**java集合 :**

* List的实现类主要有: LinkedList, ArrayList, Vector, Stack , 都实现与Collection集合接口, 继承于AbstractList抽象类
  + Vector 是数组实现的矢量队列，它也一个动态数组；不过和ArrayList不同的是，Vector是线程安全的，它支持并发
    - **为什么线程安全?**
      * 因为add, remove等操作加了synchronized关键字同步锁
    - **问Vector的遍历效率 ?**
      * 跟ArrayList一样，因为内部也是用数组实现的, 所以效率还是get最高,迭代起效率最低, 不过Vector还多了一个Enumeration迭代.(这个东西在用在Vector和HashTable里面)
  + Stack是Vector实现的栈；和Vector一样，它也是线程安全的.
    - **为什么是线程安全的?**
      * 因为起pop, peak, push等操作同样增加了synchronized关键字
  + ArrayList 是数组实现的队列，它是一个动态数组；它也不是线程安全的，只适用于单线程
    - **问 ArrayList的遍历效率 ?**
      * get()效率最高, foreach()排中间 , 迭代器效率最低, 这个跟内部实现有关系, 我猜数组的嘛, 如果是单链表, 那肯定是迭代器效率最高
    - 注意 :  toArray()这个接口可能会抛出ClassNotCastException()
  + LinkedList 是双向链表实现的双端队列；它不是线程安全的，只适用于单线程
    - LinkedList的add. get方法使用了二分查找法
    - **问 LinkedList的遍历效率 ?**
      * LinkedList有七种遍历方式
      * removeFist()或removeLast()的效率最高, 但读取数值的话 还是用foreach遍历;
  + Queue队列 接口 , LinkedList实现了该接口
    - add 增加一个元索 如果队列已满，则抛出一个IIIegaISlabEepeplian异常
    - remove 移除并返回队列头部的元素 如果队列为空，则抛出一个NoSuchElementException异常
    - element 返回队列头部的元素 如果队列为空，则抛出一个NoSuchElementException异常
    - offer 添加一个元素并返回true 如果队列已满，则返回false
    - poll 移除并返问队列头部的元素 如果队列为空，则返回null
    - peek 返回队列头部的元素 如果队列为空，则返回null
* **问 ArrayList,LinkedList,Vector使用场景及性能分析 ?**
  + 对于需要快速插入，删除元素，应该使用LinkedList。
  + 对于需要快速随机访问元素，应该使用ArrayList。
  + 对于“单线程环境” 或者 “多线程环境，但List仅仅只会被单个线程操作”，此时应该使用非同步的类。
    - **问 为什么LinkedList的插入删除比ArrayList快 ?**
      * 链表的话只需要插入节点即可, 数组的话 System.arraycopy会移动index之后所有元素
    - **问 为什么LinkedList随机访问慢 ?**
      * 因为用二分查找肯定比直接访问数组元素下标要慢
  + **问 Vector 和 ArrayList的区别 ?**
    - 除了线程安全
    - Vector实现了Enumeration接口, 支持Enumeration遍历
    - ArrayList支持序列化, 但Vector不支持序列化
* 问 Arrays实现快速排序的方式 ?
  + 归并排序
* Set的实现类主要有: HashSet和TreeSet
  + HashSet是一个没有重复元素的集合，它通过HashMap实现的；HashSet不是线程安全的，只适用于单线程
    - **问HashSet实现线程安全有两种方式 ?**
      * 使用CopyOnWriteArraySet , 他是线程安全的HashSet, 其实现方式主要是通过CopyOnWriteArrayList来实现的
      * 使用Collections.synchronizedSet这个接口, 他主要是将增删操作都加了synchronized关键字
        + 注意Collections还提供了map.list.sortedmap线程安全的封装接口
  + TreeSet也是一个没有重复元素的集合，不过和HashSet不同的是，TreeSet中的元素是有序的；它是通过TreeMap实现的；TreeSet也不是线程安全的，只适用于单线程。
    - **问TreeSet线程安全的实现两种方式 ?**
      * ConcurrentSkipListSet,这个东西是通过ConcurrentSkipListMap这个来实现的, 其内部实现是跳表, 非红黑树(ConcurrentHashMap)
      * Collections.synchronizedSortedSet, 所有这个类似api的实现方式都是底层SynchronizedCollection这个来实现的
    - **问如何实现有序的 ?**
      * 其iterator遍历还是调用的TreeMap的遍历, 通过TreeMap实现有序遍历

* Map的实现类主要有: HashMap，WeakHashMap, Hashtable和TreeMap
  + HashMap 是存储“键-值对”的哈希表；它不是线程安全的，只适用于单线程
    - HashMap主要是通过hash算法实现键的存储 ,
    - hash算法精髓是 模运算 和 扰动函数, 但具体原理 我不懂 <https://www.zhihu.com/question/20733617>
      * hash部分，在stackoverflow上也有介绍，使用移位，异或操作之后能使hash值分布较为均匀
    - HashMap的值是一个单向链表 , 即Entry
  + HashMap是一个散列表, 支持序列化
  + HashMap是通过"拉链法"解决Hash冲突的 <http://xiaolu123456.iteye.com/blog/1485349>
    - **问 Hash冲突是什么 ?**
      * key值的object.hashCode值是一样的, 但key的值不一样
    - **拉链法是什么 ?**
      * HashMapEntry他是一个链表, 就会产生一个Entry链
    - 问 HashMap的线程安全实现方式 :
      * Collections.synchronizedMap, synchronized加锁
      * HashTable , 但是key的hash算法不一样
      * ConcurrentHashMap 加锁策略不一样, 这个是分段锁, segment, 效率比synchronized高
* HashTable :
  + 散列表 , 同步的 , put, remove都是synchronized
  + 同HashMap一样 , 也是通过"拉链法"解决Hash冲突
  + Hashtable的key、value都不可以为null , 然则HashMap可以
* WeakHashMap 是也是哈希表(散列表)；和HashMap不同的是，HashMap的“键”是强引用类型，而WeakHashMap的“键”是弱引用类型，也就是说当WeakHashMap 中的某个键不再正常使用时，会被从WeakHashMap中被自动移除。WeakHashMap也不是线程安全的，只适用于单线程。
* TreeMap 也是哈希表，不过TreeMap中的“键-值对”是有序的，它是通过R-B Tree(红黑树)实现的；TreeMap不是线程安全的，只适用于单线程。
* **问  HashMap和Hashtable的不同点 ?**
  + HashMap继承于AbstractMap，而Hashtable继承于Dictionary, Dictionary一般是通过Enumeration(枚举类)去遍历，Map则是通过Iterator(迭代器)去遍历。 然而‘由于Hashtable也实现了Map接口，所以，它即支持Enumeration遍历，也支持Iterator遍历
  + Hashtable线程安全
  + HashMap添加元素时，是使用自定义的哈希算法。Hashtable没有自定义哈希算法，而直接采用的key的hashCode()
* android 有个 SparseArray , ArrayMap, 其实现原理 和 效率分析：　可以直接看结论
  + <http://www.jianshu.com/p/7b9a1b386265>

**juc集合 :** 支持高并发类时 ; 保持API接口的一致性

* List的线程安全类主要是CopyOnWriteArrayList, CopyOnWriteArraySet
  + CopyOnWriteArrayList : 相当于线程安全的ArrayList，它实现了List接口。CopyOnWriteArrayList是支持高并发的
    - 内部数组的实现加了关键字transient . <http://www.cnblogs.com/lanxuezaipiao/p/3369962.html>
      * 问transient的含义 ? java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候,这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化
        + 1）一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。
        + 2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。
        + 3）被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。
      * 注 : 若实现的是Externalizable接口，则没有任何东西可以自动序列化，需要在writeExternal方法中进行手工指定所要序列化的变量，这与是否被transient修饰无关
* “动态数组”机制 - 内部有个“volatile数组”(array)来保持数据,在“添加/修改/删除”数据时，都会新建一个数组，并将更新后的数据拷贝到新建的数组中. 所以涉及到修改数据的操作，CopyOnWriteArrayList效率很低；但是单单只是进行遍历查找的话，效率比较高
* java 1.8 取消了ReentrantLock来保证多线操作, 1.5-1.7的版本还有该锁
* “线程安全”机制 - volatile的内存语义, 保证了多线程的单次读写操作, 读都能获取到最后写入的数据；即对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入
* CopyOnWriteArraySet : 相当于线程安全的HashSet，它继承于AbstractSet类。CopyOnWriteArraySet内部包含一个CopyOnWriteArrayList对象，它是通过CopyOnWriteArrayList实现的
  + 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大
* JUC集合包中Map的实现类包括: ConcurrentHashMap和ConcurrentSkipListMap。
  + ConcurrentHashMap : 是线程安全的哈希表(相当于线程安全的HashMap) , 是通过“锁分段”来实现的，它支持并发
    - **问HashMap, HashTable, ConcurrentHashMap的区别 ?**
      * HashMap是非线程安全的哈希表，常用于单线程程序中
      * Hashtable是线程安全的哈希表，它是通过synchronized来保证线程安全的, 多线程通过同一个“对象的同步锁”来实现并发控制。Hashtable在线程竞争激烈时，效率比较低.因为当一个线程访问Hashtable的同步方法时，其它线程就访问Hashtable的同步方法时，可能会进入阻塞状态
      * ConcurrentHashMap : 通过“锁分段”来保证线程安全的, ConcurrentHashMap将哈希表分成许多片段(Segment)，每一个片段除了保存哈希表之外，本质上也是一个“可重入的互斥锁”(ReentrantLock)。多线程对同一个片段的访问，是互斥的；但是，对于不同片段的访问，却是可以同步进行的。ConcurrentHashMap对并发的控制更加细腻，它也更加适应于高并发场景
* HashEntry, 是单向链表节点，存储着key-value键值对, Segment与HashEntry是组合关系，Segment类中存在“HashEntry数组”成员, ps : 1.8源码则没有HashEntry对象了
* 增加/删除等操作, 都是先通过key的hash值, 找是否存在Segment片段, 如果存在,再在Segment里面去
* 源码一些亮点吧 :
  + trylock获取锁失败,有个scanAndLockForPut自旋函数通过循环遍历获取锁
* 线程安全原理总结 :
  + 对于ConcurrentHashMap的添加，删除操作，在操作开始前，线程都会获取Segment的互斥锁；操作完毕之后，才会释放。而对于读取操作，它是通过volatile去实现的，HashEntry数组是volatile类型的
* ConcurrentSkipListMap : 是线程安全的有序的哈希表(相当于线程安全的TreeMap), 通过“跳表”来实现的，它支持并发
  + 关于跳表(Skip List)，它是平衡树的一种替代的数据结构，但是和红黑树不相同的是，跳表对于树的平衡的实现是基于一种随机化的算法的，这样也就是说跳表的插入和删除的工作是比较简单的
  + HeadIndex是“跳表的表头” , Index是跳表中的索引，它包含“右索引的指针(right)”，“下索引的指针(down)”和“哈希表节点node”
  + 跳跃表Map和TreeMap不是哈希表(HashMap)，博主可能误解了Map(映射表)和HashMap(哈希表)。而且并不是因为跳跃表采用了随机化，就插入和删除简单，而是结构导致的。随机化是用来随机分配节点阶数的，保证整体结构性质。TreeMap因为红黑树结构插入或删除后需要对树进行很大的调整，所以ConcurrentTreeMap实现并没有意义(需要全局锁)，而SkipList变更是局部的更简单
* ConcurrentSkipListSet是线程安全的有序的集合(相当于线程安全的TreeSet), 通过ConcurrentSkipListMap实现的，它也支持并发
* JUC集合包中Queue的实现类包括 : ArrayBlockingQueue, LinkedBlockingQueue, LinkedBlockingDeque, ConcurrentLinkedQueue和ConcurrentLinkedDeque
  + ArrayBlockingQueue : 是数组实现的线程安全的有界的阻塞队列。
    - 内部是数组实现, 通过“互斥锁”保护竞争资源, 有界限的, 按 FIFO（先进先出）原则对元素进行排序
    - 包含一个ReentrantLock，　含两个Condition对象(notEmpty和notFull), 并且是个有限数组
    - 遍历的实现 主要是通过Itr这个类来实现的, 遍历跟ArrayBlockingQueue共享同一把锁
  + LinkedBlockingQueue : 是单向链表实现的(指定大小)阻塞队列，该队列按 FIFO（先进先出）排序元素。
    - Node单向链表 , 内部两把锁, putLock是插入锁，takeLock是取出锁；notEmpty是“非空条件”，notFull是“未满条件”。通过它们对链表进行并发控制
    - 遍历的实现 主要是通过Itr这个类来实现的, 遍历跟LinkedBlockingQueue共享同两把锁 , 他会把插入和提取两个锁同时锁住, 锁分离
    - **问 LinkedBlockingQueue为什么插入\提取是双锁?ArrayBlockingQueue是单锁 ?**<http://www.imooc.com/article/18726>
      * 这个主要是循环队列的原因，主要是数组和链表不同，链表队列的添加和头部的删除，都是只和一个节点相关，添加只往后加就可以，删除只从头部去掉就好。为了防止head和tail相互影响出现问题，这里就需要原子性的计数器，头部要移除，首先得看计数器是否大于0，每个添加操作，都是先加入队列，然后计数器加1，这样保证了，队列在移除的时候，长度是大于等于计数器的，通过原子性的计数器，双锁才能互不干扰。数组的一个问题就是位置的选择没有办法原子化，因为位置会循环，走到最后一个位置后就返回到第一个位置，这样的操作无法原子化，所以只能是加锁来解决。
  + LinkedBlockingDeque : 是双向链表实现的(指定大小)双向并发阻塞队列，该阻塞队列同时支持FIFO和FILO两种操作方式。
    - Node双向链表, 一个互斥锁, 两个condition;
    - FIFO和FILO两种操作方式的实现, 内部其实就是两个Itr的实现
    - **问 LinkedBlockingQueue 内部是两个锁, 但LinkedBlockingDeque内部只有一个独占锁, 为什么 ?**
      * **两个锁的话，并发性就更高一些， 同时读写, 一个独占锁的话, 读写就不能并行;**
      * 优点当然是功能足够强大，同时由于采用一个独占锁，因此实现起来也比较简单。所有对队列的操作都加锁就可以完成。同时独占锁也能够很好的支持双向阻塞的特性。
      * 凡事有利必有弊。缺点就是由于独占锁，所以不能同时进行两个操作，这样性能上就大打折扣。从性能的角度讲LinkedBlockingDeque要比LinkedBlockingQueue要低很多，比CocurrentLinkedQueue就低更多了，这在高并发情况下就比较明显了。
  + ConcurrentLinkedQueue : 是单向链表实现的无界队列，该队列按 FIFO（先进先出）排序元素。
    - 单向链表节点Node, 类型是volatile,
    - 性能最高, 因为volatile的内存语义, 并且peak等操作都是调用Unsafe.CAS
  + 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。
    - 双向链表节点Node, 类型是volatile,
    - 性能最高, 因为volatile的内存语义, 并且peak等操作都是调用Unsafe.CAS

**Android特有的数据结构 :**

* ArrayMap替代HashMap,
  + HashCode冲突的解决办法 :
    - 链地址法, "拉链法"
    - 再Hash法
    - 开放地址法
    - 建立公共溢出区
  + 比HashMap的优点 :
    - 内存占用的优化
    - HashMap内存占用不合理的原因 :
      * hash中为避免退化为数组（如openhft的实现可以退化为数组）或者链表（java.util.HashMap可能退化为链表)使用的空槽
      * java.util.HashMap.Entry的额外占用的内存，用于维持链表、内存对齐等
      * 对象内存占用：在HotSpot 64位jdk中，一个java.lang.Integer占用16字节，一个引用占用4字节，总共20字节，而一个int只占用4字节
      * 扩容会导致扩容损失
    - 链表的缺点 :
      * 链表有什么缺点呢？链表中的每一个节点，存储在内存条的不同物理位置上，这意味着对内存的访问是随机性的，而不是连续性的，而对内存条的随机访问的低效的
      * 连续访问内存只能用数组
  + 缺点 :
    - 使用二分查找法读取的复杂度为O(lgN)
* ArraySet替代HashSet,
  + 比起HashSet在内存使用上更节省
  + 设计跟ArrayMap类似
  + 容器仅包含有单一不重复元素
* SparseArray替代HashMap,
  + 避免自动装箱拆箱,节约这部分内存,可以使用SparseLongArray等等
* 总结 :
  + 1.在数据量小的时候一般认为1000以下，当你的key为int的时候，使用SparseArray确实是一个很不错的选择，内存大概能节省30%，相比用HashMap，因为它key值不需要装箱，所以时间性能平均来看也优于HashMap,建议使用！  
    2.ArrayMap相对于SparseArray，特点就是key值类型不受限，任何情况下都可以取代HashMap,但是通过研究和测试发现，ArrayMap的内存节省并不明显，也就在10%左右，但是时间性能确是最差的，当然了，1000以内的数据量也无所谓了，加上它只有在API>=19才可以使用，个人建议没必要使用！还不如用HashMap放心。估计这也是为什么我们再new一个HashMap的时候google也没有提示让我们使用的原因吧。

**fail-fast 机制 :**

* 是java集合(Collection)中的一种错误机制 , 触发场景多线程对非线程安全集合操作,抛出ConcurrentModificationException异常，产生fail-fast事件
* 解决方式 : JUC集合包里面,  使用volatile, 多线程的增改删, 都会对拷贝数据进行修改在同步到主内存去

**Iterator和Enumeration** :

* Vector、Hashtable实现Enumeration , 这两个数据结构Iterator的实现也是通过Enumeration来实现的, 效率Enumeration遍历更高一些
* Iterator支持fail-fast事件;

**装箱和拆箱机制 :**

* 自动装箱就是Java自动将原始类型值转换成对应的对象，
  + 比如将int的变量转换成Integer对象，这个过程叫做装箱，
  + 反之将Integer对象转换成int类型值，这个过程叫做拆箱
* 产生场景 :
  + 集合里面, 对象类型的参数，转化原始类型值
  + 在循环里面, 错误地声明变量类型, 会导致拆箱过程中, 生成大量无用的对象

ThreadLocal :

* ThreadLocal与线程同步无关，并不是为了解决多线程共享变量问题. ThreadLocal的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度
* Thread - ThreadLoacl - ThreadLocalMap 之间的关系 ?
  + 一个Thread对象, 拥有一个静态的ThreadLocalMap
  + ThreadLocalMap的key默认是一个ThreadLocal实例的弱引用, value则是一个真正的线程局部变量
  + ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value
* ThreadLocal的内存泄漏 :
  + 这个说起来可能会有点绕, ThreadLocal泄漏, 有可能会造成java堆永久代的内存溢出, 因为永久代存储的类相关信息, 越来越大 , 而且你GC回收的是新生代和老年代
  + 原因 :
    - 因为你的key是weakreference嘛, 但是Value始终都在内存里面,都会在java堆里面分配一块内存地址, 除非是跟单独的线程绑定生命周期在一起, 如果是线程池, 线程不会被回收, 那么你就得手动调用remove,在threadlocal使用完毕的时候, 避免内存泄漏
    - 只要这个线程对象被gc回收，就不会出现内存泄露，但在threadLocal设为null和线程结束这段时间不会被回收的，就发生了我们认为的内存泄露
* **同步机制**
* **volatile** - 修饰一个变量 , 具有可见性, 但其操作是不具有原子性的
* **volatile和 synchronize对比** - volatile只是保证了读取的可见性(主内存) , synchronize保证了执行的代码块具有原子性
* **锁** - 锁的释放 , 这个过程实质上是线程A通过主内存向线程B发送消息
* **final** - final的读写在初始化,到变量引用之前, 均不能进行重排序.  final可以修饰方法,表示不能被子类重写,
* **AQS** - 指AbstractQueuedSynchronizer类 , 管理锁的抽象类, 锁的许多公共方法都是在这个类里面实现的. 独占锁ReentrantLock和共享锁Semaphore的公共父类
* **AQS锁的类别** -- 分为“**独占锁**”和“**共享锁**”两种
* **独占锁** - 锁在一个时间点只能被一个线程锁占有. 又分公平锁和非公平锁.
* **公平锁** - 通过CLH等待线程按照先来先得的顺序
* **非公平锁** - 无视CLH等待队列直接获取锁
* 区别 : 在获取锁的机制上不一样, 公平锁，只有在当前线程是CLH等待队列的表头时，才获取锁；而非公平锁，只要当前锁处于空闲状态，则直接获取锁
* **共享锁** - 能被多个线程同时拥有, 能被共享的锁
* **CLH队列** - AQS中“等待锁”的线程队列 , 是一个非阻塞的FIFO队列, 在并发条件先并不会阻塞, 通过自旋锁和CAS节点保证插入和移除的原子性.
* **CAS函数** - compareAndSwap , 是USafe这个类代理实现的. 保证操作的原子性.
* 自旋锁 和 独占锁的 区别 : <http://www.jianshu.com/p/a7f349ddcf82>
* 自旋锁是一种互斥锁的实现方式而已，
* 相比一般的互斥锁会在等待期间放弃cpu，
* 自旋锁（spinlock）则是不断循环并测试锁的状态，这样就一直占着cpu
* 自旋锁使用的场景 :
* <http://www.jianshu.com/p/9ff426a784ad>
* 同步锁机制锁保证的"**先行发生原则(happen before)**"过于的粗力度，它虽然可以保证线程T1的操作如果早于线程T2获取锁，那么T1一定在T2之前完成操作；
* 而CAS操作却不能保证这样的顺序的一致性，但是CAS操作保证了关键的修改一步具有先行发生原则

**Lock** -

* 公平锁
* 非公平锁
* 重入锁

**共享锁** - 多个线程可共享的锁, 一个线程获取该锁的状态, 会去唤醒其他线程同样有该锁的其他线程

**独占锁** - 一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁

**ReentrantLock** - 独占锁 , 可重入的锁

* "公平的ReentrantLock"是指"不同线程获取锁的机制是公平的"
* "非公平的　　ReentrantLock"则是指"不同线程获取锁的机制是非公平的"

ReentrantLock 独占锁 配合 **Condition** :  用法 看JDK Condition 源码  和  <http://www.cnblogs.com/skywang12345/p/3496101.html>

**公平锁** - (FairSync 来实现的)

* AQS -- AbstractQueuedSynchronizer
* AQS锁的类别(独占锁) -- 锁在一个时间点只能被一个线程锁占有
* AQS锁的类别(共享锁) -- 锁在一个时间点只能被一个线程锁占有
* CLH是一个非阻塞的 FIFO 队列 , 内部还是由Node链表节点构成的

**非公平锁** - (NonfairSync 来实现的)

**Condition** -

     Condition的作用是对锁进行更精确的控制。await(), signal(),signalAll()

     Object中的wait(),notify(),notifyAll()方法是和"同步锁"(synchronized关键字)捆绑使用的；

     而Condition是需要与"互斥锁"/"共享锁"捆绑使用的。

     能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition.

     Condition能够更加精细的控制多线程的休眠与唤醒, Object的notify()或notifyAll()不能明确的指定唤醒,只能唤醒synchronized锁住的对象的线程,通过Condition，就能明确的指定唤醒读线程

**LockSupport** -

     LockSupport中的park() 和 unpark() 的作用分别是阻塞线程和解除阻塞线程; 区别于wait() 和 notify() , 是因为wait()在线程阻塞前, 必须通过synchronized获取同步锁. 而LockSupport则不用synchronized锁,

**ReentrantReadWriteLock** -

     共享锁 , 读是共享锁, 写是独占锁

     区别是 : 共享锁, 在线程未运行完的时候, 其他线程也可以获取该共享锁, 并且跑起来.

                 独占锁是,必须等待独占锁线程跑完了, 其他线程才可以继续获得该锁

**CountDownLatch** - (对多线程的控制)

     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.

**CyclicBarrier** - (对多线程的控制)

     CyclicBarrier是一个同步辅助类，允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。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

     CyclicBarrier是通过ReentrantLock(独占锁)和Condition来实现的

**CountDownLatch和CyclicBarrier区别**：

* CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。
* CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。

**Semaphore** - (对多线程的控制)

     Semaphore是一个计数信号量, 信号量维护了一个信号量许可集。线程可以通过调用acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可.

从java源码来看 :

* Executor , ExecutorService为接口 ,
* 具体的实现在AbstractExecutorService,
* 还有个抽象类是ScheduledExecutorService, 主要是功能是提供"延时"和"周期执行"
* Executor是, 静态工厂类, 主要作用是创建线程池对象

ThreadPoolExecutor :

* 存放一定数量线程的一个线程集合
* 当添加的到线程池中的线程超过它的容量时，会有一部分线程阻塞等待。线程池会通过相应的调度策略和拒绝策略，对添加到线程池中的线程进行管理
* 内部参数 :

               workers : 继承AbstractQueuedSynchronizer, 属于一个线程集合, 因此该线程的操作则具有原子性

               workQueue : 即它是一个阻塞队列。当线程池中的线程数超过它的容量的时候，线程会进入阻塞队列进行阻塞等待

               mainLock : 是互斥锁，通过mainLock实现了对线程池的互斥访问 , 也就是说线程池中断等操作具有原子性, 同步

               ThreadFactory :   ThreadFactory对象。它是一个线程工厂类，"线程池通过ThreadFactory创建线程" , 类似于Rx,glide的线程创建都是继承于他的子类;ThreadFactory的作用就是提供创建线程的功能的线程工厂

               handler : RejectedExecutionHandler 所谓拒绝策略，是指将任务添加到线程池中时，线程池拒绝该任务所采取的相应策略

当线程池中的任务数量 < "核心池大小"时，即线程池中少于corePoolSize个任务 , 创建任务

当线程池中的任务数量 >= "核心池大小"时 , 则尝试将任务添加到阻塞队列中

函数 :

submit()实际上也是通过调用execute()实现的

**线程池的5种状态是：**

**Running**, **SHUTDOWN**, **STOP**, **TIDYING**, **TERMINATED**。

1. RUNNING

(01) 状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。

(02) 状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态！

道理很简单，在ctl的初始化代码中(如下)，就将它初始化为RUNNING状态，并且"任务数量"初始化为0。

2. SHUTDOWN

(01) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。

(02) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

3. STOP

(01) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。

(02) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

4. TIDYING

(01) 状态说明：当所有的任务已终止，ctl记录的"任务数量"为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

(02) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

5. TERMINATED

(01) 状态说明：线程池彻底终止，就变成TERMINATED状态。

(02) 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

RejectedExecutionHandler :

是指当任务添加到线程池中被拒绝，而采取的处理措施

第一，线程池异常关闭。第二，任务数量超过线程池的最大限制

1. DiscardPolicy : 丢弃

2. DiscardOldestPolicy : 丢弃老的

3. AbortPolicy : 这个策略类,直接会抛出异常

4. CallerRunsPolicy : 当有任务添加到线程池被拒绝时，线程池会将被拒绝的任务添加到"线程池正在运行的线程"中取运行

**Callable :**

* 当我们需要获取线程的执行结果时，就需要用到它们。Callable用于产生结果，Future用于获取结果
* **Comparable 和 Comparator 比较** :
  + 若一个类实现了Comparable接口，就意味着“该类支持排序”
  + Comparator 就是一个比较器 , 重写compare函数
  + 这两个一般配合Collections.sort(List<T> list, Comparator<? super T> c) 来实现
* **hashCode() 和 equals() :**
  + equals() :
    - == : 它的作用是判断两个对象的地址是不是相等
    - equals() : 它的作用也是判断两个对象是否相等; 重写则是对象内容是否相等
* hashCode() :
  + hashCode() 在散列表中才有用，在其它情况下没用. (Java集合中本质是散列表的类，如HashMap，Hashtable，HashSet)
  + 作用 : 获取散列码；获取散列码作为"键"存储值
  + 若两个元素相等，它们的散列码一定相等；但反过来确不一定。在散列表中，
    - 1、如果两个对象相等，那么它们的hashCode()值一定要相同；
    - 2、如果两个对象hashCode()相等，它们并不一定相等。
* **Annotation注解 :**
  + RetentionPolicy(保留策略) :
    - SOURCE，则意味着：Annotation仅存在于编译器处理期间，编译器处理完之后，该Annotation就没用了
    - CLASS，则意味着：编译器将Annotation存储于类对应的.class文件中，它是Annotation的默认行为
    - RUNTIME，则意味着：编译器将Annotation存储于class文件中，并且可由JVM读入
  + ElementType(类属性) : 修饰变量, 方法, 构造函数等
  + @Documented修饰该Annotation，则表示它可以出现在javadoc中
* 作用 :
  + Annotation具有“让编译器进行编译检查的作用”
  + 根据Annotation生成帮助文档
  + 能够帮忙查看查看代码
  + 我们可以在反射中解析并使用Annotation(runtime 类型 能够被jvm读入)
* **Java 反射 :**根据“类的部分已经的信息”来还原“类的全部的信息”
  + 常用获取Class对象的方法
    - Class.forName
    - 类名.class
    - 实例对象.getClass()
    - "类名字符串".getClass()
* **java序列化 与 反序列化** :
  + 序列化 : 保存对象的状态 , 仅支持保存/恢复类的成员变量，但不支持保存类的成员方法
  + 反序列化 : 把保存的对象状态再读出来
  + 使用场景 :
    - 把的内存中的对象状态在 一个文件中或者数据库中 写入或者读取的时候
    - 用套接字在网络上传送对象的时候
    - 想通过RMI传输对象的时候
  + 序列化对static和transient变量，是不会自动进行状态保存的
  + 于Socket, Thread类，不支持序列化
  + 实现序列化的两种方式 : Serializable / Externalizable
    - Externalizable的实现 :
      * 必须实现writeExternal()和readExternal()接口
      * 必须定义不带参数的构造函数