**java.util.concurrent.atomic**

该包是JDK1.5开始提供的，它提供了类的小工具，支持在单个变量上解除锁的线程安全编程。此包中的类可将 volatile 值、字段和数组元素的概念扩展到那些也提供原子条件更新操作的类，其形式如下：

boolean compareAndSet(expectedValue, updateValue);

**CAS思想**

我们看到了上面提到的一个在java并发中非常重要的一类算法 -- CAS: Compare And Set 比较并设置； 什么意思呢，我们以 boolean compareAndSet(expectedValue, updateValue);方法为例来解释CAS的思想， 内存中可见的值如果和期望值(expectedValue)一致， 则将内存中的值修改为新值(updateValue)，并且返回true； 否则返回false；*注意* ： 该操作是原子性的，意思是线程安全的。当多个线程同时访问某个对象时，如果其中一个线程通过CAS操作获得了访问权限，则其他线程只能在该线程处理完之后才能访问。 这类似于同步字 **synchronized** 但是效率更高因为并没有锁的机制，即使在JDK7 之后对其进行过优化。

**AtomicBoolean实例详解**

/\*\*

\*

\*/

package byron4j.dlzd.curr.atomic;

import java.time.LocalDate;

import java.time.LocalTime;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicBoolean;

public class AtomicDemo {

private static volatile AtomicBoolean canExecutingFlag = new AtomicBoolean(true);

/\*\*

\*

\* 业务逻辑处理:

\* <ol>

\* <li>Step 1</li>

\* <li>Step 2</li>

\* </ol>

\*/

public void executeBusiLogic(){

if( canExecutingFlag.compareAndSet(true, false) ){

try{

System.out.println(LocalDate.now() + " " + LocalTime.now() + "--" + Thread.currentThread().getName() + "--处理业务逻辑开始...");

Thread.sleep(5000);

System.out.println(LocalDate.now() + " " + LocalTime.now() + "--" + Thread.currentThread().getName() + "--处理业务逻辑完毕.");

}catch(Exception e){

System.out.println(LocalDate.now() + " " + LocalTime.now() + "--" + Thread.currentThread().getName() + "--处理业务逻辑失败!!!");

}finally{

canExecutingFlag.set(true);

}

}else{

System.out.println(LocalDate.now() + " " + LocalTime.now() + "--" + Thread.currentThread().getName() + "--已经存在处理中的业务，请稍后再试!");

}

}

public static void main(String[] args) {

ExecutorService es = Executors.newFixedThreadPool(10);

AtomicDemo demo = new AtomicDemo();

for(int i = 0; i < 10; i++){

es.execute(new Runnable() {

@Override

public void run() {

demo.executeBusiLogic();

}

});

}

es.shutdown();

}

}

运行结果如下：

2017-09-13 22:13:45.081--pool-1-thread-3--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.082--pool-1-thread-2--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.082--pool-1-thread-6--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.081--pool-1-thread-1--处理业务逻辑开始...

2017-09-13 22:13:45.081--pool-1-thread-10--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.081--pool-1-thread-9--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.082--pool-1-thread-4--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.081--pool-1-thread-7--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.082--pool-1-thread-5--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:45.083--pool-1-thread-8--已经存在处理中的业务，请稍后再试!

2017-09-13 22:13:50.082--pool-1-thread-1--处理业务逻辑完毕.

我们看到thread-1首先获得操作权限canExecutingFlag 值为true，CAS验证通过并且将canExecutingFlag 值置为false，所以其他线程均未获得进入资格，因为处理业务逻辑花了5秒钟，其他线程得到了"已经在处理中"的提示。 为了模拟耗时操作，我们在 executeBusiLogic 方法中通过sleep使执行线程睡眠。

在实际生产中，我们可以使用该方式来处理并发问题， 比如金融领域，请求支付单做资金放款时，为了避免在同一时间请求多次，就可以使用 CAS 来控制。

**CAS的缺陷--CAS的ABA问题**

问题描述：

因为CAS是基于内存共享机制实现的，比如在AtomicBoolean类中使用了关键字 **volatile** 修饰的属性： private volatile int value;

线程t1在共享变量中读到值为A

线程t1被抢占了，线程t2执行

线程t2把共享变量里的值从A改成了B，再改回到A，此时被线程t1抢占。

线程t1回来看到共享变量里的值没有被改变，于是继续执行。

虽然线程t1以为变量值没有改变，继续执行了，但是这个过程中(即A的值被t2改变期间)会引发一些潜在的问题。ABA问题最容易发生在lock free 的算法中的，CAS首当其冲，因为CAS判断的是指针的地址。如果这个地址被重用了呢，问题就很大了。（地址被重用是很经常发生的，一个内存分配后释放了，再分配，很有可能还是原来的地址）

*举一个例子：*

我们进机场过安检的时候，有一个人和你的背包是一样的(瑞士牌)，安检完后他把你的背包拿走了，你看下包一样的于是很淡定地登记去了，但是你的Mac Pro不见了。。

这就是ABA的问题。

作者：东陆之滇  
链接：https://www.jianshu.com/p/eabecdbc6bd9  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。