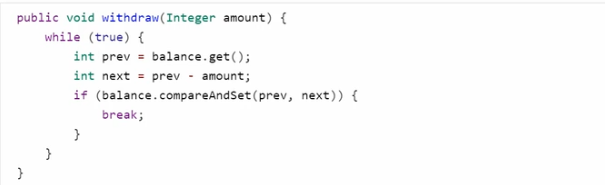
CAS

前面看到的 AtomicInteger 的解决方法，内部并没有用锁来保护共享变量的线程安全。那么它是如何实现的呢？

使用do...while

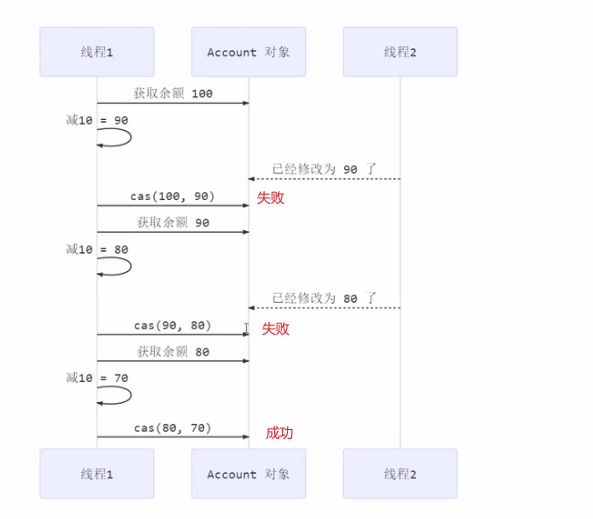
*/\*\*  
 \* 存在线程安全问题：  
 \* （1）使用synchronized解决  
 \* （2）使用ReentrantLock解决、  
 \* （3）使用CAS+volatile解决(无锁)  
 \*/***public void** cost(**int** x) {  
 **int** t;  
 **do** {  
 t = **balance**.get(); *//快照* } **while** (!**balance**.compareAndSet(t, t - x)); *//CAS*}

使用while



其中的关键是 compareAndSet，它的简称就是 CAS （也有 Compare And Swap 的说法），它必须是原子操作。

是一种CPU指令级的原子性。



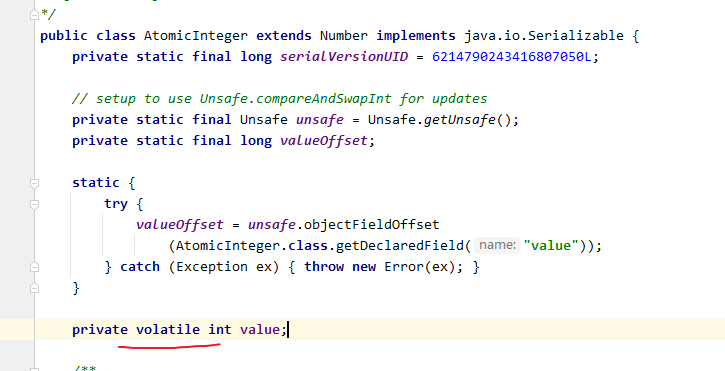
其实 CAS 的底层是 lock cmpxchg 指令（X86 架构），在单核 CPU 和多核 CPU 下都能够保证【比较-交换】的原子性。

在多核状态下，某个核执行到带 lock 的指令时，CPU 会让总线锁住，当这个核把此指令执行完毕，再开启总线。这个过程中不会被线程的调度机制所打断，保证了多个线程对内存操作的准确性，是原子的。

volatile

CAS操作需要volatile支持。

源码上可以看出：



获取共享变量时，为了保证该变量的可见性，需要使用 volatile 修饰。

它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。即一个线程对 volatile 变量的修改，对另一个线程可见。

volatile 仅仅保证了共享变量的可见性，让其它线程能够看到最新值，但不能解决指令交错问题（不能保证原子性）

CAS 必须借助 volatile 才能读取到共享变量的最新值来实现【比较并交换】的效果。

为什么无锁效率高

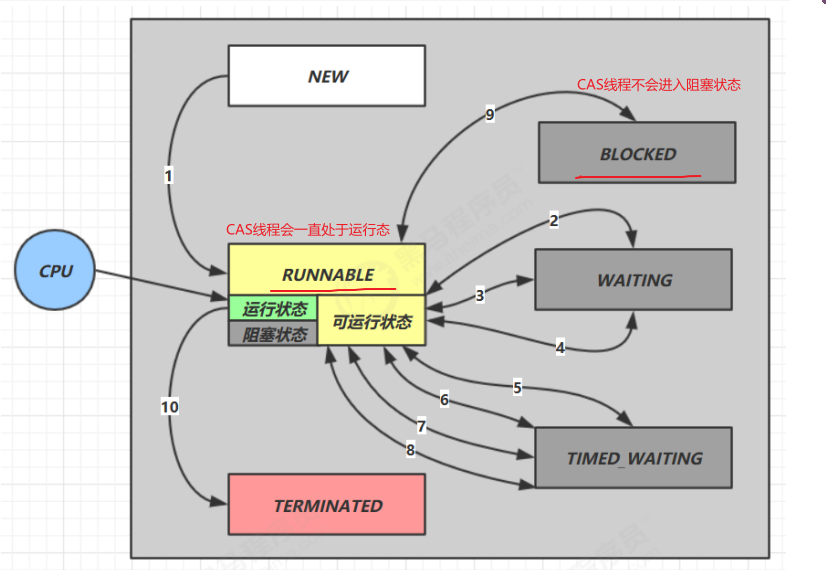
使用CAS的优点是吞吐量高。

缺点是需要频繁使用cpu，资源开销大。

无锁情况下，即使重试失败，线程始终在高速运行，没有停歇，而 synchronized 会让线程在没有获得锁的时候，发生上下文切换，进入阻塞。

打个比喻线程就好像高速跑道上的赛车，高速运行时，速度超快，一旦发生上下文切换，就好比赛车要减速、熄火，等被唤醒又得重新打火、启动、加速... 恢复到高速运行，代价比较大。

但无锁情况下，因为线程要保持运行，需要额外 CPU 的支持，CPU 在这里就好比高速跑道，没有额外的跑道，线程想高速运行也无从谈起，虽然不会进入阻塞，但由于没有分到时间片，仍然会进入可运行状态，还是会导致上下文切换。



CAS特点

结合 CAS 和 volatile 可以实现无锁并发，适用于线程数少、多核 CPU 的场景下。

当线程数大于CPU的最大核心数，CAS效率不会提升。

* CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
* synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

CAS 体现的是无锁并发（不会加锁，使用while循环）、无阻塞并发（CAS线程会一直运行，不会进入阻塞态发生上下文切换），请仔细体会这两句话的意思。

因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一。

但如果竞争激烈，可以想到重试必然频繁发生，反而效率会受影响。

原子整数

juc并发包提供了：

AtomicBoolean

AtomicInteger

AtomicLong

以 AtomicInteger 为例，

AtomicInteger i = new AtomicInteger(0);  
// 获取并自增（i = 0, 结果 i = 1, 返回 0），类似于 i++  
System.out.println(i.getAndIncrement());  
// 自增并获取（i = 1, 结果 i = 2, 返回 2），类似于 ++i  
System.out.println(i.incrementAndGet());  
// 自减并获取（i = 2, 结果 i = 1, 返回 1），类似于 --i  
System.out.println(i.decrementAndGet());  
// 获取并自减（i = 1, 结果 i = 0, 返回 1），类似于 i--  
System.out.println(i.getAndDecrement());  
// 获取并加值（i = 0, 结果 i = 5, 返回 0）  
System.out.println(i.getAndAdd(5));  
// 加值并获取（i = 5, 结果 i = 0, 返回 0）  
System.out.println(i.addAndGet(-5));  
// 获取并更新（i = 0, p 为 i 的当前值, 结果 i = -2, 返回 0）  
// 其中函数中的操作能保证原子，但函数需要无副作用  
System.out.println(i.getAndUpdate(p -> p - 2));  
// 更新并获取（i = -2, p 为 i 的当前值, 结果 i = 0, 返回 0）  
// 其中函数中的操作能保证原子，但函数需要无副作用  
System.out.println(i.updateAndGet(p -> p + 2));  
// 获取并计算（i = 0, p 为 i 的当前值, x 为参数1, 结果 i = 10, 返回 0）  
// 其中函数中的操作能保证原子，但函数需要无副作用  
// getAndUpdate 如果在 lambda 中引用了外部的局部变量，要保证该局部变量是 final 的  
// getAndAccumulate 可以通过 参数1 来引用外部的局部变量，但因为其不在 lambda 中因此不必是 final  
System.out.println(i.getAndAccumulate(10, (p, x) -> p + x));  
// 计算并获取（i = 10, p 为 i 的当前值, x 为参数1, 结果 i = 0, 返回 0）  
// 其中函数中的操作能保证原子，但函数需要无副作用  
System.out.println(i.accumulateAndGet(-10, (p, x) -> p + x));

原子引用

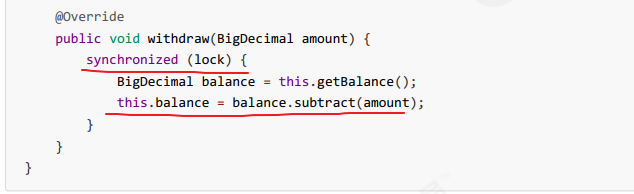
为什么需要原子引用类型？

* AtomicReference
* AtomicMarkableReference
* AtomicStampedReference

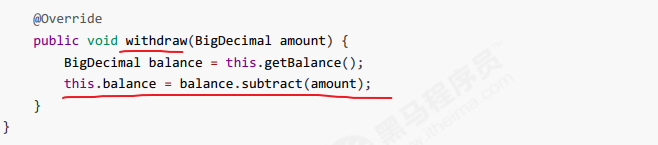
使用CAS安全实现：

**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.math.BigDecimal;  
**import** java.util.concurrent.atomic.AtomicReference;  
  
*/\*\*  
 \* 原子引用  
 \*/*@Slf4j(topic = **"c.TestAtomicReference"**)  
**public class** TestAtomicReference {  
  
 @Test  
 **public void** test\_AtomicReference\_BigDecimal() {  
 *//创建账户对象* DecimalAccountCAS account = **new** DecimalAccountCAS(**new** BigDecimal(**"1000000"**));  
 *//100个线程取款,每个线程取10000* **for** (**int** i = 0; i < 100; i++) {  
 **new** Thread(() -> {  
 account.withdraw(**new** BigDecimal(**"10000"**));  
 }).start();  
 }  
 *//主线程睡等待取款操作完成* **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//输出账户余额* ***log***.debug(**"账户余额：{}"**, account.getAccount());  
  
 }  
}  
  
**interface** DecimalAccount {  
 *//获取余额* BigDecimal getAccount();  
  
 *//取款* **void** withdraw(BigDecimal balance);  
}  
  
**class** DecimalAccountCAS **implements** DecimalAccount {  
  
 *//原子引用变量* **private** AtomicReference<BigDecimal> **balance**;  
  
 **public** DecimalAccountCAS(BigDecimal balance) {  
 **this**.**balance** = **new** AtomicReference<>(balance);  
 }  
  
 @Override  
 **public** BigDecimal getAccount() {  
 **return balance**.get();  
 }  
  
 @Override  
 **public void** withdraw(BigDecimal balance) {  
 *//CAS实现取款* BigDecimal prev, next;  
 **do** {  
 *//获取当前值* prev = **this**.**balance**.get();  
 *//修改值* next = prev.subtract(balance);  
 } **while** (!**this**.**balance**.compareAndSet(prev, next));  
 }  
}

使用synchronized锁安全实现（效率低）：



不安全的实现：



ABA问题

AtomicInteger不能解决ABA问题

主线程仅能判断出共享变量的值与最初值 A 是否相同，不能感知到这种从 A 改为 B 又 改回 A 的情况。

*/\*\*  
 \* ABA问题复现  
 \*/*@Test  
**public void** test\_ABAQuestion() {  
 AtomicInteger ai = **new** AtomicInteger(10);  
 *//线程1睡眠1秒后修改ai变量，10->20->10* Thread t1 = **new** Thread(() -> {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ai.compareAndSet(10, 20);  
 ai.compareAndSet(20, 10);  
 }, **"t1"**);  
 t1.start();  
  
 *//线程2先获取ai的值，3秒后再获取ai的值* Thread t2 = **new** Thread(() -> {  
 **int** prev = ai.get();  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//虽然t1线程修改过ai的值，但是结果仍是true* System.***out***.println(ai.compareAndSet(prev, 20)); *//true* }, **"t2"**);  
 t2.start();  
  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

AtomicStampedReference

只要有其它线程【动过了】共享变量，那么自己的 cas 就算失败，这时，仅比较值是不够的，需要再加一个版本号AtomicStampedReference。

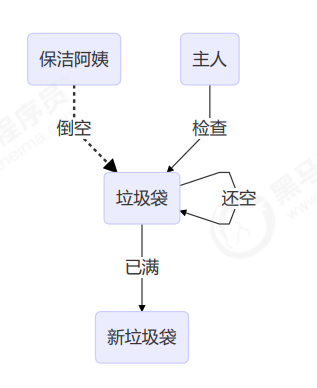
AtomicStampedReference 可以给原子引用加上版本号，追踪原子引用整个的变化过程，如： A -> B -> A ->C ，通过AtomicStampedReference，我们可以知道，引用变量中途被更改了几次。

*/\*\*  
 \* ABA问题解决  
 \* 使用 AtomicStampedReference ，带有版本号（时间戳）的原子引用  
 \*/*@Test  
**public void** test\_ABA\_AtomicStampedReference() {  
 AtomicStampedReference<Integer> ai =  
 **new** AtomicStampedReference<>(10, 1);  
 Thread t1 = **new** Thread(() -> {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 */\*\*  
 \** ***@param expectedReference*** *the expected value of the reference 期望值  
 \** ***@param newReference*** *the new value for the reference 新值  
 \** ***@param expectedStamp*** *the expected value of the stamp 期望时间戳  
 \** ***@param newStamp*** *the new value for the stamp 新时间戳  
 \*/  
 //获取当前时间戳* ai.compareAndSet(10, 20, ai.getStamp(), ai.getStamp() + 1);  
 ai.compareAndSet(20, 10, ai.getStamp(), ai.getStamp() + 1);  
 }, **"t1"**);  
 t1.start();  
  
 Thread t2 = **new** Thread(() -> {  
 *//获取时间戳* **int** stamp = ai.getStamp();  
 *//获取快照* **int** prev = ai.getReference();  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(ai.compareAndSet(prev, 20, stamp, stamp + 1)); *//false* }, **"t2"**);  
 t2.start();  
  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

AtomicMarkableReference

但是有时候，并不关心引用变量更改了几次，只是单纯的关心是否更改过，所以就有了

AtomicMarkableReference。



@Test  
**public void** test\_ABA\_AtomicMarkableReference() {  
 *//* AtomicMarkableReference<Integer> amr =  
 **new** AtomicMarkableReference<>(10, **false**);  
 *//线程1* Thread t1 = **new** Thread(() -> {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 */\*  
 \* @param expectedReference the expected value of the reference 期望值  
 \* @param newReference the new value for the reference 新值  
 \* @param expectedMark the expected value of the mark 期望标记  
 \* @param newMark the new value for the mark 新标记  
 \*/* amr.compareAndSet(10, 20, **false**, **true**);  
 }, **"t1"**);  
 t1.start();  
 *//线程2* Thread t2 = **new** Thread(() -> {  
 Integer prev = amr.getReference(); *//获取快照* **boolean** mark = amr.isMarked(); *//获取是否改变* **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//false* System.***out***.println(amr.compareAndSet(20, 10, **false**, **true**));  
 }, **"t2"**);  
 t2.start();  
  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

原子数组

AtomicIntegerArray

AtomicLongArray

AtomicReferenceArray

函数式接口说明

supplier 提供者 无中生有 ()->结果

function 函数 一个参数一个结果 (参数)->结果

BiFunction 函数 两个参数一个结果 （参数1，参数2）->结果

consumer 消费者 一个参数没有结果 (参数)->void

BiConsumer 消费者 两个参数没有结果 （参数1，参数2）->void

普通原子数组不能保证线程安全，原子数组可以保证线程安全：

**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
**import** java.util.concurrent.atomic.AtomicIntegerArray;  
**import** java.util.function.BiConsumer;  
**import** java.util.function.Consumer;  
**import** java.util.function.Function;  
**import** java.util.function.Supplier;  
  
@Slf4j(topic = **"c.TestAtomicArray"**)  
**public class** TestAtomicArray {  
  
 */\*\*  
 \* 参数1，提供数组、可以是线程不安全数组或线程安全数组  
 \* 参数2，获取数组长度的方法  
 \* 参数3，自增方法，回传 array, index  
 \* 参数4，打印数组的方法  
 \* <p>  
 \* 函数式接口说明  
 \* supplier 提供者 无中生有 ()->结果  
 \* function 函数 一个参数一个结果 (参数)->结果  
 \* BiFunction 函数 两个参数一个结果 （参数1，参数2）->结果  
 \* consumer 消费者 一个参数没有结果 (参数)->void  
 \* BiConsumer 消费者 两个参数没有结果 （参数1，参数2）->void  
 \*  
 \** ***@FunctionalInterface*** *public interface Supplier<T> {  
 \* T get();  
 \* }  
 \** ***@FunctionalInterface*** *public interface Function<T, R> {  
 \* R apply(T t);  
 \* }  
 \** ***@FunctionalInterface*** *public interface BiConsumer<T, U> {  
 \* void accept(T t, U u);  
 \* }  
 \** ***@FunctionalInterface*** *public interface Consumer<T> {  
 \* void accept(T t);  
 \* }  
 \*/* **private static** <T> **void** demo(  
 Supplier<T> arraySupplier,  
 Function<T, Integer> lengthFun,  
 BiConsumer<T, Integer> putConsumer,  
 Consumer<T> printConsumer  
 ) {  
 List<Thread> ts = **new** ArrayList<>();  
 *//创建数组* T array = arraySupplier.get();  
 *//获取数组长度* **int** length = lengthFun.apply(array);  
 **for** (**int** i = 0; i < length; i++) {  
 *//每个线程对数组做1000次操作* ts.add(  
 **new** Thread(() -> {  
 **for** (**int** j = 0; j < 10000; j++) {  
 putConsumer.accept(array, j % length);  
 }  
 })  
 );  
 }  
 *//启动所有线程* ts.forEach(t -> t.start());  
 *//等待所有线程执行结束* ts.forEach(t -> {  
 **try** {  
 t.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 *//输出数组* printConsumer.accept(array);  
 }  
  
 */\*\*  
 \* 多线程下，一个普通数组是没有线程安全性的  
 \*/* @Test  
 **public void** test\_array() {  
 *demo*(  
 () -> **new int**[10],  
 array -> array.**length**,  
 (array, index) -> {  
 array[index]++;  
 },  
 (array) -> {  
 System.***out***.println(Arrays.*toString*(array));  
 }  
 );  
 *//[9682, 9658, 9662, 9652, 9660, 9667, 9682, 9672, 9671, 9667]* }  
  
 */\*\*  
 \* 原子数组 AtomicIntegerArray  
 \*/* @Test  
 **public void** test\_AtomicIntegerArray() {  
 *demo*(  
 () -> **new** AtomicIntegerArray(10),  
 atomicArray -> atomicArray.length(),  
 (atomicIntegerArray, index) -> {  
 atomicIntegerArray.getAndIncrement(index);  
 },  
 atomicIntegerArray -> {  
 System.***out***.println(atomicIntegerArray.toString());  
 }  
 );  
 *//[10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000]* }  
}

原子更新器

AtomicReferenceFieldUpdater // 域 字段

AtomicIntegerFieldUpdater

AtomicLongFieldUpdater

**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.concurrent.atomic.AtomicReferenceFieldUpdater;  
  
@Slf4j(topic = **"c.TestAtomicReferenceFieldUpdater"**)  
**public class** TestAtomicReferenceFieldUpdater {  
  
 @Test  
 **public void** test\_AtomicReferenceFieldUpdater() {  
 *//创建原子更新器对象* AtomicReferenceFieldUpdater updater =  
 AtomicReferenceFieldUpdater.*newUpdater*(Student.**class**, String.**class**, **"name"**);  
 Student stu = **new** Student(**null**);  
 System.***out***.println(updater.compareAndSet(stu, **null**, **"张三"**)); *//true* }  
}  
  
**class** Student {  
 **volatile** String **name**;  
  
 **public** Student(String name) {  
 **this**.**name** = name;  
 }  
}

原子累加器

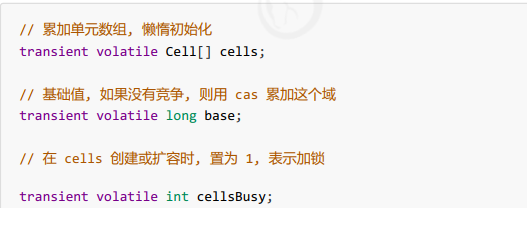
1、LongAdder比AtomicLong性能好的原因

性能提升的原因很简单，就是在有竞争时，设置多个累加单元，Therad-0 累加 Cell[0]，而 Thread-1 累加Cell[1]... 最后将结果汇总。这样它们在累加时操作的不同的 Cell 变量，因此减少了 CAS 重试失败，从而提高性能。

2、AtomicLong与LongAdder的比较代码

**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.atomic.AtomicLong;  
**import** java.util.concurrent.atomic.LongAdder;  
**import** java.util.function.Consumer;  
**import** java.util.function.Supplier;  
  
@Slf4j(topic = **"c.Test\_AtomicLong\_LongAdder"**)  
**public class** Test\_AtomicLong\_LongAdder {  
  
 */\*\*  
 \** ***@param supplier*** *()->(结果)  
 \** ***@param consumer*** *(参数)->()  
 \** ***@param <T>*** *\*/* **public static** <T> **void** demo(Supplier<T> supplier, Consumer<T> consumer) {  
 T adder = supplier.get(); *//累加器，初始值=0* List<Thread> threadList = **new** ArrayList<>();  
 *//添加计时器* **long** start = System.*currentTimeMillis*();  
 *//创建4个线程，每个线程累加500000次* **for** (**int** i = 0; i < 4; i++) {  
 threadList.add(**new** Thread(() -> {  
 **for** (**int** j = 0; j < 500000; j++) {  
 consumer.accept(adder);  
 }  
 }));  
 }  
 *//启动每个线程* threadList.forEach(t -> {  
 t.start();  
 });  
 *//等待每个线程执行结束* threadList.forEach(t -> {  
 **try** {  
 t.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 *//查看累加器* ***log***.debug(adder.toString() + **",耗时："** + (System.*currentTimeMillis*() - start));  
 }  
  
 *//测试AtomicLong累加效率* @Test  
 **public void** test\_AtomicLong() {  
 *//10:27:11.592 [main] DEBUG c.Test\_AtomicLong\_LongAdder - 2000000,耗时：33  
 demo*(  
 () -> **new** AtomicLong(0),  
 (addr) -> {  
 addr.getAndIncrement();  
 }  
 );  
 }  
  
 *//测试LongAdder累加效率  
 //LongAdder比AtomicLong累加效率高* @Test  
 **public void** test\_LongAddr() {  
 *//10:28:00.303 [main] DEBUG c.Test\_AtomicLong\_LongAdder - 2000000,耗时：25  
 demo*(  
 () -> **new** LongAdder(),  
 (addr) -> {  
 addr.increment();  
 }  
 );  
 }  
}

3、LongAdder源码解析



*/\*\*  
 \* Table of cells. When non-null, size is a power of 2.  
 \*/***transient volatile** Cell[] **cells**;  
  
*/\*\*  
 \* Base value, used mainly when there is no contention, but also as  
 \* a fallback during table initialization races. Updated via CAS.  
 \*/***transient volatile long base**;  
  
*/\*\*  
 \* Spinlock (locked via CAS) used when resizing and/or creating Cells.  
 \*/***transient volatile int cellsBusy**;

4、Cas锁

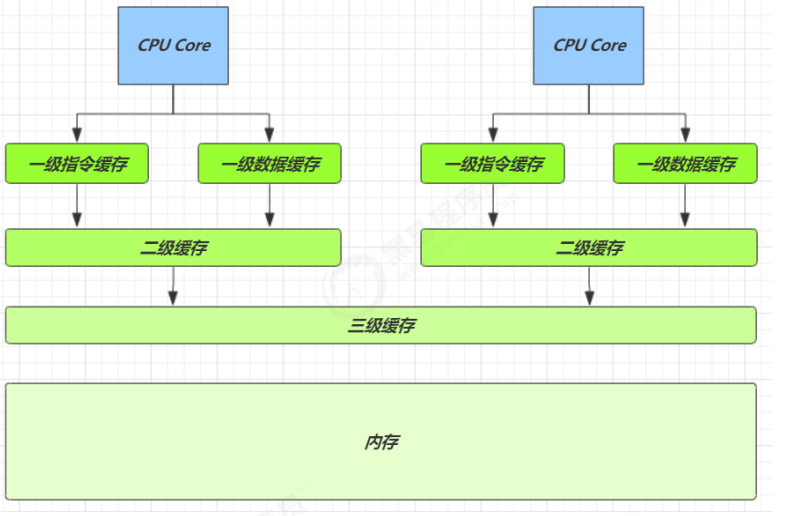
**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.concurrent.atomic.AtomicInteger;  
  
@Slf4j(topic = **"c.Test\_CasLock"**)  
**public class** Test\_CasLock {  
  
 @Test  
 **public void** testCasLock() {  
 ***log***.debug(**"测试CAS锁"**);  
 CasLock lock = **new** CasLock();  
 Thread t1 = **new** Thread(() -> {  
 lock.lock();  
 ***log***.debug(**"模拟操作"**);  
 lock.unlock();  
 }, **"t1"**);  
 t1.start();  
 Thread t2 = **new** Thread(() -> {  
 lock.lock();  
 ***log***.debug(**"模拟操作"**);  
 lock.unlock();  
 }, **"t2"**);  
 t2.start();  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 */\*\*  
 \* 10:46:07.914 [main] DEBUG c.Test\_CasLock - 测试CAS锁  
 \* 10:46:07.976 [t1] DEBUG c.CasLock - lock...  
 \* 10:46:07.977 [t2] DEBUG c.CasLock - lock...  
 \* 10:46:07.977 [t1] DEBUG c.Test\_CasLock - 模拟操作  
 \* 10:46:07.979 [t1] DEBUG c.CasLock - unlock...  
 \* 10:46:07.979 [t2] DEBUG c.Test\_CasLock - 模拟操作  
 \* 10:46:07.979 [t2] DEBUG c.CasLock - unlock...  
 \*/* }  
}  
  
@Slf4j(topic = **"c.CasLock"**)  
**class** CasLock {  
 *//定义一个标志，记录上锁的状态  
 //0：没加锁  
 //1：加锁* **private** AtomicInteger **status** = **new** AtomicInteger(0);  
  
 *//cas方式实现加锁* **public void** lock() {  
 ***log***.debug(**"lock..."**);  
 **do** {  
  
 } **while** (!**status**.compareAndSet(0, 1));  
 }  
  
 *//解锁* **public void** unlock() {  
 ***log***.debug(**"unlock..."**);  
 **status**.set(0);  
 }  
  
}

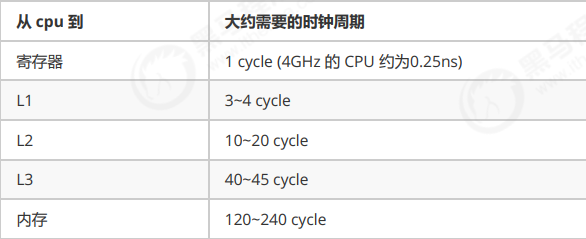
LongAdder类的cellBusy，在扩充cell时会用到Cas锁，用于当一个线程扩容时，其他线程等待。

5、Cell累加单元

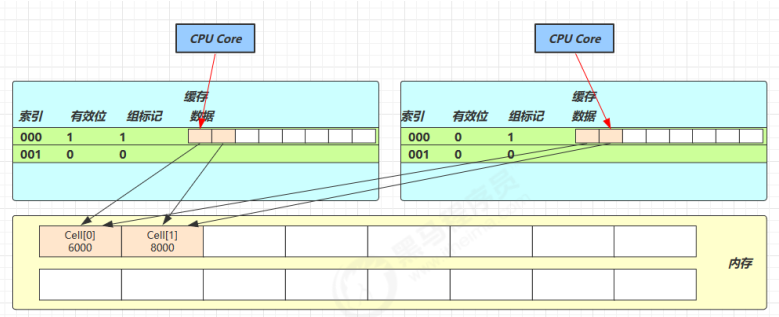
@sun.misc.Contended **static final class** Cell {  
 **volatile long value**;  
 Cell(**long** x) { **value** = x; }  
 **final boolean** cas(**long** cmp, **long** val) {  
 **return *UNSAFE***.compareAndSwapLong(**this**, ***valueOffset***, cmp, val);  
 }  
  
 *// Unsafe mechanics* **private static final** sun.misc.Unsafe ***UNSAFE***;  
 **private static final long *valueOffset***;  
 **static** {  
 **try** {  
 ***UNSAFE*** = sun.misc.Unsafe.*getUnsafe*();  
 Class<?> ak = Cell.**class**;  
 ***valueOffset*** = ***UNSAFE***.objectFieldOffset  
 (ak.getDeclaredField(**"value"**));  
 } **catch** (Exception e) {  
 **throw new** Error(e);  
 }  
 }  
}

6、@sun.misc.Contended 防止缓存行伪共享





因为 CPU 与 内存的速度差异很大，需要靠预读数据至缓存来提升效率。而缓存以缓存行为单位，每个缓存行对应着一块内存，一般是 64 byte（8 个 long）缓存的加入会造成数据副本的产生，即同一份数据会缓存在不同核心的缓存行中CPU 要保证数据的一致性，如果某个 CPU 核心更改了数据，其它 CPU 核心对应的整个缓存行必须失效。



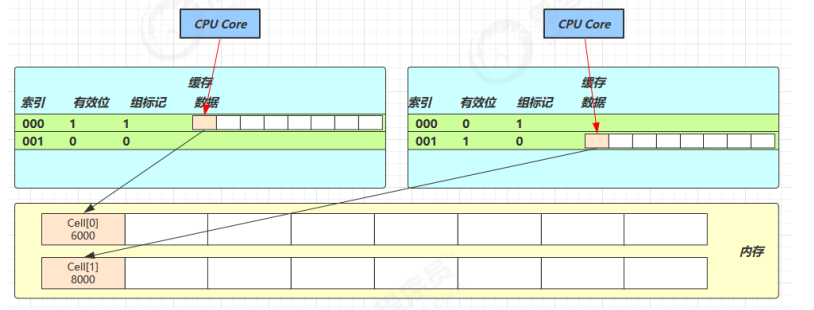
因为 Cell 是数组形式，在内存中是连续存储的，一个 Cell 为 24 字节（16 字节的对象头和 8 字节的 value），因此缓存行可以存下 2 个的 Cell 对象。这样问题来了：

Core-0 要修改 Cell[0]

Core-1 要修改 Cell[1]

无论谁修改成功，都会导致对方 Core 的缓存行失效，比如 Core-0 中 Cell[0]=6000, Cell[1]=8000 要累加Cell[0]=6001, Cell[1]=8000 ，这时会让 Core-1 的缓存行失效。

@sun.misc.Contended 用来解决这个问题，它的原理是在使用此注解的对象或字段的前后各增加 128 字节大小的padding，从而让 CPU 将对象预读至缓存时占用不同的缓存行，这样，不会造成对方缓存行的失效。



1. 累加主要调用add方法

casBase方法使用Cas更新值

*/\*\*  
 \* CASes the base field.  
 \*/***final boolean** casBase(**long** cmp, **long** val) {  
 **return *UNSAFE***.compareAndSwapLong(**this**, ***BASE***, cmp, val);  
}

**public final native boolean** compareAndSwapLong(Object var1, **long** var2, **long** var4, **long** var6);

*/\*\*  
 \* Base value, used mainly when there is no contention, but also as  
 \* a fallback during table initialization races. Updated via CAS.  
 \*/*

*//没有竞争时使用，***transient volatile long base**;

**public void** add(**long** x) {  
 Cell[] as; //as累加单元数组

**long** b, v; //b基础值，x累加值

**int** m;

Cell a;

//进入if的两个条件

//1：as有值，表示已经发生过竞争，进入if

//2：cas给base累加时失败了，表示base发生了竞争，进入if  
 **if** ((as = **cells**) != **null** || !casBase(b = **base**, b + x)) {

// uncontended表示cell没有竞争  
 **boolean** uncontended = **true**;

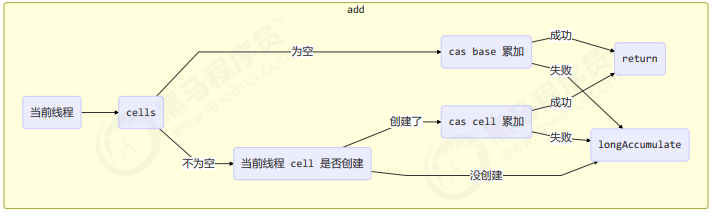
// as没有创建  
 **if** (as == **null** || (m = as.**length** - 1) < 0 ||

// 当前线程对应的cell还没有  
 (a = as[*getProbe*() & m]) == **null** ||

// cas给当前线程的cell累加失败，uncontended=fasle（a为当前线程的cell）  
 !(uncontended = a.cas(v = a.**value**, v + x)))

// 进入cell数组创建，cell创建的流程  
 longAccumulate(x, **null**, uncontended);  
 }  
}

add方法流程图：



Cas锁：0没有加锁，1加锁

*/\*\*  
 \* CASes the cellsBusy field from 0 to 1 to acquire lock.  
 \*/***final boolean** casCellsBusy() {  
 **return *UNSAFE***.compareAndSwapInt(**this**, ***CELLSBUSY***, 0, 1);  
}

8、longAccumulate方法源码分析（难）

**final void** longAccumulate(**long** x, LongBinaryOperator fn,  
 **boolean** wasUncontended) {  
 **int** h;

// 当前线程还没有对应的 cell, 需要随机生成一个 h 值用来将当前线程绑定到 cell   
 **if** ((h = *getProbe*()) == 0) {

//初始化 probe   
 ThreadLocalRandom.*current*(); *// force initialization*

//h对应新probe的值，与cell对应h = *getProbe*();  
 wasUncontended = **true**;  
 }

// collide 为 true 表示需要扩容   
 **boolean** collide = **false**; *// True if last slot nonempty* **for** (;;) {  
 Cell[] as; Cell a; **int** n; **long** v;

//1：已经有了cells  
 **if** ((as = **cells**) != **null** && (n = as.**length**) > 0) {

//还没有cell  
 **if** ((a = as[(n - 1) & h])== **null**) {

// 为 cellsBusy 加锁, 创建 cell, cell 的初始累加值为 x  
 // 成功则 break, 否则继续 continue 循环   
 **if** (**cellsBusy** == 0) { *// Try to attach new Cell* Cell r = **new** Cell(x); *// Optimistically create* **if** (**cellsBusy** == 0 && casCellsBusy()) { //casCellsBusy用于cas加锁  
 **boolean** created = **false**;  
 **try** { *// Recheck under lock* Cell[] rs; **int** m, j;  
 **if** ((rs = **cells**) != **null** &&  
 (m = rs.**length**) > 0 &&  
 rs[j = (m - 1) & h] == **null**) {  
 rs[j] = r;  
 created = **true**;  
 }  
 } **finally** {  
 **cellsBusy** = 0; //解锁  
 }  
 **if** (created) //创建cell成功，退出循环  
 **break**;  
 **continue**; *// Slot is now non-empty* }  
 }  
 collide = **false**;  
 }

//有竞争, 改变线程对应的 cell 来重试 cas   
 **else if** (!wasUncontended) *// CAS already known to fail* wasUncontended = **true**; *// Continue after rehash*

// cas 尝试累加, fn 配合 LongAccumulator 不为 null, 配合 LongAdder 为 null **else if** (a.cas(v = a.**value**, ((fn == **null**) ? v + x :  
 fn.applyAsLong(v, x))))  
 **break**;

// 如果 cells 长度已经超过了最大长度, 或者已经扩容, 改变线程对应的 cell 来重试 cas   
 **else if** (n >= ***NCPU*** || **cells** != as)  
 collide = **false**; *// At max size or stale*

// 确保 collide 为 false 进入此分支, 就不会进入下面的 else if 进行扩容了 **else if** (!collide)  
 collide = **true**;

//加锁  
 **else if** (**cellsBusy** == 0 && casCellsBusy()) {

//加锁成功，扩容  
 **try** {  
 **if** (**cells** == as) { *// Expand table unless stale* Cell[] rs = **new** Cell[n << 1];  
 **for** (**int** i = 0; i < n; ++i)  
 rs[i] = as[i];  
 **cells** = rs;  
 }  
 } **finally** {  
 **cellsBusy** = 0;  
 }  
 collide = **false**;  
 **continue**; *// Retry with expanded table* }

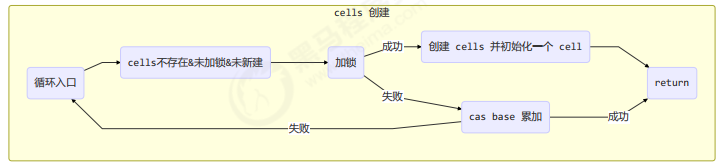
//改变线程对应的cell  
 h = *advanceProbe*(h);  
 }

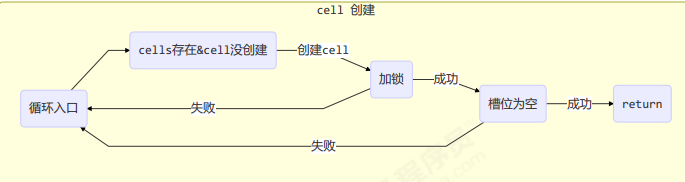
// 还没有 cells, 尝试给 cellsBusy 加锁   
 **else if** (**cellsBusy** == 0 && **cells** == as && casCellsBusy()) {

// 加锁成功, 初始化 cells, 最开始长度为 2, 并填充一个 cell  
 // 成功则 break;   
 **boolean** init = **false**;  
 **try** { *// Initialize table* **if** (**cells** == as) {  
 Cell[] rs = **new** Cell[2];  
 rs[h & 1] = **new** Cell(x);  
 **cells** = rs;  
 init = **true**;  
 }  
 } **finally** {  
 **cellsBusy** = 0;  
 }  
 **if** (init)  
 **break**;  
 }

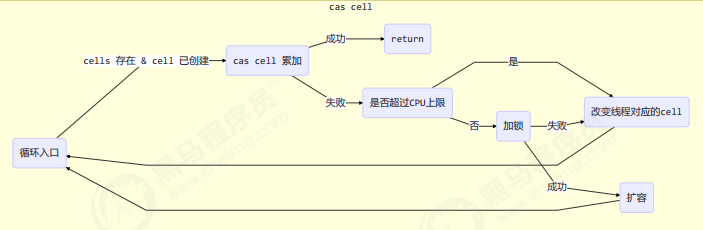
// 上两种情况失败, 尝试给 base 累加   
 **else if** (casBase(v = **base**, ((fn == **null**) ? v + x :  
 fn.applyAsLong(v, x))))  
 **break**; *// Fall back on using base* }  
}

longAccumulate流程图：





每个线程刚进入 longAccumulate 时，会尝试对应一个 cell 对象（找到一个坑位）。



1. 最后通过sum方法汇总

*/\*\*  
 \* Returns the current sum. The returned value is <em>NOT</em> an  
 \* atomic snapshot; invocation in the absence of concurrent  
 \* updates returns an accurate result, but concurrent updates that  
 \* occur while the sum is being calculated might not be  
 \* incorporated.  
 \*  
 \** ***@return*** *the sum  
 \*/***public long** sum() {  
 Cell[] as = **cells**; Cell a;  
 **long** sum = **base**;  
 **if** (as != **null**) {  
 **for** (**int** i = 0; i < as.**length**; ++i) {  
 **if** ((a = as[i]) != **null**)  
 sum += a.**value**;  
 }  
 }  
 **return** sum;  
}

Unsafe

反射创建Unsafe对象

*/\*\*  
 \* Unsafe 对象提供了非常底层的，操作内存、线程的方法，Unsafe 对象不能直接调用，  
 \* 只能通过反射获得  
 \* private static final Unsafe theUnsafe;  
 \*/*@Test  
**public void** test\_createUnsafeObj() **throws** NoSuchFieldException, IllegalAccessException {  
 *//反射获取私有的成员变量theUnsafe* **final** Field theUnsafe = Unsafe.**class**.getDeclaredField(**"theUnsafe"**);  
 *//暴力反射，可以访问private* theUnsafe.setAccessible(**true**);  
 *//静态不需要传递对象* Unsafe unsafe = (Unsafe) theUnsafe.get(**null**);  
 System.***out***.println(unsafe);  
}

Unsafe的CAS操作

*/\*\*  
 \* Unsafe的Cas操作  
 \*/*@Test  
**public void** test\_UnsafeCas() {  
 **try** {  
 Field theUnsafe = Unsafe.**class**.getDeclaredField(**"theUnsafe"**);  
 theUnsafe.setAccessible(**true**);  
 Unsafe unsafe = (Unsafe) theUnsafe.get(**null**);  
 *//获取域偏移量* **long** idOffset = unsafe.objectFieldOffset(Student.**class**.getDeclaredField(**"id"**));  
 **long** nameOffset = unsafe.objectFieldOffset(Student.**class**.getDeclaredField(**"name"**));  
 *//cas修改student对象的id,name* Student student = **new** Student();  
 System.***out***.println(student);  
 unsafe.compareAndSwapInt(student, idOffset, 0, 1);  
 unsafe.compareAndSwapObject(student, nameOffset, **null**, **"张三"**);  
 System.***out***.println(student);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 */\*\*  
 \* Student{id=0, name='null'}  
 \* Student{id=1, name='张三'}  
 \*/*}

**class** Student {  
 **private volatile int id**;  
 **private volatile** String **name**;  
  
 @Override  
 **public** String toString() {  
 **return "Student{"** +  
 **"id="** + **id** +  
 **", name='"** + **name** + **'\''** +  
 **'}'**;  
 }  
}

自己实现原子整数类

**package** com.concurrent.p7;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
**import** sun.misc.Unsafe;  
  
**import** java.lang.reflect.Field;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
*/\*\*  
 \* 利用Unsafe创建自己的AtomicInteger  
 \*/*@Slf4j(topic = **"c.Test\_MyAtomicInteger"**)  
**public class** Test\_MyAtomicInteger {  
  
 @Test  
 **public void** test\_MyAtomicInteger() {  
 *//定义共享账户变量* MyAccount myAccount = **new** MyAccount(100000);  
 List<Thread> threadList = **new** ArrayList<>();  
 **for** (**int** i = 0; i < 10; i++) {  
 threadList.add(**new** Thread(() -> {  
 **for** (**int** j = 0; j < 10; j++) {  
 myAccount.decrement(1000);  
 }  
 }));  
 }  
 threadList.forEach(t -> t.start());  
 threadList.forEach(t -> {  
 **try** {  
 t.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 *//输出账户最终余额* ***log***.debug(**"余额：{}"**, myAccount.getBalance());  
 }  
  
}  
  
**class** MyAtomicInteger {  
 *//贡献变量* **private volatile int value**;  
 *//成员对象偏移量* **private static long** *valueOffset*;  
 *//unsafe对象* **private static** Unsafe *unsafe*;  
  
 **static** {  
 **try** {  
 *//反射获取unsafe对象* Field theUnsafe = Unsafe.**class**.getDeclaredField(**"theUnsafe"**);  
 theUnsafe.setAccessible(**true**);  
 *unsafe* = (Unsafe) theUnsafe.get(**null**);  
 *//获取偏移量  
 valueOffset* = *unsafe*.objectFieldOffset(MyAtomicInteger.**class**.getDeclaredField(**"value"**));  
 } **catch** (NoSuchFieldException e) {  
 e.printStackTrace();  
 } **catch** (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
  
 }  
  
 **public** MyAtomicInteger(**int** value) {  
 **this**.**value** = value;  
 }  
  
 *//获取数值* **public int** getValue() {  
 **return value**;  
 }  
  
 *//cas减操作* **public boolean** decrement(**int** v) {  
 **int** prev, next;  
 **do** {  
 prev = getValue();  
 next = prev - v;  
 } **while** (!*unsafe*.compareAndSwapInt(**this**, *valueOffset*, prev, next));  
 **return true**;  
 }  
}  
  
*//账户类***class** MyAccount {  
 **private** MyAtomicInteger **balance**;  
  
 **public** MyAccount(**int** balance) {  
 **this**.**balance** = **new** MyAtomicInteger(balance);  
 }  
  
 **public void** decrement(**int** value) {  
 **balance**.decrement(value);  
 }  
  
 **public int** getBalance() {  
 **return balance**.getValue();  
 }  
}

运行结果：

