

应用性能监控 (APM)

产品文档



腾讯云TCE

目录

应用性能监控 (APM)	3
产品白皮书	3
产品概述	3
产品优势	4
应用场景	5
产品架构	7
功能特性	8
使用建议	10
产品术语	13
快速入门	14
快速入门	14
接入指南	15
接入概述	15
接入Java应用	16
通过增强版 OpenTelemetry Java 探针接入 (推荐)	16
通过 SkyWalking 协议上报	21
接入 Go 应用	25
通过 OpenTelemetry-Go 接入 Go 应用 (推荐)	25
通过 SkyWalking-Go 接入 Go 应用	33
接入 Node.js 应用	36
通过 OpenTelemetry-JS 方案接入 Node.js 应用	36
接入 Python 应用	40
通过 OpenTelemetry-Python 接入 Python 应用	40
接入 PHP 应用	44
通过 OpenTelemetry-PHP 接入 PHP 应用 (推荐)	44
接入 .Net 应用	56
通过 OpenTelemetry-dotnet 接入 .NET 应用 (推荐)	56
接入其他语言编写的应用	63
操作指南	65
资源管理	65
新建业务系统	65
应用监控	66
应用列表	66
应用详情	68
接口监控	70
错误监控	71
数据库概览	72
数据库分析	74
调用查询	75
调用查询	75
访问管理	78
概述	78
策略语法	79
策略授予	83
告警服务	84
新建告警	84
查看告警	85
系统配置	86
业务系统配置	86
应用配置	87
参考信息	89
OpenTelemetry 方案支持的组件和框架	89
运维管理指南	91
产品架构	91
日常巡检	92
故障处理	95
监控介绍	99
应急预案	101
最佳实践	102
节点重启	106
备份恢复	108
参考信息	110

产品白皮书

产品概述

应用性能监控 (Application Performance Monitoring , APM) 是一款应用性能管理平台，基于实时的多语言应用探针全量采集技术，为您提供分布式应用性能分析和故障自检能力，全方位保障系统的可用性和稳定性。协助您在复杂的业务系统快速定位性能问题，降低 MTTR (平均故障恢复时间)。实时了解并追踪应用性能情况，提升用户体验。应用性能监控支持多种主流编程语言以及开源框架，为您提供应用性能监控一站式解决方案。

产品优势

开箱即用

应用性能监控提供完整的 APM 服务端实现，包括数据采集、流式处理、数据存储、可视化呈现、告警管理等方面。用户可以使用无侵入式探针，在不触动业务逻辑和代码逻辑的前提下，快速接入应用，获得开箱即用的应用可观测能力。

多语言支持

应用性能监控支持 Java、Go、Python、Node、PHP 等主流开发语言。

拥抱开源生态

应用性能监控全面支持业界通用的 OpenTelemetry 标准，同时兼容 Skywalking、Jaeger 等链路协议，支持从 Skywalking、Jaeger 等方案的平滑迁移。通过简单修改上报配置，您即可将链路追踪系统从开源产品迁移到 APM 上，免除运维烦恼，减少开发成本。

功能全面

应用性能监控提供应用拓扑、多维调用链分析、接口分析、SQL 分析、多维下钻分析、高阶诊断、告警管理、自定义大盘等重要能力，全面覆盖应用性能管理领域的各类需求，能够帮助您快速定位线上性能问题，针对性的对应用进行性能优化，确保业务系统的稳定运行。

稳定性保障

应用性能监控基于云服务商在微服务领域的经验，并结合云服务商强大的基础设施以及云产品生态，为您提供可靠的可观测能力。您无需考虑分布式扩展、流量控制、容量规划等复杂的稳定性问题，即可获得稳定流畅的使用体验。

低成本

应用性能监控基于云存储、链路压缩技术以及采样技术，资源成本比开源自建低30%以上。

应用场景

定位性能瓶颈

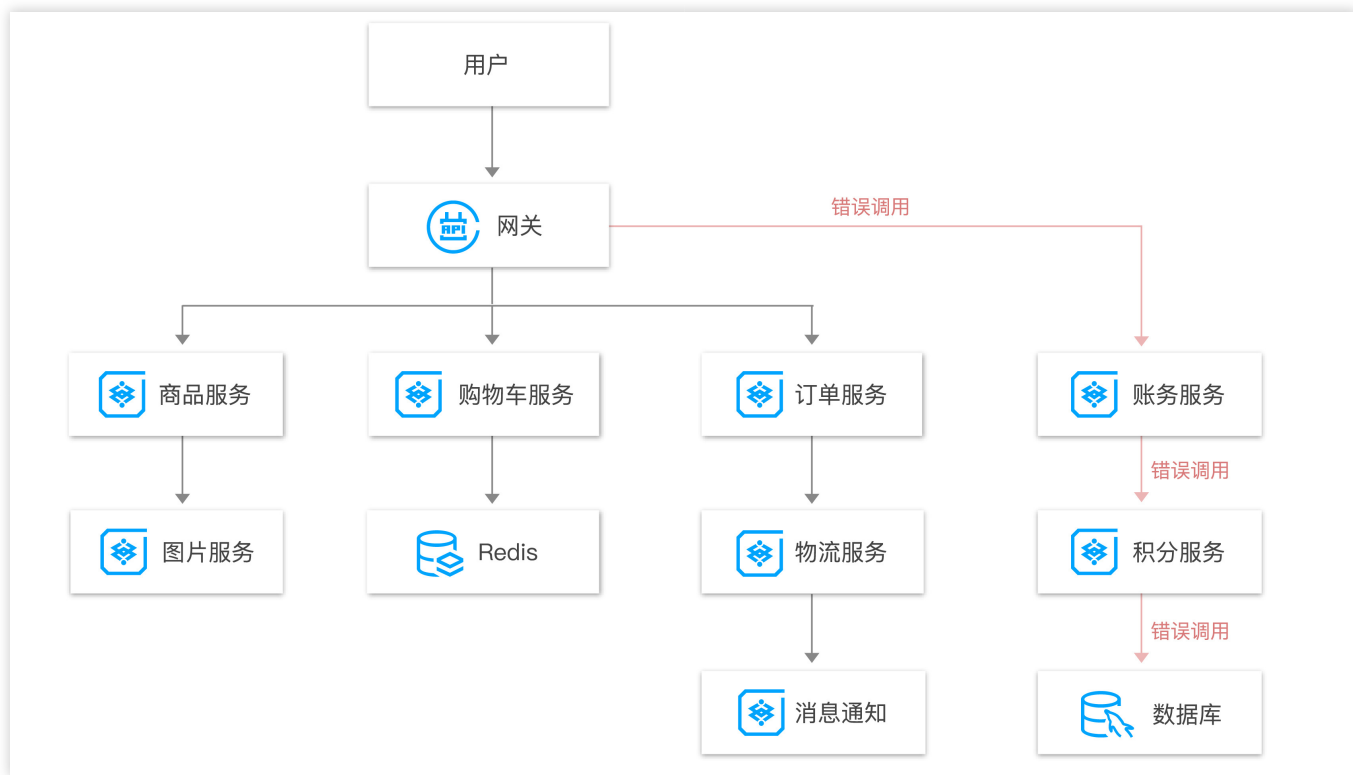
痛点

随着业务不断发展，业务逻辑和服务调用日益复杂。导致应用性能问题分析与定位日益艰难，给监控运维带来了巨大的挑战：

- 应用之间的依赖关系复杂，难以梳理。
- 调用链路长，排查和定位群体困难。
- 接口调用、数据库调用关系复杂，管理难度大。

解决方案

- 基于应用拓扑自助发现，定位性能瓶颈。
- 基于关键性能指标对比，优化应用性能。
- 根据指标变化趋势配置告警，及时了解异常。



应用性能优化，提升用户体验

痛点

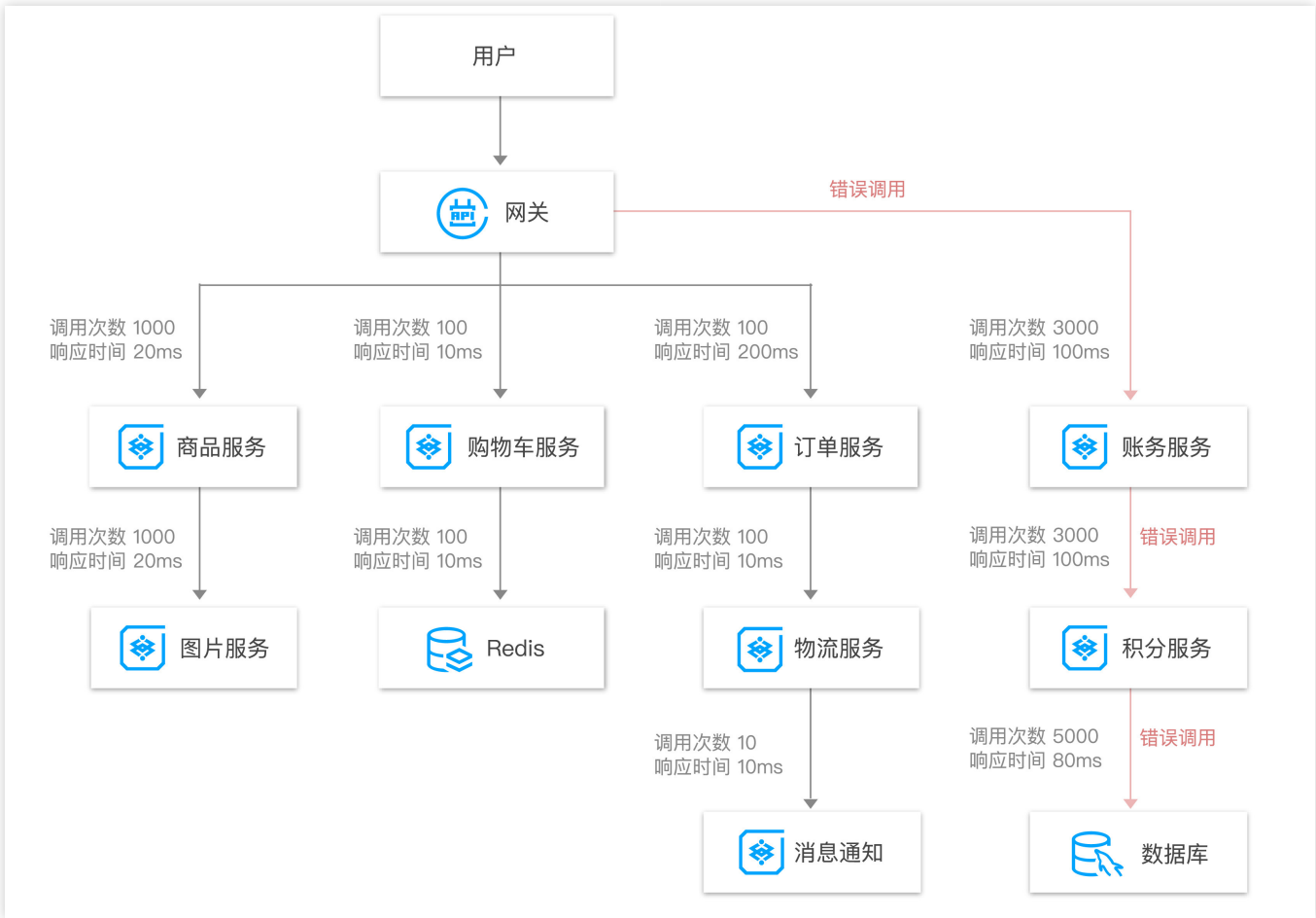
当业务量突增，各应用之间调用时间长、接口加载缓慢等问题，导致用户体验感降低。由于各应用之间的调用情况复杂，分

析应用性能问题难以实现。

解决方案

在依赖拓扑图中能对各个应用的调用次数以及耗时进行分析，找到负载较高以及负载较少的服务，对资源进行合理分配。

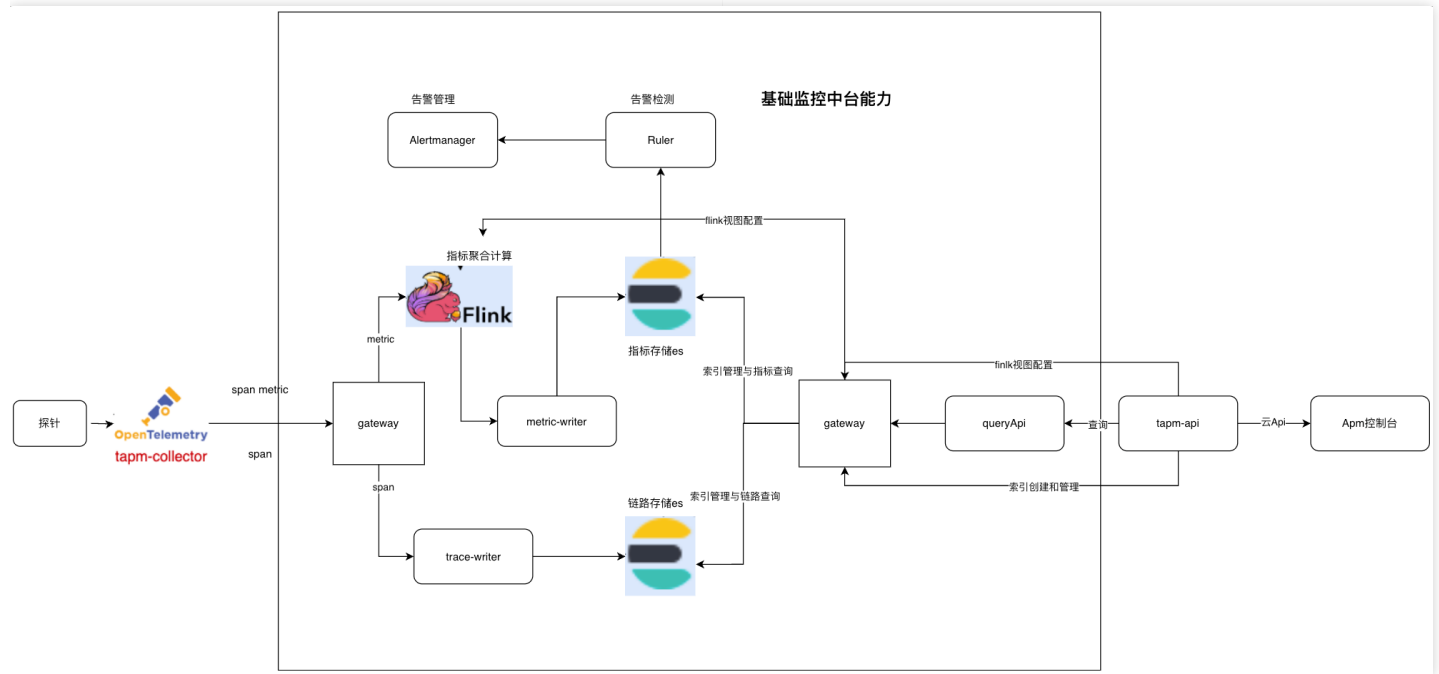
- 调用链路聚合汇总：对所有的调用信息进行聚合汇总，对各个应用的调用情况以及响应情况进行分析。
- 关键路径：快速发现整个系统调用拓扑中关键服务路径与接口路径。
- 优化不合理调用：及时发现某些不合理的调用，如频繁进行数据库操作、循环依赖等。



产品架构

应用性能监控 (APM) 集成基础监控中台能力，为用户提供应用性能监控一站式解决方案。

应用性能监控 (APM) 的详细架构设计如下图：



APM 的核心组件：

- tapm-collector：统一上报接入点，适配多种开源协议，接收用户上报数据，进行数据处理，生成链路和指标数据。
- tapm-api：控制台服务，负责对链路和指标数据进行可视化处理。

基础监控中台能力：

- gateway：统一网关，接收数据。
- flink：流计算，聚合指标，生成不同指标视图。
- trace-writer：链路写入组件，屏蔽底层存储。
- metric-writer：指标写入组件，屏蔽底层存储。
- ruler：告警检测。
- alertmanager：告警管理。
- query-api：查询组件，屏蔽底层存储。

功能特性

应用依赖拓扑自动发现

应用性能监控依托分布式调用链追踪的模型，自动发现应用逻辑拓扑，以应用为基本单元，绘制全局拓扑关系。可视化的展示繁杂应用间依赖关系，实时数据钻取，智能应用状态分析迅速定位影响业务的关键（瓶颈应用或组件）。同时，应用维度的上下游依赖关系，清晰展示上游负载、下游影响，结合上下游环境，全面分析应用健康状况和性能指标。

多维链路查询

应用性能监控支持按照应用名、接口名，响应时间，采样时间，是否包含错误异常、是否耗时过长等维度对调用链路进行过滤筛选。可以进一步帮助用户定位到包含特定异常信息，数据库查询的链路，帮助用户从海量链路数据中聚焦到重点链路，快速定位异常链路，完成故障排查。

一站式调用链分析

应用性能监控在微服务架构下，链路追踪可以跨越服务，帮助用户自动构建每次请求的完整路径。同时收集请求参数、事务数据、错误异常到方法堆栈和底层实例环境信息，实现一站式全链路问题分析，提高定位问题的效率。解决了日志分散，格式不规范，不易聚焦，上下游服务日志难以关联等故障排查痛点。

接口分析

应用性能监控在覆盖应用监控三大黄金指标基础上，增加 Apdex 指标科学评估用户满意度。继承云服务商可观测平台丰富可视化报表经验沉淀，支持用户灵活切换环比标尺线，准确判断应用动态和变化趋势。同时智能监测 TOP5 耗时和 TOP5 错误率接口，及时主动上浮问题，加速用户聚焦过程，实现应用性能的精准监控。

数据库调用监控

对于常用的关系型数据库（例如 MySQL、PostgreSQL）以及 NoSQL 数据库（例如 Redis®）调用，应用性能监控提供了慢 SQL 分析能力，并自动采集业务系统相关数据库性能指标，让您实时了解数据库慢 SQL、调用情况、读取情况等，精准定位数据库性能问题。

多维下钻分析

- 应用性能监控提供持续观测各服务、接口、服务实例的调用情况，还能够侦测到对系统内中间件的调用数据。
- 应用性能监控提供基于监控黄金指标（吞吐量、响应时间、错误率）下钻分析能力，助您高效了解系统各维度的运行状态。

应用性能诊断

应用性能监控提供了内存分析、应用性能剖析、资源池分析等高阶性能诊断能力，基于极低的性能开销，在线诊断 CPU 消耗高、内存溢出、连接池问题等应用性能领域的疑难杂症。

告警管理

基于云服务商可观测平台提供的告警管理功能，您可以针对应用性能关键指标设置告警触发规则，通过多种渠道（短信、邮件）进行通知。当监控指标异常时，让用户可以第一时间接收到异常告警通知，及时响应处理故障，避免因异常发现不及时造成的业务损失。

使用建议

软件使用建议

1. 通过 OpenTelemetry SDK 接入 Go 应用

支持 Go 的官方版本为1.21和1.22。

2. 通过 SkyWalking 协议接入 Go 应用

SkyWalking Go 探针对主流的框架和组件实现了自动埋点，详情请参考 [SkyWalking 官方文档](#)。

3. 通过增强版 OpenTelemetry Java 探针接入 Java 应用

支持的 Java 版本

支持 Java 8 及以上的 Java 版本，推荐使用 Java 8、Java 11、Java 17、Java 21等 LTS 版本，关于 LTS 版本请参考 [Java 官方文档](#)。

支持的依赖与框架

增强版 OpenTelemetry Java 探针对常用的开源组件和框架进行了自动埋点，包括 Servlet、Spring Web MVC、Apache Dubbo 等，具体清单请参考 [OpenTelemetry Java 探针支持的依赖与框架列表](#)。

此外，增强版 OpenTelemetry Java 探针还对 XXL-JOB 2.2.0+ 以及 Apache Thrift 0.9.3+ 进行了埋点增强。

支持的应用服务器

增强版 OpenTelemetry Java 探针支持常用的应用服务器，具体清单请参考 [OpenTelemetry Java 探针支持的应用服务器列表](#)。

4. 通过 SkyWalking 协议接入 Java 应用

SkyWalking Java 探针支持的 Java 版本

Java 8 ~ 21

SkyWalking 方案支持的框架和组件

SkyWalking 探针对主流的框架和组件实现了自动埋点，详情请参考 [SkyWalking 官方文档](#)。

5. 通过 OpenTelemetry SDK 接入Python 应用

支持 Python 3.6及以上版本。

6. 通过 SkyWalking 协议接入 Python 应用

SkyWalking Python 探针支持的 Python 版本

Python 3.7+

SkyWalking Python 探针支持的框架和组件

SkyWalking 探针对主流的框架和组件实现了自动埋点，详情请参考 [SkyWalking 官方文档](#)。

7. 通过 OpenTelemetry-JavaScript 接入Node 应用

支持的 Node.js 版本

Node.js v14+

支持的依赖与框架

OpenTelemetry-JavaScript 方案对于 Node.js 系的常用依赖库和框架提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考社区提供的 [完整列表](#)。

8. 通过增强版 OpenTelemetry PHP 探针接入 PHP 应用

安装如下工具：

- [PECL](#)
- [Composer](#)

并确保 Shell 中可以运行以下命令：

```
php -v
composer -v
```

自动接入

PHP 8.0+

目前自动埋点支持的框架列表详情请参见 [OpenTelemetry 官方文档](#)。

手动接入

PHP 7.4+

产品术语

应用性能监控 (APM) 的常见术语请参见下表：

术语	描述
业务系统	业务系统用于分类管理应用，每个业务系统有唯一的 Token，应用接入的时候需要指定 Token。可以在业务系统级别设置存储时长、计费方式等参数，也可以基于业务系统实现权限管理和分账。不同业务系统之间的监控数据完全隔离。
应用	应用是最重要实体，多个使用相同应用名接入的进程，会表现为相同应用下的多个实例。所以应用是一个逻辑组合，在微服务架构中，可以等同为一个服务，包含多个对等的实例。
服务	特指微服务架构中的独立服务单元，通常与“应用” 同义。
实例	实例是应用在物理或虚拟环境中的一个实际部署单元，通常对应一个进程。
分布式链接追踪	分布式链路追踪 (Distributed Tracing) 是一种用于监控和分析分布式系统中请求流转路径的技术，主要用于追踪一个请求从发起端到最终处理完成的全过程，记录其在各个服务、组件间的调用关系、执行时间及状态，从而帮助定位性能瓶颈、排查故障并优化系统可靠性。
Span	Span 是分布式追踪系统中的最小工作单元，记录某个单一操作的执行过程（如调用一个服务、执行一次数据库查询、发送一个消息等）
Span ID	Span ID 是每个 Span 的唯一标识符，通过 Parent ID 与父 Span 关联，形成层级结构。
Trace	Trace 是一组关联的 Span 构成的有向无环图 (DAG)，表示一个请求在整个系统中的完整调用路径。
Trace ID	Trace 通过全局唯一的 Trace ID 将所有相关的 Span 串联起来，用于分析跨服务的端到端性能。
探针	探针 (Agent) 是核心数据采集组件，负责从应用运行环境中收集性能数据并发送至监控后端。
埋点	在应用程序中插入监控逻辑，用于收集性能数据的过程。通常情况下，探针提供了自动埋点方案，对业务代码无侵入。同时，用户也可以通过 SDK 的方式进行手动埋点。
OpenTelemetry	开源的分布式追踪标准，定义了 API 和数据模型，支持跨工具兼容。同时，OpenTelemetry 也以探针或 SDK 的方式，为主流的编程语言提供了埋点方案。
指标	指标 (Metrics) 是对应用在特定时间点或时间段内性能的定量测量，通常以时间序列数据形式存储，支持聚合计算（如平均值、最大值、分位数）和趋势分析。

快速入门

快速入门

步骤1：授予 APM 预设策略

应用性能监控 APM 默认创建了预设策略 QcloudAPMFullAccess（应用性能监控（APM）全读写访问权限）和 QcloudAPMReadOnlyFullAccess（应用性能监控（APM）只读访问权限），您可以通过搜索策略名称快速进行预设策略授权，更多信息请参见 [策略授予](#)。

在大多数使用场景下，只需要对子用户授予 APM 预设策略，就可以让子用户正常访问应用性能监控 APM。如果需要根据具体的资源和操作进行精细化授权，请参见 [策略语法](#) 进行操作。

步骤2：创建业务系统

业务系统用于分类管理应用，每个业务系统有唯一的 Token，应用接入的时候需要指定 Token。可以在业务系统级别设置存储时长、上报限额等参数，也可以基于业务系统实现权限管理和分账，不同业务系统之间的监控数据完全隔离。

1. 在左侧菜单栏中选择应用性能监控 > 资源管理。
2. 在业务系统管理页中选择对应的地域，单击新建，按照提示填写业务名称，选择计费模式、上报地域、链路存储时长等信息，勾选相关协议并单击确定。

步骤3：接入应用

在应用列表管理页单击接入应用。

1. 在应用列表管理页单击接入应用。
2. 在数据接入页选择接入语言。
3. 在接入指南页选择接入协议类型和上报方式。
4. 根据页面指引完成应用接入，单击完成即可完成应用接入。

接入指南

接入概述

应用性能监控（APM）提供了多种应用接入方式，以适配不同的编程语言以及应用部署环境。

接入方案选型

APM 支持 OpenTelemetry、Skywalking 等多种上报协议。OpenTelemetry 社区活跃度高，技术更迭迅速，广泛兼容主流编程语言、组件与框架，请优先选择 OpenTelemetry 方案，关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。

接入方式

应用类型	OpenTelemetry 方案	Skywalking 方案
Java	通过增强版 OpenTelemetry Java 探针接入（推荐）	通过 SkyWalking 协议上报
Go	通过 OpenTelemetry-Go 接入 Go 应用（推荐）	通过 SkyWalking-Go 接入 Go 应用
Python	通过 OpenTelemetry-Python 接入 Python 应用	-
PHP	通过 OpenTelemetry-PHP 接入 PHP 应用（推荐）	-
Node.js	通过 OpenTelemetry-JS 方案接入 Node.js 应用	-
.Net	通过 OpenTelemetry-dotnet 接入 .NET 应用（推荐）	-

接入Java应用

通过增强版 OpenTelemetry Java 探针接入

(推荐)

增强版 OpenTelemetry Java 探针 (Cloud Platform-OTel Java Agent) 基于开源社区的 OpenTelemetry-java-instrumentation 进行二次开发, 遵循 Apache License 2.0 协议, 在探针包中对 OpenTelemetry License 进行了引用。在开源探针的基础上, 增强版 OpenTelemetry Java 探针在埋点密度、高阶诊断、性能保护、企业级能力等方面做了重要的增强。

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合, 用来检测、生成、收集和导出遥测数据 (指标、日志和跟踪), 帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃, 技术更迭迅速, 广泛兼容主流编程语言、组件与框架, 为云原生微服务以及容器架构的链路追踪能力广受欢迎。

本文将通过相关操作介绍如何通过增强版 OpenTelemetry 探针接入 Java 应用。

前提条件

请参考 [OpenTelemetry 方案支持的组件和框架](#), 确保 Java 版本和应用服务器在探针支持的范围内。对于自动埋点支持的依赖库和框架, 在接入成功后即可完成数据上报, 不需要修改代码。同时, 增强版 OpenTelemetry Java 探针遵循了 OpenTelemetry 协议标准, 如果自动埋点不满足您的场景, 或者需要增加业务层埋点, 请参考 [自定义埋点](#), 使用 OpenTelemetry API 进行自定义埋点。

步骤1：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控, 单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中, 单击 Java 语言。
3. 在接入 Java 应用页面, 选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 获取接入点和 Token。

步骤2：下载探针

在控制台下载探针并保存，探针名为 `opentelemetry-javaagent.jar`。

步骤3：修改上报参数

接入 Java 应用需要用到如下3个 JVM 启动参数：

```
-javaagent:<javaagent>
-Dotel.resource.attributes=service.name=<serviceName>,token=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
```

在执行 Java 命令的时候，请确保这3个 JVM 启动参数放在 `-jar` 之前。对于无法直接指定 JVM 启动参数的应用，`-Dotel.resource.attributes` 系统参数可以替换为 `OTEL_RESOURCE_ATTRIBUTES` 环境变量，`-Dotel.exporter.otlp.endpoint` 系统参数可以替换为 `OTEL_EXPORTER_OTLP_ENDPOINT` 环境变量。对应的字段说明如下：

- `<javaagent>`：探针对应的本地文件路径。
- `<serviceName>`：应用名，多个使用相同应用名接入的进程，在 APM 中会表现为相同应用下的多个实例。对于 Spring Cloud 或 Dubbo 应用，应用名通常和服务名保持一致。最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- `<token>`：步骤1中拿到业务系统 Token。
- `<endpoint>`：步骤1中拿到的接入点。

下述内容以探针路径为 `/path/to/opentelemetry-javaagent.jar`，应用名为 `myService`，业务系统 Token 为 `myToken`，接入点为 [http://pl-demo.ap-*\\$\\${Region}*.apm.tencentcs.com:4317](http://pl-demo.ap-*$${Region}*.apm.tencentcs.com:4317) 为例，介绍不同环境的完整启动脚本：

- JAR File 或 Spring Boot

```
java -javaagent:/path/to/opentelemetry-javaagent.jar \
-Dotel.resource.attributes=service.name=myService,token=myToken\
-Dotel.exporter.otlp.endpoint=http://pl-demo.ap-${Region}.apm.tencentcs.com:4317 \
-jar SpringCloudApplication.jar
```

- Linux Tomcat 7/Tomcat 8

在 `{TOMCAT_HOME}/bin/setenv.sh` 配置文件添加以下内容：

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:/path/to/opentelemetry-javaagent.jar"
export OTEL_RESOURCE_ATTRIBUTES=service.name=myService,token=myToken
```

```
export OTEL_EXPORTER_OTLP_ENDPOINT=http://pl-demo.ap-${Region}.apm.tencentcs.com:4317
```

如果您的 Tomcat 没有 setenv.sh 配置文件，请参考 [Tomcat 官方文档](#) 初始化 setenv.sh 配置文件，或者使用其他方式添加 Java 启动参数以及环境变量。

- Jetty

在 <jetty_home>\>/bin/jetty.sh 启动脚本中添加以下内容：

```
JAVA_OPTIONS="$JAVA_OPTIONS -javaagent:/path/to/opentelemetry-javaagent.jar"
export OTEL_RESOURCE_ATTRIBUTES=service.name=myService,token=myToken
export OTEL_EXPORTER_OTLP_ENDPOINT=http://pl-demo.ap-${Region}.apm.tencentcs.com:4317
```

- IDEA

在 IDEA 中本地调试 Java 应用时，可在 Run Configuration 中配置 VM options，参数配置如下：

```
-javaagent:"/path/to/opentelemetry-javaagent.jar"
-Dotel.resource.attributes=service.name=myService,token=myToken
-Dotel.exporter.otlp.endpoint=http://pl-demo.ap-${Region}.apm.tencentcs.com:4317
```

在这种情况下，请确保本地环境和接入点之间的网络连通性，通常可以使用外网上报接入点地址。

- 其他应用服务器

请参考对应的配置规范挂载探针，并添加 Java 启动参数或环境变量。

接入验证

完成3个接入步骤后，启动 Java 应用，应用程序将挂载探针，并连接到 APM 服务端上报监控数据。在有正常流量的情况下，控制台应用性能监控 > 应用监控 > 应用列表中将展示接入的应用，控制台应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

自定义埋点（可选）

当自动埋点不满足您的场景或者需要增加业务层埋点时，您可参照下述内容，使用 OpenTelemetry API 添加自定义埋点。本文仅展示最基本的自定义埋点方式，OpenTelemetry 社区提供了更多灵活的自定义埋点方式，具体使用方法可参考 [OpenTelemetry 官方文档](#)。

引入 OpenTelemetry API 依赖

```
<dependencies>
  <!-- 其他依赖 -->
  <dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-api</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.opentelemetry</groupId>
      <artifactId>opentelemetry-bom</artifactId>
      <version>1.9.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

获取 Tracer

在需要进行埋点的代码中，可以通过如下代码获取 Tracer 对象：

```
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Tracer;

public class AcquireTracerDemo {
    public void acquireTracer() {
        // scope 用于定义埋点范围，一般情况下可以直接使用类名，关于Scope的更多信息，请参考 OpenTelemetry 官方文档
        String scope = this.getClass().getName();
        OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
        Tracer tracer = openTelemetry.getTracer(scope);
    }
}
```

设置对业务方法进行埋点

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.StatusCode;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Scope;
```

```
// Trace 对象可以在业务方法中获取，或者通过参数传入业务方法
public void doTask(Tracer tracer) {
    // 创建一个 Span
    Span span = tracer.spanBuilder("doTask").startSpan();
    // 在 Span 中添加一些 Attributes
    span.setAttribute("RequestId", "5fc92ff1-8ca8-45f4-8013-24b4b5257666");
    // 将此 Span 设置为当前的Span
    try (Scope scope = span.makeCurrent()) {
        doSubTask1();
        doSubTask2();
    } catch (Throwable t) {
        // 处理异常，异常信息将记录到 Span 的对应事件中
        span.recordException(t);
        span.setStatus(StatusCode.ERROR);
        throw t;
    } finally {
        // 结束 Span
        span.end();
    }
}
```

查看自定义埋点结果

在应用性能监控 > 调用查询中，找到相关的调用链，点击 Span ID，进入链路详情页面，即可查到通过自定义埋点新增的 Span。

通过 SkyWalking 协议上报

前提条件

- 打开 [SkyWalking](#) 下载页面，下载 SkyWalking 8.5.0 以上的（包含8.5.0）版本，并将解压后的 Agent 文件夹放至 Java 进程有访问权限的目录。

Index of /dist/skywalking/8.5.0

Name	Last modified	Size	Description
 Parent Directory		-	
 apache-skywalking-apm-8.5.0-src.tgz	2021-04-09 15:46	3.1M	
 apache-skywalking-apm-8.5.0-src.tgz.asc	2021-04-09 15:46	833	
 apache-skywalking-apm-8.5.0-src.tgz.sha512	2021-04-09 15:46	166	
 apache-skywalking-apm-8.5.0.tar.gz	2021-04-09 15:46	173M	
 apache-skywalking-apm-8.5.0.tar.gz.asc	2021-04-09 15:46	833	
 apache-skywalking-apm-8.5.0.tar.gz.sha512	2021-04-09 15:46	165	
 apache-skywalking-apm-es7-8.5.0.tar.gz	2021-04-09 15:46	176M	
 apache-skywalking-apm-es7-8.5.0.tar.gz.asc	2021-04-09 15:46	833	
 apache-skywalking-apm-es7-8.5.0.tar.gz.sha512	2021-04-09 15:46	169	

- 插件均放置在 /plugins 目录中。在启动阶段将新的插件放进该目录，即可令插件生效。将插件从该目录删除，即可令其失效。另外，日志文件默认输出到 /logs 目录中。
- 新的 Agent 文件夹目录如下所示：

目录：

+-- agent

+-- activations

apm-toolkit-log4j-1.x-activation.jar

apm-toolkit-log4j-2.x-activation.jar

apm-toolkit-logback-1.x-activation.jar

.....

+-- config

agent.config

+-- plugins

apm-dubbo-plugin.jar

apm-feign-default-http-9.x.jar

apm-httpClient-4.x-plugin.jar

.....

```
+-- optional-plugins
apm-gson-2.x-plugin.jar
.....
+-- bootstrap-plugins
jdk-http-plugin.jar
.....
+-- logs
skywalking-agent.jar
```

接入步骤

步骤1：获取接入点和 Token

1. 登录应用性能监控控制台。
2. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
3. 在右侧弹出的数据接入抽屉框中，单击 Java 语言。
4. 在接入Java应用页面，选择您所要接入的地域以及业务系统。
5. 选择接入协议类型为 Skywalking。
6. 获取接入点和 Token。

步骤2：下载 Skywalking

- 若您已经使用了 SkyWalking，可跳过本步骤。
- 若您还未使用 SkyWalking，建议 [下载最新版本](#)。

步骤3：配置相应参数及名称

SkyWalking Agent 支持多种方式完成参数配置，不同配置方式之间可以彼此组合，下面给出配置方式及其示例。

方法1：使用 agent.config 文件配置

打开 agent/config/agent.config 文件，配置接入点、Token 和自定义服务名称。

```
collector.backend_service=<接入点>
agent.authentication=<Token>
agent.service_name=<上报的服务名称>
```

说明：

修改完 agent.config 需要把配置项前的反注释符号 # 去掉。否则更改的信息将无法生效。

方法2：Java VM Options

启动 Java 应用程序时，在命令行中添加相应以 -DSkywalking 开头的参数，以下给出方法一的等效范例。

```
java -javaagent:<skywalking-agent-path>/skywalking-agent.jar
```

```
-Dskywalking.collector.backend_service= <接入点>
-Dskywalking.agent.authentication= <Token>
-Dskywalking.agent.service_name= <上报的服务名称> 要启动的程序
```

方法3：设置相应环境变量

可以在系统中设置相应环境变量来完成 SkyWalking 客户端的配置，以下为 Linux 命令的示例。

```
export SW_AGENT_AUTHENTICATION= <Token> (等效于agent.authentication=Token)
export SW_AGENT_NAME= <上报的服务名称> (等效于agent.service_name= <上报的服务名称>)
export SW_AGENT_COLLECTOR_BACKEND_SERVICES= <接入点> (等效于collector.backend_service=接入点)
```

说明

- 以上三种方式读取优先级关系为：服务器配置 > 环境变量 > 配置文件。优先级高的配置会将优先级的低的配置覆盖。
- 替换对应参数值时，“< >”符号需删去，仅保留文本。

步骤4：选择相应方法指定插件路径

根据应用的运行环境，选择相应的方法来指定 SkyWalking Agent 的路径。

- Linux Tomcat 7/Tomcat 8

- 在 tomcat/bin/catalina.sh 第一行添加以下内容：

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:<skywalking-agent-path>"; export CATALINA_OPTS
```

- Jetty

在 {JETTY_HOME}/start.ini 配置文件中添加以下内容：

```
--exec # 去掉前面的井号取消注释。
-javaagent:<skywalking-agent-path>
```

- JAR File 或 Spring Boot

- 在应用程序的启动命令行中添加 -javaagent 参数（-javaagent 参数一定要放在 -jar 参数之前），参数内容如下：

```
java -javaagent:<skywalking-agent-path> -jar yourApp.jar
```

- IDEA

- 在 IDEA 中运行时，可在 Configuration 中配置应用程序的 VM option,添加 -javaagent 参数，参数配置如上一

致。

步骤5：重新启动应用

完成上述部署步骤后，参见 [SkyWalking 官网指导](#) 重新启动应用即可。

接入 Go 应用

通过 OpenTelemetry-Go 接入 Go 应用（推荐）

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合，用于检测、生成、收集和导出遥测数据（指标、日志和跟踪），帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃，技术更迭迅速，广泛兼容主流编程语言、组件与框架，为云原生微服务以及容器架构的链路追踪能力广受欢迎。

本文将通过相关操作介绍如何通过社区的 OpenTelemetry-Go 方案接入 Go 应用。OpenTelemetry-Go 提供了一系列 API，用户可以通过 SDK 将性能数据并发送到可观测平台的服务端。本文通过最常见的应用行为，例如 HTTP 服务、访问数据库等，介绍如何基于 OpenTelemetry-Go 接入应用性能监控 APM，对于 OpenTelemetry-Go 的更多用法，请参考 [项目主页](#)。

前提条件

此方案支持 Go 的官方支持版本，目前为1.21和1.22，对于更低的版本，理论上可以接入，但社区不保持完整的兼容性，具体信息请参考社区 [兼容说明](#)。

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 Go 语言。
3. 在接入 Go 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 获取接入点和 Token。

接入 Go 应用

步骤1：引入 OpenTelemetry 相关依赖，实现 SDK 初始化逻辑

```
package main
```

```
import (  
    "context"  
    "errors"  
    "time"  
    "go.opentelemetry.io/otel"  
    "go.opentelemetry.io/otel/attribute"  
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"  
    "go.opentelemetry.io/otel/exporters/stdout/stdoutmetric"  
    "go.opentelemetry.io/otel/propagation"  
    "go.opentelemetry.io/otel/sdk/metric"  
    "go.opentelemetry.io/otel/sdk/resource"  
    "go.opentelemetry.io/otel/sdk/trace"  
    sdktrace "go.opentelemetry.io/otel/sdk/trace"  
    "log"  
)  
  
func setupOTelSDK(ctx context.Context) (shutdown func(context.Context) error, err error) {  
    var shutdownFuncs []func(context.Context) error  
  
    shutdown = func(ctx context.Context) error {  
        var err error  
        for _, fn := range shutdownFuncs {  
            err = errors.Join(err, fn(ctx))  
        }  
        shutdownFuncs = nil  
        return err  
    }  
  
    handleErr := func(inErr error) {  
        err = errors.Join(inErr, shutdown(ctx))  
    }  
  
    tracerProvider, err := newTraceProvider(ctx)  
    if err != nil {  
        handleErr(err)  
        return  
    }  
    shutdownFuncs = append(shutdownFuncs, tracerProvider.Shutdown)  
    otel.SetTracerProvider(tracerProvider)  
  
    meterProvider, err := newMeterProvider()  
    if err != nil {  
        handleErr(err)  
        return  
    }  
    shutdownFuncs = append(shutdownFuncs, meterProvider.Shutdown)  
    otel.SetMeterProvider(meterProvider)
```



```

    return
}

func newTraceProvider(ctx context.Context) (*trace.TracerProvider, error) {
    opts := []otlptracegrpc.Option{
        otlptracegrpc.WithEndpoint("<endpoint>"), // <endpoint>替换为上报地址
        otlptracegrpc.WithInsecure(),
    }
    exporter, err := otlptracegrpc.New(ctx, opts...)
    if err != nil {
        log.Fatal(err)
    }

    r, err := resource.New(ctx, []resource.Option{
        resource.WithAttributes(
            attribute.KeyValue{Key: "token", Value: attribute.StringValue("<token>")}, // <token>替换为业务系统Token
            attribute.KeyValue{Key: "service.name", Value: attribute.StringValue("<serviceName>")}, // <serviceName>替换为应用名
            attribute.KeyValue{Key: "host.name", Value: attribute.StringValue("<hostName>")}, // <hostName>替换为IP地址
        ),
    }...)
    if err != nil {
        log.Fatal(err)
    }

    tp := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(r),
    )
    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(propagation.TraceContext{}, propagation.Baggage{}))
    return tp, nil
}

func newMeterProvider() (*metric.MeterProvider, error) {
    metricExporter, err := stdoutmetric.New()
    if err != nil {
        return nil, err
    }

    meterProvider := metric.NewMeterProvider(
        metric.WithReader(metric.NewPeriodicReader(metricExporter,
            // Default is 1m. Set to 3s for demonstrative purposes.

```

```

        metric.WithInterval(3*time.Second))),
    )
    return meterProvider, nil
}

```

对应的字段说明如下，请根据实际情况进行替换。

- <serviceName> ：应用名，多个使用相同 serviceName 接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- <token> ：前置步骤中拿到业务系统 Token。
- <hostName> ：该实例的主机名，是应用实例的唯一标识，通常情况下可以设置为应用实例的 IP 地址。
- <endpoint> ：前置步骤中拿到的接入点。

步骤2：SDK 初始化，启动 HTTP 服务

```

package main

import (
    "context"
    "errors"
    "fmt"
    "log"
    "net"
    "net/http"
    "os"
    "os/signal"
    "time"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

func main() {
    if err := run(); err != nil {
        log.Fatalln(err)
    }
}

func run() (err error) {
    ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt)
    defer stop()

    // 初始化 SDK
    otelShutdown, err := setupOTelSDK(ctx)
    if err != nil {
        return
    }
}

```

```
// 优雅关闭
defer func() {
    err = errors.Join(err, otelShutdown(context.Background()))
}()

// 启动HTTP服务
srv := &http.Server{
    Addr:      ":8080",
    BaseContext: func(_ net.Listener) context.Context { return ctx },
    ReadTimeout: time.Second,
    WriteTimeout: 10 * time.Second,
    Handler:    newHTTPHandler(),
}
srvErr := make(chan error, 1)
go func() {
    srvErr <- srv.ListenAndServe()
}()

select {
case err = <-srvErr:
    return
case <-ctx.Done():
    stop()
}

err = srv.Shutdown(context.Background())
return
}
```

如果通过 Gin 等框架实现 HTTP 服务，埋点方式会存在区别，具体请参考社区的 [框架列表](#) 查看其他框架的埋点方式。

步骤3：对 HTTP 接口进行埋点增强

```
func newHTTPHandler() http.Handler {
    mux := http.NewServeMux()

    handleFunc := func(pattern string, handlerFunc func(http.ResponseWriter, *http.Request)) {
        // 对HTTP路由进行埋点
        handler := otelhttp.WithRouteTag(pattern, http.HandlerFunc(handlerFunc))
        mux.Handle(pattern, handler)
    }

    // 注册接口
    handleFunc("/simple", simpleIOHandler)

    // 对所有接口进行埋点增强
```

```
handler := otelhttp.NewHandler(mux, "/")
return handler
}

func simpleIOHandler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "ok")
}
```

接入验证

启动 Go 应用后，通过8080端口访问对应的接口，例如 `https://localhost:8080/simple`，应用就会向 APM 上报处理 HTTP 请求相关的链路数据。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

更多埋点示例

访问 Redis®

- 初始化

```
import (
    "github.com/redis/go-redis/v9"
    "github.com/redis/go-redis/extra/redisotel/v9"
)

var rdb *redis.Client

// InitRedis 初始化Redis®客户端
func InitRedis() *redis.Client {
    rdb := redis.NewClient(&redis.Options{
        Addr: "127.0.0.1:6379",
        Password: "", // no password
    })
    if err := redisotel.InstrumentTracing(rdb); err != nil {
        panic(err)
    }
    if err := redisotel.InstrumentMetrics(rdb); err != nil {
        panic(err)
    }
    return rdb
}
```

- 数据访问

```
func redisRequest(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    rdb := InitRedis()
    val, err := rdb.Get(ctx, "foo").Result()
    if err != nil {
        log.Printf("redis err.....")
        panic(err)
    }
    fmt.Println("redis res: ", val)
}
```

访问 MySQL

初始化

```
import (
    "gorm.io/driver/mysql"
    "gorm.io/gorm"
    "gorm.io/gorm/schema"
    "gorm.io/plugin/opentelemetry/tracing"
)

var GormDB *gorm.DB

type TableDemo struct {
    ID   int   `gorm:"column:id"`
    Value string `gorm:"column:value"`
}

func InitGorm() {
    var err error

    dsn := "root:4T$er3deffYuD#9Q@tcp(127.0.0.1:3306)/db_demo?charset=utf8mb4&parseTime=True&loc=Local"
    GormDB, err = gorm.Open(mysql.Open(dsn), &gorm.Config{
        NamingStrategy: schema.NamingStrategy{
            SingularTable: true, // 使用单数表名
        },
    })
    if err != nil {
        panic(err)
    }
    //加入tracing上报逻辑
    //需要根据实际情况填入DBName，在APM的拓扑图中，通过DBName字段确认节点，在本示例中使用"mockdb-mysql"
    if err = GormDB.Use(tracing.NewPlugin(tracing.WithoutMetrics(), tracing.WithDBName("mockdb-my
```

```
sql")); err != nil {  
    panic(err)  
}  
}
```

数据访问

```
func gormRequest(ctx context.Context) {  
    var val string  
    if err := gormclient.GormDB.WithContext(ctx).Model(&gormclient.TableDemo{}).Where("id = ?", 1).Pluck("value", &val).Error; err != nil {  
        panic(err)  
    }  
    fmt.Println("MySQL query result: ", val)  
}
```

内部方法埋点，以及设置 Span 的属性

通过以下代码对内部方法埋点，可以在当前的链路上下文中插入一个类型为 Internal 的 Span。

```
func internalSpanFunc(w http.ResponseWriter, r *http.Request) {  
    internalInvoke(r)  
    io.WriteString(w, "ok")  
}
```

```
func internalInvoke(r *http.Request) {  
    // 创建一个 Internal Span  
    _span := tracer.Start(r.Context(), "internalInvoke")  
    defer span.End()
```

```
    // 业务逻辑省略
```

```
    // 设置Span的Attributes  
    span.SetAttributes(attribute.KeyValue{
```

通过 SkyWalking-Go 接入 Go 应用

SkyWalking Go 是 SkyWalking 社区为 Go 应用提供的性能监控方案，可以在不需要修改业务代码的情况下，将 Go 应用接入应用性能监控 APM。关于 SkyWalking Go 的更多信息，请参考 [项目文档](#)。SkyWalking Go 对 Go 系常用依赖库和框架，包括 Gin、GORM、gRPC 等，提供了自动埋点，其他支持自动埋点的依赖库和框架请参考 SkyWalking 社区提供的 [完整列表](#)。

示例 Demo 应用

通过如下 Demo 代码，可以启动一个最简单的 HTTP 服务：

```
package main

import (
    "net/http"
)

func main() {
    http.HandleFunc("/hello", func(writer http.ResponseWriter, request *http.Request) {
        writer.Write([]byte("Hello World from skywalking-go-agent"))
    })
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        panic(err)
    }
}
```

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 Go 语言。
3. 在接入 Go 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 SkyWalking。
5. 获取接入点和 Token。

接入 Go 应用

步骤1：下载 Agent

前往 SkyWalking [下载页](#)，在 Go Agent 栏目，点击 Distribution，下载 tar 格式的 Agent 包，文件名后缀为 tgz。

解包后，得到 bin 目录下的二进制文件，根据当前操作系统选择对应的二进制文件，即为 Agent 文件。例如在 Linux 系统，Agent 文件为 skywalking-go-agent-0.4.0-linux-amd64。

步骤2：安装 Agent

SkyWalking Go 提供了2种方式安装 Agent，可以选择任一种方式安装：

Agent 注入方式

如果您不需要在代码中自定义埋点，可以选择 Agent 注入方式。执行命令如下：

```
/path/to/agent -inject /path/to/your/project [-all]
```

其中，/path/to/agent 为步骤1中得到的 Agent 文件，/path/to/your/project 为 Go 项目主目录。

代码依赖方式

执行如下命令，获得需要的依赖：

```
go get github.com/apache/skywalking-go
```

在 main 中引入依赖：

```
import _ "github.com/apache/skywalking-go"
```

步骤3：修改接入 APM 的配置

从社区的 [默认配置文件](#) 获取配置文件模板，保存为文本文件，可以命名为 config.yaml。

修改配置文件，至少需要配置如下3项：

```
agent:
  service_name: "<serviceName>" # <serviceName> 替换为应用名
reporter:
  grpc:
    backend_service: "<endpoint>" # <endpoint> 替换为上报地址
    authentication: "<token>" # <token> 替换为业务系统Token
```

对应的字段说明如下：

- <serviceName> ：应用名，多个使用相同 serviceName 接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- <token> ：前置步骤中拿到业务系统 Token。
- <endpoint> ：前置步骤中拿到的接入点。

步骤4：基于 SkyWalking-Go 编译项目

在编译 Go 项目的时候，添加如下参数：

```
-toolexec="/path/to/agent" -config /path/to/config.yaml -a
```

其中，/path/to/agent 为步骤1中得到的 Agent 文件，/path/to/config.yaml 为步骤3得到的配置文件。

假设编译的产出物为 test，完整命令为：

```
go build -toolexec='/path/to/agent -config /path/to/config.yaml' -a -o test .
```

接入验证

启动 Go 应用后，通过8080端口访问对应的接口，例如 <https://localhost:8080/hello>，应用就会向 APM 上报处理 HTTP 请求相关的链路数据。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

自定义链路埋点

当自动埋点不满足您的场景或者需要增加业务层埋点时，请参考社区的 [Tracing API 文档](#)，在代码中添加自定义链路埋点。

接入 Node.js 应用

通过 OpenTelemetry-JS 方案接入 Node.js 应用

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合，用来检测、生成、收集和导出遥测数据（指标、日志和跟踪），帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃，技术更迭迅速，广泛兼容主流编程语言、组件与框架，为云原生微服务以及容器架构的链路追踪能力广受欢迎。

本文将通过相关操作介绍如何通过 OpenTelemetry-JS 方案接入 Node.js 应用。

OpenTelemetry-JS 方案对于 Node.js 系的常用模块和框架，包括 Express、mysql、gRPC 等，提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的模块和框架请参考 OpenTelemetry 社区提供的 [完整列表](#)。

示例 Demo

示例代码 main.js 通过 Express 提供3个 HTTP 接口，对应的 MySQL 和 Redis® 服务请自行搭建，或直接创建云产品实例。

```
"use strict";

const axios = require("axios").default;
const express = require("express");
const redis = require('./utils/redis');
const dbHelper = require("./utils/db");
const app = express();

app.get("/remoteInvoke", async (req, res) => {
  const result = await axios.get("http://cloud.example.com");
  return res.status(200).send(result.data);
});

app.get("/redis", async(req, res) => {
  let queryRes = await redis.getKey("foo")
  res.json({ code: 200, result: queryRes})
})

app.get("/mysql", async(req, res) => {
  let select = `select * from table_demo`;
```

```
await dbHelper.query(select);
res.json({ code: 200, result: "mysql op ended"})
})

app.use(express.json());

app.listen(8080, () => {
  console.log("Listening on http://localhost:8080");
});
```

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 Node 语言。
3. 在接入 Node 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 获取您的接入点和 Token。

接入 Node.js 应用

步骤1：安装所需的依赖包

```
npm install --save @opentelemetry/api
npm install --save @opentelemetry/auto-instrumentations-node
```

步骤2：添加运行参数

通过如下命令启动 Node.js 应用：

```
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_RESOURCE_ATTRIBUTES='token=<token>,hostName=<hostName>'
export OTEL_EXPORTER_OTLP_PROTOCOL='grpc'
export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="<endpoint>"
export OTEL_SERVICE_NAME="<serviceName>"
export NODE_OPTIONS="--require @opentelemetry/auto-instrumentations-node/register"
node main.js
```

对应的字段说明如下：

- <serviceName> ：应用名，多个使用相同 serviceName 接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- <token> ：前置步骤中拿到业务系统 Token。
- <hostName> ：该实例的主机名，是应用实例的唯一标识，通常情况下可以设置为应用实例的 IP 地址。

- <endpoint> : 前置步骤中拿到的接入点。

下述内容以应用名为 myService , 业务系统 Token 为 myToken , 主机名为 192.168.0.10 , 接入点以 [http://pl-demo.ap-\\${Region}.apm.tencentcs.com:4317](http://pl-demo.ap-${Region}.apm.tencentcs.com:4317)为例, 完整的启动命令为:

```
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_RESOURCE_ATTRIBUTES='token=myToken,hostName=192.168.0.10'
export OTEL_EXPORTER_OTLP_PROTOCOL='grpc'
export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://pl-demo.ap-${Region}.apm.tencentcs.com:4317"
export OTEL_SERVICE_NAME="myService"
export NODE_OPTIONS="--require @opentelemetry/auto-instrumentations-node/register"
node main.js
```

接入验证

启动 Node.js 应用后, 通过8080端口访问对应的接口, 例如 <https://localhost:8080/>。在有正常流量的情况下, 应用性能监控 > 应用监控 > 应用列表 中将展示接入的应用, 应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时, 如果接入后在控制台没有查询到应用或实例, 请等待30秒左右。

自定义埋点 (可选)

当自动埋点不满足您的场景或者需要增加业务层埋点时, 您可参照下述内容, 使用 OpenTelemetry API 添加自定义埋点。本文仅展示最基本的自定义埋点方式, OpenTelemetry 社区提供了更多灵活的自定义埋点方式, 具体使用方法可参考 OpenTelemetry 社区提供的 [Javascript 自定义埋点文档](#)。

```
const opentelemetry = require("@opentelemetry/api")

app.get("/attr", async(req, res) => {
  const tracer = opentelemetry.trace.getTracer(
    'my-service-tracer'
  );
  tracer.startActiveSpan('new internal span', span => {
    span.addEvent("Acquiring lock", {
      'log.severity': 'error',
      'log.message': 'data node found',
    })
    span.addEvent("Got lock, doing work...", {
      'log.severity': '11111',
      'log.message': '222222',
      'log.message1': '3333333',
    })
    span.addEvent("Unlocking")
    span.end();
  });
});
```

```
});  
res.json({ code: 200, msg: "success" });  
})
```

接入 Python 应用

通过 OpenTelemetry-Python 接入 Python 应用

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合，用来检测、生成、收集和导出遥测数据（指标、日志和跟踪），帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃，技术更迭迅速，广泛兼容主流编程语言、组件与框架，为云原生微服务以及容器架构的链路追踪能力广受欢迎。

本文将通过相关操作介绍如何通过社区的 OpenTelemetry-Python 方案接入 Python 应用。

OpenTelemetry-Python 方案对于 Python 系的常用依赖库和框架，包括 Flask、Django、FastAPI、MySQL Connector 等，提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考 OpenTelemetry 社区提供的 [完整列表](#)。

前提条件

此方案支持 Python 3.6 及以上版本。

示例 Demo

所需依赖如下：

```
pip install flask
pip install mysql-connector-python
pip install redis
pip install requests
```

示例代码 app.py 通过 Flask 框架提供3个 HTTP 接口，对应的 MySQL 和 Redis® 服务请自行搭建，或直接创建云产品实例。

```
from flask import Flask
import requests
import time
import mysql.connector
import redis
```

```
backend_addr = 'https://example.com/'
```

```
app = Flask(__name__)
```

```
# 访问外部站点
@app.route('/')
def index():
    start = time.time()
    r = requests.get(backend_addr)
    return r

# 访问数据库
@app.route('/mysql')
def func_rdb():
    cnx = mysql.connector.connect(host='127.0.0.1', database="<DB-NAME>", user='<DB-USER>', password='<DB-PASSWORD>', auth_plugin='mysql_native_password')
    cursor = cnx.cursor()
    val = "null"
    cursor.execute("select value from table_demo where id=1;")
    val = cursor.fetchone()[0]
    cursor.close()
    cnx.close()
    return "rdb res:" + val

# 访问Redis
@app.route("/redis")
def func_kvop():
    client = redis.StrictRedis(host="localhost", port=6379)
    val = "null"
    val = client.get('foo').decode("utf8")
    return "kv res:" + val

app.run(host='0.0.0.0', port=8080)
```

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 Python 语言。
3. 在接入 Python 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 获取接入点和 Token。

接入 Python 应用

步骤1：安装所需的依赖包

```
pip install opentelemetry-instrumentation-redis
pip install opentelemetry-instrumentation-mysql
```

```
pip install opentelemetry-distro opentelemetry-exporter-otlp
```

```
opentelemetry-bootstrap -a install
```

步骤2：添加运行参数

通过如下命令启动 Python 应用：

```
opentelemetry-instrument \  
--traces_exporter otlp_proto_grpc \  
--metrics_exporter none \  
--service_name <serviceName> \  
--resource_attributes token=<token>,host.name=<hostName> \  
--exporter_otlp_endpoint <endpoint> \  
python3 app.py
```

对应的字段说明如下：

- **<serviceName>**：应用名，多个使用相同 serviceName 接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- **<token>**：前置步骤中拿到业务系统 Token。
- **<hostName>**：该实例的主机名，是应用实例的唯一标识，通常情况下可以设置为应用实例的 IP 地址。
- **<endpoint>**：前置步骤中拿到的接入点。

下述内容以应用名为 myService，业务系统 Token 为 myToken，主机名为 192.168.0.10，接入点以 https://pl-demo.ap-`\${Region}`.apm.tencentcs.com:4317 为例，完整的启动命令为：

```
opentelemetry-instrument \  
--traces_exporter otlp_proto_grpc \  
--metrics_exporter none \  
--service_name myService \  
--resource_attributes token=myToken,host.name=192.168.0.10 \  
--exporter_otlp_endpoint https://pl-demo.ap-${Region}.apm.tencentcs.com:4317/ \  
python3 app.py
```

接入验证

启动 Python 应用后，通过8080端口访问对应的接口，例如 https://localhost:8080/。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

Django 应用注意事项

如果您的应用使用 Django 框架，通过 OpenTelemetry-Python 方案接入前需要注意如下事项：

1. 推荐使用 uWSGI 方式部署服务，部署方式请参考 [通过 uWSGI 托管 Django 应用](#)，直接通过 Python 命令启动可能会造成上报失败。
2. OpenTelemetry-Python 的引入，可能会导致 Django 应用不再使用默认的配置文件的，需要通过环境变量重新指定配置文件：

```
export DJANGO_SETTINGS_MODULE=mysite.settings
```

自定义埋点（可选）

当自动埋点不满足您的场景或者需要增加业务层埋点时，您可参照下述内容，使用 OpenTelemetry API 添加自定义埋点。本文仅展示最基本的自定义埋点方式，OpenTelemetry 社区提供了更多灵活的自定义埋点方式，具体使用方法可参考 OpenTelemetry 社区提供的 [Python 自定义埋点文档](#)。

```
from opentelemetry import trace
import requests
backend_addr = 'https://example.com/'
app = Flask(__name__)

@app.route('/')
def index():
    r = requests.get(backend_addr) # 对于requests.get()发起的外部请求，OpenTelemetry-Python会自动埋点
    slow() # 调用一个自定义函数
    return r

def slow():
    tracer = trace.get_tracer(__name__)
    # 自定义函数不在OpenTelemetry-Python自动埋点范围内，因此增加一个自定义埋点
    with tracer.start_as_current_span("child_span")
        time.sleep(5)
    return
```

接入 PHP 应用

通过 OpenTelemetry-PHP 接入 PHP 应用 (推荐)

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合，用来检测、生成、收集和导出遥测数据（指标、日志和跟踪），帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃，技术更迭迅速，广泛兼容主流编程语言、组件与框架，为云原生微服务以及容器架构的链路追踪能力广受欢迎。

本文将通过相关操作介绍如何通过社区的 OpenTelemetry-PHP 方案接入 PHP 应用。

OpenTelemetry-PHP 方案对于 PHP 系的常用依赖库和框架，例如 Slim 等，提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考 OpenTelemetry 社区提供的 [完整列表](#)。

前提条件

安装如下工具：

- [PECL](#)
- [composer](#)

并确保 shell 中可以运行以下命令：

```
php -v
composer -v
```

自动接入

- PHP 8.0+
- 目前自动埋点支持的框架列表详情请参见 [OpenTelemetry 官方文档](#)。

手动接入

PHP 7.4+

Demo 应用

示例代码 index.php 是一个 HTTP Server 使用 PDO 连接 MySQL 数据库数据库操作，对应的 MySQL 务请自行搭建，或直接创建云产品实例。

1. 初始化应用

```
mkdir <project-name> && cd <project-name>
```

```
composer init \  
--no-interaction \  
--stability beta \  
--require slim/slim:"^4" \  
--require slim/psr7:"^1" \  
composer update
```

2. 编写业务代码

在目录下创建一个 index.php 文件，添加如下内容。

以下内容将使用一个 HTTP Server 接口模拟使用 PDO 连接 MySQL 进行一次搜索操作。

```
<?php  
use Psr\Http\Message\ResponseInterface as Response;  
use Psr\Http\Message\ServerRequestInterface as Request;  
use Slim\Factory\AppFactory;  
  
require __DIR__ . '/vendor/autoload.php';  
  
$app = AppFactory::create();  
  
$app->get('/getID', function (Request $request, Response $response) {  
  
    $dbms = 'mysql';    // 数据库类型  
    $host = 'localhost'; // 数据库主机名  
    $dbName = 'Mydb';   // 使用的数据库  
    $user = 'root';     // 数据库连接用户名  
    $pass = '';         // 对应的密码  
    $dsn = "$dbms:host=$host;dbname=$dbName";  
  
    try {  
        $dbh = new PDO($dsn, $user, $pass); // 初始化一个PDO对象  
        echo "连接成功<br/>";  
  
        foreach ($dbh->query('SELECT id from userInfo') as $row) {  
            $response->getBody()->write($row[0] . "<br/>");  
        }  
    }  
}
```

```
$dbh = null;
} catch (PDOException $e) {
    die ("Error!: " . $e->getMessage() . "<br/>");
}

return $response;
});

$app->run();
```

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 PHP 语言。
3. 在接入 PHP 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 上报方式选择您想要的上报方式，获取您的接入点和 Token。

说明：

- 内网上报：使用此上报方式，您的服务需运行在云平台 VPC。通过 VPC 直接联通，在避免外网通信的安全风险同时，可以节省上报流量开销。
- 外网上报：当您的服务部署在本地或非云平台 VPC 内，可以通过此方式上报数据。请注意外网通信存在安全风险，同时也会造成一定上报流量。

自动接入方案（推荐）

步骤1：构建 OpenTelemetry PHP Extension

说明：

如果已经构建过 OpenTelemetry PHP extension，可跳过当前步骤。

1. 下载构建 OpenTelemetry PHP extension 所需要的工具：

- macOS

```
brew install gcc make autoconf
```

- Linux (apt)

```
sudo apt-get install gcc make autoconf
```

2. 使用 PECL 构建 OpenTelemetry PHP 扩展：

```
pecl install opentelemetry
```

注意：

构建成功时输出内容的最后几行如下（路径可能不完全一致）：

```
Build process completed successfully
Installing '/opt/homebrew/Cellar/php/8.2.8/pecl/2020829/opentelemetry.so'
install ok: channel://pecl.php.net/opentelemetry-1.0.3
Extension opentelemetry enabled in php.ini
```

3. 启用 OpenTelemetry PHP 扩展。

说明：

如果上一步输出了 Extension opentelemetry enabled in php.ini，表明已经启用，请跳过当前步骤。

在 php.ini 文件中添加如下内容：

```
[opentelemetry]
extension=opentelemetry.so
```

php.ini 文件可能存在的位置：

OS	PATH
Linux	/etc/php.ini /usr/bin/php5/bin/php.ini /etc/php/php.ini /etc/php5/apache2/php.ini
Mac OSX	/private/etc/php.ini
Windows (with XAMPP installed)	C:/xampp/php/php.ini

4. 验证是否构建并启用成功。

- 方法一：

```
php -m | grep opentelemetry
```

预期输出：

```
opentelemetry
```

- 方法二：

```
php --ri opentelemetry
```

预期输出：

```
opentelemetry
opentelemetry support => enabled
extension version => 1.0.3
```

5. 为应用添加 OpenTelemetry PHP 自动埋点需要的额外依赖。

```
pecl install grpc # 这一步构建时间较长
```

```
composer require \
    open-telemetry/sdk \
    open-telemetry/exporter-otlp \
    open-telemetry/transport-grpc \
    php-http/guzzle7-adapter \
    open-telemetry/opentelemetry-auto-slim \
    open-telemetry/opentelemetry-auto-pdo
```

- open-telemetry/sdk：OpenTelemetry PHP SDK。
- open-telemetry/exporter-otlp：OpenTelemetry PHP OTLP 协议数据上报所需的依赖。
- open-telemetry/opentelemetry-auto-slim：OpenTelemetry PHP 针对 Slim 框架实现的自动埋点包。
- open-telemetry/opentelemetry-auto-pdo：OpenTelemetry PHP 针对 PHP DataObject 实现的自动埋点包。

说明：

这里导入 open-telemetry/opentelemetry-auto-slim 和 open-telemetry/opentelemetry-auto-pdo 包是因为示例 demo 中使用了 PDO 和 Slim 框架，可以根据具体业务进行调整。如果业务中组件需要 OpenTelemetry 自动埋点，需要在项目中导入对应的自动埋点包，自动埋点包导入方式具体详情请

参见 [OpenTelemetry 官方文档](#)。

步骤2：运行应用

1. 执行以下命令：

```
env OTEL_PHP_AUTOLOAD_ENABLED=true \  
  OTEL_TRACES_EXPORTER=otlp \  
  OTEL_METRICS_EXPORTER=none \  
  OTEL_LOGS_EXPORTER=none \  
  OTEL_EXPORTER_OTLP_PROTOCOL=grpc \  
  OTEL_EXPORTER_OTLP_ENDPOINT=<endpoint> \ # 此处替换成步骤1中获得的接入点  
  OTEL_RESOURCE_ATTRIBUTES="service.name=<service-name>,token=<token>" \ # 此处<servic  
e-name>改为自定义服务名，<token>替换成步骤1中获得的token  
  OTEL_PROPAGATORS=baggage,tracecontext \  
  php -S localhost:8080
```

2. 在浏览器中访问以下链接：

<http://localhost:8080/getID>

每次进入该页面，OpenTelemetry 都会自动创建 Trace，并将链路数据上报至 APM。

接入验证

启动 PHP 应用后，通过8080端口访问对应的接口，例如 <https://localhost:8080/getID>。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表 中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控 中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

自定义埋点（可选）

当自动埋点不满足您的场景或者需要增加业务层埋点时，您可参照下述内容，使用 OpenTelemetry PHP SDK 添加自定义埋点。本文仅展示最基本的自定义埋点方式，OpenTelemetry 社区提供了更多灵活的自定义埋点方式，具体使用方法可参考 OpenTelemetry 社区提供的 [PHP 自定义埋点文档](#)。

```
<?php
```

```
use OpenTelemetry\API\Globals; // 必须的包
```

```
require __DIR__ . '/vendor/autoload.php';
```

```
function wait(): void
```

```
{
```

```
// 通过Globals包获取当前已经配置的providers
$tracerProvider = Globals::tracerProvider();
$tracer = $tracerProvider->getTracer(
    'instrumentation-scope-name', //name (required)
    'instrumentation-scope-version', //version
    'http://example.com/my-schema', //schema url
    ['foo' => 'bar'] //attributes
);

// 自定义埋点
$span = $tracer->spanBuilder("wait")->startSpan();

// 业务代码
sleep(5);

// 自定义埋点结束
$span->end();
}

wait();
```

手动接入方案

若 PHP 应用版本不能满足8.0+，但能满足7.4+，可以选择手动埋点上报。本文仅展示最基本的手动埋点方式，OpenTelemetry 社区提供了更多灵活的手动埋点方式，具体使用方法可参考 OpenTelemetry 社区提供的 [PHP 手动接入文档](#)。

导入 OpenTelemetry PHP SDK 以及 OpenTelemetry gRPC Explorer 所需依赖

1. 下载 PHP HTTP 客户端库，用于链路数据上报。

```
composer require guzzlehttp/guzzle
```

2. 下载 OpenTelemetry PHP SDK。

```
composer require \
open-telemetry/sdk \
open-telemetry/exporter-otlp
```

3. 下载使用 gRPC 上报数据时所需依赖。


```
pecl install grpc # 如果之前已经下载过grpc, 可以跳过这一步
composer require open-telemetry/transport-grpc
```

编写 OpenTelemetry 初始化工具类

在 index.php 文件所在目录中创建 opentelemetry_util.php 文件。并在文件中添加如下代码：

```
<?php
// 包含设置应用名、Trace导出方式、Trace上报接入点, 并创建全局TraceProvide

use OpenTelemetry\API\Globals;
use OpenTelemetry\API\Trace\Propagation\TraceContextPropagator;
use OpenTelemetry\Contrib\Otlp\SpanExporter;
use OpenTelemetry\SDK\Common\Attribute\Attributes;
use OpenTelemetry\SDK\Common\Export\Stream\StreamTransportFactory;
use OpenTelemetry\SDK\Resource\ResourceInfo;
use OpenTelemetry\SDK\Resource\ResourceInfoFactory;
use OpenTelemetry\SDK\Sdk;
use OpenTelemetry\SDK\Trace\Sampler\AlwaysOnSampler;
use OpenTelemetry\SDK\Trace\Sampler\ParentBased;
use OpenTelemetry\SDK\Trace\SpanProcessor\SimpleSpanProcessor;
use OpenTelemetry\SDK\Trace\SpanProcessor\BatchSpanProcessorBuilder;
use OpenTelemetry\SDK\Trace\TracerProvider;
use OpenTelemetry\SemConv\ResourceAttributes;
use OpenTelemetry\Contrib\Grpc\GrpcTransportFactory;
use OpenTelemetry\Contrib\Otlp\OtlpUtil;
use OpenTelemetry\API\Signals;

// OpenTelemetry 初始化配置 (需要在PHP应用初始化时就进行OpenTelemetry初始化配置)
function initOpenTelemetry()
{
    // 1. 设置 OpenTelemetry 资源信息
    $resource = ResourceInfoFactory::emptyResource()->merge(ResourceInfo::create(Attributes::create([
        ResourceAttributes::SERVICE_NAME => '<your-service-name>', // 应用名, 必填, 如php-opentelemetry-demo
        ResourceAttributes::HOST_NAME => '<your-host-name>' // 主机名, 选填
    ])));
    'token' => '<your-token>' // 替换成步骤1中获得的 Token
    ]));

    // 2. 创建将 Span 输出到控制台的 SpanExplorer
    // $spanExporter = new SpanExporter(
    // (new StreamTransportFactory())->create('php://stdout', 'application/json')
    // );
```

```
// 2. 创建通过 gRPC 上报 Span 的 SpanExplorer
$transport = (new GrpcTransportFactory())->create('<grpc-endpoint>' . OtlpUtil::method(Signals::TRACE)); # 替换成步骤1中获得的接入点信息
$spanExporter = new SpanExporter($transport);

// 3. 创建全局的 TracerProvider , 用于创建 tracer
$tracerProvider = TracerProvider::builder()
->addSpanProcessor(
(new BatchSpanProcessorBuilder($spanExporter))->build()
)
->setResource($resource)
->setSampler(new ParentBased(new AlwaysOnSampler()))
->build();

Sdk::builder()
->setTracerProvider($tracerProvider)
->setPropagator(TraceContextPropagator::getInstance())
->setAutoShutdown(true) // PHP 程序退出后自动关闭 tracerProvider , 保证链路数据都被上报
->buildAndRegisterGlobal(); // 将 tracerProvider 添加到全局

}
?>
```

修改应用代码，使用 OpenTelemetry API 创建 Span

1. 在 index.php 文件中导入所需包：

```
<?php

use OpenTelemetry\API\Globals;
use OpenTelemetry\API\Trace\StatusCode;
use OpenTelemetry\API\Trace\SpanKind;
use OpenTelemetry\SDK\Common\Attribute\Attributes;
use OpenTelemetry\SDK\Trace\TracerProvider;

use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use Slim\Factory\AppFactory;

require __DIR__ . '/opentelemetry_util.php';
```

2. 调用 initOpenTelemetry 方法完成初始化，需要在 PHP 应用初始化时就进行 OpenTelemetry 初始化配置：

```
// OpenTelemetry 初始化，包含设置应用名、Trace导出方式、Trace上报接入点，并创建全局TraceProvider
initOpenTelemetry();
```

3. 在 rolldice 接口中创建 Span。

```
/**
 * 1. 接口功能：模拟扔骰子，返回一个1-6之间的随机正整数
 * 并演示如何创建Span、设置属性、事件、带有属性的事件
 */
$app->get('/rolldice', function (Request $request, Response $response) {
    // 获取 tracer
    $tracer = \OpenTelemetry\API\Globals::tracerProvider()->getTracer('my-tracer');
    // 创建 Span; 设置span kind，不设置默认为KIND_INTERNAL
    $span = $tracer->spanBuilder("/rolldice")->setSpanKind(SpanKind::KIND_SERVER)->startSpan();
    // 为 Span 设置属性
    $span->setAttribute("http.method", "GET");
    // 为 Span 设置事件
    $span->addEvent("Init");
    // 设置带有属性的事件
    $eventAttributes = Attributes::create([
        "key1" => "value",
        "key2" => 3.14159,
    ]);

    // 业务代码
    $result = random_int(1,6);
    $response->getBody()->write(strval($result));

    $span->addEvent("End");
    // 销毁 Span
    $span->end();

    return $response;
});
```

4. 创建嵌套 Span。

新建一个 rolltwodices 接口，模拟扔两个骰子，返回两个1-6之间的随机正整数。以下代码演示如何创建嵌套的 Span：

```
$app->get('/rolltwodices', function (Request $request, Response $response) {
    // 获取 tracer
    $tracer = \OpenTelemetry\API\Globals::tracerProvider()->getTracer('my-tracer');
```

```
// 创建 Span
$parentSpan = $tracer->spanBuilder("/rolltwodices/parent")->setSpanKind(SpanKind::KIND_SERVER)->startSpan();
$scope = $parentSpan->activate();

$value1 = random_int(1,6);

$childSpan = $tracer->spanBuilder("/rolltwodices/parent/child")->startSpan();

// 业务代码
$value2 = random_int(1,6);
$result = "dice1: " . $value1 . ", dice2: " . $value2;

// 销毁 Span
$childSpan->end();
$parentSpan->end();
$scope->detach();

$response->getBody()->write(strval($result));
return $response;
});
```

5. 使用 Span 记录代码中发生的异常。

新建 error 接口，模拟接口发生异常。以下代码演示如何在代码发生异常时使用 Span 记录状态：

```
$app->get('/error', function (Request $request, Response $response) {
    // 获取 tracer
    $tracer = \OpenTelemetry\API\Globals::tracerProvider()->getTracer('my-tracer');
    // 创建 Span
    $span3 = $tracer->spanBuilder("/error")->setSpanKind(SpanKind::KIND_SERVER)->startSpan();
    try {
        // 模拟代码发生异常
        throw new \Exception('exception!');
    } catch (\Throwable $t) {
        // 设置Span状态为error
        $span3->setStatus(\OpenTelemetry\API\Trace\StatusCode::STATUS_ERROR, "expcetion in span 3!");
        // 记录异常栈轨迹
        $span3->recordException($t, ['exception.escaped' => true]);
    } finally {
        $span3->end();
    }
    $response->getBody()->write("error");
    return $response;
});
```

运行应用

1. 执行以下命令：

```
php -S localhost:8080
```

2. 在浏览器中访问以下链接：

```
http://localhost:8080/rolldice  
http://localhost:8080/rolltwodices  
http://localhost:8080/error
```

每次访问页面，OpenTelemetry 会创建链路数据，并将链路数据上报至 APM。

接入验证

启动 PHP 应用后，通过8080端口访问对应的接口，例如 <https://localhost:8080/getID>。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表 中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控 中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

接入 .Net 应用

通过 OpenTelemetry-dotnet 接入 .NET 应用 (推荐)

说明：

- OpenTelemetry 是工具、API 和 SDK 的集合，用于检测、生成、收集和导出遥测数据（指标、日志和跟踪），帮助用户分析软件的性能和行为。关于 OpenTelemetry 的更多信息请参考 [OpenTelemetry 官方网站](#)。
- OpenTelemetry 社区活跃，技术更迭迅速，广泛兼容主流编程语言、组件与框架，为云原生微服务以及容器架构的链路追踪能力广受欢迎。

OpenTelemetry-dotnet 方案对于 .NET 的常用依赖以及框架，包括 ASPNET、HTTPCLIENT、MYSQLCONNECTOR 等，提供了自动接入，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动接入的依赖库和框架请参考 OpenTelemetry 社区提供的 [完整列表](#)。本文将为您介绍如何使用 OpenTelemetry .NET SDK 接入 .NET 应用数据。

前提条件

OpenTelemetry .NET 支持自动接入和手动接入。

自动接入

- 支持自动接入的版本：
 - .NET SDK ≥ 6
 - .NET Framework 暂不支持自动接入
- 支持自动接入的框架请参见 [OpenTelemetry官方文档](#)。

手动接入

- 支持手动接入的版本：
 - .NET ≥ 5.0
 - .NET Core ≥ 2.0
 - .NET Framework $\geq 4.6.1$
- 支持手动接入的框架请参见 [OpenTelemetry官方文档](#)。

Demo 应用

示例代码 Program.cs 是一个 WebApplication 使用 MySQLConnector 连接 MySQL 数据库操作，对应的 MySQL 务请自行搭建，或直接创建云产品实例。

1. 初始化应用。

```
dotnet new web
```

2. 引入 Demo 中用到的依赖包。

其中 Microsoft.Extensions.Logging.Abstractions 是 MySQLConnector 所必需的包。

```
dotnet add package MySQLConnector;  
dotnet add package Microsoft.Extensions.Logging.Abstractions;
```

3. 修改 Properties/launchSettings.json 文件内容。

```
{  
  "$schema": "http://json.schemastore.org/launchsettings.json",  
  "profiles": {  
    "http": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": true,  
      "applicationUrl": "http://localhost:8080",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    }  
  }  
}
```

4. 编写业务代码 Program.cs

Program.cs 中使用 WebApplication 接口模拟使用 MySQLConnector 连接 MySQL 数据库操作。

```
using MySQLConnector;  
  
var builder = WebApplication.CreateBuilder(args);  
var app = builder.Build();  
  
app.MapGet("/", () => "Hello World!");  
  
app.MapGet("/todo", () =>
```

```
{
    string connectionString = "server=localhost;port=3306;User Id=root;password=;Database=MyDB";

    using (MySqlConnection myConnect = new MySqlConnection(connectionString))
    {
        try
        {
            myConnect.Open();
            using (MySqlCommand myCmd = new MySqlCommand("select * from MyTable", myConnect))
            {
                using (MySqlDataReader reader = myCmd.ExecuteReader())
                {
                    List<Int32> results = new List<Int32>();
                    while (reader.Read())
                    {
                        // 假设我们只处理第一列的数据ID
                        results.Add(reader.GetInt32(0));
                    }
                    return "ids:" + string.Join(", ", results);
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"数据库操作出错: {ex.Message}");
            return "数据库操作出错";
        }
    }
});

app.Run();
```

前置步骤：获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控 > 应用列表，单击接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击 .NET 语言。
3. 在接入 .NET 应用页面，选择您所要接入的地域以及业务系统。
4. 选择接入协议类型为 OpenTelemetry。
5. 上报方式选择您所要的上报方式，获取您的接入点和 Token。

说明：

- 内网上报：使用此上报方式，您的服务需运行在云平台VPC。通过 VPC 直接联通，在避免外网通

信的安全风险同时，可以节省上报流量开销。

- 外网上报：当您的服务部署在本地或非云平台VPC 内，可以通过此方式上报数据。请注意外网通信存在安全风险，同时也会造成一定上报流量。

自动接入方案（推荐）

步骤1：安装 opentelemetry-dotnet-instrumentation

1. 下载并执行 OpenTelemetry .NET 自动埋点安装脚本。

```
curl -L -O https://github.com/open-telemetry/opentelemetry-dotnet-instrumentation/releases/latest/download/otel-dotnet-auto-install.sh
```

```
chmod -x otel-dotnet-auto-install.sh
```

```
sh ./otel-dotnet-auto-install.sh
```

2. 设置环境变量并运行 OpenTelemetry .NET 自动埋点脚本。

```
chmod -x $HOME/.otel-dotnet-auto/instrument.sh
```

```
export OTEL_TRACES_EXPORTER=otlp \  
  OTEL_METRICS_EXPORTER=none \  
  OTEL_LOGS_EXPORTER=none \  
  OTEL_SERVICE_NAME=<service-name> \ # 此处<service-name>改为自定义服务名  
  OTEL_EXPORTER_OTLP_PROTOCOL=grpc \  
  OTEL_EXPORTER_OTLP_ENDPOINT=<endpoint> \ # 此处替换成步骤1中获得的接入点  
  OTEL_RESOURCE_ATTRIBUTES="token=<token>" \ # 此处替换成步骤1中获得的token  
. $HOME/.otel-dotnet-auto/instrument.sh
```

说明：

如果在 macOS 上安装 opentelemetry-dotnet-instrumentation，需要额外安装 [coreutils](#) 工具。

```
brew install coreutils
```

步骤2：运行应用

1. 构建并运行应用。

```
dotnet build  
dotnet run
```

2. 在浏览器中访问以下链接：

```
http://localhost:8080
http://localhost:8080/todo
```

每次进入该页面，OpenTelemetry 都会自动创建 Trace，并将链路数据上报至 APM。

接入验证

启动 .NET 应用后，通过8080端口访问对应的接口，例如 `https://localhost:8080/todo`。在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表 中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

自定义埋点（可选）

当自动埋点不满足您的场景或者需要增加业务层埋点时，您可参照下述内容，使用 `opentelemetry-dotnet-instrumentation` 添加自定义埋点。本文仅展示最基本的自定义埋点方式，OpenTelemetry 社区提供了更多灵活的自定义埋点方式，具体使用方法可参考 OpenTelemetry 社区提供的 [.NET 自定义埋点文档](#)。

引入 `System.Diagnostics.DiagnosticSource` 依赖包

```
dotnet add package System.Diagnostics.DiagnosticSource
```

创建 `ActivitySource` 实例

```
private static readonly ActivitySource RegisteredActivity = new ActivitySource("Examples.ManualInstrumentations.Registered");
```

创建 `Activity`

```
using (var activity = RegisteredActivity.StartActivity("Main"))
{
    // 添加 Activity Tag
    activity?.SetTag("key", "3.1415");
    // 为 Activity 添加一个 Event
    activity?.AddEvent(new("something happened"));

    // 一些业务代码
    Thread.Sleep(1000);
}
```

在 `OpenTelemetry.AutoInstrumentation` 中注册 `ActivitySource`

在环境变量中设置 `OTEL_DOTNET_AUTO_TRACES_ADDITIONAL_SOURCES`，没有通过该配置注册的 `ActivitySource` 将不会接入上报。

```
export OTEL_DOTNET_AUTO_TRACES_ADDITIONAL_SOURCES=Examples.ManualInstrumentations.Registered
```

手动接入方案

若 .NET SDK 版本不大于6或使用 .NET Framework，但能满足手动接入条件，可以选择手动接入上报。本文仅展示最基本的手动埋点方式，OpenTelemetry 社区提供了更多灵活的手动埋点方式，具体使用方法可参考 OpenTelemetry 社区提供的 [.NET 手动接入文档](#)。

步骤一：安装手动接入所需的 OpenTelemetry 相关依赖

1. 安装手动接入所需的 OpenTelemetry 相关依赖。

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
dotnet add package OpenTelemetry.Extensions.Hosting
```

2. 如果是基于 ASP.NET Core 的应用，还需要引入 [OpenTelemetry.Instrumentation.AspNetCore](#) 依赖。

```
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
```

步骤二：初始化 OpenTelemetry SDK

1. 创建一个 `tracerProvider` 并确保添加以下代码在您的程序开始处。

```
var serviceName = "<service-name>"; // <service-name> 替换成自定义服务名
using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddSource(serviceName)
    .SetResourceBuilder(
        ResourceBuilder.CreateDefault().AddService(serviceName)
    ).AddAttributes(new Dictionary<string, object>
    {
        ["token"] = "<token>", // <token> 替换成前置步骤中获得的 Token
        ["host.name"] = "<host>" // <host> 替换成自定义主机名
    })
    .AddOtlpExporter(opt =>
    {
```

```
opt.Endpoint = new Uri("<endpoint>"); // <endpoint> 替换成前置步骤中获得的接入点信息
opt.Protocol = OtlpExportProtocol.Grpc;
})
.Build();
```

步骤三：修改应用代码，使用 ActivitySource 创建 Activity

1. 在应用任意需要手动接入的地方，创建一个 ActivitySource 用于创建 Activity 。
其用途相当于可观测领域的 Tracer 。

```
var MyActivitySource = new ActivitySource(serviceName);
```

2. 当 ActivitySource 创建好后，创建一个 Activity 。
其用途相当于可观测领域的 Span 。

```
using var activity = MyActivitySource.StartActivity("SayHello");
// 添加 Activity Tag
activity?.SetTag("key", "3.1415");
// 为 Activity 添加一个 Event
activity?.AddEvent(new("something happened"));
```

步骤四：运行程序

构建并运行应用。

```
dotnet build
dotnet run
```

接入验证

运行 .NET 应用，在有正常流量的情况下，应用性能监控 > 应用监控 > 应用列表 中将展示接入的应用，应用性能监控 > 应用监控 > 应用详情 > 实例监控中将展示接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

接入其他语言编写的应用

应用性能监控 APM 遵循 OpenTelemetry 协议标准，理论上支持接入所有语言编写的应用。用户可以从开源社区获取对应的接入方案，将监控数据上报到应用性能监控 APM 服务端，基于云平台以及云 API 实现分布式链路追踪以及应用性能管理。

接入步骤

选择接入方案

根据不同的编程语言，以及在应用中引入的框架与类库，从 OpenTelemetry 开源社区获取对应的接入方案，详情请参考 [OpenTelemetry 社区 API & SDK 列表](#)。对于不同的语言，OpenTelemetry 社区提供的接入方案存在差异，请确保选择的接入方案与编程语言和框架的版本兼容。

获取接入点和 Token

1. 在左侧菜单栏中选择应用性能监控，单击应用列表 > 接入应用。
2. 在右侧弹出的数据接入抽屉框中，单击任意一种语言，选择您所要接入的地域以及业务系统。
3. 选择接入协议类型为 OpenTelemetry。
4. 获取接入点和 Token。

修改接入配置

基于从社区获取的接入方案，修改如下配置项：

- 接入点：在 OpenTelemetry 接入方案中，接入点通常用 endpoint 字段表达，代表 APM 服务端提供的上报地址，需要替换为您从控制台获取的接入点。
- 应用名：在 OpenTelemetry 接入方案中，应用名通常用 service.name 字段表达。多个使用相同应用名接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- Token：作为 Resource 的属性传入，对应的 key 为 token。需要替换为您从控制台获取的 Token。
- 实例名：作为 Resource 的属性传入，对应的 key 为 host.name。对于每一个接入的应用实例，实例名是唯一标识，通常情况下可以设置为应用实例的 IP 地址。部分接入方案可以自动获取 IP 地址作为实例名，您可以根据实际情况决定是否主动填写实例名。

关于 OpenTelemetry 标准中的 Resource，请参考 [Resource 介绍](#)。以 OpenTelemetry-Python 自动接入方案为例，修改接入配置后的启动脚本为：

```
opentelemetry-instrument \
--traces_exporter otlp_proto_grpc \
--metrics_exporter none \
--service_name myService \
```

```
--resource_attributes token=myToken,host.name=192.168.0.10 \  
--exporter_otlp_endpoint https://pl-demo.ap-guangzhou.apm.tencentcs.com:4317/ \  
python3 app.py
```

接入应用

基于社区开源方案的指引，完成接入工作。对于非自动接入方案，以及自动接入方案不能覆盖的框架与组件，可能还需要额外修改相关业务代码进行手动埋点。

接入验证

启动应用后，在有正常流量的情况下，控制台应用性能监控 > 应用列表中将展示接入的应用。点击应用名称/ID 进入应用详情页，再选择实例监控，即可看到接入的应用实例。由于可观测数据的处理存在一定延时，如果接入后在控制台没有查询到应用或实例，请等待30秒左右。

操作指南

资源管理

新建业务系统

业务系统用于分类管理您的应用，您可以根据不同业务系统设置不同的存储时长、上报限额等。本文将为您介绍如何新建业务系统。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 资源管理。
2. 进入资源管理页，单击新建，进入创建业务系统弹框。
3. 根据下表配置系统信息。

配置项	说明
业务系统名称	自定义业务系统名称。
存储时长	链路数据存储时长将决定您的存储成本，默认7天，支持按需调整
业务系统简介	可以简单描述业务系统用途等。

4. 完成后，单击确定即可。

应用监控

应用列表

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 应用监控。
2. 进入应用列表页面。

模块说明

切换业务系统

您可以在页面左上方切换业务系统查看数据。

系统概览

系统概览展示所选地域的所有服务和组件的核心性能指标及调用依赖关系。您可以查看已接入应用的健康情况、实例个数、吞吐量、平均响应时间、平均错误率或平均错误数、Apdex 以及各指标的变化率。

每个指标数据都提供了昨日同比数据，方便您更好的监控应用性能上的波动变化。您还可以单击应用名称，下钻到应用监控页面，查看该应用更为详细的监控数据。

接入应用

若您需要接入新的服务或者添加更多服务，可以参见下列步骤添加应用。

1. 在概览页上方，单击接入应用。
2. 在接入指南页，选择接入语言、接入方式、上报方式，并根据控制台指引完成上报。

排序条件

可根据平均响应时间、平均吞吐量、平均错误率、Apdex、应用状态进行排序，帮助聚焦高错误率，慢响应的重点服务。通过修改看板左上角的排序标准选项，支持根据不同关注维度，按照平均响应时间、平均吞吐量、平均错误率、Apdex 和健康状态排序。例如选择平均响应时间，系统看板将按照平均响应时间排序，响应时间越慢排序越靠前。

切换各端应用数据

支持自由切换客户端和服务端应用数据，方便您进行各端数据隔离分析。

- 服务端：被调用方，即提供服务方，服务端接收客户端请求并提供服务。
- 客户端：主调方，即请求方，客户端发送的请求交给服务端处理。

指标说明

指标名称	说明
应用状态	根据响应时间和错误率评估，您的应用是否健康状态。 - 健康：根据响应时间和错误率评估，您的应用处于健康状态。 - 警示：您的应用当前平均响应时间超出满意阈值，但平均错误率未超出警戒阈值。 - 异常：您的应用平均错误率超出警戒阈值。平均响应时间满意阈值默认 500ms，平均错误率警戒阈值默认5%。
平均响应时间	应用所有对外提供服务接口的平均响应时间；下方百分比展示昨日同比变化量。
平均吞吐量	1分钟粒度下平均请求数。
平均错误率	根据所选服务调用关系中，所有接口的平均错误率，时间粒度1分钟；单击曲线上数据点，可下钻到相应时间内的典型请求。
Apdex	根据所选服务调用关系中，所有接口响应时间指标为基准，计算的用户满意度指标，时间粒度为1分钟。

应用详情

本文将为您介绍如何查看应用详情，了解应用拓扑、请求数、响应时间、错误数、吞吐量等信息。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 应用列表。
2. 单击具体的应用名称，进入应用详情页面。

模块说明

性能概览

性能概览模块为您展示了所服务端或客户端、某时间段内应用平均响应时间、平均吞吐量、平均错误率以及 Apdex 的变化趋势。通过单击每个图表右上角的时钟图标，您可以自定义30天内任意日期的数据与当前时间段进行对比分析。

应用拓扑

应用拓扑当前以筛选服务为中心，展示上下游局部依赖拓扑。将鼠标悬浮在代表应用的节点上，您可以看到对应应用平均吞吐量，响应时间，错误率。

应用性能监控使用不同的拓扑图标颜色进行标识，绿色图表示应用健康、橙色图标表示应用有延时情况、红色表示该业务出现异常情况。

JVM 监控

该模块用于展示重要的 JVM 指标变化趋势，包括 GC (Garbage Collection) 平均/最大次数、CPU 利用率、Heap 空间、NoHeap 空间、Heap 空间细化和 JVM 线程数等。

接口监控

该模块展示应用接口性能指标。您还可以单击接口名查看该接口的更多详情。

数据库调用分析

该模块展示应用访问数据库的情况，由类型、数据库名称、数据库地址组成的三元组，可以确定唯一的数据库。您还可以单击查看概览和查看 SQL 查看更多数据库调用情况。

SQL 分析

该模块展示应用访问数据库时每一种调用语句的执行情况。您还可以单击查看调用查看相关调用链的执行情况。

实例监控

该模块展示每一个应用实例的性能指标。

链路追踪

该模块展示该应用相关调用链路，包括 Trace 和 Span 信息。您还可以单击 Trace ID 查看链路详情。

错误监控

该模块展示该应用相关异常信息，包括对每种异常类型的统计。

接口监控

接口监控展示客户端调用、服务端调用和本地调用中的接口、链路上游和链路下游的接口调用情况。包括请求次数、平均响应时间、错误率、吞吐量等接口调用关键指标。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 应用监控 > 接口监控。
2. 进入接口监控页面，即可查看相关模块信息。

模块说明

接口总览

在接口监控页面，用户在左侧选择接口，右侧会展示出该接口对应的接口分析和异常分析，可在顶部选择调用者进一步查看该调用者的详情。

接口分析

接口分析可以选择接口对应的服务，查看该服务的总吞吐量和 TOP5 调用者、平均响应时间（以及99pct，90pct，50pct）、错误率和错误代码分解。

指标名称	说明
TOP5 调用者	查看调用所选应用接口频率最高的5个上游应用/组件
错误代码分解	查看当前接口返回的错误代码分布趋势

说明：

- 99pct：升序排列后排在99%位置的数据。
- 90pct：升序排列后排在90%位置的数据。
- 50pct：升序排列后排在50%位置的数据。

异常统计

除基本指标外，异常分析模块帮您智能筛出了当前应用平均响应时间和错误率最高的 TOP5 接口。您可以将鼠标移动到曲线上方单击查看详情查看选中时间点向前回溯15分钟里，途径该接口的调用链，一键下钻，完成故障排查。

上下游分析

您可以在子窗口切换上下游分析菜单，分析上下游调用情况，快速排查性能瓶颈。

错误监控

本文将为您介绍如何查看应用异常情况，进行错误分析。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 应用监控 > 错误监控。
2. 进入错误监控页面，即可查看相关模块信息。

模块说明

错误列表

在错误监控页面左侧所选时间内的所有错误，包括服务错误类型、接口、错误发生次数等信息。

错误趋势

右侧图表会显示出错误趋势所筛选时间粒度下错误的发生次数。单击曲线上的数据点可进行下钻分析错误情况。

错误分析

单击相关应用，在错误监控页面右侧将会展示该错误的错误次数时序曲线，以及昨日同时间段对比曲线。

说明：

您可以单击图表上方的对比线选择，添加一个月内特定日期的同比曲线。还可以单击曲线上的任意数据点，查看相关请求列表。在列表中单击请求详情，可以下钻到调用链路详情中分析错误。

数据库概览

数据库概览页展示当前数据库调用总体情况，包括调用拓扑、数据库响应时间、吞吐量、TOP5 慢调用、TOP5 调用者和相关异常列表等。

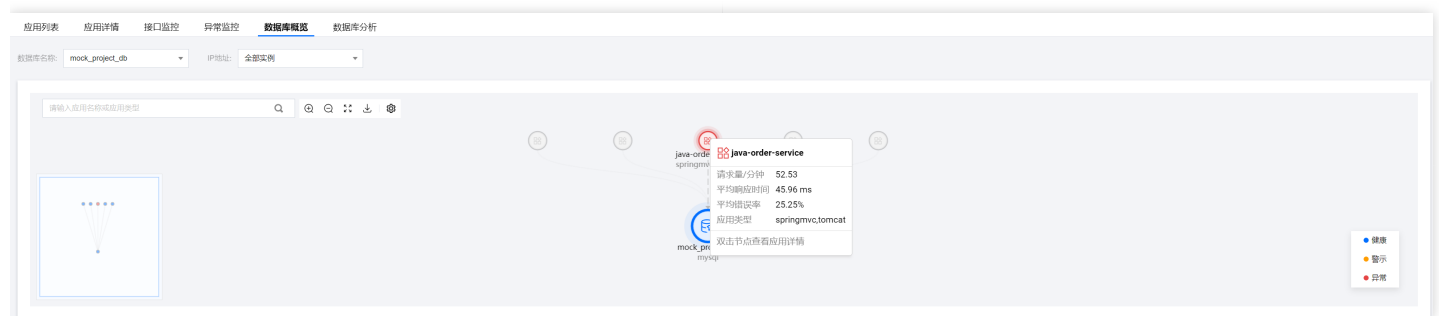
操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 数据库调用 > 数据库概览。
2. 进入数据库概览页面，即可查看相关模块信息。

模块说明

数据库调用拓扑

支持以拓扑形式展示数据库调用关系，方便您查看应用与数据库之间的调用情况。
拓扑图中红色节点为异常，您可以双击节点查看应用详情。



健康情况

- ****健康****：根据响应时间和错误率评估，您的应用处于健康状态。
- ****警示****：您的应用当前平均响应时间超出满意阈值，但平均错误率未超出警戒阈值。
- ****异常****：您的应用平均错误率超出警戒阈值。
 - 平均响应时间满意阈值默认500ms
 - 平均错误率警戒阈值默认5%

异常统计

支持展示数据库相关异常列表信息，包括异常类型、调用者、异常次数和最后发生时间；您可以单击查看详情查看数据库具体异常情况，或单击调用链路查看链路详情。

指标说明

指标名称	说明
吞吐量	当前数据库的平均吞吐量
平均响应时间	所选数据库及实例所有调用的平均响应时间，时间粒度1分钟
TOP5 调用者	查看调用所选应用接口频率最高的5个上游应用/组件
Top5 慢调用	展示 Top5 调用较慢的语句

数据库分析

本文主要介绍数据库调用检索分析相关能力和操作流程。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 数据库调用 > 数据库分析。
2. 进入数据库分析页面，即可查看相关模块信息。

模块说明

SQL 监控列表

该页面左侧 SQL 列表为您展示了所选时间范围内对应的应用和数据库之间所有 SQL 调用，同时展示了每个 SQL 的调用次数、耗时、错误率以及昨日同比变化量。您可以通过列表上方的搜索栏直接定位目标 SQL，也可以通过 SQL 语句搜索对应的 SQL。

SQL 分析

该页面右侧 SQL 分析模块，展示当前选中 SQL 一段时间内调用次数、响应时间的变化趋势。在趋势图下方，展示当前数据库异常统计列表信息，包括异常类型、调用者、异常次数、开始发生时间和最后发生时间；您可以单击查看详情数据库具体异常情况，或单击调用链路查看链路详情。

异常分析

您可以切换页面右侧的子窗口菜单，进入异常分析页面。该页面展示 SQL 异常列表，包括服务异常类型、接口、异常发生次数等信息。

相关调用链

您可以切换页面右侧的子窗口菜单，进入相关调用链页面。查询应用下相关调用情况，分析数据库异常原因。

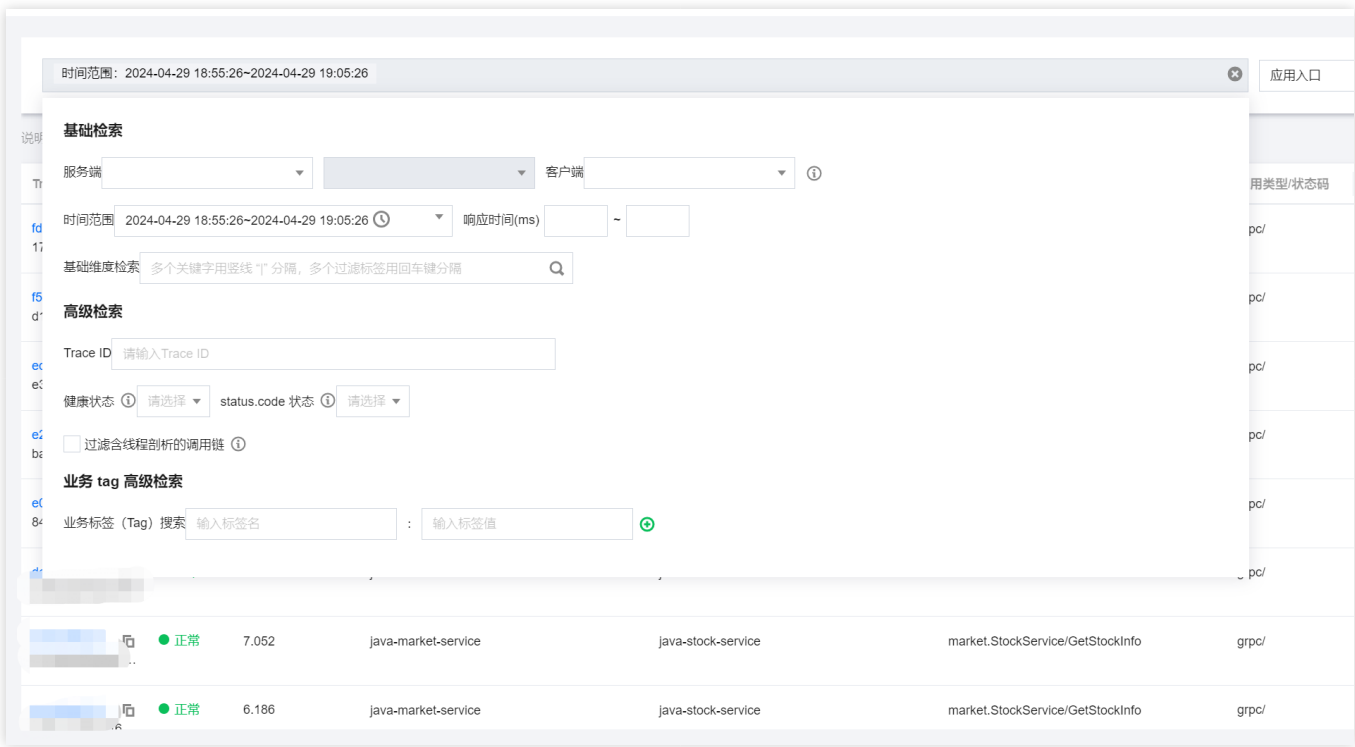
调用查询

调用查询

本文主要介绍接口调用检索分析相关能力和操作流程。使用该功能，您可以通过多种维度查询调用链信息，并通过相关 Trace ID 进一步查询调用链路详情。

操作步骤

1. 在左侧菜单栏中选择应用性能监控 > 调用查询。
2. 进入调用查询页面。



调用链检查

基础检索

基础检索提供服务端、客户端、时间范围、响应时间等基础维度检索功能。您可以通过在基础检索模块中输入信息来查询对应的接口调用情况。

高级检索

高级检索提供 Trace ID、健康状态、包含线程剖析等高级检索功能。您可以通过在高级模块中输入信息来查询对应的接口调用情况。

业务标签 (Tag)

用户可根据业务类型在上报数据时自定义标签，例如订单标签，购物车标签等。若用户订单出现异常，您可以快速查询到订单调用情况，包括响应时间、执行结果、服务状态等。

例如在上报 PHP 应用数据时自定义标签，您可以通过输入标签键、标签值 key:value 来搜索到该标签所绑定的业务。

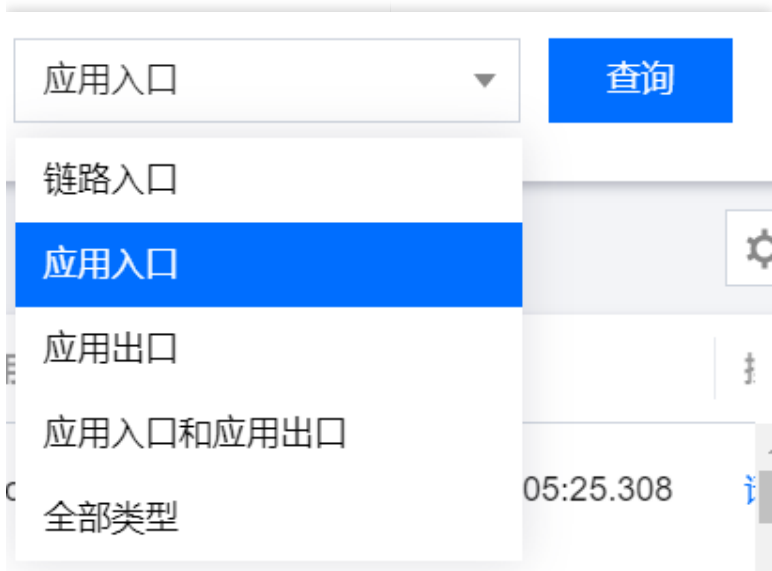
```
$span->tag('key', $value);
```

业务 tag 高级检索

业务标签 (Tag) 搜索 : 

Span 类型

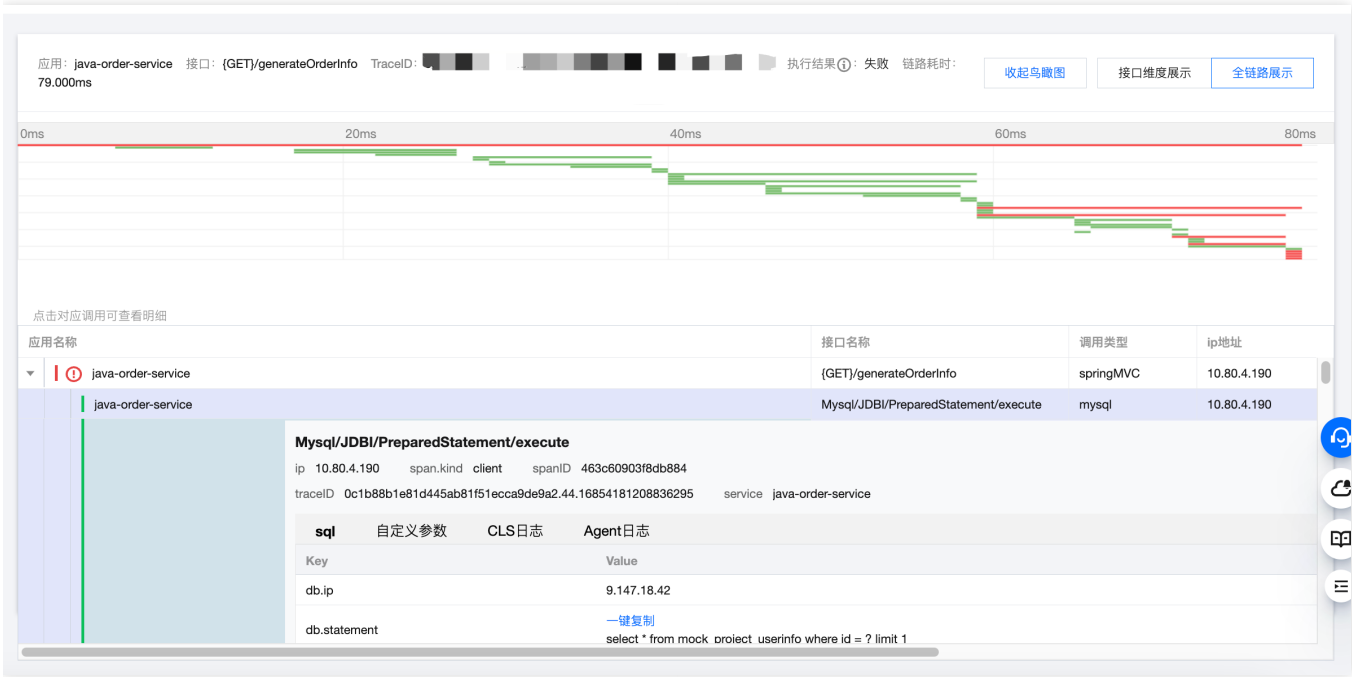
通过 Span 类型下拉过滤框，可以根据 Span 的类型检索调用情况。



- 链路入口：整条链路的第一个 Span，相当于根 Span。
- 应用入口：链路进入某个应用的第一个 Span，包含 Server 或 Consumer 类型的 Span。
- 应用出口：链路离开某个应用的最后一个 Span，包含 Client 或 Producer 类型的 Span。

调用链路详情

单击相应接口操作列的请求详情或 TraceID/SpanID，即可进入调用链路分析页面查看链路各阶段耗时以及全链路上报信息，包含调用链路健康状况、耗时等情况。



访问管理

概述

如果您使用到了应用性能监控，该服务由不同的人管理，但都共享您的云账号密钥，将存在以下问题：

- 您的密钥由多人共享，泄密风险高。
- 您无法限制其它人的访问权限，易产生误操作造成安全风险。

此时，您就可以通过子用户实现不同的人员管理不同的服务，来规避以上的问题。默认情况下，子用户无使用应用性能监控权限。因此，我们需要创建策略来允许子用户使用他们所需要资源的权限。

简介

授权方式

子用户默认没有应用性能监控任何权限。需要主账号授予子用户相关权限，子用户才能正常访问应用性能监控 APM。

授予 APM 预设策略

应用性能监控 APM 默认创建了预设策略 QcloudAPMFullAccess（应用性能监控（APM）全读写访问权限）和 QcloudAPMReadOnlyFullAccess（应用性能监控（APM）只读访问权限），您可以通过搜索策略名称快速进行预设策略授权，更多信息请参考 [策略授予](#)。

在大数多使用场景下，只需要对子用户授予 APM 预设策略，就可以让子用户正常访问应用性能监控 APM。

授予自定义策略

如果需要根据具体的资源和操作进行精细化授权，请参考 [自定义策略](#) 进行操作。

策略语法

概述

访问策略可用于授予访问应用性能监控相关的权限。访问策略使用基于 JSON 的访问策略语言。您可以通过访问策略语言授权指定委托人（principal）对指定的应用性能监控资源执行指定的操作。

策略语法

CAM 策略：

```
{
  "version": "2.0",
  "statement": [
    {
      "effect": "effect",
      "action": ["action"],
      "resource": ["resource"],
      "condition": {"key": {"value"}}
    }
  ]
}
```

元素用法

- 版本 version 是必填项，目前仅允许值为"2.0"。
- 语句 statement 是用来描述一条或多条权限的详细信息。该元素包括 effect、action、resource、condition 等多个其他元素的权限或权限集合。一条策略有且仅有一个 statement 元素。
 - 影响 effect 描述声明产生的结果是“允许”还是“显式拒绝”。包括 allow (允许)和 deny (显式拒绝)两种情况。该元素是必填项。
 - 操作 action 用来描述允许或拒绝的操作。操作可以是 API（以 name 前缀描述）或者功能集（一组特定的 API，以 permid 前缀描述）。该元素是必填项。
 - 资源 resource 描述授权的具体数据。资源是用六段式描述。每款产品的资源定义详情会有所区别。有关如何指定资源的信息，请参阅您编写的资源声明所对应的产品文档。该元素是必填项。
 - 生效条件 condition 描述策略生效的约束条件。条件包括操作符、操作键和操作值组成。条件值可包括时间、IP 地址等信息。有些服务允许您在条件中指定其他值。该元素是非必填项。

指定效力

如果没有显式授予（允许）对资源的访问权限，则隐式拒绝访问。同时，也可以显式拒绝（deny）对资源的访问，这样可确保用户无法访问该资源，即使有其他策略授予了访问权限的情况下也无法访问。下面是指定允许效力的示例：

```
"effect": "allow"
```

指定操作

应用性能监控定义了可在策略中指定一类控制台的操作，指定的操作按照操作性质分为读取部分接口 `apm:Describe\*` 和全部接口 `apm:\*` 。

指定允许操作的示例如下：

```
"action": [
  "name/apm:Describe*"
]
```

指定资源

资源（resource）元素描述一个或多个操作对象，如应用性能监控资源等。所有资源均可采用下述的六段式描述方式。

```
qcs:project_id:service_type:region:account:resource
```

参数说明如下：

参数	描述	是否必选
qcs	是 qcloud service 的简称，表示是云服务	是
project_id	描述项目信息，仅为了兼容 CAM 早期逻辑，一般不填	否
service_type	产品简称，这里为 apm	是
region	描述地域信息	是
account	描述资源拥有者的主账号信息，即主账号的 ID，表示为 <code>uin/\${OwnerUin}</code> ，如 <code>uin/1000000000001</code>	是
resource	描述具体资源详情，前缀为 instance	是

下面是应用性能监控的六段式示例：

```
"resource":["qcs::apm:ap-City A:uin/1250000000:apm/apm-btzsrI123"]
```

实际案例

基于资源 ID，分配指定资源的读写权限，主账号 ID 为 1250000000：

示例：为子用户分配业务系统（ID：apm-btzsr1123）修改权限。

```
{
  "version": "2.0",
  "statement": [
    {
      "effect": "allow",
      "action": [
        "apm:ModifyApmInstance"
      ],
      "resource": [
        "qcs::apm:ap-City A:uin/1250000000:apm-instance/apm-btzsr1123"
      ]
    }
  ]
}
```

支持资源级授权的 API 列表

API 操作名	API 描述
DescribeApmAgent	获取 Apm agent
DescribeApmInstances	查询 Apm 业务系统
DescribeApmRegions	获取 APM 地域
DescribeGeneralSpanList	Span 通用查询接口
DescribeInstanceBriefs	业务系统简介
DescribeMetricLineData	曲线指标数据拉取
DescribeMetricRecords	通用列表接口
DescribePAASGeneralSpanList	查询 Span 通用接口
DescribePAASMetricLineData	查询指标曲线数据
DescribePAASMetricPointData	查询指标点数据
DescribePAASTagValues	查询维度信息
DescribePAASTopology	查询拓扑数据

API 操作名	API 描述
DescribeServiceNodes	服务列表接口
DescribeServiceOverview	APM 系统概览接口
CreateApmInstance	创建 APM 业务系统
CreatePAASInstance	创建 PAAS APM 业务系统
DeletePAASInstance	删除 APM 业务系统
ModifyApmInstance	修改 APM 业务系统
TerminateApmInstance	销毁 APM 业务系统

策略授予

子用户默认没有应用性能监控任何权限。需要主账号授予子用户相关权限，子用户才能正常访问应用性能监控 APM。

操作前提

使用主账号或拥有 QcloudCamFullAccess 权限的子用户登录云控制台，并创建子用户。

APM 预设策略

应用性能监控 APM 默认创建了预设策略 QcloudAPMFullAccess（应用性能监控（APM）全读写访问权限）和 QcloudAPMReadOnlyFullAccess（应用性能监控（APM）只读访问权限），您可以通过搜索策略名称快速进行默认策略授权。

自定义策略

1. 使用主账号或拥有 QcloudCamFullAccess 权限的子用户进入用户中心 > 策略管理。
2. 单击新建自定义策略，根据不同的需求，选择对应的创建策略方式。

应用性能监控支持资源级授权和按标签授权两种自定义策略方式：

资源级授权

您可以通过策略语法或默认策略给子账号单个资源的管理权限，详细请参见 [策略语法](#)。

策略关联

1. 进入策略管理页，在策略名称搜索框中输入对应的策略名称。
2. 选择只读访问或全读写访问权限，在操作列中单击关联用户/组/角色。
3. 在弹框中勾选对应的用户，单击确定即可。

告警服务

新建告警

本文将为您介绍如何为应用性能关键指标设置告警，在指标发生异常时及时通知您。

操作步骤

1. 登录云产品监控控制台。
2. 在左侧菜单栏中单击告警策略，进入策略管理页面。
3. 单击新建告警策略，配置告警策略，配置说明如下：

配置类型	配置项	说明
基本信息	策略名称	自定义策略名称。
配置告警规则	告警产品	选择 APM 。
	对象类型	选择对象类型。
	触发条件	支持满足任意条件、满足所有条件或满足复合条件。
告警通知	通知策略	选择通知策略

4. 配置完以上信息后单击完成，即成功创建告警策略。在指标发生异常时，将会通过您配置的告警渠道发送告警通知。

查看告警

本文将为您介绍如何查看应用性能监控告警历史。

查看告警历史

1. 登录云产品监控控制台。
2. 在左侧菜单栏中单击告警查询。
3. 通过上方进行条件筛选，告警产品选择“APM”。选择完后单击确定即可进行查询。
4. 您还可以单击上方的时间段进行筛选，筛选需要查看告警历史的时间范围。

系统配置

业务系统配置

本文将为您介绍业务系统的相关配置信息。

操作步骤

1. 在左侧菜单栏选择应用性能监控 > 系统配置，进入业务系统配置页面。
2. 在页面顶部选择地域和目标业务系统。
3. 针对如下的分组栏目，修改业务系统配置。
 - 基本信息：该分组支持业务系统基本信息配置，包括以下配置项：

配置项	说明
业务系统名称	自定义业务系统名称。
链路存储时长	支持选择1天、3天、7天、15天、30天链路数据存储时长，试用期间默认存储时长为1天。存储时长越长，收费越高。超过存储时长的链路将不会展示在应用性能监控控制台。
业务系统简介	可以简单描述业务系统用途等。
 - 自定义配置：

支持自定义展示列配置，每添加一个自定义展示列以后，在链路追踪等列表视图中，将使用额外的一列展示对应的 Span 属性。

应用配置

操作步骤

1. 在左侧菜单栏选择应用性能监控 > 系统配置，进入应用配置页面。

2. 在下拉框选择目标业务系统和目标应用。

3. 针对如下的分组栏目，修改应用配置。
- 指标配置：用于自定义指标数据的处理规则以及展现形式，包括如下配置项：

配置项	说明
接口过滤	通过接口名过滤不需要展示的接口，例如： <code>operationName(?.?),operation()</code> 。多个规则之间用英文逗号分割。
错误类型过滤	被过滤的错误类型不会在错误分析中被统计。请按照正则表达式输入需要过滤的错误类型，多个错误类型之间请使用英文逗号分隔。
HTTP 状态码过滤	被过滤的 HTTP 状态码不会在错误分析中被统计。默认情况下，HTTP 状态码在大于 400被认为是错误，多个 HTTP 状态码之间请使用英文逗号分隔。
URL 收敛	开启后，系统将根据自定义的收敛规则正则，以及收敛阈值，将同一类接口进行聚合展示。
收敛阈值	收敛阈值是指要进行 URL 收敛的最低数量条件，例如：当阈值为100时，则符合规则正则表达式的接口数量达到100时进行收敛。
收敛规则正则	以正则表达式自定义 URL 收敛规则。例如：当前半部分都为 <code>/service/demo/</code> 的接口数超过100个时，将该类接口进行聚合展示，则收敛阈值需填写为100，收敛规则正则则为 <code>/service/demo/(.*)</code> 。
排除规则正则	排除规则正则表示具有该特性的 URL 将不纳入 URL 收敛。例如：前半部分为 <code>service/demo/example</code> 不需要进行 URL 收敛，则表达式为 <code>/service/demo/example/(.*)</code> 。
- 线程剖析：仅适用于增加版 OpenTelemetry Java 探针，在修改探针配置后，系统会将配置项推送给探针，应用不需要重启。

配置项	说明
启线程剖析	开启线程剖析以后，对于耗时达到阈值的调用，会针对此接口自动生成线程剖析快照。在链路详情页面，如果一个接口包含了线程剖析快照，会展示线程剖析按钮**，点击后可以查看本地方法栈的执行详情。在调用链路**页面，用户可以在高级搜索中，指定过滤包含线程剖析快照的调用链。

配置项	说明
慢调用监听触发阈值	默认2000ms，耗时高于该阈值的调用，才有可能自动生成线程剖析快照。

参考信息

OpenTelemetry 方案支持的组件和框架

增强版 OpenTelemetry Java 探针支持的 Java 版本和框架

支持的 Java 版本

增强版 OpenTelemetry Java 探针支持 Java 8 及以上的 Java 版本，推荐使用 Java 8、Java 11、Java 17等 LTS 版本。

支持的依赖与框架

增强版 OpenTelemetry Java 探针对常用的开源组件和框架进行了自动埋点，包括 Servlet、Spring Web MVC、Apache Dubbo 等，具体清单请参考 [OpenTelemetry Java 探针支持的依赖与框架列表](#)。

此外，增强版 OpenTelemetry Java 探针还对 XXL-JOB 2.2.0+ 以及 Apache Thrift 0.9.3+ 进行了埋点增强。

支持的应用服务器

增强版 OpenTelemetry Java 探针支持常用的应用服务器，具体清单请参考 [OpenTelemetry Java 探针支持的应用服务器列表](#)。

OpenTelemetry-Go 方案支持的组件和框架

支持的 Go 版本

Go 的官方支持版本，目前为1.21和1.22，对于更低的版本，理论上可以接入，但社区不保持完整的兼容性，具体信息请参考社区 [兼容说明](#)。

支持的依赖与框架

OpenTelemetry-Python 方案对于 Go 系的常用依赖库和框架，提供了基于 SDK 的上报方案，请参考社区提供的 [完整列表](#)。

OpenTelemetry-Python 方案支持的组件和框架

支持的 Python 版本

3.6+

支持的依赖与框架

OpenTelemetry-Python 方案对于 Python 系的常用依赖库和框架，包括 Flask、Django、FastAPI、MySQL

Connector 等，提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考社区提供的 [完整列表](#)。

OpenTelemetry-JavaScript 方案支持的组件和框架

支持的 Node.js 版本

Node.JS v14+

支持的依赖与框架

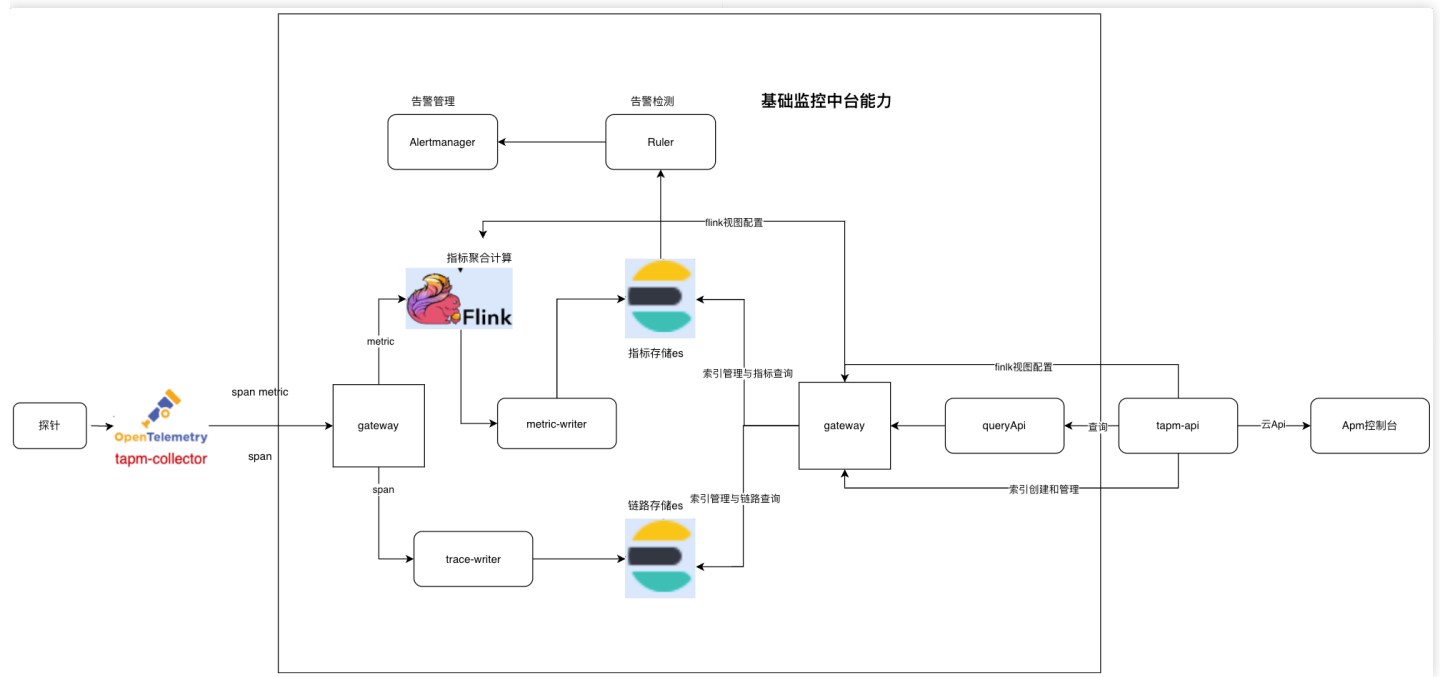
OpenTelemetry-JavaScript 方案对于 Node.js 系的常用依赖库和框架提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考社区提供的 [完整列表](#)。

运维管理指南

产品架构

应用性能监控 (APM) 集成基础监控中台能力，为用户提供应用性能监控一站式解决方案。

应用性能监控 (APM) 的详细架构设计如下图：



APM 的核心组件：

- tapm-collector：统一上报接入点，适配多种开源协议，接收用户上报数据，进行数据处理，生成链路和指标数据。

- tapm-api：控制台服务，负责对链路和指标数据进行可视化处理。

基础监控中台能力：

- gateway：统一网关，接收数据。
- flink：流计算，聚合指标，生成不同指标视图。
- trace-writer：链路写入组件，屏蔽底层存储。
- metric-writer：指标写入组件，屏蔽底层存储。
- ruler：告警检测。
- alertmanager：告警管理。
- query-api：查询组件，屏蔽底层存储。

日常巡检

1. 运维巡检总览

标号	巡检项	说明
1	日常巡检项列表	日常巡检项概览
2	检查应用性能监控控制台	查看应用性能监控是否正常
3	检查Kafka状态	查看kafka是否正常
4	检查Elasticsearch状态	查看Elasticsearch是否正常
5	检查Flink状态	查看Flink是否正常
6	检查其他pod状态	查看其他模块是否正常
7	检查其他模块	查看依赖是否正常

2. 检查监控控制台

登录租户端控制台，查看应用性能整体的功能是否正常

- 前提条件:
已获取租户端账号。
- 操作步骤:
登录租户端应用性能监控控制台，查看应用详情等列表的功能。
- 检查结果:
正常：页面无报错，列表可以正常展示，曲线正常展示。
异常：页面有报错，或无监控数据。
- 异常处理:
按照《运维手册》中故障处理章节进行排查与处理。
- 运维经验
无。

3. 检查kafka状态

登录zk和kafka节点，检查kafka状态是否正常。

- 前提条件:
已获取zk节点和kafka节点登录信息。
- 操作步骤:
进入zk部署的bin目录下，运行zkCli.sh，运行 ls kafka/kafka/brokers/ids。
- 检查结果:

如果返回[1, 2, 3]或[0,1,2] 则表示有3台kafka节点在运行中，如果缺少，有可能节点有异常，但不一定影响服务，但仍需尽快修复。

如果上述命令返回少于3个，说明有节点异常，执行get kafka/kafka/brokers/ids/0，可以得到节点ip信息（id的编号从上边的返回结果取），将返回的id依次执行，跟kafka部署的ip对照，既可知道异常的kafka节点信息。

- 异常处理:

登录异常的kafka节点，ps -ef|grep kafka查看进程是否存在，如不存在，进入bin目录，执行sh kafka-server-start.sh则重新拉起。

- 运维经验:

Kafka节点部署机器应加上crontab，如进程挂掉则自动拉起。

4. 检查Elasticsearch状态

检查Elasticsearch状态是否正常。

- 操作步骤:

curls1.barad:9200/_cat/health?v，查看返回的status是否为green。

curls1.barad:5100/search/clusters，查看返回结果中当前集群的磁盘使用量。

- 检查结果:

正常：集群状态为green，当前集群的disk_used / capacity.disk <= 0.8。

异常：集群状态为red或yellow，当前集群的disk_used /capacity.disk > 0.8。

- 异常处理:

集群状态为red和yellow可能是es的node故障，需要登录node查看进程是否挂掉，如果挂掉需要执行./bin/elasticsearch重新拉起。

集群disk_used /capacity.disk > 0.8，表明集群容量不足，需要进行扩容。

- 运维经验:

将Elasticsearch节点的磁盘使用率监控起来。

5. 检查其他pod状态

登录tcs-master主机，查看apm相关pod是否正常。

- 前提条件:

已获取tcs-master主机登录信息。

- 操作步骤:

登录tce-master主机。

执行命令'kg tcloud-apm，查看pod是否是running状态。

- 检查结果:

正常：apm相关pod都在运行中。

异常：apm相关的pod有非运行中的状态。

- 异常处理:

查看异常pod的日志和事件信息，根据事件信息进行处理。

- 运维经验:

查看如果遇到异常pod，可以尝试重启pod。

6. 其他模块巡检

其他模块均可依赖tcs的健康检查进行巡检。

故障处理

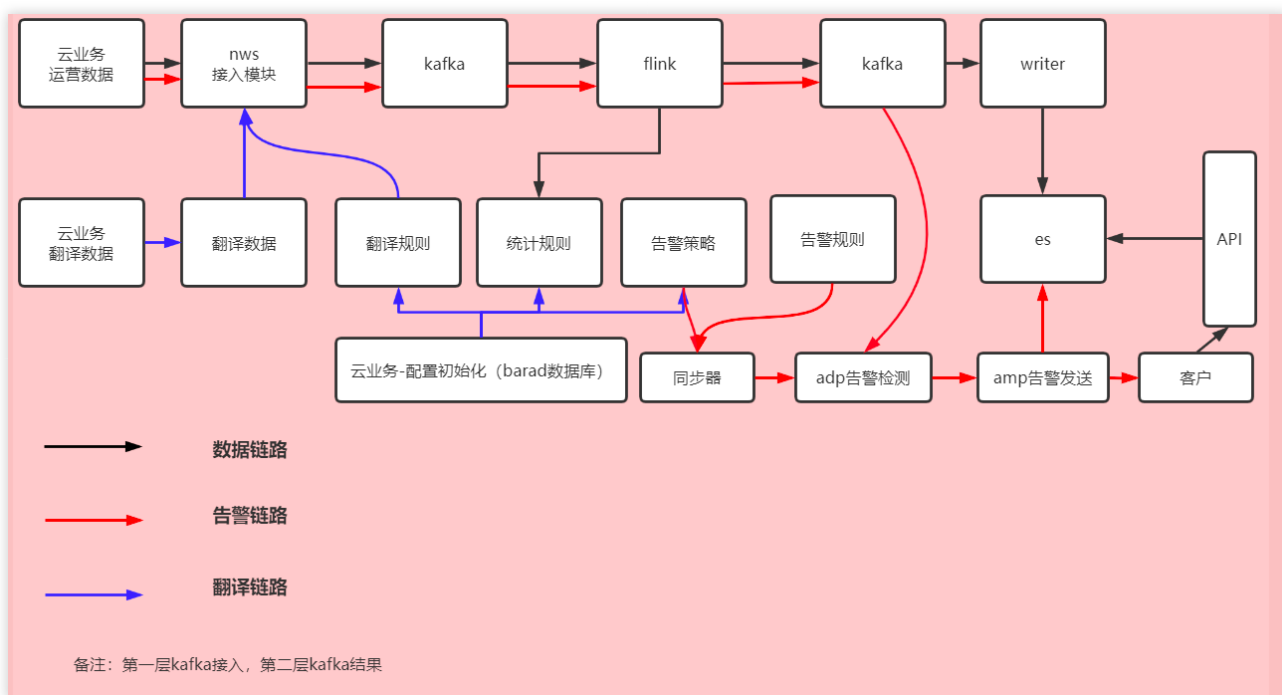
1. 故障处理思路

i. Apm指标数据链路:

Apm通过collector组件，将指标数据上报到barad-nws，barad-nws对维度信息翻译以后缓存到kafka，kafka数据流入到flink，flink再将数据计算结果流入kafka，kafka再将数据写入writer，writer写入es，客户在前端查看数据，调用api，api从es拉取数据展示。

ii. 告警链路:

创建了告警策略之后，同步器会去同步告警策略和告警规则到adp告警检测，adp告警检测会从第二层kafka获取监控数据进行检测是否符合告警规则，符合则到amp告警发送，amp告警发送发送消息给到客户，并且将告警历史等信息存到es保存。



iii. 排查思路:

- 分析具体场景，缩小范围。
- 针对具体case分析。
- 全链路分析，逐一排查。

2. 故障处理CASE

i. 指标无监控数据

■ 排查思路:

根据数据链路去查每一个组件。

■ 故障现象:

租户端控制台所有产品都看不到监控数据。

■ 故障定位及处理:

- 检查ES集群是否正常部署：

- 在pod内curl es1.barad:9200/_cat/health?v, 如果集群状态为green则正常。
- b. 查看存储表是否正常初始化：
 - curl es1.barad:9200/_metrics, 正常应返回viewName index的集合, 若报无权限或无该index, 则创建集群的初始化参数有问题, 开启了鉴权, 或集群类型创建的不是ctsdb, 需将集群销毁重新创建, 具体可参考ctsdb oss部署文档。
 - 查看建表的时间戳格式是否正常: curl es1.barad:9200/_metric/cvm_device-60, 若返回的format为epoch_second则为正常, 若为epoch_mills则为异常。如异常需要将该表删掉重建 (如管控刚刚拉起, 存量数据无需保留的话, 可将所以metric删掉后重新执行初始化脚本)。
- c. 如以上都正常, 且集群本身运营有一段时间后突然没数据, 可查看es存储是否被写满,
 - curl es1.barad:9200/cat/allocation?v的disk.indices可查看当前已使用的node空间,
 - curl es1.barad:5100/_search/clusters可以看到对应集群预先分配的node磁盘空间。如已达到分配容量, 则需要对ES进行扩容。

ii. 拓扑图无数据

- 排查思路:

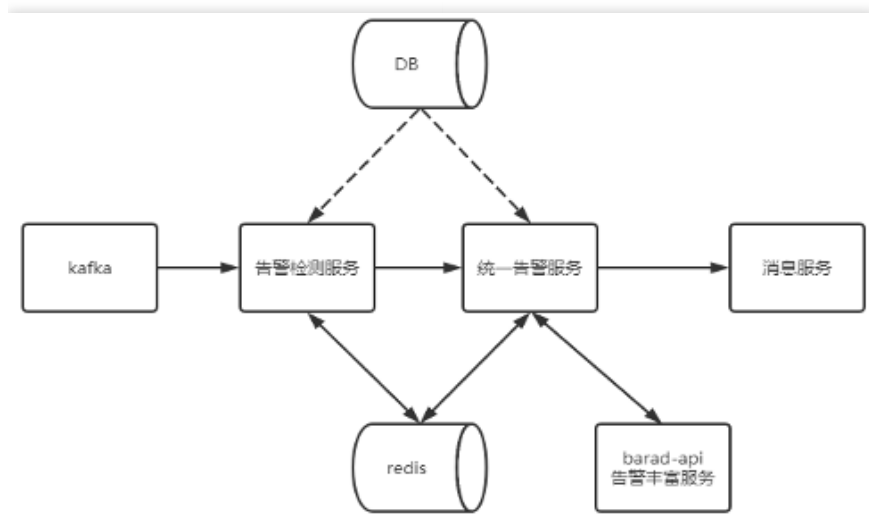
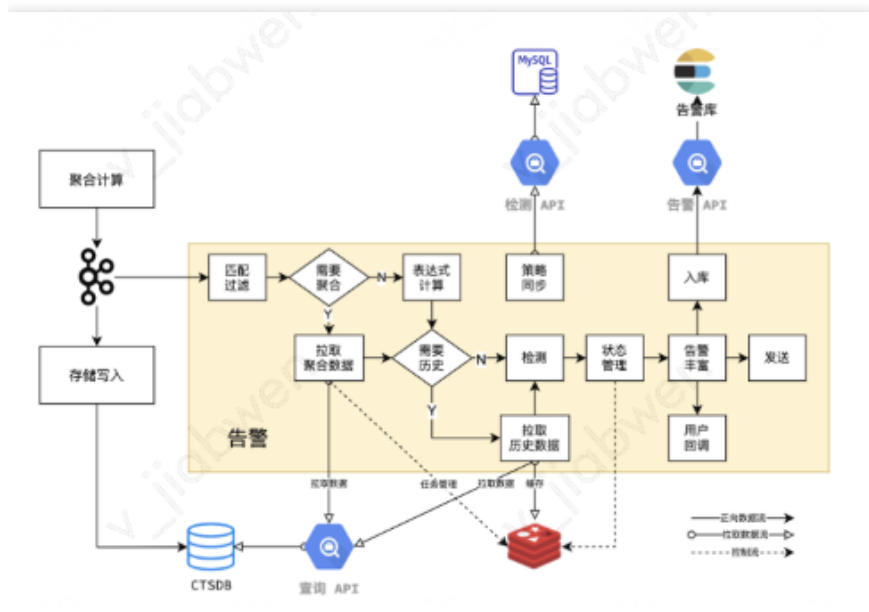
这种情况可以肯定的是taw-streaming组件那个的链路存在问题,
- 故障现象:

拓扑图无数据, 无法生成关系图。
- 故障定位:
 - a. 确认上报方collector组件是否有写入kafka数据中。
 - b. 如确实有写入, 检查streaming的任务是否有丢弃数据, 过滤。
 - c. 如没有, 检查writer-trace是否正常消费写入es中。
- 运维经验:
 - d. 集群的初始化操作一定要做。
 - e. 配置的管理切记不要搞错。

iii. 业务无告警

- 告警架构:

监控中台将告警能力分为检测和告警



- 告警组件:

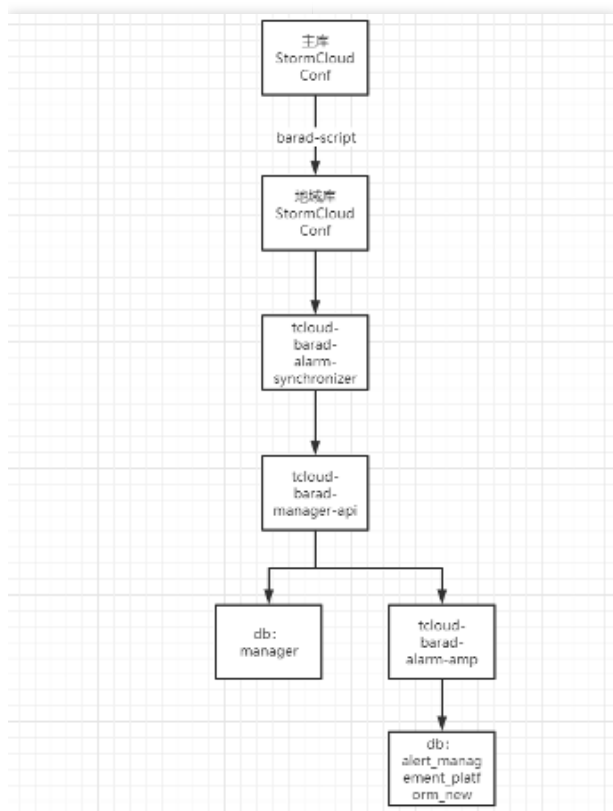
同步组件tcloud-barad-alarm-synchronizer

告警发送tcloud-barad-alarm-amp

告警检测tcloud-barad-alarm-detector

库 StormCloudConf , alert_management_platform_new , manager。

- 数据同步逻辑:



1. 查看是否有监控数据，若无监控数据，则先按照《无监控数据》进行定位。
2. 若有监控数据但无告警，则需要查看告警组件的日志，根据日志报错进行排查。

监控介绍

1. 应急预案Pod 自监控

当前APM所有模块和组件均已经容器化，Kubernetes集群使用livenessprobe（存活探针）来确定何时重启容器，使用readiness probe（就绪探针）来确定容器是否已经就绪可以接受流量。在tcs集群中存活探针和就绪探针的定义如下：

livenessProbe:

```
exec:
  command:
    - bash
    - /tce/healthchk.sh
failureThreshold: 3
initialDelaySeconds: 30
periodSeconds: 5
successThreshold: 1
timeoutSeconds: 5
```

readinessProbe:

```
exec:
  command:
    - bash
    - /tce/healthchk.sh
failureThreshold: 5
initialDelaySeconds: 30
periodSeconds: 5
successThreshold: 1
timeoutSeconds: 3
```

两个探针使用的都是脚本/tce/healthchk.sh进行检查，在不同镜像此脚本会根据镜像中的组件进行定义，如果组件运行正常则以0退出，否则以非0退出。借助存活探针和就绪探针kubernetes集群保证业务请求的正常处理。

2. 应用自检查

在容器内部，组件通过crontab任务对进程定期检查，当发现进程没有启动时会自动把进程启动起来，避免kubernetes对pod重启带来的流量损失。如在barad-api容器中，配置有如下定时任务：

```
[root@tcloud-barad-api-7b5dc6d55b-6hd65/]# crontab -l|grep monitor.sh
*/1 * * * */usr/local/services/ConfigClient-1.0/admin/monitor.sh >/usr/local/services/ConfigClient-1.0/log/crontab.log 2>&1 &
*/1 * * * */usr/local/services/barad-php-5-6-29-1.0/admin/monitor.sh >/usr/local/services/barad-php-5-6-29-1.0/log/crontab.log 2>&1 &
*/3 * * * */usr/local/services/barad-php-5-6-29-1.0/tools/php-master-monitor.sh &
*/1 * * * */usr/local/services/tnginx_1_0_0-1.0/admin/monitor.sh > /usr/local/services/tnginx_1_0_
```

```
0-1.0/log/crontab.log2>&1 &  
[root@tcloud-barad-api-7b5dc6d55b-6hd65/]#
```

3. APM自监控指标

对APM自身的运行状态的监控是通过云生监控系统prometheus进行的。存储到TSDB中，同时检查交警策略，如果满足条件则通过alertManager模块进行短信或邮件告警。其中云监控的监控数据包括如下内容：

标号	巡检项	说明
1	机器的负载	CPU，内存
2	Flink反压指标	背压通常产生于这样一种场景：当一个系统接收数据的速率高于它在一个瞬时脉冲内能处理的数据。
3	Kafka数据积压量	Kafka topic中消息积压数量
4	NWS接收量	NWS接收到的信息数量统计
5	ES的状态监控	集群监控，节点监控，索引监控

应急预案

1. 机房掉电处理预案

- 故障影响:
机房可能由于供电、空调、机柜故障或交换机故障等因素导致承载云监控业务的服务器全局发生掉电，导致业务中断。
- 恢复步骤:
 - a. 使用批量脚本或命令，确认所有机器已开机并恢复网络连通性。
 - b. 确认各机器上运行的进程是否已正常拉起。
 - c. 确认yarn服务已启动，可观察对应ui界面观察。

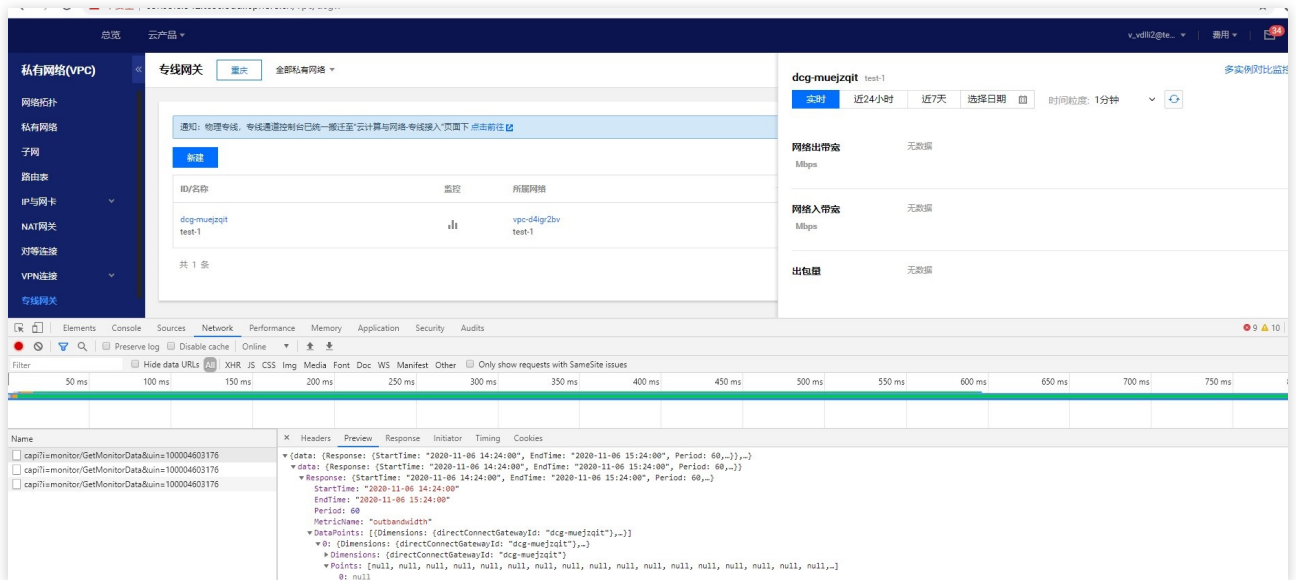
2. Kafka服务整体集群故障处理预案

- 前置检查:
检查kafka状态是都正常
 - a. 进入zk部署的bin目录下，运行zkCli.sh，运行 ls/kafka/kafka/okers/ids
 - b. 如果上述命令返回少于3个，说明有节点异常，执行get /kafka/kafka/brokers/ids/0，可以得到节点
 - c. ip信息（id的编号从上边的返回结果取），将返回的id依次执行，跟kafka部署的ip对照，既可知道异常的kafka节点信息。
- 故障影响:
Kafka服务作为监控数据的缓存节点，是流式计算的重要一环，正常情况下是集群模式工作。集群崩溃后，数据无法写入缓存，导致监控数据中断，isd.barad数据无法显示。监控告警无法正常发出，严重影响运营。
- 恢复步骤:
 - d. 尽快恢复启动所有kafka服务器，并启动对应进程。
 - e. 如有机器无法启动，需要寻找一台新的存储类型机器代替。
 - f. 新机器的kafka=id要和旧机器一致，kafka-id信息可以从zk中获取。
 - g. 异常的kafka进程恢复后，会自动进入kafka数据恢复。
 - h. 登录kafka机器，执行kafka-topic-describe命令，获取实时状态。

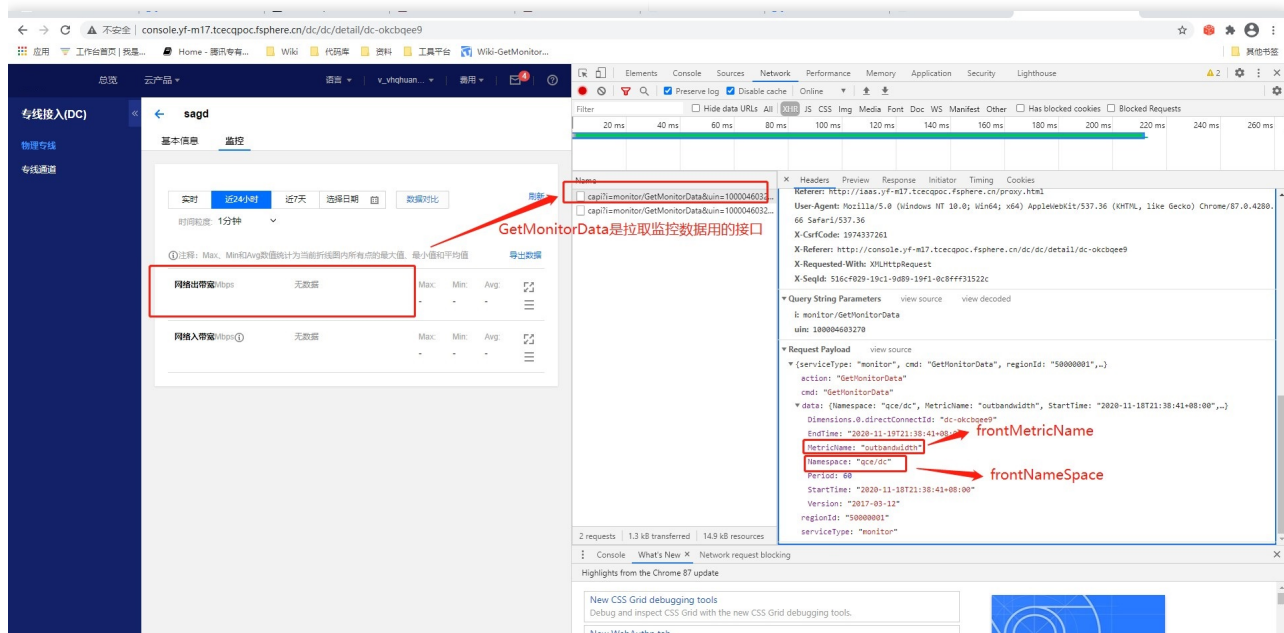
最佳实践

1. 物理专线和专线通道、专线网关监控无数据

i. 现象：



- 前端观察无数据，F12获取入参的 frontNameSpace 和 frontMetricName



- 将前端抓包得到的 frontNameSpace 和 frontMetricName 拿到后端抓包查看是否有数据上报，

2. 手动抓包:

进云监控数据库地域库（一般mysql22003可以直接进入地域库，我们做了别名。如果客户环境mysql22003进不去，以客

户的实际情况为准)

例如：select *from StormCloudConf.cApiMetric where frontNameSpace = 'qce/dc';

```
mysql [(none)]> select * from StormCloudConf.cApiMetric where frontNameSpace = 'qce/dc';
```

metricId	statType	frontNameSpace	namespace	vmName	metric	statTypeMin	statTypeMax	statTypeMin	frontMetricUnit	frontMetricUnit	unitMap	unitMapId	apiInterfaceTranslateId	meaning_zh	meaning_en	frontNameSpaceV3	frontMetricNameV3	frontStatus	statTypeDay										
189	qce/dc	inbandwidth	qce/dc	dc_detail	inbandwidth	max	last			bit/s		bit/s		0.00001			40		网络带宽平均每秒吞吐量		NULL		NULL		0		NULL		N
190	qce/dc	outbandwidth	qce/dc	dc_detail	outbandwidth	max	last			bit/s		bit/s		0.00001			40		网络带宽平均每秒吞吐量		NULL		NULL		0		NULL		N

2 rows in set (0.00 sec)

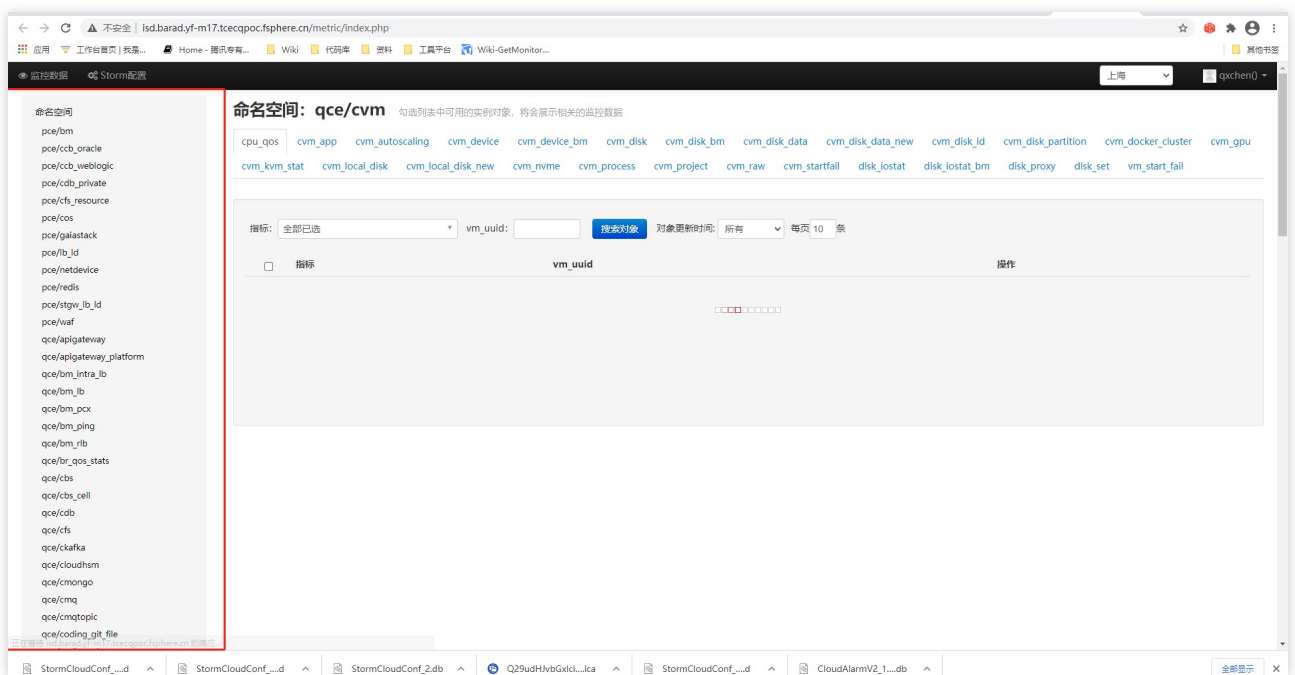
后端对应的命名空间和指标

查到映射的后端命名空间和后端的指标，再去barad-nwspod里进行抓包例如：tcpdump -i any -s0 -nn -A 'dst port 80'| grep'qce/dc'|grep 'inbandwidth'

3. 接入配置是否有问题确认：前端报错网络错误，查看数据库没有相关 namespace 的接入配置，跟归档的.db文件比较,下面几种方式基本可以确认是否接入配置问题，检查发现 qce/dc qce/dcxqce/dcg 没有接入。

- 观察barad-isd提供的监控配置中心页面左侧栏是否缺少相关的命名空间。

示例地址：<http://isd.barad.Example.com> (具体域名可在tcs master获取，kubectl get ingress -A | grepispd.barad)



- 例如在StormCloudConf主库执行select * fromcShowProductConfig where namespace = '{\$namespace}'; 查询无记录。
- select * from cApiMetric where namespace = '{\$namespace}';无数据或者数据对不上，特例是没有接Api指标的。
- select * from cObservationMetric where namespace = '{\$namespace}' 无数据或者数据对不上

4. 解决方案：

重新导入对的接入配置。有的客户还需要更新barad-isd镜像包。

更新镜像后（或者barad-isd镜像的接入配置qce/dc qce/dcxqce/dcg 是好的）

进入barad-isd pod ,

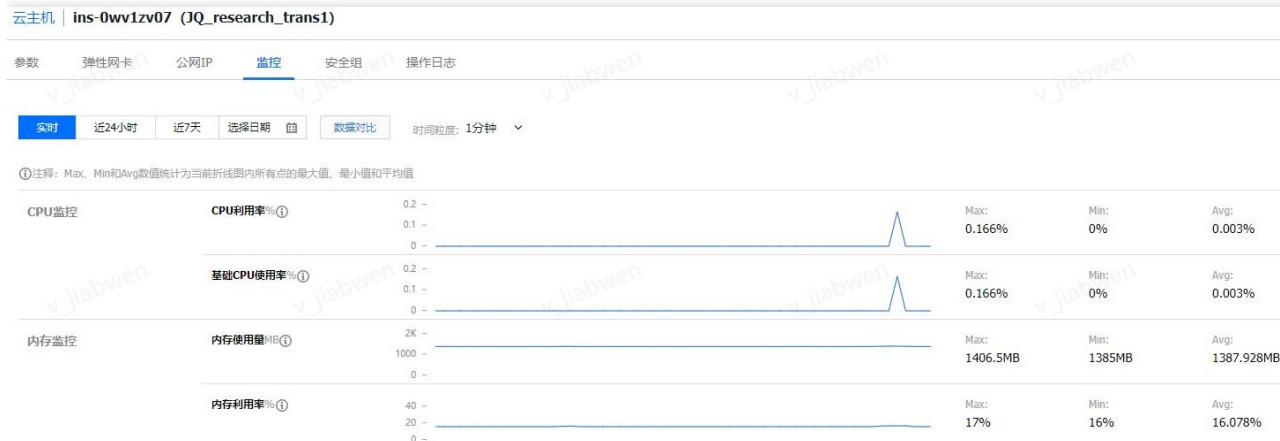
```
cd /data/release/product.pkg/tools , shsync_product.sh qce/dc qce/dcx qce/dcg
```

执行后没有报错，再去页面查看监控数据展示是否恢复。

5. 七天监控无数据，实时有数据

i. 现象如下：

实时有数据，7天数据异常：



2. 7天数据异常原因：

云监控的聚合规则，如果配置了聚合方式，则优先使用该规则聚合数据，反之则依次寻找下一粒度，对天级别数据和小时级别数据，优先5分钟，其次1分钟。

3. 解决方案：

在从库执行下面的SQL，删除天级聚合方式，使用默认的5分钟来计算3600的数据，注意操作前备份好数据库。

```
DELETE a from cAssociationMiningRule as a, cDimensionView as b, cDimensionGroup as c where
a.outputDimGroupId=c.dimGroupId and a.outputPeriod>300 and b.dimensionGroup=c.dimensionGroup and
b.namespace=c.namespace and b.namespacein("qcce/nat_gw_bm","qce/bm_lb","qce/cm_q","qce/cvm","qce/
vpc_region_conn","qce/cmongo","qce/bm_intra_lb","qce/cmqttopic","qce/internal_loadbalance_rs","qce/
docker","qce/lb","tnm/vsr_vpnconn","qce/nat_qos_stats","qce/cbs","qce/dcg","qce/cos","qce/pcx","qqce/dc","qce/
dcx","qce/lb_total","qce/cdp","qce/cdn","qce/ckafka","qce/bm_pcx","qce/scf","qce/ov_cdn","qce/
apigateway","qce/loadbalance","qce/qaap","qce/nat_rs_bm","qce/internal_scf","qce/rlb","qce/cdb","qce/
```

```
redis","qce/phy_host","qce/bm_rlb","qcce/sqlserver","qce/tdsql","qce/ctsdb","qce/iotmq","qce/csg","qce/vpc_net_detect","qce/ces","qce/emr","pce/bm","pce/cos","pce/netdevice","pce/stgw_lb_id","pce/lb_id","pce/cfs_resource","qce/vpc_gateway","qce/vpc_nat","qce/vpc_jnsgw","pce/cdb_privaaate","pce/redis","pce/ccb_weblogic");
```

节点重启

1. 产品部署架构

介绍产品各个节点的角色、作用、组件。

模块/容器	节点类型	组件介绍
tcloud-apm-api	容器	查询链路数据和监控数据
tcloud-apm-collector	容器	将用户上报的数据进行收集，转换成apm的协议写入到Kafka中
tcloud-apm-offline-quota	离线计费对账	离线计费对账
tcloud-apm-writer	容器	写入基础监控数据
tcloud-apm-writer-trace	容器	写入链路数据
yunapi-apm		Apm的云图配置

2. 一般情况下pod重启

一般情况下直接重启组件所在的pod，k8s会自动完成负载流量转移，delete之后会自动拉起并回切到集群。

- 获取barad pod列表：kubectlget pod --all-namespaces -o wide |grep tcloud-apm
- Delete Pod：kubectl delete pod -n tce \${pod}
- 等待拉起状态为：

```
1/1      Running
```

3. 按组件重启

- 获取barad pod列表：kubectl get pod --all-namespaces-o wide |grep tcloud-apm-api
- 进入pod：kubectl exec -it -n tce \${pod}bash
- 一般情况组件所在的目录：/usr/local/services/


```
[root@tcloud-barad-aws-88f47995f-7slvh services]# ll
total 0
lrwxrwxrwx 1 root root 36 Oct 26 16:40 ConfigClient -> /usr/local/services/ConfigClient-1.0
drwxr-xr-x 1 root users 42 Nov 2 19:44 ConfigClient-1.0
drwxr-xr-x 1 root users 42 Nov 2 19:44 barad_aws-1.0
drwxr-xr-x 1 root users 42 Nov 2 19:44 custom_alarm_aws-1.0
drwxr-xr-x 2 root root 20 Nov 2 19:44 etc
drwxr-xr-x 1 root users 64 Nov 2 19:44 aws_dict_loader-1.0
```

4. sh/usr/local/services/barad_aws-1.0/admin/restart.sh all 正常：

```
[2020-11-07 17:45:55,536] INFO: start all successfully
[2020-11-07 17:45:55,536] INFO: restart all successfully
```

备份恢复

1. 数据库

- i. mysql数据库主要存储的应用资源的数据。

备份数据库，以tapm_db为例，可使用如下命令。

备份整库

```
mysqldump-h${host} -P22001 -umysql_user -p'mysql_passwd' tapm_db> tapm_db.sql
```

恢复数据库，使用上述备份文件进行恢复：

```
mysql-h${host} -P22001 -umysql_user -pmysql_passwd -D tapm_db< tapm_db.sql
```

2. 监控数据

ElasticSearch引入仓库与快照的概念实现了数据的备份与恢复。

- i. 创建仓库

```
curl-xPOST http://${ES_IP}:9200/_snapshot/my_backup_1 -d'{
  "type": "fs",
  "settings": {
    "location": "\Vdata\elasticsearch-5.1.1\data\back\my_backup_1",
    "max_snapshot_bytes_per_sec":"20mb",
    "max_restore_bytes_per_sec":"20mb",
    "compress": true
  }
}'
```

其中参数解释如下：

type：仓库的类型为共享文件系统

location:指定仓库的路径，必须为path.repo 的子目录

max_snapshot_bytes_per_sec:快照数据进入仓库时,该参数可以控制过程的限流情况,默认为每秒20M

max_restore_bytes_per_sec:从仓库恢复数据时,该参数控制过程限流情况，默认值：每秒20M

注意：如果挂在的路径为远程目录时，应该合理配置该值，不至于网络流量被占满。

compress:数据是否压缩

2. 快照索引

快照所有打开的索引

```
curl-XPUT http://${ES_IP}:9200/_snapshot/my_backup_1/snapshot_1?wait_for_completion=true
```

快照指定索引

```
curl-XPUT http://${ES_IP}:9200/_snapshot/my_backup_1/snapshot_3 -d'{
  "indices": "my_index_1,my_index_2",
  "ignore_unavailable": true,
  "include_global_state": true
}'
```

3. 恢复快照

```
curl-XPOST http://${ES_IP}:9200/_snapshot/my_backup/snapshot_1/_restore?wait_for_completion=true
```

4. 快照信息

获取单个快照的信息

```
curl-XGET http://${ES_IP}:9200/_snapshot/my_backup/snapshot_1
```

获取仓库下所有快照的信息

```
curl-XGET http://${ES_IP}:9200/_snapshot/my_backup/_all
```

参考信息

1. 配置文件列表

镜像名	模块名	配置文件名	配置文件路径
tcloud-apm-api	查询服务模块	config.yaml	/usr/local/services/tapm-api/conf
tcloud-apm-collector	数据采集模块	config.yaml	/usr/local/services/apm-collector/conf
tcloud-apm-offline-quota	计费模块	config.yaml	/usr/local/services/tapm-offline-quota/con
tcloud-apm-writer	监控数据写入模块	metric-writer.toml	/usr/local/services/metric-writer/conf
tcloud-apm-writer-trace	链路数据写入模块	metric-writer.toml	/usr/local/services/metric-writer/conf

2. 配置文件详细说明

i. 组件日志

模块	日志位置	日志作用
tcloud-apm-api	/usr/local/services/tapm-api/log	Apm查询请求的日志
tcloud-apm-collector	/usr/local/services/apm-collector/log	收集上报的日志
tcloud-apm-offline-quota	/usr/local/services/tapm-offline-quota/log	计费对账的数据
tcloud-apm-writer	/usr/local/services/metric-writer/log	kafka消费数据写入es日志。
tcloud-apm-writer-trace	//usr/local/services/metric-writer/log	kafka消费数据写入es日志
metric-writer	/usr/local/services/metric-writer/log	kafka消费数据写入es日志
oceanus-ctcw1	/data/logs/tstream_cluster_admin /data/logs/tstream_cluster_master /data/logs/tstream_taskcenter /data/logs/tstream_watchdog	任务执行的情况日志
oceanus-galileo1	/data/logs/tstream_galile /data/logs/tstream_netcenter	对外接口日志

模块	日志位置	日志作用
oceanus-servant1	/data/logs/tstream_servant	服务检测日志