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

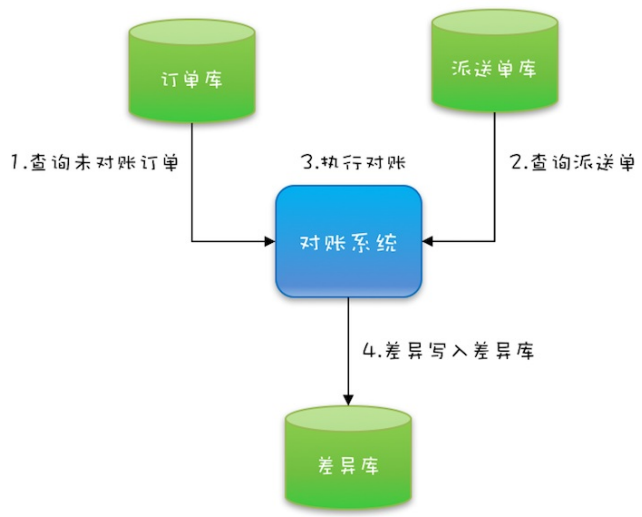
王宝令 2019-04-11

00:00 10:00

讲述：王宝令　　大小：9.17M

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

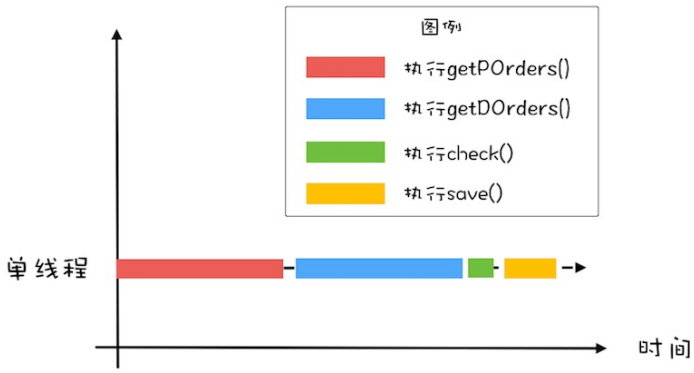
对账系统的代码抽象之后，也很简单，核心代码如下，就是在一个单线程里面循环查询订单、派送单，然后执行对账，最后将写入差异库。

```
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

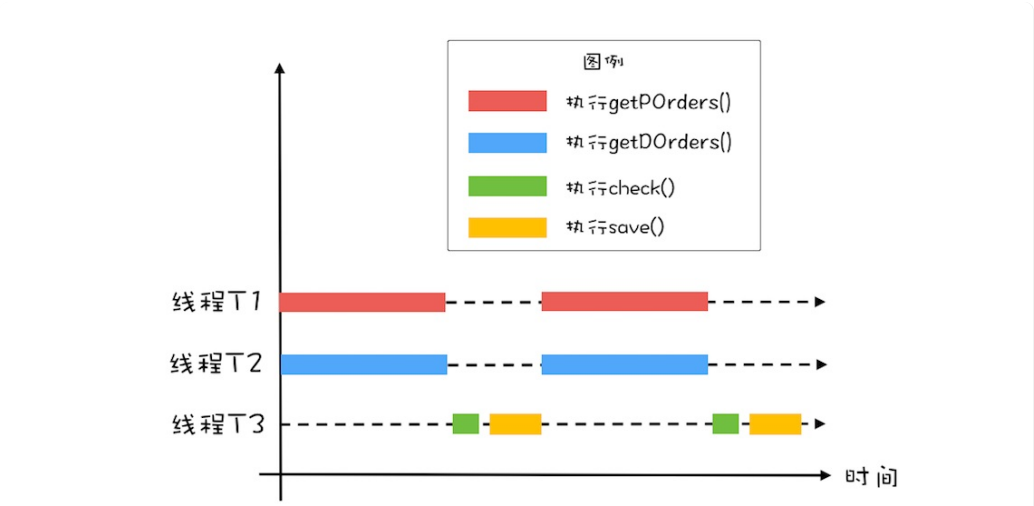
老板要我优化性能，那我就首先要找到这个对账系统的瓶颈所在。

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()。

复制代码

```
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。

复制代码

```
// 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。

复制代码

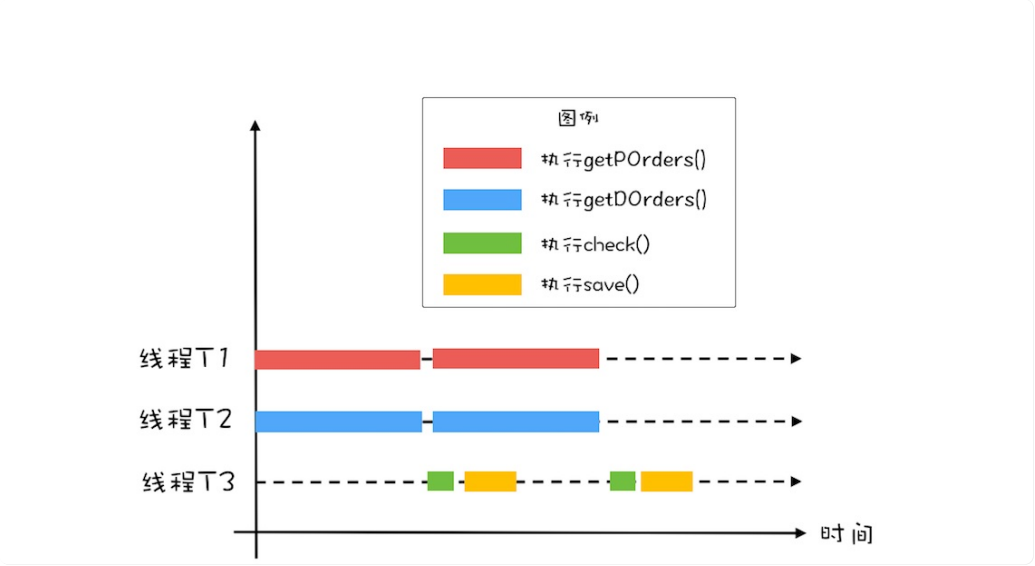
```
// 创建 2 个线程的线程池
Executor executor =
  Executors.newFixedThreadPool(2);
while(存在未对账订单){
  // 计数器初始化为 2
  CountDownLatch latch =
    new CountDownLatch(2);
  // 查询未对账订单
  executor.execute(()-> {
    pos = getPOrders();
    latch.countDown();
  });
  // 查询派送单
  executor.execute(()-> {
```

```
    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。

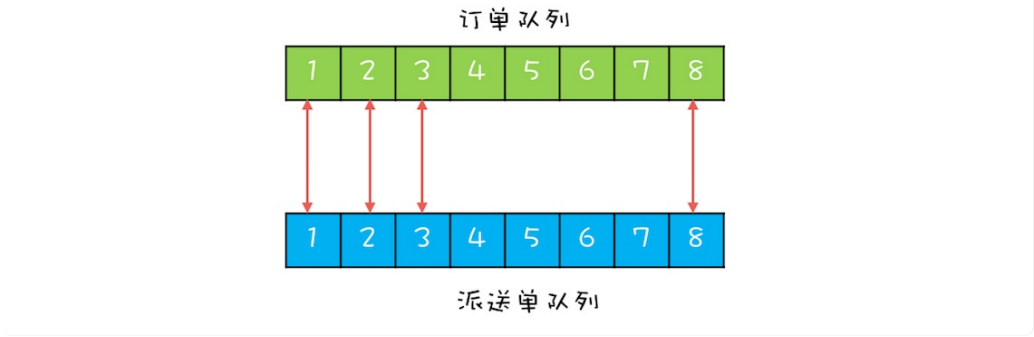
前面我们将 getPOrders() 和 getDOrders() 这两个查询操作并行了，但这两个查询操作和对账操作 check()、save() 之间还是串行的。很显然，这两个查询操作和对账操作也是可以并行的，也就是说，在执行对账操作的时候，可以同时去执行下一轮的查询操作，这个过程可以形象化地表述为下面这幅示意图。



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。

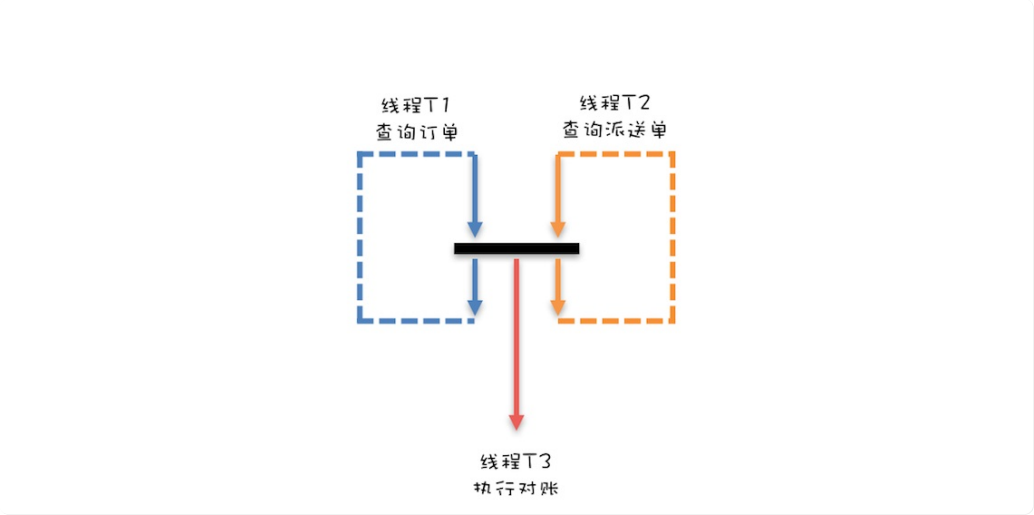
不过针对对账这个项目，我设计了两个队列，并且两个队列的元素之间还有对应关系。具体如下图所示，订单查询操作将订单查询结果插入订单队列，派送单查询操作将派送单插入派送单队列，这两个队列的元素之间是有一一对应的关系的。两个队列的好处是，对账操作可以每次从订单队列出一个元素，从派送单队列出一个元素，然后对这两个元素执行对账操作，这样数据一定不会乱掉。



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。

复制代码

```
// 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(25)

探索无止境

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

4　2019-04-11

西西弗与卡夫卡

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

3 2019-04-11

nanquanmama

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？

1 2019-04-11

> 作者回复: 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

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

1 2019-04-11

波波

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。

2019-04-11

IT小白

这个示例确实将这两个工具类的使用讲清楚了，但是对于这个具体业务这么改造是不合理的。因为整个对账过程里的所有操作必须是原子的，这么拆分到多个线程里执行，这个原子性的保证相对来说比较复杂，示例中也没有考虑。我觉得这里的性能优化可以很简单，比如：搞一组线程，将所有订单平均分摊到这一组线程上，所有线程并行对账，每个线程做的活都一样～ 注意每个线程维护独立的数据库连接～

2019-04-11

Lemon

今天的分享有意思

有意义，避免重复创建线程

2019-04-11

甄的猛士

今天课程思路清晰，容易理解接受。值得学习这个课程思路

2019-04-11

朱晋君

赞成西西弗与卡·的观点

2019-04-11

周治慧

订单队列和派送队列在比对数据的过程中可能是134 1234这样就不是remove（0）的操作了。还有个问

题请教一下老师，当每天需要对账的订单和派送队列数据都较大的时候，怎么合适的分批量取出id都在同一个区间的数据去对账。

2019-04-11

张建磊

王老师好，今天分享有实际案例和解决方法流程，期待以后分享都可以解决案例方式。
对于题中的线程池1还是有必要。如果是多个线程池，可能会出现出队的乱序。能解决出队乱序，就得阻塞部分线程，效率应该还是在伯仲之间。

2019-04-11

Hou

会出现计数器被同一个线程减到0的情况吗？

2019-04-11

WL

想问一下老师如果在CyclicBarrier的例子中, checkAll()方法抛出异常了怎么办?

2019-04-11

> 作者回复: 找bug,重新来过😁

刘章周

有必要，避免消费者线程多次创建，消耗资源。

2019-04-11

undifined

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

2019-04-11

> 作者回复: 对，👍👍👍

张德

感觉今天的这个场景真棒 感觉又学到了好多 每次我都是听三遍左右才能理解😊

2019-04-11

WL

我感觉用CyclicBarrier用一个固定大小的线程池好像没啥用, 直接主线程执行一下不就可以了, 因为从订单表和运单表中取出的数据的操作已经用单独线程实现了.

2019-04-11

木木匠

有必要限制只有一个线程的线程池，主要是为了回调函数的线程安全，因为对于check 方法本身来说并不是线程安全的，虽然有vector 来保证队列中的操作是线程安全的，但是有可能造成两个队列取出数据不同步的情况，如果用了单线程串行可以避免这个问题。

2019-04-11

心中无剑

我在想，回调采用固定线程池，并且设置核心数是1。原因也许跟 save(diff) 有关，为了保证数据一致性。save操作失败后，事务回滚吗？ 一条线程一个事务，不然save这一步如果发生异常，怎么保证原子性呢

2019-04-11

邋遢的流浪剑客

使用了固定大小为1的线程池，check方法是非线程安全的，让它串行执行

2019-04-11

CRT

关于思考题。由上图看到check操作耗时比较少的，所以理论上单线程可以完成任务，不会造成队列阻塞，设置线程池为1即可，同时也防止出现多线程调用check。

2019-04-11

杨春鹏

为什么最后一种情况的线程池的线程数量是1呢？

2019-04-11

张三

CyclicBarrier那段代码只要调用一次checkAll（）方法就会一直执行了吧？里边的每次循环都会在计数器等于0的时候自动回调check（）方法对账，然后两个线程分别进行下一次循环。

2019-04-11

> 作者回复: 是的

magict4

> 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。

2019-04-11

Darren

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

2019-04-11