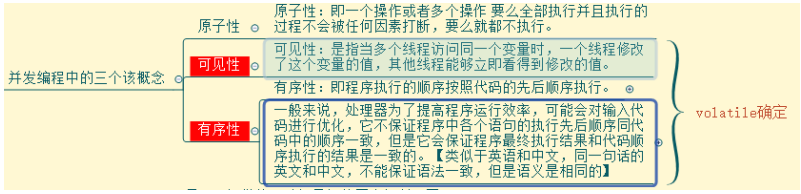
注意：笔记中所有被我加了注释的代码都被我添加到笔记中了，没有添加到笔记中的代码基本上都能看懂，有实在看不懂的地方去回顾下视频

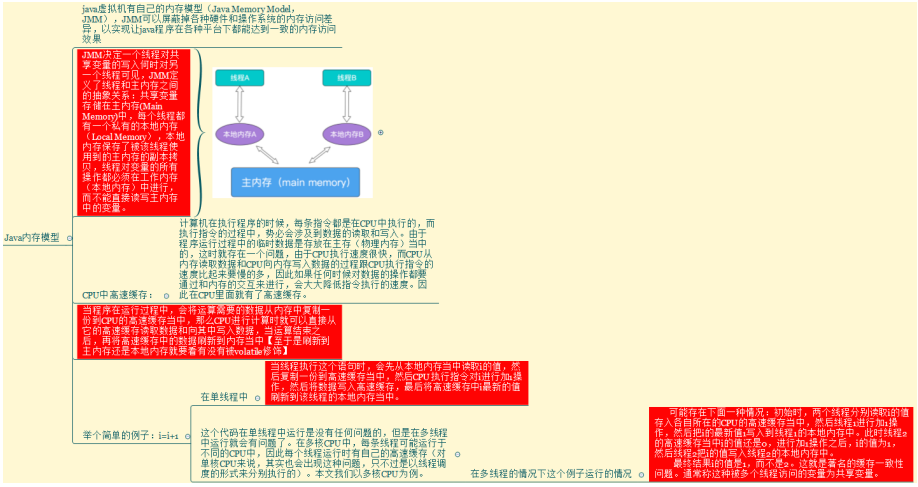
1. 课程内容介绍

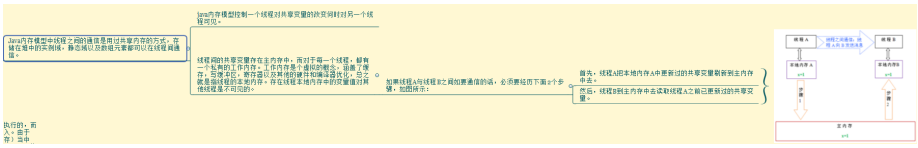
- volatile
- Actomic
- ThreadLocal
- 同步类容器
- 并发类容器
- 并发无阻塞式队列
- 并发阻塞式队列

2. volatile关键字

- 用法：： private volatile int a = 0;
- 强制线程到共享内存中读取数据，而不从线程工作内存中读取，从而使变量在多个线程间可见。







- 示例： DemoThread13（展示这个关键字的可见性）

- volatile无法保证原子性，volatile属于轻量级的同步，性能比synchronized强很多(不加锁)，但是只保证线程见的可见性、有序性，并不能替代synchronized的同步功能，netty框架中大量使用了volatile
- 示例：DemoThread14（使用AtomicInteger展示原子性）

3. volatile与static的区别

- 两者说的不是一个层面的问题
- static保证唯一性, 不保证一致性，多个实例共享一个静态变量(在多线程环境下，静态变量和非静态变量之间没有区别)。
- volatile保证一致性（可见性），不保证唯一性，多个实例有多个volatile变量。

4. Atomic类的原子性

- 使用java.util.concurrent.atomic包下的AtomicInteger等原子类可以保证**Atomic*类型的共享变量的原子性**这些共享变量的原子性是依靠Atomic*类中的原子性方法（加了一些类似于锁的机制，如CAS）实现的。
  AtomicXX类大量采用Unsafe类完成底层操作。
- 示例：DemoThread15

- 但是不能保证使用Atomic*类型的成员变量所在的类中的成员方法的原子性（我看上去是句废话）
- 示例：DemoThread16

- Actomic类采用了CAS这种非锁机制

4.1. CAS

4.1.1 CAS —— 原理

知识点

1. 本质是一个乐观锁

2. JDK提供的非阻塞原子操作，通过硬件保证了比较、更新操作的原子性
3. JDK的Unsafe类提供了一系列的compareAndSwap*方法来支持CAS操作

原理

4.1.2 CAS —— ABA问题

> *知识点*

1. 如果程序按照1~5的顺序执行，依然是成功的，然而线程1修改时x的值时其实已经从x= A =>B =>A，但是线程1不知道此时的A不彼时的A。
2. 由于变量的值产生了环形转换，从A变为B又变回了A。如果不存在环形转换也就不存在ABA问题。

4.1.3 CAS -- 解决ABA问题

知识点

1. 给变量分配时间戳、版本来解决ABA问题
2. JDK中使用java.util.concurrent.atomic.AtomicStampedReference类给每个变量的状态都分配一个时间戳（也就是版本号），避免ABA问题产生。

示例：Demo：com.mkevin.demo2.CasDemo0（使用带有版本号的CAS的示例）

```
//初始化值是100，版本是0
private static AtomicStampedReference<Integer> atomic =
new AtomicStampedReference<>(100, 0);

public static void main(String[] args) throws
InterruptedException {
    Thread t0 = new Thread(new Runnable() {
        /**
```

```
        这个线程的作用是进行ABA操作，目的是为了查看JDK是否能解决线程实例对象t1产生误会的问题
        */
        @Override
        public void run() {
            try {
                //线程一启动直接先sleep 1 s
                TimeUnit.SECONDS.sleep(1);
                //调用方法对比并且设置值，方法中第一个参数是期望的值为100，想要修改城的值为101，想要对比的版本号是atomic.getStamp()，如果更新成功了时间戳（版本号）会加1；返回boolean代表设置成功还是失败。
                boolean sucess = atomic.compareAndSet(100, 101, atomic.getStamp(), atomic.getStamp() + 1);

 System.out.println(Thread.currentThread().getName()+" set 100>101 : "+sucess);
                //将101改为100，参数的内容同上
                sucess = atomic.compareAndSet(101, 100, atomic.getStamp(), atomic.getStamp() + 1);

 System.out.println(Thread.currentThread().getName()+" set 101>100 : "+sucess);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    t0.start();

    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                //先获取atomic对象的时间戳（版本），这个时间戳是线程实例对象t0修改之前的，因为t0线程刚开始运行就sleep 1s
                int stamp = atomic.getStamp();

 System.out.println(Thread.currentThread().getName()+" 修改之前 : " +stamp);
                //等待两秒之后再次获取时间戳，也就是线程实例对象t0修改之后的时间戳
                TimeUnit.SECONDS.sleep(2);
                int stamp1 = atomic.getStamp();

 System.out.println(Thread.currentThread().getName()+" 等待两秒之后,版本被t0线程修改为 : " +stamp1);

                //注意：这里对比的时间戳是线程实例对象t0修改之前的时间戳stamp，想要展示的是ABA问题，接下来的操作就和t0线程实例对象的操作差不多，只是时间戳有改动
```

```
                //Kevin提醒: 一下两次修改都不会成功,因为版本不
符,虽然期待值是相同的,因此解决了ABA问题
                boolean success =
atomic.compareAndSet(100, 101, stamp, stamp + 1);

 System.out.println(Thread.currentThread().getName()+" set
100>101 使用错误的时间戳: " + success);
                success =
atomic.compareAndSet(101,100,stamp,stamp+1);

 System.out.println(Thread.currentThread().getName()+" set
101>100 使用错误的时间戳: " + success);

                //Kevin提醒：以下修改是成功的,因为使用了正确的版
本号,正确的期待值
                success =
atomic.compareAndSet(100,101,stamp1,stamp1+1);

 System.out.println(Thread.currentThread().getName()+" set
100>101 使用正确的时间戳: " + success);

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    t1.start();

    t0.join();
    t1.join();

    System.out.println("main is over");
}
```

运行的结果:

```
Thread-1 修改之前 : 0
Thread-0 set 100>101 : true
Thread-0 set 101>100 : true
Thread-1 等待两秒之后,版本被t0线程修改为 : 2
Thread-1 set 100>101 使用错误的时间戳: false
Thread-1 set 101>100 使用错误的时间戳: false
Thread-1 set 100>101 使用正确的时间戳: true
main is over
```

AtomicStampedReference类的核心内部类:

```
private static class Pair<T> {
    final T reference;
    final int stamp;
    private Pair(T reference, int stamp) {
        this.reference = reference;
        this.stamp = stamp;
    }
    static <T> Pair<T> of(T reference, int stamp) {
        return new Pair<T>(reference, stamp);
    }
}
```

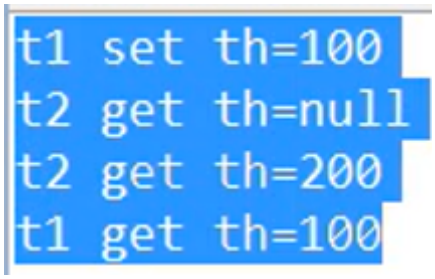

5. ThreadLocal源码解析

该类的好像一个工具类，用于维护线程内部变量。

ThreadLocalMap类才是真正存储（隔离）数据的东西。

1. 使用ThreadLocal维护变量的时候，该类使用时包裹在一个变量/类型的外面，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不影响其他的线程对应的副本
2. 示例：DemoThread21

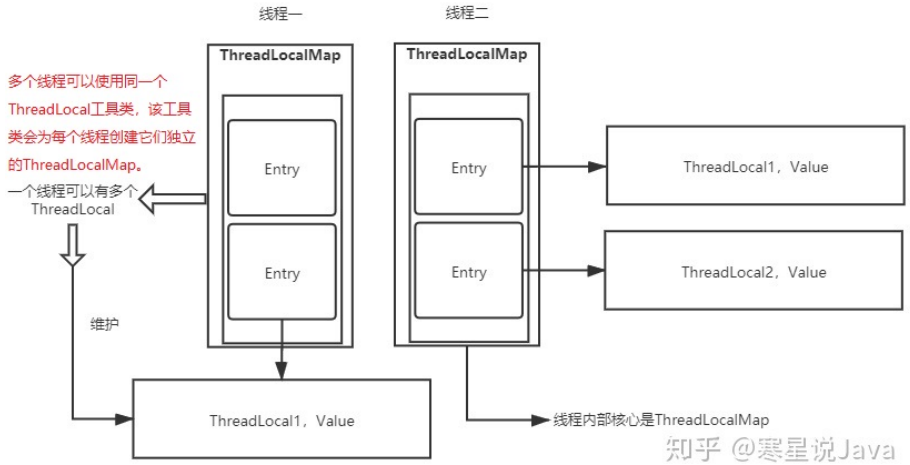
   程序运行的结果：

   在照片中，线程t1将ThreadLocal中的变量设置成100，之后睡眠2s，在1s之后线程t2开始运行，这个时候线程t2是得不到同一个ThreadLocal实例对象中为线程t1设置的值100的；
   因为ThreadLocal类中封装了一个ThreadLocalMap类型的结构，ThreadLocalMap类是ThreadLocal类中的一个静态内部类，ThreadLocalMap类中封装了一个Entry静态内部类，这个静态内部类继承了一个泛型为ThreadLocalMap类型的弱引用，这个Entry结构中存储的是一个个线程与它们对应的局部变量（类似于映射），在ThreadLocalMap类中使用数组存储这个Entry类的实例对象，而t2线程此时在ThreadLocalMap的Entry类型的数组中还未存储映射，所以得到的是null。

5.1 ThreadLocal源码解析

关于Thread、ThreadLocal、ThreadLocalMap、Entry之间的关系图



图简介：

1. 每个Thread线程的实例对象中都有一个自己的ThreadLocalMap，多个线程可以使用同一个ThreadLocal实例对象
2. Thread内部的ThreadLocalMap是由ThreadLoal维护的；
   Thread类中没有使用ThreadLocal类中的方法对ThreadLocalMap进行操作；
   ThreadLocal通过Thread类的静态方法获取当前正在运行的线程或通过参数获得当前正在运行的线程的实例对象，使用这个实例对象获得当前线程中的ThreadLocalMap类型的字段的值（记作TLM），之后ThreadLocal使用自己的方法操作TLM（向ThreadLocalMap中获取、设置、移除线程的变量值）
3. 每个ThreadLocalMap中都存储本地对象ThreadLocal（key，不同线程的ThreadLocal不同）和线程的变量副本（value）
4. 一个Thread可以有多个ThreadLocal
5. 每个线程都有其独立的ThredLocalMap，而Map中存的是Threalocal为Key变量副本为value的键值对，以此达到变量隔离的目的

知识点

1. Thread类中的threadLocals、inheritableThreadLocals成员变量为ThreadLocal.ThreadLocalMap对象，这两个对象一个是继承一个是非继承。
2. ThreadLocal类中有一个createMap(Thread t, T firstValue)方法，在该方法中会给参数中的线程（一般传递进来的就是当前正在运行的线程）的实例对象中ThreadLocal.ThreadLocalMap类型的threadlocals属性设置值，也就是给当前线程的实例对象创建一个ThreadLocalMap类型的值用于存放数组Entry的实例对象
3. ThreadLocalMap的key值是ThreadLocal对象本身，查看get、set、remove方法
    a. 在调用上面这三个方法的时候都会使用Thread.currentThread()先获取当前正在运行的线程，以便对ThreadLocalMap的底层数组中保存的Entry对象进行操作。

4. ThreadLocal无法解决继承问题（也就是在父线程中定义的线程局部变量子线程无法访问，例如在main线程中使用ThreadLocal包装了数值100，但是在main线程中创建的子线程中无法直接得到这个100），而InheritableThreadLocal可以，在父子线程中子线程仅仅继承父线程的局部变量，但是子线程修改该局部变量的值不会影响父线程中的值，同样也不会影响同级子限线程中的局部变量的值。
5. 每个ThreadLocal只能保存一个变量副本，如果想要一个线程能够保存多个副本以上，就需要创建多个ThreadLocal。
6. 每次使用完ThreadLocal，都调用它的remove()方法，清除数据，防止内存泄漏。
7. InheritableThreadLocal继承自ThreadLocal，子线程在初始化的时候（创建Thread类的时候，就会判断父线程中InheritableThreadLocal类型的字段inheritableThreadLocals是否为空，如果不为空，就会父线程的ThreadLocalMap初始化子线程的InheritableThreadLocal）
8. InheritableThreadLocal可以帮助我们做链路追踪

Demo
com.mkevin.demo2.ThreadLocalDemo1
com.mkevin.demo2.ThreadLocalDemo0

补充：ThreadLocal导致的内存泄漏问题

ThreadLocal在没有外部对象强引用时（如Thread），发生GC时弱引用Key会被回收，而Value是强引用不会回收，如果创建ThreadLocal的线程一直持续运行如线程池中的线程，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

> *其中ThreadLocalMap类的静态内部类Entry中保存的ThreadLocal类型的键是WeadReference类型的原因如下：*
>
> 1. *key 如果使用强引用：引用的ThreadLocal的对象需要被回收时，由于ThreadLocalMap还持有ThreadLocal的强引用，如果没有手动删除，ThreadLocal不会被回收，导致Entry内存泄漏。*
> 2. *key 使用弱引用：引用的ThreadLocal的对象需要被回收时，由于ThreadLocalMap持有ThreadLocal的弱引用，即使没有手动删除，ThreadLocal也会被回收。value在下一次ThreadLocalMap调用set,get，remove的时候会被清除。*

6. Unsafe方法详解

Unsafe实操

6.1 Unsafe——突破Unsafe类的安全限制

AtomicXX类大量采用Unsafe类完成底层操作。

相关源码在OpenJDK中的sun.misc.Unsafe

知识点

1. 通过反射模式可以突破Unsafe类的安全限制
2. 这个类是不安全的，可以直接操作内存（释放内存、分配内存地址、重新分配内存.....）

实操演练：

1. com.mkevin.demo3.UnsafeDemo0
    a. 单例模式
    b. 抛出java.lang.SecurityException:Unsafe，只有BootStrapClassLoader加载的类可以调用

```
private int age;

public int getAge() {
    return age;
}

public static void main(String[] args) {
    UnsafeDemo0 demo0 = new
UnsafeDemo0();

    // 获取Unsafe类实例
    Unsafe unsafe = Unsafe.getUnsafe();
    try {
        //获取对象中age属性的内存偏移地址
        long ageOffset =
unsafe.objectFieldOffset(UnsafeDemo0.clas
s.getDeclaredField("age"));
        //设置age的值为11，通过直接操作内存的
方式给对象的属性赋值

 unsafe.putInt(demo0,ageOffset,11);
        //输出结果

 System.out.println(demo0.getAge());
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    }
}
```

6.2 Unsafe——普通字段操作

知识点 地址类操作

1. objectFieldOffset 获取普通字段偏移地址
2. staticFieldOffset 获取静态字段偏移地址
3. arrayBaseOffset 获取数组中第一个元素的地址
4. arrayIndexScale 获取数组中一个元素占用的字节

知识点 get、put

1. getInt、getLong、getBoolean、getChar、getFloat、getByte、getDouble、getObject 获取字段值
2. putInt、putLong、putBoolean、putChar、putFloat、putByte、putDouble、putObject 设置字段值
3. 直接操作内存地址
4. 通过对象内存地址操作

知识点 volatile类操作

1. getIntVolatile、getLongVolatile、getBooleanVolatile、getObjectVolatile、getByteVolatile、getShortVolatile、getCharVolatile、getFloatVolatile、getDoubleVolatile 获取volatile字段值，保证可见性
2. putIntVolatile、putLongVolatile、putBooleanVolatile、putObjectVolatile、putByteVolatile、putShortVolatile、putCharVolatile、putFloatVolatile、putDoubleVolatile 设置volatile字段值，保证可见性

知识点 and类操作

1. putOrderedInt、 putOrderedLong、 putOrderedObject 保证顺序性、具有lazy特性、不保证可见性
2. getAndSetInt、getAndSetLong、getAndSetObject 自旋操作、先获取后设置
3. getAndAddInt、getAndAddLong 自旋操作、先获取后设置
4. compareAndSwapInt、compareAndSwapLong、compareAndSwapObject CAS相关操作

示例：
com.mkevin.demo3.UnsafeDemo1（对实例字段的操作）
com.mkevin.demo3.UnsafeDemo2（对数组的操作，将数组作为一个整体的字段）
com.mkevin.demo3.UnsafeDemo3（对数组的操作，操作数组中的每个值）
com.mkevin.demo3.UnsafeDemo4（对静态字段的操作）
com.mkevin.demo3.UnsafeDemo6（对volatile修饰的操作）

6.3 Unsafe——内存操作

1. public native long allocateMemory(long bytes); 分配内存
2. public native long reallocateMemory(long address, long bytes); 重新分配内存
3. public native void setMemory(Object o, long offset, long bytes, byte value);
4. public void setMemory(long address, long bytes, byte value) 初始化内存

5. public native void copyMemory(Object srcBase, long srcOffset, Object destBase, long destOffset, long bytes);
6. public void copyMemory(long srcAddress, long destAddress, long bytes) 复制内存
7. public native void freeMemory(long address);释放内存，释放之后就不能再使用这块内存了

Demo：com.mkevin.demo3.UnsafeDemo5

6.4 Unsafe——线程调度

线程调度

1. public native void park(boolean isAbsolute, long time); 挂起线程，第一个参数是判断第二个参数是绝对时间还是相对时间，绝对时间是从格林尼治时间开始延迟参数time，相对时间是从现在开始延迟参数time。
2. public native void unpark(Object thread); 唤醒线程
3. 需要注意线程的interrupt方法同样能唤醒线程，但是调用park处不抛出异常，可以在调用park后使用isInterrupted方法判断是被中断
4. java.util.concurrent.locks.LockSupport使用unsafe实现

Demo：com.mkevin.demo3.UnsafeDemo7

6.5 Unsafe

内存屏障

1. public native void loadFence(); 保证在这个屏障之前的所有读操作都已经完成
2. public native void storeFence(); 保证在这个屏障之前的所有写操作都已经完成
3. public native void fullFence(); 保证在这个屏障之前的所有读写操作都已经完成

类加载

1. public native Class<?> defineClass(String name, byte[] b, int off, int len, ClassLoader loader, ProtectionDomain protectionDomain); 方法定义一个类，用于动态地创建类。
2. public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[] data, Object[] cpPatches);用于动态的创建一个匿名内部类。
3. public native Object allocateInstance(Class<?> cls) throws InstantiationException;方法用于创建一个类的实例，但是不会调用这个实例的构造方法，如果这个类还未被初始化，则初始化这个类。
4. public native boolean shouldBeInitialized(Class<?> c);方法用于判断是否需要初始化一个类。
5. public native void ensureClassInitialized(Class<?> c);方法用于保证已经初始化过一个类。

7. 同步类容器

- Vector、HashTable等古老的并发容器，都是使用Collections.synchronizedXXX等工厂方法创建的，并发状态下只能有一个线程访问容器对象，性能很低
- 示例：DemoThread26 （古老容器的线程安全实现方法）

8. 并发类容器

- JDK5.0之后提供了多种并发类容易可以替代同步类容器，提升性能、吞吐量
- ConcurrentHashMap替代HashMap、HashTable（底层将整个数据结构加上一把锁），无序
- ConcurrentSkipListMap替代TreeMap，有序
- ConcurrentHashMap将hash表分为16个segment，每个segment单独进行锁控制，从而减小了锁的粒度，提升了性能
- 例子： DemoThread27 (ConcurrentHashMap、ConcurrentSkipListMap)

9. 并发类容器

- Copy On Write容器,简称COW;写时复制容器（读写分离容器），向容器中添加元素时，先将容器进行Copy出一个新容器，然后将元素添加到新容器中，再将原容器的引用指向新容器。并发读的时候不需要锁定容器， 因为原容器没有变化，使用的是一种读写分离的思想。由于每次更新都会复制新容器，所以如果数 据量较大，并且更新操作频繁则对内存消耗很高，建议在高并发读的场景下使用
- CopyOnWriteArraySet基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法,addIfAbsent方法同样采用锁保护，并创建一个新的大小+1的Object数组。遍历当前Object数组，如Object数组中已有了当前元素，则直接返回，如果没有则放入Object数组的尾部，并返回。从以上分析可见，CopyOnWriteArraySet在add时每次都要进行数组的遍历，因此其性能会低于CopyOnWriteArrayList.
- 示例： DemoThread28

9.1 COW迭代器的弱一致性

在使用CopyOnWriteXXX容器的iterator的时候，实际返回的是COWIterator实例，遍历的数据为快照的数据，其它线程对容器元素增加、删除、修改不对快照产生影响。
对于java.util.concurrent.CopyWriteArrayList、

java.util.concurrent.CopyOnWriteArraySet均适用。

示例：
com.mkevin.demo7.SampleDemo（这个示例演示的是快照的含义）
com.mkevin.demo7.COWDemo0
com.mkevin.demo7.COWDemo1

10. 并发——无阻塞队列

- ConcurrentLinkedQueue并发无阻塞队列，BlockingQueue并发阻塞队列，均实现自Queue接口
- ConcurrentLinkedQueue无阻塞、无锁、高性能、无界、线程安全，性能优于BlockingQueue、不允许null值
- 示例：DemoThread29 (ConcurrentLinkedQueue)

11. 并发——阻塞队列ArrayBlockingQueue

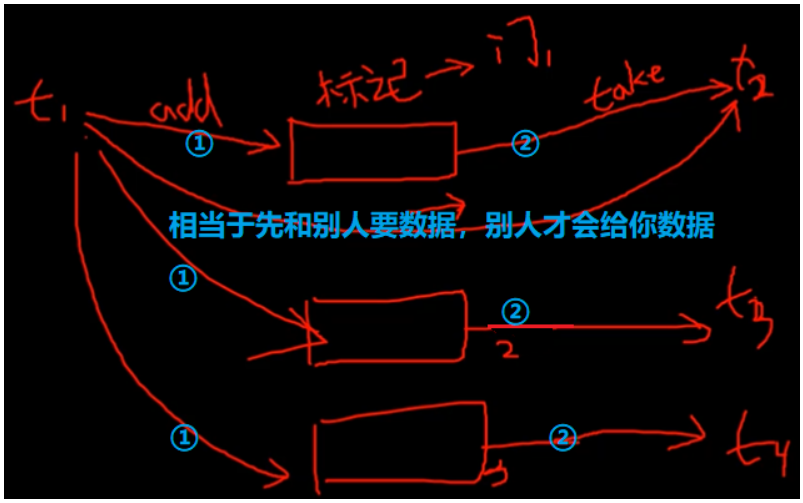
- ArrayBlockingQueue：基于数组实现的阻塞有界队列、创建时可指定长度，内部实现维护了一个定长数组用于缓存数据,内部没有采用读写分离，写入和读取数据不能同时进行，不允许null值
- 示例： DemoThread30

12. 并发——阻塞队列 LinkedBlockingQueue

- LinkedBlockingQueue ：基于链表的阻塞队列,内部维护一个链表存储缓存数据， 支持写入和读取的并发操作， 创建时可指定长度也可以不指定，不指定时代表无界队列， 不允许null值
- 示例： DemoThread31

13. 并发——阻塞队列SynchronousQueue

- SynchronousQueue ：没有任何容量，必须先有线程先从队列中take,才能向queue中add数据，否则会抛出队列已满的异常，可以先使用put方法向队列中添加数组，只是会阻塞住。不能使用peek方法取数据,此方法底层没有实现,会直接返回null
- 方便进行线程间传送数据，效率高，不会出现队列中数据被争抢的问题



- 示例： DemoThread32

14. 并发——阻塞队列 PriorityBlockingQueue

- PriorityBlockingQueue：一个无界阻塞队列，默认初始化长度11，也可以手动指定，但是队列会自动扩容。资源被耗尽时导致OutOfMemoryError。不允许使用 null元素。不允许插入不可比较的对象（导致抛出 ClassCastException）, 加入的对象实现Comparable接口
- 示例： DemoThread33

15. 并发——阻塞队列DelayQueue

- DelayQueue：Delayed 元素的一个无界阻塞队列，只有在延迟期满时才能从中提取元素，底层使用PriorityQueue来实现。该队列的头部是延迟期满后保存时间最长的 Delayed 元素。如果延迟都还没有期满，则队列没有头部， 并且 poll 将返回 null。当一个元素的 getDelay(TimeUnit.NANOSECONDS) 方法返回一个小于等于0 的值时，将发生到期。即使无法使用 take 或 poll 移除未到期的元素，也不会将这些元素作为正常元素对待。例如，size 方法同时返回到期和未到期元素的计数。此队列不允许使用 null 元素。内部元素需实现Delayed接口
- 场景：缓存到期删除、任务超时处理、空闲链接关闭等
- 示例： DemoThread34