JUC

笔记本:	工作笔记		
创建时间:	2020/10/30 10:37	**更新时间:**	2020/11/3 21:42
作者:	438842220@qq.com		
URL:	about:blank		

JAVA.UTIL.CONCURRENT

一些概念:

安全失败和快速失败：Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败的，而java.util.concurrent包下面的所有的类都是安全失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。
UNSAFE类：https://www.cnblogs.com/pkufork/p/java_unsafe.html
内存屏障：https://blog.csdn.net/breakout_alex/article/details/94379895 这部分属于JVM相关知识。
Synchronized: https://www.cnblogs.com/paddix/p/5367116.html
看完上面的，接wait/notify:https://www.cnblogs.com/Draymonder/p/9607670.html
volatile :https://www.cnblogs.com/zhengbin/p/5654805.html

注：以下绿色为接口，蓝色为类

Runnable/Callable
前一个没有返回值，不能抛异常，后一个相反
Executor
excute一个Runnable对象
ExecutorService
它比executor更广泛，提供了一系列生命周期管理方法。接受Runnable和Callable对象，返回Futrure。因此一般用这个接口管理和实现多线程。
AbstractExecutorService
它是ExecutorService的默认实现，同时它是一个抽象类。
ScheduledExecutorService
它是一个提供定时调度的接口。
ForkJoinPool
java7引入的ForkJoin框架，同时引入了一种新的线程池ForkJoinPool。
展开讲讲fork/join。见示例代码：jucDemo.calculator
ThreadPoolExecutor
继承并实现了AbstractExecutorService的功能。
入参包括:
corePoolSize/maximumPoolSize/keepAliveTime/unit/workQueue
Executors的四种线程池都是从这里来的。
ScheduledThreadPoolExecutor
正常情况下，定时器我们都是用Timer和TimerTask这两个类就能完成定时任务，并且设置延长时间和循环时间间隔。
ScheduledThreadPoolExecutor也能完成Timer一样的定时任务，并且时间间隔更加准确。
Executors
包含五种实现:

newFixedThreadPool/newWorkStealingPool/newSingleThreadExecutor/newCachedThreadPool/newScheduledThreadPool
分别用于:
定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
（1.8新增）这个线程池不会保证任务的顺序执行，也就是 WorkStealing 的意思，抢占式的工作。
单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO/LIFO/优先级)执行。
可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
定长线程池，支持定时及周期性任务执行。
BlockingQueue
继承了Queue接口
阻塞队列一共有四套方法分别用来进行insert、remove和examine，当每套方法对应的操作不能马上执行时会有不同的反应，下面这个表格就分类列出了这些方法:

-	Throws Exception	Special Value	Blocks	Times Out
Insert	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)

```
        Remove  | remove(o)        poll()          take()     poll(timeout, timeunit)
        Examine | element()       peek()

        ThrowsException：如果操作不能马上进行，则抛出异常
        SpecialValue：如果操作不能马上进行，将会返回一个特殊的值，一般是true或者false
        Blocks:如果操作不能马上进行，操作会被阻塞
        TimesOut:如果操作不能马上进行，操作会被阻塞指定的时间，如果指定时间没执行，则返回一个特殊值，一般是true或者false
         需要注意的是，我们不能向BlockingQueue中插入null，否则会报NullPointerException。
        SynchronousQueue
            队列内部仅允许容纳一个元素。当一个线程插入一个元素后会被阻塞，除非这个元素被另一个线程消费。
        PriorityBlockingQueue
            底层数据结构是最小二叉堆。支持优先级。同PriorityQueue一样，可以自定义实现compareTo方法来指定元素排序规则。
        LinkedBlockingQueue/ArrayBlockingQueue
            类似LinkedList和ArrayList。一个基于链表，一个基于数组。
            链表的数据结构：单向指针，记录头结点尾节点。
        DelayQueue
            是一个无界的BlockingQueue，用于放置实现了Delayed接口的对象，其中的对象只能在其到期时才能从队列中取走。
            队列有序，即队头对象的延迟到期时间最长。
            实现例子：下单之后如果三十分钟之内没有付款就自动取消订单。
        BlockingDeque
            双端队列。继承了BlockingQueue。
            LinkedBlockingDeque
                BlockingDeque的实现。和LinkedBlockingQueue不同的是，双端均可出入。
        TransferQueue
            java7中加入。参考文章1 参考文章2
                LinkedTransferQueue
                    是以上接口的实现类。主要方法：transfer/tryTransfer
                    分别的作用是：若有空闲消费者就移交，否则将元素插到队尾，同时将线程阻塞。/尝试移交，若不可就返回FALSE
                     （tryTransfer这个方法还有重载，增加timeout作为阻塞时间）
    Delayed
        这个接口只有一个方法，getDelay，顾名思义。
    Future
        这个接口提供了以下几个方法：cancel/isCanceled/isDone/get(有timeout和无timeout)
        ScheduledFuture
        在Future的基础上加入了Delayed和Comparable接口。
        RunnableFuture
        在Future的基础上加入了Runnable接口。
            RunnableScheduledFuture
                顾名思义。
            FutureTask
                它实现了RunnableFuture接口。因此它既是Future的实现，又是Runnable的实现。可以交给Executor的实现类执行。
            ForkJoinTask
                它是一个抽象类。以下三个抽象类继承了它。
                CountedCompleter
                    待完成
                RecursiveAction
                    无返回值
                RecursiveTask
                    有返回值
    ConcurrentMap
        它继承了Map。新增了四个方法。putIfAbsent/remove/replace(重载)，含义见下：
            如果插入的key相同，则不替换原有的value值。（原替换）
            新remove方法中增加了对value的判断，如果要删除的key--value不能与Map中原有的key--value对应上，则不会删除该元素；
            replace (K,V,V)：增加了对value值的判断，如果key--oldValue能与Map中原有的key--value对应上，才进行替换操作；
            replace(K,V)：与上面的replace不同的是，此replace不会对Map中原有的key--value进行比较，如果key存在则直接替换；
        ConcurrentHashMap
            ConcurrentMap的哈希表实现。
            一篇不错的参考：https://www.cnblogs.com/lfs2640666960/p/9621461.html
```

一些其他相关的：
HashTable在Java1.1就已经出现了，使用同步的方法来实现线程安全，Hashtable里的方法都是Synchronized的，这使得它在线程增多的时候会变得很慢。 Synchronized Map和 Hashtable并没有太大的不同，以及它在并发编程的时候提供了相似的性能。hashtable 和 Synchronized Map之间的唯一不同在于后者可以使用`Collections.synchronizedMap()`方法把任何的Map变成同步版本。
1.7 ref-link: https://my.oschina.net/7001/blog/896587
segment数组+hashEntry数组，每个hashEntry是一个链表。
segment继承了重入锁，尝试获取会存在并发竞争自旋阻塞。
构造函数入参： initialCapacity/loadFactor/concurrencyLevel(初始容量/负载因子/并发度default:16/0.75/16)
底层使用了UNSAFE类使得非volatile类具有了valatile的语义。 ConcurrentHashMap利用了这些高性能的原子读写来避免加锁带来的开销，提高了性能。
方法：
ConcurrentHashMap的get，containsKey，clear，iterator 都是弱一致性的。get和containsKey是因为原子读之后的遍历过程中可能会加入新的数据。clear是因为删除一个后可能被加入了新的数据。迭代器是因为安全失败。
如果想要强一致性，需要Collections.synchronizedMap()。
1.8
数组链表红黑树
CAS+synchronized
方法实现：https://www.cnblogs.com/ylspace/p/12726672.html
对以上版本方法的总结：
扩容：
1.7 segment数目固定，锁segment。
触发条件是单个segment到达阈值（capacity * loadfactor）
扩容方法是new一个双倍大小的hashEntry数组。然后把数据放进去。
在此期间其他segment独立。get/put/remove不受影响。
size：先乐观锁：遍历加segment元素数据，核对修改次数，如果不变结果正确。如果变动重试。重试次数达到一定程度，全锁计算。
1.8 扩容时锁整张表
触发条件是整个到达阈值。
在一个线程发起扩容的时候，修改volatile变量sizeCtl：-1初始化 -N n-1线程在扩容 未初始化表示table需要初始化的大小 初始化后表示阈值
触发条件是当map长度小于64，链表长度到8扩容，否则链表扩成红黑树。
单线程创建双倍大小的表。然后多线程复制旧表到新表。
不会起冲突：每个线程被分配的一部分表。线程执行的时候锁住自己的table，别人无权访问。已完成的部分会标记。未完成的部分别人做完就可以帮忙。
结束的时候刚好吧sizeCtl置0，最后一个线程将sizeCtl设置完毕，并将table指向新的地址。
get/put在扩容时：get未迁移完成（单线程创建表的时候）会正常读。已迁移完成会来帮助扩容。
put/remove未迁移完成会阻塞，已迁移完成会帮助扩容。
对于size和迭代器是弱一致性
volatile修饰的数组引用是强可见的，但是其元素却不一定，所以，这导致size的根据sumCount的方法并不准确。
同理Iteritor的迭代器也一样，并不能准确反映最新的实际情况
红黑树相关：待补充

countdownlatch
给定计数器，计数器至0则触发。
semaphore
操作系统中的信号量
cyclicbarrier
N个线程相互等待，而且可以循环
其他
Lock https://blog.csdn.net/qq_36974281/article/details/81986973#lock-juc%E9%94%81
重入锁和不可重入锁的区别：https://blog.csdn.net/qq_29519041/article/details/86583945
ReentrantLock
既可以是公平锁又可以是非公平锁，在初始化的时候设置。相比于synchronized需要显示设置加锁和解锁。因此被称为显式锁。synchronized被称为隐式锁。
AQS
它是一个抽象类。由一个volatile修饰的int state和一个双端双向队列组成。多线程阻塞时进入队列。
它支持共享模式和独占模式。
使用了模板方法，不同的自定义使用者继承并重写指定的方法。
JUC包中许多类都是继承AQS来实现的。

synchronized的锁升级过程：轻量锁/偏向锁/自旋锁/重量锁
众所周知开始的时候对象是没有被访问的，因此是无锁状态。
在对象第一次被访问之后，无锁升级为偏向锁。偏向锁里存着获取锁的线程ID，并且是否偏向标志的为1。
在偏向锁遇到冲突之后，偏向锁撤销，升级为自旋锁。用CPU空转来取代上下文开销。
在自旋多次失败后，升级为重量锁。即jdk1.6之前的synchronized。
拓展：identityHashCode和偏向锁/自旋锁/重量锁 https://www.imooc.com/article/273515

Atomic
CAS（compareAndSet）
它是一条原子操作。功能是比较并交换。用来实现原子性。UNSAFE里面有CAS的实现。
Atomic正是通过UNSAFE提供的实现实现原子性/用volatile来实现可见性。
一共有12个类。分别是：
AtomicBoolean/AtomicInteger/AtomicLong/AtomicReference
其中AtomicReference是对象的引用。
AtomicIntegerArray/AtomicLongArray/AtomicReferenceArray
这三个是对上面的数组进行操作的。
AtomicLongFieldUpdater/AtomicIntegerFieldUpdater/AtomicReferenceFieldUpdater
用来更新类里面的某个字段。
AtomicMarkableReference/AtomicStampedReference
都是用来处理ABA问题的。
前一个增加了bool值用来判断变量是否被更改。
后一个将bool变成了整数值时间戳。