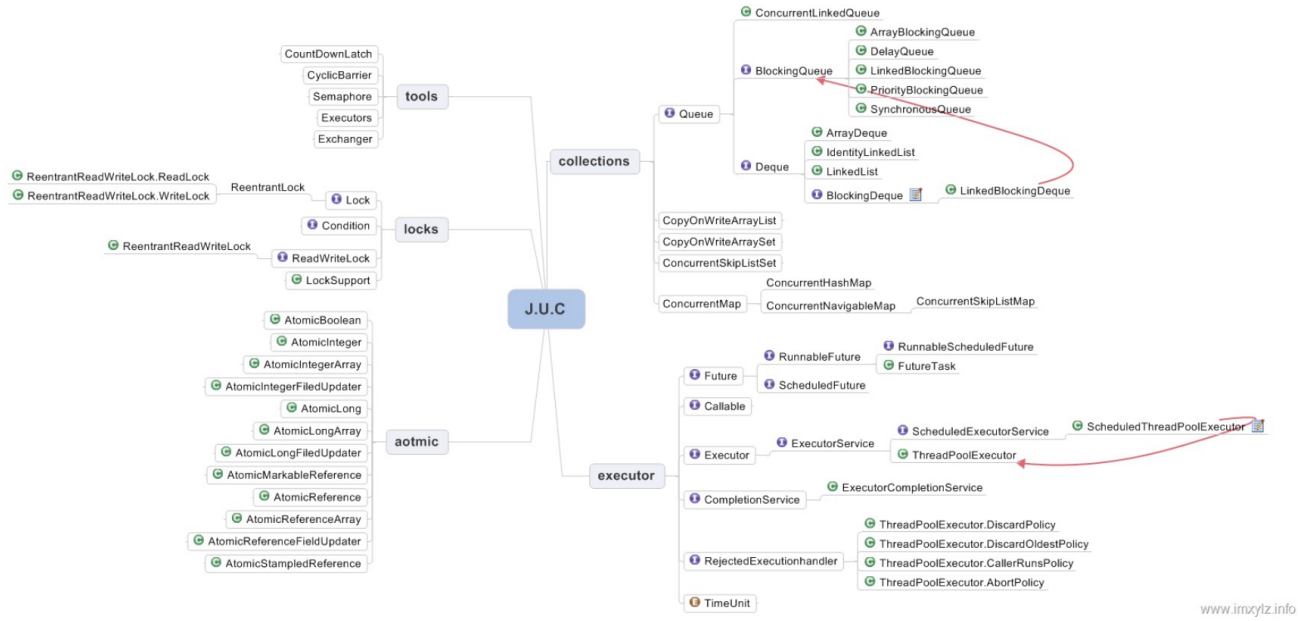
# JUC - 类汇总和学习指南

## 面试问题去理解

- JUC框架包含几个部分?
- 每个部分有哪些核心的类?
- 最最核心的类有哪些?

五个部分:

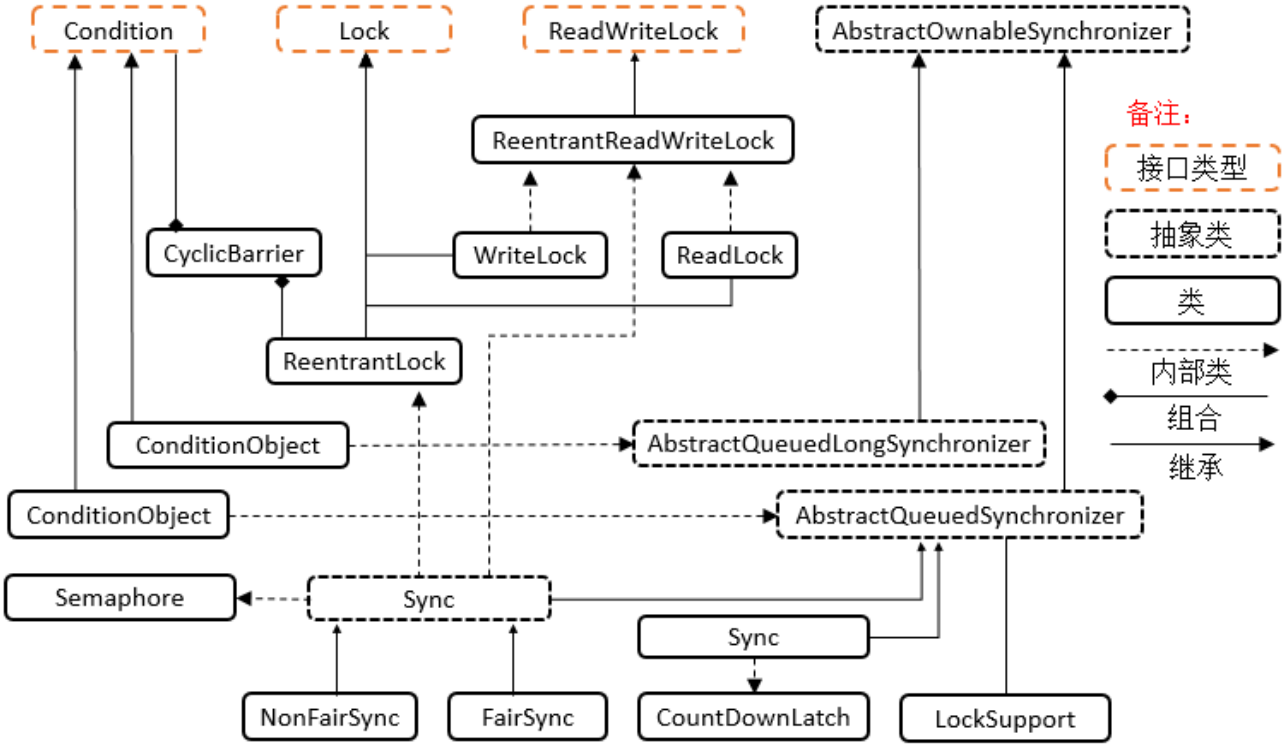



主要包含: (注意: 上图是网上找的图，无法表述一些继承关系，同时少了部分类；但是主体上可以看出其分类关系也够了)

- Lock框架和Tools类(把图中这两个放到一起理解)
- Collections: 并发集合
- Atomic: 原子类
- Executors: 线程池

## Lock框架和Tools类

### 类结构总览




## 接口: *Condition*

> Condition为接口类型，它将 Object 监视器方法(wait、notify 和 notifyAll)分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set (wait-set)。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。可以通过await(),signal()来休眠/唤醒线程。

## 接口: *Lock*

> Lock为接口类型，Lock实现提供了比使用synchronized方法和语句可获得的更广泛的锁定操作。此实现允许更灵活的结构，可以具有差别很大的属性，可以支持多个相关的Condition对象。

## 接口: *ReadWriteLock*

> ReadWriteLock为接口类型， 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

## 抽象类: *AbstractOwnableSynchonizer*

> AbstractOwnableSynchonizer为抽象类，可以由线程以独占方式拥有的同步器。此类为创建锁和相关同步器(伴随着所有权的概念)提供了基础。AbstractOwnableSynchronizer 类本身不管理或使用此信息。但是，子类和工具可以使用适当维护的值帮助控制和监视访问以及提供诊断。

## 抽象类*(long): AbstractQueuedLongSynchronizer*

AbstractQueuedLongSynchronizer为抽象类，以 long 形式维护同步状态的一个 AbstractQueuedSynchronizer 版本。此类具有的结构、属性和方法与 AbstractQueuedSynchronizer 完全相同，但所有与状态相关的参数和结果都定义为 long 而不是 int。当创建需要 64 位状态的多级别锁和屏障等同步器时，此类很有用。

## 核心抽象类*(int): AbstractQueuedSynchonizer*

AbstractQueuedSynchonizer为抽象类，其为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器(信号量、事件，等等)提供一个框架。此类的设计目标是成为依靠单个原子 int 值来表示状态的大多数同步器的一个有用基础。

## 锁常用类*: LockSupport*

LockSupport为常用类，用来创建锁和其他同步类的基本线程阻塞原语。LockSupport的功能和"Thread中的Thread.suspend()和Thread.resume()有点类似"，LockSupport中的park() 和 unpark() 的作用分别是阻塞线程和解除阻塞线程。但是park()和unpark()不会遇到“Thread.suspend 和 Thread.resume所可能引发的死锁”问题。

## 锁常用类*: ReentrantLock*

ReentrantLock为常用类，它是一个可重入的互斥锁 Lock，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。

## 锁常用类*: ReentrantReadWriteLock*

ReentrantReadWriteLock是读写锁接口ReadWriteLock的实现类，它包括Lock子类ReadLock和WriteLock。ReadLock是共享锁，WriteLock是独占锁。

## 锁常用类*: StampedLock*

它是java8在java.util.concurrent.locks新增的一个API。StampedLock控制锁有三种模式(写，读，乐观读)，一个StampedLock状态是由版本和模式两个部分组成，锁获取方法返回一个数字作为票据stamp，它用相应的锁状态表示并控制访问，数字0表示没有写锁被授权访问。在读锁上分为悲观锁和乐观锁。

## 工具常用类: *CountDownLatch*

> CountDownLatch为常用类，它是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

## 工具常用类: *CyclicBarrier*

> CyclicBarrier为常用类，其是一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

## 工具常用类: *Phaser*

> Phaser是JDK 7新增的一个同步辅助类，它可以实现CyclicBarrier和CountDownLatch类似的功能，而且它支持对任务的动态调整，并支持分层结构来达到更高的吞吐量。

## 工具常用类: *Semaphore*

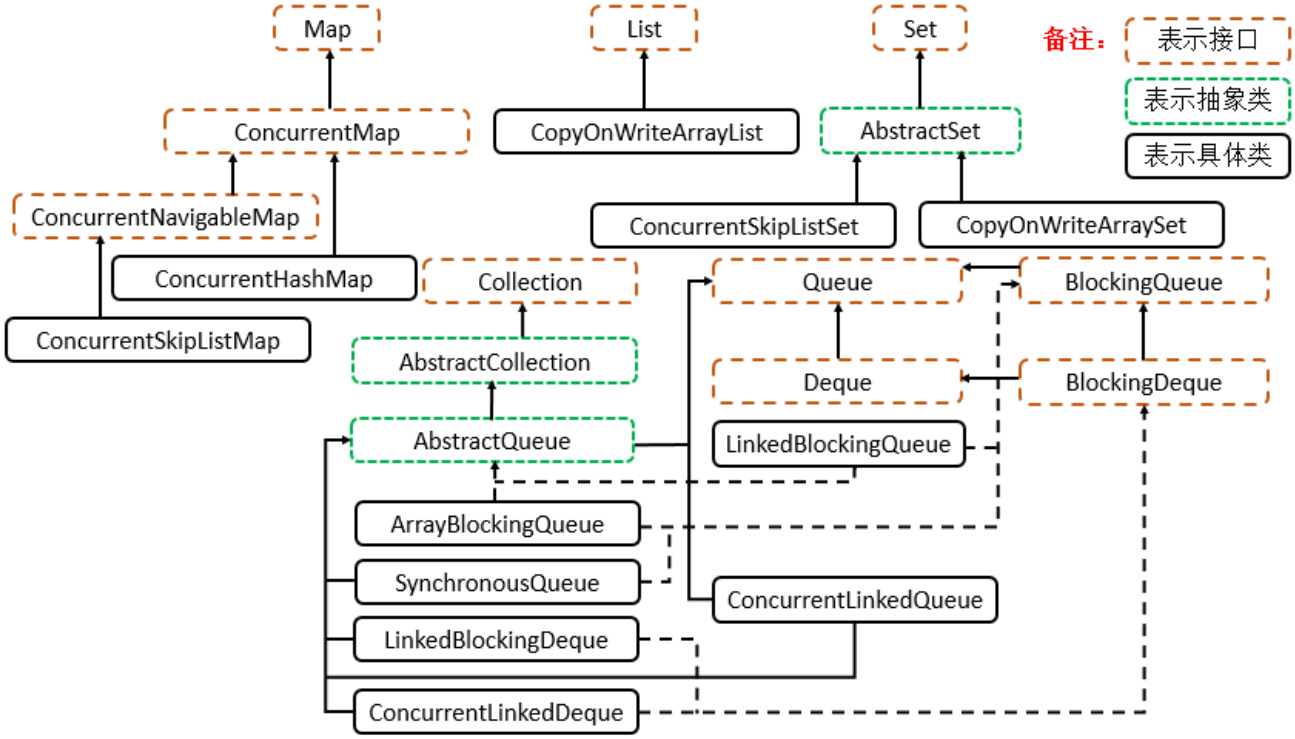
> Semaphore为常用类，其是一个计数信号量，从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。通常用于限制可以访问某些资源(物理或逻辑的)的线程数目。

## 工具常用类: *Exchanger*

> Exchanger是用于线程协作的工具类, 主要用于两个线程之间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange()方法交换数据，当一个线程先执行exchange()方法后，它会一直等待第二个线程也执行exchange()方法，当这两个线程到达同步点时，这两个线程就可以交换数据了。

# Collections: 并发集合

## 类结构关系




## *Queue: ArrayBlockingQueue*

> 一个由数组支持的有界阻塞队列。此队列按 FIFO(先进先出)原则对元素进行排序。队列的头部 是在队列中存在时间最长的元素。队列的尾部 是在队列中存在时间最短的元素。新元素插入到队列的尾部，队列获取操作则是从队列头部开始获得元素。

## *Queue: LinkedBlockingQueue*

> 一个基于已链接节点的、范围任意的 blocking queue。此队列按 FIFO(先进先出)排序元素。队列的头部 是在队列中时间最长的元素。队列的尾部 是在队列中时间最短的元素。新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

## *Queue: LinkedBlockingDeque*

> 一个基于已链接节点的、任选范围的阻塞双端队列。

## *Queue: ConcurrentLinkedQueue*

> 一个基于链接节点的无界线程安全队列。此队列按照 FIFO(先进先出)原则对元素进行排序。队列的头部 是队列中时间最长的元素。队列的尾部 是队列中时间最短的元素。新的元素插入到队列的尾部，队列获取操作从队列头部获得元素。当多个线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。此队列不允许使用 null 元素。

## *Queue: ConcurrentLinkedDeque*

是双向链表实现的无界队列，该队列同时支持FIFO和FILO两种操作方式。

## *Queue: DelayQueue*

延时无界阻塞队列，使用Lock机制实现并发访问。队列里只允许放可以“延期”的元素，队列中的head是最先“到期”的元素。如果队里中没有元素到“到期”，那么就算队列中有元素也不能获取到。

## *Queue: PriorityBlockingQueue*

无界优先级阻塞队列，使用Lock机制实现并发访问。priorityQueue的线程安全版，不允许存放null值，依赖于comparable的排序，不允许存放不可比较的对象类型。

## *Queue: SynchronousQueue*

没有容量的同步队列，通过CAS实现并发访问，支持FIFO和FILO。

## *Queue: LinkedTransferQueue*

JDK 7新增，单向链表实现的无界阻塞队列，通过CAS实现并发访问，队列元素使用 FIFO(先进先出)方式。LinkedTransferQueue可以说是ConcurrentLinkedQueue、 SynchronousQueue(公平模式)和LinkedBlockingQueue的超集, 它不仅仅综合了这几个类的功能，同时也提供了更高效的实现。

## *List: CopyOnWriteArrayList*

ArrayList 的一个线程安全的变体，其中所有可变操作(add、set 等等)都是通过对底层数组进行一次新的复制来实现的。这一般需要很大的开销，但是当遍历操作的数量大大超过可变操作的数量时，这种方法可能比其他替代方法更 有效。在不能或不想进行同步遍历，但又需要从并发线程中排除冲突时，它也很有用。

## *Set: CopyOnWriteArraySet*

对其所有操作使用内部CopyOnWriteArrayList的Set。即将所有操作转发至CopyOnWriteArayList来进行操作，能够保证线程安全。在add时，会调用addIfAbsent，由于每次add时都要进行数组遍历，因此性能会略低于CopyOnWriteArrayList。

### *Set: ConcurrentSkipListSet*

> 一个基于ConcurrentSkipListMap 的可缩放并发 NavigableSet 实现。set 的元素可以根据它们的自然顺序进行排序，也可以根据创建 set 时所提供的 Comparator 进行排序，具体取决于使用的构造方法。

### *Map: ConcurrentHashMap*

> 是线程安全HashMap的。ConcurrentHashMap在JDK 7之前是通过Lock和segment(分段锁)实现，JDK 8 之后改为CAS+synchronized来保证并发安全。

### *Map: ConcurrentSkipListMap*

> 线程安全的有序的哈希表(相当于线程安全的TreeMap);映射可以根据键的自然顺序进行排序，也可以根据创建映射时所提供的 Comparator 进行排序，具体取决于使用的构造方法。

## Atomic: 原子类

其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。实际上是借助硬件的相关指令来实现的，不会阻塞线程(或者说只是在硬件级别上阻塞了)。)

### 基础类型：*AtomicBoolean*，*AtomicInteger*，*AtomicLong*

> AtomicBoolean，AtomicInteger，AtomicLong是类似的，分别针对bool，interger，long的原子类。

### 数组：*AtomicIntegerArray*，*AtomicLongArray*，*BooleanArray*

> AtomicIntegerArray，AtomicLongArray，AtomicBooleanArray是数组原子类。

### 引用：*AtomicReference*，*AtomicMarkedReference*，*AtomicStampedReference*

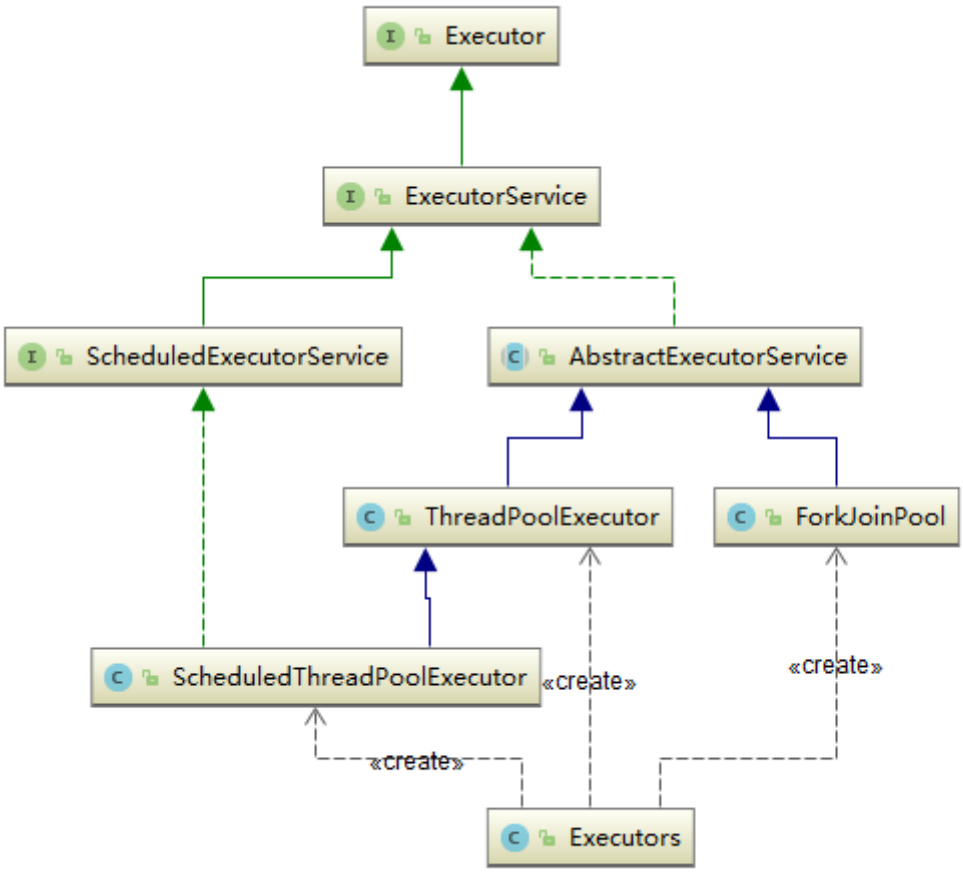
> AtomicReference，AtomicMarkedReference，AtomicStampedReference是引用相关的原子类。

## *FieldUpdater：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater*

> AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater是FieldUpdater原子类。

# Executors: 线程池

## 类结构关系




## 接口: *Executor*

> Executor接口提供一种将任务提交与每个任务将如何运行的机制(包括线程使用的细节、调度等)分离开来的方法。通常使用 Executor 而不是显式地创建线程。

## *ExecutorService*

> ExecutorService继承自Executor接口，ExecutorService提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。 可以关闭 ExecutorService，这将导致其停止接受新任务。关闭后，执行程序将最后终止，这时没有任务在执行，也没有任务在等待执行，并且无法提交新任务。

## *ScheduledExecutorService*

> ScheduledExecutorService继承自ExecutorService接口，可安排在给定的延迟后运行或定期执行的命令。

## *AbstractExecutorService*

> AbstractExecutorService继承自ExecutorService接口，其提供 ExecutorService 执行方法的默认实现。此类使用 newTaskFor 返回的 RunnableFuture 实现 submit、invokeAny 和 invokeAll 方法，默认情况下，RunnableFuture 是此包中提供的 FutureTask 类。

## *FutureTask*

> FutureTask 为 Future 提供了基础实现，如获取任务执行结果(get)和取消任务(cancel)等。如果任务尚未完成，获取任务执行结果时将会阻塞。一旦执行结束，任务就不能被重启或取消(除非使用runAndReset执行计算)。FutureTask 常用来封装 Callable 和 Runnable，也可以作为一个任务提交到线程池中执行。除了作为一个独立的类之外，此类也提供了一些功能性函数供我们创建自定义 task 类使用。FutureTask 的线程安全由CAS来保证。

## 核心*: ThreadPoolExecutor*

> ThreadPoolExecutor实现了AbstractExecutorService接口，也是一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。 线程池可以解决两个不同问题: 由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源(包括执行任务集时使用的线程)的方法。每个 ThreadPoolExecutor 还维护着一些基本的统计数据，如完成的任务数。

## 核心*: ScheduledThreadExecutor*

> ScheduledThreadPoolExecutor实现ScheduledExecutorService接口，可安排在给定的延迟后运行命令，或者定期执行命令。需要多个辅助线程时，或者要求 ThreadPoolExecutor 具有额外的灵活性或功能时，此类要优于 Timer。

## 核心: *Fork/Join*框架

> ForkJoinPool 是JDK 7加入的一个线程池类。Fork/Join 技术是分治算法(Divide-and-Conquer)的并行实现，它是一项可以获得良好的并行性能的简单且高效的设计技术。目的是为了帮助我们更好地利用多处理器带来的好处，使用所有可用的运算能力来提升应用的性能。

## 工具类: *Executors*

> Executors是一个工具类，用其可以创建ExecutorService、ScheduledExecutorService、ThreadFactory、Callable等对象。它的使用融入到了ThreadPoolExecutor, ScheduledThreadExecutor和ForkJoinPool中。