Opentracing

contents

1. Why ?

1.1 场景

老板 A 总 安排 小 B 开发游戏币下单业务，按照接口设计 小 B 只需要对前端请求做验证、转换，然后调用一个 小 C 提供的下单 API 即可，无需关心后续具体的流程。但是在调试测试过程中， 小 C 的下单接口频频返回 internal error 500 。

小 C 不得不查看日志，然后说是调用在 小 D 写的一个服务的时候发生了 500 错误。。。就这样， 一下午过去了，小 B 联系了 24 个人，才发现是 小 Z 的写的一个 slice 切片操作没有进行长度判断，返回 500.

1.1 场景

一天，玩家 Pony 血槽马上要空了，急需购入药箱，但游戏频频提示药箱正在路上，请您耐心等待。

这晚无数个像 Pony 一样的玩家，打爆了 A 总的电话，而 A 总不得不紧急拨打 25 个电话，叫醒小 B-Z ，后面的事情可想而知，小 B-Z 失业了。

1.1 场景

后来， Pony 立志让天下没有难买的药箱，于是他收购了 A 总的公司，召回小 B-Z ，发现之前的错误是因为小 X 忘记做余额不足的判断，而小 Y 将账户 余额的类型设置为了 uint64 ， 引发了 500 错误。

Pony 调研后，决定引入 OpenTrace 和监控系统，当类似的问题发生时，系统自动发现 500 错误是小 Z 服务引起的，并自动拨通了 小 Z 的电话。

1.1 场景

游戏重新上线一段时间后， Pony 喜忧参半，喜的是系统稳定，买药箱的玩家越来越多。但是系统的响应速度却越来越慢，想优化却无从入手。

虽说日志记录的非常详细， 但确是分散的，有的服务使用了 RequestID 对请求进行标记，只要在各个服务的日志中，找到 RequestID 对应的日志，最后将他们整合到一起，如果日志中恰好 记录有时间，就可以进行运行时长问题的研究了。但随着服务和机器的增多，可操作行越来越差。

其实，这个问题利用 OpenTracing 就可以解决。

1.2 小结

微服务架构多应用，多实例化后需要程序面临的一些问题：

- 我是谁？
- 我从哪里来？
- 我要到那里去？

1.2 小结

openTracing 可以回答一个程序的哲学三问 , 基于此就可以很清晰的解决下面三个难题 :

- 错误原因快速定位
- 用户体验优化 (响应时长)
- 架构 (调用链路) 优化

2. What ?

2.1 OpenTracing API

OpenTracing API 理念源于 google 的 dapper 论文，制定了解决分布式系统服务追踪的标准接口。

OpenTracing 组织提供了各个语言平台的 Opentracing API 接口实现，同时提供了大量的辅助类库。

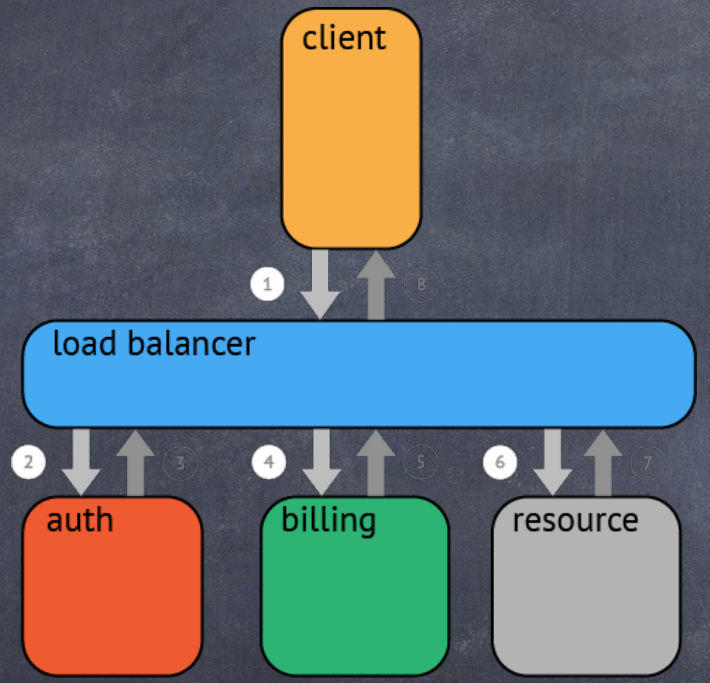
https://github.com/opentracing
https://github.com/opentracing-contrib
https://opentracing.io/registry/?s=go

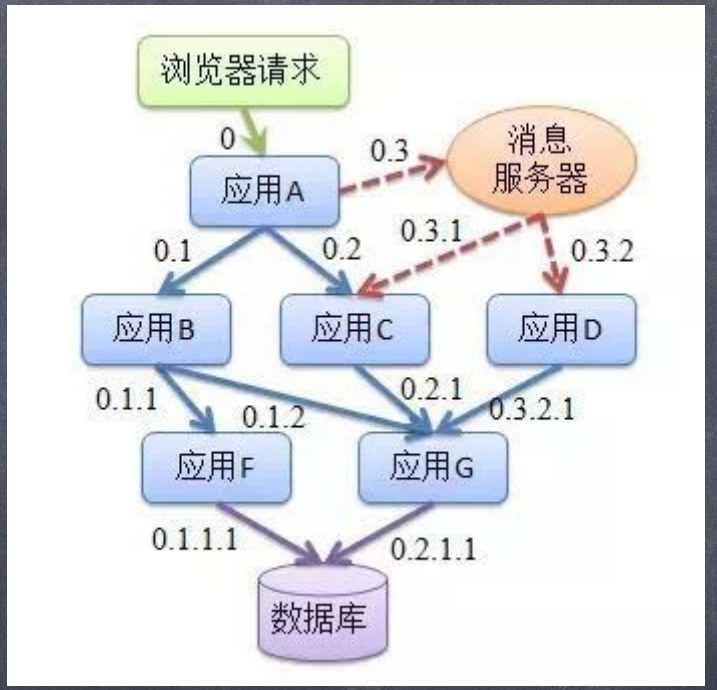
2.2 Traces (跟踪，追踪)

大家可能听过有人听过一个名词：分布式链路追踪，但如果咬文嚼字的话，其实“链 (link)” 这个字并不贴切，链的形状是 A->B->C ，显然我们的写的程序并不都是这样的，标准的说，是一个有向无环图。

2.2 Traces (跟踪，追踪)

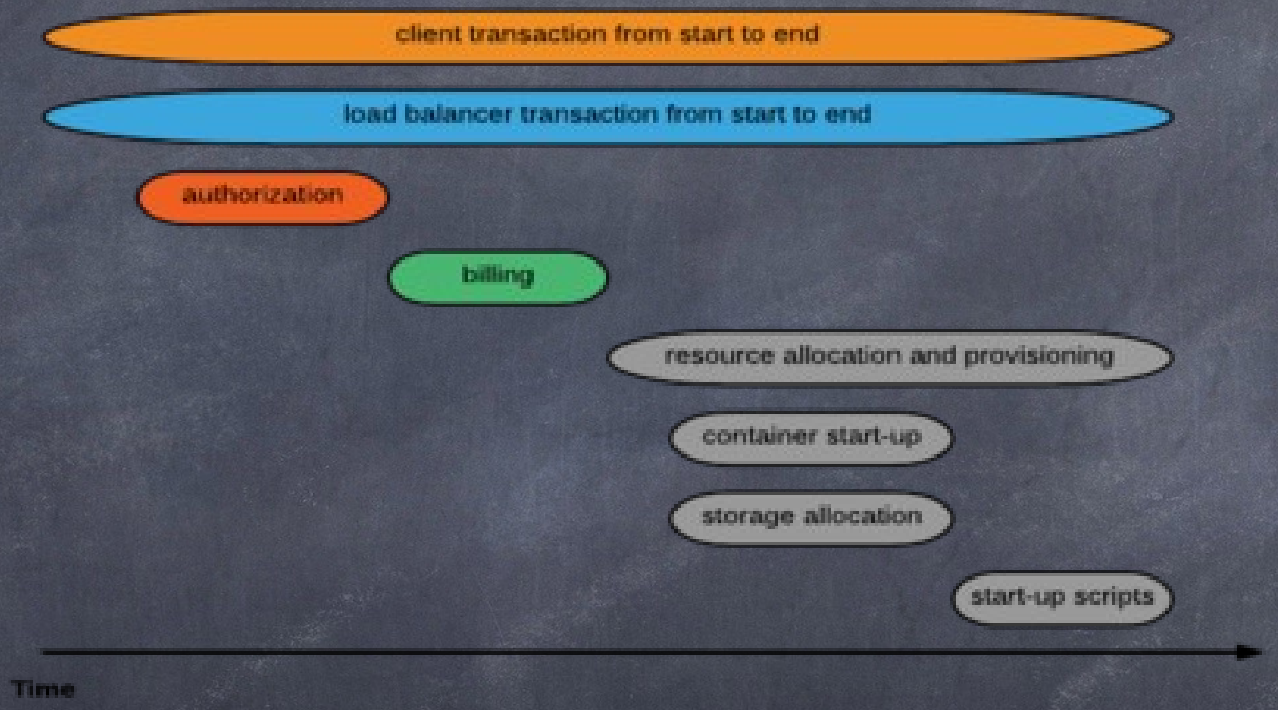
流程图表示法：





2.2 Traces (跟踪，追踪)

时序图表示法：



2.2 Traces (跟踪，追踪)

流程图 or 时序图?

- 流程图易于看组件间的调用关系，但不方便看调用时间，有局限性
- 时序图方便看执行时间，方便表示串行、并行关系。
- Trace 一般使用时序图展示。

2.3 Spans

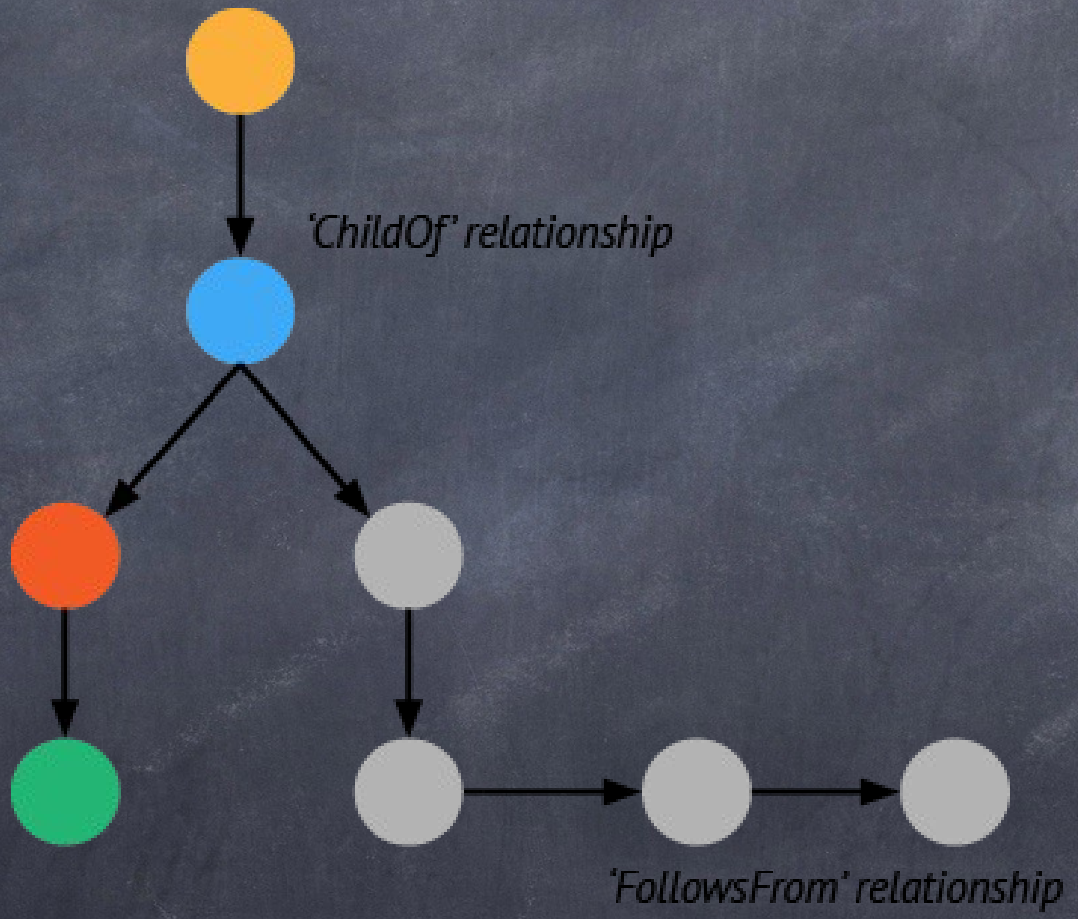
一个 `span` 代表一个逻辑运行单元。 OpenTracing 推荐在 RPC 的客户端和服务端，至少各有一个 span, 用于记录 RPC 调用的客户端和服务端信息。

三要素：

- 操作名称 （`get` or `get_user/999` or `get_user`）
- 开始时间
- 结束时间

2.3 Spans

span 引用关系：



2.3 Spans

ChildOf ：
- - 嵌套关系
- - 父 span 的运行时间，取决于子 span 的运行时间
- - 父 span 可以有多个并行运行的子 span

FollowsFrom ：
- 顺序排列关系
- A 、 B span 只能串行运行， B span 必须在 A span 执行完后执行
- B 独立与 A ，比如在异步进程中运行
- B 运行时间与 A 运行时间无关，与 A 的父 span 的运行时间无关。

2.4 Logs

包含时间戳的日志， OpenTracing 规范建议所有的日志声明都包含 event 字段，用于描述记录整个事件，事件提供的其他属性可以作为额外的字段记录。

2.5 SpanContext

- span 的上下文 , 跨越整个 tracing 周期 , 主要用于在 RPC 请求中传递 span
- 携带 trace_id, span_id, sample 等信息 .
- 可以用 Baggage 在整个 trace 中携带信息 .

2.6 Tags & Baggage

Tags：

- key-value 键值对 , 记录关于 span 的信息 .
- 通用语义约定 : https://opentracing.io/specification/conventions/

Baggage：

- Baggage 是存储在 SpanContext 上的键值对 , 可以在一个 trace 中的所有 span 中传递
- 要非常克制的在 Baggage 中存数据
- 可以同于传递顶层调用者的身份信息 , 或是一些一直存在请求参数中的信息 , 比如 wuid, meid

Baggage vs. Span Tags：

- Baggage 在一个 trace 周期存储
- Tags 只在一个 span 周期存储

2.7 Inject & Extract （非常重要）

Carrier 跨进程数据携带者：

- text map （基于字符串的 map ）
- binary （二进制 ）

Inject & Extract：

SpanContexts 可以通过 Injected 操作向 Carrier 增加，或者通 Extracted 从 Carrier 中获取，跨进程通讯数据 , 实现全链路追踪 .

2.8 self talk

一开始接触这个东西，我觉得制定 log 规范都很难让每个人遵守，跟何况是 trace 呢， trace 不仅要考虑代码段要不要记录成一个 span ， 还要考虑要记录的细致程度 , 如果在一个 trace 中使用 baggage 传递信息，那更是需要我们去和符合上下游的同学进行联调才行。

但是转念一想，这都不是事儿 . 关于 span 的粒度问题 , 官方早有建议 (Focus on Areas of Value), 总的来说 , 就是由粗到细 .trace 具有严格的规范性，如果在链路的某一层丢失了 span 上下文，那么链路追踪就会断开，一次调用就会产生多个 trace ，这可以很直观的在 UI 中体现 , 这是什么，这就是 bug ，这就必须得改。

2.8 self talk

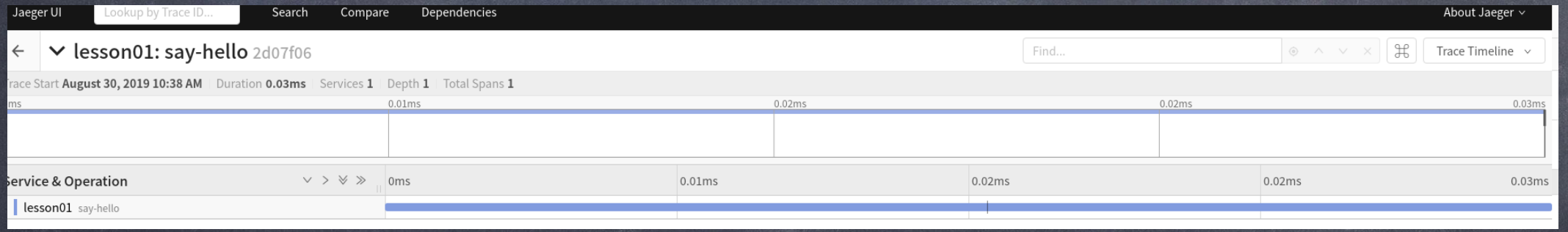
但不管再怎么做规范，始终是对代码有侵入性，每个服务的代码都要在一种约束下完成，都要为了实现追踪监控写代码，这并不是一宗优雅的实现方式。

后面的演示中，会先介绍这种传统的代码实现的方式，再介绍一种无需写代码的方式。

3. How ?

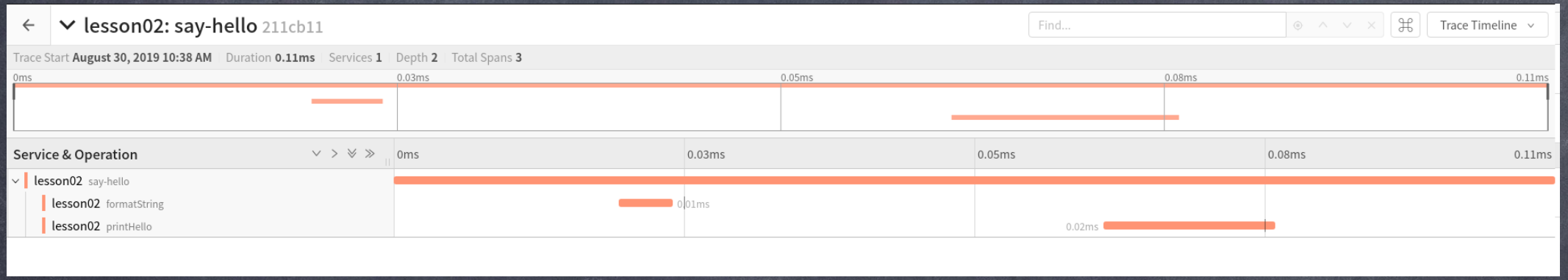
3.1 HelloWorld（本地查看代码并运行演示）

1. 简单示例



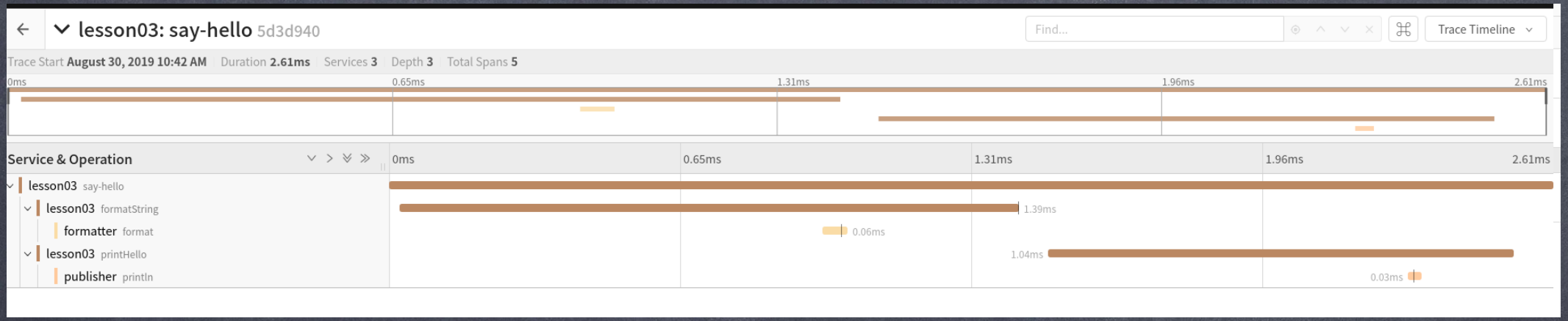
3.1 HelloWorld (本地查看代码并运行演示)

2. 多个函数示例



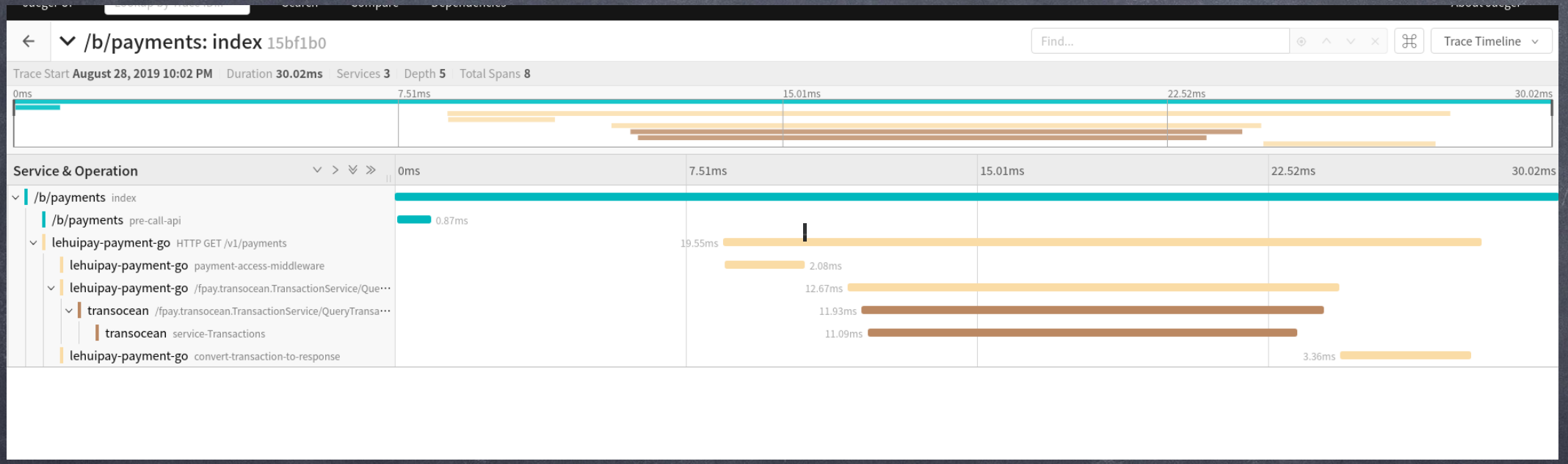
3.1 HelloWorld（本地查看代码并运行演示）

3. 多个服务示例



3.2 Payments 本地查看代码 dev 环境运行演示

Jaeger UI 效果：



3.2 Payments 本地查看代码 dev 环境运行演示

PHP 端的改动：

```php
// 初始化追踪器
$config = Config::getInstance();
$tracer = $config->initTracer( serverName: '/b/payments',  agentHostPort: '127.0.0.1:6831');
Log::info(json_encode($tracer));
$span = $tracer->startSpan( operationName: 'index');
// 重要操作,将 spanContext 注入到 header 中,这样在 go http 服务中,就可以用 extract 提取出出 spanContext
$injectHeaders = [];
$tracer->inject($span->spanContext,  format: Formats\TEXT_MAP,  &carrier: $injectHeaders);

$childSpan = $tracer->startSpan( operationName: 'pre-call-api', ['child_of' => $span->spanContext]);
```

```php
    // span 一定要finish
    $span->finish();
    $config->flush();
    $config->setDisabled( disabled: true);
    return response()->json($data);
}
```

3.2 Payments 本地查看代码 dev 环境运行演示

lehuipay-payment-go 端的改动：

查看代码:Tracer 初始化

```go
func main() {
    // 初始化追踪器,每个微服务都需要初始化一个追踪器,并传入 service_name
    tracer, closer := tracing.Init( service: "lehuipay-payment-go")
    defer closer.Close()
    opentracing.SetGlobalTracer(tracer)
    cmd.Execute()
}
```

查看代码:HTTP 中间件

```go
    groupV1.GET(
        path: "/payments",
        paymentController.PaymentListAction,
        // 在中间件中解析出下游通过 header 发来的 trace 信息,如果没有就新建一个 trace
        trace.TracerMiddleware(),
        middlewares.NewPaymentAccessMiddleware(bootstrap.AccessClient),
    )
```

path string, h HandlerFunc, m ...MiddlewareFu

3.2 Payments 本地查看代码 dev 环境运行演示

lehuipay-payment-go 端的改动 :

```
func NewPaymentAccessMiddleware(accessClient pb.AccessControlServiceClient) gopress.MiddlewareFunc {
    return func(next gopress.HandlerFunc) gopress.HandlerFunc {
        return func(ctx gopress.Context) error {
            // payment-access中间件负责权限验证,可以认为是一个 span. 所以需要从上下文中取出父span,并开启自己的 span
            span := trace.ExtractSpan(ctx)
            span = span.Tracer().StartSpan( operationName: "payment-access-middleware", opentracing.ChildOf(span.Context()))
            qp := new(payment.QueryParams)
            logger := logging.Extract(ctx)
            err := ctx.Bind(qp)
```

3.2 Payments 本地查看代码 dev 环境运行演示

lehuipay-payment-go 端的改动：

```go
// 在创建 grpc 客户端的时候,要传入 tracing 的拦截器
func newTransoceanClient(opts tc.Options) (*Client, error) {
    if opts.DialTimeout == 0 {...}
    if opts.KeepAliveTime == 0 {...}
    if opts.KeepAliveTimeout == 0 {...}

    ctx, cancel := context.WithTimeout(context.Background(), time.Duration(opts.DialTimeout)*time.Second)
    defer cancel()

    conn, err := grpc.DialContext(ctx, opts.Address, []grpc.DialOption{
        grpc.WithInsecure(),
        grpc.WithUnaryInterceptor(otgrpc.OpenTracingClientInterceptor(opentracing.GlobalTracer())),
        grpc.WithKeepaliveParams(keepalive.ClientParameters{...}),
    }...)
    if err != nil {...}

    return &Client{...}, nil
}
```

3.2 Payments 本地查看代码 dev 环境运行演示

lehuipay-payment-go 端的改动 :

```
filter := transocean.QueryTransactionsRequest{...}

// 在调用 grpc 服务时,要将当前的 span 存入ctx
transactions, err := s.Transocean.QueryTransactions(opentracing.ContextWithSpan(ctx, span), &filter)

if err == nil && transactions != nil {
    subSpan := span.Tracer().StartSpan( operationName: "convert-transaction-to-response", opentracing.ChildO
```

3.2 Payments 本地查看代码 dev 环境运行演示

transocean 端的改动：

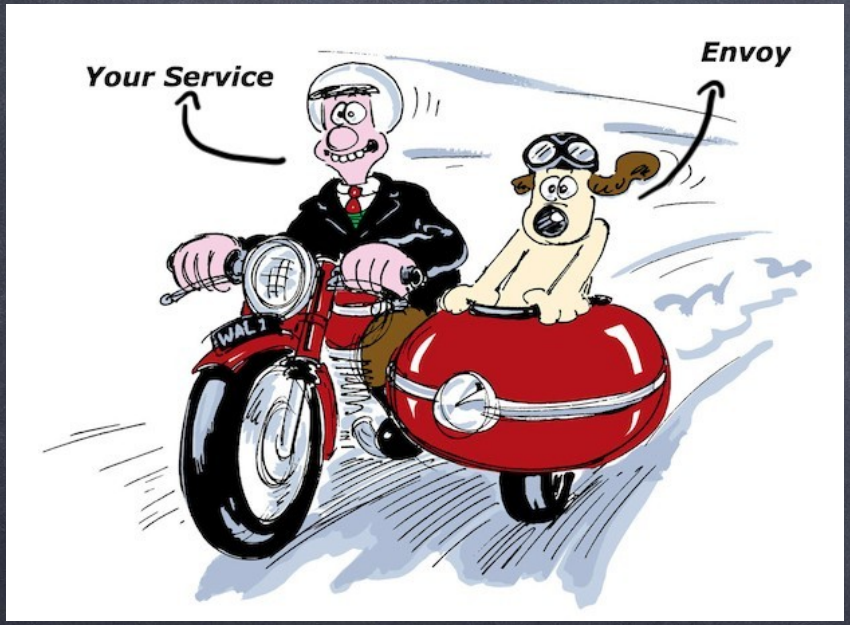
```
        gs := grpc.NewServer(
            grpc.KeepaliveParams(keepalive.ServerParameters{
                Time: 10 * time.Minute,
            }),
            grpc_middleware.WithUnaryServerChain(
                // 在 grpc 的服务端也要传入 tracing 的拦截器
                otgrpc.OpenTracingServerInterceptor(opentracing.GlobalTracer()),
                grpc_ctxtags.UnaryServerInterceptor(
                    grpc_ctxtags.WithFieldExtractor(grpc_ctxtags.CodeGenRequestFieldExtractor),
                    grpc_ctxtags.WithFieldExtractor(func(fullMethod string, req interface{}) map[string
                        fields := map[string]interface{}{"request_id": xid.New().String()}
                        return fields
                    }),
                ),
```

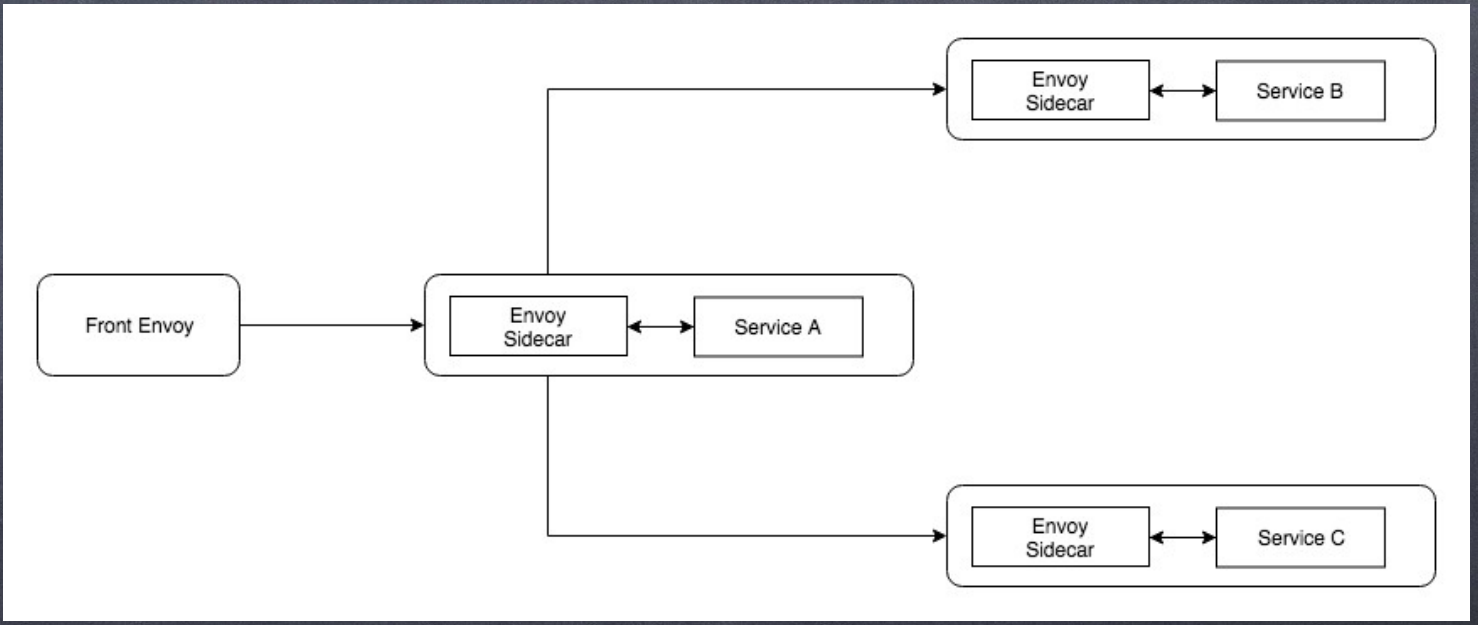
```
// Transactions 批量获取订单列表
func (ts *TransactionService) Transactions(ctx context.Context, r *api.QueryTransactionsRequest) (*api.C
    // 因为在 grpc client 端将 span 存入到了上下文,所以在服务端,可以直接从上下文中取出 span
    span, _ := opentracing.StartSpanFromContext(
        ctx,
        operationName: "service-Transactions",
    )
    defer span.Finish()
```

3.3 Envoy Tracing / istio Tracing

通过前面的演示 , 我们可以发现 , 传统的方式 OpenTracing 对代码的侵入性非常大 , trace,span,spanContext,tags,logs 随处可见 , 想在整个后端开发中推行非常困难 .

服务网格的流行为 Tracing 带来了新的方式 , 首先了解一下 sidecar 模式 :





3.3 Envoy Tracing / istio Tracing

Envoy 的追踪其实并不神秘，它只是帮我们完成的 span 的上报。

在整个调用链中，它其实还是会以一个 x-request-id 来对服务进行追踪。

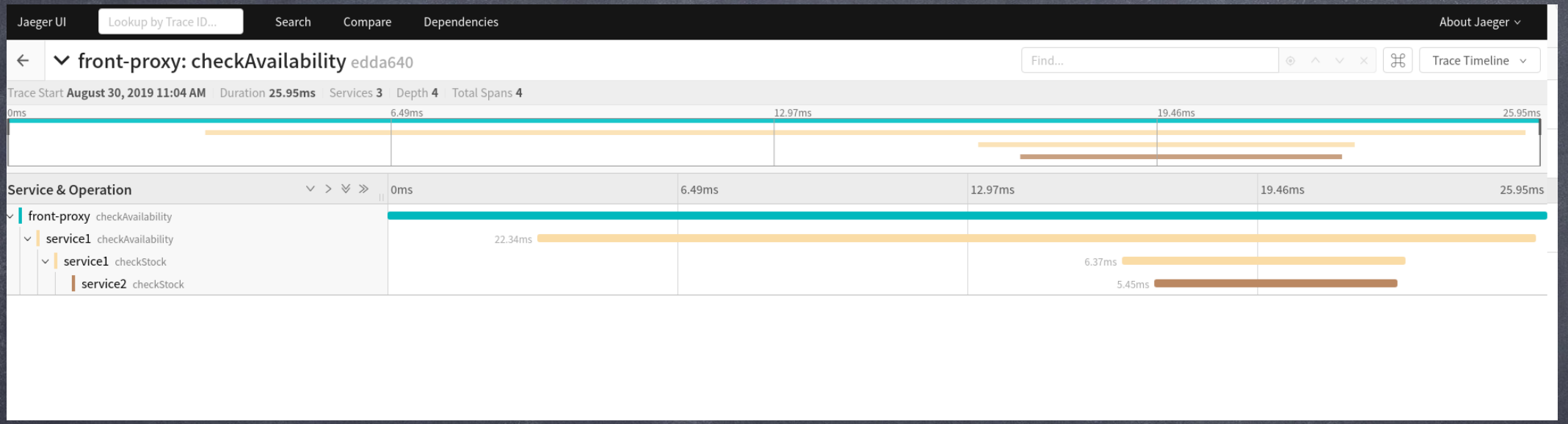
所以，即使是 Envoy 也比不能完全保证不对代码有侵入性，在服务 A 中调用 B 服务，需要把 A 服务接收到的 x-request-id 传给 B 服务的请求头。

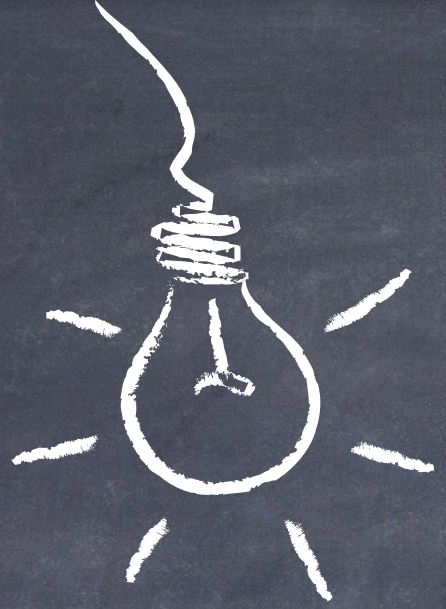
下面是官方 sandbox 追踪示例中的一段代码：
https://github.com/envoyproxy/envoy/blob/64243c97628369ceb365b4da6d73b43dd8bccba5/examples/front-proxy/service.py#L38

同理，istio 是基于 Envoy 的，所以在 istio 上实现的追踪，也需要在服务中传递一些请求头。https://istio.io/zh/docs/tasks/telemetry/distributed-tracing/overview/

3.3 Envoy Tracing / istio Tracing

官方示例演示：





谢谢您的聆听