乐观锁的思想就是CAS的运用,下面是CAS算法介绍:

CAS有3个操作数，内存值V，预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

值A是之前读取到内存值V，值B是值A基础上计算出来的，当V不等于A，说明在刚才计算B的期间内，内存值V已经被修改了，那么CAS不应该再修改了，可以为了避免多人修改导致出错（保证多人修改的线程安全）。

CAS的缺点:

1. **ABA问题**：如果内存值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。

   ABA的解决办法：使用版本号，在变量前面追加上版本号，比如每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A-2B-3A。

   扩展：**从Java1.5**开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
2. **循环时间长开销大**：自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。
3. **只能保证一个共享变量的原子操作**：对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。

   扩展：或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i = 2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了**AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。**

CAS 的源码:

```
public class SimulatedCAS {
    private int value;
    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue) {
            value = newValue;
        }
        return oldValue;
    }
}
//我们使用CAS的时候调用compareAndSwap就可以。
```

什么时候会用到 CAS:

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;
```

```
7           if (tab == null || (n = tab.length) == 0)
8               tab = initTable();
9           else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
10              if (casTabAt(tab, i, null,
11                           new Node<K,V>(hash, key, value, null)))
12                  break;                   // no lock when adding to empty bin
13          }
14      //以下部分省略
15      ...
16  }
```

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：

```
1  static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
2                                       Node<K,V> c, Node<K,V> v) {
3      return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
4  }
```

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：

```
1  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：

```
1  public boolean offer(E e) {
2      checkNotNull(e);
3      final Node<E> newNode = new Node<E>(e);
4
5      for (Node<E> t = tail, p = t;;) {
6          Node<E> q = p.next;
7          if (q == null) {
8              if (p.casNext(null, newNode)) {
9                  if (p != t)
10                     casTail(t, newNode);
11                 return true;
12             }
13         }
14         else if (p == q)
15             p = (t != (t = tail)) ? t : head;
16         else
17             p = (p != t && t != (t = tail)) ? t : q;
18     }
19 }
```

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：

```
1  UPDATE student        SET            name = '小王',          version = 2
    WHERE  id = 10              AND version = 1
```

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：

```
1  public final int getAndAdd(int delta) {
2      return unsafe.getAndAddInt(this, valueOffset, delta);
3  }
```

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：

```
1  public final int getAndAddInt(Object var1, long var2, int var4) {
2      int var5;
3      do {
4          var5 = this.getIntVolatile(var1, var2);
5      } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
6      return var5;
7  }
```

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。