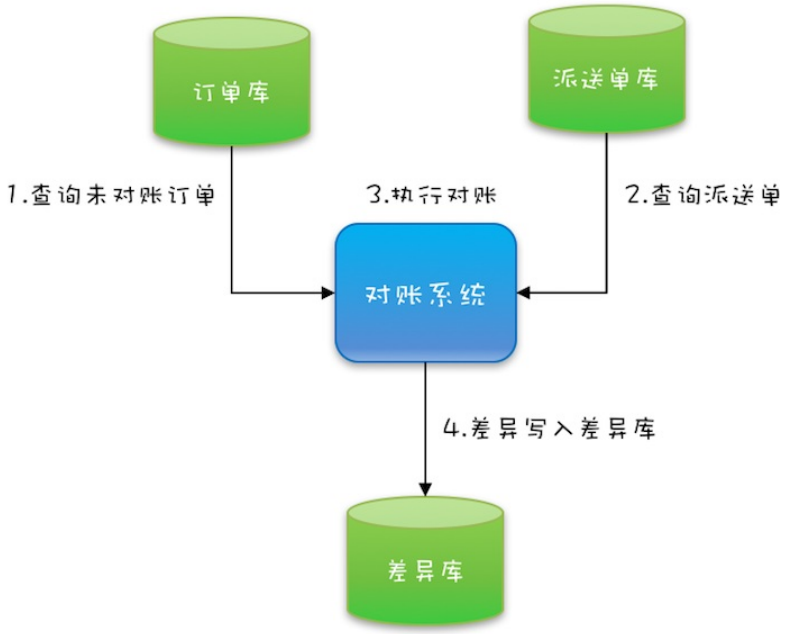
19-CountDownLatch和CyclicBarrier：如何让多线程步调一致？

前几天老板突然匆匆忙忙过来，说对账系统最近越来越慢了，能不能快速优化一下。我了解了对账系统的业务后，发现还是挺简单的，用户通过在线商城下单，会生成电子订单，保存在订单库；之后物流会生成派送单给用户发货，派送单保存在派送单库。为了防止漏派送或者重复派送，对账系统每天还会校验是否存在异常订单。

对账系统的处理逻辑很简单，你可以参考下面的对账系统流程图。目前对账系统的处理逻辑是首先查询订单，然后查询派送单，之后对比订单和派送单，将差异写入差异库。



对账系统流程图

对账系统的代码抽象之后，也很简单，核心代码如下，就是在一个单线程里面循环查询订单、派送单，然后执行对账，最后将写入差异库。

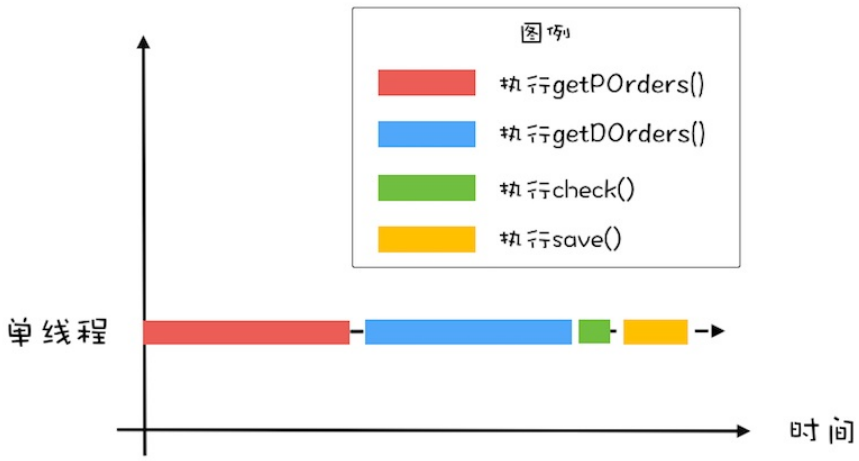
```
while(存在未对账订单){
  // 查询未对账订单
  pos = getPOrders();
  // 查询派送单
  dos = getDOrders();
  // 执行对账操作
  diff = check(pos, dos);
  // 差异写入差异库
  save(diff);
}
```

利用并行优化对账系统

老板要我优化性能，那我就首先要找到这个对账系统的瓶颈所在。

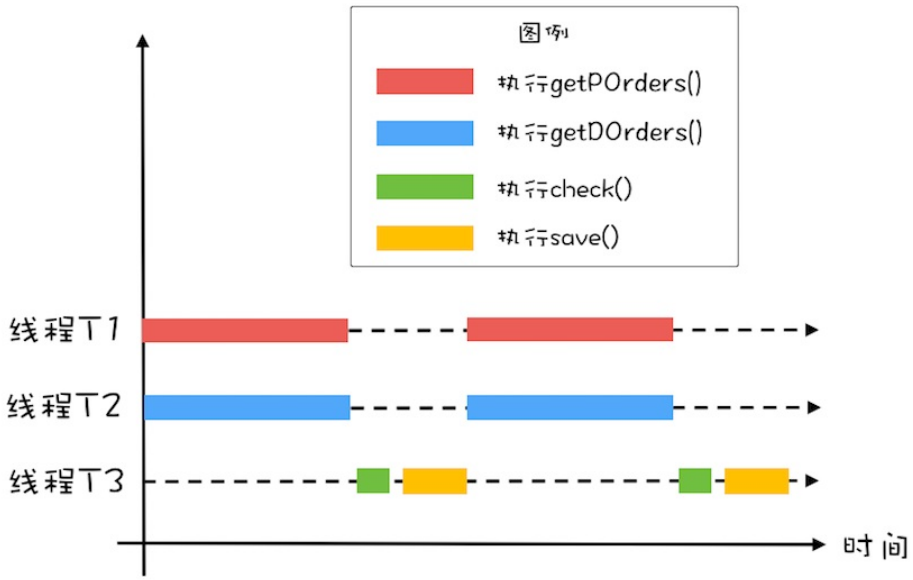
目前的对账系统，由于订单量和派送单量巨大，所以查询未对账订单getPOrders()和查询派送单

getDOrders()相对较慢，那有没有办法快速优化一下呢？目前对账系统是单线程执行的，图形化后是下图这个样子。对于串行化的系统，优化性能首先想到的是能否**利用多线程并行处理**。



对账系统单线程执行示意图

所以，这里你应该能够看出来这个对账系统里的瓶颈：查询未对账订单getPOrders()和查询派送单getDOrders()是否可以并行处理呢？显然是可以的，因为这两个操作并没有先后顺序的依赖。这两个最耗时的操作并行之后，执行过程如下图所示。对比一下单线程的执行示意图，你会发现同等时间里，并行执行的吞吐量近乎单线程的2倍，优化效果还是相对明显的。



对账系统并行执行示意图

思路有了，下面我们再来看看如何用代码实现。在下面的代码中，我们创建了两个线程T1和T2，并行执行查询未对账订单getPOrders()和查询派送单getDOrders()这两个操作。在主线程中执行对账操作check()和差异写入save()两个操作。不过需要注意的是：主线程需要等待线程T1和T2执行完才能执行check()和save()这两个操作，为此我们通过调用T1.join()和T2.join()来实现等待，当T1和T2线程退出时，调用T1.join()和T2.join()的主线程就会从阻塞态被唤醒，从而执行之后的check()和save()。

```
while(存在未对账订单){
  // 查询未对账订单
  Thread T1 = new Thread(()->{
    pos = getPOrders();
  });
  T1.start();
  // 查询派送单
  Thread T2 = new Thread(()->{
    dos = getDOrders();
  });
  T2.start();
  // 等待T1、T2结束
  T1.join();
  T2.join();
  // 执行对账操作
  diff = check(pos, dos);
  // 差异写入差异库
  save(diff);
}
```

用CountDownLatch实现线程等待

经过上面的优化之后，基本上可以跟老板汇报收工了，但还是有点美中不足，相信你也发现了，while循环里面每次都会创建新的线程，而创建线程可是个耗时的操作。所以最好是创建出来的线程能够循环利用，估计这时你已经想到线程池了，是的，线程池就能解决这个问题。

而下面的代码就是用线程池优化后的：我们首先创建了一个固定大小为2的线程池，之后在while循环里重复利用。一切看上去都很顺利，但是有个问题好像无解了，那就是主线程如何知道getPOrders()和getDOrders()这两个操作什么时候执行完。前面主线程通过调用线程T1和T2的join()方法来等待线程T1和T2退出，但是在线程池的方案里，线程根本就不会退出，所以join()方法已经失效了。

```
// 创建2个线程的线程池
Executor executor =
  Executors.newFixedThreadPool(2);
while(存在未对账订单){
  // 查询未对账订单
  executor.execute(()-> {
    pos = getPOrders();
  });
  // 查询派送单
  executor.execute(()-> {
    dos = getDOrders();
  });

  /* ？？如何实现等待？？*/

  // 执行对账操作
  diff = check(pos, dos);
  // 差异写入差异库
  save(diff);
}
```

那如何解决这个问题呢？你可以开动脑筋想出很多办法，最直接的办法是弄一个计数器，初始值设置成2，

当执行完pos = getPOrders();这个操作之后将计数器减1，执行完dos = getDOrders();之后也将计数器减1，在主线程里，等待计数器等于0；当计数器等于0时，说明这两个查询操作执行完了。等待计数器等于0其实就是一个条件变量，用管程实现起来也很简单。

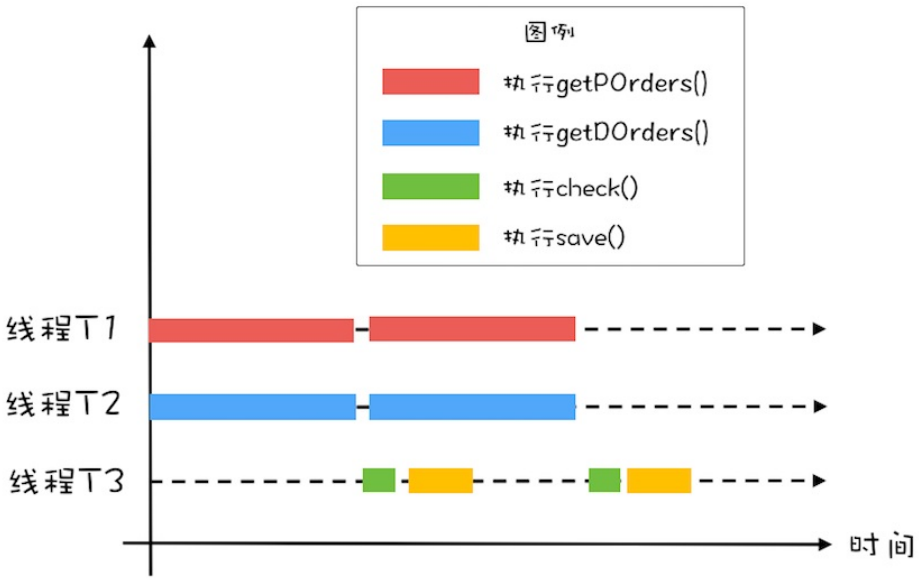
不过我并不建议你在实际项目中去实现上面的方案，因为Java并发包里已经提供了实现类似功能的工具类：**CountDownLatch**，我们直接使用就可以了。下面的代码示例中，在while循环里面，我们首先创建了一个CountDownLatch，计数器的初始值等于2，之后在pos = getPOrders();和dos = getDOrders();两条语句的后面对计数器执行减1操作，这个对计数器减1的操作是通过调用 latch.countDown(); 来实现的。在主线程中，我们通过调用 latch.await() 来实现对计数器等于0的等待。

```
// 创建2个线程的线程池
Executor executor =
  Executors.newFixedThreadPool(2);
while(存在未对账订单){
  // 计数器初始化为2
  CountDownLatch latch =
    new CountDownLatch(2);
  // 查询未对账订单
  executor.execute(()-> {
    pos = getPOrders();
    latch.countDown();
  });
  // 查询派送单
  executor.execute(()-> {
    dos = getDOrders();
    latch.countDown();
  });

  // 等待两个查询操作结束
  latch.await();

  // 执行对账操作
  diff = check(pos, dos);
  // 差异写入差异库
  save(diff);
}
```

进一步优化性能

经过上面的重重优化之后，长出一口气，终于可以交付了。不过在交付之前还需要再次审视一番，看看还有没有优化的余地，仔细看还是有的。

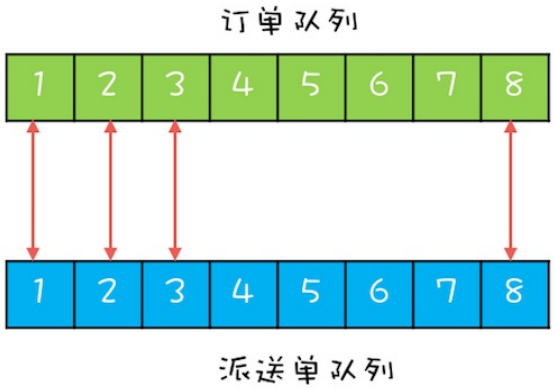
前面我们将getPOrders()和getDOrders()这两个查询操作并行了，但这两个查询操作和对账操作check()、save()之间还是串行的。很显然，这两个查询操作和对账操作也是可以并行的，也就是说，在执行对账操作的时候，可以同时去执行下一轮的查询操作，这个过程可以形象化地表述为下面这幅示意图。



完全并行执行示意图

那接下来我们再来思考一下如何实现这步优化，两次查询操作能够和对账操作并行，对账操作还依赖查询操作的结果，这明显有点生产者-消费者的意思，两次查询操作是生产者，对账操作是消费者。既然是生产者-消费者模型，那就需要有个队列，来保存生产者生产的数据，而消费者则从这个队列消费数据。

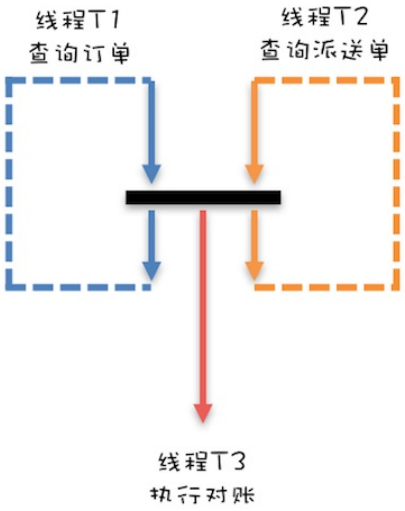
不过针对对账这个项目，我设计了两个队列，并且两个队列的元素之间还有对应关系。具体如下图所示，订单查询操作将订单查询结果插入订单队列，派送单查询操作将派送单插入派送单队列，这两个队列的元素之间是有一一对应的关系的。两个队列的好处是，对账操作可以每次从订单队列出一个元素，从派送单队列出一个元素，然后对这两个元素执行对账操作，这样数据一定不会乱掉。



双队列示意图

下面再来看如何用双队列来实现完全的并行。一个最直接的想法是：一个线程T1执行订单的查询工作，一个线程T2执行派送单的查询工作，当线程T1和T2都各自生产完1条数据的时候，通知线程T3执行对账操作。这个想法虽看上去简单，但其实还隐藏着一个条件，那就是线程T1和线程T2的工作要步调一致，不能一个跑得太快，一个跑得太慢，只有这样才能做到各自生产完1条数据的时候，通知线程T3。

下面这幅图形象地描述了上面的意图：线程T1和线程T2只有都生产完1条数据的时候，才能一起向下执行，也就是说，线程T1和线程T2要互相等待，步调要一致；同时当线程T1和T2都生产完一条数据的时候，还要能够通知线程T3执行对账操作。



同步执行示意图

用CyclicBarrier实现线程同步

下面我们就来实现上面提到的方案。这个方案的难点有两个：一个是线程T1和T2要做到步调一致，另一个是要能够通知到线程T3。

你依然可以利用一个计数器来解决这两个难点，计数器初始化为2，线程T1和T2生产完一条数据都将计数器减1，如果计数器大于0则线程T1或者T2等待。如果计数器等于0，则通知线程T3，并唤醒等待的线程T1或者T2，与此同时，将计数器重置为2，这样线程T1和线程T2生产下一条数据的时候就可以继续使用这个计数器了。

同样，还是建议你不要在实际项目中这么做，因为Java并发包里也已经提供了相关的工具类：**CyclicBarrier**。在下面的代码中，我们首先创建了一个计数器初始值为2的CyclicBarrier，你需要注意的是创建CyclicBarrier的时候，我们还传入了一个回调函数，当计数器减到0的时候，会调用这个回调函数。

线程T1负责查询订单，当查出一条时，调用 `barrier.await()` 来将计数器减1，同时等待计数器变成0；线程T2负责查询派送单，当查出一条时，也调用 `barrier.await()` 来将计数器减1，同时等待计数器变成0；当T1和T2都调用 `barrier.await()` 的时候，计数器会减到0，此时T1和T2就可以执行下一条语句了，同时会调用barrier的回调函数来执行对账操作。

非常值得一提的是，CyclicBarrier的计数器有自动重置的功能，当减到0的时候，会自动重置你设置的初始值。这个功能用起来实在是太方便了。

```
// 订单队列
Vector<P> pos;
// 派送单队列
Vector<D> dos;
```

```
// 执行回调的线程池
Executor executor =
  Executors.newFixedThreadPool(1);
final CyclicBarrier barrier =
  new CyclicBarrier(2, ()->{
    executor.execute(()->check());
  });

void check(){
  P p = pos.remove(0);
  D d = dos.remove(0);
  // 执行对账操作
  diff = check(p, d);
  // 差异写入差异库
  save(diff);
}

void checkAll(){
  // 循环查询订单库
  Thread T1 = new Thread(()->{
    while(存在未对账订单){
      // 查询订单库
      pos.add(getPOrders());
      // 等待
      barrier.await();
    }
  });
  T1.start();
  // 循环查询运单库
  Thread T2 = new Thread(()->{
    while(存在未对账订单){
      // 查询运单库
      dos.add(getDOrders());
      // 等待
      barrier.await();
    }
  });
  T2.start();
}
```

总结

CountDownLatch和CyclicBarrier是Java并发包提供的两个非常易用的线程同步工具类，这两个工具类用法的区别在这里还是有必要再强调一下：**CountDownLatch主要用来解决一个线程等待多个线程的场景**，可以类比旅游团团长要等待所有的游客到齐才能去下一个景点；而**CyclicBarrier是一组线程之间互相等待**，更像是几个驴友之间不离不弃。除此之外CountDownLatch的计数器是不能循环利用的，也就是说一旦计数器减到0，再有线程调用await()，该线程会直接通过。但**CyclicBarrier的计数器是可以循环利用的**，而且具备自动重置的功能，一旦计数器减到0会自动重置到你设置的初始值。除此之外，CyclicBarrier还可以设置回调函数，可以说是功能丰富。

本章的示例代码中有两处用到了线程池，你现在只需要大概了解即可，因为线程池相关的知识咱们专栏后面还会有详细介绍。另外，线程池提供了Future特性，我们也可以利用Future特性来实现线程之间的等待，这个后面我们也会详细介绍。

课后思考

本章最后的示例代码中，CyclicBarrier的回调函数我们使用了一个固定大小的线程池，你觉得是否有必要

呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

猜你喜欢

精选留言：

- 探索无止境 2019-04-11 07:46:14
  今天的文章很精彩，有案例有递进，一气呵成！设置线程池为单个线程可以保证对账的操作按顺序执行 [25赞]

- undifined 2019-04-11 08:40:30
  线程池大小为1是必要的，如果设置为多个，有可能会两个线程 A 和 B 同时查询，A 的订单先返回，B 的派送单先返回，造成队列中的数据不匹配；所以1个线程实现生产数据串行执行，保证数据安全

  如果用Future 的话可以更方便一些：

  CompletableFuture<List> pOrderFuture = CompletableFuture.supplyAsync(this::getPOrders);
  CompletableFuture<List> dOrderFuture = CompletableFuture.supplyAsync(this::getDOrders);
  pOrderFuture.thenCombine(dOrderFuture, this::check)
  .thenAccept(this::save);

  老师这样理解对吗，谢谢老师
  [10赞]

  作者回复2019-04-11 09:15:42
  对，👍👍👍

- 西西弗与卡夫卡 2019-04-11 01:34:36
  回调中的线程池用单线程是为了确保从两个队列取数时可以一对一获取，避免错乱。比如说，如果有两个线程，则可能出现线程1获取PO1，线程获取PO2和DO1，线程获取DO2的乱序。

  其实线程池改成多线程也可以，要把两个remove(0)放到一个同步块中 [7赞]

- 波波 2019-04-11 09:30:04
  思考题中，如果生产者比较快，消费者比较慢，生产者通知的时候，消费者还在对账，这个时候会怎么处理？会不会导致消费者错失通知，导致队列满了，但是消费者却没有收到通知。 [5赞]

作者回复2019-04-12 19:17:45
有这种可能，还能oom

- nanquanmama 2019-04-11 08:32:55
最后的那个例子，业务逻辑的部分已经变得很不直观，并发控制的逻辑掩盖住了业务逻辑。请问一下老师，实际项目开发中，并发控制逻辑如何做，才能和业务逻辑分离出来？ [4赞]

作者回复2019-04-11 09:22:52
放到不同的类里，这方面传统的面向对象可以解决，lambda也能解决，这个模块的最后几章能解决你说的这个问题，但是更复杂的场景还得自己设计

- Darren 2019-04-11 00:35:42
而且其实可以直接single线程池的，但是最好不要Executors提供的线程池，都有弊端，最好自定义线程池 [3赞]

作者回复2019-04-12 19:21:03
👍

- Darren 2019-04-11 00:29:44
有一定的必要，因为CyclicBarrier是可以重制的，所以如果不用线程池的话，每次都需要新建线程，浪费资源。也不知道对不对，请大佬指点 [3赞]

- 2019-04-12 17:30:18
追问：如果线程池是单线程的话。那假如生产者速度快运check函数执行时间。那是不是就会出现堵塞情况了。久而久之，是不是会出现队列内存溢出 [2赞]

作者回复2019-04-12 18:36:43
会

- 西兹兹 2019-04-11 22:53:02
undefind同学的意思差不多对。 只有一个线程的线程池，是因为，订单队列和派单队列读取数据存在竞态条件。 如果要开多个线程，则需要一个lock进行同步那两个remove方法。 个人推荐的思路是，如果生产者速度比消费者快的情况下，放入一个双向的阻塞队列尾部，每次从双向队列头部取两个对象，根据对象属性来区别订单类型，也能开多个线程进行check操作。 但本文业务里check速度很快，所以这个场景只需要开1个线程的线程池是合理的。 [2赞]

作者回复2019-04-12 00:09:46
一次多取几个然后批量执行，这个办法非常实用！

- 木偶人King 2019-04-11 17:31:27
老师，最后checkAll（ ）这里为什么new 了两个Thread 而不是使用线程池

[1赞]

作者回复2019-04-11 19:57:41
反正也不会反复创建，用不用都没关系

- 空知 2019-04-11 14:55:47
老师,关于CyclicBarrier回调函数,请教下
自己写了个 CyclicBarrier的例子,回调函数总是在计数器归0时候执行,但是线程T1 T2要等回调函数执行结

束之后才会再次执行...看了下CyclicBarrier 的源码,当内部计数器 index == 0时候,

final Runnable command = barrierCommand;

if (command != null)

command.run();
没有开启子线程吧.也就是说 对账还是同步执行的,结束之后才是下一次的查询 [1赞]

作者回复2019-04-11 20:07:39
所以才需要线程池来异步执行回调函数，你一不小心把答案找到了😁

- 维斯有条河 2019-04-11 13:24:39
感谢老师，一直不太明白什么时候用CyclicBarrier，今天看到案例了，刚看到join那段我想到了CompletableFuture

[1赞]

作者回复2019-04-11 20:08:59
👍

- iron_man 2019-04-11 11:39:24
王老师，cyclicbarrier，具体是在什么时候清零计数器呢？是在所有线程await返回后还是在回调函数调用后？await和回掉函数的调用顺序是怎样的 [1赞]

作者回复2019-04-14 19:34:35
回调函数执行完之后才会唤醒等待的线程。

- magict4 2019-04-11 05:51:29
> 另外，线程池提供了 Future 特性，我们也可以利用 Fu...

挺期待老师可以讲讲什么时候用 Future.get()，什么时候用 CountDownLatch 来等待线程完成。 [1赞]

- 碳水化合物 2019-04-17 02:28:04
如果最后的线程池有两个线程，会有什么问题？老师能举个例子吗？想了好久没想出来啊，有点折磨人

- Zach_ 2019-04-15 09:16:37
日常打卡，期待老师后续更精彩的讲解！

- 小美 2019-04-14 23:11:11
为什么cycliyBarrier 中不用大小为3的线程池

作者回复2019-04-15 10:31:14
那我得先讲线程池

- 姜小白 2019-04-14 20:37:09
老师，你的最后一个例子代码中，T1.start()之前，对于T1的定义，是不是少了一个"
)"，还有T2也是。

作者回复2019-04-14 21:16:11
火焰金精！！！多谢多谢！

- 橘子不酸 2019-04-14 11:43:05
  老师，barrier的 barrierAction是不是最后一个线程进入后就会执行，如果这个时候之前进入的线程还没有执行完的话，在这个应用场景下，对账是不是有问题。

  作者回复2019-04-14 19:19:41
  barrierAction是同步执行，只有执行完才会开启下一轮。 所以用一个单线程的线程池，既能解决同步的性能问题，也能解决数据的一致性问题。

- 曾轼麟 2019-04-13 21:52:19
  老师推荐您使用ThreadPoolExecutor去实现线程池，并且实现里面的RejectedExecutionHandler和ThreadFactory，这样可以方便当调用订单查询和派送单查询的时候出现full gc的时候 dump文件 可以快速定位出现问题的线程是哪个业务线程，如果是CountDownLatch，建议设置超时时间，避免由于业务死锁没有调用countDown()导致现线程睡死的情况

  作者回复2019-04-13 23:32:06
  好建议，所有的阻塞操作，都需要设置超时时间，这是个很好的习惯。