二

65 CAS 和乐观锁的关系，什么时候会用到 CAS?

在本课时中，我将讲解 CAS 的应用场景，什么时候会用到 CAS?

并发容器

Doug Lea 大神在 **JUC** 包中大量使用了 CAS 技术，该技术既能保证安全性，又不需要使用互斥锁，能大大提升工具类的性能。下面我将通过两个例子来展示 CAS 在并发容器中的使用情况。

案例一：ConcurrentHashMap

先来看看并发容器 ConcurrentHashMap 的例子，我们截取部分 putVal 方法的代码，如下所示：

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
```

```
    //以下部分省略
    ...
}
```

在第 10 行，有一个醒目的方法，它就是 “casTabAt”，这个方法名就带有 “CAS”，可以猜测它一定是和 CAS 密不可分了，下面给出 casTabAt 方法的代码实现：

```
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

该方法里面只有一行代码，即调用变量 U 的 **compareAndSwapObject** 的方法，那么，这个变量 U 是什么类型的呢？U 的定义是：

```
private static final sun.misc.Unsafe U
```

可以看出，U 是 **Unsafe** 类型的，Unsafe 类包含 compareAndSwapInt、compareAndSwapLong、compareAndSwapObject 等和 CAS 密切相关的 native 层的方法，其底层正是利用 CPU 对 CAS 指令的支持实现的。

上面介绍的 casTabAt 方法，不仅被用在了 ConcurrentHashMap 的 putVal 方法中，还被用在了 merge、compute、computeIfAbsent、transfer 等重要的方法中，所以 ConcurrentHashMap 对于 CAS 的应用是比较广泛的。

案例二：ConcurrentLinkedQueue

接下来，我们来看并发容器的第二个案例。非阻塞并发队列 ConcurrentLinkedQueue 的 offer 方法里也有 CAS 的身影，offer 方法的代码如下所示：

```
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);
    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
```

```
            if (q == null) {
                if (p.casNext(null, newNode)) {
                    if (p != t)
                        casTail(t, newNode);
                    return true;
                }
            }
            else if (p == q)
                p = (t != (t = tail)) ? t : head;
            else
                p = (p != t && t != (t = tail)) ? t : q;
        }
    }
```

可以看出，在 offer 方法中，有一个 for 循环，这是一个死循环，在第 8 行有一个与 CAS 相关的方法，是 **casNext** 方法，用于更新节点。那么如果执行 p 的 casNext 方法失败的话，casNext 会返回 false，那么显然代码会继续在 for 循环中进行下一次的尝试。所以在这里也可以很明显的看出 ConcurrentLinkedQueue 的 offer 方法使用到了 CAS。

以上就是 CAS 在并发容器中应用的两个例子，我们再来看一看 CAS 在数据库中有哪些应用。

数据库

在我们的数据库中，也存在对乐观锁和 CAS 思想的应用。在更新数据时，我们可以利用 version 字段在数据库中实现乐观锁和 CAS 操作，而在获取和修改数据时都不需要加悲观锁。

具体思路如下：当我们获取完数据，并计算完毕，准备更新数据时，会检查现在的版本号与之前获取数据时的版本号是否一致，如果一致就说明在计算期间数据没有被更新过，可以直接更新本次数据；如果版本号不一致，则说明计算期间已经有其他线程修改过这个数据了，那就可以选择重新获取数据，重新计算，然后再次尝试更新数据。

假设取出数据的时候 version 版本为 1，相应的 SQL 语句示例如下所示：

```
UPDATE student        SET            name = ‘小王’,            version = 2        WH
```

这样一来就可以用 CAS 的思想去实现本次的更新操作，它会先去比较 version 是不是最开始获取到的 1，如果和初始值相同才去进行 name 字段的修改，同时也要把 version 的值加一。

原子类

在原子类中，例如 AtomicInteger，也使用了 CAS，原子类的内容我们在第 39 课时中已经具体分析过了，现在我们复习一下和 CAS 相关的重点内容，也就是 AtomicInteger 的 getAndAdd 方法，该方法代码如下所示：

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}
```

从上面的三行代码中可以看到，return 的内容是 Unsafe 的 getAndAddInt 方法的执行结果，接下来我们来看一下 getAndAddInt 方法的具体实现，代码如下所示：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
    return var5;
}
```

在这里，我们看到上述方法中有对 var5 的赋值，调用了 unsafe 的 getIntVolatile(var1, var2) 方法，这是一个 native 方法，作用是获取变量 var1 中偏移量 var2 处的值。这里传入 var1 的是 AtomicInteger 对象的引用，而 var2 就是 AtomicInteger 里面所存储的数值（也就是 value）的偏移量 valueOffset，所以此时得到的 var5 实际上代表当前时刻下的原子类中存储的数值。

接下来重点来了，我们看到有一个 **compareAndSwapInt** 方法，这里会传入多个参数，分别是 var1、var2、 var5、var5 + var4，其实它们代表 object、offset、expectedValue 和

newValue。

- 第一个参数 object 就是将要修改的对象，传入的是 this，也就是 atomicInteger 这个对象本身；
- 第二个参数是 offset，也就是偏移量，借助它就可以获取到 value 的数值；
- 第三个参数 expectedValue，代表“期望值”，传入的是刚才获取到的 var5；
- 而最后一个参数 newValue 是希望修改为的新值 ，等于之前取到的数值 var5 再加上 var4，而 var4 就是我们之前所传入的 delta，delta 就是我们希望原子类所改变的数值，比如可以传入 +1，也可以传入 -1。

所以 compareAndSwapInt 方法的作用就是，判断如果现在原子类里 value 的值和之前获取到的 var5 相等的话，那么就把计算出来的 var5 + var4 给更新上去，所以说这行代码就实现了 CAS 的过程。

一旦 CAS 操作成功，就会退出这个 while 循环，但是也有可能操作失败。如果操作失败就意味着在获取到 var5 之后，并且在 CAS 操作之前，value 的数值已经发生变化了，证明有其他线程修改过这个变量。

这样一来，就会再次执行循环体里面的代码，重新获取 var5 的值，也就是获取最新的原子变量的数值，并且再次利用 CAS 去尝试更新，直到更新成功为止，所以这是一个死循环。

我们总结一下，Unsafe 的 getAndAddInt 方法是通过**循环 + CAS** 的方式来实现的，在此过程中，它会通过 compareAndSwapInt 方法来尝试更新 value 的值，如果更新失败就重新获取，然后再次尝试更新，直到更新成功。

总结

在本课时中，我们讲解了 CAS 的应用场景。在**并发容器**、**数据库**以及**原子类**中都有很多和 CAS 相关的代码，所以 CAS 有着广泛的应用场景，你需要清楚的了解什么情况下会用到 CAS。

上一页 下一页