Java中并发工具包

superliu213

Published
with GitBook

Table of Contents

Java中并发工具包

对并发工具包核心功能、接口、类的进行了分析，并用实例进行展示，帮助快速的学习并发工具包的相关功能

通过学习之后需要掌握的技能：

1. 了解并发工具包的体系结构及各部分组成
2. 掌握如何在并发应用中如何使用五种同步器
3. 能够使用Java并发工具包中五种同步器开发多线程应用
4. 能够完成Callable与Future在执行器的使用
5. 能够使用锁于原子操作对象
6. 能够使用流完成并发/并行运算
7. 能够使用Fork/Join框架完成并行编程

并发工具包 concurrent包

并发工具处于java.util.concurrent包下，主要包含：

1. 同步器:为每种特定的同步问题提供了解决方案
2. 执行器：用来管理线程的执行
3. locks包：使用Lock接口为并发编程提供了同步的另外一种替代方案
4. atomic包：提供了不需要锁即可完成并发环境变量使用的原子性操作
5. 并发集合：提供了集合框架中集合的并发版本
6. Fork/Join框架：提供了对并行编程的支持

同步器Semaphore

概念

经典的信号量，通过计数器控制对共享资源的访问

- Semaphore(int count)：创建拥有count个许可证的信号量
- acquire()/acquire(int num)：获取1/num个许可证
- release()/release(int num):释放1/num个许可证

案例

模拟银行柜员服务，两个柜台，三个客户需要柜台提供服务

```
public class SeDemo {
  public static void main(String[] args) {
      Semaphore semaphore = new Semaphore(2);

      Person p1 = new Person("A", semaphore);
      p1.start();
      Person p2 = new Person("B", semaphore);
      p2.start();
      Person p3 = new Person("C", semaphore);
      p3.start();
  }

}

class Person extends Thread {

    private Semaphore semaphore;

    public Person(String name, Semaphore semaphore) {
        super(name);
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        System.out.println(getName() + "is waiting ...");
        try {
            semaphore.acquire();
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(getName() + "is done!");
        semaphore.release();
    }
}
```

同步器CountDownLatch

概念

必须发生指定数量的事件后才可以继续运行

- CountDownLatch(int count):必须发生count个数量才可以打开锁存器
- await()：等待锁存器
- countDown()：触发事件

案例

模拟赛跑比赛，裁判员喊3、2、1，运动员同时起跑

```
public class CDDemo {

    public static void main(String[] args) {
        CountDownLatch countDownLatch = new CountDownLatch(3);
        new Racer("A", countDownLatch).start();
        new Racer("B", countDownLatch).start();
        new Racer("C", countDownLatch).start();
        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(3 - i);
            countDownLatch.countDown();
            if (i == 2) {
                System.out.println("Start");
            }
        }
    }

}
```

```
class Racer extends Thread {

    private CountDownLatch countDownLatch;

    public Racer(String name, CountDownLatch countDownLatch) {
        super(name);
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run() {
        try {
            countDownLatch.await();
            for (int i = 0; i < 3; i++) {
                System.out.println(getName() + ":" + i);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

同步器CyclicBarrier循环屏障

概念

适用于只有多个线程都到达预定点时才可以继续执行

- CyclicBarrier(int num):等待线程的数量
- CyclicBarrier(int num,Runnable action):等待线程的数量以及所有 线程到达后的操作
- await()：到达临界点后暂停线程

案例

斗地主，等待3个人都到了才开始游戏

```
public class CDemo {

    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(3, new Runn
            @Override
            public void run() {
                System.out.println("Game start");
            }
        });

        new Player("A", cyclicBarrier).start();
        new Player("B", cyclicBarrier).start();
        new Player("C", cyclicBarrier).start();
    }
}

class Player extends Thread {
    private CyclicBarrier cyclicBarrier;

    public Player(String name, CyclicBarrier cyclicBarrier) {
        super(name);
        this.cyclicBarrier = cyclicBarrier;
    }

    @Override
    public void run() {
        System.out.println(getName() + " is waiting other players.
        try {
            cyclicBarrier.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

同步器Exchanger

概念

简化两个线程间数据的交换

- Exchanger:指定进行交换的数据类型
- V exchange(V object):等待线程到达，交换数据

案例

模拟简单对话

```
public class EDemo {
    public static void main(String[] args) throws InterruptedExcept
       Exchanger<String> ex = new Exchanger<>();
        new A(ex).start();
        Thread.sleep(1000);
        new B(ex).start();
    }
}

class A extends Thread{
    private Exchanger<String> ex;

    public A(Exchanger<String> ex) {
        this.ex = ex;
    }

    @Override
    public void run() {
        String str = null;

        try {
            str = ex.exchange("Hello?");
            System.out.println(str);
```

```
            str = ex.exchange("A");
            System.out.println(str);

            str = ex.exchange("B");
            System.out.println(str);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class B extends Thread{
    private Exchanger<String> ex;

    public B(Exchanger<String> ex) {
        this.ex = ex;
    }

    @Override
    public void run() {
        String str = null;

        try {
            str = ex.exchange("Hi!");
            System.out.println(str);

            str = ex.exchange("1");
            System.out.println(str);

            str = ex.exchange("2");
            System.out.println(str);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

同步器Phaser

概念

工作方式与CyclicBarrier类似，但是可以定义多个阶段

- Phaser()/Phaser(int num):使用指定0/num个party创建Phaser
- register():注册party
- arriveAndAdvance():到达等待到所有party到达
- arriveAndDeregister():到达时注销线程自己

案例

模拟饭店服务，传菜的，厨师，上菜的

```
public class PDemo {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(1);
        System.out.println("starting...");

        new Worker("服务员", phaser).start();
        new Worker("厨师", phaser).start();
        new Worker("上菜员", phaser).start();

        for (int i = 0; i <= 3; i++) {
            phaser.arriveAndAwaitAdvance();
            System.out.println("Order " + i + " finished!");
        }
        phaser.arriveAndDeregister();
        System.out.println("All done!");
    }
}

class Worker extends Thread {
    private Phaser phaser;

    public Worker(String name, Phaser phaser) {
```

```
            super(name);
            this.phaser = phaser;
            phaser.register();
        }

        @Override
        public void run() {
            for (int i = 0; i < 3; i++) {
                System.out.println("current order is:" + i + ":" + getN
                if (i == 3) {
                    phaser.arriveAndDeregister();
                } else {
                    phaser.arriveAndAwaitAdvance();
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

- 首先，构造函数中要将自己注册到phaser当中
- 其次，处理完成则arriveAndDeregister，处理未完成arriveAndAwaitAdvance
- 还有，有三个订单，对于每个订单必须所有人处理完毕之后，才能继续执行
- 最后，解除注册线程

执行器Callable与Future

概念

用于启动并控制线程的执行 核心接口为Executor，包含一个execute(Runnable)用于指定被执行的线程 ，ExecutorService接口用于控制线程执行和管理线程（ExecutorService继承了Executor）
预定义了如下执行器：
ThreadPoolExecutor/ScheduledThreadPoolExecutor/ForkJoinPool

- Callable: 表示具有返回值的线程 V: 表示返回值类型 call(): 执行任务
- Future: 表示Callable的返回值 V: 返回值类型 get():获取返回值

案例

实现1到100，100到10000的计算

```
public class ESDemo {
    public static void main(String[] args) throws ExecutionExceptio
        ExecutorService es = Executors.newFixedThreadPool(2);

        Future<Integer> r1 = es.submit (new MC (1,100));
        Future<Integer> r2 = es.submit (new MC (100,10000));

        System.out.println (r1.get ()+":"+r2.get ());
        es.shutdown ();
    }
}

class MC implements Callable<Integer> {
    private int begin, end;

    public MC(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = begin; i < end; i++) {
            sum += i;
        }
        return sum;
    }
}
```

锁与原子操作

概念

java.util.concurrent.lock包中提供了对锁的支持，为使用synchronized控制对资源访问提供了替代机制

- 基本操作模型： 访问资源之前申请锁，访问完毕后释放锁
- lock/tryLock: 申请锁
- unlock: 释放锁
- 具体锁类ReentrantLock实现了Lock接口

java.util.concurrent.atom包中提供了对原子操作的支持 ，提供了不需要锁以及其他同步机制就可以进行的一些不可中断操作 ，主要操作为：获取、设置、比较等

案例

用Lock和Atom分别实现多线程

```
public class LDemo {
    public static void main(String[] args) {
        new MT ().start ();
        new MT ().start ();
        new MT ().start ();
        new MT ().start ();
    }
}

class Data {
    static int i = 0;
    static Lock lock = new ReentrantLock ();

    static void operate() {
        lock.lock ();
        i++;
        System.out.println (i);
        lock.unlock ();
    }
}

class MT extends Thread {
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep (1000);
            } catch (InterruptedException e) {
                e.printStackTrace ();
            }
            Data.operate ();
        }
    }
}
```

Atom实现方式主要对Date方法进行修改

```
class Data {
    static AtomicInteger ai = new AtomicInteger (0);

    static void operate() {
        System.out.println (ai.incrementAndGet ());
    }
}
```

流编程

概念

流编程并不属于并发工具包，但是并发编程中过多过少的会使用到 表示数据移动，移动过程中可能会对数据进行处理
不同于IO流，表示流对象
操作分为中间操作和终端操作
中间操作会产生一个新流
终端操作会消费流

- 获取流： stream/parallelSteam
- 操作：sort/max/min/…
  过滤、排序、缩减、映射、收集、迭代

案例

简单展示通过Stream进行获取最大值、排序、计数

```
public class SDemo {
    public static void main(String[] args) {
        List<String> ls = new ArrayList<> ();
        ls.add ("abc");
        ls.add ("def");
        ls.add ("ddd");
        ls.add ("eee");
        ls.add ("def");
        ls.add ("cha");

        Optional<String> max = ls.stream ().max (String::compareTo)
        System.out.println (max.get ());

        ls.stream ().sorted ().forEach (e-> System.out.println (e))

        System.out.println (ls.stream ().distinct ().count ());
    }
}
```

Fork Join框架

概念

分而治之策略，即将任务递归划分成更小的子任务，直到子任务足够小，从而能够被连续地处理掉为止，优势是处理过程可以使用并行发生，这种情况特别适合基于多核处理器的并行编程，根据Java API中定义，分而治之的建议临界点定义在100-1000个操作中的某个位置

- ForkJoinTask：描述任务的抽象类
- ForkJoinPool：管理ForkJoinTask的线程池
- RecursiveAction：ForkJoinTask子类，描述无返回值的任务
- RecursiveTask: ForkJoinTask子类，描述有返回值的任务

案例

计算1-1000000的和

```
public class FJDemo {
    public static void main(String[] args) throws ExecutionExceptio
        ForkJoinPool forkJoinPool = new ForkJoinPool ();

        Future<Long> result = forkJoinPool.submit (new MTask (0, 10
        System.out.println (result.get ());
        forkJoinPool.shutdown ();

    }
}

class MTask extends RecursiveTask<Long> {
    static final int THRESHOLD = 1000;

    private int begin, end;

    public MTask(int begin, int end) {
        this.begin = begin;
        this.end = end;
```

```
    }

    @Override
    protected Long compute() {
        long sum = 0;
        if ((end - begin) <= THRESHOLD) {
            for (int i = begin; i < end; i++) {
                sum += i;
            }
        } else {
            int mid = (begin + end) / 2;
            MTask left = new MTask (begin, mid);
            left.fork ();
            MTask right = new MTask (mid + 1, end);
            right.fork ();
            Long lr = left.join ();
            System.out.println (begin + "-" + mid + ":" + lr);
            Long rr = right.join ();
            System.out.println ((mid + 1) + "-" + end + ":" + rr);
            sum = lr + rr;
        }

        return sum;
    }
}
```