链滴

Java 高并发与多线程（一）——CAS 操作与 Unsafe

作者：xiaoyao2102

原文链接：https://ld246.com/article/1581651528306

来源网站：链滴

1. CAS (Compare and Swap)

1.1 什么是 CAS 操作?

CAS 是**Compare And Swap**的缩写，有时也称作 Compare And Change，也叫**自旋**、**自旋锁**等，几个都是一回事。这是一种经常用来解决并发时数据一致性问题的操作。

CAS 操作包括三个操作数：当前值、预期当前值和修改后变量值。举个例子来说，对于一个变量 i，前值是 x，预计当前值为 0，现在想把 i 的值改为 1，那么三个操作数对应就为：x、0、1。

如果 i 的当前值就是 0 的话，那么将 i 的值修改为 1，操作成功。否则不做任何操作，返回操作失败（特别说明一下，为了不混淆，前面提到的 i 是变量名，而具体的值是 x）

下面的代码简单描述了这个过程。

```
public boolean cas(int i, int expect, int target) {
    if (i == expect) {
        i = target;
        return true;
    } else {
        return False;
    }
}
```

1.2 CAS 操作的常用模式

CAS 操作最常见的使用方式是放到循环中，包括死循环和计数循环。

比如下面这个例子，其中的 modifyAndGet 方法完成的功能是：**将 i 修改为目标值并且返回修改前值。**

```
public class TestCompareAndSwap {

    public int i = 0;

    public int modifyAndGet(int target) {
        int expect;
        do {
            expect = read(); // read方法可以读取内存中i的值
        } while (!cas(i, expect, target));

        return expect;
    }
}
```

上面这段代码就是 CAS 操作的常见使用方式，里面用到一个死循环，意思为**直到 CAS 操作成功执行才退出循环。**当然有时候也会使用计数循环，为的是避免CAS操作长期失败而导致当前整个功能无法续。

这种做法本质其实就是**乐观锁**，因此相比阻塞方式的悲观锁，这样的操作效率会提高很多。

1.3 CAS 操作中的 ABA 问题

在前面的操作过程描述中，我们可以发现，想要成功执行 CAS 操作，唯一的标准就是目标变量的当值和期望值相等。

因此，很有可能发生这样一种情况：

```
sequenceDiagram
线程1 ->> 目标变量i = 5: 读取expect = 5
线程2 ->> 目标变量i = 5: 修改i = 6
线程2 ->> 目标变量i = 5: 修改i = 5
线程1 ->> 目标变量i = 5: 对比expect == i
线程1 ->> 目标变量i = 5: 执行CAS操作
```

可以看出，虽然线程 1 成功执行了 CAS 操作，但事实上在这之中，i 的值发生过一次变化。这样的现就被称为**ABA 问题**。

所以基于上面的描述，解决 ABA 问题的办法也很简单明了。就是在变量 i 上再加一个版本号，用以述 i 当前被修改的状况，就可以知道 CAS 操作过程中 i 的值有没有发生过变化。

2. Unsafe

JDK 中有一个神秘莫测的类叫做**Unsafe**。从名字中就可以看出，使用这个类无异于与恶魔对话。如说 Java 中的反射操作已经是一种外挂的话，那 Unsafe 就是挂王之王。

本文中主要涉及 Unsafe 中与 CAS 相关的部分，其它内容或许将来会单开一篇讲讲吧。

2.1 简单吹一下 Unsafe

Java 与 C/C++ 的一个非常明显区别就是，Java 中不可以直接操作内存。当然这并不完全正确，因为Unsafe 就可以做到。

笔者数了一下，Unsafe 类中有 112 个 public 方法，其中 86 个 native 方法，剩余 34 个，也都是于这 86 个 native 方法的，相当于换个 API。所以虽说 Unsafe 可以直接操作内存，但它也只不过是用了 C/C++ 的代码。

2.2 Unsafe 在 AtomicInteger 中的应用

java.util.concurrent 包在 Java 并发编程中有着举足轻重的地位。JUC 包中的类大量使用了 Unsafe 方法，用底层的方式保证线程安全。我们这里只讲一下 AtomicInteger类中的getAndIncrement()方。

我们都知道，普通的 i++ 操作，是线程不安全的。如果同时有多个线程对一个变量进行 ++ 操作，么最终结果基本不会是正确的。但是如果我们把变量换成 AtomicInteger 类型并且使用 getAndIncrment()方法，就可以保证线程安全。这里面的原因，其实看一下 JDK 源码就一目了然。

首先是 AtomicInteger 对外的 API。

```
/**
 * Atomically increments by one the current value.
 *
 * @return the previous value
 */
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
```

```
}
```

显然这里是调用了 Unsafe 来直接对内存进行操作的。我们继续深入一层，进到 Unsafe 类中。

```
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;
}
```

这里源码因为是反编译出来的，所以变量的命名都是无意义的。不过我们可以看出，这里面就是 CAS 操作的典型应用，在 while 中不断进行 CAS 直到成功。

这里面用到两个方法：getIntVolatile 和 compareAndSwapInt。其中 getIntVolatile 跟 volatile 关字相关，我们会在接下来的文章中提到。另外一个 compareAndSwapInt，就是我们今天提到的 CAS 操作。

```
public native int getIntVolatile(Object var1, long var2);

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

想要知道它们做了什么，就必须深入到 C++ 的源码中。

笔者用**JDK8u**的源码为例，在**unsafe.cpp**文件中，我们可以找到这么一个方法

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject ob
, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

即使不了解C的语法知识，我们可以大概看出，这个函数的参数列表和Java中方法的参数列表是可以应的，C++中最后两个参数e和x就代表了expect和target的值，并且返回值是一个boolean类型，对了修改后的变量值是不是和期望值相等。这个方法实际上还是调用了**Atomic类**里面的**cmpxchg**函数我们再继续深入一下，到**atomic_linux_86.inline.hpp**中（当然不同的系统与CPU类型有不同的实，这里是用x86举例），我们可以找到cmpxchg函数

```
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```

这里面做了两件事，第一行执行is_MP()，MP是Multi Processor的缩写，意为获取当前环境是不是处理器环境，因为多处理器环境才有线程不安全的情况发生。第二行是执行一条汇编指令，LOCK_IF_P是一个宏定义，内容是

```
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

把指令字符串拼接到一起，其实核心就是一条：

```
lock cmpxchgl
```

至此，我们终于找到了AtomicInteger的getAndIncrement()线程安全的最根本原因。