应用性能监控

接入指南

腾讯云

文档目录

接入指南
应用接入概述

最近更新时间：2024-10-16 16:35:41

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-Go 接入 Go 应用（推荐）

最近更新时间：2025-02-28 15:16:42

> **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 。

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 **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。

接入 Go 应用

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

```
package main

import (
    "context"
    "errors"
    "time"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/exporters/stdout/stdoutmetric"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/metric"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    "log"
)

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(prop
agation.TraceContext{}, propagation.Baggage{}))
    return tp, nil
}


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


    meterProvider := metric.NewMeterProvider(
        metric.WithReader(metric.NewPeriodicReader(metricExporter,
            // Default is 1m. Set to 3s for demonstrative purposes.
            metric.WithInterval(3*time.Second))),
    )
    return meterProvider, nil
}
```

对应的字段说明如下，请根据实际情况进行替换。

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

步骤2：SDK 初始化，启动 HTTP 服务

```go
package main

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

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

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

    // 初始化 SDK
    otelShutdown, err := setupOTelSDK(ctx)
    if err != nil {
        return
    }
    // 优雅关闭
    defer func() {
        err = errors.Join(err, otelShutdown(context.Background()))
    }()

    // 启动HTTP服务
    srv := &http.Server{
        Addr:         ":8080",
        BaseContext:  func(_ net.Listener) context.Context { return ctx },
        ReadTimeout:  time.Second,
        WriteTimeout: 10 * time.Second,
        Handler:      newHTTPHandler(),
    }
```

```
    srvErr := make(chan error, 1)
    go func() {
        srvErr <- srv.ListenAndServe()
    }()

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

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

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

步骤3： 对 HTTP 接口进行埋点增强

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

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

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

    // 对所有接口进行埋点增强
    handler := otelhttp.NewHandler(mux, "/")
    return handler
}

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

```
    io.WriteString(w, "ok")
}
```

接入验证

启动 Go 应用后，通过8080端口访问对应的接口，例如 `https://localhost:8080/simple`，应用就会向 APM 上报处理 HTTP 请求相关的链路数据。在有正常流量的情况下，应用性能监控 > 应用列表 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.WithDBNa
me("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{}).Wher
e("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：2024-09-23 09:10:01

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：2025-01-08 14:14:32

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。

接入流程

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 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。

通过 Jaeger 协议上报
通过 Jaeger 原始 SDK 上报

最近更新时间：2024-05-16 11:45:51

本文将为您介绍如何使用 Jaeger 原始 SDK 上报 Go 应用数据。

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 **Jaeger**。
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。

7. 选择 **agent 接入方式**。

步骤2：安装 Jaeger Agent

1. 下载 Jaeger Agent 。
2. 在控制台执行以下命令启动 Agent。

```
nohup ./jaeger-agent --reporter.grpc.host-port={{接入点}} --
agent.tags=token={{token}}
```

> **说明**
> 对于 Jaeger Agent v1.15.0及以下版本，请将启动命令中 `--agent.tags` 替换为 `--jaeger.tags` 。

步骤3：上报数据

通过 Jaeger 原始 SDK 上报数据：

1. 客户端侧由于需要模拟 HTTP 请求，引入 `opentracing-contrib/go-stdlib/nethttp` 依赖。
   - 依赖路径：`github.com/opentracing-contrib/go-stdlib/nethttp`
   - 版本要求：≥ `dv1.0.0`
2. 配置 Jaeger，创建 Trace 对象。示例如下：

```
cfg := &jaegerConfig.Configuration{
  ServiceName: ginClientName, //对其发起请求的的调用链，叫什么服务
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
    Type:  "const",
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}

tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 构建 span 并把 span 放入 conext 中，示例如下：

```
span := tracer.StartSpan("CallDemoServer") //构建span
ctx := opentracing.ContextWithSpan(context.Background(), span) //将
span的引用放入conext中
```

4. 构建带 tracer 的 Request 请求，示例如下：

```
//构建http的请求
req, err := http.NewRequest(
```

```
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
req = req.WithContext(ctx)
//构建带tracer的请求
req, ht := nethttp.TraceRequest(tracer, req)
```

5. 发起 HTTP 请求，并获得返回结果。

```
httpClient := &http.Client{Transport: &nethttp.Transport{}} //初始化
http客户端
res, err := httpClient.Do(req)
//..省略err判断
body, err := ioutil.ReadAll(res.Body)
//..省略err判断
log.Printf(" %s recevice: %s\n", clientServerName, string(body))
```

完整代码如下：

```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "context"
    "fmt"
    "github.com/opentracing-contrib/go-stdlib/nethttp"
    "github.com/opentracing/opentracing-go"
    "github.com/opentracing/opentracing-go/ext"
    opentracingLog "github.com/opentracing/opentracing-go/log"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "io/ioutil"
    "log"
    "net/http"
```

```
)

const (
    // 服务名 服务唯一标示，服务指标聚合过滤依据。
    ginClientName = "demo-gin-client"
    ginPort       = ":8080"
    endPoint      = "xxxxx:6831" // 本地agent地址
    token         = "abc"
)

// StartClient gin client 也是标准的 http client.
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginClientName, //对其发起请求的的调用链，叫什么服务
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信
息，所有字段都是可选的
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }

    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到tracer
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    //构建span，并将span放入context中
    span := tracer.StartSpan("CallDemoServer")
    ctx := opentracing.ContextWithSpan(context.Background(), span)
    defer span.Finish()

    // 构建http请求
    req, err := http.NewRequest(
        http.MethodGet,
```

```
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
    if err != nil {
        HandlerError(span, err)
        return
    }
    // 构建带tracer的请求
    req = req.WithContext(ctx)
    req, ht := nethttp.TraceRequest(tracer, req)
    defer ht.Finish()

    // 初始化http客户端
    httpClient := &http.Client{Transport: &nethttp.Transport{}}
    // 发起请求
    res, err := httpClient.Do(req)
    if err != nil {
        HandlerError(span, err)
        return
    }
    defer res.Body.Close()
    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        HandlerError(span, err)
        return
    }
    log.Printf(" %s recevice: %s\n", ginClientName, string(body))
}

// HandlerError handle error to span.
func HandlerError(span opentracing.Span, err error) {
    span.SetTag(string(ext.Error), true)
    span.LogKV(opentracingLog.Error(err))
}
```

通过 gin Jaeger 中间件上报

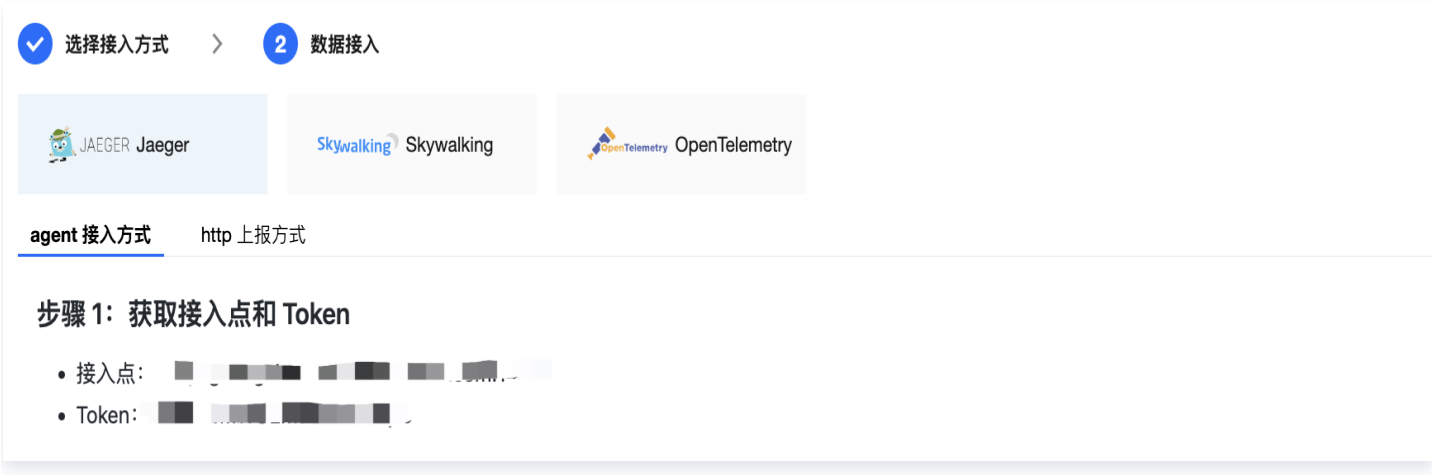
最近更新时间：2024-08-14 15:48:31

本文将为您介绍如何使用 gin Jaeger 中间件上报 Go 应用数据。

操作步骤

步骤1：获取接入点和 Token

进入 应用性能监控控制台 **应用监控 > 应用列表**，单击**接入应用**页面，在接入应用时选择 GO 语言与 gin Jaeger 中间组件的数据采集方式。
在选择接入方式步骤获取您的接入点和 Token，如下图所示：



步骤2：安装 Jaeger Agent

1. 下载 Jaeger Agent 。
2. 执行下列命令启动 Agent 。

```
nohup ./jaeger-agent --reporter.grpc.host-port=
{{collectorRPCHostPort}} --agent.tags=token={{token}}
```

步骤3：选择上报端类型上报应用数据

选择上报端类型，通过 gin Jaeger 中间件上报 Go 应用数据：

服务端

1. 在服务端侧引入 `opentracing-contrib/go-gin` 依赖
   - 依赖路径： `github.com/opentracing-contrib/go-gin`

- 版本要求：≥ `v0.0.0-20201220185307-1dd2273433a4`

2. 配置 Jaeger，创建 Trace 对象。示例如下：

```
cfg := &jaegerConfig.Configuration{
  ServiceName: ginServerName, //对服务发起请求的调用链，填写服务名称
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见4.1.1
    Type:  "const",
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}

tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 配置中间件

```
r := gin.Default()
//传入tracer
r.Use(ginhttp.Middleware(tracer))
```

> **说明**
> 官方默认 OperationName 是 HTTP + HttpMethod，建议使用 `HTTP + HttpMethod + URL` 可以分析到具体接口，接口主要 URL 应是参数名，不是具体参数值。具体用法如下：
> 正确：`/user/{id}`，错误：`/user/1`

```
r.Use(ginhttp.Middleware(tracer, ginhttp.OperationNameFunc(func(r
*http.Request) string {
```

```
        return fmt.Sprintf("testtestheling  HTTP %s %s", r.Method,
r.URL.String())
    })))
```

完整代码如下：

```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "github.com/opentracing-contrib/go-gin/ginhttp"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "net/http"
)

// 服务名 服务唯一标示，服务指标聚合过滤依据。
const ginServerName = "demo-gin-server"

// StartServer
func StartServer() {
    //初始化jaeger，得到tracer
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginServerName, //对服务发起请求的调用链，填写服务名称
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见
4.1.1
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace
信息，所有字段都是可选的
            LogSpans:          true,
            LocalAgentHostPort: endPoint,
        },
```

```
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }

    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    defer closer.Close()
    r := gin.Default()
    //这里说明一下，官方默认 OperationName 是 HTTP + HttpMethod,
    //建议使用 HTTP + HttpMethod + URL 可以分析到具体接口，具体用法如下
    //PS：Restful 接口主要URL应该是参数名，不是具体参数值。 如： 正
确：/user/{id}， 错误：/user/1
    r.Use(ginhttp.Middleware(tracer, ginhttp.OperationNameFunc(func(r
*http.Request) string {
        return fmt.Sprintf("HTTP %s %s", r.Method, r.URL.String())
    })))
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    r.Run() // 监听 0.0.0.0:8080
}
```

客户端

1. 客户端侧由于需要模拟 HTTP 请求，引入 `opentracing-contrib/go-stdlib/nethttp` 依赖。
   - 依赖路径：`github.com/opentracing-contrib/go-stdlib/nethttp`
   - 版本要求：≥ `v1.0.0`
2. 配置 Jaeger，创建 Trace 对象。示例如下：

```
cfg := &jaegerConfig.Configuration{
  ServiceName: ginClientName, //对服务发起请求的的调用链，填写服务名称
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见4.1.1
    Type:  "const",
```

```
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:           true,
    LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}

tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 构建 span 并把 span 放入 conext 中，示例如下：

```
span := tracer.StartSpan("CallDemoServer") //构建span
ctx := opentracing.ContextWithSpan(context.Background(), span) //将
span的引用放入conext中
```

4. 构建带 tracer 的 Request 请求，示例如下：

```
//构建http的请求
req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
req = req.WithContext(ctx)
//构建带tracer的请求
req, ht := nethttp.TraceRequest(tracer, req)
```

5. 发起 HTTP 请求，并获得返回结果。

```
httpClient := &http.Client{Transport: &nethttp.Transport{}} //初始化
http客户端
```

```
    res, err := httpClient.Do(req)
    //..省略err判断
    body, err := ioutil.ReadAll(res.Body)
    //..省略err判断
    log.Printf(" %s recevice: %s\n", clientServerName, string(body))
```

完整代码如下：

```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package gindemo

import (
    "context"
    "fmt"
    "github.com/opentracing-contrib/go-stdlib/nethttp"
    "github.com/opentracing/opentracing-go"
    "github.com/opentracing/opentracing-go/ext"
    opentracingLog "github.com/opentracing/opentracing-go/log"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "io/ioutil"
    "log"
    "net/http"
)

const (
    // 服务名 服务唯一标示，服务指标聚合过滤依据。
    ginClientName = "demo-gin-client"
    ginPort       = ":8080"
    endPoint      = "xxxxx:6831" // 本地agent地址
    token         = "abc"
)

// StartClient 下的 gin client 也是标准的 http client.
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: ginClientName, //对服务发起请求的调用链，填写服务名称
```

```
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见4.1.1
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信
息，所有字段都是可选的
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }

    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到tracer
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    //构建span，并将span放入context中
    span := tracer.StartSpan("CallDemoServer")
    ctx := opentracing.ContextWithSpan(context.Background(), span)
    defer span.Finish()

    // 构建http请求
    req, err := http.NewRequest(
        http.MethodGet,
        fmt.Sprintf("http://localhost%s/ping", ginPort),
        nil,
    )
    if err != nil {
        HandlerError(span, err)
        return
    }
    // 构建带tracer的请求
    req = req.WithContext(ctx)
    req, ht := nethttp.TraceRequest(tracer, req)
    defer ht.Finish()

    // 初始化http客户端
    httpClient := &http.Client{Transport: &nethttp.Transport{}}
```

```
    // 发起请求
    res, err := httpClient.Do(req)
    if err != nil {
        HandlerError(span, err)
        return
    }
    defer res.Body.Close()
    body, err := ioutil.ReadAll(res.Body)
    if err != nil {
        HandlerError(span, err)
        return
    }
    log.Printf(" %s recevice: %s\n", ginClientName, string(body))
}

// HandlerError handle error to span.
func HandlerError(span opentracing.Span, err error) {
    span.SetTag(string(ext.Error), true)
    span.LogKV(opentracingLog.Error(err))
}
```

通过 goredis 中间件上报

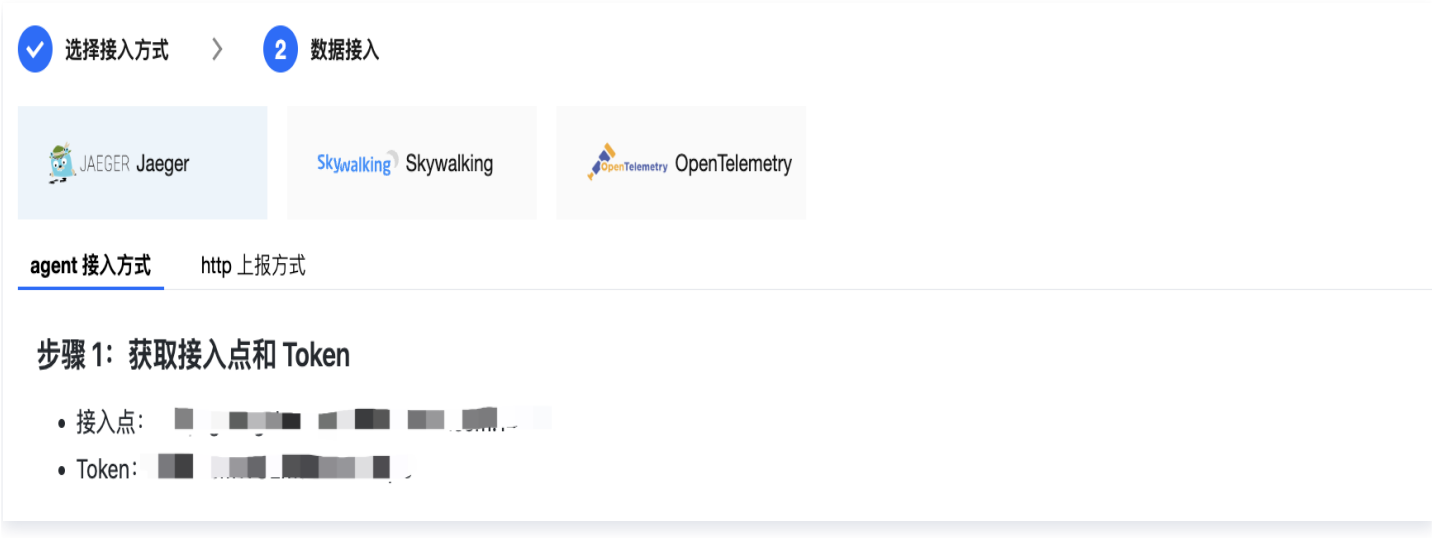
最近更新时间：2024-08-07 19:43:31

本文将为您介绍如何使用 go redis 中间件上报Go应用数据

操作步骤

步骤1：获取接入点和 Token

进入 应用性能监控控制台 > **应用列表**页面，单击**接入应用**，在接入应用时选择 GO 语言与 goredis 中间件的数据采集方式。
在选择接入方式步骤获取您的接入点和 Token，如下图所示：



步骤2：安装 Jaeger Agent

1. 下载 Jaeger Agent 。
2. 执行下列命令启动 Agent 。

```
nohup ./jaeger-agent --reporter.grpc.host-port=
{{collectorRPCHostPort}} --agent.tags=token={{token}}
```

步骤3：选择上报端类型上报应用数据

选择上报端类型，通过 go redis 中间件上报 Go 应用数据：

客户端

1. 引入 `opentracing-contrib/goredis` 埋点依赖。

- 依赖路径：`github.com/opentracing-contrib/goredis`
- 版本要求：≥ `v0.0.0-20190807091203-90a2649c5f87`

2. 配置 Jaeger，创建Trace对象并设置 GlobalTracer。示例如下：

```
cfg := &jaegerConfig.Configuration{
  ServiceName: clientServerName, //对其发起请求的的调用链，叫什么服务
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见4.1.1
    Type:  "const",
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:           true,
        LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}
tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 初始化 Redis 连接，示例如下：

```
func InitRedisConnector() error {
    redisClient = redis.NewUniversalClient(&redis.UniversalOptions{
        Addrs:    []string{redisAddress},
        Password: redisPassword,
        DB:       0,
    })
    if err := redisClient.Ping().Err(); err != nil {
        log.Println("redisClient.Ping() error:", err.Error())
        return err
    }
    return nil
}
```

4. 获取 Redis 连接，示例如下：

```
func GetRedisDBConnector(ctx context.Context) redis.UniversalClient {
    client := apmgoredis.Wrap(redisClient).WithContext(ctx)
    return client
}
```

完整代码如下：

```
package main

import (
    "context"
    "fmt"
    "github.com/go-redis/redis"
    apmgoredis "github.com/opentracing-contrib/goredis"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "log"
    "time"
)

const (
    redisAddress     = "127.0.0.1:6379"
    redisPassword    = ""
    clientServerName = "redis-client-demo"
    testKey          = "redis-demo-key"
    endPoint         = "xxxxx:6831" // HTTP 直接上报地址
    token            = "abc"
)

func main() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: clientServerName, //对其发起请求的的调用链，叫什么服务
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置，详情见
4.1.1
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace
信息，所有字段都是可选的
            LogSpans:           true,
```

```
            LocalAgentHostPort: endPoint,
        },
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }
    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
    opentracing.SetGlobalTracer(tracer)
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    InitRedisConnector()
    redisClient := GetRedisDBConnector(context.Background())
    redisClient.Set(testKey, "redis-client-demo",
time.Duration(1000)*time.Second)
    redisClient.Get(testKey)
}

var (
    redisClient redis.UniversalClient
)

func GetRedisDBConnector(ctx context.Context) redis.UniversalClient {
    client := apmgoredis.Wrap(redisClient).WithContext(ctx)
    return client
}
func InitRedisConnector() error {
    redisClient = redis.NewUniversalClient(&redis.UniversalOptions{
        Addrs:    []string{redisAddress},
        Password: redisPassword,
        DB:       0,
    })
    if err := redisClient.Ping().Err(); err != nil {
        log.Println("redisClient.Ping() error:", err.Error())
        return err
    }
    return nil
}
```

通过 gRPC-Jaeger 拦截器上报

最近更新时间：2024-05-29 18:06:11

本文将为您介绍如何使用 gRPC-Jaeger 拦截器上报 Go 应用数据。

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 **Jaeger**。
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 Jaeger Agent

1. 下载 Jaeger Agent 。
2. 在控制台执行以下命令启动 Agent。

```
nohup ./jaeger-agent --reporter.grpc.host-port=
{{collectorRPCHostPort}} --agent.tags=token={{token}}
```

步骤3：选择上报端类型上报应用数据

选择上报端类型，通过 gRPC-Jaeger 拦截器上报 Go 应用数据：

服务端配置

1. 在服务端侧引入 `opentracing-contrib/go-grpc` 埋点依赖。
- 依赖路径： `github.com/opentracing-contrib/go-grpc`
- 版本要求： ≥ `v0.0.0-20210225150812-73cb765af46e`

2. 配置 Jaeger，创建 Trace 对象。示例如下：

```
cfg := &jaegerConfig.Configuration{
  ServiceName: grpcServerName, //对其发起请求的的调用链，叫什么服务
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
    Type:  "const",
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:          true,
    LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}
tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 配置拦截器。

```
s :=
grpc.NewServer(grpc.UnaryInterceptor(otgrpc.OpenTracingServerIntercept
or(tracer)))
```

4. 启动 Server 服务。

```
// 在gRPC服务器处注册我们的服务
pb.RegisterHelloTraceServer(s, &server{})
if err := s.Serve(lis); err != nil {
   log.Fatalf("failed to serve: %v", err)
}
```

完整代码如下：

```
// Copyright © 2019-2020 Tencent Co., Ltd.
```

```
// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package grpcdemo

import (
    "context"
    "fmt"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "log"
    "net"

    "github.com/opentracing-contrib/go-grpc"
    "google.golang.org/grpc"
)

const (
    // 服务名 服务唯一标示，服务指标聚合过滤依据。
    grpcServerName = "demo-grpc-server"
    serverPort     = ":9090"
)

// server is used to implement proto.HelloTraceServer.
type server struct {
    UnimplementedHelloTraceServer
}

// SayHello implements proto.HelloTraceServer
func (s *server) SayHello(ctx context.Context, in *TraceRequest)
(*TraceResponse, error) {
    log.Printf("Received: %v", in.GetName())
    return &TraceResponse{Message: "Hello " + in.GetName()}, nil
}

// StartServer
func StartServer() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: grpcServerName, //对其发起请求的的调用链，叫什么服务
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
            Type:  "const",
```

```
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信
息，所有字段都是可选的
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }
    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到tracer
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    lis, err := net.Listen("tcp", serverPort)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s :=
grpc.NewServer(grpc.UnaryInterceptor(otgrpc.OpenTracingServerInterceptor
(tracer)))

    // 在gRPC服务器处注册我们的服务
    RegisterHelloTraceServer(s, &server{})
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

客户端配置

1. 客户端侧由于需要模拟 HTTP 请求，引入 `opentracing-contrib/go-stdlib/nethttp` 依赖。
   - 依赖路径：`github.com/opentracing-contrib/go-stdlib/nethttp`
   - 版本要求：≥ `v1.0.0`
2. 配置 Jaeger，创建 Trace 对象。

```
cfg := &jaegerConfig.Configuration{
  ServiceName: grpcClientName, //对其发起请求的的调用链，叫什么服务
  Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
    Type:  "const",
    Param: 1,
  },
  Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace信息，
所有字段都是可选的
    LogSpans:           true,
    LocalAgentHostPort: endPoint,
  },
  //Token配置
  Tags:         []opentracing.Tag{ //设置tag，token等信息可存于此
    opentracing.Tag{Key: "token", Value: token}, //设置token
  },
}
tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

3. 建立连接，配置拦截器。

```
// 向服务端建立连接，配置拦截器
conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(),
grpc.WithBlock(),

grpc.WithUnaryInterceptor(otgrpc.OpenTracingClientInterceptor(tracer))
)
```

4. 进行 gRPC 调用，验证是否接入成功。
   完整代码如下：

```
// Copyright © 2019-2020 Tencent Co., Ltd.

// This file is part of tencent project.
// Do not copy, cite, or distribute without the express
// permission from Cloud Monitor group.

package grpcdemo
```

```
import (
    "context"
    "fmt"
    "github.com/opentracing-contrib/go-grpc"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"
    "google.golang.org/grpc"
    "log"
    "time"
)

const (
    // 服务名 服务唯一标示，服务指标聚合过滤依据。
    grpcClientName = "demo-grpc-client"
    defaultName    = "TAW Tracing"
    serverAddress  = "localhost:9090"
    endPoint       = "xxxxx:6831" // 本地agent地址
    token          = "abc"
)

// StartClient
func StartClient() {
    cfg := &jaegerConfig.Configuration{
        ServiceName: grpcClientName, //对其发起请求的的调用链，叫什么服务
        Sampler: &jaegerConfig.SamplerConfig{ //采样策略的配置
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{ //配置客户端如何上报trace
信息，所有字段都是可选的
            LogSpans:           true,
            LocalAgentHostPort: endPoint,
        },
        //Token配置
        Tags: []opentracing.Tag{ //设置tag，token等信息可存于此
            opentracing.Tag{Key: "token", Value: token}, //设置token
        },
    }
    tracer, closer, err :=
cfg.NewTracer(jaegerConfig.Logger(jaeger.StdLogger)) //根据配置得到
tracer
```

```
    defer closer.Close()
    if err != nil {
        panic(fmt.Sprintf("ERROR: fail init Jaeger: %v\n", err))
    }
    // 向服务端建立链接，配置拦截器
    conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(),
grpc.WithBlock(),

grpc.WithUnaryInterceptor(otgrpc.OpenTracingClientInterceptor(tracer))
)
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()

    //
    c := NewHelloTraceClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(),
time.Second)
    defer cancel()
    // 发起RPC调用
    r, err := c.SayHello(ctx, &TraceRequest{Name: defaultName})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("RPC Client receive: %s", r.GetMessage())
}
```

接入 JAVA 应用
K8s 环境自动接入 Java 应用（推荐）

最近更新时间：2024-10-24 16:18:42

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。

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. 点击**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，点击**更多**，点击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，点击**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，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：2024-11-20 14:51:01

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**，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 **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://pl-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,ho
st.name=10.10.1.1_192.168.1.2
```

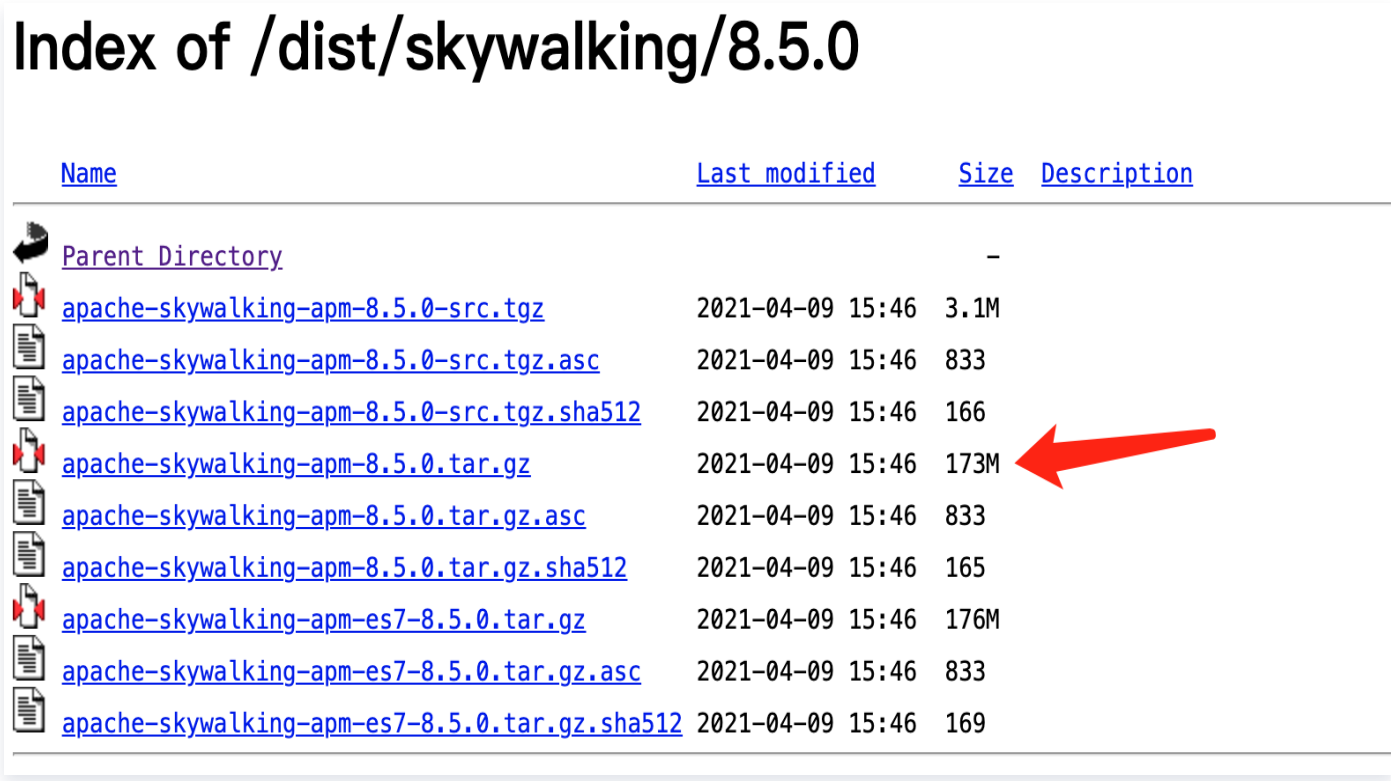
通过 SkyWalking 协议接入 Java 应用

最近更新时间：2024-07-19 15:30:01

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。

接入其他语言编写的应用

最近更新时间：2024-09-23 09:10:01

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。

接入 Python 应用
K8s 环境自动接入 Python 应用（推荐）

最近更新时间：2024-10-24 16:18:42

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 。

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. 点击**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，点击**更多**，点击**编辑yaml**。
4. 在 Pod annotation 中添加如下内容，点击**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，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。

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：

1. 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；
2. 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。
3. 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
```

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：2024-09-23 09:10:01

说明：

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

本文将通过相关操作介绍如何通过社区的 OpenTelemetry-Python 方案接入 Python 应用。OpenTelemetry-Python 方案对于 Python 系的常用依赖库和框架，包括 Flask、Django、FastAPI、MySQL Connector 等，提供了自动埋点，在不需要修改代码的情况下就能实现链路信息的上报。其他支持自动埋点的依赖库和框架请参考 OpenTelemetry 社区提供的 完整列表 。

前提条件

此方案支持 Python 3.6 及以上版本。

示例 Demo

所需依赖如下：

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

示例代码 app.py 通过 Flask 框架提供3个 HTTP 接口，对应的 MySQL 和 Redis 服务请自行搭建，或直接购买云产品。

```
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)
```

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 **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：

```
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，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。

Django 应用注意事项

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

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

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

自定义埋点（可选）

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

```
from opentelemetry import trace
import requests
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
```

通过 Jaeger 协议上报

最近更新时间：2024-08-14 15:48:31

本文将为您介绍如何使用 Jaeger 协议上报 Python 应用数据。

操作步骤

步骤1：获取接入点和 Token

进入 应用性能监控控制台 **应用监控** > **应用列表**页面，单击**接入应用**，在接入应用时选择 Python 语言与 Jaeger 协议的数据采集方式。在选择接入方式步骤获取您的接入点和 Token，如下图所示：

步骤2：安装 Jaeger Agent

1. 下载官方 Jaeger Agent 。
2. 执行下列命令启动 Agent。

```
nohup ./jaeger-agent --reporter.grpc.host-port={{接入点}} --
jaeger.tags=token={{token}}
```

步骤3：通过 Jaeger 上报数据

1. 执行下列命令安装 jaeger_client 包。

```
pip install jaeger_client
```

2. 创建如下 Python 文件和 Tracer 对象，跟踪所有的 Request。

```
from jaeger_client import Config
import time
from os import getenv

# 配置jaeger代理的地址，默认本机localhost
JAEGER_HOST = getenv('JAEGER_HOST', 'localhost')
SERVICE_NAME = getenv('JAEGER_HOST', 'my_service_test')


def build_your_span(tracer):
    with tracer.start_span('yourTestSpan') as span:
        span.log_kv({'event': 'test your message', 'life': 42})
        span.set_tag("span.kind", "server")
        return span


def build_your_tracer():
    my_config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'local_agent': {
                'reporting_host': JAEGER_HOST,
                'reporting_port': 6831,
            },
            'logging': True,
        },
        service_name=SERVICE_NAME,
        validate=True
    )
    tracer = my_config.initialize_tracer()

    return tracer


if __name__ == "__main__":
    tracer = build_your_tracer()
    span = build_your_span(tracer)
    time.sleep(2)
    tracer.close()
```

说明：

目前 Jaeger 支持 Flask、Django 和 Grpc 等框架进行上报，更多请参见：

- jaeger-client-python
- OpenTracing API Contributions

通过 SkyWalking 协议接入 Python 应用

最近更新时间：2024-09-23 09:10:01

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="xxxxxxxxxxxxxx", # 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：2024-09-23 09:10:01

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();
```

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 **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，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，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。

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：2024-12-10 14:54:42

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：2024-09-23 09:10:01

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");
});
```

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 **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，完整的启动命令为：

```
export OTEL_TRACES_EXPORTER="otlp"
export OTEL_RESOURCE_ATTRIBUTES='token=myToken,hostName=192.168.0.10'
export OTEL_EXPORTER_OTLP_PROTOCOL='grpc'
export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://pl-demo.ap-
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，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" });
})
```

通过 Jaeger 原始 SDK 上报

最近更新时间：2024-08-14 15:48:31

本文将为您介绍如何使用 Jaeger 原始 SDK 上报 Node.js 应用数据。

操作步骤

步骤1：获取接入点和 Token

进入 应用性能监控控制台，进入**应用监控 > 应用列表**页面，单击**接入应用**，在接入应用时选择 Node.js 语言与 Jaeger 的数据采集方式。在选择接入方式步骤获取您的接入点和 Token，如下图所示：

步骤2：安装依赖

使用 npm 安装依赖

```
$ npm i jaeger-client
```

步骤3：引入 SDK 并且进行数据上报

1. 引入SDK，示例如下：

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

// jaeger 配置
const config = {
 serviceName: 'service-name', // 服务名称，根据业务自行修改
 sampler: {
     type: 'const',
     param: 1,
```

```
  },
  reporter: {
      logSpans: true,
      collectorEndpoint: 'http://ap-
guangzhou.apm.tencentcs.com:14268/api/traces', // 接入点，此前在应用性能监
控获取的接入点多了 api/traces
  },
};

const options = {
 tags: {
      token: 'Vds************CrKck' // 业务申请的 token
  },
};
```

说明：
Node 使用 API 直接进行数据上报，因此不需要启动 Jaeger agent。接入点选择自己对应的网络环境，并且在后面加入 `/api/traces` 后缀即可。

2. 进行数据上报 ，示例如下：

```
// 初始化 tracer 实例对象
const tracer = initTracer(config, options);

// 初始化 span 实例对象
const span = tracer.startSpan('spanStart');

// 当前服务为 server
span.setTag('span.kind', 'server');

// 设置标签（可选，支持多个）
span.setTag('tagName', 'tagValue');

// 设置事件（可选，支持多个）
span.log({ event: 'timestamp', value: Date.now() });

// 标记Span结束
span.finish();
```

接入 .NET 应用
通过 OpenTelemetry-dotnet 接入 .NET 应用（推荐）

最近更新时间：2024-09-23 09:10:01

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。

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();
```

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 > 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 **.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，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.ManualInstrumentatio
ns.Registered
```

手动接入方案

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

步骤一：安装手动接入所需的 OpenTelemetry 相关依赖

1. 安装手动接入所需的 OpenTelemetry 相关依赖。

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

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

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

步骤二：初始化 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();
```

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

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

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

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

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

步骤四：运行程序

构建并运行应用。

```
dotnet build
```

```
dotnet run
```

接入验证

运行 .NET 应用，在有正常流量的情况下，应用性能监控 > 应用列表 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。

K8s 环境自动接入 .NET 应用（推荐）

最近更新时间：2024-10-24 16:18:42

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. 点击**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，点击**更多**，点击**编辑 yaml**。
4. 在 Pod annotation 中添加如下内容，点击**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，进入对应的应用 > **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。

接入 Nginx

最近更新时间：2025-02-13 15:00:02

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）。

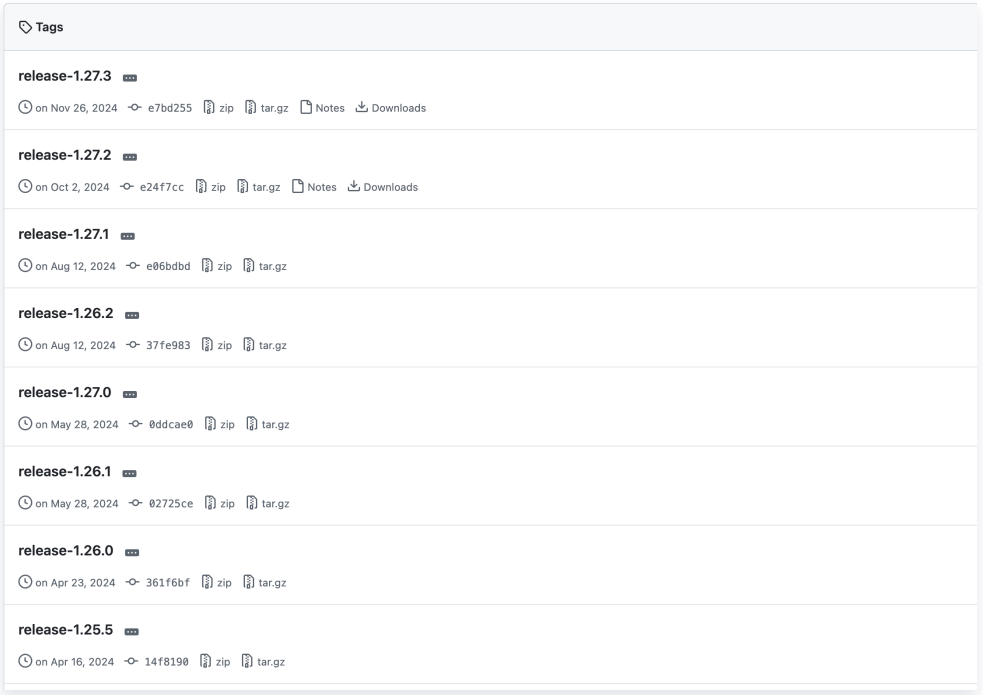
```
nginx -V
```

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。

接入其他语言编写的应用

最近更新时间：2025-02-13 15:00:02

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 15:16:42

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 09:22:22

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。