学习内容：  
1. 熟悉/util/concurrent包下的相关api使用  
2. 了解Java内存模型和happen—before原则  
3. Executor框架，不同线程池的区别

任务：  
1. 做一个简单的生产者消费者demo, 符合以下原则  
   a. 任务队列长度固定  
   b. 生产者、消费者都是多线程  
   c. 当队列已满，生产者处于阻塞状态，并通知消费者消费  
   d. 当队列为空，消费者处于阻塞状态，并通知生产者生产  
   e. 避免惊群效应？

**并发 Map(映射) ConcurrentMap**

ConcurrentHashMap 和 java.util.HashTable 类很相似，但 ConcurrentHashMap 能够提供比 HashTable 更好的并发性能。在你从中写入或者读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住。它的内部只是把 Map 中正在被写入的部分进行锁定。

在ConcurrentHashMap中，随处可以看到U, 大量使用了U.compareAndSwap的方法，这个方法是利用一个CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。CAS（Compare and Swap）有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

**并发导航映射 ConcurrentNavigableMap**

headMap(T toKey) 方法返回一个包含了小于给定 toKey 的 key 的子 map。

tailMap(T fromKey) 方法返回一个包含了不小于给定 fromKey 的 key 的子 map。

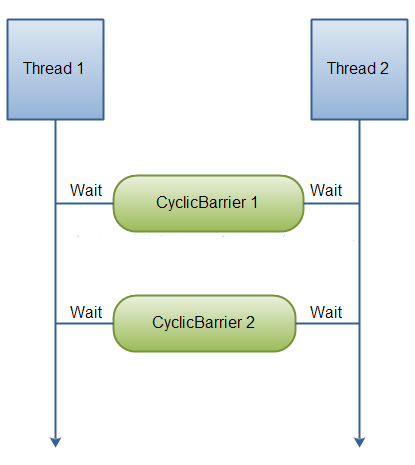
subMap() 方法返回原始 map 中，键介于 from(包含) 和 to (不包含) 之间的子 map。

**计数器CountDownLatch**

CountDownLatch是JAVA提供在java.util.concurrent包下的一个辅助类，可以把它看成是一个计数器，其内部维护着一个count计数，只不过对这个计数器的操作都是原子操作，同时只能有一个线程去操作这个计数器，CountDownLatch通过构造函数传入一个初始计数值，调用者可以通过调用CounDownLatch对象的cutDown()方法，来使计数减1；如果调用对象上的await()方法，那么调用者就会一直阻塞在这里，直到别人通过cutDown方法，将计数减到0，才可以继续执行。

案例地址：https://github.com/ben201708/learn\_java/blob/master/src/com/thread/TestCountDownLatch16.java

**同步工具CyclicBarrier**

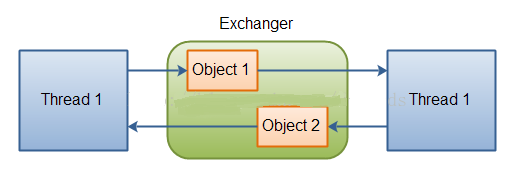


**两个线程在栅栏旁等待对方。**  
通过调用 CyclicBarrier 对象的 await() 方法，两个线程可以实现互相等待。一旦 N 个线程在等待 CyclicBarrier 达成，所有线程将被释放掉去继续运行。

案例地址：https://github.com/ben201708/learn\_java/blob/master/src/com/thread/TestCyclicBarrier15.java

**数据交换 Exchanger**

java.util.concurrent.Exchanger 类表示一种两个线程可以进行互相交换对象的会和点。这种机制图示如下：



**信号灯 Semaphore**

信号量主要有两种用途：

1. 保护一个重要(代码)部分防止一次超过 N 个线程进入。

2. 在两个线程之间发送信号。

案例地址：https://github.com/ben201708/learn\_java/blob/master/src/com/thread/TestSemaphore14.java

**原子性布尔 AtomicBoolean**

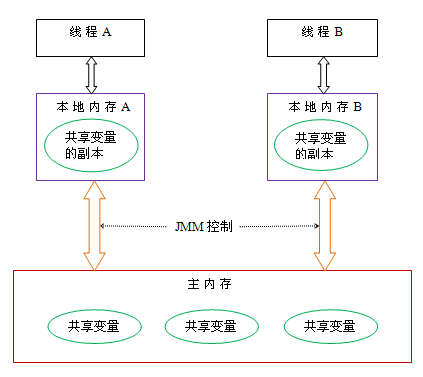
**原子性整型 AtomicInteger**

**原子性长整型 AtomicLong**

**原子性引用型 AtomicReference**

**Java内存模型**

Java线程之间的通信采用的是共享内存模型，这里提到的共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

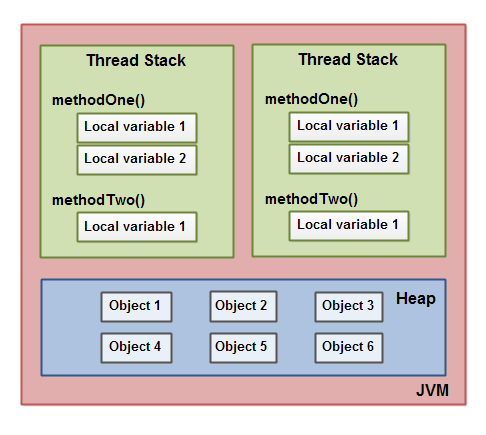


AB线程建通过主内存共享数据过程：



**JVM对Java内存模型的实现**

在JVM内部，Java内存模型把内存分成了两部分：线程栈区和堆区。

下图展示了调用栈和本地变量都存储在栈区，对象都存储在堆区：  


堆中的对象可以被多线程共享。如果一个线程获得一个对象的引用，它便可访问这个对象的成员变量。如果两个线程同时调用了同一个对象的同一个方法，那么这两个线程便可同时访问这个对象的成员变量，但是对于本地变量，每个线程都会拷贝一份到自己的线程栈中。如下图：



**happens-before**

从jdk5开始，java使用新的JSR-133内存模型，基于happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

与程序员密切相关的happens-before规则如下：

程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。

监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。

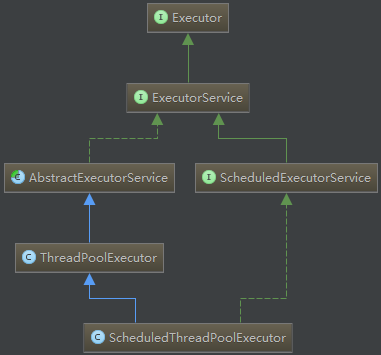
volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

注意：两个操作之间具有happens-before关系，并不意味前一个操作必须要在后一个操作之前执行！仅仅要求前一个操作的执行结果，对于后一个操作是可见的，且前一个操作按顺序排在后一个操作之前。

**Eexecutor**作为灵活且强大的异步执行框架，其支持多种不同类型的任务执行策略，提供了一种标准的方法将任务的提交过程和执行过程解耦开发，基于生产者-消费者模式，其提交任务的线程相当于生产者，执行任务的线程相当于消费者，并用Runnable来表示任务，Executor的实现还提供了对生命周期的支持，以及统计信息收集，应用程序管理机制和性能监视等机制。

Executor的UML图：（常用的几个接口和子类）

****

**ScheduledThreadPoolExecutor**：ScheduledExecutorService的实现，一个可定时调度任务的线程池

**ThreadPoolExecutor**：线程池，可以通过调用Executors以下静态工厂方法来创建线程池并返回一个ExecutorService对象。

**Executors：提供了一系列静态工厂方法用于创建各种线程池**

**①newFixedThreadPool**:创建可重用且固定线程数的线程池，如果线程池中的所有线程都处于活动状态，此时再提交任务就在队列中等待，直到有可用线程；如果线程池中的某个线程由于异常而结束时，线程池就会再补充一条新线程。

案例地址：https://github.com/ben201708/learn\_java/blob/master/src/com/thread

②**newSingleThreadExecutor**:创建一个单线程的Executor，如果该线程因为异常而结束就新建一条线程来继续执行后续的任务。

③**newScheduledThreadPool**:创建一个可延迟执行或定期执行的线程池

④**newCachedThreadPool**:创建可缓存的线程池，如果线程池中的线程在60秒未被使用就将被移除，在执行新的任务时，当线程池中有之前创建的可用线程就重用可用线程，否则就新建一条线程。

**生产者消费者demo**

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/thread2