彩蛋 1 | 竟然还有分布式的 JVM?

经过前面的 21 讲，我们应该说是自底向上、从理论到实践，对实时流计算技术有了一个系统的理解。我们在分析和对比了四种不同的开源流计算框架后，认为 Flink 是当前最好的开源流计算框架。

不过今天，我们会从另外一个有趣的角度来分析 Flink，也就是将 Flink 视为一个分布式的 JVM。你会发现，通过这种认识方式，我们对 Flink 和分布式计算甚至还有微服务的理解，会更上一个层次。

冯诺依曼结构计算机

要说分布式的 JVM，得先从单节点的 JVM（Java 虚拟机）说起。单节点的 JVM，说白了就是一个冯诺依曼结构的计算机，也就是下面图 1 所示的系统。

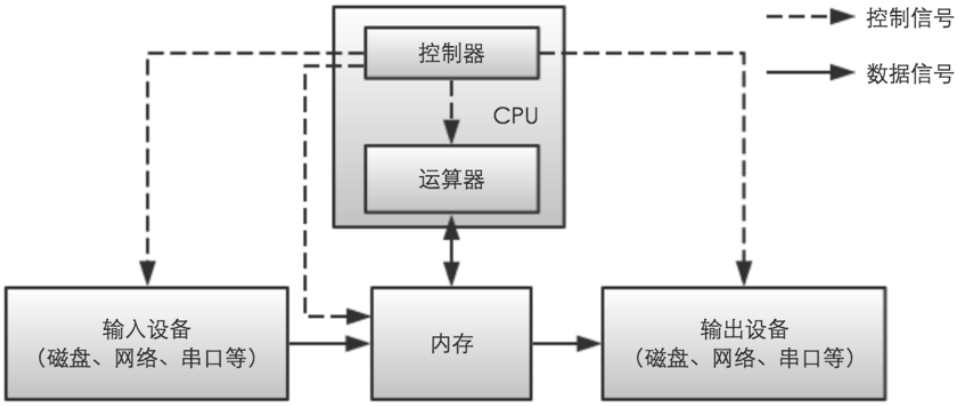

图 1 冯诺依曼结构计算机

在上面图 1 展示的冯诺依曼结构计算机中，计算机由三个部分构成，即 CPU、内存和 I/O（输入/输出）设备。其中，CPU 在执行计算任务时，主要是通过运算器从内存读写数据。在 CPU 需要访问磁盘（属于I/O设备的一种）时，它并非自己直接访问磁盘上的数据，而是先通过控制器控制 I/O 设备驱动程序，将磁盘上的数据加载到内存，然后再由运算器读写内存上的数据。

所以总的来说，在冯诺依曼结构计算机中，**CPU 在进行计算时，最主要的数据交互对象是内存**，而磁盘在其中的作用只是进行数据的持久化存储。

分布式的JVM

我们在第 19 课时讨论 Flink 时，曾讲到过 Flink 的系统架构如下图 2 所示。

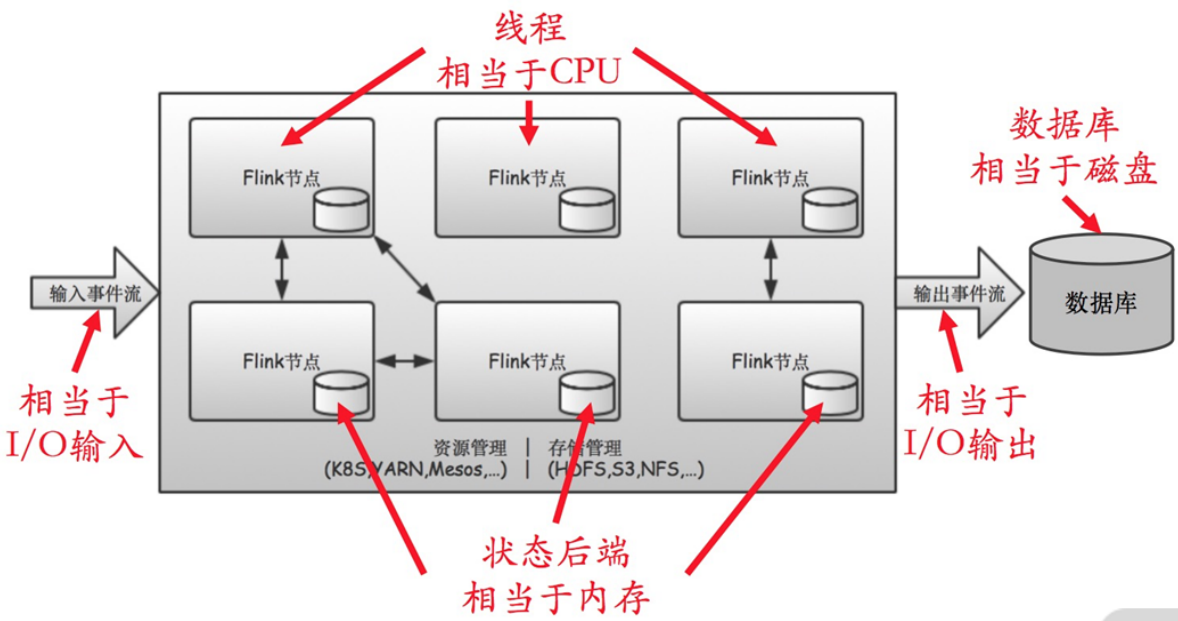

图 2　Flink系统架构图

在 Flink 系统架构中，它明确地将**状态管理**纳入到了它的系统架构中。在各个 Flink 节点进行计算时，它将**状态**保存到本地，并通过 checkpoint 机制和诸如 HDFS 这样的分布式文件系统，实现了状态的分布式管理。

在 Flink 作业的计算过程中，**每个 Flink 计算节点（说得更具体点就是执行计算任务的线程）最主要的数据交互对象是状态后端。这和冯诺依曼结构计算机中，CPU 最主要的数据交互对象是内存的情况，是完全对应的！**

而且，Flink 在处理完数据后，结果是通过输出流再保存到数据库。这与冯诺依曼结构计算机中，计算完成后结果经过输出设备驱动保存到磁盘的过程，也是完全相似的！

所以说，如果我们将 Flink 集群整体视为一个冯诺伊曼结构计算机的话，那么**Flink 的计算节点（说得更具体点就是执行计算任务的线程）就对应着 CPU，而 Flink 计算节点上的状态后端就对应着内存，输入、输出流就对应着 I/O 设备了**。而且更加厉害的是，Flink 的计算节点是可以水平扩展的，计算节点上的状态后端（比如内存、文件和 RocksDB）也是实现了分布式存储和管理的。

这样的话，Flink 就成了一个**CPU 和内存都可以近乎无限扩展的冯诺依曼机器，** 是不是非常惊艳！这不正是任何一个开发人员都梦寐以求的最顶配“服务器”吗？

另外，如果你已经下载和运行过，我在前面课时中给出的 Flink 示例代码的话，你会发现这些代码不仅可以运行在单独的 JVM 里，它们也可以运行在 Flink 集群里。换句话说，我们所编写的代码，既可以在 JVM 这个单节点冯诺依曼机器上运行，也可以在 Flink 这个分布式冯诺依曼机器上运行，我们只需要写一套代码即可！

所以说，你能够在单节点 JVM 上做的事情，是不是就可以用完全相同的代码，在 Flink 这个分布式 JVM 上，轻轻松松实现性能的水平扩展。要知道，曾经我们开发一个分布式的系统是多么麻烦，而 Flink 为我们提供的这个分布式 JVM，拥有近乎无限的 CPU 和“内存”，让我们可以方便快捷地开发出一个既可靠又具有高度可扩展性的分布式系统。

讨论到这里，相信你也一定和我一样，内心激动不已吧！

不过这还没完，接下来我们更进一步，做件更加有趣的事情，这就是用 Flink 开发微服务！

使用 Flink 开发微服务的原理

回想下你最常用 JVM 做的事情是什么？是不是开发微服务。比如，最常见的就是 Spring Boot 和 Spring Cloud 这类基于 JVM 的微服务系统了。

现在，我们既然已经有了 Flink 这个惊艳的分布式 JVM，很容易就会想到用它来开发微服务。

下面的图 3 就展示了用 Flink 开发微服务系统的思路。

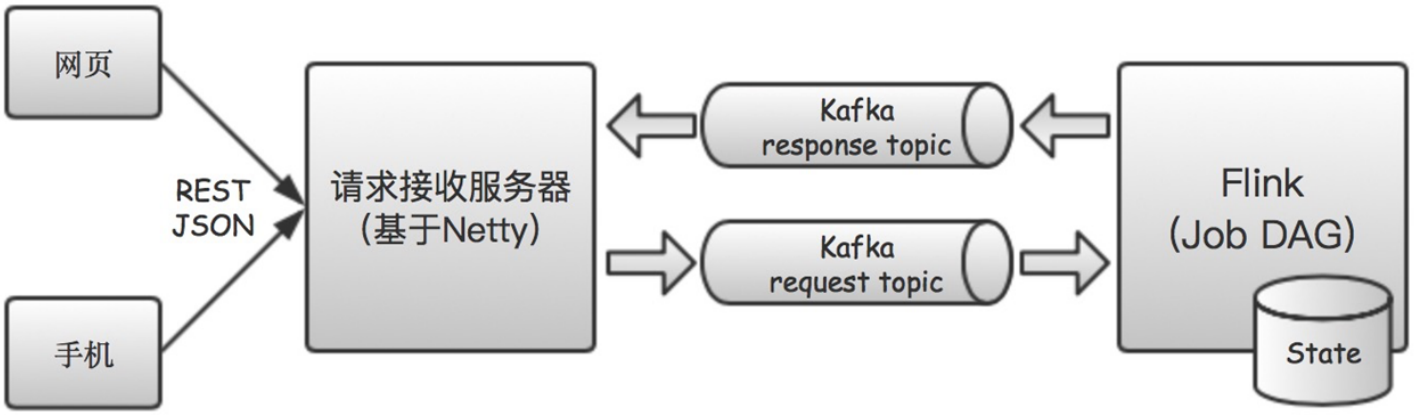

图 3 使用Flink实现微服务系统

在上面的图 3 中，我们的微服务系统整体上采用了事件驱动（也就是异步执行）的方式。首先，REST 请求接收服务器采用基于 Netty 的 NIO 和异步编程框架，将接收到的 REST 请求发送到 Kafka。然后，Flink 从 Kafka 中将请求读取出来进行处理。之后，Flink 再将请求处理结果发送回 Kafka。最后，请求接收服务器从 Kafka 中取出请求处理结果，并将请求处理结果返回给客户端。

这种基于 Flink 的微服务方案，有以下四点优势。

- 一是，整个系统完全是异步的，可以极大提升系统的整体性能，包括请求处理的**吞吐量**和**平均响应时延**。
- 二是，在 Flink 上我们是通过 DAG 来描述业务流程的。由于 DAG 可以描绘得非常复杂，这就意味着我们在 Flink 上可以**实现各种复杂的业务逻辑**。并且，因为 DAG 通常是在程序的一段代码中集中描述的，还可以通过 UI 界面直接观察到，所以我们可以**一目了然地看清整个业务的执行流程**。这样即使是非常复杂的业务流程，开发和管理起来都非常方便，完胜调用关系复杂的传统微服务架构。
- 三是，我们可以灵活地实现资源水平扩展或收缩，只需要设置不同的 Flink 算子并行度即可。这样就可以轻轻松松地提升或降低系统的处理性能。
- 四是，我们可以通过反向压力，轻松方便地实现**自适应的流量控制**，充分发挥出资源的使用效率。

当然，虽然有以上这些优势，但这种基于 Flink 的微服务方案，也并非要取代传统微服务架构，它更多的是对传统微服务架构的补充。因为 DAG 虽然可以描述非常复杂的业务流程，但其中的业务步骤通常是关联比较强、关系比较紧的。相比完全松耦合的传统微服务能够实现的复杂业务逻辑而言，DAG 描述的业务逻辑还是相对紧密和简单些。

以上就介绍完了使用 Flink 开发微服务的原理。接下来，我们再来看看具体怎么实现。

使用 Flink 开发微服务的具体实现

根据前面用 Flink 开发微服务的原理，我们可以看出要开发的模块包括两个，一个是基于 Netty 的请求接收服务器，另一个是基于 Flink 的请求处理服务器集群。

我们先来看基于 Netty 的请求接收服务器。主要代码如下（完整代码请参考这里）：

```
void channelRead0(ChannelHandlerContext ctx, HttpRequest req)
        throws Exception {
    CompletableFuture
            .supplyAsync(() -> this.httpDecode(ctx, req), httpDecoderExecutor)
            .thenAcceptAsync(e -> this.sendRequestToKafka(ctx, req, e, refController), requestExe
}
JSONObject httpDecode(ChannelHandlerContext ctx, HttpRequest req) {
    byte[] body = readRequestBodyAsString((HttpContent) req);
    String jsonString = new String(body, Charsets.UTF_8);
    return JSON.parseObject(jsonString);
}
void sendRequestToKafka(ChannelHandlerContext ctx, HttpRequest req,
                                JSONObject event, RefController ref) {
    // 这里简单地用 UUID 来生成唯一ID。
    // 更严格的唯一ID，应该考虑"主机名 + IP地址 + 进程PID + timestamp + 随机数"等因素，可以参考MongoDb的_id
    String eventId = UUID.randomUUID().toString();
    // EVENT_ID 用于后续接收到响应后，从 blockingMap 中找回之前的请求
    event.put(EVENT_ID_TAG, eventId);
    // 由于请求只是发送到kafka，并不知道什么时候响应能够返回，
    // 所以将请求上下文和相关信息先保存到blockingMap里，
    // 等到之后从kafka里读出响应后，再到blockingMap里找到之前的请求上下文和相关信息，从而返回客户端。
    RequestItem requestItem = new RequestItem(ctx, req, event, ref);
    blockingMap.put(eventId, requestItem);
    // 将请求发送到kafka的request topic，
    // 之后Flink会从kafka中将该请求读取出来并进行处理，并将处理的结果再发送到kafka的response topic。
    kafkaWriter.send(requestTopic, event.toJSONString().getBytes(Charsets.UTF_8));
    // 当请求已经发送到kafka后，就启动一个超时任务，
    // 如果到时候超时设置的时间到了，但是请求对应的响应还没有来，就当超时返回。
    CompletableFuture<Void> timeoutFuture = TimeoutHandler.timeoutAfter(10000, TimeUnit.MILLISECO
    timeoutFuture.thenAcceptAsync(v -> this.timeout(eventId), this.timeoutExecutor);
}
```

上面的代码主要完成了接收 REST 请求，并将请求发送给 Kafka 的过程。由于需要等待返回结果，所以在发送 Kafka 之前，我们还将请求相关信息保存在一个 BlockingMap 里。这样，之后当请求处理结果返回时，就可以从这个 BlockingMap 中取出请求，并将请求处理结果返回给客户端。

上面这个过程中，还有另外两个需要注意的点。

- 一是，BlockingMap 是为了实现反向压力的功能，避免异步系统的 OOM 问题。具体而言，BlockingMap 的实现思路是在将请求添加到内部的 map 前，先进行 map size 的判断。如果发现 map size 已经达到设置的容量上限，就暂停添加，直到内部 map 重新有空位可用为止（BlockingMap 的完整实现可以 参考这里）。
- 二是，每次将请求发送到 Kafka 后，我都会通过 TimeoutHandler.timeoutAfter 方法，启动一个超时任务。如果到时候超时任务设置的时间到了，但是请求对应的响应还没有来的话，这个超时任务就调用其超时回调函数，进行超时处理。这样保证了 BlockingMap 里的请求不管成功或失败，都会在一段时间内被处理并返回给客户端，并且不会造成 BlockingMap 容量耗尽的问题。

另外，在请求接收服务器上，我们还需要实现从 Kafka 中读取请求处理结果，并将请求处理结果返回给客户端的功能。这部分功能是由 KafkaResponseHandler 类实现的，具体代码如下（完整代码请参考这里）：

```
KafkaResponseHandler kafkaReader = new KafkaResponseHandler(
        zookeeperConnect, responseTopic, groupId, 2, responseExecutor) {
    @Override
    public Void process(byte[] body) {
        String jsonString = new String(body, Charsets.UTF_8);
        JSONObject e = JSON.parseObject(jsonString);
        String eventId = e.getString(EVENT_ID_TAG);
        // 从 blockingMap 中取出请求上下文信息
        RequestItem requestItem = blockingMap.remove(eventId);
        try {
            // 将请求处理结果返回给客户端
            sendResponse(requestItem.ctx, OK, RestHelper.genResponse(OK.code(), OK.toString(), e)
        } finally {
            requestItem.ref.release();
        }
        return null;
    }
};
```

在上面的代码中，我们从 Kafka 的 response topic 中读取出请求处理的结果后，用 process() 函数进行了处理。处理过程是从 blockingMap 中根据 eventId 取出请求上下文信息，然后将请求处理结果返回给发起该请求的客户端。

以上就是请求接收服务器的处理逻辑了。接下来是 Flink 进行请求处理的代码（完整代码请参考这里）。

```
public static void main(String[] args) throws Exception {
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
    FlinkKafkaConsumer010<String> myConsumer = createKafkaConsumer();
    DataStream<String> stream = env.addSource(myConsumer);
    DataStream<String> counts = stream
            .map(new MapFunction<String, JSONObject>() {
                @Override
                public JSONObject map(String s) throws Exception {
                    if (StringUtils.isEmpty(s)) {
                        return new JSONObject();
                    }
                    return JSONObject.parseObject(s);
                }
            })
            .map(new MapFunction<JSONObject, String>() {
                @Override
                public String map(JSONObject value) throws Exception {
                    value.put("result", "ok");
                    return JSONObject.toJSONString(value);
                }
            });

    counts.addSink(createKafkaProducer()).name("flink-connectors-kafka").setParallelism(4);
    env.execute("FlinkService");
}
```

在上面的代码中，我们用 Flink 从 Kafka 的 request topic 中取出请求后，用两个 map 函数进行了处理。第一个 map 函数用于将请求解码为 JSON 对象，第二个 map 函数用于在请求 JSON 对象中添加了一个 result 字段，以表示处理结果。最后，调用 addSink 函数，将请求处理结果发送给 Kakfa 的 response topic 中。之后，请求接收服务器的 KafkaResponseHandler 类，就是从 Kafka 的 response topic 读取请求处理结果的。
当然，上面我实现的 Flink DAG 比较简单，只用了两个 map 函数，这里主要是为了演示 Flink 实现微服务的整体过程。你可以根据业务的实际需要，将 Flink DAG 实现的更复杂些。有必要的话，还可以把 KeyedStream 和 Keyed State 都用上。

至此，我们就实现了一个完整的基于 Flink 的微服务系统。

小结

今天，我们从分布式 JVM 的角度，对 Flink 进行了分析。

总的来说，Flink 的计算模式非常像是把单 JVM 进程内的流计算过程扩展到了分布式集群。如果我们不强调“流”这种计算模式，那么完全可以将 Flink 理解为一个分布式的 JVM。执行各个任务的线程相当于是 CPU，而 State 则相当于是内存。

由于 Flink 的 State 可以使用磁盘存储，计算节点也是可以水平扩展，所以理论上 Flink 这个分布式 JVM 的“CPU”和“内存”都是“无限”的。

如果按照分布式 JVM 理解 Flink 的话，可以大大扩展我们对 Flink 的使用场景，而不仅仅将其视为一个只用于流计算的工具而已。比如今天我们就用 Flink 实现了一个微服务系统。当然反过来，当我们用 Flink 来解决问题时，又会发现很多业务场景，其实都是可以用流计算过程来解决的。

所以不得不说，**就像 Unix 哲学“万物皆文件”一样，这种“万物皆流”的思想确实会帮助我们打开解决问题的新大门**。

你在工作中遇到的业务场景，哪些是可以用 Flink 这个分布式 JVM 来实现的呢？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图。