ALIBABA CLOUD

阿里云

链路追踪

链路追踪公共云合集

文档版本：20220706

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.动态与公告

1.1. 【产品变更】链路追踪部分地域商用通知

链路追踪Tracing Analysis于2022年07月05日00:00起对以下地域正式商用。

链路追踪Tracing Analysis定价详情，请参见链路追踪Tracing Analysis价格说明。

按量付费：

- 华北5（呼和浩特）
- 华北6（乌兰察布）
- 华南2（河源）
- 华南3（广州）
- 西南1（成都）
- 印度（孟买）
- 马来西亚（吉隆坡）
- 印度尼西亚（雅加达）
- 日本（东京）
- 新加坡（新加坡）
- 澳大利亚（悉尼）
- 德国（法兰克福）
- 英国（伦敦）
- 美国（硅谷）
- 美国（弗吉尼亚）

2.产品简介

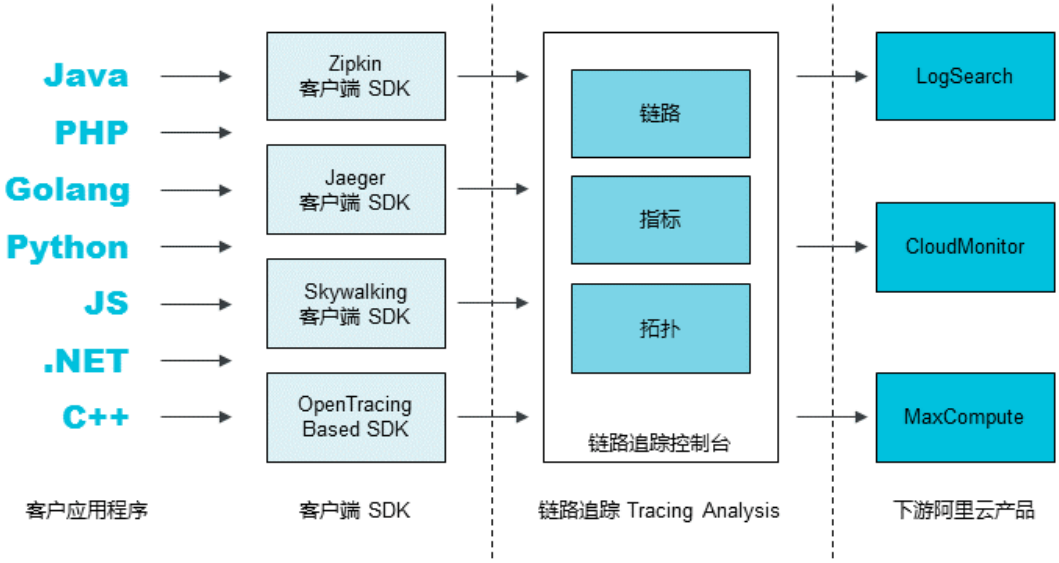
2.1. 什么是链路追踪Tracing Analysis

链路追踪Tracing Analysis为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等工具，可以帮助开发者快速分析和诊断分布式应用架构下的性能瓶颈，提高微服务时代下的开发诊断效率。

产品架构

链路追踪的产品架构如下图所示。

链路追踪产品架构



链路追踪的主要工作流程如下：

1. 客户侧的应用程序通过集成链路追踪的多语言客户端SDK上报服务调用数据。链路追踪支持多种开源社区的SDK，且支持OpenTracing标准。
2. 数据上报至链路追踪控制台后，链路追踪组件进行实时聚合计算和持久化，形成链路明细、性能总览、实时拓扑等监控数据。您可以据此进行问题排查与诊断。
3. 调用链数据可对接下游阿里云产品，例如LogSearch、CloudMonitor、MaxCompute等，用于离线分析、报警等场景。

产品功能

链路追踪的主要功能如下：

- 分布式调用链查询和诊断：追踪分布式架构中的所有微服务用户请求，并将它们汇总成分布式调用链。
- 应用性能实时汇总：通过追踪整个应用程序的用户请求，来实时汇总组成应用程序的单个服务和资源。
- 分布式拓扑动态发现：用户的所有分布式微服务应用和相关PaaS产品可以通过链路追踪收集到分布式调用信息。
- 多语言开发程序接入：基于OpenTracing标准，全面兼容开源社区，例如Jaeger、Zipkin。
- 丰富的下游对接场景：收集的链路可直接用于日志分析，且可对接到MaxCompute等下游分析平台。

2.2. 基本概念

本文介绍在使用链路追踪之前需要了解的基本概念，包括分布式追踪系统的作用，什么是调用链，链路追踪所依赖的OpenTracing数据模型，以及在链路追踪产品里数据是如何上报的。

为什么需要分布式追踪系统？

为了应对各种复杂的业务，开发工程师开始采用敏捷开发、持续集成等开发方式。系统架构也从单机大型软件演化成微服务架构。微服务构建在不同的软件集上，这些软件模块可能是由不同团队开发的，可能使用不同的编程语言来实现，还可能发布在多台服务器上。因此，如果一个服务出现问题，可能导致几十个应用都出现服务异常。

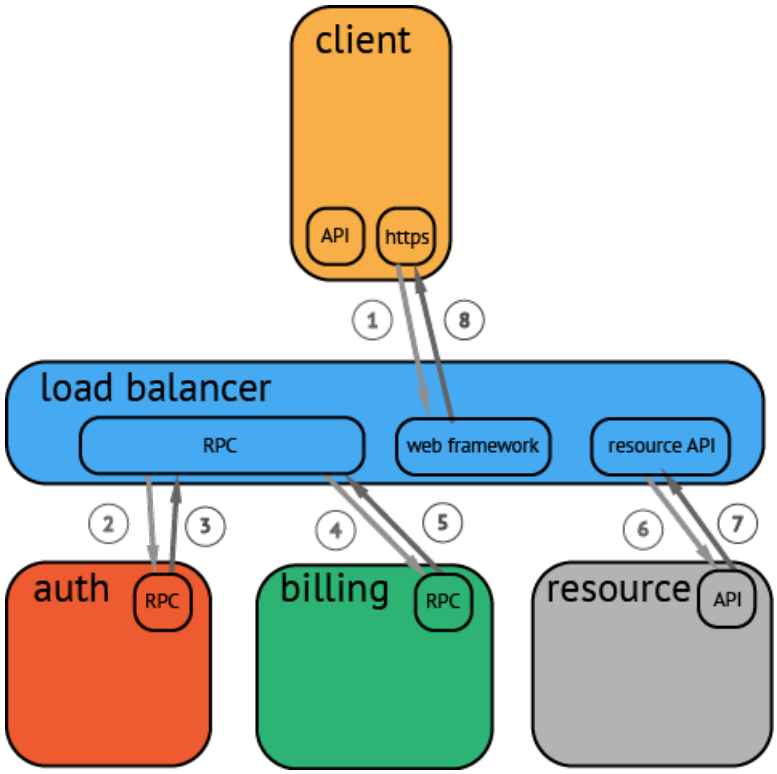
分布式追踪系统可以记录请求范围内的信息，例如一次远程方法调用的执行过程和耗时，是我们排查系统问题和系统性能的重要工具。

什么是调用链（Trace）？

在广义上，一个调用链代表一个事务或者流程在（分布式）系统中的执行过程。在OpenTracing标准中，调用链是多个Span组成的一个有向无环图（Directed Acyclic Graph，简称DAG），每一个Span代表调用链中被命名并计时的连续性执行片段。

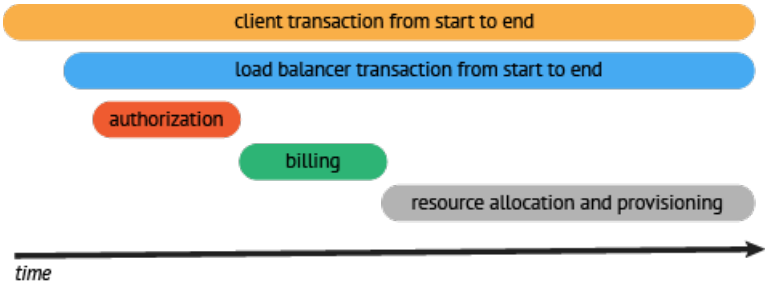
下图是一个分布式调用的例子：客户端发起请求，请求首先到达负载均衡器，接着经过认证服务、计费服务，然后请求资源，最后返回结果。

分布式调用示例



数据被采集存储后，分布式追踪系统一般会选择使用包含时间轴的时序图来呈现这个调用链。

包含时间轴的链路图



OpenTracing数据模型

整体概念

OpenTracing中的调用链（Trace）通过归属于此调用链的Span来隐性地定义。一条调用链可以视为一个由多个Span组成的有向无环图（DAG图）。Span之间的关系被命名为References。例如下面的示例调用链就是由8个Span组成的。

```
单个Trace中Span间的因果关系
        [Span A]  ←←←(The root span)
            |
     +------+------+
     |             |
 [Span B]      [Span C] ←←←(Span C是Span A的子节点，ChildOf)
     |             |
 [Span D]      +---+-------+
               |           |
           [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                          ↑
                                          ↑
                                          ↑
                         (Span G在Span F后被调用，FollowsFrom)
```

有些情况下，使用下面这种基于时间轴的时序图可以更好地展现调用链。

```
单个Trace中Span间的时间关系
--|-------|-------|-------|-------|-------|-------|-------|-> time
 [Span A···················································]
   [Span B·····························································]
      [Span D·····························································]
    [Span C·····························································]
         [Span E·····]        [Span F··] [Span G··] [Span H··]
```

链路

Tracer接口用于创建Span（startSpan函数）、解析上下文（Extract函数）和透传上下文（Inject函数）。它具有以下能力：

- 创建一个新Span或者设置Span属性

```
/** 创建和开始一个span，返回一个span，span中包括操作名称和设置的选项。
** 例如:
**     创建一个无parentSpan的Span:
**     sp := tracer.StartSpan("GetFeed")
**    创建一个有parentSpan的Span
**     sp := tracer.StartSpan("GetFeed",opentracing.ChildOf(parentSpan.Context()))
**/
StartSpan(operationName string, opts ...StartSpanOption) Span
```

每个Span包含以下对象：

- Operation name：操作名称 （也可以称作Span name）。
- Start timestamp：起始时间。
- Finish timestamp：结束时间。
- Span tag：一组键值对构成的Span标签集合。键值对中，键必须为String，值可以是字符串、布尔或者数字类型。
- Span log：一组Span的日志集合。每次Log操作包含一个键值对和一个时间戳。键值对中，键必须为String，值可以是任意类型。
- SpanContext：Span上下文对象。每个SpanContext包含以下状态：
  - 要实现任何一个OpenTracing，都需要依赖一个独特的Span去跨进程边界传输当前调用链的状态（例如：Trace和Span的ID）。
  - Baggage Items是Trace的随行数据，是一个键值对集合，存在于Trace中，也需要跨进程边界传输。
- References（Span间关系）：相关的零个或者多个Span（Span间通过SpanContext建立这种关系）。

- 透传数据
  透传数据分为两步：
  i. 从请求中解析出SpanContext。

```
// Inject() takes the `sm` SpanContext instance and injects it for
// propagation within `carrier`. The actual type of `carrier` depends on
// the value of `format`.
/** 根据format参数从请求（Carrier）中解析出SpanContext（包括traceId、spanId、baggage）。
** 例如:
**  carrier := opentracing.HTTPHeadersCarrier(httpReq.Header)
**  clientContext, err := tracer.Extract(opentracing.HTTPHeaders, carrier)
**/
Extract(format interface{}, carrier interface{}) (SpanContext, error)
```

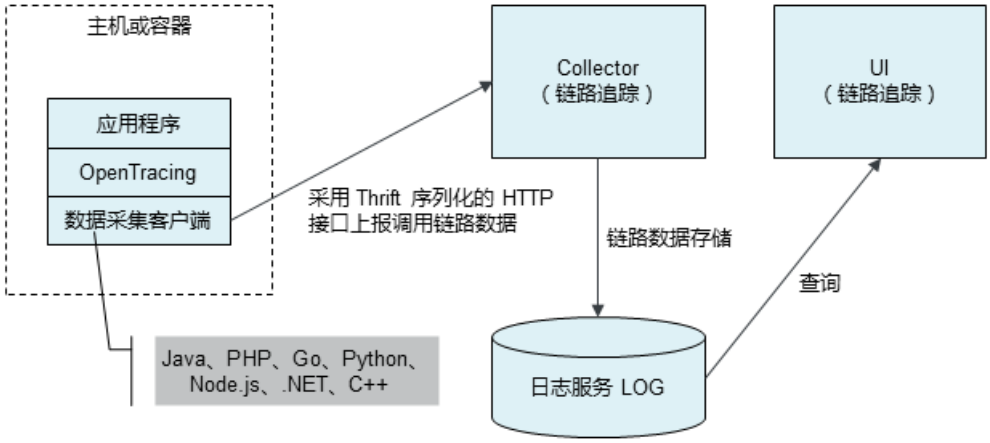
  ii. 将SpanContext注入到请求中。

```
/**
** 将SpanContext中的traceId，spanId，Baggage等根据format参数注入到请求中（Carrier），
** e.g
** carrier := opentracing.HTTPHeadersCarrier(httpReq.Header)
** err := tracer.Inject(span.Context(), opentracing.HTTPHeaders, carrier)
**/
Inject(sm SpanContext, format interface{}, carrier interface{}) error
```

数据是如何上报的？

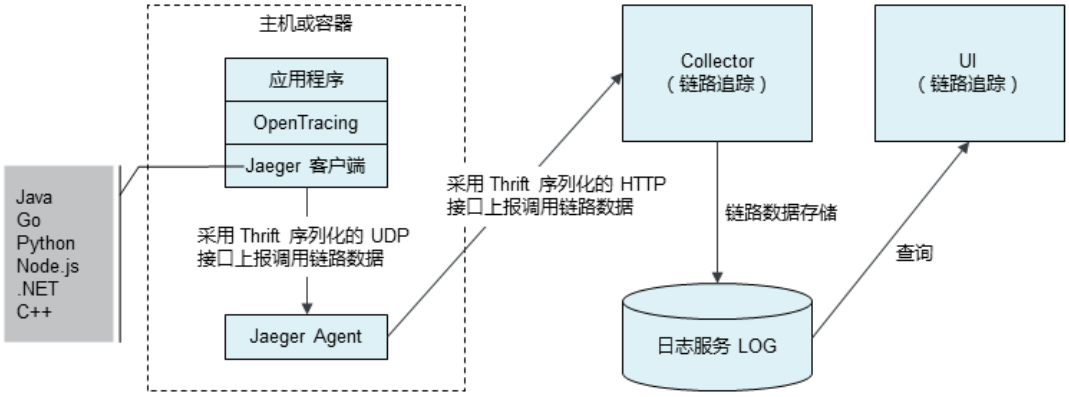
不通过Agent而直接上报数据的原理如下图所示。

直接上报数据



通过Agent上报数据的原理如下图所示。

通过Agent上报数据



3.产品计费

3.1. 关于欠费停服

欠费停服指服务因账号欠费而自动停止，处于欠费停服状态的服务不计费。本文说明与欠费停服相关的注意事项。

> **注意** 当服务欠费后有停止服务的风险，系统会提醒或通知您，请及时续费，避免对您的服务造成影响。

如果您账号的可用额度（含阿里云账号余额、代金券、优惠券等）小于待结算的账单，即视为账号欠费。账号欠费后停止计费，并且欠费24小时后停止服务，数据默认保留30天。续费后不会恢复已清理的数据，并且不会自动恢复服务，您必须在控制台手动开启服务。

3.2. 收费规则

您可以免费开通链路追踪Tracing Analysis，以及链路追踪Tracing Analysis中的日志服务SLS。
关于链路追踪Tracing Analysis通用收费项目和收费规则，请参见链路追踪Tracing Analysis收费标准。

付费模式

链路追踪Tracing Analysis采用后付费模式，您不需要提前支付费用。

付费方式

链路追踪Tracing Analysis的付费方式为按量付费。

扣费时间

链路追踪Tracing Analysis将会在次日00:00扣除前一天的费用。

3.3. 收费调整说明

链路追踪Tracing Analysis于2020年10月11日0点对售价模型做出全新的升级，将以更优惠和更灵活的方式提供服务。您可以采用原先计费方案，直到资源包消耗完毕，也可以直接在界面上升级到新版本计费。

基本概念

- 调用链（Trace）：一个调用链代表一个事务或者流程在（分布式）系统中的执行过程。
- Span：一次调用请求代表一个Span，每一个Span代表调用链中被命名并计时的连续性执行片段。
- 指标（Metric）：统计数据，例如应用、接口、数据库的请求数据、响应时间、异常数等。
- 调用链与Span的关系：在一个账号下TraceID相同的所有运动轨迹，视为一次调用链。一个调用链最多包含10个请求（Span），每个Span最大不超过2KB，超出部分将会丢弃。关于调用链与Span关系的更多信息，请参考链路追踪基本概念
- 调用链与指标（Metric）的关系：上报的调用链请求会聚合生成统计指标，一个调用链对应生成一个统计指标。

价格优势

链路追踪Tracing Analysis作为一款对标开源自建的产品，一直在追求更高的性价比。在大部分主流开源APM都仅存储7天数据的情况下，链路追踪默认30天的存储显得不够灵活。

为了让您更好的按需使用，链路追踪把计算和存储的价格进行了分割，同时提供实时调整存储时长的功能，让您可以按需调节自己数据的存储时长，从而更好的控制成本。

与开源自建成本的对比

客户类型	客户节点数	每日请求数	机器配置	开源成本（元/月）	链路追踪（元/月）
小型客户	80	2000万请求（请求复杂度：平均5个Span）	• 4台Elasticsearch（4CPU+16 G内存+1 T SSD硬盘） • 4台Collector（4CPU+8 G内存）	5836	764.8
中型客户	300	3亿请求（请求复杂度：平均7个Span）	• 8台Elasticsearch（8CPU+16 G内存+6 T SSD硬盘） • 8台Collector（4CPU+8 G内存）	22480	16065
大型客户	1000	10亿请求（请求复杂读：平均8个Span）	• 12台Elasticsearch（16CPU+64 G内存+21 T SSD硬盘） • 16台Collector（4CPU+8 G内存）	65088	64800

> **说明**
> - 客户节点数为ECS数量或者Docker数量。
> - 每日请求数参考Apache官方说明，1个Trace写入10 KB。
> - 机器配置按照统计数据存15天，全量明细数据存7天计算。
> - 链路追踪按照存储的应用总请求数（每天请求数*时长）计费，采用开源探针，其他组件由阿里云维护。

新旧售价模型对比

- 旧版：按照请求（Span）次数收费，所有数据存储30天时间。
- 新版：将原来30天一口价细分为计算和存储费用，计费明细如下：
  - 调用链计算（读写和聚合量）费用：每上报百万条Span 0.09元。
  - 调用链存储费用：每存储百万条Span 0.02元。
  - 统计指标存储费用：每存储百万条指标0.01元。

> 说明
>
> - 调用链计算费用根据实际上报的调用链数决定，您可以在链路追踪控制台的**集群配置 > 采样存储**页面通过设置采样率调整上报量。
>   示例：每天上报1百万调用链。平均每天上报约1000万Span，费用为10（10个百万Span）*0.09=0.9元。
> - 调用链存储费用按实际存储量计算，每天统计总的存储量，和存储天数有关。您可以在链路追踪控制台的**集群配置 > 集群配置**页面调整适当的存储天数。
>   示例：每天上报1百万调用链并存储15天。平均每天上报约1000万Span，存储15天总的Span量为15（天）* 10（10个百万Span）=150个百万条，每天的存储费用为总调用链量150（百万）*0.02（单价）=3元。
> - 统计指标存储费用按实际存储量计算，每天统计总的存储量，和存储天数有关。您可以在的**集群配置 > 集群配置**页面调整适当的存储天数。
>   示例：每天上报1百万调用链并存储15天。平均每天产生约100万指标，存储15天总的指标量为15（天）* 1（1个百万指标）=15个百万条，每天的存储费用为15（百万）* 0.01=0.15元，每天需要0.15元。
> - 计费方式：
>   - 链路追踪按天计费，每天0点计费前一天费用。
>   - 调用链计算费用只计算当天的流量。

新版售价模型案例

案例一：每天上报4亿调用链（约40亿Span、4亿指标）数据。

- 方案一：调用链数据和统计指标数据全量存储30天。
  调用链每天计算费用：4000（4000个百万Span）* 0.09（百万Span计算单价）=360元/天
  调用链每天存储费用：4000（4000个百万Span）*30（天）* 0.02（百万Span存储单价）=2400元/天
  指标每天存储费用：400（400个百万指标） * 30（天）* 0.01（百万指标存储单价）=120元/天
  总计：2880元/天
- 方案二：调用链数据存储7天，统计指标存储30天。
  调用链每天计算费用：4000（4000个百万Span）* 0.09（百万Span计算单价）=360元/天
  调用链每天存储费用：4000（4000个百万Span）*7（天）* 0.02（百万Span存储单价）=560元/天
  指标每天存储费用：400（400个百万指标） *30（天）* 0.01（百万指标存储单价）=120元/天
  总计：1040元/天

案例二：每天上报1千万调用链（约1亿Span、1千万指标）数据。

调用链数据存储7天，统计指标存储30天。

调用链每天计算费用：100（100个百万Span）*0.09（百万Span计算单价）=9元/天

调用链每天存储费用：100（100个百万Span）*7（天）*0.02（百万Span存储单价）=14元/天

指标每天存储费用：10（10个百万指标） *30（天）*0.01（百万指标单价）=3元/天

总计：26元/天

升级新版本计费

- 没有购买过资源包的客户，登录链路追踪Tracing Analysis控制台，在**集群配置 > 集群配置**页签单击**保存**，即可完成升级。第二天可以在控制台首页右上角查看资源使用情况来预估新版的计费。
- 购买过资源包的客户，需要先对资源包进行退费（提交工单申请资源包退费），然后才可以升级。

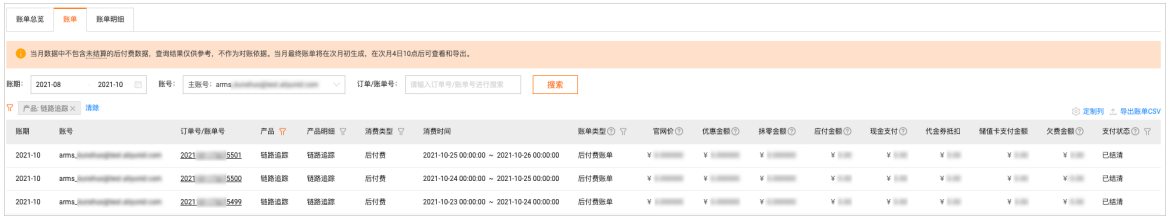
3.4. 查看消费明细

本文将介绍如何查看链路追踪的费用账单和资源使用情况。

查看费用账单

1. 登录用户中心。
2. 在左侧菜单栏选择**费用账单 > 费用账单**
3. 在**费用账单**页面单击**账单**页签。
4. 在**账单**页签选择**账期**和**订单/账单号**等过滤条件，完成后单击**搜索**。在账单列表中单击产品右侧的图标，选择**链路追踪**。

   单击产品明细、消费类型、账单类型和支付状态右侧的图标，可进一步筛选您的消费账单。



查看资源使用情况

1. 登录链路追踪Tracing Analysis控制台。
2. 在**概览**页面右侧**资源状况**区域，单击**查看历史资源使用**。
   在**资源使用详情**页面可以查看每日的资源使用情况。

4.快速入门

4.1. 快速上手链路追踪

本文以Java应用为例，介绍从开通链路追踪服务及相关依赖服务并授权，到将应用接入链路追踪的流程，帮助您快速上手链路追踪。

前提条件

- 阿里云账号注册流程
- 开通链路追踪
- 开通日志服务
- 开通访问控制

授权链路追踪读写您的日志服务数据

1. 登录链路追踪Tracing Analysis控制台。
2. 在**提示**页面上，单击**授权**，授权链路追踪读写您的日志服务。
3. 在**提示**对话框，单击**确认**。

通过Jaeger客户端上报Java应用数据

本文以Jaeger客户端上报Java应用数据为例，介绍如何上报数据。通过其他客户端上报数据和上报其他语言应用数据的方法，请参见文末的相关文档。

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或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**页面里的上相应客户端和地域的接入点。

```
// 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.Config
uration.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().withSend
er(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. 等待2分钟~3分钟后，登录链路追踪Tracing Analysis控制台，在**应用列表**页面查看上报的链路数据。

后续步骤

将应用数据上报至链路追踪控制台后，您可以在链路追踪控制台执行以下操作：

- 查看应用性能关键指标和拓扑图
- 查看应用详情
- 查看接口调用情况
- 查看SQL性能分析
- 管理应用和标签

链路追踪支持将Java、Go、Python、JS、.NET、C++等语言的应用数据上报至控制台。支持的上报数据客户端包括Jaeger、Zipkin和SkyWalking。请在下方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 Go应用 .NET应用

SkyWalking

Java应用

5.准备工作

5.1. 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、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 Go应用 .NET应用

SkyWalking
Java应用

5.2. 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**链路追踪**页面上选中**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**立即开通/登录**，然后在新页面上单击**开通日志服务**。
4. 在**日志服务**页面上选中**日志服务服务协议**，并单击**立即开通**。

开通访问控制RAM

完成实名认证的阿里云账号即可使用RAM访问控制，按照以下步骤为阿里云账号进行实名认证。

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**RAM管理控制台**，然后在**欢迎使用RAM访问控制**页面上单击**前往认证**。
4. 根据页面提示完成实名认证。具体操作，请参见个人实名认证。

授权链路追踪读写您的日志服务数据

1. 登录链路追踪Tracing Analysis控制台。
2. 在**提示**页面上，单击**授权**，授权链路追踪读写您的日志服务。
3. 在**提示**对话框，单击**确认**。

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。
准备工作概述

5.3. 链路追踪接入和鉴权说明

链路追踪服务提供了ARMS OpenTelemetry Collector、开源OpenTelemetry SDK/Collector、Jaeger SDK/Agent、Zipkin和Skywalking客户端的接入方式。本文介绍了各地域接入点域名以及不同端口对应的数据上报协议。

直接上报

公有云各区域接入点域名

> **说明** 如果应用部署于阿里云生产环境，请选择内网接入点，否则选择公网接入点。

地域	Region ID	阿里云内网Endpoint	公网Endpoint
华东1（杭州）	cn-hangzhou	tracing-analysis-dc-hz-internal.aliyuncs.com	tracing-analysis-dc-hz.aliyuncs.com
华东2（上海）	cn-shanghai	tracing-analysis-dc-sh-internal.aliyuncs.com	tracing-analysis-dc-sh.aliyuncs.com
华北1（青岛）	cn-qingdao	tracing-analysis-dc-qd-internal.aliyuncs.com	tracing-analysis-dc-qd.aliyuncs.com
华北2（北京）	cn-beijing	tracing-analysis-dc-bj-internal.aliyuncs.com	tracing-analysis-dc-bj.aliyuncs.com
华北 3（张家口）	cn-zhangjiakou	tracing-analysis-dc-zb-internal.aliyuncs.com	tracing-analysis-dc-zb.aliyuncs.com

地域	Region ID	阿里云内网Endpoint	公网Endpoint
华北5（呼和浩特）	cn-huhehaote	tracing-cn-huhehaote-internal.arms.aliyuncs.com	tracing-cn-huhehaote.arms.aliyuncs.com
华北6（乌兰察布）	cn-wulanchabu	tracing-cn-wulanchabu-internal.arms.aliyuncs.com	tracing-cn-wulanchabu.arms.aliyuncs.com
华南1（深圳）	cn-shenzhen	tracing-analysis-dc-sz-internal.aliyuncs.com	tracing-analysis-dc-sz.aliyuncs.com
华南2（河源）	cn-heyuan	tracing-cn-heyuan-internal.arms.aliyuncs.com	tracing-cn-heyuan.arms.aliyuncs.com
华南3（广州）	cn-guangzhou	tracing-cn-guangzhou-internal.arms.aliyuncs.com	tracing-cn-guangzhou.arms.aliyuncs.com
西南1（成都）	cn-chengdu	tracing-cn-chengdu-internal.arms.aliyuncs.com	tracing-cn-chengdu.arms.aliyuncs.com
中国（香港）	cn-hongkong	tracing-analysis-dc-hk-internal.aliyuncs.com	tracing-analysis-dc-hk.aliyuncs.com
日本（东京）	ap-northeast-1	tracing-analysis-dc-jp-internal.aliyuncs.com	tracing-analysis-dc-jp.aliyuncs.com
亚太东南（新加坡）	ap-southeast-1	tracing-analysis-dc-sg-internal.aliyuncs.com	tracing-analysis-dc-sg.aliyuncs.com
澳大利亚（悉尼）	ap-southeast-2	tracing-ap-southeast-2-internal.arms.aliyuncs.com	tracing-ap-southeast-2.arms.aliyuncs.com
马来西亚（吉隆坡）	ap-southeast-3	tracing-ap-southeast-3-internal.arms.aliyuncs.com	tracing-ap-southeast-3.arms.aliyuncs.com
印度尼西亚（雅加达）	ap-southeast-5	tracing-analysis-dc-indonesia-internal.aliyuncs.com	tracing-analysis-dc-indonesia.aliyuncs.com
德国（法兰克福）	eu-central-1	tracing-analysis-dc-frankfurt-internal.aliyuncs.com	tracing-analysis-dc-frankfurt.aliyuncs.com
英国（伦敦）	eu-west-1	tracing-analysis-dc-lundun-internal.aliyuncs.com	tracing-analysis-dc-lundun.aliyuncs.com
美国（硅谷）	us-west-1	tracing-analysis-dc-usw-internal.aliyuncs.com	tracing-analysis-dc-usw.aliyuncs.com
美国（弗吉尼亚）	us-east-1	tracing-us-east-1-internal.arms.aliyuncs.com	tracing-us-east-1.arms.aliyuncs.com
印度（孟买）	ap-south-1	tracing-ap-south-1-internal.arms.aliyuncs.com	tracing-ap-south-1.arms.aliyuncs.com

不同端口支持的协议

端口号	协议	说明

80	• OpenTelemetry HTTP
	• Jaeger HTTP
	• Jaeger Remote Sampling
	• Zipkin HTTP
		通过URL后缀区分不同协议。
		• OpenTelemetry HTTP：/api/otlp/traces
		• Jaeger HTTP：/api/traces
		• Jaeger Remote Sampling：/api/sampling
		• Zipkin HTTP：
		  ○ v1：/api/v1/spans
		  ○ v2：/api/v2/spans

443	• OpenTelemetry HTTPS
	• Jaeger HTTPS
	• Jaeger Remote Sampling
	• Zipkin HTTPS
		通过URL后缀区分不同协议。
		• OpenTelemetry HTTPS：/api/otlp/traces
		• Jaeger HTTPS：/api/traces
		• Jaeger Remote Sampling：/api/sampling
		• Zipkin HTTPS：
		  ○ v1：/api/v1/spans
		  ○ v2：/api/v2/spans

1883	• Jaeger gRPC • Skywalking gRPC	无
8000	Skywalking v8 gRPC	无
8090	OpenTelemetry gRPC	无

通过ARMS OpenTelemetry Collector转发

说明 对于ACK集群应用，建议通过ARMS OpenTelemetry Collector上报应用数据。ARMS OpenTelemetry Collector支持本地集群内的链路采样与指标无损统计，在降低链路传输、存储成本的同时，不影响监控或告警指标的准确性。更多信息，请参见ARMS OpenTelemetry Collector。

默认Receivers配置说明

组件安装后默认配置的Receivers包括OTLP、Jaeger和Zipkin，端口与开源保持一致。

Receivers	protocols	port
OTLP	gRPC	4317
	http	4318
	gRPC	14250

Jaeger Receivers	protocols	port
	thrift_http	14268
Zipkin	无	9411

将应用数据上报到ARMS OpenTelemetry Collector

组件安装后会在arms-prom命名空间下默认创建名称为otel-collector-service的Service资源，用于将Receivers中不同组件的相应端口进行暴露。Trace上报的Endpoint格式为 `otel-collector-service.arms-prom:<port>` ，其中 `otel-collector-service` 是ARMS OpenTelemetry Collector的Service名称， `arms-prom` 是其所在的命名空间。

示例：将生成的Traces通过OTLP的gRPC协议上报到ARMS OpenTelemetry Collector中。

```
func initProvider() func() {
    ctx := context.Background()
    endpoint := "otel-collector-service.arms-prom:4317"
    if !ok {
        log.Fatalf("Cannot init OpenTelemetry, exit")
        os.Exit(-1)
    }
    metricClient := otlpmetricgrpc.NewClient(
        otlpmetricgrpc.WithInsecure(),
        otlpmetricgrpc.WithEndpoint(endpoint))
    metricExp, err := otlpmetric.New(ctx, metricClient)
    handleErr(err, "Failed to create the collector metric exporter")
    pusher := controller.New(
        processor.NewFactory(
            simple.NewWithExactDistribution(),
            metricExp,
        ),
        controller.WithExporter(metricExp),
        controller.WithCollectPeriod(2*time.Second),
    )
    global.SetMeterProvider(pusher)
    err = pusher.Start(ctx)
    handleErr(err, "Failed to start metric pusher")
    serviceName, ok := common.GetClientServiceName()
    if !ok {
        serviceName = common.ClientServiceName
    }
    headers := map[string]string{"Authentication": ""}
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint(otelAgentAddr),
        otlptracegrpc.WithHeaders(headers), // 鉴权信息
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
```

```
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // the service name used to display traces in backends
            semconv.ServiceNameKey.String(serviceName),
        ),
    )
    handleErr(err, "failed to create resource")
    bsp := sdktrace.NewBatchSpanProcessor(traceExp)
    tracerProvider := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithResource(res),
        sdktrace.WithSpanProcessor(bsp),
    )
    // set global propagator to tracecontext (the default is no-op).
    otel.SetTextMapPropagator(propagation.TraceContext{})
    otel.SetTracerProvider(tracerProvider)
    log.Println("OTEL init success")
    return func() {
        cxt, cancel := context.WithTimeout(ctx, time.Second)
        defer cancel()
        if err := traceExp.Shutdown(cxt); err != nil {
            otel.Handle(err)
        }
        // pushes any last exports to the receiver
        if err := pusher.Stop(cxt); err != nil {
            otel.Handle(err)
        }
    }
}
```

获取鉴权Token

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

1. 登录链路追踪控制台，在页面顶部选择需要接入的地域。
2. 在**接入点信息**页签的**集群信息**区域打开**显示Token**开关。
3. 在**客户端采集工具**区域单击需要使用的链路数据采集客户端。
4. 在下方表格的**相关信息**列中，单击接入点信息末尾的复制图标。

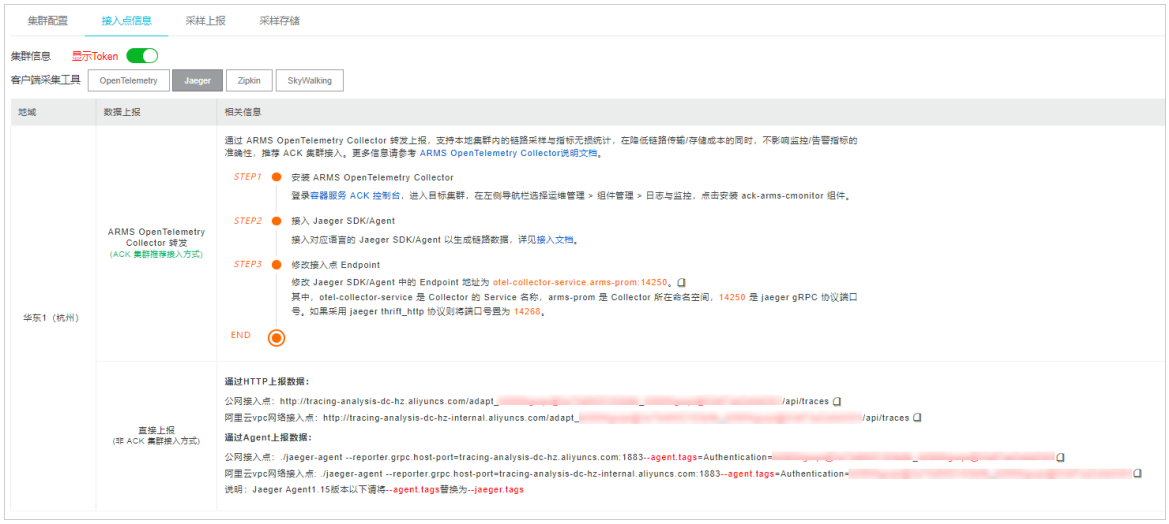
- OpenTelemetry HTTP、Zipkin HTTP和Jaeger HTTP：将Token作为HTTP URL的一部分。
- Skywalking gRPC、OpenTelemetry gRPC和Jaeger gRPC：将 `Authentication=<token>` 作为gRPC的 `metadata` 提供给服务端鉴权。

5.4. 安装Jaeger Agent

本文介绍安装Jaeger Agent的方法。

步骤一：在链路追踪控制台获取接入点信息

1. 登录链路追踪控制台，在左侧导航栏，单击**集群配置**。
2. 在**集群配置**页面顶部选择地域，然后单击**接入点信息**页签，在**集群信息**区域打开**显示Token**开关。
3. 在**客户端采集工具**区域单击需要使用的链路数据采集客户端。
4. 在下方表格的**相关信息**列中，单击接入点信息末尾的复制图标。



> **说明** 如果应用部署于阿里云生产环境，则选择私网接入点，否则选择公网接入点。

步骤二：下载并启动Jaeger Agent

ECS环境

若您使用ECS，则可以通过如下方式启动Jaeger Agent。

1. 下载Jaeger Agent安装包并完成解压。

   > **说明** 建议使用最新Jaeger Agent版本。

2. 使用以下命令行启动Jaeger Agent。

```
nohup ./jaeger-agent --reporter.grpc.host-port=<endpoint> --agent.tags=<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` 。
> - `<endpoint>` 为链路追踪控制台**概览**页面上相应客户端和相应地域的接入点。
> - `<token>` 为步骤一中获取的接入点信息。

Docker环境

若您使用Docker部署，则建议使用容器方式启动Jaeger Agent，以减少您的运维成本。启动命令如下：

```
docker run \
  --rm \
  -p5775:5775/udp \
  -p6831:6831/udp \
  -p6832:6832/udp \
  -p5778:5778/tcp \
  jaegertracing/jaeger-agent:<version> \
  --reporter.grpc.host-port=<endpoint> \
  --agent.tags=<token>
```

> 说明 在上述启动命令中：
> - 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` 。
> - `<version>` 为Jaeger Agent版本，例如1.23。其他可用版本，请参见Docker Hub。
> - `<endpoint>` 为链路追踪控制台**概览**页面上相应客户端和相应地域的接入点。
> - `<token>` 为步骤一中获取的接入点信息。

5.5. OpenTelemetry接入说明

使用OpenTelemetry SDK接入Trace数据时，可以通过直接发送或通过OpenTelemetry Collector转发两种方式上报数据到链路追踪服务端。

前提条件

在控制台获取鉴权Token，具体操作，请参见获取鉴权Token。

直接上报

如果您的应用使用了OpenTelemetry SDK，可以通过OpenTelemetry gRPC协议直接向链路追踪服务端发送数据，您只需要配置接入点信息以及鉴权信息。

- 接入点信息：将前提条件中获取的接入点记为<endpoint>。
- 鉴权信息：将前提条件中 `Authentication: <token>` 添加为gRPC的Header。

以Java语言为例：

```
SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
                .addSpanProcessor(BatchSpanProcessor.builder(OtlpGrpcSpanExporter.builder()
                        .setEndpoint("<endpoint>")
                        .addHeader("Authentication", "<token>")
                        .build()).build())
                .build();
```

说明

- 将 `<endpoint>` 替换为您上报区域对应的Endpoint，例如：`http://tracing-analysis-dc-bj.aliyuncs.com:8090`。
- 将 `<token>` 替换为前提条件中获取的Token，例如：`b590lhguqs@3a7xxxxxxx9b_b590lhguqs@53dxxxxx8301`。

通过开源OpenTelemetry Collector转发

如果您的应用使用了OpenTelemetry SDK和OpenTelemetry Collector，将Endpoint配置为本地部署的OpenTelemetry Collector地址后，无需配置鉴权信息，只需在OpenTelemetry Collector配置OTLP Exporter（包含Endpoint以及鉴权信息）向链路追踪服务端上报数据。

1. 下载OpenTelemetry Collector。
2. 参考OpenTelemetry官方示例创建*otel-config.yaml*配置文件，并按照下方代码修改 `exporters` 部分。

```
exporters:
  otlp:
    endpoint: <endpoint>:8090
    tls:
      insecure: true
    headers:
      Authentication: <token>
```

说明

- 将 `<endpoint>` 替换为您上报区域对应的Endpoint，例如：`http://tracing-analysis-dc-bj.aliyuncs.com:8090`。
- 将 `<token>` 替换为前提条件中获取的Token，例如：`b590lhguqs@3a7xxxxxxx9b_b590lhguqs@53dxxxxx8301`。

完整示例如下：

```
exporters:
  otlp:
    endpoint: tracing-analysis-dc-bj.aliyuncs.com
    tls:
      insecure: true
    headers:
      Authentication: b590lhguqs@3a7xxxxxxx9b_b590lhguqs@53dxxxxx8301
```

3. 执行以下命令启动OpenTelemetry Collector。

```
./ocb_0.44.0_linux_amd64 --config="./otel-config.yaml"
```

5.6. 开始监控Java应用

5.6.1. 通过OpenTelemetry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，需要将应用数据上报至链路追踪。本文介绍如何为应用埋点并上报链路数据。

前提条件

获取接入点信息 ❯

使用OpenTelemetry Java Agent自动埋点

OpenTelemetry Java Agent提供了无侵入的接入方式，支持数十种框架。更多信息，请参见OpenTelemetry官方文档。

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

```
-javaagent:/path/to/opentelemetry-javaagent.jar     //请将路径修改为您文件下载的实际地址。
-Dotel.resource.attributes=service.name=<appName>      //<appName> 为应用名。
-Dotel.exporter.otlp.headers=Authentication=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
```

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

```
-javaagent:/Users/carpela/Downloads/opentelemetry-javaagent.jar
-Dotel.resource.attributes=service.name=ot-java-agent-sample
-Dotel.exporter.otlp.headers=Authentication=b590xxxxuqs@3a75d95xxxxx9b_b59xxxxguqs@53dxxxx2afe8301
-Dotel.exporter.otlp.endpoint=http://tracing-analysis-dc-bj:8090
```

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

使用OpenTelemetry Java SDK手动埋点

OpenTelemetry Java SDK是OpenTelemetry Java Agent实现的基础，提供了丰富的自定义能力。当OpenTelemetry Java Agent的埋点不满足您的场景或者需要增加一些自定义业务埋点时，可以使用以下方式接入。

1. 引入Maven POM依赖。

```
<dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-api</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-sdk-trace</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-exporter-otlp</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-sdk</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.opentelemetry</groupId>
                <artifactId>opentelemetry-bom</artifactId>
                <version>1.9.0</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

2. 获取OpenTelemetry Tracer。

```
SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
                .addSpanProcessor(BatchSpanProcessor.builder(OtlpGrpcSpanExporter.build
er()
                        .setEndpoint("<endpoint>")
                        .addHeader("Authentication", "<token>")
                        .build()).build())
                .build();
OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
                .setTracerProvider(sdkTracerProvider)
                .setPropagators(ContextPropagators.create(W3CTraceContextPropagator.get
Instance()))
                .buildAndRegisterGlobal();
Tracer tracer = openTelemetry.getTracer("instrumentation-library-name", "1.0.0");
```

3. 参考OpenTelemetry官方文档修改以下代码。

```
package com.alibaba.arms.brightroar.console.controller;
import com.alibaba.arms.brightroar.console.service.UserService;
import com.alibaba.arms.brightroar.console.util.OpenTelemetrySupport;
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.SpanContext;
```

```
import io.opentelemetry.api.trace.StatusCode;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Context;
import io.opentelemetry.context.Scope;
import io.opentelemetry.extension.annotations.SpanAttribute;
import io.opentelemetry.extension.annotations.WithSpan;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/**
 * 参考文档:
 * 1. https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/do
cs/manual-instrumentation.md#creating-spans-manually-with-a-tracer
 * 2. https://opentelemetry.io/docs/java/manual_instrumentation/
 */
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;
    private ExecutorService es = Executors.newFixedThreadPool(5);
    // 第三种：获得Tracer纯手工埋点
    private void biz() {
        Tracer tracer = OpenTelemetrySupport.getTracer();
        Span span = tracer.spanBuilder("biz (manual)")
                .setParent(Context.current().with(Span.current())) // 可选，自动设置
                .startSpan();
        try (Scope scope = span.makeCurrent()) {
            span.setAttribute("biz-id", "111");
            es.submit(new Runnable() {
                @Override
                public void run() {
                    Span asyncSpan = tracer.spanBuilder("async")
                            .setParent(Context.current().with(span))
                            .startSpan();
                    try {
                        Thread.sleep(1000L); // some async jobs
                    } catch (Throwable e) {
                    }
                    asyncSpan.end();
                }
            });
            Thread.sleep(1000); // fake biz logic
            System.out.println("biz done");
            OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
            openTelemetry.getPropagators();
        } catch (Throwable t) {
            span.setStatus(StatusCode.ERROR, "handle biz error");
        } finally {
            span.end();
        }
    }
```

```
}
```

4. 通过修改Java启动的VM参数上报链路数据。

```
-javaagent:/path/to/opentelemetry-javaagent.jar    //请将路径修改为您文件下载的实际地址。
-Dotel.resource.attributes=service.name=<appName>     //<appName> 为应用名。
-Dotel.exporter.otlp.headers=Authentication=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
```

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

     ```
     -javaagent:/Users/carpela/Downloads/opentelemetry-javaagent.jar
     -Dotel.resource.attributes=service.name=ot-java-agent-sample
     -Dotel.exporter.otlp.headers=Authentication=b590xxxxuqs@3a75d95xxxxx9b_b59xxxxguqs@53dxxxx2afe8301
     -Dotel.exporter.otlp.endpoint=http://tracing-analysis-dc-bj:8090
     ```

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

5. 启动应用。

   在链路追踪Tracing Analysis控制台的**应用列表**页面选择新创建的应用，查看链路数据。

同时使用Java Agent和Java SDK埋点

您可以在使用Java Agent获得自动埋点能力的同时，使用Java SDK添加自定义业务埋点。

1. 通过Java Agent方式接入应用。
2. 在Java SDK的基础上新增Maven依赖。

```
<dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-extension-annotations</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
            <version>1.9.0-alpha</version>
        </dependency>
```

> **说明** 其中 `opentelemetry-sdk-extension-autoconfigure` 完成了SDK的自动配置，将Java Agent的配置传递到Java SDK中。

3. 获得OpenTelemetry Tracer。

```
OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
Tracer tracer = openTelemetry.getTracer("instrumentation-library-name", "1.0.0");
```

4. 修改以下代码，建议使用代码中的第一种和第二种方式。更多信息，请参见OpenTelemetry官方文档。

```
package com.alibaba.arms.brightroar.console.controller;
import com.alibaba.arms.brightroar.console.service.UserService;
import com.alibaba.arms.brightroar.console.util.OpenTelemetrySupport;
```

```
import com.alibaba.arms.brightroad.console.util.OpenTelemetrySupport;
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.SpanContext;
import io.opentelemetry.api.trace.StatusCode;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Context;
import io.opentelemetry.context.Scope;
import io.opentelemetry.extension.annotations.SpanAttribute;
import io.opentelemetry.extension.annotations.WithSpan;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/**
 * 参考文档:
 * 1. https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/manual-instrumentation.md#creating-spans-manually-with-a-tracer
 * 2. https://opentelemetry.io/docs/java/manual_instrumentation/
 */
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;
    private ExecutorService es = Executors.newFixedThreadPool(5);
    // 第三种：获得Tracer纯手工埋点
    private void biz() {
        Tracer tracer = OpenTelemetrySupport.getTracer();
        Span span = tracer.spanBuilder("biz (manual)")
                .setParent(Context.current().with(Span.current())) // 可选，自动设置
                .startSpan();
        try (Scope scope = span.makeCurrent()) {
            span.setAttribute("biz-id", "111");
            es.submit(new Runnable() {
                @Override
                public void run() {
                    Span asyncSpan = tracer.spanBuilder("async")
                            .setParent(Context.current().with(span))
                            .startSpan();
                    try {
                        Thread.sleep(1000L); // some async jobs
                    } catch (Throwable e) {
                    }
                    asyncSpan.end();
                }
            });
            Thread.sleep(1000); // fake biz logic
            System.out.println("biz done");
            OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
            openTelemetry.getPropagators();
        } catch (Throwable t) {
            span.setStatus(StatusCode.ERROR, "handle biz error");
```

```
        } finally {
            span.end();
        }
    }
    // 第二种：基于标签手工埋点
    @WithSpan
    private void child(@SpanAttribute("user.type") String userType) {
        System.out.println(userType);
        biz();
    }
    // 第一种：自动埋点，基于API手工添加信息
    @RequestMapping("/async")
    public String async() {
        System.out.println("UserController.async -- " + Thread.currentThread().getId())
;
        Span span = Span.current();
        span.setAttribute("user.id", "123456");
        userService.async();
        child("vip");
        return "async";
    }
}
```

5. 通过修改Java启动的VM参数上报链路数据。

```
-javaagent:/path/to/opentelemetry-javaagent.jar     //请将路径修改为您文件下载的实际地址。
-Dotel.resource.attributes=service.name=<appName>      //<appName> 为应用名。
-Dotel.exporter.otlp.headers=Authentication=<token>
-Dotel.exporter.otlp.endpoint=<endpoint>
```

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

     ```
     -javaagent:/Users/carpela/Downloads/opentelemetry-javaagent.jar
     -Dotel.resource.attributes=service.name=ot-java-agent-sample
     -Dotel.exporter.otlp.headers=Authentication=b590xxxxuqs@3a75d95xxxxx9b_b59xxxxguqs@53
     dxxxx2afe8301
     -Dotel.exporter.otlp.endpoint=http://tracing-analysis-dc-bj:8090
     ```

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

6. 启动应用。

   在链路追踪Tracing Analysis控制台的**应用列表**页面选择新创建的应用，查看链路数据。

相关文档

- Open Telemetry官方文档

5.6.2. 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**集群配置**页面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.Config
uration.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().withSend
er(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();
```

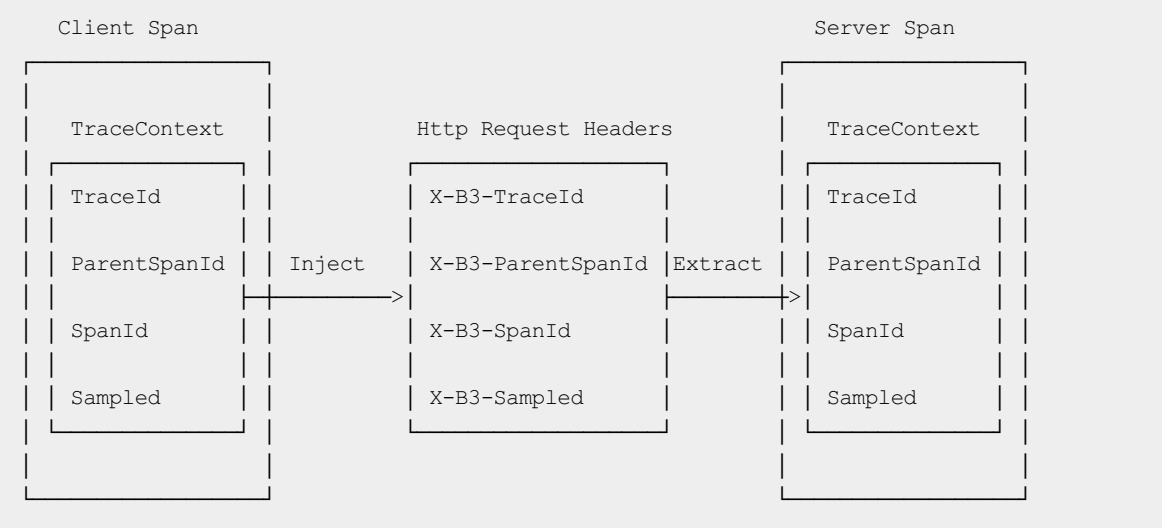
5. （可选）上一步用于记录请求的根操作，如果需要记录请求的上一步和下一步操作，则需要传入上下文。

```
Tracer tracer = GlobalTracer.get();
Span parentspan = tracer.activeSpan();
Tracer.SpanBuilder spanBuilder = tracer.buildSpan("childSpan").withTag("myTag", "spanSe
cond");
if (parentspan !=null) {
    spanBuilder.asChildOf(parentspan).start();
}
Span childSpan = spanBuilder.start();
Scope scope = tracer.scopeManager().activate(childSpan); // 请求开始执行一次
// 业务逻辑。 可以执行多次 buildSpan
childSpan.finish();
tracer.activeSpan().setTag("methodName", "testCall");
// 请求结束执行一次
scope.close();
```

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 shoul
d 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 TextM
apExtractAdapter(request.getAttachments()));
        if (spanContext !=null) {
            spanBuilder.asChildOf(spanContext);
        }
    } catch (Exception e) {
        spanBuilder.withTag("Error", "extract from request fail, error msg:" + e.ge
tMessage());
    }
    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**集群配置**页面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("springF
rontend");
    io.jaegertracing.Configuration.SenderConfiguration sender = new io.jaegertracing.Co
nfiguration.SenderConfiguration();
    sender.withEndpoint("<endpoint>");
    config.withSampler(new io.jaegertracing.Configuration.SamplerConfiguration().withTy
pe("const").withParam(1));
    config.withReporter(new io.jaegertracing.Configuration.ReporterConfiguration().with
Sender(sender).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 ServerTracingIntercept
or();
            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 ClientTracingIntercept
or();
            blockingStub = GreeterGrpc.newBlockingStub(tracingInterceptor.intercept(cha
nnel));
        }
    }
```

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。

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

5.6.3. 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>
```

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(report
er).build();
```

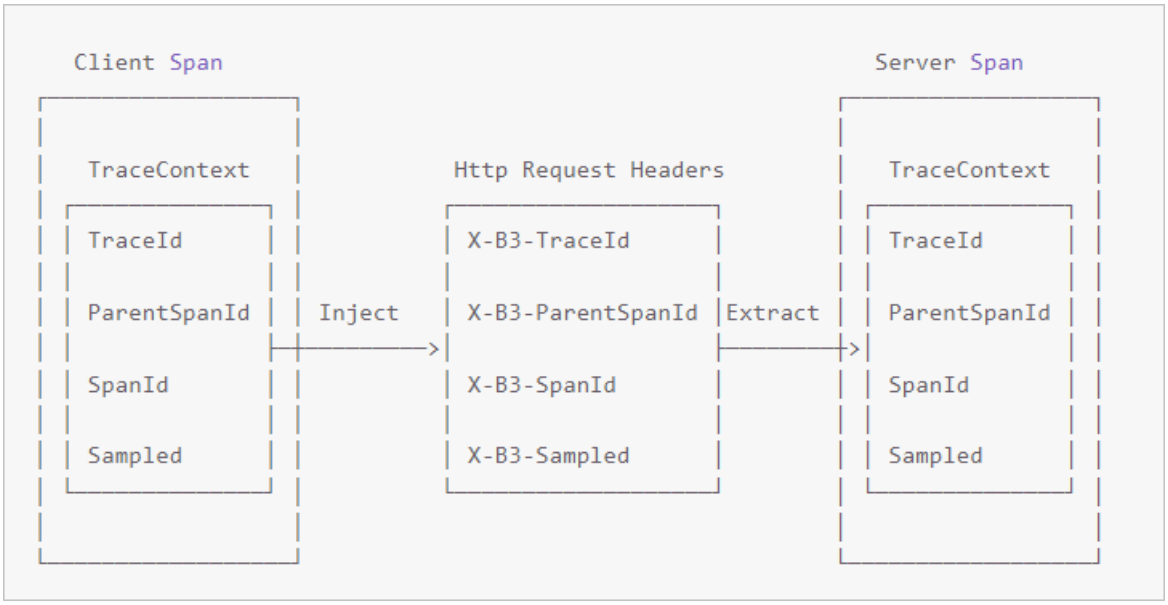
3. 构建Span和Child Span。

```
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().curren
tSpan().context());
        Span childSpan =  tracing.tracer().currentSpan();
        childSpan.tag("key", "secondBiz");
        childSpan.finish();
        System.out.println("end tracing,id:" + childSpan.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"/>
    </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.DefaultAnnotationHandlerM
apping">
    <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, "us
er-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 conf
iguration
  @Bean BeanPostProcessor connectionFactoryDecorator(final BeanFactory beanFactory) {
    return new BeanPostProcessor() {
      @Override public Object postProcessBeforeInitialization(Object bean, String beanN
ame) {
        return bean;
      }
      @Override public Object postProcessAfterInitialization(Object bean, String beanNa
me) {
        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(HttpTraci
ng.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 addres
s
     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
```

3. 发起HTTP请求，例如 `http://localhost:3380/traced` 。

> **说明** 更多请求路径，请参见Demo中 `com.alibaba.apm.SampleController` 下的方法。

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。

相关文档

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

5.6.4. 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，日志文件默认输出到 `/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。

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_OP
TS
```

- 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：

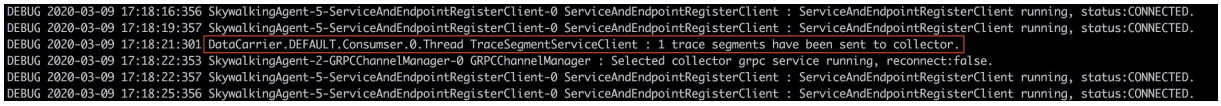
```
--exec    # 去掉前面的井号取消注释。
-javaagent:<skywalking-agent-path>
```

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。

相关文档

- SkyWalking官网
- 下载SkyWalking
- 部署SkyWalking Java Agent

5.7. 开始监控Go应用

5.7.1. 通过OpenTelemetry上报Go应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。本文介绍如何通过OpenTelemetry Go SDK上报Go应用数据。

前提条件

获取接入点信息 ❯

背景信息

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

示例Demo

oltp-exporter

使用gRPC协议上报

1. 添加OpenTelemetry Go依赖。

```
go get go.opentelemetry.io/otel
go get go.opentelemetry.io/otel/trace
go get go.opentelemetry.io/otel/sdk
go get go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc # otlp grpc
```

2. 初始化OpenTelemetry Go SDK。

   您可以选择通过OpenTelemetry SDK直接上报或通过开源OpenTelemetry Collector转发。

   - 如果选择直接上报，请将otelAgentAddr和xtraceToken替换为前提条件获取的接入点和鉴权Token。
   - 如果选择使用OpenTelemetry Collector转发，请将otelAgentAddr替换为您本地部署的服务地址，并删除Header信息。

```
func initProvider() func() {
    ctx := context.Background()
    otelAgentAddr, xtraceToken, ok := common.ObtainXTraceInfo()
    if !ok {
        log.Fatalf("Cannot init OpenTelemetry, exit")
        os.Exit(-1)
    }
    headers := map[string]string{"Authentication": xtraceToken}   //将xtraceToken替换为前提条件中获取的鉴权Token。
    traceClient := otlptracegrpc.NewClient(
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint(otelAgentAddr),    //将otelAgentAddr替换为前提条件中获取的接入点。
        otlptracegrpc.WithHeaders(headers),
        otlptracegrpc.WithDialOption(grpc.WithBlock()))
    log.Println("start to connect to server")
    traceExp, err := otlptrace.New(ctx, traceClient)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在链路追踪后端显示的服务名称。
            semconv.ServiceNameKey.String(common.ServerServiceName),
        ),
    )
    handleErr(err, "failed to create resource")
    bsp := sdktrace.NewBatchSpanProcessor(traceExp)
    tracerProvider := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithResource(res),
        sdktrace.WithSpanProcessor(bsp),
    )
    // 设置全局propagator为tracecontext（默认不设置）。
    otel.SetTextMapPropagator(propagation.TraceContext{})
    otel.SetTracerProvider(tracerProvider)
    return func() {
        cxt, cancel := context.WithTimeout(ctx, time.Second)
        defer cancel()
        if err := traceExp.Shutdown(cxt); err != nil {
            otel.Handle(err)
        }
    }
}
```

3. 添加埋点。

```
shutdown := initProvider()
    defer shutdown()
    //meter := global.Meter("demo-server-meter")
    serverAttribute := attribute.String("server-attribute", "foo")
    fmt.Println("start to gen chars for trace data")
    initTraceDemoData()
    fmt.Println("gen trace data done")
    tracer := otel.Tracer(common.TraceInstrumentationName)
    // 在OpenTelemetry中创建一个handler。
    handler := http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
        //  模拟延迟
        var sleep int64
        switch modulus := time.Now().Unix() % 5; modulus {
        case 0:
            sleep = rng.Int63n(2000)
        case 1:
            sleep = rng.Int63n(15)
        case 2:
            sleep = rng.Int63n(917)
        case 3:
            sleep = rng.Int63n(87)
        case 4:
            sleep = rng.Int63n(1173)
        }
        ctx := req.Context()
        span := trace.SpanFromContext(ctx)
        span.SetAttributes(serverAttribute)
        actionChild(tracer, ctx, sleep)
        w.Write([]byte("Hello World"))
    })
    wrappedHandler := otelhttp.NewHandler(handler, "/hello")
    http.Handle("/hello", wrappedHandler)
    http.ListenAndServe(":7080", nil)
```

4. 启动应用程序。

```
go run main.go
```

在链路追踪Tracing Analysis控制台的**应用列表**页面选择新创建的应用，查看链路数据。

使用HTTP协议上报

1. 添加OpenTelemetry Go依赖。

```
go get go.opentelemetry.io/otel
go get go.opentelemetry.io/otel/trace
go get go.opentelemetry.io/otel/sdk
go get go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracehttp # otlp http
```

2. 初始化OpenTelemetry Go SDK，其中Endpoint和URLPath需替换为前提条件中获取的接入点信息。

   您可以选择通过OpenTelemetry SDK直接上报或通过开源OpenTelemetry Collector转发。

   - 如果选择直接上报，请将Endpoint和URLPath替换为前提条件获取的接入点信息。
   - 如果选择使用OpenTelemetry Collector转发，请将Endpoint替换为您本地部署的服务地址，并删除Header信息。

```
func initProvider() func() {
    ctx := context.Background()
    traceClientHttp := otlptracehttp.NewClient(
        otlptracehttp.WithEndpoint("127.0.XX.XX:8080"),      //Endpoint需替换为前提条件中获取的接入点信息。
        otlptracehttp.WithURLPath("/adapt_xxxxx/api/otlp/traces"),   //URLPath需替换为前提条件中获取的接入点信息。
        otlptracehttp.WithInsecure())
    otlptracehttp.WithCompression(1)
    traceExp, err := otlptrace.New(ctx, traceClientHttp)
    handleErr(err, "Failed to create the collector trace exporter")
    res, err := resource.New(ctx,
        resource.WithFromEnv(),
        resource.WithProcess(),
        resource.WithTelemetrySDK(),
        resource.WithHost(),
        resource.WithAttributes(
            // 在链路追踪后端显示的服务名称。
            semconv.ServiceNameKey.String(common.ClientServiceName),
        ),
    )
    handleErr(err, "failed to create resource")
    bsp := sdktrace.NewBatchSpanProcessor(traceExp)
    tracerProvider := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithResource(res),
        sdktrace.WithSpanProcessor(bsp),
    )
    // 设置全局propagator为tracecontext（默认不设置）。
    otel.SetTextMapPropagator(propagation.TraceContext{})
    otel.SetTracerProvider(tracerProvider)
    log.Println("OTEL init success")
    return func() {
        cxt, cancel := context.WithTimeout(ctx, time.Second)
        defer cancel()
        if err := traceExp.Shutdown(cxt); err != nil {
            otel.Handle(err)
        }
    }
}
```

3. 添加埋点。

```
tracer := otel.Tracer(common.TraceInstrumentationName)
    method, _ := baggage.NewMember("method", "repl")
    client, _ := baggage.NewMember("client", "cli")
    bag, _ := baggage.New(method, client)
    defaultCtx := baggage.ContextWithBaggage(context.Background(), bag)
    for {
        ctx, span := tracer.Start(defaultCtx, "ExecuteRequest")
        makeRequest(ctx)
        span.End()
        time.Sleep(time.Duration(1) * time.Second)
    }
```

4. 启动应用程序。

```
go run main.go
```

在链路追踪Tracing Analysis控制台的**应用列表**页面选择新创建的应用，查看链路数据。

相关文档

- Open Telemetry官方文档

5.7.2. Kitex接入链路追踪

本文演示如何将Kitex应用接入ARMS链路追踪。

前提条件

已开通ARMS链路追踪。

背景信息

CloudWeGo-Kitex是字节跳动开源的Golang微服务RPC框架，具有高性能、强可扩展的主要特点。其默认支持Thrift、Kitex Protobuf和gRPC消息协议，且支持丰富服务治理能力及扩展能力。

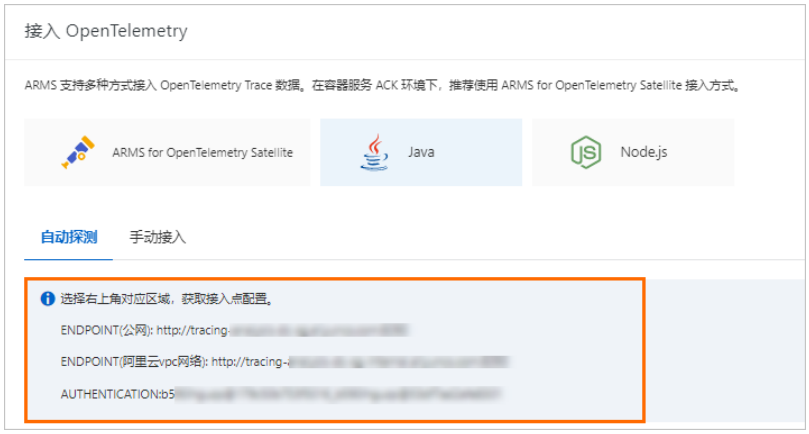
步骤一：接入Kitex

在您的Kitex应用中，添加如下代码。

```
import (
    "github.com/kitex-contrib/obs-opentelemetry/provider"
    "github.com/kitex-contrib/obs-opentelemetry/tracing"
    // ...
)
func main() {
    // 省略部分初始化代码
    // **接入OpenTelemetry，默认从环境变量配置**
    p := provider.NewOpenTelemetryProvider(
        provider.WithServiceName(constants.NoteServiceName),
        provider.WithEnableMetrics(false),
    )
    defer p.Shutdown(context.Background())
    Init()
    svr := note.NewServer(new(NoteServiceImpl),
        server.WithServerBasicInfo(&rpcinfo.EndpointBasicInfo{ServiceName: constants.NoteSe
rviceName}), // server name
        // ...
        // **注入tracing到server实例中**
        server.WithSuite(tracing.NewServerSuite()),
    )
    err = svr.Run()
    if err != nil {
        klog.Fatal(err)
    }
}
```

步骤二：部署应用

1. 如果您使用的是ACK服务，可以在容器服务管理控制台目标集群的**工作负载 > 无状态**页面创建Deployment。具体操作，请参见创建无状态工作负载Deployment。
2. 获取链路追踪接入点信息。
   i. 在ARMS控制台左侧导航栏单击**接入中心**。
   ii. 在**开源监控系统**区域单击**OpenTelemetry**。
   iii. 在**接入OpenTelemetry**面板选择对应的应用语言，然后获取接入点Endpoint和Authentication。



3. 配置如下环境变量，从而上报数据到ARMS链路追踪。

Key	说明	示例值
OTEL_EXPORTER_OTLP_TRACES_ENDPOINT	Trace上报的Endpoint。	http://tracing-analysis-dc-zb.aliyuncs.com:8090
OTEL_EXPORTER_OTLP_TRACES_HEADERS	OTEL上报会带上的Header，用于鉴权。	authentication=cx0xxxxx@285xxxxx_cx0xxxxx@53xxxxx

验证

启动应用并引入流量后，在ARMS控制台**应用监控 > 应用列表**页面选择新创建的应用，查看链路数据。

5.7.3. 通过Jaeger上报Go应用数据

在追踪应用的链路数据之前，您需要通过客户端将应用数据上报至链路追踪服务端。本文介绍如何通过Jaeger客户端上报Go应用数据，包括使用Jaeger SDK上报和使用Jaeger Agent上报两种方式，并在文末提供示例。

前提条件

获取接入点信息 ❯

背景信息

数据是如何上报的？ ❯

方式一：通过Jaeger SDK上报数据

此处以依赖管理工具Go Modules为例，您可以通过Jaeger SDK埋点直接将数据上报到链路追踪服务端。若使用其他依赖管理工具，请按实际情况操作。

1. 引入jaeger-client-go。

```
go get github.com/uber/jaeger-client-go
```

2. 创建Tracer对象。

> **说明** 请将 `<endpoint>` 替换成链路追踪控制台**接入点信息**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，请参见本文前提条件。

```
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.Header))
...
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()
```

更多详细使用方法，请参见Go Doc。

方式二：通过Jaeger Agent上报数据

1. 启动Jaeger Agent。具体操作，请参见安装Jaeger Agent。
2. 引入jaeger-client-go。

```
go get github.com/uber/jaeger-client-go
```

3. 创建Tracer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
}
```

4. 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.Header))
...
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()
```

更多详细使用方法，请参见Go Doc。

使用示例

通过Jaeger SDK直接上报

1. 获取接入点信息。具体操作方法，请参见本文前提条件。
2. 运行以下命令，下载示例文件。

```
wget https://arms-apm-cn-hangzhou.oss-cn-hangzhou.aliyuncs.com/demo/tracing-demo.zip &&
unzip tracing-demo.zip
```

3. 打开示例文件夹中的*examples/settings.go*文件，配置TracingAnalysisEndpoint，将图示①替换为步骤1中获取的接入点信息。

settings.go

```
package examples

// SDK上报需要：设置链路追踪的网关（不同region对应不同的值，从http://tracing.console.aliyun.com/ 的配置查看中获取）
const TracingAnalysisEndpoint = "http://tracing-analysis-dc-hz.aliyuncs.com/adapt_xxxxxx_xxxxx/api/traces"

// Agent上报：true （需要本地启动 jaeger-agent） SDK上报：false（默认值）
const AgentSwitch = false

// 链路追踪控制台对应的应用名
const (
    CliTracerServiceName = "cliDemo"

    GrpcServerName = "grpcServer"
    GrpcClientName = "grpcClient"

    HttpServerName = "httpServer"
    HttpClientName = "httpClient"
)
```

4. 使用 `go mod tidy` 命令整理依赖。
5. 使用 `go run tracingdemo` 命令运行示例。

通过Jaeger Agent上报

1. 获取接入点信息。具体操作方法，请参见本文前提条件。
2. 运行以下命令，下载示例文件。

```
wget https://arms-apm-cn-hangzhou.oss-cn-hangzhou.aliyuncs.com/demo/tracing-demo.zip &&
unzip tracing-demo.zip
```

3. 启动Jaeger Agent。具体操作，请参见安装Jaeger Agent。
4. 打开示例文件夹中的*examples/settings.go*文件，将 `AgentSwitch` 修改为 `true` 。

settings.go

```go
package examples

// SDK上报需要：设置链路追踪的网关（不同region对应不同的值，从http://tracing.console.aliyun.com/ 的配置查看中获取）
const TracingAnalysisEndpoint = "http://tracing-analysis-dc-hz.aliyuncs.com/adapt_xxxxxx_xxxxx/api/traces"

// Agent上报：true （需要本地启动 jaeger-agent） SDK上报：false（默认值）
const AgentSwitch = false

// 链路追踪控制台对应的应用名
const (
    CliTracerServiceName = "cliDemo"

    GrpcServerName = "grpcServer"
    GrpcClientName = "grpcClient"

    HttpServerName = "httpServer"
    HttpClientName = "httpClient"
)

```

5. 使用 `go mod tidy` 命令整理依赖。
6. 使用 `go run tracingdemo` 命令运行示例。

常见问题

Q：在运行过程中，为什么会出现以下报错？

```
2021/06/28 21:11:54 ERROR: error when flushing the buffer: error from collector: 403
2021/06/28 21:11:54 ERROR: error when flushing the buffer: error from collector: 403
```

A：出现上述报错，说明输入的接入点信息不正确。请更正并重试。

5.7.4. 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/adap
t_aokcdqnxyz@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。

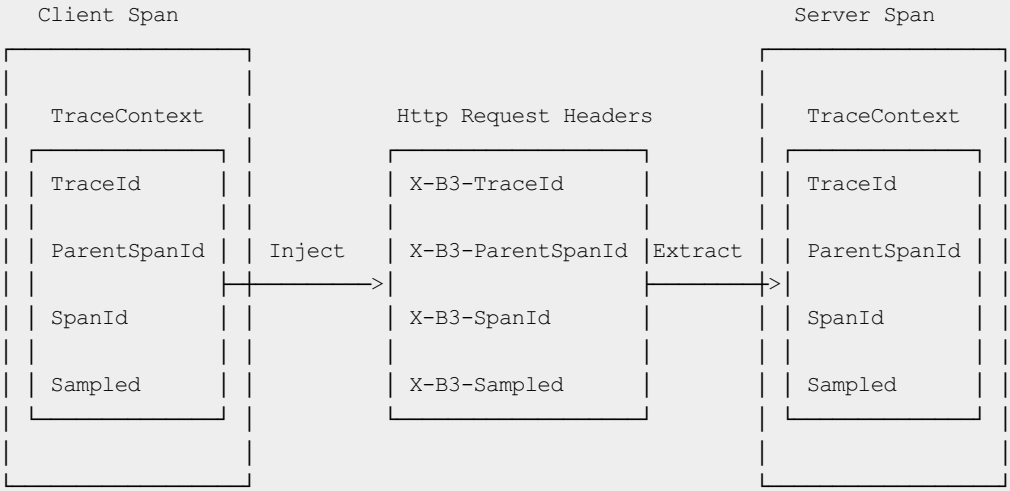
```
// tracer can now be used to create spans.
span := tracer.StartSpan("some_operation")
// ... do some work ...
span.Finish()
// Output:
```

> **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（endpoint URL）。

> **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。

相关文档

- Zipkin官网
- Zipkin-go源码

5.8. 开始监控Python应用

5.8.1. 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？ ❯

注意事项

- 针对Python语言，最新v1.25版本的Jaeger仅支持通过Jaeger Agent而不支持直接使用HTTP协议上报调用链路数据。更多信息，请参见Jaeger官方文档。
- 针对Python语言，最新v1.25版本的Jaeger仅支持通过UDP协议从Jaeger Client端上报至Jaeger Agent端。由于UDP协议并不保证通信的可靠性，因此为了保证调用链路数据的可靠性，一般情况下需要将Jaeger Client端和Jaeger Agent端运行在同一个主机内。

步骤一：搭建环境

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

Docker版本：20.10.7

Jaeger Agent版本：1.25

1. 在DockerHub中执行以下命令拉取v1.25版本的Jaeger Agent镜像。

```
docker pull jaegertracing/jaeger-agent:1.25
```

2. 执行以下命令运行v1.25版本的Jaeger Agent。

```
docker run -d --name jaeger-agent  -p 5775:5775/udp -p 6831:6831/udp -p 6832:6832/udp -p 5778:5778/tcp jaegertracing/jaeger-agent:1.25 --reporter.type=grpc --reporter.grpc.host-port=<endpoint>(填写对应的接入点信息) --agent.tags=<auth>(填写对应的认证信息)
```

> **说明** 请将 `<endpoint>` 和 `<auth>` 替换成控制台**集群配置**页面相应客户端地域的接入点信息。获取接入点信息的方法，请参见前提条件。

Python版本：3.8.5

Jaeger Client版本：4.6.0

在Python中安装以下Python包配置Jaeger Client环境。

```
certifi==2021.5.30
charset-normalizer==2.0.4
idna==3.2
jaeger-client==4.6.0
opentracing==2.4.0
requests==2.26.0
six==1.16.0
threadloop==1.0.2
thrift==0.13.0
tornado==6.1
urllib3==1.26.6
```

Docker和Jaeger Agent环境

Python和Jaeger Client环境

步骤二：创建Tracer对象

1. 创建包含如下内容的Python文件。

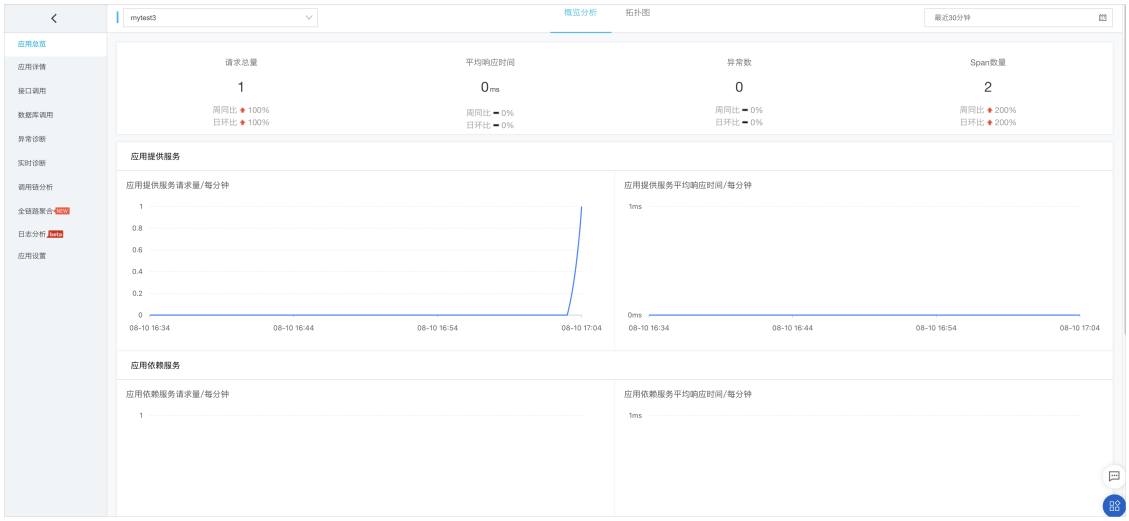
   通过以下代码创建Tracer对象，并通过Tracer对象创建Span来上报数据至链路追踪后台。

```
import logging
import time
from jaeger_client import Config
def construct_span(tracer):
    with tracer.start_span('AliyunTestSpan') as span:
        span.log_kv({'event': 'test message', 'life': 42})
        print("tracer.tages: ", tracer.tags)
        with tracer.start_span('AliyunTestChildSpan', child_of=span) as child_span:
            span.log_kv({'event': 'down below'})
        return span
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,
            },
            'local_agent': {
                # 注意这里是指定了JaegerAgent的host和port。
                # 根据官方建议为了保证数据可靠性，JaegerClient和JaegerAgent运行在同一台主机内
，因此reporting_host填写为127.0.0.1。
                'reporting_host': '127.0.0.1',
                'reporting_port': 6831,
            },
            'logging': True,
        },
        #这里填写应用名称
        service_name="mytest3",
        validate=True
    )
    # this call also sets opentracing.tracer
    tracer = config.initialize_tracer()
    span = construct_span(tracer)
    time.sleep(2)   # yield to IOLoop to flush the spans - https://github.com/jaegertra
cing/jaeger-client-python/issues/50
    tracer.close()  # flush any buffered spans
```

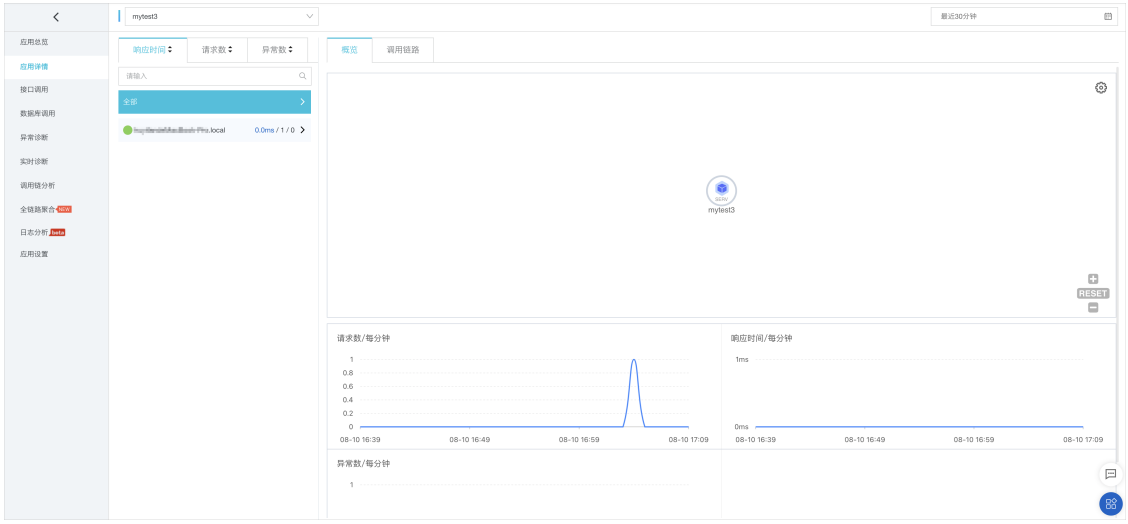
2. 执行新建的Python文件。

在链路追踪控制台查看数据

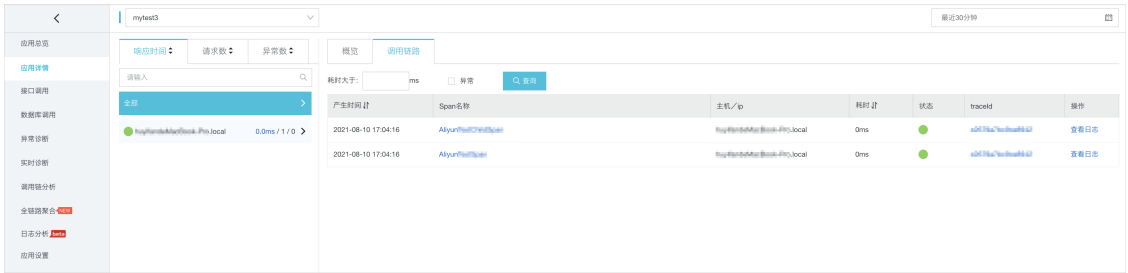
1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域。
3. 单击Python应用名称。
   在弹出的**应用总览**页面，您可以查看该应用的性能关键指标和拓扑图详情。



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



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



参考信息

此处提供了Jaeger常见的使用方法，更多信息，请参见Jaeger官方文档。

- 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={})
```

相关文档

- OpenTracing指南
- jaeger-client-python

5.9. 开始监控Node.js应用

5.9.1. 通过OpenTelemetry上报Node.js应用数据

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

前提条件

已在*package.json*中配置对OpenTelemetry的依赖。

```
"dependencies": {
   "@opentelemetry/api": "^1.0.4",
   "@opentelemetry/exporter-trace-otlp-grpc": "^0.27.0",
   "@opentelemetry/instrumentation": "^0.27.0",
   "@opentelemetry/instrumentation-express": "^0.27.0",
   "@opentelemetry/resources": "^1.0.1",
   "@opentelemetry/sdk-trace-base": "^1.0.1",
   "@opentelemetry/sdk-trace-node": "^1.0.1",
   "instrumentation-http": "^0.27.0",
 }
```

操作步骤

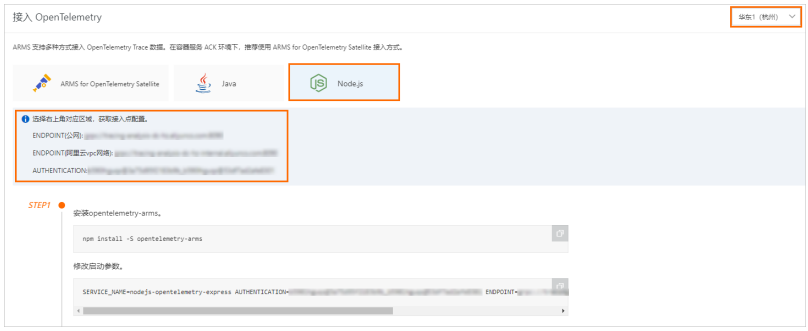
1. 获取接入点信息。
    i. 登录ARMS控制台。

ii. 在左侧导航栏单击**接入中心**，然后在**开源监控系统**区域单击**OpenTelemetry**。

iii. 在**接入OpenTelemetry**面板单击**Node.js**页签，然后在右上角选择应用需要接入的地域。

iv. 复制并保存Endpoint和Authentication信息。

> **说明** 如果需要将应用部署于阿里云生产环境，则接入点选阿里云VPC网络下的Endpoint，否则选择公网Endpoint。



2. 创建Provider。

```
const { Resource } = require("@opentelemetry/resources");
const { NodeTracerProvider } = require("@opentelemetry/sdk-trace-node");
const {
  SemanticResourceAttributes,
} = require("@opentelemetry/semantic-conventions");
const provider = new NodeTracerProvider({
  resource: new Resource({
    [SemanticResourceAttributes.HOST_NAME]: require("os").hostname(),
    [SemanticResourceAttributes.SERVICE_NAME]: "opentelemetry-express",    //opentelemetry-express可替换为任意名称。
  }),
});
```

3. 通过Provider注册HTTP和Express框架，自动监测并拦截HTTP和Express。

> **说明** 如需监测其他框架下的Node.js应用，请参见OpenTelemetry官方文档。

```
const { registerInstrumentations } = require("@opentelemetry/instrumentation");
const { HttpInstrumentation } = require("instrumentation-http");
const {
  ExpressInstrumentation,
} = require("@opentelemetry/instrumentation-express");
registerInstrumentations({
  tracerProvider: provider,
  instrumentations: [HttpInstrumentation, ExpressInstrumentation],
});
```

4. 配置Exporter，导出数据到ARMS链路追踪。

请将下面代码中的 `<ENDPOINT>` 和 `<AUTHENTICATION>` 替换成步骤1中获取的Endpoint和Authentication。

```
const metadata = new grpc.Metadata();
metadata.set("Authentication", "<AUTHENTICATION>");
const exporter = new OTLPTraceExporter({ url: "<ENDPOINT>", metadata });
provider.addSpanProcessor(new SimpleSpanProcessor(exporter));
provider.register();
```

5. （可选）添加自定义事件和属性。

> **说明** OpenTelemetry API的使用方法，请参见OpenTelemetry官方文档。

```
const api = require("@opentelemetry/api");
const currentSpan = api.trace.getSpan(api.context.active());
currentSpan.addEvent("timestamp", { value: Date.now() });
currentSpan.setAttribute("tagKey-01", "tagValue-01");
```

在ARMS控制台查看Trace

在ARMS控制台的**应用监控 > Trace Explorer**页面可以查看Node.js应用上报的监控数据。

基于Express框架的Node.js应用完整示例

```
"use strict";
const { Resource } = require("@opentelemetry/resources");
const {
  OTLPTraceExporter,
} = require("@opentelemetry/exporter-trace-otlp-grpc");
const { NodeTracerProvider } = require("@opentelemetry/sdk-trace-node");
const { HttpInstrumentation } = require("instrumentation-http");
const { SimpleSpanProcessor } = require("@opentelemetry/sdk-trace-base");
const {
  ExpressInstrumentation,
} = require("@opentelemetry/instrumentation-express");
const { registerInstrumentations } = require("@opentelemetry/instrumentation");
const {
  SemanticResourceAttributes,
} = require("@opentelemetry/semantic-conventions");
const grpc = require("@grpc/grpc-js");
const provider = new NodeTracerProvider({
  resource: new Resource({
    [SemanticResourceAttributes.HOST_NAME]: require("os").hostname(),
    [SemanticResourceAttributes.SERVICE_NAME]: "opentelemetry-express",
  }),
});
registerInstrumentations({
  tracerProvider: provider,
  instrumentations: [HttpInstrumentation, ExpressInstrumentation],
});
const metadata = new grpc.Metadata();
metadata.set("Authentication", "<AUTHENTICATION>");
const exporter = new OTLPTraceExporter({ url: "<ENDPOINT>", metadata });
provider.addSpanProcessor(new SimpleSpanProcessor(exporter));
provider.register();
// 应用代码
const api = require("@opentelemetry/api");
const axios = require("axios").default;
const express = require("express");
const app = express();
app.get("/", async (req, res) => {
  const result = await axios.get("http://localhost:7001/api");
  return res.status(201).send(result.data);
});
app.get("/api", async (req, res) => {
  const currentSpan = api.trace.getSpan(api.context.active());
  currentSpan.addEvent("timestamp", { value: Date.now() });
  currentSpan.setAttribute("tagKey-01", "tagValue-01");
  res.json({ code: 200, msg: "success" });
});
app.use(express.json());
app.listen(7001, () => {
  console.log("Listening on http://localhost:7001");
});
```

5.9.2. 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');
});
```

相关文档

- jaeger-client-node
- opentracing-javascript

5.10. 开始监控.NET应用

5.10.1. 通过OpenTelemetry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OpenTelemetry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 ❯

示例Demo

示例代码仓库地址：dotnet-demo

使用OpenTelemetry .NET SDK手动埋点

.NET 6.0埋点

1. 进入示例代码仓库的路径*dotnet-demo/opentelemetry-demo*，然后安装手动埋点所需的OpenTelemetry相关依赖。

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

2. 在*OpentelemetryExporterDemo.cs*文件中创建OpenTelemetry TracerProvider，并添加基于HTTP协议的OtlpExporter，修改上报应用名和上报数据的Endpoint。

```
using System.Diagnostics;
using OpenTelemetry;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;
using OpenTelemetry.Exporter;
namespace Demo
{
    internal static class OpentelemetryExporterDemo
    {
        internal static void Run()
        {
            Console.WriteLine("otlp running");
            // OpenTelemetry上报应用名
            var serviceName = "otlp-test";
            using var tracerProvider = Sdk.CreateTracerProviderBuilder()
                .AddSource(serviceName)
                .SetResourceBuilder(
                ResourceBuilder.CreateDefault().AddService(serviceName))
                .AddOtlpExporter(opt =>
                                {
                                    // 根据前提条件中获取的接入点信息进行修改
                                    opt.Endpoint = new Uri("http://localhost/api/otlp/
traces");
                                    // 使用HTTP协议上报数据
                                    opt.Protocol = OtlpExportProtocol.HttpProtobuf;
                                })
                .Build();
            for(int i = 0; i<10; i++)
            {
                var MyActivitySource = new ActivitySource(serviceName);
                using var activity = MyActivitySource.StartActivity("SayHello");
                activity?.SetTag("bar", "Hello World");
            }
        }
    }
}
```

3. 在当前路径下运行以下命令。

```
dotnet run --framework:net6.0
```

4. 启动本地示例程序，在链路追踪Tracing Analysis控制台按照自定义的应用名称搜索应用，查看上报的数据。

5.10.2. 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？ ❯

示例Demo

示例代码仓库：dotnet-demo

.NET 6.0埋点

通过OpenTracing组件自动埋点

Demo源码的运行版本要求：

- Jaeger：1.0.2版本
- .NET：6.0版本

1. 在示例项目的*dotnet-demo/net6.0/Shared/JaegerServiceCollectionExtensions.cs*文件中填写上报数据端口并修改上报服务名，用于实现ITracer对象的初始化和注册逻辑。

```
public static class JaegerServiceCollectionExtensions
{
    // 参考前提条件获取Jaeger Endpoint并填入参数
    private static readonly Uri _jaegerUri = new Uri("http://tracing-analysis-dc-sz.ali
yuncs.com/adapt_your_token/api/traces");
    public static IServiceCollection AddJaeger(this IServiceCollection services)
    {
        if (services == null)
            throw new ArgumentNullException(nameof(services));
        services.AddSingleton<ITracer>(serviceProvider =>
        {
            // 可以将其修改为自定义的服务名
            string serviceName = Assembly.GetEntryAssembly().GetName().Name;
            ILoggerFactory loggerFactory = serviceProvider.GetRequiredService<ILoggerFa
ctory>();
            ISampler sampler = new ConstSampler(sample: true);
            IReporter reporter = new RemoteReporter.Builder()
                .WithSender(new HttpSender.Builder(_jaegerUri.ToString()).Build())
                .Build();
            ITracer tracer = new Tracer.Builder(serviceName)
                .WithLoggerFactory(loggerFactory)
                .WithSampler(sampler)
                .WithReporter(reporter)
                .Build();
            GlobalTracer.Register(tracer);
            return tracer;
        });
        // Prevent endless loops when OpenTracing is tracking HTTP requests to Jaeger.
        services.Configure<HttpHandlerDiagnosticOptions>(options =>
        {
            options.IgnorePatterns.Add(request => _jaegerUri.IsBaseOf(request.RequestUr
i));
        });
        return services;
    }
}
```

2. 进入项目目录*dotnet-demo/net6.0/CustomersApi*，然后运行以下命令。

```
dotnet run --framework:net6.0
```

3. 启动本地服务，并访问以下地址。

```
http://localhost:5001/health
```

4. 在链路追踪Tracing Analysis控制台通过自定义的serviceName搜索应用，查看上报的数据。

NET Core 3.1埋点

Demo源码的运行版本要求：

- Jaeger：1.0.2版本
- .NET：3.1版本

通过NetCore组件自动埋点

1. 在项目的*dotnet-demo/netcoreapp3.1/Shared/JaegerServiceCollectionExtensions.cs*中填写上报数据端口并修改上报服务名，用于实现ITracer对象的初始化和注册逻辑。

```
public static class JaegerServiceCollectionExtensions
{
    // 参考前提条件获取Jaeger Endpoint并填入参数
    private static readonly Uri _jaegerUri = new Uri("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/api/traces");
    public static IServiceCollection AddJaeger(this IServiceCollection services)
    {
        if (services == null)
            throw new ArgumentNullException(nameof(services));
        services.AddSingleton<ITracer>(serviceProvider =>
        {
            // 可以将其修改为自定义的服务名
            string serviceName = Assembly.GetEntryAssembly().GetName().Name;
            ILoggerFactory loggerFactory = serviceProvider.GetRequiredService<ILoggerFactory>();
            ISampler sampler = new ConstSampler(sample: true);
            IReporter reporter = new RemoteReporter.Builder()
                .WithSender(new HttpSender.Builder(_jaegerUri.ToString()).Build())
                .Build();
            ITracer tracer = new Tracer.Builder(serviceName)
                .WithLoggerFactory(loggerFactory)
                .WithSampler(sampler)
                .WithReporter(reporter)
                .Build();
            GlobalTracer.Register(tracer);
            return tracer;
        });
        // Prevent endless loops when OpenTracing is tracking HTTP requests to Jaeger.
        services.Configure<HttpHandlerDiagnosticOptions>(options =>
        {
            options.IgnorePatterns.Add(request => _jaegerUri.IsBaseOf(request.RequestUri));
        });
        return services;
    }
}
```

2. 进入项目目录*dotnet-demo/netcoreapp3.1/Shared*，然后运行以下命令。

```
// 添加aspnetcore中间件
dotnet add  package OpenTracing.Contrib.NetCore
```

3. 进入项目目录*dotnet-demo/netcoreapp3.1/CustomersApi*，然后运行以下命令。

```
// 添加aspnetcore中间件
dotnet add  package OpenTracing.Contrib.NetCore
// 运行示例程序
dotnet run --framework:netcoreapp3.1
```

4. 启动本地服务，并访问以下地址。

```
http://localhost:5001/health
```

5. 在链路追踪Tracing Analysis控制台通过自定义的serviceName搜索应用，查看上报的数据。

通过gRPC组件自动埋点

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
.SamplerConfiguration(loggerFactory)
                .WithType(ConstSampler.Type)
                .WithParam(1);
            Configuration.SenderConfiguration senderConfiguration = new Configuration.S
enderConfiguration(loggerFactory)
                    // 参考前提条件获取Jaeger Endpoint并填入参数
                   .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your
_token/api/traces");
            Configuration.ReporterConfiguration reporterConfiguration = new Configurati
on.ReporterConfiguration(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));
```

5. 进入项目目录*dotnet-demo/netcoreapp3.1/GreeterServer*，然后在终端运行以下命令开启gRPC的服务端。

```
dotnet run --framework:netcoreapp3.1
```

6. 进入项目目录*dotnet-demo/netcoreapp3.1/GreeterClient*，然后在另一个终端运行以下命令开启gRPC的客户端。

```
dotnet run --framework:netcoreapp3.1
```

如果终端输出 `Greeting: Hello you` ，说明服务端和客户端之间通信成功，在链路追踪Tracing Analysis控制台可以看到示例应用dotnetGrpcServer和dotnetGrpcClient上报的数据。

手动埋点

除了利用各种现有插件实现埋点外，还可以使用手动埋点的方法通过Jaeger将.NET应用数据上报至链路追踪控制台。

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.Sampler
Configuration(loggerFactory)
          .WithType(ConstSampler.Type)
          .WithParam(1);
    Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderCon
figuration(loggerFactory)
           // 在链路追踪控制台获取Jaeger Endpoint。
          .WithEndpoint("http://tracing-analysis-dc-sz.aliyuncs.com/adapt_your_token/ap
i/traces");
   Configuration.ReporterConfiguration reporterConfiguration = new Configuration.Report
erConfiguration(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。

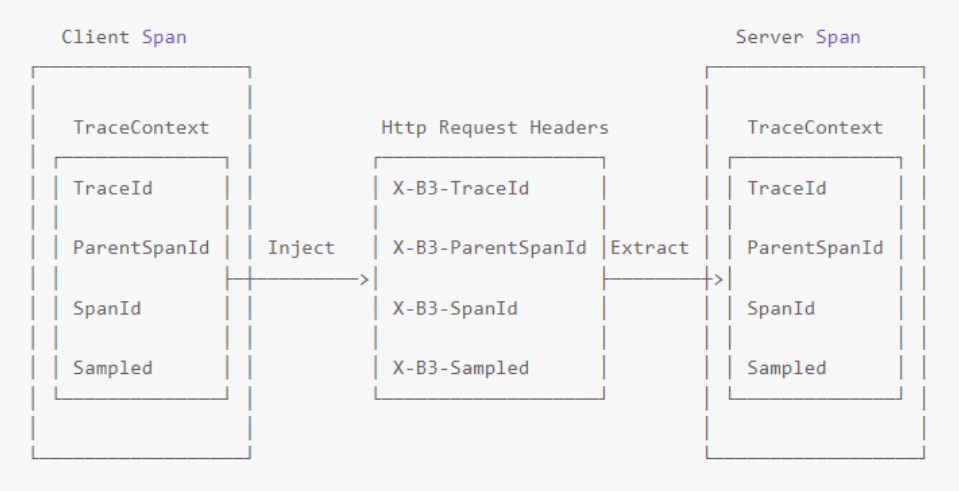
示例：

```
ITracer tracer = GlobalTracer.Instance;
ISpan parentSpan = tracer.ActiveSpan;
ISpan childSpan =tracer.BuildSpan("childSpan").AsChildOf(parentSpan).WithTag("mytag", "
spanSecond").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 HttpHeadersInjectAdapte
r(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);
```

7. 进入项目目录*dotnet-demo/netcoreapp3.1/ManualDemo*，然后运行以下命令，示例程序将会上报数据。

```
dotnet run --framework:netcoreapp3.1
```

在链路追踪Tracing Analysis控制台可以查看手动埋点的示例应用dotnetManualDemo上报的数据。

常见问题

Q1：Demo程序执行成功，为什么控制台上没有上报数据？

A1：请检查senderConfiguration配置中的Endpoint是否填写正确。

```
Configuration.SenderConfiguration senderConfiguration = new Configuration.SenderConfigurati
on(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/tr
aces");
```

Q2：如何设置采样率？

A2：具体详情，请参见Jaeger采样率文档。

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

5.10.3. 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 zipkin4net
```

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>()["applicationName"]));
}
```

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 zipkin4net
```

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
= 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);
//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["appli
cationName"]);
//
```

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.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，并按照Readme的说明运行程序。

1. 安装NuGet包。

```
// 添加zipkin4net（追踪器）。
dotnet add  package zipkin4net
```

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。

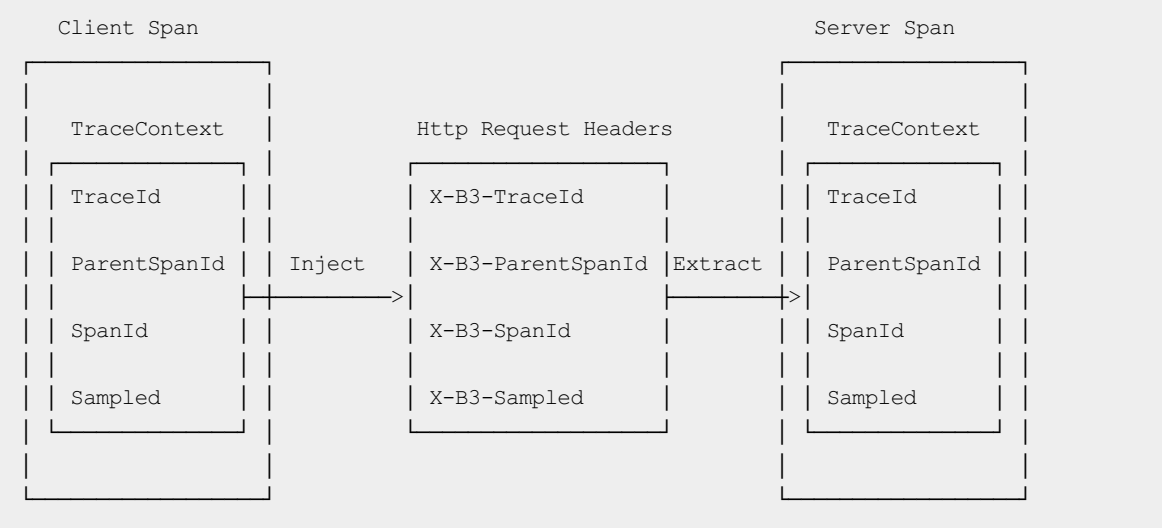
```
var trace = Trace.Create();
Trace.Current = trace;
trace.Record(Annotations.ClientSend());
trace.Record(Annotations.Rpc("client"));
trace.Record(Annotations.Tag("mytag", "spanFrist"));
trace.Record(Annotations.ServiceName("dotnetManual"));
// ...do something
testCall();
trace.Record(Annotations.ClientRecv());
```

> **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_you
r_token", "application/json");
```

相关文档

- Zipkin官网
- Zipkin的dotnet客户端

5.11. 开始监控C++应用

5.11.1. 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？ ❯

快速开始

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 mas
ter.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，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:18
83 --jaeger.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，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:18
83 --jaeger.tags=<endpoint>
```

5.12. 开始监控Nginx

5.12.1. 使用Jaeger对Nginx进行链路追踪

Nginx是一款自由的、开源的、高性能的HTTP服务器和反向代理服务器，对其进行跟踪可以帮助我们更好的了解应用服务的运行状况。本文将详细介绍Nginx的链路跟踪的安装过程。

前提条件

获取接入点信息 ❯

教程概述

当Nginx代理的微服务出现假死现象时，因为采集不到任何数据，所以无法评估造成的影响。借助链路追踪，我们追踪微服务的上游Nginx，并快速统计出假死现象影响的访问量。

操作步骤

1. 从Registry中拉取镜像。

```
docker pull registry.cn-hangzhou.aliyuncs.com/public-community/jaeger-nginx:0.1
```

2. 运行Nignx Docker。

```
docker run --rm  -p 80:80 -e "GRPC_HOST=${GRPC_HOST}" -e "GRPC_AUTH=${GRPC_AUTH}" -d re
gistry.cn-hangzhou.aliyuncs.com/public-community/jaeger-nginx:0.1
```

`${GRPC_HOST}` 和 `${GRPC_AUTH}` 是前提条件中保存的Agent接入点信息。

例如：

```
docker run --rm  -p 80:80 -e "GRPC_HOST=tracing-analysis-dc-hz.aliyuncs.com:1883" -e "G
RPC_AUTH=123abc@123abc_789abc@456abc}" -d jaeger-nginx:0.1
```

3. 访问Nginx页面。

   在浏览器访问localhost/nginx.conf或者curl "localhost/nginx.conf"。

4. 查看Nginx链路数据。

   登录链路追踪Tracing Analysis控制台可以查看应用nginx-jaeger的链路数据。

在Docker上部署和跟踪Nginx

1. 下载Dockerfile并编译部署。

```
wget https://arms-apm.oss-cn-hangzhou.aliyuncs.com/demo/nginx-jaeger-docker.tgz
tar -xzvf nginx-jaeger-docker.tgz
cd nginx-jaeger
// 编译docker
docker build --rm --tag nginx-jaeger:0.1 .
```

2. 运行Docker。

```
docker run --rm  -p 80:80 -e "GRPC_HOST=${GRPC_HOST}" -e "GRPC_AUTH=${DRPC_AUTH}" -d ja
eger-nginx:0.1
```

`${GRPC_HOST}` 和 `${GRPC_AUTH}` 是前提条件中保存的Agent接入点信息。

例如：

```
docker run --rm  -p 80:80 -e "GRPC_HOST=tracing-analysis-dc-hz.aliyuncs.com:1883" -e "G
RPC_AUTH=123abc@123abc_789abc@456abc}" -d jaeger-nginx:0.1
```

在ECS上部署和跟踪Nginx

1. 安装Nginx。

   i. 下载并解压Nginx源码。

```
wget http://nginx.org/download/nginx-1.14.2.tar.gz
tar -xzvf nginx-1.14.2.tar.gz
```

   ii. 编译Nginx源码。

```
cd nginx-1.14.2
./configure --with-compat
make
sudo make install
```

2. 安装OpenTracing插件。

i. 下载OpenTracing插件并解压。

```
wget https://github.com/opentracing-contrib/nginx-opentracing/releases/download/v0.7.0/linux-amd64-nginx-1.14.0-ngx_http_module.so.tgz
tar -xzvf linux-amd64-nginx-1.14.0-ngx_http_module.so.tgz
```

ii. 拷贝.so文件至Nginx的*modules*目录。如果不存在该目录则需要先创建。

```
sudo mkdir /usr/local/nginx/modules
sudo cp ngx_http_opentracing_module.so /usr/local/nginx/modules/ngx_http_opentracing_module.so
```

3. 使用Jaeger进行链路追踪。

i. 下载Jaeger插件并将其拷贝至任意工作目录。

```
wget https://github.com/jaegertracing/jaeger-client-cpp/releases/download/v0.4.2/libjaegertracing_plugin.linux_amd64.so
sudo cp /usr/local/lib/libjaegertracing_plugin.so /usr/local/lib/libjaegertracing.so
```

ii. 配置*/usr/local/nginx/conf/nginx.conf*文件。

```
load_module modules/ngx_http_opentracing_module.so;
events {}
http {
  opentracing on;
  opentracing_load_tracer /usr/local/lib/libjaegertracing_plugin.so /etc/jaeger-config.json;
  server {
    error_log /var/log/nginx/debug.log debug;
    listen 80;
    location  ~ {
      opentracing_operation_name $uri;
      opentracing_trace_locations off;
      # 跳转到代理的服务，用户根据需要替换。
      proxy_pass http://127.0.0.1:8081;
      opentracing_propagate_context;
    }
  }
}
```

> **说明** 详细配置说明，请参见opentracing-contrib配置。

iii. 在*/etc/jaeger-config.json*文件中配置Jaeger参数。

```
{
  "service_name": "nginx",
  // 配置采样
  "sampler": {
    "type": "const",
    "param": 1
  },
  "reporter": {
    "localAgentHostPort": "localhost:6831"
  }
}
```

iv. 采用以下方法之一配置Jaeger Agent。

- 若使用自建的Jaeger服务，则下载原生Jaeger Agent，并配置collector.host-port。

```
nohup ./jaeger-agent  --collector.host-port=10.100.**.**:142**   1>1.log 2>2.log
&
```

- 若使用阿里云的Jaeger托管服务，则下载tracing-analysis-agent，并用以下参数启动Agent，以将数据上报至链路追踪Tracing Analysis。
  请将*<endpoint>*替换成前提条件中保存的Agent接入点信息。请删除页面上显示的接入点信息末尾的“/api/traces”，例如 `http://tracing-analysis-dc-sh.aliyuncs.com/adapt_abc123@abc123_efg123@efg123)` 。

```
// collector.host-port 用于设置网关，网关因地域而异。例如：
nohup ./tracing-analysis-agent-linux-amd64 --collector.host-port=<endpoint>
```

v. 运行Nginx并访问Nginx服务。

```
sudo /usr/local/nginx/sbin/nginx
curl "http://localhost"
```

查看结果

稍等片刻后到链路追踪控制台查看，如果有监控数据则表示追踪成功。如有关于Nginx监控的疑问，欢迎通过钉钉账号osfriend交流。

相关文档

- nginx-opentracing项目
- jaeger-client-cpp项目
- zipkin-cpp-opentracing
- Jaeger的Nginx Example
- Zipkin的Nginx Example

5.12.2. 使用Zipkin对Nginx进行链路追踪

Nginx是一款自由的、开源的、高性能的HTTP服务器和反向代理服务器，对其进行跟踪可以帮助我们更好的了解应用服务的运行状况。本文将详细介绍Nginx的链路跟踪的安装过程。

前提条件

获取接入点信息 ❯

教程概述

当Nginx代理的微服务出现假死现象时，因为采集不到任何数据，所以无法评估造成的影响。借助链路追踪，我们追踪微服务的上游Nginx，并快速统计出假死现象影响的访问量。

操作步骤

1. 从Registry中拉取镜像。

```
sudo docker pull registry.cn-hangzhou.aliyuncs.com/public-community/nginx-zipkin:0.1
```

2. 运行Nignx Docker。

```
docker run --rm  -p 80:80 -e "COLLECTOR_HOST=${ZIPKIN_ENDPOINT}?" -d registry.cn-hangzh
ou.aliyuncs.com/public-community/nginx-zipkin:0.1
```

`${ZIPKIN_ENDPOINT}` 是前提条件中保存的v1版本的Agent接入点信息。不包括“http://”部分，且以英文问号（?）结尾。
例如：

```
docker run --rm  -p 80:80 -e "COLLECTOR_HOST=tracing-analysis-dc-hz.aliyuncs.com/adapt_
******_******/api/v1/spans?" -d registry.cn-hangzhou.aliyuncs.com/public-community/ngin
x-zipkin:0.1
```

3. 访问Nginx页面。

   在浏览器访问localhost/nginx.conf或者curl "localhost/nginx.conf"。

4. 查看Nginx链路数据。

   登录链路追踪Tracing Analysis控制台可以查看应用nginx-zipkin的链路数据。

在Docker上部署和跟踪Nginx

1. 下载Dockerfile并编译部署。

```
wget https://arms-apm.oss-cn-hangzhou.aliyuncs.com/demo/nginx-zipkin-docker.tgz
tar -xzvf nginx-zipkin-docker.tgz
cd nginx-zipkin
// 编译docker
docker build --rm --tag nginx-zipkin:0.1
```

2. 运行Docker。

```
docker run --rm  -p 80:80 -e "COLLECTOR_HOST=${ZIPKIN_ENDPOINT}?" -d nginx-zipkin:0.1
```

`${ZIPKIN_ENDPOINT}` 是前提条件中保存的v1版本的Agent接入点信息。不包括“http://”部分，且以英文问号（?）结尾。
例如：

```
docker run --rm  -p 80:80 -e "COLLECTOR_HOST=tracing-analysis-dc-hz.aliyuncs.com/adapt_
123@abc_456@efg/api/v1/spans?" -d nginx-zipkin:0.1
```

在ECS上部署和跟踪Nginx

1. 安装Nginx。

i. 下载并解压Nginx源码。

```
wget http://nginx.org/download/nginx-1.14.2.tar.gz
tar -xzvf nginx-1.14.2.tar.gz
```

ii. 编译Nginx源码。

```
cd nginx-1.14.2
./configure --with-compat
make
sudo make install
```

2. 安装OpenTracing插件。

i. 下载OpenTracing插件并解压。

```
wget https://github.com/opentracing-contrib/nginx-opentracing/releases/download/v0.
7.0/linux-amd64-nginx-1.14.0-ngx_http_module.so.tgz
tar -xzvf linux-amd64-nginx-1.14.0-ngx_http_module.so.tgz
```

ii. 拷贝.so文件至Nginx的*modules*目录。如果不存在该目录则需要先创建。

```
sudo mkdir /usr/local/nginx/modules
sudo cp ngx_http_opentracing_module.so /usr/local/nginx/modules/ngx_http_opentracin
g_module.so
```

3. 使用Zipkin进行链路追踪。

i. 下载Zipkin插件并将其拷贝至任意工作目录。

```
wget  https://github.com/rnburn/zipkin-cpp-opentracing/releases/download/v0.5.2/lin
ux-amd64-libzipkin_opentracing_plugin.so.gz
gunzip linux-amd64-libzipkin_opentracing_plugin.so.gz
sudo cp linux-amd64-libzipkin_opentracing_plugin.so /usr/local/lib/libzipkin_opentr
acing_plugin.so
```

ii. 配置*/usr/local/nginx/conf/nginx.conf*文件。

```
load_module modules/ngx_http_opentracing_module.so;
events {}
http {
  opentracing on;
  opentracing_load_tracer /usr/local/lib/libzipkin_opentracing.so /etc/zipkin-confi
g.json;
  server {
    error_log /var/log/nginx/debug.log debug;
    listen 80;
    location  ~ {
      opentracing_operation_name $uri;
      opentracing_trace_locations off;
      # 跳转到代理的服务，用户根据需要替换。
      proxy_pass http://127.0.0.1:8081;
      opentracing_propagate_context;
    }
  }
}
```

> **说明** 详细配置说明，请参见opentracing-contrib配置。

iii. 在*/etc/zipkin-config.json*文件中配置Zipkin参数。

```
{
  "service_name": "nginx",
  "collector_host": "zipkin"
}
```

若使用阿里云的Zipkin托管服务，则将collector_host配置为Zipkin接口。

> **说明** Zipkin接口的值为前提条件中保存的v1版本的Agent接入点信息，不包括“http://”部分，且以英文问号（?）结尾。

```
"collector_host": "tracing-analysis-dc-hz.aliyuncs.com/adapt_abc123@abc456_abc123@a
bc356/api/v1/spans?"
```

iv. 通过sample_rate配置采样比例。

```
// 10%的采样比例
"sample_rate":0.1
```

v. 运行Nginx并访问Nginx服务。

```
sudo /usr/local/nginx/sbin/nginx
curl "http://localhost"
```

查看结果

稍等片刻后到链路追踪控制台查看，如果有监控数据则表示追踪成功。如有关于Nginx监控的疑问，欢迎通过钉钉账号osfriend交流。

相关文档

- nginx-opentracing项目
- jaeger-client-cpp项目
- zipkin-cpp-opentracing
- Jaeger的Nginx Example
- Zipkin的Nginx Example

5.12.3. 使用Skywalking对Nginx进行链路追踪

在使用链路追踪控制台追踪Nginx的链路数据之前，需要将链路数据上报至链路追踪服务台。本文介绍如何通过SkyWalking Nginx LUA module上报Nginx的链路追踪数据。

前提条件

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应用性能监控APM（Application Performance Monitoring）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。

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既支持自动埋点（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。

通过Docker镜像快速配置skywalking-nginx-lua

使用打包好的Docker配置skywalking-nginx-lua。

1. 从Registry中拉取镜像。

   ```
   docker pull registry.cn-hangzhou.aliyuncs.com/public-community/skywalking-nginx-lua:0.2
   ```

2. 运行Nignx Docker。

   ```
   docker run --rm  -p 80:80 -e "BACKEND_URL=$skywalking-nginx-lua"  -d registry.cn-hangzhou.aliyuncs.com/public-community/skywalking-nginx-lua:0.2
   ```

   `$skywalking-nginx-lua` 是前提条件中保存的nginx-lua接入点信息。

   例如：

   ```
   docker run --rm  -p 80:80 -e "BACKEND_URL=http://tracing-analysis-dc-hz.aliyuncs.com/adapt_123@abc_456@efg"  -d registry.cn-hangzhou.aliyuncs.com/public-community/skywalking-nginx-lua:0.2
   ```

3. 访问Nginx页面。
   - 在浏览器上访问 `localhost/nginx.conf` 。
   - 执行命令 `curl "localhost/nginx.conf"` 。

通过Dockerfile配置skywalking-nginx-lua

1. 下载Dockerfile。

```
wget https://arms-apm.oss-cn-hangzhou.aliyuncs.com/demo/nginx-skywalking-docker.tgz
tar -xzvf nginx-skywalking-docker.tgz
cd nginx-lua
```

2. 编译Docker。

```
docker build --rm --tag skywalking-nginx-lua:0.2 .
```

3. 运行Docker。

```
docker run --rm  -p 80:80 -e "BACKEND_URL=$skywalking-nginx-lua"  -d skywalking-nginx-l
ua:0.2
```

`$skywalking-nginx-lua` 是前提条件中保存的nginx-lua接入点信息。
例如：

```
docker run --rm  -p 80:80 -e "BACKEND_URL=http://tracing-analysis-dc-hz.aliyuncs.com/ad
apt_123@abc_456@efg"  -d skywalking-nginx-lua:0.2
```

4. 访问Nginx页面。
   - 在浏览器上访问 `localhost/nginx.conf` 。
   - 执行命令 `curl "localhost/nginx.conf"` 。

在ECS上配置skywalking-nginx-lua

此处以在CentOS 7.0上的操作为例。

1. 配置Lua运行环境。
   i. 安装工具库。

```
yum install gcc gcc-c++ kernel-devel -y
yum install readline-devel -y
yum install ncurses-devel -y
```

   ii. 下载并安装Lua 5.3.5。

```
cd /usr/local/src
wget http://www.lua.org/ftp/lua-5.3.5.tar.gz
tar -zxvf lua-5.3.5.tar.gz
cd /usr/local/src/lua-5.3.5 && echo "INSTALL_TOP= /usr/local/lua_5.3.5" >> Makefile
&& make linux && make install
```

   iii. 下载并安装LuaRocks 2.2.2。

```
cd /usr/local/src
wget http://keplerproject.github.io/luarocks/releases/luarocks-2.2.2.tar.gz
tar -xzvf luarocks-2.2.2.tar.gz
cd luarocks-2.2.2
./configure --prefix=/usr/local/luarocks_2.2.2 --with-lua=/usr/local/lua_5.3.5
make build
make install
```

iv. 在*/etc/profile*文件中添加以下内容。

```
export LUA_HOME=/usr/local/lua_5.3.5
export LUALOCKS_HOME=/usr/local/luarocks_2.2.2
PATH=$PATH:$HOME/bin:$LUALOCKS_HOME/bin:$LUA_HOME/bin
export PATH
export LUA_PATH="$LUALOCKS_HOME/share/lua/5.3/?.lua;?.lua;;"
export LUA_CPATH="$LUALOCKS_HOME/lib/lua/5.3/?.so;?.so;;"
```

v. 刷新*/etc/profile*配置文件。

```
source /etc/profile
```

vi. 安装Lua组件。

```
luarocks install luasocket
luarocks install lua-resty-jit-uuid
luarocks install luaunit
luarocks install lua-cjson 2.1.0-1
```

vii. 确认Lua组件是否安装成功。

```
luarocks list
```

2. 下载并安装OpenResty Nginx。

```
yum install pcre-devel openssl-devel gcc curl postgresql-devel
cd /usr/local/src
wget -c https://openresty.org/download/openresty-1.15.8.1rc2.tar.gz
tar -zxvf openresty-1.15.8.1rc2.tar.gz
cd openresty-1.15.8.1rc2
./configure --prefix=/usr/local/openresty/ --with-http_stub_status_module --with-luajit
--without-http_redis2_module --with-http_iconv_module --with-http_postgres_module --wit
h-stream && gmake && gmake install
export PATH=/usr/local/openresty/nginx/sbin:$PATH
```

3. 下载并安装skywalking-nginx-lua。

i. 下载并解压skywalking-nginx-lua安装包。

```
cd /usr/local/skywalking-nginx-lua
wget https://mirrors.tuna.tsinghua.edu.cn/apache/skywalking/nginx-lua/0.5.0/skywalk
ing-nginx-lua-0.5.0-src.tgz
tar -xzvf skywalking-nginx-lua-0.5.0-src.tgz
```

ii. 修改*nginx.conf*文件中的*lua_package_path*和*startBackendTimer*。

例如：

- *lua_package_path*改为 `/usr/local/skywalking-nginx-lua/lib/?.lua;;"` 。
- *startBackendTimer*改为 `require("skywalking.client"):startBackendTimer("http://tracing-analysis-dc-hz.aliyuncs.com/adapt_***")` 。

iii. 启动skywalking-nginx-lua。

```
nginx -c /usr/local/skywalking-nginx-lua/examples/nginx.conf
```

查看结果

稍等片刻后到链路追踪Tracing Analysis控制台查看，如果有监控数据则表示追踪成功。如有关于Nginx监控的疑问，欢迎通过钉钉账号osfriend交流。

5.13. 开始监控Ingress

5.13.1. 使用Nginx-Ingress-tracing实现链路追踪

ACK提供了Nginx-Ingress-tracing链路追踪功能，通过Nginx-Ingress-tracing链路追踪功能可以接入到链路追踪控制台中。您可以在链路追踪控制台查看调用链路、链路拓扑等。本文介绍如何使用Nginx-Ingress-tracing实现链路追踪。

前提条件

- 开通相关服务并授权
- 创建Kubernetes托管版集群

背景信息

阿里云提供了链路追踪服务，为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑等，能够帮助快速分析和诊断分布式应用架构下的性能瓶颈，提高微服务时代下的开发诊断效率。ACK提供的Ingress-Nginx-Controller支持集成链路追踪服务，您可以根据需求开启该功能，查看链路追踪数据。

步骤一：获取接入点信息

获取接入点信息，根据实际情况选择客户端采集工具，本文以Zipkin为例。

1. 登录链路追踪Tracing Analysis控制台。
2. 在控制台页面顶部选择地域。

   > **说明** 地域需要与步骤二：在Ingress中开启链路追踪服务的ACK集群保持一致。

3. 在左侧导航栏单击**集群配置**，然后单击**接入点信息**页签。
4. 打开**显示Token**开关，然后在下方表格的**相关信息**列中，单击接入点信息末尾的复制图标。

   > **说明**
   >
   > - 建议选择v1版本接入点。
   > - 对于ACK集群应用，建议通过ARMS OpenTelemetry Collector上报应用数据。ARMS OpenTelemetry Collector支持本地集群内的链路采样与指标无损统计，在降低链路传输、存储成本的同时，不影响监控或告警指标的准确性。更多信息，请参见ARMS OpenTelemetry Collector

步骤二：在Ingress中开启链路追踪服务

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，选择**配置管理 > 配置项**。
5. 在配置项页面上方设置命名空间为**kube-system**，然后在名称搜索框中搜索nginx-configuration，找到nginx-configuration，然后单击nginx-configuration**操作**列的**编辑**。
6. 配置链路追踪方式。

   根据使用的链路追踪方式，需要配置不同的字段，本文采用zipkin来提供链路追踪服务。关于更多链路追踪方式信息，请参见准备工作概述。
   在**编辑**面板单击**添加**，设置**名称**为*zipkin-collector-host*，**值**为步骤一：获取接入点信息获取的接入点信息。

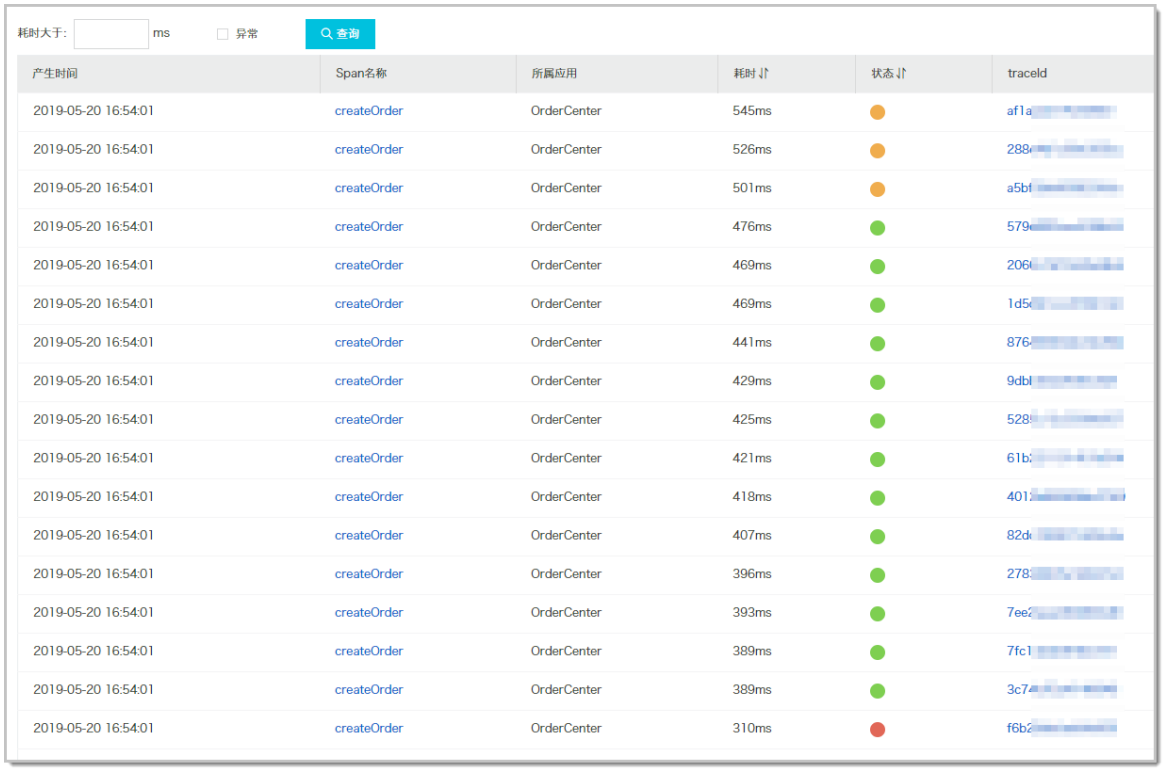
   > **说明** 该接入点信息需要去掉HTTP，并在末尾加上问号。例如接入点信息为 `http://tracing-analysis-dc-hz-internal.aliyuncs.com/adapt_******_******/api/v1/spans` ，则输入的值为 `tracing-analysis-dc-hz-internal.aliyuncs.com/adapt_******_******/api/v1/spans?` 。

7. 开启链路追踪服务。

   单击**添加**，设置**名称**为*enable-opentracing*，**值**为*true*，然后单击**确定**。

步骤三：查看调用链路数据

1. 登录链路追踪Tracing Analysis控制台。
2. 在控制台左侧导航栏中单击**应用列表**。
3. 在**应用列表**页面顶部选择地域，然后单击应用名称。
4. 在应用详情页面左侧导航栏中单击**接口调用**，在左侧的接口列表中单击调用链路页签，调用链路页签列出了该应用耗时最长的至多100个调用链路。关于更多链路数据，请参见查看接口调用情况。


耗时大于: ms 异常 查询

产生时间	Span名称	所属应用	耗时	状态	traceId
2019-05-20 16:54:01	createOrder	OrderCenter	545ms		af1a
2019-05-20 16:54:01	createOrder	OrderCenter	526ms		288
2019-05-20 16:54:01	createOrder	OrderCenter	501ms		a5bf
2019-05-20 16:54:01	createOrder	OrderCenter	476ms		579
2019-05-20 16:54:01	createOrder	OrderCenter	469ms		206
2019-05-20 16:54:01	createOrder	OrderCenter	469ms		1d5
2019-05-20 16:54:01	createOrder	OrderCenter	441ms		876
2019-05-20 16:54:01	createOrder	OrderCenter	429ms		9dbl
2019-05-20 16:54:01	createOrder	OrderCenter	425ms		528
2019-05-20 16:54:01	createOrder	OrderCenter	421ms		61b
2019-05-20 16:54:01	createOrder	OrderCenter	418ms		401
2019-05-20 16:54:01	createOrder	OrderCenter	407ms		82d
2019-05-20 16:54:01	createOrder	OrderCenter	396ms		278
2019-05-20 16:54:01	createOrder	OrderCenter	393ms		7ee2
2019-05-20 16:54:01	createOrder	OrderCenter	389ms		7fc1
2019-05-20 16:54:01	createOrder	OrderCenter	389ms		3c7
2019-05-20 16:54:01	createOrder	OrderCenter	310ms		f6b2

6.教程

6.1. 概览

在概览页面，您可以查看应用的性能关键指标，包括入口请求数、平均响应时间等总体指标，以及应用、入口和报警的相关指标。此外，您还可以查看接入流程与接入点信息。

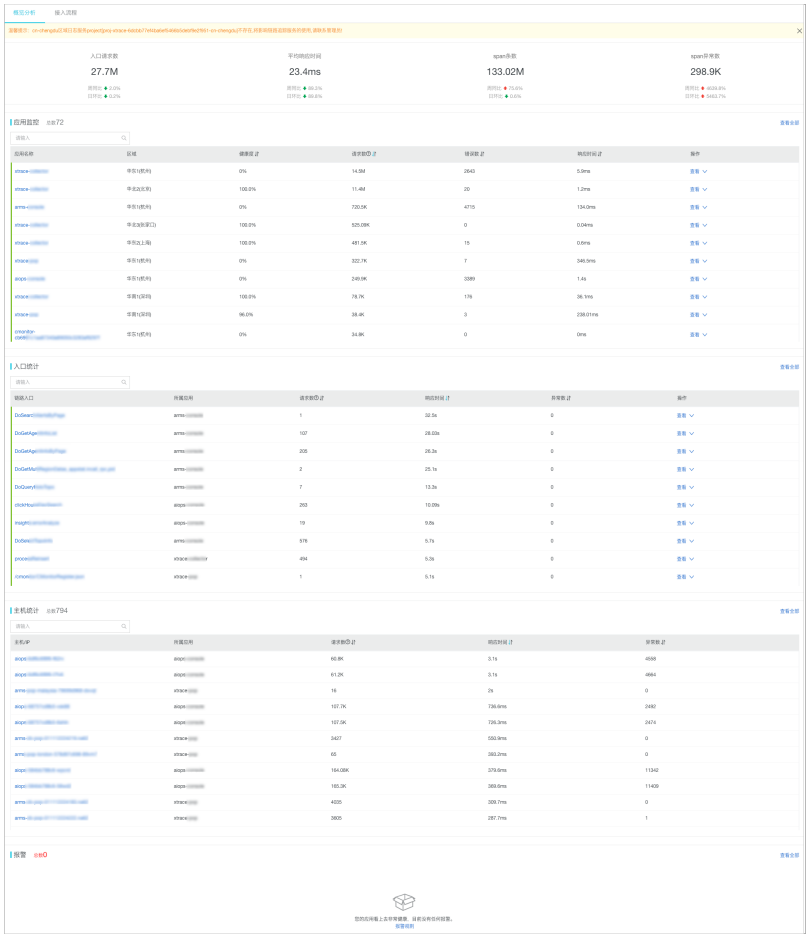
功能入口

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**概览**，在顶部单击**概览分析**页签。

概览分析

概览分析页签上展示以下关键指标：

- 选定时间内的入口请求数、平均响应时间、Span条数和Span异常数，以及这些指标和上周的同比、上一天的环比升降幅度。
- 应用的名称、地域、健康度、请求数、错误数和响应时间。
- 入口的名称、所属应用、请求数、响应时间和异常数。
- 主机的名称或IP、所属应用、请求数、响应时间和异常数。
- 报警的名称、触发状态、发生时间、报警内容、等级和所属规则。



接入流程

接入流程页签上展示接入流程以及您的接入进程。此外，您还可以查看接入点信息。

相关文档

- 开通相关服务并授权

6.2. 查看应用列表

应用列表页面展示了所有被监控应用的健康度得分、本日请求数、本日错误数等关键指标。您还可以为应用设置自定义标签并使用标签来筛选。

背景信息

应用列表页面会展示被监控应用的多项关键指标，其中的健康度得分是根据APDEX性能指数（Application Performance Index）性能指数计算的。该指数是国际通用的应用性能计算标准，将用户对应用的使用感受定义为三个等级：

- 满意（0~T）
- 可容忍（T~4T）
- 不满意（>4T）

计算公式为：

```
APDEX=(满意数+可容忍数/2)/总样本量
```

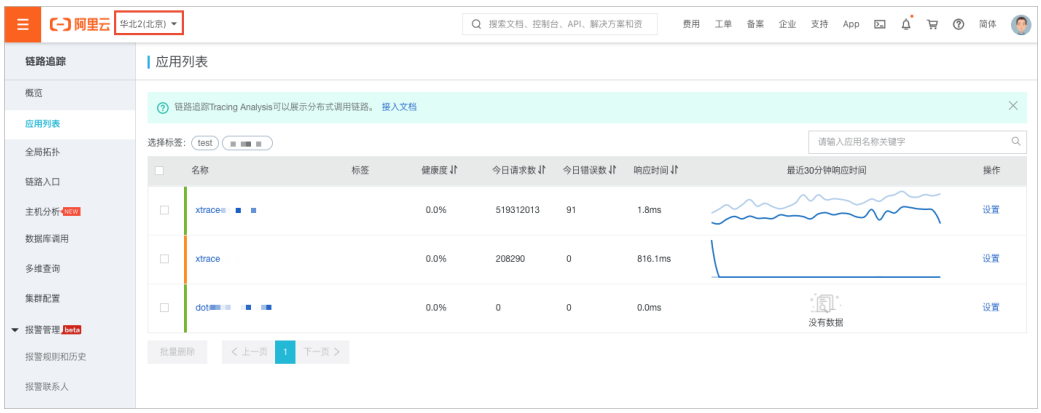
链路追踪取应用的平均响应时间作为计算指标，并将T定义为500毫秒。

功能入口

请按照以下步骤进入**应用列表**页面。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择目标地域。

应用列表页面



排序应用

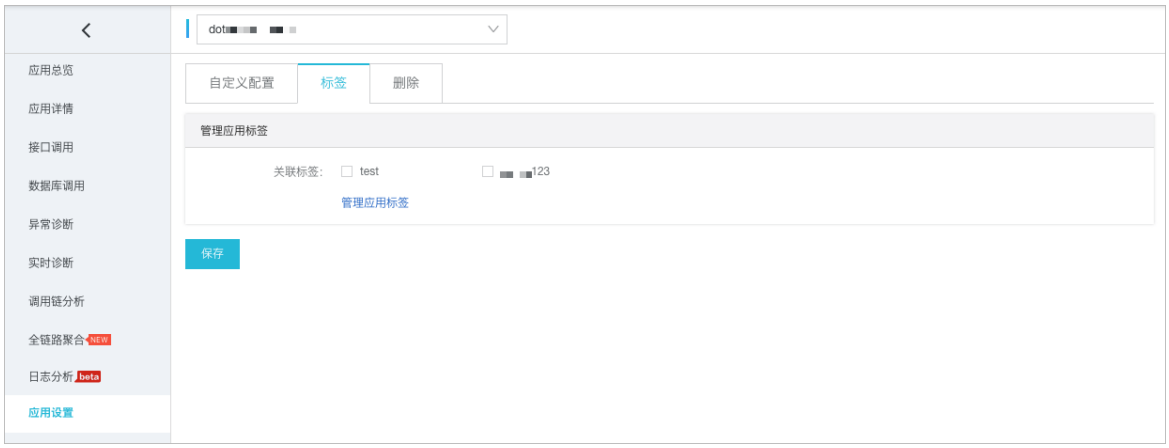
单击以下列标题旁边的箭头，即可按照相应条件升序或降序排列所有应用：

- **健康度**
- **今日请求数**
- **今日错误数**
- **响应时间**

设置应用标签

为应用设置自定义的标签后，可利用这些标签来筛选应用。

1. 将鼠标悬浮于指定应用的**标签**列上，并单击铅笔图标，然后选择**管理标签**。
   页面会跳转至**应用设置**下的**标签**页签。



2. 在**标签**页签下，单击**管理应用标签**。
3. 在弹出的**管理应用标签**的对话框中，单击⊕图标，输入您自定义的标签名称并单击**确定**。
4. 在弹出的**成功**对话框中单击**确定**，然后单击页面底部的**保存**。

利用标签筛选应用

在应用列表上方的**选择标签**区域框中单击一个或多个标签，即可筛选出具有至少其中一个标签的所有应用。

相关文档

- 准备工作概述
- 管理应用和标签

6.3. 应用管理

6.3.1. 查看应用性能关键指标和拓扑图

应用总览页面展示应用的性能关键指标和拓扑结构。

背景信息

当应用数据被上报至链路追踪Tracing Analysis后，链路追踪Tracing Analysis会开始全方位监控您的应用。通过**应用总览**页面，您可以快速查看应用性能关键指标，并通过应用拓扑图浏览应用的上下游依赖关系。

功能入口

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域，然后单击应用名称。
   在弹出的**应用总览**页面，您可以查看该应用的性能关键指标和拓扑图详情。

查看应用性能关键指标

在**应用总览**页面上，单击**概览分析**页签，您可以查看以下性能关键指标：

- 选定时间内的请求总量、平均响应时间、异常次数和Span数量，以及这些指标和上周的同比、前一天的环比升降幅度。
- 应用被上游调用的次数/耗时、调用下游服务次数/耗时的时序曲线。
- 调用最慢的10个接口及其平均响应时间时序曲线。

概览分析页签

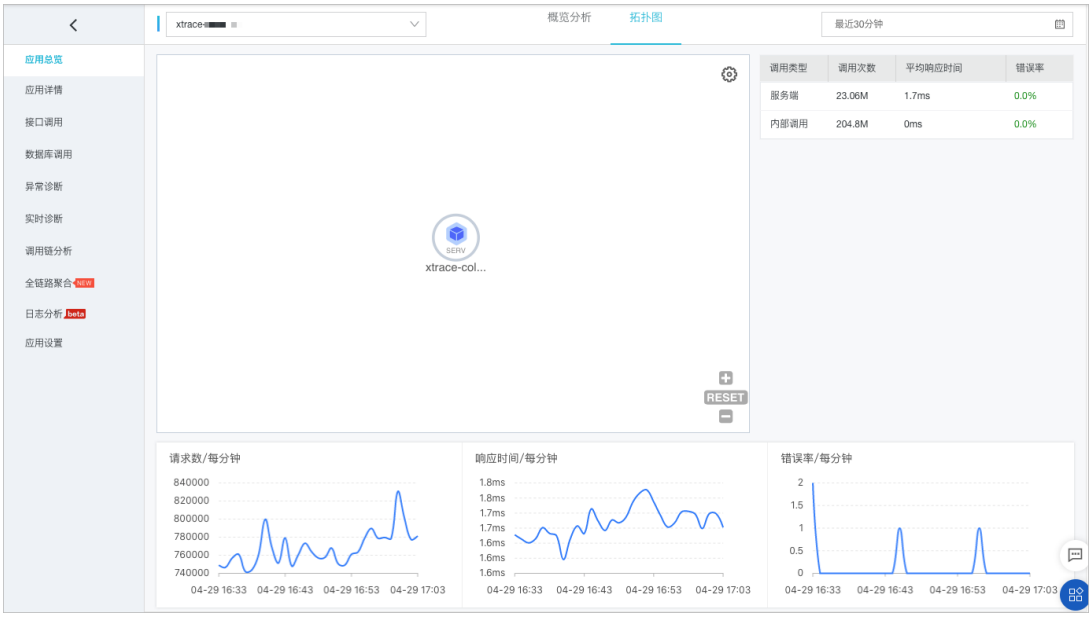
查看应用拓扑图

在**应用总览**页面上单击**拓扑图**页签，您可以更加直观地看到应用的上下游组件以及与它们的调用关系，从而更快速地找出应用的性能瓶颈。

拓扑图页签展示以下信息：

- 选定时间内的应用调用关系拓扑图。
- 选定时间内客户端、服务端和内部调用的调用次数、平均响应时间和错误率。
- 选定时间内每分钟的请求数、响应时间和错误率时序图。

拓扑图页签



设置查询时间范围

您可以选择预设的时间范围，或者输入自定义的时间范围。

- 单击页面右上角的时间选择框，然后单击一个预设的时间范围，例如**最近30分钟**、**今天**、**本周**。
- 如果没有符合需求的预设时间范围，则单击**自定义**，然后在日历中选择起始和截止时间，或者在文本框内手动输入，并单击**确定**。

> **说明** 日期的格式为 `YYYY-MM-DD`，时间的格式为 `HH:MM`。

查询时间范围选择器

相关文档

- 准备工作概述

- 管理应用和标签

6.3.2. 查看应用详情

应用详情页面可展示应用在所部属的每一台机器上的关键性能指标、调用拓扑图和调用链路。

查看关键性能指标和拓扑图

应用详情页面的**概览**页签列出了部署该应用的所有机器。您可以按照响应时间、请求数或错误数对该列表排序。在机器列表中选中一台机器，即可在**概览**页签上查看应用的详细调用拓扑，以及请求数、响应时间、错误数的时序曲线。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域，然后单击应用名称。
3. 在左侧导航栏中单击**应用详情**，在左侧的机器列表中单击**全部**或一台机器，然后在**概览**页签上查看调用拓扑图和关键性能指标。

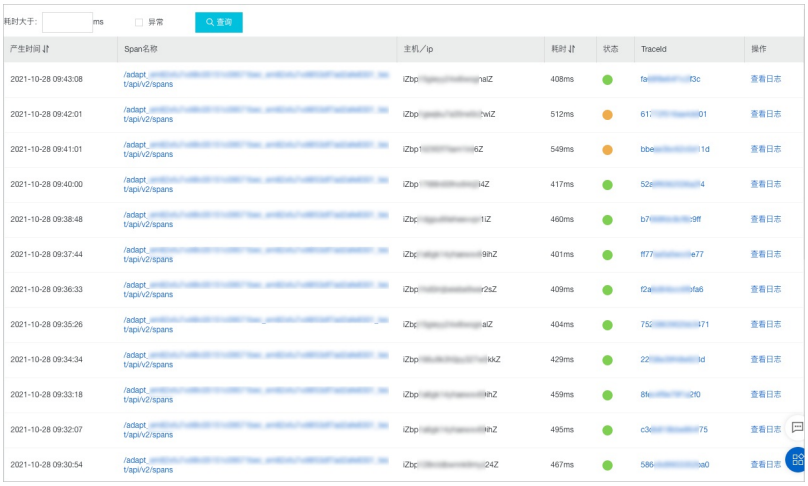
> **说明** 单击**响应时间**、**请求数**、**异常数**页签，并单击旁边的箭头，即可按照相应的条件对部署该应用的所有机器进行排序。在搜索框中输入关键字，即可动态筛选出符合关键字的机器。

> **说明** 如需切换至相同地域的其他应用，可单击页面左上角的应用名称下拉列表并选择其他应用。

查看调用链路

调用链路页签列出了该应用在所选时间段内所选机器上耗时最长的至多100个调用链路。

调用链路页签



> **说明** **状态**列中的绿色图标表示耗时小于500毫秒，黄色图标表示耗时介于500毫秒至1000毫秒之间，红色图标表示耗时大于1000毫秒或者有Tag Key为 `error` 。

在**调用链路**页签上可以按需执行以下操作：

- 在**耗时大于**调整框中输入一个时间值（单位为毫秒），并单击**查询**，即可筛选出耗时大于指定值的调用链路。
- 选择**异常**并单击**查询**，即可筛选出有异常的调用链路。
- 单击**产生时间**或**耗时**右侧的上下箭头，即可按照对应的条件进行升序或降序排列。

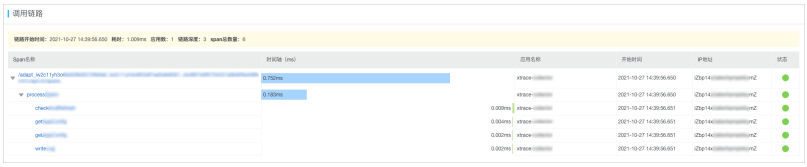
- 单击TraceID，即可在新窗口中打开**调用链路**页面，并查看该调用链路的瀑布图。

查看调用链瀑布图

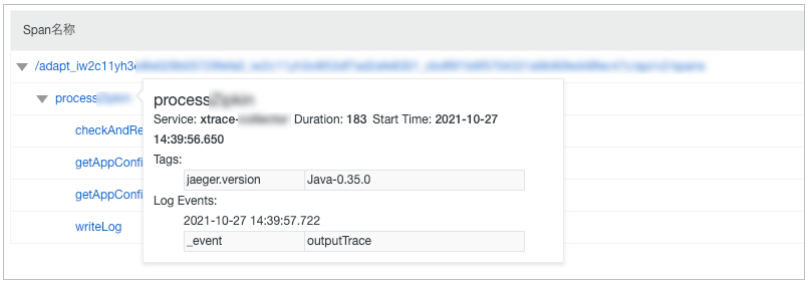
在**调用链路**页面上，您可以看到调用链路的Span名称、时间轴、应用名称、开始时间、IP地址（或机器名称）以及状态。

> **说明** **IP地址**字段显示的是IP地址还是机器名称，取决于**应用设置**页面上的显示配置。详情请参见管理应用和标签。

调用链路页面



将鼠标悬浮于单个Span名称上，还可以查看该服务的时长、开始时间、Tag和日志事件等信息。



设置查询时间范围

您可以选择预设的时间范围，或者输入自定义的时间范围。

- 单击页面右上角的时间选择框，然后单击一个预设的时间范围，例如**最近30分钟**、**今天**、**本周**。
- 如果没有符合需求的预设时间范围，则单击**自定义**，然后在日历中选择起始和截止时间，或者在文本框内手动输入，并单击**确定**。

> **说明** 日期的格式为 `YYYY-MM-DD` ，时间的格式为 `HH:MM` 。

查询时间范围选择器

6.3.3. 查看接口调用情况

接口调用页面展示客户端调用、服务端调用和本地调用中的接口（Span）调用性能指标，以及链路上游和链路下游的接口调用情况。

查看接口调用性能指标

接口调用页面列出了应用调用中涉及的全部接口（Span）。您可以按照响应时间、请求数或错误数对该列表排序。在接口列表中选中一个接口，即可在**概览**页签上查看应用的拓扑图和接口调用性能指标的时序曲线，包括请求数、响应时间和错误数。

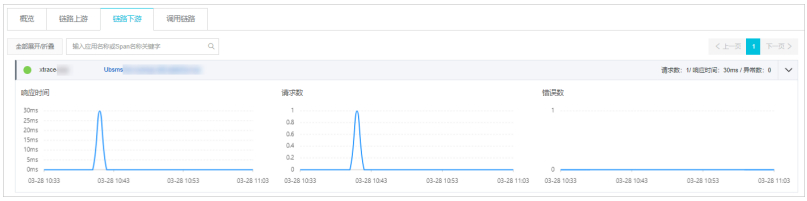
1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域，然后单击应用名称。
3. 在左侧导航栏中单击**接口调用**，在左侧的接口列表中单击一个接口，然后在**概览**页签上查看该接口的拓扑图和性能指标。
   - 单击**响应时间**、**请求数**、**错误数**页签，并单击页签标题旁边的上箭头或下箭头，即可按照相应的条件对所有接口进行升序或降序排序。
   - 在搜索框中输入关键字，即可动态筛选出符合关键字的接口。

> **说明** 如需切换至相同地域的其他应用，可单击页面左上角的应用名称下拉列表并选择其他应用。

查看链路上游和链路下游的接口调用情况

链路上游和**链路下游**页签分别列出了应用上游（调用应用的一方）和应用下游（被应用调用的一方）的接口及其调用性能指标，包括响应时间、请求数和错误数。

链路下游页签



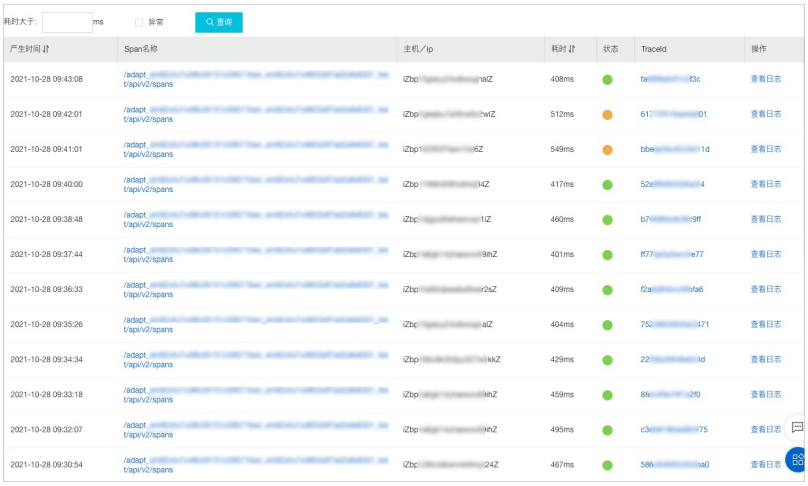
在**链路上游**和**链路下游**页签上，可按需执行以下操作：

- 在页签顶部单击**全部折叠/展开**，即可折叠或展开下方的所有接口。
- 在页签顶部的搜索框内输入应用名称或接口（Span）名称的关键字，并单击放大镜图标，即可筛选出符合条件的接口。
- 单击接口信息所在的折叠面板，或者单击行末的上箭头或下箭头，即可展开或折叠该接口的性能指标信息。

查看调用链路

调用链路页签列出了该应用在所选时间段内所选机器上耗时最长的至多100个调用链路。

调用链路页签



> **说明** **状态**列中的绿色图标表示耗时小于500毫秒，黄色图标表示耗时介于500毫秒至1000毫秒之间，红色图标表示耗时大于1000毫秒或者有Tag Key为 `error` 。

在**调用链路**页签上可以按需执行以下操作：

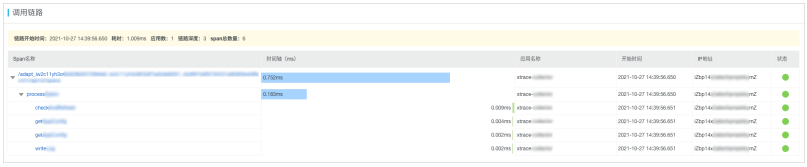
- 在**耗时大于**调整框中输入一个时间值（单位为毫秒），并单击**查询**，即可筛选出耗时大于指定值的调用链路。
- 选择**异常**并单击**查询**，即可筛选出有异常的调用链路。
- 单击**产生时间**或**耗时**右侧的上下箭头，即可按照对应的条件进行升序或降序排列。
- 单击TraceID，即可在新窗口中打开**调用链路**页面，并查看该调用链路的瀑布图。

查看调用链瀑布图

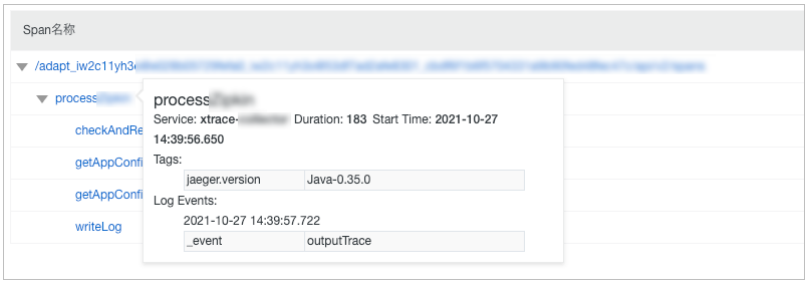
在**调用链路**页面上，您可以看到调用链路的Span名称、时间轴、应用名称、开始时间、IP地址（或机器名称）以及状态。

> **说明** **IP地址**字段显示的是IP地址还是机器名称，取决于**应用设置**页面上的显示配置。详情请参见管理应用和标签。

调用链路页面



将鼠标悬浮于单个Span名称上，还可以查看该服务的时长、开始时间、Tag和日志事件等信息。



设置查询时间范围

您可以选择预设的时间范围，或者输入自定义的时间范围。

- 单击页面右上角的时间选择框，然后单击一个预设的时间范围，例如**最近30分钟**、**今天**、**本周**。
- 如果没有符合需求的预设时间范围，则单击**自定义**，然后在日历中选择起始和截止时间，或者在文本框内手动输入，并单击**确定**。

> **说明** 日期的格式为 `YYYY-MM-DD`，时间的格式为 `HH:MM`。

查询时间范围选择器

相关文档

- 准备工作概述
- 管理应用和标签

6.3.4. 查看数据库调用

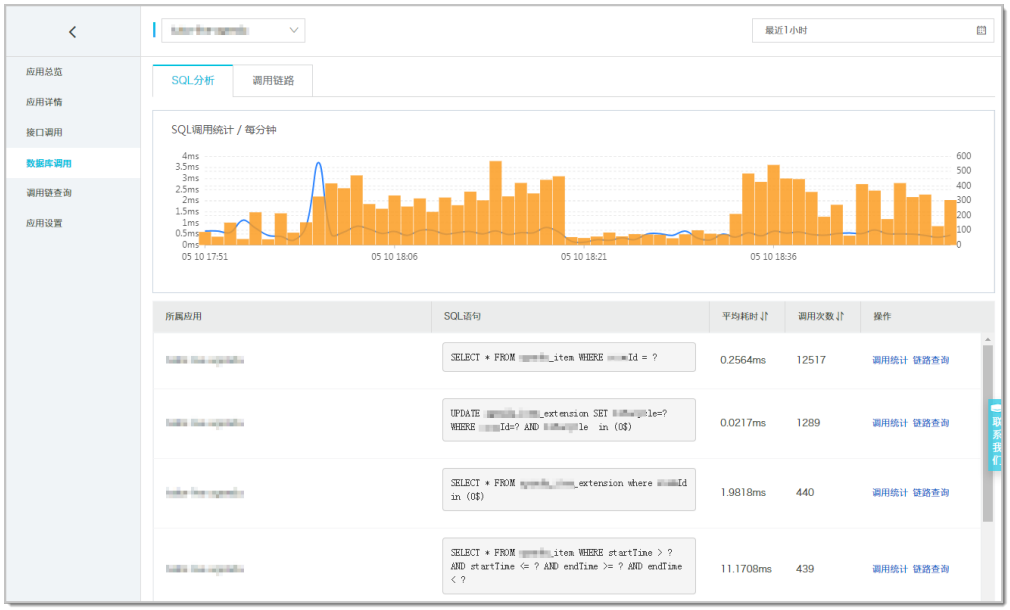
6.3.4.1. 查看SQL性能分析

数据库调用页面可展示各SQL语句的调用次数、平均耗时和相关调用链路，帮助您定位SQL性能问题。

查看SQL分析

请按照以下步骤操作，查看应用的SQL统计和分析。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域，然后单击应用名称。
3. 在左侧导航栏中单击**数据库调用**，然后在**SQL分析**页签上查看以下指标：
   - 选定时间内的每分钟SQL调用次数和平均耗时图表。
   - 选定时间内具体SQL语句的调用次数和平均耗时。
4. 在**SQL分析**页签上，按需执行以下操作：
   - 在**操作**列中单击**调用统计**，即可查看选定时间内具体SQL语句的每分钟调用次数和平均耗时图表。
   - 在**操作**列中单击**链路查询**，即可在**调用链路**页签上查看与对应SQL语句相关的所有调用链路。



> **说明** 如需切换至相同地域的其他应用，可单击页面左上角的应用名称下拉列表并选择其他应用。

设置查询时间范围

您可以选择预设的时间范围，或者输入自定义的时间范围。

- 单击页面右上角的时间选择框，然后单击一个预设的时间范围，例如**最近30分钟**、**今天**、**本周**。
- 如果没有符合需求的预设时间范围，则单击**自定义**，然后在日历中选择起始和截止时间，或者在文本框内手动输入，并单击**确定**。

> **说明** 日期的格式为 `YYYY-MM-DD` ，时间的格式为 `HH:MM` 。

查询时间范围选择器

相关文档

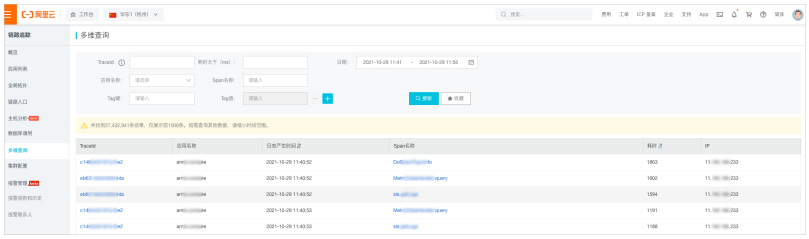
- 准备工作概述
- 管理应用和标签

6.3.4.2. 查询调用链

本文介绍如何利用多维查询功能查询调用链。

操作步骤

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**多维查询**。



3. 在**多维查询**页面上，按需输入以下参数的值，并单击**搜索**。
   - 固定查询参数
     - TraceID
     - 耗时大于（毫秒）
     - 日期
     - 应用名称
     - Span名称

- 可选查询参数：Tag键
  从**Tag键**下拉列表选择Tag的键，然后在**Tag值**下拉列表选择Tag的值。如需添加Tag键和Tag值作为查询条件，请单击字段右侧的蓝色加号图标。

4. 在查询结果中，单击TraceID即可进入**调用链路**标签页查看调用链详情。

后续步骤

- 如需保存当前查询参数配置，请单击**收藏**，并设置别名。
- 如需删除收藏的所有查询参数配置，单击别名右侧的删除图标。

相关文档

- 准备工作概述
- 查看应用列表

6.3.5. 实时诊断

数据上报至链路追踪后，链路追踪组件对其进行实时聚合计算，一般会存在一定的延迟。您在定位问题时，如果希望看到实时结果，则可以使用实时诊断功能快速展示诊断结果。

背景信息

- 实时诊断会启动临时存储机制，上报数据后，无需进行实时统计即可快速展示诊断结果。
- 实时诊断不会影响正常统计，5分钟后会自动关闭此功能。关闭后，可重新开启此功能。
- 实时诊断页面默认定时刷新，每10秒刷新一次。您也可以关闭定时刷新功能。

查看调用链信息

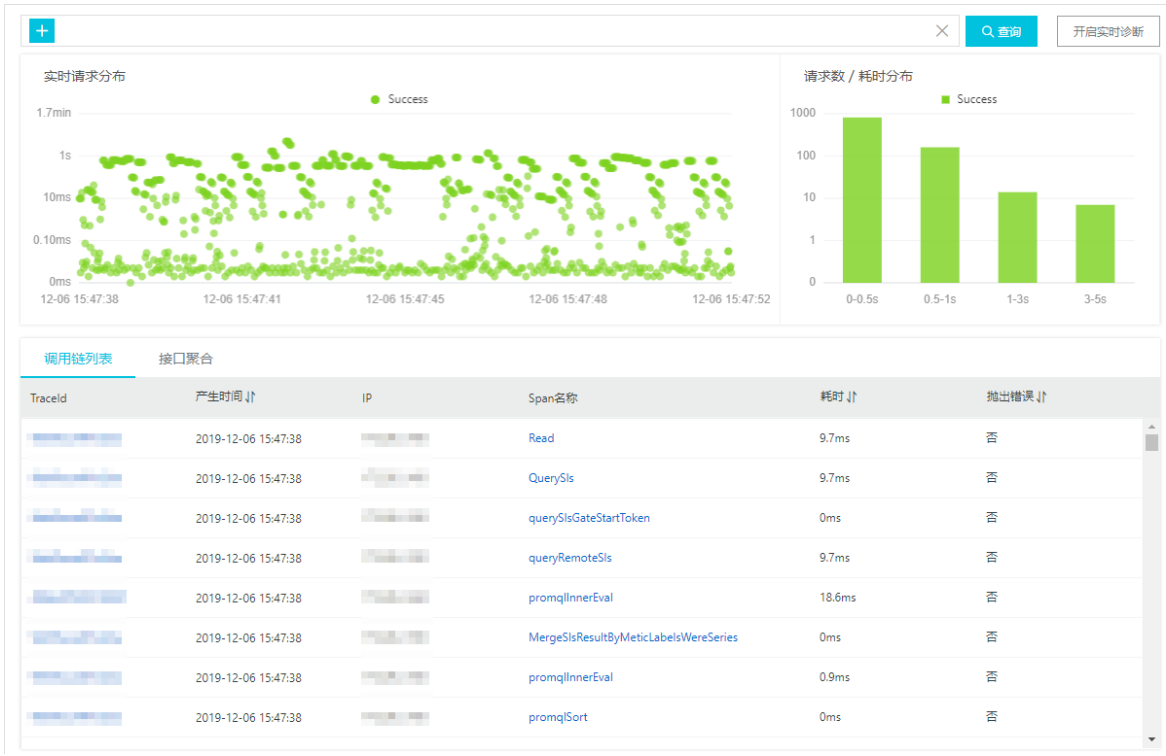
1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域，然后单击目标应用名称。
3. 在左侧导航栏中单击**实时诊断**，在**实时诊断**页面上，单击上方的 + 图标，可同时或分别添加**Span名称**、**IP**和**Tag**三个过滤条件。

   > **说明** 在添加过滤条件时，可添加单个Span名称或IP和多个Tag。

4. 单击**查询**，可查看过滤后的调用链信息，包括：
   - 实时请求响应时间分布的点阵图。

     > **说明** 使用鼠标框选点阵图中的某个区域，可以拉取此区域的实时诊断结果。

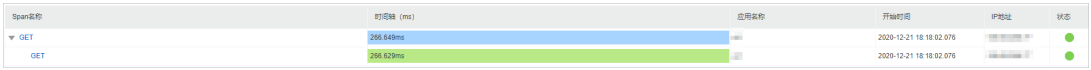
   - 请求数 / 耗时分布图。
   - 调用链信息列表。



调用链瀑布图

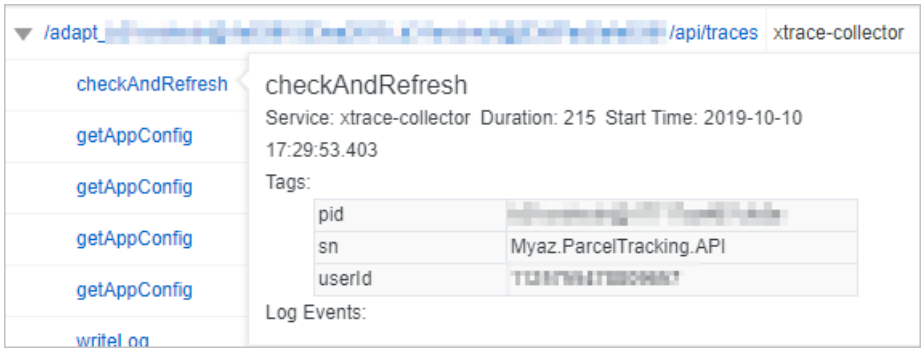
在**调用链路**页面上，您可以看到调用链的Span名称、时间轴、应用名、开始时间、IP地址和状态等信息。

> **说明** **IP地址**字段显示的是IP地址还是机器名称，取决于**应用设置**页面上的显示配置。更多信息，请参见管理应用和标签。



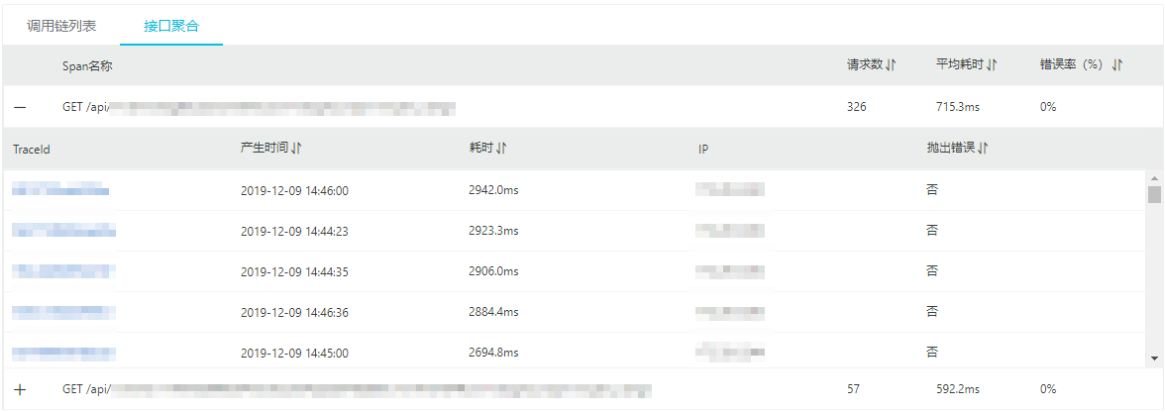
1. 单击**调用链列表**页签的单个Trace ID，即可在新窗口打开**调用链路**页面，并查看该调用链路的瀑布图。
2. （可选）在**调用链**区域下，您可以执行以下操作：

   将鼠标悬浮于Span名称上，可以查看该Span的时长、开始时间、Tag和日志事件等信息。



查看接口聚合列表

1. 单击**接口聚合**页签，查看将调用链按照Span名称聚合后的接口聚合列表。

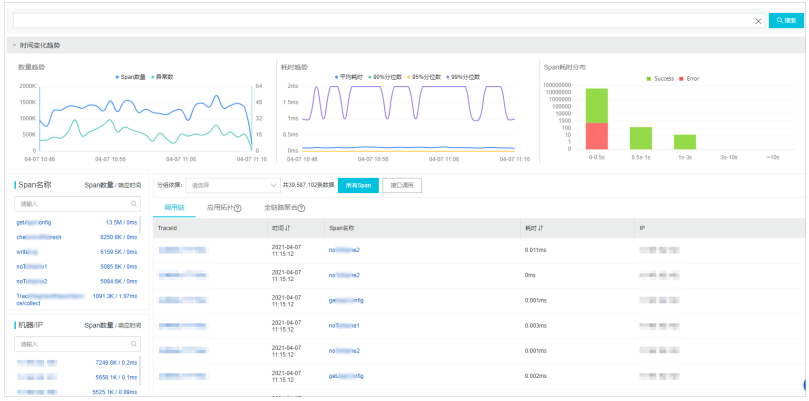


6.3.6. 调用链分析

通过**调用链分析**页面对应用的调用链信息进行分析后，您可以查看链路中部分监控指标的时序曲线，以及调用链列表、应用拓扑和全链路聚合信息。

功能入口

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**。
3. 在**应用列表**页面顶部选择地域，然后单击目标应用名称。
4. 在左侧导航栏中单击**调用链分析**。



开启全量实时分析

如果您需要对指定采样比例的存储数据进行全量实时分析，您可以在设置指定采样比例后，开启实时全量分析功能，查看对应的调用链信息。设置指定采样比例的操作，请参见采样存储。

> **注意** 如果没有设置采样比例或者采样比例设置为100%，那么开启实时全量分析功能后，显示的数据为当前账号下的存储数据。

1. 在**调用链分析**页面右上角单击**开启全量实时分析**。
2. 在右侧选择需要全量分析的时间段。
   调用链分析页面将实时刷新当前时间段的全量分析数据。

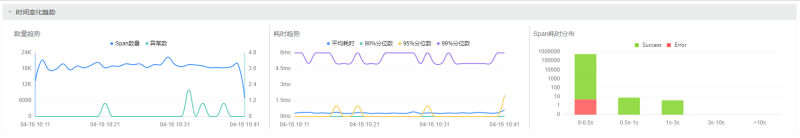
筛选调用链

1. 单击**调用链分析**页面右上角的时间选择框，设置查询时间范围。
2. 单击**调用链分析**页面顶部文本框，在弹出的筛选窗口中进行以下操作后，单击**搜索**。

- 在**耗时**参数后输入响应时间的最小值和最大值，可查询此耗时区间的调用链信息。
- 选中**异常**复选框，可查询有异常的调用链信息。
- 添加**Span名称**、**机器/IP**或**标签**三个过滤条件。在添加过滤条件时，可添加多个**Span名称**、**机器/IP**或**标签**。

时间变化趋势

时间变化趋势区域显示Span数量和异常数的时序曲线、Span耗时趋势曲线图和Span耗时分布图。

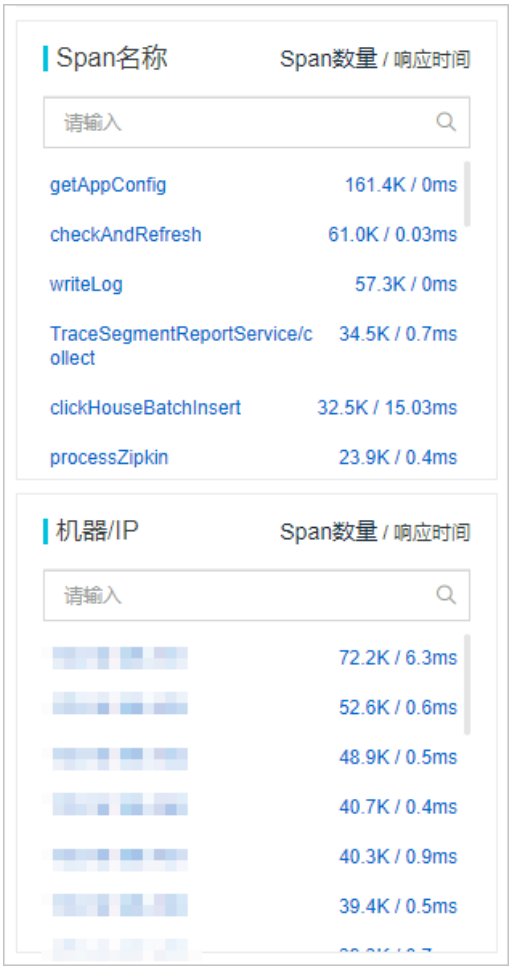


在**时间变化趋势**区域下，您可以执行以下操作：

- 将光标移到统计图上，查看统计情况。
- 使用光标选中一段时间，查看指定时间段的统计情况。
- 单击图例，隐藏或显示数据。

Span名称和机器/IP列表

在**Span名称**和**机器/IP**区域显示了所有Span和机器的列表。



在**Span名称**和**机器/IP**区域下，您可以执行以下操作：

- 在列表文本框中输入关键字，单击搜索图标，可以筛选当前列表中的Span名称或机器/IP。

- 单击列表右上角的**Span数量**或**响应时间**，可以将列表按Span数量或响应时间从大到小排序。
- 单击Span名称或机器的IP，可以将目标Span名称或机器的IP设置为筛选条件。

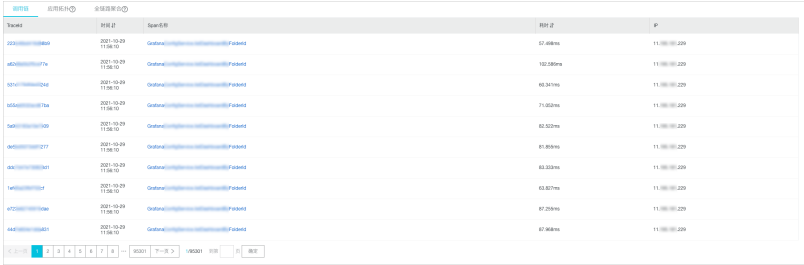
分组列表

1. 在**调用链分析**页面的**分组依据**区域设置分组字段。
2. 在右侧选择查询列表范围为所有Span或接口调用。
   - 所有Span：基于所有Span查询的当前字段的分组列表。
   - 接口调用：基于第一个Span查询的当前字段的分组列表。

   单击字段名称，可以查看当前字段名称对应的调用链列表。

调用链

在**调用链**区域显示了调用链的Trace ID、时间、Span名称、耗时和IP。

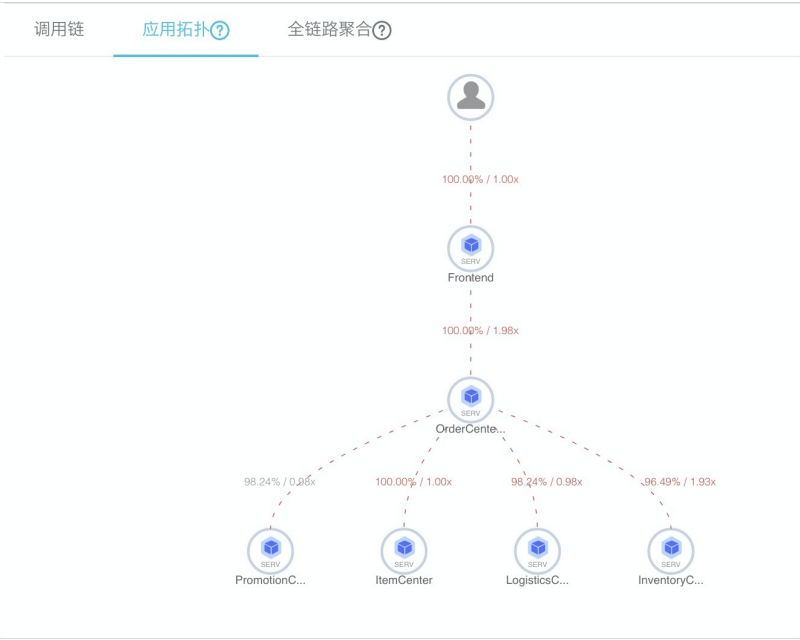


在**调用链**区域下，您可以执行以下操作：

- 单击调用链Trace ID，可以查询目标调用链的调用链瀑布图。更多信息，请参见调用链瀑布图。
- 将鼠标悬浮于Span名称上，可以查看该Span的时长、开始时间、Tag和日志事件等信息。

应用拓扑

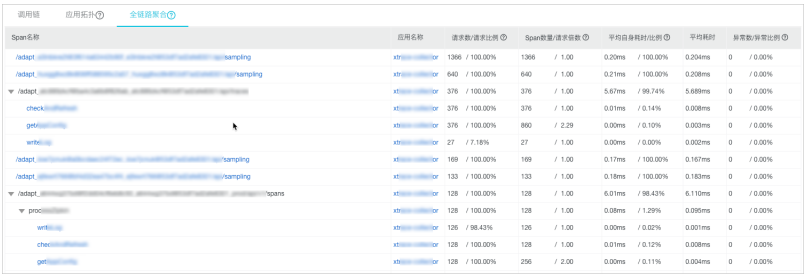
应用拓扑主要展示应用间依赖关系的拓扑图，以及各应用之间的请求比例、调用倍数和耗时比例等信息。基于性能体验考虑，应用拓扑最多支持拉取5000条链路请求进行聚合。



> **说明**
>
> - 请求比例=应用对外调用的请求数/应用总请求数。 例如有100个请求进入上层应用A， 而从A调用下层应用B的只有90个请求，那么A到B的请求比例为90%。（因为在应用A中，可能存在if判断进行过滤，导致一些请求不会进入应用B。）
> - 调用倍数=应用对外调用的Span数/应用总Span数。例如有100个Span进入上层应用A， 而从A调用下层应用B的有300个Span，那么A到B的调用倍数为3。 例如A到B显示为90%/3x， 表示应用A中有90%的请求会去调用应用B，应用A平均调用3次应用B。

全链路聚合

全链路聚合是将调用链根据Span名称和应用名进行聚合的调用链路表。基于性能体验考虑，实时聚合最多支持拉取5000条链路请求进行聚合。



> **说明**
>
> - 请求数/请求比例：请求比例表示调用当前Span节点的请求比例数。例如总请求数为100个，请求比例为10%表示有10个请求调用当前Span。计算公式=当前Span的请求数/总请求数×100%。
> - Span数/请求倍数：请求倍数表示平均每个请求数调用当前Span的次数，例如1.5x表示平均每个请求会调用当前Span 1.5次 。计算公式=Span数/Span的请求数。
> - 平均自身耗时/比例：平均自身耗时表示不包括子Span的平均耗时，例如Span A到B中，A耗时为10毫秒，B耗时为8毫秒，那么A的自身耗时为2毫秒。计算公式=Span耗时-Sum（子Span耗时）。如果是异步调用的话，将不会减去子耗时，计算公式=Span耗时。
> - 异常数/异常比例：异常比例表示出现异常的请求比例，例如3%表示有3%的请求出现异常。计算公式=异常请求数/总请求数。异常请求数不等于异常数，当请求倍数大于1时，一个异常请求可能对应多个异常数。

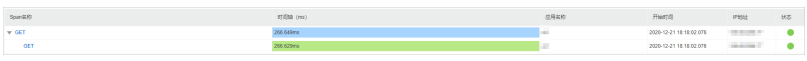
在**全链路聚合**区域下，您可以执行以下操作：

- 将鼠标悬浮于蓝色Span名称上，显示**推荐调用链**提示信息，可查看与此Span关联的调用链。单击选定的Trace ID，可以查看调用链瀑布图。更多信息，请参见调用链瀑布图。
- 单击应用名称，可以查看目标应用的应用总览。更多信息，请参见查看应用性能关键指标和拓扑图。

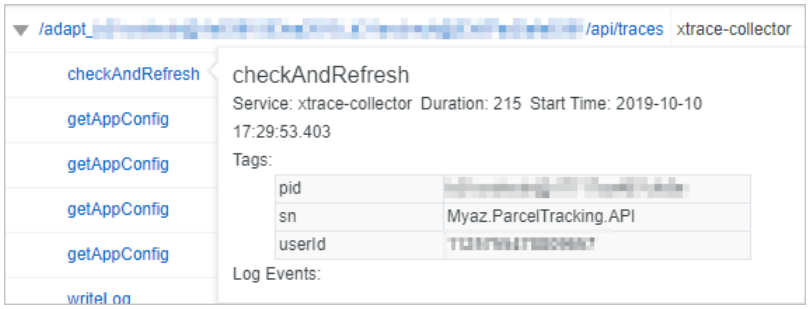
调用链瀑布图

在**调用链路**页面上，您可以看到调用链的Span名称、时间轴、应用名称、开始时间、IP地址和状态等信息。

> **说明** **IP地址**字段显示的是IP地址还是机器名称，取决于**应用设置**页面上的显示配置。更多信息，请参见管理应用和标签。



在**Span名称**列，您可以将鼠标悬浮于单个Span名称上，查看该Span的时长、开始时间、Tag和日志事件等信息。



6.3.7. 日志分析

链路追踪的日志分析功能依赖日志服务，在链路追踪控制台直接展示日志中心页面。当应用出现业务异常问题时，您可以在链路追踪控制台查看日志分析，精准定位业务异常。

前提条件

- 已接入应用监控。具体操作，请参见应用监控接入概述。
- 已开通日志服务SLS。登录日志服务控制台，根据页面提示开通日志服务。
- 已创建Project。具体操作，请参见创建Project。
- 已创建Logstore，具体操作，请参见创建Logstore。
- **已经将应用的业务日志上报到日志服务，具体操作，请参见数据采集概述。**

步骤一：关联业务日志

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**。
3. 在**应用列表**页面顶部选择地域，然后单击目标应用名称。
4. 在左侧导航栏中单击**应用设置**，并在右侧单击**自定义配置**页签。
5. 在**自定义配置**页签的**业务日志关联设置**区域，绑定Project和Logstore，设置关联索引。

   > **说明**
   > 关联索引用于检索TraceID，查看链路对应的日志。
   > - 如果日志服务中配置的索引为全文索引，此处设置关联索引为全文索引。
   > - 如果日志服务中配置的索引为 `traceId` 的字段索引，此处设置关联索引为对应的 `traceId` 的索引。在日志中打印TraceID的操作，可以查看对应的链路追踪框架的TraceID打印功能说明。
   >
   > 配置索引的操作，请参见配置索引。

6. 在**自定义配置**页签左下角单击**保存**。

步骤二：查询并分析日志

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**。
3. 在**应用列表**页面顶部选择地域，然后单击目标应用名称。

4. 在左侧导航栏，单击**日志分析**。
5. 在右侧页面，执行以下操作：
   i. 在搜索框中输入查询分析语句。

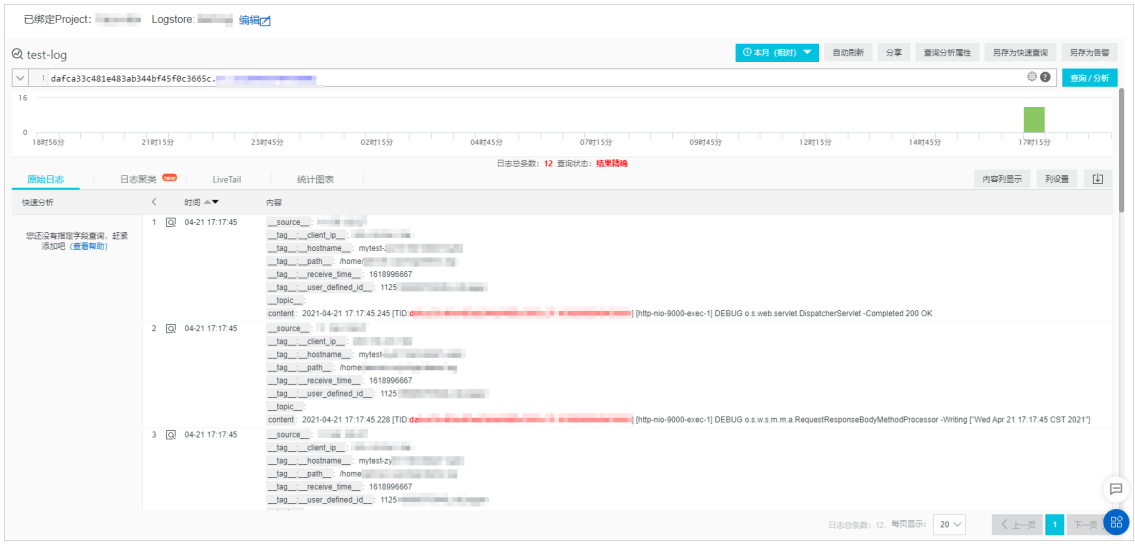
      查询分析语句由查询语句和分析语句构成，格式为查询语句|分析语句，查询分析语句语法请参见查询语法、SQL分析语法。

   ii. 设置查询分析的时间范围。

      您可以设置相对时间、整点时间和自定义时间。

      > **说明** 查询结果有1分钟以内的误差。

   iii. 单击**查询/分析**，查看查询分析结果。



6.3.8. 管理应用和标签

在应用设置页面上，您可以控制是否显示机器名称和采集应用数据，也可以管理应用的自定义标签和删除应用。

背景信息

在链路追踪Tracing Analysis控制台上，当需要显示应用所部属的机器时，默认情况下显示的是机器的IP地址，但是您也可以在**应用设置**页面上选择改为显示机器名称。

> **说明** 仅设置生效之后产生的新数据会受影响。例如，如果将**显示机器名**开关打开并保存，则仅此后生成的新数据中才会显示机器名，以前产生的数据中仍显示IP地址。

示例：显示机器的IP地址

示例：显示机器的名称

针对一个应用，如需停止对其计费，关闭数据采集即可。

在**应用设置**页面上，您也可以设置是否为当前应用使用特定的标签，以及管理账号下的全部标签。当不再需要一个应用时，您可以删除应用。

打开应用设置

请按照以下步骤打开应用设置页面。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**应用列表**，并在**应用列表**页面顶部选择地域。
3. 在**应用列表**页面上，单击目标应用**操作**列中的**设置**。

显示机器名

按照以下步骤操作，可改为显示应用所部属机器的名称。

1. 在**应用设置**页面上单击**自定义配置**页签（默认显示）。
2. 在**自定义配置**页签的**显示配置**区域框中，打开**显示机器名称**开关。
3. 单击页面底部的**保存**。

> **说明** 仅设置生效之后产生的新数据会受影响。例如，如果将**显示机器名**开关打开并保存，则仅此后生成的新数据中才会显示机器名，以前产生的数据中仍显示IP地址。

停止采集应用数据

如需停止应用计费，请按照以下步骤停止采集应用数据。

> **注意** 关闭数据采集将丢弃此后上报的所有数据。如果目标应用有下游链路，则调用链路将不完整。如果集群设置中的“采集数据”设为“开启”或“关闭”，则本设置无效。

1. 在**应用设置**页面上单击**自定义配置**页签（默认显示）。
2. 在**采集配置**区域框中，选择**关闭**。
3. 单击页面底部的**保存**。

为应用启用或停用标签

请按照以下步骤为应用启用或停用现有的标签。

1. 在**应用设置**页面上单击**标签**页签。
2. 在**应用标签管理**区域框中，勾选要为应用启用的标签，取消勾选不用的标签。
3. 单击页面底部的**保存**。

管理账号下的全部标签

如需管理账号下的全部标签，例如添加可供所有应用选用的标签，或全局删除一个现有的标签，请按照以下步骤操作。

1. 在**应用设置**页面上单击**标签**页签。
2. 在**管理应用标签**区域框中单击**管理应用标签**。
3. 在**管理应用标签**对话框中，按需采取以下操作。

   管理应用标签对话框

   - 如需创建新标签，则单击+图标，并在文本框内输入标签。

- 如需删除现有标签，则将鼠标悬浮于标签上，并单击左侧的**X**图标。

  > **注意** 如果在**管理应用标签**对话框中删除现有标签，则所有已经启用该标签的应用将会失去该标签。

- 如需修改已有的标签名称，则将鼠标悬浮于标签上，单击右侧的图标，在文本框内输入新标签名称，并单击任意空白处使修改生效。

4. 单击对话框底部的**确定**。
5. 在**成功**对话框中单击**确定**。

删除应用

请按照以下步骤删除不需要的应用。

1. 在**应用设置**页面上单击**删除**页签。
2. 在**删除应用**区域框中单击**删除**，并在**删除应用**对话框中单击**删除**。

6.4. 集群配置

6.4.1. 采样存储

在采样存储页面可以查看上报和存储的Span数据，并设置数据分析的采样比例。

背景信息

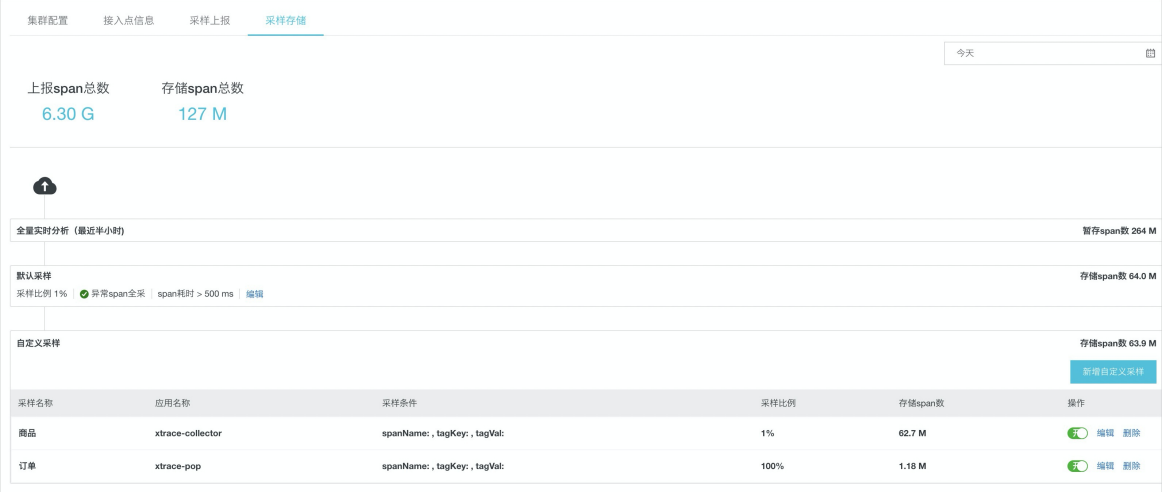
调用链路中90%以上的数据是重复的，价值并不大。您可以通过配置采样规则，只存储符合规格的数据，达到节约成本的目的。

例如：上报100万条Span数据，通过配置采样规则后，只存储符合规则的1万条Span数据，存储成本和查询效率都会得到提升。

> **注意** 当一个Span符合异常或者耗时规则时候，为保证链路的完整性，将会保存这个Span对应的完整链路。

功能入口

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中单击**集群配置**。
3. 在右侧页面单击**采样存储**页签。



数据说明

- 上报Span总数：当前账号下收到的所有上报Span总数。
- 存储Span总数：基于设置的自定义采样应用，按照一定采样规则过滤后存储的Span总数。
- 全量实时分析（最近半小时）：最近半小时内，上报的用于全量实时分析的Span总数。

上报的数据先运行默认采样规则，再运行自定义采样。

例如：上报100万条Span数据，其中符合异常和耗时规则的为1万条。自定义规则中的Span数为2万，默认采样比例为10%，采样存储逻辑如下：

1. 运行默认采样。优先存储符合异常和耗时规则的为1万条Span数据。再存储10%的条Span数据，即10万条。
2. 运行自定义采样。检查符合自定义采样规则的Span数据，符合采样规则的Span数据为2万条。
3. 可得出一个共存储13万条Span数据。

设置默认采样规则

1. 在**采样存储**页签的**默认采样**区域，单击**编辑**。
2. 设置以下参数后，单击**保存**。
   - 采样比例：设置调用链的采样率，输入百分比的数字部分即可，例如输入10代表采样10%。
   - 异常全采：存在`error`标签的调用链自动存储。
   - Span耗时：设置采样的Span耗时的最低阈值，即大于该数值的Span会被存储。

   设置完成后，您可以在**默认采样**区域右侧查看该采样规则下存储的Span数量。

新增自定义采样

1. 在**采样存储**页签的**自定义采样**区域，单击**新增自定义采样**。
2. 在**添加自定义采样**面板中设置采样名称。
3. 在**过滤条件**区域设置应用名，根据需要设置span名称、tag键和tag值。

   单击**查询**，可以预览当前过滤条件下的所有调用链。

参数	说明
应用名	采样的应用的名称。

参数	说明
span名称	采样的Span名称。
tag键	应用的标签键。
tag值	应用的标签值。

4. 设置采样比例，然后单击**确定**。
   设置完成后，您可以在**自定义采样**区域对采样应用进行开启或关闭、编辑、删除等操作，并可以在右上角查看该采样规则下存储的Span数量。

6.5. 报警管理

6.5.1. 创建报警

通过创建报警，您可以制定针对特定监控对象的报警规则。当规则被触发时，系统会以您指定的报警方式向报警联系人分组发送报警信息，以提醒您采取必要的问题解决措施。

前提条件

- 创建联系人：仅可将联系人分组设为报警的通知对象。

背景信息

默认报警条件：

- 为避免您在短时间内收到大量报警信息，系统24小时内对于持续的重复报警信息仅发送一条消息。
- 如果5分钟内没有重复报警，则会发送恢复邮件，通知数据恢复正常。
- 发送恢复邮件后，报警的状态会重置。如果该报警再次出现，会被视为新报警。

报警控件本质是数据集的数据展示方式，所以在创建报警控件的同时，会创建一个数据集来存储报警控件的底层数据。

> **说明** 新建报警大约在10分钟内生效，报警判断会存在1~3分钟的延时。

创建报警

若需为应用监控任务创建一个JVM-GC次数同比报警，具体操作步骤如下：

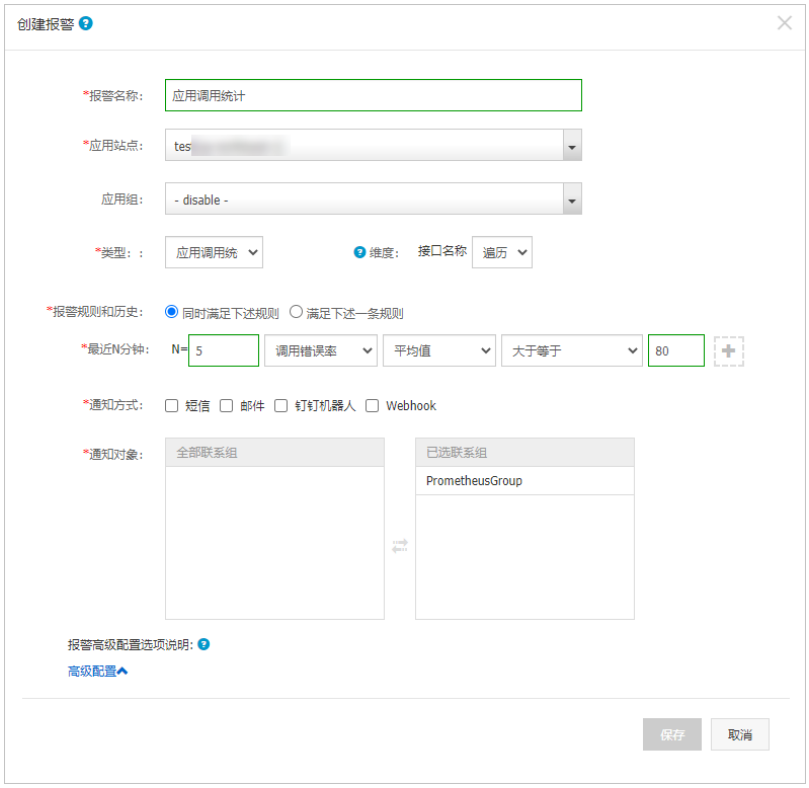
1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警规则和历史**。
3. 在**报警规则和历史**页面的右上角单击**创建报警**。
4. 在**创建报警**对话框中输入所有必填信息，完成后单击**保存**。
   i. 填写**报警名称**，例如：应用调用统计。
   ii. 在**应用站点**列表中选择应用。
   iii. 在**类型**列表中选择监控指标的类型，例如：**应用调用统计**。
   iv. 设置**维度**为**遍历**。
   v. 设置**报警规则和历史**为**同时满足下述规则**。

vi. 编辑**最近N分钟**设置报警规则，例如：N=5时调用错误率平均值大于等于100%时则报警。

> **说明** 若需设置多条报警规则，单击**最近N分钟**右侧的图标，即可编辑第二条报警规则。

vii. 选择**通知方式**。例如：邮件。

viii. 设置**通知对象**。在**全部联系组**框中单击联系人分组的名称，该联系人分组出现在**已选联系组**框中，则设置成功。



通用基础字段含义

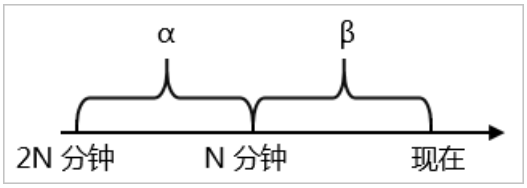
创建报警对话框的基础字段含义见下表。

字段	含义	说明
应用站点	已创建的监控任务。	在下拉菜单中选择。

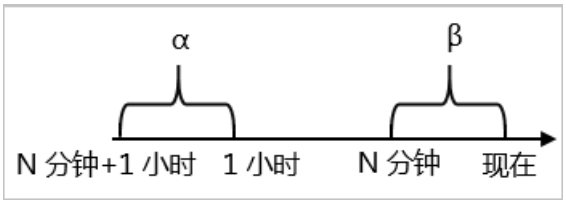
字段	含义	说明
报警维度	配置报警指标（数据集）的维度，可选择为：无、= 、遍历。	• 配置为无：报警内容中透出这个维度所有数值的和。 • 配置为= ：具体内容需手动填写。 • 配置为遍历：会在报警内容中透出实际触发报警的维度内容。
最近N分钟	报警判断最近N分钟内数据结果是否达到触发条件。	N的范围为：1~60分钟。
通知方式	支持邮件、短信、钉钉机器人和Webhook四种方式。	可勾选多种方式。若需设置钉钉机器人报警请参见。
报警静默期开关	可选择为开启或关闭，默认为开启状态。	• 开启报警静默期开关：若数据一直处于触发状态，首次触发报警后，24小时后才会发送第二次报警信息。当数据恢复正常，会收到数据恢复通知并解除报警。若数据再次触发报警，则会再次发送报警信息。 • 关闭报警静默期开关：若报警连续触发，将会每分钟发送一次报警信息。
报警级别	包括警告、错误和致命。	无
通知时间	报警发送时的通知时间。此时间范围外将不发送报警通知，但仍会有报警事件记录。	查看报警事件记录请参见。
通知内容	自定义的报警通知内容。	您可以编辑默认模板。在模板中，除$报警名称、$筛选、$报警时间和$报警内容等4个变量（暂不支持其它变量）为固定搭配，其余内容均可自定义。

通用复杂字段含义：环比与同比

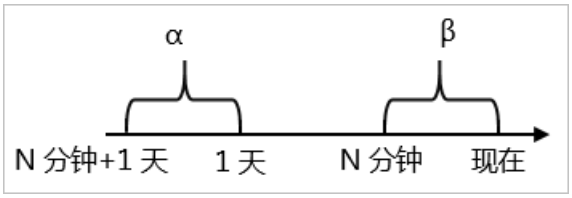
- 环比上升 / 下降%：若β为最近N分钟的数据（可选择为平均值、总和、最大值和最小值），α为前2N分钟到前N分钟的数据， 环比为β与α做比较。



- 与上小时同比上升 / 下降%：若β为最近N分钟的数据（可选择为平均值、总和、最大值和最小值），α为上小时最近N分钟的数据，与上小时同比为β与α做比较。



- 与昨日同比上升 / 下降%：若β为最近N分钟的数据（可选择为平均值、总和、最大值和最小值），α为昨日同一时刻最近N分钟的数据，与昨日同比为β与α做比较。



通用复杂字段含义：报警数据修订策略

报警数据修订策略可选择为补零、补一或补零Null（默认）。此功能一般用于无数据、复合指标和环比同比等异常的数据修复。

- 补零： 将被判断的数值修复为0。
- 补一： 将被判断的数值修复为1。
- 补零Null： 不会触发报警。

应用场景：

- 异常情况一：无数据
  用户A想利用报警功能监控页面访问量。创建报警时，选择前端监控报警，设置报警规则为N=5时页面访问量的总和小于等于10则报警。若该页面一直没有被访问，则没有数据上报，不会发送报警。为解决此类问题，可将报警数据修订策略勾选为补零，将没有收到数据视为收到零条数据，符合报警规则，即可发送报警。
- 异常情况二：复合指标异常
  用户B想利用报警功能监控商品的实时单价。创建报警时，选择自定义监控报警，设置变量a的数据集为当前总价，变量b的数据集为当前商品总数，报警规则为N=3时 （当前总价）/（当前商品总数）的最小值小于等于10则报警。若当前商品总数为0时，复合指标（当前总价）/（当前商品总数）的值不存在，则不会发送报警。为解决此类问题，可将报警数据修订策略勾选为补零，将复合指标（当前总价）/（当前商品总数）的值视为0，符合报警规则，即可发送报警。
- 异常情况三：指标环比、同比异常
  用户C想利用报警功能监控节点机用户使用CPU百分比。创建报警时，选择应用监控，设置报警规则为N=3时节点机用户使用CPU百分比的平均值环比下降100%则报警。若最近N分钟用户的CPU故障无法工作，即α无法获取，导致环比结果不存在，则不会发送报警。为解决此类问题，可将报警数据修订策略勾选为补一，将环比结果视为下降100%，符合报警规则，即可发送报警。

后续步骤

您可以在管理报警系统中查询和删除报警记录。

6.5.2. 管理报警

在报警规则和历史页面上，您可以管理账号下的所有报警规则，并查询报警事件和报警通知的历史记录。

管理报警规则

您创建的报警规则均会显示在**报警规则和历史**页面上。您可以对报警规则执行启动、停止、编辑、删除、查看报警详情等操作。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警规则和历史**。
3. （可选）在**报警规则和历史**页面的搜索框中输入报警名称，并单击**搜索**。

   > **说明** 您可以输入报警名称的一部分内容进行模糊搜索。

4. 在搜索结果列表的**操作**列中，按需对目标报警规则采取以下操作：

- 如需编辑报警规则，请单击**编辑**，在**编辑报警**对话框中编辑报警规则，并单击**保存**。
- 如需删除报警规则，请单击**删除**，并在**删除**对话框中单击**删除**。
- 如需启动已停止的报警规则，请单击**启动**，并在**启动**对话框中单击**启动**。
- 如需停止已启动的报警规则，请单击**停止**，并在**停止**对话框中单击**确定**。
- 如需查看报警事件历史和报警发送历史，请单击**报警历史**，并在**报警事件历史**和**报警发送历史**页签上查看相关记录。

查询报警历史

关于报警规则何时因为什么事件被触发的历史记录，以及触发报警规则后发送给指定报警联系人的报警通知历史记录，都可以在**报警历史**页面搜索。

1. 在左侧导航栏中选择**报警管理 > 报警规则和历史**。
2. 在**报警历史**页面上选择或输入报警的**类型**、**事件触发状态**和**报警名称**，并单击**搜索**。
3. 在**报警规则和历史**页签，可查看报警事件的历史记录。

> **说明** 仅触发状态为**已触发**（**触发**列中显示红色圆点）时才会发送报警通知。

4. 单击**报警发送历史**页签，可查看已触发报警发送的报警通知（短信、邮件等）记录。

6.5.3. 创建联系人

报警规则被触发时控制台会向您指定的联系人分组发送通知，而在创建联系人分组之前必须先创建联系人。创建联系人时，您可以指定联系人用于接收通知的手机号码和邮箱地址，也可以提供用于自动发送报警通知的钉钉机器人地址。

操作步骤

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警联系人**。
3. 在**联系人**页签上，单击右上角的**新建联系人**。

4. 在**新建联系人**对话框中输入联系人姓名，根据实际需求输入联系人手机号码、邮箱或钉钉机器人地址，设置是否接收系统通知，然后单击**确认**。

> 说明
> - 手机号码、邮箱和钉钉机器人至少填写一项。每个手机号码或邮箱只能用于一个联系人。
> - 至多可添加100个联系人。
> - 获取钉钉机器人地址的方法，请参见设置钉钉机器人报警。

相关操作

创建联系人后，您可以在**联系人管理**页面查询、编辑或删除联系人：

- 如需搜索联系人，请在**联系人**页签上，从搜索下拉框中选择**姓名**、**手机号**或**Email**，然后在搜索框中输入联系人姓名、手机号码或邮箱的全部或部分字符，并单击搜索
- 如需编辑联系人，请单击联系人右侧**操作**列中的**编辑**，在**编辑联系人**对话框中编辑信息，并单击**确定**。
- 如需删除单个联系人，请单击联系人右侧**操作**列中的**删除**，并在提示对话框中单击**删除**。
- 如需删除多个联系人，请选中目标联系人，单击**批量删除联系人**，并在提示对话框中单击**确定**。

6.5.4. 创建联系人分组

创建报警规则时，您可以将联系人分组指定为报警通知对象，当报警规则被触发时，链路追踪会向该联系人分组中的联系人发送报警通知。本文介绍如何创建联系人分组。

前提条件

创建联系人

操作步骤

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警联系人**。
3. 单击**联系人组**页签，并单击右上角的**新建联系组**。
4. 在**新建联系组**对话框中填写**组名**，选择**报警联系人**，并单击**确认**。

> 说明 如果**报警联系人**列表中没有选项，则您需要先创建联系人。

后续操作

- 如需搜索联系组，请在**联系人组**页签的搜索框中输入联系人组名称的全部或部分字符，并单击图标。

> 注意 英文搜索关键字区分大小写。

- 如需编辑联系组，请单击联系人组右侧的图标，并在**编辑联系组**对话框中编辑相关信息，然后单击**确认**。
- 如需查看联系组中的联系人信息，请单击联系人组左侧的图标来展开联系组。

> **说明** 您可以在展开模式下移除联系组中的联系人。如需移除，请单击目标联系人**操作**列中的**移除**。

- 如需删除联系组，请单击联系人组右侧的×图标，然后在弹出的提示对话框中单击**确认**。

> **注意** 删除联系组之前，请确保没有正在运行的监控任务，否则可能导致报警等功能失效。

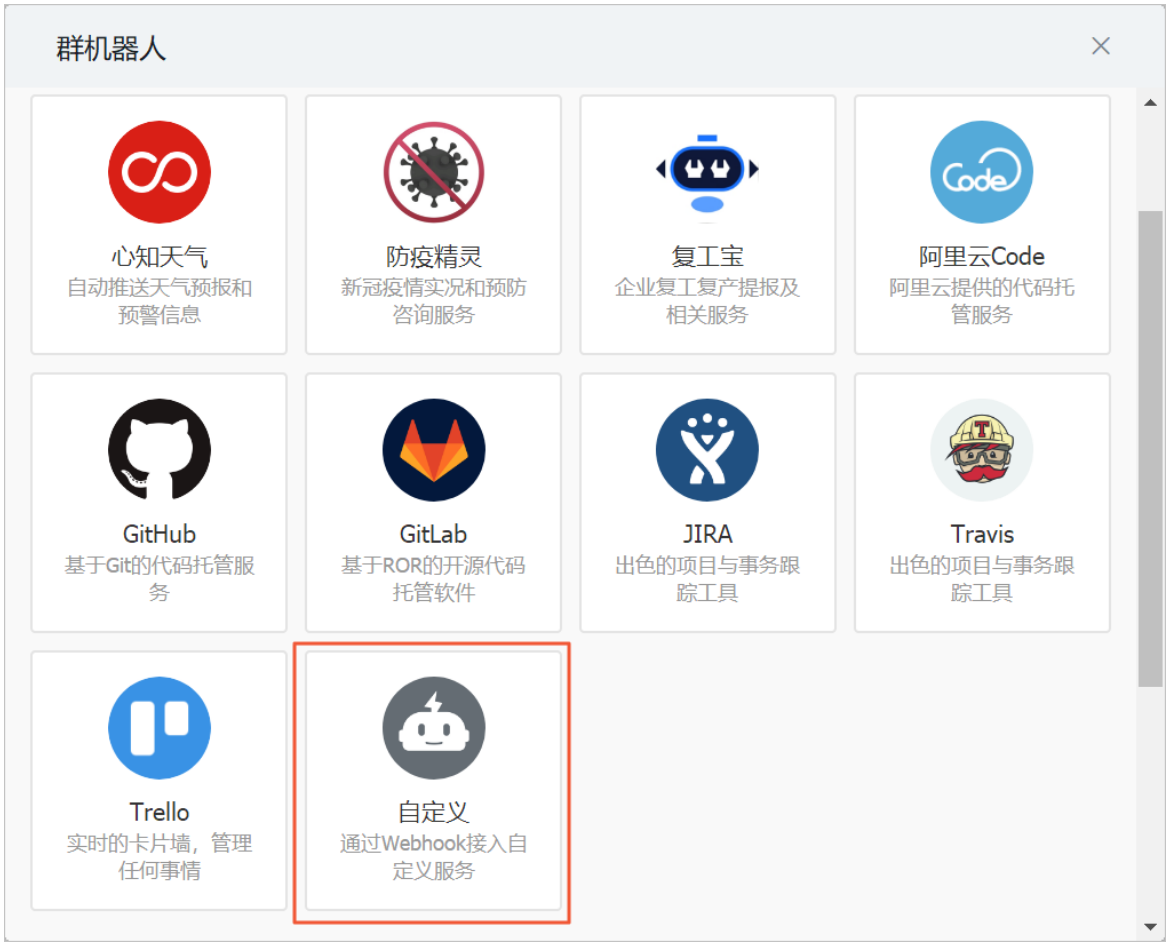
6.5.5. 设置钉钉机器人报警

设置钉钉机器人报警后，您可以通过指定钉钉群接收报警通知。本文介绍设置钉钉机器人报警的操作步骤。

添加自定义钉钉机器人并获取Webhook地址

请按照以下步骤添加自定义钉钉机器人并获取Webhook地址。

1. 在PC版钉钉上打开您想要添加报警机器人的钉钉群，并单击右上角的群设置图标。
2. 在**群设置**面板中单击**智能群助手**。
3. 在**智能群助手**面板单击**添加机器人**。
4. 在**群机器人**对话框单击**添加机器人**区域的 **+** 图标，然后选择添加**自定义**。



5. 在**机器人详情**对话框单击**添加**。
6. 在**添加机器人**对话框中编辑机器人头像和名称，选中必要的**安全设置**（至少选择一种），选中**我已阅读并同意《自定义机器人服务及免责条款》**。单击**完成**。

7. 在**添加机器人**对话框中复制生成的机器人Webhook地址，然后单击**完成**。

创建联系人

您可以创建一个新的联系人，或者将该地址添加到已有联系人信息中。此处以创建新的联系人为例。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警联系人**。
3. 在**联系人**页签上，单击右上角的**新建联系人**。
4. 在**新建联系人**对话框中，填写在添加自定义钉钉机器人并获取Webhook地址中获取的钉钉机器人Webhook地址，并单击**确认**。

创建联系组

您需要新建联系组，因为报警规则中只能将报警联系人分组设置为报警通知接收人，而不能直接发送给报警联系人。

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警联系人**。
3. 单击**联系人组**页签，然后单击右上角的**新建联系组**。
4. 在**新建联系组**对话框中填写**组名**，将在创建联系人中创建的联系人设置为**报警联系人**，并单击**确认**。

创建报警

如果您未创建报警，请先创建报警。前端监控报警、应用监控报警、自定义监控报警、Prometheus监控报警都可以设置钉钉机器人报警，以创建应用监控报警为例。

1. 登录链路追踪Tracing Analysis控制台。

2. 在左侧导航栏中选择**报警管理 > 报警规则和历史**。
3. 在**报警规则与历史**页面的右上角单击**创建报警 > 应用监控报警**。
4. 在**创建报警**对话框中输入所有必填信息，完成后单击**保存**。
   i. 填写**报警名称**，例如：JVM-GC次数同比报警。
   ii. 在**应用站点**列表中选择应用。
   iii. 在**类型**列表中选择监控指标的类型，例如：**JVM监控**。
   iv. 设置**维度**为**遍历**。
   v. 设置报警规则。
      a. 单击**同时满足下述规则**。
      b. 编辑报警规则，例如：N=5时JVM_FullGC次数的平均值与上小时同比上升100%时则报警。

      > **说明** 若需设置多条报警规则，单击**最近N分钟**右侧的 + 图标，即可编辑第二条报警规则。

   vi. 在**通知方式**区域选中**钉钉机器人**。
   vii. 将通知对象设置为创建联系组中创建的联系组。您可以在**全部联系组**列表中单击联系人分组的名称，该联系人分组出现在**已选联系组**列表中，则表示设置成功。

编辑报警

如需编辑告警规则，请按照以下步骤操作。以编辑应用监控报警为例。

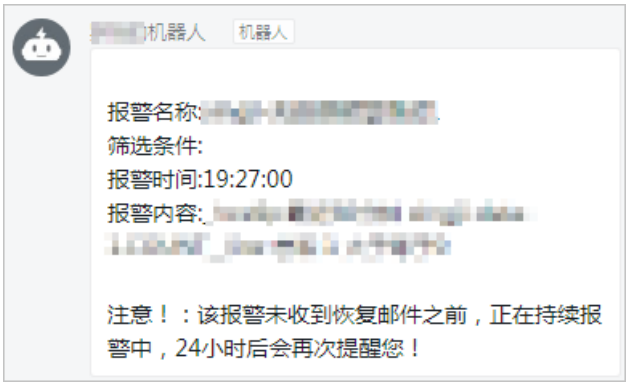
1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏中选择**报警管理 > 报警规则和历史**。
3. （可选）在**报警规则和历史**页面的搜索框中输入报警名称，并单击**搜索**。

   > **说明** 您可以输入报警名称的一部分内容进行模糊搜索。

4. 在搜索结果列表的**操作**列中，单击**编辑**。
5. 在**编辑报警**对话框中，进行相关操作，完成后单击**保存**。
   i. 将通知方式修改为**钉钉机器人**。
   ii. 将通知对象修改为创建联系组中创建的联系组。您可以在**全部联系组**框中单击联系人分组的名称，该联系人分组出现在**已选联系组**框中，则表示修改成功。

执行结果

操作至此，您已成功设置一个钉钉机器人报警。当报警触发时，您将在设置接收报警的钉钉群中收到报警通知。例如：



7.SDK参考

7.1. SDK简介

链路追踪Tracing Analysis支持基于OpenTracing标准的调用链路上报。链路追踪支持Jaeger、Zipkin和Skywalking客户端，以及Java、Go、Python、JS、C++、C#等语言。
链路追踪支持的语言及相应的客户端如下表所示。

链路追踪支持的语言及相应的客户端

语言	Jaeger源码工程	Zipkin源码工程	Skywalking源码工程
Java	https://github.com/jaegertracing/jaeger-client-java	https://github.com/openzipkin/zipkin	https://github.com/apache/skywalking
Go	https://github.com/jaegertracing/jaeger-client-go	https://github.com/openzipkin/zipkin-go	无
Python	https://github.com/jaegertracing/jaeger-client-python	https://github.com/Yelp/py_zipkin	无
JS	https://github.com/jaegertracing/jaeger-client-node	https://github.com/openzipkin/zipkin-js	https://github.com/SkyAPM/SkyAPM-nodejs
.NET	https://github.com/jaegertracing/jaeger-client-csharp	https://github.com/openzipkin/zipkin4net	https://github.com/SkyAPM/SkyAPM-dotnet
C++	https://github.com/jaegertracing/jaeger-client-cpp	https://github.com/flier/zipkin-cpp	无

请在下方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 Go应用 .NET应用

SkyWalking

Java应用

8.最佳实践

8.1. 使用SkyWalking和链路追踪进行代码级慢请求分析

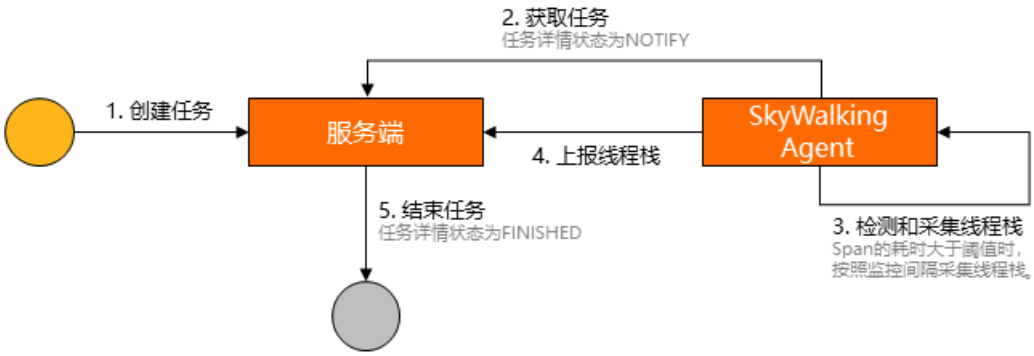
借助Java字节码注入技术，许多基于Java的框架可以实现自动埋点，从而帮助您了解慢请求具体发生在哪两个埋点之间，但这不足以定位代码层的问题。如需精确定位导致慢请求出现的代码方法，您可以搭配使用SkyWalking的慢请求分析功能和链路追踪。

前提条件

您已通过SkyWalking接入链路追踪，详情请参见通过SkyWalking上报Java应用数据。

背景信息

SkyWalking采集慢请求数据的工作流程如图所示。



> **说明** 仅SkyWalking 7.0及更高版本具备慢请求分析功能。

创建慢请求采集任务

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏单击**应用列表**，在页面左上角选择目标地域，并在**应用列表**页面单击目标应用的名称。
3. 在左侧导航栏单击**慢请求分析**，并单击**新建任务**。
4. 在**新建任务**对话框中输入以下信息，并单击**确认**。

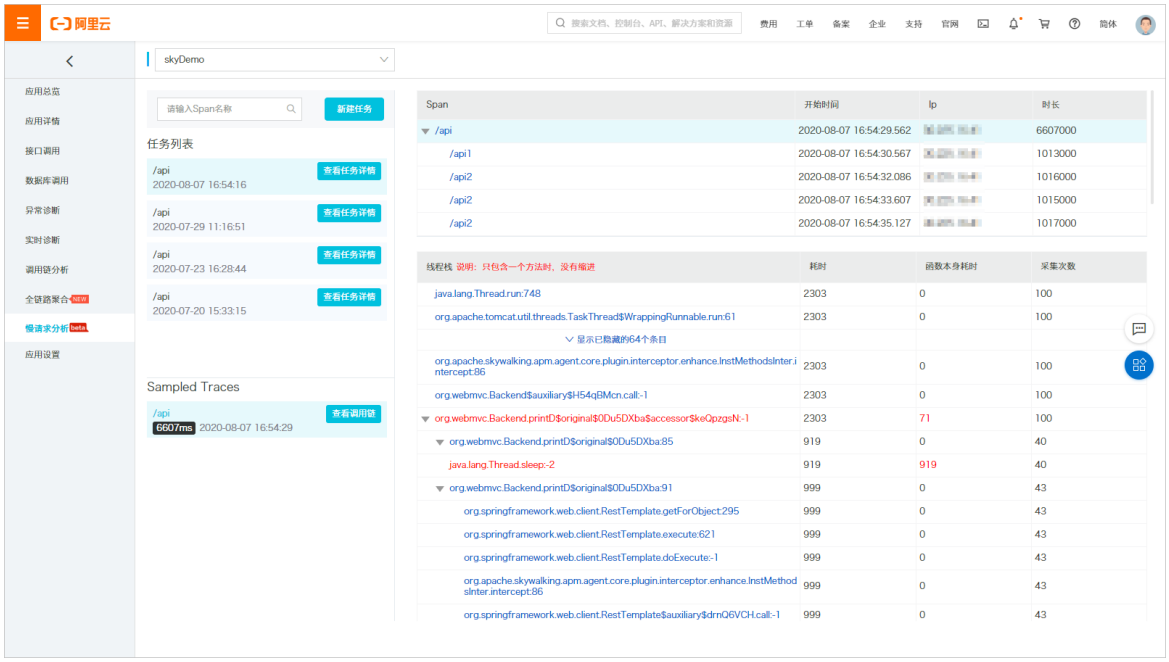
参数	描述	是否必填	示例值
Span名称	需监控的Span。	是	/api
监控时间	开始监控的时间。可选择**此刻**，或选择**设置时间**并设置一个自定义时间。	否	此刻
监控持续时间	监控持续的时间。	否	5 min
Span耗时阈值	仅分析耗时超过该阈值的Span。单位为ms。	否	30 ms
监控间隔	采集监控数据的时间间隔。	否	20 ms
最大采样数	被采集数据的Span最大数量。取值范围为1~9。	否	5

创建好的任务将显示在**任务列表**中。

定位导致慢请求出现的方法

以您设置的监控时间为起点，经过指定的监控持续时间后，耗时超过阈值的Span将显示在**Sampled Traces**区域。请按照以下步骤在页面右侧的线程栈详细信息定位导致慢请求出现的方法。

1. 在**慢请求分析**页面的**任务列表**区域单击目标任务。



耗时超过阈值的Span将显示在**Sampled Traces**区域。

2. 在**Sampled Traces**区域单击目标Span，并观察页面右侧的**线程栈**区域。
   以红色字体显示的方法名即为耗时超过所设置阈值的方法。您可以针对性地优化这些方法。

为什么监控持续时间结束后未采样到任何线程栈？

如果监控持续时间结束后未采样到任何线程栈，请按照以下步骤排查。

1. 在**慢请求分析**页面的**任务列表**区域单击目标任务右侧的**查看任务详情**。
2. 在**任务详情**对话框的**日志**区域查看详细信息。

- 如果**实例**字段末尾含有监控的Agent IP地址，且**操作类型**为 `EXECUTION_FINISHED` 或 `EXECUTION_NOTIFY` ，则说明网络连接正常，只是因为没有耗时超过阈值的Span。
- 如果不符合上述描述，则说明网络连接有问题，请稍后重试。

相关文档

- 博客：在线代码级性能剖析方面补全分布式追踪的最后一块短板

8.2. 使用Jaeger进行远程采样策略配置

通过远程采样策略配置，您可以直接在链路追踪控制台上配置采样策略，而不需要修改代码。

背景信息

采样指从全量采集的所有链路数据中，采集部分数据进行分析。采样决策包括采集数据和不采集数据。采样有以下类型：

- 事前采样：在用户访问开始时进行采样决策，Jaeger的采样是事前采样。
- 事后采样：在用户访问执行过程中或者访问过程后进行采样决策。

采样流程

以一个简单的调用关系为例：A->B->C（服务A调用服务B，同时，服务B调用服务C），服务A为头节点。当服务A收到不包含跟踪信息的请求时，Jaeger跟踪器将开始新的跟踪：

1. Jaeger跟踪器将为服务A、服务B以及服务C分配同一个随机Trace ID，并根据当前配置的采样策略做出采样决策。
2. 采样决策将与请求一起传播到服务B和服务C，这些服务将不做采样决策，而是接受头节点服务A的采样决策。

这种方法保证链路上所有Span都被记录在后端。 如果每个服务都做出自己的采样决策，那么您将很难在后端获得完整的调用链路。

> **说明** 只有头节点的采样策略会生效，例如服务A、B、C中都配置采样策略：
>
> - 对于调用链路为A->B->C，只有A的采样策略会生效。
> - 对于调用链路为B->C->A，只有B的采样策略会生效。

配置客户端

您需要在构建Trace对象时，将采样类型配置成Remote，采样的服务地址配置成链路追踪的采样地址。详情请参见通过Jaeger上报Java应用数据。将控制台上的接入点信息做简单修改后，您可以得到采样地址：

- 将 `api/traces` 改成 `/api/sampling` 。
- 去掉 `http://` 。

例如，将

```
http://tracing-analysis-dc-hz.aliyuncs.com/adapt_******_******/api/traces
```

修改为

```
tracing-analysis-dc-hz.aliyuncs.com/adapt_******_******/api/sampling
```

代码示例如下：

```
 io.jaegertracing.Configuration config = new io.jaegertracing.Configuration("your applicati
on name");
config.withSampler(new Configuration.SamplerConfiguration().withType("remote")
.withManagerHostPort("tracing-analysis-dc-hz.aliyuncs.com/adapt_ao123458@abcdefg_ao123458@f
ghijklm/api/sampling")
.withParam(0.1));
```

说明 如果您在客户端配置成远程采样方式，但是没有在服务端未配置采样策略，系统将会按照固定比例0.001（0.1%）进行采样。

配置服务端

您需要先在客户端完成配置，才能在服务端配置采样策略。您的配置将对所有配置了远程采样方式的Jaeger Client生效。

功能入口

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏单击**集群配置**，然后单击**采样上报**页签。

采样策略级别

- `default_strategy`：默认策略，必须配置。它还包括共享的per-operation策略，这些per-operation策略将适用于配置中未列出的，没有Service级别和Span级别的任何所有服务。
- `service_strategies`：Service级别的采样策略，可选。
- `operation_strategies`：Span级别的采样策略，可选。

采样策略类型

- 比例采样：`default_strategy`、`service_strategies`以及`operation_strategies`可配置。
- 速率采样：`default_strategy`以及`service_strategies`可配置。

示例如下：

```
{
  "service_strategies": [
    {
      "service": "foo",
      "type": "probabilistic",
      "param": 0.8,
      "operation_strategies": [
        {
          "operation": "op1",
          "type": "probabilistic",
          "param": 0.2
        },
        {
          "operation": "op2",
          "type": "probabilistic",
          "param": 0.4
        }
      ]
    },
    {
      "service": "bar",
      "type": "ratelimiting",
      "param": 5
    }
  ],
  "default_strategy": {
    "type": "probabilistic",
    "param": 0.5,
    "operation_strategies": [
      {
        "operation": "/health",
        "type": "probabilistic",
        "param": 0.0
      },
      {
        "operation": "/metrics",
        "type": "probabilistic",
        "param": 0.0
      }
    ]
  }
}
```

在示例中：

- 应用foo的所有操作均以0.8的比例进行采样，但op1和op2分别以0.2的比例和0.4的比例进行采样。
- 应用bar的所有Span埋点均以每秒5条链路的速率进行采样。
- 其他应用将以default_strategy定义的概率0.5进行采样。

另外，在此示例中，我们通过使用概率0禁用了对所有服务的/health和/metrics端点的跟踪。

8.3. 将链路追踪控制台页面嵌入自建Web应用

如需在自建Web应用中免登录查看链路追踪控制台的页面，您可将链路追踪控制台嵌入自建Web应用，以此避免系统间的来回切换。

教程概述

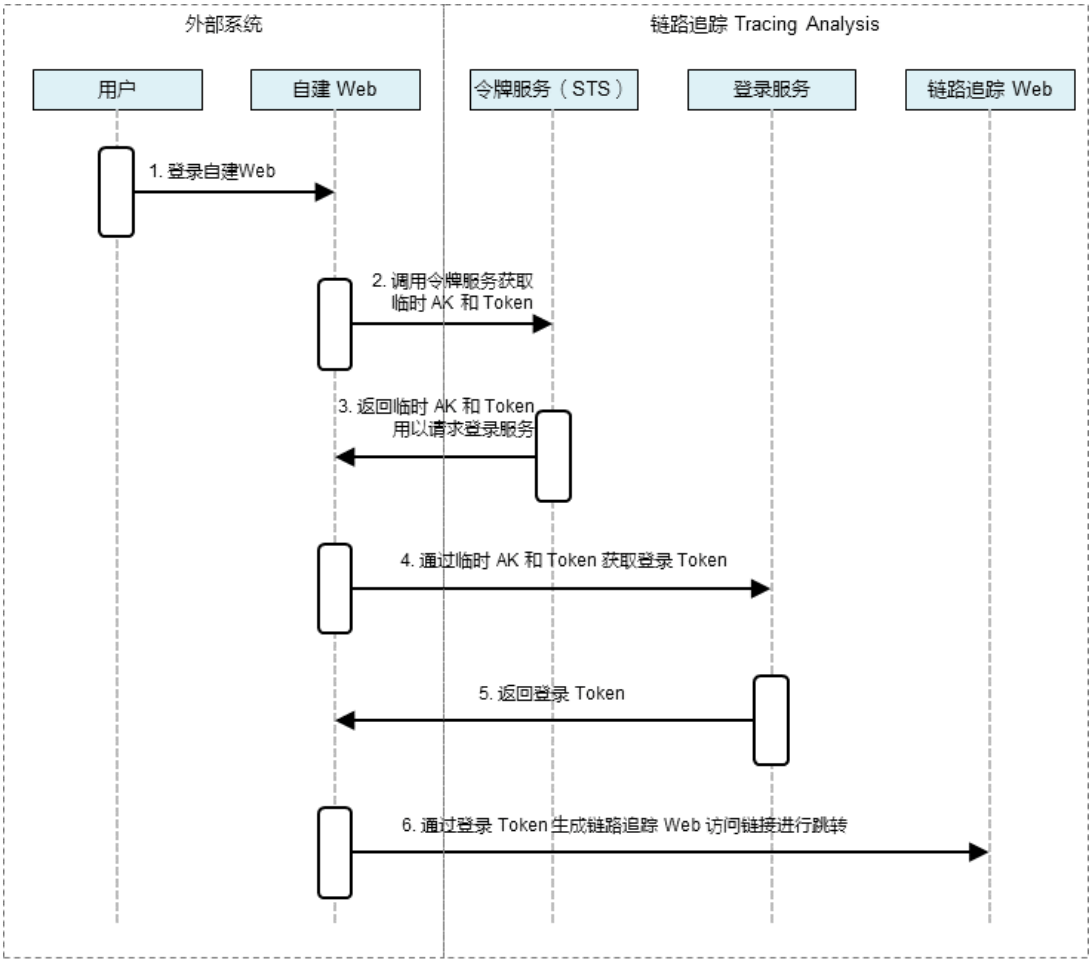
预期结果

参照本教程的方法进行实际操作后将实现以下效果：

- 可登录您的自有系统并浏览嵌入的应用列表、应用详情、调用查询等页面。
- 可隐藏链路追踪页面的顶部导航栏和左侧导航栏。
- 可通过访问控制RAM系统控制操作权限，例如将完整权限改为只读权限等。

访问流程

使用本教程所述方法访问链路追踪页面的流程如图所示。



准备工作：创建RAM用户并添加权限

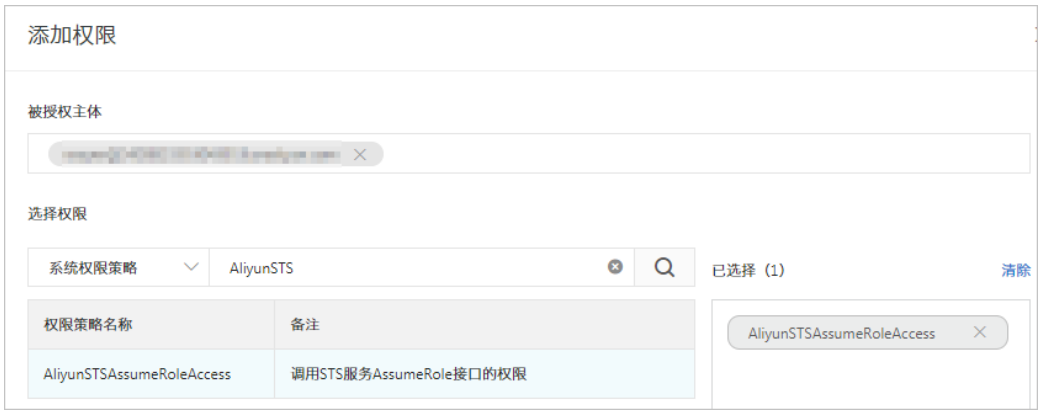
首先使用阿里云账号创建RAM用户并为其添加调用STS服务扮演RAM角色的权限。

1. 登录RAM控制台，在左侧导航栏中选择**身份管理 > 用户**，并在**用户**页面上单击**创建用户**。
2. 在**创建用户**页面的**用户账号信息**区域框中，输入**登录名称**和**显示名称**。在**访问方式**区域框中，勾

选Open API调用访问，并单击**确认**。

> 注意 RAM会自动为RAM用户创建AccessKey（API访问密钥）。出于安全考虑，RAM控制台只提供一次查看或下载AccessKeySecret的机会，即创建AccessKey时，因此请务必将AccessKeySecret记录到安全的地方。

3. 在**手机验证**对话框中单击**获取验证码**，并输入收到的手机验证码，然后单击**确定**。 创建的RAM用户显示在**用户**页面上，但此时没有任何权限。
4. 在**用户**页面上找到创建好的用户，单击**操作**列中的**添加权限**。
5. 在**添加权限**面板的**选择权限**区域框中，通过关键字搜索需要添加的权限策略AliyunSTSAssumeRoleAccess，并单击权限策略将其添加至右侧的**已选择**列表中，然后单击**确定**。



6. 在**添加权限**的**授权结果**页面上，查看授权信息摘要，并单击**完成**。

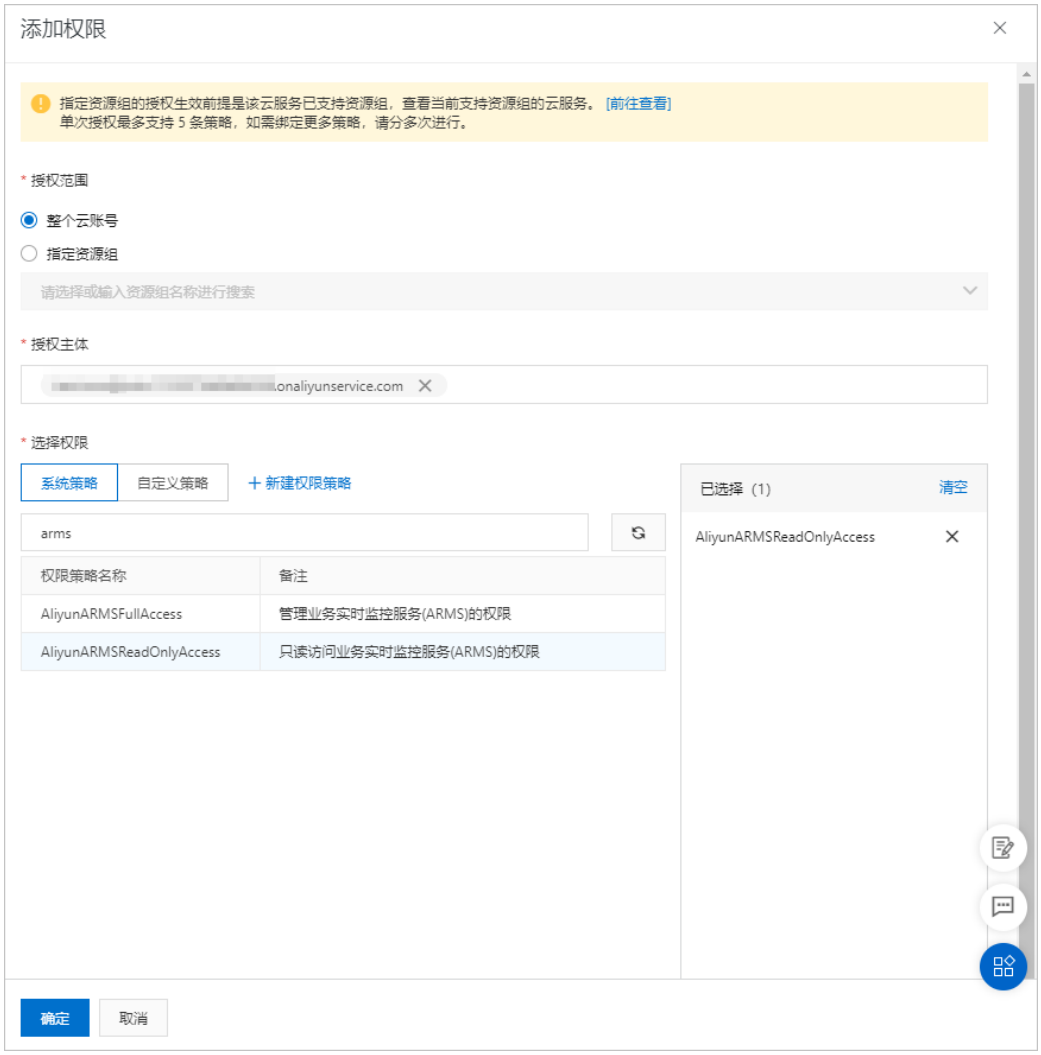
准备工作：创建RAM角色并添加权限

接下来创建RAM角色并为其添加访问链路追踪控制台的权限。上一步创建的RAM用户将会扮演该RAM角色访问链路追踪控制台。

1. 登录RAM控制台，在左侧导航栏中选择**身份管理 > 角色**，并在**角色**页面上单击**创建角色**。
2. 在**创建角色**面板中执行以下操作。
   i. 在**选择类型**页面的**选择可信实体类型**区域框中选择**阿里云账号**，并单击**下一步**。
   ii. 在**配置角色**页面的**角色名称**文本框内输入角色名称，并单击**完成**。

iii. 在**创建完成**页面上单击**为角色授权**。

3. 在**添加权限**面板的**选择权限**区域框中，通过关键字搜索需要添加的权限策略，并单击权限策略将其添加至右侧的**已选择**列表中，然后单击**确定**。



- 如需添加链路追踪的完整权限，则添加AliyunTracingAnalysisFullAccess。
- 如需添加链路追踪的只读权限，则添加AliyunTracingAnalysisReadOnlyAccess。

4. 在**添加权限**面板的**授权结果**页面上，查看授权信息摘要，并单击**完成**。

步骤一：获取临时AccessKey和Token

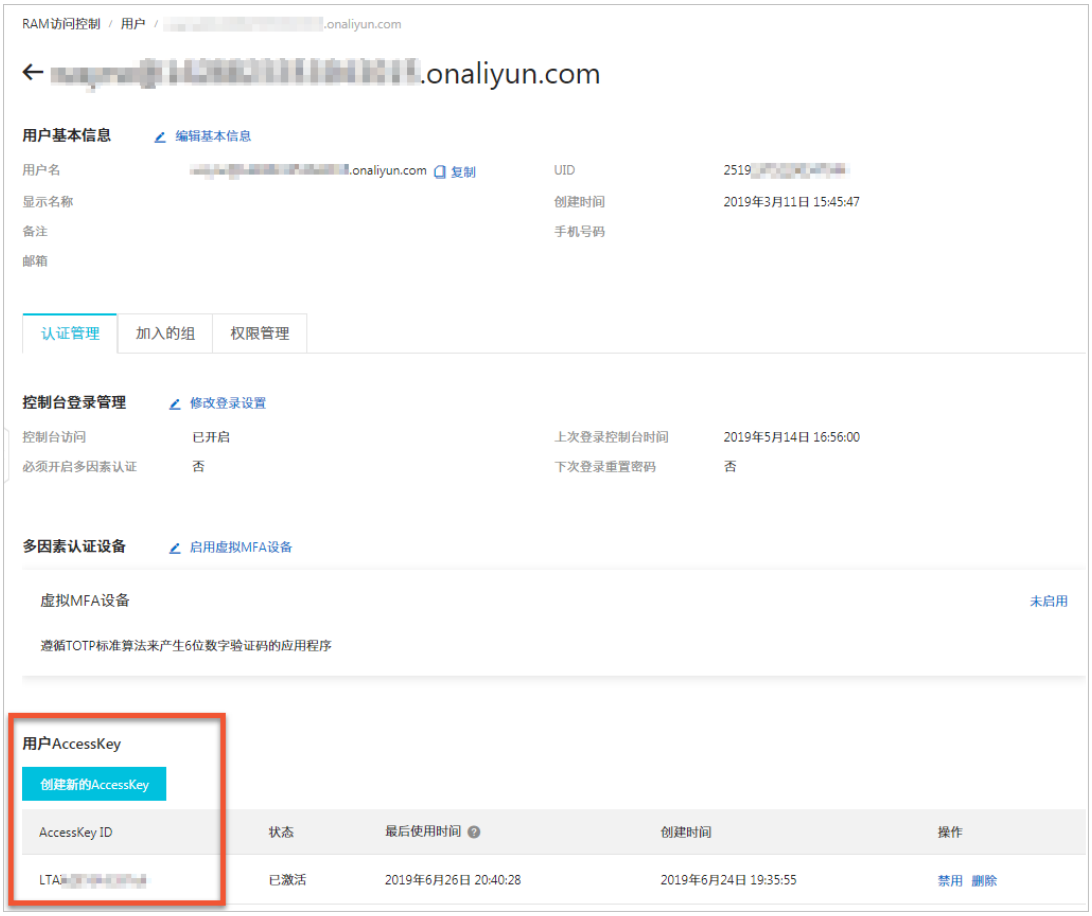
登录自建Web后，在Web服务端调用STS AssumeRole接口获取临时AccessKey和Token，即临时身份。请选择一种方式调用该接口：

- 通过OpenAPI在线调用。
- 通过Java示例调用。

请注意，在示例代码中，您首先需要将以下参数替换为真实的值。

```
String akId = "<accessKeyId>";
String ak = "<accessKeySecret>";
String roleArn = "<roleArn>";
```

其中，<accessKeyId>和<accessKeySecret>是准备工作中创建的用户的AccessKeyId和AccessKeySecret。



<roleArn>是准备工作中创建的RAM角色的标识ARN，可在RAM控制台的角色基本信息页面获取。

步骤二：获取登录Token

在通过STS AssumeRole接口获取临时AccessKey和Token后，调用登录服务接口获取登录Token。

> **注意** STS返回的安全Token中可能包含特殊字符，请对特殊字符进行URL编码后再输入。

请求样例：

```
http://signin4service.aliyun.com/federation?Action=GetSigninToken
    &AccessKeyId=<STS返回的临时AccessKeyId>
    &AccessKeySecret=<STS返回的临时AccessKeySecret>
    &SecurityToken=<STS返回的安全Token>
    &TicketType=mini
```

步骤三：生成免登录链接

利用获取到的登录Token与待嵌入的链路追踪控制台页面链接生成免登录访问链接，以最终实现在自建Web中免登录访问链路追踪控制台页面的目的。

> **说明** 由于Token有效期为3小时，建议在自建Web应用中将链接设置为每次请求时生成新登录Token，通过302请求返回进行跳转。

1. 在链路追踪控制台获取待嵌入的控制台页面链接。例如杭州地域的**应用列表**页面的链接为：

   ```
   https://tracing-analysis.console.aliyun.com/?hideTopbar=true&hideSidebar=true#/appList/cn-hangzhou
   ```

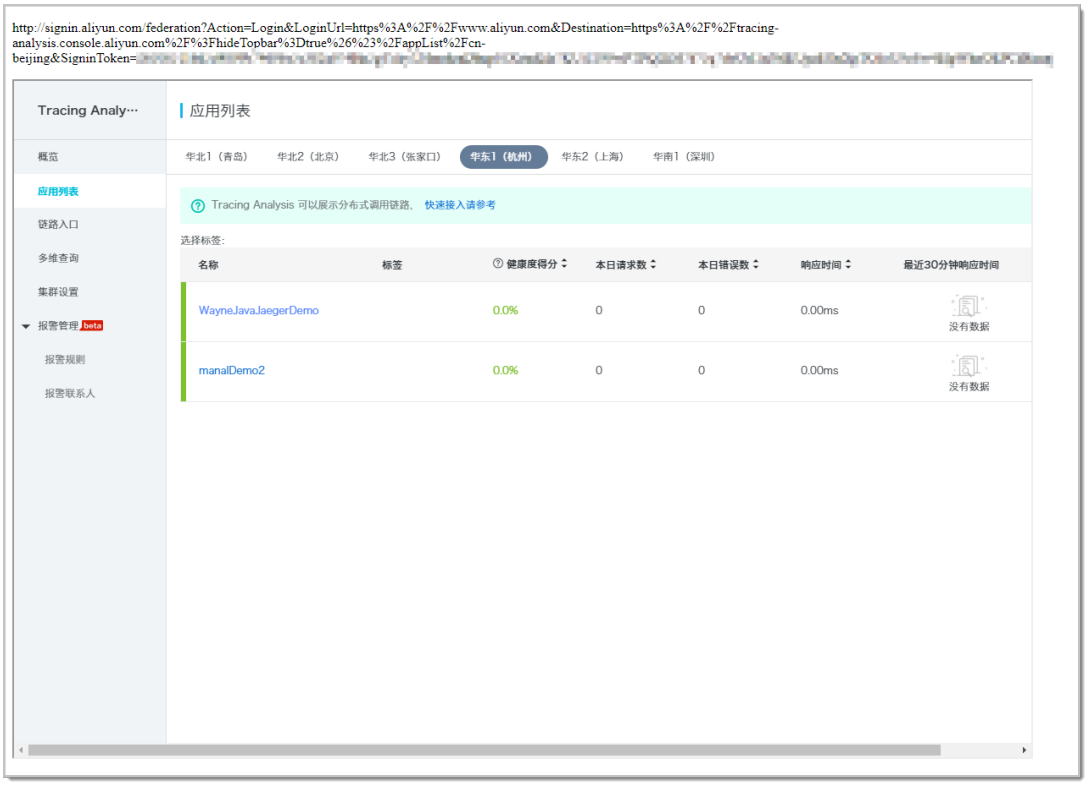
   > **说明** 将hideTopbar设为true表示隐藏顶部导航栏，将hideSidebar设为true表示隐藏左侧导航栏。

2. 利用步骤二中获取到的登录Token与链路追踪控制台页面链接生成免登录访问链接。 请求样例：

   ```
   http://signin.aliyun.com/federation?Action=Login
       &LoginUrl=<登录失效跳转的地址，一般配置为自建Web配置302跳转的URL>
       &Destination=<链路追踪控制台页面链接>
       &SigninToken=<获取到的登录Token>
   ```

执行结果

登录自建Web应用后，可见嵌入的链路追踪控制台页面效果如下图所示：



本教程的示例代码基于Java SDK，以嵌入链路追踪控制台的应用列表为例。
获取示例代码

9.API参考

9.1. API概览

链路追踪Tracing Analysis提供以下相关API接口。

API	描述
GetTagKey	调用GetTagKey获取上报链路数据中的Tag Key。
GetTagVal	调用GetTagVal获取上报的链路数据中指定Tag Key对应的Tag Value。
GetTrace	调用GetTrace获取调用链路详情。
ListIpOrHosts	调用ListIpOrHosts获取链路追踪数据中的IP地址或者机器名称，可获取整个地域或某个应用下的所有IP地址。
ListServices	调用ListServices获取某个地域下的所有应用列表。
ListSpanNames	调用ListSpanNames获取某个地域下所有的Span名称，也可获取某个微服务的所有Span名称。
QueryMetric	调用QueryMetric查询相关监控指标。
SearchTraces	调用SearchTraces查询调用链列表，可根据时间、应用名称、IP地址、Span名称和Tag等信息对调用链进行过滤查询。

9.2. 调用方式

链路追踪Tracing Analysis接口调用是向链路追踪Tracing Analysis API的服务端地址发送HTTP GET请求，并按照接口说明在请求中加入相应请求参数，调用后系统会返回处理结果。请求及返回结果都使用UTF-8字符集进行编码。

请求结构

链路追踪Tracing Analysis的API是RPC风格，您可以通过发送HTTP GET请求调用链路追踪Tracing Analysis API。

其请求结构如下：

```
http://Endpoint/?Action=xx&Parameters
```

其中：

- Endpoint： 链路追踪Tracing Analysis API的服务接入地址为*xtrace.[regionId].aliyuncs.com*。
- Action：要执行的操作，如调用GetTagKey获取上报链路数据中的Tag Key。
- Version：要使用的API版本，链路追踪Tracing Analysis的API版本是*2019-08-08*。
- Parameters：请求参数，每个参数之间用“&”分隔。
  请求参数由公共请求参数和API自定义参数组成。公共参数中包含API版本号、身份验证等信息，详情请参见公共参数。

下面是一个调用GetTagKey 接口获取上报链路数据中的Tag Key的示例：

> **说明** 为了便于用户查看，本文档中的示例都做了格式化处理。

```
https://xtrace.cn-hangzhou.aliyuncs.com/?Action=GetTagKey
&AccessKeyId=key-test
&Format=JSON
&RegionId=cn-hangzhou
&SecureTransport=true
&SignatureMethod=HMAC-SHA1
&SignatureVersion=1.0
&Timestamp=2020-03-18T06%3A06%3A30Z
&Version=2019-08-08
...
```

9.3. RAM鉴权

在使用RAM账号调用阿里云API前，需要主账号通过创建授权策略对RAM账号进行授权。

权限策略

链路追踪支持的系统权限策略如下所示。

权限策略	类型	说明
AliyunTracingAnalysisFullAccess	系统	链路追踪的完整权限
AliyunTracingAnalysisReadOnlyAccess	系统	链路追踪的只读权限

以上两种权限策略都可直接访问链路追踪的API接口。

更多信息，请参见借助RAM用户实现权限分割。

9.4. 签名机制

为保证API的安全调用，在调用API时阿里云会对每个API请求通过签名（Signature）进行身份验证。无论使用HTTP还是HTTPS协议提交请求，都需要在请求中包含签名信息。

概述

RPC API要按如下格式在API请求的Query中增加签名（Signature）：

```
https://Endpoint/?SignatureVersion=1.0&SignatureMethod=HMAC-SHA1&Signature=CT9X0VtwR86fNWSnsc6v8YGOjuE%3D&SignatureNonce=3ee8c1b8-83d3-44af-a94f-4e0ad82fd6cf
```

其中：

- SignatureMethod：签名方式，目前支持HMAC-SHA1。
- SignatureVersion：签名算法版本，目前版本是 1.0。
- SignatureNonce：唯一随机数，用于防止网络重放攻击。用户在不同请求间要使用不同的随机数值，建议使用通用唯一识别码（Universally Unique Identifier, UUID）。
- Signature: 使用AccessKey Secret对请求进行对称加密后生成的签名。

计算签名

签名算法遵循RFC 2104 HMAC-SHA1规范，使用AccessSecret对编码、排序后的整个请求串计算HMAC值作为签名。签名的元素是请求自身的一些参数，由于每个API请求内容不同，所以签名的结果也不尽相同。

```
Signature = Base64( HMAC-SHA1( AccessSecret, UTF-8-Encoding-Of(
StringToSign)) )
```

完成以下操作，计算签名：

1. 构建待签名字符串

i. 使用请求参数构造规范化的请求字符串（Canonicalized Query String）：

a. 按照参数名称的字典顺序对请求中所有的请求参数（包括公共请求参数和接口的自定义参数，但不包括公共请求参数中的Signature参数）进行排序。
当使用GET方法提交请求时，这些参数就是请求URI中的参数部分，即URI中“?”之后由“&”连接的部分。

b. 对排序之后的请求参数的名称和值分别用UTF-8字符集进行URL编码。编码规则如下：

字符	编码方式
A-Z、a-z和0-9以及“-”、“_”、“.”和“~”	不编码。
其它字符	编码成 `%XY` 的格式，其中 `XY` 是字符对应ASCII码的16进制表示。例如英文的双引号（”）对应的编码为 `%22` 。
扩展的UTF-8字符	编码成 `%XY%ZA...` 的格式。
英文空格	编码成 `%20` ，而不是加号（+）。 该编码方式和一般采用的 `application/x-www-form-urlencoded` MIME格式编码算法（例如 Java标准库中的 `java.net.URLEncoder` 的实现）存在区别。编码时可以先用标准库的方式进行编码，然后把编码后的字符串中的加号（+）替换成 `%20` ，星号（*）替换成 `%2A` ， `%7E` 替换回波浪号（~），即可得到上述规则描述的编码字符串。本算法可以用下面的 `percentEncode` 方法来实现： `private static final String ENCODING = "UTF-8"; private static String percentEncode(String value) throws UnsupportedEncodingException { return value !=null ? URLEncoder.encode(value, ENCODING).replace("+", "%20").replace("*", "%2A").replace("%7E", "~") : null; }`

c. 将编码后的参数名称和值用英文等号（=）进行连接。

d. 将等号连接得到的参数组合按步骤 i 排好的顺序依次使用“&”符号连接，即得到规范化请求字符串。

ii. 将第一步构造的规范化字符串按照下面的规则构造成待签名的字符串。

```
StringToSign=
      HTTPMethod + "&" +
      percentEncode("/") + "&" +
       percentEncode(CanonicalizedQueryString)
```

其中：

- HTTPMethod 是提交请求用的HTTP方法，例如GET。
- percentEncode(“/”) 是按照步骤1.1中描述的 URL 编码规则对字符 “/” 进行编码得到的值，即 %2F。
- percentEncode(CanonicalizedQueryString) 是对步骤1中构造的规范化请求字符串按步骤 1.2 中描述的URL编码规则编码后得到的字符串。

2. 计算签名

i. 按照RFC2104的定义，计算待签名字符串（StringToSign）的HMAC值。

> **说明** 计算签名时使用的Key就是您持有的AccessKey Secret并加上一个 “&” 字符（ASCII:38），使用的哈希算法是SHA1。

ii. 按照Base64编码规则把上面的HMAC值编码成字符串，即得到签名值（Signature）。

iii. 将得到的签名值作为Signature参数添加到请求参数中。

> **说明** 得到的签名值在作为最后的请求参数值提交时要和其它参数一样，按照RFC3986的规则进行URL编码。

示例

以DescribeRegionsAPI 为例，假设使用的 `AccessKey Id` 为 `testid` ， `AccessKey Secret` 为 `testsecret` 。签名前的请求URL如下：

```
http://ecs.aliyuncs.com/?Timestamp=2016-02-23T12:46:24Z&Format=XML&AccessKeyId=testid&Action=DescribeRegions&SignatureMethod=HMAC-SHA1&SignatureNonce=3ee8c1b8-83d3-44af-a94f-4e0ad82fd6cf&Version=2014-05-26&SignatureVersion=1.0
```

使用 `testsecret&` ，计算得到的签名值是：

```
OLeaidS1JvxuMvnyHOwuJ+uX5qY=
```

最后将签名作为Signature参数加入到URL请求中，最后得到的URL为：

```
http://ecs.aliyuncs.com/?SignatureVersion=1.0&Action=DescribeRegions&Format=XML&SignatureNonce=3ee8c1b8-83d3-44af-a94f-4e0ad82fd6cf&Version=2014-05-26&AccessKeyId=testid&Signature=OLeaidS1JvxuMvnyHOwuJ+uX5qY=&SignatureMethod=HMAC-SHA1&Timestamp=2016-02-23T12%3A46%3A24Z
```

9.5. 公共参数

介绍每个接口都需要使用的请求参数和返回参数。

公共请求参数

公共请求参数表

名称	类型	是否必须	描述
Format	String	否	返回消息的格式。取值： *JSON（默认值）| XML*
Version	String	是	API版本号，使用YYYY-MM-DD日期格式。取值： *2019-08-08*
AccessKeyId	String	是	访问服务使用的密钥ID。
Signature	String	是	签名结果串。
SignatureMethod	String	是	签名方式，取值： *HMAC-SHA1*
Timestamp	String	是	请求的时间戳，为日期格式。使用UTC时间按照 ISO8601标准，格式为YYYY-MM-DDThh:mm:ssZ。 例如，北京时间2013年1月10日20点0分0秒，表示为2013-01-10T12:00:00Z。
SignatureVersion	String	是	签名算法版本，取值： *1.0*
SignatureNonce	String	是	唯一随机数，用于防止网络重放攻击。 在不同请求间要使用不同的随机数值。
ResourceOwnerAccount	String	否	本次API请求访问到的资源拥有者账户，即登录用户名。

示例

```
https://xtrace.cn-hangzhou.aliyuncs.com/?Action=GetTagKey
&AccessKeyId=key-test
&Format=JSON
&RegionId=cn-hangzhou
&SecureTransport=true
&SignatureMethod=HMAC-SHA1
&SignatureVersion=1.0
&Timestamp=2020-03-18T06%3A06%3A30Z
&Version=2019-08-08
...
```

公共返回参数

API返回结果采用统一格式，调用成功返回的数据格式有XML和JSON两种，可以在发送请求时指定返回的数据格式，默认为XML格式。每次接口调用，无论成功与否，系统都会返回一个唯一识别码RequestId。

- 返回 `2xx` HTTP状态码表示调用成功。
- 返回 `4xx` 或 `5xx` HTTP状态码表示调用失败。

公共返回参数示例如下：

- XML格式

```
<?xml version="1.0" encoding="utf-8"?>
    <!--结果的根结点-->
    <接口名称+Response>
        <!--返回请求标签-->
        <RequestId>4C467B38-3910-447D-87BC-AC049166F216</RequestId>
        <!--返回结果数据-->
    </接口名称+Response>
```

- JSON格式

```
{
    "RequestId":"4C467B38-3910-447D-87BC-AC049166F216",
    /*返回结果数据*/
    }
```

9.6. 获取AccessKey

您可以为阿里云主账号和子账号创建一个访问密钥（AccessKey）。在调用阿里云API时您需要使用AccessKey完成身份验证。

背景信息

AccessKey包括AccessKey ID和AccessKey Secret。

- AccessKeyId：用于标识用户。
- AccessKeySecret：用于验证用户的密钥。AccessKeySecret必须保密。

> **警告** 主账号AccessKey泄露会威胁您所有资源的安全。建议使用子账号（RAM用户）AccessKey进行操作，可以有效降低AccessKey泄露的风险。

操作步骤

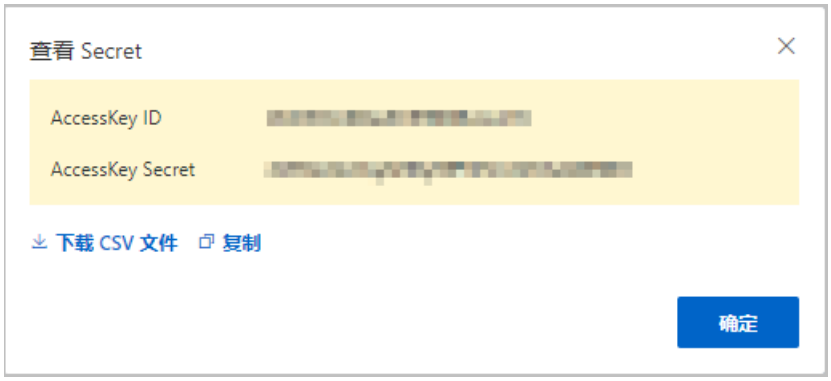
1. 以主账号登录阿里云管理控制台。
2. 将鼠标置于页面右上方的账号图标，单击**accesskeys**。
3. 在**安全提示**页面，选择获取主账号还是子账号的Accesskey。

4. 获取账号AccessKey。
   - 获取主账号AccessKey
     a. 单击**继续使用AccessKey**。

b. 在**安全信息管理**页面，单击**创建AccessKey**。

c. 在**手机验证**页面，获取验证码，完成手机验证，单击**确定**。

d. 在**新建用户AccessKey**页面，展开**AccessKey详情**，查看AccessKeyId和AccessKeySecret。可以单击**保存AK信息**，下载AccessKey信息。



- 获取子账号AccessKey

a. 单击**开始使用子用户AccessKey**。

b. 如果未创建RAM用户，在系统跳转的RAM访问控制台的**新建用户**页面，创建RAM用户。如果是获取已有RAM用户的AccessKey，则跳过此步骤。

c. 在RAM访问控制台的左侧导航栏，选择**人员管理 > 用户**，搜索需要获取AccessKey的用户。

d. 单击用户登录名称，在用户详情页**认证管理**页签下的**用户AccessKey**区域，单击**创建新的AccessKey**

e. 在**手机验证**页面，获取验证码，完成手机验证，单击**确定**。

f. 在**创建AccessKey**页面，查看AccessKeyId和AccessKeySecret。可以单击**下载CSV文件**，下载AccessKey信息或者单击**复制**，复制AccessKey信息。

9.7. ListSpanNames

调用ListSpanNames接口查询Span名称列表。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	ListSpanNames	系统规定参数，取值为ListSpanNames。
RegionId	String	是	cn-beijing	地域ID。
ServiceName	String	否	service 1	服务名称，又称为应用名称。
StartTime	Long	否	1575561600000	开始时间，精确到毫秒（ms）。
EndTime	Long	否	1575622455686	结束时间，精确到毫秒（ms）。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。
SpanNames	List	{"SpanName":["rpc0","rpc1.1","rpc1.1.1"]}	Span名称列表。

示例

请求示例

```
http(s)://[Endpoint]/?Action=ListSpanNames
&RegionId=cn-beijing
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<ListSpanNamesResponse>
    <SpanNames>
        <SpanName>rpc0</SpanName>
        <SpanName>rpc1.1</SpanName>
        <SpanName>rpc1.1.1</SpanName>
    </SpanNames>
    <RequestId>79C84C64-9951-477E-96F3-7FA44128C601</RequestId>
</ListSpanNamesResponse>
```

`JSON` 格式

```
{
  "SpanNames": {
    "SpanName": [
      "rpc0",
      "rpc1.1",
      "rpc1.1.1"]
      },
  "RequestId": "79C84C64-9951-477E-96F3-7FA44128C601"
}
```

错误码

访问错误中心查看更多错误码。

9.8. GetTagKey

调用GetTagKey接口获取标签键。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	GetTagKey	系统规定参数，取值为GetTagKey。
RegionId	String	是	cn-beijing	地域ID。
ServiceName	String	否	appTest	服务名称，又称为应用名称。
SpanName	String	否	createOrder	Span名称，又称为Operation名称。
StartTime	Long	否	1575561600000	开始时间，精确到毫秒（ms）。
EndTime	Long	否	1575622455686	结束时间，精确到毫秒（ms）。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。
TagKeys	List	{"TagKey":["date","resultCount","aTid"]}	标签键列表。

示例

请求示例

```
http(s)://[Endpoint]/?Action=GetTagKey
&RegionId=cn-beijing
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<GetTagKeyResponse>
    <TagKeys>
        <TagKey>date</TagKey>
        <TagKey>resultCount</TagKey>
        <TagKey>aTid</TagKey>
    </TagKeys>
    <RequestId>7D6519C1-A92A-4F07-AC83-706D48204242</RequestId>
</GetTagKeyResponse>
```

`JSON` 格式

```
{
 "TagKeys": {
  "TagKey": [
   "date",
   "resultCount",
   "aTid"
  ]
 },
 "RequestId": "7D6519C1-A92A-4F07-AC83-706D48204242"
}
```

错误码

访问错误中心查看更多错误码。

9.9. GetTagVal

调用GetTagVal接口获取标签键的标签值。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	GetTagVal	系统规定参数，取值为GetTagVal。
TagKey	String	是	span.kind	标签键。
RegionId	String	否	cn-beijing	地域ID。

名称	类型	是否必选	示例值	描述
ServiceName	String	否	appTest	服务名称，又称为应用名称。
SpanName	String	否	createOrder	Span名称，又称为Operation名称。
StartTime	Long	否	1575561600000	开始时间，精确到毫秒（ms）。
EndTime	Long	否	1575622455686	结束时间，精确到毫秒（ms）。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。
TagValues	List	{"TagValue": ["server"]}	标签值列表。

示例

请求示例

```
http(s)://[Endpoint]/?Action=GetTagVal
&TagKey=span.kind
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<GetTagValResponse>
    <RequestId>D36507D4-FD30-430B-BEC4-738661CFB86C</RequestId>
    <TagValues>
        <TagValue>server</TagValue>
    </TagValues>
</GetTagValResponse>
```

`JSON` 格式

```
{
 "RequestId": "D36507D4-FD30-430B-BEC4-738661CFB86C",
 "TagValues": {
  "TagValue": [
   "server"
  ]
 }
}
```

错误码

访问错误中心查看更多错误码。

9.10. GetTrace

调用GetTrace接口获取调用链路详情。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	GetTrace	系统规定参数。取值：**GetTrace**。
TraceID	String	是	1c6881aab84191a4	调用链ID，链路请求的唯一标识。可在控制台的**调用链分析**页面获取。
AppType	String	否	XTRACE	应用类型，取值为`XTRACE`或空。
RegionId	String	是	cn-beijing	地域ID。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。
Spans	Array of Span		Span列表。
Span			
SpanId	String	fec891bb8f8XXX	Span ID。
HaveStack	Boolean	false	是否有子Span。取值： • `true`：有子Span。 • `false`：无子Span。
ServiceIp	String	192.168.XXX.XXX	服务IP地址，即Span所在的机器IP地址。
OperationName	String	/api	又称为Span名称。
ParentSpanId	String	fec891bb8f8XXX	父Span ID。
ResultCode	String	200	返回码。
Duration	Long	1000	耗时，单位为毫秒（ms）。

名称	类型	示例值	描述
RpcId	String	1.1	表示Span之间的父子兄弟关系。 例如1.1是1.1.1的父亲Span， 而1.1.2和1.1.1是兄弟Span。
Timestamp	Long	1583683202047000	Span的产生时间戳。
ServiceName	String	server1	服务名称，又称为应用名称。
TraceID	String	1c6881aab84191a4	调用链ID，链路请求的唯一标识。
TagEntryList	Array of TagEntry		标签列表。
TagEntry			
Key	String	logLevel	Span的标签键。
Value	String	Warning	Span的标签值。
LogEventList	Array of LogEvent		日志事件列表。
LogEvent			
Timestamp	Long	1583683202047000	日志事件的产生时间戳。
TagEntryList	Array of TagEntry		标签列表。
TagEntry			
Key	String	logLevel	日志事件的标签键。
Value	String	Warning	日志事件的标签值。

示例

请求示例

```
http(s)://[Endpoint]/?Action=GetTrace
&RegionId=cn-beijing
&TraceID=1c6881aab84191a4
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
HTTP/1.1 200 OK
Content-Type:application/xml
<GetTraceResponse>
  <RequestId>1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED</RequestId>
  <Spans>
    <Span>
      <HaveStack>false</HaveStack>
      <ParentSpanId>fec891bb8f8XXX</ParentSpanId>
      <ServiceIp>192.168.XXX.XXX</ServiceIp>
      <ServiceName>server1</ServiceName>
      <OperationName>/api</OperationName>
      <RpcId>1.1</RpcId>
      <TraceID>1c6881aab84191a4</TraceID>
      <Duration>1000</Duration>
      <Timestamp>1583683202047000</Timestamp>
      <ResultCode>200</ResultCode>
      <SpanId>fec891bb8f8XXX</SpanId>
      <TagEntryList>
        <TagEntry>
          <Value>Warning</Value>
          <Key>logLevel</Key>
        </TagEntry>
      </TagEntryList>
      <LogEventList>
        <LogEvent>
          <Timestamp>1583683202047000</Timestamp>
          <TagEntryList>
            <TagEntry>
              <Value>Warning</Value>
              <Key>logLevel</Key>
            </TagEntry>
          </TagEntryList>
        </LogEvent>
      </LogEventList>
    </Span>
  </Spans>
</GetTraceResponse>
```

`JSON` 格式

```
HTTP/1.1 200 OK
Content-Type:application/json
{
  "RequestId" : "1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED",
  "Spans" : {
    "Span" : [ {
      "HaveStack" : "false",
      "ParentSpanId" : "fec891bb8f8XXX",
      "ServiceIp" : "192.168.XXX.XXX",
      "ServiceName" : "server1",
      "OperationName" : "/api",
      "RpcId" : "1.1",
      "TraceID" : "1c6881aab84191a4",
      "Duration" : "1000",
      "Timestamp" : "1583683202047000",
      "ResultCode" : "200",
      "SpanId" : "fec891bb8f8XXX",
      "TagEntryList" : {
        "TagEntry" : [ {
          "Value" : "Warning",
          "Key" : "logLevel"
        } ]
      },
      "LogEventList" : {
        "LogEvent" : [ {
          "Timestamp" : "1583683202047000",
          "TagEntryList" : {
            "TagEntry" : [ {
              "Value" : "Warning",
              "Key" : "logLevel"
            } ]
          }
        } ]
      }
    } ]
  }
}
```

错误码

访问错误中心查看更多错误码。

9.11. ListIpOrHosts

调用ListIpOrHosts获取应用的IP地址。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	ListIpOrHosts	系统规定参数。取值：**ListIpOrHosts**。
RegionId	String	是	cn-beijing	地域ID。
ServiceName	String	否	service1	应用名称。若为空则查询该地域下所有应用的IP地址。
StartTime	Long	否	1583683200000	开始时间的时间戳，精确到毫秒（ms）。
EndTime	Long	否	1583723776974	结束时间的时间戳，精确到毫秒（ms）。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC4****	请求ID。
IpNames	Array of String	["172.19.XXX.XXX", "10.0.X.X"]	IP地址列表。

示例

请求示例

```
http(s)://[Endpoint]/?Action=ListIpOrHosts
&RegionId=cn-beijing
&ServiceName=service1
&StartTime=1583683200000
&EndTime=1583723776974
&公共请求参数
```

正常返回示例

`XML` 格式

```
HTTP/1.1 200 OK
Content-Type:application/xml
<ListIpOrHostsResponse>
    <RequestId>23D9757C-CB58-41A5-BBE7-2A9A0652****</RequestId>
    <IpNames>
        <IpName>172.19.XXX.XXX</IpName>
        <IpName>10.0.X.X</IpName>
    </IpNames>
</ListIpOrHostsResponse>
```

`JSON` 格式

```
HTTP/1.1 200 OK
Content-Type:application/json
{
  "RequestId" : "23D9757C-CB58-41A5-BBE7-2A9A0652****",
  "IpNames" : {
    "IpName" : [ "172.19.XXX.XXX", "10.0.X.X" ]
  }
}
```

错误码

访问错误中心查看更多错误码。

9.12. ListServices

调用ListServices获取应用列表。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	ListServices	系统规定参数，取值为 `ListServices`。
RegionId	String	是	cn-beijing	地域ID。
AppType	String	否	XTRACE	应用类型，取值为 `XTRACE` 或空。

返回数据

名称	类型	示例值	描述
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。
Services	Array of Service		应用列表。
Service			
Pid	String	XXXqn3ly@741623b4e915df8	应用ID。
RegionId	String	cn-hangzhou	地域ID。
ServiceName	String	a3	应用名称。

示例

请求示例

```
http(s)://[Endpoint]/?Action=ListServices
&RegionId=cn-beijing
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<ListServicesResponse>
    <Services>
        <Service>
            <ServiceName>a3</ServiceName>
            <Pid>XXXqn3ly@741623b4e915df8</Pid>
            <RegionId>cn-hangzhou</RegionId>
        </Service>
    </Services>
</ListServicesResponse>
```

`JSON` 格式

```
{
  "Services": {
    "Service": [
      {
        "ServiceName": "a3",
        "Pid": "XXXqn3ly@741623b4e915df8",
        "RegionId": "cn-hangzhou"
      }
      ]
    }
}
```

错误码

访问错误中心查看更多错误码。

9.13. QueryMetric

调用QueryMetric查询相关监控指标。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述

名称	类型	是否必选	示例值	描述
Action	String	是	QueryMetric	系统规定参数，取值为 `QueryMetric` 。
EndTime	Long	是	1575622455686	结束时间，精确到毫秒（ms）。
IntervalInSec	Integer	是	60	时间间隔，单位为秒（s）。
Measures.N	RepeatList	是	count	查询字段。 **Metric**取值为 `appstat.incall` 时，**Measures.N**取值： • rt：响应时间。 • count：总数。 • error：错误数。 • exception：异常。 • errorrate：错误率。 • qps：每秒请求数。 **Metric**取值为 `appstat.sql` 时，**Measures.N**取值： • rt：响应时间。 • count：总数。 • error：错误数。 • slowcount：慢调用次数。
Metric	String	是	appstat.incall	指标名称，取值： • `appstat.incall` ：链路统计。 • `appstat.sql` ：SQL统计。
StartTime	Long	是	1575561600000	开始时间，精确到毫秒（ms）。
OrderBy	String	否	count	排序字段，根据查询返回字段中的某个字段排序。
Limit	Integer	否	100	返回数据条数。
Order	String	否	ASC	排序，取值： • `ASC` ：升序。 • `DESC` ：降序。
RegionId	String	否	cn-beijing	地域ID。
ProxyUserId	String	否	testefgag12	代理用户ID。

名称	类型	是否必选	示例值	描述
Filters.N.Key	String	否	http.status_code	过滤字段的Key。
Filters.N.Value	String	否	200	过滤字段的Value。
Dimensions.N	RepeatList	否	RT	维度，即GroupBy分组。

返回数据

名称	类型	示例值	描述
Data	String	{ "RequestId": "E2373982-D8CD-413D-B991-8EB678C01B4F", "Data": "{\"data\":[{\"date\":1583686800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan3\"}}	返回统计信息。
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。

示例

请求示例

```
http(s)://[Endpoint]/?Action=QueryMetric
&EndTime=1575622455686
&IntervalInSec=60
&Measures.1=count
&Metric=appstat.incall
&StartTime=1575561600000
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<QueryMetricResponse>
    <RequestId>E2373982-D8CD-413D-B991-8EB678C01B4F</RequestId>
    <Data>{"data":[{"date":1583686800000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583686800000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583686800000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583686800000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583690400000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583690400000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583690400000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583690400000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583694000000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583694000000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583694000000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583694000000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583697600000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583697600000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583697600000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583697600000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583701200000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583701200000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583701200000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583701200000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583704800000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583704800000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583704800000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583704800000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583708400000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583708400000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583708400000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583708400000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583712000000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583712000000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583712000000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583712000000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583715600000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583715600000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583715600000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583715600000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583719200000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583719200000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583719200000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583719200000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583722800000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583722800000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583722800000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583722800000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583726400000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583726400000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583726400000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583726400000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583730000000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583730000000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583730000000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583730000000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583733600000,"count":0,"rt":0,"rpc":"childSpan3"},{"date":1583733600000,"count":0,"rt":0,"rpc":"childSpan2"},{"date":1583733600000,"count":0,"rt":0,"rpc":"childSpan"},{"date":1583733600000,"count":0,"rt":0,"rpc":"parentSpan"},{"date":1583737200000,"count":100,"__time__":1583683200,"rt":0.01,"rpc":"childSpan3"},{"date":1583737200000,"count":100,"__time__":1583683200,"rt":305.18,"rpc":"childSpan2"},{"date":1583737200000,"count":100,"__time__":1583683200,"rt":305.25,"rpc":"childSpan"},{"date":1583737200000,"count":100,"__time__":1583683200,"rt":306.25,"rpc":"parentSpan"}],"actualSizeFromDB":0,"intervalInSec":3600,"resultSize":0,"successful":true}</Data>
</QueryMetricResponse>
```

`JSON` 格式

```
{
  "RequestId": "E2373982-D8CD-413D-B991-8EB678C01B4F",
  "Data": "{\"data\":[{\"date\":1583686800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan3\"},
{\"date\":1583686800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":158368680000
0,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583686800000,\"count\":0,\"rt\":0,
\"rpc\":\"parentSpan\"},{\"date\":1583690400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan3\"
},{\"date\":1583690400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":1583690400
000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583690400000,\"count\":0,\"rt\":
0,\"rpc\":\"parentSpan\"},{\"date\":1583694000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan3
\"},{\"date\":1583694000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":15836940
00000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583694000000,\"count\":0,\"rt\
":0,\"rpc\":\"parentSpan\"},{\"date\":1583697600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpa
n3\"},{\"date\":1583697600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":158369
7600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583697600000,\"count\":0,\"r
t\":0,\"rpc\":\"parentSpan\"},{\"date\":1583701200000,\"count\":0,\"rt\":0,\"rpc\":\"childS
pan3\"},{\"date\":1583701200000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":1583
701200000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583701200000,\"count\":0,\
"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583704800000,\"count\":0,\"rt\":0,\"rpc\":\"chil
dSpan3\"},{\"date\":1583704800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":15
83704800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583704800000,\"count\":0
,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583708400000,\"count\":0,\"rt\":0,\"rpc\":\"ch
ildSpan3\"},{\"date\":1583708400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\":
1583708400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583708400000,\"count\"
:0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583712000000,\"count\":0,\"rt\":0,\"rpc\":\"
childSpan3\"},{\"date\":1583712000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"date\
":1583712000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583712000000,\"count
\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583715600000,\"count\":0,\"rt\":0,\"rpc\":
\"childSpan3\"},{\"date\":1583715600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"dat
e\":1583715600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583715600000,\"cou
nt\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583719200000,\"count\":0,\"rt\":0,\"rpc\
":\"childSpan3\"},{\"date\":1583719200000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\"d
ate\":1583719200000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583719200000,\"c
ount\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583722800000,\"count\":0,\"rt\":0,\"rp
c\":\"childSpan3\"},{\"date\":1583722800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},{\
"date\":1583722800000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583722800000,\
"count\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583726400000,\"count\":0,\"rt\":0,\"
rpc\":\"childSpan3\"},{\"date\":1583726400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"},
{\"date\":1583726400000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":1583726400000
,\"count\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583730000000,\"count\":0,\"rt\":0,
\"rpc\":\"childSpan3\"},{\"date\":1583730000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2\"
},{\"date\":1583730000000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":15837300000
00,\"count\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583733600000,\"count\":0,\"rt\":
0,\"rpc\":\"childSpan3\"},{\"date\":1583733600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan2
\"},{\"date\":1583733600000,\"count\":0,\"rt\":0,\"rpc\":\"childSpan\"},{\"date\":158373360
0000,\"count\":0,\"rt\":0,\"rpc\":\"parentSpan\"},{\"date\":1583737200000,\"count\":100,\"_
_time__\":1583683200,\"rt\":0.01,\"rpc\":\"childSpan3\"},{\"date\":1583737200000,\"count\":
100,\"__time__\":1583683200,\"rt\":305.18,\"rpc\":\"childSpan2\"},{\"date\":1583737200000,\
"count\":100,\"__time__\":1583683200,\"rt\":305.25,\"rpc\":\"childSpan\"},{\"date\":1583737
200000,\"count\":100,\"__time__\":1583683200,\"rt\":306.25,\"rpc\":\"parentSpan\"}],\"actua
lSizeFromDB\":0,\"intervalInSec\":3600,\"resultSize\":0,\"successful\":true}"
}
```

错误码

访问错误中心查看更多错误码。

9.14. SearchTraces

调用SearchTraces查询调用链列表，可根据时间、应用名称、IP地址、Span名称、Tag等信息对调用链进行过滤查询。

调试

您可以在OpenAPI Explorer中直接运行该接口，免去您计算签名的困扰。运行成功后，OpenAPI Explorer可以自动生成SDK代码示例。

请求参数

名称	类型	是否必选	示例值	描述
Action	String	是	SearchTraces	系统规定参数，取值为 `SearchTraces`。
EndTime	Long	是	1575622455686	结束时间，精确到毫秒（ms）。
RegionId	String	是	cn-beijing	地域ID。
StartTime	Long	是	1575561600000	开始时间，精确到毫秒（ms）。
ServiceName	String	否	service 1	微服务名称，又称为应用名称。
OperationName	String	否	/api	Span名称，即某个跟踪点或埋点的名称。
MinDuration	Long	否	1000	大于某个时间跨度范围，单位为毫秒（ms）。例如，100表示大于100毫秒的数据。
AppType	String	否	XTRACE	应用类型，取值为 `XTRACE` 或空。
Tag.N.Key	String	否	http.status_cod	标签键。
Tag.N.Value	String	否	200	标签值。
PageNumber	Integer	否	1	页码，例如，5表示第5页。
PageSize	Integer	否	100	每页的查询数据条数。
Reverse	Boolean	否	false	按照时间正序或者倒序排列。默认为 `false`。取值： • `true`：倒序。 • `false`：顺序。
ServiceIp	String	否	10.0.0.0	Span对应的IP地址。

返回数据

名称	类型	示例值	描述
PageBean	Struct		返回数据的位置信息。
PageNumber	Integer	1	页码。
PageSize	Integer	100	每页显示的数据条数。
TotalCount	Long	1000	总条数。
TraceInfos	Array of TraceInfo		返回的调用链信息。
TraceInfo			
Duration	Long	100	耗时，单位为毫秒（ms）。
OperationName	String	/api	Span名称。
ServiceIp	String	192.163.XXX.XXX	Span所在的IP地址。
ServiceName	String	service1	微服务名称，又称为应用名称。
Timestamp	Long	1575561600000000	Span产生时间，单位为微秒（μs）。
TraceID	String	1c6881aab84191a4	调用链ID。
RequestId	String	1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED	请求ID。

示例

请求示例

```
http(s)://[Endpoint]/?Action=SearchTraces
&EndTime=1575622455686
&RegionId=cn-beijing
&StartTime=1575561600000
&<公共请求参数>
```

正常返回示例

`XML` 格式

```
<SearchTracesResponse>
    <PageBean>
        <TotalCount>1000</TotalCount>
        <PageSize>100</PageSize>
        <PageNumber>1</PageNumber>
        <TraceInfos>
            <TraceInfo>
                <ServiceIp>192.163.XXX.XXX</ServiceIp>
                <ServiceName>service1</ServiceName>
                <OperationName>/api</OperationName>
                <TraceID>1c6881aab84191a4</TraceID>
                <Duration>100</Duration>
                <Timestamp>1575561600000000</Timestamp>
            </TraceInfo>
        </TraceInfos>
    </PageBean>
    <RequestId>1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED</RequestId>
</SearchTracesResponse>
```

`JSON` 格式

```
{
    "PageBean": {
        "TotalCount": 1000,
        "PageSize": 100,
        "PageNumber": 1,
        "TraceInfos": {
            "TraceInfo": {
                "ServiceIp": "192.163.XXX.XXX",
                "ServiceName": "service1",
                "OperationName": "/api",
                "TraceID": "1c6881aab84191a4",
                "Duration": 100,
                "Timestamp": 1575561600000000
            }
        }
    },
    "RequestId": "1E2B6A4C-6B83-4062-8B6F-AEEC1FC47DED"
}
```

错误码

访问错误中心查看更多错误码。

10.访问控制

10.1. 访问控制概述

当您的企业有多用户协同操作资源的需求时，访问控制RAM可以让您避免与其他用户共享阿里云主账号密钥，并且按需为用户分配最小权限，从而降低您的企业信息安全风险。

应用场景

借助RAM用户实现分权的典型场景如下。

企业A的某个项目（Project-X）上云，购买了多种阿里云产品，例如：ECS实例、RDS实例、SLB实例、OSS存储空间等。项目里有多个员工需要操作这些云资源，由于每个员工的工作职责不同，需要的权限也不一样。企业A希望能够达到以下要求：

- 出于安全或信任的考虑，A不希望将云账号密钥直接透露给员工，而希望能给员工创建独立账号。
- 用户账号只能在授权的前提下操作资源。A随时可以撤销用户账号身上的权限，也可以随时删除其创建的用户账号。
- 不需要对用户账号进行独立的计量计费，所有开销都由A来承担。

针对以上需求，可以借助RAM的授权管理功能实现用户分权及资源统一管理。

权限策略

链路追踪支持的系统权限策略如下所示。

权限策略	类型	说明
AliyunTracingAnalysisFullAccess	系统	链路追踪的完整权限
AliyunTracingAnalysisReadOnlyAccess	系统	链路追踪的只读权限

更多信息，请参见借助RAM用户实现权限分割。

10.2. 链路追踪服务关联角色

本文介绍链路追踪服务关联角色AliyunServiceRoleForXtrace以及如何删除该角色。

背景信息

链路追踪服务关联角色AliyunServiceRoleForXtrace是链路追踪在某些情况下，为了完成自身的某个功能，需要获取其他云服务的访问权限而提供的RAM角色。更多关于服务关联角色的信息，请参见服务关联角色。

AliyunServiceRoleForXtrace应用场景

链路追踪监控功能需要访问容器服务ACK、日志服务SLS、云服务器ECS和专有网络VPC云服务的资源时，可通过自动创建的链路追踪服务关联角色AliyunServiceRoleForXtrace获取访问权限。

AliyunServiceRoleForXtrace权限说明

AliyunServiceRoleForXtrace具备以下云服务的访问权限。

容器服务ACK的访问权限 ❯

日志服务SLS的访问权限 >
云服务器ECS的访问权限 >
专有网络VPC的访问权限 >
SLB的访问和配置权限 >

删除AliyunServiceRoleForXtrace

如果您使用了链路追踪的监控功能，然后需要删除链路追踪服务关联角色AliyunServiceRoleForXtrace，例如您出于安全考虑，需要删除该角色，则需要先明确删除后的影响：删除AliyunServiceRoleForXtrace后，无法将当前账号下的数据进行存储和展示。

删除AliyunServiceRoleForXtrace的操作步骤如下：

> **说明** 如果当前账号下还有应用数据，则需先删除所有应用后才能删除AliyunServiceRoleForXtrace。

1. 登录RAM控制台，在左侧导航栏中选择**身份管理 > 角色**。
2. 在**角色**页面的搜索框中，输入AliyunServiceRoleForXtrace，自动搜索到名称为AliyunServiceRoleForXtrace的RAM角色。
3. 在右侧**操作**列，单击**删除**。
4. 在**删除RAM角色**对话框，单击**确定**。
   - 如果当前账号下还有链路追踪的应用，则需先删除所有应用才能删除AliyunServiceRoleForXtrace，否则提示删除失败。
   - 如果当前账号下所有应用已经删除，则可直接删除AliyunServiceRoleForXtrace。

常见问题

为什么我的XTRACE用户无法自动创建ARMS服务关联角色AliyunServiceRoleForXtrace?

您需要拥有指定的权限，才能自动创建或删除AliyunServiceRoleForXtrace。因此，在RAM用户无法自动创建AliyunServiceRoleForXtrace时，您需为其添加以下权限策略。

```
{
    "Statement": [
        {
            "Action": [
                "ram:CreateServiceLinkedRole"
            ],
            "Resource": "acs:ram:*:主账号ID:role/*",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "ram:ServiceName": [
                        "xtrace.aliyuncs.com"
                    ]
                }
            }
        }
    ],
    "Version": "1"
}
```

10.3. 借助RAM用户实现分权

借助访问控制RAM的RAM用户，您可以实现权限分割的目的，按需为RAM用户赋予不同权限，并避免因暴露阿里云账号密钥造成的安全风险。

背景信息

出于安全考虑，您可以为阿里云账号创建RAM用户，并根据需要为这些RAM用户赋予不同的权限，这样就能在不暴露阿里云账号密钥的情况下，实现让RAM用户各司其职的目的。在本文中，假设企业A希望让部分员工处理日常运维工作，则企业A可以创建RAM用户，并为RAM用户赋予相应权限，此后员工即可使用这些RAM用户登录控制台或调用API。

链路追踪支持的系统权限策略如下所示。

权限策略	类型	说明
AliyunTracingAnalysisFullAccess	系统	链路追踪的完整权限
AliyunTracingAnalysisReadOnlyAccess	系统	链路追踪的只读权限

步骤一：创建RAM用户

1. 使用阿里云账号登录RAM控制台。
2. 在左侧导航栏，选择**身份管理 > 用户**。
3. 在**用户**页面，单击**创建用户**。
4. 在**创建用户**页面的**用户账号信息**区域，输入**登录名称**和**显示名称**。

   > **说明** 单击**添加用户**，可一次性创建多个RAM用户。

5. 在**访问方式**区域，选择访问方式。
   - **控制台访问**：设置控制台登录密码、重置密码策略和多因素认证策略。

     > **说明** 自定义登录密码时，密码必须满足密码复杂度规则。关于如何设置密码复杂度规则，请参见设置RAM用户密码强度。

   - **OpenAPI调用访问**：自动为RAM用户生成访问密钥（AccessKey），支持通过API或其他开发工具访问阿里云。

     > **说明** 为了保障账号安全，建议仅为RAM用户选择一种登录方式，避免RAM用户离开组织后仍可以通过访问密钥访问阿里云资源。

6. 单击**确定**。

步骤二：为RAM用户添加权限

1. 使用阿里云账号登录RAM控制台。
2. 在左侧导航栏，选择**身份管理 > 用户**。
3. 在**用户**页面，单击目标RAM用户**操作**列的**添加权限**。
4. 在**添加权限**面板，为RAM用户添加权限。

i. 选择授权应用范围。
   - **整个云账号**：权限在当前阿里云账号内生效。
   - **指定资源组**：权限在指定的资源组内生效。

   > **说明** 指定资源组授权生效的前提是该云服务已支持资源组。更多信息，请参见支持资源组的云服务。

ii. 输入授权主体。

   授权主体即需要授权的RAM用户，系统会自动填入当前的RAM用户，您也可以添加其他RAM用户。

iii. 选择权限策略。

   > **说明** 每次最多绑定5条策略，如需绑定更多策略，请分次操作。

5. 单击**确定**。
6. 单击**完成**。

后续步骤

使用阿里云账号创建好RAM用户后，即可将RAM用户的登录名称及密码或者AccessKey信息分发给其他用户。其他用户可以按照以下步骤使用RAM用户登录控制台或调用API。

- 登录控制台
  i. 在浏览器中打开RAM用户登录入口。
  ii. 在**RAM用户登录**页面，输入RAM用户登录名称，单击**下一步**，并输入RAM用户密码，然后单击**登录**。

  > **说明** RAM用户登录名称的格式为*<$username>@<$AccountAlias>*或*<$username>@<$AccountAlias>.onaliyun.com*。*<$AccountAlias>*为账号别名，如果没有设置账号别名，则默认值为阿里云账号的ID。

  iii. 在**子用户用户中心**页面单击有权限的产品，即可访问控制台。
- 使用RAM用户的AccessKey调用API
  在代码中使用RAM用户的AccessKeyId和AccessKeySecret即可。

相关文档

- 访问控制概述

11.常见问题

11.1. 如何自定义检索数据？

链路追踪的数据存储在SLS里面，如果您需要自定义检索数据，可以根据数据格式写SQL进行数据检索。
链路追踪数据在SLS里面的Project名称格式为：

```
proj-xtrade-***-{regionId}
```

例如：

```
proj-xtrace-6dcbb77ef4ba6ef5466b5debf9e2f951-cn-beijing
```

链路详情（Span的格式）

对应的LogStore名称格式为：

```
logstore-xtrace-{userId}-{regionId}
```

例如：

```
logstore-xtrace-123456789-cn-beijing
```

字段名	字段说明
traceId	调用链ID，链路请求的唯一标识。例如：fec891bb8f81e7fb。
timestamp	Span的产生时间，单位为微秒。例如：1607410144095000。
rpc	Span名称或者是Operator名称。例如：/health。
serviceName	Span所在的微服务名称，又称为应用名称。例如：order。
serverIp	Span所在的机器IP地址。例如：127.0.0.1。
elapsed	耗时，单位是微秒。例如：1000表示1毫秒。
spanId	Span ID，链路中的唯一标识。例如：fec891bb8f81e7fc。
parentSpanId	父亲spanId，用来记录当前Span的父亲spanId。 例如：fec891bb8f81e7fb。
anno	记录Span的tag信息。例如：lb=prod&。

实际场景使用举例：

- 查找耗时大于1秒的请求。

  ```
  elapsed > 1000 | select distinct traceId
  ```

- 查找耗时大于1秒的请求，而且经过172.16.0.0这台机器。

```
* and serverIp = "172.16.0.0" and elapsed > 1000 | select distinct traceId
```

链路统计（Metric的格式）

对应的LogStore名称格式为：

```
logstore-xtrace-{userId}-stat-{regionId}
```

例如：

```
logstore-xtrace-123456789-stat-cn-beijing
```

字段名	字段说明
serviceName	Span所在的微服务名称，又称为应用名称。例如：order。
rpc	Span名称或者是Operator名称。例如：/health。
elapsed	统计时间内的总耗时，单位为毫秒。例如：值为10， 表示10毫秒。
avg_elapsed	统计时间内的平均耗时，单位为毫秒。例如：值为10， 表示10毫秒。
count	统计时间内的数据。例如：10。
timestamp	统计时间，单位为毫秒。例如：1607410144000。

实际场景使用举例：

- 查找Span名称为/api1的平均请求数。

```
rpc: "/api1" |select sum(count) qps timestamp GROUP by timestamp
```

- 查找Span名称为/api1的平均耗时。

```
rpc: "/api1" |select sum(elapsed) /sum(count) rt, timestamp GROUP by timestamp
```

11.2. 如何区分生产环境和测试环境？

链路追踪Tracing Analysis本身没有区分生产环境和测试环境，但是您可以利用地域或标签来达到区分环境的目的。

利用地域来区分环境

您可以选择当前地域或最近地域用于生产环境，选择一个较远的地域用于测试环境，以此达到区分环境的目的。例如，选择北京地域用于生产环境，选择青岛或者张家口地域用于测试环境。

- 优点：不同地域的资源是互相隔离的，这样做可以便利地隔离各个环境。
- 缺点：阿里云的部分VPC用户无法通过内网跨地域上报数据。如果是这种情况，建议利用标签来区分环境。

利用标签来区分环境

对于同一个地域内的应用，只要在接入点信息中的Token后添加标签（格式为 `_<label>` ），即可实现以下效果：

- 应用名称后自动加上一对括号，括号内为 `<label>` 。
- 链路追踪Tracing Analysis会自动为该应用创建一个 `<label>` 标签，该标签可用于筛选应用。

假设应用名称为 `demo` ，分别在Token后添加标签 `_prod` 和 `_test` ，则上报数据成功后可实现以下效果：

> **说明** 在应用管理页面可以修改或删除标签，具体操作，请参见管理应用和标签。

Token是指链路追踪Tracing Analysis控制台概览页所显示的接入点信息中的红色字符。

在Token后添加标签 `test` 的示例如下：

- 使用Jaeger客户端上报数据时
  - 添加标签前

    ```
    http://tracing-analysis-dc-hz.aliyuncs.com/adapt_abcefg123@abcefg123_abcefg456@abcefg456/api/traces
    ```

  - 添加标签后

    ```
    http://tracing-analysis-dc-hz.aliyuncs.com/adapt_abcefg123@abcefg123_abcefg456@abcefg456_test/api/traces
    ```

- 使用Zipkin客户端上报数据时

- 添加标签前

```
http://tracing-analysis-dc-hz.aliyuncs.com/adapt_abcefg123@abcefg123_abcefg456@abcefg456/api/v2/spans
```

- 添加标签后

```
http://tracing-analysis-dc-hz.aliyuncs.com/adapt_abcefg123@abcefg123_abcefg456@abcefg456_test/api/v2/spans
```

- 使用SkyWalking客户端上报数据时
  - 添加标签前的agent.authentication字段

```
agent.authentication=abcefg123@abcefg123_abcefg456@abcefg456
```

  - 添加标签后的agent.authentication字段

```
agent.authentication=abcefg123@abcefg123_abcefg456@abcefg456_test
```

相关文档

- 准备工作概述
- 查看应用列表
- 管理应用和标签

12.相关协议

12.1. 链路追踪服务协议

版本生效日期：2019年5月17日。
欢迎您与阿里云计算有限公司（以下简称“阿里云”）共同签署本《链路追踪服务协议》（下称“本协议”）并使用阿里云服务！

协议中条款前所列索引关键词仅为帮助您理解该条款表达的主旨之用，不影响或限制本协议条款的含义或解释。为维护您自身权益，建议您仔细阅读各条款具体表述。

【审慎阅读】您在同意本协议之前，应当认真阅读本协议。请您务必审慎阅读、充分理解各条款的内容，特别是免除或者限制责任的条款、法律适用和争议解决条款，这些条款将以粗体下划线标识，您应重点阅读。如您对协议有任何疑问，可以向客服和相关业务部门进行咨询。

【签约动作】当您阅读并点击同意本协议或以其他方式选择接受本协议后，即表示您已充分阅读、理解并接受本协议的全部内容，并与阿里云达成一致。本协议自您通过网络页面点击确认或以其他方式选择接受本协议之日起成立。阅读本协议的过程中，如果您不同意本协议或其中任何条款约定，请勿进行签约动作。

通用服务条款

1. 签约主体及协议范围

本服务协议是**您与阿里云计算有限公司**就您使用**阿里云链路追踪**所签署的服务协议。

2. 服务内容

本条款中“服务”指：阿里云www.aliyun.com网站和客户端（以下单独或统称“阿里云网站”）所展示的链路追踪服务以及相关的技术及网络支持服务。

3. 服务费用

3.1.服务费用将在您订购页面予以列明公示，您可自行选择具体服务类型并按列明的价格予以支付。您可选择先付费或后付费的服务。

3.2. 先付费：

3.2.1.在您付费之后，阿里云才开始为您提供服务。您未在下单后立即付费的，订单将为您保留7天，7天内您仍未付费或者7天内阿里云服务售罄的，订单失效，订单失效后阿里云与您就服务所达成的合意失效。

3.2.2.服务期满双方愿意继续合作的，您至少应在服务期满前7天内支付续费款项，以使服务得以继续进行。

3.3. 后付费：您先使用后付费。具体扣费规则请查看www.aliyun.com上的页面展示且以页面公布的后付费服务当时有效的计费模式、标准为准。

3.4. 阿里云保留在您未按照约定支付全部费用之前不向您提供服务和/或技术支持，或者终止服务和/或技术支持的权利。同时，阿里云保留对您的欠费要求您按日承担万分之五的违约金以及追究其他法律责任的权利。

3.5.您完全理解阿里云价格体系中所有的赠送服务项目或优惠活动均为阿里云在正常服务价格之外的一次性特别优惠，赠送的服务项目或优惠活动不可折价、冲抵服务价格。

4. 您的权利和义务

4.1.成功订购服务后，您有权要求阿里云按照本服务协议以及阿里云网站相关页面所展示的服务说明、技术规范等内容向您提供服务。

4.2.您订购阿里云的服务后，您可享受免费的售后服务。除此之外阿里云并提供其他付费的技术服务。

4.3. 您应按照阿里云的页面提示及本服务协议的约定支付相应服务费用。

4.4. 就阿里云服务的使用应符合附件阿里云的《服务使用规则》以及本服务协议。

4.5.您对自己存放在阿里云云平台上的数据以及进入和管理阿里云云平台上各类产品与服务的口令、密码的完整性和保密性负责。因您维护不当或保密不当致使上述数据、口令、密码等丢失或泄漏所引起的损失和后果均由您自行承担。

4.6.您须依照《互联网信息服务管理办法》、《互联网电子公告服务管理规定》等法律法规的规定保留自己网站的访问日志记录，包括发布的信息内容及其发布时间、互联网地址（IP）、域名等，国家有关机关依法查询时应配合提供。您自行承担未按规定保留相关记录而引起的法律责任。

4.7.为了数据的安全，您应负责您数据的备份工作。阿里云的产品或服务可能会为您配置数据备份的功能或工具，您负责操作以完成备份。

4.8.您应对您的用户业务数据的来源及内容负责，阿里云提示您谨慎判断数据来源及内容的合法性。因您的用户业务数据内容违反法律法规、部门规章或国家政策而造成的相应结果均由您承担。

4.9.您理解并同意，中华人民共和国的国家秘密受法律保护，您有保守中华人民共和国的国家秘密的义务；您使用阿里云服务应遵守相关保密法律法规的要求，并不得危害中华人民共和国国家秘密的安全。

4.10.您还应仔细阅读并遵守阿里云在网站页面上展示的相应服务说明、技术规范、使用流程、操作文档等内容（以上简称“操作指引”），依照相关操作指引进行操作。您将自行承担违反相关操作指引所引起的后果；同时，阿里云郑重提示您，请把握风险谨慎操作。

5. 阿里云的权利、义务

5.1.阿里云应按照约定提供服务。

5.2.服务期限内，阿里云将为您提供如下售后服务：

5.2.1.阿里云将提供7×24电话咨询服务以及在线工单咨询服务，解答您在使用中的问题；

5.2.2.阿里云将为您提供故障支持服务，您应通过在线工单申报故障；阿里云将及时就您非人为操作所出现的故障提供支持，但因您的人为原因和/或不可抗力、以及其他非阿里云控制范围内的事项除外。

您还可通过阿里云获得其他付费的售后服务，具体详见阿里云的网站相关页面的收费售后服务内容。

5.3.阿里云仅负责操作系统以下的底层部分及阿里云提供的软件的运营维护，即服务的相关技术架构及阿里云提供的操作系统等。操作系统之上部分（如您在系统上安装的应用程序）由您自行负责。此外，您自行升级操作系统可能会造成宕机等不良影响，请自行把握风险并谨慎操作。

5.4.您了解阿里云无法保证其所提供的服务毫无瑕疵（如阿里云安全产品并不能保证您的硬件或软件的绝对安全），但阿里云承诺不断提升服务质量及服务水平。所以您同意：即使阿里云提供的服务存在瑕疵，但上述瑕疵是当时行业技术水平所无法避免的，其将不被视为阿里云违约。您同意和阿里云一同合作解决上述瑕疵问题。

5.5.阿里云的某些服务可能具备账户授权管理功能，即您可将您对服务的全部或部分操作权限授权给您指定的一个或多个被授权账户，此种情况下，任一被授权账户下进行的所有操作行为，均将被视为您通过本人账户所进行的行为，都将由您承担相应结果，并承担相应的服务费用。

5.6.您理解并认可，阿里云将为您提供基于某些服务的安全防护（如“云盾安骑士服务”）以及管理与监控的相关功能及服务（如“云监控”），尽管阿里云对该等服务经过详细的测试，但并不能保证其与所有的软硬件系统完全兼容，亦不能保证其软件及服务的完全准确性。如果出现不兼容及软件错误的情况，您应立即关闭或停止使用相关功能，并及时联系阿里云，获得技术支持。

6. 用户业务数据

6.1.阿里云理解并认可，您通过阿里云提供的服务，加工、存储、上传、下载、分发以及通过其他方式处理的数据，均为您的用户业务数据，您完全拥有您的用户业务数据。

6.2.就用户业务数据，阿里云除执行您的服务要求外，不进行任何未获授权的使用及披露；但以下情形除外：

6.2.1.在国家有关机关依法查询或调阅用户业务数据时，阿里云具有按照相关法律法规或政策文件要求提供配合，并向第三方或者行政、司法等机构披露的义务；

6.2.2.您和阿里云另行协商一致。

6.3. 您可自行对您的用户业务数据进行删除、更改等操作。如您自行释放服务或删除数据的，阿里云将删除您的数据，按照您的指令不再保留该等数据。就数据的删除、更改等操作，您应谨慎操作。

6.4. 当服务期届满、服务提前终止（包括双方协商一致提前终止，其他原因导致的提前终止等）或您发生欠费时，除法律法规明确约定、主管部门要求或双方另有约定外，阿里云仅在一定的缓冲期（以您所订购的服务适用的产品文档、服务说明等所载明的时限为准）内继续存储您的用户业务数据（如有），缓冲期届满阿里云将删除所有用户业务数据，包括所有缓存或者备份的副本，不再保留您的任何用户业务数据。

6.5. 用户业务数据一经删除，即不可恢复；您应自行承担数据因此被删除所引发的后果和责任，您理解并同意，阿里云没有继续保留、导出或者返还用户业务数据的义务。

6.6. 根据您与阿里云协商一致，阿里云在您选定的数据中心存储用户业务数据。阿里云恪守对用户的安全承诺，根据适用的法律保护用户存储在阿里云数据中心的数据。

7. 知识产权

7.1.在本协议项下一方向对方提供的任何资料、技术或技术支持、软件、服务等的知识产权均属于提供一方或其合法权利人所有； 除提供方或合法权利人明示同意外，另一方无权复制、传播、转让、许可或提供他人使用上述知识成果，否则应承担相应的责任。

7.2.您应保证提交阿里云的素材、对阿里云服务的使用及使用阿里云服务所产生的成果未侵犯任何第三方的合法权益。阿里云应保证向您提供的服务未侵犯任何第三方的合法权益。

7.3.如果第三方机构或个人对您使用阿里云服务所涉及的相关素材的知识产权归属提出质疑或投诉，或对您使用的阿里云的服务的知识产权的归属提出质疑或投诉，您和阿里云均有责任出具相关知识产权证明材料，并配合对方相关投诉处理工作。对于因此引起的索赔、诉讼或可能向其提起诉讼，违约方应负责解决，承担费用和损失，以及使另一方完全免责。

8. 保密条款

8.1.本服务条款所称保密信息，是指一方（以下简称“接受方”）从对方（以下简称“披露方”）取得的、获知的、或因双方履行本协议而产生的商业秘密（包括财务秘密）、技术秘密、经营诀窍和（或）其他应予保密的信息和资料（包括产品资料、产品计划、价格、财务及营销规划、业务战略、客户信息、客户数据、研发、软件、硬件、API应用数据接口、技术说明、设计、特殊公式、特殊算法等），无论上述信息和资料以何种形式或载于何种载体，无论披露方在披露时是否以口头、图像或书面等方式表明其具有保密性。

8.2.双方应采取适当措施妥善保存对方提供的保密信息，措施的审慎程度不少于其保护自身的保密信息时的审慎程度。双方仅能将保密信息用于与本协议项下的有关用途或目的。

8.3.双方保证保密信息仅可在各自一方从事该业务的负责人和雇员范围内知悉，并严格限制接触上述保密信息的员工遵守本条之保密义务。

8.4.本条上述限制条款不适用于以下情况：

8.4.1.在签署本协议之时或之前，该保密信息已以合法方式属接受方所有；

8.4.2.保密信息在通知给接受方时，已经公开或能从公开领域获得；

8.4.3.保密信息是接受方从与其没有保密或不透露义务的第三方获得的；

8.4.4.在不违反本协议约定责任的前提下，该保密信息已经公开或能从公开领域获得；

8.4.5.该保密信息是接受方或其关联或附属公司独立开发，而且未从通知方或其关联或附属公司获得的信息中获益；

8.4.6.接受方应法院或其它法律、行政管理部门要求（通过口头提问、询问、要求资料或文件、传唤、民事或刑事调查或其他程序）因而透露保密信息；

8.4.7.接受方为向行政管理部门、行业协会等机构申请某项业务资质、获得某项认定、或符合国家、行业标准/认证，需结合对方情况向前述机构提交材料或进行说明的而披露的信息，在该等情况下，接受方应秉持必要情况下最少披露原则及要求因此获知保密信息的机构按不低于本协议的标准予以保密。

8.5. 您和阿里云都应尽最大的努力保护上述保密信息不被披露。一旦发现有上述保密信息泄露事件，双方应合作采取相应的合理措施避免或者减轻损害后果的产生。如因此给对方造成损失的，应赔偿因此给对方造成的直接经济损失。

9. 服务的开通、终止与变更

9.1. 先付费的服务：

9.1.1.您付费后服务即开通，开通后您获得阿里云向您发送的登录、使用服务的密钥、口令即可使用服务，服务期限自开通之时起算（而非自您获得登录、使用服务的密钥、口令时起算）；

9.1.2.以包年包月形式售卖的服务，服务期限至订购的期限届满为止；以资源包（或套餐包）形式售卖的服务，服务期限则至您订购的资源包服务期限到期或资源包中的服务被使用完毕为止（以前述二者早发生为准）；

9.1.3. 您应在服务期限内将资源包的服务数量使用完毕，如资源包的服务期限届满，您已订购但未使用完毕的服务将被作废且阿里云将不提供其他替代或补充。

9.1.4. 您对于服务的使用将优先消耗订购的资源包，除法定及双方另行约定外，如资源包中的各项服务使用完毕或者服务期限到期，且您未继续订购资源包服务但持续使用此项服务的，阿里云将视为您使用阿里云以后付费形式售卖的该服务（如有），阿里云将持续计费并根据计费结果予以扣划服务费用。

9.2. 后付费的服务：

除非另有其他约定或您未结清其他应付款项的，您开通服务即可使用阿里云的服务。您应确保您的账户余额充足，以便持续使用服务至法律规定或本服务条款约定的终止情形出现时为止。

9.3.发生下列情形之一的，服务期限提前终止：

9.3.1.双方协商一致提前终止的；

9.3.2.您严重违反本协议（包括您严重违反相关法律法规规定，或您严重违反本协议项下之任一承诺内容等），阿里云有权提前终止服务**直至清除您的全部数据**；

9.3.3.您理解并充分认可，虽然阿里云已经建立（并将根据技术的发展不断完善）必要的技术措施来防御包括计算机病毒、网络入侵和攻击破坏（如DDoS等）危害网络安全事项或行为（以下统称该等行为），**但鉴于网络安全技术的局限性、相对性以及该等行为的不可预见性，因此如因您网站遭遇该等行为而给阿里云或者阿里云的其他网络或服务器（包括本地及外地和国际的网络、服务器等）带来危害，或影响阿里云与国际互联网或者阿里云与特定网络、服务器及阿里云内部的通畅联系，阿里云可决定暂停或终止服务。如果终止服务的，将按照实际提供服务月份计算（不足一个月的按天计）服务费用，将剩余款项（如有）返还；**

9.3.4. 阿里云可提前30天在www.aliyun.com上通告或给您发网站内通知或书面通知的方式终止本服务服务协议；届时阿里云应将您已支付但未消费的款项退还至您的阿里云账户；

9.4. 您理解并认可，为技术升级、服务体系升级、或因经营策略调整或配合国家重大技术、法规政策等变化，阿里云不保证永久的提供某种服务，并有权变更所提供服务的形式、规格或其他方面（如服务的价格和计费模式），在终止该种服务或进行此种变更前，阿里云将尽最大努力且提前以网站公告、站内信、邮件或短信等一种或多种方式进行事先通知。

10. 违约责任

10.1. 您违反本协议中的承诺、保证条款、服务使用规则或义务的任一内容，或阿里云根据其判断认为您的使用行为存在异常的，阿里云均有权就其情节，判断并采取以下措施中的一种或多种：（1）限制、中止使用服务；（2）终止提供服务，终止本协议；（3）追究您的法律责任；（4）其他阿里云认为适合的处理措施。阿里云依据前述约定采取中止服务、终止服务等措施而造成的用户损失由您承担。

10.2. 如因您违反有关法律法规或者本协议、相关规则之规定，使阿里云遭受任何损失、受到其他用户、任何第三方的索赔或任何行政管理部门的处罚，您应对阿里云、其他用户或相关第三方的实际损失进行全额赔偿，包括合理的律师费用。

10.3. 您理解且同意，鉴于计算机、互联网的特殊性，下述情况不属于阿里云违约：

10.3.1. 阿里云在进行系统及服务器配置、维护、升级时，需要短时间中断服务；

10.3.2. 由于Internet上的通路阻塞造成您网站访问速度下降。

10.4.如果因阿里云原因造成您连续72小时不能正常使用服务的，您可终止接受服务，但非阿里云控制之内的原因引起的除外。

10.5.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、特殊性的损害，包括您使用阿里云服务而遭受的利润损失承担责任（即使您已被告知该等损失的可能性）。

10.6. 在法律允许的情况下，阿里云在本协议项下所承担的损失赔偿责任不超过就该服务过往12个月所缴纳的服务费用的总和。

11. 通知

11.1.您在使用阿里云服务时，您应该向阿里云提供真实有效的联系方式（包括您的电子邮件地址、联系电话、联系地址等），对于联系方式发生变更的，您有义务及时更新有关信息，并保持可被联系的状态。您接收站内信、系统消息的会员账号（包括子账号），也作为您的有效联系方式。

11.2阿里云将向您的上述联系方式的其中之一或其中若干向您送达各类通知，而此类通知的内容可能对您的权利义务产生重大的有利或不利影响，请您务必及时关注。

11.3.阿里云通过上述联系方式向您发出通知，其中以电子的方式发出的书面通知，包括但不限于公告，向您提供的联系电话发送手机短信，向您提供的电子邮件地址发送电子邮件，向您的账号发送系统消息以及站内信信息，在发送成功后即视为送达；以纸质载体发出的书面通知，按照提供联系地址交邮后的第五个自然日即视为送达。

11.4.您应当保证所提供的联系方式是准确、有效的，并进行实时更新。如果因提供的联系方式不确切，或不及时告知变更后的联系方式，使法律文书无法送达或未及时送达，由您自行承担由此可能产生的法律后果。

12. 不可抗力

12.1.因不可抗力或者其他意外事件，使得本服务条款的履行不可能、不必要或者无意义的，遭受不可抗力、意外事件的一方不承担责任。

12.2.不可抗力、意外事件是指不能预见、不能克服并不能避免且对一方或双方当事人造成重大影响的客观事件，包括但不限于自然灾害如洪水、地震、瘟疫流行等以及社会事件如战争、动乱、政府行为、电信主干线路中断、黑客、网路堵塞、电信部门技术调整和政府管制等。

13. 法律适用及争议解决

13.1. 本协议之订立、生效、解释、修订、补充、终止、执行与争议解决均适用中华人民共和国大陆法律；如法律无相关规定的，参照商业惯例及/或行业惯例。

13.2. 您因使用阿里云服务所产生及与阿里云服务有关的争议，由阿里云与您协商解决。协商不成时，任何一方均可向被告所在地有管辖权的人民法院提起诉讼。

14. 附则

14.1.本协议的附件，以及阿里云在阿里云网站相关页面上的服务说明、价格说明和您确认同意的订购页面（包括特定产品的专用条款、服务说明、操作文档等）均为本协议不可分割的一部分。如遇不一致之处，以（1）服务说明、价格说明、其他订购页面，（2）专用条款和4.3. 附件，（3）本协议通用条款的顺序予以适用。

14.2. 如本协议内容发生变动，阿里云应通过提前30天在阿里云网站的适当版面公告向您提示修改内容；如您继续使用阿里云服务，则视为您接受阿里云所做的相关修改。

14.3. 阿里云有权经提前将本协议的权利义务全部或者部分转移给阿里云的关联公司。

14.4.阿里云于您过失或违约时放弃本协议规定的权利，不应视为其对您的其他或以后同类之过失或违约行为弃权。

14.5.本协议任一条款被视为废止、无效或不可执行，该条应视为可分的且并不影响本协议其余条款的有效性及可执行性。

14.6.本协议项下之保证条款、保密条款、知识产权条款、法律适用及争议解决条款等内容，不因本协议的终止而失效。

链路追踪服务专用条款

1. 到期及欠费

1.1.如包年、包月和包周等固定服务期限到期的：

1.1.1.就包年、包月和包周的链路追踪服务，服务期到期后，如您继续使用阿里云服务的，您应及时续费；

1.1.2. 您服务到期后，如未续费，阿里云将在到期时暂停您就该服务的操作权限，但仍保留实例、继续存储您的数据7日（自操作权限被暂停之日的暂停开始时刻至第7日相同时刻为期限届满）；如7日届满仍未续费，阿里云将释放您的实例，并删除实例数据。

1.2.就按量付费的链路追踪服务，您应及时充值、缴纳服务费用以保证服务的持续使用，如您发生欠费：

1.2.1. 自账户首次欠费之时起，阿里云将暂停您对账户下所有按量付费服务的操作权限，但仍保留实例、继续存储您的数据7日（自操作权限被暂停之日的暂停开始时刻至第7日相同时刻为期限届满）；

1.2.2. 如自您首次欠费发生之时起7日届满，您仍未成功充值并足以支付所欠服务费用的，阿里云将释放您的欠费实例，并删除数据。

附件

服务使用规则

1.您承诺，如果您利用阿里云提供的服务进行经营或非经营的活动需要获得国家有关部门的许可或批准的，应获得该有关的许可或批准。包括但不限于以下内容：

1.1.如果您在云服务器服务上开办了多个网站，须保证所开办的全部网站均获得国家有关部门的许可或批准；

1.2.如您网站提供非经营性互联网信息服务的，必须办理非经营性网站备案，并保证所提交的所有备案信息真实有效，在备案信息发生变化时及时在备案系统中提交更新信息；

1.3.如您网站提供经营性互联网信息服务的，还应自行在当地通信管理部门取得经营性网站许可证；

1.4.如您提供BBS等电子公告服务的，也需根据相关法规政策要求备案或获得相应批准；

1.5.如您经营互联网游戏网站的，您应依法获得网络文化经营许可证；

1.6.如您经营互联网视频网站的，您应依法获得信息网络传播视听节目许可证；

1.7.若您从事新闻、出版、教育、医疗保健、药品和医疗器械等互联网信息服务，依照法律、行政法规以及国家有关规定须经有关主管部门审核同意，在申请经营许可或者履行备案手续前，应当依法经有关主管部门审核同意。

您理解并认可，以上列举并不能穷尽您进行经营或非经营活动需要获得国家有关部门的许可或批准的全部类型，您应获得有关的许可或批准，并应符合国家及地方不时颁布相关法律法规之要求。

2.除阿里云明示许可外，您不应修改、翻译、改编、出租、转许可、在信息网络上传播或转让阿里云提供的服务或软件，也不得逆向工程、反编译或试图以其他方式发现阿里云提供的服务或软件的源代码。

3.您不应不散布电子邮件广告、垃圾邮件（SPAM）：不利用阿里云提供的服务散发大量不受欢迎的或者未经请求的电子邮件、电子广告或包含反动、色情等有害信息的电子邮件。

4.您不应利用阿里云提供的资源和服务上传（Upload）、下载（Download）、储存、发布如下信息或者内容，不为他人发布该等信息提供任何便利（包括但不限于设置URL、BANNER链接等）：

4.1.违反国家规定的政治宣传和/或新闻信息；

4.2.涉及国家秘密和/或安全的信息；

4.3.封建迷信和/或淫秽、色情、下流的信息或教唆犯罪的信息；

4.4.博彩有奖、赌博游戏、“私服”、“外挂”等非法互联网出版活动；

4.5.违反国家民族和宗教政策的信息；

4.6.妨碍互联网运行安全的信息；

4.7.侵害他人合法权益的信息和/或其他有损于社会秩序、社会治安、公共道德的信息或内容；

4.8.其他违反法律法规、部门规章或国家政策的内容。

5.您不应大量占用，亦不得导致如程序或进程等大量占用阿里云云计算资源（如云服务器、网络带宽、存储空间等）所组成的平台（以下简称“云平台”）中服务器内存、CPU或者网络带宽资源（比如但不限于互联网挖矿等行为），并给阿里云云平台或者阿里云的其他用户的网络、服务器（包括但不限于本地及外地和国际的网络、服务器等）、产品/应用等带来严重的、不合理的负荷，影响阿里云与国际互联网或者阿里云与特定网络、服务器及阿里云内部正常通畅的联系，或者导致阿里云云平台产品与服务或者阿里云的其他用户的服务器宕机、死机或者用户基于云平台的产品/应用不可访问等。

6.您不应进行任何破坏或试图破坏网络安全的行为（包括但不限于钓鱼、黑客、网络诈骗、网站或空间中含有或涉嫌散播：病毒、木马、恶意代码，及通过虚拟服务器对其他网站、服务器进行涉嫌攻击行为如扫描、嗅探、ARP欺骗、DOS等）。

7.您不应进行任何改变或试图改变阿里云提供的系统配置或破坏系统安全的行为。

8.除非您购买了专门用于此目的的服务，您不利用本服务从事DDoS防护、DNS防护等防护售卖业务。

9.不利用阿里云的服务提供任何不经网络审查或依靠技术手段成为境内获取境外非法信息的途径。

10.您不应在阿里云服务或平台之上安装、使用盗版软件；您对自己行为（如自行安装的软件和进行的操作）承担责任。

12.2. 链路追踪Tracing Analysis服务等级协议

本产品最新版服务等级协议，请在阿里云服务等级协议汇总页查找获取。