美团点评技术团队

分布式会话跟踪系统架构设计与实践

志桐 ·2016-10-14 18:13

本文整理自美团点评技术沙龙第08期：大规模集群的服务治理设计与实践。

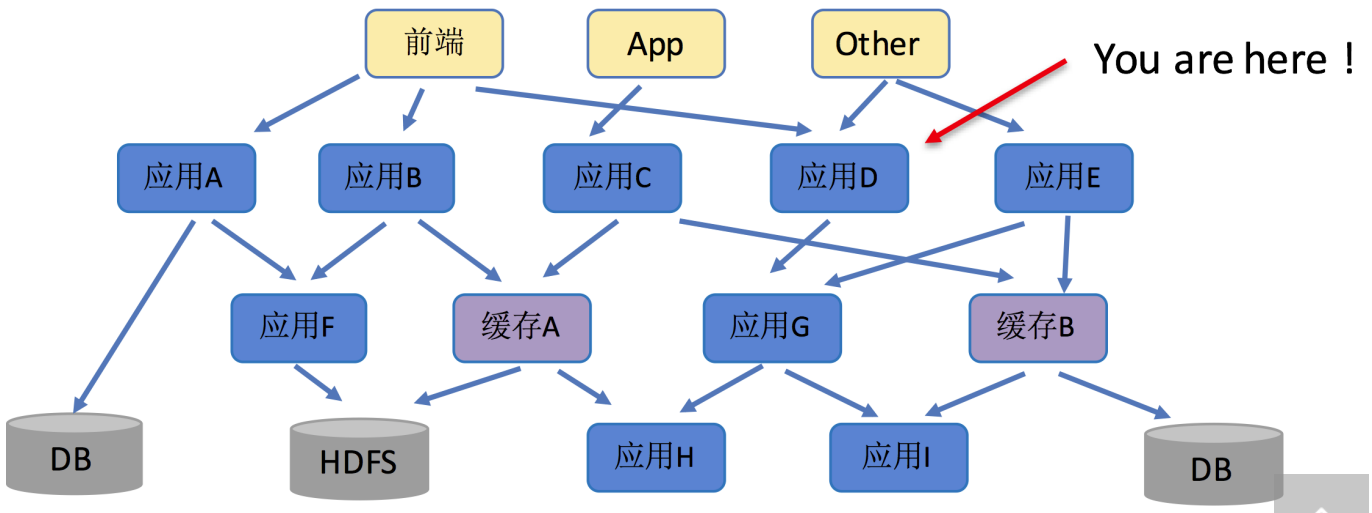
美团点评技术沙龙由美团点评技术团队主办，每月一期。每期沙龙邀请美团点评及其它互联网公司的技术专家分享来自一线的实践经验，覆盖各主要技术领域。

目前沙龙会分别在北京、上海和厦门等地举行，要参加下一次最新沙龙活动？赶快关注微信公众号“美团点评技术团队”。

这期沙龙主要内容有：分布式服务通信框架及服务治理系统、分布式监控系统实践、分布式会话跟踪系统架构设计与实践，特邀美恰CTO讲解时下热门话题“微服务”。其中既包括关键系统设计、在美团点评内部的实践经验，也包括一些项目在业界开源的运营实践。

前言

随着美团点评的业务发展，公司的分布式系统变得越来越复杂，我们亟需一个工具能够梳理内部服务之间的关系，感知上下游服务的形态。比如一次请求的流量从哪个服务而来、最终落到了哪个服务中去？服务之间是RPC调用，还是HTTP调用？一次分布式请求中的瓶颈节点是哪一个，等等。



简介

MTrace，美团点评内部的分布式会话跟踪系统，其核心理念就是调用链：通过一个全局的ID将分布在各个服务节点上的同一次请求串联起来，还原原有的调用关系、追踪系统问题、分析调用数据、统计系统指标。这套系统借鉴了2010年Google发表的一篇论文《dapper》，并参考了Twitter的Zipkin以及阿里的Eagle Eye的实现。
那么我们先来看一下什么是调用链，调用链其实就是将一次分布式请求还原成调用链路。显式的在后端查看一次分布式请求的调用情况，比如各个节点上的耗时、请求具体打到了哪台机器上、每个服务节点的请求状态，等等。它能反映出一次请求中经历了多少个服务以及服务层级等信息（比如你的系统A调用B，B调用C，那么这次请求的层级就是3），如果你发现有些请求层级大于10，那这个服务很有可能需要优化了。

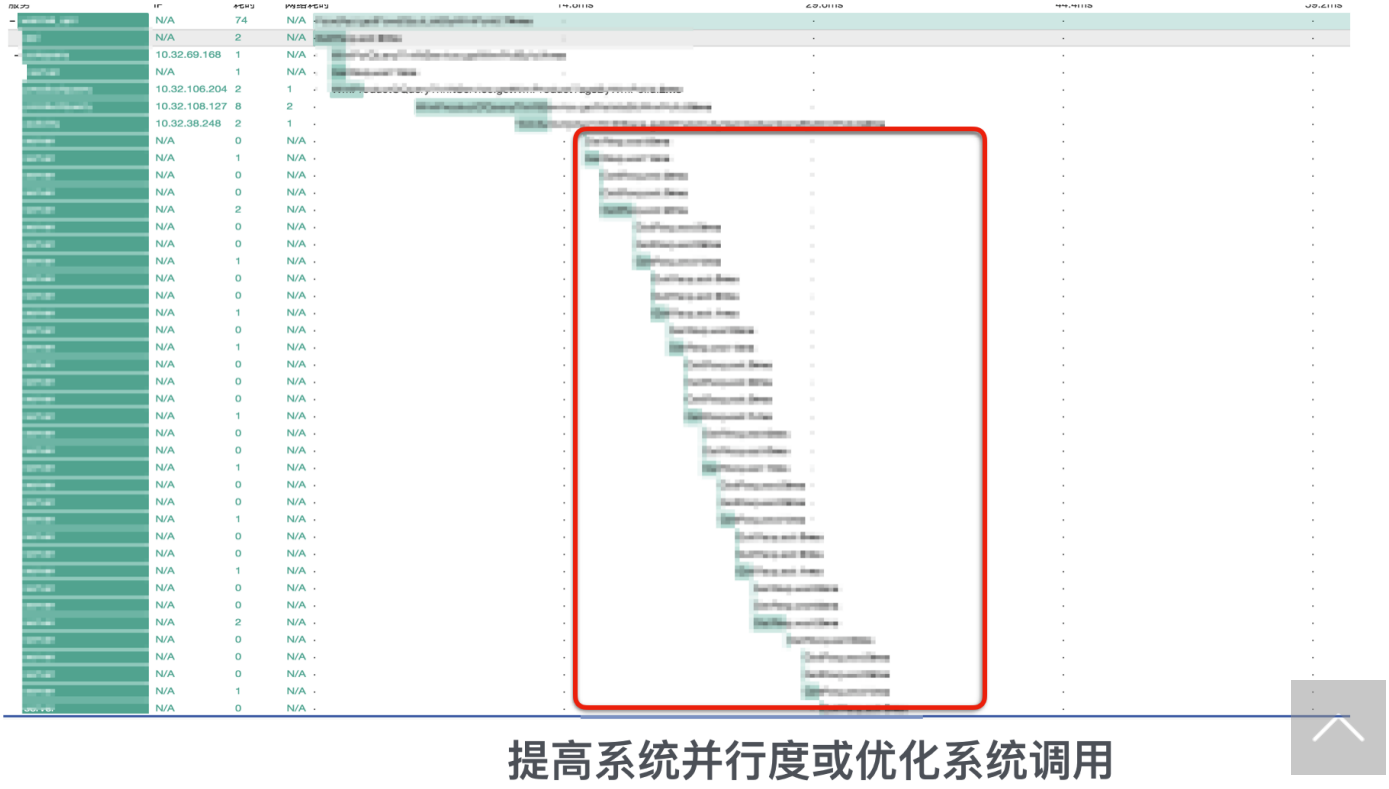
网络优化

如上图所示，红框内显示了一次分布式请求经过各个服务节点的具体IP，通过该IP就可以查询一次分布式请求是否有跨机房调用等信息，优化调用链路的网络结构。

瓶颈查询

再比如上图，红框部分显示的是系统调用的瓶颈节点，由于该节点的耗时，导致了整个系统调用的耗时延长，因此该节点需要进行优化，进而优化整个系统的效率。这种问题通过调用链路能很快发现下游服务的瓶颈节点；但是假如没有这样的系统，我们会怎样做呢？首先我会发现下游服务超时造成了我的服务超时，这时我会去找这个下游服务的负责人，然后该负责人发现也不是他自己服务的问题，而是他们调用了其他人的接口造成的问题，紧接着他又去找下游的服务负责人。我们都知道跨部门之间的沟通成本很高的，这么找下去会花费大量的不必要时间，而有了MTrace之后，你只需要点开链路就能发现超时问题的瓶颈所在。

优化链路



我们再来看下上面这张图，红框部分都是同一个接口的调用，一次请求调用相同的接口10几次甚至是几十次，这是我们不想看到的事情，那么整个系统能不能对这样的请求进行优化，比如改成批量接口或者提高整个系统调用的并行度？在美团点评内部我们会针对这样的链路进行筛选分析，然后提供给业务方进行优化。

异常log绑定

通过MTrace不仅能做上述这些事情，通过它的特性，还能携带很多业务感兴趣的数据。因为MTrace可以做到数据和一次请求的绑定以及数据在一次请求的网络中传递。比如一些关键的异常log，一般服务的异常log很有可能是因为上游或者下游的异常造成的，那就需要我们手动地对各个不同服务的异常log做mapping。看这次的异常log对应到上游服务的哪个log上，是不是因为上游传递的一些参数造成了该次异常？而通过MTrace就可以将请求的参数、异常log等信息通过traceId进行绑定，很容易地就把这些信息聚合到了一起，方便业务端查询问题。

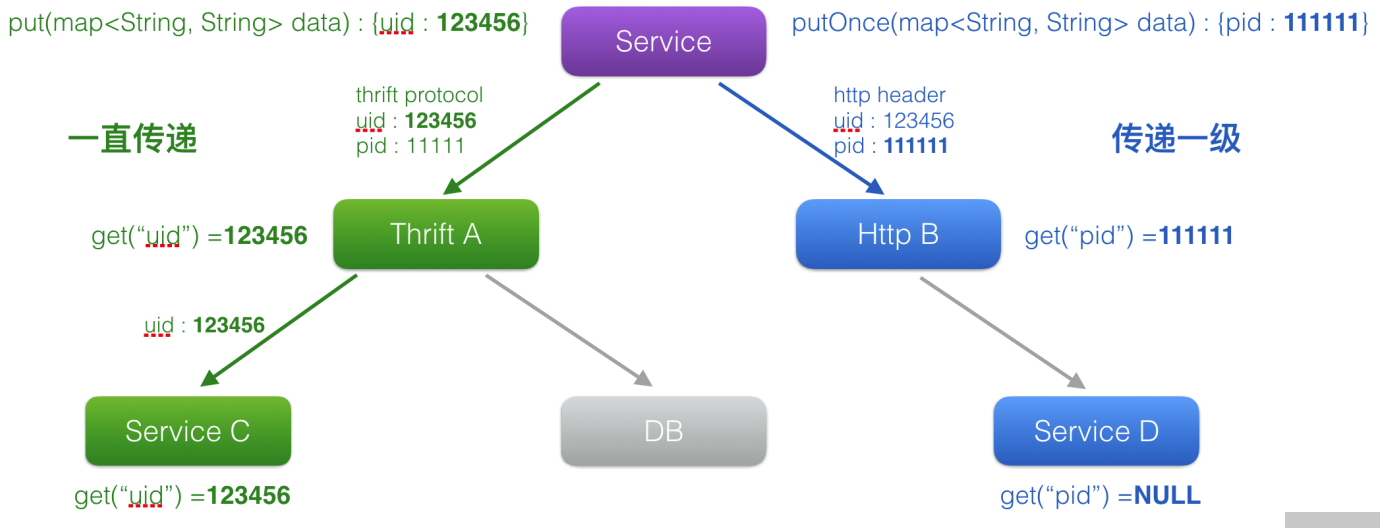
透明传输数据

业务端往往有这样的需求，它希望一些参数能在一次分布式请求一直传递下去，并且可以在不同的RPC中间件间传递。MTrace对该类需求提供了两个接口：

```
put(map<String, String> data)
putOnce(map<String, String> data)
```

- put 接口：参数可以在一次分布式请求中一直传递。
- putOnce 接口：参数在一次分布式请求中只传递一级。

如下图所示



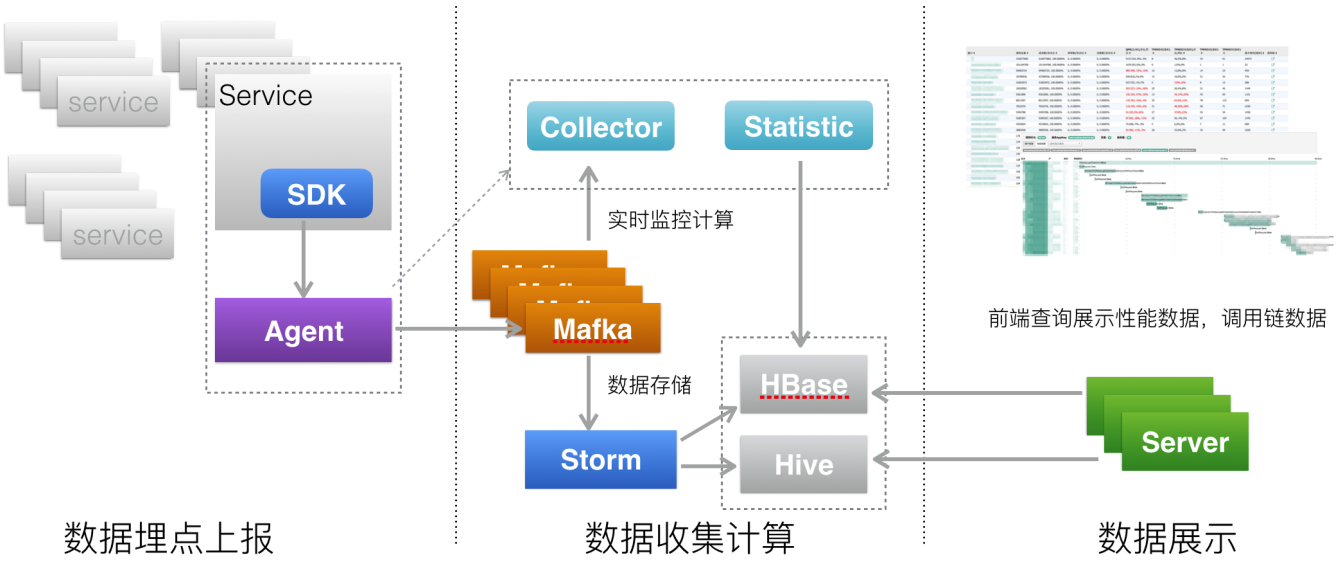
- 左侧绿色部分是put接口，service中调用了put接口传递了uid=123456这个参数，它会在网络中一直传递，

可以在服务A中通过get("uid")的方式获取参数值，也可以在服务C中通过get("uid")的方式获取参数值。

- 右侧蓝色部分是putOnce接口，service中调用了putOnce接口传递pid=11111，它只会传递一级，可以在服务B中通过get("pid")的方式获取参数值，但是在服务D中就获取不到pid的值了。

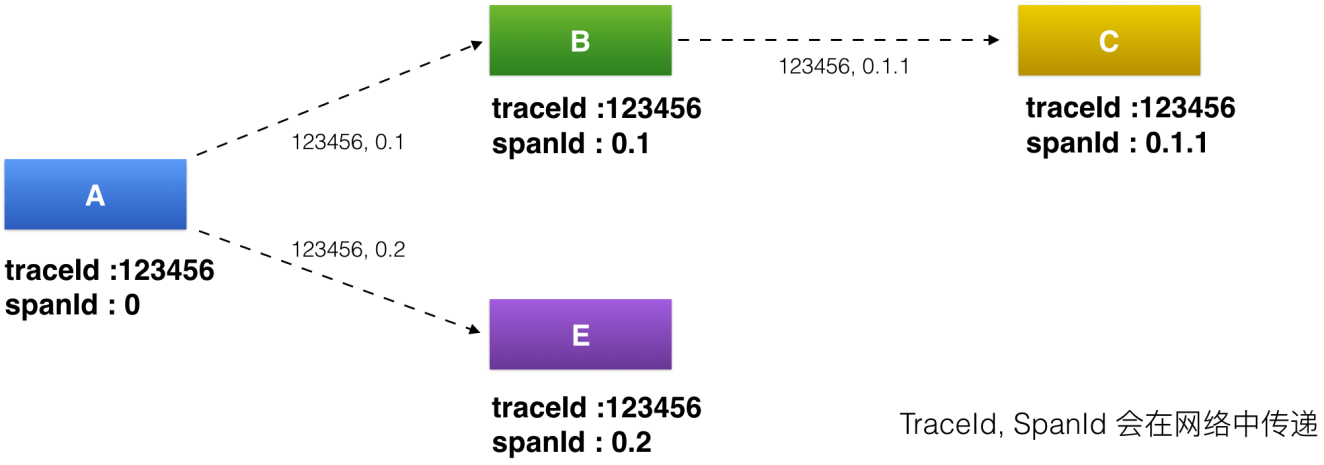
以上的两种接口可以用于业务自定义传递数据，比如通过传递一个服务标识，用于AB test，下游的所有服务获取到test的标识就会走test的策略，即上游服务可以传递一些参数，控制所有下游服务的逻辑。当然业务也可以通过该接口传递一些临时性的数据。

系统架构



主要分为三层：数据埋点上报、数据收集计算、数据前端展示。

基本概念



traceId

全局唯一 64位整数 用于标识一次分布式请求 会在RPC调用的网络中传递

全局唯一，64位整数，用于标识一次分布式请求，会在RPC调用的网络中传递。

spanId

签名方式生成:0, 0.1, 0.1.1, 0.2。用于标识一次RPC在分布式请求中的位置，比如0.2就是0节点服务调用的第二个服务。

annotation

业务端自定义埋点，业务感兴趣的想上传到后端的数据，比如该次请求的用户ID等。

数据埋点

埋点SDK

提供统一的SDK，在各个中间件中埋点，生成traceID等核心数据，上报服务的调用数据信息。

- 生成调用上下文；
- 同步调用上下文存放在ThreadLocal, 异步调用通过显式调用API的方式支持；
- 网络中传输关键埋点数据，用于中间件间的数据传递，支持Thrift, HTTP协议。

业内有些系统是使用注解的方式实现的埋点，这种方式看似很优雅，但是需要业务方显式依赖一些AOP库，这部分很容易出现问题，因为AOP方式太过透明，导致查问题很麻烦，而且业务方配置的东西越多越容易引起一些意想不到的问题，所以我们的经验是尽量在各个统一的中间件中进行显式埋点，虽然会导致代码间耦合度增加，但是方便后续定位问题。其次，为了整个框架的统一，MTrace并非仅支持Java一种语言，而AOP的特性很多语言是不支持的。

Agent

- 透传数据，用作数据转发；
- 做流量控制；
- 控制反转，很多策略可以通过agent实现，而不需要每次都升级业务代码中的SDK。

Agent仅仅会转发数据，由Agent判断将数据转发到哪里，这样就可以通过Agent做数据路由、流量控制等操作。也正是由于Agent的存在，使得我们可以在Agent层实现一些功能，而不需要业务端做SDK的升级，要知道业务端SDK升级的过程是很缓慢的，这对于整个调用链的系统来说是不可接受的，因为MTrace整个系统是针对庞大的分布式系统而言的，有一环的服务缺失也会造成一定的问题。

目前MTrace支持的中间件有:

- 公司内部RPC中间件

- http中间件
- mysql中间件
- tair中间件
- mq中间件

数据埋点的四个阶段：

- Client Send：客户端发起请求时埋点，需要传递一些参数，比如服务的方法名等

```
Span span = Tracer.clientSend(param);
```

- Server Recieve：服务端接收请求时埋点，需要回填一些参数，比如traceId，spanId

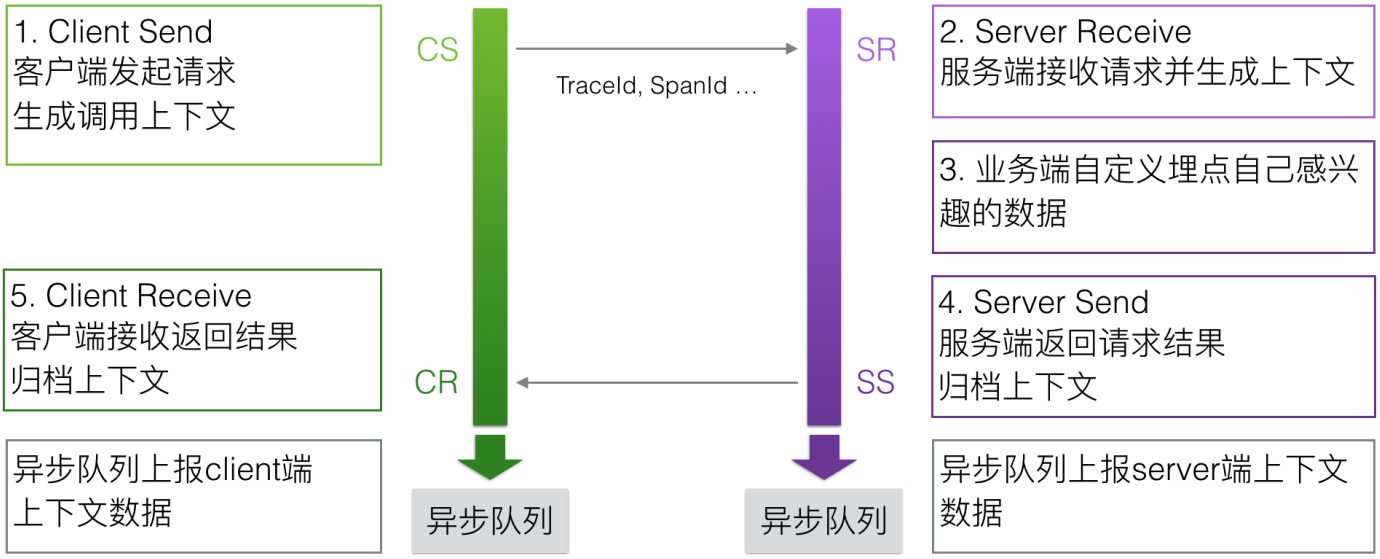
```
Tracer.serverRecv(param);
```

- ServerSend：服务端返回请求时埋点，这时会将上下文数据传递到异步上传队列中

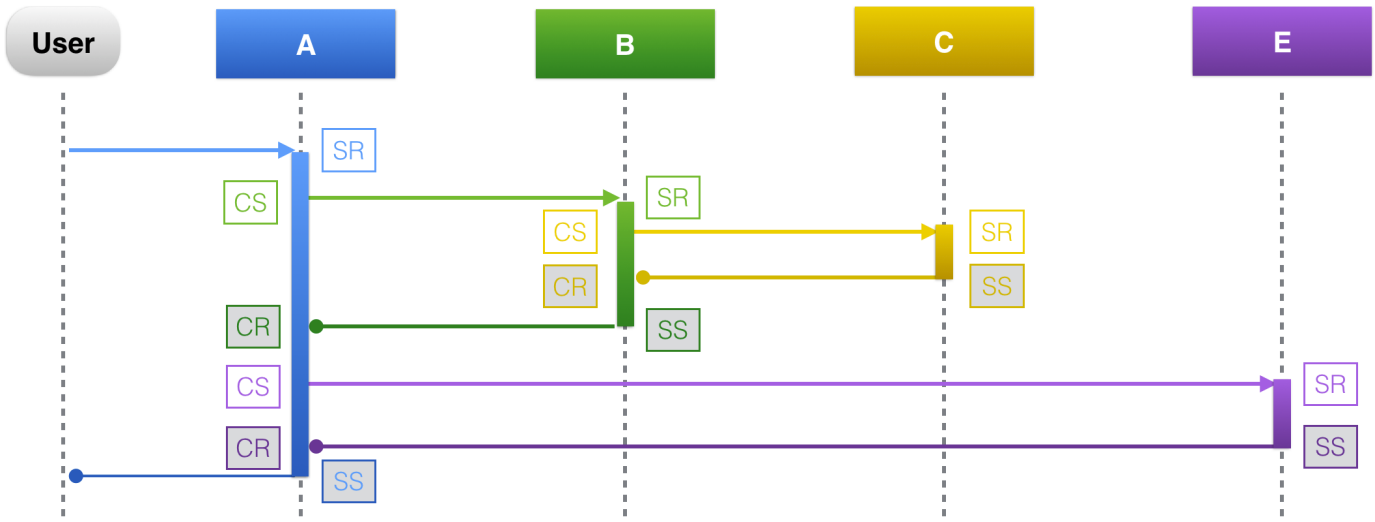
```
Tracer.serverSend();
```

- Client Recieve：客户端接收返回结果时埋点，这时会将上下文数据传递到异步上传队列中

```
Tracer.clientRecv();
```



埋点上下文

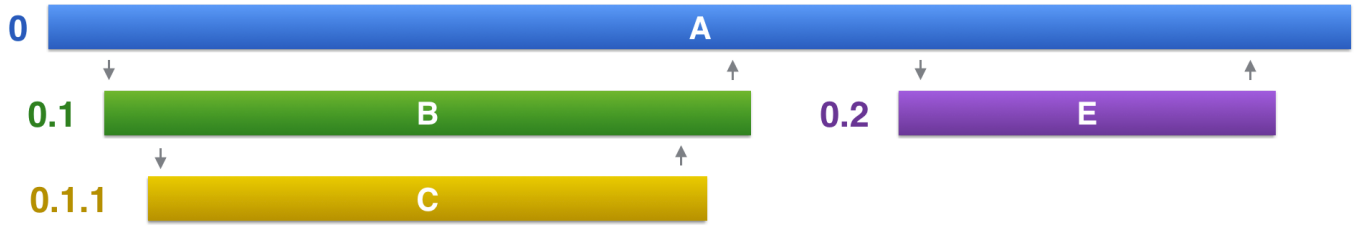


CS, SR：创建上下文 **CR, SS**：归档上下文

上图CS、SR为创建上下文的位置，CR、SS为归档上下文的位置。

上下文归档

上下文归档，会把上下文数据异步上传到后端，为了减轻对业务端的影响，上下文上报采用的是异步队列的方式，数据不会落地，直接通过网络形式传递到后端服务，在传递之前会对数据做一层压缩，主要是压缩比很可观，可以达到10倍以上，所以就算牺牲一点CPU资源也是值得的。具体上报的数据如图所示：




```
traceId 123456, spanId 0.1.1, appKey C, method C.method, start 106, duration 30, side server
traceId 123456, spanId 0.1.1, appKey B, method C.method, start 105, duration 33, side client
traceId 123456, spanId 0.1  , appKey B, method B.method, start 103, duration 38, side server
traceId 123456, spanId 0.1  , appKey A, method B.method, start 103, duration 38, side client
traceId 123456, spanId 0.2  , appKey E, method E.method, start 148, duration 12, side server
traceId 123456, spanId 0.2  , appKey A, method E.method, start 146, duration 15, side client
traceId 123456, spanId 0    , appKey A, method A.method, start 100, duration 82, side server
```

我们之前在数据埋点时遇到了一些问题：

- 异步调用

- 异步IO造成的线程切换，不能通过ThreadLocal传递上下文。
- 显式的通过API进行埋点传递，切换前保存，切换后还原。
- 提供封装好的ThreadPool库。

- 数据量大，每天千亿级别的数据
  - 批量上报
  - 数据压缩
  - 极端情况下采样

数据存储

Kafka使用

我们在SDK与后端服务之间加了一层Kafka，这样做既可以实现两边工程的解耦，又可以实现数据的延迟消费。我们不希望因为瞬时QPS过高而引起的数据丢失，当然为此也付出了一些实效性上的代价。

实时数据Hbase

调用链路数据的实时查询主要是通过Hbase，使用traceID作为RowKey，能天然的把一整条调用链聚合在一起，提高查询效率。

离线数据Hive

离线数据主要是使用Hive，可以通过SQL进行一些结构化数据的定制分析。比如链路的离线形态，服务的出度入度(有多少服务调用了该服务，该服务又调用了多少下游服务)

前端展示

前端展示，主要遇到的问题是NTP同步的问题，因为调用链的数据是从不同机器上收集上来的，那么聚合展示的时候就会有NTP时间戳不同步的问题，这个问题很难解决，于是我们采取的方式是前端做一层适配，通过SpanId定位调用的位置而不是时间，比如0.2一定是发生在0.1这个Span之后的调用，所以如果时间出现漂移，就会根据SpanId做一次校正。即判断时间顺序的优先级为最高是spanid,然后是时间戳。

总结

核心概念：调用链；
用途：定位系统瓶颈，优化系统结构、统计系统指标、分析系统数据；
架构：埋点上报、收集计算、展示分析