

21 | 原子类：无锁工具类的典范

王宝令 2019-04-16



00:00

12:32

讲述：王宝令 大小：11.48M

前面我们多次提到一个累加器的例子，示例代码如下。在这个例子中，`add10K()` 这个方法不是线程安全的，问题就出在变量 `count` 的可见性和 `count+=1` 的原子性上。可见性问题可以用 `volatile` 来解决，而原子性问题我们前面一直都是采用的互斥锁方案。

```
1 public class Test {
2     long count = 0;
3     void add10K() {
4         int idx = 0;
5         while(idx++ < 10000) {
6             count += 1;
7         }
8     }
9 }
10
```

复制代码

其实对于简单的原子性问题，还有一种**无锁方案**。Java SDK 并发包将这种无锁方案封装提炼之后，实现了一系列的原子类。不过，在深入介绍原子类的实现之前，我们先看看如何利用原子类解决累加器问题，这样你会对原子类有个初步的认识。

在下面的代码中，我们将原来的 `long` 型变量 `count` 替换为了原子类 `AtomicLong`，原来的 `count += 1`

替换成了 `count.getAndIncrement()`，仅需要这两处简单的改动就能使 `add10K()` 方法变成线程安全的，原子类的使用还是挺简单的。

```

1 public class Test {
2     AtomicLong count =
3         new AtomicLong(0);
4     void add10K() {
5         int idx = 0;
6         while(idx++ < 10000) {
7             count.getAndIncrement();
8         }
9     }
10 }
11

```

无锁方案相对互斥锁方案，最大的好处就是**性能**。互斥锁方案为了保证互斥性，需要执行加锁、解锁操作，而加锁、解锁操作本身就消耗性能；同时拿不到锁的线程还会进入阻塞状态，进而触发线程切换，线程切换对性能的消费也很大。相比之下，无锁方案则完全没有加锁、解锁的性能消耗，同时还能保证互斥性，既解决了问题，又没有带来新的问题，可谓绝佳方案。那它是如何做到的呢？

无锁方案的实现原理

其实原子类性能高的秘密很简单，硬件支持而已。CPU 为了解决并发问题，提供了 CAS 指令（CAS，全称是 Compare And Swap，即“比较并交换”）。CAS 指令包含 3 个参数：共享变量的内存地址 A、用于比较的值 B 和共享变量的新值 C；并且只有当内存中地址 A 处的值等于 B 时，才能将内存中地址 A 处的值更新为新值 C。**作为一条 CPU 指令，CAS 指令本身是能够保证原子性的。**

你可以通过下面 CAS 指令的模拟代码来理解 CAS 的工作原理。在下面的模拟程序中有两个参数，一个是期望值 expect，另一个是需要写入的新值 newValue，**只有当目前 count 的值和期望值 expect 相等时，才会将 count 更新为 newValue。**

```

1 class SimulatedCAS{
2     int count;
3     synchronized int cas(
4         int expect, int newValue){
5         // 读目前 count 的值
6         int curValue = count;
7         // 比较目前 count 值是否 == 期望值
8         if(curValue == expect){
9             // 如果是，则更新 count 的值
10            count = newValue;
11        }
12        // 返回写入前的值
13        return curValue;
14    }
15 }
16

```

你仔细地再次思考一下这句话，“**只有当目前 count 的值和期望值 expect 相等时，才会将 count 更新为 newValue。**”要怎么理解这句话呢？


对于前面提到的累加器的例子，count += 1

的一个核心问题是：基于内存中 count 的当前值 A 计算出来的 count+=1 为 A+1，在将 A+1 写入

内存的时候，很可能此时内存中 count 已经被其他线程更新过了，这样就会导致错误地覆盖其他线程写入的值（如果你觉得理解起来还有困难，建议你再重新看看[《01 | 可见性、原子性和有序性问题：并发编程 Bug 的源头》](#)）。也就是说，只有当内存中 count 的值等于期望值 A 时，才能将内存中 count 的值更新为计算结果 A+1，这不就是 CAS 的语义吗！

使用 CAS 来解决并发问题，一般都会伴随着自旋，而所谓自旋，其实就是循环尝试。例如，实现一个线程安全的count += 1

操作，“CAS+ 自旋”的实现方案如下所示，首先计算 newValue = count+1，如果 cas(count,newValue) 返回的值不等于 count，则意味着线程在执行完代码①处之后，执行代码②处之前，count 的值被其他线程更新过。那此时该怎么处理呢？可以采用自旋方案，就像下面代码中展示的，可以重新读 count 最新的值来计算 newValue 并尝试再次更新，直到成功。

 复制代码

```
1 class SimulatedCAS{
2     volatile int count;
3     // 实现 count+=1
4     addOne(){
5         do {
6             newValue = count+1; //①
7         }while(count !=
8             cas(count,newValue) //②
9         }
10    // 模拟实现 CAS，仅用来帮助理解
11    synchronized int cas(
12        int expect, int newValue){
13        // 读目前 count 的值
14        int curValue = count;
15        // 比较目前 count 值是否 == 期望值
16        if(curValue == expect){
17            // 如果是，则更新 count 的值
18            count= newValue;
19        }
20        // 返回写入前的值
21        return curValue;
22    }
23 }
24
```

通过上面的示例代码，想必你已经发现了，CAS 这种无锁方案，完全没有加锁、解锁操作，即便两个线程完全同时执行 addOne() 方法，也不会有线程被阻塞，所以相对于互斥锁方案来说，性能好了很多。

但是在 CAS 方案中，有一个问题可能会常被你忽略，那就是ABA的问题。什么是 ABA 问题呢？

前面我们提到“如果 cas(count,newValue) 返回的值**不等于**count，意味着线程在执行完代码①处之后，执行代码②处之前，count 的值被其他线程**更新过**”，那如果 cas(count,newValue) 返回的值**等于**count，是否就能够认为 count 的值没有被其他线程**更新过**呢？显然不是的，假设 count 原本是 A，线程 T1 在执行完代码①处之后，执行代码②处之前，有可能 count 被线程 T2 更新成了 B，之后又被 T3 更新回了 A，这样线程 T1 虽然看到的一直是 A，但是其实已经被其他线程更新过了，这就是 ABA 问题。

可能大多数情况下我们并不关心 ABA 问题，例如数值的原子递增，但也不能所有情况下都不关心，例如原子化的更新对象很可能就需要关心 ABA 问题，因为两个 A 虽然相等，但是第二个 A 的属性可能已经发生变化了。所以在使用 CAS 方案的时候，一定要先 check 一下。

看 Java 如何实现原子化的 count += 1

在本文开始部分，我们使用原子类 AtomicLong 的 getAndIncrement() 方法替代了 count += 1

，从而实现了线程安全。原子类 AtomicLong 的 getAndIncrement() 方法内部就是基于 CAS 实现的，下面我们来看看 Java 是如何使用 CAS 来实现原子化的 count += 1 的。

在 Java 1.8 版本中，getAndIncrement() 方法会转调 unsafe.getAndAddLong() 方法。这里 this 和 valueOffset 两个参数可以唯一确定共享变量的内存地址。

```
1 final long getAndIncrement() {
2     return unsafe.getAndAddLong(
3         this, valueOffset, 1L);
4 }
5
```

复制代码

unsafe.getAndAddLong() 方法的源码如下，该方法首先会在内存中读取共享变量的值，之后循环调用 compareAndSwapLong() 方法来尝试设置共享变量的值，直到成功为止。

compareAndSwapLong() 是一个 native 方法，只有当内存中共享变量的值等于 expected 时，才会将共享变量的值更新为 x，并且返回 true；否则返回 false。compareAndSwapLong 的语义和 CAS 指令的语义的差别仅仅是返回值不同而已。

```
1 public final long getAndAddLong(
2     Object o, long offset, long delta){
3     long v;
4     do {
5         // 读取内存中的值
6         v = getLongVolatile(o, offset);
7     } while (!compareAndSwapLong(
8         o, offset, v, v + delta));
9     return v;
10 }
11 // 原子性地将变量更新为 x
12 // 条件是内存中的值等于 expected
13 // 更新成功则返回 true
14 native boolean compareAndSwapLong(
15     Object o, long offset,
16     long expected,
17     long x);
18
```

复制代码

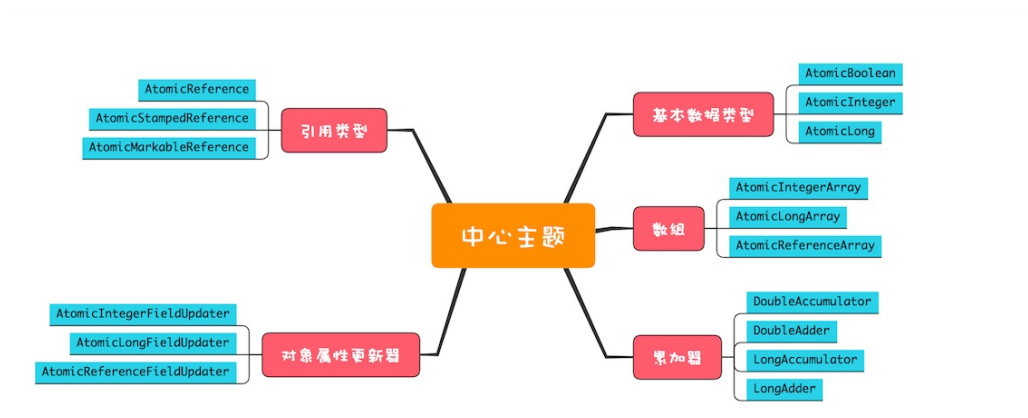
另外，需要你注意的是，getAndAddLong() 方法的实现，基本上就是 CAS 使用的经典范例。所以请你再次体会下面这段抽象后的代码片段，它在很多无锁程序中经常出现。Java 提供的原子类里面 CAS 一般被实现为 compareAndSet()，compareAndSet() 的语义和 CAS 指令的语义的差别仅仅是返回值不同而已，compareAndSet() 里面如果更新成功，则会返回 true，否则返回 false。

```
1 do {
2     // 获取当前值
3     oldV = xxx;
4     // 根据当前值计算新值
5     newV = ...oldV...
6 }while(!compareAndSet(oldV,newV);
7
```

复制代码

原子类概览

Java SDK 并发包里提供的原子类内容很丰富，我们可以将它们分为五个类别：**原子化的基本数据类型**、**原子化的对象引用类型**、**原子化数组**、**原子化对象属性更新器**和**原子化的累加器**。这五个类别提供的方法基本上是相似的，并且每个类别都有若干原子类，你可以通过下面的原子类组成概览图来获得一个全局的印象。下面我们详细解读这五个类别。



原子类组成概览图

1. 原子化的基本数据类型

相关实现有 AtomicBoolean、AtomicInteger 和 AtomicLong，提供的方法主要有以下这些，详情你可以参考 SDK 的源代码，都很简单，这里就不详细介绍了。

```

1 getAndIncrement() // 原子化 i++
2 getAndDecrement() // 原子化的 i--
3 incrementAndGet() // 原子化的 ++i
4 decrementAndGet() // 原子化的 --i
5 // 当前值 +=delta, 返回 += 前的值
6 getAndAdd(delta)
7 // 当前值 +=delta, 返回 += 后的值
8 addAndGet(delta)
9 //CAS 操作, 返回是否成功
10 compareAndSet(expect, update)
11 // 以下四个方法
12 // 新值可以通过传入 func 函数来计算
13 getAndUpdate(func)
14 updateAndGet(func)
15 getAndAccumulate(x, func)
16 accumulateAndGet(x, func)
17

```

复制代码


2. 原子化的对象引用类型

相关实现有 AtomicReference、AtomicStampedReference 和 AtomicMarkableReference，利用它们可以实现对象引用的原子化更新。AtomicReference 提供的方法和原子化的基本数据类型差不多，这里不再赘述。不过需要注意的是，对象引用的更新需要重点关注 ABA 问题，AtomicStampedReference 和 AtomicMarkableReference 这两个原子类可以解决 ABA 问题。

解决 ABA 问题的思路其实很简单，增加一个版本号维度就可以了，这个和我们在 [《18 | StampedLock：有没有比读写锁更快的锁？》](#) 介绍的乐观锁机制很类似，每次执行 CAS 操作，附

加再更新一个版本号，只要保证版本号是递增的，那么即便 A 变成 B 之后再变回 A，版本号也不会变回来（版本号递增的）。AtomicStampedReference 实现的 CAS 方法就增加了版本号参数，方法签名如下：

```
1 boolean compareAndSet(  
2     V expectedReference,  
3     V newReference,  
4     int expectedStamp,  
5     int newStamp)  
6
```

 复制代码

AtomicMarkableReference 的实现机制则更简单，将版本号简化成了一个 Boolean 值，方法签名如下：

```
1 boolean compareAndSet(  
2     V expectedReference,  
3     V newReference,  
4     boolean expectedMark,  
5     boolean newMark)  
6
```

 复制代码

3. 原子化数组

相关实现有 AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray，利用这些原子类，我们可以原子化地更新数组里面的每一个元素。这些类提供的方法和原子化的基本数据类型的区别仅仅是：每个方法多了一个数组的索引参数，所以这里也不再赘述了。

4. 原子化对象属性更新器

相关实现有 AtomicIntegerFieldUpdater、AtomicLongFieldUpdater 和 AtomicReferenceFieldUpdater，利用它们可以原子化地更新对象的属性，这三个方法都是利用反射机制实现的，创建更新器的方法如下：

```
1 public static <U>  
2     AtomicXXXFieldUpdater<U>  
3     newUpdater(Class<U> tclass,  
4         String fieldName)  
5
```

 复制代码

需要注意的是，**对象属性必须是 volatile 类型的，只有这样才能保证可见性**；如果对象属性不是 volatile 类型的，newUpdater() 方法会抛出 IllegalArgumentException 这个运行时异常。

你会发现 newUpdater() 的方法参数只有类的信息，没有对象的引用，而更新**对象**的属性，一定需要对象的引用，那这个参数是在哪里传入的呢？是在原子操作的方法参数中传入的。例如 compareAndSet() 这个原子操作，相比原子化的基本数据类型多了一个对象引用 obj。原子化对象属性更新器相关的方法，相比原子化的基本数据类型仅仅是多了对象引用参数，所以这里也不再赘述了。

 复制代码

```
1 boolean compareAndSet(  
2     T obj,  
3     int expect,  
4     int update)  
5
```

5. 原子化的累加器

DoubleAccumulator、DoubleAdder、LongAccumulator 和 LongAdder，这四个类仅仅用来执行累加操作，相比原子化的基本数据类型，速度更快，但是不支持 compareAndSet() 方法。如果你仅仅需要累加操作，使用原子化的累加器性能会更好。


总结

无锁方案相对于互斥锁方案，优点非常多，首先性能好，其次是基本不会出现死锁问题（但可能出现饥饿和活锁问题，因为自旋会反复重试）。Java 提供的原子类大部分都实现了 compareAndSet() 方法，基于 compareAndSet() 方法，你可以构建自己的无锁数据结构，但是**建议你不要这样做，这个工作最好还是让大师们去完成**，原因是无锁算法没你想象的那么简单。

Java 提供的原子类能够解决一些简单的原子性问题，但你可能会发现，上面我们所有原子类的方法都是针对一个共享变量的，如果你需要解决多个变量的原子性问题，建议还是使用互斥锁方案。原子类虽好，但使用要慎之又慎。

课后思考

下面的示例代码是合理库存的原子化实现，仅实现了设置库存上限 setUpper() 方法，你觉得 setUpper() 方法的实现是否正确呢？

 复制代码

```
1 public class SafeWM {  
2     class WMRange{  
3         final int upper;  
4         final int lower;  
5         WMRange(int upper,int lower){  
6             // 省略构造函数实现  
7         }  
8     }  
9     final AtomicReference<WMRange>  
10    rf = new AtomicReference<>(  
11        new WMRange(0,0)  
12    );  
13    // 设置库存上限  
14    void setUpper(int v){  
15        WMRange nr;  
16        WMRange or = rf.get();  
17        do{  
18            // 检查参数合法性  
19            if(v < or.lower){  
20                throw new IllegalArgumentException();  
21            }  
22            nr = new  
23                WMRange(v, or.lower);  
24        }while(!rf.compareAndSet(or, nr));  
25    }  
26 }  
27
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



©



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(21)



郑晨Cc

or是原始的 nr是new出来的 指向不同的内存地址 compareandset的结果永远返回false 结果是死循环？是不是应该用atomicfieldreference？

👍 5 2019-04-16

作者回复: 👍，不过我觉得没必要用atomicfieldreference



aroll

ABA问题，用AtomicStampedReference 或 AtomicMarkableReference比较好

👍 1 2019-04-16



小萝卜

第16行应该放在循环体内

👍 2019-04-16



张天屹

如果线程1 运行到WMRange or = rf.get();停止，切换到线程2 更新了值，切换回到线程1，进入循环将永远比较失败死循环，解决方案是将读取的那一句放入循环里，CAS每次自旋必须要重新检查新的值才有意义

👍 2019-04-16



magict4

可能会陷入死循环。

线程1执行完23行之后，被暂停。

线程2执行，并成功更新rf的内容。

线程1继续执行，24行while语句返回为false(因为rf内容已经被线程2更新)。

线程1重新进入do循环。注意此时or并没有被重新读取。while语句继续返回false，如此往复。

感觉把 WMRRange or = rf.get(); 这一行放到 do 内部，就可以了，不知道是否正确？



2019-04-16



crazypokerk

课后问题不正确，AtomicReference不能解决ABA问题。



2019-04-16



Zach_

```
public class SafeWM {
    class WMRRange{
        final int upper;
        final int lower;
        WMRRange(int upper,int lower){
            // 省略构造函数实现
        }
    }
    final AtomicReference<WMRange>
    rf = new AtomicReference<>{
        new WMRRange(0,0)
    };
    // 设置库存上限
    void setUpper(int v){
        WMRRange nr;
        WMRRange or = rf.get();
        do{
            // 检查参数合法性
            if(v < or.lower){
                throw new IllegalArgumentException();
            }
            nr = new
                WMRRange(v, or.lower);
        }while(!rf.compareAndSet(or, nr));
    }
}
```

不是很清楚AtomicReference.CAS比较的是对象内存中的地址，还是包括了属性内存中的地址。我猜想思考题这里写的是对的吧？

可是如果我说它是错的，我又不知道它错在哪里。



2019-04-16



周治慧

两个对象比较的时候是比较的地址值，old和new地址值一直是不等的，设置上限应该是去更新对象的字段的值保证字段值的cas



2019-04-16



zhangtnty

王老师好, 文中题目我认为不存在 ABA 的问题。问题是O r 的值应该放在 do 循环体内,如果两个线程 A , B。同时执行方法, A 执行完 B 却始终拿不到 A 的新值, 致 B 进入 死循环。

另外, 王老师能否针对无锁原子类的实际应用场景列举一些, 谢谢!

👍 2019-04-16



张三

对文中的 do{}while()循环有点困惑, 为什么不是while(){}?

👍 2019-04-16

作者回复: 等价的



张三

Java如何实现原子化的count+=1中, getLongVolatile () 仅仅是获取当前值, return回去的v并没有对当前值+1啊?

👍 2019-04-16



苏志辉

不安全, 因为cas只能保证引用一样, 没法保证属性没变

👍 2019-04-16



Felix Envy

WMRange or = rf.get(); 这段应该放到循环体里面去, 不然一旦发生并发就会有线程进入死循环。

👍 2019-04-16



狼

ABA问题

👍 2019-04-16



密码123456

饥饿的情况, 我能够想象出来, 可是出现活锁的情况, 我想象不出来? 知道的同学麻烦告知下。

👍 2019-04-16



张德

先打卡 一般老师讲的课 我都是看3遍

👍 2019-04-16



密码123456

我觉得可能会出现死循环。WMRange or = rf.get(); 应该放在do里面。每次比较交换失败后, 重新获取一次。

👍 2019-04-16

作者回复: 👍



Kid☺

打卡！对CAS理解还是不够，要反复多读几次文章了。另外有个小疑问，CAS中的自旋循环是否会有性能问题？



2019-04-16

作者回复: 有可能有，这个得看场景



kyle

synchronized 关键字不是隐含加锁操作吗？
为什么说cas是无锁的呢！



2019-04-16

作者回复: 只是为了让你理解原理，cas不是用sync实现的



张三

看了一遍，先打卡，对CAS+自旋还不太明白。看到上一篇文章留言人数少了很多。



2019-04-16



唐荣轩

某些指标统计程序、trace模型两ID模型中的SpanID（0.1.2，0.1.5）生成器可以基于原子类进行实现



2019-04-16