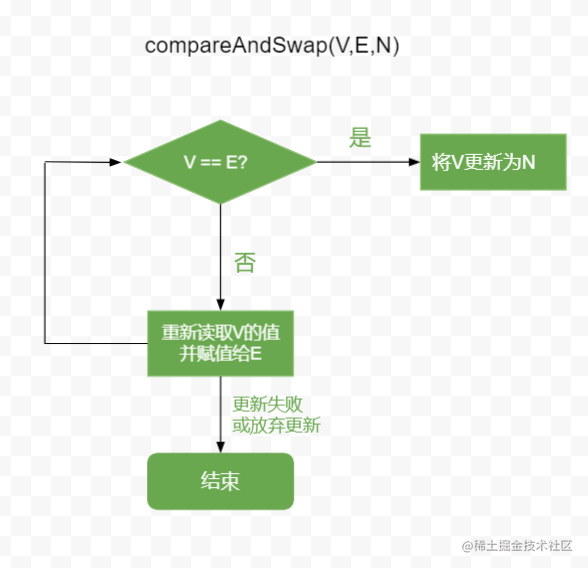
Atomic包下的类是无锁操作，无锁的实现就得益于CAS。

CAS的函数公式：compareAndSwap(V,E,N)； 其中V表示要更新的变量，E表示预期值，N表示期望更新的值。调用compareAndSwap函数来更新变量V，如果V的值等于期望值E，那么将其更新为N，如果V的值不等于期望值E，则说明有其它线程跟新了这个变量，此时不会执行更新操作，而是重新读取该变量的值再次尝试调用compareAndSwap来更新。

可见CAS其实存在一个循环的过程，如果有多个线程在同时修改这一个变量V，在修改之前会先拿到这个变量的值，再和变量对比看是否相等，如果相等，则说明没有其它线程修改这个变量，自己更新变量即可。如果发现要修改的变量和期望值不一样，则说明再读取变量V的值后，有其它线程对变量V做了修改，那么，放弃本次更新，重新读变量V的值，并再次尝试修改，直到修改成功为止。这个循环过程一般也称作自旋，CAS操作的整个过程如下图所示：



### CAS存在的缺点

虽然通过CAS可以实现无锁同步，但是CAS也有其局限性和问题所在。

（1）只能保证一个共享变量的原子性

CAS不像synchronized和RetranLock一样可以保证一段代码和多个变量的同步。对于多个共享变量操作是CAS是无法保证的，这时候必须使用枷锁来是实现。

（2）存在性能开销问题

由于CAS是一个自旋操作，如果长时间的CAS不成功会给CPU带来很大的开销。

（3）ABA问题

因为CAS是通过检查值有没有发生改变来保证原子性的，假若一个变量V的值为A，线程1和线程2同时都读取到了这个变量的值A，此时线程1将V的值改为了B，然后又改回了A，期间线程2一直没有抢到CPU时间片。知道线程1将V的值改回A后线程2才得到执行。那么此时，线程2并不知道V的值曾经改变过。这个问题就被成为ABA问题。

ABA问题的解决其实也容易处理，即添加一个版本号，更次更新值同时也更新版本号即可。上文中提到的AtomicStampedReference就是用来解决ABA问题的。

CPU对CAS的支持

在操作系统中CAS是一种系统原语，原语由多条指令组成，且原语的执行是连续不可中断的。因此CAS实际上是一条CPU的原子指令，虽然看上去CAS是一个先比较再交换的操作，但实际上这个过程是由CPU保证了原子操作。

CAS与Atomic原子类

了解了CAS，我们就来看下Atomic包中的原子类是如何使用CAS实现原子操作的。我们以AtomicInteger为例来看.

public class AtomicInteger extends Number implements java.io.Serializable {

private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();

public final int getAndSet(int newValue) {

return U.getAndSetInt(this, VALUE, newValue);

}

public final int getAndIncrement() {

return U.getAndAddInt(this, VALUE, 1);

}

public final int getAndDecrement() {

return U.getAndAddInt(this, VALUE, -1);

}

public final int getAndAdd(int delta) {

return U.getAndAddInt(this, VALUE, delta);

}

public final int incrementAndGet() {

return U.getAndAddInt(this, VALUE, 1) + 1;

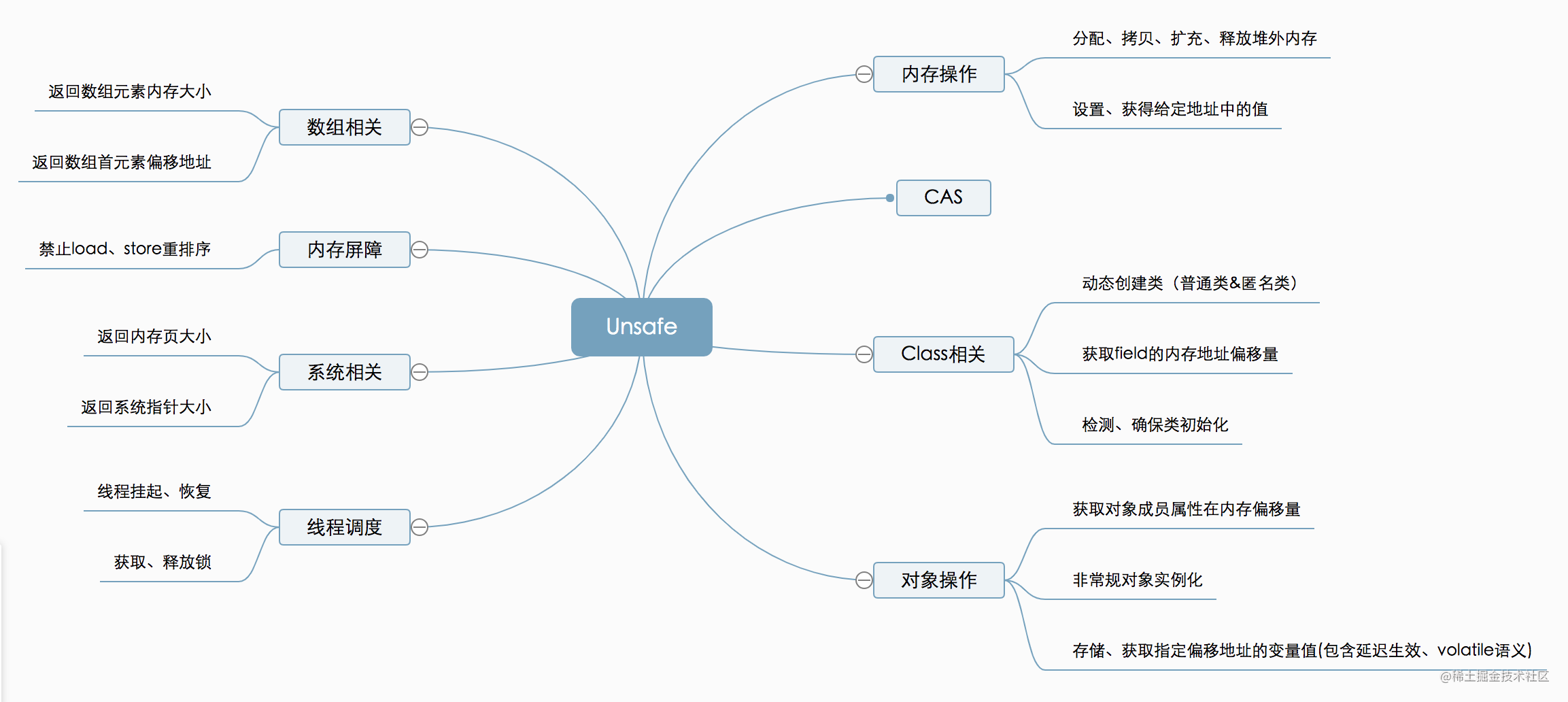
}

public final int decrementAndGet() {

return U.getAndAddInt(this, VALUE, -1) - 1;

}

Unsafe是一个神奇且鲜为人知的Java类，因为在平时开发中很少用到它。但是这个类中为我们提供了相当多的功能，它即可以让Java语言像C语言指针一样操作内存，同时还提供了CAS、内存屏障、线程调度、对象操作、数组操作等能力,如下图。



Unsafe类是一个单例，并且提供了一个getUnsafe的方法来获取Unsafe的实例。但是，这个方法只有在引导类加载器加载Unsafe类是调用才合法，否则会抛出一个SecurityException异常，如下：

Exception in thread "main" java.lang.SecurityException: Unsafe

at jdk.unsupported/sun.misc.Unsafe.getUnsafe(Unsafe.java:99)

at atomic.AtomicDemo.increase(AtomicDemo.java:28)

at atomic.AtomicDemo.main(AtomicDemo.java:34)

因此，想要获取Unsafe类的实例就需要另辟蹊径了。使用反射来获取Unsafe实例是一个比较好的方案，实现代码如下：

try {

Field field = Unsafe.class.getDeclaredField("theUnsafe");

field.setAccessible(true);

Unsafe unsafe = (Unsafe) field.get(null);

} catch (Exception e) {

e.printStackTrace();

}

### Unsafe类中的CAS

Unsafe类中与CAS相关的主要有以下几个方法

// 第一个参数o为给定对象，offset为对象内存的偏移量，通过这个偏移量迅速定位字段并设置或获取该字段的值，expected表示期望值，x表示要设置的值，下面3个方法都通过CAS原子指令执行操作。

public final native boolean compareAndSetInt(Object o,long offset,int expected,int x);

public final native boolean compareAndSetObject(Object o, long offset,Object expected,Object x);

public final native boolean compareAndSetLong(Object o, long offset,long expected,long x);

可以看到，这些方法都是native方法，调用的底层代码实现。在JDK1.8中还引入了getAndAddInt、getAndAddLong、getAndSetInt、getAndSetLong、getAndSetObject等方法来支持不同类型CAS操作。

而AtomicInteger中也正是使用了这里的方法才实现的CAS操作。

### 线程调度相关

在Unsafe中提供了线程挂起、恢复及锁机制相关的方法。

//取消阻塞线程

public native void unpark(Object thread);

//阻塞线程

public native void park(boolean isAbsolute, long time);

//获得对象锁（可重入锁）

@Deprecated

public native void monitorEnter(Object o);

//释放对象锁

@Deprecated

public native void monitorExit(Object o);

//尝试获取对象锁

@Deprecated

public native boolean tryMonitorEnter(Object o);

RetranLock与AQS时涉及到线程挂起的操作其实也是调用的Unsafe的park方法。

// LockSupport

private static final Unsafe U = Unsafe.getUnsafe();

public static void park(Object blocker) {

Thread t = Thread.currentThread();

setBlocker(t, blocker);

U.park(false, 0L);

setBlocker(t, null);

}

对象操作 Unsafe还提供了对象实例化及操作对象属性相关的方法

//返回对象成员属性在内存地址相对于此对象的内存地址的偏移量

public native long objectFieldOffset(Field f);

//获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，getChar等

public native Object getObject(Object o, long offset);

//给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，putChar等

public native void putObject(Object o, long offset, Object x);

//从对象的指定偏移量处获取变量的引用，使用volatile的加载语义

public native Object getObjectVolatile(Object o, long offset);

//存储变量的引用到对象的指定的偏移量处，使用volatile的存储语义

public native void putObjectVolatile(Object o, long offset, Object x);

//有序、延迟版本的putObjectVolatile方法，不保证值的改变被其他线程立即看到。只有在field被volatile修饰符修饰时有效

public native void putOrderedObject(Object o, long offset, Object x);

//绕过构造方法、初始化代码来创建对象

public native Object allocateInstance(Class<?> cls) throws InstantiationException;

Unsafe中提供的allocateInstance方法可以绕过对象的构造方法直接创建对象，Gson解析json反序列化对象时就有用到这个方法。

// 来自Gson#UnsafeAllocator

public abstract <T> T newInstance(Class<T> var1) throws Exception;

public static UnsafeAllocator create() {

try {

Class<?> unsafeClass = Class.forName("sun.misc.Unsafe");

Field f = unsafeClass.getDeclaredField("theUnsafe");

f.setAccessible(true);

final Object unsafe = f.get((Object)null);

final Method allocateInstance = unsafeClass.getMethod("allocateInstance", Class.class);

return new UnsafeAllocator() {

public <T> T newInstance(Class<T> c) throws Exception {

assertInstantiable(c);

return allocateInstance.invoke(unsafe, c);

}

};

} catch (Exception var6) {

// ...省略异常处理

}

}

Unsafe的其它功能

除了CAS、线程调度、对象相关的功能外，Unsafe还提供了内存操作，可以实现堆外内存的分配。提供的数组相关的方法来定位数组中每个元素在内存中的位置。