以同时执行代码（1）获取 id 对应的记录并把记录放入线程本地栈里面，然后可以同时执行代码（2）对自己栈上的记录进行修改，多个线程修改后各自的 entry 里面的属性应该都不一样了。然后多个线程可以同时执行代码（3），代码（3）中的 update 语句的 where 条件里面加入了 version=#{version} 条件，并且 set 语句中多了 version=${version}+1 表达式，该表达式的意思是，如果数据库里面 id =#{id} and version=#{version} 的记录存在，则更新 version 的值为原来的值加 1，这有点 CAS 操作的意思。

假设多个线程同时执行 updateEntry 并传递相同的 id，那么它们执行代码（1）时获取的 Entry 是同一个，获取的 Entry 里面的 version 值都是相同的（这里假设 version=0）。当多个线程执行代码（3）时，由于 update 语句本身是原子性的，假如线程 A 执行 update 成功了，那么这时候 id 对应的记录的 version 值由原始 version 值变为了 1。其他线程执行代码（3）更新时发现数据库里面已经没有了 version=0 的语句，所以会返回影响行号 0。在业务上根据返回值为 0 就可以知道当前更新没有成功，那么接下来有两个做法，如果业务发现更新失败了，下面可以什么都不做，也可以选择重试，如果选择重试，则 updateEntry 的代码可以修改为如下。

```
public boolean updateEntry(long id){

    boolean result = false;
    int retryNum = 5;
    while(retryNum>0){

        //(1.1)使用乐观锁获取指定记录
        EntryObject entry = query("select * from table1 where id = #{id}",id);

       //(2.1)修改记录内容，version字段不能被修改
        String name = generatorName(entry);
        entry.setName(name);
        。。。。

       //(3.1)update操作
       int count = update("update table1 set name=#{name},age=#{age},version=${versi
         on}+1 where id =#{id} and version=#{version}",entry);

       if(count == 1){
          result = true;
          break;
       }
```

```
            retryNum--;
        }

        return result;

}
```

如上代码使用 retryNum 设置更新失败后的重试次数，如果代码（3.1）执行后返回 0，则说明代码（1.1）获取的记录已经被修改了，则循环一次，重新通过代码（1.1）获取最新的数据，然后再次执行代码（3.1）尝试更新。这类似 CAS 的自旋操作，只是这里没有使用死循环，而是指定了尝试次数。

乐观锁并不会使用数据库提供的锁机制，一般在表中添加 version 字段或者使用业务状态来实现。乐观锁直到提交时才锁定，所以不会产生任何死锁。

2.12.2　公平锁与非公平锁

根据线程获取锁的抢占机制，锁可以分为公平锁和非公平锁，公平锁表示线程获取锁的顺序是按照线程请求锁的时间早晚来决定的，也就是最早请求锁的线程将最早获取到锁。而非公平锁则在运行时闯入，也就是先来不一定先得。

ReentrantLock 提供了公平和非公平锁的实现。

- 公平锁：ReentrantLock pairLock = new ReentrantLock(true)。
- 非公平锁：ReentrantLock pairLock = new ReentrantLock(false)。如果构造函数不传递参数，则默认是非公平锁。

例如，假设线程 A 已经持有了锁，这时候线程 B 请求该锁其将会被挂起。当线程 A 释放锁后，假如当前有线程 C 也需要获取该锁，如果采用非公平锁方式，则根据线程调度策略，线程 B 和 线程 C 两者之一可能获取锁，这时候不需要任何其他干涉，而如果使用公平锁则需要把 C 挂起，让 B 获取当前锁。

在没有公平性需求的前提下尽量使用非公平锁，因为公平锁会带来性能开销。

2.12.3　独占锁与共享锁

根据锁只能被单个线程持有还是能被多个线程共同持有，锁可以分为独占锁和共享锁。

独占锁保证任何时候都只有一个线程能得到锁，ReentrantLock 就是以独占方式实现的。共享锁则可以同时由多个线程持有，例如 ReadWriteLock 读写锁，它允许一个资源可以被多线程同时进行读操作。

独占锁是一种悲观锁，由于每次访问资源都先加上互斥锁，这限制了并发性，因为读操作并不会影响数据的一致性，而独占锁只允许在同一时间由一个线程读取数据，其他线程必须等待当前线程释放锁才能进行读取。

共享锁则是一种乐观锁，它放宽了加锁的条件，允许多个线程同时进行读操作。

2.12.4 什么是可重入锁

当一个线程要获取一个被其他线程持有的独占锁时，该线程会被阻塞，那么当一个线程再次获取它自己已经获取的锁时是否会被阻塞呢？如果不被阻塞，那么我们说该锁是可重入的，也就是只要该线程获取了该锁，那么可以无限次数（在高级篇中我们将知道，严格来说是有限次数）地进入被该锁锁住的代码。

下面看一个例子，看看在什么情况下会使用可重入锁。

```
public class Hello{
     public synchronized void helloA(){
        System.out.println("hello");
     }

     public synchronized void helloB(){
        System.out.println("hello B");
        helloA();
     }

}
```

在如上代码中，调用 helloB 方法前会先获取内置锁，然后打印输出。之后调用 helloA 方法，在调用前会先去获取内置锁，如果内置锁不是可重入的，那么调用线程将会一直被阻塞。

实际上，synchronized 内部锁是可重入锁。可重入锁的原理是在锁内部维护一个线程标示，用来标示该锁目前被哪个线程占用，然后关联一个计数器。一开始计数器值为 0，说明该锁没有被任何线程占用。当一个线程获取了该锁时，计数器的值会变成 1，这时其

他线程再来获取该锁时会发现锁的所有者不是自己而被阻塞挂起。

但是当获取了该锁的线程再次获取锁时发现锁拥有者是自己，就会把计数器值加 +1，当释放锁后计数器值 -1。当计数器值为 0 时，锁里面的线程标示被重置为 null，这时候被阻塞的线程会被唤醒来竞争获取该锁。

2.12.5　自旋锁

由于 Java 中的线程是与操作系统中的线程一一对应的，所以当一个线程在获取锁（比如独占锁）失败后，会被切换到内核状态而被挂起。当该线程获取到锁时又需要将其切换到内核状态而唤醒该线程。而从用户状态切换到内核状态的开销是比较大的，在一定程度上会影响并发性能。自旋锁则是，当前线程在获取锁时，如果发现锁已经被其他线程占有，它不马上阻塞自己，在不放弃 CPU 使用权的情况下，多次尝试获取（默认次数是 10，可以使用 -XX:PreBlockSpinsh 参数设置该值），很有可能在后面几次尝试中其他线程已经释放了锁。如果尝试指定的次数后仍没有获取到锁则当前线程才会被阻塞挂起。由此看来自旋锁是使用 CPU 时间换取线程阻塞与调度的开销，但是很有可能这些 CPU 时间白白浪费了。

2.13　总结

本章主要介绍了并发编程的基础知识，为后面在高级篇讲解并发包源码打下了基础，并结合图示形象地讲述了为什么要使用多线程编程，多线程编程存在的线程安全问题，以及什么是内存可见性问题。然后讲解了 synchronized 和 volatile 关键字，并且强调前者既保证内存的可见性又保证原子性，而后者则主要保证内存可见性，但是二者的内存语义很相似。最后讲解了什么是 CAS 和线程间同步以及各种锁的概念，这些都为后面讲解 JUC 包源码奠定了基础。

第二部分

Java并发编程高级篇

在第一部分中我们介绍了并发编程的基础知识，而本部分则主要讲解并发包中一些主要组件的实现原理。

第3章

Java并发包中ThreadLocalRandom类原理剖析

ThreadLocalRandom 类是 JDK 7 在 JUC 包下新增的随机数生成器，它弥补了 Random 类在多线程下的缺陷。本章讲解为何要在 JUC 下新增该类，以及该类的实现原理。

3.1 Random 类及其局限性

在 JDK 7 之前包括现在，java.util.Random 都是使用比较广泛的随机数生成工具类，而且 java.lang.Math 中的随机数生成也使用的是 java.util.Random 的实例。下面先看看 java.util.Random 的使用方法。

```
public class RandomTest {
    public static void main(String[] args) {

        //(1)创建一个默认种子的随机数生成器
        Random random = new Random();
        //(2)输出10个在0~5（包含0，不包含5）之间的随机数
        for (int i = 0; i < 10; ++i) {
            System.out.println(random.nextInt(5));
        }
    }
}
```

代码（1）创建一个默认随机数生成器，并使用默认的种子。

代码（2）输出 10 个在 0~5（包含 0，不包含 5）之间的随机数。

随机数的生成需要一个默认的种子，这个种子其实是一个 long 类型的数字，你可以在

创建 Random 对象时通过构造函数指定，如果不指定则在默认构造函数内部生成一个默认的值。有了默认的种子后，如何生成随机数呢？

```
public int nextInt(int bound) {
    //(3)参数检查
    if (bound <= 0)
        throw new IllegalArgumentException(BadBound);
    //(4)根据老的种子生成新的种子
    int r = next(31);
    //(5)根据新的种子计算随机数
    ...
    return r;
}
```

由此可见，新的随机数的生成需要两个步骤：

- 首先根据老的种子生成新的种子。
- 然后根据新的种子来计算新的随机数。

其中步骤（4）我们可以抽象为 seed=f(seed), 其中 f 是一个固定的函数，比如 seed=f(seed)=a*seed+b；步骤（5）也可以抽象为 g(seed,bound)，其中 g 是一个固定的函数，比如 g(seed,bound)=(int)((bound* (long)seed) >> 31)。在单线程情况下每次调用 nextInt 都是根据老的种子计算出新的种子，这是可以保证随机数产生的随机性的。但是在多线程下多个线程可能都拿同一个老的种子去执行步骤（4）以计算新的种子，这会导致多个线程产生的新种子是一样的，由于步骤（5）的算法是固定的，所以会导致多个线程产生相同的随机值，这并不是我们想要的。所以步骤（4）要保证原子性，也就是说当多个线程根据同一个老种子计算新种子时，第一个线程的新种子被计算出来后，第二个线程要丢弃自己老的种子，而使用第一个线程的新种子来计算自己的新种子，依此类推，只有保证了这个，才能保证在多线程下产生的随机数是随机的。Random 函数使用一个原子变量达到了这个效果，在创建 Random 对象时初始化的种子就被保存到了种子原子变量里面，下面看 next() 的代码。

```
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        //(6)
        oldseed = seed.get();
        //(7)
```

```
            nextseed = (oldseed * multiplier + addend) & mask;
            //(8)
        } while (!seed.compareAndSet(oldseed, nextseed));
        //(9)
        return (int)(nextseed >>> (48 - bits));
    }
```

代码（6）获取当前原子变量种子的值。

代码（7）根据当前种子值计算新的种子。

代码（8）使用CAS操作，它使用新的种子去更新老的种子，在多线程下可能多个线程都同时执行到了代码（6），那么可能多个线程拿到的当前种子的值是同一个，然后执行步骤（7）计算的新种子也都是一样的，但是步骤（8）的CAS操作会保证只有一个线程可以更新老的种子为新的，失败的线程会通过循环重新获取更新后的种子作为当前种子去计算老的种子，这就解决了上面提到的问题，保证了随机数的随机性。

代码（9）使用固定算法根据新的种子计算随机数。

总结：每个Random实例里面都有一个原子性的种子变量用来记录当前的种子值，当要生成新的随机数时需要根据当前种子计算新的种子并更新回原子变量。在多线程下使用单个Random实例生成随机数时，当多个线程同时计算随机数来计算新的种子时，多个线程会竞争同一个原子变量的更新操作，由于原子变量的更新是CAS操作，同时只有一个线程会成功，所以会造成大量线程进行自旋重试，这会降低并发性能，所以ThreadLocalRandom应运而生。

3.2 ThreadLocalRandom

为了弥补多线程高并发情况下Random的缺陷，在JUC包下新增了ThreadLocalRandom类。下面首先看下如何使用它。

```
public class RandomTest {

    public static void main(String[] args) {
        //(10)获取一个随机数生成器
        ThreadLocalRandom random =  ThreadLocalRandom.current();

        //(11)输出10个在0~5（包含0，不包含5）之间的随机数
        for (int i = 0; i < 10; ++i) {
```

```
            System.out.println(random.nextInt(5));
        }
    }
}
```

其中，代码（10）调用 ThreadLocalRandom.current() 来获取当前线程的随机数生成器。下面来分析下 ThreadLocalRandom 的实现原理。从名字上看它会让我们联想到在基础篇中讲解的 ThreadLocal：ThreadLocal 通过让每一个线程复制一份变量，使得在每个线程对变量进行操作时实际是操作自己本地内存里面的副本，从而避免了对共享变量进行同步。实际上 ThreadLocalRandom 的实现也是这个原理，Random 的缺点是多个线程会使用同一个原子性种子变量，从而导致对原子变量更新的竞争，如图 3-1 所示。

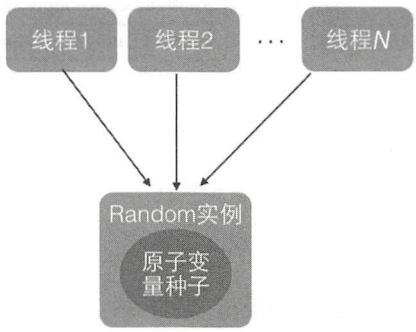


图 3-1

那么，如果每个线程都维护一个种子变量，则每个线程生成随机数时都根据自己老的种子计算新的种子，并使用新种子更新老的种子，再根据新种子计算随机数，就不会存在竞争问题了，这会大大提高并发性能。ThreadLocalRandom 原理如图 3-2 所示。

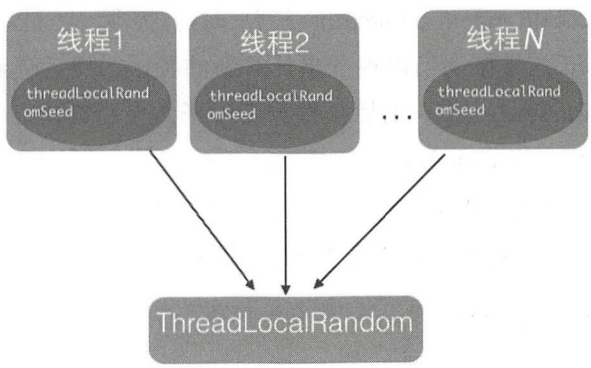


图 3-2

3.3 源码分析

首先看下 ThreadLocalRandom 的类图结构，如图 3-3 所示。

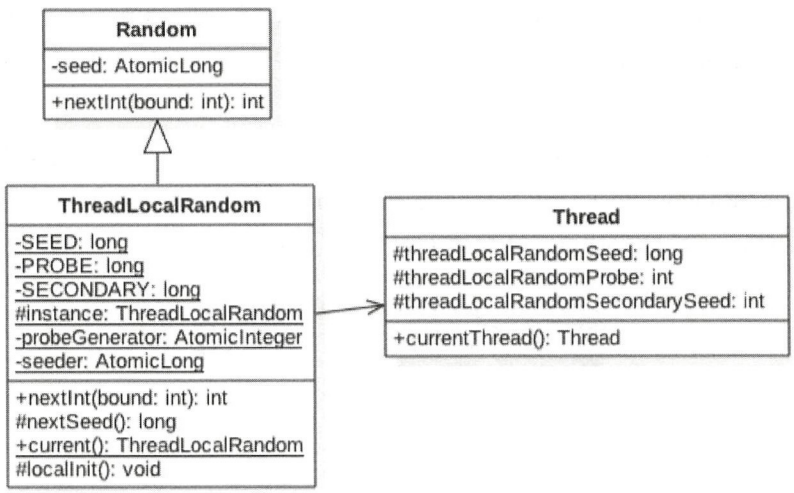


图 3-3

从图中可以看出 ThreadLocalRandom 类继承了 Random 类并重写了 nextInt 方法，在 ThreadLocalRandom 类中并没有使用继承自 Random 类的原子性种子变量。在 ThreadLocalRandom 中并没有存放具体的种子，具体的种子存放在具体的调用线程的 threadLocalRandomSeed 变量里面。ThreadLocalRandom 类似于 ThreadLocal 类，就是个工具类。当线程调用 ThreadLocalRandom 的 current 方法时，ThreadLocalRandom 负责初始化调用线程的 threadLocalRandomSeed 变量，也就是初始化种子。

当调用 ThreadLocalRandom 的 nextInt 方法时，实际上是获取当前线程的 threadLocalRandomSeed 变量作为当前种子来计算新的种子，然后更新新的种子到当前线程的 threadLocalRandomSeed 变量，而后再根据新种子并使用具体算法计算随机数。这里需要注意的是，threadLocalRandomSeed 变量就是 Thread 类里面的一个普通 long 变量，它并不是原子性变量。其实道理很简单，因为这个变量是线程级别的，所以根本不需要使用原子性变量，如果你还是不理解可以思考下 ThreadLocal 的原理。

其中 seeder 和 probeGenerator 是两个原子性变量，在初始化调用线程的种子和探针变量时会用到它们，每个线程只会使用一次。

另外，变量 instance 是 ThreadLocalRandom 的一个实例，该变量是 static 的。当多线程通过 ThreadLocalRandom 的 current 方法获取 ThreadLocalRandom 的实例时，其实获取的是同一个实例。但是由于具体的种子是存放在线程里面的，所以在 ThreadLocalRandom 的实例里面只包含与线程无关的通用算法，所以它是线程安全的。

下面看看 ThreadLocalRandom 的主要代码的实现逻辑。

1. Unsafe 机制

```
private static final sun.misc.Unsafe UNSAFE;
private static final long SEED;
private static final long PROBE;
private static final long SECONDARY;
static {
    try {
        //获取unsafe实例
        UNSAFE = sun.misc.Unsafe.getUnsafe();
        Class<?> tk = Thread.class;
        //获取Thread类里面threadLocalRandomSeed变量在Thread实例里面的偏移量
        SEED = UNSAFE.objectFieldOffset
            (tk.getDeclaredField("threadLocalRandomSeed"));
        //获取Thread类里面threadLocalRandomProbe变量在Thread实例里面的偏移量
        PROBE = UNSAFE.objectFieldOffset
            (tk.getDeclaredField("threadLocalRandomProbe"));
        //获取Thread类里面threadLocalRandomSecondarySeed变量在Thread实例里面的偏移
          量，这个值在后面讲解LongAdder时会用到
        SECONDARY = UNSAFE.objectFieldOffset
            (tk.getDeclaredField("threadLocalRandomSecondarySeed"));
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

2. ThreadLocalRandom current() 方法

该方法获取 ThreadLocalRandom 实例，并初始化调用线程中的 threadLocalRandomSeed 和 threadLocalRandomProbe 变量。

```
static final ThreadLocalRandom instance = new ThreadLocalRandom();
public static ThreadLocalRandom current() {
    //(12)
```

```
    if (UNSAFE.getInt(Thread.currentThread(), PROBE) == 0)
        //(13)
        localInit();
    //(14)
    return instance;
}

static final void localInit() {
    int p = probeGenerator.addAndGet(PROBE_INCREMENT);
    int probe = (p == 0) ? 1 : p; // skip 0
    long seed = mix64(seeder.getAndAdd(SEEDER_INCREMENT));
    Thread t = Thread.currentThread();
    UNSAFE.putLong(t, SEED, seed);
    UNSAFE.putInt(t, PROBE, probe);
}
```

在如上代码（12）中，如果当前线程中 threadLocalRandomProbe 的变量值为 0（默认情况下线程的这个变量值为 0），则说明当前线程是第一次调用 ThreadLocalRandom 的 current 方法，那么就需要调用 localInit 方法计算当前线程的初始化种子变量。这里为了延迟初始化，在不需要使用随机数功能时就不初始化 Thread 类中的种子变量，这是一种优化。

代码（13）首先根据 probeGenerator 计算当前线程中 threadLocalRandomProbe 的初始化值，然后根据 seeder 计算当前线程的初始化种子，而后把这两个变量设置到当前线程。代码（14）返回 ThreadLocalRandom 的实例。需要注意的是，这个方法是静态方法，多个线程返回的是同一个 ThreadLocalRandom 实例。

3. int nextInt(int bound) 方法

计算当前线程的下一个随机数。

```
public int nextInt(int bound) {
    //(15)参数校验
    if (bound <= 0)
        throw new IllegalArgumentException(BadBound);
    //(16) 根据当前线程中的种子计算新种子
    int r = mix32(nextSeed());
    //(17)根据新种子和bound计算随机数
    int m = bound - 1;
    if ((bound & m) == 0) // power of two
        r &= m;
    else { // reject over-represented candidates
        for (int u = r >>> 1;
```

```
                u + m - (r = u % bound) < 0;
                u = mix32(nextSeed()) >>> 1)
               ;
        }
        return r;
    }
```

如上代码的逻辑步骤与 Random 相似，我们重点看下 nextSeed() 方法。

```
final long nextSeed() {
    Thread t; long r; //
    UNSAFE.putLong(t = Thread.currentThread(), SEED,
                   r = UNSAFE.getLong(t, SEED) + GAMMA);
    return r;
}
```

在如上代码中，首先使用 r = UNSAFE.getLong(t, SEED) 获取当前线程中 threadLocalRandomSeed 变量的值，然后在种子的基础上累加 GAMMA 值作为新种子，而后使用 UNSAFE 的 putLong 方法把新种子放入当前线程的 threadLocalRandomSeed 变量中。

3.4　总结

本章首先讲解了 Random 的实现原理以及 Random 在多线程下需要竞争种子原子变量更新操作的缺点，从而引出 ThreadLocalRandom 类。ThreadLocalRandom 使用 ThreadLocal 的原理，让每个线程都持有一个本地的种子变量，该种子变量只有在使用随机数时才会被初始化。在多线程下计算新种子时是根据自己线程内维护的种子变量进行更新，从而避免了竞争。

第4章

Java并发包中原子操作类原理剖析

JUC包提供了一系列的原子性操作类，这些类都是使用非阻塞算法CAS实现的，相比使用锁实现原子性操作这在性能上有很大提高。由于原子性操作类的原理都大致相同，所以本章只讲解最简单的AtomicLong类的实现原理以及JDK 8中新增的LongAdder和LongAccumulator类的原理。有了这些基础，再去理解其他原子性操作类的实现就不会感到困难了。

4.1 原子变量操作类

JUC并发包中包含有AtomicInteger、AtomicLong和AtomicBoolean等原子性操作类，它们的原理类似，本章讲解AtomicLong类。AtomicLong是原子性递增或者递减类，其内部使用Unsafe来实现，我们看下面的代码。

```
public class AtomicLong extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 1927816293512124184L;

    // (1)获取Unsafe实例
    private static final Unsafe unsafe = Unsafe.getUnsafe();

    //(2)存放变量value的偏移量
    private static final long valueOffset;

     //(3)判断JVM是否支持Long类型无锁CAS
    static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();
    private static native boolean VMSupportsCS8();

    static {
```

```
        try {
            //(4)获取value在AtomicLong中的偏移量
            valueOffset = unsafe.objectFieldOffset
                (AtomicLong.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    //(5)实际变量值
    private volatile long value;

    public AtomicLong(long initialValue) {
        value = initialValue;
    }
    ....
}
```

代码(1)通过 Unsafe.getUnsafe()方法获取到 Unsafe 类的实例,这里你可能会有疑问,为何能通过 Unsafe.getUnsafe() 方法获取到 Unsafe 类的实例?其实这是因为 AtomicLong 类也是在 rt.jar 包下面的,AtomicLong 类就是通过 BootStarp 类加载器进行加载的。

代码(5)中的 value 被声明为 volatile 的,这是为了在多线程下保证内存可见性,value 是具体存放计数的变量。

代码(2)(4)获取 value 变量在 AtomicLong 类中的偏移量。

下面重点看下 AtomicLong 中的主要函数。

1. 递增和递减操作代码

```
//(6)调用unsafe方法,原子性设置value值为原始值+1,返回值为递增后的值
public final long incrementAndGet() {
    return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;
}

//(7)调用unsafe方法,原子性设置value值为原始值-1,返回值为递减之后的值
public final long decrementAndGet() {
    return unsafe.getAndAddLong(this, valueOffset, -1L) - 1L;
}

//(8)调用unsafe方法,原子性设置value值为原始值+1,返回值为原始值
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}
```

```
//(9)调用unsafe方法，原子性设置value值为原始值-1，返回值为原始值
public final long getAndDecrement() {
    return unsafe.getAndAddLong(this, valueOffset, -1L);
}
```

在如上代码内部都是通过调用 Unsafe 的 getAndAddLong 方法来实现操作，这个函数是个原子性操作，这里第一个参数是 AtomicLong 实例的引用，第二个参数是 value 变量在 AtomicLong 中的偏移值，第三个参数是要设置的第二个变量的值。

其中，getAndIncrement 方法在 JDK 7 中的实现逻辑为

```
public final long getAndIncrement() {
        while (true) {
            long current = get();
            long next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
```

在如上代码中，每个线程是先拿到变量的当前值（由于 value 是 volatile 变量，所以这里拿到的是最新的值），然后在工作内存中对其进行增 1 操作，而后使用 CAS 修改变量的值。如果设置失败，则循环继续尝试，直到设置成功。

而 JDK 8 中的逻辑为

```
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}
```

其中 JDK 8 中 unsafe.getAndAddLong 的代码为

```
public final long getAndAddLong(Object paramObject, long paramLong1, long
paramLong2)
  {
    long l;
    do
    {
      l = getLongvolatile(paramObject, paramLong1);
    } while (!compareAndSwapLong(paramObject, paramLong1, l, l + paramLong2));
    return l;
  }
```

可以看到，JDK 7 的 AtomicLong 中的循环逻辑已经被 JDK 8 中的原子操作类 UNsafe 内置了，之所以内置应该是考虑到这个函数在其他地方也会用到，而内置可以提高复用性。

2. boolean compareAndSet(long expect, long update) 方法

```
public final boolean compareAndSet(long expect, long update) {
    return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
}
```

由如上代码可知，在内部还是调用了 unsafe.compareAndSwapLong 方法。如果原子变量中的 value 值等于 expect，则使用 update 值更新该值并返回 true, 否则返回 false。

下面通过一个多线程使用 AtomicLong 统计 0 的个数的例子来加深对 AtomicLong 的理解。

```
/**
  统计0的个数
 */
public class Atomic
{
   //(10)创建Long型原子计数器
   private static  AtomicLong atomicLong = new AtomicLong();
   //(11)创建数据源
   private static Integer[] arrayOne = new Integer[]{0,1,2,3,0,5,6,0,56,0};
   private static Integer[] arrayTwo = new Integer[]{10,1,2,3,0,5,6,0,56,0};

    public static void main( String[] args ) throws InterruptedException
    {
     //(12)线程one统计数组arrayOne中0的个数
        Thread threadOne = new Thread(new Runnable() {

            @Override
            public void run() {

             int size = arrayOne.length;
             for(int i=0;i<size;++i){
                 if(arrayOne[i].intValue() == 0){

                     atomicLong.incrementAndGet();
                 }
             }
```

```
            }
        });
        //(13)线程two统计数组arrayTwo中0的个数
        Thread threadTwo = new Thread(new Runnable() {

            @Override
            public void run() {

                int size = arrayTwo.length;
                for(int i=0;i<size;++i){
                    if(arrayTwo[i].intValue() == 0){

                        atomicLong.incrementAndGet();
                    }
                }

            }
        });

        //(14)启动子线程
        threadOne.start();
        threadTwo.start();

        //(15)等待线程执行完毕
        threadOne.join();
        threadTwo.join();

        System.out.println("count 0:" + atomicLong.get());

    }
}
```

输出结果为

```
count 0:7
```

如上代码中的两个线程各自统计自己所持数据中 0 的个数，每当找到一个 0 就会调用 AtomicLong 的原子性递增方法。

在没有原子类的情况下，实现计数器需要使用一定的同步措施，比如使用 synchronized 关键字等，但是这些都是阻塞算法，对性能有一定损耗，而本章介绍的这些原子操作类都使用 CAS 非阻塞算法，性能更好。但是在高并发情况下 AtomicLong 还会存

在性能问题。JDK 8 提供了一个在高并发下性能更好的 LongAdder 类，下面我们来讲解这个类。

4.2　JDK 8 新增的原子操作类 LongAdder

4.2.1　LongAdder 简单介绍

前面讲过，AtomicLong 通过 CAS 提供了非阻塞的原子性操作，相比使用阻塞算法的同步器来说它的性能已经很好了，但是 JDK 开发组并不满足于此。使用 AtomicLong 时，在高并发下大量线程会同时去竞争更新同一个原子变量，但是由于同时只有一个线程的 CAS 操作会成功，这就造成了大量线程竞争失败后，会通过无限循环不断进行自旋尝试 CAS 的操作，而这会白白浪费 CPU 资源。

因此 JDK 8 新增了一个原子性递增或者递减类 LongAdder 用来克服在高并发下使用 AtomicLong 的缺点。既然 AtomicLong 的性能瓶颈是由于过多线程同时去竞争一个变量的更新而产生的，那么如果把一个变量分解为多个变量，让同样多的线程去竞争多个资源，是不是就解决了性能问题？是的，LongAdder 就是这个思路。下面通过图来理解两者设计的不同之处，如图 4-1 所示。

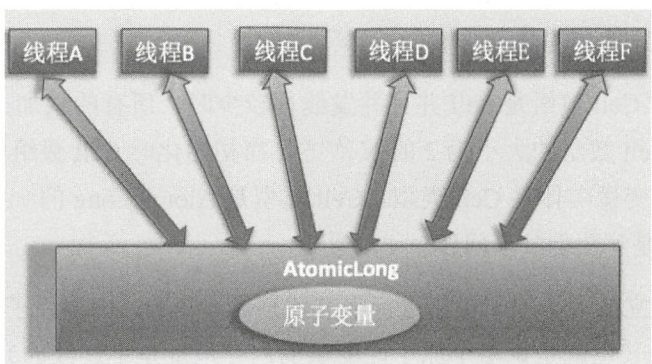


图 4-1

如图 4-1 所示，使用 AtomicLong 时，是多个线程同时竞争同一个原子变量。

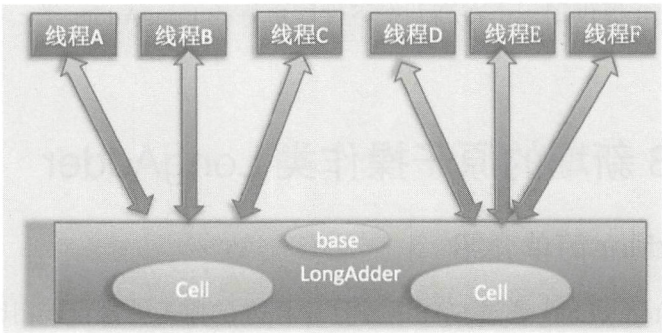


图4-2

如图4-2所示，使用LongAdder时，则是在内部维护多个Cell变量，每个Cell里面有一个初始值为0的long型变量，这样，在同等并发量的情况下，争夺单个变量更新操作的线程量会减少，这变相地减少了争夺共享资源的并发量。另外，多个线程在争夺同一个Cell原子变量时如果失败了，它并不是在当前Cell变量上一直自旋CAS重试，而是尝试在其他Cell的变量上进行CAS尝试，这个改变增加了当前线程重试CAS成功的可能性。最后，在获取LongAdder当前值时，是把所有Cell变量的value值累加后再加上base返回的。

LongAdder维护了一个延迟初始化的原子性更新数组（默认情况下Cell数组是null）和一个基值变量base。由于Cells占用的内存是相对比较大的，所以一开始并不创建它，而是在需要时创建，也就是惰性加载。

当一开始判断Cell数组是null并且并发线程较少时，所有的累加操作都是对base变量进行的。保持Cell数组的大小为2的N次方，在初始化时Cell数组中的Cell元素个数为2，数组里面的变量实体是Cell类型。Cell类型是AtomicLong的一个改进，用来减少缓存的争用，也就是解决伪共享问题。

对于大多数孤立的多个原子操作进行字节填充是浪费的，因为原子性操作都是无规律地分散在内存中的（也就是说多个原子性变量的内存地址是不连续的），多个原子变量被放入同一个缓存行的可能性很小。但是原子性数组元素的内存地址是连续的，所以数组内的多个元素能经常共享缓存行，因此这里使用@sun.misc.Contended注解对Cell类进行字节填充，这防止了数组中多个元素共享一个缓存行，在性能上是一个提升。

4.2.2　LongAdder 代码分析

为了解决高并发下多线程对一个变量 CAS 争夺失败后进行自旋而造成的降低并发性能问题，LongAdder 在内部维护多个 Cell 元素（一个动态的 Cell 数组）来分担对单个变量进行争夺的开销。下面围绕以下话题从源码角度来分析 LongAdder 的实现：（1）LongAdder 的结构是怎样的？（2）当前线程应该访问 Cell 数组里面的哪一个 Cell 元素？（3）如何初始化 Cell 数组？（4）Cell 数组如何扩容？（5）线程访问分配的 Cell 元素有冲突后如何处理？（6）如何保证线程操作被分配的 Cell 元素的原子性？

首先看下 LongAdder 的类图结构，如图 4-3 所示。

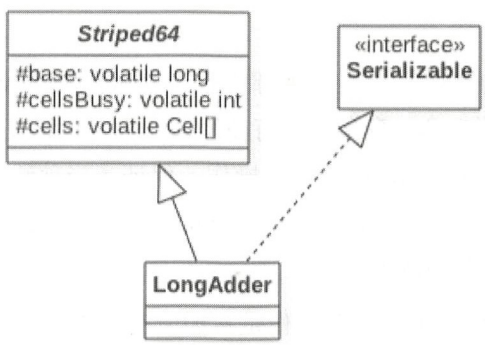


图 4-3

由该图可知，LongAdder 类继承自 Striped64 类，在 Striped64 内部维护着三个变量。LongAdder 的真实值其实是 base 的值与 Cell 数组里面所有 Cell 元素中的 value 值的累加，base 是个基础值，默认为 0。cellsBusy 用来实现自旋锁，状态值只有 0 和 1，当创建 Cell 元素，扩容 Cell 数组或者初始化 Cell 数组时，使用 CAS 操作该变量来保证同时只有一个线程可以进行其中之一的操作。

下面看 Cell 的构造。

```
@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);
        }
    }
}
```

可以看到，Cell 的构造很简单，其内部维护一个被声明为 volatile 的变量，这里声明为 volatile 是因为线程操作 value 变量时没有使用锁，为了保证变量的内存可见性这里将其声明为 volatile 的。另外 cas 函数通过 CAS 操作，保证了当前线程更新时被分配的 Cell 元素中 value 值的原子性。另外，Cell 类使用 @sun.misc.Contended 修饰是为了避免伪共享。到这里我们回答了问题 1 和问题 6。

- long sum() 返回当前的值，内部操作是累加所有 Cell 内部的 value 值后再累加 base。例如下面的代码，由于计算总和时没有对 Cell 数组进行加锁，所以在累加过程中可能有其他线程对 Cell 中的值进行了修改，也有可能对数组进行了扩容，所以 sum 返回的值并不是非常精确的，其返回值并不是一个调用 sum 方法时的原子快照值。

```
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;
}
```

- void reset() 为重置操作，如下代码把 base 置为 0，如果 Cell 数组有元素，则元素值被重置为 0。

```
public void reset() {
       Cell[] as = cells; Cell a;
       base = 0L;
       if (as != null) {
           for (int i = 0; i < as.length; ++i) {
                if ((a = as[i]) != null)
                     a.value = 0L;
           }
       }
}
```

- long sumThenReset() 是 sum 的改造版本，如下代码在使用 sum 累加对应的 Cell 值后，把当前 Cell 的值重置为 0，base 重置为 0。这样，当多线程调用该方法时会有问题，比如考虑第一个调用线程清空 Cell 的值，则后一个线程调用时累加的都是 0 值。

```
public long sumThenReset() {
       Cell[] as = cells; Cell a;
       long sum = base;
       base = 0L;
       if (as != null) {
           for (int i = 0; i < as.length; ++i) {
                if ((a = as[i]) != null) {
                     sum += a.value;
                     a.value = 0L;
                 }
            }
         }
         return sum;
}
```

- long longValue() 等价于 sum()。

下面主要看下 add 方法的实现，从这个方法里面就可以找到其他问题的答案。

```
public void add(long x) {
    Cell[] as; long b, v; int m; Cell a;
    if ((as = cells) != null || !casBase(b = base, b + x)) {//(1)
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||//(2)
            (a = as[getProbe() & m]) == null ||//(3)
            !(uncontended = a.cas(v = a.value, v + x)))//(4)
            longAccumulate(x, null, uncontended);//(5)
     }
  }
```

```
final boolean casBase(long cmp, long val) {
    return UNSAFE.compareAndSwapLong(this, BASE, cmp, val);
}
```

代码（1）首先看 cells 是否为 null，如果为 null 则当前在基础变量 base 上进行累加，这时候就类似 AtomicLong 的操作。

如果 cells 不为 null 或者线程执行代码（1）的 CAS 操作失败了，则会去执行代码（2）。代码（2）（3）决定当前线程应该访问 cells 数组里面的哪一个 Cell 元素，如果当前线程映射的元素存在则执行代码（4），使用 CAS 操作去更新分配的 Cell 元素的 value 值，如果当前线程映射的元素不存在或者存在但是 CAS 操作失败则执行代码（5）。其实将代码（2）（3）（4）合起来看就是获取当前线程应该访问的 cells 数组的 Cell 元素，然后进行 CAS 更新操作，只是在获取期间如果有些条件不满足则会跳转到代码（5）执行。另外当前线程应该访问 cells 数组的哪一个 Cell 元素是通过 getProbe() & m 进行计算的，其中 m 是当前 cells 数组元素个数 -1，getProbe() 则用于获取当前线程中变量 threadLocalRandomProbe 的值，这个值一开始为 0，在代码（5）里面会对其进行初始化。并且当前线程通过分配的 Cell 元素的 cas 函数来保证对 Cell 元素 value 值更新的原子性，到这里我们回答了问题 2 和问题 6。

下面重点研究 longAccumulate 的代码逻辑，这是 cells 数组被初始化和扩容的地方。

```
final void longAccumulate(long x, LongBinaryOperator fn,
                          boolean wasUncontended) {
    //(6) 初始化当前线程的变量threadLocalRandomProbe的值
    int h;
    if ((h = getProbe()) == 0) {
        ThreadLocalRandom.current(); //
        h = getProbe();
        wasUncontended = true;
    }
    boolean collide = false;
    for (;;) {
        Cell[] as; Cell a; int n; long v;
        if ((as = cells) != null && (n = as.length) > 0) {//(7)
            if ((a = as[(n - 1) & h]) == null) {//(8)
                if (cellsBusy == 0) {       // Try to attach new Cell
                    Cell r = new Cell(x);   // Optimistically create
                    if (cellsBusy == 0 && casCellsBusy()) {
                        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;
                }
                if (created)
                    break;
                continue;           // Slot is now non-empty
            }
        }
        collide = false;
    }
    else if (!wasUncontended)       // CAS already known to fail
        wasUncontended = true;
    //当前Cell存在，则执行CAS设置（9）
    else if (a.cas(v = a.value, ((fn == null) ? v + x :
                                 fn.applyAsLong(v, x))))
        break;
    //当前Cell数组元素个数大于CPU个数（10）
    else if (n >= NCPU || cells != as)
        collide = false;            // At max size or stale
    //是否有冲突（11）
    else if (!collide)
        collide = true;
    //如果当前元素个数没有达到CPU个数并且有冲突则扩容（12）
    else if (cellsBusy == 0 && casCellsBusy()) {
        try {
            if (cells == as) {      // Expand table unless stale
                //12.1
                Cell[] rs = new Cell[n << 1];
                for (int i = 0; i < n; ++i)
                    rs[i] = as[i];
                cells = rs;
            }
        } finally {
            //12.2
            cellsBusy = 0;
```

```
                }
                //12.3
                collide = false;
                continue;                    // Retry with expanded table
            }

            //(13)为了能够找到一个空闲的Cell，重新计算hash值,xorshift算法生成随机数
            h = advanceProbe(h);
        }
        //初始化Cell数组(14)
        else if (cellsBusy == 0 && cells == as && casCellsBusy()) {
            boolean init = false;
            try {
                if (cells == as) {
                    //14.1
                    Cell[] rs = new Cell[2];
                    //14.2
                    rs[h & 1] = new Cell(x);
                    cells = rs;
                    init = true;
                }
            } finally {
                //14.3
                cellsBusy = 0;
            }
            if (init)
                break;
        }
        else if (casBase(v = base, ((fn == null) ? v + x :
                                    fn.applyAsLong(v, x))))
            break;                           // Fall back on using base
    }
}
```

上面代码比较复杂，这里我们主要关注问题 3、问题 4 和问题 5。

当每个线程第一次执行到代码（6）时，会初始化当前线程变量 threadLocalRandomProbe 的值，上面也说了，这个变量在计算当前线程应该被分配到 cells 数组的哪一个 Cell 元素时会用到。

cells 数组的初始化是在代码（14）中进行的，其中 cellsBusy 是一个标示，为 0 说明当前 cells 数组没有在被初始化或者扩容，也没有在新建 Cell 元素，为 1 则说明 cells 数组

在被初始化或者扩容，或者当前在创建新的 Cell 元素、通过 CAS 操作来进行 0 或 1 状态的切换，这里使用 casCellsBusy 函数。假设当前线程通过 CAS 设置 cellsBusy 为 1，则当前线程开始初始化操作，那么这时候其他线程就不能进行扩容了。如代码（14.1）初始化 cells 数组元素个数为 2，然后使用 h&1 计算当前线程应该访问 celll 数组的哪个位置，也就是使用当前线程的 threadLocalRandomProbe 变量值 &（cells 数组元素个数 -1），然后标示 cells 数组已经被初始化，最后代码（14.3）重置了 cellsBusy 标记。显然这里没有使用 CAS 操作，却是线程安全的，原因是 cellsBusy 是 volatile 类型的，这保证了变量的内存可见性，另外此时其他地方的代码没有机会修改 cellsBusy 的值。在这里初始化的 cells 数组里面的两个元素的值目前还是 null。这里回答了问题 3，知道了 cells 数组如何被初始化。

cells 数组的扩容是在代码（12）中进行的，对 cells 扩容是有条件的，也就是代码（10）（11）的条件都不满足的时候。具体就是当前 cells 的元素个数小于当前机器 CPU 个数并且当前多个线程访问了 cells 中同一个元素，从而导致冲突使其中一个线程 CAS 失败时才会进行扩容操作。这里为何要涉及 CPU 个数呢？其实在基础篇中已经讲过，只有当每个 CPU 都运行一个线程时才会使多线程的效果最佳，也就是当 cells 数组元素个数与 CPU 个数一致时，每个 Cell 都使用一个 CPU 进行处理，这时性能才是最佳的。代码（12）中的扩容操作也是先通过 CAS 设置 cellsBusy 为 1，然后才能进行扩容。假设 CAS 成功则执行代码（12.1）将容量扩充为之前的 2 倍，并复制 Cell 元素到扩容后数组。另外，扩容后 cells 数组里面除了包含复制过来的元素外，还包含其他新元素，这些元素的值目前还是 null。这里回答了问题 4。

在代码（7）(8) 中，当前线程调用 add 方法并根据当前线程的随机数 threadLocalRandomProbe 和 cells 元素个数计算要访问的 Cell 元素下标，然后如果发现对应下标元素的值为 null，则新增一个 Cell 元素到 cells 数组，并且在将其添加到 cells 数组之前要竞争设置 cellsBusy 为 1。

代码（13）对 CAS 失败的线程重新计算当前线程的随机值 threadLocalRandomProbe，以减少下次访问 cells 元素时的冲突机会。这里回答了问题 5。

4.2.3　小结

本节介绍了 JDK 8 中新增的 LongAdder 原子性操作类，该类通过内部 cells 数组分担

了高并发下多线程同时对一个原子变量进行更新时的竞争量，让多个线程可以同时对 cells 数组里面的元素进行并行的更新操作。另外，数组元素 Cell 使用 @sun.misc.Contended 注解进行修饰，这避免了 cells 数组内多个原子变量被放入同一个缓存行，也就是避免了伪共享，这对性能也是一个提升。

4.3 LongAccumulator 类原理探究

LongAdder 类是 LongAccumulator 的一个特例，LongAccumulator 比 LongAdder 的功能更强大。例如下面的构造函数，其中 accumulatorFunction 是一个双目运算器接口，其根据输入的两个参数返回一个计算值，identity 则是 LongAccumulator 累加器的初始值。

```
    public LongAccumulator(LongBinaryOperator accumulatorFunction,
                           long identity) {
        this.function = accumulatorFunction;
        base = this.identity = identity;
    }
```

```
public interface LongBinaryOperator {

        //根据两个参数计算并返回一个值
        long applyAsLong(long left, long right);
}
```

上面提到，LongAdder 其实是 LongAccumulator 的一个特例，调用 LongAdder 就相当于使用下面的方式调用 LongAccumulator：

```
LongAdder adder = new LongAdder();
```

```
    LongAccumulator accumulator = new LongAccumulator(new LongBinaryOperator() {

        @Override
        public long applyAsLong(long left, long right) {
            return left + right;
        }
    }, 0);
```

LongAccumulator 相比于 LongAdder，可以为累加器提供非 0 的初始值，后者只能提供默认的 0 值。另外，前者还可以指定累加规则，比如不进行累加而进行相乘，只需要在构造 LongAccumulator 时传入自定义的双目运算器即可，后者则内置累加的规则。

从下面代码我们可以知道，LongAccumulator 相比于 LongAdder 的不同在于，在调用

casBase 时后者传递的是 b+x, 前者则使用了 r = function.applyAsLong(b = base, x) 来计算。

```
//LongAdder的add
public void add(long x) {
        Cell[] as; long b, v; int m; Cell a;
        if ((as = cells) != null || !casBase(b = base, b + x)) {
            boolean uncontended = true;
            if (as == null || (m = as.length - 1) < 0 ||
                (a = as[getProbe() & m]) == null ||
                !(uncontended = a.cas(v = a.value, v + x)))
                longAccumulate(x, null, uncontended);
        }
    }
//LongAccumulator的accumulate
    public void accumulate(long x) {
        Cell[] as; long b, v, r; int m; Cell a;
        if ((as = cells) != null ||
            (r = function.applyAsLong(b = base, x)) != b && !casBase(b, r)) {
            boolean uncontended = true;
            if (as == null || (m = as.length - 1) < 0 ||
                (a = as[getProbe() & m]) == null ||
                !(uncontended =
                  (r = function.applyAsLong(v = a.value, x)) == v ||
                  a.cas(v, r)))
                longAccumulate(x, function, uncontended);
        }
    }
```

另外，前者在调用 longAccumulate 时传递的是 function, 而后者是 null。从下面的代码可知，当 fn 为 null 时就使用 v+x 加法运算，这时候就等价于 LongAdder，当 fn 不为 null 时则使用传递的 fn 函数计算。

```
  else if (casBase(v = base, ((fn == null) ? v + x :
                                        fn.applyAsLong(v, x))))
                break;                          // Fall back on using base
```

总结：本节简单介绍了 LongAccumulator 的原理。LongAdder 类是 LongAccumulator 的一个特例，只是后者提供了更加强大的功能，可以让用户自定义累加规则。

4.4 总结

本章介绍了并发包中的原子性操作类，这些类都是使用非阻塞算法 CAS 实现的，这相比使用锁实现原子性操作在性能上有很大提高。首先讲解了最简单的 AtomicLong 类的实现原理，然后讲解了 JDK 8 中新增的 LongAdder 类和 LongAccumulator 类的原理。学习完本章后，希望读者在实际项目环境中能因地制宜地使用原子性操作类来提升系统性能。

第5章

Java并发包中并发List源码剖析

5.1 介绍

并发包中的并发 List 只有 CopyOnWriteArrayList。CopyOnWriteArrayList 是一个线程安全的 ArrayList，对其进行的修改操作都是在底层的一个复制的数组（快照）上进行的，也就是使用了写时复制策略。Copy On WriteArraylist 的类图结构如图 5-1 所示。

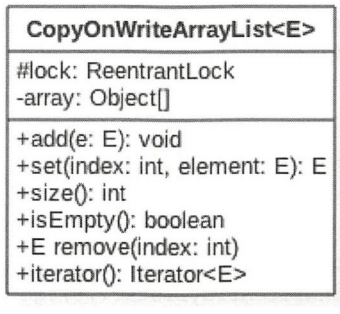


图 5-1

在 CopyOnWriteArrayList 的类图中，每个 CopyOnWriteArrayList 对象里面有一个 array 数组对象用来存放具体元素，ReentrantLock 独占锁对象用来保证同时只有一个线程对 array 进行修改。这里只要记得 ReentrantLock 是独占锁，同时只有一个线程可以获取就可以了，后面会专门对 JUC 中的锁进行介绍。

如果让我们自己做一个写时复制的线程安全的 list 我们会怎么做，有哪些点需要考虑？

- 何时初始化 list，初始化的 list 元素个数为多少，list 是有限大小吗？

- 如何保证线程安全，比如多个线程进行读写时如何保证是线程安全的？
- 如何保证使用迭代器遍历 list 时的数据一致性？

下面我们看看 CopyOnWriteArrayList 的作者 Doug Lea 是如何设计的。

5.2 主要方法源码解析

5.2.1 初始化

首先看下无参构造函数，如下代码在内部创建了一个大小为 0 的 Object 数组作为 array 的初始值。

```
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}
```

然后看下有参构造函数。

```
//创建一个list，其内部元素是入参toCopyIn的副本
public CopyOnWriteArrayList(E[] toCopyIn) {
    setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, Object[].class));
}

//入参为集合，将集合里面的元素复制到本list
public CopyOnWriteArrayList(Collection<? extends E> c) {
    Object[] elements;
    if (c.getClass() == CopyOnWriteArrayList.class)
        elements = ((CopyOnWriteArrayList<?>)c).getArray();
    else {
        elements = c.toArray();
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elements.getClass() != Object[].class)
            elements = Arrays.copyOf(elements, elements.length, Object[].class);
    }
    setArray(elements);
}
```

5.2.2 添加元素

CopyOnWriteArrayList 中用来添加元素的函数有 add(E e)、add(int index, E element)、

addIfAbsent(E e) 和 addAllAbsent(Collection<? extends E> c) 等，它们的原理类似，所以本节以 add(E e) 为例来讲解。

```
public boolean add(E e) {

        //获取独占锁（1）
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            //(2)获取array
            Object[] elements = getArray();

            //(3)复制array到新数组，添加元素到新数组
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            newElements[len] = e;

            //(4)使用新数组替换添加前的数组
            setArray(newElements);
            return true;
        } finally {
            //(5)释放独占锁
            lock.unlock();
        }
}
```

在如上代码中，调用 add 方法的线程会首先执行代码（1）去获取独占锁，如果多个线程都调用 add 方法则只有一个线程会获取到该锁，其他线程会被阻塞挂起直到锁被释放。

所以一个线程获取到锁后，就保证了在该线程添加元素的过程中其他线程不会对 array 进行修改。

线程获取锁后执行代码（2）获取 array，然后执行代码（3）复制 array 到一个新数组（从这里可以知道新数组的大小是原来数组大小增加 1，所以 CopyOnWriteArrayList 是无界 list），并把新增的元素添加到新数组。

然后执行代码（4）使用新数组替换原数组，并在返回前释放锁。由于加了锁，所以整个 add 过程是个原子性操作。需要注意的是，在添加元素时，首先复制了一个快照，然后在快照上进行添加，而不是直接在原来数组上进行。

5.2.3 获取指定位置元素

使用E get(int index)获取下标为index的元素，如果元素不存在则抛出IndexOutOfBoundsException异常。

```
public E get(int index) {
    return get(getArray(), index);
}

final Object[] getArray() {
    return array;
}

private E get(Object[] a, int index) {
    return (E) a[index];
}
```

在如上代码中，当线程x调用get方法获取指定位置的元素时，分两步走，首先获取array数组（这里命名为步骤A），然后通过下标访问指定位置的元素（这里命名为步骤B），这是两步操作，但是在整个过程中并没有进行加锁同步。假设这时候List内容如图5-2所示，里面有1、2、3三个元素。

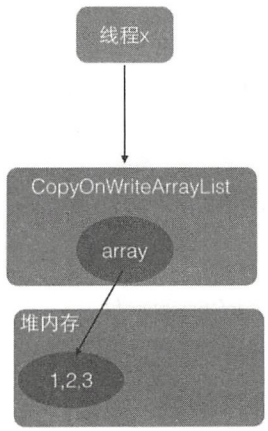


图5-2

由于执行步骤A和步骤B没有加锁，这就可能导致在线程x执行完步骤A后执行步骤B前，另外一个线程y进行了remove操作，假设要删除元素1。remove操作首先会获取独占锁，然后进行写时复制操作，也就是复制一份当前array数组，然后在复制的数组

里面删除线程 x 通过 get 方法要访问的元素 1，之后让 array 指向复制的数组。而这时候 array 之前指向的数组的引用计数为 1 而不是 0，因为线程 x 还在使用它，这时线程 x 开始执行步骤 B，步骤 B 操作的数组是线程 y 删除元素之前的数组，如图 5-3 所示。

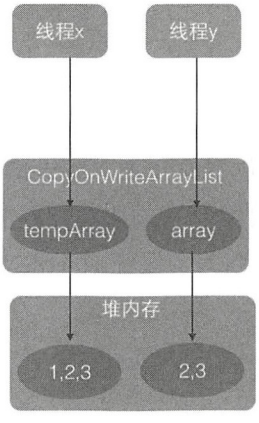


图 5-3

所以，虽然线程 y 已经删除了 index 处的元素，但是线程 x 的步骤 B 还是会返回 index 处的元素，这其实就是写时复制策略产生的弱一致性问题。

5.2.4 修改指定元素

使用 E set（int index,E element）修改 list 中指定元素的值，如果指定位置的元素不存在则抛出 IndexOutOfBoundsException 异常，代码如下。

```
public E set(int index, E element) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            E oldValue = get(elements, index);

            if (oldValue != element) {
                int len = elements.length;
                Object[] newElements = Arrays.copyOf(elements, len);
                newElements[index] = element;
                setArray(newElements);
            } else {
```

```
                // Not quite a no-op; ensures volatile write semantics
                setArray(elements);
            }
            return oldValue;
        } finally {
            lock.unlock();
        }
    }
```

如上代码首先获取了独占锁，从而阻止其他线程对 array 数组进行修改，然后获取当前数组，并调用 get 方法获取指定位置的元素，如果指定位置的元素值与新值不一致则创建新数组并复制元素，然后在新数组上修改指定位置的元素值并设置新数组到 array。如果指定位置的元素值与新值一样，则为了保证 volatile 语义，还是需要重新设置 array，虽然 array 的内容并没有改变。

5.2.5 删除元素

删除 list 里面指定的元素 , 可以使用 E remove(int index) 、boolean remove(Object o) 和 boolean remove(Object o, Object[] snapshot, int index) 等方法，它们的原理一样。下面讲解下 remove(int index) 方法。

```
public E remove(int index) {

        //获取独占锁
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {

            //获取数组
            Object[] elements = getArray();
            int len = elements.length;

            //获取指定元素
            E oldValue = get(elements, index);
            int numMoved = len - index - 1;

            //如果要删除的是最后一个元素
            if (numMoved == 0)
                setArray(Arrays.copyOf(elements, len - 1));
            else {
                //分两次复制删除后剩余的元素到新数组
```

```
                Object[] newElements = new Object[len - 1];
                System.arraycopy(elements, 0, newElements, 0, index);
                System.arraycopy(elements, index + 1, newElements, index,
                                 numMoved);
                //使用新数组代替老数组
                setArray(newElements);
            }
            return oldValue;
        } finally {
            //释放锁
            lock.unlock();
        }
    }
```

如上代码其实和新增元素的代码类似，首先获取独占锁以保证删除数据期间其他线程不能对 array 进行修改，然后获取数组中要被删除的元素，并把剩余的元素复制到新数组，之后使用新数组替换原来的数组，最后在返回前释放锁。

5.2.6 弱一致性的迭代器

遍历列表元素可以使用迭代器。在讲解什么是迭代器的弱一致性前，先举一个例子来说明如何使用迭代器。

```
public static void main( String[] args )
    {
       CopyOnWriteArrayList<String> arrayList = new CopyOnWriteArrayList<>();
       arrayList.add("hello");
       arrayList.add("alibaba");

       Iterator<String> itr = arrayList.iterator();
       while(itr.hasNext()){
           System.out.println(itr.next());
       }

}
```

输出如下。

```
<terminated> copylist [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
hello
alibaba
```

迭代器的 hasNext 方法用于判断列表中是否还有元素，next 方法则具体返回元素。好了，

下面来看 CopyOnWriteArrayList 中迭代器的弱一致性是怎么回事，所谓弱一致性是指返回迭代器后，其他线程对 list 的增删改对迭代器是不可见的，下面看看这是如何做到的。

```
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}

static final class COWIterator<E> implements ListIterator<E> {
    //array的快照版本
    private final Object[] snapshot;

    //数组下标
    private int cursor;

    //构造函数
    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }

    //是否遍历结束
    public boolean hasNext() {
        return cursor < snapshot.length;
    }

    //获取元素
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }
```

在如上代码中，当调用 iterator() 方法获取迭代器时实际上会返回一个 COWIterator 对象，COWIterator 对象的 snapshot 变量保存了当前 list 的内容，cursor 是遍历 list 时数据的下标。

为什么说 snapshot 是 list 的快照呢？明明是指针传递的引用啊，而不是副本。如果在该线程使用返回的迭代器遍历元素的过程中，其他线程没有对 list 进行增删改，那么 snapshot 本身就是 list 的 array，因为它们是引用关系。但是如果在遍历期间其他线程对该 list 进行了增删改，那么 snapshot 就是快照了，因为增删改后 list 里面的数组被新数组替换了，这时候老数组被 snapshot 引用。这也说明获取迭代器后，使用该迭代器元素时，其他线程

对该 list 进行的增删改不可见，因为它们操作的是两个不同的数组，这就是弱一致性。

下面通过一个例子来演示多线程下迭代器的弱一致性的效果。

```
public class copylist
{
    private static volatile CopyOnWriteArrayList<String> arrayList = new
CopyOnWriteArrayList<>();

    public static void main( String[] args ) throws InterruptedException
    {
       arrayList.add("hello");
       arrayList.add("alibaba");
       arrayList.add("welcome");
       arrayList.add("to");
       arrayList.add("hangzhou");

       Thread threadOne = new Thread(new Runnable() {

            @Override
            public void run() {

                 //修改list中下标为1的元素为baba
                 arrayList.set(1, "baba");
                 //删除元素
                 arrayList.remove(2);
                 arrayList.remove(3);

            }
        });

       //保证在修改线程启动前获取迭代器
       Iterator<String> itr = arrayList.iterator();

       //启动线程
       threadOne.start();

       //等待子线程执行完毕
          threadOne.join();

         //迭代元素
       while(itr.hasNext()){
           System.out.println(itr.next());
       }
```

```
    }
}
```

输出结果如下。

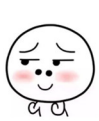
```
<terminated> copylist [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
hello
alibaba
welcome
to
hangzhou
```

在如上代码中，main 函数首先初始化了 arrayList，然后在启动线程前获取到了 arrayList 迭代器。子线程 threadOne 启动后首先修改了 arrayList 的第一个元素的值，然后删除了 arrayList 中下标为 2 和 3 的元素。

主线程在子线程执行完毕后使用获取的迭代器遍历数组元素，从输出结果我们知道，在子线程里面进行的操作一个都没有生效，这就是迭代器弱一致性的体现。需要注意的是，获取迭代器的操作必须在子线程操作之前进行。

5.3 总结

CopyOnWriteArrayList 使用写时复制的策略来保证 list 的一致性，而获取—修改—写入三步操作并不是原子性的，所以在增删改的过程中都使用了独占锁，来保证在某个时间只有一个线程能对 list 数组进行修改。另外 CopyOnWriteArrayList 提供了弱一致性的迭代器，从而保证在获取迭代器后，其他线程对 list 的修改是不可见的，迭代器遍历的数组是一个快照。另外，CopyOnWriteArraySet 的底层就是使用它实现的，感兴趣的读者可以查阅相关源码。



第6章

Java并发包中锁原理剖析

6.1 LockSupport 工具类

JDK 中的 rt.jar 包里面的 LockSupport 是个工具类，它的主要作用是挂起和唤醒线程，该工具类是创建锁和其他同步类的基础。

LockSupport 类与每个使用它的线程都会关联一个许可证，在默认情况下调用 LockSupport 类的方法的线程是不持有许可证的。LockSupport 是使用 Unsafe 类实现的，下面介绍 LockSupport 中的几个主要函数。

1. void park() 方法

如果调用 park 方法的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 时会马上返回，否则调用线程会被禁止参与线程的调度，也就是会被阻塞挂起。

如下代码直接在 main 函数里面调用 park 方法，最终只会输出 begin park!，然后当前线程被挂起，这是因为在默认情况下调用线程是不持有许可证的。

```
public static void main( String[] args )
{
    System.out.println( "begin park!" );

    LockSupport.park();

    System.out.println( "end park!" );
```

```
    }
```

在其他线程调用 unpark(Thread thread) 方法并且将当前线程作为参数时，调用 park 方法而被阻塞的线程会返回。另外，如果其他线程调用了阻塞线程的 interrupt() 方法，设置了中断标志或者线程被虚假唤醒，则阻塞线程也会返回。所以在调用 park 方法时最好也使用循环条件判断方式。

需要注意的是，因调用 park() 方法而被阻塞的线程被其他线程中断而返回时并不会抛出 InterruptedException 异常。

2. void unpark(Thread thread) 方法

当一个线程调用 unpark 时，如果参数 thread 线程没有持有 thread 与 LockSupport 类关联的许可证，则让 thread 线程持有。如果 thread 之前因调用 park() 而被挂起，则调用 unpark 后，该线程会被唤醒。如果 thread 之前没有调用 park，则调用 unpark 方法后，再调用 park 方法，其会立刻返回。修改代码如下。

```
public static void main( String[] args )
    {
        System.out.println( "begin park!" );

        //使当前线程获取到许可证
        LockSupport.unpark(Thread.currentThread());

        //再次调用park方法
        LockSupport.park();

        System.out.println( "end park!" );

    }
```

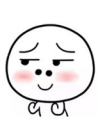
该代码会输出

```
begin park!
end park!
```

下面再来看一个例子以加深对 park 和 unpark 的理解。

```
public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Runnable() {

            @Override
```



```
            public void run() {

                System.out.println("child thread begin park!");

                // 调用park方法，挂起自己
                LockSupport.park();

                System.out.println("child thread unpark!");

            }
        });

        //启动子线程
        thread.start();

        //主线程休眠1s
        Thread.sleep(1000);

        System.out.println("main thread begin unpark!");

        //调用unpark方法让thread线程持有许可证，然后park方法返回
        LockSupport.unpark(thread);

    }
```

输出结果为

```
child thread begin park!
main thread begin unpark!
child thread unpark!
```

上面代码首先创建了一个子线程 thread, 子线程启动后调用 park 方法，由于在默认情况下子线程没有持有许可证，因而它会把自己挂起。

主线程休眠 1s 是为了让主线程调用 unpark 方法前让子线程输出 child thread begin park! 并阻塞。

主线程然后执行 unpark 方法，参数为子线程，这样做的目的是让子线程持有许可证，然后子线程调用的 park 方法就返回了。

park 方法返回时不会告诉你因何种原因返回，所以调用者需要根据之前调用 park 方法的原因，再次检查条件是否满足，如果不满足则还需要再次调用 park 方法。

例如，根据调用前后中断状态的对比就可以判断是不是因为被中断才返回的。

为了说明调用 park 方法后的线程被中断后会返回，我们修改上面的例子代码，删除 LockSupport.unpark(thread);，然后添加 thread.interrupt();，具体代码如下。

```
public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(new Runnable() {

        @Override
        public void run() {

            System.out.println("child thread begin park!");

            // 调用park方法，挂起自己,只有被中断才会退出循环
            while (!Thread.currentThread().isInterrupted()) {
                LockSupport.park();

            }

            System.out.println("child thread unpark!");

        }
    });

    // 启动子线程
    thread.start();

    // 主线程休眠1s
    Thread.sleep(1000);

    System.out.println("main thread begin unpark!");

    // 中断子线程
    thread.interrupt();

}
```

输出结果为

```
child thread begin park!
main thread begin unpark!
child thread unpark!
```

在如上代码中，只有中断子线程，子线程才会运行结束，如果子线程不被中断，即使

你调用 unpark(thread) 方法子线程也不会结束。

3. void parkNanos(long nanos) 方法

和 park 方法类似，如果调用 park 方法的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.parkNanos(long nanos) 方法后会马上返回。该方法的不同在于，如果没有拿到许可证，则调用线程会被挂起 nanos 时间后修改为自动返回。

另外 park 方法还支持带有 blocker 参数的方法 void park(Object blocker) 方法，当线程在没有持有许可证的情况下调用 park 方法而被阻塞挂起时，这个 blocker 对象会被记录到该线程内部。

使用诊断工具可以观察线程被阻塞的原因，诊断工具是通过调用 getBlocker(Thread) 方法来获取 blocker 对象的，所以 JDK 推荐我们使用带有 blocker 参数的 park 方法，并且 blocker 被设置为 this，这样当在打印线程堆栈排查问题时就能知道是哪个类被阻塞了。

例如下面的代码。

```
public class TestPark {

    public  void testPark(){
       LockSupport.park();//(1)

    }
    public static void main(String[] args) {

        TestPark testPark = new TestPark();
        testPark.testPark();

    }

}
```

运行代码后，使用 jstack pid 命令查看线程堆栈时可以看到如下输出结果。

```
"main" prio=5 tid=0x00007feba2802800 nid=0xd03 waiting on condition [0x000000010946a000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:315)
```

修改代码（1）为 LockSupport.park(this) 后运行代码，则 jstack pid 的输出结果为

```
"main" prio=5 tid=0x00007fe844001800 nid=0xd03 waiting on condition [0x000000010b942000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00000007d5666d90> (a com.zlx.park.TestPark)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
```

使用带 blocker 参数的 park 方法，线程堆栈可以提供更多有关阻塞对象的信息。

4. park(Object blocker) 方法

```
public static void park(Object blocker) {
   //获取调用线程
    Thread t = Thread.currentThread();

   //设置该线程的blocker变量
    setBlocker(t, blocker);

    //挂起线程
    UNSAFE.park(false, 0L);

   //线程被激活后清除blocker变量，因为一般都是在线程阻塞时才分析原因
    setBlocker(t, null);
}
```

Thread 类里面有个变量 volatile Object parkBlocker，用来存放 park 方法传递的 blocker 对象，也就是把 blocker 变量存放到了调用 park 方法的线程的成员变量里面。

5. void parkNanos(Object blocker, long nanos) 方法

相比 park(Object blocker) 方法多了个超时时间。

6. void parkUntil(Object blocker, long deadline) 方法

它的代码如下：

```
public static void parkUntil(Object blocker, long deadline) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);
   //isAbsolute=true,time=deadline;表示到deadline时间后返回
    UNSAFE.park(true, deadline);
    setBlocker(t, null);
}
```

其中参数 deadline 的时间单位为 ms，该时间是从 1970 年到现在某一个时间点的毫秒值。这个方法和 parkNanos(Object blocker, long nanos) 方法的区别是，后者是从当前算等

待 nanos 秒时间，而前者是指定一个时间点，比如需要等到 2017.12.11 日 12:00:00，则把这个时间点转换为从 1970 年到这个时间点的总毫秒数。

最后再看一个例子。

```
class FIFOMutex {
    private final AtomicBoolean locked = new AtomicBoolean(false);
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
        boolean wasInterrupted = false;
        Thread current = Thread.currentThread();
        waiters.add(current);

        // 只有队首的线程可以获取锁（1）
        while (waiters.peek() != current || !locked.compareAndSet(false, true)) {
            LockSupport.park(this);
            if (Thread.interrupted()) // （2）
                wasInterrupted = true;
        }

        waiters.remove();
        if (wasInterrupted) // （3）
            current.interrupt();
    }

    public void unlock() {
        locked.set(false);
        LockSupport.unpark(waiters.peek());
    }
}
```

这是一个先进先出的锁，也就是只有队列的首元素可以获取锁。在代码（1）处，如果当前线程不是队首或者当前锁已经被其他线程获取，则调用 park 方法挂起自己。

然后在代码（2）处判断，如果 park 方法是因为被中断而返回，则忽略中断，并且重置中断标志，做个标记，然后再次判断当前线程是不是队首元素或者当前锁是否已经被其他线程获取，如果是则继续调用 park 方法挂起自己。

然后在代码（3）中，判断标记，如果标记为 true 则中断该线程，这个怎么理解呢？其实就是其他线程中断了该线程，虽然我对中断信号不感兴趣，忽略它，但是不代表其他线程对该标志不感兴趣，所以要恢复下。