## Java并发编程

### 并发编程相关概念

**线程安全类**：当多个线程访问某个类时,这个类始终保持正确行为那么这个类是线程安全的

无状态对象:既不包含任何域,也不包含对任何其他域的引用。使用的局部变量存于栈中的局部变量里,只能由正在执行的线程访问。

**原子性**：一个或多个操作,要么全部执行并且执行的过程不会被任何因素打断,要么都不执行

**可见性**：当多个线程访问同一个变量时, 一个线程修改了这个变量的值,其他线程能够立即看到修改的值

**有序性**：即程序执行的顺序按照代码的先后顺序执行；在Java内存模型中,允许编译器和处理器对指令进行重排序,但是重排序过程不会影响到单线程程序的执行,却会影响到多线程并发执行的正确性。

**内存模型的相关概念：**计算机在执行程序时,每条指令都是在CPU中执行的,而执行指令过程中,势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的,这时就存在一个问题,由于CPU执行速度很快,而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多,因此如果任何时候对数据的操作都要通过和内存的交互来进行,会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。也就是,当程序在运行过程中,会将运算需要的数据从主存复制一份到CPU的高速缓存当中,那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据,当运算结束之后,再将高速缓存中的数据刷新到主存当中。如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现）,那么就可能存在缓存不一致的问题

**重入**：如果某个线程试图获得一个已经由它自己持有的锁,那么这个请求就会成功。实现方法：为每个锁关联一个获取计数值和一个所有者线程；当计数值为0时这个锁被认为是没有被任何线程持有。当线程请求一个未被持有的锁时,jvm将记下锁的持有者并将获取计数值置位1.如果同一个线程再次获取这个锁,计数值将+1,当线程退出同步代码块时计数器会相应递减,计数值为0时这个锁会被释放。

### Synchronized关键字

Synchronized是java中的关键字,是一种同步锁,修饰对象有：

1. 修饰一个代码块,作为对象是调用这个代码块的对象 sychronized(this){} 也可以指定对某个对象加锁同步代码块包括两部分：一个作为锁的对象引用,一个由该锁保护的代码块。以synchronized修饰的方法的锁就是方法调用所在的对象,静态的synchronized方法以Class对象作为锁。线程进入同步代码块之前自动获得锁,并在退出同步代码块时自动释放锁
2. 修饰一个方法,作用的对象是调用这个方法的对象；synchronized关键字不能继承
3. 修饰一个静态方法,作用对象是这个类的所有对象
4. 修饰一个类,其作用的范围是synchronized后面括号括起来的部分,作用主的对象是这个类的所有对象 sychronized(ClassName.class){} 不在类定义时修饰

无论synchronized关键字加在方法上还是对象上,如果它作用的对象是非静态的,则它取得的**锁是对象**；如果synchronized作用的对象是一个静态方法或一个类,则它取得的锁是对类,该类所有的对象同一把锁。

每个对象只有一个锁（lock）与之相关联,谁拿到这个锁谁就可以运行它所控制的那段代码

### Volatile关键字

Java提供volatile关键字来保证可见性。当一个共享变量被volatile修饰时,他会保证修改的值会立即被更新到主存,当其他线程需要读取时会去内存中读取新值。普通的共享变量不能保证可见性,因为普通共享变量被修改后什么时候被写入主存是不确定的,当其他线程去读取时内存中可能还是旧值,无法保证可见性。

一个共享变量(类的成员变量、类的静态成员变量)被volatile修饰之后就具备了两层语义：

保证不同线程对这个变量操作时的可见性;禁止进行指令重排序

访问volatile变量时不会执行加锁操作,也不会使执行线程阻塞,因此volatile是一种比sychronized关键字更轻量级的同步机制。

当且仅当满足以下所有条件时,才应该使用volatile变量：

对变量的写入操作不依赖变量当前的值,或者只有单个线程更新变量的值；

该变量不会与其他状态变量一起纳入不变性条件中。

Volatile不保证对变量操作的原子性。

### ThreadLocal类

每个Thread维护一个ThreadLocalMap映射表,这个映射表的key是ThreadLocal实例本身,value是真正需要存储的object. 也就是说ThreadLocal本身并不存储值,它只是作为一个key来让线程从ThreadLocalMap获取value。ThreadLocalMap是使用ThreadLocal的弱引用作为Key的,弱引用的对象在GC时会被回收.

这个类能使线程中的某个值与保存值的对象关联起来,ThreadLocal类提供了get与set等访问接口或方法,这些方法为每个使用该变量的线程都存有一份独立的副本,因为get总返回由当前执行线程在调用set时设置的最新值。

ThreadLocal对象通常用于放置对可变的单实例变量或全局变量进行共享。比如将JDBC链接保存到ThreadLocal对象中,每个线程都会拥有属于自己的连接。

ThreadLocal引发的内存泄漏问题

ThreadLocalMap使用ThreadLocal的弱引用作为key,如果一个ThreadLocal没有外部强引用来引用它,那么系统GC的时候,这个ThreadLocal对象势必会被回收(GC时会回收只有弱引用的对象),这样ThreadLocalMap中就会出现key为null的Entry,就没有办法访问这些key为null的Entry的value,如果当前线程再迟迟不结束的话,这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收,造成内存泄漏

由于ThreadLocalMap的生命周期跟Thread一样长，如果都没有手动删除对应key，都会导致内存泄漏。

**Synchronized关键字与lock**

Lock是一个接口,是JDK层面的实现；而synchronized是Java中的关键字,是Java的内置特性,是JVM层面的实现；

synchronized 在发生异常时,会自动释放线程占有的锁,因此不会导致死锁现象发生；而Lock在发生异常时,如果没有主动通过unLock()去释放锁,则很可能造成死锁现象,因此使

Lock 可以让等待锁的线程响应中断:lock.interrupt(),而使用synchronized时,等待的线程会一直等待下去,

通过Lock可以知道有没有成功获取锁:lock.tryLock(),而synchronized却无法办到；

Lock可以提高多个线程进行读操作的效率

## Java线程池

什么是线程池：java.util.concurrent.Executors提供了一个 java.util.concurrent.

Executor接口的实现用于创建线程池。多线程技术主要解决处理器单元内多个线程执行的问题,它可以显著减少处理器单元的闲置时间,增加处理器单元的吞吐能力。假设一个服务器完成一项任务所需时间为：T1 创建线程时间,T2 在线程中执行任务的时间,T3 销毁线程时间。

如果：T1 + T3 远大于 T2,则可以采用线程池,以提高服务器性能。

一个线程池包括以下四个基本组成部分：

1、线程池管理器（ThreadPool）：用于创建并管理线程池,包括创建线程池,销毁线程池,添加新任务；

2、工作线程（PoolWorker）：线程池中线程,在没有任务时处于等待状态,可以循环的执行任务；

3、任务接口（Task）：每个任务必须实现的接口,以供工作线程调度任务的执行,它主要规定了任务的入口,任务执行完后的收尾工作,任务的执行状态等；

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

### 任务提交给线程池之后

如果当前线程池中的线程数目小于corePoolSize,则每来一个任务,就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize,则每来一个任务,会尝试将其添加到任务缓存队列当中,若添加成功,则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满）,则会尝试创建新的线程去执行这个任务；

如果当前线程池中的线程数目达到maximumPoolSize,则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时,如果某线程空闲时间超过keepAliveTime,线程将被终止,直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间,那么核心池中的线程空闲时间超过keepAliveTime,线程也会被终止。

### 四种常见线程池

Java通过Executors工厂类可以创建四种线程池,分别为：

newCachedThreadPool创建一个可缓存线程池。CachedThreadPool是一种可以无限扩容的线程池,比较适合执行时间片比较小的任务； keepAliveTime为60,线程空闲时间超过60s就会被杀死；阻塞队列采用SynchronousQueue,这种阻塞队列没有存储空间,意味着只要有任务到来,就必须得有一个工作线程来处理,如果当前没有空闲线程,就再创建一个新的线程。

newFixedThreadPool 创建一个定长线程池,可控制线程最大并发数,超出的线程会在队列中等待。阻塞队列采用LinkedBlockingQueue,它是一种无界队列,因此永远不可能拒绝执行任务；由于采用无界队列,实际线程数将永远维持在nThreads,因此maximumPoolSize和keepAliveTime将无效。

newScheduledThreadPool 创建一个定长线程池,支持定时及周期性任务执行。

newSingleThreadExecutor 创建一个单线程化的线程池,它只会用唯一的工作线程来执行任务,保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

### 任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列,即workQueue,它用来存放等待执行的任务。

workQueue的类型为BlockingQueue<Runnable>,通常可以取下面三种类型：

（1）ArrayBlockingQueue：基于数组的先进先出队列,此队列创建时必须指定大小；

（2）LinkedBlockingQueue：基于链表的先进先出队列,如果创建时没有指定此队列大小,则默认为Integer.MAX\_VALUE；

（3）synchronousQueue：这个队列比较特殊,它不会保存提交的任务,而是将直接新建一个线程来执行新来的任务。

### 四种拒绝策略

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

### 根据任务的类型来配置线程池大小

如果是CPU密集型任务,就需要尽量压榨CPU,参考值可以设为 CPU数+1

如果是IO密集型任务,参考值可以设置为2\*CPU数

只是一个参考值,具体的设置还需要根据实际情况进行调整

## java集合类

Java集合类有两大分支:Collection和Map

### Collection

├List 列表  
 │├LinkedList   
 │├ArrayList 通过ensureCapacity(int n)方法可提高ArrayList的初始化速度  
 │└Vector  
 │　└Stack  
└Set 集合 不允许有重复元素

├LinkedHashSet外部按成员的插入顺序遍历成员

├treeset外部有序地遍历成员;附加实现了SortedSet,支持子集等要求顺序的操作

├hashset

**LinkedList和ArrayList**

LinkedList底层采用双向循环列表实现,对列表中任何位置的成员的增加和删除支持较好,但对基于索引的成员访问支持性能较差;可以使用LinkedList来实现队列和栈stack以及双端队列dequeue。

ArrayList是实现了基于动态数组的数据结构,对于随机访问(get()方法) 它的速度要比LinkedList快。LinkedList中的get方法是按照顺序从列表的一端开始检查,直到另外一端。

1．对ArrayList和LinkedList而言,在列表末尾增加一个元素所花的开销都是固定的。对ArrayList而言,主要是在内部数组中增加一项,指向所添加的元素,偶尔可能会导致对数组重新进行分配；而对LinkedList而言,这个开销是统一的,分配一个内部Entry对象。

2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾预留一定的容量空间,而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间.(保存下一个元素的位置)

**HashSet和TreeSet**

HashSet是由一个hash表来实现的,因此它的元素是无序的。add(),remove(),contains()方法的时间复杂度是O(1)。 HashSet,为快速查找而设计的Set。存入HashSet的对象必须实现hashCode()和equals()。

另一方面,TreeSet是由一个树形的结构来实现的,它里面的元素是有序的。因此,add(),remove(),contains()方法的时间复杂度是O(logn)。添加到TreeSet中的元素必须是可排序的。

**线程安全的列表**

CopyOnWriteArrayList和Collections.synchronizedList是实现线程安全的列表的两种方式。两种实现方式分别针对不同情况有不同的性能表现,其中CopyOnWriteArrayList的写操作性能较差(复制数组),而多线程的读操作性能较好；而Collections.synchronizedList的写操作性能比CopyOnWriteArrayList在多线程操作的情况下要好很多,而读操作因为是采用了synchronized关键字的方式,其读操作性能并不如CopyOnWriteArrayList。

### Map

├Hashtable key和value都不允许出现null值,Hashtable 中的方法是同步的

├TreeMap支持对键有序地遍历

├HashMap中null可以作为键,这样的键只有一个；

└LinkedHashMap 保留了键的插入顺序

HashMap的实现:本质是通过拉链法实现的哈希表,键值对都存放在一个Entry(单向链表)数组中。当向hashmap中put键值对时如果多个键的hash发生碰撞则会插入到数组同一位置的链表末端;hashMap不是线程安全的。

ConcurrentHashMap: 本质上,ConcurrentHashMap就是一个Segment数组,而一个Segment实例则是一个小的哈希表。由于Segment类继承于ReentrantLock类,从而使得Segment对象能充当锁的角色,这样,每个 Segment对象就可以守护整个ConcurrentHashMap的若干个桶,其中每个桶是由若干个HashEntry 对象链接起来的链表。通过使用段(Segment)将ConcurrentHashMap划分为不同的部分,ConcurrentHashMap就可以使用不同的锁来控制对哈希表的不同部分的修改,从而允许多个修改操作并发进行, 这正是ConcurrentHashMap锁分段技术的核心内涵。进一步地,如果把整个ConcurrentHashMap看作是一个父哈希表的话,那么每个Segment就可以看作是一个子哈希表。

ConcurrentHashMap对Segment的put操作是加锁完成的。Segment是ReentrantLock的子类,因此Segment本身就是一种可重入的Lock,所以我们可以直接调用其继承而来的lock()方法和unlock()方法对代码进行上锁/解锁。需要注意的是,这里的加锁操作是针对某个具体的Segment,锁定的也是该Segment而不是整个ConcurrentHashMap。因为插入键/值对操作只是在这个Segment包含的某个桶中完成,不需要锁定整个ConcurrentHashMap。因此,其他写线程对另外15个Segment的加锁并不会因为当前线程对这个Segment的加锁而阻塞。故而 相比较于 HashTable 和由同步包装器包装的HashMap每次只能有一个线程执行读或写操作,ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下,ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16）,及任意数量线程的读操作。

HashEntry用来封装具体的键值对,是个典型的四元组。与HashMap中的Entry类似,HashEntry也包括同样的四个域,分别是key、hash、value和next。不同的是,在HashEntry类中,key,hash和next域都被声明为final的,value域被volatile所修饰,因此HashEntry对象几乎是不可变的,这是ConcurrentHashmap读操作并不需要加锁的一个重要原因. next域被声明为final本身就意味着我们不能从hash链的中间或尾部添加或删除节点,因为这需要修改next引用值,因此所有的节点的修改只能从头部开始。对于put操作,可以一律添加到Hash链的头部。但是对于remove操作,可能需要从中间删除一个节点,这就需要将要删除节点的前面所有节点整个复制(重新new)一遍,最后一个节点指向要删除结点的下一个结点(这在谈到ConcurrentHashMap的删除操作时还会详述)。特别地,由于value域被volatile修饰,所以其可以确保被读线程读到最新的值,这是ConcurrentHashmap读操作并不需要加锁的另一个重要原因.

HashMap输出是无序的,这个无序不是每次遍历的结果顺序不一样,而是与插入顺序不一样。

对hashmap的按值排序的实现：

public static void sortHashMap(Map<Integer , String> hashmap){

//第一步,用HashMap构造一个LinkedList

Set<Entry<Integer , String>> sets = hashmap.entrySet();

LinkedList<Entry<Integer , String>> linkedList = new LinkedList<Entry<Integer , String>>(sets);

//用Collections的sort方法排序

Collections.sort(linkedList , new Comparator<Entry<Integer , String>>(){

public int compare(Entry<Integer , String> o1, Entry<Integer , String> o2) {

// TODO Auto-generated method stub

/\*String object1 = (String) o1.getValue();

String object2 = (String) o2.getValue();

return object1.compareTo(object2);\*/

return o1.getValue().compareTo(o2.getValue());

}

});

//第三步,将排序后的list赋值给LinkedHashMap

Map<Integer , String> map = new LinkedHashMap();

for(Entry<Integer , String> entry : linkedList){

map.put(entry.getKey(), entry.getValue());

}

}

Hashmap按键排序的实现：TreeMap默认是按键排序的,默认升序,所以先用comparator对象构造一个TreeMap,再将要排序的HashMap添加到TreeMap中。

//第一步：先创建一个TreeMap实例,构造函数传入一个Comparator对象。

TreeMap<Integer , String> treemap = new TreeMap<Integer , String>(new Comparator<Integer>(){

@Override

public int compare(Integer o1,Integer o2) {

// TODO Auto-generated method stub

return Integer.compare(o1, o2);

}

});

//第二步：将要排序的HashMap添加到我们构造的TreeMap中。

treemap.putAll(hashmap);

### 支持并发的容器

Map:hashtable,该类的方法都是通过synchronized来进行方法层次同步的,以达到线程安全的作用。性能更好的Map为ConcurrenthashMap,内部使用lock而不是synchronize,相比Hashtable效率上提高很大同时兼顾线程安全。ConcurrentHashMap允许多个修改并发操作,因为ConcurrentHaspMap采用了内部使用段机制,将ConcurrentHashMap分成了很多段,只要不是在同一个小段上写就可以并发。

Collection：CopyOnWriteArraySet、CopyOnWriteArrayList/Vector.使用CopyOnWrite机制,即当往容器里添加元素的时候,先对这个容器进行一次复制,对副本进行写操作。写操作结束后,将原容器的引用指向新副本容器,就完成了写的刷新。缺点是如果容器较大会占用较多内存,还有在写操作时只能在最终结束后使数据同步,不能实时同步。适用于读操作多写操作少的场景。

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。Skip list让已排序的数据分布在多层链表中,以0-1随机数决定一个数据的向上攀升与否,通过空间来换取时间的一个算法。ConcurrentSkipListMap提供了一种线程安全的并发访问的排序映射表。内部是SkipList（跳表）结构实现,在理论上能够在O(log(n))时间内完成查找、插入、删除操作。

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）

LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue

对应的非并发容器：BlockingQueue

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

实现类：

LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列

ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列

PriorityBlockingQueue：按优先级排序的队列

haspMap的遍历:

for(Entry entry:map.entrySet()){

K key = entry.getKey();

V value = entry.getValue();

}

entrySet()方法返回了set<Map.entry<K,V>>,即Map内部接口Map.entry的集合,Map.entry提供了getKey()和getValue()方法

## java克隆

浅克隆 对象中其他对象的成员变量会克隆引用；

深克隆 会对对象成员变量也克隆(实现深克隆的方法：序列化与反序列化(需要实现Serializable接口))

public Object deepClone()

{

//将对象写到流里

ByteArrayOutoutStream bo=new ByteArrayOutputStream();

ObjectOutputStream oo=new ObjectOutputStream(bo);

oo.writeObject(this);

//从流里读出来

ByteArrayInputStream bi=new ByteArrayInputStream(bo.toByteArray());

ObjectInputStream oi=new ObjectInputStream(bi);

return(oi.readObject());

}

这样做的前提是对象以及对象内部所有引用到的对象都是可串行化的

## String、StringBuffer与StringBuilder

String为字符串常量,StringBuffer与StringBuilder是字符串变量。

当声明一个String变量时:String s = “a” s是指向堆内存中的引用,存储了“a”在堆中的地址；s本身存储在栈内存中。当对s重新赋值时:s=“b” 此时s将指向“b”在堆中的地址。此时堆中将同时存在“a”和“b”对象,不过“a”如果不被其他变量引用则会被虚拟机垃圾回收掉。JVM为了提高性能和减少内存开销,内部维护了一个字符串常量池,每当创建字符串常量时,JVM首先检查字符串常量池,如果常量池中已经存在,则返回池中的字符串对象引用,否则创建该字符串对象并放入池中。

1.String中使用 + 字符串连接符进行字符串连接时,连接操作最开始时如果都是字符串常量,编译后将尽可能多的直接将字符串常量连接起来,形成新的字符串常量参与后续连接；

2.接下来的字符串连接是从左向右依次进行,对于不同的字符串,首先以最左边的字符串为参数创建StringBuilder对象,然后依次对右边进行append操作,最后将StringBuilder对象通过toString()方法转换成String对象。

result += str;会编译成类似代码result = new StringBuilder(result)

.append(str).toString();当在循环中对string进行连接操作时每次连接都会新建一个stringbuilder对象,此时效率会比append差

也就是说String c = "xx" + "yy " + a + "zz" + "mm" + b; 实质上的实现过程是： String c = new StringBuilder("xxyy").append(a).append("zz").

append("mm").append(b).toString();

由于得出结论：当使用+进行多个字符串连接时,实际上是产生了一个StringBuilder对象和一个String对象。

StringBuffer是**线程安全**的,StringBuilder是非线程安全的。执行速度方面StringBuilder快于StringBuffer。(都实现了append()方法)

如果操作少量的数据使用String,单线程操作字符缓冲区下大量数据用StringBuilder,多线程字符串缓冲区下操作大量数据用StringBuffer.

## Java存储区

Java中把存储区分为6类,分别为寄存器(register)、栈(stack)、堆(heap)、静态存储区(static storage)、常量存储区(constant storage jdk1.7之后字符串常量池存放在堆空间中)以及非随机存取存储区（Non-RAM）。

**栈内存**储了**局部变量**(定义在方法中的都是局部变量)、保存基础数据类型的值和**对象以及基础数据的引用**；方法先进栈,然后定义变量,变量有自己的作用域一旦离开作用域变量就会被释放。栈内存更新速度快于堆内存。

栈有一个很重要的特殊性,就是存在栈中的数据可以共享。假设我们同时定义 int a = 3;int b = 3; 编译器先处理int a = 3;首先它会在栈中创建一个变量为a的引用,然后查找有没有字面值为3的地址,没找到,就开辟一个存放3这个字面值的地址,然后将a指向3的地址。接着处理int b = 3；在创建完b的引用变量后,由于在栈中已经有3这个字面值,便将b直接指向3的地址。这样,就出现了a与b同时均指向3的情况。

　　 特别注意的是,这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象,如果一个对象引用变量修改了这个对象的内部状态,那么另一个对象引用变量也即刻反映出这个变化。相反,通过字面值的引用来修改其值,不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例,我们定义完a与 b的值后,再令a=4；那么,b不会等于4,还是等于3。在编译器内部,遇到a=4；时,它就会重新搜索栈中是否有4的字面值,如果没有,重新开辟地址存放4的值；如果已经有了,则直接将a指向这个地址。因此a值的改变不会影响到b的值

**堆内存**储了数组和**对象实体**;new关键字建立的对象都在堆中；堆中的实体不会被释放,但是java有垃圾回收机制不定时回收。

寄存器:位于CPU内部,由编译器根据需求分配无法直接控制

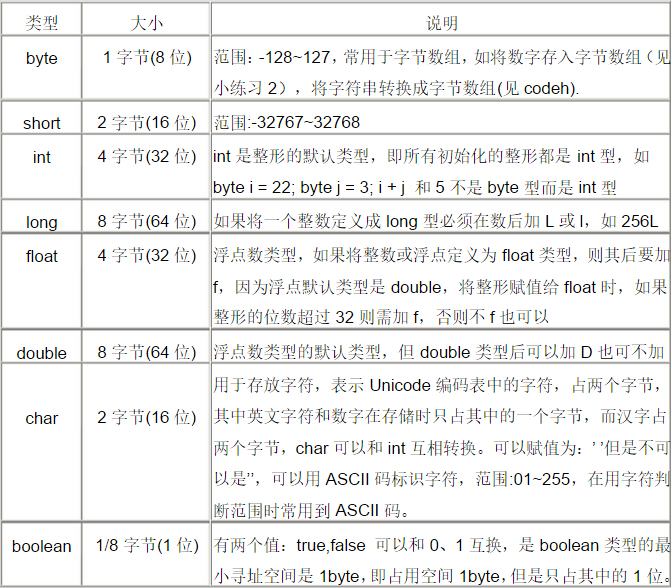
静态存储区(方法区):static关键字指定的对象/变量以及Class文件;在整个运行期都有效；

常量存储区(constant storage)：常量通常被直接放置在程序代码当中.

非RAM存储区(non-RAM storage).如果数据完全存在于程序之外,在程序未被运行时数据也能存在,而不受程序的控制.对于此类情况,有两个主要的例子来说明:一,对象流(streamed objects),这种对象通常以字节流的形式被送到其他的机器去.二,持久化对象;这种对象存放在磁盘上,即便是程序中止以后它们仍然保持原有状态.这些存储器都是将对象以某种形式保存在其他的介质中然后在需要的时候再把它恢复为常规的基于RAM的对象.

## Java变量长度

一位 bit



## JVM内存结构

**程序计数器**:JVM中一块较小的内存区域,不在RAM上,而是直接划分在CPU上;保存着当前线程执行的虚拟机字节码指令的内存地址。Java多线程的实现,其实是通过线程间的轮流切换并分配处理器执行时间的方式来实现的,在任何时刻,处理器都只会执行一个线程中的指令。在多线程场景下,为了保证线程切换回来后,还能恢复到原先状态,找到原先执行的指令,所以每个线程都会设立一个程序计数器,并且各个线程之间不会互相影响,程序计数器为"线程私有"的内存区域.

**虚拟机栈:** 虚拟机栈和线程是紧密联系的,每创建一个线程时就会对应创建一个Java栈,所以Java栈也是"线程私有"的内存区域,这个栈中又会对应包含多个栈帧,每调用一个方法时就会往栈中创建并压入一个栈帧,栈帧是用来存储方法数据和部分过程结果的数据结构,每一个方法从调用到最终返回结果的过程,就对应一个栈帧从入栈到出栈的过程.

**本地方法栈**: 本地方法栈和虚拟机栈的作用相似,不过虚拟机栈是为Java方法服务的,而本地方法栈是为Native方法服务的。

**方法区**: 方法区（Method Area）是用于存储类结构信息的地方,包括常量池、静态变量、构造函数等类型信息,类型信息是由类加载器在类加载时从类文件中提取出来的. 在Hotspot虚拟机中,这块区域对应持久代。

方法区同样存在垃圾收集,因为用户通过自定义加载器加载的一些类同样会成为垃圾,JVM会回收一个未被引用类所占的空间,以使方法区的空间达到最小.

方法区中还存在着常量池,常量池包含着一些常量和符号引用.方法区是线程共享的。

**堆:** 堆是存储java实例或者对象的地方,是GC的主要区域,同样是线程共享的内存区域。被划分为新生代、老年代(和持久代)

1、所有线程共享的内存数据区：方法区,堆。而虚拟机栈,本地方法栈和程序计数器都是线程私有的。

2、存放于栈中的东西如下：

　　2.1 每个线程包含一个栈区,栈中只保存基础数据类型的对象和自定义对象的引用(不是对象)。对象都存放在堆区中。

　　2.2 每个栈中的数据(基础数据类型和对象引用)都是私有的,其他栈不能访问。

　 2.3 方法的形式参数,方法调用完后从栈空间回收

　　 2.4 引用对象的地址,引用完后,栈空间地址立即被回收,堆空间等待GC

3、存放于堆中的东西如下：

　　　3.1 存储的全部是对象,每个对象包含一个与之对应的class信息

　　　3.2 Jvm只有一个堆区（heap）被所有线程共享,堆区中不存放基本类型和对象引用,只存放对象本身

4、存放于方法区中的东西如下：

　　4.1 存放线程所执行的字节码指令

　　　4.2 跟堆一样.被所有线程共享.方法区包含：所有的class和static变量

　　　4.3 常量池位于方法区中,见如下图示说明

**java内存模型**

java内存模型定义了程序中各个变量的访问规则,即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处是指实例字段(实例对象)、静态字段和构成数组对象的元素。不包括局部变量和方法参数,后者为线程私有不会被共享。

Java内存模型中规定了所有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使用到的变量到主内存副本拷贝,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中的变量。(工作内存只会拷贝对象的引用,而不是直接拷贝整个对象)不同线程之间无法直接访问对方工作内存中的变量,线程间变量值的传递均需要在主内存来完成。

本地内存中存储了该线程以读/写共享变量的副本。**本地内存是JMM的一个抽象概念,并不真实存在。它涵盖了缓存,写缓冲区,寄存器以及其他的硬件和编译器优化**。

## 常用GC算法

**垃圾检测算法**:

引用计数法: 给一个对象添加引用计数器,每当有个地方引用它,计数器就加1；引用失效就减1.会有循环引用的问题.

可达性分析算法：以根集对象为起始点进行搜索,如果有对象不可达的话,即是垃圾对象。

### 垃圾回收算法：

复制；标记清除；标记压缩

分代收集算法(大部分JVM采用的算法 年轻代使用复制算法,年老代使用标记整理算法)：

年轻代：是所有新对象产生的地方。年轻代被分为3个部分——Enden区和两个Survivor区（From和to）当Eden区被对象填满时,就会执行Minor GC。并把所有存活下来的对象转移到其中一个survivor区。Minor GC同样会检查存活下来的对象,并把它们转移到另一个survivor区;使用复制算法。这样在一段时间内,总会有一个空的survivor区。经过多次GC周期后,仍然存活下来的对象会被转移到年老代内存空间。通常这是在年轻代有资格提升到年老代前通过设定年龄阈值来完成的。需要注意,Survivor的两个区是对称的,没先后关系,from和to是相对的。

年老代：在年轻代中经历了N次回收后仍然没有被清除的对象,就会被放到年老代中,可以说他们都是久经沙场而不亡的一代,都是生命周期较长的对象。对于年老代和永久代,就不能再采用像年轻代中那样搬移腾挪的回收算法,因为那些对于这些回收战场上的老兵来说是小儿科。通常会在老年代内存被占满时将会触发Full GC,回收整个堆内存;使用标记整理算法。

持久代：用于存放静态文件,比如java类、方法等。持久代对垃圾回收没有显著的影响。

JDK8中已经把持久代（PermGen Space） 干掉了,取而代之的元空间（Metaspace）。Metaspace占用的是本地内存,不再占用虚拟机内存。

### Java中垃圾回收器的类型

Java提供多种类型的垃圾回收器。JVM中的垃圾收集一般都采用“分代收集”,不同的堆内存区域采用不同的收集算法,主要目的就是为了增加吞吐量或降低停顿时间。

Serial收集器：新生代收集器,使用复制算法,使用一个线程进行GC,串行,其它工作线程暂停。

ParNew收集器：新生代收集器,使用复制算法,Serial收集器的多线程版,用多个线程进行GC,并行,其它工作线程暂停。使用-XX:+UseParNewGC开关来控制使用ParNew+Serial Old收集器组合收集内存；使用-XX:ParallelGCThreads来设置执行内存回收的线程数。

Parallel Scavenge 收集器：吞吐量优先的垃圾回收器,作用在新生代,使用复制算法,关注CPU吞吐量,即运行用户代码的时间/总时间。使用-XX:+UseParallelGC开关控制使用Parallel Scavenge+Serial Old收集器组合回收垃圾。

Serial Old收集器：老年代收集器,单线程收集器,串行,使用标记整理算法,使用单线程进行GC,其它工作线程暂停。

Parallel Old收集器：吞吐量优先的垃圾回收器,作用在老年代,多线程,并行,多线程机制与Parallel Scavenge差不错,使用标记整理算法,在Parallel Old执行时,仍然需要暂停其它线程。

CMS（Concurrent Mark Sweep）收集器：老年代收集器,致力于获取最短回收停顿时间（即缩短垃圾回收的时间）,使用标记清除算法,多线程,优点是并发收集（用户线程可以和GC线程同时工作）,停顿小。使用-XX:+UseConcMarkSweepGC进行ParNew+CMS+Serial Old进行内存回收,优先使用ParNew+CMS（原因见Full GC和并发垃圾回收一节）,当用户线程内存不足时,采用备用方案Serial Old收集。

## java8 lambda表达式

定义：一个不用被绑定到一个标识符上并且可能被调用的函数。可以理解为一段带有输入参数的可执行语句块。

格式：

(Type1 param1, Type2 param2, ..., TypeN paramN) -> {

statment1;

statment2;

//.............

return statmentM;

}

简化写法：参数类型可以省略,编译器可以推断lambda表达式的参数类型；当lambda表达式参数只有一个时可以省略小括号

## Java泛型

<>中间加类名表示泛型

定义泛型类: 在类名后跟泛型参数,

eg：public class Test<T>

定义泛型方法,需将泛型参数列表置于返回值之前,eg:

public <T> void f(T x) {

System.out.println(x.getClass().getName());

}

泛型中上界和下界的定义

上界 <? extend Fruit>

下界 <? super Apple>

## java Class

class (首字母小写)关键字为声明类的关键字java.lang.Class类

java是运行在JVM之上的,编写的类代码,在经过编译器编译之后,会为每个类生成对应的.class文件,这个就是JVM可以加载执行的字节码。运行时期间,当我们需要实例化任何一个类时,JVM会首先尝试看看在内存中是否有这个类,如果有,那么会直接创建类实例；如果没有,那么就会根据类名去加载这个类,当加载一个类,或者当加载器(class loader)的defineClass()被JVM调用,便会为这个类产生一个Class对象（一个Class类的实例）,用来表达这个类,该类的所有实例都共同拥有着这个Class对象,而且是唯一的。

Java 获取到类的Class对象的方法：

1.Class.forName("类名字符串")  （注意：类名字符串必须是全称,包名+类名）

2.类名.class 3.实例对象.getClass()

多次调用获取Class对象,只会同时生成一个Class对象(java虚拟机为每个类型维持一个Class对象)。在生成Class对象的时候,首先判断内存中是否已经加载。

从JVM的角度看,我们使用关键字new创建一个类的时候,这个类可以没有被加载。但是使用Class对象的newInstance()方法的时候,就必须保证：1、这个 类已经加载；2、这个类已经连接了。而完成上面两个步骤的正是Class的静态方法forName()所完成的,这个静态方法调用了启动类加载器,即加载 java API的那个加载器。

Class.forname()方法:加载参数指定的类,并且初始化它。

## Java抽象类和接口

抽象类是指在普通类的结构里面增加了抽象方法的类,无法被实例化; 抽象方法是指没有方法体的方法,同时抽象类和方法还必须使用关键字abstract做修饰.

使用原则：

抽象方法**必须为public或者protected**（因为如果为private,则不能被子类继承,子类便无法实现该方法）,缺省情况下默认为public；

抽象类不能直接实例化,也不能用final修饰,需要依靠子类采用向上转型的方式处理；

子类（如果不是抽象类）则必须覆写抽象类之中的全部抽象方法（如果子类没有实现父类的抽象方法,则必须将子类也定义为为abstract类）；

其他方面抽象类和普通类并无区别。(成员变量方面)

**外部抽象类不允许使用static声明**,而内部的抽象类运行使用static声明。使用static声明的内部抽象类相当于一个外部抽象类,继承的时候使用“外部类.内部类”的形式表示类名称

接口中可以含有变量和方法。但是要注意，**接口中的变量会被隐式地指定为public static final变量**（并且只能是public static final变量，用private修饰会报编译错误），而**方法会被隐式地指定为public abstract方法**且只能是public abstract方法（用其他关键字，比如private、protected、static、 final等修饰会报编译错误），并且接口中所有的方法不能有具体的实现，也就是说，接口中的方法必须都是抽象方法

抽象类和接口的区别

(1).抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；

(2).抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；

(3).接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；

(4).一个类只能继承一个抽象类，而一个类却可以实现多个接口。

## Java静态类

使用场景仅为静态内部类。静态内部类使用场景一般是当外部类需要使用内部类,而内部类无需外部类资源,并且内部类可以单独创建的时候会考虑采用静态内部类的设计。静态内部类可以有非静态的方法和成员变量。

与普通内部类区别是可以单独初始化：

Inner i = new Outer.Inner();

普通内部类初始化：

Outer o = new Outer();

Inner i = o.new Inner();

内部静态类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

非静态内部类能够访问外部类的静态和非静态成员。**静态类不能访问外部类的非静态成员**。他只能访问外部类的静态成员。

静态内部类可以声明静态方法,但是**普通内部类不可以声明静态方法**。静态内内部可以定义静态成员变量,但是普通类只能定义final static的静态变量。

可以将静态类看做外部类的静态变量,使用时不要外部类实例

## Java内部类

1. 成员内部类：定义在另一个类的内部, 成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。内部类可以拥有private访问权限、protected访问权限、public访问权限及包访问权限.如果成员内部类Inner用private修饰,则只能在外部类的内部访问,如果用public修饰,则任何地方都能访问；如果用protected修饰,则只能在同一个包下或者继承外部类的情况下访问；如果是默认访问权限,则只能在同一个包下访问。**外部类只能被public和包访问(默认)两种权限修饰**。
2. 局部内部类是定义在一个方法或者一个作用域里面的类,它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。
3. 匿名内部类
4. 静态内部类 static修饰的内部类

## Java参数传递方式

Java语言参数传递都是值传递,但是当一个实例对象作为参数被传递到方法时,参数的值是该对象引用的一个副本,指向对象本身,虽然是按置传递,但是在被调用的方法中仍然可以通过引用改变方法本身的属性(如调用set方法等) 但是对象的引用本身是不会改变的(即对传递的参数做赋值运算,此时改变的指示对象引用的副本)

String对象做为参数传递时,走的依然是引用传递,但是String的值在创建之后不能被更改

每一次内容的更改都是重现创建出来的新对象。所以在调用方法中对string参数重新赋值不会改变原来的string引用指向的值

## 类加载器ClassLoader

类加载器就是根据指定全限定名称将class文件加载到JVM内存，转为Class对象。如果站在JVM的角度来看，只存在两种类加载器:

**启动类加载器**（Bootstrap ClassLoader）：由C++语言实现（针对HotSpot）,负责将存放在<JAVA\_HOME>\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中。

其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：

**扩展类加载器**（Extension ClassLoader）：负责加载<JAVA\_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。

**应用程序类加载器**（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

**双亲委派模型**工作过程是：如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。使用双亲委派模型去加载类最大的好处就是可以避免重复加载类。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。Java虚拟机先从最核心的API开始查找，防止不可信的类扮演被信任的类。

## Java类加载顺序

静态代码块,在虚拟机加载类的时候就会加载执行,而且只执行一次；非静态代码块,在创建对象的时候（即new一个对象的时候）执行,每次创建对象都会执行一次

静态代码块,非静态代码块,构造函数调用顺序：

1.父类静态变量赋值

2.执行父类静态代码块

3.子类静态变量赋值

4.子类静态内容

5.父类成员变量赋值

6.父类的非静态代码块

7.父类的构造方法

8.子类成员变量赋值

9.子类的非静态代码块

10.子类的构造方法

**Java四种引用类型**

引用包含4种不同的类型：强引用、软引用、弱引用、虚引用；不同的引用类型,主要体现的是对象不同的可达性状态和对垃圾回收的影响。

**强引用**

强引用是使用最普遍的引用。如果一个对象具有强引用,那垃圾回收器绝不会回收它。当内存空间不足,Java虚拟机宁愿抛出OutOfMemoryError错误,使程序异常终止,也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**软引用**

软引用是用来描述一些有用但并不是必需的对象,在Java中用java.lang.ref.

SoftReference类来表示。对于软引用关联着的对象,只有在内存不足的时候JVM才会回收该对象。因此,这一点可以很好地用来解决OOM的问题,并且这个特性很适合用来实现缓存：比如网页缓存、图片缓存等。

**弱引用**

弱引用也是用来描述非必需对象的,当JVM进行垃圾回收时,无论内存是否充足,都会回收被弱引用关联的对象。在java中,用java.lang.ref.WeakReference类来表示。

**虚引用**

和前面的软引用、弱引用不同,它并不影响对象的生命周期。在java中用java.lang.ref.

PhantomReference类表示。如果一个对象与虚引用关联,则跟没有引用与之关联一样,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关联使用,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,就会把这个虚引用加入到与之关联的引用队列中。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必要的行动。

**Java虚拟机 jvm**