GIAC | 全球互联网架构大会
GLOBAL INTERNET ARCHITECTURE CONFERENCE

bilibili分布式链路监控

毛剑
bilibili研发总监

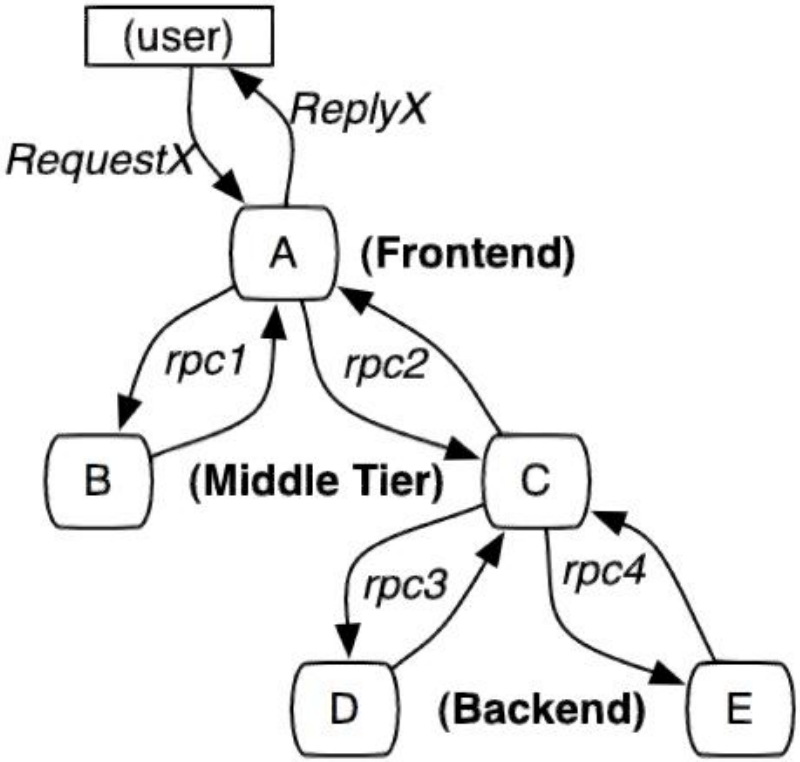
2017.thegiac.com

设计目标

- 无处不在的部署
- 持续的监控
- 低消耗
- 应用级的透明
- 延展性
- 低延迟

分布式跟踪

参考Google Dapper 论文实现，为每个请求都生成一个全局唯一的traceid，端到端透传到上下游所有节点，每一层生成一个spanid,通过traceid将不同系统孤立的调用日志和异常信息串联一起，通过spanid和level 表达节点的父子关系；



核心概念：

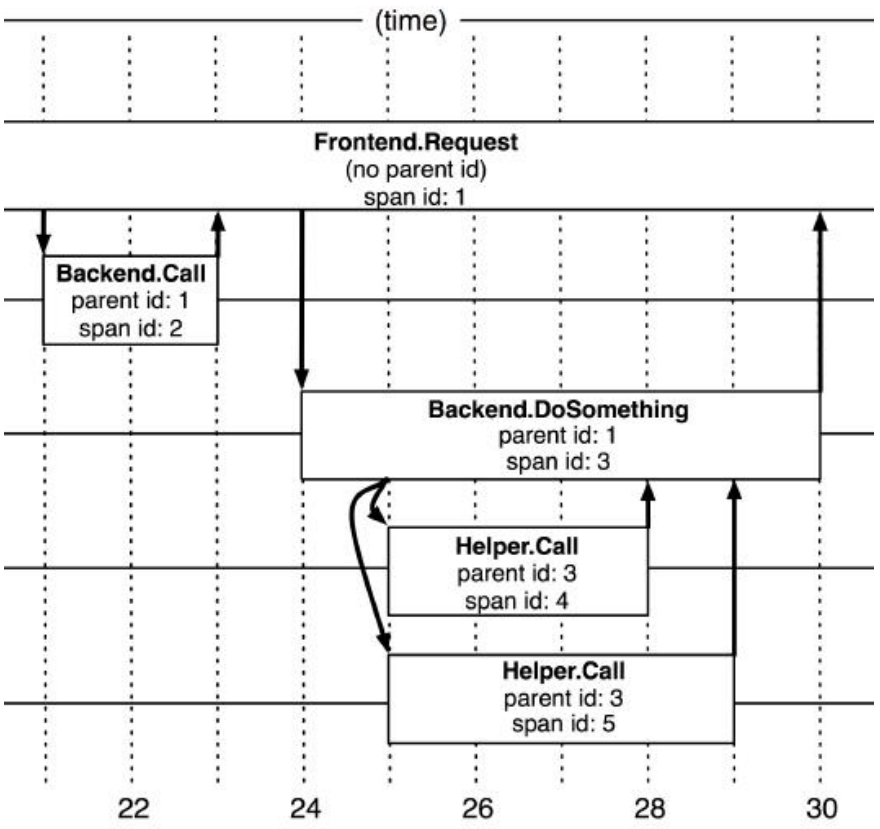
- Tree
- Span
- Annotation

跟踪树

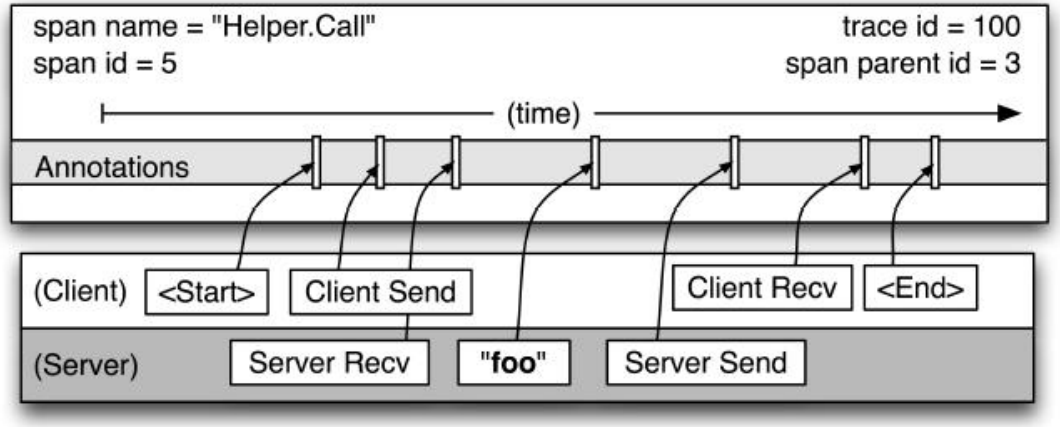
在跟踪树结构中，树节点是整个架构的基本单元，而每一个节点又是对span的引用。虽然span在日志文件中只是简单的代表span的开始和结束时间，他们在整个树形结构中却是相对独立的；

核心概念：

- TraceID
- SpanID
- ParentID
- Family & Title



跟踪Span



- 追踪信息包含时间戳、事件、方法名（Family+Title）、注释（TAG/Comment）
- 客户端和服务器上的时间戳来自不同的主机，我们必须考虑到时间偏差，RPC客户端发送一个请求之后，服务器端才能接收到，对于响应也是一样的（服务器先响应，然后客户端才能接收到这个响应）。这样一来，服务器端的RPC就有一个时间戳的一个上限和下限

接口抽象

```
// Trace trace common interface.
type Trace interface {
    // Fork fork a trace with client trace.
    Fork(family, title string) Trace

    // Annotation record the trace with a event UserDefine and called it when
    // you want more info, support *net.URL, []byte, string.
    Annotation(a string)
    //
    // Finish when trace finish call it.
    Finish(err *error)

    // SetInfo set address, comment info into trace.
    SetInfo(address, comment string)

    // Scan scan trace into info.
    Scan(ti *Info)
}
```

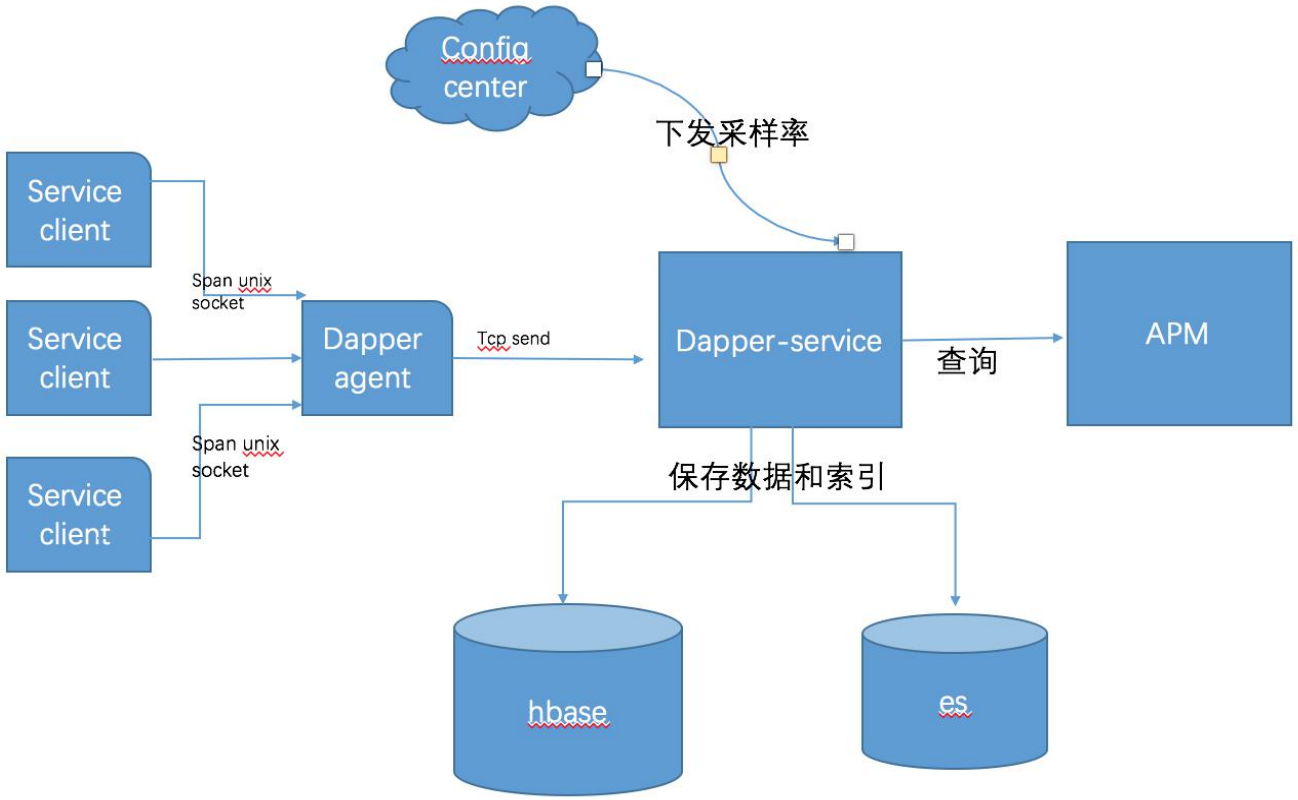
跟踪植入点

Dapper可以以对应用开发者近乎零浸入的成本对分布式控制路径进行跟踪，几乎完全依赖于基于少量通用组件库的改造。如下：

- 当一个线程在处理跟踪控制路径的过程中，Dapper把这次跟踪的上下文的在ThreadLocal中进行存储，在Go语言中，约定每个方法首参数为context（上下文）
- 覆盖通用的中间件&通讯框架、不限于：redis、memcache、rpc、http、database、queue

```
func (engine *Engine) handleContext(c *Context) {
    var cancel func()
    req := c.Request
    req.ParseForm()
    // handle http request
    // get derived trace from http request header
    tr := trace.NewTrace(req.URL.Path, httpx.Trace(req
    tr.SetInfo(engine.address, req.Method)
    defer tr.Finish(nil)
    // get derived timeout from http request header
    tm := httpx.Timeout(req)
    md := metadata.New(map[string]interface{}{
        "trace":   tr,
        "user":    httpx.User(req),
        "timeout": tm,
    })
    ctx := metadata.NewContext(context.Background(), m
    if tm > 0 {
        c.Context, cancel = context.WithTimeout(ctx, t
    } else {
        c.Context, cancel = context.WithCancel(ctx)
    }
    defer cancel()
    c.Next()
}
```

跟踪的收集


Config center
下发采样率
Service client
Service client
Service client
Span unix socket
Span unix socket
Dapper agent
Tcp send
Dapper-service
查询
APM
保存数据和索引
hbase
es

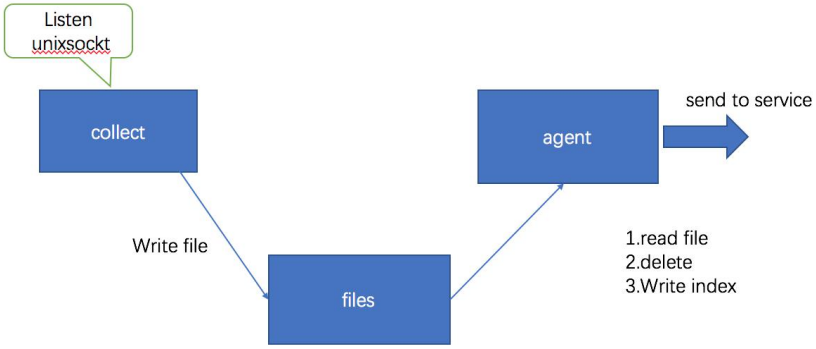
收集优化&存储

- tarce对象使用对象池分配，减轻GC压力；
- 使用异步队列方式发送采样信息，在consumer中聚合后定时或者定量发送，提升吞吐，减少系统调用；
- HBase存储所有采样的全量数据traceid作为rowkey、每一个span存储为column；
- ES为主要字段建立索引，索引文件按天存储，family和title单独索引、用户依赖关系存储；

跟踪采集Agent

客户端通过unixsockt的进行异步发送trace信息给agent，宿主机上面部署一个collect 日志收集进程，监听本地sock文件，共享给容器或者进程，聚合收集日志，流程：

- collect 将收集到的日志批量写入磁盘，以时间戳作为文件名，存储为固定大小的小文件；
- agent 与service建立tcp连接，读取相应路径下面的所有日志文件，按照先后顺序发送到服务器端，每10秒将读取位置写入索引文件。读取完成并删除日志文件；



跟踪损耗

- 处理跟踪消耗：1. 正在被监控的系统在生成追踪和收集追踪数据的消耗导致系统性能下降，2. 需要使用一部分资源来存储和分析跟踪数据：

    1. 是Dapper性能影响中最关键的部分，因为收集和分析可以更容易在紧急情况下被关闭，ID生成耗时、创建Span等；

    1. 修改agent nice值，以防在一台高负载的服务器上发生cpu竞争；

- 采样：如果一个显着的操作在系统中出现一次，他就会出现上千次，基于这个事情我们**不全量收集数据**，通过模型来预估真实情况，Reference：Uncertainty in Aggregate Estimates from Sampled Distributed Traces

跟踪采样

- 固定采样，1/1024：

这个简单的方案是对我们的高吞吐量的线上服务来说是非常有用，因为那些感兴趣的事件(在大吞吐量的情况下)仍然很有可能经常出现，并且通常足以被捕捉到。然而，在较低的采样率和较低的传输负载下可能会导致错过重要事件，而想用较高的采样率就需要能接受的性能损耗。对于这样的系统的解决方案就是覆盖默认的采样率，这需要手动干预的，这种情况是我们试图避免在dapper中出现的；

- 应对积极采样：

我们理解为单位时间期望采集样本的条目，在高QPS下，采样率自然下降，在低QPS下，采样率自然增加；比如1s内某个接口采集1条；

```
if _, ok := _ignoreTitles[title]; ok {
    t.sampled = false
} else if _ratio <= 0 {
    t.sampled = false
} else if sample := _ratio / float32(atomic.LoadInt32(&_rqps[index])); rand.Float32() < sample {
    t.sampled = true
    t.sample = sample
}
```

跟踪采样

- 二级采样：

容器节点数量多，即使使用积极采样仍然会导致采样样本非常多，所以需要控制写入中央仓库的数据的总规模，利用所有span都来自一个特定的跟踪并分享同一个跟踪ID这个事实，虽然这些span有可能横跨了数千个主机。对于在收集系统中的每一个span，我们用hash算法把跟踪ID转成一个标量Z，这里0<=Z<=1，我们选择了运行期采样率，这样就可以优雅的去掉我们无法写入到仓库中的多余数据，我们还可以通过调节收集系统中的二级采样率系数来调整这个运行期采样率，最终我们通过后端存储压力把策略下发给Agent采集系统，实现精准的二级采样；

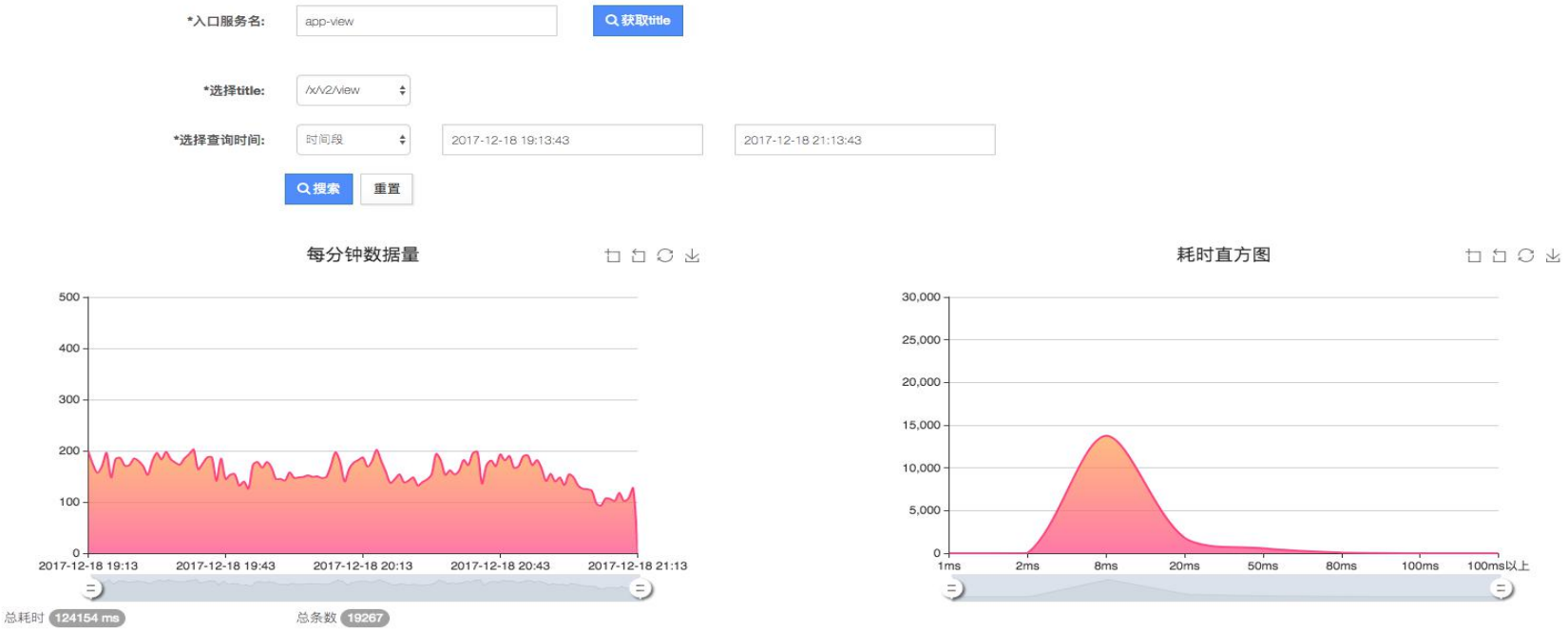
- 下游采样：

越被依赖多的服务，网关层使用积极采样以后，对于Downstream的服务采样率仍然很高，我们会结合第二篇论文来解决，目前TODO中；

API

- 搜索：

按照Family（服务名）、Title（接口）、时间、调用者等维度进行搜索，依据Cost Metric（如：时间成本)进行排序，列出被采集的样本的基本信息、耗时占比等；



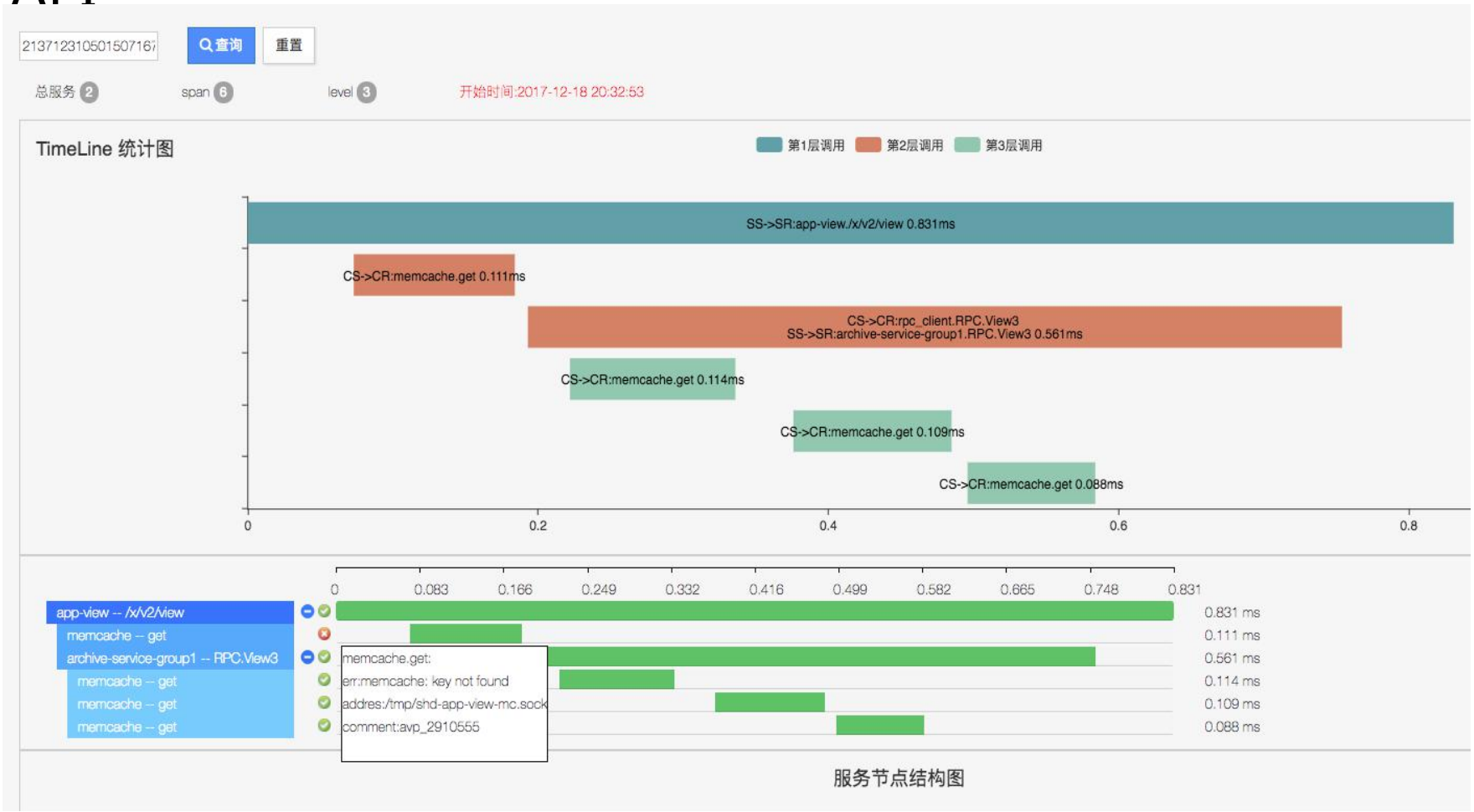
API

业务名	tid	时间	耗时(单位:毫秒)	耗时占比
app-view	947121184192796366	2017-12-18 19:50:27	151.797067	0.0001%
app-view	7661322608626581778	2017-12-18 20:43:30	119.329046	0.0001%
app-view	7684744523414570077	2017-12-18 20:32:24	112.495233	0.0001%
app-view	2109500531160215698	2017-12-18 21:03:23	112.390763	0.0001%
app-view	6894161702837853564	2017-12-18 19:41:53	108.979142	0.0001%
app-view	5871474359080073289	2017-12-18 20:04:50	105.573471	0.0001%
app-view	5824935287687710677	2017-12-18 19:47:57	103.624325	0.0001%
app-view	7344005161463025713	2017-12-18 19:47:55	103.234574	0.0001%

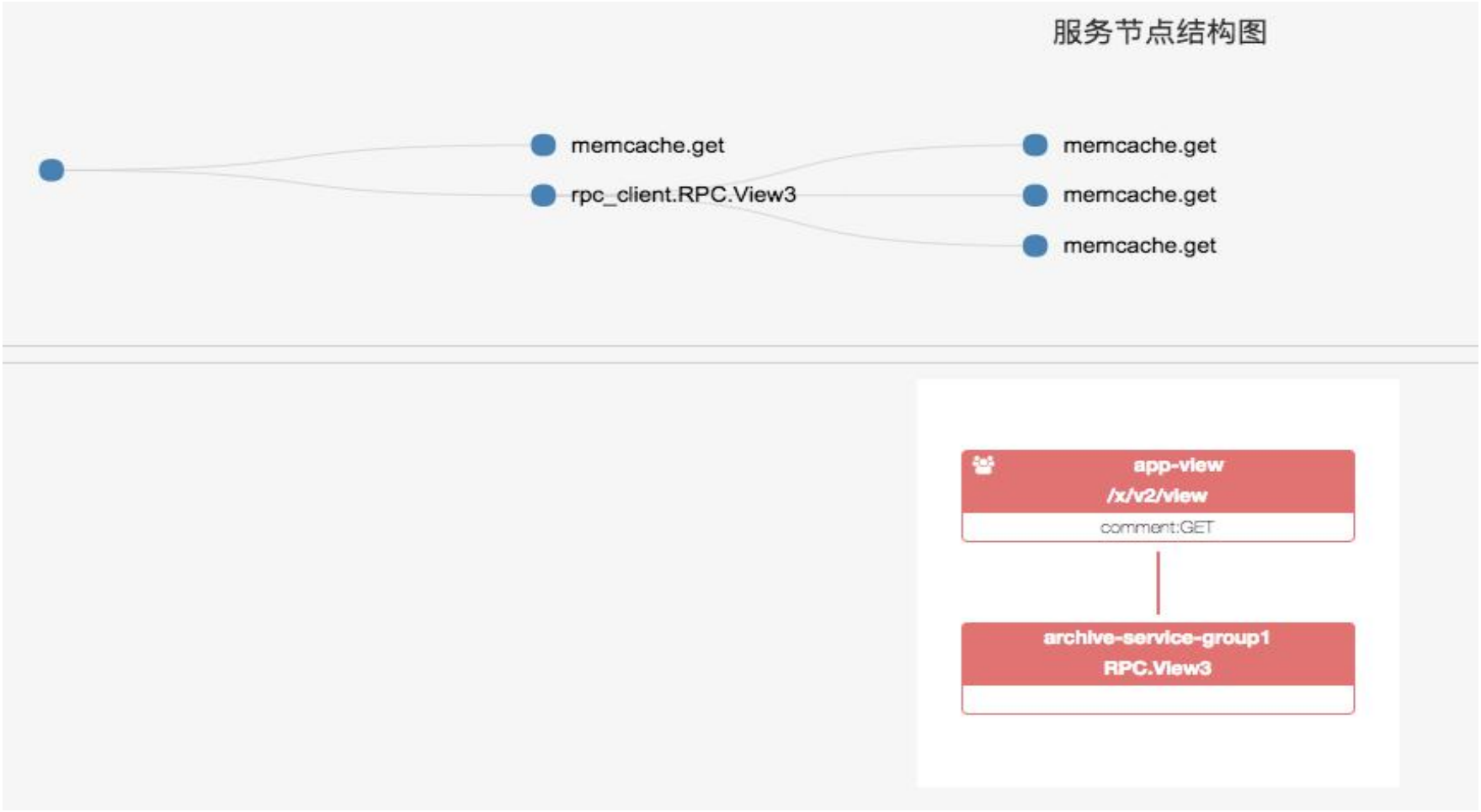
- 详情：

根据单个traceid，查看整体链路信息，包含span、level统计，span详情，依赖的服务、组件信息等；

API


查询
重置
总服务 2
span 6
level 3
开始时间:2017-12-18 20:32:53
TimeLine 统计图
第1层调用
第2层调用
第3层调用
SS->SR:app-view./x/v2/view 0.831ms
CS->CR:memcache.get 0.111ms
CS->CR:rpc_client.RPC.View3
SS->SR:archive-service-group1.RPC.View3 0.561ms
CS->CR:memcache.get 0.114ms
CS->CR:memcache.get 0.109ms
CS->CR:memcache.get 0.088ms
0
0.2
0.4
0.6
0.8
0.083
0.166
0.249
0.332
0.416
0.499
0.582
0.665
0.748
0.831
app-view -- /x/v2/view
memcache -- get
archive-service-group1 -- RPC.View3
memcache -- get
memcache -- get
memcache -- get
memcache.get:
err:memcache: key not found
addres:/tmp/shd-app-view-mc.sock
comment:avp_2910555
0.831 ms
0.111 ms
0.561 ms
0.114 ms
0.109 ms
0.088 ms
服务节点结构图

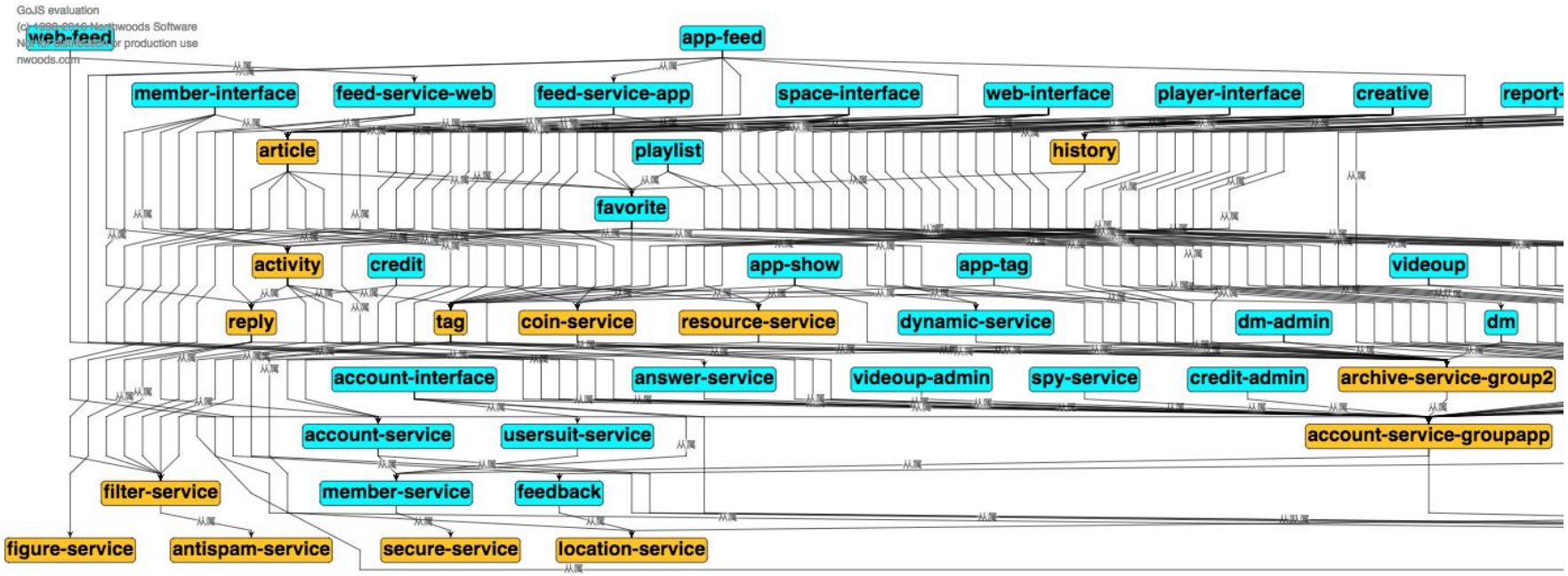
API


服务节点结构图
memcache.get
rpc_client.RPC.View3
memcache.get
memcache.get
memcache.get
app-view
/x/v2/view
comment:GET
archive-service-group1
RPC.View3

API

- 依赖：

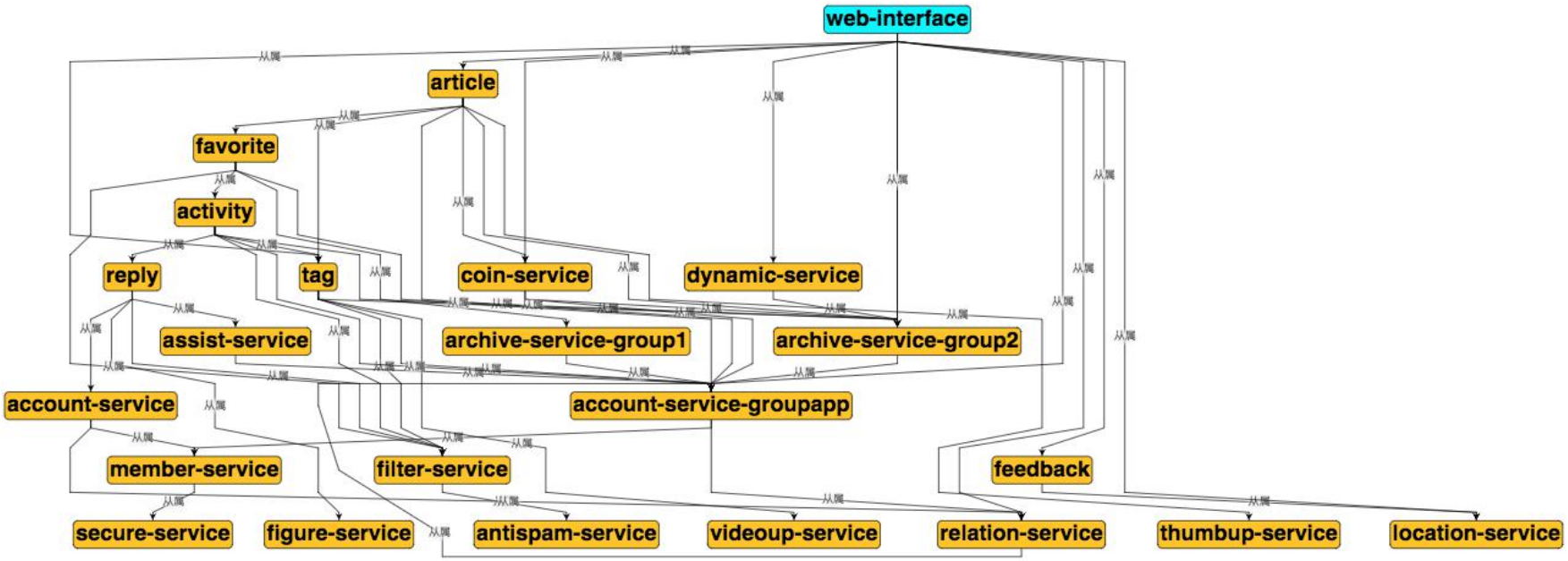
由于服务之间的依赖是动态改变的，所以不可能仅从配置信息上推断出所有这些服务之间的依赖关系，能够推算出任务各自之间的依赖，以及任务和其他软件组件之间的依赖。



API

- 依赖搜索：

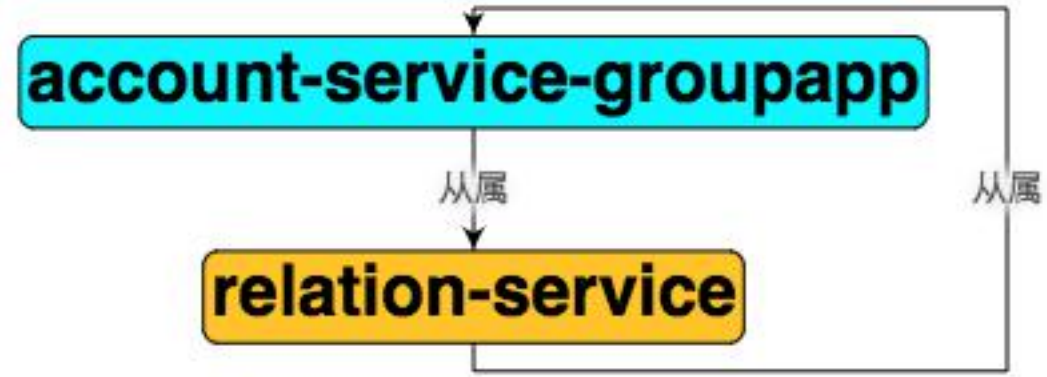
搜索单个服务的依赖情况，方便我们做“异地多活”时候来全局考虑资源的部署情况，以及区分服务是否属于多活范畴，也可以方便我们经常性的梳理依赖服务和层级来优化我们的整体架构可用性；



API

- 推断环依赖：

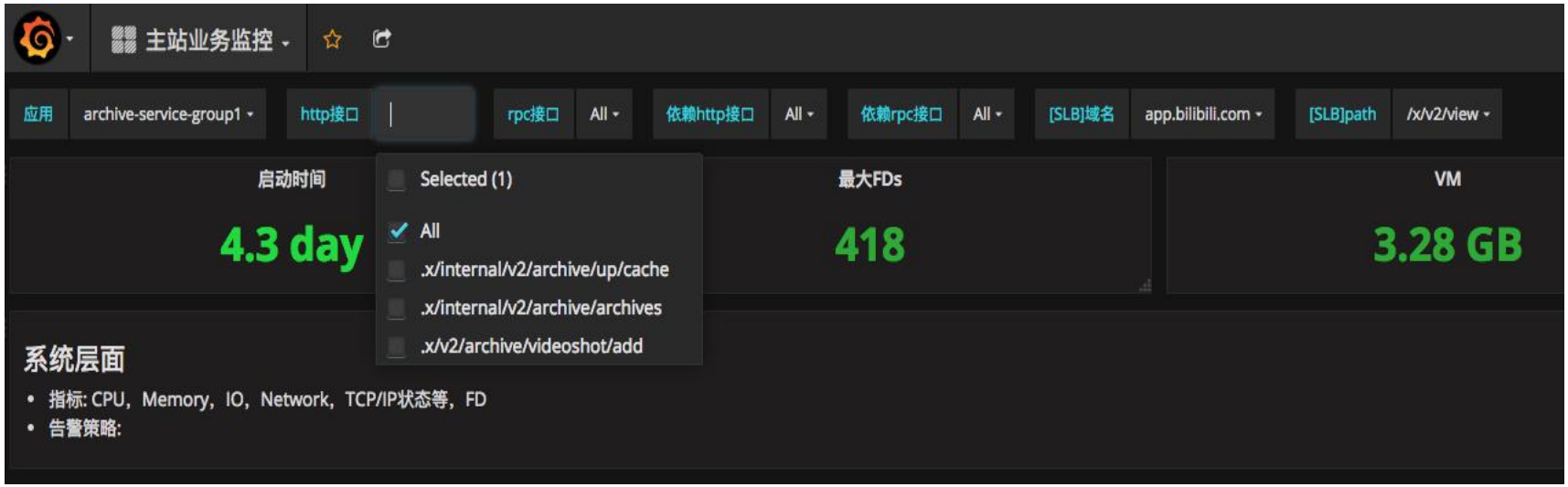
一个复杂的业务架构，很难避免全部是层级关系的调用，但是我们要尽可能保证一点：调用栈永远向下，即：不产生环依赖；



API

- 服务元数据：

我们发现，服务内部的HTTP、RPC、Comment信息全部可以自动根据Dapper API生成，因此我们不再开发针对框架层的服务树元数据注册，依赖这些信息，很容易把监控系统串联的很好、以及运维内部的服务树系统；



链路系统经验

- 性能优化：

1、不必要的串行调用；2、缓存读放大；3、数据库写放大；4、服务接口聚合调用；

- 异常日志系统集成：

如果这些异常发生在Dapper跟踪采样的上下文中，那么相应的traceid和spanid也会作为元数据记录在异常日志中。异常监测服务的前端会提供一个链接，从特定的异常信息的报告直接导向到他们各自的分布式跟踪；

- 用户日志集成：

在请求的头中返回traceid，当用户遇到故障或者上报客服我们可以根据traceid作为整个请求链路的关键字，再根据接口级的服务依赖接口所涉及的服务并行搜索ES Index，聚合排序数据，就比较直观的诊断问题了；

TODO

- 容量预估：

根据入口网关服务，推断整体下游服务的调用扇出来精确预估流量再各个系统的占比

- 网络热点&易故障点：

我们内部RPC框架还不够统一，以及基础库的组件部分还没解决拿到应用层协议大小，如果我们收集起来，可以很简单的实现流量热点、机房热点、异常流量等情况。同理容易失败的span，很容易统计出来，方便我们辨识服务的易故障点；

- opentraceing：

标准化的推广，上面几个特性，都依赖span TAG来进行计算，因此我们会逐步完成标准化协议，也更方便我们开源，而不是一个内部“特殊系统”；

GIAC | 全球互联网架构大会
GLOBAL INTERNET ARCHITECTURE CONFERENCE

GIAC

全球互联网架构大会

GLOBAL INTERNET ARCHITECTURE CONFERENCE

扫码关注GIAC公众号

2017.thegiac.com