服务器出问题，目前部分恢复

33 链路跟踪：如何基于 Hook 机制以及 OpenTracing 协议实现数据访问链路跟踪?

今天我们来讨论 ShardingSphere 中关于编排治理的另一个主题，即链路跟踪。在分布式系统开发过程中，链路跟踪是一项基础设施类的功能。作为一款分布式数据库中间件，ShardingSphere 中也内置了简单而完整的链路跟踪机制。

链路跟踪基本原理和工具

在介绍具体的实现过程之前，我们有必要先来了解一些关于链路跟踪的理论知识。

1.链路跟踪基本原理

分布式环境下的服务跟踪原理上实际上并不复杂，我们首先需要引入两个基本概念，即 TraceId 和 SpanId。

- **TraceId**

TraceId 即跟踪 Id。在微服务架构中，每个请求生成一个全局的唯一性 Id，通过这个 Id 可以串联起整个调用链，也就是说请求在分布式系统内部流转时，系统需要始终保持传递其唯一性 Id，直到请求返回，这个唯一性 Id 就是 TraceId。

- **SpanId**

除了 TraceId 外，我们还需要 SpanId，SpanId 一般被称为跨度 Id。当请求到达各个服务组件时，通过 SpanId 来标识它的开始、具体执行过程和结束。对于每个 Span 而言，它必须有开始和结束两个节点，通过记录开始 Span 和结束 Span 的时间戳统计其 Span 的时间延迟。

整个调用过程中每个请求都要透传 TraceId 和 SpanId。每个服务将该次请求附带的 SpanId 作为父 SpanId 进行记录，并且生成自己的 SpanId。一个没有父 SpanId 的 Span 即为根 Span，可以看成调用链入口。所以要查看某次完整的调用只需根据 TraceId 查出所有调用记录，然后通过父 SpanId 和 SpanId 组织起整个调用父子关系。事实上，围绕如何构建 Trace 和 Span 之间统一的关联关系，业界也存在一个通用的链接跟踪协议，这就是 OpenTracing 协议。

2.OpenTracing 协议和应用方式

OpenTracing 是一种协议，也使用与上面介绍的类似的术语来表示链路跟踪的过程。通过提供平台无关、厂商无关的 API，OpenTracing 使得开发人员能够方便的添加或更换链路跟踪系统的实现。目前，诸如 Java、Go、Python 等主流开发语言都提供了对 OpenTracing 协议的支持。

我们以 Java 语言为例来介绍 OpenTracing 协议的应用方式，OpenTracing API 中存在相互关联的最重要的对象，也就是 Tracer 和 Span 接口。

对于 Tracer 接口而言，最重要就是如下所示的 buildSpan 方法，该方法用来根据某一个操作创建一个 Span 对象：

```
SpanBuilder buildSpan(String operationName);
```

我们看到上述 buildSpan 方法返回的实际上是一个 SpanBuilder 对象，而 SpanBuilder 中则存在一组 withTag 重载方法，用于为当前 Span 添加一个标签。标签的作用是供用户进行自定义，可以用来检索查询的标记，是一组键值对。withTag 方法的其中一种定义如下所示：

```
SpanBuilder withTag(String key, String value);
```

我们可以为一个 Span 添加多个 Tag，当把 Tag 添加完毕之后，我们就可以调用如下所示的 start 方法来启动这个 Span：

```
Span start();
```

注意这个方法会返回一个 Span 对象，一旦获取了 Span 对象，我们就可以调用该对象中的 finish 方法来结束这个 Span，该方法会为 Span 自动填充结束时间：

```
void finish();
```

基于以上 OpenTracing API 的介绍，在日常开发过程中，我们在业务代码中嵌入链路跟踪的常见实现方法可以用如下所示的代码片段进行抽象：

```
//从 OpenTracing 规范的实现框架中获取 Tracer 对象
Tracer tracer = new XXXTracer();
//创建一个 Span 并启动
Span span = tracer.buildSpan("test").start();
//添加标签到 Span 中
span.setTag(Tags.COMPONENT, "test -application");
//执行相关业务逻辑
//完成 Span
```

```
span.finish();
//可以根据需要获取 Span 中的相关信息
System.out.println("Operation name = " + span.operationName());
System.out.println("Start = " + span.startMicros());
System.out.println("Finish = " + span.finishMicros());
```

事实上，ShardingSphere 集成 OpenTracing API 的做法基本与上述方法类似，让我们一起来看一下。

ShardingSphere 中的链路跟踪

对于 ShardingSphere 而言，框架本身并不负责如何采集、存储以及展示应用性能监控的相关数据，而是将整个数据分片引擎中最核心的 SQL 解析与 SQL 执行相关信息发送至应用性能监控系统，并交由其处理。

换句话说，ShardingSphere 仅负责产生具有价值的数据，并通过标准协议递交至第三方系统，而不对这些数据做二次处理。

ShardingSphere 使用 OpenTracing API 发送性能追踪数据。支持 OpenTracing 协议的具体产品都可以和 ShardingSphere 自动对接，比如常见的 SkyWalking、Zipkin 和Jaeger。在 ShardingSphere 中，使用这些具体产品的方式只需要在启动时配置 OpenTracing 协议的实现者即可。

1.通过 ShardingTracer 获取 Tracer 类

ShardingSphere 中，所有关于链路跟踪的代码都位于 sharding-opentracing 工程中。我们先来看 ShardingTracer 类，该类的 init 方法完成了 OpenTracing 协议实现类的初始化，如下所示：

```
public static void init() {
         //从环境变量中获取 OpenTracing 协议的实现类配置
        String tracerClassName = System.getProperty(OPENTRACING_TRACER_CLASS_
        Preconditions.checkNotNull(tracerClassName, "Can not find opentracing
        try {
             //初始化 OpenTracing 协议的实现类
            init((Tracer) Class.forName(tracerClassName).newInstance());
        } catch (final ReflectiveOperationException ex) {
            throw new ShardingException("Initialize opentracing tracer class
```

```
        }
    }
```

我们通过配置的 OPENTRACING_TRACER_CLASS_NAME 获取 OpenTracing 协议实现类的类名，然后通过反射创建了实例。例如，我们可以配置该类为如下所示的 Skywalking 框架中的 SkywalkingTracer 类：

```
org.apache.shardingsphere.opentracing.tracer.class=org.apache.skywalking.apm.
```

当然，ShardingTracer 类也提供了通过直接注入 OpenTracing 协议实现类的方法来进行初始化。实际上上述 init 方法最终也是调用了如下所示的 init 重载方法：

```
public static void init(final Tracer tracer) {
        if (!GlobalTracer.isRegistered()) {
            GlobalTracer.register(tracer);
        }
    }
```

该方法把 Tracer 对象存放到全局的 GlobalTracer 中。GlobalTracer 是 OpenTracing API 提供的一个工具类，使用设计模式中的单例模式来存储一个全局性的 Tracer 对象。它的变量定义、register 方法以及 get 方法如下所示：

```
private static final GlobalTracer INSTANCE = new GlobalTracer();
public static synchronized void register(final Tracer tracer) {
        if (tracer == null) {
            throw new NullPointerException("Cannot register GlobalTracer <nul
        }
        if (tracer instanceof GlobalTracer) {
            LOGGER.log(Level.FINE, "Attempted to register the GlobalTracer as
            return; // no-op
        }
        if (isRegistered() && !GlobalTracer.tracer.equals(tracer)) {
            throw new IllegalStateException("There is already a current globa
        }
```

```
        GlobalTracer.tracer = tracer;
    }

    public static Tracer get() {
        return INSTANCE;
    }
```

采用这种方式，初始化可以采用如下方法：

```
ShardingTracer.init(new SkywalkingTracer());
```

而获取具体 Tracer 对象的方法则直接调用 GlobalTracer 的同名方法即可，如下所示：

```
public static Tracer get() {
        return GlobalTracer.get();
}
```

2.基于 Hook 机制填充 Span

一旦获取 Tracer 对象，我们就可以使用该对象来构建各种 Span。ShardingSphere 采用了 Hook 机制来填充 Span。说道 Hook 机制，我们可以回想《15 | 解析引擎：SQL 解析流程应该包括哪些核心阶段（上）？》中的相关内容，在如下所示的 SQLParseEngine 类的 parse 方法中用到了 ParseHook：

```
public SQLStatement parse(final String sql, final boolean useCache) {
            //基于 Hook 机制进行监控和跟踪
        ParsingHook parsingHook = new SPIParsingHook();
        parsingHook.start(sql);
        try {
           //完成 SQL 的解析，并返回一个 SQLStatement 对象
            SQLStatement result = parse0(sql, useCache);
            parsingHook.finishSuccess(result);
            return result;
        } catch (final Exception ex) {
```

```
            parsingHook.finishFailure(ex);
            throw ex;
        }
    }
```

注意到上述代码中创建了一个 SPIParsingHook，并实现了 ParsingHook 接口，该接口的定义如下所示：

```
public interface ParsingHook {

    //开始 Parse 时进行 Hook
    void start(String sql);

    //成功完成 Parse 时进行 Hook
    void finishSuccess(SQLStatement sqlStatement);

    //Parse 失败时进行 Hook
    void finishFailure(Exception cause);
}
```

SPIParsingHook 实际上是一种容器类，将所有同类型的 Hook 通过 SPI 机制进行实例化并统一调用，SPIParsingHook 的实现方式如下所示：

```
public final class SPIParsingHook implements ParsingHook {

    private final Collection<ParsingHook> parsingHooks = NewInstanceServiceLo

    static {
        NewInstanceServiceLoader.register(ParsingHook.class);
    }

    @Override
    public void start(final String sql) {
```

```
            for (ParsingHook each : parsingHooks) {
                each.start(sql);
            }
        }

        @Override
        public void finishSuccess(final SQLStatement sqlStatement, final Sharding
            for (ParsingHook each : parsingHooks) {
                each.finishSuccess(sqlStatement, shardingTableMetaData);
            }
        }

        @Override
        public void finishFailure(final Exception cause) {
            for (ParsingHook each : parsingHooks) {
                each.finishFailure(cause);
            }
        }
    }
```

这里，我们看到了熟悉的 NewInstanceServiceLoader 工具类。这样，我们一旦实现了 ParsingHook，就会在执行 SQLParseEngine 类的 parse 方法时将 Hook 相关的功能嵌入到系统的执行流程中。

另外，OpenTracingParsingHook 同样实现了 ParsingHook 接口，如下所示：

```
public final class OpenTracingParsingHook implements ParsingHook {

    private static final String OPERATION_NAME = "/" + ShardingTags.COMPONENT

    private Span span;
```

```
    @Override
    public void start(final String sql) {
        //创建 Span 并设置 Tag
        span = ShardingTracer.get().buildSpan(OPERATION_NAME)
                .withTag(Tags.COMPONENT.getKey(), ShardingTags.COMPONENT_NAME
                .withTag(Tags.SPAN_KIND.getKey(), Tags.SPAN_KIND_CLIENT)
                .withTag(Tags.DB_STATEMENT.getKey(), sql).startManual();
    }

    @Override
    public void finishSuccess(final SQLStatement sqlStatement) {
        //成功时完成 Span
        span.finish();
    }

    @Override
    public void finishFailure(final Exception cause) {
        //失败时完成 Span（
        ShardingErrorSpan.setError(span, cause);
        span.finish();
    }
}
```

我们知道 Tracer 类提供了 buildSpan 方法创建自定义的 Span 并可以通过 withTag 方法添加自定义的标签。最后，我们可以通过 finish 方法类关闭这个 Span。在这个方法中，我们看到了这些方法的具体应用场景。

同样，在《21 | 执行引擎：分片环境下 SQL 执行的整体流程应该如何进行抽象？》中，我们在的 SQLExecuteCallback 抽象类的 execute0 方法中也看到了 SQLExecutionHook 的应用场景，SQLExecutionHook 接口的定义如下所示：

```
public interface SQLExecutionHook {
```

```
    //开始执行 SQL 时进行 Hook
    void start(String dataSourceName, String sql, List<Object> parameters, Da

    //成功完成 SQL 执行时进行 Hook
    void finishSuccess();

    //SQL 执行失败时进行 Hook
    void finishFailure(Exception cause);
}
```

在 ShardingSphere 中，同样存在一套完整的体系来完成对这个接口的实现，包括与 SPIParsingHook 同样充当容器类的 SPISQLExecutionHook，以及基于 OpenTracing 协议的 OpenTracingSQLExecutionHook，其实现过程与 OpenTracingParsingHook 一致，这里不再做具体展开。

从源码解析到日常开发

在今天的内容中，我们可以从 ShardingSphere 的源码中提炼两个可以用于日常开发过程的开发技巧。如果我们需要自己实现一个用于分布式环境下的链路监控和分析系统，那么 OpenTracing 规范以及对应的实现类是你的首选。

基于 OpenTracing 规范，业界存在一大批优秀的工具，例如 SkyWalking、Zipkin 和 Jaeger 等，这些工具都可以很容易的集成到你的业务系统中。

另一方面，我们也看到了 Hook 机制的应用场景和方式。Hook 本质上也是一种回调机制，我们可以根据需要提炼出自身所需的各种 Hook，并通过 SPI 的方式动态加载到系统中，以满足不同场景下的需要，ShardingSphere 为我们如何实现和管理系统中的 Hook 实现类提供了很好的实现参考。

小结与预告

今天的内容围绕 ShardingSphere 中的链路跟踪实现过程进行了详细展开。我们发现在 ShardingSphere 中关于链路跟踪的代码并不多，所以为了更好地理解链路跟踪的实现机制，我们也花了一些篇幅介绍了链路跟踪的基本原理，以及背后的 OpenTracing 规范的核心类。

然后，我们发现 ShardingSphere 在业务流程的执行过程中内置了一批 Hook，这些 Hook 能够帮助系统收集各种监控信息，并通过 OpenTracing 规范的各种实现类进行统一管理。

这里给你留一道思考题：ShardingSphere 中如何完成与 OpenTracing 协议的集成?

在下一个课时中，我们将介绍 ShardingSphere 源码解析部分的最后一个主题，即 ShardingSphere 的内核如何与 Spring 框架进行无缝集成以降低开发人员的学习成本。

上一页 下一页