

第 11 章 分布式链路追踪： SkyWalking

答主：“在人的身体外表有一些“端点”，比如舌头，医生通过观察舌头的颜色和形状，就能对病人的病情做出一些诊断。但是，对于有一些疾病，如果要进行更深入仔细地诊断，光靠观察舌头是不够的，比如对于肠胃疾病，医生会通过什么方法进行诊断呢？”

阿云：“医生会利用肠镜和胃镜深入肠胃内部，去了解病灶。”

答主：“同样，微服务也向 Spring Boot 的 Actuator 暴露了一些端点，软件开发人员以及运维人员通过观察端点的信息，就能了解微服务的运行情况。但是，在分布式的微服务系统中，多个微服务互相调用，形成了长长的链路，如果在运行中出现故障，要定位它们就比较困难。为了追踪链路，SkyWalking 应运而生，它就像胃镜一样，能够深入追踪微服务的调用链路，采集并分析链路上每个节点的运行状态信息。”

如图 11-1 所示，分布在云端的微服务节点彼此调用，形成了一条条链路，而 SkyWalking 自由地在“天空”漫步，能够深入追踪链路中每个节点的服务状况。这是 SkyWalking 的字面含义。

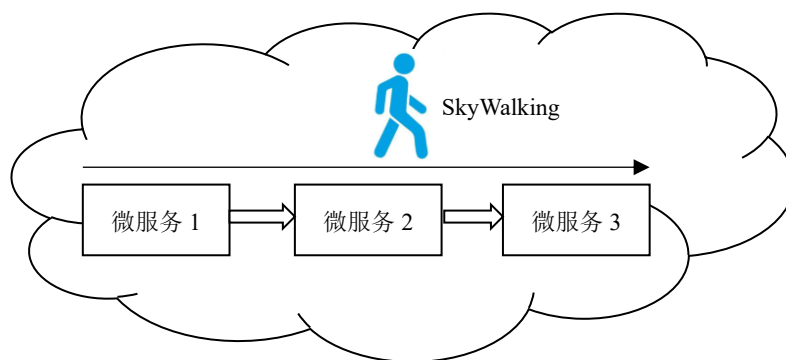


图 11-1 SkyWalking 在云端的分布式微服务链路中“漫步”

本章介绍通过 SkyWalking 来追踪微服务的调用链路，以及把监控数据持久化到 Elasticsearch 和 MySQL 数据库，还会介绍建立 SkyWalking 集群的步骤。

11.1 SkyWalking 简介

SkyWalking 是由国人吴晟开发的链路追踪工具，2017 年进入 Apache 的项目孵化器，如今已经成为 Apache 的一个开源项目。

SkyWalking 提供了强大的 APM (Application Performance Management, 应用性能管理)

功能，专门为微服务等基于容器的云原生架构提供监控服务。SkyWalking 通过探针收集应用的各项指标，并进行分布式的链路追踪。SkyWalking 会感知微服务之间的调用链路的关系，生成相应的统计数据。

SkyWalking 具有如下特性：

- (1) 支持告警。
- (2) 采用探针技术，对业务代码零侵入。所谓零侵入，是指探针不会改变业务代码，也不会改变代码的运行行为。
- (3) 轻量高效，无需额外的大数据平台。
- (4) 提供多种监控手段，支持多语言探针（Java、.Net Core 和 Node.js）。
- (5) 简洁强大的可视化后台管理界面。
- (6) 自身采用模块化架构，包括探针、UI、观测分析平台和存储模块。

如图 11-2 所示，SkyWalking 的整体架构包括四个模块：

- 探针（Agent）：就像安置在病人胃中的胃镜一样，探针也是安置在微服务中，负责收集监控数据。
- 观测分析平台（OAP, Observability Analysis Platform）：接收探针发送的监控数据，利用分析引擎对数据进行整合与运算，把统计数据存储到相应的存储设备中。
- UI：调用 OAP 接口，在可视化的界面中展示统计数据。
- 存储设备：用于存放 OAP 平台的统计数据，目前支持的数据库包括 H2（内嵌式数据库，这是 SkyWalking 的默认数据库）、ES（Elasticsearch）、MySQL、BanyanDB 等。

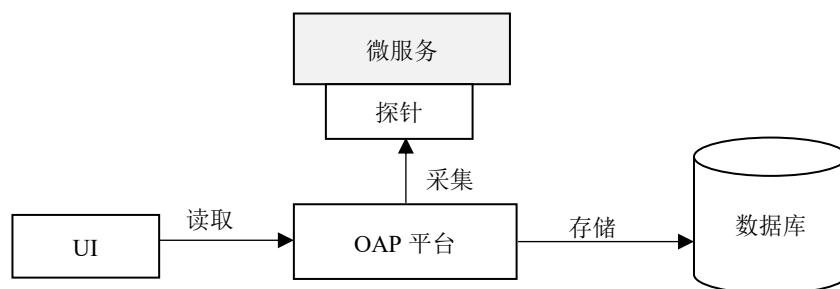


图 11-2 SkyWalking 的整体架构

11.2 比较各个链路追踪软件

目前流行的链路追踪软件包括以下几种：

- Zipkin：由 Twitter 提供的开源软件，目前在 Spring Cloud 框架中得到了广泛的使用，特点是轻量、使用部署简单。
- Pinpoint：韩国人开发的基于字节码注入的开源软件，特点是支持多种插件，UI 功能强大，接入端无代码侵入。由于收集的数据很多，整个性能会降低。

- **SkyWalking**: 国人开发的基于字节码注入的开源软件，特点是支持多种插件，UI 功能较强，接入端无代码侵入。目前已成为 Apache 的顶级开源项目。
- **CAT**: 由大众点评开发的开源软件，具体实现基于编码和配置，报表功能强大，但是对代码有侵入性，使用时需要修改应用的代码。

表 11-1 从实现原理、使用的协议以及功能特性等角度，对 Zipkin、CAT 和 SkyWalking 这三款链路追踪软件做了比较。

表 11-1 比较流行的三款链路追踪软件

软件	Zipkin	CAT	SkyWalking
实现原理	拦截请求	代码埋点（拦截器、过滤器）	探针、字节码增强
Agent 与 OAP 之间的协议	HTTP、MQ	HTTP/TCP	gRPC
OpenTracing	支持	不支持	支持
监控粒度	接口级	代码级	方法级
全局调用统计的协议	HTTP、MQ	HTTP/TCP	gRPC
JVM 监控	不支持	支持	支持
告警	不支持	支持	支持
数据存储	ES、MySQL、Cassandra、内存	MySQL、HDFS	ES、H2、MySQL、BanyanDB
可视化 UI	支持	支持	支持
聚合报表	少	非常丰富	较丰富
社区支持	国外主流	国内支持	Apache 支持
使用案例	京东、阿里定制后不开源	美团、携程、陆金所	阿里云、华为、小米、当当、百度
APM	不支持	支持	支持
WebFlux	支持	不支持	支持

11.3 安装和运行 SkyWalking

SkyWalking 的官方下载网址参见本书技术支持网页的【链接 17】。从该网址下载安装压缩包文件 `apache-skywalking-apm-9.1.0.tar.gz`，把它解压到本地。

转到 SkyWalking 的 `bin` 目录下，运行 `startup.bat` 批处理文件，就会启动 SkyWalking，实际上启动了两个服务：

- **oap-service**: OAP 平台，其实现类库位于 `oap-lib` 子目录下。
- **webapp-service**: 提供 UI 界面的 `webapp` 应用，位于 `webapp` 子目录下。

SkyWalking 的 `webapp-service` 默认监听的端口为 8080，通过浏览器访问 `http://localhost:8080`，就会访问 UI 界面，参见图 11-3。

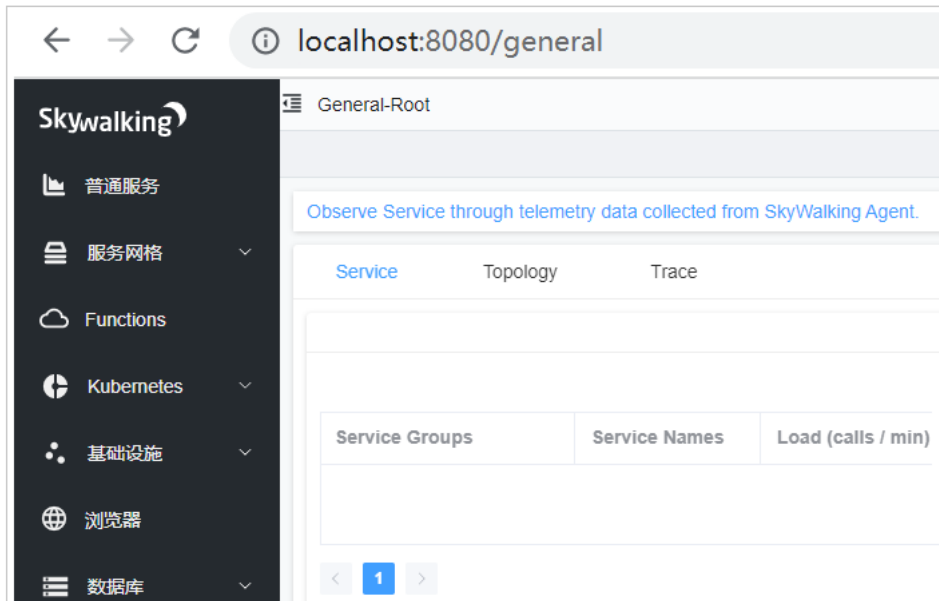


图 11-3 SkyWalking 的 UI 界面

如果要修改 webapp-service 监听的端口，可以修改 webapp/webapp.yml 文件中的如下配置代码：

```
server:
  port: 8080 #设置监听的端口
```

如图 11-4 所示，oap-service 与探针通信的默认端口为 11800，与 webapp-service 通信的默认端口为 12800。确切地说，在 oap-service 中包含了一个 collector-service 子服务，它负责收集来自探针的监控数据，它的默认端口为 11800。

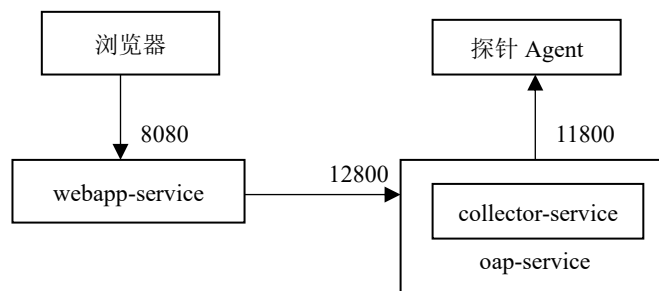


图 11-4 oap-service 默认监听的端口

在 config/application.yml 文件中，以下代码用于设置 oap-service 的监听端口：

```
restPort: ${SW_CORE_REST_PORT:12800}
grpcPort: ${SW_CORE_GRPC_PORT:11800}
```

如果修改了以上 restPort 端口，还需要对 webapp-service 的配置文件 webapp/webapp.yml 中 oap-service 的端口进行相应的修改：

```
spring:
  cloud:
```

```
gateway:
  routes:
    - id: oap-route
      uri: lb://oap-service
      predicates:
        - Path=/graphql/**
  discovery:
    client:
      simple:
        instances:
          oap-service:
            - uri: http://127.0.0.1:12800
```

11.4 在微服务中安置探针 Agent

SkyWalking 的低版本软件中自带了 Agent，而 9.0 以上的高版本需要从 SkyWalking 的官网单独下载 Agent。

从 SkyWalking 的官网(skywalking.apache.org)下载 Java 版本的 Agent 安装压缩包 `apache-skywalking-java-agent-8.11.0.tgz`，把它解压到本地，假定根目录为 `C:\skywalking-agent`。在该目录下有一个 `skywalking-agent.jar` 文件，它是 Agent 的类库文件。

下面以第 2 章的 `helloapp` 应用为例，介绍为 `hello-provider-service` 以及 `hello-consumer-service` 微服务安置探针的方法。

在 IDEA 中，选择菜单 `Run`→`Edit Configurations`，参照图 11-5，为 `HelloProviderApplication` 启动配置添加如下 VM Option 参数：

```
#skywalking-agent.jar 的本地路径
-javaagent:C:\skywalking-agent\skywalking-agent.jar
#在 SkyWalking 上显示的服务名
-Dskywalking.agent.service_name=hello-provider-service
#SkyWalking 的 collector-service 服务的 IP 及端口
-Dskywalking.collector.backend_service=127.0.0.1:11800
```

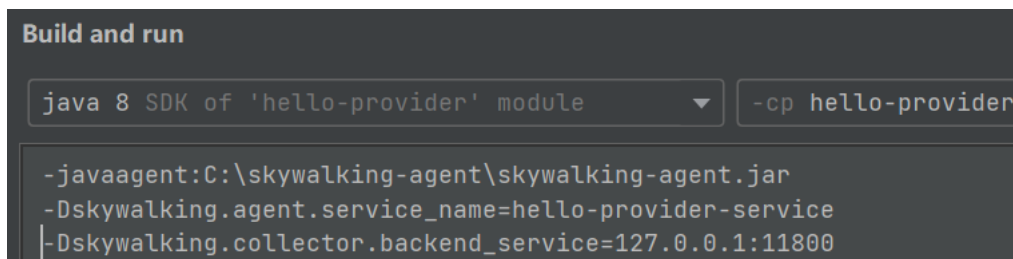


图 11-5 对 `HelloProviderApplication` 启动配置添加与 Agent 相关的参数









以此类推，对 `HelloConsumerApplication` 启动配置也添加与 Agent 相关的 VM Option 参

数，其中 `skywalking.agent.service_name` 属性的值为 `hello-consumer-service`。

阿云：“SkyWalking 的探针不需要在微服务的 `pom.xml` 文件中加入有关的依赖吗？”

答主：“探针采用底层的字节码注入技术潜伏到微服务的运行环境中，无需在微服务中加入依赖类库。这体现了 SkyWalking 对微服务代码零侵入的特性。当然，如果需要在微服务中设置一些自定义的埋点（监控端点），还是需要添加相关的依赖类库，11.6 节会对此进一步介绍。”

通过浏览器多次访问 `http://localhost:8082/enter/Tom`，再访问 SkyWalking 的 UI 界面，会看到每个微服务的服务状态的监控数据，参见图 11-6。

Service	Topology	Trace		
Service Names	Load (calls / min)	Success Rate (%)	Latency (ms)	Apdex
hello-provider-service	 0.39	 6.45	 80.10	 0.05
hello-consumer-service	 0.23	 3.23	 67.52	 0.03

展示折线图

图 11-6 SkyWalking 追踪每个微服务的服务状态的监控数据

图 11-6 展示了每个微服务的以下监控数据：

- Load（数字）：平均每分钟的请求数。
- Load（折线图）：在一个时间段内每分钟请求数的趋势图。
- Success Rate（数字）：请求成功率。
- Success Rate（折线图）：在一个时间段内请求成功率的趋势图。
- Latency(数字)：平均响应请求的时间，以毫秒为单位。
- Latency(折线图)：在一个时间段内响应请求的的时间的趋势图，以毫秒为单位。
- Service Apdex（数字）：当前服务的 Apdex（Application Performance Index，应用性能指数）评分
- Service Apdex（折线图）：在一个时间段内 Apdex 评分的趋势图。

在图 11-6 中选择展示折线图标记，就会以折线图的形式展示一段时间内相应的监控数据，参见图 11-7。

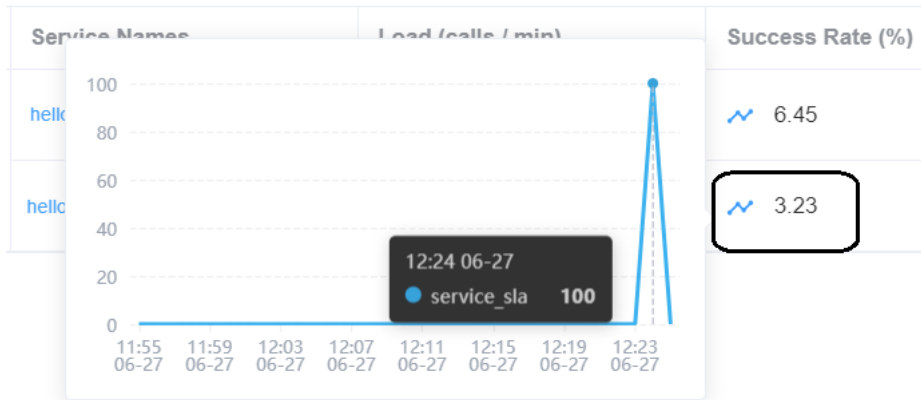


图 11-7 展示一段时间内的请求成功率

11.4.1 查看拓扑图

在图 11-6 中选择菜单 Topology，会看到微服务的拓扑图，展示了微服务之间的调用关系，参见图 11-8。如果微服务访问数据库，也会展示微服务和数据库之间的访问关系。



图 11-8 展示微服务之间调用关系的拓扑图

11.4.2 追踪链路

在图 11-6 中选择菜单 Trace，会看到每一条调用链路的详细信息，参见图 11-9。SkyWalking 为每条调用链路都分配了唯一的追踪 ID，并且还会展示链路的持续时间和使用的协议等信息。

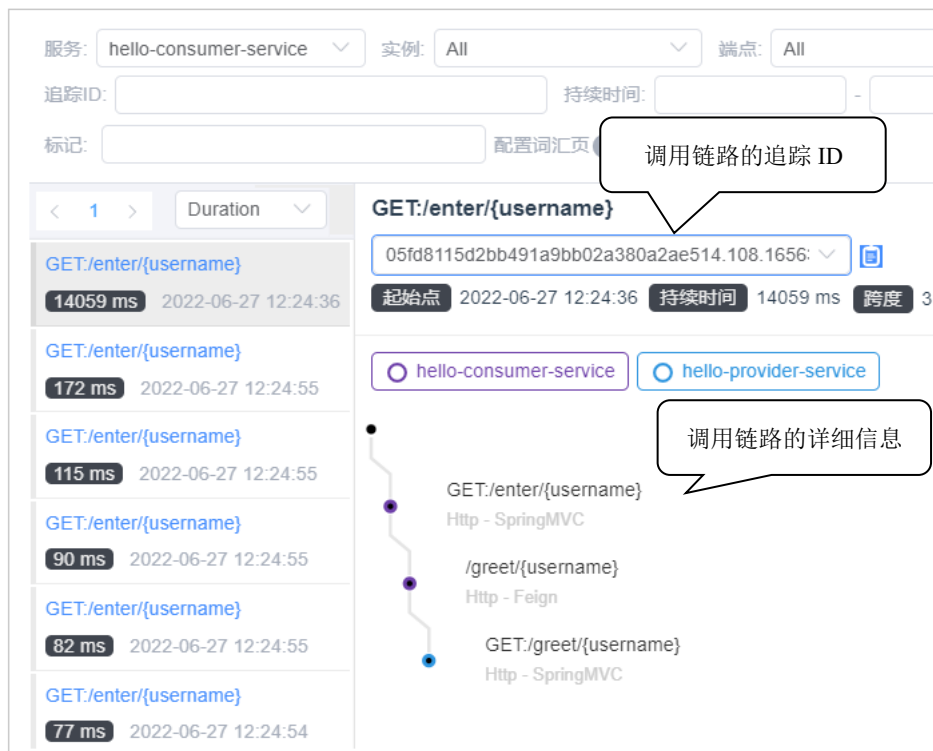


图 11-9 SkyWalking 追踪微服务的调用链路的信息

11.4.3 服务、实例和端点维度

SkyWalking 允许从服务、实例和端点这三种维度查看监控数据。在图 11-6 中选择 hello-provider-service，就会显示 hello-provider-service 微服务的监控数据，参见图 11-10。

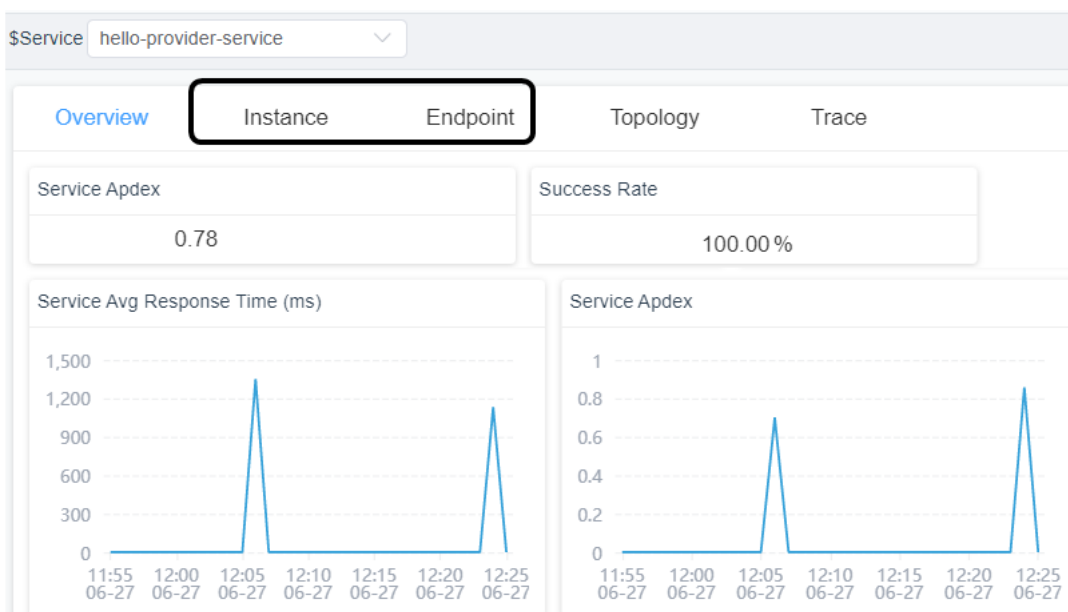


图 11-10 hello-provider-service 微服务的监控数据

在图 11-10 中选择菜单 Instance，就会从 hello-provider-service 微服务实例的维度展示监

控数据，参见图 11-11。

Service Instances	Load (calls / min)	Success Rate (%)	Latency (ms)
9163639ec42a41518c636b460c1f4f5b@19 2.168.100.105	0.00	0.00	0.00
ec6072ad32174ff69dac43b6f3a00b19@19 2.168.100.105	0.39	6.45	80.10

图 11-11 从微服务实例的维度展示监控数据

在图 11-10 中选择菜单 Endpoint，就会从端点的维度展示监控数据，参见图 11-12。这里所谓的端点是指每一个被访问的 URI，例如 GET:/greet/{username} 就是一个端点。

Endpoints	Load (calls / min)	Success Rate (%)	Latency (ms)
GET:/greet/{username}	0.39	6.45	80.10

图 11-12 从端点的维度展示监控数据

11.4.4 性能分析

SkyWalking 能够帮助运维人员分析微服务的运行性能，发现问题所在。性能分析不需要在代码中设置埋点。SkyWalking 通过周期性地对业务运行状态保存快照，来进行性能分析，资源消耗比较小。

在 SkyWalking 的 UI 界面上，选择菜单 Trace Profiling，就会进入性能分析页面，参见图 11-13。

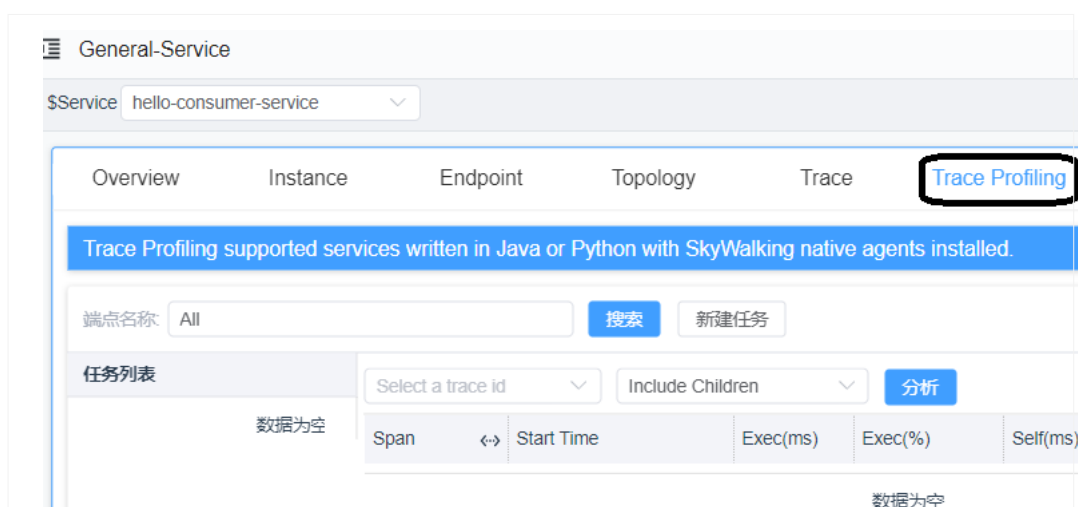


图 11-13 性能分析页面

在图 11-13 中选择新建任务，创建一个监控/enter/{username}端点的任务，参见图 11-14。

端点名称
GET:/enter/{username}

监控时间
 monitor now set start time 2022-06-30 15:52:01

监控持续时间
 5 min 10 min 15 min

起始监控时间 (ms)
- 0 +

监控间隔
 10 ms 20 ms 50 ms 100 ms

最大采样数
1

新建任务

图 11-14 创建监控/enter/{username}端点的任务

图 11-14 包括以下选项：

- 监控时间：指定从什么时候开始监控，可以选择此刻或者自定义时间。
- 监控持续时间：指定监控的时间长度。
- 起始监控时间：过多长时间采集样本。
- 监控间隔：采集样本的间隔，即执行快照的间隔。
- 最大采样数：最多采集多少次样本。

接下来通过浏览器多次访问 `http://localhost:8082/enter/Tom`，然后再观察端点 `/enter/{username}` 的性能分析数据，会看到图 11-15 所示的页面，在该页面显示了调用链路中每个端点的响应时间，自身执行代码的耗时以及访问的 API 等信息。

Span	Start Time	Exec(ms)	Exec(%)	Self(ms)	API
7b4c354a69f44d47a1997c57e496638d.65.165657 Include Children 分析 ▼ GET:/enter/{username}	2022-06-30 16:27:20	558	<div style="width: 50%;"></div>	255	SpringMVC
/greet/{username}	2022-06-30 16:27:21	303	<div style="width: 25%;"></div>	303	Feign

图 11-15 对端点/enter/{username}的性能分析

在性能分析页面的下方，还会展示方法调用堆栈的信息，参见图 11-16。

Thread Stack	↔	Duration (ms)	Self Duration (ms)
▼ java.lang.Thread.run:750		421	0
▼ org.apache.tomcat.util.threads.TaskThread\$WrappingRunnable.run:61		421	0
▼ org.apache.tomcat.util.threads.ThreadPoolExecutor\$Worker.run:659		421	0
▼ org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker:1191		421	0
▼ org.apache.tomcat.util.net.SocketProcessorBase.run:49		421	0
▼ org.apache.tomcat.util.net.NioEndpoint\$SocketProcessor.doRun:174		421	0
▼ org.apache.coyote.AbstractProtocol\$ConnectionHandler.process:88		421	0
▼ org.apache.coyote.AbstractProcessorLight.process:65		421	0
▼ org.apache.coyote.http11.Http11Processor.service:399		421	0
▼ org.apache.catalina.connector.CoyoteAdapter.service:360		421	0

图 11-16 方法调用堆栈的信息

11.5 采集日志

SkyWalking 还能采集应用程序输出的日志。以下是为 hello-consumer-service 微服务采集日志的步骤。

- (1) 在 hello-consumer 模块的 pom.xml 文件中加入以下日志工具依赖：

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-logback-1.x</artifactId>
  <version>8.11.0</version>
</dependency>
```

(2) 在 hello-consumer 模块的 src/main/resources 目录下增加 logback 日志工具的 logback-spring.xml 配置文件，参见例程 11-1。其中<Pattern>元素指定日志输出格式，tid 表示链路的追踪 ID。

例程 11-1 logback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property name="console"
    value="%date{yyyy-MM-dd HH:mm:ss}
    | %highlight(%-5level) | %boldYellow(%tid)
    | %boldYellow(%thread)
    | %boldGreen(%logger) | %msg%n"/>

  <!-- 向控制台输出日志的格式 -->
```

```

<appender name="std"
  class="ch.qos.logback.core.ConsoleAppender">

  <encoder class="ch.qos.logback.core.encoder
            .LayoutWrappingEncoder">
    <layout class="org.apache.skywalking.apm.toolkit
                .log.logback.v1.x.TraceIdPatternLogbackLayout">
      <pattern>${console}</pattern>
    </layout>
  </encoder>
</appender>

<!-- SkyWalking 采集到的日志的输出格式 -->
<appender name="grpc-log"
  class="org.apache.skywalking.apm.toolkit.log
        .logback.v1.x.log.GRPCLogClientAppender">
<encoder class="ch.qos.logback.core
              .encoder.LayoutWrappingEncoder">
  <layout class="org.apache.skywalking.apm.toolkit
              .log.logback.v1.x.mdc
              .TraceIdMDCPatternLogbackLayout">
    <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{tid}]
            [%thread] %-5level %logger{36} -%msg%n
    </Pattern>
  </layout>
</encoder>
</appender>

<root level="INFO">
  <appender-ref ref="std"/>
  <appender-ref ref="grpc-log"/>
</root>
</configuration>

```

(3) 在 C:\skywalking-agent\config 的 agent.config 配置文件中增加如下配置代码:

```

#SkyWalking 的 collector-service 服务的地址
plugin.toolkit.log.grpc.reporter.server_host=
  ${SW_GRPC_LOG_SERVER_HOST:127.0.0.1}

#SkyWalking 的 collector-service 服务的端口
plugin.toolkit.log.grpc.reporter.server_port=
  ${SW_GRPC_LOG_SERVER_PORT:11800}

```

```
plugin.toolkit.log.grpc.reporter.max_message_size=
    ${SW_GRPC_LOG_MAX_MESSAGE_SIZE:10485760}

plugin.toolkit.log.grpc.reporter.upstream_timeout=
    ${SW_GRPC_LOG_GRPC_UPSTREAM_TIMEOUT:30}
```

(4) 在 HelloConsumerController 类的 sayHello()请求处理方法中增加输出日志的代码:

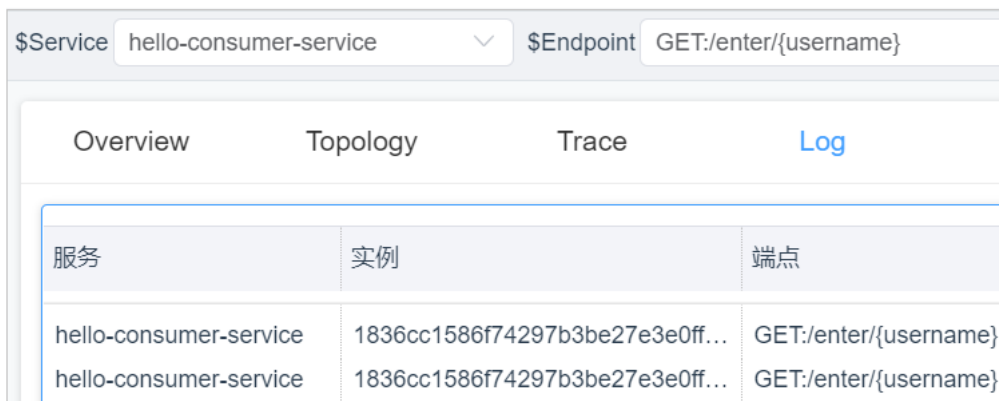
```
private final static Logger logger = LoggerFactory
    .getLogger(HelloConsumerController.class);

@GetMapping(value = "/enter/{username}")
public String sayHello(@PathVariable String username) {
    logger.info("from sayHello:"+username);
    return helloFeignService.sayHello(username);
}
```

(5) 通过浏览器多次访问 `http://localhost:8082/enter/Tom`, 在 IDEA 的控制台会输出包括 TID (Trace ID) 在内的日志信息:

```
2022-06-27 17:28:11 | INFO
| TID: 85fa936cee2e44db9cdcd1c26fe64c6e...
| http-nio-8082-exec-3
| demo.helloconsumer.HelloConsumerController
| from sayHello:Tom
```

再访问 SkyWalking 的 UI 界面, 选择 Log 菜单, 会看到 hello-consumer-service 微服务输出的日志清单, 参见图 11-17。



服务	实例	端点
hello-consumer-service	1836cc1586f74297b3be27e3e0ff...	GET:/enter/{username}
hello-consumer-service	1836cc1586f74297b3be27e3e0ff...	GET:/enter/{username}

图 11-17 hello-consumer-service 微服务输出的日志清单

选择图 11-17 中的一条日志, 就会展示该日志的详细信息, 参见图 11-18。

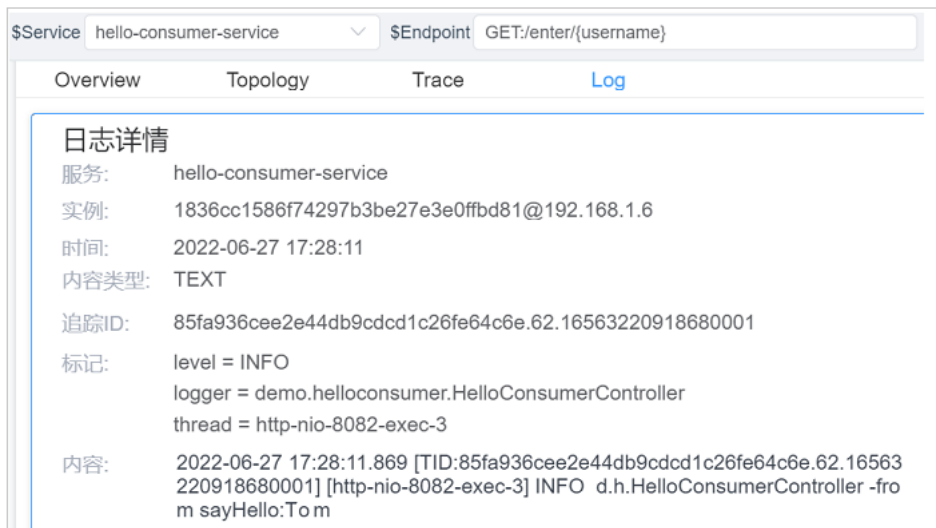


图 11-18 展示日志的详细信息

11.6 自定义链路追踪

阿云：“11.4.3 节讲到，微服务的调用链路中的 URL 路径（如/greet/{username}或/enter/{username}）会作为 SkyWalking 的监控端点。除此以外，微服务中被调用的普通方法是否也可以作为链路中的端点，被 SkyWalking 追踪呢？”

答主：“可以的。对于普通方法，需要通过 SkyWalking 的@Trace 注解把它标识为被追踪的端点。把第三方的 SkyWalking 的@Trace 注解“埋藏”在微服务的程序代码中，因此，这种端点也形象地称为埋点。”

假定在 HelloConsumerController 类中有一个普通的 add()方法，以下是把它设为监控端点的步骤。

(1) 在 hello-consumer 模块的 pom.xml 文件中加入以下追踪工具依赖：

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-trace</artifactId>
  <version>8.11.0</version>
</dependency>
```

(2) 在 HelloConsumerController 类中增加一个请求处理方法 sum()以及普通的方法 add(), sum()方法会调用 add()方法。add()方法用@Trace 注解标识：

```
@GetMapping(value = "/sum/{a}/{b}")
public String sum(@PathVariable int a,@PathVariable int b){
    return Integer.valueOf(add(a,b)).toString();
}

@Trace(operationName = "add")
```

```

//方法的第一个参数
@Tag(key = "arg1", value = "arg[0]")
//方法的第二个参数
@Tag(key = "arg2", value = "arg[1]")
//方法的返回值
@Tag(key = "result", value = "returnedObj")
public int add(int a,int b){
    TraceContext.putCorrelation("myKey", "myValue");
    Optional<String> op = TraceContext.getCorrelation("myKey");
    logger.info("myKey = {} ", op.get());

    String traceId = TraceContext.traceId();
    logger.info("traceId = {} ", traceId);

    return a+b;
}

```

@Trace 注解会使得 add()方法成为被 SkyWalking 追踪的链路中的一个端点。还可以为 add()方法加上@Tag 标签，用于追踪方法的参数和返回值。在 add()方法中，还可以通过 TraceContext 类向追踪上下文容器存取 key/value 数据，以及通过 traceId()方法，获得当前链路的 Trace ID。



@Trace 注解只能用来标识实例方法，而不能标识静态方法，因为 SkyWalking 不支持追踪静态方法。

(3) 通过浏览器多次访问 <http://localhost:8082/sum/5/6>，在 IDEA 的控制台会输出包括 TID 在内的日志信息：

```

myKey = myValue
traceId = 6849a6c2a46d4059aa2355f15a262...

```

再访问 SkyWalking 的 UI 界面，会看到 hello-consumer-service 微服务中有一条/sum/{a}/{b} 链路，参见图 11-19。



图 11-19 /sum/{a}/{b}链路

图 11-19 展示的链路中有一个 add 端点，选择该端点，会展示它的详细信息，参见图 11-20。

标记.	
服务:	hello-consumer-service
实例:	912f7e44449c43ca809747abf0e0f117@192.168.1.6
端点:	add
跨度类型:	Local
组件:	Unknown
Peer:	No Peer
错误:	false
arg1:	5
arg2:	6
result:	11

图 11-20 add 端点的详细信息

在图 11-20 中，arg1、arg2 和 result 标记由 add()方法的@Tag 标签定义。

11.7 忽略端点

11.6 节介绍了把普通方法设为可以被 SkyWalking 追踪的端点的步骤。而在有些场景，需要让 SkyWalking 忽略追踪一些端点，因为没有必要采集这些端点的监控数据。指定 SkyWalking 忽略 hello-consumer-service 微服务的一些端点的配置步骤如下。

(1) 把 C:\skywalking-agent\optional-plugins 目录下的 apm-trace-ignore-plugin-8.11.0.jar 文件复制到 C:\skywalking-agent\plugins 目录下。

(2) 在 C:\skywalking-agent\config 目录下新增一个 apm-trace-ignore-plugin.config 文件，指定需要忽略的路径：


```
trace.ignore_path=${SW_AGENT_TRACE_IGNORE_PATH:
    /actuator/health/**, /list }
```

以上代码指定 SkyWalking 忽略的路径为/actuator/health/**和/list，多个路径之间以逗号隔开。在路径中可以加入通配符，例如：

- /mypath/? #匹配单个字符
- /mypath/* #匹配多个字符
- /mypath/** #匹配多个字符并且支持多级目录

配置好忽略的路径后，通过浏览器访问 <http://localhost:8082/actuator/health> 或者 <http://localhost:8082/list>，SkyWalking 不会采集这两个端点的监控数据，在 SkyWalking 的 UI 界面中不会看到它们的链路信息。

除了在 `apm-trace-ignore-plugin.config` 文件中指定需要忽略的端点，还可以通过 `skywalking.trace.ignore_path` 系统属性来设定。步骤为在 IDEA 中，选择菜单 `Run→Edit Configurations`，对 `HelloConsumerApplication` 的启动配置增加如下 VM Option 参数：

```
-Dskywalking.trace.ignore_path=/actuator/health/**,/list
```

11.8 告警

SkyWalking 在监控微服务的调用链路的过程中，如果发现监控数据（如服务响应时间、服务响应时间百分比）达到告警规则中设置的阈值，就会发送相应的告警消息。发送告警消息是通过调用 `webhook` 接口来完成的，`webhook` 接口可以由开发人员提供具体的实现。开发人员为指定的 `webhook` 接口编写具体的告警操作，比如把告警消息发送到控制台，或向相关工作人员发送邮件和短信等。

在 SkyWalking 安装目录的 `config/alarm-settings.yml` 文件中，已经设置了如下默认的告警规则：

```
rules:
  service_resp_time_rule:
    metrics-name: service_resp_time #指标名称
    op: ">" #大于
    threshold: 1000 #阈值，以毫秒为单位
    period: 10 #间隔时间，以毫秒为单位
    count: 3 #指标达到阈值的服务次数
    silence-period: 5 #告警消息发送后多少分钟内不重复发送
    message: Response time of service {name} is #告警消息
      more than 1000ms in 3 minutes of last 10 minutes.

  service_sla_rule:
    metrics-name: service_sla
```

```
op: "<"
threshold: 8000
  period: 10
  count: 2
  silence-period: 3
message: Successful rate of service {name} is
  lower than 80% in 2 minutes of last 10 minutes

service_resp_time_percentile_rule:
  metrics-name: service_percentile
  op: ">"
  threshold: 1000,1000,1000,1000,1000
  period: 10
  count: 3
  silence-period: 5
message: Percentile response time of service {name} alarm
  in 3 minutes of last 10 minutes,
  due to more than one condition of p50 > 1000,
  p75 > 1000, p90 > 1000, p95 > 1000, p99 > 1000

service_instance_resp_time_rule:
  metrics-name: service_instance_resp_time
  op: ">"
  threshold: 1000
  period: 10
  count: 2
  silence-period: 5
message: Response time of service instance
  {name} is more than 1000ms in 2 minutes
  of last 10 minutes

database_access_resp_time_rule:
  metrics-name: database_access_resp_time
  threshold: 1000
  op: ">"
  period: 10
  count: 2
message: Response time of database access {name}
  is more than 1000ms in 2 minutes of last 10 minutes

endpoint_relation_resp_time_rule:
  metrics-name: endpoint_relation_resp_time
  threshold: 1000
```

```
op: ">"
period: 10
count: 2
message: Response time of endpoint relation
        {name} is more than 1000ms in 2 minutes of last 10
minutes
```

以上配置代码定义了默认的 6 种告警规则：

- **service_resp_time_rule**: 在最近 10 分钟期间, 有 3 分钟内服务的响应时间超过 1 秒。
- **service_sla_rule**: 在最近 10 分钟期间, 有 2 分钟内服务的成功率低于 80%。
- **service_resp_time_percentile_rule**: 在最近 10 分钟期间, 有 3 分钟内 50% 的服务的响应时间超过 1 秒, 或 75% 的服务的响应时间超过 1 秒, 或 90% 的服务的响应时间超过 1 秒, 或 95% 的服务的响应时间超过 1 秒, 或 99% 的服务的响应时间超过 1 秒。
- **service_instance_resp_time_rule**: 在最近 10 分钟期间, 有 2 分钟内服务实例的响应时间超过 1 秒。
- **database_access_resp_time_rule**: 在最近 10 分钟期间, 有 2 分钟内数据库的响应时间超过 1 秒。
- **endpoint_relation_resp_time_rule**: 在最近 10 分钟期间, 有 2 分钟内端点的响应时间超过 1 秒。

告警规则包含以下属性

- **metrics-name**: 衡量的指标名称。
- **threshold**: 阈值。
- **op**: 比较符号, 包括: >、<或=。
- **period**: 检查指标数据是否符合告警规则的时间区间, 以分钟为单位。
- **count**: 指标达到阈值的服务次数。达到 **count** 次数后, 会触发告警消息。
- **silence-period**: 不重复发送相同的告警消息的时间区间。
- **message**: 告警消息的模板。

在 `config/alarm-settings.yml` 文件的末尾, 会配置 `webhooks` 网络钩子, 用来指定产生告警时的调用地址:

```
webhooks:
  - http://127.0.0.1/notify/
  - http://127.0.0.1/go-wechat/
```

11.8.1 编写满足告警规则的方法

在 `HelloConsumerController` 类中增加一个请求处理方法 `timeout()`, 它在运行时由于响应

超时而满足告警规则:

```
//每次调用时睡眠 2 秒, 模拟响应超时
@GetMapping("/timeout")
public String timeout(){
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "timeout";
}
```

通过浏览器多次调用/timeout, 由于响应超时, SkyWalking 就会生成告警消息。

11.8.2 创建处理告警的网络钩子

在 HelloConsumerController 类中创建一个 notify()方法, 它映射的 URL 为/notify, 充当处理告警的 webhook 网络钩子:

```
@PostMapping("/notify")
public void notify(
    @RequestBody List<AlarmMessage> alarmMessageList){
    alarmMessageList.forEach(
        value->{System.out.println(value);}
    );
}
```

产生告警时, SkyWalking 会调用 notify()方法, 该方法的 HTTP 请求方式为 POST。notify()方法的参数用@RequestBody 注解来标识。SkyWalking 向 notify()方法提交的原始的告警数据的格式为:

```
[{
  "scopeId": 1,
  "scope": "SERVICE",
  "name": "serviceA",
  "id0": "aGVsbG8tY29uc3VtZXItc2VydmljZQ==.1",
  "id1": "",
  "ruleName": "service_resp_time_rule",
  "alarmMessage": "alarmMessage xxxx",
  "startTime": 1560524171000
}, {
  "scopeId": 1,
  "scope": "SERVICE",
  "name": "serviceB",
```

```

        "id0": "VXNlcm90_VXNlcm90",
        "id1": "aGVsbG8tY29uc3VtZXItc2VydmljZQ==.1_R0VU",
        "ruleName": "service_resp_time_rule",
        "alarmMessage": "alarmMessage yyy",
        "startTime": 1560524171000
    }}

```

为了便于读取告警数据，可以把 `notify()` 方法的 `alarmMessageList` 参数声明为 `List<AlarmMessage>` 类型，例程 11-2 是 `AlarmMessage` 类的源代码。

例程 11-2 AlarmMessage.java

```

public class AlarmMessage {
    private Integer scopeId;
    private String name;
    private String id0;
    private String id1;
    private String alarmMessage;    //告警的消息
    private Long startTime;        //告警的产生时间
    private String ruleName;

    public Integer getScopeId() {
        return scopeId;
    }

    public void setScopeId(Integer scopeId) {
        this.scopeId = scopeId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getId0() {
        return id0;
    }

    public void setId0(String id0) {
        this.id0 = id0;
    }

    .....
}

```

```

@Override
public String toString() {
    return "AlarmMessage{" +
        "scopeId=" + scopeId +
        ", name=" + name +
        ", id0=" + id0 +
        ", id1=" + id1 +
        ", alarmMessage=" + alarmMessage +
        ", ruleName=" + ruleName +
        ", startTime=" + startTime +"}";
}
}

```

11.8.3 测试告警

测试告警的步骤如下。

(1) 修改 SkyWalking 的告警规则配置文件 `config/alarm-settings.yml`，将 webhook 网络钩子的地址修改为：

```

webhooks:
- http://127.0.0.1:8082/notify

```

(2) 通过浏览器多次访问 `http://localhost:8082/timeout`。

(3) 在 SkyWalking 的 UI 界面，会看到图 11-21 所示的告警消息。

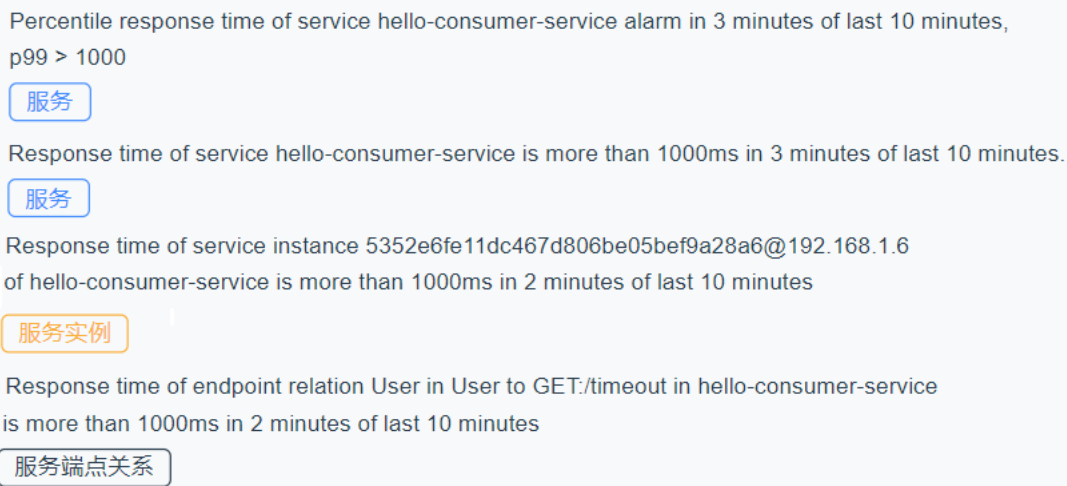


图 11-21 告警消息

(4) 当 SkyWalking 自动调用作为网络钩子的 `http://localhost:8082/notify` 时，在 IDEA 的控制台会看到 `HelloControllerConsumer` 类的 `notify()` 方法打印如下告警消息：

```

AlarmMessage{
  scopeId=2,
  name=12ad0fb.....@192.168.1.6 of hello-consumer-service,

```

```

id0=aGVsbG8tY29uc3VtZXItc2V,
id1=,
alarmMessage=Response time of
  service instance 12ad0fb04f2.....
  of hello-consumer-service is more than 1000ms
  in 2 minutes of last 10 minutes,
ruleName=service_instance_resp_time_rule,
startTime=1656470963168
}

AlarmMessage{
  scopeId=6,
  name=User in User to GET:/timeout in hello-consumer-
service,
  id0=VXNlcnQ=.0_VXNlcnQ==,
  id1=aGVsbG8tY29uc3VtZXItc2VydmljZQ.....,
  alarmMessage=Response time of endpoint relation
  User in User to GET:/timeout in hello-consumer-service
  is more than 1000ms in 2 minutes of last 10 minutes,
  ruleName=endpoint_relation_resp_time_rule,
  startTime=1656470963170
}

AlarmMessage{
  scopeId=1,
  name=hello-consumer-service,
  id0=aGVsbG8tY29uc3VtZXItc2VydmljZQ==.1,
  id1=,
  alarmMessage=Percentile response time of
  service hello-consumer-service alarm in 3 minutes
  of last 10 minutes, due to more than one condition of
  p50 > 1000, p75 > 1000, p90 > 1000, p95 > 1000, p99 >
1000,
  ruleName=service_resp_time_percentile_rule,
  startTime=1656471023167
}

```

图 11-22 展示了产生告警的流程。

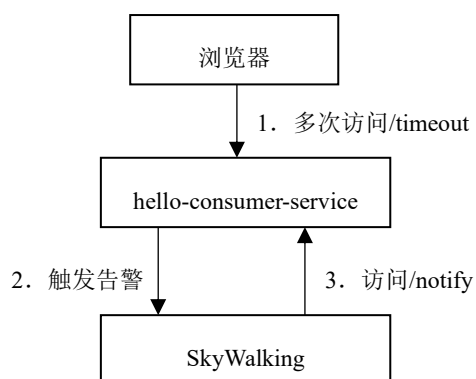


图 11-22 产生告警的流程

在实际应用中，可以在 webhook 接口的实现中对接短信、邮件等平台，确保当告警出现时，能迅速把告警消息发送给相应的处理人员，提高故障处理的速度。

11.9 整合 Elasticsearch 数据库

阿云：“SkyWalking 产生的监控数据可以永久保存吗？”

答主：“默认情况下，SkyWalking 把监控数据存储到 H2 数据库中。H2 是一个内嵌在 SkyWalking 中的数据库，以内存作为存储介质。因此当 SkyWalking 重启后，原先的监控数据就丢失了。如果希望永久地存储监控数据，可以使用 Elasticsearch 或 MySQL 数据库。”

Elasticsearch 用 Java 语言开发，是一种流行的企业级搜索引擎数据库。SkyWalking 整合 Elasticsearch 数据库的步骤如下。

(1) 从 Elasticsearch 的官网下载 Elasticsearch 安装压缩包，网址参见本书技术支持网页的【链接 18】。把安装压缩包 elasticsearch-8.3.0-windows-x86_64.zip 文件解压到本地，假定根目录为 C:\elasticsearch。

(2) 修改 C:\elasticsearch\config\elasticsearch.yml 配置文件，增加如下配置内容：

```
#集群名字，可以随便取，
#但是要要和 SkyWalking 的 storage.elasticsearch.namespace 属性一致
cluster.name: my-application
node.name: node-1 #节点名字，可以随便取
network.host: 0.0.0.0
http.port: 9200 #监听的端口
discovery.seed_hosts: ["127.0.0.1"] #主节点的地址
#主节点的名字，与 node.name 一致
cluster.initial_master_nodes: ["node-1"]
```

(3) 运行 C:\elasticsearch\bin\elasticsearch.bat，就会启动 Elasticsearch 服务器。

(4) Elasticsearch 服务器有一个登录账户，账户名为 elastic。在 DOS 命令行，转到 C:\elasticsearch\bin 目录，运行以下命令，修改 elastic 账户的口令，参见图 11-23，假定把口令改为 654321：

```
elasticsearch-reset-password --username elastic -i
```

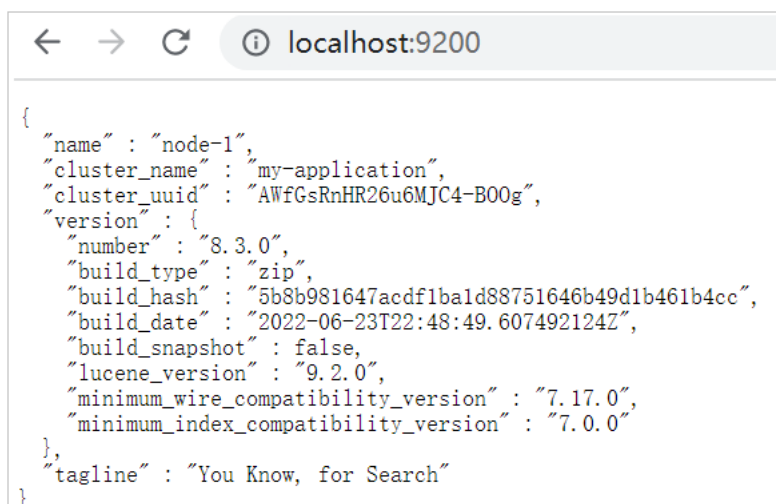


```
C:\elasticsearch\bin>elasticsearch-reset-password --username elastic -i
warning: ignoring JAVA_HOME=C:\jdk; using bundled JDK
This tool will reset the password of the [elastic] user.
You will be prompted to enter the password.
Please confirm that you would like to continue [y/N]y

Enter password for [elastic]: 654321
Re-enter password for [elastic]:
Password for the [elastic] user successfully reset.
```

图 11-23 修改 elastic 账户的口令

(5) 通过浏览器访问 <http://localhost:9200>，在登录窗口中输入用户名 elastic，口令 654321，如果看到图 11-24 所示的页面，就表明 Elasticsearch 运行正常。



```
{
  "name" : "node-1",
  "cluster_name" : "my-application",
  "cluster_uuid" : "AWfGsRnHR26u6MJC4-B00g",
  "version" : {
    "number" : "8.3.0",
    "build_type" : "zip",
    "build_hash" : "5b8b981647acdf1ba1d88751646b49d1b461b4cc",
    "build_date" : "2022-06-23T22:48:49.607492124Z",
    "build_snapshot" : false,
    "lucene_version" : "9.2.0",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

图 11-24 Elasticsearch 的主页

(6) 修改 SkyWalking 安装目录下的 config/application.yml 文件，指定把监控数据存储到 Elasticsearch 中，以下代码中的粗体字是修改的内容：

```
storage:
  selector: ${SW_STORAGE:elasticsearch}
  elasticsearch:
    namespace: ${SW_NAMESPACE:"my-application"}
    clusterNodes:
      ${SW_STORAGE_ES_CLUSTER_NODES:localhost:9200}
    protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
    connectTimeout: ${SW_STORAGE_ES_CONNECT_TIMEOUT:3000}
    socketTimeout: ${SW_STORAGE_ES_SOCKET_TIMEOUT:30000}
    responseTimeout: ${SW_STORAGE_ES_RESPONSE_TIMEOUT:15000}
    numHttpClientThread:
      ${SW_STORAGE_ES_NUM_HTTP_CLIENT_THREAD:0}
  user: ${SW_ES_USER:"elastic"}
```

```
password: ${SW_ES_PASSWORD:"654321"}
.....
```

以上 `storage.elasticsearch.namespace` 属性的值为 `my-application`，和 Elasticsearch 配置文件中的 `cluster.name` 属性保持一致。

做好上述设置后，启动 SkyWalking，监控数据就会永久保存到 Elasticsearch 中。

11.10 整合 MySQL 数据库

SkyWalking 也支持把监控数据永久存储到 MySQL 中。整合 MySQL 数据库的步骤如下：

(1) 从 MySQL 的官网（网址参见本书技术支持网页的【链接 19】）下载 MySQL 8 的安装软件，把它安装到本地。创建账号 `root`，口令为 `1234`。再创建名为 `swtest` 的数据库。

(2) 从 MySQL 的官网下载驱动程序类库 `mysql-connector-java-8.0.11.jar` 文件，把它拷贝到 SkyWalking 安装目录的 `oap-libs` 子目录下。

(3) 修改 SkyWalking 安装目录的 `config/application.yml` 文件，指定用 MySQL 存储监控数据，并且配置连接 MySQL 的属性：

```
storage:
  selector: ${SW_STORAGE:mysql}
  mysql:
    properties:
      jdbcUrl: ${SW_JDBC_URL:
        "jdbc:mysql://localhost:3306/swtest?
        useUnicode=true&characterEncoding=utf8
        &serverTimezone=Asia/Shanghai
        &useSSL=false&rewriteBatchedStatements=true"}

      #登录账户名: root
      dataSource.user: ${SW_DATA_SOURCE_USER:root}
      #登录口令: 1234
      dataSource.password:
        ${SW_DATA_SOURCE_PASSWORD:1234}

      dataSource.cachePrepStmts:
        ${SW_DATA_SOURCE_CACHE_PREP_STMTS:true}

      dataSource.prepStmtCacheSize:
        ${SW_DATA_SOURCE_PREP_STMT_CACHE_SQL_SIZE:250}
.....
```

11.11 通过 Nacos 建立 SkyWalking 集群

11.1 节讲到, SkyWalking 包括 oap-service 和 webapp-service 两个服务。对于 oap-service 服务, 可以通过 Nacos 建立集群。在图 11-25 所示的 SkyWalking 集群中, 有两个 oap-service 节点, oap-service1 节点监听的端口为 11801 和 12801, oap-service2 节点监听的端口为 11802 和 12802。这两个节点都注册到同一个 Nacos 服务器和同一个 Elasticsearch, 这样就能保证两个节点上监控数据的一致性。

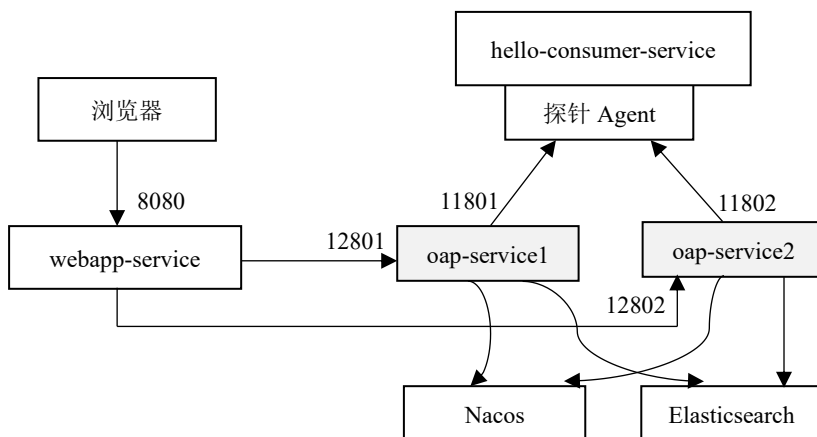


图 11-25 包含两个 oap-service 节点的 SkyWalking 集群的架构

对于 webapp-service 服务, 也可以通过 Nginx 建立集群, 如图 11-26 所示, Nginx 为两个 webapp-service 节点提供代理。5.4 节介绍了 Nginx 的用法, 读者可以参照 5.4 节为 webapp-service 服务建立集群。

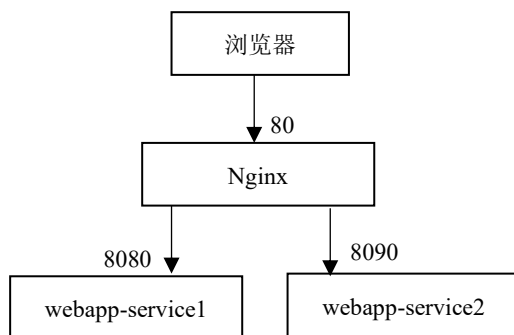


图 11-26 webapp-service 服务的集群

建立图 11-25 所示的包含两个 oap-service 节点的 SkyWalking 集群的步骤如下。

- (1) 把 SkyWalking 的安装压缩文件展开两份, 假定分别位于 C:\skywalking1 和 C:\skywalking2 目录下。
- (2) 参照 11.9 节配置两个 SkyWalking 节点, 使它们都连接到同一个 Elasticsearch 数据库。
- (3) 修改两个 SkyWalking 节点的 config/application.yml 文件, 使它们都注册到 Nacos

服务器:

```
cluster:
  selector: ${SW_CLUSTER:nacos}
  nacos:
    serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
    hostPort: ${SW_CLUSTER_NACOS_HOST_PORT:localhost:8848}
    .....
```

(4) 修改两个 SkyWalking 节点的 config/application.yml 文件, 设置它们监听的端口:

```
#第一个节点
core:
  selector: ${SW_CORE:default}
  default:
    restHost: ${SW_CORE_REST_HOST:0.0.0.0}
    restPort: ${SW_CORE_REST_PORT:12801}
    grpcHost: ${SW_CORE_GRPC_HOST:0.0.0.0}
    grpcPort: ${SW_CORE_GRPC_PORT:11801}

#第二个节点
core:
  selector: ${SW_CORE:default}
  default:
    restHost: ${SW_CORE_REST_HOST:0.0.0.0}
    restPort: ${SW_CORE_REST_PORT:12802}
    grpcHost: ${SW_CORE_GRPC_HOST:0.0.0.0}
    grpcPort: ${SW_CORE_GRPC_PORT:11802}
```

(5) 修改 C:\skywalking1\webapp\webapp.yml 文件, 加入两个 oap-service 节点的地址, 以逗号隔开:

```
spring:
  cloud:
    discovery:
      client:
        simple:
          instances:
            oap-service:
              - uri: http://127.0.0.1:12801,
                http://127.0.0.1:12802
```

(6) 修改 hello-consumer 模块的启动配置, 在 VM Option 参数中加入两个 oap-service 节点的 collector-service 服务的地址:

```
-Dskywalking.collector.backend_service=127.0.0.1:11801,
```

(7) 启动 Nacos 和 Elasticsearch，运行 C:\skywalking1\bin\oapService.bat 和 C:\skywalking2\bin\oapService.bat，分别启动两个 oap-service 节点，再运行 C:\skywalking1\bin\webappService.bat，启动一个 webapp-service 服务，再启动 hello-provider 模块和 hello-consumer 模块。这样，整个 SkyWalking 集群就搭建好了。

11.12 小结

当用户向微服务系统发出一个请求时，该请求会由多个微服务共同协作，来生成响应结果，如果有一个环节出了故障，就会导致响应失败。SkyWalking 通过探针追踪微服务的调用链路，监控每个请求的响应时间、响应状态，以及由哪些端点参与对请求的响应等。如果满足告警规则，还会发出告警，帮助运维人员及时发现链路中的隐患。

SkyWalking 支持把监控数据永久保存到 Elasticsearch 或 MySQL 等数据库中，还可以通过 Nacos 建立 oap-service 服务的集群，确保集群中 oap-service 节点之间监控数据的一致性。

SkyWalking 包括两个服务：

- oap-service 服务：收集并分析由探针采集的监控数据，其 collector-service 子服务监听探针的默认端口为 11800。向 webapp-service 服务提供监控数据的默认端口为 12800。
- webapp-service 服务：提供展示监控数据的 UI 界面，默认端口为 8080。