PHP 全链路监控接入方案

本篇着重讲述两方面内容

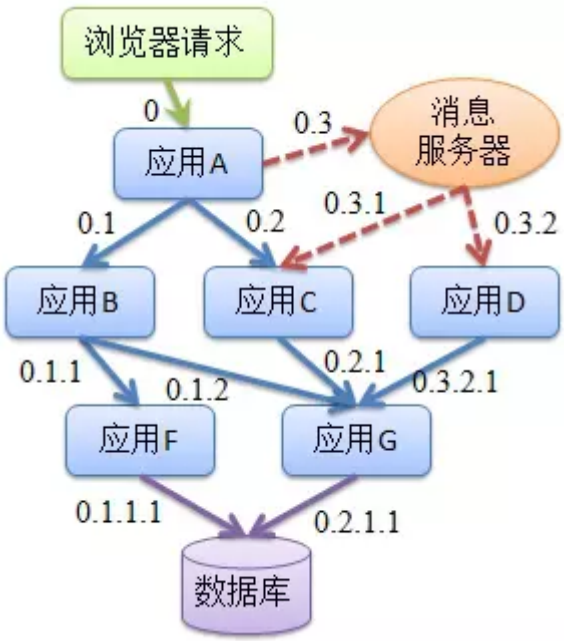
1. 全链路监控的相关阐述.
2. PHP的全链路监控的接入方案.

一. 为啥需要全链路

随着微服务架构的流行，**服务按照不同的维度进行拆分**，一次请求往往需要涉及到多个服务。**互联网应用构建在不同的软件模块集上**，这些软件模块，**有可能是由不同的团队开发、可能使用不同的编程语言来实现、有可能布在了几千台服务器，横跨多个不同的数据中心**。因此，就需要一些可以帮助理解系统行为、用于分析性能问题的工具，以便发生故障的时候，能够快速定位和解决问题。

全链路监控组件就在这样的问题背景下产生了。最出名的是谷歌公开的论文提到的 Google Dapper。**想要在这个上下文中理解分布式系统的行为，就需要监控那些横跨了不同的应用、不同的服务器之间的关联动作**。

所以，**在复杂的微服务架构系统中，几乎每一个前端请求都会形成一个复杂的分布式服务调用链路**。一个请求完整调用链可能如下图所示:



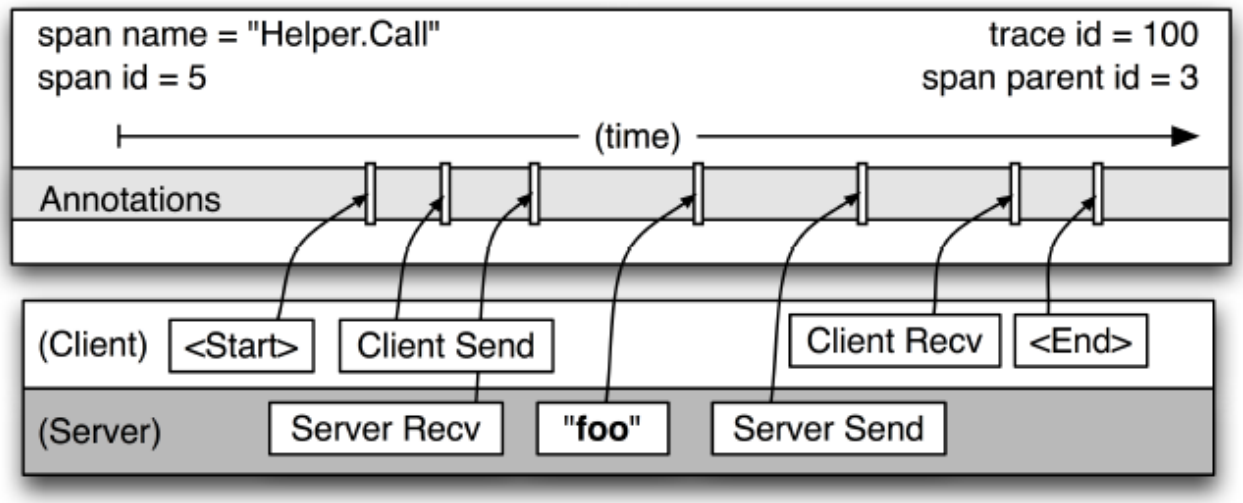
全链路性能监控 **从整体维度到局部维度展示各项指标**，将跨应用的所有调用链性能信息集中展现，可方便度量整体和局部性能，并且方便找到故障产生的源头，生产上可极大缩短故障排除时间。

二. 全链路基础概念

2.1 Span

2.1.1 工作单元

基本工作单元，一次链路调用（可以是RPC，DB等没有特定的限制）创建一个span，通过一个64位ID标识它，生成的ID的方式可以采用UUID或者任意其他算法，span中还有其他的数据，例如描述信息，时间戳，key-value对的（Annotation）tag信息，parent_id等,其中parent-id可以表示span调用链路来源。



上图说明了span在一次大的跟踪过程中是什么样的。**Dapper记录了span名称，以及每个span的ID和父ID，以重建在一次追踪过程中不同span之间的关系**。如果一个span没有父ID被称为root span。**所有span都挂在一个特定的跟踪上，也共用一个跟踪id**。

可以可以很清楚的看出，这是一次 Span 名为 `Helper.Call` 的调用，Span ID 是 5，Parent Span ID 是 3，Trace ID 是 100。 我们重点看一下 Span 对应的四个状态:

- Client Send（CS）：客户端发送时间，客户端发起一个请求，这个 Annotation 描述了这个 Span 的开始。
- Server Received（SR）：服务端接收时间，服务端获得请求并准备开始处理它，如果将其 SR 减去 CS 时间戳便可得到网络延迟。
- Server Send（SS）：服务端发送时间，Annotation 表明请求处理的完成（当请求返回客户端），如果 SS 减去 SR 时间戳便可得到服务端需要的处理请求时间。
- Client Received（CR）：客户端接收时间，表明 Span 的结束，客户端成功接收到服务端的回复，如果 CR 减去 CS 时间戳便可得到客户端从服务端获取回复的所有所需时间。

通过收集这四个时间戳，就可以在一次请求完成后计算出整个 Trace 的执行耗时和网络耗时，以及 Trace 中每个 Span 过程的执行耗时和网络耗时:

- 服务调用耗时 = CR – CS
- 服务处理耗时 = SS – SR
- 网络耗时 = 服务调用耗时 – 服务处理耗时

Span数据结构(伪代码:只为了阐述span的基础概念):

```
type Span struct {
    TraceID     int64 // 用于标示一次完整的请求id
    Name        string
    ID          int64 // 当前这次调用span_id
    ParentID    int64 // 上层服务的调用span_id  最上层服务parent_id为null
    Annotation []Annotation // 用于标记的时间戳
    Debug       bool
}
```

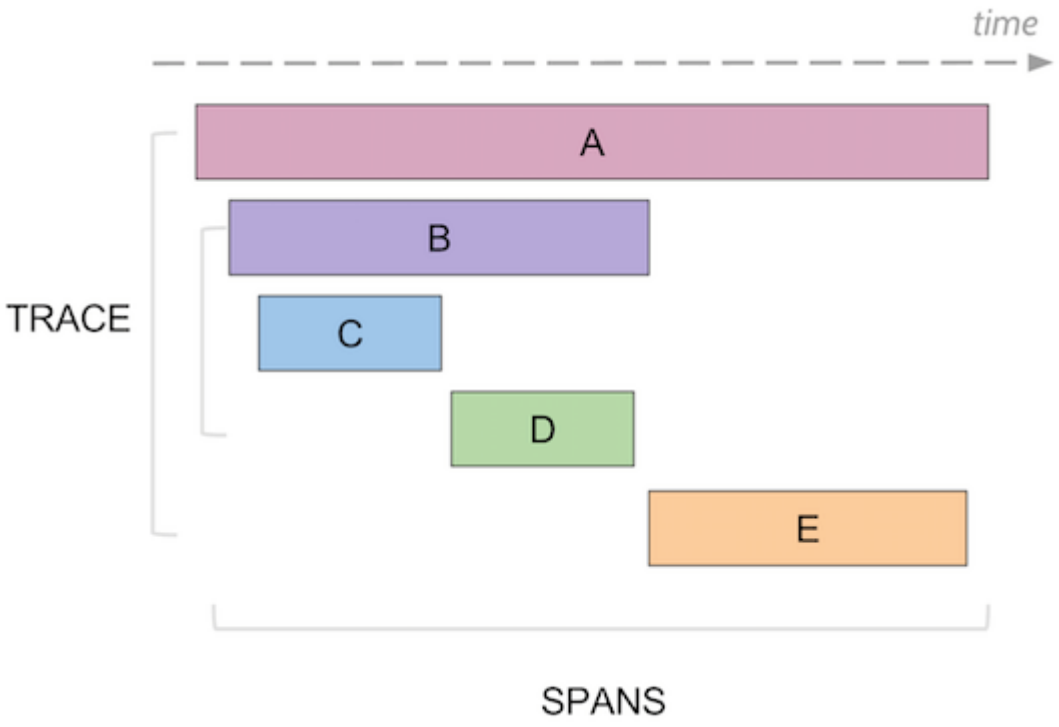
2.1.2 生成Span

我们已经初步了解了 Span 的组成，那么怎么生成 Span 呢？Google Dapper 中使用到的是基于标注 (Annotation-based) 的监控方案。此方案会有代码侵入，所以应尽可能少改动代码。

基于标注的方式就是根据请求中的 Trace ID 来获取 Trace 这个实例，各种编程语言有各自的方式。获取到 Trace 实例后就可以调用 Recorder 来记录 Span 了.

2.2 Trace

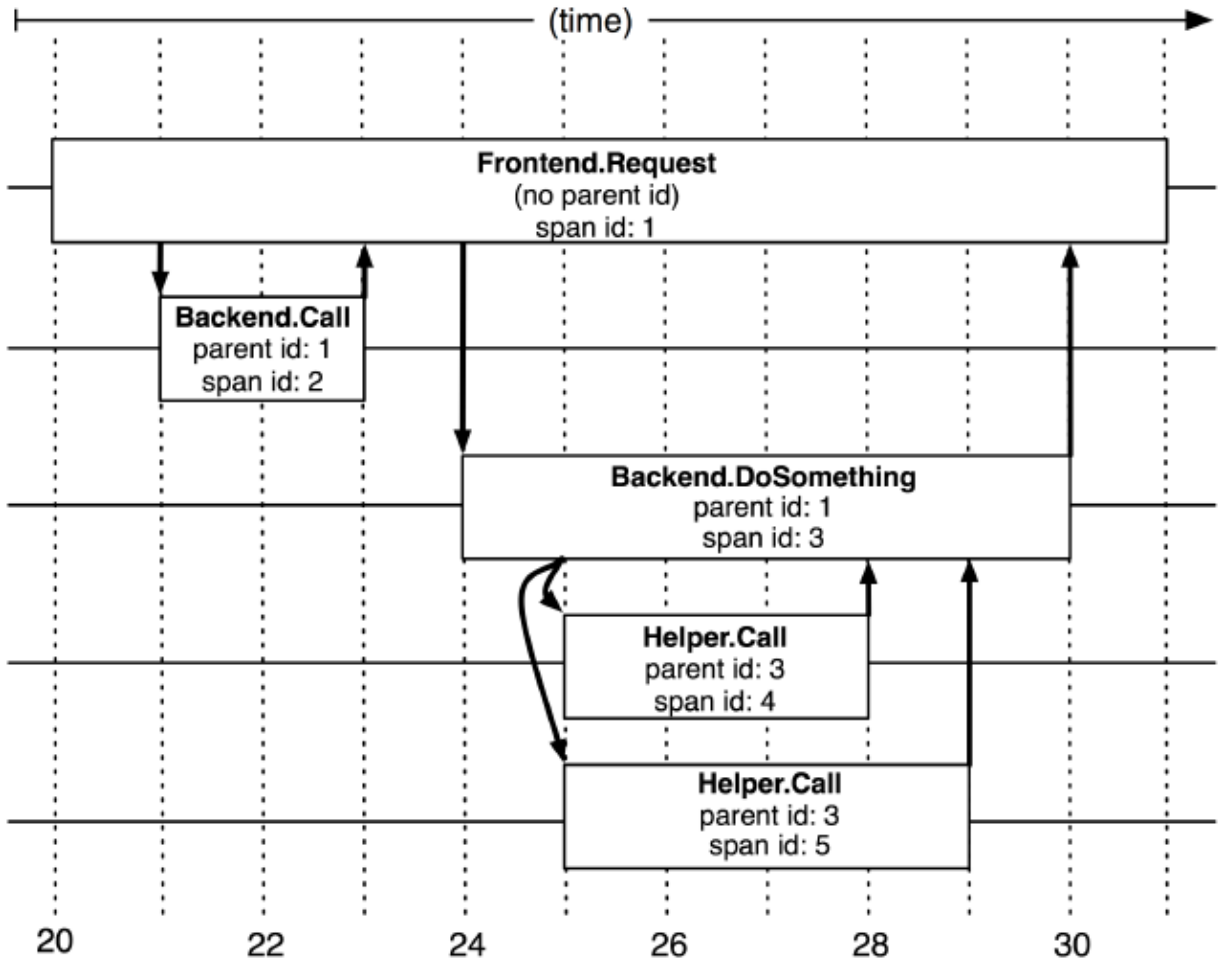
类似于 **树结构的Span集合**，表示一次完整的跟踪，从请求到服务器开始，服务器返回response结束，跟踪每次rpc调用的耗时，存在唯一标识trace_id。比如：你运行的分布式大数据存储一次Trace就由你的一次请求组成。



每种颜色的note标注了一个span，一条链路通过TraceId唯一标识，Span标识发起的请求信息。**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**。节点之间的连线表示的span和它的父span直接的关系。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。

2.3 Trace和span 关系

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。节点之间的连线表示的span和它的父span直接的关系。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，任何RPC相关的时间数据、零个或多个特定应用程序的注解元素,逐渐丰富各个span和trace,一个典型的两者关系的图形如下:



2.4 Annotation(标注,注解)

注解，用来记录请求特定事件相关信息（例如时间），一个span中会有多个annotation注解描述。通常包含四个注解信息：

> (1) **cs**：Client Start，表示客户端发起请求
>
> (2) **sr**：Server Receive，表示服务端收到请求
>
> (3) **ss**：Server Send，表示服务端完成处理，并将结果发送给客户端
>
> (4) **cr**：Client Received，表示客户端获取到服务端返回信息

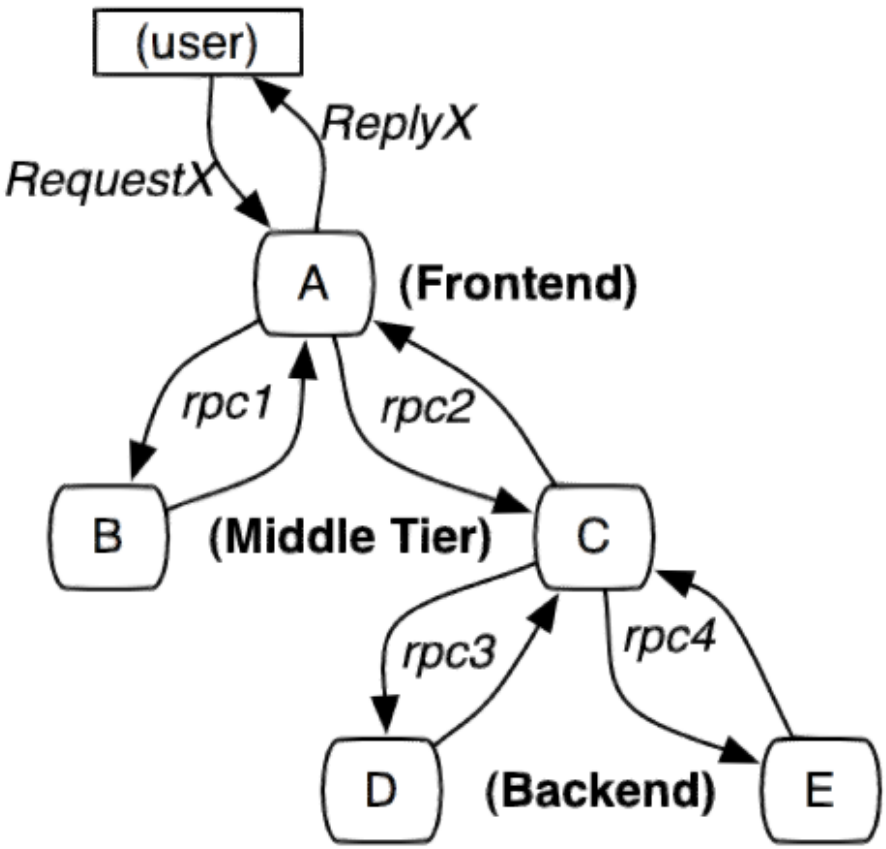
Annotation数据结构：

```
type Annotation struct {
    Timestamp int64
    Value     string
    Host      Endpoint
    Duration  int32
}
```

三. 全链路调用图解

3.1 请求调用示例

1. 当用户发起一个请求时，首先到达前端A服务，然后分别对B服务和C服务进行RPC调用；
2. B服务处理完给A做出响应，但是C服务还需要和后端的D服务和E服务交互之后再返还给A服务，最后由A服务来响应用户的请求；



3.2 调用过程追踪

请求到来生成一个全局TraceID,或者从请求信息中分析出TraceId，通过TraceID可以串联起整个调用链，一个TraceID代表一个请求链。

1. 除了TraceID外，**还需要SpanID用于记录调用父子关系**。每个服务会记录下**parent id**和**span id**，**通过他们可以组织一次完整调用链的父子关系**。
2. 一个没有parent id的span成为root span，可以看成调用链入口。
3. 所有这些ID可用全局唯一的64位整数表示。
4. **整个调用过程中每个请求都要透传TraceID和SpanID**。
5. 每个服务将该次请求附带的TraceID和附带的SpanID作为parent id记录下，并且将自己生成的SpanID也记录下。要查看某次完整的调用则 **只要根据TraceID查出所有调用记录，然后通过parent id和span id组织起整个调用父子关系**。

四. PHP的全链路监控的接入方案

4.1 组件介绍

4.1.1 组件背景

我们结合公司实际情况封装了 Ivory-jaeger-php-client 组件,这个组件是一款全链路跟踪客户端工具，基于PHP开发，项目发源自Jaeger-PHP,框架建立在OpenTracing 1.0.0-beta2 ，接下来主要讲解这个组件的接入过程。

公司内部版本组件代码地址: http://gitlab.shein.com:8088/NineWorlds/Ivory-jaeger-php-client

4.1.2 组件依赖关系

插件主体为 Ivory-jaeger-php-client 代码包，这个代码包依赖两个包:

```
1. opentracing/opentracing （ 1.0.0-beta2）
2. packaged/thrift （ 0.10.0）
```

4.2 组件库函数介绍

4.2.1 初始化Trace

```
$config = Config::getInstance();
$tracer = $config->initTrace('example', '0.0.0.0:6831');
```

4.2.2 解析上下文变量

```
$spanContext = $tracer->extract(Formats\TEXT_MAP, $_SERVER);
```

4.2.3 开始Span

```
$serverSpan = $tracer->startSpan('example HTTP', ['child_of' => $spanContext]);
```

4.2.4 添加BaggageItem

```
$serverSpan->addBaggageItem("version", "2.0.0");
```

4.2.5 注入全局变量

```
$clientTrace->inject($clientSapn1->spanContext, Formats\TEXT_MAP, $_SERVER);
```

4.2.6 添加 Tags 和 Log

```
//can search in Jaeger UI
$span->addTags(['http.status' => "200"]);
//log record
$span->log(['error' => "HTTP request timeout"]);
```

4.2. 7 关闭Trace

```
$config->setDisabled(true);
```

4.2.8 关闭Span 或者 刷新tracer

```
$span->finish();
$config->flush();
```

4.3 组件安装

4.3.1 Ivory-jaeger-php-client 代码包配置

- 复制 Ivory-jaeger-php-client 代码包到vendor目录下，相关目录如下

```
├── [  88]  ivory-jaeger-php-client
│   ├── [ 587]  composer.json
│   ├── [1.1K]  LICENSE
│   ├── [  71]  README.md
│   └── [  20]  src
│       └── [ 221]  Jaeger
│           ├── [3.2K]  Config.php
│           ├── [1.3K]  Constants.php
│           ├── [ 504]  Helper.php
│           ├── [4.7K]  Jaeger.php
│           ├── [  84]  Propagator
│           │   ├── [2.7K]  JaegerPropagator.php
│           │   ├── [ 413]  Propagator.php
│           │   └── [2.4K]  ZipkinPropagator.php
│           ├── [  52]  Reporter
│           │   ├── [ 413]  RemoteReporter.php
│           │   └── [ 152]  Reporter.php
│           ├── [  81]  Sampler
│           │   ├── [ 661]  ConstSampler.php
│           │   ├── [ 772]  ProbabilisticSampler.php
│           │   └── [ 236]  Sampler.php
│           ├── [3.5K]  SpanContext.php
│           ├── [3.2K]  Span.php
│           ├── [ 167]  Thrift
│           │   ├── [1.9K]  AgentClient.php
│           │   ├── [2.7K]  JaegerThriftSpan.php
│           │   ├── [1.8K]  Process.php
```

- 修改composer.json文件中的autoload-psr-4选项

```
    "autoload": {
        "psr-4": {
            "Jaeger\\": "./vendor/ivory-jaeger-php-client/src/Jaeger"
        },
        "files": [
            "./vendor/ivory-jaeger-php-client/src/Jaeger/Constants.php"
        ]
    },
```

- 生成自动加载索引 在根composer.json目录中执行以下命令构建自动加载索引。

```
composer dump-autoload
```

- 引入opentracing 和 thrift代码包

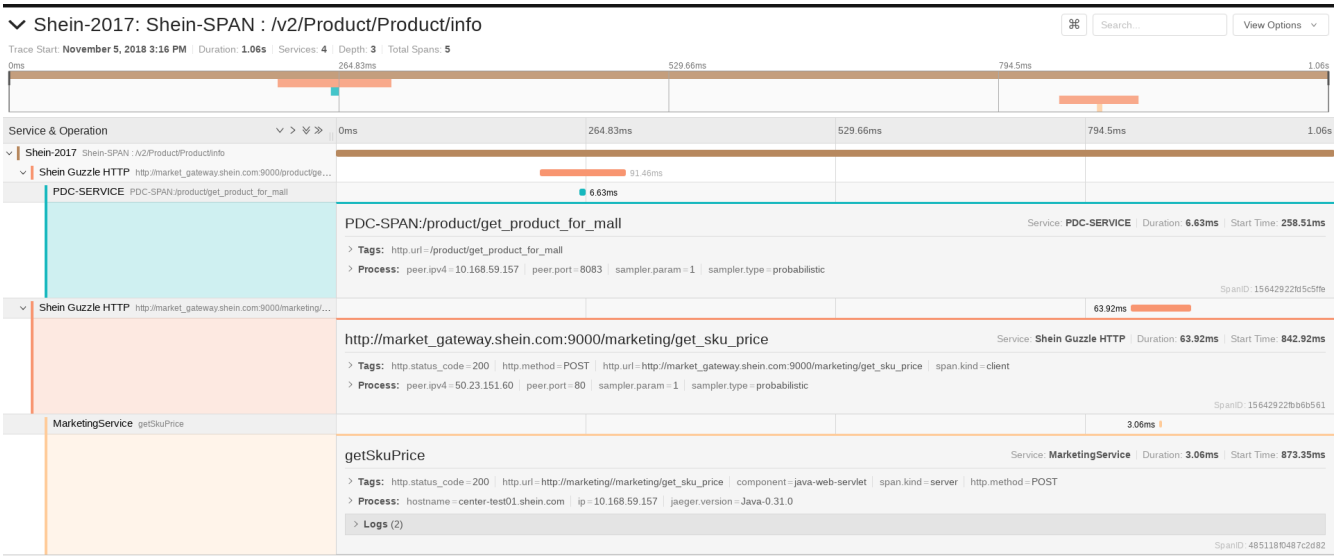
```
composer require opentracing/opentracing:1.0.0-beta2

composer require packaged/thrift:0.10.0
```

五. PHP 全链路接入样例

这里提供一个安装后的使用样例，在2017code中对guzzlehttp包的request函数进行修改，添加对于http请求的全链路跟踪功能。

5.1 效果展示：链路中包括PHP和JAVA



5.2 2017 kernel层修改

在kernel层修改修改,读取框架配置信息,并根据配置创建一个总Trace.

- 修改路径: frame/src/Kernel.php
    1. 添加新的跟踪函数

```
private function initJaegerTracer($sampleRate){
        /*Jaeger-Trace begin
            add config_shein.php config

             'jaeger_agent'=>[
               'udp_uri' =>'udp_ip:udp_port',
               ],

          */
        $shein_configs = $this->getConfig();
        if (isset($shein_configs['jaeger_agent']) &&
isset($shein_configs['jaeger_agent']['udp_uri']) &&
!empty($shein_configs['jaeger_agent']['udp_uri'])) {

            $sheinSpanName=Common::pathInfo();

            $config = JaegerConfig::getInstance();

            $config->setSampler(new ProbabilisticSampler($sampleRate));
            $tracer = $config->initTrace("Shein-2017", $shein_configs['jaeger_agent']
['udp_uri']);
            $spanContext = $tracer->extract(Formats\TEXT_MAP, $_SERVER);
```

```
            $serverSpan = $tracer->startSpan($sheinSpanName, ['child_of' =>
$spanContext]);
            $serverSpan->setTags(['http.url' =>
Common::pathInfo(),'span.kind'=>'server','component'=>'php-shein-frame']);
            $_SERVER['JAEGER_SHEIN_SERVER_SPAN'] = $serverSpan;
            $tracer->inject($serverSpan->spanContext, Formats\TEXT_MAP, $_SERVER);

            $_SERVER['SERVICE_TRACER'] = $tracer;

        }
        //Jaeger-Trace end
}
```

2. request函数中添加调用

```
$this->initJaegerTracer(1); //1代表百分之一采样
```

5.3 Guzzlehttp 包修改

对于Guzzlehttp包内部进行修改,使2017框架所有通过Guzzlehttp请求的HTTP,均可以进入全链路系统中.

- 修改路径: /frame/vendor/guzzlehttp/guzzle/src/Client.php

```
use Jaeger\Config as JaegerConfig;
use OpenTracing\Formats;
use OpenTracing\Reference;
use Order\Base\Exception;

private $baseUriString;  //修改1: 添加此变量用于记录baseuri, 因为guzzle构造url分baseuri和uri区
别

public function __construct(array $config = [])
{
        $this->baseUriString =''; //修改2: 初始化变量
        if (!isset($config['handler'])) {
            $config['handler'] = HandlerStack::create();
        }

        // Convert the base_uri to a UriInterface
        if (isset($config['base_uri'])) {
            $this->baseUriString = $config['base_uri'];  //修改3: 把base_uri中的值进行存储
            $config['base_uri'] = Psr7\uri_for($config['base_uri']);
        }

        $this->configureDefaults($config);
}

//关于request函数的修改只是构造一个tracer和span, 但是框架主tracer和主span需要在kernel层做修改。
public function request($method, $uri = '', array $options = [])
{
        $injectTarget = [];
        $clientTrace = '';
```

```
        $clientSpan = '';

        try{
            if(!empty($_SERVER['SERVICE_TRACER'])){
                $clientTrace = $_SERVER['SERVICE_TRACER'];
                $spanContext = $clientTrace->extract(Formats\TEXT_MAP, $_SERVER); //修
改4: 从$_SERVER变量解析tracer信息
                $fullUriString = $this->baseUriString.$uri;
                $strpoint = strripos($fullUriString,'/');
                $spanName=$fullUriString; //修改4: 构造完整url路径当作span name
                if ($strpoint != false){
                    $strpoint++ ;
                    $spanName = substr($spanName , $strpoint);
                }

                //开始span
                $clientSpan = $clientTrace->startSpan($spanName,
                    ['references' => [
                        //    Reference::create(Reference::FOLLOWS_FROM, $clientSapn1-
>spanContext),
                        Reference::create(Reference::CHILD_OF, $spanContext)
                    ]]);

                //修改4: 注射当前client的span内容
                $clientTrace->inject($clientSpan->spanContext, Formats\TEXT_MAP,
$injectTarget);

                //修改4: 把注射内容存储进request的header参数中，向下一级别链路传播。
                foreach ($injectTarget as $key => $val){
                    $options['headers'][$key] = $val ;
                }
            }

            $options[RequestOptions::SYNCHRONOUS] = true;
            $res = $this->requestAsync($method, $uri, $options)->wait();

            if(!empty($_SERVER['SERVICE_TRACER']) && !empty($clientSpan)){
                //修改4: 设置span的相关tag参数
                $clientSpan->setTags(['http.status_code' => $res->getStatusCode()
                    , 'http.method' => $method, 'http.url' =>
$fullUriString,'span.kind'=>'client','component'=>'php-guzzle-client']);
                //修改4: 当前clientspan终止
                $clientSpan->finish();
            }

            return $res;
        }catch (\Exception $ex){
            if(!empty($_SERVER['SERVICE_TRACER']) && !empty($clientSpan)){
                //修改4: request发生异常，把相关错误状态存入相关span
                $clientSpan->setTags(['http.status_code' => $ex->getCode()
                    , 'http.method' => $method, 'http.url' =>
$fullUriString,'error'=>true,'span.kind'=>'client','component'=>'php-guzzle-client']);
```

```
                //修改4: request发生异常, 存入一个异常日志
                $clientSpan->log(['exception' => $ex->getMessage()]);

                $backtrace = debug_backtrace();

                if(!empty($backtrace)){
                    array_shift($backtrace);
                     //修改4: request发生异常, 存入异常调用来源
                    $clientSpan->log(['exception_from' => $backtrace[0]['class'].'-
>'.$backtrace[0]['function']]);
                    $clientSpan->log(['request_options' => json_encode($options)]);
                }

                $clientSpan->finish();
            }
            throw $ex;
        }

}
```

5.4 2017 kernel层终止阶段修改

在框架的最终阶段添加tracer的终止阶段代码,此阶段主要负责必要的垃圾回收和UDP外部发包.

- 修改路径: /frame/src/Kernel.php
- 为了避免用户请求因为网络发包而造成性能损耗 , 所以请把这部分代码放在cgi通信阶段之后.

```
    use Jaeger\Config as JaegerConfig;

    public function sendResponse(Response $response)
    {
        $this->dispatchEvent(EventStore::SEND_RESPONSE, [
            'request' => $this->getParameter('request'),
            'response' => $response
        ]);
        if (function_exists('pulseflow_output_trace_list')){
            pulseflow_output_trace_list();
        }
        $response->send();

        //Jaeger-Trace after cgi close
        if (!empty($_SERVER['JAEGER_SHEIN_SERVER_SPAN'])) {
            $_SERVER['JAEGER_SHEIN_SERVER_SPAN']->finish();
            unset($_SERVER['JAEGER_SHEIN_SERVER_SPAN']);
        }

        if(!empty($_SERVER['SERVICE_TRACER'])){
            unset($_SERVER['SERVICE_TRACER']);
        }

        $config = JaegerConfig::getInstance();
        $config->flush();
```

```
        exit();
    }
```