

分布式追踪中间件开发

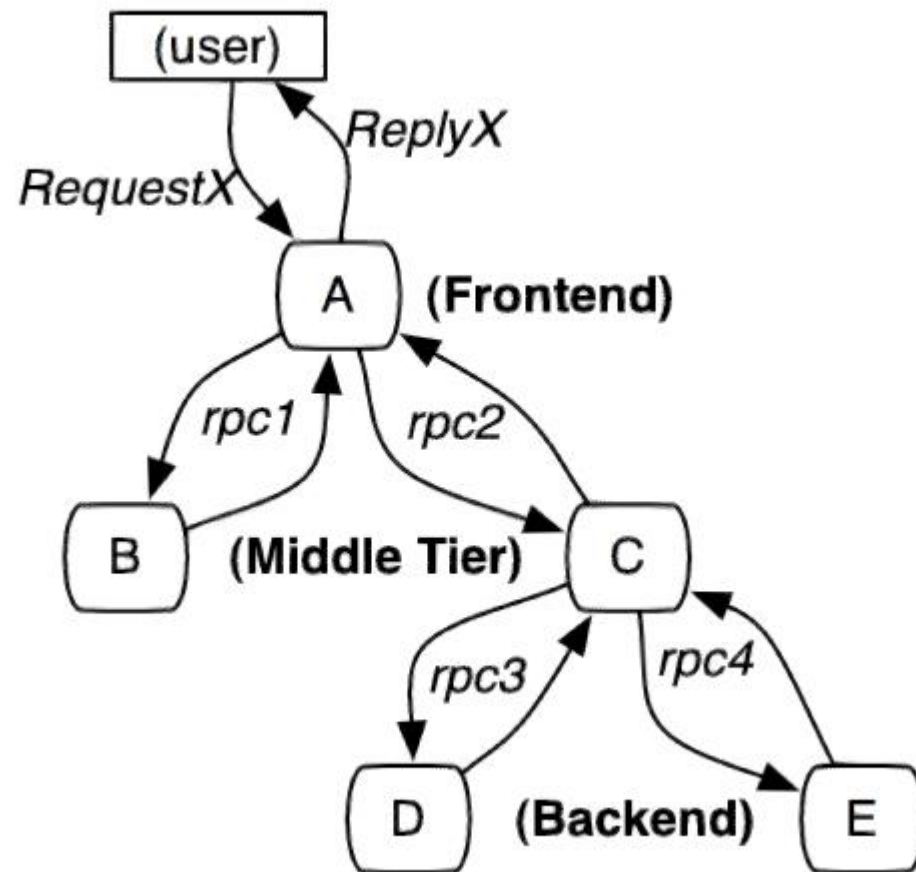
联系QQ: [2816010068](#), 加入会员群

目录

- 背景
- 思路
- Opentrace实战
- 分布式中间件开发

背景

- 调用树示例

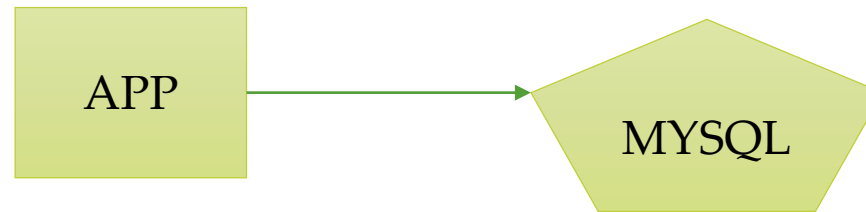


背景

- 面临的问题或痛点
 - 如果user服务某个请求特别慢, 如何去定位?
 - 如果user服务某个请求频繁报错了, 如何定位?
 - 1年以后, user服务的调用依赖图, 还有人能够快速的给出吗?

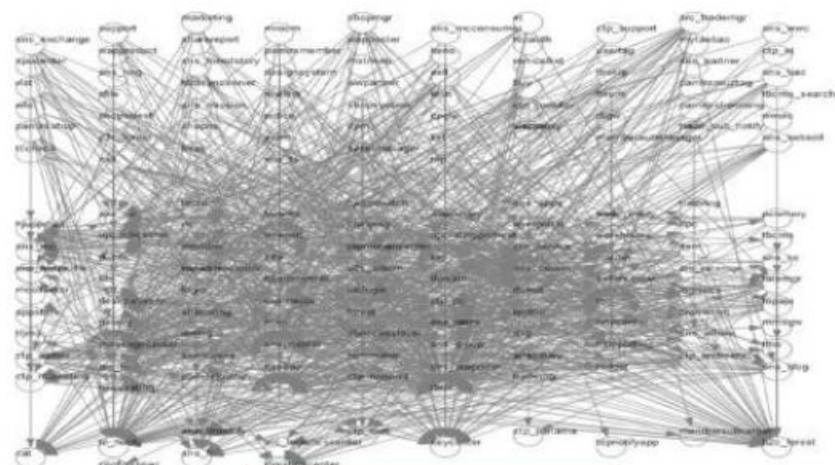
背景

- 单体应用
 - 绝大多数可以通过日志进行定位



微服务架构

- 故障定位难
- 容量预估难
- 资源浪费多
- 链路梳理难

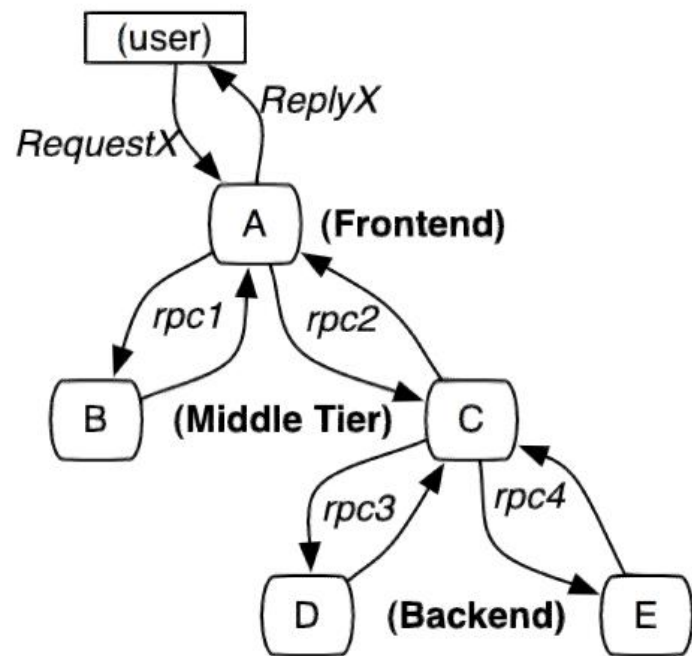


2012 淘宝核心链路应用拓扑图

故障定位难
容量预估难
资源浪费多
链路梳理难

分布式追踪如何解决这个问题？

- trace_id概念
 - 为每个请求分配唯一的id, 通常叫做trace_id
 - 日志聚合

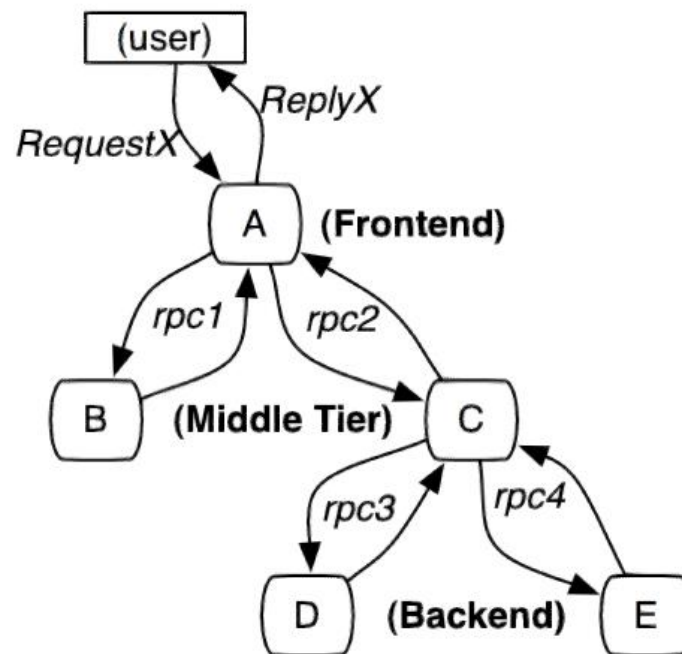


分布式追踪如何解决这个问题？

- Span概念
 - 如何知道每个子系统的详细的处理细节？
 - 通过span进行抽象

整个请求耗时3s

user request span



分布式追踪如何解决这个问题？

- Span概念
 - 如何知道每个子系统的详细的处理细节？
 - 通过span进行抽象，Span之间有父子的关系

整个请求耗时3s

user request span 3s

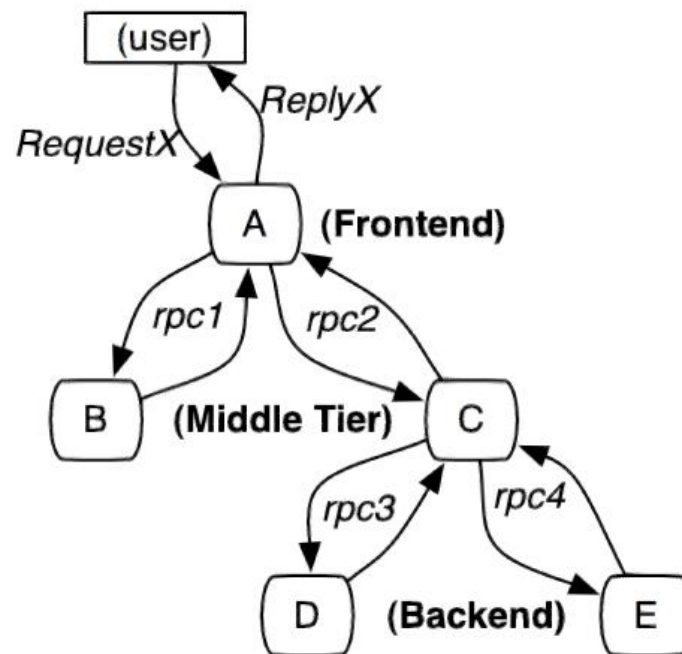
A request span 2.8s


B request span 0.8s


C request span 2s

C 0.5s

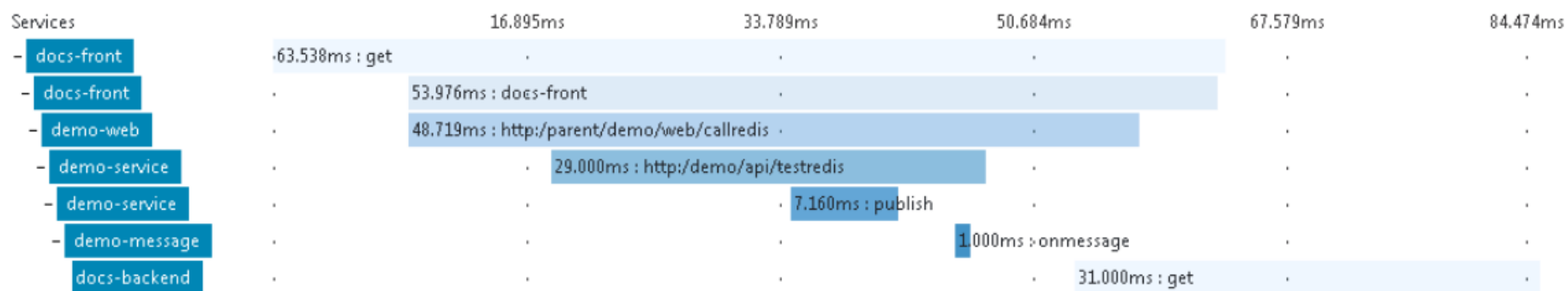
D 1.5s



Duration: 84.474ms Services: 5 Depth: 7 Total Spans: 7 [JSON](#) 

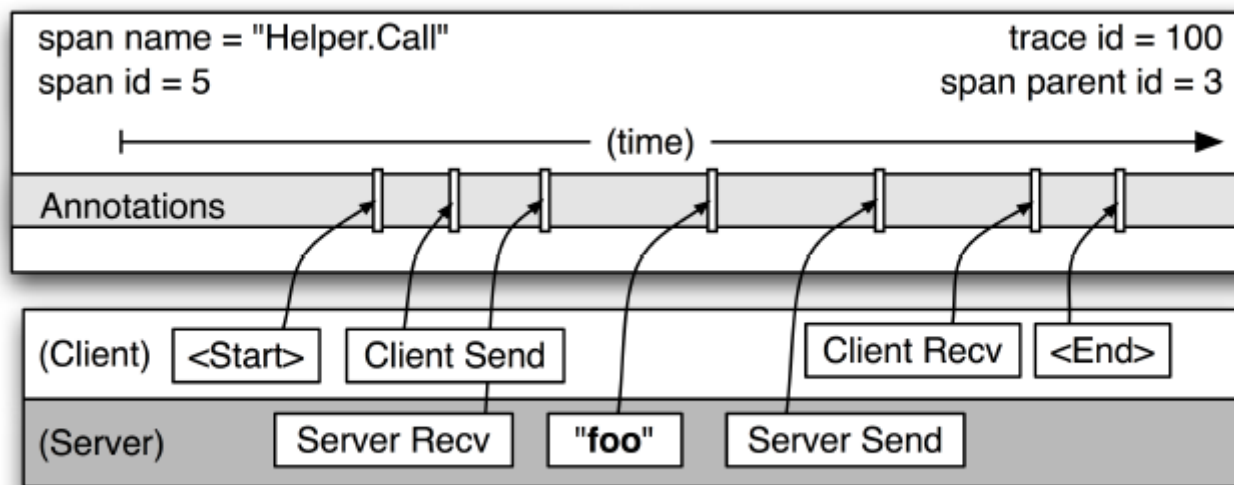
[Expand All](#) [Collapse All](#) 

[demo-message x2](#) [demo-service x2](#) [demo-web x2](#) [docs-backend x1](#) [docs-front x3](#)



分布式追踪如何解决这个问题？

- 单个span详细解剖



分布式追踪如何解决这个问题？

- Span Context概念
 - 传播问题
 - 进程内传播
 - 进程之间传播
 - http协议
 - 通过http头部进行透明传播
 - Tcp协议
 - Thrift
 - 需要改造thrift进行支持

开源实现

- Google
 - Dapper论文
 - <https://bigbully.github.io/Dapper-translation/>
- Uber
 - [Jaeger已经开源](#)
 - <https://www.jaegertracing.io/>
- Twitter
 - Zipkin
 - <https://github.com/openzipkin/zipkin>
- Opentracing, 标准化组织
 - Opentracing
 - <https://github.com/opentracing/opentracing-go>

Span context其他用途

- 测试标签

分布式trace实战

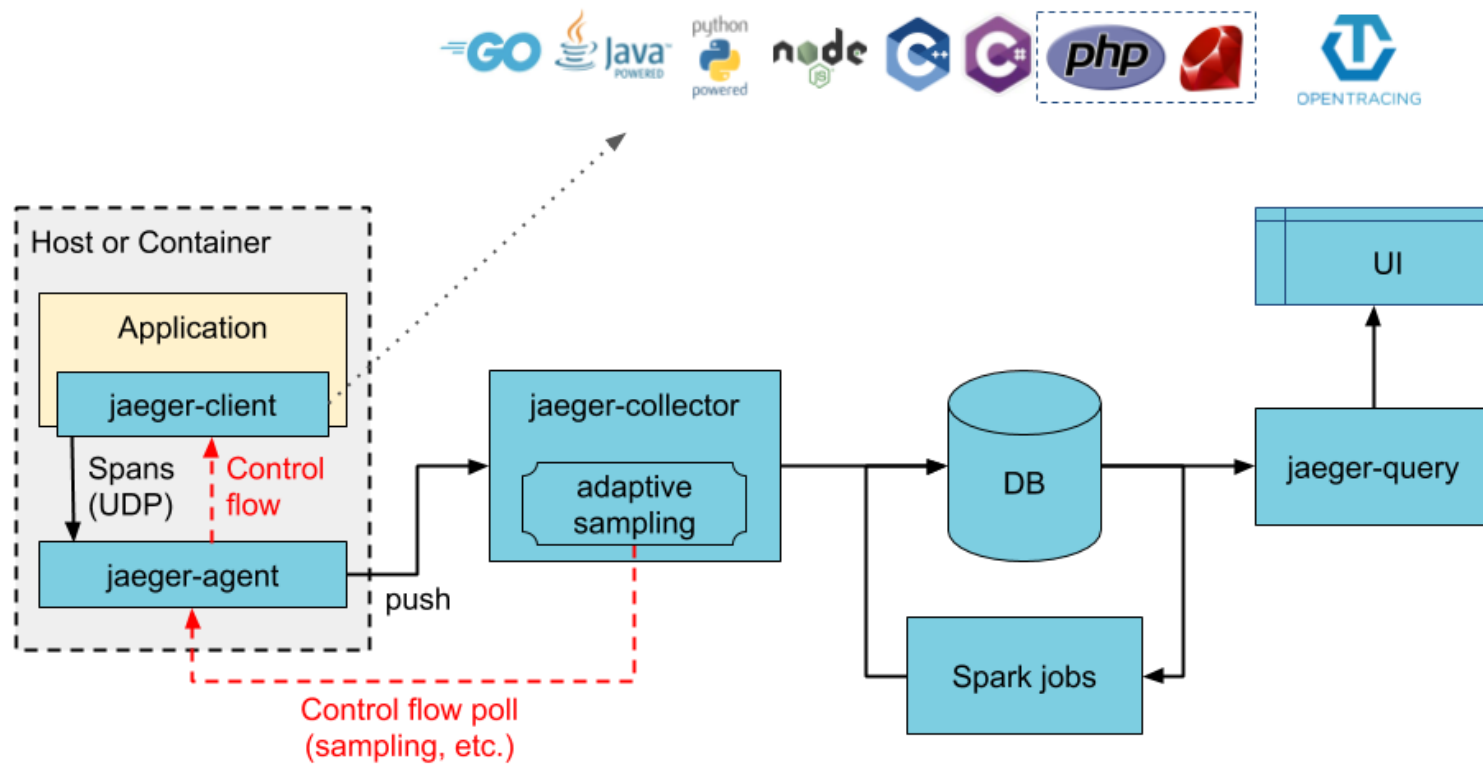
技术选型

- 使用opentracing提供的通用接口
- 底层jagger作为分布式系统

环境准备

- Jagger部署
 - 生产环境:
 - <https://juejin.im/post/5cc6c23ae51d456e660d4542>
 - 开发环境: 使用docker镜像
 - Docker安装: <https://qizhanming.com/blog/2019/01/25/how-to-install-docker-ce-on-centos-7>
 - 测试环境: <http://60.205.218.189:9411/api/v1/spans>

Jeagger架构



实战一

- Hello world
 - 代码: `koala/example/trace_example/hello`
 - 演示jeagger的基本使用

实战二

- 追踪函数
 - 追踪指定函数的执行情况
 - span和context进行结合，并在进程内进行传播
 - 代码：`koala/example/trace_example/function`

实战三

- 追踪网络调用
 - span在网络请求中进行传递

```
ext.SpanKindRPCClient.Set(span)
ext.HTTPUrl.Set(span, url)
ext.HTTPMethod.Set(span, "GET")
span.Tracer().Inject(
    span.Context(),
    opentracing.HTTPHeaders,
    opentracing.HTTPHeadersCarrier(req.Header),
)
```

```
http.HandleFunc("/format", func(w http.ResponseWriter, r *http.Request) {
    spanCtx, _ := tracer.Extract(opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(r.Header))
    span := tracer.StartSpan("format", ext.RPCServerOption(spanCtx))
    defer span.Finish()
    // 从http头部提取出传输的span

    helloTo := r.FormValue("helloTo")
    helloStr := fmt.Sprintf("Hello, %s!", helloTo)
    span.LogFields(
        otlog.String("event", "string-format"),
        otlog.String("value", helloStr)
    )
})
```

grpc metadata介绍

- grpc是基于http2.0的rpc框架
- 那么如何传递一些用户自定义的数据呢?比如trace_id, span_id等等
 - 通过http的头部进行传递
- grpc对于http头部传递数据进行了封装
 - metadata, 单独抽象了一个包
 - `google.golang.org/grpc/metadata`
 - `type MD map[string][]string` 其实就是一个map

grpc metadata介绍

- 客户端添加metadata

- 构造一个metadata

- md := metadata.New(map[string]string{"key1": "val1", "key2": "val2"})
 - 使用Pair函数

```
md := metadata.Pairs(  
    "key1", "val1",  
    "key1", "val1-2", // "key1" will have map value []string{"val1", "val1-2"}  
    "key2", "val2",  
)
```

- 客户端发送metadata

```
// create a new context with some metadata  
md := metadata.Pairs("k1", "v1", "k1", "v2", "k2", "v3")  
ctx := metadata.NewOutgoingContext(context.Background(), md)  
  
// later, add some more metadata to the context (e.g. in an interceptor)  
md, _ := metadata.FromOutgoingContext(ctx)  
newMD := metadata.Pairs("k3", "v3")  
ctx = metadata.NewContext(ctx, metadata.Join(metadata.New(send), newMD))  
  
// make unary RPC  
response, err := client.SomeRPC(ctx, someRequest)  
  
// or make streaming RPC  
stream, err := client.SomeStreamingRPC(ctx)
```

grpc metadata介绍

- 客户端发送方式二

```
// create a new context with some metadata
ctx := metadata.AppendToOutgoingContext(ctx, "k1", "v1", "k1", "v2", "k2", "v3")

// later, add some more metadata to the context (e.g. in an interceptor)
ctx := metadata.AppendToOutgoingContext(ctx, "k3", "v4")

// make unary RPC
response, err := client.SomeRPC(ctx, someRequest)

// or make streaming RPC
stream, err := client.SomeStreamingRPC(ctx)
```


grpc metadata介绍

- 服务端接收metadata

```
func (s *server) SomeRPC(ctx context.Context, in *pb.someRequest) (*pb.someResponse, error) {  
    md, ok := metadata.FromIncomingContext(ctx)  
    // do something with metadata  
}
```

- 服务端发送metadata

```
func (s *server) SomeRPC(ctx context.Context, in *pb.someRequest) (*pb.someResponse, error) {  
    // create and send header  
    header := metadata.Pairs("header-key", "val")  
    grpc.SendHeader(ctx, header)  
    // create and set trailer  
    trailer := metadata.Pairs("trailer-key", "val")  
    grpc.SetTrailer(ctx, trailer)  
}
```

koala分布式中间件开发

- 配置项开发
- 中间件开发