 **Java Concurrent**

# 队列相关

## BlockingQueue

阻塞队列

### ArrayBlockingQueue

数组式阻塞队列。

初始化的时候，使用ReentrantLock锁住，完成初始化，其实就是创建一个对象数组。

1. add(E e)

添加到队列中，实际上还是调用offer（调用无参的offer），如果添加失败的话，抛出队列已满异常。

1. offer(E e)

很简单的锁资源，添加到队列，然后释放锁。

1. put(E e)

这个就比较暴力了，如果队列满了，会一直等待（慎用）

1. offer(E e, long timeout, TimeUnit unit)

带个等待时间的。

1. poll()

获取队列元素，如果没有元素，直接返回null

1. take()

获取元素，没有的话，就一直等。

1. poll(long timeout, TimeUnit unit)

获取元素，没有就等待一段时间，没太大区别。

1. remove(Object o)

删除某个元素

1. contains(Object o)

是否存在某个元素

没有太多要说的，注意添加和获取的时候，无线等待要谨慎

### LinkedBlockingQueue

和上面的没什么区别，容器从数组改成链表

### LinkedBlockingDeque

和上面的区别不大，使用双向链表，可以双方向操作，FIFO，LIFO

### SynchronousQueue

没有太特殊的，他没有容器保存元素，采用注册消费者和生产者的方式。比如注册进来十个消费者，当有一个生产者生产完后，他会直接给消费者，本身不保存信息。当然，这样的话他也有fair和unfair模式，大概就是先到先得，或者随机取一个。

### LinkedTransferQueue

这个就厉害了，他可以在读取的同时写入，还能维持阻塞原则，增大了并发性。也就是说，他既可以保存元素，同时有支持SynchronousQueue的直接将元素push给消费者。

使用的时候无需关注这些。

特别的是，这个是google程序员写的，被招安了，弄进JDK里。还有个就是这个类中完成线程中资源共享，使用的是cas，不是其他的ReentrantLock的模式。他也是不阻塞线程的，但却继承了BlockingQueue，这个其实感觉不到。

### DelayQueue

没有太大意义，就是在获取的时候，判定如果需要等待，等几秒。

## BlockingDeque

阻塞队列，继承了Qeque，支持双向，主要是链表队列相关

### LinkedBlockingDeque

没什么特别的操作，支持反向而已

## ConcurrentLinkedQueue

1、2都是阻塞队列，3、4都是非阻塞队列。

使用cas的方式直接操作内存，避免了锁的干扰，性能要比锁快一些。效果是一样的。

自旋操作，比对内存中的tail和当前值预期，最后在末尾添加上去。

也有问题，比如ABA或者自旋时间过长，或者操作一大堆变量的时候，不过JDK都有解决。

### ConcurrentLinkedDeque

同上。

# ConcurrentHashMap

Hashmap的线程安全实现

经历了几个版本的更迭，ConcurrentHashMap也变换过各种实现方式

从之前的分段加锁，到现在全局加锁

17.及以前，会将ConcurrentHashMap中的数组分成几个段落，每个段落保存着一部分hash值，这样，当请求进来的时候，不需要对整个

数组加锁，只需要对相应的段落加锁就可以了。但效率依然没有预期的那么高。

18以后，ConcurrentHashMap改变了策略，不再使用段落的方式，转而使用unsafe高效率类来支撑线程安全问题

同时，Map中的数组也不再使用数组+链表的方式，使用数组+链表+红黑树的方式。

当链表中的数组大于8个（默认）的时候，将会转变方式，创建红黑树，

这个地方和HashMap稍有不同，因为线程安全情况下创建红黑树消耗的资源不菲，所以系统默认

如果当前数组的长度小于 64，那么会选择进行数组扩容，而不是转换为红黑树

后续扩容比较复杂，没怎么看，大概意思就是将数组扩容，对原数组采用分段的方式迁移（多线程），最大化利用CPU

同时，如果转换后的条件（链表小于8个节点），会再次转换回链表。

同时，红黑树中的元素仍然维持链表结构

# CopyOnWriteArrayList

没什么特别的，每次add的时候，重新构建一个新的array，注意，这个很吃资源，一般能不用就不用。

# LockSupport

LockSupport比synchronized+wait notify关键字好的地方在于，他维持了一个顺序，保证unpark在park之前执行，也就是有一定的顺序性。

# AtomicReferenceFieldUpdater

普通的Atomic也就更没啥可以看得了

都是使用unsafe（CAS compare and swap，大意是你告诉我内存地址，和原值，CPU去比较内存地址的值和你的原值是否一致，如果一致，set，否则返回内存地址的值）

这样可能会出现ABA的问题，比如你希望是1，之后别人改成0了，又有人改成1，这时系统是不知道有人改过的，可以使用AtomicStampedReference来解决这个问题

另外就是这种操作一般只针对一个对象比如Integer或者Boolean，不过后面出现了AtomicReference，可以把我们需要处理的对象都放到这个里，或者使用多个AtomicReferenceFieldUpdater

作为线程间共享变量的解决方式，使用park和unpark来处理线程访问变量。

# ReentrantLock

锁定线程，和synchronized关键字类似，比它灵活一些，还扩展出了读Lock和写Lock

# Semaphore

守门人。

Semaphore大概就是个看门人

比如，我有一个宾馆，有十间房子，来了十一个人，这时前十个可以进去，第十一个过去的时候，没有房间了，陷入等待，超时异常（也可以永久等待）

这个fair参数，true的话，大概就是先到先得，false的话，可能会随机给一个人，不过这个是非常不严谨的，如果要保证顺序的话，用true不一定能获得想要的效果。

# Executor

Executor这东西大概就相当于一个执行的容器，用来执行我们的Task（基本上都是线程）

## ThreadPoolExecutor

ThreadPoolExecutor中的execute函数是其中的核心函数

在execute的时候，会判定线程池个数，如果线程池个数是10个，会创建十个Worker

每一个Worker是一个资源池，如果已达到最大线程池数量，将不会创建Worker对象，所有的任务（程序所添加的线程）都将添加到队列中，如果是链表队列，则会无限添加

如果是有限队列（数组ArrayBlockQueue，或者有界链表），到达最大值得时候会抛出异常。

Worker中有个getTask函数，从Queue中获取消息，其中使用的是take函数，意味着，这里将无线等待下去。所以，我们的程序使用PoolExecutor的时候要谨慎，防止不停的创建该对象。

除非作为一个公用的线程池，系统启动后一直使用，不然的话，尽量在使用完线程池的时候，关闭当前线程池。

建议规划好代码，避免重复创建线程池，尽量使用公用对象，也就是所有请求使用同一个ThreadPoolExecutor

submit函数，本质上还是调用execute函数，在调用之前做一个RunnableFuture对象，该对象主要是实现run函数，还有get函数

run函数调用目标程序，get函数获取结果对象，这里在获取的时候先判定目标程序是否执行完成，如果没完成，主线程进入等待，在目标线程完成之后

回调告诉主线程，返回结果。阻塞主线程主要使用的是unsafe.park，新JDK的很多地方都是用的这种处理方式。

### 构造函数

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

1. corePoolSize

线程池大小

1. maximumPoolSize

线程池最大线程数

1. keepAliveTime

线程池中空闲线程存活最大时间

1. unit

线程池中线程存活时间的时间类型

1. workQueue

消息队列（阻塞线程式的消息队列，比如ArrayBlockingQueue、LinkedBlockingDeque、LinkedBlockingQueue，LinkedTransferQueue），后面会详细解析各种队列。这里使用的两种队列区别主要是，所有的链表队列是无界的（也可以设置界限），数组队列是有界的，当线程池满、队列满的时候，会抛出异常。

1. threadFactory

线程工厂，可以设置线程名，组等信息。

1. handler

错误处理，在某些情况下可用

### 属性

1. 状态参数属性

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

private static final int COUNT\_BITS = Integer.SIZE - 3;

private static final int CAPACITY = (1 << COUNT\_BITS) - 1;

// runState is stored in the high-order bits

private static final int RUNNING = -1 << COUNT\_BITS;

private static final int SHUTDOWN = 0 << COUNT\_BITS;

private static final int STOP = 1 << COUNT\_BITS;

private static final int TIDYING = 2 << COUNT\_BITS;

private static final int TERMINATED = 3 << COUNT\_BITS;

这里的所有状态参数都采用了高位操作，java在后面优化了高位操作的速度。

1. private final ReentrantLock mainLock = new ReentrantLock();

主要的锁，这里主要是在当前线程池终止、或者终止Worker的时候，用来屏蔽其他操作。

1. private final HashSet<Worker> workers = new HashSet<Worker>();

线程池中工作线程的容器，后面还有。

1. private final Condition termination = mainLock.newCondition();

终止的条件，也是和上面共同作用的

1. private long completedTaskCount;

已完成的任务总量。

### 函数

1. execute(Runnable command)

主要函数，当前工作线程不到最大工作线程的话，添加工作线程（addWorker），判定当前Executor是否是RUNNING状态，并且将目标线程添加到消息队列中。二次校验当前Executor是否RUNNING状态，不需要reject的话，如果工作线程是0的话，添加工作线程（addWorker）。

1. addWorker(Runnable firstTask, boolean core)

添加工作线程，其实没什么好讲的，判定Executor状态，将初始化工作线程Worker，将工作线程添加到workers属性中，添加的时候使用mainLock属性，对当前的Executor加锁。如果添加成功的话，启动工作线程，Worker也是实现了Runnable的。

1. Worker. runWorker(Worker w)

Worker中启动无限循环工作。使用getTask获取待执行线程。调用beforeExecute，启动待执行线程，最终调用afterExecute，在最后将当前工作线程已经执行工作计数器加一

1. beforeExecute(Thread t, Runnable r)

默认是没有需要做的事情的，当然我们可以自行实现。

1. afterExecute(Runnable r, Throwable t)

这个函数在启动待执行线程之后执行，注意，这里并不是待执行线程结束之后才调用，是启动之后就调用的

1. Worker. getTask

判定当前工作线程所在Executor的状态，如果是RUNNING状态，判定是否设定该值：allowCoreThreadTimeOut，如果设定了的话，工作线程有timeout时间。通俗点解释就是：

我们设定了allowCoreThreadTimeOut的值为true，当我们从消息队列取值的时候，将使用poll函数，并指定timeout时间为keepAliveTime设定，时间类型为NANOSECONDS。正常的情况下返回待执行线程，异常情况下返回的待执行线程为null，最终工作线程执行完毕。如果不设定allowCoreThreadTimeOut，在没有待执行线程返回的情况下，将永远等待下去。即便主线程工作线程timeout退出，仍然将重新添加工作线程—Worker

注意，是否设定allowCoreThreadTimeOut，Worker在执行线程之后，都将重新初始化。区别就是是否回收工作线程，设定该值的话，只要当前工作线程大于工作线程最小值的话，就不再重新添加工作线程了。

1. Worker.processWorkerExit(Worker w, boolean completedAbruptly)

计数器completedTaskCount++，并删除当前工作线程，详细的函数解析，参照上面的解释。