bobby · 更新于 2022-11-16

上一节 3. jaeger安装和... 5. grpc下添加ja... 下一节

OpenTracing语义标准

综述 这是正式的OpenTracing语义标准。OpenTracing是一个跨编程语言的标准，此文档会避免具有语言特性的概念。比如，我们在文档中使用"interface"，因为所有的语言都包含"interface"这种概念。 ### 版本命名策略 OpenTracing标准使用`Major.Minor`版本命名策略（即：大版本.小版本），但不包含`.Patch`版本（即：补丁版本）。如果标准做出不向前兼容的改变，则使用“主版本”号提升。如果是向前兼容的改进，则进行小版本号提升，例如加入新的标准tag, log和SpanContext引用类型。（如果你想知道更多关于制定此版本政策的原因，可参考[specification#2](https://github.com/opentracing/specification/issues/2#issuecomment-261740811)） ## OpenTracing数据模型 OpenTracing中的Trace（调用链）通过归属于此调用链的Span来隐性的定义。 特别说明，一条Trace（调用链）可以被认为是一个由多个Span组成的有向无环图（DAG图）， Span与Span的关系被命名为References。

译者注: Span，可以被翻译为跨度，可以被理解为一次方法调用, 一个程序块的调用, 或者一次RPC/数据库访问.只要是一个具有完整时间周期的程序访问，都可以被认为是一个span.在此译本中，为了便于理解，Span和其他标准内声明的词汇，全部不做名词翻译。

例如：下面的示例Trace就是由8个Span组成： ``` 1. 单个Trace中，span间的因果关系 2. 3. 4. [Span A] ←←←(the root span) 5. | 6. +------+------+ 7. | | 8. [Span B] [Span C] ←←←(Span C 是 Span A 的孩子节点, ChildOf) 9. | | 10. [Span D] +---+-------+ 11. | | 12. [Span E] [Span F] >>> [Span G] >>> [Span H] 13. ↑ 14. ↑ 15. ↑ 16. (Span G 在 Span F 后被调用, FollowsFrom) 17. ``` 有些时候，使用下面这种，基于时间轴的时序图可以更好的展现Trace（调用链）： ``` 1. 单个Trace中，span间的时间关系 2. 3. 4. –|––––––|–––––––|–––––––|–––––––|–––––––|–––––––|–––––––|-> time 5. 6. [Span A·····] 7. [Span B·····] 8. [Span D·····] 9. [Span C·····] 10. [Span E·····] [Span F··] [Span G··] [Span H··] ``` 每个Span包含以下的状态:（译者注：由于这些状态会反映在OpenTracing API中，所以会保留部分英文说明）

- An operation name，操作名称
- A start timestamp，起始时间
- A finish timestamp，结束时间
- Span Tag，一组键值对构成的Span标签集合。键值对中，键必须为string，值可以是字符串，布尔，或者数字类型。
- Span Log，一组span的日志集合。 每次log操作包含一个键值对，以及一个时间戳。 键值对中，键必须为string，值可以是任意类型。 但是需要注意，不是所有的支持OpenTracing的Tracer,都需要支持所有的值类型。
- SpanContext，Span上下文对象 (下面会详细说明)
- References(Span间关系)，相关的零个或者多个Span（Span间通过SpanContext建立这种关系）

每一个SpanContext包含以下状态:

- 任何一个OpenTracing的实现，都需要将当前调用链的状态（例如：trace和span的id），依赖一个独特的Span去跨进程边界传输
- Baggage Items，Trace的随行数据，是一个键值对集合，它存在于trace中，也需要跨进程边界传输

Span间关系

一个Span可以与一个或者多个SpanContexts存在因果关系。OpenTracing目前定义了两种关系：`ChildOf`（父子）和`FollowsFrom`（跟随）。这两种关系明确的给出了两个父子关系的Span的因果模型。 将来，OpenTracing可能提供非因果关系的span间关系。（例如：span被批量处理，span被阻塞在同一个队列中，等等）。

`ChildOf` 引用: 一个span可能是一个父级span的孩子，即"ChildOf"关系。在"ChildOf"引用关系下，父级span某种程度上取决于子span。下面这些情况会构成"ChildOf"关系：

- 一个RPC调用的服务端的span，和RPC服务客户端的span构成ChildOf关系
- 一个sql insert操作的span，和ORM的save方法的span构成ChildOf关系
- 很多span可以并行工作（或者分布式工作）都可能是一个父级的span的子项，他会合并所有子span的执行结果，并在指定期限内返回

下面都是合理的表述一个"ChildOf"关系的父子节点关系的时序图。

代码块

```
1.  [-Parent Span---------]
2.       [-Child Span----]
3.
4.  [-Parent Span--------------]
5.       [-Child Span A----]
6.        [-Child Span B----]
7.      [-Child Span C----]
8.       [-Child Span D---------------]
9.       [-Child Span E----]
```

`FollowsFrom` 引用: 一些父级节点不以任何方式依赖他们子节点的执行结果，这种情况下，我们说这些子span和父span之间是"FollowsFrom"的因果关系。"FollowsFrom"关系可以被分为很多不同的子类型，未来版本的OpenTracing中将正式的区分这些类型

下面都是合理的表述一个"FollowFrom"关系的父子节点关系的时序图。

代码块

```
1.  [-Parent Span-]  [-Child Span-]
2.
3.
4.  [-Parent Span--]
5.   [-Child Span-]
6.
7.
8.  [-Parent Span-]
9.              [-Child Span-]
```

OpenTracing API

OpenTracing标准中有三个重要的相互关联的类型，分别是`Tracer`，`Span` 和 `SpanContext`。下面，我们分别描述每种类型的行为，一般来说，每个行为都会在各语言实现层面上，会演变成一个方法，而实际上由于方法重载，很可能演变成一系列相似的方法。

当我们讨论“可选”参数时，需要强调的是，不同的语言针对可选参数有不同理解，概念和实现方式 。例如，在Go中，我们习惯使用"functional Options"，而在Java中，我们可能使用builder模式。

`Tracer` 接口用来创建 `Span`，以及处理如何处理 `Inject` (serialize) 和 `Extract` (deserialize)，用于跨进程边界传递。它具有如下官方能力：

创建一个新 `Span`

必填参数

- operation name, 操作名, 一个具有可读性的字符串，代表这个span所做的工作（例如：RPC方法名，方法名，或者一个大型计算中的某个阶段或子任务）。操作名应该是一个抽象、通用，明确、具有统计意义的名称。因此，`"get_user"` 作为操作名，比 `"get_user/314159"` 更好。

例如，假设一个获取账户信息的span会有如下可能的名称：

操作名	指导意见
`get`	太抽象
`get_account/792`	太明确
`get_account`	正确的操作名，关于 `account_id=792` 的信息应该使用Tag操作

可选参数

- 零个或者多个关联（references）的 `SpanContext`，如果可能，同时快速指定关系类型，`ChildOf` 还是 `FollowsFrom`。
- 一个可选的显性传递的开始时间；如果忽略，当前时间被用作开始时间。
- 零个或者多个tag。

返回值，返回一个已经启动 `Span` 实例（已启动，但未结束。译者注：英语上started和finished理解容易混淆）

将 `SpanContext` 上下文Inject（注入）到carrier

必填参数

- ** `SpanContext` **实例
- format（格式化）描述，一般会是一个字符串常量，但不做强制要求。通过此描述，通知 `Tracer` 实现，如何对 `SpanContext` 进行编码放入到carrier中。
- carrier，根据format确定。 `Tracer` 实现根据format声明的格式，将 `SpanContext` 序列化到carrier对象中。

将 `SpanContext` 上下文从carrier中Extract（提取）

必填参数

- format（格式化）描述，一般会是一个字符串常量，但不做强制要求。通过此描述，通知 `Tracer` 实现，如何从carrier中解码 `SpanContext`。
- carrier，根据format确定。 `Tracer` 实现根据format声明的格式，从carrier中解码 `SpanContext`。

返回值，返回一个 `SpanContext` 实例，可以使用这个 `SpanContext` 实例，通过 `Tracer` 创建新的 `Span`。

注意，对于Inject（注入）和Extract（提取），format是必须的。

Inject（注入）和Extract（提取）依赖于可扩展的format参数。format参数规定了另一个参数"carrier"的类型，同时约束了"carrier"中 `SpanContext` 是如何编码的。所有的Tracer实现，都必须支持下面的format。

- HTTP Headers: 适合作为HTTP头信息的，基于字符串：字符串的map。（RFC 7230.在工程实践中，如何处理HTTP头具有多样性，强烈建议tracer的使用者谨慎使用HTTP头的键值空间和转义符）
- Binary: 一个简单的二进制大对象，记录 `SpanContext` 的信息。

Span

当 `Span` 结束后(`span.finish()`)，除了通过 `Span` 获取 `SpanContext` 外，下列其他所有方法都不允许被调用。

除了通过 `Span` 获取 `SpanContext`

不需要任何参数。
返回值， `Span` 构建时传入的 `SpanContext` 。这个返回值在 `Span` 结束后(`span.finish()`)，依然可以使用。

复写操作名（operation name）

必填参数

- 新的操作名operation name，覆盖构建 `Span` 时，传入的操作名。

结束 `Span`

可选参数

- 一个明确的完成时间;如果省略此参数，使用当前时间作为完成时间。

为 `Span` 设置tag

必填参数

- tag key，必须是string类型
- tag value，类型为字符串，布尔或者数字

注意，OpenTracing标准包含**“standard tags，标准Tag”**，此文档中定义了Tag的标准含义。

Log结构化数据

必填参数

- 一个或者多个键值对，其中键必须是字符串类型，值可以是任意类型。某些OpenTracing实现，可能支持更多的log值类型。

可选参数

- 一个明确的时间戳。如果指定时间戳，那么它必须在span的开始和结束时间之内。

注意，OpenTracing标准包含**“standard log keys，标准log的键”**，此文档中定义了这些键的标准含义。

设置一个baggage（随行数据）元素

Baggage元素是一个键值对集合，将这些值设置给给定的 `Span` ， `Span` 的 `SpanContext` ，以及所有和此 `Span` 有直接或者间接关系的本地 `Span` 。 也就是说，baggage元素随trace一起保持在带内传递。（译者注：带内传递，在这里指，随应用程序调用过程一起传递）
Baggage元素为OpenTracing的实现全栈集成，提供了强大的功能 （例如：任意的应用程序数据，可以在移动端创建它，显然的，它会一直传递了系统最底层的存储系统。由于它如此强大的功能，他也会产生巨大的开销，请小心使用此特性。

span中，

此，总体上，他会有明显的网络和CPU开销。
必填参数

- baggage key, 字符串类型
- baggage value, 字符串类型

获取一个baggage元素

必填参数

- baggage key, 字符串类型

返回值，相应的baggage value,或者可以标识元素值不存在的返回值（译者注：如Null）。

SpanContext

相对于OpenTracing中其他的功能， SpanContext 更多的是一个“概念”。也就是说，OpenTracing实现中，需要重点考虑，并提供一套自己的API。 OpenTracing的使用者仅仅需要，在创建span、向传输协议Inject（注入）和从传输协议中Extract（提取）时，使用 SpanContext 和references，
OpenTracing要求， SpanContext 是不可变的，目的是防止由于 Span 的结束和相互关系，造成的复杂生命周期问题。

遍历所有的baggage元素

遍历模型依赖于语言，实现方式可能不一致。在语义上，要求调用者可以通过给定的 SpanContext 实例，高效的遍历所有的baggage元素

NoopTracer

所有的OpenTracing API实现，必须提供某种方式的 NoopTracer 实现。 NoopTracer 可以被用作控制或者测试时，进行无害的inject注入（等等）。例如，在 OpenTracing-Java实现中， NoopTracer 在他自己的模块中。

可选 API 元素

有些语言的OpenTracing实现，为了在串行处理中，传递活跃的 Span 或 SpanContext ，提供了一些工具类。例如， opentracing-go 中，通过 context.Context 机制，可以设置和获取活跃的 Span 。

3. jaeger安装和架构 上一节 下一节 5. grpc下添加jaeger

我要提出意见反馈