## 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

维持线程封闭性的方法。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。

## 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。

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。

### 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)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.

## 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

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；

抽象类必须有子类,使用extends继承,一个子类只能继承一个抽象类；

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）；

**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

## 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

## 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

ClassLoader的作用是将class文件加载到jvm虚拟机中去,程序就可以正确运行。但是,jvm启动的时候,并不会一次性加载所有的class文件,而是根据需要去动态加载。

**JAVA类加载流程**

Java语言系统自带有三个类加载器:   
**Bootstrap ClassLoader**最顶层的加载类,主要加载核心类库,%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。另外需要注意的是可以通过启动jvm时指定-Xbootclasspath和路径来改变Bootstrap ClassLoader的加载目录。比如java -Xbootclasspath/a:path被指定的文件追加到默认的bootstrap路径中。我们可以打开我的电脑,在上面的目录下查看,看看这些jar包是不是存在于这个目录。   
**Extention ClassLoader** 扩展的类加载器,加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。   
**Appclass Loader也称为SystemAppClass** 加载当前应用的classpath的所有类。