java并发包提供的工具类

Java并发包提供了丰富的并发工具类，包括CountDownLatch、CyclicBarrier、Semaphore等高级同步结构，以及ConcurrentHashMap和ConcurrentSkipListMap等线程安全的容器。这些工具类可以帮助开发人员实现多线程操作、资源控制、线程安全的容器操作以及任务调度等功能。掌握并发包的核心功能对于多线程编程至关重要，而深入理解工具类的设计、实现和适用场景，以及熟练掌握典型的代码用例也是必不可少的。尽管并发包提供了丰富的工具，但在实际应用中很少有机会全面使用，因此建议开发人员着重掌握核心功能，并在实际场景中不断积累经验。通过掌握这些内容，开发人员可以更好地利用Java并发包来提高程序的扩展能力、协调线程间的交互以及传递数据和状态，从而更好地满足业务需求。文章还介绍了CountDownLatch、CyclicBarrier和Semaphore的基本操作组合及使用场景，以及ConcurrentHashMap和ConcurrentSkipListMap的选择原则。文章通过示例代码和逻辑分析，帮助读者更好地理解并发包的使用方法和注意事项。

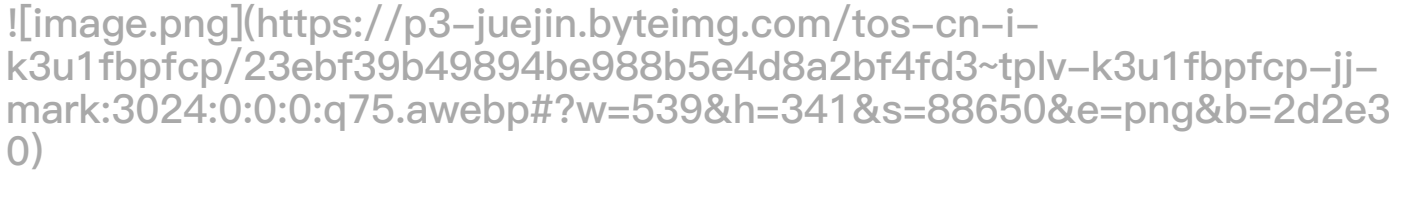
java并发包就是java.util.concurrent及其子包，集中了java并发的各种基础工具类

* 提供了比 synchronized 更加高级的各种同步结构，包括 CountDownLatch、CyclicBarrier、Semaphore 等，可以实现更加丰富的多线程操作，比如利用 Semaphore 作为资源控制器，限制同时进行工作的线程数量。
* 各种线程安全的容器，比如最常见的 ConcurrentHashMap、有序的 ConcurrentSkipListMap，或者通过类似快照机制，实现线程安全的动态数组 CopyOnWriteArrayList 等。
* 各种并发队列实现，如各种 BlockingQueue 实现，比较典型的 ArrayBlockingQueue、 SynchronousQueue 或针对特定场景的 PriorityBlockingQueue 等。
* 强大的 Executor 框架，可以创建各种不同类型的线程池，调度任务运行等，绝大部分情况下，不再需要自己从头实现线程池和任务调度器。

Semaphore信号量

打个比方来说就是在车站、机场等出租车时，当很多空出租车就位时，为防止过度拥挤，调度员指挥排队等待坐车的队伍一次进来 5 个人上车，等这 5 个人坐车出发，再放进去下一批，这和 Semaphore 的工作原理有些类似。

```
import java.util.concurrent.Semaphore;
public class UsualSemaphoreSample {
  public static void main(String[] args) throws InterruptedException {
      System.out.println("Action...GO!");
      Semaphore semaphore = new Semaphore(5);
      for (int i = 0; i < 10; i++) {
          Thread t = new Thread(new SemaphoreWorker(semaphore));
          t.start();
      }
  }
}
class SemaphoreWorker implements Runnable {
  private String name;
  private Semaphore semaphore;
  public SemaphoreWorker(Semaphore semaphore) {
      this.semaphore = semaphore;
  }
  @Override
  public void run() {
      try {
          log("is waiting for a permit!");
         semaphore.acquire();
          log("acquired a permit!");
          log("executed!");
      } catch (InterruptedException e) {
          e.printStackTrace();
      } finally {
          log("released a permit!");
          semaphore.release();
      }
  }
  private void log(String msg){
      if (name == null) {
          name = Thread.currentThread().getName();
      }
      System.out.println(name + " " + msg);
  }
}
```



这段代码是比较典型的 Semaphore 示例，其逻辑是，线程试图获得工作允许，得到许可则进行任务，然后释放许可，这时等待许可的其他线程，就可获得许可进入工作状态，直到全部处理结束。编译运行，我们就能看到 Semaphore 的允许机制对工作线程的限制。但是，从具体节奏来看，其实并不符合我们前面场景的需求，因为本例中 Semaphore 的用法实际是保证，一直有 5 个人可以试图乘车，如果有 1 个人出发了，立即就有排队的人获得许可，而这并不完全符合我们前面的要求。

```
import java.util.concurrent.Semaphore;
public class AbnormalSemaphoreSample {
  public static void main(String[] args) throws InterruptedException {
      Semaphore semaphore = new Semaphore(0);
      for (int i = 0; i < 10; i++) {
          Thread t = new Thread(new MyWorker(semaphore));
          t.start();
      }
      System.out.println(”Action...GO!”);
      semaphore.release(5);
      System.out.println(”Wait for permits off”);
      while (semaphore.availablePermits()!=0) {
          Thread.sleep(100L);
      }
      System.out.println(”Action...GO again!”);
      semaphore.release(5);
  }
}
class MyWorker implements Runnable {
  private Semaphore semaphore;
  public MyWorker(Semaphore semaphore) {
      this.semaphore = semaphore;
  }
  @Override
  public void run() {
      try {
          semaphore.acquire();
          System.out.println(”Executed!”);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
  }
}
```

注意，上面的代码，更侧重的是演示 Semaphore 的功能以及局限性，其实有很多线程编程中的反实践，比如使用了 sleep 来协调任务执行，而且使用轮询调用 availalePermits 来检测信号量获取情况，这都是很低效并且脆弱的，通常只是用在测试或者诊断场景。
总的来说，我们可以看出 Semaphore 就是个计数器，其基本逻辑基于 acquire/release，并没有太复杂的同步逻辑。
如果 Semaphore 的数值被初始化为 1，那么一个线程就可以通过 acquire 进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比如互斥锁是有持有者的，而对于 Semaphore 这种计数器结构，虽然有类似功能，但其实不存在真正意义的持有者，除非我们进行扩展包装。

CountDownLatch 和 CyclicBarrier的区别

* CountDownLatch 是不可以重置的，所以无法重用；而 CyclicBarrier 则没有这种限制，可以重用。
* CountDownLatch 的基本操作组合是 countDown/await。调用 await 的线程阻塞等待 countDown 足够的次数，不管你是在一个线程还是多个线程里 countDown，只要次数足够即可。所以就像 Brain Goetz 说过的，CountDownLatch 操作的是事件。
* CyclicBarrier 的基本操作组合，则就是 await，当所有的伙伴（parties）都调用了 await，才会继续进行任务，并自动进行重置。注意，正常情况下，CyclicBarrier 的重置都是自动发生的，如果我们调用 reset 方法，但还有线程在等待，就会导致等待线程被打扰，抛出 BrokenBarrierException 异常。
CyclicBarrier 侧重点是线程，而不是调用事件，它的典型应用场景是用来等待并发线程结束。

如果用 CountDownLatch 去实现上面的排队场景，该怎么做呢？假设有 10 个人排队，我们将其分成 5 个人一批，通过 CountDownLatch 来协调批次

```
import java.util.concurrent.CountDownLatch;
public class LatchSample {
  public static void main(String[] args) throws InterruptedException {
      CountDownLatch latch = new CountDownLatch(6);
          for (int i = 0; i < 5; i++) {
              Thread t = new Thread(new FirstBatchWorker(latch));
              t.start();
```

```
        }
        for (int i = 0; i < 5; i++) {
                Thread t = new Thread(new SecondBatchWorker(latch));
                t.start();
        }
             // 注意这里也是演示目的的逻辑，并不是推荐的协调方式
        while ( latch.getCount() != 1 ){
                Thread.sleep(100L);
        }
        System.out.println(”Wait for first batch finish”);
        latch.countDown();
    }
}
class FirstBatchWorker implements Runnable {
    private CountDownLatch latch;
    public FirstBatchWorker(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
            System.out.println(”First batch executed!”);
            latch.countDown();
    }
}
class SecondBatchWorker implements Runnable {
    private CountDownLatch latch;
    public SecondBatchWorker(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            latch.await();
            System.out.println(”Second batch executed!”);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```



```

在实际应用中的条件依赖，往往没有这么别扭，CountDownLatch 用于线程间等待操作结束是非常简单普遍的用法。通过 countDown/await 组合进行通信是很高效的，通常不建议使用例子里那个循环等待方式。

如果用 CyclicBarrier 来表达这个场景呢？我们知道 CyclicBarrier 其实反映的是线程并行运行时的协调，在下面的示例里，从逻辑上，5 个工作线程其实更像是代表了 5 个可以就绪的空车，而不再是 5 个乘客，对比前面 CountDownLatch 的例子更有助于我们区别它们的抽象模型，请看下面的示例代码：

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierSample {

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(5, () ->
System.out.println("Action...GO again!"));
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new CyclicWorker(barrier));
            t.start();
        }
    }

    static class CyclicWorker implements Runnable {
        private CyclicBarrier barrier;
        public CyclicWorker(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for (int i=0; i<3 ; i++) {
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierSample {

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(5, () ->
System.out.println("Action...GO again!"));
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new CyclicWorker(barrier));
            t.start();
        }
    }

    static class CyclicWorker implements Runnable {
        private CyclicBarrier barrier;
        public CyclicWorker(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for (int i=0; i<3 ; i++) {
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierSample {

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(5, () ->
```

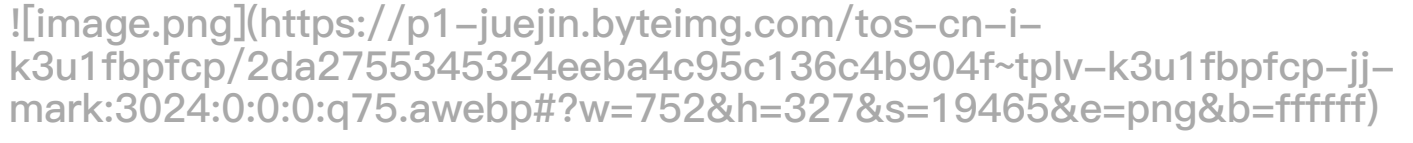
```
System.out.println("Action...GO again!"));
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new CyclicWorker(barrier));
            t.start();
        }
    }

    static class CyclicWorker implements Runnable {
        private CyclicBarrier barrier;
        public CyclicWorker(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for (int i=0; i<3 ; i++) {
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

为了让输出更能表达运行时序，我使用了 CyclicBarrier 特有的 barrierAction，当屏障被触发时，Java 会自动调度该动作。因为 CyclicBarrier 会自动进行重置，所以这个逻辑其实可以非常自然的支持更多排队人数。其编译输出如下：

并发包里提供的线程安全 Map、List 和 Set



如果我们的应用侧重于 Map 放入或者获取的速度，而不在乎顺序，大多推荐使用 ConcurrentHashMap，反之则使用 ConcurrentSkipListMap；如果我们需要对大量数据进行非常频繁地修改，ConcurrentSkipListMap 也可能表现出优势

原文链接: https://juejin.cn/post/7379786670252802059