腾讯云可观测平台

应用性能监控

腾讯云

文档目录

应用性能监控
应用性能监控简介
应用性能监控概述

最近更新时间：2024-06-13 16:15:51

应用性能监控（Application Performance Management，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 APM 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。

应用性能监控优势

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、自定义大盘等重要的能力，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

应用性能监控基于腾讯云在微服务领域的经验，并结合腾讯云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，同时支持按量付费以及预付费，您可以根据自身业务特点灵活选择付费模式。

功能介绍

最近更新时间：2024-06-13 14:25:01

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。继承腾讯云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。

自定义大盘

通过 Tencent Cloud APM Grafana 插件，您可以对接 APM-Grafana 数据源， 获取 APM 常用的指标并进行自定义扩展，通过 Grafana 面板实现灵活的可视化呈现。

告警管理

基于腾讯云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（电话、短信、邮件、微信、企业微信、钉钉、飞书、Slack 等）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。

应用场景

最近更新时间：2024-10-12 16:04:52

定位性能瓶颈

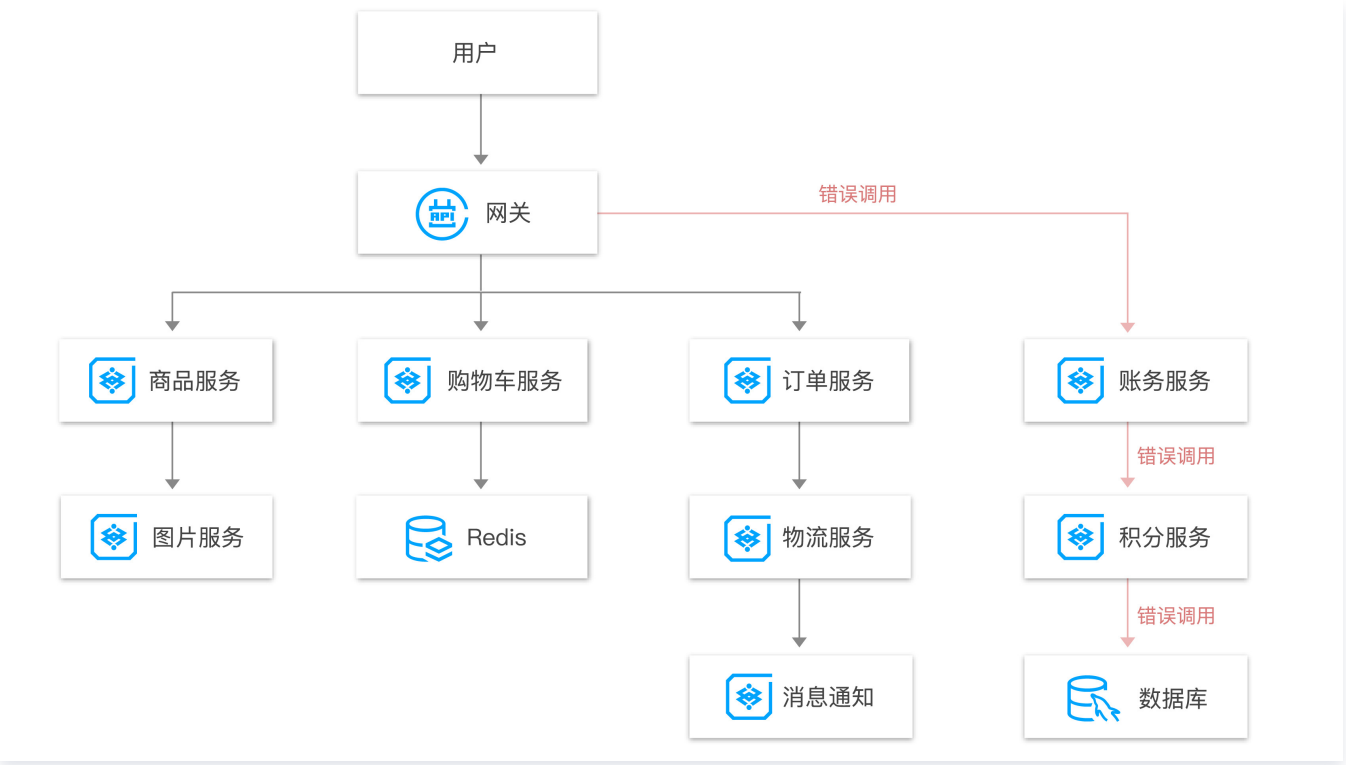
痛点

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

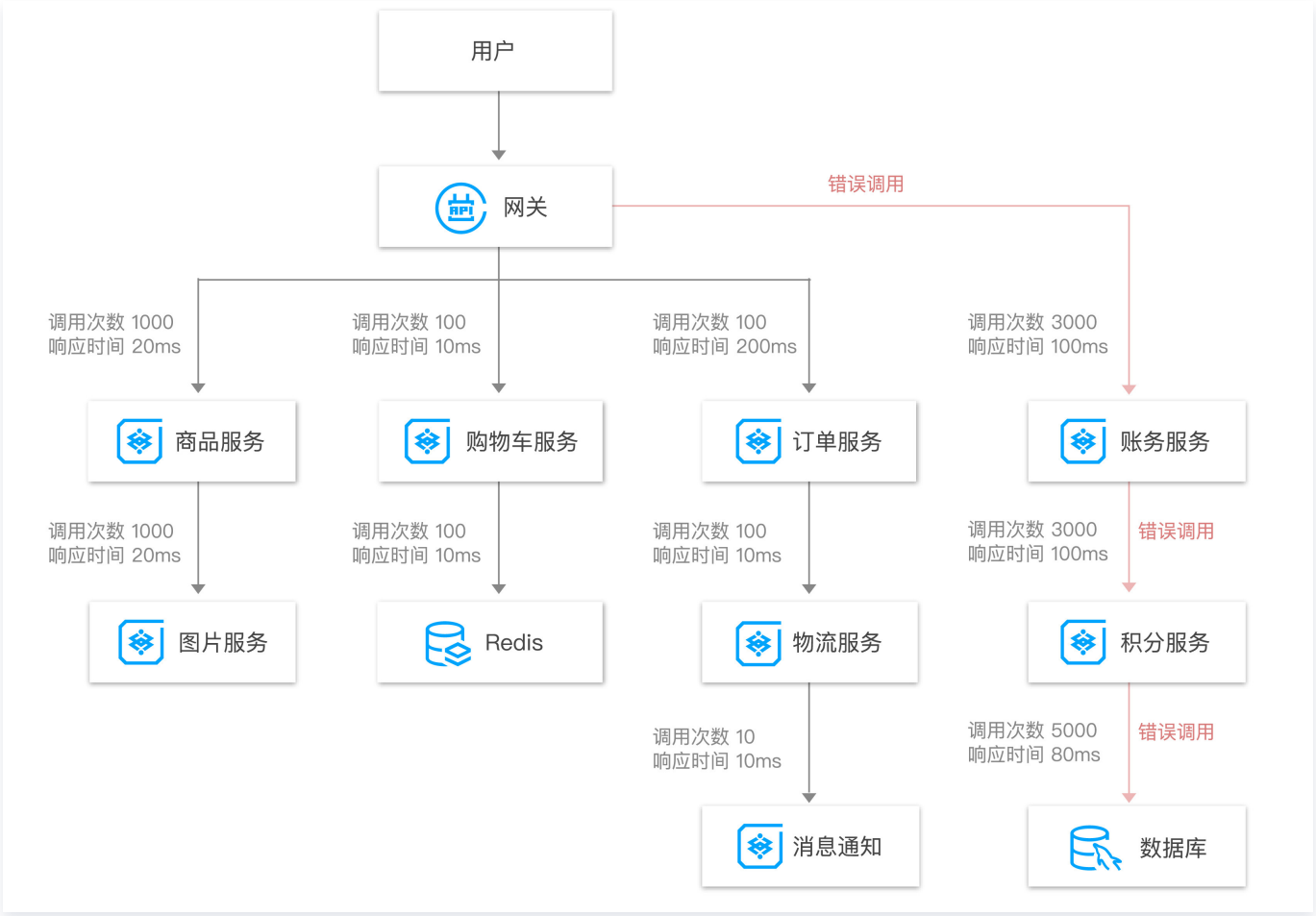
当业务量突增，各应用之间调用时间长、接口加载缓慢等问题，导致用户体验感降低。由于各应用之间的调用情况复杂，分析应用性能问题难以实现。

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。**



接入指南
应用接入概述

最近更新时间：2024-10-16 16:33:31

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。
对于部署在 Kubernetes 的应用，请优先选择 Operator 模式自动接入，节省安装探针和修改配置方面的工作量，关于 Operator 模式的更多信息请参考安装 tencent-opentelemetry-operator。

非腾讯云环境是否可以接入?

可以，只要应用和 APM 服务端之间的网络可达，就可以接入。APM 提供了公网接入点和内网接入点，对于非腾讯云环境（本地 IDC 或者其他云）的应用，可以使用公网接入点；也可以通过专线接入（Direct Connect，DC）等方式连通腾讯云 VPC，使用内网接入点，以获得更好的网络质量。

接入方式

应用类型	OpenTelemetry 方案	Skywalking 方案
Go	• 通过 OpenTelemetry-Go 接入 Go 应用（推荐） • 通过 eBPF 探针接入 Go 应用	通过 SkyWalking-Go 接入 Go 应用
Java	• K8s 环境自动接入 Java 应用（推荐） • 通过腾讯云增强版 OpenTelemetry Java 探针接入（推荐）	通过 SkyWalking 协议接入 Java 应用
Python	• K8s 环境自动接入 Python 应用（推荐） • 通过 OpenTelemetry-Python 接入 Python 应用（推荐）	通过 SkyWalking 协议接入 Python 应用
PHP	通过 OpenTelemetry-PHP 接入 PHP 应用（推荐）	-
Node.js	• K8s 环境自动接入 Node.js 应用（推荐） • 通过 OpenTelemetry-JS 方案接入 Node.js 应用（推荐）	-
.Net	• K8s 环境自动接入 .NET 应用（推荐） • 通过 OpenTelemetry-dotnet 接入 .NET 应用（推荐）	-
其他语言	接入其他语言编写的应用	-

接入 Go 应用
通过 OpenTelemetry SDK 接入 Go 应用（推荐）

最近更新时间：2025-05-26 10:22:01

> **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 接入腾讯云应用性能监控 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 。

操作步骤

步骤1：获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入应用**抽屉框中，单击 **Go 语言**。
4. 在**接入 Go 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 **OpenTelemetry**。
6. 选择您所想要的**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，您的服务需运行在腾讯云 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腾讯云 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费用。

步骤2：引入 OpenTelemetry 相关依赖，实现 SDK 初始化逻辑

> **说明：**
> 您可以参考我们的代码示例完成修改，详情请参考 tencentcloud-opentelemetry-demo-go 。

```
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"
)

var (
    tracer = otel.Tracer("otel-demo") // 这里初始化了一个全局的链路对象，用户可以自定义链路名
)

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.TraceC
ontext{}, 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3：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。

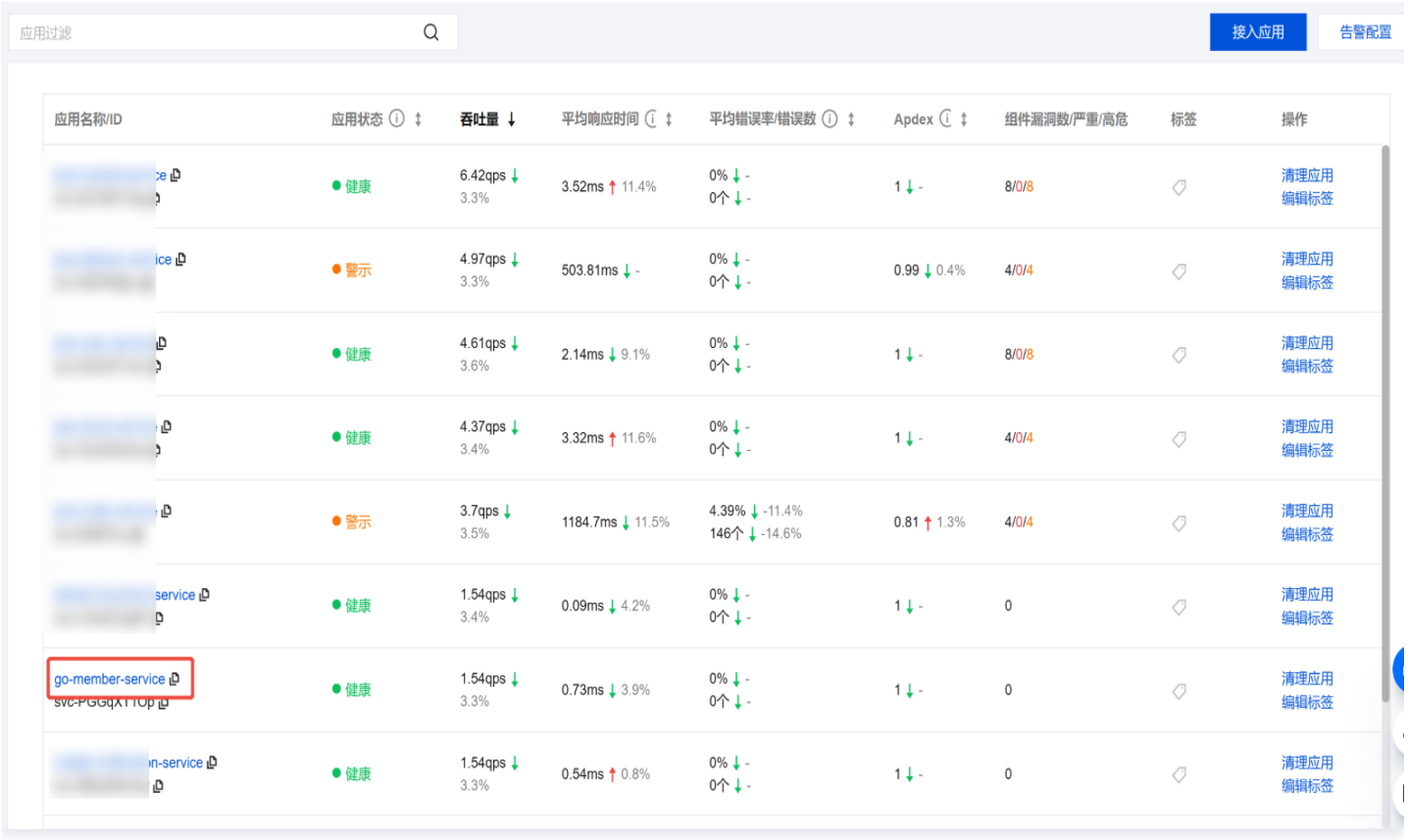
步骤4：对 HTTP 接口进行埋点增强

```
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 > 应用列表 中将展示接入的应用，单击**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。

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-
mysql"))); 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)
}
```

自定义埋点

用户可以在当前链路上下文追加一个自定义 Span，以提升链路数据的丰富度。进行自定义埋点的时候，需要通过 `tracer.WithSpanKind()` 设置 Span Kind。Span Kind 包括 `server` 、 `client` 、 `internal` 、 `consumer` 和 `producer` 五种类型，请根据具体的业务场景进行设置。本文将提供内部方法埋点，以及访问外部资源埋点的代码编写示例。

内部方法

这里演示在一个 `server` 类型 Span 里面，追加一个类型为 `internal` 类型的 Span。

```
func entryFunc(w http.ResponseWriter, r *http.Request) {
```

```
    _, span := tracer.Start(r.Context(), "entryFunc",
trace.WithSpanKind(trace.SpanKindServer)) // server span
    defer span.End()
    internalInvoke(r)
    io.WriteString(w, "ok")
}

func internalInvoke(r *http.Request) {
    // 创建一个 Internal Span
    _, span := tracer.Start(r.Context(), "internalInvoke",
trace.WithSpanKind(trace.SpanKindInternal)) // internal span
    defer span.End()

    // 业务逻辑省略
}
```

访问外部资源

访问外部资源，一般需要在当前的链路上下文中追加一个类型为 `client` 的 Span。

```
func clientInvoke(ctx context.Context) {
    _, span := tracer.Start(ctx, "child", trace.WithSpanKind(trace.SpanKindClient))
    defer span.End()
    _, err := http.Get("https://www.example.com")
    if err != nil {
        fmt.Println("err occurred when call extrnal api: ", err)
        return
    }
}
```

获取当前 Span 上下文

在代码中，可以获取当前 Span 的上下文信息，从而添加或修改 Span 属性，或者将获取到的 TraceID/SpanID 输出到日志中。

获取 TraceID/SpanID

```
func internalInvoke(ctx context.Context) {
    spanCtx := trace.SpanContextFromContext(ctx) // 获取当前 span 的上下文
    if spanCtx.HasTraceID() {
        traceID := spanCtx.TraceID()
        fmt.Println(traceID.String())
    }
    if spanCtx.HasSpanID() {
        spanID := spanCtx.SpanID()
        fmt.Println(spanID.String())
    }
    // 业务逻辑省略
}
```

设置 Span 属性

先获取 span 对象，再设置属性。

```
func gormRequest(ctx context.Context) {
    span := trace.SpanFromContext(ctx) // 获取当前的 span 对象
    if span == nil {
        fmt.Println("no active span detected")
        return
    }
    span.SetAttributes(
        attribute.String("key1", "value1"),
        attribute.Int64("key2", 123),
    )

    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)
}
```

通过 SkyWalking-Go 接入 Go 应用

最近更新时间：2024-09-20 10:24:52

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. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**数据接入**抽屉框中，单击 Go 语言。
4. 在**接入 Go 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. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 Token。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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。

自定义链路埋点

在自动埋点的基础上，通过 SkyWalking Toolkit 可以添加自定义埋点。
引入相关包：

```
package main

import (
    "github.com/apache/skywalking-go/toolkit/trace"
)
```

获取 TraceId 和 SpanId：

```
trace.GetTraceID()

trace.GetSpanID()
```

创建和销毁 Span：

```
trace.CreateLocalSpan("testAddLog")

trace.StopSpan()
```

给 Span 添加属性：

```
trace.AddLog(...string) // 添加 Log

trace.SetTag("key","value") // 添加 Tag
```

创建 Event：

```
trace.AddEvent(trace.DebugEventType, "foo")
```

更多 Tracing API 的用法，请参考 SkyWalking 官方代码 。

通过 eBPF 探针接入 Go 应用

最近更新时间：2025-03-14 15:38:32

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。
- eBPF（扩展伯克利包过滤器）允许在内核空间运行安全的用户定义代码，而不需要修改内核源代码或加载内核模块。eBPF 与 OpenTelemetry 结合，使得用户无需在代码中手动埋点就可实现完整的监控能力。
- 腾讯云 eBPF 探针基于开源社区 opentelemetry-go-instrumentation 项目二次开发，功能和特性仍在快速迭代中。

本文将介绍如何通过腾讯云 eBPF 探针接入 Go 应用。

前提条件

说明：

- eBPF 目前仅支持 Linux，内核版本需要在4.19及以上。
- 暂不支持 macOS 和 Windows 的系统（包括运行在这类宿主机中的 Linux 容器）。

Go 版本需在1.18及以上，支持的库和框架如下。

库/框架	版本
`database/sql`	`go1.12` to `go1.23.0`
`github.com/segmentio/kafka-go`	`v0.4.1` to `v0.4.47`
`google.golang.org/grpc`	`v1.14.0` to `v1.66.0`
`net/http`	`go1.12` to `go1.22.6`

注意：

基于 `net/http` 的 Web 框架，如 Gin 等都支持。

操作步骤

步骤1：获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入**抽屉框中，单击 **Go 语言**。
4. 在**接入 Go 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

步骤2：下载探针

- linux/amd64探针 下载。
- linux/arm64探针 下载。

赋予探针执行权限：

```
chmod +x otel-go-instrumentation
```

步骤3：接入并上报

1. 确认应用的运行路径

将需要接入的应用编译成二进制文件，确认应用启动时的**完整路径**。

2. 运行探针

运行探针需要有系统的 root 权限，使用以下命令接入。

```
sudo OTEL_GO_AUTO_TARGET_EXE=</path/to/executable_binary> \
OTEL_SERVICE_NAME=<serviceName> \
OTEL_EXPORTER_OTLP_PROTOCOL=grpc \
OTEL_TRACES_EXPORTER=otlp \
OTEL_EXPORTER_OTLP_ENDPOINT=<endpoint> \
OTEL_RESOURCE_ATTRIBUTES=token=<token>,host.name=<hostName> \
./otel-go-instrumentation
```

对应字段的说明如下：

- `</path/to/executable_binary>`：应用可执行文件的路径，**这里一定要是绝对路径**。
- `<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。
- `<endpoint>`：前置步骤中拿到的接入点，注意这里必须添加 `http://` 前缀。
- `<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。

下述内容以应用名为 `myService`，运行路径为 `/root`，业务系统 Token 为 `myToken`，主机名为 `192.168.0.10`，接入点以 `http://ap-guangzhou.apm.tencentcs.com:4317` 为例，完整的启动命令为：

```
sudo OTEL_GO_AUTO_TARGET_EXE=/root/myService \
OTEL_SERVICE_NAME=myService \
OTEL_EXPORTER_OTLP_PROTOCOL=grpc \
OTEL_TRACES_EXPORTER=otlp \
```

```
OTEL_EXPORTER_OTLP_ENDPOINT=http://ap-guangzhou.apm.tencentcs.com:4317 \
OTEL_RESOURCE_ATTRIBUTES=token=myToken,host.name=192.168 .0.10 \
./otel-go-instrumentation
```

了解更多接入方式，请参考 官方文档 。

3. 运行应用

运行应用，当发生接口调用时，探针会输出日志。如果应用停止运行，探针不需要停止，应用下次启动时会自动埋点。

4. 接入验证

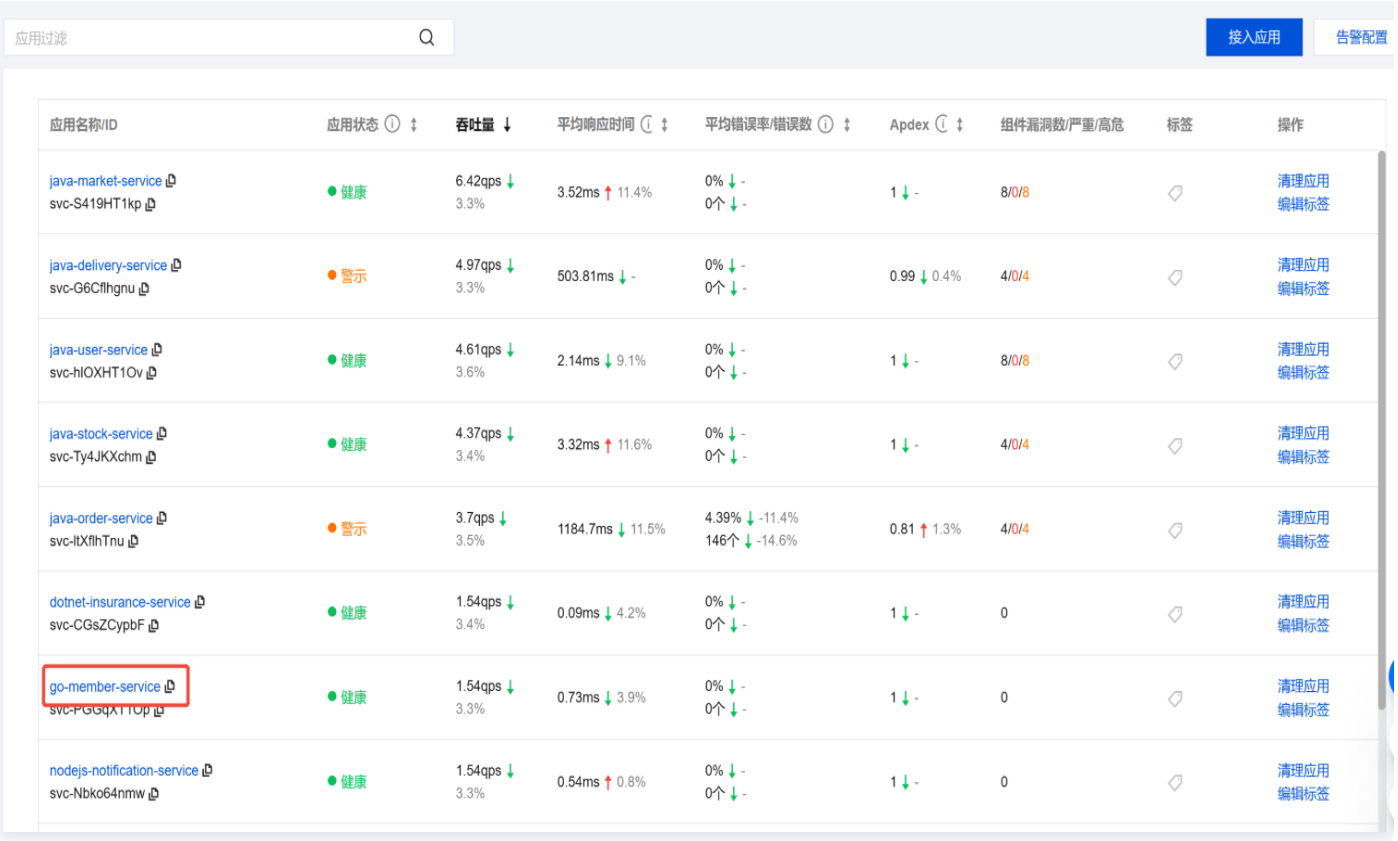
- 接入侧

  探针的日志输出中含有 `instrumentation loaded successfully` 字段表示接入成功：

```
{"level":"info","ts":1725609047.2234442,"logger":"go.opentelemetry.io/auto","caller"
:"cli/main.go:119","msg":"starting instrumentation..."}
{"level":"info","ts":1725609047.2235398,"logger":"Instrumentation.Manager","caller":
"instrumentation/manager.go:195","msg":"loading probe","name":"net/http/server"}
{"level":"info","ts":1725609047.388379,"logger":"go.opentelemetry.io/auto","caller":
"cli/main.go:115","msg":"instrumentation loaded successfully"}
```

- APM 控制台

  在发生接口调用的情况下， 应用性能监控 > 应用列表 中将展示接入的应用，单击**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。

接入错误排查

探针仅有一条日志输出：

```
{"level":"info","ts":1725609014.2038825,"logger":"go.opentelemetry.io/auto","caller":"
cli/main.go:86","msg":"building OpenTelemetry Go instrumentation
...","globalImpl":false}
```

一般是如下两种情况：

1. **应用没有启动。**
2. **应用的启动路径（** `OTEL_GO_AUTO_TARGET_EXE` **）错误。**

探针日志报错：

```
traces export: failed to exit idle mode: dns resolver: missing address
```

一般是接入点没有添加 `http://` **前缀导致。**

添加手动埋点

为了增加埋点的灵活性，eBPF 探针支持用户自定义埋点，用户可以添加与业务逻辑和内部功能相关的自定义 span 信息，示例代码如下。

```
import (
    ...
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    ...
)
...

var tracer = otel.Tracer("rolldice") // 创建 Tracer

...

func (s *Server) rollDice(w http.ResponseWriter, req *http.Request) {
    ctx, span := tracer.Start(req.Context(), "roll") // 新建一个 span
    defer span.End()

    n := s.rand.Intn(6) + 1
```

```
    _, err := s.db.ExecContext(ctx, "INSERT INTO results (roll_value) VALUES (?)", n)
    if err != nil {
        panic(err)
    }

    span.SetAttributes(attribute.Int("roll.value", n)) // 添加 span attributes

    fmt.Fprintf(w, "%v", n)
}
```

eBPF 接入代码示例

> **说明：**
> 以下将给出多组使用 eBPF 探针接入 APM 的代码示例，帮助用户更方便地把业务的链路数据上报到 APM。

使用 eBPF 探针上报数据时，几乎不需要更改已有的业务代码，但是需要一些细节，例如代码中 context 的传递。

对外提供接口的数据库操作程序

本项目的主要功能是对外提供一个 HTTP 接口，调用此接口可以操作 sqlite 数据库。基于 Go 语言的标准库 `net/http` 和 `database/sql` 实现。

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
    "os"
    _ "github.com/mattn/go-sqlite3"
    "go.uber.org/zap"
)

const (
    sqlQuery = "SELECT * FROM contacts"
    dbName   = "test.db"

    tableDefinition = `CREATE TABLE contacts (
    contact_id INTEGER PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL,
    phone TEXT NOT NULL);`

    tableInsertion = `INSERT INTO 'contacts'
    ('first_name', 'last_name', 'email', 'phone') VALUES
    ('Moshe', 'Levi', 'moshe@gmail.com', '052-1234567');`
)
```

```
type Server struct {
    db *sql.DB
}
// 初始化数据库
func CreateDb() {
    file, err := os.Create(dbName)
    if err != nil {
        panic(err)
    }
    err = file.Close()
    if err != nil {
        panic(err)
    }
}

func NewServer() *Server {
    CreateDb()
    database, err := sql.Open("sqlite3", dbName)
    if err != nil {
        panic(err)
    }
    _, err = database.Exec(tableDefinition)
    if err != nil {
        panic(err)
    }
    return &Server{
        db: database,
    }
}


func (s *Server) queryDb(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    conn, err := s.db.Conn(ctx)
    if err != nil {
        panic(err)
    }
    // 注意这里一定要传递 `ctx` 或者 `req.Context()`，使 HTTP 请求和数据库的操作记录可以保持在一
条链路中。
    // 如果在这里没有传递请求的 context，写成了 `s.db.Exec(tableInsertion)`，就会导致链路中断，
HTTP span 和 database span 出现在两条链路中！
    _, err = s.db.ExecContext(ctx, tableInsertion)
    if err != nil {
        panic(err)
    }
    rows, err := conn.QueryContext(req.Context(), sqlQuery)
    if err != nil {
        panic(err)
    }
    logger.Info("queryDb called")
    for rows.Next() {
```

```
        var id int
        var firstName string
        var lastName string
        var email string
        var phone string
        err := rows.Scan(&id, &firstName, &lastName, &email, &phone)
        if err != nil {
            panic(err)
        }
        fmt.Fprintf(w, "ID: %d, firstName: %s, lastName: %s, email: %s, phone: %s\n",
id, firstName, lastName, email, phone)
    }
}
```

注意：
请关注代码中的注释说明，一定要正确传递 context 才能正确构造 span 之间的父子关系，保证链路不会中断。

接入 Java 应用
K8s 环境自动接入 Java 应用（推荐）

最近更新时间：2025-03-14 15:38:32

对于部署在 Kubernetes 上的 Java 应用，APM 提供自动接入方案，可以实现探针自动注入，方便应用快速接入。
K8s 环境自动接入的 Java 应用将被自动注入腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-OTel Java Agent），腾讯云增强版 OpenTelemetry Java 探针基于开源社区的 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，腾讯云增强版 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。

前提条件

请参考 增强版 OpenTelemetry Java 探针支持的 Java 版本和框架，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，腾讯云增强版 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，请使用 OpenTelemetry API 进行自定义埋点。

操作步骤

步骤1：安装 Operator

在 K8s 集群安装 Operator，推荐从 APM 控制台一键安装 Operator，详情请参考 安装 tencent-opentelemetry-operator。

步骤2：在工作负载添加 annotation

以容器服务 TKE 为例，通过如下步骤可以在工作负载中添加 annotation，对于通用 K8s 集群，请通过 kubectl 等工具添加 annotation：

1. 登录 容器服务 控制台。
2. 单击**集群**，进入对应的 TKE 集群。
3. 在**工作负载**中找到需要接入 APM 的应用，单击**更多**，单击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，单击**完成**，即可以完成接入。

```
cloud.tencent.com/inject-java: "true"
cloud.tencent.com/otel-service-name: my-app  # 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。
# 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。
```

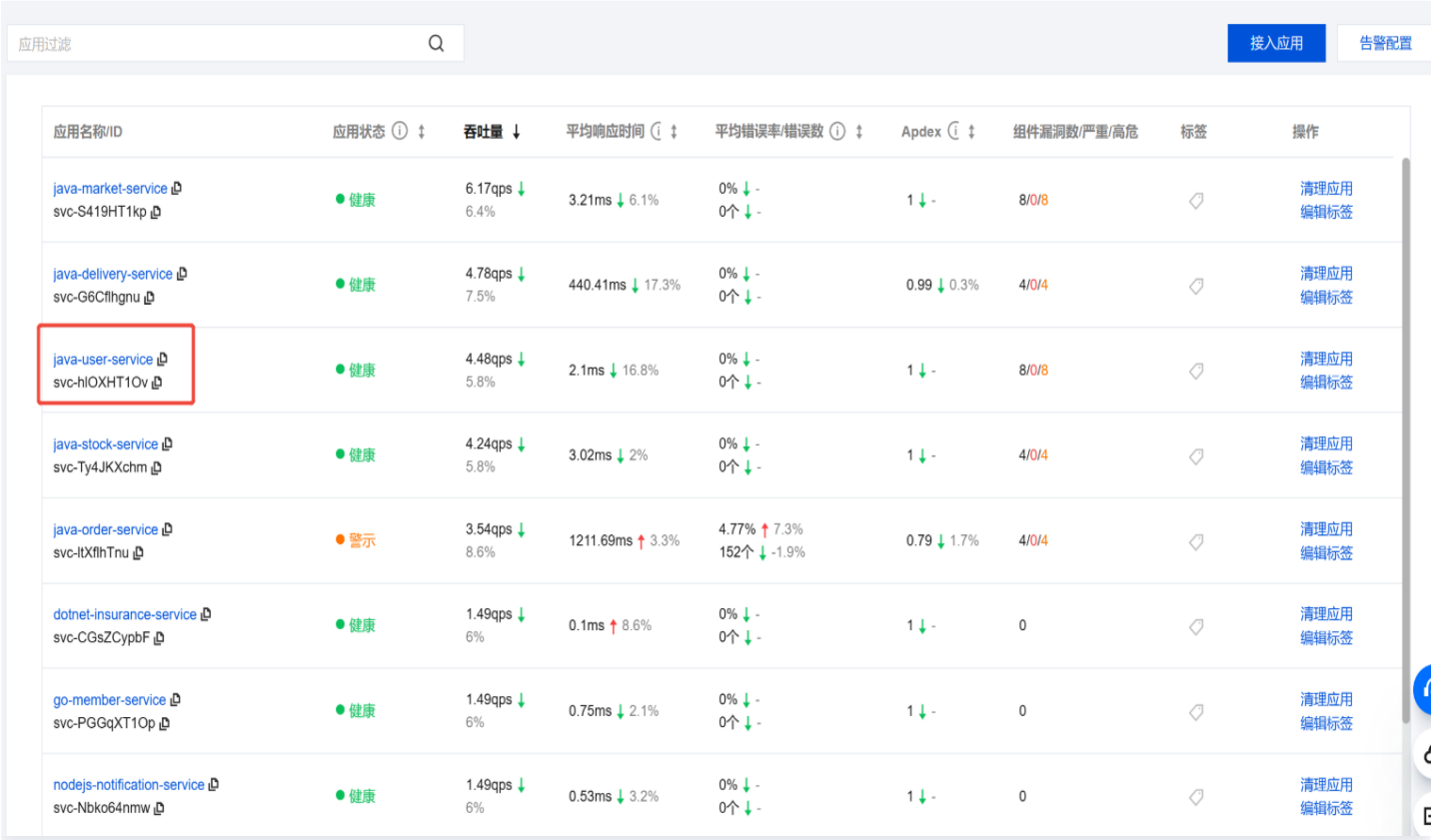
请注意，此内容需要添加到 `spec.template.metadata.annotations` 中，作用于 Pod 的 annotation，而不是工作负载的 annotation，可以参考如下代码片段。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
```

```
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-java: "true"   # 添加到此处
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

接入验证

对工作负载添加 annotation 后，基于不同的发布策略，触发应用 Pod 的重启。新启动的 Pod 会自动注入探针，并连接到 APM 服务端上报监控数据，上报的业务系统为 Operator 的默认业务系统。在有正常流量的情况下，应用性能监控 > 应用列表 中将展示接入的应用，单击**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。

自定义埋点

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 自定义埋点，使用 OpenTelemetry API 添加自定义埋点。

更多接入配置项（可选）

在工作负载级别，可以添加更多的 annotation 对接入行为进行调整。

配置项	描述
cloud.tencent.com/apm-token	指定 APM 业务系统的 Token。若不添加此项配置项，则使用 Operator的配置（对应 Operator 的 `env.APM_TOKEN` 字段）。
cloud.tencent.com/java-instr-version	指定 Java 探针版本。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.JAVA_INSTR_VERSION` 字段）。取值为 `latest`（默认）或具体版本号，具体的版本号列表请参考 探针（Agent）版本信息，非必要情况下不推荐填写此字段。
cloud.tencent.com/container-names	指定注入探针的 container。该配置项支持将探针注入到多个 container 中，只需在该配置项的值中填写多个 container 名并用英文逗号间隔开。若不添加此项配置项，则探针将默认注入到第一个 container 中。该配置项在 operator 0.88及以上版本支持。

通过腾讯云增强版 OpenTelemetry Java 探针接入（推荐）

最近更新时间：2025-03-14 15:38:32

腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-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，腾讯云增强版 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腾讯云增强版 OpenTelemetry 探针接入 Java 应用。

前提条件

请参考 增强版 OpenTelemetry Java 探针支持的 Java 版本和框架，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，腾讯云增强版 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. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入应用**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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

说明：
- 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

步骤2：下载探针

请进入 探针（Agent）版本信息 下载探针，推荐下载最新版本，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-guangzhou.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-guangzhou.apm.tencentcs.com:4317 \
-jar SpringCloudApplication.jar
```

- **Tomcat**

  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-
guangzhou.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-
guangzhou.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://p1-demo.ap-guangzhou.apm.tencentcs.com:4317
```

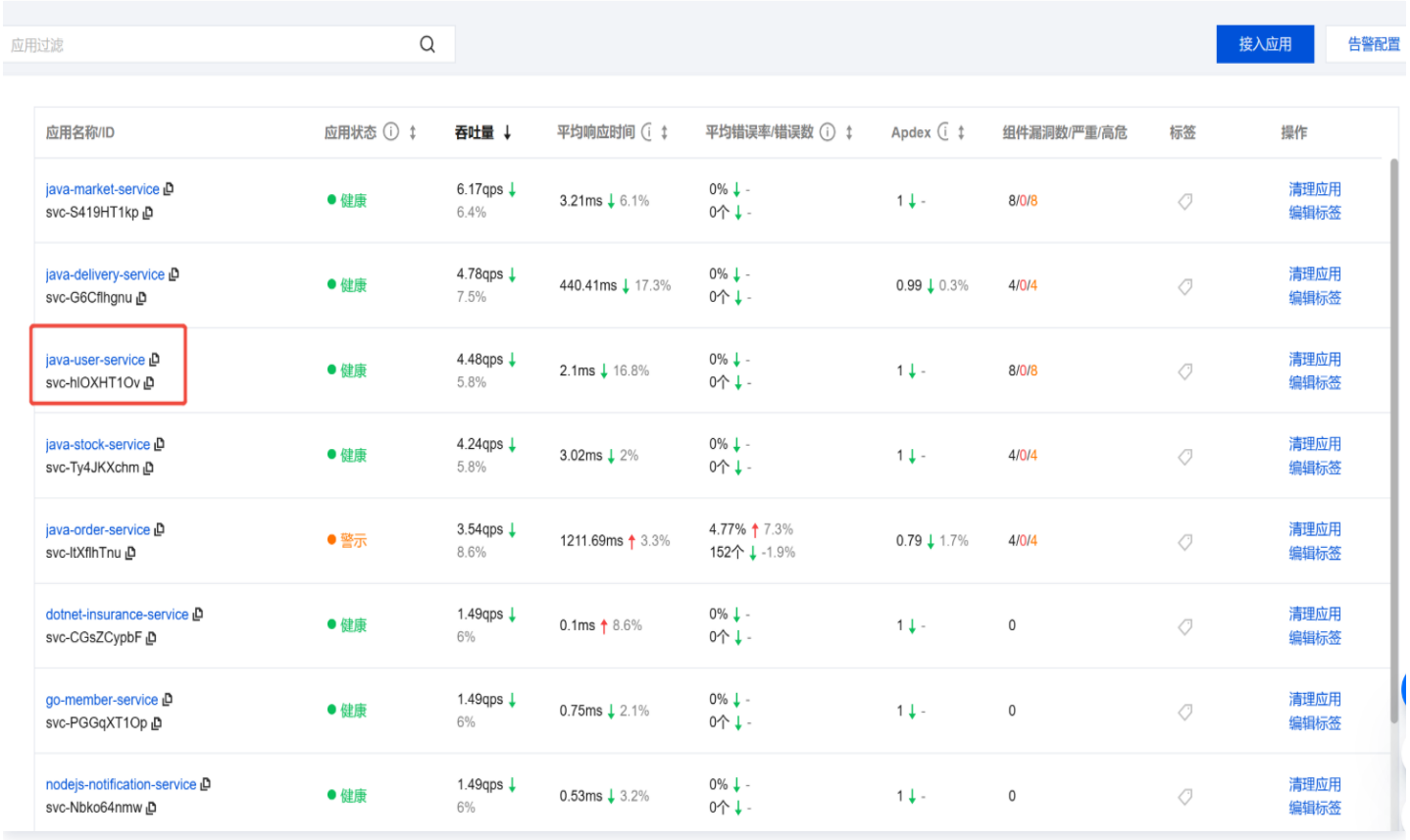
在这种情况下，请确保本地环境和接入点之间的网络连通性，通常可以使用外网上报接入点地址。

- **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，**应用性能监控 > 应用列表** 中将展示接入的应用，单击**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。

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 。

> **说明：**
> 腾讯云增强版 Java 探针2.1-20240701及以上版本支持通过 OpenTelemetry API 自定义埋点。

引入 OpenTelemetry API 依赖

```
<dependencies>
    <!-- 其他依赖 -->
    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
        <version>1.35.0</version>
    </dependency>
</dependencies>
```

获取 Tracer 对象

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 > 链路追踪 中，找到相关的调用链，单击 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。

在代码中获取 TraceId（可选）

通过 OpenTelemetry-API，可以在应用代码中获取当前的 Span，或者对当前的 Span 增加新的属性。

> **说明：**
> 腾讯云增强版 Java 探针2.1-20240701及以上版本支持通过 OpenTelemetry API 对当前 Span 进行操作。

引入 OpenTelemetry API 依赖

```
<dependencies>
    <!-- 其他依赖 -->
    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
```

```
        <version>1.35.0</version>
    </dependency>
</dependencies>
```

获取当前的 Span，并增加新的属性

```
Span span = Span.current(); // 通过静态方法获取当前 Span
String traceId = span.getSpanContext().getTraceId(); // 获取 TraceId
String spanId = span.getSpanContext().getSpanId(); // 获取 SpanId
span.setAttribute("myKey", "myValue"); // 为当前的 Span 增加新的属性
```

自定义实例名称（可选）

当多个应用进程使用相同应用名接入 APM 以后，在 APM 中会表现为相同应用下的多个应用实例。在大多数场景下，IP 地址都可以作为应用实例的唯一标识，但如果系统中的 IP 地址存在重复的情况，就需要使用其他唯一标识定义实例名称。例如，系统中的应用直接通过 Docker 启动，没有基于 Kubernetes 部署，就有可能存在容器 IP 地址重复的情况，用户可以将实例名称设置为 `主机 IP + 容器 IP` 的形式。
参考如下脚本，在接入 APM 需要用到的 JVM 启动参数 `-Dotel.resource.attributes` 中，加上 `host.name` 字段。

```
-
Dotel.resource.attributes=service.name=my_service,token=my_demo_token,host.name=10.10.
1.1_192.168.1.2
```

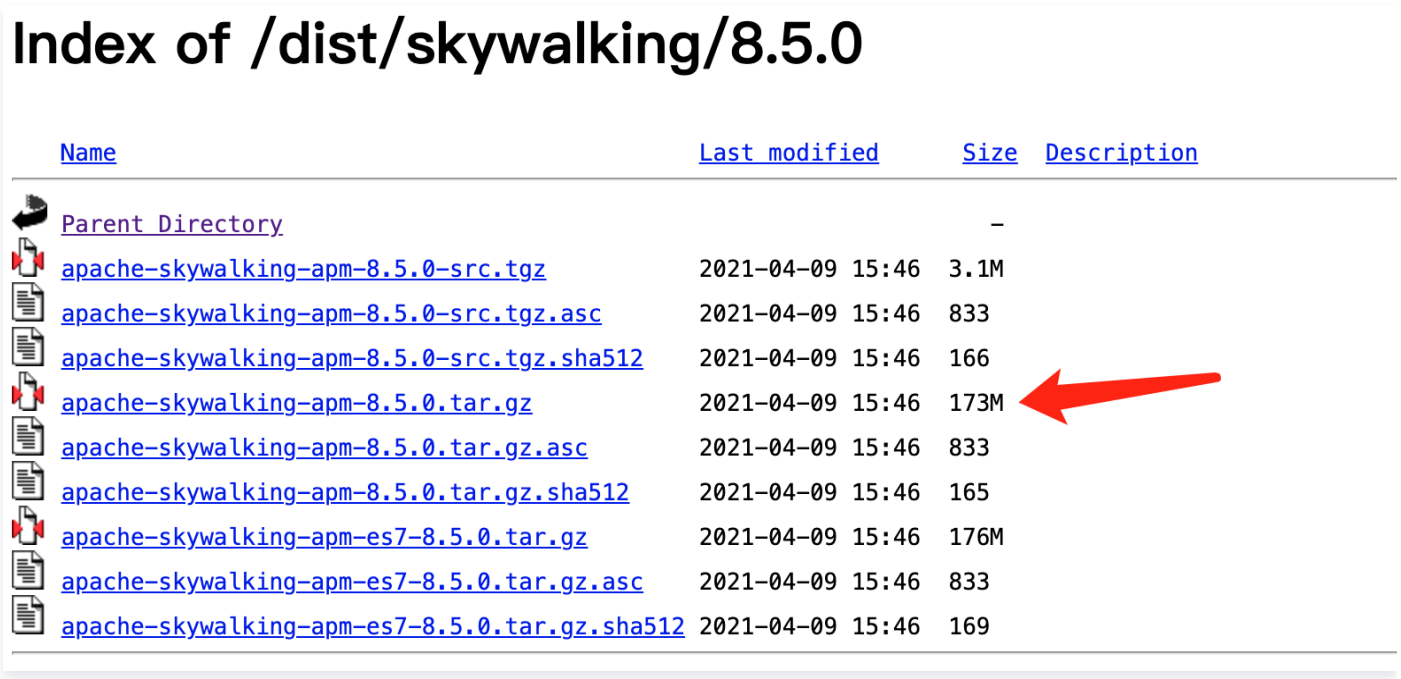
通过 SkyWalking 协议接入 Java 应用

最近更新时间：2024-05-29 17:15:22

本文将为您介绍如何使用 SkyWalking 协议上报 Java 应用数据。

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：

新 Agent 文件夹目录

```
+-- 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 腾讯云可观测平台 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. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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，下载方式参见 前提条件 。

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 Python 应用
K8s 环境自动接入 Python 应用（推荐）

最近更新时间：2025-03-17 11:59:52

对于部署在 Kubernetes 上的 Python 应用，APM 提供自动接入方案，可以实现探针自动注入，方便应用快速接入。
K8s 环境自动接入的 Python 应用将使用社区 OpenTelemetry-Python 方案注入探针，关于 OpenTelemetry-Python 的更多信息，请参考社区 OpenTelemetry-Python 项目 。

前提条件

请参考 OpenTelemetry-Python 方案支持的组件和框架 ，确保 Python 版本、依赖库与框架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。如果自动埋点不满足您的场景，或者需要增加业务层埋点，请使用 OpenTelemetry API 进行自定义埋点 。

Django 应用注意事项

如果您的应用使用 Django 框架，在接入前需要注意如下事项：

- 要在 K8s 环境通过 Operator 自动接入，需要确保 Python 版本为3.8及以上。
- 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。
- OpenTelemetry-Python 的引入，可能会导致 Django 应用不再使用默认的配置文件，需要通过环境变量重新指定配置文件。

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

  可以在项目的启动脚本中加入此设置项，或者通过 YAML 文件添加环境变量。

  ```
  env:
  - name: DJANGO_SETTINGS_MODULE
    value: mysite.settings
  ```

操作步骤

步骤1：安装 Operator

在 K8s 集群安装 Operator，推荐从 APM 控制台一键安装 Operator，详情请参考 安装 tencent-opentelemetry-operator 。

步骤2：在工作负载添加 annotation

以容器服务 TKE 为例，通过如下步骤可以在工作负载中添加 annotation。对于通用 K8s 集群，请通过 kubectl 等工具添加 annotation：

1. 登录 容器服务 控制台。
2. 单击**集群**，进入对应的 TKE 集群。
3. 在**工作负载**中找到需要接入 APM 的应用，单击**更多**，单击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，单击**完成**，即可以完成接入。

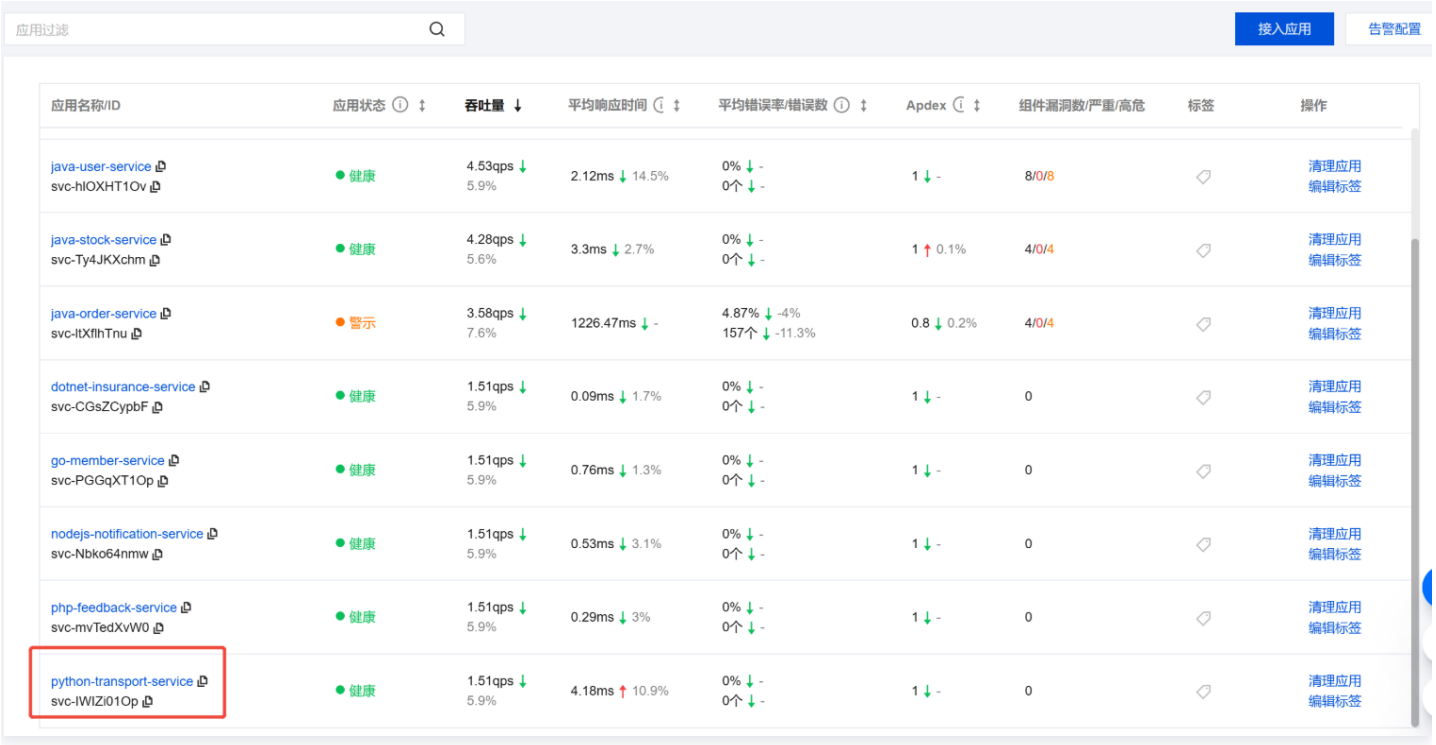
```
cloud.tencent.com/inject-python: "true"
cloud.tencent.com/otel-service-name: my-app  # 应用名，多个使用相同应用名接入的进程，在APM中会表现为相同应用下的多个实例
# 应用名最长63个字符，只能包含小写字母、数字及分隔符("-")，且必须以小写字母开头，数字或小写字母结尾。
```

请注意，此内容需要添加到 `spec.template.metadata.annotations` 中，作用于 Pod 的 annotation，而不是工作负载的 annotation，可以参考如下代码片段。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-python: "true"  # 添加到此处
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

接入验证

对工作负载添加 annotation 后，基于不同的发布策略，触发应用 Pod 的重启。新启动的 Pod 会自动注入探针，并连接到 APM 服务端上报监控数据，上报的业务系统为 Operator 的默认业务系统。在有正常流量的情况下，应用性能监控 > 应用列表 中将展示接入的应用，单击**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。

更多接入配置项（可选）

在工作负载级别，可以添加更多的 annotation 对接入行为进行调整。

配置项	描述
cloud.tencent.com/apm-token	指定 APM 业务系统的 Token。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.APM_TOKEN` 字段）。
cloud.tencent.com/python-instr-version	指定 Python 探针版本。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.PYTHON_INSTR_VERSION` 字段）。取值为 `latest`（默认）或具体版本号，具体的版本号列表请参考 探针（Agent）版本信息，非必要情况下不推荐填写此字段。
cloud.tencent.com/container-names	指定注入探针的 container。该配置项支持将探针注入到多个 container 中，只需在该配置项的值中填写多个 container 名并用英文逗号间隔开。若不添加此项配置项，则探针将默认注入到第一个 container 中。该配置项在 operator 0.88及以上版本支持。

通过 OpenTelemetry-Python 接入 Python 应用（推荐）

最近更新时间：2025-05-09 18:11:33

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。

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：

- 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。
- OpenTelemetry-Python 的引入，可能会导致 Django 应用不再使用默认的配置文件，需要通过环境变量重新指定配置文件。

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

gevent 应用注意事项

如果项目中将 gevent 作为协程库与 Python 自动埋点探针结合使用，可能会出现一系列问题，包括启动报错、Span 异常断链等问题。以下是几种可能的解决方案：

- `gevent.monkey.patch_all()` 需要尽早执行：gevent 通过 monkey patching 修改 Python 标准库中的阻塞式 I/O 操作，使其变为非阻塞式，以适应协程的并发模型。OpenTelemetry 的自动埋点探针也可能依赖或修改标准库的行为。如果 monkey patching 的顺序不当，或者 OpenTelemetry 探针在 gevent 完成 patching 之前加载了某些模块，就可能造成启动报错 `RecursionError`。

```
from gevent import monkey
monkey.patch_all() # 尽可能早地执行

# 之后再导入其他模块和 OpenTelemetry 相关库
from flask import Flask
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
# ... 其他 OpenTelemetry 配置和应用代码
```

- 如果您在使用 Gunicorn 作为 Python 程序的 HTTP 服务器，我们更推荐用 uvicorn 或 gthread 替换 gevent 作为 worker class，以避免潜在的问题。

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，或直接购买云产品。

```
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.text

# 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)
```

获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入应用**抽屉框中，单击 **Python 语言**。
4. 在**接入 Python 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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。

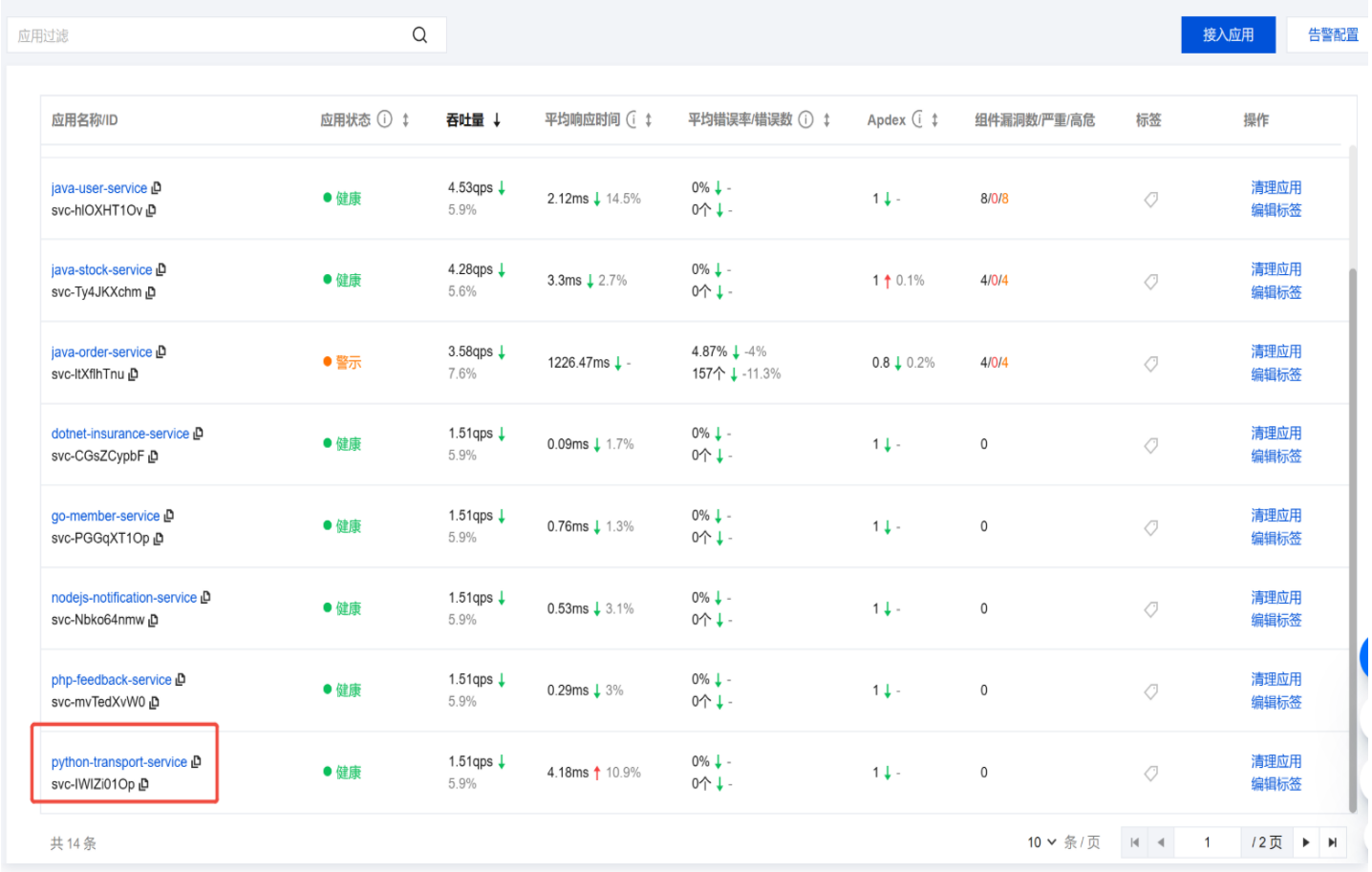
- `<endpoint>` ：前置步骤中拿到的接入点。

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-guangzhou.apm.tencentcs.com:4317` 为例，完整的启动命令为如下。

```
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 > 应用列表 中将展示接入的应用，单击**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。

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
from flask import Flask
import time

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.text

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
```

问题排查

如果发现自动埋点无法上报数据到 APM，可以从通过设置 `--traces_exporter=console,otlp_proto_grpc` 将链路数据打印到控制台，如果发现控制台中有 span 和 trace 等输出，但是 APM 中没有数据，则排查是否是 endpoint 和 token 有误；如果发现控制台中没有相关的输出，可以检查应用程序所使用的框架或组件是否在 OpenTelemetry 支持的 完整列表 中。如果无法确认具体问题，可以向我们 提交工单 。

通过 SkyWalking 协议接入 Python 应用

最近更新时间：2024-09-20 10:24:52

本文将为您介绍如何使用 SkyWalking 协议上报 Python 应用数据。

前提条件

- Python 使用 SkyWalking 可以实现自动埋点上报，需 Python 3.7及以上版本。
- 目前我们默认使用 grpc 进行上报。
- 当前支持自动埋点的组件详情请见 Python Agent 支持框架 。

操作步骤

步骤1：获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**数据接入**抽屉框中，单击 **Python** 语言。
4. 在**接入 Python 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. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

步骤2：安装 SkyWalking Agent 相关依赖

```
pip install apache-skywalking
# 当前demo使用库
pip install tornado
```

步骤3：修改启动代码

```
from skywalking import agent, config
config.init(
    # 此处替换成步骤1中获得的接入点
    agent_collector_backend_services='ap-guangzhou.apm.tencentcs.com:11800',
    agent_name='python-skywalking', # 此处可自定义应用名称
    agent_authentication="xxxxxxxxxxxxx", # 此处替换成步骤1中获得的Token
    agent_logging_level="INFO")
agent.start()
```

完整 demo 代码如下：

```
import time
import tornado.ioloop
import tornado.web


class MainHandler(tornado.web.RequestHandler):
    def get(self):
        print("call")
        self.write("Hello, world")


class SleepHandler(tornado.web.RequestHandler):
    def get(self):
        print("call sleep")
        time.sleep(0.1)
        self.write("sleep 0.1s")


def make_app():
    return tornado.web.Application([
        (r"/test", MainHandler),
        (r"/sleep", SleepHandler),
    ])


if __name__ == "__main__":
    from skywalking import agent, config
    config.init(agent_collector_backend_services='{接入点}', # 此处替换成步骤1中获得的接入点
                agent_name='python-skywalking',
                agent_authentication='{token}', # 此处替换成步骤1中获得的Token
                agent_logging_level='INFO')
    agent.start()
    app = make_app()
    port = 9008
    app.listen(port)
    print("server in %s" % port)
    tornado.ioloop.IOLoop.current().start()
```

步骤4：多线程接入

此步骤可选。完成上面三个步骤，即可在 腾讯云可观测平台 > 应用性能监控 > 应用列表 中看到上报的数据。如果想要多线程接入，可以使用 tornado 包，这里启动需要注意，SkyWalking Agent 需要在进程 fork 之后启动。

引入 tornado 依赖

```
pip install tornado
```

完整 demo 代码如下：

```
import time
import os
import tornado.ioloop
import tornado.web
from tornado.httpserver import HTTPServer


class MainHandler(tornado.web.RequestHandler):
    def get(self):
        print("call")
        self.write("Hello, world")


class Sleep01Handler(tornado.web.RequestHandler):
    def get(self):
        print("call sleep")
        time.sleep(0.1)
        self.write("sleep 0.1s")


def make_app():
    return tornado.web.Application([
        (r"/test", MainHandler),
        (r"/sleep", Sleep01Handler),
    ])


if __name__ == "__main__":
    from skywalking import agent, config
    config.init(agent_collector_backend_services='{接入点}', # 此处替换成步骤1中获得的接入点
                agent_name='python-skywalking-multi',
                agent_authentication='{token}', # 此处替换成步骤1中获得的Token
                agent_logging_level='INFO')
    app = make_app()
    port = 9009
    num_processes = 4
    sockets = tornado.netutil.bind_sockets(port)
    tornado.process.fork_processes(num_processes)
    # 多进程用法，应在fork进程之后启动
    agent.start()
    pid = os.getpid()
    print("Current process ID:%d" % pid)
    server = HTTPServer(app)
    server.add_sockets(sockets)
    print("server in %s" % port)
    tornado.ioloop.IOLoop.current().start()
```

接入 PHP 应用
通过 OpenTelemetry-PHP 接入 PHP 应用（推荐）

最近更新时间：2025-03-17 11:59:52

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，或直接购买云产品。
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<project-name>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();
```

获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**数据接入**抽屉框中，单击 **PHP** 语言。

4. 在**接入 PHP 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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
<service-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
```

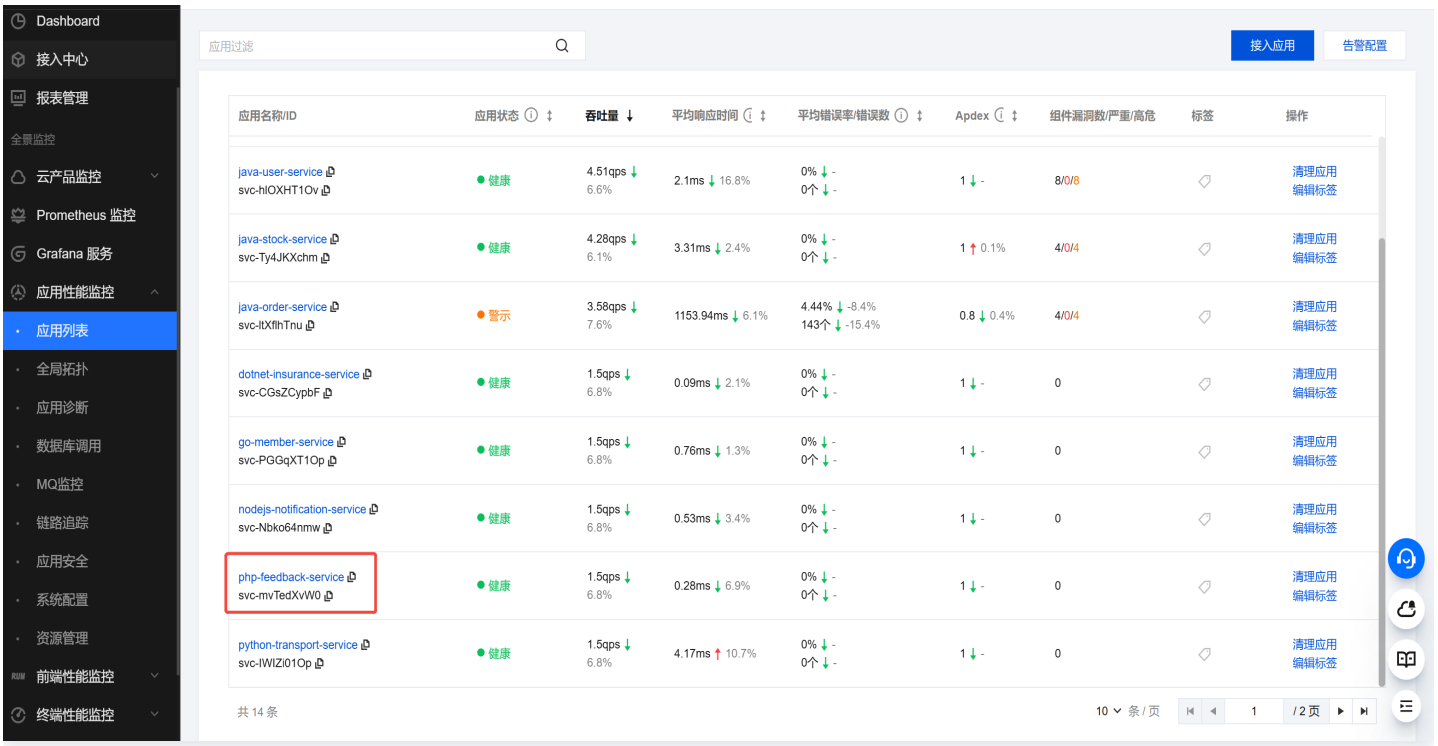
2. 在浏览器中访问以下链接。

```
http://localhost:8080/getID
```

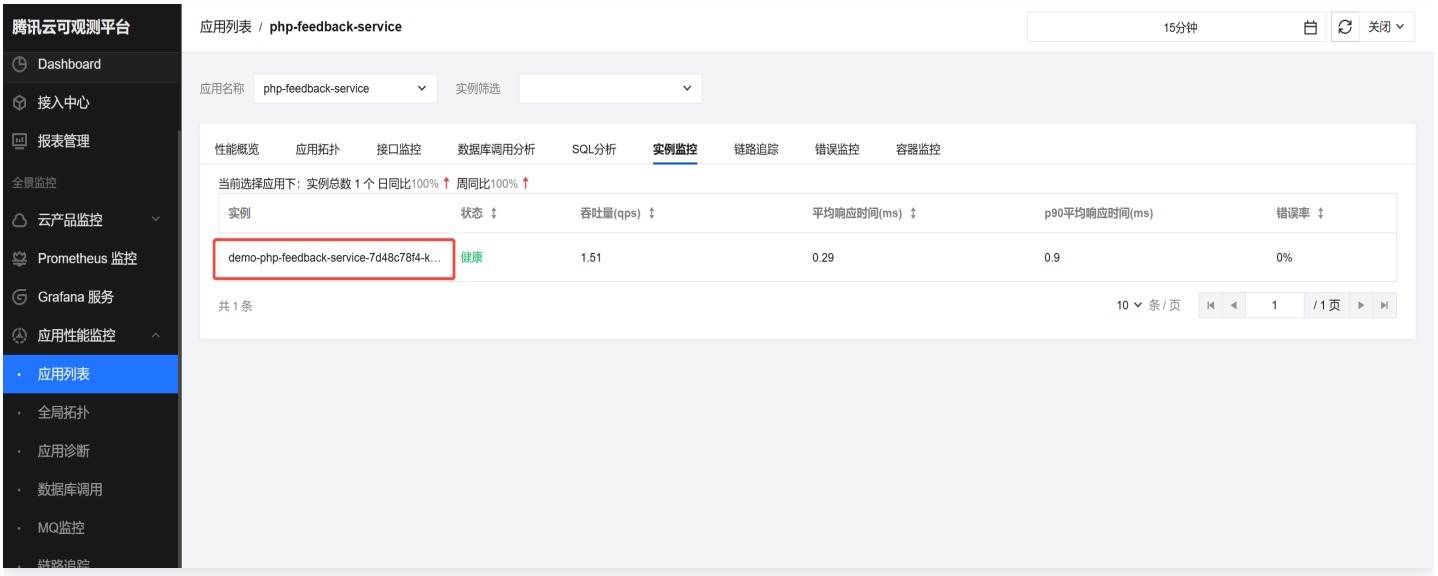
每次进入该页面，OpenTelemetry 都会自动创建 Trace，并将链路数据上报至 APM。

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 > 应用列表 中将展示接入的应用，单击**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。



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 TraceProvider，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
span3!");
 // 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 > 应用列表 中将展示接入的应用，单击**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。

接入 Node.js 应用
K8s 环境自动接入 Node.js 应用（推荐）

最近更新时间：2025-03-14 15:38:32

对于部署在 Kubernetes 上的 Node.js 应用，APM 提供自动接入方案，可以实现探针自动注入，方便应用快速接入。
K8s 环境自动接入的 Node.js 应用将使用社区 OpenTelemetry-JavaScript 方案注入探针，关于 OpenTelemetry-JavaScript 的更多信息，请参考社区 OpenTelemetry-Javascript 项目 。

前提条件

请参考 OpenTelemetry-JavaScript 方案支持的组件和框架 ，确保 Node.js 版本、依赖库与框架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。如果自动埋点不满足您的场景，或者需要增加业务层埋点，请使用 OpenTelemetry API 进行自定义埋点 。

步骤1：安装 Operator

在 K8s 集群安装 Operator，推荐从 APM 控制台一键安装 Operator，详情请参考 安装 tencent-opentelemetry-operator 。

步骤2：在工作负载添加 annotation

以容器服务 TKE 为例，通过如下步骤可以在工作负载中添加 annotation，对于通用 K8s 集群，请通过 kubectl 等工具添加 annotation。

1. 登录 容器服务 控制台。
2. 单击**集群**，进入对应的 TKE 集群。
3. 在**工作负载**中找到需要接入 APM 的应用，单击**更多**，单击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，单击**完成**，即可以完成接入。

```
cloud.tencent.com/inject-nodejs: "true"
cloud.tencent.com/otel-service-name: my-app  # 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
# 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。
```

请注意，此内容需要添加到 `spec.template.metadata.annotations` 中，作用于 Pod 的 annotation，而不是工作负载的 annotation，可以参考如下代码片段。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
```

```
      k8s-app: my-app
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-nodejs: "true"  # 添加到此处
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

Next.js 框架注意事项

Operator 默认使用 grpc 协议上报数据，但如果您的应用使用 Next.js 框架，需要将接入点地址的协议更改为 **http**，并通过环境变量 `OTEL_EXPORTER_OTLP_ENDPOINT` 进行指定。假定您在设置台获取的接入点地址为 `grpc://ap-shanghai.apm.tencentcs.com:4317`，您需要将接入点地址修改为 `http://ap-shanghai.apm.tencentcs.com:4317`。
可以在项目的启动脚本中加入此设置项。

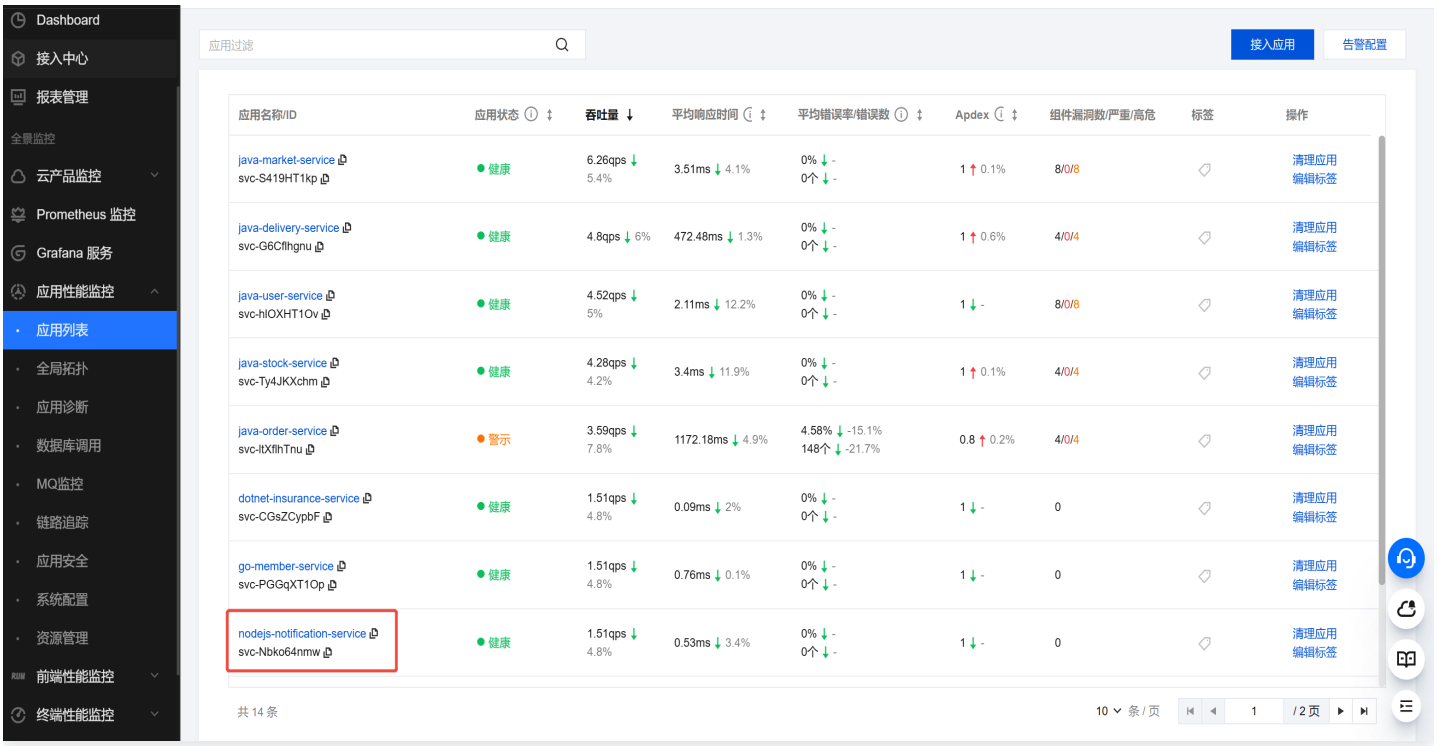
```
export OTEL_EXPORTER_OTLP_ENDPOINT=<HTTP协议接入点>
```

或者通过 YAML 文件添加环境变量。

```
env:
- name: OTEL_EXPORTER_OTLP_ENDPOINT
  value: <HTTP协议接入点>
```

接入验证

对工作负载添加 annotation 后，基于不同的发布策略，触发应用 Pod 的重启。新启动的 Pod 会自动注入探针，并连接到 APM 服务端上报监控数据，上报的业务系统为 Operator 的默认业务系统。在有正常流量的情况下，应用性能监控 > 应用列表 中将展示接入的应用，单击**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。

更多接入配置项（可选）

在工作负载级别，可以添加更多的 annotation 对接入行为进行调整。

配置项	描述
cloud.tencent.com/apm-token	指定 APM 业务系统的 Token。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.APM_TOKEN` 字段）。
cloud.tencent.com/nodejs-instr-version	指定 Node.js 探针版本。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.NODEJS_INSTR_VERSION` 字段）。取值为 `latest`（默认）或具体版本号，具体的版本号列表请参考 探针（Agent）版本信息，非必要情况下不推荐填写此字段。

cloud.tencent.com/container-names	指定注入探针的 container。该配置项支持将探针注入到多个 container 中，只需在该配置项的值中填写多个 container 名并用英文逗号间隔开。若不添加此项配置项，则探针将默认注入到第一个 container 中。该配置项在 operator 0.88及以上版本支持。

通过 OpenTelemetry-JS 方案接入 Node.js 应用（推荐）

最近更新时间：2025-03-14 15:38:32

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，或直接购买云产品。

```
"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.tencent.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");
```

```
});
```

获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入应用**抽屉框中，单击 **Node** 语言。
4. 在**接入 Node 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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>,host.name=<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：

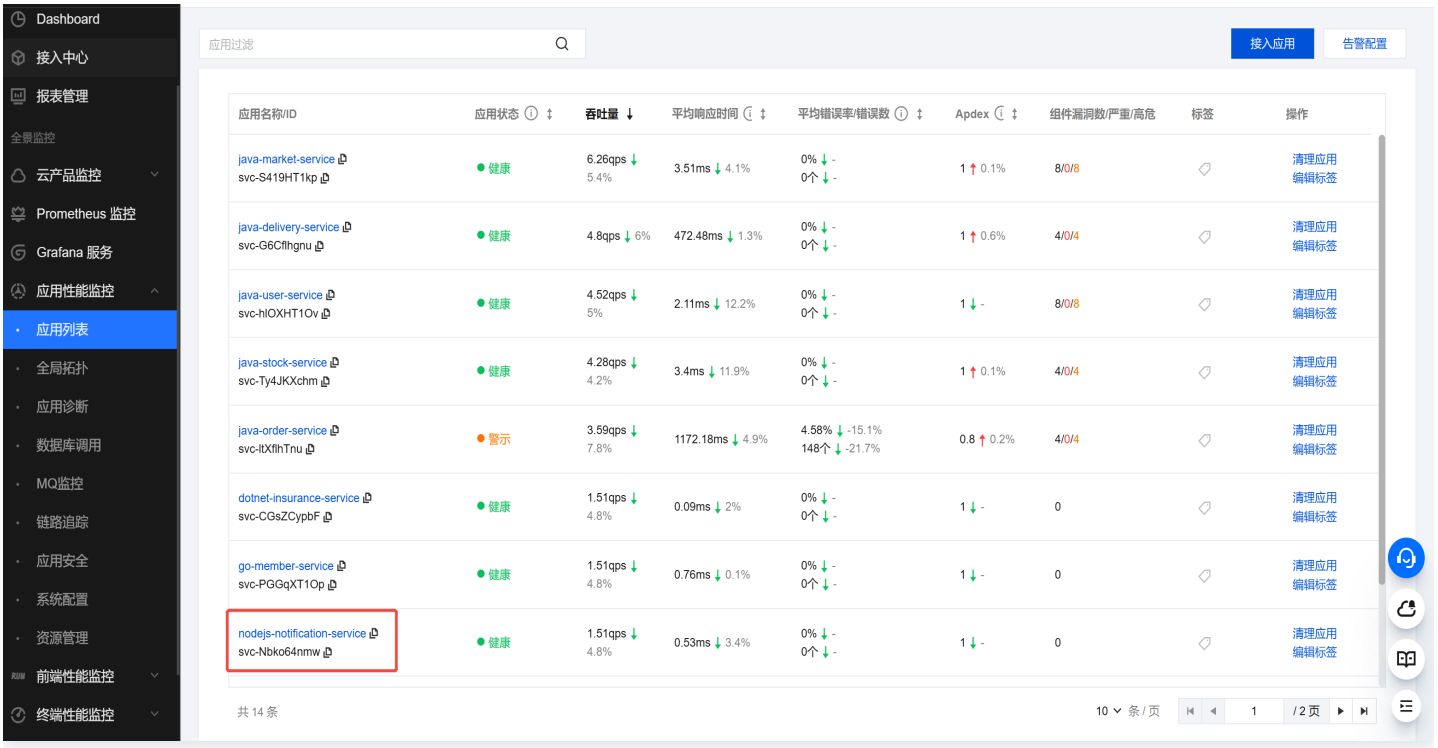
- `<serviceName>` ：应用名，多个使用相同 serviceName 接入的应用进程，在 APM 中会表现为相同应用下的多个实例。应用名最长63个字符，只能包含小写字母、数字及分隔符“ - ”，且必须以小写字母开头，数字或小写字母结尾。
- `<token>` ：前置步骤中拿到业务系统 Token。
- `<hostName>`：该实例的主机名，是应用实例的唯一标识，通常情况下可以设置为应用实例的 IP 地址。
- `<endpoint>` ：前置步骤中拿到的接入点。

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-guangzhou.apm.tencentcs.com:4317` 为例，完整的启动命令如下。

```
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-
guangzhou.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 > 应用列表 中将展示接入的应用，单击**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。

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':'2222222',
      'log.message1':'3333333',
    })
    span.addEvent("Unlocking")
    span.end();
  });
  res.json({ code: 200, msg: "success" });
})
```

接入 .NET 应用
K8s 环境自动接入 .NET 应用（推荐）

最近更新时间：2025-03-14 15:38:32

对于部署在 Kubernetes 上的 .NET 应用，APM 提供自动接入方案，可以实现探针自动注入，方便应用快速接入。
K8s 环境自动接入的 .NET 应用将使用社区 OpenTelemetry-Dotnet 方案注入探针，关于 OpenTelemetry-Dotnet 的更多信息，请参考社区 OpenTelemetry-Dotnet 项目 。

前提条件

请参考 OpenTelemetry-Dotnet 方案支持的组件和框架，确保 .NET 版本、依赖库与框架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。如果自动埋点不满足您的场景，或者需要增加业务层埋点，请使用 。

步骤1：安装 Operator

在 K8s 集群安装 Operator，推荐从 APM 控制台一键安装 Operator，详情请参考 安装 tencent-opentelemetry-operator 。

步骤2：在工作负载添加 annotation

以容器服务 TKE 为例，通过如下步骤可以在工作负载中添加 annotation。对于通用 K8s 集群，请通过 kubectl 等工具添加 annotation。

1. 登录 容器服务 控制台。
2. 单击**集群**，进入对应的 TKE 集群。
3. 在**工作负载**中找到需要接入 APM 的应用，单击**更多**，单击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，单击**完成**，即可以完成接入。

```
cloud.tencent.com/inject-dotnet: "true"
cloud.tencent.com/otel-service-name: my-app  # 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
# 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。
```

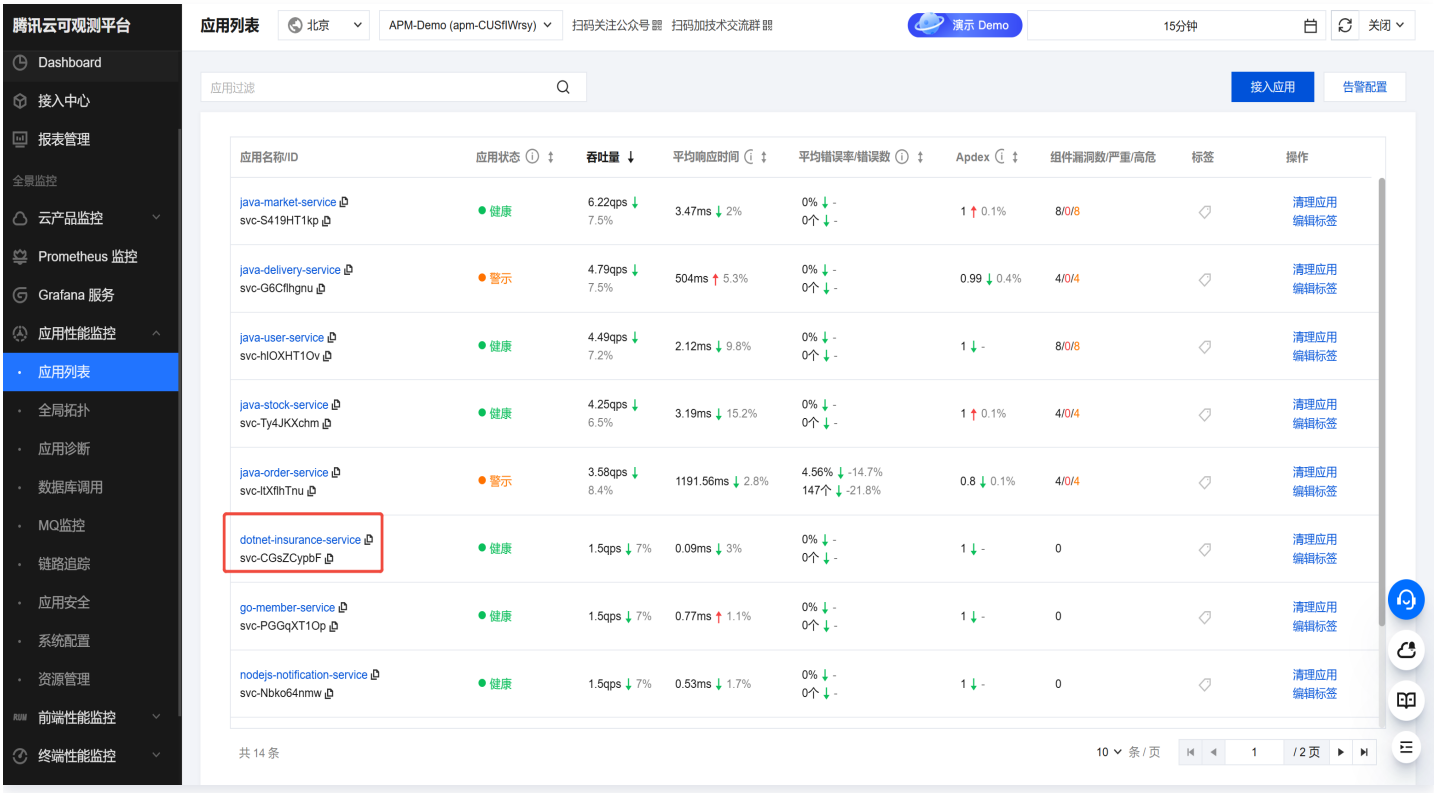
请注意，此内容需要添加到 `spec.template.metadata.annotations` 中，作用于 Pod 的 annotation，而不是工作负载的 annotation，可以参考如下代码片段。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
```

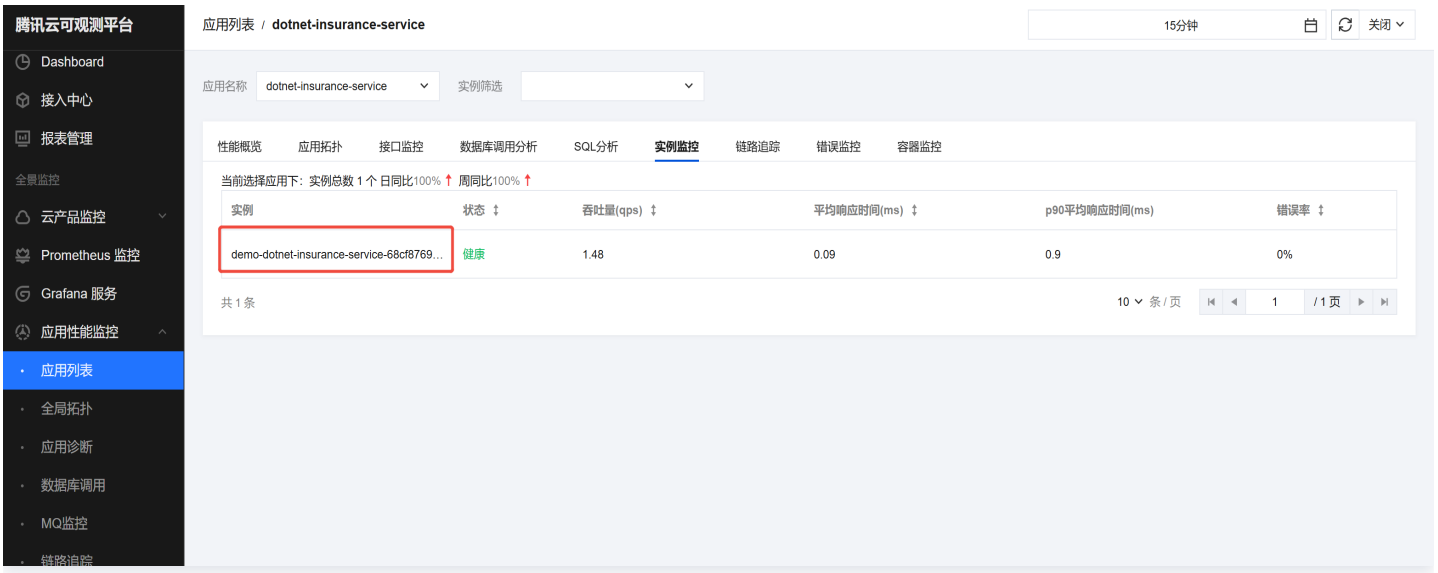
```
      k8s-app: my-app
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-dotnet: "true"  # 添加到此处
        cloud.tencent.com/otel-service-name: my-app
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

接入验证

对工作负载添加 annotation 后，基于不同的发布策略，触发应用 Pod 的重启。新启动的 Pod 会自动注入探针，并连接到 APM 服务端上报监控数据，上报的业务系统为 Operator 的默认业务系统。在有正常流量的情况下，应用性能监控 > 应用列表 中将展示接入的应用，单击**应用名称/ID** 进入对应的应用。



再选择**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。



更多接入配置项（可选）

在工作负载级别，可以添加更多的 annotation 对接入行为进行调整：

配置项	描述
cloud.tencent.com/apm-token	指定 APM 业务系统的 Token。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.APM_TOKEN` 字段）。
cloud.tencent.com/dotnet-instr-version	指定 .NET 探针版本。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.DOTNET_INSTR_VERSION` 字段）。取值为 `latest`（默认）或具体版本号，具体的版本号列表请参考 探针（Agent）版本信息，非必要情况下不推荐填写此字段。
cloud.tencent.com/container-names	指定注入探针的 container。该配置项支持将探针注入到多个 container 中，只需在该配置项的值中填写多个 container 名并用英文逗号间隔开。若不添加此项配置项，则探针将默认注入到第一个 container 中。该配置项在 operator 0.88及以上版本支持。
cloud.tencent.com/otel-dotnet-auto-runtime	指定 .NET CLR Profiler 位置。若不添加此项配置项，则使用 Operator 的配置（对应 Operator 的 `env.CORECLR_PROFILER_PATH` 字段）。非必要情况下不推荐填写此字段。该配置项在 operator 0.88及以上版本支持。 可选择的配置项： • `linux-x64`（默认）：适用于基于 Linux glibc 的镜像。 • `linux-musl-x64`：适用于基于 Linux musl 的镜像。

通过 OpenTelemetry-dotnet 接入 .NET 应用（推荐）

最近更新时间：2025-03-17 11:59:52

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 ⩾ 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，或直接购买云产品。

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 myConnnect = new MySqlConnection(connectionString))
    {
        try
        {
            myConnnect.Open();
            using (MySqlCommand myCmd = new MySqlCommand("select * from MyTable",
myConnnect))
            {
                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();
```

获取接入点和 Token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**接入应用**抽屉框中，单击 **.NET** 语言。
4. 在**接入 .NET 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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
```

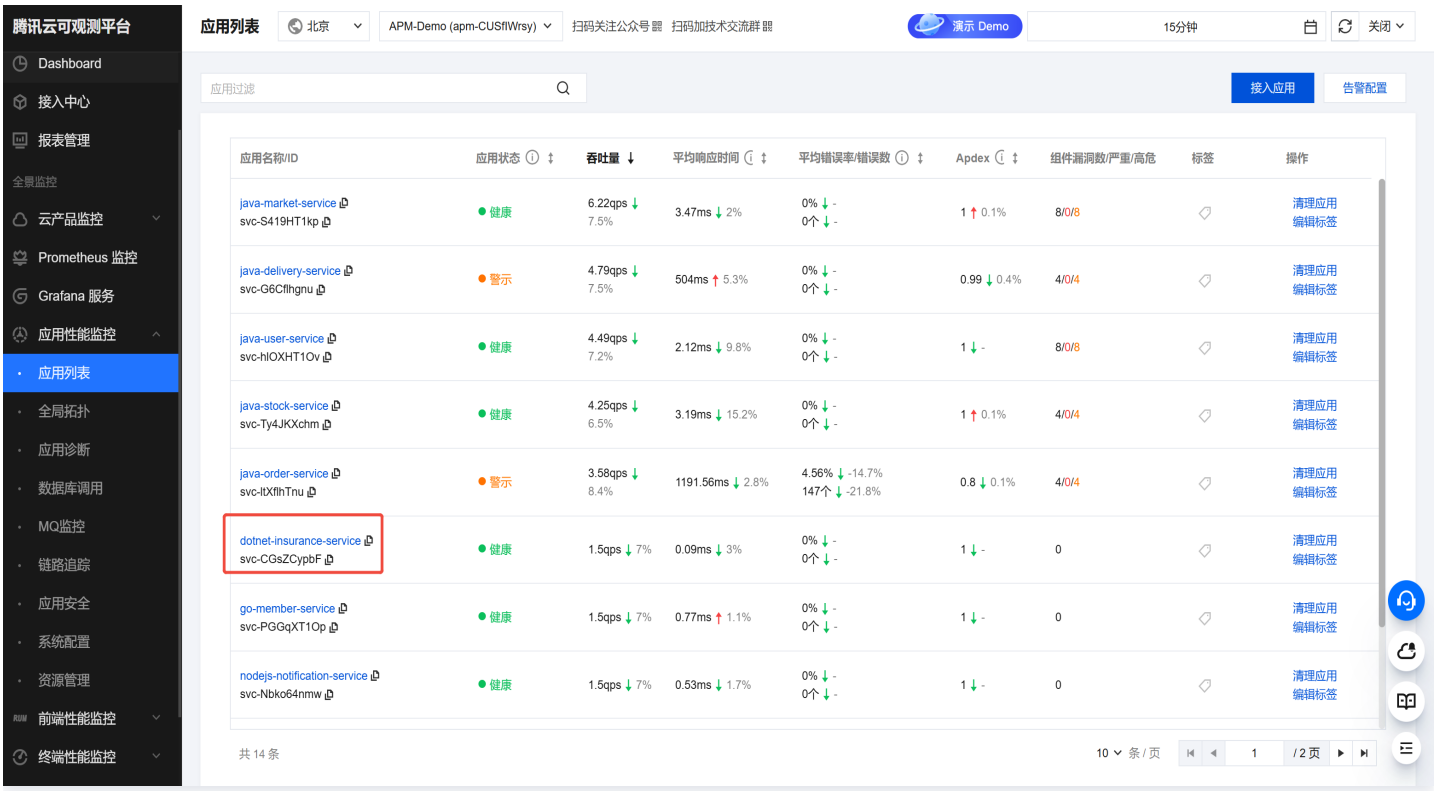
2. 在浏览器中访问以下链接：

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

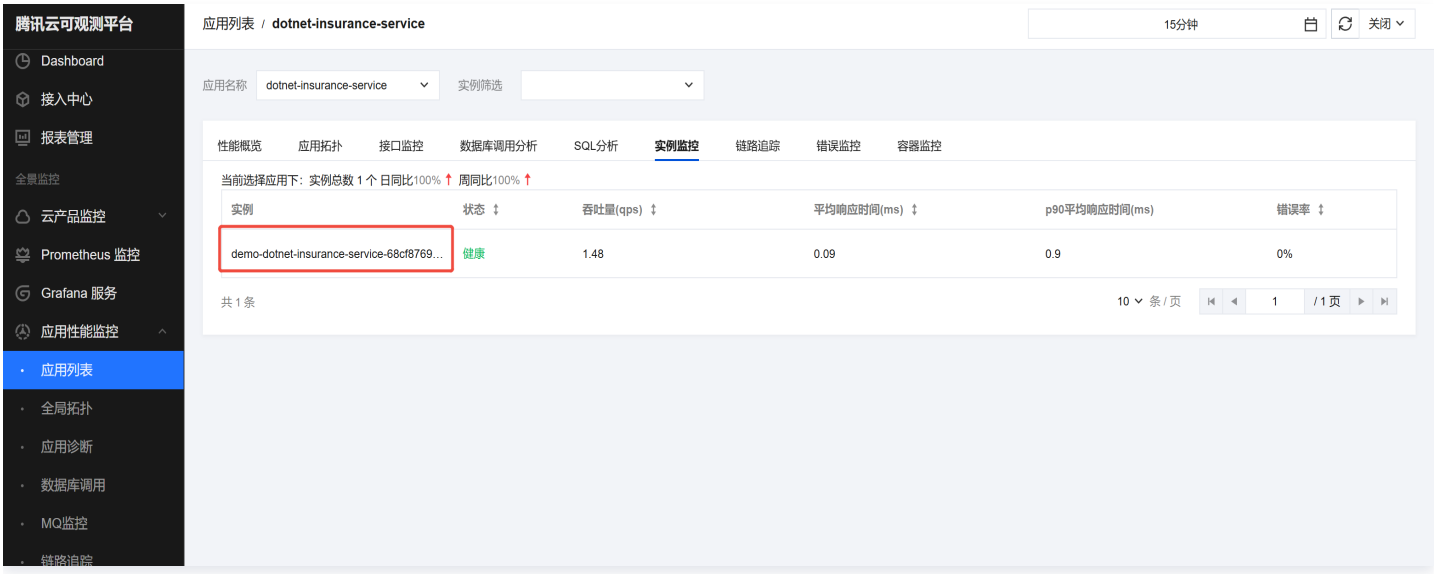
每次进入该页面，OpenTelemetry 都会自动创建 Trace，并将链路数据上报至 APM。

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 > 应用列表 中将展示接入的应用，单击**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。



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1：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2：初始化 OpenTelemtry 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3：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4：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 > 应用列表 中将展示接入的应用，单击**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。

接入 Nginx

最近更新时间：2025-02-13 10:33:52

Nginx 社区提供 nginx-module-otel 动态模块，支持 OpenTelemetry 标准上报可观测数据，安装此模块后，通过简单的配置可以接入应用性能监控 APM。

前提条件

操作系统要求

以下操作系统经过测试确认支持 Nginx 接入，对于未在该范围内的操作系统，由于在底层架构、系统配置及运行环境等方面存在差异性，我们无法对其接入效果及稳定性予以保证。

发行版	版本	架构
Ubuntu	20.04、22.04、24.04、24.10	x86_64、aarch64/arm64
RHEL 及其衍生版（CentOS 等）	8.x、9.x	x86_64、aarch64/arm64
Alpine	3.18、3.19、3.20、3.21	x86_64、aarch64/arm64
Debian	11.x、12.x	x86_64、aarch64/arm64
SLES	15 SP2+	x86_64

Nginx 版本要求

请确保已经安装 Nginx，且版本不低于1.23.4。不满足版本要求的 Nginx 可能无法安装 nginx-module-otel 动态模块。

获取接入点和 token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的数据接入抽屉框中，单击 **Go 语言**。
4. 在**接入 Go 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 **OpenTelemetry**。
6. 选择您所想要的**上报方式**，获取接入点和 Token。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

编译动态模块 nginx-module-otel

目前，在 Nginx 社区已经发布的 nginx-module-otel 版本中，不支持对 Resource Attributes 的指定。而接入 APM 需要在 Resource Attributes 中填入正确的 token，这导致我们无法直接使用 Nginx 官方仓库中已经发布的 nginx-module-otel 模

块。事实上，在 Nginx 社区的源代码仓库中，最新的代码已经支持了对 Resource Attributes 的指定，因此我们基于社区的最新代码，自行编译 nginx-module-otel 即可。本文以 Ubuntu 系统为例，介绍编译流程。

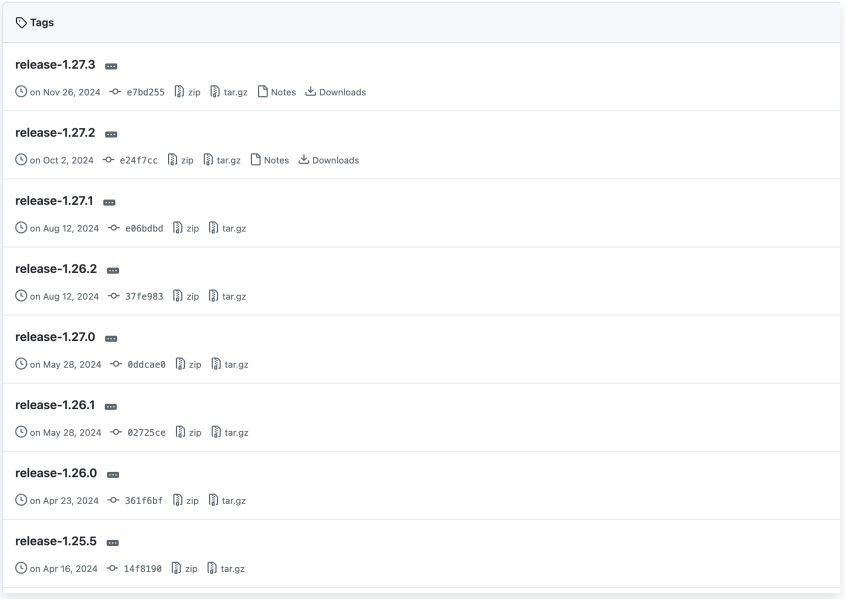
1. 获取 Nginx 的 configure 参数（编译时的所有配置选项）。

```
nginx -V
```

输出示例：

```
nginx version: nginx/1.26.2
built by gcc 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
built with OpenSSL 3.0.2 15 Mar 2022
TLS SNI support enabled
configure arguments: --prefix=/etc/nginx --sbin-path=/usr/sbin/nginx --modules-
path=/usr/lib/nginx/modules --conf-path=/etc/nginx/nginx.conf --error-log-
path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --pid-
path=/var/run/nginx.pid --lock-path=/var/run/nginx.lock --http-client-body-temp-
path=/var/cache/nginx/client_temp --http-proxy-temp-path=/var/cache/nginx/proxy_temp
--http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp --http-uwsgi-temp-
path=/var/cache/nginx/uwsgi_temp --http-scgi-temp-path=/var/cache/nginx/scgi_temp --
user=nginx --group=nginx --with-compat --with-file-aio --with-threads --with-
http_addition_module --with-http_auth_request_module --with-http_dav_module --with-
http_flv_module --with-http_gunzip_module --with-http_gzip_static_module --with-
http_mp4_module --with-http_random_index_module --with-http_realip_module --with-
http_secure_link_module --with-http_slice_module --with-http_ssl_module --with-
http_stub_status_module --with-http_sub_module --with-http_v2_module --with-
http_v3_module --with-mail --with-mail_ssl_module --with-stream --with-
stream_realip_module --with-stream_ssl_module --with-stream_ssl_preread_module --
with-cc-opt='-g -O2 -ffile-prefix-map=/data/builder/debuild/nginx-
1.26.2/debian/debuild-base/nginx-1.26.2=. -flto=auto -ffat-lto-objects -flto=auto -
ffat-lto-objects -fstack-protector-strong -Wformat -Werror=format-security -Wp,-
D_FORTIFY_SOURCE=2 -fPIC' --with-ld-opt='-Wl,-Bsymbolic-functions -flto=auto -ffat-
lto-objects -flto=auto -Wl,-z,relro -Wl,-z,now -Wl,--as-needed -pie'
```

2. 记录下输出结果中的版本号（nginx version）和配置参数（configure arguments）。根据版本号，前往 Nginx 官方仓库 下载对应版本的 Nginx 源码。



3. 以1.26版本为例，将项目下载到本地并解压。

```
wget https://github.com/nginx/nginx/archive/refs/tags/release-1.26.2.zip
unzip release-1.26.2.zip
```

4. 进入项目文件夹，配置并生成必要的编译文件，需要在 `configure` 命令后加上前面记录的 configure 参数。

```
cd nginx-release-1.26.2
auto/configure --prefix=/etc/nginx --sbin-path=/usr/sbin/nginx --modules-
path=/usr/lib/nginx/modules --conf-path=/etc/nginx/nginx.conf --error-log-
path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --pid-
path=/var/run/nginx.pid --lock-path=/var/run/nginx.lock --http-client-body-temp-
path=/var/cache/nginx/client_temp --http-proxy-temp-path=/var/cache/nginx/proxy_temp
--http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp --http-uwsgi-temp-
path=/var/cache/nginx/uwsgi_temp --http-scgi-temp-path=/var/cache/nginx/scgi_temp --
user=nginx --group=nginx --with-compat --with-file-aio --with-threads --with-
http_addition_module --with-http_auth_request_module --with-http_dav_module --with-
http_flv_module --with-http_gunzip_module --with-http_gzip_static_module --with-
http_mp4_module --with-http_random_index_module --with-http_realip_module --with-
http_secure_link_module --with-http_slice_module --with-http_ssl_module --with-
http_stub_status_module --with-http_sub_module --with-http_v2_module --with-
http_v3_module --with-mail --with-mail_ssl_module --with-stream --with-
stream_realip_module --with-stream_ssl_module --with-stream_ssl_preread_module --
with-cc-opt='-g -O2 -ffile-prefix-map=/data/builder/debuild/nginx-
1.26.2/debian/debuild-base/nginx-1.26.2=. -flto=auto -ffat-lto-objects -flto=auto -
ffat-lto-objects -fstack-protector-strong -Wformat -Werror=format-security -Wp,-
D_FORTIFY_SOURCE=2 -fPIC' --with-ld-opt='-Wl,-Bsymbolic-functions -flto=auto -ffat-
lto-objects -flto=auto -Wl,-z,relro -Wl,-z,now -Wl,--as-needed -pie'
```

5. 将 nginx-otel 的源码 clone 到本地。

```
git clone https://github.com/nginxinc/nginx-otel.git
```

6. 进入 nginx-otel 的文件夹，创建编译文件夹。

```
cd nginx-otel
mkdir build
cd build
```

7. 启用相关配置项，开始编译。

> **说明：**
> 此处的 `/path/to/configured/nginx/objs` 是之前生成的 Nginx 编译文件中 `objs` 文件夹的路径，可以在 Nginx 源码文件夹中找到。

```
cmake -DNGX_OTEL_NGINX_BUILD_DIR=/path/to/configured/nginx/objs ..
make
```

编译完成后，会在 `nginx-otel/build` 文件夹下生成动态模块文件 `ngx_otel_module.so`，以下是 `nginx-otel/build` 文件夹下的所有文件示例。

```
CMakeCache.txt  CMakeFiles  cmake_install.cmake  _deps  gens  http_archives
Makefile  ngx_otel_module.so  opentelemetry  protos
```

8. 将 `ngx_otel_module.so` 文件移动到 `/etc/nginx/modules` 文件夹下（本文使用 Ubuntu 系统中 Nginx 的配置路径，其他系统的配置路径可能不同）。

```
mv ngx_otel_module.so /etc/nginx/modules
```

配置 nginx-otel

修改 nginx 配置文件 `/etc/nginx/nginx.conf`。

```
load_module modules/ngx_otel_module.so; # 加载动态模块
...
http {
    ...

    otel_exporter {
        endpoint ${ENDPOINT}; # APM 接入点
    }

    otel_trace on;                      # 开启链路追踪
    otel_service_name ${SERVICE_NAME};  # APM 应用名
    otel_trace_context propagate;       # 将 W3C 请求头注入 TraceID，需要启用
    otel_resource_attr token ${TOKEN};  # 接入 token
    otel_resource_attr host.name ${HOST.NAME}; # 实例的主机名
    ...
}
```

对应的字段说明如下：

- **${SERVICE_NAME}**：应用名，多个使用相同应用名接入的进程，在 APM 中会表现为相同应用下的多个实例。建议直接命名为 `nginx` 。
- **${ENDPOINT}**：前置步骤中拿到的接入点。请注意该接入点为 Go 应用的 OpenTelemetry 接入点，不带 http 前缀。
- **${TOKEN}**：前置步骤中拿到的 token。
- **${HOST.NAME}**：该应用实例的主机名，是应用实例的唯一标识。通常情况下可以设置为该 Nginx 实例的 IP 地址。

完成配置后，通过 `sudo nginx -s reload` 重新加载配置文件，或者通过 `service nginx restart` 重启 Nginx 服务，即可完成接入。

接入 LLM 应用

最近更新时间：2025-04-30 10:12:42

大语言模型近年来呈现出快速增长的趋势，随着 DeepSeek 的强势出圈，AI 应用的门槛进一步降低，为各行业引入高效且低成本的解决方案，推动了人工智能落地应用的爆炸式发展。
大语言模型应用的运行涉及多个组件和复杂的交互过程，应用性能监控 APM 通过分布式追踪技术，清晰地展示请求在各个组件之间的调用链路，一旦出现故障，可以快速定位到问题所在的具体环节。同时，APM 可以实时监测模型应用的运行状态，及时发现异常行为并发出告警，使运维人员及时采取措施，避免问题扩大化。

支持的 LLM 组件与框架

LLM 应用可以基于 Traceloop 开源的 OpenLLMetry 项目进行接入，该项目完全兼容 OpenTelemetry 协议标准，能够和其他使用 OpenTelemetry 方案接入的应用实现链路信息互通。
OpenLLMetry 项目支持众多 LLM 组件与框架的自动埋点，请前往 项目主页 获取所有支持的组件与框架，以下将列出其中一部分供您参考。

支持的 LLM 组件与框架	链接
支持的 LLM 框架	Ollama、LlamaIndex、LangChain、Haystack、LiteLLM、CrewAI
支持的向量数据库	Chroma、Pinecone、Qdrant、Weaviate、Milvus、Marqo、LanceDB
支持的 LLM 厂商	VertexAI、OpenAI、MistralAI、Anthropic、Cohere、HuggingFace、Replicate 等

接入流程

获取接入点与 token

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 在右侧弹出的**数据接入**抽屉框中，单击 **Python** 语言。
4. 在**接入 Python 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 **OpenTelemetry**。
6. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

安装 traceloop-sdk

通过 `pip` 命令安装 traceloop-sdk，其中包含了对 OpenLLMetry 以及 OpenTelemetry-SDK 的相关依赖。

```
pip install traceloop-sdk
```

初始化配置

在 Python 项目的入口文件中引入 traceloop 和 OpenTelemetry-SDK，并根据前置步骤中获得的接入点以及 token 进行初始化。

> **说明：**
> 在需要埋点的目标组件导入声明之前，就必须完成初始化，建议尽可能在入口文件的起始处添加初始化代码。

```
from traceloop.sdk import Traceloop
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.baggage.propagation import W3CBaggagePropagator
# 配置链路数据发送后端
# 尽可能在文件开头声明，一定要在目标埋点组件的导入声明前初始化
Traceloop.init(
    app_name="<service_name>",
    resource_attributes={
        "token": "<token>",
        "host.name": "<host_name>"
    },
    propagator=W3CBaggagePropagator(),
    processor=BatchSpanProcessor(
        OTLPSpanExporter(endpoint="<endpoint>")
    )
)
```

对应的字段说明如下，请根据实际情况进行替换。

- `<service_name>` ：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。
- `<host.name>`：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。

示例代码

以 Ollama 搭建的本地大模型应用为例，可以参考如下代码完成初始化配置。运行示例代码，需要安装 Ollama，并下载相关模型，本文使用的模型为开源推理模型 deepseek-r1:1.5b。

```
from traceloop.sdk import Traceloop
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.baggage.propagation import W3CBaggagePropagator

Traceloop.init(
    app_name="<service_name>",
    resource_attributes={
        "token": "<token>",
        "host.name": "<host_name>"
```

```
    },
    propagator=W3CBaggagePropagator(),
    processor=BatchSpanProcessor(
        OTLPSpanExporter(endpoint="<endpoint>")
    )
)

from ollama import chat
from ollama import ChatResponse

def ollama_chat():
    response: ChatResponse = chat(model='deepseek-r1:1.5b', messages=[
    {
        'role': 'user',
        'content': 'Tell a joke of OpenTelemetry',
    },
    ])
    print(response.message.content)

if __name__ == "__main__":
    ollama_chat()
```

至此，LLM 应用已经完成了接入 APM 的所有工作。

与 OpenTelemetry 自动埋点结合

如果业务代码使用了其他与 LLM 无关的框架或组件，如 Django、Flask 等 Web 框架，或 SQLAlchemy 等 ORM 框架，需要与 OpenTelemetry 社区的自动埋点探针结合，实现对这些组件的埋点。具体的组件支持列表请参考 项目主页 。
如果要对 Flask 进行埋点，需要先安装对应的 pip package：

```
pip install opentelemetry-instrumentation-flask
```

埋点代码示例：

```
from opentelemetry.instrumentation.flask import FlaskInstrumentor

app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)

@app.route('/')
def get_response():
    # 此处为调用大模型并返回相关信息的逻辑
```

自定义埋点增强

如果需要跟踪 LLM 相关框架与类库之外的业务场景，可以参照下述内容，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 。

```python
from traceloop.sdk import Traceloop
from opentelemetry import trace # 导入链路相关 SDK
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.baggage.propagation import W3CBaggagePropagator

Traceloop.init(
    app_name="<service_name>",
    resource_attributes={
        "token": "<token>",
        "host.name": "<host_name>"
    },
    propagator=W3CBaggagePropagator(),
    processor=BatchSpanProcessor(
        OTLPSpanExporter(endpoint="<endpoint>")
    )
)

from ollama import chat
from ollama import ChatResponse

tracer = trace.get_tracer(__name__) # init tracer

def ollama_chat():
    response: ChatResponse = chat(model='deepseek-r1:1.5b', messages=[
    {
        'role': 'user',
        'content': 'Tell a joke of OpenTelemetry',
    },
    ])
    print(response.message.content)

def main():
    with tracer.start_as_current_span("entry func") as span: # create a span
        span.set_attribute("func", "main") # set span attributes
        ollama_chat()

if __name__ == "__main__":
    main()
```

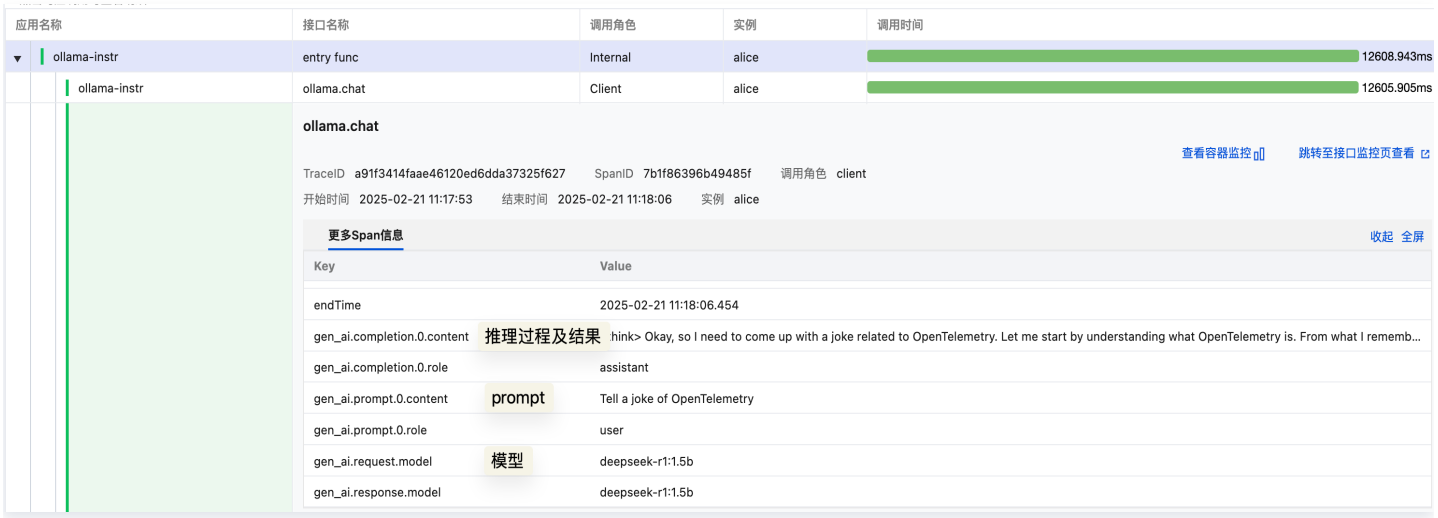
其中，`trace.get_tracer(__name__)` 获取了 OpenTelemetry Tracer 对象，用于后续的自定义埋点增强。在 `main()` 函数中，通过 `tracer.start_as_current_span()` 添加了一个新的 Span，并加入到链路上下文当中。

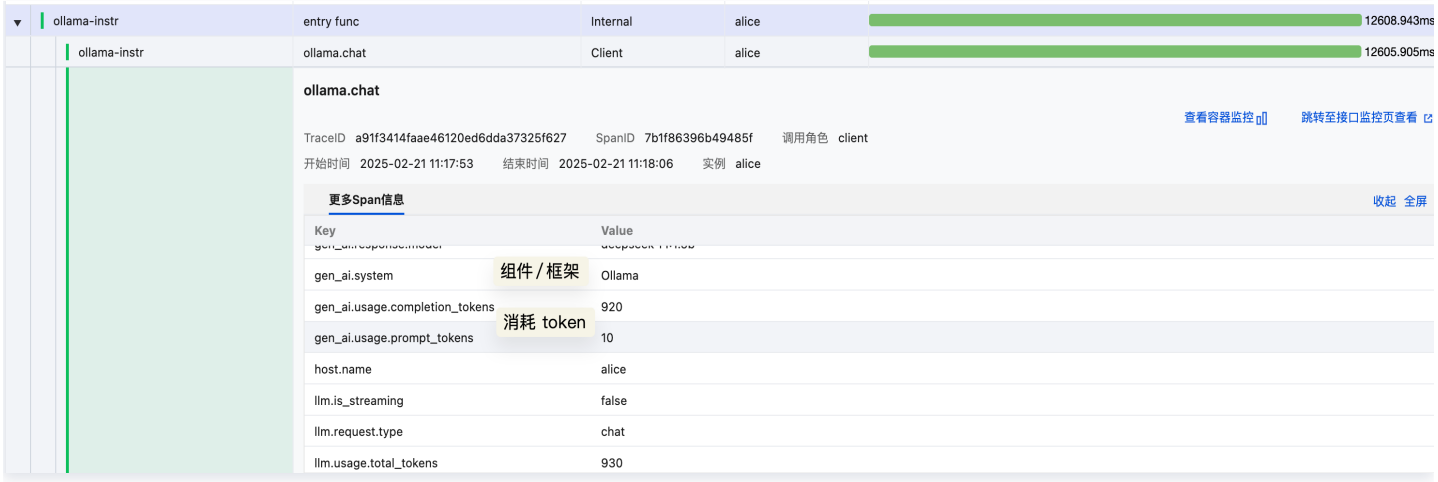
接入验证

完成接入工作后，启动 LLM 应用，应用就会向 APM 上报链路数据。在 应用性能监控 > 应用列表 页面将展示接入的应用，单击**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。

同时，在 应用性能监控 > 链路追踪 中，也能够查询到相关的 Span 记录，通过单击第一列 traceID 对应的链接，可以进入链路详情视图分析链路的每个环节。

单击其中的 Span，能够获取更多详细信息，例如下图：





接入其他语言编写的应用

最近更新时间：2024-09-20 10:24:53

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，基于腾讯云控制台以及云 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. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 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**。
4. 选择**接入协议类型**为 **OpenTelemetry**。
5. **上报方式**选择您所想要的上报方式，获取您的**接入点**和 **Token**。

> **说明：**
> - 内网上报：使用此上报方式，您的服务需运行在腾讯云 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腾讯云 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费用。

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，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。

安装 tencent-opentelemetry-operator

最近更新时间：2025-02-28 11:07:32

对于部署在 Kubernetes 上的应用，腾讯云可观测团队提供了 Operator 方案：tencent-opentelemetry-operator， 此方案在社区 opentelemetry-operator 基础上构建，可以实现探针自动注入，方便应用快速接入 APM。目前 tencent-opentelemetry-operator 支持的编程语言包括 Java、Python、Node.js 和 .Net。

> **说明：**
> tencent-opentelemetry-operator 支持 Kubernetes 版本1.19及以上版本。

配置项说明

tencent-opentelemetry-operator 通过 Helm 部署安装，所有的配置项都集中于 `values.yaml` 。请注意 YAML 文件中的参数存在层级关系，请参考如下 YAML 片段：

```
env:
  TKE_CLUSTER_ID: "cls-ky8nmlra"
  TKE_REGION: "ap-guangzhou"
  APM_ENDPOINT: "http://pl.ap-guangzhou.apm.tencentcs.com:4317"
  APM_TOKEN: "apmdemotoken"
```

必填字段

参数	描述
env.TKE_CLUSTER_ID	TKE 集群 ID。对于非 TKE 集群，请设置为 `"N/A"` 。
env.TKE_REGION	TKE 集群所在地域，例如 ap-guangzhou，详情请参考 CVM 地域和可用区 的取值。对于非 TKE 集群，请设置为 `"N/A"` 。
env.ENDPOINT	APM 接入点。每个集群只能使用唯一的 APM 接入点，请从业务系统获取接入点。如果使用公网接入点，需要同时设置 `env.FROM_INTERNET: "true"`
env.APM_TOKEN	集群默认的 APM token。APM token 代表了需要接入的业务系统，请从业务系统中获取 token。可以在工作负载中指定其他业务系统的 token，以覆盖集群默认的 token。

选填字段

参数	描述
env.JAVA_INSTR_VERSION	Java 探针版本，可以设置为 `latest` （默认）或具体的版本号，非必要情况下不推荐设置此字段。
env.PYTHON_INSTR_VERSION	Python 探针版本，可以设置为 `latest` （默认）或具体的版本号，非必要情况下不推荐设置此字段。
env.NODEJS_INSTR_VERSION	Node.js 探针版本，可以设置为 `latest` （默认）或具体的版本号，非必要情况下不推荐设置此字段。

env.DOTNET_INSTR_VERSION	.Net 探针版本，可以设置为 `latest`（默认）或具体的版本，非必要情况下不推荐设置此字段。
env.INTL_SITE	在国际站需要设置为 `"true"`。
env.FROM_INTERNET	如果从公网接入，请设置为 `"true"`，（同时需要将 env.ENDPOINT 设置为公网接入点）。

说明：
如果需要指定具体的探针版本号，请前往 探针（Agent）版本信息 获取版本号。

安装方式

通过 APM 控制台一键安装（推荐）

由于配置项的填写比较复杂，推荐您使用 APM 控制台的一键安装 tencent-opentelemetry-operator 功能，以简化安装步骤。

1. 登录 腾讯云可观测平台 控制台。
2. 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**。
3. 选择需要接入的语言，选择 **TKE 环境自动接入**的上报方式。
4. 单击**一键安装 Operator**。
5. 在弹出对话框中，选择对应的上报地域、默认业务系统、TKE 所在地域、TKE 集群，单击**安装**后即可完成安装。

说明：
- 仅支持 TKE 标准集群和 TKE Serverless 集群，暂不支持 TKE 边缘集群和 TKE 注册集群。
- 通过 APM 控制台一键安装的 tencent-opentelemetry-operator，会被安装到 kube-system 命名空间，如果需要修改相关配置项，可以在控制台对同一个 TKE 集群进行更新操作。

通过 TKE 应用市场安装

1. 登录 容器服务 控制台。
2. 在左侧菜单栏中选择**运维中心** > **应用市场**，搜索 **tencent-opentelemetry-operator** 并单击进入。
3. 单击**创建应用**，选择需要安装的 TKE 集群，填入必要参数，即可完成安装。

说明：
- 仅支持 TKE 标准集群和 TKE Serverless 集群，暂不支持 TKE 边缘集群和 TKE 注册集群。
- 通过 TKE 应用市场安装 tencent-opentelemetry-operator，可以安装在任何命名空间。在同一个 TKE 集群中，只能安装最多一个 tencent-opentelemetry-operator。

通用 K8s 集群安装

tencent-opentelemetry-operator 支持通用 K8s 集群以及混合云场景。对于部署在线下 IDC 以及其他云平台的 K8s 集群，只要 K8s 的版本符合要求，并且 APM 服务端之间的网络可达，就可以通过 Operator 模式实现快速接入。

1. 安装 kubectl 和 helm CLI，安装方式请参考 安装 kubectl 和 安装 helm CLI。
2. 下载 Chart 包 和配置文件 values.yaml。您也可以使用 wget 命令下载，对应的地址如下：

- https://operator-1258344699.cos.ap-guangzhou.myqcloud.com/tencent-opentelemetry-operator-internet.tgz
- https://operator-1258344699.cos.ap-guangzhou.myqcloud.com/values.yaml

3. 参考 配置项说明，在 values. yaml 中填入必要的字段。

> **说明：**
> - env.TKE_CLUSTER_ID 和 env.TKE_REGION 只适用于 TKE 集群，因此请求这两个配置项设置为 `"N/A"` 。
> - 如果从公网接入，请将 env.FROM_INTERNET 设置为 `"true"` ，同时将 env.ENDPOINT 设置为公网接入点。

4. 使用本地文件安装 operator，其中 `my-release` 为 Chart 名，可以自定义。 `--values` 代表 value.yaml 的文件路径。

```
helm install my-release /path/to/your/chart.tgz --values /path/to/your/values.yaml
```

> **说明：**
> 在通用 K8s 集群安装 tencent-opentelemetry-operator 的时候，需要有公网访问权限，否则无法拉取 Operator 镜像。

接入应用

安装完 tencent-opentelemetry-operator 后，在需要接入 APM 的工作负载中添加相关 annotation，就可以实现探针自动注入，并向 APM 上报监控数据。请参考如下文档完成应用接入：

- K8s 环境自动接入 Java 应用（推荐）
- K8s 环境自动接入 Python 应用（推荐）
- K8s 环境自动接入 Node.js 应用（推荐）
- K8s 环境自动接入 .NET 应用（推荐）

升级探针版本

最近更新时间：2025-02-18 09:21:32

通过 Operator 方案快速接入

对于部署在 Kubernetes 上的应用，腾讯云可观测团队提供了 Operator 方案，用于快速接入应用，详情请参见 安装 tencent-opentelemetry-operator 。通过 Operator 方案接入的应用默认使用探针自动更新策略，用户无需关注探针升级问题，在推送新版本探针之前，APM 团队会对探针进行多轮稳定性测试，保障探针的稳定性与兼容性。
如果您已经在 Operator 或工作负载级别主动指定探针版本，建议去掉用于指定探针版本的配置项，回到自动更新模式。如果确实有必要指定具体的探针版本，请参见 探针（Agent）版本信息 获取更新的版本号。

其他接入方式

请参考各接入文档的操作步骤，下载并更新探针包，或者引入更新版本的依赖。
如果您使用的是腾讯云增强版 OpenTelemetry Java 探针，可以前往 探针（Agent）版本信息 下载新版本探针。

控制台操作指南
资源管理
新建业务系统

最近更新时间：2025-02-13 10:33:52

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. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控 > 资源管理**，选择需要新建业务系统的地域。
3. 在**业务系统管理**页，单击**新建**，进入创建业务系统弹框。
4. 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。
开启免费模式	开启免费模式后，该业务系统将永久免费。关于免费模式的限制，请参见 关于免费模式的使用限制 。
计费模式	支持**按量付费**和**预付费**。
上报地域	各地域数据隔离，业务系统创建后不可更改。
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。
添加标签	应用性能监控结合腾讯云资源标签功能，为您提供按标签授予子账号权限和按标签分账功能。请参见 访问管理 设置标签。

5. 配置完成后，勾选服务条款，然后单击**确定**即可。

用量分析

最近更新时间：2025-02-14 15:32:42

通过**用量分析**页面，您可以分析每个业务系统产生的 APM 用量，以更好的评估 APM 费用。
APM 用量通过 **Span 上报数（条）**、**Span 存储量（条*天）**、**探针在线时长（探针个数*小时）**三个维度体现。为了方便评估 APM 费用，系统基于 APM 的真实使用量进行统计，不论采用哪种计费模式，接入 APM 的应用都会同时产生三个维度的数据。

> **说明：**
> 当天的用量统计会有1小时左右的延迟。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控** > **资源管理**。
3. 在资源管理页面，选择**用量分析**，在此页面可查询最近30天内的 APM 用量。

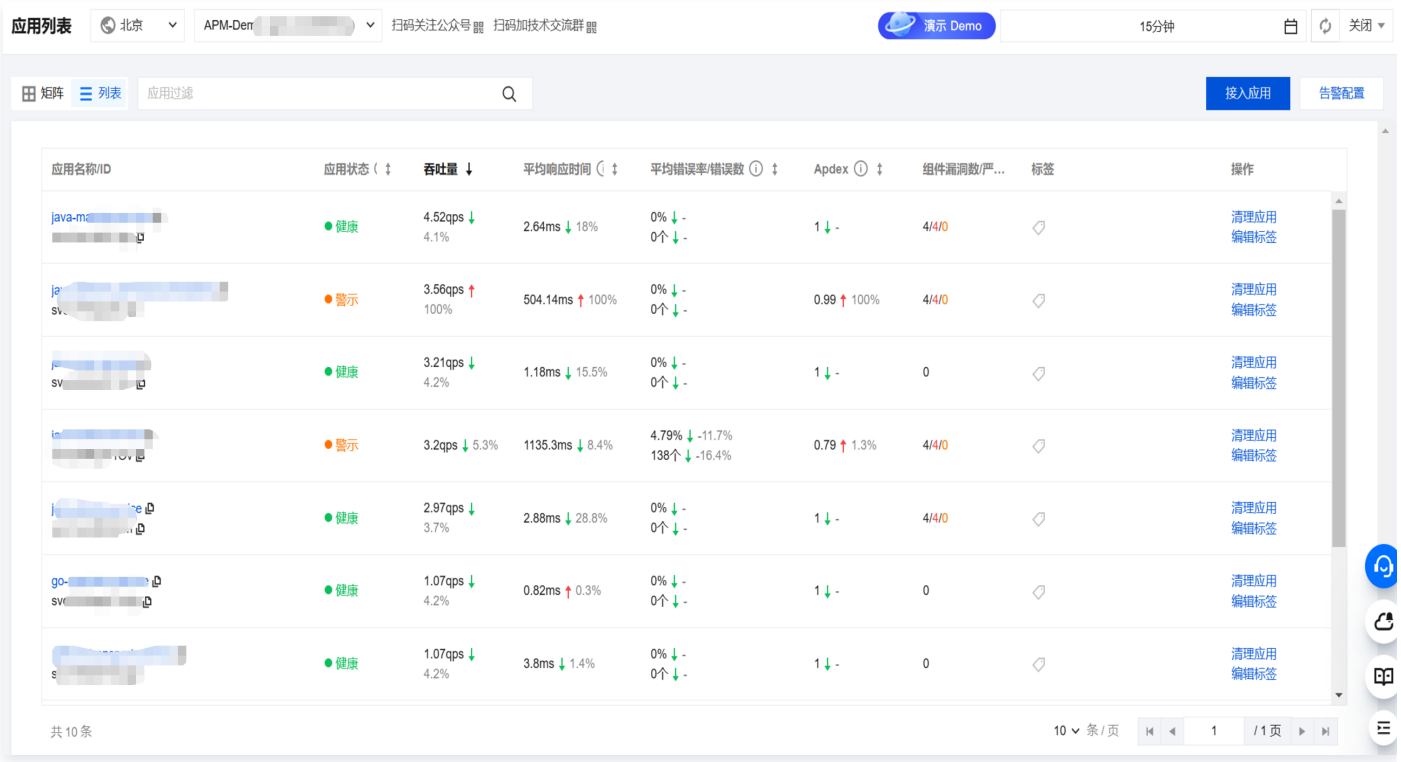
应用监控
应用列表

最近更新时间：2024-11-25 11:49:32

本文将为您介绍如何使用应用列表。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控 > 应用列表**。



功能说明

- **应用过滤：**可以基于应用名、应用 ID 和应用标签进行过滤。
- **清理应用：**清理应用功能仅针对已经不再上报监控数据的应用，请先确保该应用的探针已经成功卸载（如果通过代码上报，请删除相关逻辑）。在执行完清理操作后，如果 APM 收到监控数据上报，会再次根据应用名创建应用。
- **编辑标签：**通过该功能，可以为应用关联多个标签，用于应用查询以及细粒度的权限配置。标签键和标签值引用自腾讯云统一的标签中心，如果需要维护标签，请前往 标签列表 。

应用详情

最近更新时间：2024-11-25 11:49:32

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. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控 > 应用列表**。
3. 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。

容器监控

对于部署在容器服务 TKE 的应用，该页面可以展示应用所关联的 Deployment 的监控信息。如果应用通过 tencent-opentelemetry-operator 接入，不需要进行额外的配置；如果应用通过其他方式接入，请参见 自定义应用实例属性 设置必要的实例属性字段。

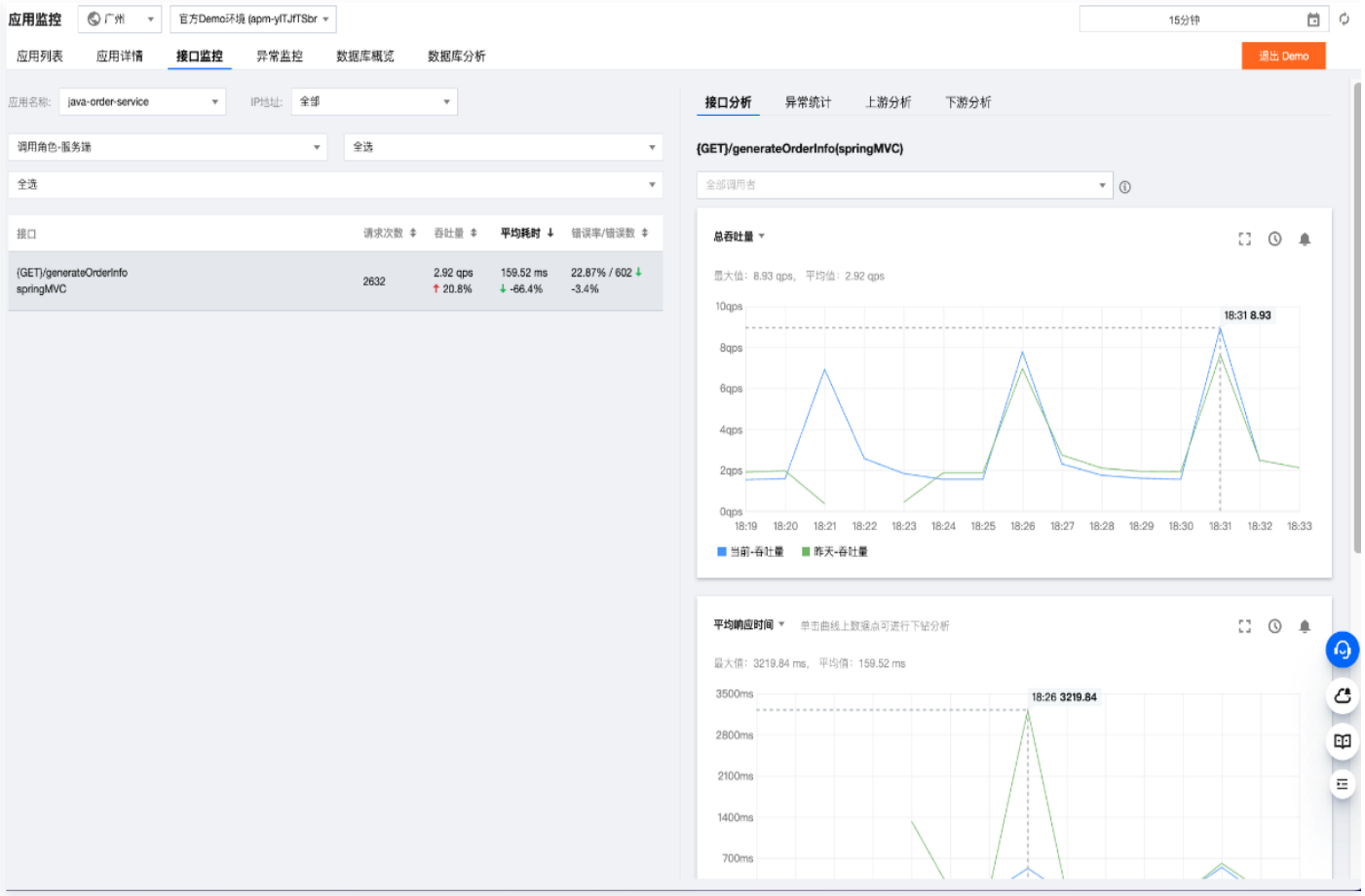
接口监控

最近更新时间：2024-10-14 20:24:01

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. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控 > 应用监控 > 接口监控**。
3. 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。


接口分析
异常统计
上游分析
下游分析
请选择
总吞吐量
最大值：40.97 qps，平均值：38.31 qps
11:15 40.97
45qps
36qps
27qps
18qps
9qps
0qps
07:48 08:11 08:34 08:57 09:20 09:43 10:06 10:29 10:52 11:15 11:38 12:01 12:24 12:47 13:10 13:33 13:56 14:19 14:42 15:05 15:28 15:51 16:14 16:37 17:00 17:23 17:46 18:09 18:32 18:55
当前-吞吐量
昨天-吞吐量
平均响应时间
最大值：271.05 ms，平均值：205.19 ms
12:12 271.05
300ms
240ms
180ms
120ms
60ms
0ms
07:48 08:11 08:34 08:57 09:20 09:43 10:06 10:29 10:52 11:15 11:38 12:01 12:24 12:47 13:10 13:33 13:56 14:19 14:42 15:05 15:28 15:51 16:14 16:37 17:00 17:23 17:46 18:09 18:32 18:55
当前-平均响应时间
昨天-平均响应时间
查看最新请求列表

异常监控

最近更新时间：2024-08-16 15:02:21

本文将为您介绍如何查看应用异常情况，进行异常分析。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控 > 应用监控 > 异常监控**。
3. 进入异常监控页面，即可查看相关模块信息。

模块说明

异常列表

在异常监控页面左侧所选时间内的所有异常，包括服务异常类型、接口、异常发生次数等信息。

异常趋势

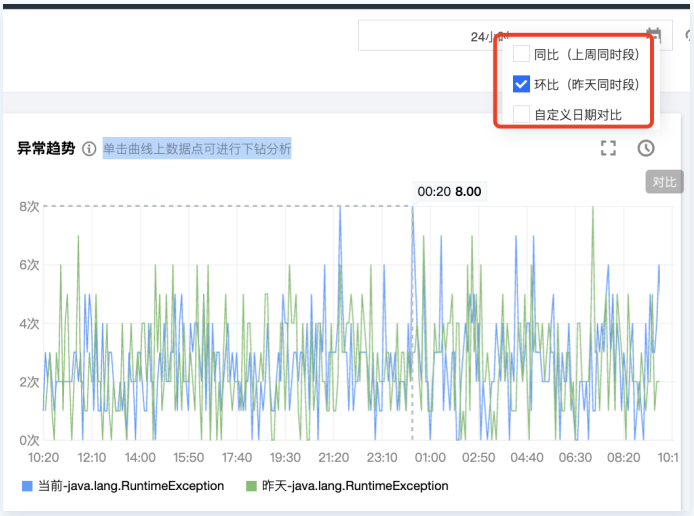
右侧图表会显示出异常趋势所筛选时间粒度下异常的发生次数。单击曲线上的数据点可进行下钻分析异常情况。

异常分析

单击相关应用，在异常监控页面右侧将会展示该异常的异常次数时序曲线，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异常。



容器监控

最近更新时间：2025-02-21 09:14:33

对于部署在容器服务 TKE 的应用，可以关联查询容器监控数据，包括事件和各类监控指标。

说明：

- 容器监控数据从容器服务 TKE 获取，暂不支持自建 Kubernetes 集群。
- 应用所在的 TKE 集群必须是当前主账号的资源，因此该功能暂不支持跨账号上报场景。
- 目前仅支持通过 OpenTelemetry 方案接入的应用，暂不支持 Skywalking 方案。

前提条件

对于 TKE 环境通过 tencent-opentelemetry-operator 自动接入的应用，可以直接使用容器监控功能。使用其他方式接入的应用，需要通过实例属性填入必要的 TKE 环境信息。必要的环境信息包括：

字段名	说明
k8s.region	TKE 集群所在地域，例如 ap-guangzhou，详情请参见 CVM 地域和可用区 的取值
k8s.cluster.id	TKE 集群 ID
k8s.deployment.name	应用所在的 Deployment
k8s.namespace.name	应用所在的命名空间
k8s.pod.name	应用所在的 Pod 的名字，该变量在 Pod 创建后才能确定，可以通过 Downward API 获取

关于自定义应用实例属性，请参见 自定义应用实例属性 。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控** > **应用列表**，单击应用的名称/ID，选择**容器监控**。

   在**容器监控**页面将展示与该应用关联的工作负载，您可以查看该工作负载的监控图表。如果该应用的多个实例分布在不同的工作负载中，请在下拉框中选择需要监控的工作负载。

应用诊断
性能剖析

最近更新时间：2024-10-31 17:12:42

性能剖析能力基于 async-profiler 技术实现，以极低的性能开销，生成性能剖析火焰图，帮助用户直观的分析 CPU/内存飙升的原因，快速定位应用性能瓶颈。相比直接使用社区的 async-profiler 方案，应用性能监控提供的性能剖析能力免去了工具的安装、命令执行、剖析结果下载等复杂的操作，大幅提升了排查应用性能问题的效率。
根据反复性能测试的结果，对一个典型的微服务应用执行性能剖析采集，采集过程中的 CPU 开销在5%以下，内存开销在50M 以下，对于TPS和响应耗时基本没有影响。

操作前提

- 应用使用腾讯云增强版 Java 探针接入，探针版本 1.16-2024030510 及以上。
- 运行环境需要安装 OpenJDK 或其他基于 HotSpot JVM 开发的 JDK，暂不支持仅安装了 JRE 的环境。
- 支持 Linux(x64, arm64) 操作系统，暂不支持 macOS 和 Windows。
- 不推荐使用 Java 8u352 以下的 JDK，可能存在 内存崩溃 风险。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏选择点击**应用性能监控 > 应用诊断 > 性能剖析**页面。
3. 在页面左侧的实例列表中找到需要进行性能剖析的实例，点击**采集**。
4. 在弹出对话框中，选择数据采集时长和采集类型，点击**确认**。
5. 采集任务提交后，可以在页面右侧的性能剖析记录中查询剖析结果，当采集状态为**采集完成**时，点击**查看性能剖析**，在弹出页面查看性能剖析火焰图。

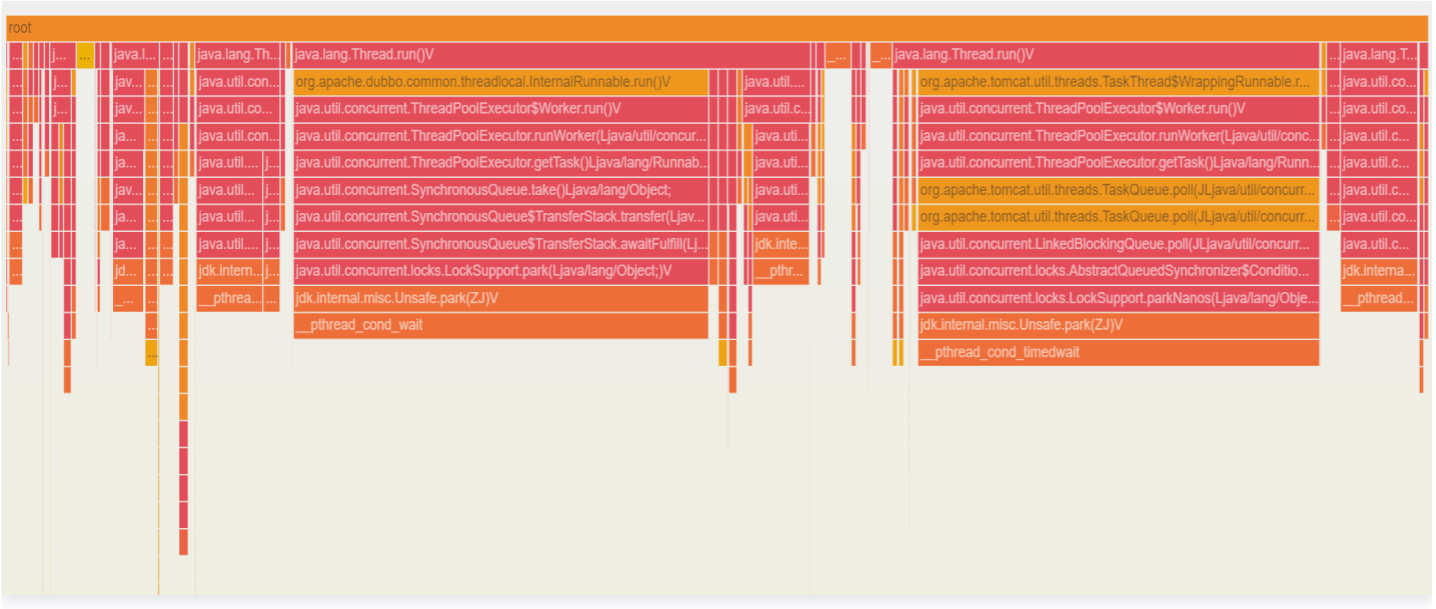
选项名	说明
数据采集时长	从应用实例收到采集任务开始，采集剖析数据的时间长度。目前支持5秒、10秒、30秒、3分钟、5分钟。
采集类型	采集类型决定了剖析数据代表的性能指标，目前支持 CPU、耗时、内存三种采集类型。 • CPU：代表 CPU 执行代码块所花费的时间。 • 耗时：耗时基于墙钟时间（Wall Clock）进行统计，从进入到退出方法所经过的实际时间。所有等待时间、锁定和线程同步的时间都包括在内，所以墙钟时间不会短于 CPU 时间。 • 内存：代表内存分配。

性能剖析火焰图使用说明

性能剖析基于抽样获取剖析数据，性能指标通过函数在抽样中出现的频率来体现，并不是绝对值，所以需要重点关注方法（函数）之间的相对占比。在火焰图中，Y 轴代表方法（函数）栈的深度，X 轴代表被抽样的次数。以 CPU 时间为例，一个方法（函数）的宽度越宽，CPU 执行代码块所花费的时间就越长。
在分析火焰图的时候，建议从深度最深的方法（函数）开始分析。栈最深，出现在火焰图中的位置会越靠近下方，也就是“火苗”的位置（不同于自然界的火焰，在 APM 目前使用的火焰图中，火焰朝向下方）。

> ⚠ **注意：**

火苗越宽，代表性能上的消耗越大，所以宽火苗往往是引发性能问题的根源。



性能剖析（新版）

最近更新时间：2025-03-10 21:38:43

性能剖析能力基于 async-profiler 等技术实现，以极低的性能开销，采集剖析数据。基于预置的智能分析引擎，新版性能剖析可以直接输出性能优化建议，并提供增强版火焰图等功能强大的性能分析工具，帮助用户快速定位应用性能瓶颈。

前提条件

- 应用使用腾讯云增强版 Java 探针接入，探针版本2.3-20250131及以上。
- 运行环境需要安装 OpenJDK 或其他基于 HotSpot JVM 开发的 JDK，暂不支持仅安装了 JRE 的环境。
- 支持 Linux（x64, arm64）以及 macOS 操作系统，暂不支持 Windows。
- 不推荐使用 Java 8u352以下的 JDK，可能存在 内存崩溃 风险。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏选择**应用性能监控** > **应用诊断**，进入**性能剖析（新版）**页面。
3. 在页面左侧的实例列表中找到需要进行性能剖析的实例，单击**发起任务**。
4. 在弹出对话框中，根据需要调整采集类型等选项，单击**确认**。

选项名	说明
采集类型	目前支持 CPU 采样分析和内存采样分析两种采集类型： • CPU 采样分析：基于代码块在 CPU 上的执行时间进行剖析。 • 内存采样分析：基于内存占用进行剖析。
采样时长	采样时长代表每一次剖析采集数据的时间长度。实际的采集时间会略大于采样时长。
目标时间	目标时间代表剖析任务创建以后，第一次剖析的执行时间： • 立即执行：不等待，立即执行剖析。 • 指定时间：指定未来24小时内的任意时间点执行剖析。
执行次数	执行次数代表一个剖析任务将被执行多少次： • 单次执行：仅执行一次。 • 重复执行：剖析任务按将照固定的时间间隔被执行多次，用户需要指定总次数以及间隔时长。

5. 采集任务提交后，可以在页面右侧的性能剖析记录中查询剖析结果，当采集状态为**采集完成**时，单击**剖析结果**，在弹出页面查看性能剖析结果。剖析结果包含5个模块，通过不同的标签页进行展示。
   - **分析建议：**

     分析建议模块基于预置的智能分析引擎输出性能优化建议。针对每条性能优化建议，单击**标记**按钮，可以对优化建议的价值进行反馈，帮助 APM 提升分析建议的准确度。
   - **活跃线程：**

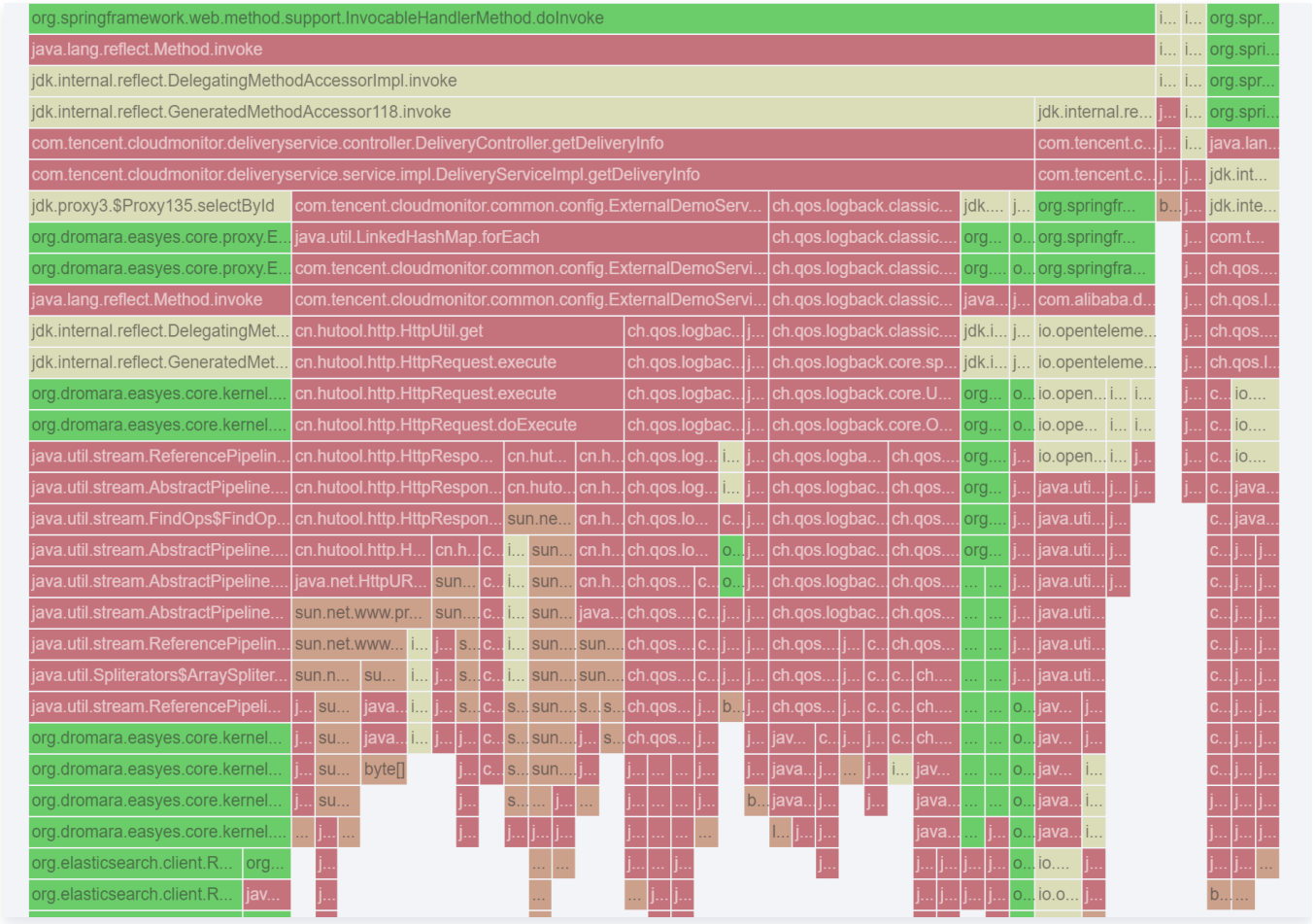
     活跃线程模块列出了每条线程被采中的次数，次数越高，代表线程的活跃度越高，可以重点关注。
   - **热点函数：**

     热点函数模块列出了每个函数被采中的次数，次数越高，代表函数的性能消耗越大，可以重点关注。
   - **火焰图：**

火焰图是一种用于可视化程序性能数据的强大工具，是一个由多个矩形条堆积而成的图形，每一层矩形条代表一个函数调用栈中的函数（在 Java 等语言中，函数可以等同于方法）。

- 水平方向表示函数的性能开销，基于不同的采样类型进行衡量，越宽表示性能开销越大。以内存采样分析为例，矩形条的宽度代表函数的内存占用情况。需要注意的是，性能开销通过函数在抽样中出现的频率来体现，并不是绝对值，所以需要重点关注它们之间的相对占比。
- 垂直方向表示函数调用栈的深度，最下面的函数是当前正在执行的函数，向上依次是它的调用者。

通过观察火焰图中宽度较大的矩形条，可以快速定位到程序中性能开销较大的热点函数。这些热点函数通常是优化的重点对象，因为它们的性能改进可能会对整个程序的性能产生较大影响。沿着火焰图的垂直方向，可以分析函数之间的调用关系，了解哪些函数调用了热点函数，以及它们的调用频率和执行时间分布，有助于深入理解程序的执行流程和性能瓶颈所在。
在分析火焰图时，建议从深度最深的函数开始分析。栈最深，出现在火焰图中的位置会越靠近下方，也就是“火苗”的位置（不同于自然界的火焰，在 APM 目前使用的火焰图中，火焰朝向下方）。苗越宽，代表性能上的消耗越大，所以宽火苗往往是引发性能问题的根源。



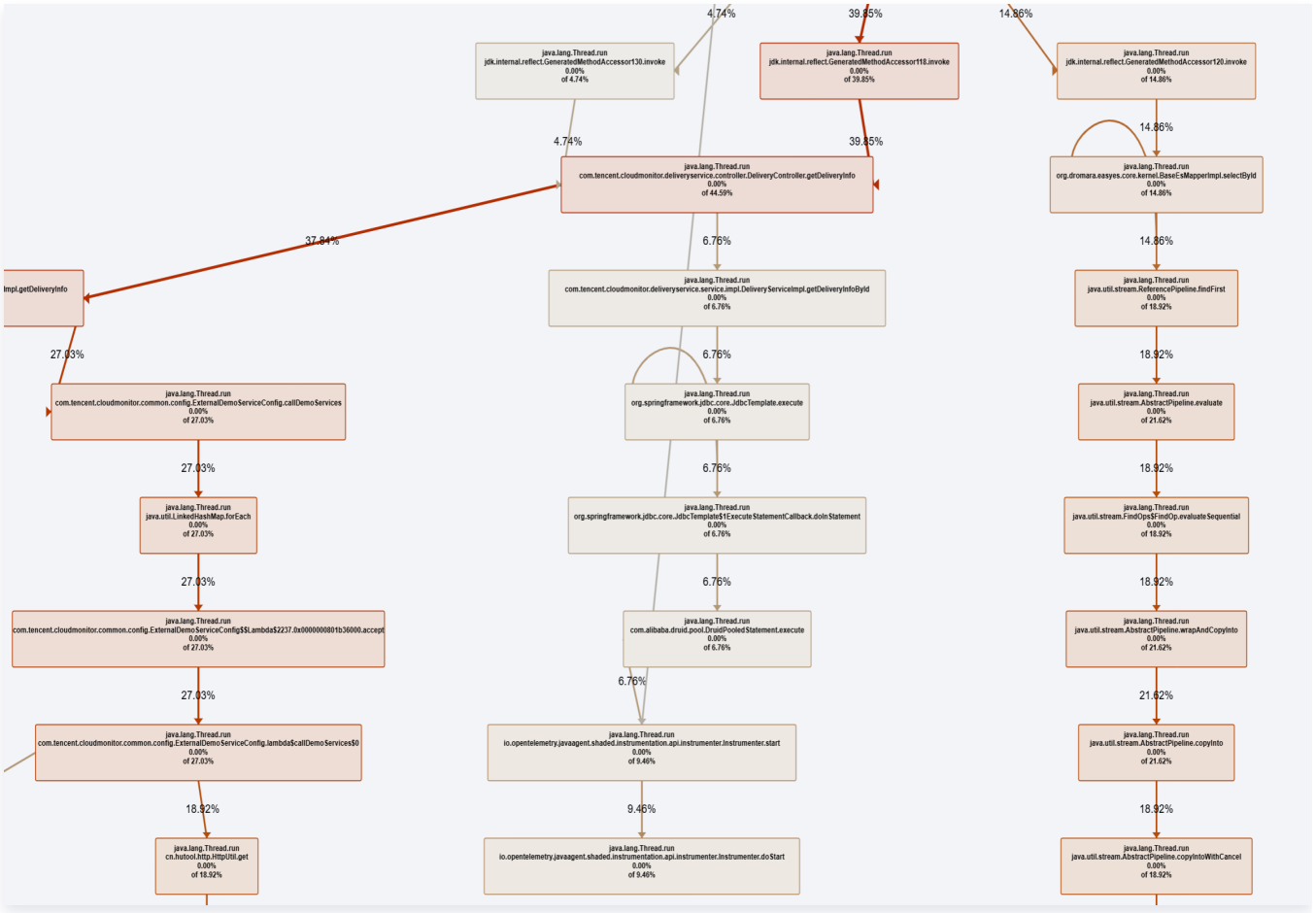
在火焰图的使用过程中，可以关注如下两类指标数据：

- **总计占比**：表示从函数调用栈的最底层到当前函数（包括当前函数及其所有子函数）的性能开销的占比。这个数据反映了当前函数及其整个调用路径在整个程序执行过程中的相对重要性。
- **自身总比**：指当前函数自身的性能开销，不包括它调用其他函数的性能开销。这个指标可以帮助开发者了解当前函数本身的性能开销，而不受到其调用的其他函数的影响。

单击火焰图中的其中一个矩形条，火焰图中将只展示在垂直方向上处于该矩形条调用路径上的矩形条。在这种情况下，该矩形条可以被称为**所选区域**，而整个应用进程所占的区域可以被称为**整体区域**（也就是最上层的 root 矩形条）。因此，**总计占比**和**自身占比**都可以基于**所选区域**或**整体区域**来计算。单击最上层的 root 矩形条，可以回到火焰图的初始状态，在这种情况下，整体区域等同于所选区域。

- **调用图：**

调用图和火焰图比较类似，同样能够体现每个函数在整个程序执行过程中的相对性能开销占比。和火焰图不同的是，当一个函数有多个调用方时，调用图更能够体现出函数之间的调用关系。但调用图不能通过矩形的宽度直观展示性能开销，因此大多数场景下，调用图都是配合火焰图来使用。



线程池分析

最近更新时间：2025-03-03 10:05:02

线程池分析功能可以展示应用核心线程池的各项关键指标，包括最大线程数、核心线程数、活跃线程数、线程池任务数、线程池大小等。

支持的线程池

腾讯云增强版 Java 探针支持线程池分析功能。

腾讯云增强版 Java 探针1.16-2024030510及以上版本

- Apache Dubbo 2.7+
- Apache Tomcat 7.0+

腾讯云增强版 Java 探针2.3-20250131及以上版本

OKhttp 2.x/3.x/4.x

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏选择**应用性能监控** > **应用诊断**，然后选择**线程池分析**。
3. 在页面左侧的实例列表中单击需要进行池程池分析的实例，在页面右侧图表即可查看线程池分析详情。

指标明细

- 最大线程数：对应 ThreadPoolExecutor.getMaximumPoolSize()
- 核心线程数：对应 ThreadPoolExecutor.getCorePoolSize()
- 活跃线程数：对应 ThreadPoolExecutor.getActiveCount()
- 线程池大小：对应 ThreadPoolExecutor.getPoolSize()
- 线程池任务数：对应 ThreadPoolExecutor.getQueue().size()
- 线程池使用率：活跃线程数/最大线程数*100%

连接池分析

最近更新时间：2025-03-03 10:05:02

连接池分析功能可以展示应用核心连接池的各项关键指标，包括当前连接数、最大连接数、活跃连接数、空闲连接数、等待连接数等。

支持的连接池

腾讯云增强版 Java 探针支持连接池分析功能。

腾讯云增强版 Java 探针1.16-2024030510及以上版本

Apache Druid 1.0+

腾讯云增强版 Java 探针2.3-20250131及以上版本

- HikariCP 3.0+
- Redisson 3.0+
- Jedis 3.0+
- Letuce 5.0+
- Apache HttpClient 4.4+
- OkHttp 2.x/3.x/4.x

操作步骤

1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择**应用性能监控** > **应用诊断**，然后选择**连接池分析**。
3. 在页面左侧的实例列表中单击需要进行连接池分析的实例，在页面右侧图表查看连接池分析详情。

指标明细

以 Druid 连接池为例，APM 会采集如下指标：

- 当前连接数：对应 DruidDataSource.getPoolingCount() + DruidDataSource.getActiveCount()
- 最大连接数：对应 DruidDataSource.getMaxActive()
- 活跃连接数：对应 DruidDataSource.getActiveCount()
- 空闲连接数：对应 DruidDataSource.getPoolingCount()
- 等待连接数：对应 DruidDataSource.getWaitThreadCount()

> **说明：**
> 对于其他连接池，APM 会按照类似的方式采集连接数指标。由于实现机制的差异，部分连接池可能不会上报等待连接数指标。

GC 日志分析

最近更新时间：2024-09-24 09:40:31

GC 日志分析功能基于 JVM 输出的 GC Log 排查有可能影响应用性能的潜在风险。

操作前提

- 腾讯云增强版 Java 探针 2.3-20240831 以上版本支持 GC 日志分析功能。
- 在使用此功能前，先确保 JVM 已经配置了打印 GC 日志相关的启动参数。

打印 GC 日志相关的启动参数

Java 8

启动参数示例如下：

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -
Xloggc:gc.log -XX:+UseGCLogFileRotation -XX:GCLogFileSize=20M -XX:NumberOfGCLogFiles=5
```

必填参数：

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc
```

Java 9 及以上版本

启动参数示例如下：

```
-Xlog:gc*:file=gc_%p_%t.log:time,pid:filecount=5,filesize=20M
```

或者直接指定GC日志绝对路径：

```
-Xlog:gc*:file=/path/to/gc.log:time,pid:filecount=5,filesize=20M
```

其中：file 为必填参数，用于指定 GC 日志文件路径。

> **说明：**
> 更多详细参数设置，请参考 Java 官方文档 。

操作步骤

1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择点击**应用性能监控 > 应用诊断 > GC日志分析**页面。
3. 在页面左侧的实例列表中找到需要进行 GC 日志分析的实例，点击**分析**，在弹出对话框中选择**数据时长**，数据时长代表从 GC 日志中向回溯的时间跨度，然后点击**确认**。
4. 分析总耗时为数秒至5分钟左右，取决于当前选择数据时长内所包含的 GC 日志内容大小，应用性能监控最多可以分析100MB 的 GC 日志数据，超出部分将被截断。
5. 在页面右侧表格中能够查到历史分析记录，分析完成的记录将展示为**采集完成**状态，点击**查看报告**获取分析结果。

分析报告解读

分析报告中包含三部分内容：

- **JVM 信息**：目前仅支持在 JDK 8 中展示，记录 JVM 版本、启动参数、系统属性等信息。
- **GC 总览（Summary）**：在选定时间段有 GC 行为时会展示。需要重点关注日志起始时间、GC 事件总数、GC 吞吐量等信息。GC 吞吐量是衡量 Java 垃圾回收器性能的重要指标，可以尝试不同的垃圾回收器，或者调整垃圾回收器的相关参数，以获取更低的 GC 次数以及更高的 GC 吞吐量。

- **分析结果（Analysis）：在选定时间段有 GC 行为时会展示。分析结果总结了 GC 方面可能存在的问题，并给出了问题的原因以及和解决建议，是分析报告的核心内容，需要重点关注结果分析中的 error 和 warn 部分。**

参考如下分析报告：

1. JVM相关

2.GC总览

3.结果分析

Report

```
gc-example.log
================================================================================
JVM:
--------------------------------------------------------------------------------
Version: Java HotSpot(TM) 64-Bit Server VM (25.102-b14) for linux-amd64 JRE (1.8.0_102-b14), b
Options: -XX:CMSInitiatingOccupancyFraction=80 -XX:+CMSParallelRemarkEnabled -XX:+DisableExpli
Memory: Memory: 4k page, physical 32878232k(18756444k free), swap 4194300k(4194300k free)
================================================================================
SUMMARY:
--------------------------------------------------------------------------------
Datestamp First: 2016-10-10T18:43:49.025-0700
Timestamp First: 1.362 secs
Datestamp Last: 2016-10-11T12:34:47.720-0700
Timestamp Last: 64260.057 secs
# GC Events: 36586
Event Types: PAR_NEW, CMS_INITIAL_MARK, CMS_CONCURRENT, CMS_REMARK, CMS_SERIAL_OLD
# Parallel Events: 36585
# Inverted Parallelism: 2
Inverted Parallelism Max: 2016-10-11T08:06:39.037-0700: 48171.374: [GC (CMS Initial Mark) [1 Cl
# Serial Events: 1
NewRatio: 54
Heap Used Max: 7092037K
Heap After GC Max: 6988066K
Heap Allocation Max: 8371584K
Metaspace Used Max: 167164K
Metaspace After GC Max: 167164K
Metaspace Allocation Max: 1204224K
GC Throughput: 96%
GC Pause Max: 7.528 secs
GC Pause Total: 2626.403 secs
================================================================================
ANALYSIS:
--------------------------------------------------------------------------------
error
--------------------------------------------------------------------------------
*The CMS_SERIAL_OLD collector is being invoked for one of the following reasons: (1) Fragmenta
*CMS promotion failed. A young generation collection is not able to complete because there is
--------------------------------------------------------------------------------
warn
--------------------------------------------------------------------------------
*CMS remark low parallelism: (1) If using JDK7 or earlier, add -XX:+CMSParallelRemarkEnabled,
*CMS initial mark low parallelism: (1) If using JDK6 or earlier, initial mark is single-thread
*Application stopped time missing. Enable with -XX:+PrintGCApplicationStoppedTime (<= JDK8) or
*Inverted parallelism. With parallel (multi-threaded) collector events, the "user" + "sys" tim
*GCLocker GC due to the following sequence of events: (1) An object allocation failed due to n
*Explicit garbage collection is disabled with -XX:+DisableExplicitGC. The JVM uses explicit ga
*Consider enabling gc log file rotation (-XX:+UseGCLogFileRotation -XX:GCLogFileSize=N[K|M|G]
--------------------------------------------------------------------------------
info
--------------------------------------------------------------------------------
*The JDK is very old (7.6 years).
*Metaspace(unlimited) = Class Metadata(unlimited) + Compressed Class Space(1024M).
*-XX:+UseParNewGC is redundant (enabled by default) and can be removed. Deprecated in JDK8 and
*The number of times an object is copied between survivor spaces is set with -XX:MaxTenuringTh
*The number of parallel garbage collection threads is set with -XX:ParallelGCThreads=N. Unless
*Consider enabling large page support, which can provide performance improvements with large h
================================================================================
```

GC 总览（Summary）

- **GC 事件总数（GC Events）：36586次。该数字偏高，可能意味着应用程序创建了大量的短暂对象，这可能会导致频繁的垃圾回收。**

- **GC 事件类型（Event Types）**：该时间内产生了 PAR_NEW、CMS_INITIAL_MARK、CMS_CONCURRENT、CMS_REMARK、CMS_SERIAL_OLD 这几种类型的 GC。
- **并行 GC 事件（Parallel Events）**：36585次。
- **串行 GC 事件（Serial Events）**：1次。
- **最大堆使用量（Heap Used Max）**：7092037 K。
- **GC 后最大堆使用量（Heap After GC Max）**：6988066 K。这个数字如果接近最大堆使用量（Heap Used Max），可能意味着应用程序的内存使用效率不高，或者存在内存泄漏。
- **最大堆分配量（Heap Allocation Max）**：8371584 K。
- **元空间最大使用量（Metaspace Used Max）**：167164 K。
- **GC 后元空间最大使用量（Metaspace After GC Max）**：167164 K。
- **最大元空间分配量（Metaspace Allocation Max）**：1204224 K。
- **GC 吞吐量（GC Throughput）**：96%。这意味着应用程序在96%的时间内在执行实际的业务逻辑，而在4%的时间内在进行垃圾回收。GC 吞吐量是一个衡量 Java 垃圾回收器性能的指标，它表示的是应用程序运行的时间占总运行时间的百分比。
- **最大 GC 暂停时间（GC Pause Max）**：7.528秒。如果单次 GC 的暂停时间过长，那么可能会影响应用程序的响应时间和延迟。例如，如果应用程序需要在1000毫秒内完成交易，那么任何一次 GC 暂停超过1000毫秒都是不可接受的。优化的方法可能包括使用并发垃圾回收器（如 G1 或 CMS），这些垃圾回收器可以在应用程序运行的同时进行垃圾回收，从而减少 GC 暂停时间。

分析结果（Analysis）

分析结果（Analysis）总结了 GC 方面可能存在的问题，并给出了问题的原因以及和解决建议，需要重点关注结果分析中的 **error** 和 **warn** 部分。例如，在如下示例中，明确指出了 CMS_SERIAL_OLD 垃圾回收器是串行运行的，回收大内存的时候可能会需要非常长的时间，建议通过调整 JVM 参数避免使用串行垃圾回收器。您可以基于分析结果对 JVM 进行调优，并通过应用性能监控的**实例监控**等功能对比调优后的效果。

ANALYSIS:

---

error

---

*The CMS_SERIAL_OLD collector is being invoked for one of the following reasons: (1) Fragmentation. The concurrent low pause collector does not compact. When fragmentation becomes an issue a serial collection compacts the heap. If the old generation has available space, the cause is likely fragmentation. Fragmentation can be avoided by increasing the heap size. (2) Metaspace class metadata or compressed class pointers allocation failure. The GC attempts to free/resize metaspace. (3) Resizing perm gen. If perm gen occupancy is near perm gen allocation, the cause is likely perm gen. Perm gen resizing can be avoided by setting the minimum perm gen size equal to the the maximum perm gen size. For example: -XX:PermSize=256M -XX:MaxPermSize=256M. (4) Undetermined reasons. Possibly the JVM requires a certain amount of heap or combination of resources that is not being met, and consequently the concurrent low pause collector is not used despite being specified with the -XX:+UseConcMarkSweepGC option. The CMS_SERIAL_OLD collector is a serial (single-threaded) collector, which means it can take a very long time to collect a large heap. For optimal performance, tune to avoid serial collections.

*CMS promotion failed. A young generation collection is not able to complete because there is not enough space in the old generation for promotion. The old generation has available space, but it is not contiguous. When fragmentation is an issue, the concurrent low pause collector invokes a slow (single-threaded) serial collector to compact the heap. Tune to avoid fragmentation: (1) Increase the heap size. (2) Use -XX:CMSInitiatingOccupancyFraction=N (default 92) to run the CMS cycle more frequently to increase sweeping of dead objects in the old generation to free lists (e.g. -XX:CMSInitiatingOccupancyFraction=85 -XX:+UseCMSInitiatingOccupancyOnly). (3) Do heap dump analysis to determine if there is unintended object retention that can be addressed to decrease heap demands. Or move to a collector that handles fragmentation more efficiently: (1) G1 compacts the young and old generations during evacuation using a multi-threaded collector. (2) Shenandoah compacts concurrently. Temporarily add -XX:PrintFLSStatistics=1 and -XX:+PrintPromotionFailure to get additional insight into fragmentation.

链路追踪

链路追踪

最近更新时间：2025-02-18 09:21:32

链路追踪是应用性能监控（APM）的重要能力，可以实现跨应用的多维度调用链检索与分析。在链路追踪主页面，您可以根据多种过滤条件组合进行调用查询，查询结果中的每一条记录代表对一次调用的记录，等同于一个 Span。您可以单击对应的 TraceID 进入链路详情视图，进一步分析链路中的每个环节。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控** > **链路追踪**。
3. 在**链路追踪**页面，选择合适的地域以及业务系统。
4. 通过右上方的时间选择器指定查询时间跨度，通过查询对话框指定更多查询条件。

字段名	说明
调用角色	调用角色等同于 Span 类型，表示这条 Span 在链路中被哪一种身份记录，包括 Server、Client、Consumer、Producer、Internal 5种类型，分别代表服务端、客户端、消费者、生产者，以及内部调用。 假设应用 A 向应用 B 发起了一次 HTTP 调用，应用 A 会记录一个调用角色为 Client 的 Span，应用 B 会记录一个调用角色为 Server 的 Span。关于 Span 类型的更多详情，请参见 OpenTelemetry 对 Span 类型的定义 。
是否为链路入口	链路入口代表一条链路中的第一个 Span，通常情况下，Trace ID 由这个 Span 生成。
应用名称	应用名称通常和服务名保持一致。应用名称在应用接入 APM 时指定，多个使用相同应用名称接入的进程，在 APM 中会表现为相同应用下的多个实例。
实例	实例名称是每个接入 APM 的进程的唯一标识。通常情况下，实例名称是进程运行环境的 IP 地址。部分接入方案可以自动设置实例名称，也可以在接入 APM 时指定，具体的设置方式请参考接入文档。
对端服务	用于标识该调用的对端服务，例如，当一个 Span 的调用角色为 Server 的时候，通常会对应另一个调用角色为 Client 的 Span，那个 Span 所在的应用，就是对端服务 。如果使用腾讯云增强版 Java 探针接入，APM 会在 Span 中自动注入对端服务。如果使用其他 OpenTelemetry 方案接入，APM 也会根据调用关系尽可能的在 Span 中注入对端服务。如果对端服务未能自动注入，您可以在上报到 APM 的 Span 中通过 `peer.service` 属性手动指定对端服务。
接口	接口名称等同于 Span 名称。
埋点组件	埋点组件用来标识上报该 Span 的框架或组件。腾讯云增强版 Java 探针和部分开源 OpenTelemetry 探针能够自动注入埋点组件。如果埋点组件未能自动注入，您可以在上报到 APM 的 Span 中通过 `component` 属性手动指定埋点组件。
状态	表示该调用是否正确。
错误类型	如果一个调用的状态为错误，错误类型用于对不同种类的错误进行区分。

包含方法栈快照的调用	如果某个接口调用耗时频繁超过慢调用监听阈值，该接口产生的调用将有机会自动生成方法栈快照。对于包含了方法栈快照的调用，可以在链路详情中查看本地方法栈的执行详情。 **说明：** 腾讯云增强版 Java 探针1.16-2023102808及以上版本支持该功能。
存在 SQL 注入风险的调用	在开启应用安全后，系统可以识别存在 SQL 注入风险的数据库调用。 **说明：** 腾讯云增强版 Java 探针2.3-20241130及以上版本支持该功能。
Span 属性	表示在 Span 中附加的其他信息，用键值对的方式表达，除了探针自动注入的 Span 属性外，用户也可以在代码中手动添加 Span 属性。

在列表中添加自定义展示列

在链路追踪列表中，可以根据实际业务需求，针对 Span 的任何属性，增加自定义列，在链路追踪结果中展示。请先前往 应用性能监控 > 系统配置 > 业务系统配置 页面，在**自定义配置**中添加需要展示的 Span 属性，这样就能通过单击 链路追踪 页面右侧的设置图标，勾选需要在列表中展示的字段。

视图功能

可以将常用的查询条件保存到视图中，方便后续使用。

1. 在查询对话框指定查询条件，然后单击**确认**。
2. 单击页面右侧的**保存视图**，可以将当前的查询条件保存到新的视图或者已有的视图中。在下次进入**链路追踪**页面时，**保存视图**左侧将展示时钟图标，单击此图标，可选择并进入任意一个已经有的视图中，将载入之前保存过的查询条件。

链路详情

最近更新时间：2025-02-20 14:09:22

链路详情功能用于分析链路中的每个环节，您可以通过瀑布视图洞察每一个环节的调用耗时以及执行状态，在链路详情页面中，所有数据都来自于同一条链路，具有唯一的 Trace ID。

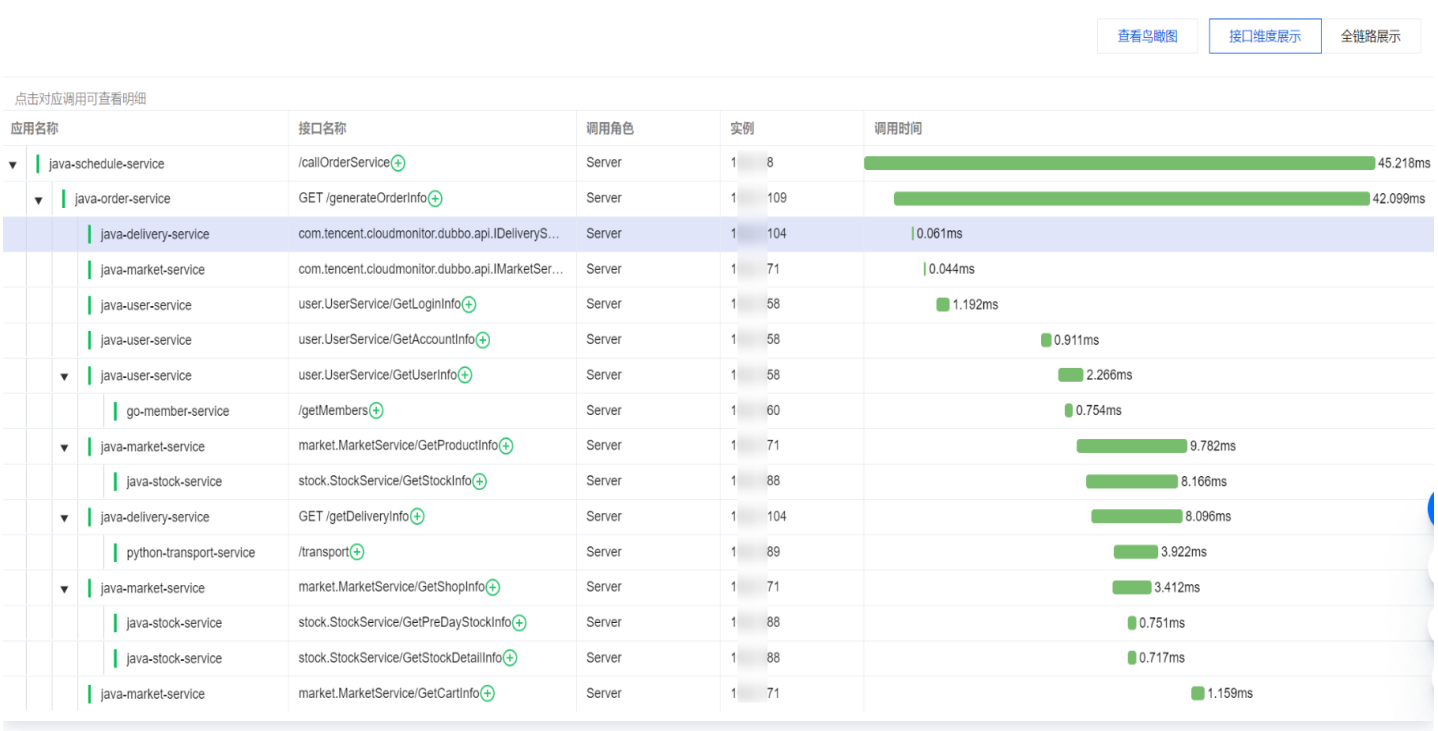
操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择**应用性能监控** > **链路追踪**。
3. 在页面顶部选择合适的地域以及业务系统。
4. 通过右上方的时间选择器指定查询时间跨度，通过查询对话框指定更多查询条件。
5. 每行查询结果的第一列展示了 Trace ID，单击 Trace ID 对应的链接，即可以进入**链路详情**页面。

链路详情介绍

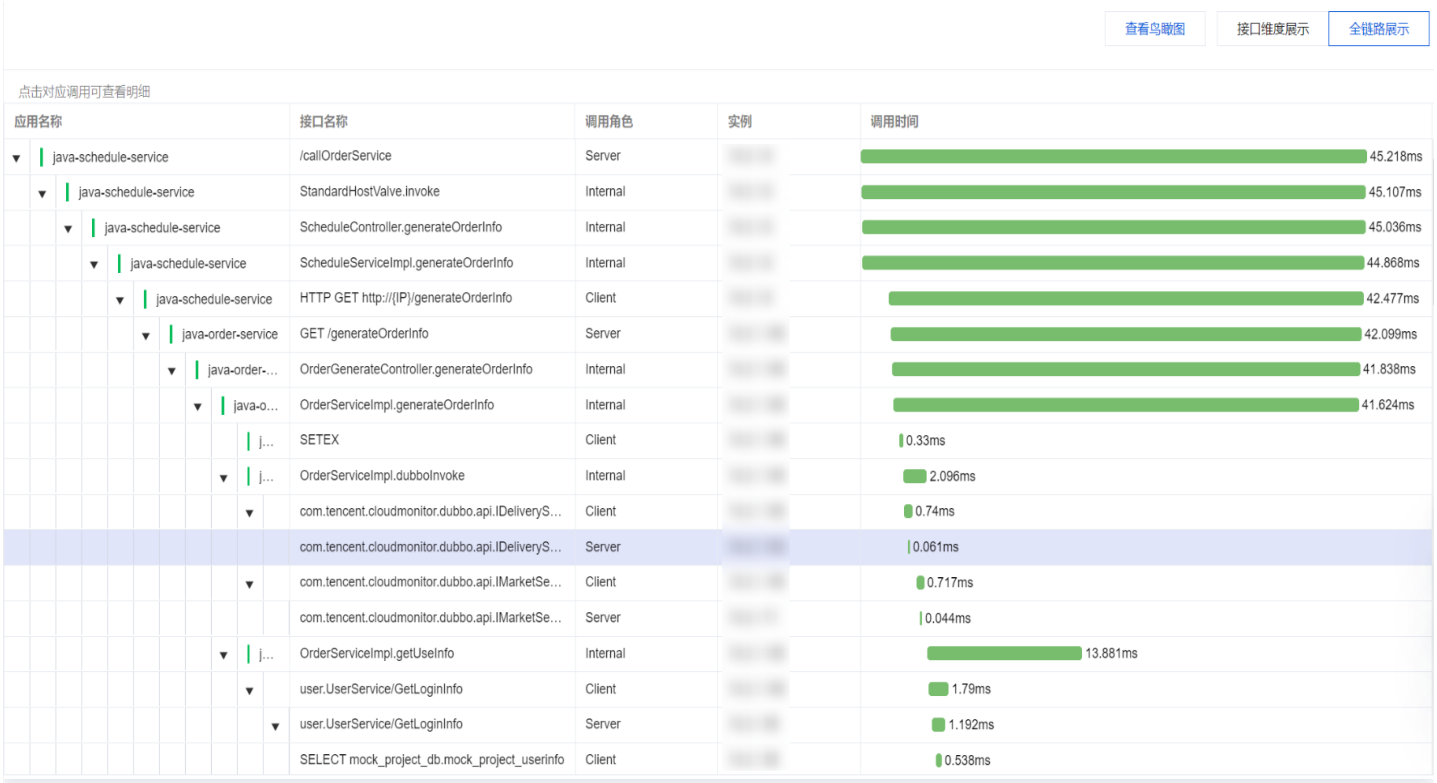
接口维度展示

单击页面右上方的**接口维度展示**，瀑布图将对展示内容进行精简，只有调用角色为 Server 或 Consumer 的 Span 会被展示。该展示方式从服务端以及消费者的角度衡量应用性能，在调用关系比较复杂时，可以从链路中快速提取重要的信息。在使用接口维度展示的情况下，可以在**接口名称**列点击绿色⊕符号，在弹出的对话框中查看该应用的所有调用。



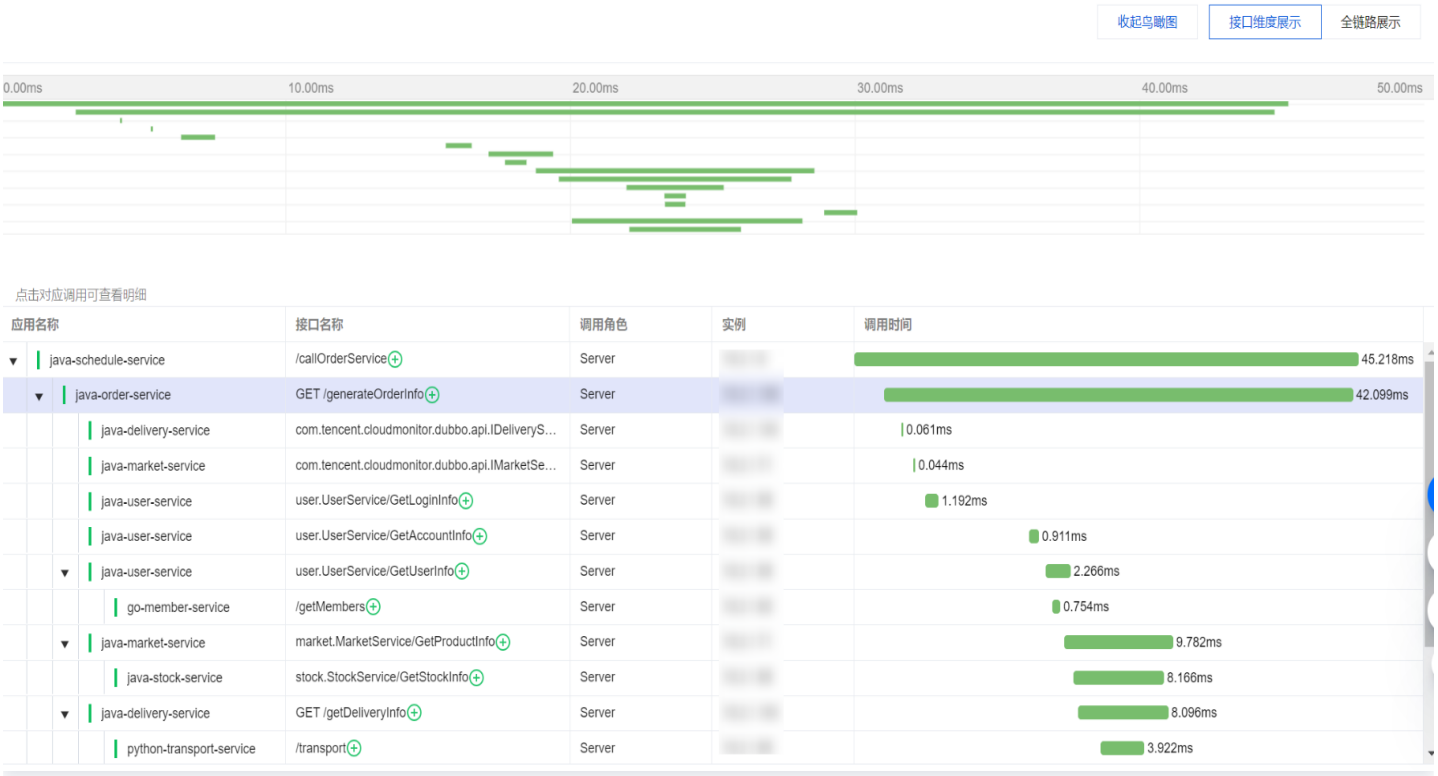
全链路展示

单击页面右上方的**全链路展示**，瀑布图将展示该链路的所有 Span。通过全链路展示，可以更全面的了解链路中每个环节的细节。



查看鸟瞰图

单击页面右上方的查看鸟瞰图，页面上方将开启鸟瞰视图，体现链路中的关键环节的调用耗时。在调用关系比较复杂的时候，通过鸟瞰图可以更快速的掌握链路的全貌。在鸟瞰视图开启的情况下，可以单击页面右上方的收起鸟瞰图关闭鸟瞰视图。

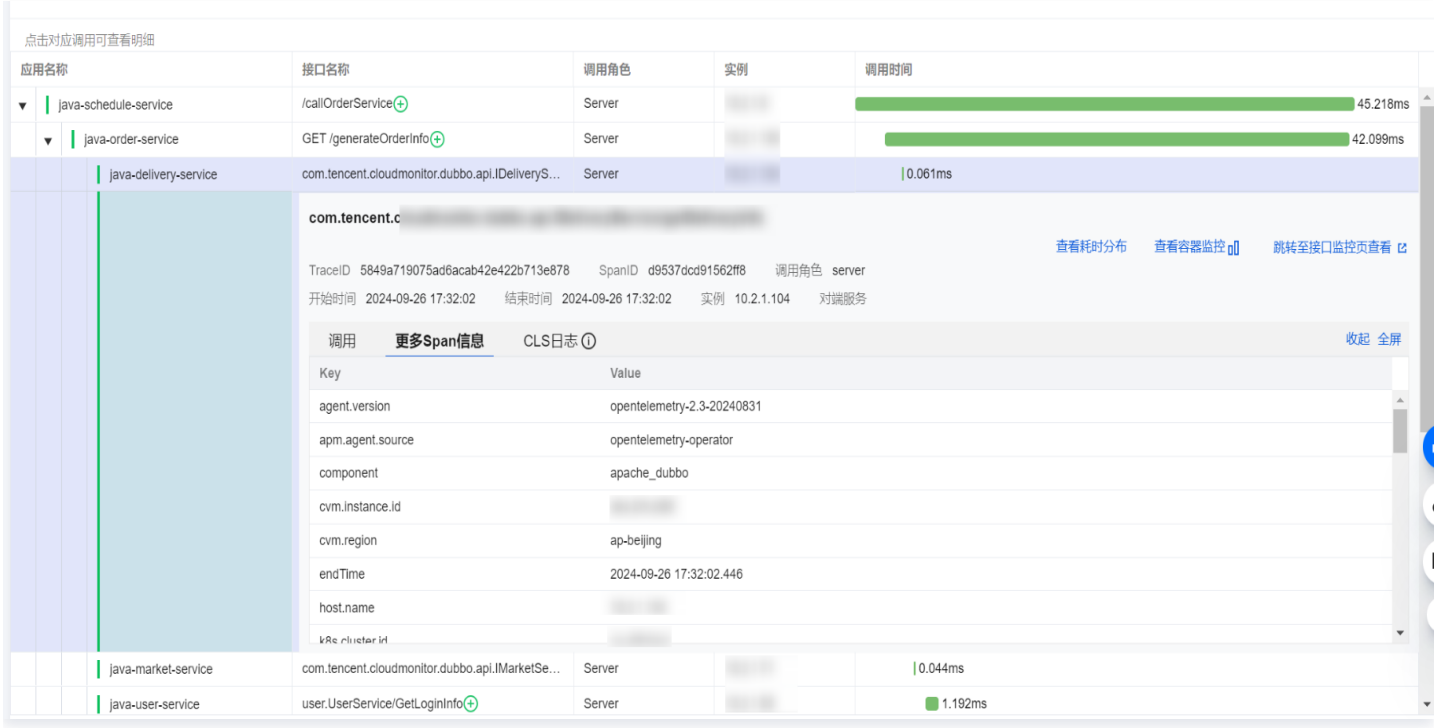


Span 详情

在瀑布图中单击任何一行，将展开 Span 详情视图，Span 中携带的所有信息都将在 Span 详情视图中体现。在标签页上方将展示 Span 名称、调用角色、开始时间、结束时间、实例名称、对端服务等最重要的 Span 信息。根据 Span 的不同特点，Span 详情视图中还将包含如下标签页：

- **错误**：如果该 Span 的状态为错误，此标签页将展示具体的错误信息。
- **调用**：如果该 Span 代表一次 RPC 或 HTTP 调用，此标签页将展示具体的调用信息，这些信息通过前缀为 `http` 和 `rpc` 的 Span 属性来承载。
- **SQL**：如果该 Span 代表一次 SQL 调用，此标签页将展示具体的调用信息，这些信息通过前缀为 `db` 的 Span 属性来承载。
- **事件**：如果该 Span 中包含事件，此标签页将展示具体的事件信息。
- **更多 Span 信息**：此标签页将补充展示所有 Span 信息，包括在 Span 中上报的所有自定义字段。
- **CLS 日志**：如果该应用关联了 CLS 日志服务，此标签页将根据 Trace ID 和 Span ID 查询并展示和该调用相关的日志，请确保关联的 CLS 主题开启了全文索引。

如果该 Span 包含了方法栈快照，在 Span 详情中单击**方法栈快照**，将展示本地方法栈中每个方法的执行耗时，详情请参见 方法栈快照 。



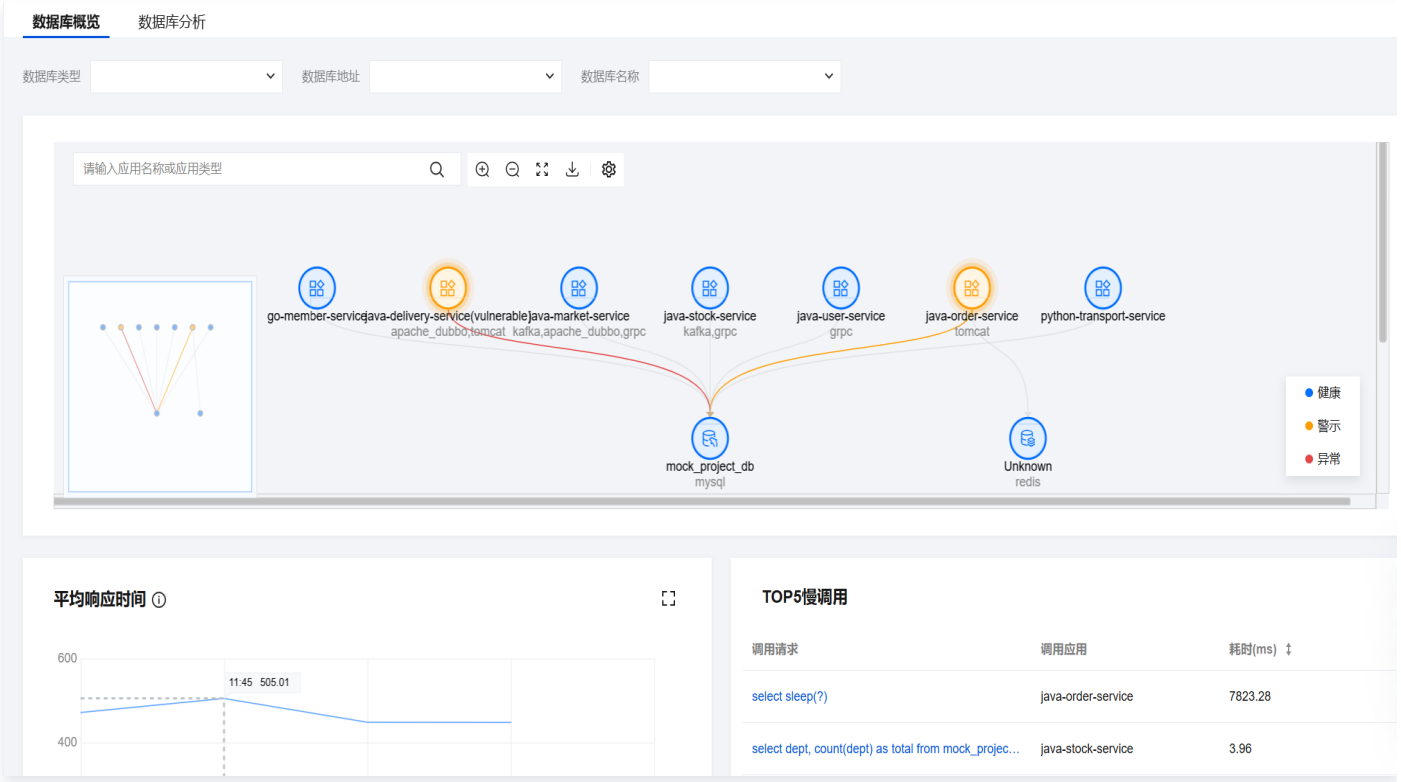
数据库调用
数据库概览

最近更新时间：2024-11-27 16:29:32

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. 登录 腾讯云可观测平台 。
2. 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**。
3. 进入数据库概览页面，即可查看相关模块信息。



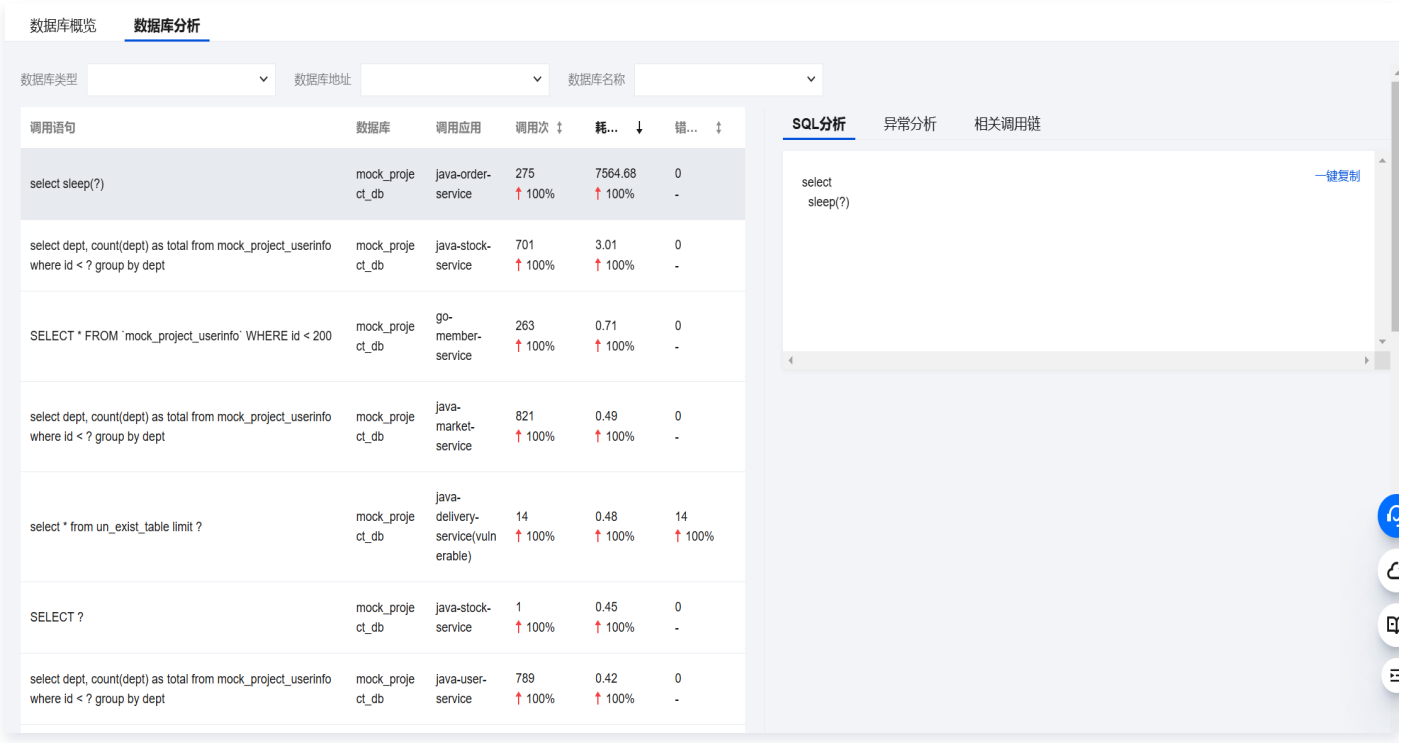
数据库分析

最近更新时间：2024-11-27 16:29:32

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. 登录 腾讯云可观测平台 。
2. 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**。
3. 进入数据库分析页面，即可查看相关模块信息。



访问管理

概述

最近更新时间：2025-05-08 14:45:22

如果多个用户使用同一个云账号使用应用性能监控 APM，将存在以下问题：

- 账号的密钥由多人共享，泄密风险高。
- 无法进行独立的权限管理，易产生误操作，导致安全风险。

此时，您可以为每个用户分配一个腾讯云子用户来解决以上问题。默认情况下，子用户没有应用性能监控 APM 的访问权限，因此，我们需要通过访问管理（CAM）来赋予子用户访问 APM 的权限。

简介

访问管理（Cloud Access Management，CAM）是腾讯云提供的一套 Web 服务，它主要用于帮助客户安全管理腾讯云账户下的资源的访问权限。通过 CAM，您可以创建、管理和销毁用户（组），并通过身份管理和策略管理控制哪些人可以使用哪些腾讯云资源。
当您使用 CAM 时，可以将策略与一个用户或一组用户关联起来，策略能够授权或者拒绝用户使用指定资源完成指定任务。有关 CAM 策略的更多相关使用信息，请参见 策略 。

> **说明：**
> 若您无需通过子用户访问应用性能监控 APM，可以跳过此章节。

授权方式

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 创建新的策略，并参考 策略授予 对子用户授予自定义策略。

自定义策略

最近更新时间：2025-05-08 14:45:22

概述

基于访问管理（CAM）提供的自定义策略，可以针对具体的资源和操作进行精细化权限管理，关于自定义策略的更多详情，可参考 CAM 策略管理 。

创建自定义策略

CAM 支持如下三种方式创建自定义策略，请参考对应的帮助文档完成自定义策略的创建。

- 通过策略生成器创建自定义策略
- 通过标签授权创建自定义策略
- 通过策略语法创建自定义策略

推荐使用 策略生成器创建自定义策略 。

APM 资源

资源（resource）元素描述一个或多个操作对象，可采用下述的六段式描述方式，更多详情请参考 资源描述方式 。

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

service_type 参数

在 APM 场景中，service_type 参数固定填写为 `apm` 。

resource 参数

APM 支持**业务系统**和**应用**两种资源，resource 参数分别使用 `apm-instance/` 和 `apm-service/` 前缀，并附加上对应的业务系统 ID 和应用 ID。例如，针对 ID 为 `apm-btzsrI123` 的业务系统进行精细化授权，resource 参数需要定义为 `apm-instance/apm-btzsrI123` ，此时完整的六段式为如下所示：

```
qcs::apm:ap-guangzhou:uin/1250000000:apm-instance/apm-btzsrI123
```

> **说明：**
> 六段式描述方式仅适用于授权粒度为 `资源级` 的接口，对于授权粒度为 `接口级` 的接口，资源（resource）需填 `*` 。

基于业务系统管理 APM 访问权限

所有授权粒度为 `资源级` 的 APM 接口，都支持基于业务系统管理访问权限。在大多数场景下，基于业务系统管理 APM 访问权限，就可以实现精细化权限管理，详情的操作步骤，请参考 快速授予业务系统级别的访问权限 。

基于应用管理 APM 访问权限

在 APM 的实体模型中，单个业务系统中包含多个应用，因此应用是从属于业务系统的二维资源。在授予一个子用户业务系统级别的访问权限后，如果要限制其只能访问特定的应用，请确保以下接口的可操作资源范围在应用级别。

- **DescribeApmService**（获取apm应用信息）

- **DescribeApmServiceBrief**（获取应用名列表）
- **DescribeApmServiceMetric**（获取 APM 应用指标）
- **ModifyApmApplicationConfig**（修改应用配置接口）

> **说明：**
> 基于应用管理 APM 访问权限，需要熟练使用 CAM 策略语法，且配置复杂度非常高，请谨慎使用这种方式。在大多数场景下，基于业务系统管理 APM 访问权限，已经可以满足精细化权限管理的需求。

策略授予

最近更新时间：2025-05-08 14:45:22

默认情况下，子用户没有应用性能监控 APM 的访问权限，需要授予子用户相关 CAM 策略，子用户才能正常访问应用性能监控 APM。

前提条件

使用主账号或拥有 QcloudCamFullAccess 权限的子用户登录腾讯云控制台，并参见 新建子用户 操作步骤创建子用户。
可以被授予的策略包括 **APM 预设策略**和**自定义策略**两种：

- **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**（应用性监控（APM）只读访问权限）。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。
  此外，APM 还提供了预设策略 **QcloudAPMOperationalPrecondition**（操作级接口前置条件），用于实现业务系统级别的访问权限，详情请参考 快速授予业务系统级别的访问权限 。
- **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 创建新的 CAM 策略。

操作步骤

1. 使用主账号或拥有 QcloudCamFullAccess 权限的子用户进入 访问管理 > 策略 。
2. 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。
3. 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**。

4. 在弹框中勾选对应的用户，单击**确定**即可。

资源标签

最近更新时间：2025-05-12 10:14:42

应用性能监控（APM）结合腾讯云资源标签，提供按标签授权和按标签分账能力，关于腾讯云标签的更多详情，请参考 腾讯云标签 。资源标签是腾讯云提供的管理资源工具。资源标签分为标签键和标签值，一个标签键可对应多个标签值，您可以参见以下步骤进行按标签授权和账单分账。

使用场景

某公司有多个业务系统接入了应用性能监控，这些系统分别由 A、B 两个部门独立研发、运营。现需要对 A、B 部门创建标签、绑定业务系统并授予权限：

- 创建 A 标签：绑定 A 部门所有业务系统。
- 创建 B 标签：绑定 B 部门所有业务系统。

按标签授权

用户 A 为 A 部门开发人员，负责 A 部门所有业务系统开发。需要授予该开发人员 A 标签权限。

按标签分账

用户 B 为公司的财务人员，负责对 A、B 部门财务支出进行独立核算。需要授予该财务人员 A、B 标签权限，并按标签进行分账核算。

准备工作

步骤1：创建标签

参见下列步骤，分别创建 A、B 标签。

1. 前往 标签控制台 > 标签列表 。
2. 单击**新建标签**，进入添加标签页面，填写标签键和对应的标签值。
3. 单击**确定**，完成标签创建。

步骤2：为业务系统分配标签

参见下列步骤，为 A 标签绑定 A 部门下的所有业务系统，为 B 标签绑定 B 部门下的所有业务系统。

1. 前往 应用性能监控控制台 > 资源管理 > 业务系统管理 。
2. 单击**新建**，在弹框中填写业务系统信息，并添加标签。也可以在下拉列表中找到已创建的业务系统，在**基本信息**栏目下单击**编辑**，并添加标签。

按标签授权

按标签授权的策略，根据下列步骤给用户 A 授予 A 标签权限，用户 B 授予 A、B 标签权限。

1. 前往 访问管理控制台 > 策略 。
2. 单击左上角的**新建自定义策略**。
3. 在弹出的选择创建方式窗口中，单击**按标签授权**，进入按标签授权页面。
4. 根据页面指标，完成自定义策略的创建。
5. 参考 策略授予 ，完成自定义策略与用户/用户组/角色的关联。

按标签分账

您可以指定特定的标签键为分账标签，在账单出具时，可以根据业务系统对应的标签实现费用统计。

步骤1：设置分账标签

1. 前往 费用中心控制台 > 分账管理 > 分账标签 。
2. 在此页面您可看到已创建的标签键列表，选择需要展示的标签键，单击**设置为分账标签**，即可将该标签键设置为账单中的分账标签。

步骤2：按标签展示账单

1. 前往 费用中心控制台 > 费用账单 > 账单查看 > 多维度汇总账单 。
2. 选择**按标签**，并指定需要分账的**标签键**，便可查看根据该标签键汇总的云产品账单图表。

告警服务
新建告警

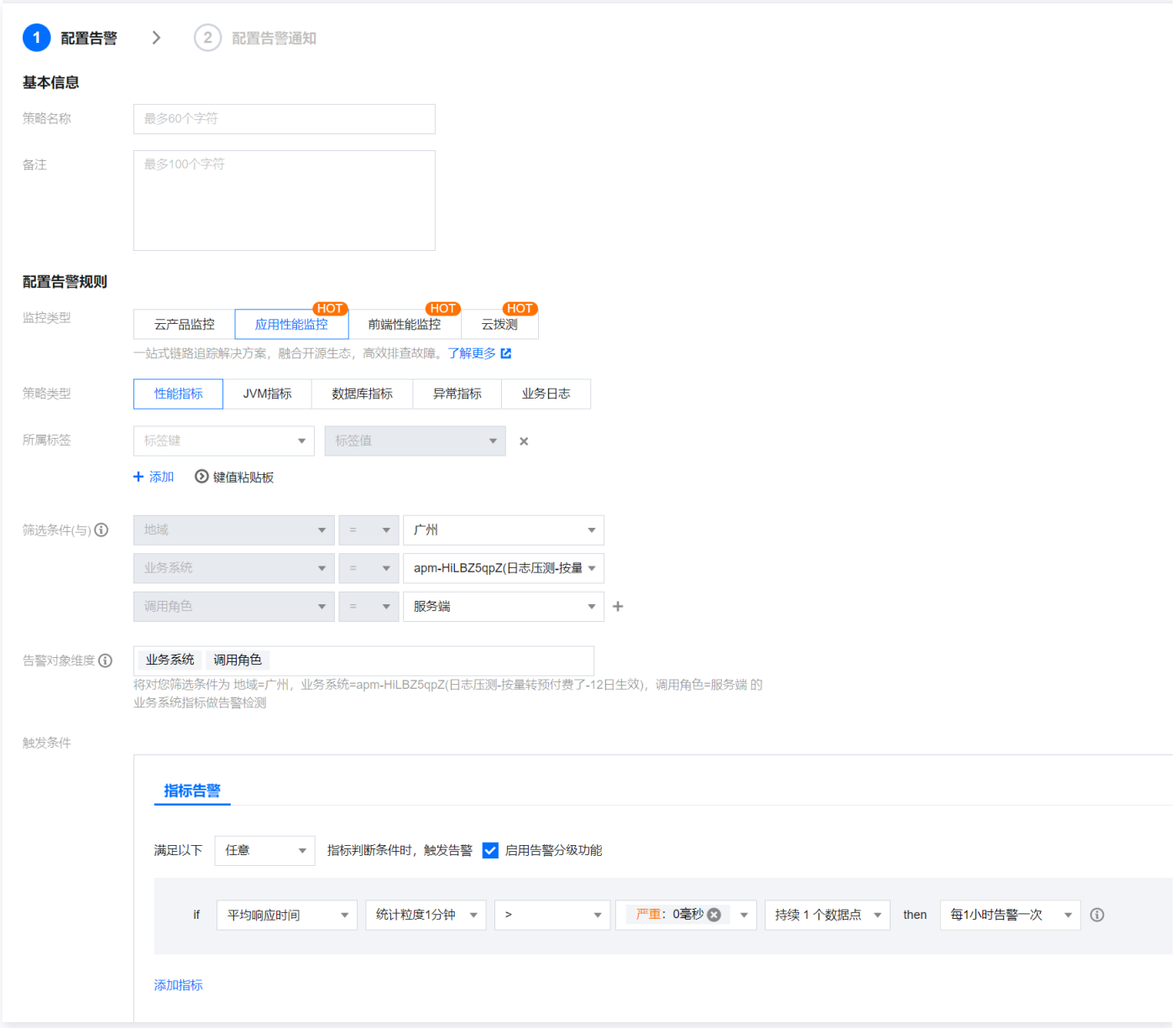
最近更新时间：2024-07-11 10:20:41

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 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**告警管理 > 告警配置 > 告警策略**。
3. 单击**新建策略**，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。
	备注	自定义策略备注。
配置告警规则	监控类型	选择**应用性能监控**类型。
	策略类型	默认选择性能指标。
	筛选条件（与）	筛选出符合条件的对象进行告警检测，各筛选条件之间为 AND 关系。筛选条件仅展示有上报数据的对象。 您可以根据需求选择需要检测的告警对象。
	告警对象维度	支持自定义告警通知内容中的告警对象，假设您选择了业务系统、应用和调用角色，则告警对象显示 业务系统、应用、调用角色。
	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。
配置告警通知	通知模板	系统为您默认配置通知模板，如需创建通知模板请参见 新建通知模板 。
高级配置	弹性伸缩	启用并配置成功后，达到告警条件可触发弹性伸缩策略并进行缩容或扩容。



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。

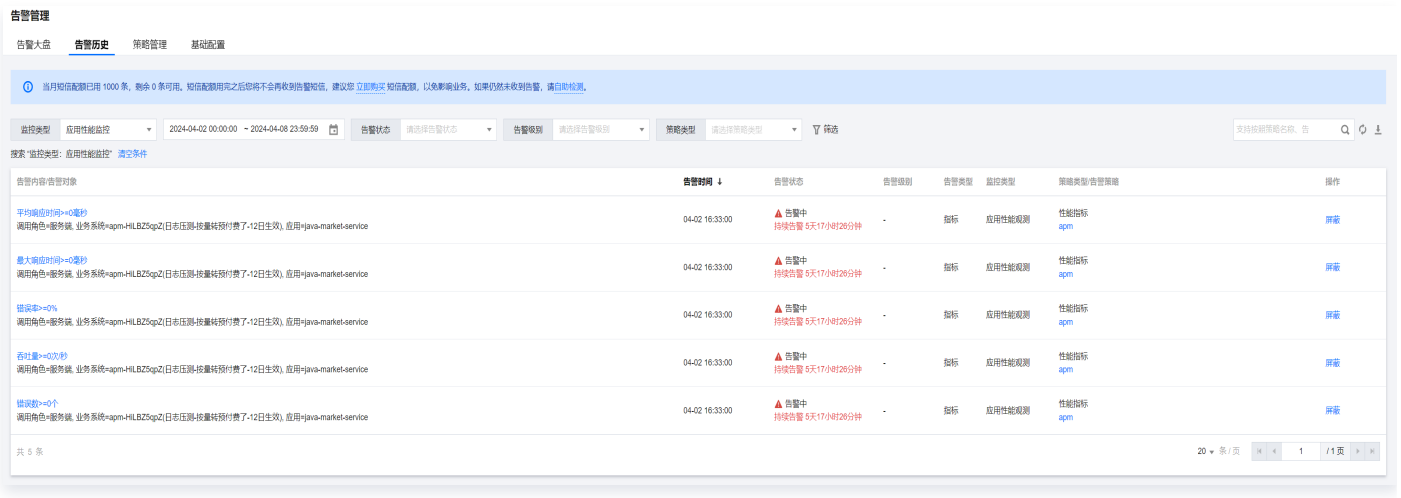
查看告警

最近更新时间：2024-07-09 10:29:11

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 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**告警管理 > 告警治理> 告警历史**。
3. 通过上方进行条件筛选，监控类型选择“应用性能监控”。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

最近更新时间：2025-02-14 15:32:42

本文将为您介绍业务系统的相关配置信息。

操作步骤

1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择选择**应用性能监控** > **系统配置**，进入**业务系统配置**页面。
3. 在页面顶部选择地域和目标业务系统。
4. 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。
开启免费模式	开启免费模式后，该业务系统将永久免费。关于免费模式的限制，请参见 关于免费模式的使用限制 。
计费模式	支持**按量付费**和**预付费**。
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。
添加标签	应用性能监控结合腾讯云资源标签功能，为您提供按标签授予子账号权限和按标签分账功能。请参见 访问管理 设置标签。

   - **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。
   - **应用健康阈值**：

     支持**响应时间警示线**和**错误率警示线**配置，在应用列表等视图中，APM 将基于应用健康阈值来判断应用健康状态。
   - **应用安全**：

     支持**组件漏洞扫描**和 **SQL 注入分析**应用安全功能项开关。在业务系统中打开或关闭应用安全功能项，将对该业务系统的所有应用生效。此外，APM 也提供了应用级别的应用安全功能项开关，用于覆盖业务系统中的配置。
   - **日志关联**：

     支持**日志地域**、**日志集**、**日志主题**配置项，用于实现链路与日志服务 CLS 的关联。此外，APM 也提供了应用级别的日志关联配置项，用于覆盖业务系统中的配置。关于链路日志关联分析的更多介绍，请参见 在链路详情中查询关联日志 。
   - **Dashboard 关联**：

     用于在应用详情中关联展示云监控 Dashboard。此外，APM 也提供了应用级别的 Dashboard 关联配置项，用于覆盖业务系统中的配置。关于 Dashboard 关联的更多介绍，请参见 Prometheus 数据集成以及 Dashboard 关联展示 。

应用配置

最近更新时间：2025-02-21 09:14:33

操作步骤

1. 登录 腾讯云可观测平台。
2. 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。
3. 在下拉框选择目标业务系统和目标应用。
4. 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：operationName(.*?),operation(*)。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 /service/demo/ 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，收敛规则正则为/service/demo/(.*?)。
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 service/demo/example 不需要进行 URL 收敛，则表达式为/service/demo/example/(.*?)。

   - **日志关联**：支持**日志地域**、**日志集**、**日志主题**配置项，用于实现链路与日志服务 CLS 的关联，可以覆盖业务系统中的配置。关于链路日志关联分析的更多介绍，请参见 在链路详情中查询关联日志 。
   - **应用安全**：支持**组件漏洞扫描**和 **SQL 注入分析**应用安全功能项开关，可以覆盖业务系统中的配置。
   - **Dashboard 关联**：用于在应用详情中关联展示云监控 Dashboard，可以覆盖业务系统中的配置。关于 Dashboard 关联的更多介绍，请参见 Prometheus 数据集成以及 Dashboard 关联展示 。
   - **探针配置**：仅适用于腾讯云增加版 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
探针总开关	可以通过探针总开关临时关闭数据上报，关闭后该应用不再产生 APM 费用，应用无需重启。如果需要永久关闭数据上报，请卸载探针，以免不必要的性能开销。 **说明：**

	该功能仅支持1.16-20241031及以上版本探针。
方法栈快照	如果某个接口调用耗时频繁超过**慢调用监听阈值**，该接口将有机会自动生成方法栈执行快照。您可以在 链路追踪 页面过滤出包含方法栈快照的调用。在**链路详情**页面，如果一个包含了方法栈快照，会展示**方法栈快照**按钮，单击后可以查看本地方法栈的执行详情。 **说明：** 该功能仅支持1.16-20240831及以上版本探针。
接口过滤	对于被匹配到的接口，探针不会上报相关链路数据。请按照正则表达式输入需要过滤的接口，多个接口之间请使用英文逗号分隔。 **说明：** 该功能仅支持1.16-20240328及以上版本探针。

采样配置

最近更新时间：2025-04-02 18:53:22

应用性能监控提供链路采样能力，通过尾部采样（Tail Based Sampling）技术实现。合理配置采样，能够降低链路数据存储量。关于采样配置的详细信息，请参考 通过采样策略降低 APM 使用成本 。

操作步骤

> **说明：**
> 目前采样策略需要通过开白名单的形式提供，请您通过 提交工单 进行申请。

1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择**应用性能监控** > **系统配置**，进入**采样配置**页面。

修改全局采样配置

1. 在**全局采样配置**模块，单击**编辑**。
2. 修改**采样率**，有效的采样百分比在10%到100%之间。
3. 修改**保存错误链路**开关，以及**慢调用保存阈值**。

配置全采样接口

当全局采样率小于100%的时候，可以自定义需要全采样的接口，如果链路经过了需要全采样的接口，整条链路都会被完整保存。

1. 单击**新增全采样接口**。
2. 在对话框中输入策略名，并指定需要进行全采样的接口。

如果在策略中指定了具体的应用，可以不填写接口匹配规则，此时代表经过此应用的所有链路都会被完整保存。您可以基于精确匹配、前缀匹配以及后缀匹配来指定接口。在实际使用场景中，可以将关注度比较高的重要接口定义为全采样接口，确保不错过任何一条针对重要接口的调用。

采样策略配置原则

- 修改采样策略相关配置，可以即时生效，无需重启应用。
- 请根据实际业务场景修改全局采样率。当全局采样率低于100%的时候，APM 不会实时保存没有被命中的链路。
- 推荐开启**保存错误链路**开关，并将**慢调用保存阈值**设置在500ms到2000ms之间。
- 对于没有被命中的链路，APM 会基于**保存错误链路**开关、**慢调用保存阈值**，以及**全采样接口**等配置项，综合判断每条链路是否需要持久化保存。错慢链路以及经过了全采样接口的链路，在经过10分钟左右的延迟后，最终将被完整保存。

Prometheus 集成

最近更新时间：2024-11-22 16:53:01

对于通过 OpenTelemetry 方案接入 APM 的应用，可以使用 OpenTelemetry API 上报自定义指标，APM 服务端可以将指标同步到腾讯云 Prometheus 监控服务。该页面用于配置 APM 业务系统和 Prometheus 实例之间的关联，关于腾讯云 Prometheus 监控服务，请参见 Prometheus 监控概述，关于 APM-Prometheus 数据集成，请参见 Prometheus 数据集成以及 Dashboard 关联展示。

操作步骤

1. 登录 APM 控制台，前往**系统配置** > **Prometheus 集成**。
2. 在**关联配置**中，开启 Prometheus 关联，并选择当前地域的任何一个 Prometheus 实例。每个 APM 业务系统只能关联同地域的最多一个 Prometheus 实例。

   在当前账号第一次做关联操作的时候，需要授予 APM 访问 Prometheus 资源的权限，根据控制台的提示完成服务授权即可，系统会在访问管理（CAM）自动创建名为 APM_QCSLinkedRoleInPromInstance 的角色，单击**同意授权**即可。

3. 单击**新增指标同步规则**，指定需要同步到 Prometheus 实例的指标。

   对于每一条同步规则，您可以基于**精确匹配**、**前缀匹配**以及**后缀匹配**这三种匹配方式来匹配指标名，也可以指定该规则的生效范围（可以是该业务系统的全部应用，或某一个具体的应用）。当 APM 服务端收到应用上报的自定义指标后，满足同步规则的指标将被写入到被关联的 Prometheus 实例中，不满足同步规则的指标将被丢弃。

关联日志
在链路详情中查询关联日志

最近更新时间：2024-11-19 17:44:42

如果您使用腾讯云日志服务（CLS）保存日志，可以实现链路数据与日志数据的关联，当应用出现性能问题的时候，可以在应用性能监控 APM 控制台快速查询一条链路所对应的业务日志，及时定位分析并解决问题。

前提条件

已了解并使用腾讯云日志服务（CLS），详情可参见 日志服务相关指引 。

操作步骤

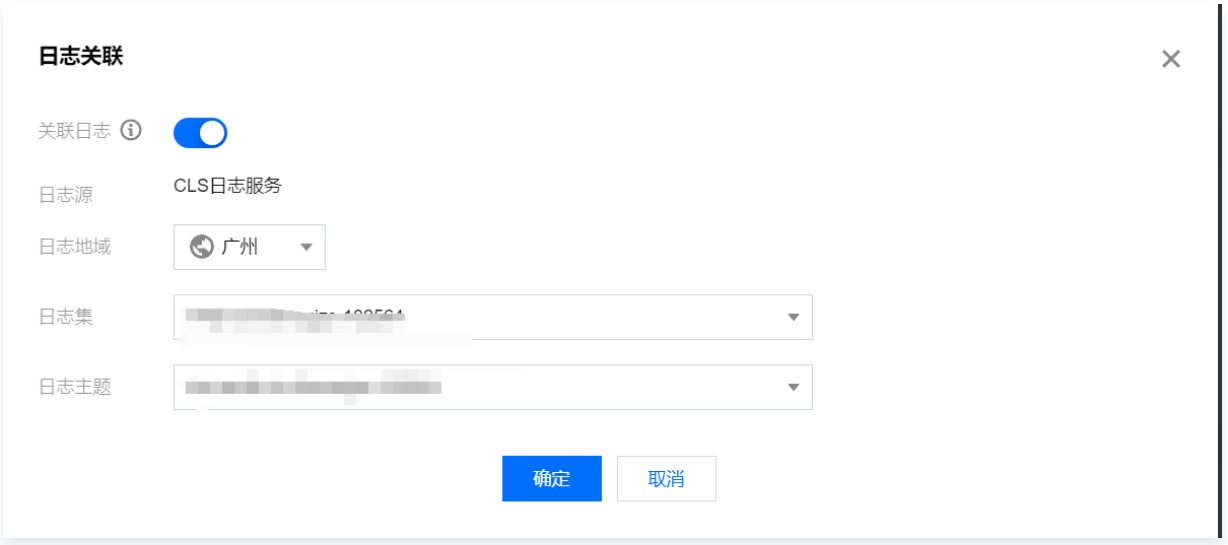
步骤1：在日志中注入 TraceID

不同语言不同协议注入 TraceID 方式不同，详情请查看以下文档指引，在日志中注入 TraceID（**其它语言和协议将会陆续支持**）。

- 通过 skywalking 协议（Java）上报
- 通过 OpenTelementry 增强版 Java 探针上报

步骤2：在系统配置页面关联日志

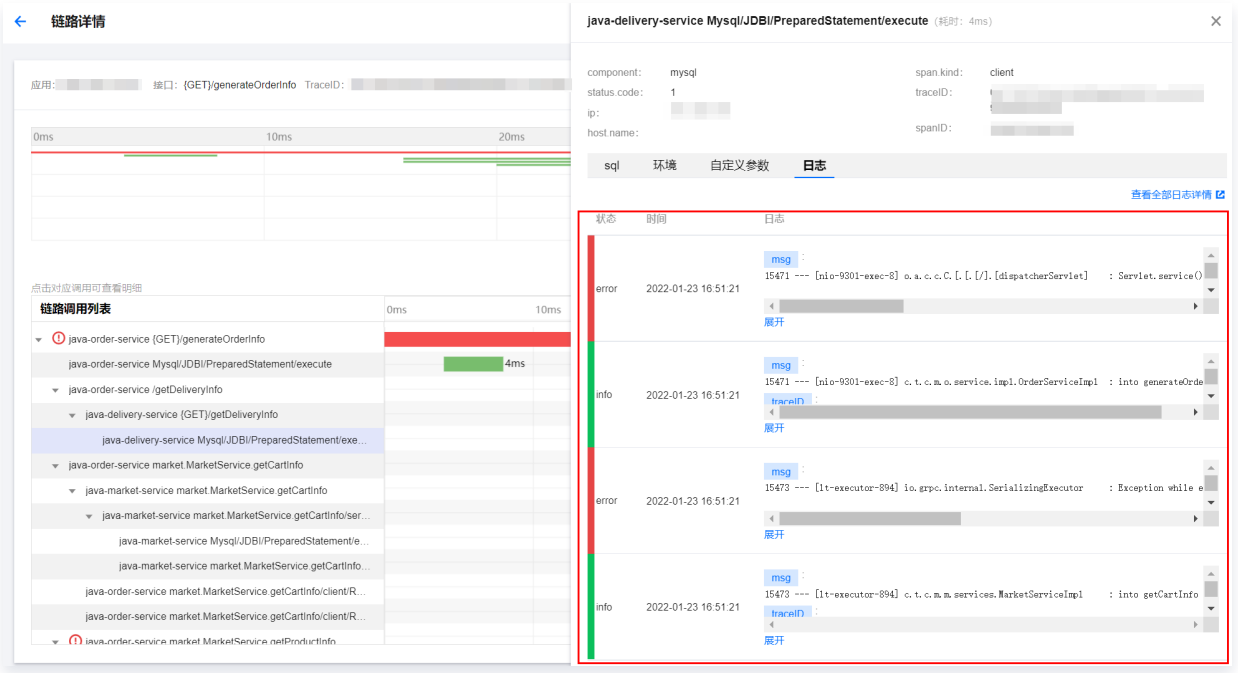
1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择单击**应用性能监控 > 系统配置 > 业务系统配置**页面。
3. 选择对应的业务系统，在日志关联页面单击**编辑**。
4. 配置日志关联信息。
   - 开启关联日志。
   - 选择对应的日志地域、日志集和日志主题。
5. 配置完后单击**确定**即可。



除了可以在业务系统级别配置日志关联以外，也可以在应用级别配置日志关联，请到 应用性能监控 > 系统配置 > 应用配置 中进行操作，具体的配置方式与业务系统级别相同。

步骤3：在链路查询页面查看日志信息

1. 登录 腾讯云可观测平台。
2. 在左侧菜单栏选择单击**应用性能监控 > 链路追踪**，进入链路追踪页面。
3. 选择对应的业务系统，找到对应的 TraceID，单击 TraceID 进入链路详情。
4. 在右侧窗口切换日志菜单，即可查看日志相关信息，排查故障。



在日志中注入 TraceID
OpenTelementry 增强版 Java 探针

最近更新时间：2024-05-17 10:47:11

通过腾讯云 OpenTelementry 增强版 Java 探针接入 APM 后，可在应用日志中轻松输出上下文 TraceID 和 SpanID，只需要修改日志配置文件中的 pattern，即可实现 TraceID 和 SpanID 的注入。OpenTelementry 增强版 Java 探针的日志注入功能支持如下日志框架：

- Log4j 1.2+
- Log4j2 2.7+
- Logback 1.0+

在日志配置文件中，TraceID 和 SpanID 分别使用 `%X{trace_id}` 和 `%X{span_id}` 字段注入，如果上下文没有TraceID 或 SpanID，不会影响整行日志的输出。

Log4j

log4j.properties 文件修改示例：

```
log4j.appender.warn.layout=org.apache.log4j.PatternLayout
log4j.appender.warn.layout.ConversionPattern=traceId: %X{trace_id} spanId: %X{span_id}
- %m%n
```

Log4j2

Log4j2支持多种格式配置文件，包括 XML、Json、YAML、Properties 等。以 XML 格式为例，Pattern 有多种写法，请根据已有的写法参考其中一种：

方式一

```
<PatternLayout pattern="traceId: %X{trace_id} spanId: %X{span_id} - %m%n"/>
```

方式二

```
<PatternLayout>
    <Pattern>traceId: %X{trace_id} spanId: %X{span_id} - %m%n</Pattern>
</PatternLayout>
```

方式三

```
<Layout type="PatternLayout" pattern="traceId: %X{trace_id} spanId: %X{span_id} -
%m%n"/>
```

Logback

logback.xml 文件修改示例：

```
<encoder>
    <pattern>traceId: %X{trace_id} spanId: %X{span_id} %msg%n</pattern>
</encoder>
```

输出效果

当上下文存在 TraceID 和 SpanID 时：

```
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: 50784806406e989
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: 50784806406e9896
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: 50784806406e9896
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: d250c1d1319a3af6
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: d250c1d1319a3af6
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: d250c1d1319a3af6
traceId: db9d6ee1073fae12d6bd07fae2fb39df spanId: d250c1d1319a3af6
```

当上下文不存在 TraceID 和 SpanID 时：

```
traceId:  spanId:
traceId:  spanId:
traceId:  spanId:
traceId:  spanId:
```

Skywalking 协议（Java）

最近更新时间：2024-07-19 14:57:51

本文将介绍 Java skywalking 上报方式如何关联 TraceID 。

操作步骤

配置 Logback 输出 TraceID

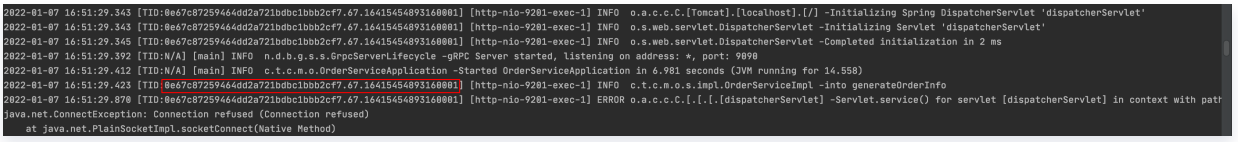
1. 引入依赖。

```
<dependency>
   <groupId>org.apache.skywalking</groupId>
   <artifactId>apm-toolkit-logback-1.x</artifactId>
   <version>8.3.0</version>
</dependency>
```

2. 修改 logback-spring.xml 中的 Appender 的 Pattern 格式。

```
<encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
 <layout
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackLayou
t">
     <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%tid] [%thread] %-5level %logger{36} -
%msg%n</Pattern>
 </layout>
</encoder>
```

3. 启动项目，打印结果如下：
   %tid 会打印 TraceID，默认 TID：N/A，当有请求调用时，会显示 TraceID。



配置 log4j-1x 输出 TraceID

1. 通过 maven 或 gradle 引入 toolkit 依赖。

```
<dependency>
<groupId>org.apache.skywalking</groupId>
<artifactId>apm-toolkit-log4j-1.x</artifactId>
<version>{project.release.version}</version>
</dependency>
```

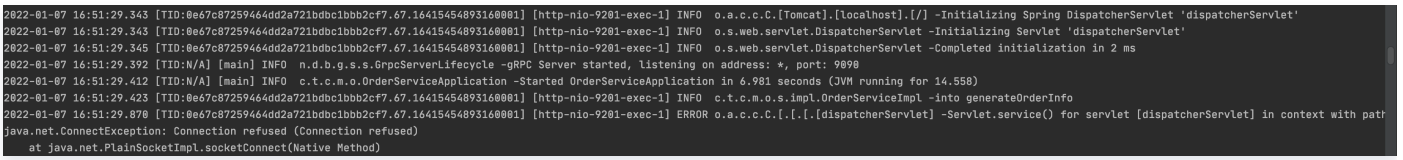
2. 修改 log4j1.properties 配置 layout。

```
log4j.appender.CONSOLE.layout=org.apache.skywalking.apm.toolkit.log.log4j.v1.x.Trace
IdPatternLayout
```

在 `layout.ConversionPattern` 中设置 `%T` （在2.0-2016中，您应该使用%x，为什么做了修改？）。

```
log4j.appender.CONSOLE.layout.ConversionPattern=%d [%T] %-5p %c{1}:%L - %m%n
```

3. 启动项目，打印结果如下：
   假设 TraceID 存在，当您使用 `-javaagent` 激活 skywalking tracer 后，log4j 将会输出 **TraceID**。如果 tracer 未激活，输出将是 `TID: N/A` 。



配置 log4j-2x 输出 TraceID

1. 使用 maven 或 gradle 引入 toolkit 依赖。

```
<dependency>
<groupId>org.apache.skywalking</groupId>
<artifactId>apm-toolkit-log4j-2.x</artifactId>
<version>{project.release.version}</version>
</dependency>
```

2. 在 log4j2.xml 的 pattern 中配置 `[%traceId]` 。

- 支持在 log4j2.xml 的 pattern 中配置 [%traceId]。

```
<Appenders>
   <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d [%traceId] %-5p %c{1}:%L - %m%n"/>
   </Console>
</Appenders>
```

- 支持 log4j2 AsyncRoot，无需其他配置。请参阅下文的 log4j2.xml 演示。有关详细信息：Log4j2异步记录器 。

```
 <Configuration>
     <Appenders>
         <Console name="Console" target="SYSTEM_OUT">
             <PatternLayout pattern="%d [%traceId] %-5p %c{1}:%L - %m%n"/>
         </Console>
     </Appenders>
     <Loggers>
         <AsyncRoot level="INFO">
             <AppenderRef ref="Console"/>
         </AsyncRoot>
```

```
    </Loggers>
  </Configuration>
```

- 支持 log4j2 AsyncAppender，不需要其他配置。请参阅下文的 log4j2.xml 演示。
  有关详细信息：All Loggers Async

> **说明**
> Log4j-2.9 和更高版本要求在类路径上使用 disruptor-3.3.4.jar 或更高版本。在 Log4j-2.9 之前，需要使用 interrupter-3.0.0.jar 或更高版本。这是最简单的配置，并提供最佳性能。要使所有记录器异步，请将 disruptor jar 添加到类路径中并且 设置系统属性 `log4j2.contextSelector` 为 `org.apache.logging.log4j.core.async.AsyncLoggerContextSelector` 。

```
<Configuration status="WARN">
  <Appenders>
    <!-- Async Loggers will auto-flush in batches, so switch off immediateFlush. -->
    <RandomAccessFile name="RandomAccessFile" fileName="async.log"
immediateFlush="false" append="false">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] [%traceId] %m %ex%n</Pattern>
      </PatternLayout>
    </RandomAccessFile>
  </Appenders>
  <Loggers>
    <Root level="info" includeLocation="false">
      <AppenderRef ref="RandomAccessFile"/>
    </Root>
  </Loggers>
</Configuration>
```

详细可参见： [Mixed Sync & Async] #MixedSync-Async

> **说明**
> Log4j-2.9 及更高版本需要类路径上使用 disruptor-3.3.4.jar 或更高版本。在 Log4j-2.9 之前，需要 disruptor-3.0.0.jar 或更高版本。不需要将系统属性 “Log4jContextSelector” 设置为任何值。

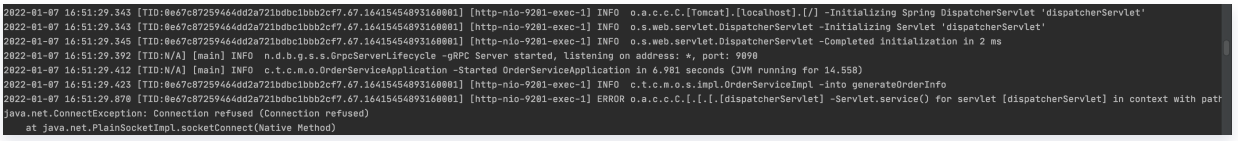
```
<Configuration status="WARN">
  <Appenders>
    <!-- Async Loggers will auto-flush in batches, so switch off immediateFlush. -->
    <RandomAccessFile name="RandomAccessFile" fileName="asyncWithLocation.log"
              immediateFlush="false" append="false">
      <PatternLayout>
        <Pattern>%d %p %class{1.} [%t] [%traceId] %location %m %ex%n</Pattern>
      </PatternLayout>
    </RandomAccessFile>
  </Appenders>
  <Loggers>
```

```
    <!-- pattern layout actually uses location, so we need to include it -->
    <AsyncLogger name="com.foo.Bar" level="trace" includeLocation="true">
      <AppenderRef ref="RandomAccessFile"/>
    </AsyncLogger>
    <Root level="info" includeLocation="true">
      <AppenderRef ref="RandomAccessFile"/>
    </Root>
  </Loggers>
</Configuration>
```

- 支持 log4j2 AsyncAppender，详细信息请参见：Log4j2 AsyncAppender

```
<Configuration>
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d [%traceId] %-5p %c{1}:%L - %m%n"/>
        </Console>
        <Async name="Async">
            <AppenderRef ref="Console"/>
        </Async>
    </Appenders>
    <Loggers>
        <Root level="INFO">
            <AppenderRef ref="Async"/>
        </Root>
    </Loggers>
</Configuration>
```

3. 启动项目，打印结果如下
   假设 TraceID 存在，当您使用 `-javaagent` 激活 skywalking tracer 后，log4j 将会输出 **TraceID** 。如果 tracer 未激活，输出将是 `TID: N/A` 。



演示 Demo

最近更新时间：2025-02-14 09:02:33

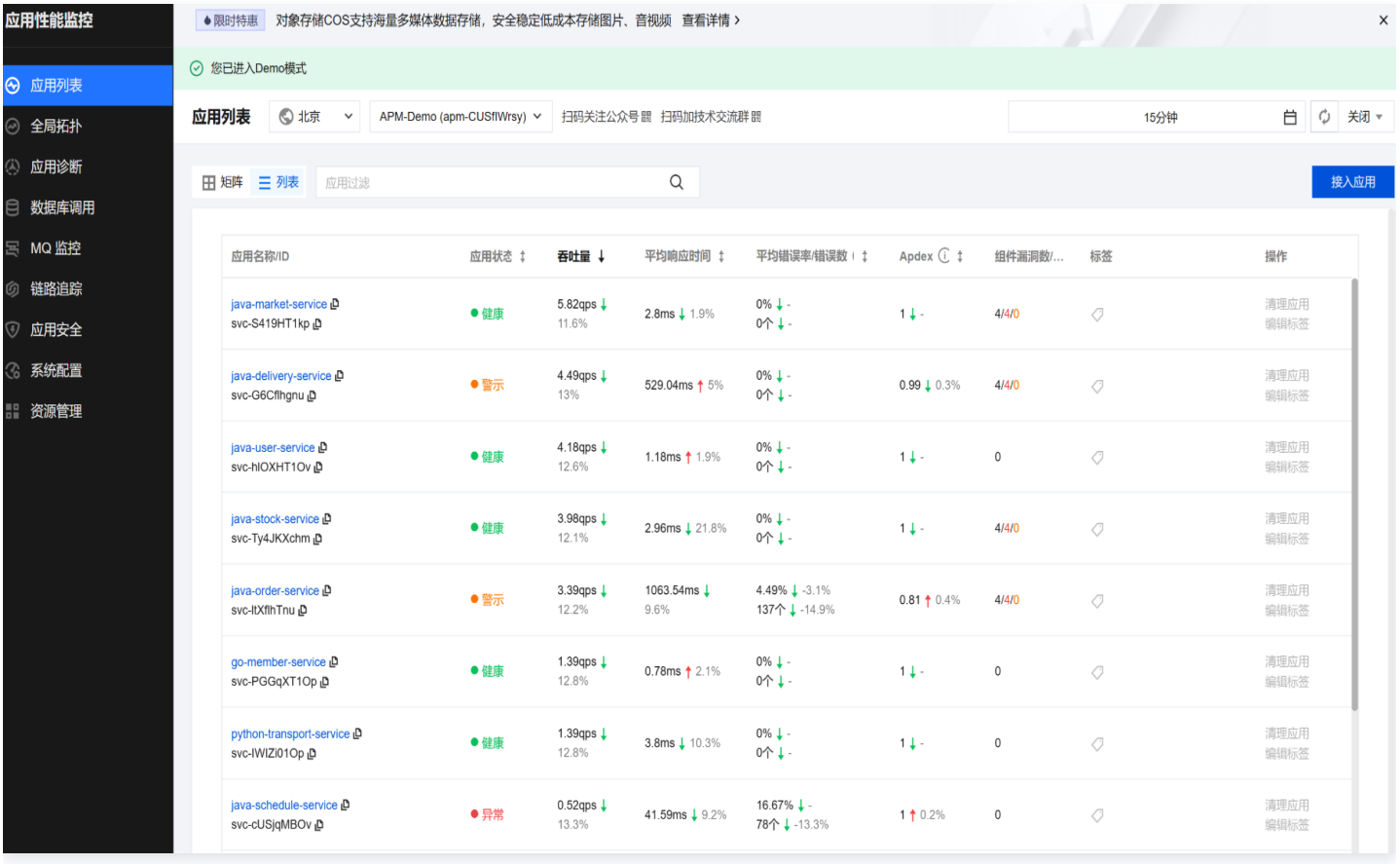
应用性能监控 APM 提供了演示 Demo，方便用户直观的感受产品功能，快速开启高效的应用可观测旅程 。

使用限制

- 业务系统被固定为北京地域的 APM-Demo，不能接入新的应用到此业务系统。
- 涉及写操作的功能被限制。
- 演示 Demo 通过腾讯云备用域名 cloud.tencent.com.cn 进行展示，因此，登录 腾讯云控制台 请不要使用此备用域名，否则会造成会话冲突。如果误用了备用域名登录腾讯云控制台，可通过 cloud.tencent.com （国内站）或 tencentcloud.com （国际站）主域名重新登录，再次进入演示 Demo 即可。

Demo 进入方式

演示 Demo 支持免登录使用，您可以点击 演示 Demo 进入体验。同时，我们在 应用性能监控 APM 的控制台 提供了进入演示 Demo 的入口，您可自行进入体验。在演示 Demo 中，相关应用已经提前完成接入并上报可观测数据，用户可以直接体验。



应用安全

最近更新时间：2025-03-13 21:57:22

应用性能监控 APM 能够实时监测应用程序运行时的行为，提供强大的威胁感知能力，精准识别各类攻击行为以及应用层安全漏洞。应用接入 APM 以后，无需修改代码，即可迅速建立安全问题的预防以及响应机制，弥补传统安全产品在应用层威胁感知方面的不足。

说明：
- 应用安全能力完全免费。
- 应用安全能力对于探针的性能开销极低，对应用运行性能的影响几乎忽略不计。

前提条件

腾讯云增强版Java探针 2.3-20241130 及以上版本支持应用安全能力。您可以参考 K8s 环境自动接入 Java 应用（推荐） 以及 通过腾讯云增强版 OpenTelemetry Java 探针接入（推荐）将应用接入 APM，获得应用安全能力。

应用安全功能开关

应用接入 APM 以后，只要探针版本符合条件，APM 将默认开启应用安全相关功能。您可以参考 业务系统配置 以及 应用配置 提供的应用安全功能开关，按需开启或关闭应用安全能力。在进行开启或关闭操作时，应用不需要重启。

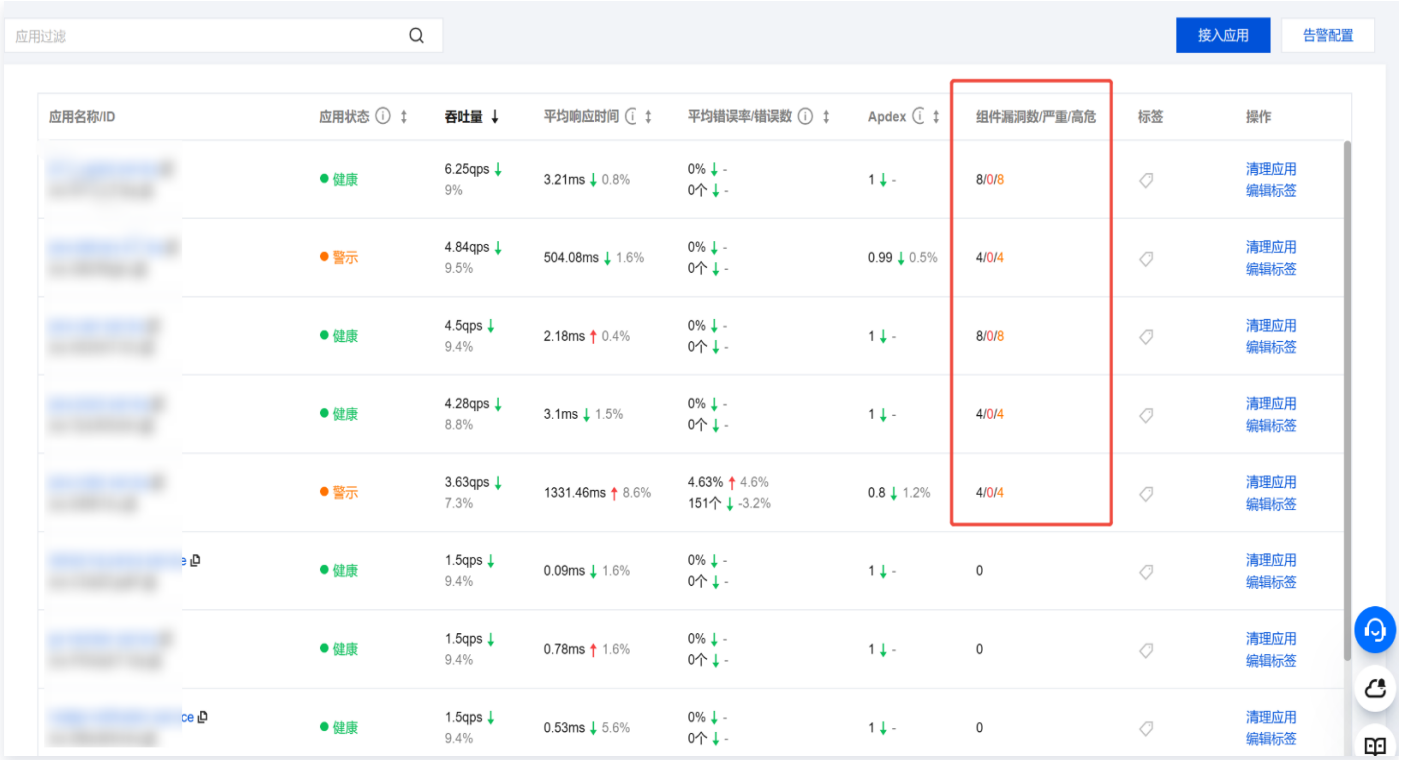
控制台使用指引

目前应用安全支持**组件漏洞安全扫描**以及 **SQL 注入分析**两项能力。

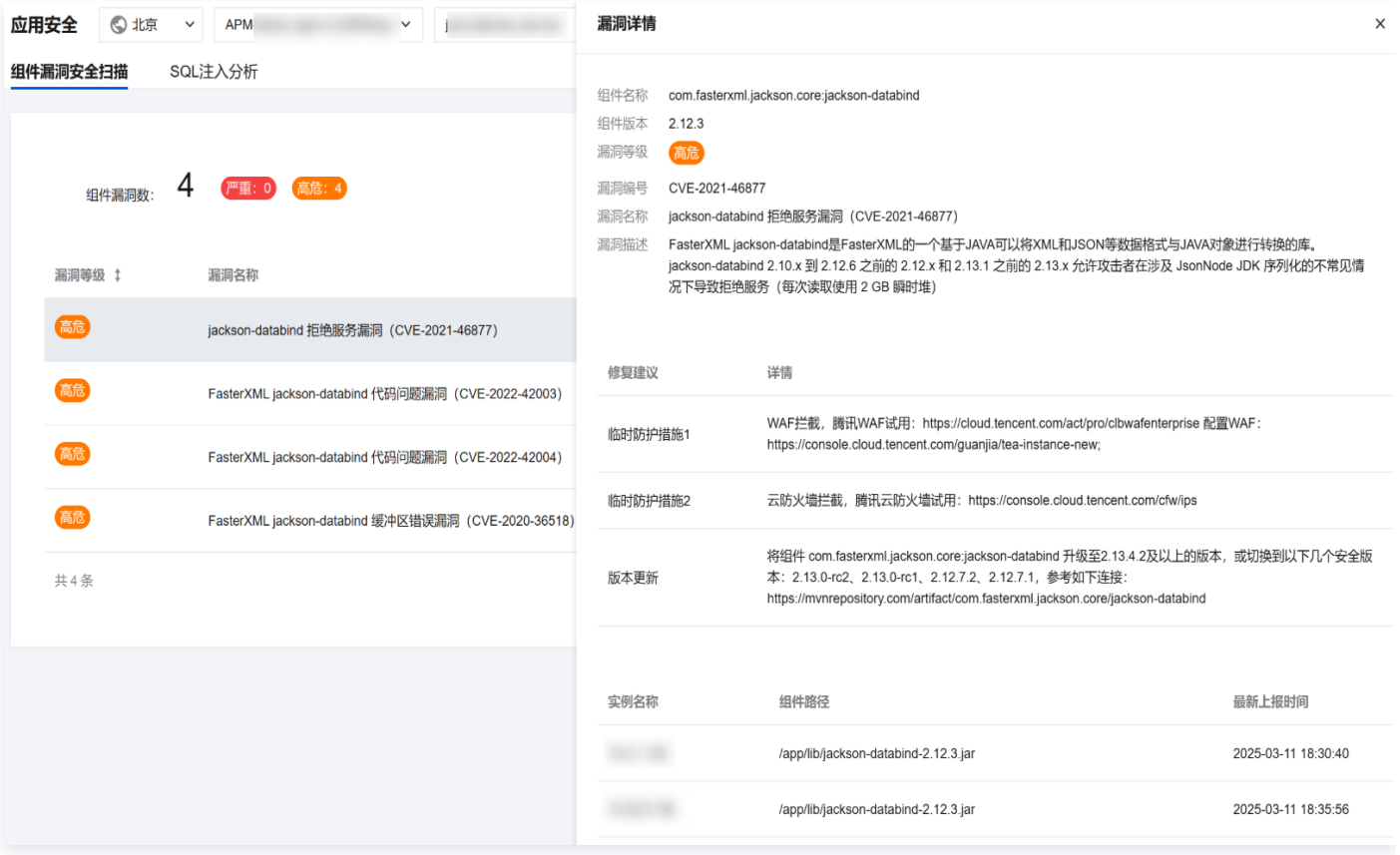
组件漏洞安全扫描

APM 基于实时更新的组件漏洞库，对存在安全漏洞的第三方组件进行自动识别，将与 CVE 编号、漏洞风险等级、修复建议等重要信息进行整合，助力研发和安全团队全面盘点危险第三方组件所带来的风险，迅速定位风险详情，并依据优先级有序开展修复工作 。

1. 开启组件漏洞安全扫描后，在 应用性能监控 > 应用列表 页面将展示每个应用被识别到的组件漏洞数量。



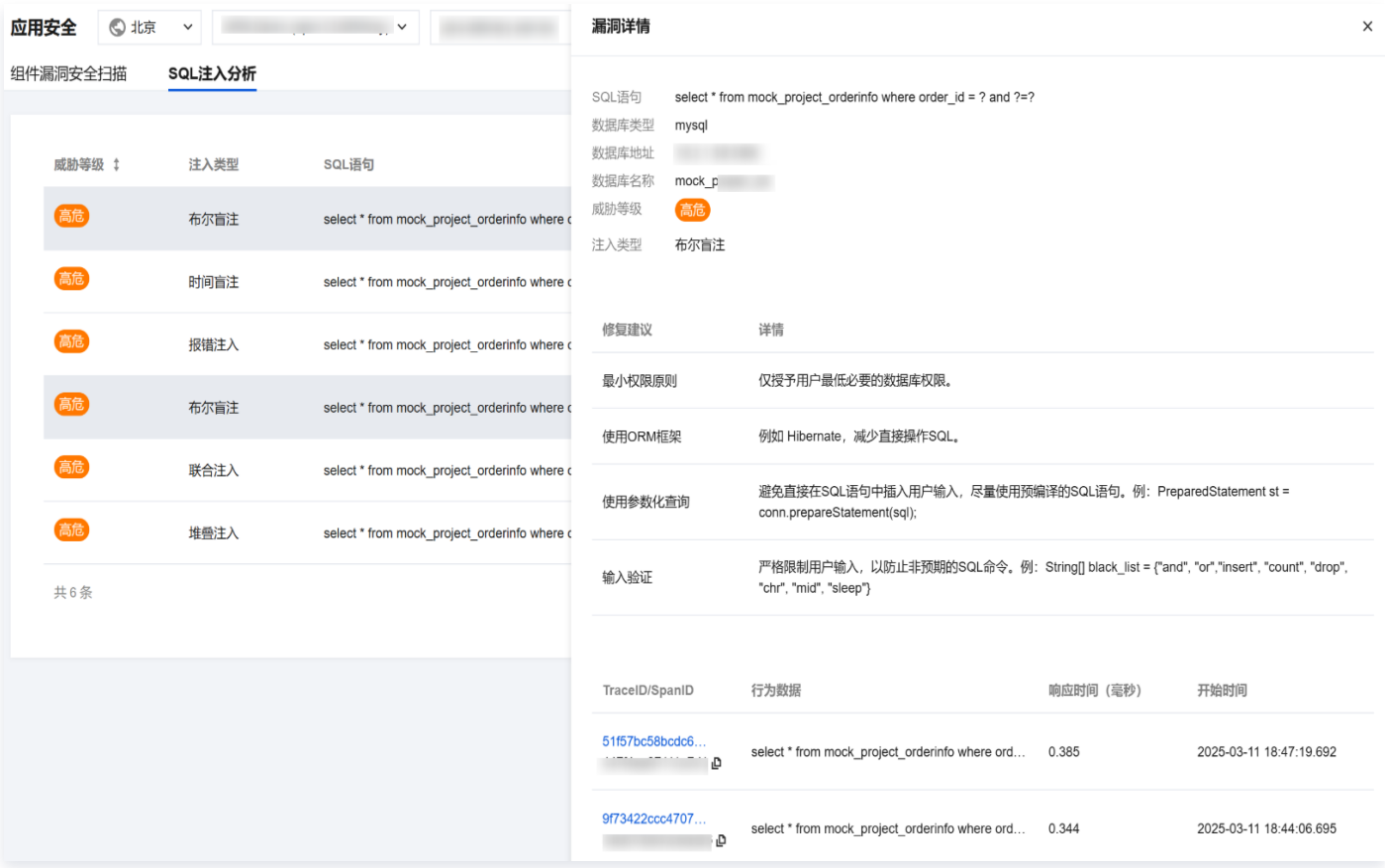
2. 单击漏洞数所对应的链接，将跳转至 **应用安全 > 组件漏洞安全扫描** 页，您可以在此页面查询应用的组件漏洞统计报告。单击列表中的漏洞，可以在滑动抽屉栏获取该漏洞的详情，包括漏洞编号、漏洞等级、漏洞描述、修复建议等重要信息，以及该漏洞在每一个应用实例中的组件路径。



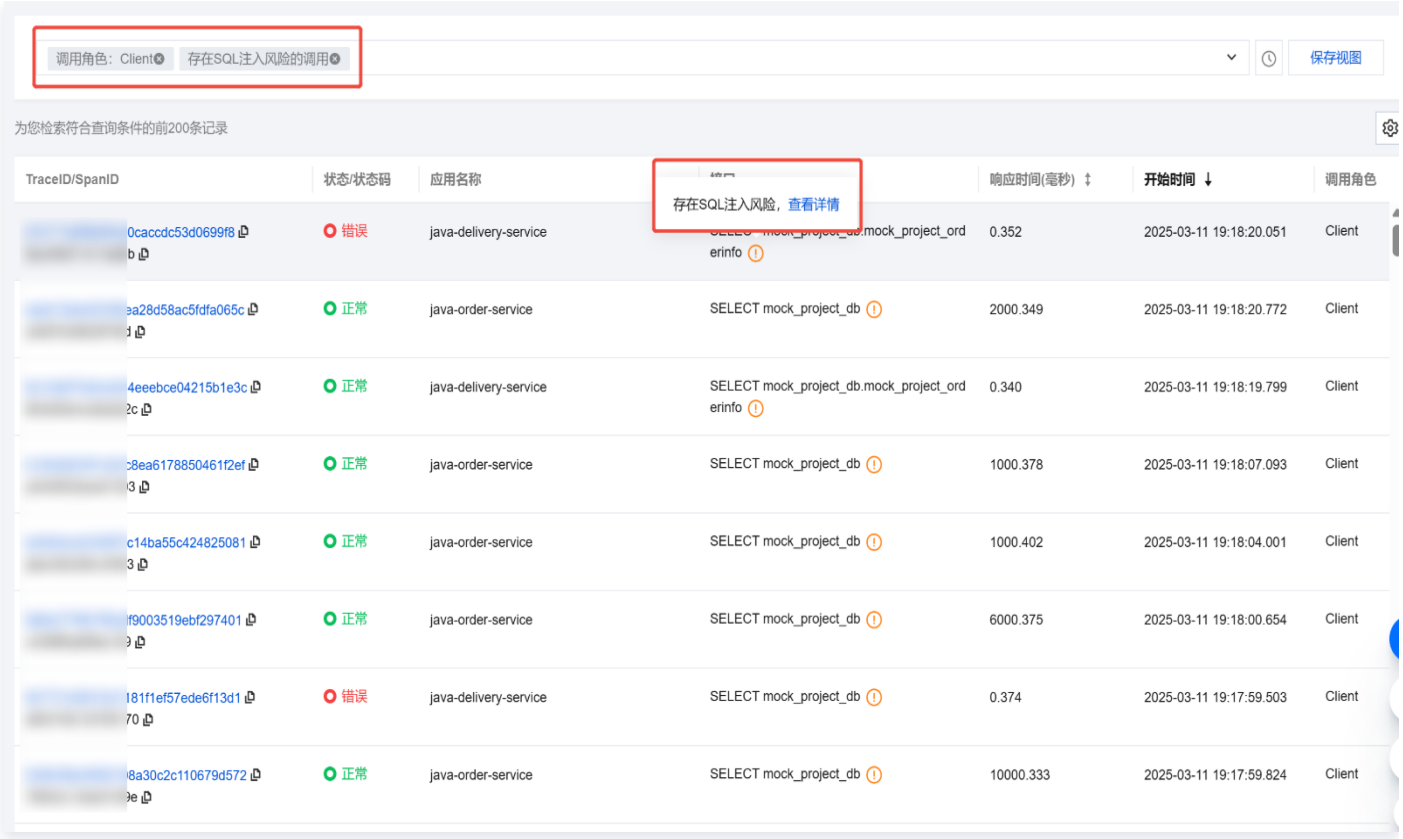
SQL 注入分析

SQL 注入分析可以实时监测应用对数据库的访问，精准识别并深度剖析潜在的 SQL 注入攻击行为，为应用数据库筑牢安全防线。

1. 开启组件漏洞安全扫描后，在 应用安全 > **SQL 注入分析**页面，您可以在此页面查询应用的 SQL 注入分析统计报告。
2. 单击列表中的 SQL 注入行为，可以在滑动抽屉栏获取该注入行为的详情，包括 SQL 语句、威胁等级、注入类型、修复建议等重要信息，以及该注入行为相关的调用链信息。



此外，在 应用性能监控 > 链路追踪 页面，可以查询存在 SQL 注入风险的调用。单击 TraceID 或者悬浮提示中的**查看详情**链接，可以在调用链路视图中获取该注入行为的详情。


调用角色：Client
存在SQL注入风险的调用
保存视图
为您检索符合查询条件的前200条记录
TraceID/SpanID
状态/状态码
应用名称
响应时间(毫秒)
开始时间
调用角色
存在SQL注入风险，查看详情
0caccdc53d0699f8
错误
java-delivery-service
erinfo
0.352
2025-03-11 19:18:20.051
Client
ea28d58ac5fdfa065c
正常
java-order-service
SELECT mock_project_db
2000.349
2025-03-11 19:18:20.772
Client
4eeebce04215b1e3c
正常
java-delivery-service
SELECT mock_project_db.mock_project_ord
erinfo
0.340
2025-03-11 19:18:19.799
Client
8ea6178850461f2ef
正常
java-order-service
SELECT mock_project_db
1000.378
2025-03-11 19:18:07.093
Client
c14ba55c424825081
正常
java-order-service
SELECT mock_project_db
1000.402
2025-03-11 19:18:04.001
Client
f9003519ebf297401
正常
java-order-service
SELECT mock_project_db
6000.375
2025-03-11 19:18:00.654
Client
181f1ef57ede6f13d1
错误
java-delivery-service
SELECT mock_project_db
0.374
2025-03-11 19:17:59.503
Client
8a30c2c110679d572
正常
java-order-service
SELECT mock_project_db
10000.333
2025-03-11 19:17:59.824
Client

实践教程
容器服务 TKE 环境应用快速接入

最近更新时间：2025-02-19 19:46:16

背景

为了尽可能的降低应用接入成本，减少 APM 工具对于整个软件研发流程的影响，腾讯云在2024年5月正式推出了 tencent-opentelemetry-operator 接入方案，对于部署在 Kubernetes 的多语言应用，提供简单高效的方式接入 APM。

应用接入方案对比

- **手动上报方案**

  这是最基本的接入方式，需要开发人员在代码中显式地调用 APM 工具提供的 API 采集和上报性能数据。优点是灵活性高，可以精确地控制需要收集的数据和收集时机；缺点是需要修改应用程序代码，工作量非常大。
- **半手动上报方案**

  在这种方案中，开发人员仍需要在代码中调用 APM 工具提供的 API，但可以基于常用的框架或库简化一部分工作。例如，一些 APM 工具提供了与常见开发框架（如 Spring、Express 等）集成的库，开发人员只需要在配置文件中启用这些库，就可以自动收集和上报性能数据。
- **探针方案**

  探针是一种自动收集性能数据的方法，它是一个运行在应用程序进程内的模块，可以在应用程序运行时动态地收集性能数据，无需修改应用程序代码。探针方案同样提供了对常见开发框架的集成，可以自动收集和上报性能数据。相比手动上报方案，部署更加简单，可以在应用编译后进行引入，甚至可以做到对整个开发过程无感知。
- **Sidecar 方案**

  Sidecar 是一种运行在应用程序进程外的模块，通常以单独的进程或容器的形式存在，可以监听应用程序的网络通信，收集性能数据。优点是无需修改应用程序代码，且对应用程序性能的影响非常小；缺点是 Sidecar 方案部署复杂，需要管理额外的进程或容器，只能基于 Service Mesh（服务网格）等特殊应用架构才能发挥价值。
- **eBPF 方案**

  eBPF（Extended Berkeley Packet Filter）是 Linux 内核中的一种技术，可以在内核中运行用户定义的程序，收集各种系统和网络性能数据。使用 eBPF 的 APM 工具可以提供非常详细和精确的性能数据，且对应用程序性能的影响极小。对于已经部署到 Kubernetes 的容器化应用，eBPF 方案的部署比较简单，但和 Sidecar 方案一样，缺少成熟的分布式链路追踪能力。

接入方案	接入难度	兼容性	功能覆盖度	成熟度
手动上报	极高	高	高	高
半手动上报	高	高	高	高
探针	低	比较高。适合大多数语言	高	高
Sidecar	低	低。依赖特定应用架构	低	高
eBPF	低	比较高。依赖高版本内核	中	低

综合来看，对于 APM 方案的选型，接入难度是首要的考虑因素。如果接入 APM 工具还需要修改业务代码，会给开发团队带来沉重的负担，长远来看必将在团队内部造成推广困难的局面，这也是很多企业在引入 APM 工具的过程中放弃的主要原因。因此，我们需要尽量排除手动上报方案，虽然 eBPF 方案代表着未来 APM 的发展方向，但成熟度还很低，功能覆盖度上也有比较大的不足，因此探针

方案依然是目前最成熟、最值得考虑的选择，特别是对于 Java、Python 等基于虚拟机运行的编程语言，天然就和探针方案有极高的匹配度。

Operator 方案

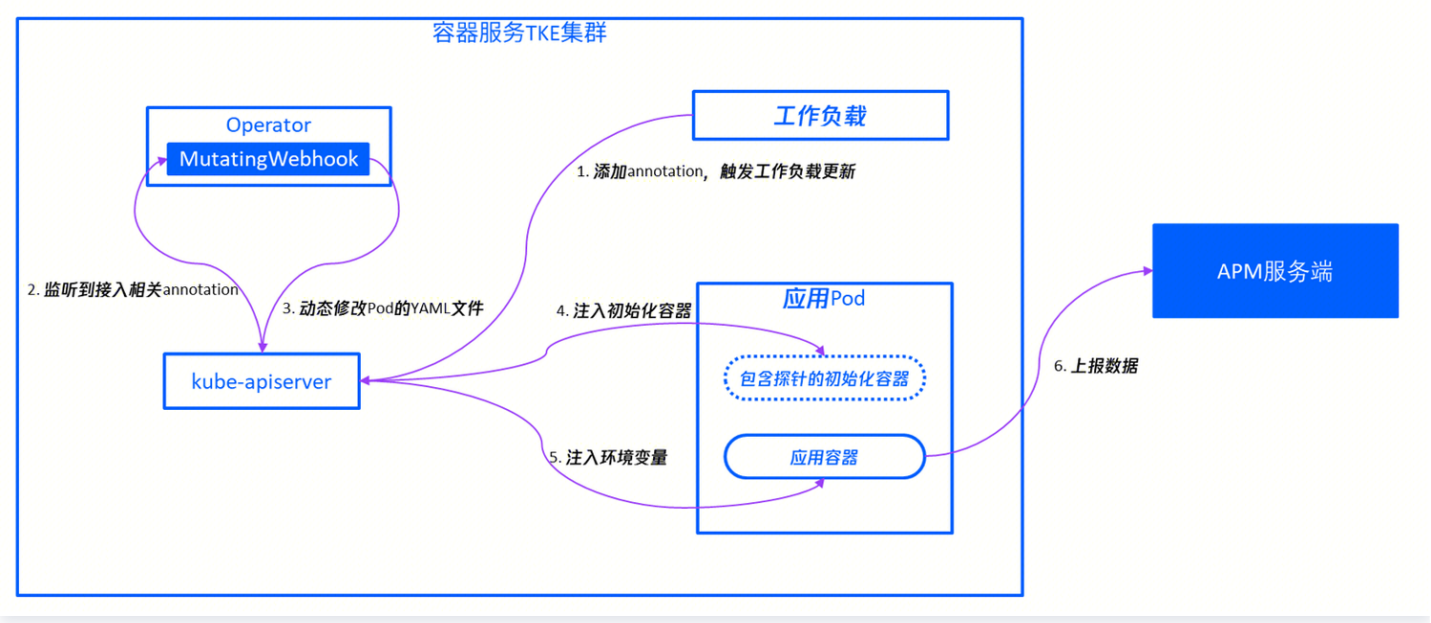
tencent-opentelemetry-operator 由腾讯云在社区 opentelemetry-operator 基础上构建，是基于探针的接入方案。由于实现了 Kubernetes 环境下探针自动注入机制，免去了探针部署和配置的工作量，让应用接入变得更加简单。目前 tencent-opentelemetry-operator 支持 Java、Python、node.js、.Net 应用的快速接入，应用程序部署到腾讯云容器服务 TKE 以后，只需要在工作负载的 YAML 文件中添加2行 annotation，就能完成整个接入流程，无需在开发态涉及任何代码的修改。

接入优势

- **简化 Operator 安装流程**：安装部署流程，不但上架了 TKE 应用市场，还能在 APM 控制台实现 Operator 一键安装。
- **通过封装减少人力**：对接入过程中需要用到的一系列配置项进行了封装，不再需要手工填写，包括 APM 接入点的地址，业务系统 token 等。
- **优化证书管理机制**：解决了 Operator 安装过程中有可能遇到的各类兼容性问题。
- **降级处理**：当 APM 平台的任何一个组件出现故障的时候，整个接入机制都以不影响业务正常运行为前提，在必要的时候对探针注入和监控数据上报逻辑行降级处理，确保业务的稳定运行。

接入原理

Operator 方案利用了 Kubernetes 提供的 Dynamic Admission Control 机制实现探针的注入，可以参考以下流程了解 Operator 方案的接入原理：



1. 用户对工作负载添加 annotation，API Server 将基于工作负载的发布策略启动工作负载的更新。
2. 在 Operator 的相关组件中，包含一个具备 Dynamic Admission Control 能力的工作负载，通过 Mutating Webhook 的形式在容器服务集群中运行，这个组件可以从 API Server 监听工作负载的变化。
3. Operator 监听到应用工作负载添加了接入相关的 annotation 之后，会通过 API Server 对即将创建的应用 Pod 进行动态修改，修改的内容通过 Pod 的 YAML 描述表现出来。
4. Operator 会在应用 Pod 中注入一个初始化容器，在容器本地路径中包含了探针文件。

5. Operator 还会在应用容器中注入一些环境变量，这些环境变量可以对应用进程进行配置，实现挂载探针的动作。以 Java 应用为例，Operator 会对应用容器添加 JAVA_TOOL_OPTIONS 环境变量，其中包括 -javaagent 参数，指向由初始化容器引入的探针文件本地路径，使 Java 进程启动的时候执行挂载探针操作。
6. 探针通过字节码增强机制自动采集监控数据，并向 APM 服务端上报。

操作步骤

安装 tencent-opentelemetry-operator

1. 登录 腾讯云可观测平台 。
2. 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**。
3. 在弹出框选择需要接入的编程语言，例如 Java。
4. 进入**接入 Java 应用**页，选择 **TKE 环境自动接入**上报方式，然后单击**一键安装 Operator**。
5. 进入一键安装 Operator 的弹出窗口，通过下拉框选择上报地域、默认业务系统、TKE 集群等信息，确认后单击**安装**即可。

> **说明：**
> 上报地域为 APM 业务系统所在的地域，每个 TKE 集群都只能指定唯一的上报地域，最好与 TKE 集群所在的地域保持一致。

6. 安装完成之后，前往 容器服务 TKE 控制台 ，在左侧菜单栏中选择**运维中心** > **应用市场**，进入**应用管理**页，就可以查询到安装在 `kube-system` 命名空间的 `tencent-opentelemetry-operator` 应用，也可以通过 Helm 客户端执行 `helm list` 命令检查安装状态。

应用接入

在 TKE 集群完成 Operator 的安装后，得益于 Operator 方案动态探针注入能力，应用接入的过程极为简单，只需要编辑应用所在工作负载的 YAML 文件，在 Pod Template 中添加2个 annotation，即可完成应用接入。以 Java 应用为例，需要添加的内容如下：

```
cloud.tencent.com/inject-java: "true" #接入APM
cloud.tencent.com/otel-service-name: my-app #指定应用名
```

其中，otel-service-name 字段代表应用名，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。完整的工作负载 YAML 文件可以参考以下代码片段：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
  template:
    metadata:
```

```
    labels:
      k8s-app: my-app
    annotations:
      cloud.tencent.com/inject-java: "true"  #接入APM
      cloud.tencent.com/otel-service-name: my-app #指定应用名
  spec:
    containers:
      image: my-app:0.1
      name: my-app
```

添加完 annotation 之后，将基于不同的工作负载发布策略对工作负载进行更新，触发应用 Pod 的重新创建。新启动的 Pod 会自动注入探针，并连接到 APM 服务端，将监控数据上报到 Operator 的默认业务系统。前往 腾讯云可观测平台控制台，在**应用性能监控 > 应用列表**中，就能查询到新接入的应用。
更多应用接入指引，请参考下述帮助文档：

应用	参考文档
Java	K8s 环境自动接入 Java 应用（推荐）
Python	K8s 环境自动接入 Python 应用（推荐）
Node.js	K8s 环境自动接入 Node.js 应用（推荐）

自定义埋点

在探针接入方案的基础上，Operator 模式通过探针注入进一步简化了应用接入 APM 的流程。tencent-opentelemetry-operator 注入的探针来自 OpenTelemetry 生态，对主流的开发框架与类库实现了运行态自动埋点。对大多数用户而言，应用在接入成功后即可完成监控数据上报，实现分布式链路追踪，不需要修改任何代码。但在某些特殊的场景中，如果探针的自动埋点机制不能满足用户需求，用户可以引入 OpenTelemetry API，在自动埋点的基础上，通过修改应用代码增加自定义埋点。例如下述场景中，可能需要引入自定义埋点：

- 应用引入的开发框架与类库没有在开发者中被广泛使用，或者版本很低。
- 应用引入的开发框架与类库由用户自己封装，或者来自于闭源商业化定制开发。
- 对于某些核心的自定义方法，需要在链路中追加方法级的埋点。

对于 tencent-opentelemetry-operator 注入的探针，可以参考 OpenTelemetry 方案支持的组件和框架 了解自动埋点的范围。自定义埋点代码的具体编写方式，可以参考如下 OpenTelemetry 的官方文档：

应用	参考链接
Java	https://opentelemetry.io/docs/languages/java/instrumentation/
Python	https://opentelemetry.io/docs/languages/python/instrumentation/
Node.js	https://opentelemetry.io/docs/languages/js/instrumentation/

以 Java 语言为例，如果我们需要对某个自定义方法在链路中追加方法级埋点，可以参考以下步骤实现。
1. 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>
```

2. 通过如下代码，在链路中增加 doTask() 的方法级埋点。

```
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();
    }
}
```

3. 前往 腾讯云可观测平台控制台，在**应用性能监控 > 链路追踪**中，找到相关的调用链，单击 Span ID 后，在**链路详情**页面中，即可查到通过自定义埋点新增的 Span。

总结

腾讯云应用性能监控（APM）为企业提供了全面、高效、易用、低成本的应用性能管理解决方案，帮助企业优化应用程序的性能。新发布的 Operator 方案，降低了接入成本，将接入 APM 工具的决策时间点从开发态后移至部署态，用户不再需要在代码中引入 APM 相关 SDK 或者将探针文件打包到容器镜像中。这实现对应用开发的零侵入，对于在团队内部全部推广 APM 工具，有着非常重要的意义。

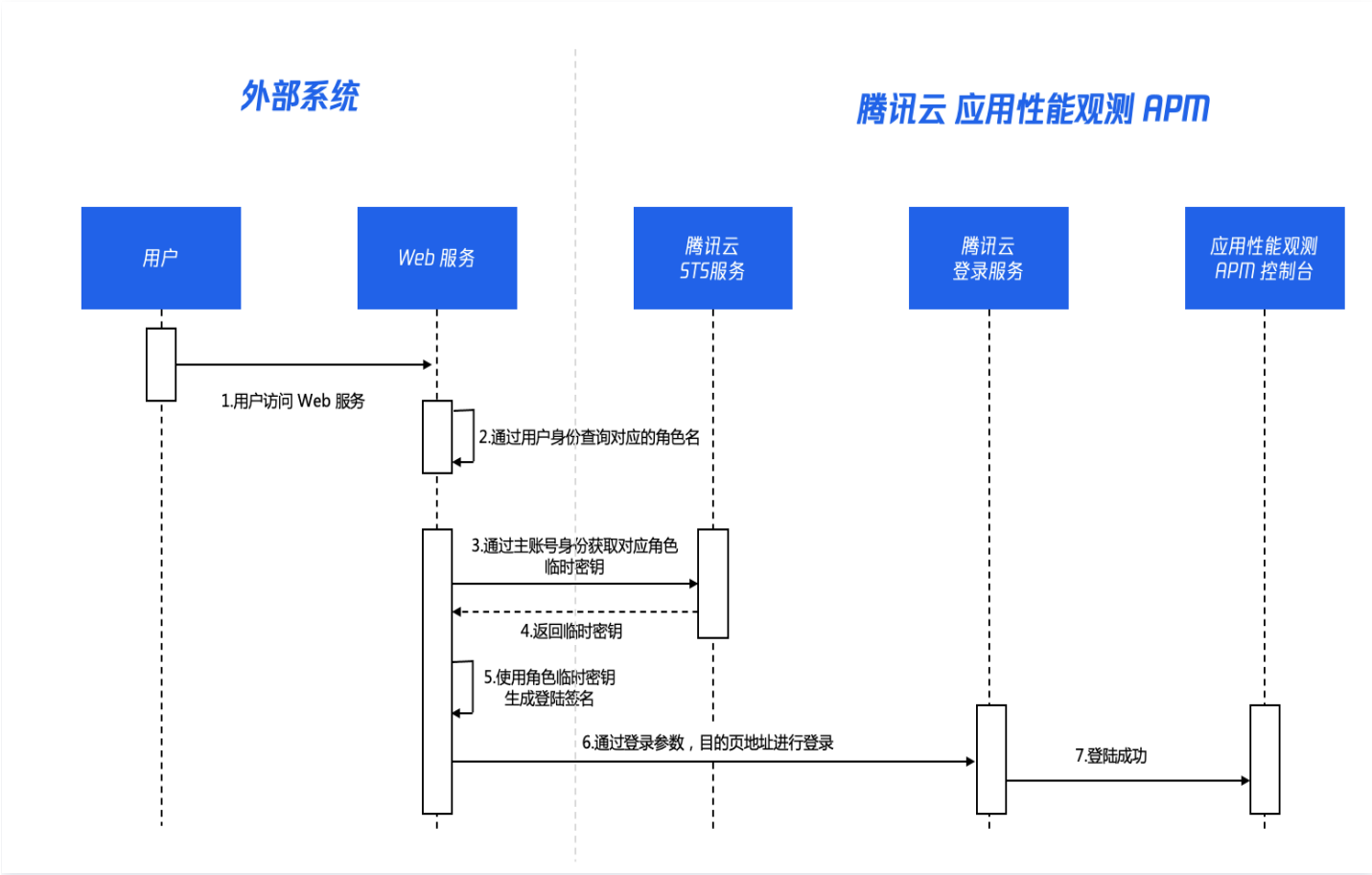
将 APM 控制台嵌入自建系统

最近更新时间：2024-08-01 14:40:04

您可以将 APM 控制台嵌入到自建 Web 服务中，为用户提供以下方便：

- 在外部系统服务中（例如公司内部运维系统）快速集成 APM 数据的查询分析能力。
- 无需管理众多腾讯云子账号，方便将 APM 数据分享给他人进行查看。
- 用户不需要登录腾讯云就可以使用 APM。

整体的交互流程如下：



前提条件

该方案需要在自建 Web 服务中，通过代码获取当前用户的访问密钥，具体的获取方式由您的业务场景自行决定，不在本文的讨论范围之内。您可以参考 子账号访问密钥管理 预先对用户访问密钥进行持久化保存。

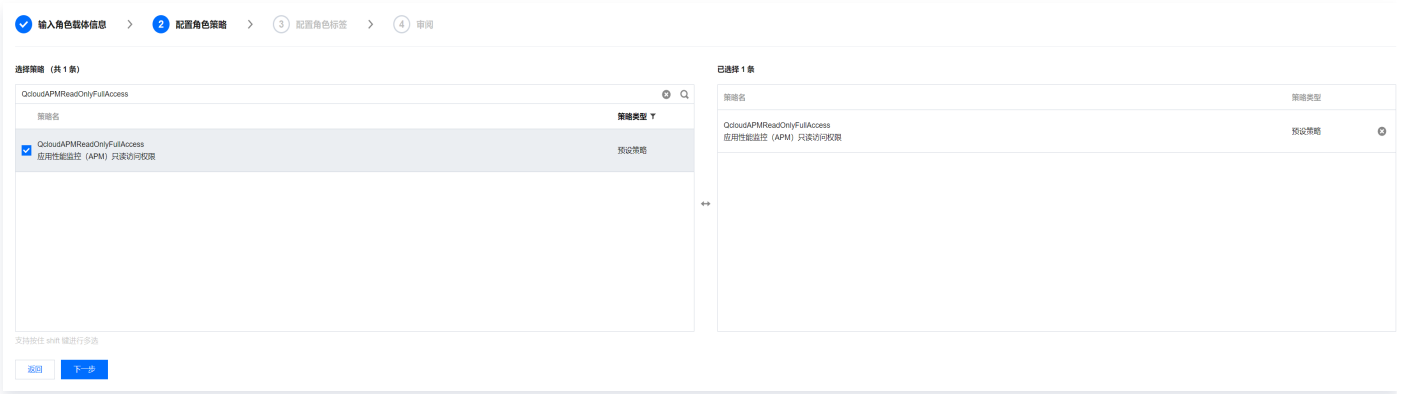
步骤1：创建 CAM 角色

用户访问 Web 服务后，需要通过扮演角色的方式登录 APM 控制台，因此需要预先创建 CAM 角色。关于角色的基本概念，请参考 角色概述 。您可以为多个用户创建统一的角色，也可以为每个用户创建单独的角色，具体的实现方式由您的业务场景决定，不在本文的讨论范围之内。您可以通过控制台或 创建角色API 创建对应的角色，本文将介绍如何通过控制台创建角色。

1. 登录 访问管理 CAM 控制台 。
2. 单击左侧菜单栏中的**角色**，进入角色页面。
3. 选择**新建角色 > 腾讯云账户**，开始新建自定义角色。

4. 选择**当前主账号**并勾选**允许当前角色访问控制台**，单击**下一步**。

5. 为角色关联 APM 的权限策略，例如全读写访问权限 `QcloudAPMFullAccess`，或者只读访问权限 `QcloudAPMReadOnlyFullAccess`，**单击下一步**。



6. 为角色配置不同维度标签，使用标签对用户进行分类管理，然后单击**下一步**。

7. 输入角色名，点击**完成**即可创建成功。

步骤2：申请扮演角色临时访问凭证

在 Web 服务中，需要基于用户的访问密钥，访问腾讯云 STS 服务的 AssumeRole 接口，申请对应 CAM 角色的临时访问凭证。本文提供了 Java 代码实现，其他语言的代码可以参考 AssumeRole 接口调试，其中的关键入参包括：

参数名称	是否必选	类型	描述
RoleArn	是	String	需要扮演的角色的 ARN，可以从 CAM 控制台获取，例如 `qcs::cam::uin/1500000688:roleName/RoleWithAPMFullAccess`
RoleSessionName	是	String	临时会话名称，由用户自定义名称
DurationSeconds	是	Int	指定临时访问凭证的有效期，单位为秒。建议设置在5分钟以内

腾讯云 STS 服务将返回扮演角色临时访问凭证，包括 Secret ID、Secret Key 和 Token。

步骤3：生成登录 URL

1. 拼接需要签名的字符串

对要求签名的参数按照字母表或数字表递增顺序的排序，先考虑第一个字母，在相同的情况下考虑第二个字母，以此类推。您可以借助编程语言中的相关排序函数来实现这一功能，如 PHP中 的 ksort 函数。其中要求签名的参数包含以下内容：

参数名称	是否必选	类型	描述
action	是	String	操作动作，固定为 roleLogin
timestamp	是	Int	当前时间戳
nonce	是	Int	随机整数，取值 10000-100000000
secretId	是	String	从 STS 服务返回的临时 Secret ID

将把上一步排序好的请求参数，按 `参数名称=参数值` 的格式拼接成请求字符串，并使用 `&` 符合分隔，例如：

```
action=roleLogin&nonce=67439&secretId=AKI***PLE&timestamp=1484793352
```

按 `请求方法 + 请求路径 + ? + 请求字符串` 的规则，拼接需要签名的字符串。

参数	必选	描述
请求路径	是	固定为 cloud.tencent.com/login/roleAccessCallback
请求方法	是	支持 GET 或 POST

需要签名的字符串示例：

```
GETcloud.tencent.com/login/roleAccessCallback?
action=roleLogin&nonce=67439&secretId=AKI***PLE&timestamp=1484793352
```

2. 签名

使用 HMAC-SHA1 或 HMAC-SHA256 算法对字符串进行签名，签名时用到的 key 是从 STS 服务获得的临时 Secret Key，具体实现方式可以参考本文的示例代码。

3. 拼接最终的登录 URL

根据以下规则，拼接最终的登录 URL，其中**每一个参数都需要通过 URLEncoder 进行编码**。

```
https://cloud.tencent.com/login/roleAccessCallback
        ?algorithm=<签名时加密算法，目前只支持sha1和sha256>
        &secretId=<从STS服务返回的临时Secret ID>
        &token=<从STS服务返回的临时Token>
        &nonce=<签名时用到的nonce>
        &timestamp=<签名时用到的timestamp>
        &signature=<签名结果>
        &s_url=<目标URL，固定为 https://console.cloud.tencent.com/apm?
hideWidget=true&hideTopNav=true >
```

步骤4：用户访问登录 URL 进入 APM 控制台

可以通过 `<iframe>` 等方式，引导用户访问登录 URL，这样就能将 APM 控制台嵌入到自建系统中，最终用户并不需要再登录腾讯云就能访问 APM。

完整代码

请参考如下 Java 代码完成开发，代码分为两部分：

1. 向腾讯云 STS 服务申请扮演角色临时访问凭证。其他语言的代码可以参考 API 文档 。
2. 拼接登录 URL。其他语言的代码可以借助于工具转换，或者参考 Java 代码自行编写。

```
import java.net.URLEncoder;
import java.util.Random;

import com.tencentcloudapi.common.Credential;
import com.tencentcloudapi.common.profile.ClientProfile;
import com.tencentcloudapi.common.profile.HttpProfile;
import com.tencentcloudapi.sts.v20180813.StsClient;
import com.tencentcloudapi.sts.v20180813.models.*;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class LoginWithoutPass {
    public static void main(String[] args) throws Exception {
        /*
        以下代码用于申请扮演角色临时访问凭证，其他语言的代码可以参考API文档
        https://cloud.tencent.com/document/product/1312/48197
        */
        // 根据自身业务逻辑，获取当前用户的访问密钥
        String secretId = getSecretId();
        String secretKey = getSecretKey();
        Credential cred = new Credential(secretId, secretKey);
        HttpProfile httpProfile = new HttpProfile();
        httpProfile.setEndpoint("sts.tencentcloudapi.com");
        ClientProfile clientProfile = new ClientProfile();
        clientProfile.setHttpProfile(httpProfile);
        // 地域列表请参考 https://cloud.tencent.com/document/product/213/6091
        StsClient client = new StsClient(cred, "ap-guangzhou", clientProfile);
        AssumeRoleRequest req = new AssumeRoleRequest();
        // 需要扮演的角色的ARN，可以从CAM控制台获取
        req.setRoleArn("qcs::cam::uin/1500000688:roleName/RoleWithAPMFullAccess");
        // 临时会话名称，由用户自定义名称
        req.setRoleSessionName("cts");
        // 临时访问凭证的有效期，建议5分钟以内
        req.setDurationSeconds(1800L);
        AssumeRoleResponse resp = client.AssumeRole(req);
        // 从腾讯云STS服务获取的临时访问凭证，包括密钥和Token
        String stsSecretId = resp.getCredentials().getTmpSecretId();
        String stsSecretKey = resp.getCredentials().getTmpSecretKey();
```

```
        String token = resp.getCredentials().getToken();


        /*
        以下代码用于拼接登录URL，其他语言的代码可以借助于工具转换，或者自行编写
        */
        int timestamp = (int) (System.currentTimeMillis() / 1000);
        // 生成10000-100000000之间的随机数
        Random rand = new Random();
        int min = 10000;
        int max = 100000000;
        int nonce = rand.nextInt((max - min) + 1) + min;
        // 拼接需要签名的字符串
        String sourceString = "GETcloud.tencent.com/login/roleAccessCallback?
action=roleLogin&nonce=" + nonce +
                "&secretId=" + stsSecretId + "&timestamp=" + timestamp;
        // 通过SHA256进行签名
        String signatureString =
Base64.getEncoder().encodeToString(hmacSha256(sourceString, stsSecretKey));
        // 拼接最终的登录URL
        String result = "https://cloud.tencent.com/login/roleAccessCallback?
algorithm=sha256&secretId=" +
                URLEncoder.encode(stsSecretId, StandardCharsets.UTF_8.toString()) +
                "&token=" + URLEncoder.encode(token,
StandardCharsets.UTF_8.toString()) +
                "&nonce=" + nonce +
                "&timestamp=" + timestamp +
                "&signature=" + URLEncoder.encode(signatureString,
StandardCharsets.UTF_8.toString()) +
                "&s_url=" + URLEncoder.encode("https://console.cloud.tencent.com/apm?
hideWidget=true&hideTopNav=true", StandardCharsets.UTF_8.toString());
         // 接下来引导用户访问生成的登录URL，进入APM控制台
    }

    private static byte[] hmacSha256(String data, String key) throws Exception {
        SecretKeySpec secretKeySpec = new
SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), "HmacSHA256");
        Mac mac = Mac.getInstance("HmacSHA256");
        mac.init(secretKeySpec);
        return mac.doFinal(data.getBytes(StandardCharsets.UTF_8));
    }
}
```

链路与日志关联分析

最近更新时间：2024-12-27 15:44:12

实践背景

随着微服务架构的逐渐流行，在复杂且庞大的系统中准确地定位一个请求的完整生命周期，逐渐成为了研发人员面临的最大痛点之一。以研发自测过程为例，开发人员通常希望在发起 HTTP/RPC 测试请求后，能够以一个简单的方式获取整个测试请求的上下文信息。这其中通常包括相关的上下游链路、各个服务内部请求的方法堆栈，以及链路上打印的日志等数据，对于指标-链路-日志的一体化监控的需求已经迫在眉睫。
腾讯云应用性能监控（APM）作为本身包含**指标-链路**的全链路监控平台，已与 腾讯云日志服务（CLS） 联手打通，实现**指标-链路-日志**的一体化监控，在查看链路详情时关联查看所对应信息，完成从链路到日志的排查流程。

前提条件

请确保应用已经接入应用性能监控（APM），并参考 CLS 入门指南 将业务日志上报到腾讯云日志服务（CLS）。

> **说明：**
> 为了保障您日志数据的可靠性以及更高效地使用日志服务，建议您使用 CLS 优化后的接口/Agent 上传结构化日志：
> - 通过 API/SDK 上报，详情请参见 上传日志 中的 contents 部分。
> - 通过 Agent 上报，详情请参见 使用 JSON 提取模式采集日志 。

操作步骤

步骤1：将 TraceID 输出到日志

为了实现链路和日志的关联查询，您需要将 TraceID 输出到日志中，对于如下2种方式接入 APM 的应用，可以在不修改代码的情况，将 TraceID 自动注入到日志中：

- 通过腾讯云增强版 OpenTelemetry 探针接入的 Java 应用
- 通过 SkyWalking 方案接入的 Java 应用

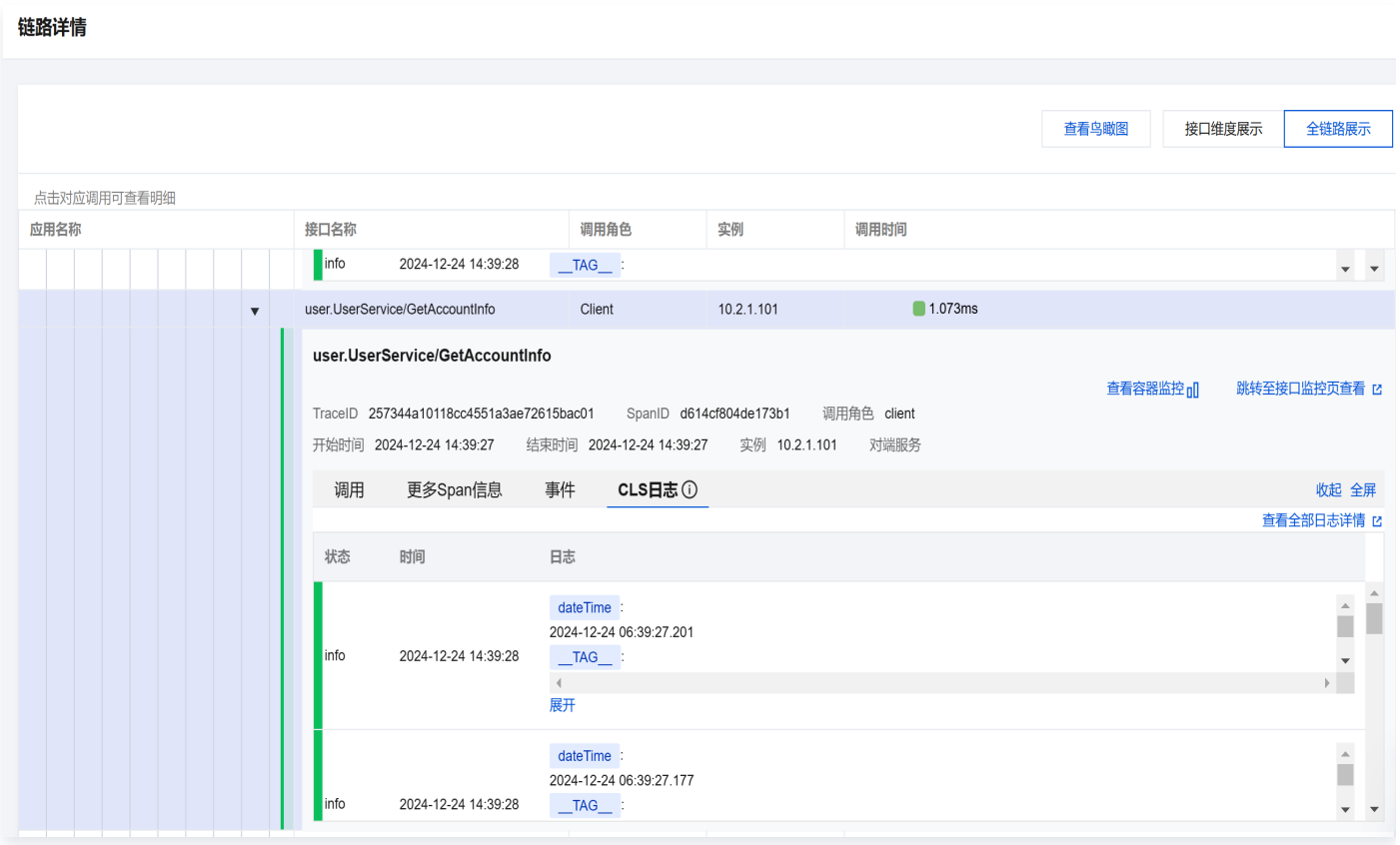
对于通过其他方式接入 APM 的应用，请参考对应的接入文档，在代码中获取 TraceID，并在业务日志中输出 TraceID。

步骤2：配置 APM 应用与 CLS 主题的关联

请参考 在系统配置页面关联日志，配置 APM 应用与 CLS 主题的关联。

步骤3：查询链路对应的业务日志

1. 登录 腾讯云可观测平台，进入**应用性能监控 > 链路追踪**页面。
2. 选择对应的业务系统，指定查询条件，查询链路调用，并单击 TraceID，即可进入**链路详情**页面。
3. 在 **CLS日志**标签页，可以查询该链路对应的业务日志。


链路详情
查看鸟瞰图
接口维度展示
全链路展示
点击对应调用可查看明细
应用名称
接口名称
调用角色
实例
调用时间
info 2024-12-24 14:39:28 __TAG__ :
user.UserService/GetAccountInfo Client 10.2.1.101 1.073ms
user.UserService/GetAccountInfo
查看容器监控
跳转至接口监控页查看
TraceID 257344a10118cc4551a3ae72615bac01 SpanID d614cf804de173b1 调用角色 client
开始时间 2024-12-24 14:39:27 结束时间 2024-12-24 14:39:27 实例 10.2.1.101 对端服务
调用
更多Span信息
事件
CLS日志
收起 全屏
查看全部日志详情
状态
时间
日志
info 2024-12-24 14:39:28 dateTime : 2024-12-24 06:39:27.201 __TAG__ :
展开
info 2024-12-24 14:39:28 dateTime : 2024-12-24 06:39:27.177 __TAG__ :

使用 Tencent Cloud APM Grafana 插件配置 APM Grafana 面板

最近更新时间：2024-07-05 11:48:11

本文将为您介绍如何使用 Tencent Cloud APM Grafana 插件配置 APM Grafana 面板。

操作背景

APM-Grafana 数据源插件提供了 APM 常用的指标功能，并且具有很好的扩展性，用户可以自定义指标视图。

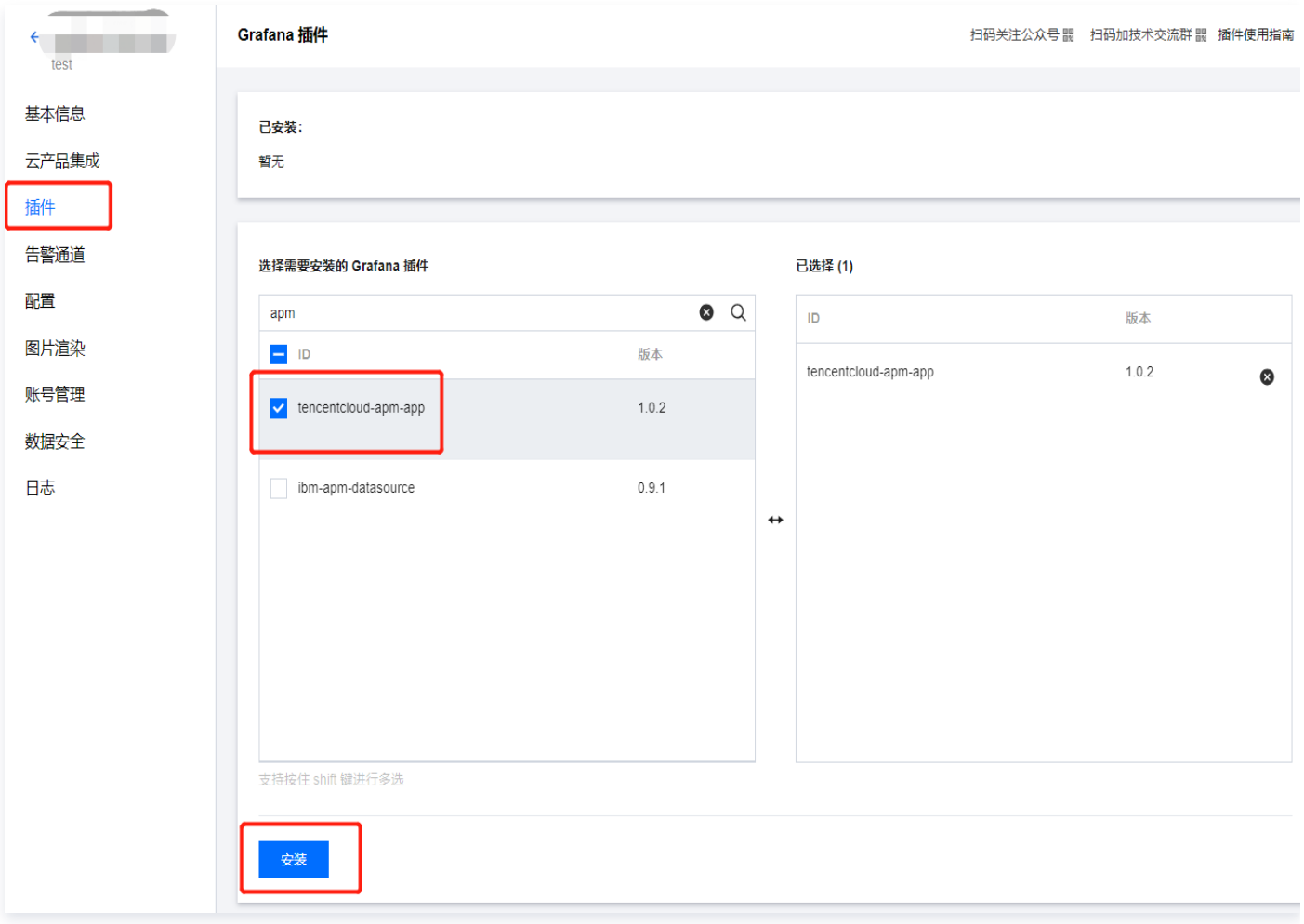
操作前提

- 采用 APM 进行数据上报
- 申请云 API SecretId 和 SecretKey

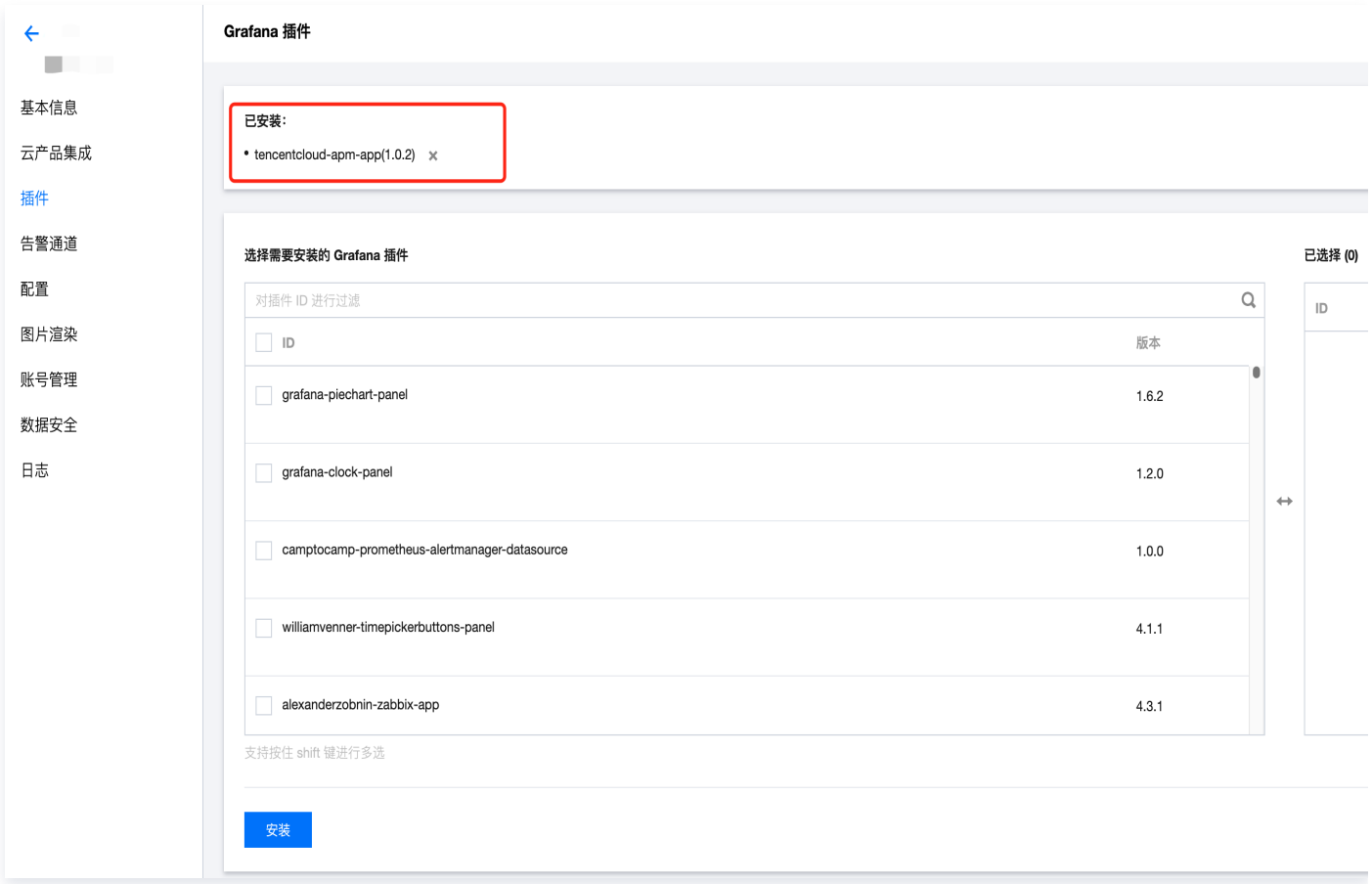
操作步骤

1. 登录 Grafana 服务，点击**管理**。

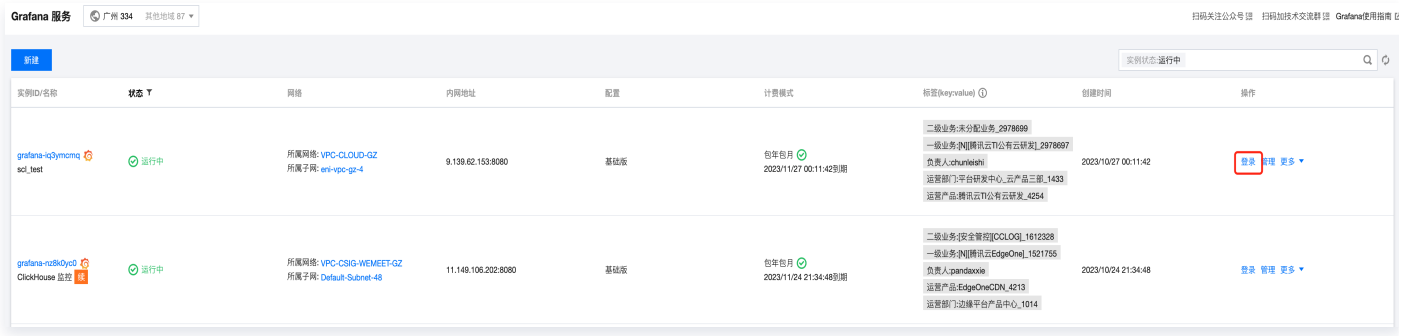
2. 点击**插件**，搜索 tencentcloud-apm-app ，选中后点击**安装**。



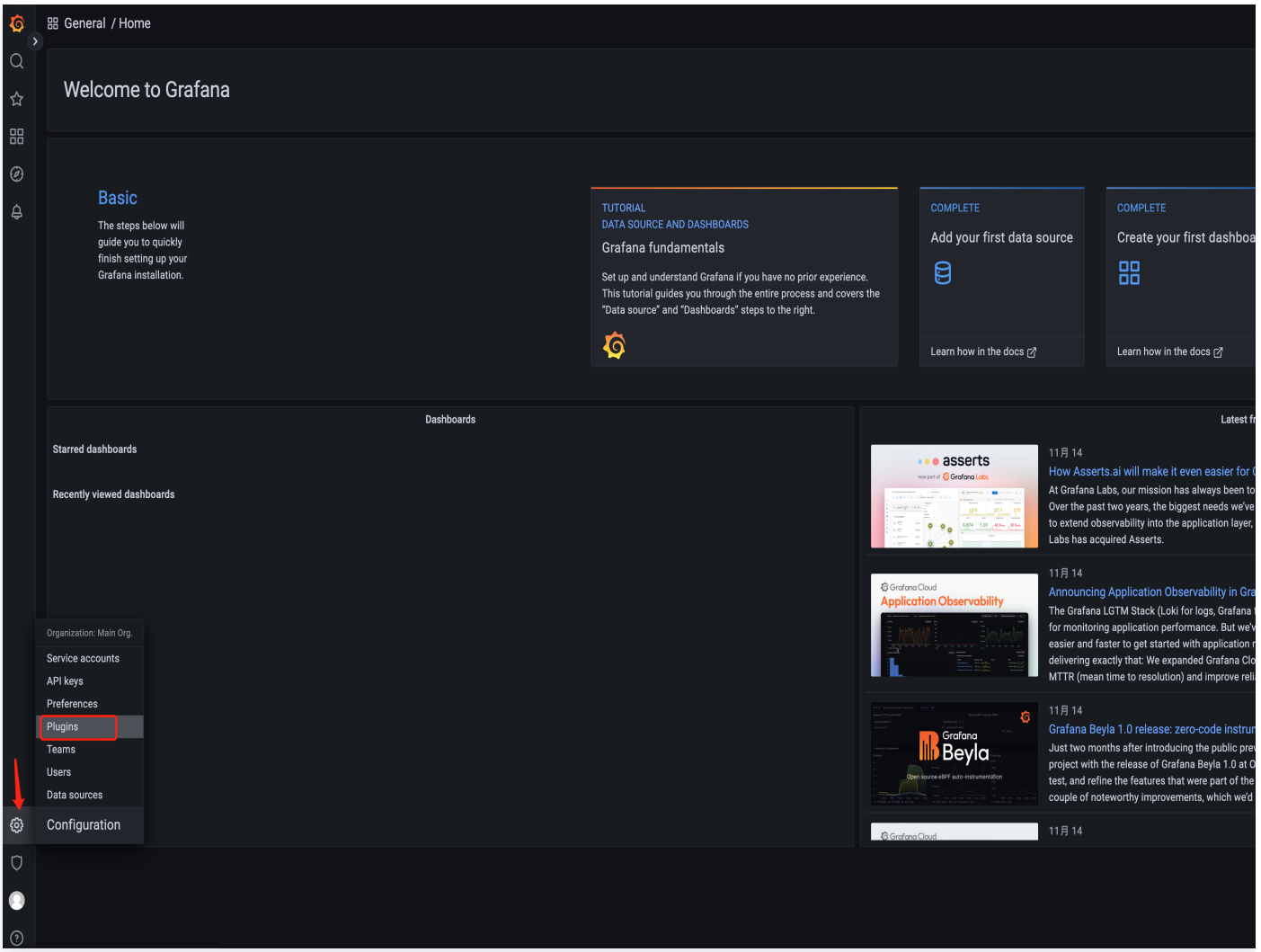
3. 安装成功后如下图所示：



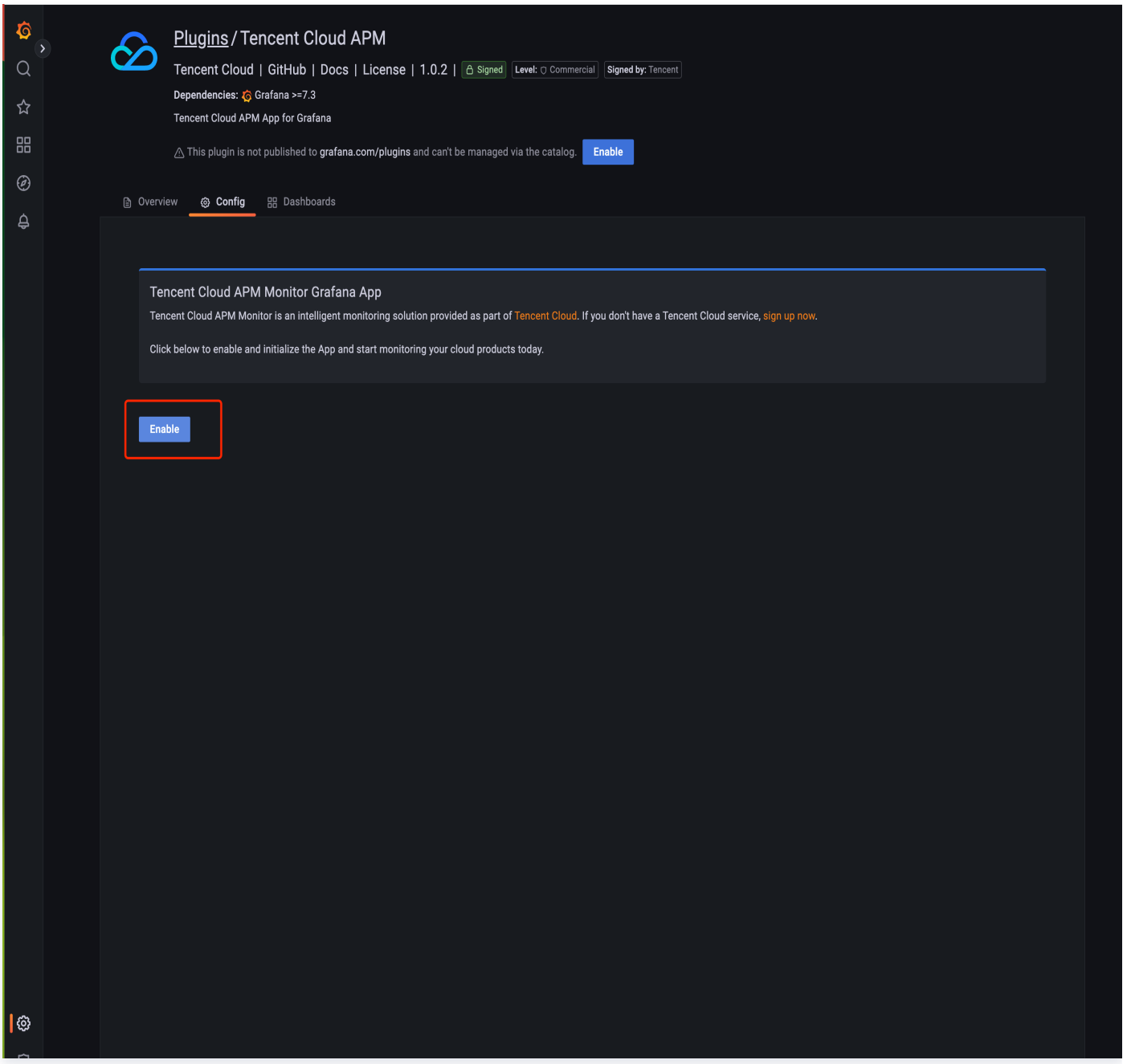
4. 返回 Grafana 服务列表页，点击**登录**。



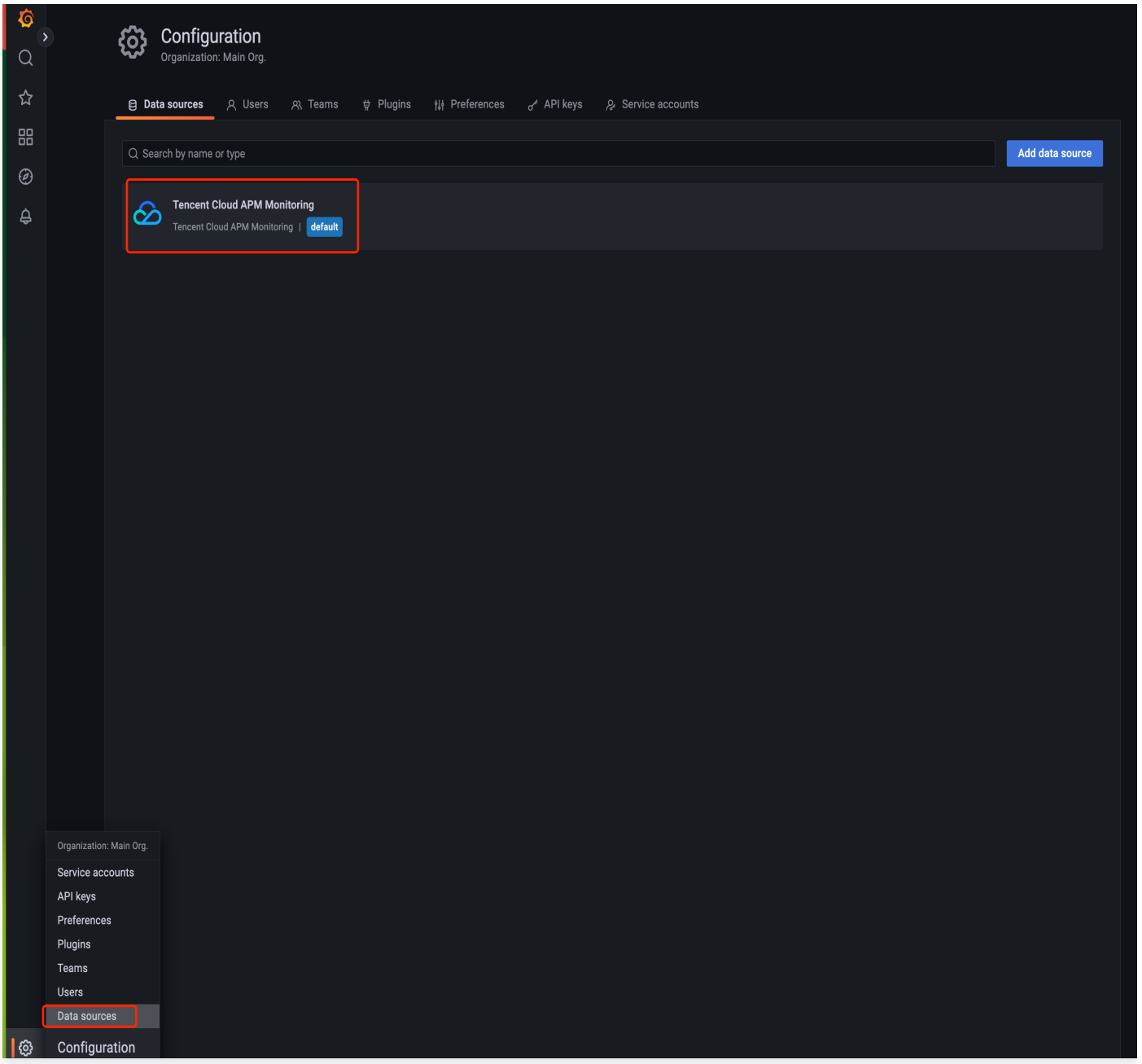
5. 点击设置下的 **Plugins**。



6. 搜索 **Tencent Cloud APM**，点击 **Enable** 使插件可用。

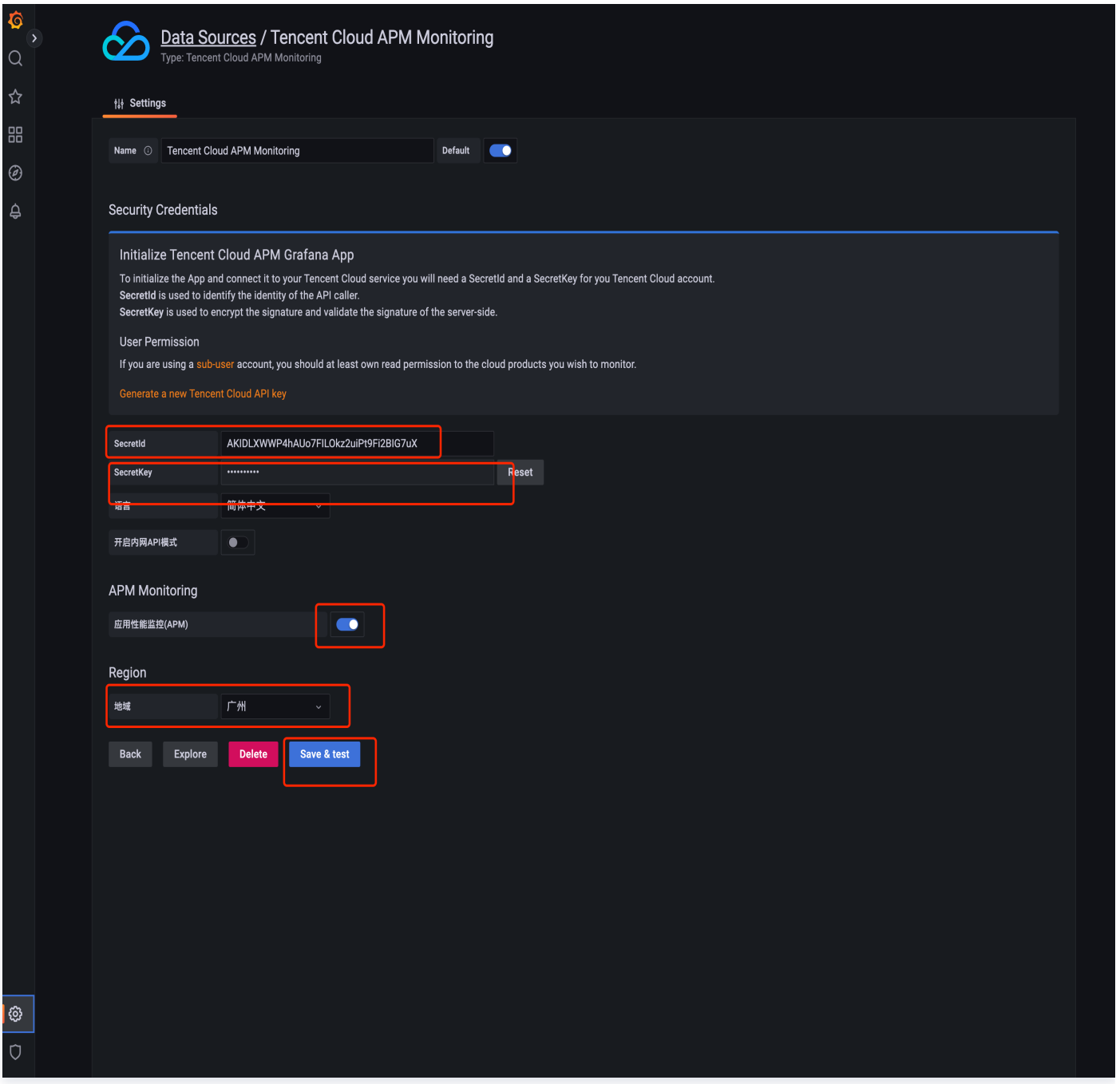


7. 点击 **Enable** 后，可在数据源中能看到 **Tencent Cloud APM Monitor** 插件已经可用。

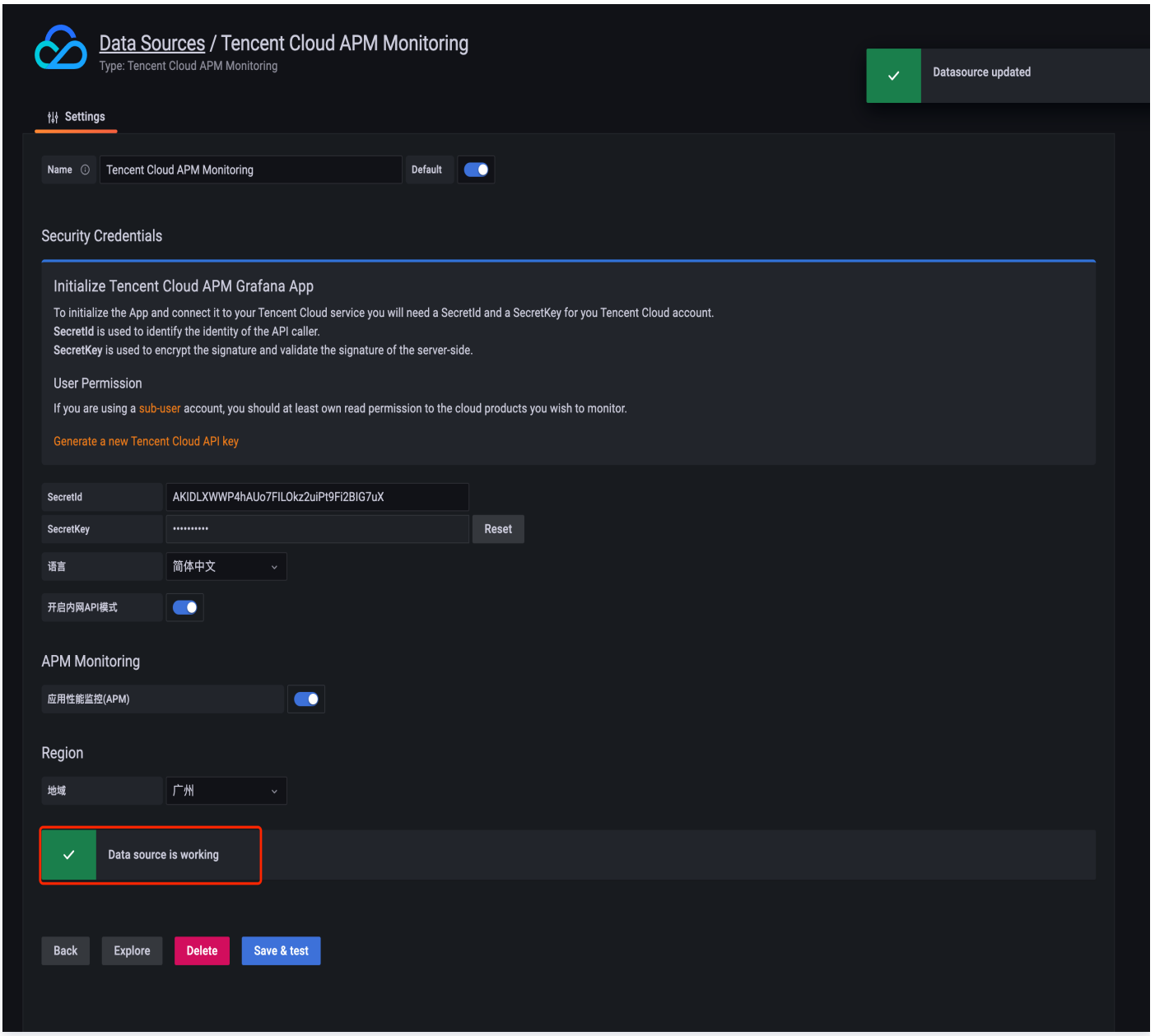


8. 通过 Grafana 设置的 Data Sources 配置具体的数据源，其中 **SecretId** 和 **SecretKey** 必填，勾选 **APM Monitoring** ，并且选取资源所在的地域，然后点击**Save & test**。

> **说明：**
> **SecretId**、**SecretKey** 为在前提条件中申请的 云 API SecretId和SecretKey 。

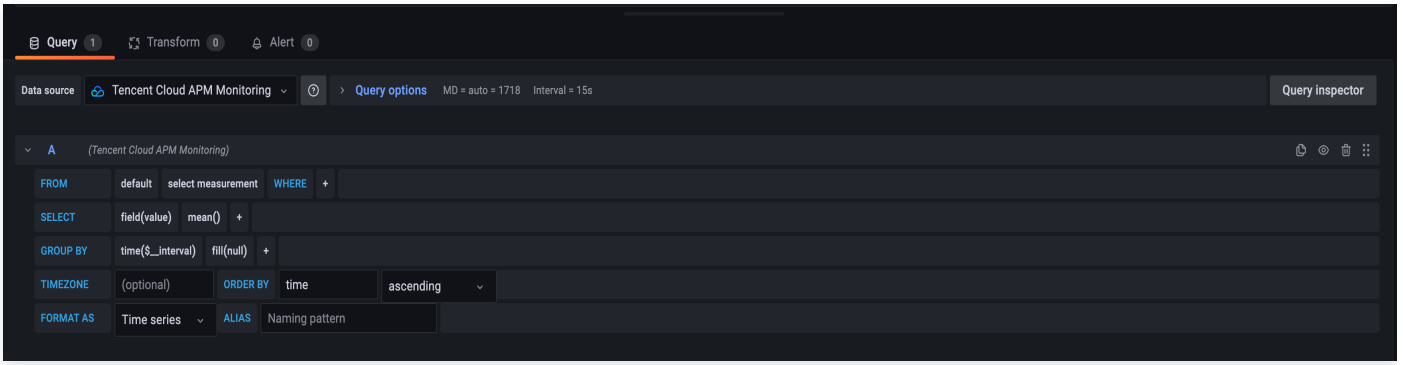


没问题将显示 **Data source is working**，接下来就可以正常使用了。



9. 打开 Grafana Home 的【预设】APM 面板，具有总请求量、慢请求、慢 SQL 、异常数量统计、平均响应时间、平均错误率、Top 错误率、Top 慢响应、错误类型列表、Top5 MySQL 慢 SQL 列表、平均响应时间、QPS 走势和耗时分位数等预设面板。预览图如下所示：

10. **自定义面板**：通过 APM 数据源自定义任何可选字段的配置 SQL。



自定义应用实例属性

最近更新时间：2024-08-28 11:32:11

在应用接入 APM 的时候，针对每个应用实例，用户可以自定义一系列实例属性，以实现基于实例属性的查询过滤以及数据聚合。在应用实例生成的可观测数据中，都会将附加上对应的实例属性信息。例如，在 APM 提供的应用详情功能中，用户可以基于实例属性对目标实例进行筛选，分析特定 K8s 集群、K8s 工作负载、应用镜像版本下的性能表现。在 APM 提供的容器监控、CVM监控等功能中，需要通过实例属性关联其他云产品，实现一体化可观测。

APM 支持的实例属性

属性名	Key	是否必填	可以自动注入的场景
实例名	host.name	是	• 通过腾讯云增强版 Java 探针接入 • 通过 Skywalking 方案接入 • TKE环境通过 tencent-opentelemetry-operator 一键接入
CVM 所在地域	cvm.region	否	通过腾讯云增强版 Java 探针接入
CVM 实例 ID	cvm.instance.id	否	通过腾讯云增强版 Java 探针接入
TKE 所在地域	k8s.region	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
TKE 集群 ID	k8s.cluster.id	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
K8s 节点 IP	k8s.node.ip	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
K8s 命名空间	k8s.namespace.name	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
K8s Deployment 名	k8s.deployment.name	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
K8s Pod 名	k8s.pod.name	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
K8s Pod IP	k8s.pod.ip	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
应用版本	service.version	否	TKE 环境通过 tencent-opentelemetry-operator 一键接入
自定义字段1	custom_key_1	否	-
自定义字段2	custom_key_2	否	-
自定义字段3	custom_key_3	否	-

手动设置实例属性

> **说明：**
> - 仅支持通过 OpenTelemetry 方案接入的应用。
> - 在可以自动注入实例属性的场景，请勿通过手动设置覆盖。

在应用接入的时候，实例属性需要连同 `token` 和 `service.name`，一起填入到 Resource Attributes 中。在 OpenTelemetry 标准中，Resource Attributes 以键值对的方式体现应用实例的基本信息。用户可以通过环境变量设置实例属性，也可以在接入代码中设置实例属性，具体的设置方式和接入应用时设置 `token` 和 `service.name` 的方式一致。

方式一：通过环境变量设置实例属性

将实例属性添加到环境变量 `OTEL_RESOURCE_ATTRIBUTES` 中，最终 `OTEL_RESOURCE_ATTRIBUTES` 会表现为类似 `key1=value1,key2=value2,key3=value3` 的形式。
参考如下代码设置 `k8s.region` 和 `k8s.cluster.id` 两个实例属性：

```
export OTEL_RESOURCE_ATTRIBUTES="service.name=myService,token=myToken"
export  OTEL_RESOURCE_ATTRIBUTES="$OTEL_RESOURCE_ATTRIBUTES,k8s.region=ap-guangzhou,k8s.cluster.id=cls-7i2n4axx"
```

对于部署在 Kubernetes 的应用（没有使用 tencent-opentelemetry-operator 一键接入），也可以在工作负载 Yaml 文件通过 `OTEL_RESOURCE_ATTRIBUTES` 环境变量设置实例属性，参考如下代码片段：

```
spec:
  containers:
  - env:
    - name: OTEL_RESOURCE_ATTRIBUTES
      value: "service.name=myService,token=myToken,k8s.region=ap-guangzhou,k8s.cluster.id=cls-7i2n4axx"
```

对于 TKE 环境通过 tencent-opentelemetry-operator 一键接入的应用，可以在工作负载 Yaml 文件设置没有被自动注入的实例属性，参考如下代码片段：

```
spec:
  containers:
  - env:
    - name: OTEL_RESOURCE_ATTRIBUTES
      value: "custom_key_1=value1,custom_key_2=value_2,cvm.region=ap-guangzhou,cvm.instance.id=ins-3flvdptq"
```

> **说明：**
> 对于 TKE 环境通过 tencent-opentelemetry-operator 一键接入的应用，除了可以自动注入的实例属性外，token 和 service.name 也已经自动注入，请勿在 Yaml 文件中通过 OTEL_RESOURCE_ATTRIBUTES 环境变量进行覆盖。

方式二：在接入代码中设置实例属性

如果通过代码接入应用，可以直接在接入代码中设置实例属性，以 Go 应用为例：

```
import (
    "context"
    "errors"
    "os"
    "go.opentelemetry.io/otel/sdk/resource"
)
    r, err := resource.New(ctx, []resource.Option{
      resource.WithAttributes(
          attribute.KeyValue{Key: "token", Value: attribute.StringValue("my-token")},
          attribute.KeyValue{Key: "service.name", Value: attribute.StringValue("my-
service")},
          attribute.KeyValue{Key: "host.name", Value:
attribute.StringValue("10.10.0.1")},
          attribute.KeyValue{Key: "k8s.region", Value: attribute.StringValue("ap-
guangzhou")},
          attribute.KeyValue{Key: "k8s.cluster.id", Value: attribute.StringValue("cls-
7i2n4ace")},
          ),
  }...)
```

如何动态获取 Pod 名称?

由于 Pod 名称、Pod IP、节点 IP 等信息需要在 Pod 创建后才能确定，在部署工作负载的时候，您可以通过 Kubernetes Downward API 进行动态获取，将这些信息写到环境变量中备用，方便在手动设置实例属性的时候进行引用。

```
spec:
  containers:
  - env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    - name: POD_IP
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: status.podIP
    - name: NODE_IP
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: status.hostIP
```

保存到环境变量后，可以直接引用，例如：

```
spec:
  containers:
  - env:
```

```
    - name: OTEL_RESOURCE_ATTRIBUTES
      value:
"service.name=myService,token=myToken,k8s.pod.ip=$(POD_IP),k8s.pod.name=$(POD_NAME)"
```

说明：
对于 TKE 环境通过 tencent-opentelemetry-operator 一键接入的应用，Pod 名称、Pod IP、节点 IP 等实例属性都已经自动注入，请勿在 Yaml 文件中通过 OTEL_RESOURCE_ATTRIBUTES 环境变量进行覆盖。

通过采样策略降低 APM 使用成本

最近更新时间：2025-04-02 18:53:22

概述

应用性能监控（APM）提供了分布式链路追踪能力，能够帮助用户自动构建每次请求的完整路径，实现一站式全链路问题分析，提高定位问题的效率。应用接入 APM 以后，会基于用户请求向 APM 系统上报链路信息，链路信息以 Trace 和 Span 的形式承载。当业务规模增长时，往 APM 上报的链路信息会随之增长，与此同时，使用 APM 的成本费用也会随之增加。

- Trace 代表一条完整的链路，描述一次用户请求多个分布式应用中经过的路径；
- Span 代表链路中的一个环节，可以是一次 RPC 调用、一次 HTTP 调用、一次消息发送，也可以是应用内部的一次本地函数调用。

在实际使用场景中，不是每一条 Trace 都值得被 APM 系统记录，因为分布式系统之间的相互调用会产生大量重复的链路信息。特别是在系统健康运行的时候，重复而冗余的链路信息对于用户分析性能问题并不会带来太大的价值。采样（Sampling）的引入，可以减少重复的链路信息，从而帮助用户将关注的重点放在更有价值的数据上，同时降低 APM 的使用成本。

常见的采样方案

采样的本质，是从大量数据中选择一部分数据进行观察和分析，以理解系统整体的行为或特性。体现在 APM 系统中，采样针对的是链路数据，这代表没有被选择到的链路数据可以不被采集，或者采集后直接被丢弃。具体哪一些链路数据被选择，在不同的采样方案，会存在比较大的差异。根据 APM 系统的特点，采样策略的引入，需要考虑到如下几个重要的方面：

- **链路完整性**。一条完整的链路（Trace）包含多个跟踪单元（Span），如果采样导致一条链路中的部分 Span 被丢弃，链路的完整性就会被打破，整条链路数据都会失去价值。
- **指标正确性**。各种统计指标不要因为采样策略的引入而出现偏差，包括吞吐量、响应时间、错误率、应用健康状态等等。
- **保留高价值数据**。在真实的分布式系统中，偶尔会出现一些关注度非常高的链路数据，需要采样策略能够选择性的保留这些数据。其中最典型的例子就是错误调用和慢调用，当一条链路中存在错慢调用的时候，不能被采样策略丢弃。

结合这几项普遍的需求，基于数据的选择和采样时机，APM 领域诞生了2种典型的采样方案。

头部采样（Head Based Sampling）

基本概念

头部采样是指在请求进入分布式系统的入口处进行采样决策，随着请求在不同应用之间的流转，采样决策会一直保持传播，从而保证每一个环节都遵循相同的行为。在系统入口处，一旦决定对某个请求进行采样，整条链路都可以完整的保留下来。

实现原理

头部采样的实现比较简单，使用通用的链路传播协议，例如 OpenTelemetry 默认使用的 W3C 链路上下文标准，都针对头采样方案提供了支持。APM 系统只需要基于链路传播协议，在提供给应用接入的探针或 SDK 中进行简单的处理，就能实现头部采样。但头部采样可能会导致某些重要事件或异常的遗漏，因为采样决策是在请求开始时就已经确定的，无法根据请求在处理过程中的实际情况进行调整。

例如，当用户请求进入分布式系统后，基于预设的采样规则，做出的决策是：此条链路不采样。但在接下来的环节，这条链路上出现了一个数据库慢调用，SQL 执行的时长超过了3秒。这本应用是一个值得重点分析的重要事件，但因为采样策略的引入，导致整条链路都被丢弃了。

此外，在 APM 系统中，链路数据和统计指标之间是存在相互关系的，例如统计某一个接口的响应耗时99分位数，往往依赖于 APM 服务端基于收到的链路进行计算。头部采样会导致指标数据跟真实情况发生非常大的偏离。

尾部采样（Tail Based Sampling）

尾部采样是指在请求完成后再进行采样决策。APM 系统会先临时收集所有链路数据，然后在请求完成后根据实际情况（如请求的响应时间、错误状态等）决定保留哪一些链路。
尾部采样的优点是能够更好地捕获重要事件和异常，也可以确保统计指标的准确性。但尾部采样需要 APM 系统在服务端引入额外的存储和计算资源，来临时保存所有请求的链路数据，实现比较复杂。在常见的开源 APM 项目中，对于尾部采样的实现，都没有提供完整的实现方式。

采样方案总结

类别	头部采样	尾部采样
决策时间点	在请求入口处决策	在请求完成之后再决策
决策者	APM 的客户端，也就是应用侧	APM 服务端
实现方式	未采样的数据不上报到 APM 服务端	全量上报，APM服务端再基于链路的特点决定是否保存
链路完整性	满足	满足
保留错慢请求	不满足	满足
指标正确性	在大多数情况下不满足，特别是 APM 服务端通过链路数据进行指标统计的场景。	满足
实现复杂度	低	高

总的来说，头部采样和尾部采样各有优缺点，需要根据实际的业务需求和系统状况进行选择。但站在 APM 使用者的角度，建议使用尾部采样方案，对于分析应用性能和排查问题能够提供更好的帮助。

在腾讯云 APM 中开启采样策略

基于尾部采样方案在分析应用性能和排查问题方面的优势，腾讯云应用性能监控（APM）实现了完整了尾部采样方案，能够在降低数据存储量的同时，确保错慢链路完整保存，并确保所有指标数据的正确。合理的使用采样策略，不仅可以降低 APM 的使用成本，而且对于 APM 的使用体验不会带来明显的影响。此外，APM 提供的采样策略适用于所有语言编写的应用，也适用于所有接入方案。

说明：

- 在按量付费模式下，应用性能监控（APM）的计费项包括上报费用和链路存储费用，详情请参考 APM 按量付费介绍 。在 APM 控制台开启采样策略以后，可以降低链路存储费用，最高可以降低90%。但因为尾部采样方案需要全量上报数据，以确保指标的正确性以及错慢链路完整保存，开启采样策略并不能降低上报费用。
- 套餐包（预付费）模式下，采样率将被固定在10%，详情请参考 APM 套餐包介绍 。
- 目前采样策略需要通过开白名单的形式提供，请您通过 提交工单 进行申请。申请完成后，在 应用性能监控 > 系统配置 中将展示**采样配置**。

修改全局采样配置

1. 登录 腾讯云可观测平台 控制台。
2. 在左侧菜单栏中选择**应用性能监控** > **系统配置**，进入**采样配置**页面。
3. 在**全局采样配置**模块，单击**编辑**。
   - 修改**采样率**，有效的采样百分比在10%到100%之间。
   - 修改**保存错误链路**开关，以及**慢调用保存阈值**。

配置全采样接口

当全局采样率小于100%的时候，可以自定义需要全采样的接口，如果链路经过了需要全采样的接口，整条链路都会被完整保存。

1. 单击**新增全采样接口**。
2. 在对话框中输入策略名，并指定需要进行全采样的接口。

如果在策略中指定了具体的应用，可以不填写接口匹配规则，此时代表经过此应用的所有链路都会被完整保存。您可以基于精确匹配、前缀匹配以及后缀匹配来指定接口。在实际使用场景中，可以将关注度比较高的重要接口定义为全采样接口，确保不错过任何一条针对重要接口的调用。

采样策略配置原则

- 修改采样策略相关配置，可以即时生效，无需重启应用。
- 请根据实际业务场景修改全局采样率。当全局采样率低于100%的时候，APM 不会实时保存没有被命中的链路。
- 推荐开启**保存错误链路**开关，并将**慢调用保存阈值**设置在500ms到2000ms之间。
- 对于没有被命中的链路，APM 会基于**保存错误链路**开关、**慢调用保存阈值**，以及**全采样接口**等配置项，综合判断每条链路是否需要持久化保存。错慢链路以及经过了全采样接口的链路，在经过10分钟左右的延迟后，最终将被完整保存。

Prometheus 数据集成以及 Dashboard 关联展示

最近更新时间：2024-12-16 15:49:33

Prometheus 是一个功能强大、灵活且扩展性强的开源可观测平台，提供了多维数据模型、丰富的采集能力，以及强大的查询语言。作为 CNCF（Cloud Native Computing Foundation）旗下最重要的开源项目之一，Prometheus 在云原生时代被广泛使用，构建了繁荣的开源生态。特别是在指标（Metric）监控领域，Prometheus 已经成为了事实的标准。
与此同时，OpenTelemetry 也成为了可观测数据在检测、生成、收集和导出方面的最重要的技术标准，对于可观测领域的三大支柱数据——链路（Trace）、指标（Metric）、日志（Log）都提供了支持。
腾讯云应用性能监控（APM）实现与 Prometheus 的数据集成：对于通过 OpenTelemetry 方案接入 APM 的应用，可以使用 OpenTelemetry API 上报自定义指标。APM 服务端负责将自定义指标同步到腾讯云 Prometheus 监控服务，帮助用户基于 Prometheus 生态挖掘数据价值。其中最典型的使用场景就是通过 Grafana 服务对接 Prometheus 数据源，获得强大的数据展示能力。此外，用户还可以通过腾讯云可观测平台提供的 Dashboard 服务对接 Prometheus 数据源，并在 APM 控制台基于应用视角查看 Dashboard 中的自定义大盘。

如何将 OpenTelemetry 指标输出到 Prometheus?

由于 OpenTelemetry 和 Prometheus 属于不同的技术体系，OpenTelemetry API 上报自定义指标，不能直接输出到 Prometheus 中，需要进行处理和转换，常见的方式有如下几种：

1. 应用引入 openTelemetry-exporter-prometheus 库，将 OpenTelemetry 指标直接以 Prometheus Exporter 形式进行暴露，再通过 Prometheus 进行拉取。
2. 应用将 OpenTelemetry 指标上报到 OpenTelemetry Collector，由 Collector 将 OpenTelemetry 指标以 Prometheus Exporter 形式进行暴露，再通过 Prometheus 进行拉取。
3. 应用将 OpenTelemetry 指标上报到 OpenTelemetry Collector，由 Collector 将 OpenTelemetry 指标转换为 Prometheus 指标，再通过 Remote Write 的方式写入到 Prometheus。

这几种方式的实用低都比较低：方式1需要引入特定的类库，在很多编程语言中都是缺失的，而且服务发现的配置工作也比较复杂。方式2和方式3需要自行搭建 OpenTelemetry Collector，部署和维护工作量比较大。
腾讯云应用性能监控（APM）引入了更便捷高效的集成方案，不需要自行搭建 OpenTelemetry Collector，也不需要复杂的配置项。在进行简单的关联后，应用直接通过 OpenTelemetry API 上报指标，即可输出到腾讯云 Prometheus 监控服务中。
Prometheus 监控服务（TencentCloud Managed Service for Prometheus，TMP）是基于开源 Prometheus 构建的高可用、全托管的服务，与腾讯云容器服务（TKE）高度集成，兼容开源生态丰富多样的应用组件，结合腾讯云可观测平台的告警功能和 Prometheus Alertmanager 能力，为用户提供免搭建的高效运维能力，减少开发及运维成本。关于腾讯云 Prometheus 监控服务的更多详情，请参见 Prometheus 监控概述 。

关联 Prometheus 实例

在应用上报自定义指标之前，需要先建立 APM 业务系统和 Prometheus 实例之间的关联，并配置需要从 APM 服务端同步到 Prometheus 实例的指标。

1. 登录 APM 控制台，请前往**系统配置 > Prometheus 集成**。
2. 在**关联配置**中，开启 Prometheus 关联，并选择当前地域的任何一个 Prometheus 实例。每个 APM 业务系统只能关联同地域的最多一个 Prometheus 实例。在当前账号第一次做关联操作的时候，需要授予 APM 访问 Prometheus 资源的权限，根据控制台的提示完成服务授权即可，系统会在访问管理（CAM）自动创建名为 `APM_QCSLinkedRoleInPromInstance` 的角色。

3. 单击**新增指标同步规则**，指定需要同步到 Prometheus 实例的指标。

   对于每一条同步规则，您可以基于**精确匹配**、**前缀匹配**以及**后缀匹配**这三种匹配方式来匹配指标名，也可以指定该规则的生效范围（可以是该业务系统的全部应用，或某一个具体的应用）。当 APM 服务端收到应用上报的自定义指标后，满足同步规则的指标将被写入到被关联的 Prometheus 实例中，不满足同步规则的指标将被丢弃。
   除了上报时填入的自定义标签外，APM 还额外增加了 `apm_instance` 和 `apm_service_name` 两个标签，分别代表 APM 业务系统 ID 以及应用名，这样每一条 Prometheus 指标样本都能关联到唯一的 APM 应用。

> **说明：**
> - 写入到 Prometheus 实例的指标，将根据腾讯云 Prometheus 监控服务的计费规则产生相关的费用，详情请参见 Prometheus 计费概述 。
> - 除了通过 OpenTelemetry API 上报的自定义指标，也可以将 APM 产生的固有指标同步到 Prometheus 中，关于 APM 固有指标的详情，请参见 APM 指标协议标准 。

指标转换原理

OpenTelemetry 旨在构建与语言无关，支持多种编程框架以及多种可观测平台的统一标准，所以在通过 OpenTelemetry 方案上报指标数据的应用场景中，Prometheus 并不是唯一的指标存储平台选择。基于这个原因，OpenTelemetry 的指标模型和 Prometheus 的指标模型不完全相同，在编写上报数据的代码之前，需要先了解从 OpenTelemetry 指标到 Prometheus 指标的转换逻辑。

OpenTelemetry 协议的数据模型

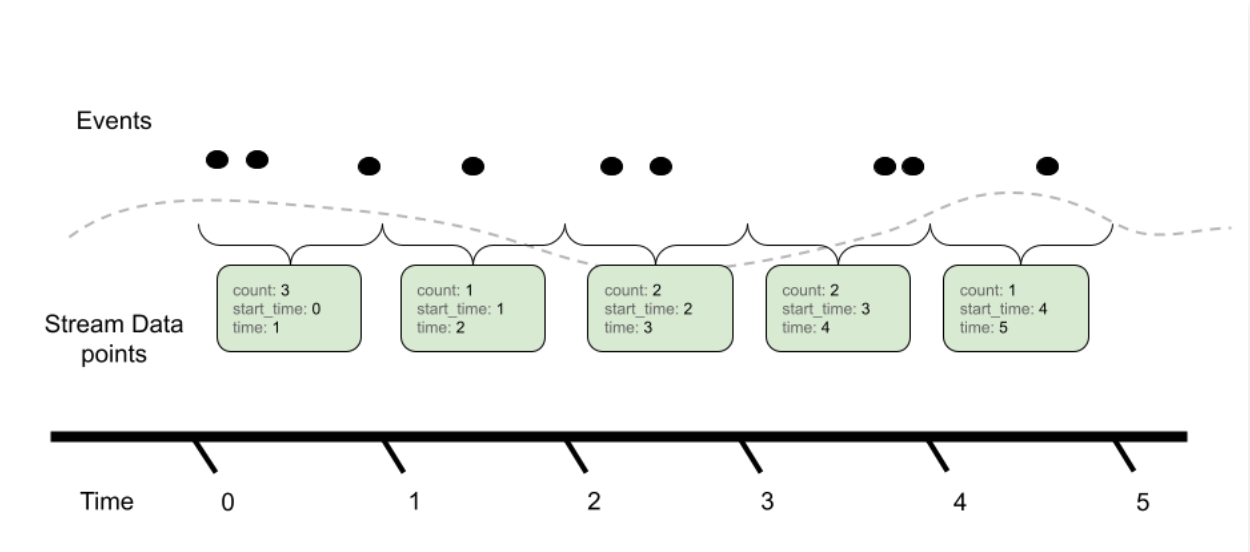
OpenTelemetry 的指标支持如下指标类型：

- **Gauge**：表示瞬时值，一般情况下，Gauge 类型的数据点是不具备累加属性的，例如当前的温度、汽车的时速等。
- **Sum**：表示一定时间间隔内所有测量值的总和。如果数据点是具备累加属性的，定义成 Sum 类型是更好的选择，例如 HTTP 请求数，网络流量等。如果测量值单调递增的，Sum 可以被标识为单调的（monotonic）。对于单调的 Sum，后一个测量值永远不会小于前一个测量值，HTTP 请求数就符合这样的特征。
- **Histogram**：表示通过聚合一定时间间隔内所有测量记录，以直方图的形式体现的数据类型。Histogram 包含通过聚合得到的 sum、count 字段，分别代表记录数以及测量值的总和。除此之外，还有可能包括表示最大值与最小值的 max、min 字段，以及一系列的数据桶（bucket）。每个数据桶会有明确的边界范围，以及落入到这个边界范围的记录数。Histogram 通过引入了聚合机制，显著降低了指标数据量，同时也能提升数据的可读性。
- **ExponentialHistogram**：ExponentialHistogram 是 Histogram 的替代表达形式，和 Histogram 的唯一区别在于 ExponentialHistogram 使用指数作为桶的边界，适合以较小的相对误差传达高动态范围数据。

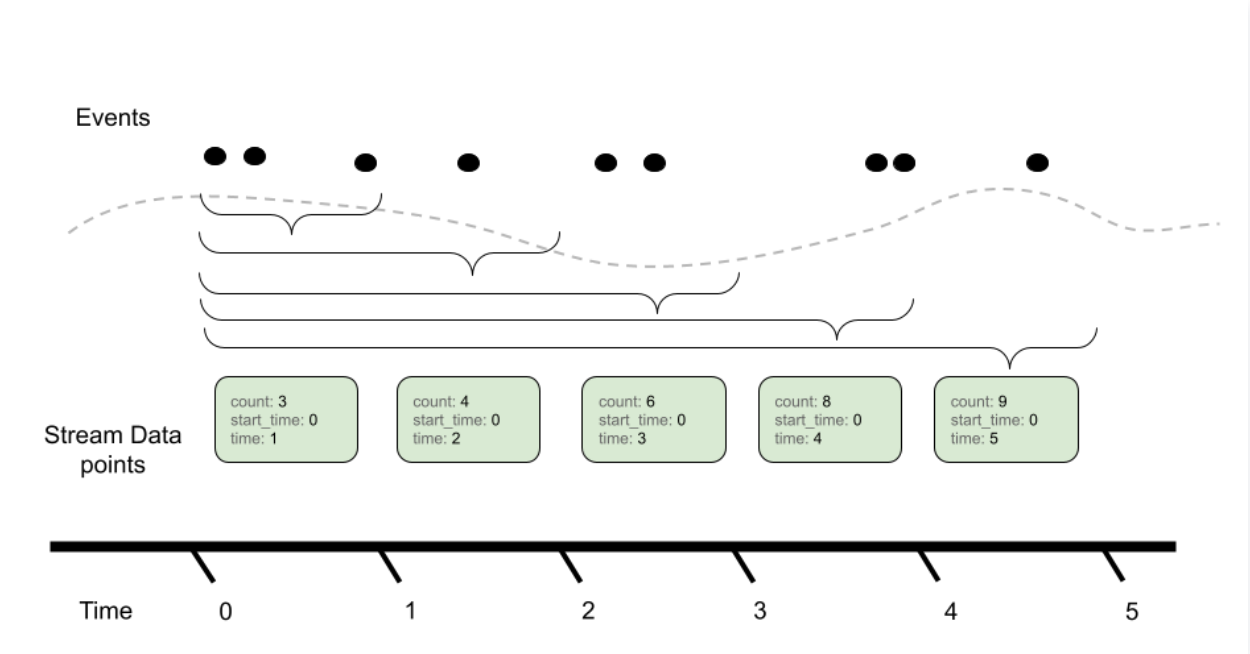
- **Summary**：Summary 通过分位数的形式表达数据，在最新的 OpenTelemetry 协议标准中，已经不推荐使用 Summary，所以尽可能不要使用这种数据点类型。

在 OpenTelemetry 指标协议中，对于 Sum、Histogram 和 ExponentialHistogram 类型，还支持两种不同的**聚合时间性**（Aggregation Temporality），用来区分数据是如何累加的，它们分别是：

- **增量（Delta）**：指标流的时间窗口不会重叠，相当于在指标流中随着时间的推移记录每个间隔时间段的数据增量。



- **累积（Cumulative）**：相当于在指标流中记录从“开始”以来的数据累积总和，“开始”通常意味着进程/应用程序的启动。



关于 OpenTelemetry 协议数据模型的更多详细介绍，请参见 OpenTelemetry 官方文档 。

Prometheus 指标模型

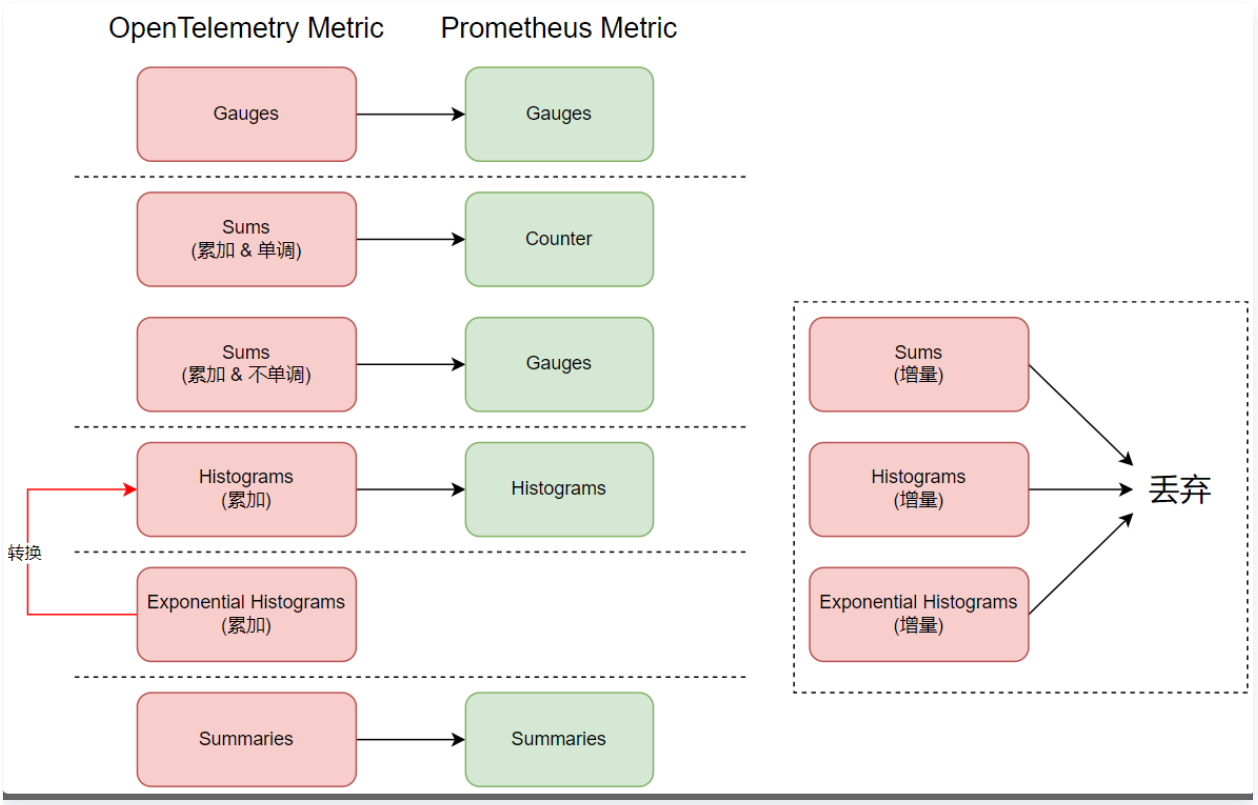
Prometheus 包含如下几种指标模型，他们在定义上和 OpenTelemetry 的数据模型很类似，在本文中不再赘述，可以参考 Prometheus 官方文档获得详细信息：

- Counter
- Gauge
- Histogram

- Summary

模型映射

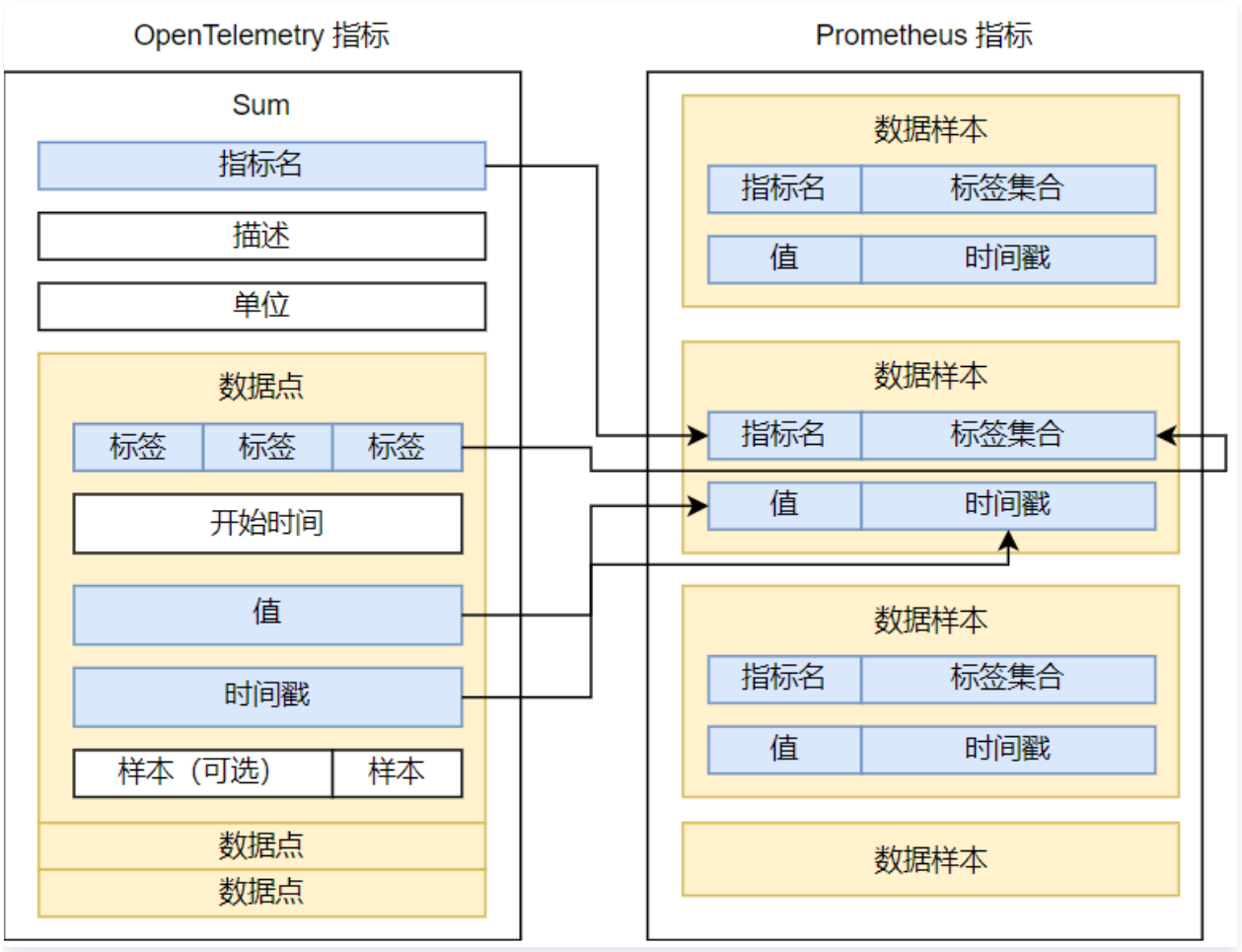
对于 OpenTelemetry 指标，将按照如下映射关系转换为 Prometheus 指标。



需要注意的是，Prometheus 的数据模型中，不能兼容增量聚合时间性（Delta Temporality），所以 APM 将**直接丢弃使用 Delta 聚合时间性的 OpenTelemetry 指标**。在 OpenTelemetry 提供的探针以及 SDK 中，存在名为 `OTEL_EXPORTER_OTLP_METRICS_TEMPORALITY_PREFERENCE` 的环境变量，或者名为 `otel.exporter.otlp.metrics.temporality.preference` 的系统参数，用于指定聚合时间性（Delta Temporality），其默认值为 `cumulative` 或 `CUMULATIVE` 。在保持默认值的情况下，不会发生因为兼容性导致的丢弃问题。
对于不单调递增的 Sum，由于 Prometheus 的 Counter 指标必须保持单调递，因此不能直接保存为 Prometheus 的 Counter 指标。APM 会将其将转换为 Prometheus 的 Gauge 指标。
对于 ExponentialHistogram，APM 会将其转换为 Prometheus 的 Histogram 指标，当然，这种转换会丢失一些精度。

Sum 的指标转换

Sum 类型的 OpenTelemetry 指标在转换为 Prometheus 指标（单调递增的 Sum 将转换为 Counter，非单调递增的 Sum 将转换为 Gauge）的时候，会遵循如下的转换原则：



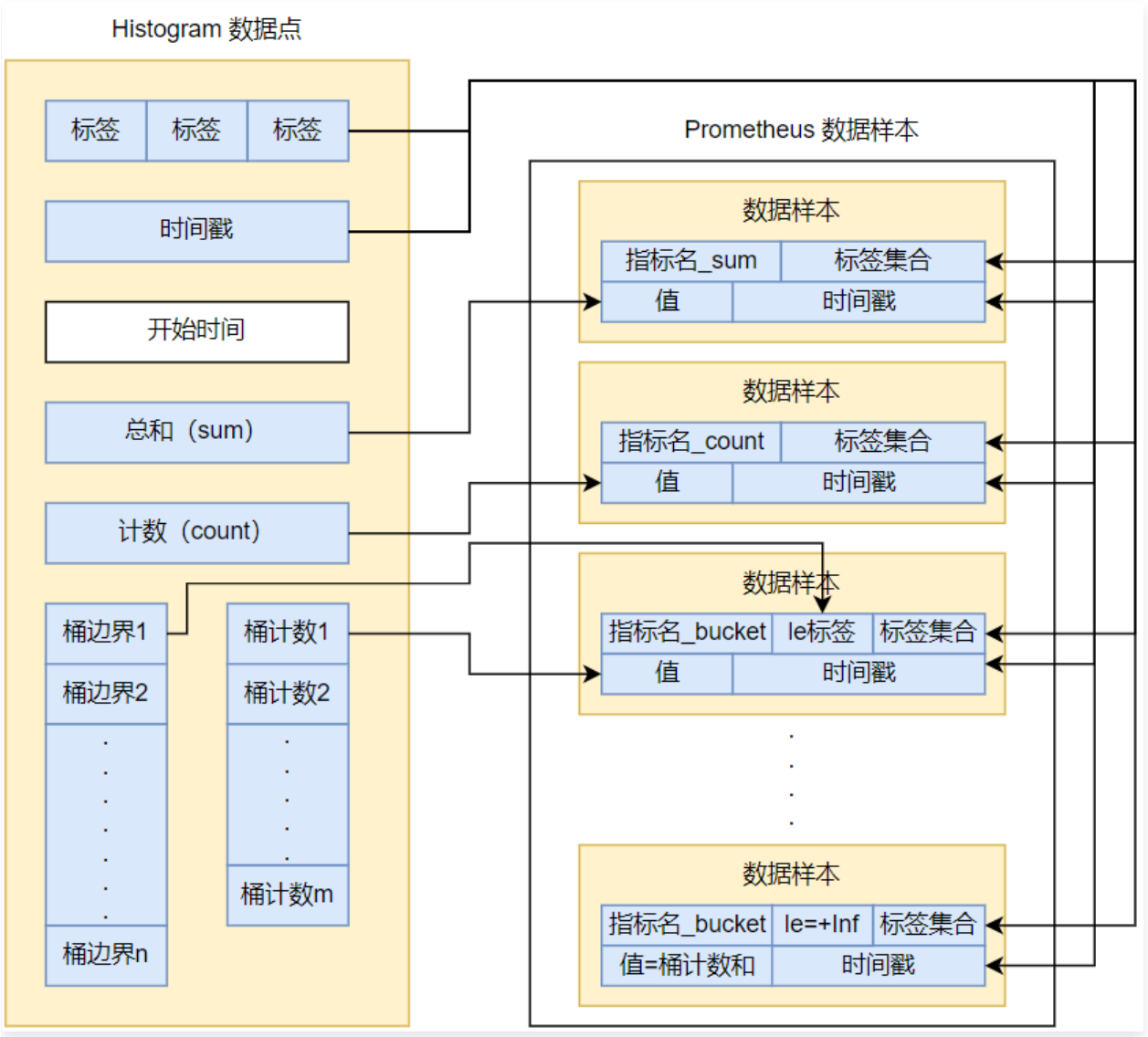
这是比较容易理解的，OpenTelemetry 指标中的指标名、标签对、时间戳等字段会被原样输出到 Prometheus 的数据样本中，每个 OpenTelemetry 数据点对应一个 Prometheus 数据样本。需要值得注意的是，由于两种数据模型并不是完全一致，在转换为 Prometheus 指标的过程中，OpenTelemetry 指标中的部分信息将被忽略，其中包括开始时间（StartTimeUnixNano）、单位（Unit）、范例（Exemplar）等。

Gauge 类型的指标转换

Gauge 类型的 OpenTelemetry 指标在数据结构上与 Sum 类型类似，转换方式与 Sum 类型没有区别，本文不再赘述。

Histogram 类型的指标转换

Histogram 类型的 OpenTelemetry 指标在转换为 Prometheus 指标的时候，会遵循如下的转换原则：



每个 Histogram 类型的 OpenTelemetry 数据点，会被转化为多个 Prometheus 数据样本（Sample），具体的个数取决于 Histogram 类型的 OpenTelemetry 数据点中带有多少个桶（Bucket）。和 Sum 类型一样，标签对和时间戳会被原样输出到 Prometheus 的数据样本中，但在指标名和指标值的处理上，会经过一系列的转换。

首先，APM 会生成一个名为 `指标名_sum` 的 Prometheus 数据样本，取值为总和（sum）字段，代表所有原始记录的累加值总和。APM 还会生成一个名为 `指标名_count` 的 Prometheus 数据样本，取值为计数（count）字段，代表原始记录的总计数。

此外，每个来自 OpenTelemetry 的桶都会被转换成一个名为 `指标名_bucket` 的 Prometheus 数据样本，同时在此数据样本中添加一个标记为 `le` 的标签，表示该桶的边界。这些数据样本从低到高排列，每个点的值都是该数据样本 le 标签边界前所有桶的计数之和，从而转换成 Prometheus 中累加的 Histogram。之所以要这样转换，是因为在 Prometheus 的 Histogram 数据模型中，`le` 标签代表有多少条原始记录小于或等于一个特定的值，实际上就一定会包含前一个样本的计数。

最后，APM 会额外生成一个名为 `指标名_bucket` 的 Prometheus 数据样本，并添加值为 `+Lnf` 的 `le` 标签，其取值实际上就是计数（count）字段，代表原始记录的总计数。

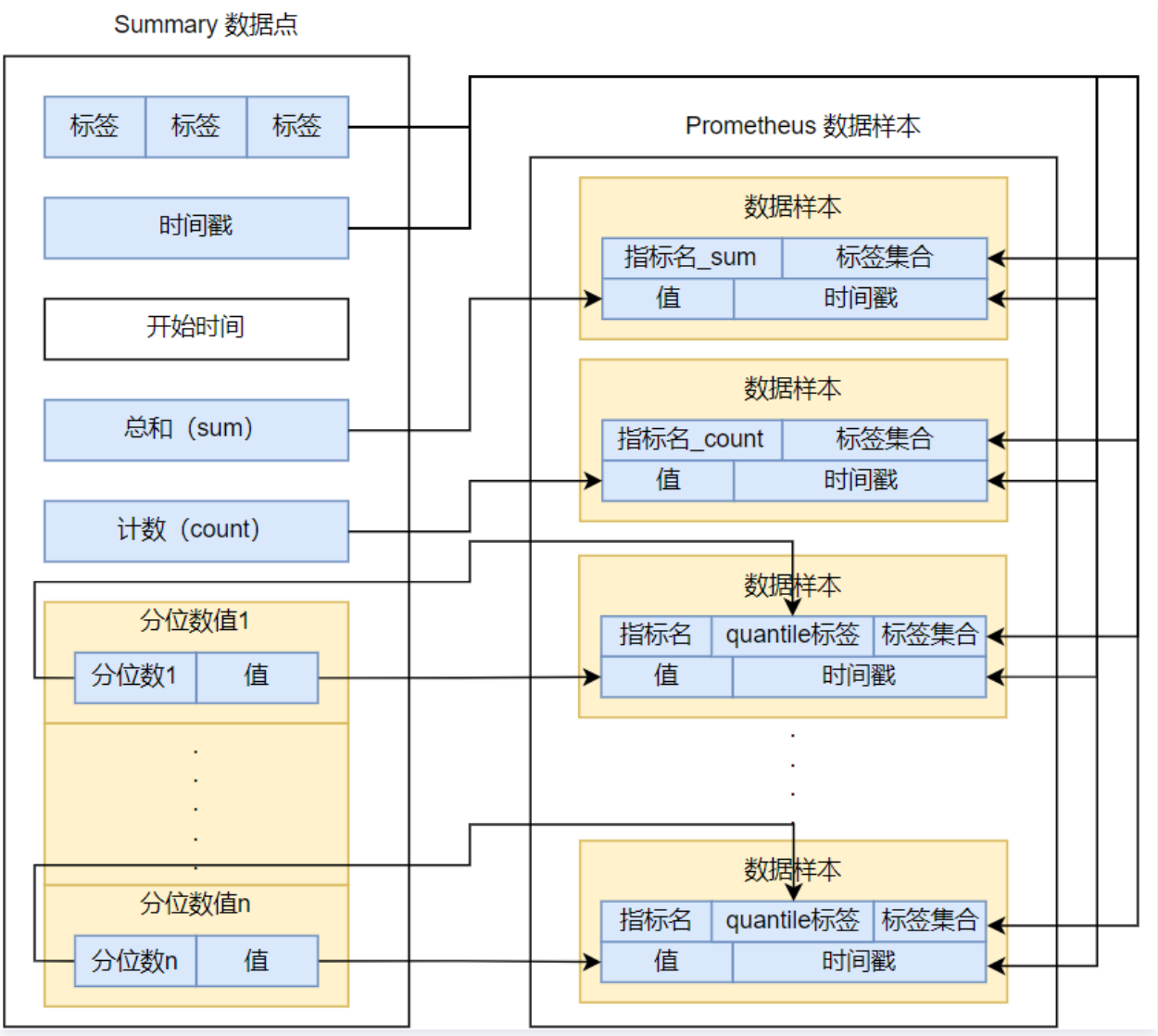
因此，如果1个 Histogram 类型的 OpenTelemetry 数据点中带有 N 个桶（Bucket），将其转换为 Prometheus 指标后，会生成 N+3 条数据样本。

ExponentialHistogram 类型的指标转换

Prometheus 目前还没有支持 ExponentialHistogram，为了支持 ExponentialHistogram 类型的数据上报，APM 会先将 ExponentialHistogram 的桶边界转换成保留两位小数的 Histogram 桶边界，再进一步按照 Histogram 类型的指标转换规则进行处理。这样会丢失一些精度，所以不建议使用 ExponentialHistogram 类型上报数据。

Summary 类型的指标转换

Summary 类型的 OpenTelemetry 指标在转换为 Prometheus 指标的时候，会遵循如下的转换原则：



Summary 类型的转换规则和 Histogram 类似，如果1个 Summary 类型的 OpenTelemetry 数据点中带有 N 个分位数，将其转换为 Prometheus 指标后，会生成 N+2 条数据样本。由于在最近的 OpenTelemetry 协议标准中，已经不推荐使用 Summary，所以尽可能不要使用这种类型上报数据。

通过 OpenTelemetry API 上报指标

了解从 OpenTelemetry 指标到 Prometheus 指标的转换逻辑，可以帮助我们更好的理解两种数据模型，更合理的设计指标结构。但在代码编写实战环节，OpenTelemetry API 通过一系列的封装，屏蔽了 OpenTelemetry 指标模型的底层细节，让开发者可以轻松上手完成指标上报。

同步 API 与异步 API

OpenTelemetry 提供了同步和异步2种 API 编码模型，开发者可以根据实际情况使用其中的一种：

- **同步方式（Synchronous Instruments）**：这是一种比较直观的编码方式，当有新的测量数据时，直接通过代码调用 API 输入数据即可。测量系统的 QPS 是同步方式的典型使用场景，每收到一条请求的时候，都主动调用 API 将请求数加1。
- **异步方式（Asynchronous Instruments）**：异步方式需要先注册一个回调方法/函数，用于读取测量数据。在系统有收集数据需要的时候（通常情况下取决于客户端 SDK 往 APM 服务端上报数据的频率），会运行回调方法获取测量数据。测量 CPU 温度

是异步方式的典型使用场景，开发者并不需要考虑何时调用 API 输入 CPU 温度数据，而是提供一个读取 CPU 温度的方法/函数，作为回调方法/函数提供给异步 API。

重要对象

在编写代码的时候，需要使用到如下几个对象：

- **MeterProvider**：MeterProvider 是使用 OpenTelemetry 指标 API 的唯一入口。通常情况下，在一个应用中只需要初始化一次。
- **Meter**：Meter 由 MeterProvider 创建而成，用来创建 Instrument 对象。
- **Instrument**：Instrument 对象用来输入测量数据。针对不同的指标类型以及不同编码模型（同步/异步），OpenTelemetry API 提供了一系列 Instrument 实现，包括 Counter、Asynchronous Counter、Histogram、Gauge、Asynchronous Gauge、UpDownCounter 等。这些对象的使用都比较简单，通过查阅相关的 OpenTelemetry API 文档，就能够轻松掌握。

代码编写示例

我们以 Java 项目为例，演示如何通过 OpenTelemetry API 上报接收到的 HTTP 请求数量。

1. 构建一个 Spring Boot 项目，并提供相关的 HTTP 服务接口，此步骤请参考 Spring Boot 官方文档，本文不再赘述。也可以将 Spring Boot 替换为任意一种 Java 系的 HTTP Server。
2. 在项目中引入必要的依赖。参考如下 Maven 配置，引入 opentelemetry-api 即可。

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-api</artifactId>
    <version>1.35.0</version>
</dependency>
```

3. 创建 Instrument 对象。

   在这个示例中，需要上报 HTTP 请求数量，很明显我们需要用到 Sum 类型的指标。每当应用接收到一次 HTTP 请求后，需要将请求数 +1，所以使用同步 API 是比较合理的。通过查阅 OpenTelemetry API 文档，我们可以确认需要使用到的 Instrument 对象是 Counter。由于请求量通过整数来表达，在 Counter 的两种实现中，需要用到的是 LongCounter。通过如下代码，我们可以非常简单的创建一个用于上报 HTTP 请求数量的 LongCounter 对象。这个对象仅需要创建一次即可，其对应的指标名为 `http_request_total`，因为我们将其作为 RestController 的成员变量。

```
@RestController
@RequestMapping("/metric")
public class MetricController {
    // 把LongCounter作为成员变量
    private LongCounter httpRequestCounter;
    @PostConstruct
    public void init() {
        String scope = this.getClass().getName();
        // 获取OpenTelemetry对象
        OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
        // 获取Meter对象。关于Scope的定义，请参考OpenTelemetry官方文档，一般情况下，可以使用
class全名
        Meter meter = openTelemetry.getMeter(scope);
        // 创建LongCounter对象。在项目中创建1次即可
```

```
        httpRequestCounter =
meter.counterBuilder("http_request_total").setDescription("Counts HTTP
request").build();
    }
}
```

4. 计数。

   参考如下代码片段，每次收到 HTTP 请求的时候，将请求数 +1 即可。在本示例中，还加入了 Key 为 `method` 的属性标签，用于标识这次请求来自哪一个方法。

```
@RestController
@RequestMapping("/metric")
public class MetricController {
    // 定义一个Key为method的属性标签
    private static final AttributeKey<String> METHOD_KEY = stringKey("method");
    /*
    上个步骤中的初始化逻辑
    */
    @RequestMapping("/order")
    public String order() {
        // order请求数+1
        httpRequestCounter.add(1, Attributes.of(METHOD_KEY, "order"));
        return "order + 1";
    }
    @RequestMapping("/pay")
    public String pay() {
        // pay请求数+1
        httpRequestCounter.add(1, Attributes.of(METHOD_KEY, "pay"));
        return "pay + 1";
    }
}
```

通过简单几个步骤，就完成了代码的编写。需要注意的是，并不是每次调用 `LongCounter.add()` 方法就会生成一条新的指标，OpenTelemetry API 会对数据进行聚合后再进行上报，所以写入到 Prometheus 的真实的指标数量取决于如下两个方面：

1. 标签的基数。在本示例中，只有一个 Key 为 `method` 的标签，那么方法的数量就决定了每次上报的指标数量。在真实场景中，可能不止一个标签，这些标签的基数共同决定了每次上报的指标数量。
2. 上报频率。可以通过 `otel.metric.export.interval` 系统参数设置上报频率，默认为60秒，一般情况下保持默认即可。更多详情请参见 Java SDK 配置。

对于其他语言编写的应用，上报指标的方式和 Java 基本是一致的，具体请参见 OpenTelemetry 社区的 API/SDK 文档。

> **说明：**
> 如果您使用腾讯云增强版 OpenTelemetry Java 探针，请确保探针版本不低于2.3-20241031，否则会导致上报失败。

通过 Grafana 查阅指标

上报完成后，前往关联的 Prometheus 实例，通过其关联的 Grafana 服务查询 `http_request_total` 指标，就能查阅到上报的数据。接下来，可以基于 PromQL 语句实现复杂的查询，并通过 Grafana 自定丰富的图表。

Table

Time	__name__	apm_instance	apm_service_name	method	tcloud_region_id	tcloud_region_name	Value
2024-10-23 19:57:24.435	http_request_total	apm-ewyzCXlxj	java-web	pay	1	ap-guangzhou	7
2024-10-23 19:57:24.435	http_request_total	apm-ewyzCXlxj	java-web	order	1	ap-guangzhou	6

如果这个时候没有查到 `http_request_total` 指标，请确保在关联 Prometheus 实例的时候，针对此指标名配置了同步规则。对于不能匹配同步规则的指标，APM 服务端会进行丢弃处理。
由于 APM 在进行指标转换的时候，额外增加了 `apm_instance` 和 `apm_service_name` 两个标签，分别代表 APM 业务系统 ID 以及应用名，这样就可以基于这两个标签创建业务系统和应用过滤条件。

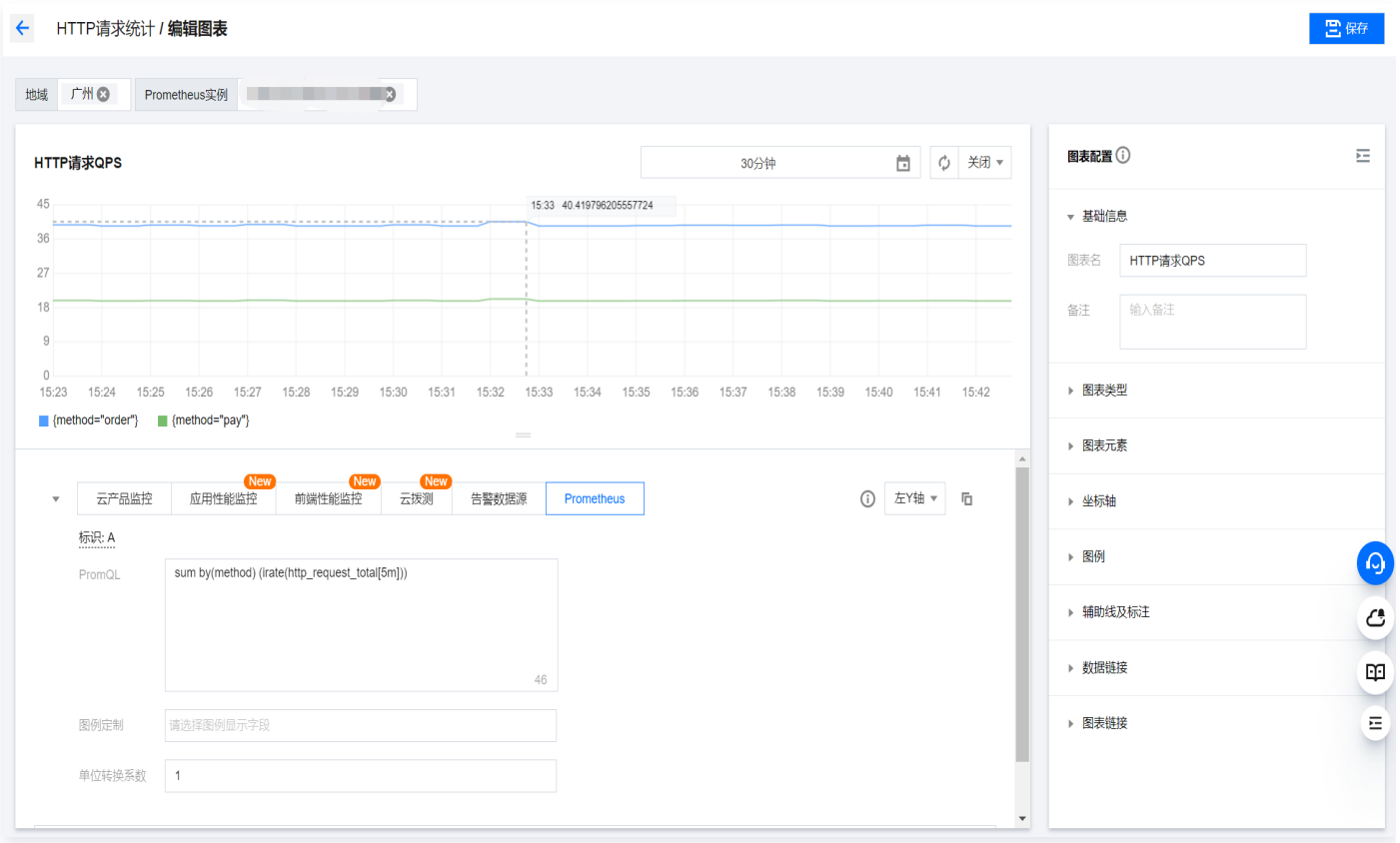
腾讯云可观测平台 Dashboard 关联展示

完成 APM-Prometheus 数据集成后，除了可以通过 Grafana 进行自定义图表展示以外，也可以通过腾讯云可观测平台 Dashboard 进行自定义图表展示，并支持将展示结果嵌入到 APM 控制台的应用详情页中。Dashboard 是腾讯云可观测平台针对云产品指标监控数据提供的具备可视化和分析功能的智能仪表盘。关于 Dashboard 的更多介绍，请参见 Dashboard 概述 。

创建以 Prometheus 为数据源的 Dashboard

1. 参考 新建 Dashboard 创建一个 Dashboard。
2. 单击 Dashboard 上方的 按钮，在弹出的下拉框中选择**模板变量**。
3. 单击**初始化 Prometheus 预设变量**，系统将自动生成**地域**和 **Prometheus 实例**两个模板变量。

4. 单击 Dashboard 右上方的**保存**后，就得到了一个以 Prometheus 为数据源的 Dashboard。通过下拉框选择具体和地域和 Prometheus 实例，可以从任何一个腾讯云 Prometheus 监控服务的实例中获取指标数据。参考 图表配置 ，可以基于 PromQL 语句以及 Dashboard 自带的图表创建丰富的自定义大盘，其使用方式和 Grafana 是类似的。



APM 控制台关联展示 Dashboard

在 Prometheus 指标以及 Dashboard 的使用过程中，存在这样一种非常普遍的场景：

- 对于配置好的 Dashboard，从 APM 应用的维度过滤指标数据，并展示图表。
- 将展示结果嵌入到 APM 控制台的应用详情中。
- 同一个 APM 业务系统中的多个应用，都共享同一个 Dashboard，不需要重复配置。

对于这种使用场景，可以通过 APM-Dashboard 关联轻松实现。

> **说明：**
> 在执行如下步骤前，需要先确保如下前置操作已经完成：
> - 完成 APM-Prometheus 关联。
> - 通过 OpenTelemetry API 往 Prometheus 实例写入指标数据。
> - 创建以 Prometheus 为数据源的 Dashboard。
> - 在 Dashboard 的过滤选项中指定 Prometheus 实例。

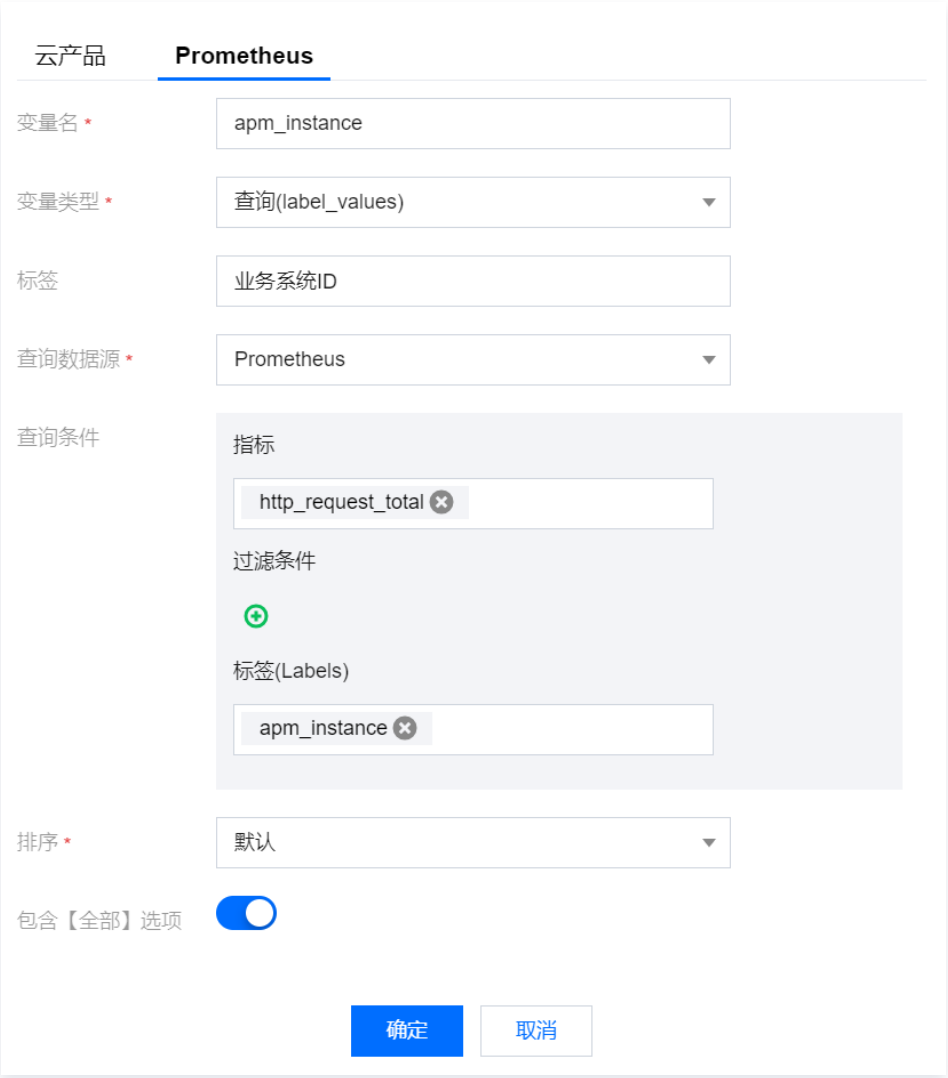
操作步骤

1. 配置应用与 Dashboard 的关联。前往 APM 控制台 > 系统配置 > 业务系统配置，在 **Dashboard 关联**栏目中，关联已经创建好的 Dashboard，需要确保该 Dashboard 使用 Prometheus 数据源。在业务系统中关联 Dashboard，相当于该业务系统的所有应用都完成了关联，您也可以在**系统配置 > 应用配置**中，针对指定的应用覆盖业务系统级别的配置。

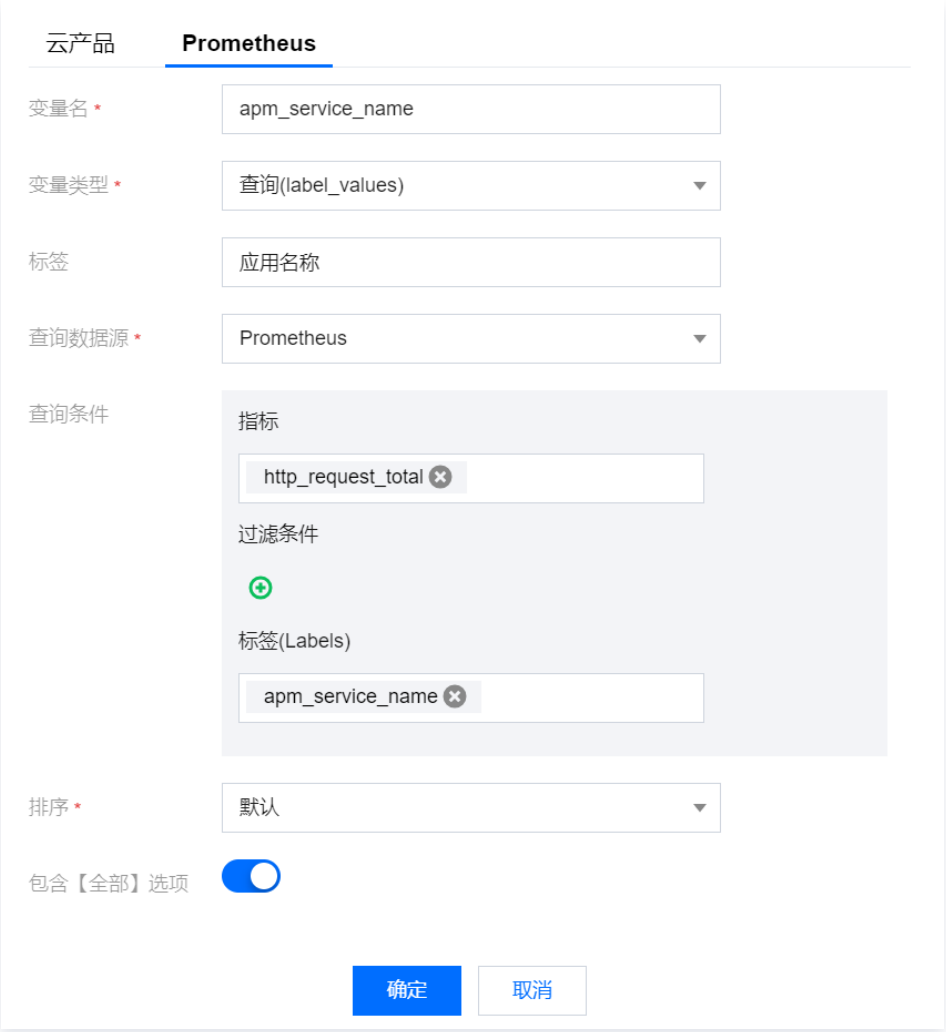
2. 配置必要的模板变量。

为了实现 Dashboard 能够基于特定的应用过滤指标，需要为 Dashboard 增加必要的应用过滤选项，包括业务系统和应用两个模板变量。回顾前面的章节，在完成了 APM-Prometheus 关联后，通过 OpenTelemetry API 写入的指标都额外增加了 `apm_instance` 和 `apm_service_name` 两个标签，分别代表 APM 业务系统 ID 以及应用名。通过这两个标签就能非常方便的为 Dashboard 增加必要的模板变量。

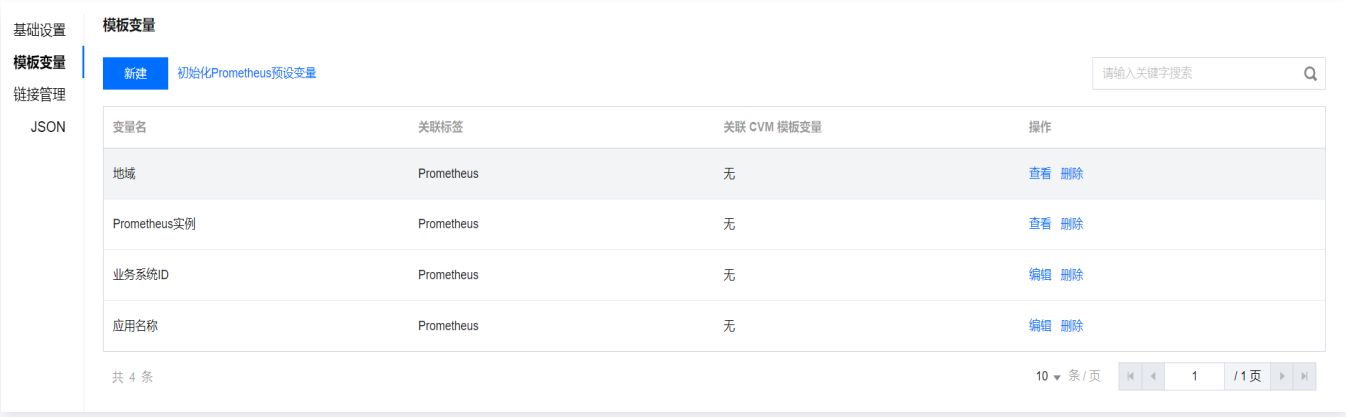
2.1 前往 Dashboard，进入 Dashboard 设置 > **模板变量**，参考下图的配置项，就可以增加业务系统 ID 模板变量。请确保将**变量名**设置为 `apm_instance`，**变量类型**设置为 **查询** `(label_values)`，**查询条件**中包含唯一的标签 `apm_instance`：



2.2 接下来，参考下图的配置项，就可以增加应用名称模板变量。请确保将**变量名**设置为 `apm_service_name`，**变量类型**设置为 **查询** `(label_values)`，**查询条件**中包含唯一的标签 `apm_service_name`。



2.3 完成了所有设置以后，再次检查 Dashboard 中是否包含如下4个必要的模板变量。



3. 在图表的 PromQL 语句中，增加业务系统 ID 和应用名称过滤条件。

在如下 HTTP请求数 图表中，PromQL 语句为

```
sum by(method)
(irate(http_request_total{apm_instance="$apm_instance",apm_service_name="$apm_service_name"}
[5m]))
```

，这样就能够基于业务系统 ID 和应用名称过滤数据。

配置完成之后，检查图表能否正确的展示，并检查**业务系统 ID** 和**应用名称**这2个过滤条件是否能正常工作。

4. 在应用详情中查看内嵌的 Dashboard。

前往 APM 控制台 > 应用列表，选择关联了 Dashboard 的应用后，就能看到 **Dashboard** 标签页，在此标签页中， 将以内嵌的方式展示 Dashboard。由于 Prometheus 地域、Prometheus 实例、业务系统 ID、应用名称这4个必要的模板变量都已经自动指定，所以在内嵌的视图中，这4个过滤条件是隐藏的。如果多个应用关联了同一个 Dashboard，通过切换不同应用，就能够方便的查看每个应用的 HTTP 请求数情况。

至此，就已经完成了 APM 与 Dashboard 关联的全部步骤。

> **说明：**
> 对于没有使用 Prometheus 数据源的 Dashboard，同样可以实现 APM 与 Dashboard 关联，通过内嵌的方式在应用详情中展现 Dashboard 图表。虽然系统无法自动填入应用相关的过滤条件，但在特定的场景中，这样的用法也能很好的结合 APM 以及 Dashboard 的能力，为用户带来更便捷的使用。

自定义 TraceID

最近更新时间：2025-04-29 16:54:34

应用接入 APM 以后，TraceID 由系统自动生成，在绝大多数的情况下，您不需要自定义 TraceID，也不需要关注 TraceID 生成逻辑。如果需要实现链路与日志关联分析，也仅需要在应用中将获取到 TraceID 输出日志，具体实现方式请参见 链路与日志关联分析 。
只有在极少数特殊的场景中，才可能存在自定义 TraceID 的需求。本文列出了其中一种典型场景，您可以参考本文实现自定义 TraceID。

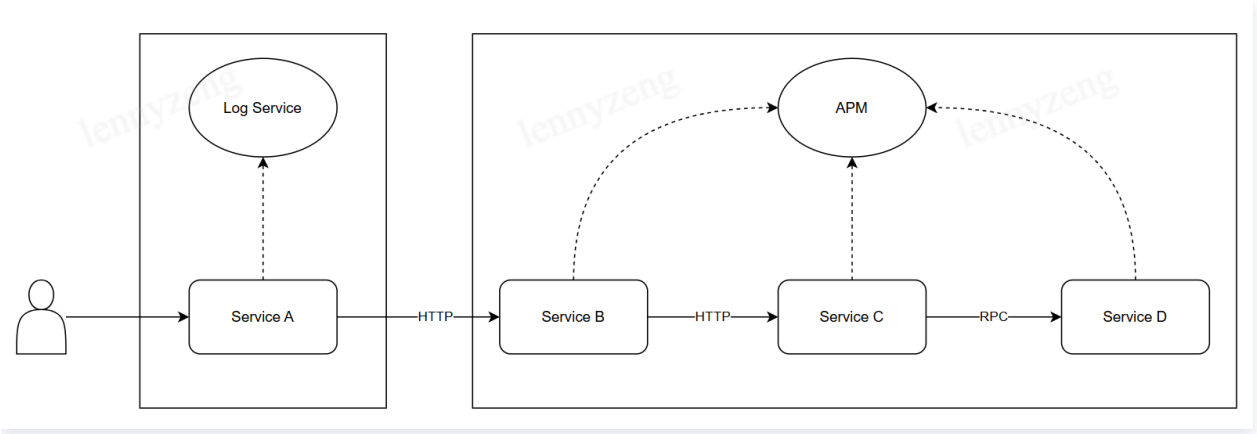
> **说明：**
> 本文适用于使用 OpenTelemetry 方案的业务系统，如果您的业务系统使用 Skywalking 等方案，需要调整 TraceID 的生成规则以及传递方式。

场景分析

在实际业务中，如果某应用同时满足如下条件，可以考虑在此应用中自定义 TraceID：

- 该应用**没有接入 APM**。
- 该应用的**下游应用接入了 APM**。
- 该应用对下游应用的请求基于 **HTTP 协议**。
- 对于经过该应用的链路，需要**通过特定的 TraceID 关联其他的业务场景**。例如存在 TraceID 与订单 ID 之间的映射关系，或者需要通过 TraceID 查询该应用的日志数据。

以如下架构图为例，Service A 就是一个符合自定义 TraceID 标准的应用。对于 Service A 收到的每条用户请求，都根据本文的步骤自定义 TraceID，就能保持 TraceID 在整个系统中的传递。



实现步骤

自定义 TraceID 的流程严格遵循 W3C Trace 规范，包括 TraceID/SpanID 的生成，以及需要在 HTTP 请求中添加的 Header。如果任何一个环节不满足规范，自定义生成的 TraceID 将被 APM 忽略，这种情况不会对实际业务有任何影响 ，但接入 APM 的应用会重新生成新的 TraceID。

生成 TraceID

TraceID 表现为一个长度为32的字符串，其中每个字符为一个16进制数字，取值范围为：0,*1,2,3,4,5,6,7,8,9,a,b,c,d,e,f*，例如 `4bf92f3577b34da6a3ce929d0e0e4736` 。针对每次用户操作，都需要生成新的 TraceID 与之对应，请自行确保 TraceID 的随机性，并注意全零情况 `00000000000000000000000000000000` 会被视为不合规 TraceID。

生成 SpanID

SpanID 表现为一个长度为16的字符串，其中每个字符为一个16进制数字，取值范围为：0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f，例如 `00f067aa0ba902b7` 。针对每次用户操作，都需要生成新的 SpanID 与之对应，请自行确保 SpanID 的随机性，并注意全零的情况 `0000000000000000` 会被视为不合规 SpanID。

> **说明：**
> 此处的 SpanID 和 W3C 规范中的 parent-id 是同一个概念。

拼接 Traceparent 字段

Traceparent 字段包含4个组成部分，通过 `-` 进行连接，这4个部分分别是：

- **version**：固定为 `00`
- **trace-id**：生成的 TraceID
- **parent-id**：生成的 SpanID
- **trace-flags**：固定为 `01`

例如， `00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01` 就是一个合法的 Traceparent 。

在 HTTP 请求中添加 Traceparent Header

在发往下游应用 HTTP 请求中，添加名为 `traceparent` 的 HTTP Header，其值为上一步拼接好的 Traceparent 字段，请注意 Header 名为小写。
这样就完成了自定义 TraceID 的全部流程，对于下游接入 APM 的应用，都将使用自定义的 TraceID 标识一条链路。

快速授予业务系统级别的访问权限

最近更新时间：2025-05-08 14:45:22

应用性能监控（APM）提供了完善了权限管理机制，您可以通过访问管理（CAM）灵活的配置访问权限，详情请参见 APM 访问管理。APM 默认创建了预设策略 QcloudAPMFullAccess（全读写访问权限）和 QcloudAPMReadOnlyFullAccess（只读访问权限），如果您希望针对 APM 进行粗粒度的权限管理，可以参见 策略授予，对子用户授予 APM 的全读写访问权限或者只读访问权限。
另外一种典型的使用场景，是根据业务系统的粒度进行权限管理。例如您的账号下有两个业务系统，它们分别是 A 和 B，您希望某个子用户只能访问业务系统 A，不能访问业务系统 B。您可以参考本文提供的方式，快速实现业务系统级别的访问权限管理。

前提条件

在一个子用户访问 APM 业务系统时，涉及部分前置检查，例如获取该账号 APM 产品的版本信息。这些前置检查涉及的操作，不属于资源级别的操作，因此需要预先对子用户授权这些操作的权限。APM 默认创建了预设策略 QcloudAPMOperationalPrecondition（操作级接口前置条件），包含了所有的前置操作，您可以直接将此预设策略关联到子用户，降低配置复杂度。

1. 使用主账号或拥有 QcloudCamFullAccess 权限的子用户进入 访问控制 > 策略 。
2. 通过搜索找到预设策略 QcloudAPMOperationalPrecondition，单击右侧的**关联用户组/角色** 。
3. 在弹出的对话框中，找到需要授予权限的用户/用户组/角色，选择后单击**确定**。

授予指定业务系统的访问权限

1. 您需要预先获取业务系统的 ID，可以前往 APM 控制台 > 资源管理 获取，业务系统 ID 带有 apm- 前缀，本文中以 apm-Y7o7aeOPA 为例。
2. 使用主账号或拥有 QcloudCamFullAccess 权限的子用户进入 访问管理 > 策略 。
3. 单击**新建自定义策略**，按照如下两种场景创建自定义策略，并完成自定义策略与用户/用户组/角色的关联。

场景一：授予该业务系统的全读写访问权限

1. 在弹出的对话框中，选择**按策略生成器创建**，进入可视化编辑器。
2. 在**服务**栏目中选择**应用性能监控（APM）**。
3. 在**操作**栏目中选择**全部操作(*)**。
4. 在**资源**栏目中选择**特定资源**，单击 apm-instance 对应的**添加资源六段式**。在页面右侧滑出的窗口中，往**资源**栏目填入业务系统 ID，并单击**确定**。
5. 单击**下一步**，为该自定义策略命名，并将其关联到指定的用户/用户组/角色。

场景二：授予该业务系统的只读访问权限

1. 在弹出的对话框中，选择**按策略语法创建**。
2. 选择**空白模板**，单击**下一步**。
3. 在**策略内容**中，填入以下 JSON。

> **注意：**
> 在 resource 字段中，uin/ 后面的数字需要替换为您的主账号 ID，apm-instance/ 后面的内容要替换为您的目标业务系统 ID。

```
{
    "statement": [
        {
            "action": [
                "apm:*"
            ],
            "condition": {
                "numeric_equal": {
                    "qcs:read_only_action": 1
                }
            },
            "effect": "allow",
            "resource": [
                "qcs::apm::uin/150000000:apm-instance/apm-Y7o7aeOPA"
            ]
        }
    ],
    "version": "2.0"
}
```

4. 填写策略名称，并确认策略内容是否为下图所示，然后单击**完成**。
5. 自定义策略创建完成后，单击**关联用户/用户组/角色**，完成与目标用户的关联。

如何同时赋予多个业务系统的访问权限?

CAM 的策略语法支持灵活的配置，您可以在 JSON 内容的 **resource** 字段中，同时指定多个 APM 业务系统。

1. 进入前一步骤中创建的自定义策略，在策略语法中进入 **JSON** 模式，单击编辑图标。
2. 在 resource 字段中，基于策略语法加入更多的业务系统，然后单击**保存**。

```
{
    "statement": [
        {
            "action": [
                "apm:*"
            ],
            "condition": {
                "numeric_equal": {
                    "qcs:read_only_action": 1
                }
            },
            "effect": "allow",
            "resource": [
                "qcs::apm::uin/150000000:apm-instance/apm-Y7o7aeOPA",
                "qcs::apm::uin/150000000:apm-instance/apm-another-instance"
            ]
        }
    ],
    "version": "2.0"
```

```
}
```

除此之外，您也可以基于可视化策略生成器修改策略语法，达到同样的效果，本文不再赘述。

参考信息

OpenTelemetry 方案支持的组件和框架的版本

最近更新时间：2024-08-23 15:00:31

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

支持的 Java 版本

腾讯云增强版 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、Java 21等 LTS 版本，关于 LTS 版本，请参考 官方文档 。

支持的依赖与框架

腾讯云增强版 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，腾讯云增强版 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

腾讯云增强版 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 。

OpenTelemetry-Dotnet 方案支持的组件和框架

支持的 .NET 版本

- .NET SDK ⩾ 6
- .NET Framework 暂不支持自动接入

支持的依赖与框架

OpenTelemetry-Dotnet 方案对于 .NET 系的常用依赖库和框架，包括 ASPNET、HTTPCLIENT、MYSQLCONNECTOR 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 。

Skywalking 方案支持的框架和组件

最近更新时间：2024-09-05 14:46:01

Skywalking Java 探针

Skywalking Java 探针支持的 Java 版本

Skywalking Java 探针支持 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 。

Skywalking Python 探针

Skywalking Python 探针支持的 Python 版本

Skywalking Python 探针支持 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 。

Skywalking Go 探针

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 。

APM 数据协议标准
APM 链路协议标准

最近更新时间：2025-03-10 21:38:43

本文将阐述应用性能监控 APM 的链路（Trace）协议标准，参与此协议标准，用户可以从 DescribeGeneralOTSpanList 云 API 中获取链路数据，也可以在应用中进行自定义埋点增强，覆盖更多的框架和类库。

链路协议设计原则

应用性能监控 APM 的链路协议设计遵循 OpenTelemetry 标准，关于链路的基本概念，可以参考 OpenTelemetry Trace 获得更详情的信息。如果应用通过 OpenTelemetry 方案接入，不管是从云 API 中获取链路数据，还是在代码中进行自定义埋点增强，都能直接对齐 OpenTelemetry 标准，这是 APM 推荐的接入方案。如果应用通过非 OpenTelemetry 方案（例如 Skywalking 方案）接入，APM 会根据链路语义将其他方案的字段映射到 OpenTelemetry 链路协议标准中。

从云 API 获取链路数据

应用性能监控 APM 提供 DescribeGeneralOTSpanList 云 API，用于获取链路数据。

响应结果转换

调用 DescribeGeneralOTSpanList 云 API 后，响应结果为如下 JSON 格式文本。

```
{
    "Response":{
        "RequestId":"f796c81b-d0c6-4cf7-b0fa-4d9194ecad10",
        "Spans":"Base64EncodedString",
        "TotalCount":11151
    }
}
```

其中，`Spans` 字段中包含了链路数据的全部内容，由于数据经过了压缩，需要对结果进行如下三步转换，以还原始的文本。

1. 将 `Spans` 字段中的文本进行 Base64解码，得到经过压缩后字节数组。
2. 使用 gzip 对压缩后的字节数组进行解压，得到压缩前的字节数组。
3. 使用 UTF-8字符集，将压缩前的字节数组转换为文本。

以 Java 语言为例，可以参考如下代码片段实现响应结果转换。

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.zip.GZIPInputStream;

public class OtApiDecoder {
    public static String decodeSpans(String spans) {
        Base64.Decoder base64Decoder = Base64.getDecoder();
        byte[] bytes = base64Decoder.decode(spans);
```

```
        try (ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
             GZIPInputStream gis = new GZIPInputStream(bais);
             ByteArrayOutputStream baos = new ByteArrayOutputStream();) {
            byte[] buffer = new byte[1024];
            int bytesRead = 0;
            while ((bytesRead = gis.read(buffer)) != -1) {
                baos.write(buffer, 0, bytesRead);
            }
            bytes = baos.toByteArray();
            String result = new String(bytes, StandardCharsets.UTF_8);
            return result;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

对于其他语言，也有标准的类库能够处理结果转换工作。

链路数据结构

经过上述转换后，得到的结果为 JSON 文本，其数据结构遵循 ptrace 项目的协议格式，可以参考如下示例了解链路数据的详细内容。

```
{
   "resourceSpans":[
      {
         "resource":{
            "attributes":[
               {
                  "key":"service.name",
                  "value":{
                     "stringValue":"java-order-service"
                  }
               }
            ]
         },
         "scopeSpans":[
            {
               "scope":{
                  "name":"apm/traces/spans",
                  "version":"v1.0"
               },
               "spans":[
                  {
                     "traceId":"9bda979bfbf9f27f94232909dc2a0a68",
                     "spanId":"e2f379eb4405dcb5",
                     "parentSpanId":"",
                     "name":"GET /slowSql",
```

```
                    "kind":2,
                    "startTimeUnixNano":"1741082699964148",
                    "endTimeUnixNano":"1741082703966151",
                    "attributes":[
                        {
                            "key":"key1",
                            "value":{
                                "stringValue":"value1"
                            }
                        },
                        {
                            "key":"key2",
                            "value":{
                                "stringValue":"value2"
                            }
                        }
                    ],
                    "status":{
                        "code":0
                    }
                }
            ]
        }
    ]
}
```


链路数据是由多个 Span 组成的结果集，在 JSON 文本中通过应用名进行分组，每个组包含了属于该应用的 Span 列表，通过 `spans` 字段体现。每个 Span 的具体内容都遵循 OpenTelemetry 协议标准，在属性集合（通过 `attributes` 字段体现）中，APM 在 OpenTelemetry 协议标准的基础上，对如下属性进行了增强：

属性	说明
service.instance	代表产生该 Span 的应用实例，取自 Resource 中的的 `host.name` 字段。在使用腾讯云增强版 Java 探针接入，或者 TKE 环境通过 tencent-opentelemetry-operator 一键接入的情况下，探针会自动附带该信息。其他接入方案下，请在接入应用的时候往 Resource 中添加 `host.name`，详情请参见 自定义应用实例属性。
peer.service	代表该 Span 的对端服务。如果当前 Span 的调用角色为 Server，APM 会将此字段设置为发起请求的应用；如果当前 Span 的调用角色为 Client，APM 会将此字段设置为请求的目标应用。
status_code_xx	代表该 Span 的状态码。APM 在此字段中整合了各种协议的状态码，以 `协议_状态码` 的形式表达，例如 `HTTP_404` 或 `gRPC_0`。
origin.operation	代表该 Span 的名称。由于 APM 具有 关于接口名称自动收敛 能力，以 `name` 字段保存的 Span 名称可能会被修改，APM 会将原始的 Span 名称保存到此字段中。
db.statement.excess	代表该 Span 的 SQL 语句，适用于数据库请求 Span。由于保存在指标数据中的 SQL 语句存在长度限制，`db.query.text` 属性中保存的 SQL 语句可能会被截断，APM 会将原始的 SQL 语

	句保存到此字段中。
component	代表该 Span 的埋点组件。腾讯云增强版 Java 探针和部分开源 OpenTelemetry 探针会自动附带该信息。
agent.version	代表应用接入使用的探针版本。腾讯云增强版 Java 探针会自动附带该信息。

自定义埋点增强

如果通过 OpenTelemetry API 进行自定义埋点增强，请关注如下重要的字段信息，如果没有在埋点的时候指定，可能会导致 APM 的部分功能无法使用。各类编程语言使用的 OpenTelemetry API 中都有便捷的方式指定此值。

- 通用场景：

字段名	说明
Span Kind	所有 Span 都必须指定，代表 Span 的类型，在 APM 中被称作调用角色。取值为 `Server`、`Client`、`Producer`、`Consumer`、`Internal` 中的一种。
Span Status	代表 Span 的状态。取值为 `Ok`、`Error`、`Unset` 中的一种。
Exception Type	代表 Span 的错误类型。当 Span 的状态为 `Error` 的时候，建议指定，APM 将基于错误类型的维度对错误 Span 进行聚合统计。
Exception Message	代表 Span 的错误信息。当 Span 的状态为 `Error` 的时候，建议指定。

- 数据库调用：

字段名	说明
db.system.name	代表数据库类型。当 Span 为数据库调用的时候，必须在属性集合（attributes）中指定此字段。
server.address	代表数据库服务的地址。当 Span 为数据库调用的时候，建议在属性集合（attributes）中指定此字段。
server.port	代表数据库服务的端口。当 Span 为数据库调用的时候，建议在属性集合（attributes）中指定此字段。
db.namespace	代表数据库名称。当 Span 为数据库调用的时候，建议在属性集合（attributes）中指定此字段。
db.query.text	代表数据库查询语句（SQL）。当 Span 为数据库调用的时候，必须在属性集合（attributes）中指定此字段。

APM 指标协议标准

最近更新时间：2025-07-01 19:08:21

本文将阐述应用性能监控 APM 固有指标的协议标准，用户可以通过如下两种方式获取应用性能监控 APM 的固有指标：

- 通过云 API 查询指标，请参见 DescribeGeneralMetricData 接口文档 。
- 将指标同步到腾讯云 Prometheus 监控服务，请参见 Prometheus 数据集成以及 Dashboard 关联展示 。

指标数据结构

指标名称

每种指标都有唯一云 API 指标名称和唯一的 Prometheus 指标名称，分别用于云 API 查询以及同步至 Prometheus 这两种获取 APM 固有指标的方式。

指标视图

APM 的固有指标包含5个类别，通过5个指标视图进行标识。在云 API 查询场景，需要在 `ViewName` 字段中指定视图名称。

视图名称	描述
service_metric	**基础性能指标**：统计应用的通用性能表现，例如请求数量、响应时间等。
runtime_metric	**运行时指标**：统计运行时指标，例如 GC 次数、CPU 利用率等。
sql_metric	**数据库调用指标**：统计应用访问数据库的性能表现，例如 SQL 执行次数、响应时间等。
error_metric	**错误调用指标**：统计链路中的错误调用情况，例如错误类型、错误码。
mq_metric	**MQ 调用指标**：统计应用访问消息队列的性能表现，例如消息生产的次数、响应时间等。

指标维度

每种指标都支持多个维度：在云 API 查询场景，维度用于数据过滤；在 Prometheus 集成场景，维度信息将以 Prometheus 标签（Label）的形式输出到 Prometheus 指标中。

通用维度

所有5类指标都包含如下通用维度，在应用接入 APM 的时候，大多数通用维度都可以基于实例属性进行自定义，详情请参见 自定义应用实例属性 。

云 API 维度名称	Prometheus 标签名称	描述
tapm.app.id	tapm_app_id	云 AppID
tapm.instance.key	apm_instance	APM 实例 ID
service.name	apm_service_name	应用名
service.instance	service_instance	应用实例
agent.version	agent_version	探针版本号
service.version	service_version	应用版本

除了通用维度之外，每个视图还支持一系列的特有维度。例如 service_metric 视图下的指标支持 `span.kind`，代表 Span 类型。详情请参见下文。

service_metric 视图（基础性能指标）

指标

云 API 指标名称	Prometheus 指标名称	类型	描述	单位	桶分布
request_count	service_request_count	Gauge	请求数量	-	-
error_request_count	service_error_request_count	Gauge	错误请求数量	-	-
duration_avg	service_duration_avg	Gauge	平均耗时	ms	-
duration_max	service_duration_max	Gauge	最大耗时	ms	-
duration_min	service_duration_min	Gauge	最小耗时	ms	-
duration_total	service_duration_total	Gauge	累计耗时	ms	-
slow_request_count	service_slow_request_count	Gauge	慢调用数量（>2000ms）	-	-
tolerate_request_count	service_tolerate_request_count	Gauge	容忍调用数量（500 - 2000ms）	-	-
range_count_duration	不支持	Histogram	耗时分布	-	0,1,5,10,50,100,200,300,400,500,600,700,800,900,1000,1500,2000,3000,4000,5000,7000,10000,15000

特有维度

云 API 维度名称	Prometheus 标签名称	描述
span.kind	span_kind	Span 类型
service.component	service_component	组件类型
peer.service	peer_service	对端应用名
peer.operation	peer_operation	对端接口名
peer.filled	peer_filled	APM 处理后的对端应用名
operation	operation	接口名

peer.instance	peer_instance	对端实例
status.message	status_message	请求状态信息
status.code	status_code	请求状态码
status_code_xx	status_code_xx	APM 处理后的状态码
error.object	error_object	错误类型
namespace	namespace	命名空间
service.view	service_view	组件视图（枚举值："sql"、"mq"，如果没有则为空）
k8s.deployment.name	k8s_deployment_name	Kubernetes 负载名
k8s.pod.name	k8s_pod_name	Kubernetes Pod 名
k8s.namespace.name	k8s_namespace_name	Kubernetes 命名空间
k8s.cluster.id	k8s_cluster_id	TKE 集群 ID
k8s.region	k8s_region	TKE 集群地域
k8s.pod.ip	k8s_pod_ip	Kubernetes Pod IP
k8s.node.ip	k8s_node_ip	Kubernetes 节点 IP
cvm.region	cvm_region	CVM 地域
cvm.instance.id	cvm_instance_id	CVM 实例 ID
custom_key_1	custom_key_1	自定义维度1
custom_key_2	custom_key_2	自定义维度2
custom_key_3	custom_key_3	自定义维度3

runtime_metric 视图（运行时性能指标）

说明：

- 腾讯云增强版 OpenTelemetry Java 探针支持 runtime_metric 视图。
- 该协议标准适用于腾讯云增强版 OpenTelemetry Java 探针**2.3-20250228**及其之后的版本。如果您使用**2.3-20250228**之前的探针版本，建议 升级探针版本 。
- 部分运行时性能指标在告警平台和云监控 Dashboard 中，被标识为**下线中**，代表该类指标不被推荐使用（例如不支持 ZGC 或者 Java 缓冲区），且不被收录到该协议标准中。在已有的告警策略或者 Dashboard 定义中，下线中的指标将继续生效，但建议尽快 升级探针版本，并通过该协议标准收录的指标进行替换。

指标

云 API 指标名称	Prometheus 指标名称	类型	描述	单位

cpu_usage_percent	runtime_cpu_usage_percent	Gauge	CPU 利用率	%
jvm_heap_usage_percent	runtime_jvm_heap_usage_percent	Gauge	内存利用率	%
jvm_gc_count	runtime_jvm_gc_count	Gauge	GC 次数	个
jvm_gc_time	runtime_jvm_gc_time	Gauge	GC 耗时	ms
jvm_memory_commit	runtime_jvm_memory_commit	Gauge	JVM 内存已提交大小	MB
jvm_memory_max	runtime_jvm_memory_max	Gauge	JVM 内存最大大小	MB
jvm_memory_used	runtime_jvm_memory_used	Gauge	JVM 内存使用大小	MB
thread_new_count	runtime_thread_new_count	Gauge	新建线程数	个
thread_runnable_count	runtime_thread_runnable_count	Gauge	可运行线程数	个
thread_blocked_count	runtime_thread_blocked_count	Gauge	阻塞线程数	个
thread_waiting_count	runtime_thread_waiting_count	Gauge	等待线程数	个
thread_time_waiting_count	runtime_thread_time_waiting_count	Gauge	定时等待线程数	个
thread_terminated_count	runtime_thread_terminated_count	Gauge	终止线程数	个
thread_daemon_count	runtime_thread_daemon_count	Gauge	守护线程数	个
thread_live_count	runtime_thread_live_count	Gauge	活跃线程数	个
thread_peak_count	runtime_thread_peak_count	Gauge	峰值活跃线程数	个
connection_pool_active_size	runtime_connection_pool_active_size	Gauge	活跃连接数	个
connection_pool_current_size	runtime_connection_pool_current_size	Gauge	当前连接数	个
connection_pool_idle_size	runtime_connection_pool_idle_size	Gauge	空闲连接数	个

connection_pool_max_size	runtime_connection_pool_max_size	Gauge	最大连接数	个
connection_pool_wait_size	runtime_connection_pool_wait_size	Gauge	等待连接数	个
connection_pool_usage_rate	runtime_connection_pool_usage_rate	Gauge	连接池使用率	%
thread_pool_active_count	runtime_thread_pool_active_count	Gauge	活跃线程数	个
thread_pool_core_pool_size	runtime_thread_pool_core_pool_size	Gauge	核心线程数	个
thread_pool_max_pool_size	runtime_thread_pool_max_pool_size	Gauge	最大线程数	个
thread_pool_size	runtime_thread_pool_size	Gauge	线程池大小	个
thread_pool_task_count	runtime_thread_pool_task_count	Gauge	线程池任务数	个
thread_pool_usage_rate	runtime_thread_pool_usage_rate	Gauge	线程池使用率	%

特有维度

云 API 维度名称	Prometheus 标签名称	描述
pool.tag	pool.tag	线程池/连接池种类
pool.type	pool.type	线程池/连接池组件类型
pool.name	pool.name	线程池/连接池名称
gc.type	gc.type	GC 类型
collector.name	collector.name	回收器名称
memory.category	memory.category	内存种类
memory.area	memory.area	内存区域
k8s.deployment.name	k8s_deployment_name	Kubernetes 负载名
k8s.pod.name	k8s_pod_name	Kubernetes Pod 名
k8s.namespace.name	k8s_namespace_name	Kubernetes 命名空间
k8s.cluster.id	k8s_cluster_id	TKE 集群 ID
k8s.region	k8s_region	TKE 集群地域
k8s.pod.ip	k8s_pod_ip	Kubernetes Pod IP

k8s.node.ip	k8s_node_ip	Kubernetes 节点 IP
cvm.region	cvm_region	CVM 地域
cvm.instance.id	cvm_instance_id	CVM 实例 ID
custom_key_1	custom_key_1	自定义维度1
custom_key_2	custom_key_2	自定义维度2
custom_key_3	custom_key_3	自定义维度3

sql_metric 视图（数据库调用指标）

指标

云 API 指标名称	Prometheus 指标名称	类型	描述	单位
duration_avg	sql_duration_avg	Gauge	平均耗时	-
duration_total	sql_duration_total	Gauge	累计耗时	-
last_error_occur_time	sql_last_error_occur_time	Gauge	最后错误发生时间	ms
error_request_count	sql_error_request_count	Gauge	错误请求数量	ms
request_count	sql_request_count	Gauge	请求数量	ms
slow_sql_count	sql_slow_sql_count	Gauge	慢 SQL 数量（>2000ms）	-

特有维度

云 API 维度名称	Prometheus 标签名称	描述
service.component	service_component	组件类型
span.kind	span_kind	Span 类型
db.statement	db_statement	执行语句
db.instance	db_instance	数据库的实例地址
db.ip	db_ip	IP 地址
db.operation	db_operation	操作类型
error.object	error_object	错误类型
db.system	db_system	数据库类型
attack.subtype	attack_subtype	注入类型

attack.symbol	attack_symbol	注入标记：0无注入，1疑似注入
attack.level	attack_level	注入等级：Critical、Important、Moderate、Low、info
custom_key_1	custom_key_1	自定义维度1
custom_key_2	custom_key_2	自定义维度2
custom_key_3	custom_key_3	自定义维度3

error_metric 视图（错误调用指标）

指标

云 API 指标名称	Prometheus 指标名称	类型	描述	单位
error_request_count	error_error_request_count	Gauge	错误请求数	-
last_error_occur_time	error_last_error_occur_time	Gauge	最后错误发生时间	ms

特有维度

云 API 维度名称	Prometheus 标签名称	描述
service.component	service_component	组件类型
span.kind	span_kind	Span 类型
operation	operation	接口名
error.object	error_object	错误类型
peer.service	peer_service	对端应用名
custom_key_1	custom_key_1	自定义维度1
custom_key_2	custom_key_2	自定义维度2
custom_key_3	custom_key_3	自定义维度3

mq_metric 视图（MQ 调用指标）

指标

云 API 指标名称	Prometheus 指标名称	类型	描述	单位	桶分布
duration_avg	mq_duration_avg	Gauge	平均耗时	ms	-
duration_total	mq_duration_total	Gauge	累计耗时	ms	-
error_request_co	mq_error_request_	Gauge	错误数量	-	-

unt	count				
request_count	mq_request_count	Gauge	请求数	-	-
range_count_duration	不支持	Histogram	耗时分布	-	0,1,5,10,50,100,200,300,400,500,600,700,800,900,1000,1500,2000,3000

特有维度

云 API 维度名称	Prometheus 标签名称	描述
component	component	组件类型
span.kind	span_kind	Span 类型
peer.service	peer_service	对端应用名
mq.topic	mq_topic	消息队列 Topic 名称
mq.broker	mq_broker	消息队列的 Broker 地址
operation	operation	接口名
status.code	status_code	请求状态
status_code_xx	status_code_xx	APM 处理后的状态码
error.object	error_object	错误类型

探针（Agent）版本信息

最近更新时间：2025-05-15 11:46:22

本文展示了腾讯云自研 APM 探针的版本信息，并提供下载地址。对于其他由开源社区提供的探针或 SDK 信息，包括 OpenTelemetry、Skywalking、Jaeger 等，请参考对应的开源社区文档。

腾讯云增强版 OpenTelemetry Java 探针

腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-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，腾讯云增强版 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。同时，腾讯云增强版 OpenTelemetry Java 探针也遵循了 OpenTelemetry 的协议标准，当自动埋点不满足用户场景时，可以参考 OpenTelemetry 官方文档 进行自定义业务埋点。

版本	发布说明	发布日期	VPC 内网下载地址	
2.11-20250509 下载探针	• 支持 Arthas。 • 支持远程命令攻击检测、内存马攻击检测。 • 支持 JVM 缓冲区指标上报。	2025年5月9日	广州	`https://tencentcloud-otel-java-agent-gz-1258344699.cos-internal.ap-guangzhou.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar`
			上海	`https://tencentcloud-otel-java-agent-sh-1258344699.cos-internal.ap-shanghai.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar`
			北京	`https://tencentcloud-otel-java-agent-bj-1258344699.cos-internal.ap-beijing.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar`
			成都	`https://tencentcloud-otel-java-agent-cd-1258344699.cos-internal.ap-chengdu.myqcloud.com/2.11-`

	20250509/opentelemetry-javaagent.jar
中国香港	https://tencentcloud-otel-java-agent-hk-1258344699.cos-internal.ap-hongkong.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar
北京金融	https://tencentcloud-otel-java-agent-bj-fsi-1258344699.cos-internal.ap-beijing-fsi.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar
上海金融	https://tencentcloud-otel-java-agent-sh-fsi-1258344699.cos-internal.ap-shanghai-fsi.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar
新加坡	https://tencentcloud-otel-java-agent-sg-1258344699.cos-internal.ap-singapore.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar
法兰克福	https://tencentcloud-otel-java-agent-eu-1258344699.cos-internal.eu-frankfurt.myqcloud.com/2.11-

				20250509/opentelemetry-javaagent.jar
			硅谷	https://tencentcloud-otel-java-agent-usw-1258344699.cos-internal.na-siliconvalley.myqcloud.com/2.11-20250509/opentelemetry-javaagent.jar
2.3-20250228 下载探针	• 修复部分环境探针无法获取实例名称的问题。 • JVM 运行时指标覆盖更多的 GC 回收器以及内存区域。	2025年2月28日	广州	https://tencentcloud-otel-java-agent-gz-1258344699.cos-internal.ap-guangzhou.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar
			上海	https://tencentcloud-otel-java-agent-sh-1258344699.cos-internal.ap-shanghai.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar
			北京	https://tencentcloud-otel-java-agent-bj-1258344699.cos-internal.ap-beijing.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar
			成都	https://tencentcloud-otel-java-agent-cd-1258344699.cos-internal.ap-chengdu.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar

中国香港	`https://tencentcloud-otel-java-agent-hk-1258344699.cos-internal.ap-hongkong.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
北京金融	`https://tencentcloud-otel-java-agent-bj-fsi-1258344699.cos-internal.ap-beijing-fsi.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
上海金融	`https://tencentcloud-otel-java-agent-sh-fsi-1258344699.cos-internal.ap-shanghai-fsi.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
新加坡	`https://tencentcloud-otel-java-agent-sg-1258344699.cos-internal.ap-singapore.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
法兰克福	`https://tencentcloud-otel-java-agent-eu-1258344699.cos-internal.eu-frankfurt.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
硅谷	`https://tencentcloud-otel-java-agent-usw-1258344699.cos-`

				`internal.na-siliconvalley.myqcloud.com/2.3-20250228/opentelemetry-javaagent.jar`
2.3-20250131	• 支持新版本性能剖析。 • 支持 OkHttp 线程池分析能力，以及 Hikari、Redisson 等框架的连接池分析能力。	2025年1月31日	-	-
2.3-20241231	• 支持方法栈分析功能。 • 优化探针在高负载场景的性能。	2024年12月31日	-	-
2.3-20241130	• 支持应用安全相关功能。 • 支持线程分析功能。 • 支持 JVM ZGC 指标上报。	2024年11月30日	-	-
2.3-20241031	• 支持通过 Skywalking 链路传播协议（SkyWalking Cross Process Propagation Headers Protocol）实现跨应用链路信息传递。 • 支持通过 OpenTelemetry API 上报自定义指标，并输出到关联 Prometheus 实例。 • 支持自定义和应用保持机制相关的阈值。	2024年10月31日	-	-
2.3-20240930	• 新增 JVM 指标上报：代码缓存区、压缩类空间。 • 针对 DB 调用优化埋点丰富度，自动上报数据库类型、数据库地址等附加信息。	2024年9月30日	-	-

2.3-20240831	• 支持多项探针配置，可以通过控制台下发，实时生效。 • 新增 ons-client 自动埋点。 • 解决在使用 Fastjson 库的情况下有可能存在的埋点性能问题。	2024年8月31日	-	-
2.3-20240730	• 支持 Java 21。 • 新增 XXL-JOB 1.9.2+ 自动埋点上报。 • 支持对 Spring Service 注解方法的自动埋点上报。	2024年7月30日	-	-
2.1-20240701	• 支持通过 OpenTelemetry API 进行自定义埋点增强。 • 新增一系列组件与框架的自动埋点支持，包括 Spring Cloud Gateway 2.0+ 以及 Pulsar 2.8+ 等。	2024年7月1日	-	-
1.16-20240328	优化高负载场景下的探针稳定性。	2024年3月28日	-	-
1.16-20240305	• 支持线程池、连接池指标上报 。 • 通过火焰图分析CPU/内存高负载的问题根因。	2024年3月5日	-	-

探针支持周期

为了不断提升腾讯云增强版 OpenTelemetry Java 探针的产品能力以及稳定性，支持开源组件的新版本，及时修复探针缺陷以及安全漏洞，应用性能监控 APM 将对探针进行持续迭代，定期发布新版本探针。建议您跟随探针发布节奏，及时更新探针版本。
每个正式发布的腾讯云增强版 OpenTelemetry Java探针，从发布日开始计算，固定支持周期为6个月。从第4个月开始，APM 会在控制台提醒您探针已经临近过期；从第6个月开始，APM 会在控制台提醒您探针已经过期。探针过期以后，不会影响 APM 的使用，如果您需要寻求探针方面的技术支持，建议您可先升级探针版本。关于探针升级，请参见 升级探针版本 。

其他探针

除了腾讯云增强版 OpenTelemetry Java 探针之外，其他的探针均由开源社区提供，包括 OpenTelemetry 社区、Skywalking 社区，请参考接入指南进行探针安装。
对于部署在容器服务 TKE 上的应用，腾讯云可观测团队提供了 Operator 方案：tencent-opentelemetry-operator，此方案在社区 opentelemetry-operator 基础上构建，可以实现探针自动注入，方便应用快速接入 APM。tencent-opentelemetry-

operator 支持如下探针版本：

语言	探针版本	发布说明	发布日期
Python	0.45b0（最新版本）	0.45b0 发布说明	2024年6月4日
	0.41b0	0.41b0 发布说明	2023年10月13日
Node.js	0.51.0（最新版本）	0.51.0 发布说明	2024年6月24日
	0.44.0	0.44.0 发布说明	2023年10月25日
.Net	1.6.0（最新版本）	1.6.0 发布说明	2024年4月29日
	1.2.0	1.2.0 发布说明	2024年1月18日

腾讯云增强版 OpenTelemetry Java 探针压测报告

最近更新时间：2025-04-11 14:42:42

腾讯云增强版 OpenTelemetry Java 探针通过运行态字节码增强，实现了零代码侵入的应用性能管理能力。字节码增强技术会对应用产生一定的性能开销，这是所有采用类似技术的应用性能管理方案都无法避免的。但腾讯云可观测团队在探针中引入了多项重要优化技术，将探针的性能开销控制在极低的范围内，打消了用户对于探针影响应用性能与稳定性的顾虑。
在本篇压测报告中，我们基于真实场景模拟腾讯云增强版 OpenTelemetry Java 探针在不同业务流量下产生的性能开销，您可以参考本篇压测报告，对 APM 产品的最终造型进行充分评估。

测试用例

本次测试所采用的场景和方式，基于 SkyAPMTest 开源社区的探针性能测试项目 实现，您可以前往社区获取项目的源代码。该项目基于 Spring 框架编写了应用程序，包含 Spring Boot、Spring MVC，模拟的 Redis 客户端，HikariCP 连接池（匹配模拟的 MySQL 客户端）。接入腾讯云增强版 OpenTelemetry Java 探针后，对于每个事务，探针会抓取5个 Span (1个 Tomcat 调用、1个 Spring MVC 框架调用、2个 Redis 请求和1个 MySQL 请求)。

- **应用部署环境**：基于 TKE 部署，2核4G Pod；
- **测试地域**：北京
- **测试时长**：10分钟
- **探针版本**：2.3-20240720
- **业务流量**：500, 1000, 2000, 4000。

基线性能指标

在不接入 APM 的情况下，性能表现如下：

业务流量（TPS）	CPU 利用率	内存利用率	平均响应时间（ms）
500	32.83%	24.30%	182
1000	32.88%	26.09%	370
2000	32.95%	26.70%	752
4000	32.52%	30.59%	1522

安装探针后的性能指标

应用接入 APM 后，性能表现如下：

业务流量（TPS）	CPU 利用率	内存利用率	平均响应时间（ms）
500	39.38%	33.42%	201
1000	39.57%	32.03%	418
2000	39.68%	33.13%	859
4000	39.69%	35.77%	1730

探针性能开销

对比安装探针后的性能指标以及基线性能指标，探针产生的性能开销如下：

业务流量（TPS）	CPU 利用率	内存利用率	平均响应时间（ms）
500	+5.96%	+9.12%	+19
1000	+6.69%	+5.94%	+48
2000	+6.73%	+6.43%	+107
4000	+7.16%	+5.18%	+208

总结

- 探针产生的 CPU 和内存开销，在10%以内。
- 探针会对请求响应时间带来一定的影响，但都在毫秒级别。

腾讯云增强版 OpenTelemetry 探针熔断保护机制

最近更新时间：2024-11-11 10:15:12

腾讯云增强版 OpenTelemetry Java 探针实现了熔断保护机制。在应用负载特别高的时候，探针会对埋点以及数据上报进行降级处理，在极端情况下降低对正常业务流程的影响。当应用负载回到正常水位后，探针会自动恢复埋点和数据上报。

表现行为

熔断保护机制的默认阈值请参考如下表格：

判断规则		丢弃部分链路数据		完全关闭数据上报	
		触发阈值	恢复阈值	触发阈值	恢复阈值
内存使用率	2.3-20241031 之前的探针版本	65%	60%	75%	70%
	2.3-20241031 以及之后的探针版本	-	-	90%	85%
CPU 使用率	2.3-20241031 之前的探针版本	80%	75%	90%	85%
	2.3-20241031 以及之后的探针版本	-	-		

其中，内存使用率 = 已使用堆内存 / JVM 配置的最大堆内存，CPU 使用率 = 统计周期内 JVM 使用的总 CPU 时间 /（统计周期跨度 * CPU 核数）。
以 CPU 使用率为例，熔断保护机制的表现行为如下：

- **2.3-20241031 之前的探针版本**：当应用的 CPU 使用率达到80%，探针会引入采样机制，随机丢弃50%的链路。如果接下来 CPU 使用率下降到 75% 或以下，探针会恢复正常的数据上报。如果 CPU 使用率达到了90%，探针会完全关闭数据上报。
- **2.3-20241031 以及之后的探针版本**：当应用的 CPU 使用率达到90%，探针会完全关闭数据上报。如果接下来 CPU 使用率下降到 85% 或以下，探针会恢复正常的数据上报。

> **说明：**
> 从 2.3-20241031 版本开始，探针对埋点和数据上报进行了优化，在提升性能的同时，**不再提供丢弃部分链路数据的保护机制**。使用 2.3-20241031 及之后的探针版本能够提升数据上报成功率，降低探针性能开销，并支持自定义熔断保护阈值，**建议您尽快 升级探针版本** 。

自定义阈值

> **说明：**
> 腾讯云增强版探针 2.3-20241031 或以上的版本支持自定义熔断保护阈值。

关于自定义阈值的相关参数，请参考如下表格：

设置方式	判断规则	关闭数据上报 - 触发阈值	关闭数据上报 - 恢复阈值

JVM 启动参数方式	内存使用率	disable.reporting.on.memory.percentage	recover.reporting.on.memory.percentage
	CPU 使用率	disable.reporting.on.cpu.percentage	recover.reporting.on.cpu.percentage
环境变量方式	内存使用率	DISABLE_REPORTING_ON_MEMORY_PERCENTAGE	RECOVER_REPORTING_ON_MEMORY_PERCENTAGE
	CPU 使用率	DISABLE_REPORTING_ON_CPU_PERCENTAGE	RECOVER_REPORTING_ON_CPU_PERCENTAGE

您可以通过添加 JVM 启动参数或设置环境变量修改默认阈值。以基于内存使用率的触发阈值例，如果要将其设置为 95%，可以在 JVM 启动参数中添加 `-Ddisable.reporting.on.memory.percentage=95`，完整的 Java 启动命令如下：

```
java -javaagent:/path/to/opentelemetry-javaagent.jar \
-Dotel.resource.attributes=service.name=myService,token=myToken\
-Dotel.exporter.otlp.endpoint=http://pl-demo.ap-guangzhou.apm.tencentcs.com:4317 \
-Ddisable.reporting.on.memory.percentage=95
-jar SpringCloudApplication.jar
```

也可以在环境变量中添加如下内容：

```
export DISABLE_REPORTING_ON_MEMORY_PERCENTAGE=95
```

如果同时添加 Java 启动参数以及环境变量，前者会具有更高的优先级。

注意事项

- 如果要禁用基于内存使用率的熔断保护机制，请将触发阈值设置为0，例如在 JVM 启动参数中添加 `-Ddisable.reporting.on.memory.percentage=0`。
- 如果要禁用基于 CPU 使用率的熔断保护机制，请将触发阈值设置为0，例如在 JVM 启动参数中添加 `-Ddisable.reporting.on.cpu.percentage=0`。
- 同时将基于内存使用率和 CPU 使用率的触发阈值设置为0，将彻底禁用熔断保护机制。
- 在熔断保护机制开启的情况下，请确保每种判断规则的恢复阈值小于触发阈值，否则熔断保护机制将不生效。在大多数情况下，恢复阈值比熔断阈值低3到10个百分点是比较好的选择。

API 参数字典

最近更新时间：2025-05-16 10:07:32

修改应用配置信息

ModifyGeneralApmApplicationConfig API 用于修改应用配置信息，应用配置通过 `Tags` 参数进行传递，`Tags` 是一个由 ApmTag 组成的数组，ApmTag 支持如下键值对：

Key	Value 类型	示例值	描述
UrlConvergenceSwitch	int64	1	URL 收敛开关。0：关，1：开
UrlConvergenceThreshold	int64	50	URL 收敛阈值
UrlConvergence	string	market.StockService.getStock(.*?),market.StockService.getPreDayS	URL 收敛规则正则，使用换行符 \n 分隔
UrlExclude	string	market.StockService.getStock(.*?),market.StockService.getPreDayS	URL 排除规则正则，使用换行符 \n 分隔
OperationNameFilter	string	RPCServer/market.MarketService2/(.*?)	接口过滤，使用换行符 \n 分隔
ExceptionFilter	string	io.grpc.StatusRuntimeException,java.sql.(.*?)	错误类型过滤，使用换行符 \n 分隔
ErrorCodeFilter	string	400,500	HTTP 状态码过滤
LogSource	string	CLS	日志源
LogRegion	string	ap-guangzhou	日志地域
IsRelatedLog	int64	1	是否开启日志。0：关，1：开
LogTopicID	string	032f3078-10b6-46a0-978f-1696e8e79696	日志主题
LogSet	string	postgres	日志集
AgentEnable	bool	true	探针总开关：true 开启，false 关闭
SnapshotEnable	bool	false	方法栈快照开关：true 开启，false 关闭
SnapshotTimeout	int64	2000	慢调用监听触发阈值（单位：ms）
AgentIgnoreOperation	string	/hello/world,/health/check	忽略接口列表，多个接口用逗号分隔

EnableSecurityConfig	bool	true	是否启用应用级别的应用安全配置，false 的情况下默认使用业务系统级别的配置
IsSqlInjectionAnalysis	int64	1	是否开启SQL注入。1：开，0：关； 在修改时仅在 `EnableSecurityConfig=true` 时有效
IsInstrumentationVulnerabilityScan	int64	1	是否开启组件漏洞检测。1：开，0：关； 在修改时仅在 `EnableSecurityConfig=true` 时有效
IsMemoryHijackingAnalysis	int64	1	是否开启内存马检测。1：开，0：关； 在修改时仅在 `EnableSecurityConfig=true` 时有效
IsRemoteCommandExecutionAnalysis	int64	1	是否开启远程命令执行检测。1：开，0：关； 在修改时仅在 `EnableSecurityConfig=true` 时有效
EnableDashboardConfig	bool	true	是否启用应用级别的 Dashboard 控制台配置，false 的情况下默认使用业务系统级别的配置
IsRelatedDashboard	int64	1	是否关联了 Dashboard 控制台。1：是， 0：否
DashboardTopicID	string	testTopicID	Dashboard TopicID

说明：

- 关于应用配置的更多详细信息，可以参考 应用配置帮助文档 。
- 使用 ModifyGeneralApmApplicationConfig API 接口修改应用配置信息的时候，仅需要在 `Tags` 中填入需要修改的配置项，不需要填入完整的应用配置。

输入示例如下：

```
{
  "InstanceId": "apm-059oXBfTL",
  "Tags": [
    {
      "Key": "UrlConvergence",
      "Value": "RPCServer/market.MarketServiceleetcode9(.*?)"
    },
    {
      "Key": "UrlConvergenceThreshold",
      "Value": "600"
    },
```

```
    {
      "Key": "UrlConvergenceSwitch",
      "Value": "1"
    },
    {
      "Key": "AgentEnable",
      "Value": "true"
    }
  ],
  "ServiceNames": [
    "stock-service",
    "profile-service"
  ]
}
```

关于免费模式的使用限制

最近更新时间：2025-03-07 19:34:52

为了让用户更好的体验产品功能，并降低使用成本，应用性能监控（APM）提供了免费模式，您可以对特定的业务系统开启免费模式，享受对 APM 产品的永久免费使用。开启免费模式后，对该业务系统的收费将**立即停止**，接入该业务系统的应用**不会有任何费用产生**，也不会产生预付费套餐包的额度消耗。直到您主动关闭免费模式后，该业务系统才会按照您指定的计费模式收取费用。

使用限制

在免费模式下，应用性能监控（APM）对部分功能进行了裁剪，需要关闭免费模式才能获得 APM 的完整功能。您需要关注如下使用限制：

- 不保存链路数据。**链路追踪**和**链路详情**这两个功能将受到影响。
- 指标数据固定保存**3天**。对于免费模式和付费模式， APM 分别使用了独立的数据源保存指标数据，当一个业务系统在免费模式和付费模式之间**相互切换的时候，历史指标数据将无法查询**。
- 对于调用角色为 `Internal` 的 Span，不产生指标数据。您可以参考 链路追踪字段描述 获得更多关于 Span 调用角色的详细介绍。在绝大多数场景下，用户只需要关注在 `Server` 、 `Client` 、 `Consumer` 、 `Producer` 这4种调用角色下产生的指标，因此这个限制对于 APM 的正常使用基本没有任何影响。
- **不支持告警功能**。

免费模式支持的功能

除了上述使用限制之外，APM 所有功能在免费模式下都可以正常使用，包括应用性能概览、应用拓扑、JVM 监控、接口监控、数据库调用分析、MQ 调用分析、错误分析，以及应用性能剖析等高阶诊断能力。

免费模式的开启方式

1. 登录 腾讯云可观测平台 控制台。
2. 选择**应用性能监控** > **资源管理** > **业务系统管理**，找到需要开启免费模式的业务系统，单击**修改配置**。
3. 在**更新业务系统**对话框中，打开**开启免费模式**开关，然后单击**确定**。

完成操作后，该业务系统就进入了免费模式，对该业务系统的收费将立即停止。在免费模式下，应用的接入与正常方式没有任何区别，免费模式和付费模式之间的相互切换可以立即生效，不需要重新接入应用。免费模式属于业务系统级别的配置项，在同一个账号下，可以同时存在免费模式和付费模式的业务系统。

关于接口名称自动收敛

最近更新时间：2025-05-30 11:48:02

在微服务、分布式架构普及的今天，单个系统可能承载成千上万个接口，而动态 URL、多版本路径、个性化参数等场景更是加剧了接口命名的复杂性。传统监控工具在遇到接口名称高基数（High Cardinality）问题时，会引发接口名称发散，导致监控数据离散、统计失效，甚至引发监控系统的雪崩。为此，应用性能监控 APM 推出接口名称自动收敛功能，通过自动化归类与动态聚合，让应用管理更加精准、高效、可洞察。

接口名称发散问题

对于应用上报的可观测数据，APM 会按照接口名称（也就是 Span 名称）维度进行聚合统计，计算吞吐量、响应时间、错误率等重要的性能指标。基于接口名称维度的聚合统计，用户可以在接口监控等功能分析每一个接口的性能表现，例如特定接口最近24小时的平均错误率以及 P99响应时间。
接口名称是由应用上报的链路数据决定的，具有极高的灵活度，具备自动埋点能力的探针会根据实例情况选择接口 URL、内部方法名等信息作为接口名称，用户也可以通过 OpenTelemetry API 等方式自定义接口名称。当一个应用的接口名称呈现高基数（High Cardinality）状态时，就会引发接口名称发散问题。例如，当用作接口名的 URL 中包括了用户 ID 信息的时候，因为用户 ID 参数的不同，会生成无数个独立的 URL：`/api/user/123`，`/api/user/124`，`/api/user/125`，`/api/user/126` … 在这种情况下，每个 URL 被视作独立的接口，系统中的接口名称数量极为庞大，在按照接口名称维度进行聚合统计时，实际上每个接口都只和一个特定的用户 ID 相关。这会导致极为严重的问题：

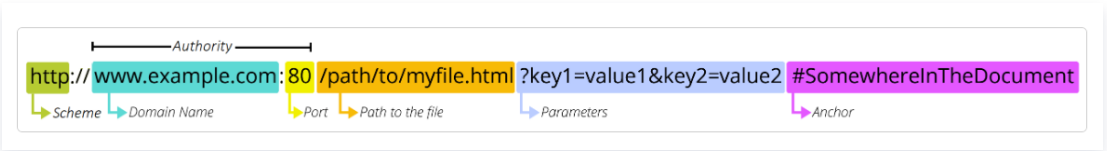
- **统计失效**：这在上述例子中有非常好的体现，`/api/user/{userid}` 被分散在众多包括不同用户 ID 的接口中，无法定位这一类接口的整体性能。
- **分析障碍**：该应用还存在另外两个重要的接口，分别是 `/api/product` 和 `/api/order`，但因为大量以 `/api/user/` 开头的接口充斥在监控数据中，导致这两个重要的接口被忽视掉。
- **查询性能**：高基数下聚合计算变得缓慢，报表加载卡顿甚至超时。尽量 APM 已经在技术架构层面对查询性能进行了优化，但当前端页面需要加载大量离散数据的时候，依然会影响用户体验。
- **告警误报与漏报**：离散的接口名称使得阈值规则难以统一配置，可能忽略共性故障（如所有支付接口超时），却对偶发异常产生误警。

这些问题不仅导致 APM 的使用效率低下，还可能掩盖系统更严重的故障。

接口自动收敛

针对接口名称发散问题，APM 系统通过智能收敛技术，实现接口的自动归类，以更好的进行统一治理以及聚合分析，其中包括多项收敛手段：

- **URL 中的 parameter 与 anchor**。当 APM 识别到指标数据中的接口名称为 URL 的时候，会截去 `?` 以及 `#` 后面的部分。例如 `www.example.com:80/path/to/myfile.html?key1=value1#SomewhereInTheDocument` 会被处理为 `www.example.com:80/path/to/myfile.html`。



image

- **URL 中的长分段**。指标数据中的 URL 通过 `/` 切分后，如果其中一段的长度达到40，会被替换为 `{LONG_STR}`。如果其中一段为纯数字，且长度达到5，则被替换为 `{LONG_NUM}`。例如：`/path/to/1234567890/d4806b8c-2d5b-481d-9598-827b6dd49c10` 会被处理为 `/path/to/{LONG_NUM}/{LONG_STR}`。

- **静态资源**。当 APM 识别到指标数据中的接口名称带有静态资源后缀的时候，会将接口名称中的静态资源部分替换为 `{STATIC_RESOURCE}.xxx`，其中 xxx 为静态资源的后缀。静态资源后缀包括 `.jpg`、`.gif`、`.flv` 等。例如：`/path/to/user_icon_007.jpg` 会被处理为 `/path/to/{STATIC_RESOURCE}.jpg`。
- **IP 地址**。当 APM 识别到指标数据中的接口名称中带有 IP 地址的时候，会将 IP 地址替换为 `{IP}`。例如 `11.146.86.42:9301/generateOrderInfo` 会被处理为 `{IP}:9301/generateOrderInfo`。
- **限制单应用接口总数量**。在比较极端的情况下，如果通过上述收敛手段依然未能彻底解决接口名称发散问题，APM 会将指标数据中的接口名称进行替换，尽量使一段时间内单应用的接口名称基数不超过1000个，超出此限制的接口名称会被替换为 `{EXCEEDED}`。系统会基于应用上报的数据实时检测接口名称发散情况，如果识别接口发散的问题已经解除，会恢复接口名称的原始内容。

APM 的接口自动收敛能力**仅作用于指标数据**，这代表链路数据（保存在 Span 中的数据）中的接口名称不会受任何影响。通过已经收敛后的接口名称，依然可以关联查询到接口所对应的链路数据。

SQL 收敛

在数据库调用分析等功能中，APM 会按照 SQL 维度进行聚合统计，计算调用次数、响应时间等性能指标。跟接口名称一样，SQL 也会导致发散问题，APM 会将数据库调用按照响应时间分为两类：响应时间在2秒以内的为正常调用，响应时间达到2秒的为慢调用。当出现 SQL 发散问题时，APM 也会将指标数据中的 SQL 进行替换，以降低单应用的 SQL 基数，超出限制的 SQL 会根据响应时间分别被替换为 `{OTHER_SLOW_SQL}` 和 `{OTHER_SQL}`。

最佳实践

- 合理利用编程框架对 URL 中可变路径的识别能力，这样在探针侧就可以完成接口名称收敛。例如 Spring MVC 中的 PathVariable：

```
@GetMapping("/api/employeeswithvariable/{id}")
@ResponseBody
public String getEmployeesByIdWithVariableName(@PathVariable("id") String
employeeId) {
    return "ID: " + employeeId;
}
```

- 在应用访问数据库的场景，使用参数化 SQL 语句，例如 JDBC 中的 Prepared Statements。
- 在应用代码中使用 OpenTelemetry API 手动埋点的时候，不要在 Span 名称中加入可变参数，尽量降低 Span 名称的基数。

常见问题

最近更新时间：2025-04-15 11:46:32

选型相关

应用性能监控支持什么语言?

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，请参见 应用接入概述 完成接入。

接入应用是否需要修改代码?

应用性能监控支持多种接入方案，由于编程语言之间特性上的差异，接入工作量也存在比较大的差别。

- Java 语言有最成熟的自动接入方案，针对常用的架构和组件都做了自动埋点，埋点覆盖范围非常高，接入过程中可以实现零代码侵入。
- Python、Node.js、PHP、.Net 等语言也有自动接入方案，但在埋点覆盖度低于 Java 语言，可能需要修改代码引入自定义埋点。
- C++、Erlang 等语言暂时没有自动接入方案，需要修改代码手动接入。

应用性能监控支持什么框架和组件?

理论上支持所有的框架和组件。自动接入方案对常用的框架和组件进行了自动埋点，但对于其他框架和组件，用户也可以修改代码引入自定义埋点，具体请参考特定语言的接入方案。

可以接入部署在其他云的应用吗?

应用性能监控支持混合云部署的场景，只需要确保网络连通性，就可以接入部署在其他云以及线下数据中心的应用。您可以选择通过公网接入点进行上网，或通过腾讯云私有网络等方案上报到内网接入点。

接入应用需要用到的探针和 SDK 是由腾讯云提供的吗?

针对 Java 语言，应用性能监控 APM 提供了腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-OTel Java Agent），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。接入其他语言应用需要用到的探针和 SDK 由开源社区提供，腾讯云不参与开源探针和 SDK 的演进与迭代。

探针需要打到容器镜像中吗?

确保应用能够访问到探针文件即可，不一定需要打到容器镜像中。

是否兼容 OpenTracing 协议?

OpenTracing 标准与 OpenCensus 标准合并后，诞生了 OpenTelemetry 协议标准。应用性能监控 APM 遵循 OpenTelemetry 协议标准，所以可以兼容 OpenTracing 协议。

我应该选择 OpenTelemetry 方案还是 Skywalking 方案?

OpenTelemetry 提供了统一可观测性标准，相比 Skywalking 方案，OpenTelemetry 方案具有更丰富的生态，更活跃的社区，并且支持更多的语言和框架。所以在接入应用的时候，OpenTelemetry 是首选的方案。如果您在之前对 Skywalking 比较熟悉，或者希望从开源 Skywalking 快速迁移到腾讯云应用性能监控 APM，也可以基于 Skywalking 方案接入。
此外，针对 Java语言，应用性能监控 APM 还提供了腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-OTel Java Agent），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。因此，强烈建议 Java 应用 通过腾讯云增强版 OpenTelemetry Java 探针接入 或者选择 K8s 环境自动接入 。

之前使用 Skywalking，如何迁移到应用性能监控?

应用性能监控已兼容 Skywalking 协议标准，您只需要修改上报地址，并在 Resource 的参数中填入业务系统的 Token，即可完成迁移。

之前使用开源 OpenTelemetry 方案，如何迁移到应用性能监控?

应用性能监控已兼容 OpenTelemetry 协议标准，您只需要修改上报地址，并在 Resource 的参数中填入业务系统的 Token，即可完成迁移。

是否支持通过 OpenTelemetry 协议标准上报指标（Metrics）和日志（Logs）?

OpenTelemetry 协议标准中定义了链路、指标、日志三种遥测数据，APM 支持链路（Traces）数据的上报和存储，并在基于链路数据计算各类指标。
APM 不会直接保存应用上报的指标（Metrics），您可以参考 Prometheus 数据集成以及 Dashboard 关联展示，将应用上报的指标（Metric）保存到 Prometheus 中。
APM 暂不支持接收应用上报的日志（Logs），建议您关闭 Log Exporter，并将应用日志输出到日志服务（CLS）中。您可以参考链路与日志关联分析 配置 APM 与日志服务（CLS）的集成，实现链路与日志关联分析。

接入相关

需要自行下载探针和 SDK 吗?

对于部署在容器服务 TKE 上的 Java、Python、Node.js、.Net 应用，APM 提供自动接入方案，可以在应用部署到 TKE 之后实现探针自动注入，方便应用快速接入。其他情况下，请自行下载探针和 SDK 并安装。

安装探针以后，日志中有报错：Failed to export spans

大概率和 Token 填写有关系，请参考接入文档正确填写 Token。对于 OpenTelemetry 方案，Token 需要和应用名 `service.name`，以及其他实例属性一起填入到 Resource Attributes 中。

通过 TKE 环境自动接入，同一个 TKE 集群的应用可以上报到不同业务系统吗?

可以的。TKE 集群在安装 tencent-opentelemetry-operator 的时候，需要指定默认业务系统。在应用工作负载中，可以通过添加 `cloud.tencent.com/apm-token` 这个 annotation 上报到特定的业务系统。如果应用工作负载不设置 `cloud.tencent.com/apm-token`，则上报到默认业务系统。

通过 TKE 环境自动接入，同一个 TKE 集群的应用可以上报到不同地域吗?

不可以。TKE 集群在安装 tencent-opentelemetry-operator 的时候，需要指定接入点。应用级别不能指定接入点。

APM 控制台安装 tencent-opentelemetry-operator 失败

Operator 的安装与更新由 TKE 应用市场来承担，缺少权限、资源不足、集群故障、集群版本不满足要求等情况都有可能导致安装失败，请前往 TKE 应用 页面检查安装失败的原因。

APM 控制台更新 tencent-opentelemetry-operator 失败

请前往 TKE 应用中心检查更新失败的原因，可以尝试取消异常的更新操作，并通过 APM 控制台 再次更新。

tencent-opentelemetry-operator 安装成功，但应用还是接入失败

请确保语言以及框架的版本符合要求。此外，请确保 annotation 添加在工作负载的 `spec.template.metadata.annotations` 中，而不是 `metadata.annotations` 。

更新 tencent-opentelemetry-operator 会影响已经接入的应用吗?

不会。

tencent-opentelemetry-operator 更新之后，无法接入应用

用户提交更新操作后，Operator 的更新需要几分钟的时间，在这个过程中，新启动的应用实例是无法接入的。因为 Operator 在更新没有完成的情况下，无法通过捕捉应用 Pod 的创建事件注入探针。您只需要等待 Operator 更新完成之后，重新创建应用 Pod，就可以接入应用。

可以跨地域接入应用吗?

可以，只要网络是连通的，就能实现跨地域接入。可以选择公网接入，或引入申请私有连接（Private Link）来打通 VPC 网络。

通过 TKE 环境自动接入，可以跨地域吗?

可以。在 APM 控制台 安装 Operator 的时候，指定正确的上报地域即可。

可以自定义应用实例名吗?

在接入的时候，通过设置 Resource 的 `host.name` 属性，就能自定义应用实例名。在大多数场景下，IP 地址都可以作为应用实例名，但如果系统中的 IP 地址存在重复的情况，就需要使用其他唯一标识定义实例名称，例如 主机 `IP` + 容器名 的方式。

通过 OpenTelemetry-Python 接入 Python 应用，在控制台看到的实例名不是 IP 地址

OpenTelemetry-Python 方案没有自动获取 IP 地址作为实例名，请通过 Resource 的 `host.name` 属性进行主动设置。

Java 应用可以直接使用 OpenTelemetry 社区的探针吗?

可以，但不推荐。应用性能监控 APM 提供腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-OTel Java Agent），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。推荐使用腾讯云增强版 OpenTelemetry Java 探针，如果直接使用 OpenTelemetry 社区的探针，会存在一定的功能缺失。

控制台功能相关

什么是业务系统?

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。

如何划分业务系统?

由于不同业务系统之间的监控数据完全隔离，可以基于隔离原则划分业务系统。如果两组应用之间不可能存在相互调用关系，可以把它们划分到2个业务系统中进行管理。按不同的环境划分业务系统是一种典型的使用场景，可以为开发环境、测试环境、生产环境各创建一个业务系统。另外一种典型的使用场景是按业务域划分业务系统，前提是这些业务域之间不存在相互调用。

如何理解应用?

在应用性能监控 APM 中，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。

可以在多个业务系统中使用同一个应用名吗?

可以。

可以通过日志关联功能将日志输出到 CLS 吗?

日志关联功能实现了日志与链路的关联查询，方便用户定位问题。用户需要自行接入腾讯云日志服务 CLS，并在日志体中输出 `trace_id` 字段。

接入了 Go 应用，为什么无法使用应用诊断功能?

应用诊断功能是腾讯云增强版 OpenTelemetry Java 探针（TencentCloud-OTel Java Agent）的增强能力，目前只能用于 Java 应用。

在日志中打印 TraceId，需要引入新的依赖吗?

如果使用腾讯云增强版 OpenTelemetry Java 探针接入，不需要引入任何新的依赖，只需要修改日志配置文件中的 pattern，就能实现日志中注入 TraceId 和 SpanId。具体请参见 OpenTelementry 增强版 Java 探针 。

数据相关

P99耗时为何不准确?

应用性能监控 APM 基于线性分布假设算法计算分位数，其结果是预估值，并不能保持精确；而最大耗时一定是精确的。此算法与 Prometheus 计算分位数采用的算法是一致的，是业界普遍采用的算法，具体请参见 Prometheus 分位数误差 。当样本数比较少时，可能会出现 P99耗时统计与实际情况出现偏差的情况。

为什么只有链路数据，没有指标数据?

可能和上报的 Span 数据中缺少 Span 类型有关。在自定义埋点的情况下，请确保上报的 Span 都设置了 `span.kind` ，且取值为 `Client` 、 `Server` 、 `Consumer` 、 `Producer` 、 `Internal` 之一。

为什么会出现断链的情况?

如下几种情况会导致调用链中断：

1. 腾讯云增强版 OpenTelemetry Java 探针实现了熔断保护机制，在应用负载特别高的时候，临时关闭了数据上报。关于熔断保护机制的详情，请参见 腾讯云增强版 OpenTelemetry 探针熔断保护机制 。
2. 腾讯云增强版 OpenTelemetry Java 探针限制了每秒最多上报5000条 Span，超过的部分会被丢弃。
3. 探针或 SDK 的自动埋点机制没有覆盖到相关的框架或组件，需要通过自定义埋点进行增强。

什么是 Apdex？

Apdex 全称是 Application Performance Index，是由 Apdex 联盟开发的用于评估应用性能的工业标准。Apdex 标准从用户的角度出发，对应用响应时间的表现，转为可量化范围为0 – 1的满意度评价。

Apdex 的计算规则是什么?

首先根据应用性能评估确定应用响应时间的最低门槛为 Apdex 阈值，然后根据实际响应时间获得3种不同的性能表现：

- **满意（Satisfied）**：应用响应时间低于或等于 Apdex 阈值。
- **可容忍（Tolerating）**：应用响应时间大于 Apdex 阈值，但同时小于或等于4倍的 Apdex 阈值。
- **沮丧（Frustrated）**：应用响应时间大于4倍的 Apdex 阈值。

Apdex =（满意数+可容忍数/2）/ 总样本量

应用性能监控支持链路采样吗?

支持。应用性能监控提供了完整的尾部采样实现，可以帮助您降低产品使用成本。详情请参见 通过采样策略降低 APM 使用成本 。

在数据库调用分析中，为什么 SQL 会被截断?

在数据库调用分析和 SQL 分析视图中，SQL 语句以维度信息的方式保存在 APM 的指标数据中，以实现基于 SQL 语言的统计分析。为了避免过长的维度信息导致指标聚合结果难以理解，APM 将维度信息的值限制在256个字符，因此 SQL 语句中超出256个字符的部分将会被截断。
SQL 截断仅存在于 APM 的指标数据中，其完整内容依然保存在每次调用对应的 Span 中。您可以通过关联查询找到对应的链路以及 Span，在**链路详情**视图中获取完整的 SQL 语句。

在接口分析中，为什么接口名称中部分内容会被替换为{LONG_NUM}等标识?

当一个应用中的接口名称具有比较高的基数（Cardinality）时，就会发生维度发散问题，严重影响用户体验。APM 引入了多项预置的收敛规则，将同类型的接口进行合并，以降低接口名称的基数，以提升统计分析的数据价值以及用户体验。关于接口名称自动收敛的更多详情，请参见 关于接口名称自动收敛 。
在数据库调用分析等功能中，APM 使用类似的技术对 SQL 语句进行了自动收敛。自动收敛仅影响指标数据，接口名称和 SQL 语句的原始值依然保存在 Span 数据中，您可以通过关联查询找到对应的调用链，获取原始的接口名以及 SQL 语句。

为什么应用的 CPU 利用率和 top 命令所看到的进程 CPU 利用率不一样?

APM 获取的 CPU 利用率来源于探针的持续上报，在观察视角、数据精度上，都和操作系统中直接获取的 CPU 利用率存在差异。以腾讯云增强版 Java 探针为例，CPU 利用率的计算公式为：**JVM 使用的 CPU 时间片段 / 时间周期 / CPU 核数**。其中，时间周期为1分钟，JVM 使用的 CPU 时间片段取自 JMX 的 `OperatingSystemMXBean.getProcessCpuTime()` 。在实际使用中，我们可以忽略两者的差异，更多的关注数据的相对变化。

费用相关

应用性能监控如何收费?

应用性能监控提供多种计费方式，请参见 应用性能计费概述 。

应用性能监控可以免费试用吗?

可以。新用户有15天试用期，15天内上报限额为1亿 Span，链路存储时长为7天。

什么情况下适合套餐包（预付费）计费模式?

需要满足如下条件：
1. 套餐包模式下，链路将按10%比例进行采样，保留全部异常链路，而且不会影响指标的准确性。
2. 用户对 Agent 的数量有明确的预期。
3. Agent 的平均数据上报量比较大，导致按量付费模式成本比较高。

套餐包模式中的 Agent 怎么理解?

一个 Agent 对应一个接入 APM 的应用进程。

Agent * Hour 是什么意思?

Agent * Hour 是套餐包模式的计费单位。每个应用进程接入 APM 后，每个小时将消耗1个 Agent * Hour，举例：10个应用进程接入 APM，每天将消耗10*24=240个 Agent * Hour。

套餐包可以叠加购买吗?

可以，确保在有效期内用完就行。

链路数据存储多久?

试用期默认存储7天，正式计费后您可以按需选择30天以内的任意时长。

指标数据存储多久?

30天。