## 链路跟踪分享&评审

#### 链路跟踪的使用场景和价值

[微服务](https://so.csdn.net/so/search?q=%E5%BE%AE%E6%9C%8D%E5%8A%A1&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_42103983/article/details/_blank)架构是一个分布式架构，它按业务划分服务单元，一个分布式系统往往有很多个服务单元。由于服务单元数量众多，业务的复杂性，如果出现了错误和异常，很难去定位。主要体现在，一个请求可能需要调用很多个服务，而内部服务的调用复杂性，决定了问题难以定位。

**所以微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎样的，从而达到每个请求的步骤清晰可见，出了问题，很快定位。**

**它可以在复杂的服务调用中定位问题，还可以在新人加入后台团队之后，让其清楚地知道自己所负责的服务在哪一环。**

除此之外，**如果某个接口突然耗时增加，也不必再逐个服务查询耗时情况，我们可以直观地分析出服务的性能瓶颈，方便在流量激增的情况下精准合理地扩容**。

对比与日志系统，链路跟踪更加聚集于服务之间的rpc调用的追踪，可以通过web界面更加直观的看到调用延时和调用顺序，可以一键点击进去看到过程中的问题。

#### 技术选型

|  |  |  |  |
| --- | --- | --- | --- |
| **框架** | SkyWalking | Zipkin | Jaeger |
| **开发团队** | **华为** | **社区** | **Uber** |
| OpenTracing | **支持** | **支持** | **支持** |
| 语言支持 | Java、NET Core、NodeJS 、PHP | Go，Java，Ruby，C++，Python（Progress） | Python，Go，Node，Java，C++，C#，PHP，Ruby |
| 存储 | ES、H2、Mysql、TIDB、Sharding sphere | 内存、Cassandra、Elasticsearch | 内存、Cassandra、Elasticsearch |
| Span 传输 | gRPC | HTTP，KAFKA | UTP，HTTP |
| 易用性 | 简单易接入 | 少数语言支持差，如Python | 接入简单，各种语言SDK丰富 |
| 业务代码侵入性 | 低 | 中 | 中 |

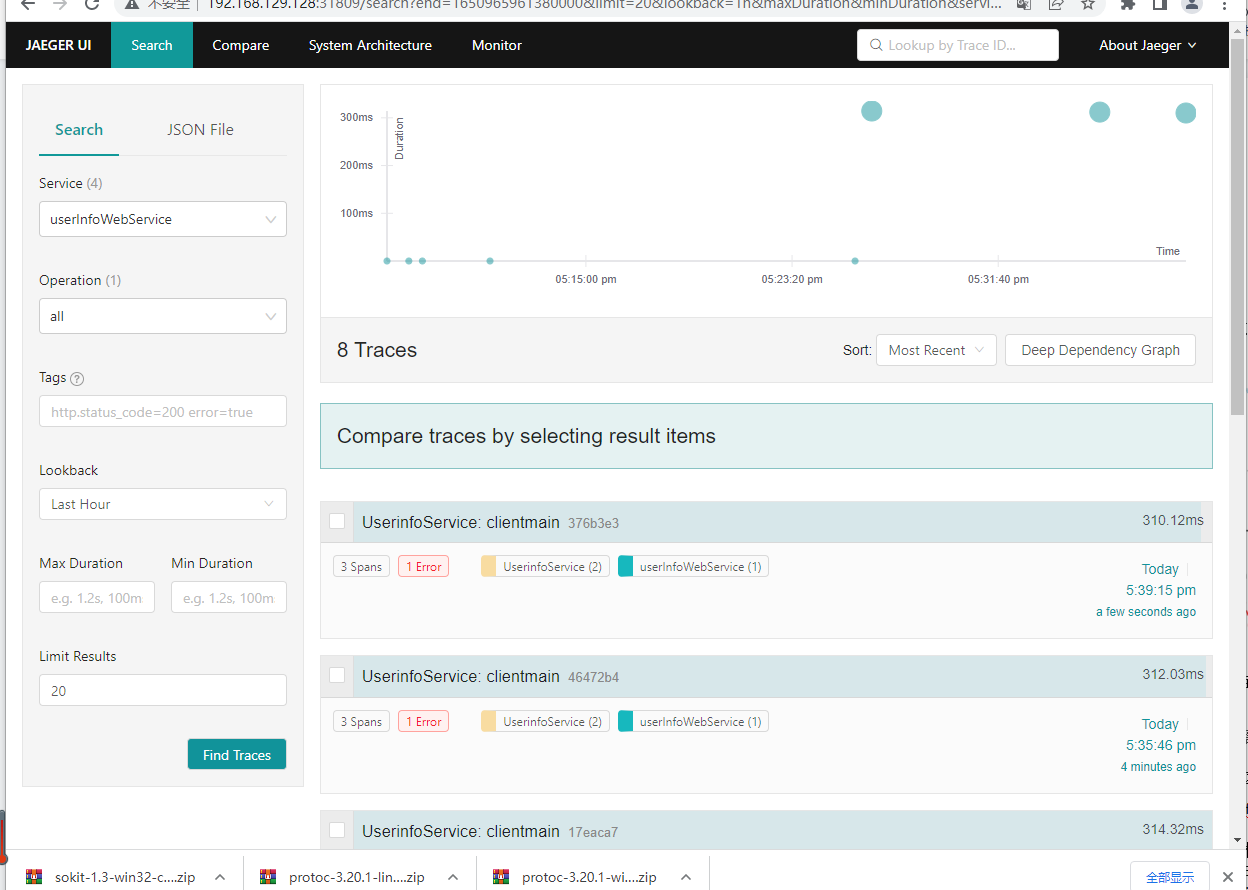
**由于SkyWalking主要是Java社区的中间件，首先淘汰。Jaeger出现的更晚，更新，有动态采样的机制较Zipkin更先进，动态采样0.1表示10调链路概率上只采一条数据上报，因此优先选用Jaeger**

#### 概念梳理

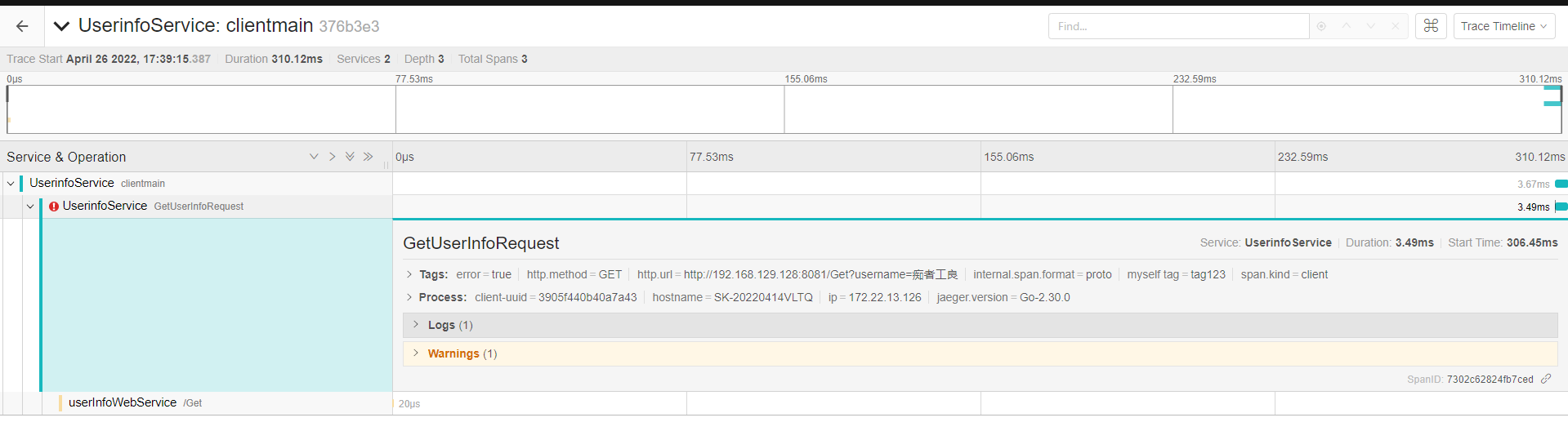
1. **Tracing: 一整条链路叫tracing,一个tracing跨多个服务进程，里面会包含一个唯一requestid.会跨越多个SPAN**
2. **SPAN,在一个Tracing中的每一次RPC调用称为SPAN。包括进程内部的需要追踪调用和跨进程的RPC，进程内部的调用最终使用Context(Golang),跨进程的SPAN通过HTTP head信息串联到一个Tracing中。**

#### 使用效果

整体图,可筛选Tracing

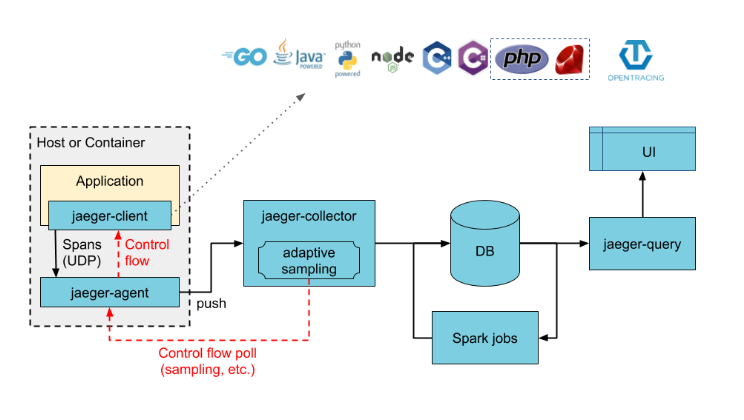


单次Tracing可以查阅具体Span



点击单次Span可以看到延时，Tag, Log， Http状态码，http method, hostname, ip

#### 基于Jaeger链路跟踪原理和架构图



整个系统由以下几个中间件构成

Jaeger-agent

Jaeger-collector

Jaeger-query

Elasticsearch

1. Jaeger-agent: 负责和客户端通信，把收集到的追踪信息上报到收集器 Jaeger Collector本地部署类似这样：

nohup ./jager-agent --config-file=/etc/jaeger-agent.yml

Docker 简单部署类似这样:

docker run --rm -p5775:5775/udp -p6831:6831/udp -p6832:6832/udp -p5778:5778/tcp jaegertracing/jaeger-agent:1.33 --reeger-agent:1.33 --reporter.grpc.host-port=192.168.129.128:31200

1. Jaeger-collector: 将Jaeger-agent的数据进行聚合汇总，然后将数据报入数据库（ES或者Cassandra）,在国内大家还是推荐用Elasticsearch.

本地部署：

nohup ./jager-collector -es.server-urls=http://127.0.0.1:9200 --es.index-prefix=push

Docker 简单部署类似这样：

docker run -d --name=jaeger-collector -p 9411:9411 -p 14250:14250 -p 14268:14268 -p 14269:14269 -e SPAN\_STORAGE\_TYPE=elasticsearch -e ES\_SERVER\_URLS=http://127.0.0.1:9200 jaegertracing/jaeger-collector:1.28

1. Jager-Query: Web查询后端服务

Docker简单部署：

docker run -d --name=jaeger-query -p 16686:16686 -p 16687:16687 -e SPAN\_STORAGE\_TYPE=elasticsearch -e ES\_SERVER\_URLS=http://127.0.0.1:9200 jaegertracing/jaeger-query:1.28

1. Elasticsearch

自行部署

1. AllInOne部署。一个进程或容器包含了所有的中间件,可以用做入门或者demo

docker run -d -p 5775:5775/udp -p 16686:16686 -p 14250:14250 -p 14268:14268 jaegertracing/all-in-one:latest

端口整理

6831 UDP agent 通过兼容性Thrift协议，接收Jaeger thrift类型数据

5778 HTTP agent 配置控制服务接口

16686 HTTP query 客户端前端界面展示端口

14268 HTTP collector 接收客户端上报Http

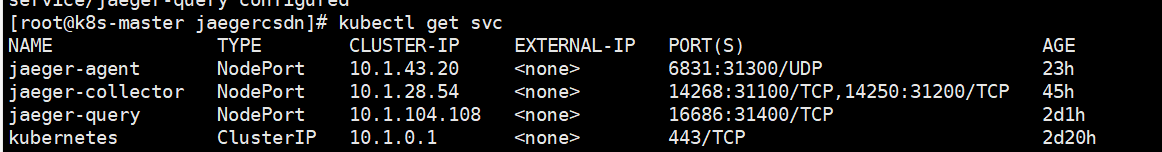
类型数据

14250 HTTP collector 接收客户端Jaeger thrift类型数据（grpc）

#### 基于K8S的部署

初步构想是把jager-agent,jaeger-collocter,jagger-query都部署在一个k8s集群，采用NodePort暴露端口对外

部署结果的效果：



1. jaeger-agent.从业务服务器收集上报数据。虽然这个服务可以部署在业务服务器本地。但是由于业务服务器大多要部署到K8s,jaeger-agent的本质还是一个服务，在公司内。所有服务都在一个VPC，因此.jaeger-agent还是放在jager的k8s里而不是放在业务的K8s里，这样业务侧可以聚焦代码接入

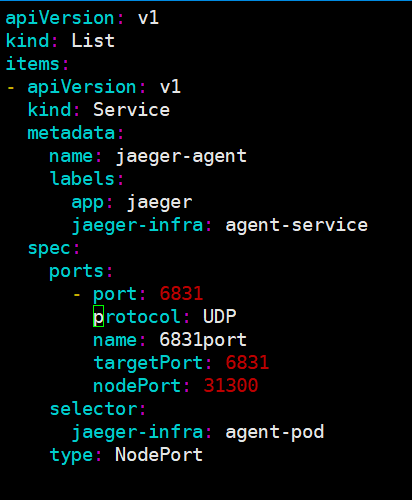
Jaeger-agent是必须暴露端口对外的，因此把k8s svc选用Nodeport型，端口31300转容器6831，jaeger-agent接收来自UDP 6831的上报数据

Deployment yml配置文件如下：



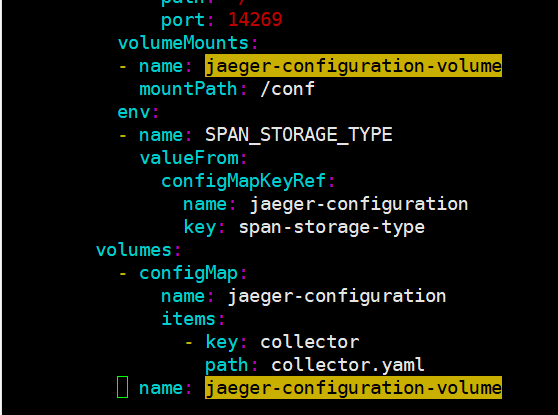
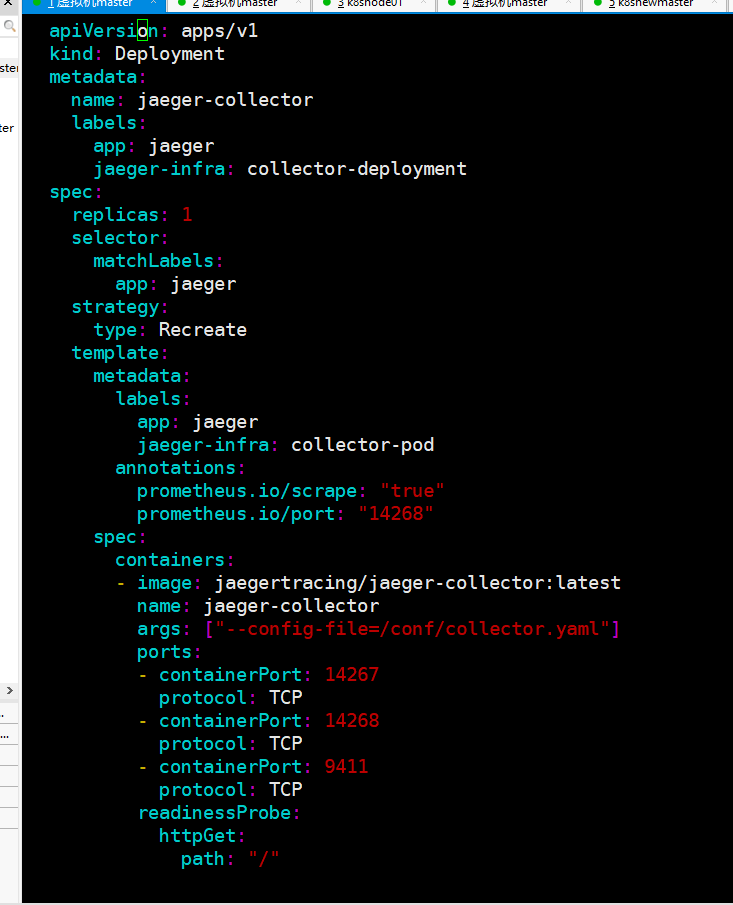
args是上报到jager-collector的地址参数

svc配置文件如下：

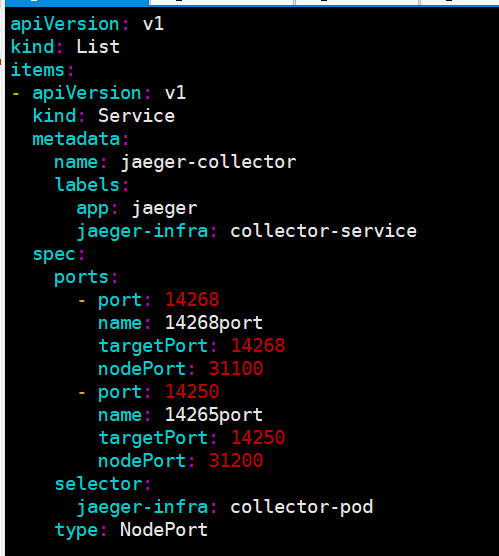


1. Jaeger-collector,从Jaeger-agent获取到数据，也可以接收业务服务器直接上报的HTTP,但不能接收grpc协议上报。grpc协议必须先走agent再走collector

Deployment yml如下：



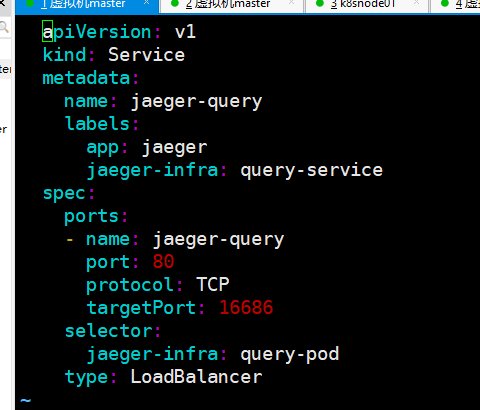
由于collector可以直接接收Http上报，可以用Nodeport暴露出来



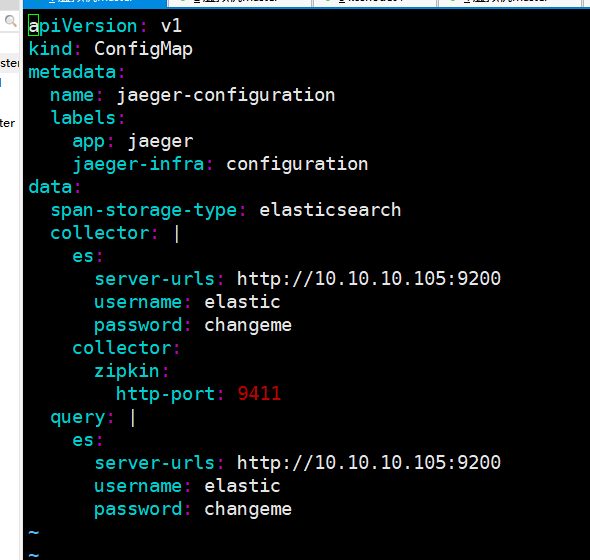
1. Jaeger-query:



Svc 如下：



ConfigMap中配置ElasticSearch的地址:



#### 七．环境部署初步方案

预发环境先使用k8s测试。测试完毕后。再切换到AllinOne模式节省成本

生产环境为了和业务服务器做物理隔离，重新申请3-4台EC2，一台master,2-3台Node,ElasticSeacch 使用现有的Log数据库，Jaeger-query是web服务，只需要一个副本，Jaeger-agent直接对接业务服务器，副本数初步定4个，Jaeger-colletor处理jaeger-agent处理过后的数据，数据量较Jaeger-agent小，初步定2个副本，采样概率使用0.5.后期接入的业务服务越来越多再持续扩容Node和Jaeger-agent和Jaeger-collector的副本数和采样率

#### 业务服务器代码接入

1. 在单进程内部的跟踪：

Step1:首先初始化一个Tracing

Step2:创建一个父Span，并根据这个Span生成一个Context

Step3:在子Rpc模块从Context生成子Span,并设置Tag和Log

Step4:在最后退出的时候，调用Spin.finish()完成上报

最后在web端可以查询到整个调用链

package main

import (

"context"

"fmt"

"github.com/opentracing/opentracing-go"

"github.com/uber/jaeger-client-go"

jaegercfg "github.com/uber/jaeger-client-go/config"

jaegerlog "github.com/uber/jaeger-client-go/log"

"io"

"time"

)

func initJaeger(servieName string) (opentracing.Tracer, io.Closer, error) {

var cfg = jaegercfg.Configuration{

ServiceName: servieName,

Sampler: &jaegercfg.SamplerConfig{

Type: jaeger.*SamplerTypeConst*,

Param: 1,

},

Reporter: &jaegercfg.ReporterConfig{

LogSpans: true,

// 按实际情况替换你的 ip

//CollectorEndpoint: "http://192.168.129.128:31100/api/traces",

LocalAgentHostPort: "192.168.129.128:31300",

},

}

jLogger := jaegerlog.StdLogger

tracer, closer, err := cfg.NewTracer(

jaegercfg.Logger(jLogger),

)

return tracer, closer, err

}

func main() {

tracer, closer,\_ := initJaeger("jaeger-demo")

defer closer.Close()

opentracing.SetGlobalTracer(tracer)//StartspanFromContext创建新span时会⽤到

span := tracer.StartSpan("span\_root")

ctx := opentracing.ContextWithSpan(context.Background(), span)

r1 := foo3("Hello foo3", ctx)

r2 := foo4("Hello foo4", ctx)

fmt.Println(r1, r2)

span.Finish()

}

func foo3(req string, ctx context.Context) (reply string){

//1.创建⼦span

span, \_ := opentracing.StartSpanFromContext(ctx, "span\_foo3")

defer func() {

//4.接⼝调⽤完，在tag中设置request和reply

span.SetTag("request", req)

span.SetTag("reply", reply)

span.Finish()

}()

println(req)

//2.模拟处理耗时

time.Sleep(time.*Second*/2)

//3.返回reply

reply = "foo3Reply"

return

}

//跟foo3⼀样逻辑

func foo4(req string, ctx context.Context) (reply string){

span, \_ := opentracing.StartSpanFromContext(ctx, "span\_foo4")

defer func() {

span.SetTag("request", req)

span.SetTag("reply", reply)

span.Finish()

}()

println(req)

time.Sleep(time.*Second*/2)

reply = "foo4Reply"

return

}

**总结：在进程内部.Span通过context进行父子Span的关联来形成完整的调用链Tracing.**

1. **在实际运用中。单进程跟踪其实不是我们的主要目标，我们的主要目标还是跨服务跨进程追踪，在Http的调用中，使用Http Head进行关联**

**调用端将自己的Span信息注入**

tracer.Inject(childSpan.Context(), opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(req.Header))

服务端：

完整代码如下：

package main

import (

"fmt"

"github.com/gin-gonic/gin"

"github.com/opentracing/opentracing-go"

"github.com/opentracing/opentracing-go/ext"

"github.com/uber/jaeger-client-go"

jaegercfg "github.com/uber/jaeger-client-go/config"

jaegerlog "github.com/uber/jaeger-client-go/log"

"io"

"net/http"

)

func main() {

r := gin.Default()

// 插入中间件处理

r.Use(UseOpenTracing())

r.GET("/Get",GetUserInfo)

r.Run("0.0.0.0:8081") // listen and serve on 0.0.0.0:8080 (for windows "localhost:8080")

}

// 从上下文中解析并创建一个新的 trace，获得传播的 上下文(SpanContext)

func CreateTracer(serviceName string, header http.Header) (opentracing.Tracer,opentracing.SpanContext, io.Closer, error) {

var cfg = jaegercfg.Configuration{

ServiceName: serviceName,

Sampler: &jaegercfg.SamplerConfig{

Type: jaeger.*SamplerTypeConst*,

Param: 1,

},

Reporter: &jaegercfg.ReporterConfig{

LogSpans: true,

// 上报给agent

LocalAgentHostPort: "192.168.128.128:31300",

//上报给collector

//CollectorEndpoint: "http://192.168.128.128:31200/api/traces",

},

}

jLogger := jaegerlog.StdLogger

tracer, closer, err := cfg.NewTracer(

jaegercfg.Logger(jLogger),

)

// 继承别的进程传递过来的上下文

spanContext, \_ := tracer.Extract(opentracing.*HTTPHeaders*,

opentracing.HTTPHeadersCarrier(header))

return tracer, spanContext, closer, err

}

func UseOpenTracing() gin.HandlerFunc {

handler := func(c \*gin.Context) {

// 使用 opentracing.GlobalTracer() 获取全局 Tracer

tracer,spanContext, closer, \_ := CreateTracer("userInfoWebService", c.Request.Header)

defer closer.Close()

// 生成依赖关系，并新建一个 span、

// 这里很重要，因为生成了 References []SpanReference 依赖关系

startSpan:= tracer.StartSpan(c.Request.URL.Path,ext.RPCServerOption(spanContext))

defer startSpan.Finish()

// 记录 tag

// 记录请求 Url

ext.HTTPUrl.Set(startSpan, c.Request.URL.Path)

// Http Method

ext.HTTPMethod.Set(startSpan, c.Request.Method)

// 记录组件名称

ext.Component.Set(startSpan, "Gin-Http")

// 在 header 中加上当前进程的上下文信息

c.Request=c.Request.WithContext(opentracing.ContextWithSpan(c.Request.Context(),startSpan))

// 传递给下一个中间件

c.Next()

// 继续设置 tag

ext.HTTPStatusCode.Set(startSpan, uint16(c.Writer.Status()))

}

return handler

}

func GetUserInfo(ctx \*gin.Context) {

userName := ctx.Param("username")

fmt.Println("收到请求，用户名称为:", userName)

ctx.String(http.*StatusOK*, "他的博客是 https://whuanle.cn")

}

**总结：可以从Http头继承之前的Spin信息Extra到新的Span中完成链路的串联**

// 继承别的进程传递过来的上下文

spanContext, \_ := tracer.Extract(opentracing.*HTTPHeaders*,

opentracing.HTTPHeadersCarrier(header))

客户端完整代码如下：

package main

import (

"bufio"

"github.com/opentracing/opentracing-go"

"github.com/opentracing/opentracing-go/ext"

"github.com/opentracing/opentracing-go/log"

"github.com/uber/jaeger-client-go"

jaegercfg "github.com/uber/jaeger-client-go/config"

jaegerlog "github.com/uber/jaeger-client-go/log"

"io"

"net/http"

"os"

"time"

)

func CreateTracer(servieName string) (opentracing.Tracer, io.Closer, error) {

var cfg = jaegercfg.Configuration{

ServiceName: servieName,

Sampler: &jaegercfg.SamplerConfig{

Type: jaeger.*SamplerTypeConst*,

Param: 1,

},

Reporter: &jaegercfg.ReporterConfig{

LogSpans: true,

// 按实际情况替换你的 ip

//CollectorEndpoint: "http://192.168.129.128:31100/api/traces",

LocalAgentHostPort: "192.168.129.128:31300",

},

}

jLogger := jaegerlog.StdLogger

tracer, closer, err := cfg.NewTracer(

jaegercfg.Logger(jLogger),

)

return tracer, closer, err

}

func main() {

tracer, closer, \_ := CreateTracer("UserinfoService")

// 创建第一个 span A

parentSpan := tracer.StartSpan(

"clientmain",

opentracing.StartTime(time.Now()),

)

// 调用其它服务

GetUserInfo(tracer, parentSpan)

// 结束 A

parentSpan.Finish()

// 结束当前 tracer

closer.Close()

reader := bufio.NewReader(os.Stdin)

\_, \_ = reader.ReadByte()

}

// 请求远程服务，获得用户信息

func GetUserInfo(tracer opentracing.Tracer, parentSpan opentracing.Span) {

// 继承上下文关系，创建子 span

childSpan := tracer.StartSpan(

"GetUserInfoRequest",

opentracing.ChildOf(parentSpan.Context()),

)

url := "http://192.168.129.128:8081/Get?username=痴者工良"

req, \_ := http.NewRequest("GET", url, nil)

// 设置 tag，这个 tag 我们后面讲

ext.SpanKindRPCClient.Set(childSpan)

childSpan.SetTag("myself tag","tag123")

ext.SpanKind.Set( childSpan, "client")

ext.HTTPUrl.Set(childSpan, url)

ext.HTTPMethod.Set(childSpan, "GET")

childSpan.LogFields(log.String("event","a test event"))

tracer.Inject(childSpan.Context(), opentracing.*HTTPHeaders*, opentracing.HTTPHeadersCarrier(req.Header))

resp, \_ := http.DefaultClient.Do(req)

\_ = resp // 丢掉

ext.Error.Set(childSpan, true)

defer childSpan.Finish()

}

**总结：在tracer中Inject本次调用的Httpheader信息，以便在服务端可以Extract出来将调用链的Span串联在一起**

**在web框架的中间件中来实现对业务代码的低侵入性**

**通过SetTag和LogFields来对跟踪信息携带其他信息**

# 对于grpc.使用3方库来操作，调研测试了grpc-jaeger，grpc-jaeger 是 Go 实现的一种 gRPC 拦截器，它基于 opentracing 和 uber/jaeger。您可以使用它来构建分布式 gRPC 跟踪系统。它的内部不需要特殊处理就可以把跨进程的调用串联到一起

使用起来非常简单，使用时只需要在创建server的时候把创建的gtracer塞入到servOpts中就可以

完整代码如下：

package main

import (

"context"

"fmt"

"google.golang.org/grpc"

gtrace "github.com/moxiaomomo/grpc-jaeger"

"net"

pb "grpc2/proto"

"os"

)

const (

*port* = ":50051"

)

type server struct {

pb.UnimplementedGreeterServer

}

func (s \*server) SayHello(ctx context.Context, in \*pb.HelloRequest) (\*pb.HelloReply, error) {

fmt.Printf("Received: %v", in.GetName())

return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil

}

func main () {

var servOpts []grpc.ServerOption

tracer, \_, err := gtrace.NewJaegerTracer("testSrv", "192.168.129.130:6831")

if err != nil {

fmt.Printf("new tracer err: %+v\n", err)

os.Exit(-1)

}

if tracer != nil {

servOpts = append(servOpts, gtrace.ServerOption(tracer))

}

lis, err := net.Listen("tcp", *port*)

if err != nil {

fmt.Printf("failed to listen: %v", err)

}

s := grpc.NewServer( servOpts... )

pb.RegisterGreeterServer(s, &server{})

fmt.Printf("server listening at %v", lis.Addr())

if err := s.Serve(lis); err != nil {

fmt.Printf("failed to serve: %v", err)

}

}

# 客户端代码如下：

package main

import (

"context"

"fmt"

"google.golang.org/grpc"

pb "grpc2/proto"

"os"

"time"

gtrace "github.com/moxiaomomo/grpc-jaeger"

)

const(

*address* = ":50051"

*defaultName* = "word"

)

func main () {

dialOpts := []grpc.DialOption{grpc.WithInsecure(),grpc.WithBlock()}

tracer, \_, err := gtrace.NewJaegerTracer("testCli", "192.168.129.130:6831")

if err != nil {

fmt.Printf("new tracer err: %+v\n", err)

os.Exit(-1)

}

if tracer != nil {

dialOpts = append(dialOpts, gtrace.DialOption(tracer))

}

conn, err := grpc.Dial(*address*, dialOpts...)

if err != nil {

fmt.Printf("did not connect: %v", err)

}

defer conn.Close()

c := pb.NewGreeterClient(conn)

name := *defaultName*

if len(os.Args) > 1 {

name = os.Args[1]

}

ctx, cancel := context.WithTimeout(context.Background(), time.*Second*)

defer cancel()

r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})

if err != nil {

fmt.Printf("could not greet: %v", err)

}

fmt.Printf("Greeting: %s", r.GetMessage())

select{}

}

和服务端代码类似。也是创建一个JagerTrace,然后塞入dialOpts 选项参数数组。然后传入grpc.Dial

1. 关键源码摘要：

1.TraceID定义

type TraceID struct {

High, Low uint64

}

2.TraceID的赋值：

在StartSpan的时候，判断是否有父Parent.如果没有就给TraceID随机生成值并赋值：

如果有ParnetSpan,那么tracingid从父Span那里继承过来.

if !hasParent || !parent.IsValid() {

newTrace = true

ctx.traceID.Low = t.randomID()

if t.options.gen128Bit {

ctx.traceID.High = t.options.highTraceIDGenerator()

}

ctx.spanID = SpanID(ctx.traceID.Low)

ctx.parentID = 0

ctx.samplingState = &samplingState{

localRootSpan: ctx.spanID,

}

if hasParent && parent.isDebugIDContainerOnly() && t.isDebugAllowed(operationName) {

ctx.samplingState.setDebugAndSampled()

internalTags = append(internalTags, Tag{key: *JaegerDebugHeader*, value: parent.debugID})

}

} else {

ctx.traceID = parent.traceID

if rpcServer && t.options.zipkinSharedRPCSpan {

// Support Zipkin's one-span-per-RPC model

ctx.spanID = parent.spanID

ctx.parentID = parent.parentID

} else {

ctx.spanID = SpanID(t.randomID())

ctx.parentID = parent.spanID

}

ctx.samplingState = parent.samplingState

if parent.remote {

ctx.samplingState.setFinal()

ctx.samplingState.localRootSpan = ctx.spanID

}

}

对于HttpSrv，首先从HttpHead里面解压HttpHead,然后根据，解压的数据得到

父Span.在根据父Spin生成新的Span,新Span在Start的时候会给tracingid赋值，这样客户端和服务器的Span就是相同的tracingid。

spanContext, \_ := tracer.Extract(opentracing.*HTTPHeaders*,

opentracing.HTTPHeadersCarrier(header))

startSpan:= tracer.StartSpan(c.Request.URL.Path,ext.RPCServerOption(spanContext))

1. PHP的代码对接

github地址:

<https://github.com/jukylin/jaeger-php#install>

Install:

composer config minimum-stability dev

composer require jukylin/jaeger-php

初始化一个tracer,example是servername. '0.0.0.0:6831'是agent的地址

$config = Config::getInstance();

$tracer = $config->initTracer('example', '0.0.0.0:6831');

$config->gen128bit();

从之前的Span继承信息以完成rpc的tracing串联:

$spanContext = $tracer->extract(Formats\TEXT\_MAP, $\_SERVER);

Start Span

$serverSpan = $tracer->startSpan('example HTTP', ['child\_of' => $spanContext]);

把本次rpc调用的信息注入，以便下游的srv能extract

clientTrace->inject($clientSpan1->spanContext, Formats\TEXT\_MAP, $\_SERVER);

设置Tag 和 Log

$span->setTag('http.status', '200');

span->log(['error' => 'HTTP request timeout']);

//最后将span上报

$span->finish();

$config->flush();