问题不大，礼多人不怪
码龄13年 暂无认证

189 原创 | 3万+ 周排名 | 9227 总排名 | 47万+ 访问 | 等级

5567 积分 | 134 粉丝 | 108 获赞 | 19 评论 | 195 收藏

私信 关注

搜博主文章

热门文章
vim 翻页 33794
db2look 用法 21019
db2 查询表、模式、用户和权限 19535
db2 查看数据库日志及其报错信息 13760

分类专栏
redis
security
nginx 2篇
vue 1篇
杂记 2篇
软考 3篇

高级并发对象（High Level Concurrency Objects）

原创 问题不大，礼多人不怪 于 2016-07-09 01:17:43 发布 1133 收藏 版权
分类专栏： JAVA 文章标签： java concurrent 并发 executor 线程池

JAVA 专栏收录该内容 0 订阅 55 篇文章 订阅专栏

目前为止，重点关注低级别的api，它们从一开始就是java平台的一部分，对于基本的任务，这些api已经足够使用了，但是，对于更高级的任务，就需要高级别的构建，尤其对当今重发利用多处理器和多核心系统的大量并发应用。

在这一节中，我们重点关注java平台5.0中提供的高级并发特性，大部分的特性都在java.util.concurrent包中实现，在java集合框架中也添加了新的并发数据结构。

- 对象锁，支持简化许多并发应用锁定方式理套件的java.util.concurrent包提供。
- 并发集合，使大量集合数据的管理更加容易，并大大减少并发的需求。
- 原子变量，拥有最小化同步的特性，并有助于避免内存一致性错误。
- ThreadLocalRandom，为多线程提供有效的随机数。

对象锁（Lock Objects）

同步代码依赖一种简单的重进入锁，这种锁易于使用，但是有很多限制。更加复杂的锁方式则由java.util.concurrent.locks包提供，我们不会详细测试这个包，但是，会重点关注最基本的接口，Lock。

对象锁工作方式类似于同步代码中使用的隐式锁，在隐式锁中，在一个时刻仅有一个线程可拥有一个对象锁。对象锁同样支持wait/notify机制，尽管他们关联条件对象。

相对隐式锁，对象锁的最大优势是它们可以收回获取锁的请求，tryLock方法会收回，当锁不能立即可用或者在时间超时之前（如果定义了的话），lockInterruptibly方法会收回，当在获取锁之前另外一个线程发送一个中断信息。

让我们使用对象锁来解决在活锁章节中碰到的死锁问题。A和B都会注意另外一个朋友将会对他躬

身，我们模拟这个提高后的模型，在继续躲身之前，我们的Friend对象必须获取所有的参与者的锁。以下是改进模型的源代码，Safelock，为了演示这种多功能的方式，我们假设A和B如此迷恋他们新的安全躲身技能，他们可以不用停止给对方躲身。

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class SafeLock{
    static class Friend{
        private final String name;
        private final Lock lock = new ReentrantLock();
        public Friend(String name){
            this.name = name;
        }
        public String getName(){
            return this.name;
        }
        public boolean impendingB
```

复制

执行器（Executor）

在前面所有的例子中，正在处理的任务和新的线程有密切的关系， Runnable对象，线程本身，这些在小的应用中可以很好的工作，但是在大规模的应用中，将线程分开管理并让应用在空闲时创建线程将会变得更加重要，总所周知包含这些功能的对象是executors，接下来的章节会详细介绍执行器。

- 接口Executor，定义三种执行对象类型。
- 线程池，最广泛的执行器实现方式
- Fork/Join，充分利用多处理器的一个框架。

执行器接口（Executor Interface）

java.util.concurrent包中定义了三种执行器接口：

- Executor，支持启动新任务的一个简单接口。
- ExecutorService，继承Executor添加了一些管理生命周期的新特性，独立任务和执行器本身。
- ScheduleExecutorService，是ExecutorService的一个子接口，支持future和定期任务执行

通常，执行器对象变量被声明为这三种接口类型，而不是executor类型

Executor接口

Executor接口提供一个简单的方法，被设计一种快速替代普通创建线程方式，假设，r是一个Runnable对象，e是一个Executor对象，那么你可以用

```
e.execute(r);
```

替代

```
(new Thread(r)).start()
```

然而，定义execute可以注意更少的细节，低级别方式创建一个新线程和快速启动它，依赖Executor实现，execute可能做同样的事情，但它更加合适让一个工作的线程来执行r对象，或者将r对象放在等待工作的队列中。

在java.util.concurrent包中的executor实现被设计成充分使用ExecutorService和ScheduleExecutorService接口的优点，尽管他们用基本的Executor接口工作。

ExecutorService接口

这个接口使用一个更加简单但是多功能的submit方法来补充执行任务，类似execute，submit方法接受一个Runnable对象，同时也接收Callable对象，这个对象允许任务返回一个值。submit方法返回一个Future类型对象，这对象用来接收Callable返回值并管理Callable和Runnable任务的状态。

ExecutorService还提供能够提交大集合Callable对象的方法，另外，它还提供很多方法来停止执行器，为了立即停止，任务应该能够正确的处理中断。

ScheduleExecutorService接口

这个接口提供schedule方法来弥补父接口，这个允许延迟指定时间来执行Runnable或者Callable任务。另外，这接口定义了scheduleAtFixedRate 和 scheduleWithFixedDelay，这允许它在指定的时间间隔来重复执行指定任务。

线程池（Thread Pools）

在java.util.concurrent包中大部分执行器使用线程池来实现，线程池包含工作线程，这种类型线程与它执行的Runnable和Callable任务分开存在，并经常被用于执行多任务。

使用工作线程可以降低资源消耗在线程创建方面，线程对象使用很多内存，而在大型应用中，创建和消除线程对象需要耗费很多内存资源。

一个普通线程池类型是固定大小线程池，这种类型线程池有一个指定数量的一直工作的线程；如果一个线程在使用的时候终止了，那么它会立即创建一个新的线程来代替它，任务通过一个内部队列来提交，这个队列用来保存额外任务，当任务数超过线程数时。

固定大小线程池的一个优势是应用能够优雅的使用它。为了理解这个，设想一个web应用服务，每个http请求都有独立的线程来处理，如果应用只是简单的创建一个新的线程来处理每一个新的请求，当系统收到的请求超过它能够立即处理的数量，应用将会突然停止响应所有的请求。使用限制线程数量，应用将不能及时响应他们接收到的请求，但是它们在系统能够支撑范围尽快的处理请求。

一种使用固定大小线程池简单创建执行器的方式是调用java.util.concurrent.Executors.newFixedThreadPool工厂方法,这个类还提供了如下工厂方法：

- newCachedThreadPool 方法创建一个可扩展线程池的执行器。这种执行器适用于启动很多短暂任务的应用。
- newSingleThreadExecutor方法创建在同一个时刻只执行一个任务的执行器
- 一些工厂方法是ScheduleExecutorService版本

如果上述工厂方法提供的执行器没有满足你的需求，构造java.util.concurrent.ThreadPoolExecutor 或者java.util.concurrent.ScheduledThreadPoolExecutor实例对象将给你额外的可选参数。

Fork/Join

Fork/Join框架是ExecutorService接口的一种实现，能够帮助你充分利用多处理器，它被设计可以递归的分解成小块来工作。目的是为了充分使用处理能力来提高你应用的性能。

和其它ExecutorService实现一样，fork/join框架将任务分开到在一个线程池的工作线程中，这框架是清晰的因为它使用抢断算法，工作线程完成任务时可以从其它正在忙的线程中抢断任务。

fork/join框架的核心是ForkJoinPool类，扩展了AbsractExecutorService类，ForkJoinPool实现工作抢断算法并且可以执行ForkJoinTask进程。

基本使用

使用fork/join框架的第一步是写执行工作部分的代码，你的代码应该和下面的伪代码类似：

```
if(my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces invoke the two pieces and wait for the res
```

在ForkJoinTask子类中封装这些代码，通常它特定的类型，RecursiveTask（能够返回一个结果）或者RecursiveAction

你的ForkJoinTask子类准备好之后，创建代表将要被完成的工作并将ForkJoinTask实例传递给invoke()方法。

模糊的清晰

为了帮助你理解fork/join框架工作原理，举如下例子，假设你想要将一张图片模糊化，原照片是由一组整数代表的，每个整数包含单一一个像素的颜色值，模糊化后的图片也是由一组和原照片同样大小整数组成。

执行模糊化处理工作是通过原数组一个时刻处理一个像素完成的。每个像素都是它周围像素（红、绿、蓝部分都是平均的）的平均值，并且结果被放在目标数组中，由于一张照片是一个很大的数组，这样处理将会花很长时间。你可以通过实现使用fork/join框架算法充分利用多处理系统上并发处理功能。

```
public class ForkBlur extends RecursiveAction{
    private int[] msource;
    private int mStart;
    private int mLength;
    private int[] mDestination;
    private int mBlurWidth = 15;
    public ForkBlur(int[] src,int start,int length,int[] dst){
        this.mSource = src;
        this.mStart = start;
        this.mLength = lenght;
        this.mDestination = dst;
    }
    protected void computeDirectly(){
        int sidePixels = (mBlurWidth - 1)/2;
        for(int index=mStart;inde art+mLength;index++){
```

现在你实现抽象方法compute(),可以直接处理模糊或者切分成两个更小的任务，一个简单数组长度阀值决定工作任务是执行还是分割。

```
protected static int sThreshold = 1000;
protected void compute(){
    if(mLength < sThreshold){
        computeDirectly();
```

```
        computeDirectly();
        return;
    }
    int split = mLength/2;
    invokeAll(new ForkBlur(mSource,mStart,split,mDestination),new ForkBlur(m
}
```

如果这个方法在RecursiveAction类的子类中，那么直接设置在ForkJoinTask中的任务就可以执行，包含如下步骤：

1. 创建一个代表所有将要处理工作的任务
2. 创建将会执行任务的ForkJoinPool

```
ForkJoinPool pool = new ForkJoinPool();
```

3. 执行任务

```
pool.invoke(fb);
```

获取完整的源代码，包括创建目标照片文件的代码，查看ForkBlur

标准实现（Standard Implementations ）

除了使用fork/join框架来实现在多处理器系统上执行并发任务的自定义算法（如ForkBlur.java），在Java se中的 一般有用的特性已经有使用fork/join框架实现了。java se8中一个实现例子，java.util.Arrays类并行排序paralleSort()方法，这些方法类似于sort()方法，但是通过使用fork/join框架达到并发效果。在多处理器系统，大数组并行排序比次序排序更快。但是这些方法怎么用fork/join框架实现不在java手册范围之内，可以在java api文档中查阅相关信息。

另外一个使用fork/join框架实现并发的是java.util.streams包中的方法，这是java se8发布的lambda表示试的一部分，了解更多信息，请查阅Lambda Expressions

并发集合（Concurrent Collections）

java.util.concurrent包含许多附加java集合框架，这些更容易用集合接口提供的分类：

- BlockingQueue，阻塞栈定义了先进先出数据结构，当你试图向满栈添加数据或者从空栈中获取数据，将发生阻塞活超时

数据，特发生阻塞活超时

- ConcurrentMap，它是java.utl.Map的一个子类，定义一些有用的原子操作，这些操作包括删除或者更新一个键值对，当键存在时；或者添加一个键值对，当键不存在时。按照这些原子操作，可以帮助避免同步，ConcurrentMap标准的实现是ConcurrentHashMap。
- ConcurrentNavigableMap，它是ConcurrentMap的一个子接口，支持近似匹配。它的一个标准实现是ConcurrentSkipListMap，这个类似于TreeMap。

所有的这些集合通过定义两个先后操作现行发生行为关系避免了内存一致性错误，这操作可能是前者往集合中添加对象，后者访问或者删除这些对象。

原子变量（Atomic Variables）

java.util.concurrent.atomic包定义了支持对单一变量原子操作的类。素有的类都有get和set方法，这些方法类似对volatile变量的读写操作，也就是，前者set和后者的get在同一个变量操作上有一个现行发生行为关系。原子操作compareAndSet方法也有内存一致性错误的特质，作为简单原子算法方法，他们适用于整型原子变量

为了了解这个包怎么被使用，返回到我们之前演示线程接口的Counter类：

```
class Counter{
    private int c = 0;
    public void increment(){
        c++;
    }
    public void decrement(){
        c--;
    }
    pubilc int value(){
        return c;
    }
}
```

一种使Counter类在多线程交互中安全的方法是使用synchronized，如SynchronizedCounter：

```
class SynchronizedCounter{
    private int c = 0;
    public synchronized void increment(){
        c++;
    }
```

```
    public synchronized void decrement(){
        c--;
    }
    public synchronized int value(){
        return c;
    }
}
```

对这个简单的类，synchronization是一个可接受的解决方式，但是对于更加复杂的类，我们可能会想要避免因为同步而造成的活跃性问题，使用AtomicInteger代替int可以避免线程冲突，而不用借助于synchronization，如下AtomicCounter:

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter{
    private AtomicInteger c = new AtomicInteger(0);
    public void increment(){
        c.incrementAndGet();
    }
    public void decrement(){
        c.decrementAndGet();
    }
    public int value(){
        return c.get();
    }
}
```

并发随机数（Concurrent Random Numbers）

在jdk7中，java.util.concurrent包含一个便捷的类，ThreadLocalRandom类，在多线程或者ForkJoinTasks应用中使用产生随机数。

为了并发访问，使用ThreadLocalRandom 代替Math.random()，能够减少争夺，而且有更好的性能。

你所需要的仅仅是调用ThreadLocalRandom.current(),然后调用它的方法来获取一个随机数，如下所示

```
int r = ThreadLocalRandom.current().nextInt(4,77);
```