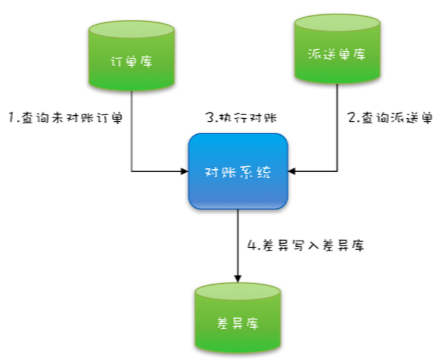
**19、CountDownLatch 和CyclicBarrier：如何让多线程步调一致？**[](" \l "19countdownlatch-cyclicbarrier" \o "Permanent link)

* 利用并行化优化对账系统
* 用CountDownLatch实现线程等待
* 进一步优化性能
* 用CyclicBarrier实现线程同步
* 总结

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*

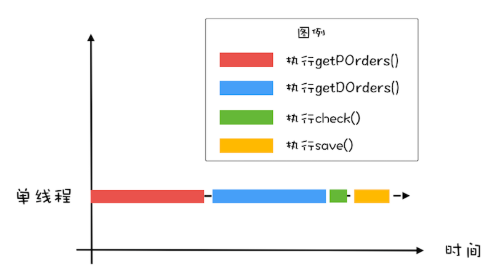
save**(**diff**);**

**}**

**利用并行优化对账系统**[](" \l "_1" \o "Permanent link)

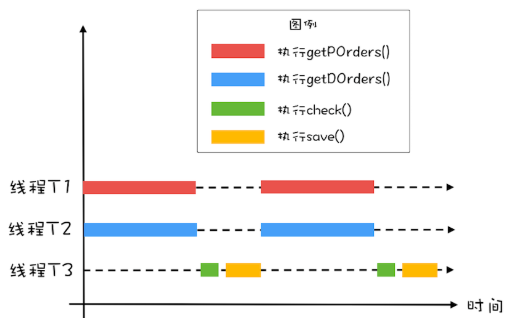
老板要我优化性能，那我就首先要找到这个对账系统的瓶颈所在。

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



对于串行化系统，优化性能首先想到的是能否利用多线程并行处理。

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。



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 **=** getDOrders**();**

**});**

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**[](" \l "countdownlatch" \o "Permanent link)

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()方法等待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 **=** getDOrders**();**

**});**

*//查询派送单*

executor**.**executor**(()->{**

dos **=** getDOrders**();**

**});**

*/\* ? 如何实现等待？？\*/*

*//执行对账操作*

diff **=** check**(**pos**,**dos**);**

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了减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。

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 **=** getDOrders**();**

latch**.**countDown**();**

**});**

*//查询派送单*

executor**.**executor**(()->{**

dos **=** getDOrders**();**

latch**.**countDown**();**

**});**

*/\* ? 如何实现等待？？\*/*

*//等待两个查询操作结束*

latch**.**await**()**

*//执行对账操作*

diff **=** check**(**pos**,**dos**);**

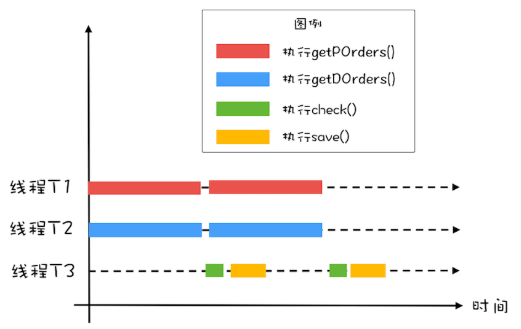
save**(**diff**)**；

**}**

**进一步优化性能**[](" \l "_2" \o "Permanent link)

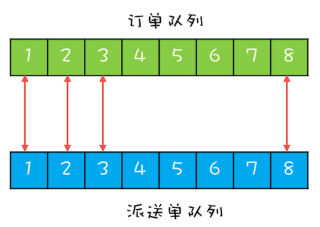
经过上面的重重优化之后，长出一口气，终于可以交付了。不过在交付之前还需要再次审视一番，看看还有没有优化的余地，仔细看还是有的。

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。



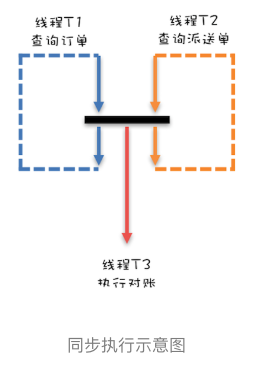
那接下来我们再来思考一下如何实现这步优化，两次查询操作能够和对账操作并行，对账操作还依赖查询操作的结果，这明显有点生产者 - 消费者的意思，两次查询操作是生产者，对账操作是消费者。既然是生产者 - 消费者模型，那就需要有个队列，来保存生产者生产的数据，而消费者则从这个队列消费数据。

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。



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。



**用CyclicBarrier 实现线程同步**[](" \l "cyclicbarrier" \o "Permanent link)

初步的想法是：

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。

*//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**);**

save**(**diff**);**

**}**

**void** **checkAll(){**

*//循环查询订单裤*

Thread t1 **=** **new** Thread**(()->{**

**while(**存在未对账的订单**){**

pos**.**add**(**getPorder**());**

*//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）**{**

dos**.**add**(**getDOrders**());**

*//等待*

barrier**.**await**();**

**}**

**});**

T2**.**start**();**

**}**

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0

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。

自己实现的 Demo

**public** **class** **CycTest** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

Vector**<**Integer**>** pos **=** **new** Vector**<>();**

Vector**<**Integer**>** dos **=** **new** Vector**<>();**

CyclicBarrier barrier **=** **new** CyclicBarrier**(**2**,()->{**

System**.**out**.**println**(**"执行回调"**+**System**.**currentTimeMillis**());**

**try** **{**

Thread**.**sleep**(**500**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**});**

**int** i**=** 0**;**

**int** j **=** 0**;**

Thread T1 **=** **new** Thread**(()->{**

**while(true){**

*// 查询订单库*

pos**.**add**(**i**+**1**);**

System**.**out**.**println**(**"dos add i:"**+**System**.**currentTimeMillis**());**

**try** **{**

Thread**.**sleep**(**1000**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

*// 等待*

**try** **{**

barrier**.**await**();**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **catch** **(**BrokenBarrierException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**});**

T1**.**start**();**

Thread T2**=** **new** Thread**(()->{**

**while(true){**

*// 查询订单库*

dos**.**add**(**j**+**1**);**

System**.**out**.**println**(**"dos add j:"**+**System**.**currentTimeMillis**());**

**try** **{**

Thread**.**sleep**(**1000**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

*// 等待*

**try** **{**

barrier**.**await**();**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **catch** **(**BrokenBarrierException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**});**

T2**.**start**();**

**}**

**}**

**总结**[](" \l "_3" \o "Permanent link)

CountDownLatch 和CyclicBarrier是java并发包提供的两个非常易用的同步工具类。

**这两个工具类的区别：**[](" \l "_4" \o "Permanent link)

* **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
* **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。