证操作的原子性。

Java 8做出的改进和努力

在Java 8中引入了4个新的计数器类型，LongAdder、LongAccumulator、DoubleAdder、DoubleAccumulator。他们都是继承于Striped64。

在LongAdder 与AtomicLong有什么区别？

Atomic\*遇到的问题是，只能运用于低并发场景。因此LongAddr在这基础上引入了分段锁的概念。大概就是当竞争不激烈的时候，所有线程都是通过CAS对同一个变量（Base）进行修改，当竞争激烈的时候，会将根据当前线程哈希到对于Cell上进行修改（多段锁）。

## ReentrantLock实现原理

ReentrantLock是基于AQS实现的，AQS内部维系一个状态值，双向链表，当线程需要获取锁的时候，本质上是通过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。

## CAS原理及缺点

CAS是通过unsafe类的compareAndSwap方法实现的， 它将内存位置的内容与给定值进行比较，只有在相同的情况下，将该内存位置的内容修改为新的给定值。

compareAndSwap方法的第一个参数是要修改的对象，第二个参数是对象中要修改变量的偏移量，第三个参数是修改之前的值，第四个参数是预想修改后的值。通过将偏移量与修改之前的值做比较，则将将偏移量修改为预想修改的值，并返回true.

当然CAS一定要volatile变量配合

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

compareAndSet这个方法主要调用unsafe.compareAndSwapInt这个方法，这个方法有四个参数，其中第一个参数为需要改变的对象，第二个为偏移量(即之前求出来的valueOffset的值)，第三个参数为期待的值，第四个为更新后的值。整个方法的作用即为若调用该方法时，value的值与expect这个值相等，那么则将value修改为update这个值，并返回一个true，如果调用该方法时，value的值与expect这个值不相等，那么不做任何操作，并范围一个false。

**CAS的缺点：**

1.CPU开销较大：在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

3.ABA问题：这是CAS机制最大的问题所在。

如线程1从内存X中取出A，这时候另一个线程2也从内存X中取出A，并且线程2进行了一些操作将内存X中的值变成了B，然后线程2又将内存X中的数据变成A，这时候线程1进行CAS操作发现内存X中仍然是A，然后线程1操作成功。虽然线程1的CAS操作成功，但是整个过程就是有问题的。比如链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。

所以JAVA中提供了AtomicStampedReference/AtomicMarkableReference来处理会发生ABA问题的场景，主要是在对象中额外再增加一个标记来标识对象是否有过变更。

## CAS与Synchronized的关系

Synchronized属于悲观锁，Synchronized关键字会让没有得到锁资源的线程进入BLOCKED状态，而后在争夺到锁资源后恢复为RUNNABLE状态，这个过程中涉及到操作系统用户模式和内核模式的转换，代价比较高。

CAS属于乐观锁，它会让当前线程处于忙循环，一直去争取获取锁。

什么时候使用CAS锁

atomic类，以及lock类的底层实现都会设计到CAS锁。在Java1.6以上的版本,synchronized转变为重量级锁之前，也会采用CAS机制。

## Lock与synchronized的区别

          1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的

          2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的     条件下提供一种退出的机制。

          3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以    及设置等待时限等方式退出条件队列。

         4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

AbstractQueuedSynchronizer通过构造一个基于阻塞的CLH队列容纳所有的阻塞线程，而对该队列的操作均通过Lock-Free（CAS）操作，但对已经获得锁的线程而言，ReentrantLock实现了偏向锁的功能。

synchronized的底层也是一个基于CAS操作的等待队列，但JVM实现的更精细，把等待队列分为ContentionList和EntryList，目的是为了降低线程的出列速度；当然也实现了偏向锁，从数据结构来说二者设计没有本质区别。但synchronized还实现了自旋锁，并针对不同的系统和硬件体系进行了优化，而Lock则完全依靠系统阻塞挂起等待线程。

当然Lock比synchronized更适合在应用层扩展，可以继承AbstractQueuedSynchronizer定义各种实现，比如实现读写锁（ReadWriteLock），公平或不公平锁；同时，Lock对应的Condition也比wait/notify要方便的多、灵活的多。

## 死锁

**死锁产生的四个条件：**

1）互斥。共享资源同时只能被一个线程访问。

2）占有且等待。线程T1在取得共享资源A的时候，请求等待资源B的时候并不释放资源A。

3）不可抢占。其他线程不能强行抢占线程的资源。

4）循环等待条件。线程T1在持有资源A1，同时在请求等待获取资源B，线程T2在持有资源B,然后在请求等待线程T1的持有资源，形成了交叉闭环申请。

处理死锁的方法可有以下4种：

1）死锁预防。由于互斥这个条件无法避免。操作系统本身所具有的特点。可对死锁产生的其他三个条件进行破坏。首先条件2，占有且等待，可以一次性申请所有的资源，可以破坏掉占有且等待。条件三不可抢占。当线程去请求其他资源时，如果获取不到锁，可以主动释放自己的锁，这样不可抢占的条件也被破坏掉了。条件四循环等待条件。可以对申请的资源进行编号，按序访问，这样线性的去申请资源，则不会造成交叉循环。

2）死锁避免。就不用去破坏形成死锁的其他条件。在资源动态分配的过程中。用某种方法判断防止进入不安全状态。从而避免发生死锁。可以使用银行家算法。死锁避免的算法会导致系统开销的增加。

3）死锁检测。死锁预防和死锁检测都是死锁发生之前的预防策略。死锁检测是通过系统设置的检测机构及时的判断当前系统是否处于死锁状态，并精确的确定当前死锁相关的进程和资源，执行死锁解除策略

4）死锁解除。这是与死锁检测结合使用的。它使用的方式是剥夺。就是讲进程所占有的资源强行收回，分配给其他进程。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**死锁排查**

1 、 使用top命令查看cpu占用资源较高的PID

2、 通过jps 找到当前用户下的java程序PID：执行 jps -l 能够打印出所有的应用的PID，找到有一个PID和这个cpu使用100%一样的ID！！就知道是哪一个服务了。

3、 使用 pidstat -p 1 3 -u -t

4 、 找到cpu占用较高的线程TID

5、 将TID转换为十六进制的表示方式

6、 通过jstack -l 输出当前进程的线程信息

7、 查找 TID对应的线程(输出的线程id为十六进制)，找到对应的代码

# 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。

**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。

## ？ThreadLocal

ThreadLocal什么时候会出现OOM的情况？为什么？

线程池的一个线程使用完ThreadLocal对象之后，再也不用，由于线程池中的线程不会退出，线程池中的线程的存在，同时ThreadLocal变量也会存在，占用内存！造成OOM溢出！

原文：<https://blog.csdn.net/GoGleTech/article/details/78318712>

ThreadLocal终极篇 https://mp.weixin.qq.com/s/\_gQ0uJTTk4Ws8fDojK61og

# 设计者模式

GOF23种设计模式精解

创建型

1. Factory Method（工厂方法）

2. Abstract Factory（抽象工厂）

3. Builder（建造者）

4. Prototype（原型）

5. Singleton（单例）

结构型

6. Adapter Class/Object（适配器）

7. Bridge（桥接）

8. Composite（组合）

9. Decorator（装饰）

10. Facade（外观）

11. Flyweight（享元）

12. Proxy（代理）

行为型

13. Interpreter（解释器）

14. Template Method（模板方法）

15. Chain of Responsibility（责任链）

16. Command（命令）

17. Iterator（迭代器）

18. Mediator（中介者）

19. Memento（备忘录）

20. Observer（观察者）

21. State（状态）

22. Strategy（策略）

23. Visitor（访问者）

## 单例模式饿汉模式的坑

即便加了synchronized与双层空判断也还存在隐患，这里涉及到了JVM编译器的指令重排。

比如java中简单的一句 instance = new Singleton，会被编译器编译成如下JVM指令：

memory =allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance =memory; //3：设置instance指向刚分配的内存地址

但是这些指令顺序并非一成不变，有可能会经过JVM和CPU的优化，指令重排成下面的顺序：

memory =allocate(); //1：分配对象的内存空间

instance =memory; //3：设置instance指向刚分配的内存地址

ctorInstance(memory); //2：初始化对象

当线程A执行完1,3,时，instance对象还未完成初始化，但已经不再指向null。此时如果线程B抢占到CPU资源，执行 if（instance == null）的结果会是false，从而返回一个没有初始化完成的instance对象。

因此，要给private static Singleton install添加volatile修饰符，volatile关键字不但可以防止指令重排，也可以保证线程访问的变量值是主内存中的最新值。

但是这样其它代码依然可以通过反射获取。

**结构型模式**



## 适配器模

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

**类的适配器模式**，核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里

**对象的适配器模式**：基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。

**接口的适配器**是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

## 外观模式（Facade）

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口。（Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖）

## 桥接模式（Bridge）

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样

## 组合模式（Composite）

组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便，

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

## 享元模式（Flyweight）

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。



## 策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。策略模式多用在算法决策系统中。

比如：A是一个接口，提供calculate(String exp)方法，B是一个抽象类，提供split(String exp,String opt)方法。C类继承B实现A，在C的calculate(String exp)方法中调用B的split方法，并做其它操作。

## 模板方法模式（Template Method）

解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用，

## 观察者模式（Observer）

当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

## 迭代子模式（Iterator）

顾名思义，迭代器模式就是顺序访问聚集中的对象，一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。

## 责任链模式（Chain of Responsibility）

接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。

## 命令模式（Command）

命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

## 备忘录模式（Memento）

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。

## 状态模式（State）

核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。

## 访问者模式（Visitor）

访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

## 中介者模式（Mediator）

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。

## 解释器模式（Interpreter）

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！