[https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933173&idx=1&sn=80eb550294677b0042fc030f90cce109&chksm=88621b0bbf15921d2274a7bf6afde912fec02a4c3ade9cfb50d03cdce73e07e33d08d35a3b27&token=1033016931&lang=zh\_CN&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933173&idx=1&sn=80eb550294677b0042fc030f90cce109&chksm=88621b0bbf15921d2274a7bf6afde912fec02a4c3ade9cfb50d03cdce73e07e33d08d35a3b27&token=1033016931&lang=zh_CN&scene=21" \l "wechat_redirect)

# JUC底层工具类Unsafe

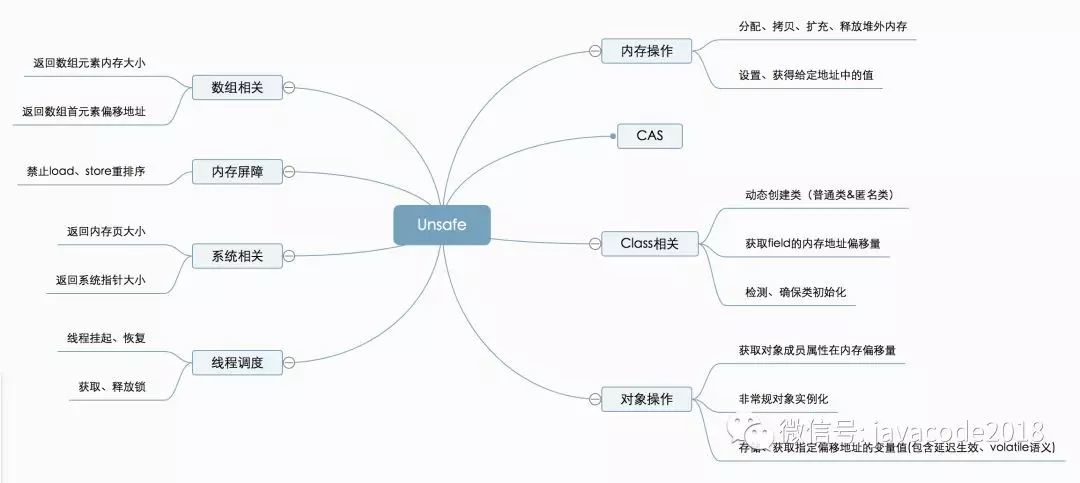
## **本文主要内容**

1. Unsafe基本介绍
2. 获取Unsafe实例
3. Unsafe中的CAS操作
4. Unsafe中原子操作相关方法介绍
5. Unsafe中线程调度相关方法介绍
6. park和unpark示例
7. Unsafe锁示例
8. Unsafe中对volatile的支持

## **基本介绍**

最近我们一直在学习java高并发，java高并发中主要涉及到类位于java.util.concurrent包中，简称juc，juc中大部分类都是依赖于Unsafe来实现的，主要用到了Unsafe中的CAS、线程挂起、线程恢复等相关功能。所以如果打算深入了解JUC原理的，必须先了解一下Unsafe类。

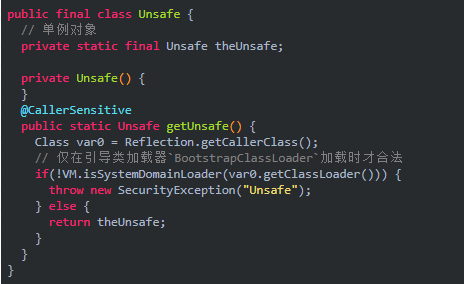
先上一幅Unsafe类的功能图：



Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用。但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此对Unsafe的使用一定要慎重。

从Unsafe功能图上看出，Unsafe提供的API大致可分为**内存操作**、**CAS**、**Class相关**、**对象操作**、**线程调度**、**系统信息获取**、**内存屏障**、**数组操作**等几类，**本文主要介绍3个常用的操作：CAS、线程调度、对象操作。**

看一下UnSafe的原码部分：



从代码中可以看出，Unsafe类为单例实现，提供静态方法getUnsafe获取Unsafe实例，内部会判断当前调用者是否是由系统类加载器加载的，如果不是系统类加载器加载的，会抛出SecurityException异常。

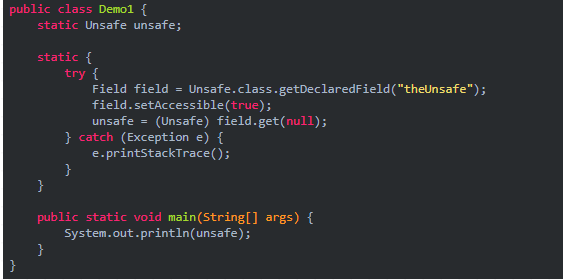
那我们想使用这个类，如何获取呢？

可以把我们的类放在jdk的lib目录下，那么启动的时候会自动加载，这种方式不是很好。

我们学过反射，通过反射可以获取到Unsafe中的theUnsafe字段的值，这样可以获取到Unsafe对象的实例。

## **通过反射获取Unsafe实例**

代码如下：



输出：

sun.misc.Unsafe@76ed5528

## **Unsafe中的CAS操作**

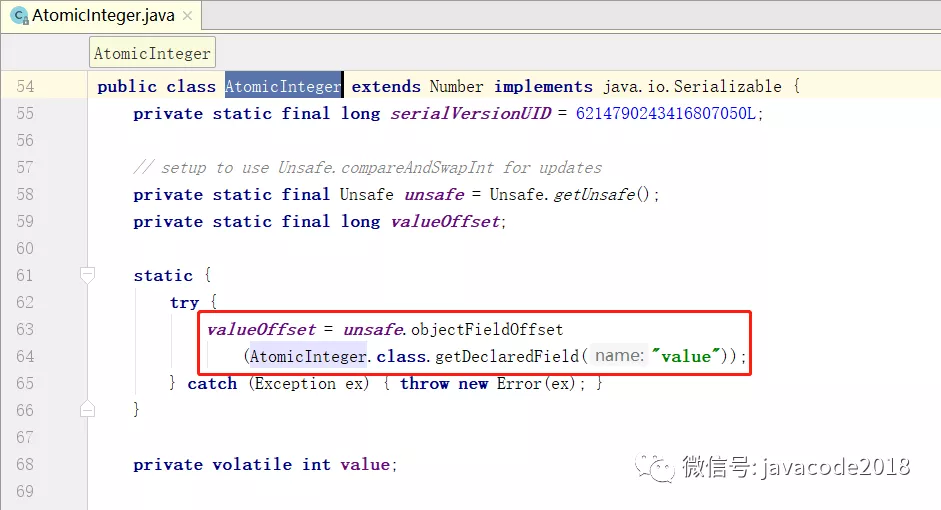
看一下Unsafe中CAS相关方法定义：

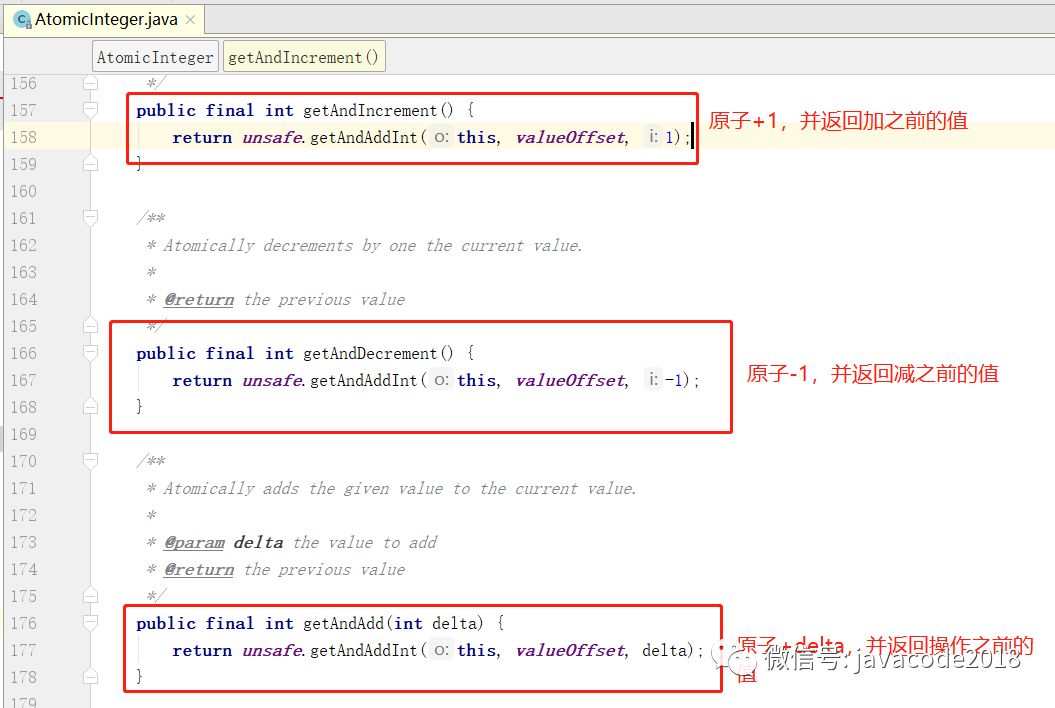
/\*\*  
 \* CAS 操作  
 \*  
 \* @param o        包含要修改field的对象  
 \* @param offset   对象中某field的偏移量  
 \* @param expected 期望值  
 \* @param update   更新值  
 \* @return true | false  
 \*/  
public final native boolean compareAndSwapObject(Object o, long offset, Object expected, Object update);  
  
public final native boolean compareAndSwapInt(Object o, long offset, int expected,int update);  
  
public final native boolean compareAndSwapLong(Object o, long offset, long expected, long update);

什么是CAS? 即比较并替换，实现并发算法时常用到的一种技术。CAS操作包含三个操作数——**内存位置、预期原值及新值**。**执行CAS操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作，多个线程同时执行cas操作，只有一个会成功**。我们都知道，CAS是一条CPU的原子指令（cmpxchg指令），不会造成所谓的数据不一致问题，**Unsafe提供的CAS方法（如compareAndSwapXXX）底层实现即为CPU指令cmpxchg**。执行cmpxchg指令的时候，会判断当前系统是否为多核系统，如果是就给总线加锁，只有一个线程会对总线加锁成功，加锁成功之后会执行cas操作，也就是说CAS的原子性实际上是CPU实现的， 其实在这一点上还是有排他锁的，只是比起用synchronized， 这里的排他时间要短的多， 所以在多线程情况下性能会比较好。

说一下offset，offeset为字段的偏移量，每个对象有个地址，offset是字段相对于对象地址的偏移量，对象地址记为baseAddress，字段偏移量记为offeset，那么字段对应的实际地址就是baseAddress+offeset，所以cas通过对象、偏移量就可以去操作字段对应的值了。

CAS在java.util.concurrent.atomic相关类、Java AQS、JUC中并发集合等实现上有非常广泛的应用，我们看一下java.util.concurrent.atomic.AtomicInteger类，这个类可以在多线程环境中对int类型的数据执行高效的原子修改操作，并保证数据的正确性，看一下此类中用到Unsafe cas的地方：





JUC中其他地方使用到CAS的地方就不列举了，有兴趣的可以去看一下源码。

## **Unsafe中原子操作相关方法介绍**

5个方法，看一下实现：





看一下上面的方法，内部通过自旋的CAS操作实现的，这些方法都可以保证操作的数据在多线程环境中的原子性，正确性。

来个示例，我们还是来实现一个网站计数功能，同时有100个人发起对网站的请求，每个人发起10次请求，每次请求算一次，最终结果是1000次，代码如下：



输出：

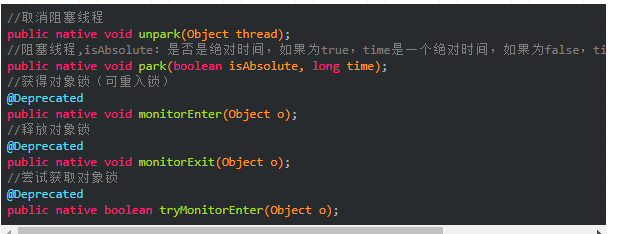


代码中我们在静态块中通过反射获取到了Unsafe类的实例，然后获取Demo2中count字段内存地址偏移量countOffset，main方法中模拟了100个人，每人发起10次请求，等到所有请求完毕之后，输出count的结果。

代码中用到了CountDownLatch，通过countDownLatch.await()让主线程等待，等待100个子线程都执行完毕之后，主线程在进行运行。

## **Unsafe中线程调度相关方法**

这部分，包括线程挂起、恢复、锁机制等方法。



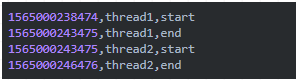
调用park后，线程将被阻塞，直到unpark调用或者超时，如果之前调用过unpark,不会进行阻塞，即park和unpark不区分先后顺序。**monitorEnter、monitorExit、tryMonitorEnter** 3个方法**已过期**，不建议使用了。

## **ark和unpark示例**

代码如下：



输出：



m1()中thread1调用park方法，park方法会将**当前线程阻塞**，被阻塞了5秒之后，被主线程调用unpark方法给唤醒了，unpark方法参数表示需要唤醒的线程。

线程中相当于有个许可，许可默认是0，调用park的时候，发现是0会阻塞当前线程，调用unpark之后，许可会被置为1，并会唤醒当前线程。如果在park之前先调用了unpark方法，执行park方法的时候，不会阻塞。park方法被唤醒之后，许可又会被置为0。多次调用unpark的效果是一样的，许可还是1。

## **Unsafe锁示例**

代码如下：



输出：



注意：

1、monitorEnter、monitorExit、tryMonitorEnter 3个方法已过期，不建议使用了

2、monitorEnter、monitorExit必须成对出现，出现的次数必须一致，也就是说锁了n次，也必须释放n次，否则会造成死锁

## **Unsafe中保证变量的可见性**

java中操作内存分为主内存和工作内存，共享数据在主内存中，线程如果需要操作主内存的数据，需要先将主内存的数据复制到线程独有的工作内存中，操作完成之后再将其刷新到主内存中。如线程A要想看到线程B修改后的数据，需要满足：线程B修改数据之后，需要将数据从自己的工作内存中刷新到主内存中，并且A需要去主内存中读取数据。

被关键字volatile修饰的数据，有2点语义：

* 如果一个变量被volatile修饰，读取这个变量时候，会强制从主内存中读取，然后将其复制到当前线程的工作内存中使用
* 给volatile修饰的变量赋值的时候，会强制将赋值的结果从工作内存刷新到主内存

上面2点语义保证了被volatile修饰的数据在多线程中的可见性。

Unsafe中提供了和volatile语义一样的功能的方法，如下：

//设置给定对象的int值，使用volatile语义，即设置后立马更新到内存对其他线程可见  
public native void  putIntVolatile(Object o, long offset, int x);  
//获得给定对象的指定偏移量offset的int值，使用volatile语义，总能获取到最新的int值。  
public native int getIntVolatile(Object o, long offset);

putIntVolatile方法，3个参数：

* o：表示需要操作的对象
* offset：表示操作对象中的某个字段地址偏移量
* x：将offset对应的字段的值修改为x，并且立即刷新到主存中

调用这个方法，会强制将工作内存中修改的数据刷新到主内存中。

getIntVolatile方法，2个参数

* o：表示需要操作的对象
* offset：表示操作对象中的某个字段地址偏移量

每次调用这个方法都会强制从主内存读取值，将其复制到工作内存中使用。

其他的还有几个putXXXVolatile、getXXXVolatile方法和上面2个类似。