Java线程池和阻塞队列

为什么使用线程池

- 降低资源消耗
  - 通过重复利用已经创建的线程降低创建和销毁的开销
- 高响应速度
  - 任务到达时，没有线程创建的等待时间
- 高线程的可管理性
  - 统一分配、调优和监控
  - 避免创建过多线程导致进程崩溃
- 特别适合场景：短执行时间、大任务量

线程池的创建

- **new** ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds, runnableTaskQueue, handler);

- corePoolSize：提交任务到线程池时，线程池创建新的线程，只要需要执行的任务数小于corePoolSize，即使线程池中有空闲的线程可以执行新任务，线程池也会创建新线程(代码中叫addWorker，addWorker执行时会再次检查需要执行的任务数，防止多线程并发情况下过多创建线程）创建新线程需要获取全局锁
- maximumPoolSize: 线程池允许创建的最大线程数，如果队列满了，并且已创建的线程数小于maximumPoolSize，则线程会再创建新的线程执行任务（如果使用了无界队列，则该参数无效，因为队列一直不会满），创建线程需要获取全局锁
- keepAliveTime：工作线程空闲后，保持存活的时间，如果任务很多，并且每个任务执行的时间比较短，可以将保活时间调大，提高线程的利用率

线程池任务队列

- ArrayBlockingQueue：有界阻塞队列，FIFO
- LinkedBlockingQueue：有界阻塞队列，FIFO
- SynchronousQueue：0元素阻塞队列，每个入队操作需等另一个线程调用移除操作，否则一直阻塞，Executors.newCachedThreadPool 使用了该队列
- PriorityBlockingQueue：无界阻塞队列，优先级队列

线程池工作线程

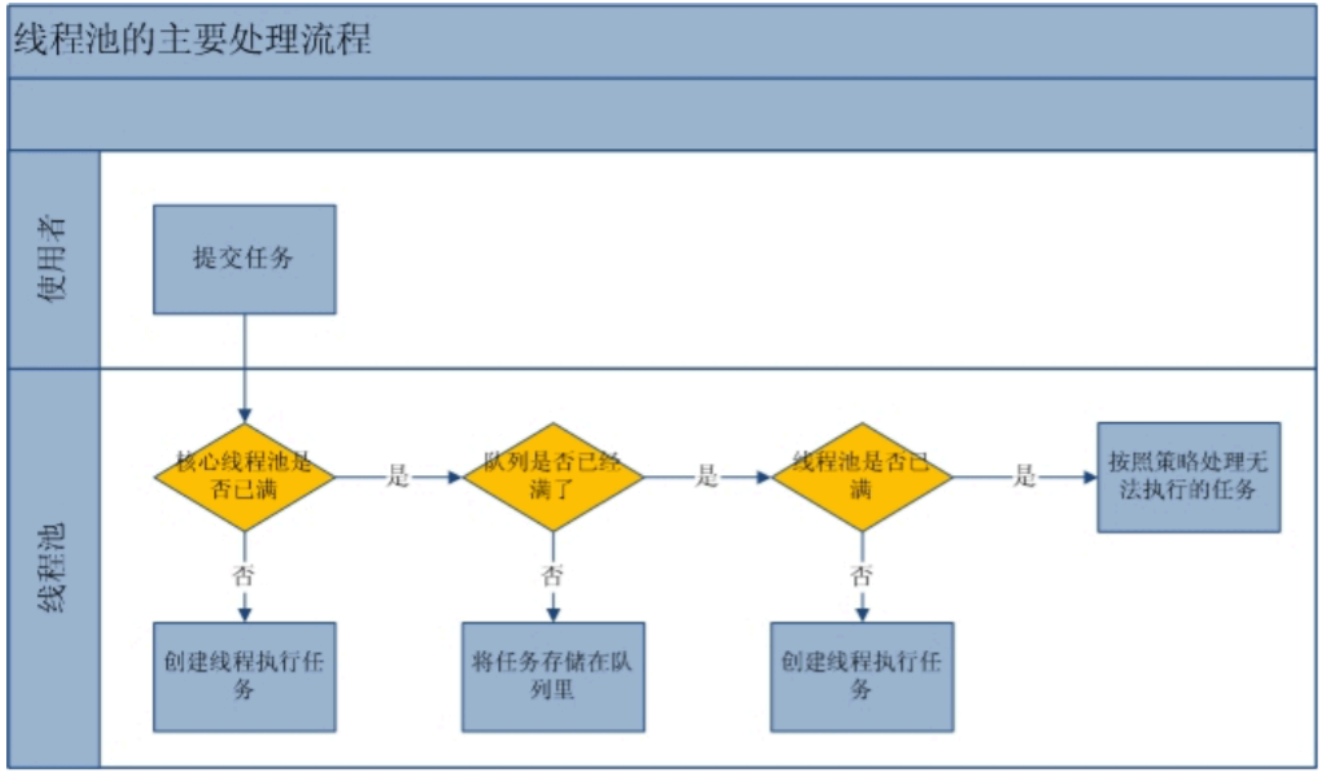
- 线程池创建线程时，会将线程封装成工作线程Worker，Worker会先执行提交给execute()的当前任务，Worker在执行完任务后，还会无限循环获取工作队列中的任务来执行。

```
public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    } finally {
        workerDone(this);
    }
}
```

线程池饱和策略

- RejectedExecutionHandler：当队列和线程池都满了的拒绝策略
  - 不处理的策略：
    - AbortPolicy：直接抛出异常，默认策略
    - DiscardPolicy：不处理，丢弃掉
  - 处理的策略：
    - CallerRunsPolicy：只用调用者线程来运行任务
    - DiscardOldestPolicy：丢弃队列里最老的一个任务，并执行当前任务
  - 其他策略：
    - 自定义策略实现接口RejectedExecutionHandler，如记录日志，持久化不能处理的任务

线程池工作流程


线程池的主要处理流程
使用者
线程池
提交任务
核心线程池是否已满
是
队列是否已经满了
是
线程池是否已满
是
按照策略处理无法执行的任务
否
否
否
创建线程执行任务
将任务存储在队列里
创建线程执行任务

线程池工作流程为什么要这么设计？

- ThreadPoolExecutor设计总体思路是为了在执行execute()时，尽可能地避免获取全局锁。
- 在ThreadPoolExecutor完成预热之后（当前运行的线程数大于等于corePoolSize），几乎所有的execute()方法调用都是往阻塞队列放任务，而这个操作不需要获取全局锁。
- 线程池时特殊的优化的生产者消费者模式在并发中的应用

向线程池提交任务

- execute: 提交Runnable，无返回值
- submit：提交Runnable，Callable，返回Future<Object>，通过future的get方法来获取返回值
  - get 方法会阻塞住直到任务完成
  - get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回,这时有可能任务没有执行完

Worker

- TheadPoolExecutor.Worker是ThreadPoolExecutor的内部类，实现了Runnable接口
- Worker.runTask（）会被execute()，或submit（）触发，所以是多线程访问的
  - beforeExecute(thread, task)
  - task.run()
  - afterExecute(task, null)

线程池的关闭

- shutdown
- shutdownNow
- isShutdown：只要调用了shutdown或shutdownNow，isShutdown就是true
- isTerminated：所有任务都关闭后，线程池关闭成功，isTerminated为true
- shutdown和shutdownNow区别
  - shutdown将线程池的状态设置成shutdown状态，然后中断所有没有正在执行任务的线程
  - shutdownNow酱线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表。

线程池的合理配置

- 任务的性质：CPU密集型、IO密集型、混合型
  - CPU密集型任务配置尽可能小的线程池，如Ncpu + 1个线程的线程池
  - IO密集型任务配置尽可能多的线程池，如2*Ncpu
- 任务的优先级：高、中、低
  - 使用PriorityBlockingQueue，让优先级高的任务先得到执行
- 任务的执行时间：长、中、短
  - 执行时间不同的任务可以交给不同规模的线程池来处理,或者也可以使用优先级队列,让执行 时间短的任务先执行
  - 执行时间长的任务交给小线程池处理，执行时间短的任务交给大线程池处理？
- 任务的依赖性：是否依赖其他系统资源，如数据库连接
  - 依赖数据库连接池的任务,因为线程 交 SQL 后需要等待数据库返回结果,如果等待的时间 越长 CPU 空闲时间就越长,那么线程数应该设置越大,这样才能更好的利用 CPU

线程池的合理配置

- 何时使用有界队列？
- 何时使用无界队列？

线程池的监控

- taskCount:线程池需要执行的任务数量
- completedTaskCount:线程池在运行过程中已完成的任务数量。小于或等于taskCount
- largestPoolSize:线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小,则表示线程池曾经满了
- getPoolSize:线程池的线程数量。如果线程池不销毁的话,池里的线程不会自动销毁,所以这个大小只增不减
- getActiveCount:获取活动的线程数。

线程池的监控

- 通过继承线程池并重写线程池的 beforeExecute, afterExecute 和 terminated 方法,我们可以在任务执行前,执行后和线程池关闭前干一些事情。如监控任务的平均 执行时间,最大执行时间和最小执行时间等。

Executor框架

- <JDK5
  - Thread即是工作单元，又是执行机制
- >=JDK5
  - 工作单元：Runnable, Callable
  - 执行机制：Executor框架

Executor框架的两级调度模型

- 用户级调度
  - 通过Exector框架将用户任务调度（映射）到Java线程
- 内核级调度
  - 通过操作系统内核将Java线程映射到本地操作系统 CPU线程

Executor框架结构和成员

- 任务：Runnable，Callable，FutureTask(FutureTask同时也实现了Runnable接口)
- 异步任务结果：Future接口和实现Future接口的FutureTask
  - get()
  - cancel(boolean mayInterruptIfRunning)
- 任务的执行：
  - Executor接口
  - ExecutorService接口，继承自Executor接口
    - ThreadPoolExecutor：ExecutorService的实现类
    - ScheduledThreadPoolExecutor：ExecutorService的实现类（ScheduledExecutorService接口的实现类）
  - execute方法和submit方法

Executors工厂

- 创建3种类型的ThreadPoolExecutor
  - FixedThreadPool：适用于负载比较重的服务器
  - CachedThreadPool：无界线程池，适用于执行很多的短期异步任务的小程序，或是负载较轻的服务器
  - SingleThreadExecutor：适用于需要保证顺序地执行各个任务，且任意时间点不会有多个线程活动的应用场景
- 创建2种类型的ScheduledThreadPoolExecutor
  - ScheduledThreadPool：适用于多个后台线程执行周期任务，同时限定后台线程数量
  - SingleThreadScheduledExecutor：适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的场景

FixedThreadPool

- corePoolSize和maximumPoolSize大小设置为一样
- 基于LinkedBlockingQueue的无界阻塞队列（容量Integer.MAX_VALUE）
  - 注意：不能说LinkedBlockingQueue是无界的，LinkedBlockingQueue默认构造方法出来的是无界（Integer.MAX_VALUE）的，但是也有带int capacity的构造方法创建出来的是有界的队列。这里说无界，是指针对FixedThreadPool来说，由于在创建ThreadPoolExecutor是用的默认构造方法，所以FixedThreadPool是无界的。
- 特点：
  - 当线程数达到corePoolSize后，新任务加入队列，由于是无界队列，线程数不会超过corePoolSize
  - 由于是无界队列，maximumPoolSize是一个无效值
  - 由于线程不会超过corePoolSize，keepAliveTime是一个无效值（keepAliveTime是当线程数超过corePoolSize时，多余的空闲线程等待新任务的最长时间，超过这个时间后，多余的线程被终止）

FixedThreadPool执行流程

- （1）如果当前运行的线程数少于corePoolSize，则创建新线程来执行任务
- （2）当运行的线程数达到corePoolSize后，任务将加入LinkedBlockingQueue
- （3）线程执行完（1）中的任务后，会在循环中反复从LinkedBlockingQueue获取任务来执行

CachedThreadPool

- corePoolSize为0，maximumPoolSize为Integer.MAX_VALUE
- keepAliveTime为60L，意味着超过corePoolSize线程时（有一个线程就符合要求，即线程池里的所有线程），空闲线程的等待新任务时间时60秒，所以长时间保持空闲的CachedThreadPool中不存在线程，不会使用任何资源。
- 使用的阻塞队列是SynchronousQueue：没有容量的阻塞队列，而maximumPoolSize为无界（Integer.MAX_VALUE），如果主线程提交任务的速度高于线程池中处理任务的速度时，CachedThreadPool会不断创建新线程，极端情况下会导致耗尽CPU和内存资源。

CachedThreadPool执行流程

- （1）来一个任务时，由于corePoolSize为0，第一个任务（以后每个任务进来也都是先尝试加入）就加入到阻塞队列（阻塞队列容量为0），调用SynchronousQueue.offer()方法，如果offer操作正好有其他线程的poll操作配对，主线程把任务交给空闲线程执行，execute()方法执行完成，
- （2）如果（1）中没有配对的操作，CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成
- （2）执行完成任务的线程会继续从阻塞队列取任务poll，poll最长等待keepAliveTime（60秒），如果60秒内有新任务offer进来，则这个线程继续执行任务，否则空闲线程被终止释放。

SingleThreadExecutor

- corePoolSize和maximumPoolSize都为1
- 使用LinkedBlockingQueue和FixedThreadPool的一样
  - 注意：不能说LinkedBlockingQueue是无界的， LinkedBlockingQueue默认构造方法出来的是无界（Integer.MAX_VALUE）的，但是也有带int capacity的构造方法创建出来的是有界的队列。这里说无界，是指针对SingleThreadExecutor来说，由于在创建ThreadPoolExecutor是用的默认构造方法，所以SingleThreadExecutor是无界的。
- 第一次执行任务时创建一个线程直接执行任务
- 后续这个线程无限循环从LinkedBlockingQueue获取任务来执行

ScheduledThreadPoolExecutor

- 使用DelayQueue（基于PriorityBlockingQueue）无界阻塞队列
- 由于是无界阻塞队列，maximumPoolSize无效
- 任务：实现了RunnableScheduledFuture接口的ScheduledFutureTask（而非Runnable或Callable）
- 提交任务的方式：将任务放倒DelayQueue
  - scheduleAtFixedRate
  - scheduleAtFixedDelay
- ScheduledFutureTask
  - time: long型，表示任务具体执行的时间
  - sequenceNumber, long型， 提交到任务的序号，用于time相同时的排序
  - period：long型，任务执行的间隔周期

ScheduledThreadPool执行周期任务流程

- 线程从DelayQueue中获取（take方法）已到期的ScheduledFutureTask
- 执行ScheduledFutureTask
- 修改这个ScheduledFutureTask的time为下次执行的时间
- 将ScheduledFutureTask放回道DelayQueue中（PriorityBlockingQueue会自动重排序）

FutureTask内部实现

- 基于AQS（AbstractQueuedSynchronizer）
- FutureTask.get()→AQS.acquireSharedInterruptibly(int arg)→FutureTask自定义内部类(继承自AQS）的Sync.tryAcquireShared()判断是否能获取锁
- FutureTask.run()/cancel(…)→AQS.compareAndSetState(int expect, int update)→AQS.releaseShared(int arg)→唤醒线程等待队列里的第一个线程
- 每个被唤醒的线程将自己从队列中删除，然后唤醒后继线程节点，以此类推，最终所有等待的线程都被级联唤醒并从get()方法返回

生产者和消费者模式

- 并发编程中使用生产者和消费者能够解决绝大多数并发问题
- 生产者和消费者的解耦需要使用一个容器：阻塞队列
- Java中的线程池其实就是一种生产者和消费者模式
- Java中线程池是高明的生产者消费者模式：
  - 给线程池生产任务时，如果线程数没有达到corePoolSize时，直接创建消费者消费了，只有达到corePoolSize时才放到阻塞队列，这种方式比普通的生产者消费者只使用阻塞队列来交换更快一些。

阻塞队列

- 阻塞队列常用于生产者和消费者的场景,生产者是往队列里添加元素的线程,消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素,消费者用来获取元素的容器。
- 阻塞队列(BlockingQueue)是一个支持两个附加操作的队列。
  - 支持阻塞的插入方法:意思是当队列满时,队列会阻塞插入元素的线程,直到队列不满。
  - 支持阻塞的移除方法:意思是在队列为空时,获取元素的线程会等待队列变为非空。

阻塞队列

- 在阻塞队列不可用时,这两个附加操作（支持阻塞的插入和移除操作）供了四种处理方式
  - 抛出异常:当队列满时,如果再往队列里插入元素,会抛出 IllegalStateException("Queue full") 异常。当队列空时,从队列里获取元素会抛出 NoSuchElementException 异常
  - 返回特殊值:当往队列插入元素时,会返回元素是否插入成功,成功返回 true。如果是移除方 法,则是从队列里拿出一个元素,如果没有则返回 null。
  - 一直阻塞:当阻塞队列满时,如果生产者继续线程往队列里 put 元素,队列会一直阻塞生产者 线程,直到队列可用或者响应中断退出。当队列空时,如果消费者线程从队列里 take 元素,队 列会阻塞住消费者线程,直到队列不为空。
  - 超时退出:当阻塞队列满时,如果生产者线程往队列里插入元素,队列会阻塞生产者线程一段 时间,如果超过了指定的时间,生产者线程就会退出。

阻塞队列

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

- 如果是无界阻塞队列,因为队列不可能会出现满的情况,所以使用 put 或 offer 方法永远不 会被阻塞,而且使用 offer 方法时,该方法永远返回 true。

Java里的阻塞队列

- 有界队列
  - ArrayBlockingQueue
  - LinkedBlockingQueue（容量Integer.MAX_VALUE，可以认为是无界队列）
  - SynchronousQueue
- 无界队列
  - PriorityBlockingQueue
  - DelayQueue
  - LinkedTransferQueue
- 有界 / 无界队列
  - LinkedBlockingDeque

ArrayBlockingQueue

- 有界阻塞队列，FIFO
- 默认情况下不保证线程公平访问队列，不按阻塞的先后顺序访问队列
- 是否让阻塞线程公平访问阻塞队列，通过ArrayBlockingQueue的ReentrantLock实现 ArrayBlockingQueue fairQueue = **new** ArrayBlockingQueue(1000,**true**); // ArrayBlockingQueue(**int** capacity, **boolean** fair)

- 实现原理：生产者消费者模式， ReentrantLock + Condition

LinkedBlockingQueue

- 有界队列，FIFO
- 默认和最大长度Integer.MAX_VALUE

SynchronousQueue

- 不存储元素的阻塞队列
- 每个put操作必须等待一个take操作
- 默认情况，线程非公平访问队列，先阻塞队列可能后访问
- 队列本身不存储任何元素，适合于传递性场景
- SynchronousQueue吞吐量高于LinkedBLockingQueue和ArrayBlockingQueue

```
public SynchronousQueue(boolean fair) {
        transferer = fair ? new TransferQueue() : new TransferStack();
    }
```

PriorityBlockingQueue

- 优先级，无界队列

DelayQueue

- 无界阻塞队列
- 内部用PriorityQueue实现
- 队列中的元素必须实现Delayed接口
- 在创建元素时可以指定多久才能从队列中获取当前元素
- 应用场景
  - 缓存系统的设计：可以用 DelayQueue 保存缓存元素的有效期,使用一个线程循环查询 DelayQueue,一旦能从 DelayQueue 中获取元素时,表示缓存有效期到了。
  - 定时任务调度：使用 DelayQueue 保存当天将会执行的任务和执行时间,一旦从 DelayQueue 中获取到任务就开始执行,比如 TimerQueue 就是使用 DelayQueue 实现的。

LinkedTransferQueue

- 无界阻塞队列
- transfer
  - 如果当前有消费者正在等待接收元素(消费者使用 take()方法或带时间限制的 poll()方法时),transfer 方法可以把生产者传入的元素立刻 transfer(传输)给消费者。
  - 如果没有消 费者在等待接收元素,transfer 方法会将元素存放在队列的 tail 节点,并等到该元素被消费者消费 了才返回。
- tyTransfer
  - 用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费 者等待接收元素,则返回 false。
  - 和 transfer 方法的区别是 tryTransfer 方法无论消费者是否接收,方 法立即返回。而 transfer 方法是必须等到消费者消费了才返回。

LinkedTransferQueue

- tryTransfer(E e, long timeout, TimeUnit unit)
  - 试图把生产者传入 的元素直接传给消费者,但是如果没有消费者消费该元素则等待指定的时间再返回,如果超时还没 消费元素,则返回 false,如果在超时时间内消费了元素,则返回 true。

LinkedBlockingDeque

- 双向阻塞队列，有界 / 无界
- 在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀
- 应用场景：
  - 工作窃取模式

阻塞队列实现原理

- 生产者消费者
- ReentrantLock + Condition
- Condition.await()→LockSupport.park(obj)→native: unsafe.park
- unsafe.park阻塞当前线程，以下四种情况之一发生，方法返回
  - 与park对应的unpark之行或已经执行时
  - 线程被中断时
  - 等待完time参数指定的毫秒数时
  - 异常现象时，这个异常现象没有任何原因
- 当线程被阻塞队列阻塞时，线程会进入 WAITING (parking)状态
  ←jstack dump

Fork/Join框架

- Java7的并行执行任务框架
- 工作窃取(work-stealing)：某个线程从其他队列里窃取任务来执行
  - 拆分大任务，子任务放到不同队列里，每个队列一个单独线程执行→减少线程间的竞争
  - 已完成任务的线程，从其他还非空的队列里窃取任务执行
  - 队列使用双端队列，WHY？→减少窃取任务线程和被窃取任务线程之间的竞争

Fork / Join框架

- ForkJoinTask
  - fork()
  - join()
  - getException()，由于无法在主线程直接捕获异常，通过该方法获取异常
- 通常我们通过继承ForkJoinTask的以下子类来创建ForkJoin任务
  - RecursiveAction:用于没有返回结果的任务
  - RecursiveTask :用于有返回结果的任务
- ForkJoinPool：ForkJoinTask需要ForkJoinPool来执行
- 如何执行：
  - 任务分割出的子任务会添加到当前工作线程所维护的双端队列中,进入队列的头部。当一个工作线程的队列里暂时没有任务时,它会随机从其他工作线程的队列的尾部获取一个任务。

Fork/Join框架实现原理

- ForkJoinPool
  - ForkJoinTask数组
  - ForkJoinWorkerThread数组
- ForkJoinTask.fork()→ForkJoinWorkerThread.pushTask()→把任务放到ForkJoinTask[] queue里→ForkJoinPool.signalWork()唤醒或创建一个工作线程来执行
- ForkJoinTask.join()：阻塞当前线程并等待获取结果