ALIBABA CLOUD

阿里云

链路追踪

准备工作

文档版本：20200911

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.准备工作概述

在使用链路追踪控制台之前，需要完成的准备工作包括开通相关服务和授权，以及接入应用。

开通相关服务和授权

使用链路追踪之前，首先需要开通链路追踪服务。由于链路追踪依赖日志服务 LOG 和访问控制 RAM 服务，所以也需要开通这两项服务，并授权链路追踪读写您的日志服务数据。

关于开通相关服务和授权的方法，请参见开通相关服务并授权。

接入应用

接入应用是指通过客户端将应用的链路数据上报至链路追踪控制台，从而实现链路追踪的目的。您可以将 Java、PHP、Go、Python、JS、.NET、C++ 等语言的应用数据上报至链路追踪控制台。支持的上报数据客户端包括 Jaeger、Zipkin 和 Skywalking。请根据您的应用语言或者使用的客户端查看相应的应用接入文档。

按应用语言

开始监控Java应用

- 通过Jaeger上报Java应用数据
- 通过Zipkin上报Java应用数据
- 通过SkyWalking上报Java应用数据

开始监控PHP应用

- 通过Zipkin上报PHP应用数据

开始监控Go应用

- 通过Jaeger上报Go应用数据
- 通过Zipkin上报Go应用数据

开始监控Python应用

- 通过Jaeger上报Python应用数据

开始监控Node.js应用

- 通过Jaeger上报Node.js应用数据

开始监控.NET应用

- 通过Jaeger上报 .NET应用数据
- 通过Zipkin上报 .NET应用数据

开始监控C++ 应用

- 通过Jaeger上报C++ 应用数据

按客户端

Jaeger

Java应用 Go应用 Python应用 Node.js应用 .NET应用 C++ 应用

Zipkin

Java应用 PHP应用 Go应用 .NET应用

SkyWalking

Java应用

2.开通相关服务并授权

使用链路追踪之前，首先需要开通链路追踪服务。由于链路追踪依赖日志服务 LOG 和访问控制 RAM 服务，所以也需要开通这两项服务，并授权链路追踪读写您的日志服务数据。本文介绍开通和授权流程。

前提条件

您已注册阿里云账号并完成实名认证。

开通链路追踪服务

1. 打开链路追踪产品主页，在页面右上角单击登录。
2. 在页面上输入您的阿里云账号和密码，并单击登录。
3. 在产品主页上单击**立即开通**，然后在**云产品开通页**页面上选择**我已阅读并同意《链路追踪服务协议》**，并单击**立即开通**。

> 注意 开通后，如果不使用则不会产生费用，仅当上报数据后才会产生费用。

开通日志服务LOG

请使用您的阿里云账号按照以下步骤开通日志服务LOG。

1. 打开日志服务LOG产品主页，在页面右上角单击登录。
2. 在页面上输入您的阿里云账号和密码，并单击登录。
3. 在产品主页上单击**管理控制台**，然后在新页面上单击**创建简单日志服务**。
4. 在**云产品开通页**页面上选择**我已阅读并同意《日志服务服务协议》**，并单击**立即开通**。

开通访问控制RAM

请使用您的阿里云账号按照以下步骤开通访问控制RAM。

1. 打开访问控制RAM产品主页，在页面右上角单击登录。
2. 在页面上输入您的阿里云账号和密码，并单击登录。
3. 在产品主页上单击立即开通，然后在访问控制开通页面上选择我已阅读并同意访问控制开通协议，并单击立即开通。

授权链路追踪读写您的日志服务数据

1. 登录链路追踪控制台。
2. 在概览页面上，单击授权，授权链路追踪读写您的日志服务。
3. 在云资源访问授权页面上，选择所需的权限，并单击同意授权。

后续步骤

开通所有相关服务并完成必要授权后，请参见以下文档提供的链接开始用链路追踪监控您的应用。

相关文档

- 准备工作概述

3.开始监控 Java 应用

3.1. 通过Jaeger上报Java应用数据

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

前提条件

获取接入点信息 ❯

背景信息

Jaeger是一款开源分布式追踪系统，兼容OpenTracing API，且已加入CNCF开源组织。其主要功能是聚合来自各个异构系统的实时监控数据。目前OpenTracing社区已有许多组件可支持各种Java框架，例如：

- Apache HttpClient
- Elasticsearch
- JDBC
- Kafka
- Memcached
- Mongo
- OkHttp
- Redis
- Spring Boot
- Spring Cloud

要通过Jaeger将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。您可以手动埋点，也可以利用各种现有插件实现埋点的目的。本文介绍以下三种埋点方法。

- 手动埋点
- 通过Spring Cloud组件埋点
- 通过gRPC组件埋点

数据是如何上报的？ ❯

为Java应用手动埋点

要通过Jaeger将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。本示例为手动埋点。

1. 下载Demo工程，进入manualDemo目录，并按照Readme的说明运行程序。
2. 打开pom.xml，添加对Jaeger客户端的依赖。

```
<dependency>
  <groupId>io.jaegertracing</groupId>
  <artifactId>jaeger-client</artifactId>
  <version>0.31.0</version>
</dependency>
```

3. 配置初始化参数并创建Tracer对象。

Tracer对象可以用来创建Span对象以便记录分布式操作时间、通过Extract/Inject方法跨机器透传数据、或设置当前Span。Tracer对象还配置了上报数据的网关地址、本机IP地址、采样率、服务名等数据。您可以通过调整采样率来减少因上报数据产生的开销。

> **说明** 请将 <endpoint> 替换成链路追踪控制台集群设置页面，https://yuque.antfin-inc.com/aliwareid/wiki/itb0o9上相应客户端和地域的接入点。获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
// 将manualDemo替换为您的应用名称。
io.jaegertracing.Configuration config = new io.jaegertracing.Configuration("manualDemo");
io.jaegertracing.Configuration.SenderConfiguration sender = new io.jaegertracing.Configuration.SenderConfiguration();
// 将 <endpoint> 替换为控制台概览页面上相应客户端和地域的接入点。
sender.withEndpoint("<endpoint>");
config.withSampler(new io.jaegertracing.Configuration.SamplerConfiguration().withType("const").withParam(1));
config.withReporter(new io.jaegertracing.Configuration.ReporterConfiguration().withSender(sender).withMaxQueueSize(10000));
GlobalTracer.register(config.getTracer());
```

4. 记录请求数据。

```
Tracer tracer = GlobalTracer.get();
// 创建Span。
Span span = tracer.buildSpan("parentSpan").withTag("myTag", "spanFirst").start();
tracer.scopeManager().activate(span, false);
tracer.activeSpan().setTag("methodName", "testTracing");
// 业务逻辑secondBiz();
span.finish();
```

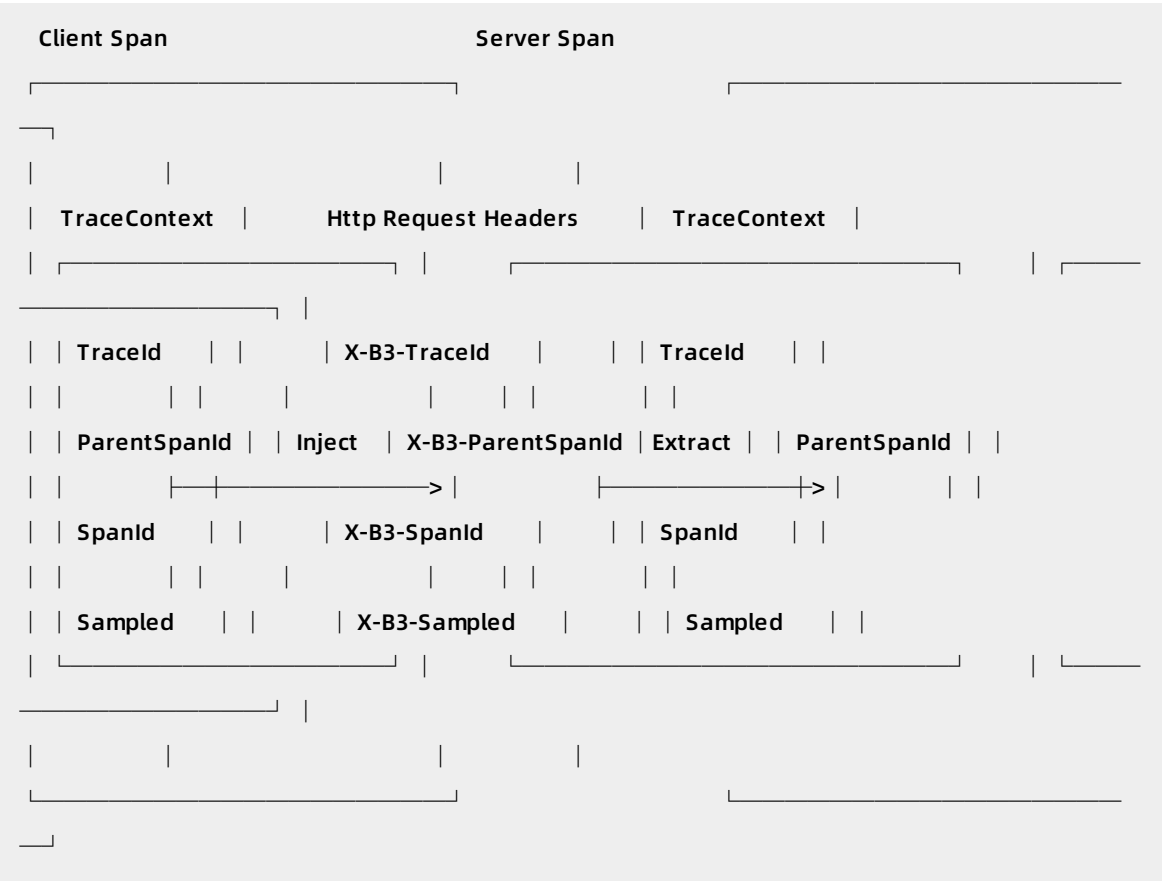
5. （可选）上一步用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要传入上下文。

```
Tracer tracer = GlobalTracer.get();
Span parentspan = tracer.activeSpan();
Tracer.SpanBuilder spanBuilder = tracer.buildSpan("childSpan").withTag("myTag", "spanSecond");
if (parentspan !=null) {
   spanBuilder.asChildOf(parentspan).start();
}
Span childSpan = spanBuilder.start();
tracer.scopeManager().activate(childSpan, false);
// 业务逻辑childSpan.finish();
tracer.activeSpan().setTag("methodName", "testCall");
```

6. （可选）为了方便排查问题，您可以为某个记录添加一些自定义标签（Tag），例如记录是否发生错误、请求的返回值等。

```
tracer.activeSpan().setTag("methodName", "testCall");
```

7. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。总体流程如下：



i. 在客户端调用Inject方法传入Context信息。

```
private void attachTraceInfo(Tracer tracer, Span span, final Request request) {
    tracer.inject(span.context(), Format.Builtin.TEXT_MAP, new TextMap() {
      @Override
      public void put(String key, String value) {
        request.setHeader(key, value);
      }
      @Override
      public Iterator<Map.Entry<String, String>> iterator() {
        throw new UnsupportedOperationException("TextMapInjectAdapter should only be
used with Tracer.inject()");
      }
    });
  }
```

ii. 在服务端调用Extract方法解析Context信息。

```
protected Span extractTraceInfo(Request request, Tracer tracer) {
  Tracer.SpanBuilder spanBuilder = tracer.buildSpan("/api/xtrace/test03");
  try {
    SpanContext spanContext = tracer.extract(Format.Builtin.TEXT_MAP, new TextMapExtrac
tAdapter(request.getAttachments()));
    if (spanContext != null) {
      spanBuilder.asChildOf(spanContext);
    }
  } catch (Exception e) {
    spanBuilder.withTag("Error", "extract from request fail, error msg:" + e.getMessage());
  }
  return spanBuilder.start();
}
```

通过Spring Cloud组件为Java应用埋点

要通过Jaeger将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。本示例为通过Spring Cloud组件埋点。Spring Cloud提供了下列组件的埋点。

- @Async, @Scheduled, Executors
- Feign, HystrixFeign
- Hystrix
- JDBC
- JMS
- Mongo

- RabbitMQ
- Redis
- RxJava
- Spring Messaging - 链路消息通过消息通道发送
- Spring Web (RestControllers, RestTemplates, WebAsyncTask)
- Standard Logging - 日志被添加至当前Span
- WebSocket STOMP
- Zuul

请按照以下步骤通过Spring Cloud组件埋点。

> 说明 请下载Demo工程，进入springMvcDemo/webmvc4-boot目录，并按照Readme的说明运行程序。

1. 打开*pom.xml*，添加Jar包依赖。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
 <artifactId>opentracing-spring-cloud-starter</artifactId>
 <version>0.2.0</version>
</dependency>
<dependency>
 <groupId>io.jaegertracing</groupId>
 <artifactId>jaeger-client</artifactId>
 <version>0.31.0</version>
</dependency>
```

2. 添加OpenTracing Tracer Bean。

> 说明 请将 `<endpoint>` 替换成链路追踪控制台集群设置页面，https://yuque.antfin-inc.com/aliwareid/wiki/itb0o9上相应客户端和地域的接入点。获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
@Bean
public io.opentracing.Tracer tracer() {
   io.jaegertracing.Configuration config = new io.jaegertracing.Configuration("springFrontend");
   io.jaegertracing.Configuration.SenderConfiguration sender = new io.jaegertracing.Configuratio
n.SenderConfiguration();
   sender.withEndpoint("<endpoint>");
   config.withSampler(new io.jaegertracing.Configuration.SamplerConfiguration().withType("const
").withParam(1));
   config.withReporter(new io.jaegertracing.Configuration.ReporterConfiguration().withSender(se
nder).withMaxQueueSize(10000));
   return config.getTracer();
}
```

通过gRPC组件为Java应用埋点

要通过Jaeger将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。本示例为通过gRPC组件埋点。

请按照以下步骤通过gRPC组件埋点。

> 说明 请下载Demo工程，进入grpcDemo目录，并按照Readme的说明运行程序。

1. 打开*pom.xml*，添加Jar包依赖。

```
<dependency>
   <groupId>io.opentracing.contrib</groupId>
   <artifactId>opentracing-grpc</artifactId>
   <version>0.0.7</version>
</dependency>
```

2. 在服务端初始化Tracing对象，创建ServerTracingInterceptor，并添加对Server的拦截。

```
import io.opentracing.Tracer;

public class YourServer {

    private int port;
    private Server server;
    private final Tracer tracer;

    private void start() throws IOException {
        ServerTracingInterceptor tracingInterceptor = new ServerTracingInterceptor(this.tracer);

        // If GlobalTracer is used:
        // ServerTracingInterceptor tracingInterceptor = new ServerTracingInterceptor();

        server = ServerBuilder.forPort(port)
            .addService(tracingInterceptor.intercept(someServiceDef))
            .build()
            .start();
    }
}
```

3. 在客户端初始化Tracing对象，创建ClientTracingInterceptor，并添加对Client Channel的拦截。

```
import io.opentracing.Tracer;

  public class YourClient {

    private final ManagedChannel channel;
    private final GreeterGrpc.GreeterBlockingStub blockingStub;
    private final Tracer tracer;

    public YourClient(String host, int port) {

      channel = ManagedChannelBuilder.forAddress(host, port)
        .usePlaintext(true)
        .build();

      ClientTracingInterceptor tracingInterceptor = new ClientTracingInterceptor(this.tracer);

      // If GlobalTracer is used:
      // ClientTracingInterceptor tracingInterceptor = new ClientTracingInterceptor();

      blockingStub = GreeterGrpc.newBlockingStub(tracingInterceptor.intercept(channel));
    }
  }
```

常见问题

问：Demo程序执行成功，为什么控制台上没有上报数据？

答：请调试方法io.jaegertracing.thrift.internal.senders.HttpSender.send(Process process, List<Span> spans)，并查看上报数据时的返回值。如果报403错误，则表明接入点配置不正确，请检查并修改。

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- 通过Zipkin上报Java应用数据
- Jaeger官网
- Jaeger的Java Client
- OpenTracing的组件库
- OpenTracing的Sping Cloud组件

3.2. 通过Zipkin上报Java应用数据

Zipkin是一款开源的分布式实时数据追踪系统（Distributed Tracking System），由Twitter公司开发和贡献。其主要功能是聚合来自各个异构系统的实时监控数据。在链路追踪Tracing Analysis中，您可以通过Zipkin上报Java应用数据。

前提条件

获取接入点信息 ❯

背景信息

Zipkin已经开发多年，对各种框架的支持比较齐全，例如以下Java框架。

- Apache HttpClient
- Dubbo
- gRPC
- JAX-RS 2.X
- Jersey Server
- JMS (Java Message Service)
- Kafka
- MySQL
- Netty
- OkHttp
- Servlet
- Spark
- Spring Boot
- Spring MVC

要通过Zipkin将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。您可以手动埋点，也可以利用各种现有插件实现埋点的目的。

数据是如何上报的？ ❯

手动埋点

如果选择手动埋点，您就需要自行编写代码。

> 说明 如需获取Demo，请单击下载源码，进入manualDemo目录，并根据Readme运行程序。

1. 添加依赖Jar包。

```
<dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave</artifactId>
        <version>5.4.2</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-sender-okhttp3</artifactId>
        <version>2.7.9</version>
    </dependency>
```

```
</dependency>
```

2. 创建Tracer。

> 说明

```
private static final String zipkinEndPoint = "<endpoint>";
 ...
 // 构建数据发送对象。
 OkHttpSender sender = OkHttpSender.newBuilder().endpoint(zipkinEndPoint).build();

 // 构建数据上报对象。
 Reporter<Span> reporter = AsyncReporter.builder(sender).build();

 tracing = Tracing.newBuilder().localServiceName(localServiceName).spanReporter(reporter).buil
d();
```

3. 构建Span和Child Span。

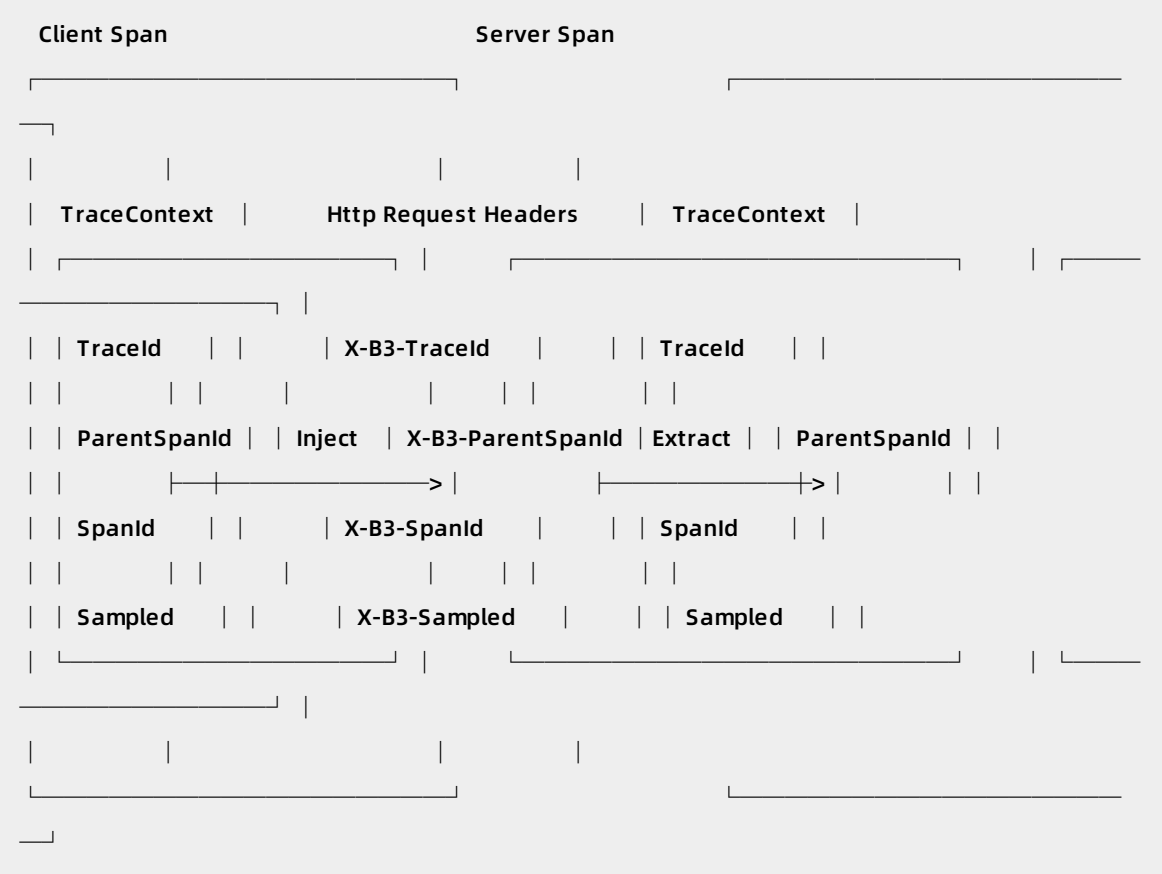
```
private void firstBiz() {
      // 创建rootspan。
      tracing.tracer().startScopedSpan("parentSpan");
      Span span =  tracing.tracer().currentSpan();
      span.tag("key", "firstBiz");
      secondBiz();
      span.finish();
   }

   private void secondBiz() {
      tracing.tracer().startScopedSpanWithParent("childSpan", tracing.tracer().currentSpan().conte
xt());
      Span chindSpan =  tracing.tracer().currentSpan();
      chindSpan.tag("key", "secondBiz");
      chindSpan.finish();
      System.out.println("end tracing,id:" + chindSpan.context().traceIdString());
   }
```

4. （可选）为了快速排查问题，您可以为某个记录添加一些自定义标签，例如记录是否发生错误、请求的返回值等。

```
tracer.activeSpan().setTag("http.status_code", "500");
```

5. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。总体流程如下：



i. 在客户端调用Inject方法传入Context信息。

```
// start a new span representing a client request
   oneWaySend = tracer.nextSpan().name(service + "/" + method).kind(CLIENT);
   --snip--

   // Add the trace context to the request, so it can bepropagated in-band
   tracing.propagation().injector(Request::addHeader)
              .inject(oneWaySend.context(), request);

  // fire off the request asynchronously, totally dropping any response
  request.execute();

  // start the client side and flush instead of finish
  oneWaySend.start().flush();
```

ii. 在服务端调用Extract方法解析Context信息。

```
// pull the context out of the incoming request
extractor = tracing.propagation().extractor(Request::getHeader);

// convert that context to a span which you can name and add tags to
oneWayReceive = nextSpan(tracer, extractor.extract(request))
    .name("process-request")
    .kind(SERVER)
    ... add tags etc.

// start the server side and flush instead of finish
oneWayReceive.start().flush();

// you should not modify this span anymore as it is complete. However,
// you can create children to represent follow-up work.
next = tracer.newSpan(oneWayReceive.context()).name("step2").start();
```

通过Spring 2.5 MVC或Spring 3.0 MVC插件埋点

您可以选择通过Spring 2.5 MVC或Spring 3.0 MVC插件进行埋点。

> 说明 如需获取Demo，请单击下载源码，进入springMvcDemo\webmvc3|webmvc25目录，并根据Readme运行程序。

1. 在applicationContext.xml中配置Tracing对象。

   > 说明

```
<bean class="zipkin2.reporter.beans.OkHttpSenderFactoryBean">
  <property name="endpoint" value="<endpoint>"/>
</bean>

<!-- allows us to read the service name from spring config -->
<context:property-placeholder/>

<bean class="brave.spring.beans.TracingFactoryBean">
  <property name="localServiceName" value="brave-webmvc3-example"/>
  <property name="spanReporter">
    <bean class="zipkin2.reporter.beans.AsyncReporterFactoryBean">
      <property name="encoder" value="JSON_V2"/>
      <property name="sender" ref="sender"/>
      <!-- wait up to half a second for any in-flight spans on close -->
      <property name="closeTimeout" value="500"/>
```

```
    <property name="closeTimeout" value="500" />
  </bean>
 </property>
 <property name="propagationFactory">
  <bean class="brave.propagation.ExtraFieldPropagation" factory-method="newFactory">
   <constructor-arg index="0">
    <util:constant static-field="brave.propagation.B3Propagation.FACTORY"/>
   </constructor-arg>
   <constructor-arg index="1">
    <list>
     <value>user-name</value>
    </list>
   </constructor-arg>
  </bean>
 </property>
 <property name="currentTraceContext">
  <bean class="brave.spring.beans.CurrentTraceContextFactoryBean">
   <property name="scopeDecorators">
    <bean class="brave.context.log4j12.MDCScopeDecorator" factory-method="create"/>
   </property>
  </bean>
 </property>
</bean>

<bean class="brave.spring.beans.HttpTracingFactoryBean">
 <property name="tracing" ref="tracing"/>
</bean>
```

2. 添加Interceptors对象。

```
<bean class="brave.httpclient.TracingHttpClientBuilder"
    factory-method="create">
  <constructor-arg type="brave.http.HttpTracing" ref="httpTracing"/>
 </bean>

 <bean factory-bean="httpClientBuilder" factory-method="build"/>

 <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
  <property name="interceptors">
   <list>
    <bean class="brave.spring.webmvc.SpanCustomizingHandlerInterceptor"/>
   </list>
  </property>
 </bean>

 <!-- Loads the controller -->
 <context:component-scan base-package="brave.webmvc"/>
```

3. 添加Filter对象。

```
<!-- Add the delegate to the standard tracing filter and map it to all paths -->
<filter>
 <filter-name>tracingFilter</filter-name>
 <filter-class>brave.spring.webmvc.DelegatingTracingFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>tracingFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

通过Spring 4.0 MVC或Spring Boot插件埋点

您可以选择通过Spring 4.0 MVC或Spring Boot插件进行埋点。

> 说明 如需获取Demo，请单击下载源码，进入springMvcDemo\webmvc4-boot|webmv4目录，并根据Readme运行程序。

1. 配置Tracing和Filter。

> 说明

```
/** Configuration for how to send spans to Zipkin */
 @Bean Sender sender() {
```

```
  return OkHttpSender.create("<endpoint>");
 }

 /** Configuration for how to buffer spans into messages for Zipkin */
 @Bean AsyncReporter<Span> spanReporter() {
  return AsyncReporter.create(sender());
 }

 /** Controls aspects of tracing such as the name that shows up in the UI */
 @Bean Tracing tracing(@Value("${spring.application.name}") String serviceName) {
  return Tracing.newBuilder()
      .localServiceName(serviceName)
      .propagationFactory(ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "user-name
"))
      .currentTraceContext(ThreadLocalCurrentTraceContext.newBuilder()
          .addScopeDecorator(MDCScopeDecorator.create()) // puts trace IDs into logs
          .build()
      )
      .spanReporter(spanReporter()).build();
 }

 /** decides how to name and tag spans. By default they are named the same as the http method.
*/
 @Bean HttpTracing httpTracing(Tracing tracing) {
  return HttpTracing.create(tracing);
 }

 /** Creates client spans for http requests */
 // We are using a BPP as the Frontend supplies a RestTemplate bean prior to this configuration
 @Bean BeanPostProcessor connectionFactoryDecorator(final BeanFactory beanFactory) {
  return new BeanPostProcessor() {
    @Override public Object postProcessBeforeInitialization(Object bean, String beanName) {
      return bean;
    }

    @Override public Object postProcessAfterInitialization(Object bean, String beanName) {
      if (!(bean instanceof RestTemplate)) return bean;

      RestTemplate restTemplate = (RestTemplate) bean;
      List<ClientHttpRequestInterceptor> interceptors =
```

```
          new ArrayList<>(restTemplate.getInterceptors());
      interceptors.add(0, getTracingInterceptor());
      restTemplate.setInterceptors(interceptors);
      return bean;
    }

    // Lazy lookup so that the BPP doesn't end up needing to proxy anything.
    ClientHttpRequestInterceptor getTracingInterceptor() {
      return TracingClientHttpRequestInterceptor.create(beanFactory.getBean(HttpTracing.class))
;
    }
  };
 }

 /** Creates server spans for http requests */
 @Bean Filter tracingFilter(HttpTracing httpTracing) {
  return TracingFilter.create(httpTracing);
 }

 @Autowired SpanCustomizingAsyncHandlerInterceptor webMvcTracingCustomizer;

 /** Decorates server spans with application-defined web tags */
 @Override public void addInterceptors(InterceptorRegistry registry) {
  registry.addInterceptor(webMvcTracingCustomizer);
 }
```

2. 配置autoconfigure（spring.factories）。

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
brave.webmvc.TracingConfiguration
```

通过Dubbo插件埋点

您可以选择通过Dubbo插进进行埋点。

> 说明 如需获取Demo，请单击下载源码，进入dubboDem目录，并根据Readme运行程序。

1. 添加依赖Jar包。

```
<dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-instrumentation-dubbo-rpc</artifactId>
        <version>5.4.2</version>
    </dependency>

    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-spring-beans</artifactId>
        <version>5.4.2</version>
    </dependency>

    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-context-slf4j</artifactId>
        <version>5.4.2</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-sender-okhttp3</artifactId>
        <version>2.7.9</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave</artifactId>
        <version>5.4.2</version>
    </dependency>

    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-sender-okhttp3</artifactId>
        <version>2.7.9</version>
    </dependency>
```

2. 配置Tracing对象。

> 说明

```
<bean class="zipkin2.reporter.beans.OkHttpSenderFactoryBean">
    <property name="endpoint" value="<endpoint>"/>
  </bean>

  <bean class="brave.spring.beans.TracingFactoryBean">
    <property name="localServiceName" value="double-provider"/>
    <property name="spanReporter">
      <bean class="zipkin2.reporter.beans.AsyncReporterFactoryBean">
        <property name="sender" ref="sender"/>
        <!-- wait up to half a second for any in-flight spans on close -->
        <property name="closeTimeout" value="500"/>
      </bean>
    </property>
    <property name="currentTraceContext">
      <bean class="brave.spring.beans.CurrentTraceContextFactoryBean">
        <property name="scopeDecorators">
          <bean class="brave.context.slf4j.MDCScopeDecorator" factory-method="create"/>
        </property>
      </bean>
    </property>
  </bean>
```

3. 添加Filter配置。

```
// 服务端配置。
<dubbo:provider filter="tracing" />
// 客户端配置。
<dubbo:consumer filter="tracing" />
```

通过Spring Sleuth插件埋点

您可以选择通过Spring Sleuth插进进行埋点。

> **说明** 如需获取Demo，请单击下载源码，进入sleuthDemo目录，并根据Readme运行程序。

1. 添加依赖Jar包。

```
<dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave</artifactId>
        <version>5.4.2</version>
    </dependency>

    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-sender-okhttp3</artifactId>
        <version>2.7.9</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.0.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-core</artifactId>
        <version>2.0.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-zipkin</artifactId>
        <version>2.0.1.RELEASE</version>
    </dependency>
```

2. 配置application.yml。

> **说明** 请将 `<endpoint_short>` 替换成控制台概览页面上相应地域的接入点（“公网接入点：”后面到“api/v2/spans”之前的内容）。

```
spring:
  application:
    # This ends up as the service name in zipkin
    name: sleuthDemo
  zipkin:
    # Uncomment to send to zipkin, replacing 192.168.99.100 with your zipkin IP address
    baseUrl: <endpoint_short>

  sleuth:
    sampler:
      probability: 1.0

sample:
  zipkin:
    # When enabled=false, traces log to the console. Comment to send to zipkin
    enabled: true
```

常见问题

问：Demo程序执行成功，但是为什么有的网站上无数据？

答：请断点调试zipkin2.reporter.okhttp3.HttpCall中的parseResponse方法，查看上报数据时返回值。如果报403错误，表示用户名配置不正确，需要检查endpoint配置。

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- brave-webmvc-example
- brave-instrumentation
- spring-cloud-zipkin-sleuth-tutorial

3.3. 通过SkyWalking上报Java应用数据

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

前提条件

- 打开SkyWalking下载页面，下载SkyWalking 6.X.X、7.X.X或8.X.X版本（建议下载SkyWalking 8.0.1），并将解压后的*Agent*文件夹放至Java进程有访问权限的目录。
- 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，日志文件默输出到 `/logs` 目录中。

> 警告 日志、插件和配置文件都在Agent文件夹中，请不要改变文件夹结构。

获取接入点和鉴权令牌 ❯

背景信息

SkyWalking是一款广受欢迎的国产APM（Application Performance Monitoring，应用性能监控）产品，主要针对微服务、Cloud Native和容器化（Docker、Kubernetes、Mesos）架构的应用。SkyWalking的核心是一个分布式追踪系统，目前是Apache基金会的顶级项目。

要通过SkyWalking将Java应用数据上报至链路追踪控制台，首先需要完成埋点工作。SkyWalking既支持自动探针（Dubbo、gRPC、JDBC、OkHttp、Spring、Tomcat、Struts、Jedis等），也支持手动埋点（OpenTracing）。本文介绍自动埋点方法。

数据是如何上报的？

用SkyWalking为Java应用自动埋点

1. 打开 `config/agent.config` ，配置接入点和令牌。

   注意 请将 `<endpoint>` 和 `<auth-token>` 分别替换成控制台概览页面上SkyWalking客户端在相应地域的接入点和鉴权令牌。关于获取方法，请参见前提条件中的**获取SkyWalking接入点和鉴权令牌**。

   ```
   collector.backend_service=<endpoint>
   agent.authentication=<auth-token>
   ```

2. 采用以下方法之一配置应用名称（Service Name）。

   注意 请将 `<ServiceName>` 替换为您的应用名称。如果同时采用以下两种方法，则仅第二种方法（在启动命令行中添加参数）生效。

   - 打开 `config/agent.config` ，配置应用名称。

     ```
     agent.service_name=<ServiceName>
     ```

   - 在应用程序的启动命令行中添加-Dskywalking.agent.service_name参数。

     ```
     java -javaagent:<skywalking-agent-path> -Dskywalking.agent.service_name=<ServiceName> -jar yourApp.jar
     ```

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

   说明 请将以下示例代码中的 `<skywalking-agent-path>` 替换为Agent文件夹中的skywalking-agent.jar的绝对路径。

   - 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
     ```

   - Windows Tomcat 7 / Tomcat 8
     在 `tomcat/bin/catalina.bat` 第一行添加以下内容：

```
set "CATALINA_OPTS=-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。

  > 注意 -javaagent参数一定要在-jar参数之前。

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

- Jetty
  在 `{JETTY_HOME}/start.ini` 配置文件中添加以下内容：

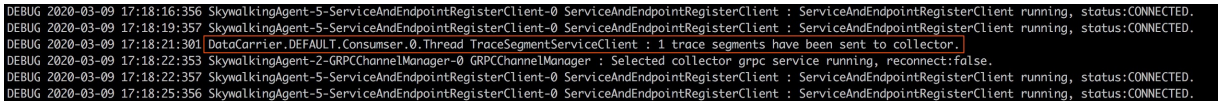
  ```
  --exec  # 去掉前面的井号取消注释。
  -javaagent:<skywalking-agent-path>
  ```

4. 重新启动应用。

常见问题

问：SkyWalking正常连接服务端后，无法创建应用？

答：可能是由于链路追踪的数据未上报。您需要检查是否有链路追踪的数据上报，可以查看*{skywalking agent path}/logs/skywalking-api.log*内容。如果有数据上报，则显示如下图所示。



如果未产生数据上报，则可能原因是：开启采样、设置过滤或未触发生成链路追踪的请求。

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- SkyWalking官网
- 下载SkyWalking
- 部署SkyWalking Java Agent

4.开始监控 PHP 应用

4.1. 通过Zipkin上报PHP应用数据

Zipkin是一款开源的分布式实时数据追踪系统（Distributed Tracking System），由Twitter公司开发和贡献。其主要功能是聚合来自各个异构系统的实时监控数据。在链路追踪Tracing Analysis中，您可以通过Zipkin上报PHP应用数据。

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

代码埋点

要通过Zipkin将PHP应用数据上报至链路追踪控制台，首先需要完成埋点工作。

1. 安装Zipkin的PHP插件。

```
composer require openzipkin/zipkin
```

2. 创建Tracer。Tracer对象可以用来创建Span对象（记录分布式操作时间）。Tracer对象还配置了上报数据的网关地址、本机IP、采样频率等数据，您可以通过调整采样率来减少因上报数据产生的开销。

```
function create_tracing($endpointName, $ipv4)
{
  $endpoint = Endpoint::create($endpointName, $ipv4, null, 2555);
  /* Do not copy this logger into production.
   * Read https://github.com/Seldaek/monolog/blob/master/doc/01-usage.md#log-levels
   */
  $logger = new \Monolog\Logger('log');
  $logger->pushHandler(new \Monolog\Handler\ErrorLogHandler());
  $reporter = new Zipkin\Reporters\Http(\Zipkin\Reporters\Http\CurlFactory::create());
  $sampler = BinarySampler::createAsAlwaysSample();
  $tracing = TracingBuilder::create()
    ->havingLocalEndpoint($endpoint)
    ->havingSampler($sampler)
    ->havingReporter($reporter)
    ->build();
  return $tracing;
}
```

3. 记录请求数据。

```
$rootSpan = $tracer->newTrace();
$rootSpan->setName('encode')
$rootSpan->start();

try {
 doSomethingExpensive();
} finally {
 $rootSpan->finish();
}
```

> 说明
> 以上代码用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要传入上下文。
> 示例：

```
$span = $tracer->newChild($parentSpan->getContext());
$span->setName('encode');
$span->start();
try {
  doSomethingExpensive();
} finally {
  $span->finish();
}
```

4. （可选）为了快速排查问题，您可以为某个记录添加一些自定义标签，例如记录是否发生错误、请求的返回值等。

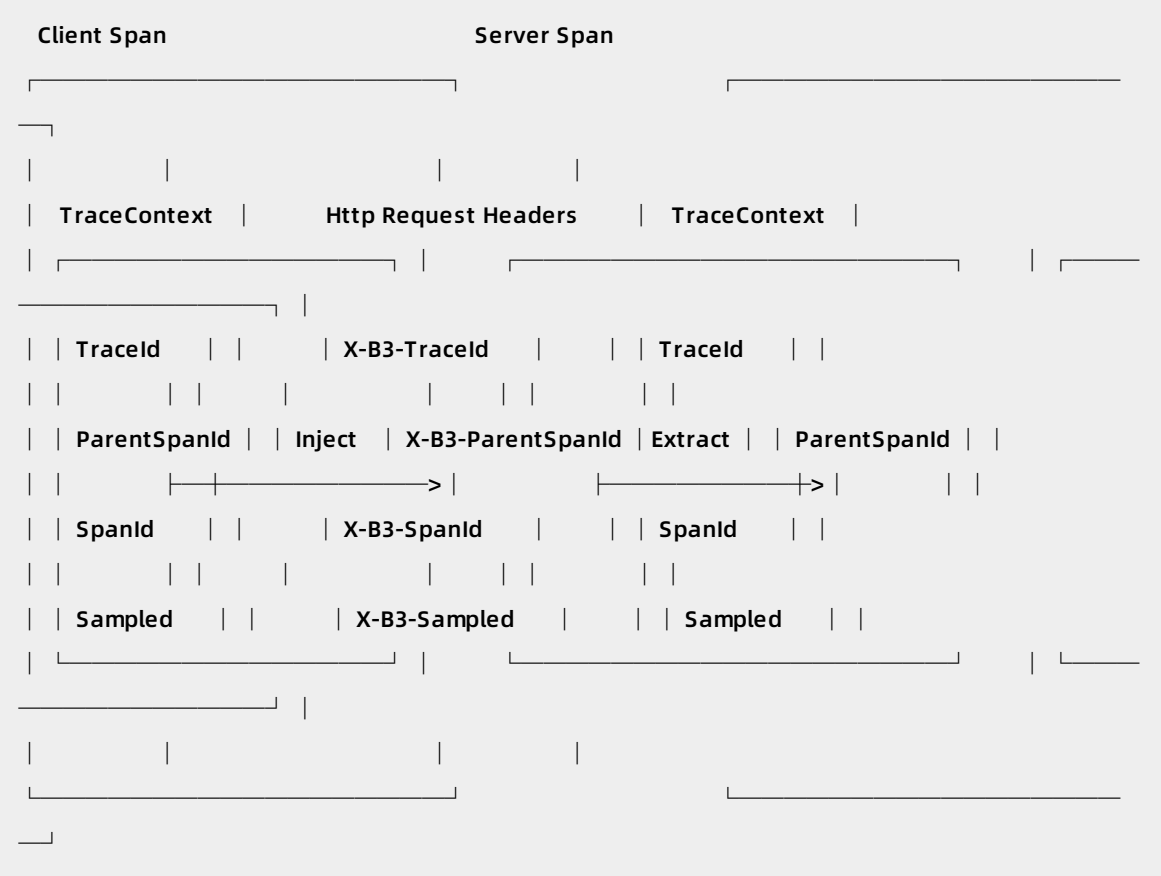
```
$span->tag('http.status_code', $resultCode);
```

5. （可选）请求上的采样率是由根操作决定的，一般在创建Tracer对象时会创建采样策略，但是您也可以在局部进行替换。例如：

```
private function newTrace(Request $request) {
  $flags = SamplingFlags::createAsEmpty();
  if (strpos($request->getUri(), '/experimental') === 0) {
    $flags = DefaultSamplingFlags::createAsSampled();
  } else if (strpos($request->getUri(), '/static') === 0) {
    $flags = DefaultSamplingFlags::createAsSampled();
  }
  return $tracer->newTrace($flags);
}
```

6. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。

总体流程如下：



i. 在客户端调用Inject方法传入Context信息。

```
// configure a function that injects a trace context into a request
$injector = $tracing->getPropagation()->getInjector(new RequestHeaders);

// before a request is sent, add the current span's context to it
$injector->inject($span->getContext(), $request);
```

ii. Extract方法解析Context信息。

```
// configure a function that extracts the trace context from a request
$extracted = $tracing->getPropagation()->extractor(new RequestHeaders);

$span = $tracer->newChild($extracted)
$span->setKind(Kind\SERVER);
```

快速入门

接下来以Zipkin的官方示例演示如何通过Zipkin上报PHP应用数据。

1. 下载示例文件。
2. 在functions.php文件中修改上报数据的网关地址，将 `$reporter = new Zipkin\Reporters\Http(\Zipkin\Reporters\Http\CurlFactory::create());` 替换为以下内容：

> 注意 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
reporter = new Zipkin\Reporters\Http(\Zipkin\Reporters\Http\CurlFactory::create(),
['endpoint_url' => '<endpoint>
',]);
```

3. 安装依赖包。

```
// As of zipkin php is still in 1.0.0-betaX
rm composer.lock && composer install
```

4. 启动服务。

```
# 在终端1中执行composer run-frontend


# 在终端2中执行composer run-backend
```

5. 访问前端页面上报数据，并在链路追踪控制台查看上报的数据。

```
curl http://localhost:8081
```

常见问题

问：为什么按照快速入门的步骤操作没有上报数据？

答：请检查endpoint_url的配置是否正确。

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- Zipkin官网
- Zipkin-php源码
- zipkin-php-example

5.开始监控 Go 应用

5.1. 通过Jaeger上报Go应用数据

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

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

快速开始

1. 运行以下命令，在*GOPATH/src*目录下载Demo文件。

```
wget http://arms-apm.oss-cn-hangzhou.aliyuncs.com/tools/tracingtest.zip && unzip tracingtest.zip
```

2. 修改配置。

> 注意 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
sender := transport.NewHTTPTransport(
    // 设置网关，网关因地域而异。
    "<endpoint>",
    )
```

3. 运行以下命令上传数据。

```
go run main.go
http done
grpc done
```

> 说明
> 如果出现以下错误，说明用户名和密码不正确，请更正并重试。

```
go run main.go
http done
2018/09/17 21:11:54 ERROR: error when flushing the buffer: error from collector: 403
2018/09/17 21:11:54 ERROR: error when flushing the buffer: error from collector: 403
```

4. 登录链路追踪控制台。执行上一步骤后等待30秒，即可查看上报的数据。

直接上报数据

1. 引入jaeger-client-go。
   - 包路径：*github.com/uber/jaeger-client-go*
   - 版本号：>=2.11.0

   以glide为例，您需要在glide.yaml中加入以下配置：

```
package: github.com/uber/jaeger-client-go
version: ^2.11.0
subpackages:
transport
```

2. 创建Trace对象。

   > 注意 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
func NewJaegerTracer(service string) (opentracing.Tracer, io.Closer) {

   sender := transport.NewHTTPTransport(
     // 设置网关，网关因地域而异。
      "<endpoint>",
   )
  tracer, closer:= jaeger.NewTracer(service,
      jaeger.NewConstSampler(true),
   jaeger.NewRemoteReporter(sender))
   return tracer, closer
}
```

3. 创建span实例对象和数据透传。
   - 如果没有parentSpan

```
// 创建Span。
span := tracer.StartSpan("myspan")
// 设置Tag。
clientSpan.SetTag("mytag", "123")
// 透传traceId。
tracer.Inject(span.Context(), opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(req.H
eader))
...
defer  span.Finish()
```

   - 如果有parentSpan

```
// 从HTTP/RPC对象解析出spanCtx。
spanCtx, _ := tracer.Extract(opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(r.Header))
span := tracer.StartSpan("myspan", opentracing.ChildOf(spanCtx))
...
defer  span.Finish()
```

通过Agent上报数据

1. 引入jaeger-client-go。
   - 包路径：*github.com/uber/jaeger-client-go*
   - 版本号：>=2.11.0

   以glide为例，您需要在glide.yaml中加入以下配置：

```
package: github.com/uber/jaeger-client-go
version: ^2.11.0
subpackages:
transport
```

2. 创建Trace对象。

```
func NewJaegerTracer(serviceName string) (opentracing.Tracer, io.Closer) {
  sender, _ := jaeger.NewUDPTransport("",0)
  tracer, closer:= jaeger.NewTracer(serviceName,
     jaeger.NewConstSampler(true),
     jaeger.NewRemoteReporter(sender))
   return tracer, closer
}
```

3. 下载原生Jaeger Agent jaeger-agent，并用以下参数启动Agent，以将数据上报至链路追踪Tracing Analysis。

> **注意** 请将 `<endpoint>` 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
// reporter.grpc.host-port用于设置网关，网关因地域而异。 例如：
$ nohup ./jaeger-agent --reporter.grpc.host-port=tracing-analysis-dc-sz.aliyuncs.com:1883 --jaeger.tags=<endpoint>
```

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

5.2. 通过Zipkin上报Go应用数据

Zipkin是一款开源的分布式实时数据追踪系统（Distributed Tracking System），由Twitter公司开发和贡献。其主要功能是聚合来自各个异构系统的实时监控数据。在链路追踪Tracing Analysis中，您可以通过Zipkin上报Go应用数据。

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

代码埋点

要通过Zipkin将Go应用数据上报至链路追踪控制台，首先需要完成埋点工作。

1. 添加组件依赖。

```
[[constraint]]
  name = "github.com/openzipkin/zipkin-go"
  version = "0.1.1"

[[constraint]]
  name = "github.com/gorilla/mux"
  version = "1.6.2"
```

2. 创建Tracer。Tracer对象可以用来创建Span对象（记录分布式操作时间）。Tracer对象还配置了上报数据的网关地址、本机IP、采样频率等数据，您可以通过调整采样率来减少因上报数据产生的开销。

```
func getTracer(serviceName string, ip string) *zipkin.Tracer {
  // create a reporter to be used by the tracer
  reporter := httpreporter.NewReporter("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_aokcd
qnxyz@123456ff_abcdef123@abcdef123/api/v2/spans")
  // set-up the local endpoint for our service
  endpoint, _ := zipkin.NewEndpoint(serviceName, ip)
  // set-up our sampling strategy
  sampler := zipkin.NewModuloSampler(1)
  // initialize the tracer
  tracer, _ := zipkin.NewTracer(
    reporter,
    zipkin.WithLocalEndpoint(endpoint),
    zipkin.WithSampler(sampler),
  )
  return tracer;
}
```

3. 记录请求数据。

```
// tracer can now be used to create spans.
span := tracer.StartSpan("some_operation")
// ... do some work ...
span.Finish()
// Output:
```

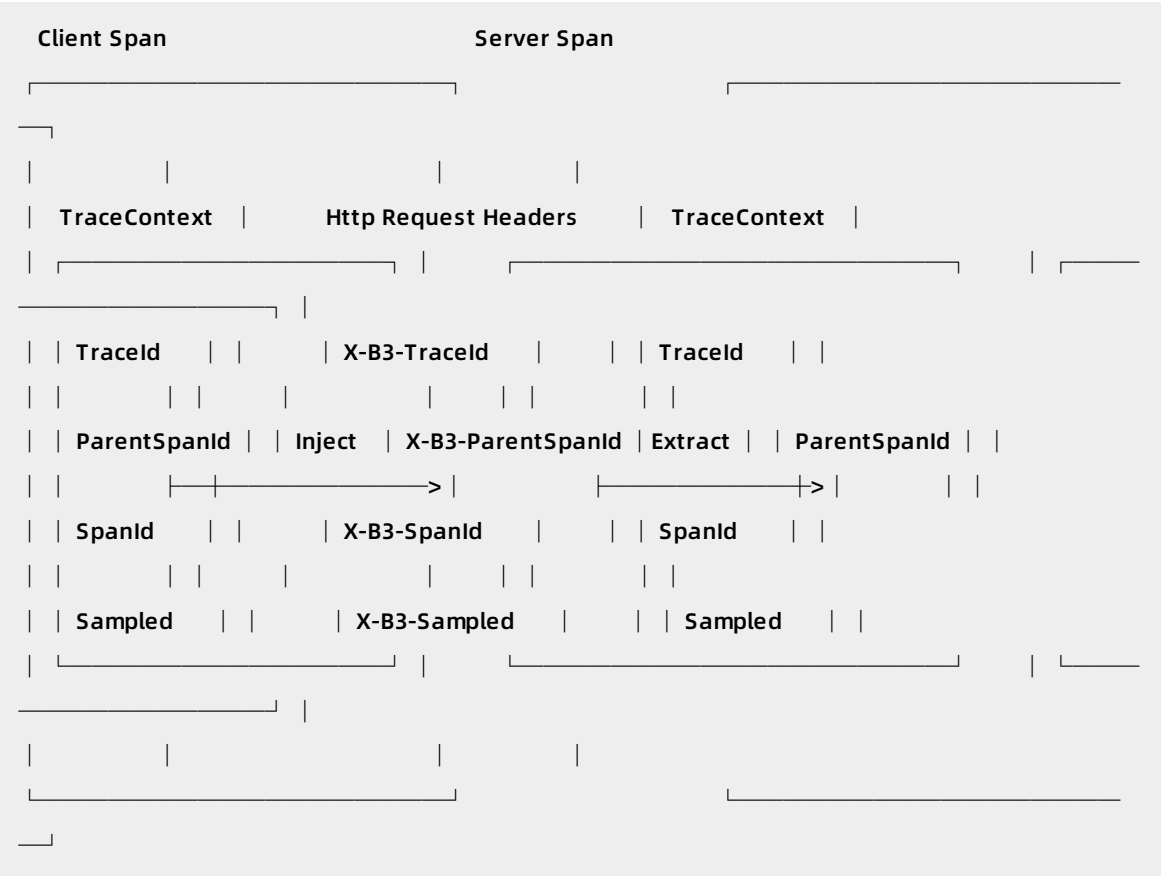
> 说明
> 以上代码用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要传入上下文。
> 示例：

```
childSpan := tracer.StartSpan("some_operation2", zipkin.Parent(span.Context()))
  // ... do some work ...
childSpan.Finish()
```

4. （可选）为了快速排查问题，您可以为某个记录添加一些自定义标签，例如记录是否发生错误、请求的返回值等。

```
childSpan.Tag("http.status_code", statusCode)
```

5. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。总体流程如下：



> 说明 目前Zipkin已有组件支持以HTTP、gRPC这两种RPC协议透传Context信息。

i. 在客户端调用Inject方法传入Context信息。

```
req, _ := http.NewRequest("GET", "/", nil)
// configure a function that injects a trace context into a reques
injector := b3.InjectHTTP(req)
injector(sp.Context())
```

ii. 在服务端调用Extract方法解析Context信息。

```
req, _ := http.NewRequest("GET", "/", nil)
    b3.InjectHTTP(req)(sp.Context())

    b.ResetTimer()
    _ = b3.ExtractHTTP(copyRequest(req))
```

快速入门

接下来以一个示例演示如何通过Zipkin上报Go应用数据。

1. 下载示例文件。
2. 在utils.go文件中修改上报数据的网关地址（endpointURL）。

> 注意 请将 `<endpoint>` 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

3. 安装依赖包。

```
dep ensure
```

4. 运行测试程序。

```
go run main.go
```

5. 在链路追踪控制台查看上报的数据。

常见问题

问：为什么按照快速入门的步骤操作没有上报数据？

答：请检查运行过程中是否有提示，并检查endpoint_url的配置是否正确。例如，错误 `failed the request with status code 403` 表明用户名或密码不正确。

更多信息

❷不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- Zipkin官网
- Zipkin-go源码

6.开始监控 Python 应用

6.1. 通过Jaeger上报Python应用数据

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

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

操作步骤

1. 运行以下命令安装jaeger-client。

   ```
   pip install jaeger-client
   ```

2. 创建Tracer对象，并通过Tracer对象创建Span来追踪业务流程。

```
import logging
import time
from jaeger_client import Config

if __name__ == "__main__":
    log_level = logging.DEBUG
    logging.getLogger('').handlers = []
    logging.basicConfig(format='%(asctime)s %(message)s', level=log_level)

    config = Config(
        config={ # usually read from some yaml config
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name='your-app-name',
    )
    # this call also sets opentracing.tracer
    tracer = config.initialize_tracer()

    with tracer.start_span('TestSpan') as span:
        span.log_kv({'event': 'test message', 'life': 42})

        with tracer.start_span('ChildSpan', child_of=span) as child_span:
            span.log_kv({'event': 'down below'})

    time.sleep(2)   # yield to IOLoop to flush the spans - https://github.com/jaegertracing/jaeger-client-python/issues/50
    tracer.close()  # flush any buffered spans
```

3. 下载原生Jaeger Agent jaeger-agent，并用以下参数启动Agent，以将数据上报至链路追踪Tracing Analysis。

> **注意** 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
// reporter.grpc.host-port用于设置网关，网关因地域而异。 例如：
$ nohup ./jaeger-agent --reporter.grpc.host-port=tracing-analysis-dc-sz.aliyuncs.com:1883 --jaeger.tags=<endpoint>
```

Jaeger使用方法

- 创建Trace

```
from jaeger_client import Config


def init_jaeger_tracer(service_name='your-app-name'):
    config = Config(config={}, service_name=service_name)
    return config.initialize_tracer()
```

- 创建和结束Span

```
// 开始无Parent的Span。
tracer.start_span('TestSpan')
// 开始有Parent的Span。
tracer.start_span('ChildSpan', child_of=span)
// 结束Span。
 span.finish()
```

- 透传SpanContext

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

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- OpenTracing指南
- jaeger-client-python

7.开始监控 Node.js 应用

7.1. 接入Node.js应用

本文介绍了如何将Node.js应用接入链路追踪。

前提条件

在Node工程的Package中配置对Jaeger Client的依赖。

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

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

操作步骤

1. 初始化配置。

   > 注意 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

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

   const config = {
     serviceName: 'node-service',
     sampler: {
       type: "const",
       param: 1
     },
     reporter: {
       collectorEndpoint: "<endpoint>"
     },
   };
   ```

2. 创建Tracer实例对象。

   ```
   const tracer = initTracer(config);
   ```

3. 创建Span实例对象。

```
const span = tracer.startSpan("say-hello");

// 设置标签（可选，支持多个）。
span.setTag("tagKey-01", "tagValue-01");

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

// 标记Span结束span.finish();
```

4. 登录链路追踪控制台并查看调用链。

基于Express的完整示例

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

const config = {
  serviceName: 'node-service',
  sampler: {
    type: "const",
    param: 1
  },
  reporter: {
    collectorEndpoint: "<endpoint>"
  },
};
const tracer = initTracer(config);

app.all('*', function (req, res, next) {
  req.span = tracer.startSpan("say-hello");
  next();
});

app.get("/api", function (req, res) {
  const span = req.span;
  span.log({event: "timestamp", value: Date.now()});
  req.span.finish();
  res.send({code: 200, msg: "success"});
});

app.listen(3000, '127.0.0.1', function () {
  console.log('start');
});
```

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- jaeger-client-node
- opentracing-javascript

8.开始监控 .NET 应用

8.1. 通过Zipkin上报 .NET应用数据

Zipkin是一款开源的分布式实时数据追踪系统（Distributed Tracking System），由Twitter公司开发和贡献。其主要功能是聚合来自各个异构系统的实时监控数据。在链路追踪Tracing Analysis中，您可以通过Zipkin上报 .NET应用数据（此方法同样适用于使用C#语言开发的应用）。

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

通过ASP.NET Core组件自动埋点

请按照以下步骤通过ASP.NET Core组件埋点。

> 说明 下载Demo源码，并按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加以下组件。
// zipkin4net.middleware.aspnetcore（aspnetcore中间件）
// zipkin4net（追踪器）

dotnet add  package zipkin4net.middleware.aspnetcore
dotnet add  package zipkin4nete
```

2. 注册和启动Zipkin。

```
lifetime.ApplicationStarted.Register(() => {
   TraceManager.SamplingRate = 1.0f;
   var logger = new TracingLogger(loggerFactory, "zipkin4net");
   // 在链路追踪控制台获取Zipkin Endpoint,注意Endpoint中不包含“/api/v2/spans”。
   var httpSender = new HttpZipkinSender("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_your_token", "application/json");

   var tracer = new ZipkinTracer(httpSender, new JSONSpanSerializer());
   TraceManager.RegisterTracer(tracer);
   TraceManager.Start(logger);
});
lifetime.ApplicationStopped.Register(() => TraceManager.Stop());
app.UseTracing(applicationName);
```

3. 在发送Get/Post请求的HttpClient中添加tracinghandler（追踪处理者）。

```
public override void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient("Tracer").AddHttpMessageHandler(provider =>
      TracingHandler.WithoutInnerHandler(provider.GetService<IConfiguration>()["applicationNam
e"]));
}
```

通过Owin组件自动埋点

请按照以下步骤通过Owin组件埋点。

> 说明 下载Demo源码，并按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加以下组件。
// zipkin4net.middleware.aspnetcore（aspnetcore中间件）
// zipkin4net（追踪器）

dotnet add  package zipkin4net.middleware.aspnetcore
dotnet add  package zipkin4nete
```

2. 注册和启动Zipkin。

```
// 设置Tracing。
TraceManager.SamplingRate = 1.0f;
var logger = new ConsoleLogger();
// 在链路追踪控制台获取Zipkin Endpoint,注意Endpoint中不包含“/api/v2/spans”。var httpSender = n
ew HttpZipkinSender("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_your_token", "applicatio
n/json");

var tracer = new ZipkinTracer(httpSender, new JSONSpanSerializer());
TraceManager.RegisterTracer(tracer);
TraceManager.Start(logger);

//Stop TraceManager on app dispose
var properties = new AppProperties(appBuilder.Properties);
var token = properties.OnAppDisposing;

if (token != CancellationToken.None)
{
token.Register(() =>
{
TraceManager.Stop();
});
}
//

// Setup Owin Middleware
appBuilder.UseZipkinTracer(System.Configuration.ConfigurationManager.AppSettings["applicatio
nName"]);
//
```

3. 在发送Get/Post请求的HttpClient中添加tracinghandler（追踪处理者）。

```
using (var httpClient = new HttpClient(new TracingHandler(applicationName)))
  {
    var response = await httpClient.GetAsync(callServiceUrl);
    var content = await response.Content.ReadAsStringAsync();

    await context.Response.WriteAsync(content);
  }
```

手动埋点

要通过Zipkin将Java应用数据上报至链路追踪控制台，除了利用各种现有插件实现埋点的目的，也可以手动埋点。

> 说明 下载Demo源码，并按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加zipkin4net（追踪器）。

dotnet add  package zipkin4nete
```

2. 注册和启动Zipkin。

```
TraceManager.SamplingRate = 1.0f;
var logger = new ConsoleLogger();
// 在链路追踪控制台获取Zipkin Endpoint,注意Endpoint中不包含“/api/v2/spans”。
var httpSender = new HttpZipkinSender("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_your_
token", "application/json");

var tracer = new ZipkinTracer(httpSender, new JSONSpanSerializer());
TraceManager.RegisterTracer(tracer);
TraceManager.Start(logger);
```

3. 记录请求数据。

```
var trace = Trace.Create();

Trace.Current = trace;
trace.Record(Annotations.ClientSend());

trace.Record(Annotations.Rpc("client"));
trace.Record(Annotations.Tag("mytag", "spanFrist"));
trace.Record(Annotations.ServiceName("dotnetManual"));
// ...do something
testCall();

trace.Record(Annotations.ClientRecv());
```

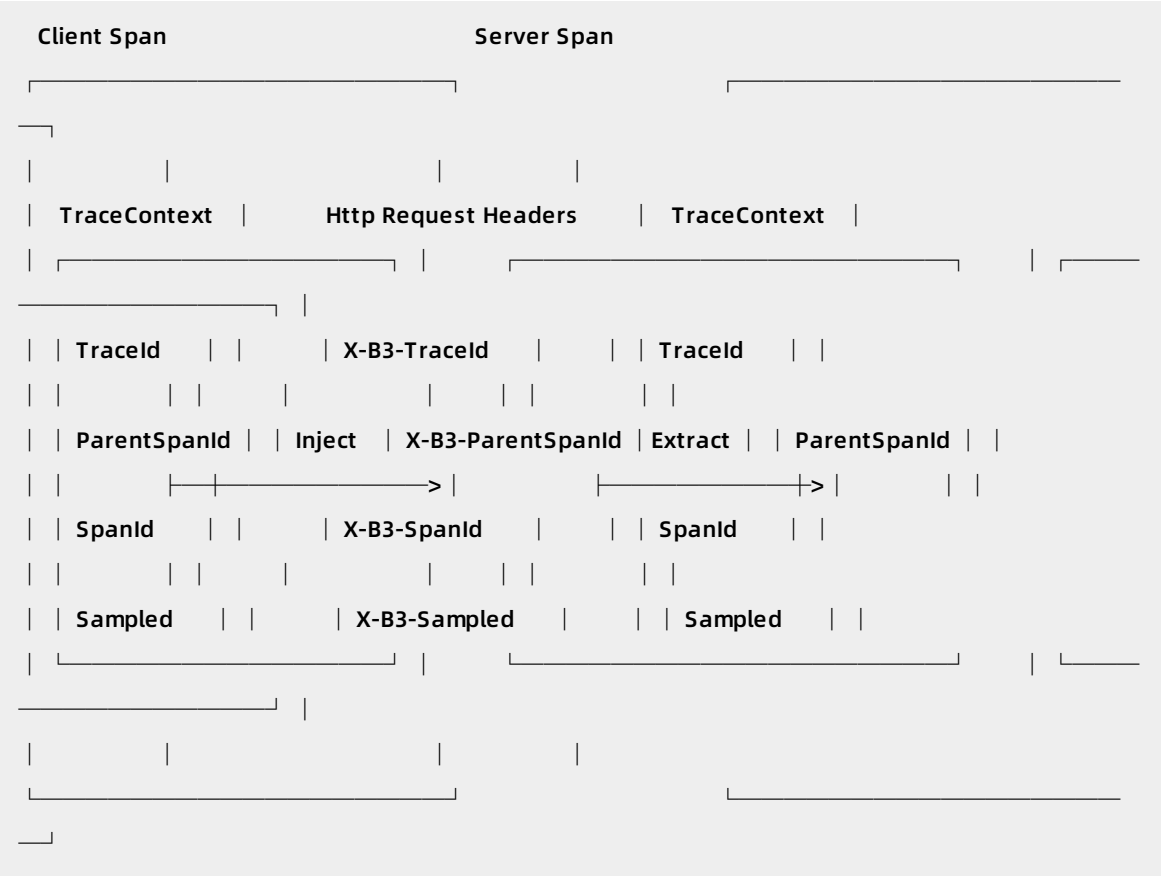
> 说明
> 以上代码用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要调用ChildOf方法。示例：

```
var trace = Trace.Current.Child();
Trace.Current = trace;
trace.Record(Annotations.ServerRecv());
trace.Record(Annotations.Rpc("server"));
trace.Record(Annotations.Tag("mytag", "spanSecond"));
trace.Record(Annotations.ServiceName("dotnetManual"));
// ...do something
trace.Record(Annotations.ServerSend());
```

4. （可选）为了快速排查问题，您可以为某个记录添加一些自定义标签，例如记录是否发生错误、请求的返回值等。

```
tracer.activeSpan().setTag("http.status_code", "200");
```

5. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。总体流程如下：



i. 在客户端调用Inject方法传入Context信息。

```
_injector.Inject(clientTrace.Trace.CurrentSpan, request.Headers);
```

ii. 在服务端调用Extract方法解析Context信息。

```
Ivar traceContext = traceExtractor.Extract(context.Request.Headers);
var trace = traceContext == null ? Trace.Create() : Trace.CreateFromId(traceContext);
```

常见问题

问：Demo程序执行成功，为什么控制台上没有上报数据？

答：请检查senderConfiguration配置中的Endpoint是否填写正确。

```
// 在链路追踪控制台获取Zipkin Endpoint,注意Endpoint中不包含“/api/v2/spans”。
var httpSender = new HttpZipkinSender("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_your_token", "application/json");
```

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- Zipkin官网
- Zipkin的dotnet客户端

8.2. 通过Jaeger上报 .NET应用数据

在使用链路追踪控制台追踪应用的链路数据之前，需要通过客户端将应用数据上报至链路追踪。本文介绍如何通过Jaeger客户端上报 .NET应用数据（此方法同样适用于使用C#语言开发的应用）。

前提条件

获取接入点信息 ❯

背景信息

Jaeger是Uber推出的一款开源分布式追踪系统，兼容OpenTracing API，已在Uber大规模使用，且已加入CNCF开源组织。其主要功能是聚合来自各个异构系统的实时监控数据。

目前OpenTracing社区已有许多组件可支持各种 .NET框架，例如：

- ASP.NET Core
- Entity Framework Core
- .NET Core BCL types (HttpClient)
- gRPC

数据是如何上报的？ ❯

通过netcore组件自动埋点

请按照以下步骤通过netcore组件埋点。

> 说明 下载Demo源码，并进入webapi.dotnetcore目录，按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加以下组件。
// OpenTracing.Contrib.NetCore（aspnetcore中间件）
// Jaeger（OpenTracing的实现组件）
// Microsoft.Extensions.Logging.Console（日志组件）

dotnet add  package OpenTracing.Contrib.NetCore
dotnet add  package Jaeger
dotnet add  package Microsoft.Extensions.Logging.Console
```

2. 用项目中Startup.cs里面的Configure方法注册中间件。

```
// 注册OpenTracing组件埋点。
services.AddOpenTracing();
// 过滤httpclient采集中的Jaeger数据上报请求。
var httpOption = new HttpHandlerDiagnosticOptions();
httpOption.IgnorePatterns.Add(req => req.RequestUri.AbsolutePath.Contains("/api/traces"));
services.AddSingleton(Options.Create(httpOption));
```

3. 初始化和注册Jaeger。

```
// 添加Jaeger Tracer。
services.AddSingleton<ITracer>(serviceProvider =>
{
   string serviceName = serviceProvider.GetRequiredService<IHostingEnvironment>().ApplicationName;
   ILoggerFactory loggerFactory = serviceProvider.GetRequiredService<ILoggerFactory>();
    Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderConfiguration(loggerFactory)
    // 在链路追踪控制台获取Jaeger Endpoint。
        .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/api/traces");

    // This will log to a default localhost installation of Jaeger.
    var tracer = new Tracer.Builder(serviceName)
        .WithSampler(new ConstSampler(true))
        .WithReporter(new RemoteReporter.Builder().WithSender(senderConfiguration.GetSender()).Build())
        .Build();

    // Allows code that can't use DI to also access the tracer.
    GlobalTracer.Register(tracer);

    return tracer;
});
```

通过gRPC组件自动埋点

请按照以下步骤通过gRPC组件埋点。

> 说明 下载Demo源码，并按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加以下组件。
// OpenTracing.Contrib.Grpc（gRPC中间件）
// Jaeger（OpenTracing的实现组件）
// Microsoft.Extensions.Logging.Console（日志组件）

dotnet add  package OpenTracing.Contrib.grpc
dotnet add  package Jaeger
```

2. 初始化ITracer对象。

```
public static Tracer InitTracer(string serviceName, ILoggerFactory loggerFactory)
    {
        Configuration.SamplerConfiguration samplerConfiguration = new Configuration.SamplerCo
nfiguration(loggerFactory)
            .WithType(ConstSampler.Type)
            .WithParam(1);
        Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderConfig
uration(loggerFactory)
              // 在链路追踪控制台获取Jaeger Endpoint。
             .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/api/trace
s");
        Configuration.ReporterConfiguration reporterConfiguration = new Configuration.ReporterC
onfiguration(loggerFactory)
            .WithSender(senderConfiguration);

        return (Tracer)new Configuration(serviceName, loggerFactory)
            .WithSampler(samplerConfiguration)
            .WithReporter(reporterConfiguration)
            .GetTracer();
    }
```

3. 在服务端埋点。构建用于埋点的ServerTracingInterceptor对象，并将ServerTracingInterceptor绑定到服务上。

```
ILoggerFactory loggerFactory = new LoggerFactory().AddConsole();
ITracer tracer = TracingHelper.InitTracer("dotnetGrpcServer", loggerFactory);
ServerTracingInterceptor tracingInterceptor = new ServerTracingInterceptor(tracer);
Server server = new Server
{
  Services = { Greeter.BindService(new GreeterImpl()).Intercept(tracingInterceptor) },
   Ports = { new ServerPort("localhost", Port, ServerCredentials.Insecure) }
};
```

4. 在客户端埋点。构建用于埋点的ClientTracingInterceptor对象，并将ClientTracingInterceptor绑定到Channel上。

```
ILoggerFactory loggerFactory = new LoggerFactory().AddConsole();
ITracer tracer = TracingHelper.InitTracer("dotnetGrpcClient", loggerFactory);
ClientTracingInterceptor tracingInterceptor = new ClientTracingInterceptor(tracer);
Channel channel = new Channel("127.0.0.1:50051", ChannelCredentials.Insecure);

var client = new Greeter.GreeterClient(channel.Intercept(tracingInterceptor));
```

手动埋点

要通过Jaeger将 .NET应用数据上报至链路追踪控制台，除了利用各种现有插件实现埋点的目的，也可以手动埋点。

> 说明 下载Demo源码，并进入ManualDemo目录，按照Readme的说明运行程序。

1. 安装NuGet包。

```
// Jaeger（OpenTracing的实现组件）
// Microsoft.Extensions.Logging.Console（日志组件）

dotnet add  package Microsoft.Extensions.Logging.Console
dotnet add  package Jaeger
```

2. 构建ITracer对象。ITracer是OpenTracing定义的对象，我们用Jaeger来构建该对象（配置网关、采样率等）。

```
public static ITracer InitTracer(string serviceName, ILoggerFactory loggerFactory)
{
   Configuration.SamplerConfiguration samplerConfiguration = new Configuration.SamplerConfiguration(loggerFactory)
        .WithType(ConstSampler.Type)
        .WithParam(1);
   Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderConfiguration(loggerFactory)
         // 在链路追踪控制台获取Jaeger Endpoint。
        .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/api/traces");

  Configuration.ReporterConfiguration reporterConfiguration = new Configuration.ReporterConfiguration(loggerFactory)
        .WithSender(senderConfiguration);

  return (Tracer)new Configuration(serviceName, loggerFactory)
        .WithSampler(samplerConfiguration)
        .WithReporter(reporterConfiguration)
      .GetTracer();
}
```

3. 将ITreacer注册到GlobalTracer中，方便调用代码。

```
GlobalTracer.Register(InitTracer("dotnetManualDemo", loggerFactory ));
```

4. 记录请求数据。

```
ITracer tracer = GlobalTracer.Instance;
ISpan span = tracer.BuildSpan("parentSpan").WithTag("mytag","parentSapn").Start();
tracer.ScopeManager.Activate(span, false);
// ...do something
..
span.Finish();
```

> 说明 以上代码用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要调用AsChildOf方法。

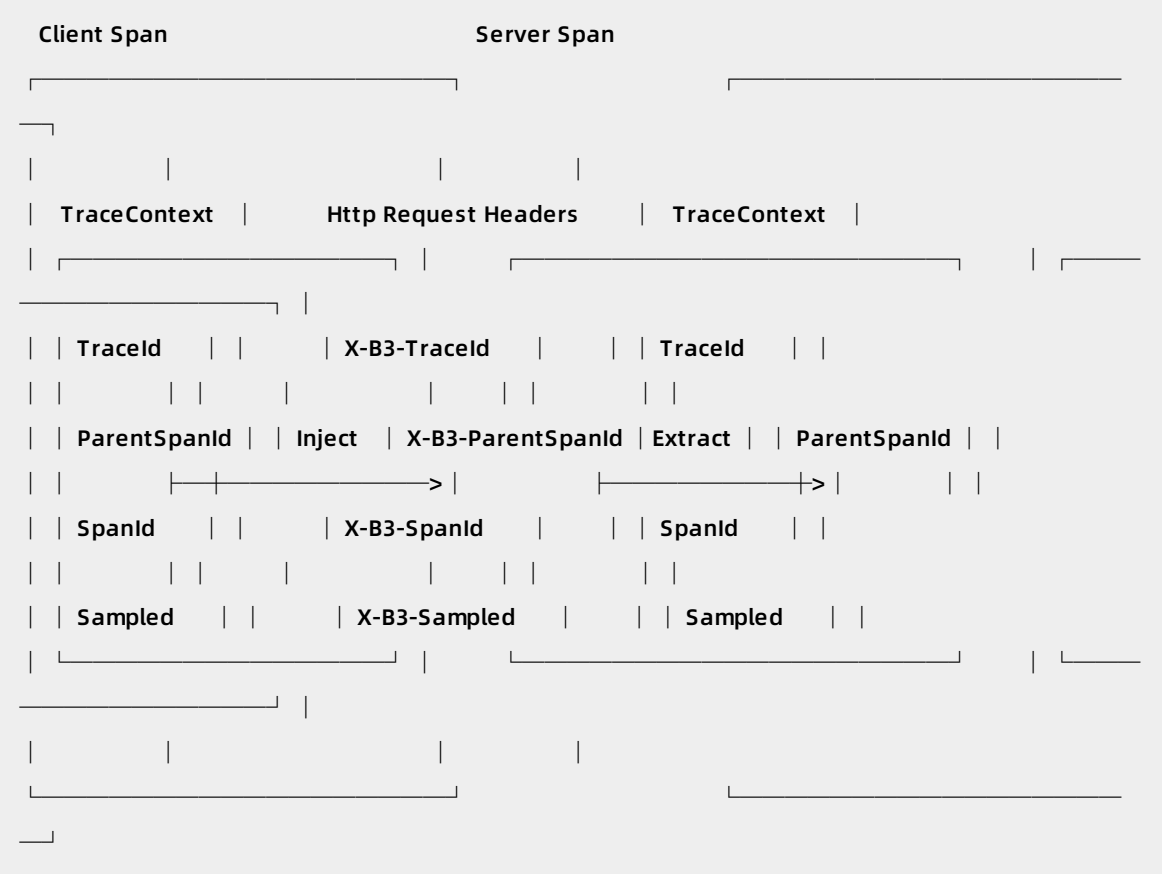
示例：

```
ITracer tracer = GlobalTracer.Instance;
ISpan parentSpan = tracer.ActiveSpan;
ISpan childSpan =tracer.BuildSpan("childSpan").AsChildOf(parentSpan).WithTag("mytag", "spanSe
cond").Start();
tracer.ScopeManager.Activate(childSpan, false);
// ... do something。。。childSpan.Finish();
```

5. （可选）为了快速排查问题，您可以为某个记录添加一些自定义标签，例如记录是否发生错误、请求的返回值等。

```
tracer.activeSpan().setTag("http.status_code", "200");
```

6. 在分布式系统中发送RPC请求时会带上Tracing数据，包括TraceId、ParentSpanId、SpanId、Sampled等。您可以在HTTP请求中使用Extract/Inject方法在HTTP Request Headers上透传数据。总体流程如下：



i. 在客户端调用Inject方法传入Context信息。

```
Tracer.Inject(span.Context, BuiltinFormats.HttpHeaders, new HttpHeadersInjectAdapter(request.Headers));
```

ii. 在服务端调用Extract方法解析Context信息。

```
ISpanContext extractedSpanContext = _tracer.Extract(BuiltinFormats.HttpHeaders, new RequestHeadersExtractAdapter(request.Headers));
ISpan childSpan = _tracer.BuildSpan(operationName).AsChildOf(extractedSpanContext);
```

常见问题

问：Demo程序执行成功，为什么控制台上没有上报数据？

答：请检查senderConfiguration配置中的Endpoint是否填写正确。

```
Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderConfiguration(loggerFactory)
        // 在链路追踪控制台获取Jaeger Endpoint。
        .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/api/traces");
```

更多信息

❷不是您要找的文档？鼠标悬浮在这里试一试。

相关文档

- Jaeger官网
- jaeger-client-csharp
- Jaeger对asp.net core组件库
- Jaeger对gRPC埋点组件

9.开始监控 C++ 应用

9.1. 通过Jaeger上报C++ 应用数据

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

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

直接上报数据

1. 运行以下命令，从官方网站获取jaeger-client-cpp。

```
wget https://github.com/jaegertracing/jaeger-client-cpp/archive/master.zip && unzip master.zip
```

2. 运行以下命令，编译jaeger-client-cpp。

> 说明 编译依赖CMake，gcc版本不低于4.9.2。

```
mkdir build
  cd build
  cmake ..
  make
```

3. 下载原生Jaeger Agent jaeger-agent，并用以下参数启动Agent，以将数据上报至链路追踪Tracing Analysis。

> 注意 请将 <endpoint> 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的获取接入点信息。

```
// reporter.grpc.host-port用于设置网关，网关因地域而异。 例如：
$ nohup ./jaeger-agent --reporter.grpc.host-port=tracing-analysis-dc-sz.aliyuncs.com:1883 --jaeg
er.tags=<endpoint>
```

4. 进入jaeger-client-cpp的example目录，运行测试用例。

```
./app ../examples/config.yml
```

5. 登录链路追踪控制台，查看上报的数据。

通过Agent上报数据

1. 安装Jaeger Client（官方下载地址）。

2. 创建Trace。
   例如，我们可以根据YAML配置生成Trace对象。

```
void setUpTracer(const char* configFilePath)
{
  auto configYAML = YAML::LoadFile(configFilePath);
  // 从YAML文件中导入配置。
  auto config = jaegertracing::Config::parse(configYAML);
  // 设置Trace的serviceName和日志。
  auto tracer = jaegertracing::Tracer::make(
    "example-service", config, jaegertracing::logging::consoleLogger());
  //将Tracer设置为全局变量。
  opentracing::Tracer::InitGlobal(
    std::static_pointer_cast<opentracing::Tracer>(tracer));
}
```

   YAML配置参考：

```
disabled: false
reporter:
  logSpans: true
sampler:
 type: const
 param: 1
```

3. 创建Span。

```
// 有parentSpan的情况下创建Span。
void tracedSubroutine(const std::unique_ptr<opentracing::Span>& parentSpan)
{
  auto span = opentracing::Tracer::Global()->StartSpan(
    "tracedSubroutine", { opentracing::ChildOf(&parentSpan->context()) });
}

// 无parentSpan的情况下创建Span。
void tracedFunction()
{
  auto span = opentracing::Tracer::Global()->StartSpan("tracedFunction");
  tracedSubroutine(span);
}
```

4. 下载原生Jaeger Agent jaeger-agent，并用以下参数启动Agent，以将数据上报至链路追踪Tracing Analysis。

> **注意** 请将 `<endpoint>` 替换成链路追踪控制台概览页面上相应客户端和相应地域的接入点。关于获取接入点信息的方法，请参见前提条件中的**获取接入点信息**。

```
// reporter.grpc.host-port用于设置网关，网关因地域而异。 例如：
$ nohup ./jaeger-agent --reporter.grpc.host-port=tracing-analysis-dc-sz.aliyuncs.com:1883 --jaeger.tags=<endpoint>
```

更多信息

❓不是您要找的文档？鼠标悬浮在这里试一试。