Java 7 并发编程实战手册

第 1 章 线程管理

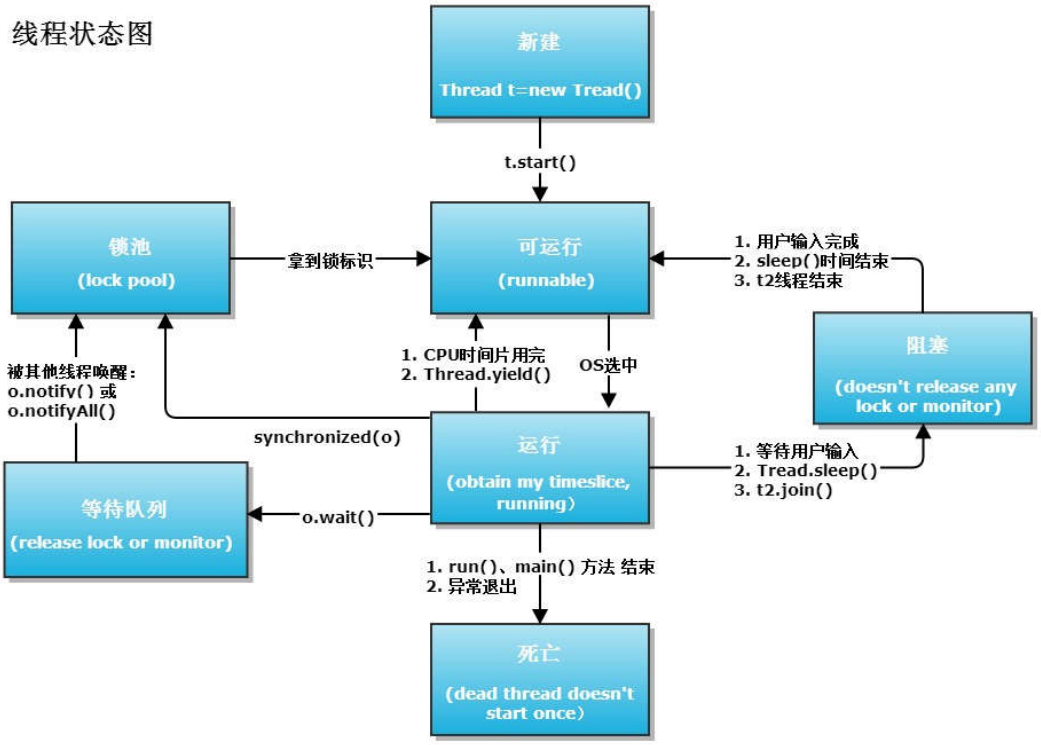
1.1 简介

并发(Concurrency)：指一系列任务的同时运行。如果电脑有多个处理器或者有一个多核处理器，这个同时性是真正意义的并发，但是一台电脑只有一个单核处理器，这个同时性并不是真正的并发。

进程级(Process_level)并发：在 windows 中同时进行多个应用程序任务，这样的是进程级并发。

线程(Thread)：在一个进程内可以有多个同时进行的任务。

并行：多核处理器中使用多线程执行应用。



1.2 线程的创建和运行

两种方式：

- 继承 Thread 类，并且覆盖 run（）方法
- 创建一个实现 Runnable 接口的类，以此类为参数调用 Thread 构造器创建 Thread 对象

线程的信息

Priority：线程的优先级

Status：线程的状态，线程的状态有 6 种：new、runnable、blocked、waiting、time waiting 或者 terminated

如果是实现 Runnable 接口，那么可以通过 Thread 的静态方法 currentThread()来获取 Thread 对象。

1.3 线程的中断

Java 提供中断机制，可以用来结束一个线程。要求线程检查它是否被中断，然后决定是否响应这个中断请求，线程允许忽略中断请求并且继续执行。

线程在休眠中,如果被中断会抛出 InterruptedException 异常。

thread.isInterrupted（）来获取线程的中断状态，isInterrupt（）不能改变 interrupted 属性的值，interrupt（）方法能设置 interrupted 属性为 false。

线程中断的控制：

如果线程实现了复杂的算法或进行递归调用等复杂任务，可使用 InterruptedException 异常来控制线程的中断。当检查到线程中断时，抛出这个异常，在 run（）中捕获并处理这个异常。

1.4 线程的休眠和恢复

线程使用 sleep（）进行休眠，在休眠时线程不占用 CPU 的资源，CPU 可以去执行其他任务。当休眠结束 JVM 会分给它 CPU 时钟，线程继续执行这个线程。

TimeUnit.SECONDS.sleep(1).

Thread.sleep(1).

1.5 等待线程的终止

在一些情形下，如需要先初始化一些资源，初始化完成之后，线程再继续执行。可以使用 Thread 类的 join（）方法，当一个线程对象的 join（）方法被调用时，调用它的线程将被挂起，直到这个线程对象完成它的任务。

Join（）的另外两种形式，如 thread1 调用 thread2.join(1000)，thread1 被挂起的时间到达 1000ms 或者 thread2 运行完成 thread1 都将结束挂起。

Join（long milliseconds）

Join（long milliseconds, long nanos）

1.6 守护线程的创建和运行

守护线程(daemon)：优先级很低，通常来说一个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的线程的时候，守护线程结束后，jvm 也就结束了这个程序。

1.7 线程中不可控异常的处理

Java 中有两种异常

- 非运行时异常（Checked Exception），这种异常必须在方法声明 throws 语句.或则在方法体中捕获。
- 运行时异常(Unchecked Exception): 这种异常不必在方法声明中指定,也不需要在方法体中捕获。

Run（）方法不支持 throw 语句，所以当线程对象 run（）抛出非运行异常，必须捕获并处理。运行时异常从 run（）方法中抛出，默认行为是在控制台输出堆栈记录并且退出程序。

1.8 线程局部变量的使用

以实现了 runnable 接口的类的实例作为参数创建多个线程对象，这些线程将共享相同的属性，为了使这个对象的属性不被所有线程共享，Java 并发 API 提供了线程局部变量（Thread-Local Variable）的机制。

如果定义一个 private 变量，每个线程都一个自己的 private 变量属性（startDate hashcode 不一样）；

```
Thread 10 startDate's hashcode: -2066788901
Starting Thread: 10 : Mon Nov 21 10:55:27 CST 2016
Thread 11 startDate's hashcode: -2066787061
Starting Thread: 11 : Mon Nov 21 10:55:29 CST 2016
Thread Finished: 10 : Mon Nov 21 10:55:29 CST 2016
Thread 12 startDate's hashcode: -2066784901
Starting Thread: 12 : Mon Nov 21 10:55:31 CST 2016
Thread Finished: 12 : Mon Nov 21 10:55:31 CST 2016
Thread Finished: 11 : Mon Nov 21 10:55:31 CST 2016
```

如果定义了**private static** ThreadLocal变量，每个线程共享一个starDate变量（startDate hashcode一样）；

```
private static ThreadLocal<Date> startDate= new ThreadLocal<Date>() {
    protected Date initialValue(){
        return new Date();
    }
};
```

```
Thread 10 startDate's hashcode: 1680256105
Starting Thread: 10 : Mon Nov 21 10:54:22 CST 2016
Thread 11 startDate's hashcode: 1680256105
Starting Thread: 11 : Mon Nov 21 10:54:24 CST 2016
Thread Finished: 10 : Mon Nov 21 10:54:22 CST 2016
Thread 12 startDate's hashcode: 1680256105
Starting Thread: 12 : Mon Nov 21 10:54:26 CST 2016
Thread Finished: 11 : Mon Nov 21 10:54:24 CST 2016
Thread Finished: 12 : Mon Nov 21 10:54:26 CST 2016
```

如果一个变量是成员变量，那么多个线程对**同一个对象**的成员变量进行操作时，它们对该成员变量是彼此影响的，一个线程对成员变量的改变会影响到另一个线程。

如果一个变量是局部变量，那么每个线程都会有一个该局部变量的拷贝，一个线程对该局部变量的改变不会影响到其他线程。

1.9 线程的分组

Java 并发 API 提供线程分组功能，把一组线程当成一个单一的单元，对组内的对象进行访问并操作它们。对于这组线程，如果想中断线程的运行只需一个单一的调用，所有的线程都会被中断。

ThreadGroup(线程组)：表示一组线程，可以包含线程对象，也可以包含线程组对象,它是一个树形结构。

线程组中不可控异常处理：

当线程抛出非捕获异常时，JVM 将寻找 3 种可能的处理器。首先寻找抛出此异常的线程的非捕获异常处理器，若不存在，找线程所在线程组的非捕获异常处理器；若也不存在，将寻找默认的非捕获异常处理器。

1.10 使用工厂类创建线程

实现 ThreadFactory 接口定义一个工厂类。

使用工厂类可以将对象的创建集中化：

- 更容易修改类，或者改变创建对象的方式；
- 更容易为有限资源限制创建对象的数目；
- 更容易为创建的对象生成统计数据；

第 2 章 线程同步基础

2.1 简介

临界区（Critical Section）：一个用以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执行。

当一个线程试图访问临界区时，若无其他线程正在执行临界区，它就可以进入临界区；若已经有其他线程进入临界区，它被同步机制挂起，直到已经进入临界区的线程离开临界区；若等待进入临界区的线程不止一个，JVM 选择其中一个进入，其余继续等待。

两种基本同步机制：

- Synchronized 关键字
- Lock 接口

2.2 使用 synchronized 实现同步

使用 synchronized 关键字控制一个方法的并发访问，一个对象用 synchronized 关键字声明，同一时间只有一个线程允许访问它，其他线程被挂起。如线程 A 正在执行同步方法 synchA（），线程 B 要执行这个对象的另一个同步方法 synchB（），线程 B 将被阻塞直到线程 A 访问完。

注意：对于 synchronized 静态方法，同时只能被一个线程访问，但其他线程可以访问这个对象的非静态方法。

临界区代码应尽可能短。

使用非依赖属性实现同步：

```
    private final Object controlCinema1, controlCinema2;
public boolean returnTickets1 (int number) {
        synchronized (controlCinema1) {
            vacanciesCinema1+=number;
            return true;
        }
}
```

2.3 在同步代码中使用条件

生产者-消费者问题

Wait（）方法：使线程休眠，释放控制这个同步代码块的对象，允许其他线程执行这个对象控制的其他同步代码块；在同步代码块中调用。

Notify（）/notifyAll（）方法：唤醒线程；在某个同步代码块中调用。

2.4 使用锁实现同步

private final Lock queueLock=new ReentrantLock();

使用 Lock 接口及其实现类：

- 支持更灵活的同步代码块结构。Synchronized 只能在同一个 synchronized 快结构中获取和释放控制。Lock 接口允许实现更复杂的临界区结构，控制的获取和释放不出现在同一个快结构中。
- tryLock（）可用来视图获取锁，如果锁已被其他线程获取，它将返回 false 并往下继续执行代码。而使用 synchronized，如果线程 A 试图执行一个同步代码块，如果 B 线程已经在执行这个代码块了，线程 A 将被挂起，直到 B 线程执行完。
- lock 接口允许读写分离，允许多个读线程和只有一个写线程。
- lock 拥有比 synchronized 更好的性能。

ReentrantLock 类也允许使用递归调用。

2.4.1 使用读写锁实现同步数据访问

ReadWriteLock 接口和它的实现类 ReentrantReadWriteLock 有两个锁，读操作锁和写操作锁。使用 ReadLock 读操作锁时，可以由多个线程同时访问，而使用 WriteLock 操作锁时，其他线程不能够执行读操作。

2.4.2 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，默认值是 false，此时锁为非公平模式。

非公平模式：很多线程在等待锁，锁将选择一个来访问临界区，这个选择是没有任何约束的；

Fair=true 公平模式：很多线程在等待锁，锁选择等待时间最长的线程；

公平和非公平策略只适用于 lock() 和 unlock()方法。而 Lock 接口的 tryLock 方法没有将线程置于休眠，fair 属性并不影响这个方法。

2.4.3 在锁中使用多条件

一个锁可能关联一个或则多个条件，这些条件通过 condition 接口声明。目的是运行线程获取锁并且查看等待的某一个条件是否满足，如果不满足则挂起直到某个线程唤醒它们。condition 接口提供了挂起线程和唤起线程的机制。

线程调用 await()方法将自动释放这个条件绑定的锁，其他线程才可以获取这个锁执行相同操作或这个锁保护的另一个临界区代码。

调用 await()方法进入休眠的线程可能会被中断，所以必须处理 InterruptedException 异常。

条件必须用在 lock() 和 unlock() 方法之间。

线程调用 await(long time, TimeUnit unit)方法直到以下情况之一之前，线程将一直处于休眠状态：

1. 其他某个线程中断当前线程
2. 其他某个线程调用了将当前线程挂起的条件的 singal()或 singalAll()方法
3. 指定的等待时间已经过去

awaitUninterruptibly（）是不可中断的，线程将休眠直到其他某个线程调用了将他挂起的条件的 singal()或则 signalAll()方法。

awaitUntil(Date date)方法直到发生以下情况之一之前，线程将一直处于休眠状态：
1. 其他某个线程中断当前线程
2. 其他某个线程调用了将他挂起的条件的 dingal()或 signalAll()方法
3. 指定的最后期限到了

第 6 章 并发集合

6.1 简介

大多数集合类不能直接用于并发应用，集合没有对本身数据的并发访问进行控制。

Java 提供了两类适用于并发场景下的集合：

- **阻塞式集合**（Blocking Collection）：这类集合包括添加和移除数据的方法，当集合已满或为空时，被调用的添加或者移除方法就不能立即被执行，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**（Non-Blocking Collection）：这类集合也包括添加和移除数据的方法，只是如果方法不能立即被执行，则返回 null 或抛出异常，但是调用这个方法的线程不会被阻塞。

6.2 使用非阻塞式线程安全列表

并发安全列表允许不同的线程在同一时间 add 或 remove 列表中的元素，而不会造成数据不一致。

ConcurrentLinkedDeque 类实现了非阻塞式并发列表。非阻塞式列表的操作中如果不能立即运行，方法会抛出异常或返回 null。

6.3 使用阻塞式线程安全列表

LinkedBlockingDeque 实现了阻塞式列表。

与非阻塞式列表的区别是，如果执行操作不能立即执行，调用这个操作的线程被阻塞直到操作可以执行。

线程阻塞时如果线程被中断，方法会抛出 InterruptException 异常，必须捕获并处理异常。

6.4 使用按优先级排序的阻塞式线程安全列表

PriorityBlockingQueue 阻塞式线程安全列表

添加进 PriorityBlockingQueue 的元素必须实现 Comparable 接口。

列表中的元素都有优先级属性，优先级最高的元素是队列的第一个元素。

6.5 使用带有延迟元素的线程安全列表

DelayQueue 存放带有激活日期的元素，调用方法从队列中返回或提取元素时，未来日期的元素（延迟未到期）将被忽略。

存放到 DelayQueue 的元素必须继承 Delayed 接口并覆盖以下两个方法：

1.getDelay(TimeUnit unit)：返回与此对象相关的剩余延迟时间，以给定的时间单位表示。该方法是在 compareTo 中调用的。

2.compareTo(Delayed o)：队列会根据此方法来获取超时的元素。

6.6 使用线程安全可遍历映射

ConcurrentSkipListMap<K,V>类存放键值对：

- 一个键值（key），元素的标识且唯一；
- 元素其他部分数据

插入元素时 ConcurrentSkipListMap 使用键值排序所有元素。

6.7 生成并发随机数

ThreadLocalRandom 类可以用来生成伪随机数

ThreadLocalRandom.current()获取生成器

每个生成随机数的线程都有一个不同的生成器，但都在同一个类中被管理。

6.8 使用原子变量

atomic variable

CAS 原子操作：线程对一个原子变量进行操作时，先取变量值，在本地改变变量值，试图用改变的值替换之前的值，若之前的值没有被其他线程改变，就执行这个替换操作，否则重新执行这个操作。

使用原子变量保证了多线程在同一时间操作一个原子变量而不会产生数据不一致，其性能优于使用同步机制保护的普通变量。

6.9 使用原子数组

Atomic Array

基于原子变量，Java 提供对 integer 或 long 数字数组的原子操作。