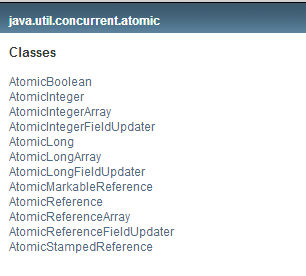
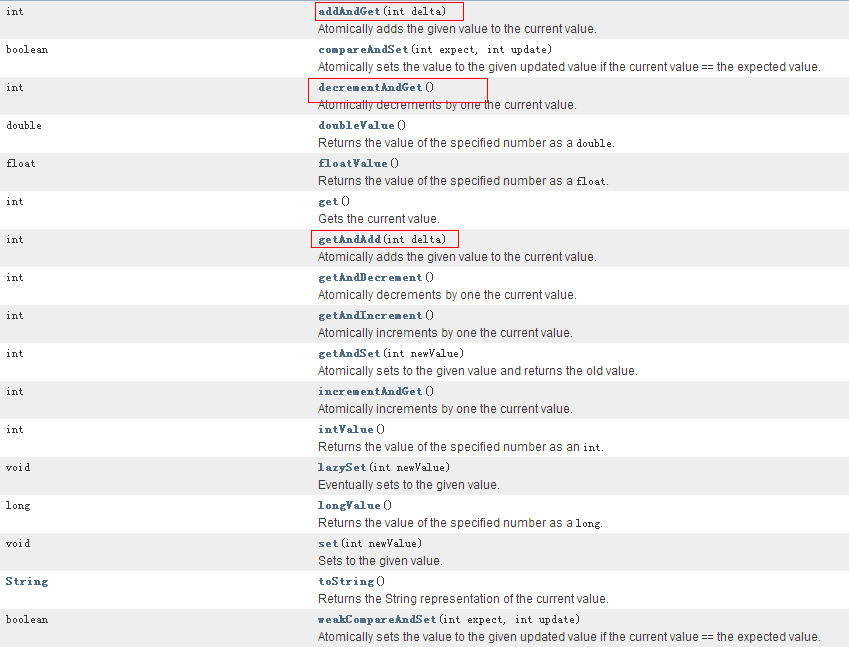
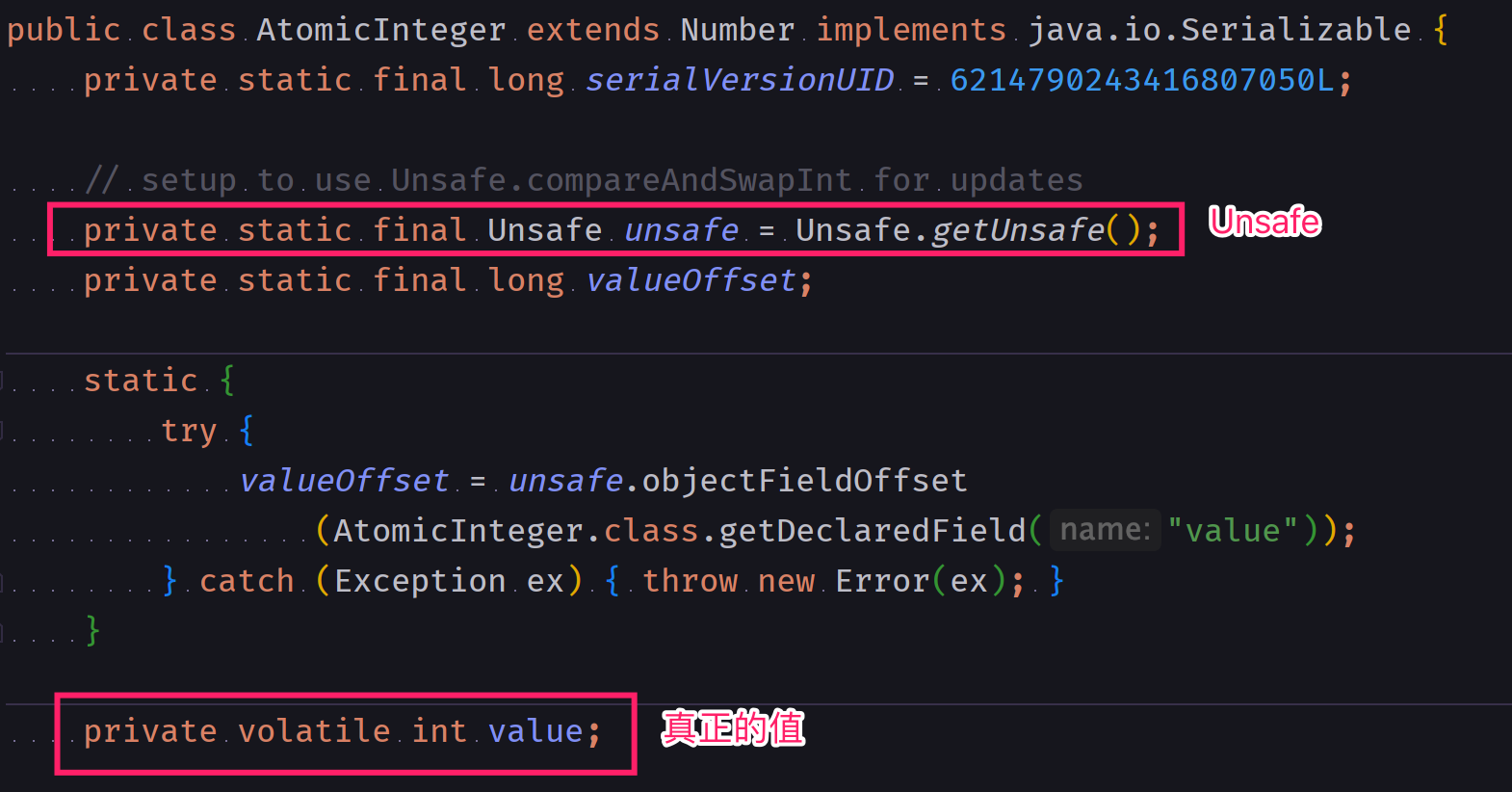
1. 需要强调的是，Atomic类被设计用来构建java.util.concurrent中的类，因此只有在特殊情况下才需要在自己的代码中使用它们，通常依赖于锁更安全一些。
2. 原子类在java.util.concurrent.atomic包中，包含如下的原子类



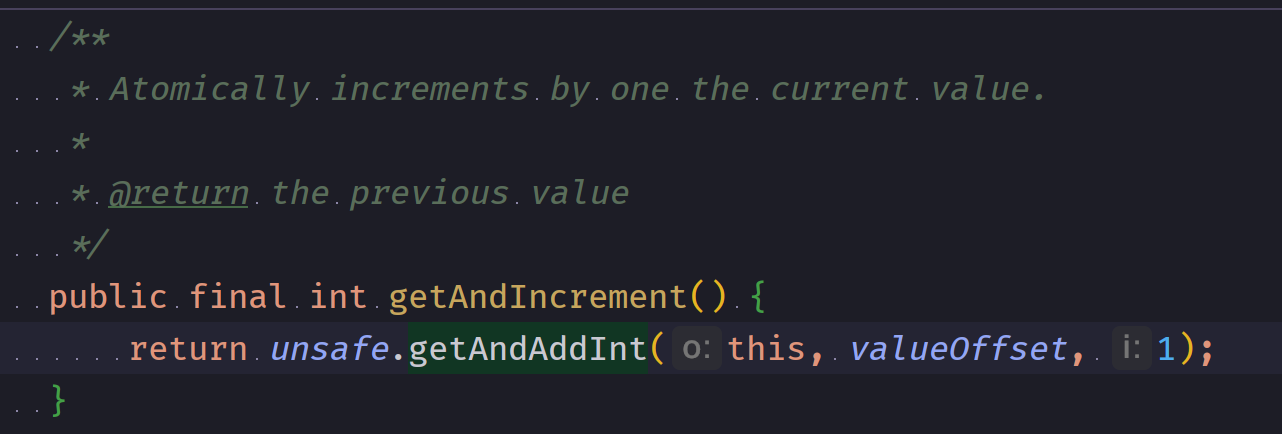
以AtomicInteger为例，其中典型的方法有

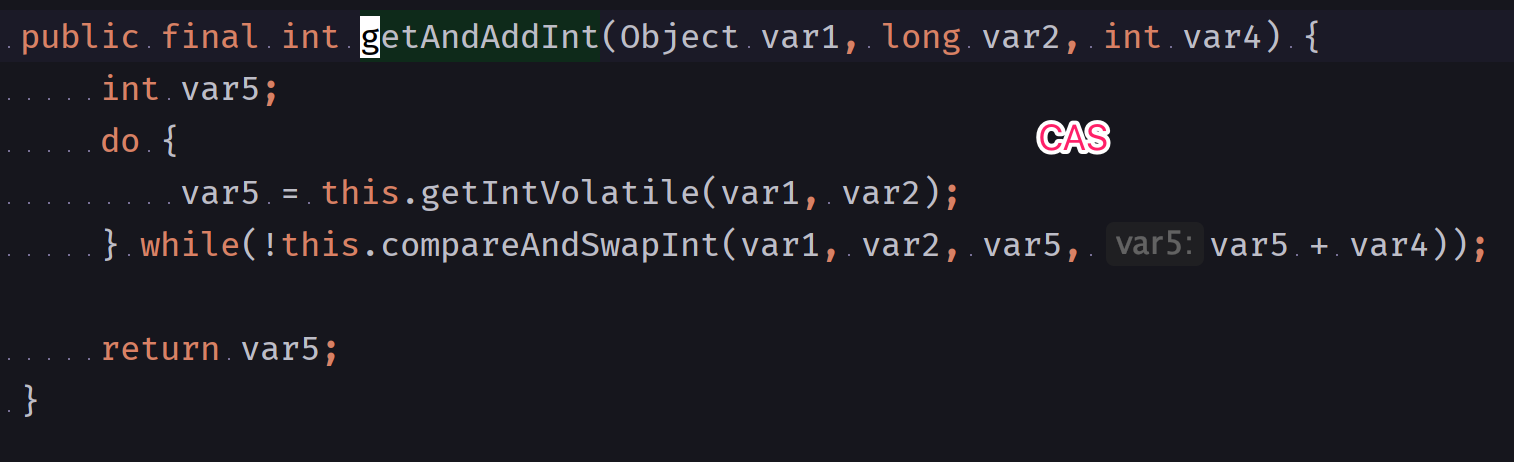


1. 这些类都是使用非阻塞算法CAS实现的，相比锁实现在性能上有很大提高，以AtomicInteger为例说明其实现，其内部使用Unsafe实现.



1. 主要方法实现
   1. java.util.concurrent.atomic.AtomicInteger#getAndDecrement

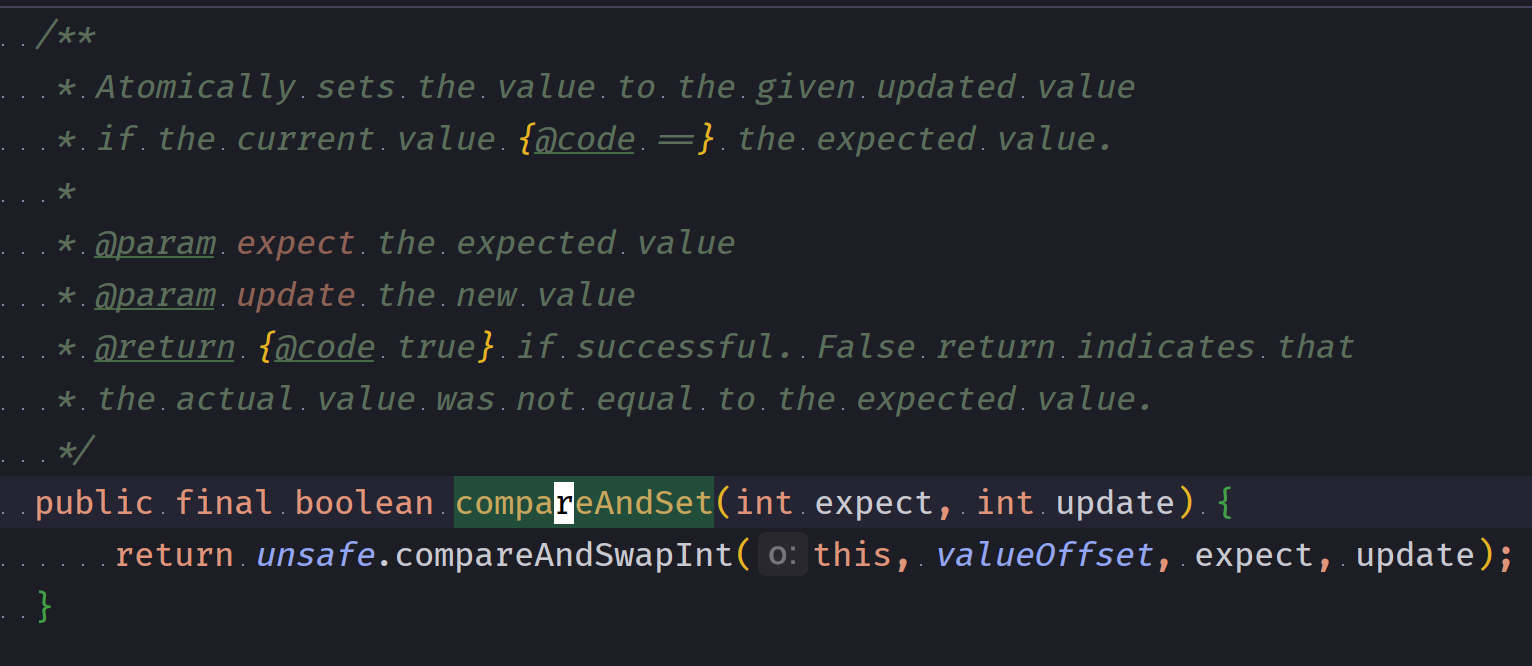


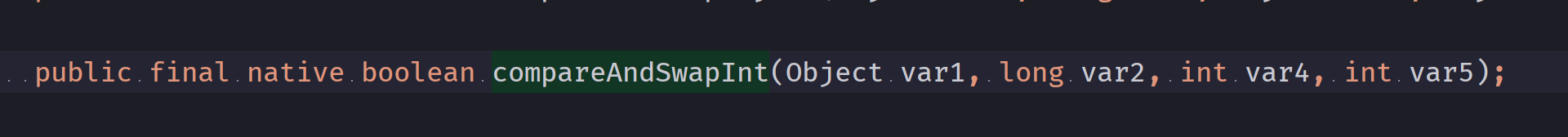


* 1. java.util.concurrent.atomic.AtomicInteger#incrementAndGet

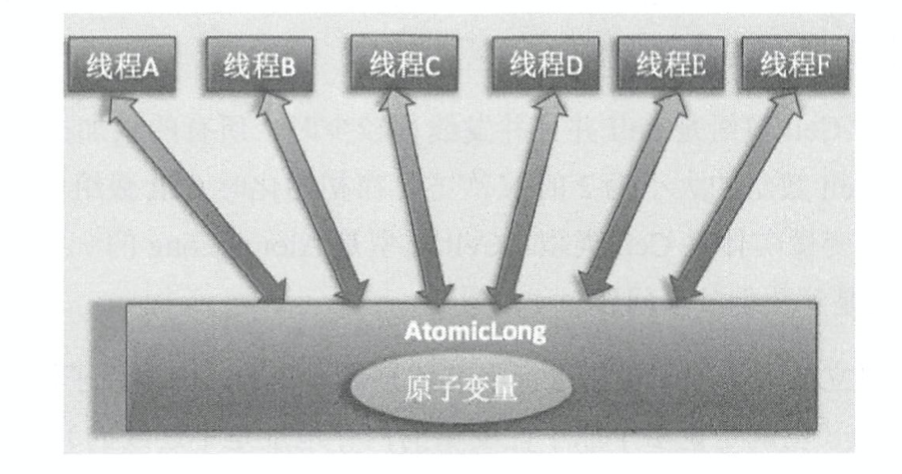


* 1. java.util.concurrent.atomic.AtomicInteger#compareAndSet

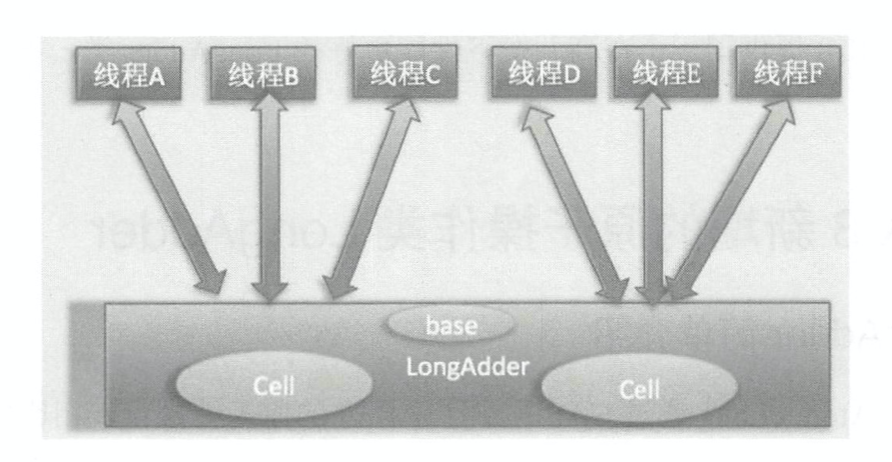




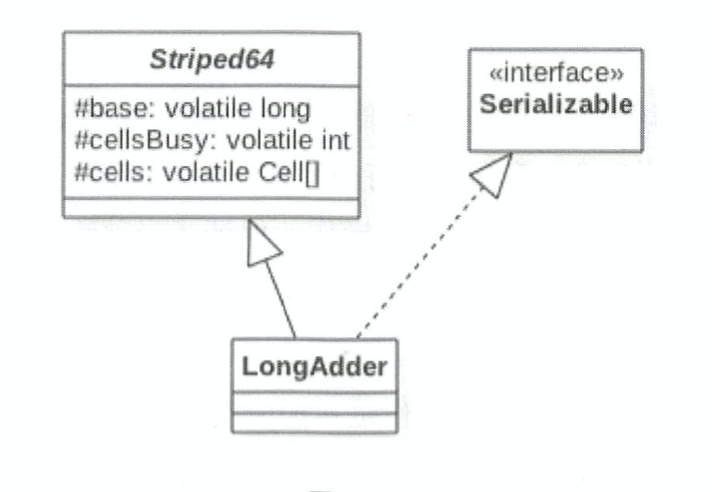
1. 在高并发下，使用AtomicInteger、AtomicLong实现计数器还是会存在性能问题，JDK8提供了高并发下场景下性能更好的java.util.concurrent.atomic.**LongAdder**类
2. AtomicLong的性能瓶颈是由于过多线程同时去竞争一个变量的更新而产生，如下图



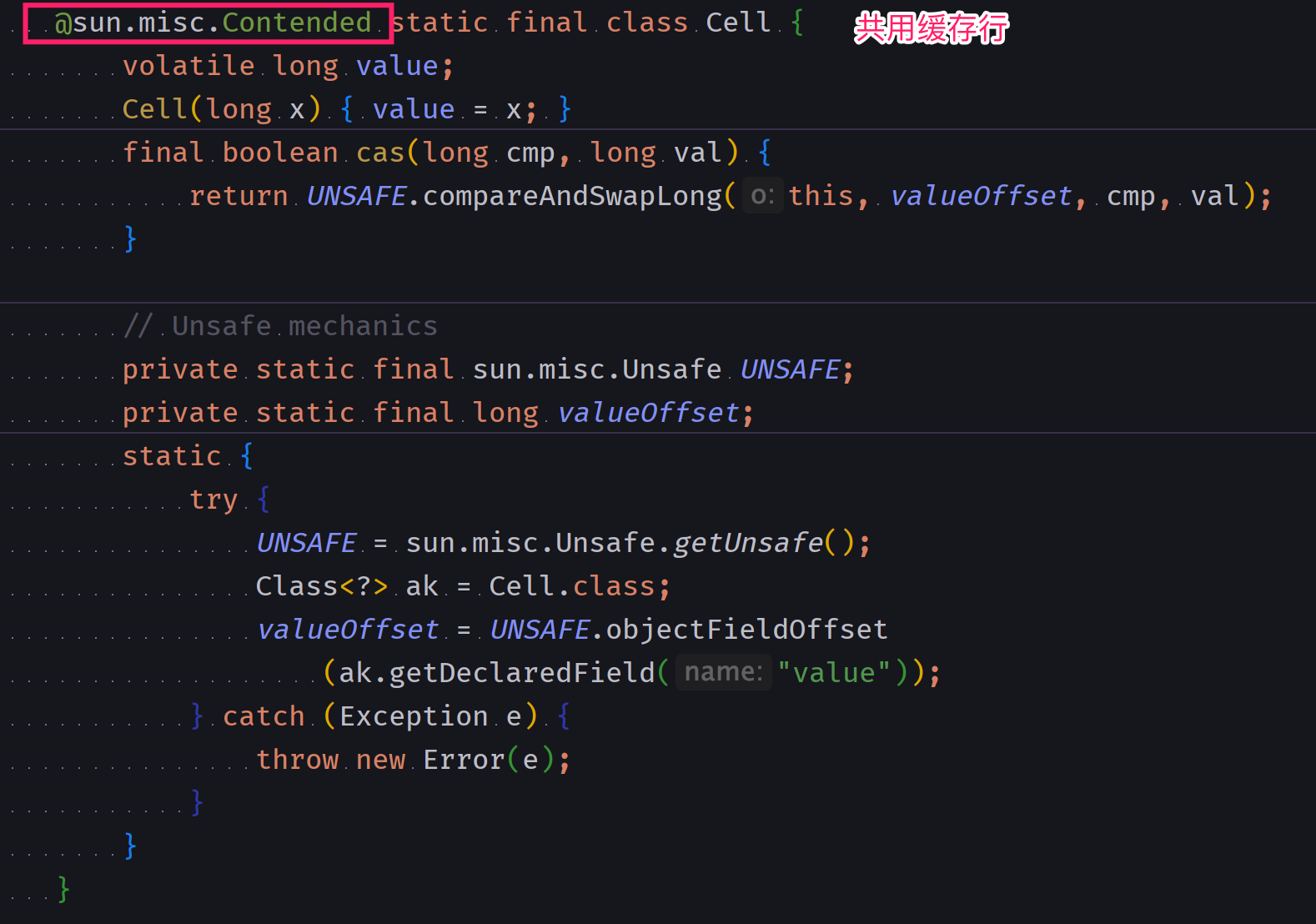
LongAdder把一个变量分解为多个变量，让同样多的线程去竞争多个资源，如下图



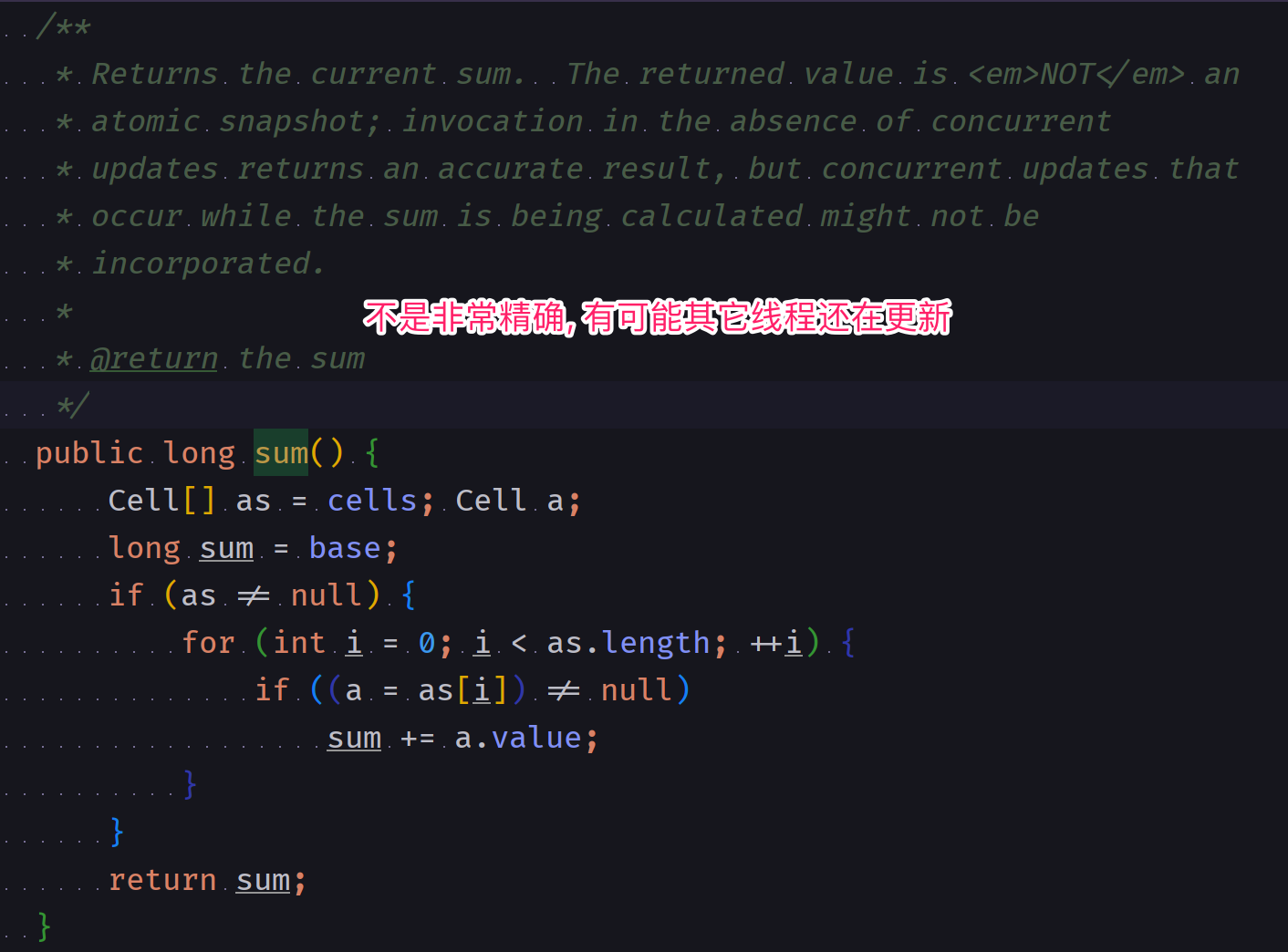
* 1. LongAdder类图结构



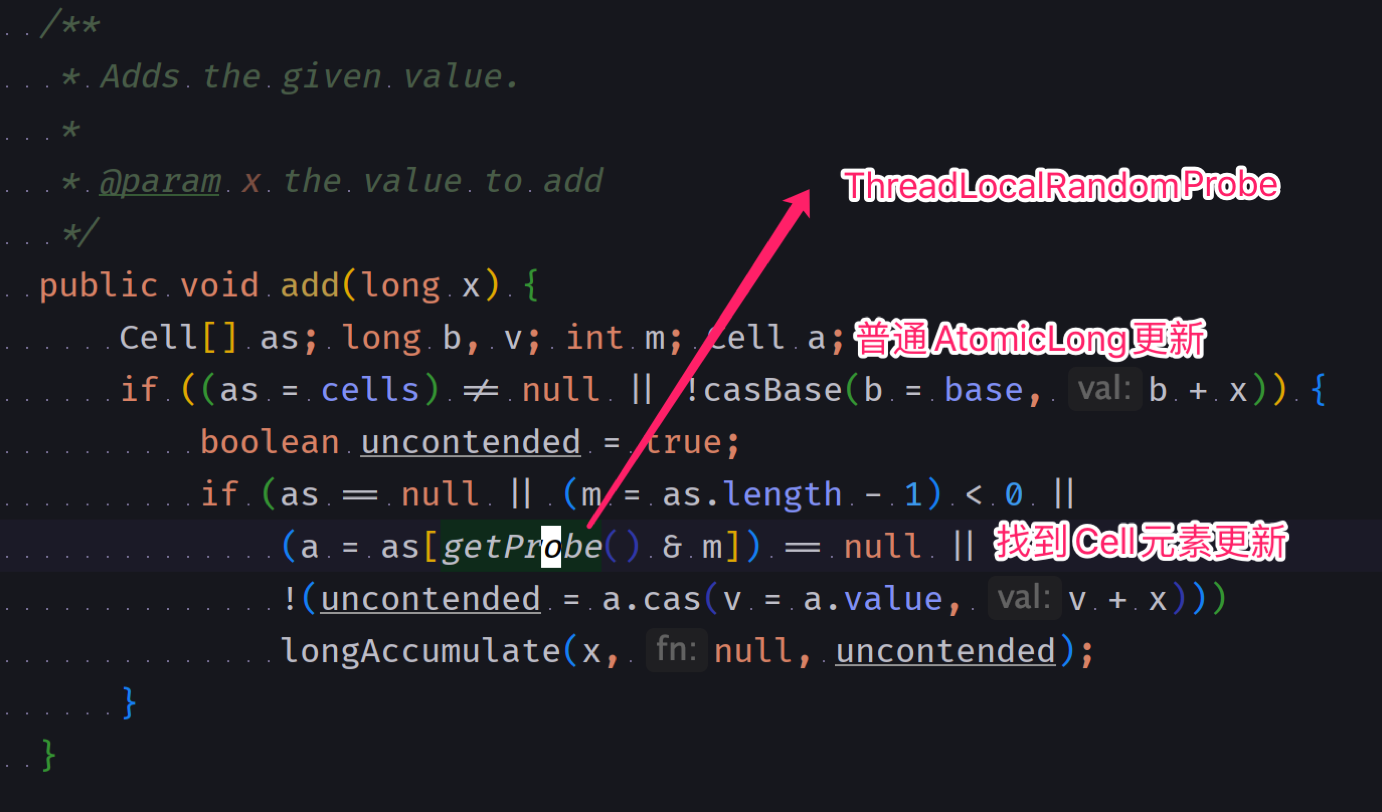
Cell元素，使用@sun.misc.Contended避免cells数组内多个原子变量被放入同一个缓存行，避免了伪共享，提升性能



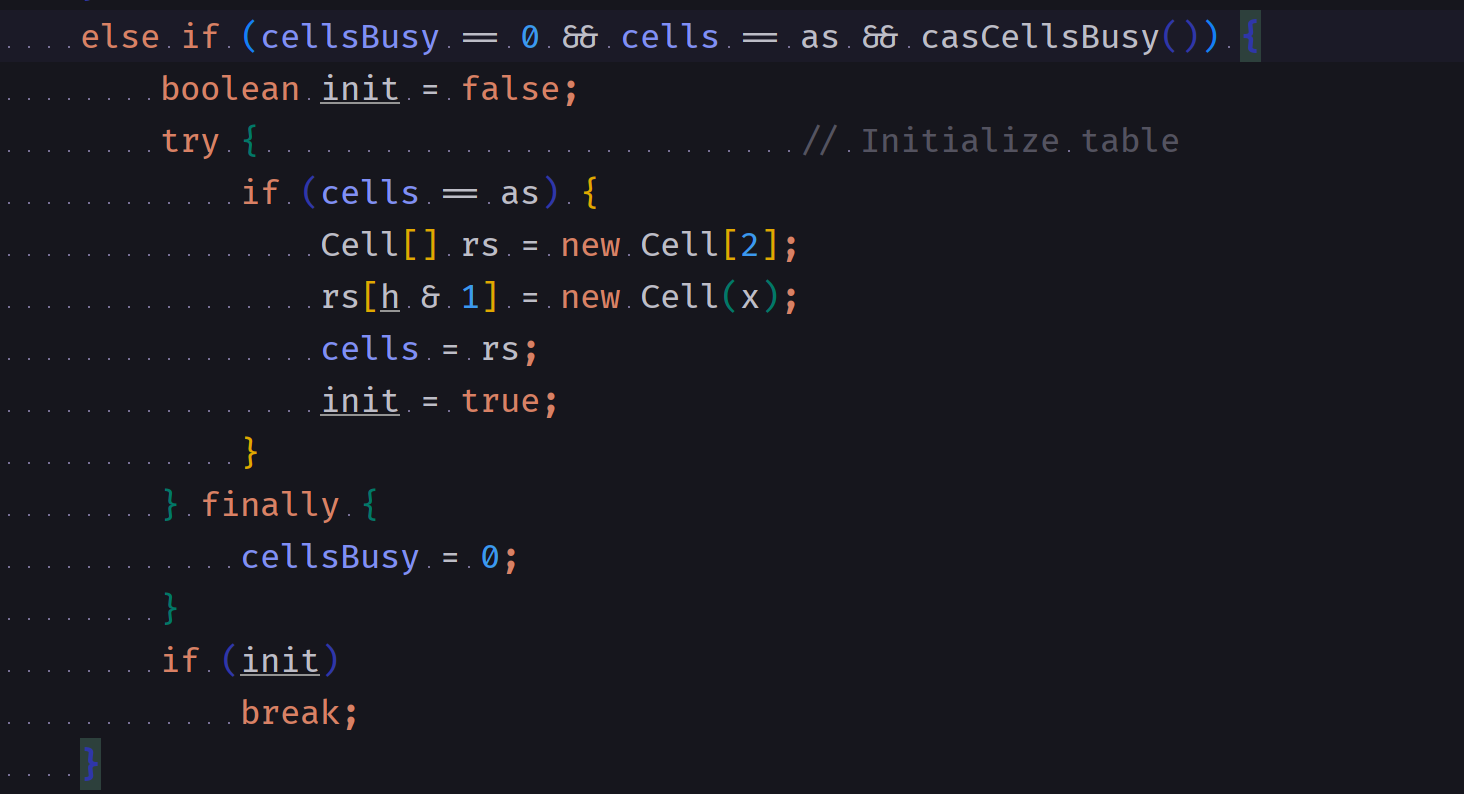
* 1. java.util.concurrent.atomic.LongAdder#sum



* 1. java.util.concurrent.atomic.LongAdder#add



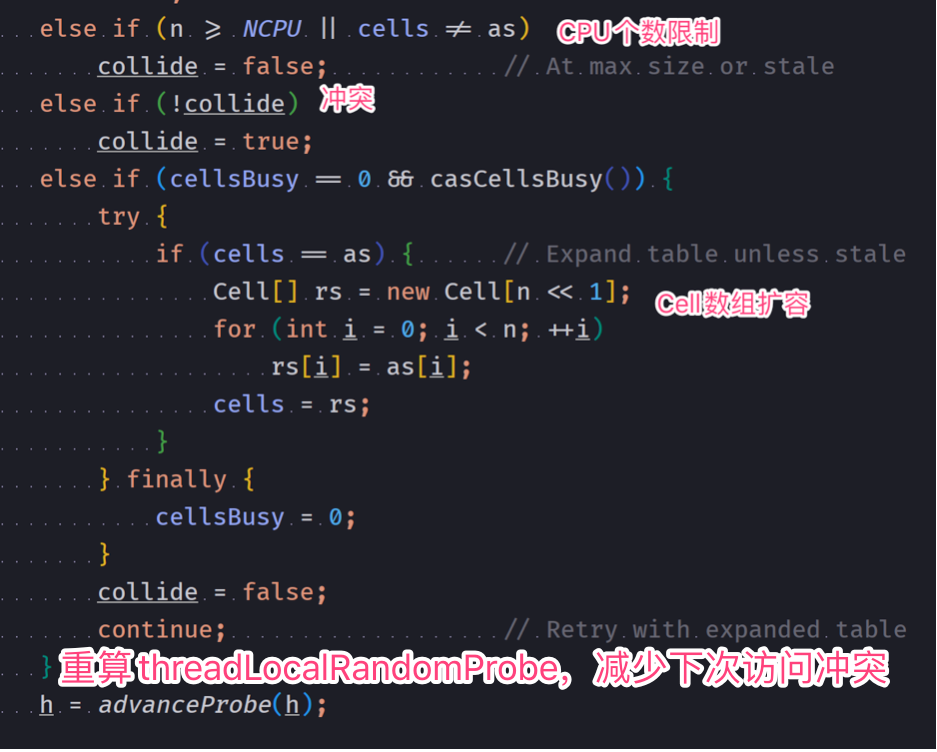
* 1. java.util.concurrent.atomic.Striped64#longAccumulate
     1. Cell数组初始化



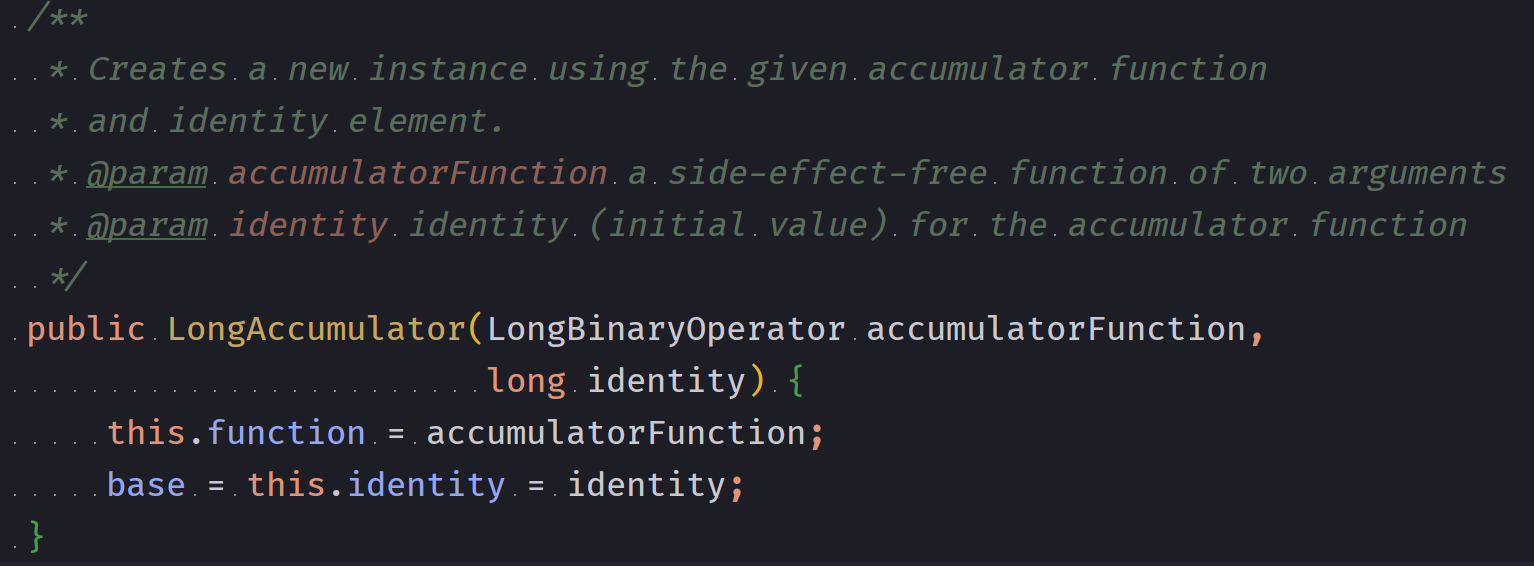
cellsBusy是一个标识，为0说明当前cells数组没有在初始化或者扩容，也没有在新建Cell元素，通过CAS来更新其值

* + 1. Cell数组扩容

当前cells数组元素个数小于当前CPU个数(保证每个Cell都使用一个CPU处理)并且当前多个线程访问了cells中的同一个元素，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。



1. java.util.concurrent.atomic.LongAccumulator，LongAccumulator提供了比LongAdder更通用的能力，内部实现基本一致，但是LongAccumulator可以指定初始值以及如何更新值的函数



LongAdder相当于以下的LongAccumulator

