chenxibobo

公告

昵称： chenxibobo
园龄： 5年11个月
粉丝： 43
关注： 2
+加关注

< 2022年3月 >

日	一	二	三	四	五	六
27	28	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类 (535)

*Java集合HashMap(21)
*Java内存与JVM(42)
*Java算法(3)
*安卓framework(18)
*安卓-跨进程通信(43)
*安卓-列表View(22)
*安卓-内存(18)
*安卓-四大组件(27)
*安卓-网络请求(30)
*安卓-性能优化(12)
*安卓-知识碎片(32)
*感悟总结与学习(5)
@Java反射(1)
@Java泛型(4)
@Java基础(25)
@Java集合(7)
@Java线程(18)
@Java线程volatile(7)
@Java线程同步与锁(8)

ABA问题的本质及其解决办法#

简介

CAS的全称是compare and swap，它是java同步类的基础，java.util.concurrent中的同步类基本上都是使用CAS来实现其原子性的。

CAS的原理其实很简单，为了保证在多线程环境下我们的更新是符合预期的，或者说一个线程在更新某个对象的时候，没有其他的线程对该对象进行修改。在线程更新某个对象（或值）之前，先保存更新前的值，然后在实际更新的时候传入之前保存的值，进行比较，如果一致的话就进行更新，否则失败。

注意，CAS在java中是用native方法来实现的，利用了系统本身提供的原子性操作。

那么CAS在使用中会有什么问题呢？一般来说CAS如果设计的不够完美的话，可能会产生ABA问题，而ABA问题又可以分为两类，我们先看来看一类问题。

第一类问题

我们考虑下面一种ABA的情况：

1. 在多线程的环境中，线程a从共享的地址X中读取到了对象A。
2. 在线程a准备对地址X进行更新之前，线程b将地址X中的值修改为了B。
3. 接着线程b将地址X中的值又修改回了A。
4. 最新线程a对地址X执行CAS，发现X中存储的还是对象A，对象匹配，CAS成功。

上面的例子中CAS成功了，但是实际上这个CAS并不是原子操作，如果我们想要依赖CAS来实现原子操作的话可能就会出现隐藏的bug。

第一类问题的关键就在2和3两步。这两步我们可以看到线程b直接替换了内存地址X中的内容。

在拥有自动GC环境的编程语言，比如说java中，2，3的情况是不可能出现的，因为在java中，只要两个对象的地址一致，就表示这两个对象是相等的。

2，3两步可能出现的情况就在像C++这种，不存在自动GC环境的编程语言中。因为可以自己控制对象的生命周期，如果我们从一个list中删除掉了一个对象，然后又重新分配了一个对象，并将其add back到list中去，那么根据 MRU memory allocation算法，这个新的对象很有可能和之前删除对象的内存地址是一样的。这样就会导致ABA的问题。

第二类问题

如果我们在拥有自动GC的编程语言中，那么是否仍然存在CAS问题呢？

考虑下面的情况，有一个链表里面的数据是A->B->C,我们希望执行一个CAS操作，将A替换成D，生成链表D->B->C。考虑下面的步骤：

1. 线程a读取链表头部节点A。
2. 线程b将链表中的B节点删掉，链表变成了A->C
3. 线程a执行CAS操作，将A替换从D。

最后我们的到的链表是D->C，而不是D->B->C。

问题出在哪呢？CAS比较的节点A和最新的头部节点是不是同一个节点，它并没有关心节点A在步骤1和3之间是否内容发生变化。

我们举个例子：

```
public void useABAReference(){
        CustUser a= new CustUser();
        CustUser b= new CustUser();
        CustUser c= new CustUser();
        AtomicReference<CustUser> atomicReference= new AtomicReference<>(
        log.info("{}",atomicReference.compareAndSet(a,b));
        log.info("{}",atomicReference.compareAndSet(b,a));
        a.setName("change for new name");
        log.info("{}",atomicReference.compareAndSet(a,c));
    }
```

上面的例子中，我们使用了AtomicReference的CAS方法来判断对象是否发生变化。在CAS b和a之后，我们将a的name进行了修改，我们看下最后的输出结果：

```
[main] INFO com.flydean.aba.ABAUsage - true
[main] INFO com.flydean.aba.ABAUsage - true
[main] INFO com.flydean.aba.ABAUsage - true
```

三个CAS的结果都是true。说明CAS确实比较的两者是否为统一对象，对其中内容的变化并不关心。

第二类问题可能会导致某些集合类的操作并不是原子性的，因为你并不能保证在CAS的过程中，有没有其他的节点发送变化。

第一类问题的解决

第一类问题在存在自动GC的编程语言中是不存在的，我们主要看下怎么在C++之类的语言中解决这个问题。

根据官方的说法，第一类问题大概有四种解法：

1. 使用中间节点 - 使用一些不代表任何数据的中间节点来表示某些节点是标记被删除的。
2. 使用自动GC。
3. 使用hazard pointers - hazard pointers 保存了当前线程正在访问的节点的地址，在这些hazard pointers中的节点不能够被修改和删除。
4. 使用read-copy update (RCU) - 在每次更新的之前，都做一份拷贝，每次更新的是拷贝出来的新结构。

第二类问题的解决

第二类问题其实算是整体集合对象的CAS问题了。一个简单的解决办法就是每次做CAS更新的时候再添加一个版本号。如果版本号不是预期的版本，就说明有其他的线程更新了集合中的某些节点，这次CAS是失败的。

我们举个AtomicStampedReference的例子：

```
public void useABAStampReference(){
        Object a= new Object();
        Object b= new Object();
        Object c= new Object();
        AtomicStampedReference<Object> atomicStampedReference= new Atomic
        log.info("{}",atomicStampedReference.compareAndSet(a,b,0,1));
        log.info("{}",atomicStampedReference.compareAndSet(b,a,1,2));
```

2015年9月(5)
2015年6月(1)
2015年5月(1)
2015年4月(7)
2015年1月(1)

阅读排行榜

1. Android Context完全解析与各种获取Context方法#(74076)
2. Groovy脚本基础全攻略(28182)
3. Android分包方案multidex*(25047)
4. Android启动Activity的两种方式与四种启动模式*(24388)
5. Java中<? extends T>和<? super T>的理解#(23581)
6. @Android OkHttp3简介和使用详解(18240)
7. Android中为什么主线程不会因为Looper.loop()方法造成阻塞*(18236)
8. @Android属性动画完全解析(15768)
9. AndroidStudio项目CMakeLists解析(14374)
10. Android Studio NDK编程-环境搭建及Hello!(10403)
11. Android如何判断当前手机是否正在播放音乐，并获取到正在播放的音乐的信息*(10343)
12. Android 获取视频照片与刷新媒体库*(9886)
13. Android静默安装实现方案*(7716)
14. Android——NativeActivity - C/C++ Apk开发(7092)
15. Android跨进程通信广播（Broadcast）*(6843)
16. @Android之View的绘制流程(5225)
17. @Android动态方式破解apk进阶篇(IDA调试so源码)(5123)
18. Android jni/ndk编程五：jni异常处理(4984)
19. @Android MVP 设计模式(4823)
20. Android开发之Theme、Style探索及源码浅析*(4803)

评论排行榜

1. Java中<? extends T>和<? super T>的理解#(5)
2. @Android属性动画完全解析(2)

```
        log.info("{}",atomicStampedReference.compareAndSet(a,c,0,1));
    }
```

AtomicStampedReference的compareAndSet方法，多出了两个参数，分别是expectedStamp和newStamp，两个参数都是int型的，需要我们手动传入。

总结

ABA问题其实是由两类问题组成的，需要我们分开来对待和解决。

分类: @Java线程

chenxibobo
关注 - 2
粉丝 - 43
+加关注

« 上一篇： Java内存模型#
» 下一篇： 一个简单的死锁代码#

posted @ 2020-07-21 17:24 chenxibobo 阅读(1614) 评论(0) 编辑 收藏 举报