

架构师

ARCHITECT

|11月刊|

热点

大数据凉了？不，
流式计算浪潮才刚刚开始





▶ 聚焦

- 高性能业务架构
- 数据基础平台构建技术
- 短视频架构和算法
- 数据工程 & 大数据智能处理
- 区块链技术实践
- 智能高效运维
- 微服务治理
- 数据驱动产品和用户增长
- 金融技术框架
- 大前端技术
- 信息安全与隐私保护
- CTO技术选型思维

▶ 实践

[菜鸟网络](#) / 菜鸟仓储物流平台架构演进及技术挑战

[Pinterest](#) / Build a dynamic and responsive Pinterest on AWS:
a system engineer's perspective

[京东](#) / 分布式BaaS的设计与实践

[Netflix](#) / Netflix API Gateway that handles 2M requests per second

[LinkedIn](#) / 大规模机器学习在LinkedIn预测模型中的应用

[满帮集团](#) / 货车帮云原生平台架构设计思路和实践

[阿里巴巴](#) / 深度学习在智慧餐厅中的应用

会议：2018年12月07–08日

培训：2018年12月09–10日

地址：北京·国际会议中心

9折报名

团/购

▶ 出品人

孙海波

京东
大数据与智慧供应链事业部
研发负责

薛君凯

LinkedIn
Senior Software Engineer/
Apache Helix PMC& Committer

马晋

百度
网页搜索部主任研发架构师

李鑫

天弘基金（余额宝）
移动平台技术总监/首席架构师

▶ 分享嘉宾

Xiuduan Fang

Google
Staff Software Engineer
&TLM

谷雪梅

菜鸟网络
CTO & 副总裁
技术产品负责人

郭平（坤宇）

阿里巴巴
高级技术专家

张立理

百度 资深前端工程师
&百度ECOMFE、
FE-CMC现任主席

马恩驰

阿里巴巴
认知实验室算法专家

边剑

新浪微博
平台架构-技术专家

胡新

LinkedIn
Tech leader and Architect

刘春伟

美团点评
高级技术专家

徐宏亮

Uber
Senior Software Engineer

报名，立减680元

/享/更/多/优/惠



联系我们：

电话：17326843116（灰灰）

微信同号

CONTENTS / 目录

热点 | Hot

微软向 Linux 社区开放 60000 多项专利

Next.js 7 发布，构建速度提升 40%

Oracle 推出轻量级 Java 微服务框架 Helidon

理论派 | Theory

使用反应式领域驱动设计来解决不确定性

推荐文章 | Article

大数据凉了？不，流式计算浪潮才刚刚开始

观点 | Opinion

Serverless 基础架构下，运维人员将何去何从？

特别专栏 | Column

Linux 4.1 内核热补丁成功实践



架构师 2018 年 11 月刊

本期主编 陈利鑫

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

卷首语

信息传递 4.0：新时代推荐系统的机遇与挑战

作者 微博算法工程师 苏传捷

近几年来，推荐系统炙手可热。推荐系统在社交媒体、资讯阅读、电子商务等各类互联网产品中起到了越来越重要的作用。本质上，推荐系统是一种新的组织信息、传递信息的方式。这种方式火热的根本原因是网络信息指数级增长和信息多媒体化。

信息组织和传递方式大概分为下面4代版本。

- 1.0版本：导航页面。在互联网发展初期，网络上的信息很少，所以简单的导航页面就可以满足用户对网络信息获取的需要。
- 2.0版本：门户网站。随着网民的增加、原来越多的专业作者和编辑在网络上生产内容，这些内容包罗万象，有关于汽车的、有关于体育的。体育里面还有分关于篮球的、关于田径的等等。因此，门户网站这种以树形结构组织网页、传递信息的方式开始爆发，得到用户的青睐。
- 3.0版本：检索系统。随着博客、论坛、垂直网站这些UGC（用户产生内容）平台模式的崛起，大量的信息被非专业人士产生。这些信息大多用来表达自我，具有语言口语化、内容不确定等特点，很难被清晰地分门别类。这时，检索系统闪亮登场。用户无需知道自己想要的信息属于什么类目，只需要把自己的需要输入

在一个简单的搜索框里，就可以得到想要的信息。

- 4.0 版本，全面进入移动互联网时代，每一个拥有智能手机的人都可以轻松产生互联网信息，而且包含更多的图片、声音、视频等更多模态类型的信息。在这个大数据时代，用户甚至很难用语言表达自己想看什么样的信息，于是，推荐系统迎来了其前所未有的机遇。

正如 Kevin Kelly 的著作《必然》一书中所讲，“过滤”是世界发展的必然。推荐系统就是一种非常直观的“过滤”机制。可以预料，互联网产品的“智能推荐化”也是未来各自升级的重要路线，同时“智能推荐化”完成的好坏也会影响竞品竞争的结果。从系统架构来看，现如今推荐系统面临着三大挑战：超大规模数据、用户实时化反馈、多模态信息。下文会分别从这个三个方面讨论。

推荐系统面临的第一挑战是超大规模的数据。大规模的数据来源于三个方面：网络信息数量、用户数量和用户交互。前文已经提到，网络信息数量是指数级增长的。用户数量虽然有人口上限，但是由于互联网产品的垂直化，动辄过亿用户量的产品已不再是少数。另外，随着前端人机交互技术的发展，用户在产品交互次数已经明显增加。大规模的数据对推荐系统的挑战主要体现在大数据的存储、处理与分析，大规模机器学习系统两个部分。大数据存储、处理与分析已经有主流的开源分布式工具，比如 HDFS、Hadoop、Spark。它们已经是业界标配，可以满足大部分公司大数据处理的需求。另一方面，大规模的机器学习架构大致分为三类解决方案：基于 Map-Reduce 的方案、基于参数服务器（Parameter Server）的方案、基于 All Reduce 的方案。其中，基于参数服务器的方案目前占据主流地位，但是没有垄断性的标配工具。这是由于机器学习任务的多样性，与各个公司业务数据的独特性造成的。很多公司、开源组织或个人相继发布了开源分布式机器学习框架，比如 MLlib、Angel、Multiverso、TensorFlow、MXNet 等等。由于推荐系统的业务场景和机器学习算法的选择多种多样，所以开源机器学习框架并没有明显的高低之分，适合业务自

身发展阶段和算法选择的就是最好的。在推荐系统构建初期，算法通常会选择线性模型，这时候使用Spark的MLlib就是一个比较好的选择。它随Spark一起安装，社区相对比较成熟，方便开发人员学习、应用和维护。但如果是重量级的超大规模机器学习，就需要根据自身业务的特点，自主研发自己的机器学习平台，以便在存储、计算、通信等多个方面进行细致的优化。

推荐系统面临的第二个挑战是用户反馈实时化。为了提供更好的用户体验，推荐系统需要实时获取到用户的行为，快速处理数据、进行在线机器学习，最终近乎实时地反馈到下次的推荐服务中。要完成这样的任务，需要从前端到后端整个链路的架构支持。在客户端，需要实时获取用户的操作并实时传回，这需要消息队列（例如，Kafka、ZeroMQ、RabbitMQ等）的有力保障。收到数据后，需要并行的流式处理框架来进行实时处理，常用的有Storm、Spark Streaming、Flink等。随后，流数据处理框架对回传的数据进行清洗、计数、计算特征等处理，另外，计算的结果通常需要立即保存，以备机器学习系统使用。这时，基于内存的、支持快速读写的NoSQL数据库可以发挥作用，目前工业界比较成熟应用的有Redis、Memcache等。由于目前的推荐系统大多是基于机器学习模型的，所以这里的实时性要求必须由在线机器学习系统来保障。机器学习算法，特别是深度学习算法对计算效率的要求非常高，所以通常会对数据进行采样以减少训练数据规模。但与此同时还要保证模型的效果不会打折扣，所以在线实时机器学习系统的研发和算法研发通常是密不可分的。最后，当短时间内用户再次发出请求时，更新后的模型返回给用户新的推荐结果，完成闭环。一个实时推荐系统要求闭环中的每一个环节都实时高效，必须没有短板，保障整个数据流通畅、高效。

推荐系统面临的第三个挑战是多模态数据。多模态是一个比较新兴的名词，其大致的意思是文字、语言、图像、视频这些不同的内容形式共同表达语义。显而易见，互联网上的内容已经从文本占主体演进到了多媒体混合形态占主体，比如近两年，随着手机摄像的快速发展以及网络成本逐

渐降低，涌现出一批短视频产品。在目前的大多数推荐系统中，文本、图片、视频通常被分别处理后送入机器学习系统，分别训练出模型。最后，推荐系统会综合应用这些子模型。而多模态数据要求机器学习推荐系统可以同时处理包含多种模态形式的内容，通过联合学习来提升算法效果。但由于图像等多媒体特征抽取速度比较慢，很容易成为瓶颈，所以架构会面临巨大的挑战。比如，传统参数服务器会将数据随机分配到各个worker。但对于多模态数据，这种分配方式是不合理的。因为不同类型的数据适用的机器类型不同，图片、视频需要GPU机器才能高效计算，而文本和其他浮点型统计数据则使用CPU会比较节约成本。显然，这要求系统架构研发强依赖于模态类型的分布和算法研发的设计方案。

综上所述，在未来，推荐系统一定会在各种互联网产品中发挥出非常积极的作用。但与此同时，仍然有各种巨大的挑战需要被克服。

微软向 Linux 社区开放 60000 多项专利： 对开源微软是认真的

作者 张婵



10月10日，微软在博客中宣布正式加入开放创新网络（Open Invention Network, 简称“OIN”），向所有开源专利联盟的成员开放其专利组合。

微软的加入意味着，旗下60000多项专利将免费开源给Linux系统，帮助其发展。这60000多项已授权的宝贵专利产品组合（Windows和桌面应用程序代码的遗留例外）几乎是微软所拥有的一切了，这也意味着微软基本上同意向其他所有OIN成员授予其整个专利组合的免版税和不受限制的许可。

关于 OIN

OIN是由IBM、Red Hat等公司在2005年创建的专利池，致力于通过收

购与免费提供专利来推广Linux与促进全球技术创新，帮助公司管理专利风险。OIN专利许可和成员交叉许可对于加入OIN社区的任何人都可免费获得。甲骨文、Google、蚂蚁金服等数百家公司随后也都加入OIN中。

在OIN成立之前，许多开源许可证只明确涵盖了版权利益，但是对专利保持沉默。OIN的建立初衷是在涵盖Linux系统技术的成员公司之间建立自愿的专利交叉许可系统来解决这一问题。OIN还积极收购专利，以帮助保护社区，并提供有关开源知识产权的教育和建议。如今，通过首席执行官Keith Bergelt及其董事会的管理，OIN已经为全球约2,650家公司提供了一个许可平台。被许可人包括个人开发商和初创公司，以及一些全球最大的技术公司和专利持有者。

对开源，微软是认真的

OIN的首席执行官Keith Bergelt在采访中说道：微软开放的60000多项专利就是微软所拥有的一切，它涵盖了与旧的开源技术相关的一切，如Android，Linux内核和OpenStack；以及更新的技术，如LF Energy和HyperLedger，以及它们的前身和后续版本。”

在一次对话中，微软公司副总裁兼首席知识产权（IP）律师Erich Andersen（即微软的顶级专利人士）说：“我们将整个专利组合贡献给Linux系统。其中不仅仅包含Linux内核，还包括其他建立在它之上的东西。”

微软在博文中提到，他们知道微软加入OIN的决定可能会让一些人感到惊讶：微软和开源社区之间在专利问题上一直存在摩擦，这已不是什么秘密。对于那些关注微软发展的人来说，微软希望这个公告能向大家传达出微软是一个倾听客户和开发人员并坚定地致力于Linux和其他开源程序的公司，微软的这一举动是一个合乎逻辑的步伐。微软在用行动表明他们将通过开放专利而不是收取专利费用来获得更多收益。

在微软看来，开发人员不希望在Windows与Linux或.NET与Java之间进行选择 - 他们希望云平台支持所有技术。他们希望在任何设备上部署技

术，以满足客户需求。微软还认识到，通过开源流程进行协作开发可以加速创新。

在过去十多年中，微软一直在致力于使公司更加开放（你知道在2008年微软开源了部分ASP.NET吗？），现在已经成为世界上最大的开源贡献者之一，为超过2000个项目做出贡献，为Azure上的所有主要Linux发行版提供一流支持，并且还有开源主要的开源项目，如.NET Core，TypeScript，VS Code和Powershell。

微软也认识到每个开发人员都可以从开源社区中受益。开源是必不可少的，它不仅仅是代码，也是社区。微软不只是在网站上抛出代码，也公开发布他们的路线图；今年微软斥75亿美元巨资收购了GitHub，现在在GitHub上有20,000名微软员工，微软现在也是世界上最大的开源项目支持者。除了Windows桌面和桌面应用程序代码的主要遗留之外，微软已经是一家开源公司。

事实上，Azure客户广泛使用Linux，微软也在Windows之外，向Linux提供SQL Server。此次加入OIN后，可以消除一大部分吐槽微软对Linux和开源事业是否真心的质疑了。

在9月的一次采访中，微软云计算和企业集团执行副总裁Scott Guthrie曾说过，微软经历了“根本性的观念变革”。“我们曾经和开源社区不太友好，”Guthrie说。但你应该“看看我们过去五六年来的行动……最终，我们通过行动表明我们对开源是认真的。”

微软经过多年的内部变革和深思熟虑，在其业务模式和软件开发方式方面做出了根本性的改变。最终，开源赢了，微软现在是一家成熟的开源公司。

总结

加入OIN反映了微软的专利实践与公司对Linux和开源界的态度一致。两年前，微软开启了Azure IP Advantage等计划，将微软的赔偿承诺扩展到支持Azure服务的开源软件。此后微软一直在积极拥抱开源的道路上：

微软与Red Hat及其他协作者一起将GPL v.3“治愈”原则应用于GPL v.2代码，最近微软加入了LOT Network，LOT Network是一个不断发展的非营利性社区，由谷歌、Dropbox等数家科技公司联合创办，是一个专门用于对抗专利流氓的组织。

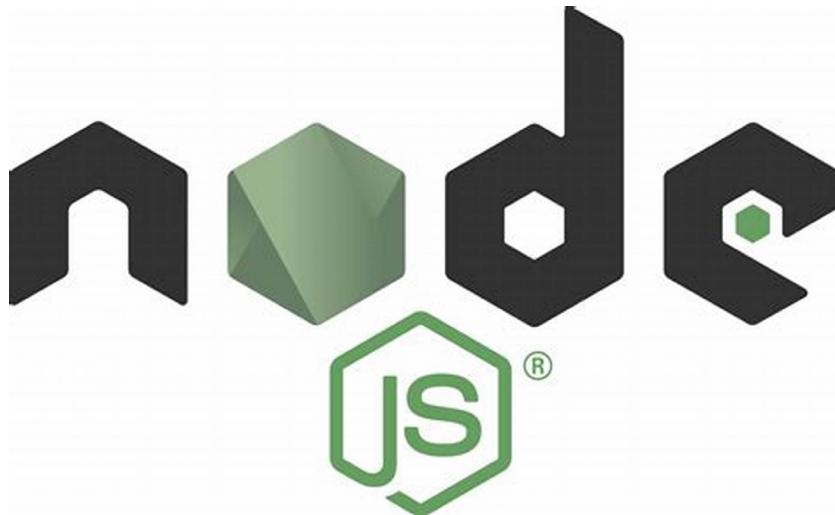
微软的博客中称，现在，当微软加入OIN时，他们相信微软将能够比以往更多地帮助保护Linux和其他重要的开源工作负载免受专利问题的侵害。微软也希望其加入OIN的决定能吸引更多其他公司加入到OIN，使得许可网络更加强大，以利于开源社区，并通过开源软件鼓励创新。

截至2014年，微软从其Android专利中获得了约34亿美元盈利。三星独自向微软支付了10亿美元，用于授权其Android专利。微软如今向安卓手机厂商收取的专利费可能也会随之走向终结。

除此以外，微软这项举措还表明，与移动设备行业不同，云计算领域很可能不会发生毫无意义、代价高昂的专利战。企业云计算严重依赖Linux和其他开源技术，任何涉及开源技术的专利纠纷都会给整个行业造成严重冲击。

Next.js 7 发布，构建速度提升 40%

作者 Jacob Clark 译者 谢丽



Next.js团队发布了其开源React框架的[7版本](#)。该版本的Next.js主要是改善整体的开发体验，包括启动速度提升57%、开发时的构建速度提升40%、改进错误报告和WebAssembly支持。

Next.js是一个React框架，它的主要目标是在生产环境中提供出色的性能和良好的开发体验。为了提供这种良好的开发体验，Next.js支持服务器端渲染、代码分割和客户端路由。

Next.js以JavaScript生态系统中的许多标准行业工具为基础构建，比如Babel和Webpack，而版本7带来了这些工具的最新版本。这些升级以及一个新的增量编译缓存意味着Next.js编译现在快了40%，一个基本应用程序的编译时间将从304ms减少到178ms。

随着Webpack升级，得益于新增的.mjs支持，Next.js 7允许捆绑所有常见的JavaScript模块，如CommonJS、AMD和ESM，同时也支持EcmaScript, JSON和WebAssembly模块。

Next.js 7还捆绑了最新版本的Babel，它提供了对TypeScript、片段语法和尚处于试验阶段的“自动填充（poly filling）”的支持。

Next.js 7的初始有效载荷大小降低了多达7.4%，一个在Next.js以前的版本中大小为1.62kB的文档变成了到1.50kB。这些改进源于Next.js团队删除了某些HTML元素并缩小了一些内联脚本。

Next.js 7的另一个主要改进是对React Context API的支持。Context API是一种跨React组件共享数据的方式，而且不必每次都显式共享。得益于Next.js能够在页面之间共享代码，这将使其内存使用减少16%。

Next.js 7支持模块的动态导入；之前，由于Next.js使用自己的导入功能，这是不可能的。现在，他们已经删除了这个功能，并且支持Webpack自带的默认导入功能，允许动态导入、命名和绑定文件。

Next.js在社区内得到了广泛好评。Reddit用户reacttricks说，“在过去一年半的时间里，我所有的项目都在使用Next.js，我建议每个人都尝试一下。”其他的反馈包括对接下来会发生什么的困惑。theineffablebob问：

“Next是一个包含了让站点启动和运行所需的所有内容的框架吗？它有点像那些样板文件？”Nextjs.org将自己视为JavaScript和React世界的PHP，nextjs.org上有这样一句话：“考虑一下如何用PHP创建web应用。创建一些文件，编写PHP代码，然后简单地部署它。我们不必太考虑路由问题，应用程序是在服务器上渲染的。

感兴趣的读者可以从[Next.js网站](#)上下载最新版本。

Oracle 推出轻量级 Java 微服务框架 Helidon

作者 Michael Redlich 译者 谢丽



近日，Oracle推出了一个新的开源框架[Helidon](#)，该项目是一个用于创建基于微服务的应用程序的Java库集合。和[Payara Micro](#)、[Thorntail](#)（之前的[WildFly Swarm](#)）[OpenLiberty](#)、TomEE等项目一样，该项目也加入了MicroProfile家族。

Helidon最初被命名为J4C（Java for Cloud），其设计以简单、快速为目标，它包括两个版本：Helidon SE和Helidon MP。Helidon SE提供了创建微服务的三个核心API：Web服务器、配置和安全，用于构建基于微服务的应用程序，不需要应用服务器。Helidon MP支持用于构建基于微服务的应用程序的MicroProfile 1.1规范。

Web 服务器

受NodeJS和其他Java框架的启发，Helidon的Web服务器是一个异步、

反应性API，运行在Netty之上。WebServer接口包括对配置、路由、错误处理以及构建度量和健康端点的支持。

下面的示例代码演示了如何启动一个简单的Helidon Web服务器，在一个随机可用的端口上显示“It works!”：

```
// 在一个随机可用的端口上启动服务器
public void startWebServerUsingRandomPort() throws Exception {
    WebServer webServer = WebServer
        .create(Routing.builder()
            .any((req,res) -> res.send("It works!" +
            "\n"))
        .build())
        .start()
        .toCompletableFuture()
        .get(10, TimeUnit.SECONDS);
    System.out.println("Server started at: http://localhost:" +
    + webServer.port() + "\n");
    webServer.shutdown().toCompletableFuture();
}
```

配置

配置组件[Config](#)加载和处理键/值格式的配置属性。在默认情况下，配置属性将从定义好的application.properties或application.yaml文件中读取，它们位于/src/main/resources目录下。

下面的示例代码基于前面的例子构建，它演示了如何使用Config，通过读取applications.yaml文件获得指定的端口启动Web服务器。

```
// application.yaml
server:
  port: 8080
  host: 0.0.0.0
// 在application.yaml预定义的端口上启动服务器
public void startWebServerUsingDefinedPort() throws Exception {
    Config config = Config.create();
```

```

        ServerConfiguration serverConfig = ServerConfiguration.
fromConfig(config.get("server"));

        WebServer webServer = WebServer
            .create(serverConfig, Routing.builder()
                .any((req,res) -> res.send("It works!" +
"\n"))
            .build())
            .start()
            .toCompletableFuture()
            .get(10, TimeUnit.SECONDS);

        System.out.println("Server started at: http://localhost:" +
webServer.port() + "\n");
        webServer.shutdown().toCompletableFuture();
    }
}

```

安全

类Security为身份验证、授权和审计提供支持。已经有许多用于Helidon应用程序的安全提供程序实现。有三种方法可以将安全性内置到Helidon应用程序中：从构建器、通过配置或者是前两者的结合。

下面的示例代码演示了如何构建Security实例、使用Config获取用户身份验证（使用加密密码）并显示服务器时间。

```

// application.yaml

http-basic-auth:

users:
    login: "mpredli"
    password: "${CLEAR=somePassword}"
    roles: ["user", "admin"]

Config config = Config.create();
Security security = Security.builder()
    .config(config)
    .addProvider(...)
    .build();

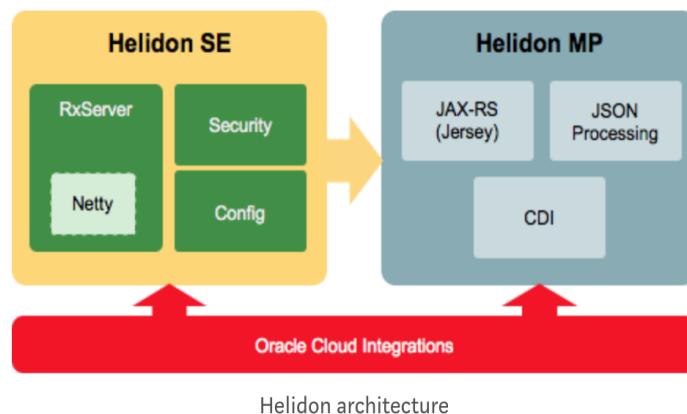
```

```
String user = config.get("http-basic-auth.users.login").  
asString();  
String password = config.get("http-basic-auth.users.password").  
asString();  
System.out.println("\n");  
System.out.println("INFO: user = " + user);  
System.out.println("INFO: password = " + password);  
SecurityTime time = SecurityTime.builder().build();  
time = security.getServerTime();  
System.out.println("INFO: server time = " + time.toString());  
System.out.println("\n");
```

GitHub提供了更详尽的安全示例。

Helidon 的架构

下面的架构图显示了Helidon SE和Helidon MP的关系。



下图说明了Helidon SE和Helidon MP所属的微服务框架类别。



入门指南

Helidon提供了快速[入门示例](#)来演示Helidon SE和Helidon MP之间的区别。

下面的Maven和Java命令将生成并打包Helidon SE示例，使用Helidon的Web服务器创建一个REST服务。

```
$ mvn archetype:generate -DinteractiveMode=false \
-DarchetypeGroupId=io.helidon.archetypes \
-DarchetypeArtifactId=helidon-quickstart-se \
-DarchetypeVersion=0.10.1 \
-DgroupId=io.helidon.examples \
-DartifactId=quickstart-se \
-Dpackage=io.helidon.examples.quickstart.se

$ cd quickstart-se
$ mvn package
$ java -jar target/quickstart-se.jar
```

下面的Maven和Java命令将生成并打包Helidon MP示例，使用MicroProfile的JAX-RS API创建一个REST服务。

```
$ mvn archetype:generate -DinteractiveMode=false \
-DarchetypeGroupId=io.helidon.archetypes \
-DarchetypeArtifactId=helidon-quickstart-mp \
-DarchetypeVersion=0.10.1 \
-DgroupId=io.helidon.examples \
-DartifactId=quickstart-mp \
-Dpackage=io.helidon.examples.quickstart.mp

$ cd quickstart-mp
$ mvn package
$ java -jar target/quickstart-mp.jar
```

一旦服务器开始运行，就可以执行一下的命令（见下页图）。

在GitHub上可以找到整个Helidon项目。

Oracle的高级软件开发经理Dmitry Kornilov向infoQ介绍了这个新项目。

```
Last login: Tue Oct  9 19:18:21 on ttys001
[mpredli01@Michaels-MacBook-Pro-4] [~]$ curl -X GET http://localhost:8080/greet
{"message": "Hello World!"} [mpredli01@Michaels-MacBook-Pro-4] [~]$ 
[mpredli01@Michaels-MacBook-Pro-4] [~]$ curl -X GET http://localhost:8080/greet/Mike
{"message": "Hello Mike!"} [mpredli01@Michaels-MacBook-Pro-4] [~]$ 
[mpredli01@Michaels-MacBook-Pro-4] [~]$ curl -X PUT http://localhost:8080/greet/greeting/Hola
{"greeting": "Hola"} [mpredli01@Michaels-MacBook-Pro-4] [~]$ 
[mpredli01@Michaels-MacBook-Pro-4] [~]$ curl -X GET http://localhost:8080/greet/Mike
{"message": "Hola Mike!"} [mpredli01@Michaels-MacBook-Pro-4] [~]$ 
[mpredli01@Michaels-MacBook-Pro-4] [~]$
```

InfoQ：是什么给了甲骨文开发这个新微服务框架的启发？

Dmitry Kornilov：有关Helidon的工作已经开始一段时间了。当创建云服务的微服务体系结构开始变得非常流行时，开发体验也需要改变。Java EE是一种稳定的技术，但是它有很多遗留代码。我们没有在Java EE上构建微服务，我们意识到，我们需要一个从头开始设计的构建微服务的新框架。Helidon就是这样出现的。

InfoQ：与OpenLiberty、Thorntail、Payara Micro和TomEE等其他MicroProfile实现相比，Helidon有什么独特之处？

Kornilov：Helidon不仅仅是一个MicroProfile实现。它有两个版本：Helidon SE和Helidon MP。

Helidon SE构成了Helidon的核心。它是一组轻量级的库，其中的库可以单独使用，但如果一起使用，就可以满足开发人员创建微服务的基本需求：配置、安全和Web服务器。它带来了一种开发人员喜欢的更现代的反应性方法。我们总是尽力明确：不使用注入“魔法”，使Helidon SE应用程序易于调试。没有特殊的jar格式，没有特殊的类加载器。你的应用程序只是一个普通的Java SE应用程序。这也意味着，它与所有IDE兼容，不需要特殊的插件。

Helidon MP是我们的MicroProfile实现，它以Helidon SE为基础构建——它不是派生自某个应用服务器。因此，没有部署模型，没有Java EE打包，没有你不需要的额外的东西。

InfoQ：为什么实现的是MicroProfile 1.1规范，而不是一个更新的版本？

Kornilov：Helidon的开发在一段时间之前就开始了，我们决定坚持

使用当时最新的MicroProfile版本。我们正在不断地改进Helidon，对新的MicroProfile版本的支持很快就会到来。

InfoQ：接下来，尤其是在**Jakarta EE**支持和**MicroProfile**规范较新版本的支持方面，**Helidon**将开展哪些工作？

Kornilov：我已经提到过，我们正致力于对MicroProfile较新版本的支持。当新的Jakarta EE 规范出现时，我们将参与它们的开发并在Helidon中支持它们。此外，我们计划向Helidon添加Oracle Cloud集成特性、HTTP客户端支持、项目启动器Web应用，并不断改进我们的示例和文档。

相关资源

[Helidon起飞](#) (Dmitry Kornilov, 2018年9月7日)

[Oracle发布新的Java微服务框架](#) (John Waters, 2018年9月10日)

[微服务从入门到部署 第一部分：Helidon入门](#) (Todd Sharp, 2018年10月3日)

使用反应式领域驱动设计来解决不确定性

作者 Vaughn Vernon 译者 无明



我认为应该通过领域驱动设计的方式来开发软件。从Evan最初出版“Domain-Driven Design: Tackling Complexity in the Heart of Software”开始，我们经历了一趟艰难的旅程，已经走了很长的一段路。现在有一些关于领域驱动设计的技术大会，人们对领域驱动设计的兴趣程度也与日俱增，包括业务人士的参与，这是非常关键的。

如今，反应式变得越来越重要，稍后我会解释为什么它获得人们的关注。我认为真正有趣的是，DDD在2003年的使用或实现方式与今天使用DDD的方式截然不同。如果你已经读过我的红皮书“Implementing Domain-Driven Design”，那么你可能已经熟悉这样的一个事实：我在这

本书中提到的有界上下文是单独的过程，需要进行单独的部署。然而，在Evan的蓝皮书中，有界上下文在逻辑上是分开的，但有时候部署在同一部署单元中，可能是在Web服务器或应用程序服务器中。现如今，越来越多的人采用DDD是因为它与单独部署机制（例如微服务）不谋而合。

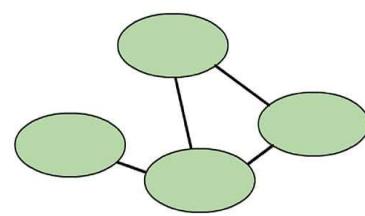
领域驱动设计的本质并没有变——在有界的上下文中建模一门通用语言。那么，什么是有界上下文？基本上，有界上下文背后的想法就是在清晰地划定模型之间的边界。围绕领域模型的边界让边界内的模型变得非常清晰，让概念、模型的元素以及团队（包括领域专家）对模型的思考方式都具备了非常明确的含义。

你会发现团队内形成了一种通用语言。例如，当有人在讨论问题时提到“产品”，他们确切地知道，在这种情况下产品代表了什么意思。而在另一个上下文中，产品可能具有不同的含义。产品可以在有界的上下文中共享身份，但一般来说，在另一个上下文中（至少）具有略微不同的含义，甚至可能具有非常不一样的含义。

我们意识到，并不存在一种实用的方法可用来建立规范的企业数据模型，模型中的每个元素都代表了企业的每个团队希望如何使用它们。这种事情是不可能发生的。因为差异总是存在的，很多时候，因为存在太多差异，导致一个团队难以使用另一个团队创建的模型。这就是为什么我们要使用带有通用语言的有界上下文。

在一个团队中，通过一种通用语言对一个实体进行非常明确的定义，一旦理解了这一点，你就会意识到，其他团队也会通过类似的方式开发其他模型。或许开发相同模型的同一个团队也可能负责开发其他模型。在某些情况下，你可能拥有多个有界上下文，因为我们无法在单个企业中或单个系统中为我们将要使用的每个概念定义含义。

Context Mapping



@VaughnVernon

图1 上下文映射

鉴于我们有多个上下文和多种语言，我们必须让它们之间进行协作和集成。为此，我们使用了一种称为上下文映射的技术或工具。

在这张图中，有界上下文之间的线表示上下文映射，可以称其为翻译。如果其中一个有界上下文使用一种语言，与其相连的上下文使用了另一种不同的语言，那么你需要在两种语言之间做些什么才能让两个模型之间相互理解？答案是翻译。通常，我们需要在模型之间进行翻译，这样才能让每个模型保持纯净。

在创建上下文映射时，不要限制连线的含义。上下文映射可能涵盖技术集成、风格或技术，但定义上下文之间的团队关系也非常重要。是使用RPC还是REST并不重要。与谁集成比怎样集成更重要。

针对不同类型的关系，有各种上下文映射工具，包括合作伙伴关系、客户与供应商关系或随从（conformist）关系。在合作伙伴关系当中，一个团队会对另一个团队的模型有很多了解。客户与供应商关系在模型（一个是上游模型，另一个是下游模型）之间引入了一个反腐（anti-corruption）层。上游模型需要进行反腐，因为它被下游模型消费。如果下游模型需要将某些内容发给上游，它将被翻译成上游模型，以便可以进行可靠而一致的数据交换，且具有明确的含义。

到目前为止，我所描述的战略设计实际上是域驱动设计的本质，因此也是最重要的部分。

在某些情况下，我们会小心翼翼地进行通用语言建模。如果你将战略设计比作使用宽笔刷画画，那么战术设计就是使用细笔刷来填充细节。

DDD 建模的一般性指南

根据我对提及DDD相关大会的观察以及我与团队进行的广泛的合作，我找到了一些有助于DDD建模的小技巧。我希望能够提供一些指导，让你走在正确的方向上，不至于跑偏。

我们必须记住，在进行建模时，特别是在战术上，我们需要领域专家

(而不只是程序员) 的帮助。但我们不能占用他们太多的时间，因为在团队中扮演领域专家角色的人通常忙于与业务相关的其他事务。

另外，我们要避免贫血领域模型。当你看到某些领域建模演示当中包含自动创建getter、setter、Equals()、GetHashCode()的注解时，请立即远离它们。如果我们的领域模型只是关于数据，那么这些演示可能会是完美的解决方案。但我们需要问一些问题，比如数据是如何产生的？我们如何让数据进入到领域模型中？它是基于业务和领域专家的心理模型进行表达的吗？getter和setter并不会明确指出模型的含义——它们只是用来移动数据。如果你正在考虑使用DDD，那么应该意识到getter和setter其实是你的敌人，因为你真正想要建模的是那些能够表示企业完成工作所需的方式的行为。

在建模时要非常明确。例如，你使用UUID来表示实体或聚合的业务标识符。使用UUID作为业务标识符没有任何问题，但为什么不将它包装在强类型的ID类型中呢？另一个不使用Java的有界上下文可能无法理解UUID是什么意思。你可能需要定义UUID ToString()，然后将字符串保存成其他类型，或者在有界上下文之间传递事件时需要对字符串进行翻译。

在使用BigDecimal时，为什么不考虑创建一个Money值对象。如果你使用过BigDecimal，你就该知道识别舍入因子是一个常见的痛点。如果我们让BigDecimal渗透到我们的模型中，那么我们该如何对金额进行舍入操作？解决办法是使用Money类型来标准化业务所说的舍入规范。

另一个小技巧是不要担心使用怎样的持久化机制，或者使用什么类型的消息传递机制。使用符合你特定服务级别协议的内容就可以了。根据实际的吞吐量和性能做出合理的选择，不要将事情复杂化。

反应式系统

至少在我的世界里，我已经看到了反应式系统的发展趋势。不仅微服务具备反应式，构建整个系统也是反应式的。在DDD中，有界上下文中也会发生反应式行为。反应式并非新鲜事物，Eric Evans在引入事件驱动

时就已涉及反应式。使用领域事件意味着我们必须对过去发生的事件做出反应，并将我们的系统带入融洽状态。

如果你对系统不同层的连接进行可视化，你就会看到这种重复的模式。无论是整个互联网、企业层面的所有应用程序，或是微服务中的单个参与者或异步组件，每一层都有很多连接和相关的复杂性。我们因此很多非常有趣的问题需要解决。我想要强调的是，我们不应该用技术来解决这些问题，而应该对它们进行建模。如果我们将开发云端微服务作为构建业务的手段，那么分布式计算就是我们所从事的业务的一部分。并不是说分布式计算形成了我们的业务（在某些情况下，它确实如此），而是我们通过分布式计算来解决问题。

我需要花一点时间来解释一些概念，一些开发人员将它们作为反对异步、并行、并发或任何其他与同时发生同个动作相关的技术的论据。人们经常引用Donald Knuth的话说，“过早优化是万恶之源”。但这有点断章取义。他的完整的说法应该是，“我们应该忽略细微的效率优化……过早的优化是万恶之源”。换句话说，如果我们的系统存在很大的瓶颈，我们应该先解决它们。

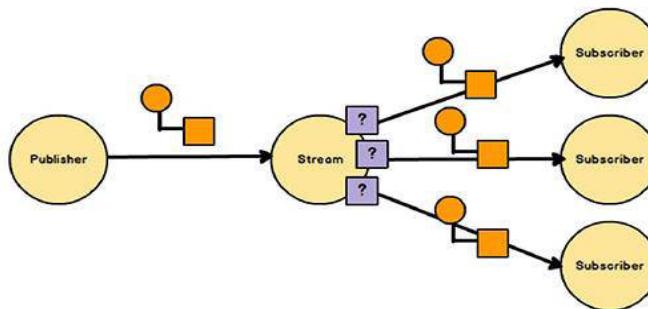
Donald Knuth还说了一些非常有趣的事情：“对计算机有一定了解的人应该至少知道底层硬件的含义。否则，他们写出来的程序将非常奇怪”。他的意思是说，我们需要通过我们今天编写软件的方式来利用硬件。

如果我们回到1973年看看处理器的制造方式，当时的晶体管数量非常少，而且时钟速度低于1MHz。根据摩尔定律，每隔几年我们就会看到晶体管数量和处理器速度翻倍。然而，到了2011年，时钟速度却开始下降。在过去，时钟速度翻倍需要一年半或两年的时间，而现在需要大约十年（如果不是更长的话）。晶体管的数量继续在增加，但时钟速度却没有。今天，我们拥有更多的处理器内核。我们拥有更多的内核，而不是更快的始终速度。那么我们应该如何使用这些核心呢？

如果你一直在使用Spring和Reactor项目（使用了反应式流），那么这

些基本上就是我们现在能够做到的事情。我们有一个发布器，这个发布器发布一些东西，我们称之为领域事件，也就是图2中的橙色方框。这些事件被传递给流的订阅者。

Reactive



@VaughnVernon

图2 反应式

请注意，流中的淡紫色方框上有问号。它们实际上是指策略，这些策略位于流和订阅者之间。例如，订阅者可能对其可以处理的事件或消息的数量有限制，策略为订阅者指定这些限制。关键是，发布者、流和三个订阅者是使用单独的线程运行的。如果这是一个大型复杂的组件，在任何时候线程都不能被阻塞。因为如果线程被阻挡，那么其他部分就会等待线程。我们必须确保内部实现也必须充分利用好线程。随着我们深入不确定性建模，这将变得更加重要。

微服务是反应式的。但当我们深入内部时，发现各种组件可以同时或并行运行。当从微服务中发布事件时，它们最终会在有界上下文之外发布到某种主题上（可能使用Kafka）。为了简单起见，我们假设只有一个主题。我们的反应式系统中的所有其他微服务都在使用发布到这个主题上的事件，并且在服务内部做一些反应式处理。

这就是我们想要达到的状态。当一切都在异步发生时，会发生什么？这带来了不确定性。

Reactive Systems

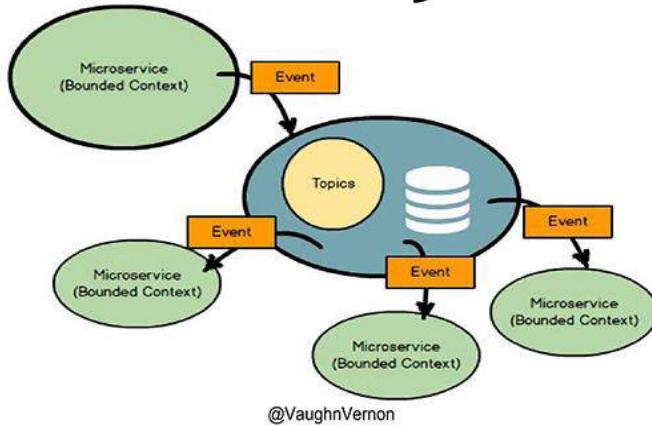


图3 反应式系统

欢迎不确定性

在理想情况下，当我们发布一系列事件时，我们希望这些事件按顺序传递，而且只需传递一次。每个订阅者将收到事件1，然后是事件2，接着是事件3，每个事件只出现一次。程序员已经被教会要小心翼翼地维持这种状态，因为这让我们对我们正在做的事情感到确定。然而，在分布式计算中，这种情况并不会发生。

微服务和反应式带来的不确定性首先是事件以怎样的顺序传递、是否会多次收到同样的事件，或者根本收不到事件。即使你在使用像Kafka这样的消息传递系统，而你认为这样就可以按顺序接收到事件，那么你是在自欺欺人。如果有任何消息可能出现乱序，那么你就必须做好应对所有消息都会出现乱序的准备。

我找到了一个对不确定性的简洁定义：不确定的状态。不确定性是一种状态，这意味着我们可以处理它，因为我们有办法推理系统的状态。最终，我们能够达到一种让我们感到舒适的确定状态，即使只持续一毫秒。蔓延的不确定性让事情变得不可预测、不可靠和风险重重。不确定性令人感到不舒服。

上瘾

大多数开发人员学会了所谓的开发软件的“正确方法”，让他们沉迷其中。如果你有两个需要相互通信的组件，可有将一个组件作为客户端，另一个组件作为服务器。但这并不意味着它必须是一个远程客户端。客户端将调用服务器的方法或函数。在发生调用时，客户端等待从服务器返回响应，或者可能抛出异常。不管怎样，客户端可以肯定自己会再次获得控制权。这种执行流程的确定性是我们必须处理的一种上瘾。

第二种常见的上瘾是对排好序且没有重复的事物的上瘾。当我们还是个小孩的时候，在学校里学习数数就是按顺序来的。当我们知道某些东西是按顺序排列时，我们会感觉良好。

另一个让我们上瘾的是数据库的锁。如果我的数据源中有三个节点，当我向其中一个节点写入时，我认为我已经将数据库牢牢地锁定。这意味着当获得成功响应时，数据应该已经持久地保存在所有三个节点上。但当你开始使用Cassandra时，它并不会锁定所有节点，那么你该如何查询尚未传播到集群中所有节点上的值呢？

所有这些都给我们带来了不舒服的感觉，我们必须学会应对它们。不过，感觉不舒服并不是不可以忍受的。

防御机制

因为我们沉迷于确定性、阻塞和同步，开发人员倾向于通过建立堡垒来应对不确定性。如果你正在基于有界上下文创建微服务，那么你会让所有东西在这个上下文中的都是阻塞、同步和非重复的。我们在这里构建了一个堡垒，所有的不确定性都存在于这个堡垒之外。

从基础设施层开始，我们创建了去重器和重排器，它们都来自企业集成模式。

去重器

如果事件1、事件2、事件1和事件3进入系统，当它们经过去重器时，

我们确定我们只会得到事件1、事件2和事件3。这似乎不是一个难以解决的问题，但在实际实现时需要考虑到其他一些问题。我们可能会使用缓存，但不能无限制地在内存中保存事件。这意味着需要使用数据库表来存储事件ID，并检查每个传入事件在之前是否已经出现过。

如果某个事件之前没有出现过，那么就往下传。如果之前已经出现过，那么就可以忽略它，对吧？那么为什么之前会出现过？是因为我们之前在收到它时没有对其进行确认吗？如果是这样，我们是否应该再次确认已经收到它了？我们需要保留这些ID多长时间？一周够吗？一个月怎么样？

现在让我们来谈谈不确定性。试图通过技术来解决这个问题可能非常困难。

重排器

假设我们看到事件3，然后是事件1，然后是事件4。出于某种原因，事件2迟到了很长一段时间。我们首先必须找到重新排序事件1、事件3和事件4的方法。如果事件2尚未到达，我们是否要关闭我们的系统？或许我们可以允许事件1通过，但我们有一些规则规定在事件2到达之前不能处理事件1。在事件2到达后，我们就可以安全地将所有四个事件放入应用程序中。

不管你选择的是哪一种实现，要解决这些问题都很难，因为存在不确定性。

最糟糕的是，我们还没有解决任何业务问题。我们只是在解决技术问题。但是，如果我们承认分布式计算现在已经成为业务的一部分，那么我们就可以解决这些存在于模型中的问题。

我们想说，“停止所有的东西，我现在准备好了”。但问题是根本就没有“现在”这种东西。如果你认为你的业务模型是一致的，它可能只能维持一纳秒，然后再次出现不一致。

例如Java中的LocalTime.now()，一旦你调用了这个方法，它就不再是

现在。当你收到返回的时间时，“现在”已经不存在了。

分布式系统的不确定性

这一切都把我们带回了分布式系统都是关于不确定性的事实。分布式系统将继续存在。如果你不喜欢它，就请离开这个领域，或许你可以去开一家餐馆。但是，你仍然需要面对分布式系统，只是不是你在开发它们。

对不确定性进行建模是非常重要的，因为这里有多个核心问题，也因为云的存在。

微服务很重要，因为这是每个人都在使用它。大多数人都向我学习领域驱动设计，因为他们认为这是实现微服务的一种好方法。我同意这一点。此外，企业希望摆脱单体系统。他们被困在单体系统的泥潭中，需要数月才能发布一个版本。

延迟也很重要。当你将网络作为分布式系统的一部分时，延迟很重要。

物联网也很重要。很多小型的便宜设备都通过网络进行交互。

DDD 和建模

我指的是好的设计、糟糕的设计和有效的设计。

你可以设计出非常好的软件，但却缺失业务需求。以SOLID为例，你可以按照SOLID原则设计出100个类，但却完全忽略业务需求。面向对象和Smalltalk的发明者Alan Kay说，对象真正重要的是在它们之间传送的消息。

在日语中，Ma的意思是“……之间的空间”。在这里，当然是指对象之间的空间。对象需要设计得足够好才能够发挥它们单一职责的作用。但这不仅仅是关于对象的内部，我们还需要关心对象的名称以及在它们之间传送的消息。这就是通用语言发挥作用的地方。如果你能够在模型中捕获到它，你的模型就会变得更有效，而不仅仅是好，因为你将可以满足业务需求。这就是领域驱动设计的意义所在。

我们想要开发反应式系统，我们接受不确定性是一种状态的事实，我们将开始解决问题。

Pat Helland在他的论文“Life Beyond Distributed Transactions, an Apostate's Opinion”中写道：“在一个不能相信分布式交易的系统中，必须在业务逻辑中实现不确定性管理”。在实现分布式事务很长时间之后，才有了这种认识。不过，在亚马逊工作期间，他确定，使用分布式事务无法实现他们所需的规模。

Helland在论文中谈到了活动。当两个伙伴实体中的一个发生状态变更，并触发一个描述状态变更的事件，另外一个实体收到这个事件，那么就可以说这两个实体之间发生了一个活动。但第二个实体什么时候会收到事件？如果收不到该怎么办？这种“假设”情况会很多，Pat Helland说这些“假设”应该由活动来处理。

在任意两个伙伴之间，每一方都必须管理来自另一方的活动。图4是我对Pat Helland的观点做出的解释。

Activity

```
public final class PartnerActivities {  
    private final Set<PartnerActivity> activities;  
    ...  
    public void record(Partner p, Activity a) {  
        activities.add(new PartnerActivity(p, a));  
    }  
  
    public boolean seen(Partner p, Activity a) {  
        activities.contains(new PartnerActivity(p, a));  
    }  
}
```

@VaughnVernon

图4 活动

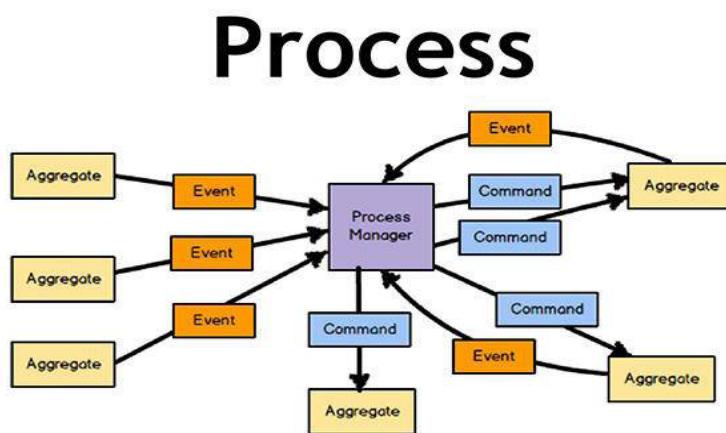
每个合作伙伴都有一个PartnerActivities对象，代表从相关实体那里收到的活动。当遇到一些直接针对我们的活动时，可以将它们记录下来。然

后，在未来的某个时间，我可以问，“我之前看到过这个活动吗？”这样既可以使用已经发生的活动，又可以检查之前是否已经看到过它。

这似乎很简单，但在长期存在的实体中会变得复杂。Pat Helland也说这种复杂性会急剧增加。为了跟踪活动，长期存在的实体（可能与很多合作伙伴一起）可能会导致大量实体聚集（DDD称之为聚合）。此外，出现这种情况的迹象并不明显——它并不会显示任何与业务本身有关的信息。

我认为，我们应该比PartnerActivity更进一步。我提议将其放在软件的核心——领域模型中。我们消除了基础设施层中的那些元素（去重器和重排器），并且在事件发生时让它们通过。我相信这将让系统变得不那么复杂。

在显式模型中，我们监听事件，并发送与业务操作相对应的命令。领域将聚合每个包含如何处理这些命令的业务逻辑，包括如何响应不确定性。事件和命令之间的通信由进程管理器来处理，如图5所示。虽然图5可能看起来很复杂，但实际的实现非常简单，并遵循反应式模式。



@VaughnVernon

图5 进程管理器

每个领域实体都负责根据收到的命令来跟踪状态。通过遵循良好的DDD实践，可以基于这些命令安全地跟踪状态，并通过事件溯源来持久

化状态变更事件。

每个实体根据领域专家做出的决定负责了解如何处理潜在的不确定性。例如，如果收到重复事件，那么聚合就会知道之前已经看到过这个事件，并知道如何做出响应。图6显示了处理这类问题的一种方法。当收到 deniedForPricing() 命令时，我们可以检查当前进度，如果我们已经看到过 PriceDenied 事件，那么我们就不会再发出新的领域事件，而是做出响应，表明已经看到过拒绝事件。

```
public final class Progress {  
    private final Set<Spec> specs;  
    ...  
    public Progress deniedForPricing() {  
        return withNewSpec(Spec.PricingDenied);  
    }  
    ...  
    private Progress withNewSpec(final Spec spec) {  
        if (!specs.contains(Spec.PricingDenied)) {  
            return ... ;  
        }  
        return Progress.ALREADY_SEEN;  
    }  
}
```

图6 处理重复消息

这个例子可能不会给人留下什么印象，但其实是有目的的。它表明，将不确定性视为领域的一部分意味着我们可以使用DDD实践让不确定性成为业务逻辑的另一个方面。而这就是DDD的意义所在——帮助管理软件核心的复杂性，例如不确定性。

作者简介

Vaughn Vernon是一名软件开发人员和架构师，在业务领域拥有35年的经验。Vaughn是领域驱动设计领域的专家，也是反应式系统的倡导者。他是vlingo/platform的创始人、首席架构师和开发者。他参与并教授DDD和反应式软件开发，帮助团队和组织挖掘业务驱动和反应式系统的潜力。

大数据凉了？不，流式计算浪潮才刚刚开始

作者 Tyler Akidau Reuven Lax 等

译者 巴真



导读：本文重点讨论了大数据系统发展的历史轨迹，行文轻松活泼，内容通俗易懂，是一篇茶余饭后用来作为大数据谈资的不严肃说明文。本文翻译自《Streaming System》最后一章《The Evolution of Large-Scale Data Processing》，在探讨流式系统方面本书是市面上难得一见的深度书籍，非常值得学习。

大规模数据处理的演化历程

大数据如果从 Google 对外发布 MapReduce 论文算起，已经前后跨越十五年，我打算在本文和你蜻蜓点水般一起浏览下大数据的发展史，我们

从最开始 MapReduce 计算模型开始，一路走马观花看看大数据这十五年关键发展变化，同时也顺便会讲解流式处理这个领域是如何发展到今天的这幅模样。这其中我也会加入一些我对一些业界知名大数据处理系统（可能里面有些也不那么出名）的观察和评论，同时考虑到我很有可能简化、低估甚至于忽略了很多重要的大数据处理系统，我也会附带一些参考材料帮助大家学习更多更详细的知识。

另外，我们仅仅讨论了大数据处理中偏 MapReduce/Hadoop 系统及其派系分支的大数据处理。我没有讨论任何 SQL 引擎 [1]，我们同样也没有讨论 HPC 或者超级计算机。尽管我这章的标题听上去领域覆盖非常广泛，但实际上我仅仅会讨论一个相对比较垂直的大数据领域。

同样需要提醒的一件事情是，我在本文里面或多或少会提到一些 Google 的技术，不用说这块是因为与我在谷歌工作了十多年的经历有关。但还有另外两个原因：1) 大数据对谷歌来说一直很重要，因此在那里创造了许多有价值的东西值得详细讨论，2) 我的经验一直是谷歌以外的人似乎更喜欢学习 Google 所做的事情，因为 Google 公司在这方面一直有点守口如瓶。所以，当我过分关注我们一直在“闭门造车”的东西时，姑且容忍下我吧。

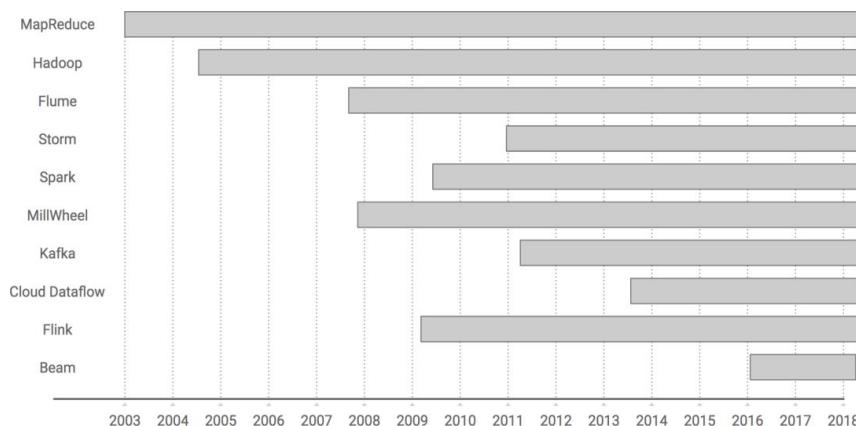


图 10-1 本章讨论各个大数据系统时间表

为了使我们这一次大数据旅行显得更加具体有条理，我们设计了图 10-1 的时间表，这张时间表概括地展示了不同系统的诞生日。

在每一个系统介绍过程中，我会尽可能说明清楚该系统的简要历史，并且我会尝试从流式处理系统的演化角度来阐释该系统对演化过程的贡献。最后，我们将回顾以上系统所有的贡献，从而全面了解上述系统如何演化并构建出现代流式处理系统的。

MapReduce

我们从 MapReduce 开始我们的旅程。

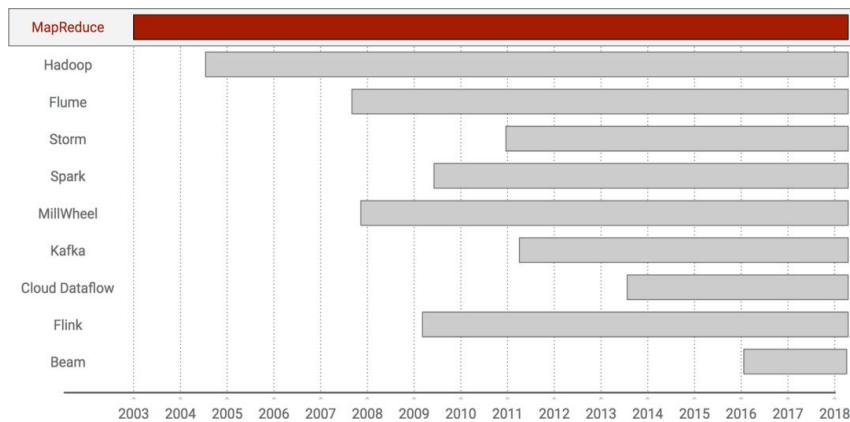


图 10-2 MapReduce 的时间表

我认为我们可以很确定地说，今天我们讨论的大规模数据处理系统都源自于 2003 年 MapReduce。当时，谷歌的工程师正在构建各种定制化系统，以解决互联网时代下大数据处理难题。当他们这样尝试去解决这些问题时候，发现有三个难以逾越的坎儿：

数据处理很难 只要是数据科学家或者工程师都很清楚。如果你能够精通于从原始数据挖掘出对企业有价值的信息，那这个技能能够保你这辈子吃喝不愁。

可伸缩性很难 本来数据处理已经够难了，要从大规模数据集中挖掘出有价值的数据更加困难。

容错很难 要从大规模数据集挖掘数据已经很难了，如果还要想办法在一批廉价机器构建的分布式集群上可容错地、准确地方式挖掘数据价值，那真是难于上青天了。

在多种应用场景中都尝试解决了上述三个问题之后，Google 的工程师们开始注意到各自构建的定制化系统之间颇有相似之处。最终，Google 工程师悟出来一个道理：如果他们能够构建一个可以解决上述问题二和问题三的框架，那么工程师就将可以完全放下问题二和三，从而集中精力解决每个业务都需要解决的问题一。于是，MapReduce 框架诞生了。

MapReduce 的基本思想是提供一套非常简洁的数据处理 API，这套 API 来自于函数式编程领域的两个非常易于理解的操作：map 和 reduce（图 10-3）。使用该 API 构建的底层数据流将在这套分布式系统框架上执行，框架负责处理所有繁琐的可扩展性和容错性问题。可扩展性和容错性问题对于分布式底层工程师来说无疑是非常有挑战的课题，但对于普通工程师而言，无益于是灾难。

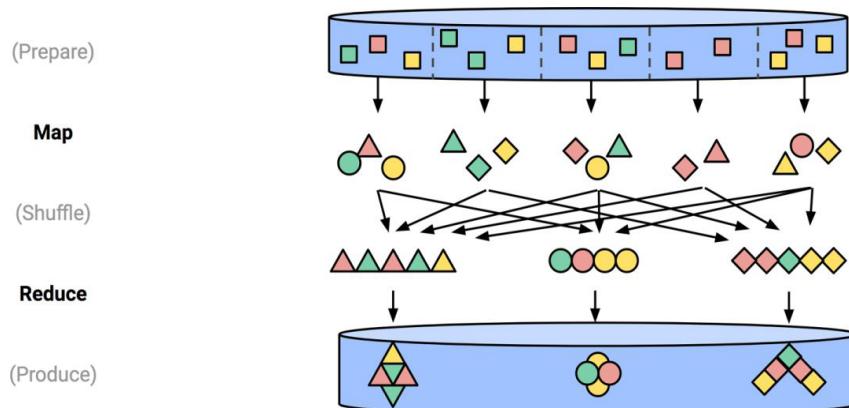


图 10-3 MapReduce 作业原理图

我们已经在第 6 章详细讨论了 MapReduce 的语义，所以我们在此不再赘述。仅仅简单地回想一下，我们将处理过程分解为六个离散阶段（MapRead, Map, MapWrite, ReduceRead, Reduce, ReduceWrite）作为对于流或者表进行分析的几个步骤。我们可以看到，整体上 Map 和 Reduce 阶段之间差异其实也不大；更高层次来看，他们都做了以下事情：

从表中读取数据，并转换为数据流（译者注：即 MapRead、ReduceRead）

针对上述数据流，将用户编写业务处理代码应用于上述数据流，转换

并形成新的一个数据流。 (译者注: 即 Map、Reduce)

将上述转换后的流根据某些规则分组，并写出到表中。 (译者注: 即 MapWrite、ReduceWrite)

随后，Google 内部将 MapReduce 投入生产使用并得到了非常广泛的业务应用，Google 认为应该和公司外的同行分享我们的研究成果，最终我们将 MapReduce 论文发表于 OSDI 2004（见图 10-4）。

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* op-

图 10-4 MapReduce 论文发表在 OSDI 2004 上

论文中，Google 详细描述了 MapReduce 项目的历史，API 的设计和实现，以及有关使用了 MapReduce 框架的许多不同生产案例的详细信息。当然，Google 没有提供任何实际的源代码，以至于最终 Google 以外的人都认为：“是的，这套系统确实牛啊！”，然后立马回头去模仿 MapReduce 去构建他们的定制化系统。

在随后这十年的过程中，MapReduce 继续在谷歌内部进行大量开发，投入大量时间将这套系统规模推进到前所未有的水平。如果读者朋友希望了解一些更加深入更加详细的 MapReduce 说明，我推荐由我们的 MapReduce 团队中负责扩展性、性能优化的大牛 Marián Dvorský 撰写的文

章 《History of massive-scale sorting experiments at Google》（图 10-5）

History of massive-scale sorting experiments at Google

Thursday, February 18, 2016

We've tested MapReduce by sorting large amounts of random data ever since we created the tool. We like sorting, because it's easy to generate an arbitrary amount of data, and it's easy to validate that the output is correct.

Even the [original MapReduce paper](#) reports a TeraSort result. Engineers run 1TB or 10TB sorts as regression tests on a regular basis, because obscure bugs tend to be more visible on a large scale. However, the real fun begins when we increase the scale even further. In this post I'll talk about our experience with some petabyte-scale sorting experiments we did a few years ago, including what we believe to be the largest MapReduce job ever: a 50PB sort.

These days, GraySort is the large scale sorting benchmark of choice. In GraySort, you must sort at least 100TB of data (as 100-byte records with the first 10 bytes being the key), lexicographically, as fast as possible. The site [sortbenchmark.org](#) tracks official winners for this benchmark. We never entered the official competition.

MapReduce happens to be a good fit for solving this problem, because the way it implements reduce is by sorting the keys. With the appropriate (lexicographic) sharding function, the output of MapReduce is a sequence of files comprising the final sorted dataset.

Once in awhile, when a new cluster in a datacenter came up (typically for use by the search indexing team), we in the MapReduce team got the opportunity to play for a few weeks before the real workload moved in. This is when we had a chance to "burn in" the cluster, stretch the limits of the hardware, destroy some hard drives, play with some really expensive equipment, learn a lot about system performance, and, win (unofficially) the sorting benchmark.

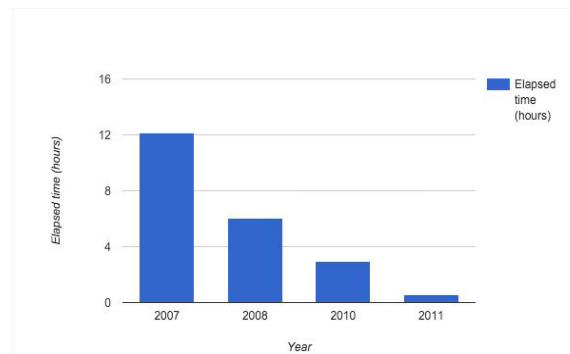


图 10-5 Mari á nDvorský 的《History of massive-scale sorting experiments at Google》博客文章

我这里希望强调的是，这么多年来看，其他任何的分布式架构最终都没有达到 MapReduce 的集群规模，甚至在 Google 内部也没有。从 MapReduce 诞生起到现在已经跨越十载之久，都未能看到真正能够超越 MapReduce 系统规模的另外一套系统，足见 MapReduce 系统之成功。14 年的光阴看似不长，对于互联网行业已然永久。

从流式处理系统来看，我想为读者朋友强调的是 MapReduce 的简单性和可扩展性。MapReduce 给我们的启发是：MapReduce 系统的设计非常勇于创新，它提供一套简便且直接的 API，用于构建业务复杂但可靠健壮的底层分布式数据 Pipeline，并足够将这套分布式数据 Pipeline 运行在廉价普通的商用服务器集群之上。

Hadoop

我们大数据旅程的下一站是 Hadoop（图 10-6）。需要着重说明的是：我为了保证我们讨论的重心不至于偏离太多，而压缩简化讨论 Hadoop 的内容。但必须承认的是，Hadoop 对我们的行业甚至整个世界的影响不容小觑，它带来的影响远远超出了我在此书讨论的范围。

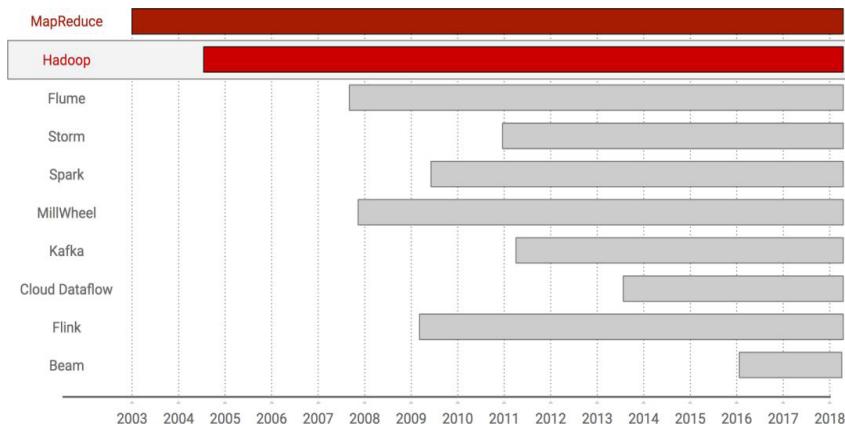


图 10-6 Hadoop 的时间表

Hadoop 于 2005 年问世，当时 Doug Cutting 和 Mike Cafarella 认为 MapReduce 论文中的想法太棒了，他们在构建 Nutch webcrawler 的分布式版本正好需要这套分布式理论基础。在这之前，他们已经实现了自己版本的 Google 分布式文件系统（最初称为 Nutch 分布式文件系统的 NDFS，后来改名为 HDFS 或 Hadoop 分布式文件系统）。因此下一步，自然而然的，基于 HDFS 之上添加 MapReduce 计算层。他们称 MapReduce 这一层为 Hadoop。

Hadoop 和 MapReduce 之间的主要区别在于 Cutting 和 Cafarella 通过开源（以及 HDFS 的源代码）确保 Hadoop 的源代码与世界各地可以共享，最终成为 Apache Hadoop 项目的一部分。雅虎聘请 Cutting 来帮助将雅虎网络爬虫项目升级为全部基于 Hadoop 架构，这个项目使得 Hadoop 有效提升了生产可用性以及工程效率。自那以后，整个开源生态的大数据处理工具生态系统得到了蓬勃发展。与 MapReduce 一样，相信其他人

已经能够比我更好地讲述了 Hadoop 的历史。我推荐一个特别好的讲解是 Marko Bonaci 的《The history of Hadoop》，它本身也是一本已经出版的纸质书籍（图 10-7）。

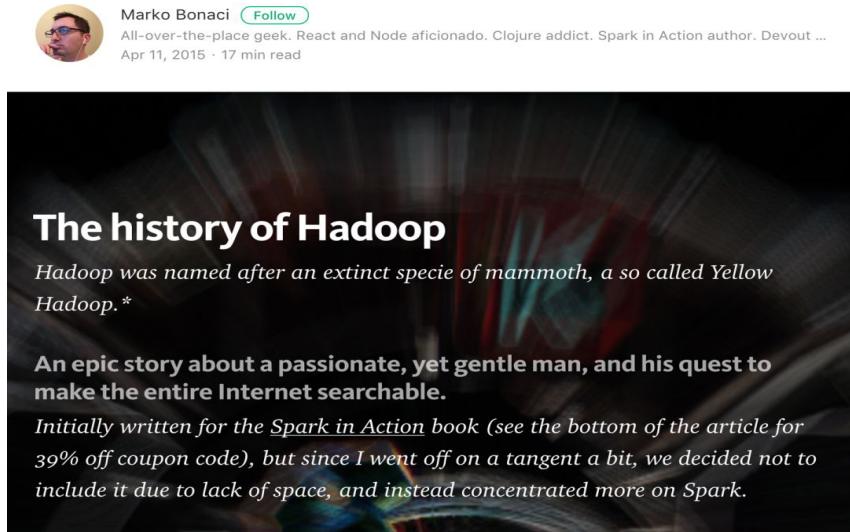


图 10-7 Marko Bonaci 的《The history of Hadoop》

在 Hadoop 这部分，我期望读者朋友能够了解到围绕 Hadoop 的开源生态系统对整个行业产生的巨大影响。通过创建一个开放的社区，工程师可以从早期的 GFS 和 MapReduce 论文中改进和扩展这些想法，这直接促进生态系统的蓬勃发展，并基于此之上产生了许多有用的工具，如 Pig，Hive，HBase，Crunch 等等。这种开放性是导致我们整个行业现有思想多样性的关键，同时 Hadoop 开放性生态亦是直接促进流计算系统发展。

Flume

我们现在再回到 Google，讨论 Google 公司中 MapReduce 的官方继承者：Flume（[图 10-8]，有时也称为 FlumeJava，这个名字起源于最初 Flume 的 Java 版本。需要注意的是，这里的 Flume 不要与 Apache Flume 混淆，这部分是面向不同领域的东西，只是恰好有同样的名字）。

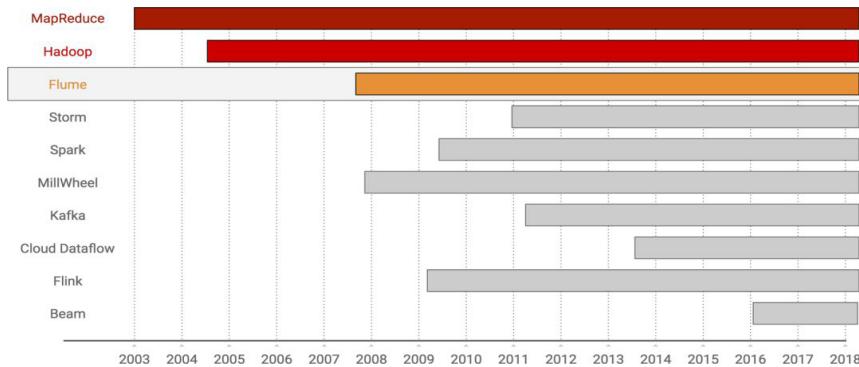


图 10-8 Flume 的时间表

Flume 项目由 Craig Chambers 在 2007 年谷歌西雅图办事处成立时发起。Flume 最初打算是希望解决 MapReduce 的一些固有缺点，这些缺点即使在 MapReduce 最初大红大紫的阶段已经非常明显。其中许多缺点都与 MapReduce 完全限定的 Map→Shuffle→Reduce 编程模型相关；这个编程模型虽然简单，但它带来了一些缺点：

由于单个 MapReduce 作业并不能完成大量实际上的业务案例，因此许多定制的编排系统开始在 Google 公司内部出现，这些编排系统主要用于协调 MapReduce 作业的顺序。这些系统基本上都在解决同一类问题，即将多个 MapReduce 作业粘合在一起，创建一个解决复杂问题的数据管道。然而，这些编排系统都是 Google 各自团队独立开发的，相互之间也完全不兼容，是一类典型的重复造轮子案例。

更糟糕的是，由于 MapReduce 设计的 API 遵循严格结构，在很多情况下严格遵循 MapReduce 编程模型会导致作业运行效率低下。例如，一个团队可能会编写一个简单地过滤掉一些元素的 MapReduce，即，仅有 Map 阶段没有 Reduce 阶段的作业。这个作业下游紧接着另一个团队同样仅有 Map 阶段的作业，进行一些字段扩展和丰富（仍然带一个空的 Reduce 阶段作业）。第二个作业的输出最终可能会被第三个团队的 MapReduce 作业作为输入，第三个作业将对数据执行某些分组聚合。这个 Pipeline，实际上由一个合并 Map 阶段（译者注：前面两个 Map 合并为一个 Map），外加一个 Reduce 阶段即可完成业务逻辑，但实际上却需要编排

三个完全独立的作业，每个作业通过 Shuffle 和 Output 两个步骤链接在一起。假设你希望保持代码的逻辑性和清洁性，于是你考虑将部分代码进行合并，但这个最终导致第三个问题。

为了优化 MapReduce 作业中的这些低效代码，工程师们开始引入手动优化，但不幸的是，这些优化会混淆 Pipeline 的简单逻辑，进而增加维护和调试成本。

Flume 通过提供可组合的高级 API 来描述数据处理流水线，从而解决了这些问题。这套设计理念同样也是 Beam 主要的抽象模型，即 PCollection 和 PTransform 概念，如图 10-9 所示。

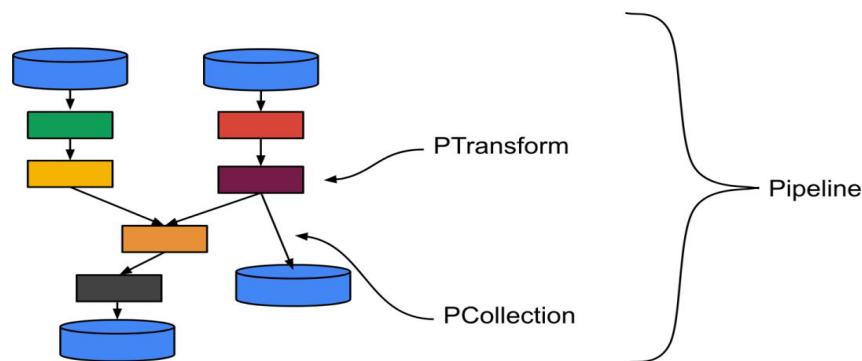


图 10-9 Flume 的高层抽象模型（图片来源：Frances Perry）

这些数据处理 Pipeline 在作业启动时将通过优化器生成，优化器将以最佳效率生成 MapReduce 作业，然后交由框架编排执行。整个编译执行原理图可以在图 10-10 中看到。

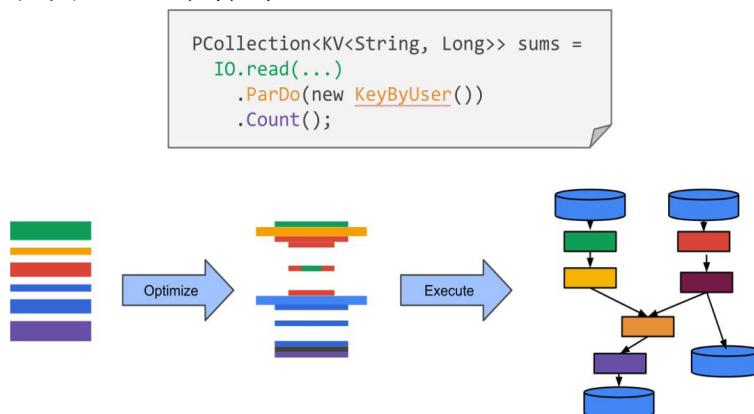


图 10-10 从逻辑管道到物理执行计划的优化

也许 Flume 在自动优化方面最重要的案例就是是合并 (Reuven 在第 5 章中讨论了这个主题) , 其中两个逻辑上独立的阶段可以在同一个作业中顺序地 (消费者 - 生产者融合) 执行或者并行执行 (兄弟融合) , 如图 10-11 所示。

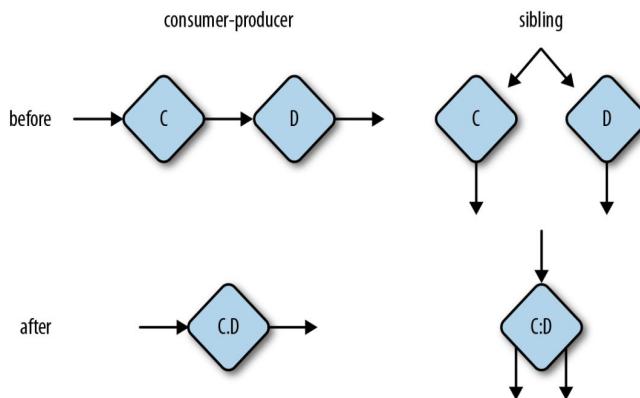


图 10-11 合并优化将顺序或并行操作 (算子) 组合在一起, 到同一个操作

将两个阶段融合在一起消除了序列化 / 反序列化和网络开销, 这在处理大量数据的底层 Pipeline 中非常重要。

另一种类型的自动优化是 combiner lifting (见图 10-12) , 当我们讨论增量合并时, 我们已经在第 7 章中讨论了这些机制。combiner lifting 只是我们在该章讨论的多级组合逻辑的编译器自动优化: 以求和操作为例, 求和的合并逻辑本来应该运算在分组 (译者注: 即 Group-By) 操作后, 由于优化的原因, 被提前到在 group-by-key 之前做局部求和 (根据 group-by-key 的语义, 经过 group-by-key 操作需要跨网络进行大量数据 Shuffle) 。在出现数据热点情况下, 将这个操作提前可以大大减少通过网络 Shuffle 的数据量, 并且还可以在多台机器上分散掉最终聚合的机器负载。

由于其更清晰的 API 定义和自动优化机制, 在 2009 年初 Google 内部推出后 FlumeJava 立即受到巨大欢迎。之后, 该团队发表了题为《Flume Java: Easy, Efficient Data-Parallel Pipelines》 (<https://storage.googleapis.com/pub-tools-public-publication-data/pdf/35650.pdf>) 的论文 (参见图 10-13) , 这篇论文本身就是一个很好的学习 FlumeJava 的资料。

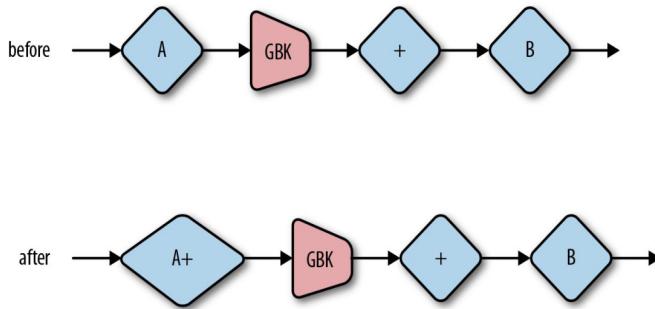


图 10-12: combiner lifting 在数据上游直接进行局部聚合后再发送给下游端
进行二次聚合

FlumeJava: Easy, Efficient Data-Parallel Pipelines

Craig Chambers, Ashish Raniwala, Frances Perry,
Stephen Adams, Robert R. Henry,
Robert Bradshaw, Nathan Weizenbaum
Google, Inc.
{chambers, raniwala, fjp, sra, rrh, robertwb, nweiz}@google.com

Abstract

MapReduce 和类似系统显著地简化了编写并行代码的任务。然而，许多现实世界的计算要求一个 MapReduce 流程，编程和管理这样的管道会很困难。我们提出了 FlumeJava，一个 Java 库，使得开发、测试和运行高效的并行管道变得容易。FlumeJava 的核心是库中的几个类，它们代表不可变的并行集合，支持处理它们的适度数量的操作。并行集合及其操作提供了一个简单、高阶、统一的抽象，跨越不同的数据表示和执行策略。为了能够并行地运行它们，FlumeJava 延迟了它们的评估，而是内部地构造了一个执行计划的数据流图。当最终的并行操作所需时，FlumeJava 首先优化执行计划，然后执行优化后的操作。在适当的底层原语上（例如，MapReduces）组合高阶抽象，为并行数据和计算提供了延迟评估和优化，并且通过并行原语提供了高效的并行性。FlumeJava 提供了一个易于使用的系统，其效率接近于手工优化的管道。FlumeJava 正在 Google 内部数百名管道开发者中使用。

MapReduce 工作得很好，适用于可以被拆分为 map 步骤、shuffle 步骤和 reduce 步骤的任务，但对于许多现实世界的计算来说，MapReduce 简单地不够。这样的管道需要一个链状的 MapReduce 阶段。这样的数据并行管道需要额外的协调代码来将这些阶段链接在一起，并且需要额外的管理工作来创建和删除管道中间阶段的临时结果。逻辑计算可能被低级别的协调细节所遮蔽，使得新开发者理解计算非常困难。此外，管道的划分到特定阶段的逻辑使得代码“烘焙”在其中并且很难在以后更改，除非逻辑计算需要进化。

在本文中，我们提出了 FlumeJava，一个新的系统，旨在支持并行管道的开发。FlumeJava 围绕一些类中心，这些类代表并行集合。并行集合支持适度数量的并行操作，这些操作被组合以实现并行计算。一个完整的管道，甚至是多个管道，都可以在单个 Java 程序中实现，使用 FlumeJava 抽象；没有理由在每个阶段都分离出逻辑计算。

FlumeJava 的并行集合抽象掉了每个阶段的细节。

图 10-13 FlumeJava 的论文

Flume C++ 版本很快于 2011 年发布。之后 2012 年初，Flume 被引入为 Google 的所有新工程师提供的 Noogler6 培训内容。MapReduce 框架于是最终被走向被替换的命运。

从那时起，Flume 已经迁移到不再使用 MapReduce 作为执行引擎；相反，Flume 底层基于一个名为 Dax 的内置自定义执行引擎。工作本身。不仅让 Flume 更加灵活选择执行计划而不必拘泥于 Map→Shuffle→Reduce MapReduce 的模型，Dax 还启用了新的优化，例如 Eugene Kirpi-chov 和 Malo Denielou 的《No shard left behind》[博客文章](#) 中描述的动态负载均衡（图 10-14）。



No shard left behind: dynamic work rebalancing in Google Cloud Dataflow

Wednesday, May 18, 2016

Posted by Eugene Kirpichov, Senior Software Engineer and Malo Denielou, Software Engineer

Introduction

Today we continue the discussion of [Google Cloud Dataflow's](#) "zero-knobs" story. Previously we showcased Cloud Dataflow's capability for [Autoscaling](#), which dynamically adjusts the number of workers to the needs of your pipeline. In this post, we discuss [Dynamic Work Rebalancing](#) (known internally at Google as *Liquid Sharding*), which keeps the workers busy.

We'll show how this feature addresses the problem of stragglers (workers that take a long time to finish their part of the work, delaying completion of the job and keeping other resources idle), greatly improving performance and cost in many scenarios, and how it enables and works in concert with autoscaling.

The problem of stragglers in big data processing systems

In all major distributed data processing engines – from Google's original MapReduce, to Hadoop, to modern systems such as Spark, Flink and Cloud Dataflow – one of the key operations is Map, which applies a function to all elements of an input in parallel (called ParDo in the terminology of [Apache Beam \(incubating\)](#) programming model).

图 10-14 帖子《No shard left behind》

尽管那篇博客主要是基于 Google DataFlow 框架下讨论问题，但动态负载均衡（或液态分片，Google 内部更习惯这样叫）可以让部分已经完成工作的 Worker 能够从另外一些繁忙的 Worker 手中分配一些额外的工作。在 Job 运行过程中，通过不断的动态调整负载分配可以将系统运行效率趋近最优，这种算法将比传统方法下有经验工程师手工设置的初始参数性能更好。Flume 甚至为 Worker 池变化进行了适配，一个拖慢整个作业进度的 Worker 会将其任务转移到其他更加高效的 Worker 上面进行执行。Flume 的这些优化手段，在 Google 内部为公司节省了大量资源。

最后一点，Flume 后来也被扩展为支持流语义。除 Dax 作为一个批处理系统引擎外，Flume 还扩展为能够在 MillWheel 流处理系统上执行作业（稍后讨论）。在 Google 内部，之前本书中讨论过的大多数高级流处理语义概念首先被整合到 Flume 中，然后才进入 Cloud Dataflow 并最终进入 Apache Beam。

总而言之，本节我们主要强调的是 Flume 产品给人引入高级管道概念，这使得能够让用户编写清晰易懂且自动优化的分布式大数据处理逻辑，从而让创建更大型更复杂的分布式大数据任务成为了可能，Flume 让我们业务代码在保持代码清晰逻辑干净的同时，自动具备编译器优化能力。

Storm

接下来是 Apache Storm (图 10-15)，这是我们研究的第一个真正的流式系统。Storm 肯定不是业界使用最早的流式处理系统，但我认为这是整个行业真正广泛采用的第一个流式处理系统，因此我们在这里需要仔细研究一下。

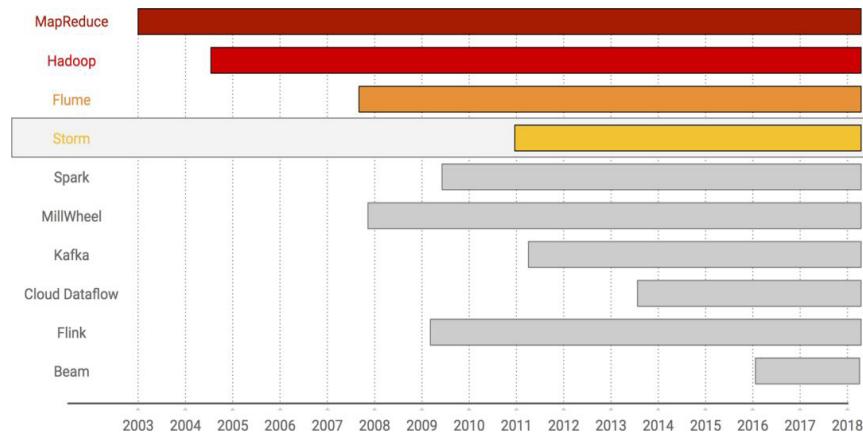


图 10-15 Storm 的时间轴

Storm 是 Nathan Marz 的心血结晶，Nathan Marz 后来在一篇题为《History of Apache Storm and lessons learned》的博客文章 (<http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html>) 中记录了其创作历史 (图 10-16)。这篇冗长的博客讲述了 BackType 这家创业公司一直在自己通过消息队列和自定义代码去处理 Twitter 信息流。Nathan 和十几年前 Google 里面设计 MapReduce 相关工程师有相同的认识：实际的业务处理的代码仅仅是系统代码很小一部分，如果有统一的流式实时处理框架负责处理各类分布式系统底层问题，那么基于之上构建我们的实时大数据处理将会轻松得多。基于此，Nathan 团队完成了 Storm 的设计和开发。

值得一提的是，Storm 的设计原则和其他系统大相径庭，Storm 更多考虑到实时流计算的处理时延而非数据的一致性保证。后者是其他大数据系统必备基础产品特征之一。Storm 针对每条流式数据进行计算处理，并

提供至多一次或者至少一次的语义保证；同时不提供任何状态存储能力。相比于 Batch 批处理系统能够提供一致性语义保证，Storm 系统能够提供更低的数据处理延迟。对于某些数据处理业务场景来说，这确实也是一个非常合理的取舍。

History of Apache Storm and lessons learned

 MONDAY, OCTOBER 6, 2014

Apache Storm recently became a [top-level project](#), marking a huge milestone for the project and for me personally. It's crazy to think that four years ago Storm was nothing more than an idea in my head, and now it's a thriving project with a large community used by [a ton of companies](#). In this post I want to look back at how Storm got to this point and the lessons I learned along the way.



图 10-16 《History of Apache Storm and lessons learned》

不幸的是，人们很快就清楚地知道他们想要什么样的流式处理系统。他们不仅希望快速得到业务结果，同时希望系统具有低延迟和准确性，但仅凭 Storm 架构实际上不可能做到这一点。针对这个情况，Nathan 后面又提出了 Lambda 架构。

鉴于 Storm 的局限性，聪明的工程师结合弱一致语义的 Storm 流处理以及强一致语义的 Hadoop 批处理。前者产生了低延迟，但不精确的结果，而后者产生了高延迟，但精确的结果，双剑合璧，整合两套系统整体提供的低延迟但最终一致的输出结果。我们在第 1 章中了解到，Lambda 架构是 Marz 的另一个创意，详见他的文章《“如何击败 CAP 定理”》(<http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>)（图 10-17）。

How to beat the CAP theorem

THURSDAY, OCTOBER 13, 2011

The CAP theorem states a database cannot guarantee consistency, availability, and partition-tolerance at the same time. But you can't sacrifice partition-tolerance (see [here](#) and [here](#)), so you must make a tradeoff between availability and consistency. Managing this tradeoff is a central focus of the NoSQL movement.

Consistency means that after you do a successful write, future reads will always take that write into account. Availability means that you can always read and write to the system. During a partition, you can only have one of these properties.

Systems that choose consistency over availability have to deal with some awkward issues. What do you do when the database isn't available? You can try buffering writes for later, but you risk losing those writes if you lose the machine with the buffer. Also, buffering writes can be a form of inconsistency because a client thinks a write has succeeded but the write isn't in the database yet. Alternatively, you can return errors back to the client when the database is unavailable. But if you've ever used a product that told you to "try again later", you know how aggravating this can be.

The other option is choosing availability over consistency. The best consistency guarantee these systems can provide is "eventual consistency". If you use an eventually consistent database, then sometimes you'll read a different result than you just wrote. Sometimes multiple readers reading the same key at the same time will

图 10-17 《How to beat the CAP theorem》

我已经花了相当多的时间来分析 Lambda 架构的缺点，以至于我不会在这里啰嗦这些问题。但我要重申一下：尽管它带来了大量成本问题，Lambda 架构当前还是非常受欢迎，仅仅是因为它满足了许多企业一个关键需求：系统提供低延迟但不准确的数据，后续通过批处理系统纠正之前数据，最终给出一致性的结果。从流处理系统演变的角度来看，Storm 确实为普罗大众带来低延迟的流式实时数据处理能力。然而，它是以牺牲数据强一致性为代价的，这反过来又带来了 Lambda 架构的兴起，导致接下来多年基于两套系统架构之上的数据处理带来无尽的麻烦和成本。

撇开其他问题先不说，Storm 是行业首次大规模尝试低延迟数据处理的系统，其影响反映在当前线上大量部署和应用各类流式处理系统。在我们要放下 Storm 开始聊其他系统之前，我觉得还是很有必要去说说 Heron 这个系统。在 2015 年，Twitter 作为 Storm 项目孵化公司以及世界上已知最大的 Storm 用户，突然宣布放弃 Storm 引擎，宣称正在研发另外一套称之为 Heron 的流式处理框架。Heron 旨在解决困扰 Storm 的一系列性能和维护问题，同时向 Storm 保持 API 兼容，详见题为《Twitter Heron: Stream Processing at scale》的论文（<https://www.semanticscholar.org/paper/>

Twitter-Heron%3A-Stream-Processing-at-Scale-Kulkarni-Bhagat/e847c3ec130da57328db79a7fea794b07dbccdd9) (图 10-18)。

Twitter Heron: Stream Processing at Scale

Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg,

Sailesh Mittal, Jignesh M. Patel^{1,1}, Karthik Ramasamy, Siddarth Taneja

@sanjeevrk, @challenger_nik, @Louis_FumaSong, @vikkyrk, @ckkellogg,

@saileshmittal, @pateljm, @karthikz, @staneja

Twitter, Inc., *University of Wisconsin – Madison

ABSTRACT

Storm has long served as the main platform for real-time analytics at Twitter. However, as the scale of data being processed in real-time at Twitter has increased, along with an increase in the diversity and the number of use cases, many limitations of Storm have become apparent. We need a system that scales better, has better debug-ability, has better performance, and is easier to manage – all while working in a shared cluster infrastructure. We considered various alternatives to meet these needs, and in the end concluded that we needed to build a new real-time stream data processing system. This paper presents the design and implementation of this new system, called Heron. Heron is now the de facto stream data processing engine inside Twitter, and in this paper we also share our experiences from running Heron in production. In this paper, we also provide empirical evidence demonstrating the efficiency and scalability of Heron.

ACM Classification

H.2.4 [Information Systems]: Database Management—systems

Keywords

Stream data processing systems; real-time data processing.

system process, which makes debugging very challenging. Thus, we needed a clearer mapping from the logical units of computation to each physical process. The importance of such clean mapping for debug-ability is really crucial when responding to pager alerts for a failing topology, especially if it is a topology that is critical to the underlying business model.

In addition, Storm needs dedicated cluster resources, which requires special hardware allocation to run Storm topologies. This approach leads to inefficiencies in using precious cluster resources, and also limits the ability to scale on demand. We needed the ability to work in a more flexible way with popular cluster scheduling software that allows sharing the cluster resources across different types of data processing systems (and not just a stream processing system). Internally at Twitter, this meant working with Aurora [1], as that is the dominant cluster management system in use.

With Storm, provisioning a new production topology requires manual isolation of machines, and conversely, when a topology is no longer needed, the machines allocated to serve that topology now have to be decommissioned. Managing machine provisioning in this way is cumbersome. Furthermore, we also wanted to be far more efficient than the Storm system in production, simply because at Twitter's scale, any improvement in performance

图 10-18 Heron 的论文

Heron 本身也是开源产品（但开源不在 Apache 项目中）。鉴于 Storm 仍然在社区中持续发展，现在又冒出一套和 Storm 竞争的软件，最终两边系统鹿死谁手，我们只能拭目以待了。

Spark

继续走起，我们现在来到 Apache Spark (图 10-19)。再次，我又将大量简化 Spark 系统对行业的总体影响探讨，仅仅关注我们的流处理领域部分。

Spark 在 2009 年左右诞生于加州大学伯克利分校的著名 AMPLab。最初推动 Spark 成名的原因是它能够经常在内存执行大量的计算工作，直到作业的最后一步才写入磁盘。工程师通过弹性分布式数据集 (RDD) 理念实现了这一目标，在底层 Pipeline 中能够获取每个阶段数据结果的所有派生关系，并且允许在机器故障时根据需要重新计算中间结果，当然，这些都基于一些假设 a) 输入是总是可重放的，b) 计算是确定性的。对于许多案例来说，这些先决条件是真实的，或者看上去足够真实，至少用户

确实在 Spark 享受到了巨大的性能提升。从那时起，Spark 逐渐建立起其作为 Hadoop 事实上的继任产品定位。

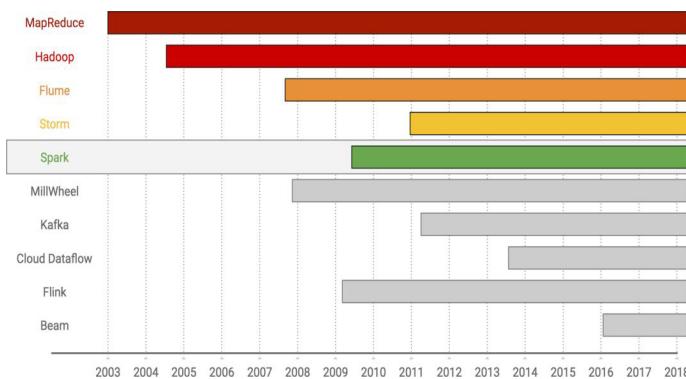


图 10-19 Spark 的时间轴

在 Spark 创建几年后，当时 AMPLab 的研究生 Tathagata Das 开始意识到：嘿，我们有这个快速的批处理引擎，如果我们将多个批次的任务串接起来，用它能否来处理流数据？于是乎，Spark Streaming 诞生了。

关于 Spark Streaming 的真正精彩之处在于：强大的批处理引擎解决了太多底层麻烦的问题，如果基于此构建流式处理引擎则整个流处理系统将简单很多，于是世界又多一个流处理引擎，而且是可以独自提供一致性语义保障的流式处理系统。换句话说，给定正确的用例，你可以不用 Lambda 架构系统直接使用 Spark Streaming 即可满足数据一致性需求。为 Spark Streaming 手工点赞！

这里的一个主要问题是“正确的用例”部分。早期版本的 Spark Streaming（1.x 版本）的一大缺点是它仅支持特定的流处理语义：即，处理时间窗口。因此，任何需要使用事件时间，需要处理延迟数据等等案例都无法让用户使用 Spark 开箱即用解决业务。这意味着 Spark Streaming 最适合于有序数据或事件时间无关的计算。而且，正如我在本书中重申的那样，在处理当今常见的大规模、以用户为中心的数据集时，这些先决条件看上去并不是那么常见。

围绕 Spark Streaming 的另一个有趣的争议是“microbatch 和 true streaming”争论。由于 Spark Streaming 建立在批处理引擎的重复运行的

基础之上，因此批评者声称 Spark Streaming 不是真正的流式引擎，因为整个系统的处理基于全局的数据切分规则。这个或多或少是实情。尽管流处理引擎几乎总是为了吞吐量而使用某种批处理或者类似的加大吞吐的系统策略，但它们可以灵活地在更精细的级别上进行处理，一直可以细化到某个 key。但基于微批处理模型的系统在基于全局切分方式处理数据包，这意味着同时具备低延迟和高吞吐是不可能的。确实我们看到许多基准测试表明这说法或多或少有点正确。当然，作业能够做到几分钟或几秒钟的延迟已经相当不错了，实际上生产中很少有用例需要严格数据正确性和低延迟保证。所以从某种意义上说，Spark 瞄准最初目标客户群体打法是非常到位的，因为大多数业务场景均属于这一类。但这并未阻止其竞争对手将此作为该平台的巨大劣势。就个人而言，在大多数情况下，我认为这只是一个很小问题。

撇开缺点不说，Spark Streaming 是流处理的分水岭：第一个广泛使用的大规模流处理引擎，它也可以提供批处理系统的正确性保证。当然，正如前面提到的，流式系统只是 Spark 整体成功故事的一小部分，Spark 在迭代处理和机器学习领域做出了重要贡献，其原生 SQL 集成以及上述快如闪电般的内存计算，都是非常值得大书特书的产品特性。

如果您想了解有关原始 Spark 1.x 架构细节的更多信息，我强烈推荐 Matei Zaharia 关于该主题的论文《“An Architecture for Fast and General Data Processing on Large Clusters》（图 10-20）。这是 113 页的 Spark 核心讲解论文，非常值得一读。

时至今日，Spark 的 2.x 版本极大地扩展了 Spark Streaming 的语义功能，其中已经包含了本书中描述流式处理模型的许多部分，同时试图简化一些更复杂的设计。Spark 甚至推出了一种全新的、真正面向流式处理的架构，用以规避掉微批架构的种种问题。但是曾经，当 Spark 第一次出现时，它带来的最重要贡献是它是第一个公开可用的流处理引擎，具有数据处理的强一致性语义，尽管这个特性只能用在有序数据或使用处理时间计算的场景。

An Architecture for Fast and General Data Processing on Large Clusters

by

Matei Alexandru Zaharia

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

The past few years have seen a major change in computing systems, as growing data volumes and stalling processor speeds require more and more applications to scale out to distributed systems. Today, a myriad data sources, from the Internet to business operations to scientific instruments, produce large and valuable data streams. However, the processing capabilities of single machines have not kept up with the size of data, making it harder and harder to put to use. As a result, a growing number of organizations—not just web companies, but traditional enterprises and research labs—need to scale out their most important computations to clusters of hundreds of machines.

At the same time, the speed and sophistication required of data processing have grown. In addition to simple queries, complex algorithms like machine learning and graph analysis are becoming common in many domains. And in addition to batch processing, streaming analysis of new real-time data sources is required to let organizations take timely action. Future computing platforms will need to not only scale out traditional workloads, but support these new applications as well.

This dissertation proposes an architecture for cluster computing systems that can tackle emerging data processing workloads while coping with larger and larger scales. Whereas early cluster computing systems, like MapReduce, handled batch

图 10-20 Spark 的学位论文

MillWheel

接下来我们讨论 MillWheel，这是我在 2008 年加入 Google 后的花 20% 时间兼职参与的项目，后来在 2010 年全职加入该团队（图 10-21）。

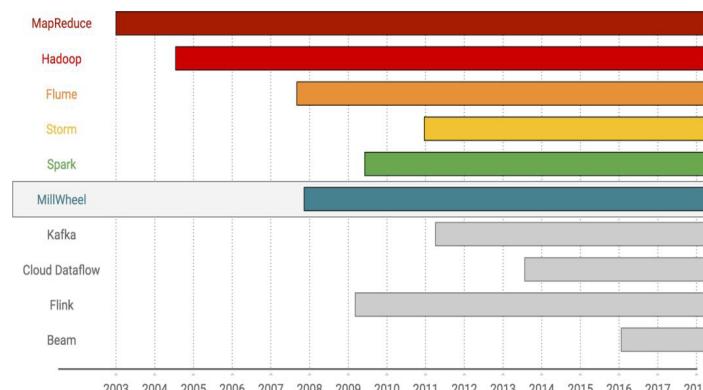


图 10-21 MillWheel 时间表

MillWheel 是 Google 最早的通用流处理架构，该项目由 Paul

Nordstrom 在 Google 西雅图办事处开业时发起。 MillWheel 在 Google 内的成功与长期以来一直致力于为无序数据提供低延迟，强一致的处理能力不无关系。在本书的讲解中，我们已经多次分别讨论了促使 MillWheel 成为一款成功产品的方方面面。

- 第五章，Reuven 详细讨论过数据精准一次的语义保证。精准一次的语义保证对于正确性至关重要。
- 第七章，我们研究了状态持久化，这为在不那么靠谱的普通硬件上执行的长时间数据处理业务并且需要保证正确性奠定了基础。
- 第三章，Slava 讨论了 Watermark。Watermark 为处理无序数据提供了基础。
- 第七章，我们研究了持久性计时器，它们提供了 Watermark 与业务逻辑之间的某些关联特性。

有点令人惊讶的是，MillWheel 项目最开始并未关注数据正确性。保罗最初的想法更接近于 Storm 的设计理论：具有弱一致性的低延迟数据处理。这是最初的 MillWheel 客户，一个关于基于用户搜索数据构建会话和另一个对搜索查询执行异常检测（来自 MillWheel 论文的 Zeitgeist 示例），这两家客户迫使项目走向了正确的方向。两者都非常需要强一致的数据结果：会话用于推断用户行为，异常检测用于推断搜索查询的趋势；如果他们提供的数据不靠谱，两者效果都会显着下降。最终，幸运的是，MillWheel 的设计被客户需求导向追求数据强一致性的结果。

支持乱序数据处理，这是现代流式处理系统的另一个核心功能。这个核心功能通常也被认为是被 MillWheel 引入到流式处理领域，和数据准确性一样，这个功能也是被客户需求推动最终加入到我们系统。Zeitgeist 项目的大数据处理过程，通常被我们拿来用作一个真正的流式处理案例来讨论。Zeitgeist 项目希望检测识别搜索查询流量中的异常，并且需要捕获异常流量。对于这个大数据项目数据消费者来说，流计算将所有计算结果产出并让用户轮询所有 key 用来识别异常显然不太现实，数据用户要求系统直接计算某个 key 出现异常的数据结果，而不需要上层再来轮询。对于

异常峰值（即查询流量的增加），这还相对来说比较简单好解决：当给定查询的计数超过查询的预期值时，系统发出异常信号。但是对于异常下降（即查询流量减少），问题有点棘手。仅仅看到给定搜索词的查询数量减少是不够的，因为在任何时间段内，计算结果总是从零开始。在这些情况下你必须确保你的数据输入真的能够代表当前这段时间真实业务流量，然后才将计算结果和预设模型进行比较。

- 真正的流式处理

“真正的流式处理用例”需要一些额外解释。流式系统的一个新的演化趋势是，舍弃掉部分产品需求以简化编程模型，从而使整个系统简单易用。例如，在撰写本文时，Spark Structured Streaming 和 Apache Kafka Streams 都将系统提供的功能限制在第 8 章中称为“物化视图语义”范围内，本质上对最终一致性的输出表不停做数据更新。当您想要将上述输出表作为结果查询使用时，物化视图语义非常匹配你的需求：任何时候我们只需查找该表中的值并且（译者注：尽管结果数据一直在不停被更新和改变）以当前查询时间请求到查询结果就是最新的结果。但在一些需要真正流式处理的场景，例如异常检测，上述物化视图并不能够很好地解决这类问题。

接下来我们会讨论到，异常检测的某些需求使其不适合纯物化视图语义（即，依次针对单条记录处理），特别当需要完整的数据集才能够识别业务异常，而这些异常恰好是由于数据的缺失或者不完整导致的。另外，不停轮询结果表以查看是否有异常其实并不是一个扩展性很好的办法。真正的流式用户场景是推动 watermark 等功能的原始需求来源。（Watermark 所代表的时间有先有后，我们需要最低的 Watermark 追踪数据的完整性，而最高的 Watermark 在数据时间发生倾斜时候非常容易导致丢数据的情况发生，类似 Spark Structured Streaming 的用法）。省略类似 Watermark 等功能的系统看上去简单不少，但换来代价是功能受限。在很多情况下，这些功能实际上有非常重要的业务价值。但如果这样的系统声称这些简化的功能会带来系统更多的普适性，不要听他们忽悠。试问一句，功能需求大量

被砍掉，如何保证系统的普适性呢？

Zeitgeist 项目首先尝试通过在计算逻辑之前插入处理时间的延迟数值来解决数据延迟问题。当数据按顺序到达时，这个思路处理逻辑正常。但业务人员随后发现数据有时可能会延迟很大，从而导致数据无序进入流式处理系统。一旦出现这个情况，系统仅仅采用处理时间的延迟是不够的，因为底层数据处理会因为数据乱序原因被错误判断为异常。最终，我们需要一种等待数据到齐的机制。

之后 Watermark 被设计出来用以解决数据乱序的问题。正如 Slava 在第 3 章中所描述的那样，基本思想是跟踪系统输入数据的当前进度，对于每个给定的数据源，构建一个数据输入进度用来表征输入数据的完整性。对于一些简单的数据源，例如一个带分区的 Kafka Topic，每个 Topic 下属的分区被写入的是业务时间持续递增的数据（例如通过 Web 前端实时记录的日志事件），这种情况下我们可以计算产生一个非常完美的 Watermark。但对于一些非常复杂的数据输入，例如动态的输入日志集，一个启发式算法可能是我们能够设计出来最能解决业务问题的 Watermark 生成算法了。但无论哪种方式，Watermark 都是解决输入事件完整性最佳方式。之前我们尝试使用处理时间来解决事件输入完整性，有点驴头不及马嘴的感觉。

得益于客户的需求推动，MillWheel 最终成为能够支持无序数据的强大流处理引擎。因此，题为《MillWheel: Fault-Tolerant Stream Processing at Internet Scale》（图 10-22）的论文花费大部分时间来讨论在这样的系统中提供正确性的各种问题，一致性保证、Watermark。如果您对这个主题感兴趣，那值得花时间去读读这篇论文。

MillWheel 论文发表后不久，MillWheel 就成为 Flume 底层提供支撑的流式处理引擎，我们称之为 Streaming Flume。今天在谷歌内部，MillWheel 被下一代理论更为领先的系统所替换：Windmill（这套系统同时也为 DataFlow 提供了执行引擎），这是一套基于 MillWheel 之上，博采众家之长的大数据处理系统，包括提供更好的调度和分发策略、更清晰的

框架和业务代码解耦。

MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman,
Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle
Google

{takidau, alexgb, kayab, chernyak, haberman,
relax, sgmc, millsd, pgn, samuelw}@google.com

ABSTRACT

MillWheel is a framework for building low-latency data-processing applications that is widely used at Google. Users specify a directed computation graph and persistent code for individual nodes, and the system manages persistent state and the continuous flow of records, all within the envelope of the framework's fault-tolerance guarantees.

This paper describes MillWheel's programming model as well as its implementation. The case study of a continuous anomaly detector in use at Google serves to motivate how many of MillWheel's features are used. MillWheel's programming model provides a notion of logical time, making it simple to write time-based aggregations. MillWheel was designed from the outset with fault tolerance and scalability in mind. In practice, we find that MillWheel's unique combination of scalability, fault tolerance, and a versatile programming model lends itself to a wide variety of problems at Google.

allowing users to create massive distributed systems that are simply expressed. By allowing users to focus solely on their application logic, this kind of programming model allows users to reason about the semantics of their system without being distributed systems experts. In particular, users are able to depend on framework-level correctness and fault-tolerance guarantees as axiomatic, vastly restricting the surface area over which bugs and errors can manifest. Supporting a variety of common programming languages further drives adoption, as users can leverage the utility and convenience of existing libraries in a familiar idiom, rather than being restricted to a domain-specific language.

MillWheel is such a programming model, tailored specifically to streaming, low-latency systems. Users write application logic as individual nodes in a directed compute graph, for which they can define an arbitrary, dynamic topology. Records are delivered continuously along edges in the graph. MillWheel provides fault tolerance at the framework level, where any node or any edge in the topology can fail at any time without affecting the correctness of

图 10-22 MillWheel 论文

MillWheel 给我们带来最大的价值是之前列出的四个概念（数据精确一次性处理，持久化的状态存储，Watermark，持久定时器）为流式计算提供了工业级生产保障：即使在不可靠的商用硬件上，也可以对无序数据进行稳定的、低延迟的处理。

Kafka

我们开始讨论 Kafka（图 10-23）。Kafka 在本章讨论的系统中是独一无二的，因为它不是数据计算框架，而是数据传输和存储的工具。但是，毫无疑问，Kafka 在我们正在讨论的所有系统中扮演了推动流处理的最有影响力的角色之一。

如果你不熟悉它，我们可以简单描述为：Kafka 本质上是一个持久的流式数据传输和存储工具，底层系统实现为一组带有分区结构的日志型存储。它最初是由 Neha Narkhede 和 Jay Kreps 等业界大牛在 LinkedIn 公司内部开发的，其卓越的特性有：

- 提供一个干净的持久性模型，让大家在流式处理领域里面可以享受到批处理的产品特性，例如持久化、可重放。

- 在生产者和消费者之间提供弹性隔离。
- 我们在第 6 章中讨论过的流和表之间的关系，揭示了思考数据处理的基本方式，同时还提供了和数据库打通的思路和概念。
- 来自于上述所有方面的影响，不仅让 Kafka 成为整个行业中大多数流处理系统的基础，而且还促进了流处理数据库和微服务运动。

MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman,
Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle
Google

{takidau, alexgb, kayab, chernyak, haberman,
relax, sgmcl, millsdc, pgn, samuelw}@google.com

ABSTRACT

MillWheel is a framework for building low-latency data-processing applications that is widely used at Google. Users specify a directed computation graph and application code for individual nodes, and the system manages persistent state and the continuous flow of records, all within the envelope of the framework's fault-tolerance guarantees.

This paper describes MillWheel's programming model as well as its implementation. The case study of a continuous anomaly detector in use at Google serves to motivate how many of MillWheel's features are used. MillWheel's programming model provides a notion of logical time, making it simple to write time-based aggregations. MillWheel was designed from the outset with fault tolerance and scalability in mind. In practice, we find that MillWheel's unique combination of scalability, fault tolerance, and a versatile programming model lends itself to a wide variety of problems at Google.

allowing users to create massive distributed systems that are simply expressed. By allowing users to focus solely on their application logic, this kind of programming model allows users to reason about the semantics of their system without being distributed systems experts. In particular, users are able to depend on framework-level correctness and fault-tolerance guarantees as axiomatic, vastly restricting the surface area over which bugs and errors can manifest. Supporting a variety of common programming languages further drives adoption, as users can leverage the utility and convenience of existing libraries in a familiar idiom, rather than being restricted to a domain-specific language.

MillWheel is such a programming model, tailored specifically to streaming, low-latency systems. Users write application logic as individual nodes in a directed compute graph, for which they can define an arbitrary, dynamic topology. Records are delivered continuously along edges in the graph. MillWheel provides fault tolerance at the framework level, where any node or any edge in the topology can fail at any time without affecting the correctness of

图 10-23 Kafka 的时间轴

在这些特性中，有两个对我来说最为突出。第一个是流数据的持久化和可重放性的应用。在 Kafka 之前，大多数流处理系统使用某种临时、短暂的消息系统，如 Rabbit MQ 甚至是普通的 TCP 套接字来发送数据。数据处理的一致性往往通过生产者数据冗余备份来实现（即，如果下游数据消费者出现故障，则上游生产者将数据进行重新发送），但是上游数据的备份通常也是临时保存一下。大多数系统设计完全忽略在开发和测试中需要重新拉取数据重新计算的需求。但 Kafka 的出现改变了这一切。从数据库持久日志概念得到启发并将其应用于流处理领域，Kafka 让我们享受到了如同 Batch 数据源一样的安全性和可靠性。凭借持久化和可重放的特点，流计算在健壮性和可靠性上面又迈出关键的一步，为后续替代批处理系统打下基础。

作为一个流式系统开发人员，Kafka 的持久化和可重放功能对业界产生一个更有意思的变化就是：当今天大量流处理引擎依赖源头数据可重放来提供端到端精确一次的计算保障。可重放的特点是 Apex, Flink, Kafka Streams, Spark 和 Storm 的端到端精确一次保证的基础。当以精确一次模式执行时，每个系统都假设 / 要求输入数据源能够重放之前的部分数据（从最近 Checkpoint 到故障发生时的数据）。当流式处理系统与不具备重放能力的输入源一起使用时（哪怕是源头数据能够保证可靠的一致性数据投递，但不能提供重放功能），这种情况下无法保证端到端的完全一次语义。这种对可重放（以及持久化等其他特点）的广泛依赖是 Kafka 在整个行业中产生巨大影响的间接证明。

Kafka 系统中第二个值得注意的重点是流和表理论的普及。我们花了整个第 6 章以及第 8 章、第 9 章来讨论流和表，可以说流和表构成了数据处理的基础，无论是 MapReduce 及其演化系统，SQL 数据库系统，还是其他分支的数据处理系统。并不是所有的数据处理方法都直接基于流或者表来进行抽象，但从概念或者理论上说，表和流的理论就是这些系统的运作方式。作为这些系统的用户和开发人员，理解我们所有系统构建的核心基础概念意义重大。我们都非常感谢 Kafka 社区的开发者，他们帮助我们

更广泛更加深入地了解到批流理论。

如果您想了解更多关于 Kafka 及其理论核心，JackKreps 的《I love Logs》（O'Reilly; 图 10-24）是一个很好的学习资料。另外，正如第 6 章中引用的那样，Kreps 和 Martin Kleppmann 有两篇文章（图 10-25），我强烈建议您阅读一下关于流和表相关理论。

Kafka 为流处理领域做出了巨大贡献，可以说比其他任何单一系统都要多。特别是，对输入和输出流的持久性

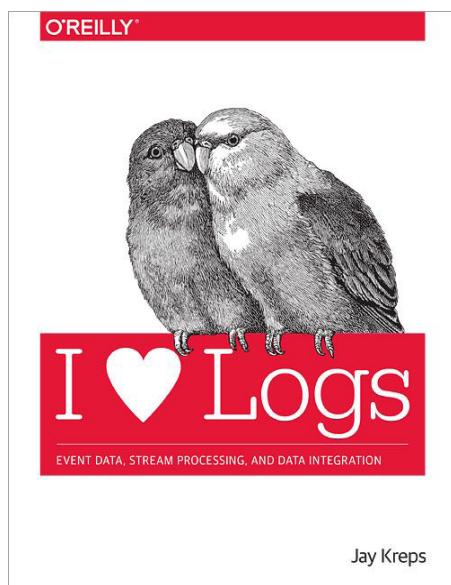


图 10-24 《I love Logs》

和可重放的设计，帮助将流计算从近似工具的小众领域发展到在大数据领域妇孺皆知的程度起了很大作用。此外，Kafka 社区推广的流和表理论对于数据处理引发了我们深入思考。

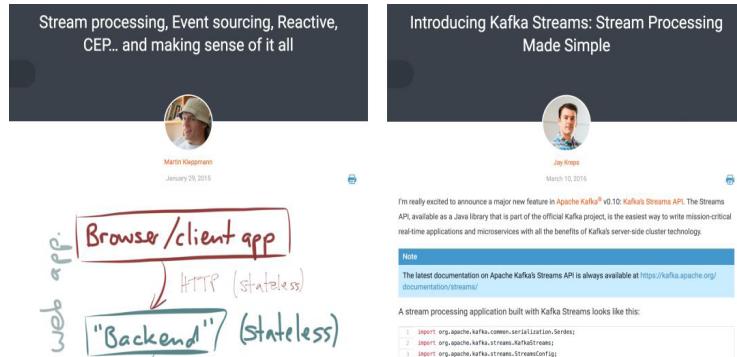


图10-25 Martin 的帖子(左边)以及 Jay 的帖子(右边)

DataFlow

Cloud Dataflow (图 10-26) 是 Google 完全托管的、基于云架构的数据处理服务。Dataflow 于 2015 年 8 月推向全球。DataFlow 将 MapReduce, Flume 和 MillWheel 的十多年经验融入其中，并将其打包成 Serverless 的云体验。

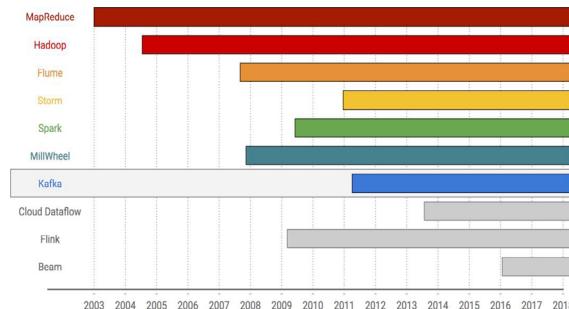


图 10-26 Google DataFlow 的时间轴

虽然 Google 的 Dataflow 的 Serverless 特点可能是从系统角度来看最具技术挑战性以及有别于其他云厂商产品的重要因素，但我想在此讨论主要是其批流统一的编程模型。编程模型包括我们在本书的大部分内容中所讨论的转换，窗口，水印，触发器和聚合计算。当然，所有这些讨论都包含

了思考问题的 what、where、when、how。

DataFlow 模型首先诞生于 Flume，因为我们希望将 MillWheel 中强大的无序数据计算能力整合到 Flume 提供的更高级别的编程模型中。这个方式可以让 Google 员工在内部使用 Flume 进行统一的批处理和流处理编程。

关于统一模型的核心关键思考在于，尽管在当时我们也没有深刻意识到，批流处理模型本质上没有区别：仅仅是在表和流的处理上有些小变化而已。正如我们在第 6 章中所讨论到的，主要的区别仅仅是在将表上增量的变化转换为流，其他一切在概念上是相同的。通过利用批处理和流处理两者大量的共性需求，可以提供一套引擎，适配于两套不同处理方式，这让流计算系统更加易于使用。

除了利用批处理和流处理之间的系统共性之外，我们还仔细查看了多年来我们在 Google 中遇到的各种案例，并使用这些案例来研究统一模型下系统各个部分。我们研究主要内容如下：

- 未对齐的事件时间窗口（如会话窗口），能够简明地表达这类复杂的分析，同时亦能处理乱序数据。
- 自定义窗口支持，系统内置窗口很少适合所有业务场景，需要提供给用户自定义窗口的能力。
- 灵活的触发和统计模式，能够满足正确性，延迟，成本的各项业务需求。
- 使用 Watermark 来推断输入数据的完整性，这对于异常检测等用例至关重要，其中异常检测逻辑会根据是否缺少数据做出异常判断。
- 底层执行环境的逻辑抽象，无论是批处理，微批处理还是流式处理，都可以在执行引擎中提供灵活的选择，并避免系统级别的参数设置（例如微批量大小）进入逻辑 API。

总之，这些平衡了灵活性，正确性，延迟和成本之间的关系，将 DataFlow 的模型应用于大量用户业务案例之中。

考虑到我们之前整本书都在讨论 DataFlow 和 Beam 模型的各类问题，我在此处重新给大家讲述这些概念纯属多此一举。但是，如果你正在寻找稍微更具学术性的内容以及一些应用案例，我推荐你看下 2015 年发表的《DataFlow 论文..》（图 10-27）。

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google

{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, mllsd, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g. Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run off-line experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also

图 10-27 DataFlow 的论文

DataFlow 还有不少可以大书特书的功能特点，但在这章内容构成来看，我认为 DataFlow 最重要的是构建了一套批流统一的大数据处理模型。DataFlow 为我们提供了一套全面的处理无界且无序数据集的能力，同时这套系统很好的平衡了正确性、延迟、成本之间的相互关系。

Flink

Flink（图 10-28）在 2015 年突然出现在大数据舞台，然后似乎在一夜之间从一个无人所知的系统迅速转变为人人皆知的流式处理引擎。

在我看来，Flink 崛起有两个主要原因：

- 采用 Dataflow/Beam 编程模型，使其成为完备语义功能的开源流式处理系统。
- 其高效的快照实现方式，源自 Chandy 和 Lamport 的原始论文“Distributed Snapshots: Determining Global States of Distributed

Systems”的研究，这为其提供了正确性所需的强一致性保证。

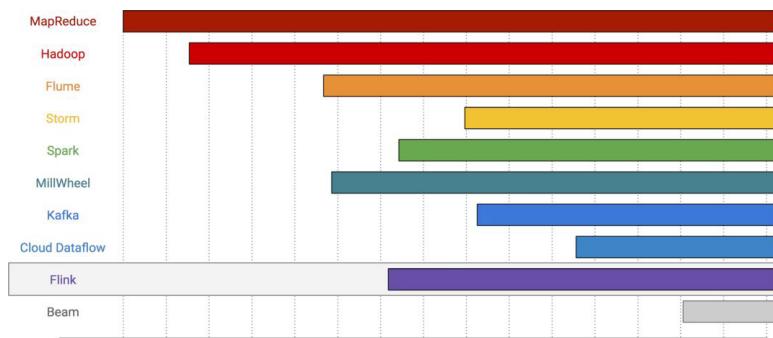


图 10-28 Flink 的时间轴

Reuven 在第 5 章中简要介绍了 Flink 的一致性机制，这里在重申一下，其基本思想是在系统中的 Worker 之间沿着数据传播路径上产生周期性 Barrier。这些 Barrier 充当了在不同 Worker 之间传输数据时的对齐机制。当一个 Worker 在其所有上游算子输入来源（即来自其所有上游一层的 Worker）上接收到全部 Barrier 时，Worker 会将当前所有 key 对应的状态写入一个持久化存储。这个过程意味着将这个 Barrier 之前的所有数据都做了持久化。

Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
 University of Texas at Austin
 and
 LESLIE LAMPORT
 Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are “computation has terminated,” “the system is deadlocked” and “all tokens in a token ring have disappeared.” The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—distributed applications; distributed databases; network operating systems; D.4.1 [Operating Systems]: Process Management—concurrency; deadlocks; multiprocessing/multiprogramming; mutual exclusion; scheduling; synchronization; D.4.5 [Operating Systems]: Reliability—backup procedures; checkpoint/restart; fault-tolerance; verification

General Terms: Algorithms

Additional Key Words and Phrases: Global States, Distributed deadlock detection, distributed systems, message communication systems

图 10-29 Chandy-Lamport 快照

通过调整 Barrier 的生成频率，可以间接调整 Checkpoint 的执行频

率，从而降低时延并最终获取更高的吞吐（其原因是做 Checkpoint 过程中涉及到对外进行持久化数据，因此会有一定的 IO 导致延时）。

Flink 既能够支持精确一次的语义处理保证，同时又能够提供支持事件时间的处理能力，这让 Flink 获取的巨大成功。接着，Jamie Grier 发表他的题为“《Extending the Yahoo! Streaming Benchmark》”（图 10-30）的文章，文章中描述了 Flink 性能具体的测试数据。在那篇文章中，杰米描述了两个令人印象深刻的特点：

构建一个用于测试的 Flink 数据管道，其拥有比 Twitter Storm 更高的准确性（归功于 Flink 的强一次性语义），但成本却降到了 1%。

Extending the Yahoo! Streaming Benchmark

February 2, 2016 - Flink Features, Resources

Jamie Grier



Until very recently, I've been working at Twitter and focusing primarily on stream processing systems. While researching the current state-of-the-art in stateful streaming systems I came across Apache Flink™. I've known for some time that having proper, fault-tolerant, managed state and exactly-once processing semantics with regard to that state was going to be a game changer for stream processing so when I came across Apache Flink™ I was understandably excited.

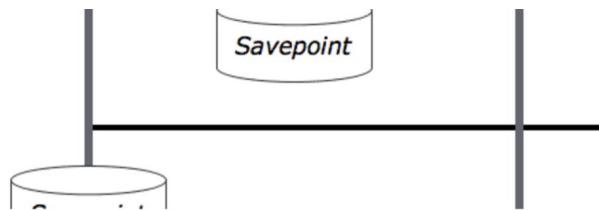
Shortly after my first introduction to Flink I saw that the [Flink Forward 2015](#) conference was about to be held in Berlin and I knew that I had to be there to learn more. I spent a few days in Berlin attending talks and having long discussions with the Flink committers and others in the Flink community. By the end of the conference it was apparent to me that Flink was far and away the most advanced stream processing system available in the open source world. I returned home determined to see what I could accomplish with Flink back at Twitter.

I had an application in mind that I knew I could make more efficient by a huge factor if I could use the stateful processing guarantees available in Flink so I set out to build a prototype to do exactly that. The end result of this was a new prototype system which computed a more accurate result than the previous one and also used less than 1% of the resources of the previous system. The better accuracy came from the fact that Flink provides exactly-once processing guarantees whereas the existing system only provided at-most-once. The efficiency improvements came from several places but the largest was the elimination of a large key-value store cluster needed for the existing system. This prototype system earned my [team](#) first prize in the infrastructure category at Twitter's December 2015 Hack Week competition!

图 10-30 《Extending the Yahoo! Streaming Benchmark》

2. Flink 在精确一次的处理语义参数设定下，仍然达到 Storm 的 7.5 倍吞吐量（而且，Storm 还不具备精确一次的处理语义）。此外，由于网络被打满导致 Flink 的性能受到限制；进一步消除网络瓶颈后 Flink 的吞吐量几乎达到 Storm 的 40 倍。

从那时起，许多其他流式处理项目（特别是 Storm 和 Apex）都采用了类似算法的数据处理一致性机制。



Savepoints: Turning Back Time

October 14, 2016 - Flink Features, Resources

Fabian Hueske and Michael Winters



This post is the first in a series where the data Artisans team will highlight some of Apache Flink's® core features. By Fabian Hueske (@fhueske) and Mike Winters (@wints)

Stream processing is commonly associated with 'data in motion', powering systems that make sense of and respond to data in nearly the same instant it's created. The most frequently discussed streaming topics, such as latency and throughput or watermarks and handling of late data, focus on the present rather than the past.

In reality, though, there are a number of cases where you'll need to reprocess data that your streaming application has already processed before. Some examples include:

- Deployment of a new version of your application with a new feature, a bug fix, or a better machine learning model
- A/B testing different versions of an application using the same source data streams, starting the

图 10-31 《Savepoints: Turning Back Time》

通过快照机制，Flink 获得了端到端数据一致性。Flink 更进了一步，利用其快照的全局特性，提供了从过去的任何一点重启整个管道的能力，这一功能称为 SavePoint（在 Fabian Hueske 和 Michael Winters 的帖子 [《Savepoints: Turning Back Time》 (<https://data-artisans.com/blog/turning-back-time-savepoints>)] 中有所描述，[图 10-31]）。Savepoints 功能参考了 Kafka 应用于流式传输层的持久化和可重放特性，并将其扩展应用到整个底层 Pipeline。流式处理仍然遗留大量开放性问题有待优化和提升，但 Flink 的 Savepoints 功能是朝着正确方向迈出的第一步，也是整个行业非常有特点的一步。如果您有兴趣了解有关 Flink 快照和保存点的系统构造的更多信息，请参阅《State Management in Apache Flink》（图 10-32），论文详细讨论了相关的实现。

State Management in Apache Flink®

Consistent Stateful Distributed Stream Processing

Paris Carbone[†]
Seif Haridi[†]

Stephan Ewen[‡]
Stefan Richter[‡]

Gyula Fóra^{*}
Kostas Tzoumas^{*}

[†]KTH Royal Institute of Technology
(parisc, haridi)@kth.se

^{*}King Digital Entertainment Limited
gyula.fora@king.com

[‡]data Artisans
{stephan.srichter,kostas}@data-artisans.com

ABSTRACT

Stream processors are emerging in industry as an apparatus that drives analytical but also mission critical services handling the core of persistent application logic. Thus, apart from scalability and low-latency, a rising system need is first-class support for application state together with strong consistency guarantees, and adaptivity to cluster reconfigurations, software patches and partial failures. Although prior systems research has addressed some of these specific problems, the practical challenge lies on how such guarantees can be materialized in a transparent, non-intrusive manner that relieves the user from unnecessary constraints. Such needs served as the main design principles of state management in Apache Flink, an open source, scalable stream processor.

We present Flink's core pipelined, in-flight mechanism which guarantees the creation of lightweight, consistent, distributed snapshots of application state, progressively, without impacting continuous execution. Consistent snapshots cover all needs for system reconfiguration, fault tolerance and version management through coarse grained rollback recovery. Application state is declared explicitly to the system, allowing efficient partitioning and transparent commits to persistent storage. We further present Flink's backend implementations and mechanisms for high availability, external state queries and output commit. Finally, we demonstrate how these mechanisms behave in practice with metrics and large-deployment insights exhibiting the low performance trade-offs of our approach and the general benefits of exploiting asynchrony in continuous, yet sustainable system deployments.

as a paradigm to implement both analytical applications on “real-time” data, but also as a paradigm to implement data-driven applications and services that would otherwise interact with a shared external database for their data access needs. The stream processing paradigm is more friendly to modern organizations that separate engineering teams vertically, each team being responsible for a specific feature or application, as it allows state to be distributed and co-located with the application instead of forcing teams to collaborate by sharing access to the database. Further, stream processing is a natural paradigm for *event-driven* applications that need to react fast to real-world events and communicate with each other via message passing.

In point of fact, stream processing is not a new concept; it has been an active research topic for the database community in the past [29, 26, 17, 21] and some (but not all) of the ideas that underpin modern stream processing technology are inspired by that research. However, what we see today is widespread adoption of stream processing across the enterprise beyond niche applications where stream processing and Complex Event Processing systems were traditionally used. There are many reasons for this: first, new stream processing technologies allow for massive scale-out, similar to MapReduce [31] and related technologies [46, 20, 22]. Second, the amount of data that is generated in the form of event streams is exploding. Processing needs now spread beyond financial transactions, to user activity in websites and mobile apps, as well as data generated by machines and sensors in manufacturing plants, cars, home devices, etc. Third, many modern state-of-the-art stream processing systems are open source allowing widespread adoption in

图 10-32 《State Management in Apache Flink》

除了保存点之外，Flink 社区还在不断创新，包括将第一个实用流式 SQL API 推向大规模分布式流处理引擎的领域，正如我们在第 8 章中所讨论的那样。总之，Flink 的迅速崛起成为流计算领军角色主要归功于三个特点：

1. 整合行业里面现有的最佳想法（例如，成为第一个开源 DataFlow/Beam 模型）
2. 创新性在表上做了大量优化，并将状态管理发挥更大价值，例如基于 Snapshot 的强一致性语义保证，Savepoints 以及流式 SQL。
3. 迅速且持续地推动上述需求落地。

另外，所有这些改进都是在开源社区中完成的，我们可以看到为什么 Flink 一直在不断提高整个行业的流计算处理标准。

Beam

我们今天谈到的最后一个系统是 Apache Beam（图 10-33）。Beam 与本章中的大多数其他系统的不同之处在于，它主要是编程模型，API 设计

和可移植层，而不是带有执行引擎的完整系统栈。但这正是我想强调的重点：正如 SQL 作为声明性数据处理的通用语言一样，Beam 的目标是成为程序化数据处理的通用语言。

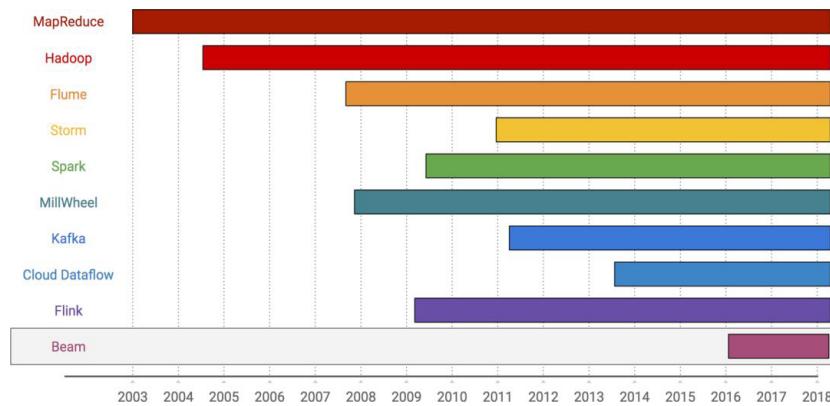


图 10-33 Apache Beam 的时间轴

具体而言，Beam 由许多组件组成：

一个统一的批量加流式编程模型，继承自 Google DataFlow 产品设计，以及我们在本书的大部分内容中讨论的细节。该模型独立于任何语言实现或 runtime 系统。您可以将此视为 Beam 等同于描述关系代数模型的 SQL。

一组实现该模型的 SDK（软件开发工具包），允许底层的 Pipeline 以不同 API 语言的惯用方式编排数据处理模型。Beam 目前提供 Java, Python 和 Go 的 SDK，可以将它们视为 Beam 的 SQL 语言本身的程序化等价物。

一组基于 SDK 的 DSL（特定于域的语言），提供专门的接口，以独特的方式描述模型在不同领域的接口设计。SDK 来描述上述模型处理能力的全集，但 DSL 描述一些特定领域的处理逻辑。Beam 目前提供了一个名为 Scio 的 Scala DSL 和一个 SQL DSL，它们都位于现有 Java SDK 之上。

一组可以执行 Beam Pipeline 的执行引擎。执行引擎采用 Beam SDK 术语中描述的逻辑 Pipeline，并尽可能高效地将它们转换为可以执行的物理计划。目前，针对 Apex, Flink, Spark 和 Google Cloud Dataflow 存在对

应的 Beam 引擎适配。在 SQL 术语中，您可以将这些引擎适配视为 Beam 在各种 SQL 数据库的实现，例如 Postgres，MySQL，Oracle 等。

Beam 的核心愿景是实现一套可移植接口层，最引人注目的功能之一是它计划支持完整的跨语言可移植性。尽管最终目标尚未完全完成（但即将面市），让 Beam 在 SDK 和引擎适配之间提供足够高效的抽象层，从而实现 SDK 和引擎适配之间的任意切换。我们畅想的是，用 JavaScript SDK 编写的数据 Pipeline 可以在用 Haskell 编写的引擎适配层上无缝地执行，即使 Haskell 编写的引擎适配本身没有执行 JavaScript 代码的能力。

作为一个抽象层，Beam 如何定位自己和底层引擎关系，对于确保 Beam 实际为社区带来价值至关重要，我们也不希望看到 Beam 引入一个不必要的抽象层。这里的关键点是，Beam 的目标永远不仅仅是其所有底层引擎功能的交集（类似最小公分母）或超集（类似厨房水槽）。相反，它旨在为整个社区大数据计算引擎提供最佳的想法指导。这里面有两个创新的角度：

Beam 本身的创新

Beam 将会提出一些 API，这些 API 需要底层 runtime 改造支持，并非所有底层引擎最初都支持这些功能。这没关系，随着时间的推移，我们希望许多底层引擎将这些功能融入未来版本中；对于那些需要这些功能的业务案例来说，具备这些功能的引擎通常会被业务方选择。

这里举一个 Beam 里面关于 SplittableDoFn 的 API 例子，这个 API 可以用来实现一个可组合的，可扩展的数据源。（具体参看 Eugene Kirpichov 在他的文章《“Powerful and modular I/O connectors with Splittable DoFn in Apache Beam”》中描述 [图 10-34]）。它设计确实很有特点且功能强大，目前我们还没有看到所有底层引擎对动态负载均衡等一些更具创新性功能进行广泛支持。然而，我们预计这些功能将随着时间的推移而持续加入底层引擎支持的范围。

底层引擎的创新

底层引擎适配可能会引入底层引擎所独特的功能，而 Beam 最初可能

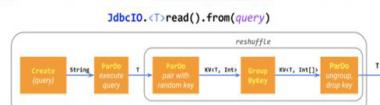
Powerful and modular IO connectors with Splittable DoFn in Apache Beam

Aug 16, 2017 · Eugene Kirpichev

One of the most important parts of the Apache Beam ecosystem is its quickly growing set of connectors that allow Beam pipelines to read and write data to various data storage systems ("IOs"). Currently, Beam ships over 20 IO connectors with many more in active development. As user demands for IO connectors grew, our work on improving the related Beam APIs (in particular, the Source API) produced an unexpected result: a generalization of Beam's most basic primitive, `DoFn`.

Connectors as mini-pipelines

One of the main reasons for this vibrant IO connector ecosystem is that developing a basic IO is relatively straightforward: many connector implementations are simply mini-pipelines (composite PTransforms) made of the basic Beam `ParDo` and `GroupByKey` primitives. For example, `ElasticsearchIO.write()` expands into a single `ParDo` with some batching for performance; `JdbcIO.read()` expands into `Create`, `of(query)`, a reshuffle and `ParDo(execute sub-query)`. Some IOs construct considerably more complicated pipelines.



This "mini-pipeline" approach is flexible, modular, and generalizes to data sources that read from a dynamically computed PCollection of locations, such as `SpannerIO.readAll()` which reads the results of a PCollection of queries from Cloud Spanner; compared to `SpannerIO.read()` which executes a single query. We believe such dynamic data sources are a very useful capability, often overlooked by other data processing frameworks.

When ParDo and GroupByKey are not enough

Despite the flexibility of `ParDo`, `GroupByKey` and their derivatives, in some cases building an efficient IO connector requires extra capabilities.

For example, imagine reading files using the sequence `ParDo(filepattern -> expand into files)`, `ParDo(filename -> read records)`, or reading a Kafka topic using `ParDo(topic -> list partitions), ParDo(topic, partition -> read records)`. This approach has two big issues:

- In the file example, some files might be much larger than others, so the second `ParDo` may have very long individual `@ProcessElement` calls. As a result, the pipeline can suffer from poor performance due to stragglers.
- In the Kafka example, implementing the second `ParDo` is simply impossible with a regular `DoFn`, because it would need to output an infinite number of records per each input element `topic, partition` (*stateful processing comes close, but it has other limitations that make it insufficient for this task*).

Beam Source API

Apache Beam historically provides a Source API (`BoundedSource` and `UnboundedSource`) which does not have these limitations and allows development of efficient data sources for batch and streaming systems. Pipelines use this API via the `Read`, `FromSource` built-in PTransform.

The Source API is largely similar to that of most other data processing frameworks, and allows the system to read data in parallel using multiple workers, as well as checkpoint and resume reading from an unbounded data source. Additionally, the Beam `BoundedSource` API provides advanced features such as progress reporting and dynamic rebalancing (which together enable autoscaling), and `UnboundedSource` supports reporting the source's watermark and backlog (until SOF, we believed that "batch" and "streaming" data sources are fundamentally different and thus require

图 10-34 《Powerful and modular I/O connectors with Splittable DoFn in Apache Beam》

并未提供 API 支持。这没关系，随着时间的推移，已证明其有用性的引擎功能将在 Beam API 逐步实现。

这里的一个例子是 Flink 中的状态快照机制，或者我们之前讨论过的 Savepoints。Flink 仍然是唯一一个以这种方式支持快照的公开流处理系统，但是 Beam 提出了一个围绕快照的 API 建议，因为我们相信数据 Pipeline 运行时优雅更新对于整个行业都至关重要。如果我们今天推出这样的 API，Flink 将是唯一支持它的底层引擎系统。但同样没关系，这里的重点是随着时间的推移，整个行业将开始迎头赶上，因为这些功能的价值会逐步为人所知。这些变化对每个人来说都是一件好事。

通过鼓励 Beam 本身以及引擎的创新，我们希望推进整个行业快速演化，而不用再接受功能妥协。通过实现跨执行引擎的可移植性承诺，我们希望将 Beam 建立为表达程序化数据处理流水线的通用语言，类似于当今 SQL 作为声明性数据处理的通用处理方式。这是一个雄心勃勃的目标，我们并没有完全实现这个计划，到目前为止我们还有很长的路要走。

总结

我们对数据处理技术的十五年发展进行了蜻蜓点水般的回顾，重点关注那些推动流式计算发展的关键系统和关键思想。来，最后，我们再做一次总结：

MapReduce: 可扩展性和简单性 通过在强大且可扩展的执行引擎之上提供一组简单的数据处理抽象，MapReduce 让我们的数据工程师专注于他们的数据处理需求的业务逻辑，而不是去构建能够适应在一大堆普通商用服务器上的大规模分布式处理程序。

Hadoop: 开源生态系统 通过构建一个关于 MapReduce 的开源平台，无意中创建了一个蓬勃发展的生态系统，其影响力所及的范围远远超出了其最初 Hadoop 的范围，每年有大量的创新性想法在 Hadoop 社区蓬勃发展。

Flume: 管道及优化 通过将逻辑流水线操作的高级概念与智能优化器相结合，Flume 可以编写简洁且可维护的 Pipeline，其功能突破了 MapReduce 的 Map→Shuffle→Reduce 的限制，而不会牺牲性能。

Storm: 弱一致性，低延迟 通过牺牲结果的正确性以减少延迟，Storm 为大众带来了流计算，并开创了 Lambda 架构的时代，其中弱一致的流处理引擎与强大一致的批处理系统一起运行，以实现真正的业务目标低延迟，最终一致型的结果。

Spark: 强一致性 通过利用强大一致的批处理引擎的重复运行来提供无界数据集的连续处理，Spark Streaming 证明至少对于有序数据集的情况，可以同时具有正确性和低延迟结果。

MillWheel: 乱序处理 通过将强一致性、精确一次处理与用于推测时间的工具（如水印和定时器）相结合，MillWheel 做到了无序数据进行准确的流式处理。

Kafka: 持久化的流式存储，流和表对偶性 通过将持久化数据日志的概念应用于流传输问题，Kafka 支持了流式数据可重放功能。通过对流和表

理论的概念进行推广，阐明数据处理的概念基础。

Cloud Dataflow: 统一批流处理引擎 通过将 MillWheel 的无序流式处理与高阶抽象、自动优化的 Flume 相结合，Cloud Dataflow 为批流数据处理提供了统一模型，并且灵活地平衡正确性、计算延迟、成本的关系。

Flink: 开源流处理创新者 通过快速将无序流式数据处理的强大功能带到开源世界，并将其与分布式快照及保存点功能等自身创新相结合，Flink 提高了开源流处理的业界标准并引领了当前流式处理创新趋势。

Beam: 可移植性 通过提供整合行业最佳创意的强大抽象层，Beam 提供了一个可移植 API 抽象，其定位为与 SQL 提供的声明性通用语言等效的程序接口，同时也鼓励在整个行业中推进创新。

可以肯定的说，我在这里强调的这 10 个项目及其成就的说明并没有超出当前大数据的历史发展。但是，它们对我来说是一系列重要且值得注意的大数据发展里程碑，它共同描绘了过去十五年中流处理演变的时间轴。自最早的 MapReduce 系统开始，尽管沿途有许多起伏波折，但不知不觉我们已经走出来很长一段征程。即便如此，在流式系统领域，未来我们仍然面临着一系列的问题亟待解决。正所谓：路漫漫其修远兮，吾将上下而求索。

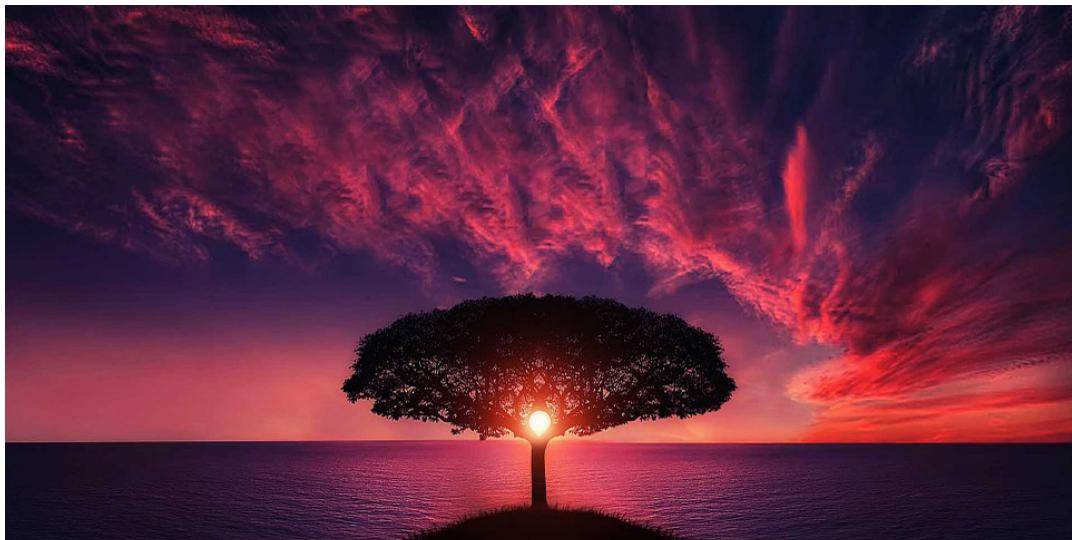
译者简介

陈守元（花名：巴真），阿里巴巴高级产品专家。阿里巴巴实时计算团队产品负责人，2010 年毕业即加入阿里集团参与淘宝数据平台建设，近 10 年的大数据从业经验，开源项目 Alibaba DataX 发起人，当前负责阿里实时计算产品 Flink 的规划与设计，致力于推动 Flink 成为下一代大数据处理标准。

《Streaming System》一书目前正由阿里巴巴实时计算团队进行翻译，预计今年年底上市，对流式系统感兴趣的同学可以关注。

Serverless 基础架构下， 运维人员将何去何从？

作者 Tom McLaughlin，译者 无明



正如serverless一词会让人感到困惑，运维一词也是如此。当谈话中出现这个术语时，人们会想到不同的东西，这会让谈话变得混乱。

什么是serverless？serverless是一种架构理念，最新的定义这样描述它：“serverless架构是基于互联网的系统，其中应用开发不使用常规的服务进程。相反，它们仅依赖于第三方服务，客户端逻辑和服务托管远程过程调用的组合。”

运维有着很广泛的定义。通常，人们会在谈话中提到一个不存在运维的场景，但实际上，他们建议作为替代品的东西在我看来仍然是运维。当我们甚至无法就所谈论的内容达成一致时，我们该如何讨论serverless对运

维所带来的影响呢？

在我们对所谈论的内容有了共同的理解之后，让我们来看看运维人员在serverless中应该属于处于什么样的位置。我认为运维团队在serverless环境中没有多大意义。但是，运维人员其实是有价值的。那么他们应该处在什么样的位置上？

什么是运维？

人们对运维的理解通常是不一样的，但通常都是正确的。选择什么样的定义取决于一个人的经验、需求和优先事项，以及他们看问题的角度。然而，这些不同的定义往往会导致人们的交谈相互脱节。

从最高层面看，运维是一种实践，让支撑业务的技术保持运作。但如果你深挖一下，运维一词可以以用在许多方面。因为这些含义紧密耦合了很长时间，人们往往会把它们混为一谈。

什么是运维？运维是：

- 一个团队
- 一种角色
- 一种责任
- 一组任务

通常，这个角色由运维团队的运维工程师承担，他们负责执行与运维相关的任务。近年来，DevOps的出现极大改变了这种局面。曾经有关所有这些定义的死板结构被拆解，由此，定义本身也就分崩离析。

在DevOps频谱的一端，运维团队和个体角色基本保持不变。开发人员和运维人员都是独立的团队，只是开发团队开始承担一部分的运维职责，运维团队和开发团队之间的沟通比之前上升了一个层次。当我们听到有人说“开发人员先收到警报”或开发人员不能再让代码“翻墙”时，我们知道，运维职责发生了变化。

在DevOps频谱的另一端没有了运维团队和个体角色。有些企业将具备运维和软件开发技能的工程师组合在一起，创建了跨职能团队。这些企

业没有运维团队，也没有计划组建一个。

在DevOps频谱的中间，运维角色各不相同。在一些企业中，除了增加了自动化技能（例如Puppet、Ansible和Chef）之外，其他几乎没有变化。其他团队已经将自动化视为强化运维职责的手段。在某些情况下，运维工程师更接近于开发人员——掌握了配置管理和其他工具的开发人员。

那么serverless对于这些运维定义有什么影响？

Serverless 之下的运维会变成什么？

以下是我对serverless运维的未来的看法：

- 运维团队将会消失。
- 运维工程师将被开发团队吸收。
- 运维工程师将负责应用程序栈的深层需求。

Serverless运维并不是NoOps，也不是反DevOps。如果传统的运维团队被解散，原先的运维工程师需要新的去处。他们的去处将是各个开发团队，而这些团队会是产品团队或职能团队。

这样，能够处理开发和运维的跨职能团队将会迅速崛起。最后，很多运维工程师会发现自己的优势将比过去更多地被应用到应用程序中。

为什么要解散运维团队？

在DevOps出现之前，我们看到的是两个孤岛：一个是开发，一个是运维，他们之间隔着一堵墙，开发人员将工程设计扔给毫无戒心的运维团队是日常。DevOps出现了之后，中间那堵墙被拆掉了，两个团队之间的协作变得更加紧密。

但实际上，“拆掉中间那堵墙”对于不同的组织而言也有不同的含义。在某些情况下，它只是意味着更多的会议。现在，有人在向运维“扔东西”之前会先告知他们。

但是，你仍然需要独立团队，让他们共同努力交付解决方案。实际上，这可能涉及两个以上的团队。

还有谁？

- 需要一个项目或产品经理负责监督交付过程，确保交付的东西能够满足公司的需求。如果你是在一家产品或SaaS公司，可能还需要UX和设计师确保产品的可用性和外观。
- 可能涉及多个开发团队，前端和后端开发可能由不同的团队负责。所有这些独立的团队需要共同努力来提供解决方案。

特别是产品和SaaS公司意识到整个过程效率低下。因此，组织开始重新调整团队，从职能团队转向产品、功能或问题领域。这些功能团队、产品团队或其他任何跨职能的团队都在解决特定领域的问题。

这些团队现在看起来是怎样的？在我看来，它们通常类似这样：

- 产品或项目经理
- 工程主管
- 前端开发人员
- 后端开发人员
- 产品设计师和/或用户体验研究员（通常是同一个人）

产品或项目经理（PM）是业务需求的所有者和代表。PM的工作是将业务需求转化为明确的目标，同时引导团队提出促进成功的想法。产品设计师或UX研究人员与PM合作，收集用户数据并将想法转化为设计和原型。技术负责人负责领导工程任务，估算所涉及的技术工作，并适当地为前端和后端工程师提供指导。

你最终得到的是一支具备多种能力的团队，他们都朝着同一个方向前进，从始至终都是这个过程的一部分。这些跨职能的技能让团队变得更加强大，从而提供更好的解决方案和服务。

然而，运维往往被排除在重新调整之外（虽然有时候运维会组建属于自己的跨职能团队，并为其他团队提供服务）。团队中的个人通常难以承担基础设施的运维需求。因此，当组织的其他部分进行重新调整时，运维仍然是一个独立的团队。

这已经很好地运作了很长一段时间。对于很多团队来说，基础设施并

不简单，他们无法在不影响主要问题领域的情况下进行可靠的交付和运维。

但serverless颠覆了这种关系。现在，开发人员可以轻松地交付自己的云基础设施。事实上，他们需要这么做，因为serverless将基础设施配置和应用程序代码结合在一起。

开发团队不需要运维团队的帮助来交付解决方案。单独的运维团队无法在不回归到守门人角色的情况下将自己插入到服务流程中。但多年来，守门人角色已经在很多组织中消失了。

运维团队的改变不是由serverless技术驱动的，而是由serverless对组织及其员工的运作方式和履行职责的影响驱动的。

当开发人员不再需要基础设施时，运维团队在很大程度上将变得可有可无。当他们遇到小问题时不会再来找你，他们可以自己解决这些问题。

运维团队不再是服务交付流程的一部分，这是迟早的事。到最后，有人会问为什么这些团队依然存在。当人们质疑你的工作为什么还会存在时，你的处境将变得很尴尬。

但从职责和任务方面来看，运维仍然很重要。构建serverless系统仍然需要这些功能。运维团队的作用在减弱，但对他们的技能仍然有需求，因此需要重新思考运维人员应该被放在什么位置上。这就是为什么说是时候解散运维团队并让成员加入产品团队和功能团队。

产品团队中的运维角色

作为产品团队成员的运维工程师将扮演怎样的角色？他们的主要职责是负责团队服务和系统的健康。这并不意味着他们每次都是第一个收到警报的人，他们应该是这些领域的专家。

软件开发人员仍然专注于各个组件，而运维工程师专注于整个系统。他们将采取整体方法确保整个系统正常可靠地运行。开发人员就可以花更少的时间在运维上，就会有更多的时间用于功能开发。

运维工程师还可以为团队提供帮助。虽然他们的主要职责是确保团队

服务的可靠性，但在必要的时候，也可以承担其他角色的职责。

对于DevOps这个术语，有很多定义和实现，但对于我来说，DevOps团队的形成才是这个术语最有力的表现。DevOps是实现成果和价值的人、流程和工具。

我们很早就意识到协作和跨职能团队在促进成功方面的价值。对于我来说，解散运维团队并让成员加入到跨职能团队中是交付价值的最有效手段。

与将人们放在单独的团队中相比，你会让合作变得有多紧密？Serverless将帮助我们实现许多人多年来一直试图实现的目标。

Linux 4.1 内核热补丁成功实践

作者 UCloud 内核团队

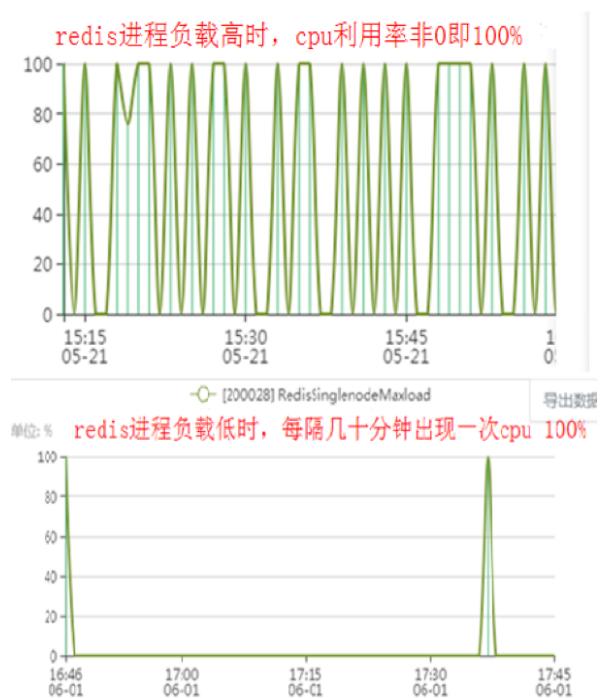


最开始公司运维同学反馈，个别宿主机上存在进程CPU峰值使用率异常的现象。而数万台机器中只出现了几例，也就是说万分之几的概率。监控产生的些小误差，不会造成宕机等严重后果，很容易就此被忽略了。但我们考虑到这个异常转瞬即逝、不易被察觉，可能还存在更多这样的机器，又或者现在正常将来又不正常，内核研发本能的好奇心让我们感到：此事必有蹊跷！于是追查下去。

问题现象

现象一：CPU监控非0即100%

该问题现象表现在Redis进程CPU监控的峰值时而100%时而为0，有的甚至是几十分钟都为0，突发1秒100%后又变为0，如下图。而从大量机器的统计规律看，这个现象在2.6.32内核不存在，在4.1内核存在几例。2.6.32是我们较早期采用的版本，为平台的稳定发展做了有力支撑，4.1可以满足很多新技术需求，如新款CPU、新板卡、RDMA、NVMe和binlog2.0等。后台无缝维护着两个版本，并为了能力提升和优化而逐步向4.1及更高版本过渡。



现象二：top显示非0即300%

登录到机器上执行`top -b -d 1 -p | grep`，可以看到进程的CPU利用率每隔几分钟到几十分钟出现一次300%，这意味着该进程3个线程占用的3个CPU都跑满了，跟监控程序呈现同样的异常。

30252	root	20	0	163m	119m	860	R	0.0	0.1	11675:32
30259	root	20	0	163m	119m	860	S	0.0	0.1	1:06.38
30259	root	20	0	163m	119m	860	R	0.0	0.1	11675:32
30261	root	20	0	163m	119m	860	S	0.0	0.1	1:06.38
30262	root	20	0	163m	119m	860	S	0.0	0.1	19:40.12
30259	root	20	0	163m	119m	860	S	300.0	0.1	11675:56
30259	root	20	0	163m	119m	860	S	0.0	0.1	11675:56
30259	root	20	0	163m	119m	860	R	0.0	0.1	11675:56
30259	root	20	0	163m	119m	860	S	0.0	0.1	11675:56
30259	root	20	0	163m	120m	860	S	0.0	0.1	11675:56
30261	root	20	0	163m	120m	860	S	0.0	0.1	1:06.38

问题分析

上述异常程序使用的是同样的数据源：`/proc/pid/stat`中进程运行占用的用户态时间`utime`和内核态时间`stime`。我们抓取`utime`和`stime`更新情况

后，发现utime或者stime每隔几分钟或者几十分钟才更新，更新的步进值达到几百到1000+，而正常进程看到的是每几秒更新，步进值是几十。

定位到异常点后，还要找出原因。排除了监控逻辑、IO负载、调用瓶颈等可能后，确认是4.1内核的CPU时间统计有bug

cputime统计逻辑

检查/proc/pid/stat中utime和stime被更新的代码执行路径，在cputime_adjust()发现了一处可疑的地方：

```
/*
 * Update userspace visible utime/stime values only if actual execution
 * time is bigger than already exported. Note that can happen, that we
 * provided bigger values due to scaling inaccuracy on big numbers.
 */
if (prev->stime + prev->utime >= rtime)
    goto out;
```

当utime+stime>=rtime的时候就直接跳出了，也就是不更新utime和stime了！这里的rtime是runtime，代表进程运行占用的所有CPU时长，正常应该等于或近似进程用户态时间+内核态时间。但内核配置了CONFIGVIRTCPUACCOUNTINGGEN选项，这会让utime和stime分别单调增长。而runtime是调度器里统计到的进程真正运行总时长。

内核每次更新/proc/pid/stat的utime和stime的时候，都会跟rtime对比。如果utime+stime很长一段时间都大于rtime，那代码直接goto out了，/proc/pid/stat就不更新了。只有当rtime持续更新追上utime+stime后，才更新utime和stime。

冷补丁和热补丁

第一回合：冷补丁

出现问题的代码位置已经找到，那就先去内核社区看看有没有成熟补丁可用，看一下kernel/sched/cputime.c的changelog，看到一个patch：确

98323534	98323492
98323534	98323497
98323534	98323503
98323534	98323509
98323534	98323515
98323534	98323520
98323534	98323525
98323534	98323530
98323534	98323535
98323539	98323540
98323545	98323546
98323551	98323552
98323558	98323559
98323564	98323565

保 $stime+utime=rtime$ 。再看描述：像top这样的工具，会出现超过100%的利用率，之后又一段时间为0，这不就是我们遇到的问题吗？真是踏破铁鞋无觅处，得来全不费工夫！（[patch链接](#)）

该补丁在4.3内核及以后版本才提交，却并未提交到4.1稳定版分支，于是移植到4.1内核。打上该补丁后进行压测，再没出现cpu time时而100%时而0%的现象，而是0-100%之间平滑波动的值。

至此，你可能觉得问题已经解决了。但是，问题才解决了一半。而往往“但是”后边才是重点。

第二回合：热补丁

给内核代码打上该冷补丁只能解决新增服务器的问题，但公司还有数万存量服务器是无法升级内核后重启的。

如果没有其它好选择，那存量更新将被迫采用如下的妥协方案：监控程序修改统计方式进行规避，不再使用utime和stime，而是通过runtime来统计进程的执行时间。

虽然该方案快速可行，但也有很大的缺点：1. 很多业务部门都要修改统计程序，研发成本较高；2. /proc/pid/stat的utime和stime是标准统计方式，一些第三方组件不容易修改；3. 并没有根本解决utime和stime不准的问题，用户、研发、运维使用ps、top命令时还会产生困惑，产生额外的沟通协调成本。

幸好，我们还可以依靠UCloud已多次成功应用的技术：热补丁技术。

所谓热补丁技术，是指在有缺陷的服务器内核或进程正在运行时，对已经加载到内存的程序二进制打上补丁，使得程序实时在线状态下执行新的正确逻辑。可以简单理解为像关二爷那样不打麻药在清醒状态下刮骨疗伤。当然，对内核刮骨疗伤内核是不会痛的，但刮不好内核就会直接死给你看，没有丝毫犹豫，非常干脆利索又耿直。

热补丁修复

而本次热补丁修复存在两个难点：

[Commit message \(Expand\)](#)

```
sched/cputime: Fix NO_HZ_FULL getrusage() monotonicity regression
sched/cputime: Fix steal_account_process_tick() to always return jiffies
sched/cputime: Fix invalid gtime in proc
kvm/x86: Hyper-V HV_X64_MSR_VP_RUNTIME support
sched/cputime: Guarantee stime + utime == rtime
sched, timer: Convert usages of ACCESS_ONCE() in the scheduler to READ_ONCE()...
sched, time: Fix build error with 64 bit cputime_t on 32 bit systems
```

sched/cputime: Guarantee stime + utime == rtime

While the current code guarantees monotonicity for stime and utime independently of one another, it does not guarantee that the sum of both is equal to the total time we started out with.

This confuses things (and peoples) who look at this sum, like top, and will report >100% usage followed by a matching period of 0%.

Rework the code to provide both individual monotonicity and a coherent sum.

难点一：热补丁制作

这次热补丁在结构体新增了spinlock成员变量，那就涉及新成员的内存分配和释放，在结构体实例被复制和释放时，都要额外的对新成员做处理，稍有遗漏可能会造成内存泄漏进而导致宕机，这就加大了风险。

再一个就是，结构体实例是在进程启动时初始化的，对于已经存在的实例如何塞进新的spinlock成员？所谓兵来将挡水来土掩，我们想到可以在原生补丁使用spinlock成员的代码路径上拦截，如果发现实例不含该成员，则进行分配、初始化、加锁、释放锁。

要解决问题，既要攀登困难的山峰，又得控制潜在的风险。团队编写了脚本进行几百万次的加载、卸载热补丁测试，并无内存泄漏，单机稳定运行，再下一城。

难点二：难以复现

另一个难题是该问题难以复现，只有在现网生产环境才有几个case可验证热补丁，而又不可以拿用户的环境去冒险。针对这种情况我们已经有标准化处理流程去应对，那就是设计完善的灰度策略，这也是UCloud内部一直在强调的核心理念和能力。经过分析，这个问题可以拆解为验证

热补丁稳定性和验证热补丁正确性。于是我们采取了如下灰度策略： 1. 稳定性验证：先拿几台机器测试正常，再拿公司内部500台次级重要的机器打热补丁，灰度运行几天正常，从而验证了稳定性，风险尽在掌控之中。 2. 正确性验证：找到一台出现问题的机器，同时打印utime+stime以及rtime，根据代码的逻辑，当rtime小于utime+stime时会执行老逻辑，当rtime大于utime+stime时会执行新的热补丁逻辑。如下图所示，进入热补

```
loaded core module
loading patch module: kpatch-cputime-hotpatch.ko-15
disabling patch module: kpatch_cputime_hotpatch
unloading patch module: kpatch_cputime_hotpatch
kpatch unload SUCCESS
test load/unload count=1506104
loaded core module
loading patch module: kpatch-cputime-hotpatch.ko-15
disabling patch module: kpatch_cputime_hotpatch
unloading patch module: kpatch_cputime_hotpatch
kpatch unload SUCCESS
test load/unload count=1506105
loaded core module
loading patch module: kpatch-cputime-hotpatch.ko-15
disabling patch module: kpatch_cputime_hotpatch
unloading patch module: kpatch_cputime_hotpatch
kpatch unload SUCCESS
test load/unload count=1506106
loaded core module
loading patch module: kpatch-cputime-hotpatch.ko-15
disabling patch module: kpatch_cputime_hotpatch
unloading patch module: kpatch_cputime_hotpatch
kpatch unload SUCCESS
test load/unload count=1506107
```

丁的新逻辑后，utime+stime打印正常且与rtime保持了同步更新，从而验证了热补丁的正确性。

全网变更：最后再分批在现网环境机器上打热补丁，执行全网变更，问题得到根本解决，此处要感谢运维同学的全力协助。

总结

综上，我们详细介绍了进程cputime统计异常问题的完整分析和解决思路。该问题并非严重的宕机问题，但却可能会让用户对监控数据产生困惑，误认为可能机器负载太高需要加资源，问题的解决会避免产生不必要的开支。此外，该问题也会让研发、运维和技术支持的同学们使用top和ps命令时产生困惑。最终我们对问题的本质仔细分析并求证，用热补丁的方式妥善的解决了问题。

新浪有云：AI上华为云

每一分钟的阅读体验都流畅即时，全凭这朵云。



华为云 | 有技术 有未来 值得信赖

华为云 普惠AI 4000-955-988

AiCon

全球人工智能与机器学习技术大会

大会聚焦

- 机器学习应用和实践
- 计算机视觉
- NLP和语音技术
- 搜索推荐与算法
- 数据智能驱动业务
- AI+行业案例



8折报名中，立减720元

联系热线：18514549229

2018年12月20-23 / 北京·国际会议中心



► 大咖助阵



颜水成
360人工智能研究院
院长及首席科学家



华先胜
阿里巴巴达摩院机器智能技术实
验室副主任
城市大脑人工智能技术负责人



裴健
京东集团副总裁
大数据与智能供应链事业部总裁



马维英
今日头条副总裁
人工智能实验室负责人



崔宝秋
小米人工智能与云平台副总裁
首席架构师

► 分享嘉宾



薄列峰
京东金融
AI实验室首席科学家



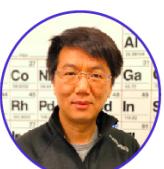
袁进辉 (老师木)
一流科技创始人



张俊林
新浪微博
AI Lab资深算法专家



王兴星
美团点评技术总监
外卖商业技术负责人



陈博兴
阿里巴巴达摩院
资深算法专家



鲍捷
文因互联CEO
人工智能领域知名专家

更多嘉宾陆续上线中 ...

8折报名中，立减720元

联系热线：18514549229

2018年12月20-23 / 北京·国际会议中心





华为云

11.11
普惠季

血拼风暴 一促即发

云服务器低至 1 元/月

点击抢购



有奖调研





架构师 月刊 2018年10月

本期主要内容：微软开源 Sketch2Code，草图秒变代码；Java 11 正式发布，新特性解读；秒杀 Redis 的 KVS 数据库上云了！伯克利重磅开源 Anna 1.0；架构师如何判断技术演进的方向？GitHub：我们为什么会弃用 jQuery？



Kubernetes指南

《Kubernetes 指南》开源电子书旨在整理平时在开发和使用 Kubernetes 时的参考指南和实践心得，更是为了形成一个系统化的参考指南以方便查阅。



2018，进击的大前端

全栈与大前端，前端工程师进阶该如何抉择？



《英雄联盟》 在线服务运维之道

拳头公司基础设施团队的工程师们分享了他们运维在线服务的发展历程，介绍了他们从手动部署到自动运维的演变过程。