链路追踪 TAS

用户操作指南

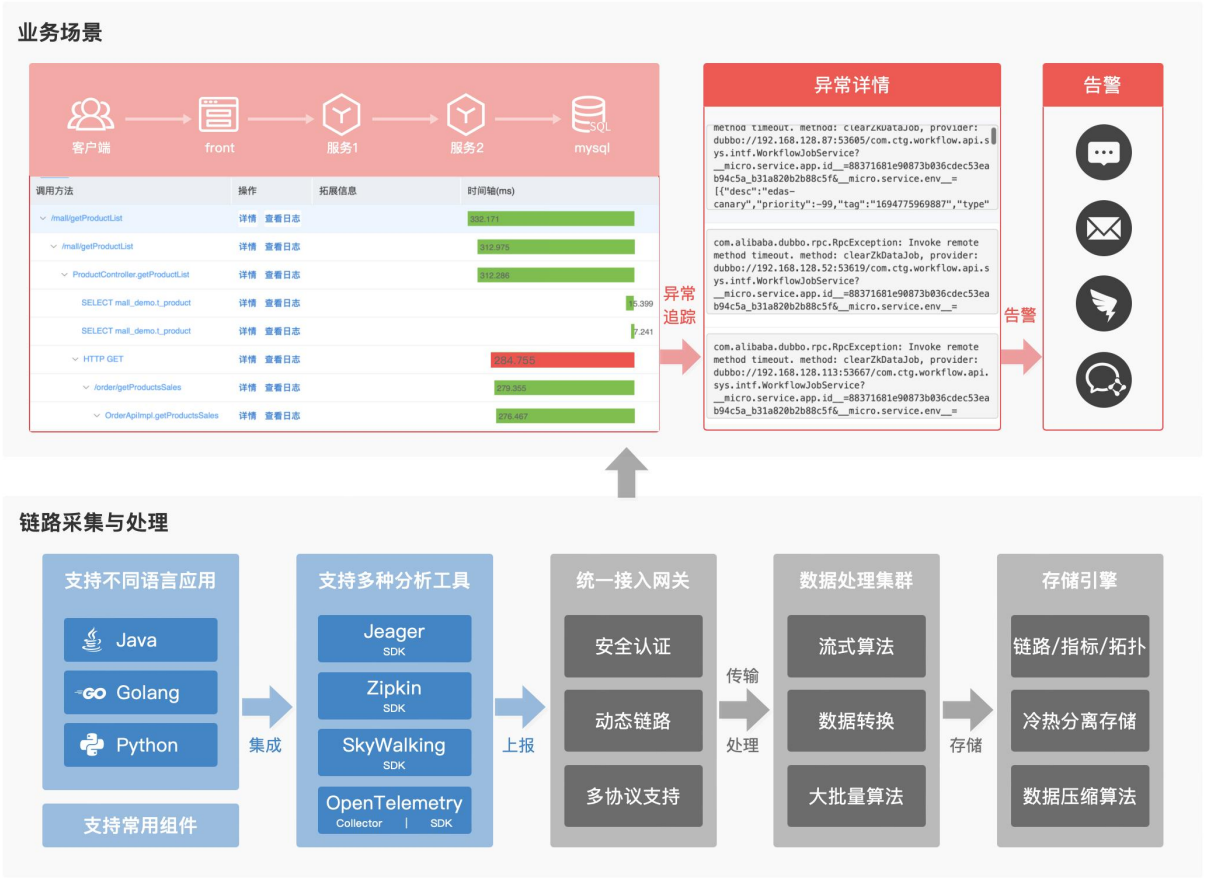
天翼云科技有限公司

一、 产品概述

1.1、产品定义

链路追踪（TracingAnalysis）是天翼云可观测体系的重要构成。

链路追踪（TracingAnalysis）为分布式应用提供完整的调用链路还原、链路分析、链路拓扑、依赖分析等能力，帮助用户快速诊断分布式应用。



数据采集

- 支持多语言接入，兼容 OpenTelemetry、Jaeger 等多种协议。
- 采用 STAM 流拓扑分析方法，即时传递拓扑关系，降低资源消耗。

数据传输与存储

- 网关与计算引擎支持按需无限扩容。
- 利用 ZSTD 数据压缩算法，提升持久化效率，节省存储空间。
- 利用稀疏索引技术实现毫秒级查询响应。

链路追踪

- 提供调用拓扑能力，直观展示应用之间、服务之间的调用关系。
- 提现调用链分析能力，支持链路数据层层拆解，帮助开发者快速诊断问题。
- 提供异常追踪能力，异常可定位到代码级。

1.2、功能特性

链路追踪 TAS 的主要功能特性如下：

分类	功能	说明
应用接入	Java 应用	支持通过 OpenTelemetry 和 Jaeger 上报 Java 应用数据。
	Go 应用	支持通过 OpenTelemetry 和 Jaeger 上报 Go 应用数据。
	Python 应用	支持通过 OpenTelemetry 和 Jaeger 上报 Python 应用数据。
	Node.js 应用	支持通过 OpenTelemetry 和 Jaeger 上报 Node.js 应用数据。

应用详情	应用总览	以应用为维度，统计整个应用的关键指标，帮助您快速掌握应用的整体状况。
	概览	应用的概览可以提供核心监控数据，帮助您快速掌握某个应用实例/接口/数据库的具体情况。
	SQL 调用分析	SQL 调用分析通常是指对一段时间内对 SQL 语句的调用次数、耗时的统计分析。
	链路上游	链路上游是当前接口的调用者。通过链路上游分析可以查看上游接口的请求量、平均延时、错误数信息。
	链路下游	链路下游是当前接口的被调用方。通过链路下游分析可以查看下游接口的请求量、平均延时、错误数信息。
	调用链查询	展示该应用实例/接口/SQL 涉及的调用链信息，包括 TraceID、产生时间、接口名称、耗时、状态信息。
	应用实例	提供应用实例级别的监控详情。展示概览和调用链信息。
	接口调用	提供应用接口级别的监控详情。通过丰富的链路分析报表，展示包括链路上下游及调用链分析。
	数据库调用	提供数据库维度的监控详情。展示 SQL 和调用链信息。
	自定义配置	自定义配置可以让您对具体某个应用进行单独管理，包是否关联业务日志。

其他	调用链查询	展示当前租户下所有调用链路信息。您可以根据多个筛选条件租户查询您想看的调用链，可以点击「TraceID」查看具体的调用链详情。
	接入点信息	根据不同的客户端采集工具，提供不同资源池（地域）的接入点信息。
	用量统计	展示当前租户下的 span 情况，包括 Span 上报数、Span 存储量。
	告警	展示当前租户下所有告警信息，支持配置告警规则，查看告警事件和告警发送历史。

1.3、应用场景

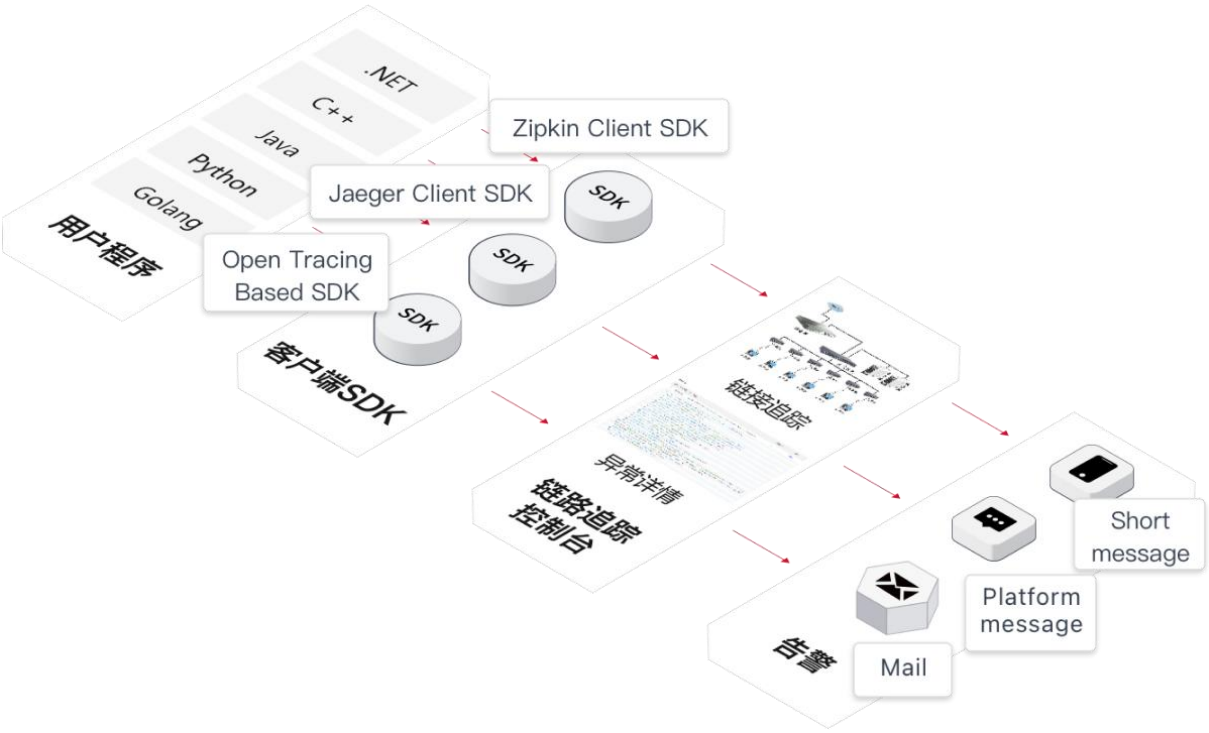
应用场景如下：

微服务架构应用监控

场景说明

微服务架构下的分布式应用在提供更灵活组合的同时也带来了更复杂的监控诊断问题。一个服务出现问题，可能其上的多个应用都会出现异常。通过链路追踪能力，可以帮助用户快速追溯异常原因，定位本质问题。

产品优势



- 应用拓扑展示

自动梳理业务应用，以拓扑图的方式全面展示相关应用调用关系。

- 层级展示，异常可定位至代码级

依据调用链分析能力，支持链路数据层层拆解，提供异常追踪能力，异常可定位到代码级。

- 多语言接入

同一套标准，支持 Java、Go、Python 等多语言接入。

- 告警管理

对应用异常提供可靠的告警管理和多渠道的告警通知。

1.4、基本概念

链路追踪基本概念如下。

链路追踪基本概念

指将一次分布式请求还原成调用链路，并进行集中展示，以便确认各个服务节点的耗时，请

求具体到达了哪台机器，每个服务节点的请求状态等等。

- Trace：从头到尾贯穿整个调用链的 ID，可以代表调用链。
- Span：比如服务 A 发起对服务 B 的调用，这个事件可以看作是一个独立的单元，就是一个 Span。Span 里包含了 ID、时间戳等信息。
- Annotation：表示一个特殊事件，一个 Span 可以包含多个 Annotation。

RPC 与 gRPC

- **RPC （Remote Procedure Call，远程过程调用追踪）**：一个完整的远程调用方案。它包含了:接口规范、传输协议、数据序列化反序列化规范。现在有两台服务器 A 和 B。部署在 A 服务器上的应用，想调用部署在 B 服务器上的另一个应用提供的方法，由于不在一个内存空间，不能直接调用，需要通过网络来达到调用的效果。而这一调用过程，对于开发人员来说是透明的。
- **gRPC**：由 google 开发的一个高性能、通用的开源 RPC 框架，主要面向移动应用开发且基于 HTTP/2 协议标准而设计，同时支持大多数流行的编程语言。

调用链分析工具

工具名称	说明	特点
Zipkin	Twitter 开源的调用链分析工具，目前基于 springcloud sleuth 得到了广泛的使用。	轻量，使用部署简单。
Pinpoint	韩国人开源的基于字节码注入的调用链分析，以及应用监控分析工具。	支持多种插件，UI 功能强大，接入端无代码侵入。

SkyWalking	本土开源的基于字节码注入的调用链分析，以及应用监控分析工具。目前已加入 Apache 孵化器。	支持多种插件，UI 功能较强，接入端无代码侵入。
CAT（Central Application Tracking）	大众点评开源的基于编码和配置的调用链分析，应用监控分析，日志采集，监控报警等一系列的监控平台工具。	/
jaeger	受到 Dapper 和 OpenZipkin 的启发，是由 Uber Technologies 创建 并捐赠给 Cloud Native Computing Foundation 的分布式追踪平台。	/

二、 产品计费

公测期间，您都可以免费使用链路追踪 TAS，span 上报、链路存储、指标存储均不会计入费用。

公测期间，链路追踪使用限制如下。

限制类别	说明
span 上报量（每日）	500w。同一租户在公测期间每天最多上报 500w 个 span。

span 存储时长	7 天。公测期间，数据默认存储 7 天。
告警规则	10 条。同一租户至多可设置 10 条告警规则。

三、 快速入门

3.1 、准备工作

3.1.1 、链路追踪 TAS 接入和鉴权说明

链路追踪 TAS 提供了开源 OpenTelemetry SDK、Jaeger SDK、SkyWalking SDK 三种接入方式。本文介绍了各地域接入点域名以及不同端口对应的数据上报协议。

直接上报

接入点域名

在 sdk 接入那里点击查看 vpce 列表，找到对应的域名。

不同端口支持的协议

端口号	协议	说明
27142	OpenTelemetry HTTP	无

27141	OpenTelemetry GRPC	无
27140	Jaeger HTTP	path：/api/traces
27139	Jaeger GRPC	无
27138	SkyWalking HTTP	无
27137	SkyWalking GRPC	无

获取鉴权 Token

在开通链路追踪服务后，您可以在控制台获得 Token 用作上报数据鉴权，获取方法如下。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开**显示 Token** 开关。
4. 在**客户端采集工具**区域单击需要使用的链路数据采集客户端。
5. 在下方表格的**相关信息**列中，单击接入点信息末尾的复制图标。

3.2、监控 Java 应用

3.2.1、通过 OpenTelemetry 上报 Java 应用数据

在使用链路追踪服务追踪应用的链路数据之前，需要先将应用数据上报上来。本文介绍如何为应用埋点并上报链路数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域单击 **OpenTelemetry**。在下方表格的相关信息列中，获取接入点信息。

使用 OpenTelemetry Java Agent 自动埋点

OpenTelemetry Java Agent 提供了无侵入的接入方式，支持上百种 Java 框架自动上传 Trace 数据。

1. 下载 Java Agent。
2. 通过修改 Java 启动的 JVM 参数上报链路数据。

```
-javaagent:/path/to/opentelemetry-javaagent.jar  //请将路径修改为您文件下载的实际地址。
-Dotel.resource.attributes=service.name=<appName>      //<appName>为应用名。
-Dotel.exporter.otlp.headers=Authentication=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
-Dotel.metrics.exporter=none
```

- 如果您选择直接上报数据，请将<token>替换成从前提条件中获取的 Token，将<endpoint>替换成对应地域的 Endpoint。

例如：

```
point=http://tracing-analysis-dc-bj:8090

-Dotel.metrics.exporter=none
```

- 如果您选择使用 OpenTelemetry Collector 转发，则需删除 -Dotel.exporter.otlp.headers=Authentication=<token>并修改<endpoint>为您本地部署的服务地址。

3.2.2、通过 Jaeger 上报 Java 应用数据

本文介绍如何通过 Jaeger SDK 的方式上报 java 链路数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域选择 **Jaeger**。在下方表格的**相关信息列**中，获取接入点信息。

背景信息

Jaeger 是一款开源分布式追踪系统，兼容 OpenTracingAPI，其主要功能是聚合来自各个异构系统的实时监控数据。目前 OpenTracing 社区已有许多组件可支持各种 Java 框架，例如：ApacheHttpClient、Elasticsearch、JDBC、Kafka、Memcached、Mongo、OkHttp、Redis、SpringBoot、SpringCloud 等。

为 Java 应用埋点

要通过 Jaeger 将 Java 应用数据上报至链路追踪控制台，首先需要完成埋点工作。本示例为 Spring Cloud 组件进行自动埋点。

1. 在已成功运行的 SpringCloud 项目的基础上，打开 pom.xml，添加 Jar 包依赖。

```
<dependency>            <groupId>io.opentracing.contrib</groupId>
<artifactId>opentracing-spring-cloud-starter</artifactId>
<version>0.5.8</version>        </dependency>        <dependency>
<groupId>io.jaegertracing</groupId>
<artifactId>jaeger-client</artifactId>    <version>1.4.0</version>
</dependency>
```

2. 添加 Tracer Bean 组件,第一次运行时 , 需要根据**接入点信息**替换<endpoint>和<token> 。

```
import
org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Bean;
@EnableAutoConfiguration public class TraceConfig {
    @Bean public io.opentracing.Tracer tracer() {
    io.jaegertracing.Configuration config = new
    io.jaegertracing.Configuration("application_name");
    io.jaegertracing.Configuration.SenderConfiguration sender = new
    io.jaegertracing.Configuration.SenderConfiguration();
/**         *  第一次运行时，请设置当前用户的对应的网关          */
    sender.withEndpoint("<endpoint>").withAuthToken("<token>");
    config.withSampler(new
io.jaegertracing.Configuration.SamplerConfiguration().withType("c
onst").withParam(1));
    config.withReporter(new
io.jaegertracing.Configuration.ReporterConfiguration().withSender
(sender).withMaxQueueSize(10000));
    return config.getTracer();
    }
}
```

3.3、监控 Go 应用

3.3.1、通过 OpenTelemetry 上报 Go 应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。

本文介绍如何通过 OpenTelemetry Go SDK 上报 Go 应用数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域单击 **OpenTelemetry**。在下方表格的相关信息列中，获取接入点信息。

背景信息

OpenTelemetry Go SDK 提供了 Go 语言的分布式链路追踪能力，您可以直接使用 OTLP gRPC 或者 HTTP 协议向链路追踪服务端上报数据。

使用 HTTP 协议上报

1. 添加 OpenTelemetry Go 依赖。

```
package main

import (
    "context"
    "flag"
    "go.opentelemetry.io/contrib/instrumentation/net/http/httptrac
e/otelhttptrace"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp
"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/baggage"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptraceht
tp"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"
    "go.opentelemetry.io/otel/trace"
    "io"
    "log"
    "net/http"
    "net/http/httptrace"
    "time"
)
```

2. 初始化 OpenTelemetry Go SDK。

```
func initProvider() func() {
    ctx := context.Background()
    auth := map[string]string{
        "x-ctg-authorization": "xxx", // 替换成 token 信息
    }
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint("xxx:port"),    //替换成 otel http
的上报地址。
       otlptracehttp.WithURLPath("/api/traces"),
        otlptracegrpc.WithHeaders(auth),
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在可观测链路 OpenTelemetry 版后端显示的服务名称。
            semconv.ServiceNameKey.String("otel-go-client-demo"),
            semconv.HostNameKey.String(""),
        )
```

3. client 端上报例子。

```
func sendReq(url string, ctx context.Context) {
    // 构造一个 trace client
    client := http.Client{
        Transport: otelhttp.NewTransport(
            http.DefaultTransport,
            otelhttp.WithClientTrace(func(ctx      context.Context)
*httptrace.ClientTrace {
                return otelhttptrace.NewClientTrace(ctx)
            }),
        ),
    }

    tr := otel.Tracer("example/client")
    err := func(ctx context.Context) error {
        ctx,     span     :=     tr.Start(ctx,     "say     hello",
trace.WithAttributes(semconv.PeerService("ExampleService")))
        defer span.End()
        ctx             =             httptrace.WithClientTrace(ctx,
otelhttptrace.NewClientTrace(ctx))
        req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
        res, err := client.Do(req)
        if err != nil {
            panic(err)
```

4. 服务端上报例子。

```
func handler() {
    uk := attribute.Key("username") //加点属性

    helloHandler := func(w http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        //继续调用下一个服务最终效果是 client-》hello-》hello1
        sendReq("http://localhost:8071/hello1", ctx)
    }

    helloHandler1 := func(w http.ResponseWriter, req *http.Request)
{
        ctx := req.Context()
        span := trace.SpanFromContext(ctx)
        bag := baggage.FromContext(ctx)
        span.AddEvent("handling                              this...",
trace.WithAttributes(uk.String(bag.Member("username").Value())))
        _, _ = io.WriteString(w, "Hello, world!\n")
    }

    otelHandler                                                  :=
otelhttp.NewHandler(http.HandlerFunc(helloHandler), "Hello")
    otelHandler1                                                 :=
otelhttp.NewHandler(http.HandlerFunc(helloHandler1), "Hello1")
```

5. 通过以上步骤，最后就在链路追踪控制台的应用列表页面选择目标应用，查看链路数据。

   具体例子参考 https://github.com/open-telemetry/opentelemetry-go-contrib/tree/main/instrumentation/net/http/httptrace/otelhttptrace/example。

使用 GRPC 协议上报

1. 如需使用 grpc 上报，只需要把上面 initProvider 函数改成如下：

```
func initProviderGrpc() func() {
    ctx := context.Background()
    auth := map[string]string{
        "x-ctg-authorization": "xxx", //接入流程里面获取 token 信息
    }
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint("ip:port"),  //替换成 grpc 接入信息
        otlptracegrpc.WithHeaders(auth),
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在可观测链路 OpenTelemetry 版后端显示的服务名称。

    semconv.ServiceNameKey.String("otel-go-client-demo_provider"),
            semconv.HostNameKey.String(""),
        ),
```

3.3.2 通过 OpenTelemetry 上报 Go 应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。

本文介绍如何通过 OpenTelemetry Go SDK 上报 Go 应用数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域单击 **OpenTelemetry**。在下方表格的相关信息列中，获取接入点信息。

背景信息

OpenTelemetry Go SDK 提供了 Go 语言的分布式链路追踪能力，您可以直接使用 OTLP gRPC 或者 HTTP 协议向链路追踪服务端上报数据。

使用 HTTP 协议上报

1. 添加 OpenTelemetry Go 依赖。

```
package main

import (
    "context"
    "flag"
    "go.opentelemetry.io/contrib/instrumentation/net/http/httptrac
e/otelhttptrace"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp
"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/baggage"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptraceht
tp"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"
    "go.opentelemetry.io/otel/trace"
    "io"
    "log"
    "net/http"
    "net/http/httptrace"
    "time"
)
```

2. 初始化 OpenTelemetry Go SDK。

```
func initProvider() func() {
    ctx := context.Background()
    auth := map[string]string{
        "x-ctg-authorization": "xxx", // 替换成 token 信息
    }
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint("xxx:port"),    //替换成 otel http
的上报地址。
       otlptracehttp.WithURLPath("/api/traces"),
        otlptracegrpc.WithHeaders(auth),
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在可观测链路 OpenTelemetry 版后端显示的服务名称。
            semconv.ServiceNameKey.String("otel-go-client-demo"),
            semconv.HostNameKey.String(""),
        )
```

3. client 端上报例子。

```
func sendReq(url string, ctx context.Context) {
    // 构造一个 trace client
    client := http.Client{
        Transport: otelhttp.NewTransport(
            http.DefaultTransport,
            otelhttp.WithClientTrace(func(ctx      context.Context)
*httptrace.ClientTrace {
                return otelhttptrace.NewClientTrace(ctx)
            }),
        ),
    }

    tr := otel.Tracer("example/client")
    err := func(ctx context.Context) error {
        ctx,     span     :=     tr.Start(ctx,     "say     hello",
trace.WithAttributes(semconv.PeerService("ExampleService")))
        defer span.End()
        ctx             =             httptrace.WithClientTrace(ctx,
otelhttptrace.NewClientTrace(ctx))
        req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
        res, err := client.Do(req)
        if err != nil {
            panic(err)
```

4. 服务端上报例子。

```
func handler() {
    uk := attribute.Key("username") //加点属性

    helloHandler := func(w http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        //继续调用下一个服务最终效果是 client-》hello-》hello1
        sendReq("http://localhost:8071/hello1", ctx)
    }

    helloHandler1 := func(w http.ResponseWriter, req *http.Request)
{
        ctx := req.Context()
        span := trace.SpanFromContext(ctx)
        bag := baggage.FromContext(ctx)
        span.AddEvent("handling                              this...",
trace.WithAttributes(uk.String(bag.Member("username").Value())))
        _, _ = io.WriteString(w, "Hello, world!\n")
    }

    otelHandler                                                  :=
otelhttp.NewHandler(http.HandlerFunc(helloHandler), "Hello")
    otelHandler1                                                 :=
otelhttp.NewHandler(http.HandlerFunc(helloHandler1), "Hello1")
```

5. 通过以上步骤，最后就在链路追踪控制台的应用列表页面选择目标应用，查看链路数据。

   具体例子参考

   https://github.com/open-telemetry/opentelemetry-go-contrib/tree/main/instrumentation/net/http/httptrace/otelhttptrace/example。

使用 GRPC 协议上报

1. 如需使用 grpc 上报，只需要把上面 initProvider 函数改成如下：

```
func initProviderGrpc() func() {
    ctx := context.Background()
    auth := map[string]string{
        "x-ctg-authorization": "xxx", //接入流程里面获取 token 信息
    }
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint("ip:port"),  //替换成 grpc 接入信息
        otlptracegrpc.WithHeaders(auth),
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在可观测链路 OpenTelemetry 版后端显示的服务名称。

    semconv.ServiceNameKey.String("otel-go-client-demo_provider"),
            semconv.HostNameKey.String(""),
        ),
```

3.3.3、通过 SkyWalking SDK 上报 Go 应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。

本文介绍如何通过 SkyWalking SDK 上报 Go 应用数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域选择 **skywalking**。在下方表格的**相关信息列**中，获取接入点信息。

使用 GRPC 协议上报

1. 添加 SkyWalking Go 依赖。

```
package main


import (
    "context"
    "flag"
    "fmt"
    "github.com/SkyAPM/go2sky"
    httpPlugin "github.com/SkyAPM/go2sky/plugins/http"
    "github.com/SkyAPM/go2sky/reporter"
    "io/ioutil"
    "log"
    "net/http"
    "time"
)
```

2. 初始化 SkyWalking Go SDK。

```
    serverPort := flag.String("port", "9891", "server port")
    auth := flag.String("auth", "", "auth") // 鉴权信息
    endpoint := flag.String("endpoint", "endpoint", "endpoint") //
grpc 端口
    authMap := map[string]string{
        "x-ctg-authorization": *auth,
    }
    flag.Parse()
    report,     err     :=     reporter.NewGRPCReporter(*endpoint,
reporter.WithAuthHeader(authMap))
    //report, err := reporter.NewLogReporter()
    tracer,         err         :=         go2sky.NewTracer(service,
go2sky.WithReporter(report))
```

3. client 端上报例子。

```
func do(tracer *go2sky.Tracer, serverPort string, client
*http.Client) {
    for {
        time.Sleep(5*time.Second)
        doClient(tracer, client, "client", "http://localhost:" +
serverPort + "/helloserver")
    }
}

func doClient(tracer *go2sky.Tracer, client *http.Client, spanName
string, url string) {
    clientReq, err1 := http.NewRequest(http.MethodGet, url, nil)
    parentSpan, newCtx, _ :=
tracer.CreateLocalSpan(context.Background())
    parentSpan.SetOperationName(spanName)
    defer parentSpan.End()
    clientReq = clientReq.WithContext(newCtx)
    res, err1 := client.Do(clientReq)
    if err1 != nil {
        fmt.Println(err1, "send req error")
        return
    }
    defer res.Body.Close()
    body, err1 := ioutil.ReadAll(res.Body)
```

4. 服务端上报例子。

```
route := http.NewServeMux()
    route.HandleFunc("/helloserver",                    func(writer
http.ResponseWriter, request *http.Request) {
       upstreamURL := "http://localhost:9891/hello1"
        client, err := httpPlugin.NewClient(tracer)
        clientReq,    err1    :=    http.NewRequest(http.MethodGet,
upstreamURL, nil)
        if err1 != nil {
            writer.WriteHeader(http.StatusInternalServerError)
            log.Printf("unable to create http request error: %v \n",
err)
            return
        }
        subSpan,              newCtx,              _              :=
tracer.CreateLocalSpan(request.Context())
        subSpan.SetOperationName("server2")
        defer subSpan.End()
        clientReq = clientReq.WithContext(newCtx)
        res, err1 := client.Do(clientReq)
        if err1 != nil {
            writer.WriteHeader(http.StatusInternalServerError)
            log.Printf("unable to do http request error: %v \n", err)
            return
        }
```

5. 通过以上步骤，最后就在链路追踪控制台的应用列表页面选择目标应用，查看链路数据。

3.4、监控 Python 应用

3.4.1、通过 OpenTelemetry 上报 Python 应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。

本文介绍如何通过 OpenTelemetry Python SDK 上报 Python 应用数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域单击 **OpenTelemetry**。在下方表格的**相关信息列**中，获取接入点信息。

手动埋点并上报

1. 下载所需包。

```
pip install opentelemetry-api

pip install opentelemetry-sdk

pip install opentelemetry-exporter-otlp
```

2. 在 manual.py 文件中设置 OpenTelemetry 初始化代码。

   - 请将代码中的<token>和<endpoint>替换成前提条件中获取的接入点信息。
   - 请根据实际情况替换代码中的<service-name>(服务名)和<host-name>(主机名)。

```
from opentelemetry import trace, baggage
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter as OTLPSpanGrpcExporter
from opentelemetry.exporter.otlp.proto.http.trace_exporter import
OTLPSpanExporter as OTLPSpanHttpExporter
from opentelemetry.sdk.resources import SERVICE_NAME, Resource,
HOST_NAME
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor


def init_opentelemetry():
    # 设置服务名、主机名
    resource = Resource(attributes={
        SERVICE_NAME: "<service-name>",
        HOST_NAME: "<host-name>"
    })
    # 使用 GRPC 协议上报
    span_processor = BatchSpanProcessor(OTLPSpanGrpcExporter(
        endpoint="<endpoint>",
        headers=("x-ctg-authorization=<token>"),
        insecure=True
    ))
    # 使用 HTTP 协议上报
    trace_provider = TracerProvider(resource=resource,
```

3. 创建 Span。

```
tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("test_span") as child_span:  # 创

建名为 test_span 的 span
```

4. 使用 OpenTelemetry Baggage API 透传业务自定义标签。

创建 **baggage_parent_span** 时通过指定 **attributes** 参数来设置属性。

```
def baggage_and_attribute_usage():
    tracer = trace.get_tracer(__name__)
    global_ctx = baggage.set_baggage("key",
"value_from_global_ctx")
    with  tracer.start_as_current_span(name='baggage_parent_span',
attributes={'attributeKey': 'value'}) as baggage_parent_span:
        parent_ctx = baggage.set_baggage("key",
"value_from_parent_ctx")
        with
tracer.start_as_current_span(name='baggage_child_span',
context=parent_ctx) as baggage_child_span:
            child_ctx = baggage.set_baggage("key",
"value_from_child_ctx")
    # 获取不同 contex 下 key 对应的值
    print(baggage.get_baggage("key", global_ctx))
    print(baggage.get_baggage("key", parent_ctx))
    print(baggage.get_baggage("key", child_ctx))
```

5. 运行程序。

```
python manual.py
```

3.4.2、通过 Jaeger 上报 Python 应用数据

在使用链路追踪控制台追踪应用的链路数据之前，需要通过客户端将应用数据上报至链路追踪。本文介绍如何通过 Jaeger 客户端上报 Python 应用数据。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域选择 **Jaeger**。在下方表格的**相关信息列**中，获取接入点信息。

注意事项

- 针对 Python 语言，最新 v1.25 版本的 Jaeger 仅支持通过 Jaeger Agent 而不支持直接使用 HTTP 协议上报调用链路数据。更多信息，请参见 Jaeger 官方文档。

- 针对 Python 语言，最新 v1.25 版本的 Jaeger 仅支持通过 UDP 协议从 Jaeger Client 端上报至 Jaeger Agent 端。由于 UDP 协议并不保证通信的可靠性，因此为了保证调用链路数据的可靠性，一般情况下需要将 Jaeger Client 端和 Jaeger Agent 端运行在同一个主机内。

接入示例

本示例演示通过 Jaeger Client、Jaeger Agent 上报 Trace 数据。

步骤一：搭建环境

本文演示示例中对 Jaeger Agent、Jaeger Client 和 Python 的版本要求如下。

Python 和 Jaeger Client 环境

Python 版本：3.8.5、Jaeger Client 版本：4.6.0。

在 Python 中安装以下 Python 包配置 Jaeger Client 环境。

```
certifi==2021.5.30

charset-normalizer==2.0.4

idna==3.2

jaeger-client==4.6.0

opentracing==2.4.0

requests==2.26.0

six==1.16.0

threadloop==1.0.2

thrift==0.13.0

tornado==6.1

urllib3==1.26.6
```

步骤二：创建 Tracer 对象

1. 创建包含如下内容的 Python 文件。通过以下代码创建 Tracer 对象，并通过 Tracer 对象创建 Span 来上报数据至链路追踪 TAS 后台。

```
import request
from jaeger_client import Config


def construct_span(tracer):
    with tracer.start_span('ctyunTestSpan') as span:
        span.log_kv({'event': 'test message', 'life': 42})
        print("tracer.tages: ", tracer.tags)
        with tracer.start_span('ctyunTestChildSpan', child_of=span)
as child_span:
            span.log_kv({'event': 'down below'})
        return span


def extract_span_data(span):
    span_data = {
    "trace_id": span.context.trace_id,
    "span_id": span.context.sapn_id,
    "operation_name": span.operation_name,
}
    return span_data


def generate_trace_data():
    # 使用 Jaeger 配置创建 Jaeger 客户端
config = Config(
    config={
```

2. 执行新建的 Python 文件。

在链路追踪控制台查看数据

1. 登录链路追踪控制台，在左侧导航栏单击「**应用列表**」。

2. 在**应用列表**页面单击目标应用名称。

3. 在**应用总览**页面，您可以查看该应用的性能关键指标和拓扑图详情。

4. 在左侧导航栏单击「**应用详情**」。在**应用详情**页面，您可以查看该应用的概览信息。

5. 在**应用详情**页面单击「**调用链查询**」页签。在**调用链查询**页签，您可以查看该应用的调用链路。

参考信息

此处提供了 Jaeger 常见的使用方法。

- 创建 Trace。

```
from jaeger_client import Config

def init_jaeger_tracer(service_name='your-app-name'):

config = Config(config={}, service_name=service_name)

return config.initialize_tracer()
```

- 创建和结束 Span。

```
# 开始无 Parent 的 Span。

tracer.start_span('TestSpan')

# 开始有 Parent 的 Span。

tracer.start_span('ChildSpan', child_of=span)

span.finish()
```

- 透传 SpanContext。

```
# 将 spanContext 透传到下一个 Span 中（序列化）。
tracer.inject(
       span_context=span.context,          format=Format.TEXT_MAP,
carrier=carrier
    )
# 解析透传过来的 spanContxt（反序列化）。
span_ctx = tracer.extract(format=Format.TEXT_MAP, carrier={})
```

3.5、监控 Node.js 应用

3.5.1、通过 OpenTelemetry 上报 Node.js 应用数据

本文介绍了如何通过 OpenTelemetry 将 Node.js Express 应用接入链路追踪。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。
2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。
3. 打开显示 Token 开关。
4. 在客户端采集工具区域单击 OpenTelemetry。在下方表格的**相关信息列**中，获取接入点信息。

接入步骤

1. 引入依赖包。

```
  "dependencies": {
      "@hapi/hapi": "^19.2.0",
      "@opentelemetry/api": "^1.0.2",
      "@opentelemetry/exporter-jaeger": "^0.25.0",
      "@opentelemetry/exporter-zipkin": "^0.25.0",
      "@opentelemetry/instrumentation": "^0.25.0",
      "@opentelemetry/instrumentation-hapi": "^0.23.0",
      "@opentelemetry/instrumentation-http": "^0.25.0",
      "@opentelemetry/sdk-trace-node": "^0.25.0",
      "@opentelemetry/sdk-trace-base": "^0.25.0",
      "axios": "^0.21.1"
    }
```

2. 初始化 node.js Provider。

```
const { Resource } = require("@opentelemetry/resources");
const {
   OTLPTraceExporter,
 } = require("@opentelemetry/exporter-trace-otlp-proto");
const { NodeTracerProvider } = require("@opentelemetry/sdk-trace-node");
const { registerInstrumentations } = require("@opentelemetry/instrumentation");
const { HttpInstrumentation } = require("@opentelemetry/instrumentation-http");
const {
  SemanticResourceAttributes,
} = require("@opentelemetry/semantic-conventions");
const {
   SimpleSpanProcessor,
   ConsoleSpanExporter,
   BatchSpanProcessor
 } = require("@opentelemetry/sdk-trace-base");

const provider = new NodeTracerProvider({
  resource: new Resource({
    [SemanticResourceAttributes.HOST_NAME]:
require("os").hostname(), // 主机名
    [SemanticResourceAttributes.SERVICE_NAME]: "service-name",
```

3. client 上报 demo。

```
const api = require('@opentelemetry/api');
const { Span } = require('@opentelemetry/sdk-trace-base');
const axios = require('axios').default;
const tracer = api.trace.getTracer('hapi-example');

function makeRequest() {
  const span = tracer.startSpan('client.makeRequest()', {
    kind: api.SpanKind.CLIENT,
  });

  api.context.with(api.trace.setSpan(api.ROOT_CONTEXT, span), async
() => {
    try {
      const res = await axios.get('http://localhost:8081/run_test');
      span.setStatus({ code: api.SpanStatusCode.OK });
      console.log(res.statusText);
    } catch (e) {
      span.setStatus({  code:  api.SpanStatusCode.ERROR,  message:
e.message });
    }
    span.end();
    console.log('Sleeping 5 seconds before shutdown to ensure all
records are flushed.');
    setTimeout(() => { console.log('Completed.'); }, 5000);
```

4. 服务端上报 demo。

```
const api = require('@opentelemetry/api');
function runTest(_, h) {
  const currentSpan = api.trace.getSpan(api.context.active());
  const { traceId } = currentSpan.spanContext();
  console.log(`traceid: ${traceId}`);
}
```

5. 通过上面步骤就可以在控制台查看 node.js 产生的链路数据了。更详细的接入 demo 请参考 https://github.com/open-telemetry/opentelemetry-js-contrib/tree/main/examples。

在控制台查看 Trace

在链路追踪控制台的「**应用列表**」页面选择目标应用，查看链路数据。

3.5.2、通过 Jaeger 上报 Node.js 应用数据

本文介绍了如何将 Node.js 应用接入链路追踪服务。

前提条件

获取接入点信息。

1. 登录链路追踪 TAS 控制台。

2. 在左侧导航栏单击「**链路配置**」，然后在右侧页面单击「**接入点信息**」页签。

3. 打开显示 Token 开关。

4. 在客户端采集工具区域选择 **Jaeger**。在下方表格的**相关信息列**中，获取接入点信息。

操作步骤

1. 引入依赖包。

```
"dependencies": {
    "jaeger-client": "^3.12.0"
  }
```

1. 初始化 provider。

```
const initTracer = require("jaeger-client").initTracer;


const config = {
    serviceName: 'name',
    sampler: {
        type: "const",
        param: 1
    },
    reporter: {
        collectorEndpoint: "" // jaeger http 上报地址
    },
};
const header = {} //鉴权 header
const tracer = initTracer(config, header);
```

1. 上报数据。

```
const span = tracer.startSpan('spanStart');

span.setTag('span.kind', 'server');

// 设置标签（可选，支持多个）

span.setTag('tagName', 'tagValue');

// 设置事件（可选，支持多个）

span.log({ event: 'timestamp', value: Date.now() });

// 标记 Span 结束

span.finish();
```

1. 通过以上方式可以在链路追踪控制台列表看到 node.js 上报的链路数据。

四、 用户指南

4.1 、应用概览

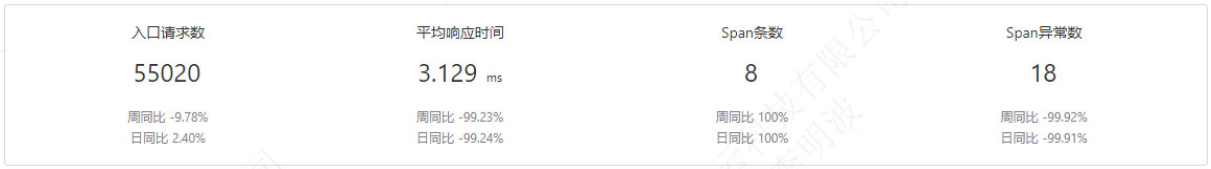
应用概览显示当前租户下所有应用整体的核心信息，包括核心指标、应用列表、告警等信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**概览**」。

功能说明

关键指标



展示链路追踪的关键指标信息，包括

- **入口请求数**：显示该租户下，当前筛选时间段内的总请求数。
- **平均响应时间**：显示该租户下，当前筛选时间段内，（所有应用提供服务响应时间+所有应用依赖服务响应时间）/总请求量。
- **Span 条数**：显示该租户下，当前筛选时间段内，span 上报的总量。
- **Span 异常数**：显示该租户下，当前筛选时间段内，上报 span 中出现异常的数量。

应用监控列表

以应用为维度，显示当前租户下所有接入链路追踪 TAS 的应用的信息，包括

- **应用名称**：该应用的名称，点击打开对应的应用详情。
- **错误率**：该应用在筛选时间段内，请求的错误数/总请求数。
- **请求总数**：该应用在筛选时间段内的 HTTP 入口的总请求数。

- **平均响应时间**：该应用在筛选时间段内的 HTTP 入口请求的平均响应时间。
- **响应时间趋势**：显示该应用在筛选时间段内响应时间的变化趋势图。

入口统计列表

以链路入口为维度，显示当前租户下所有链路信息。

- **TraceID**：调用链的唯一标识。点击显示详情弹窗如“Trace 详情”。
- **产生日志时间**：该调用链出现日志记录的时间点。
- **接口名称**：显示发起 API 调用时的接口名称，完整调用链中涉及的接口信息在 trace 详情中查看。
- **所属应用**：显示当前应用实例所属应用的名称，该调用链涉及的其他应用信息在 trace 详情中查看。
- **状态**：与 HTTP 状态码对应。
- **耗时**：一个完整的链路调用所消耗的时间。
- **服务端**：调用链中第一段的被调用的应用的 IP 端口。
- **客户端**：调用链中第一段的发起调用的应用的 IP 地址。
- **操作**：点击显示详情弹窗如“Trace 详情”。

告警

显示租户在当前筛选时间段内的告警信息，包括

- **事件名称**：显示告警规则名称。
- **事件描述**：显示根据告警规则中设置的告警内容模板产生的实际内容。
- **创建时间**：告警事件产生时间。
- **事件数量**：显示该事件产生的数量。
- **事件状态**：显示事件当前状态是告警中、已恢复、静默。

4.2、应用管理

4.2.1、应用总览

以应用为维度，统计整个应用的关键指标，帮助您快速掌握应用的整体链路状况。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。

4. 在左侧导航栏中选择「**应用总览**」，您可以在应用总览页面顶部选择「**概览**」或「**拓扑图**」页签查看相应信息。

功能说明

概览

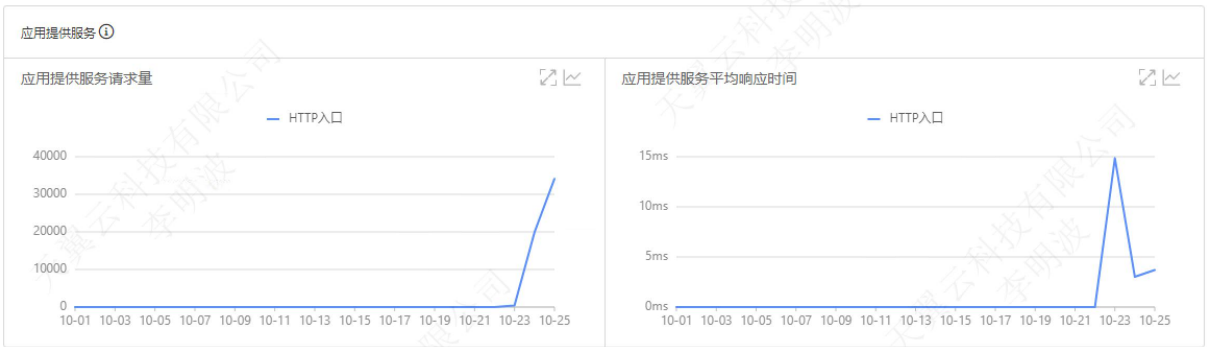
总览指标

指标：

- **总请求量**：筛选时间段内，应用提供服务请求量+应用依赖服务请求量。
- **平均响应时间**：筛选时间段内，（所有应用提供服务响应时间+所有应用依赖服务响应时间）/总请求量。
- **异常**：exception，筛选时间段内，该应用报的异常数。
- **Span 条数**：筛选时间段内，span 上报的总量。

对比：提供周同比和日同比信息。

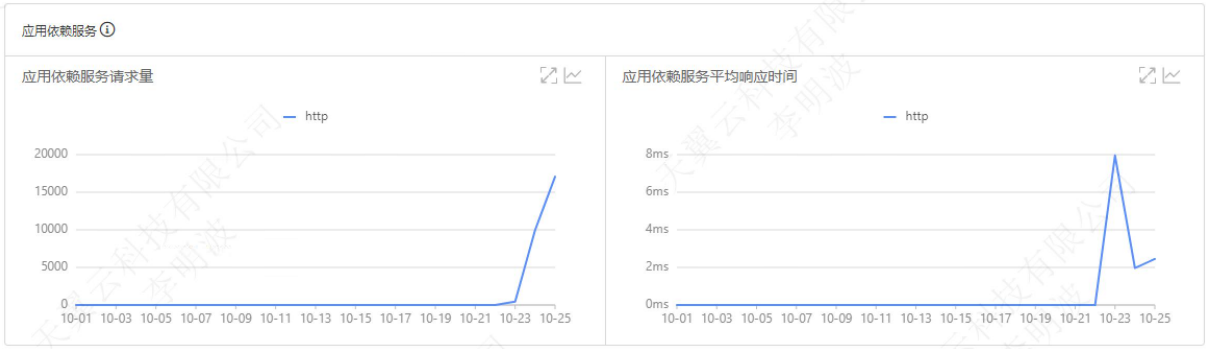
应用提供服务



因用户访问该应用而产生的数据，例如用户在浏览器中访问该应用。

- **应用提供服务请求量**：筛选时间段内，用户向该应用发起的请求数量。
- **应用提供服务平均响应时间**：响应时间是指从用户发起请求到服务端给予反馈的时长，平均响应时间是筛选时间段内，所有请求的响应时间的平均值。

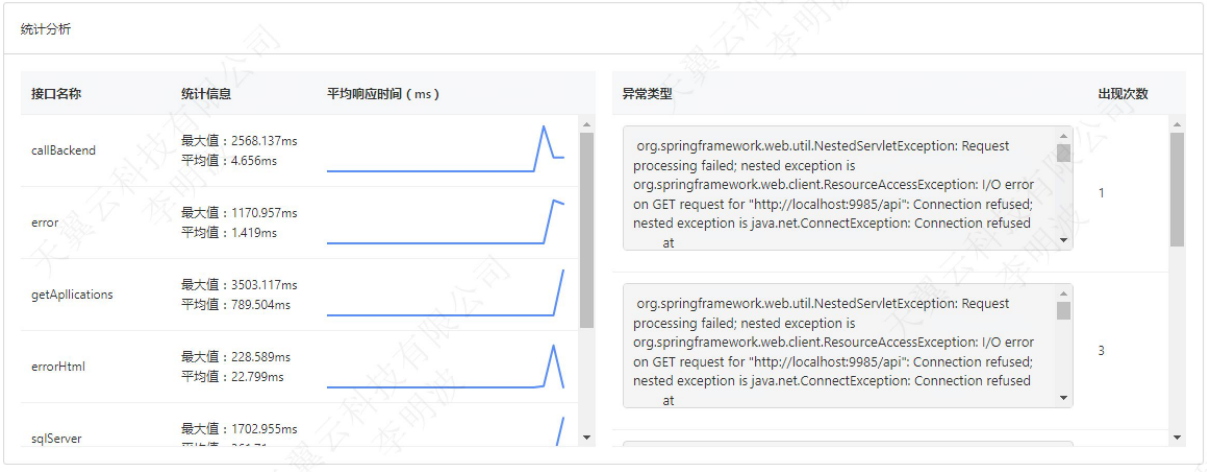
应用依赖服务



因该应用访问其他服务而产生的数据，例如该应用访问数据库。

- **应用依赖服务请求量**：筛选时间段内，该应用向其他服务发起的请求数量。
- **应用依赖服务平均响应时间**：响应时间是指从该应用发起请求到其他服务给予反馈的时长，平均响应时间是筛选时间段内，所有请求的响应时间的平均值。

统计分析



以接口维度来统计调用的情况。

接口统计

- **接口名称**：被调用的接口的名称。
- **最大值**：筛选时间段内，该接口被调用的响应时间的最大值。
- **平均值**：筛选时间段内，该接口被调用的平均响应时间。
- **平均响应时间**：筛选时间段内，每天的平均响应时间的趋势图。

异常情况

- **异常类型**：显示异常明细，与点击详情按钮看到的内容一致。
- **出现次数**：筛选时间段内，此类异常出现的次数。

拓扑分析

拓扑图

拓扑图是一种以图形化方式展示应用之间关系的图表，帮助开发人员或运维人员了解应用程序的整体结构和运行状况。

拓扑图通常包括以下信息：

- **应用或服务的组成部分**：例如数据库、缓存、消息队列、Web 服务器等。
- **组件之间的依赖关系**：例如一个组件调用另一个组件的接口。
- **基础指标**：例如请求量、时延等，帮助开发/运维人员了解组件的运行状况和性能状况。

通过分析拓扑图，开发/运维人员可以快速定位应用中的问题，并进行及时的排查和修复。拓扑图还可以帮助开发/运维人员进行容量规划和性能优化，以提高应用程序或服务的性能和可靠性。

实例健康

通过“实例平均响应时间”判断该应用所有实例的健康程度。

- **健康程度**
  - 正常：小于 500ms。
  - 警告：大于等于 500ms，小于 1s。
  - 严重：大于等于 1s。
- **实例的调用详情**
  - **调用类型**：根据该应用调用和被调用对象的不同来区分类型，包括 http 入口、调用 redis、调用 http、调用 mysql、未知调用等等。
  - **调用次数**：筛选时间段内，调用的次数。
  - **平均响应时间**：筛选时间段内，调用等待响应的平均时长。
  - **错误率**：筛选时间段内，调用错误次数/调用次数。

请求数、响应时间、错误数

将应用的调用详情按天展示，包括请求数、响应时间、错误数。

统一交互操作说明：

- 将光标移到统计图上，可以查看光标所至时间点的数据详情。
- 单击图标，可以将当前图表放大显示。
- 单击图标，查看该指标在某个时间段的统计情况或对比不同日期同一时间段的统计情况。

4.2.2、应用详情

4.2.2.1、概览

应用的概览可以提供核心监控数据，帮助您快速掌握某个应用的具体情况。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**应用详情**」或「**接口调用**」，您可以在应用详情页面切换至「**概览**」页签，在左侧关键指标中**选择不同的应用实例/接口**，可查看该应用实例/接口相应的概览信息。

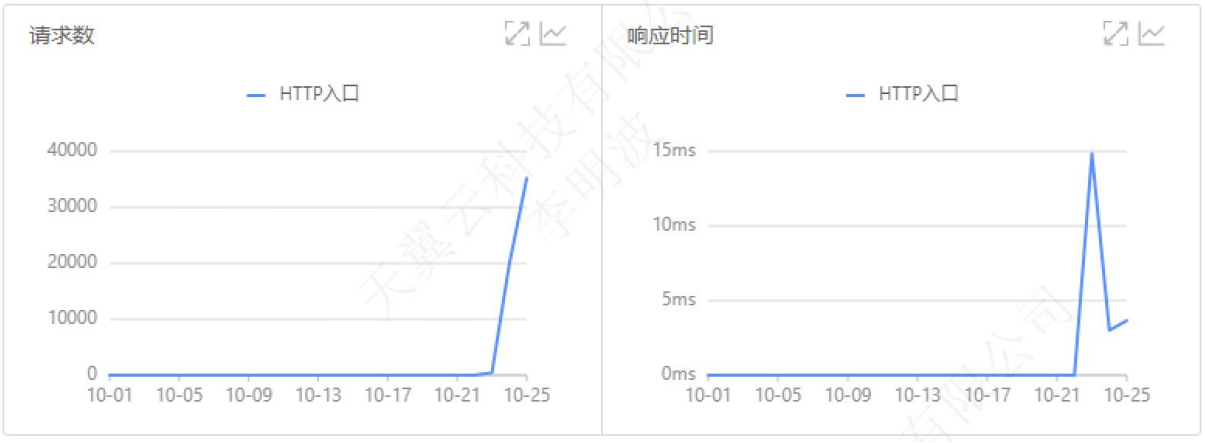
功能说明

展示拓扑图

RESET

拓扑图是一种以图形化方式展示应用之间关系的图表，帮助开发人员或运维人员了解应用程序的整体结构和运行状况。详情参见“应用总览-拓扑分析-拓扑图”

展示请求数、响应时间、错误数、HTTP-状态码统计



- **请求数**：该应用实例/接口在筛选时间段内的 HTTP 入口的总请求数。
- **响应时间**：该应用实例/接口在筛选时间段内的 HTTP 入口请求的日均响应时间。
- **错误数**：该应用实例/接口在筛选时间段内的 HTTP 入口请求的错误数。
- **HTTP-状态码统计**：该应用实例/接口在筛选时间段内的 HTTP 入口请求的状态码。
  - 5xx：服务器异常，服务器在处理请求的过程中发生错误。
  - 4xx：客户端异常，请求包含语法错误或无法完成请求。
  - 3xx：重定向问题，需要进一步操作。
  - 2xx：成功，服务器成功接收请求并执行。
  - 200：请求成功。

统一交互操作说明：

- 将光标移到统计图上，可以查看光标所至时间点的数据详情。
- 单击图标，可以将当前图表放大显示。
- 单击图标，查看该指标在某个时间段的统计情况或对比不同日期同一时间段的统计情况。

4.2.2.2、SQL 调用分析

SQL 调用分析通常是指对一段时间内对 SQL 语句的**调用次数、耗时**的统计分析。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。

2. 在左侧导航栏中选择「**应用列表**」。

3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。

4. 在左侧导航栏中选择「**数据库调用**」，您可以在应用详情页面切换至「**SQL 调用分析**」页签，在左侧关键指标中**选择不同的 SQL**，可查看该 SQL 相应的概览信息。

功能说明

SQL 调用统计图

显示该应用在筛选时间段内的所有 SQL 语句的调用次数和平均耗时。

SQL 调用统计表

所属应用	数据库类型	SQL语句	平均耗时(ms)	调用次数
mall-mall-server	mysql	CREATE TABLE IF N...	23.698	18
mall-mall-server	mysql	CREATE TABLE IF N...	19.765	18
mall-mall-server	mysql	SELECT id,name,des...	11.663	25276
mall-mall-server	mysql	SELECT COUNT(?) F...	10.88	53978
mall-mall-server	mysql	SELECT id,name,des...	9.745	54146

以 SQL 语句为维度，详细显示不同 SQL 语句各自的调用次数和平均耗时。

- **所属应用**：显示 SQL 语句所属的应用名称。
- **数据库类型**：显示当前应用使用的数据库类型。
- **SQL 语句**：显示具体执行的 SQL 语句，不显示 SQL 语句中的具体参数。
- **平均耗时**：包括执行时间和等待时间两个方面。
  - 执行时间：SQL 语句的执行时间，包括 CPU 时间和 IO 时间，可以反映 SQL 语句的性能和效率。
  - 等待时间：SQL 语句等待的时间，包括等待锁的时间、等待 IO 的时间等，可以反映 SQL 语句的并发性和资源争用情况。
- **调用次数**：SQL 语句被执行的次数，可以反映 SQL 语句的重要性和使用频率。

统一交互操作说明：

- 将光标移到统计图上，可以查看光标所至时间点的数据详情。
- 单击图标，可以将当前图表放大显示。
- 单击图标，查看该指标在某个时间段的统计情况或对比不同日期同一时间段的统计情况。

4.2.2.3、链路上游

链路上游是当前接口的调用者。上游应用向当前接口发起请求，并等待当前接口返回结果。

通过链路上游分析可以查看上游接口的**请求量、平均延时、错误数**信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**接口调用**」，您可以在详情页面切换至「**链路上游**」页签，在左侧关键指标中**选择不同的接口**，可查看该应用接口相应的概览信息。

功能说明

- 以卡片形式显示该应用实例在当前筛选时间段内的链路上游信息，默认全部展开，可以操作展开/收起，每个卡片上显示一个接口的**名称、请求量、平均延时**（即：耗时/响应时间）和**错误数**。
- 支持对接口名称进行搜索，支持对请求量/平均延时/错误数进行排序。

通过记录链路上游和链路下游，可以在链路追踪 TAS 系统中查看接口与接口之间的调用链路，并追踪请求和响应的流程。

4.2.2.4、链路下游

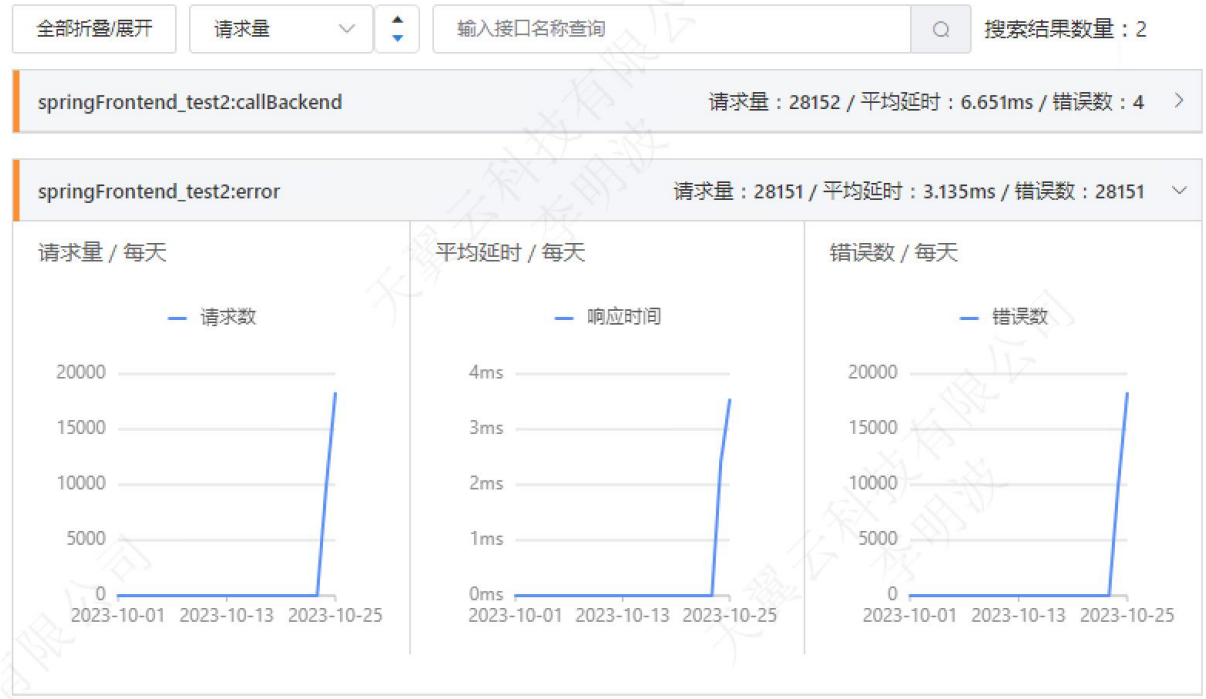
链路下游指当前应用的被调用方。当前应用程序向下游接口发起请求，并等待下游接口返回结果。

通过链路下游分析可以查看下游接口的**请求量、平均延时、错误数**信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**接口调用**」，您可以在详情页面切换至「**链路下游**」页签，在左侧关键指标中**选择不同的应用接口**，可查看该接口相应的概览信息。

功能说明



- 以卡片形式显示该应用接口在当前筛选时间段内的下游接口，默认全部展开，可以操作展开/收起，每个卡片上显示一个接口的**名称、请求量、平均延时**（即：耗时/响应时间）和**错误数**。
- 支持对接口名称进行搜索，支持对请求量/平均延时/错误数进行排序。

通过记录链路上游和链路下游，可以在链路追踪 TAS 系统中查看接口与接口之间的调用链路，并追踪请求和响应的流程。

4.2.2.5、调用链查询

展示该**应用实例/接口/SQL** 涉及的**调用链信息**，包括 TraceID、产生时间、接口名称、耗时、状态信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。

2. 在左侧导航栏中选择「**应用列表**」。

3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。

4. 在左侧导航栏中选择「**应用详情**」或「**接口调用**」或「**数据库调用**」，您可以在应用详情页面切换至「调用链查询」页签，在左侧关键指标中**选择不同的应用实例/接口/SQL**，可查看该应用实例/接口/SQL 相应的概览信息。

功能说明

产生时间	接口名称	所属应用	耗时(ms)	状态	TraceId
2023-10-25 17:06:23.158	error	springFrontend_t...	0.967		00000000000000...
2023-10-25 17:06:23.143	callBackend	springFrontend_t...	3.125		00000000000000...
2023-10-25 17:06:22.649	error	springFrontend_t...	1.433		00000000000000...
2023-10-25 17:06:22.634	callBackend	springFrontend_t...	3.873		00000000000000...
2023-10-25 17:06:19.608	error	springFrontend_t...	1.023		00000000000000...
2023-10-25 17:06:19.594	callBackend	springFrontend_t...	2.966		00000000000000...
2023-10-25 17:06:19.593	error	springFrontend_t...	1.163		00000000000000...
2023-10-25 17:06:19.578	callBackend	springFrontend_t...	3.713		00000000000000...
2023-10-25 17:06:16.541	error	springFrontend_t...	1.221		00000000000000...
2023-10-25 17:06:16.528	callBackend	springFrontend_t...	3.127		00000000000000...

10条/页 共 57551 条 1 2 3 4 5 6 ... 5756

显示筛选时间段内，该应用实例/接口/SQL 涉及的调用链信息。

- **产生时间**：该调用链产生的时间点。
- **接口名称**：显示发起 API 调用时的接口名称，完整调用链中涉及的接口信息在 trace 详情中查看。
- **所属应用**：显示当前应用实例所属应用的名称，该调用链涉及的其他应用信息在 trace 详情中查看。
- **耗时**：一个完整的链路调用所消耗的时间。

- **状态**：与 HTTP 状态码对应。
- **TraceID**：调用链的唯一标识。点击显示详情弹窗如“Trace 详情”。

4.2.3、分析视角

4.2.3.1、应用实例

提供**应用实例级别**的监控详情。展示概览和调用链信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**应用详情**」，您可以在应用详情页面切换不同的页签，诸如「**概览**」、「**调用链查询**」等等查看相应信息。

功能说明

实例列表

以实例为维度，展示各个应用实例的**名称、请求数、响应时间、错误数和异常数**。

- 搜索：支持搜索实例名称。
- 排序：支持按照请求数/响应时间/错误数/异常数进行排序。
- 操作：点击实例名，右侧窗口切换成该实例对应的数据；点击数据，对应列倒序排序，同事右侧窗口切换成对应实例的数据。
- 展示：通过不同的圆点颜色区分是否响应过慢。
  - 绿色：响应时间<500ms。
  - 橙色：500ms<响应时间<1000ms。
  - 红色：1000ms<响应时间。

各维度指标详情

- **概览**：详情见“应用详情-概览”，统计维度是实例。
- **调用链查询**：详情见“应用详情-调用链查询”，统计维度是实例。

4.2.3.2、接口调用

提供**应用接口级别**的监控详情。通过丰富的链路分析报表，展示包括链路上下游及调用链分析。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**接口调用**」，您可以在应用详情页面切换不同的页签，诸如「**概览**」、「**调用链查询**」等等查看相应信息。

功能说明

接口列表

以接口为维度，展示各个接口的**接口名称、请求数、响应时间、错误数和异常数**。

- 搜索：支持搜索接口名称。

- 排序：支持按照请求数/响应时间/错误数/异常数进行排序。
- 操作：点击接口名，右侧窗口切换成该接口对应的数据；点击数据，对应列倒序排序，同事右侧窗口切换成对应接口的数据。
- 展示：通过不同的圆点颜色区分是否响应过慢。
  - 绿色：响应时间<500ms。
  - 橙色：500ms<响应时间<1000ms。
  - 红色：1000ms<响应时间。

各维度指标详情

- **概览**：详情见“应用详情-概览”，统计维度是接口。
- **链路上游**：详情见“应用详情-链路上游”，统计维度是接口。
- **链路下游**：详情见“应用详情-链路上游”，统计维度是接口。
- **调用链查询**：详情见“应用详情-调用链查询”，统计维度是接口。

4.2.3.3、数据库调用

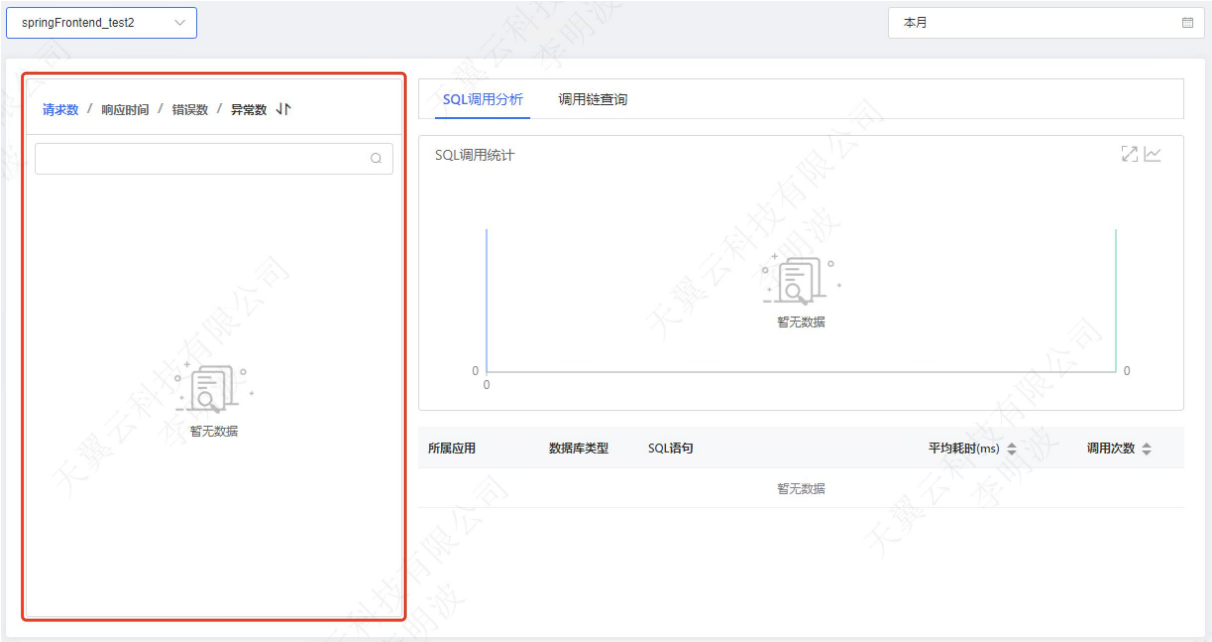
提供**数据库维度**的监控详情。展示 SQL 和调用链信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**应用详情**」，您可以在应用详情页面切换不同的页签，诸如「**SQL调用分析**」、「**调用链查询**」等等查看相应信息。

功能说明

SQL 列表



以 SQL 为维度，展示各个 SQL 的**名称、请求数、响应时间、错误数和异常数**。

- 搜索：支持搜索数据库名称。
- 排序：支持按照请求数/响应时间/错误数/异常数进行排序。
- 操作：点击数据库名，右侧窗口切换成该 SQL 对应的数据；点击数据，对应列倒序排序，同事右侧窗口切换成对应 SQL 的数据。
- 展示：通过不同的圆点颜色区分是否响应过慢。
  - 绿色：响应时间<500ms。
  - 橙色：500ms<响应时间<1000ms。
  - 红色：1000ms<响应时间。

各维度指标详情

- **概览**：详情见“应用详情-SQL 调用分析”，统计维度是数据库。
- **调用链查询**：详情见“应用详情-调用链查询”，统计维度是数据库。

4.2.4、应用设置

自定义配置可以让您对具体某个应用进行单独管理，包是否关联业务日志。

功能入口

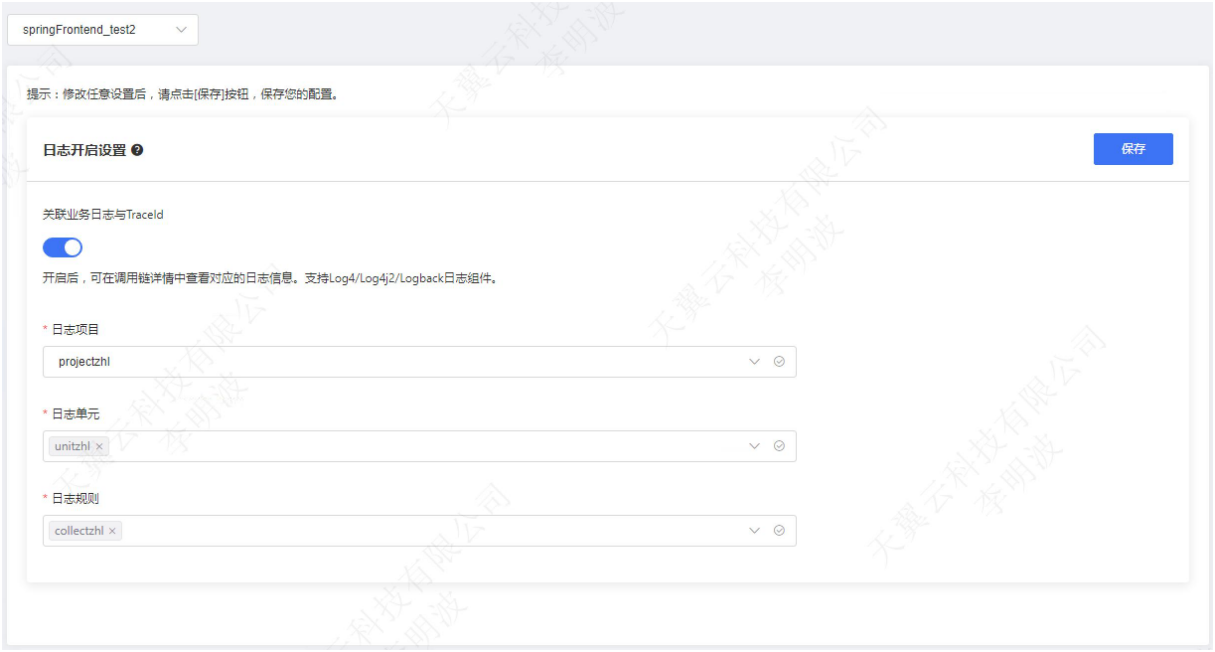
1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**应用设置**」，您可以在应用设置中修改各项可配置信息。

功能说明

日志开启设置

支持设置是否将日志和链路进行关联。如果开启，则可以在调用链详情中查看对应的日志信息；反之，则看不到日志信息。

设置时，支持选择日志存储路径：日志项目-日志单元-日志规则。



开启效果

开启“关联业务日志与 traceID”后，在链路详情看到的效果如下。

4.3、多维查询

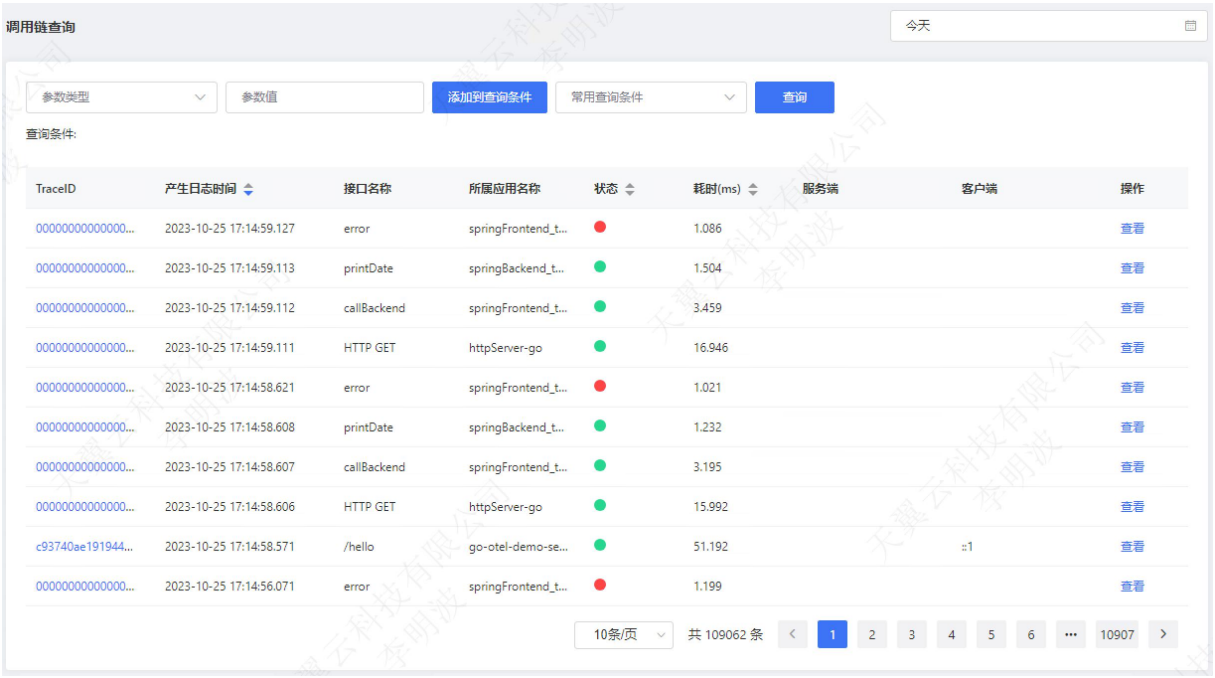
4.3.1、调用链查询

展示当前租户下所有调用链路信息。您可以根据多个筛选条件租户查询您想看的调用链，可以点击 TraceID 查看具体的调用链详情。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**多维查询**」。

功能说明



关键信息展示与查询

- **TraceID**：调用链的唯一标识。
- **产生日志时间**：该调用链产生日志的时间点。
- **接口名称**：显示发起 API 调用时的接口名称，完整调用链中涉及的接口信息在 trace 详情中查看。
- **所属应用名称**：显示当前应用实例所属应用的名称，该调用链涉及的其他应用信息在

trace 详情中查看。

- **状态**：显示正常/异常。
- **耗时**：一个完整的链路调用所消耗的时间。
- **服务端**：调用链中第一段的被调用的应用的 IP 端口。
- **客户端**：调用链中第一段的发起调用的应用的 IP 地址。
- **操作**：「查看」，点击显示详情弹窗如“Trace 详情”。

具体链路详情

详见“Trace 详情”。

4.3.2、Trace 详情

显示 Trace 的详细信息，包括调用栈及具体每个方法的耗时、日志详情等等。

功能入口

查看该租户下所有调用链路中某个的链路详情

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**多维查询**」。
3. 点击「**TraceID**」，打开 Trace 详情弹窗。

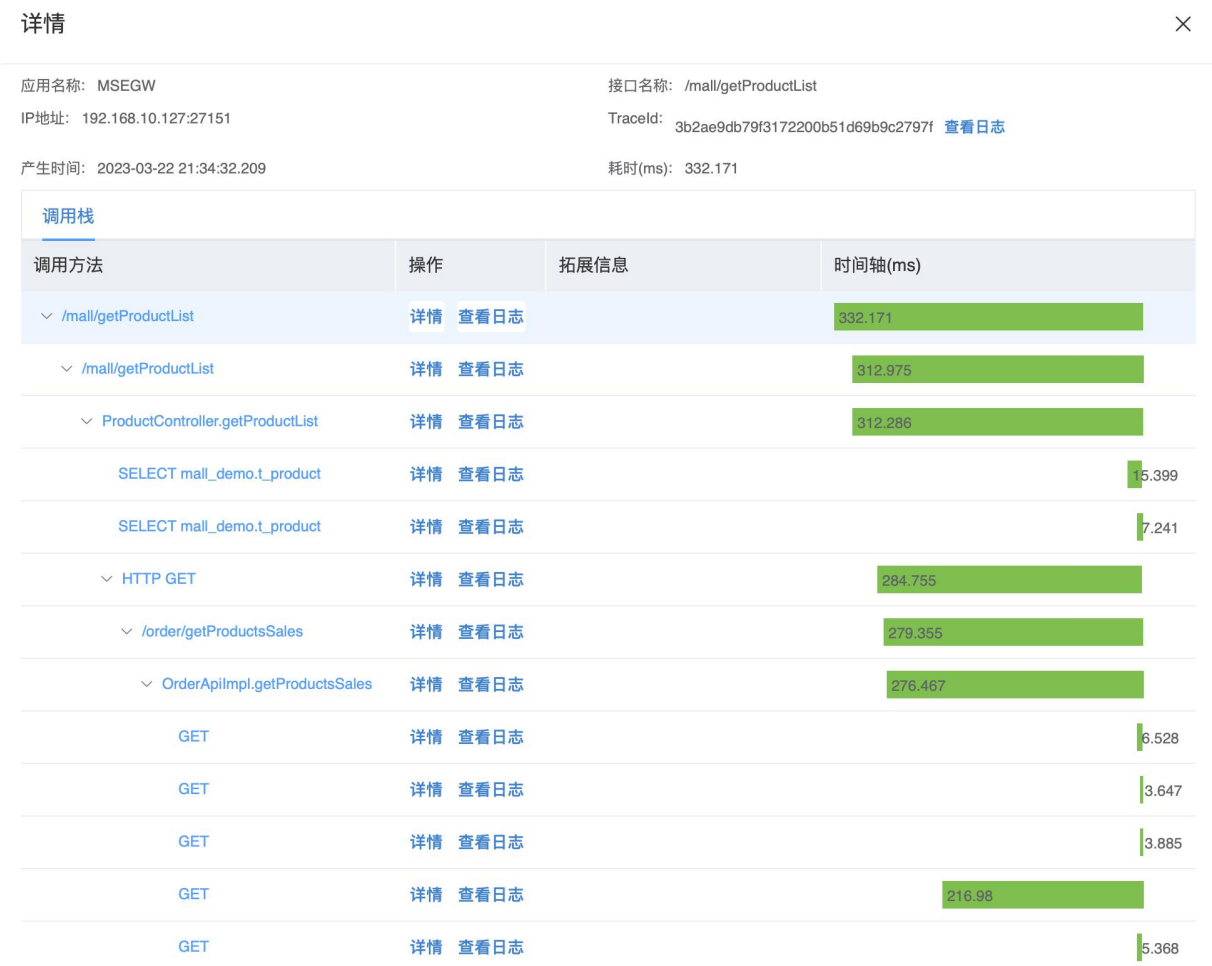
查看应用实例/接口相关的调用链路中某个的链路详情

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**应用列表**」。
3. 在应用列表中选择您想查看的应用，点击「**应用名称**」打开新的应用详情链接。
4. 在左侧导航栏中选择「**应用详情**」或「**接口调用**」或「**数据库调用**」，您可以在应用详

情页面切换至「**调用链查询**」页签，在左侧关键指标中**选择不同的应用实例/接口/SQL**，可查看该应用实例/接口/SQL 相应的概览信息。

5. 点击「**TraceID**」，打开 Trace 详情弹窗。

功能说明



基础信息

- **应用名称**：显示当前选中的“调用方法”所属的应用的名称。
- **接口名称**：显示当前选中的“调用方法”所属的接口的名称。
- **IP 地址**：显示当前选中的“调用方法”所属的应用实例的 IP 地址。
- **TraceID**：显示 TraceID，当前调用链的唯一标识。点击「查看日志」打开“ALS-检索

分析”页面。

- **产生时间**：发起调用的时间点。
- **耗时**：显示当前选中的「调用方法」从发起调用到返回结果的时间。

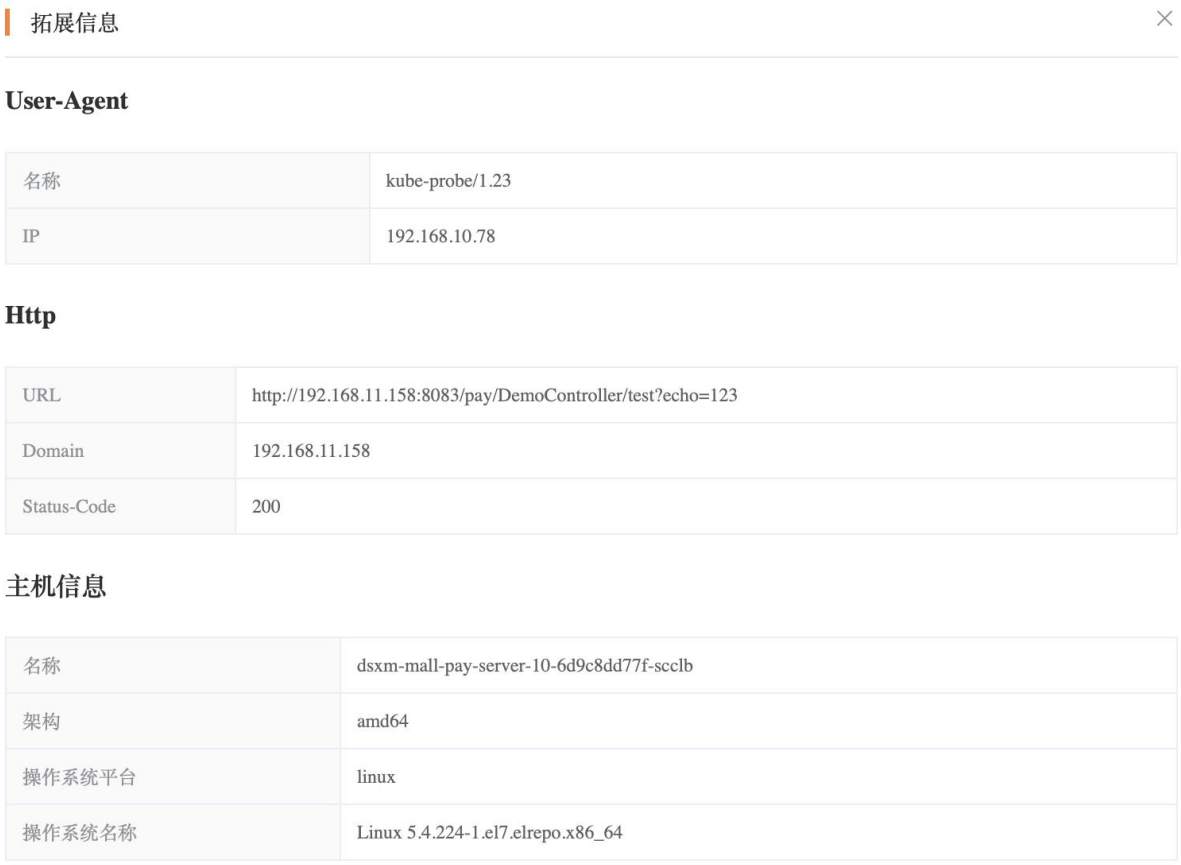
调用栈

以调用树的方式显示具体的调用信息。

- **调用方法**：显示调用方法名称。
- **操作**：
  - **详情**：点击详情，打开该调用方法的拓展信息弹窗。显示 Agent、http、主机、Process、Thread、DB、Messaging、目标服务信息。
  - **查看日志**：点击打开“ALS-检索分析”页。
- **拓展信息**：当该调用存在异常/错误信息时，会在此处显示异常堆栈信息。
- **时间轴**：以时间轴的形式显示各个调用方法的耗时。

拓展信息

点击「详情」按钮，打开弹窗如下。



显示该方法其他相关信息，包括：

User-Agent（用户探针）

- **名称**：接入该应用的探针名称。
- **IP**：该探针接入的应用实例的主机 IP。

Http（协议）

- **URL（Uniform Resource Locator）**：统一资源定位符，是一种用于标识互联网上资源的地址，包括协议类型、主机名、端口号、路径和查询参数等信息。
- **Domain（域名）**：是指互联网中的机器和服务的名称，类似于电话号码的概念。域名由多个部分组成，按照从右到左的顺序，依次表示顶级域名、二级域名、三级域名等。
- **Status-Code（状态码）**：是指 HTTP 服务器返回给客户端的响应状态码，用于表示请求的处理结果。
  - 5xx：服务器异常，服务器在处理请求的过程中发生错误。
  - 4xx：客户端异常，请求包含语法错误或无法完成请求。
  - 3xx：重定向问题，需要进一步操作。

- 2xx：成功，服务器成功接收请求并执行。
- 200：请求成功。

主机信息

- **名称**：主机的名称，通常是由用户指定的一个字符串，用于标识主机的身份。
- **架构**：指主机的硬件架构，如 x86、x86_64、ARM 等。
- **操作系统平台**：指主机所使用的操作系统的平台，如 Windows、Linux、macOS 等。
- **操作系统名称**：指主机所使用的具体操作系统的名称和版本号，如 Windows 10、Ubuntu 20.04 LTS、macOS Big Sur 等。

Process（进程）

是指正在运行的一个程序的实例。

- Pid（Process ID）：进程的唯一标识符，是一个由操作系统分配的整数，用于标识系统中的不同进程，每个进程都有一个不同的 PID。
- Command-Line（命令行）：是指启动进程时指定的命令行参数，包括程序的路径、参数等信息。它是一个字符串，可以通过操作系统提供的接口获取到。

Thread（线程）

是指进程中的一个执行单元。

- ID（Thread ID）：线程的唯一标识符，是一个由操作系统分配的整数，用于标识系统中的不同线程。每个线程都有一个不同的 ID。
- Name（线程名）：是指用户指定的线程名称，用于标识线程的身份。线程名称可以方便用户在程序中进行调试和监控。

DB（数据库）

- Connection-String（连接字符串）：是指连接到数据库所需要的信息，包括主机名、端口号、用户名、密码等。它是一个字符串，用于在程序中建立到数据库的连接。
- Operation（操作）：是指对数据库执行的操作，包括查询、插入、更新、删除等。它是一个字符串，用于表示当前执行的数据库操作。
- Instance（实例）：是指数据库的实例名，用于标识不同的数据库实例。它是一个字符串，通常是在建立数据库连接时指定的。

- **Type（类型）**：是指数据库的类型，包括关系型数据库、非关系型数据库等。它是一个字符串，用于表示当前使用的数据库类型。
- **Statement（语句）**：是指在数据库中执行的 SQL 语句，包括查询语句、插入语句、更新语句、删除语句等。它是一个字符串，用于表示当前执行的 SQL 语句。
- **User（用户）**：是指对数据库进行操作的用户，包括数据库管理员、应用程序用户等。它是一个字符串，用于表示当前执行操作的用户。

Messaging（消息传递）

是指在分布式系统中，通过消息传递实现不同组件之间的通信。

- **System（消息系统）**：是指消息传递所使用的消息系统，如 Apache Kafka、RabbitMQ、ActiveMQ 等。它是一个软件系统，用于实现异步消息传递。
- **Operation（操作）**：是指对消息执行的操作，包括发送、接收、确认等。它是一个字符串，用于表示当前执行的操作。
- **Destination-Kind（目标类型）**：是指消息的目标类型，包括队列（Queue）和主题（Topic）两种。队列用于点对点的消息传递，主题用于发布-订阅模式的消息传递。它是一个字符串，用于表示当前消息的目的地类型。

目标服务（Target Service）

是指客户端需要访问的远程服务。

- **名称（Name）**：是指目标服务的名称，用于唯一标识服务。它是一个字符串，通常由服务提供方指定，并在服务注册中心中注册。
- **类型（Type）**：是指目标服务的类型，用于表示服务的功能类别。例如，Web 服务、消息队列服务、数据库服务等。它是一个字符串，通常由服务提供方指定。
- **实例（Instance）**：是指目标服务的实例，用于标识不同的服务实例。在分布式系统中，通常会有多个服务实例提供相同的服务。它是一个字符串，通常由服务注册中心分配。

4.4、接入点信息

根据不同的客户端采集工具，提供不同资源池（地域）的接入点信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。

2. 在左侧导航栏中选择「**链路配置**」-「**接入点信息**」。

功能说明

Jeager

当通过 Jeager 采集接入应用时，需要提供对应的接入点信息，展示如下。

- **地域**：接入应用时选择哪个地域（资源池），就选择对应地域（资源池）的接入点信息。
- **数据上报**：显示数据上报方式。
- **相关信息**：接入点明细，包括
    - 通过 HTTP 上报的 vpc 接入点、用户 token。
    - 通过 Agent 上报的 vpc 接入点。

OpenTelemetry

当通过 OpenTelemetry 采集接入应用时，同样需要提供对应的接入点信息。

- **地域**：接入应用时选择哪个地域（资源池），就选择对应地域（资源池）的接入点信息。
- **数据上报**：显示数据上报方式。
- **相关信息**：接入点明细，包括
  - 通过 HTTP 上报的 vpc 接入点。
  - 通过 gRPC 上报的 vpc 接入点、鉴权 token。

SkyWalking

当通过 SkyWalking 采集接入应用时，也需要提供对应的接入点信息。

- **地域**：接入应用时选择哪个地域（资源池），就选择对应地域（资源池）的接入点信息。
- **数据上报**：显示数据上报方式。
- **相关信息**：接入点明细，包括
  - SkyWalking 8.*的方式，包括接入点链接和 token。

4.5、用量统计

展示当前租户下的 span 情况，包括 Span 上报数、Span 存储量。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**链路配置**」-「**用量统计**」。

功能说明

关键指标

- **Span 上报数**：筛选时间段内，上报的 Span 总数。
- **Span 存储量**：筛选时间段内，存储过的 Span 总数。

用量趋势

支持切换不同的时间间隔查看 Span 上报数和 Span 存储量的变化趋势。

4.6、告警管理

4.6.1、设置告警规则

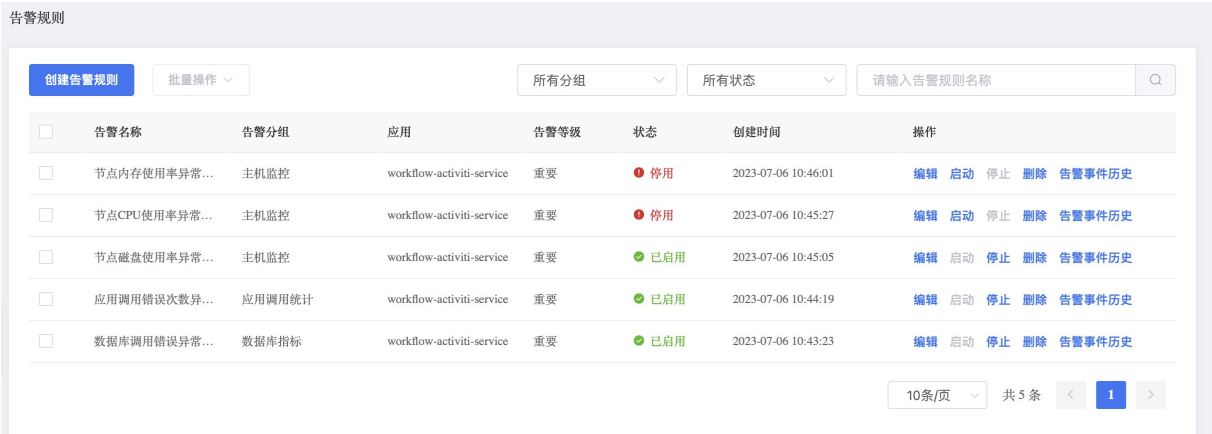
展示当前租户下所有告警规则信息，支持增删改查、起停和查看告警事件历史。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。

2. 在左侧导航栏中选择「**告警管理**」-「**告警规则**」。

功能说明



展示当前租户下所有告警规则信息，支持基础的增删改查、起停、查看告警事件历史操作。

创建/编辑告警规则

- **告警名称**：您可自定义告警名称。
- **应用**：展示应用列表中，所有已接入应用，支持多选。
  - 新增应用自动在此告警规则中追加：开启后，新增应用则自动使用该告警规则。

- **告警详情**：包含告警分组、告警指标、告警条件、筛选条件、告警内容。

告警分组	告警指标	告警条件
应用调用统计	调用错误次数	某个时间段内，该指标大于、大于等于、小于、小于等于、等于某个数值/百分比时生效
	调用错误率	
	调用次数	
	调用平均响应时间	
数据库指标	数据库调用次数	某个时间段内，该指标大于、大于等于、小于、小于等于、等于某个数值/百分比时生效
	数据库调用响应时间	
	数据库调用错误次数	

- **持续时间**：支持设置延迟告警。
- **告警等级**：一般、次要、重要、等级。
- **通知策略**：可以选择通知策略菜单里设置的策略。
- **标签**：自定义标签，用于筛选。

启动/停止

对于暂时不用的告警策略，您可以操作停止。

告警事件历史

点击跳转告警事件历史菜单，显示该告警规则触发的实际告警记录。

4.6.2、查看告警发送历史

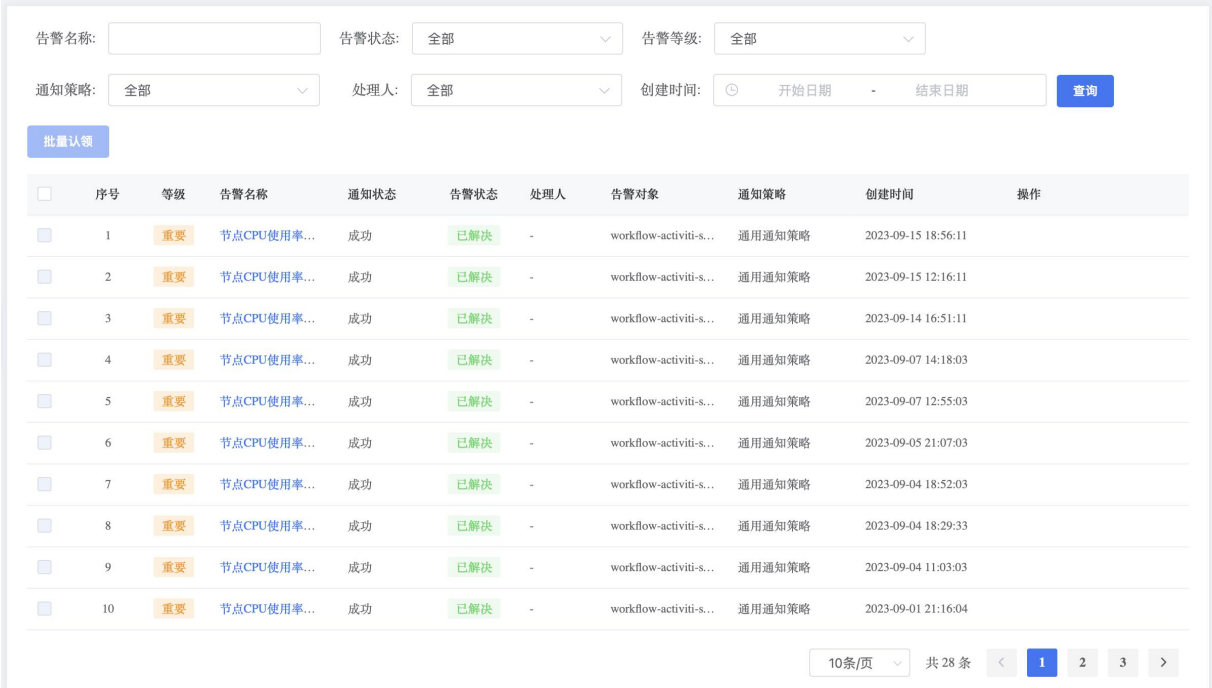
展示当前租户下所有发送的告警历史，支持筛选和对告警信息进行操作。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。

2. 在左侧导航栏中选择「**告警管理**」-「**告警发送历史**」。

功能说明



展示当前租户下所有告警发送信息。

筛选

- **告警名称**：显示告警规则名称。
- **告警状态**：显示事件当前状态是待认领、已解决、处理中。
- **告警等级**：显示告警的重要层级分布是一般、次要、重要、紧急。
- **通知策略**：告警对应的通知策略。
- **处理人**：告警的最新解决/认领人。
- **创建时间**：告警产生的时间。

操作

- **认领**：当告警处于待认领状态时，可主动领取当前告警。
- **解决**：当告警处于待认领/处理中状态时，点击解决可更新告警状态为已解决。
- **指定处理人**：指定他人处理该告警。
- **告警发送详情**
  - **详情**：显示告警发送的基础信息如告警对象、处理人、解决方案。

  - **事件**：显示触发告警的事件名称、状态和触发时间，点击可查看详情。

  - **活动**：显示包括认领、取消认领、指派处理人、告警关闭等在内的各种活动信息，支持筛选。

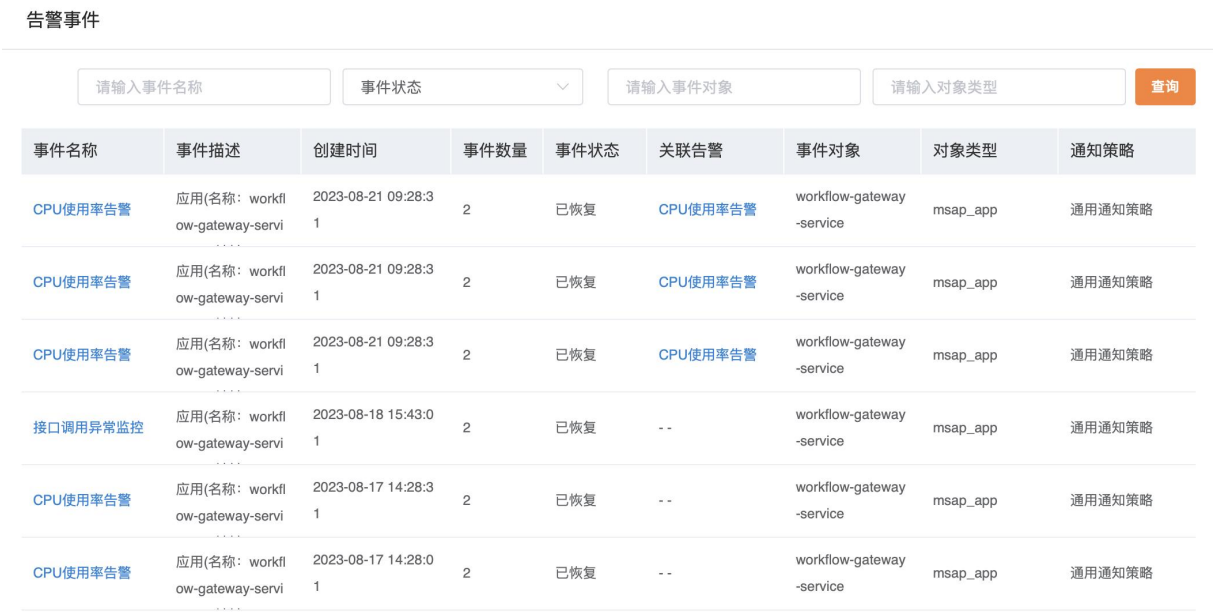
4.6.3、查看告警事件历史

展示当前租户下所有告警事件历史，支持筛选及查看详情。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**告警管理**」-「**告警事件历史**」。

功能说明

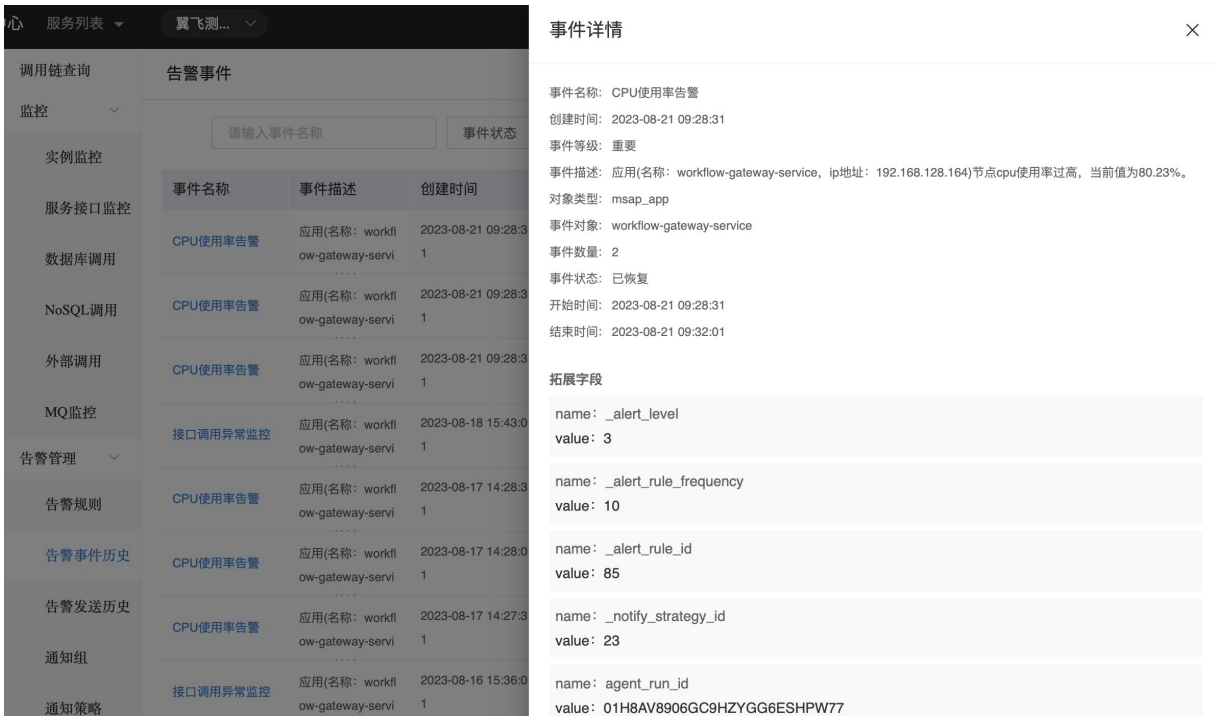


展示当前租户下所有告警事件信息。

筛选

- **事件名称**：显示告警规则名称。
- **事件状态**：显示事件当前状态是告警中、已恢复、静默。
- **事件对象**：监控对象，比如应用名称、集群名称等。
- **对象类型**：告警事件对象的类型。

支持查看事件详情



支持查看告警详情

- **详情**：显示告警发送的基础信息如告警对象、处理人、解决方案。

- **事件**：显示触发告警的事件名称、状态和触发时间，点击可查看详情。

- **活动**：显示包括认领、取消认领、指派处理人、告警关闭等在内的各种活动信息，支持筛选。

4.6.4、通知对象

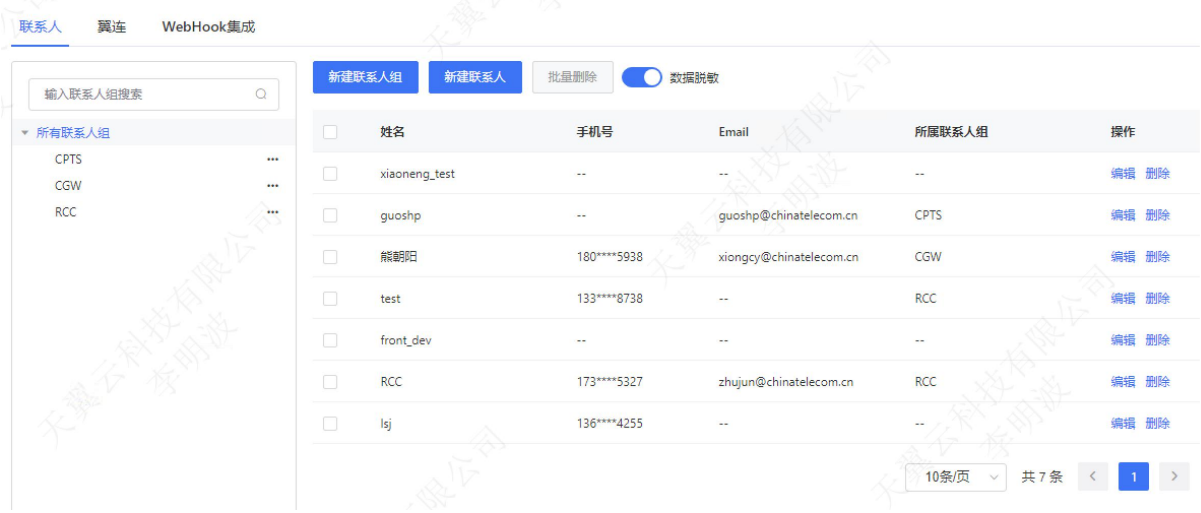
通知对象

告警支持通过短信、邮箱、翼连等方式将告警信息通知给对应接收人，我们可以在通知组中设置接收人信息。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**告警管理**」-「**通知组**」。

功能说明



联系人（短信/邮箱）

如需要通过短信/邮箱进行告警，可在此处维护联系人信息。支持对联系人信息进行增删改查，需要录入基础信息。

联系人组管理

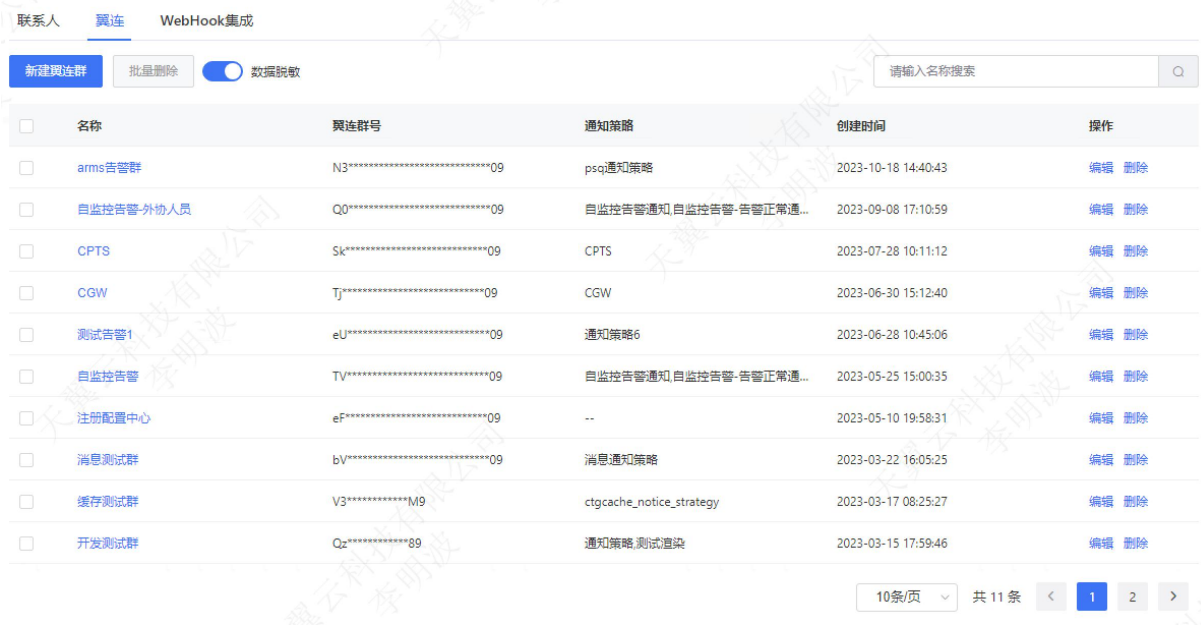
- **新增联系人组**：新增联系组，在左侧联系人组顶端新增一条。
- **编辑**：编辑联系人组，支持修改组名，支持移入/移出组员。
- **删除**：删除联系人组（不会删除组内联系人）。
- **搜索**：通过联系人组名称进行搜索。

联系人管理

- **新增联系人**：新增联系人，添加完成后，联系人列表顶端新增一条。
- **编辑**：支持修改姓名、手机号码、邮箱和联系人组。
- **删除**：删除当前联系人。
- **批量删除**：删除多位联系人。
- **数据脱敏**：对手机号进行数据脱敏。

翼连

翼连是天翼云生态中的主流 IM 工具，此处支持增删改查翼连群信息，将某个翼连群作为一个“联系人组”。



- **新增翼连群**：填写名称和群号完成添加。
- **编辑**：支持修改名称和群号。
- **删除**：支持删除当前翼连群。
- **批量删除**：删除多个翼连群。
- **数据脱敏**：对翼连群号进行数据脱敏。
- **搜索**：通过名称进行模糊搜索。

WebHook 集成

支持以 WebHook 的方式对第三方通知对象（钉钉、企业微信、飞书等）发送告警信息。

支持增删改查 WebHook 信息。

- **新增 WebHook**：填写名称、webhook 调用地址，设置渲染方式完成添加。

- **编辑**：支持修改名称、webhook 调用地址和渲染方式。
- **删除**：支持删除当前 webhook。
- **批量删除**：删除多个 webhook。
- **数据脱敏**：对地址进行数据脱敏。
- **搜索**：通过名称进行模糊搜索。

4.6.5、通知策略

创建告警通知策略，当告警触发时，只要不符合静默策略，就会依照通知策略在对应渠道发送告警信息给对应的通知对象。

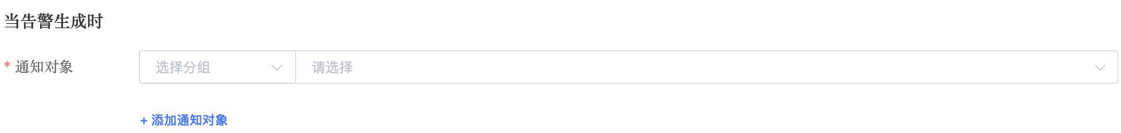
功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**告警管理**」-「**通知策略**」。

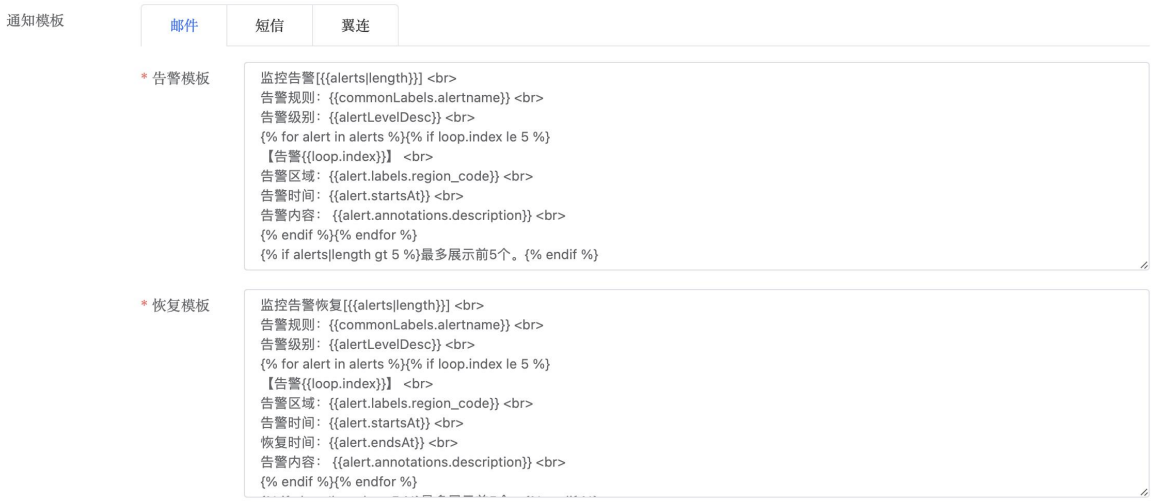
功能说明

支持增删改查告警通知策略信息。

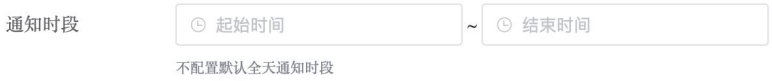
- **通知对象**：可从通知组进行选择。



- **通知模板**：可跟进不同通知渠道（邮件、短信、翼连）设置不同的通知模板。



- **通知时段**：可以设置通知时段，默认是告警触发时通知。



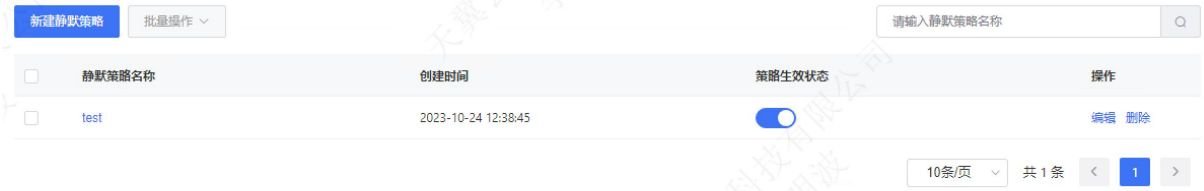
4.6.6、静默策略

如果满足静默策略则不会触发告警通知，您可以通过静默策略对告警进行收敛，避免同一时间出现大量重复告警或关联告警。

功能入口

1. 选择目标资源池，并登录 TAS 组件控制台。
2. 在左侧导航栏中选择「**告警管理**」-「**静默策略**」。

功能说明

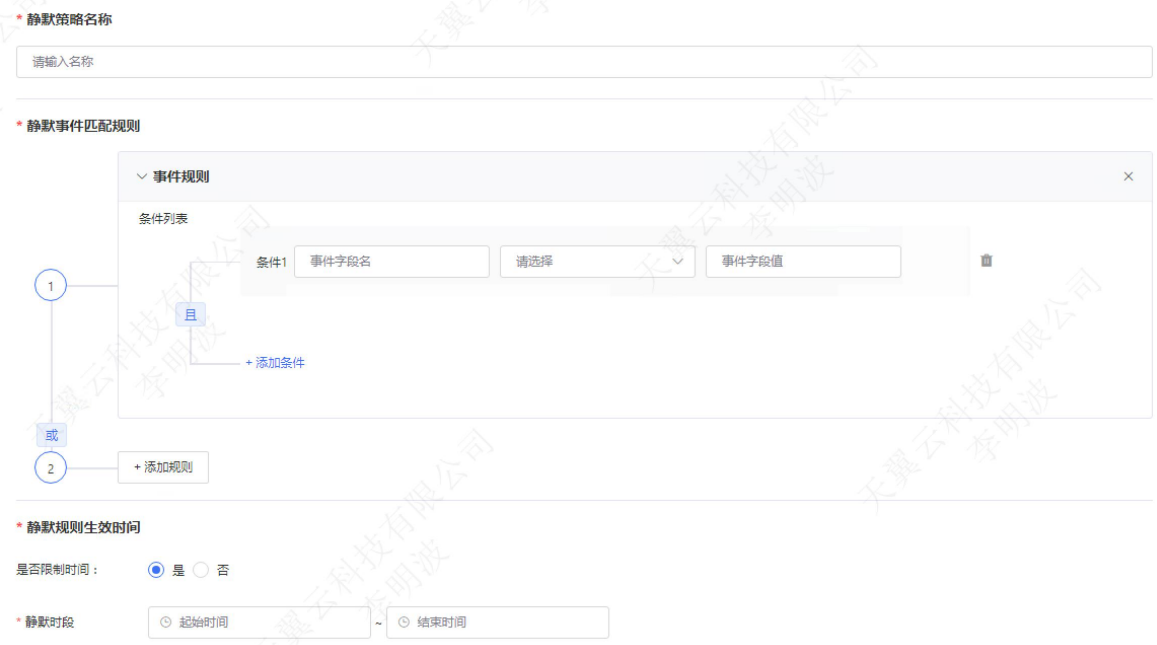


搜索

- **静默策略名称**：模糊搜索。

操作

- **新建静默策略**：新建一条静默策略。



  - **静默策略名称**：自定有名称，用于辨识策略。
  - **静默事件匹配规则**：支持通过且或关系组合创建事件规则，使得在满足当前事件规则时，触发静默策略。
  - **静默策略生效时间**：支持设置静默规则的生效时间段。

- **编辑**：支持修改静默策略，内容与新建一致。

- **删除**：删除当前静默策略。
- **策略生效状态开关**：控制当前策略是否生效。

批量操作

支持批量设置策略生效/失效，支持批量删除策略。

五、 最佳实践

5.1 、使用 TAS 进行慢请求分析

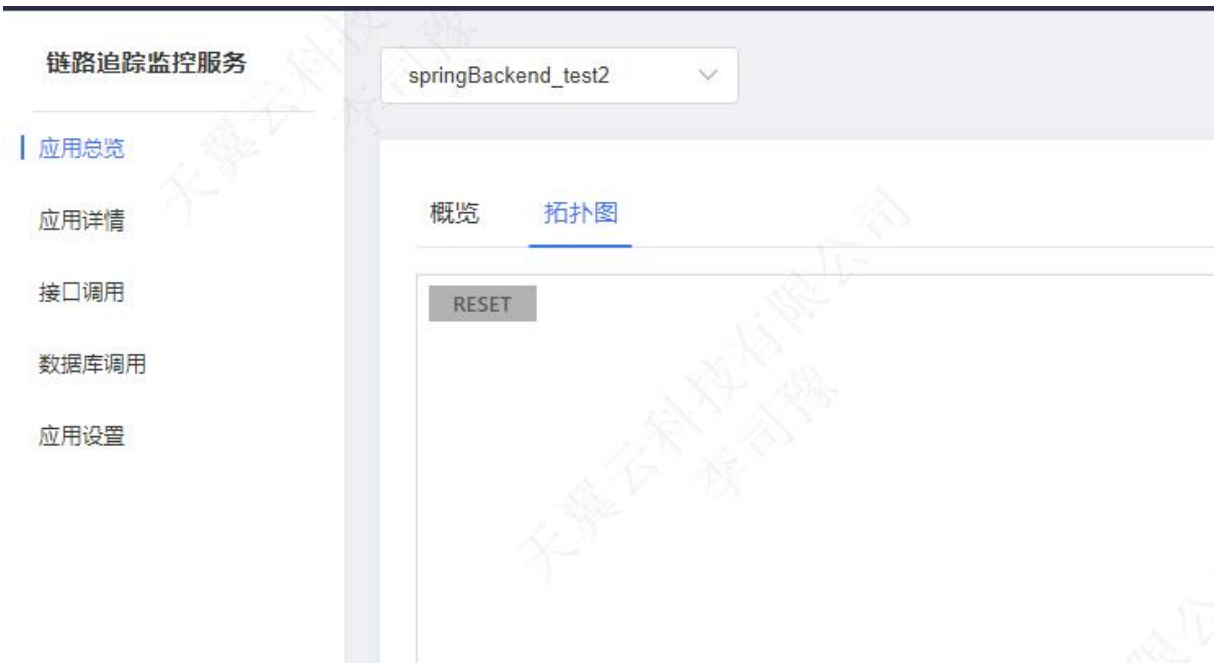
Java 应用通过 Java 字节码注入技术可以实现自动埋点，并以此知道慢请求是出现在哪两个埋点间，但如果要精确定位导致慢请求出现的代码方法，您可以使链路追踪 TAS 进行慢请求分析。

前提条件

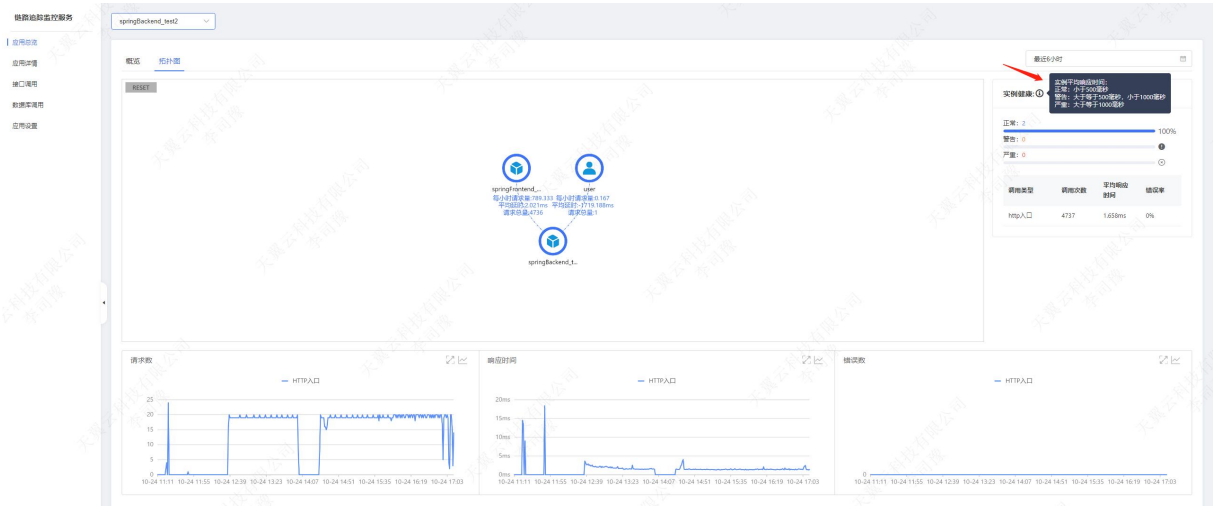
您已接入链路追踪。

慢请求分析

1. 登录链路追踪控制台，在左侧导航栏单击「**应用列表**」。
2. 在「**应用列表**」页面单击目标应用名称，打开应用详情。
3. 在左侧导航栏单击「**应用总览**」，并在右侧导航栏单击「**拓扑图**」。



4. 在页面右侧实例健康可以查看该实例的**健康情况以及调用情况**，实例健康情况能够反映是否有慢请求。将鼠标移至页面右侧实例健康旁的感叹号，即可查看实例健康的说明，实例平均响应时间小于 500 毫秒为正常，大于等于 500 毫秒小于 1000 毫秒为警告，大于等于 1000 毫秒为严重。



5.2、使用 TAS 进行慢请求分析

Java 应用通过 Java 字节码注入技术可以实现自动埋点，并以此知道慢请求是出现在哪两

个埋点间，但如果要精确定位导致慢请求出现的代码方法，您可以使链路追踪 TAS 进行慢请求分析。

前提条件

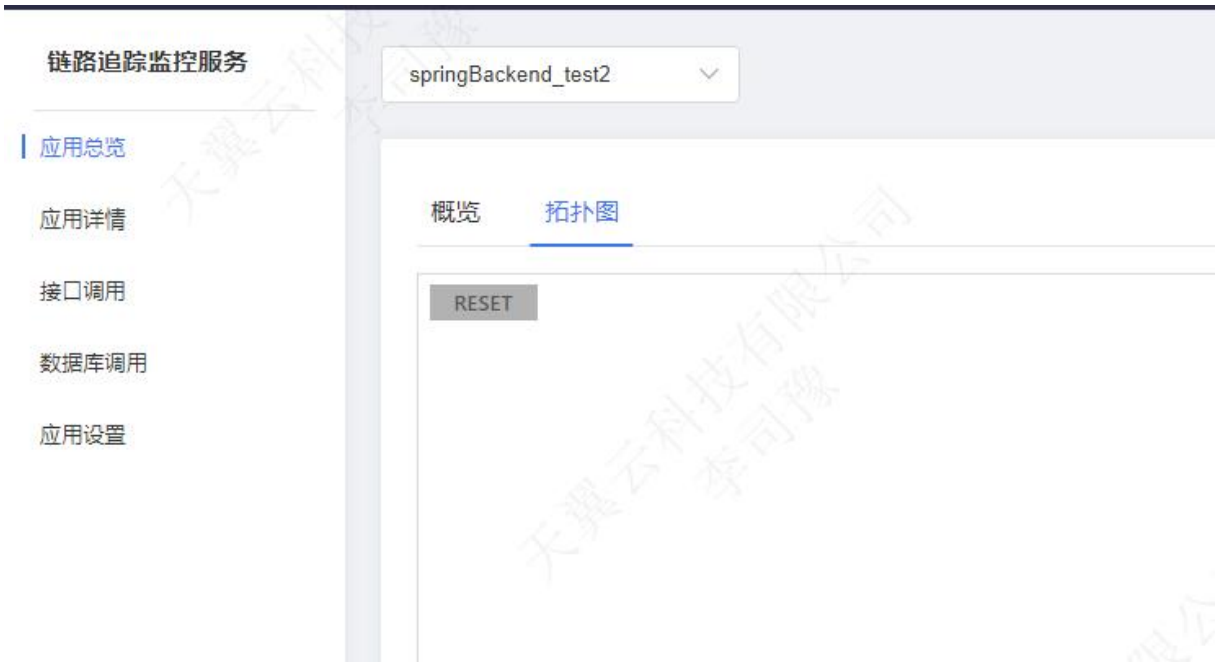
您已接入链路追踪。如未接入，请根据您应用的实际情况选择接入方式完成接入，详情可参考快速入门。

慢请求分析

1. 登录链路追踪控制台，在左侧导航栏单击「**应用列表**」。

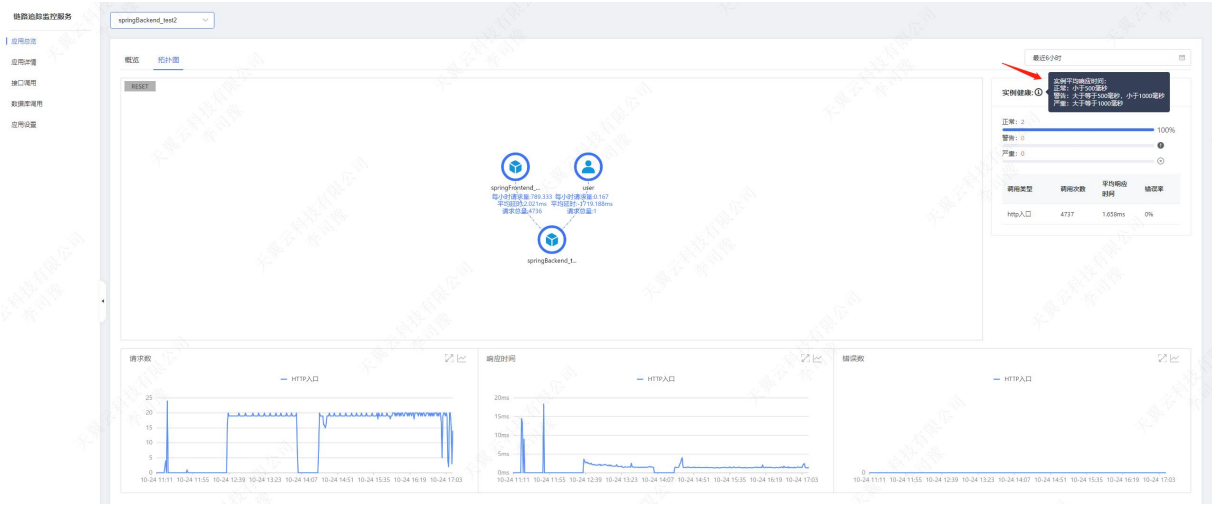
2. 在「**应用列表**」页面单击目标应用名称，打开应用详情。

3. 在左侧导航栏单击「**应用总览**」，并在右侧导航栏单击「**拓扑图**」。



4. 在页面右侧实例健康可以查看该实例的**健康情况以及调用情况**，实例健康情况能够反映是否有慢请求。将鼠标移至页面右侧实例健康旁的感叹号，即可查看实例健康的说明，实例平均响应时间小于 500 毫秒为正常，大于等于 500 毫秒小于 1000 毫秒为警告，大于等于

1000 毫秒为严重。



5.3、通过 go otel sdk 实现多级调用效果

otel 官方给的 demo 里面并没有给出多个应用之间的调用链路。只提供了简单的 a 到 b 的调用，显然这对于我们实际生产来说是远远不够的。

在生产环境，往往调用的链路都是多级的，下面我们将给大家提供一个最基础的多级调用。其中一个细节是 traceId 的透传，如果不透传，下游是服务拿到 traceId，也就无法把整个链路串起来。具体看代码：

clientA

```
package main


import (
    "context"
    "flag"
    "go.opentelemetry.io/contrib/instrumentation/net/http/httptrac
e/otelhttptrace"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp
"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/baggage"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptraceht
tp"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"
    "go.opentelemetry.io/otel/trace"
    "io"
    "log"
    "net/http"
    "net/http/httptrace"
    "time"
)
```

serverB

```
package main


import (
    "context"
    "flag"
    "fmt"
    "io"
    "log"
    "net/http"
    "net/http/httptrace"
    "time"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp
"
    "go.opentelemetry.io/contrib/instrumentation/net/http/httptrac
e/otelhttptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptraceht
tp"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"


    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/baggage"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/resource"
```

serverC

```
package main

import (
    "context"
    "flag"
    "fmt"
    "io"
    "log"
    "net/http"
    "net/http/httptrace"
    "time"

    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp
"
    "go.opentelemetry.io/contrib/instrumentation/net/http/httptrac
e/otelhttptrace"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptraceht
tp"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/baggage"
```

通过以上代码，您就可以实现 clientA->serverB->serverC 的调用了。

六、 常见问题

6.1、计费类

如何停止应用计费

如果您不再需要监控该应用以及看历史监控数据，可在应用列表中直接删除应用并停止上报，继续保留应用就算不产生上报也会收取 span 和指标的存储费。

公测期间计费

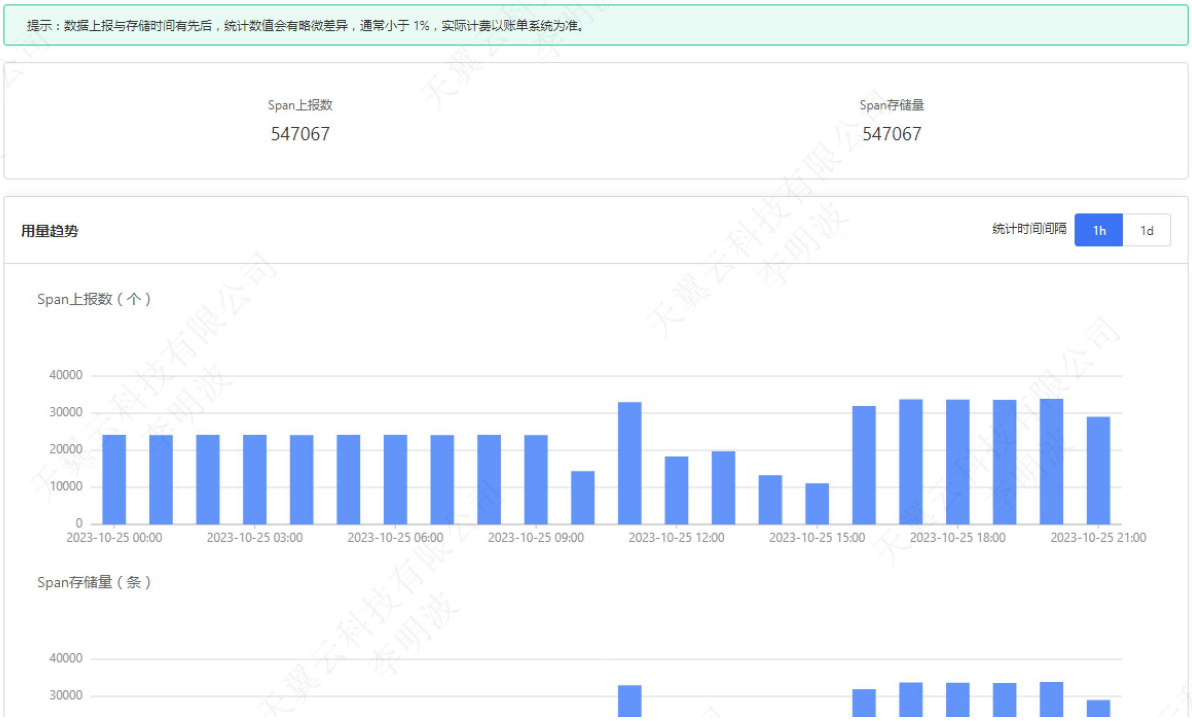
公测期间无论您如何使用都不会产生任何费用，如果公测到期后您选择继续使用，将会按照您选择的使用方式计费。

如何查看用量

链路追踪控制台为用户提供查看资源消耗的功能，您可以通过设置过滤条件查看单个或全部应用在特定时间内的资源消耗。

1. 在 TAS 控制台左侧导航栏中选择集群配置-用量统计。

2. 在用量统计页面右上角设置需要查看的时间段。



- **Span 上报数**：筛选时间段内，上报的 Span 总数。
- **Span 存储量**：筛选时间段内，存储过的 Span 总数。比如说昨天开始接入应用，昨天上报了 100span，今天上报了 100span，筛选今天存储的 span 总数就是 200。

6.2、操作类

如何检查网络联通性？

网络不通会导致数据传输异常，无法正常查看控制台数据。

可使用 telnet 命令测试目标主机与上报地址网络是否连通。例如，以测试内蒙 6 地域的连通性为例，请登录应用所部署的机器，并输入以下命令：

```
telnet arms-data-proxy.cq2b.inner.ctyun.cn 8417
```

具体每个资源池的服务器地址参考接入点信息。

为什么 go 应用 trace 链路串不起来？

使用 otel sdk 上报方式

在使用 otel sdk 上报 go 应用链路数据的时候。明明是存在 a 调用 b 再调用 c 的情况，但是在上报的数据里面只能看到 a->b 和 b->c 这两条独立的链路。

出现这种情况是因为构造 httpclient 的时候缺少了如下 WithClientTrace：详细 client 构造方式如下：

```
client := http.Client{
       Transport: otelhttp.NewTransport(
          http.DefaultTransport,
         // 必须加上这个配置才会透传 trace
          otelhttp.WithClientTrace(func(ctx       context.Context)
*httptrace.ClientTrace {
              return otelhttptrace.NewClientTrace(ctx)
          }),
       ),
   }
```

使用 jaeger sdk 上报方式

使用 jaeger sdk 方式上报 go 应用链路数据的时候。明明是存在 a 调用 b 再调用 c 的情况，但是在上报的数据里面只能看到 a->b 和 b->c 这两条独立的链路。

出现这种情况是因为构造 httpclient 的时候缺少了如下 children span 的构建和 traceId 的透传，具体方式如下：

```
//获取头信息
spanCtx,      _      :=      tracer.Extract(opentracing.HTTPHeaders,
opentracing.HTTPHeadersCarrier(r.Header))
//设置子 span
span = tracer.StartSpan(spanName, opentracing.ChildOf(spanCtx))
//透传 traceId
tracer.Inject(span.Context(),              opentracing.HTTPHeaders,
opentracing.HTTPHeadersCarrier(req.Header))
```

为什么应用详情里面不显示应用 ip

以 go 应用为例子，采用 otel sdk 上报的时候应用详情这里显示的是一串看不懂的字符串，

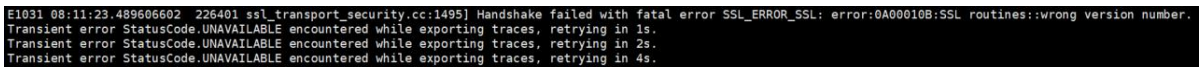
这个其实是主机名，要想显示成 ip 需要做如下设置：

```
res, _ := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // the service name used to display traces in backends
            semconv.ServiceNameKey.String("otel-go-client-demo"),
            semconv.HostNameKey.String(""),
          // 需要在这里把 ip 地址加上
            semconv.NetSockHostAddrKey.String("ip")
        ),
    )
```

python 接入常见问题

场景

使用 Python 语言应用接入链路追踪服务后，客户端运行正常，但是上报数据时，报 SSL_ERROR_SSL 错误，导致 Trace 数据上报失败。



解决方案

应用代码检查

1. 检查 Python 工程接入逻辑代码，关注【OTLPSpanExporter】对象的创建代码。

```
OTLPSpanExporter(                                endpoint="<endpoint>",
headers="x-ctg-authorization=<token>" )
```

2. 构建 【OTLPSpanExporter】 对象添加如下参数。

```
OTLPSpanExporter(                                endpoint="<endpoint>",
headers="x-ctg-authorization=<token>", insecure=True )
```

3. 重启 Python 应用。