大规模数据处理的演化历程（2003 - 2018）

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，最终一致性的结果。

* **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。

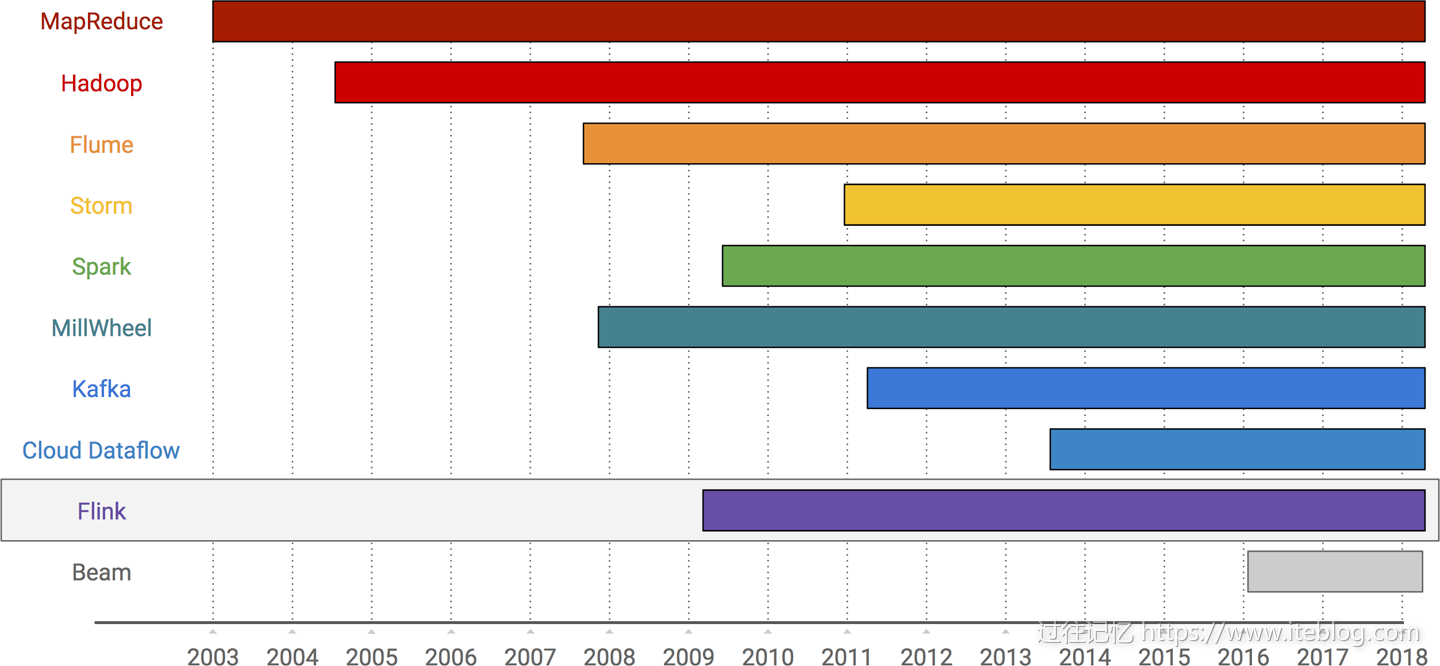
* **Beam**

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，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。

Flink



Flink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。

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为其提供了正确性所需的强一致性保证。

Flink的一致性机制，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。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获取了巨大的成功。有人曾经发布过一组针对Flink性能的具体测试数据，总结起来有以下结论：

* + 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%。
  + 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。

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。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。

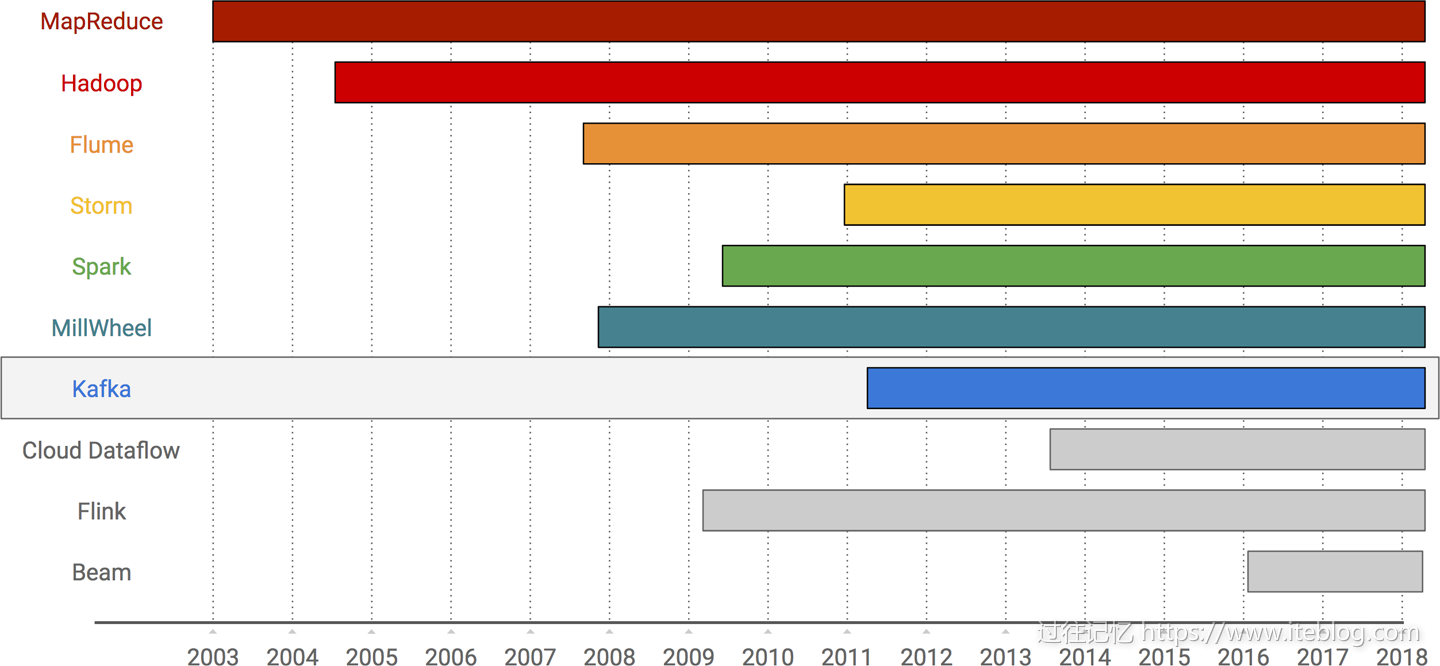
除了保存点之外，Flink社区还在不断创新，包括将第一个实用流式SQL API推向大规模分布式流处理引擎的领域。总之，Flink的迅速崛起成为流计算领军角色主要归功于三个特点：

* + 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）
  + 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。
  + 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。

Kafka

Kafka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Kafka，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。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。
* 定义了流和表之间的关系，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。

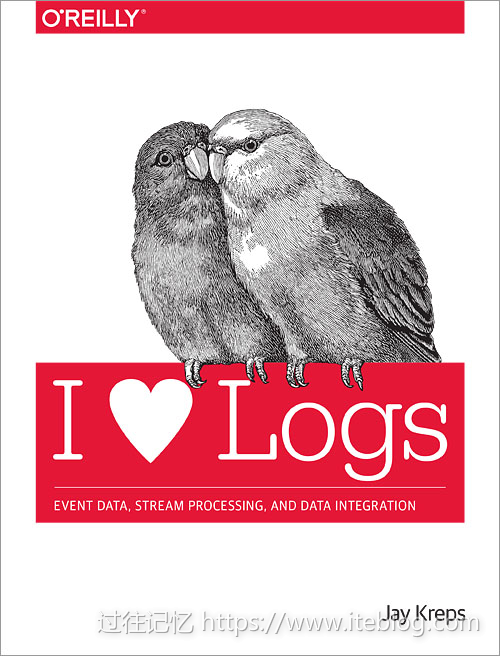
在这些特性中，有两个最为突出。

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。可重放这个特点是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。

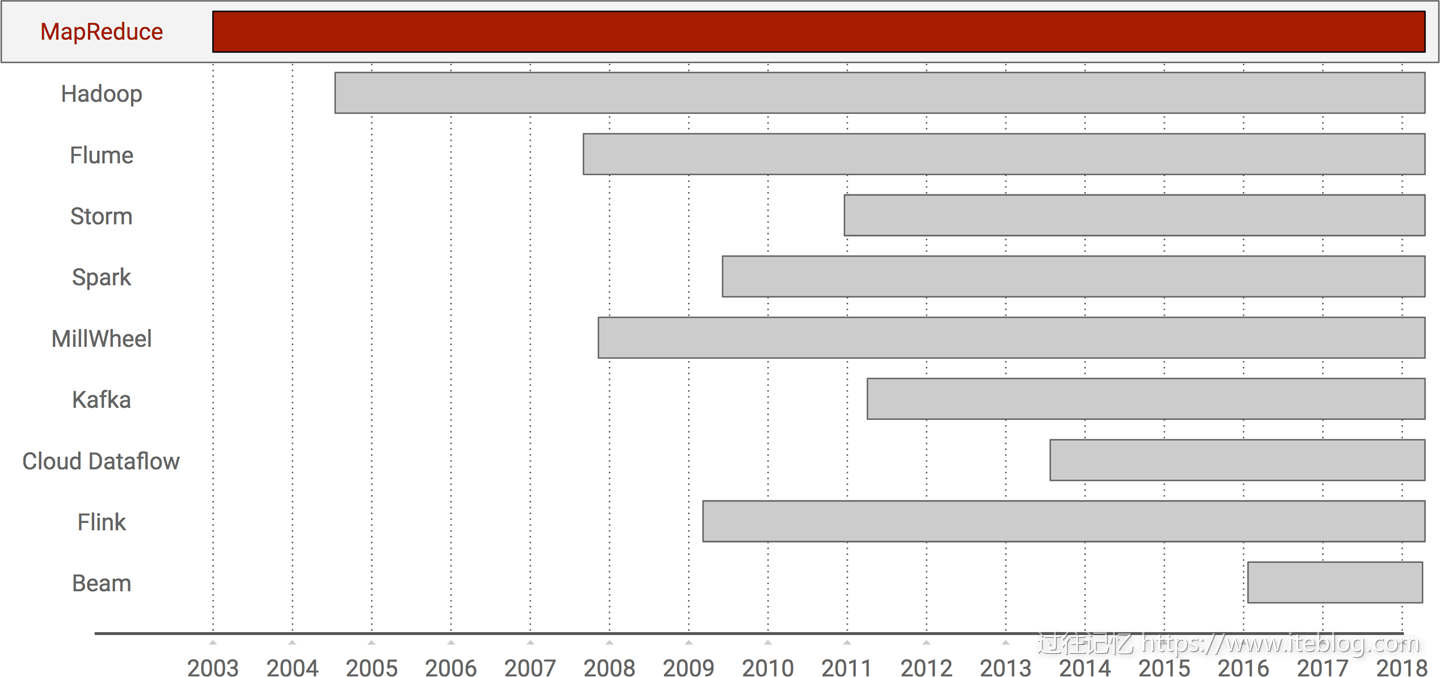
Kafka系统中第二个值得注意的重点是流和表理论的普及。可以说流和表构成了数据处理的基础，无论是MapReduce及其演化系统、SQL数据库系统，还是其他分支的数据处理系统。并不是所有的数据处理方法都直接基于流或者表来进行抽象，但从概念或者理论上说，表和流的理论就是这些系统的运作方式。作为这些系统的用户和开发人员，理解我们所有系统构建的核心基础概念意义重大。

如果您想了解更多关于Kafka及其理论核心，JackKreps的《I love Logs》是一个很好的学习资料。



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。

MapReduce



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的时候，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，那这个技术能够保你这辈子吃喝不愁。

* 可伸缩性很难

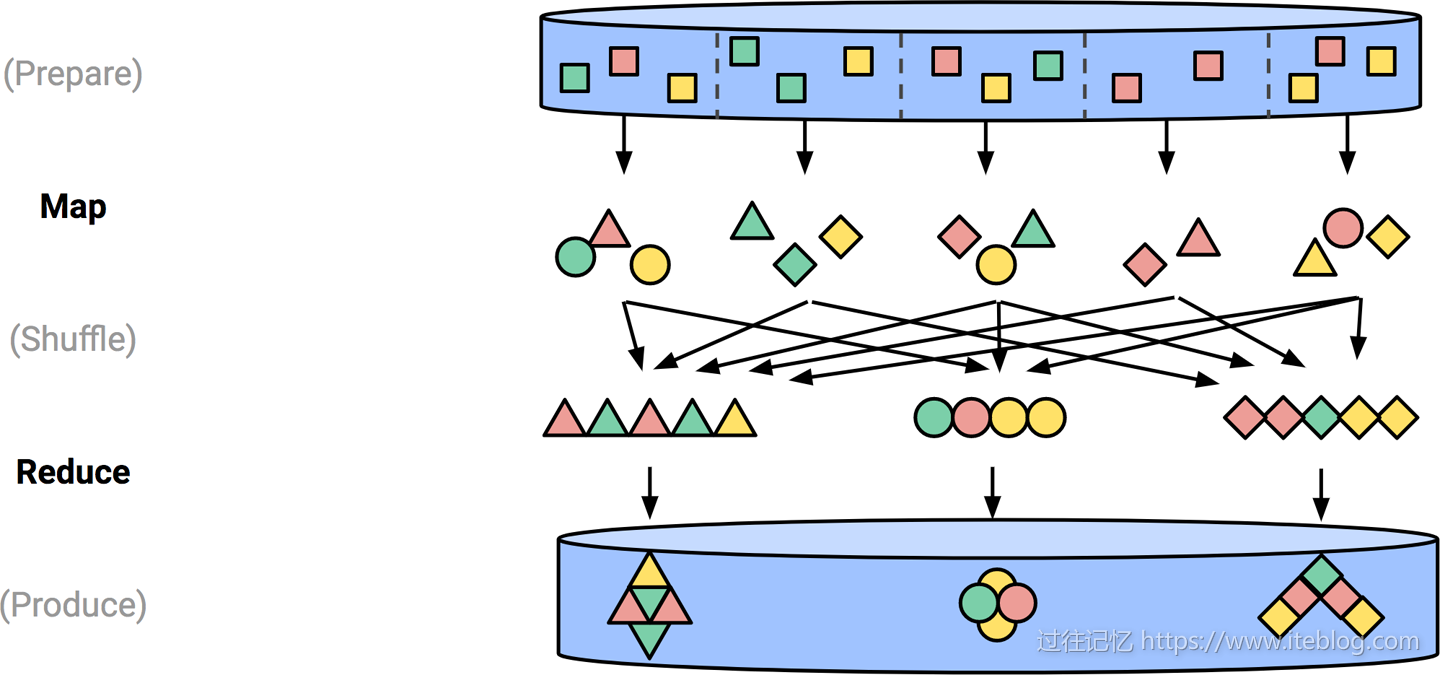
本来数据处理已经够难了，要从大规模数据集中挖掘出有价值的数据更加困难。

* 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以可容错的、准确的方式挖掘数据价值，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。



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。

MapReduce的处理过程被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。整体上看，Map和Reduce阶段之间差异其实不大；更高层次来看，它们都做了以下事情：

* 从表中读取数据，并转换为数据流
* 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。
* 将上述转换后的流根据某些规则分组，并写出到表中。

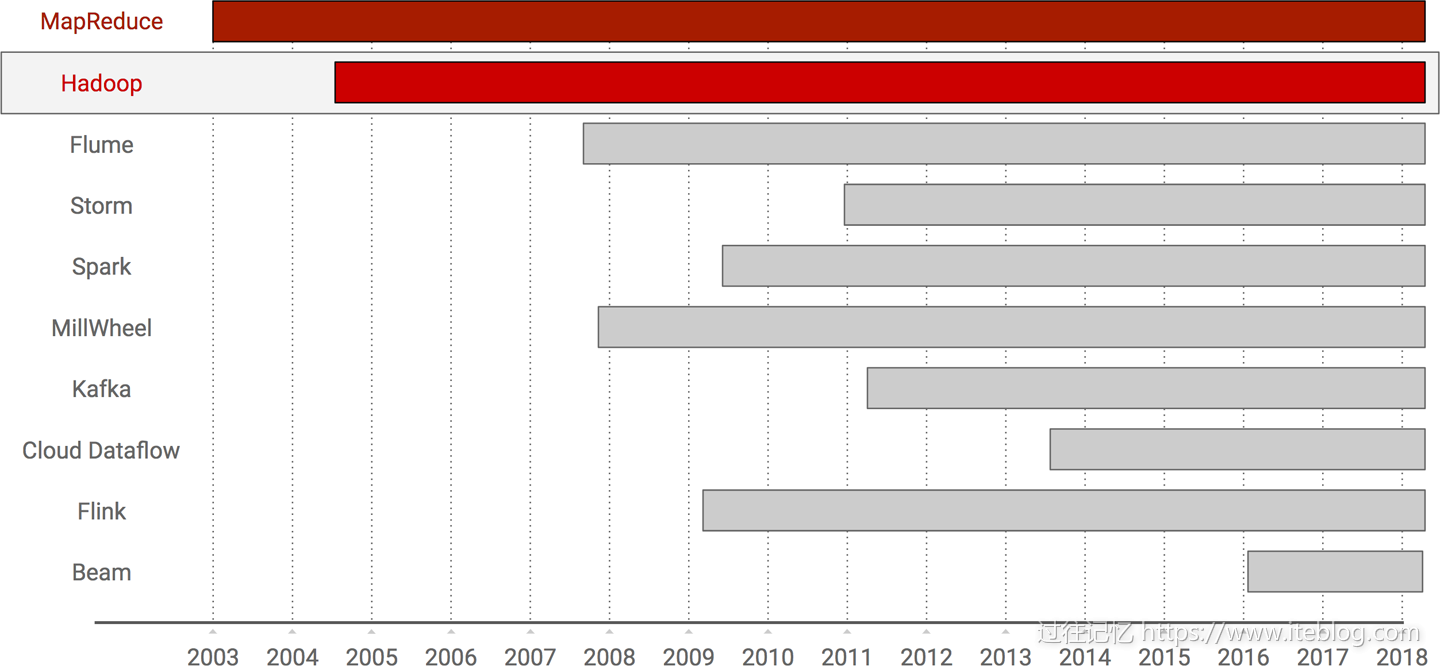
随后，Google内部将MapReduce投入生产使用并得到了非常广泛的业务应用，Google认为应该和公司外的同行分享我们的研究成果，最终将MapReduce论文发表于OSDI 2004。论文中，Google详细描述了MapReduce项目的历史，API的设计和实现，以及有关使用了MapReduce框架的许多不同生产案例的详细信息。当然，Google并没有提供任何实际的源代码，以至于最终Google以外的人都认为：“是的，这套系统确实牛啊！”，然后立刻回头去模仿MapReduce去构建他们的定制化系统。

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。

在这里要强调的是，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。

从流式处理系统来看，需要强调的是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



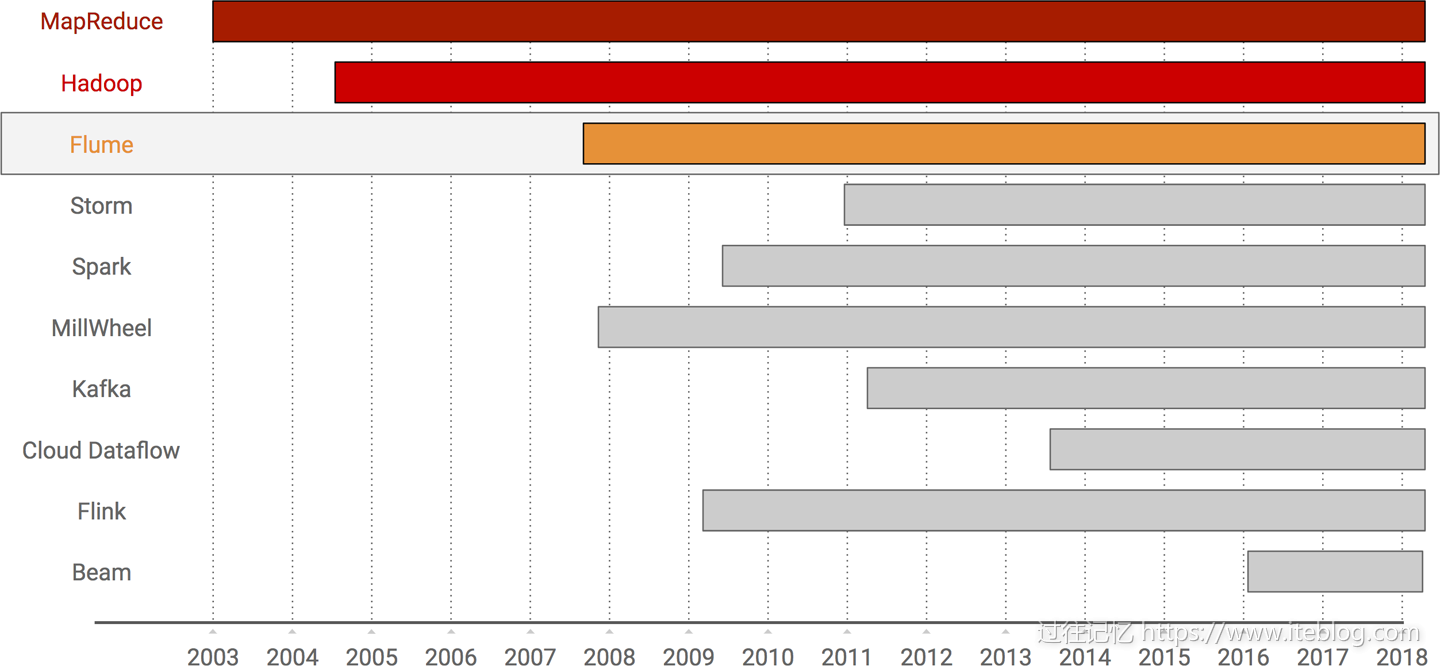
Hadoop于2005年问世，当时Doug Cutting和Mike Cafarella在构建Nutch webcrawler的分布式版本时借鉴了MapReduce论文中的想法。在这之前，他们已经实现了自己版本的Google分布式文件系统（最初称为Nutch分布式文件系统的NDFS，后来改名为HDFS或Hadoop分布式文件系统）。因此下一步，自然而然的，基于HDFS之上添加MapReduce计算层。他们称MapReduce这一层为Hadoop。

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。

到目前为止，围绕Hadoop的开源生态系统对整个行业产生了巨大的影响。通过创建一个开放的社区，工程师可以从早期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了流计算系统的发展。

Flume

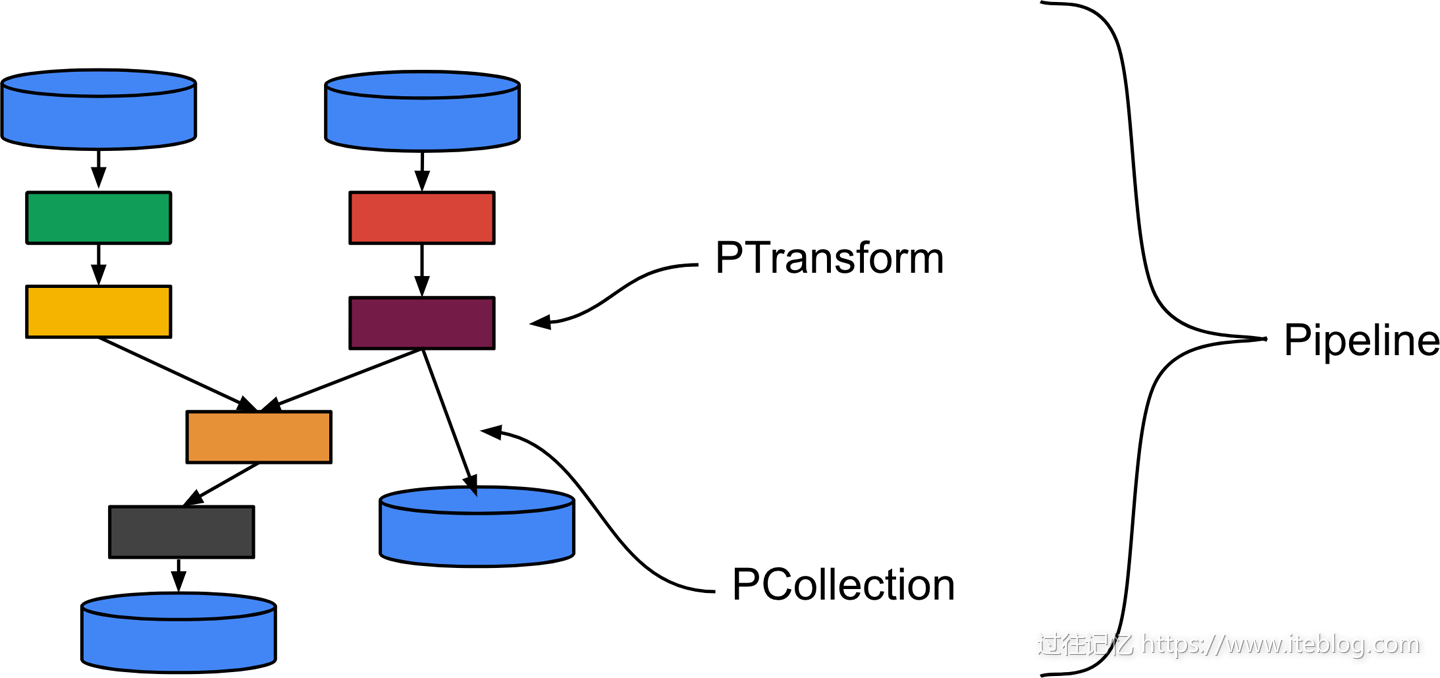
Flume是Google公司中MapReduce的官方继承者，有时也称为FlumeJava，这个名字起源于最初Flume的Java版本。需要注意的是，这里的Flume不要与Apache Flume混淆。



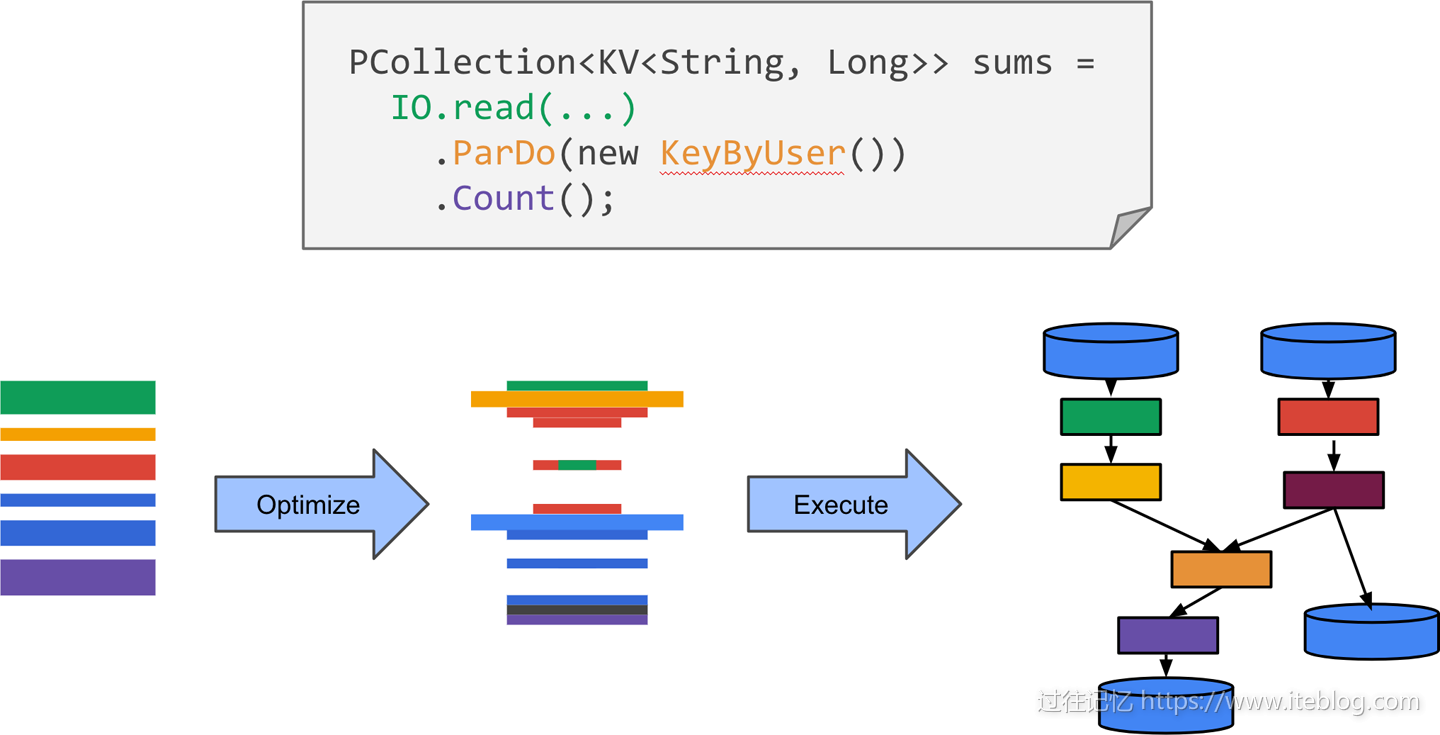
Flume项目是2007年在谷歌西雅图办事处成立时发起的。Flume最初打算是希望解决MapReduce的一些固有缺点，这些缺点即使在MapReduce最初大红大紫的阶段已经非常明显。其中许多缺点都与MapReduce完全限定的Map 🡪 Shuffle 🡪 Reduce编程模型相关；这个编程模型虽然简单，但它带来了一些缺点：

* 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。
* 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。

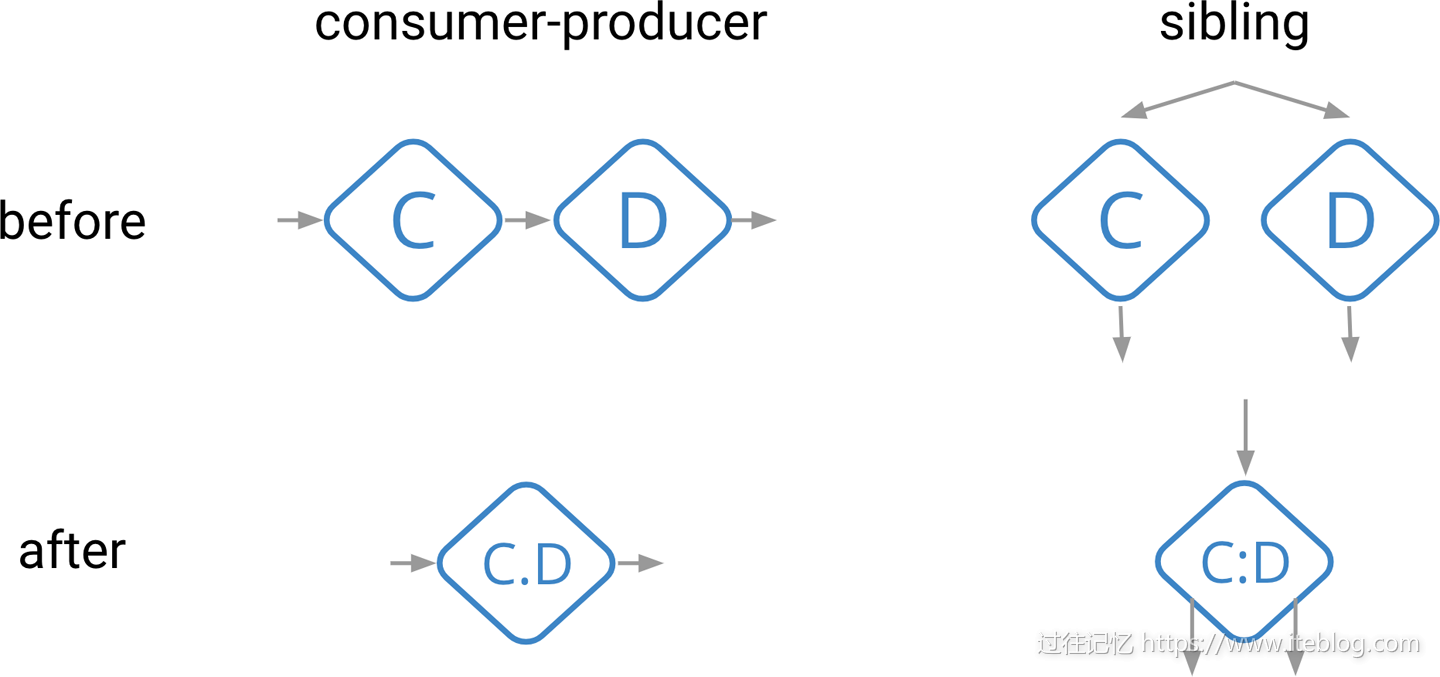
Flume通过提供可组合的高级API来描述数据处理流水线，从而解决了这些问题。这套设计理念同样也是Beam主要的抽象模型，即PCollection和PTransform概念，如图所示。



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：

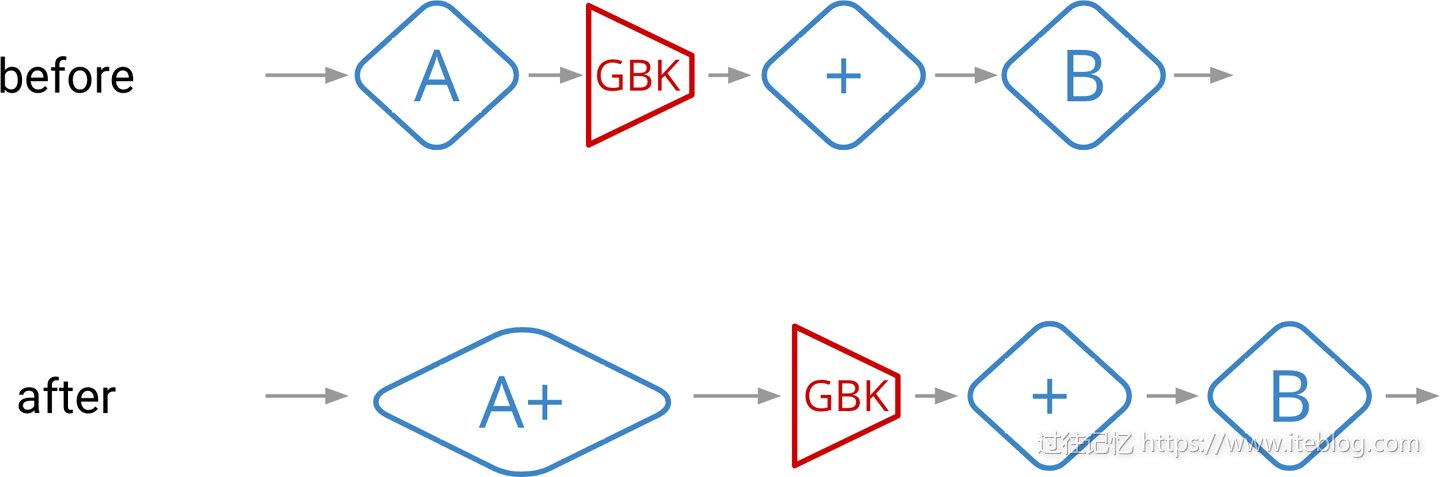


也许Flume在自动优化方面最重要的案例就是合并，其中两个逻辑上独立的阶段可以在同一个作业中顺序地（消费者 - 生产者融合）执行或者并行执行（兄弟融合），



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。combiner lifting是增量合并中的多级组合逻辑的编译器中的自动优化，以求和操作为例，求和的合并逻辑本来应该运算在分组（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。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。

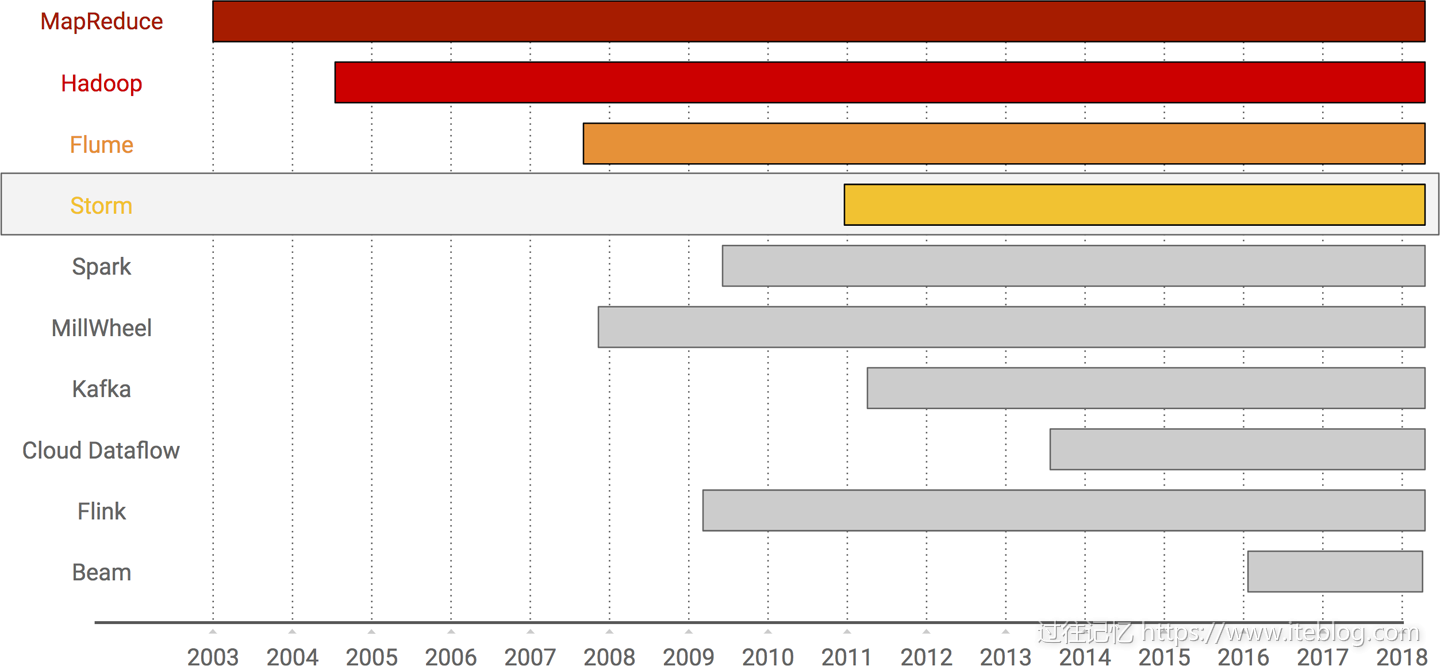
从那时起，Flume已经迁移到不再使用MapReduce作为执行引擎；相反，Flume底层基于一个名为Dax的内置自定义执行引擎。工作本身，不仅让Flume更加灵活选择执行计划而不必拘泥于Map 🡪 Shuffle 🡪 Reduce 模型，Dax还启用了新的优化，例如动态负载均衡。动态负载均衡可以让部分已经完成工作的Worker能够从另外一些繁忙的Worker手中分配一些额外的工作。在Job运行过程中，通过不断地动态调整负载分配可以将系统运行效率趋近最优，这种算法将比传统方法下有经验工程师手工设置的初始参数性能更好。Flume甚至为Worker池变化进行了适配，一个拖慢整个作业进度的Worker会将其任何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。在Google内部，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

Storm肯定不是业界使用最早的流式处理系统，但这是整个行业真正广泛采用的第一个流式处理系统。



Storm是Nathan Marz的心血结晶。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，如果有个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。

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。

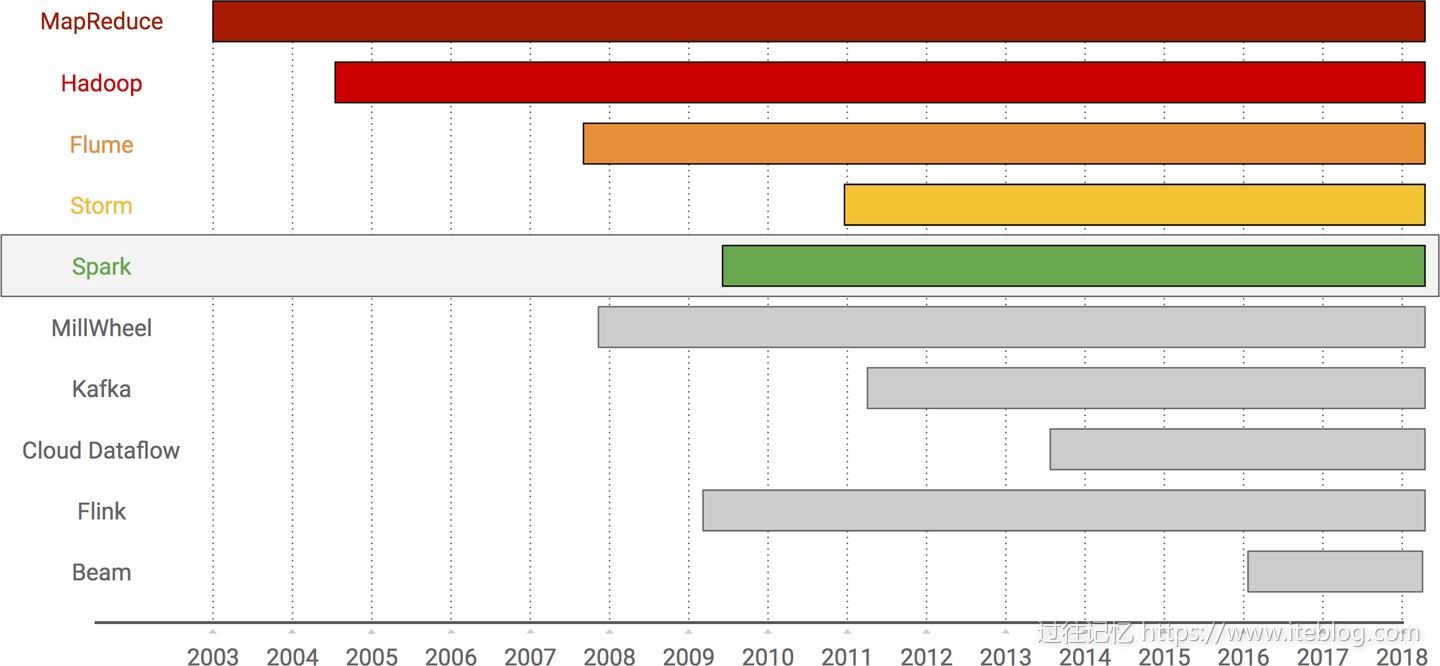
尽管Lambda带来了大量成本问题，Lambda架构当前还是非常受欢迎，仅仅是因为它满足了许多企业一个关键需求：系统提供低延迟但不准确的数据，后续通过批处理系统纠正之前数据，最终给出一致性的结果。从流处理系统演变的角度来看，Storm确实为普罗大众带来低延迟的流式实时数据处理能力。然而，它是以牺牲数据强一致性为代价的，这反过来又带来了Lambda架构的兴起，导致接下来多年基于两套系统架构之上的数据处理带来无尽的麻烦和成本。

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。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，只能拭目以待了。

Spark

这里我们仅仅关注Spark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AMP Lab。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，并且允许再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：

1. 输入总是可重放的
2. 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。

在Spark创建几年后，当时AMP Lab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，而且是可以独立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。但是，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。

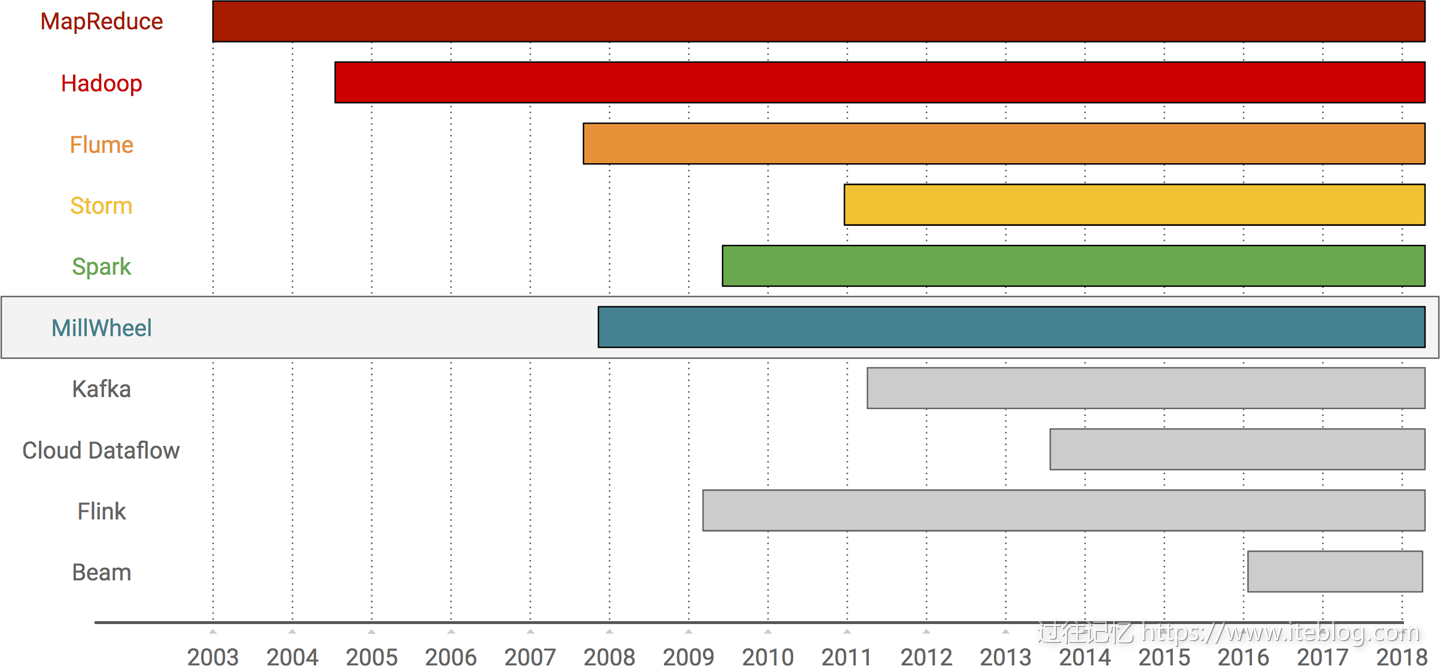
围绕Spark Streaming的另一个有趣的争议是“microbatch和true streaming”争论。由于Spark Streaming建立在批处理引擎的重复运行的基础之上，因此批评者声称Spark Streaming不是真正的流式引擎，因为整个系统的处理基于全局的数据切分规则。这个或多或少是实情。尽管流处理引擎几乎总是为了吞吐量而使用某种批处理或者类似的加大吞吐的系统策略，但它们可以灵活地在更精细的级别上进行处理，一直可以细化到某个key。但基于微批处理模型的系统在基于全局切分方式处理数据包，这意味着同时具备低延迟和高吞吐是不可能的。确实我们看到许多基准测试表明这说法或多或少有点正确。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发是非常到位的，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，这只是一个很小的问题。

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。

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。

MillWheel

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



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，两者效果都会显著下降。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，将系统提供的功能限制在称为“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，一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。

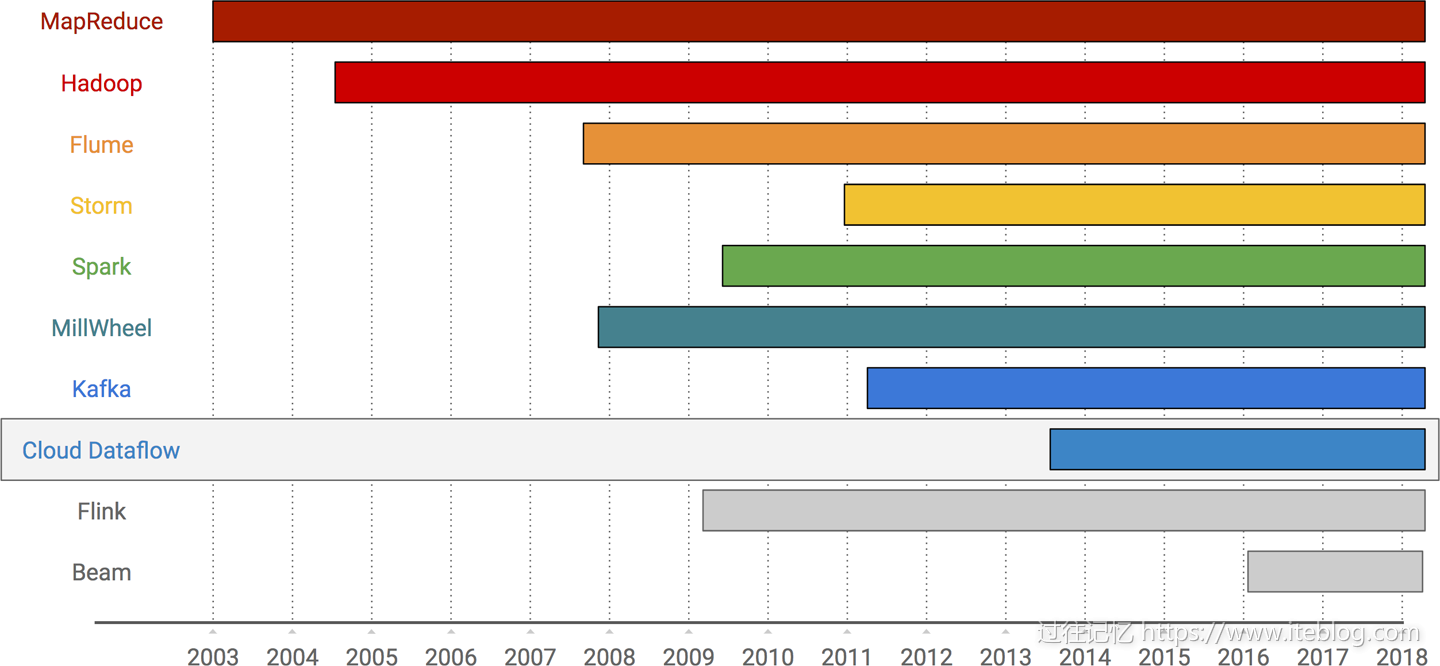
之后Watermark被设计出来用以解决数据乱序的问题。基本思想是跟踪系统输入数据的当前进度，对于每个给定的数据源，构建一个数据输入进度用来表征输入数据的完整性。对于一些简单的数据源，例如一个带分区的Kafka Topic，每个Topic下属的分区被写入的是业务时间持续递增的数（例如通过Web前端实时记录的日志事件），这种情况下我们可以计算产生一个非常完美的Watermark。但对于一些非常复杂的数据输入，例如动态的输入日志集，一个启发式算法可能是我们能够设计出来最能解决业务问题的Watermark生成算法了。但无论哪种方式，Watermark都是解决输入事件完整性最佳方式。之前我们尝试使用处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

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。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。

DataFlow

Cloud Dataflow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。



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，但在此主要讨论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。主要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。