# Java7并发编程实战手册

# 线程管理

## 线程的创建和运行

Java提供了两种方式来创建线程：

* 继承Thread类，并且覆盖run()方法
* 创建一个实现Runnable接口的类。使用带参数的Thread构造器来创建Thread对象。这个参数就是实现Runnable接口的类的一个对象。

## 线程信息的获取和设置

ID：保存了线程的唯一标识符

Name:保存了线程名称

Priority:保存了线程对象的优先级。线程的优先级从1到10,1是最低优先级，10是最高优先级。

Status:保存了线程的状态。在Java中，线程的状态有6种：new、runnable、blocked、waiting、time waiting、terminated。

## 线程中断

interrupt()方法：中断线程

isInterrupt():boolean判断线程是否被中断

interrupted():static boolean :Thread类的静态方法。可以设置interrupted属性为false。推荐使用 interrupted（）方法

## 线程中断的控制

Java提供了InterruptedException异常。当检查到线程中断的时候，就抛出这个异常，然后在run()中捕获并处理这个异常。代码路径：

D:\developer\code\jee\foundation\src\main\java\tudou\thread\java7concurrency\chapter1\interrupt\_exception

## 线程的休眠与恢复

1. sleep()方法,（ms）
2. TimeUnit.SECONDS.sleep()方法（s）

## join()：等待线程的终止

在thread1中有这样的代码：thread2.join()：等待thread2运行完成后，接着运行thread1

## 守护线程 daemon

优先级低，当一个应用程序中没有其他线程运行的时候，守护线程才运行。当守护线程是程序中唯一运行的线程时，守护线程执行结束后，JVM也就结束了这个程序。典型的守护线程时Java的垃圾收集器。

setDaemon()方法只能在start()方法被调用之前设置。一旦线程开始，将不能更改守护状态。

## 线程局部变量 ThreadLocal

线程的局部变量，对于当前线程来说可见。

## 工厂类创建线程

# 线程同步基础

临界区（Critical Section）:一个用于访问共享资源的代码块。这个代码块在同一时间内只允许一个线程访问。

当一个线程试图访问一个临界区时，它将使用一种同步机制来查看是否已经有其他线程进入临界区。如果没有其他线程进入临界区，它就可以进入临界区；如果有其他线程进入了临界区，它就被同步机制挂起，直到进入的线程离开这个临界区。如果临界区的线程不止一个，JVM会随机选取其中的一个，其余的将继续等待。

Java语言提供了两种同步机制：

* synchronized关键字机制
* Lock接口及其实现机制

## synchronized

每一个用synchronized关键字声明的方法都是临界区，在java中，同一个对象的临界区，在同一时间只有一个允许被访问。

静态方法则有不同的行为。用synchronized关键字声明的静态方法，同时只能够被一个执行线程访问，但是其他线程可以访问这个对象的非静态方法。两个线程可以访问一个对象的两个不同的synchronized方法。即其中一个是静态方法，另一个是非静态方法。

一个对象的方法采用synchronized关键字进行声明，只能被一个线程访问。如果线程A正在执行一个同步方法syncMethodA()，线程B要执行这个对象的其他同步方法syncMethodB()，线程B将被阻塞知道线程A访问完。但是如果线程B访问的是同一个类的不同对象，那么两个线程都不会被阻塞。

## 使用非依赖属性实现同步

在一个类里面声明多个Object对象，然后分别以该对象为单位实现一个完整的逻辑，被synchronized修饰的不同的对象之间彼此不可见。

## 在同步代码块中使用条件

Java在Object类中通关了wait()、notify()和notifyAll方法。线程可以在同步代码块中调用wait()方法，如果在同步代码块之间调用wait()方法，JVM将跑出IllegalMoniorStateException异常。当一个线程调用wait()方法时，JVM将这个线程置入休眠，并且释放控制这个同步块的对象，同时允许其它线程执行这个对象控制的其它同步代码块。为了唤醒这个线程，必须在这个对象控制的某个同步代码块中调用notify()或者notifyAll()方法。

**publicsynchronizedvoid** set(){

**while**(storage.size() == maxSize){

**try** {

wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

storage.add(**new** Date());

System.***out***.printf("Set：%d",storage.size());

notifyAll();

}

}

注意点：1.while 2.wait()和notify()/notifyAll()的配合

首先，set()方法检查存储列表storage是否还有空间，如果已经满了，就调用wait()方法挂起线程并等待空余空间出现。其次，当其它线程调用notifyAll()方法时，挂起的线程将被唤醒并且再次检查这个条件。notifyAll()方法并不保证哪个线程会被唤醒，这个过程持续进行指导存储列表有空余空间出现，然后生产者将生成一个新的数据并且存入存储列表storage。

## 使用锁实现同步

基于Lock接口及其实现类ReetrantLock实现。相比synchronized来说的好处：

* 支持更灵活的同步代码块结构
* tryLock(): boolean
* 允许分离读和写操作
* 更好的性能

个人感觉范围更小，在方法体内使用。

注意点：1、如果在临界区使用了try-catch块，不要忘了在finally中调用unlock()方法，否则可能会导致死锁。

1. 对于可能出现死锁情况的处理

## 使用读写锁实现同步数据访问

ReadWriteLock接口及其实现类 ReentrantReadWriteLock

包括两个锁：读操作锁和写操作锁。其中读操作锁允许多个线程同时访问，但是使用写操作锁时只允许一个线程进行。一个线程在执行读操作时，其他线程不能执行读操作。

读操作锁:readLock() 写操作锁：writeLock()

注意：当获取Lock接口的读操作锁是，不可以进行修改操作，否则会引起数据不一致的情况。

**publicdouble** getPrice(){

lock.readLock().lock();

**double**value = price;

lock.readLock().unlock();

**return**value;

}

## 修改锁的公平性

ReentrantLock和ReentrantReadWriteLock 类的构造器都含有一个布尔参数fair,它允许你控制这两个类的行为。默认fair值是false.，它成为非公平模式（Non-fair Mode）。True为公平模式。

非公平模式：当很多线程在等待锁时，锁将随机选择他们当中的一个来访问临界区。

公平模式：当很多线程在等待锁时，锁将选择他们当中等待时间最长的一个来访问临界区。

这两种模式只适用于lock和unlock()方法。

## 在锁中使用多条件（Mulpitle Condition）

一个锁可能关联一个或者多个条件，这些条件通过Condition接口声明。目的是允许线程获取锁并且查看等待的某一个条件是否满足，如果不满足就挂起直到某个线程唤醒他们。Condition接口提供了挂起线程和唤起线程的机制。

**publicvoid** insert(String line){

lock.lock();

**try** {

**while**(buffer.size() == maxSize){

space.await();

}

buffer.offer(line);

lines.signalAll();//唤醒

}**catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

注意点：在使用条件的时候，必须获取这个条件绑定的锁，所以带条件的代码必须在调用Lock对象的lock()方法和unlock()之间。

当线程调用条件的await()方法是，它将自动释放这个条件绑定的锁，其他某个线程才可以获取这个锁并且执行相同的操作，或者执行这个锁保护的另一个临界区代码。

当一个线程调用了条件对象的singal()或者singalAll()方法后，一个或者多个在该条件上挂起的线程将被唤醒。

问题:synchronized 以对象为单位，锁以什么为单位？

# 并发集合

阻塞式集合：当集合已满或者为空时，被调用的添加或移除方法就不能立即被执行，那么调用这个方法的线程将被阻塞，一直到该方法可以被成功执行。

非阻塞式集合：如果方法不能立即被执行，则返回null值或抛出异常，但是调用这个方法线程不会被阻塞。

|  |  |
| --- | --- |
| 阻塞式集合 | 非阻塞式集合 |
| LinkedBlockingDeque | ConcurrentLinkedDeque |
| LinkedTransferQueue | ConcurrentSkipListMap |
| PriorityBlockingQueue |  |
| DelayQueue |  |

## ConcurrentLinkedDeque:非阻塞式并发列表

方法：

|  |  |  |
| --- | --- | --- |
| 方法名 | 方法说明 | 列表为空 |
| getFirst()/getLast() | 返回列表的第一个/最后一个元素,元素不移除 | 抛NoSuchElementException异常 |
| Peek()/peekFirst()、peekLast() | 返回列表的第一个/最后一个元素,元素不移除 | 返回null |
| Remove()/removeFirst()/removeLast() | 返回列表的第一个/最后一个元素,元素移除 | 抛NoSuchElementException异常 |

## LinkedBlockingDeque:阻塞式线程安全列表

|  |  |  |
| --- | --- | --- |
| 方法名 | 方法说明 | 列表满/空 |
| Put() | 插入元素 | 列表满时阻塞 |
| Take() | 取字符串 | 列表空时阻塞 |
| takeFirst()/takeLast() | 返回列表的第一个/最后一个元素,元素移除 | 阻塞 |
| getFirst()/getLast() | 返回列表的第一个/最后一个元素,元素不移除 | 抛NoSuchElementException异常 |
| Peek()/peekFirst()/peekLast | 返回列表的第一个/最后一个元素,元素不移除 | 返回null |
| Poll()/pollFirst()/pollLast | 返回列表的第一个/最后一个元素,元素移除 | 返回null |
| Add()/addFirst()/addLast() | 添加到列表第一位/最后一位 | 抛IllegalStateException异常 |

## PriorityBlockingQueue：按照优先级排序的阻塞式线程安全列表

所有添加进PriorityBlocingQueue的元素必须实现Comparable接口。比较的原则是以当前对象为中心。参考TreeMap

|  |  |  |
| --- | --- | --- |
| 方法名 | 方法说明 | 集合空/满 |
| Clear | 移除队列所有元素 |  |
| Take() | 返回队列中的第一个元素，元素移除 | 阻塞 |

## DelayQueue:带有延迟元素的线程安全列表

存放在DelayQueue类中的元素必须继承Delayed接口，接口强制执行以下两个方法：

compareTo(Delayed o)：以当前对象为中心，小于参数值，返回小于零的值，大于…

getDalay(TimeUnit unit):返回激活日期的剩余时间

|  |  |
| --- | --- |
| 方法 | 方法说明 |
| offer(E e) | 参数对应元素插入到队列中 |
| peek | 返回第一个元素，不删除 |
| take | 返回第一个元素，删除 |

## ConcurrentnavigatableMap：使用线程安全可遍历映射

ConcurrentSkipListMap类

|  |  |
| --- | --- |
| 方法名 | 方法说明 |
| HeadMap(K toKey) | 返回映射中所有键值小于参数值toKey的子映射 |
| TailMap(K fromKey) | 返回映射中所有键值大于参数值fromKey的子映射 |
| putIfAbsent(K key,V value) | 如果不存在key,那么就将key和value插入到集合中。 |
| PollLastEntry() | 返回并移除映射中的最后一个Map.Entry对象 |
| Replace(K key,V value) | 如果映射中已经存在键key，则用参数中的value替换现有的值。 |

## ThreadLocalRandom：生成并发随机数

它是线程本地变量。每个生成随机数的线程都有一个不同的生成器，但是都在同一个类中被管理，对程序员来讲是透明的。

current(): static

返回与当前线程关联的TaskLocalRandom对象。如果调用这个方法的线程还没有关联随机数对象，就会生成一个新的。

## 使用原子变量

AtomicLong/AtomicBoolean/AtomicInteger/AtomicReference

CAS(Compare And Set)原子操作：当一个线程在对原子变量操作时，如果其他线程也试图对同一个原子变量进行操作，原子变量的实现类提供了一套机制来检查操作是否在一步内完成。一般来说，这个操作先获取变量值，然后在本地改变变量的值，然后试图用这个改变的值去替换之前的值。如果之前的值没有被其他线程改变，就可以执行这个替换操作。否则，方法将再执行这个操作。

|  |  |
| --- | --- |
| 方法名 | 方法说明 |
| get/set | 赋值/获取值 |
| getAndAdd | 增加值 |

## ConcurrentHashMap

|  |  |
| --- | --- |
| 方法名 | 方法说明 |
| putIfAbsent | key存在，返回value；不存在，插入 |
| replace | key存在，替换；不存在，返回null |

/\*\*

\* 单词出现次数统计

\* **@param** key 不会为null

\*/

**protected** **void** wordCount(String key){

/\*\*

\* putIfAbsent

\* key存在：返回对应的value值，

\* key不存在：使用给定的value插入key-value，返回null

\*/

Long result = wordCountMap.putIfAbsent(key,(**long**)0);

/\*\*

\* 为null:不存在给定的key，插入

\* 不为null：存在给定的key,返回value

\*/

**long** finalResult = result == **null**?0:result;

/\*\*

\* replace

\* key存在：使用给定的value替换原有key对应value值

\* key不存在：返回null

\*/

wordCountMap.replace(key, ++finalResult);

}