# 从Zipkin到Jaeger，Uber的分布式追踪之道-搜狐

http://mt.sohu.com/20170310/n482883440.shtml

作者｜Yuri Shauro

　　编辑｜大愚若智

对于希望监视复杂的微服务架构系统的组织，分布式追踪正在快速成为一种不可或缺的工具。Uber工程团队的开源分布式追踪系统Jaeger自2016年起，在公司内部实现了大范围的运用，已经集成于数百个微服务中，目前每秒钟已经可以记录数千条追踪数据。新年伊始，我们想向大家介绍一下这一切是如何实现的，从我们最开始使用现成的解决方案，如Zipkin，到我们从拉取转换为推送架构的原因，以及2017年有关分布式追踪的发展计划。

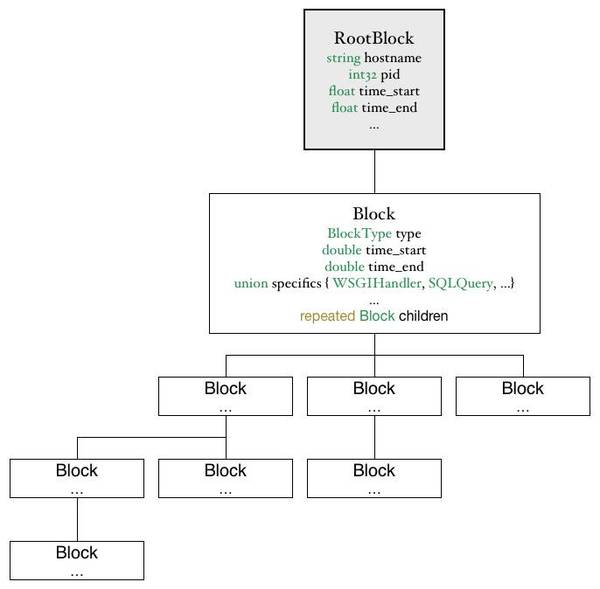
　　从整体式到微服务架构

随着Uber的业务飞速增长，软件架构的复杂度也与日俱增。大概一年多前，2015年秋季，我们有大约500个微服务，2017年初这一数量已增长至超过2000个。这样的增幅部分是由于业务该功能的增加，例如面向用户的UberEATS和UberRUSH等功能，以及类似欺诈检测、数据挖掘、地图处理等内部功能的增加。此外随着我们从大规模整体式应用程序向着分布式微服务架构迁移，也造成了复杂度的增加。

迁移到微服务生态总是会遇到独特的挑战。例如丧失对系统的能见度，服务之间开始产生复杂的交互等。Uber工程团队很清楚，我们的技术会对大家的生活产生直接影响，系统的可靠性至关重要，但这一切都离不开“可观测性”这一前提。传统的监视工具，例如度量值和分布式日志依然发挥着自己的作用，但这类工具往往无法提供跨越不同服务的能见度。分布式追踪应运而生。

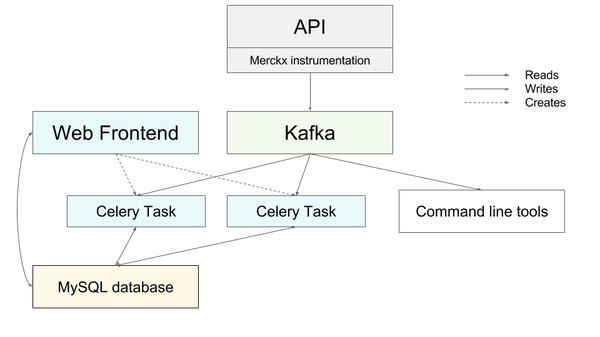
　　Uber最初的追踪系统

Uber最初广泛使用的追踪系统叫做Merckx，这一名称源自全球速度最快的自行车骑行选手。Merckx很快就帮助我们了解了有关Uber基于Python的整体式后端的很多问题。我们可以查询诸如“查找已登录用户的请求，并且请求的处理时间超过2秒钟，并且使用了某一数据库来处理，并且事务维持打开状态的时间超过500ms”这样的问题。所有待查询的数据被组织成树状块，每个块代表某一操作或某个远程调用，这种组织方式类似于OpenTracing API中“Span”这个概念。用户可以在Kafka中使用命令行工具针对数据流执行即席查询，也可以使用Web界面查看预定义的摘要，这些信息均从API端点的高级别行为和Celery任务中摘要汇总而来。



Merckx使用了一种类似于树状块的调用图，每个块代表应用程序中的一个操作，例如数据库调用、RPC，甚至库函数，例如解析JSON。

Merckx的编排调度可自动应用于使用Python编写的一系列基础架构库，包括HTTP客户端和服务器、SQL查询、Redis调用，甚至JSON的序列化。这些编排调度可记录有关每次操作的某些性能度量值和元数据，例如HTTP调用的URL，或数据库调用的SQL查询。此外还能记录其他信息，例如数据库事务维持打开状态的时长，访问了哪些数据库Shard和副本。



Merckx架构使用了拉取模式，可从Kafka的指令数据中拉取数据流。

Merckx最大的不足在于其设计主要面向Uber使用整体式API的年代。Merckx缺乏分布式上下文传播的概念，虽然可以记录SQL查询、Redis调用，甚至对其他服务的调用，但无法进一步深入。Merckx还有另一个有趣的局限：因为Merckx数据存储在一个全局线程本地存储中，诸如数据库事务追踪等大量高级功能只能在uWSGI下使用。随着Uber开始使用Tornado（一种适用于Python服务的异步应用程序框架），线程本地存储无法体现Tornado的IOLoop中同一个线程内运行的大部分并发请求。我们开始意识到不借助全局变量或全局状态，转为通过某种方式保存请求状态，并进行恰当的传播的重要性。

　　随后，使用TChannel进行追踪

2015年初，我们开始开发TChannel，这是一种适用于RPC的网络多路复用和框架协议。该协议的设计目标之一是将类似于Dapper的分布式追踪能力融入协议中，并为其提供最优秀的支持。为了实现这一目标，TChannel协议规范将追踪字段直接定义到了二进制格式中。

spanid:8 parentid:8 traceid:8 traceflags:1

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 描述 |
| spanid | int64 | 用于识别当前span |
| parentid | int64 | 前一个span |
| traceid | int64 | 负责分配的原始操作方 |
| traceflags | uint8 | 位标志字段 |

追踪字段作为二进制格式的一部分已包含在TChannel协议规范中。

除了协议规范，我们还发布了多个开源客户端库，用于以不同语言实现该协议。这些库的设计原则之一是让应用程序需要用到的请求上下文这一概念能够从服务器端点贯穿至下游的调用站点。例如在tchannel-go中，让出站调用使用JSON进行编码的签名需要通过第一个参数提供上下文：

func (c \*Client) Call(ctx Context, method string, arg, resp interface{}) error {..}

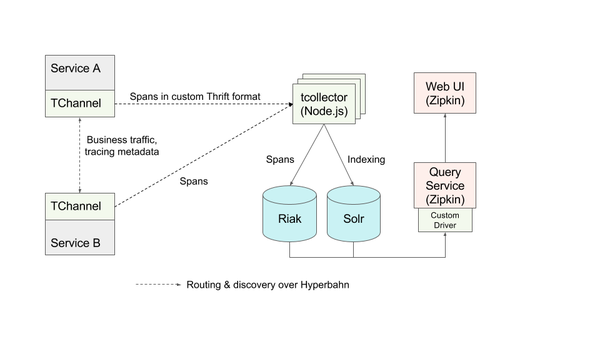
Tchannel库使得应用程序开发者在编写自己的代码时始终将分布式上下文传播这一概念铭记于心。

通过将所传输内容以及内存中的上下文对象之间的追踪上下文进行安排，并围绕服务处理程序和出站调用创建追踪Span，客户端库内建了对分布式追踪的支持。从内部来看，这些Span在格式上与Zipkin追踪系统几乎完全相同，也使用了Zipkin所定义的注释，例如“cs”（Client Send）和“cr”（Client Receive）。Tchannel使用追踪报告程序（Reporter）接口将收集到的进程外追踪Span发送至追踪系统的后端。该技术自带的库默认包含一个使用Tchannel本身和Hyperbahn实现的报告程序以及发现和路由层，借此将Thrift格式的Span发送至收集器群集。

Tchannel客户端库已经比较近似于我们所需要的分布式追踪系统，该客户端库提供了下列构建块：

* 追踪上下文的进程间传播以及带内请求
* 通过编排API记录追踪Span
* 追踪上下文的进程内传播
* 将进程外追踪数据报告至追踪后端所需的格式和机制

该系统唯独缺少了追踪后端本身。追踪上下文的传输格式和报表程序使用的默认Thrift格式在设计上都可以非常简单直接地将Tchannel与Zipkin后端集成，然而当时只能通过Scribe将Span发送至Zipkin，而Zipkin只支持使用Cassandra格式的数据存储。此外当时我们对这些技术没什么经验，因此我们开发了一套后端原型系统，并结合Zipkin UI的一些自定义组件构建了一个完整的追踪系统。



后端原型系统架构：Tchannel生成的追踪记录推送给自定义收集器、自定义存储，以及开源的Zipkin UI。

分布式追踪系统在谷歌和Twitter等主要技术公司获得的成功意味着这些公司中广泛使用的RPC框架、Stubby和Finagle是行之有效的。

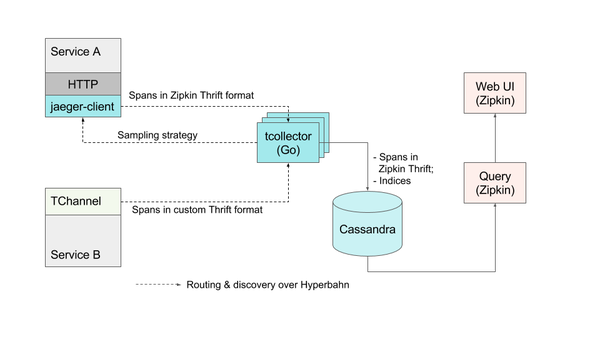
同理，Tchannel自带的追踪能力也是一个重大的飞跃。我们部署的后端原型系统已经开始从数十种服务中收集追踪信息。随后我们使用Tchannel构建了更多服务，但在生产环境中全面推广和广泛使用依然有些困难。该后端原型以及所使用的Riak/Solr存储系统无法妥善缩放以适应Uber的流量，同时很多查询功能依然无法与Zipkin UI实现足够好的互操作。尽管新构建的服务大量使用了Tchannel，Uber依然有大量服务尚未在RPC过程中使用Tchannel，实际上承担核心业务的大部分服务都没有使用Tchannel。这些服务主要是通过四大编程语言（Node.js、Python、Go和Java）实现的，在进程间通信方面使用了多种不同的框架。这种异构的技术环境使得Uber在分布式追踪系统的构建方面会面临比谷歌和Twitter更严峻的挑战。

　　在纽约市构建的Jaeger

Uber纽约工程组织始建于2015年上半年，主要包含两个团队：基础架构端的Observability以及产品（包括UberEATS和UberRUSH）端的Uber Everything。考虑到分布式追踪实际上是一种形式的生产环境监视，因此更适合交由Observability团队负责。

我们组建了分布式追踪团队，该团队由两个工程师组成，目标也有两个：将现有的原型系统转换为一种可以全局运用的生产系统，让分布式追踪功能可以适用并适应Uber的微服务。我们还需要为这个项目起一个开发代号。为新事物命名实际上是计算机科学界两大老大难问题之一，我们花了几周时间集思广益，考虑了追踪、探测、捕获等主题，最终决定命名为Jaeger（?yā-g?r），在德语中这个词代表猎手或者狩猎过程中的帮手。

纽约团队在Cassandra群集方面已经具备运维经验，该数据库直接为Zipkin后端提供着支持，因此我们决定弃用基于Riak/Solr的原型。为了接受TChannel流量并将数据以兼容Zipkin的二进制格式存储在Cassandra中，我们用Go语言重新实现了收集器。这样我们就可以无需改动，直接使用Zipkin的Web和查询服务，并通过自定义标签获得了原本不具备的追踪记录搜索功能。我们还为每个收集器构建了一套可动态配置的倍增系数（Multiplication factor），借此将入站流量倍增n次，这主要是为了通过生产数据对后端系统进行压力测试。



Jaeger的早期架构依然依赖Zipkin UI和Zipkin存储格式。

第二个业务需求希望让追踪功能可以适用于未使用TChannel进行RPC的所有现有服务。随后几个月我们使用Go、Java、Python和Node.js构建了客户端库，借此未包括HTTP服务在内各类服务的编排提供支持。尽管Zipkin后端非常著名并且流行，但依然缺乏足够完善的编排能力，尤其是在Java/Scala生态系统之外的编排能力。我们考虑过各种开源的编排库，但这些库是由不同的人维护的，无法确保互操作性，并且通常还使用了完全不同的API，大部分还需要使用Scribe或Kafka作为报表Span的传输机制。因此我们最终决定自行编写库，这样可以通过集成测试更好地保障互操作性，可以支持我们需要的传输机制，更重要的是，可以用不同的语言提供一致的编排API。我们的所有客户端库从一开始都可支持OpenTracing API。

在第一版客户端库中，我们还增加了另一个新颖的功能：可以从追踪后端轮询采样策略。当某个服务收到不包含追踪元数据的请求后，所编排的追踪功能通常会为该请求启动一个新的追踪，并生成新的随机追踪ID。然而大部分生产追踪系统，尤其是与Uber的缩放能力有关的系统无法对每个追踪进行“描绘”（Profile）或将其记录在自己的存储中。这样做会在服务与后端系统之间产生难以招架的大流量，甚至会比服务所处理的实际业务流量大出好几个数量级。我们改为让大部分追踪系统只对小比例的追踪进行采样，并只对采样的追踪进行“描绘”和记录。用于进行采样决策的算法被我们称之为“采样策略”。采样策略的例子包括：

* 采样一切。主要用于测试用途，但生产环境中使用会造成难以承受的开销！
* 基于概率的采样，按照固定概率对特定追踪进行随机采样。
* 限速采样，每个时间单位对X个追踪进行采样。例如可能会使用漏桶（Leaky bucket）算法的变体。

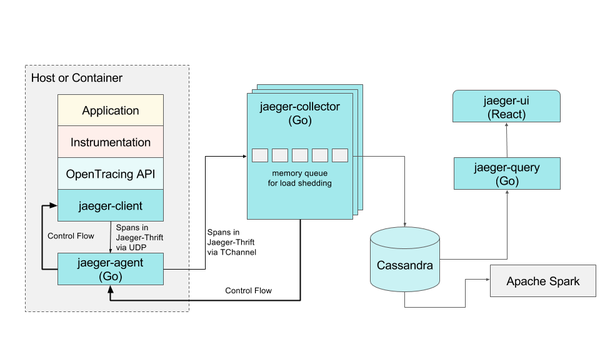
大部分兼容Zipkin的现有编排库可支持基于概率的采样，但需要在初始化过程中对采样速率进行配置。以我们的规模，这种方式会造成一些严重的问题：

* 每个服务对不同采样速率对追踪后端系统整体流量的影响知之甚少。例如，就算服务本身使用了适度的每秒查询数（QPS）速率，也可能调用扇出（Fanout）因素非常高的其他下游服务，或由于密集编排导致产生大量追踪Span。
* 对于Uber来说，每天不同时段的业务流量有着明显规律，峰值时期乘客更多。固定不变的采样概率对非峰值时刻可能显得过低，但对峰值时刻可能显得过高。

Jaeger客户端库的轮询功能按照设计可以解决这些问题。通过将有关最恰当采样策略的决策转交给追踪后端系统，服务的开发者不再需要猜测最适合的采样速率。而后端可以按照流量模式的变化动态地调整采样速率。下方的示意图显示了从收集器到客户端库的反馈环路。

第一版客户端库依然使用TChannel发送进程外追踪Span，会将其直接提交给收集器，因此这些库需要依赖Hyperbahn进行发现和路由。对于希望在自己的服务中运用追踪能力的工程师，这种依赖性造成了不必要的摩擦，这样的摩擦存在于基础架构层面，以及需要在服务中额外包含的库等方面，进而可能导致依赖性地域。

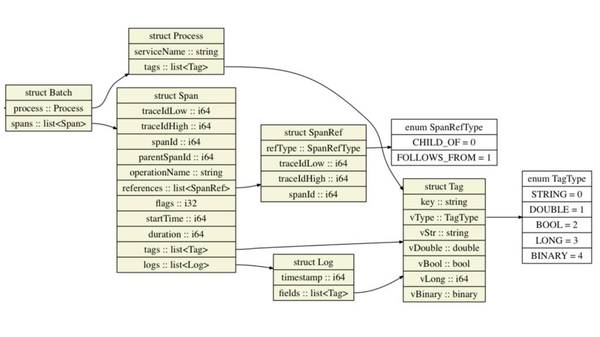
为了解决这种问题，我们实现了一种jaeger-agent边车（Sidecar）进程，并将其作为基础架构组件，与负责收集度量值的代理一起部署到所有宿主机上。所有与路由和发现有关的依赖项都封装在这个jaeger-agent中，此外我们还重新设计了客户端库，可将追踪Span报告给本地UDP端口，并能轮询本地回环接口上的代理获取采样策略。新的客户端只需要最基本的网络库。架构上的这种变化向着我们先追踪后采样的愿景迈出了一大步，我们可以在代理的内存中对追踪记录进行缓冲。



目前的Jaeger架构：后端组件使用Go语言实现，客户端库使用了四种支持OpenTracing标准的语言，一个基于React的Web前端，以及一个基于Apache Spark的后处理和聚合数据管道。

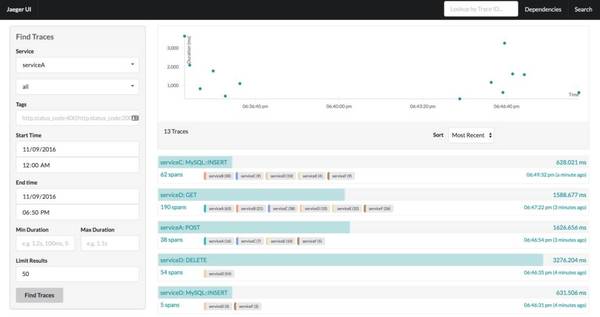
　　统包式分布式追踪

Zipkin UI是我们在Jaeger中使用的最后一个第三方软件。由于要将Span以Zipkin Thrift格式存储在Cassandra中并与UI兼容，这对我们的后端和数据模型产生了一定的限制。尤其是Zipkin模型不支持OpenTracing标准和我们的客户端库中两个非常重要的功能：键-值日志API，以及用更为通用的有向无环图（Directed acyclic graph）而非Span树所代表的追踪。因此我们毅然决定彻底革新后端所用的数据模型，并编写新的UI。如下图所示，新的数据模型可原生支持键-值日志和Span的引用，此外还对发送到进程外的数据量进行了优化，避免进程标签在每个Span上重复：



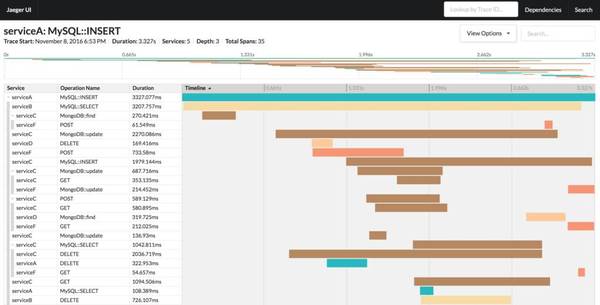
Jaeger数据模型可原生支持键-值日志和Span引用。

目前我们正在将后端管道全面升级到新的数据模型，以及全新的，更为优化的Cassandra架构。为了充分利用新的数据模型，我们还用Go语言实现了一个全新的Jaeger查询服务，并用React实现了一套全新的Web UI。最初版本的UI主要重现了Zipkin UI的原有功能，但在设计上更易于通过扩展提供新的功能和组件，并能作为React组件嵌入到其他UI。例如，用户可以选择用多种不同视图对追踪结果进行可视化，例如追踪时段内的直方图，或服务在追踪过程中的累积时间：



Jaeger UI显示的追踪信息搜索结果。右上角显示的时刻和持续时间散点图用可视化方式呈现了结果，并提供了向下挖掘能力。

另一个例子，可以根据不同用例查看同一条追踪记录。除了使用默认的时序渲染方式，还可以通过其他视图渲染为有向无环图或关键路径图：



Jaeger UI显示了一条追踪记录的详情。界面顶部是一条追踪记录的迷你地图示意图，借此可在更大规模的追踪记录中进行更轻松的导航。

通过将架构中剩余的Zipkin组件替代为Jaeger自己的组件，我们将Jaeger彻底变为一种统包式的端到端分布式追踪系统。

我们认为编排库是Jaeger固有的一部分，这样可以确保与Jaeger后端的兼容性，以及通过持续集成测试保障相互之间的互操作性。（Zipkin生态系统做不到这些。）尤其是跨越所有可支持语言（目前支持Go、Java、Python和Node.js）和可支持的传输方式（目前支持HTTP和TChannel）实现的互操作性会在每个Pull请求中测试，并用到了Uber工程部门RPC团队所开发的Crossdock框架。Jaeger客户端集成测试的详细信息请参阅jaeger-client-go crossdock代码库。目前所有Jaeger客户端库都已开源：

* Go
* Java
* Node.js
* Python

我们正在将后端和UI代码迁移至GitHub，并计划尽快将Jaeger的源代码全部公开。如果你对这个过程感兴趣，可以关注主代码库。我们欢迎大家为此做贡献，也很乐于看到更多人尝试使用Jaeger。虽然我们对目前的进展很满意，但Uber的分布式追踪工作还有很长的路要走。

Yuri Shkuro是Uber纽约工程部办公室的全职软件工程师，目前正全力从事Jaeger和其他Uber工程团队开源项目。

原文地址：https://eng.uber.com/distributed-tracing/

　　今日荐文

　　点击下方图片即可阅读



　　InfoQ观察：Java EE的未来

推荐一个对技术人员的成长很有帮助的线下会议，将于4月16～18日举行的QCon全球软件开发大会（北京站），目前已经邀请来自Google、Facebook、LinkedIn、Airbnb、百度、阿里巴巴、腾讯等公司的100多位一线技术专家，是难得的线下交流学习的机会。具体详戳 「 阅读原文 」惊喜不停！