

36 | 生产者-消费者模式：用流水线思想提高效率

2019-05-21 王宝令



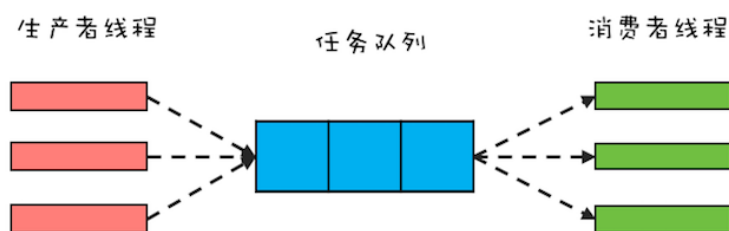
前面我们在 [《34 | Worker Thread模式：如何避免重复创建线程？》](#) 中讲到，Worker Thread模式类比的是工厂里车间工人的工作模式。但其实在现实世界，工厂里还有一种流水线的工作模式，类比到编程领域，就是**生产者-消费者模式**。

生产者-消费者模式在编程领域的应用也非常广泛，前面我们曾经提到，**Java**线程池本质上就是用生产者-消费者模式实现的，所以每当使用线程池的时候，其实就是在应用生产者-消费者模式。

当然，除了在线程池中的应用，为了提升性能，并发编程领域很多地方也都用到了生产者-消费者模式，例如**Log4j2**中异步**Appender**内部也用到了生产者-消费者模式。所以今天我们就来深入地聊聊生产者-消费者模式，看看它具体有哪些优点，以及如何提升系统的性能。

生产者-消费者模式的优点

生产者-消费者模式的核心是一个**任务队列**，生产者线程生产任务，并将任务添加到任务队列中，而消费者线程从任务队列中获取任务并执行。下面是生产者-消费者模式的一个示意图，你可以结合它来理解。



生产者-消费者模式示意图

从架构设计的角度来看，生产者-消费者模式有一个很重要的优点，就是**解耦**。解耦对于大型系统的设计非常重要，而解耦的一个关键就是组件之间的依赖关系和通信方式必须受限。在生产者-消费者模式中，生产者和消费者没有任何依赖关系，它们彼此之间的通信只能通过任务队列，所以生产者-消费者模式是一个不错的解耦方案。

除了架构设计上的优点之外，生产者-消费者模式还有一个重要的优点就是**支持异步，并且能够平衡生产者和消费者的速度差异**。在生产者-消费者模式中，生产者线程只需要将任务添加到任务队列而无需等待任务被消费者线程执行完，也就是说任务的生产和消费是异步的，这是与传统的方法之间调用的本质区别，传统的方法之间调用是同步的。

你或许会有这样的疑问，异步化处理最简单的方式就是创建一个新的线程去处理，那中间增加一个“任务队列”究竟有什么用呢？我觉得主要还是用于**平衡生产者和消费者的速度差异**。我们假设生产者的速率很慢，而消费者的速率很高，比如是1:3，如果生产者有3个线程，采用创建新的线程的方式，那么会创建3个子线程，而采用生产者-消费者模式，消费线程只需要1个就可以了。Java语言里，Java线程和操作系统线程是一一对应的，线程创建得太多，会增加上下文切换的成本，所以Java线程不是越多越好，适量即可。而生产者-消费者模式恰好能支持你用适量的线程。

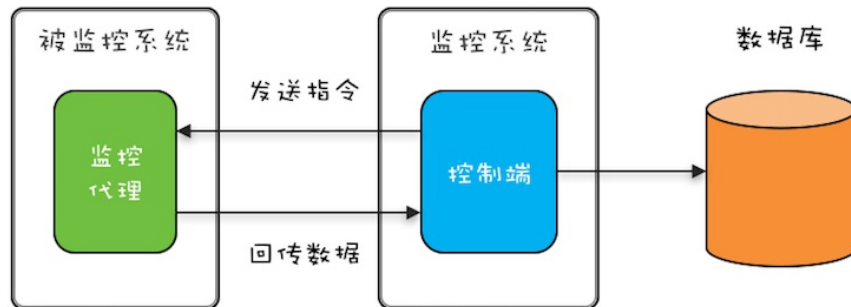
支持批量执行以提升性能

前面我们在[《33 | Thread-Per-Message模式：最简单实用的分工方法》](#)中讲过轻量级的线程，如果使用轻量级线程，就没有必要平衡生产者和消费者的速度差异了，因为轻量级线程本身就是廉价的，那是否意味着生产者-消费者模式在性能优化方面就无用武之地了呢？当然不是，有一类并发场景应用生产者-消费者模式就有奇效，那就是**批量执行任务**。

例如，我们要在数据库里INSERT 1000条数据，有两种方案：第一种方案是用1000个线程并发执行，每个线程INSERT一条数据；第二种方案是用1个线程，执行一个批量的SQL，一次性把1000条数据INSERT进去。这两种方案，显然是第二种方案效率更高，其实这样的应用场景就是我们上面提到的批量执行场景。

在[《35 | 两阶段终止模式：如何优雅地终止线程？》](#)文章中，我们提到一个监控系统动态采集的

案例，其实最终回传的监控数据还是要存入数据库的（如下图）。但被监控系统往往有很多，如果每一条回传数据都直接INSERT到数据库，那么这个方案就是上面提到的第一种方案：每个线程INSERT一条数据。很显然，更好的方案是批量执行SQL，那如何实现呢？这就要用到生产者-消费者模式了。



动态采集功能示意图

利用生产者-消费者模式实现批量执行SQL非常简单：将原来直接INSERT数据到数据库的线程作为生产者线程，生产者线程只需将数据添加到任务队列，然后消费者线程负责将任务从任务队列中批量取出并批量执行。

在下面的示例代码中，我们创建了5个消费者线程负责批量执行SQL，这5个消费者线程以while(true){}循环方式批量地获取任务并批量地执行。需要注意的是，从任务队列中获取批量任务的方法pollTasks()中，首先是以阻塞方式获取任务队列中的一条任务，而后则是以非阻塞的方式获取任务；之所以首先采用阻塞方式，是因为如果任务队列中没有任务，这样的方式能够避免无谓的循环。

```
//任务队列
BlockingQueue<Task> bq=new
    LinkedBlockingQueue<>(2000);
//启动5个消费者线程
//执行批量任务
void start() {
    ExecutorService es=xecutors
        .newFixedThreadPool(5);
    for (int i=0; i<5; i++) {
        es.execute(()->{
            try {
                while (true) {
                    //获取批量任务
```

```

        List<Task> ts=pollTasks();

        //执行批量任务

        execTasks(ts);
    }
} catch (Exception e) {
    e.printStackTrace();
}
});
}
}

//从任务队列中获取批量任务
List<Task> pollTasks()
    throws InterruptedException{
    List<Task> ts=new LinkedList<>();

    //阻塞式获取一条任务
    Task t = bq.take();
    while (t != null) {
        ts.add(t);

        //非阻塞式获取一条任务
        t = bq.poll();
    }
    return ts;
}

//批量执行任务
execTasks(List<Task> ts) {
    //省略具体代码无数
}

```

支持分阶段提交以提升性能

利用生产者-消费者模式还可以轻松地支持一种分阶段提交的应用场景。我们知道写文件如果同步刷盘性能会很慢，所以对于不是很重要的数据，我们往往采用异步刷盘的方式。我曾经参与过一个项目，其中的日志组件是自己实现的，采用的就是异步刷盘方式，刷盘的时机是：

1. **ERROR**级别的日志需要立即刷盘；
2. 数据积累到**500**条需要立即刷盘；
3. 存在未刷盘数据，且**5**秒钟内未曾刷盘，需要立即刷盘。

这个日志组件的异步刷盘操作本质上其实就是一种**分阶段提交**。下面我们具体看看用生产者-消费者模式如何实现。在下面的示例代码中，可以通过调用 `info()`和`error()` 方法写入日志，这两个方法都是创建了一个日志任务**LogMsg**，并添加到阻塞队列中，调用 `info()`和`error()` 方法的线程是生产者；而真正将日志写入文件的是消费者线程，在**Logger**这个类中，我们只创建了**1**个消费者线程，在这个消费者线程中，会根据刷盘规则执行刷盘操作，逻辑很简单，这里就不赘述了。

```
class Logger {  
    //任务队列  
    final BlockingQueue<LogMsg> bq  
        = new BlockingQueue<>();  
    //flush批量  
    static final int batchSize=500;  
    //只需要一个线程写日志  
    ExecutorService es =  
        Executors.newFixedThreadPool(1);  
    //启动写日志线程  
    void start(){  
        File file=File.createTempFile(  
            "foo", ".log");  
        final FileWriter writer=  
            new FileWriter(file);  
        this.es.execute(()->{  
            try {  
                //未刷盘日志数量  
                int curIdx = 0;  
                long preFT=System.currentTimeMillis();  
                while (true) {  
                    LogMsg log = bq.poll(  
                        5, TimeUnit.SECONDS);  
                    //写日志  
                    if (log != null) {  
                        writer.write(log.toString());  
                        ++curIdx;  
                    }  
                    //如果不存在未刷盘数据，则无需刷盘  
                    if (curIdx <= 0) {  
                        continue;  
                    }  
                }  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        });  
    }  
}
```

```

    }
    //根据规则刷盘
    if (log!=null && log.level==LEVEL.ERROR ||
        curIdx == batchSize ||
        System.currentTimeMillis()-preFT>5000){
        writer.flush();
        curIdx = 0;
        preFT=System.currentTimeMillis();
    }
}
}catch(Exception e){
    e.printStackTrace();
} finally {
    try {
        writer.flush();
        writer.close();
    }catch(IOException e){
        e.printStackTrace();
    }
}
});
}

//写INFO级别日志
void info(String msg) {
    bq.put(new LogMsg(
        LEVEL.INFO, msg));
}

//写ERROR级别日志
void error(String msg) {
    bq.put(new LogMsg(
        LEVEL.ERROR, msg));
}
}

//日志级别
enum LEVEL {
    INFO, ERROR
}

class LogMsg {

```

```

class LogMsg {
    LEVEL level;
    String msg;
    //省略构造函数实现
    LogMsg(LEVEL lM, String msg){}
    //省略toString()实现
    String toString(){
    }
}

```

总结

Java语言提供的线程池本身就是一种生产者-消费者模式的实现，但是线程池中的线程每次只能从任务队列中消费一个任务来执行，对于大部分并发场景这种策略都没有问题。但是有些场景还是需要自己来实现，例如需要批量执行以及分阶段提交的场景。

生产者-消费者模式在分布式计算中的应用也非常广泛。在分布式场景下，你可以借助分布式消息队列（**MQ**）来实现生产者-消费者模式。**MQ**一般都会支持两种消息模型，一种是点对点模型，一种是发布订阅模型。这两种模型的区别在于，点对点模型里一个消息只会被一个消费者消费，和**Java**的线程池非常类似（**Java**线程池的任务也只会被一个线程执行）；而发布订阅模型里一个消息会被多个消费者消费，本质上是一种消息的广播，在多线程编程领域，你可以结合观察者模式实现广播功能。

课后思考

在日志组件异步刷盘的示例代码中，写日志的线程以 `while(true){}` 的方式执行，你有哪些办法可以优雅地终止这个线程呢？

```

this.writer.execute()->{
    try {
        //未刷盘日志数量
        int curIdx = 0;
        long preFT=System.currentTimeMillis();
        while (true) {
            .....
        }
    } catch(Exception e) {}
}

```

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。



Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



锦

👍 21

在应用系统中，日志系统一般都是最后关闭的吧，因为它要为其他系统关闭提供写日志服务。所以日志系统关闭时需要把队列中所有日志都消费掉才能关闭。可能需要在关闭日志系统时投入一个毒丸，表示没有新的日志写入。线程池在消费到毒丸时知道没有日志写入，将所有的日志刷盘，**break**循环体。

2019-05-21

作者回复

👍

2019-05-21



PK時頭髮不亂

👍 11

极客时间有好多课程, 我觉得王老师的干货是最实际最可用的, 必须要赞一个。

2019-05-21

作者回复

感谢感谢, 有钱难买合适:)

2019-05-21



ack

👍 2

```
public class Logger {
```


...

```
volatile boolean stop;
```

```
// 启动写日志线程
```

```
void start() throws IOException {
```

```
...
```

```
this.es.execute() -> {
```

```
try {
```

```
// 未刷盘日志数量
```

```
int curIdx = 0;
```

```
long preFT = System.currentTimeMillis();
```

```
while (!stop) {
```

```
...
```

```
}
```

```
} catch (InterruptedException e) {
```

```
// 重新设置线程中断状态
```

```
Thread.currentThread().interrupt();
```

```
} catch (Exception e) {
```

```
e.printStackTrace();
```

```
} finally {
```

```
...
```

```
}
```

```
});
```

```
}
```

```
...
```

```
void stop(){
```

```
stop = true;
```

```
es.shutdown();
```

```
}
```

```
}
```

2019-05-21



泛岁月的涟漪

👍 2

1、使用线程池的shutdown或者shutdownNow关闭线程池

2、while循环条件设置为一个volatile boolean变量

3、可以使用interrupt，但是线程是线程池管理的，没有消费者线程的引用中断不了

2019-05-21



êwě n

👍 1

之前遇到过一个生产问题，一个服务启动一段时间后就不停的超时，后面结合线程栈发现很多阻塞在打印日志的地方（我们用的就是log4j2），后面查到机子硬盘问题，io直接100%以上，

日志刷盘满导致消费速度慢，队列撑满阻塞了写，这间接说明平衡好生产和消费速度以及适当的队列大小是很有必要。

2019-05-22

| 作者回复

能快速定位的问题III

2019-05-23



张三

👍 1

还是不太懂，线程池的实现是有两种模式吗？Worker Thread 和 生产者-消费者 模式？

2019-05-22



苏柏

👍 1

您好老师问个最近用到的线程池使用的问题

我的工程是springboot的，在unitTest里（@SpringBootTest）里调用了一个service A（通过@Autowired的）中的方法，A中启用了线程池，执行的任务是往数据库里插入数据。但是总抛出数据源已经被关闭的异常，我理解的是在单测主线程已经结束，所以关闭了数据源这些清理工作，而此时线程池的线程还没结束，这个时候去调用数据源是null的，不知道这么理解对不对，另外这个test主线程结束，为啥线程池的线程还没结束（通过打断点看到的）。这个怎么理解，求教

2019-05-22

| 作者回复

只有守护线程才会自动结束，线程池的线程不是守护线程

2019-05-22



berthav_ss

👍 0

宝令老师，如何优雅的停止线程池中某一组线程呢？例如我在线程a中启动了1-10线程，线程b中启动了2-30线程，如何优雅停止1-10线程呢

2019-06-11

| 作者回复

可以考虑一下毒丸的方式

2019-06-12



lingw

👍 0

worker Thread模式，worker相当于线程池里面的线程

生产者-消费者，相当于我们自己维护队列，队列，出队操作，然后比如消费者的话就可以直接使用线程池来消费，和worker Thread模式的区别和联系的话，worker Thread就是线程池工作的线程，生产者消费者就是线程池（worker Thread + 内部的队列+对内部队列的操作）+外部自己维护的队列+对外部队列的操作。

课后习题：多加个stop（）{terminated = true;es.shutdown()}方法，将上面的while(true)改成while(!terminated)

2019-05-26



聂旋

👍 0

安卓的主线程中也是采用消息队列加消息循环方式，来处理用户输入及各种事件。当应用退出

时，会发送一个处理对象为`null`的消息给队列，消息循环遇到这样的消息时就退出了。

2019-05-23



中年英雄王叔叔

👍 0

我的理解是工人和工厂是耦合的，即线程池开始就有`worker`在了，而工厂和消费者则没什么关系

2019-05-23



曾弢麟

👍 0

补充一下上面的留言，先通过创建的钩子去创建一个毒丸，然后释放资源

2019-05-23



曾弢麟

👍 0

使用`Runtime`提供的钩子，然后在关闭前，先让内部任务执行完毕，再释放资源

2019-05-23



ban

👍 0

我的想法是：弄一个`volatile`可见性的标志位，当出发标志位的时候检查当前是否还有日志，有则取出来，直到阻塞队列为空，然后刷盘写入数据，然后`break`退出即可。

老师你看是否可行

2019-05-22



ban

👍 0

```
while (t != null) {  
    ts.add(t);  
    // 非阻塞式获取一条任务  
    t = bq.poll();  
}
```

加入集合的时候，是不是加个条数限制比较好，防止数据过多，把集合撑爆了

2019-05-21



Geek_bbbda3

👍 0

`volatile stoped;`

```
while(!stoped || bq.size() >0)  
{ }
```

```
public void shutdown () {  
    Stoped =true;  
    Es.shutdown () ;  
    while (es.awaitUtilTime (5, timeutil.seconds) {  
        es.shutdownNow () ;
```

```
}
```

2019-05-21



佑儿

👍 0

我觉得这里虽然没有使用可中断操作，如果不考虑第三方的调用影响，也可以使用线程中断标识，只不过不能捕获异常。

2019-05-21



佑儿

👍 0

声明一个`volatile`变量用于表示线程结束，为`true`时，退出循环

2019-05-21

作者回复

队列中的任务就丢了

2019-05-21



晓杰

👍 0

35讲说到优雅地终止线程，首先需要线程状态转换为`runnable`状态（在终止刷盘的方法中调用`Thread.currentThread().interrupt()`方法）

然后通过设置标志位来让线程优雅终止，具体有两种方式：

- 1、通过判断线程的中断状态`Thread.currentThread().isInterrupted()`
- 2、设置自己的线程终止标志位，该标志位`volatile`修饰的共享变量。（这种方式需要在终止刷盘的方法中修改该共享变量的值）

2019-05-21



张三

👍 0

打卡！监控系统那个例子里边`pollTasks()`方法里边的`while`循环怎么退出？那个`while`循环会一直把阻塞获取的第一条任务添加到任务队列。

2019-05-21