**ReentrantLock**

abstract static class Sync extends AbstractQueuedSynchronizer {}

static final class NonfairSync extends Sync {}

static final class FairSync extends Sync {}

lock操作：lock() --fail> AQS. acquire

unlock操作：AQS. Release

**AbstractQueuedSynchronizer**

static final class Node {}

public class ConditionObject implements Condition, java.io.Serializable {}

链表（head，tail）

state（持有锁的线程个数）

acquire操作：tryAcquire（由子类实现）--fail> 添加到CLH队列阻塞

release操作：tryRealse（由子类实现）--success> 唤醒CLH队列中第一个非取消状态的结点

**ArrayBlockingQueue**

extends AbstractQueue<E> implements BlockingQueue<E>

int takeIndex;

int putIndex;

int count;

final Object[] items;

final ReentrantLock lock;

private final Condition notEmpty;

private final Condition notFull;

它是一个环形的队列FIFO，putIndex表示当前有多少个数据，takeIndex表示从队列拿出来的数量。



插入移除时，首先使用ReentrantLock进行加锁。

以插入为例，若队列已满，则add抛出异常，offer返回false，put会调用notFull.await()，加入到notFull的等待队列，直到移除元素后调用notFull.signal()唤醒。

condition内部维护了一个条件队列，调用await()时，会添加到条件队列中，并释放ReentrantLock。被唤醒时，会被移动到AQS的同步队列中等待获取ReentrantLock。

**ThreadPoolExecutor**

extends AbstractExecutorService

构造函数参数：1、corePoolSize：核心线程数 2、maximumPoolSize：最大线程数 3、keepAliveTime：非核心线程空闲存活时间 4、unit：时间单位 5、workQueue：阻塞队列 6、threadFactory：线程工厂，创建线程 7、handler：线程池已满的处理策略。

1、线程池接收任务，直到达到corePoolSize（核心线程不会被销毁）。

2、此时到达的任务会添加到workQueue等待处理。

3、若workQueue放不下了，则会创建新线程执行任务，直到达到maximumPoolSize（核心线程外的线程空闲时间达到了keepAliveTime会被销毁）。

4、若还有任务则会调用handler来处理。

拒绝策略：1、AbortPolicy：直接抛出异常（默认） 2、CallerRunsPolicy：直接调用run方法，阻塞执行 3、DiscardPolicy：直接丢弃 4、DiscardOldestPolicy：丢弃阻塞队列队首任务

阻塞队列：ArrayBlockingQueue、DelayQueue、LinkedBlockingQueue：容量默认为Integer.MAX\_VALUE、PriorityBlockingQueue、SynchronousQueue：容量为1。

**HashMap**

extends AbstractMap<K,V> implements Map<K,V>

table生成为延时加载，在插入时才生成。

解决hash冲突的方法为链地址法，由邻接表实现（数组+链表）；

在java8中，若链表长度大于8则会将线性链表转换为红黑树，在hash冲突严重时，以此来加快查询速度。

Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。

当hashmap中的键值对数量大于threshold时，会触发resize()。Java7与java8中略有不同。

在Java7中进行resize操作时，在多线程环境下可能出现死循环的链表，进行对应get操作时会进入死循环。在java8中，resize操作在移动链表元素时，与java7中一个个移动到新table不同，它会将整个链表（需要换bucket的和不需要换的分开）全取出来后，再一次性移动。

**ConcurrentHashMap**

extends AbstractMap<K, V> implements ConcurrentMap<K, V>

**java7**

static final class Segment<K,V> extends ReentrantLock

实现原理：Segment<K,V>[] + HashEntry<K,V>[]。先定位到segment，再定位到HashEntry。

put()：通过ReentrantLock对每个Segment加锁，实现线程安全。加锁时会调用tryLock()尝试获取锁，若失败则调用lock()。尝试最大次数由cpu个数决定，若cpu个数大于1则为64次，否则为1次。

构造方法：public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {}。

Segment数组大小由concurrencyLevel决定，大于concurrencyLevel的最小的2^n，默认为16。HashEntry数组大小为大于initialCapacity/Segment[].size()的最小2^n，最小为2。扩容阈值为HashEntry[].size()\*loadFactor。

进行扩容时是对单个segment对应的HashEntry进行扩容。

size()方法：遍历所有segment获取map中元素个数和modCount，获取两次，并比较两次modCount是否相同；若不相同则重试一次，比较第二次和第三次的modCount，若还不相同则对所有segment加锁遍历。

**java8**

抛弃了segment分段的方法，采用CAS操作及synchronize来保证并发的安全性。1、锁的粒度更细。2、synchronize解锁唤醒所有线程，reentrantLock（非公平）唤醒下一个等待线程及同一时间涌入的线程，在并发量低的情况下，synchronize因为偏向锁的存在甚至更优秀。

put()：1、若Node为null，采用CAS操作插入数据。2、若正在扩容则调用helpTransfer()帮助扩容。3、synchronize对第一个node加锁，插入数据。

transfer()扩容：保存新数组nextTable的引用。oldTable已移动完成的节点做标记。（待补充）

size()：java8中引入了CounterCell[]和baseCount成员变量。插入元素后，会使用CAS操作增加baseCount值，若失败则new CounterCell添加到CounterCell[]中。计算size时，遍历CounterCell[]累加CounterCell.value，并加上baseCount得到结果。

**线程的状态：**

NEW：新建。Thread对象的状态。

RUNNABLE：就绪&运行中。就绪，等待分配时间片，获得时间片后变为运行中。

BLOCKED：阻塞。被动等待，直到获得锁，才会进入RUNNABLE状态。

WAITING：等待。主动等待，直到被唤醒。

TIMED\_WAITING：超时等待。主动等待，直到被唤醒或超时。

TERMINATED：终止。线程已执行完毕。

**死锁的必要条件：**

1、互斥：独占锁。

2、不可强行占有：只能由拥有锁的线程解锁。

3、占有等待：进入阻塞状态时，拥有的锁不会释放。

4、循环等待：若干进程之间形成一种头尾相接的循环等待资源关系。

**synchronize的优化：**

**Java6引入了偏向锁及轻量级锁。**

偏向锁：偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁，降低获取锁的代价。获取锁时检查线程的ID与对象头(Mark Word, Klass Pointer)中的线程ID是否一致；若一致则无需CAS来抢占锁，若不一致则检查对象头部的线程是否存活；若没有存活，则通过CAS操作来抢占锁，使其重新偏向本线程，若存活则升级为轻量级锁。

轻量级锁：自旋，不会阻塞。超过一定次数升级为重量级锁。

**其它优化：**

适应性自旋：若线程自旋获取锁成功，则下次的自旋次数增加。

锁粗化：将多次连接在一起的加锁解锁操作合并为一次，扩大锁的范围，如StringBuffer连续append。

锁消除：如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么认为这段代码是线程安全的，则会删除锁。

**与ReentrantLock的区别：**

ReentrantLock：支持响应中断、超时、尝试获取锁，灵活；依赖AQS；显示调用unlock；支持公平锁&非公平锁；

Synchronize：不灵活；采用监视器模式；自动释放锁；只支持非公平锁。