Java 并发编程之美：并发编程高级篇之二

一、前言

Java 并发编程实践中的话：

编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却还都离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本场 Chat 作为 Java 并发编程之美系列的高级篇之二，主要讲解内容如下：（建议先阅读 Java 并发编程之美：基础篇 ）

- rt.jar 中 Unsafe 类主要函数讲解， Unsafe 类提供了硬件级别的原子操作，可以安全的直接操作内存变量，其在 JUC 源码中被广泛的使用，了解其原理为研究 JUC 源码奠定了基础。
- rt.jar 中 LockSupport 类主要函数讲解，LockSupport 是个工具类，主要作用是挂起和唤醒线程，是创建锁和其它同步类的基础，了解其原理为研究 JUC 中锁的实现奠定基础。
- 讲解 JDK8 新增原子操作类 LongAdder 实现原理，并讲解 AtomicLong 的缺点是什么，LongAdder 是如何解决 AtomicLong 的缺点的，LongAdder 和 LongAccumulator 是什么关系？
- JUC 并发包中并发组件 CopyOnWriteArrayList 的实现原理，CopyOnWriteArrayList 是如何通过写时拷贝实现并发安全的 List？

二、 Unsafe 类探究

JDK 的 rt.jar 包中的 Unsafe 类提供了硬件级别的原子操作，Unsafe 里面的方法都是 native 方法，通过使用 JNI 的方式来访问本地 C++ 实现库。下面我们看下 Unsafe 提供的几个主要方法以及编程时候如何使用 Unsafe 类做一些事情。

2.1 主要方法介绍

- long objectFieldOffset(Field field) 方法

作用：返回指定的变量在所属类的内存偏移地址，偏移地址仅仅在该 Unsafe 函数中访问指定字段时候使用。如下代码使用 unsafe 获取AtomicLong 中变量 value 在 AtomicLong 对象中的内存偏移

```
static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicLong.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
}
```

- int arrayBaseOffset(Class arrayClass) 方法
  获取数组中第一个元素的地址

- int arrayIndexScale(Class arrayClass) 方法
  获取数组中单个元素占用的字节数

- boolean compareAndSwapLong(Object obj, long offset, long expect, long update) 方法
  比较对象 obj 中偏移量为 offset 的变量的值是不是和 expect 相等，相等则使用 update 值更新，然后返回 true，否者返回 false

- public native long getLongVolatile(Object obj, long offset) 方法
  获取对象 obj 中偏移量为 offset 的变量对应的 volatile 内存语义的值。

- void putLongVolatile(Object obj, long offset, long value) 方法
  设置 obj 对象中内存偏移为 offset 的 long 型变量的值为 value，支持 volatile 内存语义。

- void putOrderedLong(Object obj, long offset, long value) 方法
  设置 obj 对象中 offset 偏移地址对应的 long 型 field 的值为 value。这是有延迟的 putLongVolatile 方法，并不保证值修改对其它线程立刻可见。变量只有使用 volatile 修饰并且期望被意外修改的时候使用才有用。

- void park(boolean isAbsolute, long time)
  阻塞当前线程，其中参数 isAbsolute 等于 false 时候，time 等于 0 表示一直阻塞，time 大于 0 表示等待指定的 time 后阻塞线程会被唤醒，这个 time 是个相对值，是个增量值，也就是相对当前时间累加 time 后当前线程就会被唤醒。
  如果 isAbsolute 等于 true，并且 time 大于 0 表示阻塞后到指定的时间点后会被唤醒，这里 time 是个绝对的时间，是某一个时间点换算为 ms 后的值。另外当其它线程调用了当前阻塞线程的 interrupt 方法中断了当前线程时候，当前线程也会返回，当其它线程调用了 unpark 方法并且把当前线程作为参数时候当前线程也会返回。

- void unpark(Object thread)
  唤醒调用 park 后阻塞的线程，参数为需要唤醒的线程。

下面是 Jdk8 新增的方法，这里简单的列出 Long 类型操作的方法

- long getAndSetLong(Object obj, long offset, long update) 方法
  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量 volatile 语义的值为 update。

```
public final long getAndSetLong(Object obj, long offset, long
update)
  {
    long l;
    do
    {
      l = getLongVolatile(obj, offset);//(1)
    } while (!compareAndSwapLong(obj, offset, l, update));
    return l;
  }
```

从代码可知内部代码 (1) 处使用 getLongVolatile 获取当前变量的值，然后使用 CAS 原子操作进行设置新值，这里使用 while 循环是考虑到多个线程同时调用的情况 CAS 失败后需要自旋重试。

- long getAndAddLong(Object obj, long offset, long addValue) 方法
  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量值为原始值 +addValue。

```
public final long getAndAddLong(Object obj, long offset, long
addValue)
  {
    long l;
    do
    {
      l = getLongVolatile(obj, offset);
    } while (!compareAndSwapLong(obj, offset, l, l + addValue));
    return l;
  }
```

类似 getAndSetLong 的实现，只是这里使用CAS的时候使用了原始值+传递的增量参数 addValue 的值。

2.2 如何使用 Unsafe 类

看到 Unsafe 这个类如此牛叉，你肯定会忍不住撸下下面代码，期望能够使用 Unsafe 做点事情。

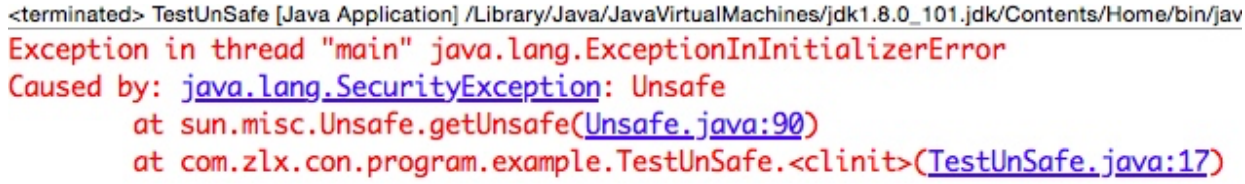
```
public class TestUnSafe {

    //获取Unsafe的实例（2.2.1）
    static final Unsafe unsafe = Unsafe.getUnsafe();

    //记录变量state在类TestUnSafe中的偏移值（2.2.2）
    static final long stateOffset;

    //变量(2.2.3)
    private volatile long state=0;

    static {

        try {
            //获取state变量在类TestUnSafe中的偏移值(2.2.4)
            stateOffset =
unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state
"));

        } catch (Exception ex) {

            System.out.println(ex.getLocalizedMessage());
            throw new Error(ex);
        }

    }

    public static void main(String[] args) {

        //创建实例，并且设置state值为1(2.2.5)
        TestUnSafe test = new TestUnSafe();
        //(2.2.6)
        Boolean sucess = unsafe.compareAndSwapInt(test,
stateOffset, 0, 1);
        System.out.println(sucess);

    }
}
```

如上代码（2.2.1）获取了 Unsafe 的一个实例，代码（2.2.3）创建了一个变量 state 初始化为 0。

代码（2.2.4）使用 unsafe.objectFieldOffset 获取 TestUnSafe 类里面的 state 变量在 TestUnSafe 对象里面的内存偏移量地址并保存到 stateOffset 变量。

代码（2.2.6）调用创建的 unsafe 实例的 compareAndSwapInt 方法，设置 test 对象的 state 变量的值，具体意思是如果 test 对象内存偏移量为 stateOffset 的 state 的变量为 0，则更新该值为 1。

运行上面代码我们期望会输出 true，然而执行后会输出如下结果：



为研究其原因，必然要翻看 getUnsafe 代码，看看里面做了啥：

```
    private static final Unsafe theUnsafe = new Unsafe();

     public static Unsafe getUnsafe()
    {
        //（2.2.7）
        Class localClass = Reflection.getCallerClass();

       //（2.2.8）
        if (!VM.isSystemDomainLoader(localClass.getClassLoader())) {
          throw new SecurityException("Unsafe");
        }
        return theUnsafe;
    }

      //判断paramClassLoader是不是BootStrap类加载器(2.2.9)
     public static boolean isSystemDomainLoader(ClassLoader
    paramClassLoader)
      {
        return paramClassLoader == null;
      }
```

代码（2.2.7）获取调用 getUnsafe 这个方法的对象的 Class 对象，这里是 TestUnSafe.class。

代码（2.2.8）判断是不是 Bootstrap 类加载器加载的 localClass，这里是看是不是 Bootstrap 加载器加载了 TestUnSafe.class。很明显由于 TestUnSafe.class 是使用 AppClassLoader 加载的（可以参考 chat http://gitbook.cn/gitchat/activity/5a751b1391d6b7067048a213），所以这里直接抛出了异常。

思考下，这里为何要有这个判断那?

我们知道 Unsafe 类是在 rt.jar 里面提供的，而 rt.jar 里面的类是使用 Bootstrap 类加载器加载的，而我们启动 main 函数所在的类是使用 AppClassLoader 加载的，所以在 main 函数里面加载 Unsafe 类时候鉴于委托机制会委托给 Bootstrap 去加载 Unsafe 类。

如果没有代码（2.2.8）这鉴权，那么我们应用程序就可以随意使用 Unsafe 做事情了，而 Unsafe 类可以直接操作内存，是不安全的，所以 JDK 开发组特意做了这个限制，不让开发人员在正规渠道下使用 Unsafe 类，而是在 rt.jar 里面的核心类里面使用 Unsafe 功能。

那么如果开发人员真的想要实例化 Unsafe 类，使用 Unsafe 的功能该如何做那?

方法有很多种，既然正规渠道访问不了，那么就玩点黑科技，使用万能的反射来获取 Unsafe 实例方法：

```
public class TestUnSafe {

    static final Unsafe unsafe;

    static final long stateOffset;

    private volatile long state = 0;

    static {

        try {

            // 反射获取 Unsafe 的成员变量 theUnsafe（2.2.10）
            Field field =
Unsafe.class.getDeclaredField("theUnsafe");

            // 设置为可存取（2.2.11）
            field.setAccessible(true);

            // 获取该变量的值（2.2.12）
            unsafe = (Unsafe) field.get(null);

            //获取 state 在 TestUnSafe 中的偏移量 （2.2.13）
            stateOffset =
unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state
"));

        } catch (Exception ex) {

            System.out.println(ex.getLocalizedMessage());
            throw new Error(ex);
        }

    }

    public static void main(String[] args) {

        TestUnSafe test = new TestUnSafe();
        Boolean sucess = unsafe.compareAndSwapInt(test,
stateOffset, 0, 1);
        System.out.println(sucess);

    }
}
```

如上代码通过代码（2.2.10），（2.2.11），（2.2.12）反射获取 unsafe 的实例，然后运行结果输出：

三、LockSupport类探究

JDK 中的 rt.jar 里面的 LockSupport 是个工具类，主要作用是挂起和唤醒线程，它是创建锁和其它同步类的基础。

LockSupport 类与每个使用它的线程都会关联一个许可证,默认调用LockSupport 类的方法的线程是不持有许可证的，LockSupport 内部使用 Unsafe 类实现，下面介绍下 LockSupport 内的几个主要函数：

- void park() 方法
  如果调用 park() 的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 会马上返回，否者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，最终结果只会输出 `begin park!`，然后当前线程会被挂起，这是因为默认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!" );

}
```

在其它线程调用 `unpark(Thread thread)` 方法并且当前线程作为参数时候，调用park 方法被阻塞的线程会返回，另外其它线程调用了阻塞线程的 interrupt() 方法，设置了中断标志时候或者由于线程的虚假唤醒原因后阻塞线程也会返回，所以调用 park() 最好也是用循环条件判断方式（关于虚假唤醒可以参考 chat http://gitbook.cn/gitchat/activity/5aa4d205c2ff6f2e120891dd）。

需要注意的是调用 park() 方法被阻塞的线程被其他线程中断后阻塞线程返回时候并不会抛出 InterruptedException 异常。

- void unpark(Thread thread) 方法
  当一个线程调用了 unpark 时候，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 之前调用了 park() 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，在调用 park() 方法，会立刻返回，上面代码修改如下：

```
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
        LockSupport.park();

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

    }
```

则会输出：
begin park!
end park!

下面再来看一个例子来加深对 park,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让thread线程持有许可证，然后park方法会返回
        LockSupport.unpark(thread);

    }
```

输出为：
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，启动后子线程调用 park 方法，由于默认子线程没有持有许可证，会把自己挂起。

主线程休眠 1s 为的是主线程在调用 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，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 方法返回时候不会告诉你是因为何种原因返回，所以调用者需要根据之前是处于什么目前调用的 park 方法，再次检查条件是否满足，如果不满足的话还需要再次调用 park 方法。

例如，线程在返回时的中断状态，根据调用前后中断状态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，修改上面例子代码，删除 LockSupport.unpark(thread); 然后添加 thread.interrupt(); 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!");

        // 中断子线程线程
        thread.interrupt();

    }
```

输出为：
child thread begin park!
main thread begin unpark!
child thread unpark!

如上代码也就是只有当子线程被中断后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) 子线程也不会结束。

- void parkNanos(long nanos)函数

和 park 类似，如果调用 park 的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 会马上返回，不同在于如果没有拿到许可调用线程会被挂起 nanos 时间后在返回。

park 还支持三个带有 blocker 参数的方法，当线程因为没有持有许可的情况下调用 park 被阻塞挂起时候，这个 blocker 对象会被记录到该线程内部。

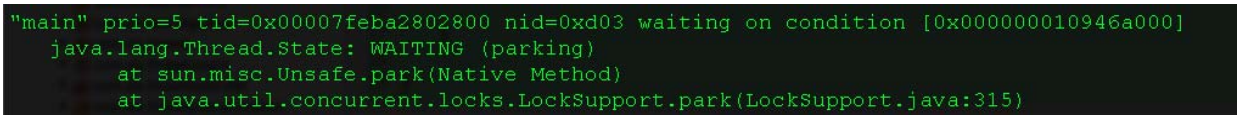
使用诊断工具可以观察线程被阻塞的原因，诊断工具是通过调 getBlocker(Thread) 方法来获取该 blocker 对象的，所以 JDK 推荐我们使用带有 blocker 参数的 park 方法,并且 blocker 设置为 this，这样当内存 dump 排查问题时候就能知道是那个类被阻塞了。

例如下面代码：

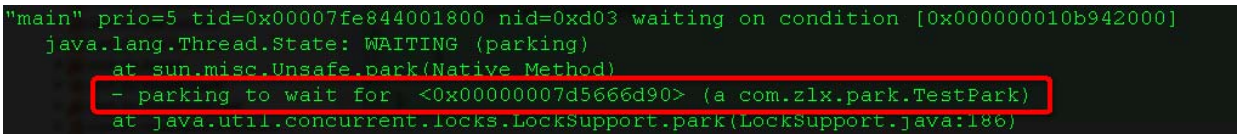
```
public class TestPark {

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

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

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

```
        }

    }
```

运行后使用 jstack pid 查看线程堆栈时候可以看到如下：



修改 代码（1）为 LockSupport.park(this) 后运行在 jstack pid 结果为：



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

- park(Object blocker) 函数

```
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，因为一般都是线程阻塞时候才分析原因
     setBlocker(t, null);
}
```

Thread 类里面有个变量 volatile Object parkBlocker 用来存放 park 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。

- void parkNanos(Object blocker, long nanos) 函数
  相比 park(Object blocker) 多了个超时时间。
- void parkUntil(Object blocker, long deadline)
  parkUntil 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 时间时候后返
回
        UNSAFE.park(true, deadline);
```

```
            setBlocker(t, null);
        }
```

可知是设置一个 deadline，时间单位为 milliseconds，是从 1970 到现在某一个时间点换算为毫秒后的值，这个和 parkNanos(Object blocker, long nanos) 区别是后者是从当前算等待 nanos 时间，而前者是指定一个时间点，比如我需要等待到 2017.12.11 日 12:00:00，则吧这个时间点转换为从 1970 年到这个时间点的总毫秒数。

最后在看一个例子

```
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，也就是只有队列首元素可以获取锁，代码（1）处如果当前线程不是队首或者当前锁已经被其它线程获取，则调用park方法挂起自己。

然后代码（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，只做个标记，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它线程获取，如果是则继续调用 park 方法挂起自己。

然后代码（3）中如果标记为 true 则中断该线程，这个怎么理解那？其实意思是其它线程中断了该线程，虽然我对中断信号不感兴趣，忽略它，但是不代表其它线程对该标志不感兴趣，所以要恢复下。

四、 LongAdder 和 LongAccumulator 原理探究

4.1 LongAdder 原理

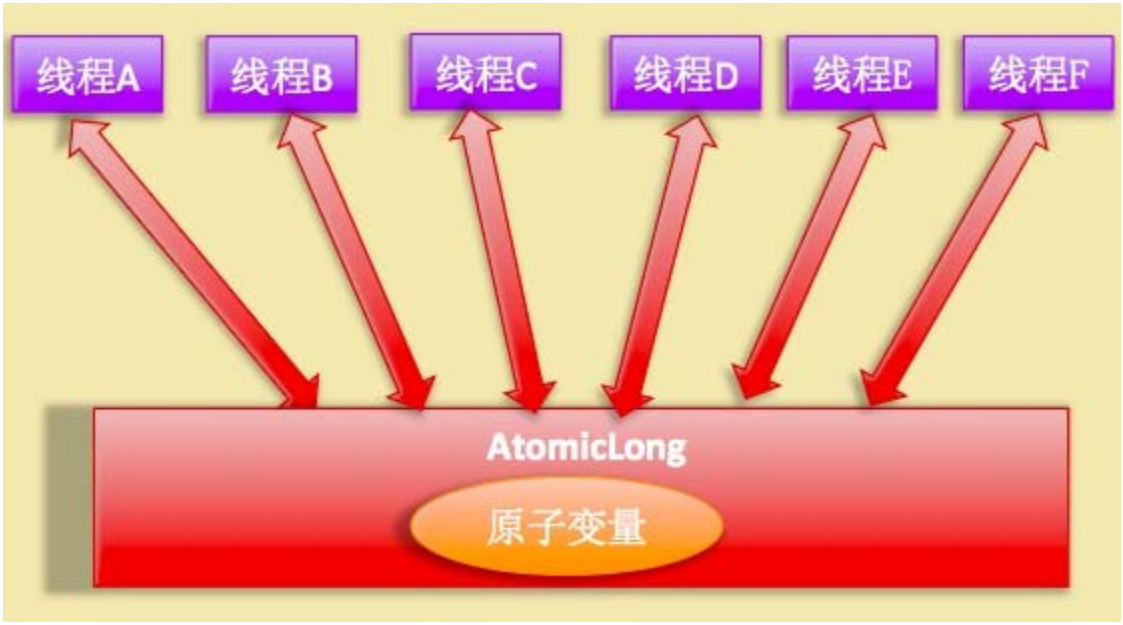
LongAdder 类是 JDK8 新增的一个原子性操作类。AtomicLong 通过 CAS 算法提供了非阻塞的原子性操作，相比使用阻塞算法的同步器来说性能已经很好了，但是 JDK 开发组并不满足于此，因为在非常高的并发请求下 AtomicLong 的性能不能让它们接受。（关于 CAS 操作可以参考 chat http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f）

如下 AtomicLong 的 incrementAndGet 的代码，虽然 AtomicLong 使用 CAS 算法，但是 CAS 失败后还是通过无限循环的自旋锁不断尝试的：

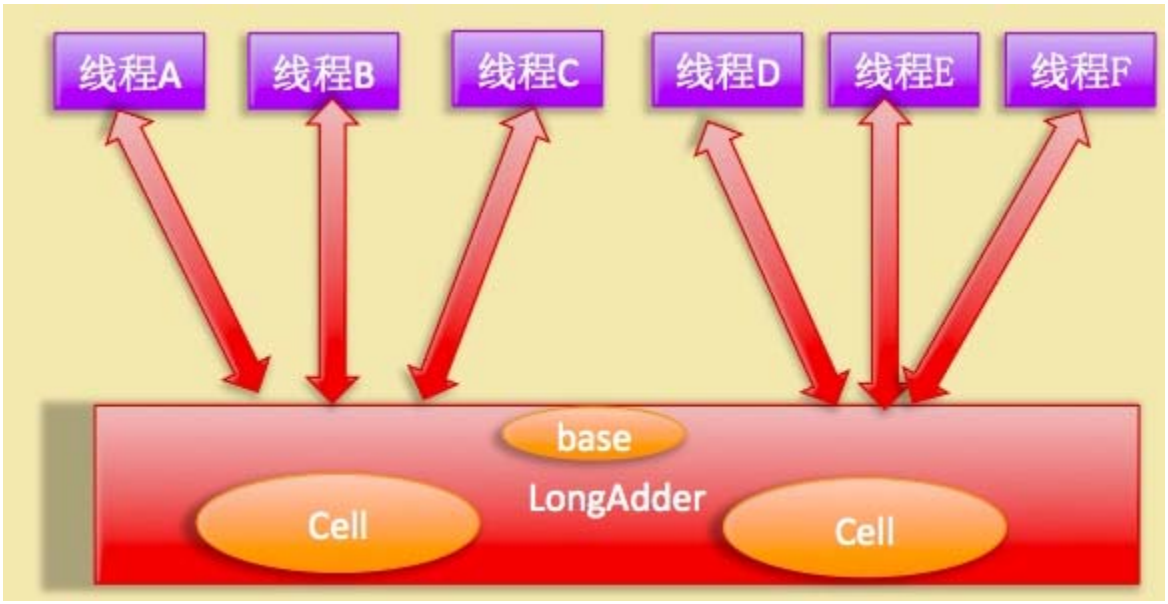
```
public final long incrementAndGet() {
    for (;;) {
        long current = get();
        long next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

在高并发下 N 多线程同时去操作一个变量会造成大量线程 CAS 失败然后处于自旋状态，这大大浪费了 cpu 资源，降低了并发性。

那么既然 AtomicLong 性能问题是由于过多线程同时去竞争同一个变量的更新而降低的，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那么性能问题不就解决了？是的，JDK 8 提供的 LongAdder 就是这个思路。下面通过图形来标示两者不同。



如上图 AtomicLong 是多个线程同时竞争同一个变量情景。



如上图 LongAdder 则是内部维护多个 Cell 变量，每个 Cell 里面有一个初始值为 0 的 long 型变量，在同等并发量的情况下，争夺单个变量的线程量会减少，这是变相的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原子变量时候如果失败并不是自旋 CAS 重试，而是尝试获取其它原子变量的锁，最后获取当前值时候是把所有变量的值累加后在加上 base 返回的。

LongAdder 维护了一个延迟初始化的原子性更新数组和一个基值变量base. 数组的大小保持是 2 的 N 次方大小，数组表的下标使用每个线程的 hashcode 值的掩码表示，数组里面的变量实体是 Cell 类型。

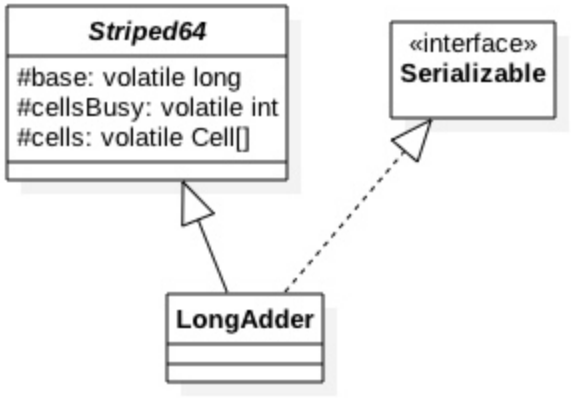
Cell 类型是 AtomicLong 的一个改进，用来减少缓存的争用，对于大多数原子操作字节填充是浪费的，因为原子性操作都是无规律的分散在内存中进行的，多个原子性操作彼此之间是没有接触的，但是原子性数组元素彼此相邻存放将能经常共享缓存行也就是伪共享（关于伪共享，可以参考 Chat http://gitbook.cn/gitchat/activity/5aafb17477918b6e8444b65f），所以这在性能上是一个提升。

另外由于 Cells 占用内存是相对比较大的，所以一开始并不创建，而是在需要时候在创建，也就是惰性加载，当一开始没有空间时候，所有的更新都是操作base变量，

4.1.1 LongAdder 代码简单分析

上节我们讲解了 LongAdder 的原理，这节简单介绍下代码实现（详细实现读者可以翻看代码研究），为了降低高并发下多线程对一个变量 CAS 争夺失败后大量线程会自旋而造成降低并发性能问题，LongAdder 内部通过根据并发请求量维护多个 Cell 元素（一个动态的 Cell 数组）来分担对单个变量进行争夺的并发量。

首先看下 LongAdder 的构造



如图可知 LongAdder 继承自 Striped64 类，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元素值的累加，base 是个基础值默认是 0，cellsBusy 用来实现自旋锁，当创建 Cell 元素或者扩容 Cell 数组时候用来进行线程间的同步。

下面看看 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);
```

```
            }
        }
    }
```

如上代码可知 Cell 的构造很简单，内部维护一个声明为 volatile 的变量，这里声明为 volatile 是因为线程操作 value 变量时候没有使用锁，为了保证变量的内存可见性这里只有声明为 volatile（关于共享变量的内存可见性问题可以参考 chat http://gitbook.cn/gitchat/activity/5aa4d205c2ff6f2e120891dd）。另外还是老生常谈的使用 UNsafe 类的方法来设置 value 的值。

- long sum()
  返回当前的值，内部操作是累加所有 Cell 内部的 value 的值后累加 base，如下代码，由于计算总和时候没有对 Cell 数组进行加锁，所以在累加过程中可能有其它线程对 Cell 中的值进行了修改，也有可能数组进行了扩容，所以 sum 返回的值并不是非常精确的，返回值并不是一个调用 sum 方法时候的一个原子快照值。

```
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()
  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()
  sum 的改造版本，如下代码，在计算 sum 累加对应的 cell 值后，把当前 cell 的值重置为 0，base 重置为 0。
  当多线程调用该方法时候会有问题，比如考虑第一个调用线程会清空 Cell 的值，后

一个线程调用时候累加时候累加的都是 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()

- void add(long x)
  累加增量 x 到原子变量，这个过程是原子性的

```
    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);
    }
```

如上代码当第一个线程 A 执行 add 时候，代码（1）会执行 casBase 方法通过 cas 设置 base 为 x，如果成功则直接返回，这时候base的值为 1。

假如多个线程同时执行 add 时候，同时执行到 casBase 则只有一个线程A成功返回其它线程由于cas失败执行代码（2），代码（2）是获取 cells 数组的长度，如果数组长度为0则执行（5），否者 cells 长度不为 0说明 cells 数组有元素则执行代码（3），代码（3）首先计算当前线程在数组中下标，然后获取当前线程对应的cell值，如果获取到则执行（4）进行 cas 操作，cas 失败则否者执行（5）。代码（5）里面是具体进行数组扩充和

初始化的，这个代码比较杂，本文只讲解原理，不深入代码细节，感兴趣的读者可以去研究下。

4.2 LongAccumulator 类原理探究

LongAdder 类是 LongAccumulator 的一个特例，LongAccumulator 提供了比 LongAdder 更强大的功能，如下构造函数其中 accumulatorFunction 一个双目运算器接口，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 相当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 时候传入自定义双面运算器就 OK，后者则内置累加的规则。

从下面代码知道 LongAccumulator 相比于 LongAdder 不同在于 casBase 时候后者传递的是 b+x，前者则是调用了 r = function.applyAsLong(b = base, x) 来计算。

```
    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);
        }
    }

    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);
        }
    }
```

另外前者调用 longAccumulate 时候传递到是 function，而后者是 null，从下面代码可知当 fn 为 null时候就是使用 v+x 加法运算这时候就等价于 LongAdder，fn 不为 null 时候则使用传递的 fn 函数计算，如果 fn 为加法则等价于 LongAdder；

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

四、JUC 并发包中 CopyOnWriteArrayList 原理探究

4.1 介绍

并发包中并发 List 只有 CopyOnWriteArrayList 这一个，CopyOnWriteArrayList 是一个线程安全的 ArrayList，对其进行的修改操作和元素迭代操作都是在底层创建一个拷贝的数组（快照）上进行的，也就是写时拷贝策略。

![image.png](http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/3bc6a42010c8cebb990bc3f4ccf1a254.png)

如上 CopyOnWriteArrayList 的类图，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，后面的 chat 会专门对 JUC 中锁进行介绍的。

考虑如果让我们自己做一个写时拷贝的线程安全的list我们会怎么做，有哪些点需要考虑那？

- list 何时初始化，初始化 list 元素个数为多少，list 是有限大小？
- 如何保证线程安全，比如多个线程进行读写时候如何保证是线程安全的
- 如何保证使用迭代器遍历list时候的数据一致性

下面就看看 CopyOnWriteArrayList 的作者 Doug Lea 是如何设计的。

4.2 主要函数源码讲解

4.2.1 初始化

首先看下无参构造，如下代码内部创建了一个大小为 0 的 Object 数据作为 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的拷贝
    public CopyOnWriteArrayList(E[] toCopyIn) {
        setArray(Arrays.copyOf(toCopyIn, toCopyIn.length,
Object[].class));
    }



//入参为集合，拷贝集合里面元素到本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);
        }
```

4.2.2 添加元素

CopyOnWriteArrayList 中添加元素函数有 `add(E e)`, `add(int index, E element)`, `addIfAbsent(E e)`, `addAllAbsent(Collection<? extends E> c)` 等操作，原理一致，所以本节以 `add(E e)` 为例来讲解。

```
    public boolean add(E e) {

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

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

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

如上代码，调用add方法的线程会首先执行（1）去获取独占锁，如果多个线程都调用add则只有一个线程会获取该锁，其它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过程中其它线程不会对array进行修改。

线程获取锁后执行（2）获取array，然后执行（3）拷贝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),并把要新增的元素添加到新数组。

然后执行到（4）把新数组替换元数组，然后返回前释放锁，由于加了锁，所以整个 add 过程是个原子性操作，需要注意的是添加元素时候首先是拷贝了一个快照，然后在快照上进行的添加，而不是直接在原来数组上进行。

4.2.3 获取指定位置元素

- E get(int index)
  获取下标为 index 的元素，如果元素不存在会抛出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];
    }
```

如上代码获取指定位置的元素分为两步，首先获取到当前 list 里面的 array 数组这里称为步骤 A，然后通过随机访问的下标方式访问指定位置的元素这里称为步骤 B。

从代码可以看到整个过程中并没有加锁，这就可能会导致当执行完步骤A后执行步骤 B 前，另外一个线程 C 进行了修改操作比如 remove 操作，就会进行写时拷贝删除当前 get 方法要访问的元素，并且修改当前 list 的 array 为新数组。而这之后步骤 B 可能才开始执行，步骤 B 操作的是线程 C 删除元素前的一个快照数组（因为步骤 A 让 array 指向的是原来的数组），所以虽然线程 C 已经删除了 index 处的元素，但是步骤 B 还是返回 index 出的元素，这其实就是写时拷贝策略带来的弱一致性。

4.2.4 修改指定元素

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控制了其它线程对 array 数组的修改，然后获取当前数组，并调用 get 方法获取指定位置元素。

如果指定位置元素与新值不一致则创建新数组并拷贝元素，在新数组上修改指定位置元素值并设置新数组到 array。

如果指定位置元素与新值一样则为了保证 volatile 语义还是需要重新设置下 array，虽然 array 内容并没有改变（为了保证 volatile 语义是考虑到 set 方法本身应该提供 volatile 的语义）。

4.2.5 删除元素

删除 list 里面指定的元素，主要方法如下：

- E remove(int index)
- boolean remove(Object o)
- boolean remove(Object o, Object[] snapshot, int index)
  等方法，原理一致，这里讲解下 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 {
                //分两次拷贝除删除后的元素到新数组
                Object[] newElements = new Object[len - 1];
                System.arraycopy(elements, 0, newElements, 0,
index);
                System.arraycopy(elements, index + 1,
newElements, index,
                                 numMoved);
                //使用新数组代替老的
                setArray(newElements);
            }
            return oldValue;
        } finally {
            //释放锁
            lock.unlock();
        }
    }
```

如上代码其实和新增元素时候类似，首先获取独占锁保证删除数据期间其它线程不能对array进行修改，然后获取数据中要给删除的元素，并把剩余的原始拷贝到新数组后把新数组替换原来的数组，最后在返回前释放锁。

4.2.6 弱一致性的迭代器

遍历列表元素可以使用迭代器进行迭代操作，讲解什么是迭代器的弱一致性前先上一个例子说明下迭代器的使用。

```
 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：

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

其中迭代器的 hasNext 方法用来判断是否还有元素，next 方法则是具体返回元素。好了，下面来看 CopyOnWriteArrayList 中迭代器是弱一致性，所谓弱一致性是指当返回迭代器后，其它线程对 list 的增删改对迭代器不可见，无感知，下面看看是如何做到的。

```
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++];
    }
```

如上代码当调用 iterator() 方法获取迭代器时候实际是返回一个 COWIterator 对象，COWIterator的snapshot 变量保存了当前list的内容，cursor 是遍历 list 数据的下标。

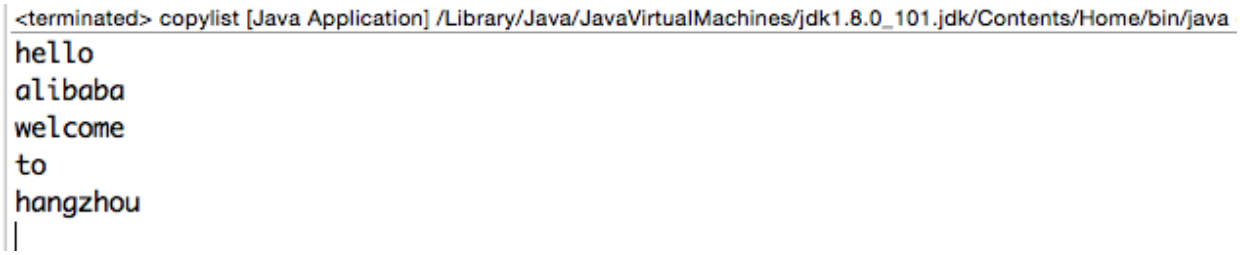
这里为什么说 snapshot 是 list 的快照那，明明是指针传递的引用哇，而不是拷贝。如果在该线程使用返回的迭代器遍历元素的过程中，其它线程没有对list进行增删改，那么 snapshot 本身就是 list 的 array，因为它们是引用关系。

但是如果在遍历期间其它线程对该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，这时候老数组只有被snapshot所引用，所以这也说明获取迭代器后，使用该迭代器进行变量元素时候，其它线程对该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，这也就是弱一致性的达成。

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();

       //等在子线程执行完毕
          threadOne.join();

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

    }
}
```

输出结果：



如上代码 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2，3 的元素。

主线程等子线程执行完毕后使用获取的迭代器遍历数组元素，从打印结果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效果，需要注意的是获取迭代器必须在子线程操作之前进行。

注：CopyOnWriteArrayList 使用写时拷贝的策略来保证list的一致性，而获取-拷贝-写入三步并不是原子性的，所以在修改增删改的过程中都使用了独占锁，保证了同时只有一个线程才能对list数组进行修改。另外CopyOnWriteArrayList提供了弱一致性的迭代器，保证在获取迭代器后，其它线程对list的修改不可见，迭代器遍历时候的数组是获取迭代器时候的一个快照，另外并发包中 CopyOnWriteArraySet 底层就是使用它进行实现，感兴趣的可以去翻翻看。

五、总结

本文首先讲解了rt.jar 中 Unsafe和 LockSupport 类主要函数，这两个类是JUC的基础，为研究 JUC 源码的实现奠定基础。然后讲解 JDK8 新增原子操作类 LongAdder 实现原理，并讲解了它如何解决 AtomicLong 的缺点的，LongAdder 和 LongAccumulator 是什么关系？最后讲解了JUC 并发包中并发组件CopyOnWriteArrayList 是如何通过写时拷贝实现并发安全的 List，何为弱一致性。