Pipeline分析

背景

对于一个任务，如果它的流程能够被清晰的分成几个部分，我们一般都会采用pipeline的模型对其进行加速。

例如我们IoTDB里的flush流程，分为sort、encoding以及io这三个部分。

每个部分之间相当于生产者与消费者之间的关系，encoding消费sort生产出的数据，io消费encoding生产出的数据。

而在使用这种模型进行加速时，我们需要特别注意在pipeline的不同阶段，不能调用同一对象的synchronized方法，即使不同阶段不会同时调用synchronized方法，即不会产生锁冲突，依旧会对运行流程产生影响。

问题

为了简化问题，我们假设pipeline只有两个阶段，阶段一负责生产出Message对象，放入队列中；阶段二负责从队列中取出Message对象进行消费。

阶段一的线程生成 Message，并且调用带 synchronized 的 set 方法 给 里边的 byte[] 的每一位赋值。最后放到队列里。

阶段二的线程从个队列中取出 Message，并调用 Message 的带和不带 synchronized 的 size() 方法进行打印。

注意，Message对象的set(int index, int add)方法是由synchronized修饰的，而获得size的方法有两个，一个是有synchronized修饰的synchronizedSize()方法，一个是无synchronized修饰的size()方法。

如果，阶段二中调用Message的synchronizedSize()方法，则阶段一和阶段二的耗时都会增加：

Task 1 cost: 4,587ms
Task 2 cost: 42ms

如果，阶段二中调用Message的size()方法，则阶段一和阶段二的耗时都会减少：

Task 1 cost: 2,678ms
Task 2 cost: 27ms

直觉上来讲，阶段一产生的对象A，阶段一生产出它后，就将它放入队列中，随后由阶段二的线程从队列中取出并消费，不论阶段二调用的是不是synchronized方法，不应该对阶段一产生任何影响，因为阶段一和阶段二不会有任何锁冲突，可是实验结果却是反直觉的。

IoTDB的flush encoding阶段和io阶段使用的PublicBAOS类，继承了ByteArrayOutputStream类，这是个线程安全的类，里面的很多write方法都是有synchronized修饰的，所以io阶段就会对encoding阶段产生了影响。

问题分析

要了解为什么会出现上述反直觉的问题，我们需要先了解java中的synchronized的实现机制，也就是偏向锁。

偏向锁（JDK15官方宣布废弃该特性）

偏向锁是jdk1.6以后， HotSpot 虚拟机使用的一项优化技术，能够减少无竞争锁定时的开销。偏向锁的目的是假定 monitor 一直由某个特定线程持有，直到另一个线程尝试获取它，这样就可以避免获取 monitor 时执行 cas 的原子操作。monitor 首次锁定时偏向该线程，这样就可以避免同一对象的后续同步操作步骤需要原子指令。

而如果后续有其他线程再次调用对象的synchronized方法，就会多出撤销偏向锁，将偏向锁升级成轻量级锁的开销（或者是偏向锁的批量再偏向（Bulk Rebias）机制引入的开销）。

具体细节可以参见这篇博客：https://blog.csdn.net/lengxiao1993/article/details/81568130

问题原因

所以，问题就可以得到解答了，阶段一调用对象A的synchronized方法时，已经将其的偏向锁指向自己；随后阶段二再次调用对象A的其他synchronized方法时，发现偏向锁指向的线程不是自己时，就会撤销偏向锁，或者反复进行偏向锁的批量再偏向。

问题验证

jvm提供了启动参数，可以关闭偏向锁这一特性，-XX:-UseBiasedLocking。我关闭偏向锁后，再做一次pipeline实验。

阶段二中调用Message的synchronizedSize()方法

Task 1 cost: 5709
Task 2 cost: 29

阶段二中调用Message的size()方法

Task 1 cost: 5814
Task 2 cost: 30

我们可以看到，关闭偏向锁后，不论阶段二调用的是不是synchronized修饰的方法，都不会对阶段一产生任何影响。

实验代码

```
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class Pipeline {

  private static final ExecutorService SUB_TASK_POOL_MANAGER =
Executors.newFixedThreadPool(4);
  private final Future<Long> task2Future;

  private final ConcurrentLinkedQueue<Message> task2Queue = new ConcurrentLinkedQueue<>();

  private volatile boolean noMoreTask2 = false;

  public Pipeline() {
    this.task2Future = SUB_TASK_POOL_MANAGER.submit(task2);
  }


  /**
   * Task 1
   */
  public void startPipeline() throws ExecutionException, InterruptedException {
    long task1Cost = 0;
    Random random = new Random();
    for (int i = 0; i < 100; i++) {
      long task1StartTime = System.currentTimeMillis();
      Message task = new Message();
      task.bytes = new byte[(int) ((random.nextFloat() + 0.5) * 1024 * 1024)];
      for (int j = 0; j < task.bytes.length; j++) {
        for (int k = 0; k < 2; k++) {
          task.set(j, random.nextInt(128));
        }
      }
      task1Cost += (System.currentTimeMillis() - task1StartTime);
      task2Queue.add(task);
    }
    noMoreTask2 = true;
    System.out.println("Task 1 cost: " + task1Cost);

    System.out.println("Task 2 cost: " + task2Future.get());

  }

  private final Callable<Long> task2 = () -> {
    long task2Cost = 0;
    boolean noMoreMessages = false;
    while (true) {
      if (noMoreTask2) {
        noMoreMessages = true;
      }
      Message task = task2Queue.poll();
      if (task == null) {
        if (noMoreMessages) {
          break;
        }
        try {
```

```
        TimeUnit.MILLISECONDS.sleep(10);
      } catch (@SuppressWarnings("squid:S2142") InterruptedException e) {
        break;
      }
    } else {
      long task2StartTime = System.currentTimeMillis();
      System.out.println("size: " + task.size());
      task2Cost += (System.currentTimeMillis() - task2StartTime);
    }
  }
  return task2Cost;
};


private static class Message {
  byte[] bytes;

  public synchronized void set(int index, int add) {
    bytes[index] += add;
  }

  public synchronized int synchronizedSize() {
    return bytes.length;
  }

  public int size() {
    return bytes.length;
  }
}


public static void main(String[] args) throws ExecutionException, InterruptedException {
  Pipeline pipeline = new Pipeline();
  pipeline.startPipeline();
}
}
```