链滴

如何编写一个 SkyWalking 插件

作者：vcjmhg

原文链接：https://ld246.com/article/1601898110516

来源网站：链滴

概述

之前几篇文章，我们着重介绍了在对SkyWalking进行二次开发之前的环境搭建问题，因此本篇文章基于SkyWalking-8.1.0版本，以开发webflux-webclent插件为例，分享一下对SkyWalking插件开发及贡献PR的过程（PR地址），以其能为大家了解SkyWalking java agent插件的开发有所帮助。

概念

span

Span应该是分布式链路追踪系统一个非常重要而且常见的一个概念。最早源自于Google Dapper 的文--Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,此处给出论文地址，感兴的小伙伴可以深入学习。简单来说，Span可以简单理解成一次服务的调用。只要是一个具有完整时周期的程序访问，都可以简单看做是一个span。

当然SkyWalking中的span与论文中的span类似，但同时也进行了一些扩展，具体来说，在SkyWalkig中span分成以下三种：

1. EntrySapn：代表服务提供者，也就是服务器的端点。我们可以简单理解成服务的提供方，比如对提供服务的Webflux服务或者MQ的消费则都是EntrySpan。

2. ExitSpan：代表服务的消费者，比如一个服务的客户端或者消息队列的生产者都可以理解成一个ExiSpan。

3. LocalSpan：与前边的EntrySpan和ExitSpan相比，LocalSpan的概念就比较特殊了，它其实本身远程服务调用没有任何关系，它更多的可能指代的的本地的java方法。它的出现可能是为了解决SkyWlking监控本地方法调用的问题。比如说，我们想知道某个本地方法的调用请求，我们便可以将该方法义成一个LocalSpan，然后OAP端便可以收集到对应的span信息，然后在web端清晰的展示该方法的用情况。

上下文载体（ContextCarrier）

因为分布式追踪，大部分情况下都是跨进程的，因此为了解决跨进程的链路绑定问题，SkyWalking入了ContextCarrier的概念。

以下是有关如何在 A -> B 分布式调用中使用 ContextCarrier 的步骤.

1. 在客户端, 创建一个新的空的 ContextCarrier.

2. 通过 ContextManager#createExitSpan 创建一个 ExitSpan 或者使用 ContextManager#inject 来初始化 ContextCarrier.

3. 将 ContextCarrier 所有信息放到请求头 (如 HTTP HEAD), 附件(如 Dubbo RPC 框架), 或者消息如 Kafka) 中

4. 通过服务调用, 将 ContextCarrier 传递到服务端.在服务端, 在对应组件的头部, 附件或消息中获取ontextCarrier 所有内容.

5. 通过 ContestManager#createEntrySpan 创建 EntrySpan 或者使用 ContextManager#extract 来绑定服务端和客户端.

异步API

因为官方关于插件具体的开发是给了比较详细的开发文档的（戳这里）point_left，因此我在此针对API部分就不详细来说了，我会重点介绍几个自己在开发webflux webclient的过程中用到的异步PI。

因为此次是对webflux WebClient来开发插件，许多方法的调用都需要时跨线程的因此，我们需要使异步API。

简单来说异步API的使用步骤如下：

1. 在原始上下文中调用 AsyncSpan#PrepareForAsync;

2. 将该Span传递到其他线程，并江湾城相关属性比如tag、log、status code等属性进行设置；

3. 全部操作就绪之后，可在任意线程中调用 #asyncFinish结束调用

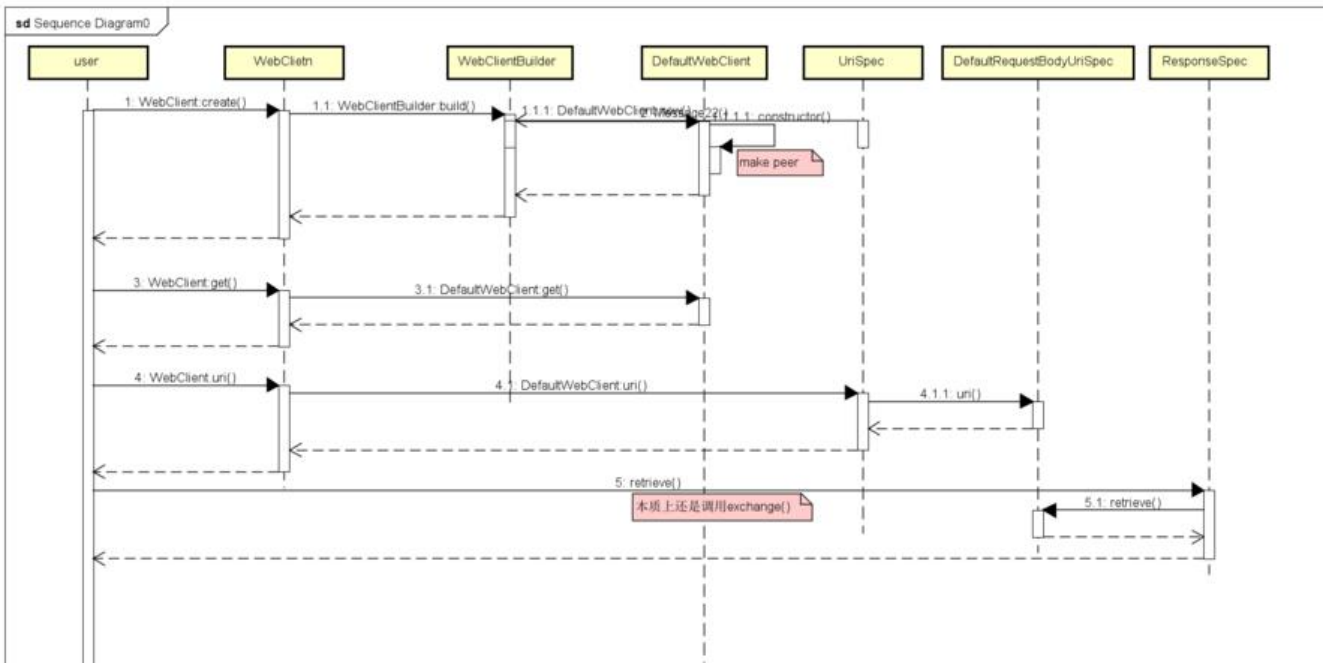
4. 当所有的 #prepareForAsync完成之后，追踪上下文就会结束，并一起被会传到后端服务（根据AI的执行次数来进行判断）。

插件编写

确定拦截点

插件本身的开发肯定有一定的业务的逻辑，因此我们在开发之前需要根据插件的业务逻辑的确定合适插入点位置。以webflux-webclient-plugin为例，因为该插件本质上是为了获取webclient在发起请时的调用信息，因此在确定插入点之前我们首先要分析，它整个的调用过程是怎么的。

因此我对WebClient从发起请求到获得相应整个过程进行了分析，画出了如下的：



分析整个过程，我发现，无论WebClient调用的是retrieve()方法还是调用的exchange()方法，最终发起请求的时候都是通过org.springframework.web.reactive.function.client.ExchangeFunctions$efaultExchangeFunction的exchange()方法实际执行异步请求，并且返回一个Mono<ClientRespone>类型的响应结果。

因此我们考虑使用DefalutExchangeFunction#exchange()方法作为插入点方法，但仅仅使用这一个入点是否是足够的哪?

这里我们先留下一个小小的悬念，在业务代码开发部分，我会详细讲解自己在开发过程中所遇到的坑!

拦截与业务代码开发

在插入点进行确定之后，我们便可以结合业务逻辑开始代码部分的开发。

定义拦截点

前边我们已经确定出了具体的拦截点，下边我们需要在插件目录中定义出该拦截点。

在创建的插件目录的Resourse目录，定义一个skywalking-plugin.def文件，添加插件定义:

```
spring-webflux-5.x-webclient=org.apache.skywalking.apm.plugin.spring.webflux.v5.webclient.efine.BodyInserterRequestInstrumentation
```

在define目录下创建Instrumentation类，以webflux-webclient插件为例，我创建了一个WebFluxWbClientInterceptor类，用来指定拦截点的具体方法。

具体代码如下所示:

```
public class WebFluxWebClientInstrumentation extends ClassEnhancePluginDefine {
    private static final String ENHANCE_CLASS = "org.springframework.web.reactive.function.client.ExchangeFunctions$DefaultExchangeFunction";
    private static final String INTERCEPT_CLASS = "org.apache.skywalking.apm.plugin.spring.wbflux.v5.webclient.WebFluxWebClientInterceptor";
```

```
@Override
protected ClassMatch enhanceClass() {
    return NameMatch.byName(ENHANCE_CLASS);
}

@Override
public ConstructorInterceptPoint[] getConstructorsInterceptPoints() {
    return new ConstructorInterceptPoint[0];
}

@Override
public InstanceMethodsInterceptPoint[] getInstanceMethodsInterceptPoints() {
    return new InstanceMethodsInterceptPoint[]{
            new InstanceMethodsInterceptPoint() {
                @Override
                public ElementMatcher<MethodDescription> getMethodsMatcher() {
                    return named("exchange");
                }

                @Override
                public String getMethodsInterceptor() {
                    return INTERCEPT_CLASS;
                }

                @Override
                public boolean isOverrideArgs() {
                    return false;
                }
            }
    };
}

@Override
public StaticMethodsInterceptPoint[] getStaticMethodsInterceptPoints() {
    return new StaticMethodsInterceptPoint[0];
}
```

实现对应的Interceptor类

有了插入点之后，我们还需要通过一个类来对插入点方法做具体增强的工作，因此我们定义了一个WebFluxWebClientInstrumentation类用来做具体的方法增强工作。

具体来说，**在该类中做了如下操作：**

1. 获取请求参数，收集链路信息
2. 创建ContextCarrier，为进程的数据管理做准备。
3. 创建ExitSpan
4. 设置span相关信息，比如请求方法的类型、访问的url等内容
5. 将ContextCarrier对象进行动态传递，传递给第二个插入点增强类

6. 将当前span进行传递，便于后续对响应信息进行判断和设置

具体代码如下(org.apache.skywalking.apm.plugin.spring.webflux.v5.webclient包下WebFluxWebClientInterceptor类)。

同时，我在后续调试的过程中发现，只定义一个拦截点是不够的，因为request只有在初始化的过程才能被操作，也就是是说，在该位置违法将span的相关信息放置到request的头文件中，进行跨链传。

因此我在org.springframework.http.client.reactive.ClientHttpRequest的构造方法处也设置了一个截点，负责讲span信息放置到request中进行跨链传输。

具体实现如下所示：

```
public void beforeMethod(EnhancedInstance objInst, Method method, Object[] allArguments,
Class<?>[] argumentsTypes,
                         MethodInterceptResult result) throws Throwable {
    ClientHttpRequest clientHttpRequest = (ClientHttpRequest) allArguments[0];
    ContextCarrier contextCarrier = (ContextCarrier) objInst.getSkyWalkingDynamicField();
    CarrierItem next = contextCarrier.items();
    while (next.hasNext()) {
        next = next.next();
        clientHttpRequest.getHeaders().set(next.getHeadKey(), next.getHeadValue());
```

编写测试用例

在插件编写完成之后，我们还需要编写一个测试用例用来做CI测试。插件开发的详细文档可以参考戳下point_left

此处我就简单说一下用例的编写流程。

用例工程是一个独立的Maven工程。该工程能将工程打包镜像, 并要求提供一个外部能够访问的Web务用例测试调用链追踪。

用例工程的目录图如下所示：

```
[plugin_testcase]
  |__ [config]
  | |__ docker-compse.yml
  | |__ expectedData.yaml
  |__ [src]
  | |__ [main]
  | |    ...
  | |__ [resources]
  | |    ...
  |__ pom.xml
  |__ testcase.yml

[] = directory
```

文件用途说明

以下是用例工程中配置文件的说明：

文件	用途
docker-compose.xml	定义用例的docker运行容环境
expectedData.yaml	定义用例期望生成的Segment的数据
testcase.yml	定义用例的基本信息，如: 被测试架名称、版本号

测试用例编写流程

1. 编写用例代码
2. 打包并测试用例镜像,确保在没有加载探针时的用例镜像能够正常运行
3. 编写期望数据文件
4. 编写用例配置文件
5. 测试用例

Pull Request

提交前的检查

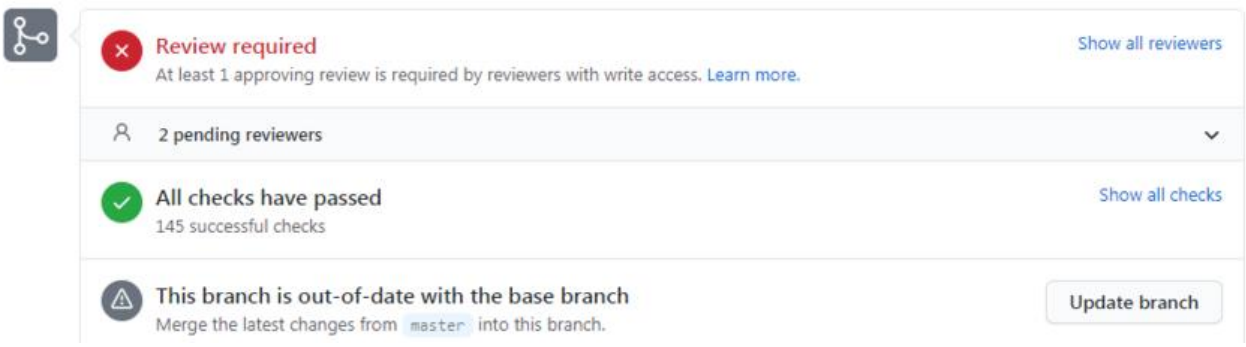
1. 在正式提交以前一定要保证集成测试在本地通过
2. 更新插件文档
   - 插件文档需要更新：Supported-list.md相关插件信息的支持。
   - 插件如果为可选插件需要在agent-optional-plugins可选插件文档中增加对应的描述。

提交PR

在提交PR时，一定要简要描述个人对插件的设计思路，这样有助于社区贡献者讨论完成codereview。

申请自动化测试

测试用例编写完成后，可以申请自动化测试，了解插件的兼容性等问题



原文链接：如何编写一个 SkyWalking 插件

在自动化测试完成之后，会有社区成员进行代码审查，审查通过后，不出意外最终会被合并到主分支。

自己在开发过程中遇到的问题

1. 在搭建开发环境，完成项目的导入工作之后，maven总报错。

解决方法：增加了国内的多个maven源之后该问题被解决

2. 在确定插入点exchange()方法之后，在调试过程中无法被拦截。

解决方法由于选择的增强类属于内部类，因此在DefaultExchangeFunction，因此在选择该类作为内类的时候应该使用#进行连接，而不是通过.。即应该写成org.springframework.web.reactive.functio.client.ExchangeFunctions$DefaultExchangeFunction的形式。

3. 在插件基本功能编写完成后，OAP端却无法收集到链路信息。

解决方法： 使用最新的OAP收集端程序来进行接收。之前一直使用的本地直接编译的OAP端，发现能工作，使用编译好的OAP端代码版本过低时也不能使用。

4. 同一服务的两个span不能够串联。

原因分析： 经过分析出现该问题的原因主要是关闭span的时机不对。由于使用的是异步接口，因此关闭span的时候必须在doFinally()方法体内进行关闭。防治span提前关闭，从而出现同一服务的spa不能串联的情况发成

解决方法： 修改span的关闭时机，在doFinally()方法体中执行span.asyncFinish()方法

5. 在本地跑集成测试时，遇到无法启动docker的问题。

原因分析： 根据保存内容发现是测试脚本在启动docker的过程中出现权限不足的问题，可能是docke的使用需要用的root的权限。

解决方法： 将当前用户增加到docker的用户组中，从而使得当前的用户具有操作docker的权限。

6. 在集成测试阶段出现SegementNotFoundException问题

原因分析：该问题的出现主要是在对Segment进行验证的过程中，发现Segement丢失的情况发生

解决方法： 该问题在经过深入分析之后发现，实际上就是因为在编写插件的时候，插入点选择不充分致的。exchange()这个插入点可以用来收集信息，但却无法用来进行链路信息绑定。因此后续重新设了插件的插入点，增加了第二个插入点，并且在第二个插入点位置进行链路的绑定，至此问题解决。