谈一谈对CAS的理解

题目标签

学习时长:30分钟

题目难度: 中等

题目描述: 提问线路CAS—> Unsafe—> CAS底层原理 —> 原子引用更新 —> 如何规避ABA问题

1. 面试题分析

1. compareAndSet怎么用？
    a. 比较并交换（compareAndSet）

```
/**
 * boolean compareAndSet(int expect,
int update)
 * - 如果主内存的值=期待值expect，就将主内
存值改为update
 * - 该方法可以检测线程a的操作变量X没有被其
他线程修改过
 * - 保证了线程安全
 */
public static void main(String[] args)
{
    AtomicInteger atomicInteger = new
AtomicInteger(5);

 System.out.println(atomicInteger.comp
areAndSet(5, 10)+ "\t" +
atomicInteger);

 System.out.println(atomicInteger.comp
areAndSet(5, 20)+ "\t" +
atomicInteger);
    //true  10
    //false 10
}
```

2. CAS底层原理简述？

1. Compare-And-Swap。是一条CPU并发原语。（原语：操作系统范畴，依赖硬件，不被中断。）
2. 功能是判断内存某个位置的值是否为预期值（Compare），是就更新（Swap），这个过程是原子的。
3. 功能描述：
    a. 判断内存某个位置的值是否为预期值（Compare），是就更新（Swap），这个过程是原子的。
    b. cas有三个操作数，内存值V，旧预期值A，要更新的值B。仅当预期值A=内存值V时，才将内存值V修改为B，否则什么都不做。
4. 自旋：比较并交换，直到比较成功
5. 底层靠Unsafe类保证原子性。
6. getAndIncrement() 源码解析（用了cas保证线程安全）

```
/**
 * 参数说明：
 * this: atomicInteger对象
 * valueOffset：对象的内存地址
 * unsafe：sun.misc.Unsafe类
 * AtomicInteger中变量value使用volatile修饰，保证内存可见。
 * 结论：底层依赖CAS操作/Unsafe类
```

```
 */
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this,
valueOffset, 1);
}
/**
 * compareAndSwapInt：即CAS
 * while: 如果修改失败，会一直尝试修改，直到
成功。
 */
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;
}
```

- 简述：
    i. 调用了Unsafe类的getAndAddInt
    ii. getAndAddInt使用cas一直循环尝试修改主内存

3. 对Unsafe的理解?

- Unsave类
    a. 该类所有方法都是native修饰，直接调用底层资源。sun.misc包中。
    b. 可以像C的指针一样直接操作内存。java的CAS操作依赖Unsafe类的方法。

4. CAS有哪些缺点？

1. 循环时间长，开销大
    a. 如果cas失败，就一直do while尝试。如果长时间不成功，可能给CPU带来很大开销。
2. 只能保证一个共享变量的原子操作
    a. 如果时多个共享变量，cas无法保证原子性，只能加锁，锁住代码段。
3. 存在ABA问题。

5. 拓展内容

1. ABA问题
2. 简述ABA问题和解决方案？
    a. ABA问题描述: 线程1做CAS操作将A改为B再改为A，而线程2再做CAS时修改成功了，这不符合设计思想。

怎么解决：AtomicStampReference时间戳原子引用

3. ABA问题描述？问题出在哪？

    a. ABA问题描述：

        - 比如线程1从内存位置V中取出A，此时线程2也取出A。且线程2做了一次cas将值改为了B，然后又做了一次cas将值改回了A。此时线程1做cas发现内存中还是A，则线程1操作成功。这个时候实际上A值已经被其他线程改变过，这与设计思想是不符合的。

    这个过程问题出在哪？

        - 如果只在乎结果，ABA不介意B的存在，没什么问题
        - 如果B的存在会造成影响，需要通过AtomicStampReference，加时间戳解决。

4. 原子更新引用是啥？

    a. AtomicStampReference，使用时间戳，解决cas中出现的ABA问题。

AtomicReference使用代码演示

demo

```
/**
 * 如果希望原子操作的变量是
User,Book,此时需要使用
AtomicReference类
 */
public static void main(String[]
args) {
    User z3 = new User("z3",18);
    User l4 = new User("l4",19);
    AtomicReference<User>
atomicReference = new
AtomicReference<>(z3);

 System.out.println(atomicRefere
nce.compareAndSet(z3, l4) + "\t"
+
atomicReference.get().toString()
);

 System.out.println(atomicRefere
nce.compareAndSet(z3, l4) + "\t"
+
atomicReference.get().toString()
);

 //truecom.mxx.juc.User@4554617c
    //false
com.mxx.juc.User@4554617c
```

```
}
```

AtomicReference存在ABA问题代码验证

demo

```
AtomicReference atomicReference
= new AtomicReference<Integer>
(100);
/**
 * ABA问题验证：
 * 1--ABA
 * 2--A,C
 * @param args
 */
public static void main(String[]
args) {
    ABADemo abaDemo = new
ABADemo();

    new Thread(()->{

 abaDemo.atomicReference.compare
AndSet(100,101);
```

```
 abaDemo.atomicReference.compare
AndSet(101,100);
    },"1").start();

    new Thread(()->{
        // 睡1s等线程1执行完ABA
        try
{TimeUnit.SECONDS.sleep(1);}
catch (InterruptedException e) {
e.printStackTrace();}

 System.out.println(abaDemo.atom
icReference.compareAndSet(100,
2020)+"\t"+abaDemo.atomicReferen
ce.get());
        //true  2020

    },"2").start();
```

AtomicStampReference解决ABA问题代码验证

解决思路：每次变量更新的时候，把变量的版本号加一，这样只要变量被某一个线程修改过，该变量版本号就会发生递增操作，从而解决了ABA变化

```
AtomicStampedReference
atomicStampedReference = new
AtomicStampedReference<Integer>
(100,1);

public static void main(String[]
args) {
    // ABAProblem();
    ABADemo abaDemo = new
ABADemo();

    new Thread(()->{
        // 等线程2读到初始版本号的值
        try
{TimeUnit.SECONDS.sleep(1);}
catch (InterruptedException e) {
e.printStackTrace();}
        System.out.println("线程1
在ABA前的版本
号: "+abaDemo.atomicStampedRefere
nce.getStamp());

 abaDemo.atomicStampedReference.
compareAndSet(100,101,abaDemo.at
omicStampedReference.getStamp(),
abaDemo.atomicStampedReference.g
etStamp()+1);
```

```
 abaDemo.atomicStampedReference.compareAndSet(101,100,abaDemo.atomicStampedReference.getStamp(),abaDemo.atomicStampedReference.getStamp()+1);
        System.out.println("线程1在ABA后的版本号："+abaDemo.atomicStampedReference.getStamp());
    },"1").start();

    new Thread(()->{
        // 存一下修改前的版本号
        int stamp = abaDemo.atomicStampedReference.getStamp();
        System.out.println("线程2在修改操作前的版本号："+stamp);
        // 睡1s等线程1执行完ABA
        try {TimeUnit.SECONDS.sleep(2);} catch (InterruptedException e) {e.printStackTrace();}
```

```
 System.out.println(abaDemo.atomicStampedReference.compareAndSet(100,2020,stamp,abaDemo.atomicStampedReference.getStamp()+1)+"\t" +
abaDemo.atomicStampedReference.getReference());
        //线程2在修改操作前的版本号：1
        //线程1在ABA前的版本号：1
        //线程1在ABA后的版本号：3
        //false 100

        },"2").start();
```