腾讯微服务观测平台 TSW

参考信息

产品文档

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100。

文档目录

参考信息
分布式链路追踪规范 Opentracing 详解

最近更新时间：2021-04-23 17:49:31

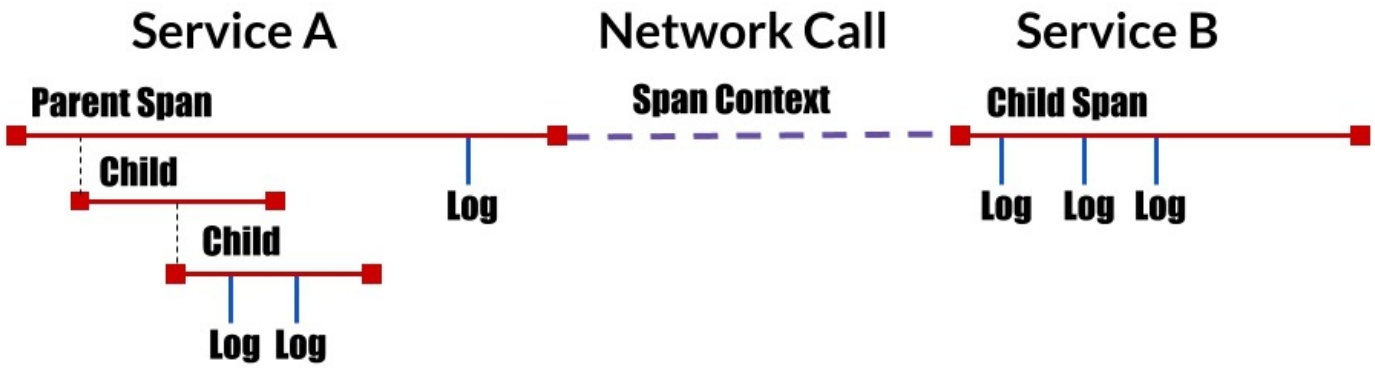
Opentracing 简介

Opentracing 是一个开放的分布式链路追踪框架。OpenTracing 通过提供平台无关、厂商无关的 API，使得开发人员能够方便的添加（或更换）追踪系统的实现。
本文主要介绍 Opentracing 的基本概念及数据模型，以帮助您更好的理解：分布式链路追踪是怎样帮助您观测分布式应用架构、微服务体系的运行状况。

基本概念

大多数分布式追踪系统的思想都源自 Google's Dapper 论文，OpenTracing 也使用相似的概念。

- **Trace**：一次完整请求、事务或流程在分布式系统中的执行工作流。
- **Span**： 一种由执行方法命名的操作，组成工作流的一部分。Spans 具有 key:value 标签（Tags），以及附加到特定 Span 实例的细粒度、带时间戳的结构化日志（调用链日志）。
- **Span Context**：携带工作流的跟踪信息，包括当它通过网络或消息总线将服务传递给服务时。Span上下文包含 Trace 标识符（TraceID）、Span 标识符（SpanID 或 Parent/Child Span）和跟踪系统需要传播到下游服务的任何其他数据。



Opentracing 数据模型

如下图所示，OpenTracing 中的 Trace 是通过 Span 隐式定义的，Trace 可以认为是 Span 的有向无环图，其中 Span 之间的边成为引用（ **Reference** ）。
Span 之间可以是以下逻辑关系：

- **父子（ ChildOf）的逻辑：如 SpanB 和 SpanC 之于 SpanA 的关系，SpanB 和 SpanC 可以是顺序执行也可以是并行。**
- **旁挂（FollowsFrom）的逻辑：如 SpanG 和 SpanF 的关系。**

```
# ref: https://github.com/opentracing/specification/blob/master/specification.md#the-opentracing-data-model
Causal relationships between Spans in a single Trace
[Span A] ←←←(the root span)
|
+------+------+
| |
[Span B] [Span C] ←←←(Span C is a `ChildOf` Span A)
| |
[Span D] +---+-------+
| |
[Span E] [Span F] >>> [Span G] >>> [Span H]
↑
↑
↑
(Span G `FollowsFrom` Span F)
```

从实用角度，通过如下所示的时序图能够更方便的理解调用链。

```
—|———————|———————|———————|———————|———————|———————|———————|-> time
[Span A·····································]
[Span B·····································]
[Span D·····································]
[Span C·····································]
[Span E·····] [Span F··] [Span G··] [Span H··]
```

Span 状态

每一个 Span 封装以下状态：

- **操作名称（Method）**
- **起始时间戳**
- **完成时间戳**

- 一组零个或多个 key:value 的 Span Tags，keys 必须是字符串，values 可以是 strings、bools、numeric 类型。
- 一组零个或多个 Span Logs，调用链日志自身是与时间戳匹配的 key:value 对。键必须是字符串，大部分 Opentracing 实现中的值可以是任何类型。
- 一个 SpanContext：包括 1. TraceID 与 SpanID。2. Baggage Items 跨进程边界的键值对集合。
- Reference：通过 SpanContext 引用零个或多个因果相关的 Spans。

OpenTracing SDK 应用举例

本章节主要介绍 Span 创建的方法，及上下游需要的信息。多个组件被集成到框架中，提供免费服务，但是也屏蔽了数据结构的创建，所以我们以基础的 OpenTracing SDK 为例，用简单的代码完成创建一个 Hello World 程序的全流程。

```
# Ref: https://opentracing.io/guides/javascript/
const http = require('http');
const opentracing = require('opentracing'); ## 这里是最上层的引用
// NOTE: the default OpenTracing tracer does not record any tracing information.
// Replace this line with the tracer implementation of your choice.
const tracer = new opentracing.Tracer(); ## 然后创建出Tracer 对象，用来创建Span
const span = tracer.startSpan('http_request'); ## 第一个和TraceID一起被创建出来的Span
const opts = {
host : 'example.com',
method: 'GET',
port : '80',
path: '/',
};
http.request(opts, res => {
res.setEncoding('utf8');
res.on('error', err => {
// assuming no retries, mark the span as failed
span.setTag(opentracing.Tags.ERROR, true); ## Span可以打Tag
span.log({'event': 'error', 'error.object': err, 'message': err.message, 'stack': er
r.stack}); ## Span也可以写log
span.finish();
});
res.on('data', chunk => {
span.log({'event': 'data_received', 'chunk_length': chunk.length}); }); res.on('end'
, () => {
span.log({'event': 'request_end'}); span.finish();
```

```
});
}).end();
```

解析 Ctx 和开启新的 Span：

```
// Use the inbound HTTP request's headers as a text map carrier.
var headersCarrier = inboundHTTPReq.headers;
var wireCtx = Tracer.extract(Tracer.FORMAT_HTTP_HEADERS, headersCarrier);
var serverSpan = Tracer.startSpan('...', { childOf : wireCtx });
```

传递信息：

```
// from https://github.com/opentracing-contrib/java-kafka-client
// Register tracer with GlobalTracer:
GlobalTracer.register(tracer);
// Add TracingProducerInterceptor to sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingProducerInterceptor.class.getName());


// Instantiate KafkaProducer
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send
producer.send(...);


// Add TracingConsumerInterceptor to consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingConsumerInterceptor.class.getName());

// Instantiate KafkaConsumer
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

//Subscribe
consumer.subscribe(Collections.singletonList("messages"));
```

```
// Get records
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// To retrieve SpanContext from polled record (Consumer side)
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext =
TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

interceptor 在 producer 发送的时候会创建 Span，并且把 SpanContext 放在 record Headers 中。

```
// Class TracingKafkaUtils
public static void inject(SpanContext spanContext, Headers headers,
Tracer tracer) {
tracer.inject(spanContext, Format.Builtin.TEXT_MAP, new
HeadersMapInjectAdapter(headers));
}
```

Consumer 读取的时会关闭 Span，表示整个周期结束。

```
// TracingConsumerInterceptor<K, V> implements ConsumerInterceptor<K, V>
@Override
public ConsumerRecords<K, V> onConsume(ConsumerRecords<K, V> records) {
for (ConsumerRecord<K, V> record : records) {
TracingKafkaUtils.buildAndFinishChildSpan(record, GlobalTracer.get());
} return records;
}
```

大部分开源组件都提供了类似 tracing 的功能。例如 pulsar-tracing 项目，专门给 pulsar 做了一套集成 opentracing 的免费功能。

```
// https://github.com/streamnative/pulsar-tracing
// Instantiate Producer with tracing interceptor.
Producer<String> producer = client
.newProducer(Schema.STRING)
.intercept(new TracingProducerInterceptor())
.topic("your-topic")
```

```
.create();

// Send messages.
producer.send("Hello OpenTracing!");

// Instantiate Consumer with tracing interceptor.
Consumer<String> consumer = client.newConsumer(Schema.STRING)
.topic("your-topic")
.intercept(new TracingConsumerInterceptor<>())
.subscriptionName("your-sub")
.subscribe();

// Receive messages.
Message<String> message = consumer.receive();

// To retrieve SpanContext from the message(Consumer side).
SpanContext spanContext = TracingPulsarUtils.extractSpanContext(message, tracer);
```

说明：

您还可参考 Opentracing 标准，了解相关概念的详细信息。