



# Java 并发编程之美



## 第6章 Java并发包中锁原理剖析

1. LockSupport 工具类	挂起/唤醒线程 * park() * unpark()
2. AQS 抽象同步队列	* 同步器的基础/并发锁对象的基础 * 根据状态来管理入队列的线程 * inner class: ConditionObject → 对应条件队列 ↓ 用于入队挂起的线程 * acquire()/release() 独占 * acquireShared()/releaseShared() 共享
3. ReentrantLock 可重入锁	* 独占锁 非公平 → 默认 公平 → 队列 可重入 → 重入线程 acquire++ * lock()/unlock()/tryLock()/
4. ReentrantReadWriteLock 可重入读写锁	* 适用于写少读多场景 * 写 → 获取独占锁 ! 读写时勿忘 release both * 读 → 获取共享锁
5. StampedLock 不可重入	<div><div><div>writeLock 独占锁</div><div>readLock 独占锁</div><div>tryOptimisticRead 读获取共享锁</div></div><div>读少写多</div><div>读多写少</div></div>

\* JUC 锁基于 AQS, 实现 3 AQS 中的 methods

\* StampedLock 不可重入, 通过比较 stamp 的手法来维护 Happens Before

# 第7章 并发队列

## 1. Concurrent Linked Queue

\* 线程安全 / 无界 / 非阻塞

\* offer(): CAS 非阻塞

\* 用 CPU 换取阻塞的开销

\* size 方法 CAS 下不加锁 不准确

\* 独占锁

\* 有界

\* 链表实现

\* ReentrantLock x 2 { takeLock 出队  
putLock 入队

\* Condition x 2 { notEmpty  
notFull

包含条件队列: 存放被阻塞的线程

\* 调用 condition.await() / signal() 前先获取锁

* 非阻塞	offer()	poll()	peek()
阻塞	put()	take()	

\* size() 方法加锁 是准确的

\* remove() 方法获取双锁 (takeLock & putLock)

## 3. Array Blocking Queue

\* 独占锁

\* 单个独占锁

\* 有界

\* NotEmpty / NotFull Condition

\* 数组实现

* 非阻塞	offer()	poll()	peek()
阻塞	put()	take()	

\* size() 方法加锁 是准确的

Array Blocking Queue 单锁 粒度粗

Linked Blocking Queue 双锁 ! 出入队列可以同时

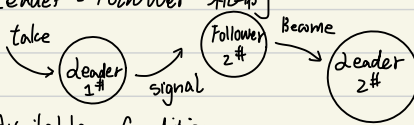
## 第7章 并发队列

### 4. PriorityBlockingQueue

- \* 优先级
- \* 无界
- \* Balanced Binary Tree
- \* 阻塞
- \* allocationSpinLock 自旋锁 CAS 机制
- \* NotEmpty Condition
- \* 扩容 queue
  - < 64  $\Rightarrow +2$
  - > 64  $\Rightarrow \times 1.5\%$

! 扩容时释放锁 少并发性  
CAS 扩容 其它线程可出入队

### 5. DelayQueue

- \* 无界 阻塞 延迟队列 头部为快过期的元素
- \* Leader - Follower 机制

```
graph LR; L1((Leader 1#)) -- take --> F2((Follower 2#)); F2 -- signal --> L1; F2 -- become --> L2((Leader 2#));
```
- \* Available Condition

## 第8章 ThreadPoolExecutor 原理

### 1. 状态

$\left\{ \begin{array}{ll} \text{RUNNING} & \text{运行态} \\ \text{SHUTDOWN} & \text{拒绝新任务, 处理阻塞队列任务} \\ \text{STOP} & \text{拒绝新任务, 抛弃队列任务} \\ \text{TIDYING} & \text{执行完毕, 活跃线程数} = 0 \\ \text{TERMINATED} & \text{TIDYING 之后状态} \end{array} \right.$

### 2. 状态转换

\*  $\text{RUNNING} \xrightarrow{\text{shutdown()}} \text{SHUTDOWN}$   
\*  $\text{RUNNING} / \text{SHUTDOWN} \xrightarrow{\text{shutdownNow()}} \text{STOP}$   
\*  $\text{SHUTDOWN} \xrightarrow[\text{任务执行完了}]{\text{线程池空, 队列空}} \text{TIDYING}$   
\*  $\text{STOP} \xrightarrow{\text{线程池空}} \text{TIDYING}$   
\*  $\text{TIDYING} \xrightarrow{\text{terminate()}} \text{TERMINATED}$

### 3. 核心参数

\* corePoolSize  
\* workQueue  
\* maximumPoolSize  
\* ThreadFactory  
\* RejectExecutionHandler  $\left\{ \begin{array}{l} \text{AbortPolicy} \text{ 抛异常} \\ \text{CallerRunsPolicy} \text{ 调用者线程运行} \\ \text{DiscardOldestPolicy} \text{ 丢老的} \\ \text{DiscardPolicy} \text{ 丢新的} \end{array} \right.$   
\* keepAliveTime  
    ↑  
    PoolSize > coreSize 存活时间

\* ThreadPoolExecutor 的本质还是一个生产者-消费者队列

## 第8章 Thread Pool Executor 原理

### 4. 线程池类型

- \* new FixedThreadPool core = max = nThreads
- \* new SingleThreadExecutor core = max = 1
- \* new CachedThreadPool 按需创建 MAX\_VALUE

### 5. 源码分析

1° execute()

提交任务 → 判断状态  $\xrightarrow{\text{addWorker}}$  增加工作线程  
\* 同时只能有一个线程  
↓ CAS增加线程数  
成功调用 execute() 方法

2° runWorker()

初始化设置 Worker 状态 - 1 → 不会被中断

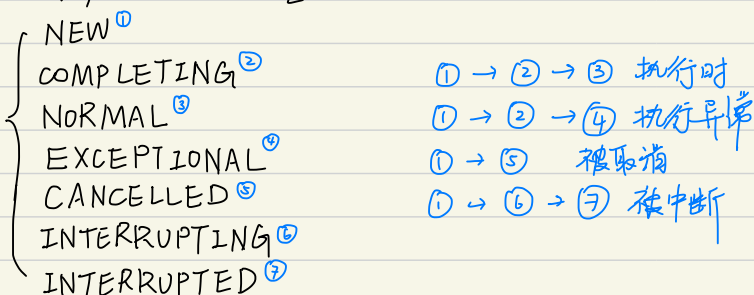
\* Worker 复用: 每个 Worker 可以处理多个任务

# 第9章 ScheduledThreadPoolExecutor

## 1. Overview

### 1.1 状态

extends ThreadPoolExecutor  
impl ScheduledExecutorService



## 2. API

schedule()  
scheduleWithFixedDelay()  
scheduleAtFixedRate()

## 第10章 线程同步器

### 1. CountDownLatch

\* 适用场景：主线程等待所有子线程执行完毕后汇总

\* 样例：`thread1.start()`

`thread2.start()`

`:`

`countDownLatch.await()`

#### 1.1 原理

\* AQS  $\Rightarrow$  委托 sync 调用 AQS 方法实现

\* 减计数器

### 2. CyclicBarrier

\* 功能：一组线程全部到同一状态再同时执行

\* 可重用

\* 调用 `await()` 方法到达屏障点

\* 再从屏障点同时出发

#### 2.1 原理

\* 独占锁 (AQS)

\* `Var count`

`Var parties` 总线程数

`count == 0  $\Rightarrow$  count = parties`

### 3. Semaphore

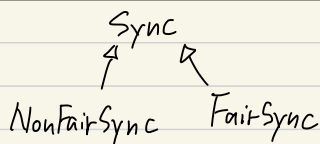
\* 使用 初始化值 `M`  $\Rightarrow$  需要被释放 `M+N` 次  
调用 `acquire(N)`

\* 可以模拟 CyclicBarrier 复用功能



## 第10章 线程同步器

### 3.1 原理



### 3.2 API

\* acquire() 获取信号量 {  $> 0$  计数 "-1"  
\* acquire(permits) {  $= 0$  AQS阻塞入队  
\* release() 计数 "+1"  
\* release(permits)

\* 由于基于 AQS  $\Rightarrow$  release() 后不需要手动通知队列中的线程执行任务

## 第11章 并发编程实践

- 1° ArrayBlockingQueue 使用  
logback 打印日志  $\Rightarrow$  多生产者-单消费者模型
- 2° Tomcat NioEndpoint's ConcurrentLinkedQueue