二

06 案例分析：缓冲区如何让代码加速

本课时将详细介绍“缓冲”这个优化手段，之前在 02 课时的复用优化中便提到过“缓冲”，你可以回看复习一下。

深入理解缓冲的本质

缓冲（Buffer）通过对数据进行暂存，然后批量进行传输或者操作，多采用顺序方式，来缓解不同设备之间次数频繁但速度缓慢的随机读写。

你可以把缓冲区，想象成一个蓄水池。放水的水龙头一直开着，如果池子里有水，它就以恒定的速度流淌，不需要暂停；供水的水龙头速度却不确定，有时候会快一些，有时候会特别慢。它通过判断水池里水的状态，就可以自由控制进水的速度。

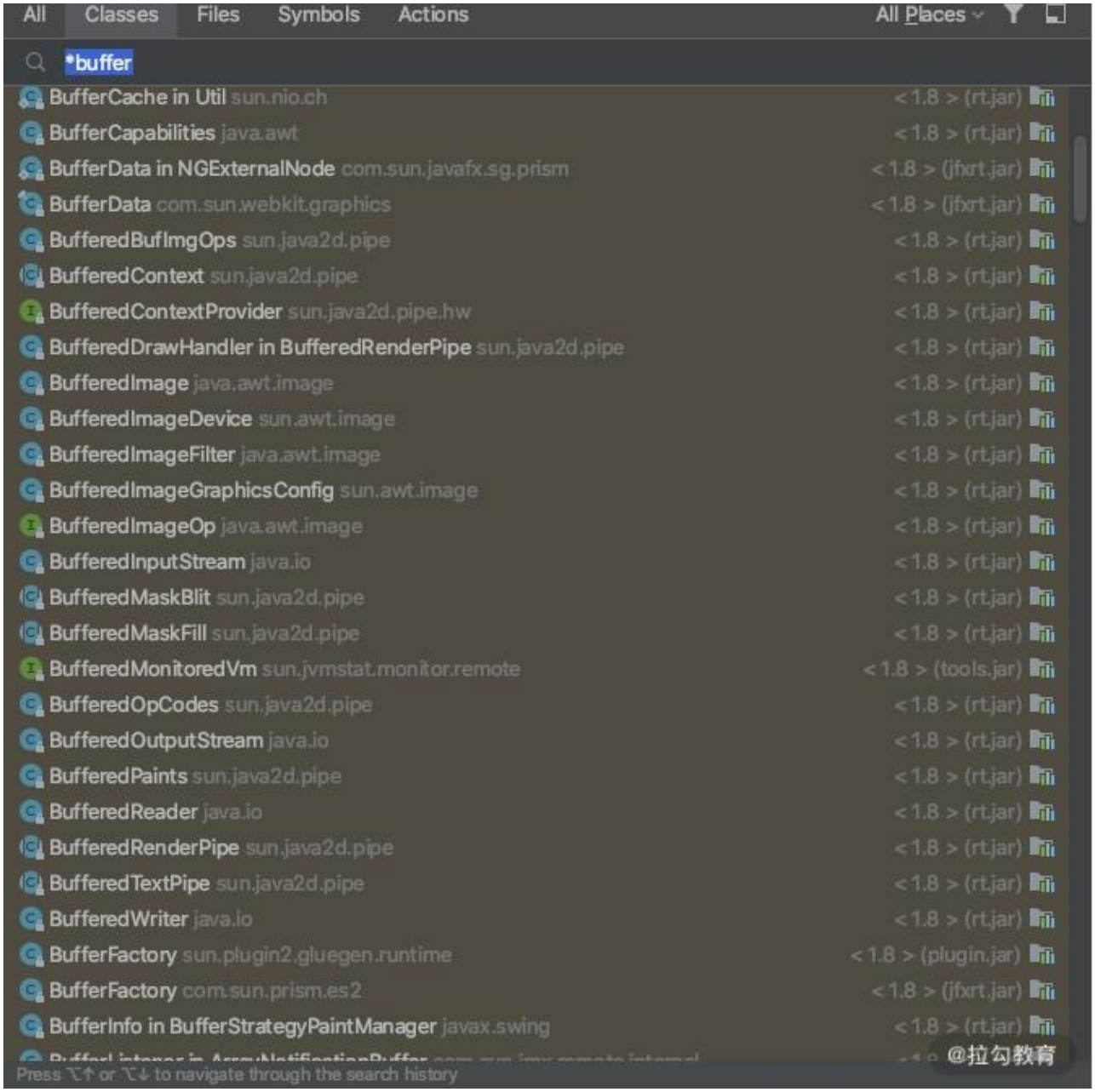
或者再想象一下包饺子的过程，包馅的需要等着擀皮的。如果擀皮的每擀一个就交给包馅的，速度就会很慢；但如果中间放一个盆子，擀皮的只管往里扔，包馅的只管从盆里取，这个过程就快得多。许多工厂流水线也经常使用这种方法，可见“缓冲”这个理念的普及性和实用性。

从宏观上来说，JVM 的堆就是一个大的缓冲区，代码不停地在堆空间中生产对象，而垃圾回收器进程则在背后默默地进行垃圾回收。

通过上述比喻和释意，你可以发现**缓冲区的好处**：

- 缓冲双方能各自保持自己的操作节奏，操作处理顺序也不会打乱，可以 one by one 顺序进行；
- 以批量的方式处理，减少网络交互和繁重的 I/O 操作，从而减少性能损耗；
- 优化用户体验，比如常见的音频/视频缓冲加载，通过提前缓冲数据，达到流畅的播放效果。

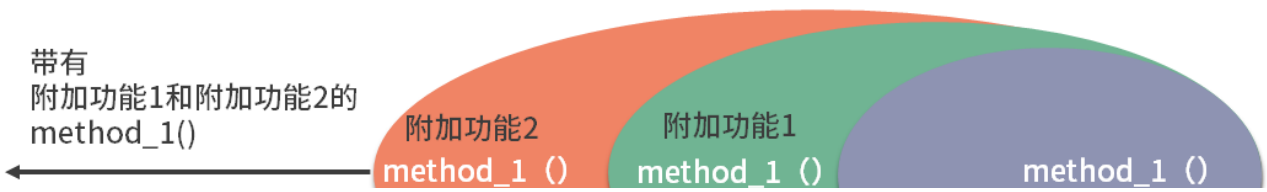
缓冲在 Java 语言中被广泛应用，在 IDEA 中搜索 Buffer，可以看到长长的类列表，其中最典型的就是**文件读取和写入字符流**。



文件读写流

接下来，我会以文件读取和写入字符流为例进行讲解。

Java 的 I/O 流设计，采用的是**装饰器模式**，当需要给类添加新的功能时，就可以将被装饰者通过参数传递到装饰者，封装成新的功能方法。下图是装饰器模式的典型示意图，就增加功能来说，装饰模式比生成子类更为灵活。



在读取和写入流的 API 中，BufferedInputStream 和 BufferedReader 可以加快读取字符的速度，BufferedOutputStream 和 BufferedWriter 可以加快写入的速度。

下面是直接读取文件的代码实现：

```
int result = 0;
try (Reader reader = new FileReader(FILE_PATH)) {
    int value;
    while ((value = reader.read()) != -1) {
        result += value;
    }
}
return result;
```

要使用缓冲方式读取，只需要将 FileReader 装饰一下即可：

```
int result = 0;
try (Reader reader = new BufferedReader(new FileReader(FILE_PATH))) {
    int value;
    while ((value = reader.read()) != -1) {
        result += value;
    }
}
return result;
```

我们先看一下与之类似的，BufferedInputStream 类的具体实现方法：

```
//代码来自JDK
public synchronized int read() throws IOException {
        if (pos >= count) {
            fill();
            if (pos >= count)
                return -1;
        }
        return getBufIfOpen()[pos++] & 0xff;
    }
```

当缓冲区的内容读取完毕，将尝试使用 fill 函数把输入流读入缓冲区：

```
//代码来自JDK
private void fill() throws IOException {
    byte[] buffer = getBufIfOpen();
```

```
        if (markpos < 0)
            pos = 0;            /* no mark: throw away the buffer */
        else if (pos >= buffer.length)  /* no room left in buffer */
            if (markpos > 0) {  /* can throw away early part of the buffer */
                int sz = pos - markpos;
                System.arraycopy(buffer, markpos, buffer, 0, sz);
                pos = sz;
                markpos = 0;
            } else if (buffer.length >= marklimit) {
                markpos = -1;   /* buffer got too big, invalidate mark */
                pos = 0;        /* drop buffer contents */
            } else if (buffer.length >= MAX_BUFFER_SIZE) {
                throw new OutOfMemoryError("Required array size too large");
            } else {            /* grow buffer */
                int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                        pos * 2 : MAX_BUFFER_SIZE;
                if (nsz > marklimit)
                    nsz = marklimit;
                byte nbuf[] = new byte[nsz];
                System.arraycopy(buffer, 0, nbuf, 0, pos);
                if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                    // Can't replace buf if there was an async close.
                    // Note: This would need to be changed if fill()
                    // is ever made accessible to multiple threads.
                    // But for now, the only way CAS can fail is via close.
                    // assert buf == null;
                    throw new IOException("Stream closed");
                }
                buffer = nbuf;
            }
        count = pos;
        int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
        if (n > 0)
            count = n + pos;
    }
```

程序会调整一些读取的位置，并对缓冲区进行位置更新，然后使用被装饰的 InputStream 进行数据读取：

```
int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
```

那么为什么要这么做呢？直接读写不行吗？

这是因为：字符流操作的对象，一般是文件或者 Socket，要从这些缓慢的设备中，通过频繁的交互获取数据，效率非常慢；而缓冲区的数据是保存在内存中的，能够显著地提升读写速度。

既然好处那么多，为什么不把所有的数据全部读到缓冲区呢？

这就是一个权衡的问题，缓冲区开得太大，会增加单次读写的时间，同时内存价格很高，不

能无限制使用，缓冲流的默认缓冲区大小是 8192 字节，也就是 8KB，算是一个比较折中的值。

这好比搬砖，如果一块一块搬，时间便都耗费在往返路上了；但若给你一个小推车，往返的次数便会大大降低，效率自然会有所提升。

下图是使用 FileReader 和 BufferedReader 读取文件的 JMH 对比（相关代码见仓库），可以看到，使用了缓冲，读取效率有了很大的提升（暂未考虑系统文件缓存）。

```
Benchmark                                Mode  Cnt    Score     Error  Units
BenchmarkReader.bufferedReaderTest       avgt    5   77.946 ±  12.640  ms/op
BenchmarkReader.fileReadTest             avgt    5  131.570 ± 111.055  ms/op
```


日志缓冲

日志是程序员们最常打交道的地方。在高并发应用中，即使对日志进行了采样，日志数量依旧惊人，所以选择高速的日志组件至关重要。

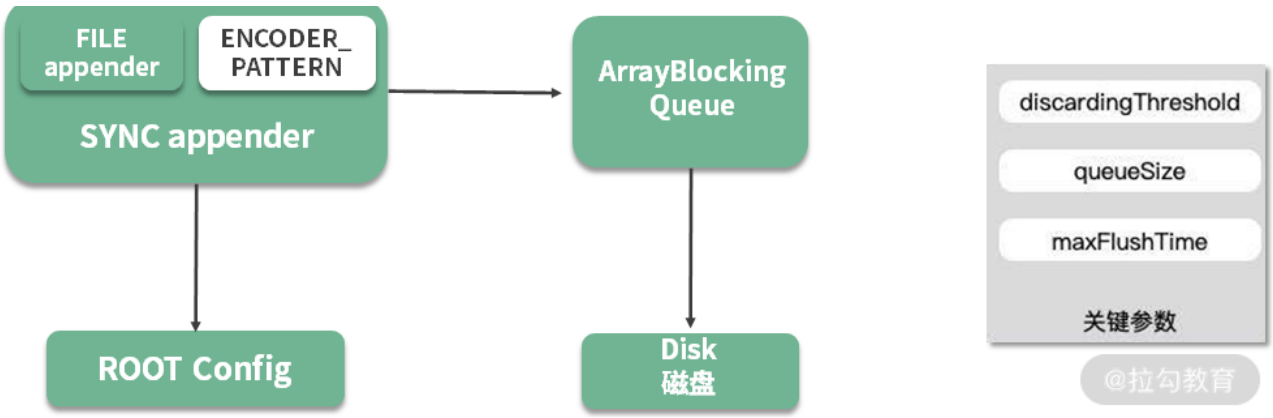
SLF4J 是 Java 里标准的日志记录库，它是一个允许你使用任何 Java 日志记录库的抽象适配层，最常用的实现是 Logback，支持修改后自动 reload，它比 Java 自带的 JUL 还要流行。

Logback 性能也很高，其中一个原因就是**异步日志**，它在记录日志时，使用了一个缓冲队列，当缓冲的内容达到一定的阈值时，才会把缓冲区的内容写到文件里。使用异步日志有两个考虑：

- 同步日志的写入，会阻塞业务，导致服务接口的耗时增加；
- 日志写入磁盘的代价是昂贵的，如果每产生一条日志就写入一次，CPU 会花很多时间在磁盘 I/O 上。

Logback 的异步日志也比较好配置，我们需要在正常配置的基础上，包装一层异步输出的逻辑（详见仓库）。

```
<appender name ="ASYNC" class= "ch.qos.logback.classic.AsyncAppender">
        <discardingThreshold >0</discardingThreshold>
        <queueSize>512</queueSize>
        <!--这里指定了一个已有的Appender-->
        <appender-ref ref ="FILE"/>
</appender>
```



如上图，异步日志输出之后，日志信息将暂存在 ArrayBlockingQueue 列表中，后台会有一个 Worker 线程不断地获取缓冲区内容，然后写入磁盘中。

上图中有三个关键参数：

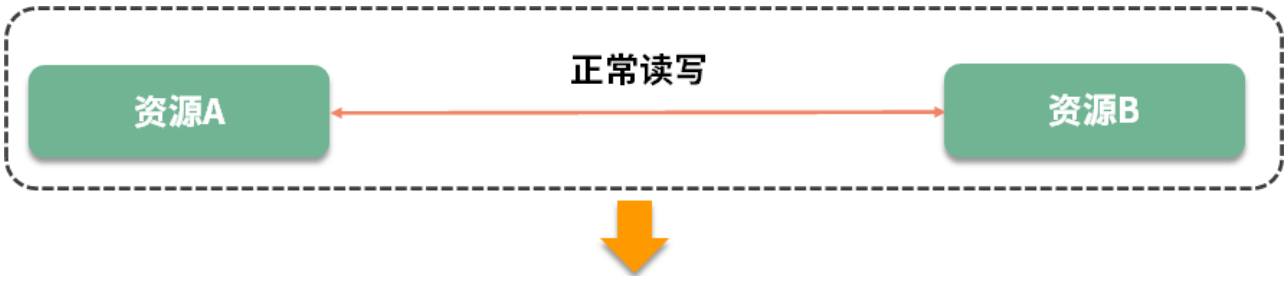
- **queueSize**，代表了队列的大小，默认是256。如果这个值设置的太大，大日志量下突然断电，会丢掉缓冲区的内容；
- **maxFlushTime**，关闭日志上下文后，继续执行写任务的时间，这是通过调用 Thread 类的 join 方法来实现的（worker.join(maxFlushTime)）；
- **discardingThreshold**，当 queueSize 快达到上限时，可以通过配置，丢弃一些级别比较低的日志，这个值默认是队列长度的 80%；但若你担心可能会丢失业务日志，则可以将这个值设置成 0，表示所有的日志都要打印。

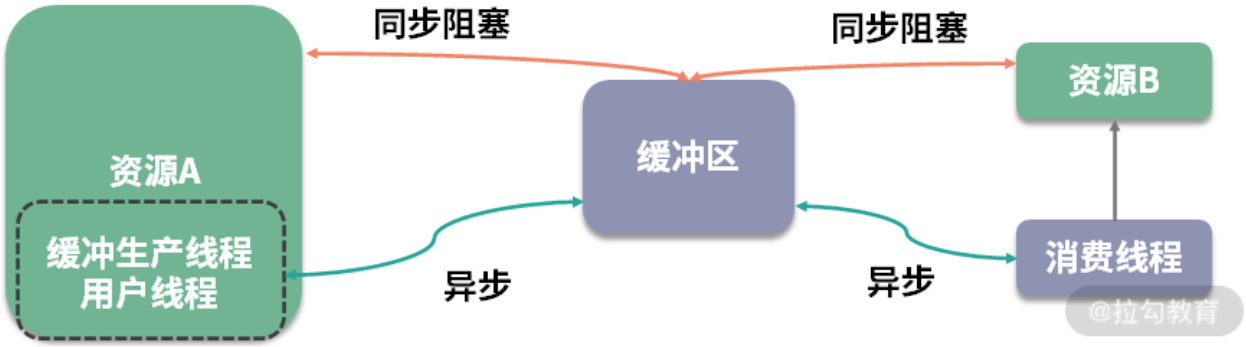
缓冲区优化思路

毫无疑问缓冲区是可以提高性能的，**但它通常会引入一个异步的问题**，使得编程模型变复杂。

通过文件读写流和 Logback 两个例子，我们来看一下对于缓冲区设计的一些常规操作。

如下图所示，资源 A 读取或写入一些操作到资源 B，这本是一个正常的操作流程，但由于中间插入了一个额外的**存储层**，所以这个流程被生生截断了，这时就需要你手动处理被截断两方的资源协调问题。



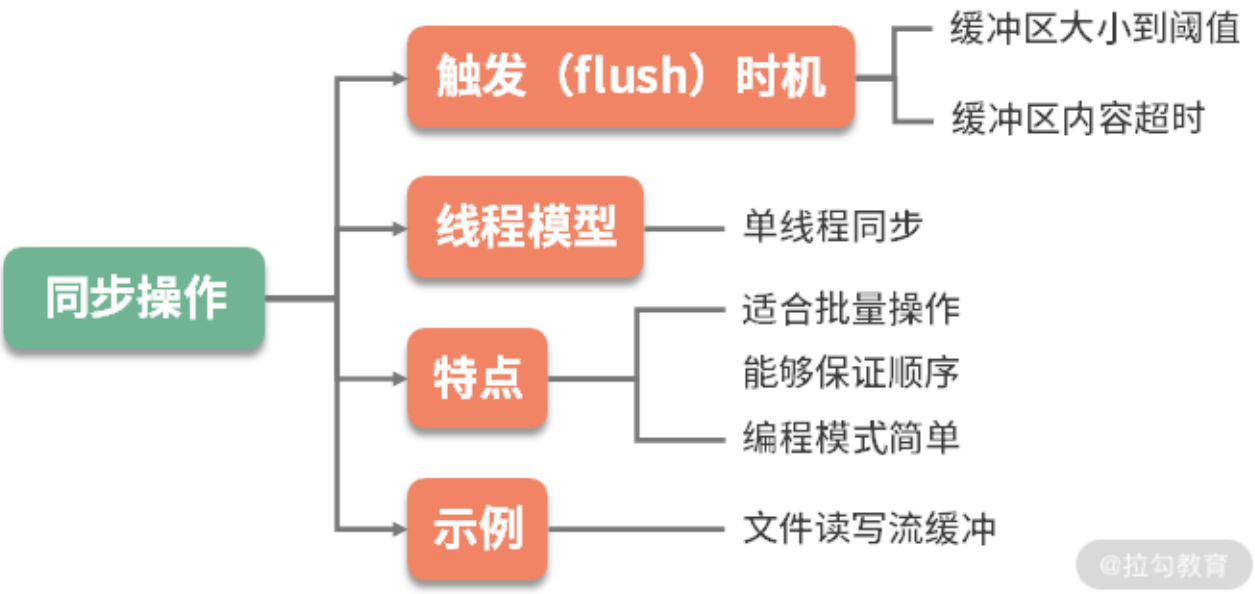


根据资源的不同，对正常业务进行截断后的操作，分为同步操作和异步操作。

1.同步操作

同步操作的编程模型相对简单，在一个线程中就可完成，你只需要控制缓冲区的大小，并把握处理的时机。比如，缓冲区**大小达到阈值**，或者缓冲区的元素在缓冲区的**停留时间超时**，这时就会触发**批量操作**。

由于所有的操作又都在**单线程**，或者同步方法块中完成，再加上资源 B 的处理能力有限，那么很多操作就会阻塞并等待在调用线程上。比如写文件时，需要等待前面的数据写入完毕，才能处理后面的请求。



2.异步操作

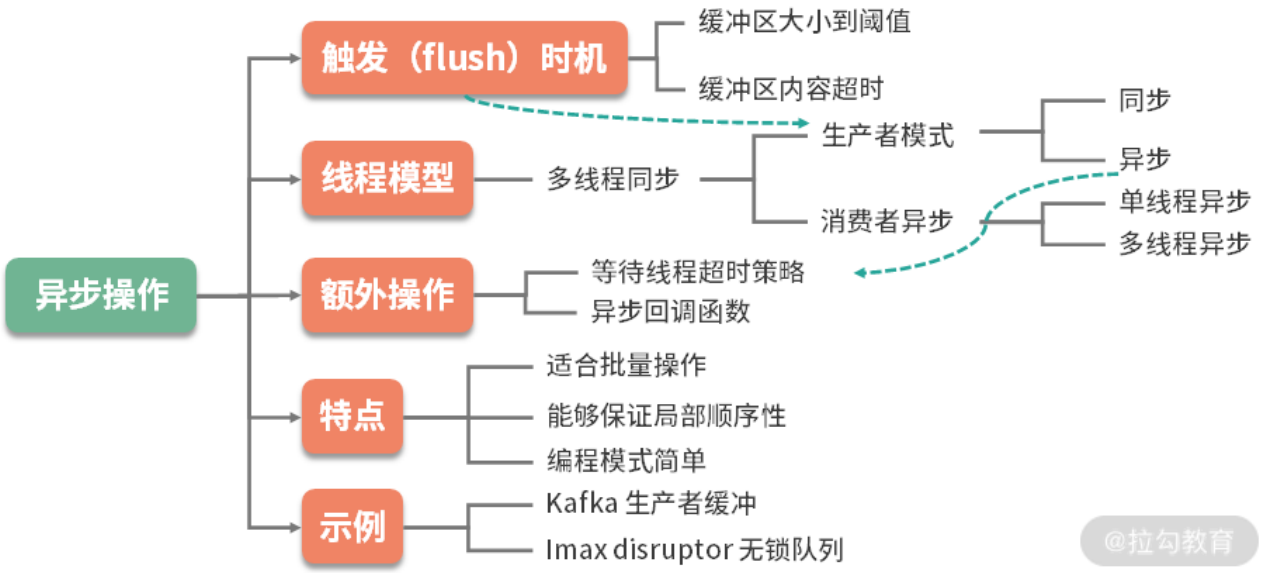
异步操作就复杂很多。

缓冲区的**生产**者一般是同步调用，但也可以采用异步方式进行填充，一旦采用异步操作，就涉及缓冲区满了以后，生产者的一些响应策略。

此时，应该将这些策略抽象出来，根据业务的属性选择，比如直接抛弃、抛出异常，或者直接在用户的线程进行等待。你会发现它与线程池的饱和策略是类似的，这部分的详细概念将在 12 课时讲解。

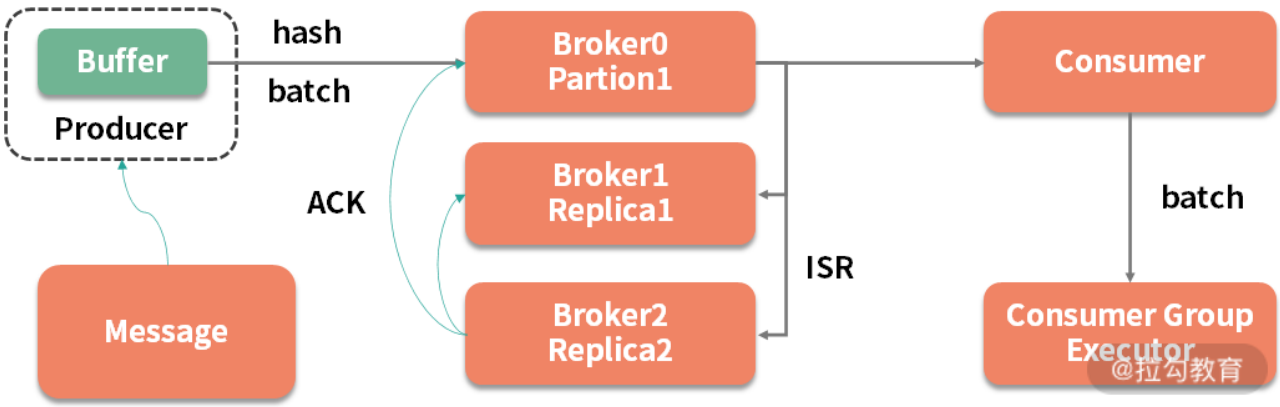
许多应用系统还会有更复杂的策略，比如在用户线程等待，设置一个超时时间，以及成功进入缓冲区之后的回调函数等。

对缓冲区的**消费**，一般采用开启线程的方式，如果有多个线程消费缓冲区，还会存在信息同步和顺序问题。



3.Kafka缓冲区示例

这里以一个常见的面试题来讲解上面的知识点：Kafka 的生产者，有可能会丢数据吗？



如图，要想解答这个问题，需要先了解 Kafka 对生产者的一些封装，其中有一个对性能影响非常大的点，就是缓冲。

生产者会把发送到同一个 partition 的多条消息，封装在一个 batch（缓冲区）中。当 batch 满了（参数 batch.size），或者消息达到了超时时间（参数 linger.ms），缓冲区中的消息就会被发送到 broker 上。

这个缓冲区默认是 16KB，如果生产者的业务突然断电，这 16KB 数据是没有机会发送出去的。此时，就造成了消息丢失。

解决的办法有两种：

- 把缓冲区设置得非常小，此时消息会退化成单条发送，这会严重影响性能；
- 消息发送前记录一条日志，消息发送成功后，通过回调再记录一条日志，通过扫描生成的日志，就可以判断哪些消息丢失了。

另外一个面试的问题是：Kafka 生产者会影响业务的高可用吗？

这同样和生产者的缓冲区有关。缓冲区大小毕竟是有限制的，如果消息产生得过快，或者生产者与 broker 节点之间有网络问题，缓冲区就会一直处于 full 的状态。此时，有新的消息到达，会如何处理呢？

通过配置生产者的超时参数和重试次数，可以让新的消息一直阻塞在业务方。一般来说，这个超时值设置成 1 秒就已经够大了，有的应用在线上把超时参数配置得非常大，比如 1 分钟，就造成了用户的线程迅速占满，整个业务不能再接受新的请求。

4.其他做法

使用缓冲区来提升性能的做法非常多，下面再举几个例子：

- StringBuilder 和 StringBuffer，通过将要处理的字符串缓冲起来，最后完成拼接，提高字符串拼接的性能；
- 操作系统在写入磁盘，或者网络 I/O 时，会开启特定的缓冲区，来提升信息流转的效率。通常可使用 flush 函数强制刷新数据，比如通过调整 Socket 的参数 SO_SNDBUF 和 SO_RCVBUF 提高网络传输性能；
- MySQL 的 InnoDB 引擎，通过配置合理的 innodb_buffer_pool_size，减少换页，增加数据库的性能；
- 在一些比较底层的工具中，也会变相地用到缓冲。比如常见的 ID 生成器，使用方通过缓冲一部分 ID 段，就可以避免频繁、耗时的交互。

5.注意事项

虽然缓冲区可以帮我们大大地提高应用程序的性能，但同时它也有不少问题，在我们设计时，要注意这些异常情况。

其中，**比较严重就是缓冲区内容的丢失。**即使你使用 addShutdownHook 做了优雅关闭，有些情形依旧难以防范避免，比如机器突然间断电，应用程序进程突然死亡等。这时，缓冲区内未处理完的信息便会丢失，尤其金融信息，电商订单信息的丢失都是比较严重的。

所以，**内容写入缓冲区之前，需要先预写日志**，故障后重启时，就会根据这些日志进行数据恢复。在数据库领域，文件缓冲的场景非常多，一般都是采用 WAL 日志（Write-Ahead Logging）解决。对数据完整性比较严格的系统，甚至会通过电池或者 UPS 来保证缓冲区的落地。这就是性能优化带来的新问题，必须要解决。

小结

可以看到，缓冲区优化是对正常的业务流程进行截断，然后加入缓冲组件的一个操作，它分为同步和异步方式，其中异步方式的实现难度相对更高。

大多数组件，从操作系统到数据库，从 Java 的 API 到一些中间件，都可以通过设置一些参数，来控制缓冲区大小，从而取得较大的性能提升。但需要注意的是，某些极端场景（断电、异常退出、kill -9等）可能会造成数据丢失，若你的业务对此容忍度较低，那么你需要花更多精力来应对这些异常。

在我们面试的时候，除了考察大家对知识细节的掌握程度，还会考察总结能力，以及遇到相似问题的分析能力。大家在平常的工作中，也要多多总结，多多思考，窥一斑而知全貌。如此回答，必会让面试官眼前一亮。

上一页 下一页