原子类

1. 什么是原子类，有什么作用?

原子类的作用和锁类似,是为了保证并发情况下线程安全。不过原子类相比于锁,有一定的优势:

1. **粒度更细:** 原子变量可以把竞争范围缩小到**变量级别**,这是我们可以获得的最细粒度的情况了,通常锁的粒度都要大于原子变量的粒度
2. **效率更高:** 通常,使用原子类的效率会比使用锁的效率更高,**除了高度竞争的情况**，为了解决高度竞争，JDK8引入了LongAdder和LongAccumulator，这方面后面会细说。

> atomic 包下的类基本上都是借助 Unsafe 类，通过 CAS 操作来封装实现的。Unsafe类和CAS会CAS章节详细讲解

6类原子类概览:

Atomic基本类型原子类	**AtomicInteger、AtomicLong、AtomicBoolean**
Atomic*Array数组类型原子类	**AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray**
Atomic*Reference引用类型原子类	**AtomicReference、AtomicStampedReference、AtomicMarkableReference**
Atomic*FieldUpdater升级类型原子类	**AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater**
Adder累加器	**LongAdder、DoubleAdder**
Accumulator累加器	**LongAccumulator、DoubleAccumulator**

2. Atomic*基本类型原子类，以AtomicInteger为例

AtomicInteger 类常用方法

```
// 取得当前值
public final int get();
// 设置当前值
public final void set(int newValue);
// 设置新值，并返回旧值
public final int getAndSet(int newValue);
// 如果当前值为expect，则设置为u
public final boolean compareAndSet(int expect, int u);
// 当前值加1，返回旧值
public final int getAndIncrement();
// 当前值减1，返回旧值
public final int getAndDecrement();
// 当前值增加delta，delata可以为负数，返回旧值
public final int getAndAdd(int delta);
// 当前值加1，返回新值
public final int incrementAndGet();
// 当前值减1，返回新值
```

```
18 public final int decrementAndGet();
19 // 当前值增加delta，返回新值
20 public final int addAndGet(int delta);
```

案例：银行存款——AtomicInteger使用方法

AtomicInteger源码分析

用Unsafe来实现底层操作

用volatile修饰value字段，保证可见性

Unsafe的getAndAddInt方法分析：自旋 + CAS（乐观锁）。在这个过程中，通过compareAndSwapInt比较并更新value值，如果更新失败，重新获取旧值，然后更新。

3. Atomic*Array数组类型原子类

数组里的元素，都可以保证原子性，AtomicIntegerArray相当于把AtomicInteger组合成一个数组，一共有3种，分别是AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

案例

4. Atomic*Reference引用类型原子类

AtomicReference

AtomicReference类的作用，和AtomicInteger并没有本质区别， AtomicInteger可以让一个整数保证原子性，而AtomicReference可以让一个对象保证原子性，当然，AtomicReference的功能明显比AtomicInteger强，因为一个对象里可以包含很多属性。用法和AtomicInteger类似。

代码案例

AtomicStampedReference——加上了时间戳，防止ABA问题

刚才说到了AtomicReference会带来ABA问题，而AtomicStampedReference的诞生，就是解决了这个问题

5. 把普通变量升级为原子类：用AtomicIntegerFieldUpdater升级原有变量

概述

对普通变量进行升级

使用场景

通常希望引用变量“normal”（即，不必总是通过原子类上的get或set方法引用它）但偶尔需要一个原子get-set操作

用法，代码演示

AtomicIntegerFieldUpdaterDemo类

```
1 package atomic;
2
3 import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;
4
```

```java
/**
 * 描述：     演示AtomicIntegerFieldUpdater的用法
 */
public class AtomicIntegerFieldUpdaterDemo implements Runnable{

    static Candidate tom;
    static Candidate peter;

    // 要传入升级类的泛型
    public static AtomicIntegerFieldUpdater<Candidate> scoreUpdater =
AtomicIntegerFieldUpdater
            .newUpdater(Candidate.class, "score");

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            peter.score++;
            // AtomicIntegerFieldUpdater中的方法需要传入升级的的对象
            scoreUpdater.getAndIncrement(tom);
        }
    }

    public static class Candidate {
        // 必须要有volatile修饰
        volatile int score;
    }

    public static void main(String[] args) throws InterruptedException {
        tom=new Candidate();
        peter=new Candidate();
        AtomicIntegerFieldUpdaterDemo r = new
AtomicIntegerFieldUpdaterDemo();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("普通变量："+peter.score);
        System.out.println("升级后的结果"+ tom.score);
    }
}
```

1. 在构建`AtomicIntegerFieldUpdater`对象时，在泛型中需要传入，要升级的类，在newUpdater(arg1，arg2)的方法中需要分别传入要升级的类的class对象、和要升级的属性
2. `AtomicIntegerFieldUpdater`只能对某一**对象**中的**属性**升级为原子属性，而无法对基本类型进行升级
3. 由于`AtomicIntegerFieldUpdater`是对某一对象中的属性进行升级，所以在调用`AtomicIntegerFieldUpdater`中的方法时，往往都需要传入要升级的对象obj

注意点

1. Updater 只能 修改 它可 见 范围内的变量。 因为 Updater 使用 反射得到 这个变量。 如果 变量不可见， 就会出错。 比如 **如果 score 申明 为 private**， 就是**不可行**的。
2. 为了确保变量被 正确的读取， 它 必须是**volatile** 类型 的。 如果我们原有代码中未申明 这个 类型， 那么简单地申明 一下就 行， 这不会引起什么问题。

3. 由于 CAS 操作会通过对象实例中的 偏移量 直接进行 赋值， 因此， 它**不支持 static字段**（Unsafe. objectFieldOffset() 不支持 静态 变量）。

6. Adder累加器

介绍

是**Java 8引入**的，相对是比较新的一个类。

高并发下LongAdder比AtomicLong效率高，不过**本质是空间换时间。**

Atomic*遇到的问题是，适合用在低并发场景，否则在**高并发下，由于CAS的冲突机会大，会导致经常自旋**，影响整体效率。而LongAdder引入了**分段锁**的概念，**1、当竞争不激烈的时候**，所有线程都是通过CAS对同一个变量（Base）进行修改，但是等到了**2、竞争激烈的时候**，LongAdder**把不同线程对应到不同的Cell**（内部的一个结构，有一个Cell数组）上进行修改，最后再**sum**，降低了冲突的概率，是多段锁的理念，提高了并发性。

演示AtomicLong的问题

这里演示多线程情况下AtomicLong的性能，有20个线程对同一个AtomicLong累加。

由于竞争很激烈，**每一次加法，都要flush和refresh**，导致很耗费资源。

```
package atomic;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicLong;

/**
 * 描述:       演示高并发场景下，LongAdder比AtomicLong性能好
 */
public class AtomicLongDemo {

    public static void main(String[] args) throws InterruptedException {
        // counter初始值为0
        AtomicLong counter = new AtomicLong(0);
        ExecutorService service = Executors.newFixedThreadPool(20);
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            service.submit(new Task(counter));
        }
        service.shutdown();
        while (!service.isTerminated()) {

        }
        long end = System.currentTimeMillis();
        System.out.println(counter.get());
        System.out.println("AtomicLong耗时: " + (end - start));
    }

    private static class Task implements Runnable {

        private AtomicLong counter;

        public Task(AtomicLong counter) {
            this.counter = counter;
        }
```

```
36
37          @Override
38          public void run() {
39              for (int i = 0; i < 10000; i++) {
40                  counter.incrementAndGet();
41              }
42          }
43      }
44  }
```

运行结果：可以发现耗时1.677s

```
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\lib\idea_rt.jar=11098:C:\Program
 Files\JetBrains\IntelliJ IDEA 2019.3.3\bin" -Dfile.encoding=UTF-8 -classpath
 E:\myproject\concurrency_tools_practice\out\production\concurrency_tools_practice;E:\myproject\concurrency_tools_practice\lib\junit-4.12.jar;
 E:\myproject\concurrency_tools_practice\lib\hamcrest-core-1.3.jar atomic.AtomicLongDemo
100000000
AtomicLong耗时：1677
```

使用LongAdder进行改进

```
1   package atomic;
2
3   import java.util.concurrent.ExecutorService;
4   import java.util.concurrent.Executors;
5   import java.util.concurrent.atomic.AtomicLong;
6   import java.util.concurrent.atomic.LongAdder;
7
8   /**
9    * 描述：       演示高并发场景下，LongAdder比AtomicLong性能好
10   */
11  public class LongAdderDemo {
12
13      public static void main(String[] args) throws InterruptedException {
14          LongAdder counter = new LongAdder();
15          ExecutorService service = Executors.newFixedThreadPool(20);
16          long start = System.currentTimeMillis();
17          for (int i = 0; i < 10000; i++) {
18              service.submit(new Task(counter));
19          }
20          service.shutdown();
21          while (!service.isTerminated()) {
22
23          }
24          long end = System.currentTimeMillis();
25          System.out.println(counter.sum());
26          System.out.println("LongAdder耗时：" + (end - start));
27      }
28
29      private static class Task implements Runnable {
30
31          private LongAdder counter;
32
33          public Task(LongAdder counter) {
34              this.counter = counter;
35          }
36
37          @Override
38          public void run() {
39              for (int i = 0; i < 10000; i++) {
```

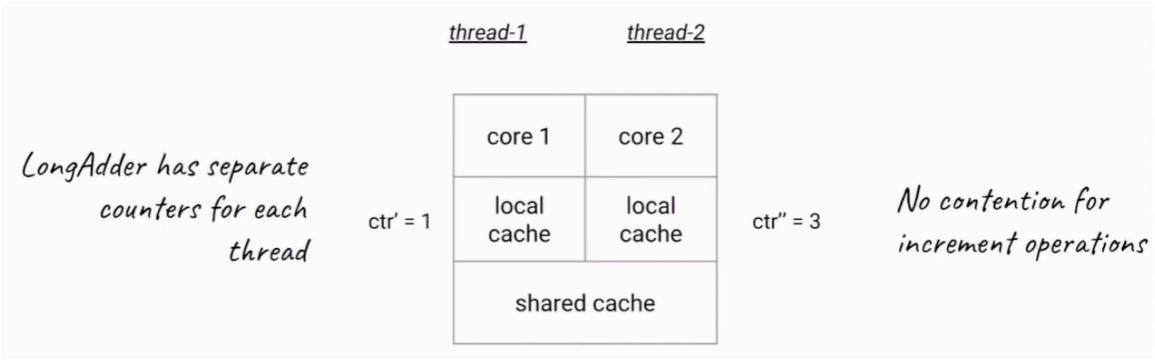
```
                counter.increment();
            }
        }
    }
}
```

运行结果：可以发现耗时明显缩短为361ms,速度提高了5倍

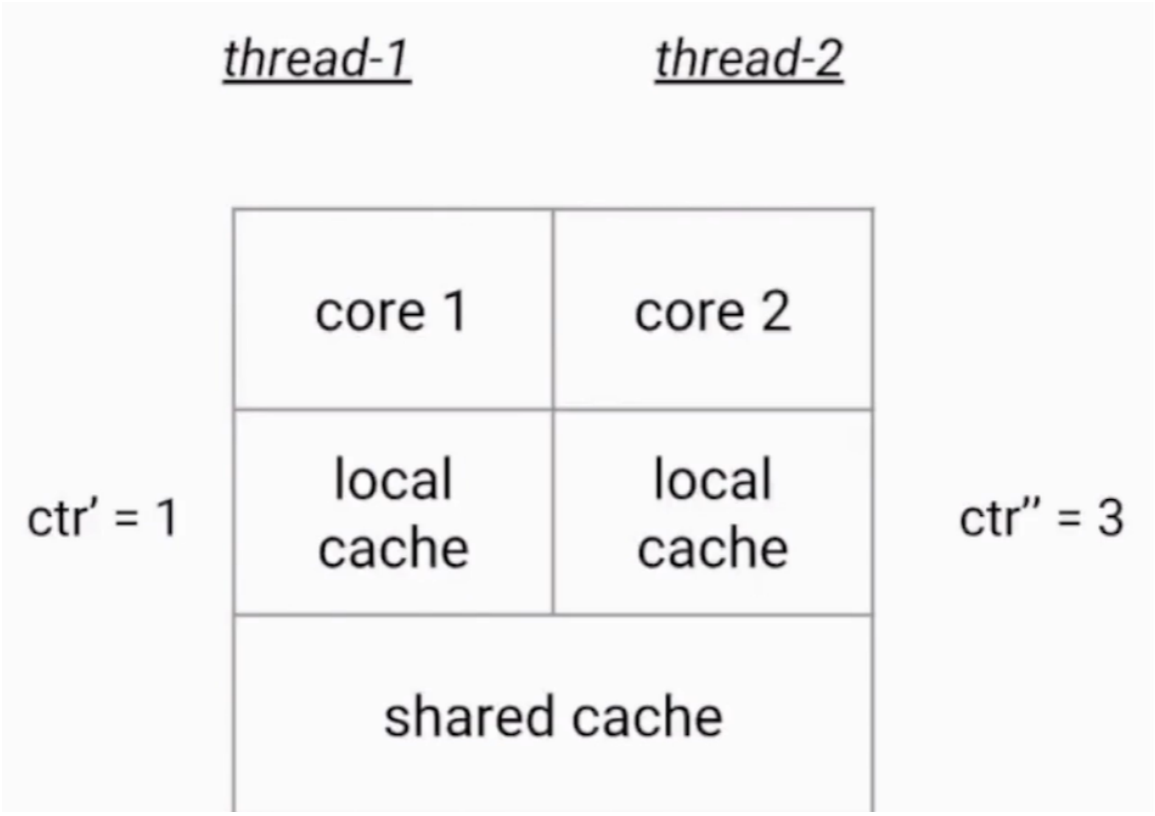
```
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\lib\idea_rt.jar=11085:C:\Program
 Files\JetBrains\IntelliJ IDEA 2019.3.3\bin" -Dfile.encoding=UTF-8 -classpath
 E:\myproject\concurrency_tools_practice\out\production\concurrency_tools_practice;E:\myproject\concurrency_tools_practice\lib\junit-4.12.jar;
 E:\myproject\concurrency_tools_practice\lib\hamcrest-core-1.3.jar atomic.LongAdderDemo
100000000
LongAdder耗时：361

Process finished with exit code 0
```

LongAdder带来的改进和原理

在内部，这个LongAdder的实现原理和刚才的AtomicLong是有不同的，刚才的AtomicLong的实现原理是，每一次加法都需要做**同步（每一次加法，都要flush和refresh**，每个线程都要把加的结构刷好主内存，并同步到其他线程），所以在高并发的时候会导致冲突比较多，也就降低了效率。



而此时的LongAdder，**每个线程会有自己的一个计数器**，仅用在自己线程内计数，这样就不会和其他线程的计数器干扰。



如上图所示，第一个线程的计数器数值，也就是ctr’ 为1的时候，可能线程2的计数器ctr”的数值已经是3了，他们之间并**不存在竞争关系**,所以在加和的过程中，根本不需要同步机制，也不需要刚才的flush和refresh。这里也没有一个公共的counter来给所有线程统一计数。

每个线程都各加各的，那么LongAdder最终是如何实现多线程计数的呢？答案就在最后一步，执行LongAdder.sum()的时候，这里是**唯一需要同步**的地方：

当我们执行sum函数的时候，LongAdder会把所有线程的计数器，也就是ctr’和ctr”等等都在同步的情况下加起来，形成最终的总和。

AtomicLong在多线程的情况下，**每个线程的每次累加**都要同步，而**LongAdder仅在最后sum的时候需要同步**，其他情况下，**多个线程可以同时运行**，这就是LongAdder的吞吐量比AtomicLong大的原因，**本质是空间换时间**

sum()源码

```
public long sum() {
    Cell[] cs = cells;
    long sum = base;
    // 判断Cell[]是否为空，如果为空，说明没有用的cs数组，也就是竞争不激烈的时候；如果Cell[]数组不为空，则遍历Cell[]数组，与sum进行累加。最后返回sum
    if (cs != null) {
        for (Cell c : cs)
            if (c != null)
                sum += c.value;
    }
    return sum;
}
```

值得注意的时，再sum方法中是没有加锁的，但是每一个Cell数组中的值在任何时候都有可能会被修改的，这就会导致在累加的时候，可能前面累加过的cs.value已经发生了变化，会导致sum没有十分的精确。

AtomicLong和LongAdder对比

- 在低争用下, Atomiclong和 Longadderi这两个类具有相似的特征。但是在竞争激烈的情况下, Longadderg的预期吞吐量要高得多,但要消耗更多的空间
- Longadderi适合的场景是统计求和计数的场景,而且Longadder基本只提供了add方法,而Atomiclong还具有cas方法

7. Accumulator累加器

Accumulator和Adder非常相似，Accumulator就是一个更通用版本的Adder

用法

LongAccumulator的构造函数的第一个参数是一个表达式，第二个参数是x的初始值。**x是每次的初始值，y是结果。**执行counter.accumulate(1)的时候，第一次x是0，y是1，**后面每次计算的y的结果会赋值给x**，然后每次的新y就是counter.accumulate(1)传入的1。

代码演示：

```
package atomic;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.LongAccumulator;
```

```java
import java.util.stream.IntStream;

/**
 * 描述：     演示LongAccumulator的用法
 */
public class LongAccumulatorDemo {

    public static void main(String[] args) {
        // 设置x的初始值为1
        LongAccumulator accumulator = new LongAccumulator((x, y) -> 2 + x * y, 1);
        ExecutorService executor = Executors.newFixedThreadPool(8);
        IntStream.range(1, 10).forEach(i -> executor.submit(() -> accumulator.accumulate(i)));

        executor.shutdown();
        // 等待线程池执行完毕
        while (!executor.isTerminated()) {

        }
        System.out.println(accumulator.getThenReset());
    }
}
```

`LongAccumulator`相比于`LongAdder`可以提供累加器初始非0值，后者只能默认为0，另外前者还可以指定累加规则比如不是累加而是相乘，只需要构造`LongAccumulator`时候传入自定义双目运算器就OK，后者则内置累加的规则。

适用场景

- 当多个线程更新用于诸如收集统计信息之类的目的，而不是用于细粒度的同步控制的公共值时，此类通常比AtomiLong更可取。
- 不能保证并且不能依赖于线程内或者线程间的叠加顺序，因此此类仅适用于叠加顺序无关紧要的函数。