第十四章 构建自定义的同步工具

1. 状态依赖类

在这些类的一些操作中有着基于状态的前提条件，例如不能从一个空的队列中删除元素。

2. 创建状态依赖类的最简单方法通常是在现有类库中的状态依赖类的基础上进行改造。

#### 一．状态依赖性的管理

1. 对于单线程程序，某个条件为假，那么这个条件将永远无法成真；

在并发程序中，基于状态的条件可能会由于其他线程的操作而改变；

2. 条件队列

使得一组线程（称之为等待线程集合）能够通过某种方式来等待特定的条件变成真（元素是一个个正在等待相关条件的线程）

* 每个对象都可以作为一个条件队列（API：wait、notify和notifyAll)
  + Object.wait会自动释放锁，并请求操作系统挂起当前线程，从而使其他线程能够获得这个锁并且修改对象的状态
  + Object.notify/notifyAll通知被挂起的线程可以重新请求资源执行
* 只有能对状态进行检查时，才能在某个条件上等待，并且只有能修改状态时，才能从条件等待中释放另一个线程
* 条件队列在CPU效率、上下文切换开销和响应性等进行了优化
* 如果某个功能无法通过“轮询和休眠”来实现，那么使用条件队列也无法实现

#### 二．使用条件队列

1. 条件谓词

条件谓词就是一个个条件，在java程序中就是变量构成的表达式。

每一次wait调用都会隐式地与特定的条件谓词关联起来。当调用某个特定条件谓词的wait时，调用者必须已经持有与条件队列相关的锁，并且这个锁必须保护着构成条件谓词的状态变量。

2. 过早唤醒

当使用条件等待时（例如Object.wait或Condition.await）

* 通常都有一个条件谓词--包括一些对象状态的测试，线程在执行前必须首先通过这些测试
* 在调用wait之前测试条件谓词，并且从wait中返回时再次进行测试
* 在一个循环中调用wait
* 确保使用与条件队列相关的锁来保护构成条件谓词的各个状态变量
* 当调用wait, notify或notifyAll等方法时，一定要持有与条件队列相关的锁
* 在检查条件谓词之后以及开始执行相应的操作之前，不要释放锁。

void stateDependentMethod() throws InterruptedException

{

synchronized(lock) // 必须通过一个锁来保护条件谓词

{

while(!condietionPredicate())

lock.wait();

}

}

状态依赖方法的标准形式。

5. 封装条件队列

6. 入口协议和出口协议

* 入口协议：该操作的条件谓词
* 出口协议：检查被该操作修改的所有状态变量，并确认它们是否使某个其他的条件谓词变为真，如果是，则通知相关的条件队列

#### 三．显示的Condition对象

1. 内置条件队列的缺点：

每个内置锁都只能有一个相关联的条件队列，而多个线程可能在同一条件队列上等待不同的条件谓词，调用notifyAll通知的线程非等待同意谓词

Condition <-> Lock，内置条件队列 <-> 内置锁

* Lock.newCondition()
* 在每个锁上可存在多个等待、条件等待可以是可中断的或不可中断的、基于时限的等待，以及公平的或非公平的队列操作
* Condition对象继承了相关的Lock对象的公平性
* 与wait、notify和notifyAll方法对应的分别是await、signal和signalAll
* 将多个条件谓词分开并放到多个等待线程集，Condition使其更容易满足单次通知的需求（signal比signalAll更高效）
* 锁、条件谓词和条件变量：件谓词中包含的变量必须由Lock来保护，并且在检查条件谓词以及调用await和signal时，必须持有Lock对象

#### 四．Synchronizer剖析

1. 在ReentrantLock和Semaphore这两个接口之间存在许多共同点。

（1） 两个类都可以用作一个”阀门“，即每次只允许一定数量的线程通过，并当线程到达阀门时，可以通过（在调用lock或acquire时成功返回），也可以等待（在调用lock或acquire时阻塞），还可以取消（在调用tryLock或tryAcquire时返回”假“，表示在指定的时间内锁是不可用的或者无法获取许可）。

（2） 这两个接口都支持中断、不可中断的以及限时的获取操作。

（3） 都支持等待线程执行公平或非公平的队列操作。

原因：都实现了同一个基类AbstractQueuedSynchronizer（AQS）

#### 五．AbstractQueuedSynchronizer

1. 最基本的操作：

* 获取操作是一种依赖状态的操作，并且通常会阻塞（同步器判断当前状态是否允许获得操作，更新同步器的状态）
* 释放并不是一个可阻塞的操作时，当执行“释放”操作时，所有在请求时被阻塞的线程都会开始执行

2. 状态管理（一个整数状态）：

* 通过getState，setState以及compareAndSetState等protected类型方法来进行操作
* 这个整数在不同子类表示任意状态。例：剩余的许可数量，任务状态
* 子类可以添加额外状态

#### 六．java.util.concurrent同步容器类中的AQS

1. ReentrantLock

（1） 维护一个变量记录当前线程，如果某个线程获取成功，则把该变量置为该线程标识。

（2） 维护一个状态，表示被获取的次数，如果是其它线程获取成功，则该状态变为1，如果一个线程获取多次，则累加。

2. Semaphore与CountDownLatch

Semaphore将AQS的同步状态用于保存当前可用许可的数量；CountDownLatch使用AQS的方式与Semaphore很相似，在同步状态中保存的是当前的计数值。

3. FutureTask

在FutureTask中，AQS同步状态被用来保存任务的状态

　　FutureTask还维护一些额外的状态变量，用来保存计算结果或者抛出的异常

4. ReentrantReadWriteLock

* 单个AQS子类将同时管理读取加锁和写入加锁
* ReentrantReadWriteLock使用了一个16位的状态来表示写入锁的计数，并且使用了另一个16位的状态来表示读取锁的计数
* 在读取锁上的操作将使用共享的获取方法与释放方法，在写入锁上的操作将使用独占的获取方法与释放方法
* AQS在内部维护了一个等待线程队列，其中记录了某个线程请求的是独占访问还是共享访问：写操作独占获取；读操作可使第一个写之前的读都获取