1. J.U.C
2. 概述

J.U.C是java.util.concurrent，是JSR 166标准规范的一个实现，该框架在JDK1.5引入，包含以下内容：

1. AbstractQueuedSynchronizer(AQS框架)， J.U.C中实现锁和同步机制的基础；
2. Locks&Condition(锁和条件变量)，比synchronized,wait,notify更加细粒度的锁机制；
3. Executor框架（线程池，Callable, Future），任务点的执行和调度框架；
4. Synchronizers(同步器)，主要用于协助线程同步，有CountDownLatch, CyclicBarrier, Semaphore, Exchanger;
5. Atomic Variables(原子变量)，无锁的原子操作，核心是CAS操作，Compare And Swap，将预期值与当变量的值进行比较，如果相等则使用新值替换当前的变量，否则不操作；
6. BlockingQueue(阻塞队列)， 提供了可阻塞的入队和出队操作；
7. Concurrent Collections（并发容器）， JDK1.5之前为了线程安全都使用同步容器，但是效率过低；
8. Fork/join并行计算框架，JDK1.7引入，方便利用多核平台的计算能力，简化并行程序的编写，开发人员仅需关注如何划分任务和组合中间结果。
9. TimeUnit枚举，TimeUnit是java.util.concurrent包下面的一个枚举类，提供了可读性更好的线程暂停操作。
10. CAS
11. 介绍

Compare And Swap，比较并交换，同步组件中大量使用了CAS技术实现Java多线程的并发操作，整个AQS同步组件，Atomic原子类都是以CAS实现的，CAS是整个JUC的基石。

1. 原理分析

CAS没有使用锁，因此性能比synchronized要好很多。它采用三个参数，一个是当前内存值V，旧的预期值A，即将更新的值B，当且仅当旧的预期值A和当前内存值V相同时，才将内存值修改为B并返回true；若当前内存值域旧的预期值不相等时，则不操作，返回false。如果CAS操作失败，则通过自旋的方式等待并再次尝试，直到成功。可以通过以下的实例检验效果：

主内存中i=10，线程A，B拥有的工作内存中都有i的副本，如果线程A进行了i++操作，则i要更新为11，主内存i=10和线程A的内存中i=10比较相等后，判定主内存中值没有修改过，则将A线程要更新的值11更新到主内存中；此时的线程B中i=10，如果此时进行i++ 操作，而主内存中的i=11与B线程中的i=10比较判定不相等，则说明主内存已经被其他线程修改过了，线程B中的值不会更新到主内存中。

CAS会存在一种“ABA”的问题，即主内存之前为10，通过线程A的3次修改，第一次是i+1，第二次是i-1，则又变为了10，B线程此时与主内存比较后判定主内存没有修改过，则B线程会将更新过的值更新到主内存上，其实主内存已经被线程A修改了两次了。我们可以通过变量的修改版本号来解决这个问题，变量每修改过一次，更新一个版本号，如果两个变量的版本号不同，则肯定被修改过，这就能够保证我们每次修改的正确性。

CAS中，直接通过硬件进行内存比较的原子操作，与synchronized对比，少了很多的逻辑步骤，使得性能大为提高。

JUC中的atomic类都是通过CAS实现的，其中使用了CAS的核心类Unsafe，其中调用了unsage类的compareAndSwap方法，该方法为native方法，有四个参数，分别代表对象，对象的地址，预期值，修改值，与CAS操作过程对应。但是，注意，Unsafe是一个不安全的类，主要用于执行低级别的方法集合，尽管这个类和所有的方法都是公开的，但是这个类的应用是受限的，你无法在自己的Java程序中直接使用该类，只有授信的代码才能够获得该类的实例。

1. native关键词

Java是无法直接访问底层操作系统的，但能够调用其他语言编写的函数，通过JNI（Java Native Interface）实现。通过native关键字告诉JVM这个方法是在外部定义的，然后通过javah命令生成.h文件，如编写c语言文件，实现生成头文件中的函数，最后生成dll共享库，Java程序load库，调用这些函数即可。

native可以和除了abstract之外的关键字连用，这也说明了这些方法都是有实体的，并且和其他java方法一样，拥有各种java的特性。native将java和其他语言整合在了一起，特别是与底层系统的交互，得到强大的操作系统的支持，使用一些Java没有封装的操作系统的特性。

1. 多CPU的CAS处理

对于单处理器来说，CAS操作很容易实现，而对于多处理器来说，实现有些复杂。CPU提供了两种方法实现CAS操作：总线加锁和缓存加锁。

总线加锁： 使用处理器提供的一个Lock信号，当一个处理器在总线上输出此信号时，其他处理器的请求将会被阻塞，那么该处理器可以独占共享内存。这样方法效率较低，在阻塞期间，其他处理器和内存之间的通信都被锁住了，甚至都不能处理其他内存地址的数据。

缓存加锁： 缩小锁的范围，保证针对某个内存地址的操作是原子性即可，不会锁住处理器，而仅仅锁住某一块内存，通过缓存一致性协议来保证原子性，仅仅能够被一个处理器修改。而此时其他的处理器与其他的内存都能够这正常地使用。

1. CAS缺陷
2. 循环时间过长

CAS操作失败时，会通过自旋操作等待后继续尝试，直到成功。但是如果CAS操作一直失败呢？那么CAS的自旋会一直进行，消耗大量CPU资源。在JUC中有些位置限制了CAS自旋的次数，如BlockingQueue的SynchronousQueue。

1. 单共享变量的原子性

CAS操作只能针对一个共享变量，对于多个共享变量只能通过锁来解决原子性。

1. ABA问题

一个线程对共享变量进行了两次操作，这两次操作改变了共享变量两次，正好互相抵消了变量的变化，该变量又回到了原来的状态。这时，如果另一个线程通过CAS检查该共享变量，判断为没有被修改过，因此将更新该共享变量的值。但是，实际上该变量已经被修改了两次，这就是ABA问题。

我们可以通过为共享变量添加版本号来解决，每次修改变量，就修改版本号，这样ABA问题就解决了。如Int类型变量，CAS中使用AtomicInteger，会存在ABA问题，Java提供了AtomicStampedReference类来结局，通过包装[E,Integer]的元组来标记对象的版本。

private static AtomicStampedReference air = new AtomicStampedReference(100,1);