

# 架构师

ARCHITECT



热点 | Hot

重磅开源KSQL

推荐文章 | Article

洞悉流程！微服务与事件协同

Kafka设计解析

AI专区 | AI Area

人工智能(极)简史

人工智能那些事儿

在移动设备上运行神经网络的新技术



# 卷首语

作者 雨多田光

最近，各种共享 XXX 如雨后春笋般出现，大家也不管市场是否真正需要，都一窝蜂地扑上去做。

**这背后，跟风、从众、盲从、随大流**就像助燃剂，不断为其助力，并且从各种泡沫事件来看，这种纯粹的“看到很多人在做就跟着做”的动机，在一个事物升温的过程中往往扮演着最重要的角色。可能你会说“炒作”也是一个原因，但是炒作可以看成是炒作者引导大众跟风。

让我们来举几个例子，看看最近的区块链、AI、微服务架构技术，它们的火爆背后存在的“水分”有多大呢？

- 区块链最近借着ICO重新吸引了大家的眼球，但是ICO本身是一个打擦边球的存在，但它确实是火，区块链技术本身反而没有那么受关注了。人们并不是由于这个技术本身的优势而去关注这个领域，大部分人只是想在这个新兴的，直接可以兑现到“货币”的事物中去投机，想分一杯羹。就像看到别人炒股，哪支股牛跟上去就是了。
- AI是当今互联网的政治正确，不管大小公司，不管什么行业，都想要往这上边去靠。但其中有多少是不切实际的幻想，有多少在踏踏实实地做事，相信就连这些AI项目参与者心里也自有判断。

- 微服务架构，其实几十年前就有这样的思想，因为现在相关技术的成熟，人们意识到“原来服务化还可以这么用啊”，微服务架构变得越来越受追捧，创业公司好像不上个微服务就算不上技术创业似的。然而微服务到底适不适合初创公司，目前业界仍处于争论之中。只是它的现状就是很火爆，往往在人们争论之前，公司就跟风先去实施微服务架构了。

还记得架构师 7 月刊的文章《你不是 Google》吗？适合于 Google 这个量级公司的技术，也许并不适合你。Google 有很多人学，这很容易理解；这就会形成“风”、“流”。重点不在于要不要学 **Google**，而是要不要跟风去学 **Google**，这是你应该慎重思考的问题。同样的东西，这里想再一次告诉你：

You are not Netflix, stop trying to be them!

— Russ Miles

我们要海纳百川，去学习各种知识，但是学习了之后用不用、怎么去用，这些是需要另外思考的问题。但往往这个应该另外去思考的问题就被“看现在主流怎么样，就跟呗”给简单地解决掉了。运气好的话，各方面条件相适应，你就能在目前得到相应的好处，运气不好的话，那就是你跟错了风、随错了流，也许就要感冒，也许就会溺亡。

当一个架构师在做技术选型的时候，先不要看大众的走向，静下来，反思一下自己的需求去做选择。**众人追捧的，可能是明星，也许是传销**。那到底用什么标准去判断这个“风”要不要跟，也许永远找不到。我们本身也并不在技术上站队，只是不停地分享各家的理论与实践；而你要做的，就是**不要停止学习**。

只是简单的几句话，这样的提醒，也许并不能很直接地让人有所触动，但是，在技术这一路上，总要有人去做这样的事，时不时呼吁一下大家，希望它至少可以起到抛砖引玉的作用，让更多更加专业的研究者去带动这个话题。**毕竟，技术的发展，事关重大；大家做出合适的而不是跟风的选择，事关重大。**

# CONTENTS / 目录

## 热点 | Hot

重磅开源 KSQL：用于 Apache Kafka 的流数据 SQL 引擎

## 推荐文章 | Article

洞悉流程！微服务与事件协同

Kafka 设计解析：流式计算的新贵 Kafka Stream

## 观点 | Opinion

软件架构图的艺术

## AI 专区 | AI Area

从金属巨人到深度学习，人工智能（极）简史

人工智能那些事儿

Google 研究人员提出在移动设备上运行神经网络的新技术



## 架构师 2017 年 9 月刊

本期主编 雨多田光

提供反馈 feedback@cn.infoq.com

流程编辑 丁晓昀

商务合作 sales@cn.infoq.com

发行人 霍泰稳

内容合作 editors@cn.infoq.com

# 重磅开源 KSQL：用于 Apache Kafka 的流数据 SQL 引擎

作者 薛命灯



Kafka 的作者 Neha Narkhede 在 Confluent 上发表了一篇博文，介绍了 Kafka 新引入的 KSQL 引擎——一个基于流的 SQL。推出 KSQL 是为了降低流式处理的门槛，为处理 Kafka 数据提供简单而完整的可交互式 SQL 接口。KSQL 目前可以支持多种流式操作，包括聚合(aggregate)、连接(join)、时间窗口(window)、会话(session)，等等。

## 与传统 SQL 的主要区别

KSQL 与关系型数据库中的 SQL 还是有很多不同的。传统的 SQL 都是即时的一次性操作，不管是查询还是更新都是在当前的数据集上进行。而 KSQL 则不同，KSQL 的查询和更新是持续进行的，而且数据集可以源源不断地增加。KSQL 所做的其实是转换操作，也就是流式处理。

## KSQL 的适用场景

### 1. 实时监控

一方面，可以通过 KSQL 自定义业务层面的度量指标，这些指标可以实时获得。底层的度量指标无法告诉我们应用程序的实际行为，所以基于应用程序生成的原始事件来自定义度量指标可以更好地了解应用程序的运行状况。另一方面，可以通过 KSQL 为应用程序定义某种标准，用于检查应用程序在生产环境中的行为是否达到预期。

### 2. 安全检测

KSQL 把事件流转换成包含数值的时间序列数据，然后通过可视化工具把这些数据展示在 UI 上，这样就可以检测到很多威胁安全的行为，比如欺诈、入侵，等等。KSQL 为此提供了一种实时、简单而完备的方案。

### 3. 在线数据集成

大部分的数据处理都会经历 ETL (Extract——Transform——Load) 这样的过程，而这样的系统通常都是通过定时的批次作业来完成数据处理的，但批次作业所带来的延时很多时候是无法被接受的。而通过使用 KSQL 和 Kafka 连接器，可以将批次数据集成转变成在线数据集成。比如，通过流与表的连接，可以用存储在数据表里的元数据来填充事件流里的数据，或者在将数据传输到其他系统之前过滤掉数据里的敏感信息。

### 4. 应用开发

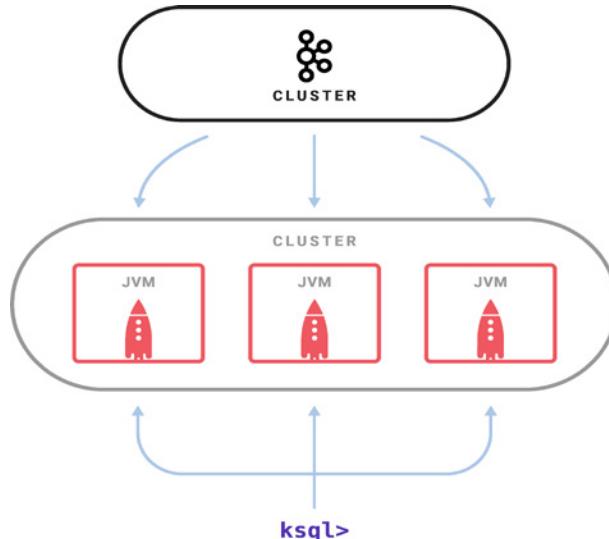
对于复杂的应用来说，使用 Kafka 的原生 Streams API 或许会更合适。不过，对于简单的应用来说，或者对于不喜欢 Java 编程的人来说，KSQL 会是更好的选择。

## KSQL 的核心抽象

KSQL 是基于 Kafka 的 Streams API 进行构建的，所以它的两个核心概念是流 (Stream) 和表 (Table)。流是没有边界的结构化数据，数据

可以被源源不断地添加到流当中，但流中已有的数据是不会发生变化的，即不会被修改也不会被删除。表就是流的视图，或者说它代表了可变数据的集合。它与传统的数据库表类似，只不过具备了一些流式语义，比如时间窗口，而且表中的数据是可变的。KSQL 将流和表集成在一起，允许将代表当前状态的表与代表当前发生事件的流连接在一起。

## KSQL 架构



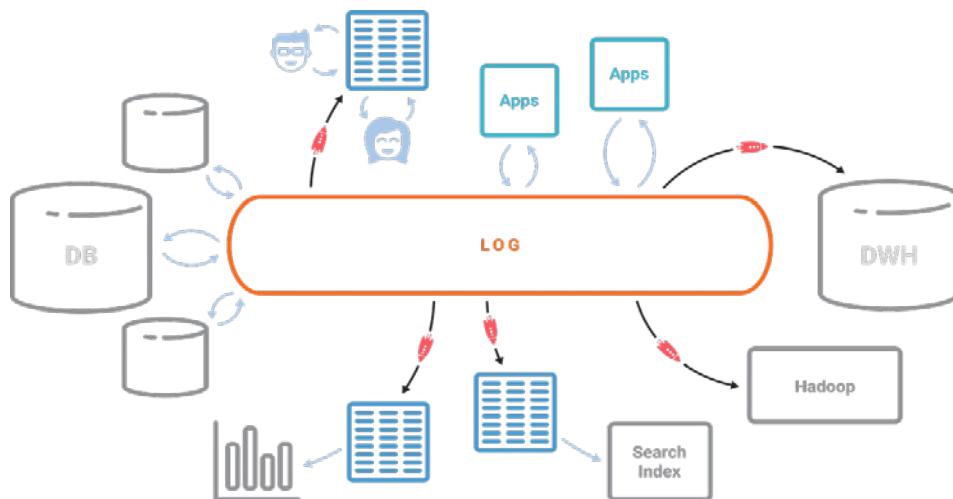
KSQL 是一个独立运行的服务器，多个 KSQL 服务器可以组成集群，可以动态地添加服务器实例。集群具有容错机制，如果一个服务器失效，其他服务器就会接管它的工作。KSQL 命令行客户端通过 REST API 向集群发起查询操作，可以查看流和表的信息、查询数据以及查看查询状态。因为是基于 Streams API 构建的，所以 KSQL 也沿袭了 Streams API 的弹性、状态管理和容错能力，同时也具备了仅一次（exactly once）语义。KSQL 服务器内嵌了这些特性，并增加了一个分布式 SQL 引擎、用于提升查询性能的自动字节码生成机制，以及用于执行查询和管理的 REST API。

## Kafka+KSQL 要颠覆传统数据库

传统关系型数据库以表为核心，日志只不过是实现手段。而在以事件为中心的世界里，情况却恰好相反。日志成为了核心，而表几乎是以日

志为基础，新的事件不断被添加到日志里，表的状态也因此发生变化。将 Kafka 作为中心日志，配置 KSQL 这个引擎，我们就可以创建出我们想要的物化视图，而且视图也会持续不断地得到更新。

## KSQL 的未来



KSQL 目前还处于开发者预览阶段，作者还在收集社区的反馈。未来计划增加更多的特性，包括支持更丰富的 SQL 语法，让 KSQL 成为生产就绪的系统。

这里有 KSQL 的快速入门指南和一个演示程序。可以在 Slack 的 #KSQL 频道上向作者提供反馈信息，或者如果发现 Bug，可以在 GitHub 上提出来。

# 洞悉流程！微服务与事件协同

作者 Bernd Rücker 等 译者 张卫滨



假设我们想要设计一个微服务架构实现一些相对复杂的“端对端”用例，例如，基于网络的零售订单履行。显然，这会涉及到多个微服务。这样的话，我们就需要管理跨越服务边界的业务过程。一旦涉及到解耦微服务，这种跨服务流的场景就会带来很多挑战。在本文中，我们会介绍名为“事件命令转换”（event command transformation）的模式，并提供技术手段来解决跨微服务进行流程编码所带来的复杂性，在这个过程中不会引入任何的中央控制器。

首先，我们需要提供一个初始化的微服务环境，并定义微服务的边界和范围。我们的目标是把不同服务之间的耦合最小化，并让它们保持独立部署。这样做是因为我们想最大化团队的自主权；对于每一个微服务，都

应该有一个跨功能的团队来负责。由于这对我们尤其重要，我们决定采用一种更为粗粒度的方法，围绕业务功能构建一些自包含的服务。因此，就形成了以下的微服务：

- Payment Service（支付服务）： 团队负责处理跟“钱”有关的一切事务；
- Inventory Service（库存服务）： 团队负责管理库存物品；
- Shipment Service（送货服务）： 团队负责“把东西送到客户处”。

网上商店本身可能是由更多微服务构成的，例如：搜索、目录等等。因为我们专注于订单履行，我们只对网上商店中跟客户下订单相关的服务感兴趣：

- Checkout Service（结账服务）： 团队负责客户购物车的结账业务。

这个服务将最终触发订单履行服务的启动。

## 长周期运行的流

我们必须考虑整个订单履行服务的一个重要特征：这是一个长周期运行的流（long running flow），其中包含了要执行的操作。关于“长周期运行”，我们指的是它可能需要运行几分钟、几小时，甚至几个星期，直到订单处理过程结束。

请考虑下面的这个例子：在支付过程中，如果发生信用卡被拒的情况，顾客有一周的时间来提供新的付款明细。这意味着订单也许要等上一个星期。这种长周期运行的行为会给实现方式带来新的需求，我们会在本文中详细讨论这种情况。

## 事件协作

在本文中不会讨论各种通信模式的利弊，在我们的服务之间，将会采用以事件为中心的通信模式来阐述我们的主题。

事件协作 (event collaboration) 理念的核心是，如果微服务的内部发生了业务相关的事情，它们将会发布事件。其他的服务可能会订阅这个事件，并对其产生一定的响应，比如将相关的信息按照最适合其目的的方式来进行存储。这样，在以后的某个时间点，进行订阅的微服务就能使用该信息执行自己的服务，而无需调用其他的服务。由此，通过事件协作，服务之间的高度解耦就水到渠成了。另外，在微服务架构中，我们总是希望能够达到数据管理的去中心化，借助这种方式实现起来会更加容易和自然。

领域驱动设计 (Domain Driven Design) 很好地理解了这个概念，随着微服务的流行和分布式系统中的交互成为“新常态” (new normal)，这种范式的发展变得越来越迅速。

需要注意的是，事件协作可以通过异步消息的方式来实现，也可以通过其他的方式来实现。比如，微服务可以发布基于 REST 的 feed，其中包含它们的事件，这些 feed 可以被其他的服务定期消费。

## 事件命令转换

我们的订单履行从 Order Placed 事件开始。在我们的订单履行服务中，第一件必须要发生的事情就是顾客付款。支付服务成功完成之后，就会发布 Payment Received 事件，然后我们要从库存中发货 (Goods Fetched) 并交付给顾客 (Goods Shipped)。这样的话，就形成了一个清晰的事件流，至少在“乐观”的场景下如此。我们可以很容易地创建一个事件链，如图 1 所示。

在支持了事件协作的基本理念之后，这些事件链提供了一个次优的方式来实现整个业务流程的端到端逻辑。我们看到了它降低耦合的崇高目标，但是这些方案可能会适得其反，反而会增加耦合。让我们深入探讨一下。

根据定义，事件会通知我们发生了相关的事情，其他服务可能会对此感兴趣。但是，接收到事件后，我们需要有个服务来应对事件，在使用事件时，我们将其视为具备命令的语义。这样的结果就是产生了不必要的更

紧密的耦合。

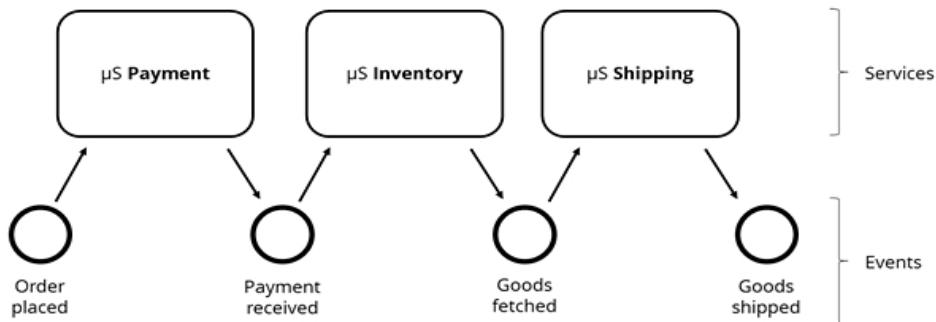


图 1 在链中，每个微服务都在监听前一个服务

在我们的样例中，支付服务监听结账服务的 Order Placed 事件。支付服务必须要知道结账的相关信息，但是如果能够不这样做的话会更好，原因如下：

- 假设我们的组织因为各种原因都需要支付服务，它不仅用于零售订单中。当我们想把支付服务绑定到新事件时，支付服务就需要调整或重新部署，尽管完成支付的细节根本不会改变。
- 假如要完成一些简单的业务需求，比如改变某些步骤的顺序。如果业务上需要确保用户在支付之前，所有的商品都要准备好，那么这三个服务需要同时做出变更：Payment 现在要监听 Goods Fetched，而 Inventory 要监听 Order Placed，shipment 则要订阅 Payment Received。
- 假设我们要为特殊的订单开具发票，比如 VIP 的顾客。现在，支付服务不仅要理解规则，这样的话才能决定下完订单之后是否需要马上完成支付，同时库存服务需要明白，如果只监听 Payment Received 事件的话，将会导致 VIP 用户的整体流程出现停顿！

因此，我们推荐采用事件命令转换 (event command transformation) 模式。确保负责进行业务决策 (这里就是所需的支付) 的组件将事件 (Order Placed) 转换为命令 (Retrieve Payment)。这个命令可以发送给接收服务 (支付)，服务不需要知道客户端的情况，也不会产生上述的缺点。

需要注意，对于长周期运行的流，“生成命令（issuing command）”并不一定必须采用面向请求 / 答复（request/reply）的协议，也可以通过其他的方式来实现。微服务按照类似的方式监听异步命令消息，就像监听事件一样。另外，当事件订阅者将事件转换为内部命令时，也会发生事件命令转换。在这个过程中，会有一个参与者负责决定“接下来需要发生什么事情”，我们建议由它来进行这种转换。

但是，在我们的样例中，这个参与者是谁呢？结账服务（checkout service）应该发出 Retrieve Payment 命令吗？并非如此。让我们再考虑一下上面的场景变化。所有这些都表明我们需要一个单独的微服务来处理订单履行的某些端到端的逻辑。

- Order：该团队负责处理客户所面临的业务核心功能中的端到端逻辑，即订单履行。

该服务执行事件命令转换。它把 Order Placed 转换为 Retrieve Payment。它会针对非 VIP 客户自主地执行该逻辑。它可能需要先咨询另一个确定客户是否为 VIP 的微服务，这个微服务封装了 VIP 客户的构成规则。与之前所描述的纯粹事件协作相比，这样的端到端服务能够在很大程度上提高解耦水平。

但是，引入端到端的服务有可能会形成一种“神一般（God-like）”的服务，这种服务涵盖了大多数的核心业务逻辑，并将功能委托给“贫血的（anemic）”的（CRUD）服务，我们该如何避免这种状况呢？这种做法会损耗掉事件协同（Choreography）的很多收益，很多的作者都不建议采用 God 服务，比如 Sam Newman 在 Building Microservices 中就持这样观点。另外，编排（orchestration）原则被视为实现松耦合的障碍，这是一个采用编排规则的命令服务吗？

## 协同与编排：业务流程的去中心化治理

对于看重团队职责和团队自治的组织来说，避免 God 服务和中心化控制器确实是一个大问题。在高度去中心化的组织中，具有端到端职责的

Order 服务并不意味着我们就要干扰其他的团队（比如支付团队）的职责，而是恰恰相反！订单具有端到端的职责，这样“支付”就变成了一个黑盒。我们只负责要求它履行其职责（Retrieve Payment）并等待它完成：Payment Received。

考虑到前面提到的业务需求，一旦信用卡被拒，顾客有一周时间用以提供新的付款明细。我们能试着在 Order 服务中实现这样的逻辑，但是只有在支付服务提供的命令是非常细粒度的时候才可以这么做。如果支付团队认真对待自己的业务功能和相关的责任，那么就算这比单纯的信用卡扣款花费更多的时间，也要由他们来承担相关的职责。支付团队可以提供一些粗粒度的、可能长周期运行的功能，而不是仅仅是大量细粒度，甚至像 CRUD 这样的功能，从而避免出现 God 服务这种趋势。这种理念如下面的图 2 所示。

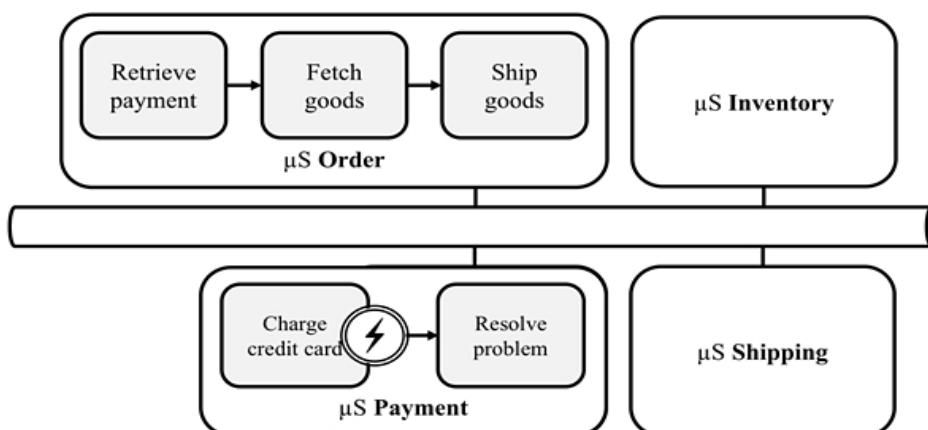


图 2 端到端服务是去中心化治理的，其职责是分布式的

在一个高度去中心化的组织里，端到端的订单服务要尽可能精简，因为在其端到端的处理流程中，大部分内容都会是自治的，有其他专门的服务功能来进行管理。上面的支付服务就是一个这样的例子：支付团队需要实现支付相关的所有事情。

在讨论业务流程实现时，有一个重要的考虑因素和一个常见的误解：这并不意味着我们需要将整个流程设计为一个整体，通过一个中心化的 orchestrator 来执行，就像以前在 SOA 和 BPM 中所作的那样。流程的所

有者和所需的流程逻辑可以是分布式的。它能实现到什么程度主要依赖于组织结构，这种组织结构也会反映到服务的设计中（参见康威定律）。如果按照这种方式的话，我们最终不会形成中心化、单体的控制器。

如果你现在觉得划分端到端的流程逻辑会增加系统的复杂度，那么你的判断也许是正确的。在第一次引入微服务架构时，也存在类似的权衡：单体方式通常更容易，但是随着系统的增长，它会达到一个极限并且也无法再由一个团队来进行处理。流逻辑面临的情况与之类似。

总结一下我们到目前为止所讨论的内容：对于微服务架构来说，协同是基本的模式。我们建议遵循这个模式作为一个重要的经验法则。但是，当涉及业务流程时，不要创建纯粹的事件链，而是实施去中心化的流逻辑，采用事件命令转换模式。负责决定行动的微服务也应该负责把事件转换为命令。

(在本文中，协同对应的英文单词为Choreography，编排对应的英文单词为Orchestration。Choreography本意是舞蹈中的编舞，而Orchestration本意是管弦乐中的编曲。根据维基百科，服务协同指的是一种服务组合的方式，多个服务参与者之间的交互协议要在全局视角来定义，“舞者按照全局的场景来跳舞，没有单点控制”，而在服务编排中，逻辑是通过单个参与者的视角来定义的，被称之为orchestrator。本质上讲，服务协同和服务编排是一个硬币的两个面。服务协同可以通过一个projection过程抽取为服务编排，而已有的服务编排可以组合为协同。关于二者的关联，可以参见该论文。—译者注)

## 流程逻辑的实现

现在，我们看一下长周期运行的流的逻辑实现。长周期运行的流需要将其状态保存起来，因为我们可能等待的时间是不确定的。状态处理并不是什么新鲜的事情，数据库就是为此而生的。所以，一种较为简单的方式就是将订单状态以实体的形式进行存储，如下面的代码片段 1 所示。

### 代码片段 1：简化订单状态，作为某个实体的一部分

```
public class OrderStatus {
    boolean paymentReceived = false;
    boolean goodsFetched = false;
```

```
    boolean goodsShipped = false;  
}
```

我们还可以采用最喜欢的 Actor 框架，在[这篇文章](#)中，我们讨论了基本的可选方案。这些工作都有一定的作用，但是在为长期运行的行为实现状态时，我们很快就会面临额外的需求：我们该如何等待七天的时间？该如何实现错误处理和重试功能？该如何评估订单的处理周期？在什么情况下订单会因为没有支付而自动取消？如果在处理流水线上，在某个地方总是会积压订单，我们该如何更改流程？

这会导致大量的编码，最终会形成一个只能在本地使用的框架。负责相关项目的团队会抱怨他们付出的巨大努力都被埋没了。所以我们要看一种不同的方式：采用已有的框架。在本文中，我们使用来自 Camunda 的开源引擎阐述代码样例。现在，我们看一下代码片段 2：

### 代码片段 2：使用代码来描述订单流程，以 Java 为例

```
engine.getRepositoryService().createDeployment()  
    .addModelInstance(Bpmn.createExecutableProcess("order")  
        .startEvent()  
            .serviceTask().name("Retrieve payment").camundaClass(Ret  
rievePaymentAdapter.class)  
                .serviceTask().name("Fetch goods").camundaClass(FetchGoodsAdapter.class)  
                    .serviceTask().name("Ship goods").camundaClass(ShipGoodsAdapter.class)  
                .endEvent().camundaExecutionListenerClass("end",  
GoodsShippedAdapter.class)  
            .done()  
    ).deploy();
```

引擎现在运行流实例，跟踪它们的状态并将它们以持久化的方式进行存储，从而缓解灾害或长时间等待的影响。缺失的适配器逻辑也很容易编码实现，如代码片段 3 所示。

### 代码片段 3：额外的逻辑可以通过适配器编码实现，以 Java 为例

```

public class RetrievePaymentAdapter implements JavaDelegate {
    public void execute(ExecutionContext ctx) throws Exception {
        // Prepare payload for the outgoing command
        publishCommand("RetrievePayment", payload);
        addEventSubscription("PaymentReceived", ctx);
    }
}

```

这样的引擎也能处理更复杂的需求。以下的流能够捕获信用卡付款时发生的所有错误。这个流以一种可选择的方式来运行，要求用户更新其支付细节。我们并不知道客户何时会完成这件事情，甚至不知道他们是否会这样做，我们只是等待他们传入进来的消息（从技术上来讲，很可能会来自 UI 或其他的微服务）。但是，我们只会等待七天，然后就自动结束这个流并发布一个 Payment Failed 事件，请参照代码片段 4。

#### **代码片段 4：流逻辑允许在一周的时间范围内更新信用卡数据**

```

Bpmn.createExecutableProcess("payment")
    .startEvent()
        .serviceTask().id("charge").name("Charge credit card").camundaClass(ChargeCreditCardAdapter.class)
            .boundaryEvent().error()
                .serviceTask().name("Ask customer to update credit card").camundaClass(AskCustomerAdapter.class)
                    .receiveTask().id("wait").name("Wait for new credit card data").message("CreditCardUpdated")
                        .boundaryEvent().timerWithDuration("PT7D") // time out after 7 days
                            .endEvent().camundaExecutionListenerClass("end", PaymentFailedAdapter.class)
                                .moveToActivity("wait").connectTo("charge") // retry with new credit card data
                                    .moveToActivity("charge")
                                        .endEvent().camundaExecutionListenerClass("end", PaymentCompletedAdapter.class)
                                            .done();

```

我们将在本文后续的内容中介绍一些其他有趣的方面，比如：将这样的流进行可视化。现在，总结一下，我们可以利用这样的状态机来处理状态，并围绕着状态转换定义强大的流。

## 可嵌入的工作流

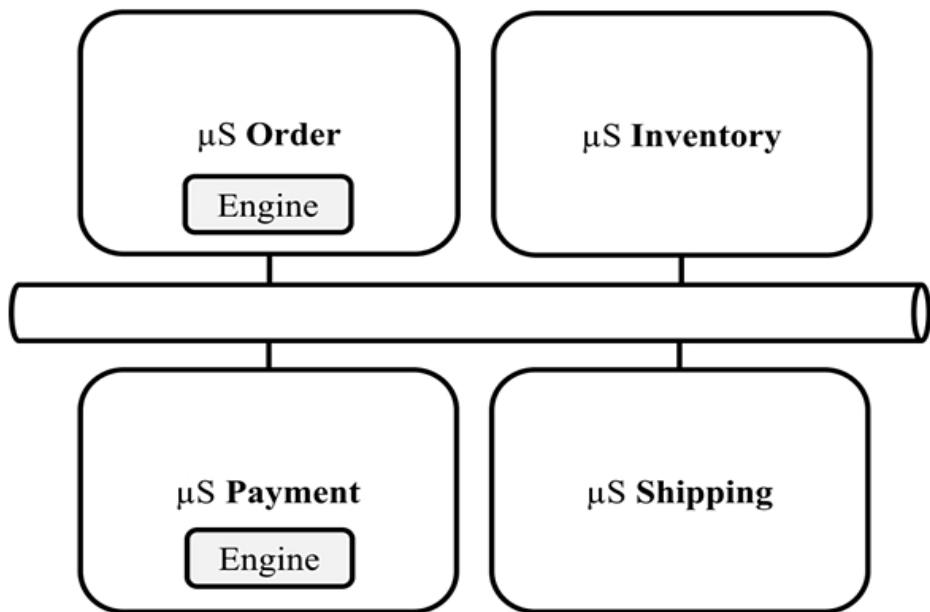
这样的状态机是一个简单的库，可以嵌入到微服务中。在本文提供的源码示例中，可以看到一个如何启动 Camunda 引擎的示例，它会作为使用 Java 所实现的微服务的一部分，它也可以通过 Spring Boot 或类似的框架来实现。

我们需要强调以下内容：实现长周期流的每个微服务必须解决流和状态机相关的需求。那么，每个微服务都应采用像 Camunda 这样的引擎吗？这要由负责该微服务的团队来决定，但是不一定所有的团队都采用这样的决策。在微服务架构中，我们通常会发现与技术选型无关的去中心化治理。某个团队尽可以采用不同的框架，甚至可以对他们的流进行硬编码。他们还可以采用相同框架的不同版本。在我们引入工作流引擎的时候，不一定要涉及中心化的组件。我们明确倡导在微服务环境中不要采用不必要的企业架构标准。

可嵌入不一定意味着我们要自己运行引擎，在微服务多语种的情况下更是如此，因为编程语言并不一定直接适用。那么，我们可以以独立的方式来部署引擎，并与之进行远程通信。这可以通过 REST 来完成，但是更高效的方式也正在出现。在这里，比较重要的一点是，引擎的职责是与拥有微服务的团队联系在一起的，它并不是一个中心化的引擎（参见图 3）。

在倡导的去中心化架构中，我们有多个工作流引擎，其中的每一个只能看到整个流的一部分。这对正确的流程监控提出了新需求，这个问题目前还未得到解决。但是根据产品的不同，可能会有变通方案，你也可以在微服务的整体范围中采用已有的监控工具（例如 Elastic stack）。它会引入一个人工交易 id 或跟踪 id，我们在链中的每个服务调用时都会传递这个 id。我们计划写篇博文专门讨论这个话题，因为在更复杂的微服务

协作运行环境中，这是尤其重要的。

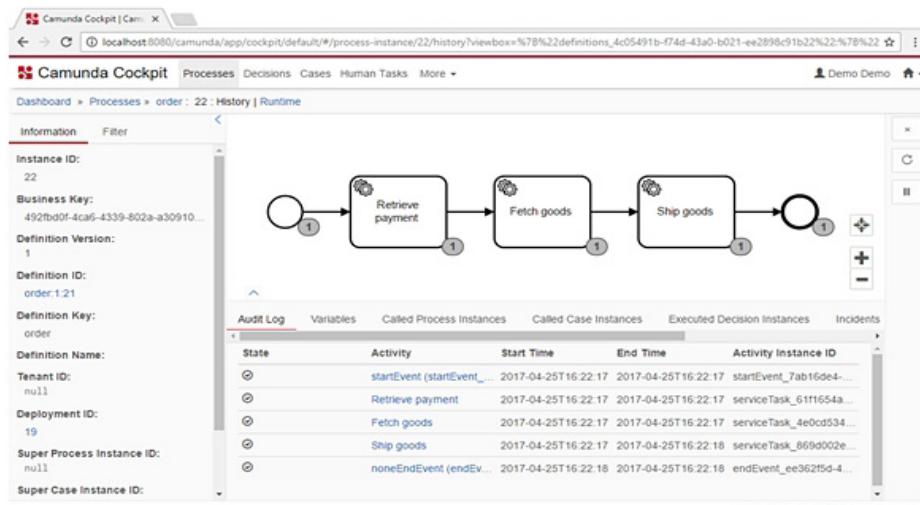


**图 3：团队决定其流逻辑是否需要按照去中心化的方式使用并嵌入引擎**

## 图形可视化的力量

“我热爱代码，我热爱 DSL，图形化 UI 却糟糕透顶”——在跟开发人员交谈时，我们经常会听到这样的说法。这是可以理解的，因为图形化模式通常会阻碍开发人员的工作方式，这被称之为“死于属性面板（death-by-properties-panel）”。这些模型可能还会隐藏幕后的复杂配置。但是编程人员的这种厌恶无法掩盖一个重要的事实：在运维过程中，图形化表述是极其便利的，因为我们不必深入代码以了解目前的状态或异常状况。在与业务利益相关方、需求工程师、运维人员或其他开发人员讨论模型时，我们可以采用图形化的方式。在简短的 workshp 会议上，当我们讨论并建模完一个流程（按照图形化的方式）后，会听到这样的评论：“我终于弄明白这么多年我们都做了些什么！”可视化也有助于对流进行变更，因为我们知道它当前是怎样实现的（请不要忘记，流是正在运行的代码），我们也可以很容易地指出哪里需要变更。

有了工作流引擎，我们就可以获得流的图形化表示。但是，我们通常会注意到这里忽略了一个很重要的方面：流不仅可以按照图形化的方式来定义，也可以按照代码或上述简单 DSL 的方式来定义。我们上面所给出的样例可以按照图形化的方式自动布局和监控，如下图所示。我们知道很多项目使用图形化的模式，因为这样更易于理解。如果是包含并行路径的复杂流，图形化的这一优势会更加突出，以代码的方式难以理解，但是以图形化的方式就一目了然了。图形化模式通常会直接存储为 BPMN 2.0 标准，但是我们了解到有些项目采用 DSL 编码也非常成功。



**图 4 图形化的力量：从业务用户到开发人员，再到运维人员**

在构建端到端监控方案时，对于我们代码所展示的图形化流，采用像 [bpnn.io](#) 这样的轻量级 JavaScript 框架就能很容易地进行可视化。我们只需要通过 API 从不同的引擎读取流程模型及其当前的状态，并展现所有人工交易 id 的运行实例。

在监控中，所显示的流粒度应该能够反映我们之前所讨论的事件协作，它们所对应的事件对域专家是有意义的。这样的话，这些流程对于项目的各种参与者都是可读的。流实际上应该被视为域逻辑的一部分，并以 DDD 所推动的[通用语言](#) (ubiquitous language) 为中心。那么“我们到底该何时付款呢？”，这个问题就非常简单了，不管是业务用户，还是开发人员，甚至运维人员，都能很轻松地回答这个问题。

## 处理复杂流的需求

众所周知，细节是魔鬼。只要离开单个微服务的舒适区，就没有了原子性的事务，我们需要经历延迟和“最终一致性”，必须要和可能不可靠的合作者进行远程通信。于是，开发人员必须处理大量的故障，还要处理无法通过原子性事务实现的业务交易。

对于这些用例，在工作流引擎中有很多强大的工具，在引入 BPMN 工具后更是如此。在图 5 中，我们给出了一个例子，这次利用了图形化的格式。我们捕获到商品缺货的错误，并触发一个所谓的补偿机制(compensation)。引擎的补偿机制知道过去成功执行了哪些操作，然后执行预先定义好的补偿活动，在本例中，也就是退款 Refund payment。我们可以采用这个功能，它很好地实现了所谓的 [Saga 模式](#)。

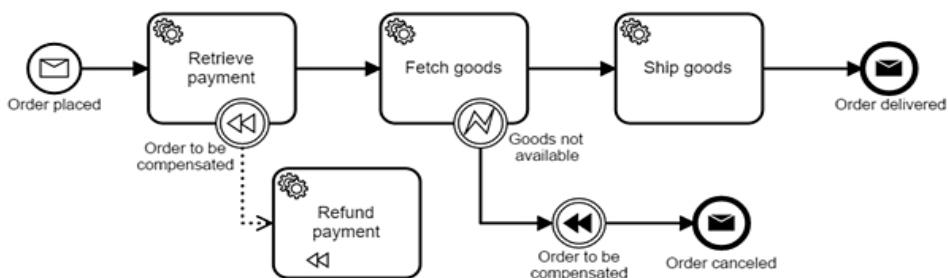


图 5 在出现商品缺货时，已支付的款项会进行退款

请注意，逻辑仍然存在于服务（可能非常精简）中，也就是 Order Service 中，整个流的其他部分将由对其负责的团队维护。无需任何中心化控制器，流逻辑是分布式的。

## 对于微服务来说，为什么状态机没有价值了呢？

在现有的工具中，能够为长时间运行的服务提供流逻辑功能的工具通常被称为工作流或 BPM 引擎。但是，在“过去的 SOA 时代”，围绕业务流程管理 (Business Process Management，简称 BPM) 出现过一些错误，尤其是在开发人员中留下了坏名声。他们认为得到的是一个僵化的、单体

的、对开发人员不友好且昂贵的工具，这个工具强制他们遵循一些模型驱动的、私有的、零编码的方式。有些 BPM 厂商所提供的平台的确在微服务领域中无法使用。但是，需要指出的是，有些轻量级的开源引擎提供了易于使用的、嵌入式的状态机，就像前面看到的那样。我们可以使用这些工具来处理流，而不必要重新发明轮子，这能够节省你的时间，我们都知道时间是非常昂贵的。

消除误解的一个重要方面是认真对待术语。我们在这里展示的流不一定是“业务流程”，如果你“只是”想要一组互相协作的微服务来形成业务交易，这些流可能也不是“工作流”，因为它通常会涉及到人类的一些手动工作。这就是为什么我们常常只谈“流”，这对于不同的用例和不同的利益相关者都适用。

## 样例代码

在这里展示的用例不仅仅是纯粹的理论。为了使概念具体化和易于解释，我们开发了由多个微服务组成的订单履行示例系统，[源码](#)可以在 GitHub 上找到。

## 结论

微服务和事件驱动架构配合得很好。事件协同能够实现去中心化的数据管理，这通常会减少耦合，能够很好地支持本文所关注的长周期“后台”运行的进程。

支持长周期业务流程的大多数端到端的流逻辑应该分布于整个微服务中。每个微服务根据自身的业务功能，实现它所负责那部分流功能。我们建议将事件转换为命令，这种转换需要在负责业务决策的服务中进行，在这种服务中需要一定的输入，进而做出决策。负责剩余端到端逻辑的服务要尽可能精简，但是按照我们的想法最好有一个事件链，而不是多个不透明且紧耦合的事件链。

为了实现流程，我们可以利用现有的轻量级状态机和工作流引擎。这

些引擎能够嵌入微服务中，避免了中心化的工具或治理。我们可以把它们视为帮助开发人员的库。作为额外的奖励，我们还能得到流的图形化展示，这种展现形式在整个项目中都会给我们提供帮助。也许在你所在的公司里，需要克服某些关于工作流或 BPM 的常见误解，但是请相信我们，这是很值得的。

## 作者简介

**Bernd Rücker** 帮助过很多客户依照长周期流实现业务逻辑，例如，帮助快速成长的初创企业 Zalando 创建订单履行流程，实现在世界范围内出售服装，还在一些大型的通信企业中实现 SIM 卡的供应流程。在这个过程中，他为各种开源项目做出了贡献，还写了两本书。他也是 Camunda 的联合创始人。目前，他在思考如何在下一代架构中实施流。

**Martin Schimak** 研究长周期流已经有 15 年之久，涉及能源交易、风洞（wind tunnel）组织以及电信企业的合同管理等不同领域。作为一个程序员，他比较热衷可读的 API 和可测试的规范，并且在 GitHub 上做了很多的贡献。在领域设计方面，因为其优秀的工作被誉为“解码器（decoder）”，他是领域驱动设计以及 BPMN、DMN、CMMN 的领军人物。他还是德语软件杂志 OBJEKTSPEKTRUM 的联合主编。

# Kafka 设计解析：流式计算的新贵 Kafka Stream

作者 郭俊



## Kafka Stream 背景

### Kafka Stream是什么

Kafka Stream 是 Apache Kafka 从 0.10 版本引入的一个新 Feature。它是提供了对存储于 Kafka 内的数据进行流式处理和分析的功能。

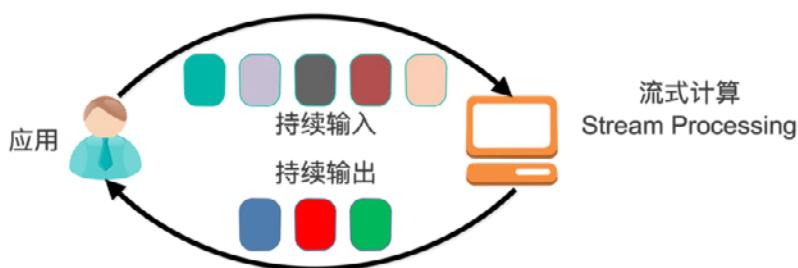
Kafka Stream 的特点如下：

- Kafka Stream 提供了一个非常简单而轻量的 Library，它可以非常方便地嵌入任意 Java 应用中，也可以任意方式打包和部署；
- 除了 Kafka 外，无任何外部依赖；
- 充分利用 Kafka 分区机制实现水平扩展和顺序性保证；

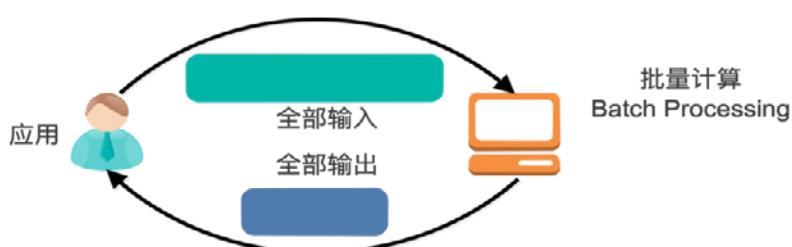
- 通过可容错的state store实现高效的状态操作（如windowed join和aggregation）；
- 支持正好一次处理语义；
- 提供记录级的处理能力，从而实现毫秒级的低延迟；
- 支持基于事件时间的窗口操作，并且可处理迟到的数据；
- 同时提供底层的处理原语Processor（类似于Storm的spout和bolt），以及高层抽象的DSL（类似于Spark的map/group/reduce）；

## 什么是流式计算

一般流式计算会与批量计算相比较。在流式计算模型中，输入是持续的，可以认为在时间上是无界的，也就意味着，永远拿不到全量数据去做计算。同时，计算结果是持续输出的，也即计算结果在时间上也是无界的。流式计算一般对实时性要求较高，同时一般是先定义目标计算，然后数据到来之后将计算逻辑应用于数据。同时为了提高计算效率，往往尽可能采用增量计算代替全量计算。



批量处理模型中，一般先有全量数据集，然后定义计算逻辑，并将计算应用于全量数据。特点是全量计算，并且计算结果一次性全量输出。

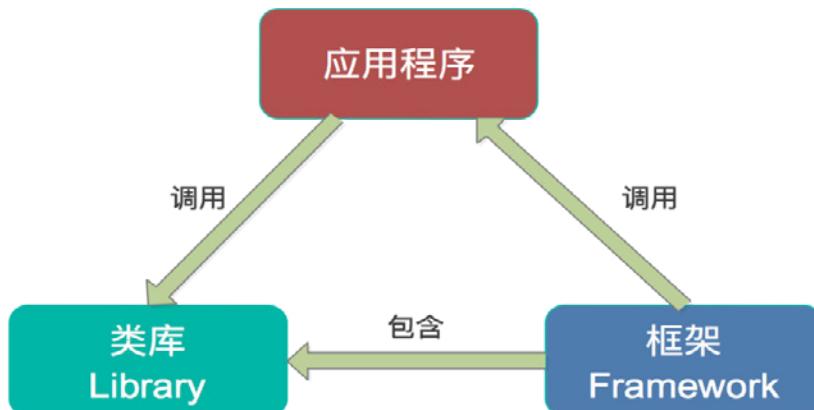


## 为什么要有 Kafka Stream

当前已经有非常多的流式处理系统，最知名且应用最多的开源流式处理系统有 Spark Streaming 和 Apache Storm。Apache Storm 发展多年，应用广泛，提供记录级别的处理能力，当前也支持 SQL on Stream。而 Spark Streaming 基于 Apache Spark，可以非常方便与图计算，SQL 处理等集成，功能强大，对于熟悉其它 Spark 应用开发的用户而言使用门槛低。另外，目前主流的 Hadoop 发行版，如 MapR，Cloudera 和 Hortonworks，都集成了 Apache Storm 和 Apache Spark，使得部署更容易。

既然 Apache Spark 与 Apache Storm 拥有如此多的优势，那为何还需要 Kafka Stream 呢？笔者认为主要有如下原因。

第一，Spark 和 Storm 都是流式处理框架，而 Kafka Stream 提供的是一个基于 Kafka 的流式处理类库。框架要求开发者按照特定的方式去开发逻辑部分，供框架调用。开发者很难了解框架的具体运行方式，从而使得调试成本高，并且使用受限。而 Kafka Stream 作为流式处理类库，直接提供具体的类给开发者调用，整个应用的运行方式主要由开发者控制，方便使用和调试。



第二，虽然 Cloudera 与 Hortonworks 方便了 Storm 和 Spark 的部署，但是这些框架的部署仍然相对复杂。而 Kafka Stream 作为类库，可以非常方便的嵌入应用程序中，它对应用的打包和部署基本没有任何要求。更

为重要的是，Kafka Stream 充分利用了 Kafka 的分区机制和 Consumer 的 Rebalance 机制，使得 Kafka Stream 可以非常方便的水平扩展，并且各个实例可以使用不同的部署方式。具体来说，每个运行 Kafka Stream 的应用程序实例都包含了 Kafka Consumer 实例，多个同一应用的实例之间并行处理数据集。而不同实例之间的部署方式并不要求一致，比如部分实例可以运行在 Web 容器中，部分实例可运行在 Docker 或 Kubernetes 中。

第三，就流式处理系统而言，基本都支持 Kafka 作为数据源。例如 Storm 具有专门的 kafka-spout，而 Spark 也提供专门的 spark-streaming-kafka 模块。事实上，Kafka 基本上是主流的流式处理系统的标准数据源。换言之，大部分流式系统中都已部署了 Kafka，此时使用 Kafka Stream 的成本非常低。

第四，使用 Storm 或 Spark Streaming 时，需要为框架本身的进程预留资源，如 Storm 的 supervisor 和 Spark on YARN 的 node manager。即使对于应用实例而言，框架本身也会占用部分资源，如 Spark Streaming 需要为 shuffle 和 storage 预留内存。

第五，由于 Kafka 本身提供数据持久化，因此 Kafka Stream 提供滚动部署和滚动升级以及重新计算的能力。

第六，由于 Kafka Consumer Rebalance 机制，Kafka Stream 可以在线动态调整并行度。

## Kafka Stream 架构

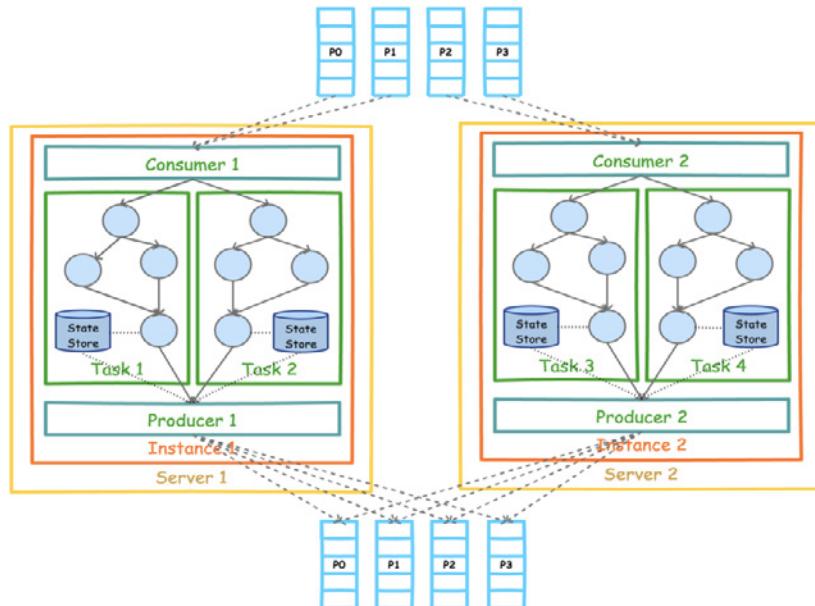
### Kafka Stream 整体架构

Kafka Stream 的整体架构图如下。

目前（Kafka 0.11.0.0）Kafka Stream 的数据源只能如上图所示是 Kafka。但是处理结果并不一定要如图所示输出到 Kafka。实际上 KStream 和 Ktable 的实例化都需要指定 Topic。

```
KStream<String, String> stream = builder.stream("words-stream");
```

```
KTable<String, String> table = builder.table("words-table",
"words-store");
```



另外，上图中的 Consumer 和 Producer 并不需要开发者在应用中显示实例化，而是由 Kafka Stream 根据参数隐式实例化和管理，从而降低了使用门槛。开发者只需要专注于开发核心业务逻辑，也即上图中 Task 内的部分。

## Processor Topology

基于 Kafka Stream 的流式应用的业务逻辑全部通过一个被称为 Processor Topology 的地方执行。它与 Storm 的 Topology 和 Spark 的 DAG 类似，都定义了数据在各个处理单元（在 Kafka Stream 中被称作 Processor）间的流动方式，或者说定义了数据的处理逻辑。

下面是一个 Processor 的示例，它实现了 Word Count 功能，并且每秒输出一次结果。

```
public class WordCountProcessor implements Processor<String,
String> {
    private ProcessorContext context;
    private KeyValueStore<String, Integer> kvStore;
    @SuppressWarnings("unchecked")
```

```
@Override
public void init(ProcessorContext context) {
    this.context = context;
    this.context.schedule(1000);
    this.kvStore = (KeyValueStore<String, Integer>) context.
getStateStore("Counts");
}

@Override
public void process(String key, String value) {
    Stream.of(value.toLowerCase().split(" ")).forEach((String
word) -> {
        Optional<Integer> counts = Optional.ofNullable(kvStore.
get(word));
        int count = counts.map(wordcount -> wordcount +
1).orElse(1);
        kvStore.put(word, count);
    });
}

@Override
public void punctuate(long timestamp) {
    KeyValueIterator<String, Integer> iterator = this.kvStore.
all();
    iterator.forEachRemaining(entry -> {
        context.forward(entry.key, entry.value);
        this.kvStore.delete(entry.key);
    });
    context.commit();
}

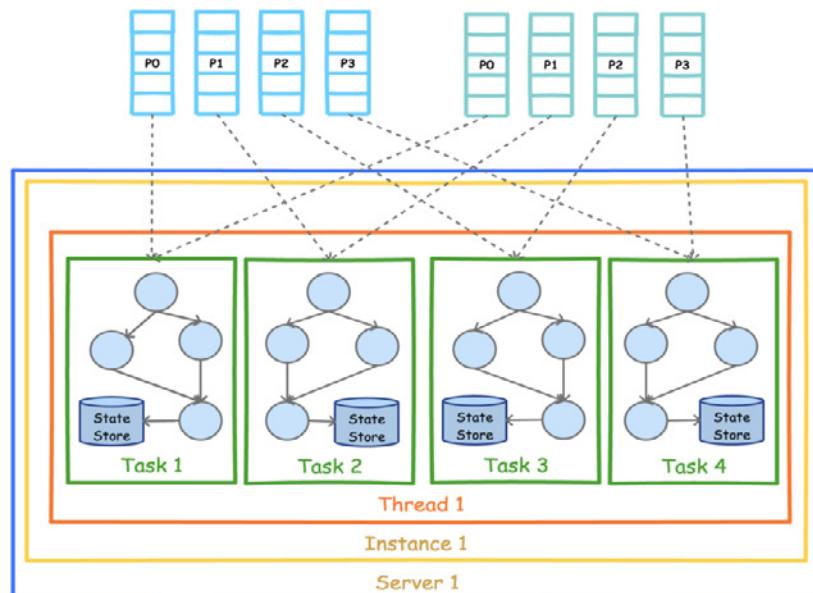
@Override
public void close() {
    this.kvStore.close(); }}
```

从上述代码中可见：

- process 定义了对每条记录的处理逻辑，也印证了 Kafka 可具有记录级的数据处理能力；
- context.scheduler 定义了 punctuate 被执行的周期，从而提供了实现窗口操作的能力；
- context.getStateStore 提供的状态存储为有状态计算（如窗口，聚合）提供了可能。

## Kafka Stream 并行模型

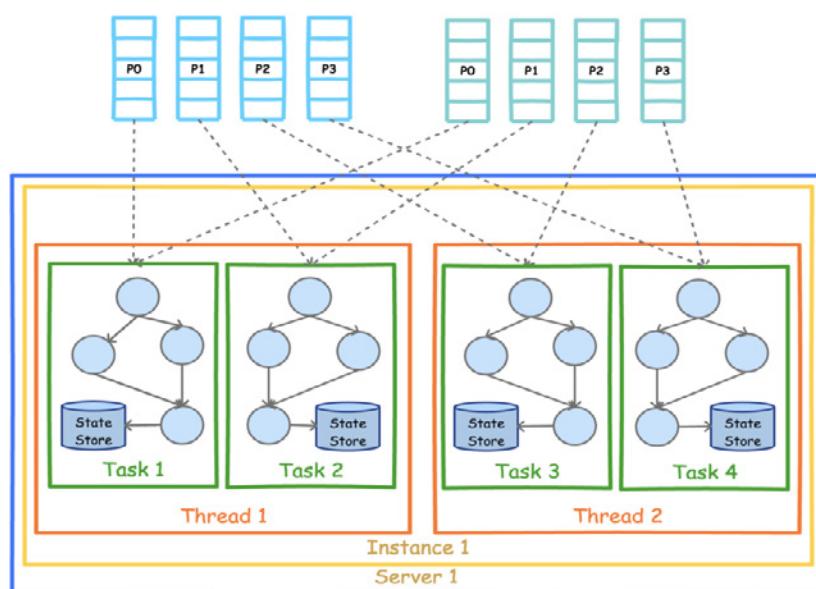
Kafka Stream 的并行模型中，最小粒度为 Task，而每个 Task 包含一个特定子 Topology 的所有 Processor。因此每个 Task 所执行的代码完全一样，唯一的不同在于所处理的数据集互补。这一点跟 Storm 的 Topology 完全不一样。Storm 的 Topology 的每一个 Task 只包含一个 Spout 或 Bolt 的实例。因此 Storm 的一个 Topology 内的不同 Task 之间需要通过网络通信传递数据，而 Kafka Stream 的 Task 包含了完整的子 Topology，所以 Task 之间不需要传递数据，也就不需要网络通信。这一点降低了系统复杂度，也提高了处理效率。



如果某个 Stream 的输入 Topic 有多个（比如 2 个 Topic，1 个 Partition 数为 4，另一个 Partition 数为 3），则总的 Task 数等于 Partition 数最多的那个 Topic 的 Partition 数 ( $\max(4, 3)=4$ )。这是因为 Kafka Stream 使用了 Consumer 的 Rebalance 机制，每个 Partition 对应一个 Task。

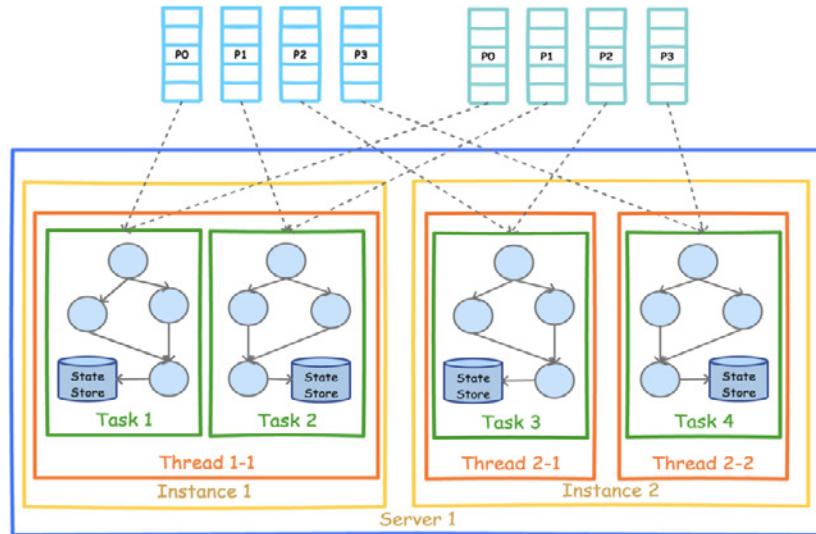
上图展示了在一个进程（Instance）中以 2 个 Topic（Partition 数均为 4）为数据源的 Kafka Stream 应用的并行模型。从图中可以看到，由于 Kafka Stream 应用的默认线程数为 1，所以 4 个 Task 全部在一个线程中运行。

为了充分利用多线程的优势，可以设置 Kafka Stream 的线程数。下图展示了线程数为 2 时的并行模型。

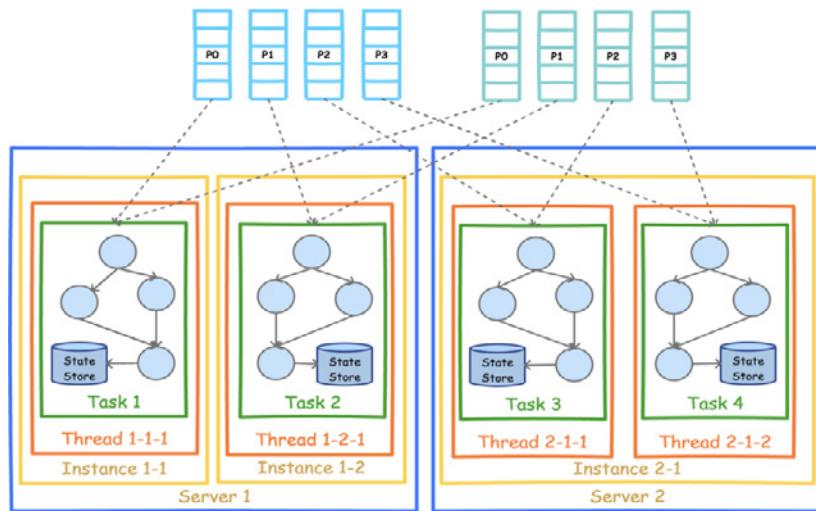


前文有提到，Kafka Stream 可被嵌入任意 Java 应用（理论上基于 JVM 的应用都可以）中，下图展示了在同一台机器的不同进程中同时启动同一 Kafka Stream 应用时的并行模型。注意，这里要保证两个进程的 StreamsConfig.APPLICATION\_ID\_CONFIG 完全一样。因为 Kafka Stream 将 APPLICATION\_ID\_CONFIG 作为隐式启动的 Consumer 的 Group ID。只有保证 APPLICATION\_ID\_CONFIG 相同，才能保证这两个进程的 Consumer 属于

同一个 Group, 从而可以通过 Consumer Rebalance 机制拿到互补的数据集。



既然实现了多进程部署，可以以同样的方式实现多机器部署。该部署方式也要求所有进程的 APPLICATION\_ID\_CONFIG 完全一样。从图上也可以看到，每个实例中的线程数并不要求一样。但是无论如何部署，Task 总数总会保证一致。



注意: Kafka Stream 的并行模型，非常依赖于《Kafka 设计解析（一） – Kafka 背景及架构介绍》一文中介绍的 Kafka 分区机制和《Kafka 设计解析（四） – Kafka Consumer 设计解析》中介绍的 Consumer 的 Rebalance

机制。强烈建议不太熟悉这两种机制的朋友，先行阅读这两篇文章。

这里对比一下 Kafka Stream 的 Processor Topology 与 Storm 的 Topology。

Storm 的 Topology 由 Spout 和 Bolt 组成，Spout 提供数据源，而 Bolt 提供计算和数据导出。Kafka Stream 的 Processor Topology 完全由 Processor 组成，因为它的数据固定由 Kafka 的 Topic 提供。

Storm 的不同 Bolt 运行在不同的 Executor 中，很可能位于不同的机器，需要通过网络通信传输数据。而 Kafka Stream 的 Processor Topology 的不同 Processor 完全运行于同一个 Task 中，也就完全处于同一个线程，无需网络通信。

Storm 的 Topology 可以同时包含 Shuffle 部分和非 Shuffle 部分，并且往往一个 Topology 就是一个完整的应用。而 Kafka Stream 的一个物理 Topology 只包含非 Shuffle 部分，而 Shuffle 部分需要通过 through 操作显示完成，该操作将一个大的 Topology 分成了 2 个子 Topology。

Storm 的 Topology 内，不同 Bolt/Spout 的并行度可以不一样，而 Kafka Stream 的子 Topology 内，所有 Processor 的并行度完全一样。

Storm 的一个 Task 只包含一个 Spout 或者 Bolt 的实例，而 Kafka Stream 的一个 Task 包含了一个子 Topology 的所有 Processor。

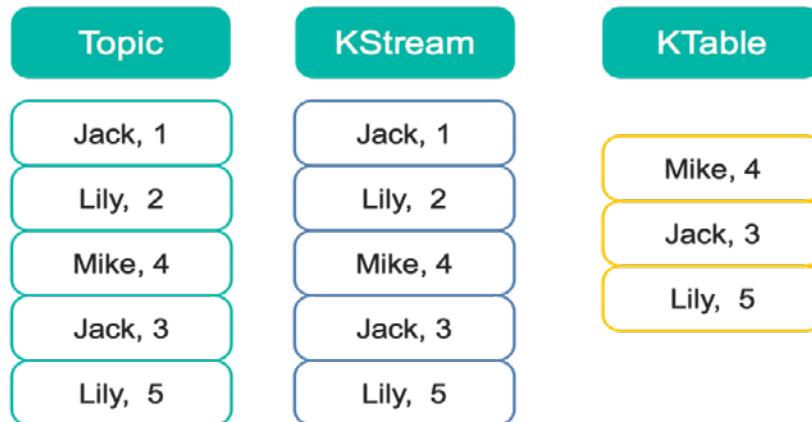
## KTable vs. KStream

KTable 和 KStream 是 Kafka Stream 中非常重要的两个概念，它们是 Kafka 实现各种语义的基础。因此这里有必要分析下二者的区别。

KStream 是一个数据流，可以认为所有记录都通过 Insert only 的方式插入进这个数据流里。而 KTable 代表一个完整的数据集，可以理解为数据库中的表。由于每条记录都是 Key-Value 对，这里可以将 Key 理解为数据库中的 Primary Key，而 Value 可以理解为一行记录。可以认为 KTable 中的数据都是通过 Update only 的方式进入的。也就意味着，如果 KTable 对应的 Topic 中新进入的数据的 Key 已经存在，那么从 KTable

只会取出同一 Key 对应的最后一条数据，相当于新的数据更新了旧的数据。

以下图为例，假设有一个 KStream 和 KTable，基于同一个 Topic 创建，并且该 Topic 中包含如下图所示 5 条数据。此时遍历 KStream 将得到与 Topic 内数据完全一样的所有 5 条数据，且顺序不变。而此时遍历 KTable 时，因为这 5 条记录中有 3 个不同的 Key，所以将得到 3 条记录，每个 Key 对应最新的值，并且这三条数据之间的顺序与原来在 Topic 中的顺序保持一致。这一点与 Kafka 的日志 compact 相同。



此时如果对该 KStream 和 KTable 分别基于 key 做 Group，对 Value 进行 Sum，得到的结果将会不同。对 KStream 的计算结果是 <Jack, 4>, <Lily, 7>, <Mike, 4>。而对 KTable 的计算结果是 <Mike, 4>, <Jack, 3>, <Lily, 5>。

## State store

流式处理中，部分操作是无状态的，例如过滤操作（Kafka Stream DSL 中用 filter 方法实现）。而部分操作是有状态的，需要记录中间状态，如 Window 操作和聚合计算。State store 被用来存储中间状态。它既可以是一个持久化的 Key-Value 存储，也可以是内存中的 HashMap，或者是数据库。Kafka 提供了基于 Topic 的状态存储。

Topic 中存储的数据记录本身是 Key-Value 形式的，同时 Kafka 的 log compaction 机制可对历史数据做 compact 操作，保留每个 Key 对应

的最后一个 Value，从而在保证 Key 不丢失的前提下，减少总数据量，从而提高查询效率。

构造 KTable 时，需要指定其 state store name。默认情况下，该名字也即用于存储该 KTable 的状态的 Topic 的名字，遍历 KTable 的过程，实际就是遍历它对应的 state store，或者说遍历 Topic 的所有 key，并取每个 Key 最新值的过程。为了使得该过程更加高效，默认情况下会对该 Topic 进行 compact 操作。

另外，除了 KTable，所有状态计算，都需要指定 state store name，从而记录中间状态。

## Kafka Stream 如何解决流式系统中关键问题

### 时间

在流式数据处理中，时间是数据的一个非常重要的属性。从 Kafka 0.10 开始，每条记录除了 Key 和 Value 外，还增加了 timestamp 属性。目前 Kafka Stream 支持三种时间

事件发生时间。事件发生的时间，包含在数据记录中。发生时间由 Producer 在构造 ProducerRecord 时指定。并且需要 Broker 或者 Topic 将 message.timestamp.type 设置为 CreateTime（默认值）才能生效。

消息接收时间，也即消息存入 Broker 的时间。当 Broker 或 Topic 将 message.timestamp.type 设置为 LogAppendTime 时生效。此时 Broker 会在接收到消息后，存入磁盘前，将其 timestamp 属性值设置为当前机器时间。一般消息接收时间比较接近于事件发生时间，部分场景下可代替事件发生时间。

消息处理时间，也即 Kafka Stream 处理消息时的时间。

注：Kafka Stream 允许通过实现 org.apache.kafka.streams.processor.TimestampExtractor 接口自定义记录时间。

### 窗口

前文提到，流式数据是在时间上无界的。而聚合操作只能作用在特定的数据集，也即有界的数据集上。因此需要通过某种方式从无界的数 据集上按特定的语义选取有界的数据。窗口是一种非常常用的设定计算边界的方式。不同的流式处理系统支持的窗口类似，但不尽相同。

Kafka Stream 支持的窗口如下。

1. Hopping Time Window 该窗口定义如下图所示。它有两个属性，一个是Window size，一个是Advance interval。Window size 指定了窗口的大小，也即每次计算的数据集的大小。而Advance interval 定义输出的时间间隔。一个典型的应用场景是，每隔5秒钟输出一次过去1个小时内网站的PV或者UV。

● Hopping Time Window



2. Tumbling Time Window 该窗口定义如下图所示。可以认为它是 Hopping Time Window 的一种特例，也即Window size 和 Advance interval 相等。它的特点是各个Window 之间完全不相交。

● Tumbling Time Window



Sliding Window 该窗口只用于 2 个 KStream 进行 Join 计算时。该窗口的大小定义了 Join 两侧 KStream 的数据记录被认为在同一个窗口的最大时间差。假设该窗口的大小为 5 秒，则参与 Join 的 2 个 KStream 中，记录时间差小于 5 的记录被认为在同一个窗口中，可以进行 Join 计算。

Session Window 该窗口用于对 Key 做 Group 后的聚合操作中。它需要对 Key 做分组，然后对组内的数据根据业务需求定义一个窗口的起始点和结束点。一个典型的案例是，希望通过 Session Window 计算某个用户访问网站的时间。对于一个特定的用户（用 Key 表示）而言，当发生登录操作时，该用户（Key）的窗口即开始，当发生退出操作或者超时时，该

用户（Key）的窗口即结束。窗口结束时，可计算该用户的访问时间或者点击次数等。

## Join

Kafka Stream 由于包含 KStream 和 KTable 两种数据集，因此提供如下 Join 计算。

- KTable Join KTable 结果仍为KTable。任意一边有更新，结果 KTable都会更新。
- KStream Join KStream 结果为KStream。必须带窗口操作，否则会造成Join操作一直不结束。
- KStream Join KTable / GlobalKTable 结果为KStream。只有当 KStream中有新数据时，才会触发Join计算并输出结果。KStream 无新数据时，KTable的更新并不会触发Join计算，也不会输出数据。并且该更新只对下次Join生效。一个典型的使用场景是，KStream中的订单信息与KTable中的用户信息做关联计算。

对于 Join 操作，如果要得到正确的计算结果，需要保证参与 Join 的 KTable 或 KStream 中 Key 相同的数据被分配到同一个 Task。具体方法是：

- 参与Join的KTable或KStream的Key类型相同（实际上，业务含义也应该相同）；
- 参与Join的KTable或KStream对应的Topic的Partition数相同；
- Partitioner策略的最终结果等效（实现不需要完全一样，只要效果一样即可），也即Key相同的情况下，被分配到ID相同的 Partition内。

如果上述条件不满足，可通过调用如下方法使得它满足上述条件。

```
KStream<K, V> through(Serde<K> keySerde, Serde<V> valSerde,
    StreamPartitioner<K, V> partitioner, String topic)
```

## 聚合与乱序处理

聚合操作可应用于 KStream 和 KTable。当聚合发生在 KStream 上时

必须指定窗口，从而限定计算的目标数据集。

需要说明的是，聚合操作的结果肯定是 KTable。因为 KTable 是可更新的，可以在晚到的数据到来时（也即发生数据乱序时）更新结果 KTable。

这里举例说明。假设对 KStream 以 5 秒为窗口大小，进行 Tumbling Time Window 上的 Count 操作。并且 KStream 先后出现时间为 1 秒，3 秒，5 秒的数据，此时 5 秒的窗口已达上限，Kafka Stream 关闭该窗口，触发 Count 操作并将结果 3 输出到 KTable 中（假设该结果表示为  $\langle 1-5, 3 \rangle$ ）。若 1 秒后，又收到了时间为 2 秒的记录，由于 1-5 秒的窗口已关闭，若直接抛弃该数据，则可认为之前的结果  $\langle 1-5, 3 \rangle$  不准确。而如果直接将完整的结果  $\langle 1-5, 4 \rangle$  输出到 KStream 中，则 KStream 中将会包含该窗口的 2 条记录， $\langle 1-5, 3 \rangle$ ,  $\langle 1-5, 4 \rangle$ ，也会存在航数据。因此 Kafka Stream 选择将聚合结果存于 KTable 中，此时新的结果  $\langle 1-5, 4 \rangle$  会替代旧的结果  $\langle 1-5, 3 \rangle$ 。用户可得到完整的正确的结果。

这种方式保证了数据准确性，同时也提高了容错性。

但需要说明的是，Kafka Stream 并不会对所有晚到的数据都重新计算并更新结果集，而是让用户设置一个 retention period，将每个窗口的结果集在内存中保留一定时间，该窗口内的数据晚到时，直接合并计算，并更新结果 KTable。超过 retention period 后，该窗口结果将从内存中删除，并且晚到的数据即使落入窗口，也会被直接丢弃。

## 容错

Kafka Stream 从如下几个方面进行容错。

高可用的 Partition 保证无数据丢失。每个 Task 计算一个 Partition，而 Kafka 数据复制机制保证了 Partition 内数据的高可用性，故无数据丢失风险。同时由于数据是持久化的，即使任务失败，依然可以重新计算。

状态存储实现快速故障恢复和从故障点继续处理。对于 Join 和聚合

及窗口等有状态计算，状态存储可保存中间状态。即使发生 Failover 或 Consumer Rebalance，仍然可以通过状态存储恢复中间状态，从而可以继续从 Failover 或 Consumer Rebalance 前的点继续计算。

KTable 与 retention period 提供了对乱序数据的处理能力。

## Kafka Stream 应用示例

下面结合一个案例来讲解如何开发 Kafka Stream 应用。本例完整代码可从作者 Github 获取。

订单 KStream（名为 orderStream），底层 Topic 的 Partition 数为 3，Key 为用户名，Value 包含用户名，商品名，订单时间，数量。用户 KTable（名为 userTable），底层 Topic 的 Partition 数为 3，Key 为用户名，Value 包含性别，地址和年龄。商品 KTable（名为 itemTable），底层 Topic 的 Partition 数为 6，Key 为商品名，价格，种类和产地。现在希望计算每小时购买产地与自己所在地相同的用户总数。

首先由于希望使用订单时间，而它包含在 orderStream 的 Value 中，需要通过提供一个实现 TimestampExtractor 接口的类从 orderStream 对应的 Topic 中抽取出订单时间。

```
public class OrderTimestampExtractor implements
TimestampExtractor {
    @Override
    public long extract(ConsumerRecord<Object, Object> record) {
        if(record instanceof Order) {
            return ((Order)record).getTS();
        } else {
            return 0;
        }
    }
}
```

接着通过将 orderStream 与 userTable 进行 Join，来获取订单用户

所在地。由于二者对应的 Topic 的 Partition 数相同，且 Key 都为用户名，再假设 Producer 往这两个 Topic 写数据时所用的 Partitioner 实现相同，则此时上文所述 Join 条件满足，可直接进行 Join。

```
orderUserStream = orderStream
    .leftJoin(userTable,
        // 该 lambda 表达式定义了如何从 orderStream 与 userTable 生成结果
        // 集的 Value
        (Order order, User user) -> OrderUser.
        fromOrderUser(order, user),
        // 结果集 Key 序列化方式
        Serdes.String(),
        // 结果集 Value 序列化方式
        SerdesFactory.serdeFrom(Order.class))
    .filter((String userName, OrderUser orderUser) -> orderUser.
        userAddress != null)
```

从上述代码中，可以看到，Join 时需要指定如何从参与 Join 双方的记录生成结果记录的 Value。Key 不需要指定，因为结果记录的 Key 与 Join Key 相同，故无须指定。Join 结果存于名为 orderUserStream 的 KStream 中。

接下来需要将 orderUserStream 与 itemTable 进行 Join，从而获取商品产地。此时 orderUserStream 的 Key 仍为用户名，而 itemTable 对应的 Topic 的 Key 为产品名，并且二者的 Partition 数不一样，因此无法直接 Join。此时需要通过 through 方法，对其中一方或双方进行重新分区，使得二者满足 Join 条件。这一过程相当于 Spark 的 Shuffle 过程和 Storm 的 FieldGrouping。

```
orderUserStrea
    .through(
        // Key 的序列化方式
        Serdes.String(),
        // Value 的序列化方式
        SerdesFactory.serdeFrom(OrderUser.class),
```

```
// 重新按照商品名进行分区，具体取商品名的哈希值，然后对分区数取模
        (String key, OrderUser orderUser, int numPartitions)
-> (orderUser.getItemName().hashCode() & 0xFFFFFFFF) %
numPartitions,
        "orderuser-repartition-by-item")
        .leftJoin(itemTable, (OrderUser orderUser, Item item) ->
OrderUserItem.fromOrderUser(orderUser, item), Serdes.String(),
SerdesFactory.serdeFrom(OrderUser.class))
```

从上述代码可见，through 时需要指定 Key 的序列化器，Value 的序列化器，以及分区方式和结果集所在的 Topic。这里要注意，该 Topic (orderuser-repartition-by-item) 的 Partition 数必须与 itemTable 对应 Topic 的 Partition 数相同，并且 through 使用的分区方法必须与 itemTable 对应 Topic 的分区方式一样。经过这种 through 操作，orderUserStream 与 itemTable 满足了 Join 条件，可直接进行 Join。

## 总结

- Kafka Stream 的并行模型完全基于 Kafka 的分区机制和 Rebalance 机制，实现了在线动态调整并行度。
- 同一 Task 包含了一个子 Topology 的所有 Processor，使得所有处理逻辑都在同一线程内完成，避免了不必要的网络通信开销，从而提高了效率。
- through 方法提供了类似 Spark 的 Shuffle 机制，为使用不同分区策略的数据提供了 Join 的可能。
- log compact 提高了基于 Kafka 的 state store 的加载效率。
- state store 为状态计算提供了可能。
- 基于 offset 的计算进度管理以及基于 state store 的中间状态管理为发生 Consumer rebalance 或 Failover 时从断点处继续处理提供了可能，并为系统容错性提供了保障。
- KTable 的引入，使得聚合计算拥有了处理乱序问题的能力。

# 软件架构图的艺术

作者 Ionut Balosin 译者 薛命灯



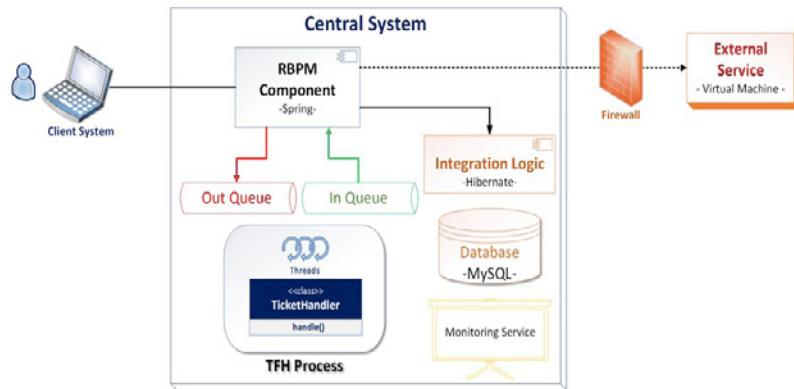
曾几何时，我们的每一个软件项目都需要一个架构图。不管我们是否遵循正式的架构模型（比如 Kruchten 4+1、Rozanski & Woods 等），都有必要通过图表来对应用程序的某些部分进行文档化。在软件架构里，这些图表一般都是按照某些视图进行设计的，这些视图本身就是模型的一部分，不过在这篇文章里，我倾向于使用架构图这个术语，因为它看起来不是那么正式，至于其他方面的内容并不会在这篇文章里涉及到。

作为一个软件架构师和技术培训师，从我的经验来看，不同项目之间以及同一个团队的不同开发人员之间创建架构图的方式也是很不一样的。我看到过很多问题，比如一致性问题、碎片化问题、信息粒度大小的问题，以及图表的外观问题。相比架构模型的正式和标准化，架构图倒是不必要那么正式或者遵循什么标准。

不过，架构图必须是自描述的，并且要具备一致性和足够的准确性，能够与代码相呼应。这也就是为什么架构师或软件工程师在创建架构图时需要依赖各种指导性原则，因为它们是理解应用架构（比如结构、元素、关系、属性、原则）的基石，同时也是具有不同技术背景和关注点的利益相关者的沟通基础。

## 当前架构图的不足之处

在深入展开说明之前，我想引用一句英语谚语：“一张图片胜过千言万语”。Wiki 上解释说，“这句谚语的意思是，一个复杂的想法可以通过一张静态的图片表达出来，或者图片可以表达一个主题的意思，又或者图片比文字描述来得更加直接有效”。对于架构图来说也是一样的：如果它所导致的疑问比它能解释的问题还要多，那么它就不是一张好的架构图。一张好的架构图不需要多余的文字解释！



**图 1 一张不是很好的架构图一般会存在如下几个问题**

现在让我们来过一下不好的架构图都有哪些问题，这些问题会阻碍我们创建好的架构图。

### 方框或其他形状表示什么意思？

随意使用方框或其他形状可能会引起误解。这些形状有可能表示数据、代码或者流程。一个简单的方框可能会引起多种猜想，所以很有必要显式地给这些形状添加有意义的说明。

## 形状的边线表示什么意思？

在一个糟糕的架构图里，形状的边线（虚线、点线，等等）可能会引起误解。边线是否表示某种组件类型（比如虚线表示容器、微服务、层，等等），或者只是因为设计者想让架构图的观感看起来更加丰富一些？所以，在使用多种边线或非标准边线时，要在图例里提供准确的说明。

## 线条或箭头表示什么意思？

线条或箭头可以被理解为数据流（比如从系统 A 到系统 B 的数据流）或元素间的关系（比如组件 A 依赖组件 B）。在大多数情况下，箭头所表示的关系或数据流并不会总是汇聚到同一个方向上，所以很有必要在图例里说明清楚。

## 线条或箭头表示哪一种类型的交互或关联？

尽管线条可以表示数据流或组件间的关系，但用于表示交互类型（对于数据流而言）或关联类型（对于关系而言）的线条或箭头仍然需要详细说明。例如，如果使用线条表示数据流，那么交互类型可以是同步或异步的，但如果线条表示的是关系，那么关联类型有可能是指依赖、继承、实现，等等。这些细节必须在图例里说明。

## 颜色代表什么意思？

一个使用了多种颜色的架构图却没有适当的文档说明很容易引起误解（比如为什么有些方框是绿色的，而其他是红色的？为什么有些线条是黑色的，而有些是蓝色的？）。颜色在架构图里的作用不是非常大，添加太多的颜色并不会给架构图带来更多有价值的信息。一个仅仅使用了黑白两色的架构图也应该是不言自明的，除非非常有必要使用特定的颜色来强调图中的某些部分。在任何情况下都要保持颜色的简单性，如果一定要用多种颜色，不要忘了添加说明。

## 单独的元素或实体

如果架构图中出现了单独的元素或实体，说明架构图有可能是不完整

的。不管是从结构还是从行为角度来看，每一个元素或实体都应该依赖系统的其他部分，或者与它们之间存在某些联系。

## 费解的缩略语或含糊不清的名词

在为架构图中的元素添加标签时，切忌使用费解的缩略语，这样容易引起困惑。如果没有恰当的说明，字母的堆叠（比如 TFH、RBPM，等等）就毫无意义，最好在图例里进行说明（比如 TFH 表示 ticket feed handler，RBPM 表示 rates business process manager）。

在给元素命名时，另一个常见的问题是使用了含糊不清的名词（比如业务逻辑、集成逻辑），这些名词并不能做到自解释。代码里可能也会存在这样的问题，我们建议遵循清晰的代码原则，使用自解释的名字。

## 在架构图中详述技术、框架、编程语言、IDE或开发方法论

架构图并非以技术、框架、编程语言、IDE 或开发方法论为基础，它们只是用来实现架构的工具，而不是中心关注点。它们不应该被包含在架构图里，不过可以在架构描述里简述使用它们的理由。

## 在同一个架构图里混杂运行时元素和静态元素

运行时元素（比如线程、进程、虚拟机、容器、服务、防火墙、数据仓库，等等）不会出现在编译阶段，而且不应该与静态元素（比如组件、包、类）同时出现在同一个架构图里。针对运行时元素有专门的类型图（比如并发图、部署图），一定要注意区别这两种元素，尽可能避免把它们混杂在一起。

## “稍后详述”或“稍后解释”

不包含在架构图里的任何信息都有可能丢失，而架构图里并没有额外的地方可以容纳口头说明。所有的口头说明都会丢失，当其他利益相关者（比如开发人员、架构师）查看架构图时，他们并不知道原来还有所谓的口头说明。试着把所有必要的信息都包含在架构图里，而不是事后加以说明。

## 层级冲突或混合抽象

在同一个架构图里添加不同层级的抽象可能会导致冲突的出现，因为它们是从不同的角度描述问题的。例如，把组件添加到上下文架构图里，或者把类加到部署图里，这些都会偏离架构图原先的目的。在创建架构图时，试着保持相同的抽象层级。

## 使用混乱或含糊不清的架构图表达过量的信息或无效的细节

爱因斯坦曾经说过，“凡事应该尽可能简单，简单到极致”。对于架构图来说也是一样的，我们需要谨慎地选择信息的层级和粒度，这并不是一件容易的事情，它取决于所使用的架构模型、架构师的经验和系统的复杂度。

## 在创建架构图时可参考的指南

除了上面列出的问题清单，还可以参考如下的一些指南来创建更好的架构图。

## 选择最优的架构图数量

Philippe Kruchten 说过，“架构是一项复杂的工作，只使用单个图表来表示架构很容易造成莫名其妙的语义混乱”。要对系统进行良好的文档化，我们不能只使用一种图表。不过在创建架构图的时候，我们也难以确定该使用哪些类型的架构图以及应该创建多少个。在做出决定之前，我们需要考虑多方面的因素。例如，架构的属性和复杂度、架构师的技能和经验、可用的时间、维护成本，以及利益相关者的关注点。网络工程师可能想看到包含了主机、通信端口和协议的网络模型，数据库管理员更关心如何系统是如何操作、管理和分布数据的。所以，我们要选择最优的架构图数量，不管这个数字是多少。

糟糕的架构图（比如缺乏文档）可能会缺失一些信息，反过来说，如果架构图太多（比如过度文档化），那么用于保持架构图一致性和更新架构图的工作量也会相应增加。

## 保持架构图的结构一致性和语义一致性

架构图之间应该在方框、形状、边框、线条、颜色等方面保持一致。架构图的结构外观应该是一样的，团队不同成员创建的架构图不应该给任何一个利益相关者造成理解上的障碍。理想情况下，可以在所有项目里使用相同的建模工具。

从语义角度来看，所有的架构图与最新的代码变更之间以及架构图与架构图之间都应该定期保持同步，因为一个架构图的变更可能会影响到其他架构图。同步可以通过手动进行，也可以通过建模工具自动触发。通过建模工具自动触发会更好一些，不过这也取决于具体的项目。最终的目的是要保持架构图和代码之间的一致性，至于使用什么样的方法或工具可以自行决定。Simon Brown 说，“如果架构图与代码失去了联系，那么就无法用来改进架构”。他的话其实是在强调保持语义一致性的重要性。

## 避免架构图碎片化

架构图越多就越难以理解，而且维护起来也很费劲。最直接的后果就是有可能出现碎片化（比如，通过两到三个架构图来描述同样的质量属性——性能、伸缩性，等等——但每一个架构图都无法完整地描述它们）。在这种情况下，建议移除不能反映相关质量属性的架构图，或者把它们合并起来。

## 保持架构图的可追踪性

保留架构图的变更记录、比较不同版本架构图之间的不同点，以及可以很容易地进行回退，这些都是很重要的。如果建模工具不支持这几点，那么对我们来说可能会是个问题。最近业界倾向于使用简单而直接的文本语言来生成架构图，这样似乎可以解决可追踪性问题。这样做的另一个好处是可以保持架构图之间的结构一致性。

## 在架构图旁边加上图例

如果你没有使用标准的架构描述语言（比如 UML、Archimate），那

么就要在图例里注明每个架构图元素的用意（比如方框、形状、边框、线条、颜色、缩略语，等等）。

如果使用了标准的架构描述语言，只要在图例里添加关键性的架构描述，不需要太多额外的信息，因为看图的人都知道如何按照标准的描述语言规范来理解你的架构图。

## 使用架构描述语言（比如UML、ArchiMate等）会有不一样的效果吗？

关于如何在项目里使用正确的架构描述语言有很多不同的看法。有些人认为 UML 太过死板，用来做架构设计缺乏灵活性，在某种程度上我认同这个看法。有时候，在不依赖 UML 的 profile 和 stereotype 特性的情况下也能很好地完成架构图设计。至于说到其他的架构描述语言，我认为 [ArchiMate](#) 更加强大，相比 UML，它更适合用来为企业系统建模。还有 [BPMN](#)，它更专注业务流程的建模。对这些工具进行深入比较已经超出这篇文章的范围，所以不再累述。

在选择架构描述语言时，综合性和灵活性是首要考虑的因素。但据我所知，完全不使用架构文档的情况也很常见。有些人觉得创建架构文档是一件很无聊的事情，而且觉得它们没有什么意义。这样的项目应该不在少数。人们因为没有使用合适的架构描述语言，所以创建不出很好的架构图，相反，如果他们使用了更好的工具，结果可能会大不一样。但事实并非如此，实际上他们根本不想去创建什么架构文档（包括架构图），更糟糕的是，他们可能不知道该如何创建架构图。这是我们首要解决的问题——了解文档的重要性以及如何创建它们（给软件工程师做培训），然后选择合适的工具。

## 在系统和架构发生变化时如何更新架构图？

更新架构图有几种方式，我将会介绍其中的三种。第一种，也是最简单的一种，就是直接从代码生成架构图。这样可以保证架构图与代码是一致的。不过目前的工具还不能完全支持这种方式，在没有人工介入的情况下

下，工具还无法基于代码生成精确且有意义的架构图。Len Bass 说，“在最理想的开发环境里，只需要一个按钮就能得到我们想要的文档”。他指的就是自动生成架构图，但我们还远远没有达到那种程度。

第二种，先用建模工具创建架构图，然后生成代码骨架（比如组件或包、API），随后开发人员可以在骨架上添加代码。每次架构图发生变更，需要从架构图端重新生成代码骨架。

第三种，在每次加入新特性时手动更新架构图。为了保证代码与架构图的一致性，建议把更新架构图作为开发流程的一部分。不过我们不推荐这种方式，因为这样很容易造成架构图过时或出现不一致（比如开发人员总是忘记或者不想更新架构图）。

我的建议是混合使用现有的工具，结合手动和自动的方式来创建架构图。例如，尝试自动生成架构图，使用工具基于代码渲染出符合基本要求的架构图，不包含混乱无用的信息。架构图可以高度可变（比如可以适应频繁的开发变更，这类架构图一般具有较低层次的抽象），也可以是静态的。这类图表可以是上下文架构图、参考架构图、包图、类图、实体图，等等。不过有时候仅仅基于代码无法生成满足需求的架构图，所以在这种情况下，自动创建架构图不是理想的方式。这个时候需要手动建模作为补充，这类架构图包括时序图、状态图、并发图、部署图、运营图，等等。

## 现代架构（如微服务）对架构图有什么影响？

微服务或其他任何一个现代架构风格（如无服务器、事件驱动）只会影响到系统的结构、组件间的交互方式（比如组件间的关系）和原则。在我看来，我不认为架构风格会改变架构图原先的含义。不过，相比传统的系统（比如单体），我们所说的现代系统架构具有更高层次的复杂性，它们对架构描述和架构图确实会有一些影响，这些是我们需要注意的。我们需要考虑分布式组件（比如分布式微服务）、每种组件的类型、组件间的交互方式（比如边界、API、消息）、他们的生命周期以及从属关系。

综上所述，我们需要在架构图中体现系统的分解、开发、部署和运维。

假设有一个包含了大量微服务的系统，它就会有很多的架构图，因为每个微服务都可能有自己的架构图。一致性（例如，改变一个服务的 API 会影响到其他服务，所有相关的架构图都需要做出修改）、碎片化（例如，一个架构图无法反映分布式服务的高可用性和性能）和横断面（例如，是谁在负责处理系统的监控或安全问题）问题会让人手忙脚乱。我们首要面对的挑战是如何进行良好的团队协作，不仅仅是开发，也包括后续的维护。

总而言之，现代系统的复杂性会带来额外的问题，它们会在架构图层面造成一定程度的影响。

## 关于作者

**Ionut Balosin** 是 Luxoft 的软件架构师，拥有十年以上的应用程序开发经验，专注性能调优和软件架构。他是开发大会的演讲常客，也是一名技术培训师。

# ArchSummit 全球架构师峰会 2017

12.8-9 北京·国际会议中心

## 聚焦

- 大前端技术与管理
- 互联网产品与创业
- 大型游戏架构
- 数据库架构
- 新一代 DevOps
- 深入机器学习
- 金融应用架构
- 微服务架构
- 人工智能与业务应用
- 硅谷前沿技术
- 内容分发与精准推荐
- 大数据平台架构
- 业务架构
- 国际化架构设计
- 大数据平台架构
- 架构升级与优化
- 工程师文化与团队建设
- 流媒体架构

## 分享

**京东硅谷研究院 | 自动深度语法分析与自然语言应用**  
如何结合知识图谱和大数据舆情挖掘，展示深度语法分析的原理和威力

**腾讯 | 黑产对抗与千万在线后台服务演进**  
核心模块如何不断优化重构，如何应对层出不穷的黑色产业模式与内容

**百度 | 数据仓库开源架构解读与应用**  
数百TB乃至PB级别的数据量，如何达到毫秒/秒级分析

**饿了么 | 移动性能可视化之路**  
如何实现85%页面秒开，包体积减小20%+

**58转转 | C2C电商平台推荐系统架构演进**  
多策略推荐系统架构如何设计以及实时化升级

.....

## 联席主席



崔宝秋  
小米 首席架构师，  
云平台负责人



赵宇辰  
销售易 技术VP  
首席科学家



刘海峰  
京东  
商城首席架构师



张木喜  
musical.ly  
高级技术副总裁



谭待  
百度  
主任架构师

## 出品人与分享嘉宾



侯兆炜  
Tech Lead  
Manager  
@Facebook



吴惠君  
Data Platform  
/Engineer  
@Twitter



李维（博士）  
京东硅谷研究院  
主任研究员



何登成  
阿里巴巴  
资深技术专家



黄斯亮  
腾讯音乐  
后台技术总监



姜宁  
华为  
技术专家



牟宇航  
百度  
大数据部技术经理



于冰  
快手  
视频团队技术总监



胡彪  
饿了么  
移动技术总监

.....

## 知名互联网公司系统架构图

登入archsummit.com  
快速获取ArchSummit历届大会讲师的  
分享结晶与创意



使用限时优惠码AS200，以目前**最优惠价格**报名ArchSummit  
仅限前20名用户，优惠码有效期至9月19日，扫描右方二维码即可使用  
如果在使用过程中遇到任何问题，可联系大会主办方，欢迎咨询！

微信: aschina666

电话: 15201647919

# 从金属巨人到深度学习，人工智能（极）简史

作者 Mark Aduol 译者 大愚若智



为了保护克里特岛防御海盗和入侵者，人们创造了巨型青铜战士塔罗斯（Talos）。他每天环绕全岛三圈，勇武的造型吓得海盗们只能另觅他处。但在勇猛外表下，塔罗斯并没有所谓的“勇士之心”，他只是个机器人。就像稻草人一样，生来只是为了对外表现出这种骁勇形象。然而信徒们认为，匠人已经为塔罗斯这样的作品灌注了真正的心智、喜怒哀乐、思想，以及智慧。当然这不是真的。塔罗斯也仅仅是梦想的一种外在表现，而这样的梦想几乎贯穿了人类的整个历史：我们多想创造出如同我们自己一样栩栩如生的智慧生命啊。

**亦或如作家 Pamela McCorduck 所说：“这就是一种锻造众神的古老愿望”。**

科学家、数学家、哲学家，甚至作家，对于创造所谓“会思考的机器”的方法已经思考了很久。同时，又有什么比人类自身更像是“会思考的机

器”呢？

自从创造出诸如塔罗斯这样会动的机器后，我们身边的匠人们对于简单的“拟人”智慧就不再感兴趣了，他们开始追求真正的智慧。这些“没头脑”的机器人仅让他们管窥到智慧之表，却并未揭示智慧之本。为此他们必须深入领略智慧最明确的体现：人类的心灵。

**正如经济学人所说：“如欲开悟，必先自省”（For enlightenment, look within）。**

人们很快意识到，人类与其他不那么智慧的生物间最大的差别，并不在于脑容量或在地球上生存时间的长短，真相其实很简单，仅仅在于我们卓越的推理能力。因此首个可编程计算机的构想产生后，我们会理所当然地认为，这样的计算机将能模拟任何形式的推理过程，至少能够像人一样进行推理。事实上，“计算机（Computer）”这个词的首次使用可以上溯至 1640 年代的英格兰，当时这个词被用于代表“会进行计算的人”。

最开始，这个过程的进展非常缓慢。1940 年代，当时最先进的哈佛马克一号（Harvard Mark I）是一个重达 10,000 磅，由数千个机械组件驱动的“怪兽”，为了让这个机器动起来，内部共使用了长达 500 英里的线缆。尽管有如此精心巧妙的设计，这个机器每秒钟只能执行三次加法运算。但随着摩尔定律的影响，计算机很快在形式推理各种任务的执行方面获得了超出人类能力的表现。研究人员对所取得的进展感到惊喜，并断言只要按照这样的速度继续发展，首个真正完善的“会思考的机器”变为现实将仅仅是时间问题。1960 年代，20 世纪知名学者司马贺（Herbert Simon）甚至宣称：“20 年内，机器将能从事人能做到的一切工作”。很可惜，虽然足够惊人，但这个预言没能实现。

实际上计算机确实很擅长解决能够通过一系列逻辑和数学规则定义的问题，但更大的挑战在于让计算机解决无法通过这种以“声明”方式归纳提炼的问题，例如识别图片中的人脸，或者翻译人的语言。

整个世界始终混乱不堪，机器下象棋的水平也许远胜于人类，甚至可能赢得象棋锦标赛冠军，但放眼现实世界，机器的作用其实和橡皮小黄鸭

差不多（除非你从事的本身就是小黄鸭调试法，那就要另说了）。

意识到这一点后，很多 AI 领域的研究者开始拒绝承认符号化 AI (Symbolic AI，一种描述形式推理方法的涵盖性术语，至今依然在 AI 研究领域处于支配地位) 是创建人工智能机器的最佳方式这一原则。符号化 AI 的基石，例如 Situation Calculus(情景演算)和 First-Order Logic(一阶逻辑) 被证明因为过于形式化并且过于严格而无法容纳现实世界中的所有不确定性。我们需要新的方法。

一些研究人员决定通过更为巧妙的“模糊逻辑”(Fuzzy Logic) 寻求答案，在这种逻辑范式中，真实的值不是简单的 0 和 1，而可以是介于这两个数之间的任何值。还有其他研究人员决定专注于别的新兴领域，例如“机器学习”。

机器学习弥补了形式逻辑的不足，可顺利解决真实世界的不确定性问题。这种方式并不需要将有关现实世界的所有知识“硬编码”至一系列严格的逻辑公式中，而是可以教计算机自行推导出所需知识。也就是说，我们并不需要告诉计算机“这是一把椅子”或“这是一张桌子”，我们可以教计算机学习如何将椅子和桌子的概念区分开来。机器学习领域的研究人员会谨慎地避免使用确定性概念描述整个世界，因为这种严格的描述特性与现实世界的本质是截然相悖的。

于是他们决定使用统计学和概率论语言来描述整个世界。

**机器学习算法并不需要了解真理和谬误，只需要了解真实和虚假的程度，也就是概率。**

这种使用概率，以数值方式了解现实世界中所存在不确定性的想法，使得贝氏统计学(Bayesian statistics)成为机器学习的基石。[“频率学派”](#)(Frequentists) 对此有不同看法，不过这个分歧还是另行撰文介绍吧。

很快，诸如逻辑回归和朴素贝叶斯等简单的机器学习算法已经可以教计算机区分合法邮件和垃圾邮件，并能根据面积预测房屋价格。逻辑回归是一种相当简单的算法：给出一个输入向量  $x$ ，模型会直接将这个  $x$  分类至  $\{1, 2, \dots, k\}$  多个类别之一。

然而这就会导致一个问题。

**这种简单算法的效果严重依赖所使用的数据表达方法 ( Goodfellow et al. 2017 )。**

为了更形象地理解这个问题，可以试着假设构建一种使用逻辑回归判断是否建议进行剖腹产的机器学习系统。系统无法直接检查产妇，因此需要通过医生提供的信息来判断。这种信息可能包含是否存在子宫疤痕、怀孕月数、产妇年龄等。每个信息可以算作一个特征，通过将不同特征结合起来，AI 系统就可以全面了解产妇的表征。

通过提供训练数据，逻辑回归算法可以学习产妇的不同特征与各种结果之间的关系。例如，算法可以从训练数据中发现，随着产妇年龄的增长，分娩过程中出现“恶心反胃”情况的风险会增加，因此算法会降低向高龄产妇推荐自然分娩的概率。

虽然逻辑回归可将表征与结果对应，但实际上并不能决定哪些特征可以组成产妇的表征。

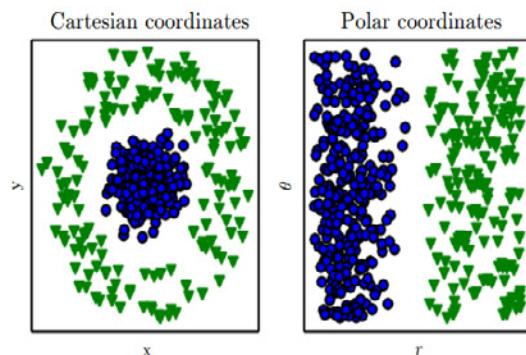
**如果直接为逻辑回归算法提供患者的 MRI 扫描结果，而非医生的正式报告，那么算法将无法提供有用的预测 ( Goodfellow et al. 2017 )。**

单纯就 MRI 扫描结果中的每个像素来说，几乎无法帮助我们判断产妇分娩过程中遇到并发症的可能。

这种足够好的表征，与足够好结果之间的依赖性广泛存在于计算机科学和我们的日常生活中。例如，我们几乎可以瞬间在 Spotify 上找到任何歌曲，因为他们的曲库很可能就是用智能的数据结构来存储的，例如三元搜索尝试 (Ternary search tries)，而非常见的简单结构，例如无序数组。另一个例子：学童可以使用阿拉伯数字轻松进行数学计算，但如果使用罗马数字，情况就截然不同了。机器学习也是如此，输入表征的选择将对学习算法的效果产生巨大影响。

下图是不同表征的范例。假设我们需要在散点图上画一根线将两类数据分开。左图使用笛卡尔坐标系呈现这些数据，此时几乎无法做到；右图对同一批数据使用了极坐标系，一条竖线即可解决问题。此图与 David

Warde-Farley 合作制作。



**David Warde-Farley, Goodfellow et al. 2017**

因此人工智能领域的很多问题实际上可以通过为输入数据寻找更适合的表征这种方式进行简化。例如，假设我们要设计一套算法来学习识别 Instagram 照片中的汉堡。首先要构建一个用来描述所有汉堡的特征集。最初我们可能会用图片中的原始像素值来描述汉堡，一开始你也许觉得这种做法很合理，但很快会发现根本不是这样。

单凭原始像素值，很难描述汉堡看起来是什么样的。想想你自己在麦当劳点汉堡时的场景吧（如果你还会在他家吃饭的话）。你也许会用不同“特征”来描述自己想要怎样的汉堡，例如奶酪、三分熟的牛肉饼、表面撒有芝麻的圆面包、生菜、红洋葱，以及各种酱料。结合这种情况考虑，也许可以用类似的方式构造我们需要的特征集。我们可以将汉堡描述成一种不同成分的集合，每个成分又可以用各自不同的特征集来描述。大部分汉堡的成分都可以用其颜色和外形来描述，进而汉堡作为整体也就可以使用不同成分的颜色和外形来描述了。

但如果汉堡不在照片正中央，周围有其他颜色相近的物体，或者是一间风格迥异的餐厅，他们提供没有“组装”在一起的汉堡，此时又该怎么办？算法该如何区分这些颜色或几何造型？最显而易见的解决方式无疑是增加更多（可分辨的）特征，但这也仅仅是权宜之计，很快你将会遇到更多边缘案例，需要增加更多特征才能区分类似图片。输入的表征越来越复杂，计算成本增加，同时会让情况变得更棘手。因此从业者现在不仅需要

关注数量，同时也要关注所输入表征中，所有特征的表现能力。对于任何机器学习算法，寻找完美的特征集都是一个复杂过程，需要花费大量时间精力，甚至需要大量有经验的研究人员投入数十年的时间。

确定如何以最佳方式呈现输入给学习算法的数据，行话来说实际上是一种“表征”问题。

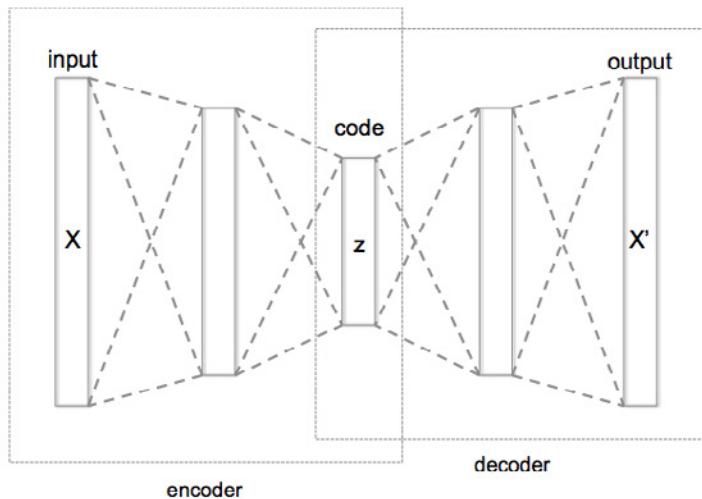
1990 年代末到 2000 年代初，机器学习算法在不完美输入表征方面的局限对 AI 发展产生了巨大阻碍。在设计输入特征的表征时，工程师们没有任何选择，只能依赖人类自身的才智以及围绕问题所在领域的先验知识 (Prior knowledge) 克服这些局限。长久以来，这样的“特征工程”始终站不住脚，如果某个学习算法无法从未筛选的原始输入数据中提取出任何见解，那么用更具哲学意义的话来说，它就无法理解我们的世界。

面对这些困难，研究人员快速发现了一种应对之道。如果机器学习算法的目标是学着将表征与输出结果进行映射，为何不教它们学习表征本身。这种方式也叫做表征学习。最著名的例子可能就是 autoencoder，这是一种神经网络，根据人脑和神经系统进行建模的计算机系统。

Autoencoder 实际上是编码器 (Encoder) 函数和解码器 (Decoder) 函数的组合，编码器函数负责将输入的数据转换为不同表征，解码器函数负责将中间态的表征重新转换为原始格式，并在这一过程中尽可能多地保留信息。这样就可以在编码器和解码器之间产生一个分界 (Split)，输入的“噪音”图像可解码出更有用的表征。例如，噪音图像可能是一张 Instagram 照片，其中有一个汉堡，周围还有很多颜色近似的物体。解码器可以消除这些“噪音”，只保留描述汉堡本身所需的图片特征。

但就算有了 autoencoder，问题依然存在。为了消除噪音，autoencoder (以及任何其他表征学习算法) 必须能精确确定哪些因素对输入数据的描述是最重要的。我们希望自己的算法能选择恰当的因素，使其更好地识别出真正感兴趣的图片 (例如包含汉堡的图片)，并排除不感兴趣的图片。在汉堡这个例子中，我们已经明确，如果能更专注于图片中不同元素的外形和颜色，而非只关注图片的原始像素值，就可以很好地区

分包含和不包含汉堡的图片。然而永远都是知易行难。重点在于教算法如何从不重要的因素中解读出重要的因素，也就是说，需要教算法识别所谓的因素变体（Factors of variation）。



**作者：Chervinskii，自行制作，依 CC BY-SA 4.0 方式许可**

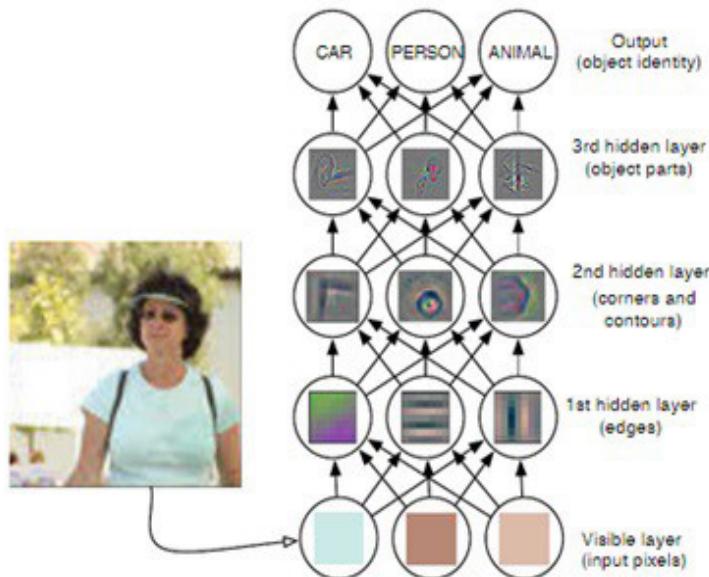
初看起来，表征学习似乎没法解决这个问题，但还是仔细看看吧。

编码器接受输入的表征并通过传入一个隐藏层（中间层），将输入结果压缩为略小一点的格式。解码器的作用截然相反：将输入内容重新解压缩为原始格式，并尽可能多地保留信息。两种情况下，如果隐藏层能够知道哪些因素是描述输入内容时最重要的，并尽可能确保这些因素在该层传递过程中不会从输入数据中消除，输入数据中包含的信息将得到最大程度的保留。

在上图示例中，编码器和解码器分别只包含一个隐藏层：一层用于压缩，一层用于解压缩。这种粗粒度的层数意味着算法在判断如何以最佳方式对输入数据进行压缩和解压缩，进而保留最大量信息的过程中缺乏足够灵活性。但如果略微改动一下设计，引入多个隐藏层并按顺序堆叠在一起，在选择重要因素时，算法就可以更自由地判断对输入数据压缩和解压缩的最佳方式。

**这种在神经网络中使用多个隐藏层的方法就是深度学习。**

但事情还没完，深度学习还可以更进一步。在使用多个隐藏层后，只需构造一个更简单的层就可以建立复杂的表征。通过按顺序堆叠隐藏层，我们可以在每一层中识别新的因素变体，这样算法就可以用更简单的层表达更复杂的概念。



### Zeiler and Fergus (2014)

深度学习有着深远悠久的历史。这一领域的中心思想早在 1960 年代就已通过多层感知器 (Multi-layer perceptrons) 的形式诞生，后来在 1970 年代首次出现了更实用的反向传播算法，1980 年代出现了人工神经网络。尽管历史悠久，这些技术依然花了数十年才变得实用。这些算法本身并不差（尽管很多人这样想），我们只是没有意识到为了让他们变得足够实用需要提供多大量的数据。

由于统计噪声的影响，小规模数据样本更有可能获得极端的结果。然而只要增大数据量，就可以降低噪声影响让深度学习模型更精确地确定输入数据最适合的描述因素。

毫无疑问，21 世纪初，深度学习终于一飞冲天，与此同时很多大型科技公司也发现自己正坐在有待开发的数据金矿顶端。

# 人工智能那些事儿

作者 Erik Brynjolfsson 等 译者 NER



在过去的 250 多年里，技术创新一直是经济发展的根本推动力。这些技术创新中最重要的就是经济学家所说的“通用技术”，包括蒸汽机、电力，以及内燃机。它们中的每一个都催化了互补性创新与机遇的浪潮。举例来说，内燃机让汽车、卡车、飞机、链锯、割草机，甚至大型零售商、购物中心、交叉对接仓库、新供应链以及郊区得以出现。像沃尔玛、UPS 和 Uber 这样拥有多样性的公司找到了利用新技术创造新商业模式的方法。

我们这个时代最重要的通用技术就是人工智能，尤其是机器学习，也就是说机器能够持续提高自己的性能，而无须人类明确解释所有这些任务要怎样完成。在过去几年的时间里，机器学习已经变得越来越高效和广泛地使用。我们现在已经能建造出自己学习如何完成任务的系统了。

为什么这件事非常重要呢？有两个原因。第一，人类的知识比我们能

表达出的更多，我们不能解释为什么人类能完成那么多的事情，从识别出一张人脸到在古老的亚洲策略游戏围棋中走出绝妙的一招。在机器学习之前，我们无法精确表达出我们的知识，这种无能正意味着我们不能自动化很多事情，而现在我们可以做到了。

第二，机器学习系统是非常出色的学习者。这些系统能在广泛的领域中达到超人类性能，包括检测欺诈和诊断疾病等。人们在整个经济领域中都部署了这样出色的数字学习者，它们的影响力将会十分深刻。

在商业领域，人工智能在早期通用技术的阶段就被认为拥有变革性的影响。虽然它目前已经被应用于全球上千家公司，但大多数重大的机遇并没有被利用开发出来。随着制造业、零售业、交通运输、金融业、医疗保健行业、法律、广告业、保险业、娱乐、教育业，以及事实上每一个其他领域转变其核心进程和商业模式，并从机器学习中受益，人工智能的影响，在即将到来的这个十年中一定会被放大。现在的瓶颈在于管理、执行，以及商业想象力。

然而，就像很多其他新技术一样，人工智能也催生出了一大批不切实际的期望。我们看到有大量商业计划随意挥洒在机器学习、神经网络，以及各种其他形式的技术方面，但却几乎与其真正的功能没有联系。举个例子来说，简单地把一个约会网站叫做“人工智能驱动的网站”，并不能让它变得更高效，但那或许有助于网站融资。这篇文章将穿过这些喧嚣的噪音，描述出人工智能的真正潜力、它的实践意义，以及它在被采用的过程中面临的障碍。

## 人工智能今天能够做些什么？

“人工智能”这个词是1955年由约翰·麦卡锡(John McCarthy)创造的，麦卡锡是达特茅斯学院的一位数学教授，他组织了之后一年那场具有开创意义的人工智能大会。从那以后，也许部分原因是因其令人回味的名称，人工智能这个领域开始崛起，而不仅仅停留在梦幻般的主张和承诺上了。在1957年，经济学家赫伯特·西蒙(Herbert Simon)预测，十年之内，

计算机将在国际象棋方面打败人类。（事实上，计算机只花了 40 年时间。）在 1967 年，认知科学家马文·明斯基（Marvin Minsky）说：“在一代人之内，创造‘人工智能’这个问题将会得到实质解决。”西蒙和明斯基二人都是知识分子中的巨擘，但他们都错了。所以，对未来突破的戏剧性主张遭到了一定程度的怀疑。

我们先来看看，人工智能现在在做些什么，以及它在以多快的速度发展。最大的两个进步发生在这样两个广阔的领域：感知和认知。在早期的分类中，最有实用性的进步都是跟语音有关的。语音识别还差强人意，但现在有百万计的人们在使用它，想想 Siri、Alexa，以及 Google 的语音助手。你现在在读的这篇文章，最开始是我口述给一台计算机并让它以足够的精确度转写出来的，这样比打字要快。由斯坦福计算机科学家詹姆斯·兰迪（James Landay）和他的同事们进行的一项研究发现，平均来说，语音识别比在手机上打字要快三倍，其错误率已经由曾经的 8.5% 降低到了 4.9%。令人震惊的是，这个显著的改进并不是经过 10 多年时间才实现的，而仅仅是从 2016 年夏天才开始。

同样地，图像识别也进步得非常惊人。你可能已经注意到，Facebook 和其他应用程序现在可以在发出的图片中识别出你很多朋友的脸，并且提示你给他们贴标签。一个在你智能手机上的应用就能识别出野外的任何一只鸟。在一些公司总部，图片识别甚至正在取代身份证件。视觉系统，比如那些用在自动驾驶汽车上的视觉系统，以前在识别行人方面每 30 帧中就会出现一次错误，而现在它们的错误率比在 3000 万帧中出错一次还要低。图片识别有一个巨大的数据库，叫 ImageNet，它拥有几百万常见的、模糊的或完全怪异的照片，顶级系统的图片识别错误率已经从 2010 年的



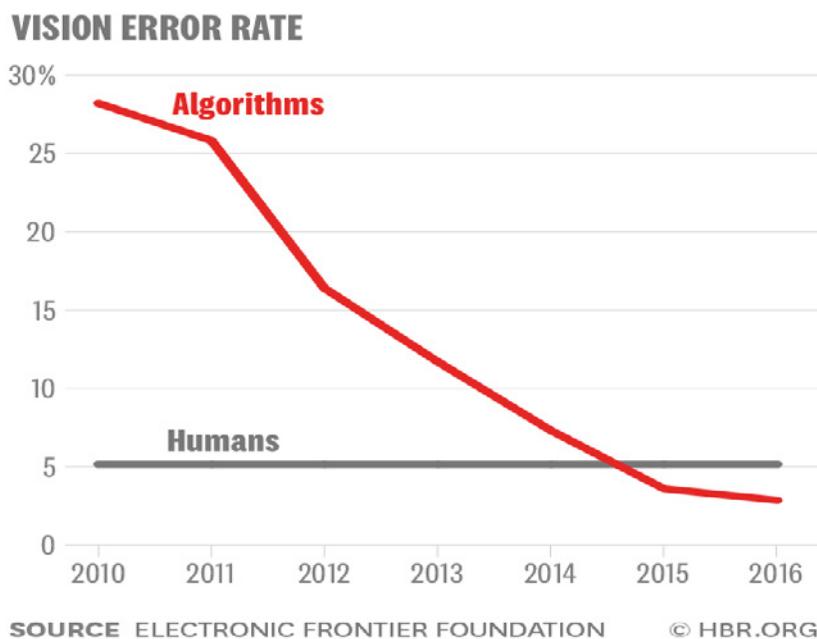
30% 多，降低到了 2016 年的 4%。

最近几年来，由于采用了基于庞大的或“深度的”神经网络的新方法，图片识别的改进速度迅速加快。视觉系统的机器学习还远非完美，但即使是人类，也可能会在快速识别出一只小狗方面有困难，人们也可能会在根本没有可爱小狗的地方看出小狗来。

小狗还是玛芬蛋糕？图像识别的进步。

机器已经在识别相似图像的类别方面取得了长足的进步。

## 图片识别的错误率



第二个主要改进的方面是认知和问题解决。机器已经战胜了最棒的人类扑克选手和围棋选手，这是一项专家们之前预测需要至少又一个十年的时间才能实现的成就。Google 的 DeepMind 团队用机器学习系统提高冷却数据中心的效率，高达 15 个百分点，即使人类专家已经优化过数据中心，它也还是达到了这个水平。网络安全公司 Deep Instinct 正在使用智能代理来检测恶意软件，PayPal 用智能代理防止洗钱行为。使用 IBM 技术的一家新加坡保险公司支持用户自动进行索赔流程，而数据科学平台

Lumidatum 能提供实时建议来改进客户支持系统。一大批公司在使用机器学习来决定接受华尔街的哪一笔买卖，有越来越多的信用决策都在机器学习的帮助下做出。亚马逊采用机器学习优化库存并改善他们向用户推送的产品推荐。Infinite Analytics 开发了一个机器学习系统，来预测用户是否会点击一个特定的广告，改善全球消费品商品公司的在线广告布局，另一个系统则是为了优化用户在一个巴西网络零售商的搜索和发现过程。第一个系统将广告投资回报率提高了三倍，第二个系统则增加了 1.25 亿美元的年收入。

机器学习系统不仅在很多应用里代替着原始的算法，而且在很多人类曾经表现最好的工作任务中更为出色。虽然这个系统还有待改进，它们在 ImageNet 数据库的错误率（大约是 5%）已经达到或比人类水平的表现更好了。语音识别也是这样，即使在嘈杂的环境下，现在也几乎和人类水平持平。机器学习系统达到了这样的门槛，进而为改变工作场所和经济开辟出了新的广阔可能性。一旦以人工智能为基础的系统在某个领域超越了人类的表现，它们就更容易快速广泛传播。举个例子，比如 Aptonomy 和 Sanbot，它们分别是无人机和机器人制造者，它们正在使用改进过的视觉系统，使很多安保工作自动化。在很多同类公司中，软件公司 Affectiva 正在使用机器学习在特定群体中识别快乐、惊讶和气愤等情绪。Enlitic 则是一家使用机器学习扫描医学图像进而帮助诊断癌症的深度学习创业公司，而这样的公司已经有好几家了。

这些都是了不起的成就，但以人工智能为基础的系统，其适用性依然非常狭窄。举个例子，机器学习在拥有数百万图片的 ImageNet 数据库中的出色表现，并不意味着它总能“在野外环境中”取得一样的成功，在野外环境中，光线条件、角度、图片分辨率以及情境都可能非常不同。更为根本地，我们可能会惊叹于一个系统能理解中国话并把它翻译成英文，但我们不能指望这个系统理解一个特定中文字的意义，更不用说在北京去哪里吃饭好了。如果一个人能出色地完成一项任务，那很自然也可以假设他有能力完成一些相关的工作。但是机器学习系统就是为了某些特定任务而

训练出来的，它一贯的知识并不会扩展延伸。一个典型的谬论就是认为计算机狭窄的理解力意味着它能扩展到更广阔的理解力中，这可能是一个最大的混淆之源，更为夸张的宣称就是认为人工智能能够自己取得进步。我们离具备在多领域中拥有通用智能的机器还非常遥远。

## 理解机器学习

要理解机器学习，最重要的一点就是明白机器学习代表了一条创造软件的完全不同的道路。举个例子，机器是去学习一件事情，而不是为某一明确结果被明确编程成什么样子。在过去 50 年的绝大多数时间里，信息技术领域的进步及其应用都聚焦于把某种已有的知识和程序编成指令，再把这些指令植入机器中。确实，“编程”这个词总是意味着这样一种艰苦的过程，即开发者把自己头脑中的知识转化成一种机器能理解和执行的格式。这种方法有一个根本上的弱点：我们现有的很多知识都是大家心照不宣的，也就是说我们无法完全解释它们。对我们来说，写下每一条指令让另一个人明白如何骑自行车、如何识别出一个朋友的脸庞，这几乎是不可能的。



上图：这就是使用人工智能的意义。结果是人又不是人，可识别但又不是你期望中的那样，它们美丽吗，可怕吗，能让人感到愉悦吗？

换句话说，我们所知的比我们能表达的更多。机器学习正在克服这个困难。在第二次机器革命的这第二波浪潮中，人类制造的机器正在从实例中学习，并且使用结构清晰的反馈来解决自己的问题，比如面部识别。

## 机器学习的不同特色

人工智能和机器学习有很多种特色，但近年来大多数成功的案例都集中在监督学习方面，也就是关于某特定问题，赋予机器大量正确的实例学习。这个过程几乎总涉及从一组输入 X，到一组输出 Y 的映射。比如，输入可能是一些各种动物的图片，正确的输出就是关于这些动物的标签：猫、狗、马等。输入也可以是一段音频的声音波形，正确的输出就是一些词汇：是、否、你好、再见等。

成功的系统通常使用几千个甚至几百万个实例的训练数据集，每个实例都已经被标记出正确的答案，系统会再大体看一下新的实例，如果训练顺利，系统就会以高度的精确度来预测答案。

算法的成功多半要依仗一种叫“深度学习”的方式，而深度学习利用的是神经网络。和早期机器学习算法相比，深度学习算法有一个重要的优点：深度学习能够更好地使用大得多的数据库。旧的系统会随着训练数据实例的增加而改进，但会到达一个点，在那个点之后再增加数据并不能带来更好的预测。这个领域的领军人之一吴恩达说：“深度神经网络就不会在这种方式下失效，更多的数据的确会带来更好的预测。”一些非常大的系统是由 3600 万或更多实例训练出来的。当然，要使用极大的数据库就需要更加强大的处理能力，这就是为什么非常大的系统通常在超级计算机或专用计算机上运行。

如果你有很多有关行为的数据并试图预测结果，这就是监督学习系统的潜在应用机会。亚马逊的全球消费者部门的 CEO 杰夫·威尔克（Jeff Wilke）说：“监督学习系统已经在很大程度上取代了用于向客户提供个性化建议的基于内存的过滤算法。”摩根大通则引入了一个系统来检查商业贷款合同，这项工作以前需要负责贷款的员工用 360000 个小时来完成，而现在只需要几秒钟了。监督学习系统还被用于诊断皮肤癌。上面所说的只是部分例子而已。

相对来说，标记一组数据并把它用于训练监督学习系统是比较简单直

接的。这也是为什么监督学习式机器学习系统比无监督学习系统更为常见，至少目前是如此。无监督学习系统想要自己学习。我们人类就是出色的无监督学习者，我们用很少的没有标签的数据就能从这个世界上获取大部分知识，比如识别出一棵树，但是开发出一个如此运行的成功的机器学习系统就极端困难。

如果我们能建立强大的无监督学习系统，就将开启令人振奋的新的可能性。这些机器将能够用全新的方法审视复杂的问题，帮我们找出其中的模式，可用于观察疾病传播、市场证券价格走势、客户的购买行为等等。正是这种可能性引领着 Facebook 的 AI 研究主管、纽约大学教授 Yann LeCun，他把监督学习系统比作在蛋糕上撒糖霜，而把无监督学习比作蛋糕本身。

在这个领域里，另一个渺小但是在成长中的领域就是强化学习。它已经被嵌入了雅达利电子游戏和围棋这样的棋盘游戏中。它还能帮助优化数据中心的电力使用，甚至为股票制定交易策略。Kindred 公司制造的机器人能用机器学习来辨识和归类它们从没遇到过的物体，还能加快消费品配送中心的运送速度。在强化学习系统中，编程人员会具体说明系统的现状和目标，列出可被允许的行为，描述会影响和限制行为结果的环境因素。在可被允许的行为下，系统要找出尽可能接近目标的方法。人类可以具体说明目标而不需要说明如何做到，在这种情况下系统运行得最好。比如，微软利用强化学习来为 MSN 网站的新闻报道选标题，方法就是在点开链接的用户更多的时候，给系统打更高的分数作为奖励。系统会尝试着在编程人员给定规则的基础上最大化它的分数。当然，这就意味着强化学习系统会针对你明确奖励的目标进行自身优化，而不一定针对你真正关心的目标来优化，因此，准确而清晰地指定目标至关重要。

下图：今天的人工智能应用都是由人类来驱动的，医生尝试着去解决一个癌症患者的病痛，家庭厨师在寻找新的菜谱，通勤上班族决定着如何开车出门。



## 把机器学习带入工作中

对那些期望把机器学习付诸实践的组织来说，现在有三个好消息。第一，人工智能在广泛地传播。这个世界上还远没有足够的数据科学家和机器学习专家，但在线教育资源和大学院校正在努力迎合这种需求。其中最好的资源包括Udacity、Coursera 和 fast.ai，他们不仅教授概念性的东西，而且能真正让学生们去实现工业级别的机器学习部署。除了培养自己的员工之外，感兴趣的公司还可以利用 Upwork、Topcoder 和 Kaggle 这样的在线人才平台寻找具备专业知识的机器学习专家。

第二，对现代人工智能来说十分必要的算法和硬件已经可以被买到或租赁到。Google、亚马逊、微软和 Salesforce 等公司都在建构强大的机器学习基础设施，并且都可以通过云系统得到。在这些竞争对手之间存在激烈的竞争，这就意味着，随着时间推移，那些想要尝试和部署机器学习的公司将看到越来越多可获得的平价功能。

第三，也许你并不需要那么大量的数据才能开始利用机器学习。大多数机器学习系统的表现都会随着它们得到更多数据而提升，所以，似乎拥有最多数据的公司将会取得胜利。在这种情况下，“胜利”意味着“控制某一单一应用，比如广告定位或者语音识别的全球市场”。但如果胜利的定义被转变为“显著提高性能”，那么其实充足的数据是非常容易获得的。

机器学习正在三个层面推动变革：任务和职业、商业进程、商业模式。用机器视觉系统识别出潜在的癌细胞就是第一个层面变革的极好例证，它

把放射学家解放出来，让他们能够专注于真正重要的事情，能够更好地和病人沟通，和其他医生协作。对商业进程的变革也有一个例子，就是亚马逊引入了机器人，并使用以机器学习为基础的优化算法，重新发明了工作流程，重新布局了亚马逊的各个履职中心。同样地，商业模式也需要利用机器学习系统来重新思考，这些系统可以智能地定制化地推荐音乐、电影等。更好的模式不是以消费者选择为基础销售单曲，而是提供一种预订和播放特定用户可能会喜欢的音乐这样一种个性化订阅服务，即使这个用户可能根本没听说这些音乐。

## 风险和极限

第二次机器革命的第二波浪潮也带来了新的风险。尤其是，机器学习系统是“难以解释的”，也就是说我们人类很难理解系统是如何作出决定的。深度神经网络可能拥有数亿个连接，每一个连接都为最终的决策贡献了一点力量。结果就是这些系统的预测是无法简单明晰地被解释出来的，机器知道的比它们能告诉我们的更多。

这就带来了三个方面的风险。第一，机器可能会有隐藏的偏见，这些偏见不是来自机器设计者的意图，而是来自训练它们的数据。比如，如果一个系统利用人类数据库的决策学习可以接受面试中的哪些工作申请，它可能会不经意间评估应聘者的种族、性别、民族等。更进一步，它们的偏见可能不会表现成明确的规则，而是嵌入在上千种考虑因素的细微互动之中。

第二，与建立在明确逻辑规则上的传统系统不同，神经网络系统处理的是数据事实，而不是绝对的事实。可能很难证明这个系统是完全确定可以在任何情况下正常工作，尤其是在训练数据时没有涉及到的情况下。缺乏确定性可能是在处理关键任务时的一个问题，比如控制核电厂，或者涉及生死攸关的决定。

第三，当机器学习系统犯错的时候（犯错几乎不可避免），诊断和纠正错误都极端困难。得出解决方案的基础结构可能是我们难以想象地复杂

的，如果系统的训练条件改变了，得出的解决方案可能远非最优。

这些风险都非常真实，合适的基准不是追求完美，而是追求最优的可选项。毕竟我们人类也会有偏见、犯错误，还觉得诚实解释我们做出决定的过程很困难。以机器为基础的系统，其优点在于它可以随着时间推移而改进，而且你给它什么样的数据它都会得出一致的回答。

这是否意味着人工智能和机器学习能做的事情就没有极限呢？感知和认知覆盖了绝大部分的领域，从开汽车到预测销售，甚至还能决定雇佣什么人、提拔什么人。我们相信，在绝大多数领域，人工智能很快就会超越人类水平的表现。那么，人工智能和机器学习不能做什么呢？

我们有时会听到这样的说法：人工智能永远无法估计评估我们这些情绪化的、灵巧的、狡猾的人类，它太呆板太非人化了。我们不同意这样的说法。在通过声音语气、面部表情来识别一个人的情绪状态方面，机器学习系统已经处于或者已经超越了人类水平的表现。有些系统甚至能识别世界最顶级的扑克选手是否在虚张声势。这是一个非常精细的工作，但它不是魔法。它需要知觉和认知，这正是机器学习现在正变得越来越强大的地方。

讨论人工智能的极限，最好从毕加索开始，毕加索通过对计算机的观察得出结论：“它们没有用，它们只能给你答案罢了。”事实上，计算机当然不是没有用，但是毕加索的观察依然提供了某些洞见。计算机是用来回答问题的设备，而不是提出问题的设备。那就意味着，企业家、创新者、科学家、创造者和其他那些寻找下一个问题与机会的人，那些探索新领域的人，他们依然至关重要。

我们认为，在这个超级强大的机器学习时代，对人类智慧来说，最大最重要的机遇在于两个领域的交叉：弄清楚下一步要解决什么问题，说服很多人解决这个问题，一起去寻求解决方案。这也是对“领导力”的一种合适的定义，而领导力已经在第二机器时代变得越来越重要了。

我们认为，人工智能，尤其是机器学习，这些是我们这个时代最重要的通用技术。这些创新对企业和经济的影响不仅仅体现在它们的直接贡献

中，而且还体现在它们启发互补创新的能力方面。通过更好的视觉系统、语音识别、智能解决问题系统，以及由机器学习所提供的很多其他功能，新的产品和流程正在成为现实。

虽然预测具体地哪个公司会在新环境中居于统治地位很难，但一个通用的原则很明晰，那就是：最为灵活的、有适应能力的公司和经营管理者会走向繁荣。能迅速感知到机遇，并对此有所反应的组织，终将会在人工智能这片热土上占据优势。所以成功的策略就是，乐于快速实验并学习。如果管理者们没有在机器学习领域开展实验，那么他们就没有做好自己的工作。在未来的十年时间里，人工智能并不会取代管理者，但是，那些善用人工智能的管理者将会取代那些没有这样做的人。



上图：仔细凝视，你将会看到算法中的人类；更仔细地凝视一会儿，你将会看到智能中的算法。

## 作者简介

**Erik Brynjolfsson** 是麻省理工学院数字经济先驱，斯隆管理学院的教授，数字商务中心主任。

**Andrew McAfee** 是麻省理工斯隆管理学院数字经济中心的首席研究科学家和副主任。

二人合著有《第二次机器革命》（The Second Machine Age）。

# Google 研究人员提出在移动设备上运行神经网络的新技术

作者 Roland Meertens 译者 盖磊



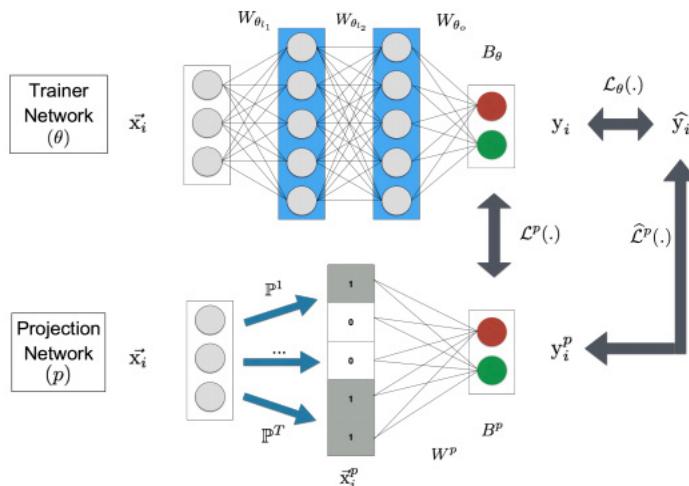
近期不少企业纷纷发布了使用深度神经网络的应用。神经网络需要做大量的计算，鉴于此，它们通常运行于具备 GPU 计算服务器的 SaaS 上。但是一些应用需要在没有因特网连接条件下运行、计算速度必须很快并且结果可靠、或是存在隐私上的考虑，这时不可能采用运行于服务器上的神经网络。

最近，多家企业宣布正致力于实现运行于移动设备上的神经网络。Apple 已经在 WWDC 2017 大会上发布了 [CoreML 平台](#)。Google 正在实现称为“Tensorflow Lite”移动设备通用 Tensorflow 工具集。Google 还发布了一些已预先训练的图像识别模型，开发人员可以根据自身需求在效率和准确性上做出 [权衡](#)。

虽然开发人员可以在移动设备上运行他们的神经网络算法，但在构建

快速神经网络应用上的可选方法依然有限。一类方法需要开发人员降低他们的神经网络规模，这通常会导致准确性的降低。另一类方法是在训练整个神经网络后降低浮点精度，这对性能的影响难以估量。还可以使用其它一些先期算法，例如 Facebook 的 AI 研究负责人 Yann Le Cun 提出的 [OBD 算法](#)（Optimal Brain Damage）。这些神经网络推理优化方法都没有得到广泛的采用。

为此，Google 研究人员 Sujith Ravi 提出一种新方法，即同时训练两个神经网络。其中一个是完备的神经网络，称为“训练网络”。另一个神经网络占用内存很小，表示了输入和训练网络中间结果，称为“投影网络”。它使用了高效函数，并学习自训练网络。一旦两个神经网络都被训练好可供使用，较大的网络依然运行在服务器上，较小的高效网络可被用户下载到智能手机上。



Sujith Ravis 已将论文提供在 [ArXiv 上](#)。论文中还对多个常用数据集上得分更好所需的比特位数进行了讨论。

# QCon 全球软件开发大会2017 [上海站] 上海宝华万豪酒店 | 10.17~19

9折优惠中  
9月17日前报名立减680元

团购报名更多优惠

扫描访问大会官网获取更多内容  
购票热线: 010-84782011



## 专题: 硅谷人工智能与云计算技术

多位来自硅谷互联网公司具有实战经验的架构师和技术专家来深度探索各种人工智能前沿技术和在各个系统中的应用。



梁真 Airbnb 机器学习工程师

《BOT: Boosting 决策表》



罗昶 Google Tech Lead/Senior Software Engineer

《用谷歌人工智能造聊天机器人》



Eric Kim LinkedIn 数据基础设施团队高级经理

《Nuage——LinkedIn如何使其分布式数据系统更易用》

## 专题: 大数据实时流计算与人工智能

随着互联网软硬件的飞速发展，传统行业开始向各种新经济类型转型。这些新经济往往依托于两个核心技术：大数据计算和人工智能。



黄明 腾讯 数据平台部T4专家

《方圆并济：基于 Spark on Angel 的高性能机器学习》



伍翀 阿里巴巴 资源研发工程师

《Blink SQL 和 Table API 在阿里巴巴的大规模应用》



付翔 Uber 实时数据平台架构师团队技术主管

《Realtime data analytics platform @ Uber》



翟佳 Streamlio 核心创始成员

《Feron 的 Exactly-Once 实现》



梁义(毅行) 阿里巴巴 高级搜索研发专家

《举重若轻——阿里实时机器学习平台 Porsche 介绍》



李友林 Facebook 工程部高级技术经理

《Facebook 实时数据连接及模型训练系统的演进》

## 专题: 快速变化的互联网架构

本专题聚焦于互联网圈超快速发展的公司在业务架构进行的各种创新和尝试，这些创新和尝试确保了现在公司业务架构可以持续领先于业务的发展，为业务提供稳定的技术大后方。



王兴朝 携程 研发总监

《携程第四代架构之软负载SLB实践之路》



许欣芃 恒生 贯深技术专家

《恒生中间件如何助力证券经纪业务发展》



许洋波(伏魔) 阿里巴巴 架构师 / 技术专家

《阿里巴巴集团千亿级别的店铺系统架构平台化技术实践》



陈霖 百度外卖 资深架构师

《百度外卖基础服务体系演进过程》



付强 腾讯 QQ空间后台开发Leader

《亿万图片服务平台——QQ 空间海量照片服务平台架构设计与实践》

## 专题: 团队管理

在当今的互联网+时代尤其明显，我们就缺一个CTO，我们就差一个饱满的产品经理，然而现实却并不那么温柔。



乔梁 腾讯 高级管理顾问

《从“作坊”到“专业”，团队管理三要点》



张灿 百度外卖 研发中心总监

《向前一步——年轻技术管理者的涅槃重生》



张浩 情了么 副总裁

《我所经历的中美互联网公司管理之异同》



黄俊伟 腾讯 CTO / 技术 VP

《由创业到独角兽，技术团队如何从跟随业务到驱动业务》

更多专题请访问官网 2017.qconshanghai.com

# InfoQ 中文站 2017迷你书



## 架构师 月刊 2017年8月

本期主要内容：Google 发布 IPv6 的应用情况；Java 老矣，尚能饭否？当 DevOps 遇见 AI，智能运维的黄金时代；学会思考，而不只是编程；为什么 AI 工程师要懂一点架构？整洁代码之道——重构



## 架构师特刊 大前端

本期主要内容：当我们在谈大前端的时候，我们谈的是什么；如何落地和管理一个“大前端”团队？



## 顶尖技术团队访谈录 第九季

本次的《中国顶尖技术团队访谈录》第八季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



## 架构师特刊 用户画像实践

本电子书中几个作者介绍一个公司如何从无到有的搭建用户画像系统，以及其中的技术难点与实际操作中的注意事项，实为用户画像的实操精华之选，推荐各位收藏阅读。