并发编程进阶篇

尹洪亮 | Kevin.Yin

互联网架构师 / 自由讲师

每天都要让自己比别人多努力一分钟

Kevin、让你每天进步一点点

我的微信 liang19871023liang

加微信，获取**项目源码、高清课件**

加微信，所有**新课七折**优惠，职业规划，互动答疑，免费资料

加微信，受邀进入**KEVIN社区**，与大咖和同龄人会面

关注公众号，每周推送**Kevin原创文章**，不定期优惠活动

1、课程内容介绍

- volatile
- Actomic
- ThreadLocal
- 同步类容器
- 并发类容器
- 并发无阻塞式队列
- 并发阻塞式队列

2、 volatile关键字

- 作用：：private volatile int a = 0;
- 强制线程到共享内存中读取数据，而不从线程工作内存中读取，从而使变量在多个线程间可见。
- 示例： DemoThread13

- volatile无法保证原子性，volatile属于轻量级的同步，性能比synchronized强很多(不加锁)，但是只保证线程见的可见性，并不能替代synchronized的同步功能，netty框架中大量使用了volatile
- 示例：DemoThread14

3、 volatile与static区别

- Static保证唯一性, 不保证一致性，多个实例共享一个静态变量。
- Volatile保证一致性，不保证唯一性，多个实例有多个volatile变量。

4、 Atomic类的原子性

- 使用AtomicInteger等原子类可以保证共享变量的原子性
- 示例：DemoThread15

- 但是Atomic类不能保证成员方法的原子性
- 示例：DemoThread16

- Actomic类采用了CAS这种非锁机制

CAS

Compare and swap

CAS – 原理

知识点

1. JDK提供的非阻塞原子操作，通过硬件保证了比较、更新操作的原子性
2. JDK的Unsafe类提供了一系列的compareAndSwap*方法来支持CAS操作

原理

线程1
希望修改x=2

1. 读取x
2. 比当前x是否还等于(1)读取的值
3. 如果等于(1)读取的值则更新x=2
4. 如果不等于重新回到(1)执行

共享变量x

int x =1

线程2
希望修改x=3

1. 读取x
2. 比当前x是否还等于(1)读取的值
3. 如果等于(1)读取的值则更新x=3
4. 如果不等于重新回到(1)执行

CAS – ABA问题

ABA问题

线程1
希望修改x=C

1.读取x=A
4.比当前x是否还等于A读取的值
5.发现等于1则更新x=C

共享变量x

int x =A

线程2
希望修改x=B、再修改x=A

2.修改x=B成功
3.修改x=A成功

知识点

1. 如果程序按照1~5的顺序执行，依然是成功的，然而线程1修改时x的值时其实已经从x= A =>B =>A。
2. 由于变量的值产生了环形转换，从A变为B又变回了A。如果不存在环形转换也就不存在ABA问题。

CAS – 解决ABA问题

知识点

1. 给变量分配时间戳、版本来解决ABA问题
2. JDK中使用java.util.concurrent.atomic.AtomicStampedReference类给每个变量的状态都分配一个时间戳，避免ABA问题产生。

Demo

com.mkevin.demo2.CasDemo0

5、ThreadLocal

- 使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本
- 示例： DemoThread21

ThreadLocal源码解析

源码解析

ThreadLocal源码解析

知识点

1. Thread类中的threadLocals、inheritableThreadLocals成员变量为ThreadLocal.ThreadLocalMap对象
2. Map的key值是ThreadLocal对象本身，查看set、set、remove方法
3. Thread无法解决继承问题，而InheritableThreadLocal可以
4. InheritableThreadLocal继承自ThreadLocal
5. InheritableThreadLocal可以帮助我们做链路追踪

Demo

com.mkevin.demo2.ThreadLocalDemo1
com.mkevin.demo2.ThreadLocalDemo0

Unsafe方法详解

Unsafe实操

Unsafe – 突破安全限制

知识点

1. 通过反射模式可以突破Unsafe类的安全限制

实操演练

com.mkevin.demo3.UnsafeDemo1

Unsafe – 普通字段操作

知识点 地址类操作

1. objectFieldOffset 获取字段偏移地址
2. staticFieldOffset 获取静态字段偏移地址
3. arrayBaseOffset 获取数组中第一个元素的地址
4. arrayIndexScale 获取数组中一个元素占用的字节

知识点 get、put

1. getInt、getLong、getBoolean、getChar、getFloat、getByte、getDouble、getObject 获取字段值
2. putInt、putLong、putBoolean、putChar、putFloat、putByte、putDouble、putObject 设置字段值
3. 直接操作内存地址
4. 通过对象内存地址操作

Unsafe – 普通字段操作

知识点 volatile类操作

1. getIntVolatile、getLongVolatile、getBooleanVolatile、getObjectVolatile、getByteVolatile、getShortVolatile、getCharVolatile、getFloatVolatile、getDoubleVolatile
   获取volatile字段值，保证可见性
1. putIntVolatile、putLongVolatile、putBooleanVolatile、putObjectVolatile、putByteVolatile、putShortVolatile、putCharVolatile、putFloatVolatile、putDoubleVolatile
   设置volatile字段值，保证可见性

知识点 and类操作

1. putOrderedInt、 putOrderedLong、 putOrderedObject 保证顺序性、具有lazy特性、不保证可见性
2. getAndSetInt、getAndSetLong、getAndSetObject 自旋操作、先获取后设置
3. getAndAddInt、getAndAddLong 自旋操作、先获取后设置
4. compareAndSwapInt、compareAndSwapLong、compareAndSwapObject CAS相关操作

Unsafe – 内存操作

内存操作

1. public native long allocateMemory(long bytes); 分配内存
2. public native long reallocateMemory(long address, long bytes); 重新分配内存
3. public native void setMemory(Object o, long offset, long bytes, byte value);
4. public void setMemory(long address, long bytes, byte value) 初始化内存
5. public native void copyMemory(Object srcBase, long srcOffset, Object destBase, long destOffset, long bytes);
6. public void copyMemory(long srcAddress, long destAddress, long bytes) 复制内存
7. public native void freeMemory(long address);

Demo

com.mkevin.demo3.UnsafeDemo5

Unsafe – 线程调度

线程调度

1. public native void park(boolean isAbsolute, long time)；挂起线程
2. public native void unpark(Object thread)；唤醒线程
3. 需要注意线程的interrupt方法同样能唤醒线程，但是不报错
4. java.util.concurrent.locks.LockSupport使用unsafe实现

Demo

com.mkevin.demo3.UnsafeDemo7

Unsafe

内存屏障

1. public native void loadFence**();** 保证在这个屏障之前的所有读操作都已经完成
2. public native void storeFence**();** 保证在这个屏障之前的所有写操作都已经完成
3. public native void fullFence**();** 保证在这个屏障之前的所有读写操作都已经完成

类加载

1. public native Class<?> defineClass(String name, byte[] b, int off, int len, ClassLoader loader, ProtectionDomain protectionDomain); 方法定义一个类，用于动态地创建类。
2. public native Class**<?>** defineAnonymousClass**(**Class**<?>** hostClass**,** byte**[]** data**,** Object**[]** cpPatches**);**用于动态的创建一个匿名内部类。
3. public native Object allocateInstance**(**Class**<?>** cls**) throws** InstantiationException**;**方法用于创建一个类的实例，但是不会调用这个实例的构造方法，如果这个类还未被初始化，则初始化这个类。
4. public native boolean shouldBeInitialized**(**Class**<?>** c**);**方法用于判断是否需要初始化一个类。
5. public native void ensureClassInitialized**(**Class**<?>** c**);**方法用于保证已经初始化过一个类。

7、同步类容器

- Vector、HashTable等古老的并发容器，都是使用Collections.synchronizedXXX等工厂方法创建的，并发状态下只能有一个线程访问容器对象，性能很低

- 示例：DemoThread26（古老容器的线程安全实现方法）

8、并发类容器

- JDK5.0之后提供了多种并发类容易可以替代同步类容器，提升性能、吞吐量

- ConcurrentHashMap替代HashMap、HashTable
- ConcurrentSkipListMap替代TreeMap

- ConcurrentHashMap将hash表分为16个segment，每个segment单独进行锁控制，从而减小了锁的粒度，提升了性能

- 例子： DemoThread27 (ConcurrentHashMap、 ConcurrentSkipListMap)

9、并发类容器

- Copy On Write容器,简称COW;写时复制容器，向容器中添加元素时，先将容器进行Copy出一个新容器，然后将元素添加到新容器中，再将原容器的引用指向新容器。并发读的时候不需要锁定容器，因为原容器没有变化，使用的是一种读写分离的思想。由于每次更新都会复制新容器，所以如果数据量较大，并且更新操作频繁则对内存消耗很高，建议在高并发读的场景下使用
- CopyOnWriteArraySet基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法, adIfAbsent方法同样采用锁保护，并创建一个新的大小+1的Object数组。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。从以上分析可见，CopyOnWriteArraySet在add时每次都要进行数组的遍历，因此其性能会低于CopyOnWriteArrayList.
- 示例： DemoThread28

10、并发-无阻赛队列

- ConcurrentLinkedQueue并发无阻塞队列，BlockingQueue并发阻塞队列，均实现自Queue接口
- ConcurrentLinkedQueue无阻塞、无锁、高性能、无界、线程安全，性能优于BlockingQueue、不允许null值
- 示例：DemoThread29 (ConcurrentLinkedQueue)

11、并发-阻塞队列- ArrayBlockingQueue

- ArrayBlockingQueue：基于数组实现的阻塞有界队列、创建时可指定长度，内部实现维护了一个定长数组用于缓存数据,内部没有采用读写分离，写入和读取数据不能同时进行，不允许null值
- 示例： DemoThread30

12、并发-阻塞队列- LinkedBlockingQueue

- LinkedBlockingQueue ：基于链表的阻塞队列,内部维护一个链表存储缓存数据， 支持写入和读取的并发操作， 创建时可指定长度也可以不指定，不指定时代表无界队列， 不允许null值
- 示例： DemoThread31

13、并发-阻塞队列- SynchronousQueue

- SynchronousQueue ：没有任何容量，必须现有线程先从队列中take,才能向queue中add数据，否则会抛出队列已满的异常。不能使用peek方法取数据,此方法底层没有实现,会直接返回null

- 示例： DemoThread32

14、并发-阻塞队列- PriorityBlockingQueue

- PriorityBlockingQueue：一个无界阻塞队列，默认初始化长度11，也可以手动指定，但是队列会自动扩容。资源被耗尽时导致 OutOfMemoryError。不允许使用 null元素。不允许插入不可比较的对象（导致抛出 ClassCastException），加入的对象实现Comparable接口

- 示例： DemoThread33

15、并发-阻塞队列- DelayQueue

- DelayQueue：Delayed 元素的一个无界阻塞队列，只有在延迟期满时才能从中提取元素。该队列的头部 是延迟期满后保存时间最长的 Delayed 元素。如果延迟都还没有期满，则队列没有头部，并且 poll 将返回 null。当一个元素的 getDelay(TimeUnit.NANOSECONDS) 方法返回一个小于等于 0 的值时，将发生到期。即使无法使用 take 或 poll 移除未到期的元素，也不会将这些元素作为正常元素对待。例如，size 方法同时返回到期和未到期元素的计数。此队列不允许使用 null 元素。内部元素需实现Delayed接口
- 场景：缓存到期删除、任务超时处理、空闲链接关闭等

- 示例： DemoThread34

精品教程

JAVA并发编程系列

一次性搞定数据库事务

Memcached系列

一次性精通JVM

Disruptor高并发框架

SpringCloud微服务架构

RateLimiter访问限流

程序员转型项目经理

合作平台

扫描二维码学习更多教程

腾讯课堂 学习成就梦想

网易云课堂 我的职业课堂

51CTO.com 技术成就梦想