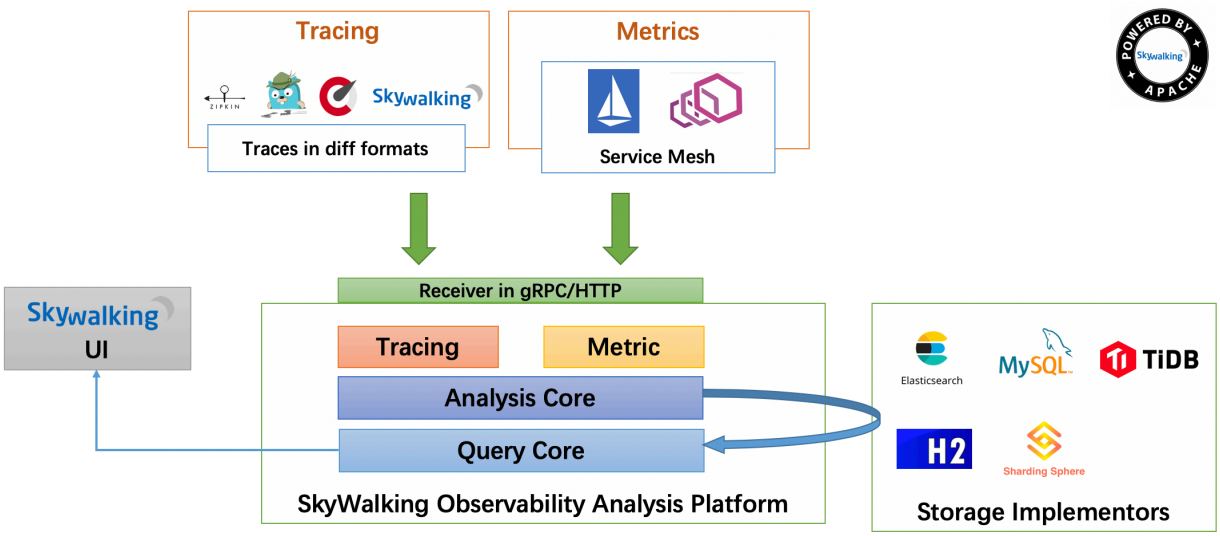
Skywalking数据采集与收集源码分析

skywalking作为一个分布式APM（应用性能管理）系统，目前被广泛使用在各种系统中。

skywalking官网：https://skywalking.apache.org/

skywalking的架构图如下：



Skywalking的agent负责采集数据，发送到collector，collector聚合，并且存储这些数据，且提供了一个简洁使用的UI端，可共我们查看监控的指标。

下面我们来开始分析skywalking的源码。

下载源码并构建

因为skywalking为了实现高性能通信，采用的是grpc的方式来实现服务器与客户端的数据传输的，所以导入之后我们需要稍微做一些事情，我们可以参考`docs/en/guides/How-to-build.md`这篇文档来构建。

打包构建

我们可以在github上面将skywalking源码fork一份，然后下载到自己的本地。

```
// 直接
git clone --recurse-submodules https://github.com/apache/skywalking.git
// 或者
git clone https://github.com/apache/skywalking.git
cd skywalking/
git submodule init
git submodule update
```

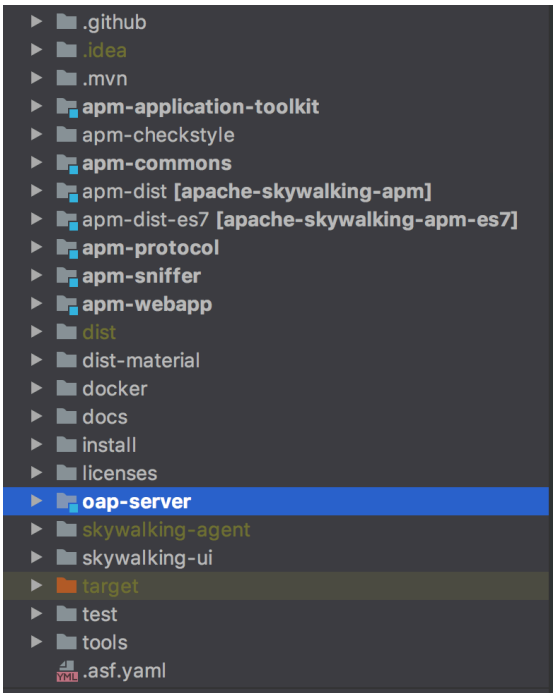
执行命令：

```
./mvnw clean package -DskipTests
```

最终打好的包在dist目录下面

在IDEA里面构建源码

用IDEA打开skywalking项目（作为maven项目导入）

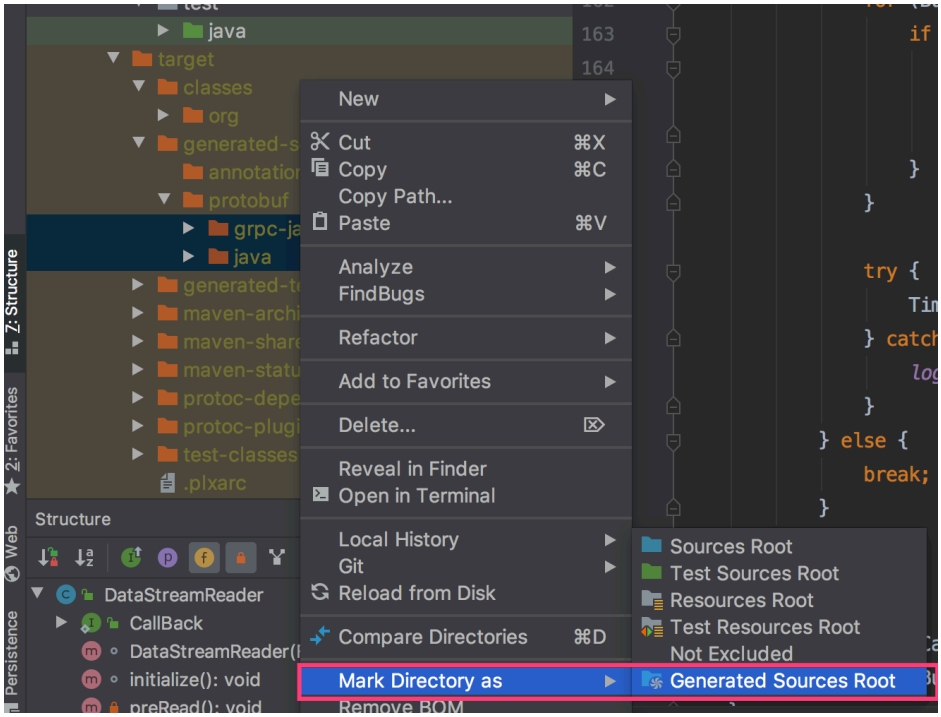


然后在skywalking所在目录命令行运行脚本去编译构建（因为skywalking用到了grpc）：

```
./mvnw compile -Dmaven.test.skip=false
```

然后查看设置生成的源代码（主要是看potobuf文件编译生成的源代码）

- **apm-protocol/apm-network/target/generated-sources/protobuf** 选中这个目录下面的`grpc-java`和`java`，然后右键选择`Mark Directory As`-->`Generated Sources Root`如下图所示



- **oap-server/server-core/target/generated-sources/protobuf**目录的`grpc-java` 和 `java` 文件夹Mark Directory As `-->` Generated Sources Root`
- **oap-server/server-receiver-plugin/receiver-proto/target/generated-sources/protobuf** 目录的`grpc-java` 和 `java` 文件夹Mark Directory As `-->` Generated Sources Root`
- **oap-server/exporter/target/generated-sources/protobuf**目录的`grpc-java` 和 `java` 文件夹Mark Directory As `-->` Generated Sources Root`
- **oap-server/server-configuration/grpc-configuration-sync/target/generated-sources/protobuf**目录的`grpc-java` 和 `java` 文件夹Mark Directory As `-->` Generated Sources Root`
- **oap-server/oal-grammar/target/generated-sources**目录的`grpc-java` 和 `java` 文件夹 Mark Directory As `-->` Generated Sources Root`

在Eclipse里面构建源码

1、按照maven项目导入到eclipse中

2、添加一下内容到`skywalking/pom.xml`中

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>1.8</version>
    <executions>
        <execution>
            <id>add-source</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
```

```
                <sources>
                    <source>src/java/main</source>
                    <source>apm-protocol/apm-network/target/generated-
sources/protobuf</source>
                    <source>apm-collector/apm-collector-remote/collector-
remote-grpc-provider/target/generated-sources/protobuf</source>
                </sources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

3、添加如下内容，使得eclipse的M2e插件能够支持扩展配置

```
<pluginManagement>
    <plugins>
    <!--This plugin's configuration is used to store Eclipse m2e settings
    only. It has no influence on the Maven build itself. -->
        <plugin>
            <groupId>org.eclipse.m2e</groupId>
            <artifactId>lifecycle-mapping</artifactId>
            <version>1.0.0</version>
            <configuration>
                <lifecycleMappingMetadata>
                    <pluginExecutions>
                        <pluginExecution>
                            <pluginExecutionFilter>
                                <groupId>org.codehaus.mojo</groupId>
                                <artifactId>build-helper-maven-
plugin</artifactId>
                                <versionRange>[1.8,)</versionRange>
                                <goals>
                                    <goal>add-source</goal>
                                </goals>
                            </pluginExecutionFilter>
                        </pluginExecution>
                    </pluginExecutions>
                </lifecycleMappingMetadata>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>
```

4、`apm-collector-remote/collector-remote-grpc-provider/pom.xml`文件中添加如下依赖

```
<dependency>
   <groupId>com.google.guava</groupId>
   <artifactId>guava</artifactId>
   <version>24.0-jre</version>
</dependency>
```

5、执行命令

```
./mvnw compile -Dmaven.test.skip=true
```

6、执行命令

先执行maven clean，然后`maven update`

7、执行命令：

```
./mvnw compile
```

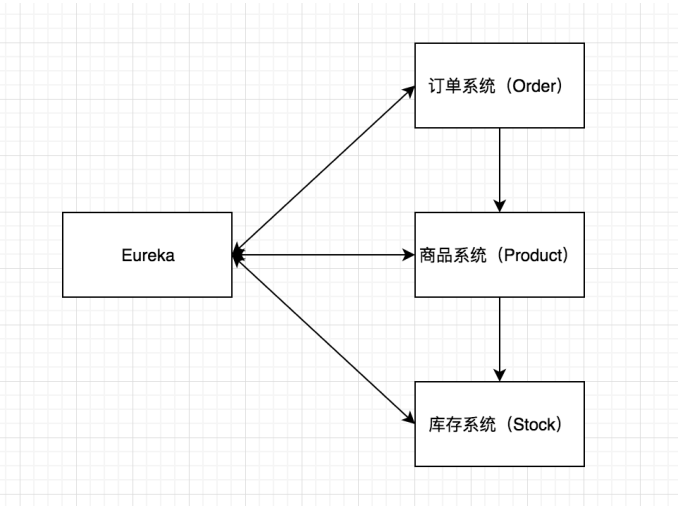
8、刷新项目

源码分析

skywalking的分布式链路追踪流程大致如下：

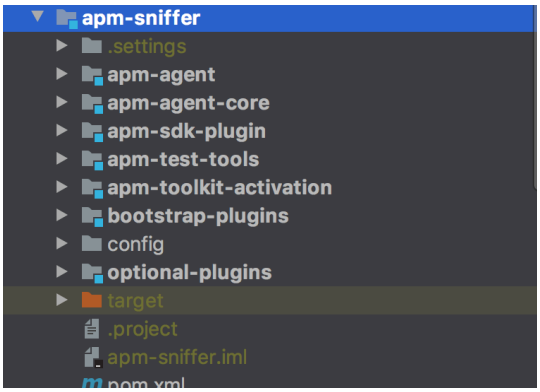
1. Agent采集数据
2. Agent发送数据到Collector
3. Collector接收数据
4. Collector将接收的数据存储到持久层

我们这里主要探探Agent采集Java类系统的数据比如spring等，以spring cloud的一个简易的分布式系统来讲：



这是一个建议的订单系统，有`Eureka`，`Order`，`Product`，`Stock`，当下订单的时候，order会调用product，product又会调用stock。

我们从apm-sniffer工程开始出发（sniffer即嗅探器、探针的意思）



apm-agent工程

我们看到这个工程只有一个类

`org.apache.skywalking.apm.agent.SkyWalkingAgent`

这个类有一个方法：

```
/**
 * 主入口，使用byte-buddy来实现来增强插件里面定义的所有的类。
 */
public static void premain(String agentArgs, Instrumentation instrumentation)
throws PluginException, IOException {

}
```

Agent采集数据

我们这里主要介绍JVM的数据和spring相关的数据

JVM的数据

我们看到在apm-agent-core里面有
类：`org.apache.skywalking.apm.agent.core.jvm.JVMService`

这个类实现了`BootService`和`java.lang.Runnable`接口，而这个类是怎么执行里面的一些方法的呢？在`apm-agent-core`这个工程的`/src/main/resources/META-INF/services/org.apache.skywalking.apm.agent.core.boot.BootService`文件里面有很多类的全限定名信息：

```
org.apache.skywalking.apm.agent.core.remote.TraceSegmentServiceClient
org.apache.skywalking.apm.agent.core.context.ContextManager
org.apache.skywalking.apm.agent.core.sampling.SamplingService
org.apache.skywalking.apm.agent.core.remote.GRPCChannelManager
org.apache.skywalking.apm.agent.core.jvm.JVMService
org.apache.skywalking.apm.agent.core.remote.ServiceAndEndpointRegisterClient
org.apache.skywalking.apm.agent.core.context.ContextManagerExtendService
org.apache.skywalking.apm.agent.core.commands.CommandService
org.apache.skywalking.apm.agent.core.commands.CommandExecutorService
org.apache.skywalking.apm.agent.core.context.OperationNameFormatService
```

而这每个类都实现了`BootService`这个借口，`BootService`是所有当插件机制开始起作用时需要启动的远程交换需要实现的接口。`BootService`启动的时候将调用`boot`方法。

`org.apache.skywalking.apm.agent.core.boot.ServiceManager`这个类里面会将所有实现`BootService`的类的实例都执行一遍。

`JVMService`类实例化后执行的`boot`方法内容如下

```
@Override
public void boot() throws Throwable {
   // 创建一个持续收集（生产）指标的单一线程的线程池，这个线程池会定期（每秒）执行，而且执行的是JVMService的run方法
    collectMetricFuture = Executors
        .newSingleThreadScheduledExecutor(new DefaultNamedThreadFactory("JVMService-produce"))
        .scheduleAtFixedRate(new RunnableWithExceptionProtection(this, new RunnableWithExceptionProtection.CallbackWhenException() {
            @Override public void handle(Throwable t) {
                logger.error("JVMService produces metrics failure.", t);
            }
        }), 0, 1, TimeUnit.SECONDS);
   // 创建一个持续发送（消费）数据的单一线程的线程池，这个线程池会定期（每秒）执行，而且执行的是JVMService的内部类Sender的run方法
    sendMetricFuture = Executors
        .newSingleThreadScheduledExecutor(new DefaultNamedThreadFactory("JVMService-consume"))
        .scheduleAtFixedRate(new RunnableWithExceptionProtection(sender, new RunnableWithExceptionProtection.CallbackWhenException() {
            @Override public void handle(Throwable t) {
                logger.error("JVMService consumes and upload failure.", t);
            }
```

```
        }
        ), 0, 1, TimeUnit.SECONDS);
}
```

`JVMService`类的`run`方法：

```
  public void run() {
      if (RemoteDownstreamConfig.Agent.SERVICE_ID != DictionaryUtil.nullValue()
          && RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID !=
DictionaryUtil.nullValue()
      ) {
          long currentTimeMillis = System.currentTimeMillis();
          try {
              JVMMetric.Builder jvmBuilder = JVMMetric.newBuilder();
              jvmBuilder.setTime(currentTimeMillis);
              jvmBuilder.setCpu(CPUProvider.INSTANCE.getCpuMetric());

 jvmBuilder.addAllMemory(MemoryProvider.INSTANCE.getMemoryMetricList());

 jvmBuilder.addAllMemoryPool(MemoryPoolProvider.INSTANCE.getMemoryPoolMetricsLi
st());
              jvmBuilder.addAllGc(GCProvider.INSTANCE.getGCList());

            // JVM指标数据
              JVMMetric jvmMetric = jvmBuilder.build();
            // 收集数据后，放到消息队列LinkedBlockingQueue<JVMMetric> queue中
              if (!queue.offer(jvmMetric)) {
                  queue.poll();
                  queue.offer(jvmMetric);
              }
          } catch (Exception e) {
              logger.error(e, "Collect JVM info fail.");
          }
      }
  }
```

内部Sender类的run方法：

```
  @Override
  public void run() {
      if (RemoteDownstreamConfig.Agent.SERVICE_ID != DictionaryUtil.nullValue()
          && RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID !=
DictionaryUtil.nullValue()
      ) {
          if (status == GRPCChannelStatus.CONNECTED) {
              try {
                  JVMMetricCollection.Builder builder =
JVMMetricCollection.newBuilder();
```

```
                    LinkedList<JVMMetric> buffer = new LinkedList<JVMMetric>();
                    queue.drainTo(buffer);
                    if (buffer.size() > 0) {
                        builder.addAllMetrics(buffer);

 builder.setServiceInstanceId(RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID)
;
                      // 发送数据并接收返回的结果
                        Commands commands =
stub.withDeadlineAfter(GRPC_UPSTREAM_TIMEOUT,
TimeUnit.SECONDS).collect(builder.build());

 ServiceManager.INSTANCE.findService(CommandService.class).receiveCommand(comma
nds);
                    }
                } catch (Throwable t) {
                    logger.error(t, "send JVM metrics to Collector fail.");
                }
            }
        }
    }
```

而具体数据是怎么发送的呢？我们来看采集的指标类JVMMetric.java

```
public  final class JVMMetric extends
    com.google.protobuf.GeneratedMessageV3 implements
    // @@protoc_insertion_point(message_implements:JVMMetric)
    JVMMetricOrBuilder {
    ...
}
```

其实这个类是JVMMetric.proto编译后生成的，而JVMMetric.proto内容如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.skywalking.apm.network.language.agent.v2";
option csharp_namespace = "SkyWalking.NetworkProtocol";

import "common/common.proto";
import "common/JVM.proto";

service JVMMetricReportService {
    // grpc定义的方法，参数类型JVMMetricCollection，返回类型为：Commands
    rpc collect (JVMMetricCollection) returns (Commands) {
    }
}
```

```
message JVMMetricCollection {
    repeated JVMMetric metrics = 1;
    int32 serviceInstanceId = 2;
}
```

common.proto内容如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.skywalking.apm.network.common";
option csharp_namespace = "SkyWalking.NetworkProtocol";

message KeyStringValuePair {
    string key = 1;
    string value = 2;
}

message KeyIntValuePair {
    string key = 1;
    int32 value = 2;
}

message CPU {
    double usagePercent = 2;
}

// In most cases, detect point should be `server` or `client`.
// Even in service mesh, this means `server`/`client` side sidecar
// `proxy` is reserved only.
enum DetectPoint {
    client = 0;
    server = 1;
    proxy = 2;
}

message Commands {
    repeated Command commands = 1;
}

message Command {
    string command = 1;
    repeated KeyStringValuePair args = 2;
}

enum ServiceType {
```

```
    // An agent works inside the normal business application.
    normal = 0;
    // An agent works inside the database.
    database = 1;
    // An agent works inside the MQ.
    mq = 2;
    // An agent works inside the cache server.
    cache = 3;
    // An agent works inside the browser.
    browser = 4;
}
```

jvm.proto内容如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.skywalking.apm.network.language.agent";
option csharp_namespace = "SkyWalking.NetworkProtocol";

import "common/common.proto";

message JVMMetric {
    int64 time = 1;
    CPU cpu = 2;
    repeated Memory memory = 3;
    repeated MemoryPool memoryPool = 4;
    repeated GC gc = 5;
}

message Memory {
    bool isHeap = 1;
    int64 init = 2;
    int64 max = 3;
    int64 used = 4;
    int64 committed = 5;
}

message MemoryPool {
    PoolType type = 1;
    int64 init = 2;
    int64 max = 3;
    int64 used = 4;
    int64 commited = 5;
}

enum PoolType {
```

```
    CODE_CACHE_USAGE = 0;
    NEWGEN_USAGE = 1;
    OLDGEN_USAGE = 2;
    SURVIVOR_USAGE = 3;
    PERMGEN_USAGE = 4;
    METASPACE_USAGE = 5;
}

message GC {
    GCPhrase phrase = 1;
    int64 count = 2;
    int64 time = 3;
}

enum GCPhrase {
    NEW = 0;
    OLD = 1;
}
```

而服务接收端，即collector是怎么接收的呢?

接收端有一个类`JVMMetricsServiceHandler`专门用来处理JVM的监控数据，这个类的`collect`方法如下:

```
  @Override public void collect(JVMMetrics request, StreamObserver<Downstream>
responseObserver) {
      int serviceInstanceId = request.getApplicationInstanceId();

      if (logger.isDebugEnabled()) {
          logger.debug("receive the jvm metrics from service instance, id: {}",
serviceInstanceId);
      }

      // 处理数据，jvmSourceDispatcher发送到下一环节处理
      request.getMetricsList().forEach(metrics -> {
          long minuteTimeBucket =
TimeBucket.getMinuteTimeBucket(metrics.getTime());
          jvmSourceDispatcher.sendMetric(serviceInstanceId, minuteTimeBucket,
metrics);
      });

      responseObserver.onNext(Downstream.newBuilder().build());
      responseObserver.onCompleted();
  }
```

然后我们看一下JVMSourceDispatcher的sendMetric方法

```
void sendMetric(int serviceInstanceId, long minuteTimeBucket, JVMMetric
metrics) {
    ServiceInstanceInventory serviceInstanceInventory =
instanceInventoryCache.get(serviceInstanceId);
    int serviceId;
    if (Objects.nonNull(serviceInstanceInventory)) {
        serviceId = serviceInstanceInventory.getServiceId();
    } else {
        logger.warn("Can't find service by service instance id from cache,
service instance id is: {}", serviceInstanceId);
        return;
    }

    this.sendToCpuMetricProcess(serviceId, serviceInstanceId, minuteTimeBucket,
metrics.getCpu());
    this.sendToMemoryMetricProcess(serviceId, serviceInstanceId,
minuteTimeBucket, metrics.getMemoryList());
    this.sendToMemoryPoolMetricProcess(serviceId, serviceInstanceId,
minuteTimeBucket, metrics.getMemoryPoolList());
    this.sendToGCMetricProcess(serviceId, serviceInstanceId, minuteTimeBucket,
metrics.getGcList());
}
```

然后我们看sendTopCpuMetricProcess方法

```
  private void sendToCpuMetricProcess(int serviceId, int serviceInstanceId,
long timeBucket, CPU cpu) {
      ServiceInstanceJVMCPU serviceInstanceJVMCPU = new
ServiceInstanceJVMCPU();
      serviceInstanceJVMCPU.setId(serviceInstanceId);
      serviceInstanceJVMCPU.setName(Const.EMPTY_STRING);
      serviceInstanceJVMCPU.setServiceId(serviceId);
      serviceInstanceJVMCPU.setServiceName(Const.EMPTY_STRING);
      serviceInstanceJVMCPU.setUsePercent(cpu.getUsagePercent());
      serviceInstanceJVMCPU.setTimeBucket(timeBucket);
      sourceReceiver.receive(serviceInstanceJVMCPU);
  }
```

`SourceReceiver`的`receive`来接收数据

然而`SourceReceiver`是一个接口

```
public interface SourceReceiver extends Service {
    void receive(Source source);
}
```

这个接口只有一个实现类

```
public class SourceReceiverImpl implements SourceReceiver {
}
```

receive的实现：

```
@Override public void receive(Source source) {
    dispatcherManager.forward(source);
}
```

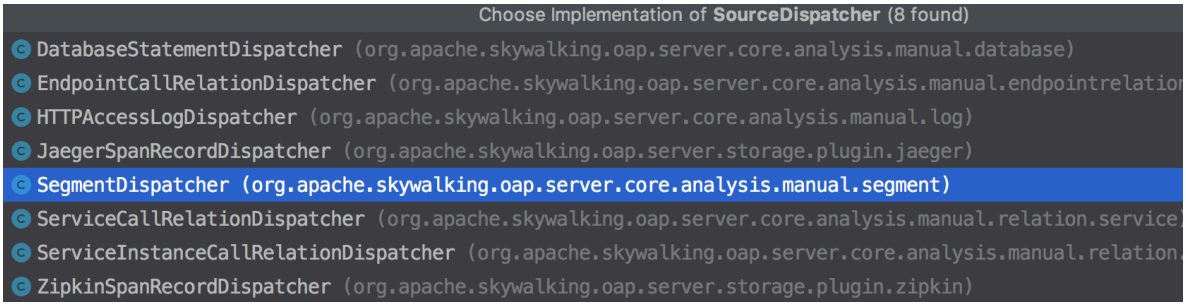
我们看到又调用了`DispatcherManager`的`forward`方法

```
public void forward(Source source) {
    if (source == null) {
        return;
    }

    List<SourceDispatcher> dispatchers = dispatcherMap.get(source.scope());

    /**
     * Dispatcher is only generated by oal script analysis result.
     * So these will/could be possible, the given source doesn't have the
dispatcher,
     * when the receiver is open, and oal script doesn't ask for analysis.
     */
    if (dispatchers != null) {
        for (SourceDispatcher dispatcher : dispatchers) {
            dispatcher.dispatch(source);
        }
    }
}
```

然后会调用`SourceDispatcher`的`dispatch`方法



而我们看到有这么多类实现了`SourceDispatcher`接口，具体是那个方法实现了呢？我们可以打日志也可以简单分析一下，首先可以排除的
书`EndpointCallRElationDispatcher`、`HttpAccessLogDispatcher`、

`JaegerSpanRecordDispatcher`、`ServiceCallRelationDispatcher`、`ServiceInstanceCallRelationDispatcher`、`ZipkinSpanRecordDispatcher`这几个类也就是说我们可以重点关注

DatabaseStatementDispatcher和SegmentDispatcher这两个类，而这个`DatabaseStatementDispatcher`并没有被使用，所以我们可以重点分析`SegmentDispatcher`这个类

```
public class SegmentDispatcher implements SourceDispatcher<Segment> {

    @Override public void dispatch(Segment source) {
        SegmentRecord segment = new SegmentRecord();
        segment.setSegmentId(source.getSegmentId());
        segment.setTraceId(source.getTraceId());
        segment.setServiceId(source.getServiceId());
        segment.setServiceInstanceId(source.getServiceInstanceId());
        segment.setEndpointName(source.getEndpointName());
        segment.setEndpointId(source.getEndpointId());
        segment.setStartTime(source.getStartTime());
        segment.setEndTime(source.getEndTime());
        segment.setLatency(source.getLatency());
        segment.setIsError(source.getIsError());
        segment.setDataBinary(source.getDataBinary());
        segment.setTimeBucket(source.getTimeBucket());
        segment.setVersion(source.getVersion());
        // 构造SegmentRecord对象，然后RecordStreamProcessor的in方法去处理（消费）
segment信息
        RecordStreamProcessor.getInstance().in(segment);
    }
}
```

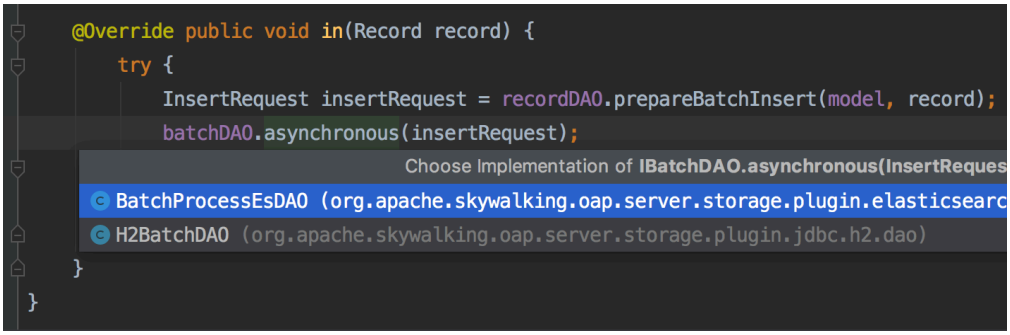
然后我们看一下`RecordStreamProcessor`的in方法

```
public void in(Record record) {
    RecordPersistentWorker worker = workers.get(record.getClass());
    if (worker != null) {
        worker.in(record);
    }
}
```

然后是`RecordPersistentWorker`的in方法

```
@Override public void in(Record record) {
    try {
        InsertRequest insertRequest = recordDAO.prepareBatchInsert(model,
record);
        batchDAO.asynchronous(insertRequest);
    } catch (IOException e) {
        logger.error(e.getMessage(), e);
    }
}
```

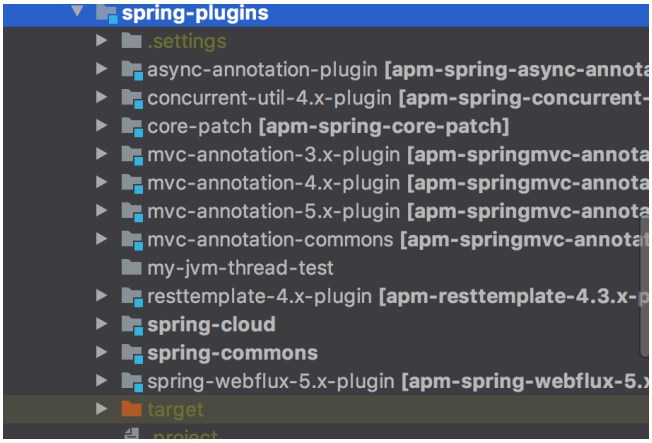
到此我们能够看到持久化到数据库的操作（调用es或者h2的相关接口实现）



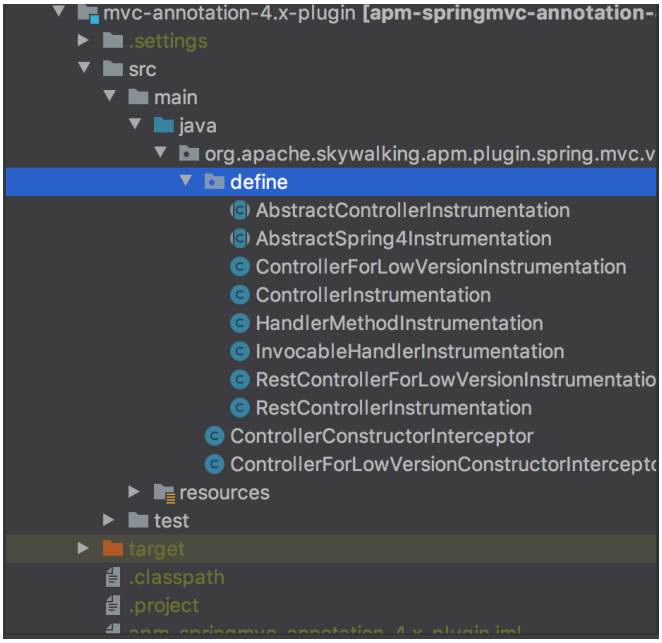
整个过程我们看到JVM的数据是在agent发送到collector后立马就持久化存储了（h2或者es）。

插件源码分析

关于插件开发，我们可以参考**Java-Plugin-Development-Guide.md**这篇文档，或者我翻译过来后的中文文档，接下来我们看看spring框架的数据是如何采集的，在`apm-sniffer/apm-sdk-plugin`目录下，有个字项目`spring-plugins`里面放的都是spring相关的插件用来实现spring框架的数据采集



我们以`mvc-annotation-4.x-plugin`项目为例来看，skywalking的插件是如何开发的。



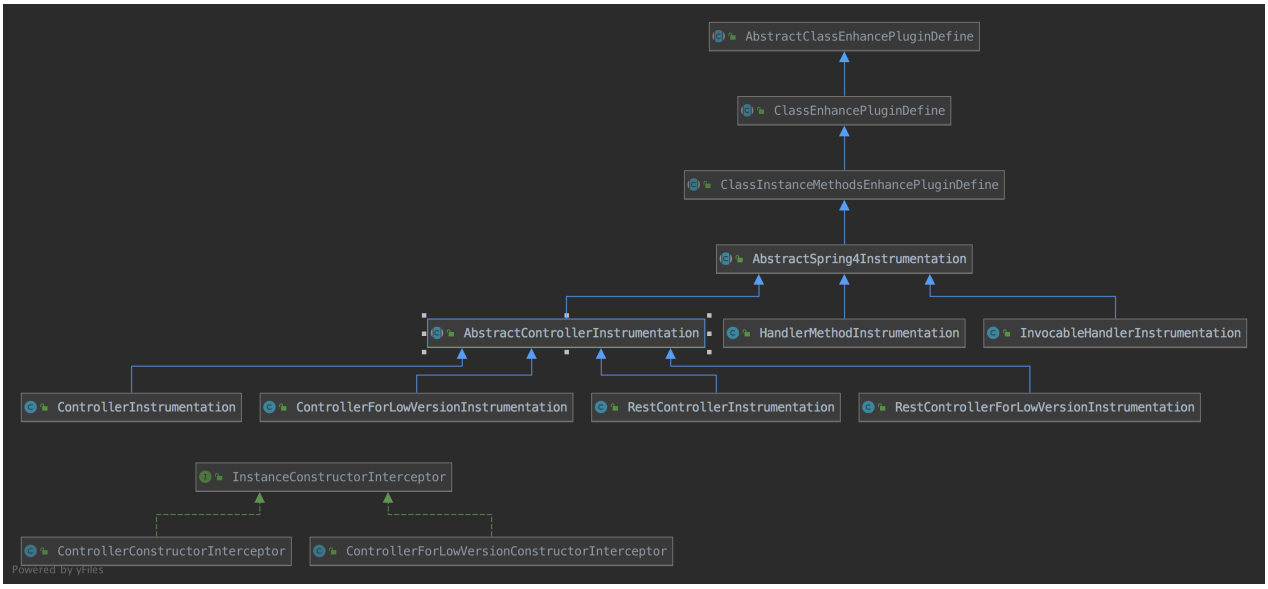
我们可以看到resources目录的文件`src/main/resources/skywalking-plugin.def`这个skywalking-plugin.def就是用来定义插件的。

```
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.ControllerInstrumenta
tion
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.RestControllerInstrum
entation
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.HandlerMethodInstrume
ntation
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.InvocableHandlerInstr
umentation
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.ControllerForLowVersi
onInstrumentation
spring-mvc-annotation-
4.x=org.apache.skywalking.apm.plugin.spring.mvc.v4.define.RestControllerForLowV
ersionInstrumentation
```

这个里面定义了一下几个插件

- ControllerInstrumentation
- RestControllerInstrumentation
- HandlerMethodInstrumentation
- InvocableHandlerInstrumentation
- ControllerForLowVersionInstrumentation
- RestControllerForLowVersionInstrumentation

我们根据plugin的开发流程来分析代码，应该是有一个类定义拦截机制，另外一个类是增强机制。我们先看一下类的结构图：



我们看到`AbstractClassEnhancePluginDefine`、`ClassEnhancePluginDefine`、`ClassInstanceMethodsEnhancePluginDefine`都是skywalking提供的基础类，而这个插件里面的类增强都是继承这些父类的。

我们先来看`org.apache.skywalking.apm.plugin.spring.mvc.v4.define.ControllerInstrumentation`，我们先看抽象类`AbstractSpring4Instrumentation`的内容：

```
public abstract class AbstractSpring4Instrumentation extends
ClassInstanceMethodsEnhancePluginDefine {
    // 这块个人感觉应该是witness_class写错了，
    public static final String WITHNESS_CLASSES =
"org.springframework.cache.interceptor.SimpleKey";

    @Override
    protected String[] witnessClasses() {
        return new String[] {WITHNESS_CLASSES,
"org.springframework.cache.interceptor.DefaultKeyGenerator"};
    }
}
```

然后它的子类：

```
/**
 * ControllerInstrumentation 增强所有有RequestMapping注解和Controller注解的类的构造函数和方法
 * ControllerConstructorInterceptor 在执行构造函数之前将controller的base path（路径）放到动态
 * 字段里面
 * RequestMappingMethodInterceptor先从动态字段里面获取request path，如果没找到
```

```
 * RequestMappingMethodInterceptor会结合路径和当前方法上面的注解和base path将新的路径放到动态
 * 字段里面
 * @author zhangxin
 */
public abstract class AbstractControllerInstrumentation extends AbstractSpring4Instrumentation {
    // 构造函数拦截点
    @Override
    public ConstructorInterceptPoint[] getConstructorsInterceptPoints() {
        return new ConstructorInterceptPoint[] {
            new ConstructorInterceptPoint() {
                // 匹配方式，这里是返回了一个any()即总是匹配
                @Override
                public ElementMatcher<MethodDescription> getConstructorMatcher() {
                    return any();
                }

                // 拦截器类
                @Override
                public String getConstructorInterceptor() {
                    return "org.apache.skywalking.apm.plugin.spring.mvc.v4.ControllerConstructorInterceptor";
                }
            }
        };
    }

    // 实例方法拦截点，返回了一个数组，一个是针对@RequestMapping这种类型的注解，一个是针对
    // @GetMapping、@PostMapping、@PutMapping、@DeleteMapping、@PatchMapping这些类型的注解
    @Override
    public InstanceMethodsInterceptPoint[] getInstanceMethodsInterceptPoints() {
        return new InstanceMethodsInterceptPoint[] {
            new DeclaredInstanceMethodsInterceptPoint() {
                // 所有有RequestMapping这个注解的
                @Override
                public ElementMatcher<MethodDescription> getMethodsMatcher() {
                    return isAnnotatedWith(named("org.springframework.web.bind.annotation.RequestMapping"));
                }

                // RequestMappingMethodInterceptor
                @Override
                public String getMethodsInterceptor() {
```

```
                    return Constants.REQUEST_MAPPING_METHOD_INTERCEPTOR;
                }

                @Override
                public boolean isOverrideArgs() {
                    return false;
                }
            },
            new DeclaredInstanceMethodsInterceptPoint() {
                @Override
                public ElementMatcher<MethodDescription> getMethodsMatcher() {
                    return
isAnnotatedWith(named("org.springframework.web.bind.annotation.GetMapping"))

.or(isAnnotatedWith(named("org.springframework.web.bind.annotation.PostMapping"
)))

.or(isAnnotatedWith(named("org.springframework.web.bind.annotation.PutMapping")
))

.or(isAnnotatedWith(named("org.springframework.web.bind.annotation.DeleteMappin
g")))

.or(isAnnotatedWith(named("org.springframework.web.bind.annotation.PatchMapping
")));
                }

                // RestMappingMethodInterceptor
                @Override
                public String getMethodsInterceptor() {
                    return Constants.REST_MAPPING_METHOD_INTERCEPTOR;
                }

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

    // 需要增强的类的匹配方式
    @Override
    protected ClassMatch enhanceClass() {
        // 抽象类不定义具体匹配方式，而是交给子类，让子类去实现getEnhanceAnnotations方
法。
        return
ClassAnnotationMatch.byClassAnnotationMatch(getEnhanceAnnotations());
    }
```

```
    protected abstract String[] getEnhanceAnnotations();

}
```

`AbstractControllerInstrumentation`这个类并没有定义确定的类的匹配 然后是`ControllerInstrumentation`

```
public class ControllerInstrumentation extends
AbstractControllerInstrumentation {

    public static final String ENHANCE_ANNOTATION =
"org.springframework.stereotype.Controller";

    // 匹配所有有@Controller注解的类
    @Override protected String[] getEnhanceAnnotations() {
        return new String[] {ENHANCE_ANNOTATION};
    }
}
```

接下来我们来看构造函数的拦截器类`ControllerConstructorInterceptor`

```
/**
 * The <code>ControllerConstructorInterceptor</code> intercepts the
Controller's constructor, in order to acquire the
 * mapping annotation, if exist.
 *
 * But, you can see we only use the first mapping value, <B>Why?</B>
 *
 * Right now, we intercept the controller by annotation as you known, so we
CAN'T know which uri patten is actually
 * matched. Even we know, that costs a lot.
 *
 * If we want to resolve that, we must intercept the Spring MVC core codes,
that is not a good choice for now.
 *
 * Comment by @wu-sheng
 */
public class ControllerConstructorInterceptor implements
InstanceConstructorInterceptor {

    @Override
    public void onConstruct(EnhancedInstance objInst, Object[] allArguments) {
        String basePath = "";
        // 获取@RequestMapping的信息，其实主要是想要获取到路径信息
        RequestMapping basePathRequestMapping =
objInst.getClass().getAnnotation(RequestMapping.class);
```

```
        if (basePathRequestMapping != null) {
            if (basePathRequestMapping.value().length > 0) {
                basePath = basePathRequestMapping.value()[0];
            } else if (basePathRequestMapping.path().length > 0) {
                basePath = basePathRequestMapping.path()[0];
            }
        }
        EnhanceRequireObjectCache enhanceRequireObjectCache = new
EnhanceRequireObjectCache();
        enhanceRequireObjectCache.setPathMappingCache(new
PathMappingCache(basePath));
        objInst.setSkyWalkingDynamicField(enhanceRequireObjectCache);
    }
}
```

然后我们看到这个插件里面只是定义了需要增强的类的匹配形式，并没有具体的创建`EntrySpan`，`ExitSpan`的处理逻辑。其实这块处理逻辑是在`AbstractControllerInstrumentation`方法拦截定义设置好具体由哪个类来处理的主要是两个类：`RequestMappingMethodInterceptor`，`RestMappingMethodInterceptor`。

一个是针对`@RequestMapping`这种注解的，一个是针对`@GetMapping`这类注解的。其实`@GetMapping`也是又`@RequestMapping`而来的。`GetMapping`本身就用了`@RequestMapping`，相当于是指定`method`的`@RequestMapping`。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {
}
```

`RequestMappingMethodInterceptor`，`RestMappingMethodInterceptor`继承了同一个父类：`AbstractMethodInterceptor`。这两个类本身只重写了父类的两个方法：

```
    public abstract String getRequestURL(Method method);

    public abstract String getAcceptedMethodTypes(Method method);
```

所以我们重点关注父类（`AbstractMethodInterceptor`）的两个方法：

- beforeMethod （方法调用前的逻辑）
- afterMethod（方法调用后的逻辑）

```
    @Override
```

```
    public void beforeMethod(EnhancedInstance objInst, Method method, Object[]
allArguments, Class<?>[] argumentsTypes,
        MethodInterceptResult result) throws Throwable {

        // forwardRequestFlag
        Boolean forwardRequestFlag =
(Boolean)ContextManager.getRuntimeContext().get(FORWARD_REQUEST_FLAG);
        // 如果是forwardRequest就直接返回
        /**
         * Spring MVC plugin do nothing if current request is forward request.
         * Ref: https://github.com/apache/skywalking/pull/1325
         */
        if (forwardRequestFlag != null && forwardRequestFlag) {
            return;
        }

        String operationName;
        if (Config.Plugin.SpringMVC.USE_QUALIFIED_NAME_AS_ENDPOINT_NAME) {
            operationName = MethodUtil.generateOperationName(method);
        } else {
            EnhanceRequireObjectCache pathMappingCache =
(EnhanceRequireObjectCache)objInst.getSkyWalkingDynamicField();
            String requestURL = pathMappingCache.findPathMapping(method);
            if (requestURL == null) {
                requestURL = getRequestURL(method);
                pathMappingCache.addPathMapping(method, requestURL);
                requestURL = getAcceptedMethodTypes(method) +
pathMappingCache.findPathMapping(method);
            }
            operationName = requestURL;
        }

        // 设置operationName为requestURL
        // 获取HttpServletRequest
        HttpServletRequest request =
(HttpServletRequest)ContextManager.getRuntimeContext().get(REQUEST_KEY_IN_RUNTI
ME_CONTEXT);
        if (request != null) {
            // 拿到StackDepth
            StackDepth stackDepth =
(StackDepth)ContextManager.getRuntimeContext().get(CONTROLLER_METHOD_STACK_DEPT
H);

            if (stackDepth == null) {
                // new一个ContextCarrier
                ContextCarrier contextCarrier = new ContextCarrier();
                CarrierItem next = contextCarrier.items();
                while (next.hasNext()) {
                    next = next.next();
```

```
                    next.setHeadValue(request.getHeader(next.getHeadKey()));
                }
                // 创建EntrySpan
                AbstractSpan span =
ContextManager.createEntrySpan(operationName, contextCarrier);
                Tags.URL.set(span, request.getRequestURL().toString());
                Tags.HTTP.METHOD.set(span, request.getMethod());
                span.setComponent(ComponentsDefine.SPRING_MVC_ANNOTATION);
                SpanLayer.asHttp(span);

                if (Config.Plugin.SpringMVC.COLLECT_HTTP_PARAMS) {
                    final Map<String, String[]> parameterMap =
request.getParameterMap();
                    if (parameterMap != null && !parameterMap.isEmpty()) {
                        String tagValue =
CollectionUtil.toString(parameterMap);
                        tagValue =
Config.Plugin.Http.HTTP_PARAMS_LENGTH_THRESHOLD > 0
                            ? StringUtil.cut(tagValue,
Config.Plugin.Http.HTTP_PARAMS_LENGTH_THRESHOLD)
                            : tagValue;
                        Tags.HTTP.PARAMS.set(span, tagValue);
                    }
                }

                stackDepth = new StackDepth();

 ContextManager.getRuntimeContext().put(CONTROLLER_METHOD_STACK_DEPTH,
stackDepth);
            } else {
                AbstractSpan span =
                    ContextManager.createLocalSpan(buildOperationName(objInst,
method));
                span.setComponent(ComponentsDefine.SPRING_MVC_ANNOTATION);
            }

            stackDepth.increment();
        }
    }

    private String buildOperationName(Object invoker, Method method) {
        StringBuilder operationName = new
StringBuilder(invoker.getClass().getName())
            .append(".").append(method.getName()).append("(");
        for (Class<?> type : method.getParameterTypes()) {
            operationName.append(type.getName()).append(",");
        }

        if (method.getParameterTypes().length > 0) {
```

```
            operationName = operationName.deleteCharAt(operationName.length() -
1);
        }

        return operationName.append(")").toString();
    }
```

afterMethod

```
@Override
public Object afterMethod(EnhancedInstance objInst, Method method, Object[]
allArguments, Class<?>[] argumentsTypes,
    Object ret) throws Throwable {
    Boolean forwardRequestFlag =
(Boolean)ContextManager.getRuntimeContext().get(FORWARD_REQUEST_FLAG);
    /**
     * Spring MVC plugin do nothing if current request is forward request.
     * Ref: https://github.com/apache/skywalking/pull/1325
     */
    if (forwardRequestFlag != null && forwardRequestFlag) {
        return ret;
    }

    HttpServletRequest request =
(HttpServletRequest)ContextManager.getRuntimeContext().get(REQUEST_KEY_IN_RUNTI
ME_CONTEXT);

    if (request != null) {
        StackDepth stackDepth =
(StackDepth)ContextManager.getRuntimeContext().get(CONTROLLER_METHOD_STACK_DEPT
H);
        if (stackDepth == null) {
            throw new IllegalMethodStackDepthException();
        } else {
            stackDepth.decrement();
        }
        // 获取当前的span
        AbstractSpan span = ContextManager.activeSpan();

        if (stackDepth.depth() == 0) {
            HttpServletResponse response =
(HttpServletResponse)ContextManager.getRuntimeContext().get(RESPONSE_KEY_IN_RUN
TIME_CONTEXT);
            if (response == null) {
                throw new ServletResponseNotFoundException();
            }
```

```
            if (IS_SERVLET_GET_STATUS_METHOD_EXIST && response.getStatus() >=
400) {
                span.errorOccurred();
                Tags.STATUS_CODE.set(span,
Integer.toString(response.getStatus()));
            }

            // 清楚一些上下文信息

 ContextManager.getRuntimeContext().remove(REQUEST_KEY_IN_RUNTIME_CONTEXT);

 ContextManager.getRuntimeContext().remove(RESPONSE_KEY_IN_RUNTIME_CONTEXT);

 ContextManager.getRuntimeContext().remove(CONTROLLER_METHOD_STACK_DEPTH);
        }
        // 停止span
        ContextManager.stopSpan();
    }

    return ret;
}
```